Kehittäjien vaihtamisen negatiivinen vaikutus ohjelmistokehitykseen

DI-tutkielma
Turun yliopisto
Tietotekniikan laitos
Turun Yliopisto
2022
Antti Vuorinen

TURUN YLIOPISTO
Tietotekniikan laitos

Antti Vuorinen: Kehittäjien vaihtamisen negatiivinen vaikutus ohjelmistokehitykseen

DI-tutkielma, 69 s.
Turun Yliopisto
Marraskuu 2022

Ohjelmistokehitys ja tietotekniikka on ollut osa ihmisten elämään, jo kauan. Tämän myötä on tehty ohjelmia, joiden elinikä on pitkä, ja ohjelmistokehittäjät ovat ehtinyt vaihtumaan kehityksen elinkaaren aikana useampaan otteeseen. Ohjelmistokehittäjien vaihtuessa tapahtuu kehityksen luovutus. Yleinen tapahtuma, jonka useimmat ohjelmistokehitysprosessit tulevat kokemaan. Aihe on hyvin vähän tutkittu. Käyn läpi mahdollisia tekniikoita, jotka voivat auttaa ohjelmistokehityksen luovutuksessa. Tutkin aikaisempia tutkimuksia, joissa on käsitelty ohjelmistokehityksen luovutusta. Määrittelen samalla mitä negatiivisia vaikutuksia huonolla luovutuksella voi olla.
Tutkin yli 10 vuotta vanhaa ohjelmisto ja sen elinkaaressa tapahtunutta luovutusta. Arvioin mitä huonoja vaikutuksia ohjelmistokehityksellä voi olla ohjelmistokehitykselle, kuten kehityksen hidastuminen ja tekninen velka. Tutkin järjestelmää käymällä läpi Git sovelluksen Git kommitteja, jotka pystyvät kertomaan menneisyydessä tapahtuneista muutoksista ohjelmaan. Teknisen velan mahdolliseen muutokseen järjestelmässä käytän Sonarquben staattista koodin analysointia arvioimaan, kuinka paljon tekninen velka tulee kasvamaan uuden kehittäjän aloittaessa.
Tulosten keräämisen jälkeen pohdin niiden perusteella mahdollisia ongelmia, jotka voivat vääristää tulosta. Arvioin myös mahdollisia syitä, jotka ovat voineet aiheuttaa negatiivisia vaikutuksia ohjelmistokehityksen luovutuksessa. Viimeisenä ehdotan uusia tutkimuksia, jotka ovat isommassa skaalassa, kuin yhteen ohjelmistokehitysprojektiin ja tutkimusta ohjelmistokehitys projektiin, jossa luovutus on onnistunut paremmin.

Asiasanat: ohjelmistokehitys, tietotekniikka, Git, ohjelmiston luovutus, Vincit, Fonecta, Kotisivukone

Sisällys

1 Johdanto

Ohjelmistokehityksellä alkaa olemaan jo hieman historiaa. Teoriapohja ohjelmoinnille ja tietokoneille on jo yli 85 vuotta. Ensimmäiset korkean tason ohjelmointikielet julkaistiin yli 70 vuotta sitten.[1] Mikä myös tarkoittaa, että kehitysprosessia on ehditty tutkimaan ja tehostamaan vuosien varrella. Tehostaakseen ohjelmiston kehitystä, on suunniteltu ohjelmiston elinkaaren kehitysmalleja ja käytänteitä. Mallien ideana oli tehdä selkeät toimintatavat vaatimusten analysointiin, suunnitteluun, ohjelmiston/arkkitehtuurin mallintamiseen, kehitykseen, testaamiseen, julkaisuun ja ylläpidon välillä. Tällöin luotiin selkeä suunta, miten hallita ohjelmiston elinkaarta. Elinkaarimallien ansiosta pystyttiin luomaan paremman laatuisia ohjelmia, ja hallitsemaan paremmin ohjelman elinkaarta. [2]

Ensimmäinen tunnettu malli oli vesiputousmalli. Vesiputousmallin ideana oli, että suoritetaan jokainen ohjelmistokehityksen vaihe yksitellen. Seuraavaan vaiheeseen ei menty, ennen kuin aikaisempi oli tehty valmiiksi. Myöhemmin kehitettiin muitakin malleja, joilla parannettiin sovelluskehityksen elinkaarta. Esimerkkinä tästä on iteratiivinen malli. Tässä mallissa koko kehityksen elinkaari käydään läpi useita kertoja, tekemällä inkrementaalisia muutoksia ohjelmistoon. Iteratiivinen malli, tai siihen pohjautuvat mallit, ovat nykyään käytetyimpiä ohjelmistokehityksen malleja. Näiden lisäksi on kehitetty ohjelmistokehityksen käytänteitä, jotka tukevat elinkaaren ylläpidossa.

Käytännöt ja mallit käsittelevät mittavasti koko kehityksen toiminnallisuuden, ja

näistä on tullut merkittävä osa ohjelmistokehitystä. Niistä on oma ISO-määritelmä[3]. Silti missään ohjelmistokehityksen malleissa tai käytänteissä ei käsitellä kehitysvastuun luovutusta toiselle taholle. Oletuksena vaikuttaa olevan, että samat henkilöt työskentelevät projektissa koko sen elinkaaren läpi. Tässä ei oteta huomioon, että ohjelmistokehityksessä monet tuotteet ovat tuotannossa jopa vuosia. Esimerkkinä tästä; vuonna 2018 julkaistu God of War-videopelin kehitys kesti noin 4 vuotta.[4][5] Neljässä vuodessa kehitysvastuu on voinut hyvin helposti vaihtua.

Vanhojen tuotteiden ylläpito on myös tärkeää. Käytössä tulee olemaan yhä enemmän tuotteita, joita pitää ylläpitää erittäin pitkään. Googlen hakukone on tästä hyvä esimerkki. Hakukone on ollut käytössä jo yli 20 vuotta.[6] Tämä tuote tulee todennäköisesti elämään pidempään, kuin tämänhetkiset kehittäjänsä. Huomataan, että luovutukset ovat yleisiä ja alati toistuvia tapahtumia ohjelmistokehityksen alalla. Tällöin on tärkeää tehdä luovutus huolellisesti, sillä luovutuksella voi olla negatiivisia vaikutuksia. Luovutuksessa voidaan menettää paljonkin tietoa, minkä poisjäänyt kehittäjä vie mukanaan. Kehitys voi hidastua tiedon menettämisen takia, tai aiheuttaa vahinkoja ohjelmistolle. Suomessa tämä tulee olemaan tärkeä pohdinnan aihe, kun iso määrä työntekijöitä menee eläkkeelle. Tällöin on tärkeää varmistaa, että mahdollisimman paljon tietoa siirtyy uudelle kehittäjälle.

Tieteelliset tutkimukset ohjelmistokehityksen vastuun luovuttamisesta ovat hyvin vanhentuneita, ja aiheesta löytyy vain muutama ajankohtainen tutkimus. Muut tutkimukset ovat joko liian yleistäviä tai liian vanhoja. Eivätkä ota huomioon nykyajan kehitysmalleja tai käytänteitä. [7]

Tässä diplomityössä tutkimus kysymyksenä on tutkia mahdollisia negatiivisia vaikutuksia, joita huonolla luovutuksella voi olla ohjelmistokehitykseen elinkaareen. Negatiivisia vaikutuksia arvioidaan kolmella hypoteesilla:

K1 Kommittien granulaisuus kasvaa luovutuksen alussa. Uudella kehittäjällä on paljon vähemmän ymmärrystä tuotteesta, kuin luovutuksen antajalla. Vähäisempi

kokemus tuotteesta voi johtaa pienempiin Git-kommittien muutoksiin, kun verrataan muutoksia koodirivien määrällä Gitin kommiteissa. Uudella kehittäjällä ei ole kokemuksen tuomaa varmuutta ja ymmärrystä tuotteesta, jolloin hän ei uskalla tehdä isoja muutoksia ohjelmaan. Hän voi pelätä ohjelman rikkomista tai aiheuttavan vahinkoa tuotteelle. Tämä kuitenkin tulee parantumaan kokemuksen myötä. Kehittäjä voi mahdollisesti nousta samalle tasolle, kuin vanha kehittäjä. Tämä tapahtuu, kun kehittäjä alkaa ymmärtämään ohjelmiston toimivuutta.

K2 Tuotteen luovutuksen yhteydessä tullaan huomaamaan selkeä hidastuminen kehityksen elinkaaressa. Uudella kehittäjällä menee aikaa tuotteen ymmärtämiseen ja sisäistämiseen. Uusien kehittäjien pitää hidastaa tuotteen kehitystä, jotta heillä olisi aikaa opiskella tuotteen järjestelmää. Osa tarvittavasta osaamisesta tullaan oppimaan vasta, kun kehitysvastuu on luovutettu. Tämän toivotaan parantuvan tietyn ajanjakson jälkeen, kun uudet kehittäjät ovat sisäistäneet tarvittavat tiedot tuotteesta.

K3 Tekninen velka tulee luovutuksen jälkeen kasvamaan. Luovutuksen yhteydessä tai sen välittömässä läheisyydessä tulee muodostumaan teknistä velkaa. Syynä voi olla uuden kehittäjän vähäinen osaaminen ja riittävä tuntemuksen puute tuotteesta. Tällöin kehittäjä tekee 'ei niin hyviä' (not so good) ratkaisuja tuotteeseen tehdessään muutoksia. Toisena riskinä voi olla olemassa oleva tekninen velka, josta uudella kehittäjällä ei ole tietoa. Tällöin tekninen velka tulee mätänemään ja pahentumaan ajan myötä.

Luvussa 2 käsitellään teknisen velan merkitystä ja mitä se tarkoittaa ohjelmiston kehityksessä. Käsittelen kehityksessä käytettyjä malleja ja käytänteitä, jotka auttavat ymmärtämään kehitysprosessia. Luvussa 3 käyn läpi aikaisemmat tutkimukset, jotka käsittelevät sovelluskehityksen luovutusta. Luvussa 4 käyn läpi tutkimuksen kohteena olleen Fonectan kotisivukoneen, jonka kehitysvastuu tätä työtä tehdessä on Vincit yhtiö. Luvussa 5 kerron löydetyt tuloksen. Luvussa 6 pohdin tuloksia ja ar-

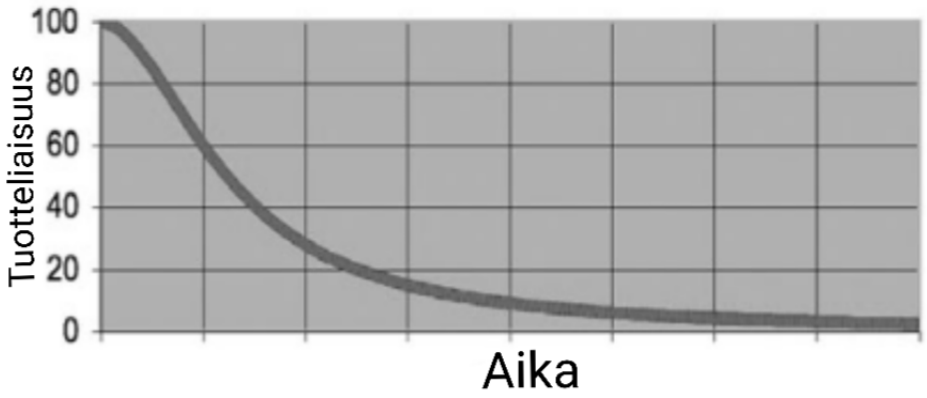
vioimme mahdollisia johtopäätöksiä. Viimeisenä kritisoin ongelmakohtia ja pohdin mahdollisia jatkotutkimuksia.

2 Alustus

2.1 Tekninen velka

Termi tekninen velka "technical debt"esiteltiin ensimmäistä kertaa 1990-luvun alussa. Termin luojana pidetään Ward Cunninghamia, joka yritti selittää osakkeiden omistajille mitä haittavaikutuksia voi olla liian nopeasti tehdystä ICT-projektista. Esimerkkinä hän käytti talousmaailmaa: Tekninen velka on kuin normaali velka. Otamme velkaa, että saamme toteutettua asioita nopeasti ja maksamme hinnan myöhemmin. Teknisessä kehityksessä tämä voi tarkoittaa huonolaatuista koodia tai hätäisesti tehtyjä ratkaisuja, joiden avulla yritetään saavuttaa tuloksia nopeammin. Huonolaatuista koodia Ward kuvaili koodiksi, joka ei ole optimaalista, "not quite right". Samoin kuin normaalissa velassa. Tuote tulee kärsimään, jos velkaa ei makseta.[8]

Teknisen velan mahdolliset haitat ovat hyvin selkeät ja vaarallisia. Robert C. Martin kirjoitti aikoinaan kirjassaan "Clean Code: A Handbook of Agile Software Craftsmanship", että huono koodi tulee hankaloittamaan tulevaisuudessa ohjelman kehitystä, ja pahimmassa tapauksessa hidastaa merkittävästi, tai pysäyttää täysin tuotteen kehityksen. Vaikka tekninen velka ei tuhoaisi täysin ohjelman kehitystä se silti voi aiheuttaa merkittäviä lisäkuluja projektin omistajalle. Kuva 2.1 Clean code kirjasta kuvaa teknisen velan ongelmaa, kun sen annetaan kasaantua ohjelmisto kehityksessä. [9]



Kuva 2.1: Teknisen velan hinta

Teknisen velan syntyhetkellä tai sen alkuaikoina ei mahdollisesti ole näkyviä haittavaikutuksia. Kuitenkin yksinkertainenkin muutos koodiin voi vaatia päiviä tai viikkoja koodin huonon laadun takia. Muutos voi rikkoa tuotteen toiminnallisuuksia helposti muissa paikoissa koodia, ja uusien vikojen korjaaminen voi aiheuttaa vielä enemmän teknistä velkaa. Silloin syynä tällaiseen on hyvin todennäköisesti entinen tekninen velka. Teknisen velan kasaantuessa on mahdollista, että sitä luodaan vielä lisää. Sillä ongelmat halutaan mahdollisimman nopeasti piiloon. Tämä uusi velka voi olla haitallisempaa ja vaikeammin ratkaistavaa. Tällaisissa tilanteissa kehittäjät joutuvat tekemään hyvin massiivisen koodin uudelleen läpikäymisen, joka voi kestää viikkoja tai jopa kuukausia. Pahimmassa tapauksessa tuote joudutaan hylkäämään ja sen tilalle luomaan uusi, koska vanhan tuotteen teknistä velkaa on mahdotonta maksaa tai se ei ole kannattavaa. Vaikka usein tekninen velka on haitaksi, se voi olla hyväksi tuotteen kehitykselle. Välillä tuotteelle voi olla hyväksi, että otetaan teknistä velkaa. Sen ansiota tuotetta voidaan saada kehitettyä markkinoilla nopeammin, mikä voi olla hyväksi tuotteelle. Tärkeämpää on velan hallinta, jolloin pystytään minimoimaan teknisen velan aiheuttama rasitus. Yhtiön hallitessa teknistä velkaa hyvin sen maksaminen takaisin on myös helpompaa. Teknisen velan selkeä dokumentointi sen luonti hetkellä auttaa velan hallinnassa.

Teknisen velka pystytään jakamaan kolmeen kategoriaan. Ensimmäinen on tahallinen tekninen velka, jossa kehittäjät tarkoituksella hankkivat teknistä velkaa. Useimmiten tämä tehdään, jotta tuote saadaan julkaistua mahdollisimman nopeasti. Optimaalisessa tilanteessa tekninen velka olisi ylläpidettyä ja hoidettaisiin pois lähitulevaisuudessa.

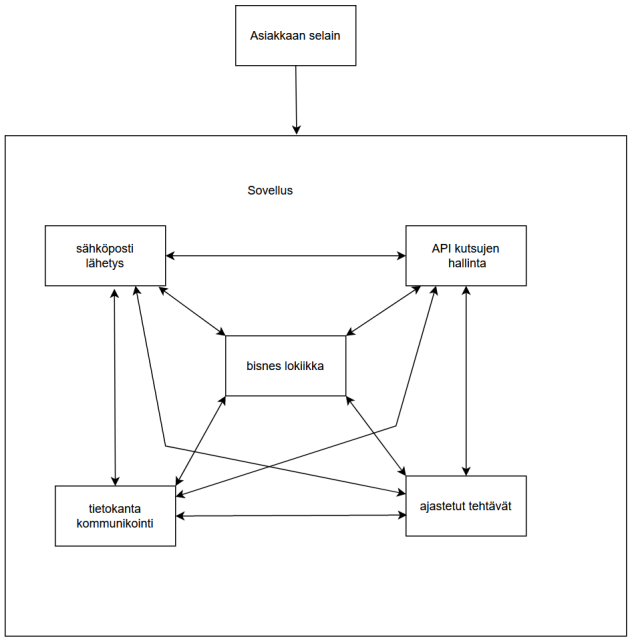
Toisena teknisenä velka on tahaton. Tuotteen tekijöillä ei ole tarpeeksi ymmärrystä käyttämistään työkaluista, ja kehittäjä tekee parhaimman päätöksensä sen hetkisillä tiedoillaan. Kehittäjä aiheuttaa teknistä velkaa tietämättään liian vähäisellä ymmärryksellään. Esimerkkinä, henkilö luo duplikaattisia koodiosuuksia ohjelmaan, koska hänellä ei ole tarpeeksi osaamista luodakseen parempaa koodia.

Kolmas teknisen velan muodostumana on mätäneminen, jota tapahtuu paljon nykyaikaisessa yhteiskunnassa. Monet tuotteet tarvitsevat jatkuvia päivityksiä, sillä erityisesti tietoturvallisuutta pitää päivittää ja nämä tarvitsevat myös alati ylläpitoa. Mätäneminen tapahtuu, kun unohdetaan järjestelmän ylläpitäminrn. Mätäneminen myös pahentuu mitä kauemmin ylläpitoa viivytetään [10]

Tekniset velat voidaan myös luokitella erilaisiin tyyppeihin. Tyypittäminen auttaa arvioimaan kuinka kriittinen tekninen velka on. Samalla pystytään arkistoimaan tekninen velka erin laisiin kategorioihin, jotka auttavat hahmottamaan mitä teknistä velkaa on järjestelmässä. Käymme läpi yleisimmät teknisen velan tyypit mitä voi ohjelmisto kehitys projekteissa muodostua.

Arkkitehtuurinen velka on huonoja ratkaisuja arkkitehtuuriin ilman selkeää visiota tai suunnitelmaa. Tämä tarkoittaa, että kommunikaatiota tietokannan ja muun ohjelmiston kanssa ei ole tehty hyvin. Tällöin mahdollinen ohjelman ominaisuuksien laajentaminen vaikeutuu. Toisena esimerkkinä on monoliittinen arkkitehtuuri, jolloin yksi ohjelma hoitaa kaiken. Tässä ei synnynnäisesti ole ongelmaa. Ongelmia kuitenkin ilmaantuu, jos järjestelmää halutaan tulevaisuudessa muuttaa. Muutosten tekeminen tulee olemaan vaikeata monoliittisen arkkitehtuurin takia.

Monoliittinen arkkitehtuuri ei ole hyvä, jos ohjelmalta toivotaan jatkuvia muutoksia. Nämä muutokset monoliittisessa arkkitehtuurissa voivat rikkoa ohjelman. Tämän myötä koko ohjelma voi olla poissa käytöstä satunnaisia aikoja. Muutoksia tehdään hyvin hitaasti, koska pitää olla varma, ettei muutos aiheuta uusia ongelmia tuotannossa. Vääränlaisen arkkitehtuurin vaihtaminen voi olla hyvin vaikeaa tai mahdotonta. Kuvassa 2.2 on hyvä esimerkki monoliittisesta arkkitehtuurista. Jos muutamme tietokannassa kyselyä. Tietokannalla on kytköksiä muihin ohjelman osiin. Vaikutukset tietokantaan voivat säteillä muihin ohjelman osiin. Syntyy ei haluttuja vaikutuksia ohjelman muihin osiin. Ilman kehittäjän tietoa. Pahimmassa tapauksessa muutos rikkoo tietokannan, jolloin ohjelman muutkin toiminnot voivat lakata toiminnasta. [11] [12]



Kuva 2.2: Monoliittinen arkkitehtuuri

Testivelka on riittämätön määrä testejä ohjelman toiminnasta. Hyvässä ohjelmassa testit kattaisivat suurimman osan tuotteen tärkeimmistä ominaisuuksista. Tällöin voidaan todeta nopeasti ohjelmistossa tapahtuvia virheitä, kun kehittäjä tekee muutoksia tuotteeseen. Samoin muutosten toteaminen toimiviksi myös hel-

pottuu. Liian vähäinen ymmärrys tuotteesta voi haitata testien luomista, ja kasvattaa testivelkaa. Tällöin kattavuus laskee ja ongelmien löytäminen hidastuu. Samalla niiden korjaaminen vaikeutuu ja hidastuu. Testeillä on esimerkiksi hyvä testata rajatapauksia. Mitä tapahtuu, kun järjestelmä saa vääränlaista tietoa tai hyvin laskennallisesti vaativaa tietoa. Tällaiset asiat on hyvä testata hallitussa ympäristössä. Hallitussa ympäristössä pystytään todentamaan järjestelmän toimivan oletetusti ilman, että vaarannetaan tuotantoympäristöä. Näiden puuttuminen hidastaa selkeästi muutosten tekoa ja niiden julkaisemista käyttäjille. Pahimmassa tapauksessa helposti testattavat asiat huomataan vasta tuotannossa. Ongelmien huomaaminen vasta tuotannossa voi aiheuttaa hyvin vaarallista komplikaatiota. Vaikutukset voivat olla pienestä toiminnallisuuden hidastumisesta jopa järjestelmän toiminnallisuuden pysähtymiseen. Tätä riskiä saadaan vähennettyä tekemällä kattavia testejä, jotka testaavat suurimman osan järjestelmän toiminnallisuudesta ja käyvät läpi mahdolliset tapaukset, joita voisi ilmetä. Suositeltavaa olisi tehdä testit ennen ominaisuuden tekoa tai heti sen jälkeen.[13]

Dokumentaatiovelka Tapahtuu, kun unohdetaan dokumentoida tuotteen toiminnallisuus. Miten hoidetaan yleiset ylläpidolliset toimenpiteet tai miten toimita hätätilanteissa. Usein kehitys vaiheessa dokumentaatio suoritetaan velkana ja se tehdään vasta tuotteen kehityksen lopussa. Silloin kuitenkin dokumentaatio vaikeuttuu huomattavasti, koska kehittäjä ei mahdollisesti enää muista mitä kaikkea piti kirjoittaa. Paras ratkaisu olisi dokumentoida samalla hetkellä, kun muutoksia tapahtuu. Huono dokumentti tulee hidastamaan uusien työntekijöiden integrointia tuotteen kehitykseen. Lisäksi tärkeitä tietoja voi kadota esim. hyvin olennaiset tunnukset, mikäli ne ovat vain tiettyjen kehittäjien muistissa. Toinen mahdollinen dokumentaatioon liittyvä velka on sen ylläpidosta aiheutuva. Ylläpitämätön dokumentaatio alkaa mätänemään. Mädäntynyt dokumentaatio voi olla turhaa tai jopa haitallista, kun tämä ei ole ajan tasalla. Mahdollinen ratkaisu dokumentaatioon liittyviin

ongelmiin on hankkia työntekijä, joka keskittyy dokumentien tekoon.[14]

Vaatimusvelka muodostuu usein, kun tuote tehdään hätäisesti ja julkaistaan nopeasti. Tällöin voi syntyä ominaisuuksia, jotka eivät ole haluttuja tai ne eivät tuo lisäarvoa tuotteelle. Tällainen voidaan välttää maltillisella iteroinnilla, jossa ominaisuuksia testataan loppukäyttäjien kanssa. Parasta kuitenkin olisi, että loppukäyttäjän tarpeet on huomioitu alusta alkaen. Esimerkkinä voimme käyttää kuvaa 2.2 sovelluksesta. Asiakas on halunnut tuotteeseen mahdollisuuden lähettää sähköpostia käyttäjille. Kukaan ei kuitenkaan käytä sitä, sillä käyttäjät eivät näe sitä tarpeelliseksi.

Suunnitteluvelka syntyy, kun kehittäjät päättävät luopua optimaalisesta suunnittelu ratkaisusta. Tämä voi myös syntyä, kun henkilöllä ei ole tarpeeksi ymmärrystä hyvästä suunnitelmasta. Tämä voi näkyä esimerkiksi huonosti navigoituvana sivustona, kun värisokean tarvetta kovaan kontrastiin ei olla otettu huomioon. Toinen suunnittelu velan muoto syntyy, kun kehittäjä luopuu lupaamistaan ominaisuuksista ja julkaisee tuotteen vajavaisena. Tämänlainen päätös tehdään usein uskosta, että tästä syntyvät tappiot ja huono maine ovat hyväksyttäviä. Esimerkkinä tällaisesta on Warcraft 3 Reforged. Sen kehittäjä Blizzard ei tuottanut suurinta osaa mainostamistaan ominaisuuksista. Syynä tähän voi olla häviöiden minimoiminen tai usko, että yhtiö voi myydä keskeneräistä tuotetta voitolla [15].

Koodivelka Muodostuu hyvin monesta eri tekijästä. Tekijällä on heikko osaaminen käytettävästä teknologiasta luodessaan tuotetta. Esimerkiksi nimiavaruus koodissa on hyvin kryptistä ja vaikea selkoista. Tämä on huonoa koodia, joka rikkoo yleisesti pidettyjä hyviä tapoja koodatessa. Muita huonoja tapoja on esimerkiksi vaikeasti ymmärrettävää koodia. Koodivelka muodostuu myös liian vähäisestä ajasta koodin suunnitteluun tai ymmärtämään miten koodi toimii. Huonot tai nopeat ratkaisut eivät ole suoraan haitallisia tai vaarallisia. Heikon tai nopeasti luodulla koodilla voidaan saada tuote mahdollisimman nopeasti markkinoille ja tuottamaan

rahaa luojille. Nopeus tuotteen julkaisussa on tärkeää, koska useimmiten ensimmäinen markkinoille ehtinyt tuote saa vallattua siitä eniten. Tällainen taktiikka on hyväksyttävää, jos velka maksetaan myöhemmin. Esimerkkinä huonosta koodista otos 1 kärjistetyssä otoksessa näkyy erittäin huono nimiavaruus. Luokan, muuttujien ja funktioiden nimet ovat korkeintaan kolmekirjaimen pituisia. Niiden toiminnallisuutta on hyvin vaikea ymmärtää lukemalla nimeä. Kehittäjä joutuu käyttämään hyvin paljon aikaa selvittämään mitä kaikki osat tekevät. Tällöin kehittäjä uhraa paljon aikaa ja henkisiä voimavaroja selvittämään toiminnallisuuksia, jotka eivät mahdollisesti ole tärkeitä hänelle. Korjatussa otoksessa 2 pystytään huomaamaan, että luokka on mitä todennäköisimmin shakin lähettinappulalle tarkoitettu ja funktiot ovat nappulan liikkumiseen. Tämän kaiken pystyy kehittäjä nopealla analysoinnilla saamaan selville, kun nimiavaruus on tehty hyvin. Kehittäjän henkiset voimavarat säästyvät. Kehittäjä tehokkaasti paikallistaa onko hänen tarvitsema tieto täällä ja tarvitseeko luokkaan tehdä muutoksia.

Hypoteesi K3:lla on suuret mahdollisuudet realisoitua koodivelan takia. Teknistä velkaa on hyvin helppo muodostaa, jopa vaivihkaa ilman kehittäjän huomaamatta. Ilman, että tekisimme kehityksen luovutusta. Toisaalta tekninen velka voi myös korreloida K2 kanssa, koska liian suuri tai hallitsematon tekninen velka voi selkeästi hidastaa kehitysprosessia. Samalla tämä korreloi K1:n kanssa, koska mahdollinen haasteet mitä tekninen velka voi tuoda voivat myös pakottaa granulamaisiin muutoksiin. Esimerkiksi arkkitehtuuri velkojen takia ei voida tehdä suuria muutoksia järjestelmään nopeasti. Samalla teknisen velan aiheuttama rasite aiheuttaa pelkoa kehittäjissä. Tällöin kehittäjä alkaa epäröimään muutoksia ja yrittää, jopa lykätä niitä. Teknistä velkaa on tunnetusti hyvin vaikea mitata empiirisesti tai tarkasti, kuitenkin nykyään on kehitetty paljon staattisia koodianalyysi ohjelmia, jotka pystyvät antamaan suuntaa näyttäviä tilastoja. Ohjelmat ovat suuntaa antavia, koska on vaikeaa arvioida oikeaa aikaa ongelman ratkaisemiseen. Kaikki kehittäjät ovat

erin laisia ja ratkaisevat ongelmia eri tavalla ja nopeudelle. Samalla kehittä voi kehittyä ajan kanssa, jolloin tulos voi olla ihan erin lainen ensi kerralla. Tällaiset vaikeuttavat tarkkaa arviointia. Toisena rajoittavana tekijänä näissä ohjelmissa on mitä ne pystyvät näyttämään. Ohjelmalla pystytään varmistamaan kolmannen osapuolen kirjastosta tullut uusi versio. Mutta se ei pysty tietämään onko funktion nimi harhaan johtava, kun tutkailemme funktion toiminnallisuutta.

2.2 Ohjelmistokehitysprosessi

2.2.1 Elinkaari

Ohjelmistokehityksen elinkaari kuvaa kehityksen eri vaiheita, joita se tulee käymään läpi sen edetessä. Nämä merkitsevät selkeää saavutusta ja edistystä projektista. Kaikilla ohjelmistojärjestelmillä on elinkaari ja ohjelmiston elinkaari pystytään pilkkomaan eri vaiheisiin. Elinkaaren vaiheita on vaihtelevasti riippuen projektista ja mahdollisista määrittelyistä. Samalla ei ole yhtään täydellistä vaiheita kuvaavaa esittelyä, joka olisi hyväksytty totuus.[16] [3]

Kaikkia yleisimpiä vaiheita ei jokaisessa ohjelmiston elinkaaressa ole, mutta yleensä kaikissa on kolme vaihetta. Ensimmäinen vaihe on kehitys, jonka voi määritellä muutamaan osaan: Asiakkaalta kerätään tietoja ja kohdeyleisön tarpeita. Näistä pystytään hahmottamaan tarvittava tuote, jonka asiakas tarvitsee. Tämän myötä tehdään alustava teknologinen suunnitelma, miten asiakkaan tarpeet saadaan tyydytettyä parhaiten. Alustavien suunnitelmien jälkeen aloitetaan tuotteen luominen. Tässä vaiheessa siirrytään usein erityyppisiin kehittämisen tyyleihin mm. vesiputous, scrum tai devops. Näistä tyyleistä myöhemmin lisää. Kehityksen aikana luodaan suurin osa olennaisista osista, jotka ovat tarpeellisia tehostamaan tuotteen kehitystä. Näitä ovat kehitysputket, testi ympäristöt jne. Kehityksen lopussa tuote on valmis julkaistavaksi kohdeyleisölle. Kehityksen kesto on hyvin vaihtelevaa.

Se on oleellisen riippuvainen omistajan investoinnista ohjelmaan, sillä tuote ei hanki näitä investointeja takaisin ennen julkaistua. Tämän kehitysvaiheen voi rajata aikaan, jolloin tuotetta ei ole vielä julkaistu markkinoille. Sitä myös suunnitellaan edelleen suljetussa ympäristössä, jossa käyttäjät eivät näe sitä. Usein tuote siirtyy seuraavaan vaiheeseen, kun tuote julkaistaan käyttöön.

Toisessa vaiheessa tuote siirtyy ylläpitoon. Tuote usein siirtyy ei täysin valmiina tuotteena ylläpitoon, koska kehityksen aikana on hankittu teknistä velkaa nopeuttaakseen tuotantoa. Tässä vaiheessa kommittien määrä selkeästi vähenee kehitykseen nähden. Vaiheen aikana suurimmaksi osaksi maksetaan takaisin tekninen velka korjailemalla koodi ja ylläpitämällä tarpeellisia osia ja päivittämään vanhoja osia. Mahdollisesti suuriakin muutoksia tehdään mutta harvinaisempaa. Ylläpito vaihe voi kestää hyvinkin kauan riippuen kuinka hyvin tekninen velka on saatu maksettua, ohjelman kyvystä muokkautua tulevaisuudessa tarvittaviin muutoksiin ja tärkeydestä omistajalle. Hyvä esimerkkinä pitkä aikaisestä järjestelmästä on Google.

Kehitys ja ylläpito ovat hyvin lähellä toisia. Oikeastaan kummassakin voidaan käydä kehityksen elinkaaren vaiheet suunnittelusta, ylläpitoon. Suurempana erona on, että kehityksessä lähdetään usein mahdollisesti tyhjältä pöydältä tai kehitetään uutta asiaa. Ylläpidossa meillä on jo tuote, joka on todettu toimivaksi ja enää pitää mahdollisesti varmistaa sen toiminta nyt ja tulevaisuudessa. Huolehtia mahdollisista kehityksistä mitä halutaan tehdä. Usein tällaiset kehitykset tarkoittavat ajan kanssa huomattuja asioita, joita ei nähty kehitys vaiheessa. Myös mahdollisia muutoksia ympäristössä mihin ohjelman pitää vastata. Hyvänä esimerkkinä EU:n laki muutos GDPR.

Kolmannessa vaiheessa tuote siirtyy eläköitymiseen. Ohjelman on palvellut tarpeensa, tai sen korvaa toinen tuote. Tuotteen käytön lopussa alustavasti sammutetaan tuotteen käyttö kokona, jolloin kukaan ei pääse enää käyttämään sitä. Tämän jälkeen poistetaan kaikki osat käytöstä kokonaan esimerkiksi. poistetaan AWS:ssä

pyörivät instanssit ja muut tarpeelliset asetukset, joita on tuotteen kanssa käytetty. Kaikki mikä ei ole tarpeellista poistetaan mutta tehdään myös, arvio onko vanhan projektin tiedoissa jotain, mikä on arkistoimisen arvoista, kuten järjestelmien pystytys aws:ään tai kehitysputkien pystytys. Tällaiset tiedot on hyvä säilyttää tulevia projekteja varten, jolloin voidaan nopeuttaa uuden tuotteen kehitystä. [17]

Kolmeen kuvailemaani vaiheeseen sisältyy myös hyvin paljon muita asioita. Kolmen vaiheen kuvailu on iso yleistys kaikista projekteista. Se on pätevä mutta jokaiseen vaihe voi olla hyvin erin lainen projekteista riippuen. Syvällisemmässä vaiheiden kuvailussa käytänkin Yhdysvaltojen oikeusministeriön tekemää määrittelyä ohjelmiston elinkaaren vaiheista, koska se antaa kattavan kuvailun mitä vaiheita ohjelman elinkaaressa voi olla. Silti tämäkään määrittely ei ole kaikkea kattava. Mutta antaa hyvää osviitta mitä voi olla.

Alullepano Kehittäjä tai sijoitta huomaa, että mahdolliselle idealle on tarvetta tai pystytään tuottamaan arvoa. Tällöin idea voi olla vain hyvin epämääräinen idea henkilön päässä. Vaikka henkilö tajuaa, että olisi tarvetta nopeuttaa ja helpottaa laskujen maksamista. Usein monet ideat pysähtyvät, jo tässä vaiheessa, koska monet idea jäävät vain ajatuksiksi eikä niitä aleta toteuttamaan. Tarvitaan myös halua lähteä tekemään. Hypoteeseihin nähden tämän ei pitäisi vaikuttaa tutkimukseen. Vaiheen idea on enemmän, että sitä kannattaisi tutkia. Idea lähdetään tutkimaan tai se unohdetaan.

Konseptin suunnittelu Idea on nähty järkevänä ja päätetty, että sitä kannattaa tutkia. Määritellään mitkä ovat konseptin rajat. Mikä on ohjelman tavoite. Mitä sillä yritetään saavuttaa. Tavoitellaanko ohjelmalla laskujen maksujen nopeuttamista tekemällä se elektronisesti. Mihin se pystyy. Luomaan elektronisesti laskuja ja lähettämään ne laskun osapuolille. Mihin ohjelma ei pysty. Toistamaan videoita. Mikä on mahdollinen budjetti. Arvioida kuinka kauan tuotteella tulee kestämään ennen kuin se on julkaisu kunnossa. Arvioidaan taloudelliset kulut mitä konseptin

toteutuksesta tulisi esimerkiksi ohjelmoijille maksettavat palkat. Mahdolliset riskit mitä voi ilmetä konseptin toteutuksessa. Kuinka todennäköisesti tuote tulee valmistumaan. Minkälaisia tappioita tulee, jos tuote ei valmistu. Arvioida mahdollisia voittoja. Miten paljon järjestelmä pystyy tuottamaan rahaa tekijöille, jos ohjelman tarkoitus on tuottaa rahaa tekijöille. Miten konsepti pitäisi toteuttaa, ja kuinka mahdollinen on toteuttaa tämä konsepti näissä rajoissa. Suurissa projekteissa nämä määritelmät voivat muuttua ohjelman elinkaaren aikana, koska kaikkia asioita ei ole helppo suunnitella hyvin suuressa projektissa. Tällaiset määritelmät ja dokumentit on hyvä pitää tallessa. Ne auttavat seuraavia henkiköitä, jotka liittyvät projektiin ymmärtämään mitä halutaan. Hypoteesien puolesta. Nämä olisi hyvä säilyttää, koska näiden puuttuminen voi vaikeuttaa tulevaisuudessa ohjelman hahmottamista. Varsinkin, jos ohjelmisto luovutetaan uudelle kehittäjälle, eikä hänelle anneta pohjustavaa ymmärrystä ohjelmasta ja mitä siltä on vaadittu. Samalla pystytään varmistamaan, että ohjelma toteuttaa edelleen sen mitä halutaan. Näiden puuttuessa uuden kehittäjän kehitys tulee hidastumaan ja hankaloitumaan.

Suunnittelu Suunnitellaan miten tullaan toimimaan projektin kanssa. Mitä kaikkea tarvitaan, että projekti tulee toteutumaan vaaditussa aika rajassa ja budjetissa. Arvioidaan mahdolliset tietotuvallisuus vaatimukset ja mihin se johtaa esimerkiksi tarvittaviin sertifikaatteihin tai salattuihin asiakas tiedot. Tästäkin tehdyt suunnitelmat on tärkeä dokumentoida. Ne antavat pohja ymmärrystä projektista uusille kehittäjille. Hypoteesien puolesta K1 ja K2 voi suunnittelun dokumentoinnin puuttuminen haitata. Suunnittelu dokumenttien puuttumien heikentää uusien kehittäjien mahdollisuuksia ymmärtää paremmin projektia. Tällöin kehittäjä tekee muutoksia hitaammin, koska hän joutuu käyttämään enemmän aikaa lähdekoodin tutkimisessa. Tämä on erittäin aikaa kuluttavaa. K2 voi realisoitua dokumenttien syyn myötä.

Vaatimusten analysointi Käyttäjien vaatimusten analysointi. Miten tehokas

järjestelmän tarvitsee olla. Tarvitseeko järjestelmän käsitellä sata asiakasta päivässä vai miljoona. Minkälainen tietoturvallisuus järjestelmällä pitää olla. Käsitteleekö se julkisia tietoja vai salassa pidettäviä tietoja, kuten henkilöllisyys tunnus. Arvioida mahdolliset ylläpito vaatimukset. Pitääkö järjestelmän olla ympäri vuorokauden saavutettavissa, vai tarvitseeko sen olla vain arkipäivisin toiminnassa. Ongelmia voi muodostua, jos tietoa ei ole tarpeeksi tai tulkittu väärin. Vaatimukset on hyvä dokumentoida. Tiedot ovat hyödyllisiä luovutuksien yhteydessä tai uuden kehittäjän tuomisessa projektiin. K1 ja K2 voivat realisoitua samanlaisesti, kuin suunnittelu dokumenteissa. Samalla K3 kolme voi realisoitua tässä kysymyksessä. Vaatimus dokumentit ovat tärkeitä mutta ne pitää ajan tasalla. Muuten muodostetaan dokumentaatiovelkaa. Samalla huonosti ylläpidetty dokumentaatio voi hidastaa tai haitata ohjelmisto kehitystä. Tällöin taas kerran vaatimus dokumenteilla on vaikutusta K1 ja K2 hypoteeseihin.

Mallinnus Suunnitellaan miten järjestelmä tulee oikeasti toimimaan. Miten arkkitehtuurillisesti ohjelmisto toteutetaan ja mitä teknologia ratkaisuja tehdään järjestelmän toteuttamiseksi. Miten logiikka tulee toimimaan järjestelmässä ohjelmisto tasolla. Mitä ohjelmointi kieliä ja työkaluja tullaan käyttämään. Suurissa projekteissa tällaisen laatiminen heti alussa on hyvin vaikeaa, tai jopa mahdotonta. Suositeltavaa olisi tehdä järjestelmän osia inkrementaalisesti ja mallinnuksia tehdään osissa. Tällöin on hyvä pitää dokumenttia, jotta tiedetään mitä on mallinnettu ja mitä puuttuu. Kohtaamme saman ongelman dokumentteja tehdessä, mikä oli aikaisemmissakin vaiheissa. Dokumenteilla on vaara vanheta. Tällöin K3 hypoteesi realisoituu. Samalla vanhentunut dokumentaatio voi hidastaa kehitystä tai pakottaa kehittäjä tekemään pienempiä muutoksia. Tällöin K1 ja K2 realisoituu. Parhaiten pystytään ongelmat välttämään säännöllisellä dokumenttien päivityksellä ja niiden läpi käynnillä.

Kehitys Järjestelmän kehitys. Luodaan toimiva järjestelmä luodusta suunnit-

telu vaiheen tehdyistä vaatimuksista. Sisältää myös esimerkiksi pilvi ympäristöjen asennuksen ja kehitys putkien pystytyksen. Testien luominen varmistaakseen ohjelmiston toimivuus kehityksen aikana. Integroida järjestelmä toimimaan tarvittavien ulkopuolisten järjestellmien kanssa esimerkiksi pankkien kanssa hoidettava maksu tunnistautuminen. Tietokannan pystytys ja toimivuuden testaaminen. Tekninen velka mitä todennäköisemmin muodostuu tässä vaiheessa, kun halutaan nopeuttaa kehitystä prosessia tekemällä ei niin optimaalisia ratkaisuja koodia luodessa. K3 kysymykseen nähden tässä vaiheessa luotu tekninen velka voi tulla haitta tekijäksi myöhemmin luovutuksissa vastaa, jos velkaa ei olla maksettu tai sitä ei olla dokumentoitu. Usein kehityksen alussa dokumentaatio jää vähälle, joten kehityksen aikana tehty tahallinen tai tahaton tekninen velka voi hyvinkin unohtua ja realisoitua vasta luovutuksessa. Uusi kehittäjä ei tiedä lähdekoodia läheskään niin hyvin kuin sen luoja. Tällaisten ongelmien kohdalla kehitys tulee hidastumaan, jolloin K2 kysymys tulee realisoitumaan. Mahdollinen K1 kysymyskin realisoituu, koska hidastumisen takia kehittäjä tekee helppoja ja pieniä muutoksia mitä asiakas vaatii. Mutta isommat muutokset vaativat pitkä aikaista työskentelyä ja ymmärrystä. Mahdollisesti, jopa pienetkin muutokset vaativat paljon työtä, koska tekninen velka vaikeuttaa ohjelmisto kehitystä. K1 pohtien alussa, kun aloitetaan tyhjältä pöydältä, kehitys on nopeaa ja iso muutoksia tehdään jatkuvasti, koska ei ole aikaisempaan pohjaa järjestelmässä mitä pitää ottaa huomioon, jolloin kehitys tapahtuu nopeasti. K2 kysymyksessä tilanne on sama. Kehityselinkaari on nopea ainakin alku vaiheessa. K1 ja K2 tulevat hidastumaan mitä pidemmälle kehitys etenee. Syynä on kasva monimutkaisuus järjestelmässä ja aikaisempi koodi mikä pitää ottaa huomioon, kun tekee uutta koodia.

Integraatio ja testaaminen Järjestelmän integrointi tarpeellisiin osiin varmistetaan käyttäjän avulla. Järjestelmän ominaisuudet testataan käyttäjän kanssa. Toimivuuden tulee täyttää tarvittavat vaatimukset, jotka on aikaisemmissa vaiheis-

sa dokumentoitu. Tätä voitaisiin kutsua kenraali harjoitukseksi, jossa ympäristö ja toiminallisuus on identtinen tuotantoon nähden mutta ei olla vielä julkaistu asiakkaiden käyttöön. Ei vaikutusta tutkimuksen hypoteeseihin.

Toteutus Kehityksen luoma järjestelmä viedään tuotantoon. Järjestelmä todetaan olevan valmis vietäväksi tuotantoon. Järjestelmä tai siihen tehnyt muutokset julkaistaan kuluttajille. Ei tarkoita, ettei kehitystä tai päivitystä lopetettaisiin. Ainoastaan että ohjelma on tarpeeksi hyvä, että sitä voidaan alkaa käyttämään. K2 ja K1 hypoteesiin peilaten. Kehitys tulee tosiaan hidastumaan tämän jälkeen, koska pääpainote järjestelmässä ei mahdollisesti enää ole kehitys vaan enemmän ylläpitoa, johon kuuluu hidasta kehitystä. K3 voi alkaa tämän jälkeen ensimmäisen kerran tulla haasteeksi, koska usein tuotantoon viennin jälkeen usein tehdään ensimmäinen luovutus.

Ylläpito Järjestelmän on julkaistu käyttäjille. Järjestelmän toimintoja valvotaan. Arvioidaan se toimivuutta tuotannossa ja arvioidaan, onko mahdollista korjata tai parantaa järjestelmän toimivuutta. Löydettäessä mahdollisia muutoksia siirrytään takaisin suunnittelu vaiheeseen. Kehitystä voi edelleen jatkua mutta sen lisäksi ohjelma on asiakkaiden käytettävänä. Tällöin sitä pitää valvoa ja tehdä tarvittavia korjauksia. Usein tässä kohtaa luovutus voi tapahtua kehitys tiimiltä ylläpito tiimille, jos tällaisia ryhmiä on yhtiössä. Mahdollisesti kehitys vaihtuu myös enemmän ylläpitoiseen toimintaan. Tämä ei tarkoita, ettei voisi olla kehitystä. Mutta kehityksen nopeus hidastuu selkeästi ja resursseja allokoidaan enemmän järjestelmän tarkkailuun ja kriittisten ongelmien korjaamiseen tuotannossa. Vaikutukset K1 ja K2 hypoteesien pitäisi tässä kohtaa toteutua mutta se on oletettavaa, koska keskityskin muuttuu. Tutkimuksellisesti tätä kohtaa on vaikea tutkia. Kehitys hidastuu mutta mikä on hyväksyttävä määrä tai liian paljon hidastumista on vaikea sanoa. K3 nähden tämä on erittäin mielenkiintoinen kohta. Usein tässä kohtaa aletaan katsomaan teknistä velkaa. Ylläpidossa aloitetaan mahdollisesti ensimmäistä kertaa korjaamaan

teknistä velkaa. Huonossa tapauksessa tämä on se kohta, jossa voidaan myös alkaa kärsimään otetusta teknisestä velasta. Luonnostaan kehitys hidastuu tässä kohtaa, joten on vaikea arvioida mikä luonnollista hidastumista, ja mikä on teknisestä velasta. Ylläpidon kohdalla pystytään hyvin kuitenkin arvioimaan ensimmäisen kerran, miten hyvin ollaan, pystyt pitämään tekninen velka kurissa ja dokumentoimaan muodostunut tekninen velka.

Hävitys Hallittu järjestelmän lopetus. Tuotteella ei ole enää tarvetta käyttää. Järjestelmä on tehnyt tarvittavan asia. Sitä ei ole enää taloudellisesti kannattavaa ylläpitää. Uusi järjestelmä on korvannut vanhan. Järjestelmä sammutetaan hallitusti. Varmistetaan, että dokumentoidaan tarvittavat tiedot, joita voidaan tarvita tulevaisuudessa. Samalla varmistetaan, että kaikki arkaluonteiset tai salassa pidettävät tiedot on turvallisesti tuhottu. Tällaisissa tilanteissa voi tulla vaikeuksia, jos ei ole hyviä dokumentteja. Vaikutukset eivät ole ehkä suoraan näkyvissä. K3 muodostama tekninen velka on voinut aiheuttaa, että kaikkia järjestelmän osia ei ole tiedossa. Tällöin mahdollinen hävitys voi viivästyä merkittävästi, joka johtaa K2 hypoteesiin. Pahimmassa tapauksessa servereillä jää pyörimään käyttämättömiä servereitä, joissa on arkaluonteista tietoa.

Ohjelmistokehityksen elinkaaressa usein pystyy löytämään nämä vaiheet. Mutta erovaisuuksia on paljon ja malleja, joilla yritän hallita elinkaarta keskittyvät eniten suunnittelusta ylläpito alueeseen. [17] [3]

2.2.2 Mallit

Ohjelmistokehityksen tehostamiseksi on suunniteltu erilaisia malleja ja käytänteitä, joilla on pyritty paremmin tekemään nopeaa ja hyvä laatuisia tuotteita. Elinkaari mallit kuvailevat abstraktilla mallilla ihmisten ja organisaation tekemiä päätöksiä ohjelmistoa tehdessä. Samalla malleilla yritetään ohjeistaa työskentely prosessia. Näillä yritetään saada luotua tehokkaampia ja parempia ohjelmistoja luomalla oh-

jeistus, miten ohjelmiston elinkaari etenee. Tunnetuimpia ja yleisempiä malleja on vesiputouksen, inkrementaalisen ja iteratiivisen mallit, ja niiden variaatiot. Keskitymme iteratiivisiin malleihin, koska tutkittavassa kohteessa on käytetty iteratiivisia ja inkrementaalisen mallien hybridillistä yhdistelmää.[3]

Ensimmäisiä tunnettuja julkaisu ohjelmisto kehitys malleja on 1950 luvulta. Tällöin ensimmäinen julkaistu kehitys malli oli vesiputous malli.[18]

Iteratiivinen työskentely konsepti ensimmäisen kerran esiteltiin 1930-luvulla. Walter Schewart ehdotti lyhyitä syklejä, jotta voitaisiin tehdä parannuksia, jokaisen pienen muutoksen jälkeen. Walter ehdotti inkrementaalista työskentelyä lentokoneiden kehitys työhön. 40 ja 50-luvulla iteratiivista mallia käytettiin onnistuneesti erilaisiin projekteihin muihin vaativiin isoihin projekteihin. Iteratiivista kehitys mallia käytettiin ensimmäisen kerran ohjelmisto kehityksessä 1960-luvulla. Iteratiivisen mallien yleistyminen ei kuitenkaan yleistynyt ohjelmistokehityksessä. Suosioon nousi 1970-luvulla vesiputousmalli. Vesiputous mallin luoja oli iteratiivisen mallin kannattaja ja sen suosija. Valitettavasti hänen mallinsa ymmärrettiin väärin. Tämä johtui myös osittain luojaan asetetuista rajoituksista mallia tehdessä. Hän teki mallin valtion prosesseja myötäillen. Vesiputous malli tuli suosituksi virallisten julkaisujen ansiota. Pitkään ohjelmisto maailmalla yleisin käytetty malli oli vesiputous. Iteratiivista mallia alettiin suosimaan 2000-luvulla. Tällöin alettiin myös kehittää ja muunnella iteratiivista mallin pohja ideaa. Syntyi ketteriä malleja, joilla yritettiin paremmin vastata nykyajan kiihtyvää tarvetta vasta nopeaan ja hyvä laatuiseen ohjelmisto kehitykseen. Näissä monesti käytettiin ketterän manifestilla ja inkrementaalisen mallin ideologeja.[19]

Iteratiivisessa elinkaaressa ideana on tehdä kehitys sykleinä. Tehdään yhdessä syklissä toimiva kokonaisuus, tai toimiva muutos projektiin. Lopputulosta arvioidaan ja tutkitaan. Tämän jälkeen aloitetaan uusi sykli, jossa aloitetaan suunnittelusta ja mennään aina ylläpitoon. Ohjelmisto kehityksen elinkaaressa voi olla useita

syklejä. riippuen aina projektin koosta. Yhden syklin ideana olisi, kuitenkin olla lyhyt noin yhdestä viikosta kymmeneen viikkoon. Määrä vaihtelee nykyään monet mallit pitävät näitä suosittelevat pienempiäkin syklejä mutta alkuperäisessä iteratiivisessa mallissa suositeltiin 10-1 viikkoa. [19]

Ketterä kehitys oli 2000-luvun alussa kehitetty työskentely malli- Tavoitteena oli luoda tekniikoita, joiden avulla saataisiin nopeammin kehitettyä projekteja halvemmalla hinnalla. Alkuperäisenä ideana oli luoda parempi vanhoille kehitys strategioille. Yleisesti käytetty vesiputous malli oli raskas toteuttaa. Malli ei tuottanut haluttua lopputulosta. Syynä usein tähän oli vesiputousmallin vaikeus muuttaa kehityksen suunnitelmia nopeasti. Tarvittiin malli, joka nopeammin ja paremmin pystyy vastaan ottamaan vaihtelua kesken projektin. Kehiteltiin malleja, jotka yhdistäisivät iteratiivisen ja inkrementaalisen mallin osia yhteen. Samalla näihin yritettiin pohjauttaa ketterän mallin ideoita. Ketterän mallin ideana oli keskittyä enemmän muuttuvaan ympäristöön ja asiakkaan mielipiteisiin.

Manifesti syntyi yhteistyönä vuonna 2001 jossa monet ohjelmisto kehityksen suuret nimet loivat toimivan manifestin, jonka tarkoituksena oli luoda pohja kaikille ketterän kehityksen malleille. Manifestissa on neljä arvoa, joita tekijät pitävät eritän arvokkaina ketterässä kehityksessä.[20]

- Yksilöt ja kommunikointi on tärkeämpää kuin prosessi ja välineet.
- Toimiva järjestelmä on tärkeämpi, kuin kattava dokumentaation.
- Asiakkaan kanssa työskentely on tärkeämpää kuin sopimuksesta neuvottelu.
- Joustaminen muutoksissa on tärkeämpää kuin suunnitelmassa pysyminen.

Manifestissa on kuitenkin toimintaperiaatetta, joita suositellaan, että saadaan toimiva ketterä kehitys.

- Suurin prioriteetti on pitää asiakas tyytyväisenä tuottamalla aikaisin ja jatkuvasti arvokasta tuotetta

- Hyväksy vaihtuvat vaatimukset, jopa kehityksen loppuvaiheessa. Ketterän hyödyntää muutosta saadakseen asiakkaalle kilpailu edun.
- Tuota toimivaa ohjelmaa usein. Aikavälinä parista viikosta pariin kuukauteen. Suosien pienempiä aikavälejä.
- Talous ihmiset ja Ohjelmointi ihmisten pitää työskennellä yhdessä päivittäin
- Rakenna projekti motivoituneiden ihmisten ympärille. Anna heille ympäristö ja välineet työskennelle. Luota että he tekevät hyvää jälkeä.
- Tehokkain tapa kommunikoida tai välittää tietoa on keskustella kasvotusten.
- Toimiva ohjelma on pääsääntöinen edistyminen arviointi kohde.
- Ketterä kehitys tukee kestävää kehitystä. Sponsorit, kehittäjät ja käyttäjät pitäisi pystyä saavuttamaan tasainen tahti jatkuvasti.
- Jatkuva huomio tekniseen osaamiseen ja hyvä arkkitehtuuri tehostavat ketterää kehitystä.
- Yksinkertaisuus maksimoi tehdyn työn määrän.
- Parhaat arkkitehtuuri, vaatimus ja suunnitelmat syntyvät itseohjautuvasta ryhmästä.
- Säännöllisten jaksojen välissä tiimi arvioi omaa osaamista ja kehittää itseään näiden tulosten pohjalta.

Ketterän kehityksen pohjana toimii aina nämä säännöt. Silti ketterästä kehityksestä on tullut laaja käsite, jonka alle kuuluu monta erilaista ohjelmistokehitys mallia. Ketterä kehitys määrittelee vain, että tehdään iteratiivisella mallilla. Lyhyissä sykleissä ja pieninä palasina. Esimerkiksi Windowsin sivuilla ketterä kehitys on tehdä työ pienissä palasissa. Pieniä palasia on alettu kutsumaan sprinteiksi, jotka

kestävät useimmiten 1-4 viikkoa. Tässä aikana ominaisuus on luotu, testattu ja viety tuotantoon. Huomaamme kuitenkin, että Windows on tehnyt paljon omia määritelmiä mikä voisi kuulua ketterään kehitykseen esim. teknisen velan ylläpito ja käsittelevät tarkemmin strategioita työntekoon, jotka toteuttavat ketterää kehitystä, kuten scrum. Ketterä kehitys on loppujen lopuksi filosofia miten toimia kehityksessä. Strategiat eivät auta, ellei toteuta ja sisäistä ketterän kehityksen ideologioille. Kehittäjät voivat käyttää ketteriä strategioita, kuten scrummia tai XP(extreme programming) mutta niiden toiminnallisuus on suurimmaksi osaksi turhaa, ellei toteuta ketterän kehityksen filosofiaa.[21]

Ketterä kehityksessä on myös huonoja puolia. Vesiputous malli, jota harrastettiin ennen ei ole synnynnäisesti täysin ongelmallinen. Samoin ketterä kehitys ei ole täydellinen. Enemmänkin kyse on siitä mikä toimii parhaiten tilanteeseen. Ketterä kehitys vähättelee alustavia speksejä vaatimuksista ja suunnitelmista. Ei ole negatiivistä kerätä selkeitä vaatimuksia ennen kehityksen aloittamista. Vaikka nämä voivat muuttua kehityksen aikana. Alustava tutkimus ja sen dokumentit antavat hyvän pohjan ja siihen on hyvä peilata työtä ja nähdä pitkäaikaisempi tavoite. Muita negatiivia osia mitä ketterässä kehityksessä on mm. liiallinen painotus asiakas kertomuksiin tai kieltäytyminen projekti managerista. Liiallinen painotus asiakas kertomuksiin voi ohjata pois alkuperäisistä vaatimuksista. Projekti manageri voi olla täysin toimiva ratkaisu. Varsinkin hyvin suurissa projekteissa, joissa kokonaisuutta on vaikea hahmottaa.[22][21]

Hypoteeseihin nähden ketterän kehityksen haitta puolet voivat tulla haittaamaan luovutuksessa. Nopeasti tehdyt muutokset kehityksessä voivat helposti tuoda teknistä velkaa, ellei sitä hallita. Usein nopeat muutokset kehityksessä voivat myös rasittaa aika taulua ja silloin tehdään ei niin hyvä ratkaisuja. Nämä voivat näkyä vasta luovutuksessa uudelle kehittäjälle, jolla ei ole tietoa näistä. Tällöin kehitys voi hidastu, jolloin K2 toteutuu. mahdollisesti K1 toteutuu myös, koska kehittäjän

voi tulla epävarmaksi ja tehdä pienempiä muutoksia. K3 voi toteutua, koska henkilö voi luoda kehittää eteenpäin huonoa toteutusta, joka vaikeuttaa sen korjaamista tai poistamista. Toisena ongelmana mikä voi kasvattaa hypoteesien todennäköisyyttä on ketterän kehityksen vähäinen arvostus dokumentaatiosta. Se auttaa paljon luovutus vaiheessa, jos suunnitelmista ja visioista on dokumentaatio. Tämä auttaa luovutuksessa uutta kehittäjää paremmin ymmärtämään projektia. Puutuva dokumentti voi hidastaa uuden kehittäjän ymmärrystä projektissa. Tällöin vaikutukset ovat hyvin samanlaiset kuin aikaisemmassakin skenaariossa. Vaikutukset tulisivat näkymään K1, K2 ja K3 hypoteeseissa.

Devops tulee englannin kielen sanoista kehittäjä(development) ja operatiivinen(operative). Devopsin tarkoituksena on tuoda kehitys toiminto ja operatiivinen toiminto lähemmäksi toisiaan. Devopsissa käytetään ketterää kehitystä ja lean periaatteita toteutuksessa. Gene Kim kiteytti devopsin kolmeen periaatteeseen, joiden mukaan ohjelmisto kehityksessä devops tulisi toimia. Periaatteet pitävät sisällään ketterän ja leanin filosofian. Optimoi työnkulku poistamalla kaikki turha. Esimerkiksi selkeät linjaukset miten toimia luomalla raamit, joiden puitteissa toimia esim. scrum. Luo tehokas palaute ketju, jonka avulla kehittäjä pystyy saamaan mahdollisimman nopeasti palautetta tehdyistä muutoksista. Luo kulttuuri, jossa palkitaan kokeilua ja oppimista. Luomalla tällaisen ympäristön palkitsemme luovaa ajattelua, joka auttaa kehittäjiä ratkaisemaan tehokkaammin virheitä. Samalla luomme ilmapiirin, ettei ole vaarallista tehdä tai tulla virheitä. Niiden kautta pystytään oppimaan. Tällöin myös läpinäkyvyys kasvaa kehittäjien välillä. Devopsiin ei ole kuitenkaan tarkkaa määritystä, miten se tarkalleen toteutetaan. Hyvänä tästä on esimerkiksi Wikipedian määritelmä devopsille. Devops on tekniikoita ja tapoja, joilla tehostetaan kehitystä ja julkaisu prosessia. Mikään tyyli ei ole täysin oikein eikä väärin. Käsittelen kuitenkin osa alueet mitkä on normaalisti pidetty devopsin alla.

Yleisin tapa millä luodaan devops näkemystä on luomalla jatkuvalla integraatio

ja toimitus (CI/CD) Kuva 2.3. Jatkuva kehitys (CI) on kehittäjän osuus ja Jatkuva toimitus (CD) on operatiivinen osuus. Ideana on automatisoida iso osa kehityksen vaiheista mitä tarvitsee tehdä.

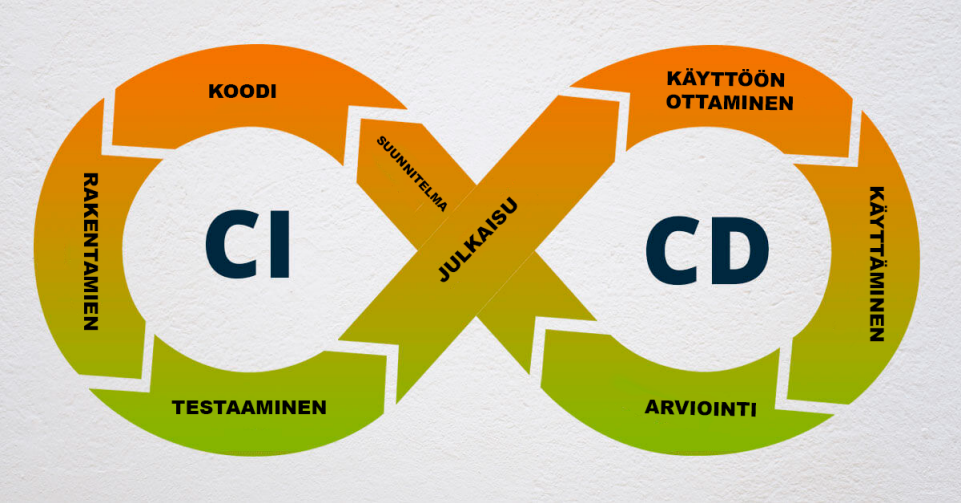
Tällainen jatkuva toistuva ohjelmisto kehityksessä saadaan luotu erilaisilla välineillä. Suunnitelmia pystytään luomaan ja hallitsemaan yksin kertaisilla laitteilla, kuten Google drive tai Trello hallinoidakseen suunnitelmia. Koodissa ideana on luoda toiminnallisuus missä pystymme tekemään hallittavan version hallinnan tekemistä muutoksista. Tällaisia järjestelmiä on muun muassa git. Rakentamisvaiheessa käännämme järjestelmän sopivaksi vietäväksi julkaisuun ja toimivaan muotoon. Esimerkiksi luomme Java ohjelmasta jar tiedoston. Testi vaiheessa teemme mahdolliset erilaiset automaatio testit ja staattiset koodi analyysit, joiden avulla tarkistamme, että tehdyt muutokset ovat hyväksyttäviä ja turvallisia julkaisuun, kun nämä ovat hyväksytty ohjelma siirtyy julkaisuun. Testien luomiseen esimerkiksi Javalla käytetään Junittia. Staattisen koodin analyysiin on käytössä esimerkiksi Sonarqube. Julkaisussa viemme toimivan version pilveen. Pilvessä pystytään hallitsemaan toimivia julkaisu versioita, jotka ovat valmiita julkaistavaksi tuotantoon. Tällaisia välineitä on esimerkiksi Github tai Gitlab. Version julkaisun jälkeen CI/CD putki automaattisesti julkaise käyttäjille uudet ominaisuudet. Toteutuksessa voi muun muassa käyttää AWS pilvipalvelua. Tuote viedään käyttöön otettavaksi AWS pilvipalveluun, jossa se tarjotaan käytettäväksi asiakkaille. Julkaisussa tuotantoon käytetään myös työkaluja, joiden avulla pystytään luomaan jatkuva julkaisu ja automatisoimaan prosessia. Tähän tarkoitukseen käytetään mahdollisena vaihtoehtona ansiblea, chefiä ja terraformia. Lopuksi arviointi putkeen kuuluu mahdollisesti muutosten toiminnallisuuden monitorointi tuotannossa. Tällöin saadaan informaatiota mahdollisista odottamattomista haitoista. Valvonta välineitä on muun muassa Graylog, Cloudwatch tai Splunk.

Saadakseen tehokas ja toimiva devops-ympäristö näille osa alueille kehittäjät

tekevät jatkuvan integraation, josta putki kuljettaa muutokset suoraan automatisoituun julkaisuun. Kehittäjillä on monta erilaisia ohjelmaa, jotka orkestroivat kokoa kehitys putkea esimerkiksi Jenkins. Putken toiminnallisuudessa käynnistyy, kun kehittä julkaisee tekemänsä muutokset koodiin. Putki kääntää ohjelman, testaa, julkaisee version hallintaa ja ottaa käyttöön testi serverillä. Kehittäjä arvioi tarkkailu laitteilla toimiiko muutokset ja julkaisee muutokset automaattisesti putkella tuotantoon. Tällaisella putkella pystytään tehokkaasti luomaan automatisoitu toiminto, jolla saadaan nopeasti tehdyt muutokset julkaistavaksi, testattavaksi. Palaute on nopeaa eikä kehittäjän tarvitse hukata aikaa turhiin odotuksiin tai muistella kaikkia välivaiheita mitä tarvitsee tehdä.[23] [24]

Devops pystyy hyvin auttamaan kehityksen luomisessa. Devopsin avulla pystytään nopeuttamaan kehityksen vaiheita ja dokumentoitua ilman, että tarvitsee pitää erillistä dokumenttia kaikesta. Devops tuottaa yhtenevän toiminnallisuuden, joka auttaa uutta kehittäjää. Ymmärtämään paremmin, miten projekti toimii. Hänen ei tarvitse heti ymmärtää kaikkea mitä järjestelmässä tapahtuu. Kehitys ja julkaisu putket hoitavat asia hänen puolestaan. Tällöin kehittäjä voi keskittyä muutosten tekemiseen, ja kun tarve tulee hahmottaa putket ovat koodattu selkeästi kertomaan mitä ne tekevät.

Arviolta devopsin pitäisi auttaa kaikissa hypoteeseissa. Sen pitäisi helpottaa luovutuksen jälkeen uuden kehittäjän kehitys prosessia ja auttaa ylläpitämään nopea kehitys tahtia. Samalla putken pitäisi auttaa pitämään tekninen velka hallittavan. Toisaalta kehitys putket voivat olla myös haitaksi luovutuksen jälkeen. Ne ovat normi projektin lisäksi asia mitä pitää ylläpitää. Niitä pitää osata huoltaa. Tällöin kehittäjä menettää projektin kehitys aikaa. Uuden kehittäjän pitää osata käyttää välineitä. Liian vähäinen ymmärtäminen tai helposti hajoava putki on vain haitaksi. Tällöin vähintään K2 ja K3 voivat toteutua. Teknistä velkaa tulee itse putken toteutuksesta ja tämä hidastaa kehityksen kulkua.



Kuva 2.3: Jatkuva kehitys ja jatkuva toimitus

2.3 Kehityksen luovuttaminen

Suomeksi käännettynä handover on luovuttamien ja Cambridgen määritys sanalle on antaa kontrolli tai vastuu luovutus jollekin toiselle taholle. [25]. Ohjelmisto kehityksessä luovutukset ovat hyvin erilaisia ja riippuvat minkälainen luovutus ollaan tekemässä. Kaikissa on silti hyvin paljon samanlaisia piirteitä, jotka määrittävät nämä saman luokittelun alle, kuitenkin jokaista luovutus kategoriaa pitää lähestyä erinaisella näkökannalla. Tämän tutkimuksen puitteissa luovutuksella tarkoitetaan ohjelmisto kehitykseen vastuun luovuttamista toiselle taholle.

Siirtymä luovutus ohjelma siirtyy uuteen vaiheeseen kehityksessä. Usein tällainen tapahtuma on, kun ohjelma siirtyy kehitys vaiheesta ylläpitoon ja toisen kerran, kun siirrytään ylläpidosta lopetukseen. Siirtymässä kehitys tiimi voi pysyä samana. Useimmiten tiimi vaihtuu kehitys tiimistä ylläpito tiimiin, joka saa kehitys vastuun projektista. Kolmantena vaihtoehtona on, että tiimi vaihtuu mutta osa kehitys tiimin jäsenistä jää luovutuksen jälkeen kehittämään edelleen projektia.

Kolmannelle osapuolelle luovutus Ohjelma luovutetaan toiselle taholle täysin. Mahdollinen omistus säilyy vanhalla omistajalla, mutta ohjelman kehitys luovutetaan toiselle taholle. Huomattavaa että tämä ei sulje pois siirtymä luovutusta.

Tuote voi olla mahdollisesti tekemässä siirtymää kehityksestä ylläpitoon, ja sen ottaa haltuun kolmas osa puoli. Usein tällaisia tehdään isoissa firmoissa. Firma itse ei ole teknologia osaava. Tällöin yhtiö palkkaa kolmannen osapuolen hoitamaan ohjelmisto kehityksen heidän puolestaan.

Kehittäjän luovutus Yksi tai useampi kehittäjä lopettaa tuotteen kanssa työskentelyn ja luovuttavat kehitys vastuun toiselle kehittäjälle. Tällaisia on kahden laisia. Vanha kehittäjä auttaa uutta kehittäjää työskentelemään projektissa ja lopettaa kun uusi pystyy lähes samanlaiseen kehitykseen kuin vanha yksin. Toisessa vaihtoehdossa vanha kehittäjä jättää kehitys työn enne kuin uusi kehittäjä saapuu paikalle. Tällöin uusi kehittäjä pitää pärjätä mahdollisella dokumentilla. Näissä tapauksissa on mahdollista, että menetetään paljon hiljaista tieto, koska uudella ja vanhalla kehittäjällä ei ole ollut mahdollisuutta kommunikoida.

Huomattavaa että mikään näistä ei ole toisiaan pois sulkevia, Kaikki kolme voi tapahtua samaan aikaan. Tuote luovutetaan kehityksestä ylläpitoon. Ylläpidon hoitaa kehitys tiimistä ulkopuolinen taho, jonka kehittäjillä ei ole ollut aikaisempaa kokemusta projektista.

Ohjelmalistaus 1 Huonosti ohjelmoitu java luokka

```
{java}
public class B extends P implements Serializable{


final private String n="Bishop";


public B(C c, Coo coo) {
super(c, coo);
}


@Override
public boolean isM(Coo toCo){
if(Math.abs(toCo.X() - X) == Math.abs(toCo.Y() - Y)){
return true;
}
return false;
}


@Override
public boolean isA(Coo toC) {
return isM(toC);
}


public String gM(){
return n;
}
}
```

Ohjelmalistaus 2 Paremmin ohjelmoitu java luokka

```
{java}
public class Bishop extends Piece implements Serializable{

final private String name="Bishop";

public Bishop(Colour colour, CartesianCoordinate coordinate) {
super(colour, coordinate);
}

@Override
public boolean isMovePossible(CartesianCoordinate toCoordinate){
int toX = toCoordinate.getX();
int toY = toCoordinate.getY();
if(Math.abs(toX - fromX) == Math.abs(toY - fromY)){
return true;
}
return false;
}

@Override
public boolean isAttackPossible(CartesianCoordinate toCoordinate) {
return isMovePossible(toCoordinate);
}

public String getName(){
return name;
}
}
}
```

3 Tutkimukset kehityksen luovutuksesta

Tutkimuksia luovutuksesta on tehty hyvin vähän. Käsittelen tässä kattavimmat tutkimukset, jotka tutkivat luovutusta.

3.1 Ydinongelmat luovutuksessa

Khan ja Kajiko-Mattsso tutkivat viiden yhtiön luovutus prosessia tutkimuksessa "Core handover problems"[7]. Tutkittavien yhtiöiden henkilömäärät vaihtelivat 8-80 000 työntekijän välillä. Tutkimus tehtiinn haastattelemalla yhtiön valitsemien edustus henkilöiden kanssa. Haastatteluissa huomattiin, että tutkittavissa ihemistokehitys yhtiössä huomattiin yhteneviä ongelmia, kun ohjelmisto kehityksen luovutusta tehtiin. Vaikka kehitys prosessit ja projektien kokoluokat erosivat toisistaan. Haastatteluissa löydettiin viisi ongelmaa, jotka ilmenivät useammassa ohjelmistokehityksen luovutuksessa.[7]

Liian vähäinen järjestelmän ymmärrys tulevilta kehittäjiltä haittaa ohjelmistokehitystä. Huonolla järjestelmän ymmärryksellä kehitys hidastuu. Kaikissa tutkimuksen tutkimissa luovutuksissa tämä ilmeni suurimpana ongelmana. Uuden ja vanhan tarvitsemat tiedot, jotta he pystyvät tehokkaaseen kehitykseen voivat erota paljon. Esimerkiksi järjestelmän ylläpitäjä ja kehittäjä tarvitseevat täysin erilaisen ymmärryksen järjestelmästä voidakseen kehittää järjestelmää tehokkaasti. Puuttu-

van ymmärryksen takia uusien kehittäjien/ylläpitäjien kehitys voi hidastua tai aiheuttaa vahinkoa järjestelmälle. K2 hypoteesiin todettiin ilmenevän tutkimuksen löytämissä tuloksissa. Tämä juuri johtuu vähäisestä ymmärryksestä mikä pakottaa kehittäjän enemmän käyttää aikaa järjestelmän oppimiseen. Samalla kehittäjän ymmärryksen takia hänen luottamuksensa on heikkoa. Tällöin tärkeiden päätösten tekeminen on hidasta. Lopulta vähäinen ymmärrys aiheuttaa mahdollisesti ei niin hyviä ratkaisuja, koska ei ole tarpeeksi hyvää ymmärrystä. Sekin tutkimuksessa mainittiin, että yleistä oli, että ohjelmiin tuli toimintahäiriöitä luovutuksen jälkeen. Mahdollisesti huonon ymmärryksen takia, mutta tätä ei voitu todistaa tutkimuksen puitteissa. Tätä pystyttäisiin mahdollisesti havainnoimaan K3 hypoteesia tutkimalla. Tutkimuksen antama korjaus ehdotus ongelmaan olisi tulevien kehittäjien integroiminen, ja tutustuminen vanhoihin kehittäjiin aikaisemmin. Ideana olisi välittää uusien ja vanhojen välillä tietoa ennen luovutusta. Uudet pystyvät oppimaan vanhoilta hiljaista tietoa mitä ei pysty dokumentoimaan. Samalla uudet voivat läpikäydä dokumentaatiota, ja varmistaa, että dokumentaatio on ajan tasalla. Ennen luovutusta voivat uudet vielä pyytää vanhoja korjaamaan dokumentaatiota, jos virheitä tai puutteita löytyy.[7]

Riittämätön dokumentaatio projektista. Robert C. Martin sanoi kirjassaan, että dokumentaatio on pahasta[9]. Johtuen tämän vaarasta vanhentua, ja antaa kehittäjälle vääränlaista tietoa järjestelmästä. Tutkimuksessa kuitenkin huomataan, että tietyn asteinen dokumentaation on hyvästä ja auttaa kehitystä. Varsinkin luovutuksessa on elintärkeää ja nopeuttaa luovutuksen tekoa.

Hyvällä dokumentaatiolla pystytään luovuttamaan uusien ja vanhojen välillä tietoa. Uusi pystyy käymään läpi tietoja ilman vanhan läsnäoloa. Dokumentin avulla uusi kehittäjä pystyy luomaan pohjan ymmärrykselleen. Sen avulla hän voi tehdä tärkeitä kysymyksiä vanhalle kehittäjälle, joilla syventää ymmärrystä. Samalla uusi kehittä pystyy tarkistamaan tietojen paikkansa pitävyyden tai relevantisuuden. Lä-

pikäynnin ansiota kehittäjä pystyy korjaamaan dokumentaatiota ennen luovutusta. Huono dokumentaatio voi aiheuttaa uudelle kehittäjälle paljonkin vaikeuksia. Uusi kehittäjä ei voi luottaa vanhaan dokumenttiin. Huonon dokumentaatio pakottaa uuden kyselemään kaikkista sioista vanhaa kehittäjää. Kummaltakin kehittäjältä menee paljon aikaa vain tiedon siirtämiseen, kun vanha luo uuden kehittäjän ymmärrys pohjan, koska uusi kehittäjä ei voi hyödyntää dokumentaatiota. Dokumentaatio on myös ainut asia mikä voi jäädä vanhan lähtemisen jälkeen. Erittäin huonossa tilanteessa vanhaan ei ole enää mahdollista saada yhteyttäkään. Tällöin dokumentaation tärkeys kasvaa merkittävästi. Huono dokumentaatio voi hidastaa tai aiheuttaa vahinkoa, jos dokumentoitu informaatio on vääristynyttä. Vaikutukset tulevat näkymään myös hyvin tutkimuksen kysymyksissä. K3 hypoteesi tulee todnnäköisesti toistumaan, jos dokumentaatio on huono. Syinä tähän voi olla harhaan johtava dokumentaatio. Vanhentunut dokumentaatio, jolla ei ole merkitystä, tai puuttuva dokumentaatio. Usein kaikissa näissä tapauksissa kehittäjä joutuu käyttämään paljon aikaa vain järjestelmän toiminnallisuuden oppimiseen lukemalla lähdekoodia. Tällöin kehittäjälle jää hyvin vähän aikaa normaaliin kehitykseen. K2 on todennäköistä, että tulee toteutumaan. Kehittäjä ei pysty käyttämään aikaansa kehittämiseen. Hänen pitää käyttää suurin osa ajasta järjestelmän ymmärtämiseen. Ilman pohjaa, joka pystyttäisiin luomaan dokumentaatiolla, tämä on erittäin hidasta. Huonoilla dokumenteilla myös on selkeä vaikutus K1 hypoteesissa. Kommittien granulamaisuus kasvaa, koska vähäisen informaation takia kehittäjä on paljon epävarmempi tekemistään muutoksista, jolloin hän tekee mieluummin pieniä muutoksia kerralla järjestelmään. Ilman tiedon tuomaa varmuutta kehittäjä tekee granulaisia muutoksia. Saadakseen paremman ymmärryksen järjestelmästä ja minimoimalla riskejä aiheuttaa suurempaa ongelmia järjestelmässa. Usein ongelmaksi huonon dokumentaation muodostumiseen on liiallinen kiire kehityksen aikana. Usein teknistä velkaa hankitaan jättämällä dokumentaation teko vasta kehityksen loppupuolelle. Pahim-

massa tapauksessa dokumentaatiota ei ole yhtään. Usein kehittäjä on unohtanut tärkeitä asioita mitä pitäisi dokumentoida, jos hän tekee sen vasta ihan lopussa. Mahdollisia ratkaisu ehdotuksia, joita annettiin tutkimuksessa on nimittää henkilö, joka keskittyy täysin dokumentaation tekemiseen. Dokumentaation tekemisen lisäksi pitää myös ylläpitää dokumentaatiota, jotta se ei vanhene. Vanhentunut dokumentaatio voi olla haitallista ohjelmistokehitykselle. Vanhentunut voi jopa johtaa kehittäjän väärään ymmärrykseen järjestelmästä. Tällöin kehittä voi rikkoa järjestelmän toiminnallisuutta muutoksillaan, jotka teki vanhentuneeseen dokumentiin pohjautuen.[7]

Riittämätön ymmärrys domainista jota kohde projektissa tarvitaan. Luovutus tulee selkeästi vaikeutumaan, jos esimerkiksi luovutettavilla kehittäjillä ei ole tarpeeksi ymmärrystä domainista. Khanin tutkimuksessa käytettiin esimerkkinä perus tietoturva protokollia. Kohteelle, jolle luovutettiin projekti ei ymmärtänyt tarpeeksi perus asioita tietoturvallisuudesta tai kryptologiasta. Tällöin luovutuksen kohde kehittäjät, joutuvat käyttämään suuren osan ajastaan uuden asian oppimiseen. Paljon huonommassa skenaariossa kehittäjät jatkavat kehitystä ilman tarvittavaa ymmärrystä, ja aiheuttavat mahdollisesti ongelmia itselleen ja käyttäjille tulevaisuudessa. Kehityksen jatkaminen samaan tahtiin ilman tarvittavaa pohja tietoa voi aiheuttaa pysyviä haittoja järjestelmässä. Tutkimuksessa ongelmaan ehdotettaan ratkaisuksi tarkastaa tuotteen vastaan ottavien kehittäjien ymmärrys. Onko uudella kehittäjillä tarvittava ymmärrys domaineista ja teknologeista, ennen kuin tuote luovutetaan eteenpäin. Huono ymmärrys domainista laskee kehittäjän itsevarmuutta tehdä muutoksia. Vaikutuksena tällä on K1 mahdollinen tilanne. Toisaalta jos kehittäjä ei tajua vähäistä ymmärrystään ja jatkaa kehitystä samaan tahtiin pitäisi vaikutukset näkyä myös K3 hypoteesissa. Joutuessaan opettelemaan domain tietoa kehittäjä tekee pieniä muutoksia nähdäkseen miten ne vaikuttavat, ja oppi domainia pieniä muutoksia tekemällä. Huono domaini ymmärrys pakottaa kehittäjän opiskelemaan,

joka hidastaa ohjelmokehityksen nopeutta. Tällöin K2 hypoteesi tulee toteutumaan. Huonolla domain ymmärryksellä on vaikutuksia K3. Huonolla ymmärryksellä kehittäjä voi tehdä muutoksia järjestelmään tietämättään, että ne voivat olla haitallisia järjestelmälle.[7]

Riittämätön kommunikaatio tuotteen luovuttavan ja tuotteen vastaanottavien kehittäjien välillä hidastaa luovutus prosessia. Tehokas luovutus saadaan aikaiseksi, kun kehittäjät kummallakin osapuolella pystyvät kommunikoimaan toistensa kanssa riittävästi. Tällöin pystytään varmistamaan, että mahdollisimman vähän tärkeää tietoa häviää. Tarpeellisella kommunikoinnilla pystytään myös siirtämään hiljaista tietoa, jota on vaikea dokumentoida. Tutkimuksessa huomattiin, että ketterällä kehityksellä pystyttiin vähentämään kommunikoinnista syntyviä luovutus ongelmia. Ketterän kehitys auttoi silloin, kun uusia kehittäjiä integroitiin ketterään kehitykseen ennen lopullista luovutusta. Tällöin ketterien mallien säännöllisissä palavereissa uudet ja vanhat kehittäjät pääsivät keskustelemaan järjestelmästä luonnollisesti a säännöllisesti. Tämä taas auttoi lopullisessa luovutus prosessia. Syitä huonoon kommunikaatioon on monia syitä. Liian lyhyt luovutus, tai vanhalla ja uudella kehittäjällä on hyvin vähän aikaa kommunikoida toistensa kanssa ennen luovutusta ja luovutuksen jälkeen. Usein näitä pystytään ratkaisemaan varaamalla selkeästi aikaa kommunikoinnille uuden ja vanhan kehittäjän välillä ennen luovutusta. Huonolla kommunikaatiolla on vaikutusta K1. Vähäinen kommunikaation ansiota uudella kehittäjällä voi olla heikko ymmärrys järjestelmästä. Samalla häneltä voi puuttua tärkeää tietoa järjestelmästä. Heikko ymmärryksen takia kehittäjä tekee pieniä muutoksia koodiin kerralla, jotta hän voisi minimoida riskit. Kehittäjä haluaa minimoida riskit. Tämä johtuu myös vähäisen kommunikaation takia. Kehittäjällä ei ole syntynyt itsevarmuutta järjestelmästä, koska ei ole voinut luoda hyvää pohjaa järjestelmän ymmärryksestä. Uusi kehittäjä ei ole myös voinut varmistaa keneltäkään ymmärrystä järjestelmästä. Tällöin hän yrittää varmistaa, että

kehitys vaikuttaa mahdollisimman vähän järjestelmän perustoimintaan. K2:ssa voi myös näkyä huonon kommunikaation vaikutus. Vähäisen kommunikaation takia uusi kehittäjä voi ottaa luovutuksen jälkeen vanhaan yhteyttä, koska hänellä saattaa olla edelleen tärkeää tietoa järjestelmästä. Puuttuva tieto voi hidastaa kehitystä. Sama vaikutus voimistuu, jos ei ole yhteyttä vanhaan kehittäjään luovutuksen jälkeen. Kehittäjän täytyy käyttää paljon aikaa järjestelmän ymmärtämiseen käymällä lähdekoodia läpi, joka tule hidastamaan kehitystä. Vaikutus huonolla kommunikaatiolla on myös K3. Hyvin olennaista hiljaista tietoa voi kadota, jos kehittäjät eivät kommunikoi tarpeeksi. Tällöin uusi kehittäjä voi puutteellisten tietojen takia tehdä ei niin optimaalisia ratkaisuja muokatessaan järjestelmää.[7]

Vaikeuksia seurata muutoksia johtuen suuresta määrästä vaadittavia muutoksia heti luovutuksen aikana. Syynä tällaiseen on monia. Mutta hyvänä esimerkkinä on kehityksen aikana kerätty tekninen velka, joka realisoituu luovutuksen jälkeen. Tällainen aiheuttaa hankaluutta, jos järjestelmästä tehdään seuraava iteraation ja tehdyt muutokset ylläpidossa pitää integroida seuraavaan versioon. Nykyään tällainen ongelma on hyvin minimaalien. Repositorit, kuten Git pystyvät mainiosti seuraamaan tehtyjä muutoksia. Gitin tapaiset ohjelmat pystyvät seuraamaan tehtyjä muutoksia ja valvomaan ongelmallisia integrointeja mitä voisi tapahtua. Vaikka nykyiset ohjelmat auttavat tässä ongelmassa merkittävästi eivät ne sulje täysin pois ongelmaa. Mitä suurempi yhtiö ja isompi yhteisö, joka työskentelee projektissa. Sitä suurempi todennäköisyys, että ongelmia voi ilmetä suurissa ohjelmistokehitys järjestelmissä. Suositeltavaa olisi käyttää tällöin kehitys tapoja, joissa voidaan tarkemmin seurata muutosten tilaa esimerkiksi. Kanban. Sovelluksia, jotka pystyvät auttamaan tässä ovat esimerkiksi Trello. Huonolla seurannalla on vaikutuksia K2. Mahdolliset sekaannukset ja ongelmat voivat hidastaa kehitystä. Muutos ongelmat vievät aikaa pakottaessa kehittäjät hoitamaan ongelmia, jotka ilmenevät konfliktisesta muutosten korjaamisesta. Ongelma tilanteissa pitää pysäyttää kehitys ja selvitellä missä

muutokset menevät. Vaikutuksia on myös K3. Huonolla muutosten seurannalla voi syntyä teknistä velkaa. Toinen mahdollisuus on, että sekalaisessa muutos historiassa teknisen velan seuranta vaikeutuu. Unohdettu tekninen velka voi pahentua ajan myötä, ja aheuttaa enemmän harmia mita kauemmin se on hoitamatta.[7]

3.2 Washizakin tutkimukset

Muita merkittäviä tutkimuksia on Washizakin ja kumppaneiden tutkimus luovutusten kaavoista. He tutkivat mitkä ovat mahdollisia antikaavoja. Kaavoja, jotka ovat haitallisia luovutukselle. Samalla tukija on tehnyt tutkimusta myös mahdollisista kaavoista, joilla pystyttään ratkaisemaan tai helpottamaan antikaavoista syntyviä ongelmia. Näissäkin tutkimuksissa mainittiin luovutusten vähäinen tutkiminen. Antikaavat, joita tutkittiin, olivat tutkimuksesta nimeltä "Anti-Patterns Project Management". Niiden pohjalta työpajoja yliopistoissa. Työpajojen tarkoitus oli kehittää kaavoja, joilla estää antikaavojen muodostuminen. Osa myös löydettiin haastattelemalla yhtiöitä kehityksen luovutuksesta. Huomioitavaa on, että työpajojen koot olivat hyvin pieniä. Myös haastateltavista yhtiöistä ei mainita koko luokkaa tai kuinka monta yhtiötä haastateltiin. [26][27][28]

3.2.1 Antikaavat

Päivittämätön dokumentaatio Järjestelmässä tehdyssä dokumentaatiossa huomataan virheitä tai vanhentunutta informaatiota. Dokumentaatiota ei kuitenkaan päivitetä. Mahdollisesti edelläkävijä ei voi edes varmistaa onko dokumenttia päivitetty. Uudelle kehittäjälle luovutetaan virheellistä informaatiota, joka vaikeuttaa hänen toimintaansa uutena kehittäjänä. Huono dokumentaatio tulee hidastamaan uuden kehittäjän kehitys nopeutta, koska hän joutuu käyttämään enemmän aikaa järjestelmän ymmärtämiseen. Tällöin K2 hypoteesi realisoituu. Huonon dokument-

ti mahdollisesti luo kehittäjälle epävarmuutta järjestelmän ymmärryksestä. Tällöin hän mahdollisesti tekee pienempiä muutoksia. Varoakseen että ei riko järjestelmän toiminnallisuutta. Samoin K1 hypoteesi realisoituu. K3 hypoteesi voi realisoitu myös huonosta dokumentaatiosta. Vähäinen ymmärrys voi aiheuttaa kehittäjän tekemään huonoja kehitys ratkaisuja, tai kehittäjältä puuttuu tieto olemassa olevasta teknisestä velasta. Unohdettu tekninen velka mätänee ja tulee aiheuttamaan enemmän ongelmia tulevaisuudessa. Tutkimus antaa ratkaisuksi, että dokumentaatiolla pitäisis olla tarkkailtava historia, josta voi nähdä milloin dokumentaatiota on viimeksi päivitetty. Tällaisia ratkaisuja olisi mahdollisesti laittaa dokumentaatio version hallinta järjestelmään, kuten Git. Sen avulla pystyy tarkkaan näkemään dokumenttiin tehdyt päivitykset. Toinen ohje on, että kehittäjät tarkistavat ja päivittävät dokumentaatio usein ja säännöllisesti.

Taustatieto ei ole selkeä Kaikilla ohjelmilla on taustatietoa. Taustatietoja voi esimerkiksi olla budjetti rajoitukset tai mahdolliset asiakkaan vaatimat rajoitukset. Tutkimuksessa uskoakseni tarkoitetaan enemmän hiljaista tietoa, jota on vaikea dokumentoida. Tämä tieto helposti kadotetaan luovutuksessa. Esimerkkinä voin käyttää omassa työssäni esiintyvää ongelma. Asiakkaalla kestää byrokraattisista syistä budjetointi hyvin kauan. Tämä tarkoitti, että työtä ei saanut tehdä vuoden alussa, koska budjettia ei ollut allokoitu. Budjetti päätettiin ja saimme luvan tehdä vasta maaliskuun loppupuolella. Tätä harvemmin ei dokumentoi, koska tämä on myös vaihteleva aika riippuen yhtiön päätös nopeudesta vuosittain. Tällaisten vaikutusta on vaikea suoraan mitata. Esimerkin ongelmalla on selkeä vaikutus K2 hypoteesin toteutumiseen. Kehitys pitää pysäyttää kolmeksi kuukaudeksi. Mahdollisia muita vaikutuksia epäselvällä taustatiedolla on esimerkiksi tekninen velka, jota ei ole dokumentoitu. Tällöin K3 hypoteesi realisoituisi. Kummassakin esimerkissä on vaikea suoraan sanoa, miten paljon ne vaikuttaa luovutukseen, tai onko edes ollut tautatieto puutetta, koska mahdolliset vaikutukset eivät ilmene aina luovutuksessa, tai

hyvin myöhään luovutuksen jälkeen. Omassa työpaikka esimerkissä asia ilmeni vasta vuoden jälkeen siitä, kun luovutus tehtiin.

Tarpeellinen tieto puuttuu Vanha kehittäjä päättää aina mitä tietoa jakaa ja mitä näkee tarpeelliseksi kirjoittaa dokumentaatioon. Usein ongelmaksi tulee, että päättäjänä mitä dokumentoidaan, on ainoastaan vanhan kehittäjän subjektiivinen näkemys asiasta. Tällöin mahdollisesti tietoa, jota uusi kehittäjä tarvitsee, mutta vanha kehittäjä näkee tarpeettomana voi jäädä uupumaan. Tämä hankaloittaa uuden kehittäjän kehitys työtä. Vaikutukset usein näkyvät hyvin nopeasti riippuen toki tiedon tärkeydestä. Tällaisia voisi esimerkiksi olla tärkeiden salasanojen säilytys paikan kertominen. Riippuen puuttuvan tiedon relevanttiudesta vaikutukset voivat olla saman tien luovutuksen jälkeen. Tällöin vaikutukset näkyvät kyllä kaikissa kolmessa kysymyksessä. Kehittäjä menettää uskon dokumentaation ja joutuu tutkimaan lähdekoodia. Tämä hidastaa kehitystä. Mahdollinen tärkeän tiedon puuttuminen voi aiheuttaa huonoa koodia järjestelmään.[26]

3.2.2 Luovutuskaavat

Tiedon levittäminen kehittäjä jakaa osaamistaan ja ymmärrystä muille henkilöille firmassa. Tiedon levittämisessä vähennetään tärkeän tiedon menetys riskiä. Luovutuksen jälkeen kehittäjät, joille on levitetty tietoa, pystyvät auttamaan tai neuvomaan uutta kehittäjää. Tämä on myös hyödyllistä tehdä, jos vaikka järjestelmän kehittäjä sairastuu. Tällöin toisella kehittäjällä on perusymmärrys järjestelmästä ja pystyy ylläpitämään järjestelmää sen aikaa, kun pääkehittäjä on sairas. Tekniikka käyttäminen näkyisi selkeästi kaikissa tutkimus kysymyksissä. Tiedon levityksessä uusi kehittäjä voi tehdä kysymyksiä muille kehittäjille, kun vanha on lähtenyt. Tällöin uusi kehittäjä saa paremman varmuuden mitä pitää tehdä. Muutoksia tulee nopeammin ja enemmän. Tekninen velkakin pysyy vähäisempänä, kun muut kehittäjät voivat ohjeistaa linjausta mitä projektissa kuuluisi käyttää. Tämä luo myös

hyvän turvaverkon, jos luovutus tarvitsee tehdä nopeasti ja vanhalla kehittäjällä ei ole aikaa tehdä kunnon luovutusta. Tällöin henkilöt, joille tieto on levitetty pystyvät hoitamaan järjestelmää siihen asti, että uusi kehittäjä löydetään. Positiivisia vaikutuksia pitäisi näkyä K1 ja K2 kysymyksessä, koska uudella kehittäjällä on tuki henkilö, johon tukeutua. Pystyy helpommin saamaan apua ja henkistä tuke mikä lisää itsevarmuutta järjestelmän kehittämisessä.

Luovutus toisessa huoneessa Luovutuksessa usein uusi kehittäjä ei ole niin tietoinen tai kokenut järjestelmästä, kuin vanha kehittäjä. Uuden kehittäjän tulee kysymään paljon asioita vanhalta kehittäjältä. Kysymyksien vastaamiseen menee aikaa, jotta uusi kehittäjä pystyy sisäistämään vastauksen. Vaarana on, että kysymysten aikana heitä tullaan häiritsemään. Mahdollisesti luovutus prosessi häiriintyy tällaisista tekijöistä. Arvokasta tietoa voi kadota keskustelussa. Tutkimuksessa ehdotetaan, että tällaisissa tilanteissa uusi ja vanha kehittäjä siirtyvät erilliseen tilaan, jossa voivat kahden kesken ilman häiriöitä käydä kysymykset läpi. Samalla voidaan käydä jatko kysymykset ilman häiriötekijöitä. Uusi kehittäjä pystyy tehokkaammin sisäistämään vanhan antaman tiedon, kun ei ole haittaavia häiriötekijöitä. Tutkimuksessa mainitaan, että tämä kaava tarvitsee tiedon levittämistä aluksi, koska jonkun pitää pystyä hoitamaan järjestelmää sillä aikaa, kun luovutus keskusteluja tehdään vanhan ja uuden välillä. Erillisen huoneen tapahtuessa kehityksen hoitaa henkilöt joille tieto on levitetty. Henkilöt, joille on levitetty, tieto pystyy tekemään peruskehityksen sillä aikaa, kun luovutus toisessa huoneessa on käynnissä. Tällöin saadaan paljon tietoa siirrettyä uudelle kehittäjälle ilman tiedon häviöitä. Tämä kasvattaa uudenkehittäjän ymmärrystä ja itsevarmuutta järjestelmästä. K1, K2 ja K3 hypoteeseille tällä on positiivinen vaikutus. Kehittäjä pystyy paljon nopeammin tekemään kehitystä. Hänellä ei mene aikaa hukkaan järjestelmän ymmärtämiseen. Kehittäjä luo todennäköisesti vähemmän teknistä velkaa, koska hän ymmärtää paremmin, miten järjestelmää on aikaisemmin kehitetty.

Palomuuri luovutukselle Mainitsin asiasta jo aikaisemmassa kaavassa. Usein vanhalla kehittäjällä on kiireinen aikataulu. Hänellä on vähän aikaa siirtää olennainen tieto uudella kehittäjällä. Levittämällä tietoa uusille kehittäjille vanha kehittäjä luo itselleen palomuurin. Vanha kehittäjä pystyy keskittymään rauhassa tiedon luovuttamiseen uudelle kehittäjälle. Samaan aikaan kehittäjät, joille tietoa on levitetty tekevät tärkeät asiat, jotka vaativat vanhan kehittäjän panosta. Tällöin henkilöt, joille levitettiin tietoa pystyvät toimimaan muurina, joka estää häiriöt luovutuksessa uuden ja vanhan välillä. Tällöin mahdollisimman paljon tietoa siirtyy vanhan ja uuden välillä. Vaikutukset ovat kaikissa kolmessa tutkimuksen kysymyksessä. Uudelle kehittäjälle on saatu mahdollisimman paljon siirrettyä vanhan tietoa. Uuden kehittäjän tietotaso on korkeampi ja sitä kautta myös itsevarmuus. Tällöin pystytään minimoimaan tekninen velka ja uuden kehittäjän hidas kehityksen aloitus hänelle uudessa järjestelmässä. [27]

Osaamisen tarkistus Luovutuksessa on usein hyvin rajattu aika enne kuin vanha kehittäjä poistuu. On tärkeää pystyä löytämään henkilö, joka on soveltuva jatkamaan järjestelmän kehitystä vanhan poistuttua. Tutkimuksessa suositellaan, että uusi kehittä tuodaan työskentelemään vanhan kehittäjän kanssa ennen lopullista luovutusta. Kummatkin kehittäjät tekevät kehitystä järjestelmän kanssa. Vanha kehittäjä pystyy helpommin arvioimaan uuden kehittäjän soveltuvuutta. Tällöin pystytään nopeasti arvioimaan henkilön soveltuvuus tehtävään, ja vaihtamaan nopeasti uuteen tarpeen vaatiessa. Positiivisia vaikutuksia tällä on K1, K2 ja K3. Sopiva henkilö ymmärtää paremmin järjestelmän ominaisuuksia heti alusta asti. Tiedon siirtäminen on helppoa, koska uusi kehittä pystyy ymmärtämään tarvittavat perusasia, jotka auttavat uuden järjestelmän hallinnassa. Tällöin kehitys nopeus tulee olemaan korkeampi heti alusta. Henkilöllä on suurempi itsevarmuus tehdä isoja muutoksia, koska on aikaisemminkin hoitanut samanlaisia ongelmia. Samalla tekninen velan muodostumisen todennäköisyys laskee, koska henkilö ymmärtää minkälaista teknis-

tä velkaa voisi tämän tyyppisessä järjestelmässä olla. Samalla hänellä on kokemusta korjata mahdollisesti syntynyttä teknistä velkaa.

Kaksi kehittäjää Tutkimuksessa todetaan, että on mahdotonta saavuttaa tilannetta, jossa luovutuksessa ei menetettäisi yhtään tietoja. Yksi parhaimpia tapoja minimoida tiedon häviäminen on pistää enemmän, kuin yksi ihminen työskentelemään järjestelmän kanssa. Tällöin jos yksi lähtee ja tekee luovutuksen järjestelmällä, on vielä toinen, jolla säilyy suurin osa tiedosta. Tutkimuksessa kuitenkin mainitaan, että tämän kaavan suurin heikkous on korkeammat kustannukset kehityksessä. Samalla on haasteita valita kaksi henkilöä, jotka toimivat hyvin yhdessä kehittäessä. Tutkimus kysymyksissä tulokset kuitenkin olisivat hyvin positiivisia. Uusi kehittäjä pääsi hyvin nopeasti kärryille järjestelmässä. K1 ja K2 kysymyksissä henkilö pystyy hyvin nopeasti saavuttamaan vanhan kehittäjän työ tason kiitos toisen kehittäjän ohjeistuksen. Tämän pitäisi näkyä lyhyempänä aika, että uusi kehittäjä saavuttaa saman kehitys tahdin kuin vanha kommittien granulaisuudessa ja kehityksen nopeudessa. K3 nähden teknisen velan kasvaminen tulee olemaan hyvin minimaalista, koska toinen kehittäjä pystyy jatkuvasti tarkkailemaan, tuleeko sellaista muodostumaan.

Ymmärryksen vertailu Luovutuksessa luovutettava tieto voi olla hyvin vaikea ymmärtää. Järjestelmä on hyvin monimutkainen ja vaikea selittää. Vanha kehittäjä ei ole tarpeeksi hyvä selittämään olennaista informaatiota. Samalla vanha kehittäjä ei ole ehkä paras selittämään asiaa. Mahdollisesti vanhan kehittäjän tyyli kertoa tieto ei sovellu uudelle kehittäjälle. Kaavan ideana uusi kehittä tekee muistiinpanoja vanhan selityksestä ja täydentää niitä omalla ymmärryksellä. Tämän jälkeen uusi kehittäjä voi kysellä asioita muistiin panojen pohjalta. Samalla vanha kehittäjä voi korjata, jos uuden kehittäjän muistiinpanot ovat virheellisiä. Muistiinpanojen vertailu kasvattaa uuden kehittäjän itsevarmuutta järjestelmässä. K1 ja K2 vaikutukset pienevät, kun kehittäjällä on parempi ymmärrys järjestelmästä ja itsevarmuus teh-

dä muutoksia. K3 velan kasvukin pysyy matalana, kun uusikehittäjän vääränlaiset ymmärrykset järjestelmästä saadaan vähennettyä.[28]

4 Tutkimuksen kohteet

4.1 Fonecta

Vuonna 2002 perustettu Digitaalista markkinointia Suomessa tarjoava yritys. Osa European Directories konsernia, joka tarjoaa samanlaisia palveluja ympäri Eurooppaa. Auttaa ihmisiä saamaan yksityisen tai yrityksen sivustot paremmin näkymään netissä ja kerää markkinointi dataa käyttäjälleen miten tehokasta asiakkaan markkinointi ja mainonta digitaalisesti on esim. kuinka monta kertaa asiakkaan YouTube mainos näytettiin katsojille, tai menikö kukaan niistä asiakkaan sivustolle.[29] Fonecta hankki Kotisivukoneen itselleen vuonna 2013. Suurimman osan ajasta ylläpito ja kehitys on ollut talon sisäistä. Välillä ollut ulkopuolisia henkilöitä työskentelemässä projektissa.

4.2 Vincit

Teknologia yritys, jonka tarjoaa konsultointia, markkinointi apua digitaalisessa markkinoinnissa, uusien digitaalisien tuotteiden kehittämistä ja ylläpitoa asiakkaiden järjestelmille. Vuonna 2007 perustettu. Yhtiön perusti kaksi henkilöä Mikko Kuitunen ja Olli-Pekka Virtanen. Firman perustuksen ideana oli, että maanantaina töihin meneminen ei vituta. Firman kulttuurin myös muodostunut hyvin alhainen hierarkia järjestelmä. Tämä mahdollistaa ison vapauden työntekijöiden tekemissä päätöksissä. Työllisti vuonna 2022 yli 800 henkilöä. Toimipisteitä on Suomen ulkopuolellakin

muun muassa USA:ssa ja Euroopan maissa. Vincit sai kotisivukoneen kehitys ja ylläpito vastuun vuonna 2017, josta on vastannut nykypäivään asti.

4.3 Kotisivukoneen

Kotisivukone on SaaS (Software as Service) palvelu. Suomennettuna tarkoittaa ohjelmisto tarjotaan palveluna asiakkaille. Ideana on vuokrata ihmisille internetin kautta palvelua, jota asiakas ei itse omista. Kotisivukoneen tapauksessa tämä on omat nettisivut. Kuukausi maksua vastaan käyttäjä saa oman nettisivuston uniikilla nettisivu osoitteella. Samalla hänellä on laaja valikoima mahdollisuuksia muokata sivusto haluamansa tapaiseksi.

Aikoinaan 2005 luvulla henkilöt nimeltä Mikko Nurminen ja Antti Ala-Ilkka huomasivat netsivujen tarpeen kasvaneen. Parivaljakko päätti luoda palvelun, joka pystyisi luomaan helposti käyttäjille omat nettisivut. Sivustojen ympärille perustettiin firma Ideakone, joka alkoi kauppaamaan kotisivukonetta asiakkaille.

GitHubin commiteista tutkimalla näemmä, että ohjelmisto kehityksestä vastasi Antti suurimmaksi osaksi. Kotisivukone oli uutisia tutkimalla ainakin ennen vuotta 2010 kehuttu ja menestynyt sivusto, jonka kehuttiin olevan aikansa huippu. Kotisivukoneen huipulla ohjelmaa alettiin, jopa kauppaamaan ulkomaille Groom nimellä esimerkiksi Ruotsiin.

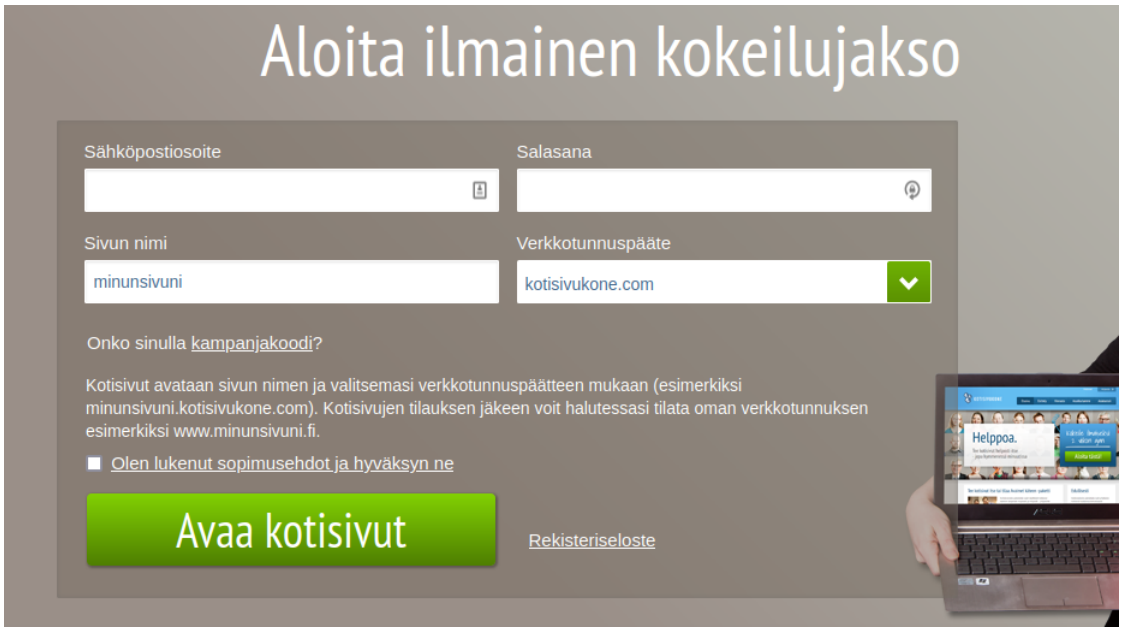
Ideakone ja sen mukana kotisivukone myytiin Fonectalle 2013 luvun vaihteessa. Luoja Antti lähti oston hetkellä pois kehittämästä tuotetta. Useimmat muut kehittäjät siirtyivät työskentelemään Fonectalle ja jatkoivat kotisivukoneen kehittämistä. Ohjelma siirtyi aikoinaan Futurice nimiselle firmalle ylläpidettäväksi. Lopulta ohjelma siirtyi Vincitin ylläpidettäväksi. Alkujaan ohjelmaa koodasi ja ylläpiti vain yksi henkilö Vincitillä. Valitettavasti henkilö päätti lähteä firmasta, jolloin tuli kiire etsiä uusia ohjelmoijia projektiin. Tilalle pistettiin kaksi harjoittelijaa, kun kokeneemmat ohjelmoijat olivat täysin työllistettyjä. Haltuun otto oli muutaman viikon

ja dokumentaatio oli vähäistä.[30]

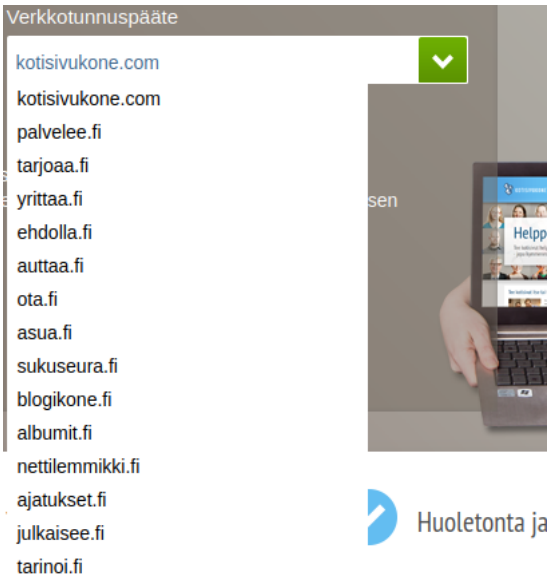
Kotisivukone tarjoaa käyttäjälle 5-20 euron kuukausi maksulla omat sivut riippuen oletko firma vain yksityinen käyttäjä. Sivustoja voi kokeilla kaksi viikkoa ilman maksua.

Sivujen luominen on hyvin suoraviivaista ja helppoa. Asiakas pääsee suoraan etusivulta luomaan oman sivuston itselleen. Käyttääksesi ilmaista koe aikaa tarvitsee vain luoda tunnukset, joka voidaan nähdä kuvasta 4.1, ja luoda oman sivuston osoitte, mikä voidaan nähdä kuvasta 4.2. Tunnuksia käytetään, jotta käyttäjä voi kirjautua ylläpitäjänä sivustoilleen. Sivuston nimi on hyvin vapaa. Ainoana rajoituksena on vain, että se ei saa olla jo entuudestaan oleva nimi. Kotisivukoneen luomissa sivustoissa on aina perässä verkkopäätunnuste <sivun nimi>.kotisivukone.com esimerkiksi. kauppa.kotisivukone.fi. Oletus arvona on kotisivukone mutta vaihtoehtoina on myös 14 muuta verkkotunnus päätettä, mikä voidaan huomata kuvasta 4.3.

Kuva 4.1: Kotisivukoneen etusivu

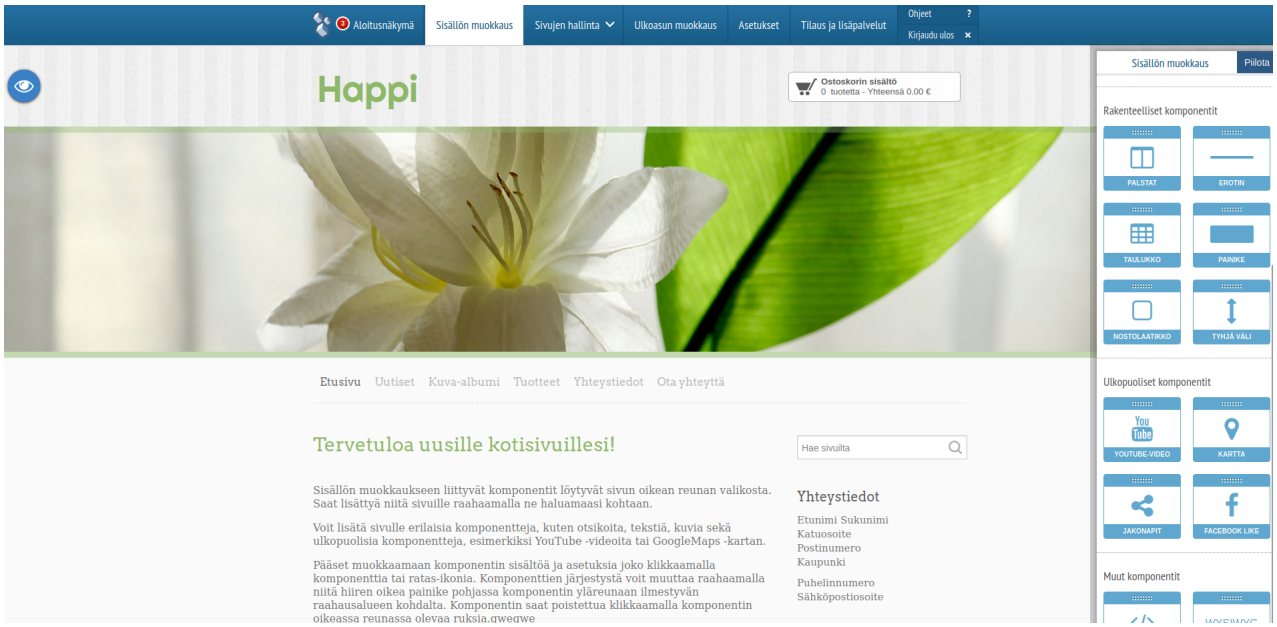


Kuva 4.2: Kotisivukoneen rekistöröinti



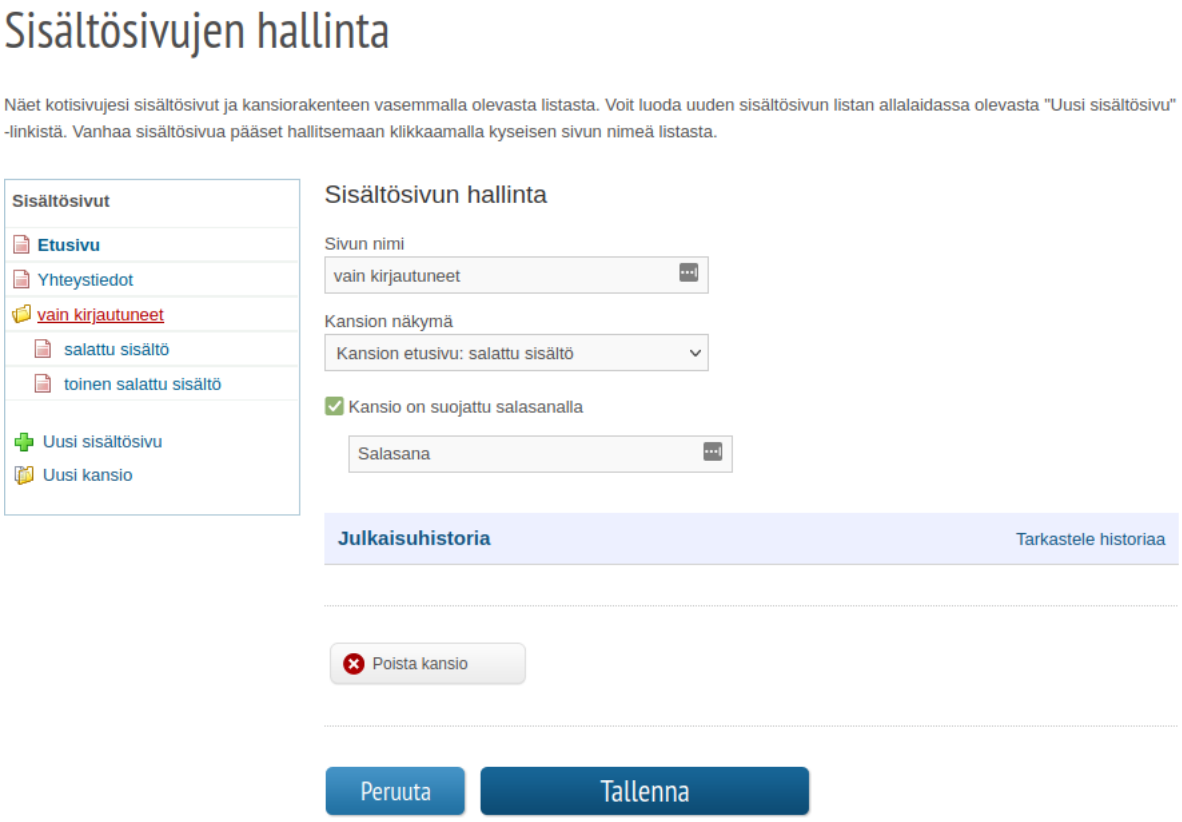
Kuva 4.3: Verkkopäätunnisteet

Sivujen luomisen jälkeen pääsee ylläpitäjän sivulle. Ensimmäiseksi avautuu sisällön muokkaus, jossa voi lisätä erilaista materiaalia sivulle, vaihtoehtoina löytyy esimerkiksi. taulukko, YouTube video tai html koodi. Tämä voidaan huomata kuvasta 4.4



Kuva 4.4: Sisällön Muokkaus

Sivustojen hallinnassa pystytään paremmin kontrolloimaan sivuston kokonaisuutta, kuten kuvan 4.6 osiosta voidaan huomataan. Päävalikon muokkauksessa pystytään vaihtamaan etusivulla olevan valikkopalkin järjestystä ja luoma linkityksiä uusille sivustoille. Sisältösivujen hallinnassa pystyt luomaan uusia sisältö sivuja omalle nettisivullesi. Samalla pystyt myös laittamaan sisältösivuja kansioihin, joiden avulla pysty kategorisoimaan paremmin sisältösivuja ja määrittelemään mitkä ovat nähtävissä kirjautumisella ja mitkä julkisesti kaikkien nähtävissä. Kuvassa 4.5 voidaan havainnoida tätä.



Kuva 4.5: Sisältösivujen hallinta

Kuva 4.6: Sivujen Hallinta

Kolmantena suurena asetuksena on sivujen hallinnassa ulkoasun muokkaus. Siellä voidaan perusteellisemmin muokata sivun ulkoasua. Sisällön muokkauksessa pystyimme vain lisäämään tai poistamaan sisältöä tietyltä sivulta. Ulkoasun muokkauksessa pystytään muokkaamaan muuta esim. kuinka iso teksti aloituksessa on miltä yläpalkki näyttää tai vaihtaa väri valikoimaa. Muokkauksia voidaan havainnoida kuvasta 4.7.

Kuva 4.7: Ulkoasun muokkaus

Järjestelmän kehitys vaihetta ja siirtymistä ylläpitoon on vaikea sanoa. Dokumentaatiota tällaisesta ei ole mutta Git kommiteista voi tehdä karkeita arvioita. Kehittäjiä projektissa on ollut sen elinkaaren aikana yhteensä 28 eri ihmistä. Sivusto on

suurimmaksi osaksi toteutettu Javalla. Sivujen generointi selaimeen on toteutettu xls tekniikalla.

5 Tutkimusmenetelmät

Tutkimus kysymyksiä on pystyttävä arvioimaan jollain tavalla minkälaisia negatiivisia vaikutuksia luovutuksella voi olla. K1 ja K2 kysymysten arvioimiseen on käytetty version hallintaa gittiä tutkimaan vaikutuksia. K3 vaikutuksia on tutkittu Sonarqube järjestelmän avulla.

5.0.1 Git

Git on Linus Torvaldin luoma versionhallinta ohjelma. Ohjelma julkaistiin vuonna 2005. Linus Torvald ei enää vastaa ohjelman ylläpidosta. Tällä hetkellä siitä vastaa Junio Hamano. Git on avoimella lähdekoodilla toimiva ohjelma, jota jaetaan GPL-2.0 lisenssillä.[31]

Ohjelman ideana on tehdä järjestelmän pilvi arkistosta täydellinen lokaali kopio, johon voi tehdä aluksi haluamat muutokset. Tämän jälkeen muutokset synkronoidaan pilvessä olevan arkiston kanssa. Git toimii kommiteilla. Aina kun järjestelmään tehdään muutos ja se voidaan kommittaa gitin säilytyspaikkaan(repository). Kommitti on, kuin järjestelmän kaikista tiedostoista otettaisiin kuva sillä hetkellä. Git vertailee vanhaa ottamaansa kuvaa ja uutta, jos muutoksia ei ole käytetään vanhaa. Huomatessaan muutoksia git tekee muistiin uuden niin kutsutun valokuvan muistiin. Git luo valokuvien välillä linkityksen, jolloin on helppo seurata muutoksien järjestystä ja mitä on milloinkin tehty. Tämän avulla pystytään tarkkailemaan selkeästi mitä ajallisesti on tapahtunut gitin hallitsemissa tiedostoissa. [32] [33]

Git pystyy hyvin seuraamaan muutoksia mutta se ei ole suunniteltu näyttämään tilastollisesti muutoksia. Tähän tarkoitukseen otin käyttöön ohjelman mikä pystyy keräämään dataa paremmin muotoon, jota ihminen pystyy lukemaan. Datan luettavaan muotoon keräämiseen tein eazyBi ohjelmalla.

Tutkimme kahta arvoa ja niiden muutosta. Kommittien määrä tutkitussa ajan jaksossa. Tämä antaa hyvän kuvan miten paljon muutoksia tehdään. Kommittien koko voi olla hyvinkin vaihteleva kuitenkin. Tämän takia toiseksi tutkituksi arvoksi otamme muutosten koon keskiarvon kommiteissa. Tämä lasketaan lisäämällä yhteen koodiin tehdyt lisäykset ja poistot, ja niiden tulos jaetaan kommittien määrällä ajan jaksossa.

5.0.2 Sonarqube

Vuonna 2006 luotu ohjelma, jonka ideana on jatkuvasti analysoida lähdekoodin laatua. Sonarqube on avoimen lähdekoodin alla oleva järjestelmä, joka pystyy tekemään staattista koodin analyysia. Se pystyy löytämään bugeja, huonosti optimoitua koodia ja tietoturvallisuus ongelmia. Sonarquben toiminto perustuu kolmeen periaatteeseen. Skanneriin, joka ottaa vastaan lähdekoodin ja analysoi sen. Toisena tietokanta, johon sonarqube kerää ja tallentaa analysoimansa tiedot. Kolmanneksi helposti ymmärrettävä käyttöliittymä, joka näyttää analysoidut tiedot ihmiselle luettavassa formaatissa. Sonarqube toimii avoimen lähdekoodin lisenssin LGPL 3 alla. Sonarqube määrittele tekniseksi velaksi kohdat, jotka rikkovat yli 600 java koodi sääntöä. Sääntöjen laatua yllapitää Sonarqubin kehittäjät, jotka on todennut sääntojen rikkomisen haitallisiksi. Naiden lisäksi kayttäjät voivat lisätä omia sääntöjä, joita Sonarqube etsii ja merkkaa kehittäjän määrittelemäksi ongelmaksi.[34][35][36]

Sonarqube tuottaa viiden laista dataa. Bugeja ihan vain yksinkertaisia virheitä, joita löytyy järjestelmästä esimerkkinä, vaikka ongelmallinen for lause, johon ohjelma voi jäädä jumiin. Haavoittuvuus ongelmat ovat Sonarqubin mielestä asioi-

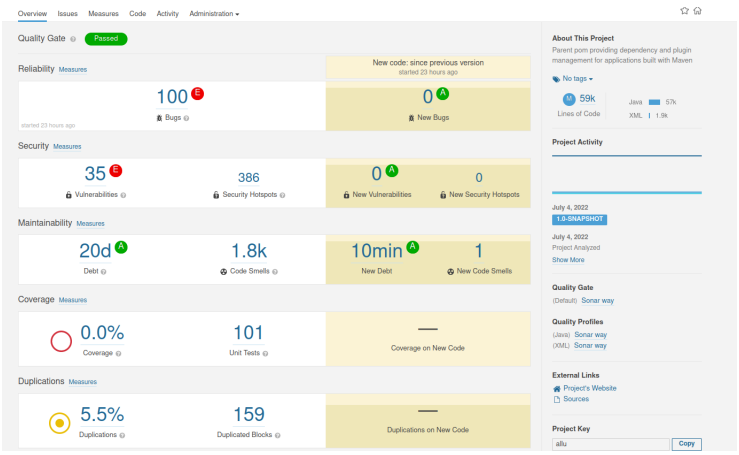
ta, jotka ovat tietoturvallisesti riski tekijöitä järjestelmässä. Sonarqube ottaa myös huomioon ohjelman sisäisiä riski tekijöitä, tai huonosti toteutettua koodia. Hyvänä esimerkkinä koodin osa, joka on aivan liian monimutkainen ja siitä on mahdollista tehdä helpommin ymmärrettävää. Neljäntenä on testi kattavuus. Sonarqube arvioi prosentuaalisesti, kuinka paljon järjestelmän koodista testit kattavat. Viimeisenä on data, joka kertoo lähdekoodissa missä on toistuvia ratkaisuja. Toistuvat ratkaisut antavat ymmarrystä missä voisi parantaa koodia luomalla yhteinen ratkaisu toistuvien alueiden välillä.

Bugit, haavoittuvuudet ja mätänevä koodi luokitellaan myös niiden vakavuuden mukaan. Blokkaava(Blocker) kategorian ongelma on hyvin vaarallinen ja pitäisi saman tien korjata ennen kuin tekee muita muutoksia järjestelmään. Tällaisia olisi esimerkiksi pysyvästi näkyvälle tallennetut salasanat. Kriittinen (Critical) ongelmat ovat järjestelmän toimintoa merkittävästi haittaavia, jotka voivat pahimmassa tapauksessa kaataa järjestelmän toiminnan. Esimerkiksi tyhjän muuttujan käsitteleminen ohjelmassa. Merkittävä (Major) kategoriassa ongelmat alkavat olla enemmän koodaus tyyliin kuuluvia. Näillä ei mahdollisesti ole vaikutusta ohjelman toimintaan mutta voivat haitata kehitystä. Esimerkkinä kommentoitu koodi tai liian monimutkainen metodi. Vähäinen kategoria huomauttaa koodin semanttisuudesta. Esimerkiksi saako funktio palauttaa tyhjän arvon tai käyttämätöntä koodia ohjelmassa. Ei mahdollisesti edes tarpeellista korjata, jos kehityksessä on syy tälle semantiikalle. Informatiivinen(information) tuo kehittäjän tietoon asioita, joita on hyvä tietää järjestelmän koodista. Esimerkiksi ilmoituksia, jotka kertovat, että tämä on vanhentunut. Usein näitä ei tarvitse hyvin nopeasti ottaa huomioon mutta hyvä tietää tulevaisuutta ajatellen.

Sonarqube antaa myös ylläpito arvion lähdekoodista. Arviot tehdään yksinkertaisella lasku kaavalla: ylläpito arvo = tekninen velka/kehityksen hinta. Riippuen laskun tuloksesta tälle annetaan arvosana:

- A: 0-0,1
- B: 0,11-0,2
- C:0,21-0,5
- D: 0,51-1
- E: >1

Huomiona pitää kuitenkin ottaa, että koskaan ei voi tarkalleen sanoa miten kauan kehityksessä menee aikaa. Samalla pitää pohtia lasketaanko näihin kaikki sivullinen toiminta. Ongelman tutkiminen, tiedon kerääminen ja testaaminen. Sonarqube arvioi, että jokaiseen riviin koodia mikä kirjoitetaan, menee 30 minuuttia. Riippuen koodi rivistä. Lasku esimerkkinä, jos on 2500 riviä koodia. Teknistä velkaa on 50 päivää ja työpäivä on 8 tuntia. Henkilö ehtii tekemään 16 riviä koodia tai 0.0625 päivästä tarvitaan jokaiseen riviin. Tulos 50/(0.0625*2500) = 0.32. Taulukon mukaan arvosana on C. Kuitenkin huomataan, että arvosanojen perusteella A arvosanan saa, jos teknistä velkaa on vain kymmenen prosenttia lähdekoodista. Tämä tulee helposti toteutumaan mitä isompi on lähdekoodi. Esimerkiksi miljoonan koodi rivin lähdekoodissa saa olla 100 000 riviä teknistä velkaa mikä on huomattavan paljon mutta saa silti A arvosanan Sonarqubilta. Toisaalta Sonarqubin arvioit ovat hyvää suntaa antavia. Tutkimuksissa on todettu kuitenkin, että Sonarqubin 30 minuutin aika arvio koodi rivin kirjoittamiseen on erittäin paljon yläkanttiin. Kuva referenssi: 5.1 [34] [35] [37]



Kuva 5.1: Sonarqube analyysi tulokset

6 Tiedon analysointi

6.1 Sonarqube

Ensimmäinen otos kohta on juuri ennen luovutusta. Sonarqubilla tutkittiin kolmea ensimmäistä vuotta luovutuksen jälkeen. Tulokset näkyvät taulukossa 6.1. Tulosten ensimmäinen 7.4.2017 otos on viimeinen muutos minkä vanha kehittäjä teki kotisivukoneeseen. Merkittävä määrä numeraalisesti teknistä velkaa vanha kehittäjä on jättänyt kotisivukoneeseen. Sonarqube arvio teknisen velan korjaamiseen menevän 250 vrk. Teknisen velan määrä on numeraalisesti suuri mutta suhteellisesti verrattuna koko projektin kokoon sen määrä pysyy alhaalla. Laskennallisesti määrä suhteutettuna koko projektin kokoon velka pysyy alle 10 prosentin, jolloin ohjelmiston teknisen velan määrän saa parhaimman arvosanan A Sonarqubilta. Saamme tästä tuloksesta pohjan mistä verrata muutoksia.

Rivi määrällisesti katsottuna muutoksia vaihdon jälkeen ei tehty paljoa. Huomataan kuitenkin selkeä nousua teknisen velan määrässä. 2017 vuonna suuria muutoksia ei tapahtunut lähdekoodissa. Vasta 2018 puolella voidaan huomata, että mätänevä koodin määrä on noussut noin 0,18 prosenttia. Trendi ylöspäin jatkuu seuraavassa otoksessa, jossa määrän kasvu on samaa luokkaa noin 0,17 prosenttia. 2019 luvulla muutoksia koodiin tehdään hyvin vähän, jolloin vaikutukset ohjelmistoon ovat minimaalisia. Edelleen pientä ylöspäin suuntautuvaa trendiä on mutta minimaalista. 2020 vuoden alkupuolella puolella velan määrä muuttuu selkeän laskuun

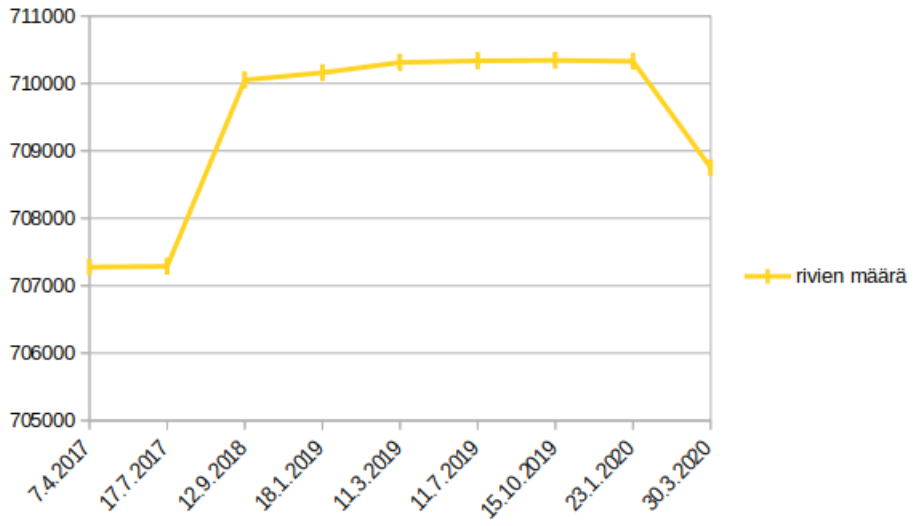
tippumalla hyvin lähelle alkuperäisiä arvoja ennen luovutusta. Samanlainen trendi nähdään muissakin arvoissa. Loppu vuodesta kuitenkin muutokset aiheuttavat merkittävästi taas lisää teknistä velkaa ja noustaan selkeästi aikaisempien tulosten yli. Prosentuaalisesti arviolta ennen luovutusta uusi kehittäjä on kasvattanut tässä kohtaa teknisen velan määrää 1,35 prosenttia. Tulokset näkyvät paremmin Kuvissa 6.1 ja 6.2. Näiden tulosten avulla pystymme toteamaan, että tekninen velka on noussut selkeästi uuden kehittäjän aloittaessa. Tämä vahvistaa K3 hypoteesin. Tuloksista huomaa myös, että kehitys oli myös erittäin hidasta, kun kotisivukone kehitys vastuu luovutettiin Vincitille. Tämä vahvistaa K2 hypoteesia.

Päivämäärä	Bugit	Haavoittuvuudet	Mätänevä koodi	testit	rivien määrä
7.4.2017	1447	387	13389	227	707276
17.7.2017	1447	387	13392	227	707287
12.9.2018	1446	388	13416	227	710052
18.1.2019	1446	388	13439	236	710160
11.3.2019	1446	388	13443	236	710312
11.7.2019	1449	388	13444	236	710337
15.10.2019	1449	387	13448	236	710344
23.1.2020	1449	387	13444	236	710329
30.3.2020	1447	375	13406	210	708753
27.11.2020	1447	375	13570	210	717859

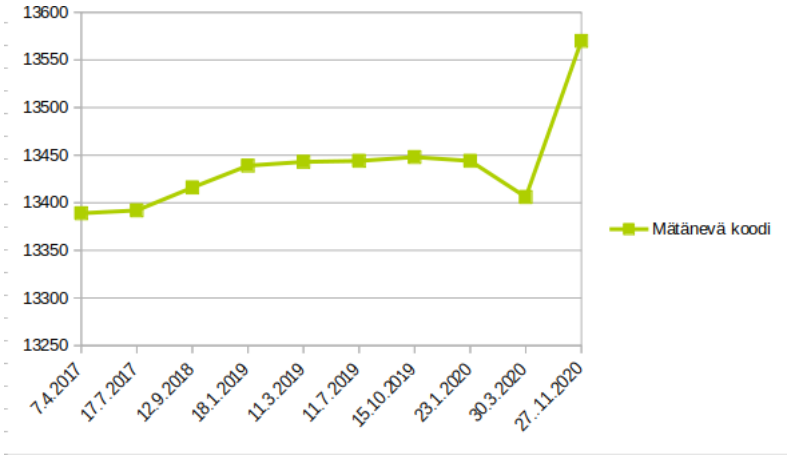
Taulukko 6.1: Sonarqubin staattisen analyysin tulokset

6.2 Git

Muutosten määrä kommiteissa heti luovutuksen jälkeen oli lähes olematonta, kun tutkimme kommittien määrää kuvasta 6.4. Vasta 2017-luvun loppu puolella tule merkittävä piikki muutoksessa. Nousu oli kuitenkin vain hetkellinen, ja aloittaa jyr-

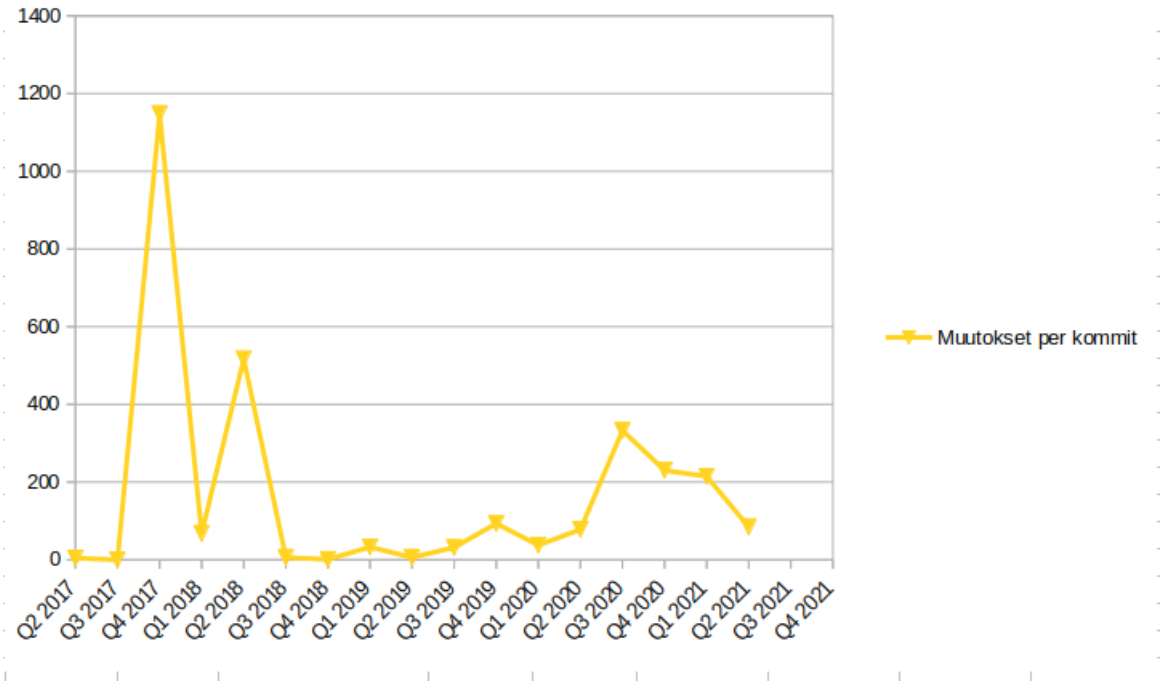


Kuva 6.1: Rivimäärä muutos kotisivukoneessa



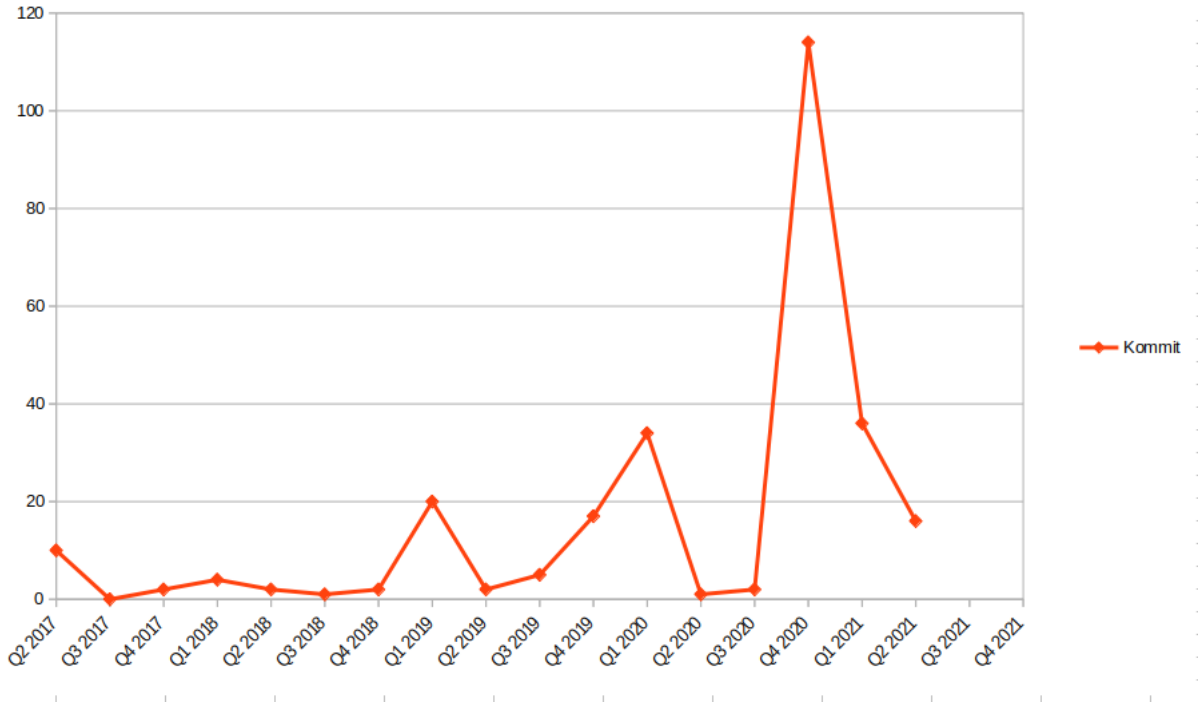
Kuva 6.2: Mätänevän koodin muutos kotisivukoneessa

kän laskun seuraavissa otos pisteissä. Suurimman osan ajasta muutoksia mitä tehdään kommiteissa ovat hyvin minimaalisia. Näiden kommittien suuruus luokka oli sadan rivin muutoksia. Kommittien puolesta tulokset näyttävät hyvin selkeiltä. Tuloksista näkyy, että K1 on toteutunut. Kommittien koot ovat selkeästi pienempiä verrattuna vanhojen tekijöiden kommittien kokoihin. Kommittien määrässäkin on K2 kysymys tullut toteen. Projektiin on tehty alle kymmenen kommittia kolmessa kuukaudessa luovutuksen jälkeen. Vasta vuonna 2019 Vincitin kehittäjä tekee selkeästi enemmän kommitteja. Huomaamme myös että kommittien teossa näyttäisi

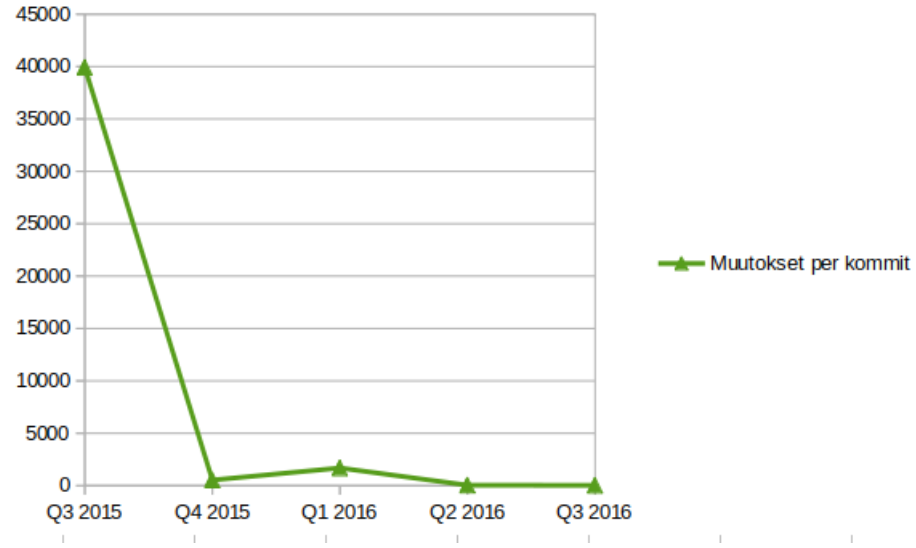


Kuva 6.3: Vincit työntekijän muutokset per kommitti

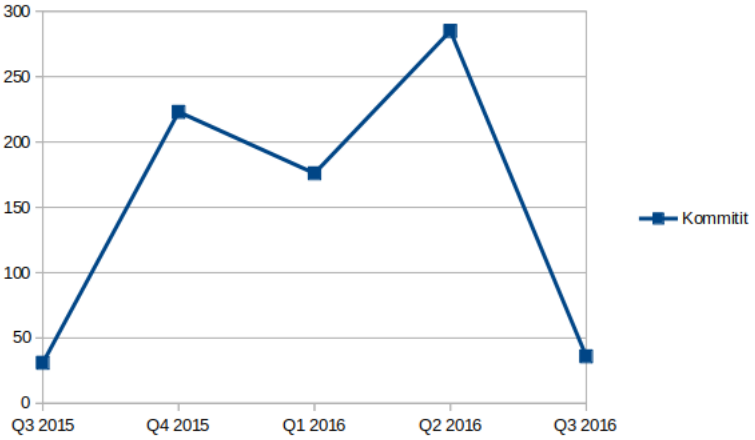
olevan vuosittainen syklisyys. Kesä kaudella muutoksia tehdään hyvin vähän mutta syksyllä ja talvella muutosten määrä kasvaa tasaisesti verrattuna edellisien vuosien samoihin aikoihin Kuvat: 6.3 ja 6.4. Vincitin kehittäjällä on vaikeuksia päästä samanlaiseen kehitys tasoon kuin vanha, kun vertaa lukuja. Tämä voidaan huomata vertailemalaa kahden aikaisemman kehittäjän tuloksia kuvista 6.5 ja 6.7. Tilastoissa myös huomataan, että aikaisemmat kehittäjät lopettivat kehityksen jo 2016-luvun alkupuolella kun katsommet tehtyjä kommitteja kuvista 6.6 ja 6.8. Tällöin kehityksessä oli kokonainen vuosi taukoa ilman että järjestelmässä tehtiin yhtään ohjelmisto kehitystä.



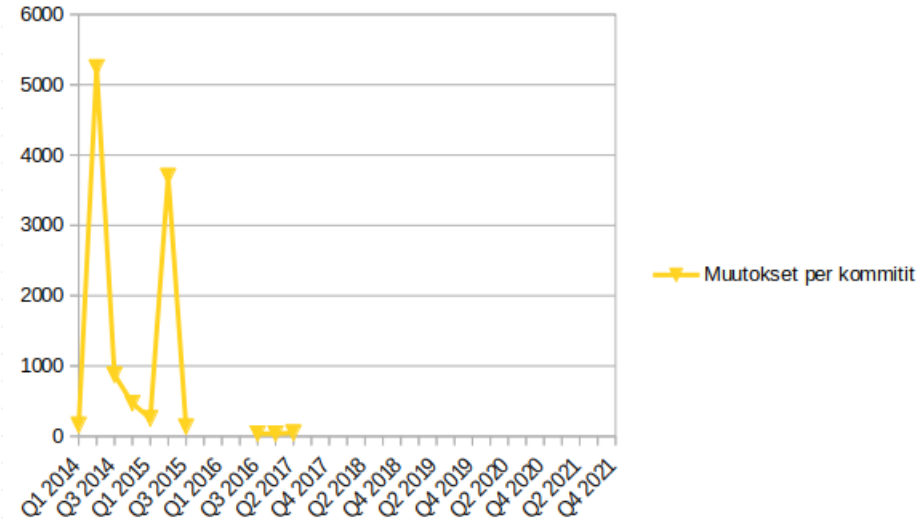
Kuva 6.4: Vincit työntekijän kommittien määrä neljässä vuodessa



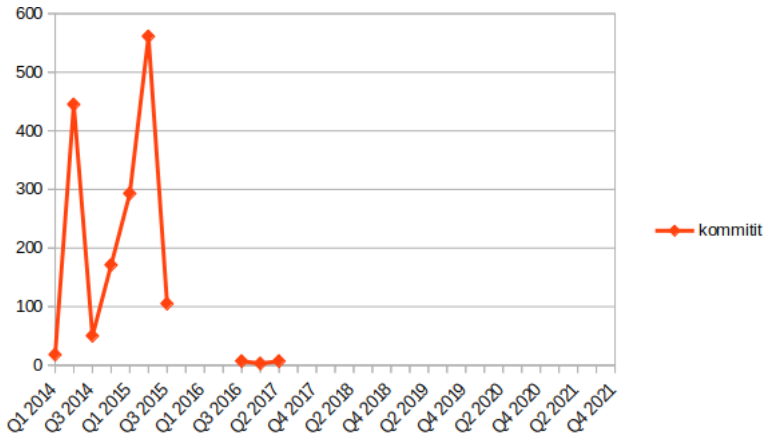
Kuva 6.5: Fonecta työntekijän muutokset keskimääräinen koko yksittäisessä kommitissa



Kuva 6.6: Fonecta työntekijän kommittien määrä neljännes vuodessa



Kuva 6.7: Toisen Fonecta työntekijän kommittien suuruus riveinä



Kuva 6.8: Toisen Fonecta työntekijän kommittien suuruus riveinä

7 Yhteenveto

Huomaamme tuloksista, että suuria piikkejä kommittien muutos suuruuksissa on luovutuksen jälkeen. Tämä mahdollisesti todistaisi hypoteesin vääräksi. Piikit kuitenkin johtuvat kehittäjän lisäämistä kirjastoista järjestelmään, jotka on lisätty lähdekoodiin. Tämä vääristää tuloksia merkittävästi. Ohjelmassa on kirjaston hallinta ohjelma Gradle. Siitä huolimatta kehittäjä on laittanut koko kirjaston lähdekoodin ohjelman omaan lähdekoodiin. Tällöin sen koodisto tulee näkymään tuloksissa 2017 vuonna ilmestyneissä piikeissä. Kirjaston lisääminen lähdekoodiin on huonoa ohjelmistokehitystä ja tuo mukanaan muita hankaluuksia. Tämä vaikeuttaa tilastollisen arvion tekemistä onko kehityksessä tapahtunut granularimaisuutta, ja vaikeuttaa ulkopuolisten kirjastojen seuraamista. Tällaiset kirjastot ovat myös ongelmallisia tulosten tulkinnassa. Sonarqube ei esimerkiksi huomaa näiden kirjastojen mahdollisesti tuomaa teknistä velkaa, että kirjasto on voinut vanhentua.

Hypoteesi K2 on myös toteutunut. Kehitys on merkittävästi hidastunut verrattuna luovutusta edeltäviin kehittäjiin. Tässäkin kuitenkin kehittäjä on harrastanut huonoja tapoja, jotka aiheuttavat vaikeuksia tulosten tarkkaan arvioimiseen. Ongelmaksi ilmeni kommittien päivämäärät. Kommitti jonka päivämäärä on 2017 vuonna saattoi löytyä 2018 kommittien välistä. Tutkimalla asiaa löysin, että kehittäjä oli kehittänyt pitkään erillisessä haarassa muutosta. Tämän valmistuttua hän murskasi(squash) kommitit yhteen ja liitti ne päähaaraan. Tällöin kommitin päivämääräksi tuli kehityshaaran ensimmäisen kommitin päivämäärä. Vaikka kehityshaara liitettiin

päähaaraan vuotta myöhemmin, milloin ensimmäinen kommitti tehtiin.

K1 ja K2 hypoteesien toteutumisessa pitää ottaa huomioon työnteon tehokkuuden vaihtelu vuosi tasolla. Usein kesäisin ihmiset ovat lomilla ja silloin kehitys on pysähtynyt. Vaikka tämä huomioidaan tuloksissa näemme, että kehitys on merkittävästi hidastunut kahteen aikaisempaan kehittäjään. Vaikka otamme huomioon piikit ja sivuhaarat, joissa kehittäjä on tehnyt kommitteja tulokset näyttäisivät suosivan hypoteesia. Tulosta vaikeuttavien tekijöiden takia ei voida kuitenkaan varmistaa varmasti 1 ja 2 hypoteesia.

Kolmas hypoteesi on selkeästi tullut toteen. Teknisen velan määrä on noussut luovutuksen jälkeen. Prosentuaalisesti tämä on hyvin pieni. Tämä pitää kuitenkin verrata määrään koodia, joka on tehty ja miten paljon on entuudestaan teknistä velkaa. Kolmen vuoden aikana uusi kehittäjä loi noin 10 000 uutta riviä koodia. Prosentuaalisesti tämä koodi rivien määrä kasvoi 1,5 prosenttia. Analysoinnissa huomataan teknisen velan kasvun olevan 1,35 prosenttia. Pystymme huomaamaan, että tilastollisesti tenistä velkaa on muodostunut lähes samaan tahtiin kuin uutta koodia. Huomaamme myös, että testien määrä on tilastoissa vähentynyt. Tämä johtuu testien kommentoinnista. Mahdollisia syitä tälle voi olla rikkinäiset testit, joita uusi kehittäjä ei osannut korjata, ja päätti ratkaista ongelman ottamalla teknistä velkaa. Poistamalla tai kommentoimalla testin. Vaikkakin Sonarqube antoi selkeää tulosta, että teknistä velkaa on muodostunut, jos katsoo staattista koodi analyysia ja koodi semantiikka. Pitää ottaa huomioon, että Sonarqube ei tutki mahdollisia ongelmia kirjastoissa. Teknisen velan määrän takia on vaikeaa käyttää valineitä, joilla pystyisi tutkimaan kolmannen osapuolen kirjastojen vanhettumista ja tietoturvallisuutta. Mutta vähäisellä tutkimuksella löydetään kirjastoja, joiden versiot ovat yli 7 vuotta vanhoja ja sisältävät yli 50 tietoturvallisuus haavoittuvuutta. Kaikkea teknisen velan vaikutuksia ja vahinkoja on vaikea arvioida, koska teknisen velan luonnon takia voi olla, että kaikki eivät ole vielä edes ilmaantuneet mitä on kerätty 2017 vuoden

jälkeen. Vaarana on edelleen, että on kerätty teknistä velkaa, josta ei ole tietoa ja keräämme jatkuvasti enemmän korkoa ja ongelmat realisoituvat tilanteessa, jossa on hyvin vaikeata maksaa kertynyttä teknistä velkaa.

Mahdollista teknisen velan muodostumista olisi pystytty paremmin hallitsemaan luomalla parempi jatkuvan kehityksen ja jatkuvan julkaisun putki CI/CD. Projektilla on putki olemassa mutta sen toteutus on vain osittainen. Putkessa on testit, kokoaminen ja julkaisu tuotantoon. Kuitenkin testien määrä oli kehityksen aikana laskussa, joka vähensi niiden arvoa ja hyötyä. Samoin putkesta puuttui koodin analysointi työkalut, joilla pystyttäisiin tarkkailemaan teknisen velan muodostumista ja paremmin hallitsemaan sitä. Tällöin uusikin kehittäjä pystyisi paremmin varmistamaan, että ei aiheuta uutta tenistä velkaa.

Luovutuksessa oli monia tekijöitä, jotka vaikeuttivat uuden kehittäjän kehitys ja sulavaa siirtymää. Aikaisemmat kehittäjät suurimmaksi osaksi lopettivat kehityksen vuosi ennen kuin Vincitin työntekijä aloitti kehityksen. Tilastollisesti näyttäisi, että selkeä ylläpito ja kehitys toimet lopetettiin vuosi aikaisemmin. Lukuun ottamatta muutamaa minimaalista muutosta 2017 vuonna. Luovutus tehtiin ilmaan, että uudella kehittäjällä oli mitään kontaktia vanhojen kanssa. Uusi kehittäjä joutui kaiken selvittämään itse, koska ei voinut selvittää asioita vanhoilta kehittäjiltä. Toisena suurena ongelmana oli dokumentaatio. Järjestelmästä ei uuden kehittäjän mukaan ollut mitään dokumentaatiota. Tämä on merkittävä hidaste kehityksen saralla, koska kehittäjällä ei ollut mitään tietoa, miten kaikki toimi tai miten kehitystä kuuluisi tehdä. Vasta 2021 vuonna saatiin selville, että järjestelmästä oli dokumentaatio, kuitenkin tästä ei ollut kenelläkään tietoa aikaisemmin tai henkilöt jotka tiesi ei tiedetty kysyä.

Isona ongelma selkeästi siirtymässä oli, ettei vanhat ja uudet kehittäjät ehtineet tehdä tiedon siirtoa ja luovuttaa uudelle kehittäjälle tärkeitä tietoja järjestelmästä. Uudella kehittäjällä oli tarpeeksi hyvä osaaminen jatkaakseen ohjelman kehitystä.

Mutta tiedon menetys vaikutti merkittävästi kehitykseen nopeuteen.

Tuloksia pitää myös katsoa kriittisesti. Kommittien pistäminen yhteen vaikeuttaa tarkkojen tulosten saamista, koska tutkimalla mahdollisia haaroja, jotka liitettiin pää haaraan huomaamme, että kehitystä oli 2017 luvulla enemmän mitä pää haara antaa ymmärtää. Silti pitää huomioida, että mitään uutta ei viety käyttäjille yhden vuoden aikana kaikki oli vain sivuhaarassa. Mutta kommittien pistäminen yhteen emme voi olla täysin varmoja tuloksissa. Kehittäjän on mahdollisesti tehnyt enemmänkin tämän laisia toimen piteitä. Nämä muokkaavat paljonkin data tuloksia. Tarvitaan enemmän tutkimuksia, jotka tutkisivat enemmän tämänkaltaisia tilanteita ja avaisivat mahdollisia ongelmia mitä luovutuksessa on. Tutkimus on osviittaa antava, jotta haluamme varmempia tuloksia meidän pitää saada enemmän järjestelmiä tutkimukseen ja tutkia niissä tehtyjä luovutuksia.

Mahdollisia tulevaisuuden tutkimuksia on myös paremmin onnistuneet luovutukset. Aikaisempi kehittäjä on tavoitettavissa. Luovutuksessa on vanha ja uusi kehittäjä mukana. Dokumentaatiota on tarjolla. Vertaamalla paremmin onnistunut luovutusta ja huonosti mennyttä luovutusta pystymme paremmin arvioimaan onnistuneita elementtejä ja mahdollisia ongelma kohtia, joita kehittäjien kannattaa ottaa huomioon luovutuksessa. Myös mittavampi tutkimus, jossa olisi isompi otanta huonosti tehtyjä luovutuksia, jolloin pystyisimme varmemmin arvioimaan luovutuksen vaikutusta kehitykseen ja kuinka suuri vaikutus sillä voi olla.

Kehitys ehdotuksia mitä voisi antaa parantaakseen, jotta tämän tutkimuksen luovutus olisi mennyt paremmin. Vanhat ja uudet kehittäjät olisi pitänyt päästä keskustelemaan ja vaihtamaan ideoita ennen luovutusta. Talloin tärkeää informaatiota siirtyy uusien ja vanhojen kehittäjien välillä. Samalla siirtyy hiljaista tietoa mitä on vaikea dokumentoida. Luovuttaa uudelle kehittäjälle dokumentaatio järjestelmästä. Tämä auttaa kehittäjää jo ennen luovutusta. Dokumentaation avulla kehittäjä pystyy keräämään perusymmärryksen järjestelmästä ja luomaan tarvitta-

via kysymyksiä kehittäjälle. Teknisen velan hallinta. Mahdollisia tapoja paremmin hallita teknistä velkaa olisi lisätä työkaluja, jotka valvovat teknisen velan osuutta järjestelmässä. Tällaisia työkaluja on esimerkiksi Sonarqube. Niiden avulla pystytään paremmin visualisoimaan järjestelmän tekninen velka ja valvomaan teknisen velan kasvua.

Asiat, jotka tehostavat kehitystä ja mahdollisesti auttavat ohjelmakehityksen luovutuksessa. Arkkitehtuuri koodina helpottaa dokumentointia ja auttaa uutta kehittäjää ymmärtämään nopeammin järjestelmää samalla helpottaa uutta kehittäjää toistamaan luodun ympäristön perustamista uudelleen. Kattava testaus. Luo uudelle kehittäjälle itse varmuutta ja antaa nopeamman palautus kierroksen, jolloin pystyy nopeammin ymmärtämään järjestelmää. Tällaista pystytään tehostamaan testi painotteisella kehityksellä. Kummallakin on mahdollista olla positiivinen vaikutus kehityksen luovutukseen. Tutkimuksia näistä hyödyistä ei ole mutta ovat varteen otettavia tutkimus kohteita.

Lähdeluettelo

[1] W. K. Giloi, ”Konrad Zuse’s Plankalku/spl uml/l: the first high-level,"non von Neumann"programming language”, *IEEE Annals of the History of Computing*, vol. 19, nro 2, s. 17–24, 1997.

[2] M. K. Sharma, ”A study of SDLC to develop well engineered software.”, *International Journal of Advanced Research in Computer Science*, vol. 8, nro 3, 2017.

[3] I. ISO et al., ”ISO/IEC/IEEE 12207: 2017, Systems and software engineering–Software life cycle processes”, *International Organization for Standardization*, 2017.

[4] C. P. ja Alessandro Fillari. ”E3 2017: New God Of War Trailer Released, Release Date Set For Early 2018”. (), url: `https://web.archive.org/web/20180413195439/https://www.gamespot.com/articles/e3-2017-new-god-of-war-trailer-released-release-da/1100-6450844/`. (haettu: 26.05.2022).

[5] E. Maiberg. ”New God of War in Development at Sony Santa Monica”. (), url: `https://www.gamespot.com/articles/new-god-of-war-in-development-at-sony-santa-monica/1100-6424051/`. (haettu: 26.05.2022).

[6] T. Long. ”Sept. 7, 1998: If the Check Says ’Google Inc.,’ We’re ’Google Inc.’” (), url: `https://www.wired.com/2007/09/dayintech-0907/`. (haettu: 26.05.2022).

[7] A. S. Khan ja M. Kajko-Mattsson, "Core handover problems", teoksessa *Proceedings of the 11th International Conference on Product Focused Software*, 2010, s. 135–139.

[8] W. Cunningham. "The WyCash Portfolio Management System". (1992), url: `http://wiki.c2.com/?TechnicalDebt`. (haettu: 26.10.2022).

[9] R. C. Martin, *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

[10] FirstMark. "3 Main Types of Technical Debt and How to Manage Them". (2020), url: `https://hackernoon.com/there-are-3-main-types-of-technical-debt-heres-how-to-manage-them-4a3328a4c50c`. (haettu: 26.10.2022).

[11] LeanIx. "Technical Architecture". (), url: `https://www.leanix.net/en/wiki/ea/technical-architecture#form`. (haettu: 24.01.2022).

[12] A. Martini ja J. Bosch, "The danger of architectural technical debt: Contagious debt and vicious circles", teoksessa *2015 12th Working IEEE/IFIP Conference on Software Architecture*, IEEE, 2015, s. 1–10.

[13] H. Mohanty, J. Mohanty ja A. Balakrishnan, *Trends in software testing*. Springer, 2017.

[14] Y. Shmerlin, D. Kliger ja H. Makabee, "Reducing technical debt: using persuasive technology for encouraging software developers to document Code", teoksessa *International Conference on Advanced Information Systems Engineering*, Springer, 2014, s. 207–212.

[15] B. community. "Warcraft 3 refunded". (), url: `https://www.warcraft3refunded.com/`. (haettu: 13.02.2022).

[16] T. U. S. D. of Justice. ”The Department of Justice Systems Development Life Cycle Guidance Document”. (), url: https://www.justice.gov/archive/jmd/irm/lifecycle/ch1.htm#para1.2. (haettu: 27.05.2022).

[17] N. I. of Standards ja Technology. ”THE SYSTEM DEVELOPMENT LIFE CYCLE (SDLC)”. (2009), url: https://csrc.nist.gov/CSRC/media/Publications/Shared/documents/itl-bulletin/itlbul2009-04.pdf. (haettu: 27.01.2022).

[18] U. N. M. C. A. Panel, ”Symposium on advanced programming methods for digital computers”, *Washington, DC: Office of Naval Research, Dept. of the Navy, OCLC*, vol. 10794738, 1956.

[19] C. Larman ja V. R. Basili, ”Iterative and incremental developments. a brief history”, *Computer*, vol. 36, nro 6, s. 47–56, 2003.

[20] W. C. etl. ”Agile Mannifesto”. (), url: https://agilemanifesto.org/iso/fi/manifesto.html. (haettu: 11.09.2022).

[21] J. Shore ja S. Warden, *The art of agile development*. "O’Reilly Media, Inc.", 2021.

[22] M. Bertrand, *Agile! The good, the hype and the ugly*, 2014.

[23] G. Kim, J. Humble, P. Debois, J. Willis ja N. Forsgren, *The DevOps handbook: How to create world-class agility, reliability, & security in technology organizations*. IT Revolution, 2021.

[24] V. Blomberg, ”Adopting DevOps Principles, Practices and Tools Case Identity Access Management”, Turun kauppakorkeakoulu, 2019.

[25] C. University. ”handover”. (), url: https://dictionary.cambridge.org/dictionary/english/handover. (haettu: 27.01.2022).

[26] K. Ito, H. Washizaki ja Y. Fukazawa, ”Handover anti-patterns”, teoksessa *Proceedings of the 5th Asian Conference on Pattern Language of Programs (Asian PLoP 2016), Taipei, Taiwan*, 2016.

[27] K. Ito, H. Washizaki, J. W. Yoder ja Y. Fukazawa, ”A pattern language for handovers”, teoksessa *Proceedings of the 23rd Conference on Pattern Languages of Programs*, 2016, s. 1–13.

[28] H. WASHIZAKI, J. W. YODER ja Y. FUKAZAWA, ”MORE HANDOVER SOLUTION PATTERNS”,

[29] Fonecta. ”Fonetca Yrityksnä”. (), url: `https://www.fonecta.fi/yritys`. (haettu: 18.10.2022).

[30] A. Kolehmainen. ”Kotisivukoneen omistaja vaihtuu”. (2013), url: `https://www.tivi.fi/uutiset/kotisivukoneen-omistaja-vaihtuu/c8ca8b01-9218-399c-95d6-85fc16e6d96b`. (haettu: 24.09.2021).

[31] Git. ”Getting Started - A Short History of Git”. (), url: `https://git-scm.com/book/en/v2/Getting-Started-A-Short-History-of-Git`. (haettu: 03.07.2022).

[32] B. Chacon Scott ja Straub, *Pro git*. Springer Nature, 2014.

[33] mijacobs ja EdKaim. ”What is Git?” (), url: `https://docs.microsoft.com/en-us/devops/develop/git/what-is-git`. (haettu: 03.07.2022).

[34] J. von Gizycki. ”Statistical Code Analysis with SonarQube”. (), url: `https://www.triology.de/en/blog-entries/statistical-code-analysis-with-sonarqube`. (haettu: 04.07.2022).

[35] Sonarqube. ”Sonarqube Docs”. (), url: `https://docs.sonarqube.org/latest/`. (haettu: 04.07.2022).

[36] Sonarqube. ”Java static code analysis”. (), url: `https://rules.sonarsource.com/java`. (haettu: 31.10.2022).

[37] A. Katin, V. Lenarduzzi, D. Taibi ja V. Mandić, ”On the Technical Debt Prioritization and Cost Estimation with SonarQube Tool”, teoksessa *Industrial Innovation in Digital Age*, Springer, 2022, s. 302–309.