Refaktorointi osana järjestelmän uudistamista

Henri Ritvanen

Pro gradu –tutkielma

ITÄ-SUOMEN YLIOPISTO

Tietojenkäsittelytieteen laitos

Tietojenkäsittelytiede

Marraskuu 2019

ITÄ-SUOMEN YLIOPISTO, Luonnontieteiden ja metsätieteiden tiedekunta, Kuopio
Tietojenkäsittelytieteen laitos
Tietojenkäsittelytiede

Opiskelija, Henri Ritvanen: Refaktorointi osana järjestelmän uudistamista
Maisterintutkielma, 61 s.
Maisterintutkielman ohjaaja: FM Katja Pietiäinen
Marraskuu 2019

Tutkielman tavoitteena oli kuvata, mitä kaikkia osa-alueita järjestelmän refaktorointi koskee, miten näillä osa-alueilla refaktorointia voidaan suorittaa käytännössä, mitä refaktoroinnilla halutaan saavuttaa ja milloin refaktorointi on kannattava toimenpide järjestelmän uudistamiseksi ja milloin jokin muu, kuten uudelleenmallintaminen tai uudelleenkirjoittaminen on parempi vaihtoehto.

Tutkielmassa käytiin läpi yleisimmät refaktoroinnin riskit ja kerrottiin, miten hyvä refaktorointi voidaan mahdollisesti saavuttaa. Tutkielmassa käytiin myös läpi yleisesti refaktoroinnin historiaa, nykytilannetta ja mahdollista tulevaisuuden refaktorointia.

Tutkielman tekijä työskentelee Tiedolla potilastietojärjestelmän ohjelmistokehittäjänä. Tutkielmaa varten haastateltiin Tiedon asiantuntijoita tarkoituksena löytää yhtäläisyyksiä Tiedon järjestelmän uudistamisen ja kirjallisuusosion tarjoamien refaktorointiasioiden välillä ja hyödyntää tätä tietoa tulevaisuudessa. Tavoitteena on edesauttaa Tiedon järjestelmän ohjelmistokehittämistä löytämällä tutkielman avulla tärkeimmät refaktoroinnin tavoitteet ja toimintatavat osana järjestelmän uudistamista.

Avainsanat: refaktorointi, järjestelmän uudistaminen, automaattinen testaaminen

ACM-luokat (ACM Computing Classification System, 2012 versio):
• Software and its engineering • Software and its engineering~Software system structures • Software and its engineering~Software architectures • Software and its engineering~Requirements analysis • Software and its engineering~Software design engineering • Software and its engineering~Software implementation planning • Software and its engineering~Software testing and debugging • Software and its engineering~Software evolution • Software and its engineering~Maintaining software • Software and its engineering~Interoperability • Software and its engineering~Software reliability • Software and its engineering~Software usability • Software and its engineering~Risk management • Software and its engineering~Software performance • Software and its engineering~Software safety

UNIVERSITY OF EASTERN FINLAND, Faculty of Science and Forestry, Kuopio
School of Computing
Computer Science

Student, Henri Ritvanen: Refactoring as a part of a software system renewal
Master's Thesis, 61 p.
Supervisor of the Master's Thesis: M.Sc. Katja Pietiäinen
November 2019

The goal of the study was to reveal what are all the areas where refactoring can be used in a software system, how refactoring can be put into practice in these areas, what is to be achieved by using refactoring and when refactoring is the way to go for software system renewal compared to for example reengineering or rewriting the system.

The study reviewed the most common risks of refactoring and told how it is possible to achieve a good refactoring. The study also goes through the history, the present and the possible future of refactoring.

The author works as a software developer at Tieto. Besides using literature, Tieto's experts were interviewed for the purpose of finding similarities between renewal of Tieto's software system and refactoring cases of literature and using this information in the future. The goal is to help development of Tieto's software system by finding the most important goals and methods in refactoring as a part of software system renewal.

Keywords: refactoring, renewal of a software system, automated testing

CR Categories (ACM Computing Classification System, 2012 version):
• Software and its engineering • Software and its engineering~Software system structures • Software and its engineering~Software architectures • Software and its engineering~Requirements analysis • Software and its engineering~Software design engineering • Software and its engineering~Software implementation planning • Software and its engineering~Software testing and debugging • Software and its engineering~Software evolution • Software and its engineering~Maintaining software • Software and its engineering~Interoperability • Software and its engineering~Software reliability • Software and its engineering~Software usability • Software and its engineering~Risk management • Software and its engineering~Software performance • Software and its engineering~Software safety

Lyhenneluettelo

API	Application Programming Interface; ohjelmointirajapinta määrittelee, miten ohjelmisto kommunikoi muiden ohjelmistojen kanssa eli miten ohjelmisto tekee pyyntöjä ja tarjoaa tietoja tai palveluita sovelluksille tai muille tietojärjestelmille.
APO	Area Product Owner; APOn päävastuina ovat tuotteen etenemis- ja julkaisusuunnitelmat, tuotteen materiaali ja tuotteistaminen, myyntituki, kommunikointi, tiedonjako ja yhteistyö.
HTTP	HyperText Transfer Protocol; HTTP on protokolla, jota selaimet ja WWW-palvelimet käyttävät tiedonsiirtoon. HTTP määrittää miten viestit muotoillaan ja välitetään, ja mitä toimintoja WWW-palvelimien ja selaimien pitää tehdä vastauksena erilaisiin käskyihin.
IT	Information Technology; informaatioteknologia on tietokoneiden ja digitaalisen tietoliikenteen avulla tehtävää tietojen muokkaamista, siirtoa, tallennusta ja hakua.
MDA	Model-Driven Architecture; mallipohjaisen arkkitehtuurin tavoitteena on tehdä malleista ohjelmistokehittämisen päätuotteita ja sen myötä mahdollistaa mallien muuttaminen ja kehittyminen.
OPO	Operative Product Owner; OPOn tehtävänä on varmistaa, että tiimit kehittävät oikeita asioita, jotka tuottavat asiakasarvoa. OPO omistaa tuotteen kehitysjonon.
REST	REpresentational State Transfer; REST on HTTP-protokollaan perustuva arkkitehtuurimalli, jonka tarkoitus on asettaa standardeja webissä olevien tietokonejärjestelmien välille ja helpottaa kommunikaatiota näiden järjestelmien välillä. RESTin avulla asiakkaan ja palvelimen toteutukset voidaan luoda ilman, että niiden tarvitsee tietää toisistaan.
SigTest	Signature Test Tool; SigTest on avoimen lähdekoodin työkalukokoelma, jolla voidaan vertailla ohjelmointirajapintoja ja mitata ohjelmointirajapintojen testien kattavuutta.
TDD	Test-Driven Development; TDD:ssä eli testivetoisessa kehittämisessä on tarkoitus kirjoittaa lyhyellä syklillä testi ja koodi, joka saa testin toimimaan sekä suorittaa refaktorointi siten, että lopputulos on mahdollisimman hyvä.
UI	User Interface; käyttöliittymän kautta käyttäjä käyttää ohjelmistoa, laitetta tai muuta tuotteen osaa.

UML — Unified Modeling Language; UML sisältää erilaisia rakenne-, käyttäytymis- ja vuorovaikutuskaavioita, joiden avulla voidaan kuvata järjestelmää erilaisista näkökulmista. UML:ää käytetään useissa ohjelmistokehityksen vaiheissa lähdekoodin ja testitapausten generointiin, järjestelmän dokumentointiin, järjestelmän laadun ennustamiseen ja kommunikaatioon.

Sisällysluettelo

1 Johdanto 1
2 Miksi refaktorointia tehdään? 3
2.1 Ohjelmiston malli/suunnittelu 4
2.2 Ohjelmiston kehittämisnopeus 5
2.3 Ohjelmiston ylläpidettävyys 6
2.4 Ohjelmiston suorituskyky 7
2.5 Ohjelmiston käytettävyys 8
3 Milloin refaktoroida? 12
3.1 Milloin refaktorointia tehdään? 12
3.2 Milloin refaktorointi ei kannata eikä riitä? 14
4 Refaktoroinnin historia, nykytilanne ja tulevaisuus 16
4.1 Refaktoroinnin historia 16
4.2 Refaktorointi nykyään 17
4.3 Refaktoroinnin tulevaisuus 18
5 Mitä järjestelmän osa-alueita refaktorointi koskee? 20
5.1 Koodi 20
5.2 Arkkitehtuuri 24
5.3 Ohjelmointirajapinnat 30
5.4 Testaaminen 31
6 Refaktoroinnin onnistuminen ja riskit 35
6.1 Refaktoroinnin onnistuminen 35
6.2 Refaktoroinnin riskit 37
7 Tutkimusmenetelmät 39
8 Tiedon laskutusjärjestelmän uudistaminen 41
8.1 Järjestelmän historiaa 41
8.2 Nykyisen järjestelmän yleiskatsaus 42
8.3 Järjestelmän uudistamistavat yleisellä tasolla 43
8.4 Refaktoroitavan järjestelmän vaatimukset 46
8.5 Järjestelmän suositellut uudistamisalueet 48
8.6 Mitä onnistunut refaktorointi vaatii ja mitä riskejä se sisältää? 51
8.7 Uudistamisen vaikutus bisnekseen ja käytettävät resurssit 53
9 Yhteenveto 55
Viitteet 58

1 JOHDANTO

Ohjelmiston jatkuva kehittyminen on pakollista, jotta yritys voi pysyä kilpailukykyisenä. Alkuperäinen suunnitelma harvoin on kelvollinen muuttuvien teknologioiden ja ympäristöjen myötä. (Santos et al., 2018.) Refaktoroinnilla tarkoitetaan ohjelmiston koodin suunnittelun parantamista sen jälkeen, kun kyseinen koodi on kirjoitettu (Fowler, 2018). Refaktorointi voi kuitenkin koskea myös läheisesti arkkitehtuuritasoa. Samaa refaktoroinnin käsitettä, ohjelmiston sisäisen rakenteen muuttamista ilman vaikutusta ohjelmiston toimintoihin, käytetään myös arkkitehtuurin puolella. Arkkitehtuurin refaktoroinnissa ohjelmiston toimintomallia muutetaan, mutta semantiikka pidetään samana (Kumar & Kumar, 2011).

Refaktorointi tarkoittaa ohjelmiston sisäisen rakenteen muuttamista muutosten joukolla ilman, että ohjelmiston ulospäin näkyvät toiminnot muuttuvat (Fowler, 2018). Esimerkiksi ohjelmistossa halutaan päästä eroon liiasta globaalista datasta, joka voi aiheuttaa hankaluuksia ohjelmointivirheiden etsimisessä. Ratkaisuna tähän voidaan data kapseloida funktioihin, jolloin nähdään missä datan muokkaaminen tapahtuu. (Fowler, 2018.) Refaktoroinnin pääasiallisena tavoitteena on helpottaa ohjelmiston ymmärtämistä, poistaa ohjelmiston nykyisiä ongelmia ja tehdä ohjelmiston muokkaaminen halvemmaksi (Kaur & Singh, 2017).

Leppänen et al. (2015) haastattelivat yhteensä 12 kokenutta, vähintään 10 vuotta työskennellyttä, ohjelmistoarkkitehtiä ja -kehittäjää 10 eri yrityksestä. Leppänen et al. (2015) huomasivat, että nämä ohjelmistoarkkitehdit ja -kehittäjät ymmärsivät usein refaktoroinnin järjestelmän uudelleenrakentamisena tai -mallintamisena eivätkä he niinkään ajatelleet refaktoroinnin koskevan päivittäisiä, pieniä, koodin laatua parantavia muutoksia. Refaktoroinnin käsitettä ei siis ole ymmärretty aina täysin oikein. On olemassa eri kokoluokkien refaktorointeja, mutta yleisesti suotavinta olisi suorittaa pieniä, jatkuvia refaktorointeja, koska suurten refaktorointien suorittaminen tuo mukanaan riskin epäonnistua. (Leppänen et al., 2015.)

Tämän tutkielman tarkoituksena on selventää refaktoroinnin käsitettä sekä tarkastella kahta refaktorointiin liittyvää tutkimuskysymystä:

1. Mitkä ovat tärkeimmät tavoitteet järjestelmän refaktoroinnissa ja miksi?
2. Milloin refaktorointi on hyvä vaihtoehto järjestelmän uudistamiselle ja milloin kannattaa tehdä jotain muuta kuin refaktoroida, esimerkiksi uudelleenmallintaa tai uudelleenkirjoittaa järjestelmää?

Luvussa 2 käydään läpi miksi refaktorointia ylipäätään tehdään ja mitä mahdollisia konkreettisia hyötyjä siitä on, kun se tehdään oikein. Luvussa 3 käsitellään yleisesti, milloin refaktorointia tulisi tehdä sekä pohditaan milloin kannattaisi mahdollisesti suunnata käyttämään muita mahdollisia järjestelmän uudistamistapoja. Luvussa 4 kerrotaan refaktoroinnin historiasta, nykytilanteesta sekä mahdollisesta tulevaisuudesta. Luvussa 5 käydään läpi mitä kaikkia osa-alueita refaktorointi koskee. Refaktorointi itsessään ulottuu koodipuolen lisäksi esimerkiksi arkkitehtuurin, ohjelmointirajapintojen ja testaamisen puolelle. Kyseisessä luvussa käydään läpi, mitä asioita näitä osa-alueita käsitellessä tulee ottaa huomioon refaktorointia tehdessä ja mistä voi huomata, että järjestelmä vaatii refaktorointia. Luvussa 6 käsitellään refaktorointia riskien ja onnistumisien näkökulmasta. Luvussa avataan, mitä tulee ottaa huomioon, jotta riskit saadaan mahdollisimman matalalle tasolle ja mitä refaktoroinnin onnistumisia edistäviä toimintamalleja on todettu. Luvussa 7 kerrotaan Tiedon laskutusjärjestelmän uudistamista varten käytetyistä tutkimusmenetelmistä. Luvussa 8 käydään läpi Tiedon laskutusjärjestelmän tilannetta ja refaktorointia järjestelmän mahdollisena uudistamistapana. Luvussa 9 on yhteenveto, jossa vastataan tutkimuskysymyksiin.

2 MIKSI REFAKTOROINTIA TEHDÄÄN?

Refaktorointia pidetään ehdottomana osana ketterää ohjelmistokehittämistä. Vaikka ohjelmistokehittäjien oletetaan jatkuvasti parantavan ohjelmiston laatua, heitä usein painostetaan refaktoroinnin sijaan lisäämään ohjelmistoon uusia ominaisuuksia, koska liiketoiminnan kasvaminen vaatii nopeaa uusien toiminnallisuuksien toimittamista käyttäjille. (Leppänen et al., 2015). Leppänen et al.:in (2015) haastattelemat ohjelmistoarkkitehdit ja -kehittäjät kokivat, että refaktorointi on arvokas toimenpide, mutta sen käyttöä on hankalaa perustella asiakkaille tai johtoryhmälle johtuen siitä, että refaktorointi ei määritelmänsä mukaan muuta koodin käyttäytymistä. Jos kehittämistyö suuntautui ohjelmistoa kehittävän yrityksen sisälle, refaktorointi oli yleisempää kuin kiireellisessä tilanteessa, jossa haluttiin tarjota uusia ominaisuuksia asiakkaille.

Välillä ohjelmistokehittäjät voivat kiireen takia joutua tekemään nopeita ja välttäviä ratkaisuja, vaikka ohjelmistokehittäjät ovatkin tietoisia tällaisen toimintatavan mahdollisista seurauksista (Leppänen et al., 2015). Tämän takia voidaan joskus joutua tekemään isompia refaktorointeja, joilla pyritään kerralla parantamaan isompaa joukkoa järjestelmän ongelmakohtia. Ennakkoon suunniteltu refaktorointi ei aina ole huono ratkaisu, mutta sen tulisi kuitenkin olla harvinainen tapa refaktoroida sen mukanaan tuomien mahdollisten ongelmien takia. Jos joudutaan tekemään isompia, ennakkoon suunniteltuja refaktorointeja, tiimin tulee mahdollisuuksien mukaan eristää refaktorointityö ja uusien ominaisuuksien lisäämistyö toisistaan, jolloin muutosten arviointi ja hyväksyntä voidaan toteuttaa erillisesti. Tämä on tärkeää, jotta tiimi voisi tulevaisuudessa lisätä paremmin uusia ominaisuuksia ohjelmistoon. (Fowler, 2018.)

Refaktorointia voidaan käyttää myös järjestelmän uudistamiseen isommassa mittakaavassa. Joskus on tarve uudistaa lähes koko järjestelmä ja tällöin järjestelmän uudelleen tekeminen alusta asti voi olla houkuttelevaa, mutta myös hankalaa ja kallista. Refaktorointia voidaan käyttää myös kompromissina järjestelmän kokonaisen uudelleenkirjoittamisen ja pienempien muutosten välillä. Yhteensopivuus aikaisemman tuotteen kanssa on todella tärkeä tekijä monelle monimutkaiselle ohjelmistolle. Kokonaan uu-

delleenkirjoitetulla järjestelmällä on paljon suurempi riski epäonnistua yhteensopivuuden kanssa kuin järjestelmällä, joka refaktoroidaan ja johon sitten lisätään uusia ominaisuuksia. Suurissa järjestelmissä täydellinen uudelleenkirjoittaminen voi aiheuttaa vanhan järjestelmän toimintojen väärin käyttämistä, etenkin jos kehittäjätiimi on uudehko tai saatavilla ei ole tarpeeksi kattavaa dokumentaatiota. (Mancl, 2001.)

Refaktorointi on arvokas työkalu järjestelmän laadun ylläpitämiseen (Leppänen et al., 2015). Refaktoroinnista on hyötyä arkkitehtuuripuolella, ohjelmiston kehittämisnopeudessa ja ymmärrettävyydessä, ohjelmointivirheiden löytymisessä sekä mahdollisesti suorituskyvyssä ja käytettävyydessä. (Fowler, 2018.) Refaktorointi voi auttaa ohjelmistokehittäjiä myös kehittymään henkilökohtaisella tasolla ohjelmoinnissa, koska refaktoroidessa täytyy miettiä parhaita ohjelmointiratkaisutapoja erilaisista näkökulmista. (Leppänen et al., 2015)

Näistä syistä on kerrottu tarkemmin luvussa 2.1 ja sen aliluvuissa. Luvussa 2.1 on mainittu refaktoroinnin vaikutuksia, muun muassa hyötyjä, koskien ohjelmiston mallia/arkkitehtuuria, luvussa 2.2 puolestaan kuvataan refaktoroinnin vaikutusta ohjelmiston kehittämisnopeuteen, luvussa 2.3 puhutaan refaktoroinnin suhteesta ohjelmiston ylläpidettävyyteen, luvussa 2.4 kerrotaan refaktoroinnin vaikutuksista suorituskykyyn ja luvussa 2.5 kerrotaan refaktoroinnin osuudesta ohjelmiston käytettävyyteen.

2.1 Ohjelmiston malli/suunnittelu

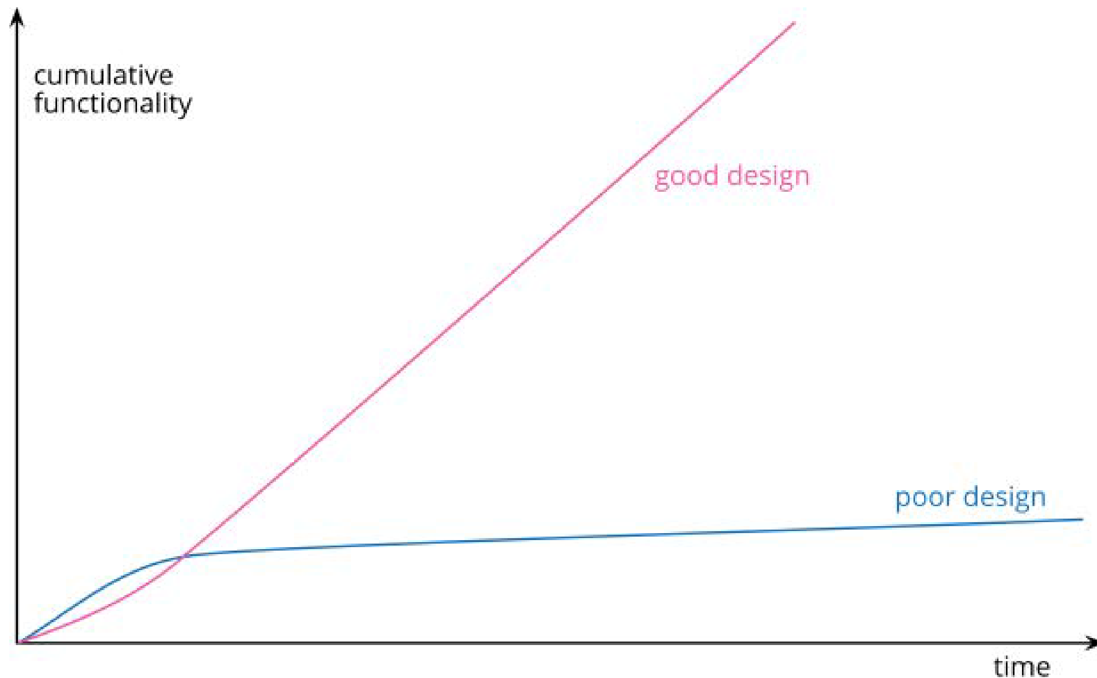
Ohjelmiston arkkitehtuurilla on tapana heikentyä ilman refaktorointia. Koodin rakenne on uhattuna, jos koodin jatkuvaa muuttamista tehdään vain lyhyen tähtäimen tavoitteita ajatellen ymmärtämättä taustalla olevaa arkkitehtuuria. Jos tällaista kehittämistä on tehty, tarvitaan usein enemmän koodia tekemään asioita kuin muuten tarvittaisiin. Jos ohjelmiston kehittämisessä on puolestaan otettu huomioon pidemmän tähtäimen tavoitteet, kun koodia on tehty, niin ohjelmiston onnistunut muokkaaminen on helpompaa. Esimerkiksi huonosti ylläpidetyssä ohjelmistossa muutos yhdessä paikkaa

koodia ei välttämättä tuota järjestelmään haluttua toiminnallisuutta, mikäli samanlaista, kahdennettua koodia käytetään muualla ohjelmistossa ja täten sitäkin tulee muuttaa. (Fowler, 2018.)

2.2 Ohjelmiston kehittämisnopeus

Refaktoroinnin tarkoituksena on ennen kaikkea helpottaa ohjelmistokehittäjien työtä. Fowlerin (2018) mukaan ohjelmistokehittäjien yleinen ongelma on ohjelmiston kehittämisnopeuden hidastuminen. Ohjelmistokehittäjät kertoivat, että aluksi uusien ominaisuuksien lisääminen oli nopeaa, mutta tahti on hiipunut, kun ohjelmistoon on lisättynä lisää ominaisuuksia.

Kuvassa 1 (Fowler, 2018) on kuvattu ohjelmistokehittämisen kokonaisnopeutta, kun ohjelmistokehittämisessä käytetty malli/arkkitehtuuri on joko huonolla tai hyvällä tasolla. Kuten kuvasta 1 nähdään, huonon rakenteen omaavan ohjelmiston kehittäminen voi olla aluksi nopeaa johtuen nopeasti tehdyistä päätöksistä koskien rakennetta. Tahti kuitenkin hidastuu pian ja pysyy matalalla tasolla koko ohjelmiston kehittämisen loppuajan. Hyvällä tavalla suunnitellun ohjelmiston kehittäminen puolestaan voi alussa tuntua hitaalta, mutta nopea tahti kyetään ylläpitämään ohjelmistokehittämisen kannalta myös tulevaisuudessa. Myös Leppänen et al.:in (2015) haastattelemat ohjelmistoarkkitehdit ja -kehittäjät toteavat, että refaktoroinnista on selvästi hyötyä kehittämisnopeudelle, kun koodi selkeytyy ja täten uusien ominaisuuksien lisääminen helpottuu.



Kuva 1. Ohjelmistokehittämisen nopeus (Fowler, 2018)

2.3 Ohjelmiston ylläpidettävyys

Ongelmana koodin kehittämisessä on usein se, että unohdetaan ottaa huomioon mahdolliset tulevaisuudessa ohjelmiston kehittämiseen osallistuvat kehittäjät. Refaktoroinnin avulla voidaan lisätä ohjelmiston ymmärrettävyyttä ja ylläpidettävyyttä sekä nykyisten että uusien ohjelmistokehittäjien kohdalla. (Fowler, 2018 & Leppänen et al., 2015). Sitä myötä, kun koodin ymmärrettävyys lisääntyy, myös ohjelmointivirheiden löytäminen nopeutuu ja ohjelmointivirheiden määrä vähenee. (Fowler, 2018.) On huomattu, että yksittäisistä refaktoroinneista ei ole suurta hyötyä ohjelmiston ylläpidettävyyden parantamiseksi, vaan refaktoroinnin tulee tapahtua ohjelmiston sisällä isossa mittakaavassa, jotta muutos olisi kattavampi koko koodissa (Szoke et al., 2018).

Gatrellin & Counsellin (2015) tutkimuksessa vertailtiin refaktoroinnin vaikutusta laajan, kaupallisen C#-ohjelmiston muutos- ja vikaherkkyyteen. Tutkimuksessa käytettiin 1971 refaktorointia ja vertailtiin keskenään refaktoroituja ja refaktoroimattomia luokkia 12 kuukauden ajanjakson jälkeen. Vertailussa arvioitiin luokkien välillä sitä, kuinka usein niitä jouduttiin muokkaamaan tai, virheiden takia, korjaamaan. Tutkimuksessa huomattiin, että refaktoroidun järjestelmän luokkien muutostarve oli hieman

pienempi refaktoroinnin jälkeen kuin ennen refaktorointia ja virhealttius oli vähentynyt huomattavasti refaktoroinnin jälkeen.

2.4 Ohjelmiston suorituskyky

Ohjelmiston suorituskyky on etenkin nykyään tärkeä tekijä ohjelmiston menestymisen kannalta. Suorituskykyä on harvoin käsitelty osana refaktorointia, vaan on enemmänkin keskitytty toiminnallisiin vaatimuksiin. Suorituskyvyn kaltaiset, ei-toiminnalliset ominaisuudet koostuvat useista ohjelmiston näkökulmista, jotka voivat olla staattisuuteen, dynaamisuuteen tai käyttöönottoon liittyviä, joten niitä on vaikeaa hallita. (Arcelli et al., 2018.)

Refaktorointi itsessään ei ole tapa, jolla välttämättä saadaan suoraan lisättyä ohjelmiston suorituskykyä. Kun halutaan luoda ohjelmistosta entistä suorituskykyisempi, on tärkeää ensiksi refaktoroida ohjelmistoa ja sen jälkeen muuttaa ohjelmiston suorituskykyä paremmaksi. Tarkoituksena on löytää koodista ensimmäiseksi ne osa-alueet, jotka vaikuttavat eniten suorituskykyyn, sillä, jos lähdetään muuttamaan ohjelmiston jokaisen osa-alueen koodia tehokkaammaksi, tuhlataan 90 prosenttia työstä. Tämä johtuu siitä, että ohjelmistossa on yleensä tietyt osa-alueet, jotka aiheuttavat suurimman osan suorituskykyongelmista. (Fowler, 2018.) Refaktorointi auttaa tässä, koska sen jälkeen ymmärrys koodista on paremmalla tasolla ja muutosten tekeminen on helpompaa. (Fowler, 2018.) Lyhyellä aikavälillä refaktorointi voi toki hidastaa ohjelmiston nopeuttamista, koska selkeämpi koodi itsessään ei takaa nopeampaa ohjelmistoa, mutta refaktoroinnin jälkeen ohjelmistoa on helpompi muokata suorituskykyisemmäksi kuin mitä se ennen refaktorointia oli (Fowler, 2018).

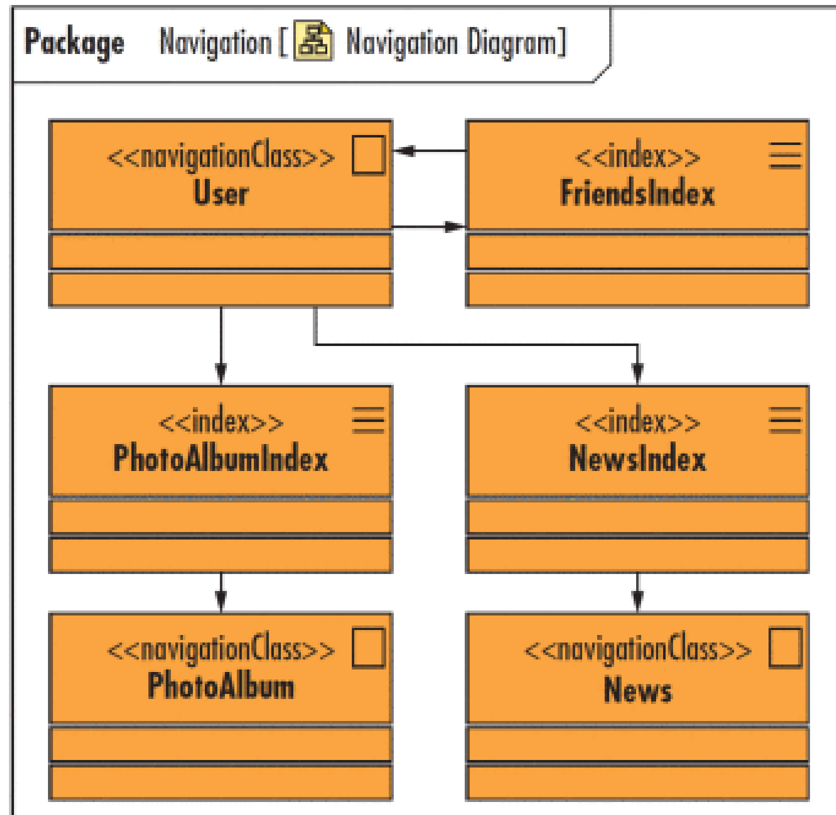
Refaktoroinnissa muutokset tehdään pienissä askelissa ja jokaisen askeleen jälkeen mitataan suorituskyvyn mittaajalla, onko suorituskyky parantunut. Jos ei ole, niin muutokset perutaan ja oikeanlaisen muutoksen etsimistä jatketaan, kunnes suorituskyky on halutulla tasolla. Hyvin ositettu ohjelmisto mahdollistaa ajan käyttämisen suorituskyvyn parantamiseen, koska toiminnallisuuksien lisääminen ohjelmistoon on nopeampaa. Hyvin ositettu ohjelmisto auttaa myös löytämään ohjelmistosta paremmalla

tarkkuudella tietyt kohdat, joita muokkaamalla voidaan pyrkiä parantaa suorituskykyä. Selkeämpi koodi auttaa ymmärtämään, mitä vaihtoehtoja muokkaamiselle on ja miten toteutettu muokkaus toimii käytännössä. (Fowler, 2018.)

2.5 Ohjelmiston käytettävyys

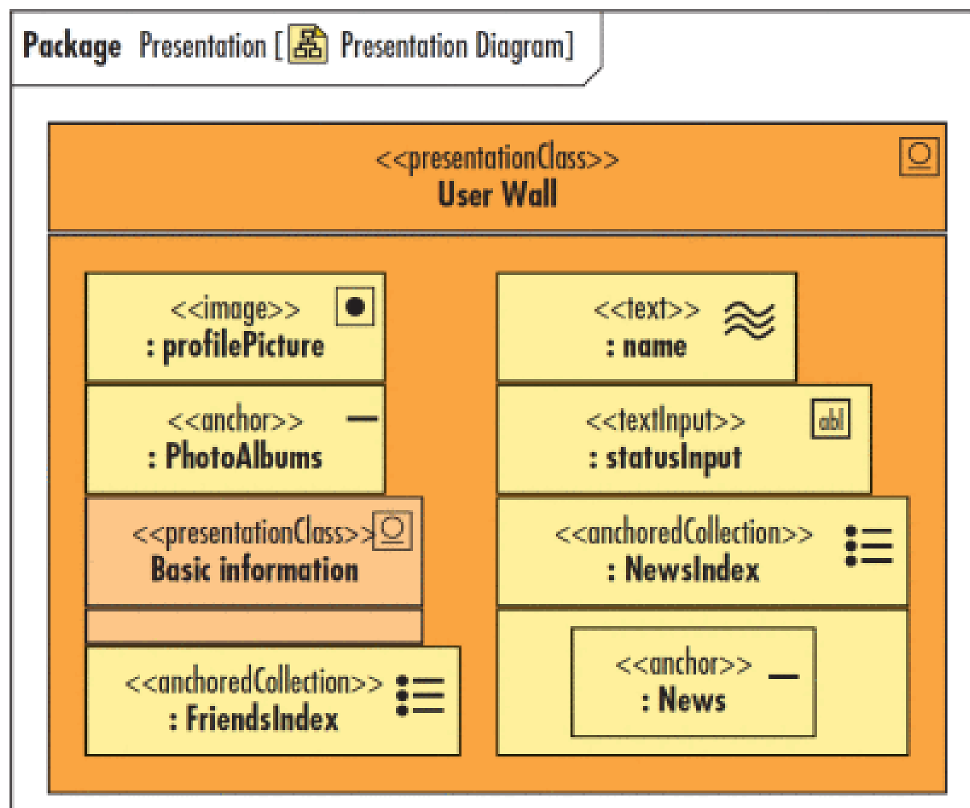
Käytettävyyden puolelta löytyy esimerkkejä refaktoroinnista web-sovelluksien osalta (Garrido et al., 2010 & Garrido et al., 2017). Garrido et al. (2010) jakavat web-sovellusten refaktoroinnin navigaatio- ja presentaatiomalleihin sekä erillisiin käytettävyystekijöihin.

Refaktoroinnin jakaminen navigaatio- ja presentaatiomalleihin helpottaa refaktorointia, koska se tarjoaa vahvan abstraktiotason verrattuna toteutustason refaktorointiin. Alla olevat kuvat 2 ja 3 ovat stereotyyppisiä UML-luokkakaavioita. Navigaatiomalli kartoittaa sisältömallin luokat navigaatiosolmuiksi ja presentaatiomalli määrittelee abstraktin käyttäjärajapinnan. Navigaatiokaavion (Kuva 2) luokat kuvaavat siis solmuja tai indeksejä ja presentaatiokaavion (Kuva 3) luokat kuvaavat web-sovelluksen sivuja. Solmuilla tarkoitetaan informaatio- tai käytösyksikköjä, joita käyttäjä havaitsee. (Garrido et al., 2010.)



Kuva 2. Navigaatiomallikaavio (Garrido et al., 2010)

Navigaatiomallin refaktorointina käsitetään muutokset web-sovelluksen navigaatiomalliin siten, että saatavilla olevien solmujen toiminnallisuus mallissa ja kunkin toiminnon saavutettavuus kotisolmusta pitkin navigaatiopolkua eivät muutu. Navigaatiopuolen refaktorointi sisältää muun muassa solmujen, solmuattribuuttien ja solmuoperaatioiden uudelleennimeämisen, solmujen lisäämisen, ylimääräisten saavuttamattomien solmujen poistamisen, sisällön tai operaatioiden siirtämisen saatavilla olevien solmujen välillä sekä linkkien lisäämisen ja tarpeettomien linkkien ja saavuttamattomien solmujen linkkien poistamisen. (Garrido et al., 2010.)



Kuva 3. Presentaatiomallikaavio (Garrido et al., 2010)

Presentaatiomallin refaktorointina käsitetään muutokset web-sovelluksen presentaatiomalliin siten, että sivujen tarjoamat operaatiot ja semantiikka käsitetään yhtenäisenä ja navigaatiomallin elementtien abstraktien rajapintojen saatavuus ei muutu. Presentaatiomallin refaktoroinnissa jaetaan tai yhdistetään sivuja, vaihdetaan abstraktin pienoisohjelman tyyppiä, jos uusi tyyppi säilyttää taustalla olevan toiminnallisuuden, uudelleenjärjestetään pienoisohjelmien järjestelyä sivulla ja lisätään tai vaihdetaan saatavilla olevia käyttöliittymän efektejä. (Garrido et al., 2010.)

Navigaatio- ja presentaatiomallien lisäksi käytettävyys-refaktorointiin kuuluu osaksi käytettävyyttä suoraan koskevien osa-alueiden refaktorointi. Ohjelmistokehittäjien tulisi miettiä seuraavia osa-alueita, kun tarkoituksena on miettiä, mitä refaktoroinnilla voitaisiin parantaa (Garrido et al., 2010):

- Esteettömyys: Kuinka helppoa web-sovelluksen käyttö on fyysisen vamman omaaville ihmisille tai ihmisille, joilla on käytössä avustavaa teknologiaa?

- Navigoitavuus: Kuinka helppoa on navigoida sovelluksen sisältöä linkkien kautta?

- Tehokkuus: Tarjoaako sovellus nopeita oikoteitä edistyneille käyttäjille?

- Uskottavuus: Voiko sovelluksen toimintaan luottaa ja tukeeko se pitkiä sovelluksen käyttösuhteita?

- Ymmärrettävyys: Ymmärtääkö käyttäjä helposti, mitä kaikkea sovellus tarjoaa, kuinka päästä asioihin käsiksi ja mikä on sovelluksen nykyinen tila?

- Muokattavuus: Voidaanko vastata käyttäjien haluamiin tarpeisiin perustuen aikaisempaan käyttäytymiseen tai käyttöyhteyteen ja voidaanko näyttää kerralla tarpeeksi informaatiota ilman, että informaation määrä häiritsee käyttäjiä?

- Opittavuus: Kuinka helppoa sovellusta on käyttää ja kuinka helppoa oppiminen on ohjatun tuen avulla?

Garrido et al. (2010) suosivat heuristista lähestymistapaa refaktoroinnin tarpeellisuuden arvioinnissa käytettävyyden osalta. Heuristinen arviointi perustuu käytettävyysperusteisiin, käytettävyysongelmien raportointiin ja parannusehdotuksiin. Samoin Garrido et al. (2017) suosivat käyttäjien hyödyntämistä osana web-sovelluksen refaktorointia. He ovat keskittyneet erityisesti käyttäjän vuorovaikutukseen liittyviin kaavoihin, jotka aiheuttavat ongelmallisia vuorovaikutustilanteita.

3 MILLOIN REFAKTOROIDA?

Refaktorointia suodaan yleisesti tehtävän jatkuvana prosessina. Tämä ei kuitenkaan ole aina mahdollista, jos esimerkiksi refaktorointia on laiminlyöty ja siirretty myöhemmäksi kiireiden takia tai vaadittava muutos koskee esimerkiksi riippuvuuksien käsittelyä, mikä vie aikansa, jotta refaktorointiprosessi saadaan onnistuneesti maaliin (Leppänen et al., 2015 & Fowler, 2018). Tämä vaatii isompaa uudistamisprosessia, jota varten refaktorointi on yksi vaihtoehto, järjestelmän uudelleenmallintamisen ja uudelleenkirjoittamisen ohella.

Luvussa 3.1 käydään läpi, milloin refaktorointia tulisi tehdä ja luvussa 3.2 käsitellään tilanteita, jolloin refaktorointi ei ole kannattavin eikä riittävä vaihtoehto, vaan tulee kääntyä muiden ratkaisujen puoleen järjestelmän uudistamista tehtäessä.

3.1 Milloin refaktorointia tehdään?

Kooditasolla refaktoroinnin tulisi olla jatkuvaa. Paras hetki refaktoroida on juuri ennen uuden ominaisuuden lisäämistä koodiin. Tässä vaiheessa on helpompaa nähdä, missä rakenteessa koodi toimii parhaiten. Kun tarvittava muutos on tehty, on uuden ominaisuuden lisääminen helpompaa. Ilman refaktorointia voidaan helposti alkaa kahdentaa koodia ja tämän seurauksena saman koodialueen muuttaminen joudutaan toistamaan jatkossa useammassa kohdassa. (Fowler, 2018.)

Jatkuva refaktorointi ei kuitenkaan vaikuta olevan aina mahdollista. Esimerkiksi Leppänen et al.:in (2015) haastattelemat ohjelmistokehittäjät ja -arkkitehdit kertoivat, että heillä on töissä jatkuva kiire eivätkä he sen takia ehdi luoda hyvää koodia. Nämä henkilöt kertoivat tekevänsä välillä tietoisesti nopeita, välttäviä ratkaisua ja olevansa tietoisia valinnoistaan ja niiden seurauksista.

Ennakkoon suunniteltu refaktorointi ei kuitenkaan ole välttämättä huono ratkaisu. Tämä lähestymistapa voi toisaalta vaatia ohjelmistokehitystiimiltä paljon aikaa, jos

tiimi on aikaisemmin laiminlyönyt refaktorointia. Jotta tiimi voisi tulevaisuudessa lisätä paremmin uusia ominaisuuksia ohjelmistoon, täytyy tiimin mahdollisesti eristää refaktorointityö ja uusien ominaisuuksien lisäämistyö toisistaan, jolloin muutosten arviointi ja hyväksyntä voidaan toteuttaa erillään. Tällaisten refaktorointitapojen tulisi kuitenkin Fowlerin (2018) mukaan olla harvinaisia, koska hänen mukaansa refaktoroinnin on tarkoitus tapahtua jatkuvana kehittämisenä pienissä osissa.

Pitkään kestävä refaktorointi voi johtua esimerkiksi tarpeesta siirtää koodin osioita komponentteihin, joita muut tiimit voivat käyttää, tai selventää koodissa esiintyviä riippuvuuksia. Fowler (2018) on vastahakoinen tällaista omistettua refaktorointia kohtaan ja hänen mukaansa toimiva strategia on refaktoroida asteittaisesti viikkojen aikana. Hyötynä tässä asteittaisessa tavassa on se, että refaktorointi ei missään vaiheessa riko koodia toimimattomaksi. Esimerkkinä Fowler (2018) mainitsee ohjelmistokirjastosta toiseen vaihtamisen, joka aloitetaan tekemällä uusi abstraktio, joka voi toimia joko rajapintana tai kirjastona. Kun koodista kutsutaan tätä abstraktiota, on huomattavasti helpompaa vaihtaa kirjastosta toiseen (Branch By Abstraction -taktiikka).

Arkkitehtuuripuolella ohjelmiston refaktorointi kannattaa, jos se auttaa parantamaan arkkitehtuurimallia paikallisessa näkyvyysalueessa. Jos tavoitteena on selkeyttää monimutkaista mallia tai koodin kehittämistä, korjata ohjelmointivirheitä tai ratkaista arkkitehtuuri- tai kooditason ongelmia, jotka voivat olla peräisin syvemmältä järjestelmästä, on refaktorointi hyvä ratkaisu arkkitehtuurin päivittämiseen. (Stal, 2013.)

Leppänen et al.:n (2015) haastattelemat ohjelmistokehittäjät ja -arkkitehdit kertoivat yleisellä tasolla refaktorointiin johtaviksi syiksi muun muassa kehittämisnopeuden hitauden, koodin sekavuuden, vähäisen yleiskäyttöisyyden tai modulaarisuuden, heikon algoritmisen suorituskyvyn, monimutkaisuuden ja teknologisen velan.

Ketterässä, iteratiivis-inkrementaalisessa kehittämisessä arkkitehtuurin refaktorointi tulisi suorittaa vähintään kerran per iteraatio ja refaktoroinnin tulisi olla myös osa päivittäistä työtä. Etenkin testipainotteisessa kehittämisessä refaktorointi on pakollista. (Stal, 2013.)

3.2 Milloin refaktorointi ei kannata eikä riitä?

Jos koodi on rumannäköistä, mutta sitä ei ole tarvetta muokata jatkossa, ei välttämättä tarvitse tehdä refaktorointia. Kooditasolla refaktorointi ei kannata esimerkiksi silloin, jos rumalta näyttävä, mutta hyvin toimiva koodi on ohjelmointirajapintakäytössä eikä se tarvitse jatkokehittämistä. Ainoastaan silloin kannattaa refaktoroida, kun on tarve ymmärtää koodin toimintalogiikka. Päätös koodin refaktoroinnin ja koodin uudelleenkirjoittamisen välillä vaatii hyvää arviointikykyä ja kokemusta. On todella hankalaa päättää yleisellä tasolla, milloin refaktorointia ei enää kannata tehdä, vaan koodi kannattaa kirjoittaa uudelleen alusta asti. Tämä vaatii yleensä perehtymistä koodiin ja analysointia siitä, miten hankalaa koodista on saada selvää. (Fowler, 2018.)

Arkkitehtuuritasolla tilanteissa, joissa refaktoroinnilla saavutetaan vain oireiden poistuminen, mutta ei juuritason ongelmien korjautumista, kannattaa miettiä muita näkökulmia arkkitehtuurin uudistamiseen, esimerkiksi uudelleenmallintamista tai uudelleenkirjoittamista. Voidaan esimerkiksi huomata, että ohjelmiston nykyinen toimintamalli (design) on jo huonontunut hoitamattoman arkkitehtuurikulumisen myötä. Näiden tilanteiden havaitsemiseen kannattaa käyttää hyödyksi ohjelmistoinsinöörien tietotaitoa. (Stal, 2013.)

Uudelleenmallintaminen on varteenotettava vaihtoehto, jos refaktorointi ei riitä ja muutosta vaativat seuraavat syyt:

- Ohjelmointivirheiden korjaukset, jotka aiheuttavat vikoja muualla ohjelmistossa
- Uudet toiminnalliset ja operatiiviset vaatimukset
- Muuttunut liiketoiminta

Uudelleenmallintamisessa täytyy ottaa lisäksi huomioon, että se vaatii oman erillisen projektin ja lopputuloksena on uusi järjestelmä. Uudelleenmallintaminen edellyttää aina järjestelmällisiä muutoksia perustana olevaan ohjelmistojärjestelmään.

Uudelleenmallintamisen ensimmäisessä vaiheessa koko järjestelmä takaisinmallinnetaan (reverse-engineering) ja sen komponentit arvioidaan käyttäen SWOT (Strenghts, Weaknesses, Opportunities and Threats) -analyysia. Ohjelmistoinsinöörien tehtävänä on ottaa selvää, mitkä komponentit ovat heidän mielestänsä arvokkaita ja uudelleenkäytettäviä. Valitut komponentit toteutetaan osaksi uudelleen suunniteltua ja rakennettua ohjelmistojärjestelmää hyödyntäen refaktorointia osana komponenttien sovittamista. (Stal, 2013.)

Uudelleenkirjoittaminen on varteenotettava vaihtoehto silloin, kun refaktorointi eikä uudelleenmallintaminen riitä ja muutosta vaativat seuraavat syyt:

- Koodin ja toimintamallin epävakaus
- Uudet toiminnalliset ja operatiiviset vaatimukset
- Muuttunut liiketoiminta

Verrattaessa refaktorointia, uudelleenmallintamista ja uudelleenkirjoittamista lopputuloksena refaktoroinnissa kehittämistyö helpottuu ja paranee, kun taas uudelleenmallintamisessa ja uudelleenkirjoittamisessa vaikutukset ulottuvat lisäksi myös toiminnallisuuden ja toimintakyvyn puolelle. (Stal, 2013.)

Taulukko 1. Uudelleenmallintamisen ja uudelleenkirjoittamisen syyt (Stal, 2013)

Uudelleenmallintaminen	**Uudelleenkirjoittaminen**
Ohjelmointivirheiden korjaukset, jotka aiheuttavat vikoja muualla ohjelmistossa	Koodin ja toimintamallin epävakaus
Uudet toiminnalliset ja operatiiviset vaatimukset	Uudet toiminnalliset ja operatiiviset vaatimukset
Muuttunut liiketoiminta	Muuttunut liiketoiminta

4 REFAKTOROINNIN HISTORIA, NYKYTILANNE JA TULEVAISUUS

Aikoinaan, 20 vuotta sitten, ajateltiin, että ohjelmistomallin/-arkkitehtuurin tuli olla valmis ennen kuin ohjelmointityötä aloitettiin (Fowler, 2018). Refaktorointi muuttaa tämän näkemyksen täysin. Refaktoroinnin käsityksen syntymää on vaikea ajoittaa täsmällisesti, sillä hyvät ohjelmoijat ovat aina käyttäneet aikaa koodin siistimiseen. Refaktorointi on kuitenkin laajempi käsite, sillä refaktorointi koskee koko ohjelmistokehittämisprosessia.

Luvussa 4.1 käydään läpi refaktoroinnin historiaa, luvussa 4.2 käydään läpi sitä, miltä refaktorointi näyttää tällä hetkellä ja luvussa 4.3 on pohdittu, mihin suuntaan refaktorointi mahdollisesti tulevaisuudessa tulee menemään.

4.1 Refaktoroinnin historia

Kent Beck ja Ward Cunningham ovat kaksi ensimmäistä ihmisistä, jotka ymmärsivät refaktoroinnin tärkeyden. He työskentelivät Smalltalk-ohjelmointiympäristön parissa 1980-luvulla. Smalltalk on dynaaminen ohjelmointiympäristö, joka mahdollistaa hyvin toimivan ohjelmiston tekemisen nopeaan tahtiin. Ward ja Kent perehtyivät ohjelmistokehittämiseen, joka koskee Smalltalkin tapaisia ohjelmointiympäristöjä ja heidän työnsä kehittyi lopulta Extreme Programmingiksi, joka painottaa korkealaatuista ohjelmistokehittämistä ja korkean laadun ylläpitämistä ohjelmistokehittäjätiimiä ajatellen. (Fowler, 2018.)

Ensimmäinen Extreme Programming -projekti aloitettiin vuonna 1996. (Fowler, 2018, Agile Alliance, 2019 & Extreme Programming, 2013.) Wardin ja Kentin ideat saivat suosiota Smalltalk-yhteisössä ja levisivät hiljalleen eteenpäin. Martin Fowler työskenteli Kentin kanssa ja näki refaktoroinnin tuoman hyödyn ohjelmistokehityksessä. Tämä innosti Martin Fowleria kirjoittamaan Kent Beckin, John Brantin, William Opdyken ja Don Robertsin kanssa refaktoroinnista kirjan (Refactoring: Improving the

Design of Existing Code, 1999), joka auttoi refaktorointia leviämään laajemmin ohjelmistokehittäjien parissa. (Fowler, 2018.)

4.2 Refaktorointi nykyään

Nykyään tiedetään, että on hyvin vaikeaa tehdä etukäteen hyvää ohjelmistomallia/-arkkitehtuuria (design) sovellukselle. Ohjelmiston mallin muuttaminen jälkikäteen on mahdollista ja oleellinen osa ohjelmistokehittämistä, koska sen avulla saadaan nopeasti kehitettyä toiminnallisuuksia ohjelmistoon. (Fowler, 2018.)

Suurin muutos viimeisimmän 10 vuoden aikana, koskien refaktorointia, on automaattinen refaktorointi. Automaattisella refaktoroinnilla tarkoitetaan työkaluja, jotka ehdottavat ohjelmistokehittäjälle refaktorointimuutoksia koodiin. (Fowler, 2018.) Yhdistämällä automatisoidut refaktorointitekniikat, ohjelmistometriikan ja metaheuristiset haut, automaattinen refaktorointityökalu voi parantaa järjestelmän rakennetta vaikuttamatta sen toiminnallisuuteen (Mohan et al., 2016). Mohan et al.:in (2016) tutkimuksessa todettiin, että täysin automatisoitu refaktorointi voi vähentää ohjelmistojärjestelmän teknologista velkaa.

Toisaalta Leppänen et al.:in (2015) tutkimuksessa kyseenalaistetaan automaattisten refaktorointimetriikkatyökalujen hyödyt. Ohjelmistoarkkitehdeistä tai -kehittäjistä osa totesi, että koodin laatua mittaavan metriikan käyttö jopa huononsi kehittämistä, koska johtoryhmälle ei merkinnyt enää mikään muu kuin tietty lukuarvo, jonka arviointityökalu tuotti. Turvallisuuteen liittyvien ohjelmistoyritysten haastateltavat puolestaan käyttivät staattista koodin analysointia löytääkseen ongelmia koodista, mutta he eivät käyttäneet hyödykseen mittareita. Tämä vahvistaa näkemystä siitä, että kehittäjät keskittyvät enimmäkseen konkreettisten ongelmien löytämiseen, eikä mittareiden tuottamien arvojen parantamiseen tai huonojen toimintamallien poistamiseen. Toisaalta osa haastateltavista puolestaan totesi, että työkalut ja metriikka teknologisen velan ja refaktoroinnin tarpeen mittaamista varten olivat hyödyllisiä. Esimerkiksi sisäisen laadun mittareita voitiin käyttää perustelemaan asiakkaalle, minkä takia refaktorointia tulisi tehdä.

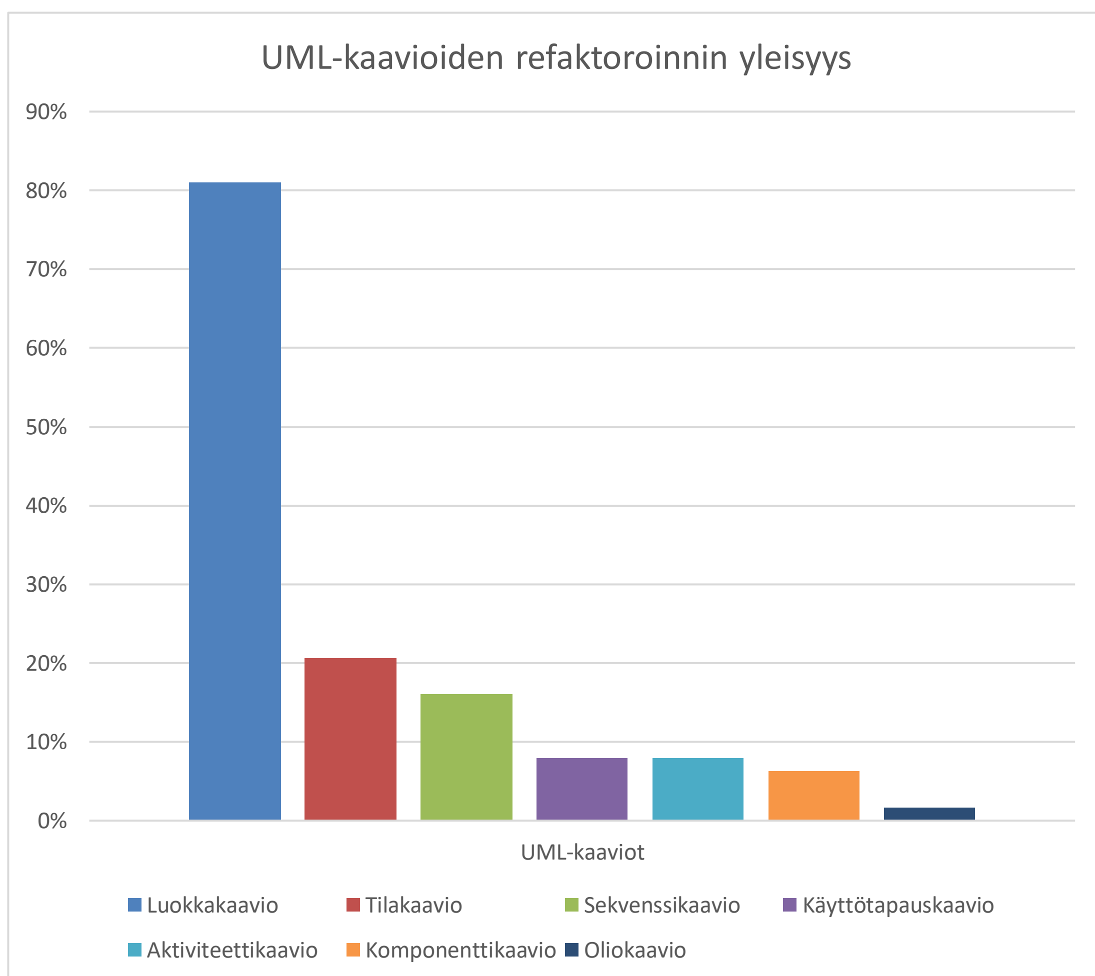
Käytettäessä automaattista refaktorointia on tärkeää tarkistaa, että ehdotettu refaktorointi on oikeanlainen. Automaattiset refaktorointityökalut voivat tarjota hyviä ehdotuksia, joista voi olla apua refaktoroinnissa alkuun pääsemiseksi. On hyvin yleistä nähdä nykyään refaktorointityökaluja useissa ohjelmointieditoreissa, mutta täytyy muistaa, että työkalujen välillä voi olla suuriakin eroja johtuen muun muassa erilaisten ohjelmistokielten rajoitteista koskien automaattista refaktorointia. (Fowler, 2018.) Monet refaktorointityökalut ovat puoliautomaattisia, koska täysin automaattiset refaktorointityökalut kärsivät korkean asteen vääristä positiivisista refaktoroinneista. Täysin automaattisen refaktorointityökalun suorituskykyä voidaan parantaa käyttämällä, saatavuuden mukaan, refaktoroitavien asioiden kartoittamisominaisuutta. Automaatioasteen valinta tapahtuu yleensä asiantuntijan näkemyksen hyödyntämisen ja refaktoroinnin helppouden välillä. (Misbhauddin & Alshayeb, 2013.)

4.3 Refaktoroinnin tulevaisuus

Misbhauddin & Alshayeb (2013) selvittivät 63 erilaisen mallien refaktoroinnista kertovan tutkimuksen avulla, että UML-mallien refaktorointi on nousussa. UML:ään sisältyy erilaisia rakenne-, käyttäytymis- ja vuorovaikutuskaavioita, joiden avulla voidaan kuvata järjestelmää erilaisista näkökulmista (UML, 2019). UML:ää käytetään useissa ohjelmistokehityksen vaiheissa lähdekoodin ja testitapausten generointiin, järjestelmän dokumentointiin, järjestelmän laadun ennustamiseen ja kommunikaatioon. (Misbhauddin & Alshayeb, 2013.)

Mallipohjaisen arkkitehtuurin (Model-Driven Architecture (MDA)) tavoitteena onkin tehdä malleista ohjelmistokehittämisen päätuotteita ja sen myötä mahdollistaa mallien muuttaminen ja kehittyminen. On odotettavissa, että ohjelmistokehityksessä tulee olemaan yhä isommassa roolissa mallit ja niiden käyttö. Sovelluksen refaktorointi mallitasolla tuo mukanaan selkeitä hyötyjä kuten suunnittelutasolla piilossa olevien ongelmien löytämisen, paremman visualisoinnin mahdollisista rakennemuutoksista sekä vaihtoehtoisten suunnittelupolkujen välisen valintamahdollisuuden. (Misbhauddin & Alshayeb, 2013.)

Misbhauddinin & Alshayebin (2013) mukaan UML-mallien refaktorointia täytyy vielä kehittää eteenpäin avoimien ongelmien takia, vaikka niitä varten onkin jo olemassa melko paljon laadukkaita lähestymistapoja ja tekniikoita. Esimerkiksi Misbhauddin & Alshayeb (2013) huomasivat (Kuva 4), että mallien refaktoroinnista kertovissa tutkimuksissa käytettiin refaktoroinnissa vain seitsemää kaaviotyyppiä UML 2.4 standardin 14 kaaviotyypistä. Nämä seitsemän kaaviota olivat laskevassa käyttöasteessa luokkakaavio (81 %), tilakaavio (20,6 %), sekvenssikaavio (16 %), käyttötapauskaavio (7,9 %), aktiviteettikaavio (7,9 %), komponenttikaavio (6,3 %) ja oliokaavio (1,6 %).



Kuva 4. UML-kaavioiden refaktoroinnin yleisyys (Misbhauddin & Alshayeb, 2013)

5 MITÄ JÄRJESTELMÄN OSA-ALUEITA REFAKTOROINTI KOSKEE?

Refaktorointi voi ulottua kattavasti monelle osa-alueelle ohjelmistotuotannossa. Perinteisesti refaktoroinnin on ajateltu koskevan pitkälti bisneslogiikkaa ja siihen liittyvää testaamista (Fowler, 2018), mutta refaktorointi koskee myös arkkitehtuurin uudistamista (Stal, 2013) sekä ohjelmointirajapintoja (Fowler, 2018).

Seuraavassa käydään läpi refaktorointia koodin (luku 5.1), arkkitehtuurin (luku 5.2), ohjelmointirajapintojen (luku 5.3) ja testaamisen (luku 5.4) osalta.

5.1 Koodi

Kooditason refaktoroinnissa metriikkaa voidaan käyttää apuna refaktoroitavien osien löytämisessä, mutta ihmisen intuitio on silti paras väline. Fowler (2018) on koostanut kokoelman syitä, jotka vihjaavat kyseisen koodin kaipaavan refaktorointia:

Dataluokat: Luokat, jotka ovat vain kenttien arvojen hakemista ja asettamista varten. Dataluokat kertovat usein siitä, että toiminta on väärässä paikassa eli on mahdollista edistyä paljon, jos viedään käytös samaan luokkaan, missä data on.

Datarykelmät: Data on usein monessa paikassa rykelmänä: samoja muuttujia käytetään yhdessä parametreina metodeille tai esimerkiksi sama datakimppu esiintyy samojen luokkien kenttinä yhdessä rykelmässä. Jotta tämä saadaan korjattua, tulisi etsiä kohtia, joissa nämä rykelmät esiintyvät kenttinä. Sitten datarykelmä tulisi muuttaa olioksi. Tämän jälkeen tätä oliota kannattaa käyttää mahdollisuuksien mukaan parametrina. Näin parametrilistat pienenevät ja metodien kutsuminen helpottuu huomattavasti.

Globaali data: Globaalin datan ongelma on se, että sitä voidaan muokata mistä tahansa. Esimerkiksi, jos löytyy uusi ohjelmointivirhe, joka aiheutuu globaalista datasta, on todella hankalaa korjata kyseistä ohjelmointivirhettä, koska globaalin datan muok-

kaus on voinut tapahtua lähes mistä tahansa päin ohjelmistoa. Tämän takia on suositeltavaa kapseloida data funktioihin, jolloin nähdään mistä datan muokkaaminen on tapahtunut.

Kahdennettu koodi: Kahdennettu koodi tuo tarpeetonta epäselvyyttä ja vaatii kehittäjiltä enemmän aikaa, koska muutokset täytyy mahdollisesti tehdä useaan kohtaan koodia.

Koodin käyttö: Käyttö ei tapahdu enimmäkseen omassa moduulissa: Esimerkiksi funktio käyttää enemmän aikaa kommunikointiin toisen moduulin funktioiden tai datan kanssa kuin kommunikoi oman moduulin sisällä. Ratkaisuna on siirtää funktio sitä enemmän tarvitsevan moduulin sisään.

Laajeneva muutos: Laajeneva muutos esiintyy, kun moduulia muutetaan erilaisista syistä. Esimerkiksi joka kerta, kun lisätään uusi tietokanta järjestelmään, täytyy tiettyjä funktioita muuttaa. Tietokantalogiikka on erillinen kokonaisuus, joten se täytyy eriyttää koodista erilliseen moduuliin.

Laiska elementti: Elementtejä ohjelmoidessa tulee usein lisättyä rakenne, joka tuo mahdollisuuden muunneltavuuteen tai uudelleenkäyttöön. Jos kuitenkin on huomattavissa, että tämä rakenne on turha, kannattaa se yhdistää osaksi toista elementtiä. Esimerkiksi funktioiden kohdalla yhdistetään funktio toisen sisälle.

Liiallinen kommentointi: Jos tarvitaan kommentti koodilohkon toiminnallisuuden selittämiseen, kannattaa tämä lohko eristää omaksi metodikseen. Jos metodi on jo eristetty, niin sitten metodi tulee uudelleen nimetä kuvaavammin. Yleisenä sääntönä voidaan pitää sitä, että jos on tarvetta kommentoida, kannattaa kokeilla ensin refaktoroida koodia siihen pisteeseen, että kommentointi tuntuu liialliselta.

Liiallinen tiedonvälitys: Ohjelmistotuotannossa pyritään pitämään moduulit selkeästi erillään toisistaan, joten turha tiedonvaihto moduulien välillä tulee pitää minimaalisena.

Muuttuva data: Liiallinen datan muuttaminen voi aiheuttaa yllättäviä ohjelmointivirhetilanteita. Esimerkiksi, jos dataa muutetaan yhdessä paikassa ja toisessa paikassa ohjelmisto odottaa datan arvon pysyneen samana, niin ohjelmisto ei toimi odotetusti. Tässä voidaan ottaa avuksi datan kapselointi, muuttujan jakaminen niin, että muuttujat pysyvät erillään, logiikan eristäminen koodista, joka prosessoi datan päivityksen, ja funktion siirtäminen toiseen paikkaan, jolloin pyritään pitämään koodiosio mahdollisimman erillään kaikesta, mikä päivittää datan.

Oudot nimet: Koodin tulee olla selkeää, jotta sitä on jokaisen helppo ymmärtää. Funktioiden, moduulien, muuttujien ja luokkien nimien tulee olla selkeitä, jotta niistä selviää, mitä ne tekevät.

Pienet vaadittavat muutokset joka puolella ("Shotgun surgery"). Kun tehdään muutos koodiin, täytyy tehdä paljon pieniä muutoksia joka puolelle koodia, jolloin on hankalaa löytää jokainen muutosta tarvitseva kohta.

Pitkä parametrilista: Pitkät parametrilistat voivat olla hankalia käsittää. Tämä ei tarkoita sitä, että kannattaisi suosia globaalia dataa, vaan on suositeltavaa esimerkiksi antaa funktiolle parametrina alkuperäinen datarakenne tai korvata parametri kyselyllä funktion sisällä.

Pitkät funktiot: Mitä pidempi funktio on, sitä epäselvempi se on. Kannattaa tehdä lyhyitä funktioita, jotka nimetään niin, että usein funktion sisälle ei tarvitse edes katsoa, koska nimi kuvaa niin hyvin funktion toimintaa.

Primitiivisten datatyyppien liiallinen käyttö: Joskus on hyödyllistä käyttää omia olioita primitiivisten datatyyppien sijaan. Tällöin voidaan välttää ongelmat, joita esimerkiksi voi esiintyä, jos yritetään lisätä tuumia millimetreihin tai käsitellä puhelinnumeroa stringinä. Omien olioiden käytöllä voidaan helpottaa logiikan käsittelyä, jota esimerkiksi puhelinnumerokin sisältää.

Spekulatiivinen yleistys: Ohjelmistoa kehittäessä saattaa tulla mieleen ominaisuuksia, joita varten olisi hyvä olla reilusti ennakkoon koodi valmiina. Yleensä tällaisen

koodin ylläpitäminen ja ymmärtäminen etukäteen on hankalaa, joten ideasta kannattaa luopua.

Suuri luokka: Suuren luokan huomaa yleensä siitä, että sillä on liian monta kenttää. Kun kenttiä on useita, on kahdennetun koodin riskikin suurempi. Luokan ulkoistamisella pois isosta luokasta omaksi komponentiksi saadaan alkuperäisen luokan kokoa, mukaan lukien kenttien määrää, pienemmäksi. Lisäksi tulee pyrkiä vähentämään tarpeetonta koodia. Jos on isoja, satojen rivien pituisia metodeja, joilla on paljon yhteistä, tulee tätä hyödyntää muuttamalla metodit pienemmiksi ja pyrkiä löytämään yhtäläisyydet ja käyttämään näitä omina yleisinä metodeina.

Tarpeeton data yläluokalta: Jos aliluokka ei tarvitse kuin pientä osaa datasta, jota se saa yläluokalta, on hierarkia luultavasti väärällä tavalla rakennettu. Tässä tilanteessa tulee rakentaa sisarluokka ja viedä kaikki käyttämätön koodi tälle luokalle.

Tilapäinen kenttä: Luokassa voi olla kenttiä, jotka ovat vain tiettyjä tilanteita varten. Näitä on vaikeaa ymmärtää, koska yleensä oletetaan, että näille löytyy olio, joka käyttää kaikkia näitä kenttiä. Tällaiset muuttujat tulisi eristää omaan luokkaansa ja kaikki koodi, joka käyttää näitä kenttiä, laitetaan samaan luokkaan.

Toistorakenteet: Toistorakenteista voi olla välillä hankalaa saada selvää. On suositeltavaa käyttää perinteisen toistorakenteen sijaan ”pipeline”-tyylin koodaamista, jossa koodi on luettavampaa, koska se etenee käyttäjän silmin lineaarisesti.

Toistuvat switch-ehtolauseet: Monimutkainen ehdollinen logiikka, jota switch-ehtolause voi sisältää, on yksi hankalimmista asioista päätellä ohjelmoinnissa. Eri tyyppien ehdollinen logiikka käsitellään erilaisin tavoin kahdennettuna. Kannattaa poistaa yleisen switch-ehtolauseen logiikka, luoda luokat jokaiselle tapaukselle ja käyttää polymorfismia tyyppikohtaisen käytöksen löytämiseksi.

Vaihtelevat luokat, joilla on erilaiset rajapinnat: On turhaa käyttää useita luokkia, jos on mahdollista korvata näitä toisillansa. Tämä vaatii rajapintojen yhtenäistämistä, mikä tapahtuu muuttamalla funktiot samanlaisiksi.

Viestiketjut: Viestiketjussa asiakas voi pyytää oliota, jolta asiakas kysyy toista oliota ja tältä oliolta kolmatta oliota ja näin jatketaan pitkälle. Tällainen navigointi tarkoittaa, että asiakas on kytkeytynyt navigointirakenteeseen ja mikä tahansa muutos välittäjään aiheuttaa asiakkaan muutoksen. Ratkaisuna voidaan selvittää, mihin oliota käytetään, koettaa siirtää tätä oliota käyttävä koodi omaksi funktiokseen pois toisesta funktiosta ja sitten siirtää tämä funktio alaspäin ketjussa. Jos yhden olion useat asiakkaat haluavat navigoida ketjussa koko matkan, lisätään tätä varten metodi.

Välittäjä: Olioiden käytön etuna on kapselointi, jolla saadaan sisäiset yksityiskohdat piilotettua, mutta se tuo mukanaan myös välittämisen. Tämä voi kuitenkin mennä liian pitkälle esimerkiksi silloin, kun useat metodit välittävät viestejä toiseen luokkaan.

5.2 Arkkitehtuuri

Arkkitehtuurin refaktoroinnilla tarkoitetaan ohjelmiston arkkitehtuurin muuttamista ilman, että muutetaan ohjelmiston toiminnallista käyttäytymistä. Arkkitehtuurin refaktorointi sisältää pelkän koodin ja toimintamallin refaktoroinnin lisäksi tietokantojen, suoritusympäristön ja tärkeimpien prosessin kulkujen refaktoroinnin. Näissä tapauksissa arkkitehtuurin refaktorointi kestää usean ylläpidettävän julkaisun ajan. Muuttuvien, ajonaikaisten ympäristöjen ja liiketoimintaodotusten takia refaktorointia voidaan joutua tekemään inkrementaalisesti useiden ohjelmiston julkaisujen välillä. (Kumar & Kumar, 2011.)

Suuria ohjelmistoja rakentaessa pelkästään kooditason refaktorointi ei aina tuo haluttua muutosta, koska ongelmat voivat olla syvemmällä arkkitehtuuritasolla. Arkkitehtuurin refaktoroinnilla voidaan arvioida ohjelmiston perusteita ja motivaationaalisia vaikuttajia, joiden päälle ohjelmisto rakennettiin. Arkkitehtuurin refaktoroinnilla voidaan tasapainottaa järjestelmän laatuominaisuuksia. (Kumar & Kumar, 2011.)

Aikaisemmin on ajateltu, että kun ohjelmiston arkkitehtuuri ja toimintamalli on kertaalleen tehty, sitä ei enää muuteta. Ongelmana tässä ajatustavassa on se, että ohjelmiston vaatimusten kuviteltiin olevan tiedossa jo aikaisessa vaiheessa. Nykyään tilanne on toinen. Refaktoroinnin ansiosta voidaan muuttaa ohjelmiston arkkitehtuuria,

vaikka ohjelmisto olisi ollut tuotannossa useita vuosia. Refaktoroinnin vaikutus arkkitehtuurin kohdalla tulee siitä, kuinka refaktorointia voidaan käyttää hyvän, muutoksiin vastaavan koodimallin muodostamiseen. Arkkitehtuurin refaktoroinnissa käsitellään uudelleen arkkitehtonisia päätöksiä ja valitaan vaihtoehtoinen ratkaisu kuhunkin suunnitteluongelmaan. (Fowler, 2018.)

Arkkitehtuurin refaktorointi voi kohdistua usealle osa-alueelle. Näitä osa-alueita ovat muun muassa ohjelmiston toiminnallisuus, samanaikaisuus/informaatio, käyttöönotto ja toimintakyky. Näitä osa-alueita voidaan käsitellä muokkaamisen, mukauttamisen ja yksinkertaistamisen kautta. Taulukossa 2 on kuvattu tarkemmin osa-alueita, joita arkkitehtuurissa voidaan Zimmermannin (2015) mukaan refaktoroida:

Taulukko 2. Arkkitehtuurin refaktoroinnin näkökulmia (Zimmermann, 2015)

Näkökulmat / osa-alueet	Muokkaaminen	Mukauttaminen	Yksinkertaistaminen
Toiminnallisuus	Jaa komponenttien vastuualueet	Paljasta sisäinen ominaisuus komponentin vastuu-alueena	Yhdistä komponenttien vastuualueet
	Siirrä vastuualue uudelle komponentille	Siirrä vastuualue olemassa olevalle komponentille	Yhdistä komponentit
	Jaa kerrokset (Siirrä komponentit uuteen kerrokseen)	Korvaa kerros	Liitä läheiset kerrokset yhteen

Taulukko 2. Arkkitehtuurin refaktoroinnin näkökulmia (Zimmermann, 2015)

Näkökulmat / osa-alueet	Muokkaaminen	Mukauttaminen	Yksinkertaista-minen
Samanaikaisuus, informaatio	Lajittele proses-sointi samanaikai-suudella	Vaihda lajittelualgo-ritmia	Poista samanai-kaisuus
	Ota käyttöön väli-muisti	Vaihda välimuistin puhdistusavainta	Poista välimuisti
	Täytä ennakkoon välimuistia (lataa ahneemmin)	Vaihda välimuistin puhdistamisstrate-giaa	Aloita tyhjällä välimuistilla (la-taa laiskemmin)
Käyttöönotto	Siirrä/määrää loo-ginen kompo-nentti uuteen käyttöönottoyk-sikköön	Vaihda skaalausstra-tegiaa (esimerkiksi nykyisten resurssien kapasiteetin lisäämi-sestä uusien resurs-sien lisäämiseen)	Yhdistä käyt-töönottoyksiköt
	Jaa käyttöönotto-yksikkö, jotta voi-daan ottaa käyt-töön erillisissä olemassa olevissa tai uusissa sol-muissa	Siirrä käyttöönotto-yksikkö yhdeltä ser-verisolmulta toiselle	Vahvista sol-muja
Toimintakyky	Koosta solmu uu-deksi tasoksi	Jaa taso	Pura tasot
	Ota käyttöön ryh-mittäminen	Vaihda kuormanta-saamista ja varajär-jestelmään vaihta-mispolitiikkaa	Poista ryhmittä-minen

Arkkitehtuurin refaktoroinnissa on tarkoitus parantaa ohjelmiston laatua vaikuttamalla ohjelmiston ylläpidettävyyteen, yksinkertaisuuteen, vakauteen ja suorituskykyyn (Kumar & Kumar, 2011).

Taulukossa 3 on kuvattu, millä tasolla refaktorointia tehdään liittyen erilaisiin laatuominaisuuksiin ja kauanko refaktoroinnin voi olettaa vievän aikaa erilaisissa tilanteissa.

Taulukko 3. Laatuominaisuuksien refaktorointi ja ajankäyttö (Kumar & Kumar, 2011)

Laatuominaisuus	Refaktorointitaso	Ajankäyttö
Vakaus	1. Käyttöönottotaso 2. Sovellustaso	Lyhyt ja keskipitkä ajanjakso
Vasteaika	1. Sovellustaso	Lyhyt ajanjakso
Saatavuus	1. Tietokantataso 2. Käyttöönottotaso	Keskipitkä ja pitkä ajanjakso
Suoritusteho/läpivienti	1. Sovellustaso 2. Tietokantataso	Keskipitkä ajanjakso
Tarpeettomuus	1. Päästä päähän prosessitaso	Pitkä ajanjakso
Päällekkyys	1. Päästä päähän prosessitaso	Pitkä ajanjakso
Monimutkaisuus	1. Kaikki tasot	Keskipitkä ja pitkä ajanjakso

Arkkitehtuurista voi löytyä useita asioita, joiden takia refaktorointia kannattaa harkita (Stal, 2013):

Arkkitehtuurin keskittyneisyys: Ohjelmistoinsinöörit voivat suosia liikaa keskittyneitä menettelytapoja, vaikka hajautetummat toimintatavat olisivat parempia. Keskittyneissä arkkitehtuureissa skaalautuminen on hankalampaa.

Epäselvät entiteettien roolit: Komponenttien tulisi olla mahdollisimman kuvaavia, jotta niiden vastuualueet ovat selvät ja sidosryhmät voivat helposti ymmärtää suunnittelua. Jokaisella komponentilla tulisi olla vain yksi vastuualue.

Epäsuorat riippuvuudet: Ohjelmistokehittäjät voivat suunnata pois halutusta ja toteutetusta arkkitehtuurista, jos he lisäävät epäsuoria riippuvaisuuksia toteutukseen. Esimerkiksi globaalien muuttujien jatkuva käyttö aiheuttaa tällaista epäsuoraa riippuvuutta ohjelmistoon. Toinen esimerkki on arkkitehtuurin korkeampien kerroksien hajottaminen, vaikka arkkitehtuuri edellyttäisi tiukkaa kerrostamiskäytäntöä.

Epäsymmetrinen rakenne tai käytös: Arkkitehtuurilla on olemassa kahdenlaista symmetriaa: rakennesymmetria ja käytössymmetria. Rakenteen symmetria osoittaa, että identtiset ongelmat ratkaistaan aina identtisillä malleilla. Symmetria usein kertoo, että arkkitehtuurin taso on laadukas, mutta joissain tapauksissa epäsymmetria voi olla suotava tai jopa pakollinen asia. Esimerkiksi erilaisten komponenttien täytyy hoitaa resurssien varaaminen ja vapauttaminen. Käytössymmetria käsittelee toiminnallista puolta, kuten toimintojen aloittamista. Tällaisia toiminnallisia tilanteita ovat esimerkiksi transaktiot, jotka tarvitsevat myös tapahtuman vahvistamisen tai palauttamisen, tai haarautuminen, joka tarvitsee myös liittämisen, tai avoin metodi, joka tarvitsee myös sulkemisen.

Itsekehitetyt arkkitehtuuriratkaisut parhaiden käytäntöjen sijaan: Ohjelmistoinsinöörien kannattaa ottaa selvää, löytyykö arkkitehtuuritasolla jo valmiita ratkaisuja arkkitehtuuriongelmiin, mutta on myös tilanteita, jolloin joudutaan itse miettimään omia ratkaisuja. Tällaisia tilanteita voivat aiheuttaa esimerkiksi valmiit mallit, jotka eivät tue reaaliaikaisia ympäristöjä johtuen mallien liiasta dynaamisuudesta.

Liian geneerinen suunnittelumalli: Geneeristen mallien käyttö voi auttaa muuttuvien asioiden mukauttamisessa myöhemmissä vaiheissa kehittämistä. Liika geneerisyys voi kuitenkin luoda ohjelmistosta hankalan muunnella, kehittää ja ylläpitää, jos geneerisyyttä on lisätty paikkoihin, joissa pärjättäisiin ilmankin tai joissa geneerisyys ei välttämättä ole lainkaan hyvä ratkaisu. Arkkitehtuurimallin tulisi olla niin spesifinen kuin mahdollista ja geneerisyyttä ja muunneltavuutta tulisi käyttää vain tarpeen mukaan.

Monimutkainen tai ilmeetön arkkitehtuuri: Ohjelmistoista voi tulla monimutkaisia ja ilmeettömiä, jos arkkitehtuurin entiteettien nimet ovat epäselviä, entiteeteillä on liiallisia komponentteja tai riippuvaisuuksia tai arkkitehtuuri on liian pienissä osissa, jolloin kokonaisarkkitehtuuria on vaikea ymmärtää. Ohjelmistoarkkitehtuurin tulisi aina olla mahdollisimman helposti ymmärrettävällä tasolla.

Riippuvuus useiden komponenttien välillä: Riippuvuudet voivat aiheuttaa vaikeuksia testaamisen, muokattavuuden tai vaikuttavuuden saralla. Arkkitehdillä voi esimerkiksi olla vaikeuksia muokata tai testata komponenttia yksinkertaisesti, koska tämä komponentti on yhteydessä moneen muuhun komponenttiin.

Suunnittelumallin rikkomiset: Esimerkiksi rennon kerrostamisen käyttäminen voi aiheuttaa hallitsemattomalla tasolla toistuvia ongelmia ratkaisuissa, mikä vaikuttaa negatiivisesti näkyvyyteen tai vaikuttavuuteen.

Tarpeettomat riippuvuudet: Riippuvuuksien määrän tulisi olla aina mahdollisimman vähäinen, jotta ohjelmistossa ei olisi paljoa monimutkaisuutta. Kaikki ylimääräiset riippuvuudet voivat vaikuttaa muokattavuuteen ja suorituskykyyn.

Toistuvat suunnitteluvirhehavainnot: On vaikeaa tiedostaa, miten iso määrä toistoa on hyväksyttävää ja milloin on kyse arkkitehtuuriongelmasta. Ohjelmistokehityksessä yleisten toimintojen pitäisi olla modulaarisia.

Vaillinainen toiminnallisuuden jakaminen: Vastuualueiden vaillinainen kartoittaminen alijärjestelmiin lisää sattumanvaraista monimutkaisuutta. Alijärjestelmien rakenneosien tulisi vaikuttaa yhtenäisiltä, mutta rakenneosien välisen liitännäisyyden tulisi olla vähäistä. Ohjelmistoinsinöörit voivat hyödyntää metriikkaa yhtenäisyyden ja

liitännäisyyden löytämiseksi. Erittäin suuri tai erittäin pieni alijärjestelmien määrä kertoo yleensä siitä, että toiminnallisuuden jakaminen on vaillinaista.

Nämä syyt eivät kuitenkaan välttämättä ole todisteita arkkitehtonisista ongelmista. Kyseiset esimerkit ovat vain indikaattoreita siitä, että tämänkaltaisia ongelmia saattaa esiintyä arkkitehtuurissa. (Stal, 2013.)

5.3 Ohjelmointirajapinnat

Ohjelmointirajapinnat (API = Application Programming Interface) ovat liitoksia, jotka liittävät moduulit ja niiden funktiot toisiinsa. Hyvän ohjelmointirajapinnan tunnistaa siitä, että se erottaa selvästi dataa päivittävät funktiot dataa pelkästään lukevista funktioista. Tietorakenteita puretaan usein turhaan, kun niitä välitetään funktioiden välillä. (Fowler, 2018.)

Ohjelmointirajapintojen asiakkaiden tulee siirtyä käyttämään uutta rajapintaa, jos ohjelmointirajapintojen refaktorointia tehdään. Kun useissa järjestelmissä käytettävää luokkakirjastoa refaktoroidaan, kaikkien järjestelmien tulee muuttua. Ongelmana tässä on se, että kirjastoa kehittävät ohjelmistokehittäjät eivät tiedä kaikkia niitä järjestelmiä, jotka käyttävät kyseistä kirjastoa. (Dig & Johnson, 2005.) Dig & Johnson (2005) huomasivat tutkimuksessaan, että yli 80 prosenttia ohjelmointirajapinnan rikkovista muutoksista ovat refaktorointeja eli rakenteellisia muutoksia, jotka eivät vaikuta toiminnallisuuteen.

Kim et al.:in (2011) tutkimuksessa tutkittiin kolmea isoa avoimen rajapinnan projektia ohjelmointirajapintojen refaktoroinnin suhteen ja huomattiin, että ohjelmointirajapintojen refaktorointi ei aina ole hyödyllistä. Ohjelmointirajapintojen refaktorointi voi aiheuttaa enemmän ohjelmointivirhekorjauksia kuin mitä niitä oli aikaisemmin. Ohjelmointivirheiden korjausaste nousi 26,1 ja 30,3 prosentin välille. Tämä johtui usein refaktoroinnin tekemisestä yhtä aikaa toiminnallisten muutosten tekemisen kanssa. Kuitenkin ohjelmointirajapintojen refaktoroinnin jälkeen ohjelmointivirheiden korjaaminen nopeutui huomattavasti, ja korjausnopeuden parannus oli 35,0 ja 63,1 prosentin

väliltä. Tutkimuksessa (Kim et al., 2011) huomattiin myös, että ohjelmointirajapintojen refaktorointi saattoi tuoda esille myös uusia ohjelmointivirheitä.

Ohjelmointirajapintojen refaktorointi voi siis olla ongelmallista, mutta se on myös välttämätöntä, jos halutaan säästää tulevaisuudessa ylläpitokustannuksissa ja välttää ohjelmiston versioiden lisääntyminen (Dig & Johnson, 2005). Tämän takia olisi tarve ohjelmistomallinnustyökalulle, joka havaitsee ja korjaa epäjohdonmukaiset ohjelmistopäivitykset refaktorointia tehdessä. Jezekin & Dietrichin (2017) tutkimuksessa perehdyttiin ohjelmointirajapintojen evoluutioon ja yhteensopivuuteen ja huomattiin, että ohjelmointirajapintojen evoluution analysointiin löytyy vain harvoja työkaluja. Nämä työkalut ovat usein pienien yhteisöjen ylläpitämiä ja epäonnistuvat havaitsemaan joitain ohjelmointirajapintojen yhteensopimattomuuksia, mutta ovat kaiken kaikkiaan käyttökelpoisia.

Parhaiten työkaluista menestyi SigTest (Signature Test Tool), joka on avoimen lähdekoodin työkalukokoelma, jolla voidaan vertailla ohjelmointirajapintoja ja mitata ohjelmointirajapintojen testien kattavuutta. (Jezek & Dietrich 2017). SigTest raportoi, kun uusi jäsen lisätään, varmistaa, että kaikki jäsenet ovat läsnä, ja tarkistaa jokaisen ohjelmointirajapintajäsenelle määritellyn käyttäytymisen samalla, kun se vertaa saman ohjelmointirajapinnan erillisiä toteutuksia. SigTest myös tarkistaa, voiko vanhan ohjelmointirajapintaversion korvata uudemmalla ilman haitallisia vaikutuksia olemassa oleviin ohjelmointirajapinnan asiakkaisiin. Ohjelmointirajapintojen testien kattavuuden SigTest arvioi analysoimalla, kuinka moneen julkiseen luokkajäseneen testijoukko viittaa ohjelmointirajapintamäärittelyssä. (OpenJDKWiki, 2017.)

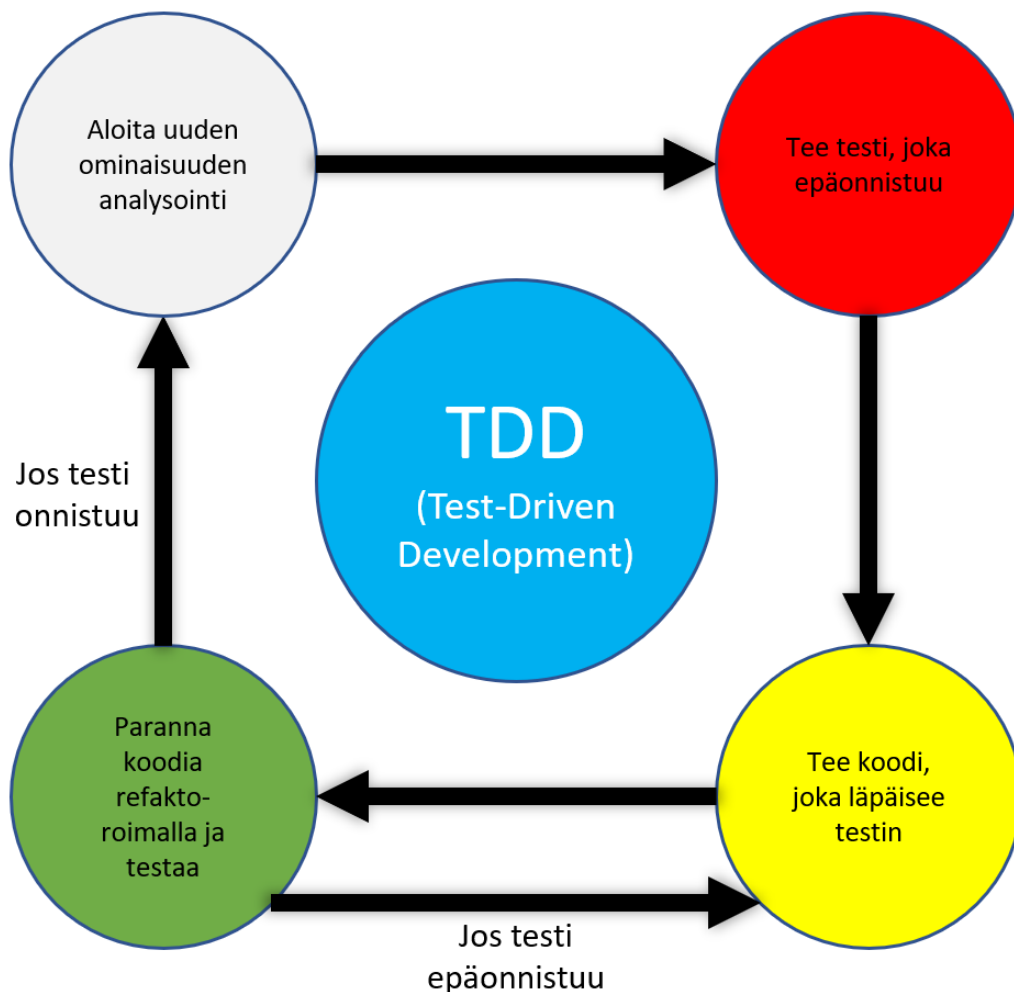
5.4 Testaaminen

Jotta refaktorointia voidaan tehdä, tulee testijoukon olla kattava, jotta se löytää välttämättömät virheet. Refaktorointia tehdään kooditasolla jatkuvasti, joten on tärkeää, että koodatessa mahdollisesti syntyneet virheet saadaan nopeasti kiinni. Suuri osa ohjelmoijien ajasta menee ohjelmointivirheiden etsimiseen ja korjaamiseen, joten auto-

maattisten testien käyttäminen voi auttaa huomattavasti ohjelmointivirheiden karsimisessa ja ylipäänsä ajan säästämisessä. (Fowler, 2018.) Testauksen automatisoinnilla voidaan turvata manuaalista testaamista parempi koodin kelvollisuus, kun tehdään refaktorointia (Leppänen et al., 2015).

Kun on tarve lisätä uusi ominaisuus ohjelmistoon, on hyödyllistä aloittaa testin tekemisestä kyseiselle ominaisuudelle. Tällä tavoin ohjelmistokehittäjä joutuu miettimään tarkkaan, mitä täytyy tehdä funktion lisäämiseksi, ja hänen täytyy keskittyä rajapintaan enemmän kuin toteutukseen. Näin ollen ohjelmistokehittäjällä on myös tarkka käsitys siitä, missä vaiheessa koodi on valmis eli testi on mennyt läpi. (Fowler, 2018.) Testeillä ja esimerkeillä voidaan helpottaa vaatimusten ja liiketoimintasääntöjen havainnollistamista, kunhan testit ja esimerkit ovat ymmärrettäviä sekä liiketoiminta- että teknisille tiimeille (Crispin, 2006).

Testin kirjoittaminen ensimmäiseksi on myös hyvä käytäntö, kun löydetään uusi ohjelmointivirhe. Ohjelmistokehittäjän tulisi ensiksi kirjoittaa yksikkötesti, joka tuo esille ohjelmointivirheen (Fowler, 2018). Testivetoisessa kehittämisessä (Test-Driven Development (TDD)) on tarkoitus kirjoittaa lyhyellä syklillä testi ja koodi, joka saa testin toimimaan, sekä suorittaa refaktorointi siten, että lopputulos on mahdollisimman hyvä. Testin, koodin ja refaktoroinnin muodostama sykli tulisi tapahtua useita kertoja tunnissa. (Fowler, 2018.) Kuvassa 5 on kuvattu, kuinka TDD:n sykli etenee.



Kuva 5. Test-Driven Development vaiheineen (Fowler, 2018)

Napaggan et. al.:in (2008) tutkimuksessa käytiin läpi TDD:n hyötyjä laadun parantamisen suhteen. Tutkimuksessa oli mukana IBM-, Microsoft Windows-, Microsoft MSN- ja Microsoft Visual Studio -tiimit. Tutkimuksessa todettiin, että TDD:etä voidaan käyttää useilla erilaisilla toimialueilla ja sen avulla voidaan vähentää huomattavasti ohjelmiston virheiden määrää. Tämä saavutettiin ilman, että tuottavuus väheni merkittävissä määrin kehitystiimeissä. (Napaggan et al., 2008.)

Napaggan et al. (2008) suosittelevat tutkimustulostensa pohjalta aloittamaan TDD:n käytön heti projektin alusta lähtien, koska muuten tehtävän työn määrä on todella suuri. Tiimi, jolle TDD on uusi asia, tulisi tutustuttaa automaattiseen koontiversion testi-integraatioon, kun kehittämistyö on noin puolivälissä. Tällöin tiimi ehtii ensin perehtyä TDD:hen. Myös testitiimi tulee tutustuttaa TDD:n käytäntöön.

Kehitystiimiä tulisi rohkaista tekemään uusia testejä aina, kun uusi ongelma on löydetty, jotta yksikkötestijoukot paranisivat, ja alustavat yksikkötestisuunnitelmat tulisi toteuttaa mahdollisimman laadukkaasti, jolloin koodi katetaan mahdollisimman hyvin. Yksikkötestien tulee olla nopeasti suoritettavia, koska testejä pitäisi ajaa jatkuvasti, jotta järjestelmän tila on aina tiedossa. Yksikkötestejä on myös jaettava kehittäjien kesken, jotta huomataan mahdolliset integraatio-ongelmat ajoissa. (Napaggan et al., 2008.)

Testien tilanteesta tulee olla myös selvillä, jotta voidaan selvittää ongelmia. On suositeltavaa laskea testien määrä, testien kattavuus koodin osalta, löydettyjen ja korjattujen ohjelmointivirheiden määrä, lähdekoodin määrä, testikoodin määrä ja ajankäyttö. (Napaggan et al., 2008.)

Jotta tiedetään, mitä mieltä tiimi on TDD:n käytöstä osana ohjelmistokehittämistä, tulee tiimin moraalitasoa arvioida ennen ja jälkeen projektien. Kyselyjen tulee kohdistua yleisesti TDD-prosessiin ja kehittäjien halukkuuteen jatkaa TDD:n käyttöä osana ohjelmiston kehittämistä. (Napaggan et al., 2008.)

6 REFAKTOROINNIN ONNISTUMINEN JA RISKIT

Refaktorointi ei ole helppoa ja riskitöntä, vaan siihen voi sisältyä useita erilaisia asioita, jotka tulee ottaa huomioon refaktorointia tehdessä. Luvussa 6.1 käydään läpi refaktoroinnin onnistumista puoltavia tekijöitä ja luvussa 6.2 esitellään refaktoroinnin mukanaan tuomia riskejä.

6.1 Refaktoroinnin onnistuminen

Leppänen et al.:in (2015) tutkimuksessa haastatellut ohjelmistoarkkitehdit ja -kehittäjät oppivat refaktorointien aikana seuraavien toimintatapojen ja tekniikoiden olevan hyödyllisiä päivittäisessä työssä:

1. Konsultoi ohjelmoijia refaktorointitarpeiden mittaamista varten, koska he ovat tekemisisissä koodin kanssa jatkuvasti ja näin ollen tietävät sen hyvät ja huonot osat.
2. Pienten tiimien kannattaa antaa kommunikoida toistensa kanssa, jotta refaktorointi onnistuisi ilman suurempia ongelmia. Tällä estetään ongelmat, joita voi seurata esimerkiksi, kun käsitellään komponentteja, joita myös toinen ohjelmistokehitystiimi tarvitsee.
3. Toimiva testaaminen ja järkevä versionhallinta ovat oleellisia tekijöitä, jotta refaktorointi onnistuu hyvin.
4. Tarkista ja päivitä yksikkötestit ennen refaktorointia, koska refaktorointi usein mitätöi yksikkötestit.
5. Jotta saadaan selville refaktorointiaskeleet, tulee refaktorointioperaatiot kirjata versiohallintaan. Tällöin pysytään perillä kaikista refaktorointiaskeleista.
6. Varaa erillinen refaktorointipäivä joka viikolle. Optimaaliset ratkaisut lopullista tuotetta varten eivät yleensä ole heti tiedossa, etenkään aikaisessa kehittämisvaiheessa. Käytä ensimmäistä järkevää ratkaisua, älä jää miettimään optimaalista ratkaisua. Koodi käydään läpi, siistitään ja refaktoroidaan refaktorointipäivänä.

7. Tee pieniä refaktorointeja, kun lisäät ominaisuuksia tai korjaat ohjelmointivirheitä. Tällä tavoin koodipohjan laatu pysyy parempana.
8. Sisällytä pienet refaktorointitehtävät tehtävien arviointeihin.
9. Lisättäessä uutta ominaisuutta ohjelmistoon kannattaa ensiksi refaktoroida koodi ja sen jälkeen vasta lisätä uusi ominaisuus osaksi ohjelmistoa.
10. Joskus on kannattavampaa poistaa ja tehdä uusiksi refaktoroitava osa kuin korjata sitä. Tämä vaatii tueksi luotettavan testijoukon, jotta voidaan varmistaa, että toiminnallisuus vastaa aikaisempaa toteutusta.
11. Käytä koodikatselmointeja jokaista suurta refaktorointia varten.
12. On tärkeää ajoittaa refaktorointi oikein. Refaktorointia ei esimerkiksi tulisi tehdä lähellä julkaisupäivää. Jos käytössä on jatkuva käyttöönotto, niin älä tee laajoja refaktorointeja juuri ennen viikonloppua, jotta viikonloppuna ei tarvitse tehdä hätäohjelmointivirhekorjauksia.
13. Ota asiakasorganisaatiosta mukaan teknologiasuuntautuneita ihmisiä, jotta refaktorointitarpeista keskusteleminen olisi helpompaa.

Mancl (2001) puolestaan toteaa legacy-järjestelmän refaktoroinnissa tärkeimmiksi asioiksi seuraavat:

1. Suunnittele havaintotyön ajankäyttö ja luo korkean tason käyttötapauksia ja oliomalleja. Ajankäyttö on todella tärkeä piirre legacy-järjestelmän uudistamisessa, koska olemassa olevan järjestelmän arkkitehtoniset asiat eivät selviä vain koodia katsomalla, vaan vaativat käyttötapausten mallintamista ja korkean tason kaavioita selventämään riittämätöntä legacy-dokumentaatiota.
2. Käytä standardisuunnittelumalleja uudelleensuunnittelutyössä ja etsi mahdollisuuksia refaktoroida olemassa olevaa koodia.
3. Muokkaa koodia pienin askelin ja testaa uudelleen useasti muokkaamisen yhteydessä.

Kumar & Kumar (2011) toteavat, että arkkitehtuuria refaktoroidessa on tärkeää käyttää päivittäin metriikkaa koskien järjestelmän resurssien käyttöasteen yksityiskohtia, sovelluksen kriittisten parannusten merkkipaaluja ja suorituskyvyn/läpiviennin yksityis-

kohtia liittyen loogisen toiminnallisuuden yksiköihin. Kumar & Kumar (2011) kehottavat myös analysoimaan kattavasti organisaatioyksikön sovellusalueen, jotta ei rakenneta sovelluksia, joilla on useita tarpeettomia liiketoimintapalveluita. On siis tärkeää ottaa huomioon järjestelmässä olevat muut sovellukset ennen kuin aletaan rakentamaan laajoja sovelluksia. Lisäksi Kumar & Kumar (2011) sanovat, että on tärkeää ottaa huomioon erilaiset työmäärät arkkitehtuurissa: Esimerkiksi ei kannata käyttää yleistä arkkitehtuuria monimuotoiselle, erilaisia työmääriä sisältävälle työlle, koska tällöin järjestelmän suorituskykyominaisuudet voivat kärsiä.

6.2 Refaktoroinnin riskit

Fowlerin (2018) mukaan koodipuolen refaktorointi on riskialtista, jos sitä ei tehdä oikein. Pahimmillaan aikataulut voivat heittää jopa viikkoja, etenkin jos refaktorointia lähdetään tekemään miettimättä sitä ensin tarkasti. Miettimättä tekeminen johtaa siihen, että koodia refaktoroidessa huomataan jatkuvasti uusia refaktorointia tarvitsevia kohteita ja tämä kierre jatkuu. Refaktorointi voi aiheuttaa toimivaan koodiin muutoksia, jotka voivat nostaa esille ovelia ohjelmointivirheitä. Näiden asioiden takia refaktorointi tulee tehdä systemaattisesti, jotta saadaan selville kaikki osa-alueet, jotka vaativat refaktorointia.

Arkkitehtuuripuolen refaktoroinnissa on myös erittäin tärkeää muistaa, että käytössä on systemaattinen prosessi, jota hyödynnetään arkkitehtuurin parantamisessa. Refaktorointimalli voi olla vaillinainen, jos se ei esimerkiksi ota huomioon arkkitehtuurin kannalta oleellisia ominaisuuksia, kuten ohjelmiston reaaliajassa toimivien osien asettamia vaatimuksia. (Stal, 2013)

Leppänen et al. (2015) haastattelema arkkitehti totesi hankalaksi muokata metodien julistuksia ja rajapintoja, joista ulkoiset palvelut ovat riippuvaisia. Tällaisen koodin refaktorointi voi rikkoa ulkoisen toiminnan ja aiheuttaa ongelmia ulkoiselle toimijalle. Useat muut haastateltavat tiedostivat myös, että refaktoroinnin onnistuminen ei aina ollut selvä asia, koska koodin rakenteet eivät välttämättä parantuneet laisinkaan tai

aikaansaatu sisäinen koodin laatu saattoi jopa huonontua. Yksi haastatelluista arkkitehdeistä totesi, että jopa kuukausien ajan tehty refaktorointi ei välttämättä korjannut teknologista velkaa ja lopputulemana mitään hyödyllistä ei tullut korjattua. Yleisinä riskeinä haastateltavat kokivat toimivien asioiden rikkomisen, ulkoisesti havaittavien muutosten syntymisen, koodin laadun huonontamisen ja ajan ja resurssien tuhlaamisen. (Leppänen et al., 2015.) Manclin (2001) mukaan refaktoroinnilla ei onnistuta tekemään järjestelmästä helpommin ymmärrettävää, jos refaktorointitiimi ei ymmärrä hyvin nykyisen järjestelmän rakennetta. Tämän takia refaktorointia tekevän tiimin tulisi tutkia järjestelmän koodia ja rakennetta tarkasti ennen muutosten tekemistä legacy-koodiin.

7 TUTKIMUSMENETELMÄT

Idea tutkimuksen tekemiseen tuli Tiedolta. Tiedolla halutaan tietää tarkemmin, miten voidaan päätellä milloin refaktorointi on kannattavaa, koska refaktorointi ja muu uudistamistyö vie paljon resursseja. Tähän tutkimusongelmaan pyrittiin vastaamaan selvittämällä, mitkä ovat refaktoroinnin tärkeimmät tavoitteet ja milloin refaktorointi on järkevä vaihtoehto rakenteelliselle uudistamiselle verrattuna uudelleenmallintamiseen ja uudelleenkirjoittamiseen ja minkälaisen uudistamisen ohella refaktorointia voidaan käyttää. Tästä seurasi idea tehdä tutkimus refaktoroinnista koskien Tiedon laskutusjärjestelmää.

Tutkimusmenetelmänä käytettiin haastattelututkimusta, jossa tiedon kerääminen laskutusjärjestelmän uudistamiseen liittyen tapahtui tapaamisten ja avointen haastattelujen pohjalta. Kesäkuussa 2019 käytiin läpi tutkielman yleinen sisältö ja mitä Tiedolla haluttiin saada selville laskutusjärjestelmän refaktorointitarpeista tutkielman avulla. Alustavaan tapaamiseen osallistui Tiedon johtotason työntekijä, tuotetuen työntekijöitä, järjestelmäarkkitehtejä sekä ohjelmistokehittäjiä.

Kesä-, heinä- ja elokuu menivät pitkälti kirjallisuusosion tekemiseen. Kirjallisuusosio loi pohjaa Tiedon laskutusjärjestelmän uudistamisen käsittelylle muun muassa vastaamalla siihen, mitä refaktoroinnilla käytännössä tarkoitetaan ja milloin on suotavaa käyttää hyödyksi jotain muuta uudistamistapaa kuin refaktorointia. Elo-, syys- ja lokakuun aikana kokoonnuttiin noin 2 kertaa kuukaudessa tunnin kestäviin tapaamisiin laskutusjärjestelmän pääarkkitehdin ja kokeneen ohjelmistokehittäjän kanssa. Näissä tapaamisissa käytiin läpi mitä asioita laskutusjärjestelmän uudistamisessa tulisi painottaa, mitä vaatimuksia uudistamistyö asettaa, mitkä ovat uudistamista vaativia alueita, mitä onnistunut refaktorointi vaatii ja mitä riskejä se sisältää, miten uudistaminen vaikuttaa bisnekseen ja mitä resursseja uudistamista varten tarvitaan.

Tapaamisten lisäksi haastattelin elo-, syys- ja lokakuun aikana yleisesti laskutusjärjestelmän ohjelmistokehittäjiä sekä kysyin heidän näkemyksiänsä nykyisen laskutusjärjestelmän tilasta. Haastatteluissa käytiin läpi yleisesti laskutusjärjestelmän historiaa,

puhuttiin laskutusjärjestelmän nykyisestä yleistilasta sekä käytiin läpi, mitkä ovat laskutusjärjestelmän alueita, joita ohjelmistokehittäjien näkemyksien mukaan olisi hyvä uudistaa.

Näiden tapaamisten, haastattelujen ja näkemysten pohjalta keräsin ylös tarvittavaa tietoa Tiedon laskutusjärjestelmän uudistamista koskevan luvun käsittelyyn. Lopullinen materiaalin käyttö ja asioiden painotus valikoitui sen mukaan, miten tärkeäksi laskutustiimin jäsenet kokivat minkäkin esille nousseen asian.

Luvussa 8 käydään läpi laskutusjärjestelmän historiaa (luku 8.1), laskutusjärjestelmän yleiskatsaus (luku 8.2), uudistamistapoja yleisellä tasolla (luku 8.3), refaktoroidun järjestelmän vaatimukset (luku 8.4) ja järjestelmän suositellut uudistamisalueet (luku 8.5). Lisäksi lopuksi pohditaan, mitä onnistunut refaktorointi vaatii ja mitä riskejä refaktorointi sisältää (luku 8.6) sekä käsitellään uudistamisen vaikutusta bisnekseen ja käytettäviin resursseihin (luku 8.7).

8 TIEDON LASKUTUSJÄRJESTELMÄN UUDISTAMINEN

8.1 Järjestelmän historiaa

Tiedon laskutusjärjestelmällä on pitkä historia. Taustalla on jo vuonna 2011 alkanut uudistaminen, joka valmistui 2016-2017, ja tästä versiosta syntyi nykyinen ratkaisu. Pohjalogiikkaa ja tietokantarakenteita on otettu Nykylaskutuksesta nykyiseen Effica-laskutukseen. Nykylaskutuksen toimintoja on jätetty pois.

Laskutukseen tuli oma tietokanta vuonna 2011. Haluttiin, että laskutus pystyttäisiin tekemään omassa tietokannassaan erillisenä, vaikka olikin tarkoitettu, että laskutus on linkitetty Efficaan. Kaikki suomenkieliset taulut, joita oli Efficassa, ovat vieläkin käytössä, koska ne ovat valmiiksi hinnoiteltuja ja menevät läpilaskutettavana.

Effican haasteena oli tietokannat, joiden päälle lähdettiin rakentamaan uudistamista. Alkuun yritettiin päivittää laskutuskantaa ja tauluja triggereillä siten, että päivitys tapahtuisi, kun Efficassa tapahtui muutos, mutta tämä osoittautui todella haastavaksi. Tämä vaikutti laskutuksen suunnitteluun, sillä piti tehdä ratkaisuja, jotka pystyttiin itse toteuttamaan.

Nykyisessä järjestelmässä on osalla näytöistä UI-tasolla bisneslogiikkaa ja bisneskerroksen kautta mennään datasettiin. Näitä on tarkoitus selventää ja saada korjattua. Ei ole ollut yleistä arkkitehtuuria, vaan käytettävissä oli tiimin arkkitehti, joka loi säännöt, mutta näihin tuli muutoksia kesken tekemisen. Nykyään tilanne on parempi, kun löytyy myös ylemmän tason arkkitehtuuria ja arkkitehdit käyvät keskenään paljon tarkempaa keskustelua siitä, miten asioita tulisi tehdä. Nykyään OPOt (Operative Product Owner) ja APOt (Area Product Owner) juttelevat enemmän ja OPO tuntee järjestelmän hyvin, mikä edesauttaa tiimien välistä kommunikaatiota.

Koodin suhteen on huomattavissa parannettavia asioita. Esimerkiksi ajan mittaan yhtä metodia on muokattu hiljalleen enemmän ja enemmän, jolloin nämä muutaman rivin

muutokset ovat yhdessä paisuneet ja tuottaneet liian pitkiä ja vaikeasti ymmärrettäviä metodeja. Näiden metodien refaktorointi voi viedä aikaa ja vaatii paljon testaamista, jotta toiminnallisuus säilytetään oikeanlaisena.

Tavoitteena on käyttää mahdollisimman yhdenmukaista järjestelmää. Haasteena on se, että jokainen asiakas haluaisi räätälöidyn version, vaikka versiot olisivat näennäisesti samanlaisia. Tämä johtuu siitä, että useilla asiakkailla on erilaisia sääntöjä, joita järjestelmän tulee noudattaa. Tulisi jo alussa tietää, mihin kaikkeen uuden järjestelmän tulisi pystyä, jotta tällaisilta tilanteilta vältyttäisiin. Ei ole helppoa pitää yllä yhtä aikaa vanhaa ja uutta järjestelmää. Haasteita tuo se, että vanhan hinnoittelun tulee toimia vielä pitkään uuden hinnoittelun ohessa. Voidaan esimerkiksi joutua tekemään väliluokkia, jotka hoitavat tämän kompromissin.

8.2 Nykyisen järjestelmän yleiskatsaus

Nykyinen järjestelmä sisältää alueita, joita voitaisiin parantaa refaktoroinnilla. Järjestelmän ylläpidettävyys ei ole nykyisellään helpoimmasta päästä. Tiedon toistuminen koodissa ja saman tiedon tarvitseminen erilaisissa toimintaympäristöissä on yksi parannettava alue. Arkkitehtuuritasolla on riippuvuuksia järjestelmien välillä, mikä vaikeuttaa kehittämistä. Tiedon käsittelylle tulisi löytää yhdistämistapoja. Tietokantamalli on osittain monimutkainen ja sisältää toistuvaa dataa ja datan synkronointia tietokantojen välillä.

Moni ohjelmistokehittäjä ohjelmoi eri tavalla, mistä seuraa liian paljon toisistaan eriävän tyylistä koodia. Koodin parempaan yhtenäistämiseen olisi hyvä pyrkiä panostamaan ennen kuin uudistamista aletaan tehdä. Pohjia on ollut ennenkin olemassa koodaamista varten, mutta ohjelmointitapojen yhtenäistämisessä on vielä parannettavaa.

Ylläpidettävyyden ja laadun parantaminen sekä uusien järjestelmälle asetettujen vaatimusten täyttäminen vaatisi luvussa 5.1 (Koodi) esitettyjen kohtien läpikäymistä ja korjaamista. Esimerkiksi luokkien ja funktioiden koot sekä niiden sisältö ovat parannettavia asioita, joihin ohjelmistossa törmää usein. Funktiot saattavat toteuttaa asioita, joita voitaisiin toteuttaa myös funktion ulkopuolella erillisessä funktiossaan, jolloin

koodin luettavuus selkeytyisi ja funktioiden nimet vastaisivat paremmin sitä, mitä ne oikeasti sisältävät ja tekevät.

Laskutusjärjestelmän käytettävyyttä voitaisiin parantaa muokkaamalla nykyisen laskutusjärjestelmän ulkonäkö vastamaan paremmin Lifecaren yleistä ulkonäköä. Laskutusjärjestelmässä on myös käyttöliittymälogiikkaa, jota voitaisiin hieman selkeyttää.

8.3 Järjestelmän uudistamistavat yleisellä tasolla

Järjestelmän refaktorointi on vain yksi tapa uudistaa järjestelmää ja sitä voidaan hyödyntää osana muita järjestelmän uudistamistapoja. Legacy-järjestelmän uudelleenkirjoittaminen kokonaan vaatii mielellään kaupallisesti tarjolla olevia, osittain valmiita järjestelmiä, joiden päälle voidaan rakentaa uutta ja lisätä valmiita ohjelmiston komponentteja hyödynnettäväksi. Järjestelmän uudistaminen kokonaan ei ole järkevää, jos legacy-järjestelmä sisältää kriittistä bisneslogiikkaa ja on olemassa riski, että osa tästä koodista korvataan uudella koodilla. Riippuen legacy-järjestelmän komponenteista voidaan päättää mitä osia legacy-järjestelmästä otetaan käyttöön uuteen järjestelmään. (Jain & Chana, 2015.) Tiedon laskutusjärjestelmä sisältää paljon koodia, jota varten ei löydy hyödynnettävää kaupallista koodia, jonka pohjalle voitaisiin tehdä uusi laskutusjärjestelmä.

Järjestelmän uudelleenmallintaminen sisältää kaksi erilaista lähestymistapaa. Järjestelmä voidaan joko mallintaa kerralla tai inkrementaalisesti osissa. Inkrementaalisesti uudistaessa on tarkoituksena, että uuteen järjestelmään tulee osaksi sekä legacy-komponentteja että uudelleenmallinnettuja moduuleita ja näiden moduulien välille tulee luoda rajapinnat, joiden kautta kommunikaatio moduulien välillä tapahtuu. Pieniä ohjelmistomuutoksia varten voidaan käyttää uudelleenmallinnettuja moduuleja, jotka uudelleenintegroidaan olemassa olevaan vanhaan järjestelmään. Toiminnallisesti koossapitävät moduulit täytyy selvittää, eristää ja uudelleenmallintaa. Nämä moduulit jätetään integroimatta legacy-järjestelmään. Kun lähdetään tekemään osittaista legacy-järjestelmän uudistamista, tulee tarkkaan määritellä lähde- ja tavoitearkkitehtuuri, analy-

soida legacy-lähdekoodi ja uudelleenjärjestää moduulit. (Jain & Chana, 2015.) Osittainen järjestelmän uudistaminen kuulostaa Tiedon järjestelmän kannalta paremmalta, koska nykyisen järjestelmän täytyy pysyä toiminnassa uuden järjestelmän toteuttamisen ohessa.

Pilviympäristöä hyödyntämällä voidaan vähentää IT-laitteisto- ja ylläpitokuluja ja kyetään tarjoamaan jaetussa verkossa sijaitseva IT-palveluiden kokoelma, joka koostuu erilaisista ohjelmistoista ja laitteista. Käyttämällä pilvipalveluita voidaan siis tarjota joustavaa palveluiden ja laitteiden käyttöä järjestelmän käyttäjille. (Buyya et. al., 2009.) Siirtyminen pilviympäristöön on vaativa prosessi, joka vaatii sekä uudelleenmallintamista että järjestelmän rajapintojen rajaamista ja selventämistä. Näiden asioiden takia siirtoa tehdessä tiimin tulee tietää todella tarkkaan legacy-järjestelmän osa-alueet ja toiminnot. Tärkeimmät toimenpiteet pilvipalveluun siirtymisessä ovat koodin tunnistaminen, eristäminen ja irrottaminen. (Jain & Chana, 2015.) Pilviympäristö tuo mahdollisesti mukanaan turvallisuus- ja latenssiongelmia. Jainin & Chanan (2015) mukaan ei ole suotavaa toteuttaa pilviympäristöön siirtymistä kriittisille järjestelmille ennen kuin pilvipalvelun tarjoajat ovat tarpeeksi luotettavia, jotta ne voivat tarjota korkeita siirtonopeuksia kriittisiä sovelluksia varten. Pilviympäristöjä varten ei ole myöskään vielä tarpeeksi automaatiotyökaluja siirtoprosessia varten (Jain & Chana, 2015).

Mikropalvelut voidaan ajatella palvelukeskeisen arkkitehtuurin muunnelmana. Mikropalveluissa ideana on kehittää sovellus, joka koostuu useista pienistä palveluista, jotka toimivat omina prosesseinaan ja kommunikoivat kevyesti HTTP-rajapinnan kautta. (Zimmermann, 2017.) Mikropalvelut käyttävät hyödykseen nykyaikaisia ohjelmistokehittämisen tapoja ja web-teknologiaa (Zimmermann, 2017):

- Sovellusalakeskeinen suunnittelu, jonka avulla tunnistetaan ja käsitteellistetään palveluita.
- HTTP REST-protokollan avulla luodaan etäyksiköitä, jotka muodostavat palveluita, joita voidaan ottaa käyttöön, muuttaa, korvata ja skaalata toisistaan riippumattomasti. RESTillä voidaan luoda rajapintoja, joilla yksiköiden vastuualueet saadaan selkeiksi ja data kapseloidaan ja prosessointilogiikka toimii etänä.

- Pilvisovellusten arkkitehtuuri.
- Useat tietojenkäsittelyn paradigmat, kuten esimerkiksi funktionaaliset ja imperatiiviset tavat sekä tiedonsäilytysparadigmat, kuten esimerkiksi relaatiotietokannat ja NoSQL-tallettamiset.
- Kevytrakenteiset komponenttisäilöt, jotka voivat sisältää lapsikomponentteja ja joita käytetään palveluiden käyttöönotossa.
- Jatkuva julkaisu palveluiden kehittämisessä. Tämä vaatii, että automaatio on korkealla tasolla.
- Ketterän kehittämisen tukeminen vahvasti automatisoidulla DevOps-lähestymistavalla, jota hyödynnetään osana konfiguraatiota, suorituskykyä ja virheidenhallintaa.

Mikropalveluiden kehittäminen vaatii taitavaa järjestelmäarkkitehtuuriosaamista, sillä mikropalveluiden kehittäminen edellyttää uusien suunnitteluvaihtoehtojen valintaa ja uusien arkkitehtuuripäätöksien läpikäyntiä perinteisten ”jakeluklassikoiden” ohella. Mikropalveluiden onnistunut kehittäminen vaatii myös paljon panostamista muuhun suunnitteluun, testaamiseen ja ylläpitoon. (Zimmermann, 2017.)

Tiedon asiantuntijan mukaan huonosti suunniteltu mikropalvelu voi helposti tuoda uusia ongelmia esimerkiksi tiedon kopiointiin liittyen, kun tietoa käytetään erilaisissa palveluissa ja tietokannoissa. Suurien järjestelmien kohdalla järjestelmän pilkkominen mikropalveluihin voi olla transaktioiden hallinnan takia riskialtis prosessi järjestelmän suorituskyvyn säilyttämisen kannalta. Etenkin osittaisesti etenevässä uudistamisessa on olemassa riski suorituskyvyn heikkenemiselle. (Knoche, 2016.)

Knoche (2016) ehdottaa osittaisesti etenevässä mikropalveluiksi muuttamisessa käyttämään seuraavaa lähestymistapaa:

1. Nykyinen, korvattava palvelu tunnistetaan hyödyntäen järjestelmän asiantuntijoiden ymmärrystä. Tarkoituksena on käydä läpi nykyisen järjestelmän käyttötapauksia ja selvittää nykyinen työkuorma ja suorituskykyrajoitteet, joita ovat muun muassa vasteajat, transaktioiden kesto ja läpimeno.

2. Nykyisen palvelun tuottavien osien kuten moduulien, metodien ja tietokantataulujen vaikutusalueet tunnistetaan.
3. Kehitetään uusi erillinen palvelu, joka voi aluksi olla vain aikaisempien palveluiden pohjalta tuotettu kokoelma tai sisältää hyvin pitkälti vain olemassa olevan järjestelmän toiminnallisuutta.
4. Kun uusi palvelutoteutus on valmis, nykyisen toteutuksen yhteys korvataan käyttämään uutta palvelua.
5. Usein joudutaan käyttämään monia palveluita, jotta saadaan korvattua olemassa oleva yhteys. Jos kohdassa 3 kehitettiin väliaikainen, aikaisempia ratkaisuja hyödyksi käyttävä palvelu, niin tässä vaiheessa on hyvä korvata se uudella versiolla.
6. Kun vanhan palvelun korvaus on suoritettu, vanha toteutus voidaan poistaa käytöstä.

8.4 Refaktoroitavan järjestelmän vaatimukset

Luvussa 2 (Miksi refaktorointia tehdään?) mainittiin erilaisia asioita, jotka voivat toimia vaatimuksina refaktoroinnille:

- Ohjelmiston mallin parantaminen
- Kehittämisnopeuden parantaminen
- Ylläpidettävyys
- Suorituskyky
- Käytettävyys

Lisäksi luvun 5.2 taulukossa 3 on esitetty etenkin arkkitehtuurin kannalta ohjelmiston laatuominaisuuksia, jotka voivat toimia vaatimuksina ohjelmiston refaktoroinnille:

- Vakaus
- Vasteaika
- Saatavuus
- Suoritusteho/läpivienti

- Tarpeettomuus/päällekkäisyys
- Monimutkaisuus

Laskutusjärjestelmän kohdalla etenkin ylläpidettävyydessä tulee ottaa huomioon päällekkäisten järjestelmien ylläpitäminen. Tällä hetkellä tuotannossa on käytössä useita erilaisia versioita laskutusjärjestelmästä, mikä tuo oman haasteensa mukaan ylläpidettävyyteen. Tiedolla nähdään, että ylläpidettävyys sisältää koodin ymmärtämisen lisäksi testattavuuden, modulaarisuuden ja uudelleenkäytettävyyden.

Tiedolla halutaan näiden vaatimusten lisäksi painottaa tietoturvaa, vikasietoisuutta ja virhetilanteista toipumista sekä mukautettavuutta ja skaalautuvuutta tärkeinä vaatimuksina. Osan näistä voidaan ajatella sisältyvän aikaisemmin mainittuihin vaatimuksiin, kuten esimerkiksi vikasietoisuuden ja virhetilanteista toipumisen voidaan ajatella kuuluvan saatavuuteen, ja mukautettavuuden ja skaalautuvuuden voidaan ajatella sisältyvän suoritustehon käsitykseen, mutta nämä ovat kaikki vaatimuksia, joilla on suuri merkitys ohjelmiston laadun kannalta.

Tiedolla tietoturvan/tietosuojan ajatellaan sisältävän seuraavat perustavoitteet:

- Luottamuksellisuus
- Eheys
- Käytettävyys, saatavuus
- Autenttisuus, oikeellisuus
- Pääsynvalvonta
- Kiistämättömyys

Uusien toimintaympäristöjen (toimintamallien) kuten esimerkiksi soten (sosiaali- ja terveydenhuollon) tukeminen voi tuoda paljon muutoksia ja vaatimuksia laskutusjärjestelmään. Myös asiakkaan prosesseissa voi tapahtua muutoksia, esimerkiksi kotisairaalan tuotteiden hallinta ja laskujen tuottaminen voivat muuttua ja näihin muutoksiin täytyy kyetä vastaamaan. Nykyinen järjestelmä tarvitsee uudistamista, jotta sitä on tulevaisuudessa helpompaa muokata vastaamaan uusiin toiminnallisiin vaatimuksiin, joita uudet toimintaympäristöt ja asiakkaan muuttuvat prosessit voivat aiheuttaa.

8.5 Järjestelmän suositellut uudistamisalueet

Järjestelmän uudistaminen sisältää useita erilaisia osa-alueita, joista osa voidaan toteuttaa refaktoroimalla, mutta osa vaatii isompaa muutosta.

Seuraavaksi on esitetty tärkeysjärjestyksessä uudistamisalueet, jotka vaativat vähintään järjestelmän osittaista uudelleenmallintamista tai uudelleenkirjoittamista:

Koodipuolella on parannettavaa joidenkin funktioiden selkeydessä. Osa funktioista on liian pitkiä, jotta niistä saisi helposti selvää.

- Funktioita tulisi paloitella selkeämpiin kokonaisuuksiin. Tarkoituksena olisi, että funktio tekisi juuri sen asian, minkä sen nimi kuvaa. (Fowler 2018.) Tällöin myös tarvittavan kommentoinnin määrä vähenee.
- Funktioiden pitkä pituus johtuu myös osittain monimutkaisesta bisneslogiikasta, jota järjestelmä sisältää. Tätä logiikkaa voitaisiin mahdollisesti helpottaa käyttämällä uudenlaista dynaamisempaa tapaa käsitellä logiikkaa, jotta päästään eroon if-else-logiikasta. Kovakoodatut laskutussäännöt pitäisi korvata käyttäen muokattavia toteutuksia, eli laskutussäännöt tulisi voida konfiguroida tilanteen mukaan. Ensiksi kehittäjien tulee käydä tarkasti läpi nykyinen toimintatapa, poimia oleellisimmat asiat ylös ja ymmärrettävä tarkkaan millaisia vaatimuksia järjestelmän logiikka sisältää. Esimerkiksi sääntömoottorin (tietokannassa säännöt, ei kovakoodausta) käyttö voisi helpottaa logiikan kanssa, jolloin if-else-logiikan määrä vähenisi järjestelmässä.
  - Tämä tosin tarkoittaa sitä, että bisneslogiikkaa joudutaan muokkaamaan isolla kädellä eli kyseessä on bisneslogiikan osittainen uudelleenkirjoittaminen. Tässä on mahdollisesti suuria ongelmia, koska nykyinen bisneslogiikka on tehty ja sitä on muokattu monen ohjelmistokehittäjän toimesta pitkällä aikavälillä, joten täydellisen selkeyden saaminen järjestelmän säännöstöstä on hankalaa.
  - Säännöstön ehdot eivät ole pelkästään yhdessä tai kahdessa luokassa, vaan ovat levinneet vähän joka puolelle. Ilman sääntömoottoria tilanne on se, että, jos halutaan muuttaa jotain sääntöä, esimerkiksi asiakkaan pyynnöstä

hinnoitteluperustelu muuttuu jonkin asian kohdalla, pitää muuttaa koodia, kääntää koko järjestelmän koodi ja asentaa koko järjestelmä, eli periaatteessa asiakaspää, uudelleen. Sääntömoottorissa ideana olisi se, että tietokannassa olisi säännöt (sääntömoottori), jonka mukaan suoritetaan koodi, joka asianosaiseen sääntöön liittyy. Riippuu sääntömoottorista, missä muodossa säännöstö on, mutta tyypillisesti se voi olla jokin luokka, joka toteuttaa jonkin säännön. Ero on se, että se ei ole kovakoodattu koodiin, vaan se, säännön sisältävä koodi, on periaatteessa tietokannassa, josta valitaan, että nyt suoritetaan tämä tietty asia. Esimerkiksi tietokannasta valitaan C#-koodatut sääntöosuudet, jotka tulee suorittaa. Tällainen säännöstö on mukautettavissa ja laajennettavissa, eikä vaadi aina asiakas- tai palvelinkoodin uudelleenkääntämistä eikä asiakassovelluksen tai palvelun kokonaan uudelleenasentamista. Säännöstöä voidaan jopa, DevOps-tyyliin, muokata asiakkaalle tai tehdä uusia sääntöjä ja asentaa tietokantaan jokin päivitys/paketti, tarvitsematta asentaa koko järjestelmää uudelleen.

- Koodipuolen säännöstön käsittely on osoittautunut ohjelmistokehittäjien parissa tärkeimmäksi parannettavaksi osa-alueeksi. Ohjelmistokehittäjillä kestää pitkään päästä perille säännöstöstä silloin, kun sitä pitää lähteä muokkaamaan joltain osalta. Ohjelmistokehittäjillä on ollut vaikeus löytää kaikki osa-alueet, joihin säännöstöön tehtävät muutokset vaikuttavat. Näiden asioiden takia sääntömoottorin toteuttaminen olisi suotavaa ja parhaimmillaan selkeyttäisi nykyistä ohjelmistokehittäjien työtä.

Koodiluokkien pituuksia voitaisiin lyhentää jakamalla koodi useampiin luokkiin koodin vastuualueisiin perustuen. Ensiksi tulisi selvittää, mikä osuus milläkin funktiolla on koodissa ja mitä yhtenäistä funktioiden väliltä löytyy. Voitaisiin hyödyntää myös enemmän koodia uudelleen käyttävää lähestymistapaa, jossa hyödynnetään useasti toistuvien asioiden uudelleenkäyttöä. Luokat tulisi jakaa selkeästi omiin kerroksiinsa, eikä esimerkiksi datalogiikkaa saisi esiintyä bisneslogiikalle tarkoitetuissa komponenteissa. Tässäkin tapauksessa sääntömoottorin käyttäminen vähentäisi huomattavasti nykyisten säännöstöä sisältävien luokkien kokoa.

- Pitkät luokat myös haittaavat säännöstön ymmärtämistä, minkä takia sääntömoottorin toteuttaminen olisi suotavaa.

Seuraavassa on suositellut uudistamisalueet, jotka koskevat läheisesti refaktorointia:

Automaattisen testaamisen kasvattaminen on yksi oleellisimmista osa-alueista koskien refaktorointia. Toimiva testaaminen on oleellinen tekijä, kun järjestelmään tehdään muutoksia (Leppänen et al., 2015). Automaattisilla testeillä voidaan taata vähemmällä toistolla, että tehty koodi oikeasti toimii kuten pitääkin. Automaattisten testien käyttäminen voi auttaa huomattavasti ohjelmointivirheiden karsimisessa ja ylipäänsä ajan säästämisessä, sillä suuri osa ohjelmoijien ajasta menee ohjelmistovirheiden etsimiseen ja korjaamiseen. (Fowler, 2018.) Ohjelmistovirheet saataisiin automaattisella testaamisella helpommin, ja ennen kaikkea nopeammin, kiinni, eikä vahinkoja bisneslogiikan käsittelyssä pääsisi esiintymään, koska ei päästetä pinnalle uusia ohjelmistovirheitä.

- Automatiikan lisääminen testaamisen puolella on refaktorointiasioista korkeimmalla prioriteetilla, koska ilman automaattista testaamista jokainen refaktorointimuutos vaatii manuaalisen testaamisen, mikä hidastaa huomattavasti refaktorointiprosessia. Kuitenkin on hyvin tärkeää muistaa myös manuaalinen testaaminen, jota ehdottomasti tarvitaan varmistamaan etenkin monimutkaisten muutosten onnistuminen.

Tietokantamallia voisi refaktoroida ja optimoida. Refaktoroimalla saadaan selkeytettyä nykyistä tietokantamallia ja mahdollisesti myös optimoitua tietokantojen käsittelyä. Optimointi voi tulla myös hieman sivutuotteena osana refaktorointia, koska refaktoroidessa tietokantamallia saadaan samalla selkeytettyä nykyistä toimintalogiikkaa ja täten tietokantojen käsittelyn tehostaminen on helpompaa ja nopeampaa. Myös kahdennettu data ja synkronaatio tietokantojen välillä pitää poistaa ja useiden laskutustuotteiden tuotehintojen lähteet pitäisi yhdistää. Nämä luetaan osaksi refaktorointia. Kahdennetun datan käsittely vaatii ohjelmistokehittäjiltä enemmän aikaa, koska muutokset täytyy mahdollisesti tehdä useaan kohtaan järjestelmää, mikä tekee järjestelmän kehittämisestä sekavampaa. (Fowler, 2018). Lisäksi kahdennetun datan synkronointi

voi tuoda esille ongelmia, jos synkronoinnissa pääsee tapahtumaan vikatilanne ja data ei kahdennukaan tai kulje oikein järjestelmän tietokannan taulujen välillä. Vähentämällä synkronointitarpeita pienenee myös riski kohdata synkronointivirheitä.

- Refaktoroimalla ja optimoimalla nykyistä tietokantamallia saavutettaisiin selkeyttä nykyiseen arkkitehtuuriin, mikä näkyisi myös ohjelmistokehittäjien tasolla. Optimointi toisi kaivattua lisänopeutta suurien tietomäärien käsittelyyn.

Potilastietojärjestelmän ja -laskutusjärjestelmän vastuualueiden noudattaminen. Tämä vaatii rajapintojen selventämistä ja voidaan korjata refaktoroimalla koodin vastuualueet kuntoon.

Nykyisistä palvelusovelluksista ja rajapinnoista osa vaatii refaktorointia, jotta ohjelmistokehittäminen muuttuisi vielä selkeämmäksi ja joustavammaksi tulevaisuutta ajatellen.

Käyttäjärajapinnat. Tämä on mahdollista toteuttaa refaktoroimalla nykyistä järjestelmää. Käytettävyyden parantaminen on yksi mahdollisista refaktoroinnin osa-alueista. Garrido et al. (2010) jakavat web-sovellusten refaktoroinnin navigaatio- ja presentaatiomalleihin sekä erillisiin käytettävyystekijöihin ja vastaavaa refaktorointia voisi tehdä myös muille kuin web-sovelluksille.

8.6 Mitä onnistunut refaktorointi vaatii ja mitä riskejä se sisältää?

Onnistunut refaktorointi vaatii, että ymmärretään nykyiset parannettavat osa-alueet ja uudet muuttuneet vaatimukset sekä aikataulut. Täytyy ymmärtää, mitä halutaan refaktoroida ja miten voidaan tehdä refaktorointi vaiheittain. Toimintaympäristötuntemus täytyy olla tiedossa. Täytyy tietää, mitä tarkoittaa esimerkiksi kooditasolla se, että asiat saadaan toteutettua ja saadaan laadulliset tavoitteet saavutettua. Aina, kun tehdään uudistuksia, tulisi olla käytettävissä asiantuntijoita, jotka pystyvät hahmottamaan koko uudistamisketjun alusta asti joka puolella organisaatiota.

Jotta jatkossa voitaisiin mahdollisesti tehdä toimivampaa uusien ominaisuuksien lisäämistä ohjelmistoon, olisi kehittäjien hyvä käyttää enemmän TDD:n tapaista ohjelmistokehittämistapaa hyödyksi. Tällä toiminnalla voidaan ehkäistä tulevaisuudessa suurempien refaktorointien tarvetta, koska koodin laatu pidetään jatkuvasti mahdollisimman hyvänä. Myös riskit tehdä asioita väärin vähenee, kun asioita on kerralla vähemmän korjattavana ja parannettavana. Testejä tulisi siis jatkossa kirjoittaa ensimmäiseksi, kun luodaan uusia ominaisuuksia tai kun löydetään uusi ohjelmointivirhe. (Fowler, 2018). Tällä tavoin ohjelmistokehittäjä joutuu miettimään tarkkaan, mitä täytyy tehdä funktion lisäämiseksi ja hänen täytyy keskittyä rajapintaan enemmän kuin toteutukseen. (Fowler, 2018.) TDD:n testin, koodaamisen ja refaktoroinnin muodostama sykli tulisi tapahtua ihanteellisesti useita kertoja tunnissa. (Fowler, 2018.)

Yksi riskeistä on järjestelmän muutosten negatiiviset vaikutukset muihin järjestelmiin. Järjestelmä on liitoksissa rajapinnoilla useisiin muihin järjestelmiin. Järjestelmien välillä kulkee paljon dataa, joten synkronointi on jatkuvaa ja täten on turvattava, että myös refaktorointien jälkeen järjestelmät toimivat oikealla tavalla yhdessä. Työntekijöiden perehtyneisyys aikaisempaan järjestelmän versioon sekä perehtyminen tulevaan refaktorointiprosessiin on myös tärkeässä roolissa. Jos refaktorointia tekevä tiimi ei ymmärrä hyvin nykyisen järjestelmän rakennetta, järjestelmästä ei saada refaktoroinnin avulla tehtyä aiempaa selkeämpää (Manclin, 2001.) Refaktorointitiimin tulisi aina tutkia järjestelmän koodia ja rakennetta tarkasti ennen muutosten tekemistä legacy-koodiin.

Koodipuolen refaktorointi voi olla riskialtista. On ensiarvoisen tärkeää suorittaa refaktorointi systemaattisesti ja täten saada selville jokainen refaktorointia vaativa osa-alue, koska muuten refaktorointia tehdessä huomataan, että aina löytyy lisää refaktorointia tarvitsevia kohteita ja aikataulut voivat heittää useita viikkoja. Refaktorointi voi myös aiheuttaa toimivaan koodiin muutoksia, jotka voivat nostaa esille ovelia ohjelmointivirheitä, joten automaattinen testaaminen on tärkeässä roolissa. (Fowler, 2018.)

Arkkitehtuuripuolella kahdentuneen datan poistaminen käytöstä voi tuottaa haasteita, koska tämän toteuttaminen vaatii tietokantamalleihin muutoksia. Kaikkien järjestelmän riippuvaisuuksien selvittäminen ja arkkitehtuurimallin selkeyttäminen sen jälkeen

on työläs prosessi. Leppänen et al.:in (2015) haastattelema arkkitehti totesi hankalaksi muokata metodeja ja rajapintoja, joista ulkoiset palvelut ovat riippuvaisia. Kun muokataan tällaista logiikkaa, se voi rikkoa ulkoisen toiminnan ja aiheuttaa ongelmia ulkoiselle toimijalle. Myöskään koodin rakenteet eivät aina parantuneet tai aikaansaatu sisäinen koodin laatu saattoi jopa huonontua refaktoroinnin myötä. Tässä korostuu se, että isompiin refaktorointeihin ei tule ryhtyä ilman aikaisemman järjestelmän todella hyvää tuntemusta ja panostamista refaktorointivaiheiden ymmärtämiseen.

8.7 Uudistamisen vaikutus bisnekseen ja käytettävät resurssit

Refaktorointi hidastaa hetkellisesti uusien toiminnallisuuksien lisäämistä (Fowler, 2018), mikä osaltaan vaikuttaa negatiivisesti asiakkaiden muutosvaatimuksiin vastaamiseen ja voi vaikuttaa negatiivisesti myös bisnekseen. Refaktoroinnilla saataisiin kuitenkin pidemmällä aikatähtäimellä helpotettua ohjelmistokehittämistä, mikä puolestaan nopeuttaisi asiakkaiden vaatimiin muutoksiin vastaamista ja mahdollisesti alentaisi myös asiakaspalvelun kuluja. Asiakaskunnan luottamuksen ylläpitäminen ja bisneksen jatkuvuus ovat tärkeässä osassa ja siksi tuleekin miettiä, mikä tapa uudistaa järjestelmää on viisainta. Osittainen uudistaminen, jonka ohessa pidetään yllä vanhaa järjestelmää, vaikuttaa parhaalta ratkaisulta nykytilanteessa. Jotta voitaisiin tehdä suurempaa muutosta kerralla ja samanaikaisesti pitää yllä nykyistä järjestelmää, vaatisi se myös enemmän ohjelmistokehittäjäresursseja.

Uudistusten jälkeen ohjelmistokehittäminen voisi olla nopeampaa, mikä parantaisi järjestelmän kehittämiskustannuksia. Jos vanhaa järjestelmää joudutaan ylläpitämään yhtä aikaa uuden kehittämisen ohella, kuinka paljon voidaan varata resursseja? Jos järjestelmää lähdetään uudistamaan ohjelmistokehittämisen helpottamiseksi, olisi sääntömoottori yksi tärkeimmistä asioista. Luultavasti tämä veisi muutaman ohjelmistokehittäjän ajan pitkällä aikavälillä, johtuen järjestelmän säännöstön hankaluudesta. Automaattisen testaamisen kasvattaminen vie myös ohjelmistokehittäjiltä aikaa, mutta jos automaattinen testaaminen olisi pitkälle valmiina ennen kuin sääntömoottoria kehitettäisiin, voitaisiin kenties huomata ajoissa, jos uudessa sääntömoottorissa tulisi

vastaan bisneslogiikkavirheitä tai -muutoksia verrattuna vanhan järjestelmän säännöstöön. Järjestelmän rajapintoja joudutaan uudelleenmallintamaan tai kirjoittamaan uusiksi, riippuen siitä kuinka isoja muutoksia datan käsittelyyn tulee, mutta alustava refaktorointi voisi selkeyttää tulevaa prosessia.

9 YHTEENVETO

Tutkielman tavoitteena oli löytää vastauksia ja selkeyttä seuraaviin tutkimuskysymyksiin:

1. Mitkä ovat tärkeimmät tavoitteet järjestelmän refaktoroinnissa ja miksi?

Järjestelmän refaktoroinnissa tärkeimpänä tavoitteena on selkeyttää nykyisen järjestelmän kehittämistyötä. Myös Tiedolla huomattiin, että järjestelmän ymmärrettävyys on yksi pääsyistä, minkä takia refaktorointia ja muuta uudistamista kannattaa ja halutaan tehdä. Tiedon asiantuntijat kokevat, että nykyisen järjestelmän ylläpitäminen ja uusien asioiden kehittäminen on hankalaa. Refaktoroinnin ajatellaan tuovan selkeyttä ja helpotusta, niin ohjelmistokehittäjien, -arkkitehtien kuin testaajien työhön. Lisäksi Tiedolla halutaan refaktoroinnin avulla helpottaa tuleviin vaatimuksiin vastaamista ja mahdollisesti, refaktoroinnin tuoman kehittämistyön helpottumisen ja selkeyden kautta, parantaa myös järjestelmän suorituskykyä. Suorituskyky itsessään ei ole refaktoroinnin päätavoite, mutta esimerkiksi Tiedolla refaktoroinnin jälkeen voidaan ajatella saavutettavan helpommin suorituskykyparannuksia järjestelmään.

2. Milloin refaktorointi on hyvä vaihtoehto järjestelmän uudistamiselle ja milloin kannattaa tehdä jotain muuta kuin refaktoroida, esimerkiksi uudelleenmallintaa tai uudelleenkirjoittaa järjestelmää?

Refaktorointia itsessään tulisi tehdä jatkuvana työnä. Refaktorointi on hyvä vaihtoehto järjestelmän uudistamiselle, jos sen voidaan ajatella kooditasolla helpottavan ohjelmistovirheiden korjaamista, eivätkä nämä korjaukset tuo esille mahdollisesti uusia tai vanhoja ohjelmistovirheitä. Jos järjestelmä vaatii toimintalogiikan muutoksia, niin tarvittava muutos ei enää vastaa refaktoroinnin määritelmää, vaan joudutaan tekemään isompia muutoksia, kuten uudelleenmallintamista tai uudelleenkirjoittamista. Uudelleenmallintamisessa ja uudelleenkirjoittamisessa muutosvaikutukset ulottuvat lisäksi myös toiminnallisuuden ja toimintakyvyn puolelle.

Päätös koodin refaktoroinnin ja koodin uudelleenkirjoittamisen välillä vaatii hyvää arviointikykyä ja kokemusta. On hankalaa päättää yleisellä tasolla, milloin refaktorointia ei enää kannata tehdä, vaan koodi kannattaa kirjoittaa uudelleen alusta asti. Tämä vaatii yleensä perehtymistä koodiin ja analysointia siitä, miten hankalaa koodista on saada selvää. Arkkitehtuuritasolla tilanteissa, joissa refaktoroinnilla saavutetaan vain oireiden poistuminen, mutta ei juuritason ongelmien korjautumista, kannattaa miettiä muita näkökulmia arkkitehtuurin uudistamiseen, esimerkiksi uudelleenmallintamista tai uudelleenkirjoittamista. Voidaan esimerkiksi huomata, että ohjelmiston nykyinen toimintamalli on jo huonontunut hoitamattoman arkkitehtuurikulumisen myötä. Näiden tilanteiden havaitsemiseen kannattaa käyttää hyödyksi ohjelmistoinsinöörien tietotaitoa.

Tutkimuskysymykset olivat mielestäni onnistuneet. Refaktorointi on välillä hieman väärin ymmärretty asia, joten refaktoroinnin käsite ja sen tavoitteet oli hyvä selvittää monelta näkökulmalta katsottuna. Refaktoroinnin käyttömahdollisuuksien arvioiminen osana järjestelmän uudistamista oli hankalampi asia tutkia ja olikin huomattavissa, että välillä voi olla vaikeaa päättää, kannattaako järjestelmää refaktoroida vai kannattaako järjestelmään tehdä isompia muutoksia.

Tutkimustulokset kuitenkin selkeyttivät asiaa, ja mahdollisessa jatkotutkimuksessa voitaisiin selvittää konkreettisella tasolla, kuinka refaktorointi vertautuu kannattavuudessa uudelleenmallintamiseen ja uudelleenkirjoittamiseen. On hankalaa sanoa tämän tutkimuksen pohjalta varmasti, milloin refaktorointi on paras vaihtoehto järjestelmän rakenteelliselle uudistamiselle. Tutkimustulokset nojautuivat pitkälti aikaisempien tutkimusten toteamuksiin ja tuloksiin, sekä Tiedon järjestelmän tutkimiseen. Jotta tutkimus olisi luotettavampi, tulisi refaktorointia, uudelleenmallintamista ja uudelleenkirjoittamista vertailla toisiinsa samoja asioita uudistaessa. Tutkimustulosten luotettavuuden osalta on rajoittavana tekijänä myös se, että tapaustutkimusta varten tutkittiin vain yhtä potilastietojärjestelmän osaa. Tähän vaikutti pitkälti aikataulu, koska muihin potilastietojärjestelmän osiin tarvittava perehtyminen veisi huomattavan paljon aikaa.

Jos Tiedolla päätetään uudistaa laskutusjärjestelmää luvussa 8 mainituilla tavoilla, on mielenkiintoista nähdä kuinka refaktorointi ja muu uudistaminen tulee onnistumaan.

Tämän perusteella voitaisiin vielä paremmin päätellä, miten refaktorointi vertautuu uudelleenmallintamiseen ja uudelleenkirjoittamiseen konkreettisella tasolla, esimerkiksi tarvittavien resurssien osalta. Järjestelmän uudistamisesta saatavasta tiedosta olisi mahdollisesti paljon hyötyä muita tulevia uudistamisia suunniteltaessa, niin laskutusjärjestelmän kuin muidenkin järjestelmien puolella.

Viitteet

Agile Alliance (2019): *Extreme Programming*. https://www.agilealliance.org/glossary/xp/ [Viitattu 9.6.2019].

Arcelli, D., Cortellessa, V., Di Pompeo, D. (2018): Performance-Driven Software Model Refactoring. Teoksessa: *Information and Software Technology*, 95. S. 366-397. (DOI: https://doi.org/10.1016/j.infsof.2017.09.006).

Buyya, R., Yeo, C., Venugopal, S., Broberg, J., Brandic, I. (2009): Cloud Computing and Emerging IT Platforms: Vision, Hype, and Reality for Delivering Computing as the 5th Utility. *Future Generation Computer Systems*, 25(6). S. 599-616. (DOI: https://doi.org/10.1016/j.future.2008.12.001).

Crispin, L. (2006): Driving Software Quality: How Test-Driven Development Impacts Software Quality. *IEEE Software*, 23(6). S. 70-71. (DOI: 10.1109/MS.2006.157).

Dig, D., Johnson, R. (2005): The Role of Refactoring in API Evolution. Teoksessa: Cantarella, J. D. *21st IEEE International Conference on Software Maintenance (ICSM'05)*. Budapest, Unkari. 26.-29.9.2005. Los Alamitos, Kalifornia, USA: IEEE Computer Society. S. 389-398. (DOI: 10.1109/ICSM.2005.90).

Extreme Programming (2013): *Extreme Programming: A Gentle Introduction*. http://www.extremeprogramming.org/ [Viitattu 9.6.2019].

Fowler, M. (2018): *Refactoring: Improving the Design of Existing Code*. Boston, USA: Addison-Wesley Professional.

Garrido, A., Rossi, G., Distante, D. (2010): Refactoring for Usability in Web Applications. *IEEE Software*, 28(3). S. 60-67. (DOI: 10.1109/MS.2010.114).

Garrido, A., Firmenich, S., Grigera, J., Rossi, G. (2017): Data-Driven Usability Refactoring: Tools and Challenges. Teoksessa: Li, M., Wang, X., Lo, D. *2017 6th International Workshop on Software Mining (SoftwareMining)*. Urbana, Illinois, USA. 3.11.2017. Los Alamitos, Kalifornia, USA: IEEE Computer Society. S. 52-55. (DOI: 10.1109/SOFTWAREMINING.2017.8100854).

Gatrell, M., Counsell, S. (2015): The Effect of Refactoring on Change and Fault-Proneness in Commercial C# Software. *Science of Computer Programming*, 102. S. 44-56. (DOI: https://doi.org/10.1016/j.scico.2014.12.002).

Jain, S., Chana, I. (2015): Modernization of Legacy Systems: A Generalised Roadmap. Teoksessa: *ICCCT' 15 Proceedings of the Sixth International Conference on Computer and Communication Technology 2015*. Allahabad, Intia. 25.-27.9.2015. New York, USA: ACM. S. 62-67. (DOI: 10.1145/2818567.2818579).

Jezek, K., Dietrich, J. (2017): API Evolution and Compatibility: A Data Corpus and Tool Evaluation. *Journal of Object Technology*, 16(4). S. 1-23. (DOI: 10.5381/jot.2017.16.4.a2).

Kaur, G., Singh, B. (2017): Improving the Quality of Software by Refactoring. Teoksessa: *2017 International Conference on Intelligent Computing and Control Systems*. Madurai, Intia. 15.-16.6.2017. Piscataway, New Jersey, USA: Institute of Electrical and Electronics Engineers (IEEE). S. 185-191. (DOI: 10.1109/ICCONS.2017.8250707).

Kim, S., Cai, D., Kim, M. (2011): An Empirical Investigation into the Role of API-Level Refactorings during Software Evolution. Teoksessa: *2011 33rd International Conference on Software Engineering (ICSE)*. Waikiki, Honolulu, USA. 21.-28.5.2011. New York, USA: ACM. S. 151-160. (DOI: 10.1145/1985793.1985815).

Knoche, H. (2016): Sustaining Runtime Performance while Incrementally Modernizing Transactional Monolithic Software towards Microservices. Teoksessa: *ICPE '16*

Proceeding of the 7th ACM/SPEC on International Conference on Performance Engineering. Delft, Alankomaat. 12.-16.3.2016. New York, USA: ACM. S. 121-124. (DOI: 10.1145/2851553.2892039).

Kumar, M., Kumar, R. (2011): Architectural Refactoring of a Mission Critical Integration Application: A Case Study. Teoksessa: *ISEC '11 Proceedings of the 4th India Software Engineering Conference*. Thiruvananthapuram, Kerala, Intia. 24.-27.2.2011. New York, USA: ACM. S. 77-83. (DOI: 10.1145/1953355.1953365).

Leppänen, M., Mäkinen, S., Lahtinen, S., Sievi-Korte O., Tuovinen A., Männistö, T. (2015): Refactoring—a Shot in the Dark? *IEEE Software*, 32(6). S. 62-70. (DOI: 10.1109/MS.2015.132).

Mancl, D. (2001): Refactoring for Software Migration. *IEEE Communications Magazine*, 39(10). S. 88-93. (DOI: 10.1109/35.956119).

Misbhauddin, M., Alshayeb, M. (2013): UML Model Refactoring: A Systematic Literature Review. *Empirical Software Engineering*, 20(1). S. 206-261. (DOI: 10.1007/s10664-013-9283-7).

Mohan, M., Greer, D., McMullan, P. (2016): Technical Debt Reduction Using Search Based Automated Refactoring. *Journal of Systems and Software*, 120. S. 183-194. (DOI: https://doi.org/10.1016/j.jss.2016.05.019).

Napaggan, N., Maximilien, E., Bhat, T., Williams, L. (2008): Realizing Quality Improvement Through Test Driven Development: Results and Experiences of Four Industrial Teams. *Empirical Software Engineering*, 13(3). S. 289-302. (DOI: 10.1007/s10664-008-9062-z).

OpenJDKWiki (2017): *SigTest*. https://wiki.openjdk.java.net/display/CodeTools/sigtest [Viitattu 17.6.2019].

Santos, B., Guzman, I., Camargo, V., Piattini, M., Ebert C. (2018): Software Refactoring for System Modernization. *IEEE Software*, 35(6). S. 62-67. (DOI: 10.1109/MS.2018.4321236).

Stal, M. (2013): Refactoring Software Architecture. Teoksessa: Babar, A.M., Brown, A., Koskimies K., Ivan M. (2013): *Agile Software Architecture: Aligning Agile Processes and Software Architectures*. Amsterdam, Alankomaat: Elsevier Science & technology.

Szoke, G., Antal, G., Nagy, C., Ferenc, R., Gyimóthy, T. (2017): Empirical Study on Refactoring Large-scale Industrial Systems and its Effects on Maintainability. *The Journal of Systems and Software*, 129. S. 107-126. (DOI: https://doi.org/10.1016/j.jss.2016.08.071).

UML (2019): *What is UML*. https://www.uml.org/what-is-uml.htm [Viitattu 15.6.2019].

Zimmermann, O. (2015): Architectural Refactoring: A Task-Centric View on Software Evolution. *IEEE Software*, 32(2). S. 26-29. (DOI: 10.1109/MS.2015.37).

Zimmermann, O. (2017): Microservices Tenets. *Computer Science – Research and Development*, 32(3-4). S. 301-310. (DOI: 10.1007/s00450-016-0337-0).