ROOSA VIHLMAN

Koodipohjan refaktorointi ja ylläpi-dettävyys

SÄHKÖ- JA AUTOMAATIOTEKNIIKAN KOULUTUSOHJELMA

2022

<table>
<tr><td rowspan="2">Tekijä(t)
Vihlman, Roosa</td><td>Julkaisun laji
Opinnäytetyö, AMK</td><td>Päivämäärä
maaliskuu 2022</td></tr>
<tr><td>Sivumäärä
51</td><td>Julkaisun kieli
suomi</td></tr>
<tr><td colspan="3">Julkaisun nimi
Koodipohjan refaktorointi ja ylläpidettävyys</td></tr>
<tr><td colspan="3">Tutkinto-ohjelma
Sähkö- ja automaatiotekniikan koulutusohjelma</td></tr>
</table>

Tiivistelmä

Tämä opinnäytetyö suoritettiin Profit Software Oy:n toimeksiantona. Opinnäytetyössä käsiteltiin koodipohjan refaktorointia ja ylläpidettävyyttä sekä näiden välistä yhteyttä. Ohjelmistotuotteen elinkaaren vaiheista ylläpitovaihe on pisin, minkä vuoksi koodin laadullisista ominaisuuksista ylläpidettävyys on merkittävä tekijä. Opinnäytetyössä selvitettiin, millaisilla tavoilla refaktorointi vaikuttaa sekä koodin ylläpidettävyyteen että ohjelmistokehityksen eri osa-alueisiin. Työssä nostettiin esille myös automaattisten refaktorointityökalujen maine ohjelmistokehittäjien keskuudessa sekä syitä skeptiselle suhtautumiselle työkaluja kohtaan.

Opinnäytetyön toiminnallinen osuus ja tulosten analysointi toteutettiin Profit Softwarella tehtyjen refaktorointien pohjalta. Toiminnallisessa osuudessa käytiin yhdessä esimerkkien kanssa läpi, miten refaktorointia voidaan toteuttaa koodissa ja kuinka moninaista refaktorointi voi olla. Tuloksissa peilataan refaktorointien vaikutusta koodin ylläpidettävyyteen ja analysoidaan, saavuttivatko Profit Softwarella tehdyt refaktoroinnit niille asetetut tavoitteet.

Johtopäätöksissä todettiin, että refaktorointi olisi hyvä nostaa esille jo ohjelmiston kehitysvaiheessa. Koodia olisi hyvä refaktoroida aina silloin, kun sille on tarve ja mahdollisuus. Kun tarpeellinen refaktorointi otetaan huomioon jo riittävän ajoissa, voidaan välttyä ylimääräiseltä ajan ja vaivan käytöltä tulevaisuudessa.

Avainsanat
koodin laatu, refaktorointi, tekninen velka, ylläpito, ylläpidettävyys

Author(s) Vihlman, Roosa	Type of Publication Bachelor’s thesis	Date March 2022
	Number of pages 51	Language of publication Finnish

Title of publication
Code refactoring and maintainability

Degree programme
Electrical and Automation Engineering

Abstract

This thesis was commissioned by Profit Software Ltd. This thesis focused on code refactoring and maintainability and the connection between these topics. Maintenance phase is the longest phase of software product’s lifecycle which is the reason why maintainability plays significant role in code quality properties. In this thesis was investigated in which ways refactoring can affect on both code maintainability and different areas of software development. There were also noted the reputation of automatic refactoring tools and reasons why software developers act as skeptical as they do towards these tools.

The practical part of this thesis and result analyzing were based on refactoring made in Profit Software. In the practical part there were introduced together with examples how refactoring can be done in code and how diverse it can be. Effects of refactoring are mirrored in results and analyzed if the refactorings made in Profit Software achieved the goals which were set to them.

There were stated in the conclusions of this thesis that refactoring should be emphasized already in the phase of software development. Code would be good to be refactored when ever it is necessary and there is chance to refactor it. Waste of time and effort can be saved when necessary refactoring is taken noticed in sufficient time.

Keywords
code quality, refactoring, technical debt, maintenance, maintainability

SISÄLLYS

1 JOHDANTO 6
2 KOODIN YLLÄPITO 7
2.1 SDLC 7
2.2 Koodin ylläpito 8
2.3 Huomioita koodin ylläpidon haasteista 10
3 KOODIN LAATU JA YLLÄPIDETTÄVYYS 13
4 REFAKTOROINTI 15
4.1 Mistä refaktorointi tulee? 15
4.2 Refaktoroinnin vaikutusalueita 16
4.2.1 Vaikutus koodiin ja koodin ylläpidettävyyteen 16
4.2.2 Vaikutus taloudellisesti 17
4.2.3 Vaikutus tietoturvaan 18
4.3 Refaktoroinnin automatisointi 19
4.4 Refaktoroinnin onnistumisen arviointi 21
5 ONGELMAN KUVAUS JA LÄHTÖKOHDAT 22
6 ONGELMAN ERITTELY JA RATKAISUT 24
6.1 Any-tyypityksen eliminointi 24
6.2 React Inlt-kirjaston päivitys 28
6.3 Indeksi-avaimen eliminointi 31
6.4 Moment.js-kirjaston korvaus 35
6.5 useFormik()-funktion eliminointi 38
7 REFAKTOROINNIN LOPPUTULOS 44
8 POHDINTAA 46

LÄHTEET

SYMBOLI- JA LYHENNELUETTELO

Backend-koodi	Backend-koodi on ohjelmiston palvelinpuoli ja hoitaa kaiken sen toiminnallisuuden, jota käyttäjä ei näe.
Frontend-koodi	Frontend-koodilla tehdään se ohjelmiston osa, jonka käyttäjä näkee ja jonka kautta käyttäjä on vuorovaikutuksessa ohjelmiston kanssa.
Hookki	Hookki tulee englannin kielen sanasta ”hook”. Hookkeja käytetään React-ohjelmointikielessä ja ne mahdollistavat muun muassa tilojen ja metodien yksinkertaisen käytön ilman luokkien tekoa.
IntelliJ	JetBrains-ohjelmistokehitysyrityksen luoma ohjelmistokehitysympäristö.
JSON-tiedosto	JavaScript Object Notation -tiedosto.
Käyttöliittymä	Ohjelmiston osa, joka on tehty frontend-koodilla.
Lokalisointi	Tekstin ominaisuudet on muokattu vastaamaan sitä kieltä, jolla sitä käytetään. Esimerkiksi kielen käännökset ja päivämäärien oikeat muodot.
Parametri	Esimerkiksi komponentille tai metodille syötettävä tieto.
Span-elementti	Tekstitason elementti.
Wrapata	Wrapata tulee englannin kielen sanasta ”wrap”. Komponentteja voidaan wrapata eli kääriä tai paketoida koodissa toisen komponentin tai funktion sisään.

1 JOHDANTO

Koodipohja saattaa toisinaan olla erittäin pitkäikäinen. Jotta koodia pystytään sen elin-iän aikana ylläpitämään mahdollisimman vaivattomasti, on koodin ylläpidettävyydellä suuri vaikutus. Koodin ylläpidettävyys olisi hyvä ottaa aktiivisesti huomioon koko ohjelmiston elinkaaren aikana. Erilaisten toimintatapojen avulla koodin ylläpidettävyyttä voidaan parantaa, kuten esimerkiksi refaktoroimalla.

Refaktorointi on ohjelmiston sisäisen rakenteen muutosta siten, että sen toiminnallisuus ei muutu. Refaktoroinnin avulla koodia on helpompi ymmärtää ja koodin muokkaaminen voi jatkossa olla taloudellisesti edullisempaa. (Fowler 1999, 53.) Refaktoroinnin päätarkoitus on vähentää koodin teknistä velkaa (Pal 2021). Koodin refaktorointi on osa koodin ylläpitoa ja se voidaan lukea kuuluvaksi ennaltaehkäisevään ylläpitoon. Refaktorointi ei kuitenkaan ole virhekorjausta, vaan sen tarkoituksena on kehittää koodia paremmaksi, jotta virheiltä voitaisiin välttyä tulevaisuudessa (Data Respons n.d.). Refaktorointi on arvokas työväline, jota voi käyttää useisiin tarkoituksiin ja joka auttaa ohjelmistokehittäjää pitämään hyvän otteen koodista (Fowler 1999, 55).

Opinnäytetyön toimeksiantajana toimii Profit Software, joka on vuonna 1992 perustettu ohjelmistoyritys. Profit Software tarjoaa moderneja järjestelmäratkaisuja pohjoismaisille pankeille ja vakuutusyhtiöille. Profit Softwarella työskentelee tällä hetkellä noin 300 työntekijää ja sillä on toimipaikkoja Espoossa, Lahdessa, Lappeenrannassa, Porissa, Tampereella, Tallinnassa ja Tukholmassa. Erilaisia yksikköjä Profit Softwarella on neljä: Pension & Insurance Solutions, Financial Sector Solutions, Business Intelligence & Analytics sekä Management & Administration. Profit Softwaren liikevaihto on noin 30 miljoonaa euroa ja Profit Softwaren asiakkaisiin kuuluu muun muassa Aktia, Elo, Lähitapiola, Länsförsäkringar, Mandatum Life, OP ja Säästöpankki.

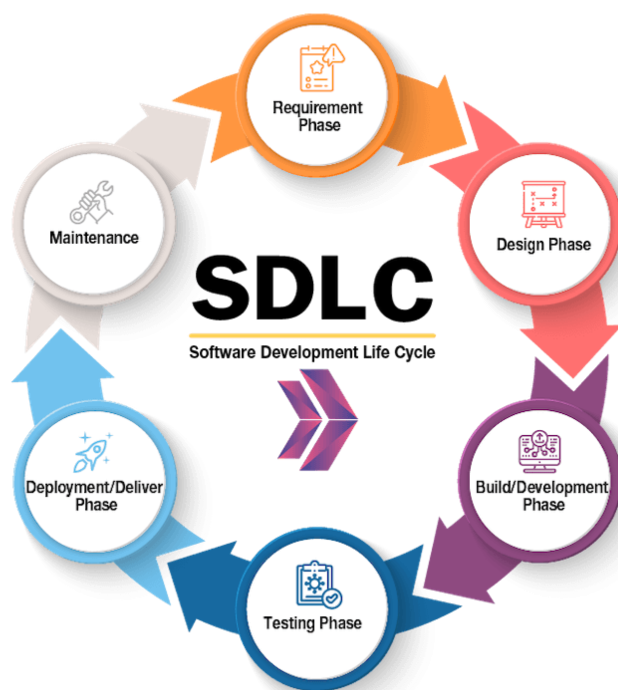
Profit Software käyttää työvälineinään automatisaatiota, ketteriä menetelmiä, modernia ohjelmistokehitystä sekä dataa ja analytiikkaa. Profit Softwaren työnkuvaan ja palveluihin kuuluu muun muassa tiedolla johtaminen, asiantuntijapalvelut ja konsultointi, sovellushallinta, perusjärjestelmät ja tuotteistetut ratkaisut sekä SaaS ja pilvipohjaiset ratkaisut.

2 KOODIN YLLÄPITO

Koodin ylläpidon edellytyksenä on koodin laadullinen kyky olla ylläpidettävissä. Koodin ylläpito on moninaista ja olennainen osa ohjelmistokehitystä sen jokaisella osa-alueella. Koodin ylläpitoa voi olla esimerkiksi virhekorjaukset ja ohjelmiston parannukset. Koodin ylläpito on yleensä pisin ajanjakso ohjelmiston elinkaaresta ja koodin ylläpito voi jatkua jopa vuosikymmenien ajan. Jotta ohjelmisto voidaan pitää relevanttina, käytettävänä ja tuottavana, sitä pitää ylläpitää (Kuipers 2016).

2.1 SDLC

Teknisillä prosesseilla kuvataan toimintoja, joiden avulla teknisten päätösten ja tekojen kautta nousseita etuja voidaan optimoida ja riskejä voidaan vähentää organisaatiossa ja projektitehtävissä (SFS-ISO/IEC/IEEE 12207:en 2020, 62). Ohjelmistokehityksessä teknistä prosessia kutsutaan SDLC:ksi. SDLC on lyhenne englannin kielen sanoista ”systems development life cycle” tai ”software development life cycle”, eli suomeksi ohjelmistokehityksen elinkaari. SDLC on prosessi, jonka avulla voidaan saattaa ohjelmisto suunnittelutasolta eri vaiheiden kautta valmiiksi lopputuotteeksi. (Preston 2021.) SDLC:n vaiheita on yleensä n. kuusi kappaletta: vaatimusmäärittely-, suunnittelu-, kehittämis-, testaus-, toimitus- ja ylläpitovaihe. SDCL:n vaiheet on esitetty kuvassa 1. Tämän opinnäytetyön osalta SDLC:n vaiheista olennaisin osa on ”maintenance” eli ylläpito ja sen vuoksi SDLC:n muita vaiheita ei erikseen esitellä tässä työssä.



Kuva 1. SDLC:n kehitysvaiheet (Clarusway 2021)

2.2 Koodin ylläpito

Koodin ylläpitoprosessin tarkoituksena on ylläpitää ohjelmiston kykyä tarjota palveluita eli ylläpitää koodin toimintaa. Ylläpitoprosessi tekee korjauksia, muutoksia ja kehityksiä käyttöön otetulle ohjelmistolle. Ohjelmiston ylläpidon tarve voi ilmetä useista eri syistä, kuten piilevien virheiden, tietoturvallisuuden parantamisen tai ohjelmiston elementtien vanhenemisen vuoksi. Ohjelmistojärjestelmien elementtien ylläpito voi sisältää laitteiston, ohjelmiston ja palveluiden, kuten web-palveluiden, ylläpitoa. (SFS-ISO/IEC/IEEE 12207:en 2020, 105.)
Koodia voidaan ylläpitää

- ennaltaehkäisevästi
- mukauttavasti
- parantelevasti
- täydellistävästi.

(ISO/IEC/IEEE 14764:en 2006.)

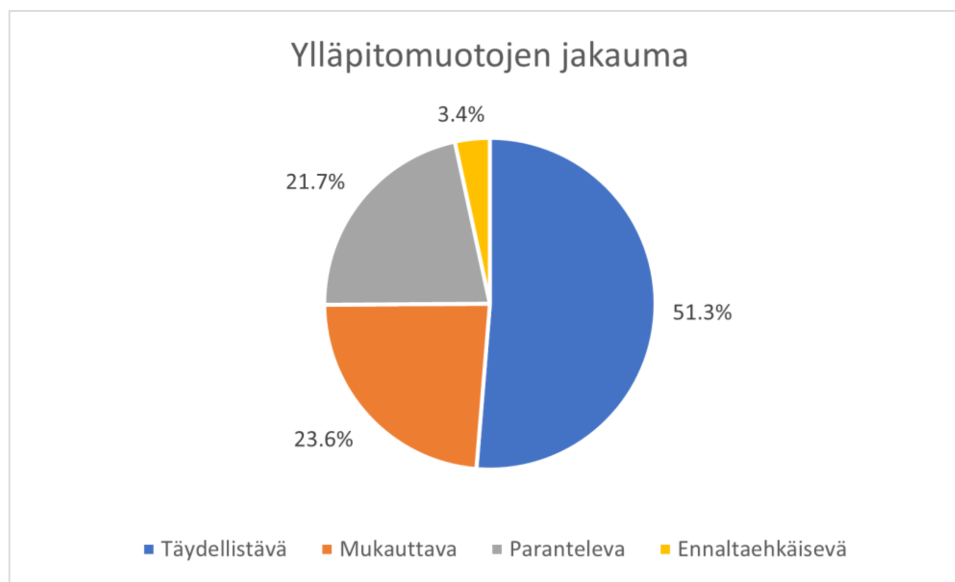
Ennaltaehkäisevä ylläpito pyrkii säilyttämään koodin ylläpidettävyyden tulevaisuudessa ja tarjoamaan hyvän pohjan tuleville koodimuutoksille (Singh & Goel 2007). Ennaltaehkäisevän ylläpidon tarkoituksena on havaita ja korjata piileviä ongelmia ennen kuin ongelmista muodostuu operatiivisia (Grubb & Takang 2003; ISO/IEC/IEEE 14764:en 2006, viitattu lähteessä Computer Society 2014, 107). Ennaltaehkäisevä ylläpito keskittyy estämään koodin huonontumisen, kun koodi muuttuu ja mukautuu. Ennaltaehkäisevän ylläpidon avulla voidaan koodista saada vakaampi, ymmärrettävämpi ja ylläpidettävämpi. (CAST 2021.)

Mukauttavaa ylläpitoa tehdään usein silloin, kun ohjelmiston ympäristö muuttuu. Mukauttavan ylläpidon tarkoituksena on pitää ohjelmisto käytettävänä muuttuneessa tai muuttuvassa ympäristössä. (Grubb & Takang 2003; ISO/IEC/IEEE 14764:en 2006, viitattu lähteessä IEEE Computer Society 2014, 107.) Muutoksia voi olla esimerkiksi käyttöjärjestelmän tai ohjelmiston ominaisuuksien muutokset (CAST 2021). Myös ohjelmiston ympäristön ulkopuoliset asiat voivat vaikuttaa ohjelmistossa ja näin ollen koodissa tehtävään mukauttavaan muutokseen. Tällaisia syitä voi olla esimerkiksi GDPR-asetus eli yleinen tietosuoja-asetus. (Omeyer 2021.)

Parantelevan ylläpidon tarkoituksena on korjata virheitä, jotka ilmenevät ohjelmiston käyttöönoton jälkeen (Singh ym. 2007). Paranteleva ylläpito keskittyy käytännössä virhekorjauksiin tai ammattikielellä ”bugikorjauksiin”. Parantelevaa ylläpitoa tarvitaan, kun esimerkiksi ohjelmisto ei toimi halutulla tavalla, koska koodin logiikassa on virhe. Yleensä ohjelmiston käyttäjät tekevät ohjelmiston virheellisestä toiminnasta virheilmoitukset, jotka ylläpidosta vastaavat ohjelmistokehittäjät katselmoivat. Katselmoinnin jälkeen ohjelmistokehittäjät tekevät koodiin tarpeelliset korjaavat muutokset. Mikäli ohjelmistossa tai sen toiminnassa havaitaan virheitä ennen käyttäjää, luetellaan virhekorjaus ennaltaehkäiseväksi tai mukauttavaksi ylläpidoksi. (Merrill 2019.)

Täydellistävä ylläpito keskittyy ohjelmiston vaatimusten ja ominaisuuksien kehittämiseen. Täydellistävä ylläpito voi olla ohjelmiston parantelua, kuten uusien ominaisuuksien lisäämistä tai tarpeettomien ominaisuuksien poistamista, ohjelmiston dokumentaation kehittämistä tai uudelleenkoodausta, jonka tarkoituksena on parantaa koodin toimintaa tai ylläpidettävyyttä. (Grubb & Takang 2003, viitattu lähteessä IEEE Computer Society 2014; ISO/IEC/IEEE 14764:en 2006, viitattu lähteessä IEEE

Computer Society 2014, 107; CAST 2021.) Uudelleenkoodauksella voidaan tarkoittaa koodin muokkaamista siten, että koodin sisäinen rakenne ja koodin toiminnallisuus muuttuu tai siten, että koodin sisäinen rakenne muuttuu, mutta toiminnallisuus pysyy samana. Jälkimmäistä tapaa kutsutaan refaktoroinniksi. Täydellistävän ylläpidon avulla pyritään vastaamaan asiakkaan vaatimuksia ohjelmiston ominaisuuksista tai toiminnasta. Täydellistävää ylläpitoa ovat muun muassa käyttöliittymän, ohjelmiston käytettävyyden ja toiminnallisuuden parannukset. (Merrill 2019.) Täydellistävä ylläpito on ylläpitomuodoista yleisin (Lientz & Swanson 1980, viitattu lähteessä Bird 2011).



Kuvio 1. Koodin ylläpitomuotojen jakauma (Lientz & Swnason 1980, viitattu lähteessä Bird 2011)

2.3 Huomioita koodin ylläpidon haasteista

Koodin elinikä voi olla pitkä, toisinaan jopa vuosikymmeniä. Koodin pitkän iän aikana ohjelmistokehittäjät saattavat vaihtua useasti. Mikäli koodin ylläpidettävyyteen ei ole kiinnitetty tarpeeksi huomiota aiemmin, saattaa se tuottaa rajoitteita koodin ymmärtämisessä tulevaisuudessa. Rajoittunut ymmärryskyky koodin toiminnasta viittaa siihen, kuinka nopeasti ohjelmistokehittäjä pystyy ymmärtämään sellaisen koodin rakennetta, jota hän ei ole itse kehittänyt (Grubb & Takang 2003; ISO/IEC/IEEE 14764:en 2006, viitattu lähteessä IEEE Computer Society 2014, 107-108).

Hyvällä koodin toiminnan dokumentaatiolla voidaan helpottaa sekä uusia että vanhoja ohjelmistokehittäjiä ymmärtämään koodin toimintaa. Mikäli koodin toimintaa halutaan dokumentoida, olisi se hyvä aloittaa jo koodin luontivaiheessa. Tämä saattaa silti olla haasteellista, sillä ohjelmistokehityksen varhainen vaihe, eli ohjelmiston kehittämisprosessi, on yleensä projektipohjainen, aikaskaalattu ja budjetoitu. Mikäli projektin aikataulussa tai budjetissa ei ole erikseen otettu huomioon koodin dokumentointia, ohjelmistokehittäjillä ei välttämättä ole aikaa tehdä dokumentaatiota tai siihen ei ole resursseja, eli toisin sanoen ohjelmistokehittäjien tulisi tehdä dokumentointi vapaaehtoisesti. Vaihtoehtoisesti koodin dokumentoinnin voi myös tehdä vasta koodin kehittämisen jälkeen, mutta mikäli sen tekee joku muu kuin koodin alkuperäinen tekijä, joutuu dokumentaation tekijä ensin selvittämään, miten koodi toimii. Jos dokumentaatio halutaan pitää ajan tasalla, tulisi sitä päivittää aina samassa yhteydessä, kun koodi päivittyy. Dokumentointi itsessään saattaa olla haasteellista, sillä toisinaan koodin toiminnan selittäminen sanallisesti voi olla vaikeaa. Vaikka dokumentoinnin tarkoitus on helpottaa ja nopeuttaa ohjelmistokehittäjien työtä, on koodin dokumentointi edellä mainittujen seikkojen vuoksi melko työläs, resursseja vaativa ja haasteellinen toimenpide.

Eräs koodin ylläpidon haasteita on koodin ulkoasu. Hilton (2018) avaa blogikirjoituksessaan, kuinka koodin konkreettisten ylläpitotehtävien haasteet keskittyvät ohjelmistokehittäjien mukaan usein ensisijaisesti koodiin itseensä, kuten esimerkiksi koodin elementit on huonosti nimetty, koodi on monimutkaista eikä koodia ole kirjoitettu johdonmukaisesti. Koodia työstettäessä tulisi pitää erityisesti mielessä sen yksinkertaisuus (Dooley 2017). Saman koodipohjan parissa voi työskennellä yhtäaikaisesti useita ohjelmistokehittäjiä, jolloin koodin yksinkertainen ja ymmärrettävä rakenne nostaa arvoaan, sillä nämä koodin ominaisuudet mahdollistavat koodin tehokkaan ja helpon ylläpidon. Koodin järkevällä rakenteella voidaan varmistaa sen helppo ylläpidettävyys tulevaisuudessa.

Koodin ulkoasuun liittyvät ylläpidon haasteet saattavat kuitenkin olla vaikeita kehityskohteita. Jokaisella ohjelmistokehittäjällä on pääasiassa oma tapa koodata. Toisen kirjoittama koodi saattaa olla toiselle liian monimutkainen ja komponenttien nimeämiset saattavat esiintyä huonoina. Koska koodin ulkoasulliset haasteet saattavat olla subjektiivisia, on erittäin olennaista, että ohjelmistokehitystiimeillä tai yrityksillä on ennalta

määritellyt raamit ja yhdenmukaiset tavat koodin kirjoittamiselle. Yhteisillä ohjeilla voidaan koodista tehdä johdonmukaisempaa ja subjektiivisia haasteita pystytään minimoimaan esimerkiksi yhdenmukaisella komponenttien nimeämistavalla tai määrittelemällä perusrakenteen, jonka mukaan luokkakomponentit on koodattava.

Yksi haastava ongelma ohjelmiston ja koodin ylläpidossa on selvittää uusien koodimuutosten vaikutus muuhun koodiin. Koodimuutosten vaikutusta ja yllättäviä sivuvaikutuksia voidaan arvioida ja selvittää vaikutusanalyysin avulla. (Gupta ym. 2015.) Vaikutusanalyysi voi auttaa hahmottamaan ne koodin osat, joihin tehtävä koodimuutos mahdollisesti aiheuttaa virheitä ja arvioimaan resurssit, jotka muutokseen tarvitaan. Näin voidaan myös kuvata, kuinka koodimuutokset voidaan suorittaa taloudellisesti mahdollisimman tehokkaasti. Vaikutusanalyysin tekeminen vaatii hyvää tietämystä koodin rakenteesta ja sen sisällöstä. (Grubb & Takang 2003; ISO/IEC/IEEE 14764:en 2006, viitattu lähteessä Computer Society 2014, 108.)

Ohjelmiston elinkaaren kuluista ohjelmiston ylläpidon on tutkittu vievän 60-80 % (Gupta ym. 2015). Ymmärtämällä ohjelmiston ylläpidon eri kategoriat ja tekijät, jotka vaikuttavat ohjelmiston ylläpitoon, voivat helpottaa hahmottamaan ohjelmiston ylläpitokulujen rakennetta. Yleinen käsitys on, että ohjelmiston ylläpito on pääasiassa virheiden korjausta. Vuosien saatossa tehdyt tutkimukset kuitenkin osoittavat, että yli 80 % ohjelmiston ylläpitoon käytetystä ajasta kuluu niin sanottuihin ei-korjaaviin ylläpitotehtäviin, kuten esimerkiksi ohjelmiston parannuksiin. (Grubb & Takang 2003, viitattu lähteessä IEEE Computer Society 2014, 106.)

3 KOODIN LAATU JA YLLÄPIDETTÄVYYS

Koodin laatua voidaan arvioida monesta eri näkökulmasta. Ohjelmistokehittäjälle ja ohjelmiston loppukäyttäjälle koodin laatu saattaa tarkoittaa eri asioita. Loppukäyttäjälle koodin hyvä laatu voi tarkoittaa ohjelmiston toimimista vaatimusten mukaisesti, kun taas ohjelmistokehittäjälle koodin hyvä laatu voi tarkoittaa koodin helppolukuisuutta ja selkeää rakennetta. Näkökulmasta huolimatta tulisi laadulliset ominaisuudet ottaa huomioon, jotta ohjelmisto ja sen koodi palvelisi kaikkia käyttäjiään mahdollisimman hyvin.

Koodin laadukkuutta voi olla toisinaan vaikea määritellä, mutta koodin laadukkuudesta voidaan tehdä huomioita. Esimerkiksi mikäli työskentely koodin parissa vaatii paljon vaivannäköä ja aikaa tai koodin toiminnallisuudessa ilmenee jatkuvasti ongelmia, saattaa koodin laadussa olla puutteita. Laadukkaan koodin kirjoittaminen tulisi olla yksi tärkeimmistä ja välttämättömistä tavoitteista ohjelmistokehityksessä. Laadukkaan koodin avulla koodista tulee luettavampaa, siistimpää, testattavampaa ja sillä on pienempi taipumus lisätä virheiden määrää (Cleverti 2018). Koodin laatuun vaikuttavia tekijöitä on useita. Koodin laatuun vaikuttavat sisäiset ja ulkoiset tekijät voidaan jakaa kuuteen isompaan kategoriaan:

- käytettävyys
- luotettavuus
- siirrettävyys
- tehokkuus
- toiminnallisuus
- ylläpidettävyys.

(ISO/IEC 9126:en 1988, viitattu lähteessä Spinellis 2006, 4.)

Tämän opinnäytetyön tarkoituksena on keskittyä koodin laadullisista ominaisuuksista ylläpidettävyyteen, minkä vuoksi muita koodin laatuominaisuuksia ei tässä yhteydessä esitellä tarkemmin.

Verrattaessa muihin koodin laatuominaisuuksiin, *koodin ylläpidettävyyttä* on Spinelliksen (2006, 7) mukaan helpointa lähestyä koodin suunnittelu- ja kooditasolla. Koodin ylläpidettävyyden voidaan määritellä olevan koodin kykyä olla muokattavissa (ISO/IEC/IEEE 14764:en 2006, viitattu lähteessä IEEE Computer Society 2014, 108).

Spinellis mainitsee muokattavuuden lisäksi kolme muuta ominaisuutta, jotka ovat osa koodin ylläpidettävyyttä: analysoitavuus, vakaus ja testattavuus. Analysoitavuudella tarkoitetaan kykyä hahmottaa koodia ja tunnistaa halutut ja etsityt kohdat koodista esimerkiksi virhekorjausten yhteydessä. Vakaudella tarkoitetaan sitä, että koodimuutokset eivät johda muun kuin muokatun koodin toiminnallisuuden muutoksiin. Testattavuudella tarkoitetaan kykyä validoida tehdyt muutokset.

IEEE Computer Societyn (Grubb & Takang 2003; ISO/IEC/IEEE 14764:en 2006, viitattu lähteessä IEEE Computer Society 2014, 108) teoksen mukaan koodin ylläpidettävyyden olisi hyvä olla määritelty osa ohjelmistokehitystä ja ylläpidettävyyttä olisi hyvä läpikäydä sekä kontrolloida ohjelmistokehitysaktiviteettien aikana, jotta ylläpitokustannuksia voitaisiin vähentää. Toisin sanoen, koodin ylläpidettävyys tulisi ottaa huomioon jo ohjelmistokehityksen aikana. Ohjelmistokehitysprosessin aikana ylläpidettävyyden kaltaiset koodin ominaisuudet ovat kuitenkin harvemmin keskiössä, minkä vuoksi ylläpidettävyyttä saattaa olla vaikea saavuttaa ohjelmistokehityksen aikana. Ylläpidettävyyttä edistävien aktiviteettien sivuuttaminen saattaa esiintyä puutteina esimerkiksi koodin toiminnan dokumentaatiossa tai luettavuudessa. Koodin ylläpidettävyyttä voidaan parantaa huolellisten ja systemaattisten työkalujen ja tekniikoiden, kuten refaktoroinnin, avulla.

4 REFAKTOROINTI

4.1 Mistä refaktorointi tulee?

Ensimmäiset ohjelmistokehittäjät, jotka kiinnittivät huomiota refaktoroinnin tärkeyteen, olivat Ward Cunningham ja Kent Beck. Cunningham ja Beck työskentelivät Smalltalk-ohjelmointikielen parissa 1980-luvulta eteenpäin ja heillä oli vahva vaikutus Smalltalk-yhteisössä. Smalltalk on dynaaminen olio-ohjelmointikieli, joka mahdollistaa erittäin toiminnallisen ohjelmiston kirjoittamisen nopeasti. Cunningham ja Beck tekivät kovasti töitä kehittääkseen ohjelmistokehitysprosessin, joka toimii Smalltalkin kaltaisessa ympäristössä. Cunningham ja Beck ymmärsivät, että refaktorointi oli tärkeää heidän produktiivisuutensa kannalta ja siitä lähtien he hyödynsivät refaktorointia työtehtävissään. (Fowler 1999, 71.)

Ward Cunninghammin ja Kent Beckin lisäksi Ralph Johnson oli yksi vaikuttava henkilö Smalltalk-yhteisössä. Johnson tutki, millä tavoin refaktorointi voisi auttaa tehokkaan ja joustavan ohjelmistokehyksen (engl. software framework) kehityksessä. Bill Opdyke oli yksi Johnsonin tohtoriopiskelijoista ja hän oli sitä mieltä, että refaktorointia voisi hyödyntää Smalltalkin lisäksi monessa muussa yhteydessä, kuten esimerkiksi C++-ohjelmointikielen ohjelmistokehyksen kehityksessä. (Fowler 1999, 71.)

John Brant ja Don Roberts veivät idean refaktorointityökaluista pidemmälle kehittääkseen Refactoring Browserin, joka oli refaktorointityökalu Smalltalkille (Fowler 1999, 72). Brantin ja Robertsin (1999, 401) mukaan refaktorointityökalulla tarkoitetaan työkalua, joka refaktoroi koodia automaattisesti. Manuaalinen refaktorointi on Brantin ja Robertsin mukaan aikaa vievää, mikä estää ohjelmistokehittäjiä tekemästä niitä refaktorointeja, joita heidän pitäisi tehdä, sillä refaktorointi yksinkertaisesti maksaa liikaa.

Martin Fowler (1999, 72) kuvailee teoksessaan, kuinka hän on aina ollut halukas siistimään koodia, mutta ei ole ikinä ajatellut sen olevan niin tärkeää. Työskenneltyään Kent Beckin kanssa Fowler näki, miten Beck käytti refaktorointia ja vakuuttui refaktoroinnin vaikutuksista. Koska refaktoroinnista kertovia kirjoja ei vielä ollut, eikä kokeneilla ohjelmistokehittäjillä ollut tarkoituksena sellaista kirjoittaa, päätti Fowler heidän avullaan kirjoittaa refaktoroinnista kirjan.

4.2 Refaktoroinnin vaikutusalueita

Huonolaatuisesta koodista käytetään termiä ”code smells”, eli koodi haisee. Termi on saanut alkunsa Kent Beckin äidiltä, joka ohjeisti Beckiä vauvan vaipanvaihdon suhteen. ”Jos se haisee, vaihda se”, sanoi Beckin äiti ja Beck alkoi hyödyntämään tätä ajatusta myös huonolaatuisen koodin suhteen. (Fowler 1999, 75.) Haisevan koodin tunnistamisella ja refaktoroinnilla on monia vaikutusalueita sekä koodi- että ohjelmistotuotannon tasolla

4.2.1 Vaikutus koodiin ja koodin ylläpidettävyyteen

Koodin refaktorointi tekee koodista ymmärrettävämpää. Ohjelmistokehityksessä sekä virhekorjauksissa että uusien toiminnallisuuksien lisäämisessä tulee ymmärtää myös jo olemassa olevan koodin toimintaa. Jotta koodin parissa työskentely ei aiheuttaisi ylimääräistä vaivaa, tulisi vaikeasti ymmärrettävissä oleva koodi refaktoroida selkeämpään muotoon (Gillis 2021). Selkeämpään muotoon refaktorointi voi olla esimerkiksi koodin jäsentelyä uudelleen siten, että koodista saadaan johdonmukaisempaa ja ulkoasultaan siistimpää. Kuten luvussa 1.3 kuvailtiin, haasteellisuus ja monimutkaisuus voivat olla subjektiivisia näkemyksiä, minkä vuoksi esimerkiksi yleiset ohjeet koodin rakenteesta voivat auttaa refaktoroidessa. Myös koodin pituus voi tehdä koodista vaikeasti ymmärrettävää, kuten esimerkiksi rivimäärällisesti pitkien metodien toimintaa saattaa olla vaikea hahmottaa tai ylläpitää. Mikäli metodin pituutta ei pysty supistamaan tulisi se nimetä mahdollisimman kuvaavasti, jotta metodin nimestä pystyy helposti päättelemään sen toiminnan. (Trucchia & Romei 2010.) Kaiken kaikkiaan koodi aina tarkoittaa jotain. Mitä enemmän on koodia, sitä enemmän siinä on luettavaa ja ymmärrettävää. Refaktoroinnilla voidaan mahdollisesti saada eliminoitua turhaa koodia pois ja näin ollen vaikuttaa koodin määrään ja pituuteen saaden koodista mahdollisesti ymmärrettävämpää ja luettavampaa.

Refaktoroimalla koodi luettavampaan muotoon kehittää koodin rakennetta. Mitä vaikeampaa on tunnistaa koodin rakennetta, sitä vaikeampaa sitä on säilyttää ja sitä nopeammin koodi ”rappeutuu” eli monimutkaistuu. Eräs esimerkki koodin rakenteen refaktoroinnista on duplikaattien eliminointi. Mitä enemmän koodissa on duplikaatteja,

sitä enemmän on koodia ja sitä vaikeampaa on muokata koodia oikealla tavalla. Eliminoimalla duplikaatit voidaan varmistua, että koodin osat esiintyvät koodissa vain kerran, mikä on hyvän koodirakenteen tunnusmerkki. (Fowler 1999, 55-56.) Koodissa olevien duplikaattien määrä voi olla ylläpidon kannalta haasteellista myös siinä suhteessa, että koodimuutoksia tehdessä kaikkia duplikaatteja ei tule huomattua. Tämä tarkoittaa käytännössä sitä, että koodiin saattaa jäädä vanhoja koodin osia, mikä saattaa aiheuttaa ohjelmiston virheellisen toiminnan. Esimerkiksi, jos koodissa on useassa kohdassa samankaltainen ehdollisuus, on järkevämpää tehdä ehdollisuuden tarkastamisesta erillinen metodi ja näin ollen, mikäli ehdollisuuteen tulee muutoksia, muutos tarvitsee tehdä ainoastaan metodiin.

Refaktoroinnin avulla koodista voi löytää virheitä helpommin. Koska refaktoroinnissa tulee käytyä koodin toimintaa tarkasti lävitse, myös virheiden huomaaminen helpottuu. (Fowler 1999, 57.) Ymmärtämällä sen mitä koodi tekee ja mitä se ei tee nostaa todennäköisyyttä löytää koodista virheitä ennen kuin ne päätyvät lopulliseen tuotteeseen (Trucchia ym. 2010).

Refaktoroinnin yksi tärkeistä tarkoituksista on nopeuttaa ohjelmistokehitystä. Kuten aiemmin mainittu, refaktorointi kehittää koodin rakennetta, koodin luettavuutta ja vähentää virheiden määrää koodissa. Yhdessä nämä osa-alueet kehittävät koodin laatua. (Fowler 1999, 57.) Ohjelmistokehittäjän näkökulmasta koodin hyvä rakenne ja luettavuus ovat olennaisia ominaisuuksia laadullisesti hyvälle koodille. Näiden ominaisuuksien avulla myös ohjelmistokehittäminen on helpompaa ja nopeampaa. Ilman koodin hyvää rakennetta ohjelmistokehitys on vaivatonta jonkin aikaa, mutta aikanaan huono rakenne alkaa hidastaa koodaamista ja aikaa kuluu enemmän koodin tulkintaan kuin ohjelmistokehittämiseen (Fowler 1999, 57). Jotta ohjelmistokehitys olisi mahdollisimman tehokasta, tulisi koodia refaktoroida aina, kun siihen on tarve ja mahdollisuus.

4.2.2 Vaikutus taloudellisesti

Koodin refaktorointi ei pääsääntöisesti vaikuta ohjelmistotuotteen toimintaan. Data Respons (Data Respons n.d.) esittää asiaan liittyen verkkosivuillaan seuraavanlaisen kysymyksen: miksi tulisi refaktoroida, jos koodin jättäminen sellaisekseen ei maksa

mitään, eikä koodi ole rikki? Refaktoroinnin tarkoituksena on katsoa koodia laajemmin eikä niinkään keskittyä yksittäisiin ongelmakohtiin. Refaktoroinnilla ennaltaehkäistään virheiden syntymistä, mikä ajan pitkään on huomattavasti taloudellisesti edullisempaa kuin laadullisesti huonon koodin virhekorjaukset.

Ohjelmistot ovat taipuvaisia keräämään puutteita niiden sisäiseen laatuun eli koodiin ja nämä puutteet tekevät koodin muokkaamisen ja kehittämisen vaikeammaksi kuin se ideaalisessa tilanteessa olisi. Ward Cunningham loi metaforan ”technical debt” eli tekninen velka. Teknisellä velalla tarkoitetaan ajatusta käsitellä ohjelmiston laadullisia puutteita samanlaisena velkana kuin taloudellinen velka. Ylimääräinen vaiva, mitä uusien toimintojen lisääminen koodiin vie, on korko, mikä teknisestä velasta maksetaan. Toisinaan teknisen velan ottaminen on parempi vaihtoehto kuin refaktorointi. Esimerkiksi mikäli ohjelmistoon halutaan välittömästi uusia ominaisuuksia, kannattaa tekninen velka käsitellä tulevaisuudessa. (Fowler 2019.)

Refaktorointi voi aiheuttaa myös negatiivista vaikutusta taloudellisesti. Aina ei voida olla varmoja, kuinka paljon aikaa manuaalinen refaktorointi vie ja sen vuoksi ei ole järkevää refaktoroida esimerkiksi silloin, kun ohjelmiston toimituksen määräaika on lähellä (Fowler 1999, 66). Mikäli toimituksen määräaika on lähellä, voi olla parempi vaihtoehto ottaa teknistä velkaa refaktoroinnin sijaan, sillä refaktorointi saattaa rikkoa koodin toimintaa ja niiden korjaamisen vuoksi toimitus saattaa myöhästyä. Toimituksen myöhästymisen vuoksi refaktorointi voi tulla yritykselle kalliiksi myöhästymisestä saatujen sanktioiden vuoksi.

4.2.3 Vaikutus tietoturvaan

Refaktoroinnilla voidaan parantaa koodin tietoturvallisuutta kahdella tavalla: perinteisesti tai keskittymällä koodin turvallisuuteen. Perinteinen refaktorointi keskittyy pääasiassa koodin ylläpidettävyyden parantamiseen, kun taas turvallisen refaktoroinnin (”secure refactoring”) tarkoitus on parantaa koodin turvallisuutta. Perinteisen refaktoroinnin tuloksena voidaan ylläpidettävämmän koodin ansiosta saavuttaa lisäksi myös turvallisempi koodi. (Maruyama 2007.)

Kuten aiemmin mainittu, yksi perinteisen refaktoroinnin tarkoituksista on tehdä koodista ymmärrettävämpää ja yksinkertaisempaa. Mitä paremmin ohjelmistokehittäjät ymmärtävät koodin toimintaa sitä paremmin he voivat erottaa koodissa piilevät tietoturvariskit (Campbell 2020). Yksinkertaisemman koodin avulla voidaan paremmin hahmottaa ne koodin osat, jotka tarvitsevat toimenpiteitä, jotta koodin tietoturvallisuus voidaan varmistaa. Myös koodin huonolla rakenteella voi olla vaikutuksia koodin tietoturvallisuuteen. Esimerkiksi mikäli luvussa 4.2.1 mainitussa duplikaattikoodissa on tietoturvallisuuteen vaikuttavia virheitä eli ”bugeja”, tuplaantuu myös tietoturvariskien määrä. (Vieira 2017.)

Turvallinen refaktorointi keskittyy niihin koodin osiin, jotka tekevät koodista turvallisen. Turvallista refaktorointia voi olla esimerkiksi koodin osien, kuten kirjastojen, päivitykset. Kirjastot ovat kokoelmia valmiiksi kirjoitettua koodia, jotka auttavat ohjelmistokehittäjää säästämään aikaa ja vaivaa, sillä kirjastosta tuotuja ominaisuuksia ei tarvitse koodata uudestaan (Barone 2020; Papadopoulo n.d.). Kirjastojen tarjoajat ylläpitävät ja päivittävät kirjastojen sisältöjä, minkä vuoksi kirjastojen käyttäjien tulisi huomioida kirjastojen päivittäminen tarvittaessa etenkin koodin turvallisuuden kannalta. Kirjastojen päivityksen voivat sisältää muun muassa uusia ominaisuuksia, parannuksia koodin suorituksessa, virhekorjauksia ja tietoturvallisuuden parannuksia. Vaikka kirjastoja päivitetään ja parannellaan, ne voivat sisältää tietoturvariskejä. (Cruz 2019.) Enenevässä määrin avoimen lähdekoodin kirjastot ovat alttiita tietoturvariskeille, sillä hakkerit ottavat avoimen lähdekoodin tietoturvaheikkoudet kohteekseen, koska kirjastot ilmaiseksi saatavilla ja käytettävissä ilman erillistä lupaa (OmniSci n.d., Papadopoulo n.d.).

4.3 Refaktoroinnin automatisointi

Refaktorointityökalujen tarkoitus on tukea ohjelmistokehittäjän refaktorointeja ja tehdä refaktoroinnista nopeampaa vähentäen samalla mahdollisuutta luoda manuaalisella refaktoroinnilla virheitä koodiin (Brant & Roberts 1999, 401; Eilertsen & Murphy 2021b). Refaktoroinnin avuksi erilaiset ohjelmistoympäristöt eli IDEt ovat kehittäneet automatisoituja refaktorointityökaluja. Automaattisilla refaktorointityökaluilla voi

IDEn mukaan muun muassa uudelleen nimetä luokkia tai muuttujia (”rename class/variable”), poimia metodeja (”extract method”) tai siirtää koodia kansiosta toiseen (”move”). Vaikka nykyisin erilaiset IDEt tarjoavat paljon automaattisia refaktorointityökaluja, ohjelmistokehittäjät suhtautuvat työkaluihin varauksella.

Golubev, Kurbatova, Abdullah, Bryskin ja Mkaouer (2017, 2, 6, 10) tekivät tutkimuksen, jossa he tutkivat 1 183:n IntelliJ-alustan erilaisten IDEjen käyttäjien tapoja hyödyntää IDEn automaattisia refaktorointityökaluja ja heidän suhtautumistaan automaattisiin työkaluihin. Golubevin ym. suorittaman tutkimuksen tulosten perusteella ohjelmistokehittäjät luottavat hieman varoen automaattisiin työkaluihin ja niiden mahdollisiin sivuvaikutuksiin. Myös ohjelmistokehittäjien tietoisuus refaktorointityökaluista tai niiden toiminnasta vaikuttaa siihen, kuinka paljon he käyttävät mahdollisia IDEssä olevia refaktorointityökaluja. Annetuista vastausvaihtoehdoista yleisimmät syyt, miksi ohjelmistokehittäjät eivät käyttäneet erilaisissa tilanteissa automaattisia refaktorointityökaluja, olivat tutkimuksen perusteella seuraavat: manuaalisesti tehty refaktorointi oli helpompi tehdä ja tietämättömyys automaattisen refaktorointityökalun toimimisesta halutulla tavalla. Golubevin ym. (2017, 6) tutkimuksen tulosten perusteella ohjelmistokehittäjät ovat vastahakoisia käyttämään IDEjen automaattisia refaktorointityökaluja ja kokevat manuaalisen refaktoroinnin olevan intuitiivisempi valinta.

Erään toisen tutkimuksen automaattisiin refaktorointityökaluihin liittyen tekivät Pinto ja Kamei (2013, 1, 3). Tutkimuksessaan Pinto ja Kamei tutkivat refaktorointityökaluihin liittyviä kysymyksiä Stack Overflow -foorumissa, joka on ohjelmistokehitysfoorumeista maailman suosituin. Pinton ja Kamein tutkimustulokset osoittivat, että Stack Overflowssa esiintyneiden kysymysten ja vastausten perusteella syyt, miksi ohjelmistokehittäjät eivät omaksu automaattisia refaktorointityökaluja käyttöön, ovat tietämättömyys saatavilla olevista refaktorointityökaluista, niitä on vaikea opetella ja käyttää ja luottamuksen puute refaktorointityökaluja kohtaan.

Edellä mainittujen tutkimusten perusteella, jotta ohjelmistokehittäjät omaksuisivat refaktorointityökalujen käytön osaksi omaa tapaansa refaktoroida, refaktorointityökalujen tulisi olla helposti lähestyttäviä, työkalujen ohjeistukset tulisi olla helposti ymmärrettäviä ja työkaluja tulisi olla helppo käyttää. Ohjelmistokehittäjien luottamuksen saamiseksi refaktorointityökalujen suhteen on tehty parannuksia. Esimerkiksi IntelliJ IDE

tarjoaa mahdollisuuden esikatsella tehtävää muutosta ennen automaattisen refaktorointityökalun komennon toteuttamista (IntelliJ IDEA n.d.). Refaktoroinnin käsitteen, aiheen lähestymisen, työkalujen ja tekniikoiden parissa on tehty paljon töitä viimeisen kahdenkymmenen vuoden aikana (Naveed, Zeeshan & Ghulam 2021, 177). Kuitenkin vielä viimeisten vuosien aikana tehtyjen tutkimusten mukaan syitä, miksi ohjelmistokehittäjät eivät käyttäneet refaktorointityökaluja avuksi refaktoroidessaan, oli luottamuksen ja saatavilla olevien työkalujen tietoisuuden puute sekä niiden vaikeakäyttöisyys (Eilertsen & Murphy 2021a; Jain & Saha 2019, 6).

4.4 Refaktoroinnin onnistumisen arviointi

Refaktoroinnin onnistumista voidaan ensisijaisesti arvioida testaamalla. Testien tarkoitus on varmistaa, että koodin toiminnallisuus on sama kuin ennen refaktorointia (Mengerink 2013). Koska refaktoroinnin jälkeen koodin tulisi toimia kuten ennen, myös testejä tulisi läpäistä vähintään yhtä paljon kuin ennen. Uudet epäonnistuneet testit ovat merkki siitä, että refaktorointi on rikkonut koodin toiminnallisuutta. Ennen refaktoroinnin aloittamista on tärkeää kiinnittää huomiota, että koodilla on kunnolliset testit, jotta toiminnallisuuden säilyminen voidaan varmistaa refaktoroinnin jälkeen (Stone 2018).

Kuten luvussa 4.2.1 mainittiin, koodia olisi hyvä refaktoroida aina, kun siihen on tarve ja mahdollisuus. Jos refaktoroinnin kohde on iso tai refaktorointi suoritetaan omana tehtävänään, kannattaa tehtävälle refaktoroinnille tehdä suunnitelma. Refaktoroinnin suunnittelu tukee ohjelmistokehittäjää hahmottamaan halutun tavoitteen ja määrittelemään rajat refaktoroinnille estäen refaktoroinnin paisumisen (Gorbachenko 2021). Refaktoroinnin suunnittelu voi auttaa myös määrittelemään refaktorointiin tarvittavaa aikaa (Stone 2018). Suunnitelman avulla voidaan refaktoroinnin jälkeen yhdessä testien kanssa arvioida saavutettiinko refaktoroinnille asetetut tavoitteet.

5 ONGELMAN KUVAUS JA LÄHTÖKOHDAT

Opinnäytetyön aihe tuli työn toimeksiantajalta Profit Softwarelta. Opinnäytetyön tarkoituksena oli tutkia koodin refaktorointia, ylläpidettävyyttä ja niiden välistä yhteyttä. Toiminnallisena osuutena opinnäytetyöhön käytetään esimerkkejä Profit Softwarella tehdyistä refaktoroinneista kesän 2021 aikana ja tutkitaan refaktorointien vaikutusta koodin ylläpidettävyyteen. Tehdyt refaktoroinnit Profit Softwarella keskittyivät vain frontend-koodiin ja refaktoroinnit suoritettiin irrallisena jokapäiväisestä ohjelmistokehityksestä.

Profit Software käytti ennen käyttöliittymäkehityksessä Java-pohjaista Strutsia. Profit Software päätti luopua Struts-käyttöliittymästä, sillä JavaScript-pohjaiset sovelluskehikot ja kirjastot tarjosivat nopeampaa kehityssykliä. JavaScript-ohjelmointikielistä käyttöön valikoitui React, joka miellytti muun muassa suoraviivaisella ohjelmointimallillaan ja loivalla oppimiskynnyksellään. React-käyttöliittymässä otettiin käyttöön TypeScript ja MobX. TypeScript on JavaScript-ohjelmointikielen päälle rakennettu ohjelmointikieli, joka on vahvasti tyypitetty. MobX on kirjasto, joka toimii tilanhallintatyökaluna.

Yksi merkittävistä lähtökohdista frontend-koodin refaktoroinnin tarpeelle oli päivittää käytössä oleva React-käyttöliittymä nykyaikaan. React ja sen toimintaan vaadittavat paketit haluttiin päivittää ja kustomoida uudelleen. Osana refaktorointeja oli tarkoitus päivittää myös kaikki Reactiin liittyvät kirjastot, millä saataisiin kehitysympäristöihin enemmän nopeutta ja ohjelmistokehittäjille uusia työkaluja. Yksi päivitetyistä kirjastoista oli muun muassa tilanhallintakirjastoksi valittu MobX-kirjasto, jonka uusimmassa versiossa vanhoja käyttötapoja oli korvattu uusilla. Kirjastojen päivitysten myötä koodissa tuli korjata tarvittavia muutoksia. React-pakettien ja -kirjastojen päivitysten lisäksi React haluttiin tuoda nykyaikaan korvaamalla luokkakomponentit funktionaalisilla komponenteilla. Päivitysten jälkeen koodi haluttiin vielä käydä läpi, jotta koodia saataisiin paranneltua ja tuotua siihen enemmän tehokkuutta. (Sarin henkilökohtainen tiedonanto 17.2.2022.)

Koodin laadun parantamisen ja nykyaikaistamisen lisäksi refaktorointien motiivina oli ohjelmistokehityksen mielekkyys. Nykyaikaisessa ohjelmointiympäristössä voidaan

seurata mahdollisimman hyvin hyviä ohjelmointitapoja ja kehittäminen on mukavampaa, kun ohjelmointityökalut on päivitetty. Päivitetyn koodin avulla mahdollisille uusille työntekijöille, joilla on tietotaito ajan tasalla, ympäristö olisi tutumpi. (Sarin henkilökohtainen tiedonanto 17.2.2022.)

6 ONGELMAN ERITTELY JA RATKAISUT

Tehdyt refaktoroinnit tehtiin manuaalisesti ja ne koskivat laajasti koko frontend-koodia. Varmistus refaktorointien onnistumisesta saatiin automaattisten testien avulla, sillä testit kävivät koko sovelluksen läpi ja sen tuloksia pystyttiin vertaamaan ennen refaktorointeja tehtyihin testeihin. Automaattisten testien lisäksi suoritin käyttöliittymällä aktiivisesti manuaalista testausta refaktorointien aikana. Tässä kappaleessa kuvaan Profit Softwarella tekemieni refaktorointien kohteita omin esimerkein ennen ja jälkeen refaktoroinnin. Luvuissa avataan syitä refaktoroinnin tarpeelle yhdessä esimerkkien kanssa. Lukujen esimerkit osoittavat, miksi refaktorointeja haluttiin tehdä kyseisiin koodin osiin.

6.1 Any-tyypityksen eliminointi

TypeScriptissä any-tyypitystä voidaan käyttää, kun ei haluta TypeScriptin tarkistavan muuttujan tyypityksen oikeellisuutta (TypeScript n.d.). Any-tyyppinen muuttuja voi olla mitä vain tyyppiä, kuten esimerkiksi teksityyppi (”string”), numerotyyppi (”number”), totuustyyppi (”boolean”) tai objekti (”object”). Objektien tyypin muodostukseen voidaan käyttää hyödyksi rajapintoja. Rajapinnan avulla voidaan objektin sisältämät muuttujat nimetä ja tyypittää kuvan 2 mukaisesti, jossa on määritelty IUser-rajapinta.

```
1  export interface IUser {
2      firstName: string;
3      lastName: string;
4      age: number;
5      status: boolean;
6  }
7
```

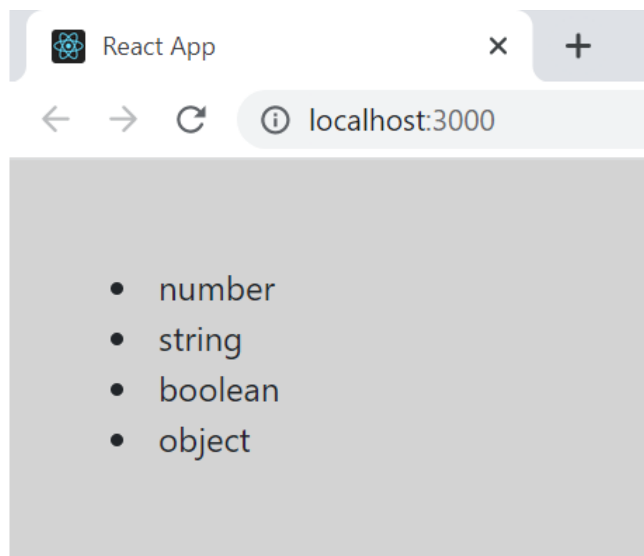
Kuva 2. IUser-rajapinnan määrittely.

Kuvan 3 esimerkkikoodissa kuvataan tilannetta, jossa funktiolle annetaan parametriksi neljässä eri tilanteessa erityyppiset arvot. Arvot ovat tyyppejä number, string, boolean

ja object. Funktio palauttaa parametrina syötetyn arvon tyypin ja tulostaa sen käyttöliittymälle. Tulosteet nähdään kuvassa 4. Funktio hyväksyy kaikki syötetyt parametrityypit any-tyypityksen vuoksi.

```
1  import React from 'react';
2  import { IUser } from './types/types';
3
4  const ExampleComponent: React.FC = () => {
5    const checkValueType = (value: any) => {
6      return typeof value;
7    };
8
9    const user: IUser = {
10     firstName: 'Maija',
11     lastName: 'Meikäläinen',
12     age: 25,
13     status: true
14   };
15
16   return (
17     <div className="pt-5 ps-5 pb-5" style={{ backgroundColor: 'lightgrey' }}>
18       <li>{checkValueType(2)}</li>
19       <li>{checkValueType('String-input')}</li>
20       <li>{checkValueType(true)}</li>
21       <li>{checkValueType(user)}</li>
22     </div>
23   );
24 };
25 export default ExampleComponent;
26
```

Kuva 3. Esimerkkikoodi, jossa funktio palauttaa sille annetun parametrin tyypin.



Kuva 4. Kuvan 3 koodin tulosteet käyttöliittymällä.

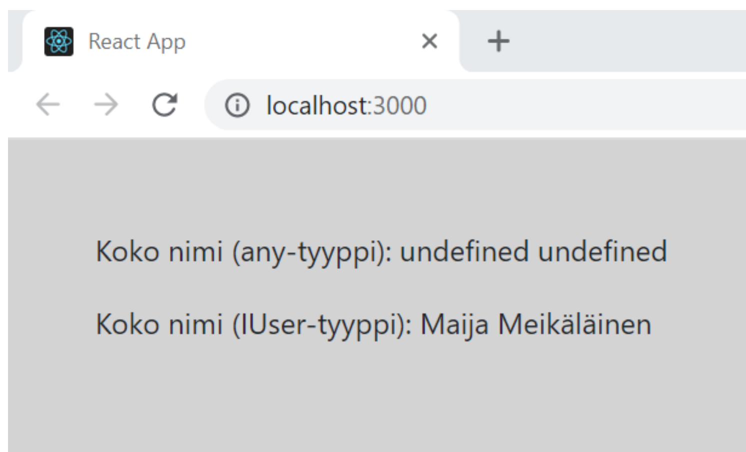
Yleensä funktion saamia parametreja halutaan käsitellä jollain tavalla, minkä vuoksi on tärkeää tietää tai varmistaa syötettyjen parametrien tyypit. TypeScriptin vahvalla

tyypitysominaisuudella voidaan automaattisesti varmistua siitä, että funktioille annetaan oikean tyyppiset parametrit. Parametrien alustus any-tyyppiseksi voi johtaa tilanteeseen, jossa parametrien käyttö ei toimi halutulla tavalla tai ollenkaan.

Kuvassa 4 on kuvailtu tilanne, jossa funktio luo etu- ja sukunimestä nimiyhdistelmän syötetyn parametrin mukaan. Funktiossa on syötettävä parametri asetettu any-tyyppiseksi, mutta sitä käsitellään IUser-tyyppisenä. Koska funktio ottaa vastaan kaikentyyppiset arvot, se ei nosta virhettä virheellisestä parametrityypistä. Näin ollen funktio ei toimi halutulla tavalla ensimmäisessä tapauksessa, jossa sille syötetään string-tyyppinen arvo. Virheellinen parametri tulostuu käyttöliittymälle ”undefined undefined”. Sen sijaan toisessa funktiokutsussa nimiyhdistelmän tulostuu oikein, sillä funktiolle syötetyltä parametrilta löytyy haetut muuttujat. Tulokset nähdään kuvassa 5.

```
1   import React from 'react';
2   import { IUser } from './types/types';
3
4   const ExampleComponent: React.FC = () => {
5     const user: IUser = {
6       firstName: 'Maija',
7       lastName: 'Meikäläinen',
8       age: 25,
9       status: true
10    };
11
12    const makeFullNameAny = (value: any) => {
13      return value.firstName + ' ' + value.lastName;
14    };
15
16    return (
17      <div className="pt-5 ps-5 pb-5" style={{ backgroundColor: 'lightgrey' }}>
18        <p>Koko nimi (string-tyyppi): {makeFullNameAny('String-value')}</p>
19        <p>Koko nimi (IUser-tyyppi): {makeFullNameAny(user)}</p>
20      </div>
21    );
22  };
23
24  export default ExampleComponent;
25
```

Kuva 4. Koodissa alustetaan IUser-objekti ja testataan makeFullName()-funktion toimintaa.



Kuva 5. Kuvan 4 funktion tulosteet käyttöliittymällä.

Kun tyypitykset ovat oikein, pystytään koodissa havaitsemaan virheet jo ohjelmistokehitysvaiheessa ja korjaamaan ne. Kuvassa 6 on esimerkki virheestä, joka syntyy, kun funktiolle yritetään antaa väärän tyyppistä arvoa. Virhe kertoo, ettei string-tyyppistä muuttujaa voi sijoittaa IUser-tyyppiseen muuttujaan.

```
1  import React from 'react';
2  import { IUser } from './types/types';
3
4  const ExampleComponent: React.FC = () => {
5    const user: IUser = {
6      firstName: 'Maija',
7      lastName: 'Meikäläinen',
8      age: 25,
9      status: true
10   };
11
12   const makeFullNameAny = (value: IUser) => {
13     return value.firstName + ' ' + value.lastName;
14   };
15
16   return (
17     <div className="pt-5 ps-5 pb-5" style={{ backgrou   Argument of type 'string' is not assignable to parameter of type 'IUser'. ts(2345)
                                                          View Problem   No quick fixes available
18       <p>Koko nimi (string-tyyppi): {makeFullNameAny('String-value')}</p>
19       <p>Koko nimi (IUser-tyyppi): {makeFullNameAny(user)}</p>
20     </div>
21   );
22 };
23
24 export default ExampleComponent;
25
```

Kuva 6. Funktiolle makeFullNameAny() yritetään syöttää väärän tyyppistä arvoa.

Toisinaan koodissa saattaa olla tapauksia, jossa käytettävän arvon tyyppiä ei tiedetä. Refaktoroinneissa haluttiin pystyä eliminoimaan kaikki ne tapaukset, joissa muuttujan tai parametrin tyyppi oli selvitettävissä. Eliminoimalla kaikki mahdolliset any-tyypitykset TypeScriptiä pystyttäisiin käyttämään sen tarjoaman edun mukaisesti. Any-tyypityksen eliminointi oli laajin ja työläin niistä refaktoroinneista, joita tein. Refaktoroinnissa koodista tuli etsiä kaikki ne kohdat, joissa any-tyypitystä oli käy-

tetty. Tyypitettyä arvoa tuli tutkia sen käyttötavan kautta, jotta se saataisiin tyypitettyä oikein. Esimerkiksi funktioiden any-tyyppisten parametrien oikean tyypin löysi helpoiten tutkimalla, missä funktiota kutsuttiin ja mistä se sai parametrinsa.

6.2 React Inlt-kirjaston päivitys

React Intl on kirjasto, jonka avulla käyttöliittymä voidaan lokalisoida halutuille kielille. Lokalisointia varten projektiin tarvitaan erillinen JSON-tiedosto, joka sisältää kielten käännökset. Käännöksiä voidaan käyttää koodissa niille määriteltyä id:tä vasten. Käyttöliittymä voidaan lokalisoida muun muassa selaimen kielen mukaan tai valitsemalla haluttu kieli alasvetovalikosta. Esimerkiksi mikäli kieleksi on valittu suomi, haetaan käyttöliittymälle suomenkieliset käännökset id:n mukaisesti JSON-tiedostosta. Jotta lokalisointi toimii, tulee koodi ”wrapata” React Intl:n IntlProvider-komponentin sisään. IntlProvider-komponentin avulla lokalisoinnit ovat saatavilla muille React Intl-komponenteille (Format.JS n.d.).

Kuvassa 7 on esitetty koodiesimerkki React Intl-kirjaston käytöstä. Esimerkissä IntlProvider-komponentin ominaisuuksiin kuuluu kieli (”locale”), tekstikäännökset (”messages”) ja tekstikomponenttien oletustyyppi (”textComponent”). IntlProviderin ja näin ollen käyttöliittymän kieli määräytyy esimerkissä muuttujan ”language” tilan mukaan. IntlProvider saa tekstikäännökset käyttöönsä funktion ”getMessages()” avulla. Käännös id:llä ”exampleSentence” tulostetaan FormattedMessage-komponentilla rivillä 26.

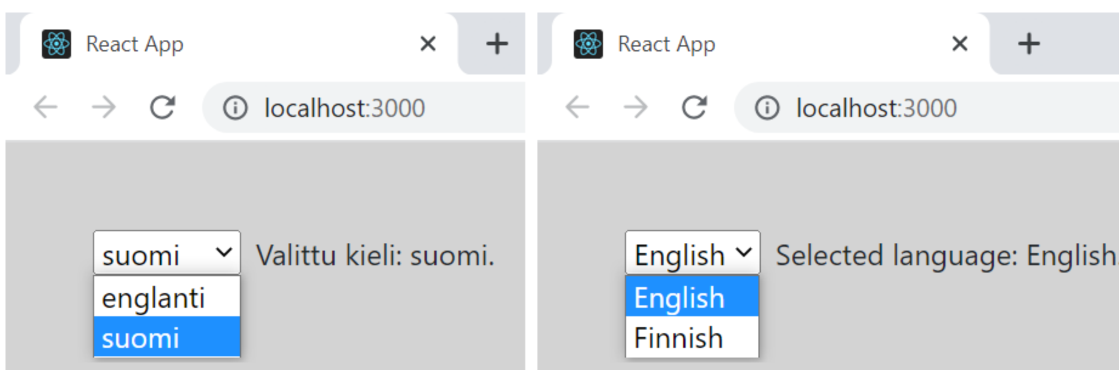
```tsx
import React, { useState } from 'react';
import { FormattedMessage, IntlProvider } from 'react-intl';
import DropDown from './DropDown';
import localizations from './localization/messages.json';

const ExampleComponent: React.FC = () => {
  const [language, setLanguage] = useState('en');

  const getMessages = (value: string) => {
    switch (value) {
      case 'fi':
        return localizations.fi;
      case 'en':
        return localizations.en;
    }
  };

  return (
    <IntlProvider
      locale={language}
      messages={getMessages(language)}
      textComponent="span"
    >
      <div className="pt-5 ps-5 pb-5" style={{ backgroundColor: 'lightgrey' }}>
        <DropDown setLanguage={setLanguage} />
        <FormattedMessage id={'exampleSentence'} />
      </div>
    </IntlProvider>
  );
};

export default ExampleComponent;

```

Kuva 7. Esimerkki React Intl-kirjaston käytöstä koodissa.

Joskus koodissa voi olla tilanteita, jossa ei voida suoraan käyttää React Intl-kirjaston tarjoamia komponentteja. Tällaisissa tilanteissa voidaan käyttää React Intl-kirjaston useIntl()-hookkia, joka luo IntlShape-tyyppisen muuttujan. IntlShape-tyyppisellä muuttujalla voidaan kutsua React Intl-kirjaston funktioita, jotka toimivat kuten kirjaston komponentit. Kuvassa 8 on esitetty esimerkkinä DropDown-komponentti, jossa on käytetty React Intl:n FormattedMessage-komponentin sijasta IntlShape-tyyppisen muuttujan funktiota ”formatMessage()” lokalisoitaessa alasvetovalikon vaihtoehtoja. Koska IntlProvider-komponentissa määriteltiin, että tekstikomponentit ovat span-elementtejä, ei FormattedMessage-komponenttia voida sijoittaa option-elementin sisälle. Tämä on sen vuoksi, että HTML ei anna sijoittaa option-elementin sisälle muita elementtejä. Alasvetovalikon lokalisoinnit käyttöliittymällä on esitetty kuvassa 9.

```
import { Dispatch, SetStateAction } from 'react';
import { useIntl } from 'react-intl';

const DropDown: React.FC<{ setLanguage: Dispatch<SetStateAction<string>> }> = ({
  setLanguage
}) => {
  const intl = useIntl();

  return (
    <select className="me-2" onChange={(e) => setLanguage(e.target.value)}>
      <option value="en">{intl.formatMessage({ id: 'english' })}</option>
      <option value="fi">{intl.formatMessage({ id: 'finnish' })}</option>
    </select>
  );
};

export default DropDown;

```

Kuva 8. Alasvetovalikon vaihtoehdot on lokalisoitu käyttämällä IntlShape-tyyppisen muuttujan funktiota ”formatMessage()”.



Kuva 9. Alasvetovalikon ja tekstin lokalisoinnit käyttöliittymällä.

Tarve React Intl-kirjaston päivitykselle huomattiin eliminoitaessa any-tyypitystä koodista. Koodissa käytössä ollut React Intl-kirjaston versio ei sisältänyt useIntl()-hookkia, minkä vuoksi tarvittavat IntlShape-tyyppiset muuttujat oli tuotu halutuille komponenteille injectIntl()-wrapper-komponentin avulla. Koodin komponenteille oli erikseen määritelty rajapinnat ja koska IntlShape-tyypitystä ei voitu tuoda kirjastosta koodiin, oli IntlShape-muuttujat tyypitetty any-tyypisiksi. React Intl-kirjaston päivittäminen ja useIntl()-hookin tuominen koodiin poisti ylimääräisiä rajapintoja sekä parametrisyöttöjä komponenteille.

Any-tyypityksen eliminoinnin lisäksi koodissa tuli tehdä kirjastopäivityksen myötä myös muita muutoksia. Yksinkertaisin muutos oli addLocaleData()-funktiokutsun

poistaminen koodista, sillä se ilmeni vain kerran koodissa ja se loi vain yhden virheviestin. React Intl-kirjasto ei enää uudemmassa versiossa tarjonnut addLocaleData()-funktiota, joka lokalisoi muun muassa päivän, kellonajan ja numerot (Grill n.d.). Funktiokutsu tuli näin ollen poistaa.

Toinen koodin rikkovista muutoksista oli IntlProviderin tekstikomponentin oletuselementin muuttuminen. Koko sovelluksen ”wrappaava” IntlProvider-komponentti käytti versiota 3 aiemmissa versioissa Formatted-komponenteissaan oletetusti span-elementtiä. Versiosta 3 eteenpäin oletuselementti muuttui ”React.Fragmentiksi”, minkä vuoksi IntlProvideriin tuli manuaalisesti määritellä tekstikomponentiksi span-elementti. Määrittelyn avulla vanha käyttöliittymä voitiin pitää toimivana ja näin ollen päivityksestä aiheutunut koodin rikkoutuminen oli helppo korjata.

Toinen koodin rikkova muutos oli FormattedDate-komponentin tyylittelyominaisuus. React Intl-kirjaston käytössä olleessa versiossa FormattedDate-komponentille voitiin määritellä tyylittelyt komponentin sisällä, mikä ei uudemmassa versiossa ollut enää mahdollista. React Intl-kirjaston kehittäjät ovat kuvailleet tehtyjä muutoksia GitHubissa, jossa he avaavat muun muassa tyylittelyn osalta tehtyjä muutoksia. Kehittäjät kertovat, kuinka he mieltävät Formatted-komponentit (esimerkiksi FormattedMessage, -Date ja -Number) tekstinä ja suosittelivat React Intl-kirjaston uudemmissa versioissa ”wrappaamaan” Formatted-komponentit ja tyylittelemään nämä ”wrapperit” (Ferraiuolo 2015). Koodin korjaaminen vaati vaivannäköä, sillä rikkoutuneet kohdat tuli hakea koodista ja ne tuli muokata uuden kirjaston kanssa yhteensopiviksi.

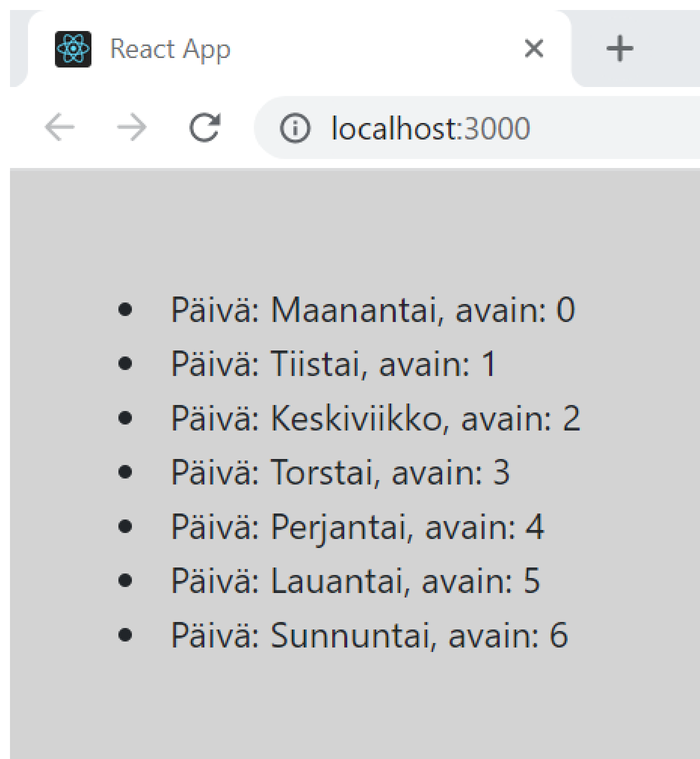
6.3 Indeksi-avaimen eliminointi

Tulostettaessa React-käyttöliittymälle listan jäseniä kuuluu hyviin ohjelmointitapoihin antaa kullekin listan jäsenelle oma avain. Avaimen tarkoitus on tunnistaa listan jäsenet toisistaan. Indeksi-avain tarkoittaa, että avaimeksi määritellään listan jäsenen paikka listassa eli indeksi. Esimerkiksi listan jäsen indeksillä 0 on listan ensimmäinen jäsen.

Kuvassa 10 on esitetty esimerkki indeksi-avaimen käytöstä koodissa. Listaksi on määritelty kaikki viikonpäivät. Koodi tulostaa selaimeen aina riveittäin listan jäsenen ja sen avaimen eli tässä tapauksessa indeksin. Tulosteet on esitetty kuvassa 11.

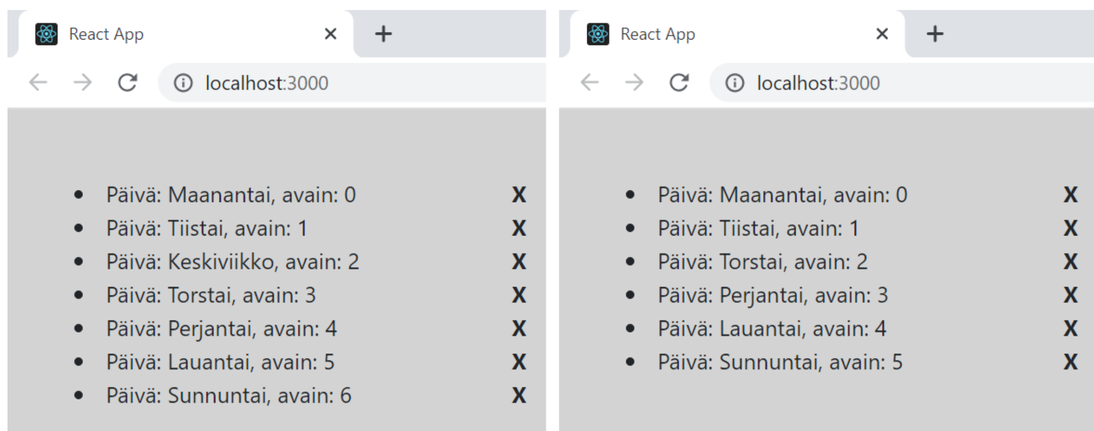
```
1   import React from 'react';
2
3   const ExampleComponent: React.FC = () => {
4     const weekdays = [
5       'Maanantai',
6       'Tiistai',
7       'Keskiviikko',
8       'Torstai',
9       'Perjantai',
10      'Lauantai',
11      'Sunnuntai'
12    ];
13
14    return (
15      <div className="pt-5 ps-5 pb-5" style={{ backgroundColor: 'lightgrey' }}>
16        {weekdays.map((day, index) => {
17          return (
18            <li key={index}>
19              Päivä: {day}, avain: {index}
20            </li>
21          );
22        })}
23      </div>
24    );
25  };
26
27  export default ExampleComponent;
28
```

Kuva 10. Esimerkkikoodi, jossa on listan jäsenelle annetaan avaimeksi sen indeksi.



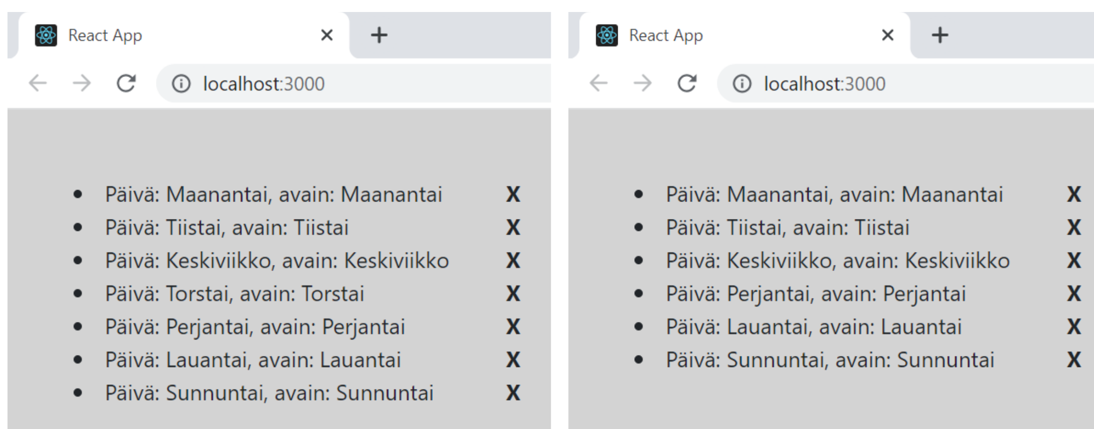
Kuva 11. Kuvan 10 koodin tulosteet käyttöliittymällä.

Indeksin käyttämistä avaimena ei suositella, sillä indeksi-avain voi vaikuttaa negatiivisesti koodin toimintaan (React n.d.b). Esimerkiksi poistettaessa listan keskeltä jäsen, React-ohjelmointikieli ei erota, onko listasta poistettu jäsen vai onko jäsenien sisältöä muutettu. Kuvassa 12 on esitetty tilanne, jossa kuvan 10 koodiin on lisätty mahdollisuus poistaa listan jäseniä käyttöliittymältä. Vasemmanpuoleinen kuva on lähtötilanne, jossa keskiviikkopäivällä on avaimena numero kaksi. Kun keskiviikko poistetaan listasta, nähdään oikeanpuoleisesta kuvasta päivien torstai - sunnuntai avaimien arvojen olevan yhtä numeroa pienemmät. Mikäli indeksiä käytetään avaimena, listajäsenen poistaminen tai lisääminen listan keskelle muuttaa myös avainarvoja. Haluttu tilanne on, että avaimet pysyvät samana, jotta jäsenet voitaisiin tunnistaa toisistaan myös muuttuvassa tilanteessa.

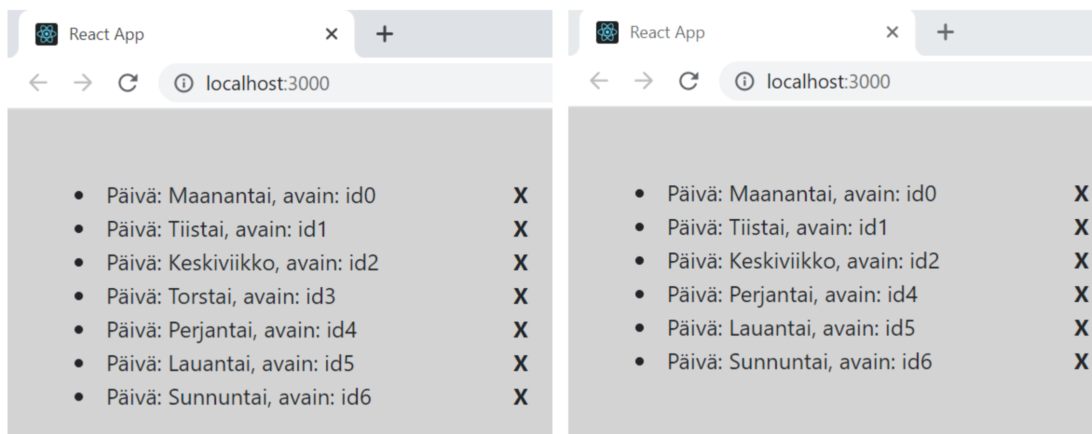


Kuva 12. Indeksi-avaimien arvo voi muuttua, kun listaa muutetaan keskeltä.

Hyvä tapa avaimen määrittelyyn on esimerkiksi määritellä listan jäsenille yksilölliset tunnisteet. Tunniste voisi olla esimerkiksi juokseva luku tai henkilötunnuksen kaltainen tunniste. Jos tiedetään, että listan kaikki jäsenet ovat uniikkeja, voidaan avaimena käyttää myös jäsenen arvoa. Esitetyn esimerkin tapauksessa tiedetään, että kukin viikonpäivä esiintyy listassa vain kerran ja näin ollen voidaan avaimena käyttää viikonpäivän arvoa. Listan muokkaus päivän ollessa avainarvona on esitetty kuvassa 13 ja listan muokkaus erillisen tunnisteen ollessa avaimena on esitelty kuvassa 14.



Kuva 13. Viikonpäivät esiintyvät kukin vain kerran listassa, minkä vuoksi avainarvona voidaan käyttää päivän arvoa.



Kuva 14. Listan jäsenille voidaan määritellä henkilökohtaiset tunnisteet.

Indeksin käyttö avaimena voi aiheuttaa virheen koodin toiminnassa. Jotta voitaisiin ennaltaehkäistä mahdollisia virheitä, indeksi-avaimen käyttö haluttiin refaktoroida pois koodista. Useimmissa tapauksissa listattavat jäsenet olivat objekteja tai niiden osia. Koodin objekteilta löytyi lähes aina määriteltynä uniikki tunniste. Refaktoroinnissa tuli hakea koodista kaikki ne osat, joissa oli indeksiä käytetty avaimena. Sopivan avaimen löytäminen vaati objektien rakenteen tutkimista ja mahdollisen uuden avaimen uniikkiuden varmistamista.

6.4 Moment.js-kirjaston korvaus

Moment.js on kirjasto, jonka avulla voidaan käsitellä päiviä ja kellonaikoja. Syyskuussa 2020 Moment-kirjaston kehittäjät ilmoittivat, ettei kirjastoa enää tulla kehittämään (Moment.js, 2020). Moment-kirjastolle ei vielä oltu päätetty korvaavaa kirjastoa, minkä vuoksi ennen refaktorointia tuli vertailla Momentin kaltaisia tarjolla olevia päivienkäsittelykirjastoja ja valita niistä koodiin sopivin.

Moment.js suosittelee kotisivuillaan erilaisia vaihtoehtoja, joilla voi korvata Moment-kirjaston. Suosituksissa mainitaan Luxon, Day.js, Date-fns, js-Joda ja kirjastoton vaihtoehto (Moment.js 2020). Aloin selvittämään korvaavaa kirjastoa suositeltujen kirjastojen listalta. Tutkin kirjastoja sekä suositeltujen kirjastojen kotisivujen kautta että ohjelmistokehittäjien suosimien keskustelupalstojen, kuten Stack Overflown, kautta. Keskustelupalstoilla kehittäjät olivat kertoneet mihin kirjastoon he olivat korvanneet

Moment-kirjaston ja miksi. Pidin uuden kirjaston valinnassa tärkeänä laajaa käyttömahdollisuutta sekä sitä, että koodissa ei tarvitsisi tehdä massiivisia muutoksia uuden kirjaston myötä.

Vertailun ja tutkinnan tuloksena Date-fns-kirjasto valikoitui Moment-kirjaston korvaajaksi. Date-fns-kirjasto tuli valituksi, sillä se oli yksi suositelluista ja suosituimmista kirjastoista sekä koodissamme oli jo joissain osissa käytetty kyseistä kirjastoa. Date-fns-kirjastolla oli myös hyvät ja selkeät dokumentaatiot esimerkkeineen nettisivuillaan, mikä helpotti kirjaston omaksumista.

Moment- ja Date-fns-kirjaston välillä on useita eroja, mikä osoittautui joiltain osin refaktoroinnin haasteellisuutena. Merkittävä ero kirjastojen välillä on, että Moment-kirjastoa käytetään Moment-objektien kautta. Date-fns-kirjastossa funktioita voidaan kutsua ilman objektia antamalla funktioille Date-tyyppinen muuttuja parametrina. Näin ollen kaikki Moment-tyypitykset tuli refaktoroinnissa muuttaa Date-tyyppisiksi. Tämän myötä koodista voitiin poistaa toDate()-funktiot, joita oli käytetty, kun Moment-objektia haluttiin käsitellä Date-tyyppisenä.

```
1   import React from 'react';
2   import moment from 'moment';
3   import { getYear } from 'date-fns';
4
5   const ExampleComponent: React.FC = () => {
6     const momentDate = moment();
7     const dateFnsDate = new Date();
8
9     return (
10      <div className="pt-5 ps-5 pb-5" style={{ backgroundColor: 'lightgrey' }}>
11        {momentDate.year()}     {/*  2022  */}
12        {getYear(dateFnsDate)}  {/*  2022  */}
13      </div>
14    );
15  };
16
17  export default ExampleComponent;
18
```

Kuva 15. Moment-kirjastoa käytetään Moment-tyyppisen objektin kautta, kun taas Date-fns-kirjastoa käytetään syöttämällä Date-tyyppinen parametri funktiolle.

Kirjastoa korvattaessa jouduttiin koodissa muokkaamaan kaikkia niitä osia, joissa oli käytetty Moment-tyyppisiä muuttujia. Date-fns-kirjaston kattavan dokumentaation an-

siosta tarvittavat koodimuutokset pystyttiin tekemään suhteellisen vaivattomasti. Lisäksi Moment ja Date-fns jakavat muutamia saman nimisiä tai melko saman nimisiä funktioita. Esimerkiksi tutkittaessa, onko päivä A päivän B jälkeen, käytetään molemmissa kirjastoissa funktiota isAfter(). Formatoinnissa molemmat kirjastot käyttävät funktiota format(). Formatoinnin tyypitykset ovat kirjastojen välillä kuitenkin erilaiset, kuten esimerkiksi samanlainen vuosi-kuukausi-päivä-merkintä on Momentissa ”YYYY-MM-DD” ja Date-fns:ssä ”yyyy-MM-dd”. Samankaltaisiin funktiokutsuihin kuuluu muun muassa päivän alkamisajankohdan haku, joka on Momentissa date.startOf(’day’) ja Date-fns:ssä startOfDay(date).

```
1   import React from 'react';
2   import moment from 'moment';
3   import { format, isAfter, startOfDay } from 'date-fns';
4
5   const ExampleComponent: React.FC = () => {
6     const momentStartDate = moment(new Date(2022, 0, 1));
7     const momentEndDate = moment(new Date(2023, 0, 1));
8     const dateFnsStartDate = new Date(2022, 0, 1);
9     const dateFnsEndDate = new Date(2023, 0, 1);
10
11    return (
12      <div className="pt-5 ps-5 pb-5" style={{ backgroundColor: 'lightgrey' }}>
13        {momentEndDate.isAfter(momentStartDate) ? 'true' : 'false'}     {/*  true  */}
14        {isAfter(dateFnsEndDate, dateFnsStartDate) ? 'true' : 'false'}  {/*  true  */}
15
16        {momentStartDate.format('YYYY-MM-DD')}                          {/* 2022-01-01 */}
17        {format(dateFnsStartDate, 'yyyy-MM-dd')}                        {/* 2022-01-01 */}
18
19        {momentEndDate.startOf('day').toString()}    {/* Sun Jan 01 2023 00:00:00 GMT+0200 */}
20        {startOfDay(dateFnsEndDate).toString()}      {/* Sun Jan 01 2023 00:00:00 GMT+0200 (Eastern European Standard Time) */}
21      </div>
22    );
23  };
24
25  export default ExampleComponent;
26
```

Kuva 16. Samankaltaisuudet Moment- ja Date-fns-kirjaston funktiokutsuissa.

Date-fns-kirjaston funktioiden käyttö osoittautui useissa tapauksissa olevan yksinkertaisempi kuin Moment-kirjaston funktioiden. Esimerkiksi laskettaessa kahden päivän välistä päiväeroa on Date-fns-kirjastolla funktiokutsu muotoa: differenceInDays(endDate, startDate). Momentissa puolestaan funktiokutsu on huomattavasti pidempi: moment.duration(endDate.diff(StartDate)).asDays().

```
1   import React from 'react';
2   import moment from 'moment';
3   import { differenceInDays } from 'date-fns';
4
5   const ExampleComponent: React.FC = () => {
6     const momentStartDate = moment(new Date(2022, 0, 1));
7     const momentEndDate = moment(new Date(2023, 0, 1));
8     const dateFnsStartDate = new Date(2022, 0, 1);
9     const dateFnsEndDate = new Date(2023, 0, 1);
10
11    return (
12      <div className="pt-5 ps-5 pb-5" style={{ backgroundColor: 'lightgrey' }}>
13        {moment.duration(momentEndDate.diff(momentStartDate)).asDays()}   {/* 365 */}
14        {differenceInDays(dateFnsEndDate, dateFnsStartDate)}             {/* 365 */}
15      </div>
16    );
17  };
18
19  export default ExampleComponent;
20
```

Kuva 17. Date-fns-kirjaston funktiokutsut ovat joissain tapauksissa huomattavasti yksinkertaisempia ja selkeämpiä kuin Moment-kirjaston funktiokutsut.

6.5 useFormik()-funktion eliminointi

Formik-kirjaston tarkoitus on tehdä React-käyttöliittymän lomakkeiden käsittelystä helpompaa. Kirjaston Formik-komponentin avulla on helpompi käsitellä lomakkeeseen syötettyjä arvoja, tehdä validointeja ja lisätä virheviestejä sekä käsitellä valmista lomaketta. Formik-kirjasto sisältää useFormik()-hookin, jonka avulla Formik voidaan ottaa suoraan käyttöön, sillä hookki luo sisäisesti Formik-komponentin. Hookkia ei kuitenkaan ole tarkoitettu yleisesti käytettäväksi. UseFormik()-hookkia suositellaan käytettäväksi esimerkiksi silloin, kun halutaan välttää React Contextin käyttöä. (Formik n.d.b.) React Contextin avulla datan syöttäminen komponenttipuussa eteenpäin voidaan tehdä ilman ylimääräisten parametrien syöttämistä komponenteille (React n.d.a).

Kuvissa 18 ja 19 on esitetty koodiesimerkki useFormik()-hookilla tehdystä lomakkeesta. Hookki palauttaa kuvassa 18 Formikin tilan ja metodit formik-muuttujaan rivillä 8. FormComponent-komponentti kuvassa 19 käyttää Formikin ominaisuuksia, minkä vuoksi formik-muuttuja joudutaan syöttämään FormComponent-komponentille

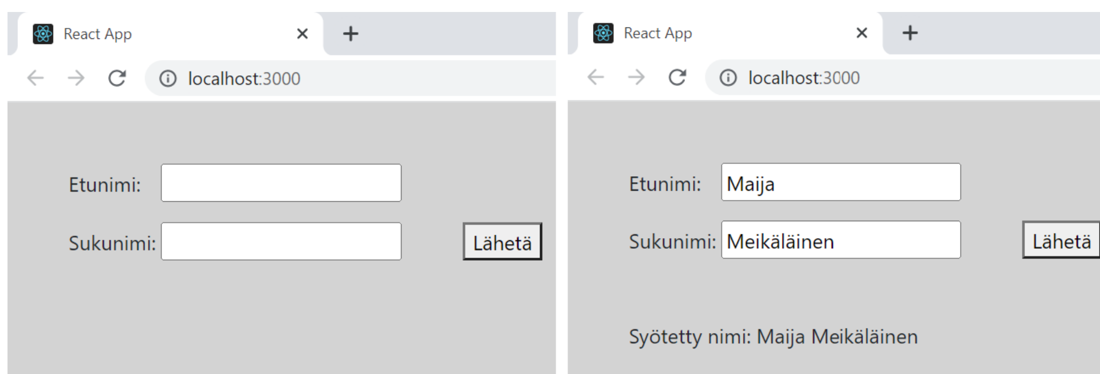
parametrina. Kuvassa 20 on esitetty lomakkeen ulkoasu käyttöliittymällä ennen ja jälkeen arvojen syötön. Kun Formikin handleSubmit-funktiota on kutsuttu Lähetä-nappia painamalla, käyttöliittymälle tulostuu syötettyjen arvojen nimiyhdistelmä.

```
1   import { useFormik } from 'formik';
2   import React, { useState } from 'react';
3   import FormComponent from './FormComponent';
4
5   const ExampleComponent: React.FC = () => {
6     const [name, setName] = useState('');
7
8     const formik = useFormik({
9       initialValues: {
10        firstName: '',
11        lastName: ''
12      },
13      onSubmit: (values) => {
14        setName('Syötetty nimi: ' + values.firstName + ' ' + values.lastName);
15      }
16    });
17
18    return (
19      <div className="pt-5 ps-5 pb-5" style={{ backgroundColor: 'lightgrey' }}>
20        <form onSubmit={formik.handleSubmit}>
21          <FormComponent formik={formik} />
22        </form>
23        {name}
24      </div>
25    );
26  };
27
28  export default ExampleComponent;
29
```

Kuva 18. Esimerkki Formik-lomakkeen luonnista useFormik()-hookilla. Formik-muuttuja on syötetty alikomponentille parametrina.

```
1  import React from 'react';
2
3  const FormComponent: React.FC<{ formik: any }> = ({ formik }) => {
4    return (
5      <>
6        <span className="me-3">Etunimi:</span>
7        <input
8          className="mb-3"
9          id="firstName"
10         name="firstName"
11         type="text"
12         onChange={formik.handleChange}
13         value={formik.values.firstName}
14       />
15       <br />
16       <span className="me-1">Sukunimi:</span>
17       <input
18         id="lastName"
19         name="lastName"
20         type="text"
21         onChange={formik.handleChange}
22         value={formik.values.lastName}
23       />
24       <button className="ms-5 mb-5" type="submit">
25         Lähetä
26       </button>
27     </>
28   );
29 };
30
31 export default FormComponent;
32
```

Kuva 19. Lomakkeelle syötetään Formik-muuttuja parametrina, jotta Formikin ominaisuuksia voitaisiin käyttää.



Kuva 20. UseFormik()-hookilla tehdyn lomakkeen ulkoasu käyttöliittymällä.

Kun Formik-lomake tehdään käyttäen Formik-komponenttia, voidaan sen sisällä olevilla alikomponenteilla käyttää Formik-komponentin ominaisuuksia FormikContextin avulla. FormikContextin ansiosta alikomponenteille ei tarvitse erikseen syöttää Formik-muuttujaa parametrina. Kuvien 21 ja 22 esimerkeissä on Formik-komponentin sisällä käytetty Field-komponentteja. Field-komponentti vastaa HTML:n input-

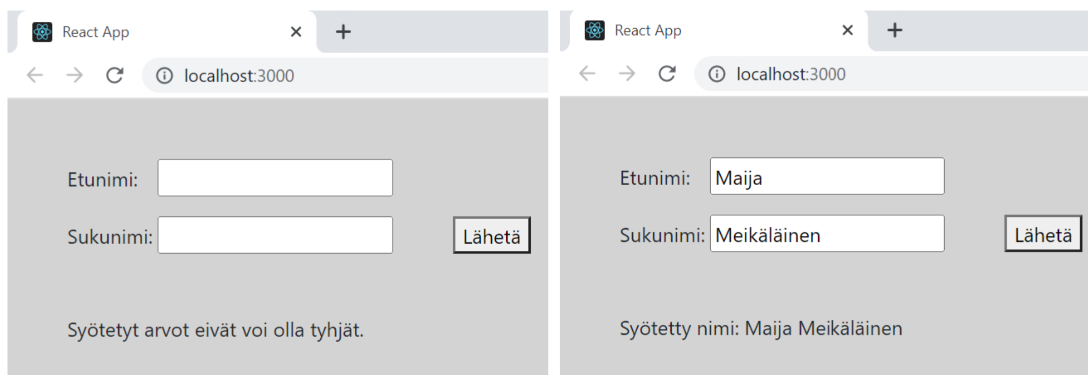
elementtiä ja se päivittää automaattisesti Formikin initialValues:a muuttuessaan. Erillistä handleChange-funktiota ei näin ollen tarvitse kutsua, kun käyttöliittymällä tekstikenttiin syötetään arvoja. FormikContextin käyttöesimerkki on esitetty kuvassa 22, jossa Formikin ominaisuuksista alikomponentille on tuotu arvot (”values”) ja handleSubmit-funktio. Arvoilla tarkoitetaan Formikin initialValues:a. Esimerkissä handleSubmit-funktiota kutsutaan vasta sitten, kun ensin on tarkastettu etteivät syötetyt arvot ole tyhjiä merkkijonoja. Jos syötetyt arvot ovat tyhjiä merkkijonoja, virheviesti näytetään käyttöliittymällä.

```
1  import { Form, Formik } from 'formik';
2  import React, { useState } from 'react';
3  import FormComponent from './FormComponent';
4
5  const ExampleComponent: React.FC = () => {
6    const [name, setName] = useState('');
7    return (
8      <div className="pt-5 ps-5 pb-5" style={{ backgroundColor: 'lightgrey' }}>
9        <Formik
10         initialValues={{
11           firstName: '',
12           lastName: ''
13         }}
14         onSubmit={(values) => {
15           setName('Syötetty nimi: ' + values.firstName + ' ' + values.lastName);
16         }}
17       >
18         <Form>
19           <FormComponent />
20         </Form>
21       </Formik>
22       {name}
23     </div>
24   );
25 };
26
27 export default ExampleComponent;
28
```

Kuva 21. Formik-lomake tehtynä Formik-komponentilla.

```tsx
import { Field, FormikValues, useFormikContext } from 'formik';
import React, { useState } from 'react';

const FormComponent: React.FC = () => {
  const [error, setError] = useState(false);
  const { handleSubmit, values } = useFormikContext<FormikValues>();

  return (
    <>
      <span className="me-3">Etunimi:</span>
      <Field className="mb-3" id="firstName" name="firstName" />
      <br />
      <span className="me-1">Sukunimi:</span>
      <Field id="lastName" name="lastName" />
      <button
        className="ms-5 mb-5"
        type="button"
        onClick={() =>
          values.firstName.length !== 0 && values.lastName.length !== 0
            ? (handleSubmit(), setError(false))
            : setError(true)
        }
      >
        Lähetä
      </button>
      {error && <p>Syötetyt arvot eivät voi olla tyhjät.</p>}
    </>
  );
};

export default FormComponent;

```

Kuva 22. Field-komponentti päivittää automaattisesti Formikin tilaa ilman erillistä handleChange-funktiokutsua.



Kuva 23. Käyttöliittymällä näytetään virheviesti, jos lomakkeella yritetään lähettää tyhjät arvot.

Ennen refaktorointia koodissa oli käytetty joissain osissa useFormik()-hookkia. Hookin käyttö haluttiin refaktoroida Formik-komponentteihin, sillä Formikin dataa käytettiin myös lomakkeen alikomponenteissa. Tämän vuoksi hookin luoma Formik-muuttuja oli jouduttu syöttämään koodissa parametrina alikomponenteille. Formikkia oli

käytetty näin ollen koodissa väärin sen käyttötarpeeseen nähden ja se haluttiin refaktoroinneissa korjata.

UseFormik()-hookin refaktorointiin kuului toisinaan isoja osia koodista, sillä refaktoroinnit koskivat koko useFormik():lla tehtyä lomaketta ja sen sisältämiä alikomponentteja. Koska refaktoroitavat lomakkeet saattoivat olla isoja, loi se haastetta muutosten teolle. Refaktoroitavassa koodissa lomakkeet ja näin ollen Formik ovat merkittävä tekijä frontend- ja backend-koodin välisessä keskustelussa, minkä vuoksi koodimuutoksia tehtäessä käyttöliittymällä tuli aktiivisesti tehdä manuaalista testausta ja varmistaa lomakkeiden oikeanlainen toiminnallisuus.

7 REFAKTOROINNIN LOPPUTULOS

Kappaleessa 6 esitellyt refaktoroinnit yhdessä muiden kesällä 2021 tehtyjen refaktorointien kanssa saavuttivat niille asetetut tavoitteet. Frontend-koodin refaktorointien avulla saatiin merkittäviä parannuksia niihin sovelluksiin Profit Softwarella, joihin pääasiallinen sisällönkehitys tuli kohdistumaan lähitulevaisuudessa.

Koodi saatiin luettavammaksi ja varmemmaksi eliminoimalla any-tyypitykset sekä indeksi-avaimet koodista. Mahdollisia virheitä saatiin ennaltaehkäistyä indeksi-avainten eliminoinnilla, sillä indeksi-avaimia käytettäessä listojen käsittely muuttuvassa tilanteessa ei välttämättä toimi oikein. Virheiden ennaltaehkäisyyn pystyttiin vaikuttamaan myös eliminoimalla any-tyypitykset koodista. Oikeilla tyypityksillä sekä käytössä olevan TypeScriptin avulla koodista kyetään havaitsemaan mahdolliset virheet jo ohjelmistokehitysvaiheessa virheviestien kautta. Kun tyypitykset ovat oikeat, koodista voidaan helposti ymmärtää, millaisia muuttujia se käsittelee. Koodin ymmärrettävyys ja virheiden minimointi voidaan mieltää koodin analysoitavuudeksi ja vakaudeksi, jotka ovat osa koodin ylläpidettävyyttä.

Kirjastojen ja koodin ylläpito on helpompaa, kun kirjastopäivitykset pitää ajan tasalla. Pitkittynyt kirjaston päivitysväli saattaa kerryttää tehtävien koodimuutosten määrää tulevaisuudessa. Päivittämällä React Intl-kirjasto ja korvaamalla Moment.js-kirjasto vastaavaan ylläpidettyyn Date-fns-kirjastoon koodia saatiin ajantasaisemmaksi. React Intl-kirjaston päivityksen myötä koodiin saatiin sekä uusia että tarvittuja ominaisuuksia käyttöön. Date-fns-kirjaston käyttöönotto osoittautui tekemään koodista luettavampaa yksinkertaisen käyttötapansa ansiosta. Date-fns-kirjaston kattava dokumentaatio tekee kirjaston käytön omaksumisesta helpompaa. Ajantasaiset kirjastot eivät ole yhtä alttiita tietoturvariskeille vanhempiin versioihin verrattuna. Lisäksi nykyaikaisten kirjastojen myötä ohjelmistokehityksestä saadaan mielekkäämpää, kun käytössä on modernit työkalut.

Refaktoroimalla lomakkeiden luonti useFormik()-hookista Formik-komponenttiin saatiin koodista eliminoitua ylimääräisiä parametrisyöttöjä sekä tehtyä koodista eheämpää. Formik-komponentin sisällä olevilla alikomponenteilla ei tarvittu enää erikseen kutsua Formikin ominaisuuksia, sillä käyttöön saatiin Field-komponentti. Formikin tilan päivitys onnistui jatkossa automaattisesti Field-komponenttien avulla ja input-elementtejä saatiin näin ollen yksinkertaistettua. Käyttämällä Formik-komponentteja lomakkeista saatiin luotua koodissa selkeämpiä kokonaisuuksia, jotka helpottivat koodin lukemista ja käsittelyä jatkossa.

8 POHDINTAA

Tämän opinnäytetyön aiheena oli käsitellä koodin refaktorointa ja ylläpidettävyyttä. Työn aihe tuli toimeksiantajalta Profit Softwarelta. Opinnäytetyö esittelee refaktoroinnin käsitettä yhdessä esimerkkien avuin ja peilaa refaktoroidun koodin vaikutuksia koodin ylläpidettävyyteen. Työssä huomioidaan ylläpidettävyyden myös muun muassa refaktoroinnin taloudelliset vaikutukset ja vaikutukset tietoturvaan. Lisäksi opinnäytetyössä pohditaan refaktoroinnin automatisoinnin haasteita sekä refaktoroinnin onnistumisen arviontia.

Tämän opinnäytetyön pohjalta voidaan todeta, että refaktorointia ja sen vaikutuksien tärkeyttä osana ohjelmistokehitystä tulisi korostaa enemmän. Refaktoroinnin voi mieltää olevan toissijainen osa ohjelmistokehitystä, sillä se ei suoraan vaikuta ohjelmiston toimintaan tai esimerkiksi käyttöliittymän ulkoasuun. Refaktorointia olisi hyvä tehdä aina silloin, kun siihen on tarve ja mahdollisuus. Tekemättä jätetyt refaktoroinnit kasvattavat teknistä velkaa ja mitä enemmän tekninen velka koodissa kasvaa sitä enemmän sen maksaminen vie tulevaisuudessa aikaa ja luo haasteita koodin rappeuduttua. Näiden huomioiden myötä refaktorointi ja refaktoroimattomuus olisi hyvä ottaa huomioon myös projektimuotoisessa ja aikataulutetussa ohjelmistokehityksessä.

Refaktoroinnissa on myös haasteita. Refaktoroitaessa tulisi olla tieto siitä, milloin on refaktoroitu tarpeeksi. Ohjelmistokehitystiimien kesken olisi hyvä olla etukäteen sovitut raamit siitä, mitä refaktoroidaan ja miten. Näin ollen voidaan välttää subjektiivisia näkemyksiä refaktoroidessa sekä tukea ohjelmistokehittäjää hahmottamaan halutun tavoitteen. Sovitut määritelmät refaktoroinneista auttavat myös versiollisessa ohjelmistokehityksessä, jolloin kahden eri ohjelmistotuotteen välillä voidaan haluttaessa säilyttää yhteensopivuus, kun koodipohjat eivät pääse eroamaan liiaksi toisistaan.

Toinen mainittava haaste on automaattiset refaktorointityökalut. Työn teoriaosuudessa esiteltyjen tutkimusten sekä oman kokemukseni mukaan ohjelmistokehittäjien suhtautuminen refaktorointityökaluihin on skeptinen. Ilman luotettavia ja helposti omaksuttavia refaktorointityökaluja kaikki refaktoroinnit tulee tehdä manuaalisesti, mikä vie

huomattavasti aikaa verrattuna siihen, jos saman työn saisi tehtyä muutamalla klikkauksella. Jotta tulevaisuudessa automaattisiin refaktorointityökaluihin ei suhtauduttaisi niin varauksella, tulisi niiden ansaita laajasti ohjelmistokehittäjien luottamus.

Tämän opinnäytetyön teko sekä Profit Softwarella tekemäni refaktoroinnit opettivat minulle paitsi paljon refaktoroinnista myös koodin ylläpidettävyyden tekijöistä. Refaktoroinnin vaikutukset ohjelmistotuotteessa saattavat olla näkymättömiä, mutta vaikutukset koodin parissa työskentelylle merkittäviä. Ohjelmiston elinkaaresta ohjelmiston ylläpito on pisin ajanjakso, minkä tulisi motivoida kiinnittämään huomiota koodin ylläpidettävyyteen jo ohjelmiston kehitysvaiheessa. Sanonta: ”Milli vinossa pohjalla on metri huipulla”, kuvaa hyvin sitä, miten tekninen velkakin saattaa kasvaa koodissa ajan pitkään, mikäli siihen ei kiinnitetä tarpeeksi huomiota riittävän ajoissa. Jos koodi ei pysy ylläpidettävänä, sen parissa työskentely on hetki hetkeltä vaikeampaa ja aikanaan mahdotonta.

LÄHTEET

Barone, R. 2020. What are Libraries in Programming? Viitattu 18.2.2022. https://www.idtech.com/blog/what-are-libraries-in-coding

Bird, J. 2011. Lientz and Swanson on Software Maintenance. Viitattu 15.1.2022. https://dzone.com/articles/lientz-and-swanson-software

Brant, J. & Roberts, D. 1999. Teoksessa M. Fowler (toim.) Refactoring: improving the design of existing code. Addison-Wesley, 401.

Campbell, G. A. 2020. For secure code, maintainability matters. Viitattu 12.2.2022. https://blog.sonarsource.com/for-secure-code-maintainability-matters

CAST 2021. The 4 Types Of Software Maintenance & How They Help. Viitattu 18.11.2021. https://www.castsoftware.com/glossary/Four-Types-Of-Software-Maintenance-How-They-Help-Your-Organization-Preventive-Perfective-Adaptive-corrective

Clarusway 2021. What Is SDLC (Software Development Life Cycle). Viitattu 11.1.2022. https://clarusway.com/what-is-software-development-life-cycle/

Cleverti 2018. Why is code quality such a big deal for developers? Viitattu 17.1.2022. https://medium.com/@cleverti/why-is-code-quality-such-a-big-deal-for-developers-91bdace85d44

Cruz, N. 2019. Give your outdated libraries some respect. Viitattu 18.2.2022. https://medium.com/feedzaitech/give-your-outdated-libraries-some-respect-7dd74173b42e

Data Respons n.d. The Importance of Refactoring. Viitattu 7.1.2022. https://datarespons.com/the-importance-of-refactoring/

Dooley, J. F. 2017. Software development, design and coding : with patterns, debugging, unit testing, and refactoring (Second edition). Appress, 320.

Eilertsen, A. M. & Murphy, G. C. 2021a. The Usability (or Not) of Refactoring Tools. Proceedings - 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) 9.3.2021. Viitattu 20.1.2022. https://doi.org/10.1109/SANER50967.2021.00030

Eilertsen, A. M. & Murphy, G. C. 2021b. A study of refactorings during software change tasks. Journal of Software: Evolution and Process. Viitattu 20.1.2022. https://doi.org/10.1002/SMR.2378

Ferraiuolo, E. 2015. [RFC] React Intl v2, Issue #162. Viitattu 14.3.2022. https://github.com/formatjs/formatjs/issues/162

Format.JS, n.d. Components. Viitattu 14.3.2022. https://formatjs.io/docs/react-intl/components/

Formik n.d. useFormik(). Viitattu 15.3.2022. https://formik.org/docs/api/useFormik

Fowler, M. 1999. Refactoring : improving the design of existing code. Addison-Wesley, 53, 55-57, 66, 71-72, 75.

Fowler, M. 2019. TechnicalDebt. Viitattu 7.1.2022. https://mart

Gillis, A. S. 2021. What is Refactoring (Code Refactoring)? Viitattu 3.1.2022. https://searchapparchitecture.techtarget.com/definition/refactoring

Golubev, Y., Kurbatova, Z., AlOmar, E. A., Bryksin, T. & Mkaouer, M. W. 2017. One Thousand and One Stories: A Large-Scale Survey of Software Refactoring. Viitattu 19.1.2022. https://dl.acm.org/doi/10.1145/3468264.3473924

Gorbachenko, P. 2021. Code Refactoring: the Definitive Guide. Viitattu 20.2.2022. https://enkonix.com/blog/code-refactoring/

Grill, J. n.d. How to translate your React app with react-intl + Example. Viitattu 14.3.2022. https://www.codeandweb.com/babeledit/tutorials/how-to-translate-your-react-app-with-react-intl

Gupta, A. & Sharma, S. 2015. Software Maintenance: Challenges and Issues. International Journal of Computer Science Engineering 4. Viitattu 15.1.2022. http://www.ijcse.net/docs/IJCSE15-04-01-037.pdf

Hilton, P. 2018. The code maintenance problem. Viitattu 18.1.2022. https://hilton.org.uk/blog/code-maintenance-problem

IEEE Computer Society 2014. Guide to the Software Engineering Body of Knowledge. IEEE, 106-108. https://cs.fit.edu/~kgallagher/Schtick/Serious/SWEBOKv3.pdf

IntelliJ IDEA n.d. Code refactoring. Viitattu 20.1.2022. https://www.jetbrains.com/help/idea/refactoring-source-code.html

ISO/IEC/IEEE 14764:en 2006. Software Engineering - Software Life Cycle Processes - Maintenance. Viitattu 18.11.2021. https://www.iso.org/obp/ui/#iso:std:iso-iec:14764:ed-2:v1:en

Kuipers, T. 2016. Why you need to know about code maintainability. Viitattu 18.11.2021. https://www.oreilly.com/content/why-you-need-to-know-about-code-maintainability/

Maruyama, K. 2007. Secure Refactoring - Improving the Security Level of Existing Code. Proceedings of the Second International Conference on Software and Data Technologies SE. Viitattu 12.2.2022. https://www.researchgate.net/publication/220738149_Secure_Refactoring_-_Improving_the_Security_Level_of_Existing_Code_Improving_the_Security_Level_of_Existing_Code (Accessed: 12 February 2022).

Mengerink, J. 2013. How to Test Refactoring. Viitattu 20.2.2022. http://xunitpatterns.com.

Merrill, C. 2019. Software Maintenance: Perfective, Adaptive, Corrective & Preventive. Viitattu 13.1.2022. https://www.zibtek.com/blog/software-maintenance-understanding-the-4-main-types/

Moment.js 2020. Documentation. Viitattu 12.3.2022. https://momentjs.com/docs/#/-project-status/

Naveed, J., Zeeshan, A. & Ghulam, M. 2021. Historical Perspective of Software Refactoring Tools towards Future Solution. Journal of the Punjab University Historical Society 34, 177. Viitattu 19.1.2022. http://pu.edu.pk/images/journal/HistoryPStudies/PDF_Files/13-v34_1_2021.pdf

Omeyer, A. 2021. Software Maintenance Types: Corrective, Adaptive, Perfective, and Preventive. Viitattu 14.1.2022. https://hackernoon.com/what-do-you-need-to-know-about-software-maintenance-types-as-an-engineer-421335fl

OmniSci n.d. What is an Open Source Library? Viitattu 18.2.2022. https://www.omnisci.com/technical-glossary/open-source-library

Pal, T. 2021. Using the Visual Studio Code Refactoring Tools. Viitattu 19.1.2022. https://www.codeguru.com/tools/using-the-visual-studio-code-refactoring-tools/

Papadopoulo, A. n.d. Should Developers Use Third Party Libraries? Viitattu 18.2.2022. https://www.scalablepath.com/back-end/third-party-libraries

Pinto, G. & Kamei, F. 2013. What Programmers Say About Refactoring Tools? An Empirical Investigation of Stack Overflow. Viitattu 19.1.2022. https://dl.acm.org/doi/10.1145/2541348.2541357

Preston, M. 2021. 7 Phases of the System Development Life Cycle Guide. Viitattu 11.1.2022. https://www.clouddefense.ai/blog/system-development-life-cycle

React n.d.a. Context. Viitattu 15.3.2022. https://reactjs.org/docs/context.html

React n.d.b. Lists and Keys. Viitattu 15.3.2022. https://reactjs.org/docs/lists-and-keys.html

Sarin, H. 2022. Software Developer, Profit Software. Pori. Henkilökohtainen tiedonanto 17.2.2022.

SFS-ISO/IEC/IEEE 12207:en. Systems and software engineering - Software life cycle processes. 2020. Finnish Standards Association SFS. Helsinki: SFS. Viitattu 12.1.2022. https://online.sfs.fi/fi/index/tuotteet/SFS/ISO/ID2/1/953694.html.stx

Singh, Y., & Goel, B. 2007. A step towards software preventive maintenance. ACM SIGSOFT Software Engineering Notes 32, 1. https://www.researchgate.net/publication/220631405_A_step_towards_software_preventive_maintenance

Spinellis, D. 2006. Code quality : the open source perspective. Adobe Press, 1–7. https://books.google.com/books/about/Code_Quality.html?hl=fi&id=vEN-ckcdtCwC

Stone, S. 2018. Code Refactoring Best Practices: When (and When Not) to Do It. Viitattu 19.2.2022. https://www.altexsoft.com/blog/engineering/code-refactoring-best-practices-when-and-when-not-to-do-it/

Trucchia, F. & Romei, J. 2010. Pro PHP refactoring. Apress Media LLC. https://doi.org/10.1007/978-1-4302-2728-1

TypeScript n.d. Everyday Types. Viitattu 10.3.2022. https://www.typescriptlang.org/docs/handbook/2/everyday-types.html

Vieira, B. 2017. Maintainable Security. Viitattu 12.2.2022. https://medium.com/soft-wareimprovementgroup/maintainable-security-4bd4bf055438