Koneoppimisen käyttäminen ohjelmistojen refaktoroinnissa

University of Oulu
Faculty of Information Technology and Electrical Engineering / UNIT
LuK-Tutkielma
Juho Räisänen
29.11.2021

Tiivistelmä

Refaktoroinnilla tarkoitetaan ohjelmakoodin muokkaamista ja siistimistä helpommin ymmärrettävään muotoon, muuttamatta koodin ulkoista toimintaa. Refaktoroinnin seurauksena ohjelmointityö etenee sujuvammin ja samalla vältetään projektin ajautumista ongelmiin. Refaktorointi ei kuitenkaan ole helppo toimenpide, ja epäonnistuessaan se voi aiheuttaa huomattavia kustannuksia. Tämän vuoksi suositellaan laajojen operaatioiden sijasta kevyttä refaktorointia päivittäisen ohjelmointityön ohessa. Refaktoroinnin haasteita ovat käytössä olevan ajan vähyys, monimutkaiset ongelmat, puutteellinen työkalujen tuki ja riskit toimivan koodin rikkomisesta.

Koneoppimisella tarkoitetaan tietokonejärjestelmiä, joiden toiminta kehittyy automaattisesti kokemuksen kautta. Koneoppimista voidaan soveltaa monimutkaisten ongelmien ratkaisemisessa, joihin kuuluu esimerkiksi automaattinen kasvojen tunnistaminen. Tämän tutkielman tarkoituksena on selvittää koneoppimisen käyttömahdollisuuksia automaattisessa refaktoroinnissa. Tutkielma osoitti, että koneoppimisen avulla voidaan tunnistaa lähdekoodista kohtia refaktorointia varten, mutta menetelmät eivät ole vielä valmiita ja aiheen tutkimus on yhä kesken. Työkaluilta vaaditaan paljon, sillä tehtävä on vaikea ja vääriä tuloksia ei voida hyväksyä. Manuaalisessa refaktoroinnissa tärkeää on versionhallinnan ja kattavien yksikkötestien käyttö turvallisuuden takaamiseksi. Automaattinen refaktorointi vähentäisi kuitenkin kehittäjien työtä huomattavasti.

Avainsanat
refaktorointi, lähdekoodi, code smell, koneoppiminen, syväoppiminen, tekoäly, data, algoritmi

Ohjaaja
Leevi Rantala

Sisällysluettelo

1. Johdanto

Tässä tutkielmassa tutustutaan refaktorointiin, joka kuuluu menetelmänä keskeisesti ammattimaiseen ohjelmointityöhön. Ohjelmiston refaktoroinnilla tarkoitetaan ohjelmakoodin rakenteen kehittämistä kuitenkaan muuttamatta ohjelmiston ulkoista toimintaa (Fowler, 1999). Refaktorointi on siis tavallaan koodin siistimistä tehokkailla ja hallituilla menetelmillä (Fowler, 1999). Kim et al. (2012) toteuttamassa empiirisessä tutkimuksessa suurin osa haastatelluista määritteli refaktoroinnin olevan "koodin muokkaamista, jolla kehitetään ohjelman tiettyjä ominaisuuksia, kuten ymmärrettävyyttä, huollettavuutta ja suorituskykyä". Tarkoituksena on silti säilyttää koodin toiminta ennallaan. Käytännössä refaktoroinnin yhteydessä saatetaan myös lisätä ominaisuuksia tai korjata virheitä ohjelmassa (Kim et al., 2012).

Miksi ohjelmakoodin refaktorointia tulisi toteuttaa? Fowlerin (1999) mukaan refaktoroinnin seurauksena saadaan aikaan ymmärrettävämpää koodia. Refaktorointi auttaa löytämään bugeja eli virheitä ja ohjelmoimaan jatkossa tehokkaammin. Refaktoroinnin seurauksena koodia on myös helpompi käyttää uudelleen, esimerkiksi pieneksi pilkottuja moduuleja on helpompi hyödyntää kuin yhtä suurta moduulia (Leppänen et al., 2015 b). Koodin ymmärrettävyyden tärkeyttä painottaa muun muassa seuraava kuuluisa lause: "Kuka tahansa voi kirjoittaa koodia, jota tietokone ymmärtää. Hyvät ohjelmoijat kirjoittavat koodia, jota ihmiset voivat ymmärtää" (Fowler, 1999). Refaktoroinnin laiminlyömisestä seuraa ajan kuluessa sotkuinen ohjelma, johon on hyvin vaikeaa lisätä uusia ominaisuuksia (Martin, 2019).

Ohjelmiston refaktorointiin liittyy kuitenkin myös riskejä. Ensinnäkin, koodin muokkaaminen vie yleensä oman aikansa, eikä aikaa ole yleensä tuhlattavaksi aikataulutetuissa projekteissa. Siksi ajan järjestäminen refaktorointiin koetaan haastavaksi (Sharma et al., 2015). Refaktoroinnin esteenä on usein myös pelko toimivan koodin rikkomisesta (Sharma et al., 2015). Tulisi siis löytää oikea tasapaino, jossa refaktorointi ei uhkaa merkittävästi projektin kulkua, mutta tarvittavat refaktoroinnin toimenpiteet pystytään toteuttamaan sopivassa mittasuhteessa.

Refaktorointi on haastava operaatio, joten sen toteuttamista varten on kehitetty lukuisia tietokoneohjelmia. Nämä työkalut pääsääntöisesti etsivät lähdekoodista sopivia kohtia refaktoroimista varten, eli niin sanottua "haisevaa" koodia. Tulokset kuitenkin vaihtelevat työkalujen välillä ja lisäksi työkalut ennustavat usein tarpeettomia tuloksia (Fontana & Zanoni, 2017). Kuitenkin viime vuosikymmenenä tekoälytutkimuksen keskipisteeseen noussut koneoppiminen voisi olla menetelmänä ratkaisu refaktoroinnin automatisointiin. Koneoppimisella tarkoitetaan tietokonejärjestelmiä, jotka kehittävät toimintaansa automaattisesti kokemuksen kautta (Jordan & Mitchell, 2015). Ne ovat tehokkaita ratkaisemaan ongelmia, joihin aikaisemmat käsin suunnitellut algoritmit eivät ole pystyneet. Esimerkiksi erilaiset tunnistustehtävät, kuten kasvojen, kuvien ja äänen automaattinen tunnistaminen ja luokittelu voidaan toteuttaa koneoppimisen avulla.

Tämän tutkielman tutkimusongelmana on selvittää koneoppimisen merkitystä refaktoroinnin toteuttamisessa: voidaanko koneoppimisen avulla välttää refaktorointiin liittyviä riskejä? Aluksi tutustutaan refaktoroimiseen ja siinä esiintyviin haasteisiin, jonka jälkeen perehdytään koneoppimisen käsitteisiin ja käyttömahdollisuuksiin. Tutkielmassa ei perehdytä koneoppimisessa käytettävien algoritmien yksityiskohtaiseen

toimintaan, sillä ne ansaitsevat matemaattisen monimutkaisuutensa vuoksi oman tutkimuksensa.

Seuraavaksi luvussa kaksi kerrotaan tutkimusmenetelmistä. Sen jälkeen luvussa kolme esitetään aiheen aiempi tutkimus. Tutkimus on jaettu alaotsikoihin *Refaktoroinnin toteuttaminen*, *Refaktoroinnin haasteet ja riskitekijät*, *Koneoppiminen* sekä *Koneoppimisen käyttäminen refaktoroinnissa*. Luvussa neljä pohditaan tutkimuksesta johdettuja havaintoja sekä vastataan tutkimusongelmaan. Lopuksi esitetään yhteenveto. Tämä tutkielma pohjautuu JTT-kurssilla toteuttamaani tutkielmaan *Ohjelmiston refaktoroinnin toteuttaminen turvallisesti* (2021).

2. Tutkimusmenetelmät

Tutkielma toteutetaan kirjallisuuskatsauksena aikaisempiin tutkimuksiin. Kirjallisuuskatsauksessa löydetty tutkimustieto esitetään luvussa kolme. Luvussa käytetään tieteellisiä lähteitä väitteiden tukena sekä lähteisiin viitataan APA 7-käytäntöä noudattaen. Oma pohdinta aiheesta on rajattu lukuun neljä. Tutkimusongelmana on selvittää koneoppimisen merkitystä refaktoroinnin toteuttamisessa: voidaanko koneoppimisen avulla välttää refaktorointiin liittyviä riskejä? Avustavia tutkimuskysymyksiä ovat seuraavat: “miten refaktorointia toteutetaan?” sekä “mitä haasteita ja riskejä refaktorointiin sisältyy?” ja “miten koneoppimista voidaan käyttää refaktoroinnin apuvälineenä?”.

Tutkimustietoa aiheesta etsittiin Scopus-, Google Scholar- ja IEEE Xplore-tietokannoista. Etsin aluksi yleistietoa refaktoroinnista sekä siihen liittyviä haasteita hakulauseella “software refactoring AND (risk OR challenge)”. Scopus-tietokannan haku kohdistettuna otsikkoon, tiivistelmään ja avainsanoihin tuotti 367 hakutulosta. Rajoitin seuraavaksi hakua vain Computer Science -aihealueen artikkeleihin, joissa esiintyy avainsanat Refactorings ja Refactoring: silloinkin tuloksena saatiin 176 hakutulosta. Koska hakutuloksia oli aika paljon, määräsin hakusanojen “refactoring” ja (risk OR challenge) maksimiväliksi neljä sanaa. Nyt päivitetyllä hakulauseella “software refactoring w/4 (risk OR challenge)” löytyi 38 hakutulosta, joista useat vaikuttivat tärkeiltä omaa tutkielmaa ajatellen. Etsin lisäksi yleistietoa refaktoroinnista hakulauseella “software refactoring” ja järjestin hakutulokset niiden saamien viittausten lukumäärän mukaan. Jotkut artikkelit eivät olleet saatavilla Scopuksen kautta, mutta ne löytyivät joko Google Scholarin tai IEEE Xploren kautta.

Seuraavassa vaiheessa keräsin tutkimustietoa koneoppimisen käytöstä refaktoroinnissa Scopus- ja Google Scholar -tietokannan haulla. Käytin hakulausetta (refactoring AND ("machine learning" OR "artificial intelligence")), joka tuotti Scopuksessa 227 hakutulosta. Tuloksia arvioitiin niiden relevanssin sekä niiden saamien viittausten lukumäärän mukaan. Google Scholarissa toteutettu vastaava haku tuotti huomattavasti enemmän hakutuloksia, joista kävin läpi joitakin kymmeniä relevanssin mukaan järjestettynä. Suurin osa löydetyistä lähteistä käsitteli koneoppimisen käyttöä haisevan koodin tunnistamisessa, joten se on myös tämän tutkielman painopiste. Hakutulosten joukossa oli paljon kehittämistutkimuksia, mutta tapaustutkimuksia koneoppimisen käytöstä yritysten automaattisessa refaktoroinnissa ei löytynyt. Tämä kertoo siitä, että tutkimusaihe on uusi ja sovellukset eivät ole vielä laajemmin käytössä. Kaikkiaan tutkielmassa käytettiin 32 lähdettä, joista lähes kaikki ovat tieteellisiä artikkeleja. Mukana on myös yksi refaktorointia käsittelevä luentovideo.

3. Aiempi tutkimus

Luvussa toteutetaan kirjallisuuskatsaus, jonka tarkoituksena on selvittää refaktoroinnin taustoja ja sen toteuttamiseen liittyviä haasteita, sekä tutustua koneoppimisen käyttöön refaktoroinnin apuvälineenä. Luku on jaettu alaotsikoihin *Refaktoroinnin toteuttaminen*, *Refaktoroinnin haasteet ja riskitekijät*, *Koneoppiminen* sekä *Koneoppimisen käyttäminen refaktoroinnissa*.

3.1. Refaktoroinnin toteuttaminen

Aluksi pohditaan, milloin ja minkä vuoksi refaktorointia tulisi toteuttaa. Esitellään myös koodin “pahat hajut”, jotka toimivat usein refaktoroinnin kohteena. Luvussa tunnistetaan refaktorointiin kuuluvat työvaiheet sekä esitetään kaksi erilaista refaktorointitapaa: kevyt floss-refaktorointi ja laajempi root-canal refaktorointi. Myös malli laajojen refaktorointien toteuttamiseen esitetään.

Suurin osa ohjelmistokehittäjien ammattilaisista toteuttaa refaktorointia, kun koodia on vaikea ymmärtää tai ylläpitää (Jain & Saha, 2019). Myös ohjelman hidas suorituskyky sekä lukuisat moduulien väliset riippuvuudet ovat usein syynä refaktoroinnille (Jain & Saha, 2019). Fowler (1999) vastustaa aikataulun järjestämistä refaktoroinnille, sillä hänen mukaansa sitä tulisi toteuttaa jatkuvasti pienissä erissä ohjelmointityön ohessa. Refaktorointia on hyvä toteuttaa silloin, kun koodin toimintaa täytyy tarkastella ja ymmärtää. Esimerkiksi uuden funktion lisääminen, bugien eli virheiden korjaaminen ja koodin arviointi ovat toimenpiteitä, joiden yhteyteen refaktorointi sopii (Fowler, 1999). Vanhaa koodia joudutaan usein refaktoroimaan, sillä kehittäjien ymmärrys projektista lisääntyy ajan myötä ja vanhat ratkaisut nähdään uudessa valossa (Leppänen et al., 2015a).

Microsoftilla toteutetussa tutkimuksessa puolet kyselyyn vastanneista kehittäjistä vastasivat tekevänsä refaktorointia bugien korjaamisen ja uusien ominaisuuksien lisäämisen yhteydessä (Kim et al., 2014). Myös yleisesti refaktorointia toteutettiin ohjelmistoon tarvittavien konkreettisten muutosten vuoksi, eikä ohjelmiston huoltoa pitkällä tähtäimellä ajatellen (Kim et al., 2014). Keskimäärin Microsoftin kehittäjät käyttivät 13 tuntia kuukaudessa refaktorointiin, mikä vastaa lähes kymmentä prosenttia heidän koko työstään (Kim et al., 2014).

Jatkuva kiire on tärkeimpiä refaktoroinnin aiheuttajia, sillä kehittäjillä ei ole aikaa kirjoittaa laadukasta koodia (Leppänen et al., 2015a sekä Martin, 2019). Ohjelmaversioita julkaistaan nykyisin nopeaan tahtiin, joten kehittäjien täytyy keskittyä uusien ominaisuuksien toteuttamiseen refaktoroinnin sijaan (Leppänen et al., 2015a). On myös todettava, että ihminen kykenee harvoin tuottamaan laadukasta koodia ensimmäisellä yrityksellä, sillä ohjelmoiminen on hyvin monimutkaista. Tämän vuoksi ohjelmoijan tulisi heti refaktoroida, kun hän on saanut tekemänsä ohjelmakomponentin toimimaan (Martin, 2019). Jos refaktorointia ei tehdä, ohjelmistosta kasvaa kuukausien kuluessa valtava sotku, johon on hyvin vaikeaa ellei mahdotonta lisätä ominaisuuksia (Martin, 2019).

Mens & Tourwe (2004) jakavat refaktoroinnin kuuteen työvaiheeseen, jotka ovat:

1. Tunnista kohdat, joista ohjelmistoa tulee refaktoroida
2. Arvioi mitä refaktoroinnin tekniikoita tulisi käyttää näissä kohdissa
3. Varmista, että nämä tekniikat säilyttävät ohjelmiston toiminnan ennallaan
4. Suorita refaktorointi
5. Arvioi toteutetun refaktoroinnin vaikutuksia ohjelmiston tai prosessin laatuun
6. Säilytä eheys refaktoroidun ohjelmakoodin ja muiden ohjelmistoartefaktien välillä.

Refaktorointia aloitettaessa tulee ensimmäisenä päättää, millä abstraktiotasolla refaktorointi toteutetaan eli mihin kohteisiin (lähdekoodi, suunnittelumallit, vaatimusmäärittely) muutoksia tulee tehdä (Mens & Tourwe, 2004). Tässä tutkielmassa keskitytään lähdekoodissa tapahtuvaan refaktorointiin. Lähdekoodin refaktorointia vaativat osat tunnistetaan etsimällä koodin "pahoja hajuja" (Mens & Tourwe, 2004). Tämä Kent Beckin esittelemä termi tarkoittaa koodissa piileviä ongelmakohtia, jotka mahdollisesti vaativat refaktorointia (Fowler, 1999). Ohjelman bugista poiketen haju ei aina aiheuta virhettä ohjelmaan, mutta se voi aiheuttaa negatiivisia seurauksia ohjelman kehitykseen (Lacerda et al., 2020). Fowlerin (1999) mukaan tyypillisin haju on kopioitu koodi eli saman koodirakenteen käyttö useassa paikassa.

Hajujen tunnistaminen ei ole aina helppoa, sillä ihmisillä voi olla vastakkaisia näkemyksiä koodin refaktoroinnin tarpeesta (Mäntylä & Lassenius, 2006). Heidän mukaansa vaikeampien ongelmien tunnistamiseen vaaditaan myös paljon työkokemusta. Kuitenkaan refaktoroinnin tarkoitus ei aina ole koodin hajujen poistaminen, vaan tavoitteet voivat vaihdella paljonkin (Cedrim et al., 2017). Näitä tavoitteita ovat muun muassa heikkojen suunnitteluratkaisujen korjaaminen, koodin huoltokustannusten vähentäminen sekä koodin saaminen helpommin muokattavaksi (Cedrim et al., 2017).

Kun refaktorointia vaativa kohta tunnistetaan ja se on päätetty korjata, suunnitellaan refaktoroinnin toteutus ja pohditaan tarvittavia tekniikoita. Seuraavaksi esitetään muutamia esimerkkejä refaktoroinnin tekniikoista Fowlerin (1999) mukaan.

Ensinnäkin on metodikutsuja käsitteleviä tekniikoita. Refaktorointi voi olla yksinkertaisimmillaan muuttujien, metodien tai luokkien uudelleennimeämistä. Metodin tarkoituksen tulisi käydä ilmi sen nimestä, jotta koodin ymmärtäminen olisi nopeampaa ja helpompaa (Fowler, 1999). Hän kuvaa uudelleennimeämisen vaiheet tarkemmin kirjassaan, ja operaatioon viitataan nimellä *Rename Method.* Metodikutsujen pitkät parametrilistat ovat usein ongelmallisia, joten *Add Parameter* sekä *Remove Parameter* -tekniikat ovat hyödyllisiä parametrien käsittelyssä. Useampien parametrien välittämiseen kannattaa käyttää objekteja (Fowler, 1999). Sekaannusten välttämiseksi tulisi myös erottaa muokkauksia tekevät metodit kyselyjä toteuttavista metodeista. Jos metodi toteuttaa kumpaakin, se voidaan korjata *Separate Query from Modifier* -tekniikalla (Fowler, 1999).

On myös metodeja käsitteleviä tekniikoita, kuten *Extract Method, Inline Method* ja *Replace Temp with Query* (Fowler, 1999). *Extract Method* jakaa osan suuren metodin sisällöstä omaksi metodikseen, kun taas *Inline Method* poistaa metodin ja korvaa sen koodikappaleella. Martinin (2019) mukaan *Extract method* on tärkeimpiä menetelmiä säännöllisessä refaktoroinnissa. Jokaisella funktiolla tulisi olla vain yksi tehtävä, jonka esittämiseen ei tarvita kymmeniä koodirivejä (Martin, 2019). Kun ohjelma on jaettu

riittävän pieniin funktioihin, joilla on hyvin määritelty nimi, ohjelmakoodia voidaan ymmärtää paremmin ja siitä tulee enemmän selkokielen kaltaista (Martin, 2019). Metodien muokkaamisen aikana voi ilmetä ongelmia paikallisten muuttujien käsittelyssä, jolloin käytetään esimerkiksi *Replace Temp with Query* -tekniikkaa korvaamaan paikallisia muuttujia (Fowler, 1999).

Refaktorointia tehdään myös luokkien ja objektien välillä. *Move Method* ja *Move Field* -operaatiot siirtävät toiminnallisuutta luokkien välillä (Fowler, 1999). Jos luokalla on liikaa vastuuta, voidaan käyttää *Extract Class* -tekniikkaa. Metodien siirtäminen kuuluu refaktoroinnin perusasioihin, ja sillä voidaan saavuttaa yksinkertaisempia ja toiminnaltaan tehokkaita luokkia (Fowler, 1999).

Seuraavaksi kuvataan datan käsittelyyn liittyviä refaktoroinnin tekniikoita. *Self Encapsulate Field* -tekniikalla tehdään sisäänpääsy objektin sisältämään dataan luomalla get- ja set- metodeja (Fowler, 1999). Jos data sisältää kiinteästi toisiinsa liittyviä muuttujia, niistä voidaan muodostaa uusi luokka *Replace Data Value with Object* -tekniikalla. Myös tietorakenteena toimiva taulukko kannattaa lisätä luokan sisälle *Replace Array with Object* -tekniikalla (Fowler, 1999). Jos käyttöliittymän osat sisältävät liikaa toiminnallista logiikkaa, tulee logiikka erottaa omaan luokkaansa käyttäen *Duplicate Observed Data* -refaktorointia. Tämä yksinkertaistaa käyttöliittymän lähdekoodia sekä ohjelman toiminnallisten osien kehitystä (Fowler, 1999).

Refaktoroinnin toteutuksessa kolmas työvaihe Mens & Tourwen (2004) mukaan on varmistaa ohjelmiston toiminnan säilyminen ennallaan. Tämän vuoksi tulisi käyttää “turvaverkkoa”, jonka muodostavat versionhallinnan käyttö ja luotettavat yksikkötestit (Lahtinen & Leppänen, 2016). Näin voidaan ehkäistä riskiä ohjelman rikkomisesta, sillä yksikkötesteillä löydetään tapahtuneet virheet ja versionhallinnalla voidaan kumota ja jäljittää tehtyjä muutoksia. Fowler (1999) suosittelee refaktoroinnin toteuttamista pienissä askelissa sekä testien ajamista jokaisen askeleen jälkeen. Myös suurempia kokonaisuuksia voidaan toteuttaa kerralla, mutta virheen sattuessa hän kumoaa muutokset ja toteuttaa kokonaisuuden uudestaan pienemmin askelin (Fowler, 1999).

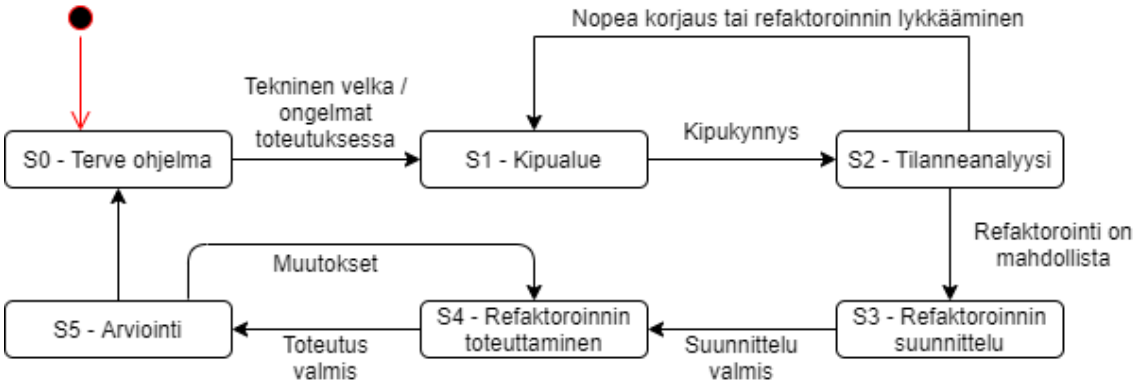
Refaktoroinnin määritelmän mukaan koodin ulkoinen toiminta ei saisi muuttua refaktoroinnin aikana. Kuitenkin Fowler (1999) ja Microsoftin työntekijät (Kim et al., 2014) toteuttivat refaktorointia myös ohjelman muutosten yhteydessä. Tämän vuoksi Murphy-Hill & Black (2008) jakavat refaktoroinnin toteuttamisen kahteen toisistaan poikkeavaan periaatteeseen: “floss refactoring” ja “root canal refactoring”. Floss-refaktoroinnilla eli “hampaiden puhdistamisella” tarkoitetaan säännöllistä ja lyhyttä refaktorointia muiden ohjelman muutosten yhteydessä, jonka tarkoituksena on ylläpitää laadukasta koodia (Murphy-Hill & Black, 2008). Siis Fowler (1999) ja Microsoftin työntekijät (Kim et al., 2014) suosivat erityisesti floss-refaktorointia.

Root canal -refaktoroinnilla tarkoitetaan pidempiä refaktorointisessioita, jolloin kehittäjät tekevät ainoastaan refaktorointia, jotta sotkuinen järjestelmä saadaan korjattua (Murphy-Hill & Black, 2008). Heidän mukaansa floss-refaktorointia tulisi toteuttaa enemmän, jotta kallista ja aikaa vievää root canal -refaktorointia ei tarvittaisi. Tämä toteutuu käytännössäkin, sillä lähes 90 prosenttia yrityksissä tehdyistä refaktoroinneista ovat floss-refaktorointeja (Liu et al., 2012).

Vaikka työn ohessa tapahtuvaa säännöllistä floss-refaktorointia tulisi suosia, joskus laajemmat root-canal refaktoroinnit ovat välttämättömiä (Leppänen et al., 2015 b). Laajat refaktoroinnit syntyivät usein ongelmista lisätä uusi ominaisuus ohjelmaan, sekä

teknisen velan kasautumisesta tai huonosta koodista (Leppänen et al., 2015 b). Tällöin kehitystiimiltä vaaditaan hyvää päätöksentekoa. Oikean ajankohdan tunnistaminen refaktoroinnille on vaikeaa, jotta refaktoroinnilla olisi tehokas vaikutus (Leppänen et al., 2015 b).

Leppänen et al. (2015 b) kehittivät empiirisen tutkimuksensa pohjalta seuraavan mallin laajoille refaktoroinneille. Mallissa prosessi on jaettu viiteen vaiheeseen, kuten kuvasta 1 havaitaan. Prosessin ensimmäinen vaihe (S0) kuvaa tervettä ohjelmaa, jolloin refaktorointia ei tarvita. Kun ohjelmassa ilmenee näkyviä puutteita, saavutaan kipualueelle (S1). Ongelmat tiedostetaan, mutta tyypillisesti refaktorointia viivytetään niin pitkään kuin mahdollista. Kun ongelmat ylittävät tietyn kynnyksen, tehdään tilanneanalyysi (S2). Kipukynnyksenä voi toimia kehittäjän oma tuntuma, koodin metriikat, pahat hajut tai kehitystyön huomattava hidastuminen. Tilanneanalyysissä kuunnellaan sidosryhmiä sekä vakavissa tapauksissa myös asiakasta, ja päätetään refaktoroinnin toteutuksesta. Laajaa refaktorointia voidaan yhä lykätä tai voidaan toteuttaa jokin nopea korjaus, joka vie prosessin takaisin kipualueelle. Jos resurssit sallivat laajan refaktoroinnin, toteutetaan refaktoroinnin suunnittelu (S3), jossa päätetään vaatimuksista, mahdollisista ratkaisuista ja aikatauluista. Refaktoroinnin toteuttamisessa (S4) tehdään versionhallintaa, refaktoroinnin operaatioita, testausta ja koodiarviointeja. Lopuksi (S5) refaktoroinnin vaikutuksia voidaan vielä arvioida kehittäjien ja metriikoiden palautteen pohjalta. (Leppänen et al., 2015 b).



Kuva 1. Laajojen refaktorointien toteuttaminen ja päätöksenteko (piirretty Leppänen et al., 2015b mukaan).

Yhteenvetona voidaan todeta, että refaktorointia tehdään siitä saatavien selkeiden hyötyjen vuoksi, joita ovat esimerkiksi koodin ymmärrettävyyden ja muokattavuuden parantuminen. Ymmärrettävää ja hyvälaatuista koodia syntyy harvoin ensimmäisellä yrityksellä, sillä käytössä oleva aika on rajallista ja tehtävät haastavia. Refaktorointia voidaan tehdä yksinkertaisten tekniikoiden avulla, joita on määritellyt muun muassa Fowler (1999). Versionhallinta ja kattavien yksikkötestien käyttö vähentää siihen sisältyviä riskejä koodin rikkomisesta. Refaktorointia kannattaa tehdä työn ohessa vähän kerrallaan, jotta ohjelma pysyisi terveenä ja vältyttäisiin laajalta refaktoroinnilta.

3.2. Refaktoroinnin haasteet ja riskitekijät

Seuraavaksi tutkitaan refaktoroinnin yleisiä haasteita, kohdistuen esimerkiksi testauksiin, tiimityöhön ja automaattisiin työkaluihin. Tutkitaan myös, kuinka paljon refaktorointi aiheuttaa virheitä ja poistaako sen toteuttaminen hajuja koodista.

Microsoftin ohjelmistokehittäjien mukaan haasteita refaktorointiin aiheuttaa erityisesti suuren ohjelmiston parissa työskentely, joka sisältää paljon moduulien välisiä riippuvuuksia (Kim et al., 2014). Suuresta ja monimutkaisesta ohjelmasta on hankalaa löytää kaikki refaktorointia vaativat kohdat (Liu et al., 2012). Löydetyt ongelmat tulee myös asettaa tärkeysjärjestykseen, ja ratkaista ne sen mukaisesti (Liu et al., 2012). Myös Tempero et al. (2017) mainitsevat refaktoroinnin vaikeustason ja tekniset ongelmat yhtenä esteenä refaktorointiin.

Siemensillä koodin hajottamisen pelko koettiin suurimmaksi esteeksi refaktoroinnille (Sharma et al., 2015). Tätä ongelmaa voidaan ratkaista esimerkiksi luomalla kattavia testejä refaktorointia varten. Manuaalista refaktorointia voidaan tehdä turvallisemmin, jos sen tukena on riittävästi regressiotestejä (Rachatasumrit & Kim, 2012). Usein regressiotestit ovat riittämättömiä refaktoroinnin turvallisuuden takaamiseen, jolloin kehittäjät eivät voi toteuttaa refaktorointeja (Kim et al., 2014). Tutkimalla kolmea avoimen lähdekoodin Java-projektia Rachatasumrit & Kim (2012) tulivat siihen tulokseen, että ainoastaan 22 prosenttia refaktoroinneista testattiin. Tämä osoittaa, että usein refaktorointeja ei testata tarpeeksi ja että regressiotestit eivät ole riittävän kattavia (Rachatasumrit & Kim, 2012).

Microsoftilla ongelmia aiheutti koordinaation puuttuminen työntekijöiden ja tiimien välillä refaktorointia toteutettaessa (Kim et al., 2014). Tämän vuoksi suositellaan pieniä tiimikokoja refaktoroinnin toteuttamisessa, jolloin kommunikaatio on helpompaa (Lahtinen & Leppänen, 2016). Kommunikaatiosta on syytä huolehtia myös ohjelmiston versionhallinnan piirissä. Usein dokumentaatiossa käytetään vaihtelevia nimityksiä refaktoroimisesta, kuten esimerkiksi redesign tai reorganizing (AlOmar et al., 2021). Refaktorointien kommentointi myös vaihtelee hyvin suurpiirteisistä kuvauksista yksityiskohtaisempiin operaatioiden kuvauksiin. Tämä ilmentää yleisten sääntöjen puuttumista refaktoroinnin dokumentoinnissa (AlOmar et al., 2021).

Refaktoroinnin toteuttamiseksi tarvitaan resursseja, joita on yrityksissä saatavilla rajoitetusti. On tyypillistä, että johtoporras ei anna kehittäjille riittävästi aikaa refaktorointiin (Sharma et al., 2015). Välillä työntekijöiden ja johdon tietämättömyys refaktoroinnin vaikutuksista tuotteen laatuun vähentää refaktoroinnin toteuttamista (Sharma et al., 2015). Ajan puute ja tiukat deadlinet rajoittavatkin huomattavasti refaktorointia (Tempero et al, 2017). Refaktoroinnin vaikutukset eivät näy ulkoisesti, joten sen arvoa on myös vaikea selittää asiakkaille (Leppänen et al., 2015 b).

Koska refaktorointi on pääsääntöisesti haastavaa, sitä varten kehitetään jatkuvasti automaattisia työkaluja. Työkalut eivät kuitenkaan ole vielä valmiita, vaan niidenkin käytössä esiintyy ongelmia. Siemensillä tehdyn empiirisen tutkimuksen mukaan useimmat kehittäjät olivat tyytymättömiä työkaluihin, sillä ne tuottivat tuloksenaan liian paljon tarpeettomia refaktorointeja (Sharma et al., 2015). Myös työkalujen kyky ennustaa refaktoroinnin vaikutuksia koettiin riittämättömäksi (Sharma et al., 2015). Useimmat työkalut eivät myöskään tiedä suunnitteilla olevan refaktoroinnin kohdetta ja rajauksia, eivätkä siten osaa suositella oikeita kohteita (Rebai et al., 2020). Kehittäjällä

voi olla tavoitteenaan tietyn laatuattribuutin (esimerkiksi testattavuuden) parantaminen, jolloin työkalun tulisi suositella niitä vastaavia refaktorointeja.

Microsoftilla kehittäjät toteuttivat keskimäärin 86 prosenttia refaktoroinneista manuaalisesti, ja jopa puolet kehittäjistä toteuttivat kaiken refaktoroinnin manuaalisesti (Kim et al., 2014). Luottamusta työkaluihin ei siis vielä ollut edellisen vuosikymmenen alkupuolella. Rebai et al. (2020) jopa esittävät, että refaktorointia ei voi täysin automatisoida sen subjektiivisen luonteen vuoksi: kehittäjillä on yleensä oma visionsa ratkaisusta ja he päättävät siten refaktoroinnin toteutuksesta. Kehittäjien tulee tietenkin ymmärtää oma työnsä perin pohjin, joten työkalun automaattisesti tekemä muokkaus voisi aiheuttaa sekaannusta.

Tyypillisesti automaattiset työkalut osoittavat lähdekoodista kohteita refaktorointia varten, eli tunnistavat haisevaa koodia. Työkalujen olisi hyvä tarjota enemmän sisältöä, kuten esimerkiksi muutosten integrointia, refaktoroinnin kustannusarvioita ja validointia (Kim et al., 2014). Myös ohjelman visualisoiminen olisi avuksi (Jain & Saha, 2019).

Refaktorointiin liittyy paljon riskejä, joten sen aloittamista tulee aina harkita tarkoin. Esimerkiksi refaktorointi ei aina paranna koodia lainkaan tai koodin laatu voi jopa huonontua, jolloin tuhlataan samalla aikaa ja vaivaa (Leppänen et al., 2015 b). Refaktorointiin liittyvä tyypillisin riski on Microsoftin ohjelmistokehittäjien mukaan vaikeasti havaittavien bugien syntyminen (Kim et al., 2014). Kuinka usein näitä bugeja keskimäärin syntyy? Bavota et al. (2012) mukaan yleensä refaktoroidut luokat olivat bugien korjausten kohteena yhtä usein kuin ei-refaktoroidut luokat. Tietyt refaktoroinnin tekniikat olivat kuitenkin virhealttiimpia kuin toiset, ja erityisesti tekniikat *Pull-up method*, *Extract Subclass*, *Inline Temp*, *Replace Method With Method Object* ja *Extract Method* aiheuttivat virheen noin 25 - 40 prosentin todennäköisyydellä (Bavota et al., 2012). Microsoftilla riskejä syntyi myös refaktoroinnin aiheuttamista ylimääräisistä testauskustannuksista sekä merge-operaatioiden eli ohjelmiston kehityshaarojen yhdistämisen ongelmista (Kim et al., 2014).

Refaktorointia tehdään usein koodin hajujen pohjalta, mutta tuloksena ei välttämättä poisteta hajuja ohjelmasta. Cedrim et al. (2017) tutkimuksessa tarkasteltiin 23 avoimen lähdekoodin ohjelmaa, ja he analysoivat niissä toteutettuja refaktorointeja. He havaitsivat, että 80 prosenttia refaktoroinneista käsittelivät pahanhajuisia rakenteita, mutta ainoastaan 10 prosenttia refaktoroinneista poistivat hajuja. Sen sijaan kolmasosa refaktoroinneista lisäsi hajujen määrää ohjelmassa. Esimerkiksi metodeja liikuttavat refaktoroinnit *Move Method* ja *Pull Up Method* aiheuttivat hajun *God Class* noin 30 prosentin todennäköisyydellä, sekä *Extract Superclass* -refaktorointi aiheutti hajun *Speculative Generality* 68 prosentin todennäköisyydellä (Cedrim et al., 2017).

Vaikka refaktoroinnissa piileekin lukuisia ongelmia, sen hyödyt tunnustetaan pääsääntöisesti riskejä suuremmiksi. Vaikka metriikat eivät aina puhu refaktoroimisen puolesta, on mielestäni tärkeää että lähdekoodi on itsessään ymmärrettävää myös muille kuin koodin alkuperäiselle kirjoittajalle. Tällöin koodin arviointi ja muokkaaminen on myöhemmin nopeampaa, sillä ohjelmoijan ei tarvitse käyttää kaikkea aikaansa ohjelman toiminnan ymmärtämiseksi. Ymmärrettävää koodia on usein vaikea kirjoittaa ensi yrittämällä, jolloin refaktorointi on tarpeellinen toimenpide.

3.3. Koneoppiminen

Koneoppimisella tarkoitetaan tietokonejärjestelmiä, jotka kehittävät toimintaansa automaattisesti kokemuksen kautta (Jordan & Mitchell, 2015). Koneoppiminen on nykyisin tekoälytutkimuksen ytimessä ja sen sovellukset vaikuttavat jo lukuisilla liiketoiminta-aloilla. Koneoppimisen käyttö ja tutkimus koki suuren harppauksen viime vuosikymmenenä, jota selittää pilvidatan ja halvan laskentatehon huikea nousu, sekä algoritmeissa ja teoriassa tapahtuneet edistysaskeleet (Jordan & Mitchell, 2015).

Aikaisemmin tekoälyjärjestelmiä luotiin määrittelemällä joukko sääntöjä, joiden mukaan tekoäly laskee syötteelle (input) tuloksen (output). Tämä on kuitenkin vaivalloista ja kattavia sääntöjä onkin mahdotonta luoda monimutkaisiin ongelmiin, kuten esimerkiksi kasvojentunnistamiseen. Koneoppimista käyttämällä ei tarvita monimutkaista tekoälyn ohjelmointia, mutta tulee kuitenkin laatia suuret määrät dataa tekoälyn kouluttamista varten. Koulutusdatan pohjalta algoritmi oppii automaattisesti muodostamaan yhteydet syötteistä oikeisiin tuloksiin. Konseptitasolla algoritmi ikään kuin etsii lukemattomien vaihtoehtojen joukosta oikean ohjelman, joka on optimaalinen valittuun tehtävään (Jordan & Mitchell, 2015).

Yleisimmin käytetty koneoppimisen menetelmä on ohjattu oppiminen, jossa koulutusdata muodostuu oikeiksi määritellyistä syöte/tulos pareista (Jordan & Mitchell, 2015). Algoritmille siis kerrotaan, millainen tulos tulisi palauttaa kunkin syötteen kohdalla. Koulutusdata otsikoidaan tulosten mukaan, eli jokaisella syötteellä on otsikkona sitä vastaava tulos. Kun ohjattua oppimista sovelletaan esimerkiksi haisevan koodin tunnistamiseen, koulutusdata sisältää sekä haisevaa että puhdasta koodia. Kun algoritmille syötetään pieni kappale haisevaa koodia, sen tulee palauttaa tulos “haiseva” sekä vastaavasti puhtaan koodin tapauksessa tulos “puhdas”.

On olemassa myös ei-ohjattua oppimista, jolloin koulutusdataan ei ole määritetty oikeita tuloksia, vaan algoritmin annetaan itse muodostaa omat johtopäätöksensä. Ei-ohjatussa oppimisessa dataa voidaan kerätä automaattisesti suuria määriä, sillä ihmisen prosessointia ei juuri tarvita. Tulee kuitenkin kiinnittää huomiota koulutusdatan laatuun, sillä virheellinen data aiheuttaa epäsuotuisia tuloksia algoritmin toiminnassa. Ei-ohjattu oppiminen voi olla tulevaisuudessa entistä tärkeämpää, sillä se on esimerkiksi ihmisten ja eläinten luontainen oppimismalli (LeCun et al., 2015).

Syväoppiminen on eräs koneoppimisen alalaji, jossa käytetään laajoja neuroverkkoja, joissa on useita kerroksia (LeCun et al., 2015). Syväoppimisen avulla löydetään raakadatasta monimutkaisia rakenteita automaattisesti. Ohjatussa syväoppimisessa käytetään otsikoitua dataa (esimerkiksi haiseva/puhdas koodi), mutta enempää ihmisen ohjausta kuten parametreja tai metriikoita ei tarvita. Syväoppiminen vaatii kuitenkin hyvin suuren määrän koulutusdataa (Liu et al., 2021).

Koulutuksen tuloksena koneoppimisalgoritmi muodostaa kuvausfunktion, josta käytetään myös nimeä luokittelija (classifier). Se voi olla muodoltaan päätöspuu, logistinen regressiomalli, tukivektorikone, neuroverkko tai Bayes-luokittelija (Jordan & Mitchell, 2015). Luokittelija f(x) = y palauttaa tuloksen y jokaiselle syötteelle x. Tuloksena voi olla myös todennäköisyysjakauma y:n arvoista. Tässä tutkielmassa käsitellään ohjattua koneoppimista sekä syväoppimista, ja yleisimpiä algoritmeja ovat päätöspuuhun, neuroverkkoihin ja Bayes-luokittelijoihin perustuvat algoritmit.

3.4. Koneoppimisen käyttäminen refaktoroinnissa

Koneoppiminen on tehokas keino monimutkaisten ongelmien kuvaamiseksi, kun kattavia sääntöjä ongelman kuvaamiseksi ei voida laatia. Tyypillisesti koneoppimismenetelmät selviytyvät hyvin erilaisista tunnistustehtävistä, kuten esimerkiksi kasvojen, kuvien tai äänen tunnistamisesta. Seuraavaksi tutkitaan koneoppimisen käyttöä haisevan koodin tunnistamisessa. Tässä tapauksessa tunnistamisen kohteena toimivat lähdekoodin osat, joista algoritmi etsii hajuja tai arvioi niiden esiintymisen todennäköisyyttä. Jos tärkeimmät refaktorointia vaativat kohdat voidaan tunnistaa lähdekoodista automaattisesti, kehittäjiltä vaadittava manuaalisen työn määrä vähenee refaktorointia suorittaessa. Toinen tutkimuskohde on hajujen vakavuustason arvioiminen: on tärkeää että ohjelma esittää vakavimmat hajut ensimmäisenä, jotta kehittäjät voivat priorisoida tulosten välillä.

Hajujen tunnistaminen vaatii kehittäjältä laajaa tietämystä ja kokemusta aiheesta, joten koneoppimiseen perustuva algoritmi olisi hyödyllinen tunnistamisessa (Jain & Saha, 2021). Koneoppimisen käyttäminen auttaisi kehittäjiä näkemään refaktorointia vaativat kohdat nopeammin ja tarkemmin (Imazato et al., 2017). Useita työkaluja on kehitetty koodin hajujen tunnistamiseen, mutta tulokset vaihtelevat työkalujen välillä sekä vääriä positiivisia tuloksia esiintyy paljon (Fontana & Zanoni, 2017). Työkaluissa on siis vielä puutteita. Väärät positiiviset tulokset ovat työkalun löytämiä hajuja, jotka eivät todellisuudessa ole ongelmallisia. Mielestäni ne vähentävät huomattavasti kehittäjien kiinnostusta työkalujen käyttöön, koska ne tarjoavat heidän kannaltaan typeriä vaihtoehtoja. Myös Yue et al. (2018) mukaan kaikkea koodissa esiintyvää hajua ei tarvitse refaktoroida, joten automaattisen työkalun tulisi esittää ainoastaan tärkeimmät tapaukset refaktorointia varten.

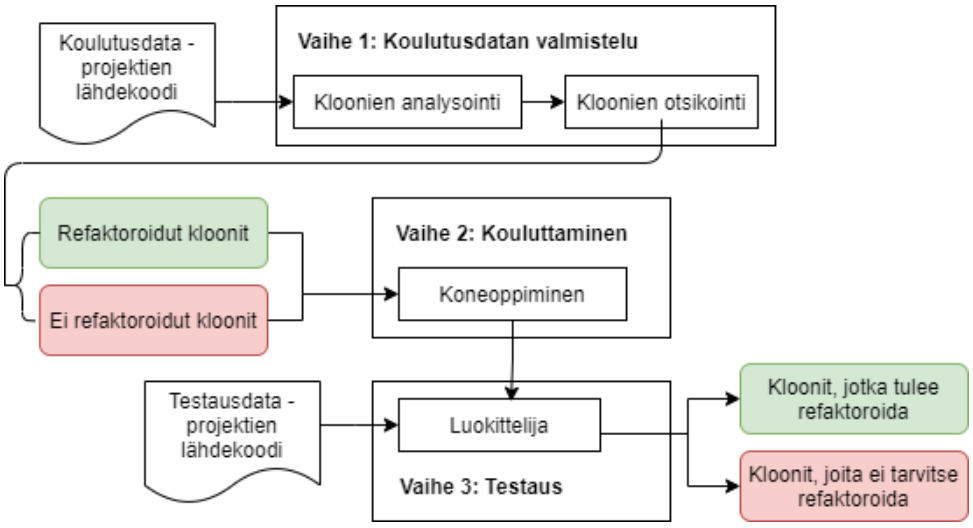
Haisevan koodin tunnistamisessa suurin osa aikaisemmista tutkimuksista pohjautuu kahteen eri menetelmään: toiset käyttävät sääntöihin perustuvia menetelmiä ja toiset taas koneoppimiseen ja koodin metriikoihin perustuvia menetelmiä (Fontana et al., 2016). Heidän mukaansa kummallekin lähestymistavalle voi olla paikkansa, jotta ongelma automaattisesta haisevan koodin tunnistamisesta saataisiin ratkaistua. Sääntöihin perustuvat työkalut vaativat toimiakseen manuaalisesti kehitettyjä sääntökokoelmia, joiden avulla työkalu tunnistaa koodin hajuja. Tämä vie kehittäjiltä paljon aikaa, eikä sääntöihin perustuvia menetelmiä ole myöskään osoitettu koneoppimista paremmiksi (Fontana et al., 2016). Myös optimaalisten sääntöjen valinta on vaikeaa sekä riippuvaista suunnittelijan henkilökohtaisista näkemyksistä (Liu et al., 2021). Eräs ratkaisu on syväoppimisen käyttäminen, joka ei tarvitse otsikoidun koulutusdatan lisäksi enempää ihmisen ohjausta (Liu et al., 2021).

Koneoppimisen käyttäminen vähentää selvästi algoritmin suunnittelussa tarvittavan manuaalisen työn määrää, mutta kehittäjien pitää vielä määritellä algoritmin tarvitsema koulutusdata (Fontana et al., 2016). Tyypillisesti koulutusdata sisältää useista järjestelmistä kerättyä avointa lähdekoodia. Ohjatun oppimisen periaatteiden mukaisesti data täytyy jakaa hajuja sisältävään dataan sekä puhtaaseen dataan, jotta algoritmi voi tehdä analyysiä niiden välillä. Manuaalinen koulutusdatan jaottelu on kuitenkin hidasta. Jotta algoritmeille saataisiin syötettyä enemmän koulutusdataa, automaattisia menetelmiä tarvitaan datan keräämiseen.

RefactoringMiner on eräs suhteellisen tuore työkalu, joka analysoi ohjelmiston versionhallintaa ja palauttaa listana kaikki toteutetut refaktoroinnit kahden eri ohjelmaversion välillä (Tsantalis et al., 2018). Testeissä ohjelma saavutti 98% tarkkuuden ja 87% tunnistuskyvyn (Tsantalis et al., 2018). Ohjelmaa voidaan käyttää tunnistamaan refaktoroitua lähdekoodia, joka on oletettavasti hyvälaatuista. Ottamalla saman lähdekoodin aiempi versio, johon ei ole tehty refaktorointeja, saadaan tarvittavaa haisevaa koodia koulutusdataa varten (Aniche et al., 2020). Liu et al. (2021) kehittivät myös automaattisen menetelmän koulutusdatan generoimiseen. Menetelmä suorittaa "käänteisiä refaktorointeja" hyvin suunniteltuun lähdekoodiin, eli luo haisevaa koodia jota voidaan käyttää algoritmin kouluttamisessa.

Seuraavaksi esitetään useita tutkimuksia, joissa testataan erilaisia menetelmiä refaktorointia vaativan koodin automaattiseen tunnistamiseen. Tutkimuksissa käytetään arviointimetriikoina tarkkuutta sekä tunnistuskykyä. Tarkkuudella tarkoitetaan tekoälyn antamien oikeiden tulosten osuutta kaikista sen antamista tuloksista. Tunnistuskyky taas vertaa tekoälyn antamia oikeita tuloksia kaikkiin datassa esiintyviin tuloksiin (Imazato et al., 2017). Ensimmäisenä esitetään lyhyesti CREC-työkalun kehitysprosessi sekä siihen kohdistettujen testien tulokset. Useat seuraavat tutkimukset käyttävät myös samaa periaatetta koneoppimisen toteuttamisessa.

Yue et al. (2018) kehittivät koneoppimiseen perustuvan työkalun nimeltä CREC, joka tunnistaa lähdekoodista automaattisesti kopioidun koodin tapauksia eli klooneja refaktorointia varten. Heidän menetelmänsä jakaantuu kolmeen vaiheeseen, kuten kuvasta 2 selviää. Vaiheessa I tunnistetaan lähdekoodista klooneja, jotka analysoidaan ja otsikoidaan sen mukaan, onko niihin kohdistettu refaktorointia. Vaiheessa II syötetään data koneoppimisalgoritmille sekä koulutetaan luokittelija. Vaiheessa III testataan työkalua tunnistamalla lähdekoodista refaktorointia vaativia kohtia luokittelijan avulla. CREC -työkalua testattiin kuudessa avoimen lähdekoodin ohjelmistoprojektissa, joista se löysi tehokkaasti kopioitua koodia refaktorointia varten. Ohjelma saavutti keskimäärin 82% tarkkuuden ja 86% tunnistuskyvyn. Algoritmeista parhaiten suoriutuivat AdaBoost ja RandomForest, jotka perustuvat päätöspuuhun. Sen sijaan Naive Bayes suoriutui testeissä huonosti. Viiden testatun algoritmin joukosta päätöspuuhun perustuvat algoritmit olivat selvästi parempia kuin muut (Yue et al., 2018).



Kuva 2. CREC-työkalun toiminnan kolme vaihetta (piirretty Yue et al., 2018 mukaan).

Imazato et al. (2017) tutkivat myös automaattista refaktorointia koneoppimisen avulla tavoitteenaan löytää kohteita *Extract Method* -refaktoroinnille. He toteuttivat työkalun, joka etsii syötteenä annetun ohjelmiston versionhallinnasta aikaisemmin tehtyjä *Extract Method* -operaatioita ja tähän informaatioon perustuen suosittelee uusia kohteita refaktoroinnille. Työkalua testattiin viidessä ohjelmistoprojektissa sekä viiden koneoppimisalgoritmin tuloksia vertailtiin. Päätöspuuhun perustuvat algoritmit J48 ja RandomForest saavuttivat yli 89% tarkkuuden ja tunnistuskyvyn, joten Imazato et al. (2017) mukaan koneoppimisen käyttö on hyödyllistä refaktoroinnin mahdollisuuksien havaitsemiseksi. Heidän mukaansa algoritmin tarkkuus ei myöskään riippunut kohdeohjelman versionhallinnan laajuudesta, vaan tulokset olivat tarkkoja myös uusissa ja verrattain pienemmissä kohteissa. Koneoppimisen avulla on mahdollista löytää jokaisen kohdeohjelman erityispiirteisiin sopivia tuloksia, sillä algoritmi koulutetaan kohteessa aikaisemmin tehtyjen refaktorointien pohjalta. (Imazato et al., 2017).

Fontana et al. (2016) tutkivat koneoppimisen käyttöä haisevan koodin tunnistamisessa. Vertailuaineistona toimi 74 ohjelmistoprojektia, joista algoritmit etsivät neljää hajua: Data Class, Large Class, Feature Envy ja Long Method. 16 erilaista koneoppimisen algoritmia vertailtiin, joista kaikki suoriutuivat tunnistamisesta korkealla, jopa yli 95 prosentin tarkkuudella. Silti parhaat suorituskyvyt löytyivät RandomForest- ja J48-algoritmeista. Tulokset osoittavat, että koneoppimista voidaan käyttää haisevan koodin tunnistamiseen ainakin tiettyjen hajujen osalta (Fontana et al., 2016). Koska suuria eroja algoritmien välillä ei löytynyt, suorituskyvyn kannalta tärkeintä ei olekaan oikean algoritmin valinta. Tärkeämpää on tapa, jolla algoritmit koulutetaan ja pohjustetaan (Fontana et al., 2016).

Koneoppimisen käyttämisessä on ongelmana epätasainen data, sillä hajuja löytyy lähdekoodista suhteellisen harvoin. Algoritmien suorituskyky heikkeni, kun ne käsittelivät vähemmän hajuja sisältävää dataa (Fontana et al., 2016). Samaa mieltä ovat Di Nucci et al. (2018), jotka toistivat Fontana et al. (2016) tutkimuksen käyttäen eri tavalla muodostettua koulutusdataa. He vähensivät datan sisältämien hajujen määrää sekä sekoittivat hajuja keskenään, jotta data vastaisi paremmin todellisuutta. Tuloksena mitattu tunnistuskyky putosi huomattavasti, eli algoritmit eivät pystyneet kunnolla tunnistamaan hajuja (Di Nucci et al., 2018). Algoritmin tarkkuus laski jonkin verran, saavuttaen noin 76 prosenttia. Heidän mukaansa koneoppimisen käyttämisessä on siis vielä huomattavia rajoituksia ja lisää tutkimusta aiheesta tarvitaan.

Aniche et al. (2020) käyttivät tutkimuksessaan koneoppimista tavoitteenaan tunnistaa automaattisesti refaktoroinnin tarpeessa olevaa ohjelmakoodia. He käyttivät koulutusdatan keräämisessä RefactoringMiner -ohjelmaa. Ohjelman avulla analysoitiin tuhansien ohjelmistoprojektien versionhallintaa, joista eriteltiin koulutusdataan koodia ennen ja jälkeen refaktoroinnin. Koulutusdata sisälsi kaikkiaan 20 eri tyyppistä refaktoroinnin operaatioita, ja jokaista operaatiota kohden koulutettiin oma luokittelija tunnistamaan kyseistä refaktoroinnin operaatioita vaativaa koodia. Lajittelualgoritmeista RandomForest suoriutui parhaiten, saavuttaen keskimäärin 87% tarkkuuden ja 84% tunnistuskyvyn (Aniche et al., 2020). Algoritmia testattiin koulutusdatan ulkopuolisella ohjelmalla, joten tulos vaikuttaa lupaavalta käytännön sovelluksia ajatellen.

Liu et al. (2021) käyttivät tutkimuksessaan syväoppimista koodin hajujen tunnistamiseksi. He keräsivät tarvitsemansa koulutusdatan avoimen lähdekoodin ohjelmista automaattisesti käänteisiä refaktorointeja käyttäen, joita kuvailin luvun alkupuolella. Datan avulla he kouluttivat useita neuroverkkopohjaisia luokittelijoita

erilaisten hajujen tunnistamista varten. Kun menetelmä arvioi koodia, jokainen luokittelija muodostaa siitä oman tuloksensa ja lopullinen tulos ratkeaa äänestämällä käyttäen bootstrap-koostamista (Liu et al., 2021). Bootstrap-koostaminen on yksinkertainen ja tehokas menetelmä, jossa tulos valitaan äänienemmistön mukaisesti (Sagi & Rokach, 2018). Se pohjaa periaatteeseen, jonka mukaan useasta mielipiteestä yhdistetty tulos on usein parempi kuin yksittäinen arvio (Sagi & Rokach, 2018). Menetelmää testattiin avoimen lähdekoodin sovelluksissa, joista se löysi hajuja tehokkaasti ylittäen verrokkina käytetyn JDeodorant -työkalun (Liu et al., 2021). Eri hajujen välillä tunnistuskyky oli noin 80-90 prosenttia. Tarkkuus oli kuitenkin keskimäärin vain 40 prosenttia, joten ohjelma antoi paljon vääriä positiivisia tuloksia.

Yhteenvetona voidaan todeta, että automaattista haisevan koodin tunnistamista sekä refaktorointimahdollisuuksien tunnistamista on tutkittu useissa tutkimuksissa, joiden tulokset ovat pääosin lupaavia. Tutkimuksissa koneoppiminen tunnistaa hajuja yli 80 prosentin tarkkuudella sekä tunnistuskyvyllä. Menetelmässä on vielä rajoituksia ja lisää tutkimusta aiheesta tarvitaan, kuten Di Nucci et al. (2018) esittää. Tutkimukset käyttävät ohjatun oppimisen menetelmiä, jotka vaativat otsikoitua koulutusdataa. Dataa voidaan kerätä ja otsikoida manuaalisesti tai automaattisia työkaluja käyttäen. Datan avulla koulutetaan yksi tai useampi luokittelija, joka tunnistaa hajuja sille syötetystä lähdekoodista. Testeissä koulutusprosessi toteutettiin useaan kertaan erilaisia algoritmeja vertaillen. Päätöspuuhun perustuvat algoritmit, kuten RandomForest suoriutuivat testeissä yleisesti parhaiten. Algoritmien väliset erot vaihtelivat tutkimusten välillä, ja aina selkeää voittajaa ei löytynyt. Fontana et al. (2016) mukaan algoritmin valintaa tärkeämpää onkin tapa, jolla ne koulutetaan ja pohjustetaan.

4. Pohdinta

Mikä on koneoppimisen merkitys refaktoroinnin toteuttamisessa: voidaanko koneoppimisen avulla välttää refaktorointiin liittyviä riskejä? Vastaan tutkimusongelmaan käyttämällä pohjana edellisessä luvussa esitettyjä tutkimuksia, sekä teen niistä joitakin johtopäätöksiä. Lukuisia haisevan koodin tunnistamista käsitteleviä tutkimuksia esitettiin, joissa saavutettiin kohtalaisen hyviä tuloksia. Tulosten arvioinnissa kiinnitän huomiota sekä algoritmin tarkkuuteen että tunnistuskykyyn, sillä kummankin arvon tulee olla korkea algoritmin käytännöllisyyden kannalta (Imazato et al., 2017). Jos tarkkuus on huono, algoritmi antaa paljon vääriä positiivisia tuloksia, jotka katsottiin useissa tutkimuksissa vähentävän työntekijöiden kiinnostusta työkalun käyttöön. Toisaalta tunnistuskykyäkin tarvitaan, jotta tärkeitä refaktorointia vaativia kohtia ei jäisi huomaamatta.

Useimpien koneoppimismenetelmien tarkkuus ja tunnistuskyky oli testeissä noin 80-90 prosentin väliltä, joka on hyvä saavutus. Poikkeuksena on viimeinen syväoppimista koskeva tutkimus, jossa tarkkuus jäi yllättävästi vain 40 prosenttiin. Miksi tulos on näin alhainen, vaikka syväoppiminen on menetelmänä laajempi ja käyttää enemmän laskentatehoa? Syy voi piillä esimerkiksi koulutustavassa, sillä Liu et al. (2021) käyttivät keinotekoisesti luotua haisevaa koodia, joka ei vastaa ihmisten tekemiä suunnitteluratkaisuja. Tekoälyä myös testattiin käyttämällä laajoja avoimen lähdekoodin ohjelmia, joista tekoäly oli nähnyt koulutuksessa vain keinotekoisesti muokattuja versioita. Testausasetelma oli siis haastava. Eri tutkimusten tuloksia on vaikea verrata toisiinsa, sillä niitä ei testattu samoilla ohjelmilla ja asetuksilla.

Erilaisen tekoälyn koulutustavan vaikutus huomattiin Di Nucci et al. (2018) tekemässä tutkimuksessa, kun he toistivat Fontana et al. (2016) tutkimuksen. Di Nucci et al. (2018) muuttivat tutkimuksessa käytettyä koulutusdataa vähentämällä hajujen määrää datassa. He myös kouluttivat eri tyypin hajut samassa datakokoelmassa, kun taas Fontana et al (2016) jakoivat eri tyypin hajut erillisiin datakokoelmiin. Selvästi tekoälylle asetettiin vaikeampi tehtävä toistotutkimuksessa, joten ei ole ihme että tulokset laskivat selvästi. Ei ole kuitenkaan tarkoituksenmukaista vaikeuttaa tekoälyn oppimisprosessia liikaa. Tutkimusten mukaan koneoppimisella ei saavuteta hyviä tuloksia, jos koulutusdata on epätasainen (Fontana et al., 2016) eli tässä tapauksessa sisältää liian vähän positiivisia hajuja negatiivisiin tapauksiin nähden. Fontana et al. (2016) koulutusdata sisälsi 140 positiivista ja 280 negatiivista tapausta yhtä koodin hajua kohden. He panostivat selvästi koulutusdatan laatuun, jolloin sen määrä jäi melko vähäiseksi. Toinen lähestymistapa on käyttää automaattisesti luotua mutta laadullisesti heikompaa koulutusdataa. Tällöin positiivisia ja negatiivisia tapauksia voidaan syöttää huomattavasti enemmän, kymmeniä tai satoja tuhansia.

Myös Imazato et al. (2017) tulokset ovat mielenkiintoisia. Heidän menetelmänsä tarvitsee syötteeksi kohteena olevan ohjelmiston dokumentaation, mutta löytää siten yksilöllisiä ehdotuksia refaktoroinnille tehokkaasti. Kaiken kaikkiaan useat tutkimukset antavat suuntaa sille, että koneoppimisen käyttäminen tulee olemaan varteenotettava vaihtoehto refaktoroimisen apuvälineenä. Sen sovellukset eivät ilmeisesti vielä ole laajasti käytössä, sillä empiirisiä tutkimuksia sovellusten käytöstä yritysmaailmassa ei löytynyt.

Esitetyt tutkimukset keskittyivät selvästi automaattisen haisevan koodin tunnistamiseen, mutta hajujen vakavuusasteen arviointi on vähemmän tutkimuksen kohteena. Olisi tärkeää, että automaattinen työkalu voisi myös luotettavasti asettaa tulokset tärkeysjärjestykseen, jotta tärkeimmät ongelmat erottuisivat selvästi. Tämä voi olla seuraava tutkimusaihe, kun automaattiset tunnistusmenetelmät kehittyvät entisestään. Esitetyistä tutkimuksista ilmenee myös, että koneoppiminen ei ole vielä täydellinen menetelmä automaattiseen haisevan koodin tunnistamiseen. Aihetta tutkitaan kuitenkin jatkuvasti, ja niin koneoppimisalgoritmit kuin niiden soveltaminen automaattisessa refaktoroinnissa tulee kehittymään. Tilausta olisi esimerkiksi laajalle syväoppimis-projektille, jolle tulisi kerätä koulutusdataa tuhansista avoimen lähdekoodin ohjelmista. Viime vuosina syväoppimisen avulla on luotu esimerkiksi maailman vahvimmat shakki- ja go-pelimoottorit, jotka ovat ihmisen taitoja huomattavasti edellä. Siksi syväoppiminen on varmasti varteenotettava ratkaisu myös refaktoroimiseen.

On myös pohdittava, mikä tulee olemaan automaattisen haisevan koodin tunnistamisen merkitys refaktorointityössä. Onko näille työkaluille tarvetta ohjelmoijien keskuudessa? Työkaluja on jo olemassa, mutta ne eivät ole vielä tarkkoja ja niiden käyttöaste on vähäistä (Sharma et al., 2015, Kim et al., 2014). Tarkoille tunnistustyökaluille voisi olla käyttöä, sillä kaikilla ohjelmoijilla ei ole koodin hajujen tunnistamisessa vaadittavaa kokemusta (Mäntylä & Lassenius, 2006). Myös refaktoroinnin automaattinen toteuttaminen olisi tärkeää, kun tärkeimmät kohdat sitä varten on saatu tunnistettua. Refaktoroinnin päätösten tulee kuitenkin lähteä kehittäjältä itseltään, sillä he päättävät työnsä toteutuksesta ja heidän tulee ymmärtää oma koodinsa (Rebai et al., 2020).

Kuten aiemmassa tutkimuksessa osoitettiin, refaktorointi sisältää paljon riskejä. Esimerkiksi koodin hajottaminen tai virheiden syntyminen on varteenotettava uhka refaktoroinnissa. Näitä riskejä voidaan kuitenkin hallita käyttämällä “turvaverkkona” versionhallintaa sekä kattavia yksikkötestejä (Lahtinen & Leppänen, 2016). Koneoppimisen käyttäminen ei ratkaise näitä riskejä, sillä tutkimuksen tämän hetken menetelmät ainoastaan tunnistavat sopivia kohtia refaktorointiin, mutta eivät toteuta niitä. Ohjelmointiympäristöt (IDE:t) tarjoavat jo työkaluja refaktorointien toteuttamiseen, mutta ne eivät vielä tunnista ja priorisoi refaktorointeja ideaalisella tavalla. Kun refaktorointien automaattinen tunnistaminen ja toteuttaminen voidaan luotettavasti yhdistää, saadaan tuloksena työkalu joka säästäisi kehittäjiltä aikaa ja vaivaa, ja tarjoaisi turvallisen menetelmän refaktoroinnin toteuttamiseen.

5. Yhteenveto

Ohjelmiston refaktoroinnilla tarkoitetaan ohjelmakoodin rakenteen kehittämistä kuitenkaan muuttamatta ohjelmiston ulkoista toimintaa (Fowler, 1999). Refaktorointi on siis tavallaan ohjelmiston siistimistä, jonka seurauksena koodin ymmärrettävyys paranee ja ohjelmoiminen on jatkossa tehokkaampaa (Fowler, 1999). Refaktorointia tehtiin usein bugien korjaamisen ja uusien ominaisuuksien lisäämisen yhteydessä (Kim et al., 2014). Refaktorointia vaativat kohdat tunnistettiin etsimällä koodin “hajuja”, jotka ovat merkki huonosta toteutuksesta (Mens & Tourwe, 2004). Refaktorointi toteutettiin joko kevyenä floss-refaktorointina tai laajana root canal -refaktorointina (Murphy-Hill & Black, 2008). Koska laaja refaktorointi aiheuttaa usein suuria kustannuksia, kannattaa suosia kevyttä ja säännöllistä refaktorointia, joka ylläpitää siistiä koodia.

Refaktorointiin liittyvä tyypillisin riski oli Microsoftin ohjelmistokehittäjien mukaan vaikeasti havaittavien bugien syntyminen (Kim et al., 2014). Usein ohjelmoijilla ei ollut käytössään riittävästi testejä turvalliseen refaktorointiin (Kim et al., 2014). Tämän vuoksi refaktorointeja ei yleensä testattu riittävästi (Rachatasumrit & Kim, 2012). Muita haasteita refaktoroinnissa olivat ohjelmamoduulien monimutkaiset riippuvuudet, tiukka aikataulu sekä kommunikaation ja työkalujen puutteellisuudet.

Koneoppimisella tarkoitetaan tietokonejärjestelmiä, jotka kehittävät toimintaansa automaattisesti kokemuksen kautta (Jordan & Mitchell, 2015). Koneoppimisessa käytetään koulutusdataa, jonka avulla luodaan dataa kuvaava optimoitu funktio. Yleisimmin käytetty koneoppimisen menetelmä oli ohjattu oppiminen, jossa koulutusdata muodostuu oikeiksi määritellyistä input/output pareista (Jordan & Mitchell, 2015). Useita tutkimuksia koneoppimisen käyttämisestä haisevan koodin tunnistamiseksi esitettiin. Tutkimukset osoittivat, että koneoppiminen soveltuu haisevan koodin tunnistamiseen (Fontana et al., 2016, Imazato et al., 2017). Parhaiten testeissä suoriutuivat päätöspuuhun perustuvat algoritmit (Imazato et al., 2017, Yue et al., 2018). Tärkeää on myös tapa, miten algoritmit koulutetaan (Fontana et al., 2016). Tutkimuksissa havaittiin myös rajoituksia koneoppimisen käyttämisessä, ja tarvitaan lisää tutkimusta aiheesta (Di Nucci et al., 2018).

Lähteet

AlOmar, E. A., Peruma, A., Mkaouer, M. W., Newman, C., Ouni, A., & Kessentini, M. (2021). How we refactor and how we document it? On the use of supervised machine learning algorithms to classify refactoring documentation. *Expert Systems with Applications*, 167, 114176.

Aniche, M., Maziero, E., Durelli, R., & Durelli, V. (2020). The effectiveness of supervised machine learning algorithms in predicting software refactoring. *IEEE Transactions on Software Engineering*.

Bavota, G., De Carluccio, B., De Lucia, A., Di Penta, M., Oliveto, R., & Strollo, O. (2012). When does a refactoring induce bugs? An empirical study. *Proceedings - 2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation, SCAM 2012,* pp. 104-113. https://doi.org/10.1109/SCAM.2012.20

Cedrim, D., Garcia, A., Mongiovi, M., Gheyi, R., Sousa, L., de Mello, R., Fonseca, B., Ribeiro, M. & Chavez, A. (2017). Understanding the Impact of Refactoring on Smells: A Longitudinal Study of 23 Software Projects. *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Volume Part F130154, 21 August 2017*, pp. 465-475.

Di Nucci, D., Palomba, F., Tamburri, D. A., Serebrenik, A., & De Lucia, A. (2018). Detecting code smells using machine learning techniques: are we there yet?. In *2018 IEEE 25th international conference on software analysis, evolution and reengineering (saner)* (pp. 612-621).

Fontana, F. A., Mäntylä, M. V., Zanoni, M., & Marino, A. (2016). Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering*, 21(3), 1143-1191. http://dx.doi.org/10.1007/s10664-015-9378-4

Fontana, F. A. & Zanoni, M. (2017). Code smell severity classification using machine learning techniques. *Knowledge-Based Systems* (128), pp. 43-58.

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code.* Boston: Addison-Wesley.

Imazato, A., Higo, Y., Hotta, K., & Kusumoto, S. (2017). Finding extract method refactoring opportunities by analyzing development history. *IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)* (Vol. 1, pp. 190-195).

Jain, S. & Saha, A. (2019). An Empirical Study on Research and Developmental Opportunities in Refactoring Practices. *Proceedings of the International Conference on Software Engineering and Knowledge Engineering, SEKE, 2019-July*, pp. 313-318.

Jain, S. & Saha, A. (2021). Improving performance with hybrid feature selection and ensemble machine learning techniques for code smell detection. *Science of Computer Programming,* 212. https://doi.org/10.1016/j.scico.2021.102713

Jordan, M. I., & Mitchell, T. M. (2015). Machine learning: Trends, perspectives, and prospects. *Science*, 349(6245), 255-260.

Kim, M., Zimmermann, T., & Nagappan, N. (2012). A field study of refactoring challenges and benefits. *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering.* https://doi.org/10.1145/2393596.2393655

Kim, M., Zimmermann, T., & Nagappan, N. (2014). An Empirical Study of Refactoring Challenges and Benefits at Microsoft. *IEEE Transactions on Software Engineering, 40(7)*, pp. 633-649. https://doi.org/10.1109/TSE.2014.2318734

Lacerda, G., Petrillo, F., Pimenta, M. & Guéhéneuc, Y. (2020). Code Smells and Refactoring: A Tertiary Systematic Review of Challenges and Observations. *Journal of Systems and Software*, *167(9)*. https://doi.org/10.1016/j.jss.2020.110610

Lahtinen, S. & Leppänen, M. (2016). Refactoring Patterns, Practices for Daily Work. *10th Travelling Conference on Pattern Languages of Programs, VikingPLoP 2016.* https://doi.org/10.1145/3022636.3022642

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, 521(7553), 436-444.

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

Leppänen, M., Lahtinen, S., Kuusinen, K., Mäkinen, S., Männistö, T., Itkonen, J., Yli-Huumo, J., & Lehtonen, T. (2015 b) Decision-making framework for refactoring. *2015 IEEE 7th International Workshop on Managing Technical Debt,* pp. 61-68. https://doi.org/10.1109/MTD.2015.7332627

Liu, H., Gao, Y. & Niu, Z. (2012). An Initial Study on Refactoring Tactics. *2012 IEEE 36th International Conference on Computer Software and Applications.* https://doi.org/10.1109/COMPSAC.2012.31

Liu, H., Jin, J., Xu, Z., Zou, Y., Bu, Y. & Zhang, L. (2021). Deep Learning Based Code Smell Detection. *IEEE Transactions on software engineering,* 47(9). https://doi.org/10.1145/3238147.3238166

Martin, R, C. (2019). *Clean Code - Uncle Bob / Lesson 1.* Youtube. 9.8.2019. https://www.youtube.com/watch?v=7EmboKQH8lM, katsottu 1.10.2021.

Mens, T. & Tourwe, T. (2004). A Survey of Software Refactoring. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING 30(2)*, pp. 126-139. https://doi.org/10.1109/TSE.2004.1265817

Murphy-Hill, E. & Black, A. (2008). Refactoring Tools: Fitness for Purpose. *IEEE Software, 25(5)*, pp. 38-44. https://doi.org/10.1109/MS.2008.123

Mäntylä, M. & Lassenius, C. (2006). Drivers for software refactoring decisions. *Proceedings of the 5th ACM-IEEE International Symposium on Empirical Software Engineering 2006*, pp. 297-306. https://doi.org/10.1145/1159733.1159778

Rachatasumrit, N. & Miryung, K. (2012). An Empirical Investigation into the Impact of Refactoring on Regression Testing. *2012 28th IEEE International Conference on Software Maintenance (ICSM)*

Rebai, S., Alizadeh, V., Kessentini, M., Fehri, H., & Kazman, R. (2020). Enabling decision and objective space exploration for interactive multi-objective refactoring. *IEEE Transactions on Software Engineering*.

Sagi, O., & Rokach, L. (2018). Ensemble learning: A survey. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 8(4), e1249.

Sharma, T., Suryanarayana, G., Samarthyam, G. (2015). Challenges to and Solutions for Refactoring Adoption: An Industrial Perspective. *IEEE Software 32(6)*, pp. 44-51. https://doi.org/10.1109/MS.2015.105

Tempero, E., Gorschek, T., & Angelis, L. (2017). Barriers to refactoring. *Communications of the ACM, 60(10)*, pp. 54-61. https://doi.org/10.1145/3131873

Tsantalis, N., Mansouri, M., Eshkevari, L., Mazinanian, D., & Dig, D. (2018). Accurate and efficient refactoring detection in commit history. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)* (pp. 483-494).

Yue, R., Gao, Z., Meng, N., Xiong, Y., Wang., X. & Morgenthaler, J., D. (2018). Automatic Clone Recommendation for Refactoring Based on the Present and the Past. *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. https://doi.org/10.1109/ICSME.2018.00021