Tampereen yliopisto

Juha Vepsä

TEKNINEN VELKA MODERNISSA OHJELMISTOTYÖSSÄ

Informaatioteknologian ja viestinnän tiedekunta
Diplomityö
Maaliskuu 2019

TIIVISTELMÄ

JUHA VEPSÄ: Tekninen velka modernissa ohjelmistotyössä
Tampereen yliopisto
Diplomityö, 48 sivua, 3 liitesivua
Maaliskuu 2019
Tietotekniikan diplomi-insinöörin tutkinto-ohjelma
Pääaine: Ohjelmistotuotanto
Tarkastajat: yliopistonlehtori Terhi Kilamo, professori Kari Systä
Avainsanat: tekninen velka, ketterät menetelmät, SonarQube

Ohjelmistokehityksen nopeus ja luonne ovat muuttuneet paljon sitten vesiputousmallin. Moderni ohjelmistotyö on nopeatempoista ja se pyrkii vastaamaan jatkuviin toimintaympäristön muutoksiin. Modernissa ohjelmistotyössä painotetaan muun muassa kehitystyön edistämistä kaiken kattavien suunnitelmien laatimisen sijaan. Modernin ohjelmistotyön toimintakulttuurissa pyritään ylipäänsä ketteryyteen ja joustavuuteen.

Kasvanut ohjelmistotuotantonopeus ja ohjelmistokehityskulttuurin muutos voivat kuitenkin altistaa kehitettävän ohjelmiston teknisen velan synnylle. Teknisellä velalla tarkoitetaan ohjelmiston lähdekoodissa esiintyviä, ohjelmointityön seurauksena syntyviä virheitä, jotka vaikeuttavat ohjelmiston jatkokehitystä ja tekevät lähdekoodin tulkinnasta vaikeampaa. Runsaasti teknistä velkaa sisältävää koodia on hidas jatkokehittää. Teknistä velkaa voi syntyä esimerkiksi aikataulupaineiden takia ja tekninen velka tulisikin ottaa huomioon modernissa ohjelmistotyössä.

Tässä diplomityössä on tutkittu miten tekninen velka esiintyy ja miten se kehittyy modernissa ohjelmistotyössä. Työn tutkimusosassa on tutkittu kahta nykyaikaisin ohjelmistokehitysmenetelmin toteutettua ohjelmistoa: Vaadin Framework 7 ja Yle Areena. Tutkimuksessa analysoitiin molempien ohjelmistojen koodipohjien kehitystä SonarQubella. SonarQube on staattinen koodin analysointityökalu, joka tuottaa sille syötetystä koodista tietoa koskien muun muassa teknistä velkaa, ohjelmointivirheitä ja haavoittuvuuksia. Työn tuloksissa käsitellään muun muassa teknisen velan kehittymistä ohjelmistoissa pitkällä aikavälillä.

Tuloksissa havaittiin, että tekninen velka kasvaa samassa suhteessa lähdekoodin määrän kanssa, mikäli siihen ei kiinnitetä huomioita. Kuitenkin jo vähäiset toimenpiteet lähdekoodin laadun ja siisteyden ylläpitämiseksi auttavat myös teknisen velan hallinnassa, vaikka toimenpiteiden tarkoitus ei alunperin olisikaan torjua nimenomaan teknistä velkaa. Yleisimmät teknisen velan lähteet liittyvät muun muassa perinnekoodiin, ohjelmointityyliin ja literaalien käyttöön vakioiden sijaan. Tuloksissa esiintyy jonkin verran ”kouluesimerkkivirheitä” esimerkiksi liukulukujen välillä tapahtuvasta vertailusta yhtäsuuruusoperaattorilla tai geneeristen poikkeuksien käyttämistä erikoistettujen sijaan.

ABSTRACT

JUHA VEPSÄ: Technical debt in modern software development
Tampere University
Master of Science Thesis, 48 pages, 3 Appendix pages
March 2019
Master's Degree Programme in Information Technology
Major: Software engineering
Examiners: University lecturer Terhi Kilamo, Professor Kari Systä
Keywords: technical debt, agile methodologies, SonarQube

The speed and nature of software development has changed a lot since the waterfall model. Modern software work is fast-paced and strives to respond to constant changes in the operating environment. Modern software work emphasizes the development work instead of composing comprehensive plans. The culture of action is generally about agility and flexibility.

However, increased software production speed and a change in software development culture can expose the software being developed to the emergence of technical debt. Technical debt refers to errors and structural shortcomings in the source code resulting from poor or rushed programming work. Technical debt makes software development and source code interpretation more difficult. A source code containing considerably amounts of technical debt is slow to develop. Technical debt may arise, for example, due to scheduling pressures and technical debt should be taken into account in modern software work.

In this master's thesis it has been studied how technical debt occurs and how it develops in modern software development work. In the research part of the thesis, two software programs with the latest software development methods have been studied: Vaadin Framework 7 and Yle Areena. The codebases of both software projects were analyzed using SonarQube. SonarQube is a static code analysis tool that generates information about the quality of the source code you enter. SonarQube provides information for example about technical debt, programming errors and vulnerabilities in the source code. The results of the thesis include the development of technical debt in software over the long term.

The results showed that the technical debt increases in proportion to the amount of source code, if not addressed. However, even small measures to maintain the quality and cleanliness of the source code will also help to manage the technical debt, although the purpose of the measures would not have been to combat the technical debt. The most common sources of technical debt are related to the legacy code, the programming style and the use of literals instead of constants in the code. The results show some traditional errors for example comparisons done between the floating point numbers with the equality operator, or the use of generic exceptions instead of specialized ones.

ALKUSANAT

Tämä diplomityö on ollut työn alla pitkään. Kolmisen vuotta on kulunut siitä, kun aloittelin työn tekemistä ensimmäisen kerran. Näihin kolmeen vuoteen sisältyykin pitkiä vaikeita jaksoja, jolloin työ ei edistynyt. Monen tekijän summa aiheutti motivaation painumisen pakkasen puolelle, ja työn hidas eteneminen vain pahensi tilannetta. Noidankehä oli valmis.

Kuitenkaan missään vaiheessa ei mielessäni käynytkään luovuttaminen. Syksyllä 2018 aloin edistää työtä pienin askelin pitkän tauon jälkeen. Työ alkoi edetä ja valmistui hyvää vauhtia talven aikana. Nyt, kun työ on valmis, putoaa harteilta samalla melko iso taakka. Jos kuvailisin työn etenemistä hiihtotermein, voisin sanoa, että alussa ja lopussa kulki hyvin, keskivaiheen pitkä työpätkä oli hapottava.

Tässä yhteydessä haluaisin myös kiittää eri tahoja, jotka ovat edesauttaneet tämän työn edistymistä sen eri vaiheissa. Kiitokset Terhi Kilamolle työni ohjaamisesta, kaikesta avusta ja neuvoista sekä kärsivällisyydestä, jota tämän pitkittyneen projektin läpi vieminen on häneltä vaatinut. Kiitos myös Kari Syställe neuvoista ja vinkeistä, jotka auttoivat työni edistymistä sen alkuvaiheessa. Lisäksi haluaisin kiittää Vaadin Oy:n Jurka Rahikkalaa ja Yleisradion Olli Vistbackaa, jotka mahdollistivat työni tutkimusosuuden toteuttamisen, sekä siskoani Marja Vepsä-Taipaletta, joka auttoi minua tekstin oikolukemisessa ja viimeistelyssä. Lopuksi vielä kiitos kaikille opiskelijayhteisöille, joiden toiminnassa olen saanut olla mukana, sekä etenkin kaikille hyville ystävilleni ja opiskelukavereilleni, joiden kanssa on ollut ilo jakaa monet hyvät hetket pitkällä opiskelutaipaleellani.

Lisäksi vanhempieni tuki on ollut minulle korvaamattoman arvokasta läpi koko opiskeluaikani, ja ilman sitä olisi valmistuminen varmasti jäänyt haaveeksi.

Tampereella, 29.3.2019

Juha Vepsä

SISÄLLYS

LYHENTEET JA MERKINNÄT

AJAX	engl. asynchronous JavaScript and XML, teknologia asynkroniseen tiedonvaihtoon asiakasohjelman ja palvelimen välillä
GWT	engl. Google Web Toolkit, työkalupaketti web-kehitykseen

1. JOHDANTO

Moderni ohjelmistotyö on kehittynyt 1990-luvulla ja 2000-luvun alussa esiteltyjen *ketterien menetelmien* hengessä nopeatempoiseksi ja joustavaksi. Nykyaikaiset ohjelmistoyritykset pystyvät toimittamaan asiakkailleen uusia ohjelmistopäivityksiä tiheään tahtiin, jopa useita kertoja päivässä. Ohjelmistoalalla viime vuosien ja vuosikymmenien aikana kehittyneet menetelmät, työkalut ja infrastruktuuri mahdollistavat nopean ohjelmistokehityksen muuttuvassa toimintaympäristössä.

Nykyaikaiselta ohjelmistoyritykseltä vaaditaankin nopeutta ja joustavuutta. Tuotantonopeus ja toki myös hinta ovat merkittävissä rooleissa ohjelmistoyritysten välisessä asiakaskilpailussa. Lienee houkuttelevaa tilata ohjelmisto sen edullisimmin ja nopeimmin toteuttavalta taholta. Nopeatempoisessa ohjelmistotyössä on kuitenkin myös varjopuolensa: se saattaa altistaa kehitettävän ohjelmiston herkemmin *teknisen velan* synnylle, mikäli teknistä velkaa ei oteta huomioon kehitystyön aikana. Tekninen velka tarkoittaa ohjelmiston lähdekoodissa sijaitsevia, lähinnä ohjelmiston kehittäjälle näkyviä puutteita, virheitä tai rakenteellisia ongelmia, jotka haittaavat ohjelmiston kehitystyötä. Näin ollen ohjelmistoala on ollut viime vuosina yhä kiinnostuneempi teknistä velasta ja sen vaikutuksista ohjelmistotyötä kohtaan.

Nopeatempoinen ohjelmistokehitys ei toki automaattisesti tarkoita, että teknistä velkaa syntyisi koodiin tavallista enempää ja nopeasti tuotettu koodi voi varsin hyvin olla erittäin laadukasta. Suunnittelematta ja hätiköidysti tuotetulla koodilla on sen sijaan suurempi todennäköisyys sisältää teknistä velkaa. Lähdekoodin laatuun ja sen rakenteeseen tulisi kuitenkin nykyään kiinnittää erityistä huomioita, sillä modernin ohjelmistokehityksen iteratiivinen luonne korostaa teknisen velan vaikutuksia. Samaa koodisegmenttiä saatetaan palata muokkaamaan useita kertoja eri iteraatiokierroksilla, jolloin kyseisen segmentin heikko laatu saattaisi hidastaa kehitystyötä.

Tekninen velka voi kuulostaa melko teoreettiselta käsitteeltä, josta voi olla vaikea saada tarttumapintaa. Yhtä lailla teknisen velan havaitseminen ja lähdekoodissa näkyväksi tekeminen voi olla haastavaa. Tämä diplomityö pyrkii lähestymään muun muassa edellä mainittuja ongelmia modernin ohjelmistotyön kontekstissa.

Tässä diplomityössä tutkitaan kahden esimerkkitapauksen kautta, miten tekninen velka esiintyy ja miten se kehittyy modernissa ohjelmistotyössä. Työssä analysoidaan

Vaadin Framework 7:n ja Yle Areenan lähdekoodit ja selvitetään, miten tekninen velka konkreettisesti esiintyy ja miten se kehittyy projektien edetessä. Lähdekoodien analysoimisessa hyödynnetään staattista koodin analysointityökalua *SonarQubea* ja ohjelmistoprojekteja seurataan pitkällä aikavälillä kehitystyön edetessä.

Tämä työ koostuu johdanto-luvusta, jota seuraa teoria-luku. Siinä käsitellään teknistä velkaa sekä moderneja ohjelmistokehitysmalleja. Kolmannessa luvussa käydään läpi työssä käytetyt tutkimusaineistot ja -menetelmät. Näiden jälkeen luvussa 4 esitellään tutkimuksen tulokset ja luvussa 5 pohditaan saatuja tuloksia. Lopuksi luvussa 6 esitellään työn johtopäätökset.

2. TEKNINEN VELKA JA MODERNI OHJELMISTOTYÖ

Tekninen velka ohjelmiston lähdekoodissa mielletään usein negatiivisena asiana [48] [8]. Tekninen velka voi kuitenkin olla myös yksi ohjelmistoprojektin työkaluista, jonka avulla voidaan esimerkiksi hetkellisesti nopeuttaa ohjelmiston kehitystyötä. Teknisen velan haittavaikutukset korostuvat modernissa, tyypillisesti iteratiivisessa ohjelmistokehitystyössä. Kun samaan lähdekoodin osaan voidaan palata tekemään muutoksia useamman kerran kehitystyön aikana, on lähdekoodin oltava selkeää ja ymmärrettävää. Monimutkainen ja epäselvä lähdekoodi voi hidastaa ohjelmoijan työtä.

Modernissa ohjelmistotyössä suositaan tyypillisesti konkreettista kehitystyötä kattavien suunnitelmien ja lukuisten määrittelydokumenttien laatimisen sijaan. Työ on muuttunut tarkasti määritellyn prosessin ja sen noudattamisen sijaan väljemmissä raameissa toimivaksi työvirraksi. Moderni ohjelmistotyö korostaa ohjelmistokehitystiimien ja yksilöiden vapauksia kehitystyössä ja uusien työmenetelmien ja työkalujen myötä ohjelmistokehityksen tempo on kasvanut.

2.1 Tekninen velka

Ward Cunningham esittelee raportissaan vuodelta 1992 ensimmäistä kertaa termin tekninen velka [10]. Hän kuvailee termiä seuraavasti: *"viimeistelemättömän, nopeasti tuotetun ohjelmakoodin toimittaminen tuotantoon on kuin velan ottamista."* Tällä hän tarkoittaa esimerkiksi ohjelmakoodissa ensimmäisen toimivan toteutuksen hyväksymistä tuotantokoodiin ilman toteutuksen tarkempaa laatuarviointia ja viimeistelyä. Teknisen velan ottaminen voi näin ollen vauhdittaa kehitystyötä hetkellisesti ja sen avulla voidaan saavuttaa ohjelmistolle asetettuja aikataulutavoitteita nopeammin. Tekninen velka näkyy siis lähinnä ohjelmistokehittäjille ohjelmakoodin puutteina, virheinä ja sen rakenteellisina ongelmina. Tekninen velka voidaan jossain tapauksissa käsittää laajempana terminä, joka koskee myös esimerkiksi ohjelman dokumentointivelkaa tai suunnitteluvelkaa [6]. Tässä työssä teknistä velkaa käsitellään sen alkuperäisessä merkityksessä; lähdekoodin virheinä, puutteina ja rakenteellisina ongelmina.

Teknistä velkaa voi tietyiltä osin verrata taloudelliseen velkaan [2]. Lainaamalla rahaa pankista on mahdollista ostaa esimerkiksi asunto nopeammin kuin siihen itse säästämällä pystyisi. Puolestaan teknistä velkaa ottamalla voi olla mahdollista saada ohjelmisto valmiiksi nopeammin verrattuna tilanteeseen, jossa velkaa ei otettaisi. Rahalaina on kuitenkin myös maksettava pankille takaisin. Vastaavasti myös teknisen velan pois maksamiseen tulisi lähtökohtaisesti kiinnittää huomioita, joskin kaikissa tapauksissa se ei ole välttämätöntä. Teknisen velan pois maksaminen tapahtuu käytännössä *refaktoroimalla* niitä lähdekoodin osia, joissa teknistä velkaa tiedetään olevan.

Kuten rahalainalla, myös teknisellä velalla on korko [10][2]. Nopeasti tuotettu, viimeistelemätön ohjelmakoodi voi olla esimerkiksi vaikeasti ymmärrettävää, ohjelmointityyliltään poikkeavaa tai sen rakenne voi poiketa ohjelman muusta arkkitehtuurista. Cunningham kuvaa teknisen velan koroksi sitä ylimääräistä aikaa, joka kuluu, kun ohjelmoijat joutuvat työskentelemään tällaisen ”likaisen” lähdekoodin parissa. Jos esimerkiksi ohjelmoijalta kuluu kaksi tuntia uuden ominaisuuden toteuttamisessa teoreettisesti velattomaan ohjelmaan ja saman ominaisuuden toteuttaminen samaan, velkaa sisältävään ohjelmaan vie kolme tuntia, on tuo ylimääräinen kulunut tunti tässä tapauksessa teknisen velan aiheuttamaa korkoa. Tekninen velka voi siis hidastaa ohjelman kehitystyötä. Hyvin runsaasti teknistä velkaa sisältävä lähdekoodi voi pahimmillaan pysäyttää ohjelmiston kehitystyön kokonaan.

Termi tekninen velka toimii myös kommunikointityökaluna ohjelmistokehittäjien ja ei-teknisen henkilöstön välillä [5][6]. Muiden kuin ohjelmistoprojektin toteuttajien on tyypillisesti vaikea hahmottaa projektin sisältämää teknistä velkaa. Näin projektiin liittyvät muut sidosryhmät eivät välttämättä ole tietoisia teknisen velan kertymisestä ja siitä mahdollisesti aiheutuvista riskeistä. Velan mittaaminen ja sen muuttaminen paremmin ymmärrettävään muotoon voi kuitenkin tuoda velan julki myös muille sidosryhmille. Muut ohjelmistoprojektiin liittyvät sidosryhmät voivat ymmärtää paremmin teknisen velan vaikutukset, jos ne esitetään heille esimerkiksi velan aiheuttamina kustannuksina tai lisääntyvän työn määränä. Velan näkyvyys kaikille sidosryhmille on tärkeää, jotta ryhmät pystyvät tekemään sen perusteella päätöksiä koskien projektin etenemistä ja teknisen velan hallintaa.

Useat erilaiset tekijät voivat aiheuttaa teknistä velkaa. Erään tutkimuksen mukaan yleisin syy teknisen velan syntyyn on tuotteen lähestyvän julkaisun tai muun määräajan aiheuttama aikataulupaine [24]. Muita yleisiä syitä ovat muun muassa riittämätön suunnittelu, ohjelmointiosaamisen puute, selkeästi määritellyn ohjelmistoprosessin puute sekä sovittujen ohjelmointikäytäntöjen puute. Ohjelmistokehittäjien osaamiseen ja työkokemukseen liityvät teemat ovat myös yksi merkittävä kokonaisuus teknisen velan syntymisessä. Ylipäänsä ei-tekniset seikat nousevat tutkimuk-

sessa teknisen velan merkittävimmiksi lähteiksi.

Ohjelmiston sisältämä perinnekoodi on usein myös yksi teknisen velan merkittävimmistä lähteistä [16]. Perinnekoodi voi sisältää esimerkiksi vanhentuneita ohjelmointikäytäntöjä tai ohjelmointikielen ominaisuuksia, joita on muutettu kielen uudemmissa versioissa. Perinnekoodin edelleen jatkunut kehitystyö voi kasvattaa teknisen velan määrää merkittävästi. Ohjelmiston heikko arkkitehtuuri voi myös altistaa ohjelmiston teknisen velan synnylle. Heikko arkkitehtuuri on usein kuitenkin seurausta esimerkiksi puutteellisesta suunnittelutyöstä tai heikosta ohjelmointikokemuksesta.

Teknistä velkaa voi syntyä ohjelman lähdekoodiin joko tarkoituksellisesti tai tahattomasti [19, 14]. Tarkoituksella otettu tekninen velka on aina tietoinen päätös ja sen avulla pyritään esimerkiksi saamaan tuote markkinoille ennen kilpailijoita tai saavuttamaan tiettyjä määräaikoja ohjelmiston kehityksessä. Teknistä velkaa otettaessa ohjelmiston kehitystyötä nopeutetaan esimerkiksi hyväksymällä lähdekoodissa nopeasti tehdyt toteutukset, jotka toimivat mutta eivät ole yleisesti tunnustettujen tai organisaation sisällä sovittujen ohjelmointikäytäntöjen mukaisia. Ohjelmiston lähdekoodin tai rakenteen laatu saattaa tällöin kärsiä. Tarkoituksella otettua teknistä velkaa tulisi myös hallita ja ylläpitää niin, että tulevaisuudessa vielä muistettaisiin, missä osissa lähdekoodia teknistä velkaa konkreettisesti on. Teknistä velkaa voi syntyä puolestaan tahattomasti esimerkiksi kokemattoman tai osaamattoman ohjelmoijan tuottaessa huonoa ohjelmakoodia tai projektille asetettujen liian tiukkojen määräaikojen vuoksi, jolloin ohjelmistokehittäjät joutuvat työskentelemään kovan paineen alla.

Steve McConnell määrittelee kirjoituksessaan tarkoituksella otetulle tekniselle velalle pitkäaikaisen ja lyhytaikaisen velan [19]. Lyhytaikaista velkaa otetaan yleensä taktisista syistä. Sen avulla voidaan reagoida esimerkiksi tuotteen lähestyvään toimitusmääräaikaan. Tällöin kehitystyötä nopeutetaan hetkellisesti tuotteen sisäisen rakenteen ja ohjelmakoodin laadun kustannuksella. Pitkäaikainen tekninen velka tarkoittaa tyypillisesti ennakoidusti tehtyjä, strategisia ja pitkäaikaisia suunnitteluratkaisuja. Esimerkiksi päätös ohjelmiston kehittämisestä aluksi vain yhdelle alustalle/käyttöjärjestelmälle ja lisätä laajempi alustatuki myöhemmin voisi olla pitkäaikaista teknistä velkaa. Lyhytaikainen tekninen velka pitäisi tyypillisesti maksaa pois pian velan ottamisen jälkeen, mahdollisesti jo tuotteen seuraavassa kehitysiteraatiossa. Pitkäaikainen velka voi olla tuotteessa pidempään, velan tyypistä riippuen jopa vuosia.

Kun ohjelmistoprojektissa otetaan tarkoituksella teknistä velkaa, tulisi velan oton syynä olla, että huolelliseen kehitystyöhön panostaminen velanotto hetkellä nähdään kalliimpana kuin tulevaisuudessa [19]. Syitä tähän voivat olla esimerkiksi:

- Ohjelmiston kiireellinen julkaisuaikataulu
- Pääoman säästäminen
- Poistuvan ohjelmiston kehittäminen

Kun ohjelmiston saaminen nopeasti markkinoille on kriittistä esimerkiksi kilpailuedun saavuttamiseksi, huolelliseen kehitystyöhön panostaminen saattaa aiheuttaa ohjelmistosta saatavien tuottojen menetyksiä. Jos esimerkiksi teknistä velkaa ottamalla yritys saa oman tuotteensa kuukautta ennen kilpailijaa markkinoille, voi teknisen velan ottaminen olla kannattavaa. Tämä riippuu tietysti otettavan teknisen velan määrästä ja sen mahdollisista takaisinmaksusta aiheutuvista kustannuksista. Mikäli teknisen velan koroista ja takaisinmaksusta aiheutuvat kustannukset ovat saatavia tuottoja huomattavasti pienempiä ja ohjelman julkaisun nopeuttamisella saavutetaan merkittävää kilpailuetua, on teknisen velan ottaminen kannattavaa.

Esimerkiksi start-up-yritysten tulot voivat olla niukat tai olemattomat, jolloin joudutaan toimimaan hyvin rajoitetuilla varoilla. Tällöin tulisikin keskittyä saamaan tuote nopeasti valmiiksi. Teknisen velan ottaminen tässä tapauksessa ja kulujen lykkääminen myöhemmäksi voi olla hyvä ratkaisu. Tällöin säästetään yritystoimintaan varattuja niukkoja varoja ja teknistä velkaa voidaan maksaa pois myöhemmin, kun tuote on valmis ja tulot ovat vakiintuneet.

Kun ohjelmisto poistuu kehityksestä, poistuu sen mukana myös sen sisältämän teknisen velan tuoma ylimääräinen rasite. Tekninen velka on rasite lähdekoodissa ainoastaan, mikäli velkaa sisältävän lähdekoodin parissa työskennellään runsaasti. Kehityksestä poistunutta ohjelmistoa ei enää jatkokehitetä, joten ei ole väliä kuinka paljon teknistä velkaa se sisältää. Mitä vähemmän velkaa sisältävän koodin parissa työskennellään ja mitä lähempänä ohjelmisto on kehityskaarensa loppua, sitä herkemmin voidaan harkita teknisen velan ottamista. Ohjelmiston kehityskaaren loppuvaiheessa on tärkeämpää tuottaa toimivaa koodia kuin käyttää aikaa koodin siisteyteen ja tyylipuhtauteen. Mikäli tekninen velka kuitenkin jollain tavalla heijastuu negatiivisesti esimerkiksi ohjelmiston loppukäyttäjälle, tulee siihen kiinnittää huomiota myös kehityskaaren lopulla.

2.2 Moderneja ohjelmistokehitysmalleja

Tässä luvussa esitellään joitakin 2000-luvulla yleistyneitä ohjelmistokehitysmalleja ja -käytäntöjä, joita tämän työn tutkimusosiossa mukana olevat yritykset hyödyntävät. Moderneilla ohjelmistokehitysmalleilla tarkoitetaan tässä yhteydessä vesiputousmallin ja muiden perinteisten prosessimallien vanavedessä kehitettyjä ketteriä menetelmiä ja näistä edelleen jalostettuja joustavia ohjelmistokehitysmalleja.

Moderneille ohjelmistokehitysmalleille tyypillisiä piirteitä ovat muun muassa iteratiivinen kehityssykli, tiivis yhteistyö asiakkaan kanssa, kehitystyön korostaminen sekä pitkälle automatisoidut työkaluketjut. Modernit ohjelmistokehitysmallit soveltuvat toimimaan muuttuvassa ympäristössä sekä vastaamaan jatkuvasti muuttuviin ohjelmistovaatimuksiin.

2.2.1 Ketterät menetelmät

Ketteriksi menetelmiksi kutsutaan joukkoa ohjelmistokehitysmalleja ja -menetelmiä, jotka perustuvat niin kutsuttuun ”ketterän ohjelmistokehityksen julistukseen” (eng. Agile manifesto) [3]. Ketterän ohjelmistokehityksen julistus on joukko arvoja ja periaatteita, joita ketterät ohjelmistokehitysmallit noudattavat. Ketterien menetelmien avulla voidaan reagoida muutokseen ja toimia epävarmassa ja jatkuvasti muuttuvassa ympäristössä [45]. Ketterän ohjelmistokehityksen julistus kuuluu:

- Yksilöitä ja kanssakäymistä enemmän kuin menetelmiä ja työkaluja.
- Toimivaa ohjelmistoa enemmän kuin kattavaa dokumentaatiota.
- Asiakasyhteistyötä enemmän kuin sopimusneuvotteluja.
- Vastaamista muutokseen enemmän kuin pitäytymistä suunnitelmassa.

Ketterän ohjelmistokehityksen julistuksen takana ovat seuraavat 12 periaatetta [4]:

- Tärkein tavoitteemme on tyydyttää asiakas toimittamalla tämän tarpeet täyttäviä versioita ohjelmistosta aikaisessa vaiheessa ja säännöllisesti.
- Otamme vastaan muuttuvat vaatimukset myös kehityksen myöhäisessä vaiheessa. Ketterät menetelmät hyödyntävät muutosta asiakkaan kilpailukyvyn edistämiseksi.
- Toimitamme versioita toimivasta ohjelmistosta säännöllisesti, parin viikon tai kuukauden välein, ja suosimme lyhyempää aikaväliä.
- Liiketoiminnan edustajien ja ohjelmistokehittäjien tulee työskennellä yhdessä päivittäin koko projektin ajan.
- Rakennamme projektit motivoituneiden yksilöiden ympärille. Annamme heille puitteet ja tuen, jonka he tarvitsevat ja luotamme siihen, että he saavat työn tehtyä.
- Tehokkain ja toimivin tapa tiedon välittämiseksi kehitystiimille ja tiimin jäsenten kesken on kasvokkain käytävä keskustelu.
- Toimiva ohjelmisto on edistymisen ensisijainen mittari.
- Ketterät menetelmät kannustavat kestävään toimintatapaan. Hankkeen omistajien, kehittäjien ja ohjelmiston käyttäjien tulisi pystyä ylläpitämään työtahtinsa hamaan tulevaisuuteen.

- Teknisen laadun ja ohjelmiston hyvän rakenteen jatkuva huomiointi edesauttaa ketteryyttä.
- Yksinkertaisuus - tekemättä jätettävän työn maksimointi - on oleellista.
- Parhaat arkkitehtuurit, vaatimukset ja suunnitelmat syntyvät itseorganisoituvissa tiimeissä.
- Tiimi tarkastelee säännöllisesti, kuinka parantaa tehokkuuttaan, ja mukauttaa toimintaansa sen mukaisesti.

Ketteryys on ennen kaikkea ajattelutapa, joita edellä mainitut periaatteet ohjaavat [45]. Periaatteet ohjaavat ja antavat vihjeitä miten toimia epävarmassa, muuttuvassa toimintaympäristössä. Kuhunkin eri kontekstiin sopivat erilaiset menetelmät ja työtavat. Kontekstin perusteella valitaan siihen soveltuvimmat menetelmät ketteriin periaatteisiin pohjautuen. Ei ole yhtä tiettyä tapaa toimia ketterästi. Ketteryyteen liittyy vahvasti omaan toimintaympäristöön liittyvien erityispiirteiden ja epävarmuuksien tunnistamista sekä niihin sopeutumista.

Kaikille ketterille menetelmille yhteistä on iteratiivinen ohjelmistokehitys [9]. Sen sijaan, että tehtäsiin kerralla valmis kokonainen ohjelmisto, sitä kehitetään palanen kerrallaan yhteistyössä asiakkaan kanssa. Iteratiivisen kehityksen tavoitteena on nopeasti tuottaa toimiva ohjelmisto, joka tuottaa arvoa asiakkaalle. Kehitystyössä ohjelmistoon lisätään uusia ominaisuuksia yksi kerrallaan pyrkien pitämään ohjelmisto koko ajan tuotantokelpoisena. Asiakas pyritään ottamaan ohjelmiston kehitykseen mahdollisimman tiiviisti mukaan alusta alkaen, jolloin hän pystyy vaikuttamaan ohjelmistoon tehtäviin muutoksiin ja siihen lisättäviin ominaisuuksiin.

Ketterissä menetelmissä korostuvat tyypillisesti poikkitieteelliset, itseohjautuvat tiimit, joiden uskotaan löytävän oikeat menetelmät ja ratkaisut kussakin kontekstissa, pitkälti ilman korkeammalta taholta tulevaa yksityiskohtaista ohjausta [45]. Organisaation tehtävänä on tarjota tiimeille näiden tarvitsemat työkalut, työtilat ja resurssit, jotta tiimit pystyvät häiriöttä työskentelemään tavoitteidensa saavuttamiseksi. Tiimi tapaa keskenään tyypillisesti päivittäin lyhyessä palaverissa, jossa käydään läpi muun muassa projektin tilaa, työntekijöiden vastuita ja mahdollisia ongelmia. Ketteriin menetelmiin kuuluu myös tietyin väliajoin tehtävä laajempi arvio projektin ja työskentelyn onnistumisesta, jonka avulla on tarkoitus edelleen kehittää organisaation toimintaa pitkällä aikavälillä.

2.2.2 Scrum

Scrum on prosessiviitekehys, jota hyödyntäen voidaan kehittää, toimittaa ja ylläpitää monimutkaisia tuotteita [27]. Scrum pyrkii tarjoamaan raamit monimutkaisten

ongelmien ratkaisemiseksi, kun kehitetään tehokkaasti ja luovasti mahdollisimman suurta lisäarvoa tarjoavia tuotteita. Scrum ei ole prosessi tai lopullinen työskentelymenetelmä, vaan se tarjoaa pikemminkin työskentelyraamit eri tekniikoiden ja menetelmien soveltamiseen. Scrumia voidaan ohjelmistotuotannon lisäksi soveltaa useilla eri teollisuudenaloilla ja erilaisissa ympäristöissä. Scrum voidaan lukea kuuluvan ketteriin menetelmiin [7].

Scrum perustuu kolmelle periaatteelle: läpinäkyvyys, tarkastelu ja sopeutuminen [27]. Prosessiin liittyvien merkittävien tekijöiden tulee olla helposti havaittavissa niille, jotka ovat vastuussa prosessin lopputuloksesta. Läpinäkyvyys vaatii, että merkittävät tekijät on määritelty yhteisesti, jotta kaikilla prosessiin osallistuvilla on niistä yhtenäinen näkemys: esimerkiksi selkeästi määritelty, yhteinen ”valmiin” määritelmä. Scrumin soveltajien on tasaisin väliajoin tarkasteltava prosessin tuotoksia ja työn edistymistä kohti sprintin tavoitteita havaitakseen mahdolliset haitalliset poikkeamat prosessissa. Tarkastelua ei tule kuitenkaan toimittaa niin usein, että se haittaa työn edistymistä. Mikäli työn tarkastelussa havaitaan työn kannalta haitallisia poikkeamia, jotka estävät työn lopputulosta saavuttamasta sille määriteltyjä tavoitteita, on prosessia tai prosessoitavaa työtä muokattava. Muokkaukset tulisi tehdä mahdollisimman nopeasti, jotta myöhemmät poikkeamat saadaan minimoitua.

Scrumia toteuttavat scrum-tiimit, jotka koostuvat tuoteomistajasta, kehitystiimistä sekä scrummasterista [27]. Tuoteomistaja vastaa, että kehitystiimin kehittämän tuotteen arvo saadaan maksimoitua. Kehitystiimin vastuulla on tuotteen kehittäminen ja uuden inkrementin saaminen valmiiksi sprintin loppuun mennessä. Scrummaster vastaa, että Scrumia toteutetaan oikeaoppisesti ja Scrumia toteuttavat tahot saavat riittävästi valmennusta ja neuvontaa Scrumin toteuttamiseksi. Scrum-tiimit ovat itseohjautuvia ja koottu siten, että ne kykenevät suorittamaan niille määritellyt tehtävät ilman riippuvuuksia ryhmän ulkopuolisiin tahoihin. Monitaitoiset ryhmät päättävät itse, miten suorittavat heille määritellyt tehtävät ilman korkeammalta taholta tulevaa yksityiskohtaista ohjausta.

Scrumin ytimenä toimii sprintti, jonka aikana scrum-tiimi kehittää tuotteesta ”valmiin” määritelmän täyttävän, käyttökelpoisen ja mahdollisesti julkaisukelpoisen inkrementin [27]. Sprintti on esimerkiksi 2-4 viikkoa kestävä aikajakso, ja jokaiselle sprintille on määritelty tavoitteet, jotka sprintin aikana tulisi saavuttaa. Sprintin päätyttyä alkaa uusi sprintti ja kukin sprintti koostuu tuotteen kehitystyön lisäksi Scrum-tapahtumista. Scrumin tapahtumia ovat:

- Sprintin suunnittelu
- Päivittäispalaveri
- Sprintin katselmointi

- Retrospektiivi

Sprintin suunnittelussa Scrum-tiimi luo yhdessä suunnitelman tulevan sprintin töistä ja tavoitteista. Suunnittelupalaverissa käydään läpi, mitä alkavassa sprintissä on mahdollista toimittaa ja miten työt järjestellään. Suunnittelupalaverissa tuoteomistaja ja kehitystiimi järjestävät tuotteen kehitysjonon yhdessä siten, että sprintille asetettavat tavoitteet on mahdollista saavuttaa.

Päivittäispalaveri on kehitystiimin päivittäin toteuttama lyhyt palaveri, jossa kehitystiimi suunnittelee työnsä seuraavaksi 24 tunniksi. Päivittäispalaverissa voidaan muun muassa käydä läpi, miten työ kohti sprintin tavoitteita etenee ja mitä mahdollisia ongelmia työssä on tullut vastaan. Päivittäispalavereilla pyritään parantamaan tiimin sisäistä kommunikointia sekä vähentämään ylimääräisten palavereiden tarvetta.

Sprintin päätteeksi järjestettävässä katselmoinnissa scrum-tiimi käy läpi yhdessä muiden sidosryhmien kanssa, mitä sprintin aikana tehtiin. Tähän tietoon sekä mahdollisiin kehitysjonoon tehtyihin muokkauksiin perustuen katselmointiin osallistuvat tahot arvioivat yhdessä, mitä tulisi kehittää jatkossa arvotuoton optimoimiseksi. Katselmoinnin tuloksena on tuotteen tarkastettu kehitysjono, joka voi toimia pohjana seuraavan sprintin kehitysjonoksi.

Retrospektiivi on sprintin katselmoinnin jälkeen järjestettävä Scrum-tiimin sisäinen tapaaminen, jossa arvioidaan päättynyttä sprinttiä. Retrospektiivissä käydään läpi, mikä onnistui ja missä on parannettavaa koskien esimerkiksi työtapoja, työkaluja ja kommunikointia. Retrospektiivissä luodaan suunnitelma parannusten toteuttamiseksi seuraavassa sprintissä. Retrospektiivi pyrkii parantamaan Scrum-tiimin työskentelyä ja sopeutumista läpi sprinttien.

2.2.3 Lean-ohjelmistokehitys

Lean-ajattelun juuret ovat alunperin Japanin autoteollisuudessa [22]. Lean-ajattelu perustuu turhien asioiden ja toimintojen poistamiseen tuotantoprosessista. Tuotantoprosessissa pyritään pitämään varastot mahdollisimman pieninä ja komponentteja pyritään valmistamaan pienissä erissä sitä mukaan, kun niitä tarvitaan. Lean-ajattelun periaatteita voidaan soveltaa myös ohjelmistotyöhön.

Lean-ohjelmistokehitys perustuu Lean-periaatteiden soveltamiseen ohjelmistotyössä [22]. Lean-ohjelmointi keskittyy itse ohjelmistoprosessin tarkan kuvaamisen sijaan ratkaisemaan, miten asiakkaille voidaan tuottaa lisäarvoa ohjelmistojen avulla.

Lean-ohjelmistotyö esittelee joukon periaatteita, joiden raameissa eri ohjelmistokäytäntöjä voidaan soveltaa. Suoraviivaisen ohjelmistoprosessin sijaan Lean-ohjelmistotyö on näin ollen enemmän yhdistelmä ohjelmistokäytäntöjä, -periaatteita ja asiakaslähtöisen ohjelmistokehityksen filosofiaa. Lean-ohjelmistotyö perustuu seitsemään periaatteeseen [23]:

- Hukan eliminointi
- Oppimisen korostaminen
- Päätösten lykkääminen mahdollisimman myöhäiseksi
- Nopea toimitus
- Tiimien osallistaminen
- Tuotteen laatu
- Kokonaisuuden havaitseminen

Leanissa hukalla tarkoitetaan kaikkea, joka ei suoraan lisää asiakkaalle tuotettavaa arvoa eikä myöskään lisää tietoa, miten arvoa voidaan tuottaa tehokkaammin tulevaisuudessa [22]. Ohjelmistotyössä hukkaa ovat esimerkiksi ohjelmistossa olevat, asiakkaan kannalta turhat ominaisuudet, kehitystyön aikana hukattu tieto, puolivalmiiksi jätetyt työt ja ohjelmistossa olevien virheiden etsimiseen ja korjaamiseen käytettävä aika ohjelmiston eteenpäin kehittämisen sijaan. Lean-ajattelussa ihannetilanne olisi selvittää, mitä asiakas tarkalleen haluaa, toteuttaa se ja lopulta toimittaa asiakkaalle tämän vaatimukset täysin täyttävä tuote, joka ei kuitenkaan sisällä mitään ylimääräistä [23]. Kaikki, jotka jollain tavalla haittaavat työtä asiakkaan tarpeiden täyttämiseksi, ovat hukkaa. Lean-ajattelun periaatteiden mukaan organisaatioiden tulisi pyrkiä eliminoimaan ohjelmistotyön hukkaa mahdollisimman tehokkaasti.

Lean-ajattelussa korostetaan jatkuvaa oppimista työn edetessä [23]. Leanissa ajatellaan, että paras tapa parantaa ohjelmistotyötä on vahvistaa työntekijöiden oppimista. Paras lopputulos kehitettävästä tuotteesta ei välttämättä synny ensimmäisellä yrityksellä, vaan vaatii useampia yrityksiä. Tämän myötä työntekijät kehittyvät myös paremmiksi töissään. Tuotteesta voidaan aluksi kehittää toimiva prototyyppi, joka sisältää pienimmän mahdollisen määrän toiminnallisuutta ja ominaisuuksia [22]. Tätä tuotetta kehitetään edelleen asiakkaalta kerättävän palautteen avulla, jolloin voidaan välttyä muun muassa asiakkaan kannalta turhien ominaisuuksien kehittämiseltä.

Kriittisten päätösten lykkääminen myöhimmälle mahdolliselle hetkelle mahdollistaa kyseisten päätösten tekemisen parhailla mahdollisilla, siihen mennessä kerätyillä tiedoilla [22]. Päätösten tekeminen myöhimmällä mahdollisella hetkellä parantaa päätösten mahdollisuutta osoittautua onnistuneiksi, kun päätökset perustuvat

enemmän faktatietoon kuin spekulointiin ja ennusteisiin. Epävarmoilla ja kehittyvillä markkinoilla on kannattavampaa jättää itselleen vaihtoehtoja, joista on varaa valita, kun valinnan aika koittaa [23]. Tämä voisi tarkoittaa ohjelmistotyössä esimerkiksi tuotteen kehittämistä siten, että tuotteeseen rakennetaan kapasiteettia muutosta varten.

Lean-ajattelussa korostetaan tuotteiden nopean julkaisun ja toimituksen tärkeyttä [23]. Nopeus tuotteen kehityksessä ja toimittamisessa asiakkaalle nähdään parempana kuin varovainen, ”älä tee yhtään virheitä” -tyyppinen lähestymistapa. Nopea toimitus mahdollistaa aiempaa enemmän ja luotettavampaa palautetta asiakkaalta. Nopeus edesauttaa oppimista, kun kehityssykli ”suunnittelu, toteutus, palaute ja parantaminen” toistuu mahdollisimman tiheästi. Nopea toimitus mahdollistaa kehitystyön pysymisen asiakkaan tarpeiden muutosten mukana. Modernit testaus- ja julkaisutyökalut mahdollistavat nopean toimituksen toteuttamisen [22].

Tiimien osallistamisella tarkoitetaan muun muassa tuotteen yksityiskohtia koskevien päätösten siirtämistä heille, jotka tietävät asiasta eniten: tuotteen kehittäjät [23]. Ohjelmistokehittäjät tuntevat parhaiten kehittämänsä tuotteen yksityiskohdat, joten heidän sisällyttäminen päätöksentekoon on olennaista onnistuneen kehitystyön kannalta. Riittävällä ammattitaidolla varustetut ja hyvin johdetut työntekijät pystyvät tekemään parhaat päätökset tuotteen teknisistä seikoista. Lisäksi, kun Leanissa päätökset pyritään tekemään mahdollisimman myöhään ja kehitysvauhti on nopeaa, eivät kehittäjien yläpuolelta annetut päätökset tulisi riittävän ajoissa pysyäkseen mukana kehitystyössä.

Lean-ajattelussa panostetaan tuotteen eheyteen ja laatuun [23]. Eheä ja laadukas ohjelmisto on ylläpidettävä, mukautuva ja laajennettava sekä yhtenäinen arkkitehtuuriltaan, hyvä käytettävyydeltään ja sopii käyttötarkoitukseensa. Järjestelmän eri osien toimiessa hyvin yhteen ja muodostaen näin saumattomasti toimivan kokonaisuuden, myös ohjelmiston sisäisestä laadusta rakentuu eheä. Tämä on merkittävä tekijä myös loppukäyttäjän kokeman laadun kannalta. Hyvä johtajuus, työntekijöiden asiantuntemus, tehokas kommunikointi ja terve kuri kehitystyössä edesauttavat eheyden rakentamista. Kun ohjelmistoa kehitetään osa kerrallaan ja osien korkea laatu varmistetaan esimerkiksi automaattisilla laadunvarmistustyökaluilla, rakentuu ohjelmistosta eheä ja laadukas kokonaisuus [22].

Ohjelmisto on sen sisältämien eri järjestelmien keskenään vallitsevien vuorovaikutusten tuote [23]. Ohjelmiston sisältämät eri järjestelmät ja näiden välinen vuorovaikutus muodostavat kokonaisuuden, joka määrittää miten ohjelmisto toimii. Näin ollen yksittäiset, hyvin toimivat järjestelmät eivät takaa koko ohjelmiston hyvää toimivuutta, vaan sen määrittää miten järjestelmien muodostama kokonaisuus toimii.

Leanissa keskitytäänkin kokonaisuuden optimointiin, jolla pyritään varmistamaan tuotteen laatu ja hyvä toimivuus.

2.2.4 Kanban

Kanban on Lean-ajattelun ohella niin ikään lähtöisin Japanin autoteollisuudesta [38]. Se kehitettiin alunperin Toyotalla tuotannonhallintajärjestelmäksi *just-in-time*-tuotantoympäristöön. Kanban on sittemmin omaksuttu menetelmäksi myös ohjelmistotyöhön [1].

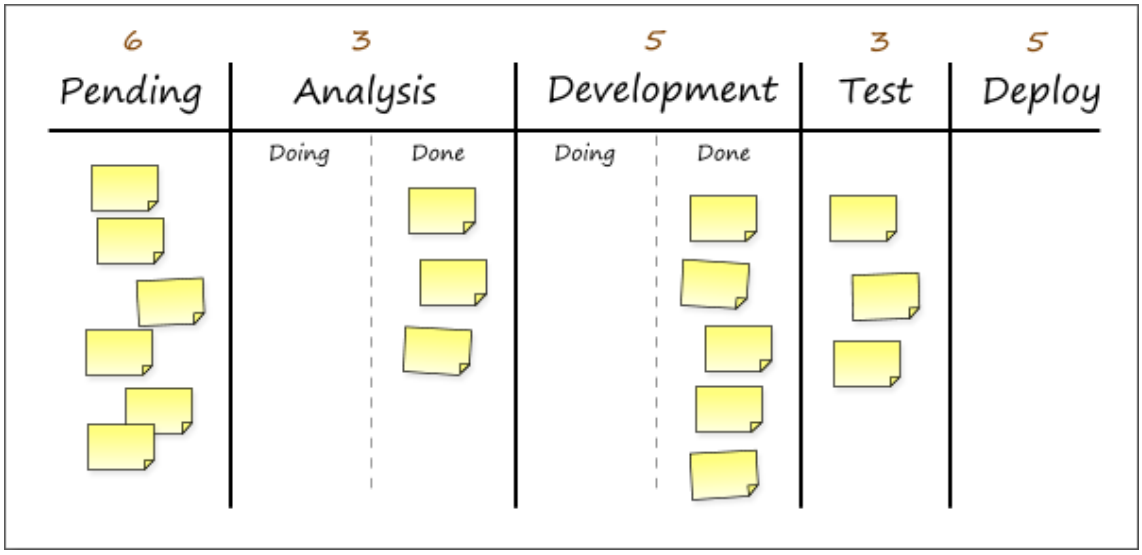
Kanban mahdollistaa esimerkiksi Lean-periaatteiden soveltamisen tarjoamalla työkalun arvotuoton optimointiin ja työvirran hallintaan. Kanban koostuu viidestä periaatteesta, jotka ovat [1]:

- Työvirran visualisointi
- Keskeneräisen työn rajoittaminen
- Työvirran arviointi ja hallinta
- Yksiselitteiset prosessikäytännöt
- Kehitysmahdollisuuksien tunnistaminen mallien avulla

Kanbanissa työvirta visualisoidaan esimerkiksi fyysisellä taululla, jossa kuvataan työvirran vaiheet eri tiloina [1]. Tilat voisivat olla esimerkiksi ”valmistelu”, ”keskeneräinen” ja ”valmis”. Kutakin työsuoritetta kuvataan yhdellä kortilla ja kortteja siirrellään taululla tilojen välillä sen mukaan, missä vaiheessa työvirtaa kyseinen työsuorite todellisuudessa kulloinkin on. Kuvassa 2.1 on esimerkki Kanban-taulusta.

Fyysinen taulu ja työvirran visualisointi mahdollistavat työsuoritteiden seurannan, ja työntekijöiden on mahdollista keskenään organisoida työnsä ilman johdon yksityiskohtaista puuttumista [1]. Työtiloille määritellään enimmäismäärä työsuoritteita, joita kukin tila saa yhtäaikaisesti sisältää. Näin voidaan rajoittaa esimerkiksi keskeneräisten työsuoritteiden liian suurta määrää. Kanban-taulun ei välttämättä tarvitse olla fyysinen, ja verkossa onkin tarjolla useita eri web-applikaatioita, jotka havainnollistavat Kanban-menetelmää digitaalisesti.

Kanban-taulu mahdollistaa työvirran arviointia [1]. Itse taulun lisäksi työvirtaa on mahdollista havainnollistaa erilaisilla kuvaajilla esimerkiksi *edistymiskäyrällä* (eng. burn-down chart) tai *arvovirtakuvauksella* (eng. value stream map). Kanban-taulu antaa jo nopealla vilkaisulla yleiskuvan esimerkiksi keskeneräisten työsuoritteiden määrästä ja tarkemmalla seurannalla sen avulla voidaan arvioida muun muassa työsuoritteiden läpimenoaikaa.



Kuva 2.1 *Esimerkki Kanban-taulusta, johon on määritelty viisi eri tilaa. Keltaiset kortit kuvaavat työsuoritteita. Oikeasti käytettävässä Kanban-taulussa kuhunkin korttiin olisi lisäksi kirjoitettu työsuoritteen lyhyt kuvaus [46].*

Jotta Kanban-taulu voisi toimia usean henkilön käyttämänä työkaluna, on tilojen välille määriteltävä yksiselitteiset tulo- ja lähtökriteerit, joiden perusteella työsuoritteita voidaan siirtää tilasta toiseen [1]. Kriteerit mahdollistavat muun muassa syy-seuraussuhteen tarkkailun, jos työprosessiin tehdään jotain muutoksia. Tarkat kriteerit auttavat lisäksi tasapainottamaan työvirtaa.

Kanbanin periaatteisiin liittyy lisäksi kehitysmahdollisuuksien tunnistaminen erityisten mallien avulla [1]. Malleista mainitaan esimerkkinä *kapeikkoajattelu* (eng. theory of constraints) sekä *systeemiajattelu* (eng. systems thinking).

2.2.5 DevOps

DevOps on uudehko toimintamalli ohjelmistoalalla, mikä pyrkii parantamaan yhteistyötä ohjelmistokehityksen ja järjestelmätuotantotoimintojen välillä [44]. Termi ”DevOps” on muodostettu englannin kielisistä sanoista *development* ja *operations*. Ohjelmistokehittäjien käyttämät työmenetelmät, ohjelmistotuotantoprosessit ja työkalut ovat kehittyneet viime vuosikymmeninä paljon, ja ohjelmistokehitys on nopeutunut. Tämä kehitys on kuitenkin painottunut ohjelmistokehitykseen jättäen järjestelmätuotantotoiminnot jälkeensä. Ohjelmistokehittäjät pystyvät siis tuottamaan ohjelmakoodia nopeammin kuin järjestelmätuotantotoiminnot ehtivät sitä käsittelemään. DevOps on kehittynyt ketterien menetelmien ja Lean-kehityksen vanavedessä, vastaamaan edellä mainittuihin haasteisiin.

DevOps:lle ei ole vielä vakiintunutta määritelmää alan kirjallisuudessa, ja se voi-

daan ymmärtää monesta eri näkökulmasta [25]. DevOps:iin voidaan liittää kuitenkin muun muassa seuraavia käsitteitä: ohjelmistokehittäjien ja järjestelmätuotantotoimintojen välisen kuilun kaventaminen ja poistaminen, kommunikaation ja yhteistyön korostaminen sekä jatkuva integraatio, jatkuva testaus ja jatkuva käyttöönotto [18].

DevOps voidaan ymmärtää kokonaisena työkulttuurina, jota automaattiset työkalut tukevat. [47]. DevOps korostaa eri osastojen välistä yhteistyötä organisaation sisällä. DevOps-kulttuurissa ei oikeastaan ole mielekästä puhua lainkaan osastoista, joilla on tietyt, tarkasti rajatut vastuualueet, vaan osastojen välisiä rajoja pyritään häivyttämään ja vastuita jakamaan osastorajojen yli. Esimerkiksi ohjelmistokehittäjillä voi olla vastuullaan myös tuotteen ylläpitoon liittyviä tehtäviä, jolloin tuotteeseen kehitetään todennäköisemmin käyttöönottoa ja ylläpitoa edesauttavia ominaisuuksia. Yhteistyötä tulisi korostaa sijoittamalla yhteistyötä tekevät tahot konkreettisesti samoihin työtiloihin. Osastojen välisten määritelmädokumentaatioiden laatimisen ja työn siirtelyn sijaan, projektiin osallistuvien tulisi suosia ratkaisun hakemista yhdessä kehityskaaren alusta lähtien. DevOps häivyttää ohjelmistokehityksen ja järjestelmätuotantotoimintojen välistä eroa ja voi lopulta poistaa sen kokonaan.

Yhteistyön lisäksi autonomisten tiimien mahdollistaminen on tärkeässä roolissa DevOps-kulttuurissa [47]. Tehokas yhteistyö edellyttää tiimeiltä itsenäisten päätösten ja muutosten tekemistä ilman monimutkaista päätöksentekomenettelyä. Tämä edellyttää organisaatiolta luottoa tiimien päätöksentekokykyyn ja virheiden tekemisen pelko on poistettava. Lisäksi riskienhallinta on muutettava vastaamaan uutta kulttuuria.

Työvaiheiden automatisointi on välttämätöntä menestyksekkään DevOps-kulttuurin luomisessa [47], ja se voidaan lukea kuuluvaksi myös osaksi DevOpsin määritelmää [18]. Työvaiheiden automatisointi edesauttaa yhteistyön onnistumista sekä vapauttaa ihmisiä ja resursseja keskittymään tärkeämpiin asioihin. Samalla automatisointi vähentää inhimillisten virheiden määrää ja toimii kehitettävän ohjelmiston dokumentaationa.

2.3 Modernien ohjelmistokehitysmallien suhteesta tekniseen velkaan

Ketterien ohjelmistokehitysmallien suhdetta tekniseen velkaan on tutkittu alan kirjallisuudessa jonkin verran. Ketterien menetelmien on havaittu helpottavan teknisen velan hallintaa [16][17], mutta samalla ketterät menetelmät koetaan olevan myös alttiina teknisen velan syntymiselle [5].

Ketterissä menetelmissä sovellettujen työskentelymenetelmien on todettu vaikuttavan positiviisesti teknisen velan hallintaan [16][17]. Esimerkiksi *testivetoinen kehitys* (eng. test-driven development), sovitut ohjelmointistandardit ja muut vastaavat menetelmät, jotka liittyvät lähimmin itse toteutukseen, todetaan olevan tehokkaimpia teknisen velan hallinnassa. Säännöllisesti kehitysiteraatioissa käytävillä arvioinneilla ja retrospektiiveillä todetaan olevan myös positiivinen vaikutus velan hallintaan. Yleinen ja tehokas tapa hallita teknistä velkaa on koodin refaktorointi [5][17]. Samalla kuitenkin todetaan, että refaktorointi voi olla tehokkaampaa pienissä kuin isoissa projekteissa ja sen käyttäminen monimutkaisissa ohjelmistoissa voi olla vaikeaa ja aikaa vievää. Teknisen velan havainnointi ja sen näkyväksi tekeminen ovat myös avainasemassa velan hallitsemisessa.

Ketterien menetelmien alttius teknisen velan syntymiselle on tyypillisesti monen tekijän summa. Ketterät menetelmät korostavat toimivan ohjelmiston toimittamista nopeasti asiakkaalle tarkkojen suunnitelmien ja dokumentaatioiden laatimisen sijaan, mikä voi altistaa herkemmin teknisen velan synnylle [13]. Ohjelmistoarkkitehtuurin suunnittelun jättäminen vähälle huomiolle voi altistaa ohjelmiston teknisen velan synnylle myöhemmin kehitystyön aikana [5]. Nopea kehitys- ja julkaisutahti luovat painetta ohjelmistokehittäjille, joiden olisi samalla tuotettava myös arvoa asiakkaalle. Ohjelmistokehittäjien on näin ollen tasapainoiltava arvontuoton ja aikataulupaineiden välillä, jolloin he ovat pakotettuja oikomaan toteutuksissaan ja aiheuttamaan koodiin teknistä velkaa. Nopean kehitystahdin ja huonon kehitystyön sekä heikkojen testaus- ja laadunvarmistusmenetelmien seurauksena organisaatiot ovat herkkiä kerryttämään teknistä velkaa [26]. Ketterien menetelmien ja modernien ohjelmistokehitysmallien soveltajien on näin ollen otettava tekninen velka jollain tavalla huomioon kehitysprosesseissaan. Huomioimalla tekninen velka riittävästi organisaatio voi muovata sen itselleen työkaluksi, jonka avulla se voi saavuttaa uusia liiketoimintamahdollisuuksia.

3. TUTKIMUSAINEISTO JA -MENETELMÄT

Tässä luvussa esitellään työssä käytetyt tutkimusaineistot ja kuvataan tutkimusmenetelmät. Työn tutkimusosuus toteutettiin tutkimalla kahta eri ohjelmistoprojektia. Kyseessä olivat Vaadin Oy:n osin avoimeen lähdekoodiin perustuva Vaadin Framework 7 web-kehitystyökalu ja Yleisradio Oy:n Yle Areena.

Tutkimusosuus suoritettiin Vaadin Framework 7:n ja Yle Areenan lähdekoodien staattisilla analyyseillä. Lähdekoodien analysointiin käytettiin SonarQubea [28]. SonarQube on staattinen lähdekoodin analysointityökalu, joka tuottaa sille syötetystä koodista monipuolista laatudataa.

3.1 Case: Vaadin

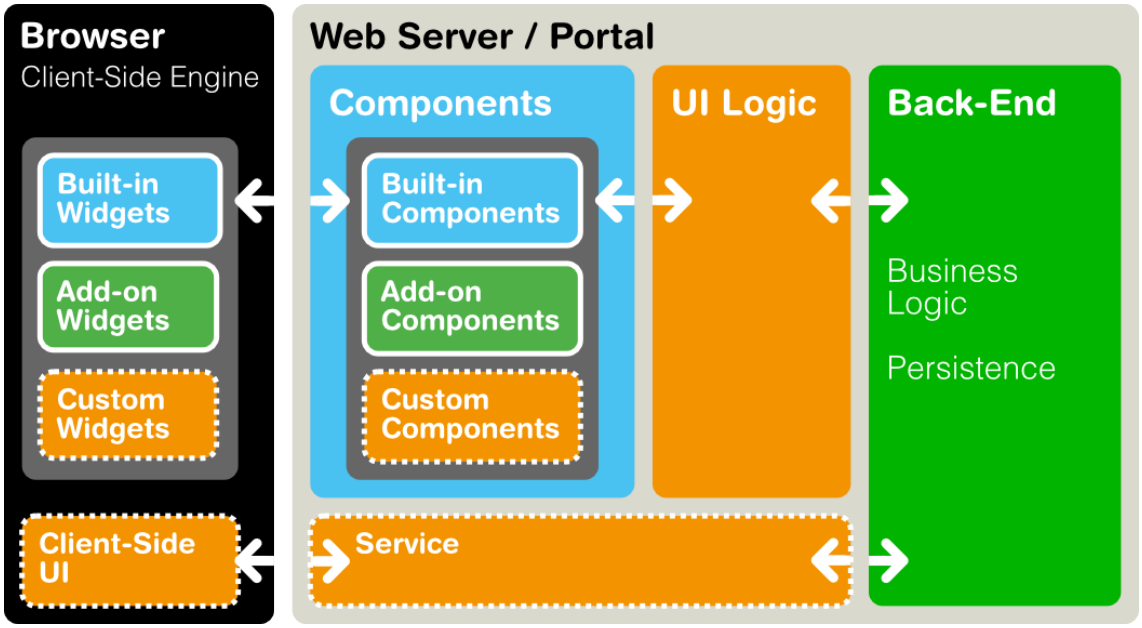
Vaadin Oy on vuonna 2000 perustettu suomalainen ohjelmistoyritys [41]. 150 työntekijää työllistävän Vaadinin päätuotteena on Vaadin Framework-web-viitekehys. Yrityksellä on toimistot Suomen lisäksi myös Saksassa ja Yhdysvalloissa.

3.1.1 Vaadin Framework 7:n esittely

Vaadin Framework 7 on web-kehitystyökalu, jonka avulla web-applikaatioiden kehittäminen onnistuu Java-ohjelmointikieltä käyttäen [42]. Vaadin-viitekehys mahdollistaa web-applikaatioiden kehittämisen ilman perinteisiä web-teknologioita. Se tukee sekä palvelin- että asiakasohjelmapään ohjelmointia. Vaadin Framework 7 on kehitetty Java-ohjelmointikielellä.

Vaadin Framework 7:n palvelinpään ohjelmointi mahdollistaa web-applikaation käyttöliittymälogiikan ja toimintalogiikan ohjelmoinnin applikaation palvelinpäässä Javaa käyttäen [42]. Applikaatio koostuu palvelinpäässä suoritettavasta viitekehyksestä sekä internet-selaimessa ajettavasta asiakasohjelmamoottorista. Moottori pyörii selaimessa JavaScriptillä ja se hyödyntää *GWT*-työkalupakettia (engl. Google Web Toolkit) [15]. GWT mahdollistaa Javan kääntämisen JavaScriptiksi selaimessa. Asiakasohjelmamoottori piirtää käyttöliittymää ja ohjaa käyttäjän antamat komennot

palvelimelle käyttöliittymälogiikan käsiteltäväksi. Applikaation käyttöliittymälogiikka pyörii palvelimella Java Servlettinä. Viitekehys huolehtii lisäksi selaimen ja palvelimen välisistä *AJAX*-kutsuista (engl. asynchronous JavaScript and XML). Vaadinilla kehitettävää web-applikaatiota voidaan näin ollen pitää kevyenä asiakasohjelmana (engl. thin client), kun sekä käyttöliittymä- että toimintalogiikka suoritetaan palvelinpäässä ja selain toimii vain näkymänä käyttäjälle. Kuva 3.1 esittää Vaadin Frameworkilla luodun web-applikaation rakenteen.



Kuva 3.1 *Tyypillinen Vaadin Frameworkilla luodun web-applikaation arkkitehtuuri [42].*

Framework 7 tukee myös asiakasohjelmapään ohjelmointia mahdollistaen esimerkiksi uusien käyttöliittymäkomponenttien kehittämisen Javaa käyttäen [42]. Framework 7:n asiakasohjelmapää hyödyntää GWT:tä, joten sekä asiakasohjelma- että palvelinpään ohjelmointi onnistuvat näin ollen Javalla. Framework 7 tukee myös CSS:ää, jota voidaan hyödyntää esimerkiksi käyttöliittymäkomponenttien ulkoasun viimeistelyssä.

3.1.2 Vaadin Framework 7:n kehitystyö

Vaadin Oy soveltaa Vaadin Frameworkin kehityksessä ketteriä menetelmiä ja käyttää sen kehittämiseen Scrum-viitekehystä. Scrumia sovelletaan pitkälti viitekehyksen alkuperäisten sääntöjen mukaan. Sprintin pituudeksi on määritelty kaksi viikkoa ja Frameworkin parissa työskentelee kaksi tiimiä lomittain siten, että jokaisella viikolla jommankumman tiimin sprintti päättyy. Tämän työn toteuttamishetkellä Vaadin Frameworkin aikaisempia versioita on kehitetty jo useita vuosia ja Vaadin 6:tta seuraavan Vaadin 7:n lähdekoodin koko analysointijakson alussa on yli 87 000 riviä

ohjelmakoodia. [20]

Vaadinilla tunnistetaan teknisen velan käsite ja sen mukanaan tuomat mahdolliset ongelmat. Mahdollisia teknisen velan aiheuttajia tunnistetaan myös; esimerkiksi erilaisten käytössä olevien kehitysympäristöjen eri konfiguraatiot voivat aiheuttaa tyylieroja kirjoitettuun koodiin. Teknistä velkaa tiedetään olevan myös Framework 7:ssä. Framework 7 päivitykset keskittyvät kuitenkin ohjelmointi- ja loppukäyttäjälle näkyvien virheiden korjaamiseen, jolloin teknisen velan takaisin maksaminen ja koodin refaktorointi jäävät vähäisemmälle huomiolle. [20]

3.2 Case: Yle

Suomen Yleisradio perustettiin vuonna 1926 [49]. Yleisradion ohjelmiston tehtävänä oli tuolloin "*Kansansivistyksen edistäminen, hyödyllisten tiedonantojen toimittaminen ja viattoman ajanvietteen hankkiminen.*" Nykyään Ylen tavoitteena on toimia suomalaisen yhteiskunnan ja kulttuurin vahvistajana [52]. Yle Areenaa kehitetään Ylen teknologiapalveluiden puolella [40].

3.2.1 Yle Areenan esittely

Yle Areena on Suomen Yleisradion tarjoama, internetissä toimiva mediatoistopalvelu [51]. Yle Areena toimii suoratoistopalveluna ja se mahdollistaa Ylen televisio-ohjelmien katselun sekä radiokanavien kuuntelun internetissä. Suoratoistopalveluiden lisäksi Areenassa on saatavilla myös menneiden ohjelmien tallenteita.

Yle Areena on toteutettu HTML5-teknologialla [50], jolloin Areena toimii millä tahansa modernilla HTML5:tä tukevalla selaimella. Areena on toteutettu pääosin *JavaScript* -ohjelmointikielellä hyödyntäen *ECMAScript 6*:n ominaisuuksia [21]. Areenasta on toteutettu myös mobiiliapplikaatioita mobiililaitteille näiden natiiveilla ohjelmointikielillä.

3.2.2 Yle Areenan kehitystyö

Yle Areenaa kehitetään ketterien periaatteiden mukaan, hyvin joustavilla työmenetelmillä. Areenan kehitystiimi ei seuraa mitään yhtä tiettyä työprosessia tai viitekehystä, vaan ohjelmistokehitykseen on otettu vaikutteita useista moderneista ohjelmistokehitysmalleista, muun muassa DevOpsista, Leanista ja Kanbanista, joista viimeeksi mainittu toimii tiimin kehitystyön runkona. Tarkasti sääntöjen mukaan

noudatettaviksi tarkoitettuja työmenetelmiä on vähän, ja esimerkiksi Scrumissa sovellettavia sprinttejä kehitystiimi ei käytä. Areenan kehitystiimi on pitkälti itseohjautuva ja muokannut moderneista ohjelmistokehitysmalleista tarpeisiinsa sopivat työskentelymenetelmät. [21]

Yle Areenan kehitystiimissä tunnistetaan teknisen velan käsite ja sen mukana mahdollisesti tulevat ongelmat. Lähdekoodin laatuun ja siisteyteen kiinnitetään jossain määrin huomioita ja kehittäjät muun muassa käyvät läpi toistensa kirjoittamaa koodia ennen sen tuotantoon viemistä. Kehitystiimi on sopinut keskenään suullisesti joistain noudatettavista yhteisistä tyylisäännöistä. Lisäksi käytössä on ajoittain toistuva, lähdekoodiin kohdistuva ”siivouspäivä”, jolloin keskitytään lähdekoodin siistimiseen ja koodin refaktorointiin. Siivouspäivää varten ylläpidetään erikseen ”siivouspankkia”, jonka täyttyessä siivouspäivä toteutetaan. [21]

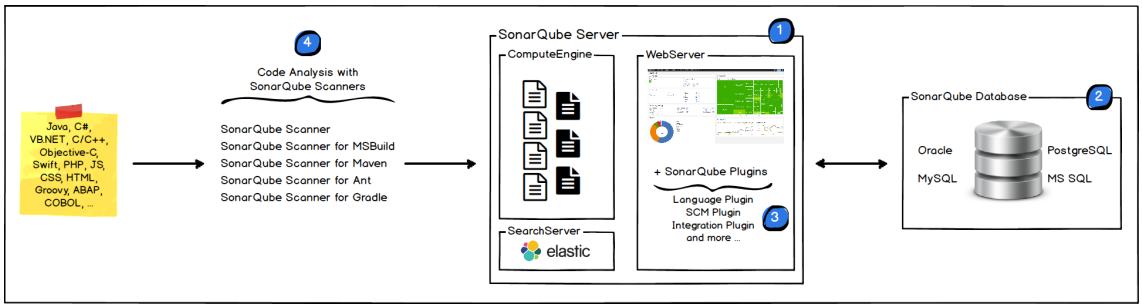
Tässä työssä käsitellään Yle Areenan uudelleen toteutettua versiota, joka tukee HTML5-teknologiaa. Analysoitavassa projektissa ei ollut siis projektin alkaessa vielä riviäkään ohjelmakoodia. Analysointijakson lopussa ohjelmakoodia oli kertynyt lopulta lähes 2 000 riviä.

3.3 SonarQube

SonarQube on avoimeen lähdekoodiin perustuva, staattinen lähdekoodin analysointi-työkalu [28]. SonarQubella voidaan muun muassa analysoida ohjelmiston lähdekoodin laatua ja mitata sen sisältämää teknistä velkaa. SonarQubelle syötetään analysoitavan ohjelmiston lähdekoodi, jonka perusteella SonarQube tuottaa raportin lähdekoodin laadusta. SonarQube voidaan liittää osaksi jatkuvan julkaisun putkea, jolloin analyysien suorittaminen voidaan automatisoida. SonarQube-analyysit voidaan tarvittaessa ajaa myös manuaalisesti. SonarQube tukee useita käyttöjärjestelmiä ja vaatii ajoympäristöltään Java-tuen. SonarQubella on mahdollista analysoida useita eri ohjelmointikieliä ja se on laajennettavissa sekä kaupallisilla että avoimeen lähdekoodiin perustuvilla liitännäisillä.

SonarQube koostuu neljästä pääkomponentista: palvelimesta, tietokannasta, SonarQuben liitännäisistä sekä SonarQube-skannerista [30]. Palvelin ylläpitää eri prosesseja; muun muassa web-palvelinta, jonka kautta voidaan tarkastella SonarQube-raportteja sekä hallitaan käynnissä olevaa SonarQube-instanssia. Tietokanta säilöö SonarQube-instanssin asetukset sekä analyyseissä tuotetut laaturaportit. SonarQuben liitännäisten avulla voidaan siihen lisätä uusia ominaisuuksia sekä hallinnoida muun muassa analysoitavia ohjelmointikieliä, mahdollisia versionhallintaintegraatioita ja SonarQube-instanssin hallintatyökaluja. SonarQube-skanneri suorittaa SonarQubelle syötetyn ohjelmakoodin analyysin. Kuva 3.2 esittää SonarQuben pää-

komponentit sekä komponenttien välisen kommunikaation.



Kuva 3.2 *SonarQuben arkkitehtuuri [30].*

SonarQube analysoi sille syötetyn lähdekoodin käyttäen hyväksi erityisiä koodisääntöjä [33]. Kutakin SonarQuben tukemaa ohjelmointikieltä kohden on omat, kyseiselle kielelle ominaiset sääntönsä. Säännöt voivat perustua esimerkiksi laajalti hyväksyttyihin hyviin ohjelmointikäytäntöihin. Säännöt pakataan niin kutsuttuihin laatuprofiileihin (eng. quality profile) ja samaa ohjelmointikieltä kohden voi olla useita eri laatuprofiileja, jotka voivat erota toisistaan esimerkiksi aktiivisten sääntöjen määrän osalta. Laatuprofiili sisältää tyypillisesti satoja yksittäisiä sääntöjä. Laatuprofiilit toimivat SonarQuben liitännäisinä, joten esimerkiksi uuden laatuprofiilin lisääminen SonarQubeen onnistuu asentamalla kyseisen profiilin liitännäinen. Joihinkin ohjelmointikieliin on myös mahdollista luoda omia liitännäisiä mukautettuine sääntöineen [29].

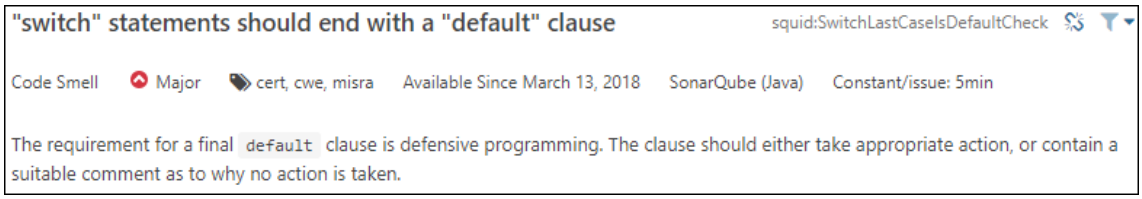
SonarQube luokittelee koodisäännöt kolmeen eri kategoriaan: luotettavuus-, ylläpidettävyys- ja haavoittuvuussääntöihin [31][37]. Luotettavuussäännöt liittyvät ohjelmakoodissa oleviin mahdollisiin ohjelmointivirheisiin, jotka voivat estää ohjelmaa toimimasta oikein tai saattavat estää ohjelman ajon kokonaan. Luotettavuussäännöt voivat nostaa esille myös koodisegmenttejä, jotka toimivat nykyhetkellä oikein, mutta ovat jollain tavalla ongelmallisia ja voivat täten aiheuttaa virheitä tulevaisuudessa. Haavoittuvuussäännöt puolestaan liittyvät analysoitavan ohjelmakoodin tietoturvaan, ja ne pyrkivät löytämään koodista mahdollisia tietoturvauhkia ja haavoittuvuuksia. Haavoittuvuussäännöt perustuvat pääosin vakiintuneisiin standardeihin [37]. Ylläpidettävyyssäännöt pyrkivät jäljittämään ylläpidettävyysongelmia ohjelmakoodista. Näitä kutsutaan myös koodihajuiksi, ja ne liittyvät suoraan ohjelmakoodin sisältämään tekniseen velkaan. Mitä enemmän ohjelmakoodista löytyy ylläpidettävyysongelmia, sitä suuremmaksi SonarQube tyypillisesti laskee koodin sisältämän teknisen velan määrän.

Absoluuttisen teknisen velan lisäksi SonarQube laskee lähdekoodista myös teknisen velan suhdeluvun (eng. technical debt ratio), joka kertoo ohjelmiston jatkokehityksen ja lähdekoodin korjaamiseen tarvittavan työmäärän välisen suhteen [34]. Suhdeluku

voidaan ymmärtää myös teknisen velan ja ohjelmiston koon välisenä suhteena.

SonarQuben sääntöstandardi on hyvin tiukka, ja SonarQube pyrkii takaamaan, ettei luotettavuus- ja ylläpidettävyyssäännöistä tulisi merkintöjä loppuraporttiin kohdista, jotka eivät oikeasti ole virheellisiä [37]. Haavoittuvuussääntöjen kohdalla tilanne on vaikeampi, sillä SonarQuben on esimerkiksi mahdotonta erottaa mikä on arkaluontoista dataa ja mikä ei. Tämä voi aiheuttaa haavoittuvuussääntöjen kohdalla suhteellisesti enemmän virheellisesti positiivisia tuloksia. SonarQube merkitsee tietoturvan kannalta epäilyttävät koodisegmentit, jotka on käytävä käyttäjän toimesta todentamassa mahdollisten virhemerkintöjen varalta.

Yksittäinen koodisääntö sisältää itse säännön määrittelyn lisäksi tietoja muun muassa säännön rikkomisesta aiheutuvan teknisen velan tai korjausvelan määrästä, vakavuusluokituksen, tyypin, kuvauksen sekä esimerkit sääntöä rikkovasta ja sääntöä noudattavasta koodista [36]. Teknisen velan ja korjausvelan määrä määritellään jokaisessa säännössä aika-arviona, joka kuluu kyseistä sääntöä rikkovan koodisegmentin korjaamiseen. Koko lähdekoodin sisältämä tekninen velka ja virheistä sekä haavoittuvuuksista erikseen laskettava korjausvelka muodostuvat näin ollen sääntörikkomusten summista. Kuvassa 3.3 on esimerkki yksittäisestä koodisäännöstä, jossa esimerkiksi säännön rikkomisesta aiheutuva velka on arvioitu kohdassa ”*Constant/issue*”.



Kuva 3.3 *Sääntö SonarQuben Java-laatuprofiilista.*

Koodisääntöjen sisältämillä vakavuusluokituksilla on viisi erilaista astetta [33]. Asteet määrittävät, kuinka vakava koodista löydetty ongelma on, ja ne vaihtelevat merkittävimmän, välittömästi korjausta vaativan ongelman (*blocker*) ja tiedotusluontoisen huomion (*info*) välillä. Jokaiselle löydetylle ongelmalle määritellään myös tila, joka on oletusarvoltaan uusilla ongelmilla ”avoin” [32]. Tilaa voi vaihtaa sen mukaan, miten ongelman korjausprosessi etenee. Löydetyille virheille voidaan määritellä myös vastuuhenkilöitä, jotka työskentelevät kyseisten ongelmien parissa. Näin SonarQubessa voidaan ylläpitää myös virheiden korjausprosessia.

Virheiden määrän ja teknisen velan lisäksi SonarQube mittaa lähdekoodista myös lukuisia muita arvoja, muun muassa koodin kompleksisuutta, kommenttirivien määrää ja koodiduplikaatteja [34]. SonarQube laskee joillekin mitattaville arvoille arvosanoja. Arvosanat kertovat käyttäjälle, mikä on mitatun määreen yleistila. Arvosana

voi saada arvon väliltä A-E, A:n ollessa paras mahdollinen. Esimerkiksi lähdekoodi saa oletusasetuksilla turvallisuusarvosanan ”A”, mikäli koodista ei löydy yhtään haavoittuvuusongelmaa.

Analysoitaville projekteille on mahdollista määritellä myös niin sanottuja laatuportteja (eng. quality gate) [35]. Laatuporttien avulla käyttäjä saa nopeasti käsityksen projektin tilasta. Käyttäjä voisi määritellä laatuportin esimerkiksi kuvaamaan ohjelman julkaisukelpoisuutta. Laatuporttiin voidaan määritellä esimerkiksi tärkeimmille mitattaville arvoille kynnysarvot, jotka analysoitavan lähdekoodin on ylitettävä. Hylätty laatuportti kertoo käyttäjälle jonkin arvon alittaneen määritellyn kynnysarvon ja paljastaa näin ohjelmiston mahdollisia puutteita tai vikoja.

SonarQuben yksittäinen laaturaportti voidaan mieltää tilannekatsauksena (eng. snapshot) analysoidun lähdekoodin tilasta tietyllä ajanhetkellä [31]. Yksi SonarQube-analyysi vastaa yhtä tilannekatsausta koodin tilasta, ja jokainen suoritettu analyysi tuottaa aina uuden tilannekatsauksen. Analysoitavasta ohjelmistosta on mahdollista luoda historiatietoja SonarQubeen analysoimalla ohjelmiston vanhoja versioita peräkkäin aikajärjestyksessä vanhimmasta versioista alkaen. Useista peräkkäisistä analyyseistä luodut tilannekuvat mahdollistavat trendi-tyyppisen tiedon muodostumisen ja ohjelmiston laadun kehittymisen seurannan.

3.4 Analyysien toteutus

Analyysit toteutettiin hakemalla sekä Vaadin Framework 7:n että Yle Areenan versionhallintajärjestelmistä peräkkäiset versionhallintajärjestelmän päivitykset tietyltä aikaväliltä. SonarQube-analyysit ajettiin näille haetuille päivityksille kronologisessa järjestyksessä alkaen vanhimmasta uusimpaan. Molemmat projektit käyttävät *Git*-versionhallintajärjestelmää. Lähdekoodien analyysit suoritettiin paikallisesti Windows-ympäristössä. Analyyseissä käytettiin SonarQuben versiota 5.6.7 ja analyysit ajettiin SonarQube-skannerin versiolla 3.2.0. Kielikohtaisina analysaattoreina olivat käytössä Javassa *SonarJava* versio 3.13.1 ja JavaScriptissä *SonarJS* 2.11.

Analyysit toteutettiin SonarQube-skannerilla, jota käytetään komentoriviltä. Analyysityön helpottamiseksi kirjoitettiin tämän työn kontekstiin riittävä Python-skripti, joka kykenee ajamaan SonarQube-skannerin uudelleen edellisen suorituksen päätyttyä. Skripti hakee sille syötetyn tekstitiedoston tietojen perusteella versionhallinnasta analysoitavan ohjelmistoversion, ajaa kyseiselle versiolle SonarQube-analyysin ja siirtyy tämän jälkeen seuraavan version käsittelyyn. Skriptin käytöllä vältyttiin jokaisen analyysin ajamisesta erikseen käsin ja se mahdollisti näin satojen analyysien ajamisen peräkkäin lyhyessä ajassa. Skripti on sisällytetty työhön liitteenä A.

3.4.1 Vaadin Framework 7:n analysointi

Vaadinista analysoitiin Vaadin Framework 7 [12]. Vaadin Framework 7 on analysoitu versiosta 7.0.0 versioon 7.7.13 saakka. Vaadinista on analysoitu jokainen niin kutsuttu *minor*- ja *maintenance*-päivitys [43][20]. Minor-päivitykset ovat tyypillisesti ylläpitopäivityksiä suurempia, ja niissä Vaadin Framework 7:ään tuodaan usein uusia ominaisuuksia. Ylläpitopäivitykset keskittyvät puolestaan Framework 7 ylläpitämiseen ja ohjelmasta mahdollisesti löytyvien virheiden korjaamiseen. Versiosta 7.0.0 versioon 7.7.13 on kaikkiaan 85 versiojulkaisua, jotka kaikki saatiin analysoitua SonarQube-skannerilla.

Vaadin Framework 7 analysoitiin SonarQuben oletusasetuksilla ja oletus-Javasäännöillä. Analyysi kohdistettiin Vaadinin lähdekoodissa hakemistoihin ”client”, ”server” ja ”shared”, sillä nämä ovat SonarQube-analyysin kannalta olennaisimmat hakemistot [20]. Vaadinin Git-versiohallinnan lokissa muutamien peräkkäisten versioiden kommit-aikaleimat olivat samat. Näissä tapauksissa myöhemmän version aikaleimaa muutettiin SonarQuben tietokantaan päivämäärän osalta yhtä päivää myöhemmäksi, sillä SonarQube ei salli analyysin aikaleiman olevan sama tai aikaisempi kuin viimeisimmän jo SonarQubessa olevan onnistuneen analyysin aikaleima. Muutokset tehtiin siten, että juokseva versionumerointi pysyi järjestykseltään muuttumattomana, ja kaikki suunnitellut julkaisuversioiden analyysit saatiin näin toteutettua oikeassa järjestyksessä. Tulosten tarkkuuden kannalta tällä muokkauksella ei ole merkitystä, koska muokkaus koski ainoastaan versioiden aikaleimoja. Ohjelmiston käännöstiedostoja ei tässä yhteydessä ole analysoitu.

3.4.2 Yle Areenan analysointi

Yle Areenasta analysoitiin jokainen versionhallintaan tehty päivitys noin viiden kuukauden ajalta. Analyysi kohdistettiin hakemistoon ”app” ja analyysissä käytettiin SonarQuben oletusasetuksia ja oletus-Javascriptsääntöjä.

SonarQube ohitti joitain analyysejä, eikä näin ollen suorittanut niitä, sillä Areenan Git-lokin mukaan versionhallinnan päivitykset eivät ole täydellisessä aikajärjestyksessä. Tämä voi johtua esimerkiksi kehitystiimissä tapahtuvasta rinnakkaisesta työskentelystä tai yksinkertaisesti päätelaitteiden erilaisista kellonajoista [11]. Areena-projektin osalta aikaleimoja ei muokattu SonarQube-analyysejä varten, sillä ohitettuja analyysejä oli yksittäisten tapausten sijaan lähes 200 kappaletta eikä Areena-projektilla ollut juoksevaa versionumerointia, josta versioiden kronologisen järjestyksen olisi voinut päätellä, kuten Vaadinilla. Ohitettujen analyysien lukumäärä ei kuitenkaan vaikuta analyysin tarkkuuteen, sillä ohjelmakoodia ei ohitettujen

analyysien vuoksi jäänyt analysoimatta. Joissain tapauksissa uusi tai muokattu ohjelmakoodi saattaa näkyä SonarQubessa muutaman tunnin tai -päivän viiveellä. Lisäksi onnistuneita ajoja kertyi riittävästi myös trendi-tyyppistä tarkastelua varten. Areena-projektin 554 päivityksestä SonarQube onnistui analysoimaan 363 kappaletta, joten ohitettuja analyysejä kertyi 191 kappaletta.

4. TULOKSET

Tässä luvussa käydään läpi keskeisimmät SonarQube-analyysien tuottamat tulokset. Tuloksissa käsitellään ohjelmistojen kokoa, teknistä velkaa ja sen lähteitä sekä lähdekoodin mahdollisesti sisältämiä ohjelmointivirheitä ja haavoittuvuuksia. Tulosten kuvaamiseen on osin käytetty SonarQuben tuottamia kuvia, joissa jokainen ympyrä kuvaa kyseisellä aikaleimalla ajettua SonarQube-analyysiä.

Tulokset esittävät teknisen velan aikana, jonka arvioidaan kuluvan velan takaisin maksamiseen. Ohjelmointivirheiden ja haavoittuvuuksien määrää tarkastellaan korjausvelkana, joka niin ikään on arvioitu aika kyseisten ongelmien korjaamiseksi. Tuloksissa käydään läpi myös koodihajujen, ohjelmointivirheiden ja haavoittuvuuksien kappalemäärät. Ohjelmistojen kokoa käsitellään varsinaisten koodirivien määrän avulla.

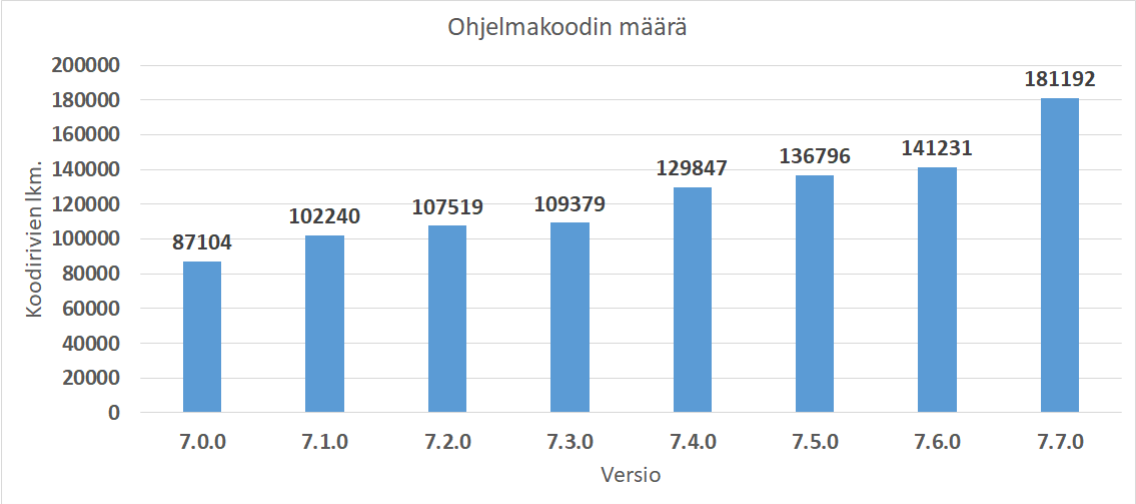
4.1 Vaadin Framework 7:n tulokset

Vaadin Framework 7:ää on analysoitu SonarQube-työkalulla versiosta 7.0.0 versioon 7.7.13 saakka. Git-lokin mukaan versio 7.0.0 on julkaistu 3.2.2013 ja versio 7.7.13 on julkaistu 2.1.2018. Kyseinen analysointijakso kattaa siis lähes viiden vuoden kehitysajan. Kyseisen jakson aikana versiojulkaisuja kertyi 85 kappaletta, joista jokainen on ajettu SonarQube-skannerin läpi.

Vaadin Framework 7 on analysoitavista projekteista selkeästi suurempi lähdekoodin määrällä mitattuna. Framework 7 sisältää analysointijakson alussa valmista koodia, joka on periytynyt aiemmista Vaadin Framework -versioista.

4.1.1 Framework 7:n koko

Vaadin Framework 7:n lähdekoodin koko analyysien aloitushetkellä on 87 104 riviä Java-koodia. Kuvassa 4.1 näkyy ohjelman lähdekoodin määrä kunkin minor-version julkaisuhetkellä. Kuvasta voimme huomata, että ohjelman koko kasvaa läpi analysointijakson, ja versiossa 7.7.0 lähdekoodin koko on kasvanut 181 192 riviin. Lähdekoodin koko kasvaa vielä versioon 7.7.13 mennessä 185 665 riviin, joten ohjelmakoodin määrä kasvaa siis 98 561 rivillä versioiden 7.0.0 ja 7.7.13 välillä.



Kuva 4.1 *Vaadin Framework 7: ohjelmakoodin määrän kehitys.*

Ohjelmakoodin määrä kasvaa eniten uusien minor-julkaisuversioiden kohdalla. Minor-versiot tuovat ohjelmaan yhteensä 86 592 riviä uutta koodia, joka on noin 88 %:n osuus kaikesta uudesta koodista. Kukin minor-päivitys kasvattaa lähdekoodin kokoa noin 1 000–39 000 rivillä. Taulukko 4.1 näyttää koodirivien määrien muutokset jokaisen minor-julkaisun osalta. Koodirivin määrän muutos on laskettu kussakin tapauksessa julkaistun minor-version ja edellisen minor-version viimeisen ylläpitopäivityksen välisenä erotuksena. Taulukosta voidaan havaita, että minor-päivitysten koot koodiriveinä mitattuna vaihtelevat paljon. Kolme minor-päivitystä erottuu selkeästi muita suurempina: 7.1, 7.4 ja 7.7. Näistä versioista eniten uutta koodia on versiossa 7.7: 39 394 riviä. Vähiten ohjelma kasvaa minor-versioiden osalta version 7.3 julkaisun yhteydessä, jolloin se kasvaa vain 1 008 rivillä.

Taulukko 4.1 *Koodirivien määrien muutokset kunkin minor-julkaisun yhteydessä.*

Versiojulkaisu	**Koodirivien määrän muutos (kpl)**
7.1.0	12 845
7.2.0	3 786
7.3.0	1 008
7.4.0	19 553
7.5.0	6 107
7.6.0	3 899
7.7.0	39 394
	Yht. 86 592

Ylläpitopäivitykset kasvattavat lähdekoodia maltillisemmin. Ylläpitopäivitykset tuovat ohjelmaan uutta koodia yhteensä 11 969 rivin verran. Tämä muodostaa noin 12 %:n osuuden kaikesta uuden lähdekoodin määrästä. Yksittäinen ylläpitopäivitys tuo ohjelmaan tyypillisesti kymmeniä tai satoja uusia koodirivejä. Yksittäisiä yli tuhannen koodirivin ylläpitopäivityksiä on muutama.

Minor-versioiden ylläpitopäivityksistä muodostuvat ylläpitojaksot kasvattavat kukin lähdekoodia noin 500–4 500 rivin verran. Taulukossa 4.2 on nähtävillä jokaisen minor-version ylläpitojakson aikana tapahtuneet muutokset koodin määrässä. Taulukosta huomaamme, että version 7.7 ylläpitopäivitykset kasvattavat koodia ylläpitopäivitysten osalta eniten, jolloin koodin määrä on kasvanut versiosta 7.7.0 versioon 7.7.13 tultaessa 4 473 rivillä. Vähiten lähdekoodin määrä kasvaa version 7.5 aikana: 536 rivin verran. Kyseisen version ylläpitojakson aikana lähdekoodin määrä myös vähenee kahdessa tapauksessa: versioissa 7.5.2 ja 7.5.5 yhteensä noin 200 rivin verran. Lähdekoodin määrä vähenee lisäksi myös versiossa 7.4.2 noin 100 rivin verran.

Taulukko 4.2 *Koodirivien määrien muutokset kunkin minor-version ylläpitojakson aikana.*

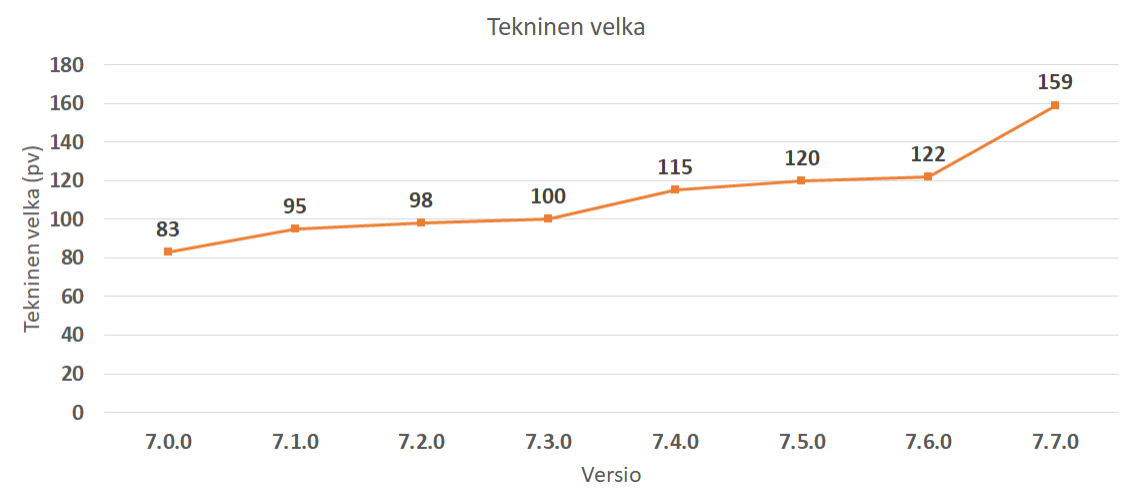
Version ylläpitojakso	**Koodirivien määrän muutos (kpl)**
7.0.0 - 7.0.7	2 291
7.1.0 - 7.1.15	1 493
7.2.0 - 7.2.7	852
7.3.0 - 7.3.10	915
7.4.0 - 7.4.8	842
7.5.0 - 7.5.10	536
7.6.0 - 7.6.8	567
7.7.0 - 7.7.13	4 473
	Yht. 11 969

Vaadin Framework 7 koodirivien määrä kasvaa läpi sen kehitysajan. Uutta koodia syntyy kaikkiaan noin 99 000 riviä viiden vuoden aikana. Minor-päivitysten mukana tullut koodi muodostaa selkeästi ylläpitopäivityksiä suuremman osuuden kaikesta uudesta ohjelmakoodista. Yksittäisistä versioista eniten uutta koodia on esitelty versioiden 7.1, 7.4 ja 7.7 julkaisuissa. Myös ylläpitopäivitykset kasvattavat lähdekoodin kokoa, mutta kuitenkin selkeästi minor-julkaisuja vähemmän. Kolmen versiopäivityksen yhteydessä tapahtuva koodin määrän väheneminen ei juurikaan näy kokonaiskuvassa, sillä koodin määrä vähenee yhteensä vain muutamilla sadoilla riveillä. Vaadin Framework 7:n lähdekoodin koko analysointijakson lopussa on noin 186 000 riviä.

4.1.2 Framework 7:n tekninen velka

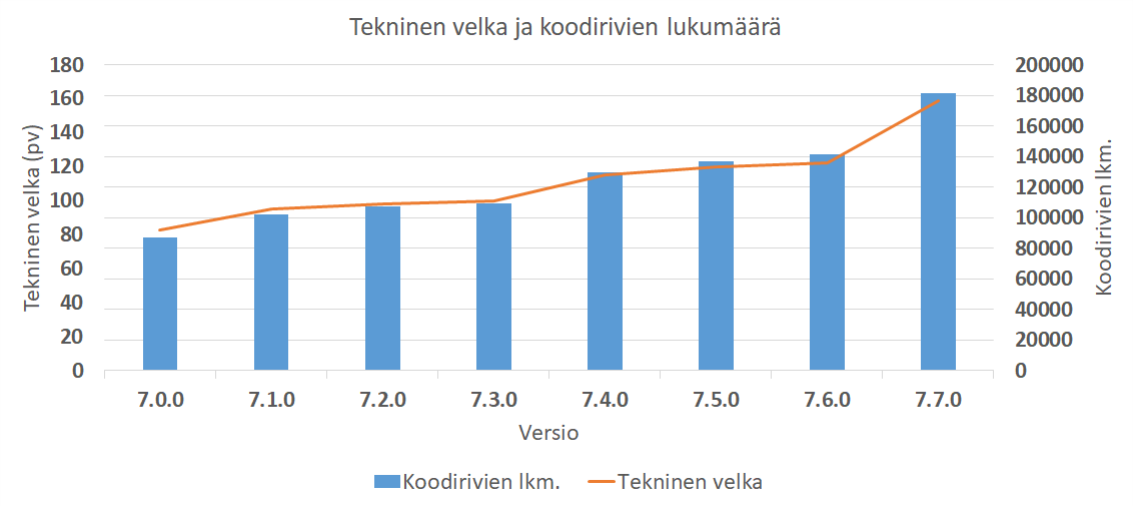
Vaadin Framework 7 sisältää teknistä velkaa ensimmäisistä versioista alkaen. Kuva 4.2 näyttää teknisen velan määrän kunkin minor-julkaisun kohdalla. Kuvasta huomaamme, että ensimmäinen analyysissä mukana oleva versio 7.0.0 sisältää teknistä velkaa 83 vuorokautta. Tekninen velka kasvaa läpi koko analysointijakson ja versiossa 7.7.0 tekninen velka on kasvanut 159 vuorokauteen. Tarkastelujaksossa viimeisenä

mukana olevassa versiossa 7.7.13 tekninen velka nousee vielä 163 vuorokauteen, joten teknisen velan määrä on kasvanut versioiden 7.0.0 ja 7.7.13 välillä siis 80 vuorokaudella.



Kuva 4.2 *Vaadin Framework 7: teknisen velan kehittyminen.*

Teknisen velan kehittyminen seurailee lähdekoodin määrän kehittymistä. Kuvassa 4.3 samaan kuvaajaan on piirretty sekä lähdekoodin koko koodiriveinä että sen sisältämä tekninen velka. Kuvasta voidaan havaita, että tekninen velka kasvaa karkeasti samassa suhteessa lähdekoodin määrän kanssa.



Kuva 4.3 *Vaadin Framework 7: teknisen velan ja ohjelmiston koon välinen korrelaatio.*

Lähdekoodin määrä kasvaa tyypillisesti isoin harppauksin uusien minor-päivitysten yhteydessä ja maltillisemmin ylläpitopäivitysten yhteydessä. Samanlainen kehitys on havaittavissa koodin sisältämän teknisen velan kohdalla. Tekninen velka kasvaa uusien minor-julkaisuiden yhteydessä pääsääntöisesti enemmän kuin ylläpitojaksojen aikana. Taulukossa 4.3 on eriteltynä kunkin minor-version yhteydessä tapahtuva

teknisen velan määrän muutos. Teknisen velan muutos on laskettu uuden minor-version ja edellisen minor-version viimeisen ylläpitopäivityksen sisältämien teknisen velan määrien välisenä erotuksena. Taulukosta huomaamme, että tekninen velkaa kasvaa selkeästi eniten versioiden 7.1, 7.4 ja 7.7 yhteydessä. Näissä kolmessa versiossa myös lähdekoodin määrä kasvoi eniten. Minor-versiot kasvattavat teknistä velkaa yhteensä 72 päivällä, joka on 90 % osuus kaikesta analysointijakson aikana syntyneestä teknisestä velasta.

Taulukko 4.3 *Teknisen velan määrien muutokset kunkin minor-julkaisun yhteydessä.*

Versiojulkaisu	**Teknisen velan määrän muutos (pv)**
7.1.0	10
7.2.0	2
7.3.0	1
7.4.0	14
7.5.0	6
7.6.0	2
7.7.0	37
	Yht. 72

Tekninen velka kasvaa vähän tai ei ollenkaan ylläpitopäivitysten yhteydessä. Taulukko 4.4 näyttää teknisen velan määrän muutoksen kunkin ylläpitojakson aikana. Taulukosta voidaan havaita kuinka tekninen velka myös vähenee version 7.4 ylläpitojakson aikana 1 päivällä. Version 7.7 ylläpitojakso kasvattaa teknistä velkaa 4 päivällä, joka on ylläpitojaksojen osalta eniten. Ylläpitojaksot kasvattavat teknistä velkaa yhteensä 8 päivällä, joka muodostaa 10 % osuuden kaikesta uudesta velasta.

Taulukko 4.4 *Teknisen velan määrien muutokset kunkin minor-version ylläpitojakson aikana.*

Version ylläpitojakso	**Teknisen velan määrän muutos (pv)**
7.0.0 - 7.0.7	2
7.1.0 - 7.1.15	1
7.2.0 - 7.2.7	1
7.3.0 - 7.3.10	1
7.4.0 - 7.4.8	-1
7.5.0 - 7.5.10	0
7.6.0 - 7.6.8	0
7.7.0 - 7.7.13	4
	Yht. 8

Vaadin Framework 7:n teknisen velan suhdeluku pysyy melko tasaisena läpi analysointijakson. Suhdeluku on 1,5 % versioissa 7.0.0 - 7.3.10. Versiossa 7.4.0 suhdeluku tippuu 1,4 % säilyttäen tämän versioon 7.7.13 saakka. Melko tasaisena pysyvä suhdeluku selittää myös aiemmin kuvassa 4.3 havaitun koodin määrän ja teknisen velan

välisen suhteen. Mikäli teknisen velan suhdeluku vaihtelisi enemmän analysointijakson aikana, näkyisi se myös kuvassa, eivätkä teknisen velan ja koodin määrän kuvaajat seuraisi samassa suhteessa toisiaan.

Teknistä velkaa aiheuttavia koodihajuja on versiossa 7.0.0 yhteensä 5 135 kappaletta. Versioon 7.7.13 mennessä koodihajuja on kertynyt 10 146 kappaletta, joten niiden määrä on lähes kaksinkertaistunut analysointijakson aikana. Taulukko 4.5 esittää lähdekoodissa esiintyvien koodihajujen viisi yleisintä lähdettä. Huomaamme, että yleisin koodihajujen lähde koskee <> -operaattorin käyttöä Java-koodissa. Näitä esiintymiä löytyi yhteensä 1 180 kappaletta, joka on noin 12 %:n osuus kaikista koodihajuista. Viisi yleisintä koodihajujen lähdettä muodostavat yhdessä noin 36 %:n osuuden kaikista SonarQuben löytämistä koodihajuista.

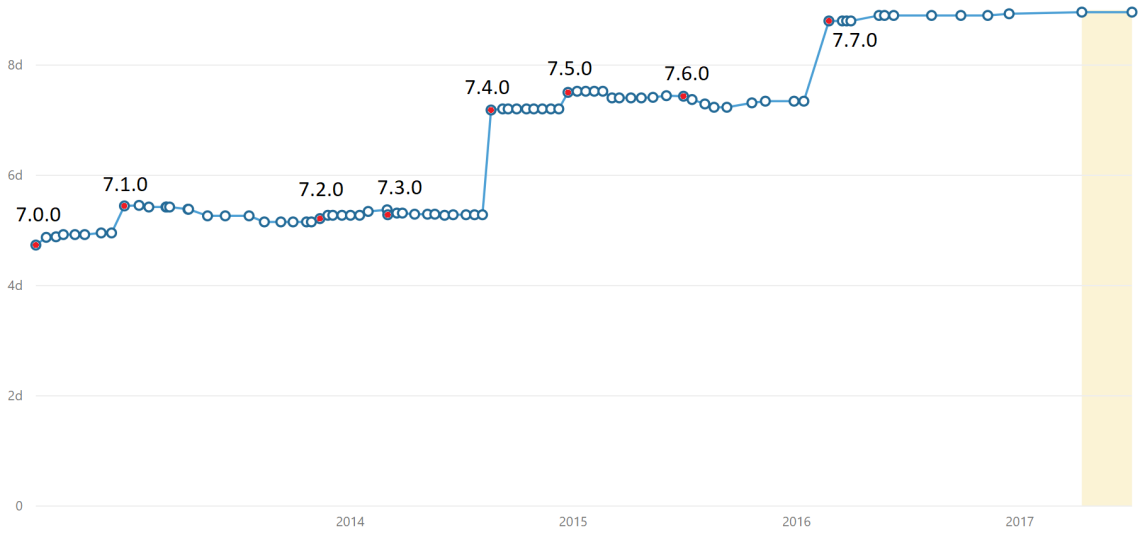
Taulukko 4.5 *Vaadin Framework 7: viisi yleisintä SonarQuben löytämää koodihajua ja koodihajujen esiintymismäärät versiossa 7.7.13.*

Koodihaju	**Koodihajujen määrä (kpl)**
Timanttioperaattoria <> tulisi käyttää olioiden rakentajakutsuissa.	1 180
Vanhentuneeksi merkattu koodi tulisi poistaa.	687
Metodien nimeämiskäytäntö tulisi olla yhtenäinen läpi koodin.	628
String-literaalin käyttäminen useassa paikassa tulisi korvata vakiomuuttujalla.	615
Luokkamuuttujien tulisi olla luokan yksityisellä puolella julkisen sijaan.	537
	Yht. 3 647

Vaadin Framework 7 sisältää analysointijakson alusta lähtien teknistä velkaa, joka on kertynyt luultavasti aiempia Frameworkin versioita kehitettäessä. Tekninen velka kasvaa läpi Framework 7 analysointijakson, ja jakson lopussa se on kasvanut 83 päivästä 163 päivään. Tekninen velka kasvaa eniten minor-päivitysten yhteydessä, jolloin esitellään myös tyypillisesti suuri määrä uutta lähdekoodia. Minor-päivitysten yhteydessä kasvava tekninen velka muodostaa 90 % osuuden kaikesta analysointijakson aikana syntyvästä uudesta velasta. Teknisen velan ja lähdekoodin koon välistä suhdetta kuvaava suhdeluku pysyy kuitenkin melko tasaisena läpi analysointijakson, laskien arvosta 1,5 %:a arvoon 1,4 %:a versiossa 7.4.0. SonarQuben Framework 7:lle antama ylläpitoarvosana on näin ollen paras mahdollinen ”A”.

4.1.3 Framework 7:n ohjelmointivirheet ja haavoittuvuudet

SonarQuben havaitsemien ohjelmointivirheiden määrän trendi on niin ikään nouseva läpi analysointijakson. Virheitä kuitenkin myös korjataan jonkun verran kehi-



Kuva 4.4 *Vaadin Framework 7: SonarQuben laskeman ohjelmointivirheiden korjausvelan kehittyminen.*

tystyön edetessä. SonarQubesta otettu kuva 4.4 näyttää ohjelmointivirheiden määrästä johtuvan korjausvelan kehityksen. Kuvaan on muokattu näkymään minor-versiopäivitykset. Kuvasta voidaan havaita, kuinka virheiden korjausvelka paikoin laskee kehitystyön edetessä. Korjausvelan trendi on kuitenkin nouseva läpi analysointijakson. Versiossa 7.0.0 korjausvelka oli arvioitu 4 päiväksi ja 7 tunniksi ja versiossa 7.7.13 korjausvelka oli noussut 9 päivään ja 2 tuntiin. Lukumäärällisesti ohjelmointivirheitä on analysointijakson alussa 179 kappaletta ja jakson lopussa 382 kappaletta. Ohjelmointivirheitä on tullut siis 203 kappaletta lisää jakson aikana.

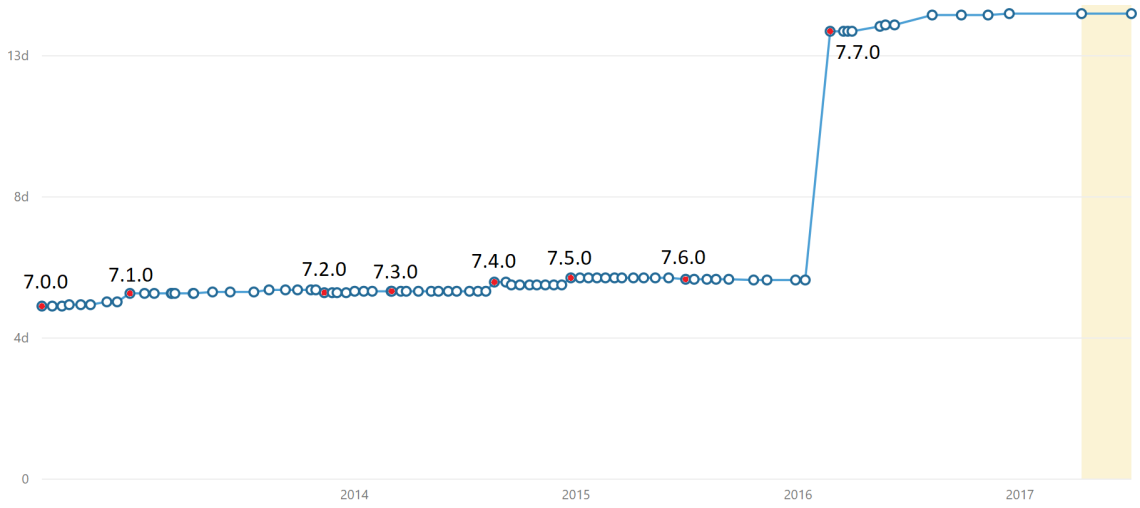
Taulukossa 4.6 esitellään SonarQuben löytämien ohjelmointivirheiden viisi yleisintä syytä. Kyseiset virheet ovat Framework 7 versiossa 7.7.13. Eniten rikottu sääntö koskee liukulukujen välistä vertailua yhtäsuuruusoperaattoreilla, mitä esiintyy lähdekoodissa 61 kertaa. Tämä on noin 16 %:a kaikista virheistä. Viisi eniten löydettyä sääntörikkomusta aiheuttavat yhteensä 209 kappaletta kaikista rikkomuksista, joka on noin 55 %:n osuus ohjelmointivirheistä versiossa 7.7.13.

Vaadin Framework 7:stä löydetyt haavoittuvuuksien määrät pysyvät pitkään melko tasaisena, kunnes versiossa 7.7.0 ne lisääntyvät merkittävästi. SonarQubesta otettu kuva 4.5 näyttää haavoittuvuuksien aiheuttaman korjausvelan kehittymisen. Kuvaan on muokattu näkymään minor-versiopäivitykset. Versiossa 7.0.0 haavoittuvuuksien korjausvelka on 5 päivää, josta se nousee 5 päivään ja 7 tuntiin versioon 7.6.8 tultaessa. Päivitettäessä versiosta 7.6.8 versioon 7.7.0 haavoittuvuuksien korjausvelka kasvaa 13 päivään. Versioon 7.7.13 tultaessa korjausvelka nousee vielä 14 päivään. Lukumäärällisesti haavoittuvuuksia on analysointijakson alussa 124 kappaletta ja jakson lopussa 342 kappaletta. Haavoittuvuudet kasvavat siis 218 kappaleella koko

Taulukko 4.6 *Vaadin Framework 7: viisi yleisintä SonarQuben löytämää ohjelmointivirhettä ja virheiden esiintymismäärät versiossa 7.7.13.*

Ohjelmointivirhe	**Virheiden määrä (kpl)**
Liukulukuja ei tulisi vertailla yhtäsuuruusoperaattoreilla.	61
Sarjallistettavan luokan attribuuttien tulisi myös olla sarjallistettuja.	45
Ehtolausekkeilla tulisi olla mahdollisuus evaluoitua sekä todeksi että epätodeksi.	41
Olioita ei tulisi luoda, mikäli sitä ei aiota käyttää.	37
Sijoituslauseita tulisi välttää lausekkeiden paikalla. Esimerkiksi: doSomething(i=15);	25
	Yht. 209

analysointijakson aikana. Ohjelmistopäivitys versiosta 7.6.8 versioon 7.7.0 kasvattaa yksinään haavoittuvuuksien lukumäärää 183 kappaleella.



Kuva 4.5 *Vaadin Framework 7: SonarQuben laskeman haavoittuvuuksien korjausvelan kehittyminen.*

Taulukko 4.7 esittää Framework 7:stä löytyneet viisi yleisintä haavoittuvuutta versiossa 7.7.13. Ylivoimaisesti eniten huomautuksia löytyy geneeristen poikkeusten käytöstä ohjelmakoodissa. Näitä huomautuksia on versiossa 7.7.13 yhteensä 247 kappaletta, joka muodostaa noin 72 %:n osuuden kaikista haavoittuvuuksista. Viisi yleisintä haavoittuvuutta aiheuttavat yhteensä 336 kappaletta kaikista haavoittuvuuksista ja ne muodostavat täten noin 98 %:n osuuden kaikista SonarQuben löytämistä haavoittuvuuksista.

SonarQube tunnistaa lähdekoodista virheitä ja haavoittuvuuksia analysointijakson alusta lähtien. Ohjelmointivirheiden muodostaman korjausvelan trendi on kasvava

Taulukko 4.7 *Vaadin Framework 7: viisi yleisintä SonarQuben löytämää haavoittuvuutta ja niiden esiintymismäärät versiossa 7.7.13.*

Haavoittuvuus	**Haavoittuvuuksien määrä (kpl)**
Geneerisiä poikkeuksia ei tulisi heittää.	247
Julkiset ja staattiset attribuutit tulisi merkitä vakioiksi.	35
Throwable.printStackTrace(...) ei tulisi kutsua.	29
Throwable- ja Error-yliluokkia ei tulisi napata sellaisinaan poikkeuskäsittelyssä.	19
Pääsytietoja ei tulisi kova-koodata lähdekoodiin.	6
	Yht. 336

läpi projektin, vaikka paikoin ylläpitopäivitysten yhteydessä havaitaan myös korjausvelan hienoista laskua. Yleisin lähdekoodissa esiintyvä ohjelmointivirhe on liukuluvuille suoritettava vertailu yhtäsuuruus-operaattorilla, mikä muodostaa kaikkiaan noin 16 %:n osuuden kaikista virheistä. Haavoittuvuuksien muodostama korjausvelka pysyy pitkään melko tasaisena versioon 7.7.0 saakka, jossa sen määrä kasvaa äkillisesti noin 8 päivällä. SonarQuben eniten tunnistama haavoittuvuus on geneeristen poikkeuksien käyttäminen erikoistettujen, itse määriteltyjen poikkeuksien sijaan. Näitä haavoittuvuuksia on kaikkiaan 247 kappaletta, joka muodostaa noin 72 % osuuden kaikista haavoittuvuuksista.

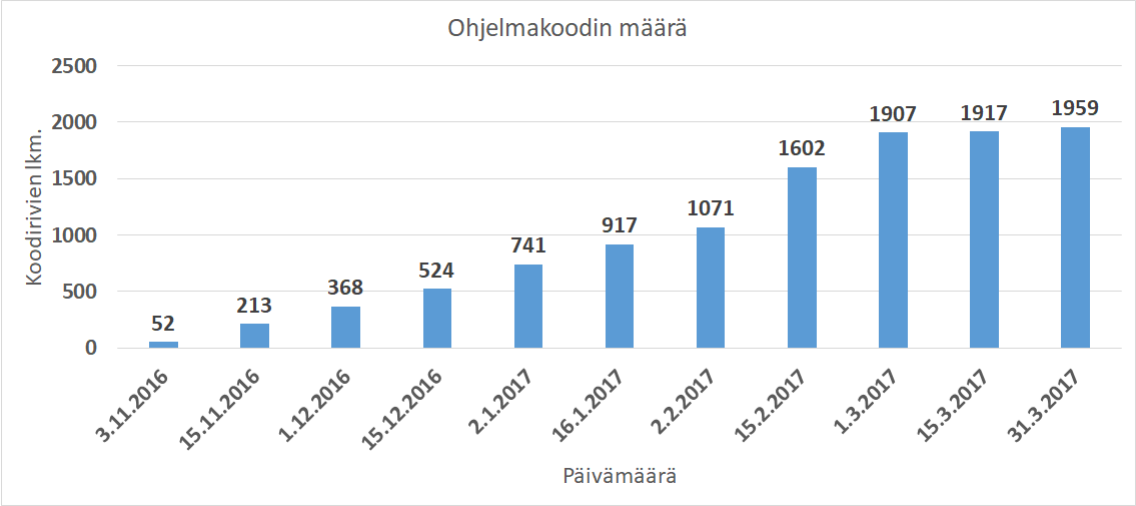
4.2 Yle Areenan tulokset

Yle Areenasta on analysoitu uudelleen toteutettua sovellusta, joka on toteutettu tukemaan HTML5-teknologiaa. Sovellusta alettiin kehittää talvella 2016-2017. Ensimmäiset versionhallintaan päivitetyt lähdekoodirivit on päivätty 3.11.2016 ja viimeisen analyysissä mukana olevan version päivämäärä on 31.3.2017. Analysointijakso kattaa siis noin viiden kuukauden kehitysjakson. Onnistuneita SonarQube-analyysejä kertyi tälle ajalle 363 kappaletta.

Yle Areenan kehitystyössä ei ole samanlaisia kiintopisteitä kuin Vaadin Framework 7:n julkaisuversiot ovat, joten Areenan kehittymistä tarkastellaan esimerkiksi monissa kuvaajissa noin kahden viikon välein. Tuloksia tarkastellessa mukana ovat versionhallinnasta saatavilla olevat ensimmäinen ja viimeinen versio sekä näiden väliltä versioita analysointijakson jokaisen kuun 1. ja 15. päivältä. Mikäli versionhallinnassa ei ole päivitystä kuun 1. tai 15. päivänä, käytetään seuraavan päivän päivitystä. Mikäli samana päivänä on tehty useita päivityksiä versionhallintaan, valitaan näistä kellonajan mukaan myöhäisimpänä ajankohtana tehty päivitys mukaan tarkasteluun.

4.2.1 Areenan koko

Analysoidun Areenan kehitys alkoi teknisen toteutuksen osalta tyhjästä, joten pohjalla ei ollut vanhaa lähdekoodia. Versionhallintaan ensimmäiset lähdekoodirivit syntyivät 3.11.2016 ja kyseisen päivän lopussa lähdekoodia oli kertynyt 52 riviä. Analysointijakson loppuun mennessä lähdekoodi oli kasvanut 1 959 riviin. Kuva 4.6 näyttää lähdekoodin määrän kehittymisen. Kuvasta voidaan havaita, että ohjelma kasvaa melko tasaisesti läpi analysointijakson, joskin päivämäärien 2.2.–15.2.2017 aikana ohjelma kasvaa hieman keskimääräistä enemmän ja maaliskuun aikana lähdekoodin määrän kasvu puolestaan hidastuu.



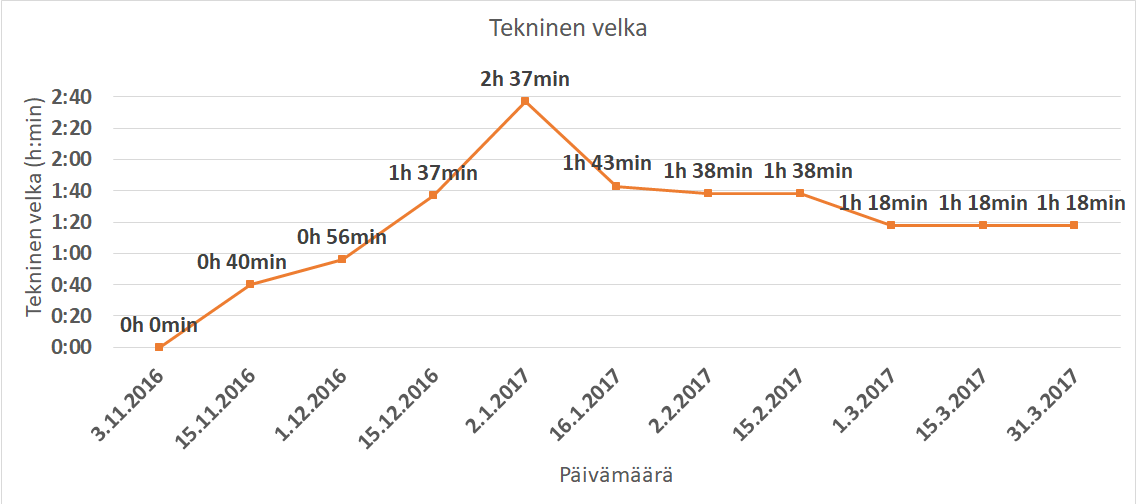
Kuva 4.6 *Yle Areena: lähdekoodin määrä noin kahden viikon välein tarkasteltuna.*

Yle Areena on näin ollen Vaadin Framework 7:ää hyvin paljon pienempi ohjelmisto lähdekoodin määrällä mitattuna. Myös käytettävissä oleva analysointijakso oli vain 5 kuukauden mittainen Framework 7:n noin 5 vuoden sijaan.

4.2.2 Areenan tekninen velka

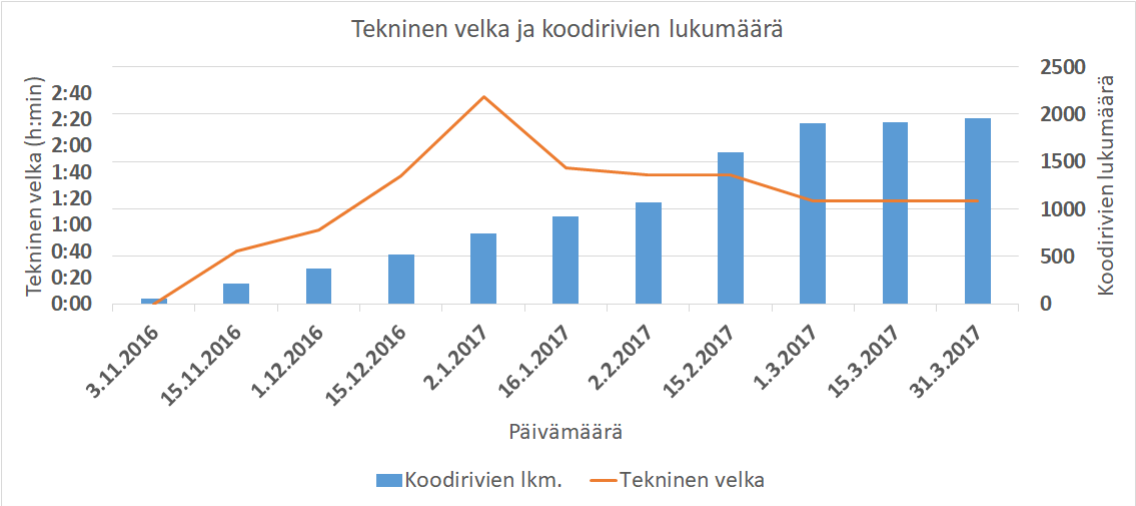
Yle Areena ei sisällä analysointijakson alussa riviäkään lähdekoodia, joten se ei myöskään sisällä teknistä velkaa. Kehitystyön edetessä lähdekoodiin alkaa kuitenkin syntyä teknistä velkaa. Kuva 4.7 näyttää teknisen velan määrät lähdekoodissa noin kahden viikon välein tarkasteltuna. Kuvasta huomaamme, että tekninen velka kasvaa kahden ensimmäisen kuukauden ajan tammikuun alkuun saakka, jolloin se saavuttaa korkeimman arvonsa, 2 tuntia ja 37 minuuttia. Tämän jälkeen tekninen velka alkaa laskea, kunnes se analysointijakson lopulla vakiintuu 1 tuntiin ja 18 minuuttiin.

Yle Areenan sisältämän teknisen velan kehitys seuraa kehitystyön alussa lähdekoodin määrän kehitystä, mutta poikkeaa tästä analysointijakson loppua kohti. Kuvassa



Kuva 4.7 *Yle Areena: lähdekoodin sisältämä tekninen velka.*

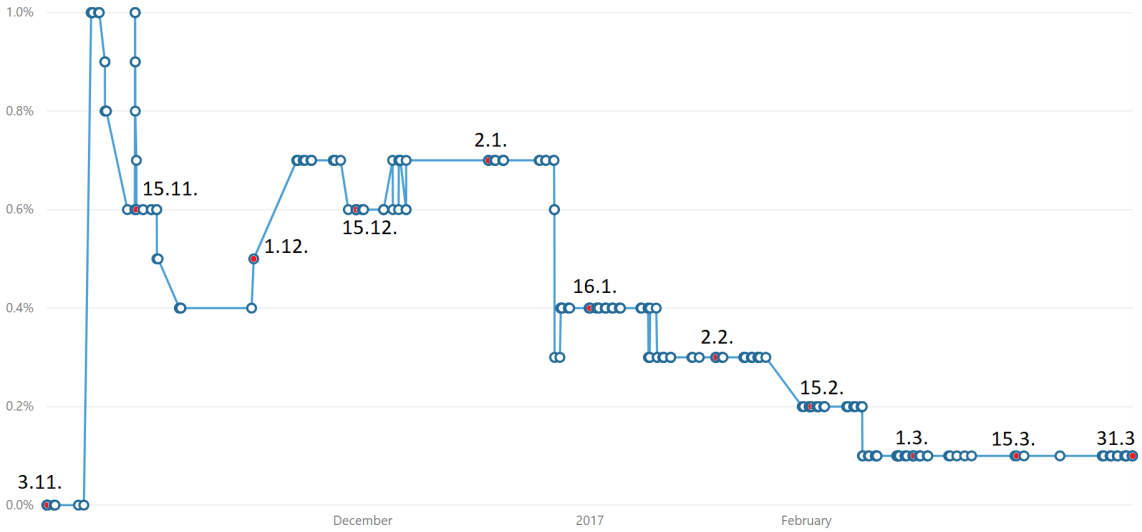
4.8 on esitetty sekä lähdekoodin määrän että teknisen velan kehitys. Kuvasta voidaan havaita, miten sekä koodin määrä että teknisen velan määrä kasvavat tammikuun alkuun saakka, jonka jälkeen tekninen velka alkaa laskea, mutta koodin määrä jatkaa kasvamista. Tekninen velka ei kasva enää tammikuun alun jälkeen jatkuvasta lähdekoodin määrän kasvusta huolimatta.



Kuva 4.8 *Yle Areena: teknisen velan ja lähdekoodin määrän kehittyminen.*

Teknisen velan suhdeluku saavuttaa korkeimmillaan arvon 1 % analysointijakson aikana. SonarQubesta otetusta kuvasta 4.9 on esitetty teknisen velan suhdeluvun kehittyminen. Kuvaan on muokattu näkymään muissa kuvaajissa käytetyt kahden viikon väliset tarkastelupisteet. Kuvasta huomaamme, kuinka teknisen velan suhdeluku saavuttaa korkeimman arvonsa 1 % heti projektin alussa, ensimmäisen kahden viikon pituisen tarkastelujakson aikana. Teknisen velan suhdeluku on itse tarkastelupäivänä vielä arvossa 1 %, mutta tippuu arvoon 0,6 % tarkastelupisteeksi valitussa,

vuorokauden viimeisessä versionhallinnan päivityksessä. Kuvaajasta huomaamme muitakin yhden vuorokauden sisällä tapahtuvia suhdeluvun muutoksia. Absoluuttisen teknisen velan ollessa korkeimmillaan 2.1., on teknisen velan suhdeluku 0,6 %. Absoluuttisen teknisen velan tavoin myös suhdeluvun trendi alkaa projektin alkuvaiheen jälkeen laskea ja päätyy lopulta arvoon 0,1 %. Teknisen velan suhdeluku on kaiken kaikkiaan hyvin pieni läpi analysointijakson.



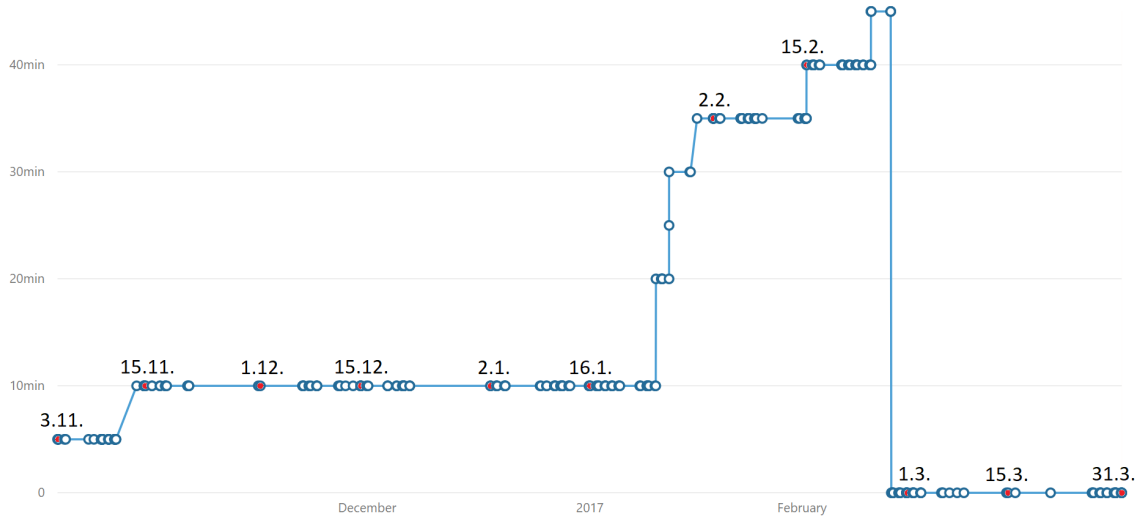
Kuva 4.9 *Yle Areena: teknisen velan suhdeluvun vaihtelu analysointijakson aikana.*

Teknistä velkaa aiheuttavia koodihajuja on Areenassa vain 7 kappaletta analysointijakson lopulla. Koodihajut koskevat esimerkiksi ”TODO”-merkintöjen suorittamista lähdekoodissa ja useiden ohjelmointilauseiden kirjoittamista peräkkäin samalle riville.

4.2.3 Areenan ohjelmointivirheet ja haavoittuvuudet

Areenassa on ohjelmointivirheitä analysointijakson lopussa vain 2 kappaletta. Nämä muodostavat yhteensä 15 minuuttia korjausvelkaa ohjelmistoon. Ohjelmointivirheiden määrä on läpi analysointijakson varsin alhainen, ja se vaihtelee 0-2 kappaleen välillä.

Haavoittuvuuksia Areenassa ei ole yhtään analysointijakson lopulla. Analysointijakson aikana haavoittuvuuksia on kuitenkin esiintynyt, ja SonarQubesta otettu kuva 4.10 esittää haavoittuvuuksien korjausvelan määrän projektin eri vaiheissa. Kuvaan on muokattu näkymään tarkastelupisteet noin kahden viikon välein. Kuvasta huomaamme, että haavoittuvuuksien korjausvelka saavuttaa suurimman arvonsa 27.2. Korjausvelan määrä on tällöin 45 minuuttia, ja haavoittuvuuksia on lähdekoodissa 9 kappaletta. Korjausvelka tippuu kuitenkin nollaan vielä saman vuorokauden kuluessa ja pysyy muuttumattomana analysointijakson loppuun asti.



Kuva 4.10 *Yle Areena: haavoittuvuuksien aiheuttaman korjausvelan määrä projektin eri vaiheissa.*

SonarQube säilyttää ainoastaan viimeisimpänä analysoidun lähdekoodin tietokannassaan, joten esimerkiksi lähdekoodin tilaa ohjelmiston historiassa ei pääse tarkastelemaan. Tämä aiheuttaa myös sen, ettei esimerkiksi ohjelmointivirheiden ja haavoittuvuuksien lähteitä pääse tarkastelemaan vanhoista versioista. Lähdekoodille on ajettava erillinen analyysi vanhasta versiosta, mikäli tätä halutaan tarkastella tarkemmin. Näin tässä työssä on toimittu haavoittuvuuksien lähteiden kohdalla, ja SonarQube-analyysi on ajettu erikseen 27.2. -versiolle, jolloin haavoittuvuuksien korjausvelka oli korkeimmillaan.

Erillinen SonarQube-analyysi osoittaa, että kaikki 9 haavoittuvuutta rikkovat samaa sääntöä, joka kieltää konsoli-tulosteiden käytön tuotantokoodissa. Konsoli-tulosteilla voisi olla mahdollista esimerkiksi vahingossa paljastaa käyttäjälle ei-haluttuja tietoja ohjelmistosta.

5. TULOSTEN POHDINTA

Työssä tutkitut ohjelmistoprojektit ovat hyvin erilaisia keskenään. Vaadin Framework 7 on kaupallinen, esimerkiksi ohjelmistokehittäjille suunnattu työkalu, jonka avulla on mahdollista toteuttaa web-applikaatioita Java-kielellä. Yle Areena on lähtökohtaisesti kuluttajille suunnattu verkkomediapalvelu, jolla voi toistaa Ylen toimittamia televisio- ja radio-ohjelmia. Molempien organisaatioiden ohjelmistokehitysmallit perustuvat ketterälle kehitykselle, mutta eroavat muutoin jonkin verran toisistaan. Vaadinilla ohjelmistokehityksessä sovelletaan Scrum-mallia pitkälti Scrumin sääntöjen mukaan. Ylellä kehitystyöhön on otettu vaikutteita useista eri ohjelmistokehitysmenetelmistä, joista Kanban toimii kehitysprosessin runkona. Muutoin työvirta on hyvin joustava, ja kehitystiimi on muovannut sen tarpeisiinsa sopivaksi useita eri ketteriä menetelmiä hyödyntäen.

Framework 7 on moninkerroin Areenaa suurempi ohjelmisto lähdekoodin määrällä mitattuna, ja se on ollut kehityksessä pidempään. Framework 7:n koko koodiriveinä mitattuna on analysointijakson lopussa noin 186 000 riviä, kun taas Areenan koko on noin 2 000 riviä. Pelkästään tässä työssä käytetyt analysointijaksot olivat Vaadinin kohdalla lähes viisi vuotta ja Areenan kohdalla noin viisi kuukautta. Framework 7 on seitsemäs versio Vaadin Framework -työkalusta ja Areenasta analysoitavana oli tyhjästä lähtenyt, uudelleen toteutettu sovellus. Areena ei näin ollen sisältänyt perinnekoodia Framework 7:n tavoin analyysien alussa. Ohjelmistojen välinen kokoero heijastelee myös tuloksiin, joissa Framework 7:stä tehdyt havainnot ovat tyypillisesti moninkertaisia Areenan vastaaviin arvoihin verrattuna.

Ohjelmistojen erilaisuus antaa kuitenkin myös mielenkiintoisen näkökulman tuloksille, kun tarkasteltavina ovat uusi, puhtaalta pöydältä lähtenyt projekti sekä jo pitkään kehityksessä ollut, vakiintunut projekti. On oletettavaakin, että ikääntynyt ohjelmisto ja täysin uusi projekti saavat erilaisia tuloksia esimerkiksi teknisen velan osalta. Vaikka kahden esimerkkitapauksen kautta tuloksia ei voikaan yleispätevöittää koskemaan kaikkia, vastaavanlaisia ohjelmistoprojekteja, on tuloksista kuitenkin saatavilla joitain viitteitä niihin. Tuloksista voidaan havaita oletettavia uuden projektin piirteitä sekä toisaalta tehdä huomioita vastaavasti jo vakiintuneen, iäkkäämmän projektin lähdekoodin laatuun liittyen.

Vaadin Framework 7 kasvaa analysointijakson aikana 99 000 lähdekoodirivillä. Analysointijakso on noin viiden vuoden pituinen. Tänä aikana ohjelmaan on tuotu seitsemän laajempaa päivitystä, jotka lisäävät käyttäjälle uusia ominaisuuksia. Nämä seitsemän minor-päivitystä kasvattavat ohjelmaa selkeästi eniten, sillä niiden mukana tullut uusi koodi muodostaa 88 % osuuden kaikesta uudesta koodista.

Framework 7:n sisältämä tekninen velka kasvaa niin ikään läpi analysointijakson. Framework 7 sisältää jo analysointijakson alussa lähdekoodia vanhojen Vaadin Framework -versioiden jäljiltä noin 87 000 riviä, joten myös teknistä velkaa on ensimmäisistä analysoiduista versioista lähtien. Framework 7:n absoluuttinen tekninen velka kasvaa läpi analysointijakson 80 vuorokaudella. Jakson alussa velkaa on 83 vuorokautta ja jakson lopussa velkaa on 163 vuorokautta. Tekninen velka kasvaa melko tasaisessa suhteessa koodin määrän kanssa: ison ohjelmistopäivityksen yhteydessä syntyy tyypillisesti myös enemmän teknistä velkaa kuin pienen päivityksen yhteydessä. Tekninen velka ei juurikaan vähene analysointijakson aikana yhtä poikkeusta lukuunottamatta. Teknisen velan kehittyminen voi heijastella kehitystiimin suhtautumista tekniseen velkaan: tekninen velka tunnistetaan, mutta sen poistamiseksi ei ole järjestelmällisiä toimenpiteitä.

Toisaalta teknisen velan suhdeluku on läpi analysointijakson varsin alhainen: 1,4–1,5 %. Teknisen velan määrä suhteessa ohjelman kokoon on siis vähäinen ja se pysyy melko tasaisena läpi jakson. Eli vaikka absoluuttinen tekninen velka kasvaa jatkuvasti, ei teknisen velan osuus kaikesta lähdekoodista kuitenkaan kasva, vaan pysyy tasaisena. Tämä on tietenkin positiivinen asia Framework 7:n kehitystyön kannalta.

Framework 7:n yleisin koodihaju on niin kutsutun timanttioperaattorin (<>) käyttämättä jättäminen olion rakentajakutsuissa. Timanttioperaattori esiteltiin Javaan versiossa 7 vuonna 2011. Vaadin Framework on ollut kehityksessä kuitenkin jo ennen tätä, joten on mahdollista, että osa kyseisen koodihajun lähteistä on syntynyt ennen vuotta 2011. Tämä olisi tällöin esimerkki niin kutsutun perinnekoodin aiheuttamasta teknisestä velasta, joka myös alan julkaisuiden mukaan on yksi merkittävä teknisen velan lähde [16].

SonarQuben löytämiä ohjelmointivirheitä on Framework 7:ssä 382 kappaletta. Vaikka virheiden aiheuttama korjausvelka usein laskeekin hieman ylläpitopäivitysten yhteydessä, tuovat suurimmat minor-päivitykset virheitä runsaasti lisää ja virheiden määrä kokonaisuudessaan on kasvava läpi projektin. Yleisin virhe on liukuluvuille suoritettava vertailu yhtäsuuruusoperaattorilla, jota tulisi välttää sen epätarkkuuden takia. Kyseinen vertailu saattaa tuottaa odottamattomia tuloksia jossain tapauksissa ja näin ollen ohjelma voi toimia väärin.

Teknisen velan hallinta muodostaa haasteen Vaadin Framework 7:ssä. Framework 7:n koodipohjan refaktorointi voi osoittautua haasteelliseksi koodipohjan koon vuoksi. Onnistuuko esimerkiksi timanttioperaattorin käyttöön liittyvien koodihajujen korjaaminen ilman sivuvaikutuksia vai joudutaanko koodia muokkamaan laajemmalti kyseisten koodihajujen korjaamiseksi? Timanttioperaattorin käyttöön liittyviä koodihajuja on noin 1 200 kappaletta, joten refaktoroinnin vaikutukset koodiin voivat olla laajat ja työläät. Sama koskee tietysti myös muitakin koodihajuja. Alan kirjallisuudessa on myös viitattu refaktorointiin liittyviin haasteisiin suurien ohjelmistojen yhteydessä [39][5]. Jossain tapauksissa runsaskin määrä teknistä velkaa lähdekoodissa on hyväksyttävä, sillä refaktoroinnista aiheutuvat kustannukset ja mahdolliset riskit voivat olla teknisen velan aiheuttamia ongelmia suuremmat [48]. Tekninen velka tulisikin ottaa huomioon ohjelmistokehityksessä alusta alkaen. Pitämällä refaktoroitavat osiot pieninä, saatetaan ennaltaehkäistä myöhemmin esiin nousevia ongelmia, kun ohjelmiston koko on kasvanut.

Yle Areenan uudelleen kehitetty toteutus antaa vertailukohtaa jo vakiintuneelle ja ikääntyneelle Framework 7:lle. Analyysien alkaessa Areenassa ei ollut vielä riviäkään lähdekoodia. Noin viiden kuukauden aikana koodia oli syntynyt noin 2 000 riviä. Kyseessä on siis Vaadin Framework 7:ää moninverroin pienempi ohjelma lähdekoodin määrän kannalta.

Teknisen velan osalta puhutaan myös aivan eri määristä kuin Framework 7:n kohdalla. Kun Framework 7 :ssä tekninen velka mitataan kymmenissä ja sadoissa vuorokausissa, on se Areenassa joitakin tunteja. Teknisen velan käyttäytyminen on myös hyvin erilaista Areenassa. Kehitystyön käynnistyttyä tekninen velka kasvaa melko tasaisesti yhdessä lähdekoodin määrän kanssa. Hieman ennen analysointijakson puolta väliä tekninen velka saavuttaa korkeimman arvonsa 2 h 37 min. Tämän jälkeen tekninen velka alkaa laskea päätyen lopulta arvoon 1 h 18 min. Teknisen velan suhdeluku käyttäytyy myös samankaltaisesti käyden korkeimmillaan 1 %:ssa ja pudoten lopuksi vain 0,1 %:iin. Ohjelmointivirheitä on Areenassa vain 0-2 kappaletta analysointijakson aikana. Haavoittuvuuksia on korkeimmillaan 9 kappaletta jakson aikana, ja jakson lopulla niitä ei ole yhtään.

Yle Areenan tuloksissa voivat heijastua kehitystiimin käyttämät työskentelymenetelmät ja etenkin ”siivouspäivien” soveltaminen. Tulokset kertovat myös jollain tasolla tapahtuvasta teknisen velan hallinnasta kehitystyön aikana, vaikka velan hallinta olisikin vain ”tavallista”, rutiinilla suoritettavaa kehitystyötä. Esimerkkinä mainittakoon konsolitulosteiden käyttäminen, joita kehittäjät voivat paikoin hyödyntää todetakseen koodin suoriutuvan oikein. Areenan kohdalla on otettava huomioon myös ohjelman koko verrattuna Vaadin Framework 7:ään. Uuteen projektiin on helpompaa soveltaa teknisen velan hallintaan liittyviä menetelmiä heti alusta alkaen, jolloin

teknisen velan määrä on mahdollista pitää alhaisena. Pienempi koodipohja mahdollistaa myös paremmat mahdollisuudet toteuttaa koodin refaktorointia.

Teknisen velan kokonaismäärä ja teknisen velan suhdeluku kertovat kuitenkin vain yleistilanteen teknisen velan mahdollisista vaikutuksista ohjelmiston kehitystyöhön. Teknisen velan konkreettinen sijainti lähdekoodissa on merkittävä tekijä, kun arvioidaan teknisen velan vaikutuksia kehitystyöhön. Mikäli runsas määrä teknistä velkaa sijaitsee moduulissa, jonka pariin joudutaan palaamaan useaan kertaan tai jota muokataan usein kehitystyön aikana, on se paljon vahingollisempaa kehitystyölle, kuin jos tekninen velka sijaitsisi vähemmän käytetyssä osassa ohjelmistoa. Teknisen velan tarkempaa sijaintia ja sen suhdetta kehitystyöhön ei tässä työssä tutkittu.

Muitakin tässä työssä saatuja tuloksia olisi mahdollista tarkentaa tulevissa tutkimuksissa. Ensinnäkin otosmäärää tulisi kasvattaa, jotta tuloksia voisi pitää yleispätevämpinä. Tämä edellyttäisi usean samankaltaisen ohjelmistoprojektin (esimerkiksi samankaltainen koko, tavoite, kehitysmenetelmät) tutkimista samanaikaisesti. Useiden samankaltaisten ohjelmistoprojektien tutkiminen eri segmenteistä tarkentaisi tuloksia erilaisten ohjelmistokehitysmenetelmien vaikutuksista teknisen velan syntyyn. Tulokset teknisen velan määrästä tarkentuisivat myös organisaatioiden sisällä tehtävällä tutkimuksella tai hyvin tiiviillä yhteistyöllä esimerkiksi kehitystiimin kanssa. SonarQuben sääntöjen muokkaaminen yhdessä kehitystiimin kanssa voisi tarkentaa SonarQuben tekemiä havaintoja lähdekoodin laadusta. Sääntöjä voisi muokata vastaamaan esimerkiksi organisaation sisäisesti käyttämiä ohjelmointikäytäntöjä. Samalla staattista lähdekoodianalyysiä voisi laajentaa esimerkiksi kehitystiimille suoritettavilla strukturoiduilla haastatteluilla.

6. JOHTOPÄÄTÖKSET

Tässä työssä on ollut tarkoitus tutkia miten, tekninen velka esiintyy ja kehittyy modernissa ohjelmistotyössä. Tutkimusta varten analysoitiin kaksi moderneilla ohjelmistokehitysmalleilla toteutettua ohjelmistoa käyttäen hyväksi SonarQubea, joka on lähdekoodin staattiseen analysointiin kehitetty työkalu. Analysoidut ohjelmistot olivat Vaadin Oy:n Vaadin Framework 7 ja Yleisradio Oy:n Yle Areena.

Tämän työn puitteissa voidaan todeta, että tekninen velka esiintyy lähdekoodissa melko perinteisinä, ”kouluesimerkkivirheinä”. Esimerkiksi koodin epäyhtenäiset nimeämiskäytännöt, literaalien käyttäminen vakioiden sijaan ja kapseloinnin rikkominen oliopohjaisessa kielessä olivat Vaadin Framework 7:ssä viiden useimman esiintyvän koodihajun joukossa. Perinnekoodi aiheutti myös osan ohjelmiston sisältämästä teknisestä velasta.

Tekninen velka kehittyi molemmissa projekteissa, kuten saattoi ennakoida. Vaadinilla, jossa teknisen velan hallintaa ei järjestelmällisesti toteuteta, velka kasvoi samassa suhteessa lähdekoodin määrän kanssa. Areena-projektissa, jossa ei myöskään ole järjestetty varsinaisesti teknisen velan hallintaan kohdistuvia toimenpiteitä, mutta koodipohjaa ajoittain kuitenkin refaktoroidaan, teknisen velan voidaan nähdä puolestaan laskevan kehitystyön edetessä.

Täysin velattomaan ohjelmistoon pyrkiminen modernissa ohjelmistotyössä ei ole realistista eikä järkevää. Joskus teknistä velkaa on pakko ottaa, joskus sitä kertyy huomaamattomasti, ja parhaassa tapauksessa tekninen velka on ohjelmistotyössä hyödynnettävä työkalu, jonka avulla oma tuote on kilpailijoita askeleen edellä. Avainasemassa ovat teknisen velan huomioiminen, havaitseminen ja hallinta, joiden laatu määrittää miten teknisen velan kanssa menestyy. Tekninen velka on osa modernia ohjelmistotyötä.

LÄHTEET

[1] M. O. Ahmad, D. Dennehy, K. Conboy, M. Oivo, "Kanban in software engineering: A systematic mapping study," *Journal of Systems and Software*, vol. 137, 2018, pp. 96–113.

[2] E. Allman, "Managing technical debt," *Queue*, vol. 10, no. 3, mar 2012, pp. 10–17.

[3] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, D. Thomas. (2001) Ketterän ohjelmistokehityksen julistus. Verkkosivu, saatavissa: http://agilemanifesto.org/iso/fi/manifesto.html. (Viitattu: 6.11.2018)

[4] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, D. Thomas. (2011) Julistuksen takana olevat periaatteet. Verkkosivu, saatavissa: http://agilemanifesto.org/iso/fi/principles.html. (Viitattu: 6.11.2018)

[5] W. N. Behutiye, P. Rodríguez, M. Oivo, A. Tosun, "Analyzing the concept of technical debt in the context of agile software development: A systematic literature review," *Information and Software Technology*, vol. 82, 2017, pp. 139–158.

[6] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, R. Sangwan, C. Seaman, K. Sullivan, N. Zazworka, "Managing technical debt in software-reliant systems," in *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, ser. FoSER '10. New York, NY, USA: ACM, 2010, pp. 47–52.

[7] H. F. Cervone, "Understanding agile project management methods using scrum," *OCLC Systems & Services*, vol. 27, no. 1, 2011, pp. 18–22.

[8] Z. Codabux, B. J. Williams, G. L. Bradshaw, M. Cantor, "An empirical assessment of technical debt practices in industry," *Journal of Software: Evolution and Process*, vol. 29, no. 10, 2017.

[9] G. Coleman, "Agile software development," *Software Quality Professional; Milwaukee*, vol. 19, no. 1, Dec 2016, pp. 23–29.

[10] W. Cunningham, "The wycash portfolio management system," *SIGPLAN OOPS Mess.*, vol. 4, no. 2, dec 1992, pp. 29–30.

[11] Git - git-log documentation. Verkkosivu, saatavissa: https://git-scm.com/docs/git-log. (Viitattu: 21.1.2019)

[12] Github - vaadin/framework: Vaadin 6, 7, 8 is a java framework for modern web applications. Verkkosivu, saatavissa: https://github.com/vaadin/framework. (Viitattu: 18.1.2019)

[13] Y. Guo, R. O. Spínola, C. Seaman, "Exploring the costs of technical debt management – a case study," *Empirical Software Engineering*, vol. 21, no. 1, 2016, pp. 159–182.

[14] Y. Guo C. Seaman, "A portfolio approach to technical debt management," in *Proceedings of the 2Nd Workshop on Managing Technical Debt*, ser. MTD '11. New York, NY, USA: ACM, 2011, pp. 31–34.

[15] Gwt. Verkkosivu, saatavissa: http://www.gwtproject.org/overview.html. (Viitattu: 8.11.2018)

[16] J. Holvitie, V. Leppnen, S. Hyrynsalmi, "Technical debt and the effect of agile software development practices on it - an industry practitioner survey," in *2014 Sixth International Workshop on Managing Technical Debt*, 2014, pp. 35–42.

[17] J. Holvitie, S. A. Licorish, R. O. Spínola, S. Hyrynsalmi, S. G. MacDonell, T. S. Mendes, J. Buchan, V. Leppänen, "Technical debt and agile software development practices and processes: An industry practitioner survey," *Information and Software Technology*, vol. 96, 2018, pp. 141–160.

[18] R. Jabbari, N. bin Ali, K. Petersen, B. Tanveer, "What is devops? a systematic mapping study on definitions and practices," in *Proceedings of the Scientific Workshop Proceedings of XP2016*, ser. XP '16 Workshops. New York, NY, USA: ACM, 2016.

[19] S. McConnell. (2007) Technical debt. Verkkosivu, saatavissa: http://www.construx.com/10x_Software_Development/Technical_Debt. (Viitattu: 30.1.2017)

[20] "Palaveri," Vaadin Oy, Ruukinkatu 2-4, Turku, lokakuu 19, 2016.

[21] "Palaveri," Yleisradio Oy, Uutiskatu 5, Helsinki, marraskuu 11, 2016.

[22] M. Poppendieck M. A. Cusumano, "Lean software development: A tutorial," *IEEE Software*, vol. 29, no. 5, 2012, pp. 26–32.

[23] M. Poppendieck T. Poppendieck, *Lean software development: an agile toolkit*. Boston, MA: Addison Wesley, 2003.

[24] N. Rios, R. O. Spínola, M. Mendonça, C. Seaman, "The most common causes and effects of technical debt: First results from a global family of industrial surveys," in *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '18. New York, NY, USA: ACM, 2018.

[25] J. Roche, "Adopting devops practices in quality assurance," *Commun. ACM*, vol. 56, no. 11, nov 2013, pp. 38–43.

[26] P. Rodríguez, A. Haghighatkhah, L. E. Lwakatare, S. Teppola, T. Suomalainen, J. Eskeli, T. Karvonen, P. Kuvaja, J. M. Verner, M. Oivo, "Continuous deployment of software intensive products and services: A systematic mapping study," *Journal of Systems and Software*, vol. 123, 2017, pp. 263–291.

[27] K. Schwaber J. Sutherland, *The Scrum Guide*, Nov 2017. Verkkosivu, saatavissa: https://www.scrumguides.org/docs/scrumguide/v2017/2017-Scrum-Guide-US.pdf

[28] Sonarqube. Verkkosivu, saatavissa: https://www.sonarqube.org/. (Viitattu: 26.4.2017)

[29] Sonarqube documentation - adding coding rules. Verkkosivu, saatavissa: https://docs.sonarqube.org/display/DEV/Adding+Coding+Rules. (Viitattu: 17.12.2018)

[30] Sonarqube documentation - architecture and integration. Verkkosivu, saatavissa: https://docs.sonarqube.org/display/SONARQUBE56/Architecture+and+Integration. (Viitattu: 14.12.2018)

[31] Sonarqube documentation - concepts. Verkkosivu, saatavissa: https://docs.sonarqube.org/display/SONARQUBE56/Concepts. (Viitattu: 18.12.2018)

[32] Sonarqube documentation - issue lifecycle. Verkkosivu, saatavissa: https://docs.sonarqube.org/display/SONARQUBE56/Issue. (Viitattu: 20.12.2018)

[33] Sonarqube documentation - issues. Verkkosivu, saatavissa: https://docs.sonarqube.org/display/SONARQUBE56/Issues. (Viitattu: 17.12.2018)

[34] Sonarqube documentation - metric definitions. Verkkosivu, saatavissa: https://docs.sonarqube.org/display/SONARQUBE56/Metric+Definitions. (Viitattu: 20.12.2018)

[35] Sonarqube documentation - quality gates. Verkkosivu, saatavissa: https://docs.sonarqube.org/display/SONARQUBE56/Quality+Gates. (Viitattu: 20.12.2018)

[36] Sonarqube documentation - rules. Verkkosivu, saatavissa: https://docs.sonarqube.org/display/SONARQUBE56/Rules. (Viitattu: 7.1.2019)

[37] Sonarqube documentation - security-related rules. Verkkosivu, saatavissa: https://docs.sonarqube.org/display/SONARQUBE56/Security-related+rules. (Viitattu: 18.12.2018)

[38] Y. Sugimori, K. Kusunoki, F. Cho, S. Uchikawa, "Toyota production system and kanban system materialization of just-in-time and respect-for-human system," *International Journal of Production Research*, vol. 15, no. 6, 1977, pp. 553–564.

[39] G. Suryanarayana, G. Samarthyam, T. Sharma, *Refactoring for Software Design Smells: Managing Technical Debt.* San Francisco: Elsevier Science & Technology, 2014, pp. 206–207.

[40] tech.yle.fi. Verkkosivu, saatavissa: https://yle.fi/aihe/yleisradio/techylefi. (Viitattu: 4.3.2019)

[41] Vaadin - company. Verkkosivu, saatavissa: https://vaadin.com/company. (Viitattu: 4.3.2019)

[42] Vaadin framework 7. Verkkosivu, saatavissa: https://vaadin.com/docs/v7/framework/introduction/intro-overview.html. (Viitattu: 7.11.2018)

[43] Vaadin roadmap. Verkkosivu, saatavissa: https://vaadin.com/roadmap. (Viitattu: 25.4.2017)

[44] M. Virmani, "Understanding devops & bridging the gap from continuous integration to continuous delivery," in *Fifth International Conference on the Innovative Computing Technology (INTECH 2015)*, 2015, pp. 78–82.

[45] What is agile software development? Verkkosivu, saatavissa: https://www.agilealliance.org/agile101. (Viitattu: 7.1.2019)

[46] What is kanban? Verkkosivu, saatavissa: https://kanbanblog.com/explained/. (Viitattu: 13.3.2019)

[47] R. Wilsenach. Devopsculture. Verkkosivu, saatavissa: https://martinfowler.com/bliki/DevOpsCulture.html. (Viitattu: 24.10.2018)

[48] E. Wolff S. Johann, "Technical debt," *IEEE Software*, vol. 32, no. 4, July 2015, pp. 94–97.

[49] Yhteinen taival – ylen vuosikymmenet. Verkkosivu, saatavissa: http://yle.fi/aihe/artikkeli/2016/04/04/yhteinen-taival-ylen-vuosikymmenet. (Viitattu: 4.3.2019)

[50] Yle areena ohjeet - mediasoitin. Verkkosivu, saatavissa: http://ohjeet.areena.yle.fi/hc/fi/articles/115002968829-Mediasoitin. (Viitattu: 8.11.2018)

[51] Yle areena ohjeet - usein kysytyt kysymykset. Verkkosivu, saatavissa: http://ohjeet.areena.yle.fi/hc/fi/articles/115002983165-Usein-kysytyt-kysymykset. (Viitattu: 8.11.2018)

[52] Ylen strategia. Verkkosivu, saatavissa: https://yle.fi/aihe/artikkeli/2017/10/24/ylen-strategia. (Viitattu: 4.3.2019)

LIITE A: SKRIPTI SONARQUBE-ANALYYSIEN AJAMISEEN

```
# python 3.x
# sq-analyzer
# Script runs SonarQube analysis for source code fetched from git-repository.
# Script needs an input file which contains the information about which commits
# SQ is supposed to analyze. Each line in input file contains the information of
# a single commit.
# Line's syntax is in form of:
# "<commit checksum> <version/name> <date of the commit in form YYYY-MM-DD>"
# without double quotes and <> -signs.
# For example:
# dt2lkn7 1.1 2016-08-24
#
# Usage:
# Run sq-analyzer.py in project's root folder where .git -folder is located.
# sq-analyzer.py needs an input file (txt) which contains information about the
# commits as first parameter and absolute path to the sonar-scanner.bat as
# second parameter.
#-------------------------------------------------------------------------------

import sys
import subprocess
import time

def shutDown():
   print("Program shutting down")
   exit()

def openFile(fileName):
   try:
      return open(fileName, "r")
   except FileNotFoundError:
      return False

def legalArguments(arguments):
   if len(arguments) != 3:
      return False
   if len(arguments[0]) != 7: # Commit checksum length
      return False
   if legalDate(arguments[2]) == False:
      return False
   return True

def legalDate(date):
   date = date.split("-")
   if len(date) != 3:
      return False
   for index in range(len(date)):
      if date[index].isdigit() == False:
         return False
      if index == 0 and len(date[index]) != 4:
         return False
      elif index != 0 and len(date[index]) != 2:
         return False
   return True
```

```
def checkout(commit, testRun):
   printCmd("git␣checkout␣" + commit)
   if(testRun == False):
      process = subprocess.run(["git", "checkout", commit],
                               stderr=subprocess.PIPE,
                               universal_newlines=True)
      if "error:" in process.stderr:
         shutDown()

def runSQ(version, date, scannerPath, testRun):
   projectVersion = str("sonar.projectVersion=" + version)
   projectDate = str("sonar.projectDate=" + date)
   printCmd(scannerPath + "␣—D␣" + projectVersion + "␣—D␣" + projectDate)
   if(testRun == False):
      print("SonarQube␣analysis␣underway...␣this␣might␣take␣a␣moment.")
      process = subprocess.run([scannerPath,
                               "—D",
                               projectVersion,
                               "—D",
                               projectDate],
                               stdout=subprocess.DEVNULL,
                               universal_newlines=True)

def skipLine(arguments):
   print("There␣is␣a␣problem␣with␣commit␣info␣file␣at␣line␣" + str(arguments))
   print("Skipping␣line.")

def printCmd(cmd):
   print("Running␣subprocess␣command:␣" + cmd)

def fileLen(fileName):
   for i, line in enumerate(fileName):
      pass
   return i + 1

#————————————————————————————————————————————————————————————————————————————
# Main program starts
#————————————————————————————————————————————————————————————————————————————

print("sq—analyzer␣v0.1\n")
startTime = time.time()

# Check that the parameters given to program are legal
test = False
fileName = sys.argv[1]
sqPath = sys.argv[2]

if len(sys.argv) == 4 and sys.argv[3] == "test":
   test = True
elif len(sys.argv) < 3 or len(sys.argv) > 4:
   print("Please␣specify␣the␣commit␣info␣file␣and␣sonar␣scanner␣path␣as␣the␣"
   "parameters␣for␣program.")
   shutDown()

f = openFile(fileName)
if f == False:
   print("File␣'%s'␣was␣not␣found." % (fileName))
   shutDown()

lineNb = 0
for line in f:
```

```
    lineNb += 1
    print("Handling␣the␣line␣number␣%d:␣%s" % (lineNb, line))
    arguments = line.split()
    if legalArguments(arguments):
        commit = arguments[0]
        version = arguments[1]
        date = arguments[2]

        checkout(commit, test)
        runSQ(version, date, sqPath, test)
    else:
        skipLine(arguments)

f.close()

endTime = time.time()
elapsedTime = endTime - startTime
print("Program␣run␣in␣'%d'␣seconds" % (elapsedTime))
```