jamk.fi

Extreme Programming ohjelmistokehi-tyksen opetuksessa

Case: Jyväskylän Ammattikorkeakoulu, Tietojenkäsitte-lyn tutkinto-ohjelma

Janne Hanhela

Opinnäytetyö
Elokuu 2017
Liiketalouden ala
Tietojenkäsittelyn tutkinto-ohjelma

Jyväskylän ammattikorkeakoulu
JAMK University of Applied Sciences

jamk.fi

Kuvailulehti

Tekijä(t)	Julkaisun laji	Päivämäärä
Hanhela, Janne	Opinnäytetyö, AMK	Elokuu 2017
	Sivumäärä 43	Julkaisun kieli Suomi
		Verkkojulkaisulupa myönnetty: x

Työn nimi

Extreme programming ohjelmistokehityksen opetuksessa

Case: Jyväskylän Ammattikorkeakoulu, Tietojenkäsittelyn tutkinto-ohjelma

Tutkinto-ohjelma

Tietojenkäsittelyn tutkinto-ohjelma

Työn ohjaaja(t)

Jarkko Immonen

Toimeksiantaja(t)

Tietojenkäsittelyn tutkinto-ohjelma, Jyväskylän Ammattikorkeakoulu

Tiivistelmä

Jyväskylän ammattikorkeakoulun tietojenkäsittelyn tutkinto-ohjelmassa opetetaan ohjelmistokehitystä erityisesti web- ja pelialoille. Ohjelmistokehitys on kuitenkin hyvin haastavaa ja syvempi oppiminen tapahtuu kurssien jälkeen eikä niiden aikana.

Tarkoituksena oli selvittää, miten Extreme programmingin käytäntöjä voitaisiin hyödyntää ohjelmistokehityksen opetuksessa, jotta opiskelijoiden oppiminen helpottuisi ja he voisivat toimia paremmin ohjelmistokehitysprojekteissa työelämässä.

Kehittämistutkimus toteutettiin kolmessa syklissä, joiden aiheet olivat: mikä opetuksen nykytilanne on, mitä Extreme programming on ja miten sitä on käytetty ohjelmistokehityksen opetuksessa tähän mennessä. Kussakin syklissä käytiin läpi seuraavat vaiheet: selvitys, analyysi ja reflektio. Ensin käytiin läpi aineisto, sen jälkeen aineiston pohjalta tehtiin päätelmiä ja tulkintoja ja lopuksi reflektoitiin uutta tietoa ja sen pohjalta kehitettiin ongelmakuvausta ja kehitysehdotusta. Näiden syklien lopputuloksena luotiin kehitysehdotus tutkinto-ohjelmalle.

Kehitysehdotuksessa nousi erityisesti esiin pariohjelmointi opetuksen apuna ja testipohjaisen kehityksen opettaminen. Lisäksi käytännön projektityöskentelyyn tuotiin mukaan mallintaminen.

Työskentelyn aikana nousi esiin myös kysymys: pitäisikö ohjelmistokehityksen koulutukseen kehittää oma prosessimalli, joka antaisi hyvän yleiskuvan ja osaamisen sekä perinteisen että ketterän kehityksen työskentelytavoista ja sovellutuksista.

Avainsanat (asiasanat)

Extreme programming, ohjelmistokehitys, opetus

Muut tiedot

jamk.fi

Description

Author(s)	Type of publication	Date
Hanhela, Janne	Bachelor's thesis	August 2017
		Language of publication: Finnish
	Number of pages: 43	Permission for web publication: x

Title of publication

Extreme programming in teaching Software development

Case: JAMK University of Applied Sciences, Business Information Technology

Degree programme

Business Information Technology

Supervisor(s)

Immonen, Jarkko

Assigned by

Business Information Technology degree programme, JAMK University of Applied Sciences

Abstract

JAMK University of Applied Sciences provides its students with a degree programme in Business Information Technology containing software development and related business studies, especially in the fields of web/mobile and games. The curriculum assumes no knowledge of programming and with limited time allocated to programming related skills, many students struggle with programming. As programming is arguably one of the best ways to find employment after graduating, this is a problem.

Extreme programming is an agile development methodology and many of its practices have gained widespread use. Out of those practices, particularly pair programming has proven successful in helping students to learn programming, which lead to the question, if incorporating Extreme programming into the curriculum could help with learning.

Using this question as the starting point, three research and analysis cycles were carried out, each focusing on a different question: How is programming currently taught in Business Information Technology degree programme, what is Extreme Programming and how has Extreme Programming been used in software development education thus far.

The resulting suggestion focuses on pair programming and test driven development, while highlighting the importance of providing students with some more practical experience in software modelling and other more traditional software development practices.

One possible idea for future research was also discovered. Could it be possible to develop a more tailored methodology for software development education? Such methodology could focus more on learning and give a more complete understanding of both the agile and the traditional side of software development.

Keywords/tags (subjects)

Extreme programming, software development, teaching.

Miscellaneous

Sisältö

1 Johdanto 3

2 Tutkimusasetelma 3

2.1 Tutkimuksen kohteet ja tutkimusongelman löytyminen 3

2.2 Tutkimusmenetelmä ja -kysymykset 4

3 Ohjelmistokehityksen opetuksen nykytilanne Tikossa 6

4 Extreme programming 9

4.1 Arvot 9

4.2 Periaatteet 10

4.3 Käytännöt 15

5 Extreme programming opetuksessa 18

5.1 Käytännön kokemuksia 18

5.2 Osana ohjelmistokehityksen opetusta 22

5.3 Pariohjelmointi opetuksessa 25

6 Kehitysehdotus 28

6.1 Ennen Ticorporatea 28

6.2 Ticorporate 29

6.3 Ehdotuksen perustelut 30

7 Pohdinta 32

7.1 Työn tulos 32

7.2 Luotettavuus ja pätevyys 33

7.3 Tulevaisuuden suuntia 33

Lähteet 35

Liitteet 37

Liite 1. Esimerkkitehtävä ohjelmoinnin alkeiskurssille 37

Kuviot

Kuvio 1 Tutkimuksen toteutus 5

Taulukot

Taulukko 1 Ohjelmointikurssit Tikossa 6
Taulukko 2 Periaatteiden yhteydet painosten välillä 14
Taulukko 3 Extreme programmingin 12 käytäntöä 15

1 Johdanto

Jyväskylän ammattikorkeakoulun tietojenkäsittelyn tutkinto-ohjelmassa (myöhemmin Tiko) opetetaan ohjelmistokehitystä erityisesti web- ja pelialoille. Tähän liittyen opiskelijoille opetetaan ohjelmointia yhdellä alkeiskurssilla sekä useammalla eri kieliin ja teknologioihin keskittyvillä kurseilla. Lisäksi he osallistuvat ohjelmistokehitysprojektiin Ticorporate-nimisessä yrityssimulaatiossa. Erityisesti opiskelijat, jotka aloittavat ohjelmoinnin vasta Tikoon tullessaan, kokevat ohjelmointikurssit hyvin vaikeiksi. He joutuvat oppimaan kielen, jossa kielioppi- eli syntaksivirheet ja kirjoitusvirheet ovat etenemisen usein täysin pysäyttäviä tilanteita. Tämän lisäksi heidän tulee oppia uudenlainen ajattelutapa, joka vaatii ongelmanpaloittelukykyä ja tiedonhakutaitoa.

Extreme programming (lyhennettynä XP) on ohjelmistokehityksen ajattelutapa, joka kuuluu ketterän kehityksen menetelmiin ja se erottuu muista sisältämällä ohjelmointiin ja suunnitteluun liittyviä käytäntöjä. Osa näistä käytännöistä on koettu niin hyviksi, että ne ovat levinneet myös XP:n ulkopuolelle. Tutkimuksen tarkoituksena on selvittää, miten XP:n käytäntöjä voitaisiin hyödyntää Tikon ohjelmistokehityksen opetuksessa, jotta opiskelijoiden oppiminen helpottuisi ja he voisivat toimia paremmin ohjelmistokehitysprojekteissa työelämässä.

XP:tä on käytetty erilaisina versioina yksittäisissä korkeakouluissa, mutta se ei ole vielä saanut laajempaa käyttöönottoa. Näistä toteutuksista on tehty tieteellisiä julkaisuja, joten tämä tutkimus keskittyy tuottamaan kehitysehdotuksen juuri Tikolle. Ehdotuksen on tarkoitus helpottaa opiskelijoiden oppimista ja tätä kautta edistää asiantuntijuuden kehittymistä työelämässä.

2 Tutkimusasetelma

2.1 Tutkimuksen kohteet ja tutkimusongelman löytyminen

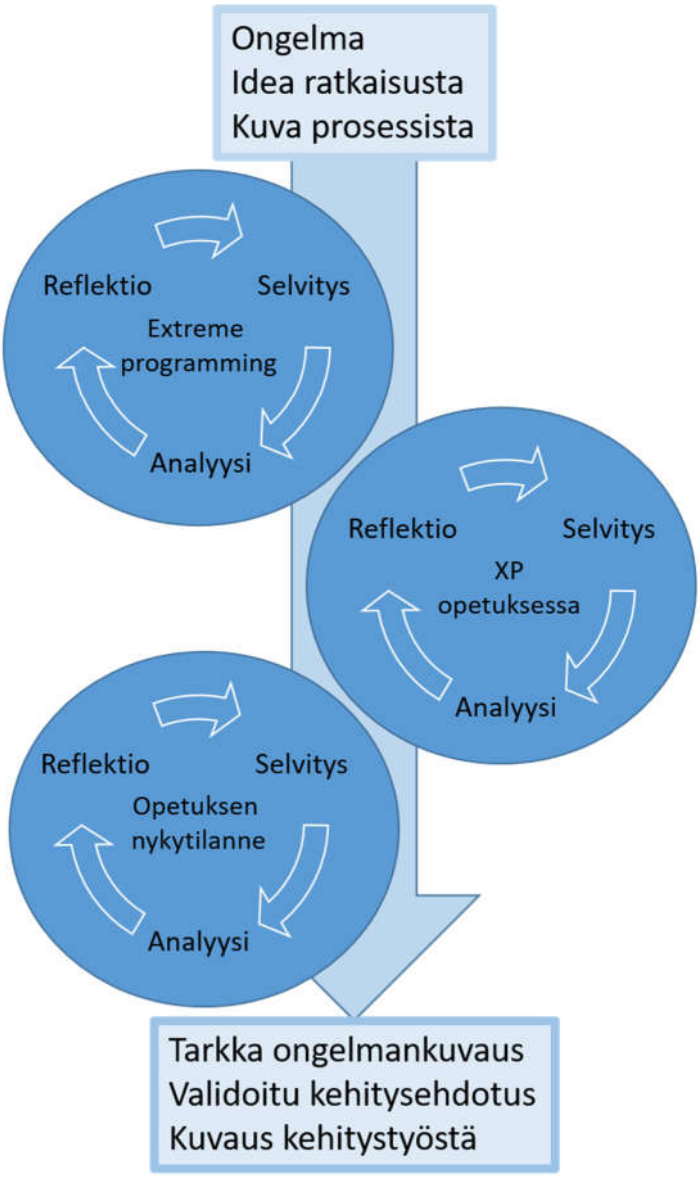
Jyväskylän Ammattikorkeakoulussa (JAMK) opiskelee n. 8 500 opiskelijaa, joista vuosittain valmistuu n. 1 500. JAMK tarjoaa n. 30 eri tutkintoa, joista yksi on tradenomin tutkinto tietojenkäsittelyn tutkinto-ohjelmasta. (AMK-tutkinnot n.d.) Tikon tavoite on

tuottaa yritysmaailman tarvitsemia osaajia, joilla on kyky tuottaa ICT-alan ohjelmistoja ja palveluja.

Työn tekijä on osallistunut Tikon opetukseen opiskelijana syyskuusta 2014 lähtien ja 1.2015–6.2016 välisenä aikana hän on toiminut kertauksen ohjaajana. Syksystä 2016 lähtien hän on osallistunut tuntiopettajana Tikon ohjelmointi-, pelinkehityksen teknologiat- ja tuotekehitysprojekti -kursseihin. Tänä aikana hän on havainnut ja tuonut esille opiskelijoiden oppimisvaikeuksia erityisesti ohjelmoinnin osalta. Näiden huomioiden kautta tutkimusongelmaksi määrittyi: miten helpottaa opiskelijoiden ohjelmoinnin ja ohjelmistokehityksen oppimista.

2.2 Tutkimusmenetelmä ja -kysymykset

Opinnäytetyö toteutettiin kehittämistutkimuksena. Kehittämistutkimus on monimenetelmäinen tutkimusote, jonka tarkoituksena on tuottaa muutos parempaan (Kananen 2012, 19). Kehittämistutkimuksen ja toimintatutkimuksen erot eivät ole täysin selviä, koska englanninkielisissä tiedejulkaisuissa se on usein saman action research -termin alla (Kananen 2012, 42). Kehittämistutkimuksen ja yleisen toiminnan kehittämisen erona on tutkimuksellisuus. Tutkimuksellisuuteen kuuluu alkutilanteen tarkka selvittäminen, teoriaan perustuva muutos ja kriittinen arviointi muutoksen vaikutuksista. (Kananen 2012, 21–23.) Kehittämistutkimus alkaa yleensä havaitusta ongelmasta ja alustavasta ajatuksesta sen ratkaisemiseksi (Edelson 2002, 5). Lisäksi on olemassa jonkinlainen kuva, miten kehittäminen toteutetaan. Verrattuna muihin tutkimuksen muotoihin kehittämistutkimuksessa nämä asiat kuitenkin muokkautuvat tasaisin väliajoin suoritettavissa reflektioissa, joissa uuden hankitun tiedon pohjalta niitä muokataan tarkemmiksi. Näin lopputuloksena saadaan kokonaiskuva kehitysprosessista, kokonainen ongelmakuvaus ja ratkaisu. (Edelson 2002, 2–3.)



Kuvio 1 Tutkimuksen toteutus

Tutkimus toteutettiin kolmessa syklissä, jotka koostuivat selvityksestä, analyysistä ja reflektiosta (kuvio 1). Selvitysvaiheessa käytiin läpi syklin aiheeseen liittyvä aineisto. Analyysivaiheessa aineiston pohjalta tehtiin tutkimuksen aiheeseen liittyviä päätelmiä ja tulkintoja. Lopuksi reflektion avulla selvitettiin syklin aikana saadun tiedon aiheuttamat muutokset ja tarkennukset ongelmankuvaukseen ja ratkaisuehdotukseen sekä kirjattiin ylös kehitysprosessin eteneminen.

Luvussa 3 käydään läpi ensimmäinen sykli, jossa selvitettiin ohjelmoinnin opetuksen nykytilanne Tikossa. Toinen sykli tutki Extreme programmingia ja syklin tuloksena syntyi luku 4. Sen jälkeen tutkittiin miten Extreme programmingia ja sen osia on käytetty ohjelmoinnin ja ohjelmistokehityksen opetuksessa (luku 5). Tutkimuksen

tuottama uudistamisehdotus ja sen perustelut sekä analyysi ovat luvussa 6. Näiden lukujen jälkeen on luku 7, joka reflektoi tutkimusta toteutusta.

Tutkimuskysymyksinä toimivat:

- Mitä on Extreme Programming?

- Miten Extreme Programmingilla voitaisiin kehittää Tietojenkäsittelyn tutkinto-ohjelman ohjelmoinnin opetusta?

Ensimmäiseen kysymykseen vastataan luvussa 4 ja toiseen kysymykseen kehitysehdotuksella luvussa 6.

3 Ohjelmistokehityksen opetuksen nykytilanne Tikossa

Tikossa ohjelmistokehitykseen liittyviä kursseja on viisi, jotka jakautuvat kaikille Tikon opiskelijoille yhteisiin perusopintoihin, vaihtoehtoisiin linjaopintoihin ja vaihtoehtoisiin ammattiopintoihin (Taulukko 1). Vaihtoehtoinen linjaopinto tarkoittaa, että opiskelijan pitää sisällyttää joku linjaopinnoista opintoihinsa. Tämän lisäksi on kaikille opiskelijoille pakollinen Tuotekehitysprojekti (40 op), joka tunnetaan myös Ticorporate-nimellä. Tuotekehitysprojektin kanssa samaan aikaan pidetään myös Ohjelmistotuotanto-kurssi (Opintojen rakenteet n.d.)

Taulukko 1 Ohjelmointikurssit Tikossa

Kurssikoodi	**Kurssinnimi**	**Opintopisteet**	**Kurssin tyyppi**
HTO10104	Ohjelmointi	7 op	Yhteiset perusopinnot
HTO10105	Web-sovelluskehitys	8 op	Vaihtoehtoiset linjaopinnot *
HTP10800	Pelikehityksen teknologiat	8 op	Vaihtoehtoiset linjaopinnot *
HTO10106	Web-sovelluskehitys 2	4 op	Vaihtoehtoiset ammattiopinnot
HTS30110	Java EE sovelluskehitys	5 op	Vaihtoehtoiset ammattiopinnot

Ohjelmointi-kurssin sisältönä on ohjelmoinnin ja olio-ohjelmoinnin perusteet. Web-sovelluskehitys sisältää javascriptiä ja sen yleisimpiä sovelluskehyksiä (framework) ja Web-sovelluskehitys 2 syventää ja laajentaa näitä sisältöjä. Pelikehityksen teknologiat -kurssilla käydään läpi Phaser- ja Unity-pelimoottoreiden käytön lisäksi hiukan HTML5:llä pelintekemisen alkeita. Java EE sovelluskehityksessä opetetaan

miten Java EE:llä voi tehdä web-sovelluksia ja isomman mittakaavan ohjelmistoja. Näillä viidellä kurssilla on hyvin samanlaiset opetusmetodit. Kontaktitunnit sisältävät opettajan esittämää teoriaa ja vaihtelevissa määrin harjoitusten tekoa. Kontaktien lisäksi opiskelijoiden tulee tehdä harjoituksia omalla ajallaan. Vaikka opiskelijoilta ei vaadita yksilötyöskentelyä, suurin osa heistä työskentelee yksin. Arviointi perustuu joko arviointikeskusteluun tai harjoitustyöhön. Arviointikeskustelussa opiskelija esittelee tekemänsä harjoitukset ja opettaja selvittää esittelyn ja esittämiensä kysymyksien pohjalta opiskelijan taitotason. Harjoitukset palautetaan henkilökohtaisesti, mutta tehtäviä saa tehdä kaverin kanssa tai ryhmissä. Harjoitustyö esitellään seminaarissa, jonka aikana opiskelija esittelee tuotoksensa ja sen ohjelmoinnillisesti haastavimmat osat. Opettajat esittävät myös kysymyksiä, joiden pohjalta selvitetään, kuinka hyvin opiskelija ymmärtää esittelemäänsä harjoitustyötä.

Ticorporate (tuotekehitysprojektin yrityssimuulation nimi) on toisen vuoden opintoina tehtävä sovelluskehitysprojekti, jossa opiskelijat tuottavat 5–7 hengen ryhmissä pelin tai web-sovelluksen. Opiskelijat työskentelevät projektissa tiistaista perjantaihin ja heidän työaikansa on 25 tuntia viikossa. Opettajat ovat ohjaamassa tarvittaessa ja järjestävät opetusta.

Ticorporaten toteutus on vaihdellut vuosi vuodelta opiskelijapalautteiden ja opettajien huomioiden pohjalta. Perusrunko on kuitenkin pysynyt viimeiset vuodet suunnilleen seuraavana. Ticorporaten alussa pidetään opetusta ideoinnista ja konseptoinnista, ohjelmistokehityksestä ja mahdollisista teknologioista. Samaan aikaan opiskelijoiden kanssa käydään opiskeluun liittyvät kehityskeskustelut, joiden avulla he myös selvittävät mitä he tekevät tulevassa projektissaan. Samaan aikaan opiskelijat ideoivat mahdollisia tuotteita, joista osa valitaan jatkoon äänestyksellä. Näiden ideoiden ympärille muotoutuu pieniä ryhmiä, jotka tuottavat ideasta laajemman konseptin ja suunnitelman mitä siitä toteutetaan Ticorporaten aikana. Nämä konseptit esitellään ja niistä valitaan toteutettavat projektit. Jatkoon päässet konseptit kertovat, mitä osaamista he tarvitsevat. Opiskelijat, joiden konsepteja ei valittu, ilmoittavat missä projekteissa he mieluiten työskentelevät ja mitä he haluavat tehdä. Näiden tietojen pohjalta opettajat luovat projektiryhmät.

Seuraavaksi alkaa prototyyppivaihe, jonka aikana projektiryhmät voivat testata teknisiä ratkaisuvaihtoehtoja ja tehdä alustavan suunnitelman miten he konseptinsa toteuttavat. Tämä vaihe kestää kolmesta neljään viikkoa syysloman alkuun asti. Syysloman jälkeen käynnistyy tuotantovaihe, joka kestää kevään loppuun asti.

Opiskelijoita ohjeistetaan käyttämään Scrum-viitekehystä projektinhallintaan. Scrum on kevyt prosessimalli erilaisten alojen projektien hallintaan ja näin ollen ei ota kantaa työskentelytapoihin. Opiskelijoilta vaaditaan myös versionhallinnan käyttöä. Opiskelijoita ei kuitenkaan vaadita noudattamaan mitään ohjelmointikäytänteitä tai muita työskentelytapoja.

Ohjelmistotuotanto järjestetään Ticorporaten kanssa samaan aikaan. Kurssilla annetaan laaja yleiskatsaus ohjelmistotuotantoon. Kurssiin sisältyy muun muassa ohjelmistojen elinkaari, vaatimusten hallinta, projektin suunnittelu ja ohjaus sekä tiimityöskentely. Ohjelmistotuotannossa esitellään myös Ticorporaten kevyen mallin vaihtoehtona perinteisempiä ohjelmistokehityksen malleja ja käydään läpi milloin mitäkin kannattaa soveltaa.

Analyysi

Ohjelmointikurssien arviointikeskustelu on hyvä ja työelämään valmistava tapa arvioida opiskelijoiden osaamista. Alkeiskurssilla on ollut hyvä tasapaino teorian ja käytännön tekemisen välillä. Web-sovelluskehityksen molemmissa versioissa on sisällön laajuudesta johtuen teorialla suuri ylivalta ja opiskelijoilla on hankalaa saada tukea käytännön tekemiseen. Kurssit eivät erityisemmin tue hyviä ohjelmointikäytänteitä ja tapoja. Niitä kommentoidaan mahdollisesti arviointikeskusteluissa, jolloin palauteväli on suuri ja ohjeistus sekä tuki ovat vähäistä.

Ticorporate on korkeakoulumaailmassa harvinainen kokopäiväinen pitkäaikainen työelämän projektityötä simuloiva kurssi. Opiskelijoiden itsensäkin mukaan he eivät kuitenkaan saa tarpeeksi tukea ja opetusta työntekemiseen. Kurssi vaatii itseohjautuvia opiskelijoita toimiakseen, mutta opiskelijoita ei tueta tämän taidon hankkimisessa. Tarvitaan siis enemmän ohjausta ja selkeämpää prosessia työskennellä.

4 Extreme programming

Extreme programming sisältää arvoja (values), periaatteita (principles) ja sääntöjä (rules) tai käytäntöjä (practices). Ne kuvaavat XP:tä eri abstraktiuden tasoilla, arvojen ollessa abstraktein versio sekä lähtökohta josta muut sisällöt on johdettu ja voidaan tarvittaessa johtaa uudelleen (Beck & Andres 2004, 14). Käytettäessä Extreme programmingia projekteissa XP:n arvot eivät muutu, mutta niiden joukkoon saatetaan lisätä yrityksen tai tiimin tärkeäksi kokemia muita arvoja. Jos näitä lisäyksiä tulee, käytännöt muuttuvat, koska ne johdetaan arvoista (Beck & Andres 2004, 22). Arvot ovat yleisiä eli niiden mukaan voi toimia missä tahansa tilanteessa elämässään. Käytännöt ovat vahvasti kontekstisidonnaisia, jolloin ne saattavat vaihtua projektista toiseen. Abstraktiotasojen suuri eroavaisuus vaatii väliin periaatteet, jotka yhdistävät nämä kaksi ääripäätä. Periaatteet ovat sidottuja ohjelmistokehityksen kontekstiin. (Beck & Andres 2004, 14–15.) Ne ovat kuitenkin tarpeeksi yleisiä, jotta niiden avulla voidaan luoda uusia käytäntöjä erikoisiin tilanteisiin (Beck & Andres 2004, 34). Extreme programmingin prosessin mukaan toimiminen vaatii, että jokainen tiimin jäsenen ymmärtää nämä arvot, periaatteet ja käytännöt ja toimii niiden mukaan.

4.1 Arvot

Extreme programming arvot ovat (Beck & Andres 2004, 18) Communication (Kommunikaatio), Simplicity (Yksinkertaisuus), Feedback (Palaute), Courage (Rohkeus) ja Respect (Kunnioitus). Kommunikaatio tarkoittaa esimerkiksi tiedon jakamista tiimin kesken, päätöksien tekemistä yhdessä keskustellen ja ongelmatilanteissa tehtävää pohdintaa olisiko tilanteen voinut välttää jollakin kommunikaatiolla. Tarkoituksena on saada projektin jäsenet kokemaan itsensä tiimiksi ja luomaan tehokasta yhteistyötä jäsenten välillä. (Beck & Andres 2004, 18) Yksinkertaisuus arvona edellyttä, että ongelmanratkaisussa keskitytään tämän päivän ongelmien ratkaisuun, sekä etsimään yksinkertaisinta toimivaa ratkaisua (Beck & Andres 2004, 18). Tämä varmistaa, että ohjelmisto on aina arvokkain mahdollinen, koska halutut ominaisuudet saavutetaan vähimmällä koodilla (Wells 2009).

Palaute-arvon mukaan toimimiseen kuuluu esimerkiksi se, että tiimi esittelee ohjelmistoaan asiakkaalle usein saadakseen palautetta ja muokkaa suunnitelmiaan

sen pohjalta (Wells 2009). Yleisesti tarkoituksena on pienentää validoinnin aikaa pienemmäksi kaikilla tasoilla (Beck & Andres 2004, 20). Rohkeutta tarvitaan esimerkiksi rehelliseen kommunikaation ja toimimattomien ratkaisujen hylkäämiseen (Beck & Andres 2004, 21). Viimeisenä arvona on kunnioitus, jonka takia kaikki muut arvot toimivat. Tiimin jäsenten tulee kunnioittaa muita jäseniä ja kaikkia projektiin kuuluvia, jotta projekti voi onnistua. (Beck & Andres 2004, 21.)

4.2 Periaatteet

Extreme programmingin periaatteet ovat saaneet vähemmän huomiota kuin XP:n arvot ja käytännöt, mutta niillä on tärkeä rooli tiimin kehittäessä XP:stä omaa versiotaan (Fowler 2003). Fowlerin (2003) mukaan periaatteet esiteltiin alunperin Kent Beckin Extreme programming explained -kirjan ensimmäisessa painoksessa, jossa listattiin seuraavat perusperiaatteet:

- Rapid feedback (Nopea palaute)
- Assume simplicity (Oleta yksinkertaisuutta)
- Incremental change (Pienet toistuvat muutokset)
- Embracing Change (Muutoksen hyväksyminen) ja
- Quality Work (Laadukas työ).

Nopea palaute voimistaa palaute-arvoa suosimalla käytäntöjä, joissa palaute saadaan mahdollisimman pian. Erityisen tärkeää on palaute asiakkaalta kehittäjälle ja tietokoneelta kehittäjälle. **Oleta yksinkertaisuutta** muistuttaa, että ohjelmiston vaatimukset ja projektin tila tulevat muuttumaan arvaattomilla tavoilla tulevaisuudessa. Tämän takia ei kannata suunnittella monimutkaisia rakenteita ja toteutuksia, koska todennäköisesti ”et tule tarvitsemaan sitä.” Toinen tähän periaatteeseen liittyvä lausahdus on ”älä toista itseäsi”, joka ohjaa kohti yksinkertaisinta rakennetta poistamalla toiston koodista. **Pienet toistuvat muutokset** sanoo, että muutokset kaikilla projektin osa-alueilla kannattaa paloitella pieniksi muutoksiksi. Tämä vähentää epäonnistumisen riskiä, koska muutokset ovat selkeämmin hahmotettavissa ja ymmärrettävissä. Samalla se myös nopeuttaa muutosta, koska pienten askelten ottaminen on nopeaa. **Muutoksen hyväksyminen** tarkoittaa, että paremmat ratkaisut jättävät enemmän ovia auki, jotta ohjelmisto ja tiimi pystyvät reagoimaan helpommin muutokseen. **Laadukas työ** muistuttaa, että

ihmiset nauttivat tehdessään hyvää työtä ja tällöin työskentelevät tehokkaammin. (Extreme programming principles n.d.)

Näiden perusperiaatteiden lisäksi listattiin vähemmän keskeiset, mutta silti tärkeät lisäperiaatteet: (Fowler 2003)

- Teach learning (Opeta oppimista)
- Small initial investment (Pieni alkuinvestointi)
- Play to win (Pelaa voittaaksesi)
- Concrete experiments (Kokeile konkreettisesti)
- Open, honest communication (Avoin ja rehellinen kommunikaatio)
- Work with people's instincts - not against them (Toimi ihmisten luontaisten taipumusten mukaisesti, eikä niitä vastaan)
- Accepted responsibility (Hyväksytty vastuu)
- Local adaptation (Tiimikohtainen versio)
- Travel light (Matkusta kevyesti) ja
- Honest measurement (Rehellinen mittaaminen).

Opeta oppimista muistuttaa, että kehittäjille on parempi antaa strategioita, joiden avulla he voivat tehdä oman tilanteen sopivan päätöksensä kuin absoluuttisia totuuksia. Esimerkiksi testipohjainen kehitys on strategia, joka tuottaa kattavat testit ilman määräyksiä, että pitää olla tietty määrä testejä kutakin koodiriviä kohti. **Pieni alkuinvestointi** ohjaa aina valitsemaan asiakkaalle tärkeimmät ja tuottamisen kannalta halvimmat toiminnallisuudet, jotta maksimoidaan ohjelmiston arvo suhteessa käytettyyn rahaan. **Pelaa voittaaksesi** ohjaa työskentelemään täysillä projektin ja tiimin eteen, eikä vain välttämään henkilökohtaisia epäonnistumisia.

Päätökset ilman testattua tietoa ovat aina arvauksia ja niihin liittyy siten aina isompi riski. Näiden riskien välttämiseksi **kokeile konkreettisesti** -periaate ohjaa testaamaan käytännössä vaihtoehtoja ennen päätöstä. Tiimin pitää pystyä kommunikoimaan avoimesti ja rehellisesti, muuten tiimityöskentely ei toimi. On tärkeää tunnistaa ihmisten luontaiset taipumukset ja luoda käytäntöjä, joiden mukaan kehittäjien on luontaista toimia. Esimerkiksi kiire houkuttaa ihmisiä käyttämään oikoteitä, jotka heikentävät koodin laatua. Käytäntöjen pitää olla houkuttelevia myös kiireen alla eli ne eivät saa hidastaa työtehoa ja kehittäjän pitää uskoa, että se on nopein reitti valmiiseen tuotteeseen.

Hyväksytty vastuu kertoo, että oikeaa vastuuta ei voida määrätä, se voidaan vain ottaa vastaan. Määrätty vastuu, varsinkin mahdottomista tehtävistä, aiheuttaa turhautumista, joka tulee vaikuttamaan tiimin ja henkilön työhön. **Tiimikohtainen versio**

muistuttaa, että XP ei voi koskaan toimia suoraan, vaan tiimin pitää luoda sen esittämien ajatusten avulla oma versionsa. **Matkusta kevyesti** vaatii, että prosessi on kevyt ja sen vaatimat toimet ovat yksinkertaisia ja projektin etenemisen kannalta arvokkaita. Tiimin pitää pystyä mukautumaan nopeasti vaihtuviin tilanteisiin. **Rehellinen mittaaminen** korostaa rehellisyyden tärkeyttä erityisesti arvioinnissa, jossa tarkkojen arvojen antaminen on mahdotonta. Myös mitattavat asiat on tärkeää valita huolella, jotta mittaustulokset ovat hyödyllisiä. Esimerkiksi kirjoitettujen koodirivien määrä ei ole järkevä mittari tehokkuudelle alalla, jossa parempi rakenne usein vähentää rivien määrää huomattavasti ja uudet ominaisuudet usein lisäävät rivien määrää. (Principles of XP n.d.)

Beckin ja Andresin (2004) viisi vuotta myöhemmin julkaisema toinen painos esittelee seuraavat periaatteet:

- Humanity (Inhimillisyys)
- Economics (Talous)
- Mutual Benefit (Yhteinen etu)
- Self-similarity (Samankaltaisuus)
- Improvement (Kehittyminen)
- Diversity (Monimuotoisuus)
- Reflection (Pohdiskelu)
- Flow (Virtaus)
- Opportunity (Mahdollisuus)
- Redundancy (Päällekkäisyys)
- Failure (Epäonnistuminen)
- Quality (Laatu)
- Baby Steps (Pienet askeleet)
- Accepted Responsibility (Hyväksytty vastuu)

Inhimillisyys muistuttaa ottamaan huomioon, että ohjelmistokehittäjät ovat ihmisiä, joilla on inhimillisiä tarpeita (Beck & Andres 2004, 24). **Talous**-periaatteen tarkoitus on pitää tiimin jäsenten mielessä, että kaiken mitä he tekevät tulee tuottaa mahdollisimman paljon arvoa yritykselle. Tämä tarkoittaa esimerkiksi sitä, että ohjelmaan toteutetaan tärkeimmät ominaisuudet ensin, jolloin vähimmällä työmäärällä ollaan saatu arvokkain ohjelma. Samalla tulee tiedostaa, että ohjelman ylläpidettävyys ja testit ovat myös arvokkaita asioita yritykselle, koska tällöin ohjelmaa voidaan helposti laajentaa. (Beck & Andres 2004, 25.)

Paineen alla monia houkuttaa valita ratkaisu, joka on helpoin nyt, vaikka se aiheuttaisi ongelmia tulevaisuudessa. **Yhteinen etu** ohjaa valitsemaan käytäntöjä,

jotka helpottavat asioita nyt ja tulevaisuudessa. Esimerkiksi ohjelmoija kirjoittaa testit ennen ohjelmointia, koska tämä käytäntö helpottaa suunnittelua sillä hetkellä. Lisäksi testit jäävät tulevaisuuden ohjelmoijien käyttöön, jotta he voivat tarkistaa, että heidän tekemänsä muutokset eivät rikkoneet mitään vanhoja toiminnallisuuksia. (Beck & Andres 2004, 26.)

Samankaltaisuus viittaa siihen, että samanlainen ratkaisu voi toimia hyvinkin erilaisissa konteksteissa (Beck & Andres 2004, 27). **Kehittyminen** muistuttaa, että tänään tulee tehdä mahdollisimman hyvää työtä samalla kun etsii keinoja toimia huomenna paremmin (Beck & Andres 2004, 28). **Monimuotoisuus** haluaa mahdollisimman erilaisia näkökulmia tiimiin, koska silloin jokainen tiimin jäsen huomaa erilaisia ongelmia ja esittää erilaisia ratkaisuja (Beck & Andres 2004, 29). **Pohdiskelun** tarkoituksena on työn tekemisen lisäksi analysoida työn tekemistä, oppia virheistä ja etsiä onnistumisen syitä (Beck & Andres 2004, 29).

Virtaus viittaa siihen, että ohjelmistokehityksen tulisi tuottaa tasaisesti kehittyvä ohjelmisto tasaisena virtauksena isojen kokonaisuuksien sijaan. Tämä toimintatapa mahdollistaa kehityksen tarkemman ohjauksen, koska palautetta saadaan useammin. (Beck & Andres 2004, 30.) **Mahdollisuus** muistuttaa ottamaan vastaantulevat ongelmat mahdollisuuksina oppia ja kehittyä (Beck & Andres 2004, 31). Jotkin ongelmat ohjelmistokehityksessä ovat niin kriittisiä ja vaikeita, että yksittäinen käytäntö tai ratkaisu riittää vain vähentämään vaikutuksia. Extreme programmingin vastaus on **Päällekkäisyys** eli toimitaan useammman käytännön mukaan, jotka kaikki auttavat ongelman hallinnassa eri näkökulmasta. (Beck & Andres 2004, 31.)

Epäonnistuminen ei tarkoita, että tiimin tulisi tahallaan epäonnistua kaikessa. Vaan kun ei ole tarpeeksi tietoa, niin riskeerataan epäonnistumista toteuttamalla nopeasti useampi ratkaisu ja näistä toteutuksilla saadulla tiedolla voidaan tehdä parempi päätös. (Beck & Andres 2004, 32.) **Laatu** muistuttaa, että laadun heikentäminen ei ole koskaan toimiva ratkaisu. Päinvastoin laadun parantamisella on usein positiivinen vaikutus myös muihin projektin ominaisuuksiin, kuten tuottavuuteen. (Beck & Andres 2004, 33.)

Isotkin muutokset kannattaa tehdä **pienillä askelilla**. Tämä vähentää epäonnistumisen riskiä ja helpottaa yksittäisen askeleen tekemistä, jolloin

muutosnopeus saattaa olla jopa nopeampi kuin isoilla askelilla. (Beck & Andres 2004, 33.) **Hyväksytty vastuu** on säilynyt sellaisenaan ensimmäisestä painoksesta.

Analyysi

Extreme programming explained -kirjan ensimmäisen ja toisen painoksen periaatteissa on muutamia samoja tai lähes samoja periaatteita: hyväksytty vastuu, pienet askeleet ja laatu. Painoksien välissä muutama periaate on jätetty pois ja muutama uusi on lisätty listalle. Taulukossa 2 kuvataan miten periaatteet liittyvät toisiinsa.

Taulukko 2 Periaatteiden yhteydet painosten välillä

Ensimmäinen painos	**Toinen painos**
Nopea palaute	Kehittyminen, pienet askeleet, pohdiskelu
Oleta yksinkertaisuutta	Kehittyminen
Pienet toistuvat muutokset	Pienet askeleet, virtaus
Muutoksen hyväksyminen	Mahdollisuus
Laadukas työ	Inhimillisyys, talous, laatu
Opeta oppimista	Kehittyminen, pohdiskelu
Pieni alkuinvestointi	Talous
Pelaa voittaaksesi	Inhimillisyys, talous
Kokeile konkreettisesti	Epäonnistuminen
Avoin ja rehellinen kommunikaatio	Inhimillisyys, pohdiskelu
Toimi ihmisten luontaisten taipumusten mukaisesti, eikä niitä vastaan	Inhimillisyys, yhteinen etu
Hyväksytty vastuu	Hyväksytty vastuu
Tiimikohtainen versio	
Matkusta kevyesti	
Rehellinen mittaaminen	
	Samankaltaisuus
	Monimuotoisuus
	Päällekkäisyys

Ensimmäisen painoksen periaatteet ovat yksinkertaisemmin nimettyjä, kun taas toisen painoksen nimet ovat abstraktimpeja käsitteitä, joiden tarkka merkitys kaipaa enemmän selitystä. Periaatteita harvemmin mainitaan puhuttaessa Extreme programmingista ja kun mainitaan sana principle, niin yleensä puhutaan arvoista tai käytännöistä. Yleensä puhe on silloin arvoista tai periaatteiden tyylisistä asioista, mutta ei näistä termeistä. Tämä periaatteiden heikompi tuntemus voi osittain selittää miksi XP ei ole saavuttanut suurempaa käyttöastetta. Monet yritykset käyttävät useimpia XP:n käytäntöjä, mutta niiden toimiva käyttäminen vaatisi näiden periaatteiden omaksumista tai edes tuntemista ja tietoista mukauttamista yrityksen sisällä.

4.3 Käytännöt

Yleisesti käytetyin käytäntölista sisältää 12 käytäntöä jaettuna 4 kategoriaan ja se on esitetty taulukkossa 3 (Extreme programming core practices 2011). Näiden lisäksi löytyy paljon lisäkäytäntöjä, jotka tukevat ja auttavat toteuttamaan pääkäytäntöjä (Beck & Andres 2004, 61–69; Extreme programming core practices 2011, Support practices.) Lisäksi on esitelty Extreme programming säännöt, jotka ohjaavat samaan toimintaan kuin käytännötkin esitystavan ollessa erilainen (Wells 1999).

Taulukko 3 Extreme programmingin 12 käytäntöä.

- Fine Scale Feedback (Tarkka palaute)
    - Test Driven Development (Testivetoinen kehitys)
    - Planning Game (Suunnittelupeli)
    - Whole Team (Koko tiimi)
    - Pair Programming (Pariohjelmointi)
- Continuous Process (Jatkuva prosessi)
    - Continuous Integration (Jatkuva integraatio)
    - Design Improvement (Rakenteen kehittyminen)
    - Small Releases (Pienet julkaisut)
- Shared Understanding (Yhteinen ymmärrys)
    - Simple Design (Yksinkertainen rakenne)
    - System Metaphor (Ohjelmiston metafora)
    - Collective Code Ownership (Koodin yhteisomistajuus)
    - Coding Standard (Ohjelmointikäytännöt)
- Programmer Welfare (Kehittäjien hyvinvointi)
    - Sustainable Pace (Ylläpidettävä tahti)

Vaikka Refaktorointi ei ole mainittu nimeltä tässä käytäntölistassa, se on tärkeä taito Extreme programmingia toteutettaessa, koska moni XP:n käytäntö vaatii sitä.

Refaktoroinnilla tarkoitetaan koodin muokkaamista ilman että sen toiminnallisuus muuttuu. Tarkoituksena on parantaa luettavuuttaa, uudelleenkäytettävyyttä ja muokattavuutta (Fowler 1999, 53–54).

Tarkka palaute

Testivetoinen kehitys (TDD) on yksi Extreme programmingin käytännöistä, joka on levinnyt XP:n ulkopuolelle (Sanches, Williams & Maximilien 2007, 1). TDD:ssä ohjelmoija kirjoittaa testin, joka testaa pientä osavaatimusta ohjelmistolta, kirjoittaa vain sen verran koodia, että testi menee läpi ja lopuksi poistaa mahdollisen toiston refaktoroimalla (Beck 2002, 1). **Suunnittelupeli** sisältää julkaisusuunnittelun (Release planning) ja iteraatiosuunnittelun (Iteration planning). Julkaisusuunnittelun tarkoituksena on tuottaa ja ylläpitää arviota siitä milloin projektin eri toiminnallisuudet valmistuvat. Suunnitelmaan kuuluu osana tärkeysjärjestyksessä oleva toiminnallisuuslista ja arviot toiminnallisuuksien vaikeudesta. Suunnitelmassa olevat ajat eivät ole määräaikoja valmistumisille, vaan ne ovat sen hetken parhaita arvioita. Iteraatiosuunnittelu tapahtuu jokaisen muutaman viikon pituisen iteraation alussa. Tällöin kehittäjät paloittelevat tärkeimmät tekemättömät toiminnallisuudet ja arvioivat niiden tekemiseen kuluvan ajan. Ottaen huomioon edellisen iteraation työmäärän, tiimi ottaa vastuulleen sopivan määrän työtä. (Jeffries 2011, Planning Game.)

Koko tiimi -käytäntö ohjaa, että kaikki tiimin jäsenet istuvat samassa tilassa, mukaan lukien Asiakas (ohjelmiston lopullinen käyttäjä tai ainakin tuntee läheisesti ohjelmiston käyttöympäristön). Projektin vaatimuksien muuttuessa tiimin kokoonpano saattaa muuttua projektin aikana. Esimerkiksi alussa tarvitaan testaajaa, joka auttaa asiakasta tekemään hyväksyntätestit toiminnallisuuksille, mutta kun asiakas on oppinut luomaan testit itse, testaaja voi siirtyä toiseen tiimiin tai ottaa vastuulleen muita tehtäviä, jotka sopivat hänen osaamisiinsa. (Jeffries 2011, Whole Team.)

Pariohjelmointi on toinen Extreme programmingin ulkopuolelle levinnyt käytäntö ja sitä on tutkittu opetusympäristöissä erillään muista käytännöistä. Tämän käytännön mukaan kaikki koodi kirjoitetaan parin kanssa ja tavoitteena on paremmin suunnitteltu ja testattu koodi. Pari myös vaikeuttaa XP:n käytännöistä lipsumista ja

tukee oikeaa toimintaa valvomalla työskentelyä. Pareja tulee myös vaihtaa sopivin väliajoin, yleensä parin tunnin välein, jotta tieto ja systeemin ymmärrys leviää koko tiimiin. (Beck & Andres 2004, 42.)

Jatkuva prosessi

Integraatio on tärkeä ja usein vaikea osa ohjelmistokehitystä. Integraatiolla tarkoitetaan tilannetta, jossa useampi ohjelmoija on muokannut samaa koodia eri syistä ja nämä muutokset pitäisi saada yhdistettyä toimivaksi versioksi. **Jatkuva integraatio** kertoo, että yhdistämällä parien koodi usein vaikeuksien määrä vähenee, koska muutokset ovat vähäisiä. (Beck & Andres 2004, 50.) Ohjelmistokehityksessä on ollut pitkään vallalla ajatus, että ohjelmisto pitää suunnitella mahdollisimman tarkkaan etukäteen, koska muutokset ovat kalliita. XP:n yhtenä päämääränä on, että muutokset olisivat helppoa ja halpoja toteuttaa. **Rakenteen kehittyminen** muistuttaa kehittäjiä pitämään mielessä systeemin rakenteen ja kun ymmärrys kehittyvästä ohjelmistosta kasvaa ja muuttuu, niin rakennetta parannetaan refaktoroimalla pienin askelin. (Beck & Andres 2004, 52–53.) **Pienet julkaisut** viittaa joka iteraation lopussa tapahtuvaan julkaisuun Asiakkaalle ja loppukäyttäjille tapahtuviin julkaisuihin. Testauksen ja integraatioon liittyvät käytänteet varmistavat, että nämä julkaisut ovat toimivia ja laadukkaita. (Jeffries 2011, Small Releases.)

Yhteinen ymmärrys

Yksinkertainen rakenne toimii läheisessä yhteistyössä Rakenteen kehittymisen kanssa. Projektit aloitetaan yksinkertaisella mutta riittävällä rakenteella, joka ottaa huomioon aina vain nykyiset vaatimukset. **Yksinkertaisuus** mahdollistaa sen, että vaatimuksien muuttuessa tai ymmärryksen kasvaessa muutokset olisivat mahdollisimman halpoja ja nopeita tehdä. **Rakenteen kehittyminen** taas varmistaa, että näiden muutoksien jälkeen rakenne on edelleen yksinkertainen mutta riittävä. (Jeffries 2011, Simple Design.) **Metafora** auttaa tiimin jäseniä ymmärtämään kehitettävän ohjelmiston rakennetta ja sen osien vastuualueita (Jeffries 2011, Metaphor).

Koodin yhteisomistajuus tarkoittaa sitä, että kukaan ei omista mitään koodin osaa, vaan kaikki saavat muokata mitä tahansa kohtaa. Tällöin kehittäjien ymmärrys systeemistä laajenee ja koodi sijoitetaan aina sille kuuluvaan paikkaan. (Jeffries 2011,

Collective Code Ownership.) **Ohjelmointikäytännöt** varmistavat, että kaikki koodi näyttää siltä kuin sen olisi kirjoittanut yksi ihminen. Tällöin kehittäjien on helpompi hahmottaa, mitä koodissa tapahtuu. Tämä vahvistaa koodin yhteisomistajuutta. (Jeffries 2011, Coding Standard.)

Kehittäjien hyvinvointi

Tiimin jäsenet työskentelevät ryhmän ja projektin parhaaksi ja tehokkaimmin tämä onnistuu **ylläpidettävällä tahdilla**, eli tahdilla jota jäsenet voivat tarvittaessa jatkaa loputtomiin. Tähän tahtiin sisältyvät myös lounaat, viikonloput ja vuosilomat. Tämä käytäntö muistuttaa myös, että ylityö ei tehosta työskentelyä, koska ohjelmistokehitys on innovaativista työtä ja väsyneet henkilöt eivät keksi ratkaisuja. (Jeffries 2011, Sustainable Pace.)

5 Extreme programming opetuksessa

Extreme programmingia on tutkittu opetuskäytössä ainakin Pohjois-Carolinan osavaltionyliopistossa, Hannoverin Gottfried Wilhelm Leibniz -yliopistossa ja São Paulon yliopistossa. Näitä käytännön kokemuksia käydään läpi luvussa 5.1. Lisäksi on tehty tutkimuksia, joissa on selvitetty teoreettisemmalla tasolla miten Extreme programmingia voitaisiin opettaa korkeakouluissa ja miten se voisi auttaa opiskelijoita kehittymään paremmiksi ohjelmistokehittäjiksi (ks. Schneider & Johnston 2003 ja Williams & Upchurch 2001). Näitä tutkimuksia analysoidaan luvussa 5.2. Pariohjelmointia on tutkittu paljon opetuskäytössä (esim. Williams, McCrickard, Layman & Hussein 2008) ja niiden tuloksista on myös luotu yleistettyjä ohjeita, miten pariohjelmointia kannattaa toteuttaa opetustilanteissa. Eräs näistä ohjeista käydään läpi tarkemmin luvussa 5.3. Tutkimuksia ei käydä läpi kokonaisuudessaan, vaan niistä nostetaan esille vain opinnäytetyön kannalta olennaiset kohdat.

5.1 Käytännön kokemuksia

Shukla ja Williams (2002) opettivat ohjelmistotuotantokurssilla perinteisempiä ohjelmistokehityksen prosesseja ja Extreme programmingia. Opiskelu tapahtui tekemällä projekteja ja XP:n oppimiseen oli varattu viimeisen neljän viikon aikana tehtävä projekti. Tutkimuksessa selvitettiin opiskelijoiden kokemuksia XP:n eri

käytännöistä. Nämä käytännöt olivat Kent Beckin Extreme programming explained -kirjan ensimmäisen painoksen mukaisia ja ne eroavat jonkin verran aikaisemmin esitetyistä käytännöistä. Alla käytännöistä puhutaan nimillä, jotka esiteltiin luvussa 4.3 tai jos käytännölle ei löydy suoraa yhteneväisyyttä, niin käytäntö esitellään ja sen yhteys luvun 4.3 käytäntöihin kuvataan. Jokaisesta käytännöstä selvitettiin käyttivätkö opiskelijat sitä, pitivätkö he siitä ja onnistuivatko he sen käytössä.

Ohjelmiston metafora koettiin vaikeaksi ja moni ei käyttänyt tätä käytäntöä ollenkaan. Koodin yhteisomistajuus ja yksinkertainen rakenne toimivat hyvin ja yli 90 % opiskelijoita onnistuivat niiden käytössä (Shukla & Williams 2002, 3). Yli puolet opiskelijoista kertoivat, että he eivät refaktoroineet ohjelman rakennetta, mutta tämä selittyy paljolti projektin lyhyellä tuotantoajalla (noin 36 tuntia per opiskelija) (mts. 4).

150 opiskelijan kurssilla opettaja ei voinut toimia asiakkaana kaikkien ryhmien suunnittelupelissä, joten tämä käytäntö simuloitiin tavalla, joka ei käynyt tutkimuksessa ilmi. Kolme neljästä opiskelijasta koki, että suunnittelu oli onnistunut. Kursilla oli ainoastaan yksi julkaisu neljän viikon kuluttua aloituksesta, mutta ryhmille suositeltiin viikoittaisia ryhmän sisäisiä julkaisuja. Puolet opiskelijoista koki onnistuneensa pienen julkaisun käytännön toteutuksessa. (Mts. 4.)

Jatkuva integraatio oli yli 80 prosentille opiskelijoista toimiva käytäntö. Paikalla oleva asiakas on käytäntö, joka sisältyy nykyään koko tiimi -käytännön sisälle. Asiakkaan tulee olla saatavilla ryhmälle koko kehitystyön ajan ja tämä tuottaa usein vaikeuksia opetustympäristössä opettajien ollessa asiakkaana. Tämän kurssin yhteydessä opettaja-asiakas oli aktiivisessa kontaktissa opiskelijoihin sähköpostin ja kurssin keskustelupalstan kautta kontaktien ulkopuolella. 56 prosenttia kokivat tämän toimineen, mikä saattaa viittata siihen, että tämä saatavuus ei välttämättä ollut riittävä. (Mts. 5.)

Suurin osa opiskelijoista koki testauksen onnistuneeksi, mutta he myönsivät, että he usein tekivät testitapaukset vasta jälkikäteen, jolloin testipohjaista kehittymistä ei tapahtunut. Testaukseen liittyi myös hyväksyntätestit. Normaalisti asiakas tuottaa nämä hyväksyntätestit esimerkiksi testaajan avustuksella. Kurssilla opiskelijoiden tuli itse esittää ryhmän käyttäjäkertomuksen kattavat hyväksyntätestit. Kurssin lopuksi

opiskelijoille annettiin arviointiin käytetyt hyväksyntätestit, jotka esittivät asiakkaan palautteen tarkkuutta. Asiakashan pystyy sanomaan tarkasti täyttääkö ohjelmisto tietyn käyttäjäkertomuksen vaatimukset vai ei. (Mts. 5.)

Kolmelle opiskelijalle neljästä pariohjelmointi oli toimiva käytäntö. Pariohjelmointiin tyytymättömistä suurin osa kertoi syyksi aikatauluongelmat tai halun työskennellä kurssin työskentelytilan ulkopuolella. Ohjelmointikäytänteet koettiin toimivaksi ja projektiryhmän ohjelmointia helpottavaksi käytännöksi. 64 prosenttia opiskelijoista koki ylläpidettävän tahdin käytännön toimivaksi, mutta tutkimuksesta ei ilmene, miten opiskelijat käytäntöä toteuttivat. (Shukla & Williams 2002, 6.)

Shuklan ja Williamsin (2002, 6) mukaan yhden lukukauden aikana ei ole aikaa opettaa sekä perinteistä että ketterää ohjelmistokehitystapaa, vaikka kummassakin on alalla toimimiselle tärkeitä taitoja. Tästä syystä Shuklan & Williams (2002, 6) suosittelee käyttämään ohjelmistotuotannon kurssilla näitä kahta ohjelmistokehitystapaa sekoittavaa hybridimallia, jolla toteutetaan projekti.

Stapel, Lübke ja Knauss (2008) esittelevät kokemuksensa XP lab -kurssin kolmesta toteutuksesta vuosina 2005–2007. Kurssi oli valittavissa maisteritason opiskelijoille ja opiskelijat olivat jo tehneet projektin perinteisemmillä ohjelmistokehitystavoilla kandidaattitason opinnoissaan. Näin he pystyivät kokemaan molempien tapojen hyvät ja huonot puolet, jolloin tulevaisuudessa he pystyvät paremmin valitsemaan tilanteeseen sopivan prosessin. Kuten muissakin Extreme programmingiin liittyvissä tutkimuksissa XP määriteltiin käytäntöjen avulla ja siksi kurssin oppimistavoitteetkin keskittyivät käytäntöihin. (Mts. 770.)

Kurssi toteutettiin korkeakoulutasolla harvinaisempana intensiivikurssina. Kurssin alussa pidettiin esittelyluentoja, joissa käytiin läpi XP ja sen käytännöt. Loppukurssi sijoittui peräkkäisille täysille päiville, jolloin opiskelijoilla ei ollut muita kursseja. Nämä päivät oli jaettu useampaan pieneen iteraatioon. Ensimmäisenä vuonna kurssi oli kolmipäiväinen ja iteraatiot olivat puolen päivän pituisia. Toisena vuonna kurssin pituutta muutettiin seitsemään päivän pituiseen iteraatioon. Kolmantena vuonna kurssi kasvoi yhdeksään päivään, joka jakautui yhden päivän prototyypin tekoon ja neljään kahden päivän iteraatioon. Samana vuonna kurssille saatiin myös toinen asiakas henkilökunnasta, jolloin pystyttiin pyörittämään kahta projektia. Nämä

projektit olivat päivän verran eri tahdissa, jotta yksi ohjaaja pystyi osallistumaan molempien projektien suunnittelupäiviin. (Mts. 771.)

Stapelin ja muiden (2008, 773) mukaan täysipäiväinen työskentely koettiin hyvin tärkeäksi oppimisen ja kokemuksen kannalta sekä opettajien että opiskelijoiden mielestä. Ensimmäisena kahtena vuonna kurssilla oli vain yksi projekti mikä tarkoitti yli 16 kehittäjää tiimissä. Tämä vaikeutti erityisesti ryhmäytymistä ja pariohjelmoinnille tärkeää parien vaihtelua. Tästä syystä Stapel ja muut (2008, 773) suosittelevatkin 8–12 hengen ryhmäkokoa.

Extreme programmingin käytännöt ovat vaikeita ja aloittelijat lipsuvat niistä helposti, jos heitä ei tueta niiden oppimisessa. Tästä syystä XP:n tunteva ohjaaja on tärkeässä roolissa opiskelijoiden motivaation kasvattamisessa ja käytäntöjen mukaan toimimisessa. Erityisesti testipohjaista kehittämistä on hankala valvoa ja siitä on helppo lipsua. Päivittäiset testien kattavuus -raportit olivat yksi hyväksi havaittu tapa, mutta nämä raportit eivät tarkista, milloin testit on kirjoitettu suhteessa koodiin, jota niillä testataan. Opiskelijat kirjoittivatkin usein testit vasta koodin jälkeen parantaakseen kattavuusarvoja.

Stapel ja muut (2008, 774–775) listaavat myös suosituksia, joita he oppivat esiteltyjen kolmen toteutuksen aikana. Ensinnäkin on tärkeää, että käyttäjäkertomukset ovat tarpeeksi pieniä, jotta ne voidaan toteuttaa iteraation aikana. Tämä koettiin tärkeämmäksi, kuin Extreme programmingin vaatimus, että käyttäjäkertomuksien pitää tulla asiakkaalta ja niiden pitää valmistuessaan tuoda selvää hyötyä asiakkaalle. Testauksen alueelta huomattiin käyttöliittymätestauksen vaikeus ja tähän kannattaa olla selkeä ratkaisu opettajilla jo etukäteen. Yksi mahdollinen ratkaisu voi olla automatisoidut käyttöliittymätesteihin tarkoitetut ohjelmistot.

Suosituksiin kuuluu myös, että opiskelijat tekevät uuden ohjelmiston eivätkä jatka vanhaa. Opiskelijoiden on vaikea saada kokonaiskuva isosta ohjelmistosta ja vanhojen ohjelmistojen testien kattavuus on harvoin tarvittavalla tasolla, jotta opiskelijat pääsisivät suoraan aloittamaan kehittämisen. Nämä laskevat opiskelijoiden motivaatiota huomattavasti. Stapel ja muut (2008, 775) myös

suosittelevat hyvää esittelyä XP:n käytäntöihin. Käytännön harjoitukseksi he suosittelevat Agile hour ja Extreme hour -harjoitteita esittelyluennoille.

Edellä mainituista kummatkin ovat tunnin mittaisia esimerkkikokemuksia, jotka käyvät läpi hyvin lyhyen version Extreme programmingin mukaan tehdystä projektista sisältäen prototyyppivaiheen ja 2 iteraatiota, jotka sisältävät 10 minuutin "suunnittelupäivät" ja 10 minuutin kehityspätkät. (Extreme Hour n.d.)

Analyysi

Ticorporaten tyylinen pitkäaikainen täysipäiväinen työskentely on harvinainen opiskelutapa korkeakoulutason opinnoissa ja Stapelin ja muiden (2008) pohjalta hyvin opettava ratkaisu. Toisaalta on myös nähtävissä vaatimuksia, kuten jatkuva tuki ja valvonta, jotka ohjaavat opiskelijoita toimimaan oikein, jotta he saavat todellisen kuvan prosessista. Shukla & Williams (2002) taas nostavat esille, että Extreme programming yksin ei anna kaikkia ohjelmistokehitykseen tarvittavia taitoja. Esimerkiksi mallintaminen on tärkeä taito, joka tulisi opettaa opiskelijoille. Oppimisen kannalta olisi tärkeää sisällyttää mallintaminen osaksi projektityötä, jotta opiskelijat saisivat käytännön harjoitusta ja huomaisivat mallintamisen hyödyn ohjelmistokehitykseen liittyvässä kommunikaatiossa. Kummatkin tutkimukset myös osoittavat, että testipohjainen kehitys ja pariohjelmointi vaativat tukea opettajilta ja molemmissa on helppo eksyä hyvien käytänteiden ulkopuolelle.

5.2 Osana ohjelmistokehityksen opetusta

Williams ja Upchurch (2001) lähestyvät Extreme programmingin käyttöä opetuksessa opetustapojen kautta ja tutkivat, miten XP ja sen käytännöt tukevat oppimista. He analysoivat neljä erilaista opetustapaa, joiden avulla voisi olla mahdollista kouluttaa taitavia suunnittelijoita, koska suunnittelutaito on tunnistettu yhdeksi tärkeimmistä ohjelmiston laadun tekijöistä. Ensimmäisenä kuvataan ajatus, jonka mukaan aloittelevasta ohjelmistokehittäjästä voitaisiin tunnistaa hänen tulevaisuuden taitonsa. Tämän ensivaikutelman perusteella voitaisiin tarkemmin tukea juuri näiden tulevaisuuden huippuosaajien kehitystä. Tätä ideaa on kuitenkin tutkittu, eikä tutkimuksissa ole huomattu, että alussa näkyvällä kyvykkyydellä olisi vaikutusta lopulliseen osaamistasoon. Tästä syystä Williams ja Upchurch (2001, 2) hylkääkin ajatuksen tarkastelun.

Toinen lähestymistapa on suunnitteluun liittyvän teoriaopetuksen määrän lisääminen. Opiskelijoille esiteltäisiin erilaisia suunnittelumalleja ja -tapoja, jotta he oppisivat sopivan ratkaisun suunnitteluongelmaansa. Tämä ratkaisu yksinään ei kuitenkaan anna opiskelijoille taitoa käyttää näitä suunnittelutyössään. (Mts. 3.)

Kolmas opetustapa on käytäntö. Opiskelijat laitetaan mahdollisimman realistiseen ympäristöön, jossa he oppisivat työelämän taitoja. Yleensä tämän tyylisissä projekteissa ei kuitenkaan selvästi määritetä opittavia taitoja, vaan oletetaan, että ympäristö riittää oppimiseen. Korkean tason asiantuntijuuteen ei kuitenkaan nousta pelkän kokemuksen avulla, vaan tarvitaan tarkoituksellista ja suunnattua harjoittelua. Asiantuntijuuteen liittyvä tutkimus on myös osoittanut, että asiantuntijat pystyvät soveltamaan osaamistaan hyvin laajasti eli heidän osaaminen on kontekstivapaata. Aloittelijat taas pystyvät soveltamaan osaamistaan vain hyvin samanlaisiin ongelmatilanteisiin, kuin missä osaaminen on hankittu. (Mts. 3.)

Neljäntenä esitellään prosessien tunteminen, jossa opettajien tulee tunnistaa ja ymmärtää suunnittelun kannalta tärkeät prosessit ja taidot. Tämän jälkeen he voivat tuottaa opetussuunnitelman, joka mahdollistaa näiden taitojen harjoittelun. Tämä mahdollistaa tarkoituksellisen harjoituksen ja valmennuksen, joka mahdollistaa korkeamman taitotason saavuttamisen. (Mts. 3.)

Näiden opetustapojen lisäksi Williams ja Upchurch (2001, 3–4) kuvaavat tiedon kolme muotoa: asiatieto, taito ja itseanalyysi. Ohjelmistokehityksen opetus keskittyy lähinnä asiatietoon (luennot ja kirjat) tai taitoihin (ohjelmointitehtävät). Opetuksessa ei yleensä vaadita tai ohjeisteta itseanalyysiä ja harva opiskelija tekee sitä itsenäisesti. Tutkimukset kuitenkin osoittavat, että se on tärkeä osa oppimista. (Mts. 3–4.)

Näiden analyysien pohjalta Williams ja Upchurch (2001, 4–5) analysoivat XP:n käytäntöjä ja miten ne tukevat oppimista. Jatkuva integraatio, testipohjainen kehitys ja hyväksyntätestit, pariohjelmointi, ohjelmointikäytänteet ja refaktorointi koetaan sellaisinaan hyvin hyödyllisiksi ja toimiviksi käytännöiksi myös opetuksen liitettynä. Yksinkertainen rakenne koetaan myös hyväksi, mutta lisäksi opiskelijoiden tulisi oppia yleisimmät suunnitteluprosessit, kuten mallintaminen ja käyttötapaukset. Suunnittelupeli sekä pienet julkaisut luovat tärkeän jatkumon harjoitteluun, joka opettaa opis-

kelijoita suunnittelemaan, arvioimaan ja muuttamaan toimintaansa saadun palautteen perusteella. Tällöin opiskelijat oppivat myös ajanhallintaa, joka tukee ylläpidettävää tahtia. Williams ja Upchurch (2001) nostavat myös esille, että arviointi perustuu usein opiskelijoiden tuottamiin ohjelmistoihin ja muihin tehtäväpalautuksiin. Opettajat eivät kuitenkaan voi osoittaa, millä tavalla nämä tuotokset on toteutettu ja miten hyvin opiskelija ymmärtää niiden yhteyden muihin opiskelualansa osa-alueisiin. Williamsin ja Upchurchin (2001, 5–6) mukaan arvioinnin tulisi keskittyä enemmän prosessin ja oppimisen arviointiin.

Schneiderin ja Johnstonin (2003) tutkimus lähestyy opetuskäyttöä oppimistavoitteiden suunnalta ja selvittää miten XP ja sen käytännöt voivat tukea oppimistavoitteiden saavuttamista. Oppimistavoitteet on otettu IEEE-CS:n (suuren kansainvälisen tekniikan järjestön tietotekniikan jaosto) ja ACM:n (tietotekniikan alan akateeminen yhdistys) esittämästä suosituksesta. Oppimistavoitteet ovat seuraavat: perusteet, ammattimainen toiminta, ohjelmiston vaatimukset, ohjelmiston rakentaminen, ohjelmiston suunnittelu, ohjelmiston testaus ja validointi, ohjelmiston kehitys, ohjelmistokehityksen prosessi, ohjelmiston laatu sekä ohjelmistokehityksen projektinhallinta. (Mts. 595.)

Schneider ja Johnston (2003) analysoivat XP:tä ja sen käytäntöjä oppimistavoitteiden ja lopputyö-tyylisen kurssin pohjalta. Lopputyöt kestävät yhdestä kahteen lukukautta ja niiden vaatima työaika on noin 10–15 tuntia viikossa. Ensimmäinen huomio perustuu nimen antamaan kuvaan. Opiskelijat tuntuvat hyväksyvän käytännöt paremmin, kun ne esitellään heille Extreme programming termin alta. Kuten muutkin ketterän kehityksen mallit XP on kevyt prosessin vaatimuksien näkökulmasta. XP ei vaadi tarkkaa vaatimusmäärittelyä, yksityiskohtaista ohjelmiston rakenteen mallintamista tai tiukkaa aikataulua. Sen sijaan se vaatii paljon kurinalaista käytäntöjen noudattamista, aktiivista itsensä ja ympäristön analysointia ja oikeaa yhteistyötä koko tiimiltä. Kurinalaisuus ja analyysikyky ei kuitenkaan ole luontaista korkeakouluopiskelijoille, joten niitä pitää tukea ulkopuolisilla vaatimuksilla opetushenkilökunnan puolelta. (Mts. 596–597.)

Extreme programmingin mukaan ohjelmistokehitysprojektin tärkein tuotos ja kehityksen mittari ovat toimiva ohjelmisto. Opetusympäristössä tämä ei kuitenkaan ole

totta. Paras oppiminen tapahtuu tekemällä virheitä ja oppimalla niistä. Näiden oppimiskokemusten arvoa tulisi painottaa enemmän osana arviointia kuin toimivaa koodia. Ohjelmistokehitysprojektin ei ole tarkoitus vain toimittaa asiakkaalle hyödyllinen ja toimiva ohjelmisto, vaan projektin kehittäjien tulee myös valmistautua tulevaisuuteen, jossa ohjelmisto kehitetään eteenpäin tai ongelmia korjataan. Tällöin korkeaan arvoon nousee dokumentaatio, noudatetut koodauskäytänteet ja hyvät testit. Myös näiden osa-alueiden pitäisi olla mukana arvioinnissa. (Mts. 598.)

Osa ongelmaa on myös nykyisen koulujärjestelmän tukema ajatusmaailma, jonka mukaan kurssin läpipääsy ja hyvä arvosana ovat tärkeämpiä kuin oppiminen ja muiden opiskelijoiden auttaminen. Yhteistyön ja ryhmätyön merkityksen korostaminen ja tukeminen ovat tärkeitä asioita, joita ohjelmistokehityksen koulutuksessa tulisi painottaa. Lopputuloksena Schneider ja Johnston (2003, 599) ilmaisevat, että pelkästään XP:n opettelu ei ole kannattavaa, vaan opiskelijoiden tulisi saada kokemusta monesta prosessimallista ja käytännöstä sekä saada ohjeistusta, miten he voisivat valita ja muokata tilanteeseen sopivan prosessimallin.

Analyysi

Sekä Williams ja Upchurch (2001) että Schneider ja Johnston (2003) nostavat esille tarpeen, että opiskelijoiden tulee tuntea erilaiset ohjelmistokehitysprosessit ja niiden yleisimmät käytössä olevat käytännöt kuten esimerkiksi mallintaminen ja testaus. Tämä ei ole mahdollista käyttämällä opiskeluprojektissa tarkalleen tiettyä prosessimallia. Opiskelijoilla tulisi olla erityisesti opetuskäyttöön kehitetty prosessi, joka sisältää tärkeimmät käytännöt ja niiden oppimista ja toteuttamista tuetaan ja arvioidaan. Lisäksi opiskelijoille tulisi antaa tietoa ja mahdollisuuksien mukaan käytännön harjoitusta erilaisista käytännöstä ja prosessimalleista.

5.3 Pariohjelmointi opetuksessa

Vuosina 2000–2007 Pohjois-Carolinan osavaltionyliopistossa käytettiin ja kehitettiin pariohjelmointia usealla eri kurssilla. Tämän kehitystyön pohjalta vuonna 2007 pariohjelmointi otettiin käyttöön myös Virginian teknillisessä yliopistossa. Tutkimuksessa arvioitiin kehitetyn ohjeistuksen toimivuutta yleisenä ohjeistuksena ja sitä kehitettiin eteenpäin. (Williams, McCrickard, Layman & Hussein 2008, 445.)

Williamsin ja muiden (2008) kuvaama ohjeistus sisältää yhdeksän suositusta ja tutkimuksen lopputuloksena yhtä suositusta muokattiin ja kaksi suositusta lisättiin. Ensimmäinen suositus kertoo, että pariohjelmoinnin käytäntö pitää ohjeistaa selkeästi opiskelijoille ja heidän tulisi saada tarpeeksi harjoitusta valvotussa tilassa. Opiskelijat eivät todennäköisesti ole kuulleet käytännöstä ja heidän siihenastinen kokemus parityöskentelystä on ollut työn jakamista kahteen osaan. Toinen suositus tarkentaa valvonnan tarkoitusta. Opettajan tulee valvoa työskentelyä, että pariohjelmointi toteutuu oikealla tavalla. Hänen tulee pitää huolta, että parit vaihtavat rooleja tarpeeksi usein ja että he osallistuvat työskentelyyn koko ajan. Myös erilaiset yhteistyöongelmat tulisi huomata ja korjata mahdollisimman nopeasti. (Mts. 447.)

Kolmas ja neljäs suositus esittävät käytäntöjä, joilla varmistetaan parien tasapuolinen työskentely. Poissaolojen ja myöhästymisien seurannat ja niistä seuraavat rangaistukset ovat tärkeitä, jotta työpari ei kärsi huomattavasti toisen poissaolosta tai myöhästymisestä ja jotta opiskelijat eivät myöhästy tahallaan välttääkseen pariohjelmoinnin. Opiskelijoiden tulee myös antaa palautetta jokaisesta paristaan, erityisesti jos pariohjelmointia tapahtuu kontaktien ulkopuolella. Pohjois-Carolinan osavaltionyliopistossa kehitetty PairEval parinarviointiohjelmisto esittää aluksi kysymyksiä, jotka kohdistuvat pariohjelmointiin osallistumisen eri osa-alueisiin ja lopuksi pyytää yleisarvosanan. Tämä yleisarvosana oli yksittäinen sana, mutta sen perässä oli kuvaileva teksti. Tämä kuvailevampi tapa tuotti laajemman kirjon vastauksia kuin pelkkä numeraalinen arviointi. Näiden palautteiden pohjalta opettajien tulee havaita ja puuttua mahdollisiin ongelmatapauksiin. Pohjois-Carolinan osavaltionyliopiston käytäntö oli, että jos opiskelija teki selvästi vähemmän töitä kuin hänen parinsa, niin hänen arvosanansa laski samassa suhteessa ja parin arvosana taas nousi. Viides suositus mainitsee, että opiskelijoiden arvosanan tulisi perustua sekä yksilö-, että ryhmätöihin. Tämän suosituksen tarkoituksena oli edelleen varmistaa, että yksikään opiskelija ei välty oppimiselta eikä läpäise kurssia vain pariensa avulla. (Mts. 448–449.)

Kuudes ja seitsemäs suositus puhuvat parien valinnoista ja vaihtelusta. Parien muodostaminen on tärkeä prosessi ja parien toimivuus on tärkeää. Yleisimmin yhteistyöongelmat parien välillä johtuvat taitotason tai motivaation eroista. Näiden ongelmien huomaaminen ja niiden ratkaisemin on opettajan vastuulla.

Aikaisemmissa suosituksissa mainitut valvonta ja työparien arviot ovat tärkeässä roolissa tässä huomioinnissa. Pareja tulee vaihdella joka tapauksessa useamman kerran kurssin aikana. Tämä auttaa opiskelijoita tutustumaan useampaan ihmiseen ja työskentelemään heidän kanssaan. Myös huonojen parien huomaamatta jäämisen vaikutus pienenee, koska pari rikkoutuu pian kuitenkin. Tutkitusti on huomattu, että opiskelijat eivät helposti ilmoita ongelmistaan parinsa kanssa ja tämän takia suositus kahdeksaan sanoo, että opiskelijoille tulee painottaa yhteistyöongelmien aikaisen ilmoittamisen tärkeyttä. (Mts. 449–450.)

Pariohjelmoinnissa roolien vaihdon pitää olla luontevaa ja helppoa. Yhdeksäs suositus muistuttaa tästä vaatimalla, että parin pitäisi pystyä istua mukavasti vierekkäin ja kummallakin pitäisi olla hyvä näköyhteys näyttöön ja hiiren ja näppäimistön siirto toiselle pitäisi olla helppoa. Ensimmäinen lisätty suositus oli Virginian teknillisen yliopiston kurssin pohjalta huomattu ja mainitsee, että opiskelijoilla tulisi olla yhteinen määränpää. Tällä kurssilla parien molempien osapuolten piti palauttaa henkilökohtainen ohjelmiston osa, joka toteutettiin pariohjelmointina. Kummallakin oli siis tavoitteena saada oma osa valmiiksi ja toisen osapuolen osan tekeminen koettiin osittain ylimääräiseksi työksi. Toinen lisätty suositus mainitsee, että pareja kannattaa usein vain ohjata oikeaan suuntaan eikä antaa oikeaa vastausta. Tämä mahdollistaa sen, että pari löytää oikean vastauksen itse, joka puolestaan kasvattaa heidän uskoaan omiin kykyihinsä. (Williams ym. 2008, 450–451.)

Analyysi

Nämä suositukset perusteluineen vaikuttavat järkeviltä ja ne pohjautuvat useamman vuoden kokemukseen. Yksin ja parityönä tehtyjen tehtävien vaatimus vaikuttaa perustuvan siihen, että he vain katsovat läpi opiskelijoiden palauttamat tehtävät. Tämä eroaa Tikon arviointikeskusteluista, joissa opiskelijoiden pitää selittää tehtävänsä ja näin opettaja pystyy arvioimaan, onko opiskelija oikeasti oppinut asian.

6 Kehitysehdotus

Kehitysehdotus kuvaa aikaa opintojen alusta Ticorporaten loppuun. Tämän ajanjakson jälkeen opiskelijat alkavat erikoistua omiin suuntiinsa sekä keskittyä yhä enemmän oman alansa teknologioihin. Tavoitteena on, että opiskelijat olisivat huomanneet opetetut käytännöt hyödyllisiksi ja käyttäisivät niitä myös tulevaisuudessa.

6.1 Ennen Ticorporatea

Ensimmäisen vuoden syksyllä tapahtuva ohjelmoinninalkeiden opetuksessa kontakteilla tuotettavat tehtävät tehdään pariohjelmointina ja kontaktien ulkopuolellakin sitä suositellaan, mutta ei valvota. Pareja kannattaa vaihtaa usein, mutta tärkeämpää on, että he ovat samantasoisia sekä osaamiseltaan että motivaatioltaan. Tästä johtuen opettaja määrää parit. Ensimmäisellä pariohjelmointikontaktilla oppilaat arvioivat oman osaamisena ja järjestäytyvät sen mukaan järjestykseen. Parit muodostetaan tästä järjestyksestä yksinkertaisesti yhdistämällä vierekkäiset opiskelijat. Kontaktin jälkeen opiskelijat antavat palautetta parinsa osaamisesta ja motivaatiosta. Opettaja päivittää näiden palautteiden mukaan opiskelijoiden tasoja ja määrää uudet parit seuraavalle kerralle.

Opettajan pitää muistuttaa ja tukea pariohjelmointia ja sen käytänteitä joka kontaktilla. Hänen tulee myös seurata parien toimintaa aktiivisesti ja tarttua nopeasti yhteistyöongelmiin. Myös ohjelmankäyttöongelmissa opettajan kannattaa auttaa pian, mutta ohjelmoinnillisissa ongelmissa parien tulisi yrittää jonkin aikaa itse tai yhteistyössä lähiparien kanssa.

Kurssin tehtävät sisältävät kuvauksen tehtävästä ja testikoodit, jotka opiskelija voi ajaa ja näin tarkistaa onko tehtävä valmis. Testien tulee olla helposti käyttöönotettavissa ja ajettavissa, mieluiten opiskelijalle annetaan valmis projekti ja koodieditorissa on valinta (mahdollisesti lisäosan avulla), josta painamalla testit ajetaan ja tulos tulee selvästi esille. Opettaja on luonut testit tehdessään tehtävän testipohjaisella kehityksellä. Kaksi esimerkkitehtävää testikoodeineen löytyy liitteestä 1. Palautuksista vaaditaan, että ne kääntyvät ja ne ovat yhdenmukaisesti muotoiltu.

Keväällä pidettävällä tekniikan kurssilla pidetään yksi kontakti testipohjaisesta kehityksestä ja jostakin yksikkötestauskehyksestä. Suositeltavaa on, että testauskehys on sama, jolla syksyn kurssin tehtävien testit oli tehty. Opiskelijoiden tulee huomata, että testipohjaisen kehityksen avulla tehdyt testit ovat samanlaisia kuin alkeiskurssilla annetut. Kontakteilla osa ajasta on tehtävien tekoa pariohjelmointina opettajan ollessa saatavilla. Tehtäväpalautuksien yhteydessä palautetaan myös luodut testit.

Toisen vuoden syksyllä pidetään yksi kontakti refaktoroinnista, jossa käydään läpi yksi isompi esimerkkikoodi. Koodin tulisi sisältää vain tuttuja teknologioita eli käytännössä alkeiskurssin loppupään tasoista koodia. Esimerkin avulla käydään läpi yleisiä ongelmia ja niiden perusrefaktorointeja.

6.2 Ticorporate

Ticorporate jatkaa edellisen vuosien tyyliin Scrumia seuraten ja ryhmät saavat muokata prosessiaan ja käytäntöjä. Muokkaus kuitenkin vaatii ensin ryhmän kesken keskustelua ja perustelujen hyväksyttämistä prosessista vastaavalla opettajalla.

Ohjelmistotuotannon alussa opetetaan suunnittelua ja sen mallintamista, jotta ryhmä voi tuottaa alustavan arkkitehtuurikaavion projektistaan. Kyseessä ei ole tarkka suunnitelma toteutuksesta, vaan alustava arvio siitä, miten tiimi ajattelee niiden toteutuvan. Kaavion tulee olla hyödyllisellä abstraktion tasolla, ei liian ylimalkainen, mutta ei liian yksityiskohtainen. Tätä kaavioita ja mahdollisia muita kaavioita tulee päivittää vähintään jokaisessa suunnittelupäivässä opiskelijoiden osaamisen ja ymmärryksen kasvaessa. Näin kaikilla tiimiläisillä säilyy kuva siitä, miten vastuu ja toiminnallisuus jakautuu ohjelmistossa.

Product owner on XP:n asiakas ja näin ollen hän tulee tekemään funktionaaliset testit käyttäjätarinoille testaajan kanssa. Funktionaalisista testeistä pidetään koulutus product ownereille ja testaajille. Testit eivät välttämättä ole koodia, mutta niiden pitää olla tarpeeksi yksityiskohtaisia kuvauksia, jotta kuka tahansa voi tarkistaa meneekö testi läpi. Tarinat jaetaan alle päivän pituisiksi tehtäviksi. Koodaus tapahtuu pariohjelmointina ja TDD:n sääntöjen mukaisesti ja Ticorporaten tiloissa päivystävän labramestarin tulee tukea näitä käytäntöjä. Aina kun tehtävä saadaan valmiiksi, tuotettu koodi ja testit lisätään projektiin versionhallinnan avulla. Pareilta vaaditaan kaikkien

testien läpimenoa ennen yhdistämistä, mutta versionhallintaohjelmiston tulee ajaa kaikki testit läpi joka tapauksessa vähintään päivittäin. Ryhmälle annetaan tai he saavat valita valmiin koodausstandardin jota heidän tulee noudattaa. He saavat muokata standardia hyvillä perusteluilla.

Opiskelijoilla tulee olla saatavilla tukea päivittäin paikallaolevalta ”labramestarilta”, joka voi olla Ticorporaten käynyt opiskelija tai opettaja. Hän on auttamassa helpommissa ongelmissa, jotka voivat liittyä tekemiseen, käytäntöihin tai prosessiin. Eri osaalueiden opettajien pitää olla myös saatavilla viikoittaisessa päivystyksessä, jotta opiskelijat voivat saada omaan tekemiseensä syvempää opastusta.

6.3 Ehdotuksen perustelut

Ehdotuksen tarkoituksena on edesauttaa osaaminen kehitystä XP:n tärkeimmistä opeista: pariohjelmoinnista, testipohjaisesta kehityksestä ja refaktoroinnista. Lisäksi Ticorporaten aikana opiskelijoiden on tarkoitus oppia myös muita XP:n käytäntöjä. Pariohjelmointi on laadukkuutta parantava käytäntö, jonka on todettu myös helpottavan oppimista ja tekevän siitä mukavampaa (Williams, McCrickard, Layman & Hussein 2008, 1–2). Kehitysehdotuksessa kuvataan miten pariohjelmointi kannattaisi toteuttaa kurssilla. Erityisesti pitää huomioida, että parien osaaminen ja motivaatio ovat suunnilleen samantasoisia ja että opettajalla on tarpeeksi kokemusta pariohjelmoinnista, jotta hän pystyy huomaamaan parien yhteistyöongelmat nopeasti. Nämä suositukset pohjautuvat Williams, McCrickard, Layman & Hussein (2008) artikkeliin.

Testipohjaisen kehityksen oppiminen alkaa myös alkeiskurssilla käyttämällä valmiita testejä osana tehtävänantoa. Tämä auttaa opiskelijoita ymmärtämään tehtävänannot paremmin. Lisäksi he huomaavat, että testit ovat hyvä tapa tietää milloin tehtävä on valmis ja täyttää vaatimukset. Ehdotus ei ota kantaa opetettavaan ohjelmointikieleen tai käytettävään työkaluun, koska valintaan vaikuttavat hyvin monet asiat, erityisesti kielen suosio Jyväskylässä ja Suomessa sekä Tikon opetuksen suuntautuminen tulevaisuudessa. Kaikille potentiaalisille valinnoille löytyy yksikkötestauksen mahdollistava työkalu, joten ohjelmointikieli ei vaikuta ehdotuksen toteuttamiskelpoisuuden suhteen.

Alkeiskurssin jälkeen opiskelijoilla pitäisi olla perusohjelmointitaidot, jolloin he voivat oppia tuottamaan testit myös itse. Tämän mahdollistamiseksi heille opetetaan testipohjaista kehittämistä keväällä pidettävällä tekniikan kurssilla. Refaktorointi kuuluu olennaisena osana TDD:hen, mutta sen tarkempi opettaminen pidetään kuitenkin vasta toisen vuoden syksyllä. Tämä toimii yhtäaikaa kesän jälkeisenä kertauksena TDD:stä ja samalla syventäen sitä refaktorointiosaamisella.

Ticorporate ei siirry Extreme programmingin termistöön, vaan tulee käyttämään edelleen Scrumin termistöä, koska se on edelleen käytetyin prosessimalli ohjelmistotuotannossa ja termistöjen erot eivät ole merkittäviä. Siihen lisätään käytäntöjä, jotka tukevat enemmän kokonaisvaltaista ohjelmistokehityksen oppimista. Tämän tarpeen Williams ja Upchurch (2001) sekä Schneiderin ja Johnstonin (2003) nostivat esille artikkeleissaan. Kun tiimi on suunnittelupäivänä valinnut tarinat, jotka he toteuttavat seuraavassa sprintissä, he luovat mallintamisen keinoin alustavan kuvan miten tarinat toteutetaan. Näin he saavat mallintamisesta käytännön harjoitusta, joka myös auttaa heidän projektinsa etenemistä. Tämän kuvan pohjalta product ownerin tulee (testaajan avustuksella) tuottaa funktionaaliset testit jokaiselle tarinalle. Nämä testit antavat paremman kuvan tarinan valmiudesta. Tämän tarkoituksena on vähentää tilanteita, joissa keskeneräisen tarinan sanotaan olevan valmis, koska ei olla huomattu mitä kaikkea tarina oikeasti pitää sisällään.

Suunnittelupäivässä tarinat jaetaan alle päivän pituisiksi tehtäviksi. Tehtävien kehitys pituus halutaan pitää alle päivässä, jotta saadaan paras hyöty jatkuvan integraation käytännöstä. Opiskelijat oppivat käyttämään versionhallintajärjestelmää sujuvammin ja korjaamaan mahdolliset ongelmatilanteet integraation aikana. Ohjelmointi-tehtävät toteutetaan pariohjelmoinnin ja testipohjaisen kehittämisen käytäntöjen mukaisesti. Edellämainittujen käytäntöjen tulisi olla tuttuja jo ennen Ticorporaten alkua, koska opiskelijoilla on jo paljon opittavaa projektityöskentelystä Ticorporatessa.

Ryhmät käyttävät valmista koodausstandardia, koska kokemattomina ohjelmoijina he eivät pysty luomaan yksiselitteistä kaiken kattavaa standardia. Valmis standardi antaa heille hyvän lähtökohdan, jonka pohjalta he voivat löytää tiimin oman tyylin. Tämä valmistaa heitä myös työelämään, jossa heidän tulee sopeutua yrityksen standardeihin ja toimintatapoihin.

7 Pohdinta

7.1 Työn tulos

Tämä opinnäytetyön tuloksena tuotettiin kehitysehdotus Jyväskylän ammattikorkeakoulun Tietojenkäsittelyn tutkinto-ohjelmalle. Ehdotuksen alkuperäisenä oletuksena oli Extreme programmingin vahva mukaantulo erityisesti Ticorporateen. Tiedon karttuessa ja analyysiä tehdessä kuitenkin työ siirtyi enemmän kohti ohjelmistokehityksen opetuksen kehittämistä. Mukaan otettiin XP:n käytäntöjä, mutta ehdotukseen sisältyy myös osia, joita Extreme programmingin prosessi ei suoraan vaadi.

Kehitysehdotuksen on tarkoitus parantaa opiskelijoiden ohjelmistokehityksen oppimista ja laajentaa heidän kuvaansa siihen liittyvistä prosessimalleista sekä käytännöistä. Kehitysehdotus on yritetty pitää toteuttamiskelpoisena eli se ei täysin riko nykyistä opetussuunnitelmaa vaan esittää muokkauksia, jotka sopivat edelleen nykyisiin toteutuksiin. Opinnäytetyön tekemiseen annetun ajan rajallisuuden ja siitä johtuvien rajauksien takia työssä ei tutkittu ehdotuksen vaikutuksia todellisuuteen, joka olisi vaatinut nykytilannetutkimuksen, ehdotuksen toteutuksen ja sen jälkeisen lopputilannetutkimuksen.

Kehitysehdotus luovutetaan Tikon opettajille ja he ottavat sen käyttöön oman ammattitaitonsa pohjalta parhaaksi kokemallaan tavalla. Ehdotus koostuu yksittäisistä osioista, jotka refaktorointia lukuun ottamatta toimivat myös yksittäisinä parannuksina opetukseen. Refaktoroinnin opettaminen on vaarallista, jos testaamista ei ole sisällytetty opetukseen, koska ilman testejä on todennäköistä, että refaktorointi aiheuttaa enemmän ongelmia kuin etua.

Ehdotus on yksilöity Tikon tarpeisiin ja siten sen yleistettävyys on heikko. Varsinkin Ticorporaten tyylinen täysipäiväinen pitkäaikainen projektityöskentely on harvinaista korkeakoulumaailmassa. Työ toimiikin todennäköisesti enemmän lähdelistana ja yhtenä idealähteenä toisien ympäristöjen toteutuksiin.

7.2 Luotettavuus ja pätevyys

Kehitysehdotus perustuu lähteiden sisältöihin ja kirjoittajan tekemään analyysiin. Pedagogisen teoriapohjan puute mahdollistaa, että lähteiden pedagogista luotettavuutta on vaikea arvioida, joka voi tuoda heikentää ehdotuksen pätevyyttä. Lähteet on valittu huolella ja tekijöiden taustat ja heidän muut tekemänsä tutkimukset käytiin läpi, jotta tekijöiden objektiivisuudesta syntyi parempi käsitys. Laurie Williams on tunnettu tutkija ketterän kehityksen saralla ja hän on tutkinut laajasti Extreme programmingia sekä pariohjelmointia erityisesti opetuskäytössä. Schneider ja Johnston (2003) osoittavat objektiivisuutta työssään nostamalla Extreme programmingin hyvät ja huonot puolet esille ja analysoimalla ne kriittisesti.

Reilun vuoden kokemus opettajan tehtävissä on antanut tekijälle laajempaa näkökulmaa opiskelijoiden käyttäytymiseen ja tarpeisiin, mutta kokemuksen puutteen vuoksi voi opiskelijoiden reaktio annetuista ratkaisuista olla erilainen kuin tekijä on arvellut. Jokainen lukija voi tehdä omat arvionsa ehdotuksesta esitettyjen perusteluiden ja oman asiantuntijuutensa pohjalta.

Ehdotuksen ratkaisut ovat värittyneet opinnäytetyön tekijän omien kokemuksien ja tiedostamattomien arvojen sekä asenteiden mukaan, joten on hyvin mahdollista, että toinen tekijä tuottaa erilaisen kehitysehdotuksen samasta materiaalista. Pariohjelmoinnin ja testipohjaisen kehityksen käytäntöjen mukaan ottaminen ja mallintamisen soveltamisen käytännössä, perustuvat vahvasti teoriaan ja tästä syystä ovat todennäköisesti tärkeä osa muissakin ehdotuksissa.

7.3 Tulevaisuuden suuntia

Sekä Williams ja Upchurch (2001) että Schneider ja Johnston (2003) nostivat esille, että mikään tietty prosessimalli ei sisällä kaikkea mitä korkeakoulutason ohjelmistokehityksen koulutuksen tulisi sisältää. He ovat siirtyneet käyttämään jonkinlaista sekoitusprosessia, joka sisältää sekä perinteisten että ketterien kehitysprosessien käytäntöjä. Tästä voidaan johtaa yksi selkeä kehittämiskohde tulevaisuuden tutkimukselle. Tulisi kehittää prosessimalli korkeakouluopetukseen, joka voisi mahdollisimman pitkälti sisältää kaikki työelämän tärkeät käytännöt ja antaa kuvan perinteisistä ja ketteristä prosesseista.

Extreme programmingiin liittyvä tutkimus on keskittynyt sen käytäntöihin sekä miten hyvin nämä käytännöt toimivat eri tilanteissa. Tämän käytäntöpohjaisen lähestymistavan vaihtoehtona voisi olla Beckin ja Andresin (2004) esittämä tapa. He esittävät, että XP:n arvot ja periaatteet voi ottaa lähtökohdaksi ja lisätä niihin mahdollisesti opetusmaailmalle tärkeitä osia. Näiden pohjalta voitaisiin luoda uusi lista käytänteistä ja näin luoda uusi opetuskäyttöön sopiva prosessimalli.

Tämän prosessimallin luomiseen tarvittaisiin selvitys, millä tavalla työelämässä nykyään työskennellään. Yleisesti puhutaan, että Scrum on selvästi yleisin prosessimalli, mutta on hyvin todennäköistä, että käytännössä yritysten välillä on suuria eroja. Siksi selvitys pitäisi tehdä käytännön tarkkailuna ja analyysinä kyselyjen sijaan. Prosessimallin luomisen aluksi voitaisiin ottaa nykyaikainen ohjelmistokehityksen opetussuunnitelma ja niiden pohjalta luoda arvo- ja periaatepohja. Yksi mahdollinen periaate voisi olla reinforced discipline (itsekurin vahvistus). Schneider ja Johnston (2003, 4) toteavat, että opiskelijoilla ei ole halua toimia käytäntöjen mukaan, varsinkaan niiden, joita prosessi ei heille pakota. Tästä syystä prosessimallin pitää tukea ja pakottaa opiskelijat seuraamaan käytäntöjä, jotka ovat heidän tulevaisuuden kannalta tärkeitä, mutta jotka on yleensä karsittu pois nykyisistä ketterän kehityksen prosessimalleista. Yksi esimerkki tämän periaatteen ohjaamista käytännöistä on kehitysehdotuksessakin nähty pakollinen arkkitehtuurikaavio. Tämä käytäntö on kevyt ja tärkeä opiskelijoiden tulevaisuuden ja oppimisen kannalta, mutta jos sitä ei pakoteta opiskelijoille, he huomaavat sen tarpeellisuuden vasta Ticorporaten jälkeen, jolloin oppimistilanteen mahdolliset hyödyt on menetetty.

Lähteet

AMK-Tutkinnot. N.d. Jyväskylän ammattikorkeakoulun www-sivut. Viitattu 19.4.2017. https://www.jamk.fi/fi/Koulutus/Tutkinnot/.

Beck, K. 2002. Test-driven development: By example. Boston: Addison-Wesley.

Beck, K. & Andres, C. 2004. Extreme programming explained: Embrace change. 2. p. Boston: Addison-Wesley.

Edelson, D. 2002. Design research: What we learn when we engage in design. Viitattu 10.3.2017. https://www.cs.uic.edu/~i523/edelson.pdf.

Extreme programming core practices. 2011. Ohjelmistokehityksen henkilöihin, projekteihin ja malleihin keskittyvä wiki. Päivitetty 2.4.2011. Viitattu 24.3.2011. http://wiki.c2.com/?ExtremeProgrammingCorePractices.

Extreme programming principles. N.d. Viitattu 2.6.2017. https://www.tutorialspoint.com/extreme_programming/extreme_programming_values_principles.htm.

Fowler, M. 1999. Refactoring: Improving the design of existing code. Boston: Addison-Wesley.

Fowler, M. 2003. PrinciplesOfXP. Viitattu 17.3.2017. https://martinfowler.com/bliki/PrinciplesOfXP.html.

Jeffries, R. 2011. What is Extreme Programming?. Yhden Extreme programmingin kehittäjän artikkelisivusto. Viitattu 27.3.2017. http://ronjeffries.com/xprog/what-is-extreme-programming/.

Kananen, J. 2012. Kehittämistutkimus opinnäytetyönä: Kehittämistutkimuksen kirjoittamisen käytännön opas. Jyväskylä: Jyväskylän ammattikorkeakoulu.

Opintojen rakenteet. N.d. Jyväskylän ammattikorkeakoulun tietojenkäsittelyn tutkinto-ohjelman opintojen rakennekuvat tutkinto-ohjelman virallissa blogissa. Viitattu 28.3.2017. http://blogit.jamk.fi/tiko/opinnot/opintojen-rakenteet/ HTK16S.

Principles of XP. N.d. Viitattu 2.6.2017. https://www.itemis.com/en/agile/scrum/compact/fundamentals-of-project-management/extreme-programming-xp.

Sanches, J., Williams, L. & Maximilien, E. 2007. On the Sustained Use of a Test-Driven Development Practice at IBM. Viitattu 27.3.2017. https://collaboration.csc.ncsu.edu/laurie/Papers/agile07.pdf.

Schneider, J. & Johnston, L. 2003. Extreme programming at universities: an educational perspective. Viitattu 13.4.2017. https://www.academia.edu/31609342/eXtreme_Programming_at_universities_an_educational_perspective.

Shukla, A. & Williams, L. 2002. Adapting Extreme programming for a core software engineering course. Viitattu 9.3.2017. https://collaboration.csc.ncsu.edu/laurie/Papers/XPCORE.PDF.

Stapel, K., Lübke, D. & Knauss, E. 2008. Best practices in Extreme programming course design. Proceedings of the 30th International Conference on Software Engineering. Viitattu 24.5.2017. https://ai2-s2-pdfs.s3.amazonaws.com/7f30/2c8f2998962e22746da0fa5276daadb18d74.pdf.

Wells, D. 1999. The rules of Extreme progmramming. Viitattu 28.3.2017. http://www.extremeprogramming.org/rules.html.

Wells, D. 2009. The values of Extreme programming. Viitattu 16.3.2017. http://www.extremeprogramming.org/values.html.

Williams, L., McCrickard, D.S., Layman, L. & Hussein, K. 2008. Eleven guidelines for implementing pair programming in the classroom. Viitattu 30.5.2017. https://pdfs.semanticscholar.org/78f6/ea83ac5b3dcf7de50beeee794be62a8cc6e0.pdf.

Williams, L. & Upchurch, R. 2001. Extreme programming for software engineering education? Viitattu 9.3.2017. https://collaboration.csc.ncsu.edu/laurie/Papers/FIE_01.pdf.

Liitteet

Liite 1. Esimerkkitehtävä ohjelmoinnin alkeiskurssille

Tehtävä:

Luo kaksi metodia kastunkoA ja kastunkoB.

KastunkoA ottaa vastaan totuusarvon sataako ulkona ja vastaa totuusarvolla kastutko.

KastunkoB ottaa vastaan totuusarvot sataako, onko sateenvarjoa ja oletko autolla ja vastaa totuusarvolla kastutko.

Testit:

```
//KastunkoA
boolean tulos =  kastunkoA(true);
Assert(tulos == true);

tulos = kastunkoA(false);
Assert(tulos == false);

//KastunkoB
boolean tulos = kastunkoB(true, false, false);
Assert (tulos == true);

tulos = kastunkoB(true, true, false);
Assert (tulos == false);
```

```
tulos = kastunkoB(true, false, true);

Assert (tulos == false);


tulos = kastunkoB(true, true, true);

Assert (tulos == false);


tulos = kastunkoB(false, true, true);

Assert (tulos == false);


tulos = kastunkoB(false, false, true);

Assert (tulos == false);


tulos = kastunkoB(false, true, false);

Assert (tulos == false);


tulos = kastunkoB(false, false, false);

Assert (tulos == false);
```

Tehtävä:

Tee Pankkitili-niminen luokka, jolla on tilinumero, tilinomistaja ja saldo-attribuutit. Pankkitili olion luomisen jälkeen tilinumero ja tilinomistajien pitää olla tyhjiä merkkijonoja ja saldon tulee olla nolla. Kaikilla attribuuteilla pitää olla get- ja set-metodit.

get-metodit palauttavat vastaavan attribuutin arvon (esim. getSaldo palauttaa attribuutin saldo arvon)

set-metodit asettavat vastaavan attribuutin arvon.

Testit:

```
TestPankkitiliCreation() {
		Pankkitili tili = new Pankkitili();
}
TestPankkitiliCreationNoParameters() {
		Pankkitili tili = new Pankkitili();
		Assert(tili.getTilinomistaja().equals(""));
		Assert(tili.getTilinumero().equals(""));
		Assert(tili.getSaldo() == 0.0);
}
TestPankkitiliSettingTilinumero() {
		Pankkitili tili = new Pankkitili();
		tili.setTilinumero("123123-456456");
		Assert(tili.getTilinumero().equals("123123-456456"));
}
TestPankkitiliSettingTilinomistaja() {
		Pankkitili tili = new Pankkitili();
		tili.setTilinomistaja("Matti Meikalainen");
		Assert(tili.getTilinomistaja.equals("Matti Meikalainen"));
}
TestPankkitiliSettingSaldo() {
		Pankkitili tili = new Pankkitili();
```

```
        tili.setSaldo(200);

        Assert(tili.getSaldo() == 200.0));

}
```