Tampereen yliopisto

Janne Taskinen

REFAKTOROINTI VAI UUDELLEENKIRJOITUS: HUOMIOITAVAT TEKIJÄT

Kandidaattitutkielma
Informaatioteknologian ja viestinnän tiedekunta
Huhtikuu 2023

TIIVISTELMÄ

Janne Taskinen: Refaktorointi vai uudelleenkirjoitus: Huomioitavat tekijät
Kandidaattitutkielma
Tampereen yliopisto
Tietojenkäsittelytieteiden tutkinto-ohjelma
Huhtikuu 2023

Kaupallisen ohjelmistotuotteen tavoitteena on tuottaa yritykselle rahallista voittoa. Jotta tuote voi menestyä, tulee sen ratkaista jokin reaalimaailman ongelma. Tämän lisäksi ohjelmiston tulee olla asiakkaiden saatavilla oikeaan aikaan. Liiketoimintarealiteettien vuoksi ohjelmistokehityksessä joudutaan monesti priorisoimaan uusien toiminnallisuuksien saattamista valmiiksi teknisten laadunparannustöiden sijaan. Mikäli teknisiä laadunparannuksia lykätään jatkuvasti, ohjelmiston tekninen velkataakka kasvaa. Tilanteessa, jossa ohjelmiston tekninen velkataakka on kasvanut sietämättömäksi, voidaan joutua tekemään päätös laajamittaisen refaktorointityön ja ohjelmiston uudelleenkirjoittamisen välillä.

Tässä tutkielmassa tarkastellaan ohjelmiston refaktorointia sekä uudelleenkirjoitusta kaupallisten ohjelmistotuotteiden näkökulmasta. Tutkielman tavoitteena on selvittää, mitä tekijöitä tulee huomioida, kun ohjelmistoprojektissa ilmenee tarve tehdä päätös refaktoroinnin ja uudelleenkirjoittamisen välillä. Tämän lisäksi tutkielmassa etsitään mahdollisia kompromisseja vaihtoehtojen välillä. Tutkielma on toteutettu kirjallisuuskatsauksena. Tutkielman lähdemateriaali on koostettu pääosin tietokannoista, kuten ACM Digital Library, IEEE Xplore, sekä Web Of Science.

Tutkielmassa ilmenee, että päätöksessä huomioitavia tekijöitä on useita, sekä niiden painoarvo vaihtelee tapauskohtaisesti. Tärkeinä tekijöinä voidaan pitää muun muassa teknisen velan määrää, työn ajoitusta markkinatilanteeseen peilaten, sekä käytettävissä olevaa budjettia. Tekijät ovat johdettu muun muassa tarkastelemalla reaalimaailman esimerkkejä toteutuneista uudelleenkirjoitushankkeista. Tutkielmassa huomataan, että refaktorointi on yleisesti suositeltu vaihtoehto, sillä uudelleenkirjoitus on monessa tapauksessa liian raskas sekä kallis prosessi refaktorointiin verraten. Refaktoroinnilla ei kuitenkaan lähtökohtaisesti saavuteta ohjelmistoon käyttäjälle näkyviä parannuksia, kuten uusia ominaisuuksia. Tutkielmassa esitellään lisäksi kuristusmenetelmä, jota voidaan pitää refaktoroinnin ja uudelleenkirjoituksen välimuotona tai kompromissivaihtoehtona.

Avainsanat: refaktorointi, tekninen velka, uudelleenkirjoitus, kuristusmenetelmä

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck –ohjelmalla.

SISÄLLYSLUETTELO

1. JOHDANTO 1
2. REFAKTOROINTI 3
2.1 Koodihajut 3
2.2 Tekninen velka 6
2.3 Refaktorointiin liittyviä ongelmia 7
3. UUDELLEENKIRJOITUS 9
3.1 Uudelleenkirjoitusprojektiin ryhtyminen 9
3.2 Esimerkkejä uudelleenkirjoitusprojekteista 11
3.2.1 Netscape 11
3.2.2 Remesh 12
3.3 Kuristusmenetelmä 14
4. PÄÄTÖKSENTEKO REFAKTOROINNIN JA UUDELLEENKIRJOITUKSEN VÄLILLÄ 16
5. TULOKSET JA POHDINTA 18
6. YHTEENVETO 21
LÄHTEET 22

1. JOHDANTO

Ohjelmistotuotetta voidaan ajatella ohjelmistoprojektina, joka ei ole koskaan valmis. Menestyvä ohjelmistotuote kehittyy sen käyttäjien ja muiden sidosryhmien tarpeiden mukaan jatkuvasti. Ohjelmistotuotteen elinkaaren aikana ohjelmistoon saatetaan esimerkiksi lisätä uusia loppukäyttäjille tarpeellisia ominaisuuksia, toteuttaa integraatioita kolmansien osapuolien ohjelmistoihin tai tehdä teknisiä laadunparannuksia. Laadunparannuksilla voidaan tarkoittaa esimerkiksi koodin yhtenäistämistä tai ohjelmiston suorituskykyyn liittyviä optimointitoimenpiteitä.

Tilanteessa, jossa ohjelmistoa on kehitetty teknisiä laadunparannuksia laiminlyöden, voidaan huomata, että uusien ominaisuuksien lisääminen ohjelmistoon alkaa hidastua. Tämä ilmiö indikoi monesti sitä, että ohjelmisto on teknisesti velkaantunut. Ohjelmiston tekninen velkaantuminen voi olla laukaiseva tekijä kysymykselle: pitäisikö ohjelmistoa refaktoroida, vai onko se jo niin velkaantunut, että se tulisi kokonaan uudelleenkirjoittaa?

Tässä tutkielmassa keskitytään kysymykseen: refaktorointi vai uudelleenkirjoitus. Tutkimuskysymyksenä on selvittää, mitä tekijöitä tulee huomioida, kun tehdään päätöstä refaktoroinnin ja uudelleenkirjoittamisen välillä. Lisäksi tutkielmassa sivutaan kysymystä myös siitä näkökulmasta, miten kyseiseen tilanteeseen joutumista voitaisiin ennaltaehkäistä, sekä etsitään mahdollisia kompromissivaihtoehtoja edellä mainittujen vaihtoehtojen väliltä. Tutkimuskysymyksessä keskitytään kaupallisiin, eli rahallista voittoa tavoitteleviin ohjelmistoihin. Tämän vuoksi esimerkiksi avoimen lähdekoodin menetelmiin perustuvat ohjelmistoprojektit ovat rajattu pois.

Aihe on tärkeä esimerkiksi siitä syystä, että puutteellisin perustein tehty päätös lähteä uudelleenkirjoittamaan ohjelmistotuotetta voi joissain tapauksissa saattaa yrityksen suuriin taloudellisiin haasteisiin, tai jopa konkurssiin. Toisaalta refaktorointi voi olla joissain tapauksissa turhan kevyt tai jopa mahdoton operaatio. Yksi esimerkki on tilanne, jossa ohjelmiston lähdekoodi on suurilta osin vanhentunutta ja pohjimmainen ymmärrys siitä on vain jo projektista poistuneilla kehittäjillä. Päätöstä refaktoroinnin ja uudelleenkirjoittamisen välillä ei siis tule tehdä puutteellisin perustein ja onkin tärkeä ymmärtää, mitkä tekijät päätökseen voivat vaikuttaa ja mitä seuraamuksia sillä voi olla. Toisaalta tulee miettiä, onko refaktorointi tai uudelleenkirjoitus ylipäätänsä ratkaisu ongelman juurisyyhyn.

Tutkielma on toteutettu kirjallisuuskatsauksena. Tutkielman lähteiden hakuun on käytetty tietokantoja, kuten ACM Digital Library, IEEE Xplore, sekä viittaustietokantaa Web of Science. Ohjelmiston uudelleenkirjoitusta käsitteleviä tieteellistä aineistoa on tarjolla niukasti, joten osa etenkin ohjelmiston uudelleenkirjoitusta käsittelevistä lähteistä on peräisin verkkosivuilta ja blogikirjoituksista.

Luvussa 2 esitellään refaktorointi-käsite ja siihen liittyviä muita käsitteitä, kuten tekninen velka. Luku sisältää lisäksi tutkimusta refaktorointiin liittyvistä ongelmista. Luvussa 3 esitellään uudelleenkirjoitus-käsite, sekä tutkitaan, mitä asioita tulisi huomioida uudelleenkirjoitus hankkeeseen lähtiessä. Luvussa annetaan lisäksi kaksi esimerkkiä toteutuneista uudelleenkirjoitushankkeista, joista toisen voidaan katsoa epäonnistuneen ja toisen onnistuneen. Luvussa 3 esitellään myös niin kutsuttu kuristusmenetelmä, jota voidaan pitää refaktoroinnin ja uudelleenkirjoituksen välimuotona tai tapana toteuttaa uudelleenkirjoitus. Luvussa 4 käsitellään päätöksentekoa refaktoroinnin ja uudelleenkirjoituksen välillä, sekä pohditaan mahdollisia vaihtoehtoisia ratkaisuja. Luku 5 sisältää tulostaulukon, jolla vastataan tutkimuskysymykseen. Luku 6 sisältää tutkielman yhteenvedon.

2. REFAKTOROINTI

Refaktoroinnilla (eng: refactoring) tarkoitetaan toimenpidettä, jossa parannetaan ohjelman koodin laatua ilman, että muutoksella on vaikutusta ohjelman käyttäjälle näkyvään toiminnallisuuteen. Refaktoroinnin päätavoitteena on tehdä ohjelman koodista ymmärrettävämpää ja kustannustehokkaampaa tulevia muutoksia ajatellen (Fowler, 2018). Esimerkkejä refaktoroinnista ovat toimenpiteet, kuten koodihajujen (eng: code smells) poisto, koodin laatumittareiden, kuten yhteen liittämisen ja koheesion (eng: coupling and cohesion) parantaminen ja ei-toiminnallisiin (eng: non-functional) vaatimuksiin liittyvät parannuskeinot, kuten ohjelman ylläpidettävyyden parantaminen (Cinnéide & Yamashita & Counsell, 2016).

Ohjelmiston koodiin tehdään ohjelmiston elinkaaren aikana muutoksia usean ohjelmistokehittäjän toimesta. Myös ulkopuolisten tekijöiden, kuten projektin tiukan aikataulun takia saatetaan poiketa hyvistä ohjelmistokehitystavoista. Tavoista poikkeaminen voi olla esimerkiksi ominaisuuden implementointi siten, että ominaisuus itsessään toimii, mutta kirjoitettu koodi on vaikealukuista tai se ei vastaa sovittua arkkitehtuurimallia. Onnistuneissa ohjelmistoprojekteissa onkin väistämätöntä palata välillä taaksepäin, eli refaktoroida jo olemassa olevaa koodia, jotta koodin sisäinen laatu pysyy hyvänä ja uusien ominaisuuksien implementointi on jouhevampaa.

2.1 Koodihajut

Koodihajuilla tarkoitetaan koodin rakenteellisia ominaisuuksia, jotka indikoivat koodiin liittyvistä suunnitteluongelmista esimerkiksi sen ylläpidettävyyden näkökulmasta (Rasool & Arshad, 2015). Koodihajujen kasvava määrä voi johtaa tilanteeseen, jossa refaktorointia on lähdettävä suorittamaan, mikäli kehitettävän ohjelmiston koodin ylläpidettävyys halutaan säilyttää kohtuullisella tasolla. Koodihajujen suuri määrä voikin olla yksi merkittävistä tekijöistä, kun tehdään päätöksiä ohjelmistokoodin refaktoroinnista.

Koodihajuja havaitaan lähtökohtaisesti automaattisilla työkaluilla. Yksi esimerkki suositusta työkalusta koodihajujen havaitsemiseen on analysointityökalu SonarQube (2023). SonarQube (2023) tarjoaa web-pohjaisen analysointityökalun, sekä ohjelmointiympäristöön (eng: Integrated development environment, IDE) liitettävän SonarLint moduulin. SonarLint -moduulin avulla koodihajuja voidaan havaita jo koodin tuottamisen aikana.

Koodihaju ei ole sama asia, kuin ohjelmistovirhe. Koodi ja sitä myötä itse ohjelmisto voi olla täysin toimivaa, vaikka se sisältäisi koodihajuiksi määriteltyjä ominaisuuksia. Refaktoring Guru (2023) kategorisoi koodihajut viiteen kategoriaan. Kategoriat ja esimerkkejä kategorioihin kuuluvista koodihajuista on listattu taulukkoon 1.

Taulukko 1. Koodihajujen kategorioita ja esimerkkejä (Refactoring Guru, 2023).

Kategoria	Esimerkkejä
Paisunut koodi	• Rivimäärältään suuri funktio tai luokka • Liiallinen primitiivien käyttö esimerkiksi pienien luokkien sijaan • Funktio, jolle annetaan syötteenä liikaa parametreja • Tietojoukot (eng: data clumps). Joukko yhtenäisiksi katsottavia muuttujia, joita käsitellään irrallisina sen sijaan, että ne muodostaisivat esimerkiksi olion
Ohjelmointiparadigman (esimerkiksi olio-ohjelmointi) väärinkäyttö	• Useampi luokka, joissa on liikaa samankaltaisuutta • Periytymisen väärinkäyttö. Esimerkiksi perityssä luokassa käytetään vain osaa periytetyn luokan ominaisuuksista • Switch … case valintarakenteen käyttö
Koodin muutoksen estäjät	• Yksittäisen muutoksen tekeminen aiheuttaa muutostarpeen useampaan paikkaan • Rinnakkaiset perintähierarkiat
Tarpeeton koodi tai muu sisältö	• Tarpeeton koodin kommentointi • Duplikaattikoodi • Käyttämätön tai vanhentunut koodi • Harvoin käytetty luokka
Liialliset liitokset	• Metodi, joka käyttää enemmän muun, kuin oman olionsa dataa • Luokka, joka käyttää toisen luokan sisäisiä muuttujia tai metodeja • Metodien liiallinen ketjutus

Paisuneella koodilla tarkoitetaan yleisesti koodia, jota ei ole pilkottu tarpeeksi pieniin palasiin. Esimerkiksi yksittäinen funktio tai luokka voi olla niin suuri, että sen koko aiheuttaa haasteita koodin ylläpidon näkökulmasta. Ohjelmointiparadigman väärinkäytöllä tarkoitetaan toimintatapaa, jossa paradigman tarjoamia ominaisuuksia tai ominaispiirteitä ei hyödynnetä oikealla tavalla, tai niitä hyödynnetään puutteellisesti. Refactoring Guru (2023) mainitsee verkkosivuillaan nimenomaan olio-ohjelmointi paradigmaan liittyviä väärinkäyttöesimerkkejä. Esimerkkinä olio-ohjelmointi paradigman tarjoamasta perintäominaisuuden väärinkäytöstä voidaan pitää tilannetta, jossa alaluokka (eng: child class) perii pääluokalta (eng: parent class) joitain ominaisuuksia, mutta perintä on toteutettu virheellisesti, sillä alaluokalla ja yläluokalla on täysin toisistaan riippumaton tarkoitusperä.

Koodissa voidaan huomata olevan muutoksen estäjiä silloin, kun yksittäisen muutoksen tekeminen vaatii muutoksen myös sellaiseen osaan koodia, jonka voidaan katsoa olevan yhteen kuulumatonta alkuperäiseen muutokseen liittyen. Refactoring Guru (2023) antaa esimerkkinä tilanteen, jossa uuden alaluokan periyttäminen jo olemassa olevasta pääluokasta vaatii myös toisen alaluokan periyttämisen jostain toisesta pääluokasta (rinnakkaiset perintähierarkiat, taulukko 1).

Koodi voi sisältää tarpeettomia asioita, kuten vanhentunutta, useaan kertaan kirjoitettua tai käyttämätöntä koodia. Tämän lisäksi Refactoring Guru (2023) pitää myös liiallista dokumentointia tarpeettomana, ja ohjeistaakin parhaan dokumentoinnin olevan kuvaava nimi luokalle tai funktiolle. Liiallisilla liitoksilla tarkoitetaan esimerkiksi tilanteita, joissa luokkia ei ole kapseloitu tarpeeksi. Kyseinen tilanne voi aiheuttaa sen, että luokkien yksityisiksi (eng: private) tarkoitettuja funktioita kutsutaan tarpeettomasti muista luokista.

Koodihajut ovat pohjimmiltaan peräisin hyvien ohjelmistosuunnitteluperiaatteiden, kuten ymmärrettävyys, käytettävyys ja modulaarisuus, poikkeamista. Vaikka osasyynä suunnitteluperiaatteiden laiminlyönnille voi olla kokematon ohjelmistokehittäjä, tai kehittäjätiimi, monesti suunnitteluperiaatteista luovutaan tilanteissa, jossa ulkopuolisen paineen aiheuttamana ohjelmiston kehityksessä keskitytään liikaa uusien toiminnallisuuksien lisäämiseen (Rasool & Arshad, 2015). Toimenpide koodihajujen poistoon on koodin muuttaminen niiltä osin, missä suunnitteluperiaatteista on poikettu, eli refaktorointi.

Koodihajujen poiston hyödyllisyydestä on esitetty myös eriäviä mielipiteitä. Ensinnäkin voidaan kysyä, kuinka pitkä on liian pitkä funktio tai luokka? Cinnéiden ja muiden (2016) mukaan mielipiteet siitä, mikä on liian pitkä, vaihtelevat. He esittävätkin, että termiä liian pitkä ei pidä ajatella pelkästään absoluuttisena lukuarvona. Hall ja muut (2014) tutkivat koodihajujen vaikutusta ohjelmistovirheisiin Eclipse, ArgoUML ja Apache Commons -

ohjelmistoprojekteissa. He huomasivat, että Switch-lausekkeen käytöllä ei ollut tilastollisesti merkittävää vaikutusta ohjelmistovirheiden määrään. Tämän lisäksi paisuneen koodin kategoriaan (taulukko 1) kuuluvien tietojoukkojen huomattiin jopa vähentävän ohjelmistovirheitä Eclipse- sekä Apache Commons -projekteissa.

2.2 Tekninen velka

Tekninen velka tarkoittaa velkaa, joka kertyy muun muassa hyviä ohjelmistokehitystapoja laiminlyöden. Yleinen tapa ottaa teknistä velkaa on priorisoida ohjelmiston tai jonkin sen osan valmiiksi saamista aikataulun näkökulmasta. Ohjelmisto, tai jokin sen osa, saatetaan toteuttaa siten, että ohjelmisto itsessään toimii, mutta koodin laatuun ei ole kiinnitetty tarpeeksi huomiota. Päätös teknisen velan ottamisesta voi olla tarkoituksellinen ja suunnitelmana voi olla maksaa velka myöhemmin takaisin (Pina et al., 2022). Teknistä velkaa maksetaan takaisin pääsääntöisesti refaktoroimalla. Teknisen velan ottaminen saattaa tuoda lyhyen tähtäimen hyötyjä, kuten ajan ja vaivan säästymistä. Liiallinen velkaantuminen saattaa kuitenkin johtaa pitkällä tähtäimellä ongelmiin, kuten uusien toiminnallisuuksien kehittämisen hidastumiseen. Lisäksi koodin ylläpitäminen voi käydä työlääksi, mikäli tekninen velkataakka on suuri (Pina et al., 2022).

Teknisen velan voidaan ajatella kasvavan koodihajujen lisääntyessä. Teknistä velkaa ei tule kuitenkaan ajatella ainoastaan joukkona koodihajuja, vaan siihen vaikuttavat ohjelmistokoodin lisäksi myös muut tekijät. Refactoring Guru (2023) listaa teknisen velan aiheuttajiksi koodihajuiksi määritettävissä olevien asioiden lisäksi esimeriksi automaattisen testauksen ja dokumentoimisen puuttumisen. Lisäksi Refactoring Guru (2023) mainitsee teknisen velan aiheuttajiksi kehitystiimin yhteistyöhaasteet. Esimerkkejä yhteistyöhaasteista voivat olla riittämätön kommunikaatio kehittäjien välillä tai tilanteet, joissa junior-tason kehittäjät eivät saa tarpeeksi perehdytystä ohjelmistoprojektiin senior-tason kehittäjiltä.

Tekninen velka voi kasvaa korkoa, jos sitä ei makseta ajoissa pois (Pina et al., 2022). Koron kasvaminen voi tapahtua esimerkiksi tilanteessa, jossa paljon teknistä velkaa sisältävään koodiin rakennetaan uutta koodia siten, että uusi koodi on riippuvainen teknistä velkaa sisältävästä koodista. Tällaisessa tilanteessa uutta koodia voi olla haastavaa tai jopa mahdotonta kirjoittaa ilman, että otetaan lisää velkaa. Pina ja muut (2022) suosittelee teknisen velan poistamiseksi velkajärjestelyä (eng: Technical debt management process), jossa kartoitetaan ensin velkaantumistilanne, eli kuinka paljon velkaa on. Tämän jälkeen suunnitellaan, miten velka, tai esimerkiksi kriittisin osa siitä, maksetaan pois.

Teknisen velan hallinnassa tavoitteena ei ole maksaa velkaa kokonaan pois. Tämä tarkoittaa sitä, että teknistä velkaa ei tule ajatella ensisijaisesti pahana asiana. Teknisen velan ottaminen on monesti välttämätöntä liiketoimintarealiteetit huomioiden. Teknistä velkaa voidaan verrata taloudelliseen velkaan. Mikäli taloudellinen velkataakka on hallinnassa ja velkarahan käyttämiselle on lisäarvoa tuottava syy, on velan olemassaolo hyväksyttävää. Sama ilmiö pätee tekniseen velkaan. Teknisen velan ottamisella voidaan saavuttaa liiketoimintakriittinen tavoite, kuten kiinnostusta herättävän prototyypin tai demoversion valmistuminen sopivaan aikaikkunaan. Oleellista on, että yrityksessä ymmärretään teknisen velan käsite sekä liiketoimintaorientoituneiden henkilöiden että teknisten henkilöiden välillä. Lim ja muut (2012) määrittelevätkin teknisen velan hallinnan olevan pohjimmiltaan tasapainottelua liiketoimintarealiteettien ja ohjelmiston teknisen laadun välillä.

2.3 Refaktorointiin liittyviä ongelmia

Refaktorointi vie aikaa ja refaktoroinnilla saavutettu tulos ei ole lähtökohtaisesti käyttäjälle näkyvää, eli sen avulla ohjelmistoon ei ilmesty esimerkiksi uusia ominaisuuksia. Tästä syystä esimerkiksi asiakkaat tai ohjelmistoprojektiin liittyvät muut ei-tekniset sidosryhmät eivät välttämättä ymmärrä refaktoroinnin tarpeellisuutta. Yksi useasti kysyttyjä kysymyksiä refaktorointiin liittyen onkin, miten projektista vastaavaa tahoa, kuten projektipäällikköä, tulisi lähestyä, kun halutaan kysyä lupaa ajan käytölle refaktorointia varten (Fowler, 2018).

Fowlerin (2018) mukaan ohjelmistokehittäjän ammattitaitoon kuuluu laadukkaan ohjelmiston tuottaminen niin nopeasti, kuin se on mahdollista. Refaktorointia voidaan Fowlerin (2018) mukaan pitää osana ohjelmistokehitysprosessia ja näin ollen refaktorointiin ei pitäisi lähtökohtaisesti kysyä lupaa. Luvan kysyminen voi olla ongelmallista erityisesti tilanteessa, jossa projektista vastaa hyvistä ohjelmistokehitystavoista tietämätön taho. Ideaalitilanteessa refaktorointia tulisi tehdä ohjelmistoprojektissa jatkuvasti, eikä uusia ominaisuuksia tulisi toimittaa ennen kuin ominaisuuksien liittyvä koodi on siistitty. Tällaisessa tilanteessa myös pienennetään todennäköisyyttä teknisen velan kasvulle.

Refaktorointi ei poista riskiä siitä, etteikö jokin toimiva ominaisuus voisi rikkoutua. Uuden ohjelmistovirheen ilmentymistä ei voida perustella sillä, että koodista on tehty esimerkiksi luettavampaa tai rakenteeltaan järkevämpää. Modernissa ohjelmistokehityksessä toimivaa ohjelmistoa tulisi ajatella tärkeimpänä mittarina (Agile Manifesto, 2001). Riskiä siitä, että refaktoroinnissa rikotaan jotain, voidaan pienentää muun muassa riittävällä automaatiotestauksella (Fowler, 2018). Puutteellinen tai kokonaan uupuva automaatiotestaus ei tarkoita sitä, etteikö refaktorointia voisi suorittaa. Moni kehitysympäristö, kuten

IntelliJ (2023) tai Visual Studio (2023), tarjoavat automaattisia refaktorointityökaluja, joilla voidaan tehdä refaktorointitoimenpiteitä. Työkalujen tarjoamat refaktorointikeinot ovat kuitenkin rajallisia. Esimerkiksi muuttujan tai metodin nimi voidaan työkalujen avulla muuttaa turvallisesti siten, että muutos tehdään kaikkialle koodiin, jossa kyseistä muuttujaa tai metodia käytetään. Suuremmat arkkitehtuurilliset refaktorointityöt vaativat tekijältä kuitenkin syvällisempää ymmärrystä ohjelman toiminnasta, kuten esimerkiksi sen liiketoimintasäännöistä (eng: business rules).

3. UUDELLEENKIRJOITUS

Ohjelmiston uudelleenkirjoittamisella (eng: software rewrite) tarkoitetaan ohjelmistoprojektin koodipohjasta luopumista ja projektin aloittamista puhtaalta pöydältä. Uudelleenkirjoitus-ratkaisuun saatetaan päätyä esimerkiksi tilanteessa, jossa ohjelmistoprojekti on teknisesti niin velkaantunut, että uudelleenkirjoittamisen on arvioitu olevan nopeampi tapa saada koodi ylläpidettäväksi, kuin maksaa velka pois. Toinen esimerkki on tilanne, jossa halutaan saavuttaa tavoite, johon nykyinen ohjelmointikieli tai teknologiavalinta ei anna mahdollisuuksia. Esimerkki tällaisesta tavoitteesta voi olla työpöytäkäyttö sovelluksesta (eng: desktop application) siirtyminen verkkoselaimella tai älypuhelimella toimivaan versioon. Syy voi olla myös organisaatiolähtöinen: esimerkiksi tilanne, jossa yrityskaupan toteutumisen jälkeen ostava yritys haluaa yhtenäistää ohjelmistokehityksessä käytettävän ohjelmointikielen.

Päätöstä ohjelmiston uudelleenkirjoitukseen ei tule tehdä kevein perustein. Birchall (2016) näkee uudelleenkirjoituksen viimeisenä vaihtoehtona, jota tulisi harkita, kun kaikki muut vaihtoehdot ovat käyty läpi. Spolskyn (2000) mielestä uudelleenkirjoitus on yrityksen pahin mahdollinen strateginen virhe. Aloite ohjelmiston uudelleenkehitystarpeelle lähtee monesti ohjelmistokehittäjistä. Ohjelmistokehittäjät saattavat tehdä huomioita ohjelmistoprojektin koodin laadun kelvottomuudesta, jonka pohjalta todetaan refaktorointikeinojen olevan liian työläitä suoritettavaksi. Syynä voi myös olla halu kokeilla jotain uutta ohjelmointikieltä tai ohjelmistokehystä (eng: Software framework). Päätöstä ei kuitenkaan tule tehdä vain teknisestä näkökulmasta ajateltuna, sillä päätöksellä on laaja-alainen vaikutus yrityksen liiketoimintaan (Plybon, 2019).

3.1 Uudelleenkirjoitusprojektiin ryhtyminen

Ennen, kuin ohjelmiston uudelleenkirjoitukseen ryhdytään, on syytä selvittää uudelleenkirjoitustarpeeseen johtaneet syyt ja ymmärtää, mihin ongelmaan tai ongelmiin uudelleenkirjoituksella haetaan ratkaisua. Tarve uudelleenkirjoitukselle voi olla oire sille, että yrityksessä tai ohjelmistokehitystiimissä on jotain vialla. Plybon (2019) mainitsee esimerkiksi mahdollisesta ongelmista tilanteen, joissa kehitystiimillä ei ole tapana katselmoida tiimin sisäisesti koodia. Ongelmat eivät välttämättä liity kuitenkaan ainoastaan kehitystiimiin. On mahdollista, että yrityksessä on tapana tehdä kehitettävien ominaisuuksien osalta ristiriitaisia päätöksiä, tai laiminlyödä esimerkiksi vaatimusmäärittelyjä. Tällaisissa tilanteissa ohjelman uudelleenkirjoitus voi poistaa oireen, mutta mikäli organisaatiossa

syvemmällä olevia ongelmia ei juurisyitä myöden korjata, on mahdollista, että uudelleenkirjoitettu ohjelma vanhenee ja altistuu tekniselle velalle nopeasti.

Ohjelmistoprojektit ovat tunnettuja siitä, että niiden aikataulut viivästyvät usein. Tämä pätee Plybonin (2019) mukaan myös uudelleenkirjoitusprojekteihin. Plybonin (2019) mukaan ihmiset saattavat ajatella, että uudelleenkirjoitusprojekti vie vähemmän aikaa, sillä ohjelmistohan on jo kertaalleen kirjoitettu. Tämä ei kuitenkaan lähtökohtaisesti pidä paikkaansa. Syy ohjelmiston uudelleenkirjoittamiseen on nykyisen ohjelmiston epäonnistuminen. Yrityksessä on siis tärkeää ymmärtää, että ohjelmiston uudelleenkirjoitus tulee viemään paljon aikaa ja resursseja.

Tärkeä huomioitava asia on myös se, että vanhan ohjelmiston ylläpitoa ei voida lopettaa samalla, kun uuden uudelleenkirjoitusprojektia aloitetaan. Tämä tarkoittaa sitä, että kehittäjät joutuvat keskittymään kahteen ohjelmistoprojektiin samaan aikaan (Plybon, 2019). Yrityksessä on tärkeä suunnitella ja päättää, miten kehitystiimi jaetaan projektien kesken. Myös mahdollisen lisäresurssien hankintamahdollisuus on hyvä selvittää budjetin puolesta. Tämän lisäksi on syytä tiedostaa riski kehittäjien työmoraalin alentumiselle (De Pauw, 2022). Kehittäjä saattaa haluta siirtyä uudelleenkirjoitusprojektiin, mutta on mahdollista, että yritys katsoo hänen tietotaitonsa sopivan paremmin työskentelemään vanhan projektin parissa. Vanhassa projektissa työskentely voi olla työmoraalia alentavaa, sillä työ on monesti ylläpitotyötä, kuten ohjelmistovirheiden korjaamista. Pahimmillaan tilanne voikin johtaa kehittäjien irtisanoutumiseen (De Pauw, 2022).

Vanha ohjelmisto tulee elämään uuden ohjelmiston rinnalla niin kauan, kunnes vanhalla ohjelmistolla ei ole enää aktiivisia käyttäjiä. Uudelleenkirjoitettu ohjelmisto voi vaatia toimenpiteitä käyttäjältä, kuten rekisteröitymisen ja uuden järjestelmän käytön opettelun. Jotkin käyttäjät saattavat nähdä vanhan ohjelmiston tarpeeksi hyvänä, eivätkä halua priorisoida uudelle alustalle siirtymistä. Tämä tarkoittaa sitä, että vanha ohjelmisto aiheuttaa infrastruktuurikustannuksia, kuten palvelinkustannuksia, pitkään, vaikka uudelleenkirjoitettu ohjelmisto olisikin saatu jo tuotantokäyttöön (De Pauw, 2022).

Mikäli uudelleenkirjoitus päätetään toteuttaa, tulee sen laajuus (eng: scope) määrittää tarkasti. Birchallin (2016) mukaan uudelleenkirjoitustapauksissa valitaan yleensä yksi kolmesta uudelleenkirjoitusmenetelmästä. Black-box rewrite -menetelmässä uudelleenkirjoituksen tarkoituksena on pitää ohjelmiston toiminnallisuus nykyisellään, mutta uudelleenkirjoittaa ohjelman sisäinen osuus siten, että ideaalitilanteessa loppukäyttäjä ei huomaisi minkään muuttuneen. Brush-up rewrite -menetelmässä saatetaan päivittää myös ohjelman käyttäjälle näkyviä toiminnallisuuksia, jotta ohjelmiston laatu samalla paranisi.

Quid pro quo rewrite -menetelmässä uudelleenkirjoituksen lisäksi tavoite on samalla kehittää uusia suuremman kokoluokan ominaisuuksia. Quid pro quo -vaihtoehtoa voi olla mahdollista perustella yrityksen liiketoimintaihmisille helpommin, jotta ajatus ohjelmiston uudelleenkirjoittamisesta saadaan myytyä päättäjille.

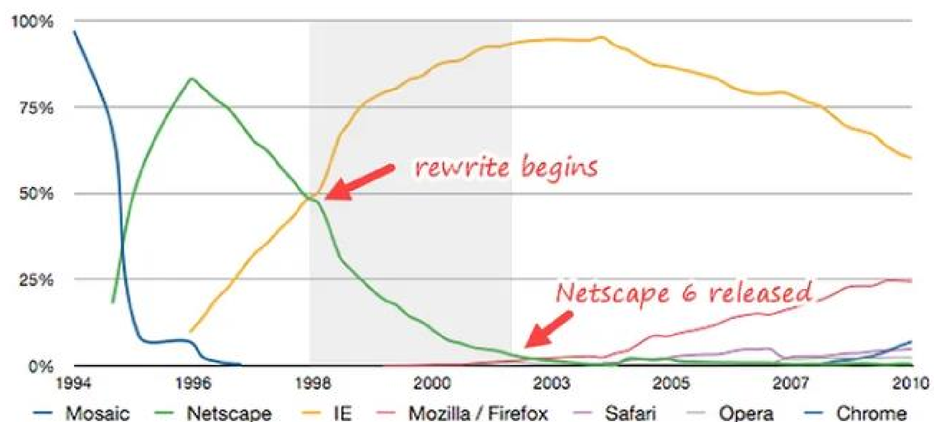
3.2 Esimerkkejä uudelleenkirjoitusprojekteista

3.2.1 Netscape

Eräs kuuluisa esimerkki epäonnistuneesta uudelleenkirjoitusprojektista on Netscape-verkkoselaimen versio 6.0. Ennen Internet Explorerin syntyä, Netscapella oli käytännössä täysi monopoli verkkoselainmarkkinassa. Netscapen ensimmäisenä kilpailijana voidaankin ajatella vuonna 1996 julkaistua Internet Exploreria. Vuonna 1998 Microsoft alkoi toimittamaan Internet Exploreria Windows -käyttöjärjestelmän mukana oletusverkkoselaimena, jonka seurauksena Internet Explorerin markkinaosuus ohitti Netscapen (Caudill, 2019).

Vuonna 1998 Netscapesta oli julkaistu versio 4.0. Versiota 5.0 lähdettiin toteuttamaan hyödyntäen osittain avoimen lähdekoodin (eng: open source) menetelmää, joka oli ennenkuulumatonta kyseisenä aikana (Kornblum, 1998). Projekti kuitenkin epäonnistui, eikä versiota 5.0 koskaan julkaistu. Versiota 5.0 kehitettiin noin yhden vuoden ajan, jonka jälkeen projekti hylättiin ja version 6.0 kehitys aloitettiin täysin puhtaalta pöydältä (Caudill, 2019).

Netscape 6.0 julkaistiin lopulta marraskuussa 2000. Uudelleenkirjoituksen aloittamisesta oli kulunut lähes kolme vuotta. Tänä aikana Netscapen markkinaosuus oli pudonnut alle kymmeneen prosenttiin, siinä missä Internet Explorerin markkinaosuus jatkoi kasvamistaan (kuva 1).



Kuva 1. Verkkoselainten markkinaosuus vuosina 1994 – 2010 (Caudill, 2019).

Uudelleenkirjoituksen jälkeen itse ohjelmistossa oli paljon puutteita. Poguen (2000) mukaan sovelluksen käynnistäminen vei aikaa lähes minuutin verran. Lisäksi sovellus käytti kohtuuttomasti käyttömuistia (eng: Random-access memory, RAM) ja oli Internet Exploreria jäljessä muun muassa käytettävyyden näkökulmasta. Ohjelmisto tarjosi hyödyllisiä ominaisuuksia, kuten lomakkeiden ja salasanakenttien automaattisen täytön ennalta tallennetun tiedon perusteella. Perusominaisuuksien osalta ohjelmisto oli kuitenkin puutteellinen. Esimerkiksi verkko-osoitteen korostamiseksi (eng: highlight) ei riittänyt klikkaus verkko-osoitekenttään, vaan koko osoite tuli korostaa hiirellä raahaamalla vasemmalta oikealle.

Vaikka Netscape-projekti on tutkielmaan verraten vanha, löytyy siitä näkökulmia, jotka pätevät myös nykypäivänä tehtäviin uudelleenkirjoitusprojekteihin. Netscape-projektista voidaan ottaa oppia siitä, miten tärkeä merkitys projektin ajoituksella on projektin onnistumisen kannalta. Uudelleenkirjoitusprojekti vie usein suuren osan yrityksen ohjelmistokehityskaistasta, jolloin tuotannossa oleva ohjelmisto ei kehity, tai kehittyy hitaasti. Tällaisessa tilanteessa kilpailijoilla on etulyöntiasema, sillä heillä voi olla mahdollisuus tuoda ohjelmistoihinsa uusia ominaisuuksia nopeammin, kuin uudelleenkirjoitusta tekevä yritys. Tilanne voikin houkutella käyttäjiä, eli maksavia asiakkaita, siirtymään käyttämään kilpailevien yritysten ohjelmistoja.

On mahdotonta sanoa, että olisiko Netscapen epäonnistumisen voinut välttää esimerkiksi päätöksellä tehdä lievempiä toimenpiteitä, kuten refaktoroida. Netscapen tarinaa voidaan kuitenkin pitää varoittavana esimerkkinä uudelleenkirjoituksen riskeistä. Spolsky (2000) mainitsee artikkelissaan uudelleenkirjoittamisen olevan missä tahansa tilanteessa yritykselle ”yksi pahin strateginen virhe”, käyttäen malliesimerkkinä Netscape 6:0:n tarinaa.

3.2.2 Remesh

Remesh on vuonna 2014 perustettu ohjelmistoalan yritys, jonka tuote on Remesh-niminen alusta. Remeshin käyttäjät voivat hyödyntää alustaa muun muassa erilaisten kaupallisten tutkimusten, kuten markkinatutkimusten, tekemiseen. Alusta poimii tutkimuksiin osallistuneiden henkilöiden vastauksista avainsanoja, sekä kategorisoi näistä oleellisia oivalluksia (eng: insights) hyödyntäen muun muassa koneoppimista (eng: machine learning).

Vuonna 2019 Remesh päätyi uudelleenkirjoittamaan ohjelmistotuotteensa. Tietz-Sokolsky (2020) avaa uudelleenkirjoituspäätökseen johtaneita syitä blogikirjoituksessaan.

Tietz-Sokolsky (2020) kertoi lukeneensa Spolskyn (2000) varoittavan artikkelin uudelleenkirjoitukseen liittyen, mutta päätti tästä huolimatta antaa äänensä uudelleenkirjoitukselle. Tiez-Sokolskyn (2020) mukaan Remesh päätti muuttaa tuotteensa sisimmäistä käyttötapausta (eng: use case) etsiessään tuotteellensa jalansijaa markkinasta. Tämän päätöksen seurauksena ohjelmiston tietyt tekniset suunnittelupäätökset (eng: design decisions) eivät enää tukeneet uutta sisimmäistä käyttötapausta parhaalla mahdollisella tavalla esimerkiksi tietokantaskeeman ja tietomallien osalta. Ohjelmisto sisälsi myös teknistä velkaa: esimerkiksi testikattavuus (eng: test coverage) oli heikkoa refaktorointia kaipaavilla alueilla. Tämän lisäksi projektissa käytettiin ohjelmointikieliä ja ohjelmointikehyksiä, joista suurimmalla osalla kehitystiimiä oli hyvin vähän kokemusta.

Remeshin uudelleenkirjoitus alkoi suunnittelulla, johon liittyi muun muassa kartoitus niistä ominaisuuksista, jotka uudelleenkirjoitettuun ohjelmistoon tulisi sisällyttää. Minimituotteen, eli MVP:n (eng: minimum viable product), määrittäminen tehtiin ennen varsinaisen koodaamistyön aloittamista. Tietz-Sokolskyn (2020) mukaan MVP:n määrittämiseen auttoi vanhan tuotteen perinpohjainen tuntemus ja uuden tuotteen selkeä visio. Edellä mainitut asiat auttoivat päättämään, mitkä ominaisuudet pystyttiin jättämään uudesta tuotteesta pois. Remeshillä ei ollut tarkoitus asennoitua uudelleenkirjoitukseen, kuten Netscape 6.0 -projektissa, jossa käytännössä joka ikinen ominaisuus tulisi säilyttää myös uudelleenkirjoitetussa versiossa. Tietz-Sokolsky (2020) piti tärkeänä onnistumiseen johtaneena tekijänä myös säännöllisiä projektin sisäisille ja ulkoisille sidosryhmille pidettyjä demoesityksiä, joista kehitystiimi sai tärkeätä palautetta.

Projektin alkuvaiheissa haasteiksi koettiin teknologiapinon (eng: technology stack) vaihtaminen. Esimerkiksi backend päätettiin toteuttaa Django-ohjelmistokehyksellä, mutta haasteena oli, että osalla kehitystiimistä ei ollut Python-ohjelmointikielestä erityisesti kokemusta. Teknisten haasteiden lisäksi Tietz-Sokolsky (2020) nostaa myös esiin kehitystiimin ajoittaiset motivaatioon liittyvät haasteet. Hänen mukaansa uudelleenkirjoitusprojekteissa alkuhuuma tuottaa innostusta tiimin sisäisesti, koska asioita pääsee tekemään puhtaalta pöydältä käyttäen uusia työkaluja. Innostus voi kuitenkin laskea nopeasti, koska suurin osa projektia on käytännössä jo kertaalleen tehtyjen ominaisuuksien uudelleenkirjoittamista, sekä jo kertaalleen korjattujen ohjelmistovirheiden korjausta.

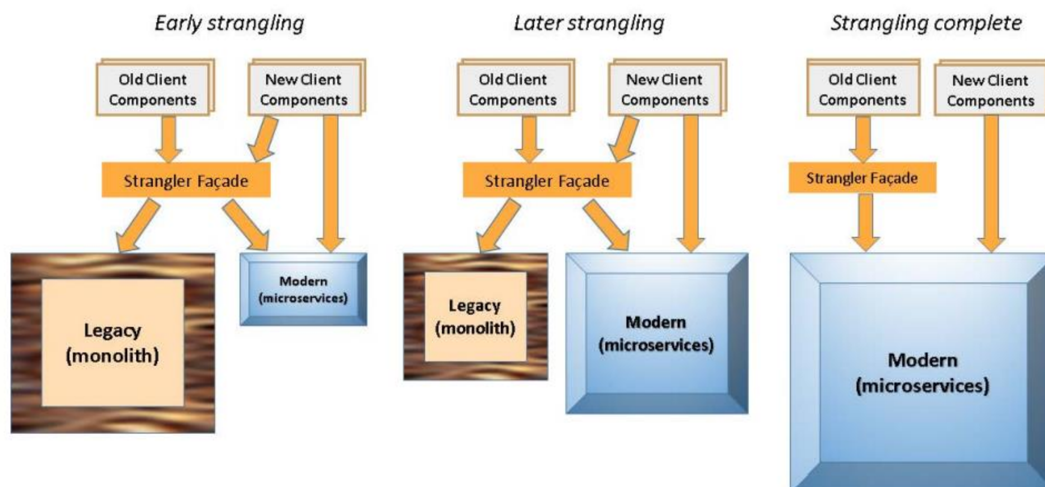
Uudelleenkirjoitettu ohjelmisto johti uuden tekoälyyn pohjautuvan tutkimusohjelmistoalustan julkaisuun (Prnewswire.com, 2020). Yritys on heidän verkkosivujensa mukaan (Remesh.ai, 2023) tänä päivänä erilaisia tunnustuksia ja palkintoja kerännyt yritys. Tietenkään ei voida varmasti sanoa, että onko vuosina 2019-2020 toteutettu uudelleenkirjoitusprojekti syy yrityksen menestykselle, mutta ainakin Tietz-Sokolsky (2020) pitää projektia onnistuneena. Tietz-Sokolsky (2020) on kuitenkin sitä mieltä, että oletuspäätös

uudelleenkirjoitusprojektiin ryhtymiseen tulisi olla kielteinen. Hänen mukaansa uudelleenkirjoitusprojektia voidaan harkita, mikäli huomataan, että ohjelmiston arkkitehtuuri ei enää korreloi liiketoiminnan tarpeiden kanssa. Korreloimattomuus ohjelmiston ja liiketoiminnan välillä voi näyttäytyä esimerkiksi uusien ominaisuuksien toteuttamisen hidastumisena.

3.3 Kuristusmenetelmä

Ohjelmiston täydessä uudelleenkirjoitusprojektissa (eng: full rewrite) tavoitteena on vanhan ohjelmiston korvaaminen uudella uudelleenkirjoituksen valmistuttua. Tämä aiheuttaa muun muassa kohdassa 3.1 mainitun ongelman, jossa vanhaa ohjelmistoa tulee ylläpitää niin kauan, kunnes käyttäjät ovat luopuneet siitä siirtymällä uuteen ohjelmistoon. Eräs kompromissiratkaisu refaktoroinnin ja täyden uudelleenkirjoituksen välillä on käyttää niin kutsuttua kuristusmenetelmää (eng: strangling). Fowler (2004) määritti aikoinaan termin ”Strangler Application” kuvaamaan ohjelmiston vaiheittaista uudelleenkirjoittamista, jossa vanha ohjelmisto korvataan uudella siten, että vanha ohjelmisto pysyy toiminnassa uuden ohjelmiston kehityksen ajan.

Kuristusmenetelmää voidaan käyttää esimerkiksi tilanteessa, jossa monoliittinen ohjelmisto korvataan mikropalveluarkkitehtuurilla. Käytännössä tämä tarkoittaa suuren, vaikeasti ylläpidettävän ja todennäköisesti paljon teknistä velkaa sisältävän ohjelmiston pilkkomista pienempiin löyhästi kytkettyihin (eng: loosely coupled) mikropalveluihin. Tyypillisesti mikropalveluiden on tarkoitus korvata monoliittinen ohjelmisto kokonaisuudessaan. Kuristusmenetelmää voidaan soveltaa myös niin, että esimerkiksi vaikeasti korvattavissa oleva osa monoliitista jätetään korvaamatta ja monoliitti jatkaa elämistään uusien mikropalveluiden rinnalla pienemmässä roolissa (Yoder & Merson, 2020).



Kuva 2. Esimerkki kuristusmenetelmän vaiheista (Yoder & Merson, 2020).

Kuristusmenetelmän toteuttamisen aloittamiselle on erilaisia tapoja. Eräs vaihtoehto on aloittaa toteuttaminen luomalla monoliitin ja sen asiakassovellusten tai komponenttien (eng: client) väliin julkisivu (Early strangling, kuva 2), jota voi ajatella esimerkiksi rajapintana (API). Asiakassovellusten ei tarvitse tällöin välittää siitä, välittyykö kutsu monoliittiin vai mikropalveluun, sillä julkisivu tekee päätöksen. Kutsu välitetään mikropalvelupuolelle, mikäli sille on olemassa mikropalvelupuolella tuki. Mikäli kutsua varten ei ole rakennettu sopivaa mikropalvelua, kutsu välittyy monoliittiin. Julkisivun pystyttäminen voi olla perusteltua varsinkin tilanteessa, jossa monoliittiin tulee esimerkiksi rajapintakutsuja kolmansilta osapuolilta (Yoder & Merson, 2020).

Julkisivun pystyttämisen jälkeen monoliitin komponentteja voidaan aloittaa korvaamaan rakentamalla uusia mikropalveluita. Uudet asiakassovellukset, kuten esimerkiksi uusi käyttöliittymä, ohjaavat kyselynsä joko pystytettyyn julkisivuun tai mikropalveluihin riippuen siitä, mistä haettava tieto on saatavilla (Later strangling, kuva 2). Tämän lisäksi, mikäli hankkeen aikana on tarve lisätä ohjelmistoon uusia ominaisuuksia, ne kannattaa toteuttaa lähtökohtaisesti mikropalveluina. Lähtökohtaisesti monoliittiin ei kuristusmenetelmä hankkeen aikana luoda enää uusia ominaisuuksia. Kun monoliitti on saatu korvattua mikropalveluilla (Strangling complete, kuva 2), jäljelle jää ensimmäisessä vaiheessa luotu julkisivu. Julkisivusta voidaan luopua, kun vanhat asiakassovellukset ovat jääneet tarpeettomiksi.

Kuristusmenetelmän hyöty täyden uudelleenkirjoitus menetelmään verrattuna on kuristusmenetelmän mahdollistama pienempi riski (Fowler, 2004). Kuristusmenetelmässä vanhasta ohjelmistosta siirrytään uuteen askel kerrallaan priorisoiden eniten arvoa tuottavien toiminnallisuuksien siirtämistä mikropalveluiksi ensin. Tämä mahdollistaa myös työstä aiheutuvien kustannusten jakamiseen osiin, sillä yrityksillä ei välttämättä ole taloudellista mahdollisuutta teettää kokonaisvaltaista uudelleenkirjoitushanketta yhdellä kertaa (Yoder & Merson, 2020).

Kuristusmenetelmää voidaan ajatella osittaisena uudelleenkirjoituksena tilanteessa, jossa monoliitista korvataan vain jokin osa mikropalveluina. Tämä tarkoittaa kuitenkin sitä, että monoliitti jää edelleen osittaiseen käyttöön. Ohjelmistokehityksen näkökulmasta tilanne voi kuitenkin olla epäoptimaalinen. Ensinnäkin tilanne voi laukaista kohdassa 3.1 mainitun työmoraaliin liittyvän haasteen. Kehittäjät eivät välttämättä ole motivoituneita työstämään monoliitissa olevaa koodia, kun sen rinnalle on tullut puhtaalta pöydältä toteutettuja mikropalveluja. Toiseksi monoliittiin voi jäädä suurikin määrä niin kutsuttua kuollutta koodia, eli koodia, jota ei suoriteta ohjelman ajon aikana. Kuristusmenetelmällä suoritettu osittainen uudelleenkirjoitus onkin kompromissivaihtoehto täyden uudelleenkirjoituksen ja laajamittaisen refaktoroinnin välillä.

4. PÄÄTÖKSENTEKO REFAKTOROINNIN JA UUDELLEENKIRJOITUKSEN VÄLILLÄ

Päätös refaktoroinnin ja uudelleenkirjoittamisen välillä voidaan joutua tekemään esimerkiksi tilanteessa, jossa ohjelmistotuote on päässyt velkaantumaan suuresti. Velkaantuminen voi johtua esimerkiksi siitä syystä, että refaktorointia ei ole tehty projektin aikana tarpeeksi. Ohjelmiston velkaantuminen ei tapahdu yhdessä yössä, vaan hiljalleen ajan kanssa. Kasvavan velan ensioireita ovat muun muassa jatkuva käyttäjäpalaute ohjelmiston ominaisuuksien toimimattomuudesta, lisääntynyt tarve tukipalveluille sekä uusien ominaisuuksien kehittämisen hidastuminen. Hidastuminen voi näyttäytyä ilmiönä, jossa ohjelmisto ei enää pysy liiketoiminnan tarpeiden perässä. Toisaalta päätös voidaan joutua tekemään myös esimerkiksi tilanteessa, jossa yrityksen liiketoimintastrategiaa muutetaan merkittävästi siten, että nykyinen ohjelmisto ei enää vastaa uuteen strategiaan parhaalla mahdollisella tavalla (alakohta 3.2.2).

Ohjelmistotuotteen pohjimmainen tarkoitus on ratkaista liiketoimintalähtöinen ongelma. Näin ollen on tärkeää ymmärtää, mihin ongelmaan refaktoroinnilla tai uudelleenkirjoittamisella etsitään ratkaisua. Olettaen, että ongelman ymmärtämisen jälkeen refaktorointi tai uudelleenkirjoitus ovat edelleen vaihtoehdot ongelman ratkaisemiseksi, tulisi refaktorointia pitää oletusvaihtoehtona. Näin ohjeistavat muun muassa Birchall (2016) ja Tiez-Sokolsky (2020). Refaktoroinnin pitäminen oletusvaihtoehtona ennaltaehkäisee riskiä harkitsemattomaan uudelleenkirjoitushankkeeseen lähtemiselle, mutta pitää vaihtoehdon kuitenkin pöydällä. Uudelleenkirjoitusta voidaan harkita, mikäli päätöksenteossa huomioitavat tekijät (luku 5), kuten aikataulu, budjetti ja ajoitus, puoltavat uudelleenkirjoitusta.

Fairbanks (2019) esittää, että kysymystä refaktoroinnin ja uudelleenkirjoituksen välillä voidaan lähteä pohtimaan kolmiosaisen arviointimenetelmän avulla. Ensimmäiseksi arvioidaan refaktorointiin ja uudelleenkirjoitukseen tarvittava aika. Tämän jälkeen arvioidaan ajankohta, jolloin refaktorointi tai uudelleenkirjoitus alkaa tuottamaan arvoa verrattuna tilanteeseen, jossa kumpaakaan vaihtoehtoa ei toteuteta. Kolmanneksi arvioidaan saavutettua lopputulosta refaktoroinnin ja uudelleenkirjoitusvaihtoehtojen näkökulmista. Edellä mainittujen arviointien perusteella voidaan tapauksesta riippuen saada päätös kallistumaan selkeästi suuntaan tai toiseen.

Fairbanksin (2019) mukaan on oleellista kysyä, onko juuri nyt sopiva aika tehdä refaktorointi tai uudelleenkirjoitustyö. Mikäli suuri uudelleenkirjoitushanke ei ole mahdollista taloudellisesti, aikatauluperusteisesti tai kiihtyneen kilpailun takia, voi pienempi refaktorointityö olla parempi vaihtoehto olettaen, että sillä voidaan saavuttaa jokin oleellinen parannus vaarantamatta liiketoimintaa.

Fairbanksin (2019) lähestymistavassa haasteena on kuitenkin se, että se perustuu monilta osin arvioihin. Realistinen työmäärän arvioiminen varsinkin suuremman projektin refaktoroinnista tai uudelleenkirjoittamisesta voi olla mahdoton tehtävä. Työhön tarvittava työpäivien, viikkojen, kuukausien tai jopa vuosien lukumäärä kannattaakin arvioida lukuvälinä.

Mikäli refaktorointi tai uudelleenkirjoitus eivät ole sopivia ratkaisuja käsillä olevaan ongelmaan, voidaan myös muita ratkaisuvaihtoehtoja kartoittaa. Naiivina ratkaisuna voidaan pitää ongelman sivuuttamista, eli toisin sanoen ongelmalle päätetään olla tekemättä mitään. Kuristusmenetelmä, eli monoliittisen sovelluksen pilkkominen mikropalveluiksi kokonaan tai osittain (kohta 3.3), voi joissain tapauksissa soveltua kompromissiratkaisuksi.

Tilanteesta riippuen ohjelmisto tai sen osa voidaan korvata kolmannen osapuolen ratkaisulla. On mahdollista, että yritys on toimittanut ohjelmistoratkaisua, jossa on määrältään paljon erilaisia ominaisuuksia, mutta ominaisuuksista mikään ei ole erityisen kilpailukykyinen. Ohjelmistoratkaisu on voinut tarjota esimerkiksi erilaisia itsetehtyjä tiedon visualisointi- ja analytiikkatyökaluja, mutta kyseiset toiminnallisuudet eivät ole olleet ratkaisun päätavoite eivätkä osa yrityksen ydinosaamista. Tällaisessa tapauksessa voi olla perusteltua tehdä päätös, jossa nykyinen ratkaisu uudelleen tuotteistetaan siten, että ratkaisu tarjoaa ainoastaan perinteisiä raportteja. Ohjelmisto integroidaan kolmannen osapuolen analytiikkasovellukseen tai alustaan. Näin ollen yritykselle jää aikaa keskittyä ohjelmistonsa ominaisuuksiin, jotka korreloivat paremmin yrityksen ydinosaamisen kanssa.

5. TULOKSET JA POHDINTA

Päätöksessä refaktoroinnin ja uudelleenkirjoituksen välillä tulee huomioida useita tekijöitä sekä näkökulmia. Jokainen tapaus on erilainen ja näin ollen tapauksesta riippuen joillain tekijöillä voi olla suurempi painoarvo, kuin toisilla. Taulukkoon 2 on koottu refaktoroinnin ja uudelleenkirjoituksen välillä huomioitavia tekijöitä. Kunkin tekijän kohdalla taulukkoon on listattu sekä refaktorointia että uudelleenkirjoitusta puoltavia näkökulmia.

Taulukko 2. Refaktoroinnin ja uudelleenkirjoittamisen välillä huomioitavia tekijöitä.

Päätöksentekoon vaikuttava tekijä	Refaktorointia puoltavat näkökulmat	Uudelleenkirjoitusta puoltavat näkökulmat
Tarvittavien muutosten laajuus ongelman ratkaisemiseksi	• Tarvittavat muutokset ovat pienikokoisia	• Tarvittavat muutokset ovat suuria
Teknisen velan laajuus	• Teknisen velan määrä on vähäistä, sekä velanhallintaan liittyvät prosessit ovat kunnossa	• Teknisen velan määrä on suurikokoinen
Käytettävissä oleva aika	• Työ on suoritettava mahdollisimman nopeasti	• Työ on mahdollista suorittaa laadukkaasti sille allokoidussa ajassa
Työn ajoitus	• Ajankohta on huono esimerkiksi markkinatilanteen näkökulmasta	• Ajankohta sallii uudellenkirjoitustyön
Työn resursointi	• Niukka resursointimahdollisuus	• Resurseja on käytettävissä riittävästi
Henkilöresurssien tekninen kompetenssi	• Tekninen kompetenssi on riittävä refaktoroinnin suorittamiseksi	• Käytettävissä olevilla resureilla ei ole tarpeeksi teknistä ymmärrystä järjestelmästä

Ratkaistavan ongelman tulisi olla päätöksenteon keskiössä. Mikäli ratkaistava ongelma ymmärretään, on helpompi lähteä arvioimaan tarvittavien muutosten laajuutta refaktoroinnin sekä uudelleenkirjoituksen näkökulmasta. Mikäli arvio työmäärästä tarvittavien muutoksen osalta on pienehkö, eikä ongelman ratkaiseminen perusteellisesti vaadi radikaaleja toimenpiteitä, kuten teknologiapinon vaihtamista, voi refaktorointi olla oikea vaihtoehto.

Ongelman ratkaiseminen refaktoroimalla voi kuitenkin vaatia suuren työn. Tämä voi johtua esimerkiksi radikaalisti muuttuneista ohjelmiston käyttötarpeesta, jotka eivät enää korreloi ohjelman alkuperäisen suunnitteluperiaatteiden kanssa (alakohta 3.2.2). On myös mahdollista, että ongelma olisi helpommin ratkaistavissa esimerkiksi siirtymällä johonkin toiseen ohjelmointikieleen tai vaihtamalla projektin teknologiapino kokonaan. Tällaisissa tapauksissa ratkaistavan ongelman näkökulmasta voi uudelleenkirjoitus olla parempi vaihtoehto.

Tekninen velka on yksi tärkeä huomioitava tekijä päätöksenteossa. Päätös voi siirtyä uudelleenkirjoituksen puolelle, mikäli huomataan, että teknisen velan määrä ei ole enää hallinnassa. Vaikka uudelleenkirjoitukseen päädyttäisin suuren tai hallitsemattoman teknisen velan takia, tulisi velan aiheuttaneet juurisyyt selvittää, jotta sama ilmiö ei toistu uudelleenkirjoitusprojektissa.

Sekä refaktorointi että uudelleenkirjoitus vievät aikaa. Ajankäyttöä voidaan pohtia sekä tarvittavan ajan näkökulmasta, että ajoituksen näkökulmasta. Mikäli työtä pidetään aikakriittisenä, mutta sille ei ole kuitenkaan mahdollista allokoida esimerkiksi uudelleenkirjoitukseen tarvittavaa määrää aikaa, voi refaktorointi olla parempi vaihtoehto. Refaktorointiin tai uudelleenkirjoitukseen käytetty aika on lähtökohtaisesti pois jonkun muun työn suorittamisesta. Etenkin uudelleenkirjoitusta harkitessa tuleekin kysyä, onko juuri nyt sopiva hetki lähteä tekemään uudelleenkirjoitusta. Uudelleenkirjoitus sitoo resursseja pitkäksi aikaa. Projekti ei myöskään tuota arvoa esimerkiksi potentiaalisille uusille käyttäjille, kun vasta sen valmistumisen jälkeen. Työn ajoitusta tuleekin pohtia erityisesti liiketoiminnan ja markkinatilanteen näkökulmasta.

Työn resursointimahdollisuuksia tulisi myös ajatella tärkeänä tekijänä. Refaktorointia voidaan tilanteesta riippuen suorittaa pala kerrallaan ajan kanssa. Uudelleenkirjoitusprojekti vaatii usein monia ja monenlaisia resursseja, jonka takia refaktorointi voi olla houkuttelevampi vaihtoehto. Toisaalta tulee huomioida myös käytettävissä olevien henkilöresurssien tekninen kyvykkyys. Refaktorointia suorittavan henkilö on tärkeä ymmärtää nykyi-

nen ohjelmisto ja sen haaste, erityisesti tilanteessa, jossa ohjelmisto sisältää paljon teknistä velkaa. Mikäli ohjelmisto on teknisesti velkaantunut, eikä sen tuntevia kehittäjiä ole enää käytössä, voi päätös kallistua uudelleenkirjoituksen puolelle.

On tärkeä huomioida, että taulukko ei ole yleispätevä jokaiseen tilanteeseen. Taulukkoa voidaan kuitenkin käyttää suuntaa antavana ohjenuorana. Taulukko voi toimia myös tarkistuslistana sille, että taulukossa mainittuja tekijöitä on päätöksenteko prosessin aikana huomioitu. Tilanteesta riippuen päätöksenteossa voidaan joutua huomioimaan myös tekijöitä, joita taulukkoon ei ole listattu.

6. YHTEENVETO

Ohjelmistoprojektin tekninen velkaantuminen voi laukaista tarpeen päätökselle, jossa tulee pohtia mahdollisen refaktorointityön tai ohjelmiston uudelleenkirjoittamisen väliltä. Toisaalta kysymystä saatetaan joutua käsittelemään myös esimerkiksi tilanteessa, jossa yrityksen liiketoiminnallinen strategia on muuttunut radikaalisti. Teknisen velan hallinta, jatkuva sidosryhmäkommunikaatio sekä hyvät suunnittelu- ja ohjelmointitavat voivat ennaltaehkäistä tarvetta laajamittaiselle refaktorointityölle tai uudelleenkirjoittamiselle.

Mikäli päätös refaktoroinnin ja uudelleenkirjoittamisen välillä on tarve tehdä, tulee siinä huomioida useita tekijöitä. Huomioitavia tekijöitä ovat muun muassa aikataulu, budjetti ja käytettävissä olevat resurssit työn suorittamiseen. Lisäksi tulee pohtia työn ajoittamista ja huomioida uudelleenkirjoitukseen liittyvät riskit, kuten kilpailijoiden mahdollisuuden kiriä ohi uudelleenkirjoitustyön aikana.

Refaktorointia voidaan monesti pitää oletusvaihtoehtona, mutta joissain tapauksissa uudelleenkirjoitus voi olla oikea vaihtoehto. Historiassa on esimerkkejä onnistuneista ja epäonnistuneista uudelleenkirjoitusprojektista. Epäonnistuneiden projektien osalta voi olla mahdoton sanoa, olisiko refaktorointi ollut parempi vaihtoehto. On mahdollista, että päätös uudelleenkirjoitukseen ryhtymisestä on yksinkertaisesti tehty liian myöhään.

Ohjelmistokehityksessä oleellista on pysyä ajan tasalla, niin teknisestä näkökulmasta, kuin markkinatilanteen näkökulmasta. Kehittäjien tulee ymmärtää, että kaupallinen ohjelmisto ei saavuta tavoitettaan, mikäli ohjelmiston koodi on kunnollista, mutta ohjelmisto itsessään ei ratkaise asiakkaan ongelmaa toiminnallisesta näkökulmasta. Toisaalta liiketoimintaorientoituneiden henkilöiden tulee ymmärtää, että ohjelmistoon ei voida jatkuvasti lisätä uusia ominaisuuksia ilman, että ohjelmiston tekniseen laadunparannukseen käytetään aikaa. Ohjelmistoyrityksessä onkin tärkeää tehdä yhteistyötä eri osastojen välillä. Näin toimimalla on mahdollista, että päätöstä refaktoroinnin ja uudelleenkirjoittamisen välillä ei tarvitse tehdä.

LÄHTEET

Agile Manifesto. (2001). *Principles*. Haettu 17.4.2023 osoitteesta https://agilemanifesto.org/principles.html

Birchall, C. (2016). *Re-Engineering Legacy Software*. Manning Publications.

Caudill, H. (2019). *Lessons from 6 software rewrite stories*. Medium.com. Haettu 17.4.2023 osoitteesta https://medium.com/@herbcaudill/lessons-from-6-software-rewrite-stories-635e4c8f7c22

Cinnéide, M. Ó., Yamashita, A. & Counsell, S. (2016). Measuring Refactoring Benefits: A Survey of the Evidence. *IWoR 2016: Proceedings of the 1st international Workshop on Software* Refactoring (s. 9-12). https://doi.org/10.1145/2975945.2975948

De Pauw, Y. (2022). *How to do a software rewrite.* Madewithlove. Haettu 17.4.2023 osoitteesta https://madewithlove.com/blog/leadership-and-team-management/how-to-do-a-software-rewrite/

Fairbanks, G. (2019). Ignore, Refactor, or Rewrite. *IEEE Software,* 36(2), 133-136. https://doi.org/10.1109/MS.2018.2880662

Fowler, M. (2004). *StranglerFigApplication*. MartinFowler.com. Haettu 17.4.2023 osoitteesta https://martinfowler.com/bliki/StranglerFigApplication.html

Fowler, M. (2018). *Refactoring: Improving the Design of Existing Code*. 2. painos. Addison-Wesley-Professional.

Hall, T., Zhang, M., Bowes, D. & Sun, Y. (2014). Some Code Smells Have a Significant but Small Effect on Faults. *ACM Transactions on Software Engineering and Methodology,* 23(4), 1-39. https://doi.org/10.1145/2629648

IntelliJ. (2023). *Kehitysympäristö*. Haettu 22.4.2023 osoitteesta https://www.jetbrains.com/idea/

Kornblum, J. (1998). *Netscape sets source code for free*. Cnet. Haettu 17.4.2023 osoitteesta https://www.cnet.com/tech/tech-industry/netscape-sets-source-code-free/

Lim, E., Taksande, N. & Seaman, C. (2012). A Balancing Act: What Software Practitioners Have to Say about Technical Debt. *IEEE Software,* 29(6), 22-27. https://doi.org/10.1109/MS.2012.130

Pina, D., Seaman, C. & Goldman, A. (2022). Technical debt priorization: a developer's perspective. *TechDebt '22: Proceedings of the International Conference on Technical Debt* (s. 46-55). https://doi.org/10.1145/3524843.3528096

Plybon, S. (2019). *When to do a Software Rewrite.* Method. Haettu 17.4.2023 osoitteesta https://www.method.com/insights/when-to-do-a-software-rewrite/

Pogue D. (2000). STATE OF THE ART; Netscape 6 Browser: Mixed Bag. *The New York Times.* Haettu 17.4.2023 osoitteesta https://www.proquest.com/docview/2233086850

Prnewswire. (2020). *Remesh Announces $25M Series A2; Launches New AI Product to Disrupt Research Industry*. Haettu 17.4.2023 osoitteesta https://www.prnewswire.com/news-releases/remesh-announces-25m-series-a2-launches-new-ai-product-to-disrupt-research-industry-301021597.html

Rasool, G. & Arshad, Z. (2015). A review of code smell mining techniques. *Journal of Software: Evolution and Process,* 27(11), 867-895. https://doi.org/10.1002/smr.1737

Refactoring Guru. (2023). *Refactoring*. Haettu 15.2.2023 osoitteesta https://refactoring.guru/refactoring/smells

Remesh. (2023). *About.* Haettu 17.4.2023 osoitteesta https://www.remesh.ai/about

SonarQube. (2023). *SonarQube Documentation*. Haettu 17.4.2023 osoitteesta https://docs.sonarqube.org/latest/

Spolsky, J. (2000). *Things you Should never do, Part I.* Joel on Software. Haettu 17.4.2023 osoitteesta https://www.joelonsoftware.com/2000/04/06/things-you-should-never-do-part-i/

Tietz-Sokolsky, N. (2020). *Refactor vs. Rewrite*. Remesh Engineering Blog. Haettu 17.4.2023 osoitteesta https://remesh.blog/refactor-vs-rewrite-7b260e80277a

Visual Studio. (2023). *Kehitystyökalu*. Haettu 22.4.2023 osoitteesta https://visualstudio.microsoft.com/

Yoder, J.W. & Merson, P. (2022). Strangler patterns. *PLoP '20: Proceedings of the 27th Conference on Pattern Languages of Programs.* (s. 1-25) https://dl.acm.org/doi/abs/10.5555/3511065.3511076