Ohjelmistotuotantomenetelmien kehittyminen 1950-luvulta nykypäivään

Lauri Suomalainen

Kandidaatintutkielma
HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Helsinki, 30. tammikuuta 2014

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Matemaattis-luonnontieteellinen	Tietojenkäsittelytieteen laitos

Tekijä — Författare — Author
Lauri Suomalainen

Työn nimi — Arbetets titel — Title

Ohjelmistotuotantomenetelmien kehittyminen1950-luvulta nykypäivään

Oppiaine — Läroämne — Subject
Tietojenkäsittelytiede

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Kandidaatintutkielma	30. tammikuuta 2014	20

Tiivistelmä — Referat — Abstract

Tiivistelmä

Avainsanat — Nyckelord — Keywords
avainsana 1, avainsana 2, avainsana 3

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — Övriga uppgifter — Additional information

Sisältö

1 Johdanto

Tämä kandidaatintutkielma tarkastelee ohjelmistotuotantomenetelmien kehittymistä ohjelmistokehittämisen alkuajoista nykypäivään ja lähitulevaisuuteen. Se käsittelee erilaisia ohjelmistotuotantomenetelmiä kronologisesti. Jokaisen menetelmän kohdalla pyrin vastaamaan seuraaviin kysymyksiin:

- Mistä ohjelmistotuotantomenetelmässä on kyse?
- Miksi sitä käytetään/käytettiin ja mitä hyötyä siitä on/oli?
- Mitkä olivat sen heikkoudet?

Ohjelmistotuotannon eri osa-alueita tarkastellaan tutkielmassa ohjelmistotuotantomenetelmiä määrittävinä piirteinä. Tämä tarkoittaa sitä, että tarkasteltaessa esimerkiksi miten vaatimusmäärittely toteutetaan jossain tietyssä ohjelmistotuotantomenetelmässä, keskitytään prosessin konkreettisen toteutuksen sijasta sen asemaan ja erityispiirteisiin menetelmän kontekstissa.

Ensimmäiset luvut toimivat pohjustuksena tutkielmalle. Luku 2 esittelee lyhyesti keskeisimmät ohjelmistotuotantoon ja tuotantomenetelmiin liittyvät käsitteet ja Luku 3 käsittelee ohjelmistotuotannon alkuaikoja, jolloin tietokoneet itsessään olivat vielä verrattain uusi ilmiö ja suuret ohjelmistot ja niiden tuottaminen oli vähäistä. Alun alkaen laitteiston rajallisuus sitoi myös ohjelmistojen mahdollisuuksia, mutta laitteiston kehittyessä syntyi tarve organisoidummalle ohjelmistojen kehittämiselle. http://steamcommunity.com/id/vlek2903/inventory/ Luku 4 käsittelee perinteistä ohjelmistotuotantomallia eli niin sanottua "vesiputousmallia". Tarkastelen mallin perusperiaatteita, sen nousemista aikansa keskeisimmäksi ohjelmistotuotantomenetelmäksi, sen onnistumisia ja epäonnistumisia sekä sen heikkouksia ja siihen kohdistettua kritiikkiä.

Luku 5 tarkastelee inkrementaalisia ja iteratiivisia ohjelmistotuotantomenetelmiä sekä prototyyppaamista. Tyyppiesimerkkeinä käsittelen RUP:ia eli Rational Unified Processia sekä spiraalimallia, jota voidaan pitää vesiputousmallin kehittyneempänä versiona.

Luvussa 6 otetaan tarkasteluun ketterät ohjelmistotuotantomenetelmät. Vaikka ketterä ohjelmistokehitys on kattokäsitteenä monelle erilaiselle mene-

telmälle kuten XP ja Kanban, käsittelen Scrumia tyyppiesimerkkinä, sillä se on kaikista ketteristä menetelmistä suosituin ja tunnetuin.

Viimeisessä luvussa käsittelen lyhyesti yhteenvetona tämän hetken vallitsevia ohjelmistotuotantomenetelmiä. Tarkastelen myös mahdollista lähitulevaisuuden trendejä ja mahdollisuuksia

2 Peruskäsitteistöä

Tietojenkäsittelytieteeseen ja ohjelmistotuotantoon liittyy monia alakohtaisia käsitteitä. Kaikkien termien merkitykset eivät ole suoraan johdettavissa itse termistä, sillä osa termistöstä on verrattain vapaita käännöksiä ulkomaisista termeistä, ja osa konsepteista on laajoja. Lisäksi joillain termeillä viitataan puhekielessä vapaammin eri asioihin. Esimerkiksi softwarella voidaan puhekielessä viitata niin yksittäiseen tietokoneohjelmaan kuin laajaan ohjelmistoonkin. Tämän takia lyhyt käsitteenmäärittely tullee tarpeeseen.

Software eli ohjelmisto käsittää tietokoneohjelman tai -ohjelmia sekä kaiken niihin liittyvän informaation ja materiaalin kuten tietokannat ja dokumentaation.
Tietokonelaitteisto eli hardware käsittää tietokoneen fyysiset osat kuten prosessorin ja kovalevyn. Laitteistoa tarvitaan ohjelmistojen suorittamiseen, ja laitteisto tarvitsee toimiakseen toimintaohjeet matalan tason tietokoneohjelmina. Käytännössä tietokoneohjelmistot ja -ohjelmat sekä tietokonelaitteisto eivät ole käyttökelpoisia yksinään, vaan kumpaakin tarvitaan toisen järkevään käyttöön.

Termi software engineering, suomeksi ohjelmistotuotanto, alkoi esiintyä kirjallisuudessa 1960-luvun puolivälissä. Termi itsessään on ollut usein keskustelun ja väittelyn kohteena, ja ohjelmistotuotannon kuulumista insinööritieteisiin on kyseenalaistettu. [10, 11, 14] Watts S. Humphrey on määritellyt ohjelmistotuotannon tarkoittavan kurinalaista laadukkaiden ohjelmistojen tuottamista hyödyntäen niin luonnontieteellisiä, matemaattisia kuin insinööritieteidenkin periaatteita ja käytänteitä [12]. IEEE Computer Society määrittelee termin viittaavan kurinalaiseen, systemaattiseen ja arvioitavissa olevaan lähestymistapaan ohjelmistojen tuotannossa, käytössä ja ylläpidossa

[2]. Ilkka Haikala ja Jukka Märijärvi tulkitsevat määrittelyjen tarkoittavan ohjelmistotyötä, jonka tuloksena syntyvät järjestelmät täyttävät käyttäjiensä kohtuulliset toiveet ja odotukset ja tämän lisäksi valmistuvat laadittujen aikataulujen ja kustannusarvioiden puitteissa [11].

Ohjelmistotuotantoon kuuluvat kaikki ohjelmistotuotantoprosessin osa-alueet. Haikala ja Märijärvi [11] määrittelevät ne seuraavasti:

Määrittely sisältää asiakasvaatimusten analyysin ja niistä johdetaan ohjelmistovaatimukset.

Suunnittelu pitää sisällään ohjelmiston määrittelyssä jäsenneltyjen toiminnallisuuksien ja ominaisuuksien suunnittelun

Toteutus tarkoittaa ohjelmiston ohjelmointia sekä testauksen toteutusta

Testaus pyrkii karsimaan ohjelmistosta ohjelmointivirheitä ja muita vikoja. Tyypillisiä testaustapoja ovat yksikkö- ja integraatiotestaus.

Dokumentointi käsittää ohjelmistoprojektin aikana tuotettavan kirjallisen materiaalin, kuten projektisuunnitelmat, testaussuunnitelmat ja jopa ohjelmakoodin kommentoinnin.

Käyttöönotto ja **ylläpito** ovat asiakkaan ongelmien ratkomista, virheiden korjaamista ja tarvittaessa uusien ominaisuuksien lisäämistä.

Laatujärjestelmällä ja **laadunvarmistuksella** on tarkoitus taata, että ohjelmisto täyttää käyttäjän ja asiakkaan toiveet ja odotukset.

Projektinhallinta on työkalu ohjelmistotuotantoprojektin organisointiin. Suuret ohjelmistoprojektit koostuvat usein useasta rinnakkain tai peräkkäin etenevistä osaprojekteista, ja tällöin niiden järjestelmällinen hallinta voi olla keskeistä koko projektin onnistumisen kannalta.

Tuotteenhallinta: Usein kaupallisella ohjelmistolla on useita eri konfiguraatioita, jolloin se voidaan aina räätälöidä yksilöllisesti kullekin asiakkaalle sopivaksi. Tuotteenhallinnan tarkoitus on varmistaa, että asiakkaalla on tarvitsemansa toimiva versio ohjelmistosta.

Ohjelmistotuotantomenetelmä on koko ohjelmistotuotantoprosessin kattava viitekehys, joka ohjaa prosessin osa-alueitten käytännön toteutusta.

3 Ohjelmistotuotannon alkutaival

1940-luvun puolivälistä 1950-luvun loppuun ensimmäiset ohjelmoitavat elektronisesti tallentavat tietokoneet olivat nykystandardeilla mitattuna valtavan kokoisia, hitaita ja muistikapasiteetiltaan mitättömiä. Rajoitustensa vuoksi tietokoneohjelmointia pidettiin toissijaisena ja yksinkertaisena toimena verrattuna itse tietokonelaitteiston suunnitteluun ja rakentamiseen. [10] Lähemmin ohjelmoinnin kanssa työskenteleville alkoi selvitä ohjelmoinnin monimutkaisuus ja ongelmat kuten rajoitetun muistin hallinta ja ohjelmien välinen interaktio. Tarve järjestelmälliselle ohjelmien tuotannolle oli syntymässä.

Edsger Dijkstran mukaan alkukantaisilla tietokoneilla ohjelmointi nähtiin tapana venyttää tietokonelaitteiston rajoja ja kehittyneempien tietokoneitten uskottiin tekevän ohjelmoinnista lähes triviaalia kun rajoja ei enää tarvitsisi venyttää. [8] Niin sanottujen kolmannen sukupolven tietokoneiden tultua markkinoille vuosien 1963 ja 1965 välisenä aikana ohjelmoijat löysivät itsensä kuitenkin uusien ongelmien keskeltä, sillä uusi laitteisto oli myös paljon monimutkaisempi kuin aikaisemmat laitteet. Dijkstra itse summaa suurimpien ongelmien johtuneen kuitenkin tietokonelaitteiston tehokkuuden valtavasta kasvusta:

> Niin kauan kun ei ollut koneita, ohjelmointi ei ollut mikään ongelma; kun meillä oli muutama heikko tietokone, ohjelmointi oli vähäinen ongelma. Nyt kun meillä on giganttisia tietokoneita, on ohjelmoinnista tullut yhtä giganttinen ongelma! [8]

Tietotekniikkayritysten hankkiessa uutta tehokkaampaa laitteistoa seurasi luonnollisesti paine myös hyödyntää niiden kapasiteettia. Sen lisäksi, että ohjelmoijat joutuivat kohtaamaan haasteita ja toteuttamaan ratkaisuja joita oli aikaisemmin vain spekuloitu [8], huomattiin että koulutetusta tietotekniikka-alan työntekijöistä alkoi olla pulaa kysynnän vain kasvaessa. [10] Ohjelmistoprojektit alkoivat kallistua, ja yhä useammin ne ylittivät reilusti budjettinsa

ja aikataulunsa mikäli valmistuivat koskaan.[20] Tätä vaihetta tietotekniikan historiassa on myöhemmin kutsuttu nimellä ohjelmistokriisi, software crisis. Dijkstran mukaan ilmiö tunnustettiin avoimesti vuoden 1968 Naton ohjelmistotuotantokonferenssissa Saksan Garmischissa [8]. Tapahtuma oli merkittävä myös siksi, että termi Software Engineering käytettiin tuolloin ensi kertaa suuressa virallisessa yhteydessä. Tapahtumassa käsiteltiin myös ohjelmistojen tuotanto- ja laaduvalvontasykliä, koodin uusiokäyttöä sekä insinööritieteiden soveltamista ohjelmistojen tuotantoon. [10]

4 Perinteiset ohjelmistotuotantomenetelmät

Perinteisillä ohjelmistotuotantomenetelmillä viitataan suunnitelmavetoisiin tuotantomenetelmiin, joita määrittää projektin alussa suoritettava laaja vaatimusmäärittely ja yksityiskohtainen suunnittelu, ja niitä sarjallisesti seuraavat toteutus-, testaus- ja käyttöönottovaiheet. Menetelmät etenevät lineaarisesti vaiheesta toiseen, ja tämän vuoksi niitä kutsutaan yleisesti kattotermillä 'vesiputousmalli'. Ensimmäisen kerran mallin dokumentoi H.D. Benington vuonna 1956 artikkelissaan *Production of large computer programs* [5]. [17] Tohtori Winston Royce julkaisi vuonna 1970 paljon keskustelua herättäneen tekstinsä *Managing the Development of Large Software Systems* [16]. Artikkelissa Royce esittelee yksinkertaisen vesiputousmallin, mutta esittää sitä kohtaan kritiikkiä esittäen siitä iteratiivisempia versioita. Siitä huolimatta, että Roycen oma suositus oli tehdä vesiputousmallin vaiheet kahdesti aluksi tuottaen prototyypin ja myöhemmin varsinaisen tuotteen, artikkeliin on viitattu vesiputousmallin keskeisenä hahmotelmana.

4.1 Vesiputousmallin rakenne

Vesiputousmalli koostuu toisiaan seuraavista työvaiheista, joista seuraava aloitetaan aina edellisen valmistuttua. Ajalleen tyypillisesti malli muistuttaa insinööritieteiden suunnitteluprosessia, jossa ennen projektin konkreettista toteuttamista tehdään perinpohjaiset suunnitelmat dokumentointeineen. Niin Benningtonin kuin Roycenkin mallin ensimmäiset askeleet liittyvät projektin

vaatimusten määrittelyyn ja kartoitukseen. Roycen mallin ensimmäiset askeleet ovat järjestelmävaatimukset ja ohjelmistovaatimukset[16]. Benington esittää vaiheet vähän laajemmin aloittaen korkeatasoisella ja yleisluontoisella toimintasuunnitelmavaiheella. Vaiheeseen kuuluvat asiakasvaatimukset sekä projektin ajalliset ja budjetilliset vaatimukset. Toimintasuunnitelmavaihetta seuraavat toiminnalliset määrittelyt ja laitteistospesifikaatiot. [5] Näiden pohjalta määritellään itse ohjelmiston toiminnalliset-, tehokkuus-, käyttö- ja laatuvaatimukset. Royce painotti dokumentaation tärkeyttä projektin onnistumisen kannalta, ja alkuvaiheiden valmistuttua tuloksina tulisivat olla määrittelydokumentti ja alustava suunnitteludokumentti. Kun vaatimukset on määritelty, siirrytään analyysivaiheeseen, jossa selvitetään vaatimusten tekniset yksityiskohdat. Analyysin perusteella suunnitellaan lopulta itse ohjelmiston yksityiskohdat, joihin kuuluvat muun muassa ohjelmiston arkkitehtuuri, rajapinnat, käyttöliittymä ja testaus. Roycen mukaan suunnitteluvaiheen lopuksi dokumentoituina tulisivat olla ohjelmiston lopullinen suunnitelma, käyttöliittymäsuunnitelma sekä ohjelmiston testaussuunnitelma. Ohjelmointivaihe toteutetaan suunnitelmien pohjalta ja niitä seuraten. Ohjelmointivaiheen ollessa ohitse siirrytään testausvaiheeseen, jossa ohjelman eri moduulit testataan yhdessä ja erikseen testaussuunnitelman pohjalta. Lopputuotoksena saadaan dokumentti testauksen tuloksista. Viimeinen vaihe on ohjelmiston käyttöönotto ja ylläpito. Tähän vaiheeseen kuuluu mahdollisten käytössä löydettyjen virheiden korjaaminen ja ohjelmiston muuttaminen tai siihen ominaisuuksien lisääminen, mikäli tarve sille syntyy.

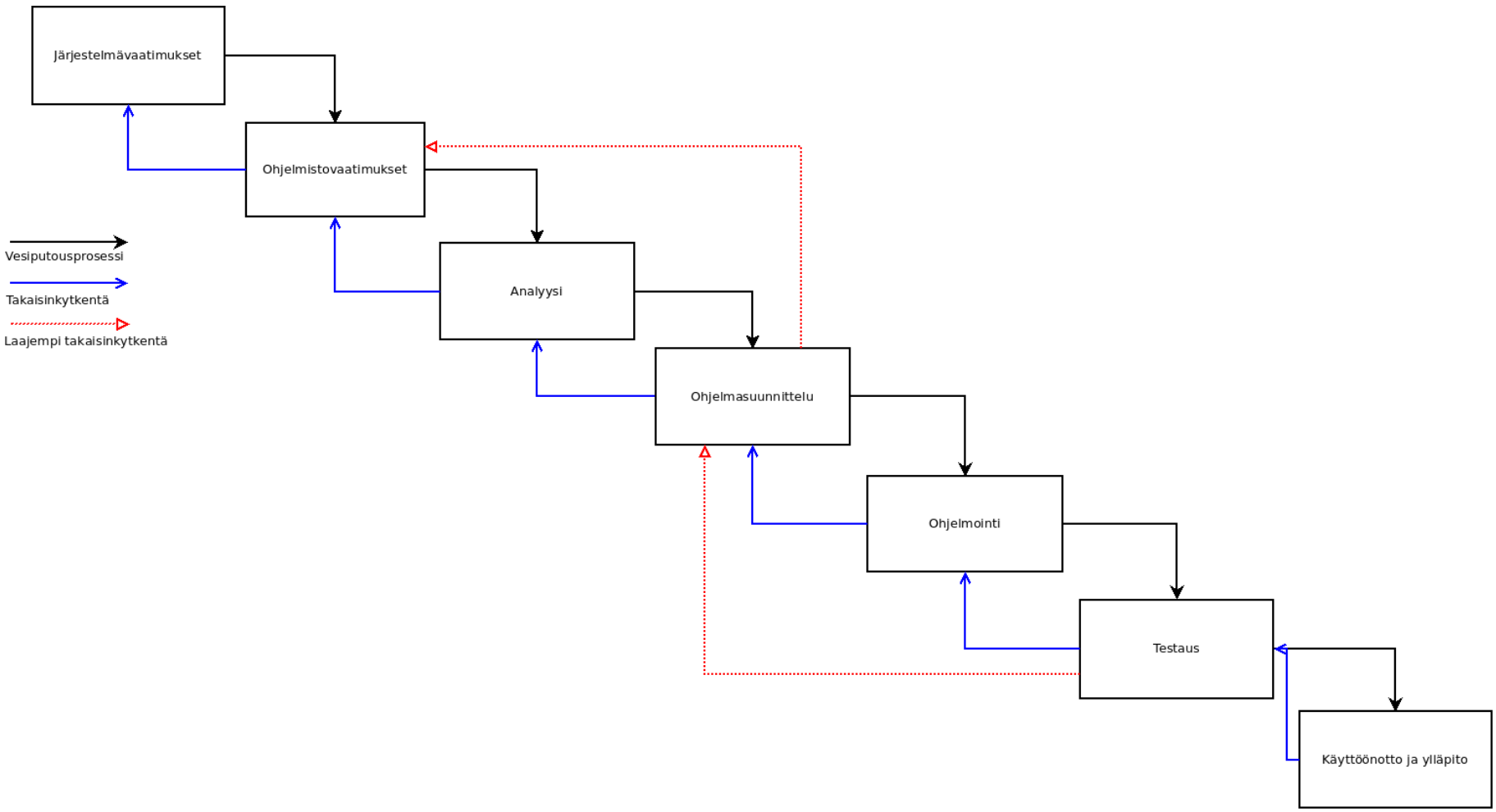
4.2 Vesiputousmallin ongelmia

Koska vesiputousmallille on haettu vaikutteita insinööritieteistä, muistuttaa se paljolti esimerkiksi rakennussuunnitteluprosessia [19]. Siltaa rakentaessa ennen itse rakentamisen aloittamista suunnitellaan sillan arkkitehtuuri, tehdään lujuuslaskelmat, allokoidaan tarvittavat resurssit ja dokumentoidaan suunnitelmat tarkasti. Tarkasti dokumentoiduilla suunnitelmilla itse rakentamisvaihe voitaisiinkin siirtää ulkopuolisen tahon vastuulle, ja näin rakennusalalla usein tehdäänkin. [9] Rakentaminen nähdään triviaalina ja mekaanisena

ohjeiden seuraamisena. Martin Fowlerin mukaan menetelmän ongelmana ohjelmistotuotannossa on se, että suunnitelmien virheet ja heikkoudet ilmenevätkin juuri ohjelmointi- ja testausvaiheissa, jolloin ohjelmiston toteutus ei olekaan enää kovin suoraviivaista. Juuri suunnittelun korostamisessa piilee myös yksi vesiputousmallin kritisoiduimmista piirteistä. Vesiputousmalli nojaa kokonaisvaltaiseen suunnitteluun ja suunnitelman noudattamiseen, mutta ohjelmistojen kehittäminen tapahtuu usein nopeasti muuttuvassa liiketoimintaympäristössä. Käytännössä on usein mahdotonta selvittää kaiken kattavia ja ulkoisille muutospaineille immuuneja järjestelmä- ja ohjelmistovaatimuksia. [19]

Joskus todelliset vaatimuksetkin selviävät vasta ohjelmistoa oikeassa ympäristössä käytettäessä. Teoriassa hyvä suunnitelma voi osoittautua huonoksi myöhemmissä konkreettisimmissa työvaiheissa, mutta vesiputousmallin rakenne vaikeuttaa muutoksien tekemistä suunnitelmiin. Managing the Development of Large Software Systems -artikkelissa [16] Royce esittääkin, että vaiheiden välillä tulisi olla kuvassa 1 havainnollistettuja takaisinkytkentöjä, jolloin uutta tietoa saadessa voitaisiin palata korjaamaan aikaisempien työvaiheiden virheitä. Lisäksi Royce ehdotti, että prosessi kannattaisi suorittaa kahdesti. Esimerkiksi neljänkymmenen kuukauden projektissa ensimmäiset kymmenen kuukautta voisi Roycen mukaan käyttää prototyypin rakentamiseen, jonka pohjalta loput kolmekymmentä kuukautta käytettäiisiin varsinaisen ohjelmiston kehittämiseen.

Toisaalta mikäli ohjelmiston vaatimukset pystytään määrittelemään tarpeeksi tarkasti, esimerkiksi ohjelmiston ollessa sidoksissa johonkin valmiiseen laitteistoon, voi vesiputousmalli olla hyvinkin toimiva ratkaisu suuren työryhmän työskentelyn ohjaukseen.



∞

Kuva 1: Wilson Roycen näkemystä mukaileva vesiputousmalli

5 Inkrementaaliset ja iteratiiviset menetelmät sekä prototyyppaus

Inkrementaalisia ja iteratiivisia ohjelmistotuotantomenetelmiä pidetään yleensä vastauksena vesiputousmallin ongelmiin. Ominaista inkrementaalisille ja iteratiivisille malleille onkin dokumenttijohtoisen lineaarisesti etenevän tuotantoprosessin välttäminen hyödyntäen muun muassa aikaan sidottuja iteraatioita, evolutiivista kehittämistä ja palautteen mukaan ohjautuvaa kehittämistä. [13] Menetelmät perustuvat ohjelmiston vaiheittaiseen kehittämiseen, jossa joka kehitysvaihetta arvioidaan ja arvioiden pohjalta kehitetään paranneltu versio, kunnes vaadittu järjestelmä on valmis. [19] Tunnettuja inkrementaalisia ja iteratiivisia ohjelmistokehitysmenetelmiä ovat muun muassa Extreme Programming eli XP, Rapid Application Development eli RAD, Spiraalimalli ja Rational Unified Process eli RUP. Kahta jälkimmäistä tarkastelen lähemmin vielä myöhemmin.

Iteratiiviset ja inkrementaaliset menetelmät nousivat suuren yleisön tietoisuuteen joulukuussa 1994 kun Yhdysvaltain puolustusministeriö julkaisi uuden standardin Mil-Std-498, jonka keskeisin muutos aikaisempaan oli iteratiivisten ja inkrementaalisten ohjelmistokehitysmenetelmien salliminen vesiputousmallin lisäksi. Kuitenkin malleja oli tunnettu ja käytetty paljon aikaisemmin. Basil ja Larman siteeraavat artikkelissaan *Iterative and Incremental Development: A Brief History* [13] NASAn Project Mercuryssa vuonna 1957 työskennellyttä Gerald M. Weinbergia, jonka mukaan projektissa hyödynnettiin erottamattomasti XP:n kaltaista iteratiivista ohjelmistotuotantomenetelmää. Yhdysvaltain puolustusministeriön projektien ulkopuolella muun muassa IBM ja TRW hyödynsivät inkrementaalisia ja iteratiivisia menetelmiä menestyksekkäästi esimerkiksi NASAn suurissa projekteissa 70-luvulla.[13] 80-luvun alun tekoälyprojektit hyödynsivät laajalti evolutiivista prototyyppaamista. Eräänä 80-luvun keskeisimmistä merkkipaaluista iteratiivisten ja inkrementaalisten ohjelmistotuotantomenetelmien kannalta pidetään Barry Boehmin vuonna 1986 julkaisemaa artikkelia *A Spiral Model of Software Development and Enhancement* [6] jossa spiraalimalli esiteltiin ja jossa formalisoitiin konsepti riskien ohjaamista iteraatioista. 90-luvulla formalisoitiin monia muitakin tun-

nettuja iteratiivisia ja inkrementaalisia ohjelmistotuotantomenetelmiä, kuten RUP, RAD ja XP [13] josta tuli 2000-luvulla ketterän ohjelmistokehityksen keskeisimpiä menetelmiä.

5.1 Spiraalimalli

Barry Boehmin spiraalimalli on iteraatioittain etenevä ohjelmistotuotantomenetelmä. Spiraalimallille ominaista on kehittämisen ohjautuminen riskien ja niiden hallinan mukaan, kun esimerkiksi vesiputousmallissa kehitys ohjautuu suunnitelmien ja määrittelyjen mukaan. Spiraalimalli on siitäkin erikoinen menetelmä, että sen puitteissa pystyy käyttämään niin lineaarisia, inkrementaalisia kuin evolutiivisia työprosesseja riippuen ohjelmiston sen hetkisestä tilasta ja siihen liittyvistä riskeistä ja suunnitelmista.

Spiraalimallia ja sen kierroksia on havainnollistettu kuvassa 2. Jokainen kierros spiraalissa edustaa yhtä vaihetta ohjelmistoprosessissa. Yksi kierros koostuu neljästä järjestyksessä toistuvasta työvaiheesta [19]:

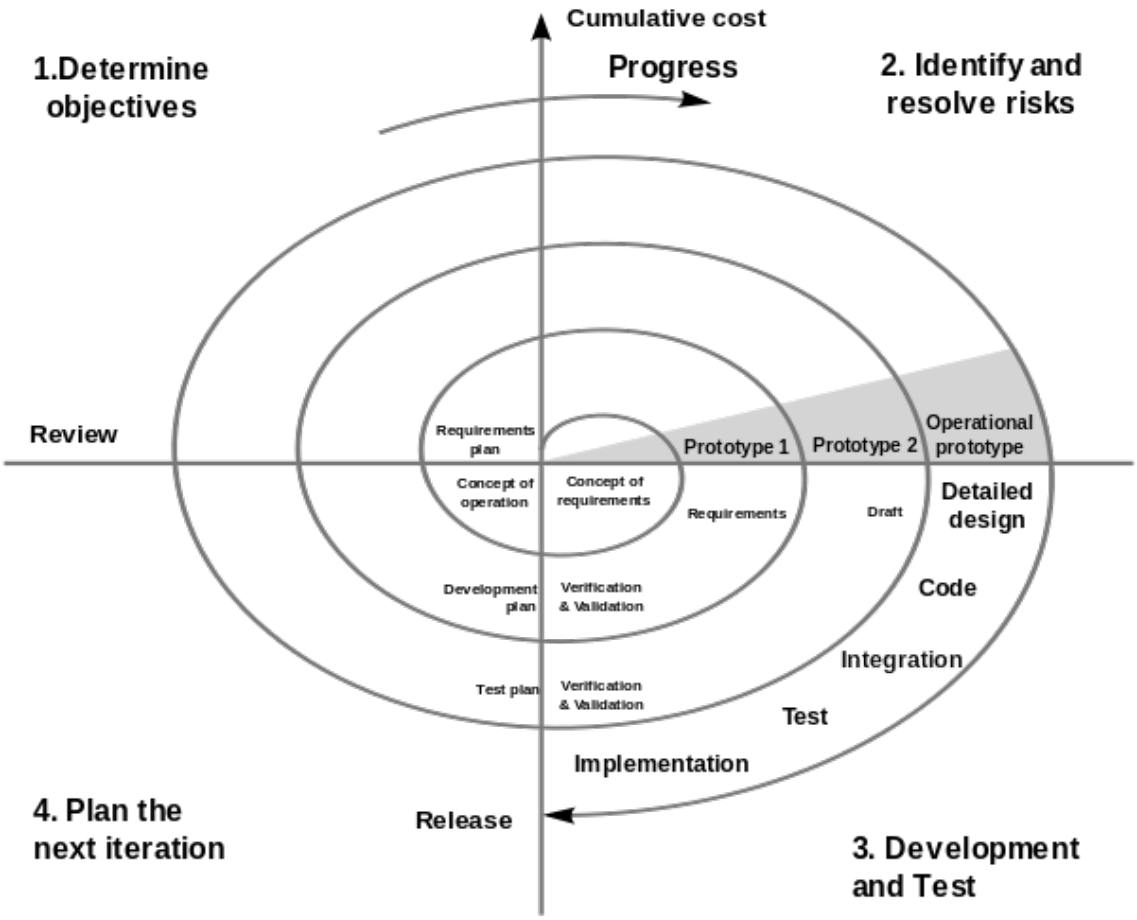
1. **Tavoitteen määrittely:** Työvaiheen tavoitteet määritellään, vaiheeseen ja tuotteeseen kohdistuvat rajoitteet ja riskit tunnistetaan. Laaditaan toimintasuunnitelma sekä riskeistä riippuen vaihtoehtoisia suunnitelmia.

2. **Riskien arviointi ja minimointi:** Jokainen tunnistettu riski analysoidaan yksityiskohtaisesti, ja analyysin pohjalta toimitaan riskien uhkien pienentämiseksi. Esimerkiksi, jos riskinä on ohjelmiston vaatimusten epätarkkuus, voidaan riskien minimoimiseksi alkaa kehittää ohjelmiston prototyyppimallia.

3. **Kehitys ja validointi:** Riskien arvioinnin jälkeen valitaan järjestelmälle tuotantomalli. Jos esimerkiksi käyttöliittymään liittyvät riskit ovat hallitsevia, voi hyvä malli kehittämiselle olla evolutiivinen prototyyppaus kun taas tilanteessa, jossa alijärjestelmien integrointi pääohjelmistoon on suurin riskien lähde, voi vesiputousmalli olla paras lähestymistapa.

4. **Seuraavan vaiheen suunnittelu:** Projekti katselmoidaan projektille keskeisten ihmisten ja organisaatioiden toimesta, ja mikäli päätetään

suorittaa seuraava spiraalin kierros, se sekä siihen varatut resurssit suunnitellaan.



Kuva 2: Boehmin spiraalimalli
(Lähde: Wikimedia Commons, `http://upload.wikimedia.org/wikipedia/commons/e/ec/Spiral_model_%28Boehm%2C_1988%29.svg`

Riski on spiraalimallin keskeisimpiä käsitteitä. Yksinkertaisesti ilmaistuna riskillä tarkoitetaan tapahtumaa jota halutaan välttää ohjelmistotuotantoprojektissa: riski on jokin asia, joka voi mennä pieleen [19]. Riskit jaetaan kolmeen eri kategoriaan, mutta käytännössä osa ohjelmistotuotannon riskeistä voi kuulua useampaankin näistä:

1. **Projektiriskit** ovat projektin aikatauluun tai resursseihin vaikuttavia riskejä. Tällaisia voivat esimerkiksi olla työntekijöiden lähtö projektista tai muutokset projektin johdossa tai organisaatiossa.

2. **Tuoteriskit** vaikuttavat kehitettävän ohjelmiston laatuun ja suorituskykyyn. Tällaisia voivat olla esimerkiksi kolmannen osapuolen tuottaman ohjelmiston osan virheellisyys tai toimimattomuus.

3. **Liikeriskit** vaikuttavat ohjelmistoa kehittävään tai tuottavaan organisaatioon. Tällaisia riskejä voivat olla esimerkiksi markkina- ja kilpailutilanteen muutokset tai ohjelmistoprojektissa käytetyn teknologian vanhentuminen.

Boehm neuvoo spiraalimallissa käytettävän riskienhallintamenetelmää, jossa aluksi tunnistetaan ja priorisoidaan kymmenen tärkeintä riskiä (Tosin tämä luku ei ole absoluuttinen ja voi vaihdella projektista riippuen) ja niitä varten laaditaan toimintasuunnitelmat. Priorisoitujen riskien listaa päivitetään iteraatioittain ja riskin realisoituessa toimitaan sille laaditun suunnitelman mukaan [6]. Ian Somervillen mukaan riskinhallintasuunnitelmat voidaan jakaa kolmeen tyyppin: välttämis-, minimointi- ja valmiusstrategioihin[19]. Välttämisstrategioilla pyritään pienentämään riskin realisoitumismahdollisuutta. Jos riskinä on esimerkiksi vialliset komponentit, voi välttämisstrategiana toimia komponenttien alihankinta luotettavalta valmistajalta. Minimointistrategioilla pyritään pienentämään riskin toteutuessa aiheutuvaa haittaa ja vahinkoa. Esimerkiksi sairastapausten riskin vaikutusta voidaan minimoida organisoimalla työskentely niin, että kukin kehittäjä ei ole yksinään vastuussa jostain osa-alueesta, vaan tarvittaessa joku toinen voi jatkaa sairaana olevan kehittäjän työtä. Valmiusstrategiat ovat toimintastrategioita, joilla varaudutaan pahimpaan tapaukseen riskien realisoituessa. Ohjenuoraksi strategioiden valinnassa Somerville neuvoo lähtökohtaisesti välttämään riskejä. Jos se ei ole mahdollista, niin riskien realisoitumismahdollisuus, ja viime kädessä vaikutukset, tulee minimoida.

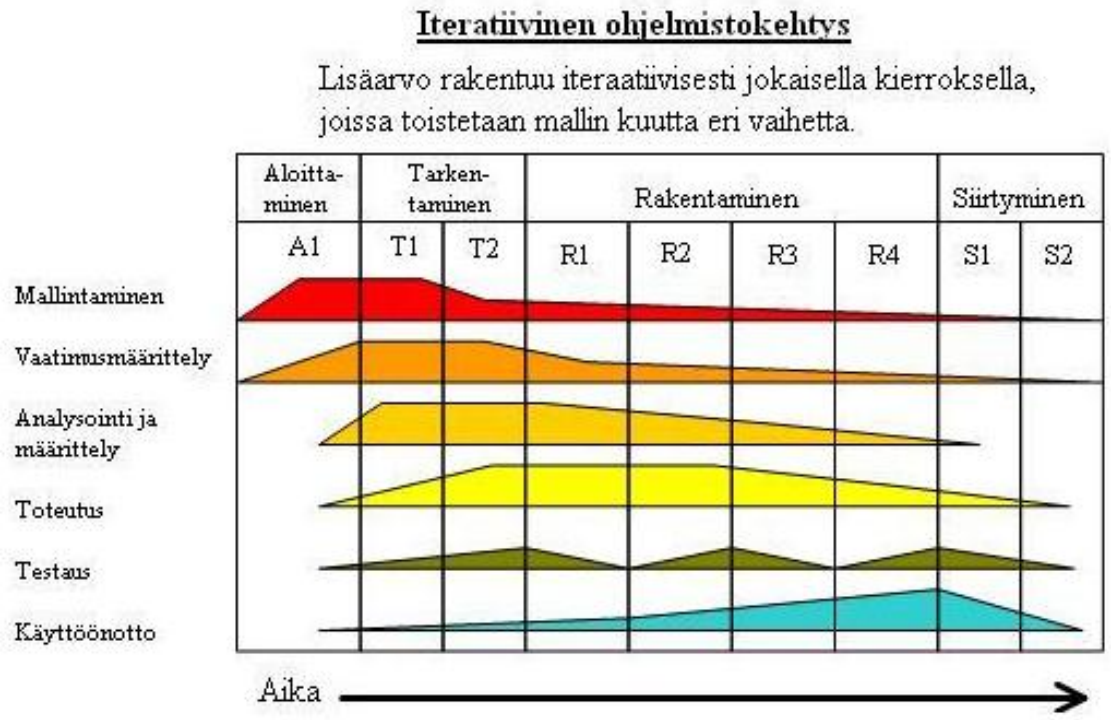
5.2 Spiraalimallin etuja ja heikkouksia

Artikkelissaan *A Spiral Model of Software Development and Enhancement* [6] Barry Boehm myös analysoi spiraalimallin hyviä ja huonoja puolia. Spiraalimallin eduiksi hän laskee mahdollisuuden hyödyntää monen olemassaolevan ohjelmistotuotanntomenetelmän hyviä puolia ja välttää huonoja [6]. Vaihtoehtojen tarkastelu rohkaisee jo olemassa olevan ohjelmiston uusiokäyttöön ja riskianalyysi sekä iteratiivinen malli edesauttavat ohjelmiston muokattavuutta ja laajennettavuutta sekä yleistä laatua. Riskien tuominen keskiöön välttää perinteisempien menetelmien sudenkuopat, jossa helpot asiat suoritetaan ensin riskialttiimpien ja vaikeampien jäädessä myöhemmiksi [21].

Tekstissään Boehm tunnistaa myös mallissa joitain heikkouksia. Koska malli nojaa vahvasti riskien hallintaan, se olettaa käyttäjänsä hallitsevan koko riskien hallinnan konseptin hyvin[6]. Lisäksi osa mallissa ja sen vaiheissa käytettävistä tekniikoista ja toimintatavoista eivät ole kovin tarkkaan määritelty. Esimerkiksi Gerard Wolff moitti menetelmää selkeyden puutteesta ja suosittelikin Boehmin spiraalimalliin suhtauduttavan vain viitteellisenä kehyksenä projektisuunnitelmalle [21].

5.3 Rational Unified Process

Rational Unified Process eli RUP on Rational Software -yhtiön 90-luvulla kehittämä prosessimalli. IBM:n ostettua Rational Softwaren vuonna 2003, on IBM ollut menetelmän keskeisin käyttäjä ja äänitorvi, tarjoten muunmuassa RUPiin liittyviä konsultointipalveluja. RUP on iteratiivinen ohjelmistotuotantomenetelmä joka hyödyntää vahvasti erilaisia dokumentaatiomalleja, kuten UML-diagrammeja, ja käyttötapauksia [17]. Mark Aked korostaa myös RUPin olevan riskiohjautuva [3]. RUP määrittelee työprosessinsa etenevän iteraatioittain. Kuvassa 3 nähdään jokaisen iteraation sisältävät neljä vaihetta: Aloitusvaihe eli inception, tarkentamisvaihe eli elaboration, rakennusvaihe eli construction ja siirtymävaihe eli transition. RUP määrittää myös spesifisti sen, mitä jokaiseen vaiheeseen kuuluu.



Kuva 3: RUP-prosessin kuvaus
(Lähde: Wikimedia Commons, `http://upload.wikimedia.org/wikipedia/fi/f/fa/IteratiivisenMallinVaiheet.JPG`

Aloitusvaiheessa projektille määritellään siihen liittyvät sidosryhmät ja -henkilöt ja muut järjestelmän kanssa tekemisissä olevat tahot. Näitä tietoja käyttäen tarkastellaan projektin taloudellista arvoa, ja päätetään jatkotoimista. Mikäli projekti on taloudellisesti tuottamaton, voidaan sen kehitys lopettaa tässä vaiheessa. Kehittelyvaiheen tarkoituksena on selvittää projektiin liittyvät vaatimukset ja ongelmat sekä määritellä arkkitehtuurinen viitekehys, projektisuunnitelma sekä keskeisimmät riskit, ja tuottaa näistä dokumentaatio. UML-kaavioiden laatimista on erikseen korostettu. Rakennusvaiheeseen kuuluu käytännön suunnittelu, ohjelmointi ja testaus. Vaiheen lopussa valmiina tulisi olla toimivaa ohjelmistoa ja siihen liittyvä dokumentaatio valmiina käyttöönottoa varten. Viimeinen vaihe on siirtymävaihe, jossa ohjelmisto siirretään todelliseeen käyttöympäristöönsä.

RUPissa iteraatioita voi toteuttaa kahdella tavalla. Ensimmäinen tapa, joka on yleinen myös muissa iteratiivisissa menetelmissä, on suorittaa kaikki neljä vaihetta peräkkäin jokaista iteraatiota kohti aina siirtymävaiheesta takaisin aloitusvaiheeseen siirtyen, ja inkrementaalisesti ohjelmistoa rakentaen. Toisaalta myös vaiheiden sisällä voidaan suorittaa iteraatioita. Esimerkkinä kuvassa 3 tarkentamisvaiheessa suoritetaan kaksi iteraatiota ja rakentamisvaiheessa neljä.

RUP myös painottaa valikoitujen parhaitten käytäntöjen hyödyntämistä ohjelmistokehityksessä. Niitä on listattu kuusi[1]:

1. **Kehitä iteratiivisesti.** Iteratiivinen kehitys vähentää riskejä ja tekee projektin etenemisestä ja tilasta helposti seurattavan.

2. **Hallitse vaatimuksia.** RUP suosittelee käyttötapausten käyttöä vaatimusten selvittämiseksi.

3. **Käytä komponentteihin perustuvaa ohjelmistoarkkitehtuuria** Komponenttien hyödyntäminen on selkeää, joustavaa ja muutoksia sekä ohjelmakoodin uudelleenkäyttöä mahdollistavaa.

4. **Mallinna ohjelmisto visuaalisesti** RUP suosittelee UML-mallinnuskielen käyttöä.

5. **Varmista ohjelmiston laatu** Laadunvalvontaa tulisi tehdä jatkuvasti ja koko työryhmän toimesta.

6. **Hallitse muutoksia ohjelmistoon** Muutosta tulisi hallita niin, että jokainen muutos on hyväksyttävä osa ohjelmistoa ja ne integroituvat ohjelmistoon ristiriidatta.

5.4 RUPin etuja ja heikkouksia

RUPin etuna on eri työvaiheiden ja niiden työnkulun erottaminen sekä ohjelmiston käyttöönoton sisällyttäminen työvaiheisiin[19]. Lisäksi RUP on tarkasti dokumentoitu, sen iteratiivisuus ja komponenttiarkkitehtuurin suosiminen tuottavat laadukasta ohjelmakoodia.

RUPin heikkouksia ovat sen huono soveltuminen todella laajoihin ohjelmistoprojekteihin[15, 17]. RUP ei myöskään huomioi valmiin ohjelmiston ylläpitoa ja käytöstäpoistoa [15].

6 Ketterät menetelmät

Ketterät ohjelmistotuotantomenetelmät eli Agile Methods on kattotermi 2000-luvun keskeisimmälle trendille ohjelmistotuotannossa. Ketterät menetelmät syntyivät reaktiona perinteisten menetelmien vaatimus- ja suunnitelmapainotteisuudelle. Ketterien menetelmien puolestapuhujien mukaan perinteiset menetelmät eivät kykene mukautumaan alan ja teknologian nopeasta muutoksesta johtuviin vaatimuksiin [7]. Monet ohjelmistotuotantoalan ammattilaiset ehdottivatkin 2000-luvun taitteessa erilaisia ketterämpiä vaihtoehtoja vastaamaan perinteisen vesiputousmallin raskaudesta ja kankeudesta johtuviin ongelmiin. Näihin kuuluvat muun muassa Extreme Programming:in kehittäjä Kent Beck, SCRUMin luojat Ken Schwaber ja Mike Beedle sekä Crystalin luoja Alistair Cockburn [19, 7]. Vuodem 2001 helmikuussa ketterien menetelmien keskeisimmät puolestapuhujat kirjoittivat ketterälle liikkeelle keskeisen julistuksen *Agile Manifeston* [4], suomeksi *Ketterän ohjelmistokehityksen julistus*. Julistus summaa ketterien menetelmien painopisteet ja arvot suhteessa perinteisiin menetelmiin seuraavasti:

> Löydämme parempia tapoja tehdä ohjelmistokehitystä, kun teemme sitä itse ja autamme muita siinä. Kokemuksemme perusteella arvostamme.
>
> - **Yksilöitä ja kanssakäymistä** enemmän kuin menetelmiä ja työkaluja.
> - **Toimivaa ohjelmistoa** enemmän kuin kattavaa dokumentaatiota.
> - **Asiakasyhteistyötä** enemmän kuin sopimusneuvotteluja.
> - **Vastaamista muutokseen** enemmän kuin pitäytymistä suunnitelmassa.

> Jälkimmäisilläkin asioilla on arvoa, mutta arvostamme ensiksi mainittuja enemmän.[4]

Suurin osa ketterien menetelmien käytänteistä on ollut jo käytössä aikaisemmin. Suurin osa menetelmistä esimerkiksi hyödyntää iteratiivista kehittämistä[7]. Kuitenkin ketterän kehittämisen arvot ja painotukset erottavat ne aikaisemmista menetelmistä. Ketterille menetelmille on ominaista kehittäjien itseohjautuvuus ja keskinäinen, hiearkiaton kommunikaatio, tiivis yhteistyö asiakastahon kanssa ja asiakkaan osallistaminen ja toimivan ohjelmiston tuottaminen ja toimittaminen lyhyissä iteraatioissa.

6.1 Scrum

Scrum on Ken Schwaberin vuonna 1996 esittelemä ketterä ohjelmistotuotantomenetelmä [7]. Schwaberin mukaan Scrum on menetelmä, joka "hyväksyy, että ohjelmistotuotanto on ennalta-arvaamatonta"[18]. Projektit jaetaan tarpeeksi pieniin osiin, jolloin ne ovat paremmin hallittavissa. Kehittäjäryhmät pidetään pieninä ryhmän sisäisen kommunikaation maksimoinniksi. Schwaber ehdottaa ryhmän kooksi maksimissaan seitsemää henkeä [18]. Kehitettävä ohjelmisto pyritään pitämään jatkuvalla testaamisella, dokumentaatiolla ja integraatiolla sellaisena, että se on teoriassa milloin tahansa valmis käyttöönottoon.

Scrumissa iteraatioita kutsutaan sprinteiksi. Sprintit ovat aikaan sidottuja ja kestävät yhdestä kuuteen viikkoa [18]. Ennen jokaista sprinttiä kehittäjäryhmä ja asiakastaho eli Scrum-termein 'product owner' pitävät kokouksen jossa määritellään ja päätetään kyseisessä sprintissä ohjelmistoon toteutettavat ominaisuudet. Jokaisen sprintin lopussa kehittäjäryhmän ja asiakastahon kesken järjestetään kokous, jossa projektin edistymistä analysoidaan ja sprintissä ohjelmistoon toteutetut ominaisuudet katselmoidaan. Näiden kokousten lisäksi Scrumiin kuuluu olennaisesti päivittäin järjestettävät lyhyet kokoukset, daily scrum:it, kehittäjäryhmän kesken. Nämä kokoukset tehostavat kommunikaatiota, pitävät kaikki projektiin liittyvät tahot tietoisina projektin tilasta, auttavat tunnistamaan ongelmia ja pitämään kehittäjätiimin selvillä projektin tavoitteista [7]. Scrumin kommunikaatiopainotus varmistaa, että tuotettu ohjelmisto on sellaista mitä projektin asiakas haluaa, ja että ohjel-

mistoa voidaan kehittää niin, että se tuottaa mahdollisimman paljon välitöntä lisäarvoa asiakkaalle muuttuvassa markkina- ja kilpailuympäristössä.

6.2 Ketterien menetelmien hyviä ja huonoja puolia

2000-luvulla ketterät menetelmät ovat olleet erittäin suosittuja ja käytössä monessa onnistuneessa ohjelmistotuotantoprojektissa.

7 Ohjelmistojen kehitys nykyään ja tulevaisuudessa

Lähteet

[1] *Rational Unified Process - Best Practices for Software Development Teams*, 2001. `https://www.ibm.com/developerworks/rational/library/content/03July/1000/1251/1251_bestpractices_TP026B.pdf`.

[2] Abran, Alain, Moore, James W., Bourque, Pierre, Dupuis, Robert ja Tripp, Leonard L.: *Guide to the Software Engineering Body of Knowledge (SWEBOK)*. IEEE, 2004. `http://www.swebok.org/`, ISO Technical Report ISO/IEC TR 19759.

[3] Aked, Mark: *Risk reduction with the RUP phase plan*. 2003. `http://www.ibm.com/developerworks/rational/library/1826.html#N100E4`.

[4] Beck, Kent, Beedle, Mike, Bennekum, Arie van, Cockburn, Alistair, Cunningham, Ward, Fowler, Martin, Grenning, James, Highsmith, Jim, Hunt, Andrew, Jeffries, Ron, Kern, Jon, Marick, Brian, Martin, Robert C., Mellor, Steve, Schwaber, Ken, Sutherland, Jeff ja Thomas, Dave: *Manifesto for Agile Software Development*, 2001. `http://www.agilemanifesto.org/`.

[5] Benington, H. D.: *Production of Large Computer Programs*. Teoksessa *Proceedings of the 9th International Conference on Software Engineering*,

ICSE '87, sivut 299–310, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press, ISBN 0-89791-216-0. http://dl.acm.org/citation.cfm?id=41765.41799.

[6] Boehm, Barry W.: *A Spiral Model of Software Development and Enhancement*. Computer, 21(5):61–72, toukokuu 1988, ISSN 0018-9162.

[7] Cohen, David, Lindvall, Mikael ja Costa, Patricia: *An introduction to agile methods*. Advances in Computers, 62:1–66, 2004.

[8] Dijkstra, E.: *Classics in software engineering*. luku The humble programmer, sivut 111–125. Yourdon Press, Upper Saddle River, NJ, USA, 1979, ISBN 0-917072-14-6. http://dl.acm.org/citation.cfm?id=1241515.1241525.

[9] Fowler, Martin: *The New Methodology*. 2005. http://martinfowler.com/articles/newMethodology.html.

[10] Grier, David Alan: *Software Engineering: History*. Teoksessa *Encyclopedia of Software Engineering*, sivut 1119–1126. 2010.

[11] Haikala, I. ja Märijärvi, J.: *Ohjelmistotuotanto*. Korkeakoulu-sarja. Satku, 2003, ISBN 9789521404863. http://books.google.fi/books?id=xIVaAAAACAAJ.

[12] Humphrey, W. S.: *The software engineering process: definition and scope*. Teoksessa *Proceedings of the 4th international software process workshop on Representing and enacting the software process*, ISPW '88, sivut 82–83, New York, NY, USA, 1988. ACM, ISBN 0-89791-314-0. http://doi.acm.org/10.1145/75110.75122.

[13] Larman, C. ja Basili, V.R.: *Iterative and Incremental Development: A Brief History*. IEEE Computer, 36(6):47–56, 2003.

[14] Mahoney, Michael S.: *Finding a History for Software Engineering*. IEEE Annals of the History of Computing, (1):8–19, ISSN 1058-6180. http://ieeexplore.ieee.org/search/wrapper.jsp?arnumber=1278847.

[15] Ramsin, Raman ja Paige, Richard F.: *Process-centered Review of Object Oriented Software Development Methodologies*. ACM Comput. Surv., 40(1):3:1–3:89, helmikuu 2008, ISSN 0360-0300. http://doi.acm.org/10.1145/1322432.1322435.

[16] Royce, Winston W.: *Managing the development of large software systems: concepts and techniques*. Teoksessa *Proc. IEEE WESTCON*. IEEE Press, August 1970. Reprinted in Proc. Int'l Conf. Software Engineering (ICSE) 1989, ACM Press, pp. 328-338.

[17] Ruparelia, Nayan B.: *Software Development Lifecycle Models*. SIGSOFT Softw. Eng. Notes, 35(3):8–13, toukokuu 2010, ISSN 0163-5948. http://doi.acm.org/10.1145/1764810.1764814.

[18] Schwaber, Ken: *Controlled Chaos: Living on the Edge*. 1996. http://controlchaos.squarespace.com/storage/scrum-articles/Living%20on%20the%20Edge.pdf.

[19] Sommerville, Ian: *Software Engineering*. Addison-Wesley, Harlow, England, 9 painos, 2010, ISBN 978-0-13-703515-1.

[20] The Standish Group: *Chaos Report*, 1995. http://www.cs.nmt.edu/~cs328/reading/Standish.pdf – last visited 15^{th} of June, 2008.

[21] Wolff, J. G.: *Software Risk Management*. luku The management of risk in system development: "project SP" and the "New Spiral Model", sivut 481–491. IEEE Press, Piscataway, NJ, USA, 1989, ISBN 0-8186-8906-4. http://dl.acm.org/citation.cfm?id=107446.107478.