hyväksymispäivä arvosana

arvostelija

Testivetoinen web-sovelluskehitys

Ville Karjalainen

Helsinki 20.3.2012
Pro gradu -seminaariesitelmän kirjallinen alustus
HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta – Fakultet –Faculty	Laitos – Institution – Department
Matemaattis-luonnontieteellinen tiedekunta	Tietojenkäsittelytieteen laitos

Tekijä – Författare – Author

Ville Karjalainen

Työn nimi – Arbetets titel – Title

Testivetoinen web-sovelluskehitys

Oppiaine – Läroämne – Subject

Tietojenkäsittelytiede

Työn laji – Arbetets art – Level	Aika – Datum – Month and year	Sivumäärä – Sidoantal – Number of pages
	20.3.2012	19 sivua

Tiivistelmä – Referat – Abstract

Avainsanat – Nyckelord – Keywords

web-sovellus, TDD, ATDD, London XP, XP, ketterät menetelmät, Spring, Java

Säilytyspaikka – Förvaringställe – Where deposited

Muita tietoja – Övriga uppgifter – Additional information

Sisältö

1 Johdanto

Web-sovelluksilla on keskeinen rooli yhteiskunnassa. Esimerkiksi monet tärkeät palvelut, kuten pankki- ja matkavarauspalvelut, ovat siirtyneet verkkoon. Samalla osa samoista palveluista on rapautunut perinteisissä muodoissaan, mikä on viimeistään lähes pakottanut ihmiset verkkoon. Vuosina 2006-2011 Internetin käyttäjien määrä maailmassa yli kaksinkertaistui, nousten 2.3 miljardiin [Int11]. Kehittyneissä maissa, kuten Suomessa, kasvu on nopeaa mobiililaitteiden kohdalla: vuosina 2009-2011 verkon käytön yleisyys matkapuhelimella laajakaistaverkossa yli kolminkertaistui Suomessa [Til11]. Käyttäjämäärien nopea kasvu sekä laitteiden nopea kehittyminen ja monipuolistuminen tekevät web-sovellusten kehittämisestä ja tutkimisesta yhä tärkeämpää ja monipuolisempaa.

XP [Bec99] (extreme programming) on ketterä menetelmä (agile method), mitä on käytetty paljon web-sovellusten kehittämiseen [TWL10]. Ketterät menetelmät sopivat hyvin web-sovellusten kehitykseen niiden hyvän muutoksiin reagoimiskyvyn ansiosta. Kyky reagoida tehokkaasti muutoksiin on XP:n määritelmässä keskeisessä osassa [Bec99], ja jatkuva muutos on web-sovelluksille ominainen piirre [RPG10].

XP:n tärkeä osa, TDD [Bec02] (test driven development), on ketterä käytänne (agile practice) ja keskeinen tekijä korkean muutoksiin reagointikyvyn taustalla [FrP09]. TDD:ssä testikoodi kirjoitetaan aina ennen tuotantokoodia yksikkötestaustasolla (unit testing) [Kos08]. Sykli pidetään erittäin nopeana jolloin ohjelma kasvaa vain pienin askelin. Tämä kurinalaisuus johtaa siihen, että koodi ei ole koskaan kaukana tilasta, mistä se voidaan toimittaa asiakkaalle. Tämä on merkittävä apu nopean toimitussyklin sekä inkrementaalisen ja iteratiivisen kehitysprosessin ylläpitämisessä. Lisäksi testin kirjoittaminen ennen koodia antaa testaamiselle uuden ulottuvuuden muuttaen testaamisen suunnitteluksi [FrP09 s.5]. Tavoitteena on tuottaa mahdollisimman laadukasta koodia, mikä helpottaa muutoksien tekoa jatkossa.

Testivetoisuuden ideaa on sovellettu myös yksikkötestausta korkeammilla tasoilla. Esimerkiksi ATTD:ssä [Bec02] (acceptance TDD) tuotetaan kaikki tarpeelliset vaatimukset kattava hyväksymistestijoukko (acceptance test) tiiviissä yhteistyössä asiakkaan kanssa [Kos08]. Toinen esimerkki on Lontoon XP-yhteisöstä kasvanut TDD:n erillinen kehityshaara [FrP09], LonTDD, missä testauksen taso vaihtelee yksikkötestauksen ja hyväksymistestauksen välillä. LonTDD:ssä uuden vaatimuksen toteuttaminen aloitetaan aina koko systeemin läpäisevän (end-to-end) testin kirjoittamisella. Ideaalitilanteessa läpäisytesti vastaa hyväksymistestiä [FrP09]. Vaatimuksen myöhemmässä kehitysvaiheessa palataan hyväksymistestaustasolta yksikkötestaustasolle ja jatketaan TDD:n mukaisesti. Kolmantena esimerkkinä web-sovellusten kehittämiseen on räätälöyty oma TDD-versionsa, Web-TDD, missä luodaan myös hyväksymistestejä, mutta painotus on erityisesti käytettävyyden suunnittelussa [RPG10].

Ketterät menetelmät nousivat it-yritysten pääasiallisiksi kehitysmenetelmiksi hiukan ennen vuotta 2010 ja menetelmien suosio näyttää kasvavan myös tulevaisuudessa [SLD07, WGG10, LSA11]. Koska TDD on yksi XP:n ydinkäytänteitä ja sitä voidaan käyttää myös kaikkien muiden menetelmien yhteydessä, voi olettaa myös TDD:n käytön olevan laajaa. Samalla kuitenkin empiirinen todistusaineisto TDD:n eduista puuttuu edelleen lähes täysin [CSP11]. TDD on usein otettu yrityksissä käyttöön ilman, että sitä on kunnolla tunnettu tai osattu käyttää. Tällöin sen vaikutukset ohjelmistoihin ovat yleensä olleet haitallisia. TDD:hen on ajan myötä kohdistunut myös yhä enemmän kritiikkiä [Kol11]. Tarve TDD:n lisätutkimukselle näyttää voimistuvan.

Tämä tutkimus käsittelee web-sovellusten kehittämistä testivetoisesti. Tutkimuksen aikana kehitetään uutta web-sovellusta käyttäen hyväksi TDD:n, LonTDD:n, ATTD:n ja Web-TDD:n ominaisuuksia. Web-sovellus kehitetään Javalla käyttäen apuna Spring sovelluskehystä [Spr12]. Spring on erittäin suosittu sovelluskehys Java-kehityksessä ja sen käyttäjien joukon koon nyt ja lähitulevaisuudessa on arvioitu olevan kaksikolmasosaa kaikista Java-kehittäjistä [Vmw12]. Spring on myös suunniteltu erityisesti testausta tukevaksi ja sen testausominaisuuksia myös kehitetään aktiivisesti [BrS11]. Tämä tutkimus suhteutetaan aiemmin TDD:hen kohdistuneeseen tutkimukseen

[CSP11, Kol11, BLM11]. Tässä tutkimuksessa pyritään löytämään yhteneväisyydet ja erot TDD:n aiemmissa tutkimuksissa havaittuihin TDD:n vaikutuksiin ja haasteisiin. Tätä kautta voidaan mahdollisesti nähdä uusimpien testivetoisten käytänteiden sekä jatkuvasti kehittyvän Springin testausominaisuuksien vaikutus testivetoisuuden toimivuuteen. Tutkimus on luonteeltaan laadullista. Tämä tutkimus on osaltaan vastaus esitettyyn tarpeeseen saada lisää tutkimustuloksia TDD:n vaikutuksista ohjelmistokehitykseen [CSP11, Kol11, BLM11].

Tutkimuksen rakenne on seuraavanlainen. Luvussa 2 esitellään testivetoiset käytänteet TDD, ATDD, LonTDD ja Web-TDD tarkemmin. Lisäksi käydään läpi aiemmin tehtyä TDD-tutkimusta ja kritiikkiä. Luvussa 3 käydään läpi kehitettävän web-sovelluksen kannalta merkittävät työkalut. Luvussa 4 käydään läpi web-sovelluksiin liittyvää testausteoriaa sekä yleisellä ja korkealla tasolla kehittävän web-sovelluksen osat. Luvussa 5 esitellään lyhyesti kehitettävä web-osovellus. Luvussa 6 tutkitaan web-sovelluksen testivetoista kehitystä. Luvussa 7 analysoidaan tulokset. Luvussa 8 on yhteenvedon aika.

2 Testivetoinen kehitys

Testivetoisessa kehityksessä testi luodaan ennen testin kohdetta. Testivetoisuuden ideaa on sovellettu monella eri tasolla koodin yksikkötestauksesta vaatimusmäärittelyyn. TDD on testivetoinen käytänne, mikä keskittyy yksikkötestaustasolle, jolloin testattava kohde on esimerkiksi Java-ohjelmointikielessä yksi luokka. ATDD:ssä yksi testi on hyväksymistesti ja vastaa asiakastason vaatimusta. Tällöin testi usein läpäisee koko systeemin päästä päähän eli esimerkiksi käyttöliittymästä tietokantaan. LonTDD:ssä uuden vaatimuksen kehitys aloitetaan aina läpäisytestillä ja sen mahdollisimman nopealla ja yksinkertaisella toteutuksella, jonka jälkeen siirrytään myöhemmin yksikkötestaustasolle. TDD:stä on kehitelty myös erityisesti web-sovelluksien kehittämistä varten räätälöityä versiota. Testivetoisuuden ja erityisesti TDD:n vaikutuksista ohjelmistojen laatuun on tehty paljon tutkimusta, mutta se on edelleen kovin puutteellista. Empiirinen tutkimustieto puuttuu lähes kokonaan.

2.1 TDD

TDD on testivetoinen käytänne, mikä keskittyy yksikkötestaustasolle. TDD syntyi XP:n sisällä, mutta sitä voidaan käyttää vapaasti myös muiden kehityskäytäntöjen yhteydessä, kuten esimerkiksi Scrumin [ScB01] ja vesiputousmallin yhteydessä. TDD on ensisijaisesti suunnittelukäytänne [FrP09]. Sen oheistuotteena syntyy kuitenkin jokaista yksikköä vasten testit. TDD tunnetaan monella nimellä, kuten testivetoinen suunnittelu (test driven design), testivetoinen kehitys (test driven development) sekä testaa ensin ohjelmointi (test first programming) [Kos08]. TDD:n kohdalla termiviidakko on kuitenkin ymmärrettävää TDD:n monipuolisuuden takia. Käytännössä TDD on kokonaisvaltainen vankkoja ohjelmointi-, suunnittelu- ja testaustaitoja vaativa menetelmä [FrP09].

2.1.1 TDD:n taustaa

TDD:n on yksi XP:n kekeisimmistä osista. XP on ketterien käytänteiden joukko, jonka muita osia ovat muun muassa pariohjelmointi (pair programming) ja jatkuva integraatio (continuous integration). XP:n kehittivät Kent Beck, Ron Jeffries ja Ward Cunningham vuonna 1996 tarpeeseen kehittää ohjelmistoja nopeammin ja laadukaammin [Sho07, Kos08].

Ohjelmiston sisäinen laatu ja kehitystyön nopeus ovat vahvasti yhteydessä toisiinsa: jos ohjelmiston koodi on kaikin puolin laadukasta on yksi olennainen tekijä ohjelmiston nopean kehittämisen kannalta kunnossa. Toisaalta laadukkaan koodin kirjoittaminen ja koodin kunnossapito vie yleensä enemmän aikaa kuin kaikkein helpoimman ratkaisun ohjelmoiminen. TDD:n taustalla on kuitenkin ymmärrys siitä, että koodin laatuun panostaminen jokaisen vaatimuksen kehittämisen aikana ei ole pitkällä tähtäimellä ongelma, vaan paljon laajemman ongelman ratkaisu.

Valtaosa kehittäjien työajasta kuluu koodin lukemiseen [FrP09]. Tutkimusten mukaan keskimäärin 40-70 prosenttia [Spi03]. Esimerkiksi Microsoftin vastavalmistuineilla kehittäjillä kului ensimmäinen vuosi lähes pelkästään vanhan koodin lukemiseen [BeS08]. TDD:n pyrkimys laadukkaan ja samalla luettavan koodin tuottamiseen

vaikuttaa tähän yhteen merkittävimmistä ongelmakohdista ohjelmistokehityksessä.

TDD etenee kurinalaisesti kolmen vaiheen syklissä: testi, koodi ja refaktorointi [Fow99] [Kos08]. Syklit täytyy pitää nopeina, paitsi jos koodin laadun takaaminen vaatii ison refaktoroinnin. Refaktoroinnin tarkoituksena on parantaa ohjelman osan sisäistä rakennetta niin, että sen ulkoinen käyttäytyminen ei muutu [Fow99]. TDD on suunnittelukäytänne ja se pakottaa suunnittelemaan jokaisen ohjelman yksikön ja tekee koodista samalla testattavaa. Koska TDD:ssä suunnittelun oheistuotteena syntyy aina testi, TDD pakottaa myös testaamaan kaikki ohjelman yksiköt. Kaikki tämä parantaa koodin laatua ja rohkeutta tehdä ohjelmaan positiivisia muutoksia [Kos08].

Yksi XP:n keskeisimpiä tavoitteita on parantaa kykyä vastata muuttuviin vaatimuksiin [Bec99]. Muuttuvat vaatimukset ja jokaisen projektin yksilöllisyys ja siitä nousevat yllätykset ovat merkittävä haaste ohjelmistojen kehityksessä [Kos08, FrP09]. Ohjelmiston sisäinen laatu on tärkeää muutoskykyisyyden säilyttämisen kannalta [FrP09 s.5]. Toinen tärkeä tekijä on muutosten mahdollisimman aikainen havaitseminen. Yksi XP:n keskeisimpiä keinoja muutoksiin varautumisessa on luoda tehokkaat puitteet palautteen saamiseksi koodistatasolta asiakasrajapintaan [Bec99]. Palautteen saaminen tukee ratkaisevasti muutostarpeiden havaitsemista ja mitä nopeammin muutostarve havaitaan, sen helpompaa siihen on reagoida.

TDD on XP:n keskeisimpiä osia iteratiivisen ja inkrementaalisen palauteinfrastruktuurin rakentamisessa. TDD:n pienin inkrementein etenevä tiivis sykli aiheuttaa sen, että koodi ei voi olla koskaan kovin kaukana siitä, että ohjelma voidaan käynnistää ja ajaa. Laadukas, refaktoroitu, testattu ja kääntyvä koodi on aina hyvässä valmiudessa asiakastoimitusta ajatellen, mitä kautta saadaan arvokkainta palautetta ohjelmistosta. TDD auttaa myös kooditason palautteen keräämisessä. Testejä kuuntelemalla voidaan saada viitteitä ohjelmiston laadusta. Jos testiä on vaikea kirjoittaa, on ongelma usein koodissa tai koko ongelmakenttää (domain) ei ymmärretä vielä tarpeeksi hyvin [FrP09].

Vuosien 2007-2010 välillä ketterien menetelmien osuus yritysten pääasiallisina kehitysmenetelminä kaksinkertaistui ja ne nousivat suosituimmaksi tavaksi kehittää ohjelmistoja 35 prosentin osuudella [SLD07, WGG10]. Ketterien menetelmien suosio näyttää olevan vahva myös tulevaisuudessa [LSA11]. On todennäköistä, että samalla myös TDD:n suosio on noussut. Samaan aikaan on havaittu, että TDD:tä käytetään usein väärin ja sitä on hankala käyttää. Toisaalta odotukset TDD:tä kohtaan ovat yleensä suuret, mikä on johtanut siihen, että TDD:n on havaittu olevan kaikkein eniten tyytymättömyyttä aiheuttava käytänne [CSP11]. Myös TDD:n oppimiskäyrä on korkea. Erään tutkimuksen mukaan eräällä ohjelmistokehitysryhmällä kesti lähes vuoden ennen kuin TDD saatiin toimimaan toivotulla tavalla kehitysprosessissa [BLM11].

Jotkut esitykset TDD:stä antavat ymmärtää TDD:n korvaavan kaikki entiset ohjelmistokehitysmenetelmät. TDD toimii kuitenkin parhaiten silloin kun sen käyttö pohjautuu mahdollisimman laajan kokemuksen tuomiin taitoihin ja arviointikykyyn [FrP09]. Käytännössä TDD on kokonaisvaltainen vankkoja ohjelmointi-, suunnittelu- ja testaustaitoja vaativa menetelmä [FrP09].

2.1.2 TDD-sykli

TDD:ssä perusideana on kirjoittaa testi aina ennen testauksen kohdetta. Tarkemmin TDD:n perusidea on kirjoitettu esimerkiksi seuraaviin kolmeen ”lakiin” [Mar09 s.122]:

1.Koodia ei saa kirjoittaa ennen kuin sitä vastaan on kirjoitettu testi.

2.Testin kirjoittaminen täytyy lopettaa heti, kun se ei mene enään läpi.

3.Koodin kirjoittaminen täytyy lopettaa heti, kun testi menee läpi.

Nämä lait lukitsevat kehittäjän noin 30 sekunnin sykliin, missä testiä ja koodia kirjoitetaan yhdessä niin, että testit kirjoitetaan aina vähän ennen koodia [Mar09 s.123]. Syklin pituudesta on annettu myös jonkin verran väljempää ohjeistusta 30 sekunnin ja parin minuutin välillä [Kos08].

Kolmantena osana TDD-sykliin kuuluu refaktorointi. Refaktorointia tehdään aina koodin kirjoittamisen jälkeen, jos siihen on tarvetta. TDD:ssä refaktorointi on työkalu, millä koodi pidetään aina laadukkaana. Toisin kuin testien ja koodin kirjoittamista, refaktorointeja ei tarvitse tehdä 30 sekunnissa. TDD:n yksi tärkeimmistä päämääristä on tuottaa koodia mikä on aina laadukasta. Tähän päämäärään päästäkseen on refaktoroitava usein ja joskus tämä voi viedä aikaa, eikä tiukkaa aikarajaa TDD:n puolesta määrätä [Kos08].

Edellä kuvattu tiivis TDD-sykli ja jokainen sen osa vaatii kehittäjiltä paljon kurinalaisuutta. Kurinalaisuuden tarvetta lisää vielä se, että jokainen uusi testi täytyy ajaa aina ennen koodin kirjoittamista. Tämä paljastaa sen, että puuttuuko testattava ominaisuus todella systeemistä, käyttäytyykö testi kuten odotettiin ja ovatko virheilmoitukset riittävän kattavia. Tarvittaessa testiä ja virheilmoituksia voi parantaa. Kun uusi koodi on valmis, testi ajetaan uudestaan. Testien tarjoaman diagnostiikan kehittämistä on pidetty niin tärkeänä, että sitä on esitetty syklin neljänneksi vaiheeksi [FrP09].

2.1.3 TDD on suunnittelutekniikka

Kun testit kirjoitetaan ennen koodia joutuu kehittäjä miettimään mitä koodilta halutaan sen sijaan, että jäätäisiin miettimään toteutuksen yksityiskohtia. Kun kehitys etenee testit edellä, testaus saa uuden ulottuvuuden ja muuttuu suunnitteluksi [FrP09]. Kun testiä kirjoitetaan eikä testattavaa koodia vielä ole olemassa, täytyy testin kirjoittajan kuvitella (programming by intention) testauksen kohde. Testaajalla on vapaus kuvitella testauksen kohde ominaisuuksiltaan mahdollisimman hyväksi testattavuudeltaan ja kirjoittaa testi sen mukaisesti [Kos08]. Tämä ohjaa koodin muodostumista testattavaksi. Testattavuus on laatuominaisuus ja on myös itsessään vahva signaali muutenkin laadukkaasta koodista. Esimerkiksi muutoksiin reagoiminen onnistuu testattavalta koodilta paremmin [FrP09 s.229].

Testien kuunteleminen auttaa suunnittelussa. Jos testin luominen on vaikeaa, pitää miettiä miettiä mistä se johtuu [Frp09 s.229]. Taustalla on monesti ongelmia ohjelman rakenteissa (design). Testit pitäisi pystyä kirjoittamaan ymmärrettäviksi ja samalla

ylläpidettäviksi koherenteiksi yksiköiksi [Frp09 s.57]. Liian pitkä ja monimutkainen testi kertoo siitä, että koodi pitää jakaa pienempiin yksiköihin. Tavoitteena on parantaa vastuiden jakoa (separation of concerns) testeissä ja koodissa, ja nostaa molempien koheesiota.

Testiä kirjoitettaessa joudutaan miettimään miten testattavaa kohdetta käytetään. Lisäksi joudutaan miettimään minkä muiden osien kanssa testattava kohde kommunikoi ja millaista kommunikoinnin pitäisi olla. Kun testiä luodaan, testattavan kohteen riippuvuudet joudutaan määrittelemään ja asetetaan kohteeseen. Riippuvuuksien asettaminen parantaa kohteen ympäristöriippumattomuutta (context independence) [FrP09 s.57]. Ympäristöriippumattomuus yksinkertaistaa kohdetta, koska tällöin kohde ei itse hallinnoi riippuvuuksiaan. Lisäksi ympäristöriippumattomuus ohjaa ohjelmistoa kohti korkeampaa koherenssia ja parantaa mahdollisuuksia muutoksien tekoon: riippuuvuudet voidaan vaihtaa ulkoisesti konfiguroimalla, ilman että koodiin välttämättä tarvitsee koskea [FrP09 s.55].

TDD:ssä pyritään mahdollisimman laadukkaaseen koodiin myös panostamalla kaikkien koodin osien nimeämiseen. Kommentteja ei tarvita jos koodin ja testien osat on nimetty oikein [Mar09]. Jos testit on kirjoitettu kuvaavasti, ne toimivat hyvin tarkkana dokumenttina testattavasta kohteesta. Ilman testejä koodin yksiköiden väliset kommunikoinnin tavat (communicational patterns) jäisivät muuten dokumentoimatta [FrP09]. Esimerkiksi Javan kohdalla testauksen työkalut kuten EasyMock [Eas12], laajentavat ohjelmointikielen ilmaisun mahdollisuuksia siten, että kommunikoinnin tavat voidaan tuoda helposti ja eksplisiittisesti esiin [FrP09]. Kommunikoinnin tavoilla tarkoitetaan esimerkiksi sitä, miten monta kertaa ja missä järjestykssä yksiköt kutsuvat toisiaan.

TDD:ssä yritetään välttää suurta ennakkosuunnittelua ennen totetuksen aloittamista. TDD:n ideana on, että toteutus itse ohjaa suunnittelua kohti parasta mahdollista rakennetta aina sille koodille, mikä on kullakin hetkellä olemassa. Tärkeimpänä työkaluna parhaan rakenteen löytämisessä on TDD-syklin refaktorointivaihe [Kos08].

Suurta ennakkosuunnittelua halutaan välttää koska vaatimukset muuttuvat jatkuvasti [Kos08]. Lisäksi jokainen projekti on erilainen, eivätkä parhaatkaan ammattilaiset pysty näkemään kaikkea etukäteen [FrP09]. TDD:ssä kaikki päätökset pyritään tekemään olettamusten sijaan tietoon perustuen [Kos08]. Tietoa saadaan kooditasolla ohjelman kehittyessä testejä kuuntelemalla ja muutenkin ymmärrys ohjelmistosta ja sen ympäristöstä kasvaa projektin edetessä.

2.1.4 TDD testauksena

TDD:llä kehitettäessä syntyy paljon yksikkötestejä: jokaista yksikkötestaustason koodin osaa kohden pitäisi jostain löytyä myös testi. Testejä myös ajetaan usein. Testi ajetaan heti kun se valmistuu. Seuraavaksi se ajetaan testin läpäisevän koodin kirjoituksen jälkeen, jolloin myös kaikki muutkin testit ajetaan uudestaan. Kaikki testit ajetaan siltä varalta, että havaitaan uuden koodin mahdollisesti aiheuttamat virheet koko systeemin laajuudessa. Tätä kutsutaan regressiotestaukseksi (regression testing) [Kos08].

TDD-sykli on niin tiivis, että yksikkötesteistä muodostuu pieniä. Ideaalitilanteessa testi lisää paljon ymmärrystä ohjelmasta ja on nopea kirjoittaa. Tälläinen testi on hyvä kandidaatti, kun mietitään seuraavaa työn alle otettavaa testiä. Aina selkeästi parasta seuraavaa testiä ei voida osoittaa. Yleisohjeeksi on annettu helposti ymmärrettävissä olevan ja nopeasti toteutettavissa olevan testin valitseminen seuraavaksi testiksi [Kos08].

TDD:ssä kaikki koodi yksikkötestataan, mutta tavoitteena ei ole sataprosenttinen kattavuus. Käytännössä mielekäs kattavuus esimerkiksi J2EE (Java enterprise edition) projekteille on keskimäärin 85% [Kos08 s.40].

Testejä kirjoitetaan niin paljon, että hyvät testaamisen taidot ovat TDD:n käytössä välttämättömiä. Käytännössä TDD:n määrittämä testaus vaatii toimiakseen myös hyvät työkalut. Testaustaitojen lisäksi myös testaustyökalujen hallinta nousee tärkeäksi. Testien kirjoittamiseen pätevät samat laadukkaan koodin kirjoittamisen säännöt kuin muunkin koodin kirjoittamiseen. Testit muodostavat TDD:ssä noin puolet kaikesta

koodista, joten niitä pitää huoltaa yhtä kurinalaisesti kuin kaikkea muutakin koodia. Pahimmillaan testit muodostuvat hidasteeksi koko kehitykselle ja ne ajaudutaan paineessa jopa poistamaan kokonaan, jos niitä ei ole pidetty kunnossa [Kos08].

Testausvaiheeseen perinteisesti tiiviisti liittynyt koodin suorituksen seuraaminen rivi riviltä (debugging) virheiden löytämisen toivossa, käy TDD:n myötä tarpeettomaksi vaiheeksi. TDD-sykli on niin tiivis, että kehittäjä tietää aina, mikä kohta koodista aiheutti virheen, ilman virheiden etsintään tarkoitettujen työkalujen käyttöä. TDD:tä perinteisimmissä ja vähemmän kurinalaisissa käytänteissä virheiden etsimiseen kuluu usein paljon aikaa [Kos08].

2.2 ATDD

Testivetoisuuden ideaa on sovellettu myös yksikkötestausta korkeammilla tasoilla. ATDD [Bec02] (acceptance TDD) vie testivetoisuuden aina asiakasrajapinnalle asti, vaatimusmäärittelyn tueksi. ATTD:n määrittämä testivetoinen määräys kehitysprosessille on se, että uuden toiminnallisuuden toteutusta ei saa aloittaa ennen kuin siitä on kirjattu hyväksymistesti [Kos08].

ATDD:n keskeinen tavoite on lisätä ja parantaa asiakkaan ja ohjelmistoa kehittävien tahojen välistä kommunikaatioita. ATDD:ssä kehittäjätiimi ja asiakas luovat tiiviin yhteistyön tuloksena vaatimuksista listan hyväksymistestejä (acceptance test). Testit kirjoitetaan niin formaalissa muodossa, että niistä voidaan luoda työkaluilla automaattisesti ajettavia testejä [Kos08].

Hyväksymistestilista toimii selkeänä työn edistymisen mittarina. Vaatimusten kirjaaminen formaalien hyväksymistestien muotoon vähentää myös merkittävästi muun dokumentaation tarvetta. Parhaimmillaan se poistaa tarpeen kirjoittaa vaatimus- ja toiminnallisuuskuvauksista pitkiä kirjallisia dokumentteja, joiden ylläpitäminen on raskasta [Kos08].

ATDD yhdistyy TDD:hen siinä vaiheessa kun hyväksymistesti otetaan työn alle. Hyväksymistesti voidaan jakaa pienempiin osiin, joista kukin toteutetaan TDD:llä [Kos08].

2.3 LonTDD

LonTDD on Lontoon XP-yhteisössä kehittynyt TDD:n muoto, missä hyväksymistesteillä on erityinen merkitys. LonTDD:ssä jokaisen lisättävän toiminnallisuuden toteuttaminen aloitetaan hyväksymistestin kirjoittamisella kooditasolla. LonTDD on yhteneväinen ATDD kanssa siinä mielessä, että toteutus ei ala ennen kuin hyväksymistesti on olemassa. Hyväksymistestin pitäisi testata koko systeemi jotain reittiä päästä päähän, eli toimia läpäisytestinä [FrP09].

LonTDD:ssä läpäisytestin läpäisevä koodi pyritään toteuttamaan aluksi mahdollisimman nopeasti ja yksinkertaisesti. Ensimmäisen toteutuksen aikana ei esimerkiksi kiinnitetä huomiota mihinkään TDD:n hyvistä käytännöistä: yksikkötestejä ei kirjoiteta, koodin laatua ei pidetä erityisemmin silmällä eikä refaktorointeja tehdä [FrP09].

Erityisesti koko projektin ensimmäisellä läpäisytestillä on merkittävä rooli LonTDD:ssä. Sen yhteydessä ei ainoastaan luoda nopeasti testin läpäisevää koodia, vaan myös koko asiakastoimitusta varten vaadittava infrastruktuuri automatisoidaan. Tämä käsittää muun muassa ohjelman kääntämiseen tarvittavien rakennus- (build) sekä käyttöönotto-skriptien (deploy) luomisen. Tätä ensimmäistä versiota ohjelmasta kutsutaan nimellä kävelevä luuranko (walking skeleton) [FrP09].

Kävelevän luurangon luominen kestää yleensä kauan, mutta se kannattaa tehdä heti koska siitä saatava hyöty kumuloituu jatkossa. Se luodaan, jotta kehittäjien ymmärrys koko toimintaympäristöstä alkaa hahmottua. Kun ohjelmisto integroidaan heti alussa luurangon avulla, on integraation toteutus paljon helpompaa kuin jos se tehtäisiin vasta koko projektin viimeisenä vaiheena [FrP09]. Jos yksikkötestit ohjaavat koodia testattavaksi, niin läpäisytestit ohjaavat sitä integroituvaksi. Lisäksi kävelevän

luurangon lähestymistapa tukee palautteen saamista asiakkaalta. Kun toimitusprosessi on automatisoitu ja ensimmäinen toiminnallisuus on toteutettu mahdollisimman nopeasti, päästään toimitussykliin kiinni suorinta reittiä ja toimitussyklin tiheys voidaan pitää korkeana jatkossa [FrP09]. Tämä tukee hyvin XP:n tavoitetta mahdollisimman tehokkaasti palautetta tuottavan infrastruktuurin luomisessa.

Kävelevän luurangon luomisen jälkeen LonTDD muuttuu TDD:ksi työn alla olevan hyväksymistestin toteutuksen loppuun asti. Yksinkertainen ja mahdollisimman nopeasti toteutettu luuranko jatkokehitetään TDD:llä oikeasti toimivaksi. Kun hyväksymistestin läpäisevä koodi on valmis, siirrytään seuraavan hyväksymistestin luomiseen joka aloittaa LonTDD:ssä uuden syklin [FrP09].

LonTDD parantaa TDD:n kykyä kerätä tietoa ja palautetta ympäristöstään. LonTDD:n filosofia on lähestyä kohdetta ulkoapäin (outside in development) [FrP09], kun taas TDD:ssä pyritään välttämään ensimmäisen testin yhteydessä koko kuvan (big picture) mietintää ja keskitytään businesslogiikkaan [Kos08 s.48].

ATDD:hen verrattuna LonTDD on kauempana asiakasrajapinnasta. LonTDD:ssä luodaan hyväksymistestejä kuten ATDD:ssä, mutta LonTDD ei ota kantaa siihen, millainen prosessi vaatimuksien keräämisen taustalla on.

2.4 Web-TDD

Web-TDD on testivetoisuuden ideaa erityisesti web-sovelluksien kehitystä varten soveltava menetelmä [RPG10]. Web-TDD:ssä testivetoisuus on samalla tasolla kuin ATDD:ssä eli hyväksymistestitasolla. Myös laajudessaan Web-TDD on lähellä ATDD:n tasoa.

Web-TDD painottaa läheistä työskentelyä asiakkaan kanssa vaatimuksien määrittämisessä. Web-TDD:n keskeisin idea on käytettävyyden tärkeyden korostaminen web-sovellusten yhteydessä ja sitä kautta erilaisten käytettävyystestien

luominen. Vaatimusten määrittämisessä asiakkaan kanssa käytetään hyväksi esimerkiksi käyttöliittymistä nopeasti tehtyjä onttoja jäljitelmiä (mock) sekä käyttäjien liikkeitä mallintavia UID-kaavioita (user interaction diagrams).

Web-TDD nostaa esiin erityisesti toiminnalliset käytettävyysvaatimukset (functional usability requirements) [RPG10]. Nämä ovat käytettävyysvaatimuksia, jotka samalla vaikuttavat merkittävästi myös ohjelmiston arkkitehtuuriin. Esimerkkeinä web-sovelluksiin liittyvistä toiminnallisista käytettävyysvaatimuksista ovat muun muassa kohteiden tallentaminen suosikkeihin (favourites) sekä monisivuiset prosessit eli wizard-tyyppiset ratkaisut. Näiden vaatimusten löytämiseksi käytetään tarkoitusta varten kehitettyjä kysymyslistoja, joiden avulla asiakkaalta kerätään tarvittavat tiedot [RPG10].

Kun vaatimus on löydetty, se kirjataan Web-TDD:ssä ensin luonnollisella kielellä. Seuraava askel on muokata se formaalimpaan muotoon, jotta vaatimuksesta voidaan generoida automaattinen testi. Automaattiset testit ovat Web-TDD:ssä Selenium-testejä [Sel12], jotka toimivat hyväksymistestitasolla. Selenium-testit vastaavat testejä, jotka manuaalisesti tehtyinä tehtäisiin selaimen kautta.

Poikkeuksellista Web-TDD:ssä on se, että hyväksymistestien toteuttava koodi luodaan puoliautomaattisesti käyttäen hyväksi MDSD (model-driven software development) tekniikoita, missä koodi generoidaan esimerkiksi Javaa paljon korkeampien suunnittelukäskyjen kautta [RPG10].

2.5 Aiempi tutkimus TDD:hen liittyen

TDD on ollut jatkuvan tutkimuksen kohteena [CSP11]. Tutkimusten pohjalta selvien johtopäätösten teko TDD:n vaikutuksista on kuitenkin usein vaikeaa. Tämä johtuu siitä, että usein eri tutkimukset ovat tuottaneet ristiriitaisia tuloksia [CSP11]. Laadukkaiden tutkimusten määrä on myös melko rajallinen. Eräässä tutkimuksessa pyrittiin löytämään kaikki TDD:stä tehty tieteellisen yhteisön arvioima (peer reviewed) empiirinen tutkimus, sisältäen akateemiset sekä teollisuudessa tehdyt kokeet, malliesimerkit (case

study), tilannekatsaukset (survey) ja kirjallisuusvertailut (literature reviews) [CSP11]. Vain 48 sopivaa tutkimusta kyettiin löytämään. Yhtenä ongelmana pidetään myös puhtaasti testivetoisuuden vaikutusten erottamisen hankaluutta muista tekijöistä.

Tutkimusten perusteella TDD:n vaikutuksista selvimmältä näyttää koodin laadun parantuminen [BLM11 ,CSP11]. Toinen selvä positiivnen vaikutus on testauksen kattavuuden nousu [BLM11 ,CSP11]. Hyvänä asiana nähdään myös kehittäjien positiivinen kuva (perception) TDD:stä [CSP11]. Muiden vaikutusten kohdalla tutkimustuloksista on vaikeaa nähdä selkeitä positiivisia tai negatiivisia käyriä. Tutkimus löysi yhteensä 18 kohtaa, joihin TDD:n nähtiin vaikuttavan tai jotka vaikuttivat TDD:n taustalla siten, että ne muuttivat TDD:n käyttöä [CSP11].

TDD:n vaikutusten lisäksi myös sen käyttöä ja erityisesti käytön haasteita on tutkittu [BLM11 ,CSP11, Kol11]. Suurimmiksi ongelmiksi TDD:n käytölle ovat nousseet heikko TDD:hen sitoutuminen, legacy-koodi, vankkojen testaustaitojen puute, ympäristöön ja työkaluihin liittyvät haasteet, ulkopuolelta tulevat tiukat aikataulupaineet, vähäinen etukäteissuunnittelu (up-front design) ja riittämätön TDD:n hallitseminen [CSP11], sekä epäusko TDD:n toimivuuteen [BLM11].

Suurin osa tunnistetuista TDD:n haasteista ovat haasteita ilman TDD:täkin. Esimerkiksi vankkoja testaustaitoja tarvitaan laadukkaassa ohjelmistokehityksessä, vaikka TDD:tä ei käytettäisikään. TDD:tä kohtaan tunnettuun skeptisyyteen on ehdotettu parannukseksi sitä, että jokainen ryhmänvetäjä on TDD:n käytön mestari (champion) sekä erittäin sitoutunut sen käyttöön [BLM11]. TDD itsessään vastaa osittain ongelmaan etukäteissuunnittelun vähäisyydestä. TDD:ssä ei ole ohjeistettu, että sitä ei saisi tehdä ollenkaan, vaan tilanne pitää arvioida aina erikseen [Kos08].

Kaikki tutkimukset painottavat TDD:n jatkotutkimuksen tärkeyttä [BLM11, CSP11, KOL11].

3 Testauksen työkalut

3.1 Junit

3.2 EasyMock

3.3 Spring-test-mvc

3.4 Selenium

4 Web-sovellukset

4.1 Web-sovellusten testaus

4.2 Käyttöliittymät

4.3 Kerrosarkkitehtuuri

4.4 Spring

4.4.1 MVC

5 Web-sovellus esimerkki

6 Web-sovellus esimerkin testivetoinen kehitys

7 Kehityksen analyysi

8 Yhteenveto

Lähteet

[Bec99] Beck, K., Extreme programming explained: embrace change. Addison-Wesley professional, 1999.

[Bec02] Beck, K., Test driven development: By example. Addison-Wesley professional, 2002.

[BeS08] Begel, A., Simon, B., Struggles of new college graduates in their first software development job. Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education, New york, USA, 2008, sivut 226-230.

[BLM11] Buchan, J., Li, L., MacDonell, S.,Causal factors, benefits and challenges of test-driven development:Practitioner perception. Proceedings of the 2011 18th Asia-Pacific Software Engineering Conference, Washington, USA, 2011, sivut 405-413.

[BrS11] Brannen S. Stoyanchev R. Https:/github.com/SpringSource/spring-test-mvc , kalvot, (14.3.2012).

[CSP11] Causevic, A. Sundmark D., Punnekkat S., Factors limiting industrial adoption of test driven development: A Systematic review. Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation, Washington, USA, 2011, sivut 337-346.

[Eas12] EasyMock kotisivut http://www.easymock.org/ (20.3.2012).

[Fow99] Fowler, M., Refactoring:Improving the design of existing code. Addison-Wesley professional, 1999.

[FrP09] Freeman, S., Pryce, N., Growing object oriented software. Addison-Wesley professional, 2009.

[Int11] International telecommunications union, The world in 2011: ICT facts and figures http://www.itu.int/ITU-D/ict/facts/2011/index.html (17.3.2012).

[Kol11] Kollanus S., Critical issues on test-driven development. Proceedings of the 12th Internationl Conference on Product-focused Software Process

Improvement, Torre Canne, Italy, 2011, sivut 322-336.

[Kos08] Koskela, L. Test driven: practical TDD and acceptance TDD for Java developers. Manning, 2008.

[LSA11] Laanti M, Salo O., Abrahamsson P., Agile methods rapidly replacing traditional methods at Nokia: A survey of opinions on agile transformation. Journal of Information and Software Technology, vol 53, issue 3, 2011, sivut 276-290.

[Mar09] Martin R., Clean code: A Handbook of agile software craftsmanship. Prentice hall, 2009.

[RPG10] Robles E., Panach J., Grigera J. et al., Incorporating usability requirements in a test/modeldriven web engineering approach. Journal of Web Engineering, vol 9, issue 2 , 2010, sivut 132-156.

[ScB01] Schwaber, K., Beedle, M., Agile software development with Scrum. Prentice Hall, 2001.

[Sel12] Seleniumin kotisivut http://seleniumhq.org/ (20.3.2012).

[Sho07] Shore J. The art of agile development.O'Reilly media, 2007.

[SLD07] Schwaber, K., Laganza, D., D'Silva, D., The truth about agile processes: Frank answers to frequently answered questions. Forrester report, 2007.

[Spi03] Spinellis, D., Code reading: The open source perspective. Addison-Wesley, 2003.

[Spr12] Springsource communityn kotisivu http://www.springsource.org/ (18.3.2012).

[Til11] Tilastokeskus, Internetin käyttö kodin ja työpaikan ulkopuolella yleistyy http://www.stat.fi/til/sutivi/2011/sutivi_2011_2011-11-02_tie_001_fi.html (17.3.2012).

[TWL10] Tian D., Wen J., Liu Y. et al., A Test-Driven web application model based on layered approach. Information Theory and Information Security (ICITIS) IEEE International Conference, Peking, Kiina, 2010, sivut 160-163.

[Vmw12] Vmwaren uuden Spring version julkaisutiedot http://www.vmware.com/company/news/releases/vmw-spring-momentum-3-14-12.html (19.3.2012).

[WGG10] West, D., Grant, T., Gerush, M. et al., Agile development:Mainstream adoption has changed agility, Forrester report, 2010.