Tampereen ammattikorkeakoulu

Kent Beck: Test Driven Development: By Example

Kirjallinen raportti

Teemu Viikeri

Tampereen ammattikorkeakoulu
4A00EZ62-3001 Backend-kehitys
Lokakuu 2020

Liiketalouden tutkinto-ohjelma
Tietojenkäsittely

SISÄLLYS

1 JOHDANTO

Tämän pakollisen kirjallisen työn kirjaksi valitsin raportin otsikon mukaisesti Kent Beckin teoksen *Test Driven Development: By Example*. Kirja käsittelee kiitettävän monipuolisesti testivetoisen kehityksen (eng. *test-driven development, TDD*) eri periaatteita, osa-alueita, käsitteitä ja esimerkkejä. Aion tässä raportissa käsitellä testivetoista kehitystä lukemani kirjan sekä muiden lähteiden pohjalta.

Pyrin vastaamaan seuraaviin kysymyksiin:

1. Mikä on testivetoisen kehityksen keskeinen periaate?
2. Miten testivetoinen kehitys käytännössä toteutetaan, eli mitä testivetoisen kehityksen syklin vaiheissa tapahtuu?

   Näihin kahteen kysymykseen vastaan raportin ensimmäisessä pääluvussa, Testivetoinen kehitys.

   Testivetoisessa kehityksessä riittää pohdittavaa ja uutta opittavaa, sillä siinä joudumme laittamaan ajattelumme hieman eri suuntaan kuin mihin olemme tottuneet aiemmin. Se, mitä testivetoisesta kehityksestä voi loppujen lopuksi ammentaa, voi olla hyvinkin suurta.

3. Miksi testien pitäisi olla toisistaan riippumattomia?

   Tähän kysymykseen vastaan raportin toisessa pääluvussa, Riippumattomuus.

   Tässä testien riippumattomuuteen liittyvässä luvussa on kyse olennaisesti sekä ohjelman että testien designista. Jos opimme sen, mistä riippumattomuudessa on oikeasti kyse, tekee se meistä loppujen lopuksi parempia ohjelmoijia.

4. Mitä haittoja syntyy koodin uudelleen kirjoittamisesta?

   Tähän kysymykseen vastaan lyhyesti tämän raportin kolmannessa pääluvussa, Koodin uudelleen kirjoittaminen.

   Tässä raportissa tulen paljon kehumaan ja testivetoista kehitystä ja sen hyötyjä, mutta mikään asia ei ole ilman myös sen haittoja tai negatiivisia puolia.

Lopuksi, pohdintakappaleessa aion koota raportin kolmen ensimmäisen pääluvun pohjalta pääteemat lyhyeksi kokonaisuudeksi. Tämän lisäksi tuon henkilökohtaisia ajatuksia siitä, mitä erinäisiä tuntemuksia kirjan lukeminen herätti. Yritän pitää tämän raportin jollain tapaa lähellä itseäni, jotta unohtaessani sen, mitä oivalsin Beckin kirjaa lukiessa ja tätä raporttia kirjottaessa, muistaisin ne oivallukset uudelleen tämän kautta.

2 TESTIVETOINEN KEHITYS

Kun testaamme jotakin, haluamme arvioida sitä, miten jokin asia toimii tai toimiiko testaamamme asia ollenkaan. Testin tarkoituksena on siis antaa meille joko hyväksyvä tai hylkäävä vastaus testaamaamme asiaan. (Beck 2002, 125).

Testaus liittyy hyvin olennaisesti ohjelmistojen kehittämiseen. Ohjelmistojen tulee tehdä se, mitä niiltä vaaditaan – useimmiten ilman ongelmia ja yllätyksiä – ja juuri tähän me pyrimme testauksella (Myers 1979, 9).

Kun alamme testaamaan jotain, yleensä testaus tapahtuu niin, että luomme ensiksi asian ja tämän jälkeen testaamme tätä asiaa. Testivetoisessa ohjelmistokehityksessä käännämme tämän ajatuksen kuitenkin päälaelleen ja lähdemme päinvastaisesta suunnasta kohti lopputulosta.

Testivetoisen ohjelmistokehityksen perusperiaatteena on, että emme ensiksi ohjelmoikaan ohjelmaamme, jota myöhemmin testaisimme, vaan luomme ensiksi testitapauksen haluamastamme ohjelmasta.

Beck (2002, ix) esittelee jo kirjansa esipuheessa testivetoisen kehityksen mantran, johon koko testivetoisen kehityksen voi tiivistää kolmeen eri vaiheeseen: punainen/vihreä/refaktoroi. Tulen seuraavaksi avaamaan, mitä nämä tiivistyksen kolme kohtaa tarkoittavat.

Tiivistyksen ensimmäinen kohta, punainen, tarkoittaa sitä, että aluksi luodaan testitapaus, joka epäonnistuu eli testi antaa punaisen, hylätyn tuloksen. Näin pitääkin tapahtua, sillä tässä kohtaa testin tarkoitus oli vasta olla pohja ja suunnitelma sille, miten haluttua toiminnallisuutta ohjelmassa halutaan lähteä kirjoittamaan seuraavassa vaiheessa. Tässä vaiheessa ei tarvitse siis olla edes valmista koodia ollenkaan valmiina, jonka pohjalta testiä kirjoitetaan. Testiä ei pysty välttämättä edes aloittamaan sen takia, koska testin sisältämä koodi sisältää sellaisia osia, mitä ei ole edes vielä olemassa. Tämäkin on täysin sallittua tässä vaiheessa.

Mantran seuraava kohta, vihreä, tarkoittaa sitä, että hylätyn tuloksen jälkeen testi pitäisi saada toimimaan keinolla millä hyvänsä ja niin vähällä määrällä kuin mahdollista. On tärkeää, että tässä vaiheessa testin läpäisemiseksi kirjoitettu koodi on kompaktia, koska kehitysvaiheiden tuleekin olla lyhyitä. Punainen, hylätty tulos pitäisi saada siis vihreäksi, hyväksytyksi. Punaisessa vaiheessa asetetaan haluttu lopputulos, vihreässä vaiheessa haluamme saada testin hyväksytyksi. Se, miten hyväksytty tulos saatiin, ei välttämättä ole toteutettu tekniseltä puolelta kauniisti, mutta vaihe on suoritettu oikein, sillä testi läpäistiin hyväksytysti.

Mantran viimeinen kohta on ”refaktoroi”. Refaktoroinnin eli lähdekoodin uudelleenjärjestämisen ideana on optimoida koodia, poistaa vihreässä vaiheessa tehdyt huonot ratkaisut, duplikaatit, siirtää koodia paikasta toiseen, missä niiden on järkevämpi olla, ja niin edelleen. (Beck 2002, ix). Ensimmäisessä vaiheessa keskityttiin designiin koodin lopputuleman kautta. Tässä vaiheessa keskitytään myös designiin, mutta tällä kertaa se tapahtuu teknisen toteutuksen kautta. Kun toisessa vaiheessa koodia kirjoitetaan mahdollisimman kompaktisti, auttaa se

myös tässä vaiheessa, kun koodia refaktoroidaan. Koodia ei tällöin tarvitse välttämättä muuttaa monessa eri paikkaa ja pitkästi.

Näiden kehitysvaiheiden tarkoitus on moninainen. Ohjelmoinnissa pyritään kirjoittamaan sellaista koodia, joka on niin sanotusti ”puhdasta koodia” (eng. *clean code*). Tämä tarkoittaa sitä, että koodi on tällöin toimivaa, ennalta arvattavaa ja se tuntuu hyvältä. (Beck 2002, viii).

Testivetoinen kehitys auttaa siinä, että yllä mainitun mantran kolme vaihetta laittaa ohjelmoijan miettimään kirjoittamansa koodin designia. Tavoitteena on halutun toiminnallisuuden sellainen design, joka on mahdollisimman kompakti. Kompakti koodi vähentää koodissa olevia asioita, mitä siinä ei tarvitsisi olla, joka puolestaan vähentää ohjelmistovirheiden määrää. Kompakti koodi on myös helpommin luettavaa ja ymmärrettävää – niin koodin kirjoittajalle kuin kirjoitetun koodin muille lukijoille. (Beck 2002, ix). Tällaista koodia voisi kutsua ”puhtaaksi koodiksi”.

Designin ohella on toinenkin asia, johon testivetoinen kehitys omine vaiheineen tarjoaa apua: testivetoinen kehitys antaa ohjelmoijalle luottamusta siihen, että kirjoitettu koodi toimii, kuten pitääkin. Testien tarkoituksena onkin siirtää arvio koodin toimivuudesta ohjelmoijan omasta arviosta testeille, jotka mahdollisuuksien mukaan ovat objektiivisempia arvioiden antajia kuin ihminen. (Beck 2002, x-xi).

Testeillä on vahva psykologinen vaikutus. Testit ovat palautteen antajia, jotka joko lisäävät ohjelmoijan stressiä tai lieventävät stressiä. Hylätyt testit tuottavat stressiä, joka puolestaan lisää mahdollisten virheiden tekemistä, joka puolestaan lisää hylättyjä testejä ja se taas lisää stressin määrää. On kyse positiivisesta palautesilmukasta stressin näkökulmasta. (Beck 2002, 129).

Testivetoinen kehitys pyrkii vastaamaan tähän positiiviseen palautesilmukkaan kehitysvaiheillaan. Ohjelmoija tiedostaa, että ensimmäinen kehitysvaihe tulee päätymään hylättyyn testitulokseen, mutta kyse onkin tässä vaiheessa vasta ohjelman toiminnallisuuden sisällön suunnittelusta. Tämän jälkeen ohjelmoija pyrkii saamaan testin toimimaan niin nopeasti ja pienellä vaivalla kuin mahdollista. Testin tulokseksi tulee vihreä, hyväksytty tulos. Jo tässä vaiheessa halutaan siis nopeasti katkaista stressipositiivinen palautesilmukka. (Beck 2002, 130).

Testivetoisessa kehityksessä halutaan siis edetä pienin askelin, joka vie eteenpäin järkevää ohjelmiston designia, joka vähentää yksinkertaisuudellaan ja kompaktiudellaan ohjelmistossa esiintyviä ohjelmistovirheitä sekä antaa ohjelmiston kehittäjälle luottamusta kirjoittamaansa koodia kohtaan. Pienistä puroista kasvaa iso joki. Tämä pitää paikkaansa myös ohjelmistokoodin kanssa. Kompaktisti ja järkevästi kirjoitetuista koodeista syntyy toiminnallisuuksia ja nämä toiminnallisuudet muodostavat kokonaisuuden, kokonaisen ohjelmiston. Kompaktit toiminnallisuudet helpottavat virheiden ylläpitoa sekä ohjelmiston kehitysmahdollisuuksia tulevaisuudessa. Kaikki siis voittavat.

3 RIIPPUMATTOMUUS

Beck (2002, 127) kirjoittaa kirjassaan seuraavasti: “Miten ohjelmistotestit tulisi vaikuttaa toisiinsa? Ei mitenkään.” Mitä Beck tarkoittaa tällä?

Beck tarkoittaa kirjoittamallaan lauseellaan sitä, että ohjelmistotestaus pitäisi koostua toisistaan riippumattomista testeistä. Kun testit luodaan siten, että ne ovat toisistaan riippumattomia, se ohjaa ohjelmoijaa luomaan testit siten, että ne ovat yksittäisiä, koherentteja komponentteja, joita pystyy kuitenkin helposti yhdistämään isommiksi kokonaisuuksiksi. Tämä auttaa moneen eri asiaan. Se auttaa etenkin ajattelemaan koodia siten, että luotavat komponentit olisivat mahdollisimman uusiokäytettävissä erilaisissa tilanteissa – ei vain siihen tilanteeseen, johon komponentti ensimmäisen kerran luotiin. Tämän lisäksi, kun testit ovat toisistaan riippumattomia, on testeissä esiintyvien virheiden käsittely paljon helpompaa. Jos testit olisivat toisistaan riippuvaisia, tällöin kaikki testit, jotka ovat sidoksissa toisiinsa, olisivat tulokseltaan hylättyjä, vaikka oikeasti hylkäyksien syy liittyisi vain yhteen testiin. Testien riippumattomuus toisistaan auttaa siis myös ohjelmoijaa ymmärtämään paremmin ohjelmistossa esiintyvien virheiden ja ongelmien taustaa. (Beck 2002, 127-128).

Testien riippumattomuus hyödyttää myös siinä tilanteessa, jos halutaan valita erilaisia testijoukkoja monista eri testeistä. Ihan kuten ohjelmia voi kasata eri moduuleista tai komponenteista, niin samalla tavalla voidaan myös kasata erilaisia testijoukkoja yksittäisistä testeistä. (Beck 2002, 127). Tämä ei päde pelkästään yksittäiseen ohjelmoijaan vaan siihen koko sovelluskehittäjien joukkoon, jotka ohjelmaa kehittävät. Joku toinen ohjelmoija voi tarvita erilaisia testijoukkoja kuin toinen ohjelmoija. Kun testit tehdään itsenäisiksi, riippumattomiksi toisistaan, voi jokainen ohjelmoija testata testijoukkoja oman halunsa mukaan.

4 KOODIN UUDELLEEN KIRJOITTAMINEN

Kun koodia kirjoitetaan uudelleen, esimerkiksi refaktoroinnin yhteydessä, voidaan tällöin tehdä paljon asioita, joista on hyötyä. Koodin uudelleenkirjoituksesta voi kuitenkin syntyä myös haittoja. Koodin uudelleen kirjoittaminen on kallista ja se vie aikaa muulta, uusien toiminnallisuuksien kehitysajasta. Vaikka uudelleen kirjoittaminen voi joissain tilanteissa olla kyllä järkevää, voi uudelleen kirjoittamisen yhteydessä poistua jotain sellaista ohjelmiston designista, mitä ei välttämättä tule uudestaan mukaan uudelleen kirjoittaessa. Uudelleenkirjoittamisen yhteydessä voi myös syntyä jotain sellaista uutta koodia, jota ohjelmiston nykyiset testit eivät tue tai sisällä. Tällöin uudelleenkirjoittaminen lisää entisestään muutettavaa ja/tai uudelleen kirjoitettavan koodin määrää. Tällöin tulisi olla varma siitä, että mitä lähdetään kirjoittamaan uudelleen ja miksi näin tehdään sekä oliko aikaisemmassa koodissa silti jotain sellaista, mitä kannattaa pitää mukana jatkossakin.

5 POHDINTA

Ennen kuin kunnolla ymmärsin, mitä testivetoinen ohjelmistokehitys oli, luulin, että kyseessä on erittäin vahva tekninen ohjelmointiympäristö ja tekniikka, jossa minulla täytyisi olla laaja ymmärrys ohjelmointiin liittyvistä asioista, jotta voisin edes ymmärtää tai hyötyä testaamisesta osana ohjelmointia. Tartuin kirjaan siinä uskossa, että nyt opin jotain sellaista, joka kiihdyttäisi osaamisen tasoani korkeammalle kuin mitä se oli ennen kirjan lukemista. Tämä osoittautui vääräksi. Ainakin suurimmalta osin.

Mitä sitten opin kirjasta? Mitä pidemmälle kirjaa luin, sitä enemmän olin sitä mieltä, että testivetoinen ohjelmistokehitys on monella tapaa arvokas tekniikka. Kyseinen tekniikka luo arvoa niin yksittäistä toiminnallisuutta kehittävälle ohjelmoijalle kuin koko ohjelmistolle sekä myös asiakkaalle ohjelmiston tyytyväisten käyttäjien kautta, jotka pääsevät käyttämään ohjelmistoa, jossa on mahdollisesti hyvin vähäisesti ohjelmistovirheitä.

Itselleni kirjan tärkein oivallus oli se, kuinka paljon testivetoinen ohjelmistokehitys voi vaikuttaa positiivisesti ohjelmoijaan psykologian kautta. Itse tunnistan olevani välillä hieman epävarma kirjoittamastani koodista; onko kirjoittamani koodi hyvää, järkevää, hyvin strukturoitua, kompaktia ja niin edelleen. Olen vakuuttunut siitä, että testivetoinen ohjelmistokehitys voisi sopia itselleni aivan todella paljon. Se ei pelkästään antaisi psykologista hyvän olon tunnetta hyväksyttyjen testien kautta vaan se myös ohjaa luomaan sellaista designia ohjelmalle, joka oikeasti on niitä asioita, mitä itsekin toivon kirjoittamiltani ohjelmilta, mutta olen epävarma siitä, onko ne sitä.

Testivetoisesta kehityksestä kannattaa myös huomata se, että se ei ole vain testien tekemistä teknisestä siellä täällä eikä se ole pelkästään omien ohjelmointitaitojen parantamista – mitä testivetoisen kehityksen aikana voi tietysti tapahtua. Jos testivetoista ohjelmistokehitystä seurataan sen perusperiaatteiden mukaisesti, se on lähestymistapa ohjelmointiin, joka ohjaa vahvasti ohjelmoijaa kirjoittamaan sellaista koodia, jonka design on kunnolla mietittyä. Testivetoisen kehityksen kokonaisvaltainen hyöty kompaktin designin edistämisestä aina positiivisiin psykologisiin palautteisiin on sen verran laaja, että en näe syytä ottaa testivetoista kehitystä haltuun. Varsinkin, jos on kyse isommasta ohjelmointitiimistä tai suuremmasta ohjelmiston kehittämisestä, mutta miksi ei loppujen lopuksi pienemmistäkin ohjelmista. Jostain pitää kuitenkin aloittaa. Testien kirjoittamista ja testivetoisen kehityksen hyötyjä pystyy tiettyyn pisteeseen asti opettelemaan ja harjoittelemaan lukemalla, mutta loppujen lopuksi suurin oppi tapahtuu itse ohjelmoidessa testejä.

Tämän raportin sekä varsinkin Kent Beckin kirjan sisällön avulla pystyy tekemään omasta ohjelmointiurastaan tai -harrastuksestaan mielekkäämpää. Itse ainakin yritän valjastaa tämän tiedon käyttööni parhaalla mahdollisella tavalla. Kuten Beck (2002, 3) itsekin sanoo kirjansa alussa, jos on nero, kirjan sääntöjä ja neuvoja ei tarvitse, jos on pölkkypää, kirjan säännöt ei auta, mutta meille, jotka olemme suuressa enemmistössä näiden kahden välissä, testivetoinen ohjelmistokehitys johdattaa meitä paljon lähemmäksi meidän potentiaaliamme.

LÄHTEET

Kent, B. 2002. Test Driven Development: By Example. Addison-Wesley Professional.

Myers, G. 2004. The Art of Software Testing, Second Edition. Hoboken: John Wiley & Sons, Inc.