Saimaan ammattikorkeakoulu
Tekniikka, Lappeenranta
Tietotekniikka
Ohjelmistotekniikka

Sirkku Berg

Ketterä ohjelmistontuotantomenetelmä: Scrum

Opinnäytetyö 2012

Tiivistelmä

Sirkku Berg
Ketterä ohjelmistotuotantomenetelmä: Scrum, 46 sivua, 1 liite
Saimaan ammattikorkeakoulu, Lappeenranta
Tekniikka, Tietotekniikka
Ohjelmointitekniikka
Opinnäytetyö 2012
Ohjaajat: Koulutuspäällikkö Päivi Ovaska, Saimaan ammattikorkeakoulu,
Site Manager Jani Tietäväinen, Digia Oyj

Opinnäytetyöni tarkoituksena oli selvittää, mitä Scrum on ja millaisiin projekteihin se soveltuu parhaiten. Lisäksi opinnäytetyössäni on käyty läpi myös muita ohjelmistontuotantomenetelmiä. Tavoitteenani oli myös tutkia, kuinka ketterissä ohjelmistontuotantomenetelmissä on otettu huomioon ohjelmistojen testaaminen.

Opinnäytetyötäni varten keräsin laajasti teoriapohjaista tietoa eri lähteistä, mutta teorian lisäksi päätin tehdä myös haastatteluja. Haastattelujen avulla pyrin saamaan selville kokemusperäistä tietoa Scrumista.

Ohjelmistojen vaatimukset muuttuvat jatkuvasti, joten ohjelmistontuotantomenetelmää valittaessa tähän tulee kiinnittää erityistä huomiota. Menetelmän tulee auttaa näiden muutosten hallinnassa. Perinteiset ohjelmistontuotantomenetelmät ovat usein hyvin järjestelmällisiä ja jäykkiä, joten yksikin muutos saattaa aiheuttaa hyvin suuria rahallisia tai aikataulullisia tappioita.

Ketterät menetelmät, kuten Scrum, on suunniteltu vastaamaan näihin haasteisiin. Mahdolliset muutokset eivät välttämättä ole muutoksia, koska alkuperäistä ominaisuutta tai toimintoa ei välttämättä ole vielä edes suunniteltu. Tämä helpottaa muutosten vastaanottamista ja hallintaa.

Testaamiseen kiinnitetään jatkuvasti enemmän huomiota. Mitä enemmän ja aikaisemmassa vaiheessa voidaan testata, sitä edullisemmaksi virheiden korjaus tulee. Myös testausmenetelmien valitsemiseen pitää panostaa.

Avainsanat: ohjelmistontuotantomenetelmä, ketterät menetelmät, Scrum, testaus

Abstract

Sirkku Berg
Agile software development: Scrum, 46 pages, 1 appendix
Saimaa University of Applied Sciences, Lappeenranta
Technology, Information Technology
Software Engineering
Bachelor´s Thesis 2011
Instructor: Training Manager Päivi Ovaska, Saimaa University of Applied Sciences, Site Manager Jani Tietäväinen, Digia Oyj

The aim of this thesis was to find out what Scrum is and for what kind of projects Scrum is the most suitable. Also other software development methods were studied. The other part of this thesis was to investigate how testing is organized in agile methods.

For this thesis, wide theory based knowledge was collected but also some empirical information by interviewing experts.

Nowadays software requirements are constantly changing, so you must pay attention to choosing a right software development method. The right method helps to manage these changes. Traditional software development is quite often very systematic and strict, so even one change can lead to very large financial losses.

Agile methods, like Scrum, are designed to answer these kinds of challenges. Possible changes are not necessarily the changes since the original feature or function does not necessarily have even been planned. This makes it easier to manage changes.

Testing is an important part of software development. It is more useful, and less expensive, to find errors at an early stage. So you must also pay attention to choose a right testing method to support the software development.

Keywords: software development, agile methods, Scrum, software testing

Sisältö

1 Johdanto

1.1 Tausta ja tarkoitus

Tämän opinnäytetyön tarkoituksena on käydä läpi erilaisia ohjelmistotuotantomenetelmiä, etenkin Scrum-nimistä ketterää menetelmää. Aluksi käydään läpi teoriapohjalta ohjelmistotuotantoa ja sen menetelmiä. Tämän jälkeen on oma kappaleensa niin Scrumin kuin testauksen teorialle.

Jotta opinnäytetyö ei jäisi vain teoriapohjalle, luvussa 5 on kuvattu, kuinka tutkimustyö tähän aiheeseen liittyen suoritettiin. Lisäksi luvussa 6 on tutkimuksen lopputulokset.

1.2 Tutkimusmenetelmät

Tutkimusmenetelmänä käytettiin henkilökohtaisia haastatteluja. Jokaiselle haastateltavalle esitettiin lähes samat kysymykset. Yksi haastattelu kesti keskimäärin 30 minuuttia. Haastateltavia oli yhteensä 7 henkilöä.

2 Ohjelmistotuotantomenetelmät

Termi ohjelmistotuotanto (ja ohjelmistotekniikka) tulee englanninkielisestä termistä ”Software Engineering”. Termi on aiheuttanut runsaasti keskustelua jo sen käyttöönotosta lähtien (1960-luvun loppupuolella). Keskusteluissa on monesti kyseenalaistettu ohjelmistotekniikan olemassaolo erillisenä tekniikan alueena. Kirjallisuudesta löytyy useita määritelmiä ohjelmistotekniikalle, kuten

> *The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software (IEEE 1990.).*
>
> *Establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines (Naur & Randell 1969.).*

Vapaasti tulkittuna termin voidaan katsoa tarkoittavan ohjelmistotyötä, jonka lopputuloksena saadaan järjestelmä, joka täyttää käyttäjiensä kohtuulliset toiveet ja odotukset. Tämän lisäksi järjestelmän tulisi valmistua ennalta laaditun aikataulun ja kustannusarvion puitteessa. Termin ”*Software Engineering*” voidaan siis katsoa käsittävän kaikki ohjelmiston tuotantoprosessiin liittyvät osa-alueet: laatujärjestelmä, projektinhallinta, dokumentointi, tuotteenhallinta, laadunvarmistus, testaus, määrittely, suunnittelu, toteutus, käyttöönotto ja ylläpito. (Haikala & Märijärvi 2004,16.)

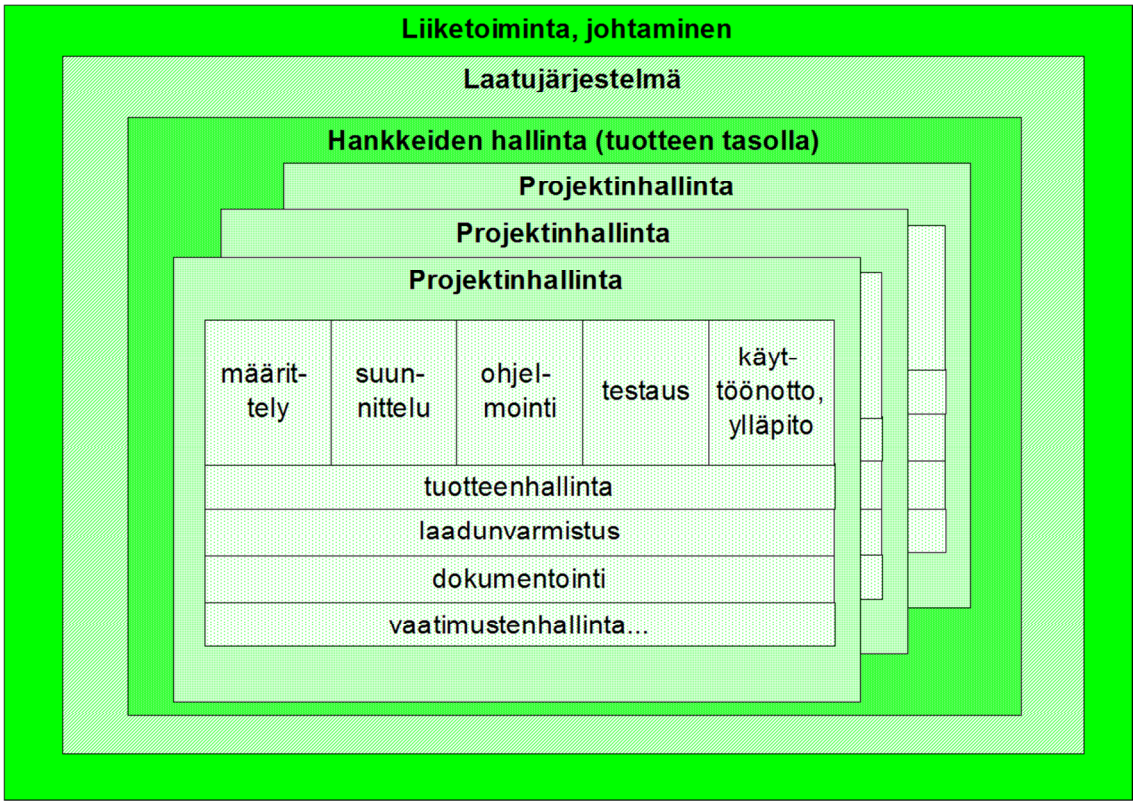
Ohjelmistotuotantomenetelmä-termi voidaan määritellä kokoelmana malleja (*patterns*), jotka määrittävät aktiviteetit, toiminnot, työtehtävät sekä projekteihin liittyvät menettelytavat. Toisin sanoen, ohjelmistotuotantomenetelmät tarjoavat mallin, johon sisältyy ohjelmistotuotantoprojektin tärkeät vaiheet. Yhdistelemällä eri menetelmiä, projektiryhmä voi rakentaa oman prosessimallin, joka vastaa parhaiten heidän tarpeitaan. (Pressman 2005, 31.)

Ohjelmistotuotantomenetelmän olemassa olo ei kuitenkaan takaa sitä, että ohjelma tai järjestelmä saataisiin valmiiksi halutussa aikataulussa tai että se vastaa asiakkaan tarpeita. Prosessimalli pitää yhdistää vahvaan ohjelmistotuotan-

non ammattitaitoon. Lisäksi, prosessimalli itsessään pitäisi määrittää niin, että se täyttää perusprosessin vaatimukset, jotka on todettu oleellisiksi menestyvälle ohjelmistotuotantoprojektille.(Pressman 2005, 34.)

2.1 Perinteiset menetelmät

Ohjelmistotuotanto voidaan jakaa eri osa-alueisiin, kuten kuvassa 2.1 on tehty. Ohjelmiston kehitysprosessiin kuuluvat yleensä seuraavat vaiheet: määrittely, suunnittelu, ohjelmointi ja testaus. Tavallisesti näiden vaiheiden jälkeen tulee käyttöönotto ja ylläpito. Projektiin liittyy myös koko sen elinkaaren aikana useita eri tukitoimintoja, kuten laadunvarmistus ja dokumentointi. (Haikala & Märijärvi 2004, 35.)



Kuva 2.1 Ohjelmistotuotannon eri osa-alueet (Ovaska 2002)

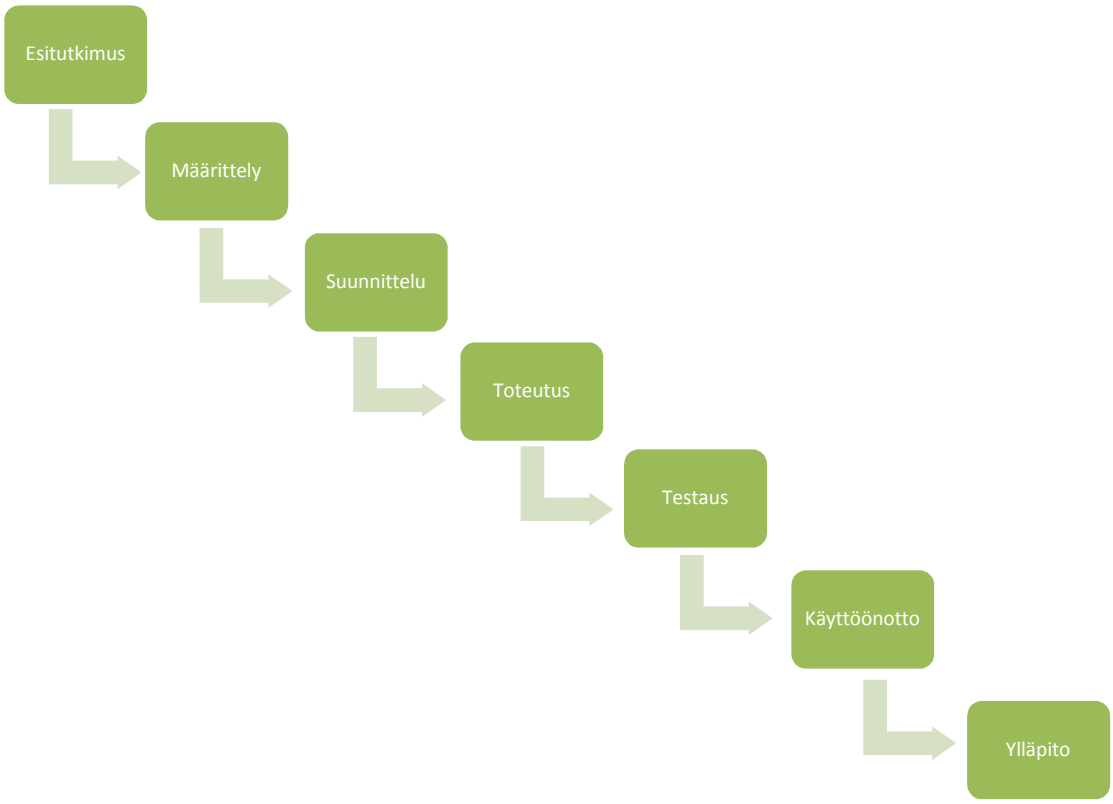
Ohjelmistotuotantomenetelmiä kutsutaan usein vaihejakomalleiksi. Nimitys tulee siitä, että tapana on jakaa ohjelmiston kehitystyö eri vaiheisiin. Perinteisin vaihejakomalli on vesiputousmalli (*waterfall model*). Muita perinteisiä vaihejakomalleja ovat esimerkiksi protoilumallit (koodaa ja korjaa) ja erilaiset inkremen-

taaliset mallit, kuten niin sanottu Evo-malli (*evolutionary delivery*). (Haikala & Märijärvi 2004, 36 – 42.)

Seuraavissa aliluvuissa on kerrottu tarkemmin vesiputous- ja ”koodaa & korjaa” -malleista.

2.1.1 Vesiputousmalli

Vesiputousmallia voisi kuvailla sanoilla: systemaattinen ja peräkkäinen lähestymistapa. Eri lähteistä riippuen vesiputousmallissa on vaiheet jaettu viidestä kuuteen osaan. Eri vaiheita ovat yleensä ainakin määrittely-, suunnittelu- ja toteutusvaiheet. Määrittelyvaihetta ennen on usein esitutkimusvaihe, mutta lähteestä riippuen esitutkimus saattaa sisältyä itse määrittelyvaiheeseen. Kuvassa 2.2 on esitetty yksi versio vesiputousmallista. (Pressman 2005; Haikala & Märijärvi 2004.)



Kuva 2.2 Vesiputousmalli

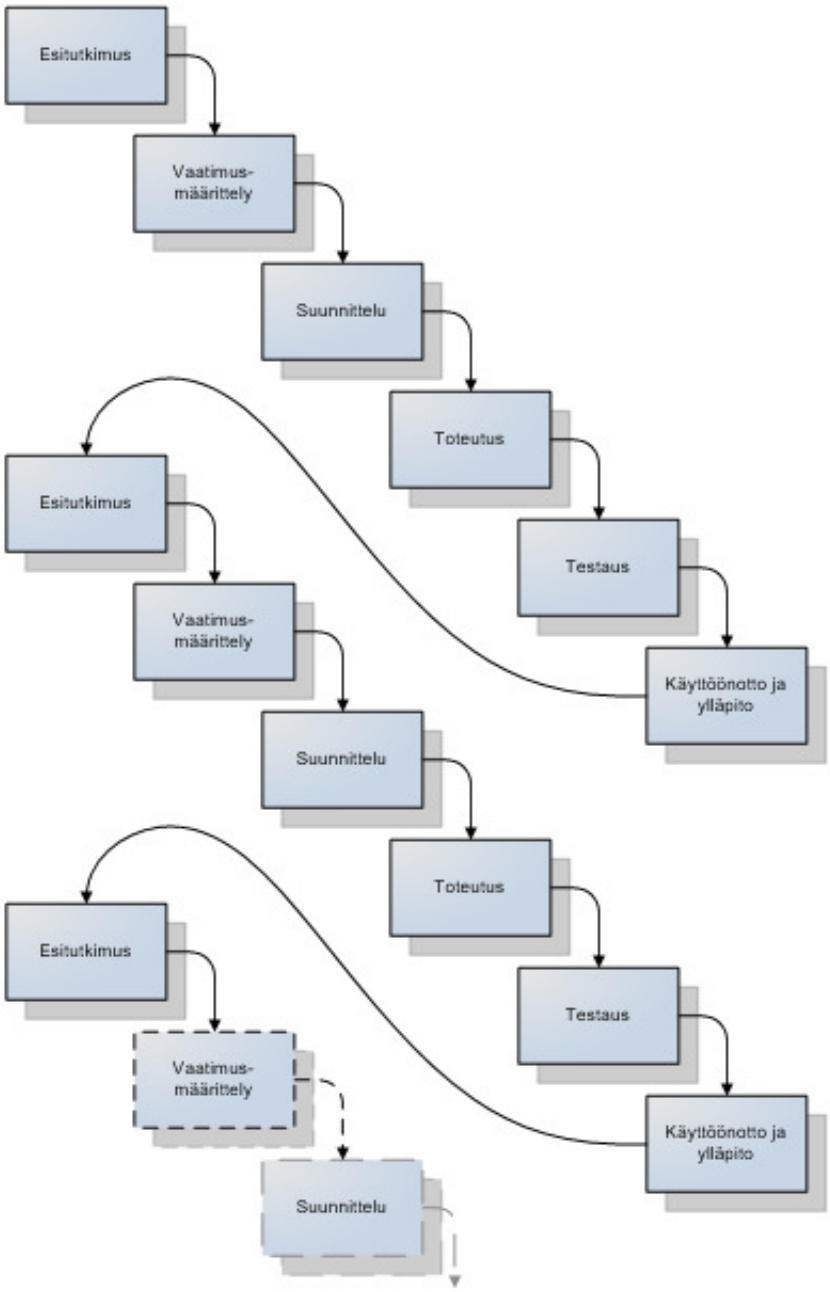
Vesiputousmalli on vanhimpia ohjelmistotuotantomenetelmiä. Kuitenkin, viime vuosikymmenten aikana vesiputousmalli on saanut osakseen suurta kritisointia

koskien sen tehokkuutta projekteissa. Kritisointi on nimenomaan aiheuttanut vesiputousmallin huonohko reagoiminen vaatimusten muutoksiin. Vesiputousmallin mukaisesti asiakkaan tulisi määritellä kaikki vaatimukset tarkasti heti projektin alussa, mutta käytäntö on osoittanut, että tämä saattaa olla usein hyvinkin hankalaa. Toisena vesiputousmallin heikkona kohtana on se, että asiakas joutuu odottamaan projektin loppuvaiheille asti ennen kuin hän saa ensimmäisen toimivan ohjelmaversion. Jos ohjelmaversiossa havaitaankin virheitä, joudutaan palaamaan takaisin projektin alkuun. Tästä seuraa aikataulun ja budjetin uudelleenlaskenta. (Pressman 2005, 47 – 48.)

Tänä päivänä ohjelmistotyöt ovat usein nopeatahtisia ja vaativat nopeaa reagointikykyä muutoksiin. Vesiputousmalli on usein epäkäytännöllinen tällaisiin projekteihin. Kuitenkin, se voi tarjota hyvinkin käytännöllisen prosessimallin tilanteisiin, joissa vaatimukset ovat tarkkaan mietittyjä ja työ on tarkoitus tehdä etenemällä lineaarisesti. (Pressman 2005, 48.)

2.1.2 Inkrementaalinen kehitys

Inkrementaalinen kehityksen pohjana on perinteinen vesiputousmalli, mutta inkrementaalisessa kehityksessä toteutusvaihe voidaan toteuttaa vaiheittain. Inkrementaalinen kehitys voi myös sisältää useita kokonaisia vesiputousmalleja peräkkäin (kuva 2.3).



Kuva 2.3 Inkrementaalinen malli, joka sisältää peräkkäisiä vesiputouksia (Ahonen 2010)

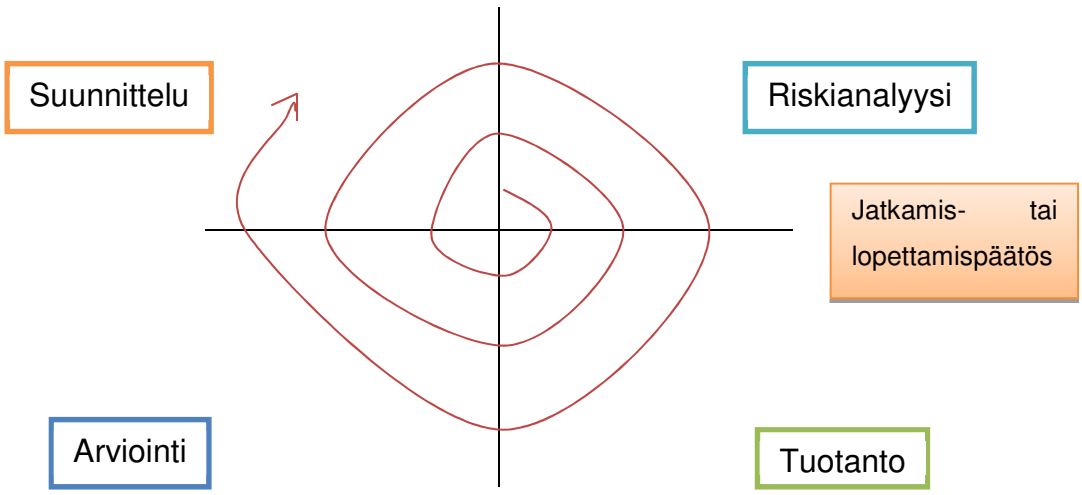
Inkrementaalisessa kehityksessä ensimmäiset vaiheet sisältävät välttämättömimmät ja tärkeimmät ominaisuudet. Vaihe vaiheelta ohjelmaversioon lisätään lisäominaisuuksia ja viimeisessä vaiheessa voidaan lopulta tehdä vain viimeistelyjä. Inkrementaalinen kehitys on askel ketterämmän kehityksen suuntaan, jossa muutoksiin reagointi on vesiputousmallia helpompaa. (Ahonen 2010.)

2.1.3 Muita perinteisiä ohjelmistotuotantomalleja

Evoluutiomallissa on tavoitteena ohjelmiston kehittäminen pienien väliversioiden kautta. Ohjelmistoa kehitetään asiakkaan palautteiden perusteella. Suunnittelu ja testaus ovat hyvin vähäistä evoluutiomallissa. Malli soveltuu parhaiten projekteihin, joissa asiakkaan vaatimukset ovat hyvin epäselvät, mutta yhteistyö asiakkaan kanssa sujuu erittäin hyvin.

Prototyyppien avulla voidaan asiakkaalle esittää nopeallakin aikataululla haluttavasta järjestelmästä demoversio. Demoversio on yleensä ulkonäöltään hyvin lähellä haluttua lopputulosta, mutta puutteellinen yksityiskohdiltaan. Tämän ohjelmistotuotantomallin ongelmana voi olla kaksinkertainen työ samasta järjestelmästä, eikä prototyyppi välttämättä paljasta kaikkia yksityiskohtaisia ongelmia. Kyseinen malli sopii hyvin projekteihin, joissa asiakas haluaa niin sanottua näyttöä tulevasta järjestelmästä ennen sen varsinaista toteutusta. Prototyyppimalli soveltuu myös hyvin uusien teknisten ratkaisujen kokeilemiseen.

Spiraalimallin keskeinen periaate on iteratiivisuus (kuva 2.4). Projektin aikana riskejä tarkastellaan jatkuvasti ja prosessia ohjataan riskianalyysien tulosten perusteella.



Kuva 2.4 Spiraalimalli

Spiraalimalli perustuu neljään toistuvaan vaiheeseen: suunnittelu, riskianalyysi, tuotanto ja arviointi. Suunnittelun aikana määritellään tavoitteet, vaihtoehdot ja rajoitukset. Riskianalyysissä arvioidaan eri vaihtoehtoihin liittyviä riskejä. Tuotantovaiheessa toteutetaan projektivaihe. Lopuksi asiakas arvioi lopputuloksen. Näitä vaiheita jatketaan, kunnes järjestelmä on valmis. Spiraalimalli ei ota kantaa itse tuotantovaiheeseen eikä siihen, miten itse tuotanto suoritetaan. Yksityiskohtia tarkennellaan jatkuvasti projektin edetessä ja jos riskit kasvavat liikaa, toteutus voidaan keskeyttää.

2.2 Ketterät menetelmät

Vuonna 2011 pidetyssä kokouksessa, johon oli kokoontunut useiden eri menetelmien kehittäjiä, julkaistiin ketterän sovelluskehityksen manifesti (*Agile Software Development Manifesto*). Manifesti määrittelee yhteiset arvot ja periaatteet ketterälle ohjelmistokehitykselle. Kaikki ketterät menetelmät noudattavat näitä yleisiä periaatteita ja perusarvoja.

> *”**Yksilöitä ja kanssakäymistä** enemmän kuin menetelmiä ja työkaluja.*
> ***Toimivaa ohjelmistoa** enemmän kuin kattavaa dokumentaatiota.*
> ***Asiakasyhteistyötä** enemmän kuin sopimusneuvotteluja.*
> ***Vastaamista muutokseen** enemmän kuin pitäytymistä suunnitelmassa.”*
> (Agile Alliance 2001.)

Julistuksen takana olevat periaatteet (Agile Alliance 2001.):

1. *Tärkein tavoitteemme on tyydyttää asiakas toimittamalla tämän tarpeet täyttäviä versioita ohjelmistosta aikaisessa vaiheessa ja säännöllisesti.*
2. *Otamme vastaan muuttuvat vaatimukset myös kehityksen myöhäisessä vaiheessa. Ketterät menetelmät hyödyntävät muutosta asiakkaan kilpailukyvyn edistämiseksi.*
3. *Toimitamme versioita toimivasta ohjelmistosta säännöllisesti, parin viikon tai kuukauden välein, ja suosimme lyhyempää aikaväliä.*
4. *Liiketoiminnan edustajien ja ohjelmistokehittäjien tulee työskennellä yhdessä päivittäin koko projektin ajan.*

5. *Rakennamme projektit motivoituneiden yksilöiden ympärille. Annamme heille puitteet ja tuen, jonka he tarvitsevat ja luotamme siihen, että he saavat työn tehtyä.*
6. *Tehokkain ja toimivin tapa tiedon välittämiseksi kehitystiimille ja tiimin jäsenten kesken on kasvokkain käytävä keskustelu.*
7. *Toimiva ohjelmisto on edistymisen ensisijainen mittari.*
8. *Ketterät menetelmät kannustavat kestävään toimintatapaan. Hankkeen omistajien, kehittäjien ja ohjelmiston käyttäjien tulisi pystyä ylläpitämään työtahtinsa hamaan tulevaisuuteen.*
9. *Teknisen laadun ja ohjelmiston hyvän rakenteen jatkuva huomiointi edesauttaa ketteryyttä.*
10. *Yksinkertaisuus – tekemättä jätettävän työn maksimointi – on oleellista.*
11. *Parhaat arkkitehtuurit, vaatimukset ja suunnitelmat syntyvät itseorganisoituvista tiimeistä.*
12. *Tiimi tarkastelee säännöllisesti, kuinka parantaa tehokkuuttaan, ja mukauttaa toimintaansa sen mukaisesti.*

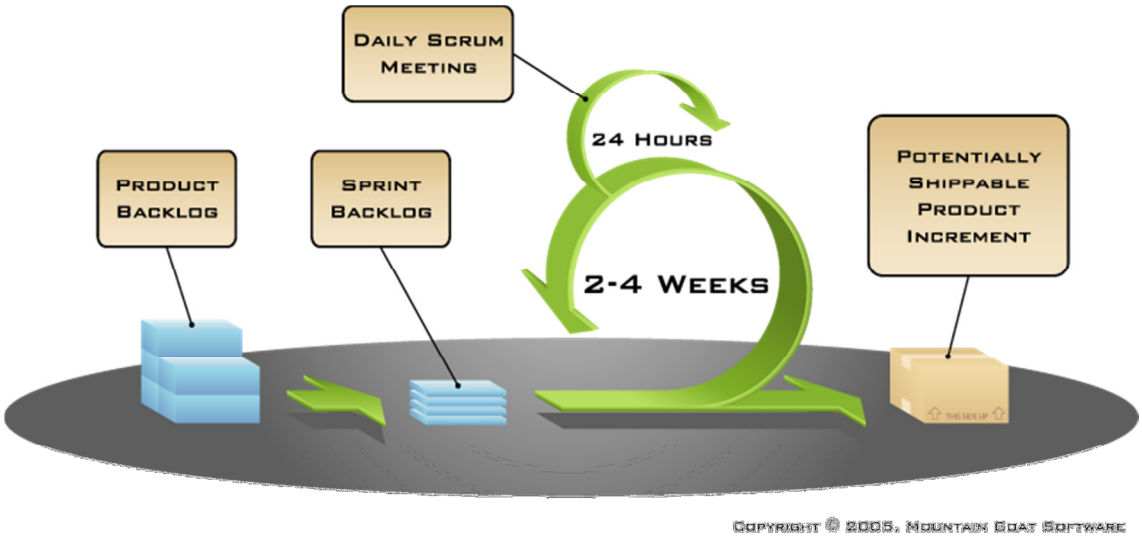
Näistä arvoista ja periaatteista voidaan kiteyttää ketterien menetelmien tunnusmerkeiksi seuraavanlaiset asiat: Ketterä ohjelmistokehitys on inkrementaalista, yhteistyössä tapahtuvaa, suoraviivaista ja sopeutuvaa (Huttunen 2006.).

2.2.1 Scrum

Scrum on yksi ketteristä ohjelmistotuotantomenetelmistä. Se ei ole niinkään mikään kokonainen prosessi tai metodologia, vaan enemmänkin eräänlainen runko. Scrum ei tarjoa valmiita, yksityiskohtaisia ohjeita, kuinka projektissa tulisi asiat tehdä, vaan se jättää päätösten tekemisen kehitystiimille. Yleensä tiimi osaa itse ratkaista ongelmat projektille sopivalla tavalla. Scrumin perustana on itseohjautuva, monitoiminen tiimi. Itseohjautuvassa tiimissä ei ole varsinaista tiimin johtajaa, joka päättää, kuka tekee mitäkin tehtävää ja kuinka ongelmat ratkaistaan. Tiimi keskenään tekee päätökset tuon kaltaisiin ongelmiin. Kun tiimi on monitoiminen, niin jokainen tiimin jäsen voi tarvittaessa suunnitella ja toteuttaa toiminnon ideasta valmiiksi tuotokseksi. (Cohn 2011.)

Näitä ketteriä kehitystiimejä tukee kaksi erityistä yksilöä: Scrum-mestari (*ScrumMaster*) ja Tuotteen omistaja (*Product Owner*). Scrum-mestarin voidaan ajatella olevan eräänlainen valmentaja tiimille. Hän auttaa tiimin jäseniä käyttämään Scrumin runkoa mahdollisimman tehokkaasti, jotta he suoriutuisivat tehtävistään korkeimmalla mahdollisella panoksella. Tuotteen omistaja edustaa yritystä, asiakasta tai käyttäjää ja ohjaa tiimiä rakentamaan oikeanlaisen tuotteen. (Cohn 2011.)

Scrum-projekti etenee useissa pyrähdyksissä (*sprint*), jotka ovat aikarajoitettuja iteraatioita (kuva 2.5). Yhden iteraation pituus tulisi olla maksimissaan yhden kuukauden mittainen. Pyrähdyksen alussa tiimin jäsenet lupautuvat tekemään tietyn määrän eri toimintoja, jotka on listattu projektin tuotetyölistassa (*product backlog*). Pyrähdyksen lopussa ennalta määrätyt toiminnot ovat valmiina, ohjelmoitu, testattu ja integroitu kehitteillä olevaan tuotteeseen tai systeemiin. Jokaisen pyrähdyksen lopulla on myös pyrähdyksen katselmointi (*sprint review*), jossa tiimi esittelee uudet toiminnallisuudet tuotteen omistajalle ja muille kiinnostuneille sidosryhmille. (Cohn 2011.)



Kuva 2.5 Scrum-prosessi (Cohn 2011)

2.2.2 XP

Scrumin ohella yksi tunnetuimpia ketteriä menetelmiä on Extreme Programming (XP). Se on nimensä mukaisesti erittäin ohjelmointikeskeinen. Poiketen useista muista ketteristä menetelmistä, XP tarjoaa monia käytäntöjä, joita tulee noudat-

taa. Käytäntöjen noudattaminen on tärkeää, sillä ne ovat rakennettu tasapainottamaan toisiaan. XP:llä on monia periaatteita sekä myös omat arvot: kommunikointi, yksikertaisuus, palaute, kunnioitus ja rohkeus. (Kosonen 2005.)

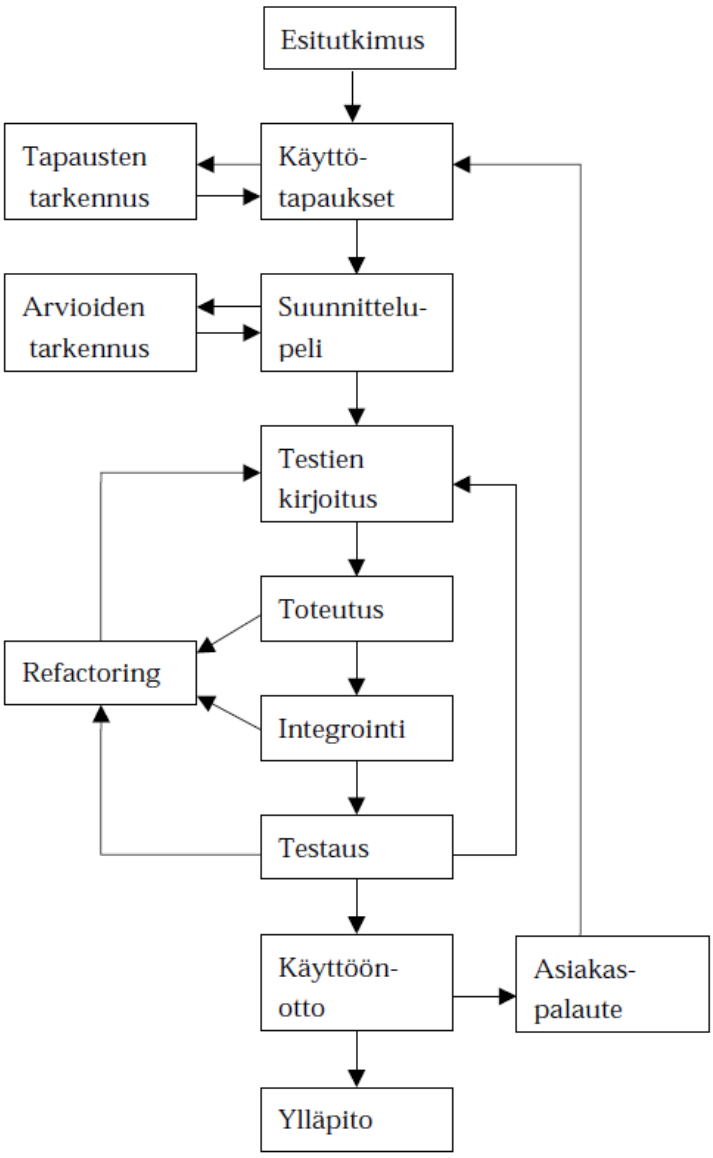
XP:n mukaisen projektin rakenne muistuttaa iteratiivisen kehityksen ja evoluutiomallin mukaisia rakenteita. Määrittelyvaihe eroaa esimerkiksi vesiputousmallista niin, että koko ohjelmistoa ei määritellä kerralla projektin alussa vaan ainoastaan ensimmäiseen versioon tarvittavat toiminnot. Määrittelyvaihetta on XP:ssä nimeltään suunnittelupeli (*planning game*). Määrittelyvaiheeseen osallistuvat vahvasti niin asiakas kuin toteuttajaryhmäkin. XP:ssä käytetään paljon metaforioita, joiden avulla asiakas ja projektiryhmä saavat yhteisen kielen. Metaforalla tarkoitetaan kielikuvaa, joka samaistaa kaksi toisiinsa periaatteessa liittymätöntä käsitettä toisiinsa niiden yhteisten piirteiden perusteella. Metaforan tarkoituksen on pitää kommunikaatio mahdollisimman yksinkertaisena ja sujuvana. (Lindberg 2003.)

Käyttötapauksia kutsutaan XP:ssä nimellä tarina (*story*). Asiakas eli loppukäyttäjä kirjoittaa useimmiten käyttötapaukset. Ennen varsinaista ohjelmointivaihetta, ohjelmoija kirjoittaa käyttötapaukselle moduulitestitapauksen (*test case*) ja vasta tämän jälkeen aloitetaan varsinaisen ohjelmakoodin kirjoitus. Ohjelmakoodi kirjoittaessa tulee käyttää yksinkertaisuuden periaatetta. Olio-ohjelmoinnille tyypillistä uudelleenkäytettävyyttä ei siis käytetä, jos uudelleenkäytön kohdetta ei ole jo valmiiksi tiedossa. Valmis ohjelmakoodi integroidaan järjestelmään heti valmistuttuaan, samalla tehdään myös automatisoidut testit. Mahdolliset virheet korjataan välittömästi. Tämän jälkeen voidaan aloittaa uuden käyttötapauksen luomista. XP:ssä käytetään ohjelmakoodia tuottaessa pariohjelmointia. Pariohjelmointi tarkoittaa siis sitä, että kaksi ohjelmoijaa työskentelee yhdessä yhdellä tietokoneella saman ohjelmakoodin parissa. Toisen ohjelmoijan koodatessa, toinen lukee kirjoitettua koodia läpi ja pyrkii korjaamaan kaikki havaitsemansa virheet heti. (Lindberg 2003.)

Kun moduulitestaukset on suoritettu, koodille tehdään vielä asiakkaan tekemä toiminnallinen testaus (*functional testing*). Asiakas suorittaa itse testin projektiryhmän avustuksella. Kaikki mahdolliset virheet korjataan heti ja testiä suoritetaan niin kauan, kunnes virheitä ei enää esiinny. Valmis ohjelmiston osa voi-

daan luovuttaa asiakkaalle, kun kaikki versioon suunnitellut käyttötapaukset ovat ohjelmoitu ja testattu. Tämän jälkeen aloitetaan uusi kehityskierros suunnittelupelillä. (Lindberg 2003.)

Niin ikään muiden ketterien menetelmien mukaisesti, myös XP noudattaa lean developmentin jätteen vähentämisen periaatetta dokumenttien suhteen. Kaikki dokumentit, joista ei ole hyötyä projektin valmistuttua, ovat XP:n mukaan jätettä. (Lindberg 2003.)



Kuva 2.6: Extreme Programming -metodologian vaiheet (Lindberg 2003)

XP:ssä on myös henkilöstöllä eri rooleja. Yhdellä henkilöllä voi olla useampi rooli samassa projektissa. Eri rooleja ovat muun muassa ohjelmoija, asiakas,

testaaja, mittaaja (*tracker*), valmentaja (*coach*), konsultti ja johtaja (*big boss*). (Lindberg 2003.)

Ohjelmoija suunnittelee ja kirjoittaa koodin muiden ohjelmoijien kanssa. Hänen vastuulla on myös muun muassa ylläpito, käyttötapaukset, suunnittelupeli, testien kirjoitus, toteutus, integrointi, testaus, käyttöönotto, asiakaspalaute ja moduulitestaus. XP-projektissa asiakkaan kanssa kommunikoivat kaikki projektiryhmän jäsenet, kun perinteisissä menetelmissä yhteyshenkilönä on toiminut ainoastaan projektipäällikkö. (Lindberg 2003.)

Asiakkaan vastuulla on kirjoittaa käyttötapaukset ja suunnitella toiminnallisia testejä. Hänen vastuullaan on myös priorisoida käyttötapaukset ja päättää, milloin jokin käyttötapaus on hyväksytysti suoritettu. (Lindberg 2003.)

Testaajat auttavat asiakasta toiminnallisten testien suunnittelussa, ja he voivat myös suorittaa varsinaisen toiminnallisen testauksen. Mittaaja seuraa projektin edistymistä ja annettujen aikatauluarvioiden toteutumista. Mittaajan rooli on perinteisessä projektiorganisaatioissa kuulunut yleensä projektipäällikölle. (Lindberg 2003.)

Valmentaja on Extreme Programming -metodologian asiantuntija ja opastaa ryhmää XP:n soveltamisessa käytäntöön. Hän myös valvoo, että XP:n arvoja ja toimintatapoja noudatetaan. Konsultti on liiketoiminnallisen tai teknisen alueen erikoisasiantuntija, joka auttaa ryhmää alueeseen liittyvissä erityiskysymyksissä. Konsultteja voi olla useita ja heidän roolinsa projektissa on lähinnä vieraileva. Johtajan vastuulla on päätöksenteko. Hän seuraa projektia ja poistaa edistymisen tiellä olevia esteitä. (Lindberg 2003.)

2.2.3 TDD

Test-Driven design (TDD, testivetoinen kehitys) on evoluutionaarinen lähestymistapa ohjelmistokehitykseen. TDD:ssä ohjelmakoodit kirjoitetaan pienissä osissa ennalta suunniteltujen testitapausten perusteella. Ohjelmakoodia kirjoitetaan juuri sen verran, että se läpäisee testin. Yksi näkemys TDD:n tavoitteista onkin nimenomaan määrittely, eikä niinkään validointi. Toisin sanoen, TDD on yksi tapa tarkastella vaatimukset ja suunnitelmat läpi ennen kuin kirjoitetaan

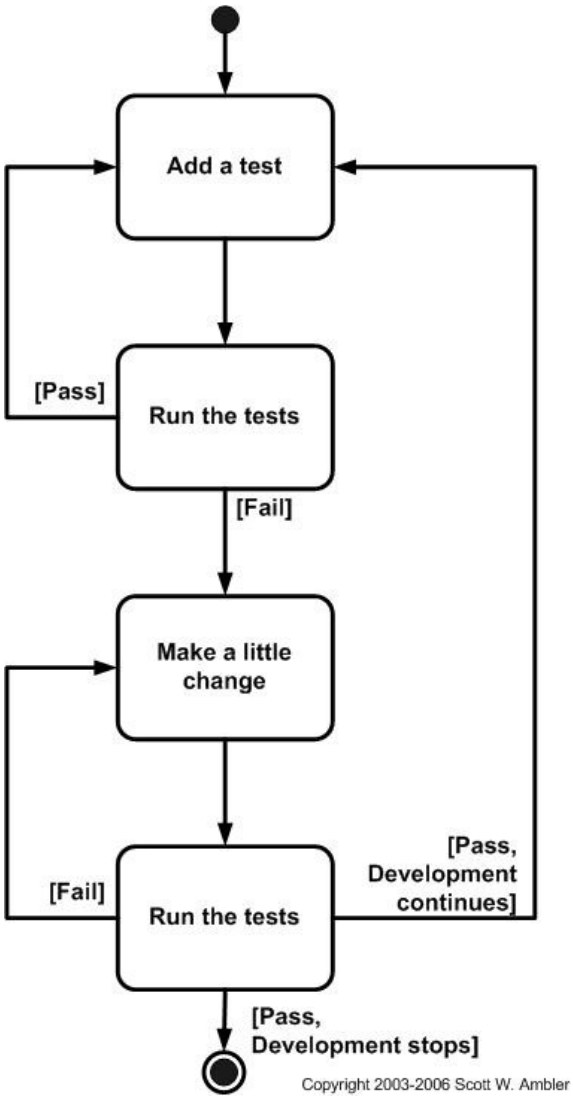
toiminnallista ohjelmakoodia. Toinen näkemys on, että TDD on ohjelmointitekniikka. TDD:n tavoitteena on saada aikaiseksi puhdasta ohjelmakoodia, joka toimii. (Ambler 2011.)

TDD:n ensimmäiset askeleet etenevät UML-aktiviteettidiagrammin mukaisesti (Kuva 2.7). Ensimmäinen tehtävä on lisätä testi ja kirjoittaa koodia juuri sen verran, että testi epäonnistuu. Seuraavana tehtävänä on ajaa testi läpi, jotta voidaan nimenomaan varmistaa testin epäonnistuminen. Kolmas askel on kirjoittaa testin läpäisevää toiminnallista ohjelmakoodia. Viimeinen kohta on ajaa testi uudelleen, ja jos testi epäonnistuu, ohjelmakoodia päivitetään ja ajetaan testi uudestaan. (Ambler 2011.)

Kun testi on saatu ajettua läpi onnistuneesti, ohjelmakoodi refaktoroidaan. Refaktorointi tarkoittaa, että ohjelmakoodi muokataan laadukkaammaksi poistamalla turhat silmukat, yhdistelemällä päällekkäisyydet sekä tuotetaan ylipäätään helposti ymmärrettävää ja ylläpidettävää ohjelmakoodia. Testien avulla huomataan helposti, jos refaktorointi rikkoo ohjelmakoodin. Näitä eri vaiheita toistetaan jokaisen uuden ominaisuuden kehityksessä, kunnes kaikki halutut ominaisuudet on toteutettu. (Vestola 2008.)

TDD:tä käyttäessä tarvitaan usein jonkinlainen testauskehysympäristö. Suosituimpia työkaluja ovat erilaiset vapaan lähdekoodin xUnit-työkalut, kuten JUnit (Java) ja NUnit (.NET). Työkaluja voi myös rakentaa itse, mutta olemassa olevat tarjoavat valmiiksi muun muassa graafisen käyttöliittymän ja muita ominaisuuksia, jotka helpottavat osittain työskentelyä. (Shore 2010.)

Yksi TDD:n tärkeimpiä ominaisuuksia on testausnopeus. Jokaisen minuutin aikana tulisi testit ajaa yhden tai jopa kahden kerran. Testien tulisi kestää enimmillään kymmenen sekuntia, jotta ne eivät aiheuta työmotivaatiokatkoksia. (Shore 2010.)



Kuva 2.7 Test-Driven-Development (Ambler 2011)

3 Scrum

Scrum on esitelty ja julkaistu OOPSLA-seminaarissa vuonna 1995. Pääsääntöisesti Scrum-menetelmää ovat kehittäneet Jeff Sutherland, Jeff McKenna, Ken Schwaber, Mike Smith ja Chris Martin. Myös Mike Beedle ja Martine Devos ovat kehittäneet merkittävästi Scrumia eteenpäin.

Scrum ei ole niinkään tuotekehitysprosessi tai -tekniikka, vaan se on enemmänkin eräänlainen viitekehys, joka antaa käyttäjilleen vapauden valita haluamansa prosessit ja tekniikat, joita projektissa käytetään. Scrumin viitekehys koostuu Scrum-tiimeistä rooleineen, aikarajoista, dokumenteista ja säännöistä. Scrum-tiimien tulee olla itseohjautuvia ja niiden tulee sisältää kaikki tarvittava osaaminen. Tiimit työskentelevät iteraatioissa, joita kutsutaan Scrumissa pyrähdyksiksi (*sprint*). Jokaisessa tiimissä on kolme erilaista roolia: Scrum-mestari, Tuotteen omistaja ja Kehitystiimi. Näistä eri rooleista on kerrottu tarkemmin seuraavissa luvuissa.

Scrumiin kuuluvat seuraavanlaiset elementit, jotka toistuvat jokaisessa projektissa: julkaisun suunnittelukokous, pyrähdyksen suunnittelukokous (*Sprint Planning Meeting*), pyrähdys (*Sprint*), päivittäinen Scrum-palaveri (*Daily Scrum*), pyrähdyksen katselmointikokous (*Sprint Review Meeting*) ja pyrähdyksen retrospektiivi (*Sprint Retrospective*).

Scrumin tärkein osa on pyrähdys, joka on enintään kuukauden mittainen iteraatio eli kehitysjakso. Pyrähdyksen pituus tulee pysyä samana koko kehityshankkeen ajan. Jokaisen pyrähdyksen aikana pyritään tuottamaan julkaisukelpoinen tuote tai ominaisuus. Seuraava pyrähdys alkaa välittömästi edellisen päätyttyä.

Scrumiin kuuluu neljä pääasiallista dokumenttia. Tuotteen kehitysjono (*Product Backlog*) on priorisoitu lista kaikista niistä ominaisuuksista, joita tuotteeseen tarvitaan. Pyrähdyksen tehtävälista (*Sprint Backlog*) koostuu yhden pyrähdyksen aikana toteutettavista tehtävistä, joilla saadaan aikaiseksi julkaisukelpoinen tuotteen osa tai ominaisuus. Julkaisun edistymiskäyrä (*Product Burndown Chart*) mittaa tuotteen edistymistä suhteessa julkaisusuunnitelman aikatauluarvioon. Pyrähdyksen edistymiskäyrä (*Sprint Burndown Chart*) mittaa sen hetki-

sen pyrähdyksen edistymistä suhteessa pyrähdyksen aikatauluarvioon. (Schwaber & Sutherland 2009)

3.1 Scrum-tiimi

Scrum-tiimiin ei kuulu perinteisiä ohjelmistotuotantorooleja, kuten ohjelmoija, suunnittelija, testaaja tai arkkitehti. Jokainen tiimin jäsen työskentelee yhdessä saadakseen valmiiksi sen työn, jonka he ovat luvanneet tehdä pyrähdyksen aikana. Tyypillisesti Scrum-tiimiin kuuluu 5 – 9 henkilöä. Scrum-tiimin kokoa ei tulisi kasvattaa yli kymmeneen vaan tällöin mieluummin perustetaan toinen tiimi, jolla on omat tehtävänsä pyrähdyksen aikana.

> *”A chicken and a pig are walking down the road. The chicken says to the pig: “Do you want open a restaurant with me?” The pig considers the question and replies: “Yes, I’d like that. What do you want to call the restaurant?” The chicken replies: “Ham and eggs!” The pig stops, pauses and replies: ”On second thought, I don’t think I want to open a restaurant with you. I’d be committed, but you’d only be involved.” (Schwaber & Beedle 2002.)*

Scrumissa tiimin jäsenet (tuotteen omistaja, Scrum-mestari ja tiiminjäsen) sitoutuvat asetettuihin tavoitteisiin ja tekevät kaikkensa, jotta ne saavutetaan. Heitä kutsutaan sioiksi, kuten sika vitsissä, he sitoutuvat projektiin täysin. Kaikkia muita kutsutaan kanoiksi. Kanat voivat osallistua päivittäisiin Scrum-palavereihin, mutta heidän tulee pysytellä taustalla. Kanoilla ei ole oikeutta osallistua palaverikeskusteluun millään tavalla. Kanat ovat vain vieraita ja heidän tulee noudattaa Scrumin sääntöjä. (Schwaber & Beedle 2002.)

3.1.1 Scrum-mestari

Scrum-mestari on vastuussa siitä, että Scrumin periaatteita, käytäntöjä ja sääntöjä käytetään oikein. Scrum-mestari auttaa organisaatiota omaksumaan Scrumin. Scrum-mestarin tehtävän on myös opastaa ja valmentaa tiimiä työskentelemään tuottavammin ja kehittämään laadukkaampia tuotteita. (Schwaber 2004.)

Päivittäisessä Scrum-palaverissa (*Daily Scrum*) Scrum-mestari kuuntelee tarkasti, mitä tiimin jäsenillä on raportoitavaa. Hän vertailee prosessin etenemistä suunnitelmiin, jotka perustuvat pyrähdyksen tavoitteisiin (*sprint goals*) ja ennustuksiin, jotka on tehty edellisen Scrum-palaverin aikana. Esimerkiksi, jos joku tiimin jäsen on tehnyt samaa helpohkoa tehtävää jo useamman päivän ajan, hän todennäköisesti kaipaa apuja. Scrum-mestari yrittää määrittää tiimin etenemisnopeutta: onko se jumissa, räpiköikö se, eteneekö se? Jos tiimi tarvitsee tukea, Scrum-mestari selvittää, kuinka hän voi auttaa. (Schwaber 2004.)

Scrum-mestari työskentelee tuotteen omistajan ja tiimin kanssa luodakseen tuotteen kehitysjonon (*product backlog*) pyrähdykselle. Pyrähdyksen aikana Scrum-mestari johtaa kaikki päivittäiset Scrum-palaverit ja on vastuussa esteiden poistamisesta sekä siitä, että päätökset ovat tehty ripeästi. Scrum-mestari tulee myös raportoida johdolle pyrähdyksen etenemisestä. (Schwaber 2004.)

Tiimin vetäjä (*team leader*), projektin vetäjä (*project leader*) tai projektipäällikkö (*project manager*) yleensä omaksuu Scrum-mestarin roolin (Schwaber 2004.).

3.1.2 Tuotteen omistaja

Tuotteen omistaja (*Product Owner*) vastaa viime kädessä tuotteen ominaisuuksista ja siitä, että projekti onnistuu. Tuotteen omistaja voi olla esimerkiksi tuotepäällikkö, asiakkaan edustaja tai toimittajan tekninen projektipäällikkö. Tuotteen omistaja vastaa tuotteen kehitysjonosta ja kehitystiimin arvon maksimoimisesta. Tuotteen omistajan vastuulla on ylläpitää tuotteen kehitysjonoa ja varmistaa, että kaikki ovat tietoisia sen sisällöstä ja prioriteeteista. Tuotteen omistajan tulisi olla yksi henkilö, mutta hän voi käyttää apunaan eri komiteoita. Tuotteen omistajan tulee yksin päättää, miten tehtävät priorisoidaan kehitysjonossa. (Schwaber 2004.)

Tuotteen omistaja edustaa pääsääntöisesti asiakasta ja muita sidosryhmiä. Hän valvoo heidän etujaan, päättää tuotteen ominaisuuksista ja niihin vaikuttavista seikoista sekä vastaa tuotteen kannattavuudesta. (Schwaber 2004.)

3.1.3 Kehitystiimi

Scrum-tiimissä kehitystiimin jäsenet tuottavat kehitysjonon sisällön julkaisukelpoiseksi tuotteeksi. Kehitystiimin tulee olla monitaitoinen ja itseohjautuva. Kehitystiimi päättää itse, kuinka he tuottavat julkaisukelpoiset ominaisuudet. Kehitystiimin optimaalinen koko on alle kymmenen henkilöä.

Tiimin tulisi olla mahdollisimman monitaitoinen ja sen jäsenillä tulisi olla kaikkia tarvittavia taitoja, jotta se pääsen pyrähdyksen asettamiin tavoitteisiin. Scrum-tiimien tulee olla myös itseohjautuvia, niin että jokainen tiimin jäsen pystyy myötävaikuttamaan lopputuloksiin. Jokaisen tiimin jäsenen tulee antaa oma panoksensa ongelmien ratkaisuihin. Testaaja auttaa muita tiimin jäseniä tuottamaan laadukkaampaa koodi ja samalla nostaa sen tuottavuutta. (Schwaber & Beedle 2002.)

Tiimi valitsee itse tuotteen kehitysjonosta ne tehtävät, jotka uskovat pystyvänsä tekevänsä sen pyrähdyksen aikana. Perinteisissä ohjelmistotuotantomenetelmissä projektin johtaja yleensä kertoo, kuka tekee ja mitä tekee, mutta Scrumin sääntöjen mukaan tiimin jäsenet jakavat keskenään työt, eikä johto saa tähän puuttua.

Scrum-tiimi koostuu parhaassa tapauksessa vähintään yhdestä kokeneesta insinööristä. Hän opastaa nuorempia insinöörejä. Jokaisen pyrähdyksen aikana tiimi on velvoitettu testaamaan omat tuotoksensa. Joskus tiimiin voi kuulua erillinen laadunvarmistustestaaja, joka tekee testaukset. Mutta joka tapauksessa, jokaisen tiimin jäsenen on vähintään tehtävä yksikkötestaukset omalle ohjelmakoodilleen. Tiimissä voi olla myös tekninen kirjoittaja, joka tuottaa projektissa tarvittavat dokumentoinnit.

Yleensä kaikki tiimin jäsenet sitoutuvat tiimiinsä täysiaikaisesti, mutta silloin tällöin saatetaan tarvita myös osa-aikaisia tiimin jäseniä.

3.2 Scrumin vaiheet

Jokainen pyrähdys etenee saman kaavan mukaan. Pyrähdyksen alussa pidetään pyrähdyksen suunnittelukokous, jossa tuotteen omistaja esittelee priorisoidun listan halutuista ominaisuuksista. Tiimi valitsee itse pyrähdyksen aikana

toteutettavat toiminnot. Nämä toiminnot siirretään tuotteen kehitysjonosta pyrähdyksen kehitysjonoon.

Joka päivä pyrähdyksen aikana pidetään päivittäinen Scrum-palaveri. Tämä palaveri auttaa tiimi asettamaan tavoitteet joka päivälle ja pysymään niissä. Jokaisen tiiminjäsenen tulee osallistua tähän palaveriin.

Pyrähdyksen lopuksi tiimi esittelee valmiiksi saadut toiminnallisuudet pyrähdyksen katselmointikokouksessa. Lisäksi jokaisen pyrähdyksen jälkeen pidetään retrospektiivikokous, jossa tiimi käy läpi opittuja asioita ja mitä parannettavaa pyrähdyksen aikana olisi ollut.

3.2.1 Tuotteen kehitysjono

Tuotteen kehitysjono (*Product Backlog*) listaa kaikki mahdolliset ideat ja ajatukset, joita projektissa toteutettavaan tuotteeseen halutaan (kuva 3.1). Tuotteen kehitysjono on siis lista kaikista toiminnoista, tehtävistä, teknologioista, parannuksista ja virheistä, jotka korjataan tulevissa julkaisuissa. Tyypillisesti Scrummestari ja Tuotteen omistaja kirjoittavat ylös kaikki mahdolliset tehtävät, jotka tulevat helposti mieleen. Tuotteen kehitysjonoa voidaan täydentää jatkuvasti projektin edetessä. Näin voidaan helpommin mukautua asiakkaan muuttuviin toiveisiin.

Tuotteen kehitysjonon tehtäväkohdat voivat olla teknisiä tehtäviä, kuten ”Sisäänkirjatumisluokan uudelleen luonti aiheuttaa poikkeuksen”, tai enemmän käyttäjäkeskeisiä, kuten ”Salli kumoa-toiminto asetukset-ikkunassa”.

Tuotteen kehitysjono on alkuaan vaillinainen, se on vain lista asioista, joita tuote tai systeemi tarvitsee. Ensimmäinen tuotteen kehityslista voi olla lista vaatimuksista, jotka ovat poimittu eräänlaisesta näkemys dokumentista (*vision document*), koottu aivoriihi keskusteluista tai päätelty markkinointivaatimusdokumenteista. (Schwaber & Beedle 2002.)

Tuotteen omistaja on yksin vastuussa tuotteen kehitysjonosta ja vain hän voi muokata jonoa projektin edetessä. Tuotteen omistaja varmistaa, että kehitysjono on kaikkien nähtävillä ja että jokainen tehtävä on priorisoitu.

Tuotteen omistaja näyttää pyrähdyksen suunnittelukokouksessa (*Sprint Planning Meeting*) priorisoidun tuotteen kehitysjonon ja kuvailee sen tärkeimmät aiheet tiimille. Tiimi päättää, mitkä ominaisuudet tai tehtävät valitaan seuraavaan pyrähdykseen. Valitut ominaisuudet tai tehtävät siirretään tuotteen kehitysjonosta pyrähdyksen kehitysjonoon (*Sprint Backlog*). Samalla jokaiseen päätehtävään eritellään pienempiä alatehtäviä, jotta tiimi voi tehokkaammin jakaa työtehtävät pyrähdyksen aikana.

	Item #	Description	Est	By
Very High				
	1	**Finish database versioning**	16	KH
	2	**Get rid of unneeded shared Java in database**	8	KH
	-	**Add licensing**	-	-
	3	Concurrent user licensing	16	TG
	4	Demo / Eval licensing	16	TG
		Analysis Manager		
	5	File formats we support are out of date	160	TG
	6	Round-trip Analyses	250	MC
High				
	-	**Enforce unique names**	-	-
	7	In main application	24	KH
	8	In import	24	AM
	-	**Admin Program**	-	-
	9	Delete users	4	JM
	-	**Analysis Manager**	-	-
	10	When items are removed from an analysis, they should show up again in the pick list in lower 1/2 of the analysis tab	8	TG
	-	**Query**	-	-
	11	Support for wildcards when searching	16	T&A
	12	Sorting of number attributes to handle negative numbers	16	T&A
	13	Horizontal scrolling	12	T&A
	-	**Population Genetics**	-	-
	14	Frequency Manager	400	T&M
	15	Query Tool	400	T&M
	16	Additional Editors (which ones)	240	T&M
	17	Study Variable Manager	240	T&M
	18	Haplotypes	320	T&M
	19	**Add icons for v1.1 or 2.0**	-	-
	-	**Pedigree Manager**	-	-
	20	Validate Derived kindred	4	KH
Medium				
	-	**Explorer**	-	-
	21	Launch tab synchronization (only show queries/analyses for logged in users)	8	T&A
	22	Delete settings (?)	4	T&A

Kuva 3.1 Esimerkki tuotteen kehitysjonosta (Cohn 2011)

Esimerkkikuvan Excel-taulukossa jokaiselle tuotteen kehitysjonon kohdalle on asetettu prioriteetti (Erittäin tärkeä ”*Very High*”, Tärkeä ”*High*”, jne.). Priorisoinnin on tehnyt tuotteen omistaja. Kehittäjät antavat jokaiselle kohdalle alustavan työaika-arvion. Nämä työaika-arviot ovat tässä vaiheessa vielä hyvin karkeita ja niitä käytetään vain, jotta saadaan suuntaa-antava aikataulu projektille. Tuotteen kehitysjonon työaika-arviot eivät sido tiimiä, vaan ne ovat enemmänkin

”parhaimpia arvauksia” siitä, kuinka kauan tehtävän tekemiseen menee aikaa. Projektin edetessä ja tietoisuuden lisääntyessä työaika-arvioita voidaan muokata realistisemmiksi. (Schwaber & Beedle 2002, 35.)

3.2.2 Pyrähdys

Scrumissa kehitystyö tapahtuu säännöllisissä, toistuvissa työsykleissä. Näitä työsyklejä kutsutaan pyrähdyksiksi (*sprint*) tai iteraatioiksi. Yhden pyrähdyksen pituus voi vaihdella kahdesta viikosta neljään viikkoon. Tiimi voi itse päättää, minkä pituisia pyrähdyksiä projektissa käytetään. Yhden projektin sisällä kaikki pyrähdykset kuitenkin pyritään pitämään samanmittaisina.

Jokainen pyrähdys aloitetaan pyrähdyksen suunnittelukokouksella (*sprint planning meeting*), jossa tuotteen omistaja esittelee priorisoidun tuotteen kehitysjonon. Tuotteen omistaja ja Scrum-tiimi päättää keskenään, mitä ominaisuuksia työstetään aina kunkin pyrähdyksen aikana. Nämä ominaisuudet siirretään tuotteen kehitysjonosta pyrähdyksen tehtävälistaan. Näistä päätöksistä tulee samalla pyrähdyksen tavoitepäämäärä. Tiimi saa käyttää niissä olosuhteissa kaikkia mahdollisia keinoja, jotta he pääsevät pyrähdyksen päämäärään. Pyrähdyksen tehtävälistaan ei voida lisätä tehtäviä enää pyrähdyksen aikana. Ainoat muutokset, joita siihen voidaan tehdä, on jättää tarvittaessa joku tehtävä seuraavaan pyrähdykseen.

Pyrähdyksen aikana tiimi pitää joka päivä päivittäisen Scrum-palaverin (*daily Scrum*). Päivittäinen Scrum-palaveri on aikarajoitettu ja sen aikana käydään läpi vain tiettyjä asioita, kuten missä mennään, onko ongelmia, mitä tehdään seuraavaksi.

Scrum-tiimiä ei saa kukaan häiritä kesken pyrähdyksen. Kaikkien viestien on mentävä vain Scrum-mestarille ja tuotteen omistajalle, jotka päättävät, otetaanko asiat esille seuraavassa pyrähdyksessä.

Jokainen pyrähdys päätetään pyrähdyksen katselmointiin (*sprint review meeting*). Pyrähdyksen katselmoinnissa tiimi esittelee aikaansaannoksensa tuotteen omistajalle. Katselmoinnin aikana tuotteen omistaja päättää, onko tiimin tekemä työ läpäissyt hyväksymiskriteerit.

Tiimi kokoontuu yleensä myös pyrähdyksen retrospektiivikokoukseen (*sprint retrospective meeting*), jossa he keskustelevat pyrähdyksen aikana eteen tulleista ongelmista, kuinka ne on ratkottu, mitkä asiat toimivat ja mitkä eivät sekä kuinka tiimityöskentelyä voisi parantaa.

Näitä pyrähdyksiä toistetaan peräkkäin niin monta, kunnes tuotteen kehitys on saatu päätökseen.

3.2.3 Pyrähdyksen suunnittelukokous

Scrum-tiimi kokoontuu Scrum-mestarin ja tuotteen omistajan kanssa suunnittelemaan jokaista pyrähdystä. Pyrähdyksen suunnittelukokous koostuu oikeastaan kahdesta peräkkäisestä kokouksesta. Ensimmäisessä kokouksessa tiimi tapaa tuotteen omistajan, johdon ja käyttäjät selvittääkseen mitä toiminnallisuuksia tuotetaan seuraavan pyrähdyksen aikana. Toisessa kokouksessa tiimi selvittää keskenään, kuinka ne tuottavat nämä halutut toiminnallisuudet tuotteen lisäykset pyrähdyksen aikana. Suunnittelukokouksen aikana tuotetaan tuotteen kehitysjono (*product backlog*). (Schwaber & Beedle 2002.)

Kuva 3.2. Pyrähdyksen suunnittelukokous

Kokouksen aikana tuotteen omistaja kertoo, miten hän on priorisoinut jäljellä olevat ominaisuudet. Tiimin vastuulla on tiedustella tarpeeksi lisätietoja aiheesta, jotta he voivat tehdä korkeantason käyttäjätarinan (*user story*) tuotteen kehi-

tysjonolle. Lisäksi heidän pitää osata pilkkoa haluttu ominaisuus tarpeeksi pieniin tehtäviin, jotka sijoitetaan pyrähdyksen kehitysjonoon. (Cohn 2011.)

Pyrähdyksen suunnittelukokouksen lopputuloksena syntyy pyrähdyksen maali (*sprint goal*) ja pyrähdyksen kehitysjono. Pyrähdyksen maali on lyhyt, yhden – kahden lauseen mittainen, kuvailu siitä, mitä tiimin on tarkoitus tehdä seuraavan pyrähdyksen aikana. Scrum-mestari ja tuotteen omistaja kirjoittavat tämän usein yhteistyössä. Pyrähdyksen kehitysjono on toinen kokouksen lopputulos. Kehitysjono on lista tuotteen kehitysjonon aiheista, jotka tiimi on sitoutunut tuottamaan sekä lista niistä tehtävistä, jotka on välttämättömiä tehdä. Jokainen tehtävä on yleensä myös aikataulutettu. (Cohn 2011.)

Tärkeä huomio on, että tiimi itse päättää, kuinka paljon työtä se pystyy tulevan pyrähdyksen aikana tekemään. Tuotteen omistaja ei saisi päästä vaikuttamaan tähän. (Cohn 2011.)

3.2.4 Pyrähdyksen katselmointikokous

Pyrähdyksen katselmointikokous (*sprint review meeting*) pidetään jokaisen pyrähdyksen päätteeksi. Kokouksen aikana Scrum-tiimi esittelee pyrähdyksen aikaansaannokset. Tyypillisesti esitetään demonstraatio tuotteen uudesta ominaisuudesta. (Cohn 2011.)

Pyrähdyksen katselmointikokous on usein hyvin epävirallinen. Kokouksessa ei saa Scrumin sääntöjen mukaan esittää esimerkiksi Power Point -esityksiä, eikä kokousta saa valmistella kahta tuntia enempää. Katselmointikokouksen ei tulisi olla tiiminjäsenille häiriöksi eikä sen pitäisi aiheuttaa ylimääräisiä paineita, vaan kokouksen tulisi olla hyvin luonnollinen päätös meneillään olevalle pyrähdykselle. (Cohn 2011.)

Osallistujina katselmointikokouksessa ovat yleensä tuotteen omistaja, Scrum-mestari ja Scrum-tiimi sekä yrityksen johto, asiakas ja kehittäjiä muista projekteista (Cohn 2011.).

3.2.5 Pyrähdyksen retrospektiivi

Pyrähdyksen retrospektiivissä on tarkoitus käydä läpi, mitä edellisen pyrähdyksen aikana opittiin ja mitä olisi voitu tehdä toisin. Tarkoituksena on kerätä tietotaitoa koko tiimille ja pyrkiä jatkuvasti parantamaan tiimin työskentelyä.

Retrospektiiviin osallistuu koko tiimi, mukaan lukien Scrum-mestari ja tuotteen omistaja. Hyvin usein tämä pidetään heti pyrähdyksen katselmointikokouksen jälkeen. Retrospektiivin kesto on noin yhdestä tunnista pariin tuntiin. Tämä on yleensä riittävän pituinen aika, jotta saadaan käytyä kaikki asiat läpi.

Palaveri voidaan esimerkiksi toteuttaa niin, että Scrum-mestari kyselee tiiminjäseniltä parannusideoita ja tästä voidaan jatkaa aivoriihityyppisesti.

3.2.6 Päivittäinen Scrum-palaveri

Päivittäinen Scrum-palaveri järjestetään nimensä mukaisesti joka päivä. Tyypillisesti palaveri pidetään myös jokainen kerta samassa paikassa ja samaan aikaan. Ideaalitapauksessa palaveri on aamuisin ennen varsinaista töiden aloittamista. Palaverin kesto on rajattu tarkasti viiteentoista (15) minuuttiin, eikä tästä saisi poiketa. Tämä auttaa pitämään keskustelun vilkkaana ja tehokkaana, eikä turhia asioita käsitellä. Palaveriin saa osallistua kuka tahansa, mutta sen aikana vai ne, jotka ovat omistautuneet tälle projektille, saavat puhua. Jokaisen tiiminjäsenen tulee osallistua tähän palaveriin, myös Scrum-mestarin ja tuotteen omistajan tulisi olla mukana.

Päivittäisessä Scrum-palaverissa ei saa ratkoa varsinaisia ongelmia, vaan nämä tehdään muulla ajalla. Palaverin aikana jokainen tiiminjäsen vastaa kolmeen kysymykseen:

- Mitä teit eilen?
- Mitä teet tänään?
- Onko esteitä, jotka estävät sinua työskentelemästä?

Näiden kysymysten avulla jokainen saa kiteytettyä eilisen tehtävät ja pystyy jäsentelemään niin itselleen kuin muille tämän päivän tehtävänsä. Tämä helpottaa tiiminjäseniä ymmärtämään paremmin, mitä tehtäviä on vielä tämän pyräh-

dyksen aikana suoritettava. Palaveri ei ole tilannepalaveri, jossa esimies tiedustelee aikataulun pitävyyttä, vaan lähinnä se on palaveri, jossa tiiminjäsenet tekevät lupauksia toisillensa.

4 Testaus

Glenford Myersin sanoin testaus on prosessi, jossa pyritään kaikin tavoin rikkomaan ohjelma ja löytää mahdolliset virheet.

> *”Testing is the process of executing a program with the intent of finding errors” (Myers, 2004).*

Ohjelmistotestauksen tarkoituksena on siis parantaa järjestelmien laatua ja estää mahdollisia virhetilanteita järjestelmän käytön aikana. Testaus kuuluu tärkeänä osana ohjelmistonkehitykseen. Virheiden korjaus jälkikäteen lisää työmäärää hyvinkin paljon. Testauksen laatuun vaikuttaa paljon testaajan kokemus. Kokenut testaaja osaa jo ennalta arvata, mitä kannattaa testata ja milloin. Kuitenkaan täysin kattavaa ohjelmistotestausta ei käytännössä koskaan pystytä tekemään. Tähän vaikuttaa jo projektin aikataulu ja budjetti. Järjestelmällisellä ja hyvin suunnitellulla testauksella päästään usein korkeaan testausprosenttiin. Mitä aiemmin mahdolliset virheet järjestelmästä löydetään, sen edullisemmaksi sen korjaus tulee.

4.1 Testauksen suunnittelu ja dokumentointi

Testauksen suunnittelu tulisi aloittaa jo heti projektin alkuvaiheessa, mielellään samanaikaisesti vaatimusmäärittelyn teon kanssa. Näin saataisiin varmistettua, että järjestelmästä testataan kaikki tarvittavat toiminnallisuudet. Testauksen suunnitteluun kuuluu testitapausten luominen. Mahdollisia testitapauksia voidaan etsiä vaatimusmäärittelystä. Testitapausten suunnittelussa kannattaa olla järjestelmällinen ja sen tulee pohjautua asiakkaan vaatimuksiin. (Jäntti 2003.)

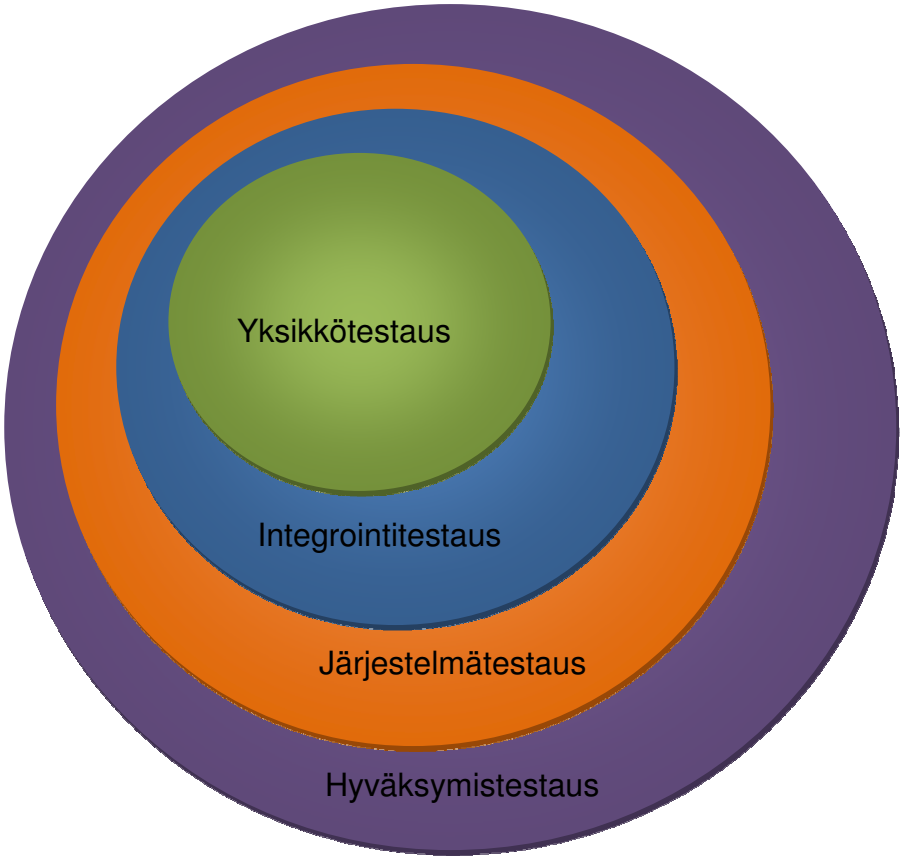
Erilaisia testausdokumentteja ovat muun muassa testaussuunnitelma, testitapausten määrittely, testauslistat ja testausraportti (Jäntti 2003).

Testaussuunnitelmassa on kuvattu ainakin seuraavat asiat: testattavat kohteet, testattavat ominaisuudet, lähestymistapa, läpäisy- ja hylkäisykriteerit, tuotettavat dokumentit, testaustehtävät, testausympäristö ja vastuut sekä aikataulu (Jäntti 2003).

Jos testauksen aikana löydetään virhe, tulisi luoda virheraportti. Raporttiin kuvataan, millainen virhe oli ja miten virhe saatiin aikaiseksi. Tämän raportin avulla voidaan korjauksen jälkeen testata uudelleen saadaanko sama virhe aikaiseksi. (Jäntti 2003.)

4.2 Testauksen vaiheet

Testaukseen kuuluu useita eri vaiheita. Tärkeimpiä niistä ovat yksikkö-, integrointi-, järjestelmä- ja hyväksymistestaus. Kuvassa 4.1 on kuvattu, kuinka nämä testausvaiheet sijoittuvat toisiinsa. Yksikkötestaus on nimensä mukaan pienimmän osan eli yhden komponentin testausta. Kun komponentteja yhdistetään toisiinsa, yhdistämisen aikana tehdään integrointitestausta. Lopulta, kun koko järjestelmä on valmiina, tehdään järjestelmätestaus. Ennen projektin päätöstä, asiakas tekee hyväksymistestauksen järjestelmälle. Tämän jälkeen voidaan antaa lupa julkaista järjestelmä sille tarkoitettuun käyttöön.



Kuva 4.1 Testaustasot

4.2.1 Yksikkötestaus

Yksittäisen ohjelmistokomponentin testausta kutsutaan nimellä yksikkötestaus (*unit testing*). Yksikkötestauksen suorittaa yleensä kyseisen komponentin kehittäjä. Yksikkötestauksesta voidaan myös käyttää nimityksiä moduulitestaus (*module testing*) tai komponenttitestaus (*component testing*). Yksikkötestaus pyritään usein automatisoimaan. Automatisoinnin hyöty on se, että kuka tahansa voi ajaa kyseiset testit, ilman, että hänen tarvitsee suoranaisesti tietää, mitä testataan. Automatisoidut testit voidaan myös suorittaa missä tahansa vaiheessa, kun tarve niin vaatii. (Sainio 2009.)

Yksikkötestaus keskittyy lähinnä ohjelmakoodin testaukseen. Testaamalla pyritään löytämään virheet kooditasolta.

Huolellinen yksikkötestaus on hyvin tärkeää. Jokainen komponentti on testattava kunnolla ennen kuin niitä yhdistellään toisiinsa ja sitä kautta isompiin kokonaisuuksiin. Valmiissa järjestelmässä voi olla monia tuhansia komponentteja ja kaikkien komponenttien huolellinen testaus on hyvin hankalaa. (Sainio 2009.)

4.2.2 Integrointitestaus

Integrointitestaus tehdään silloin, kun komponentteja yhdistetään eli integroidaan yhteen. Tällöin testauksen kohteena on eri komponenttien yhteensopivuus. Tässä vaiheessa ei varsinaisesti etsitä enää mahdollisia virheitä vaan lähinnä eri komponenttien yhteensopimattomuus vikoja. (Sainio 2009.)

4.2.3 Järjestelmätestaus

Järjestelmätestauksen tarkoituksena on testata, toimiiko valmis järjestelmä, kuten määrittelyssä on vaadittu. Järjestelmätestauksen osa-alueita ovat muun muassa toiminnallisuus-, volyymi-, kuormitus-, käytettävyys-, tietoturva-, suorituskyky-, yhteensopivuus-, luotettavuus-, toipuvuus- ja ylläpidettävyystestaus. (Sainio 2009.)

4.2.4 Hyväksymistestaus

Hyväksymistestaus sisältää osittain samoja testaustyyppejä kuin järjestelmätestaus, mutta hyväksymistestauksessa keskitytään vielä enemmän siihen, toimiiko järjestelmä loppukäyttäjän näkökulmasta niin kuin on määritelty. Hyväksymistestauksen suorittaa useimmiten asiakas.

4.2.5 Regressiotestaus

Regressiotestauksen (*regression testing*) tarkoituksena on testata, toimiiko jo aiemmin testattu järjestelmänosio vielä oikealla tavalla mahdollisten muutosten jälkeenkin. Regressiotestausta käytetään hyvin paljon ketterissä menetelmissä. Näitä testejä automatisoidaan tarpeen mukaan ja niitä suoritetaan mahdollisuuksien mukaan päivittäin yksikkötestausten ohella. Regressiotestauksella pyritään varmistamaan, etteivät muutokset ole aiheuttaneet tai paljastaneet vikoja niissä osioissa, joita ei ole muutettu. Erilaisia muutoksia voivat olla koodin muuttaminen, lisääminen tai poistaminen. Regressiotestausta voidaan myös suorittaa, jos järjestelmän käyttöympäristö on muuttunut. (Sainio 2009.)

Regressiotestauksessa käytetään uudelleen vanhoja testitapauksia, jotka on todennettu toimiviksi edellisillä testauskerroilla.

4.3 Tutkiva testaus

”Exploratory testing is simultaneous learning, test design, and test execution” (Bach 2003).

Tutkiva testaus (*exploratory testing*) on siis eräänlaista kokemuksiin perustuvaa testaamista. Tutkivassa testauksessa testaaja ei käytännössä suunnittele etukäteen, mitä ja miten hän testaa, vaan testaaja suunnittelee ja testaa samanaikaisesti. Testaustekniikat muuttuvat yleensä testauksen aikana sitä mukaa, kun testaaja oppii ymmärtämään testattavan järjestelmän tarkoitusta. (Halme 2004; Sainio 2009.)

Tutkiva testaus on hyvin hyödyllistä silloin, kun tutkittavasta kohteesta tiedetään etukäteen hyvin vähän. Tutkivaa testausta käytetään usein nimenomaan ketterissä ohjelmistontuotantomenetelmissä. Tutkivassa testauksessa testaajan ko-

kemus yleensä määrittelee, kuinka hyvin testaus onnistuu. Kokemattomalla testaajalla voi olla paljonkin hankaluuksia osata testata oikeita asioita, jolloin testikattavuus saattaa jäädä hyvin pieneksi. (Halme 2004.)

5 Tutkimus

Tutkimus toteutettiin haastattelemalla erään yrityksen työntekijöitä. Haastateltavat oli valittu niin, että jokaisesta Scrum-roolista olisi vähintään yksi henkilö kertomassa työstään. Valitettavasti ainuttakaan tuotteen omistajaa ei kuitenkaan löytynyt. Haastateltavia oli yhteensä seitsemän henkilöä.

Haastateltaville esitettiin pääsääntöisesti samanlaisen kysymykset ja he vastasivat niihin oman tietämyksensä mukaan. Haastattelukysymykset ovat liitteessä 1: Scrum haastattelukysymykset.

Haastattelut tehtiin yrityksen tiloissa kahden päivän aikana. Yksi haastattelu kesti keskimäärin 40 minuuttia. Haastattelut otettiin sanelulaitteelle, jotta itse haastattelutilanteessa pystyttiin keskittymään aihealueeseen. Kaikki haastateltavat osallistuivat vapaaehtoisesti tilanteeseen, joten itse haastattelut sujuivat mutkattomasti ja haastateltavat ottivat hyvin osaa keskusteluun.

Haastatteluhenkilöt eivät tienneet varsinaisia kysymyksiä etukäteen, mutta heille oli kerrottu taustaa opinnäytetyöstä, joten he osasivat valmistautua etukäteen mahdollisiin kysymyksiin. Haastattelun tarkoituksena olikin saada selville, minkälaisia omakohtaisia kokemuksia ja mielipiteitä henkilöillä itsellään on ketteristä ohjelmistotuotantomenetelmistä. Kaikki haastateltavat olivat käyttäneet projekteissaan Scrumia tai vastaavanlaista toista ohjelmistontuotantomenetelmää keskimäärin kaksi – kolme vuotta. Haastateltavat olivat toimineet lähes kaikissa projekteissa samoissa rooleissa. Ennen Scrumia he olivat käyttäneet pääsääntöisesti vain perinteistä vesiputousmallia sekä inkrementaalista ohjelmistokehitystä.

Tarvittaessa olisi voitu myös pitää toinenkin haastattelutilaisuus, mutta tähän ei koettu tarvetta. Jokainen haastateltava antoi suostumuksensa uudelle tilaisuudelle.

6 Tutkimuksen tulokset

Haastattelun lopputulokset olivat hyvin samankaltaisia riippumatta henkilön roolista Scrum-tiimissä.

Scrum soveltuu parhaiten projekteihin, joissa kehitetään uutta muutosherkkää ohjelmistoa. Toisaalta Scrum soveltuu lähes kaikkiin projekteihin, koska sitä on helppo muokata tilanteen mukaan. Scrum on lähinnä vain kehitysraamit, jonka osia voidaan käyttää tarpeen mukaan. Siitä voidaan jättää pois projektille turhat osiot tai ottaa lisää toiminnallisuuksia toisista ohjelmistotuotantokehitelmistä. Etenkin kehittäjät pitävät Scrumista, koska se on hyvinkin muuntuvainen ja sen avulla voidaan jättää turhat työt pois päivärutiineista.

Scrum-tiimin tulee olla itseohjautuva (*cross-functional team*), ja siihen kuuluu ideaalitapauksessa 5 – 8 henkilöä. Tiimi kostuu Scrum-mestarista, ainakin yhdestä kokeneesta testaajasta ja kehittäjistä. Tarvittaessa tiimissä olisi hyvä olla myös yksi arkkitehtuuri tai tekninen asiantuntija (*technical lead*) -tyyppinen henkilö, joka on vastuussa teknisestä kokonaisuudesta. Scrum-tiimihenkilön hyviin ominaisuuksiin kuuluu etenkin hyvät kommunikointitaidot. Nykypäivänä ei tiimihenkilö voi olla sisäänpäin sulkeutunut yksin työtä tekevä henkilö. Tiimihenkilön pitäisi myös olla avoin uusille asioille ja muutoksille, muutosvastarinta heikentää suuresti työn etenemistä. Tiimihenkilöille tärkeä ominaisuus on myös vastuunottaminen omasta työstään. Scrum-tiimin tulisi tehdä töistä samassa tilassa, jolloin päivittäinen kommunikointi on helpompaa tiimin välillä.

Muuntautuminen itseohjautuvaan tiimiin ei aina kaikille henkilöille ole helppoa. Lähinnä vaikeuksia aiheuttaa vastuunottaminen omasta työstä. Myös muuntautuminen perinteisestä projektipäälliköstä Scrum-mestariksi saattaa olla hankalaa. Ongelmia tässä tapauksessa tuottaa eniten projektipäällikön kyky luopua määräämisoikeudestaan. Scrum-mestari ei käytännössä enää määrää tiimihenkilöilleen heidän työtehtäviä, vaan tarkoituksena on, että tiimi sopii keskenään työn jakamisista.

Päivittäisiin työrutiineihin kuuluvaa Scrum-palaveria (*Daily Scrum*) pidetään erittäin tärkeänä. Palaverin aikana käydään läpi kolme asiaa: Mitä teit eilen, mitä teet tänään, onko joku este, joka estää työskentelemästä? Scrum-palaverin

kesto ei saisi ylittää 15 minuuttia ja agendassa tulisi pysyä. Päivittäinen Scrum-palaveri auttaa tiimiä tekemään tiimityötä. Palaverien myötä mahdolliset ongelmat saadaan ratkaistua nopeammin ja ne auttavat tiimin jäseniä jäsentelemään päivän työtehtävät myös itselleen paremmin. Yksi Scrumin merkittävimmistä hyödyistä on nimenomaan nämä palaverikäytännöt. Ne esimerkiksi pakottavat jatkuvaan parantamiseen. Palaverista voi harvinaisissa tapauksissa aiheutua myös haittaa, jos tiimihenkilöt eivät halua tai osaa tuoda esille hyödyllistä tietoa tai mahdollisia ongelmatilanteita. Muita palavereja, joita pidetään projektin aikana, ovat pyrähdyksen suunnittelukokous (*sprint preview meeting*) ja retrospektiivipalaveri (*retrospektive*). Keskimäärin palavereja pidetään projektin aikana kohtuullisen vähän ja niiden määrä vaihtelee projektin tarpeen mukaan.

Dokumentoinnin tarpeellisuudesta voidaan olla montaa mieltä. Kuitenkin tänä päivän asiakas määrittelee hyvin pitkälle, mitä dokumentteja tuotetaan. Jossain tapauksissa saatetaan jopa dokumentoida liian vähän, mikä voi aiheuttaa hankaluuksia ja ongelmatilanteita jatkokehityksissä tai ylläpitovaiheessa. Yleensä kuitenkin kaikissa projekteissa vähintään ohjelmistokoodin kommentoidaan. Lisäksi tuotetaan karkealla tasolla oleva projektisuunnitelma (*project plan*) sekä erilaiset budjetti- ja resursointidokumentit. Jos projektin aikana pidetään retrospektiiveja, niin myös niiden sisältö dokumentoidaan jossain määrin. Perinteisiä, suuria toiminnallisia määrittelyitä ei enää tuoteta, ainakaan tavalla, jolla se on tehty esimerkiksi vesiputousmallissa. Tuota vanhaa tapaa ei pidetä edes kovinkaan hyödyllisenä, koska nykypäivän vaatimuksiin tulee herkästi muutoksia. Hyvänä työkaluna dokumentointiin pidetään wiki-tyyppisiä asiakirjoja, joita jokainen tiiminjäsen voi tarvittaessa muokata ja päivittää.

Suoranaisesti Scrum ei välttämättä ole yhtään sen tehokkaampi tapa työskennellä, jos sitä ei osata käyttää oikein. Se on kuitenkin monella tapaa helpompi kuin esimerkiksi perinteinen vesiputousmalli, koska Scrumin avulla on mahdollista reagoida nopeammin erilaisiin muutoksiin. Se auttaa myös työskentelymotivaatioon, koska ideana on pilkkoa työmäärät sopivan mittaisiin pyrähdyksiin. Pienemmät kokonaisuudet ovat yleensä helpompia hahmottaa sekä suunnitella. Scrumin koetaan myös sitouttavan tiimiä paremmin yhteen, kun he pääsevät itse vaikuttamaan työtehtäviinsä.

Oikeaoppinen Scrum-mestari huolehtii Scrumin fasilitoinnista. Toisin sanoen hän huolehtii, että tiimillä on Scrumin peruskäytännöt kunnossa. Hän myös rohkaisee tiimiä toimimaan toivotulla tavalla ja valmentaa heitä. Lisäksi motivaation nostattaminen kuuluu Scrum-mestarin työhön. Hän huolehtii, että tiimillä on kaikki tarvittavat työkalut työn toteuttamiseen ja poistaa ongelmat mahdollisimman pian.

Suurin ero perinteisen projektipäällikön ja Scrum-mestarin välillä on tehtävien jakaminen. Projektipäällikkö määrää työtehtävät tiiminjäsenille, kun taas Scrum-mestari ei puutu tehtävien jakamiseen vaan se hoidetaan tiiminjäsenten kesken. Kun tiiminjäsenet saavat itse jakaa työtehtävänsä parhaan näkemyksensä mukaan, se edesauttaa työmotivaatiota ja tiimi saa hyvänolon tunteen, kun he ovat suorittaneet itse asettamansa tavoitteet.

Hyvä Scrum-mestari on avoin ja luotettava. Hänen pitää pystyä luottamaan tiimiin. Hän on myös kärsivällinen ja pystyy hahmottamaan työn kokonaisuuden. Scrum-mestari toimii tiimin ja asiakkaan rajapinnassa ja hänen on ymmärrettävä, mitä asiakas haluaa. Tärkein ominaisuus Scrum-mestarille on, että hän osaa kuunnella sekä tiimiä että asiakasta. Hänen tulisi olla myös muuntautumiskykyinen ja halukas oppimaan uutta sekä hyväksymään muiden mielipiteitä. Myös neuvottelutaidot ovat tärkeässä asemassa. Scrum-mestarin pitäisi olla osa tiimiä.

Scrum-projekteissa testaaminen on jaettu usealle henkilölle. Jokainen kehittäjä tekee vähintään yksikkö- ja moduulitestaukset (*unit, module*) omille tuotoksilleen. Erilliset testaajat tekevät pääsääntöisesti toiminnallista testaamista. Testaajat käyttävät nykyisin paljon tutkivaa testausta (*exploratory testing*). Testisuunnitelmat pohjautuvat monesti käyttäjätarinoihin (*user story*) ja tuotteen työlistaan. Testaajat ovat mukana käyttäjätarinoiden luonnissa, jolloin hän pystyy paremmin suunnittelemaan testitapaukset etukäteen. Testaaja ja kehittäjä sopivat keskenään, mitä osia voidaan testata ja milloin testaus voidaan aloittaa. Ideaali tilanne on, että mahdolliset virheet löydettäisiin aikaisessa vaiheessa ja saataisiin näin ollen korjattua jo saman pyrähdyksen aikana, ennen kuin tuotokset viedään asiakkaalle. Varsinaisia testaustuloksia ei juurikaan dokumentoida, jos asiakas ei tällaisia dokumentteja ole tilannut. Hyvin usein testihetket painot-

tuvat pyrähdyksen loppupuolelle ja valitettavaa on, että hyvin harvoin testaajan työkuorma saataisiin tasapainotettua koko pyrähdyksen ajalle.

Testauksia pyritään automatisoimaan tarvittavilta osin, mutta usein kannattaa miettiä, mitä kannattaa automatisoida ja mitä ei. Yksikkö- ja moduulitestaukset on pääsääntöisesti aina automatisoitu.

Testaustyökaluja käytetään jokin verran. Lähinnä niiden avulla seurataan testauskattavuutta, jolle on käytännössä aina määritelty tarkka prosenttiluku. Myös automatisointiin on omat työkalut.

Testien määrä projektissa vaihtelee asiakkaan toiveiden mukaan. Aina ei kuitenkaan saada tehtyä tarpeeksi testauksia johtuen budjetti- ja resurssipuutosten vuoksi. Testausta pidetäänkin lähinnä riskien hallintana. Kehittäjän ja testaajan näkökulmasta katsottuna yksi suuri toive olisi saada asiakkaat ymmärtämään testauksen tarpeellisuus.

Varsinaisia yhdenmukaisia käytäntöjä testauksen tekemiseen on hyvin vähän, mutta toki kaikilta löytyy jonkinlainen perusnäkemys ja yhteinen tapa hoitaa asioita. Toisaalta, kun ei ole luotu liian tiukkoja raameja testauksille, saadaan helposti tuotettua samalle ohjelmiston osiolle erilaisia testitapauksia. Tämän avulla saadaan helposti uusia näkökulmia ja tuloksia sekä testaukset tulee tehtyä paljon kattavammin.

Jokaisessa projektitiimissä olisi hyvä olla yksi henkilö, joka vastaa arkkitehtuurista. Mutta kuten Scrumiin yleensä kuuluu, ei arkkitehtuuriakaan suunnitella täysin valmiiksi projektin alussa. Korkean tason arkkitehtuuri tulisi olla olemassa, mutta tarkemmat yksityiskohdat suunnitellaan samoissa pyrähdyksissä kuin itse projekti toteutetaan. Monesti yritys todistaa asiakkaalle projektin toteutuksen onnistumisen niin sanotulla POC:illa (*Proof Of Concept*). POC tarkoittaa toteutuksen demoversiota, jonka avulla asiakas pääsee jo tietyin määrin tarkastelemaan toteutuksen mahdollista lopputulosta.

Arkkitehtuuridokumentteihin pätee sama sääntö kuin muuhunkin dokumentointiin. Asiakas päättää, mitä dokumentoidaan. Perinteisiä dokumentteja tehdään enää harvoin, jo niiden päivittämisvaikeuksien vuoksi. Nykyisin suositaankin

enemmän wiki-tyyppisiä kokoelmia. Näitä dokumentteja on kenen tahansa tiiminjäsenen helppo päivittää ja pitää ajan tasalla.

7 Yhteenveto

Mikään ohjelmistontuotantomenetelmä ei takaa varsinaisen projektin onnistumista, mutta ei myöskään aiheuta suoranaisesti sen epäonnistumista. Eri menetelmät sopivat erityylisiin projekteihin, mutta enemmän käytettävän menetelmän valintaan vaikuttavat projektihenkilöiden taidot ja osaamiset käyttää valittua menetelmää.

Nykypäivänä vaatimukset järjestelmän ominaisuuksille saattavat muuttua monta kertaa projektin aikana, joten valitun ohjelmistotuotantomenetelmän tulee pystyä vastaamaan näihin muutoksiin. Kankeat perinteiset menetelmät harvoin pystyvät taipumaan moniin muutoksiin tai ainakin nämä muutokset saattavat aiheuttaa tällöin paljon ylimääräistä työtä. Ketterät menetelmät on suunniteltu nimenomaan vastaamaan näihin muutoksiin. Ketterien menetelmien pääajatus onkin suunnitella tulevaa järjestelmää jatkuvalla prosessilla, joka on käynnissä koko projektin ajan. Muutokset järjestelmään eivät näin ollen ole käytännössä varsinaisia muutoksia, koska muutoksen alkuperäistä ominaisuutta ei välttämättä ole vielä edes suunniteltu, toisin sanoen, siihen ei ole vielä käytetty aikaa eikä rahaa. Tämä helpottaa muutoksien vastaanottamista sekä projektitiimin että asiakkaan puolelta.

Käytännössä Scrum on vain eräänlainen kehys ohjelmistoprojektille. Scrum ei määrittele, miten projektin varsinainen toteutus tehdään vaan antaa enemmän suuntaa, kuinka projektitiimin tulisi kommunikoida keskenään ja asiakkaan kanssa, jotta projekti saataisiin mahdollisimman onnistuneesti vietyä läpi.

Scrum on hyvin skaalautuva. Siihen voidaan ottaa toimintatapoja myös muista ohjelmistontuotantomenetelmistä, jos se koetaan tarpeelliseksi. Siitä voidaan myös jättää turhat osiot pois, jos meneillään oleva projekti ei niitä vaadi. Scrum on tehokas menetelmä, mutta se ei yksinään takaa projektin onnistumista, vaan sitä on osattava käyttää oikein.

Yksi Scrumin, ja muiden ketterien menetelmien, pääperiaate on niin sanottujen roskien poistaminen. Monesti tämä tarkoittaa, ettei turhia ja ylimääräisiä dokumentteja työstetä vain sen takia, että niin on määrätty. Dokumenttien kirjoittaminen ja päivittäminen vie runsaasti aikaa ja samalla myös rahaa. Tokikaan kaik-

kia dokumentteja ei kannata tai edes voi jättää tekemättä, mutta on järkevää miettiä, onko dokumentista hyötyä myös tulevaisuudessa. Myös tekniikoita, joilla dokumentit tuotetaan, kannattaa miettiä. Perinteinen Word-dokumentti on monesti hankala päivittää, joten dokumentin tuottamiseen voi esimerkiksi käyttää wiki-pohjaisia kirjastoja. Tällöin kuka tahansa saa päivitettyä haluamaansa dokumenttia.

Testaus kuuluu tänä päivänä olennaisena osana ohjelmistontuotantoon. Ei ole projektia, järjestelmää tai ohjelmaa, jossa voitaisiin jättää testaaminen kokonaan pois. Vaikka testaamiseen menee aikaa, niin se myös säästää hyvin paljon rahaa. Testauksen avulla voidaan löytää mahdolliset virheet jo varhaisessa vaiheessa, jolloin myös korjaus on yleensä helpompaa ja etenkin edullisempaa. Tärkeää olisi myös saada asiakas ymmärtämään testauksen tarpeellisuus, koska asiakas viimein määrittelee, kuinka paljon rahaa ja aikaa mihinkin toimintoon käytetään.

Taidokas ja kokenut projektitiimi saa vietyä projektin onnistuneesti läpi riippumatta käytettävistä menetelmistä. Oikein valittu menetelmä kuitenkin helpottaa sen toteutumista.

Kuvat

Lähteet

Agile Alliance. 2001.Agile Manifesto.
http://www.agilealliance.com
(Luettu14.7.2011)

Ahonen, M. 2010. Tapaustutkimus: Soveltuuko Scrum vesiputousmallin korvaajaksi yrityksen sovellus kehitysprojekteihin?
Aalto-yliopisto, diplomityö.

Ambler, S. W. 2011. Introduction to Test Driven Development (TDD).
http://www.agiledata.org/essays/tdd.html
(Luettu 14.7.2011)

Bach, J. 2003. Exploratory Testing Explained.
http://www.satisfice.com/articles/et-article.pdf
(Luettu12.3.2012)

Beck, K. 2004, Extreme Programming,
www.extremeprogramming.org
(Luettu15.4.2011)

Cohn, M. 2010.Succeeding with Agile: Software Development Using Scrum.
Addison-Wesley.

Cohn, M. 2011. Scrum Training, Agile Training from Scrum Master Mike Cohn.
http://www.mountaingoatsoftware.com/
(Luettu 13.4.2011)

Crispin, L. & Gregory, J. 2009. Agile Testing: A Practical Guide for Testers and Agile Teams.
Addison-Wesley.

Haikala & Märijärvi. 2004. Ohjelmistotuotanto.
Talentum Media Oy

Halme, E. 2004. Tutkiva testaus hyväksymistestauksen menetelmänä.
http://www.soberit.hut.fi/T-76.5650/Spring_2004/Papers/E.Halme_76650_final.pdf
(Luettu 18.5.2012)

Huttunen, J. 2006. Ketterän ohjelmistokehitysmenetelmän määrittely, vertailu ja käyttäjäkysely.
Teknillinen korkeakoulu, diplomityö.

IEEE. 1990. IEEE Std 610.12-1990: ”IEEE Standars Glossary of Software Engineering Terminology.”
Los Alamitos, CA: IEEE Computer Society Press.

Jäntti, M. 2003. Testitapausten suunnittelu UML-mallinnuksen avulla.
Kuopion yliopisto, Pro gradu –tutkielma.

Kainulainen, A. 2008. Agile-menetelmät.
Jamk. Opinnäytetyö.

Koskela, L. 2007. Scrum: Ketterien menetelmien markkinajohtaja.
http://ttlry-fi-bin.directo.fi/@Bin/9115502d22c373e22606eccc5bf9b3c8/1337343170/application/pdf/11062393/04_ScrumMarketLeaderOfAgileMethods_handout_LasseKoskela.pdf
(Luettu 14.5.2010)

Kosonen, S. 2005. Ohjelmoinnin opetus Extreme Programming –hengessä.
Jyväskylän yliopisto, Pro gradu –tutkielma.

Lindberg, H. 2003. Extreme Programming.
Tampereen yliopisto, Pro gradu –tutkielma.

Myers, G., Sandler, C., Badgett, T. & Thomas, T. 2004.The Art of Software Testing. Second edition.
John Wiley & Sons, Inc.

Naur, P. & Randell, B. 1969. Software Engineering: Report on a Conference Sponsored by the NATO Science Committee.
Garmisch, Germany, 7-11 Oct.

Ovaska, P. 2002. Ohjelmistotuotannon osa-alueet.
http://www2.it.lut.fi/kurssit/01-02/010758000/luento2.pdf
(Luettu 18.5.2012)

Poimala, Heikniemi & Blåfield. 2008. Ketterät käytännöt.fi -sivusto.
http://www.ketteratkaytannot.fi/fi-FI/Menetelmat/
(Luettu 14.5.2010)

Pressman, R. 2005, Software engineering a practitioner's approach 6th edition.
McGraw-Hill.

Pyhäjärvi, M. & Pöyhönen, E. 2006. Ohjelmistojen testaus.
http://users.jyu.fi/~kolli/testaus2006/materiaali/Maaret_27102006.pdf
(Luettu 18.5.2012)

Pyykkö, T. 2010. Ohjelmistotestaus siirryttäessä perinteisistä ohjelmistokehitysmenetelmistä Scrumiin.
Jyväskylän yliopisto, Pro gradu –tutkielma.

Pöri, M. 2008. Testaus Scrum-prosessimallissa.
Helsingin yliopisto, Pro gradu –tutkielma.

Sainio, L. 2009. Ohjelmistotestauksen menetelmät ja työvälineet.
Saimaan ammattikorkeakoulu, opinnäytetyön lukumateriaali.

Schwaber, K. 2004. Agile Project Management with Scrum.
Microsoft Professional.

Schwaber, K. Beedle, M., 2002.Agile Software Development with Scrum.
Pearson Education.

Schwaber, K., Sutherland, J. 2009. Scrum.
http://www.scrum.org/storage/scrumguides/Scrum%20Guide%20-%20FI.pdf
(Luettu 27.10.2010)

Shore, J. 2010. The Art of Agile Development: Test-Driven Development.
http://jamesshore.com/Agile-Book/test_driven_development.html
(Luettu 14.7.2011)

Stenberg, A. Ohjelmiston testaus, 2004,
http://www.pori.tut.fi/~stenberg/
(Luettu 18.4.2012)

Vestola, M. 2008. TDD:n edut - tarua vai totta?
Teknillinen korkeakoulu, kandidaatintyö.

Väyrynen, V. 2009. Onnistunut testaus ja ketterät menetelmät.
Haaga-Helia ammattikorkeakoulu, opinnäytetyö.

Liite 1. Scrum-haastattelukysymykset

- Kuinka kauan olet käyttänyt Scrumia, rooli
- Vaihteleeko roolisi eri projekteissa
- Mitä ohjelmistokehitysmenetelmiä olet käyttänyt Scrumin lisäksi?
- Minkälaisiin projekteihin Scrum soveltuu parhaiten?
- Minkälaisia henkilöitä kuuluu hyvään tiimiin ja kuinka monta (ominaisuudet, roolit)
- Oliko helppoa muuntautua itseohjautuvaan tiimiin
- Kuinka paljon käytetään aikaa ”kokouksiin” pyrähdyksen aikana
- Entä dokumentointiin
- Päivittäiset Scrum-palaverit (ovatko tarpeellisia)
- Miten työskentely käytännössä tapahtuu
- Onko SCRUM tehokkaampi tapa kuin perinteinen vesiputousmalli?
- Mitä hyvää Scrumissa on verrattuna perinteisiin ohjelmistotuotantomenetelmiin
- Entä verrattuna muihin ketteriin menetelmiin
- Miksi Scrumia kannattaa käyttää

- Mitä Scrum-mestarin työhön käytännössä kuuluu
- Esteiden poistaminen
- Onko erikseen perinteistä projektipäällikköä (hyötyä / haittaa)
- Mitä eroa on Scrum-mestarilla ja perinteisellä projektipäälliköllä
- Minkälainen henkilö soveltuu parhaiten Scrum-mestariksi (mitä ominaisuuksia vaaditaan)

- Miten testaus on organisoitu (erillinen testaaja, tiimi, kehittäjien rooli, tuotteen omistaja)
- Miten testaukset suunnitellaan etukäteen ja kuinka paljon
- Mitä vastaan testataan (jos ei ole ajan tasalla olevaa järjestelmää kuvaavaa dokumentaatiota)

- Miten testaus tapahtuu (kuinka usein, kuka tekee mitä, dokumentointi)
- Miten testaukset dokumentoidaan / Mitä dokumentoidaan ja kuinka paljon / tarkasti
- Mitä eri testausvaiheita käytetään (yksikkö-, integrointi-, järjestelmä-, hyväksymistestaus)
- Missä vaiheessa testausta suoritetaan (suhteessa pyrähdykseen / suhteessa koko projektiin)
- Automatisoidaanko testauksia
- Tehdäänkö regressiotestausta (edelliselle ohjelmistoversiolle ajettujen testitapausten uudelleen suorittaminen)
- Tutkiva testaus
- Käytetäänkö testaustyökaluja
- Testataanko tarpeeksi (riittääkö aika)
- Onko yhdenmukaista käytäntöä testaukseen

- Kuinka tarkkaan arkkitehtuuri suunnitellaan projektin alussa
- Arkkitehtuurin dokumentointi