Ketterien menetelmien ratkaisuja ohjelmistotuotannon ja suunnittelupainotteisten menetelmien ongelmiin

Jarl-Erik Malmström

Kandidaatintutkielma
Helsingin yliopisto
Tietojenkäsittelytieteen laitos

Helsinki, 20. kesäkuuta 2013

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

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

Tekijä — Författare — Author
Jarl-Erik Malmström

Työn nimi — Arbetets titel — Title

Ketterien menetelmien ratkaisuja ohjelmistotuotannon ja suunnittelupainotteisten menetelmien ongelmiin

Oppiaine — Läroämne — Subject
Tietojenkäsittelytiede

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Kandidaatintutkielma	20. kesäkuuta 2013	24

Tiivistelmä — Referat — Abstract

Suunnittelupainotteiset menetelmät eivät sovi nopeasti muuttuvaan internet-aikakauden liiketoimintaympäristöön. Tarvitaan nopeasti reagoivia ja mukautuvia kehitysmenetelmiä. Ketterät menetelmät tarjoavat ratkaisuja muuttuviin vaatimuksiin ohjelmistokehityksen aikaisella kevyellä suunnitteluprosessilla. Ketterillä menetelmillä voidaan laatua unohtamatta toteuttaa käyttäjien toiveita vastaava ohjelmistojärjestelmä. Osa ketterien menetelmien käytänteistä parantaa ohjelmiston laatua ja ovat kustannustehokkaita kun otetaan huomioon laadunvarmistuksen kustannukset.

ACM Computing Classification System (CCS):
Software development process management, Software development methods, Agile software development, Waterfall model, Spiral model

Avainsanat — Nyckelord — Keywords
agile, ketterä, iteraatiivinen, vesiputousmalli, spiraalimalli, pariohjelmointi, testauspainoitteisuus

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — Övriga uppgifter — Additional information

Sisältö

1 Johdanto

Ohjelmistotuotannossa (software development) käytetään työn suunnitteluun ja organisointiin ohjelmistotuotantomenetelmiä (software development methodologies). Menetelmät määrittelevät muodollisen prosessin, jonka lopputuloksena syntyy ohjelmistojärjestelmä.

Ohjelmistotuotannon alkuaikoina tietokoneet olivat kookkaita sekä niiden käyttökustannukset olivat ohjelmistoja tuottavien insinöörien palkkoihin verrattuna korkeat. Korkeista kustannuksista johtuen ohjelmistotuotannossa tarvittiin suunnittelua sekä järjestelmällisiä käytäntöjä. Tietojenkäsittelyä ei tutkittu itsenäisenä tieteenalana, ja ohjelmistojen parissa työskentelevät olivat muiden alojen insinöörejä sekä matemaatikkoja. Ohjelmistotuotannon menetelmät olivat omaksuttu muista insinööritieteistä [6].

Ohjelmistojen merkityksen kasvaessa ihmisten ja tietokoneiden vuorovaikutus korostui yhä enemmän. Ohjelmistoalalle tarvittiin lisää ihmisiä tuottavaan ja luovaan työhön sekä enemmän insinöörejä ja matemaatikkoja kuin oli saatavilla. Ohjelmistotuotantoprojekteihin palkattiin muiden alojen asiantuntijoita, jotka omaksuivat helpommin *ohjelmoi ja korjaa* (code and fix) lähestymistavan kuin insinöörien käytäntänteet [6]. Ohjelmoi ja korjaa menetelmällä kirjoitetaan ohjelmakoodia ilman perusteellista suunnittelua. Ohjelmistoa korjataan vikojen ilmaantuessa sekä ohjelmiston rakennetta, testausta ja ylläpitoa pohditaan myöhemmin [4].

Eroavaisuudet perinteisten insinöörimenetelmien ja ohjelmoi ja korjaa asenteiden välillä loi uutta hakkerikulttuuria merkittävien yliopistojen tietojenkäsittelylaitoksille. Nämä auktoriteetteja vastustavat luovat sankariohjelmoijat tekivät ohjelmoi ja korjaa käytänteillä usein vaikeasti muutettavaa ja ylläpidettävää ohjelmakoodia [6]. Tällainen menetelmä saattoi toimia jos tuotettava ohjelmisto on pieni, mutta järjestelmän kasvaessa uusien toiminnallisuuksien lisääminen vaikeutui. Lisäksi virheiden löytäminen ja korjaaminen vaikeutui järjestelmän kasvaessa [11].

Ohjelmistotuotannon parissa työskentelevät huomasivat, että ohjelmiston kehitykseen liittyvät ilmiöt poikkesivat huomattavasti laitteistoihin liittyvistä ilmiöistä. Laitteistoille laadituilla luotettavuusmalleilla ei voitu arvioida ohjelmistojen luotettavuutta kattavasti. Ohjelmistoprojektien aikatauluja oli vaikea ennakoida, ja henkilöstön lisääminen aikataulun nopeuttamiseksi saattoi myöhästyttää projektia entisestään [6].

Ohjelmistotuotantoon tarvittiin paremmin organisoituja menetelmiä ja kurinalaisia käytäntöjä yhä suurempien projektien hallinointiin. Reaktiona ohjelmoi ja korjaa menetelmille kehitettiin uusia prosessimalleja, joilla pyrittiin parantamaan 1950-luvun insinöörimenetelmien käytänteitä ohjelmistotuotantoon liittyvillä tekniikoilla. Näissä prosessimalleissa ohjelmointia edelsi kattava suunnittelu-, määrittely-, ja analysointivaihe. Näitä menetelmiä kutsutaan suunnittelupainotteisiksi (plan-driven) [3] tai dokumentointipainotteisiksi (document-driven) [4] menetelmiksi. Tässä tutkielmassa

käytämme näistä yhteisesti käsitettä suunnittelupainotteinen menetelmä.

Internetin laajentuminen ja *hypertekstijärjestelmän* (World Wide Web) ilmaantuminen korostivat ohjelmistojen merkitystä. Ohjelmistojen merkitys yritysten kilpailutekijänä ja liiketoimintaympäristön jatkuvat ja muutokset lisäsivät tarvetta lyhentää ohjelmistotuotantoon kuluvaa aikaa. Suunnittelua, määrittelyä ja dokumentointia painottavat menetelmät eivät sopineet jatkuvasti muuttuvaan liiketoimintaympäristöön ja nopeaan ohjelmistokehitykseen (rapid application development) [6].

Tässä tutkielmassa tarkastelemme myös ketteriä menetelmiä (agile methods). Suunnittelupainotteiset menetelmät ovat saaneet kritiikkiä osakseen ja ketterät menetelmät ovat olleet reaktio suunnittelupainotteisten menetelmien heikkouksille sopeutua muutoksiin. Ketterät menetelmät pyrkivät kompromissiin, ohjelmoi ja korjaa-menetelmän ja suunnittelupainotteisten menetelmän väliltä, tarjoamalla riittävän prosessin haluttuun lopputulokseen pääsemiseksi [11].

Käymme läpi erilaisia suunnittelupainotteisia ja ketteriä menetelmiä sekä näiden lähestymistapoja ohjelmistotuotantoon. Selvitämme ohjelmiston laatuun liittyviä käsitteitä ja tarkastelemme ohjelmistotuotannon haasteita. Esittelemme, miten ketterät menetelmät ratkaisevat esitettyjä ohjelmistotuotannon haasteita ja tarkastelemme onko ketterien menetelmien käytänteillä vaikutusta ohjelmistojen laatuun.

Tutkielman rakenne on seuraava: Kappaleessa kaksi käsittelemme ohjelmistotuotannon haasteita ja laatuun liittyviä käsitteitä. Kappaleessa kolme käymme läpi erilaisia suunnittelupainotteisia ja ketteriä menetelmiä. Kappaleessa neljä selvitämme ketterien menetelmien ratkaisuja suunnittelupainotteisten menetelmien ongelmiin ja ohjelmistotuotannon haasteisiin.

2 Ohjelmistotuotannon haasteet ja ohjelmistojen laatu

Tässä kappaleessa käymme läpi erilaisia ongelmia ja haasteita, joita ohjelmistotuotantoprojekteissa usein ilmenee. Esille tuomamme ohjelmistotuotannon haasteet liittyvät erityisesti suunnittelupainotteisten menetelmien käytänteisiin. Tämän jälkeen selvitämme ohjelmistojen laatuun liittyviä määritelmiä sekä miten ohjelmistotuotannossa ilmenevät ongelmat liittyvät ohjelmistojen laatuun.

Ohjelmistotuotantoprojekteissa usein ilmenee monia haasteita projektin koordinointiin sekä valmiin ohjelmiston toimintaan liittyen. Ohjelmistotuotannossa ilmenevät haasteet voidaan jakaa seuraaviin osa-alueisiin: henkilöstön hallinta, projektin aikataulu ja taloudelliset resurssit, vaatimusten hallinta, tekniset valmiudet ja projektin mahdolliseen ulkoistukseen liittyvät ongelmat [4].

Suunnittelupainotteisissa menetelmissä asiakas on ohjelmistoprojektis-

sa tiiviissä yhteistyössä kehittäjäorganisaation kanssa usein vain ohjelmistoa suunniteltaessa. Ohjelmistoa rakennettaessa asiakasyhteistyö on usein vähäistä itse kehittäjätiimin kanssa. Vähäinen asiakasyhteistyö voi johtaa erilaisiin ongelmiin: Ohjelmistoon saatetaan kehittää toiminnallisuuksia, joita ei tarvita tai ne ovat väärin määriteltyjä. Käyttöliittymästä voi olla vaikeakäyttöinen tai puutteellinen. Koska ohjelmoijat eivät ole tiiviissä yhteistyössä asiakkaan kanssa, tarvittavien toiminnallisuuksien yksityiskohdat saattavat jäädä epäselviksi. Jos asiakas ei ole kehittäjätiimin saatavilla, ohjelmoijat joutuvat tekemään ratkaisuja puutteellisten tietojen varassa [4].

Vaadittavien toiminnallisuuksien epäselvyydestä ja ohjelmoijien omasta ammatillisesta kiinnostuksesta johtuen, ohjelmoijat voivat lisätä tarpeettomia toiminnallisuuksia. Väärien ja tarpeettomien toiminnallisuuksien kehitys ohjelmistotuotantoprojektin resursseja ja nostaa ohjelmistokehitykseen vaadittavia kustannuksia [4].

Suunnittelupainotteisissa menetelmissä ohjelmistojärjestelmän suunnitelmat on saatava valmiiksi ennen kuin sitä voidaan alkaa rakentamaan. Ohjelmiston kattava suunnittelu ennen sen rakentamista on kuitenkin vaikeaa, koska on vaikea huomioida kaikkia yksityiskohtia ohjelmiston toimintaympäristöön ja operatiiviseen toimintaan liittyen [4].

Puutteellisista tiedoista johtuen ohjelmistot eivät usein tyydytä käyttäjien tarpeita. Ohjelmistotuotannon kehitys pitkittyy, kun ohjelmistoa on muutettava käyttäjien tarpeita vastaavaksi. Ohjelmistotuotantoon varattu aikataulu ja budjetti kasvaa, kun ohjelmistojärjestelmää pitää uudelleen suunnitella ja rakentaa [2].

Suunnittelupainotteisissa menetelmissä ohjelmistojärjestelmän oikeanlainen ja haluttu toiminta on varmistettu ohjelmistokehityksen loppuvaiheessa tehtävällä ohjelmistotestauksella: kirjoitettu ohjelmakoodi varmistetaan testaavalla koodilla, joka käy läpi ohjelmiston eri vaiheita, erilaisilla syötteillä sekä tulostuksilla. Tuloksilla voidaan päätellä toimiiko rakennettu järjestelmä oikein. Ohjelmistokehitysjakson lopussa vasta testausvaiheessa voi ilmetä ongelmia, joihin analyysilla ja suunnittelulla ei ole voitu varautua. Testauksessa ilmenevät ongelmat saattavat johtaa muutoksiin järjestelmän vaatimusmäärittelyssä ja ohjelmiston suunnittelussa. Lisääntynyt suunnittelu- ja määrittelytyö johtavat jälleen kohoaviin ohjelmistotuotannon kustannuksiin [19].

Ohjelmistotestauksessa saattaa ilmetä myös rakennetun järjestelmän odotettua heikompi suorituskyky. Toisaalta tietotekniset mahdollisuudet on voitu arvioida väärin eikä käytössä olevien tietokoneiden laskentakyky riitä suunniteltuun järjestelmään [4].

2.1 Koordinointi

Tässä tutkielmassa koordinoinnilla tarkoitamme yksilöiden ja ryhmien välistä yhteistoimintaa sekä näihin liittyvää hallinnollista työtä ja toisaalta ohjel-

mistojärjestelmän eri osien yhteensovittamista. Ohjelmistoprojektin koordinointiongelmat liittyvät henkilöstön, vaatimusten hallintaan [15].

Yksittäinen henkilö tai pieni ryhmä voi tehokkaasti koordinoida pienen ohjelmistojärjestelmän kehitystä ja hallinoida toteutuksen yksityiskohtia. Yksilöiden tai ryhmän on mahdotonta toteuttaa ja hallinnoida suuria, miljoonien rivien ohjelmakoodien, ohjelmistoja ja useiden vuosien ohjelmistokehitystä yksityiskohtaisesti [15].

Suuriin ohjelmistoprojekteihin saattaa liittyä useita kehittäjäorganisaatioita ohjelmiston tilaajan lisäksi ja nämä onnistuvat useimmin jos projektia koordinoi henkilö, jolla on tietoa ohjelmiston kohdealueelta sekä ohjelmistoalalta. Kohdealueen tieto ei ole usein käytössä kaikilla projektiin osallistuvilla, koska suuret projektit johtavat työn erikoistumiseen ja jakamiseen. Organisaatiossa työn eriytyminen johtaa toisistaan riippuvien tekijöiden jakamiseen osastoihin maantieteellisesti, organisatorisesti, sekä sosiaalisesti. Tämä vähentää mahdollisuuksia ja haluja oppia sekä jakaa tietoa etäisten työtovereiden kesken ja kohdealueen tietoa katoaa [15].

Koordinointiongelma liittyy myös rakennettavaan ohjelmistojärjestelmään: ohjelmistot ovat pääasiallisesti rakennettu useista osista, jotka on kytkettävä yhteen ja ohjelmistojärjestelmän osien on toimittava koordinoidusti. Ohjelmistojen suuri koko aiheuttaa ongelmia, koska ohjelmisto vaatii sen osajärjestelmien täsmällistä integraatiota, jotta ohjelmisto toimisi oikein [15].

Tiedon koordinointi eri organisaatioiden välillä on usein haasteellista. Ohjelmistokehityksessä haaste on, että ohjelmistoarkkitehtien ja -suunnittelijoiden sekä ohjelmoijien päätöksentekoon tarvitsema tieto ei ole saatavilla. Ohjelmiston toimintaan liittyvät tiedot usein muuttuvat ohjelmistoprojektin edetessä. Muutoksia ohjelmiston vaatimuksiin esiintyy, koska liiketoiminta, käyttäjien toiveet, tietokoneympäristö, ohjelmiston syötteet ja fyysinen maailma muuttuvat [15].

2.2 Suunnittelu ja muuttuvat vaatimukset

Asiakkaalla on vaatimuksia kehitettävän järjestelmän toiminnallisuuksista, joita ohjelmoijat toteuttavat. Vaikeuksia ilmenee näiden vaatimuksien ymmärtämisessä ja usein vaatimukset muuttuvat projektin edetessä. Vaatimukset muuttuvat koska liiketoimintaympäristöt muuttuvat tai asiakkaat saattavat keksiä uusia toiminnallisuuksia toteutettavalle järjestelmälle [11].

Ohjelmistotuotannossa asiakkaiden vaatimien toiminnallisuuksien suunnittelu ja toteutus on haasteellista: tyypillisesti suunnittelupainotteisissa menetelmissä ohjelmistotuotantoprojektiin osallistuva henkilö vaihtelevalla kohdealueen tuntemuksella haastattelee asiakkaita ja käyttäjiä. Tämän jälkeen haastattelija kirjaa asiakkaan haluamat toiminnallisuudet ohjelmistoarkkitehdeille ja -suunnittelijoille. Merkityksellistä kohdealueen tietoa katoaa usein, koska asiakas ei osaa tuoda esille kaikkia tarpeitaan ja jotkin ilmoitetut tarpeet jäävät kirjaamatta.

Teknologian ja liiketoiminnan vaatimusten muuttuessa suunnitelmat vanhentuvat nopeasti [21]. Alkuperäisen suunnitelman seuraaminen ei ole ohjelmistoprojektien päämäärä, sen sijaan toimitettavan ohjelmiston tarkoitus on asiakkaan mahdollisesti muuttuvien tarpeiden tyydyttäminen [13]. Internet liiketoimintaympäristönä vahvistaa ohjelmistotuotannon ongelmia korostamalla nopeutta. Asiakkailla on epätoivoinen kiire markkinoille. He vaativat liiketoiminnalle arvoa tuottavia ominaisuuksia yhä nopeammassa tahdissa [1].

Muutostarpeita ilmestyy todennäköisemmin, kun käyttäjät käyttävät toimivaa ohjelmistoa. Tällöin käyttäjät näkevät ohjelmiston rajoitteet ja mahdollisuudet. Kun ihmiset käyttävät ohjelmistoa, niin he todennäköisesti keksivät uusia toiminnallisuuksia olemassa olevalle järjestelmälle [15]. Ohjelmiston toiminnallisuuksien arvoja on vaikea nähdä ennen kuin ohjelmistoa käytetään oikeassa toimintaympäristössä [11].

Muuttuvat tai puutteelliset vaatimukset voidaan nähdä johtuvan heikosta suunnittelusta. Suunnittelupainotteisissa menetelmissä ajatuksena usein on saada selkeä kokonaiskuva toteutettavista toiminnallisuuksista ennen ohjelmiston rakentamista sekä rajoittaa asiakkaan vaatimia muutoksia asiakkaan ja kehittäjäorganisaation välisellä sopimuksella [11].

Ohjelmistojärjestelmältä vaadittavien toiminnallisuuksien hallinnointi on asiakkaan kannalta ongelmallista: erilaisten vaihtoehtojen vertailu on vaikeaa, koska ohjelmistokehittäjät eivät yleensä tarjoa hinta-arvioita erilaisista toiminnallisuuksista. Ilman tietoa hinnasta on vaikea arvioida halutaanko tietystä toiminnallisuudesta maksaa vai ei. Arviointi on vaikeaa, koska ohjelmistokehitys on suunnittelutyötä [11]. Toisin kuin teollinen valmistus, ohjelmistokehitys ei ole rutiininomainen toimi: ohjelmistoprojektit ovat yksilöllisiä eikä ohjelmistojärjestelmille usein ole valmiita prototyyppejä [15].

2.3 Tekninen kehitys ratkaisuna

Teknologian ja ohjelmistotuotantomenetelmien kehityksestä huolimatta, suuriin ohjelmistotuotantoprojekteihin liittyviä ongelmia ei ole onnistuttu lopullisesti ratkaisemaan. Voidaan sanoa, että aikaisemmat korjaustoimenpiteet ovat lähestyneet ongelmia seuraavasti [15]:

- Kehittämällä ohjelmistotuotannossa tarvittavia teknisiä työkaluja.
- Jakamalla ohjelmisto osiin (modularization) teknisesti, esimerkiksi olio-ohjelmoinnilla (object-oriented programming).
- Jakamalla ohjelmistotuotantoa hallinnollisesti: eriyttämällä vaatimusmäärittelyn, ohjelmoinnin ja testaustoiminnot.
- Formalisoimalla teknisiä menettelytapoja, esimerkiksi versionhallinta, testisuunnitelma ja vaatimusmäärittelydokumentit.

Nykyään ohjelmistokehittäjät käyttävät laajasti työkaluja, jotka nopeuttavat suunnittelu- ja ohjelmointiprosessia. Uuden teknologian työkalut tarjoavat toimintoja, jotka ennen oli toimitettava itse. Erilaiset ohjelmistokehykset (frameworks) tekevät osan ohjelmiston kehitystä. Ohjelmistokehitystä voidaan nopeuttaa komponenttien uudelleenkäytöllä. Sen sijaan, että rakennettaisiin ohjelmisto alusta alkaen itse, valmiita komponentteja hankintaan, yhdistetään ja kootaan nopeasti [1].

Nykyiset ohjelmistotyökalut eivät ole tarjonneet yksiselitteisiä ratkaisuja haasteisiin. Työkalut lisäävät yksilöiden tuottavuutta, mutta eivät ratkaise ohjelmistotuotannon koordinointiongelmaa. Onnistunut ohjelmistotuotantoprojekti vaatii lisäksi erilaisten näkemysten sekä ohjelmistojärjestelmän osien yhteensovittamista [15].

2.4 Ohjelmistojen laatu

Laatu on monitahoinen käsite ja sitä voidaan kuvailla viidestä eri näkökulmasta: *Täydellisen näkemyksen* (transcendental view) mukaan laatua voidaan kuvailla, mutta ei määritellä eikä täysin toteuttaa. Täydellinen näkemys kuvaa ohjelmiston laatua ihanteena (ideal), mihin ohjelmiston toteutuksessa pyritään [14].

Käyttäjänäkemys (user view) laadusta liittyy konkreettisesti tuotteeseen, joka tyydyttää käyttäjän tarpeet. Käytettävyys on esimerkiksi sellainen tuotteen piirre, mikä liittyy käyttäjän näkemykseen laadusta. Käytettävyydellä tarkoitamme ohjelmiston käytöstä aiheutuvan todellisen ja oletetun vaivannäön suhdetta [14].

Tuotantonäkemys (manufacturing view) tarkastelee laatua valmistusprosessien näkökulmasta. Näkemys liittyy tuotteen oikeanlaiseen valmistukseen ja ajatuksena on vähentää tuotteeseen tehtävistä muutoksista aiheutuvia kustannuksia [14].

Tuotenäkemys (product view) tarkastelee tuotteen sisäisiä ominaisuuksia ja näiden vaikutusta sisäiseen laatuun (internal quality). Tätä lähestymistapaa puolletaan usein oletuksella, että sisäistä laatua mittaamalla ja valvomalla parannetaan käyttäjän näkemykseen liittyvää tuotteen ulkoista laatua (external quality) [14].

Arvoperustainen näkemys (value-based view) tarkastelee laatua suhteessa siitä aiheutuviin kustannuksiin. Joskus edellä mainitut näkemykset voivat olla keskenään ristiriitaisia. Esimerkiksi jos käyttäjä haluaa uusia toiminnallisuuksia tai muutoksia ohjelmistoon ja toisaalta tuotantonäkemys pyrkii vähentämään muutoksia. Arvoperustainen näkemys auttaa löytämään tasapainon kustannusten ja laadun väliltä [14].

Oletetaan, että tilattu ohjelmistojärjestelmä toimitetaan ajallaan ilman budjettia ylittäviä kustannuksia, ja se toimii oikein sekä suorittaa tehokkaasti sille määritetyt toiminnallisuudet. Voidaanko tuotteeseen tällöin olla tyytyväisiä? Ei välttämättä kaikissa tapauksissa. Ohjelmistojärjestelmää

voi olla vaikea ymmärtää ja muuttaa. Ohjelmistoa ei välttämättä ole helppokäyttöinen. Ohjelmisto voi olla tarpeettoman laitteistoriippuvainen. Nämä seikat johtavat kohtuuttomiin ylläpitokustannuksiin [5]. Kansainvälinen standardointiorganisaatio (ISO) on suositellut laadun perustaksi seuraavia itsenäisiä piirteitä [14]:

1. toiminnallisuus (functionality)
2. luotettavuus (reliability)
3. käytettävyys (usability)
4. tehokkuus (efficiency)
5. ylläpidettävyys (maintainability)
6. siirrettävyys (portabilty)

Toiminnallisuudella ja luotettavuudella tarkoitamme seuraavaa: toiminnallisuuksien on tyydytettävä käyttäjän vaatimukset sekä kyettävä ylläpitämään vaadittu suorituskyky vaaditun ajan [14].

Käytettävyys mittaa ohjelmistojärjestelmän käytöstä aiheutuvan ja käyttäjien olettaman vaivannäön suhdetta [14].

Tehokkuudella tarkoitamme suorituskyvyn sekä käytettyjen resurssien suhdetta ja ylläpidettävyydellä ohjelmistoon tehtäviin muutoksiin tarvittavaa työmäärää, jotka voivat liittyä korjauksiin, parannuksiin tai ohjelmiston mukauttamista muuttuneisiin olosuhteisiin [14].

Siirrettävyys viittaa ohjelmiston kykyyn toimia erilaisessa ympäristössä: toisessa organisaatiossa, laitteistossa tai ohjelmistoympäristössä [14].

Yllä mainitut kansainvälisen standardointiorganisaation laatumallin piirteistä liittyvät käyttäjän näkemykseen (user view) ohjelmistosta ja miten se tyydyttää hänen tarpeensa. Tuotettavan ohjelmistojärjestelmän vaatimuksia edustaa käyttäjän tai asiakkaan tarpeet ja näkemys. Toimme esille, ohjelmistotuotannon haasteita käsitellessämme, asiakkaiden tarpeiden ja vaatimusten hallinnan aikaisemmin kappaleessa 2.2. Voimme todeta ohjelmistotuotannon haasteiden ja ohjelmiston laadun liittyvän toisiinsa sekä vaatimusten hallinnan liittyvän myös ohjelmiston laadun hallintaan [14].

Seuraavaksi tarkastelemme suunnittelupainotteisia ja ketteriä menetelmiä sekä näiden käytänteitä. Myöhemmin käymme läpi ketterien menetelmien ratkaisuja ohjelmistotuotannon sekä suunnittelupainotteisten menetelmien haasteisiin.

3 Suunnittelupainotteiset ja ketterät menetelmät

Suunnittelupainotteiset ohjelmistotuotantomenetelmät ovat ohjelmistokehityksen yksityiskohtaisia ja kurinalaisia prosesseja, joiden tarkoituksena

on tehdä ohjelmistotuotannosta ennustettavaa ja tehokasta sekä välttää ohjelmistotuotantoon liittyviä riskejä. Muista insinööritieteistä vaikutteita saaneet menetelmät ovat yksityiskohtaisia prosesseja, joissa painotetaan ennen ohjelmiston rakentamista tehtävää suunnittelutyötä [11].

Suunnittelupainotteiset menetelmien lähestymistapa perustuu oletukselle, että aikaisella ja täydellisellä vaatimusmäärittelyllä voidaan vähentää kehityksen aikana tulevia muutoksia ja niistä aiheutuvia kustannuksia. Ketterien menetelmien näkökulmasta muutosten kieltäminen merkitsee reagoimattomuutta liiketoimintaympäristön kehitykselle [13].

Ketterissä menetelmissä kehittäjäorganisaatiot ovat valmiimpia reagoimaan muutoksiin ohjelmistokehityksen aikana kuin suunnittelupainotteisissa menetelmissä: kaikki kehityksen aikana esille tulleet uudet vaatimukset tai muutoksia kaipaavat valmiit toiminnallisuudet otetaan esille asiakkaan toimesta, ja ohjelmistoa kehitetään jatkuvasti yhteistyössä hänen kanssa [21].

Käymme seuraavaksi läpi lyhyesti vesiputousmallin (waterfall model) ja spiraalimallin (spiral model) sekä näihin prosessimalleihin liittyviä ongelmia. Myöhemmin tarkastelemme miten ketterät menetelmät ovat pyrkineet ratkaisemaan näissä prosessimalleissa ilmenneitä ongelmia ja ohjelmistotuotannon haasteita.

3.1 Vesiputousmalli

Vesiputousmalli (waterfall model) on lineaarinen (kuvassa 1.) vaiheesta seuraavaan etenevä prosessimalli. 1970-luvulla vesiputousmalli vaikutti suuresti lineaarisiin ohjelmistotuotannon suunnittelupainotteisiin prosessimalleihin. Vesiputousmallin lähestymistapa auttoi poistamaan monia aiemmin ohjelmistotuotantoa vaivanneita ongelmia: ohjelmoi ja korjaa mallilla ohjelmakoodista tuli korjausvaiheiden jälkeen usein kallista muuttaa, koska ohjelmakoodi oli rakenteellisesti huonoa eikä muutoksiin ja testaamiseen oltu varauduttu. Ohjelmoi ja korjaa mallilla tehdyt ohjelmistojärjestelmät eivät usein täyttäneet käyttäjien tarpeita, koska ne olivat rakennettu ilman tarkempaa suunnitelmaa ja analyysia [4].

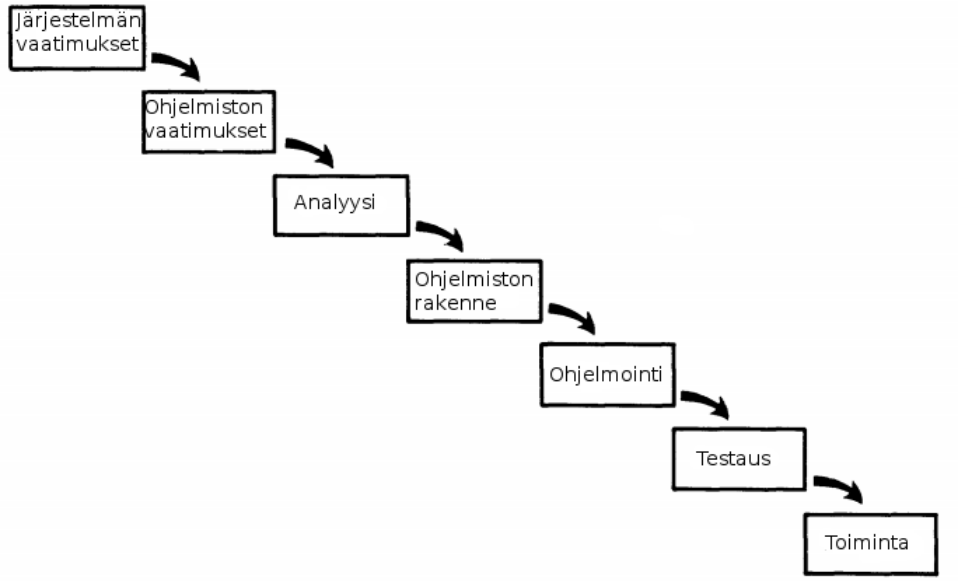
Vesiputousmallissa ohjelmistotuotanto koostuu seuraavista vaiheista: järjestelmän ja ohjelmiston vaatimusmäärittely sekä analyysi, ohjelmistonrakenteen suunnittelu, ohjelmointi, testaus ja ohjelmiston käyttö [19].

Määrittely- ja analyysivaiheessa kerätään kehitettävän järjestelmän ja ohjelmiston vaatimukset ja rajoitteet. Vaatimukset ovat joukko toiminnallisuuksia, joita loppukäyttäjä odottaa ohjelmistolta. Loppukäyttäjien vaatimuksien ja liiketoimintaympäristön analysointi on edellytys ohjelmiston rakenteen suunnittelulle [19].

Määrittely- ja suunnitteluvaiheen jälkeen suunnitellaan järjestelmän rakenne: ohjelmiston arkkitehtuuri, tarvittavat luokat ja niiden toiminnallisuus sekä komponenttien yhteensopivuus ja yhteistoiminta [19].

Ohjelmointivaiheessa kirjoitetaan ohjelmakoodi laadittujen suunnitelmien

Kuva 1: Lineaarinen ohjelmistotuotantoprosessi



perusteella ja testausvaiheessa ohjelmistojärjestelmän oikeanlainen toiminta varmistetaan etsimällä selviä virheitä tuotantokoodista visuaalisesti tarkastelemalla ja kirjoittamalla testitapauksia tuotantokoodin osille. Testitapaukset testaavat järjestelmän osia erilaisilla syötteillä sekä tarkastavat, että tulosteet ja järjestelmän tila ovat oikein [19].

Vesiputousmallissa painotetaan dokumentin tärkeyttä: Hyvän dokumentoinnin todellinen arvo ilmenee testausvaiheessa, ohjelmistoa käytettäessä sekä uudelleen suunniteltaessa. Hyvän dokumentin avulla esimies voi keskittää henkilöstön ohjelmistossa ilmenneisiin virheisiin. Ilman hyvää dokumenttia, ainoastaan ohjelmistovirheen alkuperäinen tekijä kykenee analysoimaan kyseessä olevan virheen. Dokumentin avulla ohjelmistojärjestelmää testaava henkilö voi ymmärtää järjestelmän toimintaa paremmin. Käyttöönotossa ilmenneiden ohjelmistovirheiden korjaamisessa selkeä dokumentti on välttämätön [19].

Vesiputousmallin ongelmana on dokumentaation korostuminen valmistumiskriteerinä aikaisille vaatimuksille ja suunnitteluvaiheille. Menetelmä ei sovi erityisesti interaktiivisiin loppukäyttäjien sovelluksiin, koska käyttäjät näkevät toimivaa ohjelmistoa ohjelmistotuotantoprojektin loppuvaiheessa. Dokumentointipainotteisuus pakottaa kirjaamaan käyttöliittymien vaatimukset yksityiskohtaisesti, mutta käyttäjät eivät kykene tyhjentävästi kertomaan mitä toiminnallisuuksia ohjelmistolta haluavat. Asiakas saattaa muuttaa mielensä. Tai hän osaa usein sanoa, nähdessään valmiin tuotteen, mitä olisi ohjelmistolta halunnut [2].

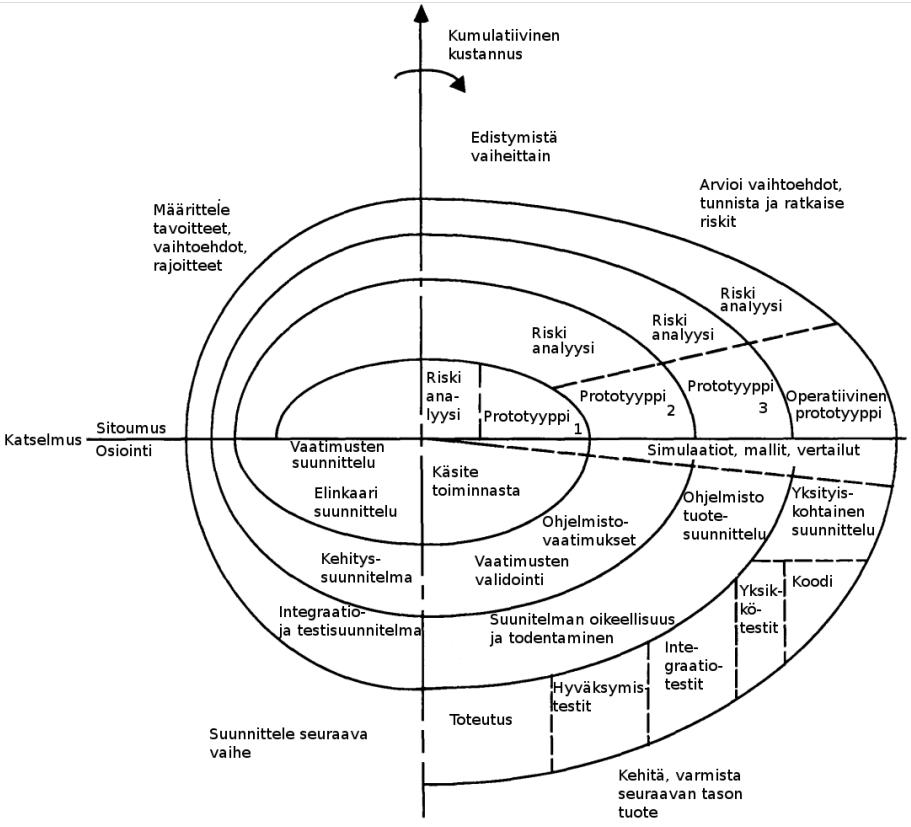
Muuttuvista vaatimuksista seuraa käyttökelvottoman ohjelmakoodin

suunnittelua ja toteutusta. Muutosten kustannukset kasvavat ohjelmiston elinkaaren aikana: mitä pidemmälle projekti etenee sitä kalliimpaa muutosten tekeminen on [13]. Lineaarisen ohjelmistotuotantomenetelmän vaiheet ovat tällaisille projekteille selvästi väärässä järjestyksessä. Joillekin ohjelmistoille ei ole tarvetta yksityiskohtaiselle dokumentaatiolle ennen toteutusta [4].

3.2 Spiraalimalli

Spiraalimallin (spiral model) tarkoitus oli edistää ohjelmistotuotantoprosessia lähestymällä ohjelmistoa iteratiivisesti (iterative) ja inkrementaalisesti (incremental) analysoimalla tuotannossa kohdattavia ongelmia. Tämä mahdollistaa mallin mukautumaan, kohdattavien riskien mukaan, painottamalla määrittelyä (specification), prototyyppien valmistusta, simulointia tai muita lähestymistapoja ohjelmiston suunnitteluun. Spiraalimalli on kehitetty vesiputousmallista saatujen kokemuksien perusteella [4]. Kuvassa 2. oleva spriraalimallin jokainen vaihe pitää sisällään samat toimenpiteet [4]:

Kuva 2: Spiraalimallin ohjelmiston elinkaari



Spiraalimallissa jokainen vaihe aloitetaan tunnistamalla [4]:

- laadittavien ohjelmisto-osien suorituskykyyn, toiminnallisuuteen sekä sopeutumiskykyyn liittyvät tavoitteet
- vaihtoehtoiset toteutustavat (ohjelmiston osto, ohjelmiston uudelleenkäyttö, vaihtoehtoiset ohjelmat)
- ohjelmiston eri vaihtoehdoille asettamien rajoitteet (rajapinnat, aikataulu, kustannukset)

Seuraava askel on arvioida vaihtoehtoja suhteessa ohjelmiston tavoitteisiin ja rajoitteisiin. Usein tämä prosessi tunnistaa epävarmoja alueita, jotka ovat merkittäviä riskin lähteitä. Seuraava askel pitää sisällään kustannustehokkaan strategian muotoilun havaittujen riskien ratkaisemiseksi. Tähän voi liittyä prototyyppien valmistamista, simulointia, vertailuanalyysia, kyselylomakkeita, analyyttista mallinnusta, tai näiden yhdistelmiä sekä muita riskien ratkaisumenetelmiä [4].

Jos suorituskykyyn tai käyttöliittymään liittyvät riskit hallitsevat ohjelman kehittämistä, seuraavassa vaiheessa määritellään ohjelmiston yleistä luonnetta, suunnitellaan seuraavan tason prototyyppiä ja kehitetään yksityiskohtaisempaa prototyyppiä riskien ratkaisemiseksi [4].

Riskienhallinnan avulla aikaa ja työmäärää voidaan kohdentaa ongelma-alueisiin: toiminnan suunnitteluun (planning), asetuksien hallintaan (configuration management), laadun varmistukseen (quality assurance), muodolliseen todentamiseen (formal verification) tai testaukseen [4].

Spiraalimallin tärkeä ominaisuus on, että jokainen iteraatio päätetään katselmukseen tuotteeseen liittyvän henkilöstön tai organisaation kanssa. Spiraalimallissa ei määritellä iteraatioiden pituutta suhteessa ohjelmistokehitykseen vaadittuun aikaan. Mallissa painotetaan vahvasti prototyypin osuutta ohjelmiston kehityskaaren aikana. Varhainen prototyyppi mahdollistaa ohjelmiston testattavaksi, jotta virheitä voidaan löytää aikaisessa vaiheessa [4].

Spriraalimallin riskien analysoinnista huolimatta, ohjelmiston kehitysprosessi sisältää haasteita ohjelmiston vaatimusten hallintaan, laatuun ja erityisesti testaukseen liittyen. Spriraalimallissa ohjelmiston testausta tehdään prototyypin kehityskaaren lopussa ja palautetta asiakkaalta saadaan vasta iteraation lopussa. Spriraalimallissa ei ole korostettu vaatimuksien tärkeysjärjestystä eikä hallinointia [4].

Seuraavaksi tarkastelemme ketteriä kehitysmenetelmiä sekä miten nämä menetelmät ovat pyrkineet ratkaisemaan aikaisemmin käsiteltyjen suunnittelupainotteisten menetelmien puutteita ja miten ketterät menetelmät lähestyvät kappaleessa kaksi käsiteltyjä ohjelmistotuotannon haasteita ja ohjelmiston laatua.

Suunnittelupainotteisten prosessimallien ongelmien seurauksena useat ohjelmistoalan ihmiset ja organisaatiot kehittivät menetelmiä ja käytäntöjä,

joille muutokset ovat hyväksyttyjä. Menetelmiä kehitettiin useita ja eri maissa: [21]

- taipuisa järjestelmän kehitysmenetelmä (Dynamic Systems Development) Euroopassa
- toiminnallisuuspainotteinen kehitysmenetelmä (Feature-Driven Development) Australiassa
- ja XP (Extreme Programming) [2], Crystal [7], mukautuva ohjelmistokehitys (Adaptive Software Development) ja Scrum [20] Yhdysvalloissa.

Kaikilla ketterien menetelmien lähestymistavoilla oli tarkoitus välttää, lineaarista vaihe kerrallaan etenevää, suunnittelu- ja dokumentointipainotteisuutta [16]. Helmikuussa 2001 17 ketterien menetelmien kehittäjää tapasi keskustellakseen prosessimallien ja kokemuksiensa yhtäläisyyksistä [21].

Osallistujat määrittelivät käytännöt ketteriksi menetelmiksi ja kirjoittivat "Manifesto for Agile Software Development"-julistuksen, mikä kuvaa ketterän kehityksen perusarvojaa [21]:

- yksilöt ja vuorovaikutus ennen prosesseja ja työkaluja
- toimiva ohjelmisto ennen kattavaa dokumentaatiota
- asiakasyhteistyö ennen sopimusneuvotteluja
- muutoksiin vastaaminen ennen suunnitelman seuraamista.

Ohjelmistotuotannon parissa työskentelevät huomasivat, että ohjelmistoala erosi huomattavasti muista insinööritieteistä. Ohjelmiston valmistaminen on empiirinen prosessi, jonka lopputuloksena syntyy uusi tuote. Autojen kokoaminen on määriteltävä prosessi. Insinöörit voivat suunnitella prosessin, määritellä kokoonpanojärjestyksen sekä työntekijöiden, koneiden tai robottien toimenpiteet [21].

Empiirinen prosessi vaatii *tarkkaile ja mukaudu* (inspect and adapt) tyyppisen lähestymistavan. Oleellista oppia ja mukautua prosessin edetessä, eikä määritellä kaikkea alussa kattavasti. Lyhyet iteraatiot mahdollistavat ketteriä menetelmiä mukautumaan ja muuttamaa liiketoimintaympäristön ennustamattomien vaatimuksien mukaan. Lyhyt kehityssyklin jälkeen ohjelmistokehityksen suuntaa voidaan tarvittaessa nopeasti muuttaa [21].

Kerromme seuraavaksi lyhyesti kahdesta ketterästä kehitysmenetelmästä. Selvitämme miten suunnittelua on lähestytty näissä menetelmissä ja miten niiden käytänteet ovat pyrkineet ratkaisemaan ohjelmistotuotannon haasteita ja laatuun liittyviä ongelmia. Käsittelemme näiden ketterien menetelmien ratkaisuja toisessa kappaleessa esitettyihin ohjelmistotuotannon haasteisiin.

3.3 Extreme programming

XP (Extreme Programming) pienentää ohjelmistonkehityksen kustannuksia tekemällä jatkuvasti suunnittelua, määrittelyä, analysointia, ohjelmointia ja testausta ohjelmistokehityksen aikana. [2].

XP:ssä asiakas suunnittelee yhdessä kehittäjätiimin kanssa ohjelmiston toiminnallisuuksia, XP:ssä tarinoita (story), ja valitsee niistä eniten arvoa tuottavat toiminnallisuudet, jotka ovat arviotavissa ja testattavissa. Ohjelmoijat jakavat tarinat pieniksi käsiteltävän kokoisiksi ja ohjelmoitaviksi tehtäviksi (task). Ohjelmoijat suunnittelevat tehtävien pohjalta testejä, joiden avulla hän ymmärtää ohjelmistosta yhä enemmän. Nämä testit osoittavat läpäistessään tehtävän valmistuneen [2].

XP:n testauspainotteinen ohjelmistokehitys (test-driven development, TDD) poikkeaa merkittävästi suunnittelupainotteisista menetelmistä, joissa vasta valmista ohjelmistoa tai prototyyppiä testataan. Testauspainotteinen ohjelmakehityksessä uutta tuotantokoodia lisätään vasta, kun tuotantokoodille on olemassa yksi tai useampi testitapaus.

XP:n käytäntöihin kuuluu myös jatkuva pariohjelmointi (pair programming). Parin kanssa työskentelemällä ohjelmoijat ajavat testejä ja kehittävät samalla mahdollisimman yksinkertaista suunnitelmaa tehtävän ratkaisemiseksi [2].

3.4 Scrum

Scrum on empiirinen, tarkkaileva ja mukautuva prosessi, jonka edetessä sitä pyritään parantamaan saatujen havaintojen perusteella. Scrum perustaa kaiken käytännön iteratiiviselle ja inkrementaaliselle prosessille: Jokaisen kehityssyklin tulos on valmiin ohjelmistotuotteen kehitys ja mahdollisesti julkaistavissa oleva ohjelmisto. [20].

Scrum hallinnoi ohjelmistotuotannon monimutkaisuutta seuraavilla yksinkertaisilla käytänteillä: Iteraatioita kutsutaan sprinteiksi (sprint). Sprintti on 30 päivän iteraatio, jonka aika tuotetaan *valmiin* (done) määritelmän mukainen ohjelmisto. Määritelmästä valmis sopii kehittäjätiimi, mitä sillä kyseessä olevassa projektissa tarkoitetaan. Usein se tarkoittaa, että toiminnallisuudet tulee olla kattavasti testattu ja ohjelmakoodi rakenteellisesti selkeää ohjelmakoodia (clean code) [20].

Sprintti aloitetaan sprintin suunnittelupalaverilla (sprint planning meeting), jossa valitaan kehitettävät toiminnallisuudet. Sprintin lopussa pidetään sprinttikatselmus (sprint review meeting), jossa ohjelmistoversio esitellään sidosryhmille [20].

Scrumissa on kolme roolia: tuoteomistaja (Product owner), kehittäjätiimi (team) ja scrummaster (Scrum master). Tuoteomistaja on asiakasedustaja, joka rahoittaa ja visioi ohjelmistotuotteen. Kehittäjätiimi vastaa ohjelmiston toteuttamisesta ja scrummaster vastaa käytänteiden toteutumisesta ja

opastaa kehittäjätiimiä saavuttamaan tavoitteensa [20].

Scrum määrittelee neljä ohjelmistokehityksessä käytettävää tuotosta (artifact): tuotteen kehitysjono (product backlog), sprintin tehtävälista (sprint backlog), edistymiskäyrä (burndown chart) ja tuoteversio (increment of potentially shippable product functionality) [20].

Asiakkaalta saadut ohjelmistojärjestelmän vaatimukset listataan tuotteen kehitysjonossa, jota käytetään suunnittelussa ja vaatimusten hallinnassa koko ohjelmistokehitysprosessin ajan. Tuoteomistaja on vastuussa tuotteen kehitysjonon sisällöstä ja vaatimusten tärkeysjärjestyksestä [20].

Sprintin tehtävälista sisältää tehtävät, jotka kehittäjätiimi toteuttaa julkaistavaksi tuotteen toiminnallisuudeksi. Sprintin tehtävälista on läpinäkyvä reaaliaikainen kuvaus työstä, jonka kehittäjätiimi suunnittelee saavansa valmiiksi sprintin aikana. Tehtävälista pitää sisällään kunkin tehtävän kuvauksen, vastuullisen toteuttajan, tehtävän tilanteen (ei aloitettu, kesken, valmis), sekä jäljellä oleva arvioitu aika työtunteina. [20].

Edistymiskäyrä ilmaisee visuaalisesti jäljellä olevan työmäärän ja ajan suhdetta. Edistymiskäyrällä voidaan peilata todellista työn edistymistä ja nopeutta projektin suunnitelmiin ja toiveisiin [20].

Scrum vaatii kehitystiimiä rakentamaan uuden tuoteversion jokaisessa sprintissä. Tuoteversio on toimiva ohjelmisto, johon jokaisessa iteraatiossa on lisätty uusia täysin testattuja toiminnallisuuksia [20].

4 Ketterien menetelmien ratkaisuja

Seuraavaksi tarkastelemme, mitä kappaleessa 3.3 ja 3.4 käsiteltyjen ketterien menetelmien käytänteet tarjoavat ratkaisuna kappaleessa kaksi esitettyihin ohjelmistotuotannon haasteisiin ja laadunhallintaan.

XP:n käytäntöihin kuuluu jatkuva pariohjelmointi ja testauspainotteisuus. Nämä käytänteet poikkeavat muista kappaleessa kolme esitettyjen menetelmien käytänteistä, joten käsittelemme tarkemmin testauspainotteisuus kappaleessa 4.4 sekä pariohjelmointia kappaleessa 4.5. Tarkastelemme erityisesti niiden vaikutusta ohjelmiston laatuun.

4.1 Koordinointi ketterissä menetelmissä

Toisessa kappaleessa totesimme ohjelmistotuotannon haasteiden liittyvän koordinointiin, ja ongelman osa-alueiden olevan muuttuvien vaatimuksien lisäksi henkilöstön hallinta sekä ajallisten ja taloudellisten resurssien käyttö.

Ketterät menetelmät lähestyvät henkilöstöhallintaa suunnittelupainotteisia menetelmiä joustavammin ja ihmisläheisemmin: kehittäjätiimit ovat usein itseorganisoituvia ja he tekevät tekniset ratkaisut itse. Ketterät menetelmät hylkäävät ajatuksen, että ihmiset olisivat vaihdettavissa olevia osia. Yksilöiden ajatellaan olevan päteviä ammattilaisia, jotka osaavat suunnitella työnsä ja tietävät keinot saavuttaa paras tulos [11].

Ketterissä menetelmissä kehittäjien on kyettävä tekemään kaikki tekniset ratkaisut [11]. XP:ssä suunnitteluprosessin (planning game) aikana tiimi arvio omien kokemuksiensa perusteella toiminnallisuuksien kehittämiseen vaadittavan ajan [2].

Scrumissa kaikki projektin hallinnolliset vastuut on jaettu kolmen, kappaleessa 3.4 mainittujen, roolien kesken. Tuoteomistajan vastuu on esitellä kehittäjätiimille ja muille projektiin osallistuville tuotteelle asetettavat vaatimukset. Tuoteomistaja laatii alustavan vaatimusmäärittelyn, sijoitettavalle pääomalle asetettavat tavoitteet (ROI) ja julkaisusuunnitelmat (relese plans) [20].

Kehittäjätiimin vastuulla on toiminnallisuuksien kehittäminen. Ohjelmoijat päättävät, miten uusi toiminnallisuus toteutetaan ja työskentelevät rauhassan lopun iteraation ajan. Kehitystiimi on eri alojen asiantuntemuksesta koostuva itseorganisoituva ryhmä. Kehitystiimi on yhdessä vastuussa jokaisen iteraation onnistumisesta [20].

Kehittäjätiimi kokoontuu kokoontuu joka päivä 15 minuutin tapaamiseen - päiväpalaveriin (Daily Scrum). Jokainen tiimin jäsen vastaa kolmeen kysymykseen: Mitä olen tehnyt viimeisen tapaamisen jälkeen? Mitä ajattelin tehdä seuraavaksi? Mikä estää minua saavuttamasta tavoitteitani? Tapaamisen tarkoituksena on koordinoida tiimin työtä päivittäin ja sopia tarvittavista tapaamisista [20].

Scrummaster on vastuussa scrumprosessista. Hänen tehtävänä on esitellä scrumin periaatteet jokaiselle projektiin osallistuvalle. Scrummaster vastaa, että scrum sopii organisaation kulttuuriin ja toteuttaa odotetut hyödyt. Scrummaster valvoo, että jokainen toteuttaa scrumin käytänteitä: Tuoteomistaja hallinnoi ja priorisoi ohjelmiston toiminnallisuuksia tuotteen kehitysjonon kautta. Kehittäjätiimin tulee pitää päiväpalaverit sekä ylläpitää sprintin tehtävälistaa ja edistymiskäyrää. Kehittäjätiimille on taattava työrauha eikä tehtävälistan toiminnallisuudet ja niiden tärkeysjärjestys saa muuttua kesken meneneillä olevaa sprinttiä [20].

Ketterissä menetelmissä asiakas on läheisessä yhteistyössä koko ohjelmistotuotantoprosessin ajan. Jokaisessa iteraatiossa asiakas voi tarkistaa kehityksen vaihetta ja muuttaa sen suuntaa. XP:ssä asiakas on jatkuvasti paikalla (on-site customer). Jos kehittäjätiimille ilmenee kysymyksiä toteutuksesta tai toiminnallisuuden laajuudesta, ohjelmoijat voivat keskustella asiakkaan kanssa [2]. Tämä johtaa läheisempään asiakassuhteeseen ohjelmiston kehittäjien kanssa. Läheinen asiakassuhde on edellytys mukautuvan prosessin onnistumiselle [11].

Scrumissa sprinttikatselmuksesssa tiimi esittelee tuoteomistajalle ja muille halukkaille sidosryhmille, iteraation aikana, kehitetyt toiminnallisuudet. Tapaamisen tarkoituksena on tuoda ihmiset yhteen, esitellä ohjelmiston toiminnallisuudet ja auttaa osallistujia yhdessä päättämään projektin seuraavasta iteraatiosta [20].

Sprinttikatselmuksen jälkeen scrummaster ja tiimi pitää sprintin retros-

pektiivin (Sprint retrospective meeting). Kehittäjätiimi käy läpi sprintin aikaisen prosessin heikkouksia ja vahvuuksia, kehittääkseen sitä edelleen parempaan suuntaan seuraavaan iteraatioon [20].

4.2 Suunnittelu ketterissä menetelmissä ja muuttuvat vaatimukset

Suunnittelupainotteisissa menetelmissä suunnittelutyöhön kuluu kaavioita, joilla voidaan kuvailla ohjelmiston rakennetta ja käyttäytymistä. Ketterien menetelmien prosesseissa suunnittelussa painotetaan joustavuutta, jotta suunnitelmaa voidaan helposti muuttaa kun vaatimukset muuttuvat [10].

XP:ssä suunnitelmien ja kaavioiden merkitys on vähäinen: kaavioita tulee käyttää, jos niistä on hyötyä. Äärimäiset XP:n toteuttajat eivät käytä kaavioita lainkaan [10].

XP:n näkökulmasta kaavioiden tarkoitus on tarjota yhteydenpitoa. Tehokkaan yhteydenpidon takaamiseksi on piirrettävään kaavioon valittava tärkeät asiat ja vältettävä vähemmän tärkeitä. Vain merkitykselliset luokat sekä niiden tärkeimmät attribuutit ja operaatiot kuvataan kaavioon [10]. Ohjelmoinnin aikaista dokumentointia voidaan muuttuviin vaatimuksiin ja suunnitelmiin sopeuttaa seuraavasti: [10]

- Käytetään vain kaavioita, joita voidaan pitää ajan tasalla helposti
- Laitetaan kaaviot paikkaan, jossa ne ovat helposti nähtävillä
- Kannustetaan ihmisiä muuttamaan kaavioita
- Heitetään pois kaaviot, joita ihmiset eivät käytä.

Usein kaavioita käytetään välittämään tietoa eri ryhmien välillä. XP:tä toteuttaville kaaviot ovat toiminnallisuuksia muiden joukossa, joiden arvon määrää asiakas. Kaaviot ovat hyödyllisiä vain jos ne auttavat viestinnässä. Ohjelmakoodin varasto (repository) on yksityiskohtaisen tiedon lähde ja kaaviot koostavat ja korostavat tärkeitä asioita [10].

Ketterissä menetelmissä suunnittelua tehdään koko ohjelmistokehityskaaren aikana: jos ohjelmistokoodi on vaikea muuttaa, ei projektin aikana tehdä riittävästi rakenteeseen liittyvää suunnittelua. Ohjelmistokehityksen aikana ohjelmakoodi on pidettävä yksinkertaisena ja selkeänä [10]. Ohjelmakoodin rakennetta on tarpeen vaatiessa jatkuvasti parannettava rakennemuutoksilla (refactoring). Ohjelmoijien on ymmärrettävä suunnittelumallien (design patterns) tarjoamat ratkaisut sekä osattava käyttää niitä. Suunnittelumallit ovat hyväksi havaittuja ratkaisuja ohjelmistokehityksessä usein esiityviin suunnitteluongelmiin [10].

Ohjelmistoa on suunniteltava ottaen huomioon, että ohjelmakoodia tullaan myöhemmin muuttamaan. Ohjelmiston rakenteesta on osattava viestiä, ohjelmakoodin, kaavioiden ja ennen kaikkea keskustelun avulla [10].

4.3 Ketterät menetelmät ja ohjelmistojen laatu

Koska muodollinen suunnittelu dokumentaation muodossa on vähäistä ja suunnitelmia heitetään pois [10], on aiheellista käsitellä, miten ohjelmiston laatu otetaan huomioon ketterien menetelmien ohjelmistosuunnittelussa.

Kolmannessa kappaleessa määrittelimme laadun kansainvälisen standarointiorganisaation laatupiirteiden mukaisesti. Niistä toiminnallisuus, luotettavuus, käytettävyys ja tehokkuus yhdistyy asiakkaan toiveisiin ja näkemykseen tuotettavasta ohjelmistojärjestelmästä [14]. Laatupiirteet liittyvät vaatimusten hallintaan, joka on tärkeä osa sekä XP:tä [2] että scrumia [20]. Toimiva ja testattu ohjelmisto on asiakkaan nähtävissä koko kehityskaaren ajan, joten asiakas näkee täyttääkö ohjelmisto hänen tarpeensa [2].

Scrumissa tuoteomistajan näkemys tarvittavista toiminnallisuuksista on listattu tuotteen kehitysjonossa. Tuotteen kehitysjono on asiakkaan priorisoima: toiminnallisuudet jotka tuottavat arvoa ovat ylimpänä listassa. Muutokset tuotteen kehitysjonossa heijastavat muuttuvaa liiketoimintaympäristöä ja asiakas itse voi priorisoimalla vaikuttaa ohjelmiston vaatimuksiin sekä laatuun nähdessään siinä puutteita [20].

XP:ssä asiakaan laatimat toiminnalliset testit (functional test) osoittavat, että toiminnallisuudet ovat asiakkaan näkemyksen mukaisia. Toiminnalliset testit ovat asiakkaan määrittämiä ohjelman käyttöön liittyviä testejä, joilla hän vakuuttuu toiminnallisuuden oikeanlaisesta toteutuksesta [2].

Ketterissä menetelmissä on minimaalinen toimiva ohjelmisto valmis ensimmäisen iteraation jälkeen, sillä jokainen toteutettava toiminnallisuus tehdään valmiiksi. Asiakkaan käyttäessä toimivaa ohjelmistojärjestelmää, hän näkee mahdolliset puutteet toiminnallisuudessa ja käyttöliittymän käytettävyydessä. Puutteet voidaan korjata seuraavassa iteraatiossa [2].

Ketterät menetelmät ottavat huomioon laadunvarmistuksen asiakkaan vaatimusten osalta. Kansainvälisen standardointiorganisaation laatupiirteet ylläpidettävyys ja siirrettävyys liittyvät myös ohjelmakoodin sisäiseen rakenteeseen ja laatuun (design quality) [14].

Asiakkaalle ohjelmakoodin sisäisen laadun havaitseminen on vaikeampaa. Sisäinen laatu on asiakkaalle yhtä tärkeää kuin kehitystiimille, koska huonosti suunniteltua ohjelmistoa on kallista muuttaa. Asiakkaan tulee kuunnella kehitystiimi, ja jos he valittavat vaikeuksista tehdä muutoksia, on heille annettava aikaa korjata tilanne [10].

Tarkkailemalla poistettavan ohjelmakoodin määrää, voidaan nähdä tapahtuuko tarpeeksi suunnittelua. Ohjelmistoprojektissa, jossa tehdään riittävästi muutoksia rakenteeseen (refactoring), poistetaan tasaisesti huonoa ohjelmakoodia [10].

XP:ssä painotetaan yksinkertaista suunnitelmaa ja rakenteen jatkuvaa parantamista. Ohjelmoijat pyrkivät mahdollisimman selkeään ohjelmakoodiin. Jos ohjelmoijat näkevät toiminnallisuudelle paremman ratkaisun hyväksytysti ajettujen testien jälkeen, heidän tulee korjata ohjelmiston rakennetta [2].

Scrumissa kehittäjätiimin määrittelee, milloin toteutettu toiminnallisuus on *valmis*. Määritelmän mukaan toteutettu toiminnallisuus on oltava hyvin suunniteltu (well-structured) ja rakennettu (well-written code) ohjelmakoodi [20].

4.4 Ohjelmistojen testaus

Testaus on XP:n ydinkäytäntöjä: kehittäjätiimin kaikki ohjelmoijat kirjoittavat testitapauksia. Ohjelmoijat varmistavat osaltaan ohjelmakoodin toimivuuden eikä ohjelmakoodia siirretä erilliselle testauksesta vastaavaalle organisaatiolle tai jäsenelle.

XP.ssä testejä tehdään aina ennen varsinaista tuotantokoodia: ohjelmoija kirjoittaa ensin testitapauksen, ja sen jälkeen tarvittavan määrän tuotantokoodia testitapauksen ratkaisemiseksi. Ohjelmistossa läsnäolevat ja ohjelmakoodin kattavat testit varmistavat jatkuvan integraation (continuos integration) ja vaakaan rakennusprosessin [11]. Integroidessa uutta ohjelmakoodia ohjelmistoon, kaikki aikaisemmin tehdyt ja uudet testit ovat läpäistävä. Hyväksytysti ajettujen testien jälkeen voidaan muutokset hyväksyä ja uusi versio julkaista [2].

Asiakas laatii kehittäjätiimin avulla hyväksymätestit (acceptance test), joiden avulla hän vakuutuu osaltaan uuden tarinan onnistuneesta toteutuksesta: Hyväksymätesti kattaa toiminnallisuuden testauksen koko ohjelmistojärjestelmätasolla kaikkiin tarvittaviin ohjelmisto-osiin käyttöliittymästä tietokantaan.

Hyväksymätestit käyttävät järjestelmää ohjelmallisesti käyttäjän tavoin: painavat käyttöliittymän painikkeita tai syöttävät tekstikenttiin tietoa sekä lukevat käyttäjälle näytettävää tulostetta. Oikeat syötteet ja tulosteet kertovat järjestelmän toiminnasta sekä takaavat ohjelmoijille ja sidosryhmille, että ohjelmisto toimii vaatimusten mukaisesti. Testit ovat ajettavissa koko ohjelmiston kehityskaaren ajan ja ne takaavat ohjelmiston toimivuuden jatkuvasti. Toiminnallisuuksia lisättäessä tai ohjelmakoodin rakennetta muutettaessa testit kertovat onko uusi versio toimiva järjestelmä [2].

Scrumissa on kaikilla projektiin osallistuvien on yhteisesti sovittava määritelmästä *valmis* toiminnallisuus. Valmiin määritelmän on vastattava organisaation standardeja, käytäntöjä ja ohjeita. Kun sprinttikatselmuksessa esitetään valmis toiminnallisuus, sen on oltava tämän sovitun määritelmän mukainen. Tuoteomistaja voi vaatia, että toiminnallisuuden valmistuessa se on oltava julkaistavissa. Toiminnallisuuden ohjelmakoodin täytyy olla tällöin hyvin rakennettu (well-structured code) ja kattavasti testattu [20].

Winston Royce kirjoitti artikkelissaan ”Managing the development of large software systems”, että ohjelmoijan ei tule testata kirjoittamaansa ohjelmakoodia. Royce arvioi, että useimmat virheet ovat ilmiselviä ja ovat löydettävissä katsomalla ohjelmakoodia [19]. XP:ssä tämä on hyväksytty tosiasia ja ongelmaa on lähestytty työskentelemällä jatkuvasti pareittain,

jolloin ohjelmoijat testaavat ja arvioivat toistensa ohjelmakoodia [2].

XP:n testauspainotteisessa menetelmässä testitapaukset määrittävät ohjelmiston haluttua käyttäytymistä ja osoittavat oikein toteutetut toiminnallisuudet. Testauspainotteisessa kehityksessä toiminnallisuutta lisätään inkrementaalisesti ja valmiiksi toteutetut sekä vielä keskeneräiset toiminnallisuudet ovat tiimille selvillä koko kehitysprosessin ajan [9].

Testauspainotteinen menetelmä eroaa merkittävästi suunnittelupainotteisista menetelmistä, joissa ohjelmiston testaus tapahtuu vasta ohjelmistokehityskaaren lopussa. Tästä johtuen on tarpeen tarkastella tarjoaako testauspainotteinen ohjelmistokehitys ratkaisuja ohjelmistotuotannon haasteisiin ja parantaako testauspainotteisuus ohjelmiston laatua. Käsittelemme seuraavaksi löytämiämme tutkimuksia ja niiden tuloksia testauspainotteisuuden hyödyistä.

Tietojenkäsittelytieteen opiskelijoille tehdyssä tutkimuksessa opiskelijat kävivät vuonna 2003 ohjelmointikurssin noudattaen testauspainotteista menetelmää. Vertailuryhmänä käytettiin samaa kurssia vuodelta 2001, jolloin opiskelijat eivät käyttäneet testauspainotteista menetelmää. Tulokset osoittivat, että testauspainotteista menetelmää käyttäneillä opiskelijoilla oli keskimäärin 45% vähemmän virheitä [9].

IBM kokeili eräässä ohjelmistokehitysryhmässä testauspainotteista kehitysmenetelmää, kun aikaisemmat laadunvarmistukset eivät tuottaneet toivottua tulosta. Käyttämällä testauspainotteista menetelmää virheet vähenivät noin 50% aikaisempaan verrattuna [17].

Toisessa tutkimuksessa 24 ohjelmistoalan ammattilaista jaettiin kahteen ryhmään pareittain toimiviksi tiimeiksi. Toinen ryhmistä teki ohjelmistokehitystä testauspainotteisesti ja toinen ohjelmoi vesiputousmallin mukaisesti. Testipainotteisesti ohjelmaa kehittäneet parit tuottivat laadukkaampaa ohjelmakoodia. Testauspainotteisen ryhmän ohjelmakoodi läpäisi 18% enemmän testitapauksia kuin kontrolliryhmän ohjelmakoodi [12].

Yllä mainitut tutkimukset osoittavat, että testauspainotteinen ohjelmistokehitys parantaa ohjelmiston laatua kun tarkastellaan ohjelmistossa esiintyviä virheiden määrää ja koodikattavuutta (code coverage). Koodikattavuus kertoo, kuinka paljon tehdystä tuotantokoodista on testattu. Tehdyissä kyselyissä testauspainotteiseissa ryhmissä osallistuneet olivat luottavaisempia tekemistään ratkaisuista [12].

4.5 Pariohjelmointi

Pariohjelmointi on toinen käytäntö, mikä erottaa XP:n muista ohjelmistotuotannon menetelmistä. Pariohjelmoinnin vaikutuksia haasteisiin ja laatuun on syytä tarkastella lähemmin. Kappaleessa 2.1 toimme haasteista esille muun muassa henkilöstön hallinnan, aikataulun ja taloudellisten resurssien käytön. Kappaleessa 2.4 käsittelimme ohjelmiston laatua. Tarkastelemme seuraavaksi pariohjelmoinnista löytämiämme tutkimuksia ja niiden tuloksia erityisesti

laatuun sekä hieman myös kustannuksiin liittyen.

Pariohjelmoinnissa kaksi ohjelmoijaa yhdessä työstävät yhtä ohjelmakoodia, algoritmia tai suunnitelmaa. Toinen parista ohjelmoi ja toinen heistä tarkkailee etsien virheitä, miettien vaihtoehtoja, tutkien lähteitä ja miettien erilaisia toteutustapoja. Parit vaihtavat roolejaan ajoittain. Molemmat ovat tasavertaisia ja aktiivisia osallistujia [22].

Vuonna 1998 John Nosek teki tutkimuksen, jossa kokeneet ohjelmoijat työskentelivät haastavien, omalle organisaatiolleen tärkeiden, tehtävien parissa omassa työskentely-ympäristössään. Kukaan osallistujista ei ollut työskennellyt annetun tehtävän kaltaisen ongelman parissa aikaisemmin. Annetun tehtävän kaltaista ongelmaa pidettiin organisaatiolle menestykselle tärkeänä ja niin vaativana, että yleensä tehtäviin palkattiin ulkopuolisia konsultteja [18].

Koehenkilöt valittiin satunnaisesti työskentelemään pareittain testiryhmään ja yksilöinä kontrolliryhmään. Ryhmiltä tehtäviin kulunut aika mitattiin. Ratkaisuista pisteytettiin luettavuus väliltä 0-2. Lukuarvo 0 tarkoitti lukukelvotonta ratkaisua ja 2 täysin luettavissa olevaa ratkaisua. Ratkaisun toimivuus pisteytettiin väliltä 0-6. Lukuarvo 0 merkitsi, että ratkaisu ei saavuttanut annettua tehtävää lainkaan. Täysin toimiva ratkaisu pisteytettiin arvolla 6. Kokonaispistemäärän maksimiarvo oli 8, joka oli luettavuuden ja toimivuuden summa [18].

Pareittain työskentelevät saivat keskimäärin kokonaispistemääräksi 7,6 ja aikaa kului 30,2 minuuttia. Vertailuryhmän keskimääräinen kokonaispistemäärä oli 5,6 ja tehtävään aikaa kului 42,6 minuuttia [18].

Vuonna 1999 Utahin yliopiston tietojenkäsittelytieteen opiskelijat osallistuivat tutkimukseen. Opiskelijat jaettiin kahteen ryhmään. Kolmetoista opiskelijaa muodosti kontrolliryhmän, jossa opiskelijat työskentelivät itsenäisesti kaikissa annetuissa tehtävissä. 28 opiskelijaa muodosti testiryhmän, jossa opiskelijat muodostivat kahden hengen ryhmän. Kokeilu vertaili tehtävistä suoriutumiseen vaadittua aikaa, tuottavuutta ja suoritettujen tehtävien laatua ryhmien välillä. Ohjelmille suoritettiin automaattiset testit ohjelmointityön laadun arvioimiseksi. Taulukossa 1 on tutkimustulos Utahin opiskelijoille tehdystä pariohjelmoinnista [22]: Taulukko 1. Läpäistyt testitapaukset prosentteina

Tehtävä	Yksin työskentelevät	Pareittain työskentelevät
Ohjelma 1	73,4	86,4
Ohjelma 2	78,1	88,6
Ohjelma 3	70,4	87,1
Ohjelma 4	78,1	94,4

Monet opiskelijat olivat epäluuloisia pariohjelmoinnin hyödyistä: he pohtivat paljonko ylimääräistä kommunikaatiota vaaditaan, miten he sopeutuvat toistensa työskentelytapoihin, ohjelmointityyliin ja miten heidän egonsa vaikuttavat työskentelyyn, sekä miten erimielisiä he ovat tehtävien toteutuksista.

Tosiasiassa ohjelmoijat käyvät läpi siirtymäajan yksinäisestä työskentelystä yhteisölliseen työskentelytapaan. Siirtymäajan kuluessa he oppivat sopeuttamaan toimintojaan käyttämään hyväksi vahvuuksiaan ja välttämään heikkouksia. Tuloksena ryhmän tuottavuus ylittää ryhmän yksilöiden tuottavuuden summan [22].

Nosekin tekemässä tutkimuksessa pareittain työskentelevät käyttivät, työskennellessään rinnakkain, yhteensä 60% enemmän ohjelmointiaikaa kuin yksin työskentelevät [18]. Utahin opiskelijoille tehdyssä tutkimuksessa saatiin samankaltaisia tuloksia: keskimäärin pareittain työskenteleviltä vaati yhteensä 60% enemmän ohjelmointiaikaa annetusta tehtävästä suoriutumiseen [22].

Siirtymäajan jälkeen pareittain työskentelevät opiskelijat paransivat tuloksiaan. Pareittain työskenteleviltä vaadittu ohjelmointiaika oli enää 15% suurempi kuin yksin työskentelevillä. Tässä vaiheessa opiskelijat olivat tottuneet pariohjelmointiin ja toistensa työskentelytapoihin [22].

Laurie Williamsin, Ward Cunninghamin ja Ron Jeffriesin haastattelemat ohjelmoijat sanoivat, että pareittain analysointi ja suunnittelu on tärkeämpää kuin toiminnallisuuden toteuttaminen. Ohjelmoijat usein toteuttavat yksilöllisesti rutiinitehtäviä ja yksinkertaisia ohjelmakoodeja. Tällaisten tehtävien toteuttaminen yksilöllisesti tehtynä tehokkaampaa [22].

Pareittain työskentelevät tuottavat luettavampaa ohjelmakoodia ja toimivampia ratkaisuja kuin yksin toimivat ohjelmoijat. Ryhmissä toimivat ratkaisevat ongelmia keskimäärin nopeammin kuin yksilöt. Lisäksi pareittain toimivat ilmaisevat korkeampaa luottamusta ratkaisuunsa ja kokevat nauttivansa prosessista enemmän kuin yksilöinä ohjelmoivat [18].

Pareittain ohjelmoivat ihmiset tekevät erityisesti parempaa suunnittelua ja analyysia kuin yksilölliset ohjelmoijat. Pari harkitsee huomattavasti enemmän vaihtoehtoja ja yhtyvät nopeasti toteutettavaan ratkaisuun. Ideoiden vaihto parin välillä vähentää huonon suunnitelman todennäköisyyttä. Yhdessä työskentelemällä pari voi toteuttaa tehtäviä, jotka voivat olla liian haastavia yhdelle. Parityöskentely pakottaa osallistujia keskittymään täysin haasteena olevaan tehtävään [22].

Pariohjelmoinnin kustannukset ovat oleellinen asia. Voisi olettaa, että pariohjelmoinnin sisällyttäminen ohjelmistotuotantoon kaksinkertaistaa kustannukset jos henkilöstömäärää on lisättävä samassa suhteessa. John Nosekin [18] sekä Williamsin, Kesslerin, Cunninghamin ja Jeffriesin [22] aiheesta tekemät tutkimukset osoittavat, että pareittain työskentelevät tiimit suoriutuvat tehokkaammin kuin yksittäiset ohjelmoijat. Tämä huomioden kustannukset eivät nouse suoraan verrannollisesti henkilöstömäärän kanssa. Laadunvarmistuksesta aiheutuvat kustannukset huomioiden pareittain ohjelmointi saattaa olla kustannustehokkaampaa kuin yksilöllisesti ohjelmoitaessa [8].

5 Yhteenveto

Tässä tutkielmassa kävimme läpi ohjelmistotuotannon haasteita ja ohjelmiston laatuun liittyviä määritelmiä. Selvitimme miten suunnittelupainotteiset ja ketterät menetelmät ovat lähestyneet ohjelmistotuotantoa. Tarkastelimme mitä ketterät menetelmät ovat tarjonneet ratkaisuna haasteille sekä miten laadunhallinta otetaan huomioon ketterissä menetelmissä, ja onko ketterien menetelmien käytänteillä ollut vaikutusta ohjelmistojen laatuun.

Ketterien menetelmien käytänteistä ja vaikutuksesta laatuun käsittelimme testauspainotteisuutta ja pariohjelmointia. Totesimme, että pariohjelmointi on usein jopa kustannustehokkaampaa kuin yksin ohjelmoitaessa, jos huomioidaan laadunvarmistuksen kustannukset. Pariohjelmointi tuottaa paremmin suunniteltua ohjelmakoodia ja parempia ratkaisuja kuin yksin ohjelmoitaessa.

Testauspainotteisuuden totesimme tuottavan laadukkaampaa ohjelmakoodia kuin vesiputousmallin mukaisesti ohjelmoitaessa. Testipainotteisesti ohjelmoineet ovat luottavaisempia omaan ratkaisuunsa.

Ohjelmistojen kehittäminen internet-aikakaudella vaatii joustavia kehitysmenetelmiä, jotka sopeutuvat nopeasti vaihtuviin vaatimuksiin ja vaativiin markkinoihin. Ketterät menetelmät sopeutuvat perinteisiä menetelmiä paremmin tällaisiin ympäristöön.

Suunnittelupainotteisissa menetelmissä ajatuksena on, että riittävällä työllä voidaan ennakoida vaatimukset täydellisesti sekä alentaa ohjelmistotuotantoon liittyviä kustannuksia vähentämällä muutoksia. Ketterät menetelmät pyrkivät alentamaan uudistusten kustannuksia [21]. Ketterät menetelmät ovat valmiimpia muutoksille. Liiketoimintaympäristö vaatii sekä odottaa innovatiivisia, korkealuokkaisia ohjelmistoja, jotka vastaavat niiltä odotettuihin vaatimuksiin [1].

Ketterät menetelmät tarjoavat vaatimusten hallinnalla ja kevyillä organisatorisilla rakenteilla ratkaisuja ohjelmistotuotannon haasteisiin ja ohjelmiston laadunhallintaan. Onnistunut ohjelmistokehityksen aikainen rakenteen suunnittelu parantaa ohjelmiston sisäistä laatua. XP:n jatkuva testaus antaa mahdollisuuden toteuttaa rakennemuutoksia toiminnallisuuksia rikkomatta. Testauspainotteisella ohjelmistokehityksellä voidaan parantaa sekä ohjelmiston sisäistä että käyttäjän kokemaa laatua. Pariohjelmoinnilla ohjelmiston suunnittelu ja laatu tehostuu merkittävästi.

6 Lähteet

[1] Baskerville, R., B. Ramesh, L. Levine, J. Pries-Heje ja S. Slaughter: *Is "Internet-speed" software development different?* Software, IEEE, 20(6):70 – 77, nov.-dec. 2003, ISSN 0740-7459.

[2] Beck, K.: *Embracing change with extreme programming.* Computer, 32(10):70–77, oct 1999, ISSN 0018-9162.

[3] Boehm, B.: *Get ready for agile methods, with care.* Computer, 35(1):64 –69, jan 2002, ISSN 0018-9162.

[4] Boehm, B. W.: *A spiral model of software development and enhancement.* Computer, 21:61–72, may 1988, ISSN 0018-9162.

[5] Boehm, B. W., J. R. Brown ja M. Lipow: *Quantitative evaluation of software quality.* Teoksessa *Proceedings of the 2nd international conference on Software engineering*, ICSE '76, sivut 592–605, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press. `http://dl.acm.org/citation.cfm?id=800253.807736`.

[6] Boehm, Barry: *A view of 20th and 21st century software engineering.* Teoksessa *Proceedings of the 28th international conference on Software engineering*, ICSE '06, sivut 12–29, New York, NY, USA, 2006. ACM, ISBN 1-59593-375-1. `http://doi.acm.org/10.1145/1134285.1134288`.

[7] Cockburn, Alistair: *Crystal clear: a human-powered methodology for small teams.* Addison-Wesley Professional, 2005.

[8] Cockburn, Alistair ja Laurie Williams: *The costs and benefits of pair programming.* Extreme programming examined, sivut 223–247, 2000.

[9] Edwards, Stephen H: *Using test-driven development in the classroom: Providing students with automatic, concrete feedback on performance.* Teoksessa *Proceedings of the International Conference on Education and Information Systems: Technologies and Applications EISTA*, nide 3, 2003.

[10] Fowler, Martin: *Is design dead?* SOFTWARE DEVELOPMENT-SAN FRANCISCO-, 9(4):42–47, 2001.

[11] Fowler, Martin: *The new methodology.* Wuhan University Journal of Natural Sciences, 6:12–24, 2001. `http://dx.doi.org/10.1007/BF03160222`.

[12] George, Boby ja Laurie Williams: *An initial investigation of test driven development in industry.* Teoksessa *Proceedings of the 2003 ACM symposium on Applied computing*, sivut 1135–1139. ACM, 2003.

[13] Highsmith, J. ja A. Cockburn: *Agile software development: the business of innovation.* Computer, 34(9):120 –127, sep 2001, ISSN 0018-9162.

[14] Kitchenham, B. ja S.L. Pfleeger: *Software quality: the elusive target [special issues section].* Software, IEEE, 13(1):12–21, 1996, ISSN 0740-7459.

[15] Kraut, Robert E. ja Lynn A. Streeter: *Coordination in software development.* Commun. ACM, 38:69–81, mar 1995, ISSN 0001-0782. http://doi.acm.org/10.1145/203330.203345.

[16] Larman, C. ja V.R. Basili: *Iterative and incremental developments. a brief history.* Computer, 36(6):47 –56, june 2003, ISSN 0018-9162.

[17] Maximilien, E Michael ja Laurie Williams: *Assessing test-driven development at IBM.* Teoksessa *Software Engineering, 2003. Proceedings. 25th International Conference on*, sivut 564–569. IEEE, 2003.

[18] Nosek, John T.: *The case for collaborative programming.* Commun. ACM, 41(3):105–108, mar 1998, ISSN 0001-0782. http://doi.acm.org/10.1145/272287.272333.

[19] Royce, Winston W: *Managing the development of large software systems.* Teoksessa *proceedings of IEEE WESCON*, nide 26. Los Angeles, 1970.

[20] Schwaber, Ken: *Agile project management with Scrum.* Microsoft Press, 2009.

[21] Williams, L. ja A. Cockburn: *Agile software development: it's about feedback and change.* Computer, 36(6):39–43, june 2003, ISSN 0018-9162.

[22] Williams, Laurie, Robert R Kessler, Ward Cunningham ja Ron Jeffries: *Strengthening the case for pair programming.* Software, IEEE, 17(4):19–25, 2000.