Tampereen yliopisto

Mikko Ala-Fossi

TEKNISEN VELAN TAKAISINMAKSU

Informaatioteknologian ja viestinnän tiedekunta
Pro gradu -tutkielma
Huhtikuu 2020

TIIVISTELMÄ

Mikko Ala-Fossi: Teknisen velan takaisinmaksu
Pro gradu -tutkielma
Tampereen yliopisto
Tietojenkäsittelytieteiden tutkinto-ohjelma
Huhtikuu 2020

Modernissa, ketterässä ohjelmistotyössä työn nopeus ja laatu ovat merkittävässä roolissa. Nopeissa sykleissä tehtävää työtä ei kuitenkaan aina kyetä suunnittelemaan tai testaamaan loppuun asti, jolloin koodin laatu, luettavuus tai ylläpidettävyys heikkenevät. Lähdekoodin laadulle voi kuitenkin asettaa tavoitteita ja sitä voidaan mitata erilaisilla työkaluilla. Lähdekoodin laatuun liittyviä poikkeamia voidaan luonnehtia tekniseksi velaksi. Tekninen velka on metafora, johon voidaan soveltaa samanlaisia periaatteita kuin rahataloudessa käytettävään velkaan tai infrastruktuurin korjausvelkaan. Velan ottamiseen kuuluu esimerkiksi, että se maksetaan jossain vaiheessa takaisin ja se kerryttää korkoa.

Tutkimuksen tarkoituksena on selvittää, onko teknisen velan takaisinmaksuun tehty työ vähentänyt teknisen velan määrää ohjelmistossa. Tutkimuksessa esitellään teknisen velan ilmenemisen esimerkkejä ohjelmiston kontekstissa ja arvioidaan staattisen analyysin, koodityylianalyysin ja testikattavuuden menetelmillä ohjelmiston laskennallista laatua.

Tutkimuksessa havaitaan, että otettu tekninen velka voi aiheuttaa ylimääräistä työtä ja voi olla havaittavissa staattisen analyysin avulla. Tuloksista päätellään, että eri mittareilla saatujen tulosten perusteella voidaan arvioida jossain määrin lähdekoodin laatua ja että hyvin määritellyt testitapaukset helpottavat ohjelmiston ylläpitämistä. Tutkimuksessa käy esimerkkien kautta ilmi, että tekninen velka ei aina ole havaittavissa automaattisilla testeillä ja että teknisen velan oireet voivat lisätä merkittävästi ohjelmiston ylläpitoon liittyvää työtaakkaa.

Avainsanat: tekninen velka, staattinen analyysi, dynaaminen testaus, testikattavuus

Sisällys

1 Johdanto

Modernissa, ketterässä ohjelmistotuotannossa ohjelmistokehittäjän tehtävänä on tuottaa nopealla aikataululla toimivaa koodia, joka toteuttaa annetut vaatimukset [Agile Alliance 2001]. Aikataulun tuoman paineen, ylimalkaisten määrittelyjen ja rajallisten resurssien ristipaineessa menestyvän ohjelmistokehittäjän piirteisiin kuuluu kyky muodostaa mahdollisimman pienellä työllä mahdollisimman tarkka toteutus vaaditusta sovelluksesta. Mahdollisimman tarkka toteutus ei aina kuitenkaan vastaa tulevia tarpeita ohjelmaan tarvittavien muutosten suhteen.

Tässä pro gradu -tutkielmassa tutkitaan, miten ohjelmistokehityksessä voidaan hyödyntää teknisen velan metaforaa. Teknisen velan ottamisen hyötyjen ja riskien tasapainoa pohditaan. Tutkimuksessa pohditaan teknisen velan välttämisen tuottamaa vaikeutta ja teknisen velan takaisinmaksamisen aiheuttamaa lisähintaa. Teknisen velan metaforan implikoimien pääoman, koron ja konkurssin merkitystä tutkitaan.

Siirrettäessä ohjelmoitua lähdekoodia tuotantoon, otetaan niin sanottua teknistä velkaa. Vietäessä ohjelmakoodia tuotantoon hyväksytään, että mahdolliset tulevat muutokset voivat vaatia ylimääräistä työtä sovelluksen vaatimusten muuttuessa tai tarkentuessa. Iteratiivisessa, ketterässä ohjelmistokehityksessä tällainen työtapa kuuluu prosessin luonteeseen. Jokaisella uudella iteraatiolla voidaan tarkentaa ja täsmentää ohjelmiston toimintaa ja toteutusta [Stoica et al. 2013]. Tällainen työtapa kuitenkin vaatii selkeää kommunikaatiota projektin johdon ja sovelluskehittäjien väliltä, sillä jos sovelluksen kehityksessä ei varata lainkaan aikaa sovelluksen rakenteen ylläpitämiseen, voi kehitystyö ja ylläpito hidastua merkittävästi tai ohjelmistoon voi muodostua ennalta odottamattomia virheitä.

Ohjelmistoon muodostuvia ennalta odottamattomia virheitä voidaan ennaltaehkäistä muun muassa ohjelmistotestauksella, kirjoittamalla yksikkötestejä, tyypittämällä funktioiden argumentit, kouluttamalla ohjelmistokehittäjiä, noudattamalla yhtenäistä koodaustyyliä sekä koodikatselmoinneilla [Green ja Ledgard 2011]. Hyödyntämällä näitä menetelmiä edesautetaan laadukkaan koodin kirjoittamista.

Lähdekoodissa olevaa teknistä velkaa voi maksaa takaisin refaktoroimalla yksittäisiä metodeja, luokkia tai moduuleja. Refaktorointi on menetelmä, jossa lähdekoodin tekninen toiminnallisuus säilyy suunnilleen muuttumattomana, mutta jossa lähdekoodin luettavuutta ja jäsentelyä parannetaan [Fowler et al. 1999]. Projektin vaatimusten muuttuessa ratkaisevalla tavalla voidaan kuitenkin joutua tilanteeseen, jossa tulee päättää merkittävien osien uudelleenkirjoittamisesta, sillä järjestelmällinen refaktorointi muodostuisi liian kalliiksi ja monimutkaiseksi tehtäväksi tai sillä olemassa olevaa teknistä toteutusta ei voi muokata uusimpien mallien mukaiseksi.

Teknistä velkaa esiintyy kaiken kokoisissa projekteissa. Sitä hallitsemalla voidaan edistää liiketoiminnan tavoitteita. Kerryttämällä liikaa teknistä velkaa voidaan kuitenkin päätyä tilanteeseen, jossa ohjelman kehittäminen muuttuu mahdottomaksi. Siksi on tärkeää tiedostaa teknisen velan merkitys ja käyttää aikaa sen hallitsemiseksi.

1.1 Tutkimuskysymykset ja aineisto

Tutkielmassa tehdään kirjallisuuskatsaus teknisen velan määritelmään, arviointiin, hallintaan ja takaisinmaksuun. Tutkielmassa kuvaillaan, millaiset ohjelmistokehitysmenetelmät ja prosessit voivat tukea teknisen velan hallinnassa sekä millaiset ratkaisut voivat teknisen velan suhteen johtaa kestämättömään tilanteeseen. Tämän lisäksi tarkastellaan hieman miten teknistä velkaa voi muodostua, miten sitä voidaan mitata ja miten sitä voidaan luokitella.

Tässä tutkimuksessa tutkitaan, miten start-up-mallisessa ohjelmistokehityksessä voidaan käyttää teknistä velkaa ja sen takaisinmaksamista eduksi. Tutkielmassa pyritään selvittämään, onko mahdollista vähentää ohjelmiston teknistä velkaa ja parantaa ohjelmiston laatua toteuttamalla ohjelmistoon teknisen velan takaisinmaksuun tarkoitetun muutosprojektin.

Tutkielma on toteutettu case-tutkimuksena. Tutkimuksessa lähtökohtana ja aineistona on kesällä 2019 alkanut ohjelmiston muutosprojekti, jossa uudelleenkirjoitettiin verkkosovelluksen käyttöliittymä. Muutosprojektissa ensisijaisena tavoitteena on ollut toteuttaa aiempaa suorituskykyisempi käyttökokemus ja käyttöliittymä asiakasrajapinnassa. Muutosprojektin toissijaisena tavoitteena oli parantaa lähdekoodin luettavuutta ja luotettavuutta, mutta tälle tavoitteelle ei asetettu raja-arvoja tai tavoitteita.

Tässä tutkielmassa tarkastellaan, millaista teknistä velkaa ohjelmistossa on ollut, millaisia ohjelmistotuotannon riskejä teknisestä velasta on aiheutunut ja miten muutosprojektissa on onnistuttu teknisen velan takaisinmaksun suhteen. Ohjelmistossa ollutta teknistä velkaa ja mahdollisia riskejä käsitellään esimerkkitapausten kautta ja muutosprojektin onnistumista arvioidaan analysoimalla lähdekoodia erilaisilla työkaluilla. Tämän lisäksi pohditaan, millaista mahdollista uutta teknistä velkaa on muodostunut muutosprojektin aikana.

1.2 Tutkimuksen rakenne

Tutkimuksen luvussa 2 esitellään teknisen velan teoreettinen konsepti ja miten se voidaan erotella erilaisiin ilmenemistyyppeihin. Ensimmäiseksi tutustutaan teknisen määritelmään ja siihen, millaisia eri tulkintoja teknisestä velasta on tehty. Tämän jälkeen tar-

kastellaan, miten teknisen velan metafora on verrattavissa rahalliseen velkaan. Lopuksi esitellään tarkemmin erityyppisiä teknisen velan ilmenemismuotoja.

Luvussa 3 käsitellään teknisen velan hallintaa ja sen vaikutuksia ohjelmistoprojekteissa. Aluksi pohditaan tarkemmin, miten ohjelmistoprojektin osana voidaan suorittaa teknisen velan hallintaa. Tämän jälkeen tarkastellaan pragmaattista ohjelmistotuotannon lähestymistapaa, jossa teknistä velkaa otetaan tietoisesti ja hallitusti sekä esitellään, millaisia vaikutuksia hallitsemattomalla teknisen velan kerryttämisellä voi olla projektihallinnan osalta. Lopuksi keskitytään tarkemmin teknisen velan aiheuttamiin riskeihin.

Luvussa 4 käsitellään teknisen velan hallintaan käytettävissä olevia mittareita ja työkaluja, kuten staattinen koodianalyysi, koodityylitarkastukset, dynaamiset testit sekä syklomaattinen kompleksisuus sekä CRAP-arvo, jolla voidaan arvioida lähdekoodin ymmärrettävyyttä ja luotettavuutta.

Luvussa 5 tutustutaan tutkimuksen kohteena olevaan projektiin, ohjelmistotuotteen tekniseen taustaan ja kehityshistoriaan sekä pohditaan aiemman ohjelmistokehityksen valintojen vaikutuksia tulevaan ohjelmistokehitykseen. Luvussa 6 käsitellään ohjelmistosta saatuja mittaustuloksia. Luvussa 7 esitellään ohjelmistosta tehdyt havainnot esimerkkien kautta. Luvussa 8 koostetaan tutkimuksen johtopäätökset.

2 Tekninen velka

Tässä luvussa esitellään tekniseen velkaan liittyvä määritelmä. Luvussa esitellään teknisen velan metafora, jonka määritelmään paneudutaan ensimmäisessä kohdassa. Toisessa kohdassa tutustutaan, millaisiin tyyppeihin teknistä velkaa voidaan jaotella. Tutkimuksessa käsiteltävälle termille *tekninen velka* ei ole yksiselitteistä määritelmää tietojenkäsittelytieteitä tutkivassa kirjallisuudessa. Teknisen velan metaforalle on esitetty useita eri tulkintatapoja, jotka kuvailevat ohjelmistokehityksen ilmiötä. Näitä eri tulkintatapoja käsitellään kohdassa 2.3.

2.1 Teknisen velan määritelmä

Termin *tekninen velka* esitteli ensiksi Ward Cunningham [1993]. Termillä Cunningham kuvaa sitä, että uuden lähdekoodin tuotantoon vieminen on verrattavissa velan ottamiseen. Riippumatta ohjelmiston kehityksen suunnitelman laajuudesta ja tarkkuudesta, ohjelmistoa kehitettäessä aina toimitaan vajavaisen tiedon varassa. Cunninghamin mukaan on tärkeää, että ohjelmisto tuotetaan parhaalla mahdollisella tavalla suunnitelmasta ja tiedossa olevasta tiedosta koostettavan näkemyksen perusteella ja siirretään mahdollisimman nopeasti tuotantoympäristöön tarkemman ymmärryksen kartuttamiseksi. Tätä määritelmää ja toimintamallia tukee Lehmanin [1980] näkemys, ettei ohjelmiston suorituskykyä, muovautumiskykyä sekä yleistä laatua voi suoraan suunnitella ja rakentaa järjestelmän ominaisuuksiksi. Tällaiset piirteet muodostuvat inkrementiaalisten muutosten ja parannusten kautta.

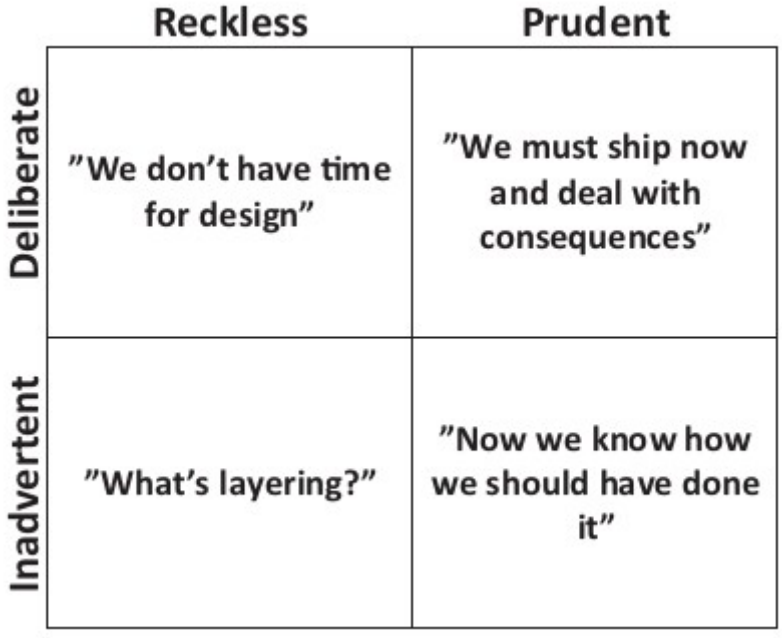
Tekninen velka on terminä vain metafora, ei yleisesti hyväksytty konsepti tai teoria eikä tekniselle velalle ei ole yhteistä sanastoa [Codabux ja Williams 2013]. Tekninen velka on kuitenkin analoginen reaalimaailman velalle. Velka on sopimus käyttöön saatavasta rahasummasta, joka kuuluu maksaa takaisin. Sen avulla saavutetaan muuta taloudellista hyötyä, esimerkiksi mahdollistetaan ohjelmiston tarjoaminen asiakkaille mahdollisimman nopeasti. Otettu velka lähtökohtaisesti kuuluu maksaa takaisin. Ohjelmistoon kertyneen teknisen velan takaisinmaksu tapahtuu muokkaamalla aiemmin toteutettua lähdekoodia luettavammaksi tai suorituskykyisemmäksi. Otetulle velalle kertyy myös korkoa, joka tässä kontekstissa voi tarkoittaa vaaditun ohjelmointityön kasvavaa määrää aiemmin kirjoitetun teknisen velan jäädessä uuden koodin alle.

Ohjelmakoodin osia voi kuvailla teknisen velan määritelmän kaltaisella ilmaisulla ”code smells” eli niin sanotut purkkaratkaisut [Fowler et al. 1999]. Purkkaratkaisut ovat muutoksia ohjelman lähdekoodiin, jotka ovat ongelmallisia ohjelmiston lähdekoodin laadun ja arkkitehtuurin kannalta. Purkkaratkaisu-termille on useita, asiaa hieman eri kannalta kuvailevia ilmaisuja, kuten ”anti-pattern” eli antisuunnittelumalli, joka voi kattaa myös suunnittelumallien puutetta ja ”spaghetti code” eli spagettikoodi, joka on läh-

dekoodia, jonka toimintaa on vaikea lukea ja jossa luokkien vastuualueet voivat olla sekaisin. Lähdekoodi, jossa esiintyy antisuunnittelumalleja tai epäselvä lähdekoodi eivät ole itsessään teknistä velkaa, vaan ne voivat olla teknisen velan ilmenemismuoto.

Teknisen velan määritelmää on laajennettu myös erottamaan tahattomasti muodostuvan teknisen velan tietoisesti otetusta teknisestä velasta. Tahattomasti tai tiedostamatta muodostuvaa teknistä velkaa voi muodostua, jos kehittäjillä ei ole kattavaa tuntemusta ohjelmistotuotannon parhaista käytännöistä [McConnell 2004] tai tuotettaessa ohjelmistoa nopeasti keskeneräisiä tai epäselviä vaatimuksia noudattaen. Tietoiset päätökset teknisen velan ottamisesta usein osataan perustella tarkemmin. Myöhemmässä teoksessaan Fowler [2009] kuitenkin esittää, että tekniseksi velaksi luettaisiin juuri Cunninghamin [1993] määrittelemällä tavoin harkitusti tehdyt oikaisut, jotka eivät välttämättä osoittaudu jatkokehittämisen kannalta hyviksi.

Kuvassa 1 on nelikenttä, joka esittää Fowlerin [2009] tulkintaa eri tavoista kerryttää teknistä velkaa, tietoisesti tai tiedostamatta. Kuvassa erotellaan velan kerryttäminen nelikenttään. Nelikenttä jakaantuu sen mukaan, onko teknistä velkaa kerrytetty tietoisesti (deliberate) vai tiedostamatta (inadvertent) sekä harkitsevaisesti (prudent) vai harkitsemattomasti (reckless). Nelikentässä on kuvailtu englanniksi, millä tavoin tekniseen velkaan johtanutta työtä voidaan reflektoida tai perustella. Esimerkiksi tietoisesti, mutta harkitsemattomasti kerrytettyä velkaa voi muodostua aloitettaessa kehitystyö ilman suunnitelmaa, jolloin projektin seuraavassa vaiheessa joudutaan takaisinmaksamaan aiemmin otettua teknistä velkaa. Jatkokehityksen kannalta ei aina kuitenkaan ole merkitystä sillä, mihin nelikentän ruuduista teknisen velan alkuperä luokitellaan.



Kuva 1. Nelikenttä teknisestä velasta [Fowler 2009].

Myöhemmässä kirjallisuudessa teknisen velan määritelmä on laajentunut. Teknisen velan voidaan ajatella kattavan myös huonosti kirjoitettua tai huonosti dokumentoitua koodia. Tällaisen laajennetun määritelmän mukaan teknisen velan voi jakaa tyypeittäin vaatimusmäärittelyyn, suunnitteluun, lähdekoodiin, testaukseen, koodin koostamiseen, dokumentointiin, infrastruktuuriin, versiointiin ja vikatiloihin liittyviin teknisiin velkoihin [Li et al. 2015b]. Lähdekoodiin kirjoitettu osa, joka ei ole loppuun asti suunniteltu voidaan itsessään laskea tekniseksi velaksi. Vaihtoehtoisesti, teknisen velan voidaan myös tulkita olevan mittausteorian mukainen mittauksen kohteena olevan asian attribuutti [Lavazza et al. 2018]. Tätä tulkintaa laajentaen voidaan päätellä, että tekninen velka esimerkiksi vaatimusmäärittelyissä tai koodissa ei ole erityyppistä teknistä velkaa, vaan teknisen velan eri tyypit kuvailevat eri asioita, jotka sisältävät teknistä velkaa. Tällainen tulkinta voi helpottaa kokonaisuuksien muodostamista ohjelmistoprojektiin kertyneestä teknisestä velasta ja auttaa ratkaisun muodostamisessa.

2.2 Teknisen velan metafora

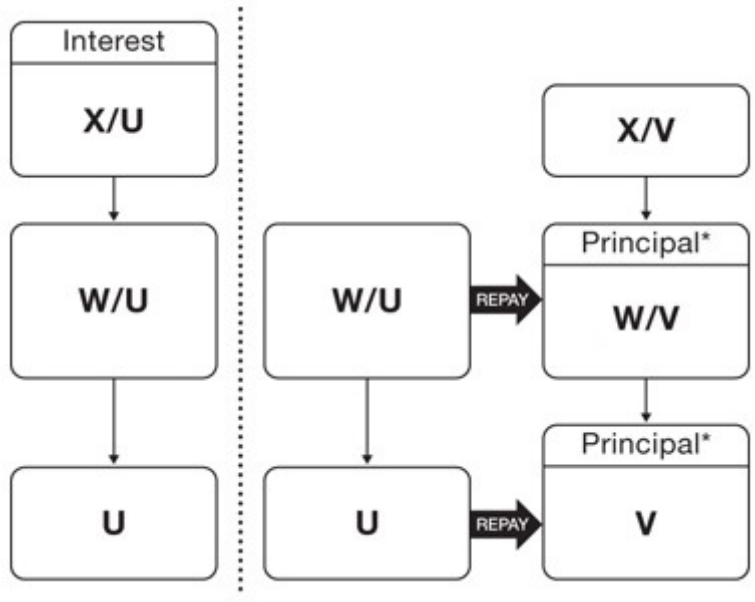
Teknisen velan alkuperäinen metafora pohjautuu reaalimaailman velkaan [Cunningham 1993]. Metaforan avulla päätöksenteon ei tarvitse perustua puhtaasti teknisellä tai taloudellisella tasolla käytävään keskusteluun. Teknisen velan metafora tarjoaa keskusteluun yhteisiä termejä [Ampatzoglou et al. 2015]. Tekninen velka viittaa ohjelmistokehityksessä rakenteeseen tai toteutukseen, joka mahdollistaa projektille asetettujen tavoitteiden täyttämisen, mutta joka on myöhemmin hankala vaihtaa toiseen toteutukseen. Tekninen velka on näkymätöntä tarkasteltaessa ohjelmistoa sen ulkoisten ominaisuuksien perusteella. Tekninen velka kuitenkin ilmenee järjestelmän muuttuessa tarvittavan työn vaikeutena ja muutostyöhön kohdistuvina lisäkustannuksina. Tässä kohdassa tarkastellaan tarkemmin, miten reaalimaailman velka ja tekninen velka vastaavat toisiaan.

Velan vaikutukset eivät välttämättä näy välittömästi ulkopuoliselle tarkastelijalle. Kuten finanssivelan ottaminenkaan, ei tekninenkään velka välttämättä näy järjestelmästä päällepäin. Asuntoa tarkastelemalla ei selviä, onko jäljellä olevaa velkaosuutta lainkaan vai vain puolet jäljellä. Finanssivelan, kuten myös teknisen velan, koron kertyminen ja velan takaisin maksamisen vaikutukset voivat kuitenkin näkyä muissa yhteyksissä. Otetun velan vaikutukset voivat näkyä esimerkiksi siinä, kuinka nopeasti tai kattavasti uusia investointeja tai ylläpitoa voidaan toteuttaa esimerkiksi lomamatkojen tai uusien sovellusominaisuuksien muodossa. [Mishkin ja Eakins 2012]

Teknisen velan metaforaa voidaan verrata myös rakennusteollisuudessa ja infrastruktuurissa yleisesti käytettyyn korjausvelan käsitteeseen, sillä molemmilla ilmaistaan laatuun kohdistuvaa kompromissia. Korjausvelka on korjausvajetta eli sen avulla voidaan ilmaista, paljonko rakennukseen tai infrastruktuuriin olisi tullut investoida, jotta se olisi voitu säilyttää hyvässä kunnossa. Kun ennakoivan kunnossapidon investoinneista

karsitaan muut paitsi kiireelliset ja välttämättömät korjaukset, syntyy korjausvelkaa. Korjausvelan oireita voivat olla muun muassa heikko sisäilmanlaatu ja rikkoutuvat vesijohdot [Virtala ja Äijö 2011]. Siinä missä tekninen velka ilmaisee muutosten tekemiseen vaadittavaa työpanosta, korjausvelkaa voidaan käyttää laiminlyödyn ylläpitämisen arvon ilmaisemiseen [Madsen 2006]. Kuten korjausvelkaa, teknistä velkaa ei voida ohittaa käytössä olevassa järjestelmässä loputtomasti. Vaikka vanhentunutta tietojärjestelmää voidaan käyttää rajatussa kontekstissa lähes rajattomasti, ympäröivän maailman tiedonkäsittelyvaatimukset saattavat ajaa sen ohi. Tämän lisäksi tietojärjestelmässä piilevä tekninen velka voi tulla ajan myötä ilmi kuten vuosiluvun tallettaminen kahdella merkitsevällä luvulla ennen vuotta 2000 [Strassmann 2000].

Velan korkovaikutus perustuu aikaan. Rahallisessa velassa velkasummalle kerrytetään korkoa ennalta sovitun sopimuksen mukaisesti vuosittain [Mishkin ja Eakins 2012]. Teknisen velan koron vaikutukset eroavat kuitenkin finanssivelasta siten, että tekniselle velalle ei välttämättä realisoidu korkoa vasta kuin siinä vaiheessa, kun teknistä sovellusta tarvitsee muuttaa [Ozkaya et al. 2019]. Kuvassa 2 on esitetty kaaviolla teknisen velan koron (*interest*) ja teknisen velan pääoman (*principal*) vaikutukset. Kuvassa on kuvattuna saman tuotteen kaksi eri kehityslinjavaihtoehtoa, nopeammin käyttöön otettu mutta haastavammin muokattavissa oleva U ja hitaammin hieman haastavammin käyttöön otettava V jonka muokkaaminen on suhteellisesti helpompaa. Tuotteisiin U ja V tarvittavat uudet ominaisuudet ovat W ja X. Jotta voidaan siirtyä käyttämään toista toteutusta, tulee korvata kaikki toiminnallisuudet toisen kehitysvaihtoehdon toteutuksilla. Teknisen velan korko tarkoittaa siis niitä uusia oikaisuratkaisuja, jotka tulee tehdä aiemmin otetun teknisen velan päälle, jotta sen kanssa voidaan jatkaa kehitystä. Ottamalla teknisen velan korkona muodostuva osuus osaksi projektia, projektissa olevan teknisen velan määrä kasvaa. Teknisen velan kerryttämisen on vaihtoehtona teknisen velan takaisinmaksu, osittain tai kokonaan. Tähän ratkaisuun saatetaan tulla tilanteessa, jossa, aiemmasta ratkaisusta johtuen uuden muutoksen käyttöönottaminen on kalliimpaa kuin jos velkaa ei olisi otettu. Tällöin joudutaan hyväksymään, että kaikki uudelleenkirjoitettavat toiminnallisuudet ovat teknisen velan pääomaa, joka tulee maksaa toteuttamalla toiminnallisuudet täysin uudelleen.



Kuva 2. Teknisen velan koron ja takaisinmaksamisen vaikutukset [Ozkaya et al. 2019].

Muutostilanteessa tulee ratkaista, miten teknistä velkaa käsitellään. Voidaan hyväksyä, että maksetaan vain tekniselle velalle laskettavissa oleva korko tai maksetaan velka kokonaan pois ja katetaan alkupääoma. Jos pääoma maksetaan kokonaisuudessaan pois, yksittäisen uuden toiminnallisuuden implementointi tulisi suhteessa kalliiksi. Toisaalta tulevien toiminnallisuuksien kanssa ei tarvitsisi työskennellä teknisen velan kanssa. Jos päätetään vain kattaa teknisen velan korko ja jatkaa toiminnallisuuksien implementointia velan kanssa, siirtyminen vaihtoehtoiseen toteutukseen muuttuu toiminnallisuus kerrallaan vaikeammaksi. Teknisen velan metafora eroaa finanssivelasta siten, ettei korkoa tai muita käsittelykuluja kerry kuukausittain vakiomäärää, vaan vaikutus voi vaihdella erilaisista syistä [Ozkaya et al. 2019].

2.3 Teknisen velan tyypit

Tekninen velka itsessään on ylätasolla kuvailtu abstrakti metafora, jolla voidaan kuvailla ohjelmistoprojektin ominaisuuksia. Ohjelmistoprojektin toteutukseen usein osallistutaan monella eri tavalla, muun muassa määrittelyyn, varsinaiseen toteuttamiseen sekä testaamiseen. Selkeyden vuoksi näiden osa-alueiden vastuiden perusteella on hahmoteltu erillisiä teknisen velan tyyppejä. Sillä teknisen velan määritelmä ei ole vielä täysin vakiintunut, on tällaisia tyyppijaotteluja useita erilaisia. Tässä kohdassa tutustutaan tekniselle velalle määriteltyihin eri tyyppeihin. Alakohdissa esitellään projektin eri vaiheisiin ja eri vastuualueisiin perustuvaa jaottelua, jossa lähdekoodissa, arkkitehtuurissa ja projektinhallinnassa esiintyvä tekninen velka tunnistetaan omiksi kokonaisuuksiksi.

Näiden lisäksi tutustutaan muuhun tekniseen velkaan, joka jää näiden osa-alueiden ulkopuolelle ja esitellään muita tapoja jaotella teknistä velkaa.

2.3.1 Lähdekoodissa esiintyvä tekninen velka

Tekninen velka usein esiintyy osana lähdekoodia. Lähdekoodia voidaan tutkia lukemalla sekä mittaamalla ja testaamalla sen ominaisuuksia automaattisilla työkaluilla. Tässä kohdassa selostetaan, millaista teknistä velkaa lähdekoodissa voi esiintyä. Lähdekoodissa esiintyvä tekniseen velkaan sisältyy muun muassa staattisessa analyysissä esiintyvät virheet sekä monimutkainen ja vaikeasti ylläpidettävää lähdekoodia. Näiden lisäksi termi peritty koodi eli *legacy code* kuvaa sellaista koodia, joka voi sisältää teknistä velkaa.

Lähdekoodissa esiintyvää teknistä velkaa voidaan arvioida mittaamalla lähdekoodin monimutkaisuutta tai suorittamalla staattisen analyysin testejä. Lähdekoodin monimutkaisuudelle käytettävä mitta-arvo on syklomaattinen kompleksisuus. Syklomaattinen kompleksisuus ilmaisee uniikkien polkujen määrää koodin polun alusta loppuun. Mitä enemmän suorituspolkuja lähdekoodissa on, sen vaikeammin se on ymmärrettävissä ja siten virheiden esiintymisen riski kasvaa [McCabe, 1976]. Syklomaattista kompleksisuutta muodostuu usein lähdekoodiin kirjoitettaessa nopeasti uutta koodia miettimättä loppuun asti, miten koodi kannattaisi pilkkoa osiin. Monimutkaisuus tarkoittaa tässä yhteydessä sitä, että lähdekoodiin tehtävät muutokset on vaikeampi toteuttaa.

Tutkielman kohdassa 4.5 tutustutaan tarkemmin syklomaattiseen kompleksisuuteen. Tämän lisäksi suorittamalla staattisen analyysin testejä voidaan saada ilmi sellaisia virheitä koodissa, jotka voivat aiheuttaa esimerkiksi tietoturva-aukkoja. Staattinen analyysi voi paljastaa useita virheitä. Staattisesta analyysistä kerrotaan tarkemmin kohdassa 4.1

Peritty koodi mielletään usein lähdekoodiksi, jota ei aktiivisesti ylläpidetä tai kehitetä. Määritelmää tarkentanut Feathers [2004] toteaa, että myös tuoreempi koodi, jolle ei kirjoiteta testejä ja joka on vaikeasti testattavaa, on myös perittyä koodia. Perittyä koodia muodostuu esimerkiksi, kun vain yksi henkilö tietää miten lähdekoodi toimii ja koodin jatkokehityksen vastuu siirtyy toiselle ilman, että alkuperäinen kehittäjä on vastaamassa koodin toiminnasta heräävііn kysymyksiin [Ritchie 2010]. Peritty koodi mielletään usein lähdekoodiksi, jonka kirjoittamisessa ei ole noudatettu yleisesti ohjelmoinnissa hyviksi havaittuja käytäntöjä kuten koodityylin tarkastusta.

Perityssä koodissa voi myös esiintyä epäselvyyksiä luokkien, metodien ja muuttujien nimeämisessä, epäselviä luokkavastuita ja monimutkaisia rakenteita sekä tyypityksen puutteen tai ohjelmassa käsiteltävän tiedon puuttumisesta johtuvia poikkeustilanteita. Hyviksi havaittuihin käytäntöihin kuuluu muun muassa myös funktioparametrien tyyppivihjaus, muuttujien selkeä nimeäminen sekä muuttujien käyttäminen koodissa määriteltyjen kovakoodattujen arvojen sijaan. Funktioparametrien tyyppivihjaus auttaa lukijaa hahmottamaan nopeammin, millainen syöte metodille voidaan odottaa. Tämä

auttaa myös virheiden välttämisessä, sillä koodin staattinen analyysi voi tunnistaa tyyppivihjauksista ennalta tilanteet, joissa väärän tyyppistä dataa olisi siirtymässä funktioon.

Tekninen velka ja peritty koodi peruskäsitteinä voidaan mieltää eri asioiksi [Cunningham 1993]. Tekninen velka voi sisältää muitakin seikkoja kuin lähdekoodissa esiintyvän teknisen velan. Lähdekoodissa esiintyvä tekninen velka ei toisaalta välttämättä ole perittyä koodia, vaan noudattaa ohjelmoinnin perusperiaatteita. Peritty koodi voi myöskin olla täysin toimivaa ja vapaata teknisestä velasta aiheutuvista ongelmista siten, ettei sen suorittamisessa päädytä virhetilanteisiin ja se toteuttaa annetut määritykset täysin. Toisaalta peritystä koodista muodostuva lähdekoodi voidaan mieltää sisältävän teknistä velkaa, sillä sitä muutettaessa joudutaan tekemään ylimääräistä työtä tulkittaessa aiemman ohjelmoijan valintoja ja refaktoroitaessa vanhaa koodia. [Feathers 2004]

Johtuen perityn koodin ja teknisen velan merkitysten eroista, voi olla vaikea määritellä, millaista jonkin ennalta tuntemattoman lähdekoodin tekninen velka on. Muodostuuko perittyä koodia ratkaistaessa ongelmia purkkaratkaisuilla vai onko tällöin kyseessä lähdekoodi, joka sisältää teknistä velkaa? Toisaalta on perusteltua pohtia, otetaanko tietoisesti tai tiedostamatta teknistä velkaa, jolloin lähdekoodista muodostuu perittyä koodia? Loppujen lopuksi ei ole kuitenkaan merkitystä, onko lähdekoodi ollut alusta alkaen perittyä koodia vai lähdekoodia, johon on kertynyt teknistä velkaa. Lähdekoodissa olevista heikkouksista aiheutuvat lisätyö ja -kustannukset voidaan mieltää teknisen velan koroksi. [Klinger et al. 2011]

2.3.2 Arkkitehtuurissa ja suunnittelussa esiintyvä tekninen velka

Ohjelmistoarkkitehtuurissa esiintyvä tekninen velka voidaan ymmärtää ohjelmiston sisäisenä luokkarakenteiden monimutkaisuutena. Tämän lisäksi arkkitehtuuriseen tekniseen velkaan voidaan sisällyttää järjestelmän riippuvuudet muista palveluista kuten erillisten mikropalveluiden monimutkaiset keskinäiset riippuvuudet sekä riippuvuus laitteistoista tai suoritusympäristöstä. [Li et al. 2015a]

Ohjelmistoarkkitehtuurinen tekninen velka voi ilmetä useilla eri tavoilla. Esiintymismuotoja ovat esimerkiksi liian heikosti määritellyt luokkavastuurajat tai luokkarakenteiden sykliset suhteet [Melton ja Tempero 2007]. Tällaiset antisuunnittelumallit voivat muodostua tarpeesta saada mahdollisimman paljon aikaiseksi mahdollisimman pienillä muutoksilla. Tämä johtaa helposti siihen, että toisistaan riippumattomat asiat ovat toisiinsa sidottuja ilman selkeitä rajoja. Ilman selkeitä luokkarajoja koodin ylläpidettävyys heikkenee ja tehdyt muutokset voivat aiheuttaa ennalta-arvaamattomia sivuvaikutuksia.

Ohjelmistoarkkitehtuurin tilaa voidaan arvioida esimerkiksi ATAM-menetelmällä (Architecture Tradeoff Analysis Method). ATAM-menetelmää käytetään arkkitehtuuristen valintojen riskien hallitsemiseksi. Menetelmä on kaksivaiheinen [Clements et al.

2000]. Ensimmäisessä vaiheessa muodostetaan ohjelmiston liiketoiminnnasta ja teknologisista ratkaisuista vastaavien tahojen kanssa käsitys ohjelmistolle osoitetuista odotuksista ja sen arkkitehtuurisista valinnoista. Toisessa vaiheessa arvioidaan järjestelmän kykyä toteuttaa sille asetettuja tavoitteita erilaisten skenaarioiden kautta ja raportoidaan tuloksista liiketoiminnasta ja arkkitehtuurista vastaaville tahoille. ATAM on mielekkäintä toteuttaa ohjelmiston tuotannon alkuvaiheessa, kun arkkitehtuuriin tehtävien muutosten hinta on hallittavissa.

Alkuperäisen määritelmän mukainen tekninen velka on lähtökohtaisesti oikotien ottamista tuntematta täysin kaikkia ohjelmiston kehityksessä vaadittuja yksityiskohtia. Siksi on luontevaa, että suunnitteluvaiheessa toteutuksen alkuvaiheessa valitaan suoraviivainen toteuttamisprosessi tarkemman määrittelyn sijaan. Erityisesti muuttuvassa liiketoimintasegmentissä toimivissa ohjelmistoprojekteissa tällaisesta ratkaisusta on etua, ettei ohjelmiston kehityksessä ei ole pakollista vakaannuttaa kehittämistä [Li et al. 2014]. Ohjelmistoprojekteissa, joilla on lain säätelemät turvallisuus- ja laatuvaatimukset, suunnitteluvaiheen teknistä velkaa ei ole mahdollista harkitusti samassa määrin ottaa. Tällaisia ohjelmistoja voivat olla esimerkiksi potilastietojärjestelmät sekä liikenteenohjausjärjestelmät.

2.3.3 Muu projektiin liittyvä velka

Teknisen velan voi jakaa tyypeittäin vaatimusmäärittelyyn, suunnitteluun, lähdekoodiin, testaukseen, koodin koostamiseen, dokumentointiin, infrastruktuuriin, versiointiin ja vikatiloihin liittyviin teknisiin velkoihin [Li et al. 2015b]. Teknisen velan voidaan ymmärtää myös kattavan muita osa-alueita projektin elinkaaren varrelta. Esimerkiksi projektihallinnolliseen tekniseen velkaan voi liittyä muun muassa ohjelmistotuotannon mallin valintaan ja tuotantomallin käyttöön liittyvät seikat ja puutteet. Tämän lisäksi myös ohjelmoijien muodostaman sosiaalisen yhteisön ominaispiirteet voivat muodostaa velkaa, sillä ohjelmistotuotanto on hyvin riippuvaista siihen osallistuvien kehittäjien hyvinvoinnista.

Osa moderneista suurista julkisista ohjelmistoprojekteista sekä suurin osa historiallisista ohjelmistoprojekteista on toteutettu vesiputousmallin (waterfall) mukaisesti. Vesiputousmallissa ohjelmistoprojekti etenee suoraviivaisesti määrittely, suunnittelu, toteutus, testaus sekä implementaatio vaiheisiin. Tällaisessa työn järjestelyn mallissa teknistä velkaa voi muodostua alun määrittelyvaiheessa, puutteellisen tietämyksen takia. Puutteellisia määrittelyjä ei varsinaisesti pääse korjaamaan. Vesiputousmallista ohjelmistokehitystä on kritisoitu siitä, että sitä noudattamalla projektit voivat myöhästyä suunnitellusta aikataulusta. [Osman 2010]

Ketterä ohjelmistokehitys usein noudattaa jossain määrin Ketterän ohjelmistokehityksen julistusta [Agile Alliance 2001], joka antaa raamit iteratiiviselle ohjelmistokehi-

tykselle. Tähän usein liittyy muun muassa kahden viikon sprint-jaksot, jotka sisältävät määrittely-, suunnittelu-, toteutus- ja testausvaiheet. Sprint-jaksolle valitaan vain rajallinen määrä tehtäviä, jotka kehittäjät itse voivat valita. Ketterän ohjelmistokehityksen julistus ei itsessään ohjeista miten työtä tulisi järjestellä. Ketterän ohjelmistokehityksen julistuksessa määritellyt kehitystyön vapaat raamit eivät ole yksiselitteiset ja tämä epäselvyys voi aiheuttaa ristiriitatilanteita ohjelmistotyön järjestämisessä. Ristiriidat voivat muodostua esimerkiksi siitä, että sprint-jaksolle valitut tehtävät viivästyvät jakson aikana ilmestyneiden muiden tehtävien takia. Tällaisessa tilanteessa, jossa sprint-jaksolle nostetaan uusia tehtäviä, tulisi myös määritellä sprintin tavoitteet uudelleen ja laskea vaadittavat työmäärät uudelleen.

Projektihallinnollista teknistä velkaa voidaan luokitella projektihallinnan antisuunnittelumalleihin. Antisuunnittelumalleja on useita [Stamelos 2009]. Yksi esimerkki projektinhallinnollisesta antisuunnittelumallista on, että projektissa ei otettu käyttöön minkäänlaista projektinhallinnollista mallia, vaan ohjelmistoa tuotetaan täysin vapaamuotoisesti. Täysin vapaamuotoisessa projektissa voi olla haastavaa ohjata ohjelmiston valmistumista ja asettaa tavoitteita, erityisesti tiimin kasvaessa. Tästä johtuen voidaan myös argumentoida, että mikäli ohjelmistokehitysmallia noudatetaan, mutta esimerkiksi yrityksen toimitusjohtajalle annetaan oikeus nostaa ohjelmistossa esiintyviä virheitä muiden työtehtävien edelle, projektinhallinnassa on riskitekijöitä. Projektihallinnolliseen tekniseen velkaan voi vaikuttaa sopimalla työpaikalla yhteisistä säännöistä projektimalliin liittyen ja opettelemalla noudattamaan projektimallin edellyttämiä taitoja. Päivittäisessä työssä käytettävät työn järjestämisen taidot edesauttavat projektin edistymistä. Ohjelmistoprojektien toteuttamiseen osallistuu ohjelmoijien lisäksi myös muuta henkilöstöä, jotka osallistuvat suunniteluun, budjetointiin ja päätöksentekoon. Jotta ohjelmistotuottamista voidaan johtaa ja ohjata kohti tavoitteita hallitusti, tulee projektin mukailla jonkinlaista ohjelmistotuotantomallia.

Monet nykyiset ohjelmistot hyödyntävät useita muita eri ohjelmistopaketteja, joihin muodostetaan riippuvuussuhde. Tällaisia ovat esimerkiksi NPM- tai Composer-pakettihallintatyökaluilla asennettavat ohjelmistopaketit. Ohjelmistopaketit esimerkiksi mahdollistavat aiemmin toteutetun toiminnallisuuden nopean implementoinnin ohjelmistoon. Riippuvuussuhteen tarjoavien ohjelmistopakettien päivittäminen on osa ohjelmiston ylläpitotyötä, sillä päivitykset voivat tarjota joko uusia ominaisuuksia tai tietoturvapäivityksiä. Ohjelmistopaketteihin tehtävät päivitykset eivät kuitenkaan aina ole taaksepäin yhteensopivia [Kaur ja Mahajan 2015]. Tämä tarkoittaa sitä, että ohjelmistoon tulee tehdä muutoksia, jotta ohjelmistopaketin uuden version voi ottaa käyttöön. Tämän lisäksi voi olla tilanteita, joissa ohjelmistopaketin ylläpitäminen on päättynyt. Ohjelmistopaketin ylläpitämisen lopettamisen perusteena voi olla esimerkiksi vastaavan tai paremman toiminnallisuuden tullessa tarjolle esimerkiksi toisessa, vastaavanlaisessa ohjel-

mistopaketissa. Tällöin riippuvuussuhde aiempaan pakettiin, jonka ylläpitäminen on päättynyt muodostaa teknistä velkaa, erityisesti, jos on mielekästä vaihtaa toteutusta noudattamaan uuden paketin tarjoamaa ohjelmointirajapintaa. Riippuvuussuhteisiin liittyvä tekninen velka on periaatteessa lähdekoodissa esiintyvää teknistä velkaa, sillä sen sijainti on osoitettavissa lähdekoodista. Tässä tutkielmassa riippuvuussuhteen muodostama tekninen velka erotellaan kuitenkin omakseen, sillä tekninen velka johtuu ohjelmistopaketin muodostamasta velkataakasta.

Tämän lisäksi ohjelmistokehityksessä voi esiintyä sosiaalista velkaa, joka voi ilmentyä esimerkiksi luottamuksen puutteena tai puutteellisina kykyinä ratkaista ohjelmistokehityksen haasteita [Tamburri et al. 2015]. Ohjelmistokehittäjien puutteellinen ymmärrys käyttämästään teknologiasta ja tietojenkäsittelytieteiden perusteista on tekijä, jonka voi laskea ohjelmistoprojektin tekniseksi velaksi. Ohjelmistokehittäjät, jotka eivät ylläpidä taitojaan tai seuraa aktiivisesti käyttämiensä ohjelmistojen, ohjelmointikielien ja ohjelmistokehitysrajapintojen kehitystä saattavat päätyä käyttämään useammin vanhentuneita tai huonoiksi todettuja ohjelmointitapoja. Tästä syystä onkin tärkeää panostaa opetukseen ja oppimiseen myös työpaikoilla itsenäisen opiskelun lisäksi.

3 Teknisen velan hallinta

Tässä luvussa kerrotaan, miten teknistä velkaa voidaan käyttää osana sovelluskehitystä. Ensin käydään läpi, miten teknistä velkaa voidaan hallita osana muuta projektinhallintaa. Tämä tarkoittaa lähinnä teknisen velan seurantaa ja teknisen velan takaisinmaksamista. Tämän jälkeen pohditaan, miten teknistä velkaa voidaan käyttää projektin tavoitteiden edistämiseksi. Teknistä velkaa voidaan parhaimmillaan käyttää työkaluna bisnesriskien kommunikoimiseen ja voittojen maksimoimiseen. Kohdassa 3.2 analysoidaan, millaisia hyötyjä teknisen velan ottamisesta voi olla, jos se tehdään hallitusti. Kohdassa 3.3 käsitellään, mitä voi tapahtua, jos teknistä velkaa ei hallita tai jos sitä kertyy järjestelmään liikaa.

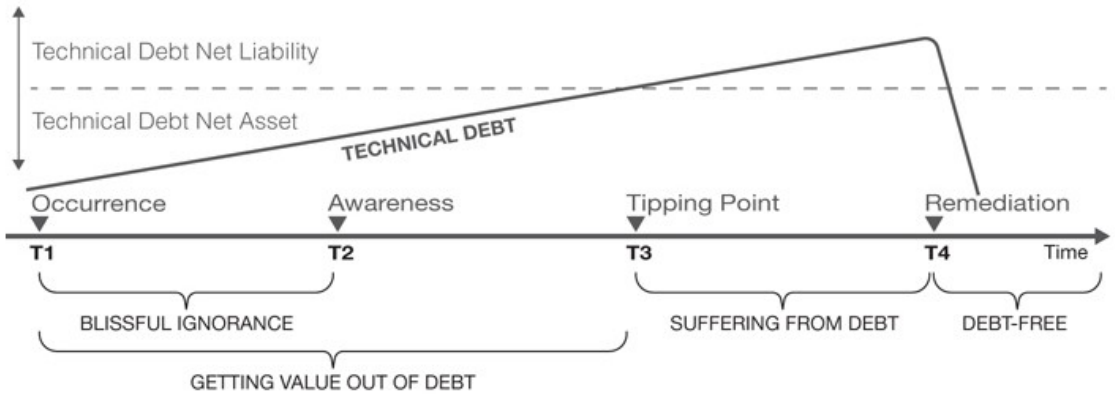
Hallittu teknisen velan ottaminen voi olla hyvin hyödyllistä uusien ohjelmistojen ja palveluiden kehityksessä, sillä tällä tavoin voidaan pienillä investoinneilla tarjota nopeasti valmis tuote. Tämän lisäksi vältetään ylimääräisen työn tekeminen, sillä yhtenä projektin mahdollisena lopputuloksena voi olla projektin päättyminen. Tällöin ylimääräistä aikaa ei tarvitse käyttää projektin myöhempiä vaiheita varten varautuen tai projektin muokattavuuden varmistamiseen. Kun projektin rahoitus tai kehityssuunta on vakiintunut, voidaan käyttää aikaa teknisen velan takaisin maksamiseen.

3.1 Teknisen velan hallitseminen osana projektia

Toimivan projektinhallinnan edellytyksenä on, että projektilla on jonkinlainen suunnitelma ja strategia. Tässä suhteessa teknisen velan hallitseminen ei eroa muista projektin osa-alueista. Tekninen velka voi olla osakokonaisuus projektinhallintasuunnitelmassa, jolloin velkaa tietoisesti voidaan hallita projektin edetessä. Projektia voidaan myös johtaa siten, ettei teknistä velkaa seurata lainkaan, jolloin teknisen velan kartoitus jää kehittäjien keskinäiseksi toiminnaksi tai se laiminlyödään täysin [Yli-Huumo et al. 2015]. Tässä luvussa kuvaillaan tarkemmin, miten tekninen velka voidaan ottaa huomioon projektia johdettaessa.

Tekninen velan merkitys realisoituu lähinnä ajan edetessä ja ohjelmiston muuttuessa. Ohjelmistoprojekteissa on usein tarkoituksena ohjelmiston muuttaminen vastaamaan uusia vaatimuksia sekä olemassa olevan ohjelmiston ylläpitäminen. Kuvassa 3 on kuvattuna aikajanalle projektiin sisältyvän teknisen velan hallinnan tasoa projektin edetessä. Aikajana alkaa siitä hetkestä, kun teknistä velkaa on muodostunut järjestelmään (T1 - Occurence). Aikajanalle on myös määritetty ajanhetki, jolloin teknisestä velasta tullaan tietoisiksi (T2 - Awareness). Tekninen velka voidaan nähdä muodostuvan ongelmaksi sellaisessa käännekohdassa (T3 - Tipping point) jonka jälkeen teknisen velan ot-

tamisen tuottama arvo ei enää ole suurempi kuin sen tuottamat ongelmat ja niiden hinta. Kun teknisestä velasta tullaan tietoisiksi, jopa ennen kuin tekninen velka alkaa muodostumaan ongelmaksi (T3), voidaan suorittaa teknisen velan takaisinmaksaminen (T4 - Remediation). Jos tekninen velka maksetaan kokonaan takaisin, kyseisen teknisen velan korkovaikutukset päättyvät. Vaihtoehtoisesti voidaan valita myös olla maksamatta teknistä velkaa takaisin, kokonaan tai osittain. Tämä ei kuitenkaan tarkoita, että teknisestä velasta kertyisi korkoa koko ajan. Tekninen velka kertyy muun muassa silloin, kun ohjelmistoa tarvitsee muokata tai jos ympäröivät tekijät muuttuvat. [Ozkaya et al. 2019]



Kuva 3. Teknisen velan aikajana [Ozkaya et al. 2019].

Teknisen velan hallitsemiseksi ohjelmistoprojektissa on oleellista, että projektissa sovitaan, miten tekninen velka määritellään. Tekninen velka on itsessään vain metafora, joten on tulkinnanvaraista, mikä tarkalleen sisältyy tekniseen velkaan. Eräs ratkaisumalli on, että suoraviivaisesti valitaan teknisen velan mittaamiseksi mielekkäät työkalut, joilla voidaan analysoida lähdekoodia, visualisoidaan tällä tavoin mitatun teknisen velan osuutta koodissa ja seurataan sen kehittymistä projektin edetessä. Tällä tavoin mitattuna merkittävä osa teknisestä velasta jää kuitenkin hahmottamatta, sillä tekniseksi velaksi voidaan ymmärtää myös ohjelmiston arkkitehtuurinen tai projektinhallinnollinen velka. Teknisen velan mittaamiseen tarkoitettuja työkaluja esitellään luvussa 4. Hyödynnettäessä työkalujen lisäksi ohjelmoijien omaa asiantuntemusta lähdekoodiin jääneistä oikaisuista ja listaamalla niitä omaan listaansa saavutetaan hieman parempi näkemys teknisen velan kattavuudesta. Kun teknisen velan määritelmä on saatu sovittua projektin sisäisesti ja se on saatu seurantaan, teknistä velkaa voidaan visualisoida ja seurata. Tämän jälkeen teknisen velan vaikutuksia, juurisyitä ja sen aiheuttamaa lisähintaa kehitykselle voidaan arvioida. [Lim et al. 2012]

3.1.1 Teknisen velan välttäminen

Yksi tehokkaimpia keinoja teknisen velan hallitsemiseksi on ylimääräisen teknisen velan ottamisen välttäminen. Tässä alakohdassa kerrotaan yksinkertaisista menetelmistä, joilla voi edistää ohjelmiston laatua ja välttää turhan teknisen velan kerryttäminen. Teknistä velkaa voidaan välttää muun muassa jatkokouluttamalla ohjelmoijia, noudattamalla hyväksi havaittuja koodauskäytäntöjä, kuten käyttämällä ennalta asetettuja koodityylisääntöjä, järjestämällä koodikatselmointeja sekä kirjoittamalla koodiin automaattisia testejä. Käytännön ohjelmistotuotannossa ja alan kirjallisuudessa on tunnistettu ohjelmointikäytäntöjen [Green ja Ledgrad 2011] ja koodikatselmointien merkitys otettavaan tekniseen velkaan [Mäntylä ja Lassenius 2009].

Ohjelmoijien valmiuksia tuottaa laadukasta lähdekoodia voidaan edistää tarjoamalla mahdollisuutta kouluttautua työajan puitteissa ja innostamalla ohjelmoijia tutustumaan itsenäisesti erilaisiin teknologioihin. Käyttämällä aikaa työajan puitteissa kouluttautumiseen tai työtaitojen harjoittelemiseen, voidaan saavuttaa parempi työtehokkuus sekä lisävarmuutta tuotettuun lähdekoodiin. Esimerkiksi opettelemalla työajan puitteissa automaattisten testien kirjoittamista projektissa voidaan helposti kasvattaa varmuutta lähdekoodin toimivuuteen. [Codabux et al. 2014]

Koodityylisäännöt voidaan mieltää ohjelmointikielen syntaksista erilliseksi tai sitä täydentäväksi säännöstöksi. Koodityylisäännöt eivät täten vaikuta lähdekoodin konekielisen merkityksen tulkitsemiseen. Koodityylisäännöistä voidaan muodostaa koodityylimäärittelyitä. Koodityylimäärittelyt sisältävät lähdekoodin kirjoitusasuun ja tyyliin liittyviä sääntöjä, jotka voidaan kokonaisuutena ottaa käyttöön joko projektin skaalassa tai osana koko ohjelmointikielen määrittelyä. Koodityylimäärittelyistä kerrotaan tarkemmin kohdassa 4.2. Koodityylisäännöt edesauttavat yhteistyössä tehtävän ohjelmointityön sujuvuutta vähentämällä koodityyliin tehtävien muutosten määrää osana varsinaisia muutoksia. Suuria kokonaisuuksia käsitellessä on tärkeää, että versiohallintaan ei muodostu ylimääräisiä muutosristiriitoja ja yhtenäisellä koodityylillä projektin tasolla voidaan välttää syntaksista riippumattomat muutokset [Zou et al. 2019].

Koodikatselmointi on lähdekoodiin tehtävän muutoksen vertaisarviointi, jossa joku toinen kuin ohjelmoija itse tarkastaa lähdekoodin. Koodikatselmoinnin puuttuminen voi johtaa lähdekoodin tuotantoon vientiin, jossa ohjelmistoon lisätään tahattomasti virheitä [McIntosh et al. 2015]. Tarkastaja voi kommentoida, hyväksyä tai hylätä muutosehdotuksen ja antaa muun muassa muutosehdotuksen rakenteeseen, sanavalintoihin, suorituskykyyn ja ylläpidettävyyteen liittyvää palautetta. Koodikatselmoinnit ovat hyvä tapa ylläpitää osaamista ja oppia uutta. Koodikatselmointeja voidaan käyttää myös opetusmenetelmänä [Wang et al. 2012]. Katselmointien järjestäminen osaksi työaikaa voi olla haastavaa etenkin, jos lähdekoodia osaa lukea vain pieni joukko. Tämän lisäksi haasteita voi syntyä, jos muutosehdotuksessa on todella paljon muutettuja rivejä, muutettu koodi

on tyyliltään vaikeaselkoista tai jos katselmoitava projekti on katselmoijalle ennalta vieras. Ennalta tuntemattomaan lähdekoodiin voi olla haastavaa antaa arviota, mutta lähtökohtaisesti koodin luettavuutta voi arvioida myös koodin visuaalisen estetiikan perusteella.

Automaattiset testit ovat lähdekoodia, jota ei suoriteta ohjelman ajon aikana, vaan niitä käytetään ohjelmiston toiminnan varmistamiseen. Automaattisten testien kirjoittaminen ja ylläpitäminen on lähdekoodin kirjoittamisesta erillistä työtä, joka kuuluu läheisesti lähdekoodin tuottamiseen [Mili ja Tchier 2015]. Automaattiset testit kirjoitetaan usein osaksi lähdekoodin versiohistoriaa ja niiden avulla voidaan varmistaa ohjelman toimivuus sen eri versioissa. Sillä automaattiset testit eivät ole ohjelman liiketoiminnan kannalta tuottavaa lähdekoodia, useassa pienellä tiimillä, nopeasti kehitettävässä ohjelmistossa testejä ei kirjoiteta tai niitä kirjoitetaan hyvin vähän. On kuitenkin huomattava, että ilman automaattisia testejä ainoat tavat varmistua ohjelmiston toimivuudesta ovat joko sisäistää koko lähdekoodi tai manuaalisesti testata jokaisen toiminnallisuus. Koko lähdekoodin ja sen toiminnan sisäistäminen voi olla mahdotonta ihmiselle. Kattavan manuaalisen testauksen suorittaminen jokaisen tehdyn muutoksen jälkeen ei myöskään ole mahdollista ilman merkittävää investointia tai työpanosta. Tästä syystä automaattisten testien kirjoittaminen on tärkeää projektin alkuvaiheista lähtien, jotta kehityksessä vältetään umpikuja lähdekoodin epävarman toiminnan takia. Automaattisesta testaamisesta kerrotaan lisää kohdissa 4.3 sekä 4.4.

3.1.2 Teknisen velan takaisinmaksu

Tässä luvussa kerrotaan, millaisia eri keinoja on takaisinmaksaa kerrytettyä teknistä velkaa. Teknisen velan takaisinmaksaminen edellyttää muutoksia ohjelmiston toteutukseen, ja menetelmät tällaisten muutosten tekemiseen ovat refaktorointi ja uudelleenkirjoittaminen. Takaisinmaksu voidaan suorittaa osittain tai kokonaisuudessaan, ennen tai jälkeen teknisen velan aiheuttamat oireet ilmenemisen.

Lähdekoodin refaktoroinnilla tarkoitetaan prosessia, jossa teknistä velkaa takaisinmaksetaan vaihtamalla olemassa olevan toteutus osittain tai kokonaan uuteen ilman että sen toimintalogiikka muuttuu. Refaktorointi on kurinalainen ja hallittu toimintatapa. Refaktorointia on esimerkiksi koodissa esiintyvän toiston siirtäminen omiin muuttujiin, funktiohin tai kokonaan uusiin ohjelman luokkiin tai moduuleihin. Tällainen toimintatapa sopii erityisesti jo käytössä olevaan oikein toimivaan lähdekoodiin hyvin, jos yksittäiset muutokset harkitaan tarkasti ja huomioidaan olemassa olevan toteutuksen rajoitteet. Refaktoroinnissa merkittävänä apuna toimii automaattinen testaus, jolla voidaan varmistaa ohjelman toiminta muutosten jälkeen. [Fowler et al. 1999]

Uudelleenkirjoittaminen tarkoittaa olemassa olevan toteutuksen korvaamista täysin uudella toteutuksella ilman, että olemassa olevan lähdekoodin toimintatapaa toisinne-

taan. Lähtökohtaisesti ohjelmiston ja sen osien täysi uudelleenkirjoittaminen on kalliimpaa kuin olemassa olevan ohjelman ylläpitäminen tai refaktorointi. Ensinnäkin olemassa olevan sovelluksen toteuttamien toiminnallisuuksien ja vaatimusten täysi uudelleenkirjoittaminen voi olla kalliimpaa kuin vanhan ylläpitäminen. Toisekseen olemassa oleva sovellus on todettu tuotannossa toimivaksi huolimatta korjattavista puutteista. Tämän lisäksi uusi sovellus sisältäisi myös oman joukkonsa uusia, kartoittamattomia epävarmuustekijöitä. [Codabux ja Williams 2013]

Teknisen velan pienimuotoista ja hallittua takaisinmaksua refaktoroiden tai täyttä takaisinmaksamista uudelleen kirjoittaen suurempia kokonaisuuksia voidaan suorittaa osana muuta ohjelmiston ylläpitoa tai kirjoitettaessa uusia toiminnallisuuksia. Tällöin teknisen velan aiheuttamat lisäkustannukset katetaan osana kehitystyötä. Tällaisessa tapauksessa on tärkeää, että teknisen velan aiheuttamat haasteet kommunikoidaan selkeästi projektin edistymistä suunnitellessa. Suuremman, tiedossa olevan teknisen velan takaisinmaksuun usein tarvitaan kuitenkin erikseen suunniteltua, systemaattista työtä, joka voi hidastaa itsessään muiden projektin osien valmistumista varatessaan ohjelmistokehitysresursseja. [Yli-Huumo et al. 2015]

3.2 Teknisen velan käyttäminen projektihallintatyökaluna

Tässä kohdassa pohditaan, miten harkittua teknisen velan ottamista voidaan käyttää projektissa strategisesti hyödyksi. Usein projektin alkuvaiheessa projektin kannalta merkittävät yksityiskohdat eivät ole ennakkoon tiedossa. Tällöin on kannattavaa toteuttaa yksinkertaisin mahdollinen toteutus ensin, jotta voidaan selvittää valmiilla tuotteella, mitkä ovat todelliset tarpeet. Panostamatta edellisessä alakohdassa 3.1.2 esiteltyihin teknisen velan välttämiseen ja ottamatta erikseen aikaa olemassa olevan teknisen velan takaisinmaksuun voidaan nopeuttaa tuloksia tuottavan koodin valmistumista. Tämä tarkoittaa tietoista lähdekooditasolla esiintyvän teknisen velan ottamista. Myöhemmässä kohdassa kerrotaan myös tarkemmin teknisen velan varsinaisista riskeistä.

Teknistä velkaa voidaan käyttää työkaluna tuottavan lähdekoodin nopeassa kehittämisessä. Tällöin otetaan tietoinen riski siitä, että teknisen velan oireet, kuten vaikeaselkoinen lähdekoodi tai määrittelemättömät virhetilanteet, voivat aiheuttaa ongelmia ja teknisestä velasta koituva haitta voi kasvaa suuremmaksi kuin teknisestä velasta saavutettu hyöty. Käytettäessä teknistä velkaa projektihallinnan työkaluna, projektin eri osapuolilla tulee olla selkeä käsitys teknisen velan vaikutuksista ja määrästä. Tällöin myös päätöksentekijät, jotka eivät työskentele päivittäin teknisen toteutuksen parissa, voivat hahmottaa teknisen velan aiheuttamia riskejä paremmin. [Ozkaya et al. 2019]

Minimivaatimustason täyttävän tuotteen (*Minimum Viable Product*) tasoisen ratkaisun toteuttaminen täysivaltaisen ja loppuun asti suunnitellun sijaan nopeuttaa tuotantovientikelpoisen tuotteen valmistumista [Moogk 2012]. Tällaisessa tapauksessa teknistä

velkaa voi muodostua siitä, että ensimmäisissä kehitysversioissa on jätetty joitain ominaisuuksia suunnittelematta loppuun asti. Näiden lisäksi toteuttamattomat ominaisuudet jätetään tuotteen tehtävälistalle (backlog). Tehtävälistalle kertyneet ominaisuudet, jotka ovat jääneet toteuttamatta eivät ole teknistä velkaa. Sillä tekninen velka liittyy olemassa olevaan toteutukseen, joka sisältää oikaisuja, ei tehtävälistalle kertynyt toteuttamaton toiminnallisuus ole teknistä velkaa. Toteutetut toiminnallisuudet, jotka eivät toimi odotetulla tavalla, voivat myös olla seurausta puutteellisesta määrittelystä tai vajavaisesta toteutuksesta.

Kaiken kaikkiaan teknistä velkaa voidaan käyttää lähdekoodin nykytilan kommunikoimiseen ohjelmoijien ja projektijohdon välillä [Li et al. 2015b]. Tällä tavoin projektijohto ymmärtää paremmin uusiin muutoksiin ja ylläpitoon liittyvät riskit ja kykenee muodostamaan tarkempia arvioita projektin tekniseen toteutukseen liittyen. Tarkalla teknisen velan hyödyntämisellä esimerkiksi voidaan saavuttaa kilpailuetua markkinoilla lisäämällä ohjelmistoon uusia ominaisuuksia tavanomaista nopeammin tai voidaan välttää projektin epäonnistuminen teknisen velan aiheuttaman haitan kasvaessa suuremmaksi kuin siitä saavutettava hyöty.

3.3 Teknisen velan riskit

Kuten myös reaalimaailman velkaan, niin myös teknisen velan ottamiseen ja sen käyttämiseen liittyy riski. Tässä kohdassa pohditaan sitä, millaisia riskejä liittyy tekniseen velkaan ja käsitellään myös, millaisia riskejä otetaan, jos projektissa päätetään kasvattaa teknisen velan määrää. Erityisesti hallitsematta ja seuraamatta kerrytetty tekninen velka sisältää riskin siitä, että kertyneen teknisen velan myötä tulee kattaa metaforan mukaisia lisäkustannuksia. Teknisen velan hallitsemista ei mielletä tärkeäksi projektin alkukehityksen aikana tai kun projektin jatkorahoituksen saamisen kannalta on edullisempaa toteuttaa ominaisuuksia ja kerryttää jonkin verran teknistä velkaa. Vaikka teknisen velan välttäminen tai takaisin maksaminen ei olisikaan ajankohtaista, on kuitenkin tietoinen valinta olla seuraamatta teknistä velkaa lainkaan [Ozkaya et al. 2019].

Yksi merkittävimmistä asioista, joka auttaa hallitsemaan teknistä velkaa, on kyky kommunikoida teknisen velan tilannetta järjestelmässä [Ozkaya et al. 2016]. Teknistä velkaa, josta ei olla tietoisia ja jota ei ole kartoitettu, ei voida kommunikoida eteenpäin. Teknisen velan merkityksen ja oireiden ymmärtäminen on helppoa ohjelmoijille, jotka työskentelevät päivittäin järjestelmän kanssa. Muutoksia tehtäessä lähdekoodiin tekninen velka käy selkeästi ilmi lähdekoodin ymmärtämisen vaikeutena. Pelkkää ohjelmoijien henkilökohtaista kokemusta ei ole kuitenkaan voi käyttää luotettavana tiedonlähteenä teknisestä velasta, jos järjestelmän alkuperäiset kehittäjät eivät enää ole käytettävissä projektissa. Alkuperäisillä kehittäjillä on lähtökohtaisesti paremmat valmiudet kertoa tiedoista, jotka eivät ole luettavissa puuttuvassa tai puutteellisessa dokumentaatiossa.

Jos projektin aikataulusta ja budjetista päättäville tahoille ei voi kommunikoida, millaisia riskejä ohjelmisto sisältää, heidän on vaikeampaa suunnitella ajankäyttöä tai ennakoida projektin kehitykseen liittyviä piileviä riskejä. Tämän lisäksi ilman tarkkaa tietoa teknisen velan tilanteesta järjestelmässä, on myös hankalampi perustella, miksi teknisestä velasta kumpuavat bugit johtuvat ja miksi niitä esiintyy.

Teknisen velan ottamisen riskeihin kuuluvat teknisen velan aiheuttama koodimuutosten vaikeutuminen, teknisen velan oireena esiintyvät virhe- ja vikatilanteet sekä ohjelmiston kehitystyön pysähtyminen [Ramasubbu ja Kemerer 2015]. Erityisesti hallitsematon teknisen velan ottaminen voi lopulta johtaa tilanteeseen, jossa tekniseen velkaan liittyvät riskit realisoituvat ennen kuin teknistä velkaa ehditään maksamaan takaisin. Tekninen velka voi ilmentyä esimerkiksi ohjelmiston ulkoisessa laadussa virhetilanteina. Tällainen riskien realisoituminen voi myös tapahtua nopeammin kuin teknistä velkaa hallittaessa osana projektia. Hallitsemattomasti kerrytettyä teknistä velkaa voi myös olla mahdotonta korjata ilman kattavaa uudelleenkirjoittamistyötä.

Tekninen velka voi ilmentyä esimerkiksi kopioituna koodina tai antisuunnittelumalleina kuten kohdassa 2.3 on kuvailtu. Tämän lisäksi teknistä velkaa sisältävästä lähdekoodista voi myös havaita korkeaa syklomaattista kompleksisuutta, joita kuvaillaan tarkemmin kohdassa 4.5.

Teknisen velan kanssa toimiminen voi aiheuttaa projektiin osallistuville ohjelmoijille ylimääräistä stressiä ja siten laskee työtehoa [Besker et al. 2018]. Työnjärjestelyssä lähdekoodin ymmärtämiselle ei usein rajata erikseen aikaa. Jos lähdekoodi on kirjoitettu noudattamatta parhaiksi havaittuja käytäntöjä, koodiin tehtävien muutosten tekeminen voi edellyttää enemmän työtä ja suunnittelemista kuin mitä ennalta voisi suunnitella. Ennakoimattomat vaikutukset voivat johtua esimerkiksi kopioiduista koodiosasista tai hallitsemattomista riippuvuusrakenteista. Hallitsemattoman ja vaikeaselkoisen lähdekoodin ylläpitäminen voi johtaa tilanteeseen, jossa ohjelmoijat voivat pelätä muutosten tekemistä lähdekoodiin. Ohjelmoijien pelko tehtäviä koodimuutoksia kohtaan johtuu siitä, että niiden vaikutukset järjestelmän muodostamassa kokonaisuudessa voivat olla täysin tuntemattomat. Tuntemattomien vaikutusten ja vaikeasti luettavan lähdekoodin aiheuttama sivuoire voi olla hidastunut tuottavuus. Tämän takia uusien sovellusversioiden julkaiseminen voi viivästyä, sillä ohjelmiston kehittäjät haluavat varmistaa kaikin keinoin, kuten manuaalisilla testeillä, ettei järjestelmään muodostu virhetilanteita.

Manuaalisessa testaamisessa käydään kaikki sovelluksen eri käyttötilanteet läpi syöttämällä tiedot käsin sovelluksen käyttöliittymän kautta. Teknisen velan vaikutukset voi huomata myös uusia työntekijöitä perehdytettäessä. Jos lähdekoodi on vaikeaselkoista, epäjohdonmukaista tai vaikeasti testattavaa, on haastavampaa perehdyttää uusi työntekijä järjestelmän kehittäjäksi kuin jos lähdekoodi olisi helpommin lähestyttävää ja sisäistettävää. Vaikeaselkoiseen ja epäjohdonmukaiseen lähdekoodiin on merkittävästi

haastavampaa tehdä uusia muutoksia, jotka eivät aiheuta ennalta odottamattomia vaikutuksia. Tällaisissa tilanteissa teknisen velan takaisinmaksua voidaan viivyttää ottamalla lisää teknistä velkaa esimerkiksi kopioimalla lähdekoodia paikasta toiseen tai periyttämällä metodeja tuntematta kokonaiskuvaa minimoidakseen muutettavien rivien määrän.

Pahimmassa tilanteessa teknistä velkaa ottaessa voi myös päätyä metaforan mukaisesti konkurssiin, eli tilanteeseen, jossa teknistä velkaa on niin paljon, että sen kanssa ei voi jatkaa, sillä velkataakka on liian suuri. Tällöin projektissa käytettävät resurssit eivät riitä teknisen velan korkojen kattamiseen eikä myöskään koko teknisen velan takaisinmaksuun. Tällaisessa tilanteessa projektin teknistä kehittämistä ei voida enää jatkaa. Teknistä velkaa ei voi maksaa takaisin, koska sitä on liikaa. [Lim et al. 2012]

4 Menetelmät ylläpidettävän lähdekoodin arvioimiseen

Tässä luvussa tutustutaan tarkemmin muun muassa siihen, miten teknistä velkaa voidaan havainnoida lähdekoodista ja miten sitä voidaan mitata. Mittaaminen on yksi harvemmin suoritetuista poikkeavien tekniseen velkaan liittyvistä toiminnoista. Toisaalta pelkästään kattavallakaan mittaamisella ei voida saavuttaa kokonaiskuvaa teknisestä velasta ohjelmistossa, sillä sen voidaan käsittää esiintyvän myös ohjelmiston arkkitehtuurissa ja projektinhallinnassa. Lähdekoodissa esiintyneitä antisuunnittelumalleja voidaan kuitenkin analysoida automaattisilla työkaluilla. Lähdekoodia voidaan testata mittaamalla esimerkiksi staattisella analyysillä, tutkimalla koodityylin noudattamista, suorittamalla dynaamisia testejä, arvioimalla testeistä laskettavaa testikattavuutta sekä laskemalla syklomaattinen kompleksisuus. Näihin tutustutaan tarkemmin myöhemmin tässä luvussa.

Tekninen velka itsessään on hyvin abstraktia. Tästä syystä sitä on lähes mahdotonta tunnistaa automaattisesti koodista tuntematta, miten järjestelmä tarkalleen toimii. Teknisen velan oireiden ja teknisestä velasta johtuvien antisuunnittelumallien tunnistamiseksi on kuitenkin kehitetty automaattisia työkaluja. Erityyppiset automaattiset teknisen velan tunnistamistyökalut, kuten staattiset koodianalyysityökalut ja koodityylisäännöstöt havaitsevat hyvin erityyppisiä teknisen velan oireita ja antisuunnittelumalleja. Siksi onkin oleellista käyttää useita eri mittareita ja työkaluja teknisen velan kartoittamiseksi. Käytettävien työkalujen ja mittareiden lisäksi on hyvä hyödyntää myös ohjelmoijien tietoa järjestelmän toiminnasta, joka on merkittävässä roolissa teknistä velkaa hahmotettaessa. [Zazworka et al. 2013]

Teknisen velan seurantaan ja mittaamiseen voidaan käyttää projektihallintatyökaluja kuten Jiraa tai teknisen velan oireiden mittaamiseen tehtyä staattisen analyysin työkaluja kuten SonarQubea [Yli-Huumo et al. 2015]. Teknistä velkaa, joka ilmenee ohjelmistoarkkitehtuurissa, ohjelman rakenteessa tai lähdekoodin ulkopuolisessa osassa ohjelmistoa, ei voida mitata automaattisesti staattisen analyysin työkaluilla.

4.1 Staattinen koodianalyysi

Tässä kohdassa esitellään staattinen koodianalyysi lähdekoodin analysoimisen työkaluna. Staattinen koodianalyysi tarkoittaa koodin analysointia ilman sen suorittamista. Staattinen koodianalyysi on tehokas menetelmä ohjelmiston laadun ja turvallisuuden varmistamiseksi, sillä staattinen analyysi voidaan suorittaa automaattisesti esimerkiksi osana tuotantoonvientiputkea. Staattisessa analyysissa käytetyt säännöt perustuvat muun muassa olemassa oleviin hyviksi koettuihin formaaleihin ohjelmointikäytäntöihin sekä tiedon käytön ja siirron oikeellisuuden varmistamiseen. Staattista koodianalyysia voi-

daan suorittaa muun muassa tutkimalla tiedon siirtymistä tai arvioimalla syötteen turvallisuutta. Staattinen koodianalyysi voidaan helposti suorittaa myös tuotantojärjestelmästä eroavalla laitteistolla, sillä se ajetaan suorittamatta lähdekoodia. [Wichmann et al. 1995]

Tiedon siirtymistä voidaan arvioida tulkitsemalla tiedon tyypitystä eri vaiheissa ohjelmakoodia. Tutkimalla, millaisessa muodossa tietoa käsitellään erilaisissa ohjelmakoodista muodostuneista poluista, voidaan arvioida ohjelman ajonaikaista turvallisuutta. Dynaamisesti tyypitettyjen kielten, kuten PHP:n tarkastelu staattisella koodianalyysillä on hieman haastavampaa kuin staattisesti tyypitettyjen kielten, sillä tiedon tyyppiä ei välttämättä ole määritelty riittävällä tarkkuudella. Staattisella koodianalyysillä varmistetaan syötteiden oikeellisuus tiedon tyypitystä tarkastelemalla. [Wögerer 2005]

Syötteen turvallisuutta voidaan arvioida tutkimalla käyttäjälähtöisen syötteen kulkemista ohjelman lähdekoodista muodostetun puun eri poluilla. Käyttäjälähtöisen syötteen siistiminen eli sanitointi on eräs tapa varmistua syötteen turvallisuudesta ja oikeasta muodosta. Ilman käyttäjäsyötteen sanitointia ohjelmistoon voi muodostua esimerkiksi SQL-injektion mahdollistavia tietoturva-aukkoja. SQL-injektio on hyökkäys, jossa tietokantakyselyyn lisätään siihen kuulumattomia komentoja. Tietokantakyselyyn kuulumattomilla komennoilla voidaan esimerkiksi pyyhkiä tietokantatauluja. Staattinen koodianalyysi voi ennakkoon varoittaa puutteellisesta sanitoinnista johtuvista tietoturva-aukoista. [Wögerer 2005]

Leksisessä analyysissä (lexical analysis) lähdekoodin osat tokenisoidaan eli lähdekoodin eri sanat tulkitaan kielen määritelmän mukaisiksi loogisiksi osasiksi. Leksisellä analyysillä voidaan varmistaa koodin syntaksin oikeellisuus [Sotirov 2005]. Varmistamalla syntaksin oikeellisuus leksisellä analyysillä voidaan varmistaa, että ohjelma voidaan kääntää tai tulkattavan ohjelmointikielen tapauksessa ohjelman suoritus ei pääty syntaksivirheeseen. Koodiesimerkissä 1 on yksinkertainen PHP-lauseke, jossa asetetaan muuttujaan merkkijono. Merkkijonoksi tässä esimerkissä on valittu sana ”ananas”. Esimerkissä on PHP-koodikielen määrittelyn mukaiset alku- ja loppumerkit sekä merkkijonon muuttujaan asetettava lauseke. Tämän lausekkeen syntaksin oikeellisuuden voi tarkistaa tokenisoimalla lausekkeen ja tulkitsemalla tokenisoidusta syötteestä koodin oikeellisuus. Koodiesimerkissä 2 on koodiesimerkin 1 tokenisoitu vastine. Tarkastelemalla koodiesimerkkiä 2 voidaan havaita, annetun lauseke on PHP-kielen sääntöjen mukaan kieliopillisesti oikein, sillä lausekkeesta muodostuva syntaksipuu on laillinen.

```
<?php $name = "ananas"; ?>
```

Koodiesimerkki 1: PHP-lauseke, merkkijonon asettaminen muuttujaan.

```
T_OPEN_TAG T_VARIABLE =
T_CONSTANT_ENCAPSED_STRING ; T_CLOSE_TAG
```

Koodiesimerkki 2: Tokenisoitu PHP-lauseke, merkkijonon asettaminen muuttujaan.

Staattisen analyysin ominaisuuksia ovat luonto (nature) ja syvyys (depth). Staattisen analyysin tapauksessa luonto tarkoittaa ohjelmakoodin siirrettävyyttä tai sen oikeellisuutta suhteessa ohjelman määrittelyihin. Syvyydellä tarkoitetaan analyysin syvyyttä eli esimerkiksi että ohjelman totuusarvon tarkastelu voidaan tehdä hyvinkin tarkasti. Syvyys ei ole toisaalta vain yksiulotteinen arvo, sillä matemaattinen oikeellisuus ei takaa ylläpidettävyyttä tai tehokkuutta. [Wichmann et al. 1995]

4.2 Koodityylin tarkistus

Koodityyliseikoista on keskusteltu jo 1970-luvulla, modernin ohjelmoinnin alkuaikoina. Tällöin havahduttiin siihen, että hyvä lähdekoodi ei välttämättä ole synonyymi tiiviisti kirjoitetulle, verboosille tai ilman GOTO-kutsuja kirjoitetulle lähdekoodille. Esimerkiksi kirjoittamalla tiivistä lähdekoodia on pyritty vähentämään lähdekoodin vaatimaa tilaa massamuistissa. Tällainen tavoite ei välttämättä edistä lähdekoodin ymmärrettävyyttä tai ylläpidettävyyttä. DRY-periaatteella (Don’t Repeat Yourself), jolla voi välttää ylimääräisen koodin kirjoittamista, ei vielä itsessään ole tiivistä koodia. 1970-luvulla kirjoitettu tiivis koodi saattoi tarkoittaa lähdekoodia, jossa muuttujien ja funktioiden nimet voivat olla lyhennetty tilan säästämiseksi. [Kernighan ja Plauger 1974]

Eri lähdekoodikielille on sittemmin sovittu erilaisia koodityylejä. Koodityyli on formaalisti määritelty joukko koodin ulkoasua määrittäviä koodityylisääntöjä. Koodityylisääntöjen perusteella on voitu kehittää koodityylin tarkistajia (linter). Ylläpidettäessä ohjelmiston lähdekoodissa yhtenäistä koodityyliä lähdekoodin luettavuus ja ylläpidettävyys paranevat suhteessa vapaamuotoiseen lähdekoodiin. Tämän lisäksi eri lähdekoodin kirjoittajien tuotokset näyttävät keskenään yhtenäisemmiltä, jolloin koodin ymmärrettävyys säilyy ja uudet koodimuutokset sisältävät vain ohjelmalogiikkaa. Yhtenäistä koodityyliä noudattamalla ohjelmoijat eivät tee lähdekoodiin muutosehdotuksia liittyen toisten ohjelmoijien tekemiin koodityylivalintoihin. Yhtenäisen koodityylin noudattaminen on myös havaittu nopeuttavan muutosehdotusten hyväksymistä avoimen lähdekoodin projekteissa [Zou et al. 2019].

Koodityylin tarkastus kattaa myös osittain siistin lähdekoodin periaatteita. Siisti lähdekoodi on lähdekoodia, joka on visuaalisesti miellyttävää. Visuaalinen miellyttävyys on projektin sisäinen sopimuskysymys, mutta usein tällaisessa lähdekoodissa esimerkiksi rivivälit, sisennykset ja kommenttiviestit ovat jäsennelty säännönmukaisesti. Koodiesimerkeissä 3 ja 4 esitellään luokat, joiden nimi on ”StatsKeeper”. Koodiesimerkeillä

esitellyt versiot ovat teknisiltä määrityksiltään identtiset, mutta ne eroavat koodin ilmaisutyyliltään ja kommenttien asemoinniltaan. Koodiesimerkki 3 on huomattavasti hitaammin ymmärrettävissä, sillä siinä ei ole noudatettu yhdenmukaista sisennystä, rivitystä ja kommentointia. Koodiesimerkki 4 on helpommin ymmärrettävissä, sillä siinä sisennykset ovat yhdenmukaiset ja kommentit ovat jäsennelty lähdekoodin yhteyteen järkevästi. Koodityyli kattaa myös koodin yhteyteen kirjoitetut kommentit [Boswell ja Foucher 2011]. Koodiesimerkkien avulla on havainnollistettavissa, että noudattamalla yhtenäistä koodityyliä ja hyödyntämällä koodityylitarkistusta voidaan edistää koodin luettavuutta ja ylläpidettävyyttä.

```
class StatsKeeper {
public:
// A class for keeping track of a series of doubles
   void Add(double d);  // and methods for quick statistics about them
  private:   int count;        /* how many so    far
*/ public:
        double Average();

private:   double minimum;
list<double>
  past_items
      ;double maximum;
};
```

Koodiesimerkki 3. Vaikeammin ymmärrettävissä oleva koodi [Boswell ja Foucher 2011].

```
// A class for keeping track of a series of doubles
// and methods for quick statistics about them.
class StatsKeeper {
  public:
    void Add(double d);
    double Average();

  private:
    list<double> past_items;
    int count;  // how many so far

    double minimum;
    double maximum;
};
```

Koodiesimerkki 4. Ymmärrettävämpi koodi [Boswell ja Foucher 2011].

4.3 Dynaaminen koodianalyysi ja testikattavuus

Tässä kohdassa esitellään testikattavuuden eri määritelmiä ja pohditaan testikattavuuden hyödyllisyyttä lähdekoodin tutkimisessa. Testikattavuus on mittari, jolla voidaan ilmaista, kuinka suuri osuus lähdekoodista suoritetaan ennalta määrättyjen testien aikana. Testikattavuudelle on erilaisia määritelmiä, jotka vaikuttavat siihen, miten testikattavuutta voidaan tulkita. Testatuiksi määriteltyjen koodirivien joukon osuus koko lähdekoodista ilmaisee testien kattavuutta. Testien kattavuudella voidaan arvioida, kuinka hyvin ohjelmiston toimivuus tunnetaan erilaisissa tilanteissa. Testimäärittelyjen perusteella voidaan alustaa erilaiset tilanteet, joita dynaamisilla testeillä halutaan kattaa.

Dynaamisessa koodianalyysissa eli dynaamisessa testauksessa ohjelmiston tai sen osien toimintaa testataan suorittamalla testattavia osia testimääritysten mukaisesti fyysisellä tai virtuaalisella suorittimella [Myers 1979]. Testeissä ohjelmiston tila alustetaan jokaisella suorituskerralla testimäärittelyjen mukaisesti. Ohjelmiston palauttamien arvojen ja lopputilan perusteella voidaan arvioida ohjelmiston suorituksen oikeellisuutta. Dynaamiset testit voidaan jakaa esimerkiksi yksikkö-, ominaisuus- ja integraatiotesteihin testien laajuuden ja testatun osuuden perusteella. Dynaamisen koodianalyysin mahdollistamiseksi voidaan joutua muuttamaan lähdekoodin toteutusta tai rakentaa esimerkiksi testausympäristö testien suorittamiseksi. Dynaamisesta testaamisesta voidaan laskea testien kattavuus, joka on testien onnistumisesta erillinen mittaustulos. Tämän lisäksi testikattavuuden ja syklomaattisen kompleksisuuden avulla laskettavaan CRAP-arvoon tutustutaan kohdassa 4.5.

Testit ovat merkittävä osa ohjelmistoprojektia, sillä ne mahdollistavat lähdekoodin laadun parantamisen ja sen toiminnan testaamisen. Korkea testikattavuus ei välttämättä korreloi lähdekoodin luotettavuuden kanssa kaupallisissa suljetun lähdekoodin projekteissa [Mockus et al. 2009] tai avoimen lähdekoodin projekteissa [Kochhar et al. 2017]. Toisaalta kasvattamalla lähdekoodin testikattavuutta voidaan kuitenkin vähentää hieman lähdekoodissa esiintyviä virheitä [Antinayn et al. 2018]. Testikattavuus parantaa luottamusta lähdekoodin ajonaikaiseen vakauteen ainakin siten, että testeillä katetut osat koodia voidaan suorittaa ilman virheitä testimäärittelyjä vastaavissa olosuhteissa. Testien kattamaa koodia on myös helpompi refaktoroida kuin testaamatonta koodia, sillä testimäärittelyjen mukaiset toiminnot säilyvät muutosten jälkeen.

Testikattavuudelle on useita eri kriteerejä. Testikattavuus voidaan määritellä muun muassa suoritettavien lausekkeiden mukaan tai suoritusreitin perusteella. Suoritettavien lausekkeiden perusteella laskettavassa testikattavuudessa lähdekoodin kattavuutta tutkitaan asettamalla kattavuuden kriteeriksi eri lausekkeiden suorittaminen. Suoritusreitin perusteella laskettavassa testikattavuudessa määritellään testatuksi, jos dynaamisia testejä ajettaessa ohjelman suoritus kohtaa koodirivin, joka suoritetaan. Koodiesimerkissä 5 on funktio, joka palauttaa merkkijonon. Koodiesimerkin 5 lähdekoodin testikattavuus

olisi merkittävästi erilainen lauseke ja suoritusreittiperusteisella testikattavuudella laskettuna. Käytettäessä testijoukkoa, joka suorittaa esimerkin koodipätkän, suoritusreittien mukaan laskettuna esimerkin testikattavuus olisi 100%, sillä kaikki ohjelmarivit tulevat suoritetuiksi. Suoritettavien lausekkeiden perusteella laskettavassa testikattavuudessa testikattavuus olisi alle 100%, sillä objektin $bar metodi exampleMethod() ei tule suoritetuksi, sillä totuusarvolausekkeen arvo on tosi jo ensimmäisellä ehdollaan ja sen suorittaminen päättyy [PHP 2020]. Vaikka testikattavuuden eri kriteerit tuottavatkin hieman erilaisia tuloksia, ne vastaavat toisiaan hyvin paljon ja ovat sikäli vertailukelpoiset [Antinyan et al. 2018].

```
public function fooBar(): ?string
{
        $foo = $this→extraCondition();
        if ( $foo || $this→exampleMethod()) {
                return 'a';
        }
}
```

Koodiesimerkki 5. Lähdekoodiesimerkki testikattavuudelle, PHP7.4.

Testimäärittelyt ja testikriteerit voivat vaikuttaa merkittävästi siihen, kuinka mielekkäänä testien kattavuutta voidaan pitää. Suoritusreittitasolla laskettava testikattavuus voi jättää osan merkittävästi poikkeavista tilanteista testaamatta vaikka koodirivit olisivatkin testeillä täysin kattettuja. Esimerkiksi koodiesimerkin 5 lähdekoodissa, olettaen että metodi $this→extraCondition() palauttaa testien ajon aikana aina arvon totuusarvon tosi, lähdekoodissa olevan ehtolausekkeen kaikkia lausekkeita ei testata. Tämä tarkoittaa siis sitä, että $this→exampleMethod() eri palautusarvoja ei testata eikä siitä saada laskettua kattavuutta. Testikattavuuden parantamiseksi ja lähdekoodin luotettavuuden edesauttamiseksi voidaan kirjoittaa niin sanottua testattavaa koodia, joka yleisimmillään yksinkertaisesti noudattaa aiemmin mainittuja hyviä koodityylikäytäntöjä.

Kattavat testit, jotka voidaan suorittaa ilman virheitä kuitenkin tarkoittavat vähintään sitä, että kaikki testeillä katettu ohjelmakoodi voidaan ainakin jollain tasolla suorittaa ilman, että ohjelman suorituksessa esiintyy odottamattomia poikkeuksia. Testien varsinainen kattavuus muodostuu siitä, mitä testeissä varsinaisesti varmistetaan. Sillä ohjelmistotestien kirjoittaminen vie aikaa muulta kehitystyöltä, ohjelmistotesteillä katettua sekä uusilla testeillä katettavaa koodia on myös tärkeää priorisoida. Koodiin jäävät testikattavuuden katvealueet voivat tarkoittaa odottamattomia vaikutuksia ohjelman suorituksen aikana. Erityisesti lähdekoodille, jota muokataan usein ja jonka merkitys lii-

ketoiminnalle on tärkeä, tulisi olla testattu. Tätä voidaan edistää esimerkiksi asettamalla yhdeksi projektinhallinnan tavoitteista testien kattavuus kriittisten ominaisuuksien osalta. Testikattavuuden kytkeminen projektin tavoitteisiin voi kuitenkin aiheuttaa tilanteita, joissa testien kirjoittajat maksimoivat tavoitteiden täyttämisen kirjoittamalla mahdollisimman monta testiä samasta asiasta tai kattamalla mahdollisimman paljon koodirivejä kevyillä testeillä. Testikattavuudestakaan ei kuitenkaan voida tehdä suoria päätelmiä koodin laadusta, sillä testikattavuus ei itsessään ole merkki virhetilanteiden tunnistamisesta [Inozemtseva ja Holmes 2014].

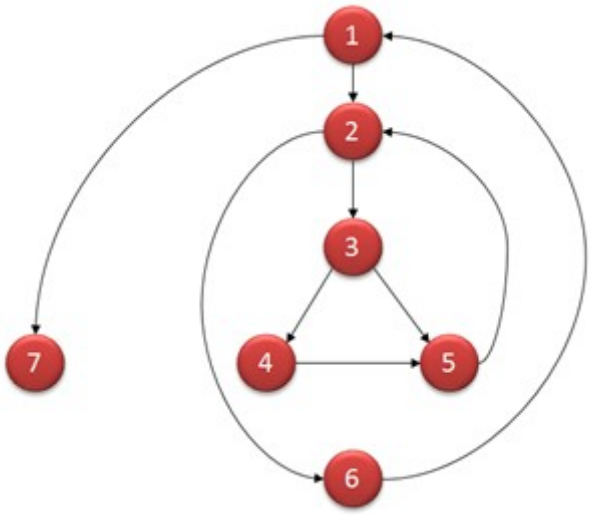
4.4 Syklomaattinen kompleksisuus

Syklomaattinen kompleksisuus (Cyclomatic Complexity) on ohjelmistokoodista laskettava arvo, joka kuvailee ohjelmistokoodin vaikeaselkoisuutta esimerkiksi luokka- ja funktiotasolla. Se korreloi koodin muovattavuuden kanssa.

Syklomaattisella kompleksisuudella voidaan arvioida laskemalla ohjelman suorittamisen eri polkujen lukumäärä. Ohjelman suorituspolkujen lukumäärä ja syklomaattisen kompleksisuuden arvo on määritelty kaavalla

$$M = E - N + 2P$$

jossa P ilmaisee kaikki erilliset lopputilat ohjelman suorituksessa ja E on ohjelmassa esiintyvät reunat, joissa hallinta siirretään muualle lähdekoodissa. Syklomaattisen kompleksisuuden kaavassa N on joukko, joka sisältää ohjelman suorituksen kaikki eri tilat. Kuvassa 4 on piirretty erään ohjelman suorittamisesta vuokaavio. Kuvassa 4 syklomaattisen kompleksisuuden kaavan mukaiset reunat E ovat ilmaistu nuolina, jotka ilmaisevat siirtymistä tilasta toiseen. Tilat, jotka vastaavat joukkoa N ovat kuvattu kuvassa 9 punaisina palloina. Kuvassa 4 lopputila P on pallo numero 7. Syklomaattisen kompleksisuuden laskukaavan perusteella voidaan päätellä, että kyseisen ohjelman syklomaattinen kompleksisuus on 9-7+2 = 4.



Kuva 4. Ohjelman suorittamisen vuokaavio [Guru99 2020].

Syklomaattinen kompleksisuus on silmämääräisesti arvioitavissa lähdekoodista, sillä useat sisäkkäiset silmukat sekä ehtolausekkeet erottuvat selkeästi visuaalisesti muusta lähdekoodista. Korkea syklomaattinen kompleksisuus ei välttämättä tarkoita, että lähdekoodi olisi huonoa tai toimisi väärin, mutta johtuen lähdekoodin monimutkaisesta, sisäkkäisestä rakenteesta sen ylläpitäminen voi olla haastavampaa kuin helpommin luettavan lähdekoodin. Toisaalta alan tutkimuksissa on löydetty korrelaatio korkean syklomaattisen kompleksisuuden ja virheiden määrän välillä [Schroeder 1999]. Siksi ohjelmistoprojekteissa tulisi tavoitella matalaa syklomaattista kompleksisuutta.

4.5 CRAP-arvo

Tässä luvussa esitellään edellisessä luvussa esitellyn syklomaattisen kompleksisuuden tueksi määritelty CRAP-arvo (*Change Risk Anti-Patterns*). CRAP-arvo ilmaisee koodiin tehtävien muutosten muodostamaa riskiä muodostaa uusia vikoja. Korkea koodikomplesisuus korreloi ylläpidossa kohdattavien vikojen kanssa. Tämän lisäksi automaattisten testien puute kasvattaa muutosten aiheuttamia riskejä. CRAP-arvo ei suoraan mittaa koodin ymmärrettävyyttä. On kuitenkin perusteltavissa, että koodi, johon tehtävät muutokset eivät aiheuta vikoja, on ymmärrettävää.

Korkea CRAP-arvo ilmaisee, että lähdekoodilla on korkea syklomaattinen kompleksisuus ja siihen kirjoitetuilla testeillä on puutteellinen kattavuus. CRAP-arvo määritellään siten, että ohjelmakoodille lasketusta syklomaattista kompleksisuudesta vähennetään testien kattavuus. CRAP -arvo voidaan laskea kaavalla

```
CRAP(m) = comp(m)^2 * (1 - cov(m)/100)^3 + comp(m)
```

jossa m on tutkittava metodi, comp(m) on tutkittavan metodin syklomaattinen kompleksisuus ja cov(m) on tutkittavan metodin testikattavuus [Savoia ja Evans 2007]. Testikattavuuden määritelmä CRAP-arvoa laskettaessa on yleisesti reitti kattavuus (path coverage), sillä syklomaattinen kompleksisuus lasketaan koodiin kirjoitetuista suoritusreiteistä. Kuten syklomaattinen kompleksisuus, korkea CRAP-arvo voi olla helposti havaittavissa lähdekoodista, sillä useat sisäkkäiset silmukat sekä ehtolausekkeet eli eri reitit koodissa selkeästi erottuvat visuaalisesti lähdekoodista.

Korkea CRAP-arvo ei automaattisesti merkitse, että koodi laadultaan olisi hyvää tai huonoa, eikä myöskään kerro ohjelmistossa esiintyvien virheiden määrää. Toisaalta, kuten edellisessä kohdassa esitellyn syklomaattisen kompleksisuuden suhteen, lähdekoodin laatuun panostaminen laskee CRAP-arvoa. CRAP-arvon seuraaminen auttaa myös hahmottamaan kuinka hyvin koodiin voidaan luottaa, sillä CRAP-arvossa otetaan huomioon testikattavuus. Päättelemällä korkeiden CRAP-arvojen perusteella ei kuitenkaan voida löytää koodista varsinaisia riskipaikkoja, sillä korkeat CRAP-arvot voivat keskittyä osiin lähdekoodia, jota ei käytetä aktiivisesti. Tämän lisäksi CRAP-arvo ei ota kantaa siihen, kuinka tarkasti lähdekoodia on oikeasti testattu. [Savoia ja Evans 2007]

4.6 Manuaalinen testaaminen

Manuaalinen testaaminen tarkoittaa sitä, että testaaja suorittaa järjestelmän toimintojen testaamista syöttämällä tiedot sovelluksen loppukäyttäjälle tarjottavan käyttöliittymän kautta ilman, että automaattinen työkalu suorittaa toimintoja. Manuaalisessa, kuten automaattisessakin testaamisessa testitapaukset on ennalta määritelty. Automaattinen testaaminen auttaa estämään virheiden muodostumista, mutta toisaalta manuaalisella testaamisella voidaan löytää olemassa olevia mutta tuntemattomia tai ennalta odottamattomia virheitä [Ramler ja Wolfmaier 2006]. Manuaalista testaamista onkin hyödyllistä käyttää esimerkiksi julkaistaessa uusia toiminnallisuuksia, jotta varmistetaan, että sovelluksen uusimpaan versio toimii odotetulla tavalla ennen sen julkaisemista.

Testaaja koostaa testien tuloksista testiraportin. Testitapauksissa on ilmaistu testin alkutila, testissä tehtävät välivaiheet ja testin lopputila. Manuaalisella testaamisella voidaan myös varmistaa, että ohjelmisto toimii odotetulla tavalla erilaisilla laitteistoilla. Tämä menetelmä ei varsinaisesti mittaa lähdekoodin ominaisuuksia, mutta sen avulla voidaan pienillä alkuinvestoinneilla saada kattava ymmärrys ja luottamus ohjelmiston toiminnan oikeellisuudesta. Ohjelmiston monimutkaisuuden kasvaessa erilaisten lähtötilanteiden ja lopputilanteiden toistamiseksi käytettävä työmäärä voi kuitenkin kasvaa kestämättömäksi. [Taipale et al. 2011]

5 Tutkittava projekti

Tässä luvussa esitellään tutkielmassa analysoitava ohjelmistoprojekti sekä työkaluja, joilla voidaan arvioida teknisen velan määrää ohjelmistoprojektissa käytetyn Laravel-kehityskehyksen konteksissa. Esitellyt avoimen lähdekoodin työkalut tarjoavat PHP-kielisten ohjelmistoprojektien staattisen analyysin, lähdekoodityylitarkistukset, dynaaminen testauksen sekä testikattavuuden laskemisen.

5.1 Tutkittava projekti

Tutkittavana aineistona käytetään Mousewell Oy:n kehittämää pienille ja keskisuurille yrityksille suunnattua kirjanpito-ohjelmistoa, joka on toteutettu Laravel-kehityskehyksellä. Ohjelmistoa tarjotaan kolmen eri brandin alla. Sovelluksen elinkaari alkoi palveluna, joka tarjosi yrittäjille mahdollisuuden muodostaa laskuja yksinkertaisesti verkkosivuilta käytettävän käyttöliittymän avulla.

Ohjelmiston kehityksessä on noudatettu Minimum Viable Product -mallia. Minimum Viable Product -mallissa toimitaan siten, että ennen kuin tiedetään tarkalleen, millaiselle tuotteelle on tarve, luodaan mahdollisimman pienellä panostuksella yksi toiminnallisuus mahdollisimman helpoksi käyttää. Kehitysmalli on ohjannut tuotteen kehitystä sen elinkaaren alusta, vuodesta 2017. Kehityksen aikana ohjelmistokoodin sisäiselle laadulle ei ole asetettu mitään konkreettisia rajoja. Tämä johti tilanteeseen, jossa modernien verkkosivuominaisuuksien käyttöönotto muuttui lähes mahdottomaksi. Tällaisessa vaiheessa kerrytetty tekninen velka oli muodostunut niin suureksi, että saavuttaakseen uusia tavoitteita, osat vanhasta lähdekoodista tulisi kirjoittaa uudelleen.

Ohjelmiston alkuperäinen toteutus noudatti MVC (Model View Controller) arkkitehtuurimallia. Uudelleenkirjoitusprojektissa asiakasrajapinnassa tarjottava käyttöliittymätoteutus korvattiin uudella. Uudistettu käyttöliittymä noudattaa SPA (Single Page Application) mallia, joka käyttää MWCloud-tuotteelle kehitettyä uutta ohjelmistorajapintaa. Uusi ohjelmistorajapinta on totetettu RESTful API (Representational state transfer -tyyppinen Application Programming Interface) arkkitehtuurimallia. SPA arkkitehtuurimallilla toteutetulla käyttöliittymällä saavutettiin responsiivisempi käyttäjäkokemus, sillä siirtymät sivulta toiselle näyttäytyvät sulavampina. Uudelleenkirjoitetussa ohjelmistossa uusi sivu voidaan esittää käyttäjälle lähes välittömästi ja siinä esitettävä sisältö ladataan sen valmistuessa taustajärjestelmässä.

Tässä tutkimuksessa ei oteta kantaa siihen, kuinka paljon uutta teknistä velkaa on kerrytetty uuden käyttöliittymän kehityksessä, sillä korvausprojektissa tuotettu uusi käyttöliittymä on itsessään erillinen projekti. Tutkimuksessa tehdyissä koodiin kohdistu-

viin mittauksiin ei ole sisällytetty uuden käyttöliittymän lähdekoodin tuloksia. Ottaen huomioon ohjelmistokehityksen nopean luonteen ja tarjotun ohjelmistorajapinnan kehityksen, voidaan olettaa, että käyttöliittymän lähdekoodissa on jokin määrä teknistä velkaa.

Tutkielman empiirisen osuuden lisäksi suoritetaan vertailu olemassa olevan sovelluksen muutosprojektin eri vaiheista. Tutkimuskohteena on verkkosovellus, jonka kehitys on aloitettu vuonna 2017. Projektin koodisäilöön on eri vaiheissa ohjelman kehityskaarta lisännyt muutoksia yhteensä 17 kehittäjää. Projektiin aloitettiin muutostyö kesällä 2019 ja se päättyi käyttöliittymältään uudistetun sovelluksen ensimmäisen version julkaisuun syyskuussa 2019. Muutostyön ensisijainen tavoite oli luoda tuotteen loppukäyttäjille entistä nopeampi käyttökokemus säilyttäen sovelluksen kaikki sen hetkiset toiminnallisuudet siirtymällä Model-View-Controller -arkkitehtuurista Restful API -rajapintaan, jota asiakas kutsuu käyttämällä Single Page Application käyttöliittymää. Roolini muutosprojektissa oli luoda Restful API -rajapinta ja kehittää uusia toiminnallisuuksia. Vastasin myös ohjelmistoon kirjoitettavista PHPUnit-testeistä ja ohjelmiston koodityylimäärittelyistä.

Projektin ensisijainen tavoite toteutettiin aikataulussa. Ensisijaiselle tavoitteelle oli asetettu selkeät mittarit ja raja-arvot, sillä oleellista oli, että loppukäyttäjän on mahdollista suorittaa sovelluksen tarjoamat toimenpiteet annettujen määrittelyjen tasoisesti. Toissijaiselle tavoitteelle ei asetettu mitään mittareita eikä ohjelmistokehityksessä varsinaisesti tavoiteltu toissijaista tavoitetta.

Tutkimuksessa tullaan käsittelemään sovelluksen toimintalogiikkaa toteuttavaa lähdekoodia, joka muutettiin MVC-mallisesta REST-rajapintaa hyödyntäväksi. Ohjelmiston muutosprojektissa toteutetun käyttöliittymän lähdekoodi ei tule sisältymään tutkimukseen. Tämä tarkoittaa, että uuteen lähdekoodiin sisältyvä tekninen velka ja sen muodostamat riskit ei välity tutkimuksessa kokonaisuudessaan. Tutkimuksen kannalta voidaan kuitenkin todeta, muutosprojektissa lähdekoodista ei poistettu alkuperäisen käyttöliittymän koodia. Käyttöliittymän monimutkaiset toiminnallisuudet toteutettiin ohjelmoimalla uutta ja REST-rajapinta mahdollistaa ohjelman käyttämisen myös ilman erillistä verkkoselaimessa esitettävää käyttöliittymää. Tästä syystä alku- ja lopputilan tulokset ovat vertailukelpoiset.

5.2 Tutkimuksessa käytetyt työkalut

Tutkimuksessa ohjelmistoa mitattiin niillä työkaluilla, jotka ohjelmiston kehityksen aikana oli löydetty ja todettu laadukkaiksi avoimen lähdekoodin ohjelmistoiksi. Mittauksessa hyödynnettiin PHP-kielisten projektien analysointiin tarkoitettuja työkaluja.

Larastan [2020] on staattisen analyysin työkalu erityisesti Laravel-kehityskehystä käyttävien projektien staattiseen analysointiin. Larastan on laajennos PHPStan-työkalus-

ta. PHPStan on avoimen lähdekoodin työkalu, jolla voidaan suorittaa staattista analyysiä PHP-projektiin. Larastan- ja PHPStan-työkalujen testimäärittelyt tarkistavat lähdekoodista muun muassa luokkien funktioiden näkyvyyden oikeellisuuden, funktioiden parametrien tyyppejä ja määrää koskevaa oikeellisuutta sekä muuttujien olemassa oloa lähdekoodin eri suoritushaaroissa. Tämän lisäksi Larastan tarjoaa sääntöjä, jotka varmistavat, että Laravel-kehityskehyksessä käytettävän Eloquent-tiedonmallintamispaketin metodeja käytetään oikein. [PHPStan 2020; Larastan 2020]

PHP_CodeSniffer on työkalupaketti, joka tarjoaa työkalun PHP-, JavaScript- ja CSS-tiedostojen analysointiin määritellyn ohjelmointityylistandardien mukaisesti kirjoitusta koodista poikkeavan tyylin havaitsemiseksi. Paketti sisältää myös työkalun löydettyjen tyylivirheiden automaattiselle korjaamiselle. [PHP_CodeSniffer 2020]

PHP_CodeSniffer hyödyntää PHP:n sisäänrakennettua tokenisointia PHP-kielisen lähdekoodin tulkkaamiseen. Lähdekoodin tokenisoinnin jälkeen PHP_CodeSniffer tarkastaa muun muassa sulkumerkkien oikeellisuuden ja lähdekoodin oikeellisuuden käytettyyn koodityylisääntöjoukkoon nähden.

PHPUnit on kehityskehys, joka tarjoaa työkalut PHP-projektien yksikkötestaamiseen. PHPUnit tarjoaa muun muassa komentorivityökalun testien suorittamiseen, joukon ohjelman suorittamiseen kohdistuvia testimääräyksiä [PHPUnit 2020].

Laravel koodikehysympäristö tarjoaa oletuksena jaottelun PHPUnit testeille niiden laajuuden mukaan. Oletuksena dynaamiset testit jaotellaan yksikkö- (Unit) ja ominaisuus- (Feature) testeiksi. Yksikkötestit ovat yksittäisiin ohjelman osiin kohdistuvia testejä, jotka voidaan ajaa ilman koko koodikehyksen tai lähdekoodikontekstin lataamista. Ominaisuustestien avulla voidaan testata kokonaisuuksia, joilla voidaan varmistua kokonaisuuksien toimintaa [Laravel 2020b]. PHPUnitin automaattiset testit voi kytkeä osaksi kehitysprojektin prosesseja kuten kehitysehdotusten liitämistä osaksi versiohistoriaa. Laravel koodikehysympäristö ja PHPUnit kuitenkin tarjoavat mahdollisuuden järjestää testit haluamallaan tavalla, jotta voidaan rajata automaattisten testien laajuus kehityksen tai tuotantoonvientien aikana.

Testikattavuuden laskemiseksi PHP-projektista voidaan käyttää useita eri työkaluja. Työkaluista huomionarvoiset ovat Xdebug sekä PCOV. Testikattavuus PHP:ssä lasketaan useimmiten käyttämällä PHP_CodeCoverage-työkalua PHPUnit testien ajon yhteydessä. PHPUnit itsessään suorittaa testit ja delegoi testikattavuuden laskennan esimerkiksi testikattavuudesta vastaavalle työkalulle kuten esimerkiksi Xdebugille. [Xdebug 2020]

Xdebug on PHP-laajennus, joka tarjoaa useita ominaisuuksia PHP-projektien kanssa työskentelyyn esimerkiksi virhetilanteiden etsimistoiminnon ja roskatiedon poistamisen tilastointia. Testikattavuuden voi laskea myös Interactive PHP Debugger (phpdbg) -työkalulla. Se on interaktiivinen PHP-debuggaustyökalu, joka hyväksytty osaksi PHP:n va-

kio-ominaisuuksia versiossa PHP 5.6. Näiden lisäksi PHP-projektien testikattavuutta voi tutkia PCOV-työkalulla. PCOV on PHP-ajuri, joka tarjoaa aiempia PHP-testikattavuustyökaluja suorituskykyisemmän testikattavuuden laskennan.

Kaikkein tarkimman testikattavuuden PHP-projektille voi laskea Xdebug-työkalulla. Huolimatta siitä, että Interactive PHP Debugger on suorityskykyisempi kuin Xdebug ja virallinen osa PHP-pakettia, sen puutteellisesti määritellyn suoritetun lähdekoodin tunnistamisen takia se ilmoittaa testikattavuuden hieman korkeammaksi kuin Xdebug. Myös PCOV laskee testikattavuuden hieman eri tavalla kuin Xdebug. Testikattavuuden tutkimiseen nopein työkalu tästä kolmen joukosta on PCOV. [PCOV 2019]

Ohjelmiston syklomaattisen kompleksisuuden arviointiin voi käyttää myös PHPUnit työkalua, joka pystyy tarjoamaan lisäksi CRAP-arvon jokaiselle yksittäiselle ohjelman tiedostolle, luokalle ja metodille. PHPUnitin tarjoama kooste sovelluksen syklomaattisesta kompleksisuudesta ja CRAP-arvosta ei kuitenkaan anna koko ohjelmiston kattavaa keskiarvoa.

PhpMetrics on staattisen analyysin työkalu, joka tarjoaa kattavan kokonaisuuden erilaisia mittaustuloksia PHP-kielisistä ohjelmistoista. PhpMetrics mittaa muun muassa ohjelman syklomaattisen kompleksisuuden ja rivimäärän. PhpMetrics antaa mittaustulokset visuaalisena rapottina. PhpMetrics tuottaa myös muita mielenkiintoisia arvoja lähdekoodista. Näistä mainittakoon *ylläpidettävyysindeksi* (maintainablity index) ja *Halsteadin vaikeus* (Halstead' difficulty). PhpMetrics on staattisen analyysin työkalu ja sen avulla voidaan mitata ohjelmistoa suorittamatta mitattavan ohjelmiston koodia. [PhpMetrics 2020]

6 Mittaustulokset

Tässä luvussa tarkastellaan tutkittavan ohjelmiston lähdekoodin laatua analysoimalla automaattisilla työkaluilla tehtyjen testien tuloksia. Ensimmäisenä käsitellään lähdekoodin koodirivimäärää. Tämän jälkeen siirrytään käsittelemään havaittujen staattisen analyysin ilmaisemia ongelmien määrää. Kohdassa 6.3 koostetaan koodityylisääntöpoikkeamien esiintymiskerrat ohjelmiston eri versioissa. Neljännessä kohdassa esitellään tutkimuksessa saadut testikattavuuden tulokset. Viidennessä kohdassa esitellään syklomaattisen kompleksisuuden kehittyminen eri versioiden välillä. Kuudennessa kohdassa pohditaan tuloksien merkitystä.

Automaattisista testeistä saadut tulokset ilmaisevat koodiin kohdistuneen muutoksen suuruusluokkaa ja muutoksilla tuotetun koodin objektiivista laatua. Tuloksista arvioidaan valittujen ohjelmointikäytäntöjen onnistumista suhteessa asetettuihin tavoitteisiin.

Analyysi toteutettiin PhpMetrics-, Larastan-, PHP_CodeSniffer- sekä PHPunit-ohjelmilla. PhpMetrics valittiin, sillä se tarjoaa ohjelmiston koodirivien määrän sekä syklomaattisen kompleksisuuden arvon. Larastan-ohjelmalla mitattiin staattisen analyysin avulla tarkasteltavat ohjelman poikkeamat. PHP_CodeSniffer-koodityylitarkastus suoritettiin ohjelman tarjoamilla oletusasetuksilla. Mittaustulokset PHPUnit-testeistä koostettiin käyttäen PCOV-testikattavuusajuria. PCOV-testiajuri valittiin testien ajamiseen, sillä se on testiajureista nopein ja testikattavuuden laskemismenetelmällä ei ole varsinaisesti merkitystä tulosten kannalta [Antinyan et al. 2018]. Ohjelmaa tutkittiin sen versioista v0.1.0, v1.0.0, v1.5.0 sekä 2020-03-25-a. Tutkimus on rajattu näihin versioihin, sillä v0.1.0 on ensimmäinen, johon on lisätty PHPUnit-testit. Ohjelmiston ensimmäinen tutkimukseen valittu versio v0.1.0 julkaistiin 9.8.2019. Tutkittu versio v1.0.0 on ensimmäinen tuotantoon julkaistu versio uudesta tuotteesta ja se julkaistiin 18.9.2019. Kolmas tutkittu versio on v1.5.0 joka julkaistiin 4.12.2019 ja se ajoittuu tuotteen toiseen julkaisuun. Ohjelman tutkimukseen valituista versioista tuorein on 2020-03-25-a. Ohjelman PHPUnit-testiseteistä rajattiin mittausten yhteydessä kaikista versioista yksi testi, jota ei voitu suorittaa käytetyllä ympäristömuuttuja-arvoilla.

6.1 PhpMetricsillä laskettu ohjelmiston koko

Ohjelman lähdekoodin kokoa arvioidaan ohjelmiston logiikkaa määrittävän lähdekoodin rivimäärällä. Lähdekoodiriveiksi lasketaan tyhjät sekä kommenttirivit. Taulukossa 1 on kuvattuna lähdekoodin rivien määrä ohjelmiston eri versioissa. Ensimmäisessä analysoidussa versiossa v0.1.0 lähdekoodin rivimäärä on 55807. Taulukosta voidaan havaita, että kussakin uudessa julkaistussa versiossa koodirivien määrä kasvaa noin 5000-28000 rivillä. Koodirivien lukumäärä ohjelman lähdekoodissa kasvoi eniten versioon v1.0.0

siirtyessä, yhteensä 28147. Tämän version julkaisuun liittyi uuden ohjelmistorajapinnan käyttöönotto. Ohjelmiston koko kasvoi koodiriveillä laskettaen vähiten siirryttäessä versiosta v1.5.0 versioon 2020-03-25-a, jolloin koodirivien lukumäärä kasvoi yhteensä 4817 rivillä.

Versio	Rivien määrä
v0.1.0	55807
v1.0.0	83954
v1.5.0	89374
2020-03-25-a	94191

Taulukko 1. PhpMetrics, lähdekoodin laajuus.

6.2 Larastan-työkalun tulokset

Ohjelman tutkimukseen valittuja versioiden teknistä laatua analysoitiin Larastan-työkalulla, työkalun tarjoamilla oletusasetuksilla. Taulukossa 2 on Larastan-työkalulla mitattuejn poikkeamien lukumäärä ohjelmiston eri versioissa. Ensimmäisessä analysoidussa versiossa v0.1.0 havaittiin yhteensä 582 poikkeamaa. Merkittävin määrä uusia poikkeamia ohjelman lähdekoodiin tuli siirryttäessä versiosta v0.1.0 versioon v1.0.0 kun Larastan-työkalulla havaittujen poikkeamien määrä kasvoi 165 poikkeamalla. Muiden versioiden välillä havaittujen poikkeamien määrä kasvoi vain noin 10 poikkeamalla. Vaikka projektia toteutettaessa on pyritty noudattamaan hyviä ohjelmointikäytäntöjä, poikkeamien määrä on kasvanut versiosta toiseen. Huomion arvoista on, että ohjelmaa kehitettäessä ei ole aktiivisesti seurattu staattisella analyysillä havaittavia poikkeamia.

Versio	Poikkeamat
v0.1.0	582
v1.0.0	747
v1.5.0	740
2020-03-25-a	750

Taulukko 2. Larastan-työkalulla havaitut poikkeamat.

6.3 PHP_CodeSniffer tulokset

Ohjelman tutkittujen versioiden PHP-koodityylipoikkeamia tutkittiin PHP_CodeSniffer-työkalulla sen oletusasetuksilla. Taulukossa 3 on PHP_CodeSniffer-työkalulla havaitut koodityylivirheiden lukumäärät ohjelmiston eri versioissa. Taulukossa 3 olevista tuloksista voidaan havaita, että koodityylivirheiksi määriteltyjen poikkeamien lukumäärä on kasvanut lähdekoodin määrän kasvaessa. Poikkeuksen tekee viimeisin ohjelmistonversio. Versioiden v1.5.0 ja 2020-03-25-a ohjelmiston lähdekoodiin lisättiin merkittävä koodityylimuutoksia sisältävä päivitys, joka vähensi koodityylivirheiksi laskettavien poikkeamien määrää lähdekoodissa. Kyseinen koodityylimuutospäivitys ei kuitenkaan toteuta samoja koodityylimäärityksiä kuin PHP_CodeSnifferin perusasetukset ja siksi tyylivirheiden kokonaismäärä ei ole nolla.

Versio	Tyylivirheiden lukumäärä
v0.1.0	20271
v1.0.0	27934
v1.5.0	29511
2020-03-25-a	23611

Taulukko 3. PHP_CodeSniffer tulokset koodityylipoikkeamista.

6.4 PCOV-testikattavuus

Tutkimuksessa ohjelmiston eri versioiden testikattavuutta tutkittiin ajamalla testisetit versiossa määritellyillä testausasetuksilla. Ohjelman kehityksen aikana suoritettavien testien kattavuuslaskennan määrityksiin on tehty muutoksia, ja tulokset eri versioiden välillä eivät ole täysin vertailukelpoiset keskenään. Versiossa v0.1.0 suoritettava ensimmäinen testi testaa vain yhtä ominaisuutta ohjelmistosta. Tämän lisäksi version testikattavuuslaskentamäärittelyt eivät ole yhteensopivat PCOV:n kanssa johtuen eräästä lähdekoodissa käytetystä riippuvuusmäärittelystä. Versioissa v1.0.0 ja v1.5.0 riippuvuusmäärittelystä johtuva ongelma on korjattu määrittelemällä tutkittava lähdekoodi siten, että ongelmalliset tiedostot ovat rajattu tutkittavan joukon ulkopuolelle. Rajauksen lavealla määrittelyllä on kuitenkin jätetty merkittävä, tutkittavissa oleva osa ohjelmiston logiikasta testikattavuuslaskennan ulkopuolelle. Versiossa 2020-03-25-a testikattavuuslaskennan määrittelyt ovat kehittyneimmät ja vain kaikkein vaikeimmin testattavissa oleva joukko lähdekooditiedostoja on rajattu laskennan ulkopuolelle.

Taulukossa 4 on kuvattuna PCOV-työkalulla mitatut testikattavuuden tulokset. Testikattavuus on kasvanut merkittävästi ensimmäisestä julkaistusta versiosta, sillä suhteellinen kattavuus on kasvanut yli yksitoista prosenttiyksikköä. Testikattavuus on myös kasvanut suhteellisesti nopeammin kuin mitä lähdekoodin koko on kasvanut, sillä versioiden v1.0.0 ja v1.5.0 välillä kaikkien lähdekoodirivien kokonaismäärä kasvoi 6,5 %, dynaamisten testien määrittelyjen kattavien lähdekoodirivien kokonaismäärä kasvoi 3,4 %, ja testattujen lähdekoodirivien määrä kasvoi 34 %. Tämän lisäksi versiosta v1.0.0 siirryttäessä versioon v1.5.0 uusilla testeillä katettiin enemmän uusia rivejä kuin mitä testimäärittelyjen mukaisen lähdekoodin kokoa lisättiin, sillä lähdekoodirivien määrä kasvoi 668 rivillä ja testikattavuuden laajuus suureni 700 rivillä.

Versio	Testatut rivit	Lähdekoodirivit	Kattavuus
v0.1.0	0	0	N/A
v1.0.0	2044	19403	10,53%
v1.5.0	2744	20071	13,67%
2020-03-25-a	6809	30991	21,97%

Taulukko 4. PHPUnit/PCOV testauskattavuus.

6.5 PhpMetrics-työkalun syklomaattinen kompleksisuus

Ohjelmiston syklomaattisen kompleksisuuden kehittymistä eri versioiden välillä mitattiin PhpMetrics-työkalulla. Taulukossa 5 on PhpMetrics-työkalun mittaustulokset. Syklomaattisen kompleksisuuden mittariksi tutkimuksessa valittiin ohjelman lähdekoodin luokan keskimääräinen syklomaattinen kompleksisuus. Ohjelmaa kehitettäessä muutosprojektin aikana on pyritty kiinnittämään huomiota hyviksi havaittuihin ohjelmointikäytäntöihin.

Versio	Luokan keskimääräinen syklomaattinen kompleksisuus
v0.1.0	9,65
v1.0.0	7,8
v1.5.0	7,39
2020-03-25-a	7

Taulukko 5. PhpMetrics tulokset luokkatason syklomaattisesta kompleksisuudesta.

6.6 Päätelmät mittaustuloksista

Ohjelmasta saaduista mittaustuloksista voidaan päätellä, että mikäli lähdekoodin laatuun halutaan kiinnittää huomiota, tulee vaaditut mittarit ottaa käyttöön työn laatua arvioitaessa. Lähdekoodin määrän kasvaminen indikoi uusien ominaisuuksien ja uuden koodin lisäämisestä ohjelmistoon. Sillä muutosprojektissa on pyritty ottamaan huomioon hyvät ohjelmointi käytännöt, tulisi sen näkyä muissa valituissa mittareissa.

Larastan-analyysin ja PHP_CodeSniffer-tuloksia arvioitaessa voidaan päätellä, että uuden toiminnallisuuden jatkokehittäminen on tuottanut suhteessa paremman laatuista lähdekoodia kuin ensimmäisen julkaisuversion kehittäminen. Versioiden v0.1.0 ja v1.0.0 välillä lähdekoodin rivimäärä, staattisen analyysin ilmaisemien virheiden ja koodityylipoikkeaminen lukumäärä kasvoi suhteessa kaikkein eniten verrattuna muihin versiomuutoksiin. Toisaalta kyseisessä versiomuutoksessa luokkatasolla keskimääräinen syklomaattinen kompleksisuus laski kaikkein eniten. Lähdekoodin keskimääräinen syklomaattinen kompleksisuus pieneni uudempiin versioihin siirryttäessä.

Mittaustuloksista voidaan päätellä, että ohjelman testikattavuus kasvoi ohjelman kehityksen myötä ja syklomaattinen kompleksisuus laski, sekä koodityylipoikkeamien määrä suhteessa koko lähdekoodin määrään on laskenut. Näiden arvojen perusteella voidaan päätellä, että lähdekoodin laatu on parantunut ohjelmistoprojektin edetessä. Toisaalta staattisella analyysillä havaittavien poikkeamien määrä ei ole laskenut. Tämä toisaalta johtuu siitä, että staattisen analyysin, syklomaattisen kompleksisuuden, koodityylipoikkeamien ja testikattavuuden mittarit esittävät täysin eri asioita lähdekoodista. Muutosprojektia toteutettaessa staattisen analyysin työkalut eivät ole olleet aktiivisesti käytössä. Poikkeamat, jotka on löydetty muilla työkaluilla, tuotantoon vientien yhteydessä ja automaattisilla testeillä, on korjattu kirjoitettaessa uutta koodia.

7 Havaintoja lähdekoodista

Tässä luvussa esitellään, millaisia tekniseen velkaan liittyviä havaintoja lähdekoodista on tehty. Ensimmäiseksi kuvaillaan havainto, joka olisi voitu välttää riittävän kattavalla testaamisella. Toiseksi käsitellään esimerkkiä, jossa tekninen velka ilmenee ohjelmiston ulkoisena ominaispiirteenä. Kolmanneksi pohditaan, kuinka kopioituna koodina ilmenevä tekninen velka on vaikuttanut projektiin. Neljänneksi aiheena on käyttämätön ja käytöstä poistunut lähdekoodi osana projektin koodikokonaisuutta. Toiminnallisuudet ja ominaisuudet, jotka ovat määritelty väärissä paikoissa, ovat viidentenä. Viimeiseksi pohditaan manuaaliseen korjaamiseen turvautumisen vaikutuksia ohjelmistoprojektiin.

Muutosprojektin aikana on koottu havaintoja. Alkuperäiset havainnot on kirjattu projektissa käytettyyn tehtävienhallintaohjelmistoon JIRA-tehtävienhallintaohjelmistoon. Havainnoista on koostettu tutkimusta varten yleistetyt esimerkkitapaukset. Esimerkkitapausten avulla esitellään teknisen velan oireiden vaikutuksia lähdekoodin luettavuuteen, ylläpidettävyyteen, muokattavuuteen ja teknisen velan arvioitavissa olevaan määrään. Ohjelmiston lähdekoodi kuten tutkielmassa käytettävät esimerkit noudattavat PHP 7.2 -määrittelyjä.

Ohjelmistoa kehitettäessä on pyritty välttämään turhan teknisen velan ottamista noudattamalla hyviksi havaittuja ohjelmointikäytäntöjä. Esimerkkitapauksien avulla avataan teknisen velan ilmenemistä ja vaikutuksia ohjelmistoprojektiin hyvistä ohjelmointikäytännöistä huolimatta. Esimerkkien avulla arvioidaan teknisen velan ottamisella saavutettavaa hyötyä ja sen aiheuttamaa riskiä projektille.

7.1 Testaamattomat osat lähdekoodissa

Kirjoitettaessa uusia toiminnallisuuksia ja suoritettaessa muuta ylläpitoa lähdekoodia on mahdollista refaktoroida. Refaktorointi on yksi merkittävimmistä keinoista parantaa lähdekoodin sisäistä laatua ja se on parhaimmillaan yhdessä kattavien dynaamisten testien kanssa. Dynaamisilla testeillä voidaan varmistaa, että lähdekoodin toiminta säilyy muuttumattomana muutosten jälkeen. Tämän lisäksi uusien ominaisuuksien ja koodimuutosehdotusten laatua on tärkeä valvoa koodikatselmoinnein sekä automaattisten koodityylitarkastusten avulla.

Muutosten kohdistuessa testaamattomaan osaan lähdekoodia tulee olla erityisen tarkka. Muutosehdotusta tarjottaessa on hyvä käytäntö päivittää myös järjestelmään kirjoitettuja testejä, jolloin uudet määrittelyt päivitetään osaksi dynaamista testausta. Jollei projektissa noudateta tällaista käytäntöä, voi ilmetä tilanne, jossa koodiin tehtävä muutos sisältää virheen koodikatselmoinnista, vahvasta tyypityksestä ja kehityskehyksen dokumentaation noudattamisesta huolimatta.

```
<php
…
use Illuminate\Database\Eloquent\Builder;
…
Public function scopeFizzBuzz(Builder $builder): Builder
{
       return $this→where('foo', '=', self::FIZZ_BUZZ)
}
```

Koodiesimerkki 6. Paikallinen kyselyrajaus.

Koodiesimerkki 6 kuvaa erään ohjelmistoprojektissa käytetyn paikallisen kyselyrajauksen rakennetta. Laravel-ohjelmistokehitysrungon tarjoama Eloquent-mallinnusmalli tarjoaa paikallisia kyselyrajauksia, joilla voi helposti laajentaa tietorakenteeseen kohdistuvaa tietokantakyselyä [Laravel 2020a]. Virallisessa dokumentaatiossa ohjeistetaan, että kyselyrajauksen tulee palauttaa kyselyinstanssi. Esimerkissä funktion parametrit sekä palautusarvo ovat vahvasti tyypitetyt, jolla varmistetaan määrittelyn mukainen toiminta. Tämä tarkoittaa sitä, että automaattiset staattiset tai dynaamiset testit eivät havaitse ohjelmakoodissa virhettä ellei virheellistä tilannetta erikseen testata. Tässä esimerkkitapauksessa dynaamisten testien testikattavuus ei kata tätä metodia. Ennalta arvaamattomalla tavalla toimivan kyselylausekkeen tuotantoon vienti voi johtaa ohjelman kriittisten osasten rikkoutumiseen tai pahimmillaan muodostaa tietovuodon.

Tällaisessa tapauksessa teknistä velkaa oli kerrytetty monessa muodossa. Teknistä velkaa on kertynyt jo alkuperäisen kyselyn tuotantoonviennissä, sillä lähdekoodin osa oli otettu käyttöön ilman automaattisia testejä. Lähdekoodimuutoksen aiheuttama virheellinen toiminta olisi havaittu, jos automaattinen testi olisi testannut kyselyn tarpeeksi kattavasti. Projektihallinnolliseen tekniseen velkaan kuului myös riittämätön koodikatselmointien järjestäminen ennen koodin tuotantovientiä. Toisaalta, sillä kysely toimii oikein kehityskehyksen määrittelyjen mukaisesti, olisi ongelman havaitseminen koodikatselmoinnilla ollut epätodennäköistä.

7.2 Rajapinnan puutteet

Ohjelmiston muutosprojektin toteutuksessa pyrittiin noudattamaan modernin Restful API:n kehittämisen periaatteita ja yleisesti hyväksyttyjä hyviä ohjelmointikäytäntöjä. Tällaisia seikkoja ovat esimerkiksi suorituskykyyn, rajapinnan määrittelyyn sekä rajapinnan dokumentointiin liittyvät seikat. Ohjelmistokehityksen nopeimmassa vaiheessa käsiteltävän tietomallin rakenne ja määrittelyt saattoivat muuttua kuukausittain. Tämä oli mahdollista, sillä ohjelmiston tuorein versio ei tällöin ollut tuotannossa ja ohjelmaa

pystyttiin testaamaan tuottamatta häiriöitä asiakkaille. Tästä huolimatta toisinaan toteutettaessa uusia toiminnallisuuksia muutosprojektin yhteydessä ohjelmoijalta on voinut jäädä huomioimatta tilanteet, jotka esiintyvät vain tuotantoympäristössä.

Rajapintatoteutus kehitettiin suurelta osin toteuttamaan aiemmin tuotetun lähdekoodin ominaisuuksia. Aiemmin toteutetussa lähdekoodissa vastuu syötteen laadun varmistamisesta ja syötettävän tiedon eheydestä oli käyttöliittymätoteutuksella. Tästä johtuen uudessa Restful API:ssa kaikki rajapinnan säännöt jouduttiin määrittelemään uudelleen. Sillä merkittävälle osalle tietomalleista ei ollut selkeää dokumentaatiota, uuden rajapinnan toteuttaminen johti odottamattomien bugien ilmenemiseen korvattaessa käyttöliittymää. Aiemman lähdekoodin arkkitehtuuriset valinnat peilautuivat myös muutosprojektin alkuvaiheen ratkaisuihin. Ensimmäisiä ominaisuuksia toteutettaessa käytettiin aiemmin toteutettuja rajapinnan reittimäärityksiä, jotka hyödynsivät käyttöliittymän ja taustajärjestelmän tiivistä vastuunjakoa. Tämä havainto liittyy myös tältä osin toiseen lähdekoodista tehtyyn havaintoon, viidennessä kohdassa käsiteltyyn ominaisuuksien sijaitsemisesta väärissä paikoissa.

Tuotantoympäristössä käsiteltävät tietomäärät ovat usein merkittävästi suurempia kuin lähdekoodiin kirjoitetuissa automaattisissa testeissä tai manuaalisissa testeissä käytetyissä esimerkkitapauksissa. Tästä syystä, jos lähdekoodin sisäisessä ja ulkoisessa laadussa on puutteita, suuria tietomääriä käsiteltäessä voi muodostua ongelmia ohjelmiston suorituskyvyssä. Tällaisia ongelmatilanteita voivat olla esimerkiksi n+1 kyselyt tai ylimääräisen tiedon hakeminen kyselyyn vastatessa.

Rajapinnan puutteita on muodostunut otettaessa tietoista teknistä velkaa rajapintaa määriteltäessä. Ennalta tuntemattomat tarpeet rajapinnan käyttämisestä, lähdekoodin ennalta tarjoamat sisäiset palvelut sekä nopealla syklillä toteutettu kehitys kerryttivät teknistä velkaa aiemman päälle. Ohjelmistorajapinnalle määritellyt automaattiset testit mahdollistavat kuitenkin tarvittavat korjaukset.

7.3 Kopioitu koodi

Lähdekoodin sisältä kopioitu koodi on usein merkki teknisestä velasta. Kopioitu koodi tarkoittaa, että ohjelmaan on toistettu samaa asiaa toteuttava lähdekoodi useamman kuin yhden kerran. Tämä tarkoittaa, että kirjoitettaessa uusia muutoksia kyseiseen osaan koodia, samat muutokset tulee kopioida useaan eri paikkaan.

Koodin kopioimista on tapahtunut uutta koodia kirjoitettaessa. Kopioimalla lähdekoodia suoraan on voitu toteuttaa uusi, aiemman kanssa samanlainen ominaisuus suoraan ilman, että on tarvinnut suunnitella tai määritellä stabiileja rajapintoja. Uuden ominaisuuden julkaisemisen jälkeen tiimi ei ole refaktoroinut lähdekoodia esimerkiksi muuttamalla kirjoitettua koodia hyödyntämään luokkien periyttämistä tai yhdistämällä samanlaisia rakenteita abstraktioiksi. Ohjelman jatkokehittämisen aikana on kuitenkin

voitu pienentää kopioidun koodin määrää hyödyntämällä hyviksi havaittuja ohjelmointikäytäntöjä ja dynaamista testaamista.

7.4 Käyttämätön koodi

Ohjelmistoa uudistaessa lähdekoodiin jäi käyttämätöntä koodia. Käyttämätöntä koodia esiintyi luokissa, joissa on koodia, joka on käytössä. Käyttämätöntä koodia ei tutkittavan ohjelman aiemman kehityksen aikana ole aktiivisesti poistettu. Poistamista ei ole tehty joko lähdekoodin osan vastuun epäselvyydestä, oletetusta tarpeesta palata puuttuvan ominaisuuden ohjelmointiin myöhempänä ajankohtana tai epähuomiossa poistettaessa muuta koodia käytöstä. Käyttämätön koodi on tutkitun ohjelmiston tapauksessa myös testaamatonta koodia, sillä muutosprojektin aikana kirjoitetut testit kohdistuvat vain ohjelmiston uusimpiin ominaisuuksiin. Tästä syystä käyttämätön koodi madaltaa lähdekoodiin kohdistuvien testien suhteellista testikattavuutta. Tutkimustuloksista voidaan päätellä, että turhaa lähdekoodia on voitu alkaa poistamaan otettaessa käyttöön automaattisia testejä, sillä niiden avulla on voitu varmistaa ohjelman toimivuus muutosten jälkeen.

Peritystä lähdekoodista on muodostunut käyttämätöntä koodia muun muassa muutosprojektin sivuvaikutuksena, järjestelmässä aiemmin käytetty käyttöliittymä on jätetty osaksi lähdekoodia muutosprojektin alkaessa. Sillä tuotteessa käytetty termistö ja sanasto on samaa vanhassa ja uudessa käyttöliittymässä, erilaisten ominaisuuksien etsiminen lähdekoodin seasta on haastavampaa kuin jos lähdekoodia olisi maltettu poistaa sitä siirrettäessä pois käytöstä. Käyttämättömän koodin osuutta koko lähdekoodista ei voi arvioida ilman profilointityökalua, joka seuraa ohjelmiston käyttöä tuotantoympäristössä.

7.5 Ominaisuudet väärissä paikoissa

Ohjelman kehityksen aikana yhden vastuun periaatteen noudattaminen ei ollut yhtenäistä ja johdonmukaista. Tämä on johtanut tilanteisiin, joissa uutta lähdekoodia kirjoitettaessa on törmätty ennalta-arvaamattomiin tiloihin ohjelmassa, sillä ohjelman määrittelyt ja lähdekoodi eivät ole olleet ohjelmoijille selkeitä. Erilaiset bugit johtuen vääristä toiminnallisuuksien vastuista ovat vaikuttaneet MVC-mallisen toteutuksen laatuun sekä uuden rajapinnan kehitykseen.

Esimerkkinä väärässä paikassa toimivasta ominaisuudesta on muun muassa syötteen validointi. Syötteen validoinnilla voidaan varmistaa, että käyttäjän antama syöte on oikean tyyppistä ja järjestelmän sääntöjen rajoissa laillista. Syötteen validointia ei ole kaikissa ohjelman osissa toteutettu parhaiden käytäntöjen mukaisesti. Laravel-kehityskehyksen kontekstissa paras käytäntö toteuttaa validointi Request-objekteissa. Request-objektit vastaavat HTTP-kutsuun liittyvästä käyttöoikeuksien hallinnasta sekä syötteen

validoinnista. Ohjelmaan on kuitenkin kirjoitettu syötteen validointimäärittelyjä myös ohjelman muuhun toteutuslogiikkaan. Validointisääntöjä on myös toteutettu alkuperäisen MVC-mallisen sovelluksen käyttöliittymään ja ohjelman suorituslogiikan yhteyteen. Ohjelman käyttöliittymään toteutetut validointisäännöt on pitänyt toteuttaa uudelleen rajapintatoteutuksessa. Validointisääntöjen toteutuksen alkuperäinen puutteellisuus on myös mahdollistanut odottamattoman tyyppisten syötteiden lähettämisen järjestelmään, mikä on ollut mahdollinen ohjelman toimintaan liittyvä riski.

Toinen esimerkkitapaus väärään paikkaan määritellystä ominaisuudesta on laskusääntöjä määriteltynä tiedon tallentamisen yhteyteen. Laravel-kehityskehys tarjoaa mahdollisuuden lisätä logiikkaa ohjelmassa määriteltyjen mallien perusmetodien kuten luomisen, tallentamisen ja poistamisen yhteyteen. Kytkemällä esimerkiksi laskentalogiikkaa mallin tallentamisen yhteyteen johtaa siihen, että tiedon tallentamisen prosessilla on liian suuri vastuu, sillä sen toteutuksessa otetaan kantaa tiedon arvoihin.

Kolmas esimerkki ominaisuuden toteutuksen väärästä vastuualueesta on näytettävän tiedon laskenta sovelluksen käyttöliittymässä. Ohjelmassa näytettävällä datalla voi olla merkitys käyttäjän antamien syötteiden kannalta, esimerkiksi siten, että näytettävällä datalla viestitään käyttäjälle sallituista raja-arvoista tai tarjotaan laskennallinen oletusarvo käyttöliittymän syötekentässä. Sillä ohjelman käyttöliittymän toteutus muutosprojektissa vaihdettiin kokonaisuudessaan, kaikki raja- ja oletusarvojen laskenta tuli uudelleen toteuttaa taustajärjestelmässä.

Siirryttäessä SPA-API-malliin väärissä paikoissa toteutetut ominaisuudet on voitu toteuttaa uudelleen määriteltäessä luokille oikeat vastuut. Tämä on ollut mahdollista automaattisten testien varmistaessa ohjelman toiminnan oikeellisuus muutosten jälkeen. Ohjelman virheellinen toiminta on kuitenkin tuottanut ylimääräistä korjaustyötä. Ohjelman aiemmassa kehitysvaiheessa otettu tekninen velka on pitänyt maksaa osana muutostyötä toteuttamalla uudelleen alun perin väärään osaan ohjelmistoa toteutetut ominaisuudet.

7.6 Turvautuminen manuaaliseen korjaamiseen

Projektin eri kehitysvaiheissa on turvauduttu manuaaliseen korjaamiseen. Manuaalinen korjaaminen tarkoittaa tässä kontekstissa sitä, että ohjelman toiminnan oikeellisuudesta vastaava henkilö käy muuttamassa ohjelman tietokantataulun riville oikean tiedon. Tällaiseen toimenpiteeseen turvaudutaan tilanteissa, joissa ohjelmiston toiminnassa on virheitä, jotka johtavat talletettavan tiedon epäjohdonmukaisuuteen tai virheellisyyteen. Manuaalinen korjaaminen voi johtua ohjelmiston määrittelyjen muuttumisesta ja tarvittavan automaattisen tietokantakorjauksen toteuttamatta jättämisestä esimerkiksi nopean bugikorjauksen jälkeen, väärin toimivasta lähdekoodista, jota ei ole ehditty korjata tai puuttuvasta toiminnallisuudesta.

Ohjelmiston määrittelyjen muuttumisen yhteydessä tehtävät automaattiset tietokantakorjaukset ovat kerran ajettavia ohjelmia, jotka muuttavat tietokantaan tallennetun tiedon vastaamaan tuoreimpia määrittelyjä. Tällaisen automaattisen tietokantakorjauksen toteuttaminen on tutkitun projektin aikana usein jäänyt tekemättä epähuomiossa. Muutoksia toteuttaessa ei esimerkiksi ole kartoitettu kaikkia pienimuotoista riskejä, mitä uuden koodin tuotantovienti voi aiheuttaa. Automaattisen tietokantakorjauksen toteuttaminen vaatii oman tuotantovientinsä ja kerta-ajon ja siksi sellaisen toteuttaminen tulisi tehdä harkitusti.

Useissa tapauksissa manuaalisella korjaamisella on ratkaistu väärin ohjelmoidun ominaisuuden aiheuttamia virhesyötteitä tietokannassa. Väärin toimiva ominaisuus on voinut esimerkiksi käsitellä tietoa väärän tyyppisenä, kuten esimerkiksi päivämääriä merkkijonoina. Tällaisessa tapauksessa päivämäärien käsittelyyn tarvittavat perusoperaatiot eivät ole käytettävissä, kuten kuukauden ensimmäiseen tai viimeiseen päivään viittaaminen. Tämän takia aikajaksojen hallittu generoiminen on haastavampaa, jolloin puutteellisesta testauksesta johtuen tuotantoon asti pääsevät bugit on jouduttu korjaamaan manuaalisesti niiden ilmetessä. Varsinaisen korjauksen toteutuminen on tällaisissa tapauksissa viivästynyt manuaalisen korjaamisen ollessa yksinkertaisempi ratkaisu kuin selvittää ongelman juurisyytä. Manuaalinen korjaaminen on nähty pienemmäksi riskiksi kuin varsinaisen korjauksen tekeminen.

Projektin elinkaaren aikana on turvauduttu myös manuaaliseen korjaamiseen puuttuvan ominaisuuden tuottamien ongelmien ratkaisemiseksi. Tällaisissa tilanteissa varsinaisen ominaisuuden toteuttamista on jouduttu viivyttämään manuaalisen korjaamisen tarvitseman työmäärän saadessa korkeamman prioriteetin kuin varsinaisen kehittämisen. ja pienemmäksi investoinniksi kuin sovelluksen ominaisuuden ohjelmoiminen.

Tiedon manuaalisesti korjaaminen on ohjelmoijalle täysin ylimääräistä työtä. Manuaalisen korjaamisen sijaan tulisi korjata ongelman aiheuttava virheellinen lähdekoodi, jonka jälkeen tulisi suorittaa yksi tai useampi automaattinen tietokantakorjauskysely, joka korjaa kaikki aiheeseen liittyvät ongelmat. Manuaalisten korjausten raportointi projektinhallintajärjestelmään on ollut puutteellista ja korjauksiin käytettyä aikaa ei ole erikseen seurattu. Tämä tarkoittaa sitä, että projektisuunnittelussa ei olla voitu ottaa huomioon manuaalisten korjausten aiheuttamaa työkuormaa. Manuaaliseen korjaamiseen turvautuminen voi vaatia tietoon oleellisesti liittyvän lähdekoodin osan sisäistämistä ulkoa tai sen toiminnan tarkistamista lähdekoodista. Lähdekoodin toiminnan ulkoa opetteleminen vaatii ylimääräistä mentaalista kapasiteettia.

Manuaalinen korjaaminen voidaan käsittää metaforan mukaisen teknisen velan koron tai juoksevien kulujen maksamista, joka ei vähennä velkaa. Manuaalinen korjaaminen on selkeä merkki teknisestä velasta muodostuvista oireista, jotka kuormittavat oh-

jelmistokehittäjiä ja hidastavat uusien toiminnallisuuksien toteuttamista. Korjausten sijaan tulisi priorisoida varsinaisten korjausten ja ominaisuuksien toteuttamista.

8 Päätelmät ja pohdinta

8.1 Tutkimuksen tulokset

Tutkimuksessa on analysoitu teknisen velan vaikutuksia sekä hallintaa ja välttämistä start-up projektin aikana. Tutkimusaineiston lähteenä on käytetty projektia, johon tehtiin kesällä 2019 käyttöliittymän muutostyö. Tutkimusaineisto muodostuu ohjelman kehittämisestä kerätyistä havainnoista koostetuista päätelmistä sekä projektista automaattisilla työkaluilla kerätystä datasta.

Tuloksista voidaan päätellä, että hyvien ohjelmointikäytäntöjen noudattaminen ei ole helppoa. Esimerkkien ja mittaustulosten perusteella voidaan päätellä, että teknistä velkaa kertyy siitäkin huolimatta, että lähdekoodin laadusta pitää kiinni. Tämä voidaan tulkita, että teknisen velan mittaaminen ja sen määrän tai laadun arvioiminen ei ole helppoa. Käytettävissä olevien mittareiden tulokset eivät välttämättä kerro teknisen velan todellisesta tilasta ohjelmistoprojektissa.

Teknisen velan välttämisen keinoja ovat yhtenäisen koodityylin noudattaminen, koodin laadun varmistaminen koodikatselmoinneilla ja automaattisten staattisten sekä dynaamisten testien käyttäminen. Yhtenäisellä koodityylillä lähdekoodin luettavuutta sekä ylläpidettävyyttä voi parantaa. Koodikatselmointien avulla lähdekoodin kirjoittajan vastuuta muutoksista voidaan jakaa ja muutosten ymmärrettävyyttä voidaan varmistaa. Automaattisten testien avulla lähdekoodissa esiintyviä virheitä on helpompi havaita ilman, että ne siirtyvät tuotantoympäristöön.

Teknistä velkaa muodostuu ohjelmistoon lähes väistämättä. Siksi on hyvä tiedostaa, että jonkin verran teknistä velkaa on hyvä ottaa. Toisaalta tekninen velka muodostaa ohjelmiston luotettavuudelle riskin, joka voi johtaa hidastuneeseen kehitystahtiin, ylimääräisen työn muodostumiseen, tietoturvapoikkeamiin tai ohjelman virhetilanteisiin. Siksi on hyvä pyrkiä arvioimaan, kuinka teknisestä velasta voidaan saada mahdollisimman paljon hyötyä minimoimalla sen aiheuttamat haitat.

8.2 Tutkimuksen haasteet

Tutkimuksen suurin haaste on ollut testikattavuuden mittausten järjestäminen vertailukelpoisesti eri versioiden välillä. Ohjelmiston testikattavuuden tulosten vertailu eri versioiden välillä on haastavaa, sillä eri versioiden testisettien toimivuus ei ole taattu ja kattavuuslaskennan määrittelyt eivät vastaa toisiaan. Koodityylipoikkeamien ja staattisen analyysin järjestäminen on ollut helpompaa, sillä niiden ajon onnistuminen ei ole riippuvaista lähdekoodin laadusta.

Tutkimuksessa haasteita tuotti myös tutkimusaineiston kerääminen. Yhtenä ohjelmiston laadun mittarina käytettävissä oleva CRAP-arvo on laskettavissa ohjelmiston lähdekoodille tiedoston tarkkuudella, mutta kokonaiskuva,a kuten keskiarvoa luokkatason CRAP-arvosta, ei käytettävissä olevilla työkaluilla ole suoraan laskettavissa. Olisi kiinnostavaa tarkastella, laskeeko lähdekoodin keskimääräinen CRAP-arvo, sillä testikattavuus kasvaa ja keskimääräinen syklomaattinen kompleksisuus laskee.

8.3 Jatkotutkimusmahdollisuudet

Tutkimuksessa esiteltiin ohjelmiston lähdekoodissa esiintyneitä teknisen velan artefakteja hyödyntämällä esimerkkejä teknisen velan aiheuttamista riskeistä ohjelmiston ylläpitämiselle ja muokkaamiselle sekä mittaamalla ohjelmiston lähdekoodia sen kehityksen aikana käytetyillä työkaluilla. Tutkimusaineisto oli kerätty ohjelmistoprojektin tuotannon aikana kerätyistä havainnoista ja mittaamalla lähdekoodia automaattisilla testeillä.

Kiinnostavaa olisi ohjelmistoprojektin jatkon kannalta jatkaa ohjelmiston ylläpitämisen ja muuttamisen riskien kartoittamista esimerkiksi ATAM- tai MPM- (maintenance prediction model) menetelmiä hyödyntäen. Formaali riskikartoitus voisi auttaa hahmottamaan projektin teknisen velan aiheuttamia riskejä.

Toinen mielenkiintoinen lähestymistapa olisi tutkia tutkimuksessa käytettyjen menetelmien kattavuutta tarkemmin. Kartoittamalla teknistä velkaa testikattavuuden, koodityylipoikkeamien ja staattisen analyysin ilmaisemien virheiden perusteella antaa kokonaiskuvan lähdekoodin laadusta. Kuten Zazworka et al. [2013] toteavat tutkimuksessaan, eri menetelmien ilmaisemat virheet eivät täsmää toisiaan. Tästä syystä olisi kiinnostavaa tutkia, olisiko tässä tutkimuksessa esitetyt teknisen velan esimerkit mahdollista ollut havaita ennakkoon automaattisilla työkaluilla.

Kolmas tutkittava aihe olisi vertailla ja arvioida teknisellä velalla saavutettua hyötyarvoa suhteessa teknisestä velan realisoituneisiin riskitekijöihin. Tutkimus voitaisiin toteuttaa tutkimalla tunnettuja teknisen velan artefakteja. Tunnettujen artefaktien mahdollistamia hyötytekijöitä voitaisiin pyrkiä mittaamaan arvioimalla niiden tuottamaa rahallista hyötyä ja vertailemalla sitä niiden mahdollisesti toteutuneisiin haittatekijöihin.

9 Lähteet

[Agile Alliance 2001] Manifesto for Agile Software Development. Saatavilla: http://agilemanifesto.org/. Tarkastettu 23.03.2020.

[Ampatzoglou et al. 2015] Areti Ampatzoglou, Apostolos Ampatzoglou, Alexander Chatzigeorgiou ja Paris Avgeriou. 2015. The financial aspect of managing technical debt: a systematic literature review. Information and Software Technology 64.C, 52–73.

[Antinyan et al. 2018] Vard Antinyan, Jesper Derehag, Anna Sandberg ja Miroslaw Staron. 2018. Mythical unit test coverage. IEEE Software, May 2018, Vol.35(3), 73-79.

[Besker et al. 2018] Terese Besker, Antonio Martini ja Jan Bosch. Technical debt cripples software developer productivity. 2018. *ACM/IEEE International Conference on Technical Debt*.

[Boswell ja Foucher 2011] Dustin Boswell ja Trevor Foucher. 2011. *The Art of Readable Code*. O'Reilly Media.

[Clements et al. 2000] Paul Clements, Rick Kazman ja Mark Klein. 2000. *ATAM: Method for Architecture Evaluation*. CMU/SEI-2000-TR-004. Carnegie-Mellon University Pittsburgh PA Software Engineering Inst.

[Codabux ja Williams. 2013], Codabux Z., Williams, B. Managing technical debt: an industrial case study. 2013. In: *4th International Workshop on Managing Technical Debt (MTD)*, 8–15.

[Codabux et al. 2014] Codabux Z., Williams, B, Niu N. 2014. A quality assurance approach to techinical debt. In: *International Conference on Software Engineering Research and Practice*.

[Cunningham 1993] Ward Cunningham. 1993. The WyCash portfolio management system. In: *Addendum to proceedings on object-oriented programming, languages, and applications (addendum)*. 29-30.

[Feathers 2004] Michael Feathers. 2004. *Working Effectively with Legacy Code*. Prentice Hall.

[Fowler et al. 1999] Martin Fowler, Kent Beck, John Brant, William Opdyke ja Don Roberts. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional.

[Fowler 2009] Martin Fowler. 2009. TechnicalDebtQuadrant, *http://martinfowler.com/bliki/TechnicalDebtQuadrant.html* Tarkastettu 23.03.2020.

[Green ja Ledgrad 2011] Green, R. ja Ledgard, H. 2011. Coding guidelines: finding the art in the science. Commun. ACM 54, 57–63.

[Guru99 2020] Guru99. 2020. Mccabe's cyclomatic complexity: calculate with flow graph. *https://www.guru99.com/cyclomatic-complexity.html* Tarkastettu 23.03.2020

[Inozemtseva ja Holmes 2014] Laura Inozemtseva ja Reid Holmes. 2014. Coverage is Not Strongly Correlated with Test Suite Effectiveness. In: *International Conference on Software Engineering*. ACM.

[Kaur ja Mahajan 2015] Aastha Mahajan ja Paminder Kaur. 2015. A review on evolution and versioning of ontology based information systems. *IOSR Journal of Computer Engineering* Vol 17, issue 2, 35-43.

[Kernighan ja Plauger 1974] B. W. Kernighan ja P. J. Plauger. 1974. *Programming Style: Examples and Counterexamples*. McGraw-Hill.

[Klinger et al. 2011] Tim Klinger, Peri Tarr, Patrick Wagstrom ja Clay Williams. 2011. An enterprise perspective on technical debt. In: *Proc. of 2 nd MTD'11*, 35-38.

[Kochhar et al. 2017] Pavneet Singh Kochhar, David Lo, Julia Lawall ja Nachiappan Nagappan. 2017. Code coverage and postrelease defects: a large-scale study on open source projects. In: *IEEE Transactions on Reliability*, Vol.66(4), 1213-1228.

[Larastan 2020] Larastan. *https://github.com/nunomaduro/larastan* Tarkastettu 23.03.2020.

[Laravel 2020a] Laravel LLC. 2020. Local Scopes. *https://laravel.com/docs/5.7/eloquent#local-scopes* Tarkastettu 23.03.2020.

[Laravel 2020b] Laravel LLC. 2020. Testing: Getting Started. *https://laravel.com/docs/5.8/testing* Tarkastettu 23.03.2020.

[Lavazza et al. 2018] Luigi Lavazza, Sandro Morasca ja Davide Tosi. 2018. Techincal debt as an external software attribute. In: *ACM/IEEE International Conference on Technical Debt '18*. 21-30.

[Lehman 1980] Lehman M. M. 1980. Programs, life cycles, and laws of software evolution. In: *Proceedings of the IEEE, 68 (9)*, 1060–1076.

[Li et al. 2014] Zengyang Li, Peng Liang ja Paris Avgeriou. 2014. Architectural debt management in value-oriented architecting. In: *Economics-Driven Software Architecture*, 183-204.

[Li et al. 2015a] Zengyang Li, Peng Liang ja Paris Avgeriou. 2015. Architectural technical debt identification based on architecture decisions and change scenarios. In: *2015 12th Working IEEE/IFIP Conference on Software Architecture*, 65-74.

[Li et al. 2015b] Zengyang Li, Peng Liang ja Paris Avgeriou. 2015. A systematic mapping study on technical debt and its management. In: *The Journal of Systems & Software*, Vol.101, 193-220.

[Lim et al. 2012] Erin Lim, Nitin Taksande ja Carolyn Seaman. 2012. A balancing act: what software practitioners have to say about technical debt. In: *IEEE Software*, vol. 29, no. 6, 22-27.

[Madsen 2006] Jana Madsen. 2006. Paying for Deferred Maintenance. In: *Buildings magazine* 06/05/2006, s. 60-62.

[Mäntylä ja Lassenius 2009] Mäntylä, M.V. ja Lassenius, C. 2009. What types of defects are really discovered in code reviews? In: *IEEE Trans. Softw. Eng.* 35, 430–448.

[McCabe 1976] Thomas J. McCabe, Sr. 1976. A complexity measure. In: *IEEE Transactions on Software Engineering* SE-2, 4: 308–320.

[McConnell 2004] Steve McConnell. 2004. *Code Complete*. Microsoft Press.

[McIntosh et al. 2015] Shane McIntosh, Yasutaka Kamei, Bram Adams ja Ahmed E. Hassan. An empirical study of the impact of modern code review practices on software quality. In: *Empir Software Eng* 21, 2146–2189.

[Melton ja Tempero 2007] Hayden Melton ja Ewan Tempero. 2007. An empirical study of cycles among classes in Java. In: *Empirical Software Engineering* 12, 4, 389–415.

[Mili ja Tchier 2015] Ali Mili ja Fairouz Tchier. 2015. *Software Testing: Concepts and Operations*. Wiley & Sons, Inc.

[Mishkin ja Eakins 2012] F. Mishkin ja S. Eakins. 2012. *Financial Markets and Institutions.* Pearson Prentice Hall.

[Mockus et al. 2009] A. Mockus, N. Nagappan, ja T. Dinh-Trong. 2009. Test coverage and post-verification defects: a multiple case study. In: *Proc. 3rd Int. Symp. Empirical Softw. Eng. Meas*. 291-301.

[Moogk 2012] Dobrila Rancic Moogk. 2012. Minimum viable product and the importance of experimentation in technology startups. In: *Technology Innovation Management Review* 2, 3, 23-26.

[Myers 1979] Myers, G. J. 1979. *The Art of Software Testing*. John Wiley and Sons.

[Osman 2010] Osman Osman. 2010. Project management processes: waterfall software development. In: Osman Osman. *Case Studies in Project*, *Program*, *and Organizational Project Management*, 36-41.

[Ozkaya et al. 2016] Ipek Ozkaya, Philippe Kruchten, Robert Nord, Paris Avgeriou ja Carolyn Seaman. 2016. Reducing friction in software development. In: *IEEE Software*, 33, 1, 66-73.

[Ozkaya et al. 2019] Ipek Ozkaya, Philippe Kruchten ja Robert Nord. 2019. *Managing Technical Debt: Reducing Friction in Software Development*. Software Engineering Institute/Carnegie Mellon.

[PHP 2020] The PHP Group. 2020. Logical Operators. *https://www.php.net/manual/en/language.operators.logical.php*. Tarkastettu 23.03.2020.

[PHP_CodeSniffer 2020] PHP_CodeSniffer. *https://github.com/squizlabs/PHP_CodeSniffer* Tarkastettu 23.03.2020.

[PhpMetrics 2020] PhpMetrics, static analysis for PHP. *https://phpmetrics.org/* Tarkastettu 23.03.2020.

[PHPStan 2020] PHP Static Analysis Tool. *https://github.com/phpstan/phpstan* Tarkastettu 23.03.2020.

[PHPUnit 2020] The PHP Unit Testing framework. *https://github.com/sebastianbergmann/phpunit* Tarkastettu 23.03.2020.

[Ramasubbu ja Kemerer 2015] Narayan Ramasubbu ja Chris Kemerer. 2015. Technical debt and the reliability of enterprise software systems: a competing risks analysis. In: *Management Science* 62, 5, 1487-1510.

[Ramler ja Wolfmaier 2006] Rudolf Ramler ja Klaus Wolfmaier. Economic perspectives in test automation: balancing automated and manual testing withopportunity cost. In: *Proceedings of the 2006 international workshop on automation of software test*, 85-91.

[Ritchie 2010] Peter Ritchie. 2010. *Refactoring with Microsoft Visual Studio 2010*. Packt Publishing Ltd.

[Savoia ja Evans 2007] Alberto Savoia ja Bob Evans. Crap4j. *http://www.crap4j.org/* Tarkastettu 23.03.2020.

[Schroeder 1999] Mark Schroeder. 1999. A Practical guide to object-oriented metrics. In: *IT Professional* 1, 6, 30-36.

[Stamelos 2009] Ioannis Stamelos. 2009. Software project management anti-patterns. In: *The Journal of Systems and Software* 83, 52–59.

[Strassmann 2000] Paul Strassmann. 2000. The Y2K 'ransom'. *Computerworld*, 34, 2, 47.

[Taipale et al. 2011] Ossi Taipale, Jussi Kasurinen, Katja Karhu ja Kari Smolander. 2011. Trade-off between automated and manual software testing. In: *International Journal of System Assurance Engineering and Management* 2, 2, 114-125.

[Tamburri et al. 2015] Damian Tamburri, Philippe Kruchten, Patricia Lago ja Hans Vliet. 2015. Social debt in software engineering: insights from industry. In: *Journal of Internet Services and Applications* 6, 1, 1-17.

[Virtala ja Äijö 2011] Pertti Virtala ja Juha Äijö. 2011. Liikenneväylien korjausvelka: Laskentamallin kehitys ja testaus. In: *Liikenneviraston tutkimuksia ja selvityksiä: 42/2011.*

[Yli-Huumo et al. 2015] Jesse Yli-Huumo, Andrey Maglyas ja Kari Smolander. 2015. How do software development teams manage technical debt? - An empirical study. In: *The Journal of Systems and Software* 120, 195-218.

[Wang et al. 2012] Yanqing Wang, Hang Li, Yuqiang Feng, Yu Jiang ja Ying Liu. 2012. Assessment of programming language learning based on peer code review model: Implementation and experience report. In: *Computers & Education* 59, 2, 412-422.

[Zazworka et al. 2013] Nico Zazworka, Antonio Vetro', Clemente Izurieta, Sunny Wong, Yuanfang Cai, Carolyn Seaman ja Forrest Shull. 2013. Comparing four approaches for technical debt identification. In: *Software Quality Journal* 22, 3, 403-426.

[Zou et al. 2019] Weiqin Zou, Jifeng Xuan, Xiaoyuan Xie, Zhenyu Chen ja Baowen Xu. How does code style inconsistency affect pull request integration? An exploratory study on 117 GitHub projects. In: *Empirical Software Engineering* 24,6, 3871-3903.

[Xdebug 2020] Code Coverage Analysis. *https://xdebug.org/docs/code_coverage* Tarkastettu 23.03.2020.