

Antero Korvala

OHJELMISTON HUOLLETTAVUUS JA SEN LAIMINLYÖMISESTÄ AIHEUTU- VAT UHKAT

OHJELMISTON HUOLLETTAVUUS JA SEN LAIMINLYÖMISESTÄ AIHEUTU- VAT UHKAT

Antero Korvala
Opinnäytetyö
Kevät 2024
Tietotekniikan tutkinto-ohjelma
Oulun ammattikorkeakoulu

TIIVISTELMÄ

Oulun ammattikorkeakoulu

Tietotekniikan tutkinto-ohjelma, Ohjelmistokehityksen suuntautumisvaihtoehto

Tekijä(t): Antero Korvala

Opinnäytetyön nimi: Ohjelmiston huollettavuus ja sen laiminlyömisestä aiheutuvat uhkat

Työn ohjaaja(t): Meija Lohiniva

Työn valmistumislukukausi ja -vuosi: Kevät 2024

Sivumäärä: 28

Opinnäytetyössä suoritettiin tutkimusta ohjelmistojen huollettavuudesta ja huollosta teoreettisella sekä empiirisellä tutkimuksella. Työn tavoitteina olivat opin syventäminen teoreettisella ja käytännön tasolla huollettavuudesta, huollettavuuden toteuttamisesta ja ohjelmistohuollosta sekä siihen vaikuttavista tekijöistä.

Teoreettista tutkimusta suoritettiin perehtymällä aiheeseen verkkolähteiden, artikkeleiden ja opinnäytetöiden avulla, jota sovellettiin työelämässä tehtyä projektia hyödyntäen. Työn teoreettisessa tutkimuksessa perehdyttiin ohjelmistohuollon alalajeihin, milloin ja miksi kutakin toteutetaan ja miten kehitystiimin ulkoiset tekijät lopullisesti vaikuttavat toteutukseen. Tietoa laajennettiin myös huollettavuuden laaduista, niiden saavuttamisesta ja laiminlyönnin aiheuttamista uhkista. Työ oli pääosin teoreettista tutkielmaa, jonka lisäksi suoritettiin empiiristä tutkimusta työelämässä tehtävää huoltoprojektia hyödyntäen, jonka avulla saatiin tilaisuus analysoida huollettavuuden ja huollon toteutusta oikean elämän tilanteessa.

Tutkielman tuloksena saavutettiin teoreettinen pohja ohjelmistohuoltoon, huollettavuuden toteutukseen, huollon ja huollettavuuteen vaikuttaviin tekijöihin sekä käytännön oppia siitä, miten ohjelmistohuoltoa ja huollettavuutta toteutetaan ohjelmistokehitystiimissä. Tutkielmassa varmistuttiin huollettavuuden toteutuksen olennaisuudesta ohjelmistoa kehittävässä yrityksessä sekä siitä, kuinka paljon se lopullisesti vaikuttaa kehittäjiin ja muun henkilöstön jokapäiväiseen työhön.

Tämän tutkimuksen empiirisessä osassa toteutettiin huoltoprojektia, jonka pääosassa oli vanhan logiikan sisällytys uudessa toteutuksessa. Huollettavuuden toteuttamisen näkökulmasta jatkokehitystä voitaisiin tehdä tuottamalla täysin uusi ohjelmistokehitysprojehti, jossa vanhan toteutuksen analysoinnin sijasta suunniteltaisiin täysin uusi järjestelmä. Projektin suorituksessa saataisiin jälleen mahdollisuus oppia huollettavuudesta teoreettisella ja käytännön tasolla tutkimalla toteutuksen matkalla herääviä kysymyksiä sekä hyödyntämällä jo tässä tutkimuksessa opittua tietoa.

Asiasanat: Ohjelmistohuolto, ohjelmisto huollettavuus, ohjelmiston ymmärrettävyys

ABSTRACT

Oulu University of Applied Sciences
Degree Programme in Information Technology, Option of Software Development

Author(s): Antero Korvala
Title of thesis: Software maintainability and the risks caused by neglect
Supervisor(s): Meija Lohiniva
Term and year when the thesis was submitted: Spring 2024
Number of pages: 28

The focus point of this thesis was on software maintainability and software maintenance. The research was conducted by doing theoretical and empirical research by studying older theses, web articles and other web sources and then applying the learned knowledge through analysis and implementation of a software maintenance project conducted by my current employer. The theoretical research gave a deeper understanding of maintainability and software maintenance as a concept and how and why it should be implemented in the software lifecycle. We also learned what risks are associated with the neglect of software maintenance and maintainability and which shareholders directly and indirectly affect the implementation of maintenance and maintainability.

Keywords: Software maintenance, software maintainability, software development, software quality, readability, understandability, performance

SISÄLLYS

1	JOHDANTO	6
2	KOODIN HUOLLETTAVUUS JA SEN SUUNNITTELU	7
2.1	Huollettavuuden määrittely	7
2.2	Huolto ja ylläpito -kategoriat	8
2.2.1	Korjaava	8
2.2.2	Mukauttava	8
2.2.3	Parantava / kehittävä	9
2.2.4	Ennaltaehkäisevä	9
2.3	Huollettavuuden edut	9
2.4	Huollettavan ohjelmiston suunnittelu	11
3	HUOLLETTAVUUDEN SÄÄSTÖT JA LAIMINLYÖNNIN UHKAT	14
4	OHJELMISTON UUDELLEENKIRJOITUS JA ANALYSOINTI	16
4.1	Rakenne	16
4.2	Java ja Spring	18
4.3	Koodauskäytännöt ja työkalut	19
5	TULOKSET	22
6	POHDINTA	24
	LÄHTEET	26

1 JOHDANTO

Ohjelmiston huollettavuus voi terminä tuottaa hämmennystä monelle, joille aihe ei ole ennestään tuttu, sillä se on erittäin laaja käsite. Käsitteen laajuudesta huolimatta kaikkien tietotekniikan ammattilaisten tulisi tietää huollettavuuden perusteista ja sen laiminlyömisestä aiheutuvista ongelmista, koska ohjelmistohuolto on ylivoimaisesti aikaa vievin tehtävä ohjelmiston elinkaareissa, joka loppuu vasta kun ohjelmisto vihdoinkin lakkautetaan.

Tässä opinnäytetyössä pyritään avaamaan ohjelmistohuollettavuuden käsitettä teoreettisella tutkimuksella sekä hyödyntämällä työtehtävässä toteutettavaa huoltoprojektia. Teoreettisessa tutkimuksessa pyritään selvittämään, mitä ohjelmisto huollettavuus oikeasti on, miten sitä voidaan toteuttaa ohjelmistoprojekteissa ja miksi sitä ylipäättään halutaan toteuttaa. Empiirisessä tutkimuksessa toteutetaan työprojektia, jossa pyritään ymmärtämään, miten teoreettista tietoa voidaan soveltaa käytännössä sekä mitä ovat huollettavuuden toteutuksen todelliset esteet ja käytänteet. Projektin tavoitteena on uudelleenkirjoittaa palvelu, joka on toteutettu pilvinatiiviarkkitehtonista tai yleisemmin tunnettuna mikropalvelu-lähestymistapaa hyödyntäen, jossa yksi sovellus koostuu useammasta löysästi kytketystä ja itsenäisesti käyttöön otettavasta pienemmästä komponentista tai palvelusta. (IBM.) Uudelleenkirjoituksessa on käytettävä uudempaa teknologiaa ja kehitystekniikkaa, jotta palvelu pysyy yrityksen omien ja nykyteknologia standardien mukana.

Projektin lopullisessa analyysissä tutkitaan projektin saavutuksia ja mahdollisia parannuksia huollettavuuden, ylläpidettävyyden ja luettavuuden kannalta. Käsittelyssä on myös, miksi tämän tyylistä huoltoa ja huollettavuutta haluttaisiin toteuttaa, toteutuksen vaikuttajista sekä mitä uhkia ja menetyksiä sillä vältetään myös liiketoiminnan tasolla.

2 KOODIN HUOLLETTAVUUS JA SEN SUUNNITTELU

Koodin huollettavuutta tutkiessa on ensin määritettävä, mitä se on ja mitä ohjelmiston huolto on. Huollettavuuden ja huollon perustan määrittämisellä pyritään ymmärtämään, miten kumpaan näistä yleisesti suunnitellaan ja toteutetaan ohjelmistokehitysprojektin elinkaaren aikana. Tämän lisäksi tarvitaan ymmärrystä siitä, miten ohjelmistoa arvioidaan huollettavuuden näkökulmasta yhteisesti ymmärretyillä kriteereillä, jotta voidaan tuottaa ja suunnitella huollettavaa ohjelmistoa.

2.1 Huollettavuuden määrittely

Huollettavuus on yksi kahdeksasta ohjelmistolaadun ominaisuudesta, jotka kuuluvat kansainväliseen standardiin ISO/IEC 25010:2011 (Visser ym. 2016). Ohjelmiston huollettavuus tai ylläpidettävyyden käsite on laaja käsite, mutta yksinkertaisuudessaan se on ohjelmiston korjauksen, parannuksen ja koodin ymmärtämisen helppoutta. Näihin vaikuttaa suuresti koodin ja ohjelmiston muut laadut. (Sealights a.) Yksityiskohtaisemmin ajatellen se on ohjelmiston sopeutuminen ympäristöön, sen kykyä tukea muutoksia ja sen nopeaa korjattavuutta vian, häiriön tai katkoksen tapahtuessa. Voidakseen sanoa, että ohjelmisto on huollettava, sen täytyy täyttää laatu vaatimuksia muutettavuudessa, laajennettavuudessa, luotettavuudessa ja kestävydessä. Näiden lisäksi ohjelmistolla täytyy olla riittävät dokumentaatiot, jotta myös ihmiset, jotka eivät tunne ohjelmistoa ennestään, voivat helposti käyttää sitä. Koodin huollettavuutta toteutettaessa se ei saisi olla jälki ajatus, jota mietitään vasta ohjelmistoa kehittäessä, vaan sen pitäisi olla jo osana ohjelmisto suunnittelu prosessia. (FlexBase 2019.)

Huollettavuudella ilmaistaan ohjelmiston kehityksen ja muuttumisen helppoutta pitkällä tähtäimellä, mikä on erityisen tärkeää nykypäivän ketterässä ympäristössä. Huollettavuus täten mahdollistaa ohjelmistohuollon, joka aloitetaan vasta, kun tuote on toimitettu asiakkaalle. (Sealights a.) Huollettavuus on yleisesti hyväksytty osa ohjelmistokehitystä, sillä sen laiminlyöminen heti kehitysprosessin alussa tulee kasvattamaan projektin kustannuksia suuresti sen elinkaaren aikana. (FlexBase 2019.) Ohjelmiston huoltaminen yleisesti vie yli puolet kehitysbudjetista, joka tekee siitä prosessin kalleimman vaiheen. Tästä syystä huoltamisen suunnittelu kehityksen elinkaareen on tärkeää, jotta huoltoa voidaan toteuttaa mahdollisimman tehokkaasti. (Sealights a.)

Huoltaminen ketterässä projektissa vaatii vanhempien järjestelmien huoltamista uuden tuotteen kehityksen rinnalla. Esiintyvien ongelmien korjaus voi tuottaa yllätyksiä sprintin kehitysjonoon, jonka takia täsmällisen suunnitelman toteuttaminen ja sprinttien hallinnointi on vaikeampaa. Ketterässä kehityksessä yleinen dokumentaation puute voi myös hankaloittaa huoltoa, koska se tekee ohjelmiston ymmärtämisestä vaikeampaa niille, jotka eivät ennestään tunne ohjelmistoa. (Sealights a.)

2.2 Huolto ja ylläpito -kategoriat

Huolto ja ylläpito on jaettu neljään eri kategoriaan: korjaavaan, mukauttavaan, parantavaan ja ennaltaehkäisevään huoltoon, joita kaikkia käytetään jossain vaiheessa ohjelmiston elinkaaren aikana. Jokaisen järjestelmän odotetaan yleisesti tarvitsevan korjaavia huoltotoimenpiteitä, toisin kuin muita huollon kategorioita. Muiden huoltokategorioiden käyttö riippuu järjestelmästä, esimerkiksi täysin uusi järjestelmä vaatii täyden huoltokattauksen, jossa otetaan kaikki neljä kategoriaa huomioon toisin kuin jo olemassa olevalle järjestelmälle, jolle on tehty esimerkiksi vain pieni funktionaalinen muutos. Todellisuudessa tämä funktionaalinen muutos voi myös laajuudestaan riippumatta vaatia enemmän huoltotoimenpiteitä. (Hayes 2014.)

2.2.1 Korjaava

Korjaavalla huollolla tarkoitetaan nimensä mukaan huoltoa, joka vaatii ohjelmistovirheen tai jonkin muun ongelman korjaamista ohjelmistossa. Tähän kategoriaan kuuluvat myös tilanteet, jolloin ohjelmistossa on havaittu häiriö tai katkos. (FlexBase 2019.)

2.2.2 Mukauttava

Mukauttavassa huollossa järjestelmää huolletaan, kun muutoksia tapahtuu teknologiapinossa, kuten laitteiston, käyttöjärjestelmän tai alustan vaihtuminen. Ohjelmisto pitäisi suunnitella siten, että mukautuvuuden helppous ja tehokkuus pidetään mielessä. Tietyissä tilanteissa mukauttavaa huoltoa täytyy tehdä säännöllisesti, esimerkiksi kun kyseinen ohjelmisto on kaupallinen standardituote (nk. COTS, Commercial off-the-shelf). (FlexBase 2019.)

2.2.3 Parantava / kehittävä

Parantavalla huollolla viitataan tarkentavaan tai jalostavaan muutokseen ohjelmiston funktionaalisuudessa jo olemassa olevien ominaisuuksien lisäksi. Nämä yleisesti perustuvat käyttäjä vaatimuksiin ja jatkuvan parantamisen prosessiin, joka on yleisessä käytössä nykypäivän teollisuudessa. (FlexBase 2019.)

2.2.4 Ennaltaehkäisevä

Ennaltaehkäisevällä huollolla pyritään estämään mahdollisia tai jopa tuntemattomia virhetilanteita ja vikoja ohjelmistossa jalostamalla ohjelmistoa mahdollisimman korkealle tasolle (FlexBase 2019). Reaalimaailmassa virheettömän koodin tuottaminen ei ole mahdollista, sillä sen tuottaminen on erittäin haastavaa ja inhimillisiä virheitä sattuu jopa erittäin kokeneille kehittäjille. Virhetilanteet ja viat ovat myös yleisesti tuntemattomia, joka tekee tästä lähes saavuttamatonta nykYTEknologia ratkaisuilla, mutta ehkä tekoälyn kehittyessä tästä voidaan tehdä mahdollista. Vaikka täydellisen koodin tuottaminen olisi mahdollista nykYTEknologialla, sen tuottaminen ei ole toivottava ratkaisu, koska se aiheuttaisi erittäin suuren määrän työtä, sillä koodin monimutkaistuesssa virheettömän koodin tuottamisesta kehittyisi kohtuuttoman aikaa vievää. Sen sijaan on paljon ajallisesti ja rahallisesti tehokkaampaa tuottaa luettavaa ja huollettavaa koodia, jota voi tarvittaessa huoltaa virhetilanteiden esiintyessä (Dhanush 2023.)

2.3 Huollettavuuden edut

Huollettavuus on laatu ominaisuus, joka tarjoaa arkkitehdeille mitattavan liiketoiminta ja teknisen kehyksen, jota he voivat hyödyntää neuvotteluissa asiakkaiden ja liiketoiminta henkilöstön kanssa. Sen avulla arkkitehdit voivat vakuuttaa puhtaan koodin sekä ei-toiminnallisten vaatimusten eli huollettavuuden arvosta, joka usein jää vähemmän teknisen ja päätösvallan omaavan henkilöstön jälkiaatteenksi. Huollettavuuden yksi tärkeimmistä ominaisuuksista, joka ohjelmistoilla voi olla on mukautuvuus. Sen ansiosta voidaan vähentää vaadittavia ulkoisia toimenpiteitä ohjelmiston ympäristön muuttuessa, joka täten myös vähentää kehittäjien työmäärää ja nopeuttaa käyttöönotto aikoja. Modulaarisuus on toinen tärkeä ominaisuus ohjelmistolle, jolla vastataan siihen, kuinka irrotettu tai eristetty koodi on. Se myös paljastaa, onko järjestelmän suunnitelma dokumentoitu hyvin ja voiko

järjestelmään tehdä muutoksia ja korjauksia ilman, että se vaikuttaa ohjelmiston muihin osiin. (FlexBase 2019.)

Hyvien suunnittelu periaatteiden noudatus ohjelmistoa tehdessä parantaa ohjelmiston suorituskyvyn ennustetta. Hyvällä suorituskyvyllä voidaan indikoida huollettavuutta ja hyvien periaatteiden käyttöä ohjelmiston suunnitteluvaiheessa. (FlexBase 2019.) Asioita mitä ottaa ohjelmiston huollettavuudessa huomioon jo suunnittelu vaiheessa:

1. Tarvitseeko ohjelmisto korkeantasoisen suorituskyvyn?
2. Tarvitseeko ohjelmisto korkeantasoisen suorituskyvyn ja odotetaanko käyttäjien lukumäärän kasvavan, aja kuluessa?
3. Pitääkö ohjelmiston pystyä kommunikoimaan muiden ohjelmistojen kanssa?
4. Onko kilpailevia teknologiaympäristöjä?
5. Mitä ovat nousevat teknologiatrendit käyttöjärjestelmissä, laitteistoissa ja kielissä?
6. Ovatko vaatimukset tarpeeksi modulaariset täyttääkseen modulaarisen arkkitehtonisen suunnitelman?
7. Onko kirjoitetusta koodista helppo etsiä ja poistaa virheitä?
8. Ymmärtävätkö kehittäjät kommentointi ja dokumentointi -standardeja, joita pitäisi seurata?
9. Aiheuttaako puutteellinen dokumentaatio ja asiaankuuluva kommentointi hämmennystä uusille kehittäjille koodin kanssa työskennellessä?
10. Ovatko ohjelmistokomponentit kehitetty tarpeeksi riippumattomaksi toisiin komponentteihin?
11. Onko uuden käyttöliittymän tekemiseen kuluva aika tunnettu?
12. Ymmärtääkö kehitystiimi koodistandardit, joita heidän pitää seurata?
13. Onko laatua varmistavaa testausta?
14. Onko käytössä oleva toimitusmalli jatkuva?

Edellä mainitut ominaisuudet osoittautuvat huollettavuuden kriittisiksi osiksi niin sanottua optimaalista ohjelmistoa tehdessä. Nämä ominaisuudet auttavat lähestymään huollettavuuden adoptioimista organisoidulla tavalla, jonka lisäksi ne auttavat saamaan päätösten tekijöiden huomion. Nämä ominaisuudet auttavat myös selvittämään ei-toiminnalliset vaatimukset, joiden yksi keskeisistä tyypeistä on huollettavuus. (FlexBase 2019.)

Uudemmissa projekteissa on helpompaa ja suoraviivaisempaa investoida enemmän aikaa uusien ominaisuuksien kehittämiseen, koska ne eivät ole vielä ehtineet kerätä teknistä velkaa. Teknisen

velan keräämisellä tarkoitetaan yleisellä tasolla uhrauksia, joita tehdään ohjelmistokehityksessä nopean tuotannon aikaansaamiseksi. Näissä tilanteissa kehittäjä tekee nopeaa työtä, uhratessaan ohjelmiston laadun, tuottaakseen tuloksia (Sealights b.) Ohjelmisto kehityksessä vähemmän painavia tehtäviä, kuten dokumentointia, testausta ja refaktorointia helposti laiminlyödään projektin loppuun asti. Jättämällä näitä keskeisiä asioita käsittelemättä, pyritään säästämään lyhyellä tähtämällä näihin kuluvaan aikaa. Mutta tämän laiminlyönnin ansiosta ohjelmistolle kertyykin teknistä velkaa, jota joudutaan hoitamaan ohjelmisto huolloilla. Jos ohjelmistoa ei ole suunniteltu pitäen huollettavuutta mielessä ja kyseiset asiat on jätetty huomioimatta, tulee huoltoon kulutettava aika olemaan jopa nelinkertainen verrattuna itse kehitysprosessiin käytettyyn aikaan. Kertynyt tekninen velka tekee usein ohjelmistojen muokkaamisesta kohtuuttoman aikaa vievää, jolloin yleisesti päätetään korvata ohjelmisto täysin uudella huoltamisen sijasta. Kehittämällä huollettavaa ohjelmistoa voidaan minimoida teknistä velkaa, sillä se vähentää tulevien ohjelmisto muutoksien vaikutusta. (Crouch.)

2.4 Huollettavan ohjelmiston suunnittelu

Huollettavan ohjelmiston suunnittelu on elintärkeää yritykselle, jos se haluaa pysyä kilpailukykyisenä iikehittivässä ohjelmistoteollisuudessa. Tämän saavuttamiseksi on tärkeää ymmärtää asiakkaan tarpeet ja tehdä se mahdollisimman ajoissa, jotta asiakkaan odotukset voidaan toteuttaa mahdollisimman helposti. Näitä voidaan selvittää esimerkiksi haastatteluilla keskeisten sidosryhmien tai potentiaalisen asiakkaan kanssa. Selkeästi asetetut tavoitteet ohjelmisto suunnittelussa tuottavat helposti seurattavaa dataa siitä mitä pitää vielä parantaa ja missä. Uusien teknologioiden mukana pysymisellä taataan se, että tuotteesta ei tule liian vanhanaikainen, joka johtaisi tuotteen kilpailukyvyttömyyteen ja ajan myötä lopetukseen. Tämä on erityisen tärkeää pienemmille yrityksille tai yrityksille, jotka toimivat maissa, joissa on vähemmän asiakaskuntaa. (Digital Delivery 2021.)

Joustavuuden sisällytys ohjelmiston alusta saakka vähentää ongelmia pitkällä tähtämällä. Kuten ongelmia tilanteissa, joissa ohjelmisto tarvitsee jonkinlaisen merkittävän muutoksen. Tämän laiminlyöminen vaikeuttaisi minkään uuden toteuttamista ja ylläpitoa, joka johtaa suurempiin kuluihin. Ylisuunnittelu on myös yleinen ansa, jota kannattaa välttää ohjelmistoa kehittäessä. Tällä vältetään ohjelmiston ylimääräistä monimutkaisuutta arkkitehtuurisella ja koodikanta tasolla. Jos

yksinkertaisuutta ei pyritä säilyttämään, kaikki ohjelmiston kanssa työskentelevät joutuvat jatkossa kuluttamaan paljon aikaa ja täten rahaa ohjelmiston tutkimiseen ja testaamiseen. (Digital Delivery 2021.)

Kun ohjelmistokehittäjät ajattelevat, huollettavuutta tai ylläpidettävyyttä moni ajattelee sitä koodin puhtauteen ja luettavuuteen. Se on mahdollisesti yksi esillä olevimmista ja puhutuimmista ominaisuuksista huollettavuudelle, jolla on myös suuri merkitys jokapäiväisessä kehittäjän työssä. Luettavan koodin ominaisuuksista on monia mielipiteitä ja standardeja, joista keskustellaan jatkuvasti kyseisessä harraste ja ammattimaailmassa. Mutta yleisellä tasolla, mitä luettavampi koodi on sitä enemmän se auttaa muita kehittäjiä ymmärtämään tuottamaasi koodi helpommin, joka täten nopeuttaa muiden kehittäjien työtä. Luettavamman koodin ansiosta kehittäjien on myös helpompaa pitää ohjelmisto ajan tasalla ja pitää huolta siitä, että ohjelmisto on helpompi siirtää toiselle alustalle tarvittaessa. (Digital Delivery 2021.)

Modulaariset komponentit, jotka omaavat hyvin määritellyn käyttöliittymän tekevät niiden uudelleenkäytämisestä helpompaa muissa projekteissa ja tekee niiden muokkaamisesta helpompaa tulevaisuudessa. Tällä säästetään kehittäjien ja asiakkaiden aikaa, joka voidaan hyväksikäyttää muualla projektissa. (Digital Delivery 2021.)

Jatkuvan integroinnin adoptioiminen on myös otettava huomioon ajoissa. Parhaita käytäntöjä käyttäen jo ennen ohjelmiston tuotantoon käyttöönottoa voi säästää kehitystiimiltä paljon aikaa pitkällä mittakaavalla sillä jatkuvan integraation työkaluilla voidaan huomattavasti nopeuttaa ohjelmiston rakentamista, kääntämistä ja käyttöönottoa. Skaalautuva tiedon varastointi ratkaisu on myös hyvin tärkeä, sillä se varmistaa sen, että kun yritys kasvaa ja näin ollen myös sen tarpeet niin tiedon varastointia ei rajoita sille alussa varattu kapasiteetti tai sen käyttöönotossa tehdyt suunnittelu virheet. (Digital Delivery 2021.)

Koodi laadun aiheuttamat huoltokulut voidaan yrittää pitää mahdollisimman pienenä, sisällyttämällä yleisesti käytettyjä koodin laatua varmistavia menetelmiä. Nämä ovat todella yleisiä käytäntöjä, joita suoritetaan eri tyylillä riippuen yrityksestä. Ensimmäinen menetelmä on koodikatselmointi, jossa muut kehittäjät käyvät koodia rivi riviltä lävitse etsien mahdollisia virheitä. Näitä voidaan suorittaa palaveri tyylisesti erittäin tarkasti tai yleisesti käytetyllä vetopyynnöllä, jossa kehittäjä pyytää muita kehittäjiä tai arkiston valtuutettuja tarkistamaan toimitettua koodia. Koodin tarkistukset voivat

poistaa jopa 90 % koodissa olevista virheistä ennen ensimmäistä testi ajoa. Tällä on suuri merkitys rahallisesti, sillä näin estetään virheellisen ohjelmiston julkaisu loppukäyttäjille.

Toinen menetelmä on pariohjelmointi, jossa nimensä mukaan on kaksi ohjelmoijaa, jotka toimivat yhteistyössä oman roolinsa mukaan. Toinen kehittäjä toimii koodarina ja toinen kehittäjä arvioi tätä kyseistä koodi sen syntyessä rivi riviltä. Näitä rooleja vaihdellaan jatkumona, kunnes ominaisuus on kehittäjien ja vaatimuksien mukaan valmis. Tämä antaa arvioijan roolissa olevalle mahdollisuuden ajatella ohjelmisto ratkaisua korkeammalla tasolla sekä sen strategista suuntaa esimerkiksi, miten ratkaisu sopii muuhun koodin, mitä parannuksia sille voisi tehdä ja ratkaisun huollettavuutta. Vaikka molemmat näistä ratkaisuista tuovat tietynlaista arvoa kehitystiimille ne vaativat valtavan määrän keskittymistä ja kurinalaisuutta. Suuri etu näissä menetelmissä on se, että koodikanta tulee tutummaksi koko kehitystiimille, jolloin vältetään tilanteita, jossa vain yksi ihminen tietää koodikannan sisällön. Tämä voi olla myös erityisen arvokas perehdytys menetelmä uusille kehittäjille, jotka eivät ole tuttuja koodikannan kanssa. (Crouch.)

3 HUOLLETTAVUUDEN SÄÄSTÖT JA LAIMINLYÖNNIN UHKAT

Ohjelmiston huoltovaihe voi yleisesti kestää kymmenen vuotta tai enemmän. Tämän aikana ohjelmisto tuottaa ongelmia jatkuvan tuotannon virtauksessa, jotka vaativat ratkaisuja korjaavan tai muokautuvan huollon tavoin. Jonka lisäksi parantavia ehdotuksia ja pyyntöjä joudutaan ratkomaan parantavalla huollolla. Näiden ratkaisujen tehokkuus on tärkeää sidosryhmille. Huoltoon kulutettu vaiva pienenee, kun ratkaisut voidaan suorittaa nopeasti ja helposti. Tämä pienempi huolto taakka voi johtaa täten pienempiin huoltokuluihin sillä vähemmän huoltoa suorittavaa henkilöstöä tarvitaan. Mutta jos huoltoa suorittavan henkilöstön määrä pidetään samana niin he suorittavat huollot nopeammin ja säästävät enemmän aikaa uuden funktionaalisuuden luomiseen. Loppujen lopuksi ongelmien ratkaisu ja parantavat ratkaisut eivät saisi olla hitaita ja hankalia, koska tämä voi johtaa aikatauluista myöhästymiseen tai järjestelmän käyttämättömyyteen. (Visser ym. 2016.)

SIG: n (SIG, Software Improvement Group) mukaan ongelmien ratkaisu ja parantavien ratkaisujen tekeminen on kaksi kertaa nopeampaa järjestelmissä, joissa on keskivertoa parempi huollettavuus verrattuna järjestelmiin, joilla on keskivertoa huonompi huollettavuus. Tämä on merkittävä ero kilpailukykyisessä markkinassa, jossa tämä merkitsee ensimmäisenä olemista markkinoilla tai olemista jopa kuukausia kilpailijan edellä. (Visser ym. 2016.)

Huollettavuus toimii myös mahdollistajana muille ohjelmisto laaduille. Yleisesti ottaen ohjelmisto, johon on, helppo tehdä huoltoa on myös helppo tehdä parannuksia muihin ohjelmiston seitsemään laatu kategoriaan, jotka ovat määritetty ISO 25010 standardissa. Helpommin huollettava ohjelmisto on myös vakaampi, koska huollettavammissa ohjelmistoissa muutokset aiheuttavat vähemmän yllättäviä sivuvaikutuksia kuin ohjelmistoissa, jotka ovat sekavia ja joita on vaikea analysoida ja testata. (Visser ym. 2016.)

Huollettavan koodin ohjeita (Visser ym. 2016):

1. Kirjoita lyhyempiä koodin pätkiä, koska lyhyemmät koodin pätkät tai metodit on helpompi analysoida, testata ja uudelleen käyttää.
2. Kirjoita yksinkertaisempaa koodia, jotta koodissa on vähemmän päätepisteitä ja sitä on helpompi analysoida ja testata.
3. Vältä saman koodin uudelleen kirjoittamista, koska jokainen versio vaatii muutoksia ja toistuvuus johtaa regressiovirheisiin.

4. Tee käyttöliittymistä mahdollisimman pieniä, koska yksiköt, jotka vaativat vähemmän parametrejä on helpompi testata ja uudelleen käyttää.
5. Löysästi kytkettyjä moduuleja ja korkeantason komponentteja on helpompi muokata ja se johtaa modulaarisempaan järjestelmään.
6. Pidä arkkitehtuurinen tasapaino suunnittelemalla sopiva määrä komponentteja, jotka ovat myös sopivan ja yhtenäisen kokoisia. Tällä säilytetään modulaarisuutta ja muokkauksien helppoutta.
7. Pidä koodikanta pienenä, jotta koodia on helpompi analysoida, muuttaa ja testata. Tämän lisäksi huoltotehokkuus koodiriviä kohti on pienempi isoissa järjestelmissä verrattuna pienempiin järjestelmiin.
8. Hyödynnä automaatio testauksen lähes välitöntä palautetta muokattavuuden tehokkuudesta.
9. Puhdas koodi tekee huoltamisesta tehokkaampaa uusille tiimiläisille, jotka eivät tunne koodikantaa ennestään. Turhien merkkauksien kuten TODOs merkkauksien ja kuolleen koodin poistaminen parantaa koodikannan luettavuutta ja uusien kehittäjien tuotteliaisuutta.

Huollon toteutus yleisesti riippuu liiketoiminta puolen päätöksistä. Joissain tilanteissa liiketoiminta puoli voi pakottaa kehitystiimit tekemään uusia ominaisuuksia, vaikka ohjelmistot tarvitsisivat huoltoa ennen kaikkea. Tämä huollon laiminlyöminen tekee uusien ominaisuuksien toteuttamisesta vaikeampaa ja hitaampaa. Liiketoiminta puolen pitäisi tietää huoltamisen tärkeydestä ja sen laiminlyömisestä aiheutuvasta teknisestä velasta. Tämä tekninen velka tulee näkymään ohjelmiston ikääntymisellä, johon kuuluu suorituskyvyn laskeminen, ja ominaisuuksien toteuttamisen vaikeudet.

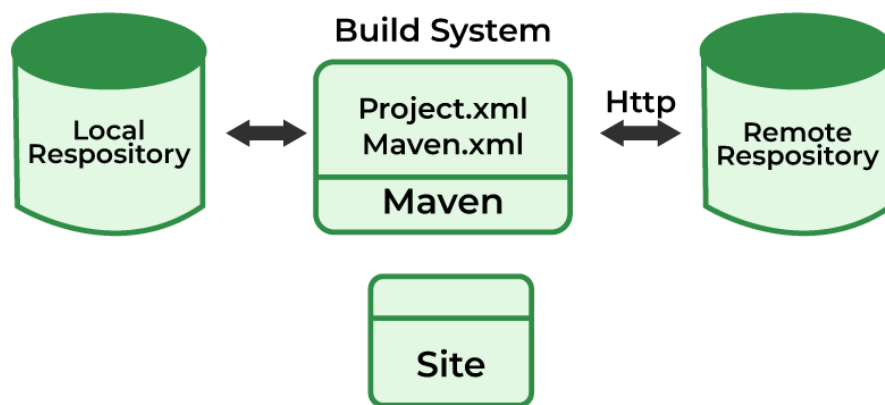
4 OHJELMISTON UUELLEENKIRJOITUS JA ANALYSOINTI

Tämä luku analysoi työssä tehtyä projektia ja sen huollettavuuden toteuttamista. Projekti itsessään on huoltotoimenpide, jossa toteutetaan jokaista huoltokategoriaa. Projektin tavoitteena on uudelleenkirjoittaa palvelu, joka käyttää HTTP(s)-protokollaa vastaanottamaan ja lähettämään pyyntöjä. Palvelun sisäinen funktionaalisuus ei ole tätä opinnäytetyötä varten tärkeä, joten sitä ei oteta huomioon. Huomion kohteena projektissa on se, miten ohjelmiston toteutettu rakenne ja sen rakentamisessa hyödynnetyt teknologiset ratkaisut vaikuttavat ohjelmiston lopulliseen huollettavuuteen.

Ohjelmiston uudelleenkirjoitus toteutettiin, koska ohjelmisto käytti vanhempaa teknologiaa, joka ei täytä tulevaisuudessa yrityksen tietoturva, suorituskyky ja huollettavuus -standardeja. Ennen uudelleenkirjoituksen aloittamista ohjelmistoa analysoitiin vanhemman kehittäjän kanssa testaamalla sen rajapintaa ja tutkimalla, miten ohjelmisto toimii sisäisesti sen hetken toteutuksella, miksi se on toteutettu juuri niin sekä mitä kaikkea voidaan säilyttää, poistaa tai parantaa, jotta uudesta toteutuksesta tulisi laadukkaampi kaikin mahdollisin tavoin ilman, että rikotaan taaksepäin yhteensopivuutta. Analysoinnin tuloksista ilmeni, että ohjelmisto oli ylisuunniteltu. Tämä näkyi koodin turhassa monimutkaisuudessa ja turhan koodin määrässä, joka oli valmistettu siltä varalta, jos palvelua olisi tulevaisuudessa laajennettu. Koodi oli myös eritelty kahteen pakettiin, joista toista hyödynnettiin ulkoisen kirjaston tavoin, mikä ei ole tarpeellista, sillä ohjelmiston kokonaisvaltainen toiminto ei ole niin laaja tai monimutkainen. Tämä monimutkaisuus voi myös osin johtua sen ajan teknologisista rajoitteista ja arkkitehtuurisista päätöksistä, jolloin tämän päivän tekniikoita ja teknologioita ei olisi voitu käyttää, koska ne olivat vielä kehitysvaiheessa tai niiden toimivuutta ei oltu vielä todistettu.

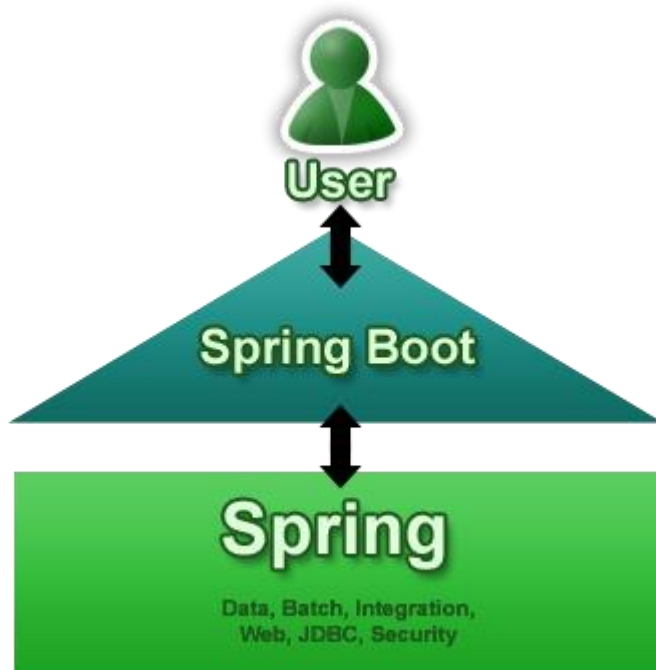
4.1 Rakenne

Ohjelmisto on Apache Maven -projekti, jonka koodi on kirjoitettu Java-ohjelmointikielellä. Apache Maven on projektinhallintatyökalu, jota päämääräisesti käytetään Java-pohjaisten projektien rakentamiseen ja hallitsemiseen. Se helpottaa Java-kehittäjien jokapäiväistä työtä ja tekee Java-projektien kokonaisuuden ymmärtämisestä helpompaa. Maven-arkkitehtuuri perustuu niin sanottuun POM.XML -tiedostoon, joka sisältää ne Java-luokat, resurssit ja muut riippuvaisuudet, joiden perusteella Maven lataa kaikki projektin riippuvaisuudet paikalliseen kansioon, joko keskus- tai etäarkistosta. (Kuva 1.) (GeeksForGeeks 2023.)



KUVA 1. Maven-arkkitehtuuri, .XML-riippuvuuksien haku ja käyttö (GeeksForGeeks 2023).

Java-koodissa hyödynnettiin Spring -sovelluskehystä ja Spring Boot -laajennusta (Spring -sovelluskehys ja Spring Boot tuovat kattavan ohjelmointi ja kokoonpanomallin (Kuva 2) (Spring.)), joka on vielä paketoitu yrityksen omaan laajennukseen, jolla tuotiin yrityksen omaa toiminnallisuutta ohjelmistoon.



KUVA 2. Vuorovaikutus käyttäjän, Spring Bootin ja Spring -sovelluskehyn välillä (Webb & Syer 2013).

Projektin Java-koodi oli kirjoitettu ohjelmointityylillä, joka teki tiedonsiirrosta synkronisen. Synkronisessa tiedonsiirrossa puhutaan estävästä arkkitehtuurista, jossa kaikki datan siirto hoituu yhden säikeen kautta. Tämä tarkoittaa sitä, että kaikki operaatiot suoritetaan yksi kerrallaan siinä järjestyksessä, jolloin kaikki muut operaatiot ovat estetty, kunnes ensimmäisenä oleva on suoritettu. (Bevans 2023.)

4.2 Java ja Spring

Uudelleenkirjoituksen pääasiallisena tavoitteena oli päivittää projektissa käytetty Java-versio, yrityksen Spring Boot -kääre ja täten myös Spring Boot -versio. Java-version päivitykseen on monia syitä mutta yksi isoimmista on päivitetyn version eli Java 17: sta ollessa tämän hetken "LTS" (Long-Term Support) -versio. Java 17: sta tullaan siis tukemaan ainakin seuraavat kolme vuotta toisin kuin Java 1.8 jonka tukeminen lopetettiin Maalikuussa 2022 (Kuva 3). Tämän lisäksi Java 17 tarjoaa parempaa suorituskkyä ja tehostettua turvallisuutta, pienentämällä käynnistämisaikoja, vähentämällä automaattisen roskankeruun aiheuttamia pysäytyksiä, olemalla tehokkaampi käsittelemään suurenmittakaavan ohjelmistoja, sisällyttämällä tuen parannelulle TLS 1.3: lle, päivittämällä kryptografiset algoritmit, ja korjaamalla potentiaalisia haavoittuvaisuuksia, jotka on löydetty aikaisemmassa versiossa (Atci 2023). Täten vanhan toteutuksen käyttämä Java 1.8 päivitettiin Java 17: sta.

Oracle Java SE Support Roadmap**†		
Release	GA Date	Premier Support Until
8 (LTS)**	March 2014	March 2022
17 (LTS)	September 2021	September 2026****

KUVA 3. Oracle Java tuki kartta (Oracle 2023).

Spring Boot myös päivitettiin versiosta 1.5.x versioon 2.7.x mutta yrityksen päivitettyjen teknologiavaatimusten mukaan tämän päivitys ei ole enää suositusten mukainen. Tämä tarkoittaa sitä, että versio 2.7.x on nyt vain vähimmäisvaatimus ja suosituksena on versio 3.x.x. Tämä perusteella voidaan olettaa, että päivitys versioon 3.x.x tullaan todennäköisesti tekemään lähitulevaisuudessa. Spring Boot 3 perustuu Spring 6 -kehykseen, joten se sisältää kaikki sen uudet ominaisuudet ja parannukset. Kokonaisuudessaan Spring Boot 3 tuo laajan kattauksen suorituskkyä parantavia muutoksia ja optimointeja. Ne tehostavat ohjelmiston reagoitukkyä ja tehokkuutta sekä vähentää käynnistysaikoja, minimoi muistinkäyttöä, ja optimoi resurssien käyttöä. Tämän lisäksi kevyemmän

käyttöympäristön vuoksi ohjelmistojen skaalautuvuus ja reagoiva tunne paranee, joka johtaa parempaa käyttäjä kokemukseen ja asiakastyytyväisyyteen. Spring Boot: n päivitys versioon 3.x.x, mahdollistaisi myös Java 17 etujen täyden käytön, Spring Boot 3: n vaatiessa joko Java 17 tai uudemman version. (Bitar ym. 2023.)

4.3 Koodauskäytännöt ja työkalut

Toissijainen tavoite uudelleenkirjoituksessa oli koodin luettavuuden parantaminen ja koodin tehostaminen teknologia päivityksien tuomien tehostuksien lisäksi. Tätä lähdettiin tavoittamaan eri tavoilla mutta suurimpia muutoksia oli kaksijakoisen arkkitehtuurisen ratkaisun eliminointi. Tämä kaksijakoisuus ei ollut tarpeellista uudessa toteutuksessa, jonka ansiosta koodikannasta saatiin paljon tiiviimpi. Tämä täten parantaa palvelun huollettavuutta kaiken ollessa samassa paketissa toisin kuin aikaisemmassa toteutuksessa. Myös luettavuus paranee, kun ei tarvitse siirtyä projektista toiseen seuratta ohjelmiston virtausta. Luettavuutta parannettiin myös hyödyntämällä Spring-kehityksen tarjoamia paketteja ja Project Lombok -kirjastoa, joilla rakennettiin REST käyttöliittymät sekä koodin kokonaisrakenne Reaktiivista ohjelmointityyliä käyttäen.

Project Lombok tai Lombok on Java-kirjasto, joka tarjoaa kommentaareja, joilla voidaan automaattisesti tuottaa näennäisesti toistuvaa koodia Java-luokille vaikuttamatta ohjelmiston suorituskäytännön (Kuva 4 ja 5). Lombokin pääasiallinen tarkoitus on koodin kokonaismäärän väheneminen, joka tekee luokista suppeampia ja helpommin luettavia. Samalla se vähentää koodaamisessa kuluvaa aikaa ja toistuvia ja virhealttiita tehtäviä. (Sahoo 2023.)

```

public class Employee {

    private Integer employeeId;
    private String name;
    private String company;
    private String emailId;

    public Employee() {}

    public Employee(Integer employeeId, String name,
                    String company, String emailId)
    {
        super();
        this.employeeId = employeeId;
        this.name = name;
        this.company = company;
        this.emailId = emailId;
    }

    public Integer getEmployeeId() { return employeeId; }

    public void setEmployeeId(Integer employeeId)
    {
        this.employeeId = employeeId;
    }

    public String getName() { return name; }

    public void setName(String name) { this.name = name; }

    public String getCompany() { return company; }

    public void setCompany(String company)
    {
        this.company = company;
    }

    public String getEmailId() { return emailId; }

    public void setEmailId(String emailId)
    {
        this.emailId = emailId;
    }

    @Override public String toString()
    {
        return "Employee ["
            + "employeeId=" + employeeId + ", name=" + name
            + ", "
            + "company=" + company + ", emailId=" + emailId
            + "]\n";
    }
}

```

KUVA 4. Java-luokan toteutus puhtaalla Javalla (GeeksForGeeks 2023).

```

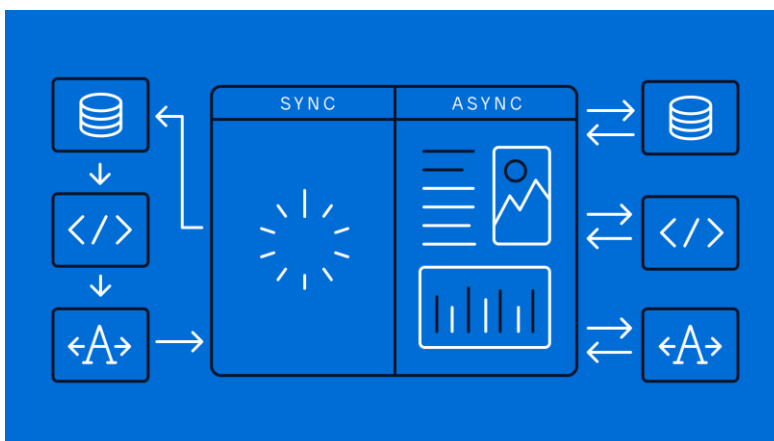
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;
import lombok.ToString;

@NoArgsConstructor
@AllArgsConstructor
@ToString
public class Employee {
    private @Getter @Setter Integer employeeId;
    private @Getter @Setter String name;
    private @Getter @Setter String company;
    private @Getter @Setter String emailId;
}

```

KUVA 5. Java-luokan toteutus Project Lombok Kirjastoa käyttäen (GeeksForGeeks 2023).

Uudessa toteutuksessa käytetty reaktiivinen ohjelmointi on deklarativinen ohjelmointimalli, joka perustuu asynkroniseen tapahtuma prosessointiin ja data virtaan. Asynkroninen prosessointi toisin kuin synkroninen (Kuva 5) on tapahtuman prosessointia ilman, että estetään muiden tapahtumien prosessointia. (Otta 2023.) Se sallii pitkien prosessien aloituksen ilman, että estetään muiden tehtävien ja tapahtumien suoritusta samanaikaisesti. Tämä tekee ohjelmistosta erittäin reagoivan, skaalautuvan ja joustavan. Reaktiivinen ohjelmointimalli keskittyy täten datan virtauksiin ja muutoksiin ohjelmistossa ja se perustuu tapahtumien reagoivaan ideologiaan toisin kuin tapahtuma virtauksien hallitsemiseen. Reaktiivisilla järjestelmillä on myös kyky käsitellä epäonnistumisia ilman, että menetetään funktionaalisuutta tai dataa, jonka lisäksi ne ovat erittäin skaalautuvia vaikuttamatta ohjelmiston suorituskykyyn tai laatuun. (Glushach 2023.)



KUVA 6. Synkroninen vs. Synkroninen tapahtuma prosessointi ja data virta (Bevans 2023).

5 TULOKSET

Tämän opinnäytetyön ohessa tehdyn kehitysprojektin korkean tason tavoitteet olivat päivittää kyseisen palvelun käyttämät teknologiat. Pääasiallisesti projektin tavoitteet on saavutettu, mutta opinnäytetyössä asetetuissa sivutavoitteissa on vielä paranneltavaa. Sivutavoitteina oli huollettavuuden parantaminen ohjelmiston arkkitehtuurin, koodityylin ja datavirtauksen muutoksilla. Sivutavoitteissa on ja tulee aina olemaan paranneltavaa omasta tai jonkin muun tahon mielestä, sillä täydellistä ja täydellisen luettavaa koodia tai virheetöntä ohjelmistoa ei ole mahdollista tuottaa. Jokainen koodimuutos tulee aiheuttamaan sivuvaikutuksia jossain ohjelmiston osissa, joka jää huomioimatta vahingossa tai jopa tarkoituksella.

Yrityksen säännösten ja kehitystiimin sisäisten päätösten perusteella tehdyt muutokset tuottivat suurta kehitystä projektin huollettavuudelle monessa eri kategoriassa. Palvelun käyttöikää nostettiin usealla vuodella, päivittämällä suunnitelman mukaisesti käytössä olevat teknologiat uudempiin versioihin yrityksen teknologiasäännösten mukaisesti. Ohjelmiston ymmärrettävyyttä parannettiin yksinkertaistamalla palvelun arkkitehtuurista ratkaisua, jossa kaksijakoinen koodikanta yhdistettiin yhdeksi kokonaisuudeksi. Koodin luettavuus selkeni vähentämällä koodin kokonaismäärää, virtaviivaistamalla sekä yhdenmukaistamalla sitä käyttämällä yleisesti parhaita käytäntöjä, yrityksen omia ohjelmistokehityksen säännöksiä, päivitettyä Lombok-kirjastoa sekä Reactor Corea. Kaikki edellä mainitut muutokset on vahvistettu tiimin sisäisillä koodinkatselmoineilla yhtenä laadunvarmistuksen osana.

Näiden muutosten pitäisi teoriassa tehdä myös palvelun suorituskyvystä paremman, teknologiatehokkuuden, koodin vähenemisen ja tehostumisen kautta. Tämä voitaisiin varmistaa suorituskykytestausvaiheissa, joissa testataan muun muassa muistinkäyttöä, prosessointitehoa, vasteaikoja ja turvallisuutta. Aikaisemman toteutuksen ollessa suhteellisen vanha, ei dokumentaatiota aikaisemmista suorituskykytestausraporteista ole saatavilla. Siksi teoreettisen suorituskyvyn paranemista ei voida helposti varmistaa.

Osa suorituskykyyn vaikuttavista ratkaisuista oli jätettävä ennalleen, jotta palvelu säilytettäisiin taaksepäin yhteensopivana. Nämä ennalleen jätetyt suorituskyvyn eri ominaisuudet jäivät projektin suurimmiksi huolenaiheiksi, koska suorituskyky vaikuttaa suoraan käyttäjäkokemukseen. Käyttäjäkokemus ja siitä saadut palautteet asiakaspalvelun tai muun lähteen kautta toimivat suurena

tekijänä huoltotoimenpiteiden toimeksiannossa. Tietoturvallisuus on myös käyttäjäkokemukseen ja turvallisuuteen liittyvä piirre. Tässä projektissa tietoturvallisuudesta tai yksityiskohtaisesta toteutuksesta ei keskusteltu, siltä varalta, että se rikkoisi salassapitosopimuksia. Mutta tietoturvallisuus on jäänyt toteutuksessa vielä esille, koska toteutus hyödyntää vanhan toteutuksen tietoturvaratkaisua, joka on kyseenalaistuksen kohteena meiltä kehittäjiltä sekä arkkitehtien puolelta. Projekti toteutettiin tiivistetyllä versiolla vanhan toteutuksen tietoturvaratkaisusta pienin muokkauksin. Ratkaisu on kuitenkin edelleen arkkitehtien ja muiden osapuolien keskustelun aiheena. Tästä syystä on hyvin mahdollista, että ratkaisu tullaan uudelleenkirjoittamaan tulevaisuudessa.

Mahdollisten suorituskyky- ja tietoturvaongelmien lisäksi uskon, että luettavuutta ja ymmärrettävyyttä olisi vielä voinut parantaa. Yhtenä esimerkkinä voin käyttää kokoonpanoluokkien rakennetta ja monimutkaisuutta. Nämä luokat olivat suurilta osin tehty kopioimalla ja liittämällä vanhaa toteutusta ylläpitääkseen taaksepäin yhteensopivuuden. Todellisuudessa näitä luokkia olisi voinut vielä yksinkertaistaa. Sillä ne sisälsivät elementtejä, jotka eivät ole välttämättömiä uudessa toteutuksessa. Tämä tarkoittaa sitä, että koodi sisältää niin sanottua kuollutta koodia, jonka ansiosta koodia on vaikeampi analysoida.

6 POHDINTA

Tässä opinnäytetyössä tutkin ohjelmistokehityksen aikana toteutettavaa ohjelmiston huollettavuutta ja huoltoa itseä. Aihe oli ennestään tuttu, esimerkiksi koodin luettavuuden, ymmärrettävyyden ja modulaarisuuden kannalta. Tutkimusta ja jokapäiväistä ohjelmistokehittäjän työtäni tehdessä aiheesta kuitenkin avautui paljon uusia asioita, joita en ollut edes ajatellut ennen tutkimuksen aloittamista. Isoimmaksi huomion kohteekseni jäi, kuinka liiketoiminnan henkilöstö ja muut tahot vaikuttavat siihen, kuinka huollettavuutta ja huoltoa toteutetaan, jos ollenkaan.

Projektissa yrityksen kannalta on vielä tekemistä. Sen pitää päästä lävitse monta tasoa laadunvarmistusta, ennen kuin se voidaan julkaista tuotantoon. Suurin osa minun osallistumisestani on kuitenkin jo takanani. Koodin huollettavuuden kannalta, testikoodia lukuun ottamatta, sen parantamisessa on onnistuttu annettujen vaatimusten ja tiimin sisäisen palautteen perusteella. Ohjelmiston toiminnallista ikää kasvatettiin teknologiapäivityksin. Ohjelmiston luettavuutta ja ymmärrettävyyttä parannettiin huomattavasti vähentämällä koodin kokonaismäärää, parantamalla modulaarisuutta ja virtaviivaistamalla koodin suoritusta. Näillä muutoksilla uskon myös, että ohjelmiston kokonaisvaltaista suorituskkyä parannettiin valtavasti, koska nyt suorituksessa käännettävä ja ajettava koodi väheni huomattavasti. Sen lisäksi, koska datavirtaus muutettiin asynkroniseksi, käyttäjän näkökulmasta suorituskyyvyn pitäisi parantua suuresti.

Todellisuudessa projektissa olisi paljon, mitä haluaisin vielä parantaa, mutta, koska kehittäjän pää tavoitteena on tuottaa yritykselle rahaa, täytyy projekti lopettaa aikatauluun mennessä. En sano, että ohjelmistossa olisi mitään vikaa, mutta haluaisin vielä parannella ohjelmistoa esimerkiksi parantamalla aikaisemmin mainittua tietoturvaratkaisua. Uskon myös, että olisin voinut tehdä koodista luettavampaa ja ohjelmistosta yleisesti ymmärrettävämpää, käyttämällä kuvaavampia metodi- ja muuttujanimiä sekä yleisesti yksinkertaistamalla luokkien rakennetta. Esimerkiksi kokoonpanoluokkien monikerroksisia rakenteita olisi voinut yksinkertaistaa, koska monikerroksisuus tekee oman kokemukseni perusteella niiden ymmärtämisestä hieman haastavaa. Ohjelmiston on kuitenkin suoriuduttava laadunvarmistuksista ennen tuotantoon julkaisua, joka voi tuottaa lisää huoltotoimenpiteitä ohjelmistolle. Tällä yritetään varmistaa, että ohjelmistossa ei ole suuria virheitä tai suorituskky- tai turvallisuusongelmia.

Tätä opinnäytetyötä tehdessä sain taas vakuuden, että valitsemallani alalla on suurenmoinen määrä opittavaa. Jopa niin paljon, että en voisi edes unelmoida oppivani sitä kaikkea. Voin vain yrittää muistaa kuulleen aiheesta työtäni tehdessä ja oppia tarvittavat asiat tehtävän suoritusta varten. Tämän opinnäytetyön jälkeen pyrin oppimaan lisää, miten liiketoimintahenkilöstö ja muut tahot vaikuttavat kehittäjien jokapäiväiseen työhön ja miten kehittäjät voivat vaikuttaa heidän tekemiin päätöksiin. Tämän lisäksi uskon oppivani työni ohessa lisää ohjelmoinnista, parhaista käytänteistä, puhtaasta ja luettavan koodin tekemisestä sekä yleisesti huollettavasta koodista.

LÄHTEET

Atci, Onur 2023. *Upgrading from Java 8 to Java 17: 15 Reasons to Do It Now*. Hakupäivä 7.12.2023. <https://betterwritecode.medium.com/upgrading-from-java-8-to-java-17-15-reasons-to-do-it-now-162335d5704c>

Bevans, David 2023. *Asynchronous vs. Synchronous Programming: Key Similarities and Differences*. Hakupäivä 7.12.2023. <https://www.mendix.com/blog/asynchronous-vs-synchronous-programming/>

Crouch, Steve -. *Developing maintainable software*. Hakupäivä 30.11.2023. <https://www.software.ac.uk/guide/developing-maintainable-software>

Dhanush, Nehru 2023. *The Myths and Realities of Bug-Free Code*. Hakupäivä 10.01.2024. <https://hackernoon.com/the-myths-and-realities-of-bug-free-code>

Digital Delivery 2021. *How to Design Maintainable and Scalable software*. Hakupäivä 26.11.2023. <https://www.adservio.fr/post/how-to-design-maintainable-and-scalable-software>

FlexBase 2019. *Topic: Software Maintainability Checklist for Software Architects*. Hakupäivä 26.11.2023. <https://medium.com/@flexbasenet/topic-software-maintainability-checklist-for-software-architects-44527ae5f2af>

GeeksForGeeks 2023. *Apache Maven*. Hakupäivä 7.12.2023. <https://www.geeksforgeeks.org/apache-maven/>

GeeksForGeeks 2023. *Introduction to Project Lombok in Java and How to get started?* Hakupäivä 11.01.2024. <https://www.geeksforgeeks.org/introduction-to-project-lombok-in-java-and-how-to-get-started/>

Glushach, Roman 2023. *Reactive Programming: Asynchronous and Event-Driven Architecture by Unlocking the Power of Data Streams*. Hakupäivä 8.12.2023.

<https://romanglushach.medium.com/reactive-programming-asynchronous-and-event-driven-architecture-by-unlocking-the-power-of-data-362bb118e3b>

Hayes, Jim 2014. *Software Maintenance Costs - Part 3*. Hakupäivä 3.12.2023. <https://www.linkedin.com/pulse/20141203004038-26267525-software-maintenance-costs-part-3/>

IBM -. *What are microservices?* Hakupäivä 25.11.2023. <https://www.ibm.com/topics/microservices>

Araujo, Jean, Melo, Carlos, Oliveira, Felipe, Pereira, Paulo & Matos, Rubens 2021. *A Software Maintenance Methodology: An Approach Applied to Software Aging. 2021 IEEE International Systems Conference (SysCon)*. IEEE. Hakupäivä 30.11.2023. <https://ieeexplore.ieee.org/document/9447082>

Leppänen, Väinö 2011. *Software maintainability*. Karelän-ammattikorkeakoulu. Tieto- ja viestintätekniikan tutkinto-ohjelma. Opinnäytetyö. Hakupäivä 30.11.2023. <https://urn.fi/URN:NBN:fi:amk-2011060611123>

Martti, Leino 2016. *Coding Standards in Web Development*. Metropolia ammattikorkeakoulu. Media teknologian tutkinto-ohjelma. Opinnäytetyö. Hakupäivä 30.11.2023. <https://urn.fi/URN:NBN:fi:amk-2016053110930>

Mattias, Nixell 2019. *Increasing Software Availability and Scalability with Microservice Architecture*. Novia ammattikorkeakoulu. Tietotekniikan tutkinto-ohjelma. Opinnäytetyö. Hakupäivä 30.11.2023 osoitteesta <https://urn.fi/URN:NBN:fi:amk-2019061416948>

Bitar, Michel, Seydi, Nassredine, Razafindrabe, Tolo & Ali, Yacine 2023. *What's new in Spring Boot 3?* Hakupäivä 8.12.2023. <https://positivethinking.tech/insights/whats-new-in-spring-boot-3/>

Oracle 2023. *Oracle Java SE Support Roadmap*. Hakupäivä 11.12.2023. <https://www.oracle.com/java/technologies/java-se-support-roadmap.html>

Otta, Maxmilian 2023. *What Is Reactive Programming?* Hakupäivä 8.12.2023. <https://www.baeldung.com/cs/reactive-programming>

Rosene, Frederick, Connolly, J. E., & Bracy, K. M. 1981. Software Maintainability - What It Means and How to Achieve It. *IEEE Transactions on Reliability*, 240 - 245. Hakupäivä 30.11.2023. <https://ieeexplore.ieee.org/abstract/document/5221065/>

Sahoo, Anil, Kumar 2023. *How to use Project Lombok*. Hakupäivä 8.12.2023. <https://medium.com/@anil7017/how-to-use-project-lombok-b2819caa5f30>

Saiful, Islam 2023. JavaScript alternative (TypeScript) and its effectiveness in web development. Tampereen ammattikorkeakoulu. Software Engineering. Bachelor's Thesis. Hakupäivä 30.11.2023. <https://urn.fi/URN:NBN:fi:amk-202305088240>

Sealights a -. *Software maintainability: What it means to build maintainable software*. Hakupäivä 26.11.2023. <https://www.sealights.io/software-quality/software-maintainability-what-it-means-to-build-maintainable-software/>

Sealights b -. Why Technical Debt is Not Our Fault. Or is it? Hakupäivä 10.01.2024. <https://www.sealights.io/sprint-velocity/why-technical-debt-is-not-our-fault-or-is-it/>

Sina, Shamshiri, Jose, Rojas, Juan, Galeotti, Neil, Walkinshaw & Gordon, Fraser 2018. How Do Automatically Generated Unit Tests Influence Software Maintenance. *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE. Hakupäivä 30.11.2023. <https://ieeexplore.ieee.org/document/8367053>

Spring -. *Spring Framework*. Hakupäivä 7.12.2023. <https://spring.io/projects/spring-framework>

Visser, Joost, Rigal, Sylvan, Leek, Rob, Eck, Pascal & Wijnholds, Gijs 2016. *What is maintainability?* Hakupäivä 26.11.2023. <https://www.oreilly.com/content/what-is-maintainability/>

Webb, Phil & Syer, Dave 2013. *Spring Boot - Simplifying Spring for Everyone*. Hakupäivä 11.12.2023. <https://spring.io/blog/2013/08/06/spring-boot-simplifying-spring-for-everyone>