Tiitus Kivikangas

TEKNINEN VELKA ERILAISISSA OHJELMISTOKEHITYSTYYPEISSÄ

JYVÄSKYLÄN YLIOPISTO
INFORMAATIOTEKNOLOGIAN TIEDEKUNTA
2020

TIIVISTELMÄ

Kivikangas, Tiitus
Tekninen velka erilaisissa ohjelmistokehitystyypeissä
Jyväskylä: Jyväskylän yliopisto, 2020, 21 s.
Tietojärjestelmätiede, kandidaatin tutkielma
Ohjaaja: Clements, Kati

Tämä kandidaatin tutkielma on tehty kirjallisuuskatsauksena. Tavoitteena oli tarkastella tieteellisten julkaisujen avulla teknisen velan eroavaisuuksia perinteisten ja ketterien ohjelmistokehitysmenetelmien välillä. Aihe on mielenkiintoinen, sillä näissä ohjelmistokehitysmenetelmissä teknistä velkaa lähestytään eri näkökulmista. Perinteisissä menetelmissä teknistä velkaa pyritään välttämään mittavalla suunnittelulla ja perusteellista työtä tekemällä. Ketterissä menetelmissä puolestaan kehitystahtia nopeutetaan tasapainottelemalla nopean kehityksen ja teknisen velan maksun välillä. Velkaa otetaan etenkin kehityksen alkuvaiheessa. Kirjallisuuskatsauksen pohjalta todettiin, että teknistä velkaa ei pystytä välttämään millään kehitysmenetelmällä. Velan takaisinmaksu tulisi olla suunnitelmallista, jotta velka ei kasvaisi liian suureksi ja tuhoaisi kehitystyötä.

Asiasanat: tekninen velka, ketterät menetelmät, perinteiset menetelmät, ohjelmistokehitys

ABSTRACT

Kivikangas, Tiitus
Technical debt in different software development methods
Jyväskylä: University of Jyväskylä, 2020, 21 p.
Information Systems, Bachelor's Thesis
Supervisor: Clements, Kati

This bachelor's thesis is conducted as a literature review. The point of the study was to review differences in approaches to technical debt in traditional and agile software development methods through scientific literature. The subject is interesting as these development methods view technical debt from different perspectives. Traditional methods aim to avoid technical debt by planning before development and taking every step thoroughly. In agile methods development speed is pursued by balancing faster development and technical debt payment. Technical debt is very prominent in early stages of development. Based on the literature review technical debt is unavoidable by any development method. Technical debt repayment should be systematical so that the debt would not gain interest and become fatale to the development.

Keywords: technical debt, agile methods, traditional methods, software development

KUVIOT

TAULUKOT

SISÄLLYS

1 JOHDANTO

Tässä kirjallisuuskatsauksessa tarkastellaan teknistä velkaa erilaisten ohjelmistokehitysmenetelmien näkökulmasta. Teknisen velan tutkimus on alkanut 1990-luvulta ja vuoden 2008 jälkeen sen tutkimus on kasvanut räjähdysmäisesti (Li, Avgeriou & Liang, 2015). Teknisen velan määritelmä on ensimmäisen ulostulonsa jälkeen supistunut ja laajentunut ja nykyisin sillä voidaan tarkoittaa lähes kaikenlaista velkaa tai epäkohtia ohjelmiston kehityksessä (Kruchten, 2012; Buschmann, 2011). Tekninen velka on tahallisia ja tahattomia virheitä, joita yritys tekee ohjelmistoa kehittäessä (Buschmann, 2011). Näiden virheiden hinta kasvaa, kunnes ne on korjattu (Cunningham, 1992). Maailmanlaajuisen teknisen velan hinnan on arvioitu olevan 500 miljardia dollaria vuonna 2010 ja kaksinkertaistuvan vuoteen 2015 (Tom, 2013). Erilaiset ohjelmistokehitystyypit jaetaan tyypillisesti perinteisiin ja ketteriin kehitysmenetelmiin. Perinteisillä kehitysmenetelmillä tarkoitetaan yleensä ylhäältä alaspäin suuntautuvaa vaiheittaista kehitystä (Boehm & Turner, 2005). Ketterät ohjelmistokehitysmenetelmät noudattavat agile manifeston periaatteita, joita on myöhemmin täydennetty (Williams, 2012). Tässä kirjallisuuskatsauksessa määritellään perinteisistä ohjelmistokehitysmenetelmistä vesiputousmalli ja spiraalimalli. Ketteristä ohjelmistokehitysmenetelmistä puolestaan määritellään Scrum, Lean ja ExtremeProgramming (XP). Teknistä velkaa tarkastellaan tässä kirjallisuuskatsauksessa näiden viiden ohjelmistokehitysmenetelmän näkökulmasta ja vertaillaan teknisen velan suhdetta perinteisiin ja ketteriin kehitysmenetelmiin.

Tämän kirjallisuuskatsauksen tavoitteena on selvittää eroavaisuuksia perinteisten ja ketterien ohjelmistokehitysmenetelmien välillä teknisen velan kontekstissa. Tutkimuskysymykseksi on asetettu:

> *Kuinka perinteisten ja ketterien ohjelmistokehitysmenetelmien tekninen velka eroavat toisistaan?*

Tutkimuskysymykseen vastataan kirjallisuuskatsauksena. Kirjallisuutta etsitään Google Scholar ja JykDok palveluista sekä aiheen artikkelien viittauksista. Keskeisinä hakusanoina käytetään seuraavia hakusanoja: ”technical debt”, “software business”, “software development”, “software risks”, “agile methods” ja

“traditional methods”. Artikkelien merkityksellisyys arvioidaan sitaattien, julkaisuajan ja julkaisun merkityksellisyyden perusteella käyttäen Julkaisufoorumipalvelua. Aihealueesta löytyi kirjallisuutta paljon, mutta suuri osa kirjallisuudesta on matalamman tason julkaisuja. Vertailevan tutkimuksen löytäminen oli myös haasteellista, sillä suuri osa ketterien kehitysmenetelmien julkaisuista sijoittui 10–20 vuotta perinteisten menetelmien tutkimusten jälkeen.

Tutkielman toisessa luvussa ja sen alaluvuissa määritellään ja perehdytään tekniseen velkaan ja esitellään sen tyyppejä ja hallintamenetelmiä. Seuraavassa luvussa käsitellään ohjelmistokehityksenmenetelmiä. Ensimmäisessä alaluvussa esitellään perinteiset ohjelmistokehitysmenetelmät ja perehdytään tarkemmin vesiputosmalliin sekä spiraalimalliin. Toisessa alaluvussa esitellään taas ketterä ohjelmistokehitys ja esitellään tarkemmin Scrum, Lean ja ExtremeProgramming ohjelmistokehitysmenetelmät. Neljännessä luvussa esitellään tulokset. Viimeisessä viidennessä luvussa tehdään tutkimuksesta yhteenveto ja pohditaan mahdollisia jatkotutkimusaiheita.

2 TEKNINEN VELKA

Teknisen velka tutkimus voidaan laskea alkaneeksi Cunninghamin 1992 julkaistun artikkelin myötä. Termiä ja ilmiötä kuitenkin tutkittiin hyvin vähän seuraavan 15 vuoden ajan, kunnes julkaistujen tutkimusten määrä kasvoi räjähdysmäisesti vuoden 2008 jälkeen. (Li ym., 2015.)

2.1 Määritelmä

Tekninen velka esiteltiin 1990-luvulla kielikuvaksi, jolla kuvataan heikon ohjelmistokehityksen kautta kerääntyvää velkaa. Velka nopeuttaa ohjelmistokehitystä, mutta kasvaa korkoa, kunnes se on korjattu eli takaisinmaksettu. Jokainen minuutti velkaannuttavalla ei-aivan-oikeanlaisella koodilla lisää korkoa, kunnes velka on maksettu (Cunningham, 1992). Tämän ensimmäisen esittelyn jälkeen teknisen velan termi on elänyt ja sitä on laajennettu ja supistettu useiden eri tutkimusten kautta. Nykyään on yleistä, että teknisen velan termiä käytetään laajemmin kuvaamaan kaikenlaista velkaa tai epäkohtia ohjelmistokehityksessä kattaen lähes kaiken, joka estää ohjelmiston myynnin, kehityksen tai käyttöönoton tai kaiken, joka lisää kitkaa, josta ohjelmistokehitys kärsii, kuten testi-, henkilöstö-, arkkitehtuuri-, vaatimus-, dokumentointi- tai vain epämääräinen kaikenkattava ohjelmistovelka. (Kruchten, 2012.) Velka voi johtua tahallisista tai ei tahallisista tekijöistä eri tasoilla ohjelmistokehitystä. (Buschmann, 2011).

Tekninen velka, kuten muukaan velka ei kuitenkaan ole aina huono asia. Harvalla on varaa ostaa taloa käteisellä, mutta asuntolainan ottaminen ei ole vastuutonta, kunhan osaa maksaa velan takaisin. Samoin yksinkertaisen, mutta hitaan algoritmin käyttö prototyypissä voi olla aivan oikea ratkaisu, kunhan suunnitellaan valmiiksi, kuinka koodi päivitetään ennen julkaisua. (Allman, 2012.) Toisinsanottuna tekninen velka voi olla strategia, jolla saavutetaan nopeasti kaupallisia tavoitteita ottamalla teknistä velkaa (Wolff & Johann, 2015).

Tekninen velka eroaa eniten tavallisesta velasta siten, että velan ottaja ei välttämättä ole se, joka joutuu maksamaan sen takaisin. Usein velan ottaja voi siirtää syntyneet kulut muille (Allman, 2012).

2.2 Teknisen velan tyypit

Tom (2013) löysi tutkimuksessaan tekniselle velalle viisi eri ulottuvuutta: koodi-, suunnittelu- ja arkkitehtuuri-, ympäristö-, dokumentaatio- ja testausvelka. Li ym. (2015) löysivät systemaattisessa kirjallisuustutkimuksessaan kymmenen eri tyyppiä: vaatimus-, arkkitehtuuri-, suunnittelu-, koodi-, testaus-, rakenne-, dokumentaatio-, infrastruktuuri-, versiohallinta- ja virhevelka. Nämä luokat jakaantuivat edelleen pienemmiksi. (Li ym., 2015.) Alves ym. (2014) löysivät

kirjallisuustutkimuksessaan viisitoista tyyppiä. Li ym. havainnoimien tyyppien lisäksi tutkimuksessa löydettiin: käytettävyys-, henkilöstö-, prosessi-, testiautomaatio- ja palveluvelka. (Alves ym., 2014.) Uudemmassa vuonna 2018 julkaistussa tutkimuksessa löydettiin 13 tyyppiä. Uusia tyyppejä olivat tietokanta- ja suorituskykyvelka (Benidris, 2018). Yhteistä kaikille tutkimuksille on se, että Li ym. (2015) tutkimuksessa havainnoidut tyypit löytyvät myös muissa tutkimuksissa, joten tässä tutkielmassa käytetään näitä kymmentä tyyppiä.

Kun paine toimittaa asiakkaille ohjelmistotuotteet nopeammin kasvavat, projektipäälliköiden, jotka ovat velvollisia saavuttamaan määräajat ja lyhyen tähtäimen liiketoiminnan edut on tapana painostaa ohjelmoijia. Tämän takia nämä ohjelmoijat tekevät vajavaista ja virheellistä koodia sekä väliaikaisratkaisuja, jotka aiheuttavat vikoja ja joita pitää uudelleen työstää, jotta ne vastaavat asiakkaiden vaatimuksia laadukkaasta ja kestävästä ohjelmistosta. Näitä virheitä voidaan pitää tahallisina ohjelmistokehitystiimin virheinä. (Mensah, Keung, Svajlenko, Bennin & Mi. 2018.)

2.3 Teknisen velan hallinta

Helppo ratkaisu teknisen velan haittavaikutuksiin olisi maksaa velka pois ennen kuin ongelman ilmenevät, mutta alan kilpailun, tiukkojen aikataulujen ja asiakastyytyväisyyden takia tämä on usein mahdotonta. Tämän takia on tärkeää havainnoida ja kehittää prosesseja, joiden avulla yritykset voivat elää teknisen velan kanssa ja joiden avulla voidaan tietää kuinka, mikä ja milloin tekninen velka tulisi maksaa. Teknisen velan hallinta sisältää aktiviteetteja, prosesseja, tekniikoita ja työkaluja, joilla voidaan havainnoida, mitata, ennaltaehkäistä ja vähentää teknistä velkaa ohjelmistoissa. (Yli-Huumo, 2016.) Samanlaisia teknisen velan aiheuttamia aktiviteetteja löysi myös Li ym. (2015). Hän jakoi aktiviteetit luokkiin: tunnistaminen, mittaaminen, priorisointi, estäminen, seuraaminen, takaisinmaksu, dokumentointi ja kommunikointi.

Velan hallinnointia ja takaisinmaksua hankaloittaa se, että suuri osa järjestelmistä, jotka kärsivät velasta, ovat myös tuottavassa toiminnassa ja täten niillä on arvoa, jota ei tulisi tuhota. Teknisen velan kanssa toimiessa on aina vaara tehdä enemmän haittaa kuin hyötyä. (Buschmann, 2011.)

Teknisen velan hallintaan kuluu merkittävästi resursseja. Martini (2018) tekemän kyselyn mukaan ohjelmistoalan ammattilaiset arvioivat, että suurten ohjelmistoyritysten projekteissa keskimäärin noin 25 % resursseista kuluu teknisen velan hallinnointiin. Tästä huolimatta vain neljännes käyttää työkaluja teknisen velan seuraamiseen.

3 OHJELMISTOKEHITYKSEN MENETELMÄT

Ohjelmistokehityksen menetelmät luovat perusrungon ohjelmistokehityksen työjärjestykselle ja organisoinnille. ”Päätarkoitus ohjelmistokehittämisen mallille on päättää järjestys vaiheille, jotka ovat osana ohjelmistokehitystä ja arvioida ja asettaa kriteerit siirtymiselle vaiheesta seuraavaan” (Boehm, 1988).

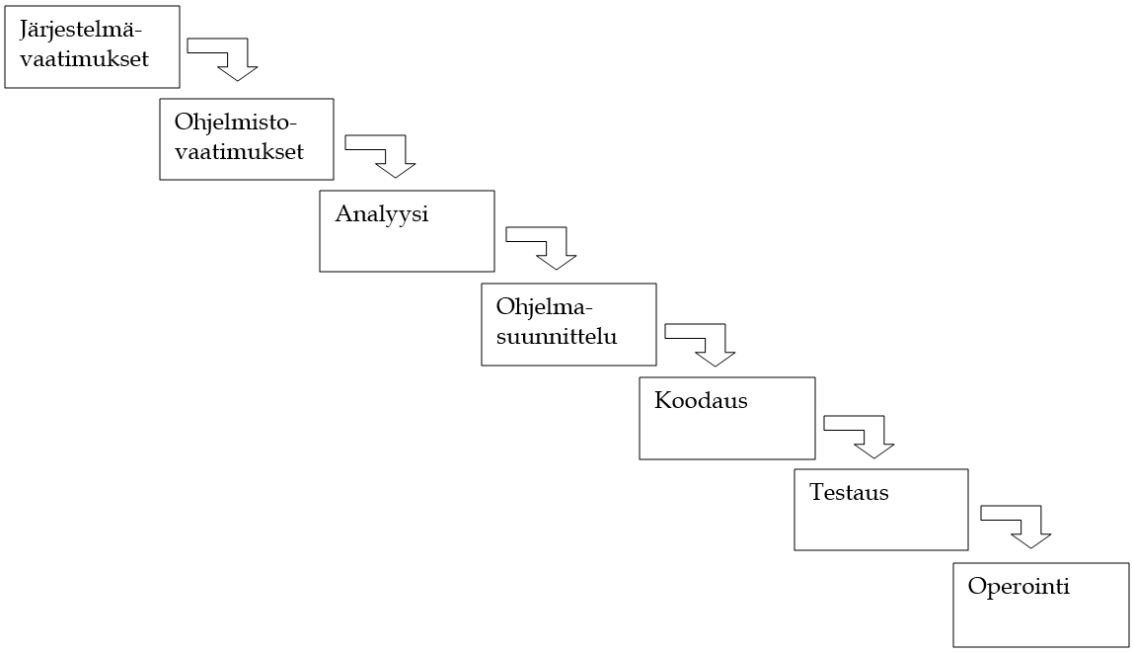
Ohjelmistokehitysmenetelmiä on useita erilaisia ja monet organisaatiot luovat ja käyttävät omia menetelmiään. Tunnetuimpia menetelmiä ovat muun muassa vesiputousmalli, V-malli, inkrementaalinen-malli, RAD-malli, ketterät menetelmät, iteratiivinen malli ja spiraalimalli. Jokaisella mallilla on omat hyötynsä ja haittansa, joten malli on valittava organisaation tarpeiden mukaan. (Dybå & Dingsøyr, 2008).

3.1 Perinteinen ohjelmistokehittäminen

Ohjelmistokehitysmenetelmiä on ollut käytössä lähes niin kauan kuin ohjelmistoja on kehitetty. Jo vuonna 1956 kehitettiin vaiheittaisia kehitysmalleja ohjelmistokehitykseen (Boehm, 1988). Perinteisellä ohjelmistokehityksellä tarkoitetaan yleensä ylhäältä alas suuntautuvaa vaiheistettua kehitystä. Menetelmissä lukitaan määrittelyt aikaisessa vaiheessa ja tuotetaan mittavasti dokumentaatiota. Perinteisissä menetelmissä on pidemmät kehityskaaret (Boehm & Turner, 2005). Perinteiset ohjelmistokehitysmenetelmät ovat vieläkin läsnä. Niitä jopa käytetään enemmän kriittisissä projekteissa, kuin ketteriä menetelmiä (Vijayasarathy, Butler, 2016)

3.1.1 Vesiputousmalli

Kenties tunnetuin perinteinen kehitysmalli on Roycen (1970) julkaisema vesiputousmalli. Kuviossa 1 näkyy, kuinka mallissa on seitsemän eri vaihetta, joita edetään vaiheittain siirtyen aina seuraavaan edellisen valmistuessa. Perinteisessä ohjelmistokehityksessä oletetaan, että jos yritämme tarpeeksi lujaa, pystymme ennakoimaan kaikki tulevat vaatimukset etukäteen ja täten vähentämään muutoksen aiheuttamia kuluja (Highsmith & Cockburn, 2001).



Kuvio 1 Vesiputousmalli Roycen (1970) mukaan

Vesiputousmallin perusvaiheet ovat: järjestelmävaatimukset, ohjelmistovaatimukset, analyysi, ohjelma suunnittelu, koodaus, testaus ja operointi. Perustaltaan malli on järkevä, mutta siihen tulee lisätä elementtejä riskien pienentämiseksi: peräkkäisten vaiheiden välillä tulee olla iteratiivista interaktiota, suunnittelua ei tule rajoittaa vain yhteen vaiheeseen, dokumentaation tulee olla kattavaa ja ajankohtaista. Lisäksi malliin voidaan lisätä kahdeksas vaihe: alustava ohjelmasuunnittelu, joka sijoittuu ohjelmistovaatimuksien ja analyysin väliin. Tämä vaihe voi itsessään muodostaa pienemmän vesiputouksen, jolla voidaan tukea kehitysprosessia. (Royce, 1970.)

Todellisuudessa vesiputousmalli ei ollut koskaan täysin sitova kehittämismalli, vaan enemmänkin hallintomalli ja suunnittelutehtäviä tehtiin myös projektin myöhemmissä vaiheissa (Armour, 2014).

3.1.2 Spiraalimalli

Kaikki projektit pitävät sisällään jonkin tasoisia riskejä ja niiden hallinta määrittelee loppujen lopuksi sen onnistuuko vai epäonnistuuko projekti. Tältä pohjalta Boehm esitteli vuonna 1988 Spiraalimallin, joka piti sisällään riskienhallinta elementtejä ja sekoitta iteratiivisuutta sekä lineaarisuutta ja vaiheistusta. (Oriogun, 1999.)

Malli oli kehitetty vesiputousmallin, sen uudistusten ja kokemusten pohjalta ohjelmistokehityksen hallintamalliksi. Malli kuvastaa perustana olevaa konseptia, jossa jokainen sykli sisältää edistystä, joka seuraa yhtenäisen järjestyksen askelia jokaiselle osalle tuotetta ja sen jokaiselle työstämisen jokaiselle tasolle kokonaiskonseptista jokaisen komponentin koodaamiseen.

Tavallisesti jokainen spiraalin sykli alkaa tuotteen kehitettävien osien, vaihtoehtoisten menetelmien ja niiden esteiden sekä rajoitteiden tunnistamisella. Seuraavassa vaiheessa tuote ja sen vaihtoehdot arvioidaan suhteessa tavoitteisiin ja rajoituksiin sekä niihin liittyvät riskit pyritään havainnoimaan. Tämän seurauksena mallissa voidaan jatkaa seuraavaan kehitysvaiheeseen prototyypin rakentamiseen. Yleensä prototyyppejä rakennetaan ja arvioidaan iteratiivisesti, kunnes se on hyväksyttävällä tasolla riskeihin nähden. Prototyyppi voidaan ottaa tuotantoon, mikäli se on tarpeeksi vakaa. Tämän jälkeen ryhmä tekee raportin sekä suunnitellee projektin jatkomahdollisuuksia. Lopuksi ydinryhmä arvioi suoriutumisen sekä jatkosuunnitelmat. Tämän vaiheen tarkoituksena on varmistaa, että kaikki osakkaat ovat sitoutuneita projektiin. Spiraali loppuu, kun uusi ohjelmisto on saatu käyttöön tai vanha muokattua tavoiteltuun muotoon. Spiraali voidaan myös lopettaa kesken, mikäli sen jatkamisella ei nähdä arvoa tai se on liian riskialtista. (Boehm, 1988.)

Myöhemmin Spiraalimallia päivitettiin Win-Win spiraalimalliksi lisäämällä jokaisen syklin alkuun W-teorian aktiviteettejä. Nämä aktiviteetit ovat järjestyksessä: seuraavan tason sidosryhmien tunnistaminen, sidosryhmien voittoedellytykset ja voittoehtojen sovittelu. Tällä tavoin malli pyrkii selkeyttämään mistä vaatimukset, rajoitteet ja vaihtoehdot syntyvät. (Boehm, Egyed, Kwan, Shah & Madachy, 1998.)

3.2 Ketterä ohjelmistokehittäminen

Ketterät ohjelmistokehittämisen menetelmät perustuvat periaatteisiin, jotka linjataan agile manifestossa. Itse manifesti on suomennettuna ”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.” (agilemanifesto.org.) Vaikka myöhemmin ketteryyden perusperiaatteita on lisätty ja muokattu alkuperäisistä, ovat alkuperäiset silti hyvin osuvia (Williams, 2012).

3.2.1 Scrum

Scrumin esitteli ensimmäisen kerran vuonna 1986 Takeuchi ja Nonaka artikkelissaan ”The new new product development game”. Artikkelissa esitellään kolme muutosta, joita organisaation hallinnassa täytyy tehdä, jotta voidaan saavuttaa nopeutta ja joustavuutta. Ensimmäiseksi organisaatioiden täytyy omaksua johtamistyyli, joka kannustaa prosessiin osallistumista. Toisena organisaatioiden tulee muuttaa oppimista. Kapean erikoistuvan syväoppimisen sijaan johdon tulisi

kerätä tietoutta laajemmin koko kentän laajuudelta. Kolmanneksi organisaatioiden tulisi asettaa erilaisia tavoitteita tuotekehitykselle pelkän tuoton sijaan. (Takeuchi & Nonaka, 1986.)

Scrum voidaan määritellä projektin hallintapainotteisena ketteränä kehitysmetodina, joka on saanut inspiraatiota laaja-alaisesti esimerkiksi monimutkaisuusteoriasta, järjestelmädynamiikasta sekä Nonakan ja Takeuchin tiedon luonnin teoriasta. Scrumissa on omaksuttu teorioita eri aloilta ja tuotu ne ohjelmistokehityksen kontekstiin. (Moe, Dingsøyr & Dybå, 2010.)

Rising ja Janoff kuvailevat Scrumia pienten tiimien kehitysprosessiksi, joka pitää sisällään lyhyitä kehitysjaksoja eli iteraatioita (”sprinttejä”). Scrum-tiimille annetaan merkittävästi vaikutusvaltaa ja vastuuta työnsä eri vaiheisiin, kuten suunnittelu, aikatauluttaminen, tehtäväjako jäsenten kesken ja päätöksenteko. (Rising & Janoff, 2000.)

Itsensä ohjaaminen on Scrumin määrittelevä piirre. Verrattuna perinteiseen komento-ja-valvonta painotteiseen johtamiseen. Scrum edustaa radikaalisti uudenlaista lähestymistä suunniteluun ja johtamiseen ohjelmistoprojekteissa, koska se tuo päätöksenteon vallan samalle tasolle operatiivisten ongelmien ja epävarmuuksien kanssa. (Moe, Dingsøyr & Dybå, 2010.)

3.2.2 Lean

Lean kehittäminen on tuotekehitysmalli, jossa on päästä päähän tavoitteena arvonluonti asiakkaalle, hukan eliminointi, arvonvirtojen optimointi, ihmisten valtuuttaminen ja jatkuva kehitys. Lean-ajattelu on saanut alkunsa teollisuudessa ja se on omaksuttu useilla toimialoilla. (Ebert, Abrahamsson & Oza, 2012.)

Lean kehitys sekoittuu usein ketterän kehittämisen sekaan tai sen synonyymiksi. Leanin voi erottaa muusta ketterästä kehittämisestä kolmella keskinäisellä osatekijällä: Lean konseptit, Lean periaatteet ja Lean käytänteet. (Wang, Conboy & Cawley, 2012).

Olennainen käsite ohjelmistokehitykselle on se, että työ tulisi jakaa osissa eteenpäin välittömästi sekä myös palauttaa takaisinpäin tuotantoketjussa välittömästi, mikäli se on virheellistä tai puutteellista. Tällä tavoin työntekijät saavat välitöntä ja säälimätöntä palautetta suoriutumisestaan. (Middleton, 2001.)

Ohjelmistoalan näkökulmasta polku kohti Leania alkoi ohjelmoinnin ketterillä menetelmillä. Viimeisen vuosikymmenen aikana suurin osa yrityksistä on omaksunut ketterän kehittämisen tavoitellakseen tehokkuutta ja vaikuttavuutta. Vaikka se onkin auttanut valtavasti käyttäjäkeskeisen ja iteratiivisen kehityksen kanssa, sen yritystason soveltuvuus on seisahtunut. Liian usein ketterät käytänteet rajoittuvat ryhmään tai projektiin. Keskittyessä liikaa lyhyeen aikaväliin, kuten dokumentaation ja tarpeettomien osien vähentämiseen huomataan kuitenkin lopuksi, että vaikutus elinkaarikustannuksiin on ollut negatiivinen. Lean kehitys pyrkii korjaamaan tämän ongelman. (Ebert, Abrahamsson & Oza, 2012.) Ebert ym. myös argumentoivat, että jotkin Leanin periaatteet eivät sovellu ohjelmistokehitykseen.

3.2.3 ExtremeProgramming

Vuosituhannen alussa ExtremeProgramming (XP) on ollut suosituin ketterä menetelmä. XP:ssä keskitytään käytettävän koodin ja automatisoitujen testien luontiin paperisten vaatimus- ja suunnitteludokumenttien sijasta. Tämä keskittyminen lähdekoodiin tekee XP:stä kiistanalaisen ja sitä verrataan jopa hakkerointiin. Tämä vertaus on epäoikeutettu, sillä XP:ssä arvostetaan yksinkertaista suunnittelua ja vastaa hakkerointiväitteisiin asettamalla painoarvoa refaktoroinnille, vahvalle regressiotestaukselle ja jatkuvalle koodin tarkastamiselle parikoodaamisen kautta. (Maurer & Martel, 2002.)

Vaikka kehittäjät voivat käyttää erilaisia XP käytäntöjä, menetelmä yleensä pitää sisällään 12 peruskäytäntöä:

1. Pelin suunnittelu: Seuraavan julkaisun nopea suunnittelu yhdistämällä taloudelliset päämäärät ja tekniset arviot. Asiakas päättää laajuuden, tärkeysjärjestyksen ja päivämäärät taloudellisesta näkökulmasta, kun taas tekniset työntekijät arvioivat ja seuraavat etenemistä.
2. Pienet julkaisut: Laitetaan yksinkertainen järjestelmä tuotantoon nopeasti. Julkaistaan uusia versioita nopealla tahdilla (kahden viikon sykli).
3. Metafora: Johdetaan kaikkea kehittämistä yksinkertaisella jaetulla tarinalla siitä, kuinka koko järjestelmän tulisi toimia.
4. Yksinkertainen suunnittelu: Suunnitellaan niin yksinkertaisesti kuin vain mahdollista sillä hetkellä.
5. Testaus: Kehittäjät jatkuvasti kirjoittavat yksikkötestejä, joiden tulee suoriutua virheettömästi. Asiakkaat kirjoittavat testejä, jotka havainnollistavat toimintojen olevan valmiita. ”Testataan ja sitten koodataan” tarkoittaa, että epäonnistunut testitapaus on aloituskriteeri koodaamiselle.
6. Refaktorointi: Uudelleen rakennetaan järjestelmä muuttamatta sen käyttäytymistä toistojen poistamiseksi, kommunikoinnin parantamiseksi, yksinkertaistamiseksi tai joustavuuden lisäämiseksi.
7. Parikoodaaminen: Kaiken tuotantokoodin kirjoittaa kaksi ohjelmoijaa yhdellä koneella.
8. Kollektiivinen omistajuus: Kaikki voivat parantaa järjestelmän koodia missä vain, milloin vain.
9. Jatkuva integraatio: Integroidaan ja rakennetaan järjestelmä monta kertaa päivässä (aina kun tehtävä saadaan valmiiksi). Jatkuva regressiotestaus estää toiminnallisuuksien regression vaatimusten muuttuessa.
10. 40-tuntiset viikot: Työtä tulisi tehdä enintään 40 tuntia viikossa, mikäli mahdollista. Ylitöitä ei tulisi koskaan tehdä kahtena peräkkäisenä viikkona.

11. Paikalla oleva asiakas: Ryhmässä tulisi olla todellinen loppukäyttäjä täysipäiväisesti vastaamassa kysymyksiin.
12. Ohjelmointistandardit: Säännöt, jotka painottavat kommunikointia läpi koodin. (Paulk, 2001.)

XP:ssä kaikki kehittäjät työskentelevät läheisesti, jotta he voivat kommunikoida informaalisti ja täten välttää ajan kulutusta suunnittelun ja päätösten dokumentointiin. Kun organisaatio kasvaa, kuluu enemmän aikaa tuotetietouden levittämiseen ja uuden henkilöstön kouluttamiseen. Tämä usein tekee XP:stä kelvottoman suurempiin ryhmiin. XP on ketterä ohjelmistokehitysprosessi, joka nopeuttaa kehitystä ja sallii ryhmien reagoida joustavammin vaatimusten muutoksiin, mutta sillä on myös ongelmansa. Yksi ongelmista on epävarma skaalautuvuus. XPtä suositellaan 5–15 ihmisen ryhmille, mutta varmaa tietoa ryhmäkoon ylikasvamisesta ei ole. (Maurer & Martel, 2002.)

XP käytänteet eivät pidä sisällään laajamittaista vaatimus- ja suunnitteluvalmistelua ennen kehittämisen aloitusta. Sen seurauksena XP on hyvin riippuvainen jatkuvasta kommunikaatioista sidosryhmien kesken sekä tiivistä palautekierrosta toimintojen toteutuksen selventämiseksi ja tarkentamiseksi ja muutoksiin vastaamiksesi. Tämä jatkuva kommunikaatio voi olla haaste globaaleille ryhmille. (Layman, Williams, Damian & Bures, 2006.)

4 TULOKSET

Tässä luvussa tarkastellaan ohjelmistokehitystyypeille ominaisia teknisen velan tyyppejä ja niiden ilmentymistä. Teknisenvelan tyyppeinä käytetään Li ym., (2015) löytämiä kymmentä teknisen velan tyyppiä: vaatimus-, arkkitehtuuri-, suunnittelu-, koodi-, testaus-, rakenne-, dokumentaatio-, infrastruktuuri-, versiohallinta- ja virhevelka (Alves ym., 2014; Benidris, 2018; Li ym., 2015.)

Näitä teknisen velan tyyppejä verrataan kehitysmenetelmien sekä velan tahallisuuden muodostamaan taulukkoon 1.

Taulukko 1 Kehitysmenetelmät sekä teknisen velan tyypit

	Tahallista teknistä velkaa (Mensah, Keung, Svajlenko, Bennin & Mi. 2018; Buschmann, 2011)	**Tahatonta teknistä velkaa . (Buschmann, 2011)**	**Sekä tahallista että tahatonta teknistä velkaa . (Buschmann, 2011)**
Vesiputousmalli (Royce, 1970; Highsmith & Cockburn, 2001; Armour, 2014)		Vaatimus, arkkitehtuuri, suunnittelu, koodi, testaus, rakenne, dokumentaatio, infrastruktuuri, versiohallinta ja virhe (Alves ym., 2014; Benidris, 2018; Li ym., 2015.)	
Spiraali-malli (Boehm, 1988; Boehm, Egyed, Kwan, Shah & Madachy, 1998; Oriogun, 1999)		Vaatimus, arkkitehtuuri, suunnittelu, koodi, testaus, rakenne, dokumentaatio, infrastruktuuri, versiohallinta ja virhe (Alves ym., 2014; Benidris, 2018; Li ym., 2015.)	
Scrum(Takeuchi & Nonaka, 1986;	Vaatimus, arkkitehtuuri,	Virhe (Alves ym., 2014;	Dokumentaatio (Alves ym., 2014;

Moe, Dingsøyr & Dybå, 2010; Rising & Janoff, 2000)	suunnittelu, koodi, testaus, rakenne, dokumentaatio, infrastruktuuri, versiohallinta ja virhe velaksi	Benidris, 2018; Li ym., 2015.)	Benidris, 2018; Li ym., 2015.)
Lean(Ebert, Abrahamsson & Oza, 2012; Wang, Conboy & Cawley, 2012; Middleton, 2001)	Testaus, rakenne, infrastruktuuri ja versiohallinta (Alves ym., 2014; Benidris, 2018; Li ym., 2015.)	Virhe (Alves ym., 2014; Benidris, 2018; Li ym., 2015.)	Vaatimus, arkkitehtuuri, suunnittelu, koodi dokumentaatio (Alves ym., 2014; Benidris, 2018; Li ym., 2015.)
Extreme Programming (Maurer & Martel, 2002; Paulk, 2001; Layman, Williams, Damian & Bures, 2006)	Vaatimus, arkkitehtuuri, suunnittelu, rakenne, infrastruktuuri ja versiohallinta (Alves ym., 2014; Benidris, 2018; Li ym., 2015.)	Virhe ja Dokumentaatio (Alves ym., 2014; Benidris, 2018; Li ym., 2015.)	

4.1 Tekninen velka perinteisessä ohjelmistokehityksessä

Projektien alkuvaiheessa projektitiimit eivät lähes koskaan hallitse ongelmaa täysin. Tämä on syvin ongelma vesiputousmallisessa ohjelmistokehittämisessä, jossa kaikki vaatimukset pyritään lyömään lukkoon ennen suunnittelun alkamista, joka taas voi olla valmis ennen kuin järjestelmä on käyttöönotettu ja niin edelleen. Tämä vaikuttaa argumenttina hyvältä, sillä muutosten hinta kasvaa eksponentiaalisesti kehityksen edetessä, joten paras tapa olisi tehdä alkuvaiheet kunnolla ja jatkaa eteenpäin. Todellisuudessa vaatimukset kuitenkin aina muuttuvat ja on yleensä parasta tehdä toimiva prototyyppi, jota asiakkaat voivat käyttää ja sen kautta tutustua järjestelmään. (Allman, 2012.)

4.2 Tekninen velka ketterässä ohjelmistokehityksessä

Ketterät kehittäjät usein kohtaavat tiukkojen aikataulujen ja asiakkaalle arvontuottamisen aiheuttamia paineita. Heidät usein pakotetaan ottamaan oikoteitä

nopeaa tuotantoa tavoitellessa, joka johtaa tekniseen velkaan. (Behutiye, Rodríguez, Oivo, & Tosun, 2017.)
Tekninen velka on keskeistä ketterälle kehitykselle. Mitä nopeammin teknistä velkaa hankitaan, sitä nopeammin voidaan tuote julkaista. Samalla kuitenkin edellisten iteraatioiden tekninen velka hidastaa tulevaa kehitystä sekä estää lisäominaisuuksien kehittämisen. Tämän takia ketterien kehitysryhmien täytyy tasapainotella haluttujen ominaisuuksien ja teknisen velan ja sen takaisin maksun välillä. (Bavani, 2012.)

Vaikka ketterät kehitysmenetelmät kannattavat yhteistyötä asiakkaan kanssa kehitysvaiheessa, voi teknisen velan kommunikointi olla haasteellista. Kehittäjät kokevat haasteelliseksi kommunikoida teknisestä velasta ja sen tarpeesta asiakassidosryhmille. (Behutiye, Rodríguez, Oivo, & Tosun, 2017.)

Ketterien kehittäjien tulisi tämän takia on erityisen tarkkoja siinä, mikä muodostaa teknistä velkaa, sen taloudellisista vaikutuksista ja strategioista sen takaisinmaksuun ketterässä kehityksessä, sekä myös velan muihin välillisiin vaikutuksiin ketteriin menetelmiin. Aina kun tekninen velka muodostuu epästrategisesti, se kasvaa tasolle, joka pakottaa ketterät kehitysryhmät käyttämään paljon resursseja virheiden korjaamiseen ja järjestelmän tasapainon ylläpitoon. Tämä taas hidastaa muuta työskentelyä ja laskee tuottavuutta. (Behutiye, Rodríguez, Oivo, & Tosun, 2017.)

Ketterässä kehityksessä velkaa kertyy huomattavasti vaihdoksissa perinteisistä menetelmistä ketteriin (Bavani, 2012). Sekä silloin kun valittuja menetelmiä ei noudateta (Behutiye, Rodríguez, Oivo, & Tosun, 2017).

Teknistä velkaa hallitakseen ketterien kehitysryhmien tulee olla tietoisia ja järjestäytyneitä. Velan maksua varten tulisi tuottaa käyttäjätarinoita, joiden avulla velkaa voidaan maksaa tulevissa iteraatioissa, jotta velka vähentyisi hiljalleen pidemmällä aikavälillä. Kehittäjien tulisi myös olla tietoisia teknisen velan hankinta- ja maksukeinoista tuotteiden kehitys- tai ylläpitotehtävissä. Lisäksi testaajien tulisi myös huomioida automaattisten testien suunnittelun, rakentamisen ja ylläpidon aiheuttama tekninen velka. (Bavani, 2012.)

5 YHTEENVETO JA JATKOTUTKIMUSAIHEET

Tämän kirjallisuuskatsauksen tavoitteena oli vastata tutkimuskysymykseen:

> *Kuinka perinteisten ja ketterien ohjelmistokehitysmenetelmien tekninen velka eroavat toisistaan?*

Luvussa kaksi esiteltiin ja tarkasteltiin teknistä velkaa, sen tyyppejä ja sen vaikutusta ohjelmistokehitykseen. Seuraavassa luvussa vastaavasti esiteltiin perinteisiä sekä ketteriä ohjelmistokehitysmenetelmiä. Perinteisiksi kehitysmenetelmiksi valikoitui vesiputousmalli sekä spiraalimalli. Ketteriksi kehitysmenetelmiksi puolestaan valikoituivat Scrum, Lean ja ExtremeProgramming. Nämä menetelmät valikoituivat tähän kirjallisuuskatsaukseen niiden suosion, merkittävyyden sekä lähdemäärän vuoksi. Tuloksena havaittiin, että kaikessa kehitystyössä syntyy teknistä velkaa. Perinteisessä kehityksessä tekninen velka pyritään välttämään kokonaisuudessaan tekemällä mahdollisimman perusteellista työtä joka vaiheessa. Tämä kuitenkaan harvoin onnistuu. Ketterässä kehityksessä teknisellä velalla pyritään kiihdyttämään kehitystahtia kuitenkaan siihen hukkumatta. Ketterässä kehityksessä velan kanssa tasapainottelu erottaa usein onnistuneet ja epäonnistuneet projektit. Tärkeää on huomioida, että tekninen velka toimii sekä positiivisena että negatiivisena tekijänä ohjelmistokehityksessä ja siltä täysin välttyminen on mahdotonta.

Mielenkiintoinen jatkotutkimusaihe on eri ohjelmistokehitysmenetelmissä syntyvän teknisen velan jakaminen eri tyyppeihin. Toisena mielenkiintoisena jatkotutkimusaiheena voisi olla kehitysmenetelmien sisäisten velkaprosessien tunnistaminen.

LÄHTEET

Allman, E. (2012). Managing technical debt. Communications of the ACM, 55(5), pp. 50-55.

Alves, N. S., Mendes, T. S., de Mendonça, M. G., Spínola, R. O., Shull, F., & Seaman, C. (2016). Identification and management of technical debt: A systematic mapping study. *Information and Software Technology*, *70*, 100-121.

Armour, P. G. (2014). Estimation is not evil.(project cost estimation and agile software development)(Viewpoints / The Business of Software). *Communications of the ACM, 57*(1), p. 42.

Bavani, R. (2012). Distributed Agile, Agile Testing, and Technical Debt. *IEEE Software, 29*(6), pp. 28-33. doi:10.1109/MS.2012.155

Behutiye, W. N., Rodríguez, P., Oivo, M. & Tosun, A. (2017). Analyzing the concept of technical debt in the context of agile software development: A systematic literature review. *Information and Software Technology, 82*, pp. 139-158. doi:10.1016/j.infsof.2016.10.004

Benidris, M. (2018). Investigate, Identify and Estimate the Technical Debt: A Systematic Mapping Study. *International Journal of Software Engineering & Applications, 9*(5), pp. 01-14. doi:10.5121/ijsea.2018.9501.

Boehm, B. W. (1988). A spiral model of software development and enhancement. Computer, 21(5), pp. 61-72. doi:10.1109/2.59

Boehm, B., Egyed, A., Kwan, J., Port, D., Shah, A. & Madachy, R. (1998). Using the WinWin spiral model: A case study. *COMPUTER, 31*(7), pp. 33-44. doi:10.1109/2.689675

Boehm, B. & Turner, R. (2005). Management challenges to implementing agile processes in traditional development organizations. *IEEE Software, 22*(5), pp. 30-39. doi:10.1109/MS.2005.129

Buschmann, F. (2011). To Pay or Not to Pay Technical Debt. *IEEE Software, 28*(6), pp. 29-31

Cunningham, W. (1992). The WyCash portfolio management system. *OOPS Messenger, 4*, 29-30.

Dybå, T., & Dingsøyr, T. (2008). Empirical studies of agile software development: A systematic review. Information and software technology, 50(9-10), 833-859.

Ebert, C., Abrahamsson, P., & Oza, N. (2012). Lean software development. IEEE Software, (5), 22-25.

Highsmith, J., & Cockburn, A. (2001). Agile software development: The business of innovation. *Computer, 34*(9), 120-127. doi:http://dx.doi.org.ezproxy.jyu.fi/10.1109/2.947100

Kruchten, P. (2012). Technical Debt: From Metaphor to Theory and Practice. IEEE Software, 29(6), pp. 18-21.

Layman, L., Williams, L., Damian, D., & Bures, H. (2006). Essential communication practices for Extreme Programming in a global software development team. Information and software technology, 48(9), 781-794.

Li, Z., Avgeriou, P., & Liang, P. (2015). A systematic mapping study on technical debt and its management. Journal of Systems and Software, 101, 193-220.

Martini, A. (2018). Technical Debt tracking: Current state of practice: A survey and multiple case study in 15 large organizations: A survey and multiple case study in 15 large organizations. *Science of Computer Programming, 163*, pp. 42-61. doi:10.1016/j.scico.2018.03.007

Maurer, F., & Martel, S. (2002). Extreme programming. Rapid development for Web-based applications. IEEE Internet computing, 6(1), 86-90.

Mensah, S., Keung, J., Svajlenko, J., Bennin, K. E. & Mi, Q. (2018). On the value of a prioritization scheme for resolving Self-admitted technical debt. *The Journal of Systems & Software, 135*, pp. 37-54. doi:10.1016/j.jss.2017.09.026

Middleton, P. (2001). Lean software development: two case studies. Software Quality Journal, 9(4), 241-252.

Moe, N. B., Dingsøyr, T., & Dybå, T. (2010). A teamwork model for understanding an agile team: A case study of a Scrum project. Information and Software Technology, 52(5), 480-491.

Rising, L., & Janoff, N. S. (2000). The Scrum software development process for small teams. IEEE software, 17(4), 26-32.

Royce, W. (1970). Managing The development of large software systems Proceedings. IEEE WESCON, s.1-9.

Takeuchi, H., & Nonaka, I. (1986). The new new product development game. Harvard business review, 64(1), 137-146.

Tom, E. (2013). An exploration of technical debt. The Journal of Systems and Software, 86(6), p. 1498.

Vijayasarathy, L. R. & Butler, C. W. (2016). Choice of Software Development Methodologies: Do Organizational, Project, and Team Characteristics Matter? IEEE Software vol. 33, no. 5, pp. 86-94.

Wang, X., Conboy, K., & Cawley, O. (2012). "Leagile" software development: An experience report analysis of the application of lean approaches in agile software development. Journal of Systems and Software, 85(6), 1287-1299.

Williams, L. (2012). What agile teams think of agile principles.(Contributed Articles)(agile software development)(Report). *Communications of the ACM, 55*(4), p. 71. doi:10.1145/2133806.2133823

Wolff, E. & Johann, S. (2015). Technical Debt. IEEE Software, 32(4), pp. 94-c3

Yli-Huumo, J. (2016). How do software development teams manage technical debt? – An empirical study. *The Journal of Systems & Software, 120*, pp. 195-218. doi:10.1016/j.jss.2016.05.018