Sami Kosonen

Ohjelmoinnin opetus Extreme Programming -hengessä

Tietotekniikan
pro gradu -tutkielma
20. elokuuta 2005

Jyväskylän yliopisto

Tietotekniikan laitos

Jyväskylä

Tekijä: Sami Kosonen
Yhteystiedot: skosone@st.jyu.fi
Työn nimi: Ohjelmoinnin opetus Extreme Programming -hengessä
Title in English: Teaching of Computer Science in the Spirit of Extreme Programming
Työ: Tietotekniikan pro gradu -tutkielma
Sivumäärä: 76
Tiivistelmä: Extreme Programming -metodologian suosio on kasvanut teollisuudessa viime vuosina ja sen käyttöä ohjelmoinnin opetuksessa on myös tutkittu ja kokeiltu eri yliopistoissa. Tässä pro gradu -tutkielmassa tutkitaan tämän metodologian eri käytäntöjä ja tutkimuksissa saatuja kokemuksia niiden soveltamisesta ohjelmoinnin opetukseen.
English abstract: Extreme Programming -methodology has become more popular in industry during the last years and its use in the teaching of programming has also been researched and tried in different universities. What is studied in this master's thesis are the different practises of this methodology and the experiences gained in research, in which these practises have been applied in the teaching of programming.
Avainsanat: tietotekniikka, pro gradu tutkielma, Extreme Programming, XP, ohjelmointi, opetus
Keywords: computer science, Master's Thesis, Extreme Programming, XP, programming, teaching

Esipuhe

Ohjelmointi on kiinnostanut minua jo 11-vuotiaasta lähtien, jolloin sain ensimmäisen tietokoneeni, Amiga 500:n. Myöhemmin opinnoissani kiinnostuin ohjelmoinnin opettamisesta ja halusin tutustua syvemmin Extreme Programming -metodologiaan. Niinpä tämä Jonne Itkosen ehdottama pro gradu -aihe tuntui minulle sopivalta ja kiinnostavalta. Työ on opettanut minulle paljon XP:stä, sekä sen mahdollisuuksista ohjelmoinnin opetuksessa.

Ensimmäisen kokemuksen Extreme Programming -metodologiasta sain syksyllä 2003 Jyväskylän yliopiston Tietotekniikan laitoksen Johdatus ohjelmistotekniikkaan -kurssilla, jolloin Jonne Itkonen luennoi aiheesta. XP vaikutti hyvin erilaiselta, mutta poikkeuksellisen hyvältä tavalta kehittää ohjelmistoja. Metodologia jäi kiinnostamaan sen verran, että halusin tutustua siihen myöhemmin lisää ja mahdollisesti kokeilla ohjelmistokehitystä käyttäen tätä prosessimallia.

Keväällä 2001 suoritin lehtori Vesa Lappalaisen pitämän Ohjelmointi++ -kurssin ja syksyllä 2002 hänen pitämänsä Graafisten käyttöliittymien kurssin. Kursseilla pidin Lappalaisen opetustyylistä. Myöhemmin opettajaopinnoissani huomasin, että hän soveltaa opetuksessaan paljon ongelmalähtöisen oppimisen (Problem Based Learning) menetelmää. Keväällä 2004 sain toimia ohjaajana Lappalaisen pitämällä Ohjelmointi 2 -kurssilla, jolloin sain hieman omakohtaistakin kokemusta ohjelmoinnin opettamisesta.

Kiitän kaikkia, jotka ovat olleet vaikuttamassa työni valmistumiseen. Erityisen kiitollinen olen ohjaajilleni Jonne Itkoselle ja lehtori Vesa Lappalaiselle heidän antamistaan neuvoista ja palautteesta, vaimolleni Aradhnalle hänen osoittamastaan kärsivällisyydestä, vanhemmilleni siitä, että suostuivat ostamaan minulle sen Amiga 500:n, sekä Jumalalle siitä armosta, että olen saanut tämän työn valmiiksi ajallaan.

Sanasto

Ant: Java-pohjainen projektin kääntämistyökalu, joka toimii eri käyttöjärjestelmissä

CVS: Concurrent Versions System – Versionhallintajärjestelmä, tärkeä osa lähdekoodin konfiguraatioiden hallintaa [11]

Java: Sun Microsystems, inc.:n kehittämä laitteistoriippumaton oliopohjainen ohjelmointikieli

JCVS: Javalla toteutettu CVS-asiakasohjelmisto

JUnit: Sovelluskehys, jonka avulla voidaan laatia ja suorittaa automaattisia ohjelmoija- ja yksikkötestejä

inkrementaalinen: Kumuloituva

iteratiivinen: Jaksottainen, vaiheittainen

Smalltalk: Olio-ohjelmointikieli, joka on kehitetty alunperin 1970-luvulla Xeroxin Palo Alton tutkimuskeskuksessa

timboxing: Tekniikka, jonka avulla määritellään seuraavan työjakson aikana suoritettavat työtehtävät

UML: The Unified Modeling Language – Visuaalinen oliopohjaisten tietojärjestelmien mallinnuskieli

Sisältö

Esipuhe **i**

Sanasto **ii**

1 Johdanto **1**

2 Perinteiset menetelmät **3**

2.1 Koodaa ja korjaa . . . 3

2.2 Vesiputousmalli . . . 3

2.2.1 Vesiputousmallin vaiheet . . . 4

2.2.2 Vesiputousmallin arviointia . . . 5

2.3 Inkrementaalinen malli . . . 6

3 Ketterät menetelmät **8**

3.1 Perinteisten ja ketterien menetelmien vertailua . . . 9

4 Extreme Programming -metodologian esittely **11**

4.1 XP:n neljä perusarvoa . . . 12

4.2 XP:n käytännöt . . . 13

4.2.1 Suunnittelupeli . . . 16

4.2.2 Pienet julkaisut . . . 17

4.2.3 Järjestelmän metafora . . . 17

4.2.4 Yksinkertainen rakenne ja suunnittelu . . . 17

4.2.5 Testaus . . . 18

4.2.6 Uudelleenrakentaminen . . . 19

4.2.7 Pariohjelmointi . . . 19

4.2.8 Yhteisomistajuus . . . 20

4.2.9 Jatkuva integrointi . . . 20

4.2.10 40-tuntinen työviikko . . . 21

4.2.11 Paikan päällä oleva asiakas . . . 21

4.2.12 Koodausstandardit . . . 22

4.3 XP:n säännöt . . . 22

4.3.1 Osallistumisen säännöt . . . 22

4.3.2 Pelaamisen säännöt 23

5 Kokemuksia XP:n soveltamisesta ohjelmoinnin opetukseen 25

5.1 Suunnittelupeli 26
5.2 Pienet julkaisut 27
5.3 Järjestelmän metafora 28
5.4 Yksinkertainen rakenne ja suunnittelu 29
5.5 Testaus 30
5.6 Uudelleenrakentaminen 32
5.7 Pariohjelmointi 33
5.8 Yhteisomistajuus 34
5.9 Jatkuva integrointi 35
5.10 40-tuntinen työviikko 36
5.11 Paikan päällä oleva asiakas 37
5.12 Koodausstandardit 38
5.13 Mestari – oppipoika -malli ohjelmistostudiossa 39
5.13.1 Mestari – oppipoika -malli 39
5.13.2 Ohjelmistostudio 40
5.13.3 Ongelmalähtöinen oppiminen 42
5.14 Opiskelijan näkökulma 43
5.14.1 XP:n pilottitutkimus 43

6 Suosituksia XP:n soveltamiseen opetuksessa 45

6.1 Suunnittelupeli 45
6.1.1 Asiakas 45
6.1.2 Työajan arviointi 45
6.1.3 Tulevaisuuteen varautuminen 46
6.1.4 Käyttäjätarinoiden hajoittaminen tehtävätasolle 46
6.2 Pienet julkaisut 47
6.3 Järjestelmän metafora 47
6.4 Yksinkertainen rakenne ja suunnittelu 48
6.4.1 Ymmärrettävää ja laadukasta ohjelmakoodia 48
6.4.2 Kevyet etukäteissuunnitelmat 48
6.5 Testaus 49
6.5.1 Motivointi 49
6.5.2 Tarve opetussuunnitelman muutokseen 50
6.6 Uudelleenrakentaminen 51
6.7 Pariohjelmointi 51

6.7.1 Parien ja ryhmien valinta 52
6.7.2 Yhteisen ajan löytäminen 52
6.7.3 Kulttuurin ja toimintatapojen muutos 53
6.7.4 Kurssin arvostelu 53
6.8 Yhteisomistajuus 54
6.8.1 Arvosteluperusteiden vaikutus 54
6.8.2 Versionhallintajärjestelmä 55
6.9 Jatkuva integrointi 55
6.10 40-tuntinen työviikko 55
6.11 Paikan päällä oleva asiakas 56
6.12 Koodausstandardit 57
6.13 Yleisiä suosituksia XP:n soveltamiseen 58
6.13.1 Hybridimetodologia 58
6.13.2 Opiskelijoiden työskentelyn tarkkailu ja arviointi 59
6.13.3 Pääperiaatteiden määrittäminen 60
6.13.4 Oma tila kehitystiimille 61
6.13.5 Laboratorioajat 61
6.13.6 Puhtaalta pöydältä aloittaminen, vai 62
6.13.7 Tunnettujen tekniikoiden käyttäminen 63
6.13.8 Ohjelmistotekniikan käytäntöjen noudattaminen 63
6.13.9 Jatkuva prosessivalmennus 64

7 Yhteenveto **65**

8 Viitteet **67**

1 Johdanto

Ohjelmistonkehitys on vielä 2000-luvullakin jonkinlaisessa kriisissä, sillä melkein kolme neljäsosaa vuoden 2000 ohjelmistoprojekteista epäonnistui [9]. Mikäli muilla insinöörialoilla projektien onnistumisprosentti olisi näin alhainen, oltaisiin todella huolestuneita ja vakavasti mietittäisiin mitä pitäisi tehdä toisin. Collins ja Miller [9] listaavat neljä pääsyytä, jotka vaikuttavat tähän tilanteeseen:

> Ihmiset eivät ole vakuuttuneita, että heillä on ongelma.
>
> Ihmiset tietävät, että heillä on ongelma, mutta pelkäävät ottaa riskiä ja tehdä asioita toisin ratkaistakseen ongelman.
>
> Ihmiset tietävät, että heillä on ongelma, ovat halukkaita yrittämään ratkaista sitä, mutta ymmärtävät väärin ongelman, jota he yrittävät ratkaista.
>
> Ihmiset tietävät, että heillä on ongelma, ovat halukkaita yrittämään ratkaista sitä, ymmärtävät ongelman, mutta ovat sidottuja vallitsevaan tilanteeseen.

Monissa ohjelmistoprojekteissa perinteiset ohjelmistonkehitysprosessimallit, kuten esimerkiksi vesiputousmalli (Waterfall), on havaittu liian jäykiksi ja kykenemättömiksi vastaamaan asiakkaan muuttuviin vaatimuksiin. Tällaisen mallin mukaan etenevän projektin alussa asiakkaan kanssa on tehty valmiit suunnitelmat siitä, millainen kehitettävän sovelluksen tulisi olla, jonka jälkeen sitä on lähdetty suunnittelemaan ja toteuttamaan. Toteutusvaihe voi kestää muutamasta kuukaudesta jopa muutamaan vuoteen ja kun asiakkaalle esitellään valmista tuotosta, se ei välttämättä vastaakaan asiakkaan toiveita. Voi olla että joku näkökulma on jäänyt projektin alussa kokonaan huomioimatta tai ymmärretty väärin, jolloin alusta alkaen on lähdetty kehittämään vääränlaista järjestelmää. Mitä myöhäisemmässä vaiheessa tällaisia muutostarpeita tulee, sitä enemmän ne maksaa. Ratkaisuksi tällaisiin ongelmiin on lähdetty kehittämään joustavampia prosessimalleja.

Ketterät (Agile) ohjelmistokehitysprosessimallit [5], eli ketterät menetelmät, pyrkivät tuomaan kaivattua joustavuutta ohjelmistokehitykseen. Tavoitteena on tuottaa asiakkaalle lyhyin väliajoin toimivia versioita kehitettävästä ohjelmistosta, pyrkiä hallitsemaan muuttuvia vaatimuksia myös myöhäisessä projektin vaiheessa, sekä olla yhteistyössä asiakkaan kanssa. Toimiva ja laadukas ohjelmisto nähdään tärkeämpänä kuin kattava dokumentaatio.

Ketterät menetelmät, ja erityisesti tässä tutkimuksessa käsitelty Extreme Programming (XP), ovat kasvattaneet teollisuudessa suosiotaan viime vuosina. Kasvava määrä ohjemistojen kehittäjätiimejä käyttävät työelämässä menestyksellisesti näitä menetelmiä, mutta miten ohjelmistotekniikan opetuksessa voidaan vastata tähän uuteen suuntaukseen? [24]

Opiskelijoille tulisi opettaa sekä ketterää että perinteistä ohjelmistotekniikkaa ja valmistaa heitä sopeutumaan erilaisiin menettelytapoihin, joita heidän mahdolliset tulevat työnantajansa ja tiiminsä käyttävät. Teollisuus tarvitsee ihmisiä, jotka ovat joustavia ja jatkuviin muutoksiin mukautuvia, joten myös koulutuksen tulisi ottaa uusi haaste vastaan ja alkaa kouluttamaan opiskelijoita tähän uuteen muuttuneeseen tilanteeseen. [24]

Työelämässä olevat opiskelijat toteavat monesti, että he ovat oppineet monia asioita opiskelujensa aikana, mutta käytännön taidot joita he tarvitsevat päivittäin, he ovat oppineet pääosin valmistumisensa jälkeen työmaailmassa. Työnantajat, jotka palkkaavat vastavalmistuneita opiskelijoita ohjelmistokehitystiimiin, ovat taas jo valmiiksi asennoituneet joutuvansa antamaan näille uusille työntekijöilleen paljon koulutusta. Heidän mukaansa vastavalmistuneilta puuttuu muun muassa kykyä ongelmanratkaisuun, suunnitteluun, sekä ihmisten väliseen kommunikointiin. [33]

Kun opetushenkilökunnalta kysytään näistä ongelmista, he joutuvat toteamaan, että nopeasti kehittyvä ala, suuret opiskelijamäärät ja ajanpuute, tekevät koulutuksen hyvin vaikeaksi. Jollakin tavalla kaikki nämä eri tahot tiedostavat ohjelmistotekniikan opetuksessa olevat todelliset ongelmat, mutta harvat osaavat esittää niihin ratkaisua. [33]

Talbotin ja Auerin [33] mukaan nykyinen luokkahuonekoulutuksen malli on jo peruslähtökohdiltaan viallinen. Yliopistoista ja korkeakouluista valmistuvilla opiskelijoilla on kyllä paljon tietoa erilaisista tärkeistä asioista, mutta vähän viisautta ja ymmärrystä soveltaa näitä tietojaan käytäntöön.

Tässä pro gradu -työssä tutkitaan, minkälaisia kokemuksia ja tutkimustuloksia XP:n käytöstä ohjelmoinnin opetuksessa on saatu maailman eri yliopistoissa, sekä miten XP:tä kannattaisi opetuksessa soveltaa.

2 Perinteiset menetelmät

Tässä luvussa esitellään joitakin perinteisiä ohjelmistokehityksessä käytettyjä prosessimalleja, kuten koodaa ja korjaa (Code 'n' fix), vesiputousmalli (Waterfall) sekä inkrementaalinen malli (Incremental Model). Näistä prosessimalleista käydään läpi niiden perusperiaatteita sekä hyviä ja huonoja puolia.

2.1 Koodaa ja korjaa

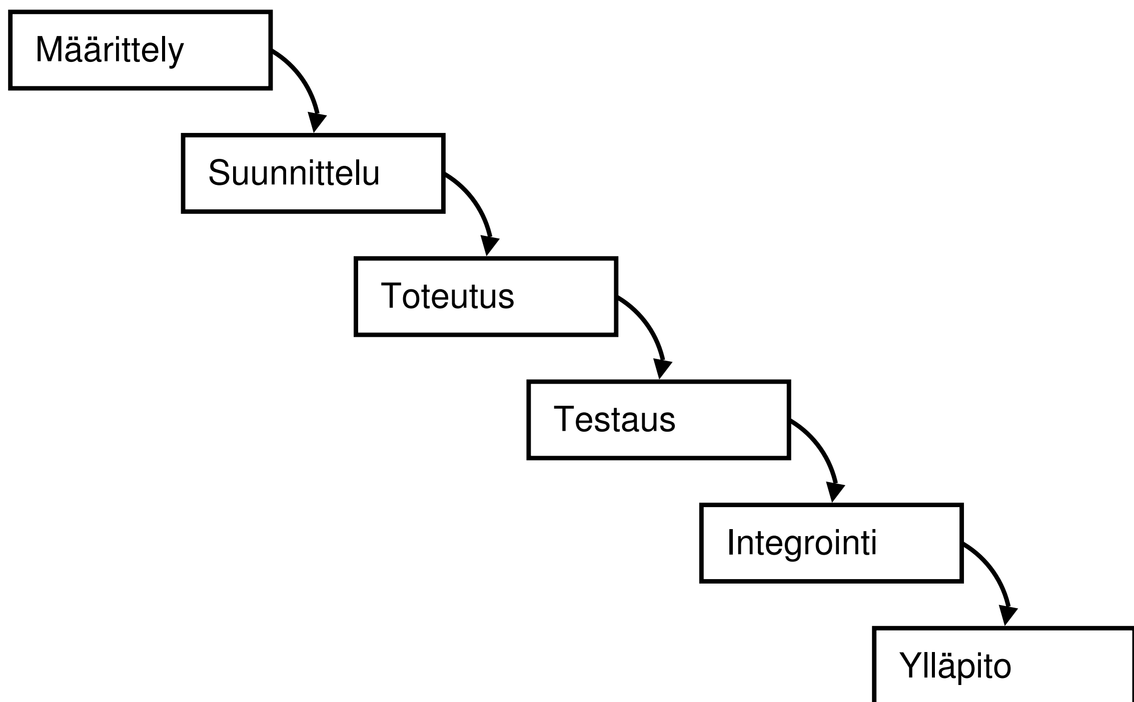
Useimmat, jotka ovat joskus jotakin ohjelmoineet, ovat varmasti tulleet käyttäneeksi koodaa ja korjaa -menetelmää. Menetelmä on yksinkertaisuudessaan se, että aluksi on olemassa jonkinlaiset vaatimukset kehitettävälle sovellukselle ja sitten koodataan ja korjataan kunnes sovellus on valmis. Koodaaminen aloitetaan suoraan ilman mitään suunnitelmia.

Prosessimalli on kevyt, eikä sen hallintaan mene aikaa, mutta samalla se on myös epävarma ja hallitsematon. On hankalaa määritellä missä vaiheessa projekti on menossa, jolloin myös tuotteen valmistumisajankohtaa on vaikeaa arvioida. Koska mitään järjestelmällistä testausta ei välttämättä tule tehdyksi, lopputuotteeseen jää helposti virheitä. Myös jatkokehittäjät saavat eteensä suuren haasteen, mikäli dokumentointi jäänyt hoitamatta.

2.2 Vesiputousmalli

Perinteisesti ohjelmistotuotantoprosessi on perustunut vesiputousmallille 2.1. Winston Royce [27] esitteli hieman tämänkaltaisen mallin vuonna 1970 tavoitteenaan saada ohjelmistotuotannosta systemaattisempaa. Royce ei itse nimittänyt mallia vesiputousmalliksi [35], ja muutenkin hänen esittämänsä malli on todennäköisesti ymmärretty hieman väärin, minkä seurauksena nykyinen vesiputousmalli on saanut alkunsa. Vesiputousmalli on sittemmin tullut perustaksi useimmille ohjelmistokehitysmalleille. Vesiputousmallista on olemassa erilaisia muunnelmia, mutta perusajatus on lähes samanlainen kaikissa.

Vesiputousmalli koostuu eri vaiheista, jotka suoritetaan loppuun ja joiden tulokset toimivat seuraavan vaiheen syötteenä hyväksymisen jälkeen. Seuraavaan vai-



Kuva 2.1: Vesiputousmalli.

heeseen ei siirrytä, ennen kuin edellinen on saatu päätökseen, eikä aikaisempiin vaiheisiin ole enää tarkoitus palata. Jokaisen vaiheen tuotoksena saadaan kattava dokumentaatio, mikä helpottaa prosessin hallintaa. Malliin on myös sisäänrakennettu laadunvarmistus ja testaus. [25]

Jokaisen vaiheen suorituksen jälkeen tarkistetaan, ollaanko tekemässä oikeanlaista ohjelmistoa (validointi), sekä ollaanko tekemässä ohjelmistoa oikein (verifiointi). Mikäli näissä testeissä huomataan, että ollaan tehty vääränlaista ohjelmistoa, tai että ohjelmistoa ei olla tehty oikein, voidaan palata takaisin tähän, tai tarvittaessa edellisiin vaiheisiin ja korjata havaitut puutteellisuudet. [25]

2.2.1 Vesiputousmallin vaiheet

Perinteisesti vesiputousmallin vaiheita, eli perustehtäviä, ovat vaatimusten analysointi ja määrittely, järjestelmän ja ohjelmiston suunnittelu, toteutus ja yksikkötestaus, integrointi ja järjestelmätestaus, sekä käyttöönotto ja ylläpito. [32]

Vaatimusten analysointi -vaiheessa ollaan tiiviissä yhteistyössä asiakkaan kanssa ja koitetaan selvittää mikä on ratkaistava ongelma. Kun ongelma on yleisesti ottaen selvillä, mietitään onko se mahdollista ratkaista olemassa olevilla resursseilla. Asiakkaalle pyritään arvioimaan mahdollisia kustannuksia, projektin kestoa, sekä

muita käytännön asioita. Vaiheen yhtenä tärkeänä päämääränä on selvittää kannattaako projektia aloittaa.

Vaatimusten määrittely -vaiheessa mietitään minkälainen järjestelmä pitäisi laatia, että vaatimusten analysointivaiheessa määritelty ongelma ratkeaisi. Asiakasvaatimuksista johdetaan sovelluksen vaatimukset. Tässä vaiheessa ei vielä mietitä käytännön toteutusta eikä tekniikoita, vaan pohditaan järjestelmän erilaisia käyttötapauksia ja toiminnallisia vaatimuksia. Vaiheen tuotoksena valmistuu vaatimusmäärittely, jossa kuvataan järjestelmälle asetetut vaatimukset.

Suunnitteluvaiheessa ei olla enää niin paljon tekemisissä asiakkaan kanssa, vaan aloitetaan jo siirtymään enemmän toteutuksen puolelle. Tässä vaiheessa mietitään kuinka määrittelyvaiheessa määritellyt vaatimukset täyttävä järjestelmä saataisiin toteutettua. Vaiheen tuotoksena saadaan järjestelmän ja ohjelmiston tekninen määrittely. [32]

Toteutusvaiheessa päästään itse ohjelmointiin ja tekniseen toteutukseen. Pohjana ovat suunnitteluvaiheessa tehdyt tekniset määrittelyt ja tarkoituksena on toteuttaa järjestelmän eri kokonaisuudet. Järjestelmän eri osaset testataan erikseen toteutuksen jälkeen. [32]

Integrointi- ja testausvaiheessa eri osaset yhdistetään toimivaksi kokonaisuudeksi ja niiden toiminta testataan järjestelmällisesti [32]. Integrointitestauksessa testataan rajapintojen toimivuus, koko järjestelmän verifioinnissa selvitetään että järjestelmä toimii, ja validoinnilla selvitetään että järjestelmä tekee varmasti mitä sen halutaan tekevän.

Käyttöönotto- ja ylläpitovaiheessa järjestelmä otetaan käyttöön ja sen ylläpito alkaa. Tässä vaiheessa järjestelmän tulisi olla toimiva ja tyydyttää asiakasta. Ylläpito sisältää mahdollisten virheiden korjausta ja järjestelmään tehtäviä laajennuksia. [32]

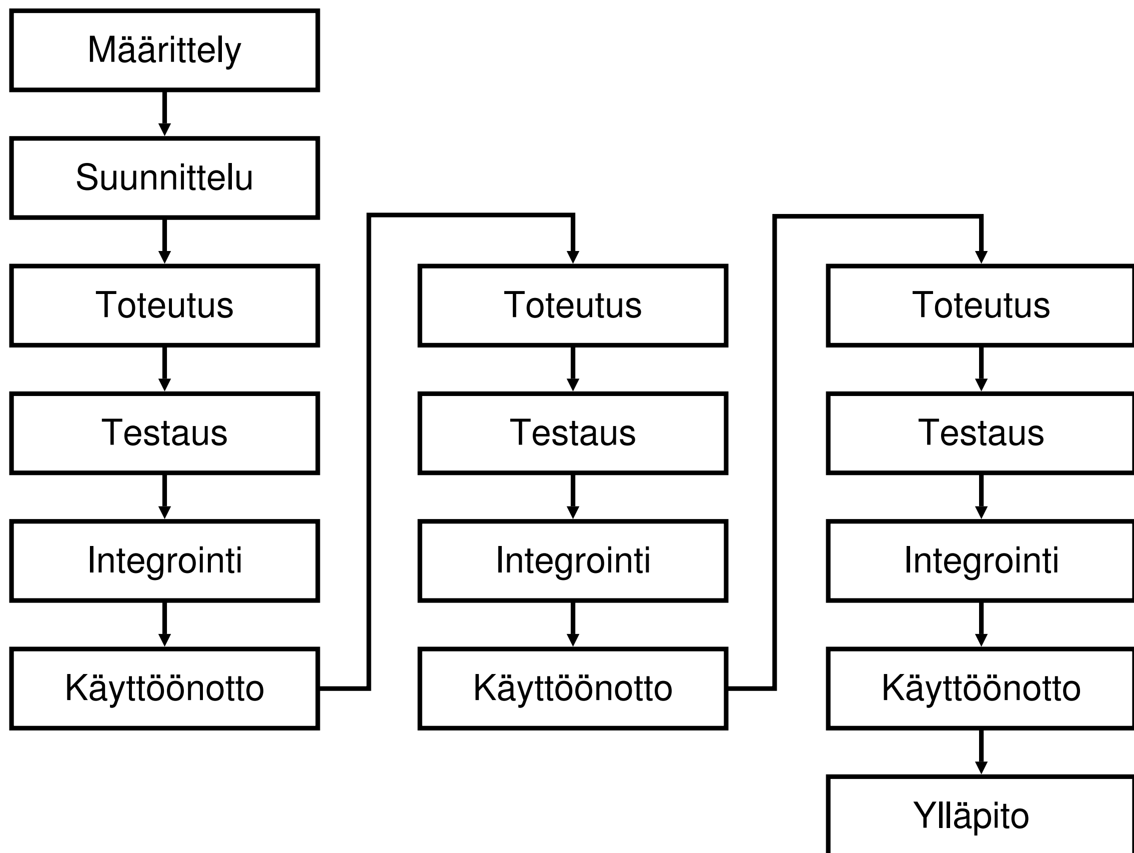
2.2.2 Vesiputousmallin arviointia

Vesiputousmallin rakenne on selkeä, se on helposti kontrolloitavissa, ja jokaisen vaiheen tuotoksena on selkeät dokumentaatiot. Toisaalta prosessi on myös kallis, hidas ja joustamaton. Prosessi voi sopia projekteille, joissa vaatimukset eivät tule muuttumaan, ja joissa työ voidaan saattaa loppuun tällaisen suoraviivaisen prosessin mukaisesti. [32, 25]

Mallissa vaiheet seuraavat toisiaan alkaen esitutkimuksesta ja päättyen ylläpitoon, mutta käytännössä tämä toteutuu vain ideaalitapauksissa. Todellisissa kehityshankkeissa vaiheet eivät etene niin suoraviivaisesti eteenpäin. Peräkkäiset vaiheet ovat yleensä toisistaan riippuvia ja tietyn vaiheen suoritus voi paljastaa edellis-

ten vaiheiden virheitä, jolloin on mahdollisesti palattava näihin vaiheisiin ja tehtävä ne uudestaan korjaten virheet. Huonona puolena on myös se, että lopullinen tuote on nähtävissä vain prosessin lopussa ja muutosten toteuttaminen on hankalaa. [25]

2.3 Inkrementaalinen malli



Kuva 2.2: Inkrementaalisen kehityksen malli.

Inkrementaalisen kehityksen malli on lähes samanlainen kuin vesiputousmalli. Aluksi määritellään vaatimukset kehitettävälle sovellukselle ja suunnitellaan sen toteutus. Tämän jälkeen aloitetaan toistamaan viimeisiä vaiheita, ja jokaisen toiston tuloksena on uusi inkrementti, eli lisäys kehitettävään ohjelmistoon.

Parannusta vesiputousmalliin inkrementaalisen kehityksen mallissa on jokaisen inkrementin tuloksena saadut lisäykset kehitettävään ohjelmistoon. Koska ohjelmakoodin eri osat testataan ja integroidaan jokaisessa inkrementissä, ohjelmakoodin integroinnissa ei tule yhtä paljon ongelmia, kuin vesiputousmallissa, jossa integrointi tehdään vasta projektin loppuvaiheessa ennen käyttöönottoa ja ylläpitoa.

Myös asiakkaalle voidaan antaa käytettäväksi toimivia versioita ohjelmasta aikaisemmassa vaiheessa kuin vesiputousmallissa.

3 Ketterät menetelmät

Tässä luvussa kerrotaan ketterien menetelmien syntyhistoriasta, pääperiaatteista ja tavotteista. Luvussa vertaillaan myös hieman perinteisiä ja ketteriä menetelmiä.

90-luvun loppupuolella useat ohjelmistokehitysmetodologiat alkoivat saamaan yhä enemmän huomiota julkisuudessa. Nämä metodologiat olivat erilaisia yhdistelmiä vanhoista ja uusista, sekä muunnelluista vanhoista ideoista. Yhteistä näille metodologioille oli läheinen yhteistyö ohjelmointitiimin ja liiketoiminnan ammattilaisten kesken, kasvokkain tapahtuva kommunikointi (tehokkaampaa kuin kirjoitettu dokumentaatio), säännöllinen uuden vastineen tuottaminen asiakkaan investoinneille, tiiviit ja itseorganisoituvat tiimit, sekä ohjelmointikäytännöt ja tiimit, jotka kykenevät vastaamaan muuttuviin vaatimuksiin. [2]

Vuoden 2001 alussa pidettiin Utahin Snowbirdissä, USA:ssa, työpaja, jossa monet näiden eri metodologioiden luojista ja harjoittajista kokoontuivat pohtimaan mitä näissä metodologioissa oli yhteistä [2]. He valitsivat termin 'ketterä' (Agile) kuvaamaan näitä metodologioita ja laativat ketterän ohjelmistokehityksen manifestin (Agile-manifest) [5]. Tämän manifestin tärkein osa on julkilausuma, jossa ilmaistaan yhteiset ohjelmistokehityksen arvot:

Yksilöt ja vuorovaikutukset vs. prosessit ja työkalut

Toimiva ohjelmisto vs. kattava dokumentaatio

Yhteistyö asiakkaan kanssa vs. sopimusneuvottelut

Muutoksiin vastaaminen vs. suunnitelman noudattaminen

Vaikka tämän listan oikeanpuoleisillakin kohdilla on arvoa, ketterissä menetelmissä arvostetaan enemmän vasemman puoleisia kohtia. [5]

Agile-manifestin taustalla on 12 periaatetta [5]:

> Tärkeintä on täyttää asiakkaan vaatimukset julkaisemalla jatkuvasti ja aikaisin uusia hyödyllisiä versioita ohjelmistosta.
>
> Hyväksytään ja otetaan vastaan muuttuvat vaatimukset, jopa kehityksen loppuvaiheessa. Ketterät menetelmät valjastavat muutoksen asiakkaan kilpailueduksi.

Luovutetaan toimivia versioita kehitettävästä ohjelmistosta säännöllisesti, mielellään lyhyin väliajoin muutamasta viikosta muutamaan kuukauteen.

Liiketoiminnan ammattilaisten ja kehittäjien täytyy työskennellä päivittäin yhdessä koko projektin ajan.

Rakennetaan projektit motivoituneiden yksilöiden ympärille ja annetaan heille ympäristö ja tuki jota he tarvitsevat, sekä luotetaan, että he saavat työn tehtyä.

Kaikkein tehokkain tapa välittää tietoa kehitystiimille ja kehitystiimissä, on kasvokkain tapahtuva keskustelu.

Toimiva ohjelmisto on ensisijainen edistymisen mitta.

Ketterät menetelmät suosivat kestävää kehitystä. Rahoittajien, kehittäjien ja käyttäjien tulisi kyetä pitämään jatkuvasti yllä tasainen työtahti.

Jatkuva huomion kiinnittäminen tekniseen laatuun, sekä hyvään rakenteeseen ja suunnitteluun, lisää ketteryyttä.

Yksinkertaisuus – taito maksimoida työn määrä, jota ei tarvitse tehdä – on olennaista.

Parhaat arkkitehtuurit, vaatimukset ja suunnitelmat nousevat itseorganisoituvista tiimeistä.

Tasaisin väliajoin tiimi miettii miten voisi tulla entistä tuottavammaksi, ja sitten säätää ja muokkaa toimintaansa sen mukaisesti.

3.1 Perinteisten ja ketterien menetelmien vertailua

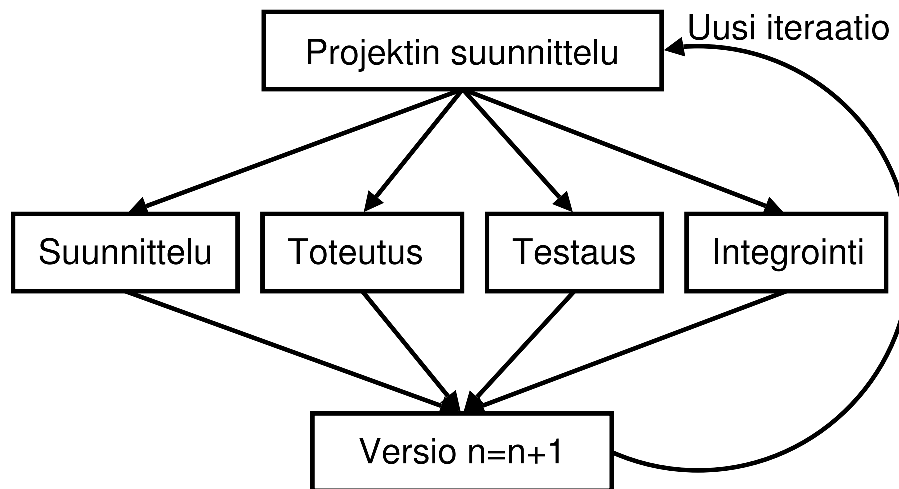
Perinteisten menetelmien ongelmana on niiden kyvyttömyys vastata projektin aikana muuttuviin vaatimuksiin. Mitä myöhemmässä vaiheessa projektia vaatimukset muuttuvat, sitä enemmän ne tulevat maksamaan. Ketterien menetelmien suurimpia vahvuuksia on taas juuri kyky reagoida muutoksiin myös projektin loppuvaiheessa.

Ketterissä menetelmissä toimivia versioita kehitettävästä ohjelmistosta toimitetaan asiakkaalle vähintäänkin muutaman kuukauden välein ja asiakas on koko ajan yhteistyössä kehittäjien kanssa. Asiakas tietää koko ajan missä ollaan menossa ja kehittäjät saavat asiakkaalta välitöntä palautetta kehittäessään ohjelmistoa. Perinteisissä menetelmissä taas asiakas pääsee yleensä näkemään lopputuloksen vasta

projektin loppuvaiheessa. Mikäli vaatimusmäärittely on ollut puutteellinen, lopputulos ei välttämättä vastaa tarpeeksi hyvin asiakkaan tarpeita.

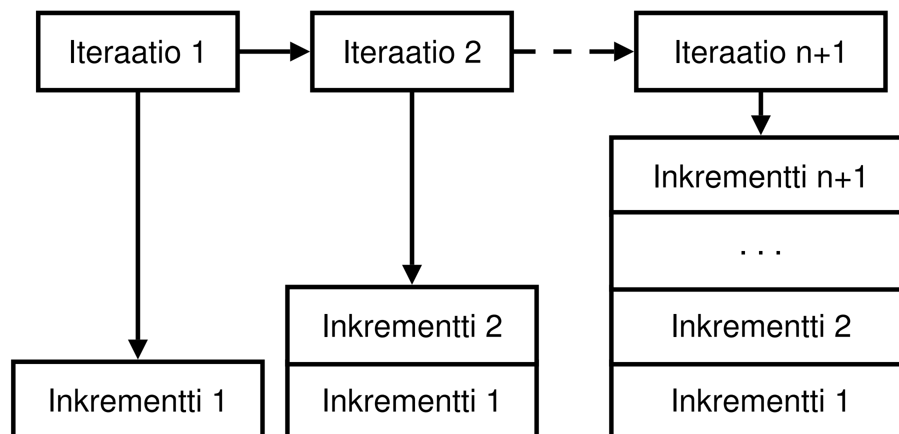
Dokumentointi on perinteisissä menetelmissä ollut tärkeässä osassa ja niitä tuotetaan koko projektin ajan. Ketterissä menetelmissä toimiva ohjelmakoodi on taas tärkeämpää kuin kattavat dokumentit.

4 Extreme Programming -metodologian esittely



Kuva 4.1: Extreme Programming -prosessi.

Extreme Programming (XP) on niin sanottu ketterä ohjelmistokehitys -metodologia, joka soveltuu erityisesti riskialttiille projekteille, joissa vaatimukset saattavat muuttua usein ja kehitystiimin koko on sopivan pieni. XP on kevyt metodologia yhdistäen olemassa olevien ohjelmistokehitysprosessien käytäntöjä niin, että kehittäjät ovat vapautetut tarpeettomasta työstä (kuten esim. kattavasta dokumentoinnista) ja voivat keskittyä oleelliseen: laadukkaan koodin kirjoittamiseen. [30]



Kuva 4.2: XP on iteratiivinen ja inkrementaalinen.

XP:n prosessi on iteratiivinen ja inkrementaalinen 4.2 [30]. Iteratiivisyys tarkoittaa, että prosessi etenee vaiheittain eteenpäin. Yhden vaiheen tuotoksena on inkrementti, joka koostuu viimeisimmän ja sitä edeltävien vaiheiden tuotoksista. XP ei ole kuitenkaan puhtaasti inkrementaalinen, sillä vaiheiden tuotoksina ei ole ainoastaan uusia lisäyksiä olemassa olevaan ohjelmakoodiin, vaan myös aiempaa ohjelmakoodia muokataan jatkuvasti paremmaksi 4.2.6. Kun vesiputousmallissa 2.2 suunnittelu, toteutus, testaus ja integrointi olivat peräkkäisiä vaiheita, XP:ssä näitä kaikkia tehdään yhden vaiheen aikana tavallaan yhtä aikaa 4.1.

XP:n prosessi alkaa yksinkertaisesta rakenteesta, joka täyttää alun vaatimukset ja kehittyy tasaisesti haluttuun joustavuuteen poistaen tarpeettoman monimutkaisuuden. Pyrkimyksenä on tuottaa yksinkertaisin mahdollinen ratkaisu, joka täyttää aina sen hetkiset vaatimukset. [30]

4.1 XP:n neljä perusarvoa

Extreme Programming -metodologian perustana on neljä arvoa: kommunikointi, yksinkertaisuus, palaute ja rohkeus. [19]

XP:n arvot ovat yhtenevät ketterien menetelmien manifestin [5] kanssa, ja voidaan sanoa että juuri nämä yhteiset arvot tekevät XP:stä ketterän menetelmän [3]. Nämä XP:n arvot auttavat ymmärtämään paremmin XP:n käytäntöjä.

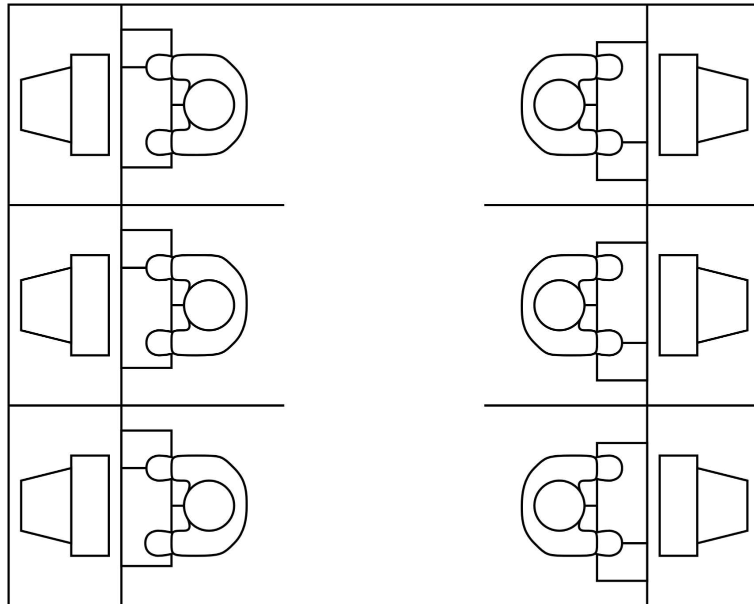
Kommunikointi: XP:n työtavat vaativat kunnollista kommunikaatiota ihmisten välillä. Suurin osa kommunikaatiosta tulisi olla suoraan kasvokkain tapahtuvaa keskustelua. Tämä pyritään mahdollistamaan perinteisten sermeillä jaettujen työtilojen 4.3 sijaan avoimella työtilalla 4.4.

Yksinkertaisuus: XP painottaa yksinkertaisen rakenteen tärkeyttä. Beckin mukaan ohjelmistojen, joita voidaan pitää yksinkertaisina, tulee noudattaa seuraavia kriteerejä: ohjelmakoodi ajaa kaikki testit, kommunikoi ohjelmoijalle kaiken tarvittavan, kopioitua koodia ei ole, ja se sisältää ainoastaan tarvittavan minimimäärän luokkia ja metodeja [4].

Palaute: Kehittäjien ja asiakkaiden tulisi saada palautetta järjestelmän tilasta niin usein kuin mahdollista [19]. Palautetta kerätään usein ja heti tehdyn työn jälkeen. Sitä kerätään toimittajalta, asiakkaalta sekä itse rakennettavalta ohjelmistolta [4].

Rohkeus: Tämä arvo perustuu tosiasialle, että kehittäjien tulee kyetä näkemään, että kehitysprosessi on ajautunut väärään suuntaan ja korjaukset ovat välttämät-

tömiä. Ongelmien korjaaminen saattaa tarkoittaa monen päivän töiden heittämistä hukkaan ja koodin uudelleenkirjoittamista, vaikka se olisikin aiemmin läpäissyt testit. [19]



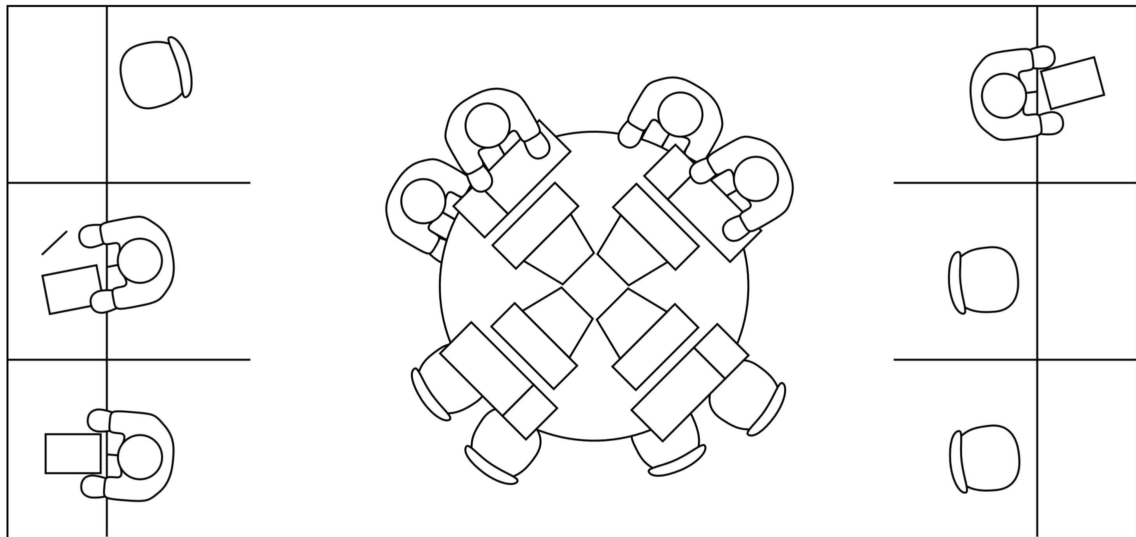
Kuva 4.3: Perinteinen sermeillä jaettu työtila.

4.2 XP:n käytännöt

XP:n arvoja toteutetaan ja sovelletaan käytäntöön noudattamalla XP:n käytäntöjä [19]. XP:n käytännöt eivät kuitenkaan varsinaisesti määrittele XP:tä, vaan sen määrittelevät XP:n säännöt 4.3 [9].

Kun Extreme Programming -metodologia esiteltiin ensimmäisen kerran vuonna 1999, se koostui 12 käytännöstä: suunnittelupeli (Planning Game), pienet julkaisut (Small Releases), järjestelmän metafora (System Metaphor), yksinkertainen rakenne ja suunnittelu (Simple Design), testaus (Testing), uudelleenrakentaminen (Refactoring), pariohjelmointi (Pair Programming), yhteisomistajuus (Collective Ownership), jatkuva integrointi (Continuous Integration), 40-tuntinen työviikko (40-Hour Week), paikan päällä oleva asiakas (On-site Client) ja koodausstandardit (Coding Standards) [12].

Mitään virallisia muutoksia näihin XP:n alkuperäisiin käytäntöihin ei ole vielä tehty [9], mutta epävirallisesti joitakin XP:n käytäntöjä on nimetty uudelleen, sekä



Kuva 4.4: XP:n avoin työtila.

muutama uusi käytäntö on lisätty entisten rinnalle. Alkuperäiset 12 käytäntöä keskittyivät hyvin pitkälle itse ohjelmointiin, mutta vähemmän liiketoimintaan.

Collins ja Miller [9] esittelevät 19 käytäntöä, jotka he ovat jaotelleet neljään kategoriaan: yhteiset käytännöt, ohjelmoijien käytännöt, hallinnon käytännöt, sekä asiakkaan käytännöt. Yhteiset käytännöt ovat tarkoitettu kaikkien projektin jäsenten noudatettaviksi ja niiden tarkoituksena on tuoda projektin eri ryhmät yhteen luoden projektin jäsenistä yhden yhtenäisen tiimin. Ohjelmoijien käytännöt kertovat miten ohjelmoijien täytyy toimia, että he saavat kehitettyä tiimin haluaman järjestelmän. Hallinnon käytännöt ohjaavat hallinnon ihmisiä luomaan tarvittavat puitteet projektille, että projektin jäsenet voivat tehdä oman työnsä hyvin. Asiakkaan käytännöt keskittyvät vaadittavien ominaisuuksien määrittelemiseen ja toteutettujen ominaisuuksien tarkastamiseen.

Nämä 19 käytäntöä pyrkivät ottamaan huomioon paremmin myös hallinnon näkökulman. Taulukossa 4.1 esitellään näiden käytäntöjen nimet ja kategoriat, joihin nämä käytännöt kuuluvat. Uusia käytäntöjä Collinsin ja Millerin [9] esittämässä listassa ovat: iteraatiot, avoin työtila, aiemmista kokemuksista oppiminen, hyväksytty vastuu, taustatuki, neljännesvuosikatsaus, tarinoiden kerronta, sekä hyväksyntätestaus. Osa näistä uusista käytännöistä on sisältynyt aiempiin käytäntöihin, kuten esimerkiksi testauksen käytäntö on pitänyt sisällään hyväksyntätestauksen, ja suunnittelupeli on sisältänyt tarinoiden kerronnan.

Koska virallisia muutoksia XP:n alkuperäisiin käytäntöihin ei ole tehty, ja useimmissa XP:n soveltamista ohjelmistotekniikan opetuksessa käsittelevissä tutkimuk-

Yhteiset käytännöt	Iteraatiot Yhteinen sanasto (metafora) Avoin työtila Aiemmista kokemuksista oppiminen
Ohjelmoijien käytännöt	Testilähtöinen ohjelmointi (testaus) Pariohjelmointi Uudelleenrakentaminen Yhteisomistajuus Jatkuva integrointi Et tule tarvitsemaan sitä (yksinkertainen rakenne)
Hallinnon käytännöt	Hyväksytty vastuu Taustatuki Neljännesvuosikatsaus Peili Tasainen työtahti (40-tuntinen työviikko)
Asiakkaan käytännöt	Tarinoiden kerronta Julkaisujen suunnittelu (suunnittelupeli) Hyväksyntätestaus Lyhyin väliajoin tuotettavat julkaisut (pienet julkaisut)

Taulukko 4.1: XP:n 19 käytäntöä Collinsin ja Millerin [9] mukaan

sissa on käytetty 12 alkuperäistä käytäntöä, niin myös tässä tutkielmassa keskitytään näihin 12 käytäntöön.

4.2.1 Suunnittelupeli

"XP suunnittelu keskittyy kahteen avainkysymykseen ohjelmiston kehityksessä: ennakoimiseen, mitä saadaan valmiiksi määräpäivään mennessä, ja päättämiseen, mitä tehdään seuraavaksi. Pääpaino on projektin ohjaamisessa – mikä on melko helppoa – ennemmin kuin tarkassa ennustamisessa, mitä tullaan tarvitsemaan ja kuinka kauan siinä kestää – mikä on melko vaikeaa." [17] XP:ssä on kaksi suunnitteluvaihetta, jotka pyrkivät antamaan vastauksen näihin kahteen avainkysymykseen.

Julkaisun suunnittelu (Release Planning) on käytäntö, missä asiakas esittää ohjelmoijille halutut toteutettavat ominaisuudet ja ohjelmoijat arvioivat niiden vaikeusasteet. Kun asiakkaalla on tiedossa toteuttavien ominaisuuksien kustannukset, hän määrittelee suunnitelman projektille sen mukaan mitkä ominaisuudet ovat tärkeimpiä. Nämä alustavat suunnitelmat ovat luonnollisesti epätarkkoja, sillä eri ominaisuuksien prioriteetit eivätkä näiden vaativuusarviot voi olla täysin luotettavia. Vasta kun projektiryhmä aloittaa ominaisuuksien toteutuksen, voidaan saada varmempaa tietoa siitä, kuinka nopeasti he kykenevät etenemään. Julkaisusuunnitelma tarkentuu ajan kuluessa, mutta jo ensimmäinenkin julkaisusuunnitelma on tarpeeksi tarkka, että voidaan tehdä päätöksiä projektin suhteen. [17]

Iteraation suunnittelu (Iteration Planning) on käytäntö, jonka mukaan tiimin etenemissuunta tarkistetaan joka toinen viikko. XP-tiimi kehittää ohjelmistoa kahden viikon iteraatioissa, joiden lopputuloksena on ajettava ja käyttökelpoinen sovellus. Iteraation suunnittelussa asiakas esittää ominaisuudet, jotka hänen mielestään olisi tärkeintä seuraavan kahden viikon aikana toteuttaa. Ohjelmoijat jakavat nämä ominaisuudet, eli käyttäjätarinat, pienempiin toteutettaviin tehtäväkokonaisuuksiin ja arvioivat niiden vaativuudet paremmalla tarkkuudella kuin julkaisun suunnittelussa. Mikäli edellisessä iteraatiossa on joidenkin ominaisuuksien toteutus jäänyt kesken, otetaan niiden vaatima työmäärä huomioon uusia tehtäviä valittaessa. [17]

XP-prosessin eteneminen on hyvin näkyvää ja asiakkaalla säilyy koko ajan kontrolli projektin etenemissuunnasta. Mikäli asiakas ei ole tyytyväinen projektin etenemiseen, hän voi lopettaa koko projektin. Mutta toisaalta koska eteneminen on niin näkyvää ja seuraavaksi toteutettavista asioista voidaan päättää niin joustavasti, XP-projekteilla on tapana antaa parempia tuloksia vähemmällä stressillä. [17]

4.2.2 Pienet julkaisut

Pienten julkaisujen käytäntöä noudatetaan kahdella tavalla: tiimi julkaisee ajettavan ja testatun ohjelmiston jokaisen iteraation lopussa antaen sen asiakkaan käyttöön, ja toiseksi XP-tiimi julkaisee uusia versioita säännöllisesti myös varsinaisille loppukäyttäjille. [17]

Asiakas voi käyttää uusinta, edellisen iteraation lopussa julkaistua, versiota ohjelmistosta niin kuin parhaaksi näkee. Hän voi arvioida sen toimintaa ja jopa antaa sen loppukäyttäjille käytettäväksi. Kun ohjelmisto annetaan jokaisen iteraation jälkeen asiakkaalle, kaikki on läpinäkyvää ja asiakas saa jatkuvasti vastinetta projektiin sijoittamilleen varoille. [17]

Varsinaisille loppukäyttäjille tarkoitettuja julkaisuja julkaistaan myös hyvin usein. WWW-projekteja julkaistaan jopa päivittäin ja talon sisäisiä kuukausittain tai vieläkin useammin. Pakattujakin tuotteita toimitetaan uusia neljännesvuosittain. [17]

Kunnollisten toimivien versioiden julkaiseminen näin usein on mahdollista jatkuvan integroinnin, asiakkaan tekemien testien ja testilähtöisen kehityksen ansiosta. [17]

4.2.3 Järjestelmän metafora

Järjestelmän metafora on yksinkertainen yhteinen tarina, joka auttaa ymmärtämään järjestelmän toimintaa ja sen osien välisiä suhteita. Tämä auttaa kehittäjiä ja asiakkaita puhumaan samaa kieltä ja ymmärtämään toisiaan. [19] Esimerkiksi metafora agenttipohjaiselle tiedonhakujärjestelmälle voisi olla seuraava: ”tämä ohjelma toimii kuin mehiläispesä: mehiläiset lähtevät ulos pesästä hakemaan mettä ja tuovat sen mukanaan pesään”. [17]

Vaikka sopivan runollista metaforaa ei onnistuttaisikaan keksimään, kuitenkin XP-tiimit käyttävät yhteisiä nimityksiä järjestelmästä ja sen osasista. Näin voidaan varmistua siitä, että kaikki ymmärtävät kuinka järjestelmä toimii, mistä jotain tiettyä toiminnallisuutta pitäisi etsiä, tai mikä olisi oikea paikka lisätä uutta toiminnallisuutta. [17]

4.2.4 Yksinkertainen rakenne ja suunnittelu

Järjestelmä pyritään pitämään alusta alkaen niin yksinkertaisena kuin mahdollista. Ylimääräiset monimutkaisuudet koetetaan löytää ja poistaa mahdollisimman nopeasti. [19]

Ohjelmoijatestauksen 4.2.5 ja uudelleenrakentamisen 4.2.6 avulla pidetään huoli siitä, että ohjelmakoodi ei pääse 'rämettymään', vaan että sen rakenne soveltuu aina senhetkiseen toiminnallisuuteen. [17]

"Suunnittelu ei ole XP:ssä vain yhdessä vaiheessa tai etukäteen tehtävä asia, vaan sitä tehdään jatkuvasti. Suunnittelua tehdään julkaisun suunnittelussa ja iteraation suunnittelussa, sekä tiimien lyhyissä suunnittelutapaamisissa ja ohjelmakoodin uudelleenrakentamisessa. Suunnittelua tehdään käytännöllisesti katsoen läpi koko projektin. Iteratiivisessa ja inkrementaalisessa prosessissa, kuten Extreme Programming, hyvä suunnittelu ja rakenne on välttämätöntä. Tämän takia suunnitteluun keskitytään niin paljon koko projektin ajan." [17]

4.2.5 Testaus

Jatkuvasti uuden koodin kirjoittamisen tai vanhan koodin muuttamisen jälkeen ajetaan vaadittavat testit, mukaan lukien ohjelmoijatestit ja asiakkaan kirjoittamat hyväksyntätestit. [19]

Kun asiakas tuo esille uuden halutun ominaisuuden, hän määrittelee yhden tai useamman automaattisen hyväksyntätestin, joiden avulla voidaan varmistua, että uusi ominaisuus toimii halutulla tavalla. Näiden testien avulla tiimi voi osoittaa itselleen ja asiakkaalle, että ominaisuus on toteutettu oikein. Automaattiset testit ovat tärkeitä, koska ajanpuutteen takia manuaalisesti tehtävät testit jäävät helposti tekemättä. [17]

On hyvä suhtautua asiakastesteihin samalla tavoin kuin ohjelmoijatesteihinkin, eli kun testit saadaan menemään läpi, pidetään huoli että jatkossakin ne menevät läpi ilman ongelmia. Näin järjestelmä kehittyy koko ajan ja menee eteenpäin. [17]

Extreme Programming -metodologia painottaa palautteen tärkeyttä ja ohjelmistokehityksessä tämä vaatii hyvää testausta. Hyvät XP-tiimit noudattavat testilähtöisen ohjelmoinnin käytäntöä, mikä tarkoittaa sitä, että työskennellään lyhyissä jaksoissa lisäten uusi testi ja tehden se toimimaan. Näin saadaan tuotettua ohjelmakoodia, jolle on olemassa lähes 100 prosentin testikattavuus. [17]

Nämä automaattiset ohjelmoijatestit on kaikki koottu yhteen ja aina kun ohjelmoija laittaa uutta ohjelmakoodia versionhallintajärjestelmän koodivarastoon, jokaisen ohjelmoijatestin on mentävä läpi virheittä. Näin ohjelmoija voi olla koko ajan varma että hänen tekemänsä uudet toiminnallisuudet integroituvat yhteen muun ohjelmakoodin kanssa, ja että ohjelmakoodin rakenteeseen tehdyt parannukset eivät ole rikkoneet mitään. [17]

4.2.6 Uudelleenrakentaminen

Uudelleenrakentaminen tarkoittaa ohjelman rakenteen muokkaamista paremmaksi muuttamatta sen toiminnallisuutta. Samanlaisina toistuvia ohjelmakoodin osia vähennetään, ohjelmakoodin ymmärrettävyyttä parannetaan, koodia yksinkertaistetaan ja muokataan joustavammaksi. [4]

Ohjelmakoodin 'koheesiota' lisätään samalla kun sen 'kytkentöjä' vähennetään [17]. Mikäli jokin moduuli keskittyy vain yhteen tehtävään, on sen koheesio suuri, ja jos eri komponenttien ja datan välillä on vain vähän riippuvuussuhteita, antaa kytkentä-mitta pieniä arvoja.

Korkea koheesio ja vähäiset kytkennät ovat yksi merkki hyvin suunnitellusta ohjelmakoodista. Kehitys aloitetaan hyvällä ja yksinkertaisella rakenteella ja projektin edetessä uudelleenrakentamisella varmistetaan, että ohjelmakoodin rakenne pysyy koko ajan yksinkertaisena ja soveltuvana senhetkiseen tilanteeseen. [17]

Kattavat testit 4.2.5 ovat vahvana tukena uudelleenrakentamisessa. Kun ohjelmakoodin rakennetta muokataan, testien ajamisella voidaan varmistaa, ettei mitään ole mennyt muutosten seurauksena rikki. Muun muassa juuri tämän takia käyttäjätestit ja ohjelmoijatestit ovat kriittisen tärkeitä. [17]

4.2.7 Pariohjelmointi

Kaikki tuotantoon menevä koodi kirjoitetaan kahden kehittäjän työskennellessä vierekkäin samalla koneella. Toinen parista ohjelmoi ja toinen etsii virheitä korjaten ne saman tien. Ohjelmoijat keskustelevat keskenään parhaista toteutusvaihtoehdoista. Tämä käytäntö varmistaa, että kaikki tuotantokoodi on ainakin yhden muun ohjelmoijan tarkastama, mikä taas johtaa parempaan rakenteeseen, testaukseen ja ohjelmakoodin laatuun [17]. [4]

Vaikka pariohjelmointi saattaakin vaikuttaa tehottomalta, pariohjelmoinnista tehdyt tutkimukset osoittavat, että pari tuottaa laadukkaampaa ohjelmakoodia lähes samassa ajassa, kuin jos ohjelmoijat työskentelisivät yksittäin [17, 8]. Jotkut ohjelmoijat vastustavat pariohjelmointia, vaikka eivät ole koskaan kunnolla kokeilleet sitä, mutta 90 prosenttia niistä ohjelmoijista, jotka ovat oppineet käytännön, pitävät sitä parempana [17].

Pareittain ohjelmointi ei ainoastaan auta tuottamaan laadukkaampaa ohjelmakoodia, vaan se auttaa myös jakamaan tietoa ja osaamista ohjelmointitiimin kesken. Parien vaihtuessa ohjelmoijat voivat oppia toistensa erityisosaamisesta. Ohjelmoijat oppivat, heidän taitonsa kehittyvät, ja heistä tulee entistä hyödyllisempiä sekä tiimille että yritykselle. [17]

4.2.8 Yhteisomistajuus

Kaikki kehittäjät ovat vastuussa kaikesta kehitettävän järjestelmän ohjelmakoodista ja näin ollen voivat myös tehdä muutoksia mihin tahansa ohjelmakoodin osaan aina kun on tarvetta [4]. Koko ohjelmakoodi on monen ihmisen tarkkailun kohteena. Tämä parantaa ohjelmakoodin laatua ja vähentää virheiden mahdollisuutta. [17]

Jos ohjelmakoodi on vain yksittäisten henkilöiden omistuksessa, vaaditut ominaisuudet tulevat helposti laitettua väärään paikkaan. Kun ohjelmoija huomaa tarvitsevansa uutta ominaisuutta sellaiseen ohjelmakoodin osaan, jota hän ei omista, ja mikäli tämän ohjelmakoodin omistaja on liian kiireinen toteuttaakseen vaaditun ominaisuuden, ohjelmoija joutuu lisäämään tarvitsemansa ominaisuuden omaan ohjelmakoodiinsa, vaikka se ei sinne loogisesti kuuluisikaan. Tästä on seurauksena vaikeasti ylläpidettävää ja rumaa ohjelmakoodia, joka on täynnä turhaa toistoa. [17]

Jos kehittäjät muokkaisivat ohjelmakoodia, jonka toimintaa he eivät kunnolla ymmärrä, ohjelmakoodin yhteisomistajuus voisi koitua ongelmaksi. XP:ssä tällaiset ongelmat vältetään kattavilla testitapauksilla, sekä niin että kehittäjä joka ei ole kunnolla perillä ohjelmakoodin osion toiminnasta, lähtee muokkaamaan sitä kokeneemman parin kanssa. Näin osaaminen leviää tiimin kesken ja muutokset ohjelmakoodiin tulee tehtyä kunnolla. [17]

Kun kaikki tiimin jäsenet ovat perillä kehitettävän ohjelmakoodin eri osista, ei projekti kaadu siihen, vaikka joku ryhmän jäsenistä joutuisikin lähtemään projektista.

4.2.9 Jatkuva integrointi

Kaikki muutokset integroidaan heti niiden toteuttamisen jälkeen, useita kertoja päivässä. Ohjelmiston kehittäjät työskentelevät myös kaikki samassa kehityshaarassa, mikä tarkoittaa sitä, että eri kehittäjille ei ole versionhallintajärjestelmässä omia ohjelmakoodivarastojaan, vaan kaikki työskentelevät saman ohjelmakoodin parissa. [19]

Jos ohjelmakoodi integroidaan vain harvoin, esimerkiksi viikon välein, integroinnista tulee helposti hankala ja aikaa vievä operaatio. Integroinnissa tulee helposti esiin virheitä ja ongelmia, joita ei ole ennen integrointia tehdyissä testeissä ilmennyt. Monesti on vielä niin, että integrointi annetaan sellaisen henkilön tehtäväksi, joka ei tunne kunnolla koko järjestelmää. Tällainen johtaa virheelliseen ohjelmakoodiin ja ajan hukkaan, kun ohjelmoijat joutuvat odottamaan joskus pitkiäkin aikoja ennenkuin voivat toteuttaa joitakin uusia ominaisuuksia, jotka riippuvat toisista,

vielä integroimattomista, järjestelmän osista. [17]

4.2.10 40-tuntinen työviikko

Kenenkään ei tulisi työskennellä yli 40 tuntia viikossa, eikä kenenkään tulisi tehdä ylitöitä kahta viikkoa peräkkäin. Näin kehittäjät pysyvät virkeinä, luovina, sekä kiinnostuneina työstä. [4, 19]

XP:ssä painotetaan laadukkaan ja rakenteeltaan yksinkertaisen ohjelmakoodin tärkeyttä. Jos kehittäjät ovat väsyneitä, he eivät kykene tuottamaan tällaista laadukasta ohjelmakoodia. Lyhyellä tähtäimellä suuri määrä ylitöitä voi näyttääkin tuottavan haluttua tulosta, mutta pitkällä tähtäimellä se voi kostautua loppuunpalamisena ja kiinnostuksen loppumisena.

4.2.11 Paikan päällä oleva asiakas

XP-tiimiin kuuluu oleellisena osana aina paikan päällä oleva asiakas. Asiakas on kehitystiimin osana oman alansa asiantuntijana auttaen kehittäjiä järjestelmän toteuttamisessa [4]. Asiakas antaa vaatimukset kehitettävälle järjestelmälle, asettaa prioriteetit halutuille uusille ominaisuuksille ja ohjaa projektin kulkua. [17]

Asiakas on sijoitettuna samaan paikkaan kehittäjien kanssa, jolloin kehittäjät voivat saada häneltä nopeasti palautetta ja vastauksia esiin tulleisiin kysymyksiin [19]. Kun asiakas on fyysisesti paikan päällä, kehittäjät voivat näyttää hänelle heti kuinka he aikovat kehittää jonkin ominaisuuden. Asiakas voi antaa näin välitöntä palautetta kehittäjille. Mikäli asiakas ei olisi fyysisesti paikan päällä, kehittäjien täytyisi mahdollisesti lähettää sähköpostia tai soittaa puhelimella hänelle. Kynnys tällaisten viestien lähettelyyn on monesti sen verran korkea, että kehittäjät eivät vaivautuisi kysymään asiakkaan mielipidettä, vaan tekisivät omia olettamuksiaan asioista.

Joku kehitystiimin jäsen, mahdollisesti testauksesta vastuussa oleva kehittäjä, auttaa asiakasta laatimaan automaattisia hyväksyntätestejä 4.2.5, ja joku toinen tiimin jäsen voi auttaa asiakasta määrittelemään sovellukselle asetettavia vaatimuksia [17]. Asiakkaan ei siis tarvitse välttämättä olla itse ohjelmointitaitoinen, tai muutenkaan vahva tietoteknisessä osaamisessa, vaan hänen on tärkeää tietää mitä sovelluksen loppukäyttäjä sovellukselta vaatii ja kuinka sovellus voisi tätä parhaiten palvella.

4.2.12 Koodausstandardit

Kehittäjät kirjoittavat kaiken koodin kehitystiimin kesken sovittujen standardien mukaisesti. Tämä mahdollistaa sen, että koodia on helpompi lukea ja ymmärtää. [4]

Tarkoituksena olisi, että kaikki ohjelmakoodi näyttää siltä, kuin sen olisi kirjoittanut yksi asiantunteva ja ammattitaitoinen henkilö. Koodausstandardin yksityiskohdat eivät ole niin tärkeitä, vaan tärkeintä on se, että kaikki ohjelmakoodi näyttää jokaiselle kehittäjälle tutulta ja näin ollen tukee ohjelmakoodin yhteisomistajuutta 4.2.8. [17]

4.3 XP:n säännöt

Auerin, ym. [3] mukaan XP:n käytännöt eivät vielä määrittele XP:tä. He vertaavat ohjelmistokehitystä pesäpallon pelaamiseen, ja samalla tavalla kuin pesäpallo on määritelty sen sääntöjen avulla, myös XP tulisi määritellä sen sääntöjen avulla.

Mikäli XP halutaan määritellä sen käytäntöjen avulla, ongelmaksi muodostuu ainakin kaksi asiaa. Koska käytäntöjä ei ole määritelty täsmällisesti, voi joku tiimi noudattaa kaikkia käytäntöjä, silti tekemättä varsinaisesti XP:tä. Toisaalta taas XP:tä voi noudattaa ilman että noudattaa kaikkia käytäntöjä. Käytännöt kyllä voivat johtaa XP:n noudattamiseen, mutta ne eivät varsinaisesti määrittele XP:tä. [3]

Auer, ym. [3] jakavat XP:n säännöt osallistumisen ja pelaamisen sääntöihin. Osallistumisen säännöt (Rules of Engagement) määrittelevät 'pelille' tarvittavat puitteet, joita ilman pelaaminen ei voi alkaa. Pelaamisen säännöt (Rules of Play) taas tapahtuvat osallistumisen sääntöjen kontekstissa ja määrittelevät yksityiskohtaisemmin pelin kulkua koskevat säännöt.

Auerin, ym. [3] mielestä XP:n prosessia voisi paremminkin kutsua nimellä 'Extreme Software Development'. Pelaamisen sääntöjen noudattamista taas voitaisiin kutsua nimellä 'Extreme Programming', jolloin kehitystiimi voisi tehdä XP:tä, vaikka osallistumisen sääntöjä ei noudatettaisikaan.

4.3.1 Osallistumisen säännöt

Auerin, ym. [3] mukaan XP:n osallistumisen säännöt ovat hyvin pitkälle vastaavat muiden ketterien menetelmien kanssa. Tässä on lista näistä XP:n osallistumisen säännöistä:

O1. XP-tiimi koostuu ohjelmistotuotteen kehitykseen omistautuneesta ryhmästä

ihmisiä. Tämä tuote voi olla, tai sitten ei, osa laajempaa tuotetta. Tiimissä voi olla monia rooleja, kuitenkin vähintään asiakkaan ja kehittäjän roolit.

O2. Asiakkaan tulee asettaa, ja jatkuvasti mukauttaa tavoitteita ja prioriteetteja perustuen kehittäjien tai muiden tiimin jäsenten antamiin arvioihin sekä muuhun informaatioon. Tavoitteet määritellään niin, että ne vastaavat kysymyksiin: mitä ja miten.

O3. Asiakas on aina tavoitettavissa ja auttaa kehittäjiä arvioiden laatimisessa tai toivottujen tuotosten toimittamisessa antamalla heille aina tarvittavat tiedot. Asiakas on kiinteä osa tiimiä.

O4. Jokaisen tiimin jäsenen on milloin tahansa kyettävä arvioimaan tiimin edistymistä suhteessa asiakkaan tavoitteisiin.

O5. Tiimin täytyy toimia tehokkaana sosiaalisena verkostona, mikä tarkoittaa:

   a. Rehellistä kommunikaatiota, joka johtaa jatkuvaan oppimiseen.
   b. Mahdollisimman vähäinen ero projektin edistymiseen tarvittavien asioiden ja nämä tarpeet täyttävien ihmisten/resurssien välillä.
   c. Valtuuksien ja vastuun tasaaminen.

O6. 'Timeboxing'-tekniikkaa käytetään kehitystyön eri segmenttien tunnistamiseen. Mikään tällainen segmentti ei ole kestoltaan enempää kuin yksi kuukausi.

4.3.2 Pelaamisen säännöt

Kun XP:n osallistumisen säännöt olivat vastaavanlaiset muiden ketterien menetelmien kanssa, niin XP:n pelaamisen säännöt taas erottavat oleellisesti XP:n muista ketteristä menetelmistä [3].

P1. Työn tulokset on jatkuvasti validoitava testauksen avulla.

P2. Kirjoita yksikkötestit ensin, ennen ohjelmoimista. Ohjelmoi pareittain, mikäli tiimissä on useampi kuin yksi kehittäjä. Uudelleenrakenna, asiakkaan vaatimusten toteuttamisen ohella, ohjelmakoodi täyttämään koodaussäännöt (P3).

P3. Kaiken ohjelmakoodin, jota mahdollisesti käytetään lopputuotteessa, tulee

   a. Läpäistä kaikki yksikkötestit

b. Ilmaista selkeästi kaikki käsitteet
c. Olla vapaa toistosta
d. Olla vapaa ylimääräisistä osista

P4. Yhteisomistajuus. Jokaisella on valtuudet ja vähintään kahdella ihmisellä on tarvittava ymmärrys minkä tahansa tehtävän suorittamiseen.

5 Kokemuksia XP:n soveltamisesta ohjelmoinnin opetukseen

XP:tä ei olla, Suomessa tai muuallakaan maailmassa, sovellettu vielä paljoakaan ohjelmoinnin opetuksessa. Joitakin tutkimuksia ja kokeiluja on maailmalla kuitenkin jo tehty. Yksi tutkimus tehtiin Karlsruhen yliopistossa käyttäen yliopisto-opiskelijoita, jotka osallistuivat käytännön harjoittelu -kurssille [19]. Toisessa tutkimuksessa tutkittiin Calgaryn yliopiston neljännen vuoden opiskelijoita, jotka olivat suorittamassa kahdeksan kuukauden mittaista sovellusprojektiaan [20]. Kolmas tutkimus tehtiin North Carolina State -yliopistossa, jossa 150 neljännen vuosikurssin opiskelijaa suorittivat ohjelmistotekniikan kurssia [31].

Valparaison yliopistossa XP:tä käytettiin ohjaavana metodologiana syventävien opintojen ohjelmiston suunnittelu -kurssilla. Kurssi oli yhden lukukauden mittainen ja sen tarkoituksena oli antaa opiskelijoille tietoa ja kokemusta ohjelmiston suunnittelu -metodologiasta sisältäen intensiivistä työtä projektiryhmässä. [18]

Opiskelijoiden näkemyksiä ketterien menetelmien ja erityisesti XP:n käytänteiden käytöstä opintoihin liittyvissä suunnittelu- ja ohjelmointitehtävissä on selvitetty Melnikin ja Maurerin tutkimuksessa [23]. Tutkimuksessa oli mukana 45 opiskelijaa, jotka olivat suorittamassa kolmea eritasoista akateemista tutkintoa kahdessa eri oppilaitoksessa. Myös Sandersin [28] tutkimuksessa on selvitetty opiskelijoiden näkemyksiä XP:n käytöstä opetuksessa.

Bevan, Werner ja McDowell [6] ovat tutkineet pariohjelmoinnin käyttöä ensimmäisen vuoden opiskelijoiden ohjelmointikurssilla.

Schneider ja Johnston [30] ovat käsitelleet julkaisussaan ketterien menetelmien, ja erityisesti XP:n, käyttöön liityviä kysymyksiä ja ongelmia kolmannen asteen koulutuksessa. Myös Melnik ja Maurer [24] ovat koonneet yhteen asioita, joita ketterien menetelmien käytöstä opetuksessa on opittu.

Swinburnen teknillisessä korkeakoulussa [19] viimeisen vuosikurssin projektiopinnoista yksi ryhmä valittiin viemään projekti läpi käyttäen XP-metodologiaa. Projektiryhmän tehtävänä oli kehittää ohjelmistoprojektin hallinnointi -simulaattori (simProject), jossa pelaajan tehtävänä on toimia projektipäällikkönä ja johtaa projekti päätökseen.

Wilson [39] on pitänyt The Johns Hopkins -yliopistossa kurssin 'Ohjelmiston kehitys XP:llä'. Kurssin tarkoituksena oli opettaa XP:n käytäntöjä kohtuullisen laajan

ja monimutkaisen ohjelmiston kehityksen avulla. Kurssilla oli 20 yliopisto-opintojensa loppuvaiheessa olevaa opiskelijaa, joista kaikki kehittivät samaa sovellusta. Projektissa tuotettu Smalltalk-kehitysympäristöjen tyylinen Java-kehitysympäristö julkaistiin GNU General Public -lisenssin alla.

Lopuksi esitellään vielä RoleModel Software -yrityksen käyttämä oppisopimusmalli, joka voisi olla myös varteenotettava vaihtoehto perinteiselle korkeakouluopiskelulle [26].

Seuraavaksi käydään käytännöittäin läpi näitä XP:n soveltamisesta saatuja kokemuksia.

5.1 Suunnittelupeli

"Tämä käytäntö (suunnittelupeli) auttoi meitä luomaan hahmotelman siitä, kuinka paljon töitä meidän tulisi saada tehtyä ja kuinka paljon se suurinpiirtein vaatisi vaivannäköä." [31]

North Carolina State -yliopistossa, neljän viikon kurssilla simuloitiin suunnittelupeliä luokkahuoneympäristössä. Kaikki kurssin 150 opiskelijaa osallistuivat tähän ja he saivat tehtäväkseen käydä läpi käyttäjätarinat ja valita niistä tärkeysjärjestyksessä sen verran tehtäviä, kun arvioivat kykenevänsä suorittamaan kurssin aikana. Tässä tutkimuksessa opetushenkilökunta toimi asiakkaan roolissa. [31]

Valparaison yliopiston kurssilla suunnittelupeliä ei toteutettu ollenkaan, vaan opiskelijoiden annettiin itse valita tarinat (ominaisuudet) jotka aikovat milloinkin toteuttaa. Tämä aiheutti opiskelijoille ongelmia ensimmäisissä julkaisuissa, eikä projektin päämäärä ollut heillä selkeänä mielessä. [18]

Swinburnen teknillisessä korkeakoulussa [19] järjestetyn projektikurssin XP:tä käyttänyt ryhmä näki suunnittelupelin pääosin positiivisena kokemuksena. Heidän mielestään suunnittelupelin positiivinen puoli oli se, että vaatimusten keräämisessä sovellukselle ei tarvinnut ottaa huomioon kaikkia muita vaatimuksia, vaan että sai keskittyä pieniin osiin kerrallaan. Toisaalta he näkivät tämän myös negatiivisena puolena, sillä he eivät tienneet, minkälaisia vaatimuksia sovellukselle tulee myöhemmin. Nämä opiskelijat olivat tottuneet perinteisempiin prosessimalleihin ja näin ollen myös siihen, että ennen suunnittelua ja toteutusta analyysivaiheessa kaikki vaatimukset on jo kerätty ja projektista on olemassa hyvä kokonaiskuva.

Swinburnen projektissa [19] asiakas osallistui tämän käytännön mukaisesti suunnittelupeleihin, mikä osoittautui projektiryhmän jäsenille tehokkaimmaksi tavaksi saada täysi ymmärrys projektin vaatimuksista. Opiskelijat saivat asiakkaalta suoraa palautetta ja saattoivat näin varmistaa, että he olivat varmasti ymmärtäneet vaati-

mukset oikein.

Ongelmia suunnittelupelin soveltamisessa Swinburnen projektiryhmän opiskelijoille tuotti eri tehtävien työmäärien arviointi. Heillä ei ollut vielä kokemusta monimutkaisempien tehtävien toteuttamisesta, eikä näiden työmäärien arvioinnista. Projektin alkuvaiheessa nämä opiskelijat yliarvioivat tehtävien suorittamiseen kuluvan ajan, jolloin he saivat toteutukset valmiiksi ennen suunniteltua aikataulua. Kun projekti eteni ja tehtävät tulivat monimutkaisemmiksi, he taas aliarvioivat tehtävien toteuttamiseen tarvitsemansa ajan. [19]

Ongelmallista Swinburnen projektiryhmälle oli myös käyttäjätarinoiden hajoittaminen tehtävätasolle. Tarinat olivat jo itsessään pieniä ja vaikka ne olisikin jaettu pienempiin tehtäviin, nämä olisivat olleet liian riippuvaisia muista tehtävistä, että ne olisi voitu toteuttaa omina yksittäisinä tehtäväkokonaisuuksinaan. [19]

The Johns Hopkins -yliopiston [39] kurssilla ei tehty projektin kokonaissuunnitelmaa, eikä julkaisujen aikataulusuunnitelmaa. Tähän oli syinä ajanpuute ja se, että ei ollut ulkopuolisia paineita saada sovellusta valmiiksi mihinkään tiettyyn päivämäärään mennessä.

The Johns Hopkins -yliopiston [39] kurssilla jokaisen iteraation alussa tehtiin XP:n mukaisesti suunnitelmat iteraatiota varten. Kurssin ohjaaja toimi asiakkaana priorisoiden käyttäjätarinat, ja opiskelijat arvioivat näiden tarinoiden vaativuudet, sekä valitsivat niistä itselleen toteutettavaksi haluamansa tarinat.

5.2 Pienet julkaisut

"Opiskelijat ovat tottuneet siihen, että he voivat nähdä edistymisen työskennellessään jossain projektissa. XP sallii opiskelijoiden kehittää sovellusta vähän kerrallaan. Nähdessään edistymisen, prosessi tuntuu opiskelijoista miellyttävämmältä." [28]

North Carolina State -yliopistossa järjestetyllä kurssilla opiskelijoita ohjattiin jakamaan julkaisu pienempiin iteraatioihin, joihin asiakkaina toimineet opettajat eivät kuitenkaan osallistuneet. Pienet julkaisut ja iteraatiot auttoivat opiskelijoita paikallistamaan ongelmat ja virheet ohjelmakoodissa. Eräs opiskelija piti myös siitä, että asiakkaalle pystyi näyttämään aina uusimpia kehitettävään sovellukseen toteutettuja ominaisuuksia. [31]

Kivi, ym. [20] havaitsivat, että inkrementaalinen julkaisujen suunnittelu ja toteutus on hyvin käyttökelpoinen tapa saada kehitettyä toimivaa sovellusta joka täyttää asiakkaiden vaatimukset lyhyessä ajassa. Ongelmana tässä tutkimuksessa mukana olleella ohjelmointitiimillä oli kokemuksen puute, mikä johti muutamiin ongelmiin

ensimmäisen inkrementin aikana. Yhtä vaatimusta ei ymmärretty täysin oikein ja myöhemmin täytyi käyttää enemmän aikaa koodin uudelleenmuokkaamiseen.

The Johns Hopkins -yliopiston [39] kurssilla julkaisujen välinä pidettiin kahta viikkoa, mutta tästä aikataulusta ei pidetty kuitenkaan aina tarkasti kiinni. Kurssin alkupuolella opiskelijat aliarvioivat ajan, jonka he tarvitsivat päästäkseen sisälle olemassa olevaan ohjelmakoodiin.

5.3 Järjestelmän metafora

Metafora-käytännön soveltamisesta ei ole monissakaan tutkimuksissa mainintaa. Shuklan ja Williamsin [31] tutkimuksessa 58 prosenttia opiskelijoista ei edes yrittänyt soveltaa tätä käytäntöä. Eräs opiskelija kommentoi, että heidän projektilleen ei ollut metaforaa, eikä hän nähnyt metaforan käyttöä hyödyllisenä. Kuitenkin 36 prosenttia heistä, jotka kokeilivat käytäntöä, pitivät siitä ja onnistuivat mielestään sen soveltamisessa.

Valparaison yliopiston [18] kurssilla metaforan käyttöä ei vaadittu, mutta sitä suositeltiin. Kullekin opiskelijaryhmälle annettiin valmiiksi lähtötarina ja ehdotus metaforaksi, eikä oman metaforan keksimistä vaadittu. Johnson ja Caristi [18] arvelivat, että juuri tämän takia projektin keskeiset asiat eivät pysyneet projektilaisilla aina selkeinä mielessä.

The Johns Hopkins -yliopiston [39] kurssilla metafora-käytäntöä ei sovellettu ollenkaan, sillä kehitettävästä sovelluksesta oli jo olemassa valmis prototyyppi, joka auttoi opiskelijoita ymmärtämään mistä projektissa oli kyse.

Johnsonin ja Caristin [18] mukaan aloittelevat ohjelmoijat tarvitsevat myös vähän enemmän kuin pelkän metaforan. Heidän tutkimuksessaan opiskelijoita pyydettiin tekemään annetuista isosta ja muutamista pienistä tarinoista käsitekaavio, jonka pohjalta laadittiin luokkakaavio. Aluksi tämä tuntui opiskelijoista ikävystyttävältä ja tarpeettomalta, mutta myöhemmin he huomasivat kaavioiden hyödyn. Kaaviot auttoivat pysymään oikealla tiellä ja ymmärtämään paremmin mitä he olivat kehittämässä.

Swinburnen teknillisen korkeakoulun [19] projektikurssin ryhmä ei myöskään pitänyt tästä käytännöstä. Projektin alussa he käyttivät metaforana 'SimCity' [10] ja 'King of Dragon Pass' [1] -simulaatiopelejä, mutta muutaman viikon päästä projektiryhmän kehittämästä 'simProject'-pelistä tuli itsessään metafora projektille. Opiskelijoiden ja asiakkaan mielestä pelin käsite oli niin yksinkertainen ja ymmärrettävä, että ulkoista metaforaa ei tarvittu.

5.4 Yksinkertainen rakenne ja suunnittelu

”Yksinkertaisen rakenteen ja suunnitteluprosessin käsite on luontaista useimmissa yliopistomaailman projekteissa.” [28]

Tämän XP:n käytännön mukaan ei pitäisi suunnitella asioita tai varautua liian pitkälle tulevaisuuteen, vaan pitäisi aina keskittyä täyttämään ainoastaan sen hetkiset ohjelmistolle asetetut vaatimukset. Useimmilla ohjelmistotekniikan kursseilla opiskelijoita kuitenkin opetetaan miettimään koodin uudelleenkäyttöä, laajennettavuutta ja jatkokehitystä, mikä tuotti Karlsruhen yliopiston kurssin ohjaajien mukaan heidän opiskelijoilleen ongelmia yksinkertaisen rakenteen käytännön omaksumisessa. Kurssin opiskelijat kutsuivat prosessia suunnitteluksi silmälappujen kanssa. [19]

Calgaryn yliopiston opiskelijoilla oli myös vaikeuksia tämän käytännön kanssa. Vain yhteen inkrementtiin keskittyminen ja tulevaisuuteen varautumisen puute aiheuttivat paljon ylimääräistä työtä toisessa inkrementissä. Arkkitehtuuria täytyi muuttaa, että inkrementin toteutus olisi mahdollinen. [20]

North Carolina State -yliopiston kurssin opiskelijat kertoivat onnistuneensa tämän käytännön soveltamisessa. Ohjaajien mukaan opiskelijoille on usein luonnollista tehdä niin yksinkertaista kuin mahdollista ja jättää dokumentointi tekemättä, kunhan vaan saavat heitä tyydyttävän arvosanan. Kurssin opettajia tämä kuitenkin huolestutti, koska he näkevät dokumentoinnissa pitkällä aikavälillä saatavaa hyötyä, jota opiskelijat eivät osaa arvostaa. [31]

Swinburnen teknillisen korkeakoulun XP-projektiryhmä [19] toteutti järjestelmän alunperin niin, että se toimi, mutta he eivät kuitenkaan pitäneet mielessä, että ohjelmakoodin rakenne olisi pidettävä mahdollisimman yksinkertaisena. Läpi projektin opiskelijat pystyivät uudelleenrakentamisen 4.2.6 avulla pitämään rakenteen niin yksinkertaisena kuin osasivat, mutta ajanpuutteen vuoksi he eivät pystyneet uudelleenrakentamaan niin paljon kuin olisivat muuten voineet.

Swinburnen teknillisen korkeakoulun [19] viimeisen vuosikurssin projektikursseilla projektin eri vaiheissa, kun edessä oli joku monimutkaisempi toteutettava osio, opiskelijoille pidettiin ennen toteutuksen aloittamista erityiset suunnitteluistunnot 5.4. Näissä ohjatuissa kokoontumisissa opiskelijat saivat tehdä kevyen etukäteissuunnitelman siitä, miten aikovat uuden osion toteuttaa. Suunnitelmat toimivat eräänlaisena karttana opiskelijoille siitä, kuinka tehtävä olisi hyvä toteuttaa ja kuinka järjestelmän eri osat liittyivät yhteen. Opiskelijat arvostivat suunnitteluistuntoja ja huomasivat niiden hyödyn, mutta eivät kuitenkaan olleet tarpeeksi kurinalaisia noudattaakseen tätä käytäntöä itsenäisesti. Itsenäisesti työskennellessään he olivat

tyytyväisiä, kun he vain saivat uuden ominaisuuden toimimaan.

Yksinkertaisen rakenteen käytäntöä painotettiin The Johns Hopkins -yliopiston [39] kurssilla paljon. Pääosin opiskelijat noudattivat tätä käytäntöä hyvin ja välttivät varautumasta liiaksi tulevaisuuteen.

5.5 Testaus

"Yleisesti ottaen testausta ei ole korostettu vahvasti opetussuunnitelmassa ja silti se on hyvin tärkeää todellisessa maailmassa. XP-lähestymistapa antaisi opiskelijoille mahdollisuuden kehittää perusteellisten testien suunnittelu- ja toteuttamistaitojaan." [28]

The Johns Hopkins -yliopiston [39] kurssilla testilähtöisen ohjelmoinnin käytäntö oli yksi vahvimmin korostetuista käytännöistä, koska kurssin ohjaajan mukaan se on hyödyllinen muutenkin kuin vain XP-ympäristössä. Vaatimuksena oli, että ennen uuden ohjelmakoodin kirjoittamista sille tehtiin testitapaukset. Lisäksi myös valmista ohjelmakoodia muokattaessa ensin oli muokattava ja lisättävä tarvittavat testitapaukset. Ennen ohjelmakoodin lähettämistä versionhallintajärjestelmään kaikkien testien oli mentävä virheittä läpi.

Testauksen käytäntöä noudatettiin The Johns Hopkins -yliopiston [39] kurssilla kiitettävästi. Kun opiskelijat oppivat käyttämään testiympäristöä, he ymmärsivät nopeasti sen arvon. Kurssin opiskelijat oppivat myös nauttimaan testilähtöisestä ohjelmoinnista yhdessä yksinkertaisen rakenteen käytännön kanssa. Kurssin alussa valmiina olleeseen ohjelmakoodiin ei oltu kirjoitettu yhtään testitapausta, mutta kurssin loppupuolella niitä oli jo yli 150. Vain pari kertaa versionhallintajärjestelmään laitettiin ohjelmakoodia, joka ei mennyt kaikista testeistä läpi.

Hyväksyntätestejä eivät The Johns Hopkins -yliopiston [39] kurssin opiskelijat tykänneet kirjoittaa yhtä paljon kuin yksikkötestejä, vaikka osaa yksikkötesteistä voitiin pitää myös hyväksyntätesteinä. Suurin vaikeus automaattisten hyväksyntätestien tekemiseen oli sellaisen työkalun puuttuminen, jolla voisi testata graafisen käyttöliittymän eri osioita.

North Carolina State -yliopiston kurssilla opiskelijoita kehotettiin kirjoittamaan testitapaukset ennen varsinaisen koodin kirjoitusta, mutta monet tunnustivat kirjoittaneensa testitapaukset vasta lopuksi, kun ohjelma oli jo tehty valmiiksi. Nämäkin testitapaukset he kirjoittivat vain täyttääkseen kurssin vaatimukset. Eräs opiskelija kuitenkin kommentoi, että heidän filosofianaan oli "testaa aikaisin ja usein". Hänen mielestään tämän käytännön etuna oli se, että he pystyivät näkemään paljonko työtä oli milloinkin jäljellä, ja että testitapauksia ei tarvinnut kirjoittaa aina

toistuvasti uudestaan. [31]

Calgaryn yliopiston opiskelijat käyttivät yksikkötestauksessaan JUnit-kehystä ja hekin kertoivat kirjoittaneensa testitapaukset vasta ohjelmakoodin kirjoittamisen jälkeen. Kun testitapauksia yritettiin jälkeenpäin kirjoittaa jo toteutettuun ohjelmiston osaan, projektin eteneminen hidastui. Koodissa ei ollut paljon virheitä, joten yksikkötestien puuttellisuus ei itsessään ollut ongelma, mutta he kuitenkin havaitsivat että olisi ollut hyödyllisempää jos projektin alusta lähtien yksikkötestit olisi kirjoitettu ennen varsinaista ohjelmakoodia. [20]

Swinburnen teknillisen korkeakoulun XP-projektiryhmä käytti projektin alkuvaiheessa, ennen JUnit-testausympäristön asentamista, luokkiensa testaamiseen erilaisia itse tehtyjä 'testipenkkejä'. Tämän ryhmän opiskelijoille oli vaikeaa tehdä testitapauksia ennen varsinaisten luokkien ohjelmakoodin kirjoittamista, joten he kirjoittivat testitapaukset vasta kun luokat oli jo osittain tai kokonaan toteutettu. Opiskelijoiden vähäinen kokemus suunnittelusta, perinteisellä- tai XP-tavalla, vaikutti tähän varmasti paljon. [19]

Toinen ongelma testauksessa Swinburnen projektiryhmällä oli JUnit-kehyksen käytön opettelemisessa. Opiskelijat eivät käyttäneet JUnit-kehystä heti projektin alusta alkaen, vaan he joutuivat opettelemaan tämän heille täysin uuden ympäristön projektin ollessa jo hyvässä vauhdissa. Heillä ei ollut tietoa, mitä kaikkea tällä testausympäristöllä voisi tehdä, joten tämän käyttöönotto tuntui hankalalta. Yksi projektiryhmän opiskelijoista totesikin jääneensä näiden vaikeuksien takia käyttämään manuaalista testausta. Kuitenkin projektin edetessä, kun opiskelijat oppivat käyttämään testausympäristöä, ja kun he käyttivät sitä säännöllisemmin, heidän asenteensa muuttui. Esimerkiksi yhdestä ryhmän opiskelijasta oli hienoa oppia tällainen uusi tekniikka ja se auttoi häntä ymmärtämään paremmin varsinaisten Java-luokkien metodien toimintaa.

Suurin osa Karlsruhen yliopiston opiskelijoista noudatti 'testitapaus ennen ohjelmakoodia' -käytäntöä. Testitapausten tekeminen antoi opiskelijoille enemmän varmuutta oman ohjelmakoodin toiminnasta ja näiden testien jatkuva suorittaminen nähtiin myös hyödyllisenä. Eräillä opiskelijoilla, jotka kehittivät ohjelman graafista esitystä, ei kuitenkaan ollut työkalua automaattiseen testaukseen, vaan he joutuivat laatimaan manuaalisia testejä, joita he sitten suorittivat käsin. [19]

North Carolina State -yliopistossa käytettiin yksikkötestauksen lisäksi myös hyväksyntätestausta. Jokaiselle ryhmälle annettiin käyttäjätarina, jota varten heidän tuli esittää mustalaatikkotestitapaukset, joiden he uskoivat osoittavan ominaisuuden oikeanlaisen toiminnallisuuden. Asiakkaan hyväksynnän kriteerien ehdottomuutta simuloitiin antamalla opiskelijoille mustalaatikkotestitapaukset, joiden tu-

los vaikutti arvosanaan. [31]

5.6 Uudelleenrakentaminen

"Jatkuva opiskelijoiden tuottaman koodin uudelleen tarkistaminen opettaisi heidät todella tuntemaan ohjelmakoodinsa. Tämä ei ainoastaan auttaisi heitä tuottamaan sitä uudestaan, vaan se auttaisi heitä myös muokkaamaan sitä, koska he tietävät kaiken mitä heidän ohjelmassaan tapahtuu." [28]

North Carolina State -yliopiston opiskelijat eivät nähneet uudelleenrakentamista kovinkaan hyödyllisenä, varmastikin siksi koska vain neljän viikon projektin aikana he eivät vielä joutuneet tilanteeseen, missä tästä olisi ollut hyötyä. Jotkut kuitenkin sovelsivat ja arvostivat tätä käytäntöä. Erään opiskelijan kokemuksen mukaan se teki koodista ymmärrettävämpää ja tarpeettomat koodit tuli poistettua. Testauksen ansiosta prosessi ei aiheuttanut ongelmia jo olemassa olevaan muuhun koodiin. [31]

Karlsruhen yliopistossa opiskelijat eivät myöskään nähneet tarpeelliseksi tehdä uudelleenrakentamista, sillä heidän mielestään ohjelmiston rakenne oli jo valmis ja tarpeeksi yksinkertainen. Tutkimuksen tekijät uskovat tähän olevan kaksi mahdollista syytä: projektin pieni koko ja se, että opiskelijat tekivät alunperin kattavan suunnitelman, eivätkä minimaalista suunnitelmaa. [19]

Calgaryn yliopistossa havaittiin, että alussa uudelleenrakentamista tarvittiin hyvin vähän. Koodi valmistui nopeasti, eikä virheitä ollut paljon. Kuitenkin myöhemmin, kun suurempi määrä koodia oli kirjoitettu, alkoi ilmetä ongelmia. Uusien ohjelman osien integroiminen tuli vaikeammaksi ja tähän kului aikaa yhä enemmän. Hankaluuksia tuotti myös asiakkaan vaatimusten lukkoon lyöminen, koska vain pieni osa systeemistä otettiin huomioon suunnitteluvaiheissa. Tämä pakotti tekemään paljon uudelleenrakentamista, sillä virheelliset toteutukset piti muuttaa. Loppupuolella suurin osa ohjelmointiajasta kului uudelleenrakentamiseen, jotta uudet ominaisuudet voitiin toteuttaa. Jotkut opiskelijat kokivat, että uudelleenrakentaminen on vaan toistuvaa työtä, eikä anna mitään lisäarvoa valmiiseen tuotteeseen. [20]

Swinburnen teknillisen korkeakoulun XP-projektiryhmä onnistui parhaiten uudelleenrakentamis-käytännön noudattamisessa. Heidän mielestään uudelleenrakentaminen oli luonnollista ja he toteuttivatkin sitä läpi koko projektin. Huolellisen uudelleenrakentamisen avulla opiskelijat onnistuivat yksinkertaistamaan ohjelmakoodin rakennetta huomattavasti. Myös opiskelijoiden ymmärrys ohjelmakoodista kasvoi uudelleenrakentamisen myötä, ja heidän ohjelmointitaitonsa kehittyivät, kun he keksivät parempia tapoja toteuttaa järjestelmän eri osioita. [19]

5.7 Pariohjelmointi

”Tulokset osoittavat että opiskelijat, jotka tekevät pariohjelmointia, suoriutuvat paremmin ohjelmointiprojekteissa ja suuremmalla todennäköisyydellä suorittavat kurssin vähintään tyydyttävällä arvosanalla. Opiskelijaparit ovat itsenäisempiä ja vähemmän riippuvaisia opettajista.” [38]

Useimmissa näistä tutkimuksista opiskelijat ovat huomanneet ratkaisevansa ongelmat nopeammin ja löytävänsä virheet aikaisemmin, kun he ohjelmoivat pareittain [31, 19]. Sandersin [28] tutkimuksessa pariohjelmoinnin positiiviseksi puoleksi opiskelijat toivat esille myös kehittyneemmät suhteet parin kanssa.

Swinburnen teknillisen korkeakoulun XP-projektiryhmä onnistui soveltamaan pariohjelmoinnin käytäntöä menestyksekkäästi. Ohjelmakoodin laatu oli parempaa, ohjelmointi oli tehokkaampaa, kommunikointi opiskelijoiden välillä lisääntyi, ongelmat kyettiin ratkaisemaan helpommin ja ohjelmointi oli miellyttävämpää. Kurssin viimeisen kuukauden aikana kurssista johtumattomien ulkoisten seikkojen takia yksi opiskelija ei voinut osallistua enää paljoakaan työskentelyyn, jolloin pariohjelmoinnista jouduttiin luopumaan. Tällöin ero pariohjelmoinnin ja yksin tehtävän ohjelmoinnin välillä tuli selkeästi näkyviin. Lähes kaikki hyvät asiat, joita pariohjelmoinnista seurasi, menivät heikommin yksin ohjelmoidessa. [19]

Pariohjelmoinnin käytäntöä korostettiin myös vahvasti The Johns Hopkins -yliopiston [39] kurssilla ja opiskelijatkin suhtautuivat tähän käytäntöön hyvin positiivisesti. Kaikki uusi ohjelmakoodi täytyi kirjoittaa pareittain, niin että molemmat tekivät yhdessä sekä testit, että tuotantoon menevän ohjelmakoodin. Ajan puutteen ja aikataulutusongelmien vuoksi jotkut opiskelijat saivat luvan kirjoittaa itsenäisesti yksikkötestejä jo ennen projektia tehtyyn ohjelmakoodiin. Myös yksinkertaisia uudelleenrakentamisia 4.2.6 annettiin tehdä ohjelmakoodiin, johon oli olemassa valmiit testitapaukset 4.2.5.

Pareja ei The Johns Hopkins -yliopiston [39] kurssilla virallisesti määritelty, vaan opiskelijoita kehoitettiin kokeilemaan työskentelyä yhdessä eri projektin jäsenten kanssa. Käytännössä monet opiskelijoista muodostivat keskenään puoliksi vakituiset parit. Myös kurssin ohjaaja pyrki työskentelemään jonkin aikaa yhdessä niin monen kurssin opiskelijan kanssa, kuin ajan ja aikataulujen puitteissa oli mahdollista.

Swinburnen projektiryhmän opiskelijat tekivät mieluiten töitä samantasoisen ja yhtä motivoituneen parin kanssa. Teknisesti heikommat opiskelijat tunsivat olonsa varmemmaksi, kun he saivat ohjelmoida pareittain, varsinkin jos pari oli samantasoinen osaamiseltaan. Opiskelijat uskoivat myös, että he kykenivät ohjelmoimaan enemmän XP-tiimissä, kuin ei-XP-tiimissä. [19]

Mikäli toinen parista on kokeneempi, toinen voi myös oppia paljon, mutta opiskelijaprojektissa tällaisia kokeneita ohjelmoijia on hankala löytää [20]. Toisaalta sekä Kalifornian yliopistossa, Santa Cruzissa [6], että Northwest Missouri State -yliopistossa, Maryvillessä [28] tehdyissä pariohjelmoinnin tutkimuksissa huomattiin, että jos toinen pareista on paljon kokeneempi kuin toinen, kokeneemmasta työskentely parin kanssa voi tuntua ajanhukalta, eikä hän jaksa aina odottaa toista ja selittää tälle omasta mielestään itsestään selviä asioita.

Hankaluuksia opiskelijoille on tuottanut aikataulujen yhteensovittaminen, mikä on joskus tehnyt pariohjelmoinnin soveltamisen mahdottomaksi [20]. Santa Cruzissa [6] tehdyssä tutkimuksessa kurssin opiskelijoita pyydettiin ilmoittamaan, mikäli pari on toistuvasti epäluotettava, tai jos pareilla on ylitsepääsemättömiä aikataulujen yhteensovittamisongelmia. Valitettavasti opiskelijat yrittivät ongelmista huolimatta jatkaa eteenpäin monta viikkoa ennen kuin toivat ne ilmi. Tällöin parien uudelleen järjestely oli luonnollisesti hankalampaa.

Yllättävää tällaisissa ongelmatapauksissa oli opiskelijoiden valmius palauttaa ohjelmointityö molempien nimillä varustettuna, vaikka toinen ei olisi osallistunut työhön lainkaan. Tutkimus paljasti että nämä opiskelijat ajattelivat olevan tärkeämpää, että he näyttävät noudattavan pariohjelmoinnin vaatimuksia, kuin että olisivat rehellisiä työn jaon suhteen. [6]

Sandersin [28] tekemässä tutkimuksessa esille tuli myös vaikeudet parien keskinäisessä kommunikaatiossa. Kurssin alussa noin puolet opiskelijoista ilmoitti näistä vaikeuksista, mutta lukukauden lopussa lähes kaikki olivat jo selvittäneet kommunikaatio-ongelmansa.

Sandersin [28] tutkimuksessa opiskelijoilta kysyttiin sekä kurssin alussa, että lopussa, onko pariohjelmointi sopiva käytäntö kyseiselle kurssille. Aluksi lähes kaikki olivat myönteisiä, erityisesti jos kurssilla on säännöllisesti järjestetyt laboratoriosessiot helpottamassa aikataulutusongelmia. Tämä alun innostus laantui kuitenkin lukukauden loppuun mennessä. Useimmat kyllä uskoivat pariohjelmoinnin olevan arvokas käytäntö, mutta se tulisi ottaa käyttöön vasta jollakin myöhemmällä kurssilla. He olivat myös hyvin vakuuttuneita siitä, että jos pariohjelmointia käytetään, parien tulisi olla mahdollisimman samantasoiset osaamiseltaan ja että kurssilla tulisi olla säännölliset laboratoriosessiot.

5.8 Yhteisomistajuus

"Työskentely versionhallintajärjestelmän kanssa yliopistossa on hyvää harjoitusta tulevaa työtä varten, sillä useimmat yritykset käyttävät sitä jollakin tavalla." [28]

Koodin yhteisomistajuus on nähty yleensä hyvin positiivisena ja työskentelyä helpottavana asiana, kun käytössä on ollut joku toimiva versionhallintajärjestelmä kuten esimerkiksi CVS. Useimpia opiskelijoita ei haitannut, että joku toinenkin saattoi muokata heidän koodiaan ja korjata mahdollisia virheitä [31, 18]. Tosin Johnsonin ja Caristin [18] tutkimuksessa joidenkin opiskelijoiden mielestä oli ikävää, jos joku muutti jotakin heidän kirjoittamaansa osaa, jonka he ajattelivat olevan kunnossa.

The Johns Hopkins -yliopiston [39] kurssilla projekti oli jaettu pienempiin osaprojekteihin, mutta yhteisomistajuuden käytäntö ulotettiin siitäkin huolimatta koskemaan kaikkea projektin ohjelmakoodia. Opiskelijat saivat ottaa toteutettavakseen uusia ominaisuuksia myös muista osaprojekteista, kuin mihin he itse varsinaisesti kuuluivat. Pariohjelmointia 4.2.7 eri osaprojektien jäsenten kesken suositeltiin, jotta tieto ja osaaminen eri projektin osista leviäisi projektin jäsenten kesken.

Calgaryn yliopiston kurssin ohjelmointitiimi ei aluksi käyttänyt varsinaista versionhallintajärjestelmää ollenkaan. He lähettivät uudet ohjelmakoodit aina sähköpostitse toisilleen ja ilmoittivat milloin alkoivat ja lopettivat tietyn osion muokkaamisen. Projektin edetessä tämä menetelmä kävi luonnollisesti hyvin hankalaksi, jolloin ryhmää kehoitettiin ottamaan käyttöön oikea versionhallintajärjestelmä. Tässä tapauksessa käyttöön otettiin JCVS ja sen tuomat hyödyt huomattiin nopeasti, eikä järjestelmän käytössä ilmennyt ongelmia juuri lainkaan. Rinnakkain tapahtuva kehitys tuli mahdolliseksi, sekä tehtyjen muutosten perustelu ja kommentointi auttoivat ymmärtämään koodin tilaa paremmin. [20]

Swinburnen teknillisen korkeakoulun XP-projektiryhmä omaksui yhteisomistajuuden käytännön hitaasti, koska he olivat tottuneet tekemään omaa työtä eikä jakaa sitä toisten kanssa. Projektissa jokaisella projektin jäsenellä oli oma näytekomponentti. Tällä oli alkuvaiheessa negatiivinen vaikutus ohjelmakoodin jakamiseen toisten ryhmän jäsenten kesken, mutta kun opiskelijoille vakuutettiin, että nämä henkilökohtaiset näytekomponentit eivät vaikuta arvosanaan laskevasti, opiskelijat alkoivat noudattaa yhteisomistajuuden käytäntöä ja jakaa ohjelmakoodia keskenään. [19]

5.9 Jatkuva integrointi

”Yliopistomaailmassa on käytännöllisesti katsoen mahdotonta saada projektiryhmiä, kuten yleensä ohjelmistotekniikassa, päivittämään ja integroimaan ohjelmistoa useita kertoja päivässä. He ovat onnekkaita jos edes löytävät kaikille sopivaa yhteistä aikaa ryhmän ohjelmointisessioon.” [28]

North Carolinan yliopistossa tehdyssä tutkimuksessa opiskelijoita ohjattiin jatkuvasti integroimaan tuottamansa ohjelmakoodi heti tekeillä olleiden osioiden valmistumisen jälkeen. Useimmille opiskelijoista tämä ei tuottanut vaikeuksia ja toi kaivattua vaihtelua. Eräs opiskelija kommentoi käytännöstä, että se auttaa projektia kasvamaan yhteen aikaisessa vaiheessa, sen sijaan että projekti olisi palasina ja täytyisi vain toivoa niiden toimivan lopussa yhdessä. [31]

Wilsonin [39] kokemuksen mukaan opiskeluympäristössä projektin jäsenet ovat innokkaampia integroimaan ja lähettämään uudet ohjelmakoodit versionhallintajärjestelmään lyhyin väliajoin, kuin teollisuudessa. Tämä johtuu Wilsonin mukaan siitä, että opiskelijat työskentelevät lyhyemmän ajan kerrallaan ja tauot kehityssessioiden välillä ovat pidemmät.

Swinburnen teknillisen korkeakoulun projektiryhmällä XP:n käytännöistä vähiten menestyksekkäimmin onnistuneita oli jatkuvan integroinnin käytännön soveltaminen. Heillä oli tämän käytännön kanssa pääasiassa kolme ongelmaa: Ant-integrointityökalu toimi kunnolla vasta projektin puolivälin jälkeen, opiskelijat eivät osanneet käyttää kunnolla CVS-versionhallintajärjestelmää, eikä yhteistyö ja koordinointi kehittäjäparien välillä ollut riittävää. [19]

5.10 40-tuntinen työviikko

”Riippumatta siitä, työskentelevätkö opiskelijat tiimeissä vai ei, on vain järkevää että heidän tulisi rajoittaa mihin tahansa projektiin käyttämänsä aika. Tietyn pisteen jälkeen, mikä tietysti vaihtelee kurssista riippuen, työskentelyyn käytetty aika on harvoin tuottavaa.” [28]

Monissa muissa tässä työssä käsitellyistä tutkimuksissa opiskelijoiden viikottainen tuntimäärä on ollut paljon pienempi, esimerkiksi 10 tuntia [39], mutta Swinburnen teknillisen korkeakoulun projektikurssilla pyrittiin noudatettiin tätä käytäntöä kirjaimellisesti ja opiskelijat pyrkivät tekemään täysiä 40 tunnin työviikkoja. Swinburnen projektikurssin aikaisemmissa perinteisiä menetelmiä käyttäneissä projekteissa teknisesti heikommat opiskelijat saattoivat käyttää paljon aikaa dokumentointiin, mutta koska XP:ssä tätä työtä ei ole paljoakaan, heillä oli vaikeuksia saada 40 viikkotuntia täyteen. Nämä opiskelijat, jotka eivät olleet niin vahvoja ohjelmoijia, tekivät töitä keskimäärin 35 tuntia viikossa. [19]

Monesti projekteilla on käynyt niin, että suuri osa työstä kasautuu projektin viimeisille viikoille, mutta Keefen ja Dickin [19] mukaan 40-tuntisen työviikon käytännön ansiosta Swinburnen XP-projektiryhmän työtunnit jakautuivat tasaisesti projektin 18 viikon ajalle.

5.11 Paikan päällä oleva asiakas

"Paikan päällä oleva asiakas on eduksi, jos on jotakin jota käyttäjä ei tarvitse, tai jos asiakkaan mieli muuttuu. Huonona puolena on, että tämä voi kannustaa asiakasta tekemään jatkuvasti muutoksia, mikä vuorostaan viivästyttää kehittäjiä." [28]

"Olisi mahtava tilanne jos opiskelijalle voitaisiin järjestää kahdenkeskinen yhteys professorin kanssa siksi ajaksi, kun opiskelija opiskelee tai ohjelmoi. Mutta oli miten tahansa, resurssit tämän toteuttamiseen ovat niukat. Ei vaan yksinkertaisesti ole tarpeeksi professoreita." [28]

Monesti kurssin ohjaajat ovat toimineet asiakkaan roolissa ja määritelleet vaatimukset projektille. Joskus jos ohjaajia on ollut vähän verrattuna projekteihin, paikanpäällä olevaa asiakasta on simuloitu olemalla tavoitettavissa esimerkiksi sähköpostitse. [31]

The Johns Hopkins -yliopiston [39] kaikki XP-kurssin 20 opiskelijaa työskentelivät samassa projektissa, jolloin asiakkaita ei tarvittu kuin yksi. Silti asiakkaan roolissa toimi kurssin ohjaaja, eikä hänkään voinut aikataulutusongelmien takia olla aina paikan päällä projektilaisten tavoitettavissa. Kurssin opiskelijat olivat melko hyvin perillä sovellukselta vaadituilta ominaisuuksilta, joten heitä kehoitettiin toimimaan itse asiakkaina ja kirjoittamaan myös, kurssin ohjaajan lisäksi, käyttäjätarinoita. Oikean paikan päällä olevan asiakkaan puuttuminen ei tuottanut paljoakaan ongelmia kurssin opiskelijoille.

Ketterissä menetelmissä, kuten myös XP:ssä, kehittäjiä kehoitetaan ongelmatilanteissa ennemmin kysymään asiakkaalta, kuin tekemään omia oletuksia vaillinaisen tiedon pohjalta [24]. Melnikin ja Maurerin [24] kurssilla opiskelijoille kerrottiin tästä säännöstä ja aluksi opiskelijat komunnikoivatkin, sekä sähköpostitse että kasvokkain, hyvin aktiivisesti asiakkaan kanssa. Myöhemmin kurssin aikana opiskelijat kuitenkin alkoivat tekemään omia oletuksiaan projektista ja vaatimuksista, usein jopa ilman muiden tiimiläisten kanssa neuvottelemista. Asiakas hylättiin vähitellen projektista ja ohjaajien täytyi muistuttaa opiskelijoita moneen kertaan, ettei tällaisia omia oletuksia pitäisi tehdä.

Keefen ja Dickin [19] mukaan Swinburnen teknillisen korkeakoulun projektissa paikan päällä oleva asiakas ei olisi ollut kovinkaan käytännöllinen ratkaisu. Asiakas oli teknillisen korkeakoulun ulkopuolelta, eikä näin ollen kyennyt olemaan fyysisesti paikan päällä jatkuvasti. Kehitystiimi ja asiakas tapasivat viikoittain ja muulloin kommunikaatio tapahtui sähköpostin välityksellä. Viikottaisissa tapaamisissa keskusteltiin projektin etenemisestä ja jatkettiin suunnittelupeliä 4.2.1.

Vaikka Swinburnen projekti ei noudattanutkaan paikalla olevan asiakkaan käy-

täntöä, silti jatkuva yhteydenpito asiakkaan kanssa osoittautui menestyksekkääksi käytännöksi. Hyödyllisimpiä projektille olivat viikottaiset tapaamiset asiakkaan kanssa, riippumatta siitä, kokivatko he tarvetta tapaamiselle tai eivät. Näissä tapaamisissa opiskelijat pystyivät tuomaan esille projektissa eteen tulleita ongelmia, ja koska asiakas tuli projektin aikana hyvin tutuksi, opiskelijat uskalsivat olla rehellisiä myöskin silloin, kun näytti että projekti on jäljessä aikataulusta. [19]

Melnikin [24] johtamassa opiskelijaprojektissa taas palkattiin ulkopuolinen asiakas. Asiakas ei ollut fyysisesti paikan päällä samassa tilassa kehittäjien kanssa, mutta tämä oli tavoitettavissa sähköpostitse ja keskustelufoorumin kautta. Vastaukset kysymyksiin asiakas antoi päivän sisällä. Opiskelijat tapasivat asiakasta projektin alkuvaiheessa, sekä kerran kuussa, jolloin he esittelivät ohjelmiston senhetkisiä toimivia osioita. Melnik ja Maurer [24] näkivät tällaisen käytännön käyttökelpoisena, sillä ainakin kerran kuussa opiskelijat saivat nähdä asiakkaan välittömän reaktion ja mielipiteen sovelluksesta.

5.12 Koodausstandardit

"Koodausstandardit olivat tärkeitä sovellusta kehitettäessä. Noudatimme nimeämiskäytäntöjä luokille, metodeille ja muuttujille. Tämä teki kaikille koodin lukemisen helpommaksi ja oli välttämätöntä, koska ei ollut mitään dokumentaatiota. Sovitut käytännöt toimivat runkona." [31]

Valparaison yliopistossa opiskelijoita pyydettiin heti kurssin alussa luomaan ja julkaisemaan koodausstandardit, joita kaikki ryhmän jäsenet noudattavat. Melkein kaikki opiskelijat kertoivat noudattaneensa näitä standardeja. Käytännön hyötynä nähtiin yleisesti, että toisten tekemää koodia oli helpompi lukea, ja että koodin yhteisomistajuus 4.2.8 toimi paremmin. [18]

The Johns Hopkins -yliopiston [39] projekti jatkoi jo aiemmin kehitettyä prototyyppiä, joten projektin jäseniä kehoitettiin noudattamaan samaa koodausstandardia, jota oli käytetty tätä ohjelmakoodia kirjoitettaessa. Tälle projektiryhmälle koodausstandardi-käytännön noudattaminen oli kaikkein ongelmallisinta XP:n käytännöistä. Vaikka yhdenmukaisten ohjelmointikäytänteiden noudattamista korostettiin opetuksessa, silti jotkut opiskelijat vaativat saada itse päättää sisennyksistä, kaarisulkeiden asettelusta, ja muista tällaisista ohjelmakoodin ulkoasullisista seikoista. Wilsonin [39] mukaan opiskelijat olisivat saattaneet noudattaa tätä käytäntöä paremmin, mikäli he olisivat itse yhdessä saaneet luoda projektille oman koodausstandardin.

North Carolina State -yliopiston kurssin opiskelijoita ei pyydetty itse luomaan

koodausstandardeja, vaan heitä neuvottiin suoraan noudattamaan Sunin Java -koodausstandardia. [31]

Swinburnen XP-projektiryhmä valitsi myös noudattaakseen Sunin Javan koodausstandardin. Käytännössä he kuitenkin noudattivat Javan koodausstandardin ja aiemmilla ohjelmointikursseilla opittujen käytäntöjen yhdistelmää. Opiskelijoille muodostui näin oma koodausstandardi, jota he kaikki noudattivat, ja joka oli heille kaikille selvä. Keefen ja Dickin [19] mukaan olisi ollut kuitenkin parempi, että projektiryhmä olisi pitäytynyt tarkemmin Sunin Javan koodausstandardissa, koska projektia oli tarkoitus jatkaa myöhemmin toisen projektiryhmän kanssa, eikä ryhmän valitsemat käytännöt ole heille välttämättä yhtä tuttuja.

5.13 Mestari – oppipoika -malli ohjelmistostudiossa

Talbotin ja Auerin [33] mukaan suuren tietomäärän antaminen ei ole niinkään oleellista ohjelmistotekniikan koulutuksessa kuin se, että opetetaan soveltamaan näitä tietoja viisaasti käytäntöön. Nykyisessä koulutusjärjestelmässä ja nykyisillä resursseilla on kuitenkin heidän mielestään hyvin haasteellista opettaa viisautta. Yksittäisten tietojen ja taitojen opettaminen on paljon helpompaa.

Nykyinen yliopistojärjestelmä, joka on käytössä ympäri maailmaa, ei ole mitenkään perinteinen opetuksessa käytetty malli. Jos tarkastelemme historiaa, huomaamme, että koulutuksessa on jo pitkään ollut käytössä malli, joka on ollut todella tehokas, mutta joka on hyvin pitkälle unohdettu nykyisessä korkeamman tason koulutuksessa. Tätä koulutuksen mallia ovat käyttäneet menestyksekkäästi maanviljelijät, kalastajat, ompelijat, kätilöt, kauppiaat, taiteilijat, ja jopa itse Jeesus omien opetuslastensa kanssa. [33, 26]

5.13.1 Mestari – oppipoika -malli

Tässä mestari – oppipoika (Apprenticeship) -mallissa henkilö, joka on mestari omalla alallaan, auttaa yhtä tai useampaa oppipoikaa nousemaan myös mestari-tasolle tällä alalla. Nämä oppilaat eivät aloita tekemällä mitään hienoja tuotoksia, vaan he tekevät opettajansa hyväksi joitakin pieniä ja arkipäiväisiä työtehtäviä tarkkaillen samalla tämän työskentelyä. Vähän kerrassaan oppilaat alkavat myös itse kokeilemaan ja tekemään yksinkertaisia alaan liittyviä työtehtäviä opettajansa tarkkailun alla. [33]

Merkittäviä etuja mestari – oppipoika -mallissa verrattuna luokkahuoneopetukseen on Talbotin ja Auerin [33] mukaan opiskelijan mahdollisuus osallistua oikei-

den ongelmien ratkaisemiseen ja mahdollisuus mestarin jatkuvaan tarkkailemiseen. Mestari tekee työtään ja antaa oppilailleen tehtäväksi sopivia työtehtäviä, jolloin nämä voivat oppia soveltamaan oppimaansa käytäntöön samalla kun auttavat mestariaan tämän työtehtävien suorittamisessa. Kun opiskelijat saavat tarkkailla mestarin työskentelyä, he voivat oppia miten tämä välttää tekemästä virheitä, sen sijaan että tekisivät ensin itse virheitä ja oppisivat niistä.

RoleModel Software -ohjelmistotalossa on vuodesta 1998 lähtien koulutettu monia erilaisia oppisopimusopiskelijoita käyttäen mestari – oppipoika -mallia. Tämä koulutusmalli toimii tämän ohjelmistotalon mukaan vaihtoehtona tavalliselle yliopistokoulutukselle, ja koska Extreme Programming -metodologian käytöllä on ollut tässä prosessissa myös huomattava rooli, on mielenkiintoista tarkastella tätä koulutusmallia tarkemmin. [26]

5.13.2 Ohjelmistostudio

RoleModel Software -ohjelmistotalon Ohjelmistostudio (Software Studio) koostuu kolmesta osatekijästä: oikeasta projektista, uudelleenjärjestetystä huoneesta ja jatkuvasta yhteistyöstä. Ohjaaja hankkii oikean asiakkaan, jolla on tarve ohjelmistoprojektille. Projekti voi olla periaatteessa mistä aiheesta tahansa, kunhan siinä tulee eteen ne asiat, joita ohjaaja tahtoo opettaa opiskelijoilleen. Projektin laajuus tulee olla mitoitettu niin, että opiskelijat ehtivät toteuttaa siihen toiminnallisuutta sinä aikana, kun ohjaaja voi olla heidän käytettävissään. Samanaikaisesti voi olla useitakin projekteja, mikäli opiskelijoita on tarpeeksi paljon ja ohjaajalla on sopiva määrä apuohjaajia. [33]

Projektin alussa ohjaaja antaa opiskelijoille lyhyen yleiskatsauksen tavoitteistaan, sekä heidän tulevasta ajastaan yhdessä. Tämän jälkeen asiakas esittelee yleisesti järjestelmää, joka opiskelijoiden olisi tarkoitus kehittää koulutuksen aikana. Yhdessä ohjaajan ja asiakkaan kanssa opiskelijat määrittelevät sitten tärkeimmät asiat, joita järjestelmään pitäisi toteuttaa. Asiakkaan tulisi olla opiskelijoiden tavoitettavissa koko projektin ajan, jolloin opiskelijat voivat saada aina tarvittaessa tältä palautetta ja selvennystä järjestelmän vaatimuksista. [33]

Kun perusasiat järjestelmän vaatimuksista on ymmärretty, opiskelijoiden täytyisi yhdessä ohjaajan kanssa jakaa vaatimukset pienempiin tehtäviin, valita niistä itselleen tehtäväksi sopiva määrä ja arvioida, kuinka kauan heillä missäkin tehtävässä kestää. [33]

Tehtävien valinnan jälkeen on aika aloittaa työskentely. Jokaisen työskentelysession alussa pidetään pieni kokoontuminen, jossa jokainen saa lyhyesti kertoa, mitä

he ovat viimeksi tehneet, mitä he aikovat tehdä seuraavaksi, sekä mitä ongelmia he ovat kohdanneet. Kun kaikki opiskelijat ovat kertoneet tilanteensa, ohjaaja voi käyttää hetken aikaa keskustelemalla heidän kanssaan näistä heidän kohtaamistaan ongelmista. Mikäli ohjaaja osaa jo ennalta odottaa opiskelijoiden kohtaavan joitakin ongelmia seuraavien tehtävien toteutuksessa, hän voi ottaa myös näitä esille tässä kokoontumisessa. Alun kokoontumisen jälkeen opiskelijat hajaantuvat tekemään töitä tehtäviensä parissa. Tämä kaikki tapahtuu tilassa, jossa tietokonepöydät ovat huoneen keskellä, niin että opiskelijat ovat kasvotusten toisiinsa päin. Tämä mahdollistaa hyvin opiskelijoiden välisen kommunikoinnin työn alla olevista tehtävistä, sekä toisten auttamisen ongelmatilanteissa. [33]

Läpi koko projektin ohjaaja voi auttaa opiskelijoita tehtävien suorittamisessa mutta myös antaa näiden tehdä joitakin virheitä. Ohjaajan tulisi analysoida, mitä virheitä oltaisiin voitu välttää kysymällä asiakkaalta parempia kysymyksiä tai pyytämällä aikaisemmin apua. Opiskelijat voivat pitää projektin aikana kirjaa, kuinka hyvin he ovat osanneet arvioida eri tehtävien vaatiman ajan, sekä pohtia mistä nämä virhearvioinnit mahdollisesti johtuivat. Ohjaajan tulisi myös sopivin väliajoin tarkkailla, miten hyvin opiskelijat osaavat jakaa vaatimukset tehtävätasolle. [33]

Opiskelijat eivät työskentele yksin omilla tietokoneillaan, vaan he työskentelevät pareittain: kaksi opiskelijaa yhdellä koneella. Parit työskentelevät työskentelysession aikana yhdessä kulloisenkin tehtävänsä parissa. Mikäli he kohtaavat työskentelyn aikana hankaluuksia, he yrittävät selvittää ongelman ensin kahdestaan, sitten toisten parien avustuksella, ja mikäli ongelma ei vieläkään ratkea, lopuksi he voivat kysyä vielä ohjaajalta neuvoa parhaan ratkaisun löytämiseen. Näin opiskelijat oppivat opastamaan toisiaan, eikä ohjaajan tarvitse käyttää niin paljon aikaa opiskelijoiden neuvomiseen. Ohjaajan tehtävänä on vaihdella pareja säännöllisesti niin, että he voivat oppia projektista mahdollisimman paljon. [33]

Aina kun pari on saanut jonkin tehtävän toteutettua valmiiksi, se tulisi integroida testitapauksineen pääjärjestelmään. Keskuskoneella ajetaan kaikki testit, mukaanlukien ohjaajan tekemät toiminnallisuustestit, sekä asiakkaan ja opiskelijoiden yhdessä tekemät hyväksyntätestit. Kun kaikki vanhat sekä uudet testit menevät läpi, voidaan tehtävä hyväksyä suoritetuksi ja siirtyä seuraavaan tehtävään. Mikäli integroinnin yhteydessä kaikki testit eivät mene läpi, täytyy selvittää, mistä se johtuu ja sitten muokata ohjelmakoodia, kunnes kaikki testit menevät läpi. Näin opiskelijat saavat kokeilla ja oppia työelämässäkin usein tarvittavaa integrointia ja siihen liittyvien ongelmien selvittelyä. [33]

Tällainen ohjelmistostudioympäristö sopii Talbotin ja Auerin [33] mukaan hyvin mestari – oppipoika -mallin käyttämiseen, sekä opittujen asioiden viisaan so-

veltamisen opettamiseen. Ohjaajan on pidettävä huoli, että kaikki opiskelijat oppivat tasapuolisesti erilaisia opittavaksi tarkoitettuja asioita. Hän voi auttaa opiskelijoita hyväksyntätestien kirjoittamisessa, jolloin voidaan yhdessä varmistua siitä, että opiskelijat ovat toteuttaneet toiminnallisuuden oikein. Joskus ohjaaja voi tuoda teollisuudesta jonkun kokeneen kehittäjän keskustelemaan parhaista ratkaisuista opiskelijoiden kanssa, tai ohjelmoimaan jonkin aikaa näiden parina. Mikäli projektin aikana ohjaaja huomaa jonkin perustavaa laatua olevan aukon opiskelijoiden ymmärryksessä, hän voi pyytää opiskelijoita keskeyttämään työnsä ja pitää näille pienen luennon aiheesta. Näin ohjaaja voi ohjata projektia ja opiskelijoiden oppimista tilanteen mukaan parhaaksi katsomallaan tavalla. [33]

5.13.3 Ongelmalähtöinen oppiminen

RoleModel Software -ohjelmistotalon käyttämä Ohjelmistostudio-malli näyttäisi pohjautuvan hyvin selkeästi ongelmalähtöisen oppimisen [14] (Problem Based Learning) mallille, sekä Extreme Programming -metodologian noudattamiseen. Ohjelmistostudion opiskelijat työskentelevät kaiken aikaa todellisten ongelmien parissa, päämääränään saada yksittäiset tehtävät, sekä lopulta koko projekti tehtyä.

Työskentelyn aikana opiskelijoille tulee eteen monenlaisia ongelmia. Osa ongelmista ratkeaa päättelemällä, mutta monet ongelmat vaativat myös uuden tiedon omaksumista. Kun opiskelijan täytyy saada ongelma ratkaistua päästäkseen työssään eteenpäin, hänellä on motivaatiota etsiä tarvittava tieto toisilta opiskelijoilta, internetistä, kirjoista, tai vaikka ohjaajalta. Kun tarvittava tieto on löytynyt, sille on jo valmiiksi olemassa konteksti, mihin se liittyy. Opiskelijan ei tarvitse kasata suurta tietomäärää etukäteen varastoon, vaan hän pääsee heti soveltamaan oppimaansa käytäntöön.

Ohjaajalla on tässä ongelmalähtöisessä oppimisessa kolme keskeistä tehtävää: opiskelijoiden oppimisen tukeminen, opiskelijoiden yhteistyön edistäminen, sekä välittäjänä toimiminen oppilaitoksen (tai vaikka ohjelmistotalon) ja opiskelijoiden välillä [14]. Ohjaaja pyrkii löytämään heikkouksia opiskelijoiden ajattelussa ja taidoissa, sekä pyrkii auttamaan heitä kehittymään näillä alueilla. Ohjaaja on oman alansa asiantuntija, joka on valmis jakamaan omaa osaamistaan ja tietämystään ohjattavilleen. Mikäli ohjaaja saa opiskelijat tekemään hyvin yhteistyötä, hänen ei tarvitse olla itse antamassa vastausta kaikkiin kysymyksiin, vaan useimpiin kysymyksiin löytyy vastaus toisilta ryhmän opiskelijoilta. Ohjaaja pitää yhteyttä myös opiskelijaryhmän ulkopuolisiin tahoihin, kuten koulutusorganisaatioon, toisiin ohjaajiin, sekä ulkopuolisiin asiantuntijoihin. Ongelmalähtoisessä oppimisessa oppilaan

vastuu oppimisesta korostuu. [14]

Tällaista ongelmalähtöistä oppimista on varmasti hyvä soveltaa myös nykyisessä yliopistomaailmassa. Esimerkiksi Jyväskylän yliopiston, tietotekniikan laitoksen Ohjelmointi 2, sekä Graafisten käyttöliitymien ohjelmointi -kursseilla opiskelijat tekevät kurssin aikana harjoitustyön, jossa tulee vastaan erilaisia ongelmia, joiden ratkaisemiseen opiskelijat saavat valmiuksia käydessään luennoilla ja tehdessään pienempiä harjoitustehtäviä. Kun opiskelijat tietävät tarvitsevansa jotain tietoa, heillä on suurempi motivaatio sen oppimiseen.

5.14 Opiskelijan näkökulma

Northwest Missouri State -yliopiston tietotekniikan laitoksella järjestetyn kahden ohjelmistotekniikan kurssin opiskelijoita pyydettiin kirjoittamaan heidän mielipiteitään Extreme Programming (XP) -ohjelmistokehitysprosessin käytöstä yliopiston tietotekniikan opiskelijoiden opetusohjelmassa. Enemmistö vastusti XP:n käyttöä ensisijaisena elinkaarimallina projektille tällä kurssilla, mutta he kannattivat kyllä joidenkin XP:n käytäntöjen esittelemistä valituilla kursseilla. Heistä tuntui että yksikkötestauksen ja koodausstandardien tulisi tulla osaksi jo ohjelmoinnin johdantokursseja, mutta muut käytännöt tulisi siirtää erityisille projektisuuntautuneille kursseilla, tai sitten niitä ei esiteltäisi ollenkaan. [28]

XP:n käyttöä kannattavilla oli neljä pääsyytä sen käyttöön elinkaarimallina ohjelmistotekniikan projektissa: (1) edistyminen on näkyvämpää, (2) kommunikointi projektin sisällä on kehittyneempää, (3) monimutkaisuus on helpompaa hallita, ja (4) testaus on kehittyneempää. Opittuja taitoja uskottiin myös tarvittavan myöhemmin työelämässä. [28]

XP:n käyttöä vastustavilla opiskelijoilla oli taas viisi pääsyytä, miksi sitä ei tulisi käyttää projektissa: (1) kurssin oppisisällön kaventuminen, (2) riittämätön valmistautuminen XP:n käyttöön, (3) aikatauluongelmat, (4) riittämättömät arviointitaidot, sekä (5) epärealistiset syklin kestoajat. [28]

5.14.1 XP:n pilottitutkimus

Northwest Missouri State -yliopiston syksyn 2001 ohjelmistotekniikan projektikurssin yksi 6:n opiskelijan ryhmä valittiin XP-pilottiprojektiksi. Projektin nimi oli Jeroo ja heidän tehtävänään oli kehittää opetus/opoppimisympäristöjossa aloittelevat opiskelijat voivat kontrolloida animoidun hahmon liikkeitä kirjoittamalla Javatyylistä koodia. Ryhmän jäsenet olivat kaikki osaamistasoltaan vahvoja ja näin ollen

hyvä valinta XP-pilottitutkimukseen. [28]

He tutustuivat XP:hen ja päättivät kuinka aikovat soveltaa tätä projektiinsa. He jakoivat ryhmänsä kahteen osaan: kahden opiskelijan graafisen käyttöliittymän ryhmään, sekä neljän opiskelijan kääntäjäryhmään. Kurssin jälkeen opiskelijoita haastateltiin ja tulokset olivat hyvin samanlaisia kuin Wilsonin [39] raportoimissa tuloksissa. [28]

Opiskelijat tunnistivat asioita, jotka menivät XP:n mukaisesti. Kokoontumiset, pariohjelmoinnin variaatiot ja CVS:n käyttö olivat näistä onnistuneimpia. Kunnioitus ja luottamus ryhmän kesken oli kasvanut ja he olivat saaneet aikaan enemmän kuin oli odotettu. Heikosti menneitä asioita olivat työaikojen aikataulutus ja kommunikaatio kahden tiimin välillä. Opiskelijat olivat sitä mieltä että XP piti työmoraalin korkealla ja voisi olla hyvä pienten sovellusten tekemiseen, mutta ei kaikkiin projekteihin tai tiimeihin. [28]

Projekti oli onnistunut ja näytti että XP:n käytöllä oli tässä vaikutusta, mutta ehkä kuitenkin tiimin jäsenten valinta vaikutti lopputulokseen enemmän. [28]

6 Suosituksia XP:n soveltamiseen opetuksessa

Kaikissa tässä tutkimuksessa käsitellyistä tapauksista joitakin XP:n käytäntöjä on muokattu ja sovellettu toimimaan paremmin opetusympäristössä. Vaikka olosuhteiden pakosta täysin puhdasta XP:tä ei ollakaan sovellettu opetuksessa käytäntöön, silti opiskelijoille on voitu antaa melko hyvä ja kattava kuva XP:stä ja sen käytöstä ohjelmistokehityksessä.

Tässä luvussa tuodaan esille suosituksia siitä, miten XP:tä olisi hyvä soveltaa ohjelmoinnin opetukseen, sekä minkälaisille kursseille se voisi soveltua.

6.1 Suunnittelupeli

Wilsonin [39] mukaan hänen pitämällään XP-kurssilla olisi ollut parempi, mikäli olisi pidetty tarkemmin kiinni julkaisuaikatauluista. Hänen mukaansa tämä olisi parempi sekä oppimisen, että XP:n oikeanlaisen noudattamisen näkökulmasta.

6.1.1 Asiakas

Suunnittelupelissä olisi hyvä olla XP:n käytännön mukaisesti asiakas, joka voi auttaa muita tiimin jäseniä ymmärtämään kunnolla projektin vaatimukset. Kehitystiimille olisi tärkeää voida saada välitöntä palautetta asiakkaalta ja näin varmistaa, että he ymmärtävät todella, mitä heidän on tehtävä. [19]

Useimmissa tässä työssä käsitellyistä tutkimuksista kurssin opetushenkilökunta on toiminut asiakkaan roolissa. Mikäli eri ryhmien suunnittelupelikokoontumiset ajoitetaan sopivasti eri aikoihin, voi opetushenkilökunta ehtiä olemaan fyysisesti paikan päällä kaikkien ryhmien suunnittelupelikokoontumisissa.

6.1.2 Työajan arviointi

XP:n suunnittelupelin käytäntöön liittyy oleellisena osana ohjelmoijan henkilökohtainen arviointi, kuinka kauan hänen valitsemiensa käyttäjätarinoiden toteuttamiseen menee aikaa. Arvioiden tulisi olla annettu realistisella tarkkuudella, esimerkiksi: ”Tämän toteuttamiseen menee noin viikko”.

Teollisuudessa kehittäjät kertovat toteutuksen arvioidun keston asiakkaalle ja

toisille kehittäjille. Opiskeluympäristössä opiskelijoiden tulisi myös raportoida arvionsa kurssin opetushenkilökunnalle. Sopiva aika arvioiden palauttamiselle on aina suunnittelupelin jälkeen, jokaisen iteraation alussa.

Iteraatioiden lopussa, kun eri käyttäjätarinat ovat valmistuneet, on opettavaista verrata miten hyvin omat arviot ovat osuneet kohdalleen. Näin opiskelijat oppivat ajattelemaan paremmin eri tehtävien vaativuuksia, pitämään kirjaa ajankäytöstään ja arvioimaan omaa työskentelyään. [24]

Jos käytetyt tekniikat ja ohjelmointikieli on opiskelijoille tuttua jo entuudestaan, kannattaa opiskelijoita muistuttaa, että projektin alkuvaiheessa eri tehtävien vaatima työmäärä saatetaan helposti yliarvioida. Kun projekti etenee, yleensä myös tehtävien monimutkaisuus kasvaa, jolloin opiskelijoita olisi taas hyvä kehottaa varaamaan tehtäville vähän enemmän aikaa. 5.1

Työajan arvioiminen on luonnollisesti hankalaa, jos ohjelmointikokemusta ei ole vielä kertynyt kovinkaan paljon. Vaikka arviot menisivätkin pieleen, ei se tarkoita epäonnistumista, vaan näin opiskelijat pääsevät vertailemaan miten heidän arvionsa ovat pitäneet paikkansa ja sitä kautta kehittymään paremmiksi arvioimaan eri tehtävien vaatimia työmääriä.

6.1.3 Tulevaisuuteen varautuminen

Monesti opiskelijat ovat saaneet enemmän opetusta perinteisistä kuin ketteristä ohjelmistonkehitys prosessimalleista. Perinteisesti projektin alkuvaiheessa kerätään kaikki projektin vaatimukset, suunnitteluvaiheessa otetaan huomioon projektin kokonaisuus, ja toteutusvaiheessa projektista on jo olemassa hyvä kokonaiskuva. Tämän takia opiskelijoille voi olla vaikeaa sopeutua siihen, että suunnittelupelissä ei oteta paljoakaan huomioon mahdollisia myöhemmin tulevia vaatimuksia, vaan keskitytään vain senhetkisiin vaatimuksiin.

'Et tule tarvitsemaan sitä' -säännöstä kannattaa muistuttaa usein. XP:ssä oletetaan, että asiakkaan vaatimukset kehitettävälle sovellukselle muuttuvat, joten kannattaa keskittyä vain siihen, mikä kulloinkin on tärkeää. Jos käytetään aikaa ominaisuuksien suunnitteluun, joita saatetaan tarvita tulevaisuudessa, hukataan aikaa niiden ominaisuuksien suunnittelulta, joita tarvitaan nyt.

6.1.4 Käyttäjätarinoiden hajoittaminen tehtävätasolle

Normaalissa ohjelmistoprojektissa kehitettävä järjestelmä on laajempi ja käyttäjätarinat ovat yleensä työläämpiä, mutta opiskelijaprojektin on oltava tarpeeksi pieni, että se saadaan valmiiksi kurssin aikana [19]. Tämän takia käyttäjätarinoista tulee

helposti jo itsessään melko pieniä, eikä niitä ole enää järkevää jakaa erillisiksi tehtäviksi.

6.2 Pienet julkaisut

Opiskelijoita voidaan pyytää jakamaan julkaisut pienempiin iteraatioihin 5.2. Tällöin opiskelijoilla on aina sopivan pieniä osia toteutettavana ja projektin edistyminen on entistä näkyvämpää.

Ketterän ohjelmistonkehitys -manifestin [5] laatijoiden mukaan ajettava ohjelmakoodi on projektin tärkein tuotos ja toimiva ohjelmisto on projektin edistymisen tärkein mittari. Schneiderin ja Johnstonin [30] mielestä opiskeluympäristössä tämä ei kuitenkaan pidä täysin paikkaansa. Projektissa työskentelevät opiskelijat oppivat tekemällä virheitä, analysoimalla niistä aiheutuvia ongelmia, sekä korjaamalla niitä. Tämä kaikki on tärkeä osa oppimisprosessia ja vie paljon aikaa, eikä tänä aikana edistymistä voi mitata uuden toimivan ohjelmakoodin tuottamisen avulla. Schneiderin ja Johnstonin [30] mukaansa myös nämä projektin aikana opitut asiat tulisi nähdä omana tuotoksenaan.

6.3 Järjestelmän metafora

Williamsin ja Upchurchin [37] mielestä metaforan idea on hyvin abstrakti ja usein hämärä kokeneillekin ammattilaisille. He suosittelevat, että tätä XP:n osa-aluetta ei painotettaisi varsinkaan yliopisto-opiskelijoiden opetuksessa.

Schneider ja Johnston [30] taas huomasivat, että metafora-käytäntö ymmärretään ja otetaan paremmin vastaan, kun se esitellään XP-kontekstissa. ’eXtreme metafora’ -nimi tekee aikaisemmin ikävystyttävänä pidetystä käytännöstä trendikkään, jolloin opiskelijat omaksuvat sen myös helpommin käyttöön.

Mikäli metaforan keksiminen on vaikeaa projektiryhmän opiskelijoille, voi kurssin ohjaaja myös ehdottaa projektille sopivaa metaforaa 5.3. Metaforan tulisi olla sellainen, että sekä projektiryhmän opiskelijat, että mahdollinen asiakas 4.2.11 kykenevät puhumaan keskenään samaa kieltä ja ymmärtämään ohjelmiston toimintaa, sekä sen eri osien välisiä suhteita.

Johnsonin ja Caristin [18] kokemuksen mukaan olisi kuitenkin parempi, että opiskelijoita vaadittaisiin itse keksimään yksinkertainen metafora projektilleen ja sitten pitämään se mielessä. Näin visio projektin keskeisistä asioista pysyisi koko ajan selkeänä.

6.4 Yksinkertainen rakenne ja suunnittelu

XP korostaa ohjelmakoodin mahdollisimman yksinkertaisena pitämisen tärkeyttä. Tarkoituksena on, että toinen ohjelmoija voi mahdollisimman helposti ymmärtää, miten ohjelmakoodi toimii. Jos ohjelmakoodin rakenne on yksinkertainen ja siinä on vältetty samanlaisten osien toistoa, ohjelmistoa on myös helpompi kehittää eteenpäin.

6.4.1 Ymmärrettävää ja laadukasta ohjelmakoodia

Jotkut opiskelijat pitävät ylpeytenään kirjoittaa ohjelmakoodia, jota ainoastaan he itse voivat ymmärtää. Tällaiset nerokkaalta tuntuvat ratkaisut eivät vaan valitettavasti lisää ohjelmiston ylläpidettävyyttä. Mikäli myöhemmin toinen kehittäjä, tai kehittäjä itse, joutuu muokkaamaan tällaista hankalasti ymmärrettävää osiota ohjelmakoodista, voi kestää kauan ennenkuin tämän ohjelmakoodin toiminta selviää ja sitä kykenee muokkaamaan. Myös ohjelmiston kehitysvaiheessa tällaiset monimutkaisuudet ohjelmakoodin rakenteessa hidastavat työtä. Tällaisia opiskelijoita olisi selkeästi opetettava pois vääränlaisesta 'hakkeroinnista' ja ohjata heitä ymmärtämään yksinkertainen rakenne -käytännön tärkeys. Olisi painotettava, että ohjelmakoodia ei kirjoiteta vain tietokonetta ja kääntäjää varten, vaan myös toisia kehittäjiä varten.

Toiset opiskelijat eivät taas malta käyttää aikaa rakenteen suunnitteluun ja yksinkertaistamiseen, vaan ovat tyytyväisiä, kun saavat ohjelmakoodin vain jollakin tavalla toimimaan. On totta, että testilähtöisen 4.2.5 ohjelmoinnin ideana on tehdä aina yksinkertaisin mahdollinen toteutus, joka läpäisee testit, mutta silti täytyy noudattaa koodausstandardeja 4.2.12 ja hyviä ohjelmistotekniikan periaatteita. Ohjelmakoodin tulee olla laadukasta, mutta yksinkertaista ja selkeää. Mikäli ohjelmakoodin rakenne kyetään pitämään yksinkertaisena ja ymmärrettävänä, sekä noudatetaan selkeitä nimeämiskäytäntöjä, varsinaista ohjelmakoodin kommentointia ei tarvita enää niin paljon.

6.4.2 Kevyet etukäteissuunnitelmat

Swinburnen projektiryhmän opiskelijat [19] olivat sitä mieltä, että heille olisi ollut hyödyllistä suunnitella enemmän etukäteen uusia toteutettavia osioita. Heidän mielestään tämä olisi auttanut heitä pitämään järjestelmän rakenteen yksinkertaisempana ja soveltamaan testauksen 4.2.5 sekä jatkuvan integroinnin 4.2.9 käytäntöjä paremmin.

On varmasti suositeltavaa pitää projektiryhmien opiskelijoille sopivin väliajoin pieniä suunnitteluistuntoja [19], joissa tehdään kevyt ja yksinkertainen suunnitelma uuden tehtävän toteutuksesta. Itsenäisesti työskennellessään opiskelijoilla ei välttämättä ole tarpeeksi itsekuria käytännön noudattamiseen, mutta ehkä heidän huomatessaan näiden yksinkertaistenkin suunnitelmien hyödyllisyyden, he alkavat myös itsenäisesti käyttämään enemmän aikaa suunnitteluun.

Ennen monimutkaisten osioiden toteutusta tehtävät suunnitelmat voivat olla hyvin yksinkertaisia, eikä niitä tarvitse välttämättä erikseen dokumentoida. Näiden kevyiden etukäteissuunnitelmien tarkoituksena on vain auttaa toteuttamaan uusi osio vähemmillä ongelmilla ja selkeämmällä rakenteella. Itse ohjelmakoodiin on luonnollisesti hyvä kommentoida ja selittää osion toiminta tarkemmin, mutta muuten nämä suunnitelmat voivat olla vaikka kynällä paperille piirrettyjä kaavioita ja ideoita.

6.5 Testaus

Kun kehitetään sovellusta kirjoittamalla ensin testi ja sitten yksinkertaisin mahdollinen toteutus, joka läpäisee testin, järjestelmästä on koko ajan olemassa kattavat testit, joilla voidaan varmistaa sen virheetön toiminta. Testilähtöinen ohjelmointi auttaa opiskelijaa hahmottamaan kuinka työ edistyy, sekä antaa varmuutta siitä että järjestelmä todella toimii.

Testilähtöisen ohjelmoinnin käytäntöä ei olla yleensä noudatettu 5.5, vaan testit on tehty vasta lopuksi, että kurssin vaatimukset on tullut muodollisesti täytettyä. Näin opiskelijat eivät opi ollenkaan tämän käytännön hyötyjä, vaan muistavat käytännön myöhemmin vain välttämättömänä pahana, joka piti suorittaa päästäkseen kurssin läpi.

6.5.1 Motivointi

Monesti kurssin kesto on niin lyhyt, että siinä ajassa ei vielä kaikki testilähtöisen ohjelmoinnin hyödyt tule riittävän hyvin esille. Varsinkin ensimmäisillä ohjelmointikursseilla opiskelijoiden harjoitustyöt ovat melko yksinkertaisia, jolloin testien tekeminen voi tuntua heistä ajanhukalta. Pidempiaikaisissa projekteissa ohjelmakoodin rakennetta joudutaan usein muokkaamaan, eikä aina voida olla varmoja, minne kaikkialle muutos heijastuu. Jos on olemassa valmiit testit, uudelleenrakentamisen jälkeen voidaan helposti tarkistaa, että kaikki toimii edelleen niin kuin pitääkin. Opiskelijat tuottavat kurssilla yleensä uutta ohjelmakoodia, eikä koodin uudelleen-

rakentamista tarvita välttämättä vielä kovinkaan paljon.

Lyhyelläkin kurssilla opiskelijat voivat nähdä tämän käytännön etuina sen, että tehdessään testin ensin, he tulevat suunnitelleeksi luokan tai metodin paremmin. On hyvä miettiä luokan rakennetta sen käyttäjän kannalta, sillä silloin siitä tulee suunniteltua käytettävämpi. Monet ongelmat ja tarvittavat ominaisuudet tulevat näin paremmin ilmi. On myös motivoivaa, jos voi tarvittaessa milloin tahansa esitellä muillekin, kuinka kaikki testit menevät läpi.

Melnikin ja Maurerin [24] ratkaisu tämän käytännön noudattamisen motivoinnin ongelmaan on aikaisemmat takarajat testien palauttamiselle, sekä niiden huomattava vaikutus arvosanaan. Myös ehdoton vaatimus ohjelmakoodin kääntymisestä ja 'siisti koodi joka toimii' -politiikka on ollut kannustimena hyvien testien tekemiseen.

Tärkeää olisi kyetä luomaan joku positiivinen kannustin tämän käytännön noudattamiseen. Ulkoinen motivaatio saa ehkä noudattamaan käytäntöä kurssin aikana, mutta tuskin enää sen jälkeen. Olisi tarkoituksenmukaista, että opiskelijat ymmärtäisivät mitä etuja testilähtöisen ohjelmoinnin käytännön noudattamisessa on. Silloin he noudattaisivat käytäntöä mielellään myös tulevassa työelämässä.

6.5.2 Tarve opetussuunnitelman muutokseen

Keefe ja Dick [19] huomasivat tutkimuksessaan, että ennen projektin alkua kehitysympäristö ja tarvittavat työkalut, kuten esimerkiksi JUnit-testausympäristö, olisi oltava asennettuna ja toimintakunnossa. Heidän kokemuksen mukaan olisi myös hyvä, että ennen projektia opiskelijoille olisi jo opetettu JUnit-kehyksen käyttö, jolloin he olisivat voineet käyttää sitä heti projektin alusta alkaen, eikä sen opettelemiseen olisi mennyt niin paljon aikaa.

Melnikin ja Maurerin [24] mukaan ohjelmiston testaustekniikat olisi opetettava opiskelijoille jo aikaisessa vaiheessa. Southern Alberta Institute of Technology:n tietotekniikan laitoksella ensimmäiselle vuodelle siirrettiin kurssi, joka käsitteli yksikkötestausta, testilähtöistä ohjelmointia, jatkuvaa integrointia ja uudelleenrakentamista. Tämän ansiosta opiskelijoilla oli enemmän työkaluja myöhemmin tuleville ohjelmointikursseille. [24]

Tämä ohjelmistokehitystekniikoiden opetus ennen ohjelmoinnin opetusta voidaan myös kyseenalaistaa, sillä jos ohjelmoinnin hyväksi havaittuja käytänteitä opetetaan ennenkuin opiskelijoilla on paljoakaan kokemusta ohjelmoinnista, he eivät välttämättä ymmärrä käytänteiden hyötyjä eivätkä näin ollen ole motivoituneita oppimaan. Opiskelijoilla ei ole vielä mitään tarvetta oppia testilähtöistä ohjelmointia

tai muita hyviä käytänteitä, jos he eivät ole ymmärtäneet miksi he tarvitsevat tällaisia käytänteitä. Sopiva aika voisi olla mahdollisesti ensimmäisten ohjelmointikurssien jälkeen, ennen varsinaisia projektikursseja.

6.6 Uudelleenrakentaminen

Uudelleenrakentaminen on tärkeä taito opetettavaksi opiskelijoille, sillä aloittelevat ohjelmoijat kirjoittavat helposti ohjelmakoodia, jonka rakenteessa on paljon hiomista. Kurssin ohjaajan kannattaa käydä säännöllisin väliajoin opiskelijoiden kirjoittamaa ohjelmakoodia läpi, merkitä uudelleenrakentamista kaipaavat kohdat ylös ja antaa opiskelijoiden toteuttaa niitä paremmalla tavalla. Ennen uudelleenrakentamisen opettamista opiskelijoilla tulisi olla kuitenkin jo jonkin verran ohjelmointikokemusta, että he osaavat erottaa hyvän ja huonon ohjelmakoodin toisistaan. [37]

Kattavat testitapaukset 4.2.5 auttavat varmistamaan, että ohjelmakoodi toimii vielä uudelleenrakentamisen jälkeenkin halutulla tavalla. Tämän takia on tärkeää, että opiskelijat ovat tehneet testitapaukset kunnolla, ja että ne ovat ajan tasalla.

Hyvä kehitysympäristö on suurena apuna ohjelmakoodin uudelleenrakentamisessa. Monissa hyvissä kehitysympäristöissä löytyy valmiina toiminnot, joiden avulla voi esimerkiksi nimetä helposti uudelleen luokkia, attribuutteja, metodeja ja muuttujia, tai jakaa pitkiä metodeja pienempiin osiin tekemällä ohjelmakoodin osista omia metodeja. Erilaisia automatisoituja uudelleenrakentamistyökaluja on olemassa paljon, ja niitä saa yleensä asennettua myös lisäosina käytettävään kehitysympäristöön.

Kivin, ym. [20] mukaan projektin alkuvaiheissa pitäisi käyttää jonkin verran aikaa koko järjestelmän suunnitteluun, jotta uudelleenrakentaminen voitaisiin pitää minimissä. Uudelleenrakentamisessa ei itsessään tarvitsisi nähdä mitään pahaa, mutta XP:ssäkin voidaan vaikka pienen koeratkaisun (Spike Solution) avulla kokeilla, mikä olisi paras ratkaisu vaikeaan tekniseen tai suunnitteluun liittyvään ongelmaan. Mikäli tulee eteen jokin suurempi ongelma, joka uhkaa järjestelmän kehitystä, voi yksi pari käyttää esimerkiksi viikon verran tutkiakseen ongelmaa ja näin pienentää mahdollista riskiä. [36]

6.7 Pariohjelmointi

Ohjelmistotekniikan ja ohjelmoinnin opetuksessa olisi hyvä kouluttaa opiskelijoita tekemään töitä ja tulemaan toimeen toisten ihmisten kanssa. Työelämässä työs-

kentely tapahtuu usein tiimeissä ja kommunikaatio- ja ryhmätyötaidot ovat hyvin tärkeitä.

Yleisimmät pariohjelmoinnin käytännön soveltamisen ongelmat opetusympäristössä johtuvat joko opiskelijoiden vääränlaisesta asenteesta, tai parin aikataulujen yhteensovittamisongelmista [6]. Asenneongelmat ovat yleisiä sekä teollisuudessa, että yliopistomaailmassa, kun taas aikataulutusongelmat tulevat enemmän esille yliopistomaailmassa. Yliopistossa opiskelijat voivat valita haluamansa määrän erilaisia kursseja suoritettavaksi samaan aikaan, jolloin yhteistä aikaa pariohjelmoinnille on vaikea löytää. Myös erilaiset harrastukset ja muut menot vaikeuttavat aikataulujen yhteensovittamista.

Pareilla voi olla myös hyvin erilainen tyyli kirjoittaa ohjelmakoodia. Tämä saattaa aiheuttaa turhia konflikteja parin kesken, joten olisi hyvä että ohjaaja antaa heille koodausstandardin 4.2.12, jota molemmat noudattavat. [6]

6.7.1 Parien ja ryhmien valinta

Parien tulisi olla mahdollisimman samantasoisia ohjelmointitaidoiltaan [6]. Tällöin työskentely on hedelmällisintä ja molemmat voivat osallistua toteutukseen tasapuolisesti. Mikäli toinen parista on kokeneempi kuin toinen, hän tekee mielellään kaiken itse, eikä toinen parista opi paljoakaan. Monesti käy vielä niin, että vaikka toinen parista olisi tehnyt työn lähes kokonaan yksin, hän silti laittaa työn tekijäksi myös parinsa.

Parin soveltuvuutta voidaan aluksi kokeilla esimerkiksi niin, että parit saavat yrittää yhdessä koota palapelin tai ratkaista jonkin muun pienen ongelmatehtävän, jossa tarvitaan molempien panosta [6]. Aluksi pareja on vielä helppo vaihtaa, mutta jos ongelmat tulevat esille vasta myöhemmin, vaihtoja on luonnollisesti hankalampi tehdä.

Mikäli ohjelmointia tehdään ryhmässä, jossa on enemmän kuin kaksi opiskelijaa, on myös hyväksi vaihdella pareja [33]. Näin osaaminen ryhmän sisällä leviää nopeammin. Tällainen parien vaihtaminen on myös hyvää valmennusta tulevaa työelämää varten, sillä sielläkin tulee usein tilanteita, jolloin täytyy tehdä yhteistyötä erilaisten ihmisten kanssa, joita ei itse ole voinut valita.

6.7.2 Yhteisen ajan löytäminen

Aikataulujen yhteensovittamisongelmaan yhtenä ratkaisuna voi olla yhteiset pakolliset laboratoriosessiot, jolloin opiskelijat voivat tulla tekemään harjoitustyötään heille varattuun tietokoneluokkaan [6]. Kun kurssin ohjelmaan on jo valmiiksi va-

rattu tällainen pakollinen aika yhteiselle ohjelmoinnille, ei opiskelijat niin helposti järjestä sille ajalle muita menoja. Parit ja ryhmät tulisi silti muodostaa niin, että heille löytyisi muutenkin mahdollisimman paljon yhteistä aikaa, sillä näissä yhteisissä laboratoriosessioissa ei yleensä ehditä tekemään kuin pieni osa harjoitustyöstä.

6.7.3 Kulttuurin ja toimintatapojen muutos

Monilla opiskelijoilla ja opettajillakin on vielä käsitys, että harjoitustyön tekeminen yhteistyössä jonkun toisen kanssa on huijausta. Tämän käsityksen muokkaamiseksi olisi hyvä totuttaa opiskelijat ohjelmoimaan pareittain jo ensimmäisillä ohjelmointikursseilla. [6]

Olisi tärkeää saada luotua kurssille pari-orientoitunut kulttuuri. Kurssin rakenne ja toimintatavat tulisi olla järjestetty niin, että pariohjelmoinnin soveltamisen mukanaan tuomat haasteet eivät lisää ohjaajien ja opettajien valvonnan tarvetta. Kurssin arvostelussa ja harjoitustyön palauttamisen takarajoissa olisi kyettävä joustamaan tarvittaessa, mikäli parilla on ollut aikataulujen yhteensovittamisongelmia, eikä harjoitustyö ole tämän takia edennyt tarpeeksi nopeasti. Kurssin käytäntöjen tulisi minimoida mahdollisuus, että toinen opiskelijoista tekisi kaiken työn ja palauttaisi harjoitustyön molempien nimillä varustettuina. [6]

Jos opiskelijoita pakotetaan olemaan riippuvaisia epäluotettavasta paristaan noudattaakseen pariohjelmoinnin käytäntöä ja kurssin vaatimuksia, tulee helposti konflikti rehellisyyden ja akateemisten vaatimuksen välillä. Jotkut opiskelijat ovat oppineet pitämään akateemisten vaatimusten noudattamista tärkeämpänä kuin muita arvoja, joten olisi tärkeää keskustella kurssilla opiskelijoiden kanssa pariohjelmoinnin ideaalien ja käytännön eroista. [6]

6.7.4 Kurssin arvostelu

Jotkut opettajat näkevät yhteistyön tekemisen ongelmallisena kurssin arvostelun kannalta ja tämän takia suosivat lähestulkoon kokonaan yksin tehtäviä harjoitustöitä. Pareittain tai ryhmässä tehtävän työn arvostelu tulisi perustua muuhunkin kuin pelkkään lopputulokseen. Jos arvioidaan pelkkää lopputulosta, monet opiskelijat pääsevät kurssista läpi vaikka eivät olisi tehneet harjoitustyön eteen mitään.

Ideaalista olisi, että parin yhteistyön onnistuminen vaikuttaisi myös arvosanaan, mutta valitettavasti on hyvin aikaa vievää selvittää, missä pareilla on kehittämisen varaa ja missä he ovat onnistuneet [6].

Opiskelijoita voisi pyytää arvioimaan itse omaa suoriutumistaan, sekä parityön onnistumista. He voisivat arvioida myös parinsa sekä muut ryhmän jäsenet. Näis-

sä arvioinneissa opiskelijat voisivat perustella ja arvioida, mikä heidän mielestään olisi heille oikeudenmukainen arvosana kurssista. Näiden opiskelijoiden omien arvioiden pohjalta on helpompi antaa heille arvosanat kurssista.

Jyväskylän yliopistossa lehtori Vesa Lappalaisen Ohjelmointi 2 -kurssilla [22] opiskelijat saavat tehdä kurssin harjoitustyön, sekä viikottaiset pienemmät demotehtävät ryhmässä, pareittain tai yksin. Harjoitustyö koostuu muutamasta vaiheesta, joista jokainen pitää käydä näyttämässä ja hyväksyttämässä ohjaajalla ennen määräaikojen umpeutumista. Opiskelijoiden tulee osata selittää esittelemänsä ohjelmakoodin toiminta ohjaajalle, jolloin ohjaaja voi nähdä, ymmärtävätkö kaikki ryhmän jäsenet mitä mikäkin ohjelmakoodin osanen tekee. Tällä tavalla voidaan puuttua tilanteisiin, joissa näyttää siltä että kaikki eivät ole oikeasti osallistuneet harjoitustyön tekemiseen. Lisäksi opiskelijoiden ratkaisut viikottaisiin demotehtäviin antavat kuvaa opiskelijoiden osaamisen tasosta, kuten myös välikoe, jossa jokainen joutuu itsenäisesti ratkomaan joitakin ohjelmointitehtäviä ja ongelmia.

6.8 Yhteisomistajuus

Kun samaa ohjelmistoa on kehittämässä useampi henkilö, on järkevää, että kaikki pääsevät muokkaamaan mitä tahansa ohjelmiston osaa. On myös tärkeää, että ohjelmakoodista ei näe kuka kyseisin osion on kirjoittanut, vaan yhteisen koodausstandardin 4.2.12 noudattamisen ansiosta kaikki ohjelmakoodi tuntuu tutulta.

6.8.1 Arvosteluperusteiden vaikutus

Opiskeluympäristössä rasitteena voi olla perinteinen ajatus siitä, että työ on tehtävä itsenäisesti ja yhteistyö toisten opiskelijoiden kanssa on huijaamista. Opiskelijat saattavat pelätä arvosanojensa laskevan, mikäli joku toinen, jonka ohjelmointitaidot eivät ole niin hyvät, käy muuttamassa pääosin heidän tekemäänsä ohjelmakoodia. Tämän takia arvosteluun vaikuttavat asiat on mietittävä huolella, ettei arvosteluperusteet vie motivaatiota yhteisomistajuuden käytännön noudattamiseen. 5.8

Jyväskylän yliopiston tietotekniikan laitoksen sovellusprojekteissa [29] kaikille ryhmän jäsenille annetaan yleensä samat arvosanat. Tämä motivoi ryhmän jäseniä yrittämään parhaansa, sillä olisi ikävää olla vaikuttamassa omalla toiminnallaan toisten arvosanoihin alentavasti. Koko ryhmän etu on, että kaikki yrittävät parhaansa, auttavat toisiaan ja tekevät yhteistyötä. Huonona puolena tässä arvostelussa taas on se, että joku voi saada todella hyvän arvosanan, vaikka ei olisikaan ollut aktiivinen ja tuottava jäsen projektissa.

6.8.2 Versionhallintajärjestelmä

Versionhallintajärjestelmä on tärkeä osa yhteisomistajuuden käytännön onnistuneessa noudattamisessa. Rinnakkain tapahtuva kehitys vaatii monesti, että samaa tiedostoa voi muokata useampi henkilö yhtä aikaa. Jokaisella ryhmän jäsenellä olisi myös tärkeää olla aina uusimmat versiot ohjelmakoodeista. Hyviä versionhallintajärjestelmiä on saatavilla ilmaiseksi ja työelämään siirtyessään opiskelijat joutuvat kuitenkin opettelemaan jonkin versionhallintajärjestelmän käytön, joten miksi tätä ei opetettaisi jo opiskeluaikana.

Calgaryn yliopiston kurssin projektissa [20] jouduttiin ongelmiin, kun yritettiin pärjätä lähettelemällä uusimpia ohjelmakoodin versioita sähköpostilla toisille ryhmän jäsenille ja sopimalla kuka milloinkin on muokkaamassa mitäkin ohjelmakoodin osaa. Kun tämä järjestelmä alkoi muodostua liian ongelmalliseksi, siirryttiin käyttämään JCVS-versionhallintajärjestelmää ja hyödyt huomattiin nopeasti.

Vaikeutena versionhallintajärjestelmän käyttöönotossa on lähinnä se, että tätä varten täytyy asentaa oma palvelin, jossa versionhallintajärjestelmä toimii ja johon eri projektien ohjelmakoodit varastoidaan.

6.9 Jatkuva integrointi

Williamsin ja Upchurchin [37] mukaan säännöllinen integrointi on opiskelijoille hyvää harjoitusta, sillä he yleensä aliarvioivat siihen liittyvät vaikeudet.

Versionhallintajärjestelmä on suurena apuna myös jatkuvan integroinnin käytännön menestyksekkäässä noudattamisessa. Toteutettuaan jonkun osion kehittäjät lähettävät uudet ohjelmakoodin muutokset versionhallintajärjestelmän koodivarastoon, josta toiset kehittäjät taas voivat käydä päivittämässä ne itselleen. Ennen muutosten lähettämistä versionhallintajärjestelmään käydään päivittämässä versionhallintajärjestelmästä uudet ohjelmakoodin muutokset, jolloin voidaan vielä tarkistaa, että uudet muutokset toimivat yhteen toisten kehittäjien tekemien muutosten kanssa [24]. Näin ohjelmakoodi tulee integroitua usein, jopa muutaman kerran päivässä.

6.10 40-tuntinen työviikko

Yliopiston kurssilla opiskelijoilla on harvoin aikaa tehdä 40 tuntia töitä viikossa, kun muitakin kursseja on menossa samaan aikaan. Käytännön tarkoituksena on estää väsyminen ja pitää työntekijät virkeinä, joten opiskeluympäristössä viikottainen työtuntimäärä pitäisi olla kurssista ja tilanteesta riippuen yleensä alhaisempi.

Williams ja Upchurch [37] tuovat tutkimuksessaan esille tämän käytännön opettamisen tarpeen. Heidän mukaansa nykyisin arvostetaan liikaa ylitöitä ja myöhään yöllä työskentelyä. Akateemisessa maailmassa on toisinaan jostain syystä tuettu näkemystä ohjelmoijasta yksineläjänä, joka koodaa yöllä saadakseen tehtävänsä valmiiksi ennen kriittistä takarajaa. Tähän ajattelutapaan täytyisi saada muutos opettamalla opiskelijoita hallitsemaan ja suunnittelemaan paremmin ajankäyttöään.

Opiskelijoita kannattaisi pyytää pitämään kirjaa tekemistään työtunneista ja raportoimaan niistä sopivin väliajoin. Tämä auttaisi opiskelijoita ja kurssin opetushenkilökuntaa saamaan totuudenmukaisen kuvan projektiin käytetyistä työtunneista.

6.11 Paikan päällä oleva asiakas

Paikan päällä olevan asiakkaan käytäntö kehittää opiskelijoiden kommunikaatiotaitoja, sekä auttaa opiskelijoita saamaan selkeämmän kuvan kehitettävälle ohjelmistolle asetetuista vaatimuksista. [37]

Ideaalitilanne olisi, mikäli opiskelijoiden projektiryhmiin saataisiin mukaan fyysisesti samassa tilassa opiskelijoiden kanssa oleva asiakas, jolta opiskelijat voisivat aina tarpeen tullen kysyä neuvoa ja mielipidettä sovelluksen kehittämisessä eteen tuleviin kysymyksiin. Esimerkiksi joillakin pidemmillä, opintojen loppuvaiheeseen sijoittuvilla projektikursseilla, voisi olla ihan realistista saada tilaajalta asiakas olemaan mukana projektissa paikan päällä. Monesti projekteissa tehdään todelliseen tarpeeseen tulevia ohjelmistoja, jolloin projektin tilaajallakin voisi olla motivaatiota irrottaa projektiin mukaan yksi työntekijä. Tämäkin työntekijä voisi tehdä omia töitään samalla kun on paikan päällä ja valmiina osallistumaan projektin työskentelyyn.

Mikäli paikan päällä olevaa asiakasta on hankalaa järjestää, käytäntöä on hieman sovellettava. Useimmissa tässä tutkielmassa käsitellyistä tutkimuksista opetushenkilökunta on toiminut asiakkaan roolissa ja ollut tavoitettavissa sähköpostitse, puhelimitse, tai verkossa olevan keskusteluryhmän kautta. Tällaisetkin kommunikointitavat voivat toimia ihan hyvin 5.11, vaikka kynnys asioiden ja mielipiteiden kysymiseen paikan päällä olevalta asiakkaalta olisikin paljon pienempi.

Maurerin [24] kokemus oli, että hänellä ei yksinkertaisesti ollut tarpeeksi aikaa työskennellä opiskelijoiden kanssa oppituntien ulkopuolella. Tämän takia loppuesittelyssä tuli ilmi, että ryhmät olivat toteuttaneet joitakin ominaisuuksia virheellisesti. Melnikin ja Maurerin [24] mukaan ratkaisuna tälle ongelmalle voisi olla palkatut ohjaajat, jotka kurssin opettajan lisäksi toimisivat paikan päällä olevan asiakkaan korvikkeina.

Jyväskylän yliopistossa lehtori Vesa Lappalaisen Ohjelmointi 2 [22], sekä Graafisten käyttöliittymien ohjelmointi [21] -kursseilla opiskelijat voivat tehdä harjoitustyönsä haluamastaan aiheesta, kun vain sille annetut vaatimuksen täyttyvät. Tällöin asiakkaana toimivat opiskelijat itse. Myös ohjaajat, joille he käyvät harjoitustöitään näyttämässä, toimivat jonkinlaisina asiakkaina, mutta pääasiassa opiskelijat itse tietävät mitä he haluavat kehitettävältä ohjelmistolta. Tämä vähentää ohjaajien työtaakkaa, mutta silti opiskelijat pääsevät näkemään asiakkaan reaktion käydessään esittelemässä harjoitustyönsä eri vaiheita ohjaajalle.

Mikäli opiskelijaprojektille ei saada asiakasta istumaan fyysisesti paikan päälle, käy helposti kuten kurssilla, josta Melnik ja Maurer [24] kertoivat. Kynnys lähettää sähköpostia tai viestiä keskustelupalstalle voi olla jo niin korkea, että mieluummin tehdään omia olettamuksia ja ratkaisuja. Varsinkin, jos vastausta tarvittaisiin nopeasti, voi päivän tai muutamankin tunnin odottelu tuntua liian pitkältä.

Vaikka asiakas olisikin palkattu ulkopuolelta niin, että häneen voi ottaa aina tarvittaessa yhteyttä, tuskin tämä viikonloppuisin ja iltaisin suostuu olemaan jatkuvasti tavoitettavissa. Parasta olisi, mikäli olisi mahdollista järjestää säännölliset työajat 4.2.10, jolloin tiimin opiskelijat ja asiakas voisivat olla samaan aikaan paikalla.

Opiskeluympäristössä on varmasti usein tehtävä tämän käytännön osalta kompromisseja, mutta silti opiskelijat voivat oppia ymmärtämään paikan päällä olevan asiakkaan hyödyn.

6.12 Koodausstandardit

Opiskelijoilla ei ole yleensä vielä paljoakaan kokemusta toisten kirjoittaman ohjelmakoodin lukemisesta, jolloin heille ei myöskään ole vielä kehittynyt selkeää näkemystä, minkälainen ohjelmakoodi on selkeää ja miellyttävää lukea. Tämän takia ei ole välttämättä tarkoituksenmukaista käyttää paljoa aikaa siihen, että opiskelijat määrittelisivät itse käyttämänsä koodausstandardin. Koodausstandardi voidaan antaa opiskelijoille valmiina, tai sitten koodausstandardi voidaan antaa projektiryhmälle valittavaksi joistakin yleisesti käytetyistä koodausstandardeista.

Kokeneemmilla ohjelmoijilla on yleensä kehittynyt oma ohjelmointityylinsä, mutta he kykenevät tarpeen vaatiessa mukautumaan myös erilaisiin koodausstandardeihin. Vähemmän kokeneet ohjelmoijat taas pitävät helposti omaa ohjelmointityyliänsä oikeana ja muunlaisia väärinä. Jos koodausstandardi annetaan opiskelijoille valmiina, ei ryhmässä pääse syntymään turhia erimielisyyksiä siitä, kenen ohjelmointityyli valitaan noudatettavaksi. Valmiiksi annettu koodausstandardi voi lisätä ryhmän tehokkuutta, sillä tällöin ryhmän ei tarvitse käyttää aikaa kiistelyyn

parhaasta ohjelmointityylistä. [6]

Olisi varmasti hyvä, että opiskelijat oppisivat jo opiskeluaikanaan kirjoittamaan ohjelmakoodia samalla tyylillä kuin mahdollisissa tulevissa työpaikoissaan. Yleisesti käytetyt standardit ovat käytännössä hyväksi havaittuja ja tällaisten koodausstandardien mukaista ohjelmakoodia on miellyttävää lukea.

6.13 Yleisiä suosituksia XP:n soveltamiseen

Sanders [28] listaa kuusi suositusta liittyen XP:hen yliopisto-opiskelijoiden opinto-ohjelmassa.

- Pidä huoli, että XP:n esittely ei ole ristiriidassa muiden laajempien opetustavoitteiden kanssa
- Opeta osio käyttäen XP:tä tai ketteriä menetelmiä ohjelmistotekniikan kurssilla, mutta projekteissa käytä näitä harkiten
- Sopiva muokattu versio XP:stä voi olla käyttökelpoinen joillekin projekteille tai tiimeille
- Johonkin tiettyyn aiheeseen keskittynyt kurssi on paras paikka käyttää ketteriä menetelmiä
- Jotkut XP:n käytännöt voivat olla jaoteltu tiettyjen kurssien kesken
- Pariohjelmointi on kaikkein tehokkainta, kun opiskelijat ovat kehittäneet yksilöllisiä taitojaan, parit ovat taidoiltaan saman tasoisia ja kurssilla on säännöllisesti laboratoriotyöskentelyä

Muissa tutkimuksissa on myös tuotu esille yleisiä XP:n soveltamiseen liittyviä suosituksia. Shukla ja Williams [31] esittelevät käyttämänsä hybridimetodologian, Melnik ja Maurer [24] tuovat esille monia yleisiä XP:n soveltamiseen liittyviä periaatteita, sekä Keefe ja Dick [19] suosittelevat prosessivalmennusta.

6.13.1 Hybridimetodologia

Usein opetussuunnitelmaan on varattu ainoastaan yksi kurssi ohjelmistotekniikalle. Tällä yhdellä kurssilla pitäisi opettaa sekä perinteisempiä että ketteriä menetelmiä. Shukla ja Williams [31] North Carolina State -yliopistosta ovat huomanneet, että yksi lukukausi ei ole riittävän pitkä aika opettaa kahta todella erilaista metodologiaa.

Kuitenkin opiskelijoille tulisi opettaa joukko erilaisia taitoja ja tekniikoita, joista he voivat sitten järkevästi valita omaan projektiinsa parhaiten sopivat käytännöt.

Shukla ja Williams [31] suosittelevat ohjelmistotekniikan kurssille käytettäväksi eräänlaista hybridimetodologiaa, joka pitää sisällään käytänteitä ketteristä ja perinteisistä metodologioista. Tämä metodologia sisältää XP:n käytännöistä suunnittelupelin, pariohjelmoinnin, testauksen, jatkuvan integroinnin, yhteisomistajuuden, uudelleenrakentamisen, koodausstandardit, sekä paikan päällä olevan asiakkaan. Paikan päällä olevana asiakkaana toimii kurssin opetushenkilökunta. Projektin alussa tehdään projektisuunnitelma, jossa dokumentoidaan tiimin organisatorinen rakenne, käyttäjätarinat ja niiden omistajat, projektin valmistumisaikataulu, konfiguraatioiden hallintasuunnitelma, alustava korkean tason suunnitelma käyttäen UML-luokkakaavioita, sekä kattava joukko hyväksyntä/mustalaatikko -testitapauksia.

Omilla kursseillaan Williams [31] antoi opiskelijoiden kokeilla myös katselmointeja sekä keskustella pariohjelmoinnista vaihtoehtona virallisille katselmoinneille. Opiskelijat saivat laatia joitakin käyttötapauksia ja keskustella näistä vaihtoehtona käyttäjätarinoille. Lisäksi opiskelijat oppivat myös muista UML-kaavioista, kuten sekvenssi- ja tilakaavioista.

Tämä hybridimalli on jouduttu luomaan olosuhteiden pakosta. Mikäli opetussuunnitelmaan on varattu useampi kurssi, jossa erilaisia metodologioita voi soveltaa käytäntöön, ei tällaista hybridimallia välttämättä tarvita. Yhdellä kurssilla voitaisiin käyttää puhtaasti XP:tä ja toisella jotain perinteisempää metodologiaa. Näin opiskelija voisi päästä paremmin vertailemaan näitä erilaisia lähestymistapoja sekä arvioimaan niiden heikkouksia ja vahvuuksia.

6.13.2 Opiskelijoiden työskentelyn tarkkailu ja arviointi

Harjoitustyön kehityksen aikana ei Melnikin ja Maurerin [24] mukaan ole tarpeen tarkkailla opiskelijoita. Toisten tiimissä olevien opiskelijoiden aiheuttama paine on tarpeeksi suuri motivaattori.

Valitettavasti aina toisten opiskelijoiden aiheuttama paine tehdä töitä ei ole riittävä, vaan saattaa pahimmillaan käydä jopa niin, että projektiryhmästä yksi tekee lähes kaiken työn, eikä uskalla tai kehtaa ilmiantaa toisia. Tämän takia olisi tärkeää olla jokin keino kontrolloida ja saada selville, minkä verran kukakin on projektin eteen todellisuudessa tehnyt.

Tehokkain tapa tarkistaa, ovatko opiskelijat työskennelleet projektissa vai eivät, on antaa heille koetehtäväksi toteuttaa itsenäisesti käytännössä joku toimiva kokonaisuus. Melnik ja Maurer [24] ovat omalla kurssillaan antaneet opiskelijoiden ottaa

kokeeseen mukaan aiemmin kirjoittamaansa koodia, sekä käyttää internetistä löytyviä lähteitä. Kokeen aikana opiskelijoita tarkkailtiin ja katsottiin, että he eivät lähetä viestejä tai tiedostoja toisilleen.

Jos käytössä on joku versionhallintajärjestelmä, kuten esimerkiksi CVS, voisi sieltä myös tarkistaa kuinka paljon kukakin käyttäjä, tai ainakin pari, on sinne lisännyt ohjelmakoodia. Toki tämäkin voi olla hieman epäluotettava keino, mutta ainakin se voisi olla jonkinlaisena tukena arvioinnille epäselvissä tilanteissa.

Wilsonin [39] kokemus hänen pitämällään kurssilla oli, että opiskelijoihin täytyisi pitää enemmän yhteyttä ja erityisesti pyytää heitä kertomaan miten projekti etenee. Kun luentojen aikana opiskelijoita pyydettiin raportoimaan projektin etenemisestä, he eivät mielellään olisi tuoneet esille mahdollisia ongelmia tehtävien ajallaan loppuun saattamisessa. Wilsonin mukaan olisi ollut parempi, mikäli jokaisesta projektin osaprojektista olisi valittu yksi edustaja, joka pitäisi säännöllisesti yhteyttä ryhmän jäsenten kanssa ja pitäisi aina ohjaajan ajan tasalla

6.13.3 Pääperiaatteiden määrittäminen

Melnik ja Maurer [24] suosittelevat määrittelemään selkeät periaatteet, joita projektissa tulee noudattaa. XP:n varsinaisten käytäntöjen ja arvojen lisäksi hyviä periaatteita ovat: vastuun ottaminen, ”Et tule tarvitsemaan sitä” -sääntö, sekä toisten kirjoittamaan ohjelmakoodiin luottaminen.

Melnik ja Maurer [24] kehoittivat myös opiskelijoita välttämään kiintymistä kirjoittamaansa ohjelmakoodiin. Jos omaan koodiin suhtaudutaan liian tunteenomaisesti, ei toisten koodia uskalleta mennä korjaamaan eikä uudelleenrakentamaan 4.2.6. Yhteisomistajuus-käytännön 4.2.8 noudattaminen vaatii, että kaikki voivat käydä muokkaamassa mitä tahansa kehitettävän ohjelmiston ohjelmakoodia. Tarpeeton koodi tulee voida poistaa ja näin tehdä ohjelmakoodista yksinkertaisempaa ja helpommin ymmärrettävää.

Schneiderin ja Johnstonin [30] mukaan yksi avainelementti XP:n onnistuneeseen soveltamiseen opetusympäristössä on saada aikaisessa vaiheessa juurrutettua opiskelijoiden mieleen epäitsekkään työskentelyn käsite. Opiskelijoille tulisi painottaa, että yhteistyö ja tiiminä toimiminen on ohjelmistotekniikassa ratkaisevan tärkeää.

Kun opiskelijoita pyydettiin [24] antamaan lisäksi muita sääntöjä, jotka auttaisivat luomaan parhaan oppimis/tuotanto -ympäristön, he toivat esille seuraavat säännöt: yritä ratkaista ongelmat ensin itse, sitten vasta kysy apua; jaa uudet ja mielenkiintoiset asiat ryhmän kanssa; ole innostunut; ole kohtelias; ei valitusta; ei huonoja tekosyitä.

Olisi varmasti hyvä määritellä periaatteet sekä itse ohjelmakoodin tuottamiseen, että myös ryhmässä käyttäytymiseen. Jos ryhmädynamiikka ei toimi, ei kehitettävä ohjelmistokaan todennäköisesti edisty niin nopeasti. Olisi tärkeää saada luotua kehitystiimistä toimiva ryhmä, joka ponnistelee yhdessä kohti samaa päämäärää.

6.13.4 Oma tila kehitystiimille

Olisi hyvä järjestää kehitystiimeille omat työhuoneet, tai ainakin joku oma tila, jossa he voivat työskennellä yhdessä silloin kuin heille sopii parhaiten. Opiskelijoille voi olla motivoivampaa, jos heille on varattu oma paikka heidän kehitystyötänsä varten.

Jyväskylän yliopiston sovellusprojekteissa [29] tämä on normaali käytäntö. Jokaisella projektiryhmällä on oma työhuoneensa, ja jokaisella projektin jäsenellä oma henkilökohtainen työasemansa. Projekteissa mukana oleville jaetaan sähköavaimet, joilla he pääsevät sisään yliopistolle ja projektitiloihin myös iltaisin ja viikonloppuisin. Tilojen puolesta Jyväskylän yliopiston projektikursseilla voitaisiin hyvin viedä joku projekti läpi käyttäen XP:tä.

Mikäli ei ole mahdollista järjestää kehitystiimeille omia työhuoneita, niin ainakin joku pysyvä työskentelytila olisi hyvä olla olemassa. Esimerkiksi joku mikroluokka voisi olla varattu ensisijaisesti kurssin opiskelijoiden käyttöön.

Omassa työhuoneessa on etuina se, että tiimi voi tehdä töitä omassa rauhassaan, ilman että täytyisi hajaantua johonkin mikroluokkaan muiden tiimien ja opiskelijoiden sekaan. XP:ssä korostetaan kasvokkain tapahtuvan kommunikaation merkitystä, ja jos samassa mikroluokassa on useita tiimejä yhtä aikaa kommunikoimassa, työrauhasta ei ole tietoakaan.

Olisi myös hyvä jos tiimillä olisi käytössään oma ilmoitustaulu, johon voitaisiin listata tehtäviä ja muistettavia asioita, sekä joku valkotaulu tai liitutaulu, johon voisi piirtää kaavioita ja suunnitella ohjelman rakennetta.

Mikäli työasemista on pulaa, pariohjelmoinnin käytännön ansiosta ei jokaiselle kehittäjälle välttämättä tarvitsisi edes varata omaa tietokonetta, koska kaikki tuotantoon menevä koodi on tarkoitus ohjelmoida pareittain yhdellä koneella. Jos jokaiselle on mahdollista järjestää oma työasema, niin kokeiluiden tekeminen, tiedon hankinta ja muut tarvittavat toiminnot on tällöin helpompaa.

6.13.5 Laboratorioajat

XP:ssä on tärkeää, että ohjelmointitiimi on työskentelemässä samanaikaisesti. Erityisesti parien olisi tärkeää olla paikalla yhtä aikaa. Työelämässä tämä on yleensä

helppoa järjestää, mutta opiskeluympäristössä opiskelijoiden hyvinkin erilaisia aikatauluja on joskus vaikeaa sovittaa yhteen.

Jos kurssille määritellään pakolliset laboratorioajat, jolloin koko kehitystiimi on paikalla, saadaan ryhmä ainakin joksikin aikaa työskentelemään samanaikaisesti. Tällaisina aikoina myös ohjaaja voisi olla paikalla avustamassa eteen tulevissa ongelmissa.

Valmiiksi aikataulutetut työskentelyajat eivät varmastikaan riitä koko sovelluksen kehittämiseen, joten vieläkin tiimeillä on edessään aikataulujen yhteensovittamisongelmia. Opiskeluympäristössä on vaan hyväksyttävä, että yhteistä aikaa ei aina löydy, mutta olisi hyvä että opiskelijat saisivat ainakin jonkinlaisen kuvan siitä, minkälaista on kehittää sovellusta pienessä tiimissä, kun kaikki ovat yhtä aikaa paikan päällä.

6.13.6 Puhtaalta pöydältä aloittaminen, vai olemassa olevan jatkokehittäminen?

Melnik ja Maurer [24] suosittelevat, jos mahdollista, että opiskelijoille annettaisiin projektiksi jonkin uuden ohjelmiston kehittäminen alusta alkaen itse. Heidän tutkimuksessaan eräällä ketteriä menetelmiä noudattelevalla kurssilla opiskelijoille annettiin jatkokehitettäväksi sovellus, jota ei oltu kehitetty aiemmin käyttäen ketteriä menetelmiä. Tämän takia olemassa olevalle koodille ei ollut valmiita testitapauksia, ja ohjelmakoodin uudelleenrakentaminen sekä uusien toiminnallisuuksien integrointi oli hankalaa. Muutokset ja uudet toiminnallisuudet rikkoivat helposti vanhan ohjelmakoodin toiminnallisuuden, eikä testitapausten puuttumisen takia ongelmia ollut niin helppo havaita.

XP:tä tai muuta ketterää menetelmää käytettäessä olisi tärkeää, että ohjelmisto olisi kehitetty alusta alkaen käyttäen ketteriä menetelmiä. Muutenkin lyhyellä kurssilla opiskelijoille voisi olla parempi, että he saisivat lähteä kehittämään sovellusta alusta alkaen itse. Olemassa olevaan järjestelmään tutustuminen vie muuten melko suuren osan kurssin ajasta, eikä varsinaista uutta toiminnallisuutta päästä toteuttamaan niin nopeasti. Vaikka opiskelijoille annettaisiinkin tehtäväksi uuden toiminnallisuuden kehittäminen olemassa olevaan järjestelmään, olisi hyvä että kehitettävä osio olisi mahdollisimman itsenäinen ja riippumaton muusta järjestelmästä. Tällöin ei välttämättä haittaa, vaikka testitapauksia aiemmin kehitetystä järjestelmästä ei olisikaan olemassa.

Molemmat tapaukset voivat olla omalla tavallaan opettavaisia. Mikäli opiskelijat pääsevät kehittämään sovellusta alusta alkaen itse, he oppivat enemmän ohjelmiston rakenteen suunnittelua, kuin jos he jatkokehittäisivät olemassa olevaa ohjelmis-

toa. Työelämässä taas aika moni löytää itsensä jatkokehittämästä jotakin olemassa olevaa järjestelmää, joten miksei tätäkin voisi harjoitella jo opiskeluaikana. On tärkeä taito osata lukea ja ymmärtää toisten kirjoittamaa ohjelmakoodia, sekä lisätä siihen toiminnallisuutta tai muokata sen rakennetta paremmaksi.

6.13.7 Tunnettujen tekniikoiden käyttäminen

Projektikurssin alussa opiskelijoilla on paljon uutta opittavaa, joten edes käytettävä teknologia olisi hyvä olla tuttua. Kun opiskelijoilla on jo kokemusta projektissa käytettävän ohjelmointikielen ja tekniikan käytöstä, he pääsevät aloittamaan varsinaisen työskentelyn nopeammin, ovat tuottavampia ja osaavat arvioida paremmin eri ohjelmointitehtäviin kuluvia aikoja. [24]

Jos opiskelijoille annetaan projektissa uusi ohjelmointikieli ja tekniikka opittavaksi, on hankalaa arvioida, kuinka kauan heillä sen oppimiseen menee. Jos projektitiimissä osa pääsee nopeasti alkuun varsinaisessa toteutuksessa ja osalla menee kauan uuden tekniikan opettelussa, hitaammat saattavat jäädä projektin aikana lähinnä sivustakatsojan rooliin.

Mikäli opiskelijoilla on jo kohtuullisen vahva ohjelmointitausta, ei uuden ohjelmointikielen tai tekniikan opettelu ole enää mikään kynnyskysymys. Kokeneemmille ohjelmoijille tällainen on vain hyvää harjoitusta, sillä työelämässäkin tulee vastaan tilanteita, joissa täytyy opiskella itsenäisesti jotakin uutta ja päivittää omaa osaamistaan.

6.13.8 Ohjelmistotekniikan käytäntöjen noudattaminen

Kun jotakin ohjelmistoa kehitetään tiimityönä, ei riitä että osataan ohjelmoida, vaan on myös osattava noudattaa vastuullisia ohjelmistotekniikan käytänteitä. Opiskelijoiden tulisi oppia ottamaan henkilökohtaisesti vastuuta projektin etenemisestä ja pitää huoli siitä, että tehtävät jotka he ovat luvanneet tehdä, tulevat tehdyksi vaatimusten mukaisesti. Myös löydetyt bugit tulee korjata.

Melnikin ja Maurerin [24] kursseilla opiskelijoiden annettiin itse muodostaa ryhmät, valita ryhmän johtaja ja sopia toteutettavista tehtävistä. Tehtäviä valitessaan he sitoutuivat pitämään huolen siitä, että ne tulevat toteutetuksi oikein. Kurssien luonteesta johtuen eri työtehtävät rakentuivat toistensa päälle, ja jos kurssin ohjaajat huomasivat palautetuissa tehtävissä joitakin tiettyjä ongelmia, ne oli ehdottomasti korjattava seuraavaksi palautuskerraksi.

6.13.9 Jatkuva prosessivalmennus

Keefe ja Dick [19] tuovat omassa tutkimuksessaan esille prosessivalmennuksen (Process Coaching) tarpeen. Heidän projektikurssinsa XP-projektiryhmä pyrki noudattamaan XP:n käytäntöjä, mutta kun he kohtasivat vaikeita tilanteita, he palasivat nopeasti takaisin perinteisiin käytäntöihin. Mikäli aika oli loppumassa kesken tai jos projektiin kohdistui muita paineita, he hylkäsivät ensimmäisenä pariohjelmoinnin, jatkuvan integroinnin, sekä testilähtöisen ohjelmoinnin käytännöt. Jopa projektin loppupuolella, kun opiskelijat olivat jo todenneet miten he työskentelivät tuottavammin ja laadukkaammin ohjelmoidessaan pareittain, he edelleen palasivat työskentelemään yksittäin, kun he halusivat saada enemmän tulosta aikaiseksi. Jatkuvalla kurssin pitäjän valmennuksella ja ohjauksella opiskelijat kuitenkin palasivat takaisin XP:n käytäntöihin.

Kun prosessivalmennusta annettiin paljon, opiskelijat noudattivat jatkuvasti XP:n käytäntöjä, eivätkä palanneet työskentelemään omien tottumustensa mukaisesti. Kurssin viimeisen kuukauden ajaksi tutkijat päättivät vähentää prosessivalmennusta, sillä yksi opiskelija ei päässyt tekemään töitä muun ryhmän kanssa, vaan joutui tekemään töitä kotoaan käsin. Kun valmennus ja ohjaus loppui, opiskelijat palasivat takaisin työskentelemään vanhojen tottumustensa mukaisesti ja monien XP:n käytäntöjen noudattaminen loppui. [19]

7 Yhteenveto

Teollisuudessa on jatkuvasti kasvava tarve ohjelmiston kehittäjille, jotka ovat valmiita noudattamaan ketteriä menetelmiä, joista Extreme Programming on kaikkein suosituin ja yleisin. Tällä hetkellä monet joutuvat opettelemaan nämä ketterän ohjelmistokehityksen vaatimat taidot vasta työelämässä, koska opiskelujen aikana niitä ei ole riittävän hyvin tuotu esille ja opetettu soveltamaan käytäntöön.

Monet XP:n käytännöistä 4.2 ovat hyödyllisiä myös muutenkin, kuin vain XP-kontekstissa. Esimerkiksi testilähtöinen ohjelmointi 4.2.5, pariohjelmointi 4.2.7 ja koodausstandardit 4.2.12 ovat suureksi hyödyksi, vaikka muita XP:n käytäntöjä ei noudatettaisikaan. XP:tä noudatettaessa myös opiskelijoiden, monesti vielä hieman heikot, kommunikaatiotaidot 4.1 kehittyvät. Tämä on tärkeää tulevaa työtä varten, jossa ohjelmistoja lähes poikkeuksetta kehitetään tiimityönä yhteistyössä erilaisten ihmisten kanssa.

Vaikka XP:tä ei ollakaan vielä laajasti otettu käyttöön ohjelmoinnin ja ohjelmistotekniikan opetuksessa, on sen soveltamisesta erilaisissa opiskeluympäristöissä jo ihan kiitettävästi kokemusta 5. Useimmiten XP:n soveltamisesta saadut kokemukset ovat olleet myönteisiä, vaikka joidenkin käytäntöjen noudattamisessa onkin ollut hankaluuksia.

Tässä tutkimuksessa käytetyn aineiston perusteella XP:n käytöllä ohjelmoinnin opetuksessa näyttäisi olevan positiiviset vaikutukset opiskelijoiden oppimiseen. Tutkimusten tekijöillä on vaihtelevia arvioita siitä, mitä XP:n osa-alueita kannattaa ottaa opetukseen sellaisenaan mukaan, mitä hieman soveltaen ja mitä ei ollenkaan.

Yksimielisimpiä tutkimuksissa oltiin suunnittelupelin 5.1, pienten julkaisujen 5.2, testauksen 5.5, yhteisomistajuuden 5.8, pariohjelmoinnin 5.7, jatkuvan integroinnin 5.9, 40-tuntisen työviikon 5.10, paikan päällä olevan asiakkaan 5.11 ja koodausstandardien 5.12 hyödyllisyydestä. Kaikkien näiden käytäntöjen soveltamisessa ei aina onnistuttu täydellisesti tai niitä ei yritettykään toteuttaa, mutta ainakin jälkeenpäin nähtiin niiden tärkeys. Vaikka pariohjelmointi 5.7 nähtiin hyödyllisenä, sen toteuttaminen tuotti usein hankaluuksia. Aikataulujen yhteensovittamisongelmat ja eritasoiset parit aiheuttivat monesti sen, että opiskelijat eivät noudattaneet tätä käytäntöä. Myös testauksessa 5.5 opiskelijat toimivat usein vastoin XP:n käytäntöä 4.2.5, sillä testitapaukset kirjoitettiin monesti vasta varsinaisen koodin kirjoittamisen jälkeen.

Suunnittelupelin 5.1, jatkuvan integroinnin 5.9, 40-tuntisen työviikon 5.10 ja paikan päällä olevan asiakkaan 5.11 käytäntöjä oltiin jouduttu muokkaamaan niin, että ne sopivat paremmin opetusympäristöön. Paikan päällä olevana asiakkaana toimi yleensä aina joku kurssin ohjaajista ja joskus yksi ohjaaja saattoi olla monelle ryhmälle tavoitettavissa esimerkiksi sähköpostin välityksellä.

On mielenkiintoista nähdä, miten tulevaisuudessa XP:n rooli ohjelmistotekniikan opetuksessa, sekä teollisuudessa tulee muuttumaan. Tutkimuksessa käytetyn materiaalin perusteella voitaisiin XP:n käyttöä suositella sekä ohjelmistotekniikan opetukseen, että teollisuuteen. Extreme Programming on rehellinen tapa kehittää ohjelmistoja [33].

8 Viitteet

[1] A Sharp, LLC. *King of Dragon Pass*. Saatavilla WWW-muodossa <URL: http://www.a-sharp.com/kodp/>, 29.9.2004.

[2] Agile Alliance. *Roadmap to Agile methods and tools*. Saatavilla WWW-muodossa <URL: http://www.agilealliance.org/>, 12.3.2005.

[3] Auer, K., Meade, E., Reeves, G. *The Rules of XP*. Saatavilla WWW-muodossa <URL: http://www.rolemodelsoftware.com/>, viitattu 7.7.2005.

[4] Beck, K. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Longman, Inc., Upper Saddle River, NJ, 2000.

[5] Beck, K., ym. *Manifesto for Agile Software Development*. Saatavilla WWW-muodossa <URL: http://www.agilemanifesto.org/>, 2001.

[6] Bevan, J., Werner, L., McDowell, C. *Guidelines for the Use of Pair Programming in a Freshman Programming Class*. Proc. of the 15th Conference on Software Engineering Education and Training, 25–27, 2002.

[7] Boehm, B., Port, D., Brown, A. *Balancing Plan-Driven and Agile Methods in Software Engineering Project Courses*. Computer Science Education, 12–3 (2002), s. 187–195.

[8] Cockburn, A., Williams, L. *The Costs and Benefits of Pair Programming*. Humans and Technology, Salt Lake City, USA, 2000. Saatavilla WWW-muodossa <URL: http://collaboration.csc.ncsu.edu/laurie/>, viitattu 14.7.2005.

[9] Collin, C., Miller, R. *XP Distilled*. Saatavilla WWW-muodossa <URL: http://www.rolemodelsoftware.com/>, viitattu 7.7.2005.

[10] Electronic Arts inc. *Sim City^{tm} 4*. Saatavilla WWW-muodossa <URL: http://simcity.ea.com/>, viitattu 23.7.2005.

[11] Free Software Foundation, Inc. *CVS - Concurrent Versions System*. Saatavilla WWW-muodossa <URL: http://www.gnu.org/software/cvs/>, 2000.

[12] Goldman, A., Kon, F., Silva, P. *Being Extreme in the Classroom: Experiences Teaching XP*. University of São Paulo, Brasilia, 2004.

[13] Haikala, I., Märijärvi, J. *Ohjelmistotuotanto*. Suomen ATK-kustannus Oy, Helsinki, 1998.

[14] Hakkarainen, K., Lonka, K., Lipponen, L. *Tutkiva oppiminen – älykkään toiminnan rajat ja niiden ylittäminen*. WSOY, Porvoo, 2000.

[15] Hedin, G., Bendix, L., Magnusson, B. *Introducing software engineering by means of Extreme Programming*, ICSE (2003), s. 586–593.

[16] Hislop, G., Lutz, M., Naveda, J., McCracken, W., Mead, N., Williams, L. *Integrating Agile Practises into Software Engineering Courses*, Computer Science Education, 12–3 (2002), s. 169–185.

[17] Jeffries, R. *What is Extreme Programming?*. Saatavilla WWW-muodossa <URL: `http://www2.umassd.edu/SWPI/xp/papers/p586-hedin.pdf`>, 11.8.2001.

[18] Johnson, D. H., Caristi, J. *Using Extreme Programming in the Software Design Course*. Computer Science Education, 12–3 (2002), s. 223–234.

[19] Keefe, K., Dick, M. *Using Extreme Programming in a Capstone Project*. CRPIT '30: Proceedings of the sixth conference on Australian computing education, s. 151–160, 2004.

[20] Kivi, J., Haydon, D., Hayes, D., Schneider, R., Succi, G. *Extreme Programming: A University Team Design Experience*. IEEE (2000).

[21] Lappalainen, V. *Graafisten käyttöliittymien ohjelmointi 2004*. Saatavilla WWW-muodossa <URL: `http://www.mit.jyu.fi/vesal/kurssit/winohj04/`>, viitattu 23.7.2005.

[22] Lappalainen, V. *Ohjelmointi 2 - 2005*. Saatavilla WWW-muodossa <URL: `http://www.mit.jyu.fi/vesal/kurssit/ohjelmointi2005/`>, viitattu 23.7.2005.

[23] Melnik, G., Maurer, F. *Perceptions of Agile Practices: A Student Survey*. Proceedings Agile Universe/ XP Universe (2002), Springer (2002).

[24] Melnik, G., Maurer, F. *Introducing Agile Methods in Learning Environments: Lessons Learned*. Proceedings XP Agile Universe (2003), Springer (2003), s. 187–198.

[25] Pressman, R. *Software Engineering: a Practitioner's Approach*, 6th ed. The McGraw-Hill Companies, Inc., New York, 2005.

[26] RoleModel Software. *Our Apprenticeship Model*. Saatavilla WWW-muodossa <URL: `http://www.rolemodelsoftware.com`>, viitattu 9.7.2005.

[27] Royce, W. *Managing the Development of Large Software Systems*. Proceeding of IEEE WESCON, August 1970. Saatavilla WWW-muodossa <URL: `http://facweb.cs.depaul.edu/jhuang/is553/Royce.pdf`>, viitattu 14.7.2005.

[28] Sanders, D. *Extreme Programming: The Student View*. Computer Science Education, 12–3 (2002), s. 235–250.

[29] Santanen, J. *Tietotekniikan Sovellusprojekteja Jyväskylän yliopistossa*. Saatavilla WWW-muodossa <URL: `http://www.mit.jyu.fi/palvelut/sovellusprojektit/`>, 21.9.2004.

[30] Schneider, J., Johnston, L. *eXtreme Programming at universities: an educational perspective*. ICSE (2003), s. 594–599.

[31] Shukla, A., Williams, L. *Adapting Extreme Programming For A Core Software Engineering Course*. CSEE (2002). Saatavilla WWW-muodossa <URL: `http://collaboration.csc.ncsu.edu/laurie/`>, viitattu 31.1.2005.

[32] Sommerville, I. *Software Engineering*, 7th edition. Pearson Education Limited, 2004.

[33] Talbott, N., Auer, K. *Apprenticeship in a Software Studio: An old and New Model for Education*. RoleModel Software, Inc. 2002-2003. Saatavilla WWW-muodossa <URL: `http://www.rolemodelsoftware.com`>, viitattu 9.7.2005.

[34] Tomayko, J. *A Comparison of Pair Programming to Inspections for Software Defect Reduction*. Computer Science Education, 12–3 (2002), s. 213–222.

[35] Weisert, C. *There's no such thing as the Waterfall Approach! (and there never was)*. Conrad, Information Disciplines, Inc., Chicago. Saatavilla WWW-muodossa <URL: `http://www.idinews.com/waterfall.html`>, 8.2.2003.

[36] Wells, D. *Extreme Programming: A gentle introduction*. Saatavilla WWW-muodossa <URL: `http://www.extremeprogramming.org/`>, 2004.

[37] Williams, L., Upchurch, R. *Extreme Programming for Software Engineering Education*. IEEE (2001).

[38] Williams, L., Wiebe, E., Yang, K., Ferzli, M., Miller, C. *In Support of Pair Programming in the Introductory Computer Science Course*. Computer Science Education, 12–3 (2002), s. 197–212.

[39] Wilson, D. *Teaching XP: A Case Study*. XP Universe Conference Papers, Raleigh, North Carolina, (2001).