Testauslähtöinen ohjelmistokehitys

Jussi Makkonen

15.5.2008

Joensuun yliopisto
Tietojenkäsittelytiede
Pro gradu -tutkielma

Tiivistelmä

Ohjelmistosuunnittelijat kehittävät tietojärjestelmiä ja ohjelmistoja erilaisten mallien sekä prosessien mukaisesti. Tässä tutkielmassa esitellään testauslähtöistä ohjelmistokehitystä, jonka mukaan testit tehdään aina ennen varsinaista ohjelmakoodin kirjoittamista, toisin kuin perinteiseksi ajatellussa ohjelmistotuotannossa. Tavoitteena tässä järjestyksessä tehtävällä ohjelmistokehityksellä on tuottaa puhdasta ohjelmakoodia, joka toimii, ja täten laadultaan parempia sekä luotettavampia ohjelmistoja. Keskeisiä asioita testauslähtöisessä ohjelmistokehityksessä ovat testien automatisointi sekä siinä apuna olevat erilaiset työkalut, jotka mahdollistavat testien automaattisen suorittamisen.

ACM-luokat (ACM Computing Classification System, 1998 version): D.2.2, D.2.3, D.2.5

Avainsanat: Testauslähtöinen ohjelmistokehitys, refaktorointi, testien automatisointi

Sisällysluettelo

1 Johdanto....1
2 Testauslähtöisen ohjelmistokehityksen vaiheet....6
2.1 Testien kirjoittaminen....7
2.2 Ohjelmakoodin kirjoittaminen....9
2.3 Ohjelmakoodin refaktorointi....10
2.4 Esimerkki....12
3 ATDD....15
3.1 ATDD-sykli....16
3.2 Yhteistyö....18
3.3 Käyttötapaustarinat....20
4 Testien automatisoiminen....23
4.1 Yksikkötestit....24
4.2 Komponenttitestit....25
4.3 Järjestelmätestit....25
4.4 Toiminnalliset testit....26
4.5 Yksinkertaisin testien automatisointistrategia....27
4.5.1 Kehitysprosessi....27
4.5.2 Asiakastestit....28
4.5.3 Yksikkötestit....29
4.5.4 Testattavuuden suunnittelu....31
4.5.5 Testien organisointi....31
5 TDD -malleja....33
5.1 Punaisen palkin malleja....33
5.1.1 Käynnistystesti....33
5.1.2 Askel kerrallaan....34
5.1.3 Perustelutesti....34
5.1.4 Oppimistesti....34
5.1.5 Lisätesti....35
5.1.6 Regressiotesti....35
5.1.7 Tauko....35
5.1.8 Remontointi....35
5.1.9 Halpa pöytä, hyvä tuoli....36

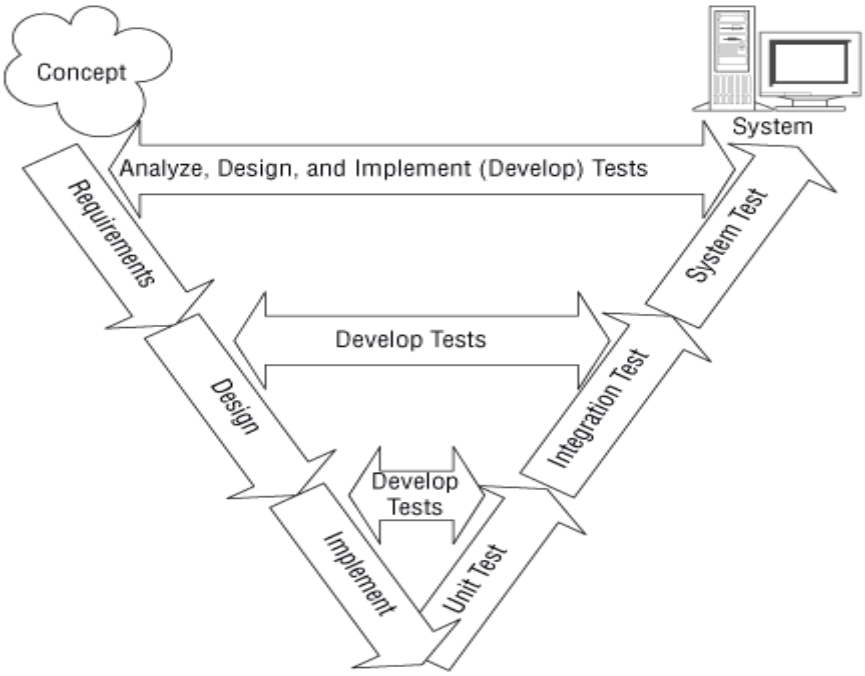
5.2 Testausmalleja....36
5.2.1 Lapsitesti....36
5.2.2 Mallinnettu olio....37
5.2.3 Itseensä siirto....37
5.2.4 Lokiviestit....37
5.2.5 Törmäysnukke....37
5.2.6 Rikkinäinen testi....38
5.2.7 Puhdas palautus....38
5.3 Vihreän palkin malleja....38
5.3.1 Lavastus....38
5.3.2 Kolmiomittaus....39
5.3.3 Itsestään selvä toteutus....39
5.3.4 Yhdestä moneen....39
6 TDD-työkaluja....41
6.1 .NET....42
6.1.1 NUnit....42
6.1.2 AnyUnit....43
6.2 C & C++....44
6.2.1 CppUnit....44
6.2.2 EasyUnit....45
6.3 Tietokannat....46
6.3.1 UtPLSQL....46
6.3.2 SQLUnit....47
6.4 GUI....48
6.4.1 Selenium....49
6.4.2 Abbot....49
6.5 Java....50
6.5.1 Unitils....51
6.5.2 jMock....52
6.5.3 TestNG....53
6.5.4 JUnit....56
7 Yhteenveto....58
Viitteet....60

1 Johdanto

Ohjelmistotuotannossa on yleensä käytössä jokin malli, jonka mukaan prosessi etenee. Ohjelmistoprosessimalli kattaa toteutettavan ohjelmiston elinkaaren sen olemassa olevasta ideasta luovutetun järjestelmän ylläpitoon ja siitä luopumiseen asti. Ohjelmistoprosessimalli ei tietenkään takaa täydellistä lopputulosta toteutettavan ohjelmiston kannalta. Se antaa kylläkin hyvän ohjenuoran suoritettavien askeleiden järjestykselle. Ei ole olemassa myöskään yhtä ainoaa oikeaa ohjelmistoprosessimallia. Tilanteesta ja projektista riippuen, oikeanlaisen mallin valinta voi auttaa pääsemään haluttuun lopputulokseen. Testauslähtöinen ohjelmistokehitys eli TDD (Test Driven Development) on eräs lähestymistyyli ohjelmistokehitykseen. Se tarjoaa XP:n (Extreme Programming) esittelemän tavan (Beck, 2000) kirjoittaa ensin testit ja vasta sen jälkeen varsinainen ohjelmakoodi. XP:n yhteydessä puhutaan käsitteestä TFD (Test First Design), joka on testauslähtöisessä ohjelmistokehityksessä muuttunut muotoon TDD huolellisen refaktoroinnin tullessa osaksi prosessia.

Black (2007) esittelee muutamia malleja, joita käytetään hyväksi ohjelmistotuotannossa. Vesiputousmalli on hyvin perinteinen ja ehkä vanhin tunnetuista ohjelmistoprosessimalleista. Siinä vaiheet seuraavat toisiaan hyvin järjestelmällisesti ja vain yhteen suuntaan. Juuri tällainen vesiputouksen tavalla etenevä prosessi tekeekin vesiputousmallista sellaisenaan hyvin jäykän ja käyttökelvottoman useimpiin projekteihin, joissa tarvittaisiin tietynlaista joustavuutta.

Vesiputousmallista hieman hienostuneempi ja hiotumpi malli on nimeltään V-malli (kuva 1). Siinä Testaaminen otetaan huomioon jo hyvin aikaisessa vaiheessa projektia. Jokainen ohjelmistoprosessin vaihe pitää sisällään testien suunnittelua. Esimerkiksi vaatimusten määrittelyn yhteydessä saadaan järjestelmän kannalta tärkeimpiä toimintoja esille ja niihin perustuen voidaan kirjoittaa alustavaa hyväksymistestaus-dokumentaatiota. V-malli on kuitenkin hyvin aikataulu- ja budjettilähtöinen prosessimalli. Yleensä resurssien, enimmäkseen rahojen, käydessä vähiin testaaminen on ensimmäinen osa, josta ollaan valmiita luopumaan, jotta saadaan ohjelmisto aikataulussaan julkaistua. Onnistuneen V-mallin noudattamisessa painotetaankin ajoissa aloitettua ja hyvin toteutettua suunnittelua ja analysointia.



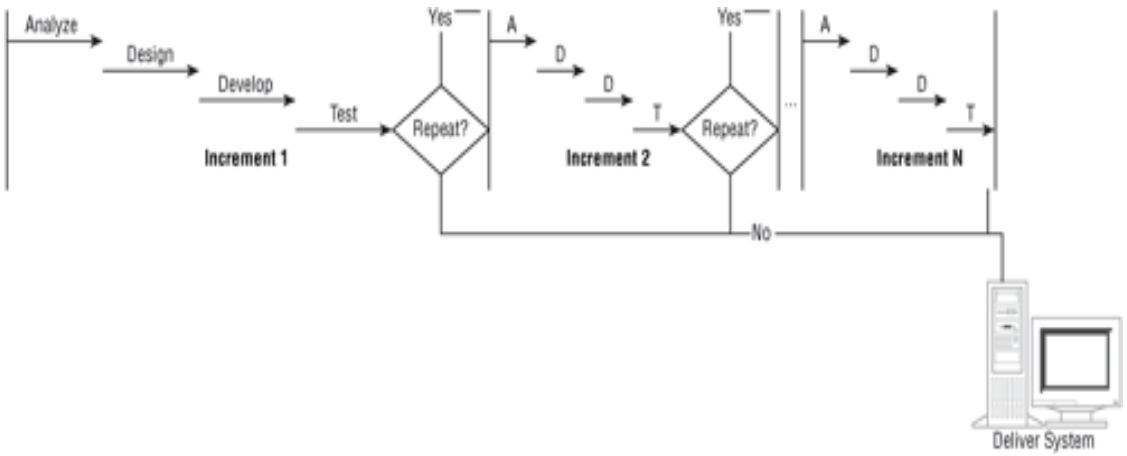
Kuva 1: V-malli (Black, 2007).

Vesiputousmallilla ja testauslähtöisellä ohjelmistokehityksellä ei ole juurikaan mitään yhteistä. Vesiputousmalli keskittyy noudattamaan orjallisesti prosessimallin mukaan etenevää tapahtumaketjua, jossa testaaminen suoritetaan lopuksi eikä takaisin edelliseen vaiheeseen voida palata menemättä aivan alkuun, kun taas TDD paneutuu testaamiseen ennen kuin ensimmäistäkään ohjelmakoodiriviä on kirjoitettu.

V-mallilla ja testauslähtöisellä ohjelmistokehityksellä on jo huomattavasti enemmän yhtäläisyyksiä, vaikka ne ovatkin vielä kaukana toisistaan. Black (2007) esittelee kirjassaan V-mallin mukaisen ohjelmistoprojektin Gant-kaavion hyvin suoritetulle projektille. Suurin ero tämän mallin ja testauslähtöisen kehityksen välillä on testien suorittamisen aikataulutus. V-mallissa varsinaiset testit suoritetaan lopuksi, kun taas TDD painottaa testien suorittamista mahdollisimman aikaisessa vaiheessa. Yhtäläistä näissä malleissa on kuitenkin se, että ne molemmat mahdollistavat huomattavasti joustavamman ohjelmiston kehityksen kuin vesiputousmalli.

Inkrementaalinen ohjelmistoprosessimalli (kuva 2) on uudemman sukupolven malli, joka eroaa hieman aiemmin mainituista vesiputousmallista ja V-mallista. Siinä kehitys tapahtuu tarkoin määritellyissä analysointi-, suunnittelu-, toteutus- ja testausvaiheissa. Aluksi yleensä ensimmäisessä versiossa on toteutettu kriittisimpiä ja asiakkaan kannalta

toivotuimpia toiminnallisuuksia. Kehitysprosessin edetessä näitä toiminnallisuuksia lisätään aiempaan versioon ja tällä tavalla loppujen lopuksi toteutetaan ohjelmistojärjestelmä, joka vastaa toiminnallisuuksiltaan määrityksiä.

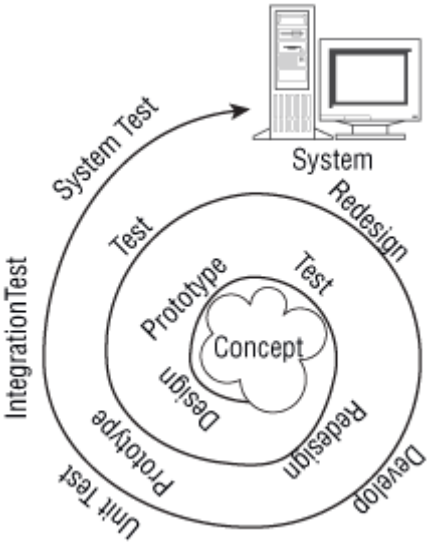


Kuva 2: Inkrementaalinen malli (Black, 2007).

Testien ajoittamisen ja aloittamisen ajankohta ovat suurimmat erot inkrementaalisen mallin ja testauslähtöisen ohjelmistokehityksen välillä. Inkrementaalisessa mallissa testit suoritetaan tarkoin määriteltyjen analyysi-, suunnittelu- toteutusvaiheiden jälkeen. Testausta ei suoriteta samaan aikaan toteutuksen kanssa ja mikä vieläkin tärkeämpää, testejä ei kirjoiteta ennen kuin varsinainen ohjelmisto on saatu siihen tilaan, mikä kyseisessä inkrementaalivaiheessa on ollut tarkoitus. Tietynlaista automaattista taantumatestausta voidaan harrastaa myös inkrementaalisessa ohjelmistokehityksessä, mutta se vaatii käyttöönotetun testausympäristö, joka sallii testien automaattisen suorittamisen haluttuna ajankohtana. Aiemmin kirjoitettuja testejä pitää myös muistaa suorittaa jokaisen inkrementaalivaiheen jälkeen, kun ollaan tuotu kehitettävään järjestelmään jotain uutta toiminnallisuutta.

Spiraalimalli (kuva 3) on Boehmin (1988) esittelemä ohjelmistoprosessimalli, jonka tarkoituksena oli muuttaa kehittäjien suuntausta pois dokumentti- ja koodilähtöisistä malleista, kohti enemmän riskilähtöistä prosessimallia. Spiraalimalli yhdistelee vesiputousmallista saamiaan lähtökohtia prototyyppimalliseen kehitykseen. Se on käytännöllinen silloin, kun projektitiimi ei ole aivan varma millainen kehitettävä järjestelmä tulee olemaan, mutta tiedossa on järjestelmälle asetetut vaatimukset vaatimusmäärittelyn pohjalta. Spiraalimallia on tarkoitettu käytettävän projekteihin, jotka ovat suuria, kalliita sekä vaatimuksiltaan monimutkaisia. Resurssien riittävyys on

elintärkeää, kun lähdetään toteuttamaan projektia spiraalimallin pohjalta. Spiraalimalli lähtee liikkeelle määrittelyjen kautta tehtävästä suunnittelusta. Ei välttämättä mene kovinkaan kauaa, kun ensimmäinen prototyyppi on jo valmis testattavaksi. Prototyyppiä kehitetään ja muutetaan tarpeen mukaan sykleissä, jotka periaatteessa toistavat samaa kaavaa. Kun kaikki halutut ominaisuudet on saatu lisättyä kehitettävään järjestelmään, voidaan sille suorittaa lopuksi perinteisemmät testit kuten yksikkötestit, integraatiotestit ja järjestelmätestit. Liian aikaisessa vaiheessa prototyyppiä on turha luovuttaa varsinaisten loppukäyttäjien ulottuville. Tämä voi johtaa odottamattomiin väärinkäsityksiin.



Kuva 3: Spiraalimalli (Black, 2007).

Jos spiraalimallia ja testauslähtöistä kehitystä lähdetään vertailemaan todella tarkalla tasolla, niin eroja löytyy paljonkin. Esimerkiksi koko prosessille tarvittavat resurssit ovat yksi suurimmista eroista. Kuitenkin löyhemmin tarkasteltaessa näistä kahdesta mallista voidaan huomata yhtäläisyyksiä. Sekä spiraalimallissa että testauslähtöisessä ohjelmistokehityksessä tehdään huomattavia määriä varsinaisen ohjelmistokoodin uudelleen kirjoitusta ja parantamista. TDD:n ollessa kyseessä puhutaan refaktoroinnista ja spiraalimallin tapauksessa uudelleensuunnittelusta. Molemmat näistä voivat tapahtua useaan otteeseen hyvinkin lyhyen ajan sisällä. Myös jos tarkastellaan spiraalimallia suurempana kokonaisuutena, voidaan huomata prosessin sisällä jatkuva testauksen tarve ja sen toteutus. Lähempää tarkasteltaessa varsinainen ohjelmakoodin kirjoitus ei välttämättä tapahdu testauslähtöisen ohjelmistokehityksen periaatteiden mukaan mutta on jo kuitenkin huomattavasti lähempänä TDD:n tyyliä kuin esimerkiksi

vesiputousmalli.

Tässä tutkielmassa keskitytään testauslähtöiseen ohjelmistokehitykseen. Luvussa 2 käydään läpi tarkemmin TDD:n vaiheet ja niihin liittyvät toimenpiteet. Luku 3 keskittyy hyväksymistestauslähtöiseen ohjelmistokehitykseen, jota voidaan käyttää TDD:n apuna toteutettaessa ohjelmistoja asiakkaille. Luvussa 4 on kerrottu testien automatisoimisesta prosessin eri vaiheissa sekä esitellään yksinkertaisin testien automatisointistrategia. Luvussa 5 esitellään testauslähtöisessä ohjelmistokehityksessä hyväksi käytettäviä malleja ja kerrotaan missä tilanteissa niitä voidaan soveltaa. Luvussa 6 on esitelty erilaisia työkaluja, joita voidaan käyttää apuna kehitettäessä ohjelmistoja testauslähtöisesti. Työkalujen esittelyn yhteydessä annetaan myös lyhyitä esimerkkejä TDD:n soveltamiseksi. Luku 7 on yhteenveto tutkielmassa käsitellyistä asioista.

2 Testauslähtöisen ohjelmistokehityksen vaiheet

Testauslähtöiseen ohjelmistokehitykseen kuuluu kiinteästi kolme eri vaihetta, jotka seuraavat toisiaan ennalta määrätyssä järjestyksessä ja toistuvat kehityksen yhteydessä nopeissa sykleissä. Nämä vaiheet ovat testien kirjoittaminen, varsinaisen ohjelmakoodin kirjoittaminen sekä ohjelmakoodin refaktorointi. Kuvassa 4 on esitetty testauslähtöisen ohjelmistokehityksen eri vaiheet ja niiden järjestys kehityssyklissä.

Kuva 4: TDD:n vaiheet.

Beck (2003) on esittänyt vaiheille oman nimeämiskäytäntönsä, jossa vaiheet ovat järjestyksessä nimetty punainen, vihreä ja refaktorointi. Nimet punainen ja vihreä tulevat yksikkötestaustyökalun, kuten JUnit, graafisesti ilmoittamasta palautteesta, kun yritetään kääntää tai suorittaa testiä sekä siihen liittyvää ohjelmakoodia. Vaiheet ovat kestoltaan hyvin lyhyitä, yleensä noin muutaman minuutin pituisia. Lyhyeen kestoon liittyen ohjelmoijat kirjoittavat itse testit. Muutamassa minuutissa ei ole aikaa siihen, että ulkopuolelta saadaan valmiiksi kirjoitettuja testejä.

Wake (2001) on antanut myös vaiheille omat nimet, jotka tulevat liikennevalojen mukaan keltainen, punainen ja vihreä. Keltaisessa vaiheessa kirjoitetaan testi ja yritetään kääntää se. Testi ei kuitenkaan käänny, koska ei ole olemassa sitä vastaavaa varsinaista ohjelmakoodia. Punaisessa vaiheessa kirjoitetaan varsinainen ohjelmakoodi ja käännetään se. Koodi kääntyy mutta testit eivät mene läpi. Vihreässä vaiheessa kirjoitetaan koodi läpäisemään testit. Järjestyksen tulee olla keltainen, punainen ja vihreä, koska muussa tapauksessa on tehty jokin virheellinen siirtymä. Wake (2001) on listannut muutamia ongelmatilanteita, joissa on tapahtunut virheellinen siirtyminen. Vihreästä vihreään siirtymisessä on tapahtunut virhe siinä tapauksessa, jos järjestelmään on lisätty jotain uutta toiminnallisuutta testien muodossa. Tämä siirtymä ei kuitenkaan

ole virheellinen siinä tapauksessa, jos refaktoroidaan vanhaa koodia ja halutaan pitää testit muuttumattomina siihen nähden. Vihreästä punaiseen siirtymisessä virhe on tapahtunut, jos yritetään lisätä varsinaista ohjelmakoodia ilman, että ollaan ensin kirjoitettu sitä vastaava testitapaus. Testauslähtöisen ohjelmistokehityksen perusolettamus on, että aina kirjoitetaan testitapaus ennen varsinaista ohjelmakoodia. Keltaisesta vihreään siirtymisessä virhe on tapahtunut, kun ollaan jätetty punainen vaihe välistä pois. Testien pitäisi aina ensin epäonnistua, jotta nähdään ohjelmakoodin refaktoroinnin jälkeen kehitystä. Punaisesta punaiseen siirtyminen tarkoittaa, että implementaatiossa on jotain vialla. Yleensä tämä on merkki siitä, että kehityksen aikana tulisi edetä pienempien askelien kautta. Keltaisesta keltaiseen siirtyminen sekä punaisesta punaiseen siirtyminen ovat merkki syntaksivirheistä. Yleensä kääntäjä kertoo missä virhe on tapahtunut, jotta se voidaan korjata ja päästään eteenpäin kehitysprosessissa.

2.1 Testien kirjoittaminen

Testauslähtöisessä ohjelmistokehityksessä lähdetään aina liikkeelle testien kirjoittamisella. Tarkoitus on saada järjestelmälle sovitut ja määritellyt toiminnallisuudet testien muotoon. Yleensä näitä toiminnallisuuksia voidaan lähteä etsimään vaatimusmäärittelystä sekä toiminnallisesta määrittelystä. Nämä kertovat mitä järjestelmän pitää tehdä sekä miten toimintojen tulee olla toteutettu, jotta kehitettävä järjestelmä on vaatimustensa mukainen.

Kirjallisuudessa (Beck, 2003) on esitetty muutamia reunaehtoja testaamisen aloittamiselle. Ainakin seuraaville strategisille kysymyksille tulisi saada vastaus ennen kuin voidaan aloittaa yksityiskohtaisempaa testauksen suunnittelua:

- Mitä testaamisella tarkoitetaan?
- Milloin testaaminen tapahtuu?
- Kuinka valitaan järjestelmän testattava logiikka?
- Kuinka valitaan testattavat tietokokonaisuudet?

Testauslähtöinen ohjelmistokehitys antaa ensimmäiseen kysymykseen suoran

vastauksen. Testaamisella tarkoitetaan järjestelmälle asetettujen vaatimusten ja toimintojen oikeellisuuden varmentamista sekä kehitysprosessin yhteydessä tapahtuvaa järjestelmän suunnittelua ja toteutusta. Myös toiseen kysymykseen testauslähtöinen ohjelmistokehitys antaa suoran vastauksen. Testaamista tapahtuu koko järjestelmän kehityksen ajan. Kolmanteen ja neljänteen kysymykseen sisältyy valintaa testattavista kokonaisuuksista. Testattavan logiikan suhteen valinta on helppo, koska testauslähtöisen ohjelmistokehityksen periaatteiden mukaan yhtään riviä varsinaista ohjelmakoodia ei saa lisätä järjestelmään, jos ei ole olemassa sitä vastaavaa testiä. Jos halutaan noudattaa TDD:n mukaisia oppeja, niin kaikki järjestelmän sisältämä toimintalogiikka tulee olla testattua. Testattavien tietokokonaisuuksien kanssa valintaan voi hyvinkin vaikuttaa ohjelmoijien erilainen kokemus sekä vaihtelevat taitotasot. Kokeneempi ohjelmistosuunnittelija pystyy hahmottamaan nopeammin mahdolliset ongelmakohdat ohjelmistokehityksessä, kun taas kokematon ohjelmistosuunnittelija saattaa keskittyä liikaa epäolennaisten tietokokonaisuuksien testaamiseen. Tavoitteena olisi, että järjestelmän toiminnan kannalta olennaisten tietokokonaisuuksien testitapaukset olisivat mahdollisimman kattavat. Erilaisten raja-arvojen testaaminen on tällaisissa tapauksissa hyödyllistä.

Testauslähtöisen ohjelmistokehityksessä eteneminen tapahtuu pienissä askelissa ja testien kirjoittaminen ei poikkea tästä periaatteesta millään tavalla. Testejä tulisi kirjoittaa vain yksi testattava ominaisuus kerrallaan. Tällä tavalla voidaan saavuttaa nopea tahti ohjelmistokehityksen aikana, koska vaiheen tulisi kestää vain muutamia minuutteja. Helpoin tapa aloittaa testien kirjoittaminen on valita sellainen ominaisuus testitapauslistalta, joka varmasti pystytään nopeasti kirjoittamaan testaustyökalulla toteutettavan testin muotoon.

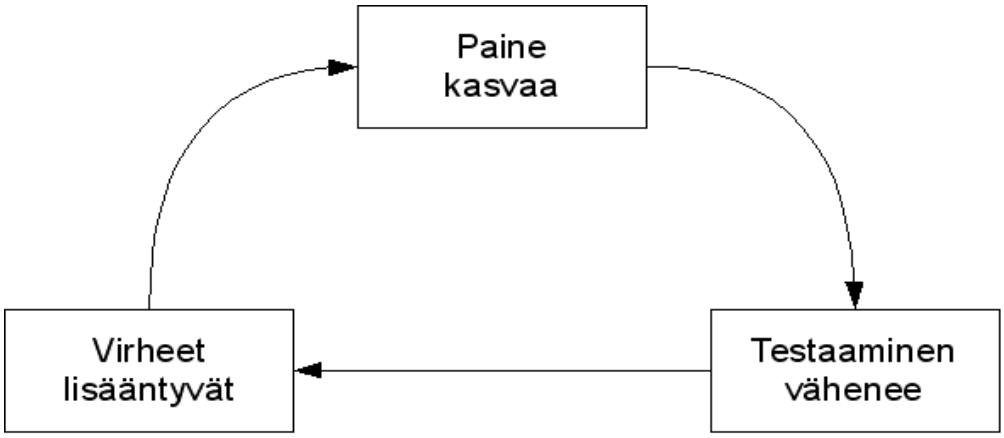
Testien kirjoittaminen voidaan myös nähdä toimivan viime kädessä pohjana järjestelmän arkkitehtuurisuunnittelulle, koska tässä vaiheessa viimeistään päätetään millä tavalla nimetään järjestelmän sisältämät luokat sekä metodit, jotka tullaan toteuttamaan seuraavassa kehitysvaiheessa. Testien kirjoittamisen yhteydessä suunnitellaan myös metodien tarvitsemat parametrit sekä niiden mahdolliset näkyvyysrajoitukset. Ohjelmistosuunnittelijoiden tulisi tämän takia suunnitella testit hyvin ennen varsinaista implementaatiota, koska tavoitteena on, että testejä ei tarvitsi

muuttaa niiden kirjoittamisen jälkeen. Ainoat muutokset tapahtuisivat varsinaiseen ohjelmakoodiin, jonka kirjoittaminen on testien kirjoittamista jälkeinen vaihe.

2.2 Ohjelmakoodin kirjoittaminen

Testauslähtöisen ohjelmistokehityksen toinen vaihe on varsinaisen ohjelmakoodin kirjoittaminen. Tarkoitus on kirjoittaa vain niin vähän ohjelmakoodia kuin on mahdollista, että saadaan testit sekä ohjelmakoodi käännettyä ja suoritettua läpi hyväksytysti. Tavoitteena on myös suorittaa kyseiset operaatiot mahdollisimman nopeasti, koska tämänkin vaiheen on tarkoitus kestää korkeintaan vain muutamia minuutteja.

Kun kirjoitetaan vain niin vähän varsinaista ohjelmakoodia kuin mahdollista, pyritään saamaan nopeasti palautetta tehdystä työstä. Ohjelmakoodia ennen kirjoitettu testi ei mene läpi ja se osoittaa kuilun sen välillä mitä ohjelmakoodi tekee ja mitä sen oletetaan tekevän (Koskela, 2008). Pienillä askelilla ja vähäisellä työllä tämä kuilu saadaan suljettua muutamassa minuutissa, joka puolestaan tarkoittaa, että varsinainen ohjelmakoodi ei ole rikkinäisessä tilassa kovinkaan pitkän aikaa. Pienillä askelilla ja edistymisen tunteella on myös psykologinen vaikutus ihmisiin. Yleisesti ottaen on paljon helpompi hahmottaa lyhyen tähtäimen tavoitteita ja käsitellä pienempiä kokonaisuuksia. Edistymisen tunteella on tarkoitus vähentää painetta sekä nostaa itsetuntoa työskentelyn yhteydessä. Kuva 5 havainnollistaa paineen, testaamisen sekä virheiden yhteyden kehitysprosessissa.



Kuva 5: Ajan puute kasvattaa painetta ja vähentää testaamista (Beck, 2003).

Kuvassa 5 kuvataan kuinka paineen kasvu johtaa testaamisen vähenemiseen. Testaaminen on taas suorassa yhteydessä virheiden lukumäärän kanssa. Virheet puolestaan kasvattavat painetta. Edistymisen ja onnistumisen tunne sekä paineen kasvun pienentäminen ovat yksi pienien askelien tarkoituksista testauslähtöisessä ohjelmistokehityksessä.

Eräs tärkeimmistä säännöistä testauslähtöisessä ohjelmistokehityksessä on, että varsinaista ohjelmakoodia ei milloinkaan kirjoiteta, ennen kuin on olemassa sitä vastaava testitapaus kirjoitettuna. Varsinaisen ohjelmakoodin kirjoittaminen ensin johtaisi perinteiseen näkemykseen ohjelmistokehityksestä, jossa suunnitellaan ensin, jonka jälkeen tapahtuu ohjelmakoodin kirjoitus ja vasta viimeisimpänä vaiheena ohjelmakoodin testaus. Ohjelmakoodin kirjoittamisen perimmäisenä tarkoituksena on vain ja ainoastaan saada testi suoriutumaan läpi (Koskela, 2008).

Koska testit on tarkoitus saada suoriutumaan mahdollisimman nopeasti läpi, silloin ei ole myöskään aikaa kiinnittää liikaa huomiota hyviin ohjelmointitapoihin, jos ne eivät tule luonnostaan kirjoittamisen yhteydessä. Mikään ei estä tässä vaiheessa käyttämästä varsinaisen ohjelmakoodin joukossa vakioita ja duplikaatteja, jos niiden avulla saadaan testit suoriutumaan hyväksytysti läpi nopeammin. Vakioiden ja duplikaattien poisto tapahtuu vasta testauslähtöisen ohjelmistokehityksen kolmannessa vaiheessa. Silloin näihin hyvien ohjelmointitapojen vastaisiin asioihin on helpompi puuttua, kun tiedetään suoraan mistä kohdasta koodia niitä täytyy ryhtyä etsimään.

2.3 Ohjelmakoodin refaktorointi

Testauslähtöisen ohjelmistokehityksen viimeisin vaihe on varsinaisen ohjelmakoodin refaktorointi. Refaktoroinnin yhteydessä otetaan muutama askel taaksepäin kehitysprosessissa ja tarkastellaan ohjelmakoodin suunnittelumalleja sekä yritetään parannella niitä. Koskelan (2008) esittämän ajatuksen mukaan juuri refaktorointivaihe tekee testauslähtöisestä ohjelmistokehityksestä vakaata ja kestävää. Ilman refaktorointivaihetta TDD olisi vain nopea tapa tuottaa viimeistelemätöntä ohjelmakoodia, jolle ei välttämättä myöhemmin löydy mitään uutta käyttötarkoitusta.

Refaktorointia ei pidä ajatella vain ja ainoastaan testauslähtöisen ohjelmistokehityksen tärkeänä osana. Sillä on myös tärkeä osa muissakin kehitysmalleissa, joissa halutaan muuttaa tai parannella vanhaa koodia.

Fowlerin (1999) esittämän ajatuksen mukaan refaktorointi on prosessi, jossa muutetaan ohjelmistoa tai järjestelmää tavalla, joka ei muuta refaktoroitavan järjestelmän ulkoista käyttäytymistä, vaan parantaa sen sisäisiä rakenteita. Testauslähtöiseen ohjelmistokehitykseen yhdistettynä refaktorointi voidaan nähdä myös varsinaisen ohjelmistokoodin suunnitteluvaiheena, kuten kuva 6 osoittaa

Kuva 6: Refaktorointi suunnitteluvaiheena.

Perinteisessä ohjelmistokehityksessä suunnitellaan ensin hyvin ja vasta sen jälkeen tulee varsinaisen ohjelmakoodin kirjoittaminen. Testauslähtöinen ohjelmistokehitys kääntää tämän ajatuksen täysin päälaelleen, joka voi osaltaan olla joillekin ohjelmistosuunnittelijoille vaikea asia omaksua. Refaktoroinnilla pyritään ohjelmiston jatkuvaan kehitykseen, koska kaikkea ei suunnitella alussa valmiiksi. Tarkoitus on oppia nopeasti rakentamaan ohjelmistoa pala palalta valmiimmaksi ja aina jokaisen pienen lisätyn toiminnon jälkeen parantamaan sen sisäistä suunnittelumallia.

Testauslähtöisessä ohjelmistokehityksessä jokaisen vaiheen tulisi kestää muutaman minuutin, joten refaktoroinninkin tulisi sujua mahdollisimman nopeasti. Refaktorointivaiheen kestoon vaikuttaa hyvin paljon edellisessä vaiheessa kirjoitetun varsinaisen ohjelmakoodin määrä. Pienet askeleet auttavat pitämään vaiheet mahdollisimman nopeina ja refaktoroinnin yhteydessä on helpompi löytää muutosta vaativat kohdat. Beckin (2003) esittämän yksinkertaistetun ajatuksen mukaan tässä vaiheessa tarvitsee vain poistaa duplikaatit sekä vakiot.

Refaktoroinnin tärkein asia testauslähtöisessä ohjelmistokehityksessä on nimenomaan varsinaisen ohjelmakoodin parantaminen muutamatta sen ulkoista käyttäytymistä. Ulkoisen käyttäytymisen pitäisi pysyä muuttumattomana, koska testit käyttävät hyväkseen aiemmassa vaiheessa kirjoitettua ohjelmakoodia ja niiden suoriutuminen läpi hyväksytysti on edistymistä prosessin syklissä. Taantumaa ei saisi tapahtua, koska tässä tapauksessa muutokset voivat kohdistua myös muihin aiemmin kirjoitettuihin testeihin. Refaktoroinnissa olisi hyvä aina muistaa eristää refaktoroitava ohjelmakoodin osa. Onnistuneen refaktorointivaiheen jälkeen on aika ottaa seuraava testitapaus työn alle ja aloittaa sykli alusta.

2.4 Esimerkki

Testauslähtöistä ohjelmistokehitystä voidaan havainnollistaa yksinkertaisen esimerkin avulla, jossa on tarkoituksena toteuttaa Javalla luokka, jonka tehtävänä on laskea kaksi parametreinaan saamaa kokonaislukua yhteen. Esimerkissä käytetään apuna JUnit-yksikkötestaustyökalua (versio 3.8), josta kerrotaan enemmän kohdassa 6.5.4.

Ensimmäiseksi on tarkoitus kirjoittaa testiluokka, joka tulee testaamaan varsinaisen ohjelmakoodin toiminnan. Esimerkissä testiluokalle annetaan nimeksi TestCalc, kuten kuva 7 havainnollistaa.

```
import junit.framework.*;
public class TestCalc extends TestCase {

}
```

Kuva 7: Testiluokan runko.

JUnit-testin suorittamiseen otetaan mukaan junit.framework-paketti. Tämän jälkeen luodaan testiluokka, joka laajentaa JUnitin TestCase-luokkaa. Testiluokkien nimeämisessä on hyvä käyttää yhtenäistä tyyliä, esimerkiksi TestTestattavaluokka.

Toteutettava luokka tarvitsee metodin, joka laskee parametreina saadut luvut yhteen. Seuraavaksi kirjoitetaan TestCalc-luokalle tällainen metodi, joka testaa lukujen

yhteenlaskun (kuva 8).

```
import junit.framework.*;
public class TestCalc extends TestCase {
	public void testAdd {
		int sum = 0;
		int num1 = 1;
		int num2 = 2;
		int total = 3;
		sum = Calc.add(num1, num2);
		assertEquals(sum, total);
	}
}
```

Kuva 8: Testiluokka ja testimetodi.

JUnitia käytettäessä luokkien metodit, jotka on tarkoitus suorittaa testeinä, aloitetaan sanalla test. Testiluokka voi kuitenkin sisältää halutun määrän apumetodeja, joita ei haluta suorittaa testeinä. Tällöin näitä apumetodeja ei tule aloittaa test -sanalla. Tämän jälkeen suoritetaan alustuksia sekä käytetään hyväksi toteutettavan luokan metodia, jota tullaan testaamaan, tässä tapauksessa Calc-luokan add-metodia. Viimeisimpänä suoritetaan assert-metodilla vertailu oletetun ja saadun tuloksen välillä.

Näiden toimenpiteiden jälkeen on valmiina testiluokka, joka ei suoriudu hyväksytysti läpi. Tarvitaan Calc-niminen luokka, joka toteuttaa add-nimisen metodin (kuva 9). Testauslähtöisen ohjelmistokehityksen mukaan ensin kirjoitetaan testit ja vasta sen jälkeen varsinainen ohjelmakoodi, jota on tarkoitus testata.

```
public class Calc {

	static public int add(int num1, int num2) {

		return 3;

	}

}
```

Kuva 9: Testattava luokka.

Testattava luokka on tässä vaiheessa toteutettu palauttamaan kokonaisluku 3, koska halutaan havainnollistaa toisessa testauslähtöisen ohjelmistokehityksen vaiheessa, eli varsinaisen ohjelmakoodin kirjoituksen yhteyedssä, vakion käyttöä osana prosessia. Nyt Java-luokat kääntyvät ja testit suoriutuvat hyväksytysti läpi, koska testAdd-metodi

vertailee kahta yhtä suurta lukua keskenään. Seuraavaksi on aika siirtyä refaktoroimaan juuri toteutettua Calc-luokan add-metodia.

Refaktoroinnin yhteydessä on tarkoitus parannella kirjoitetun Calc-luokan add-metodin rakennetta muuttamatta sen ulkoista käyttäytymistä. Rakennetta muutetaan korvaamalla vakio 3 add-metodin parametreina saatujen lukujen summalla (kuva 10).

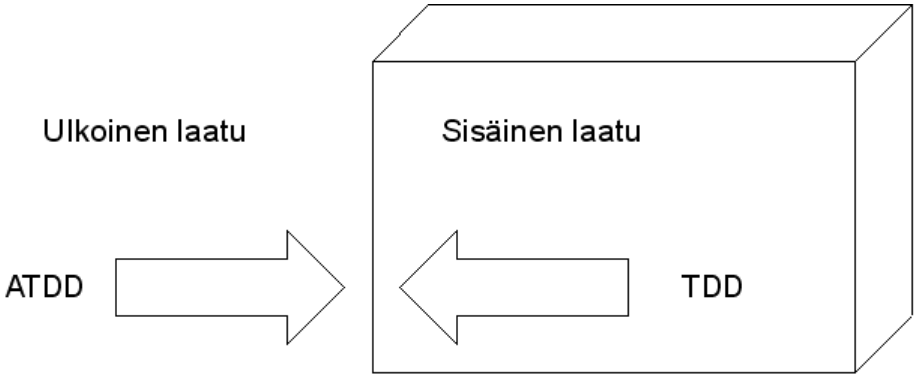
```
public class Calc {
	static public int add(int num1, int num2) {
		return num1 + num2;
	}
}
```

Kuva 10: Testattava luokka refaktoroinnin jälkeen.

Edellä suoritettujen askeleiden jälkeen ollaan toteutettu testiluokka, joka täyttää tarkoituksensa testaamalla toteutettavan ohjelmiston vastaavaa koodia. Ideaalitilanteessa varsinainen ohjelmakoodi ei sisällä yhtään riviä, joka sijaitsisi testien ulkopuolella. Pessimistinen perusolettamus on, että testaamaton ohjelmakoodi ei voi toimia.

3 ATDD

Hyväksymistestauksessa käytettävää testauslähtöistä ohjelmistokehitystä kutsutaan lyhenteellä ATDD (Acceptance Test Driven Development). Yksinkertaistettuna ajatuksena ATDD:n tarkoituksena on tuottaa hyvää ohjelmakoodia juuri oikeanlaiseen tarkoitukseen, koska pelkää TDD:tä käyttämällä on vaarana, että tuotetaan laadultaan hyvää ohjelmakoodia asiakkaille soveltumattomiin tarpeisiin. Koskelan (2008) mukaan ATDD voidaankin nähdä testauslähtöisen ohjelmistokehityksen isoveljenä. Siinä missä TDD:n tarkoitus on keskittyä lähinnä kehitettävän järjestelmän sisäiseen laatuun, ATDD keskittyy ulkoiseen laatuun, kuten kuva 11osoittaa.



Kuva 11: ATDD:n rooli (Koskela, 2008).

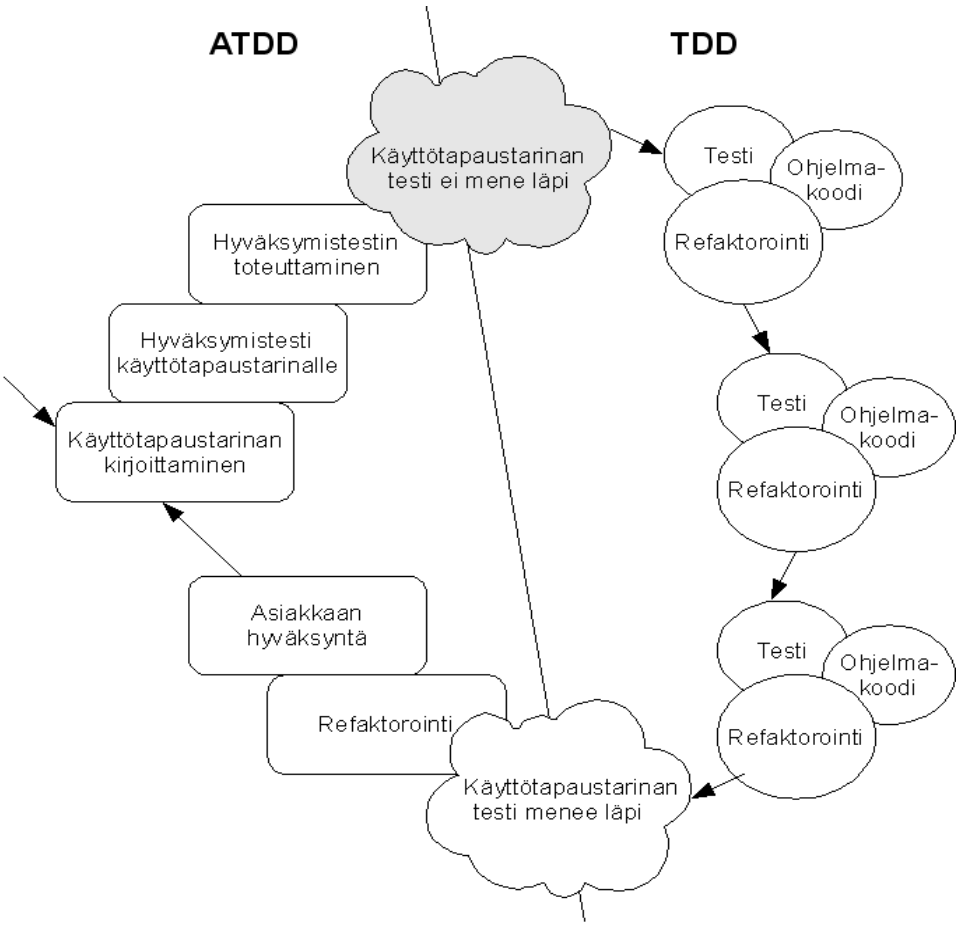
Ohjelmakooditasolla ATDD ei juurikaan eroa TDD:n syklistä, kuten kohdassa 3.1 kerrotaan tarkemmin. ATDD keskittyy vain enemmän kehitettävän järjestelmän ulkoisiin piirteisiin, jotka ovat näkyvissä asiakkaille, kun taas TDD on tarkoitettu sisäisten luokkien, objektien ja metodien piirteiden ja ominaisuuksien kehittämiseen. ATDD ja TDD ovat toisiaan tukevia menetelmiä ja ne kulkevat hyvin käsi kädessä, mutta mikään ei estä niitä olemasta toisistaan irrallisia osia ohjelmistokehityksessä.

ATDD:n yksi tärkeimmistä asioista on yhteistyön parantaminen kaikkien projektissa toimivien tahojen välillä. Tähän liittyy yhteisen kielen löytäminen ohjelmistokehittäjien, testaajien sekä asiakkaiden välillä. Halutut ja määritellyt ominaisuudet on huomattavasti helpompi toteuttaa, testata sekä hyväksyä, jos kaikki ovat puhuneet samaa kieltä projektin alusta lähtien. Yhteistyöstä puhutaan tarkemmin kohdassa 3.2.

Tekniikkana ATDD ei ole tiukasti sidottu mihinkään erityiseen vaatimustenilmaisumenettelyyn. Samat perusajatukset pätevät käyttötapauskuvausten (use cases), käyttötapaustarinoiden (user stories) sekä muiden vastaavien vaatimusten dokumentointimenettelyiden kanssa. Tässä tutkielmassa perehdytään tarkemmin käyttötapaustarinoihin kohdassa 3.3

3.1 ATDD-sykli

ATDD-sykliin on yksinkertaisimmillaan kuvattu neljä keskeistä vaihetta, jotka seuraavat toisiaan ennalta määrätyssä järjestyksessä (Koskela, 2008). Vaiheet ovat järjestyksessään käyttötapaustarinan valinta, testien kirjoittaminen, testien automatisoiminen sekä toiminnallisuuden toteuttaminen käyttötapaustarinalle. Kuva 12 havainnollistaa tarkemmin TDD:n käyttöä osana ATDD sykliä.



Kuva 12: ATDD:n ja TDD:n yhteys (Koskela, 2008).

Varsinaisen testauslähtöisen ohjelmistokehityksen osuus on siis tuottaa toimivaa

ohjelmakoodia osaksi toteutettavaa järjestelmää. ATDD:n osuudeksi jää toteutettavan järjestelmän näkyvien toiminnallisuuksien varmentaminen sekä hyväksyttäminen asiakkaalla.

Koskelan (2008) esittämän yksinkertaistetun ATDD-syklin ensimmäisessä vaiheessa valitaan käyttötapaustarina, jonka toiminnallisuudet halutaan varmentaa. Koska ATDD korostaa hyvin paljon yhteistyötä eri projektissa toimivien tahojen välillä, tämä ei välttämättä ole kovinkaan helppo tehtävä. Asiakkaalla saattaa olla hyvin erilainen näkemys siitä mikä on mahdollista toteuttaa missäkin vaiheessa ohjelmistoprojektia. Erilaisissa suunnittelu- sekä palautepalavereissa ohjelmistosuunnittelijoilla on mahdollisuus ottaa selvää tarkemmin asiakkaan tarpeista sekä yhdessä priorisoida käyttötapaustarinat tärkeysjärjestykseen. Tämän jälkeen seuraavaksi toteutettavan käyttötapaustarinan valinta helpottuu, kun kaikilla projektissa toimivilla tahoilla on samankaltaisempi näkemys asioista.

ATDD-syklin toisena vaiheena on testin kirjoittaminen käyttötapaustarinalle. Aivan kuten TDD:n tapauksessa, myös ATDD:ssa kirjoitetaan testitapaukset ennen varisnaista käyttötapaustarinan toteutusta. Valitun käyttötapaustarinan erilaisille mahdollisille skenaarioille kirjoitetaan ylös testejä yhteistyössä asiakkaan kanssa, koska juuri näiden testien varmentamisella on tarkoitus saada asiakas vakuuttuneeksi siitä, että toteutettava järjestelmä toimii oikein ja vaatimustensa mukaisesti (Koskela, 2008). Tässä vaiheessa ei ole kuitenkaan vielä tarvetta tarkentaa testejä lopulliseen muotoon, koska se saattaisi viedä liian paljon aikaa asiakkaalta ja kehitysryhmältä. Testit täydentyvät myöhemmin varsinaisen hyväksymistestien toteutusvaiheessa, joten yksinkertainen luettelo yleisimmistä tapauksista on tässä vaiheessa riittävä.

Testien automatisoiminen, jota voidaan myös kutsua testien toteuttamiseksi, on ATDD-syklin kolmas vaihe. Hyväksymistestien automatisoimisvaiheella on selkeä ero edelliseen vaiheeseen, koska nyt aiemmin ylös kirjoitetut testit implementoidaan automaattisesti suoritettavaan muotoon ja tuloksena saadaan tieto onnistuneista ja epäonnistuneista testeistä. Testien kirjoittaminen toteutettavaan muotoon ei kuitenkaan tarkoita, että olisi välttämätöntä kirjoittaa ne heti ylös jollain ohjelmointikielellä. On

mahdollista, että ensin kirjoitetaan toteutettavan testin mahdolliset toiminnot sekä syötteet ylös käyttötapaustarinan mukaisesti ja vasta myöhemmin toteutetaan tämä testi automatisoitavaan muotoon halutun työkalun avulla. Koskelan (2008) mukaan tällä tavalla on mahdollista välttää keskittymästä liikaa toteutuskielen ominaisuuksiin, koska tarkoituksena on kuitenkin saada testitapaukset toteutettua mahdollisimman kattavasti ja luotettavasti.

Viimeisin vaihe ATDD-syklissä on testattavan toiminnallisuuden toteuttaminen. ATDD ei anna sääntöä sille millä tavalla toteutuksen täytyy tapahtua, mutta niille ohjelmistokehittäjille, jotka ovat omaksuneet testauslähtöisen kehitystavan, TDD on luonnollinen tapa suorittaa toteutus. Kuva 12 havainnollisti ATDD:n ja TDD:n yhteistyön tarkemmin kuvatussa syklissä. Kun toteutetut hyväksymistestit on saatu suoritettua onnistuneesti läpi, on aika siirtyä syklin alkuun ja aloittaa uudelleen seuraavasta käyttötapaustarinasta. TDD:n tapauksessa vaiheiden keston tulisi voida laskea minuuteissa, mutta hyväksymistestien varmentaminen on huomattavasti enemmän aikaa vievä operaatio. ATDD-syklin kestolle on hankala määrittää kestoa, koska siinä vaaditaan yhteistyötä projektin eri tahojen välillä.

3.2 Yhteistyö

Läheinen yhteistyö eri tahojen välillä on olennaisen tärkeä osa jokaista monimuotoista pyrkimystä, johon liittyy kiinteästi ihmisiä ja ohjelmistokehitystä. Koskelan (2008) mukaan ATDD ei ole tässä asiassa poikkeus. Tarkoitus olisi saada kasaan tiivis ja hyvin integroitu projektiryhmä erillisten kehitys-, testaus- ja laadunvarmistusosastojen sijaan. Kuten myös TDD:n tapauksessa, ATDD:ssa on pyrkimys saada nopeasti palautetta muilta projektissa toimivilta tahoilta, jotta tilanteisiin pystytään reagoimaan nopeasti ja tehokkaasti. Tehokkaaseen yhteistyöhön pyritään myös jakamalla tietoja sekä taitoja projektiryhmän eri jäsenten välillä.

Asiakkaalla on hyvin tärkeä rooli ATDD:n mukaisessa ohjelmistokehityksessä. Asiakkaat toimivat osana projektiryhmää ja osallistuvat tiiviisti projektin aikana erilaisiin päätöksentekoihin sekä suunnittelupalavereihin. Kehitettävän ohjelmiston

vaatimusmäärittelyn valmistuttua asiakkaat osallistuvat määrittelyssä kuvattujen piirteiden mukaisten käyttötapaustarinoiden sekä niiden vaatimien testien suunnitteluun. He voivat vaatia minimitason, jonka järjestelmän on läpäistävä, jotta se voidaan todeta kelvolliseksi. Kehitysryhmä täydentää oman näkemyksensä mukaan asiakkaan kanssa suunniteltuja testejä. Kuten kuvasta 12 ilmeni, viime kädessä juuri asiakkaat varmentavat hyväksymistestien tulokset ja antavat näille oman hyväksyntänsä. Koskelan (2008) mukaan ATDD:n tyyppisessä inkrementaalisessa ohjelmistokehityksessä, jossa ohjelmisto kehittyy vaiheittain ja palaute on nopeaa, asiakkailla on enemmän vaikutusmahdollisuuksia ja valtaa kuin perinteisessä vesiputousmallissa, jossa suunnitelmat on lyöty lukkoon jo edeltä käsin. Projektiin sijoitettujen resurssien sekä teknisten rajoitteiden puitteissa asiakkailla on mahdollisuus päättää mitkä ominaisuudet järjestelmästä kehitetään ensin ja mitkä eivät ole välttämättömiä, jos aika ja raha loppuu kesken. Asiakkaat saavat siten päätäntävallan mihin heidän rahojaan käytetään ohjelmistoprojektissa.

Toinen tärkeä osa yhteistyössä on toimittajan puolelta projektiryhmään määritellyt henkilöt, jotka tekevät ohjelmiston teknisen suunnittelun sekä toteuttavat asiakkaalle tilatun järjestelmän. Heidän tehtävänään on myös antaa asiakkaille nopeaa palautetta projektin etenemisestä järjestelmän toimintojen jatkuvina lisäyksinä, jotta mahdolliset puutteet sekä virheet tulevat ilmi aikaisessa vaiheessa. Aikainen palaute vähentää osaltaan projektin riskejä sekä kustannuksia (Koskela, 2008). Toimittajan puolelta tuleva nopea palaute nostaa myös luottamusta projektiryhmän eri tahojen välillä, koska edistymistä pystytään seuraamaan konkreettisilla tuloksilla. Ohjelmistosuunnittelijat ovat myös vastuussa käyttötapaustarinoihin liittyvien testitapausten suunnittelusta yhdessä asiakkaan kanssa sekä niiden toteuttamisesta sellaiseen muotoon, josta voidaan yksiselitteisesti havaita ovatko testit suoriutuneet onnistuneesti läpi. Ohjelmisto-suunnittelijat voivat täydentää testitapauksia, jos ne tuntuvat riittämättömiltä todentamaan järjestelmän oikeanlaisen toiminnan. Hyväksymistestit suoritetaan toimittajan puolella hyväksytysti läpi ennen asiakkaan lopullista hyväksyntää.

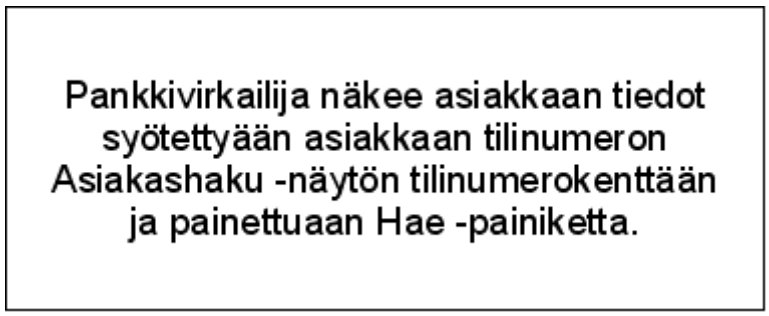
Toimivan yhteistyön kannalta on tärkeää, että kaikki osapuolet puhuvat samoista asioista kaikkien ymmärtämällä tavalla. Yhteisen kielen löytäminen voi olla vaikeaa projektiryhmässä, jossa jäseninä toimivat asiakkaat sekä ohjelmistosuunnittelijat.

Asiakkaat eivät välttämättä ymmärrä tietojärjestelmien teknisestä puolesta mitään ja ohjelmistosuunnittelijoille asiakkaiden toimiala voi olla ennestään täysin tuntematon. ATDD:n yhteydessä kehitettävät hyväksymistestit voivat toimia tässä tapauksessa projektiryhmän eri jäsenten välisenä yhteisenä kielenä (Koskela, 2008). Testitapauksia ja testejä voidaan käyttää vaatimusmäärittelyn rinnalla määrityksinä siitä mitä toteutettavan järjestelmän pitää tehdä ja millä tavalla sen tulee toimia. Testit ovat usein myös yksityiskohtaisempia kuin vaatimusmäärittelyn vastaavat kohdat, joten kaikkien projektissa työskentelevien tahojen on helpompi ymmärtää niitä samalla tavalla. Vaatimusmäärittelyssä yleensä abstrahoidaan järjestelmän erilaisia piirteitä, kun taas testeiksi muunnetut määritykset toimivat konkreettisina esimerkkeinä järjestelmän toiminnasta.

3.3 Käyttötapaustarinat

Koskelan (2008) mukaan käyttötapaustarinat ovat erittäin kevyt ja joustava tyyli ilmaista vaatimuksia kehitettäville ohjelmistoille. Yksinkertaisuudessaan käyttötapaustarinalla kerrotaan miten ohjelmiston tulisi toimia erilaisissa tapauksissa. Käyttötapaustarina antaa yleensä vastauksen kysymyksiin kuka tekee, mitä tekee ja miksi tekee. Käytännössä tarinoissa vastataan kysymyksiin kuka ja mitä, koska motiivi ilmenee yleensä tarinan kontekstista. Käyttötapaustarinat tukevat ATDD:n ajatusta tuottaa asiakkaille juuri oikealla tavalla toimiva ohjelmisto.

Käyttötapaustarinat pyritään pitämään yhden lauseen mittaisina, koska niiden ei ole tarkoitus korvata varsinaisia vaatimusmäärittelyssä kuvattuja vaatimuksia. Jotta käyttötapaustarinoista ei tulisi muodoltaan liian teknisiä, niissä voidaan käyttää projektiryhmässä yhteisesti ymmärrettävää terminologiaa. Tällä tavalla vältetään sekaannuksia tarinoiden merkityksestä sekä pyritään antamaan niiden kertomasta asiasta yksiselitteinen kuva. Tarinakortti (story card) on yksi tapa kirjoittaa käyttötapaustarina ylös. Riippuen korttien jakelutavasta sekä projektiryhmän sopimista käytännöistä kortit voivat olla esimerkiksi sähköisessä muodossa tai aivan tavallisia paperikortteja. Kuvassa 13 on esimerkki tarinakortista, jossa ilmaistaan käyttötapaustarinan muodossa kehitettävän ohjelmiston vaatimus.



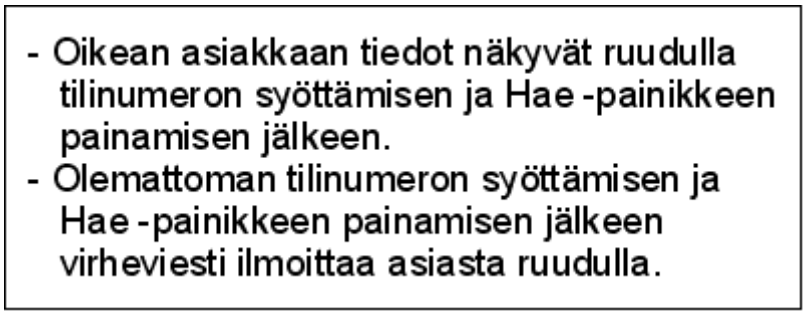
Kuva 13: Esimerkki tarinakortista.

Kuvan 13 mukainen kortti vastaa kysymyksiin kuka (pankkivirkailija) sekä mitä (syöttää asiakkaan tilinumeron, näkee asiakkaan tiedot). Itsestään selvissä tapauksissa käyttäjän motiivin selittäminen on tarpeetonta.

Jotta kuvan 13 tarina voidaan varmentaa, täytyy sille kirjoittaa testejä. Koskela on listannut hyväkymistesteille ominaisia asioita, jotka pätevät myös käyttötapaustarinoiden ja ATDD:n yhteydessä:

- Asiakas omistaa testit.
- Testit kirjoitetaan yhdessä asiakkaan, kehittäjien sekä testaajien kanssa.
- Testit keskittyvät vastaamaan kysymykseen mitä, eivät miten.
- Testit ilmaistaan kohdealueen terminologialla.
- Testit ovat lyhyitä, tarkkoja sekä yksiselitteisiä.

Käyttötapaustarinalle toteutettavat testit voidaan kirjoittaa tarinakortin kääntöpuolelle, jotta ne voidaan aina yhdistää oikeaan käyttötapaustarinaan. Kuva 14 havainnollistaa kuvan 13 tarinakortille kirjoitettuja testejä.



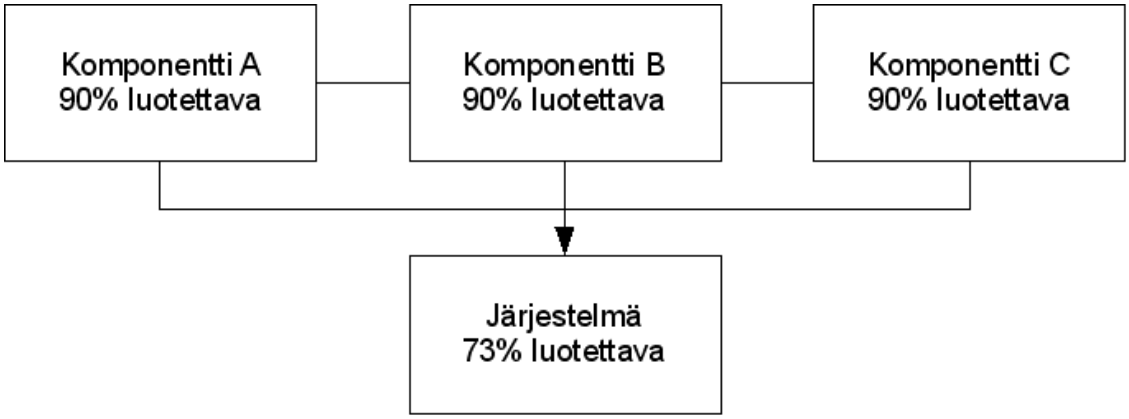
Kuva 14: Esimerkki tarinakortin kääntöpuolelle kirjoitetuista testeistä.

Kuvan 14 testit on kirjoitettu hyvin tarkalla tasolla mutta se ei ole aina välttämätöntä. Tärkeintä on, että testit ovat yksiselitteisiä jokaiselle projektissa mukana olevalle

henkilölle. Tarinakortin kääntöpuolelle kirjoitetuissa testeissä kortin etupuolella oleva käyttötapaustarina kertoo jo itsessään testien kohdealueen, joten kuvan 14 testit voisivat olla lyhimmillään kirjoitettuna oikea tilinumero sekä virheellinen tilinumero.

4 Testien automatisoiminen

Testien automatisoiminen on erittäin tärkeä osa prosessia, kun rakennetaan järjestelmiä testauslähtöisellä kehitysmenetelmällä. Testien automatisoimisen ei kuitenkaan pitäisi olla vain testauslähtöisen kehityksen yksi perusideoista. Riippumatta valitusta prosessimallista, testien automatisoimisella voidaan saavuttaa hyötyjä alentuneista kustannuksista, nousseesta tuotteen laadusta sekä kehittyneemmästä ja helpommin ymmärrettävästä ohjelmakoodista. Meszaros (2007) on listannut kirjassaan testien automatisoimisen tärkeimpiä päämääriä, joita ovat parantunut laatu, parempi ymmärrys testien kohteena olevasta järjestelmästä, vähentyneet riskit, testien helppo suorittaminen, testien helppo kirjoittaminen ja ylläpitäminen sekä testien vaatimat pienet ylläpitotoimenpiteet. Ensimmäiset kolme päämäärää kuvaavat automatisoitujen testien hyötyä ja arvoa prosessissa. Seuraavat kolme kohdistuvat automatisoitujen testien piirteisiin ja olemukseen. Duvall & al. (2007) havainnollistavat, kuinka yksittäisten komponenttien luotettavuus voi heikentää koko järjestelmän luotettavuutta (kuva 15).



Kuva 15: Kolmen komponentin muodostama järjestelmä.

Jos jokainen komponentti on erikseen mitattu 90 prosenttisesti luotettavaksi, niin silloin koko järjestelmä on yhteensä vain 73 prosenttisesti luotettava. Jos komponenttien lukumäärä kasvaa esimerkiksi sataan kappaleeseen ja jokaisen luotettavuus erikseen on 99 prosenttia, tarkoittaa se koko järjestelmän kannalta yhteensä vain 37 prosentin luotettavuutta. Testien automatisoimisella pyritään myös nostamaan erillisten komponenttien luotettavuutta, jotta koko kehitettävä järjestelmä olisi laadultaan parempi.

4.1 Yksikkötestit

Yksikkötestit toimivat kaiken perustana. Jos järjestelmän luotettavuutta ei voida mitata ja taata alimmalla mahdollisella tasolla, on se melkein mahdotonta tämän jälkeen muillakaan tasoilla. Luotettavuus onkin yksi pääkohdista, johon pyritään nimenomaan testien automatisoimisella. Ensin on varmistettava kehitteillä olevan järjestelmän perustana toimivat luokat ja tietorakenteet ennen kuin niiden päälle voidaan ryhtyä suunnittelemaan mitään uutta.

Yksikkötestit todentavat kehitettävän järjestelmän pienimpien osien oikeanlaisen toiminnan. Yleensä näitä pieniä osia ovat luokat, oliot ja metodit. Testien kirjoittamisen yhteydessä olisi tarkoitus pitää erillään testattavat luokat, mutta tämä ei välttämättä ole aina mahdollista. Näissä tapauksissa voidaan mennä takaisin suunnittelupöydän ääreen ja miettiä voisiko testiä tehdä jollain toisella tavalla, jotta saavutettaisiin korkeampi koheesio eli pystyttäisiin tekemään löyhempi kytkentä testattavien luokkien välillä. Jotkut yksikkötestit voivat olla myös riippuvaisia ulkopuolisista komponenteista. Silloin voidaan esimerkiksi käyttää hyväksi mallinnettuja olioita (ks. kohta 5.2.2), jotta pystytään suorittamaan testit ilman ulkopuolisten komponenttien mukaan ottamista. Tämä mahdollistaa myös tarpeen vaatiessa kyseisten testien ja ohjelmakoodin siirtämisen johonkin toiseen projektiin, jossa mahdollisesti halutaan ottaa mukaan uudelleenkäytettäviä komponentteja.

Yksikkötestien yksi tärkeimmistä asioista on niiden riippumattomuus järjestelmän ulkopuolisista asioista, kuten tietokannat tai kolmannen osapuolen komponentit. Näiden käyttäminen ja alustaminen on yleensä raskas sekä aikaa vievä prosessi, jota pyritään välttämään, koska yksikkötestit halutaan suorittaa mahdollisimman nopeasti jo järjestelmän kehityksen varhaisimmassa vaiheessa. Nopea sykli testien ja varsinaisen ohjelmakoodin kirjoittamisen välillä tekee yksikkötesteistä tehokkaan tavan virheiden etsimiseen (Duvall & al., 2007).

Koska yksikkötestit tapahtuvat järjestelmän kannalta oliotasolla, niiden suorittamisessa pitäisikin ottaa huomioon mahdollisimman kattava olion ominaisuuksien käyttö. Toinen tärkeä asia, joka pitää ottaa huomioon, on testien suorittaminen mahdollisimman usein (Duvall & al., 2007). Testauslähtöisen kehityksen mukaan edettäessä yksikkötestejä

kirjoitetaan määrällisesti eniten, joten näiden suorittaminen manuaalisesti on erittäin vaativa ja suuritöinen urakka. Kun testit ovat jo järjestelmän alimmalla tasolla kattavia ja niitä suoritetaan usein, voidaan olla luottavaisemmin mielin ja siirtyä kohti seuraavaa tasoa.

4.2 Komponenttitestit

Komponenttitestit varmentavat ja testaavat niitä järjestelmän osia, jotka on saatu yksikkötestien läpi hyväksytysti suoritettua ja joiden toiminta on riippuvainen järjestelmän pienempien osien tiiviistä yhteistyöstä. Komponenttitesteille alisteiset järjestelmän osat ovat usein myös riippuvaisia jostain ulkoisesta komponentista, kuten tietokanta tai tiedostojärjestelmä, joten niiden suorittamiseen kuluva aika on huomattavasti suurempi kuin yksikkötestien.

Yksi suurimmista eroista komponenttitestien ja yksikkötestien välillä on ulkoiset riippuvuussuhteet. Korkeamman tason järjestelmätestit ovat kuitenkin vielä riippuvaisempia ulkoisista osista kuin komponenttitestit. Komponenttitestien käyttämät ohjelmointirajapinnat eivät ole välttämättä näkyviä ja asiakasohjelmistoa suoraan käyttäviä. Esimerkiksi tietokannan ollessa osana järjestelmää, sen toimintaa voidaan mallintaa vain niiltä osin kuin suoritettavat testit vaativat. Tämä parantaa automaattisten testien suoritusnopeutta ja auttaa kohdistamaan testaustavoitteet vain tiettyyn ominaisuuteen kerrallaan.

Komponenttitestejä kutsutaan myös yleisesti integraatiotesteiksi (Duvall & al., 2007). Integraation alaisuuteen pyritään kuitenkin ottamaan mieluummin vain järjestelmän sisäisiä osia, vaikka aina se ei olekaan mahdollista. Yhtenä erona järjestelmätestien ja komponenttitestien välillä on testeihin valittavat ohjelmointirajapinnat. Esimerkiksi web-sovelluksen testaamisessa järjestelmätestit käyttävät hyväkseen sovelluksen web-sivuja, kun taas komponenttitestit pyrkivät käyttämään näkyvien sivujen alla olevaa toimintalogiikkaa rajapintanaan.

4.3 Järjestelmätestit

Järjestelmätestit käyttävät hyväkseen koko kehitettävän järjestelmän kaikkia osia. Tästä

johtuen ennen testien suorittamista joudutaan asentamaan kaikki mahdolliset kolmansien osapuolten komponentit ja suorittamaan niille tarpeelliset alustukset. Testien perimmäisenä ideana on varmentaa kehitettävän järjestelmän ulkoisten rajapintojen, kuten web-sivujen tai graafisten käyttöliittymien, oikeanlainen toiminta. Vaikka järjestelmätestit olisi oikealla tavalla automatisoitu, niiden läpi suorittaminen kestää selvästi pidempään kuin komponenttitestien. Yksikkötestejä ja komponenttitestejä suoritetaan varsinaisen ohjelmoinnin yhteydessä mutta ajastetut ja automatisoidut järjestelmätestit voidaan hyvinkin suorittaa esimerkiksi yöaikaan, kun työskentelyyn tarvittavien tietokoneiden resursseja ei vaadita muuhun käyttöön.

Järjestelmätestit eroavat oleellisesti toiminnallisista testeistä, jotka testaavat järjestelmää loppukäyttäjän näkökulmasta (Duvall & al., 2007). Esimerkiksi web-sivua testattaessa järjestelmätestin toiminta voi matkia selainta HTTP-protokollan välityksellä, mutta suoriutuakseen se ei tarvitse mitään varsinaista selainta käyttöönsä. Hyvin suunnitellut ja automatisoidut järjestelmätestit toimivat tukevana pohjana toiminnallisille testeille, jotka varmentavat viime kädessä järjestelmän oikeanlaisen toiminnan.

4.4 Toiminnalliset testit

Toiminnalliset testit testaavat järjestelmän määritysten yhteydessä laadittuja ja hyväksyttyjä toimintoja. Toiselta nimeltään näitä testejä kutsutaan hyväksymistesteiksi. XP määrittelee hyväksymistestit käyttötapausten perusteella laadituiksi testeiksi, joissa on kolme erillistä osaa, alustustoimenpiteet, testien suorittaminen sekä testien tulosten varmentaminen (Astels & al., 2003). Toiminnalliset testit testaavat järjestelmää loppukäyttäjän näkökulmasta mallintamalla mahdollisimman todenmukaisesti tyypillisiä käyttäjän toimintoja.

Erilaiset graafiset käyttöliittymät ovat hyvin yleisiä toiminnallisten testien kohteita. Niiden testaamisessa on perinteisesti käytetty ihmisiä, jotka manuaalisesti testaavat järjestelmän toimintoja. Tarjolla on kuitenkin myös erilaisia työkaluja, joilla toiminnallisia testejä voidaan automatisoida. Esimerkiksi ihmisen suorittamia toimenpiteitä voidaan nauhoittaa ja toistaa näitä aina uudelleen haluttuna ajankohtana. Tällaisissa tapauksissa yleensä kuitenkin jonkun testaajista pitää käydä läpi kaikki mahdolliset järjestelmän käyttöliittymän sisältämät polut, jotta testausta voidaan pitää

mahdollisimman luotettavana.

4.5 Yksinkertaisin testien automatisointistrategia

Meszaros (2007) kirjoittaa kirjassaan yksinkertaisimmasta testien automatisointi-strategiasta, joka voisi mahdollisesti toimia. Hän listaa strategian vaiheet viiteen eri osaan, jotka ovat kehitysprosessi, asiakastestit, yksikkötestit, testattavuuden suunnittelu sekä testien organisointi. Jos näitä malleja ja periaatteita käytetään tehokkaasti hyväksi, on todennäköistä, että testien automatisoiminen onnistuu hyvin. Jos tämä yksinkertaisin strategia ei toimi kehitteillä olevan järjestelmän kanssa, niin on myös olemassa vaihtoehtoisia malleja, joiden avulla voidaan saada testien automatisoiminen onnistumaan.

4.5.1 Kehitysprosessi

Ensimmäinen huomioon otettava asia on kehitysprosessi ja kuinka se vaikuttaa testeihin sekä testaukseen. Testien kirjoittamisen ajankohta ja aloittaminen on ensimmäinen kysymys, johon täytyy saada vastaus. Testauslähtöinen ohjelmistokehitys antaakin tähän helpon ratkaisun. Ennen ensimmäistäkään varsinaista ohjelmakoodiriviä kirjoitetaan testi. Suorana hyötynä tästä voidaankin nähdä, että saadaan hyväksytty määritys sille, mitä toimintoja järjestelmän tulee loppujen lopuksi toteuttaa.

Testien automatisointia ei pitäisi myöskään nähdä hidastavana tai resursseja vievänä osana sovellettavaa kehitysprosessia. Vaikka valittu prosessimalli ei olisikaan testauslähtöinen, se ei ole mikään syy olla käyttämättä automaattisia testausmenetelmiä. Käyttötapauslähtöisessä kehityksessä testien automatisoiminen voidaan aloittaa järjestelmän loppukäyttäjille soveltuvien toiminnallisten testien automatisoimisella. Näillä pyritään varmistamaan, että kaikki sovitut järjestelmän toiminnot on toteutettu ja ne toimivat oikein. Automaattiset yksikkötestit valjastetaan asiakastestien käyttöön, jotta voidaan varmentaa, että näkyvän järjestelmän taustalla tapahtuvat toiminnot ovat oikein toteutettu. Yksikkötestien kattavuutta tarkasteltaessa voidaan myös hyvin ottaa käyttöön erilaisia työkaluja, joilla saadaan selville mitä osia varsinaisesta ohjelmakoodista suoritetaan ja mille osille ohjelmakoodia ei ole olemassa minkäänlaista testiä.

Eri käyttötarkoitukseen tehdyt testit tulisi olla organisoitu sillä tavalla, että ne ovat erillisiä kokonaisuuksia ja tarpeen mukaan riippumattomia toisistaan. Yksikkötestejä suoritetaan testauslähtöisessä ohjelmistokehityksessä huomattavasti useammin kuin asiakastestejä, joten nämä on syytä irrottaa kokonaan irrallisiksi asiakastesteistä. Eri komponenttien integraatiovaiheessa voidaan valjastaa molemmat testikokonaisuudet käyttöön. Tässä vaiheessa täytyy kuitenkin varoa, ettei eri testikokonaisuuksien yhdistäminen hidasta liikaa kehitysprosessia. Testauslähtöisessä kehityksessä järjestelmän kehitettävien ominaisuuksien ja integroitavien komponenttien sisällä pitämät vaiheet tulisi kuitenkin suorittaa suhteellisen nopeasti.

Testauslähtöinen kehitys takaa myös sen, että kehitteillä oleva järjestelmä on suunniteltu hyvin testattavaksi. Yksikkötestien kirjoittaminen ennen varsinaista ohjelmakoodia auttaa kehitettävän järjestelmän suunnittelussa. Tällä tavoin voidaan keskittyä paremmin järjestelmän rakenteelliseen suunnitteluun ja toteuttaa järjestelmän toimintalogiikka hyvin määriteltyihin, erillisiin, luokkiin ja olioihin, jotka voidaan testata irrallisina kokonaisuuksina muista osista. Esimerkiksi tietokannan ollessa osana kehitettävää järjestelmää, olisi hyvin tärkeää pitää toimintalogiikan yksikkötestit riippumattomina tietokannasta.

4.5.2 Asiakastestit

Asiakastestien tärkein tehtävä on testata niitä ominaisuuksia, jotka on määritelty yhdessä asiakkaan kanssa ja joita asiakas haluaa järjestelmän toteuttavan. Testien suunnittelu ja selvittäminen on tärkeä osa prosessia, koska nämä testit määrittelevät sen miltä toimiva järjestelmä tulee näyttämään. Kehitysyksikössä työskentelevien henkilöiden on myös oltava perillä asiakkaan vaatimuksista, koska viime kädessä he tulevat toteuttamaan testit ja varsinaisen järjestelmän. Asiakastestejä voidaan automatisoida tapauksesta riippuen esimerkiksi automaattisten testaustyökalujen avulla tai tietolähtöisellä testauksella. Tietolähtöisessä testauksessa testien hyväksi käyttämät ja vaatimat tietokokonaisuudet kirjoitetaan tiedostoon, josta toteutettu tulkki lukee niitä sekä suorittaa testit. Joissain tapauksissa on myös mahdollista käyttää nauhoitettuja testejä mutta näiden yhteydessä on oltava varuillaan, koska niiden käyttäminen voi hyvinkin nostaa testien ylläpitokustannuksia.

Asiakastestien on tärkeää havainnollistaa loppukäyttäjille kuinka järjestelmää käytetään ja kuinka se toimii. Olisi kuitenkin muistettava pitää testit suhteellisen lyhyinä, jotta puutteiden ja virheiden etsiminen on helpompaa sekä testien ymmärrettävyys on parempaa. Hyvin suunniteltuja ja kirjoitettuja testejä voidaan myös käyttää dokumentaationa toteutettavasta järjestelmästä. Tämä osaltaan lisää asiakkaan luottamusta siihen, että järjestelmä toimii niin kuin sen pitääkin. Testien pitäminen lyhyinä ja yksinkertaisina auttaa myös suorittamaan testejä eristettyinä eri arkkitehtuurikerroksille.

Jokaisen testin lähtötilanteessa on myös hyvä ottaa selville, mitä toimenpiteitä täytyy tehdä ennen kuin testit voidaan suorittaa. Tällä tavalla voidaan välttää tilanteita, joissa toinen testi käyttää edellisen testin tuloksia hyväkseen. Toisistaan riippuvaiset testit eivät välttämättä tuota oikeita tuloksia ja sen takia mikään testi ei saisi käyttää sellaista tietoa hyväkseen, jota se ei ole itse alustanut. Testien alkutilanteessa pyritäänkin aina aloittamaan niin sanotusti puhtaalta pöydältä muihin testeihin nähden.

4.5.3 Yksikkötestit

Yksikkötestit ovat tehokkaita silloin, kun ne ovat täysin automaattisia. Kun yksikkötestit todentavat vain yhden vaatimuksen tai ehdon kerrallaan, on niiden avulla helpompi jäljittää puutteita ja virheitä. Yksikkötestit toimivat myös kehitettävän järjestelmän testausdokumentaationa silloin, kun ne ovat muodostettu nelivaiheisiksi testeiksi, joista jokainen vaihe on selvästi erotettavissa muista. Nämä neljä vaihetta ovat järjestyksessään: testin alustus, testin suoritus, tulosten todentaminen sekä lopputoimenpiteiden suorittaminen.

Alustusvaiheessa suoritetaan testin vaatimat toimenpiteet, joita voivat olla esimerkiksi tiettyjen muuttujien alustaminen tai haluttujen tietojen hakeminen käytetystä tietovarastosta. *Suoritusvaiheessa* testit ajetaan läpi ja ne kommunikoivat testien kohteena olevan järjestelmän kanssa. *Tulosten todentamisen* yhteydessä verrataan testeistä saatuja tuloksia odotettuihin tuloksiin. Viimeisimpänä *lopputoimenpiteiden* yhteydessä palautetaan testattava järjestelmä ja testeissä apuna käytetyt mahdolliset muut osat sellaiseen tilaan, kuin ne olivat ennen testien aloittamista.

Yksikkötestien kirjoittaminen aloitetaan kirjoittamalla testiluokka jokaista varsinaisen järjestelmän luokkaa vastaan. Jokaisen testin olisi myös hyvä luoda täysin puhdas ympäristö omaan käyttöönsä ennen suoritusta, jotta voidaan välttyä mahdollisista riippuvuuksista muiden testien välillä. Jokainen erillinen testi toteutetaan testiluokassa testimetodina. Testimetodien on myös mahdollista tehdä omat alustuksensa ennen suorittamista, jos testiluokan alustukset ja toimenpiteet eivät ole riittäviä. Tässä tapauksessa olisi hyvä suoriutua näistä toimenpiteistä pienimmällä mahdollisella vaivalla.

Jotta testit saadaan tarkastamaan itse itsensä, täytyy jokaiselle testille ilmaista odotettu tulos, jota verrataan saatuun testitulokseen. Yksikkötestaustyökaluissa tämä tapahtuu yleensä testausympäristöön sisäänrakennettujen vertailumetodien avulla mutta tapauksesta riippuen voi olla myös mahdollista kirjoittaa omia kustomoituja vertailumetodeja. Itse itsensä tarkastava testi ei vaadi minkäänlaisia toimenpiteitä, kun se suoritetaan onnistuneesti läpi. Ainoastaan virheellisen suorituksen jälkeen on syytä tarkastaa missä kohdassa virhe on tapahtunut ja selvittää millä toimenpiteillä siitä voidaan helposti selviytyä.

Kehitteillä oleva järjestelmä voi myös sisältää koodirivejä, joita mikään testi ei pääse suorittamaan. Tällöin voidaan korvata oikea olio testikohtaisella oliolla, joka antaa testauksen kohteena olevalle järjestelmälle halutut epäsuorat syötteet. Näin päästään käsiksi järjestelmän sellaisiin osiin joihin ei muuten päästä. On myös mahdollista, että järjestelmä sisältää sille asetettuja testaamattomia vaatimuksia, koska kaikkia sen toimintoja ei voida tarkastella julkisten rajapintojen kautta. Sekä testaamattomat koodirivit että testaamattomat vaatimukset ovat tuotantovirheitä, jotka ilmenevät liian suurina virhemäärinä järjestelmän muodollisen testauksen yhteydessä. Syitä näihin voivat olla muun muassa liian epäsäännöllisesti ja harvoin suoritetut yksikkötestit tai testien automatisoimisen puuttuminen. Automaattisilla yksikkötesteillä pyritäänkin saamaan jo tuotantovaiheessa kehitettävän järjestelmän jokainen mahdollinen koodirivi testattua, jotta voidaan varmentua järjestelmän oikeanlaisesta toiminnasta.

4.5.4 Testattavuuden suunnittelu

Testattavuuden suunnittelu on tärkeää, kun halutaan mahdollistaa automaattisten testien käyttöön ottaminen. Automatisoiminen onnistuu paljon helpommin silloin, kun käytetään järjestelmän suunnittelussa ja toteutuksessa kerrosarkkitehtuuria. Kerrosarkkitehtuurissa toimintalogiikka on erillään tietokannasta ja käyttäjälle näkyvästä käyttöliittymästä mahdollistaen erilaiset testijoukot jokaiselle järjestelmän eri kerrokselle. Eri kerrosten ja testijoukkojen erillään pitäminen vähentää myös niiden välisiä riippuvuuksia parantaen samalla testien luotettavuutta ja mahdollisesti suoritusnopeutta.

Graafiset käyttöliittymät lisäävät osaltaan testattavuuden suunnittelun merkitystä. Niiden osalta olisi hyvä pitää monimutkainen käyttöliittymälogiikka erillään näkyvistä luokista. Testien kannalta voi olla hyvinkin turhaa ryhtyä alustamaan käyttöliittymän näkyviä objekteja, kuten painikkeet, jos näillä ei ole mitään merkitystä varsinaisen testattavan logiikan kanssa. Usein suuret alustusmäärät saattavat myös kuormittaa turhaan testausjärjestelmän resursseja.

Kehitettävä järjestelmä voi hyvinkin olla erittäin monimutkainen ja sisältää sellaisia komponentteja, joita halutaan käyttää uudelleen jonkin toisen projektin yhteydessä. Tässä tapauksessa on hyvä vahvistaa yksikkötestejä komponenttitesteillä, jotka varmentavat jokaisen erillisen komponentin toiminnan irrallisina yksikköinä kehitteillä olevasta järjestelmästä. Apuna voidaan käyttää testejä varten tehtyjä testikohtaisia vastineita, jotka toimivat kuten niiden mallintamat alkuperäiset komponentit. Nämä vastineet ovat helposti erilleen irrotettavia kokonaisuuksia, joita voidaan käyttää hyväksi myös muissa projekteissa.

4.5.5 Testien organisointi

Testien organisointi voi joskus tuntua hankalalta, jos kehityksen yhteydessä tuntuu syntyvän liian paljon testimetodeja tiettyä testiluokkaa kohden. Testimetodit voidaan organisoida *ominaisuusperusteisesti* tietyn ominaisuuden mukaan omiin testiluokkiinsa ja näin saadaan tietyn luokan metodeilla testattua juuri tietyt järjestelmän määritysten yhteydessä sovitut ominaisuudet. Tällainen ominaisuuksiin perustuva metodien

organisointi voi tapahtua joko jakamalla yhdestä testiluokasta metodit omiin erillisiin luokkiinsa tai yhdistämällä monesta luokasta samoja ominaisuuksia testaavat juuri tiettyyn testiluokkaan.

Toinen mahdollinen organisointiperuste voi olla *testausympäristöperusteinen*: testimetodijoukon vaatimat alustustoimenpiteet ja testausta varten tarvittava testausympäristö. Jos tietyt metodit käyttävät samoja alustustoimenpiteitä ennen testien suorittamista, on niiden järkevää sijaita samassa testiluokassa, jotta testit saadaan suoritettua nopeammin läpi. Yleensä samoja alustustoimenpiteitä käyttävät metodit testaavat myös samanlaisia ominaisuuksia, joten ominaisuusperusteinen ja testausympäristöperusteinen organisointi eivät välttämättä eroa toisistaan millään tavalla.

5 TDD -malleja

Testauslähtöiseen ohjelmistokehitykseen on olemassa tiettyjä malleja ja tapoja edetä, jotka helpottavat kehitystyötä. Niistä voi olla huomattavaa apua, kun lähdetään suunnittelemaan itse testien toteutusta. Testit ovat automaattisia, joten niiden läpi ajaminen on hyvin nopeaa ja sen voi tehdä aina, kun alkaa tuntua siltä, että uudessa ohjelmakoodissa saattaisi olla jotain vialla. Ajettavien testien ei pitäisi olla millään tavalla riippuvaisia toisistaan. Riippumattomuudella pystytään poistamaan ihmisen aiheuttamien virheiden määrää ja voidaan halutessa suorittaa vain osa testeistä. Toisistaan riippumattomat testit rohkaisevat myös suunnittelemaan ratkaisut korkeammalla tasolla, jolloin saadaan tuloksena modulaarisempaa ohjelmakoodia. Beck (2003) esittelee kirjassaan muutamia testauslähtöisen ohjelmistokehityksen toimintamalleja, joiden avulla voidaan suoriutua etukäteen hankaliltakin tuntuvista tehtävistä. Tässä luvussa esitellään punaisen palkin malleja (Red Bar Patterns), testausmalleja (Testing Patterns) sekä vihreän palkin malleja (Green Bar Patterns).

5.1 Punaisen palkin malleja

Punaisen palkin mallit ovat hyödyllisiä, kun mietitään testaukseen liittyviä asioita, jotka eivät varsinaisesti liity käytettäviin työkaluihin tai ohjelmointikieliin. Ne auttavat päättämään milloin aloitetaan testien kirjoittaminen, mistä aloitetaan testien kirjoittaminen ja milloin on hyvä aika lopettaa testien kirjoittaminen.

5.1.1 Käynnistystesti

Ensimmäisen testitapauksen valinta voi olla joskus vaikeaa. Ei ole olemassa mitään yhtä oikeaa ratkaisua siihen, mistä aloittaa ongelman ratkaisua. Käynnistystestiksi (Starter Test) valittavan testin ei tarvitse olla välttämättä kehitettävän järjestelmän kannalta kovinkaan realistinen. Todenmukaisen testin yhteydessä joudutaan miettimään jo tarkemmin minne operaatiot kuuluvat, mitkä ovat metodien ja luokkien järkeviä nimiä, mitkä ovat oikeellisia syötteitä ja kuinka ne tarkistetaan sekä mitä muita testejä mahdollisesti tarvitaan. Testauslähtöisen kehityksen periaatteiden mukaan kehityssykli on hyvin nopeaa ja palautetta halutaan saada miltei koko ajan. Ensimmäiseksi testiksi

voidaan hyvinkin valita sellainen, jonka mahdolliset syötteet ja tulokset on suhteellisen helppo havaita. Näin päästään aloittamaan varsinainen kehitystyö huomattavasti nopeammin.

5.1.2 Askel kerrallaan

Testilistaan on kirjoitettu ylös kaikki ne asiat, joihin halutaan saada varmistus testitapausten kautta. Testilistan läpikäyminen saattaa johtaa osittavaan lähestymiseen, koska testitapausten kirjoittaminen voidaan aloittaa yksinkertaisesta tapauksesta, joka kuitenkin kattaa kehitettävän järjestelmän kannalta suurempaa toiminnallisuutta. Lähestyminen voidaan myös nähdä kokoavana. Tällöin aloitetaan pienimmästä osasta pala kerrallaan ja edetään kohti suurempaa kokonaisuutta. Tavoitteena on kuitenkin valita sellainen testi, joka voidaan varmasti kirjoittaa toteutettavaan muotoon ja edetä eteenpäin askel kerrallaan (One Step Test).

5.1.3 Perustelutesti

Eräs tapa edetä on ajatella kysymykset ja vastaukset testien muodossa. XP:n oppien mukaisesti kehitys voi tapahtua pariohjelmointina ja tällöin on helpompaa miettiä parin kanssa mahdollisia kysymyksiä ja vastauksia. Perustelutestiä (Explanation Test) lähestymistapana käytettäessä järjestelmän vaatimukset tulevat sellaisinaan testattavaan muotoon.

5.1.4 Oppimistesti

Kun käytetään kolmannen osapuolen tuottamaa komponenttia hyväksi ohjelmakoodissa, täytyy ensin tietää miten se toimii. Toiminnallisuuden oppiminen voi tapahtua testien kirjoittamisen muodossa. Näin varmistetaan, että ulkopuolinen komponentti todellakin toimii halutulla ja ymmärretyllä tavalla. Oppimistestin (Learning Test) mukaista toimintamallia voidaan myös käyttää hyväksi, jos on tarpeen muuttaa jonkun toisen tekemää vanhaa koodia osaksi uutta kehitettävää järjestelmää.

5.1.5 Lisätesti

Joskus kehittäjien ajatukset lähtevät harhailemaan pois varsinaisesta ongelmasta. Tämä ei kuitenkaan ole välttämättä pelkästään huono asia. Uudet ideat ja ajatukset voivat sisältää myöhemmin järjestelmän kehityksessä eteen tulevia ongelmakohtia. Lisätestejä (Another Test) on hyvä kirjoittaa heti ylös testilistaan ja palata takaisin alkuperäisen ongelman pariin. Tällä tavoin saadaan testilistaan huomaamatta lisää pituutta ja kattavuutta.

5.1.6 Regressiotesti

Huomattaessa virhe järjestelmässä, kirjoitetaan sitä vastaava testi. Kun kehitetään järjestelmää testauslähtöisesti, varsinaista regressiotestausta (Regression Test) tapahtuu koko ajan. Ennen kuin kirjoitetaan riviäkään varsinaista ohjelmiston lähdekoodia, sille ollaan jo kirjoitettu testiluokka ja metodit sekä vanhat testit ajettu läpi. Virheen tullessa esiin jo julkaistussa järjestelmässä, on hyvä miettiä kuinka se olisi voitu ennalta löytää testitapausten avulla. Missään tapauksessa sataprosenttista regressiotestien kattavuutta ei voida saavuttaa, mutta järjestelmän kehityksen edistyessä myös testitapausten kattavuus lisääntyy.

5.1.7 Tauko

Väsyneenä ohjelmoija on huomattavasti taipuvaisempi virheisiin, joka taas myöhemmin kasvattaa selkeästi työmäärää. Myös liian kauan saman ongelman parissa toimiminen saattaa estää näkemästä selkeää ratkaisua. Tällöin olisi hyvä ymmärtää pitää pieni tauko (Break) työstä. XP antaa myös ohjeita tällaisia tilanteita varten. Suositeltava työmäärä olisi 40 tuntia viikossa, kahdeksan tuntia päivässä. Joskus kuitenkin kiireessä ja paineen alla työskennellessä tällaista ohjetta on melkein mahdotonta pitää mielessä.

5.1.8 Remontointi

Joskus on helpompi remontoida (Do Over) kuin yrittää väkisin jatkaa vanhaan suuntaan. Silloin pitäisi löytyä riittävästi rohkeutta poistaa vanha koodi ja miettiä uutta lähestymistapaa. Mitä aiemmin tällainen tilanne huomataan ja aloitetaan puhtaalta

pöydältä sitä vähemmän kulutetaan aikaa projektiin. Pariohjelmoinnissa parin vaihto voi motivoida aloittamaan uudelleen koodin kirjoitusta, kun joudutaan selventämään työparille mistä vanhassa koodissa oikein on kysymys.

5.1.9 Halpa pöytä, hyvä tuoli

Fyysinen hyvinvointi on töissä hyvin tärkeää. Ihmiset, jotka eivät ole fyysisesti riittävän hyvässä kunnossa, ovat paljon alttiimpia olemaan poissa töistä kuin terveet ihmiset. Sairauspoissaolot taas vastaavasti kasvattavat työnantajan kuluja ja aiheuttavat projektien aikataulujen venymisiä. Ohjelmoijalle fyysisen hyvinvoinnin kannalta tärkein asia työpaikalla on hyvä tuoli, joka tukee selkää. Tähän asiaan panostaminen voi maksaa itsensä takaisin joskus hyvinkin nopeasti. Myös resurssien allokoinnin priorisointi voi auttaa. Vanha ja hidas kone voi toimia helposti koneena, jolla luetaan sähköpostia, mutta kehityskoneen olisi hyvä olla tarpeeksi tehokas. Halpa pöytä, hyvä tuoli (Cheap Desk, Nice Chair) auttaa palauttamaan mieleen sen mikä on työskentelyolosuhteissa tärkeintä.

5.2 Testausmalleja

Testausmallit antavat tarkemman kuvan varsinaisesta testauksesta. Niiden avulla paneudutaan testauksen tekniikoihin ja testien kirjoittamiseen.

5.2.1 Lapsitesti

Testitapaus voi joskus paisua liian suureksi ja hankalaksi hallita, jotta se voitaisiin suorittaa nopeasti. Hyvä tapa on kirjoittaa pienempi testitapaus, niin sanottu lapsitesti (Child Test), joka kuvastaa alkuperäisen, suuremman, testitapauksen osaa. Pienempi testitapaus on helpompi suorittaa ja tämän jälkeen se voidaan ajaa muiden testien yhteydessä. Tarkoituksena on kuitenkin säilyttää jatkuva, sykliltään suhteellisen nopea, kehitys koodin suhteen, joten punainen/vihreä/refaktorointi prosessin säilyttäminen on hyvin tärkeää.

5.2.2 Mallinnettu olio

Mallinnettujen olioiden (Mock Object) tarkoituksena on mallintaa todellisen ympäristön tiettyä resurssia tai osaa, esimerkiksi tietokantaa. Tietokanta saattaa olla liian raskas toimimaan tehokkaasti usein ajettujen testien yhteydessä, jolloin on hyvä mallintaa sitä tekemällä vastaavanlainen olio suppeammilla toiminnallisuuksilla. On kuitenkin hyvä muistaa, että mallinnettu olio ei välttämättä toimi samalla tavalla kuin oikea ja alkuperäinen mallinnettu komponetti. Tämän virheen mahdollisuutta voidaan pienentää tekemällä testejä mallinnetuille oliolle. Näitä kirjoitettuja testejä voidaan myöhemmin käyttää hyväksi, kun testataan oikeata oliota oikeassa ympäristössä.

5.2.3 Itseensä siirto

Olioiden välisen kommunikaation testaamisessa voidaan käyttää hyväksi toteutuksen alla olevaa testiluokkaa. Testiluokka itsessään toimii eräänlaisena mallinnettuna oliona ja kommunikoi testauksen kohteen kanssa (Feathers, 2001). Itseensä siirto (Self Shunt) voidaan nähdä mallinnettujen olioiden yhtenä erikoistapauksena.

5.2.4 Lokiviestit

Kehittävän järjestelmän sisäisen toiminnan kannalta saattaa olla tärkeää missä järjestyksessä viestit kulkevat tai mitä viestiä milloinkin kutsutaan. Tällöin voidaan pitää lokia viesteistä (Log String), joita kutsutaan ja testata niiden oikeaa järjestystä.

5.2.5 Törmäysnukke

Yksi testauslähtöisen kehityssuuntauksen perusolettamuksista on, että testaamaton ohjelmakoodi ei voi toimia. Tämän takia on tärkeää tehdä testitapauksia myös harvoin, poikkeuksen yhteydessä, suoritettavalle koodille. Törmäysnukke (Crash Test Dummy) toimii myös eräänlaisena mallinnetun olion erikoistapauksena. Tosin sillä erotuksella, että nyt ei ole mitään tarvetta mallintaa koko testauksen kohteena olevaa oliota.

5.2.6 Rikkinäinen testi

Eräs tapa lopettaa päivän ohjelmointityö yksin ohjelmoidessa on jättää viimeisin työn alla ollut testi rikkinäiseksi (Broken Test). Seuraavan kerran, kun palataan jälleen pöydän ääreen, löydetään helposti kohta, josta voidaan jatkaa. Tällä tavalla saatetaan havaita myös parempi tapa lähestyä keskeneräistä ongelmaa.

5.2.7 Puhdas palautus

XP ja TDD toimivat parhaimmillaan tehtäessä työtä tiimeissä. Rikkinäistä koodia ei kuitenkaan koskaan saisi olla versionhallinnassa muiden saatavilla. Olisi siis hyvä muistaa palauttaa puhdasta koodia versionhallintaan (Clean Check-in) mieluummin liian usein kuin harvoin. Tällä tavoin muut tiimin jäsenet ovat selvillä etenemisestä ja heillä on koko ajan saatavilla viimeisintä toimivaa koodia. Päivän päätteeksi on parempi poistaa turha koodi kokonaan kuin antaa muille virheellinen ajatus, että se on toimivaa.

5.3 Vihreän palkin malleja

Vihreän palkin malleja käytetään hyväksi, kun halutaan saada testit suoritettua onnistuneesti läpi. Toisin sanoen halutaan yksikkötestaustyökalun punainen palkki nopeasti vihreäksi. Tarkoitus ei kuitenkaan ole välttämättä tuottaa kerralla ohjelmakoodia, joka on lopullista. Tavoitteena on ainoastaan päästä eteenpäin työn alla olevassa tehtävässä.

5.3.1 Lavastus

Testauslähtöisen ohjelmistokehityksen tulisi olla nopeaa ja joustavaa. Joskus on tarpeellista saada ohjelmakoodi läpäisemään testit mahdollisimman nopeasti ilman, että suunnittelemiseen käytetään paljon aikaa. Silloin voidaan lavastaa (Fake It ('Til You Make It)) ohjelma toimimaan halutulla tavalla. Kun tiedetään, että testit on suunniteltu huolella ja ei ole tarpeen puuttua niiden toteutukseen, voidaan itse ohjelmakoodissa käyttää hyväksi vakioita. Testin ja ohjelmakoodin onnistuneen suorituksen jälkeen ryhdytään refaktoroimaan ja pystytään vähitellen muuttamaan vakioita lausekkeiksi ja muuttujiksi. Tällaisella lähestymistavalla voi olla myös psykologista vaikutusta

ohjelmiston kehittäjiin. Kun huomataan, että punainen palkki koodin kirjoittamisen jälkeen muuttuu vihreäksi, voidaan luottavaisin mielin siirtyä refaktoroimaan tietäen, että testi on ajettu onnistuneesti läpi.

5.3.2 Kolmiomittaus

Ohjelmakoodin toteutuksen suunnittelu voi joskus olla todella hankalaa, kun ei ole varmuutta onko kirjoitettu testi kattava. Tällöin voidaan kirjoittaa toinen testi eri parametreilla käyttäen hyväksi kolmiointia (Triangulate), jotta saadaan varmuus koodin toimivuudesta. Erityisesti, kun abstrahoidaan joitakin toteutettavia laskukaavoja, voidaan tarvita useampia testejä.

5.3.3 Itsestään selvä toteutus

Kun tiedetään varmasti, mitä ohjelmakoodin tulee tehdä ja miten se on parasta toteuttaa, voidaan kirjoittaa se suoraan valmiiksi ilman pieniä välivaiheita. Testit kyllä huolehtivat siitä onko koodi varmasti oikein kirjoitettu ja läpäiseekö se niiden asettamat vaatimukset, mutta ohjelmistokehittäjän tulee itse huolehtia siitä onko toteutus varmasti paras mahdollinen. Kehittyneemmät ja harjaantuneemmat testauslähtöisen ohjelmistokehityksen periaatteiden mukaan toimivat varmasti kirjoittavat ohjelmakoodia suoraan ilman refaktoroinnin tarvetta, kun kyseessä on yksinkertaisempia operaatioiden toteutuksia, mutta ne joille TDD on vähemmän tuttua, tulisi mieluummin ensin oppia menemään punainen/vihreä/refaktorointi kaavan mukaan, ennen kuin aletaan jättämään joitain askeleita pois välistä. Selvä toteutus (Obvious Implementation) auttaa pitämään mielessä TDD:n edellyttämät vaiheet ohjelmoinnin yhteydessä.

5.3.4 Yhdestä moneen

Objektikokoelmat eivät ole mitenkään harvinaisia ohjelmakoodissa ja usein tuleekin tarve niiden toteuttamiseen, että saadaan ratkaisumallista paras mahdollinen. Tällöin voidaan operaation toteutusta testata ensin yhdellä objektilla. Tämän jälkeen muutetaan testi ja sitä vastaava koodi käsittelemään kokoelmia. Tällä tavoin varmistetaan ensin toimivuus toteutuksen alla olevan objektimallin kanssa ennen kuin siirrytään

käsittelemään kokonaista kokoelmaa. Yhdestä moneen -malli (One to Many) siirtyy askeleittain pienemmästä asiakokonaisuudesta suurempaan.

6 TDD-työkaluja

Testauslähtöinen ohjelmistokehitys ei olisi kovinkaan kannattavaa, jos sen apuna ei olisi tehokkaita työkaluja. TDD:n periaatteiden mukaan kehityssyklin tulee olla joustava ja nopea, joten apuvälineiden ja työkalujen käyttämiseen ei saa kulua liikaa aikaa. Apuvälineitä on kehitetty testauksen eri vaiheisiin ja niitä löytyy useille eri kehitysympäristöille ja ohjelmointikielille. Paras tapa tutustua näihin testausvälineisiin on kokeilla itse mikä sopii parhaiten omiin tarkoituksiin. Kirjallisuudessa kuitenkin puhutaan eniten XUnit-yksikkötestausperheen testausympäristöistä. Nämä testaus-ympäristöt toimivat ohjelmointikielestä riippumatta eräänlaisina testauslähtöisen ohjelmistokehityksen peruskivinä. Beck (2003) on toiminut XUnit-ympäristöjen yhtenä ensimmäisistä kehittäjistä.

Taulukkoon 1 on koottu tässä tutkielmassa tarkasteltavat työkalut. Kaikkia työkaluja ei ole päivitetty kovin aktiivisesti, kuten viimeinen sarake ilmaisee. Mikäli erikseen ei ole muuta ilmoitettu työkalun esittely perustuu taulukossa 1 ilmaistavaan kotisivuun.

Taulukko 1: Testaustyökaluja.

Ympäristö	**Väline**	**Kotisivu**	**Viimeisin päivitys**
.NET	NUnit	http://www.nunit.org/	2007
.NET	AnyUnit	http://www.anyunit.com/	2006
C & C++	CppUnit	http://cppunit.sourceforge.net/	2008
C & C++	EasyUnit	http://easyunit.sourceforge.net/	2005
Tietokanta	UtPLSQL	http://utplsql.sourceforge.net/	2005
Tietokanta	SQLUnit	http://sqlunit.sourceforge.net/	2006
GUI	Selenium	http://selenium.openqa.org/	2008
GUI	Abbot	http://abbot.sourceforge.net/	2006
Java	Unitils	http://www.unitils.org/	2007
Java	jMock	http://www.jmock.org/	2007
Java	TestNG	http://testng.org/	2007
Java	JUnit	http://www.junit.org/	2007

6.1 .NET

.NET-ohjelmointikielille on tarjolla monia testausympäristöjä. Testauslähtöiseen ohjelmistokehitykseen perehtyneellä sivustolla, www.testdriven.com, on listattuna useita tällaisia työkaluja, joista voi olla paljonkin hyötyä projektin aikana.

6.1.1 NUnit

Ehkä yleisimmin käytetty yksikkötestausympäristö kaikille .NET-ohjelmointikielille on NUnit, joka on alunperin siirretty kohdassa 6.5.4 tarkasteltavasta Java-ohjelmointikielen vastaavasta JUnit-testausympäristöstä. TDD näkyy myös NUnitin kehityksessä. Yhtään riviä koodia ei hyväksytä mukaan varsinaiseen kehitystyökaluun, jos sille ei ole olemassa vastaavaa testiä. NUnitin kehitysyhteisö on www-sivuillaan antanut kuvan 16 mukaisen esimerkin, jonka mukaan koodia tulisi kirjoittaa.

```
namespace   NUnit.Framework
{
  using System;

  ///<summary>A set of Assert methods.</summary>
  public class Assertion
  {
    /// <summary>
    /// Asserts that a condition is true. If it isn't it
    /// throws
    /// an <see cref="AssertionFailedError"/>.
    /// </summary>
    /// <param name="message"> The condition, if
    /// false</param>
    /// <param name="condition"> The evaluated
    /// condition</param>

    static public void Assert(string message, bool condition)
    {
      if (!condition)
        Assertion.Fail (message);
    }
  }
}
```

Kuva 16: NUnit-esimerkkikoodia.

Myös Stott ja Newkirk ovat selvittäneet artikkelissaan (Stott & Newirk, 2004) TDD:n ja NUnitin käytön hyötyjä ohjelmistokehityksessä. Testauslähtöisessä ohjelmisto-kehityksessä saatetaan kirjoittaa tuhansia testejä, joten on tärkeää organisoida nämä testit järkevästi ja pitää ne helposti ajettavissa, jotta kehityssykli olisi tarpeeksi nopea. NUnit tarjoaa juuri tätä varten helppokäyttöisen työkalun. Microsoftin Visual Studio -kehitystyökalu on varmasti tuttu kaikille ohjelmistokehittäjille, jotka työskentelevät

.NET-ympäristössä. Aivan kuten Visual Studioon ladataan projekteja, voidaan NUnitiin järjestellä ja ladata testitapauksia yksittäisiksi projekteiksi. NUnitin avulla voidaan automatisoida jokaisen testitapauksen alussa ja lopussa tapahtuvat alustustoimenpiteet, jotka voivat olla ehdottoman tärkeitä, jos ajetaan peräkkäin useita testejä. Myös ohjelmiston käännösprosessi ja siihen liittyvien testien suorittaminen voidaan automatisoida tapahtuvaksi esimerkiksi yöaikaan. Tällä tavalla kehittäjät näkevät heti aamulla mitkä osat ohjelmistoa vaativat mahdollisesti korjaustoimenpiteitä.

6.1.2 AnyUnit

Toisin kuin NUnit, AnyUnit on maksullinen työkalu .NET -ympäristöissä tapahtuvaan ohjelmistokehitykseen. Se on lisäosa Microsoftin Visual Studio -työkaluun ja sen tarkoituksena on luoda sekä suorittaa eri ohjelmointikielillä kirjoitettua ohjelmakoodia. AnyUnitin avulla voidaan nopeuttaa ohjelmistokehitystä luomalla jokaiselle alkuperäisessä projektissa määritetylle luokalle ja funktiolle testikoodipohja. Nämä testikoodipohjat voidaan ottaa myöhemmin nopeasti käyttöön halutuissa projektin osissa. AnyUnit tarjoaa myös graafisen käyttöympäristön, josta nähdään mitkä testit ovat suoriutuneet hyväksytysti ja missä ohjelmakoodin osissa olisi vielä jotain korjattavaa. Kuvassa 17 on esitetty AnyUnit -esimerkkikoodia.

```
ExampleTestCase.h
#ifndef CPP_UNIT_EXAMPLETESTCASE_H
#define CPP_UNIT_EXAMPLETESTCASE_H
#include <cppunit/extensions/HelperMacros.h>

/*
 * A test case that is designed to produce
 * example errors and failures
 *
 */

class ExampleTestCase : public CPPUNIT_NS::TestFixture
{
  CPPUNIT_TEST_SUITE( ExampleTestCase );
  CPPUNIT_TEST( example );
  CPPUNIT_TEST_SUITE_END();

protected:
  double m_value1;
  double m_value2;

public:
  void setUp();

protected:
  void example();
  void anotherExample();
  void testAdd();
  void testDivideByZero();
  void testEquals();
};

#endif
```

Kuva 17: AnyUnit-esimerkkikoodia.

6.2 C & C++

Aivan kuten .NET -tapauksessa, myös ohjelmointikielet C ja C++ ovat tarvinneet omat yksikkötestausympäristönsä helpottamaan testauslähtöistä ohjelmistokehitystä. Tarjontaa näille ohjelmointikielille löytyy ehkä vieläkin enemmän kuin .Net-ympäristöille.

6.2.1 CppUnit

Michael Feathers aloitti kohdassa 6.5.4 tarkasteltavan JUnitin kehittämisen C++-ympäristöön sopivaksi ja tuloksena on syntynyt CppUnit. Se on ilmainen LPGL (Lesser General Public License) -lisenssin alainen työkalu. CppUnitin avulla suoritettavat testit voidaan automatisoida ja niiden tuottamat tulokset saadaan XML- tai tekstimuotoisina tulosteina. CppUnitin yhteyteen on mahdollista lisätä graafinen käyttöliittymä, joka nopeuttaa testien suorittamista ja auttaa näkemään virheelliset kohdat helpommin. Kuvassa 18 on CppUnitin mukaisesti kirjoitettua esimerkkikoodia yksinkertaisen testin

muodossa.

```
class ComplexNumberTest : public CppUnit::TestCase {
public:
  ComplexNumberTest( std::string name ) : CppUnit::TestCase( name ) {}

  void runTest() {
    CPPUNIT_ASSERT( Complex (10, 1) == Complex (10, 1) );
    CPPUNIT_ASSERT( !(Complex (1, 1) == Complex (2, 2)) );
  }
};
```

Kuva 18: CppUnit-esimerkkikoodia.

6.2.2 EasyUnit

CppUnitin tapaan AesyUnit on myös LGPL-lisenssin alainen yksikkötestausympäristö, joka perustuu Feathersin kehittämälle CppUnitLite-ympäristölle. EasyUnit on tarkoitettu C++ ja EC++ (Embedded C++) -ohjelmointikielillä kirjoitetuille ohjelmistoille. EasyUnit on mahdollista asentaa joko Unix- tai Windows-ympäristöihin ja sitä voidaan käyttää komentokehotteelta tai graafisen työympäristön avulla. EasyUnitin perimmäisenä ideana on olla erittäin yksinkertainen ja helppokäyttöinen työkalu, jonka avulla pystytään ottamaan automaattinen yksikkötestaus nopeasti osaksi ohjelmistokehitystä. Kuvassa 19 on esitelty EasyUnit -esimerkkikoodia.

```
// StackTest.cpp

#include "easyunit/test.h"
#include "Stack.h"

using namespace easyunit;

TEST(Stack,size)
{
  Stack s;
  ASSERT_TRUE(s.size() == 1);
}

TEST(Stack,isEmpty)
{
  Stack s;
  ASSERT_TRUE(s.isEmpty());
}
```

Kuva 19: EasyUnit-esimerkkikoodia.

6.3 Tietokannat

Myös tietokannoille on olemassa omia testaustyökaluja, joita voidaan käyttää, kun kehitetään järjestelmiä testauslähtöisesti. Näiden tarkoitus on yleensä tietokannan funktioiden ja proseduurien testaamisessa. Joissain tapauksissa näitä testaustyökaluja käytetään myös tietokannan tilan muuttamiseen ennen muiden testien suorittamista.

6.3.1 UtPLSQL

UtPLSQL on Oraclen PL/SQL -kielellä toteuttavien järjestelmäkehittäjien avuksi luotu yksikkötestaustyökalu. Sen kehittäjänä on toiminut Steven Feuerstein, joka on ollut mukana kirjoittamassa monia kirjoja liittyen Oracleen ja PL/SQL -kieleen. UtPLSQLn tarkoituksena on edesauttaa pakettien, funktioiden sekä proseduurien automaattista testausta. Kuvassa 20 on esitelty UtPLSQL -esimerkkikoodia.

```
/*file ut_str.pkb */
CREATE OR REPLACE PACKAGE BODY ut_str
IS
   PROCEDURE ut_setup
   IS
   BEGIN
      NULL;
   END;

   PROCEDURE ut_teardown
   IS
   BEGIN
      NULL;
   END;

   -- For each program to test...
   PROCEDURE ut_betwn IS
   BEGIN
      utAssert.eq (
         'Typical Valid Usage',
         str.betwn ('this is a string', 3, 7),
         'is is'
         );

      utAssert.eq (
         'Test Negative Start',
         str.betwn ('this is a string', -3, 7),
         'ing'
         );

      utAssert.isNULL (
         'Start bigger than end',
         str.betwn ('this is a string', 3, 1)
         );
   END ut_betwn;

END ut_str;
/
```

Kuva 20: UtPLSQL-esimerkkikoodia.

6.3.2 SQLUnit

Java-ohjelmointikielellä toteutettu SQLUnit on tallennettujen proseduurien yksikkö- sekä regressiotestaustyökalu. Koska SQLUnit käyttää JBDC-rajapintaa tietokannan kommunikoinnin kanssa, pitäisi ainakin teoriassa olla mahdollista toteuttaa tallennetuille proseduureille yksikkötestejä sellaisille tietokannoille, jotka tarjoavat käyttöönsä JDBC-ajurin. JUnit-yksikkötestaustyökalua käytetään hyväksi SQLUnitin kanssa konvertoimaan XML-muotoiset testispesifikaatiot JDBC-kutsuiksi ja näiden tuloksia verrataan määriteltyihin tuloksiin. Järjestelmien kehittäjät testaavat yleensä kirjoittamaansa Java-koodia, mutta tietokantojen olioabstraktiot ja niiden välinen toiminta saattaa jäädä vähemmälle huomiolle. SQLUnitin avulla tämä arkkitehtuurikerros saadaan otettua mukaan testausprosessiin. Suuremmissa ohjelmistoprojekteissa tietokantoihin liittyvien tallennnettujen proseduurien ja varsinaisen Java-koodin ohjelmoijat voivat olla eri henkilöitä. Tämä mahdollistaa sen, että tietokantaohjelmoijien ei välttämättä tarvitse olla eksperttejä käyttämään hyväkseen JUnit -työkalua. Kuvassa 21 on esitelty SQLUnit -esimerkkikoodia.

```
<?xml version="1.0"?>
<!DOCTYPE sqlunit SYSTEM "file:sqlunit/lib/sqlunit.dtd" [
  <!ENTITY connection SYSTEM "file:sqlunitConnectionConfig.xml">
  <!ENTITY data SYSTEM "file:sqlunitTestData.xml">
]>

<sqlunit>

  &connection;

  <setup>
   &data;
  </setup>

  <test name="Looking up New Member created from Register page">
   <sql>
    <stmt>select Firstname,Surname from Client where UserID=?</stmt>
    <param id="1" type="VARCHAR" inout="in">${test.newuser.id}</param>
   </sql>
   <result>
    <resultset id="1">
       <row id="1">
         <col id="1" type="VARCHAR">Dummy</col>
         <col id="2" type="VARCHAR">User</col>
       </row>
    </resultset>
   </result>
  </test>

  <teardown>
   <sql>
    <stmt>delete from Client where UserID=?</stmt>
    <param id="1" type="INTEGER" inout="in">${test.newuser.id}</param>
   </sql>
  </teardown>

</sqlunit>
```

Kuva 21: SQLUnit-esimerkkikoodia.

6.4 GUI

Graafisten käyttöliittymien testaaminen, tai varsinkin sen automatisoiminen, nähdään yleisesti hankalana tehtävänä. Kattavan käyttöliittymän testaamiseen tarvitaan mukaan ihmisen tuomaa yllätyksellisyyttä ja tilanteista riippuvaista reagointikykyä. XP kuitenkin ehdottaa aloittamaan hyvin pienistä järjestelmän palasista ja kokoamaan tällä tavalla koko ajan kasvavaa hyväksymistestaustestijoukkoa, jota voidaan käyttää jopa graafisia käyttöliittymiä omaaviin sovelluksiin (McBreen, 2003). Apuna automatisoinnissa voidaan käyttää nykyään tarjolla olevia sovelluksia, jotka esimerkiksi nauhoittavat käyttäjän tekemiä operaatioita ja sen jälkeen nämä samat operaatiot voidaan toistaa aina haluttaessa uudelleen.

6.4.1 Selenium

Selenium on nauhoitustyökalu, joka on saatavilla muun muassa Mozilla Firefox-selaimen liitännäisosana. Seleniumin avulla voidaan nauhoittaa käyttäjän toimintaa selaimessa ja toistaa nämä toiminnot myöhemmin testien yhteydessä. Seleniumin käyttö ei vaadi mitään varsinaisia ohjelmointitaitoja, koska testitapaukset voidaan nauhoittaa suoraan käyttäjän toiminnoista, mutta kattavamman testauksen suorittamisessa olisi hyvä olla vähintään perustiedot HTML– ja JavaScript-kielistä. Seleniumille on myös toteutettu etäkäyttöjärjestelmä, joka mahdollistaa automaattisten testien kirjoittamisen www-selaimelle monilla eri ohjelmointikielellä, kuten Java, .NET, PHP ja Python. Kuvassa 22 on esitetty JUnitin ja Seleniumin avulla kirjoitettua testikoodia.

```
import com.thoughtworks.selenium.*;
import junit.framework.*;
public class GoogleTest extends TestCase {
    private Selenium browser;
    public void setUp() {
        browser = new DefaultSelenium("localhost",
            4444, "*firefox", "http://www.google.com");
        browser.start();
    }

    public void testGoogle() {
        browser.open("http://www.google.com/webhp?hl=en");
        browser.type("q", "hello world");
        browser.click("btnG");
        browser.waitForPageToLoad("5000");
        assertEquals("hello world - Google Search", browser.getTitle());
    }

    public void tearDown() {
        browser.stop();
    }
}
```

Kuva 22: Selenium- ja Junit-esimerkkikoodia.

6.4.2 Abbot

Abbot (A Better Bot) on ilmainen testaustyökalu Javalla toteutettuihin graafisen käyttöliittymän omaaviin sovelluksiin. Abbot-testausympäristö on laajennus JUnit-yksikkötestaustyökaluun ja sitä voidaan käyttää kirjoittamalla yksikkötestejä Java-ohjelmointikielellä tai XML-pohjaisilla skripteillä. Abbotin avulla voidaan nauhoittaa ja myöhemmin toistaa käyttäjän tekemiä toimenpiteitä kehitettävän sovelluksen käyttöliittymässä. Tällä tavoin saavutetaan enemmän luotettavuutta järjestelmän toimintaan, kun suoritettavat testit voidaan ajaa aina halutessa uudelleen. Ruiz ja Wang

Price (2007) ovat julkaisseet IEEE Softwaren sivuilta löytyvän esimerkin testauslähtöisestä ohjelmistokehityksestä kohdassa 6.5.3 tarkasteltavan TestNG:n ja Abbotin avulla kuvan 23 mukaisesti.

```
package example;

import java.io.File;

import junit.extensions.abbot.*;
import junit.framework.Test;
//import junit.textui.TestRunner;
//import junit.ui.TestRunner;
//import junit.swingui.TestRunner;

/** Simple example of a ScriptTestSuite.  Selects all scripts of the form
 * MyCode-[0-9]*.xml.
 */
public class MyCodeTest extends ScriptFixture {

    /** Name is the name of a script filename. */
    public MyCodeTest(String name) {
        super(name);
    }

    /** Return the set of scripts we want to run. */
    public static Test suite() {
        return new ScriptTestSuite(MyCodeTest.class, "src/example") {
            /** Determine whether the given script should be included. */
            public boolean accept(File file) {
                String name = file.getName();
                return name.startsWith("MyCode-")
                    && Character.isDigit(name.charAt(7))
                    && name.endsWith(".xml");
            }
        };
    }

    public static void main(String[] args) {
        TestHelper.runTests(args, MyCodeTest.class);
    }
}
```

Kuva 23: Abbot-esimerkkikoodia.

6.5 Java

Java on eräs yleisimmin tunnetuista ja käytetyistä ohjelmointikielistä ja sille onkin toteutettu todella monia erilaisia testiympäristöjä. Www.testdriven.com-sivustolla on esitelty näitä tarjolla olevia työkaluja ja testauslähtöisestä ohjelmistokehityksestä kiinnostuneen henkilön on esimerkiksi sieltä helppo valita mieleisensä apuväline ja lähteä tutustumaan tähän kehityssuuntaukseen.

6.5.1 Unitils

Unitils on yksi Javalla toteutetuista vapaan lähdekoodin yksikkötestaustyökaluista. Se rakentuu olemassa olevien kirjastojen päälle, kuten DBUnit, joka on JUnit-laajennus tietokantalähtöiseen testaukseen, ja EasyMock, joka mahdollistaa mallinnettujen olioiden käytön JUnit-testien yhteydessä, sekä integroituu kohdissa 6.5.3 ja 6.5.4 tarkasteltaviin TestNG- ja JUnit-testaustyövälineisiin. Unitils mahdollistaa yleisten testauspiirteiden lisäksi mallinnettujen olioiden käytön, Spring-ympäristössä tapahtuvan testaamisen ja tietokantojen testaamisen sekä Hibernate- ja JPA-integraation. Ainoat pakolliset ladattavat testausvälinekirjastot ovat JUnit ja TestNG. Carleton (2007) esittelee artikkelissaan Unitilsia yhdessä Eclipse-sovelluskehittimen kanssa käytettynä. Hän havainnollistaa esimerkin avulla kuinka Unitilsia käytetään hyväksi kehitettäessä HR -sovellusta, joka päivittää haluttaessa työntekijöiden palkat toivotulle tasolle. Artikkelissa esitetään tietovaraston alustaminen halutuilla tiedolla sekä sen käyttäminen testien yhteydessä. Kuvassa 24 on esitelty Unitils -esimerkkikoodia.

```
<dataset>
      <employees id="100" salary="54200.24" title="Worker        Bee"/>
      <employees salary="70444.88"/>
      <employees salary="30200.55"/>
      <employees id="103" salary="40777.11" title="Worker        Bee"/>
      <employees salary="82100"/>
</dataset>

@Test
@DataSet
@ExpectedDataSet
public void savesSalaryChanges()
      throws DataAccessException
{
      List<Employee> raiseEmployees
                  = employeeDao.findEmployeesByTitle("Worker
Bee");
      raiseEmployees.get(0).setSalary(new
BigDecimal(58000.00));
      raiseEmployees.get(1).setSalary(new
BigDecimal(61000.00));
      employeeDao.saveEmployees(raiseEmployees);
}
```

Kuva 24: Unitils-esimerkkikoodia.

Käytetyt tietovarastot voidaan tapauksesta riippuen haluttaessa tyhjentää testien välissä, jolloin tiedetään, että mitään vanhaa tietoa ei pääse mukaan korruptoimaan muita testejä.

6.5.2 jMock

Yksikkötestien tarkoituksena on testata tiettyä järjestelmän osaa, joka on eristettynä muista osista. On kuitenkin hyvin yleistä, että testin alainen järjestelmän osa tarvitsee oikein toimiakseen jotain toista osaa järjestelmästä. Tällöin on mahdollista mallintaa tarvittavaa osaa ilman varsinaisen tuotantokoodin toteuttamista. Mallinnetut oliot tarjoavat tätä tarkoitusta varten työkalun. jMock on Javalle toteutettu yksikkötestausympäristö, joka tukee mallinnettujen olioiden käyttöä.

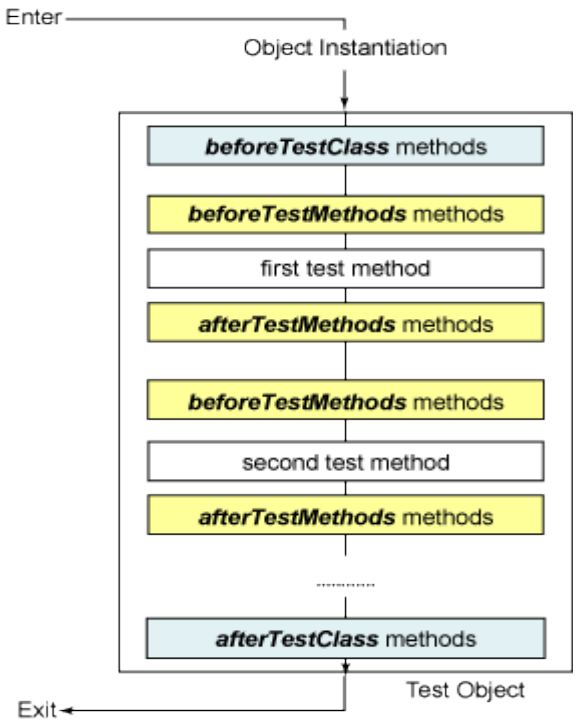
MacKinnon & al. (2007) selvittävät julkaisussaan mallinnettujen olioiden merkitystä ja käyttötarkoitusta. Testauslähtöisesti kehitettäessä ei ole tarkoitus sitoutua perimmäisiin rakenteisiin ennen kuin on aivan pakko. Tietokanta toimii tässä tapauksessa havainnollistavana esimerkkinä. Mallinnettujen olioiden avulla voidaan mallintaa tietokannan toimintaa ilman varsinaista lopullista toteutusta. Riippuen projektin antamista reunaehdoista, sovituista toteutusvälineistä sekä teknologiavalinnoista, tämä antaa vapauden vielä viime kädessä vaihtaa tietokannan toteutuksen. Se myös antaa toteuttajalle varmuuden siitä mitä oikein ollaan hakemassa tietokannan toiminnallisuuden osalta. Toinen esimerkki on publish/subscribe -viestijärjestelmä, jossa halutaan testata publisher-luokan viestin lähetystä, joka tarvitsee toimiakseen subscriber-luokan olion. Tässä tapauksessa mallinnetaan subscribe-olion toimintaa mallinnetun olion avulla. Kuvassa 25 on esitelty jMock-esimerkkikoodia työkalun kotisisuilta.

```
import org.jmock.Mockery;
import org.jmock.Expectations;

class PublisherTest extends TestCase {
    Mockery context = new Mockery();

    public void testOneSubscriberReceivesAMessage() {
        // set up
        final Subscriber subscriber = context.mock(Subscriber.class);
        Publisher publisher = new Publisher();
        publisher.add(subscriber);
        final String message = "message";
        // expectations
        context.checking(new Expectations() {{
            one (subscriber).receive(message);
        }});
        // execute
        publisher.publish(message);
         // verify
        context.assertIsSatisfied();
    }
}
```

Kuva 25: jMock-esimerkkikoodia.

6.5.3 TestNG

TestNG on Cedric Beustin luoma avoimen lähdekoodin yksikkötestausympäristö Javalla toteutetuille ohjelmistoille. Se pohjautuu vahvasti annotaatioihin. Annotaatioiden avulla määritellään kirjoitettavaan koodiin, mitä mikäkin osa tarkoittaa ja kuinka sitä on tarkoitus suorittaa. Esimerkkejä käytettävistä annotaatiosta ovat muun muassa @BeforeSuite, @Aftersuite, @BeforeTest, @AfterTest, @BeforeGroups, @AfterGroups, @BeforeClass, @AfterClass, @BeforeMethod ja @AfterMethod. Metodit, joille on määritelty jokin edellä mainituista annotaatioista, suoritetaan joka kerta testien suorituksen yhteydessä joko ennen annotaation määrittelemää tapahtumaa tai sen jälkeen. @Test-annotaatio määrittelee luokan tai testin osaksi testiä. Kuvassa 26 on esitelty TestNG -metodijoukot.



Kuva 26: TestNG-metodijoukot.

```
public class Test1 {
   @Test(groups = { "functest", "checkintest" })
   public void testMethod1() {
   }
   @Test(groups = {"functest", "checkintest"} )
   public void testMethod2() {
   }
   @Test(groups = { "functest" })
   public void testMethod3() {
   }
}
```

Kuva 27: TestNG esimerkkikoodia.

Kuvan 27 esimerkissä Test1-luokalla on kolme metodia: testMethod1, testMethod2 ja testMethod3. Nämä kaikki on määritelty testeissä suoritettaviksi metodeiksi @Test-annotaation avulla. @Test-annotaation jälkeen on vielä määritelty groups-määreellä mihin ryhmiin kyseiset testimetodit kuuluvat. TestMethod1 ja testMethod2 kuuluvat functest- ja checkintest-ryhmiin, kun taas testMethod3 kuuluu ainoastaan functest-ryhmään. Testien kirjoittaminen aloitetaan tyypillisesti sillä, että ensin kirjoitetaan testien toimintalogiikka ja sen jälkeen kerrotaan annotaatioiden avulla, millä tavalla kyseistä logiikka on tarkoitus suorittaa testien yhteydessä. Jotta edellä olevan esimerkin

mukaiset testit voidaan suorittaa, täytyy sitä varten kirjoittaa testng.xml-tiedosto, jossa määritellään mitä testejä halutaan kerralla ajoon. Toinen vaihtoehto on kirjoittaa build.xml-tiedosto, joka on Apache Ant Java build -työkalun käyttämä määrittely-tiedosto.

```
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd" >

<suite name="Suite1"  verbose="1" >
    <test name="Test1">
        <groups>
            <run>
                <include name="functest"/>
            </run>
        </groups>
        <classes>
            <class name="example1.Test1"/>
        </classes>
    </test>
</suite>
```

Kuva 28: testng.xml-tiedosto.

Testng.xml-tiedostoon (kuva 28) määritellään esimerkin mukaisesti suite-tagien sisälle suoritettavat testit. Näitä testejä voi olla yksi tai useampi. Jokaisella testillä on nimi, jolla testi tunnistetaan, sekä luokka, jossa testi tai testit sijaitsevat. Kuvan 27 Java-luokassa kirjoitetut testimetodit kuuluvat erilaisiin ryhmiin. Testng.xml-tiedostossa voidaan suorittaa testit vain niille metodeille, jotka kuuluvat haluttuihin ryhmiin. Kuvan 24 mukaisesti testit suoritettaisiin niille metodeille, jotka kuuluvat luokassa määritellyn annotaation mukaan functest-ryhmään. Jos yllä olevista esimerkeistä haluttaisiin suorittaa vain testMethod1 ja testMethod2, testng.xml-tiedostoon määriteltäisiin ajettavan ryhmän nimeksi checkintest.

TestNG on tarkoitettu työkaluksi, jota voidaan käyttää hyväksi myös suuremmissakin projekteissa, joissa on tarkoitus järjestää testaus yhden työkalun avulla yksikkötestauksesta integraatiotestaukseen. Glover (2006) esittelee artikkelissaan miksi TestNG on parempi työkalu suuremman skaalan testaukseen kuin JUnit.

6.5.4 JUnit

JUnit on ehkä yleisin ja yksi tunnetuimmista Javalle toteutetuista yksikkötestausympäristöistä. Sen kehittäjinä ovat toimineet Kent Beck sekä Erich Gamma. JUnit on kehittynyt Smalltalkille tehdyn Sunit-yksikkötestausympäristön jatkeeksi. Muita xUnit-perheen ympäristöjä ovat muun muassa PHP:lle tehty PHPUnit, Pythonille PyUnit, Fortranille fUnit, Delphille Dunit sekä JavaScriptille JsUnit.

Beck (2004) kuvaa kirjassaan JUnitin käytön hyviä puolia, kun kirjoitetaan usein toistettavia testejä:

- Testit on mahdollista suorittaa automaattisesti.
- Useita testejä pystytään suorittamaan yhdessä ja näiden tuloksista saadaan yhteenvedot.
- JUnit tarjoaa kätevän paikan kirjoitettujen testien säilyttämiseen.
- JUnitin avulla saadaan kätevästi paikka, jossa voidaan jakaa testien suorittamiseen vaadittavaa koodia.
- JUnitin avulla voidaan verrata odotettuja ja todellisia tuloksia sekä raportoida niiden eroista.

Nämä Beckin kuvaamat hyvät puolet eivät rajoitu pelkästään JUnitiin. Tässä tutkielmassa esitellyt muutkin työkalut toimivat suurimmaksi osaksi niiden mukaan.

Beck (2004) kirjoittaa myös JUnitin päämääristä sekä siitä millaisia testien tulisi olla varsinaisten ohjelmistojen kehittäjien kannalta, koska testauslähtöisen ajattelutavan mukaan juuri he kirjoittavat testit, joiden perusteella voidaan havaita suurin osa virheistä:

- Testien tulee olla helposti kirjoitettavia.
- Testien kirjoittamisen tulee olla helposti opittavaa.
- Testien suorittamisen tulee olla nopeaa.
- Testien suorittamisen tulee olla helppoa.
- Testit tulee olla eristettävissä ulkopuolisilta vaikutteilta.
- Testit tulee olla suoritettavissa millä tahansa joukolla.

Testien kirjoittamisen oppiminen sekä niiden helppo kirjoittaminen on kiinni yleensä henkilöstä, joka alkaa työskentelemään uuden työkalun sekä ohjelmointikielen kanssa. Toiset omaksuvat uutta tietoa huomattavasti nopeammin kuin toiset, mutta Javalla ohjelmakoodia kirjoittaneen henkilön ei pitäisi olla suurissa vaikeuksissa JUnitin kanssa. Testien nopeassa ja helpossa suorittamisessa auttaa huomattavasti niiden automatisoimisen mahdollisuus. Jokainen kirjoitettu testi pystytään suorittamaan haluttuna ajankohtana automaattisesti. Tällä tavalla suoritetaan ohjelmiston kehityksen aikana myös huomaamatta taantumatestausta. JUnitin avulla testit voidaan suorittaa komentokehotteesta tai vieläkin helpommin esimerkiksi Eclipse-sovelluskehittimeen saatavilla olevasta liitännäisosasta. Tämä auttaa kehittäjiä näkemään myös visuaalisesti mitkä testit on suoritettu hyväksytysti ja mitkä testit ovat epäonnistuneet. JUnit tarjoaa myös mahdollisuuden suorittaa testit eristettyinä kokonaisuuksina. Vaatimuksena sille on kuitenkin ennen testejä suoritettavat alustukset sekä suoritusten jälkeiset puhdistustoimenpiteet, joita voisi olla muun muassa tietokannan tyhjentäminen testien vaatimista tiedoista. JUnit esimerkkikoodia on esitelty kohdan 2.4 esimerkin yhteydessä.

7 Yhteenveto

Testauslähtöinen ohjelmistokehitys antaa varmasti monille kehittäjille uudenlaisen näkökulman tarkastella ohjelmistokehitysprosessia. Perinteisesti aiemmin ollaan ensin tehty varsinainen ohjelmakoodi ja vasta sen jälkeen ollaan mietitty testejä, joita sen pitäisi läpäistä. Perinteisessä tyylissä on omat vaaransa. Silloin on mahdollista, että testit tehdään tiedostamatta vain ja ainoastaan läpäisemään jo kirjoitettu ohjelmakoodi. On mahdollista, että harvinaisemmat polut ohjelmiston sisällä jäävät täysin ilman huomiota. Myös testauslähtöinen kehitys pitää sisällään vaarallisen mahdollisuuden. Jos kehittäjät eivät suunnittele testejä tarpeeksi hyvin, voi tuloksena olla rakenteellisesti hyvinkin alkeellinen järjestelmä, joka ei palvele tarkoitustaan. Tässä tapauksessa ohjelmakoodi on kirjoitettu läpäisemään vain hyvin yksinkertaiset testitapaukset.

Sanonta kuuluu, että vanha koira ei opi uusia temppuja. Testauslähtöisen ohjelmistokehityksen yksi suurimmista muutosvastarinnan kohteista voi olla hyvinkin ohjelmistokehittäjät. Yli kymmenen vuotta alalla työskennellyt henkilö, joka on joutunut opiskelemaan useita eri ohjelmointikieliä, ei välttämättä enää halua oppia uutta tapaa tehdä työtään. Otollisin maaperä testauslähtöisen kehityksen omaksumiseen voisikin olla uudet, alasta kiinnostuneet henkilöt, joilla ei vielä ole välttämättä mitään ennakkoluuloja uusia kehitysprosesseja kohtaan.

Suurilla yrityksillä voi olla varaa kokeilla testauslähtöistä kehitystä joissain projekteissa ilman minkäänlaisia tulostavoitteita, mutta pienempien yritysten on hankala lähteä kokeilulinjalle ilman takuita siitä onnistuuko projekti. Yritysjohdon olisi kuitenkin hyvä nähdä testauslähtöisessä kehityksessä mahdollisuus, jolla voidaan parantaa projektien lopputulosten laatua. Täysin virheettömään tulokseen on mahdotonta päästä mutta mikään ei estä yrittämästä.

Testauslähtöisen ohjelmistokehityksen kannalta tärkeimpänä asiana nousee esille testien automatisoiminen. Automatisoinnin ja testien nopean suorittamisen avulla saavutetaan juuri sellainen rytmi ohjelmakoodin kirjoituksessa, mikä on tarkoitus säilyttää läpi ohjelmistonkehitysprosessin. Ilman automaatiota ja uudelleensuoritettavuutta ei voida puhua testauslähtöisestä kehityksestä, koska kaikki ohjelmiston kehityksen aikana

kirjoitetut testit, joita voi olla satoja tai jopa tuhansia, tulee olla suoritettavissa melkein milloin vain.

Testauslähtöinen kehitys vaatii toteuttajaltaan sitoutumista sen sanelemiin sääntöihin. Ensin kirjoitetaan testit ja vasta sen jälkeen varsinaista ohjelmakoodia. TDD korostaa myös nopeaa sykliä testien kirjoittamisen ja sitä vastaavan ohjelmakoodin välillä. Myös pariohjelmointi on yksi esillä olevista asioista. Nämä kaikki tiukat säännöt eivät välttämättä sovi kaikille, jonka takia tulisikin enemmän ottaa esille hyötyjä testauslähtöisestä kehityksestä, kuin tyylejä ja malleja jotka noudattavat näitä sääntöjä.

TDD ei ole mikään ainoa oikea tapa tehdä asioita ja viedä projekti läpi. Se on yksi kehitystyyli muiden prosessimallien rinnalla. Toisenlaiset mallit soveltuvat paremmin toisiin projekteihin. Kaikista löytyy hyviä ja huonoja puolia, joita pitää punnita projektiin sopivimman kehitystyylin valinnassa.

Viitteet

Astels, D., Miller, G., Novak, M. (2002) *A Practical Guide to Extreme Programming*, Prentice Hall PTR, Upper Saddle River, NJ.

Beck, K. (2000) *Extreme Programming Explained*, Addison-Wesley, Boston, MA.

Beck, K. (2003) *Test-Driven Development By Example*, Addison-Wesley, Boston, MA.

Beck, K. (2004) *JUnit Pocket Guide*, O'Reilly, Sebastopol, CA.

Black, R. (2007) *Pragmatic Software Testing Becoming an Effective and Efficient Test Professional*, Wiley, Indianapolis, IN.

Boehm, B (1988) *A Spiral Model of Software Development and Enhancement*, http://www.computer.org/portal/cms_docs_computer/computer/homepage/misc/Boehm/r5061.pdf (2.2.2008).

Carleton, D. (2007) *Efficient Test-Driven Design with Unitils*, http://www.devx.com/Java/Article/35129/0/page/1 (29.12.2007).

Duvall, P., Matyas, S., Glover, A. (2007) *Continuous Integration: Improving Software Quality and Reducing Risk*, Addison-Wesley, Boston, MA.

Feathers, M. (2001) *The 'Self'-Shunt Unit Testing Pattern*, http://www.objectmentor.com/resources/articles/SelfShunPtrn.pdf (1.12.2007).

Fowler, M. (1999) *Refactoring Improving the Design of Existing Code*, Addison-Wesley, Westford, MA.

Glover, A. (2006) *In pursuit of a code quality: Junit 4 vs. TestNG*, http://www-128.ibm.com/developerworks/java/library/j-cq08296/index.html (2.2.2008).

Koskela, L. (2008) *Test Driven TDD and Acceptance TDD for Java Developers*, Manning, Greenwich, CT.

MacKinnon, T., Freeman, S., Craig, P. (2000) *Endo-Testing: Unit Testing with Mock Objects*, http://www.connextra.com/aboutUs/mockobjects.pdf (23.1.2008).

McBreen, B. (2003) *Questioning Extreme Programming*, Addison-Wesley, Boston, MA.

Meszaros, G. (2007) *xUnit Test Patterns: Refactoring Test Code*, Addison-Wesley, Boston, MA.

Ruiz, A., Wang Price, Y. (2007) *Test-Driven GUI Development with TestNG and Abbot*, http://csdl2.computer.org/comp/mags/so/2007/03/s3051_Web_Extra_2007.pdf (3.1.2008).

Stott, W., Newkirk, J (2004) *Improve the Design and Flexibility of Your Project with Extreme Programming Techniques*. WWW-sivusto, http://msdn2.microsoft.com/fi-fi/magazine/cc163982(en-us).aspx (3.3.2008).

Wake, W. (2001) *Games for Programmers*, http://xp123.com/g4p/ (3.4.2008).