Ohjelmointikäytäntöjen optimointi Angular lomakkeissa Case: LogiPlan-toiminnanohjausjärjestelmä

Jukka Heikkinen

2020 Laurea

Laurea-ammattikorkeakoulu

Ohjelmointikäytäntöjen optimointi Angular lomakkeissa Case: LogiPlan-toiminnanohjausjärjestelmä

Jukka Heikkinen

Tietojenkäsittely

Opinnäytetyö

Syyskuu, 2020

Laurea-ammattikorkeakoulu **Tiivistelmä**

Tietojenkäsittelyn koulutusohjelma

Tradenomi (AMK)

Jukka Heikkinen

Ohjelmointikäytäntöjen optimointi Angular lomakkeissa Case: LogiPlan-toiminnanohjausjärjestelmä

Vuosi 2020 Sivumäärä 63

Tässä opinnäytetyössä tutkittiin asiakasyrityksen hyvien ohjelmointitapojen noudattamista ja niiden mahdollista kehittämistä Angular-sovelluskehyksen reaktiivisien lomakkeiden käyttöön liittyen LogiPlan-toiminnanohjausjärjestelmässä. Tavoitteena oli löytää optimaaliset tavat käyttää reaktiivisia lomakkeita, pitäen samalla huolen, että hyviä ohjelmointitapoja noudatetaan.

Opinnäytetyön teoriaosassa käsiteltiin web-kehittämistä yleisesti, sovelluskehyksiä sekä erityisesti Angularia ja sen avulla toteutettavia lomakkeita. Lisäksi teoriaosassa tarkasteltiin koodikatselmointia sekä puhtaan koodin määrittelyä ja niiden tuomia etuja sovelluskehityksessä.

Opinnäytetyön osana tehdyssä tutkimuksessa saatiin tietoa havainnointi- ja haastattelumenetelmiä käyttäen, jonka lisäksi suoritettiin koodikatselmointi valikoituihin luokkiin. Haastatteluun osallistui järjestelmän kehittäjä, sillä tietoa järjestelmästä ei juurikaan ollut saatavilla yrityksen ulkopuolella. Suoritetun koodikatselmoinnin avulla saatiin tarkkaa tietoa noudatetuista hyvistä ohjelmointikäytännöistä ja koodin nykytilanteesta.

Tutkimuksessa saatujen tulosten perusteella Leanvay Oy:lle tuotettiin kehitysehdotuksia tulevaisuutta varten ja annettiin palautetta hyvien ohjelmointikäytäntöjen noudattamisesta. Tutkimustuloksien perusteella hyvien ohjelmointikäytäntöjen noudattaminen oli yrityksessä kiitettävällä tasolla eivätkä löydetyt parannuskohteet olleet vakavia. Suurin osa parannuskohteista löytyi HTML-koodin puolelta, rakenteellisiin seikkoihin liittyen sekä syötettyjen tietojen validointia koskien. Lisäksi pystyttiin paikallistamaan kohtia, joista voitiin siivota turhaa koodia pois.

Merkittävimpinä kehitysehdotuksina esiin nousivat HTML-rakenteen yhtenäistäminen ja validointien pitäminen yksinomaan TypeScript-koodissa HTML-koodin sijaan. Myös turhan koodin siivoamista automaattisen muotoilutyökalun avulla suositeltiin. Vaikka löydetyt parannuskohteet eivät olleet tutkimuksessa vakavia, isoissa projekteissa pienikin koodin yhtenäistäminen ja siivoaminen on hyödyllistä. Tämä helpottaa ylläpitoa, jatkokehitystä sekä virheiden löytämistä.

Asiasanat: hyvät ohjelmointikäytännöt, web-kehittäminen, Angular, reaktiiviset lomakkeet

Laurea University of Applied Sciences **Abstract**

Degree Programme in Business Information Technology

Bachelor's Thesis

Jukka Heikkinen

Optimization of Programming Practices in Angular Forms Case: LogiPlan-enterprise Resource Planning System

Year 2020 Pages 63

This Bachelor's thesis focused on investigating the usage of best programming practices and the possibility to develop them further in the program code used in Angular framework's reactive forms within LogiPlan enterprise resource planning system. The objective was to find optimal ways to use reactive forms, whilst ensuring that best programming practices are followed.

The theoretical part of the thesis deals with web-development in general, frameworks and especially Angular and forms that are created with it. Also code review, the definition of clean code and the benefits that they provide were examined.

In the empirical study that was conducted as a part of this thesis, information was gathered with the usage of observation and interview methods. A code review was also conducted in select classes within the program code. An interview was held with a developer of the system, because information about about the studied system was not widely available outside the company. With the code review, exact information about the usage of best programming practices and the current state of the code, was gained.

With the results yielded from the study, suggestions for the future were made to Leanvay LTD and feedback was given concerning the usage of best programming practices. Based on the results the usage of best programming practices was on a commendable level within the company and the found subjects that could be improved, were not critical. Most of the subjects that could be improved were found in the HTML code, concerning structure and input validations. Some parts with excess code that could be cleaned, were also located.

The most noteworthy suggestions were the standardization of the HTML structure and keeping the validations solely in the TypeScript code instead of the HTML code. It was also recommended to clean excess code with the help of an automatic formatting tool. Even though the flaws found in the study were not serious, even a small amount of unification and cleaning of code is beneficial. It makes maintenance, development and the finding of mistakes easier.

Keywords: best programming practices, web-development, Angular, reactive forms

Sisällys

1 Johdanto

Tässä tutkimuksellisessa kehittämistyössä tutkitaan asiakasyrityksen Angular-sovelluskehyksen reaktiivisiin lomakkeisiin liittyvän ohjelmakoodin nykytilaa ja pyritään löytämään mahdollisuuksia ohjelmointikäytäntöjen kehittämiseen reaktiivisten lomakkeiden osalta sekä tarkastamaan ohjelmointikäytäntöjen nykytila. Tutkimus- ja tiedonkeruumenetelmiin syvennytään opinnäytetyön teoriaosuudessa hyödyntäen tietoja osana ohjelmakoodin nykytilan tarkistamista. Tutkimusosassa suoritetaan katselmointi tarkoituksenmukaisesti valikoiduille luokille ohjelmakoodissa ja pyritään paikallistamaan mahdollisia parannuskohteita ohjelmointikäytännöissä.

Tutkimuksessa käytetään tiedon keräämiseen tutkijan omaa havainnointia reaktiivisien lomakkeiden kohdalta, koodikatselmointia ja kokeneen järjestelmäkehittäjän haastattelua. Lisäksi tietoa tutkimusta varten pyritään saamaan epävirallisien keskustelujen kautta kokeneemmilta ohjelmoijilta tehtäessä tutkimustyötä yrityksen tiloissa. Suoritetun koodikatselmoinnin sekä havainnoinnin tuottamien tulosten perusteella tavoitteena on antaa asiakasyritykselle kehitysehdotuksia ohjelmointikäytäntöihin liittyen.

Toimeksiantajana on vuonna 2019 perustettu Leanvay Oy, joka ylläpitää ja kehittää tutkittavaa LogiPlan-toiminnanohjausjärjestelmää. Tämä tutkimus keskittyy LogiPlan-toiminnanohjausjärjestelmän ohjelmakoodiin Angular-sovelluskehyksen reaktiivisien lomakkeiden käyttöön liittyen ja pyrkii löytämään parhaat mahdolliset ohjelmointikäytännöt niiden käyttämistä varten.

Hyvien ohjelmointikäytäntöjen noudattaminen parantaa ohjelmakoodin luettavuutta, ylläpidettävyyttä ja jatkokehittämistä. Etenkin isoissa projekteissa on tärkeää, että hyviä ohjelmointikäytäntöjä on noudatettu ja ohjelmakoodi on yhdenmukaista. Näin toimittaessa esimerkiksi virheiden löytäminen ja uusien työntekijöiden ottaminen projektin kehittämiseen on helpompaa.

2 Työn lähtökohta ja tarkoitus

Tutkimusaihe saatiin toimeksiantona Leanvay Oy:ltä ja tutkimus toteutettiin projektimuotoisena kehittämistyönä työharjoittelun aikana ja sen päätyttyä, aikavälillä 1.4.2020 – 18.6.2020. Aiheen tutkiminen tapahtui haastattelemalla järjestelmän kehittäjää sekä havainnoinnin osalta tekemällä varsinaista LogiPlan-toiminnanohjausjärjestelmän päivitystyötä työharjoittelun aikana Leanvay Oy:ssä, jossa suoritettiin LogiPlan-toiminnanohjausjärjestelmän

päivittämistä vanhasta, Wicket-ohjelmistokehystä käyttävästä käyttöliittymästä uuteen, Angular-ohjelmistokehystä käyttävään käyttöliittymään. Tarkempaa tietoa koodin nykytilasta saatiin suorittamalla koodikatselmointi tutkimukseen valikoituihin luokkiin.

Tutkimuksellisessa kehittämistyössä syvennyttiin toimeksiannon mukaisesti hyvien ohjelmointikäytäntöjen nykytilan tutkimiseen sekä mahdolliseen parantamiseen LogiPlan-toiminnanohjausjärjestelmässä käytössä olevien Angularin reaktiivisen lomakkeiden osalta. Samalla tutkittiin, onko kyseinen koodi mahdollisimman siistiä ja helposti luettavaa. Järjestelmän kehitystyössä kertyneiden omien kokemusten, koodikatselmoinnin ja haastattelussa ilmenneiden asioiden avulla pyrittiin löytämään mahdollisia parannuskohtia käytössä olleisiin ohjelmointikäytäntöihin liittyen. Ilmi tulleisiin parannuskohtiin pyrittiin tutkimuksen avulla löytämään vaihtoehtoisia ratkaisuja ja kehittämään ohjelmointikäytäntöjä lomakkeiden osalta entistä paremmiksi. Tutkimuksessa pyrittiin myös varmistamaan, että hyvät ohjelmointitavat on otettu huomioon ja niitä on noudatettu yhtenäisesti.

Tuloksia tutkimusta varten saatiin koodikatselmoinnilla, havainnoinnilla, haastattelulla ja hakemalla tietoa omatoimisesti siitä, minkä kaltaisia erilaisia toteutustapoja voidaan soveltaa toimeksiantajan projektissa Angularin reaktiivisia lomakkeita käytettäessä. Täten pyrittiin löytämään parhaalla mahdollisella tavalla toimeksiantajan tarpeita palvelevia ohjelmointikäytäntöjä.

Tarkoituksena opinnäytetyössä oli paikallistaa mahdollisia parannuskohteita liittyen reaktiivisien lomakkeiden käyttöön, jotka ovat olennainen osa Angular-sovelluskehyksen avulla luotavia järjestelmiä. Tavoitteena oli tuotettavan koodin siisteyden optimointi, jotta jatkossa sen kehittäminen ja ylläpitäminen olisi mahdollisimman vaivatonta. Lisäksi tarkoituksena oli tuottaa toimeksiantajalle tarkempaa tietoa aiheeseen liittyen.

2.1 LogiPlan-toiminnanohjausjärjestelmän kuvaus

LogiPlan-toiminnanohjausjärjestelmä on kuljetusliikkeiden käyttötarpeisiin suunniteltu, ammattikäyttöön tarkoitettu selainpohjainen järjestelmä. Järjestelmä pitää sisällään runsaasti erilaisia toiminnallisuuksia, kuten työmaatilauksien hallinnointi, kuljetussuunnittelu, laskuttaminen sekä tilitys ja varastojen hallinta. Lisäksi tarjolla on kattavasti erilaisia raportteja ja erilaisia hinnastoja. LogiPlan kattaa koko kuljetusprosessin aina asiakkaan tekemästä tilauksesta, joka välitetään ajojärjestelijälle, joka vuorostaan välittää työn aina kuljettajalle asti. Kuljettaja puolestaan vastaanottaa tarjotun työn LogiPlan-mobiilisovelluksella ja sen tehtyään kuittaa työn olevan valmis. Järjestelmässä olevien tolppajonojen avulla työt saadaan jaettua kätevästi, kuskit voivat virtuaalisesti jonottaa vapautuvia keikkoja, jotka ajojärjestelijä jakaa seuraavaksi vuorossa olevalle kuljettajalle. LogiPlan on myös integroitavissa taloushallinnon ohjelmistoihin sekä vaakoihin, joita käytetään lastauspaikoilla kuormien

punnitsemiseen. Järjestelmän avulla voidaan lisäksi seurata ajoneuvojen sijaintia ja työntekijöiden työaikatietoja. (Leanvay, 2020).

LogiPlan on luotu 2011 ja sitä käyttää tällä hetkellä 14 asiakasyritystä ja järjestelmää käyttäviä ihmisiä on arviolta noin 840. LogiPlanin kehityksessä käytetty sovelluskehys on ollut aluksi Apache Wicket, mutta sovelluskehys päätettiin vaihtaa Angulariin. Syynä sovelluskehyksen vaihtoon oli Wicketin vanhanaikaisuus ja siinä ilmenneet huonot puolet, kuten:

- Huono JavaScript-tuki
- Suuri muistin kulutus
- Hitaus
- Front end kehittäminen suurelta osin back endin puolella (HTML vähäistä, iso osa koodista Javalla)
- UI-kirjastojen vähäinen määrä

Harkittaessa uutta sovelluskehystä Wicketin tilalle AngularJS (Angularin ensimmäinen versio) oli helpohko valinta, sillä kyseisellä sovelluskehyksellä ei sillä hetkellä ollut paljon kilpailijoita, joiden sovelluskehys olisi ollut yhtä pitkälle kehittynyt. (Feng, 2020). Lisätietoja Wicketistä löytyy kappaleesta 3.1.1.

2.2 Tutkimuskysymykset ja -tavoitteet

Tutkimuksellisessa kehittämistyössä pyritään selvittämään, ovatko Leanvay Oy:n käyttämät ohjelmointikäytännöt Angularin reaktiivisiin lomakkeisiin liittyen optimaalisella tasolla ja onko hyviä ohjelmointitapoja noudatettu. Lisäksi pyritään löytämään mahdollisesti vaihtoehtoisia ohjelmointitapoja reaktiivisia lomakkeita luotaessa käytettävän koodin luettavuuden ja ylläpidettävyyden suhteen. Tavoitteena on myös löytää keinoja, joiden avulla voidaan pitää koodi mahdollisesti puhtaampana. Mikäli käytössä ennestään ollut koodi on toimivaa, halutaan selvittää voiko sitä siistiä entisestään. Samalla tutkimuksellinen kehittämistyö pyrkii myös selvittämään, voiko koodin määrää samassa yhteydessä vähentää, joka entisestään helpottaa järjestelmän ylläpitämistä sekä parantaa sen luettavuutta.

Tutkimuskysymyksiä joihin opinnäytetyössä pyrittiin vastaamaan:

- Voidaanko yrityksen ohjelmointikäytäntöjä kehittää Angularin reaktiivisien lomakkeiden osalta?
- Onko koodi mahdollisimman puhdasta ja helposti luettavaa?
- Onko hyvät ohjelmointikäytännöt huomioitu?

Tutkimustyössä pyritään löytämään vaihtoehtoisia tapoja toimeksiantajan nykyisin käytössä olevien ohjelmointikäytännöille reaktiivisien lomakkeiden osalta, vertailemalla näitä

keskenään, testaamalla mahdollisten vaihtoehtojen toimintaa ja määrittämällä mikä ohjelmointitapa on paras toimeksiantajalle.

Kehittämismenetelmänä on ensisijaisesti osallistuva havainnointi varsinaista työtä tehtäessä, joka sisältää runsaasti reaktiivisien lomakkeiden toteuttamista päivitettäessä LogiPlan-järjestelmän käyttöliittymää vanhasta Wicket-ohjelmistokehyksestä uuteen, Angular-ohjelmistokehystä käyttävään käyttöliittymään. Varsinaiset tarkat parannuskohteet pyritään löytämään suorittamalla koodikatselmointi valikoituihin luokkiin.

Lisäksi hyödynnetään kokeneempien kehittäjien ammattitaitoa muodollisemman puolistrukturoidun teemahaastattelun lisäksi epämuodollisten haastattelujen avulla, keskustelemalla tutkimusaiheesta tai pyytäen apua ongelmatilanteissa, joita ei itse saada ratkaistua. On mahdollista, että löytyy mahdollisesti erilaisia ohjeistuksia ja ”niksejä”, joiden avulla koodia voidaan puhdistaa nykyisestä. Toisaalta on myös mahdollista, että tällaisia ei löydy ja tällöin käytössä olevat ohjelmointikäytännöt reaktiivisien lomakkeiden osalta ovat parhaita mahdollisia.

Tavoitteena opinnäytetyöllä on saada tuotettua konkreettisia kehitysehdotuksia Leanvay Oy:lle reaktiivisien lomakkeiden käyttöön liittyen, siten että jatkossa yritys voisi käyttää reaktiivisia lomakkeita tehokkaammin ja niiden ylläpidettävyys sekä jatkokehittäminen olisi helpompaa.

2.3 Aiheen rajaus

Aihealue opinnäytetyössä on rajattu koskemaan Angular-ohjelmistokehyksen reaktiivisia lomakkeita, joita LogiPlan-toiminnanohjausjärjestelmässä käytetään runsaasti. Reaktiivisien lomakkeiden koodin osalta keskitytään selvittämään parhaita tapoja niiden käyttämiseen, ottaen huomioon hyvät ohjelmointitavat, pyrkien mahdollisimman siistiin, lyhyeen ja helppolukuiseen koodiin. Reaktiivisia lomakkeita koskevan ohjelmakoodin osalta aihealue rajataan koskemaan front end -koodia.

Tutkimuksessa syvennytään myös koodikatselmoinnin avulla valikoituihin luokkiin, jotka liittyivät olennaisesti reaktiivisien lomakkeiden käyttöön. Katselmoinnissa pyritään löytämään mahdollisia rakenteellisia ja kielellisiä parannuskohteita sekä pyrittiin minimoimaan tarvittavan koodin määrä hyvien ohjelmointitapojen mukaisesti. Tässä tutkimuksessa ei keskitytä muuhun LogiPlan-toiminnanohjausjärjestelmään liittyvään front end -koodiin, eikä vanhan käyttöliittymän Wicket-koodiin.

2.4 Keskeiset käsitteet

Web-kehittäminen tarkoittaa nettisivun- tai sivujen luomiseen liittyvää työskentelyä.

Front end tarkoittaa web-kehittämisessä visuaalista, käyttäjälle näkyvää puolta.

Back end tarkoittaa web-kehittämisessä käyttäjälle näkymätöntä serveripuolta.

HTML on verkkosivujen sisällön kuvaamiseen käytetty ohjelmointikieli ja lyhenne sanoista Hypertext Markup Language.

JavaScript on kevyehkö ohjelmointikieli, jota käytetään pääasiassa verkkoapplikaatioiden luomiseen.

TypeScript on JavaScriptin päälle rakennettu ohjelmointikieli, jota käytetään web-kehityksessä.

Sovelluskehys tarkoittaa ohjelmistokehystä, joka sisältää valmiiksi hyödyllisiä toiminnallisuuksia, joiden avulla saadaan nopeutettua ohjelmistokehitystä.

Angular tarkoittaa TypeScript-pohjaista sovelluskehystä, jota käytetään sovelluskehityksessä.

Lomake tarkoittaa lomaketta, jota käytetään tiedon vaihtamiseen palvelimen ja käyttäjän välillä.

Reaktiivinen lomake tarkoittaa Angularissa luotua lomaketta, jonka pääasiallinen toteutus on tehty TypeScript-koodissa HTML:n sijaan

Clean code tarkoittaa ohjelmoinnissa koodin pitämistä siistinä ja mahdollisimman helppona ymmärtää.

Riippuvuusinjektio tarkoittaa prosessia, jossa tuodaan ulkoinen riippuvuus sovelluskomponenttiin.

POJO tarkoittaa tavallista Java-oliota (Plain Old Java Object).

UI on lyhenne sanoista User Interface ja tarkoittaa käyttöliittymää.

3 Web-kehittäminen

Web-kehittäminen (web development) käsitteenä pitää sisällään kaiken työskentelyn, mikä liittyy verkkosivujen luomiseen. Web-kehittämistä tehdään sekä front end-ympäristössä UI-puolen kehittämiseen, että back end -ympäristössä serveripuolen kehittämiseen.

Front end pitää sisällään kaiken, jonka käyttäjä näkee ja käyttää selaimessaan. Kehittäjän tehtävänä frontend puolella on tuottaa koodi, joka kääntää suunnitellun sivun tai sivuston toimivaksi verkkosivuksi. Suunnitelma voi olla kehittäjän itsensä laatima, mutta usein se on

erillisen suunnittelijan luoma, esimerkiksi kuva verkkosivusta, joka kehittäjän tulee kääntää koodiksi. (McFedries, 12).

HTML (Hypertext Markup Language) on kieli, jota käytetään verkkosivujen luomiseen ja se on verkkosivujen rakennuksessa käytetty standardikieli. Hyperteksti tarkoittaa tekstiä, joka näytetään tietoteknisen laitteen ruudulla ja sen sekaan voidaan sisällyttää linkkejä toisiin hyperteksti-dokumentteihin, joihin käyttäjä voi navigoida. Hypertekstidokumentissa voidaan kuvata erilaisia olioita kuten kuvia, lomakkeita, listoja ja taulukoita. (Coremans, 1).

CSS (Cascading Style Sheets) on kieli, jolla voidaan muotoilla dokumentteja, jotka on luotu merkkauskielellä, kuten esimerkiksi HTML:llä. CSS ei pidä sisällään varsinaisen dokumentin sisältöä, vaan vaikuttaa ainoastaan sen esitystapaan. CSS:n avulla voidaan määrittää HTML-elementtien ominaisuuksia, kuten esimerkiksi väriä, fonttia, taustaa ja niin edelleen. (Coremans, 77).

Back end koostuu koodista sekä algoritmeista, jotka ajetaan piilossa käyttäjältä palvelinpuolella. Tämä puoli koostuu kolmesta osasta: Palvelin, tietokanta ja palvelinpuolen sovellukset. Mikäli verrattaisiin verkkosivun luomista talon rakentamiseen, back end pitäisi sisällään kaikki tarpeelliset, piilossa olevat asiat, joita talossa tarvitaan, kuten perustukset, talon kehikko, putkisto ja sähköjohdot. Kaikki nämä ovat toimivassa kokonaisuudessa tarvittavia asioita, vaikka käyttäjä ei suoranaisesti niitä käytäkään, aivan kuten verkkosivua käytettäessä, käyttäjä ei suoraan manipuloi esimerkiksi tietokantaa vaan käyttää front endin tarjoamia työkaluja. Vastaavasti taas front end- kehittäjät ovat vastuussa käytettävistä elementeistä, kuten vesihanoista ja valokatkaisimista, joita voisi verrata verkkosivulla esimerkiksi nappeihin ja tekstinsyöttökenttiin. (Leavitt, 23). Suosittuja back end kieliä ovat esimerkiksi Java, Python ja PHP.

Web-kehitystä tehdään usein tiimeissä asiakkaan toimeksiannon mukaisesti. Kommunikointitaidot ovat täten tärkeitä web-kehittäjälle, sillä varsinaisen koodin kirjoittamisen lisäksi suuri osa työstä pitää sisällään ihmisten kanssa keskustelemista. Oli sitten kyseessä asiakas, sisällöntuottaja tai suunnittelija, web-kehittäjän täytyy pystyä kommunikoimaan tehokkaasti siten, että puhuttava kieli ei ole liian teknistä, jotta kaikki voivat ymmärtää mistä puhutaan. Erityisen tärkeää tämä on asiakkaan kanssa, jotta saadaan aikaan asiakkaan toivoma lopputulos. (Shofner 2018, 16).

3.1 Sovelluskehys

Sovelluskehyksien (framework) avulla voidaan nopeuttaa sovelluskehitystä huomattavasti, sillä ne tarjoavat valmiina tarpeeseen sopivia toiminnallisuuksia, joiden päälle varsinainen sovellus, sivusto ja niin edelleen, voidaan rakentaa. Näin ollen, sovelluskehittäjien ei tarvitse

luoda jokaista, geneeristä ominaisuutta alusta alkaen, joka säästää huomattavasti kehittäjien aikaa ja vaivaa.

Sovelluskehyksien tarkoitus on parantaa tehokkuutta uuden softan rakentamisessa. Kehittäjien tuottavuus sekä itse luotavan softan laatu, luotettavuus ja jämäkkyys voivat parantua sovelluskehystä käytettäessä. Tuottavuuden parantuminen perustuu siihen, että kehittäjät voivat keskittyä ohjelman uniikkeihin kohtiin, eikä heidän tarvitse käyttää aikaa ohjelman infrastruktuurin rakentamiseen. (Baker, 2009).

Sovelluskehyksien lisäksi on olemassa valmiita kirjastoja, ja vaikka kehittäjät käyttävät näitä kahta termiä monesti keskenään vaihtokelpoisina, sovelluskehykset ja kirjastot eroavat toisistaan. Yhteistä molemmille on se, että ne ovat toisen kirjoittamaa, uudelleen käytettävää ohjelmakoodia, jonka tarkoituksena on auttaa ratkaisemaan tavanomaisia ongelmia. Oleellisin ero on hallinnan suunta. Kirjastoa käytettäessä kehittäjä kutsuu koodia kirjastosta, mutta sovelluskehystä käytettäessä sovelluskehys kutsuu kehittäjän luomaa koodia. (Wozniewicz, 2019).

3.1.1 Wicket sovelluskehys

Apache Wicket on avoimeen lähdekoodin pohjautuva back end -puolen web-applikaatio ohjelmistokehys, joka käyttää ohjelmointikielenään Javaa. Wicketin avulla voi luoda ylläpidettäviä, skaalautuvia ja turvallisia web-applikaatioita pelkästään Java- ja HTML-ohjelmointikieliä käyttäen. (Pat research, 2020).

Poiketen useimmista muista Javaa käyttävistä web-ohjelmistokehyksistä, Wicket ei vaadi erityissyntaksin käyttämistä HTML:ssä. Wicketin etuihin kuuluvat:

- Kaikki koodi kirjoitetaan Javalla
- Ei tarvetta XML-konfiguraatiotiedostoille
- Ohjelmointi on POJO-keskeistä
- Ei ’takaisin’-nappi ongelmia selaimessa (odottamattomat tulokset klikatessa)
- Helppo luoda kirjamerkittäviä sivuja
- Mahtava ongelmadiagnosointi koottaessa ja ajettaessa
- Komponenttien helppo testattavuus

Wicket pohjautuu uudelleen käytettäviin komponentteihin, joita tulee voida myös korvata, laajentaa ja kapseloida. Komponentilla tarkoitetaan oliota, joka on tekemisissä toisien komponenttien kanssa ja sisältää erilaisia toiminnallisuuksia. (Ceregatti & Pinto 2013, 13-14).

Wicketin fokuksena on uudelleenkäytettävän koodin helppo kirjoittaminen. Koodi saadaan pysymään hyvin järjestyksessä sillä logiikka ja merkkauskieli ovat erotettuja toisistaan, jolloin

ohjelmakoodi on pelkkää Javaa ja merkkauskieli pelkkää HTML:ää. Tämän ansiosta esimerkiksi front end-puolen ohjelmoijien on helppo avustaa projekteissa ilman pelkoa koodin hajottamisesta. (Thomerson, 2012).

3.1.2 Angular sovelluskehys

Angular on Googlen ylläpitämä, HTML:ää ja TypeScriptiä käyttävä sovelluskehys yksisivuisten applikaatioiden (Single Page Application) rakentamista varten. Angularin avulla luodut applikaatiot rakentuvat moduuleista ja jokaisella applikaatiolla on juurimoduuli, joka on nimetty AppModuleksi. Koodin järjestäminen tällä tavoin yksittäisiksi moduuleiksi auttaa monimutkaisten applikaatioiden kehittämisessä ja suunnittelussa. (Google a, 2020).

Angular sai alkunsa vuonna 2009 Googlella työskentelevän Miško Heveryn toimesta, joka noihin aikoihin työsti projektia Googlelle kahden muun kehittäjän kanssa. Projektiin kirjoitettiin kuuden kuukauden aikana 17 000 riviä koodia ja se alkoi kärsiä testausvaikeuksista ja turvonneesta koodimäärästä (code bloat). Hevery pyysi lupaa esimieheltään kirjoittaa applikaatio uusiksi käyttäen Adam Abronsin kanssa sivuprojektina luomaansa end-to-end web-kehitystyökalua, joka sisälsi online JSON varastointipalvelun ja asiakaspuolen kirjaston web-applikaatioiden rakentamista varten. Hevery kirjoitti työstämänsä Google-projektin yksinään kolmen viikon aikana ja uusi versio sisälsi 1500 riviä koodia entisen, 17000 riviä koodia sisältäneen version sijaan. Google kiinnostui uudesta sovelluskehyksestä, joka nimettiin AngularJS:ksi. (Ryan, 2019).

AngularJS:n ensimmäinen vakaa versio 0.9.0 julkaistiin lokakuussa 2010 MIT-lisenssin alla GitHubissa ja versio 1.0.0 julkaistiin kesäkuussa 2012, jolloin sovelluskehys oli saavuttanut jo huomattavaa suosiota maailmanlaajuisesti web-kehitysyhteisöissä. Suurimpina etuina AngularJS tarjosi riippuvuusinjektiot (dependency injection), direktiivit, joilla tarkoitetaan itseluotujen ja uudelleenkäytettävien HTML:n kaltaisten elementtien ja attribuuttien spesifiointia, kaksisuuntaisen datasidonnan, jonka avulla voidaan päivittää malli ja näkymä samanaikaisesti, yksisivuisen lähestymistavan, joka poistaa tarpeen sivun uudelleenlataukselle ja välimuistiystävällisyyden, koska serveripuolen ei tarvitse luoda käyttöliittymäosia. (Ryan, 2019).

AngularJS kirjoitettiin kokonaan uusiksi ja julkaistiin syyskuussa 2016 nimellä Angular 2. Se oli samasta nimestä huolimatta täysin uusi sovelluskehys, sillä koodin syntaksi oli muuttunut edelliseen versioon verrattuna ja lisäksi uusi versio oli komponenttipohjaisempi ja modulaarisempi. Lisäksi riippuvuusinjektio oli uusittu ja paranneltu sekä uusia ohjelmointikaavoja tuotiin mukaan. Tärkeimpiä muutoksia Angular 2:ssa olivat:

- Semanttinen versiointi
- TypeScript
- Serveripuolen renderöinti

- Angular mobiilityökalut
- Komentorivikäyttöliittymä
- Komponentit

Angular 4 julkaistiin Googlen toimesta maaliskuussa 2017 (Angular 3 jätettiin julkaisematta, jotta saatiin yhtenäistettyä kaikki pääversiot monista komponenteista, joita oli kehitetty toisista erillään aiemmin). Angular 4 tarjosi uuden ja parannetun reititysjärjestelmän (routing system) ja tuen TypeScriptin versiolle 2.1 ja siitä ylöspäin. Verraten AngularJS:ään ja Angular 2:een, joissa kompilointi tapahtui applikaation käynnistyessä, Angular 4:ssä kompilointi tapahtuu rakennusvaiheessa, joka nopeuttaa applikaatiota huomattavasti. Myös uusi lomakevalidaattori sisältyi Angular 4:ään, jonka avulla voidaan tarkastaa valideja sähköpostiosoitteita.

Marraskuussa 2017 julkistettu Angular 5 tarjosi tuen TypeScript-versiolle 2.3 sekä monia suorituskykyä ja vakautta parantavia parannuksia. Lisäksi uusina ominaisuuksina tulivat ominaisuus, joka sallii applikaation tilan siirtämisen serverin ja asiakkaan välillä (State Transfer rajapinta), parempi http-elinkaaren hallinta uusien router eventien avulla ja uusi http-asiakasrajapinta.

Uusi versio, Angular 6, näki päivänvalon huhtikuussa 2018 ja siinä keskityttiin lähinnä parantamaan Angularin kokonaisvaltaista johdonmukaisuutta eikä niinkään lisäämään uusia ominaisuuksia. Suurempi päivitys tuli Angular 7:n muodossa lokakuussa 2018 ja sen tärkeimmät uudistukset olivat:

- Applikaation helpompi päivittäminen vanhemmista Angular-versioista käyttämään uusinta versiota.
- Parannetut komentorivikomennot
- Uusia käyttöliittymäelementtejä, esimerkiksi komponentti (Virtual scrolling), joka näyttää vieritettävistä listanäkymistä vain näkyvän osan, joka nopeuttaa huomattavasti isoja listoja käyttäviä sivuja
- Parannettu yhteensopivuus kolmannen osapuolen yhteisöprojektien kanssa
- Päivitetyt riippuvuudet

Angular 7:n mukana tuli myös Angular Language Service, joka näyttää koodin virheet reaaliaikaisesti, ehdottaa automaattisia täydennyksiä, tarjoaa vihjeitä ja auttaa navigoinnissa.

Tärkeimpänä ominaisuutena Angular 8:ssa, joka julkaistiin toukokuussa 2019, oli uuden Ivy-nimisen kääntäjän hyödyntäminen. Muita huomionarvoisia parannuksia ja ominaisuuksia olivat esimerkiksi tuki Bazelille, joka on Googlen kehittämä työkalu softan rakentamista ja testaamista varten. Myös reitittämistä, rekisteröintistrategiaa ja työtilan konfigurointia on parannettu Angular 8:ssa. (Ryan, 2019).

Tämän tutkimuksen kirjoitushetkellä uusin versio Angularista on Angular 9 joka julkaistiin helmikuussa 2020 ja se käyttää oletuksena Angular 8:sta tuttua Ivy-kääntäjää. Ivy:ssä on useita etuja, kuten:

- Pienemmät bundle-koot (suomenna)
- Nopeampi testaus
- Parempi virheenetsintä (debuggaus)
- Parannettu CSS luokka- ja tyylisidonta
- Parannettu tyyppitarkastus
- Parannetut koontivirheet
- Parannetut koontiajat
- Parannettu kansainvälistäminen

Ivyn lisäksi Angular 9:n tärkeimpiin parannuksiin kuuluu esimerkiksi uudet komponentit, joiden avulla voidaan sisällyttää Youtuben ja Google Mapsin ominaisuuksia applikaatioon. (Fluin, 2020).

3.2 Lomakkeet

Lomakkeet tarjoavat rajapinnan datan keräämiseen verkkosivun tai applikaation käyttäjältä. Kerättävän datan kirjo on laaja, esimerkiksi kyse voi olla hausta, kirjautumisruudusta, sukupuolesta, salasanasta ja niin edelleen. Myös datan keräysmetodeja on monia, kuten vaikkapa tekstinsyöttö tai valintaruutu. Ilman lomakkeita internetin käyttö olisikin melko yksipuolista, sillä käyttäjä ei voisi tehdä hakuja, keskustella tai vaikkapa ostaa tuotteita. (Gupta 2013 ,24).

3.2.1 Lomakkeet Angularissa

Angulariin ominaisuuksiin kuuluu sisäänrakennettuna ominaisuutena Angular lomakkeet, joten Angularia käytettäessä web-kehitykseen, ei kolmannen osapuolen paketeille lomakkeisiin liittyen ole tarvetta. Angular tarjoaa kaksi lähestymistapaa lomakkeiden tekemiseen:

- Template Driven Lomakkeet
- Reaktiiviset Lomakkeet

Yllä mainittujen lähestymistapojen avulla voidaan ilmoittaa käyttäjälle, mikäli jotain meni pieleen lomaketta täytettäessä ja ennen kaikkea syy siihen. (Sanyal 2019.)

Template Driven Lomakkeet ja Reaktiiviset Lomakkeet jakavat seuraavat yhteiset rakennuspalikat:

- FormControl (Seuraa yksittäisen form control:n arvoa ja validointia)
- FormGroup (Seuraa form control kokoelman samoja arvoja ja statusta)

- FormArray (Seuraa form control taulukon samoja arvoja ja statusta)
- ControlValueAccessor (Luo sillan Angularin FormControl instanssien ja natiivien DOM-elementtien välille)

Template-driven lomakkeet ovat sopivampia yksinkertaisia lomakkeita varten ja niitä on helppo käyttää, mutta ne eivät ole yhtä hyvin skaalautuvia kuin reaktiiviset lomakkeet. Reaktiiviset lomakkeet ovat skaalautuvuuden lisäksi uudelleenkäytettäviä ja helpommin testattavia. Jos applikaatio käyttää paljon lomakkeita, reaktiivisien lomakkeiden käyttö on suotavampaa. (Google a 2020.)

3.2.2 Angularin Template Driven Lomakkeet

Template driven lomakkeet muistuttavat pitkälti perinteisiä HTML-lomakkeita. Logiikka, oikeellisuuden varmistukset ynnä muut kirjoitetaan tässä lähestymistavassa HTML-koodiin, joten myös mahdolliset virheet käsitellään HTML-koodin puolella. Angular käyttää FormControl-luokkaa seuratakseen yksittäisen lomakkeen arvoa ja validointia ja jokaista lomakkeella olevaa kenttää kohti tulee tehdä instanssi FormControl-luokasta. (Sanyal 2019.)

3.2.3 Angularin Reaktiiviset Lomakkeet

Reaktiiviset lomakkeet eroavat template driven lomakkeista siten, että HTML:ssä lomakkeen rakenteen luonnin sijaan se tapahtuu varsinaisessa koodissa TypeScript-tiedostossa. Luotua mallia käytetään tämän jälkeen HTML:n puolella. (Shah 2019.)

Monimutkaisia, joustavia lomakkeita luotaessa on syytä käyttää reaktiivisia lomakkeita, jotka luodaan Component-luokassa HTML:n sijaan. Näin toimittaessa kehittäjillä on paremmin hallittavissa lomakkeen määritteleminen ja syntyy mahdollisuus luoda monimutkaisia form groupeja sekä määrittää näille control-luokkia, joiden avulla voidaan hallita dataa sekä validointia. Käytettäessä reaktiivisia lomakkeita, on saatavilla kolme tärkeää rakennuspalikkaa: FormControl, FormGroup ja FormArray. Edellä mainituilla rakennuspalikoilla on yhteisiä metodeja, joita käytetään form-elementtien tilan tarkasteluun, arvojen saantiin ja validointiin. Useimmin käytettyjä metodeja ovat:

- value: arvo
- valid: lomake on validi
- invalid: lomake ei ole validi
- pristine: true: käyttäjä ei ole käsitellyt kyseistä kontrollia
- dirty: käyttäjä on käsitellyt kyseistä kontrollia
- touched: true: käyttäjä on triggeröinyt blur eventin kontrollissa
- untouched: käyttäjä ei ole triggeröinyt blur eventiä kontrollissa
- errors: taulukko, joka sisältää kaikki kontrollin validointivirheet

- setValue: asettaa kontrollille arvon
- patchValue: päivittää kontrollin arvon
- hasError: boolean-arvo virhetilan tarkistamista varten

Lisäksi on olemassa kaksi observable-luokkaa: valueChanges ja statusChanges, joiden avulla arvot saadaan jatkuvasti. Rakennuspalikoita voidaan käyttää lomakkeiden määrittelyyn FormBuilderilla. (Mohammed, 84-85).

3.3 Ohjelmakoodin laatu

Ohjelmakoodin laadulla (clean code) tarkoittaa mahdollisimman helposti luettavaa, seurattavaa ja ylläpidettävää koodia. Etenkin isoissa projekteissa, joissa työskentelee monia ohjelmoijia, on tärkeää, että tuotettavaa koodi on mahdollisimman helppolukuista ja yhtenäistä, jotta sitä voidaan mahdollisimman tehokkaasti ylläpitää ja kehittää edelleen.

Ohjelmakoodin laadun tärkeys piilee siinä, että miltei koskaan koodia ei kirjoiteta vain kerran ja sitten jätetä unholaan. Hyvin usein jonkun tulee työskennellä luodun koodin parissa ja tällöin on tärkeää, että koodi on ymmärrettävissä. Täten on tärkeää ymmärtää, että kirjoitettavaa koodia ei luoda vain tietokoneen ymmärrettäväksi, vaan myös muiden ihmisten. Puhdasta koodia kirjoittamalla auttaa paitsi työtovereitaan, myös itseään, sillä helpotat jatkokehitystä tekemällä uusien ominaisuuksien lisäämisen keston arvioimisesta ja bugien korjaamisesta helpompaa. Ylipäätään siististi kirjoitetun koodin parissa on mukavampi työskennellä tulevaisuudessa, helpottaen projektin parissa työskentelevien tekemistä. (Vuorinen, 2014).

Laadukkaan, eli puhtaan koodin kirjoittaminen vaatii ohjelmoijalta lukemattomien pienten tekniikoiden kurinalaista käyttämistä ja näiden käytön myötä opittua käsitystä ”puhtaudesta”. Kaikille ohjelmoijille tämä ei tule luonnostaan, vaan he joutuvat näkemään paljon vaivaa päästäkseen tilanteeseen, jossa osataan kirjoittaa koodia puhtaasti. Tämän opitun käsityksen perusteella voidaan paitsi tunnistaa hyvin kirjoitettu koodi huonosti kirjoitetusta mutta myös käyttää hankittua kurinalaisuutta huonon koodin siistimisessä. (Martin 2009, 7).

Mikäli ohjelmoijalta puuttuu käsitys siitä, millaista puhtaan koodin tulisi olla, hän voi nähdä koodin sotkuisena, mutta ei osaa korjata sitä. Tarvittavat taidot omatessaan puhtaaseen koodiin liittyen ohjelmoija osaa tunnistaa erilaisia vaihtoehtoisia tapoja toteuttaa sotkuinen koodi puhtaammin ja suunnitella tavan, jolla päästä tavoitteeseen. (Martin 2009, 7).

3.3.1 Puhtaan koodin määritelmä

Puhtaalle koodille ei ole olemassa tarkkaa määritelmää eikä sitä voi todennäköisesti mitata millään työkalulla. On silti olemassa hyödyllisiä työkaluja, jotka voivat auttaa pääsemään lähemmäksi puhdasta koodia, mutta ne eivät itsessään ole riittäviä. Koneet eivät toistaiseksi

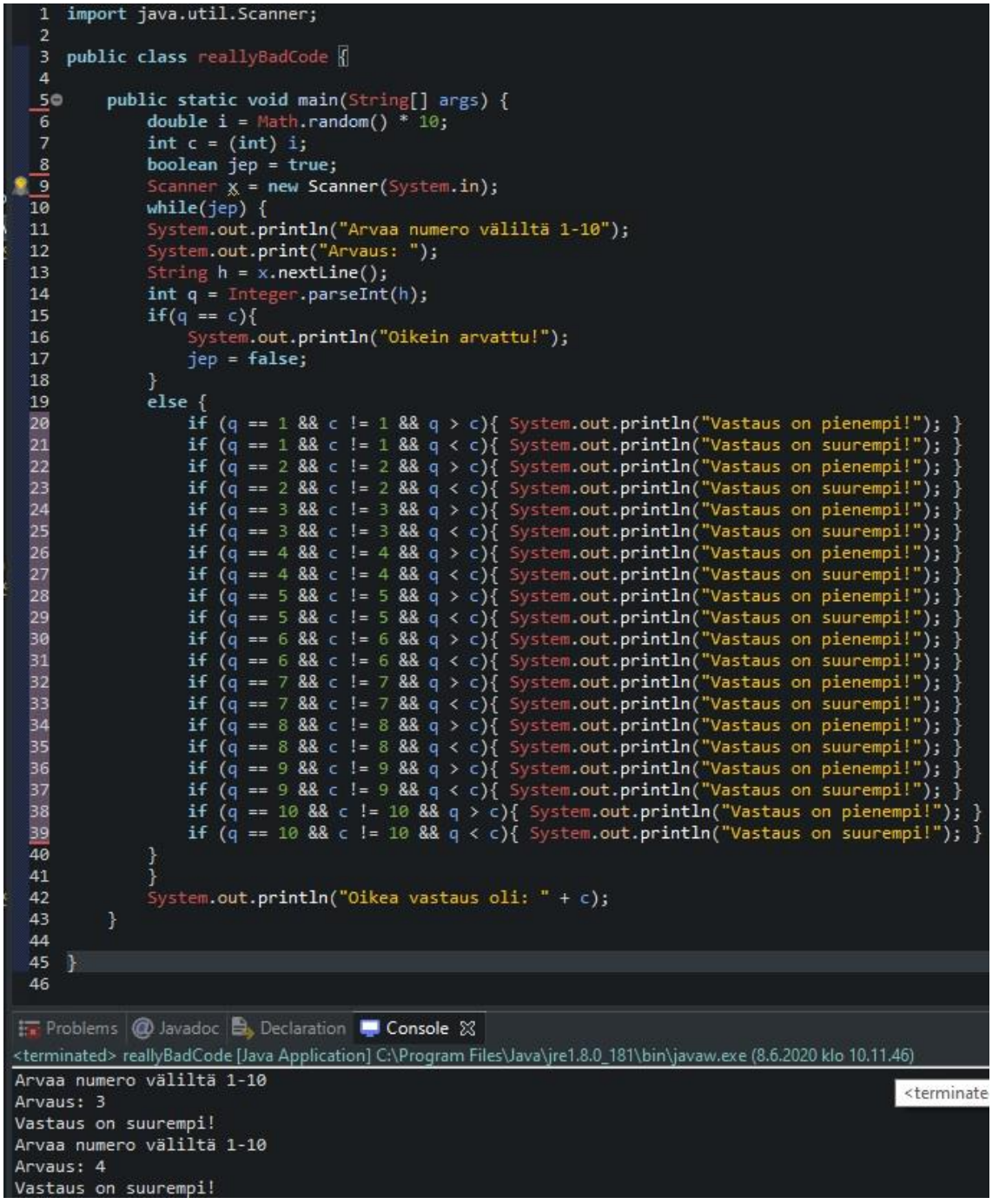
voi määrittää onko koodi puhdasta vai ei, vaan se jää ihmisten päätettäväksi. Ohjelmistokehittäjät käyttävät enemmän aikaa koodin lukemiseen kuin sen kirjoittamiseen ja joka kerta kun koodia muokata tai lisätä, on tutustuttava ympäröivään koodiin. (Anaya 2018, 8).

Koodia kirjottaessa on tärkeää ottaa huomioon, miten hyvin ihminen pystyy tulkitsemaan sitä. Luettavuus on ensimmäinen asia, jonka koodiin tutustuva ohjelmoija havaitsee, joten sen tulee olla selkeää, sillä jos toiset ohjelmoijat eivät pysty sujuvasti lukemaan tuotettua koodia, he eivät pysty tehokkaasti käyttämään sitä itse. On myös oleellista kiinnittää huomiota koodin designiin, kuten sisennyksiin, joka myös osaltaan parantaa koodin luettavuutta ja vähentää koodin lukijan taakkaa. (Padolsey 2020, 17-18).

Mikäli koodi ei ole säännönmukaisesti jäsennettyä ja sallitaan eri ohjelmoijien toimia omilla tavoillaan, lopputuloksena on hankalasti tulkittavaa koodia, joka on harhaanjohtavaa, altista virheille ja bugeille. Koodin tulisi sen sijaan olla jo ensivilkaisulla helposti luettavaa ja ymmärrettävää. Mikäli koodin rakenteesta päästään ohjelmoijien kesken yhteisymmärrykseen saadaan aikaan tiettyjä kaavoja noudattavaa koodia, jolloin on paljon helpompaa löytää mahdollisia virheitä koodin seasta. (Anaya 2018, 10).

On myös olemassa ero helposti luettavan ja helposti ylläpidettävän koodin välillä. Puhdas koodi on sellaista, että muiden on helppo jatkokehittää sitä, ei pelkästään lukea. (Martin 2009, 9)

Tätä opinnäytetyötä varten luotiin yksinkertainen esimerkkiohjelma, jonka avulla voidaan havainnollistaa huonon koodin ja puhtaan koodin eroavaisuuksia. Kuviossa 1 esitellään yksinkertaisen Java-ohjelmointikielellä kirjoitetun numeronarvaus-pelin ohjelmakoodi, joka on luotu hyvin sotkuisesti tarkoituksellisesti. Kuviosta voidaan havaita, vaikka ohjelma on erittäin yksinkertainen ja suoraviivainen, on sen koodia hankala seurata ainakin ensisilmäyksellä. Havaittavimmat puutteet koodissa ovat parametrien huono nimeäminen, jolloin ne eivät kerro lukijalle tehtäväänsä, sekava if-lauseiden käyttö ja huonosti formatoitu (sijoiteltu) koodi. Lisäksi koodissa on runsaasti toistoa, joka kasvattaa ohjelmakoodin rivimäärää.



```
import java.util.Scanner;

public class reallyBadCode {

    public static void main(String[] args) {
        double i = Math.random() * 10;
        int c = (int) i;
        boolean jep = true;
        Scanner x = new Scanner(System.in);
        while(jep) {
        System.out.println("Arvaa numero väliltä 1-10");
        System.out.print("Arvaus: ");
        String h = x.nextLine();
        int q = Integer.parseInt(h);
        if(q == c){
            System.out.println("Oikein arvattu!");
            jep = false;
        }
        else {
            if (q == 1 && c != 1 && q > c){ System.out.println("Vastaus on pienempi!"); }
            if (q == 1 && c != 1 && q < c){ System.out.println("Vastaus on suurempi!"); }
            if (q == 2 && c != 2 && q > c){ System.out.println("Vastaus on pienempi!"); }
            if (q == 2 && c != 2 && q < c){ System.out.println("Vastaus on suurempi!"); }
            if (q == 3 && c != 3 && q > c){ System.out.println("Vastaus on pienempi!"); }
            if (q == 3 && c != 3 && q < c){ System.out.println("Vastaus on suurempi!"); }
            if (q == 4 && c != 4 && q > c){ System.out.println("Vastaus on pienempi!"); }
            if (q == 4 && c != 4 && q < c){ System.out.println("Vastaus on suurempi!"); }
            if (q == 5 && c != 5 && q > c){ System.out.println("Vastaus on pienempi!"); }
            if (q == 5 && c != 5 && q < c){ System.out.println("Vastaus on suurempi!"); }
            if (q == 6 && c != 6 && q > c){ System.out.println("Vastaus on pienempi!"); }
            if (q == 6 && c != 6 && q < c){ System.out.println("Vastaus on suurempi!"); }
            if (q == 7 && c != 7 && q > c){ System.out.println("Vastaus on pienempi!"); }
            if (q == 7 && c != 7 && q < c){ System.out.println("Vastaus on suurempi!"); }
            if (q == 8 && c != 8 && q > c){ System.out.println("Vastaus on pienempi!"); }
            if (q == 8 && c != 8 && q < c){ System.out.println("Vastaus on suurempi!"); }
            if (q == 9 && c != 9 && q > c){ System.out.println("Vastaus on pienempi!"); }
            if (q == 9 && c != 9 && q < c){ System.out.println("Vastaus on suurempi!"); }
            if (q == 10 && c != 10 && q > c){ System.out.println("Vastaus on pienempi!"); }
            if (q == 10 && c != 10 && q < c){ System.out.println("Vastaus on suurempi!"); }
        }
        }
        System.out.println("Oikea vastaus oli: " + c);
    }

}
```

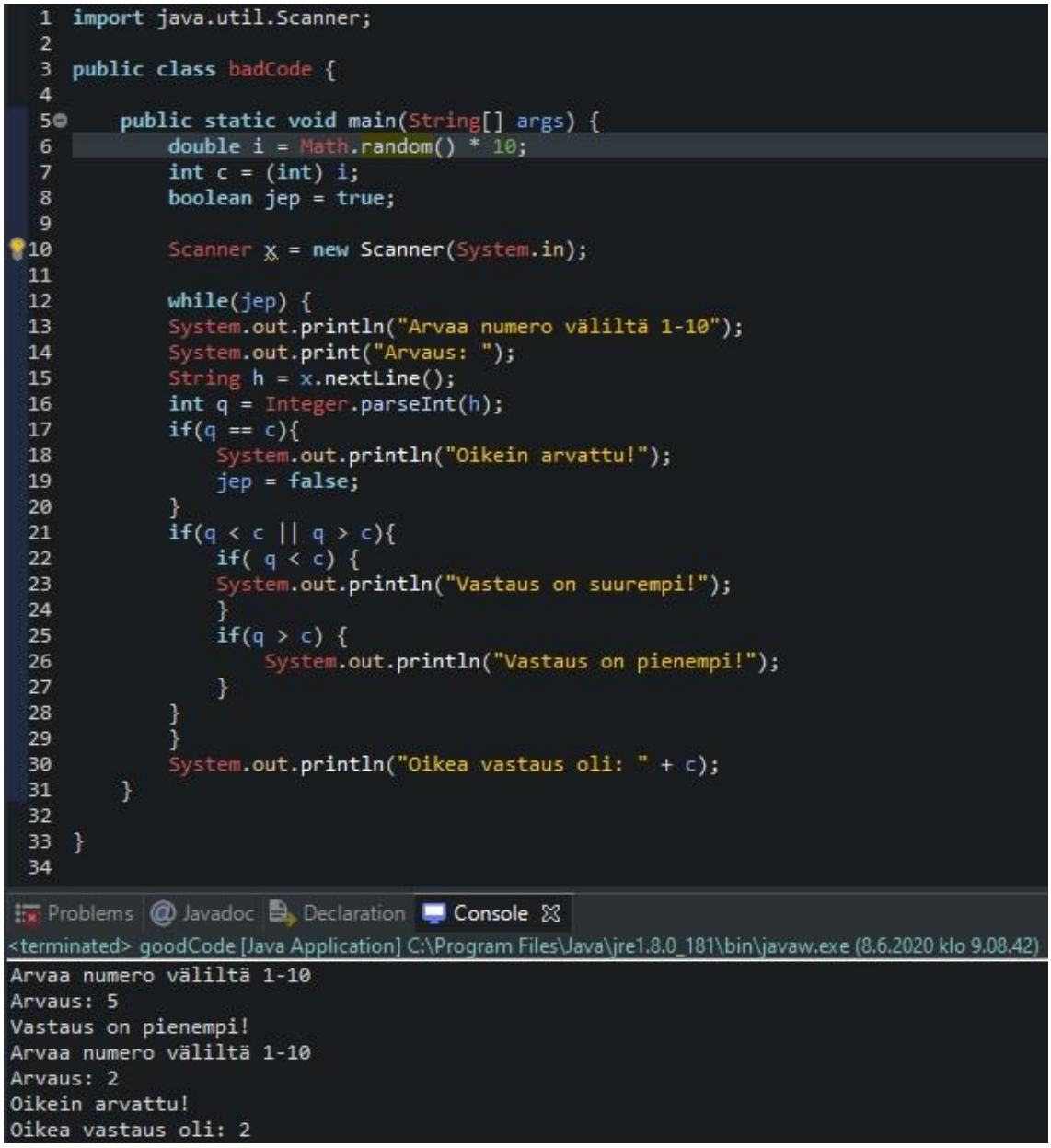
Problems | Javadoc | Declaration | Console

<terminated> reallyBadCode [Java Application] C:\Program Files\Java\jre1.8.0_181\bin\javaw.exe (8.6.2020 klo 10.11.46)

```
Arvaa numero väliltä 1-10
Arvaus: 3
Vastaus on suurempi!
Arvaa numero väliltä 1-10
Arvaus: 4
Vastaus on suurempi!
```

Kuvio 1: Esimerkki erittäin sekavasti kirjoitetusta ohjelmakoodista

Kuviossa 2 havainnollistetaan toiminnaltaan täysin sama ohjelma, mutta hieman siistitymmällä koodilla. Jo ensisilmäyksellä voidaan havaita hienoinen parannus, eikä ohjelma näytä enää aivan yhtä sekavalta. Koodi on kuitenkin vielä huonosti toteutettua, sillä parametrien nimet eivät kerro lukijalle mitään, eikä myöskään luokan nimi. If-lauseissa on myös sekavuutta ja muotoilu on toteutettu huonosti, mikä vaikeuttaa koodin lukemista.



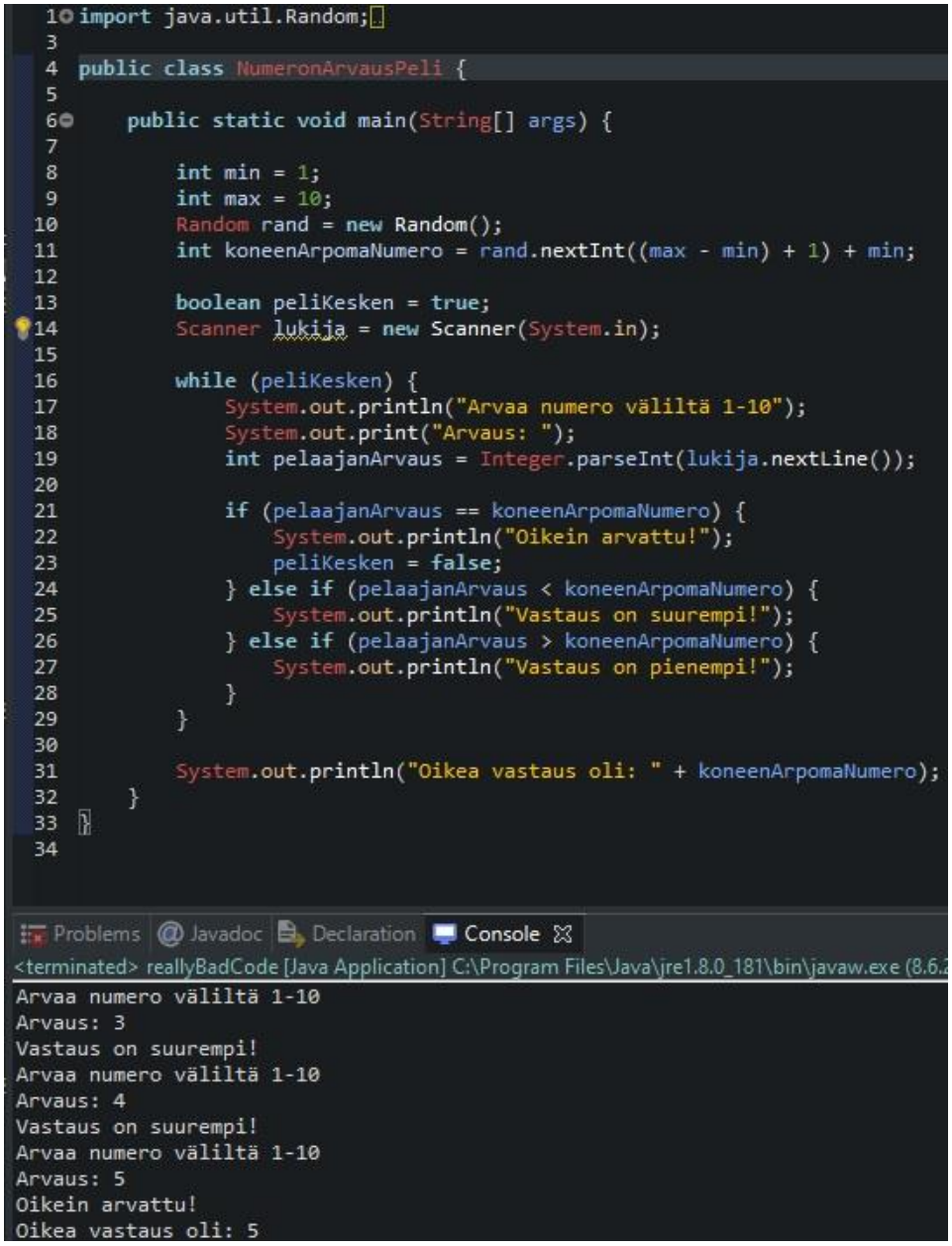
```
1  import java.util.Scanner;
2
3  public class badCode {
4
5      public static void main(String[] args) {
6          double i = Math.random() * 10;
7          int c = (int) i;
8          boolean jep = true;
9
10         Scanner x = new Scanner(System.in);
11
12         while(jep) {
13         System.out.println("Arvaa numero väliltä 1-10");
14         System.out.print("Arvaus: ");
15         String h = x.nextLine();
16         int q = Integer.parseInt(h);
17         if(q == c){
18             System.out.println("Oikein arvattu!");
19             jep = false;
20         }
21         if(q < c || q > c){
22             if( q < c) {
23             System.out.println("Vastaus on suurempi!");
24             }
25             if(q > c) {
26                 System.out.println("Vastaus on pienempi!");
27             }
28         }
29         }
30         System.out.println("Oikea vastaus oli: " + c);
31     }
32
33 }
34
```

Problems | Javadoc | Declaration | Console

```
<terminated> goodCode [Java Application] C:\Program Files\Java\jre1.8.0_181\bin\javaw.exe (8.6.2020 klo 9.08.42)
Arvaa numero väliltä 1-10
Arvaus: 5
Vastaus on pienempi!
Arvaa numero väliltä 1-10
Arvaus: 2
Oikein arvattu!
Oikea vastaus oli: 2
```

Kuvio 2: Esimerkki hieman siistitystä ohjelmakoodista

Kuviossa 3 havainnollistetaan siistitty ohjelmakoodi samasta esimerkkiohjelmasta ja jo ensisilmäyksellä voidaan havaita, että koodi on jo luokan nimestä lähtien huomattavasti aiempia esimerkkejä helpompaa lukea. Parametrit on nimetty asianmukaisesti, joten on helppo ymmärtää mitä arvoja ne sisältävät, joka myös helpottaa koodin lukemista. Muotoilu on myös toteutettu paremmin, lohkoen välilyöntien avulla eri toiminnallisuuksia. Voidaan myös havaita, että rivimäärä on pienentynyt ja turha toisto on jäänyt koodista pois.



```java
import java.util.Random;

public class NumeronArvausPeli {

    public static void main(String[] args) {

        int min = 1;
        int max = 10;
        Random rand = new Random();
        int koneenArpomaNumero = rand.nextInt((max - min) + 1) + min;

        boolean peliKesken = true;
        Scanner lukija = new Scanner(System.in);

        while (peliKesken) {
            System.out.println("Arvaa numero väliltä 1-10");
            System.out.print("Arvaus: ");
            int pelaajanArvaus = Integer.parseInt(lukija.nextLine());

            if (pelaajanArvaus == koneenArpomaNumero) {
                System.out.println("Oikein arvattu!");
                peliKesken = false;
            } else if (pelaajanArvaus < koneenArpomaNumero) {
                System.out.println("Vastaus on suurempi!");
            } else if (pelaajanArvaus > koneenArpomaNumero) {
                System.out.println("Vastaus on pienempi!");
            }
        }

        System.out.println("Oikea vastaus oli: " + koneenArpomaNumero);
    }
}
```

Problems Javadoc Declaration Console

<terminated> reallyBadCode [Java Application] C:\Program Files\Java\jre1.8.0_181\bin\javaw.exe (8.6.2

```
Arvaa numero väliltä 1-10
Arvaus: 3
Vastaus on suurempi!
Arvaa numero väliltä 1-10
Arvaus: 4
Vastaus on suurempi!
Arvaa numero väliltä 1-10
Arvaus: 5
Oikein arvattu!
Oikea vastaus oli: 5
```

Kuvio 3: Esimerkki siistitystä ohjelmakoodista

Angularin lomakkeisiin liittyvään koodiin on saatavilla tyyliopas, jonka avulla voidaan pitää koodi siistinä ja yhtenäisenä. Tutkimukseen liittyvän koodin tutkimista varten kiinnitettiin huomiota muun muassa seuraaviin tyylioppaasta (Google b, 2020) löytyviin ohjeisiin:

-Pienet funktiot (helpompi testata ja lukea)

-Yhtenäinen nimeäminen (auttavat löytämään etsityn koodin nopeammin)

-Testiluokkien nimeämiskäytäntö (helppo tunnistaa testeiksi)

-Toiston välttäminen (vähentää koodin määrää)

-Projektin kansiorakenne (helpompi kehittää projektin kasvaessa)

3.3.2 Koodikatselmointi

Koodikatselmointi tarkoittaa tuotetun koodin tarkastelua mahdollisien ongelmien löytämiseksi. Tällä pyritään ehkäisemään virheiden päätymistä testaajille tai peräti asiakkaille saakka. Esimerkkejä koodikatselmoinnin hyödyistä:

- Tehokkaampaa kuin testaus
- Voi nopeuttaa koodin valmistumista
- Voi auttaa luomaan laadukkaita ja luotettavia ohjelmistoja
- Voi kehittää kokonaista kehitystiimiä

Näiden hyötyjen avulla saadaan karsittua myös taloudellisia menoja ohjelmistokehityksessä. (Rawat 2014, 23-24).

Ohjelmakoodin laadun parantamista silmällä pitäen, koodikatselmointia voidaan pitää tehokkaimpana ja yleisimpänä metodina, testaamisen ohella. Koodikatselmoinnin historia ulottuu aina 1970-luvulle asti ja sieltä saakka sitä on pidetty parhaana tapana ohjelmien laadun parantamiseen. Koodikatselmoinnissa kollega tai kollegat tarkistelevat koodia, etsien virheitä ohjelmakoodissa, jotka kehittäjä poistaa tai korjata ja parantaa koodin laatua. Viimeisen neljän vuosikymmenen ajalta kerätty empiirinen todistusaineisto on todistanut yhä uudestaan, että koodikatselmointi on tehokasta virheiden löytämisessä ja myös kustannustehokasta, sillä viat löytyvät alhaisilla kustannuksilla. Lisäksi koodikatselmointi on käytännöllistä, sillä se on helppoa toteuttaa ja on olemassa valmiit standardit tai menettelytavat, joiden mukaan koodikatselmointi voidaan suorittaa. (Zhu 2017, 3).

Yksi koodikatselmoinnin päätavoitteista on löytää ja ratkaista mahdolliset virheet ohjelmistotuotteen elinkaaren mahdollisimman aikaisessa vaiheessa. Koodia työstävän henkilön itse voi olla erittäin vaikeaa havaita virheitä omassa tuotoksessaan, joten toisen henkilön suorittama koodikatselmointi on tarpeen. Monessa tapauksessa kehittäjä etsii virhettä pitkään tuloksetta omasta tuotoksestaan, mutta saman tien, kun pyydetään toista henkilöä katselmoimaan koodi, hän löytää virheen hyvin nopeasti. Toinen tavoite koodikatselmoinnissa on itseluottamuksen tuominen projektiin, jotta voidaan olla varmoja, että vaaditut vaatimukset täyttyvät ja asiakkaan tarpeet tulee huomioitua. (Westfall 2009, 444).

Katselmoinnin suorittaja(t) valitaan työtuotteen perusteella, ja katselmoijan/katselmoijien tulisi olla kehittäjän kollegoita ja omata tarpeeksi teknistä tietämystä, jotta katselmoitava kohde voidaan käydä läpi perusteellisesti. On myös pidettävä huoli, että on varattu tarpeeksi aikaa katselmoinnin suorittamiseen. Yleinen sääntö katselmoinnissa on, että samalla tasolla työskentelevät henkilöt katselmoivat toistensa työtä, eikä esimerkiksi esimiesasemassa oleva

henkilö tästä syystä yleensä katselmoi alaistensa töitä. Ylempiarvoisen tekemää katselmointia pidetään huonona siitä syystä, että kehittäjät voivat kokea mahdollisten virhelöydöksien heijastavan heihin huonoutena esimiehen silmissä. Lisäksi mikäli esimies seuraa samantasoisten työntekijöiden katselmointia vierestä, saatetaan jättää virheitä raportoimatta, jottei saateta tuotosta luonutta henkilöä huonoon valoon esimiehen silmissä. (Westfall 2009, 447).

Koodikatselmointi voidaan suorittaa erilaisilla metodeilla ja ne voidaan jaotella korkealla tasolla kahteen kategoriaan: muodolliseen koodikatselmointiin ja kevyeen katselmointiin. Muodollisen koodikatselmoinnin puolella suosituin implementaatio on Fagan tarkastus (Bodner, 2018), joka on hyvin strukturoitu prosessi virheiden löytämiseksi niin koodista, mutta myös spesifikaatioista ja designista. Kevyt katselmointi on tänä päivänä suositumpaa kehittäjätiimeissä ja se voidaan jakaa seuraaviin alakategorioihin:

- Pariohjelmointi (välitön katselmointi)
- Synkroninen katselmointi (olan yli)
- Epäsynkroninen katselmointi (työkaluavusteinen)
- Kokouspohjainen katselmointi silloin tällöin

Pariohjelmoinnissa toinen henkilö tuottaa koodia ja toinen katselmoi sitä samaan aikaan, kiinnittäen huomiota mahdollisiin ongelmakohtiin ja antaen ideoita koodin parantamiseksi lennossa.

Synkronisessa katselmoinnissa kehittäjä tuottaa koodia ja sen ollessa valmis pyytää katselmoijan tietokoneensa ääreen käymään tuotetun koodin läpi.

Epäsynkroninen katselmointi suoritetaan siten, että kehittäjän tuotettua koodin, hän tekee koodin näkyväksi katselmoijaa varten ja jatkaa seuraavan tehtävänsä parissa. Koodin katselmointi suoritetaan eri työpisteellä ja valittujen työkalujen avulla raportoidaan löydöksistä koodin tuottajalle.

Kokouspohjainen katselmointi suoritetaan isommassa ryhmässä kehittäjän esitellessä tuottamaansa koodia muille kokoukseen osallistujille, jotka yrittävät paikallistaa mahdollisia virheitä ja antaa ehdotuksia koodin parantamiseksi. (Bodner, 2018).

Tässä tutkimuksessa valittiin katselmointitavaksi epäsynkroninen katselmointi, sillä tarkasteltavaksi otettiin jo tuotantokäytössä olevaa koodia, jonka luomisesta oli kulunut jo muutamia viikkoja. Tutkija siis otti katselmoitavaksi jo aikaisemmin tuotettua ja yrityksen puolelta tuotantokäyttöön hyväksyttyä koodia, jotta saatiin todellinen kuva hyvien ohjelmointikäytäntöjen noudattamisesta, sen sijaan että olisi pyydetty luomaan koodia katselmointia varten, jolloin olisi todennäköisesti kiinnitetty enemmän huomiota hyvien ohjelmointikäytäntöjen huomiointiin, eivätkä tutkimustulokset olisi olleet yhtä luotettavia.

Koodin tuottamisprosessi Leanvay Oy:ssä kuvataan kappaleessa 5.2.

4 Tutkimuksellinen kehittämistyö

Tässä opinnäytetyössä tutkittiin kvalitatiivisia tutkimusmenetelmiä käyttäen olemassa olevaa LogiPlan-toiminnanohjausjärjestelmää ja sitä, onko hyviä ohjelmointikäytäntöjä noudatettu Angular-sovelluskehyksen reaktiivisiin lomakkeisiin liittyvässä koodissa. Tutkimuksessa pyrittiin löytämään mahdollisia parannuskohteita ja konkreettisia parannusehdotuksia näiden korjaamiseksi.

Tietoa kerättiin osallistuvalla havainnoilla, koodikatselmoinnilla, haastattelulla sekä vapaamuotoisilla keskusteluilla. Alkutiedot tutkimukseen saatiin perehdyttämisellä työtehtäviin, jonka jälkeen syventymistä tutkimusaiheeseen jatkettiin omatoimisella havainnoinnilla, järjestämällä puolistrukturoitu teemahaastattelu järjestelmän kehittäjälle sekä epämuodollisilla haastatteluilla yrityksen kokeneempien työntekijöiden kanssa. Tutkimuksen loppuvaiheessa suoritettiin koodikatselmointi valikoituihin luokkiin ohjelmakoodissa. Osana tutkimusta järjestetty puolistrukturoitu teemahaastattelu antaa tietoa tutkimuskohteen nykytilanteesta ja historiasta.

4.1 Laadullinen tutkimusmenetelmä

Tutkimusaineiston laatu ohittaa tärkeysjärjestyksessä määrän laadullisen tutkimuksen kohdalla ja sen tarkoituksena on auttaa ymmärtämään tai tulkitsemaan asiaa tai ilmiötä, jota tutkitaan. Sen sijaan, että pyrittäisiin saamaan aikaiseksi tilastollinen yleistys, pyrkimyksenä voi olla emansipatorinen tiedonintressi (olemassa olevien ajattelumallien kyseenalaistaminen) ja hermeneuttinen tiedonintressi (ilmiön selittäminen ymmärrettävään muotoon) jotta voidaan päästä uudenlaiseen ajattelumalliin. Tutkimusaineistoa ei tarvita määrällisesti tällöin paljon, mutta aineisto tulee analysoida hyvin tarkkaan. (Vilkka 2015, 96).

Laadullisessa tutkimuksessa pyritään analysoimaan mahdollisimman tarkasti pieni määrä tapauksia saavuttaen mahdollisimman korkeatasoinen kattavuus käsitteellistämisestä. Tutkimuskohteesta pyritään antamaan yksityiskohtainen ja tarkka kuvaus, ottaen huomioon, että tutkimuskohde täyttää tutkittavan asian tunnusmerkit, mikäli ei tutkita ennen tutkimatonta kohdetta. Ennakkoasetelmat tai määritelmät eivät ole välttämättömiä laadullista tutkimusta tehtäessä, vaan puhutaan aineistolähtöisestä analyysistä, joka kumpuaa empiirisestä aineistosta. Aineisto on tärkeää rajata, sillä ilman asianmukaista rajaamista aineistoa kertyy helposti liikaa. (Eskola & Suoranta 1998, 14).

4.2 Osallistuva havainnointi

Osallistuva havainnointi tarkoittaa tutkimustapaa, jossa tutkija itse osallistuu sovitun ajanjakson ajaksi tutkimuskohteensa toimintaan, etukäteen määritettyä, teoreettista näkökulmaa käyttäen. Edellytyksenä osallistuvalle havainnoinnille on se, että tutkija pääsee osaksi yhteisöä, jossa tutkittava asia sijaitsee. On suotavaa, että osallistuva havainnointi on suunnattua. Osallistuvaa havainnointia voidaan tehostaa kohdistettua havainnointia käyttäen, kun tutkimusongelma on saatu täsmennettyä. (Vilkka 2006, 39-40).

Suoritettaessa osallistuvaa havainnointia, tutkijan on tärkeä omaksua muitakin rooleja, eikä olla pelkästään tutkija, sillä vuorovaikutus yhteisössä, jossa tutkimusta tehdään, on tutkimuksen onnistumisen kannalta merkittävää. Havainnointi on subjektiivista toimintaa siinä mielessä, että tutkijan huomio saattaa kiinnittyä myös epäoleellisiin seikkoihin eikä fokus välttämättä pysy tutkittavassa aiheessa, jolloin merkityksellisiä asioita voi jäädä huomaamatta. Havaintojen teko on vaihtelevaa tutkijan tilan mukaan, esimerkiksi mieliala ja aktivaatiotason voivat vaikuttaa havainnointiin. (Eskola & Suoranta 1998, 74).

Tässä tutkimuksessa osallistuva havainnointi suoritettiin tutkijan toimesta asiakasyrityksen toimitiloissa suorittaen LogiPlan-toiminnanohjausjärjestelmän käyttöliittymän päivitystyötä Wicket-sovelluskehyksestä Angular-sovelluskehykseen. Näin toimittaessa tutkija kerrytti ymmärrystä siitä, miten tutkimuksen kohteena olleita reaktiivisia lomakkeita käytetään ja minkälaisia toteutustapoja niiden suhteen on olemassa.

4.3 Haastattelu

Haastattelu sopii monenlaisiin tutkimustarkoituksiin joustavuutensa ansiosta. Haastattelun aikana on mahdollista kohdistaa tiedonhankintaa tarkemmaksi kielellisen vuorovaikutuksen avulla ja lisäksi voidaan selvittää motiiveja vastausten taustalla. (Hirsjärvi & Hurme 2015, 34).

Tässä tutkimuksessa käytetään haastattelua, sillä kehittämisen kohteena olevasta LogiPlan-järjestelmästä ei ole tarjolla juurikaan julkista tietoa, jota voisi kerätä muualta kuin suoraan järjestelmän kehittäjiltä. Tarkemmaksi haastattelumenetelmäksi valikoitui puolistrukturoitu teemahaastattelu, josta Hirsjärvi ja Hurme (2015, 47) todenneet, että mitään tarkkaa määritelmää kyseiselle haastattelumuodolle ole, mutta kysymykset ovat ennalta määriteltyjä ja niiden järjestystä voi haastattelija halutessaan vaihtaa. Puolistrukturoidussa haastattelussa ei ole olemassa tiettyjä valmiita vastausvaihtoehtoja, vaan haastateltavat voivat vastata kysymyksiin vapaasti, omin sanoin.

Tutkimuksessa suoritetussa haastattelussa kysyttiin LogiPlan-toiminnanohjausjärjestelmään liittyviä kysymyksiä (liite 2) järjestelmän ylläpitäjältä ja LeanVay Oy:n osakkaalta Chao

Fengiltä. Haastattelu noudatti puolistrukturoidun teemahaastattelun kaavaa, mutta keskustelu pyrittiin pitämään mahdollisimman vapaamuotoisena, jotta saatiin mahdollisimman kattava näkemys haastateltavan kannoista esitettyihin kysymyksiin. Haastattelun avulla pyrittiin saamaan kuva järjestelmän nykytilanteesta, sen toteutuksesta ja historiasta sekä ennen kaikkea tietoa siitä, millaista Leanvay Oy haluaisi koodin reaktiivisien lomakkeiden osalta olevan tulevaisuudessa.

4.4 Validiteetti ja reliabiliteetti

Reliabiliteetilla tarkoitetaan tutkimustuloksien toistettavuutta, eli jos asia tutkitaan toistamiseen, tulisi tutkimustuloksen olla sama, tai vastaavasti jos kaksi tai useampi tutkija tutkii samaa asiaa yhtä aikaa, tulisi tuloksien vastata toisiaan. Reliabiliteettiin tulee suhtautua silti tietyllä varauksella, sillä tutkimusta tehtäessä ei voida olettaa, että esimerkiksi kaksi tutkijaa kiinnittävät huomiota samoihin seikkoihin, vaikka tutkittava asia olisikin sama. Tällöin tutkimustulokset voivat poiketa toisistaan. Tällöin kyse on muuttuneiden tilanteiden seurauksesta, eikä sitä voida pitää järjestelmän heikkoutena. (Hirsjärvi & Hurme 2015, 186).

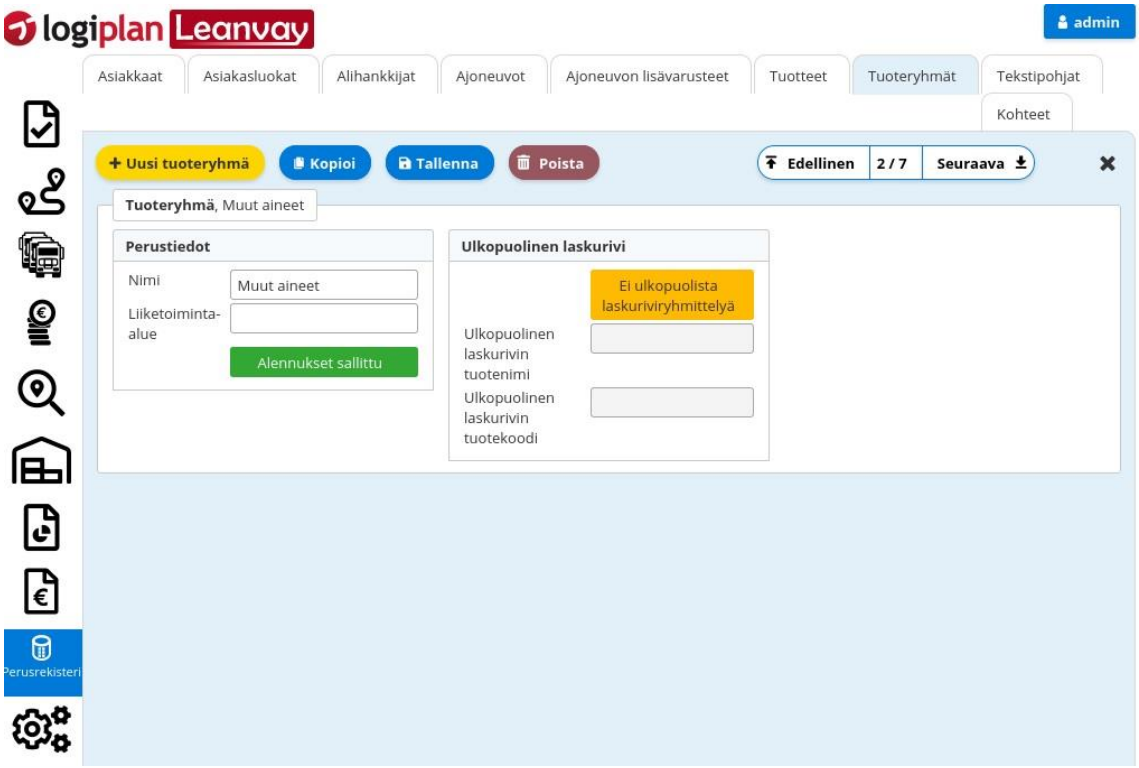
Validiteetti tarkoittaa tutkimuksessa käytetyn mittausmenetelmän kykyä mitata nimenomaan tutkimuskohdetta, ilman siitä eksymistä. (Tilastokeskus.) Validiteetilla määritetään tutkimuksen pätevyys, perusteellisuus ja saatujen tulosten oikeellisuus. Validiteetti voi kärsiä tutkijan virheiden vuoksi, kuten suhteiden ja periaatteiden virheellinen tulkinta, niiden havaitsemattomuus tai vääränlainen kysymysten asettaminen tutkimusta varten. (Saaranen-Kauppinen & Puusniekka 2006).

5 Lomakkeet LogiPlanissa

LogiPlan-toiminnanohjausjärjestelmä sisältää hyvin paljon lomakkeita ja niitä käytetään tiedon lisäämiseen, muokkaamiseen ja poistamiseen käyttäjien toimesta. Lomakkeet löytyvät tutkittavassa LogiPlan-toimistopäätteessä kunkin pääosion alta, jossa ne jakautuvat alasivuiksi. Esimerkkejä tällaisista pääosioista ovat esimerkiksi tilausten hallinta, laskutus, hinnastot ja perusrekisteri. Täydellinen lista lomakkeiden määrästä löytyy liitteestä 1.

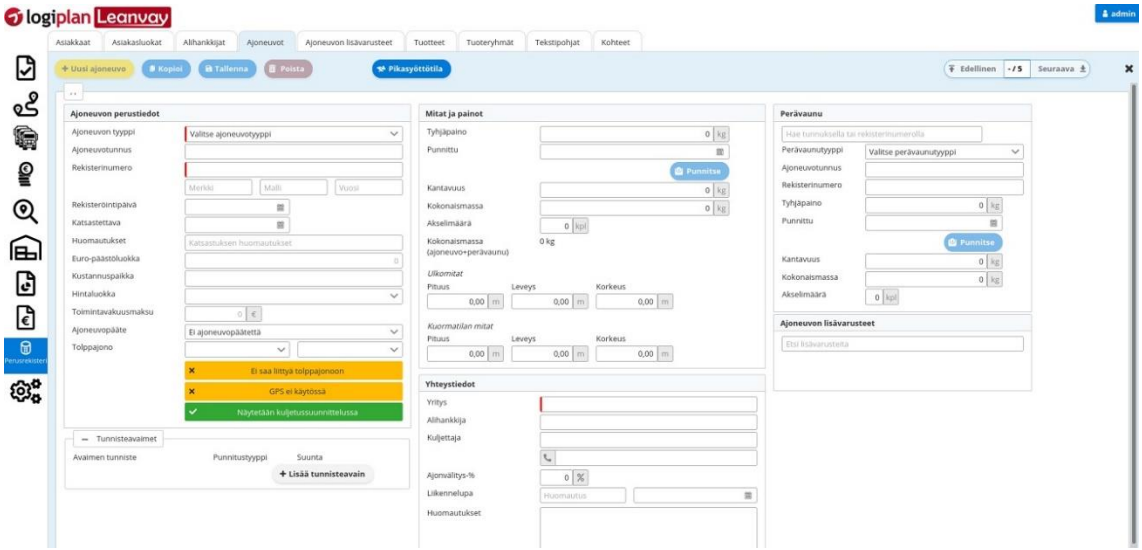
5.1 Lomakkeiden esittely

Osa järjestelmän lomakkeista on yksinkertaisia, tiedon ollessa vähäistä ja osa on hyvinkin monimutkaisia ja laajoja. Kuviossa 4 havainnollistetaan yksinkertainen tuoteryhmä-lomake LogiPlan-järjestelmästä, joka sisältää ainoastaan neljä tekstinsyöttökenttää sekä kaksi togglenappia työkalurivin lisäksi.



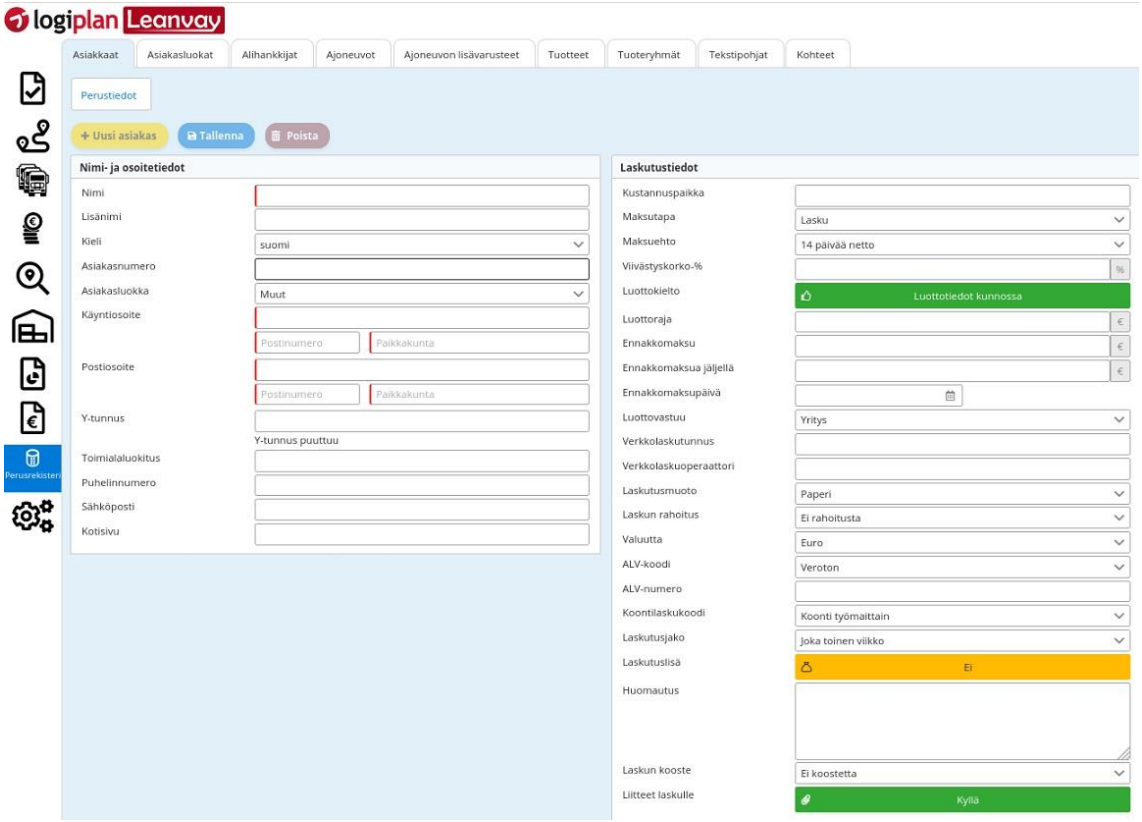
Kuvio 4: LogiPlan-järjestelmän tuoteryhmät-lomake

Kuviossa 5 havainnollistetaan huomattavasti monimutkaisempi ajoneuvolomake, joka sisältää runsaasti tiedonsyöttökenttiä.



Kuvio 5: LogiPlan-järjestelmän ajoneuvolomake

Kuviossa 6 havainnollistetaan asiakaslomake, joka on monimutkaisuudeltaan kahden edellisen esimerkin väliltä.



Kuvio 6: LogiPlan-järjestelmän asiakaslomake

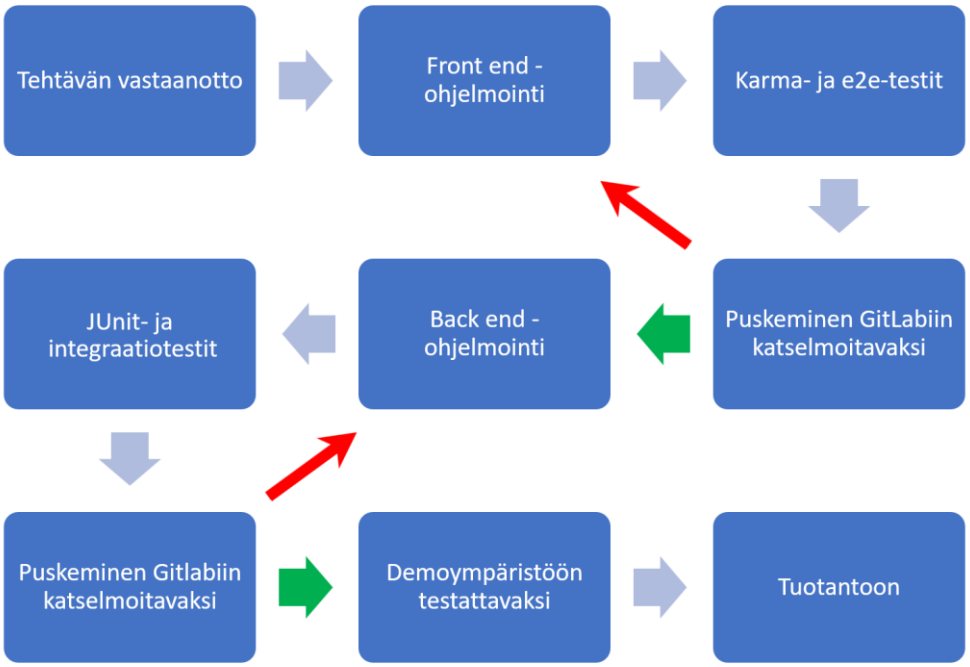
Kuten kuvioista 4, 5 ja 6 voidaan todeta, lomakkeiden monimutkaisuus vaihtelee suurestikin järjestelmässä. Suurin osa lomakkeista kallistuu monimutkaisuuden puolelle, yksinkertaiset lomakkeet ovat harvinaisempia.

5.2 Ohjelmakoodin tuottaminen lomakkeiden osalta

Lomakkeiden koodin mahdollinen parantaminen tarjoaa paljon etuja. Ylläpidettävyyden helpottuminen lienee yksi tärkeimmistä seikoista, joita koodin siivoamisella voidaan saada aikaan, sillä ylläpidettävyys on erittäin tärkeä osa luotua koodia. Mahdollisimman siisti koodi myös säästää kehittämiseen käytettävää aikaa, joka on yritykselle positiivista kehitystä vapauttaen resursseja muihin vaadittuihin toimintoihin yrityksen sisällä. Henkisen hyvinvoinnin lisääntyminen projektin parissa voidaan myös laskea etuihin, joita koodin parantamisella voidaan saavuttaa. (Feng 2020.)

Koodi tuotetaan Leanvay:ssä siten, että ohjelmoija käy merkkaamassa itsensä vastuuhenkilöksi Jiran Kanban-taululta löytyvälle työtehtävälle ja alkaa työstämään kyseistä tehtävää. Tutkijan tekemän käyttöliittymäpäivityksen osalta tehtävä sisälsi aina yhden sivun, joka tuli päivittää käyttämään Angularia Wicketin sijaan. Ohjelmointi aloitettiin sivun HTML-luokasta, edeten TypeScript-luokkiin. Kun front end oli valmis, se testattiin mock -datalla, joka saatiin itse luodusta json-tiedostosta. Tällöin luotiin myös sekä unit- että e2e-testit. Kun nämä oli ajettu onnistuneesti, tuotettu koodi puskettiin Git:llä repositorioon, jossa toinen ohjelmoija katselmoi tuotetun koodin. Kun saatiin hyväksyntä, voitiin siirtyä tekemään back end -puolta Javalla. Kaava oli back end -puolella sama, suoritettiin ohjelmointi, luotiin testit ja puskettiin lopuksi toisen katselmoitavaksi. Poikkeuksena se, että back end -puolella data sivulle saatiin tietokannasta, jota voitiin sivujen kautta muokata, toisin kuin front end -puolen mock-dataa. Kun sekä front- että back end -puoli ovat valmiita ja hyväksyttyjä, ne viedään demoympäristöön, joka vastaa tuotantoympäristöä. Tässä kohtaa oikea ihminen testaa tuotettua sivua manuaalisesti, ennen kuin tuotettu sivu viedään oikeaan tuotantoympäristöön.

Kuviossa 7 on esitetty koodin tuotantoprosessi Leanvay Oy:ssä UI-päivittämisen osalta, jota tutkija teki työharjoittelun ajan pääsääntöisesti.



Kuvio 7: Koodin tuottamisprosessi Leanvay Oy:ssä

6 Tutkimukseen valikoidut lomakkeet

Ohjelmakoodin lähtötilannetta tarkastellaan kahden eri sivun kohdalta, jotka ovat esitelty kuvioissa 4 (tuoteryhmäsivu) ja 6 (asiakassivu). Täten saadaan koodin katselmoinnin kohteeksi mahdollisimman yksinkertainen sivu sekä hieman monimutkaisempi sivu lomakkeiden osalta. Tarkasteltava koodi sisältää kummankin sivun HTML-luokan, joka vastaa sivun ulkonäöstä, sekä sitä vastaavan TypeScript-luokan, joka vastaa sivun toiminnallisuudesta.

Tuoteryhmät-sivu käydään perusteellisesti läpi koodin osalta, mutta koska logiikka, rakenne ja toteutus on samanlaista joka sivulla, asiakassivusta nostetaan esiin mahdollisesti poikkeavat asiat ohjelmoinnin osalta turhan toiston välttämiseksi. Koodin tarkastelussa keskitytään Angularin reaktiivisiin lomakkeisiin liittyvään koodiin front end-puolella, eikä back end-puolen koodia käydä läpi. Tarkasteltava koodi on jo siirretty käyttämään Angularia, joten vanhan käyttöliittymään Wicket-koodi ei sisälly ohjelmakoodin tarkasteluun.

6.1 Tuoteryhmät-sivu

Tuoteryhmät-sivulla käyttäjät voivat lisätä, muokata ja poistaa tuoteryhmiä. Sivu valittiin yksinkertaisuutensa vuoksi tutkimusta varten kontrastina monimutkaisemmalle asiakassivulle, jota käsitellään myöhemmin.

6.1.1 Käyttöliittymä

Kuviossa 4 esitetyssä tuoteryhmäkäyttöliittymässä on kahteen lohkoon jaettuna yhteensä kuusi kenttää, joissa käyttäjä voi syöttää tietoa järjestelmään. Nimi-kenttä on pakollinen, muut kentät vapaaehtoisia. Poikkeavana ominaisuutena lomakeryhmässä on kahden tekstinsyöttökentän käytöstä poisto, mikäli ulkopuolinen laskuriviryhmittely on poistettu napin avulla pois käytöstä.

6.1.2 Ohjelmakoodin lähtötilanne

Kuviossa 8 esitetään tuoteryhmä-sivun HTML-koodia sivun yläreunassa olevan työkalurivin osalta. Kuviossa on nähtävissä nappien painamisella kutsuttavat metodit sekä nappien käytöstä poisto, mikäli lomake ei ole asianmukaisesti täytetty pakollisten tietojen osalta. Ainoa pakollinen tieto tuoteryhmä-sivulla on nimi, joten sen täytettyään käyttäjä saa ”tallenna”- ja ”poista”-napit käyttöönsä. ”Uusi tuoteryhmä”- ja ”kopioi”-napit ovat poistettu käytöstä, mikäli ollaan jo luomassa uutta tuoteryhmää. Jos muokkaussivulle on saavuttu olemassa olevan tuoteryhmän kautta, ovat ”Uusi tuoteryhmä”- ja ”kopioi”-napit käytettävissä. Nappien validius toteutuu siis, kun saatavilla on olemassa olevan tuoteryhmän id.

```
<p-toolbar>
    <div class="ui-toolbar-group-left">
        <edit-default-buttons newButtonTextTranslationKey="productGroups.new"
            (onNewButtonClickAction)="newProductGroup()" [newButtonDisabled]="!productGroupForm.controls.id.value"
            (onCopyButtonClickAction)="copyProductGroup()" [copyButtonDisabled]="!productGroupForm.controls.id.value"
            (onSaveButtonClickAction)="saveProductGroup()" [saveButtonDisabled]="!productGroupForm.valid"
            (onRemoveButtonClickAction)="removeProductGroup()" [removeButtonDisabled]="!productGroupForm.valid">
        </edit-default-buttons>
    </div>
    <div class="ui-toolbar-group-right">
        <list-navigation-buttons [currentId]="productGroupForm.controls.id.value"
            [(editFormAnimation)]="productGroupEditFormAnimation" [disabled]="productGroupForm.dirty">
        </list-navigation-buttons>
        <button pButton type="button" class="ui-button-rounded transparent" icon="pi pi-times"
            [routerLink]="['/baseregister/product-groups']"></button>
    </div>
</p-toolbar>
```

Kuvio 8: Tuoteryhmä-sivun HTML-koodi (toolbar)

Nimi-kentän pakollisuus määritetään TypeScript-koodin puolella product-group-edit.component.ts-luokassa HTML-koodin sijaan, kuten on esitetty korostetusti kuviossa 9, muiden kenttien ollessa vapaaehtoisia, sillä niihin ei ole lisätty pakollisuutta indikoivaa Validators.required-ehtoa, jossa loppuosa ”required” ilmaisee tiedon olevan pakollinen.

```
43      ngOnInit() {
44          const productGroup = this.activatedRoute.snapshot.data.productGroup;
45
46          this.productGroupForm = this.formBuilder.group({
47              id: [undefined],
48              name: [undefined, Validators.required],
49              businessLine: [undefined],
50              discountAllowed: [undefined],
51              combineProductsForExternalInvoice: [undefined],
```

Kuvio 9: Tuoteryhmän muokkaussivun nimikentän pakollisuuden asettaminen TypeScript-luokassa

Tuoteryhmä-sivun HTML-rakenne on kuvattu kuviossa 10. Sivun rakenne muodostuu ylärivin työkalurivin alla seuraavasti: main-lohko pitää sisällään form-lohkon, jonka määrityksessä sidotaan product-group-edit.component.ts-luokassa luotu (kuvio 11) formGroup-ryhmä (productGroupForm), joka sisältää luodut formControl:t, id, nimi ja niin edelleen, HTML:ään.

```
19  <main id="productGroupEditPanel" [ngClass]="{ 'slit-in-horizontal': productGroupEditFormAnimation }">
20      <form [formGroup]="productGroupForm">
21          <div class="p-grid">
22              <div class="p-col">
23                  <p-fieldset>
24                      <p-header>
25                          <b>{{ 'productGroups.header' | translate }}</b>, {{ productGroupForm.controls.name.value }}
26                      </p-header>
27
28  >                   <div class="p-grid">…
99                      </div>
100                 </p-fieldset>
101             </div>
102         </div>
103     </form>
104 </main>
105
```

Kuvio 10: Tuoteryhmä-sivun HTML-rakenne

```
43      ngOnInit() {
44          const productGroup = this.activatedRoute.snapshot.data.productGroup;
45
46          this.productGroupForm = this.formBuilder.group({
47              id: [undefined],
48              name: [undefined, Validators.required],
49              businessLine: [undefined],
50              discountAllowed: [undefined],
51              combineProductsForExternalInvoice: [undefined],
52              combineProductName: [undefined],
53              combineProductCode: [undefined]
54          });
```

Kuvio 11: Tuoteryhmä-sivun formGroupin luominen

Kuviossa 10 havainnollistetaan että HTML-rakenne tuoteryhmä-sivulla jakautuu form-lohkon sisällä sivun ulkonäöllisistä seikoista johtuen edelleen ruudukkorakenteeseen, p-col-lohkoon ja lopulta p-fieldset-lohkoon, jonka sisällä varsinaiset lomakekentät sijaitsevat.

Lomakekentät ovat jaettu kahteen kolumniin: Perustiedot ja ulkopuolinen laskurivi. Lomakekenttien käyttö HTML-puolella esitetään perustietojen osalta seuraavassa kuviossa (kuvio 12).

```
28                  <div class="p-grid">
29                      <div class="p-col-4">
30                          <p-panel header="{{ 'productGroup.basicinfo' | translate }}">
31                              <div class="p-grid">
32                                  <div class="p-col-4">
33                                      {{ 'productGroups.name' | translate }}
34                                  </div>
35                                  <div class="p-col-8">
36                                      <input type="text" pInputText class="fullWidth" formControlName="name"
37                                          name="productGroupName" />
38                                  </div>
39                              </div>
40                              <div class="p-grid">
41                                  <div class="p-col-4">
42                                      {{ 'productGroup.businessLine' | translate }}
43                                  </div>
44                                  <div class="p-col-8">
45                                      <p-autoComplete formControlName="businessLine"
46                                          [suggestions]="searchBusinessLineResults"
47                                          (completeMethod)="searchBusinessLines($event)"
48                                          emptyMessage="{{'noResult' | translate}}"
49                                          [forceSelection]="true"
50                                          field="name"></p-autoComplete>
51                                  </div>
52                              </div>
53                              <div class="p-grid">
54                                  <div class="p-col-8 p-offset-4">
55                                      <p-toggleButton onLabel="{{ 'productGroups.discountAllowed' | translate }}"
56                                          class="optionOk"
57                                          offLabel="{{ 'productGroups.discountNotAllowed' | translate }}"
58                                          formControlName="discountAllowed" styleClass="fullWidth"
59                                          name="discountAllowedToggleButton">
60                                      </p-toggleButton>
61                                  </div>
62                              </div>
63                          </p-panel>
64                      </div>
```

Kuvio 12: Tuoteryhmä-sivun perustietojen HTML-koodi

Kuten kuviosta 12 voi havaita, fieldset-lohkon sisällä käytetään jälleen ruudukkolohkoa sekä asetetaan kolumni (p-col-4) perustietokenttää varten. Tämän jälkeen p-panel-lohkon sisällä annetaan otsikko sekä luodaan varsinaiset kentät tiedon keräämistä varten. P-panel lohkon sisällä luodaan ruudukot ja kolumnit nimi- ja liiketoiminta-alue-kenttiä varten sekä haetaan näiden käännetyt nimet json-tiedostosta (rivit 33 ja 42). Nimikenttä on yksinkertainen tekstinsyöttökenttä, joka määritetään koodin riveillä 36-37.

Hienostuneempi malli tekstikentästä on koodin riveillä 45-50 kuviossa 12 luotu autoComplete, joka hakee käyttäjän syötteen mukaan tietokannasta syötettä vastaavan ehdotuksen ja täyttää sen ennen sen täydellistä kirjoittamista. Ehdotuksien haun hoitaa rivillä 47 searchBusinessLines-metodi, joka on määritetty product-group-edit.component.ts-luokassa, jonka löytämät ehdotukset varastoidaan rivillä 46 olevaan product-group-edit.component.ts-luokassa luotuun searchBusinessLineResults-taulukkoon. Riviltä 48 löytyvä emptyMessage hakee json-tiedostosta löytyvän viestin, joka indikoi, ettei haettua tulosta löydy. Rivillä 49 määritetään, että ehdotetuista valinnoista on pakko valita, eikä syötettä voi antaa vapaasti kirjoittaen.

Koodin riveillä 55-60 kuviossa 12 on määritetty nappi, joka pitää sisällään boolean-arvon, jonka avulla määritetään, onko tuoteryhmän kohdalla alennukset sallittu. Napille määritetään on- ja off-labelit, joka vaihtaa tekstin napissa valinnan mukaisesti. Tekstit nappiin haetaan json-tiedostosta.

Huomion arvoista on formControlName-instanssien määrittäminen ja täten datan sitominen oikeisiin formControlleihin riveillä 36, 45 ja 58 (kuvio 12), jotka on määritetty product-group-edit.component.ts-luokassa (kuvio 11).

Viimeisenä lomakekenttänä tuoteryhmä-sivulla on ulkopuolinen laskurivi, joka sisältää yhden napin, jonka määritys tapahtuu riveillä 70-76 koodiesimerkissä (kuvio 13) joka toimii samalla logiikalla kuin aiemmin kuvailtu nappi koodin riveiltä 55-60 (kuvio 12), poikkeuksena tekstikenttien ’ulkopuolinen laskurivin tuotenimi’ ja ’ulkopuolinen laskurivin tuotekoodi’ poistaminen käytöstä, mikäli napista on valittu, ettei käytetä ulkopuolista laskuriviryhmittelyä. Lisäksi lomakkeessa on kaksi tekstinsyöttö-kenttää, riveillä 84-85 ja 93-94. Kaikki arvot on sidottu formControlName:n avulla TypeScript-koodiin luokassa product-group-edit.component.ts.

```
66                  <div class="p-col-4">
67                      <p-panel header="{{ 'productGroup.combineProduct' | translate }}">
68                          <div class="p-grid">
69                              <div class="p-col-7 p-offset-5">
70                                  <p-toggleButton
71                                      onLabel="{{ 'productGroups.combineProductsForExternalInvoice' | translate }}"
72                                      class="optionOk"
73                                      offLabel="{{ 'productGroups.dontCombineProductsForExternalInvoice' | translate }}"
74                                      formControlName="combineProductsForExternalInvoice" styleClass="fullWidth"
75                                      name="combineProductsForExternalInvoiceToggleButton">
76                                  </p-toggleButton>
77                              </div>
78                          </div>
79                          <div class="p-grid">
80                              <div class="p-col-5">
81                                  {{ 'productGroup.combineProductName' | translate }}
82                              </div>
83                              <div class="p-col-7">
84                                  <input type="text" pInputText class="fullWidth"
85                                      formControlName="combineProductName" />
86                              </div>
87                          </div>
88                          <div class="p-grid">
89                              <div class="p-col-5">
90                                  {{ 'productGroup.combineProductCode' | translate }}
91                              </div>
92                              <div class="p-col-7">
93                                  <input type="text" pInputText class="fullWidth"
94                                      formControlName="combineProductCode" />
95                              </div>
96                          </div>
97                      </p-panel>
98                  </div>
```

Kuvio 13: Tuoteryhmä-sivun ulkopuolisen laskurivin HTML-koodi

6.2 Asiakkaat-sivu

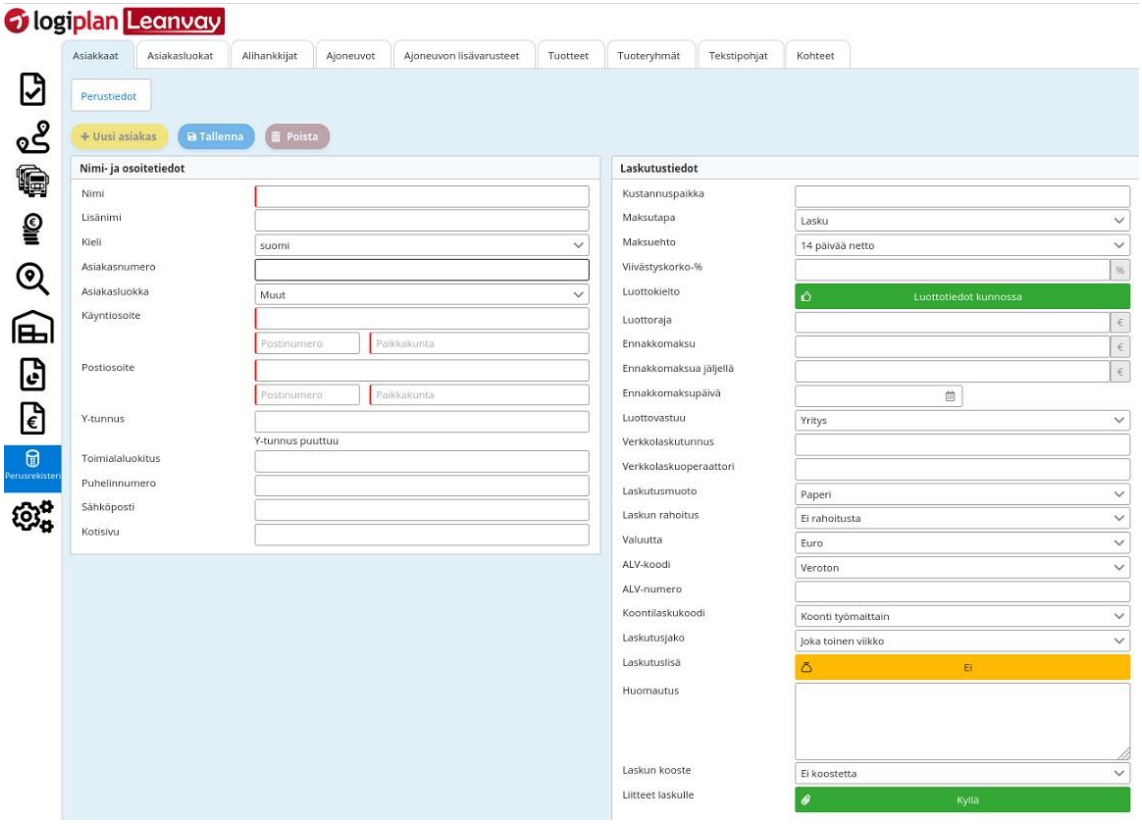
Asiakassivun koodia tutkittaessa havaitaan kompleksisuudessa selvä ero, joka on havaittavissa jo pelkkää käyttöliittymää tarkastellen kuviossa 6. Koodia ei kuitenkaan käydä läpi joka elementin osalta, sillä osa koodista vastaa tuoteryhmäsivulta aiemmin kuvattuja lomakekenttiä.

6.2.1 Käyttöliittymä

Asiakassivu koostuu kolmesta eri lomakeryhmästä: Perustiedot, työmaat ja kortit, joka on havainnollistettu korostaen kuviossa 14.

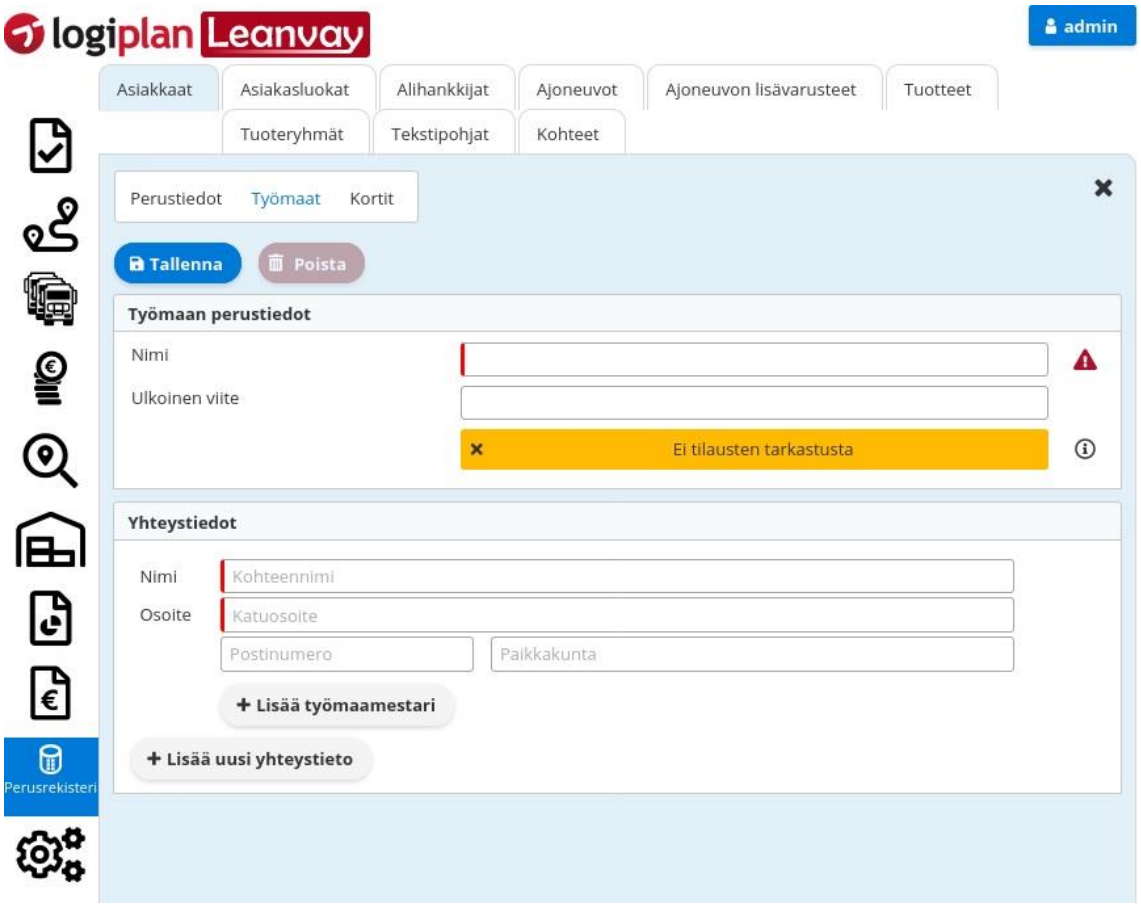
Kuvio 14: Asiakassivun alalomakkeet

Kuviossa 15 on esitetty perustietoja varten oleva lomakesivu, joka pitää sisällään nimensä mukaisesti asiakkaan perustietoja varten luodut lomakkeet.



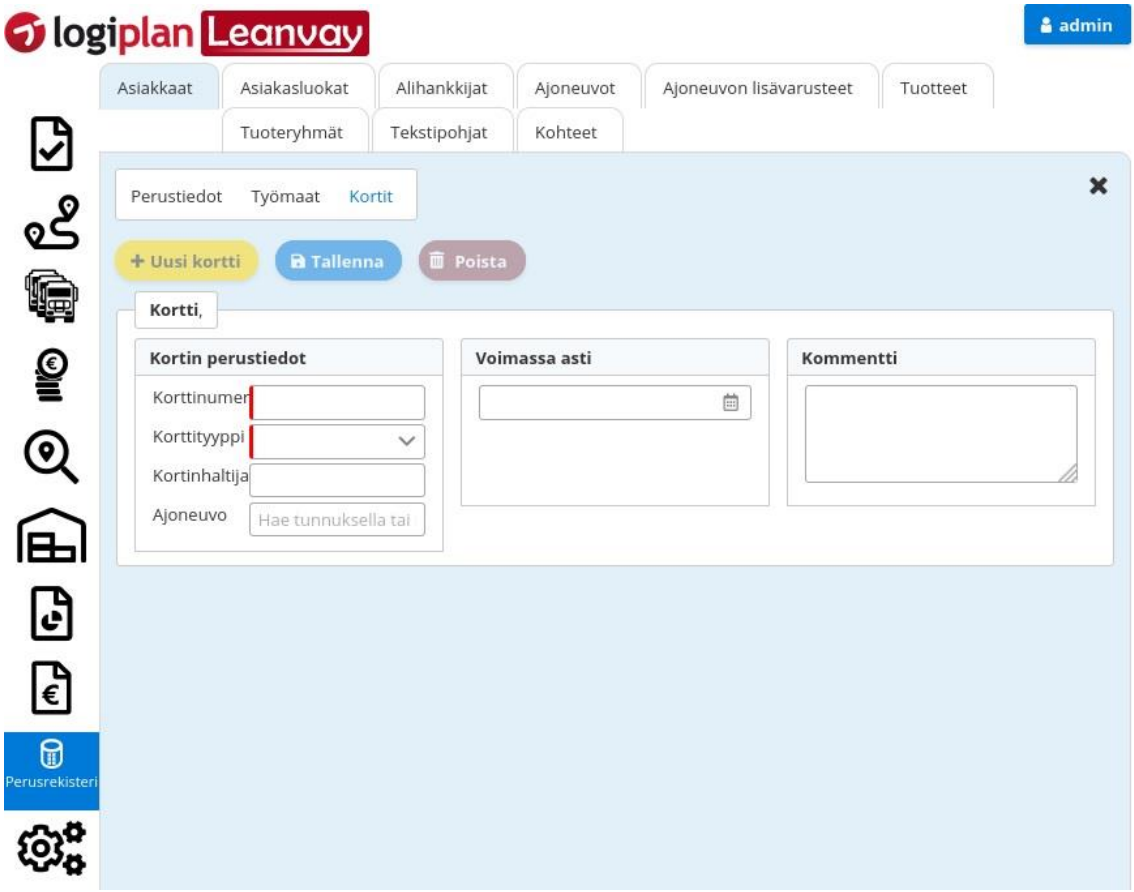
Kuvio 15: Asiakassivun perustietojen lomakeryhmä

Kuviossa 16 on havainnollistettu asiakkaan työmaat-muokkaussivu, jossa voidaan lisätä asiakaskohtainen työmaa tai työmaita. Lomakkeen perustiedot-kohta on suoraviivainen, mutta yhteystiedot-kohdan alta löytyvät napit kasvattavat lomakkeen kokoa dynaamisesti, joka tuo hieman kompleksisuutta koodiin, johon perehdytään alempana tässä kappaleessa.



Kuvio 16: Asiakassivun työmaasivun lomakeryhmä

Kuviossa 17 esitetään asiakkaan kortit-muokkaussivu, jossa voidaan hallinnoida asiakkaaseen liittyviä kortteja, kuten polttoainekortteja tai asiakaskortteja. Pakolliset kentät lomakkeessa ovat korttinumero ja korttityyppi.



Kuvio 17: Asiakassivun korttisivun lomakeryhmä

6.2.2 Ohjelmakoodin lähtötilanne

Kuviossa 15 havainnollistettu perustiedot-sivu on jaettu kahteen kolumniin: Nimi- ja osoitetietoihin sekä laskutustietoihin. Nimi- ja osoitetietojen puolella on kuusitoista kohtaa johon tietoa voi syöttää, näistä nimen ja osoitetietojen ollessa pakollisia, joka on määritetty HTML:ään sidotussa luokassa customer-basic-data.component.ts käyttämällä jo aiemmin mainittua Validators.required-määritelmää. Kuviossa 18 on esitetty customer-basic-data.component.ts-luokassa toteutettu lomakeryhmän luonti, korostettuna Angularin validaattoreista käytetty email-muotoilun validointi esimerkkinä siitä, miten erilaisia validointeja voidaan Angularin avulla toteuttaa.

```
63      ngOnInit() {
64          this.customerBasicDataEditForm = this.formBuilder.group({
65              id: [undefined],
66              customerNumber: [undefined],
67              name: [undefined, Validators.required],
68              name2: [undefined],
69              language: [undefined, Validators.required],
70              customerGroup: [undefined, Validators.required],
71              businessId: [''],
72              standardIndustrialClassification: [''],
73              visitAddress: this.formBuilder.group({
74                  id: [undefined],
75                  street: [undefined, Validators.required],
76                  postOffice: this.formBuilder.group({
77                      postalCode: [undefined, Validators.required],
78                      postOffice: [undefined, Validators.required],
79                  })
80              }),
81              postalAddress: this.formBuilder.group({
82                  id: [undefined],
83                  street: [undefined, Validators.required],
84                  postOffice: this.formBuilder.group({
85                      postalCode: [undefined, Validators.required],
86                      postOffice: [undefined, Validators.required],
87                  })
88              }),
89              phone: [undefined],
90              fax: [undefined],
91              email: [undefined, Validators.email],
92              website: [undefined],
```

Kuvio 18: Asiakassivun perustietojen validointi

Vaikka asiakassivu on aiemmin käsiteltyä tuoteryhmäsivua laajempi, koodi noudattaa pitkälti samoja kaavoja, mutta isommassa mittakaavassa. Poikkeuksena aiemmista tiedonsyöttötavoista tältä sivulta löytyy kaksi pudotusvalikkoa, joiden koodi on määritetty HTML-puolella Angularia käyttäen kuvion 19 osoittamalla tavalla. Lisäksi lomakkeessa on kenttiä, joihin voidaan syöttää pelkkiä numeroita.

```
<div class="p-col-8">
    <p-dropdown [options]="languageOptions"
        formControlName="language"
        styleClass="fullWidth"></p-dropdown>
</div>
</div>
<div class="p-grid">
    <div class="p-col-4">
        {{ 'customers.number' | translate }}
    </div>
    <div class="p-col-8 textToRight">
        <input type="text" pInputText class="fullWidth" formControlName="customerNumber" maxlength="255" />
    </div>
</div>
<div class="p-grid">
    <div class="p-col-4">
        {{ 'customerClass' | translate }}
    </div>
    <div class="p-col-8">
        <p-dropdown [options]="customerGroupOptions"
            formControlName="customerGroup"
            placeholder="{{ 'customerGroup.select' | translate }}"
            styleClass="fullWidth"></p-dropdown>
    </div>
```

Kuvio 19: Pudotusvalikko asiakassivun perustiedoissa HTML-koodissa PrimeNG-kirjastoa käyttäen

TypeScriptin puolella arvot annetaan pudotusvalikoihin kuvion 20 osoittamalla tavalla.

```
const languages: Language[] = this.activatedRoute.snapshot.data['languages'];
const customerGroups: CustomerGroup[] = this.activatedRoute.snapshot.data['customerGroups'];
const paymentTermChoices: ParameterChoice[] = this.activatedRoute.snapshot.data['paymentTermChoices'];
const currencyChoices: ParameterChoice[] = this.activatedRoute.snapshot.data['currencyChoices'];
const vatCodes: ParameterChoice[] = this.activatedRoute.snapshot.data['vatCodes'];
const collectionTypes: ParameterChoice[] = this.activatedRoute.snapshot.data['collectionTypes'];
const chargeIntervals: ParameterChoice[] = this.activatedRoute.snapshot.data['chargeIntervals'];

languages.forEach((language) => {
    this.languageOptions.push({ label: language.language, value: language });
});
customerGroups.forEach((customerGroup) => {
    this.customerGroupOptions.push({ label: customerGroup.name, value: customerGroup });
});
```

Kuvio 20: Pudotusvalikko asiakassivun perustiedoissa TypeScript-koodissa

Asiakkaiden laskutustiedoista löytyy kenttiä, jotka ottavat syötteenä pelkkiä numeroita, jottei käyttäjä voi syöttää virheellisesti kirjaimia numeroiden sijaan. Näitä kenttiä on useita, mutta niiden ollessa samalla lailla toteutettuja, esiin nostetaan kuviossa 21 viivästyskorkokentästä esimerkkinä HTML:n puolelta vain numeroita salliva input. Format-määrityksen avulla määritetään desimaalien muotoilu.

```
<number-input class="fullWidth" formControlName="latePaymentInterestRate" format="1.0-2"></number-input>
```

Kuvio 21: Numeroinput-kentän luonti HTML-koodissa

Lisäksi laskutustietolomakkeista löytyy kalenterivalikko ennakkomaksupäivää varten. Se on toteutettu kuvion 22 osoittamalla tavalla, muutoin koodi vastaa aiemmin käsiteltyä.

```
<div class="ui-inputgroup">
    <p-calendar formControlName="advancePaymentDate" dateFormat="dd.mm.yy"
                [locale]="locale" [showIcon]="true" icon="far fa-calendar-alt" styleClass="fullWidth" name="advancePayme
                appendTo="body"
                [yearNavigator]="true" [monthNavigator]="true" yearRange="1980:2100" showButtonBar="true">
    </p-calendar>
</div>
```

Kuvio 22: Kalenterin määrittäminen HTML-koodissa PrimeNG-kirjastoa käyttäen

Vaikka kenttiä on laskutustietolomakkeessa runsaasti, ohjelmoinniltaan ne vastaavat jo aiemmin käsiteltyjä tapauksia. Kenttiä on kaksikymmentäkolme kappaletta ja näistä pakollisia ovat maksutapa, maksuehto, luottovastuu, laskutusmuoto, laskun rahoitus, valuutta, ALV-koodi, koontilaskukoodi, laskutusjako ja laskun kooste. Pakollisuudet on jälleen määritetty TypeScript-luokassa validaattoreita käyttäen, kuten aiemmissa esimerkeissä. Poikkeuksena tuoteryhmät-sivuun nähden on huomautuskenttä, joka on toteutettu textarea-tagilla HTML-puolella kuviossa 23 esitetyllä tavalla.

```
<textarea pInputTextarea
    class="fullWidth"
    [rows]="4"
    formControlName="comment"
    maxlength="4000"></textarea>
```

Kuvio 23: Tekstikentän toteutus HTML-koodissa asiakkaan laskutustiedoissa

Työmaa-lomake (kuvio 16) on jaettu kahteen osioon: Työmaan perustietoihin sekä yhteystietoihin. Työmaan perustiedoissa on tekstikentät työmaan nimelle sekä ulkoiselle viitteelle ja lisäksi yksi nappi tilausten tarkastusta varten, joka pitää sisällään boolean-arvon. Ainoastaan nimikenttä eroaa aiemmin käsitellyistä tekstikentistä siltä osin, että koodiin on lisätty tarkistus sitä varten, että onko käytetty nimi jo olemassa. Jos nimi on jo käytetty, ilmoitetaan käyttäjälle tästä. Validointi on toteutettu HTML-koodissa kuvion 24 ja TypeScript-koodissa kuvion 25 osoittamalla tavalla. Vaikuttaa siltä, että koodit ovat ristiriidassa ja tarkistus tapahtuu kahdessa eri paikassa, tämä selvitetään seuraavassa luvussa koodia katselmoidessa.

```
<div class="p-col-1" id="worksiteNameValidityIconContainer">
    <i class="fas fa-check-circle worksiteNameValidityIcon elementVerticalCenter" *ngIf="worksiteNameValid"></i>
    <i class="fas fa-exclamation-triangle worksiteNameValidityIcon elementVerticalCenter"
        pTooltip="{{ 'worksite.nameAlreadyInUse' | translate }}" tooltipPosition="top"
        *ngIf="!worksiteNameValid"></i>
</div>
```

Kuvio 24: Nimen olemassaolon tarkistus HTML-koodissa työmaat-lomakkeessa

```
hasDuplicateWorksiteNameForCustomer(): void {
    if (!this.worksiteForm.controls.name.value) {
        this.worksiteNameValid = false;
        return;
    }

    this.customerResource.hasDuplicateWorksiteNameForCustomer(this.worksiteForm.controls.id.value,
        this.worksiteForm.controls.name.value, this.customerId)
        .pipe(this.unsubscribeOnDestroy())
        .subscribe((results: boolean) => {
            this.worksiteNameValid = !results;
        });
}
```

Kuvio 25: Nimen olemassaolon tarkistus TypeScript-koodissa työmaat-lomakkeessa

Työmaat-sivun yhteystietolomakkeessa kuviossa 16 annetaan tarkempi nimi kohteelle sekä osoite, jotka ovat pakollisia tietoja, tosin jostain syystä pakollisuus puuttuu postinumeron ja paikkakunnan osalta, nämä kentät voi jättää tyhjäksi. Yhdellä työmaalla voi olla useita yhteystietoja, jolloin lomakkeen koko kasvaa dynaamisesti ja jokaiselle yhteystiedolle voi lisätä työmaamestarin, joka kasvattaa lomakkeen kokoa edelleen tuomalla näkyviin uusia tiedonsyöttökenttiä. Lomakkeiden lisääminen tapahtuu työmaamestaria esimerkkinä käyttäen HTML-koodissa kuvion 26 osoittamalla tavalla ja TypeScript-koodissa kuvion 27 osoittamalla tavalla.

```
<button pButton type="button"
    class="ui-button-raised ui-button-rounded ui-button-secondary"
    icon="pi pi-plus" label="{{ 'worksite.addSiteManager' | translate }}"
    (click)="addContactPersonToDestination(destination)"></button>
```

Kuvio 26: Työmaamestari-lomakkeen lisääminen HTML-koodissa työmaat-sivulla

```
addContactPersonToDestination(destinationControl: FormGroup): void {
    (destinationControl.controls.contactPersons as FormArray).push(this.newContactPerson());
}
```

Kuvio 27: Työmaamestari-lomakkeen lisääminen TypeScript-koodissa työmaat-sivulla

Kuviossa 17 havainnollistetussa kortit-lomakkeessa voidaan määrittää asiakkaalle esimerkiksi polttoaine- tai asiakaskortti. Lomake on jaettu kolmeen lohkoon: Kortin perustiedot, voimassa asti ja kommentti.

Kortin perustiedoissa määritetään kortin numero tekstinsyöttökentässä sekä tyyppi pudotusvalikossa, jotka ovat pakollisia tietoja. Lisäksi voidaan määrittää kortille haltija tekstikentän avulla ja auto-complete-kentän avulla korttiin sidottu ajoneuvo. Vastaavien kenttien koodi on jo käsitelty, joten niitä ei käydä läpi sen tarkemmin koodin esittelyssä.

Voimassa asti-lomake pitää sisällään kalenterikentän, joka on myös käsitelty aiemmin tässä opinnäytetyössä. Kommenttikenttä on toteutettu HTML-koodissa textarea-tagilla, joka havainnollistetaan kuviossa 28.

```
<p-panel header="{{ 'card.comment' | translate }}">
    <div class="p-grid">
        <div class="p-col">
            <textarea pInputTextarea formControlName="comment" class="fullWidth" [rows]="3">
            </textarea>
        </div>
    </div>
</p-panel>
```

Kuvio 28: Tekstikentän toteutus HTML-koodissa kortit-lomakkeessa

TypeScript-koodissa kommentille on asetettu validaattori, joka rajoittaa kommentin pituuden maksimissaan neljään tuhanteen merkkiin, kuten voidaan havaita kuviossa 29. HTML-koodi on jo aiemmin käsitelty kuviossa 23, mutta koska validointi on toteutettu eri tavalla, nostettiin se uudelleen esiin.

```
this.cardEditForm = this.formBuilder.group({
    id: [undefined],
    cardNumber: ['', Validators.required],
    cardType: [undefined, Validators.required],
    validTo: [undefined],
    cardHolder: [undefined],
    vehicle: [undefined],
    comment: [undefined, Validators.maxLength(4000)]
});
```

Kuvio 29: Tekstikentän merkkimäärän rajoittaminen TypeScript-koodissa

7 Koodikatselmointi lomakkeiden osalta

Koodikatselmointi suoritettiin valikoitujen luokkien läpikäyntinä Leanvay Oy:n toimitiloissa tutkijan toimesta. Katselmoitavaksi otettiin tuoteryhmä-sivun koodit sekä asiakkaat-sivun koodit. Katselmointi suoritettiin seuraaviin luokkiin, sillä ne liittyvät olennaisesti reaktiivisien lomakkeiden käyttöön. On tosin mainittava, että TypeScript-koodien puolella kutsutaan metodeja myös ulkoisista luokista, mutta näitä ei otettu katselmoinnin osaksi sillä se olisi kasvattanut opinnäytetyön kokoa huomattavasti. Tuoteryhmä-sivuun liittyviä luokkia oli kaksi:

- product-group-edit.component.html (Tuoteryhmät-sivun ulkoasu)
- product-group-edit.component.ts (Tuoteryhmät-sivun toiminnallisuus)

Asiakkaat-sivulla luokkia oli kuusi, kaksi jokaista käsiteltävää lomakesivua varten:

- customer-basic-data.component.html (Asiakkaan perustietosivun ulkoasu)

- customer-basic-data.component.ts (Asiakkaan perustietosivun toiminnallisuus)
- card-edit.component.html (Korttisivun ulkoasu)
- card-edit.component.ts (Korttisivun toiminnallisuus)
- worksite-edit.component.html (Työmaasivun ulkoasu)
- worksite-edit.component.ts (Työmaasivun toiminnallisuus)

Valitut luokat on nimetty asianmukaisesti, jotta on helppo ymmärtää mikä niiden tehtävä on. Angular-ohjelmointi tyylioppaan (Google b, 2020) mukaan nimeämisien yhtenäisyys on tärkeää niin tiimille, projektille kuin yritykselle, parantaen valtavasti tehokkuutta.

Poikkeuksena customer-basic-data.component.ts ja customer-basic-data.component.html, joista kerrotaan tarkemmin kappaleessa 7.2.1.

Katselmoinnissa pyrittiin löytämään mahdolliset hyvien ohjelmointitapojen vastaiset ohjelmointitavat ja saamaan käsitys koodin siisteydestä. Sivut on testattu toimiviksi sekä Unit-testeillä että e2e-testeillä, jonka lisäksi ne ovat jo tuotantokäytössä, joten niiden voidaan olettaa toimivan, eikä koodissa pitäisi olla kriittisiä virheitä.

Katselmointi suoritettiin siten, että tutkija kävi aiemmin mainituissa luokissa koodin alusta loppuun rivi riviltä läpi äärimmäistä tarkkuutta noudattaen, pitäen silmällä mahdollisia parannuskohtia muotoilun ja logiikan suhteen. Löydetyt havainnot kirjattiin ylös sellaisen löytyessä ja tämän jälkeen jatkettiin rivi riviltä etenemistä. Katselmointi pyrittiin tekemään rauhallisesti, jotta kaikki mahdolliset epäkohdat havaitaan.

Kun luokat oli käyty läpi ja tulokset kirjattu, ne lisättiin tutkimukseen sanallisesti ja kuvakaappauksien kera.

7.1 Tuoteryhmät-sivun koodin katselmoinnin tulokset

Tässä kappaleessa käydään läpi koodikatselmoinnissa esiin nousseita mahdollisia parannuskohteita tuoteryhmät-sivun katselmointiin valituissa koodiluokissa product-group-edit.component.html ja product-group-edit.component.ts.

Katselmoitaessa luokkaa product-group-edit.component.html, joka vastaa tuoteryhmät-sivun ulkonäöstä, löytyi neljä parannusehdotusta koodille. Ensimmäiseksi, koodin riviltä 19 kuviossa 30 alkava main-lohko on toiminnan kannalta turha. Lomakkeen voi aloittaa pelkällä form-tagilla.

```
19  <main id="productGroupEditPanel" [ngClass]="{ 'slit-in-horizontal': productGroupEditFormAnimation }">
20      <form [formGroup]="productGroupForm">
21  >       <div class="p-grid">…
102         </div>
103     </form>
104 </main>
```

Kuvio 30: Turha main-lohko luokassa product-group-edit.component.html

Parempi toteutus voidaan nähdä kuviossa 31 luokassa product-edit.component.hml, joka nostettiin muualta koodista esimerkiksi osoittamaan parempi käytäntö.

```
19  <form [formGroup]="productEditForm" [ngClass]="{ 'slit-in-horizontal': productEditFormAnimation }">
20
21  >   <div class="p-grid">…
235     </div>
236 </form>
```

Kuvio 31: Esimerkki paremmasta toteutuksesta HTML-rakenteessa luokasta product-edit.component.hml

Tämän lisäksi löytyi kolme muuta parannuskohtaa, joissa kaikissa toistuu sama asia. Elementeille on turhaan määritetty nimet, kuten voidaan havaita kuviosta 32. Nämä ovat tarpeettomia koodin toiminnan kannalta, joskin ne helpottavat e2e-testien kirjoittamista, mutta siinäkään ne eivät ole välttämättömiä. Elementtien toiminnallisuus selviää hyvän nimeämisen ansiosta jo tagista ja formControlName:sta. Samassa kuviossa voidaan havaita formControlName, joka hoitaa datan sitomisen oikeaan paikkaan.

```
<p-toggleButton
    onLabel="{{ 'productGroups.combineProductsForExternalInvoice' | translate }}"
    class="optionOk"
    offLabel="{{ 'productGroups.dontCombineProductsForExternalInvoice' | translate }}"
    formControlName="combineProductsForExternalInvoice" styleClass="fullWidth"
    name="combineProductsForExternalInvoiceToggleButton">
</p-toggleButton>
<p-toggleButton onLabel="{{ 'productGroups.discountAllowed' | translate }}"
    class="optionOk"
    offLabel="{{ 'productGroups.discountNotAllowed' | translate }}"
    formControlName="discountAllowed" styleClass="fullWidth"
    name="discountAllowedToggleButton">
</p-toggleButton>
<div class="p-col-8">
    <input type="text" pInputText class="fullWidth" formControlName="name"
        name="productGroupName" />
</div>
```

Kuvio 32: Turhat nimien määritykset luokassa product-group-edit.component.html

Katselmoitaessa TypeScript-luokkaa product-group-edit.component.ts, löytyi kaksi käyttämätöntä importia (kuvio 33), jotka voi poistaa koodista. Muuta huomautettavaa ei luokasta löytynyt.

```
5   import { ConfirmationService, SelectItem } from 'primeng/api';
6   import { FormGroup, FormBuilder, Validators, FormControl } from '@angular/forms';
7   import { SubscriptionHandler } from '@shared/subscriptionhandler/subscription-handler';
8   import { ProductGroupResource } from '@shared/product-group/product-group.resource';
9   import { ProductGroup } from '@shared/product-group/product-group';
10  import { CostCenterResource } from '@shared/costcenter/cost-center.resource';
11  import { CostCenter } from '@shared/costcenter/cost-center';
12  import { costCenters } from '@core/mocks/json/costcenters.json';
13  import { CostCenterType } from '@shared/costcenter/cost-center-type';
```

Kuvio 33: Turhat importit luokassa product-group-edit.component.ts

7.2 Asiakkaat-sivun koodin katselmoinnin tulokset

Asiakkaat-sivun katselmointitulokset käydään läpi tässä luvussa alasivuittain, jotka olivat perustiedot, työmaat ja kortit.

7.2.1 Perustiedot-lomake

Katselmoitaessa luokkaa customer-basic-data.component.html ensimmäinen poikkeavuus on luokan nimessä, jossa yleisen muotoilun mukaisesti tulisi olla sana ”edit”, oikea muotoilu olisi customer-basic-data-edit.component.html, kuten muissa vastaavissa luokissa. Seuraavaksi esiin nousee kuviossa 34 näkyvä HTML-rakenne, jonka voisi toteuttaa samoin kuin kuviossa 31 on havainnollistettu, eli lohko voisi alkaa suoraan form-tagilla, toisin sanoen kaksi ensimmäistä div-lohkoa eivät ole pakollisia. Tässä tapauksessa muutosta ei voida kuitenkaan tehdä suoraan, sillä luokkaan on sidottu CSS-tiedosto, joka käyttää annettua id:tä.

```
18  <div id="customerBasicDataEdit" class="p-grid">
19      <div class="p-col">
20  >       <form [ngClass]="{ 'slit-in-horizontal': customerBasicDataEditFormAnimation }" [formGroup]="customerBasicDataEditForm">
480         </form>
481     </div>
482 </div>
```

Kuvio 34: HTML-rakenne luokassa customer-basic-data.component.html

Luokasta löytyi myös yksi korjauskehotus asiakasnumeron syöttöön liittyen, sillä syöttömuodoksi oli asetettu teksti, eikä numero, kuten kuviosta 35 voidaan havainnoida. Syöttövirheiden minimoimiseksi olisi syytä käyttää number-inputia, joka ei käyttöliittymässä hyväksy kirjaimia.

```
<div class="p-col-8 textToRight">
    <input type="text" pInputText class="fullWidth" formControlName="customerNumber" maxlength="255" />
</div>
```

Kuvio 35: Tekstinsyöttökenttä HTML-koodissa, jonka tulisi olla numeronsyöttökenttä

Kolmessatoista kohdassa on toteutettu syötteen maksimipituuden rajoitus HTML-koodin puolella maxlength-tagilla (esimerkki kuviossa 36), vaikka reaktiivisia lomakkeita käytettäessä voidaan maksimipituus asettaa TypeScript-koodin puolella aiemmin esiteltyjä validaattoreita käyttäen, joka helpottaa unit-testausta ja toteuttaa muutenkin hyvin koodaustapojen mukaisesti yhtenäisyyttä, sillä useimmat validoinnit on toteutettu TypeScript-koodin puolella.

```
<input type="text" pInputText class="fullWidth" formControlName="name" maxlength="255" />
```

Kuvio 36: Syötteen maksimipituuden rajaaminen HTML-koodissa

Viidessä kohdassa on määritetty kuviota 32 vastaavalla tavalla turhaan nimet käsiteltävässä luokassa, nämä määritykset voisi poistaa. Muita ongelmia HTML-koodista ei löytynyt.

Perustiedot-lomakkeen TypeScript-koodissa luokan nimen (customer-basic-data.component.ts) tulisi sisältää ”edit”-sana kuten edeltävässä HTML-luokassakin ja kymmenellä rivillä koodi venyy vaakasuunnassa turhan pitkäksi kuten kuvion 37 esimerkissä, rivinvaihdot helpottaisivat lukemista. Muutoin luokasta ei löytynyt parannettavaa.

```
174        if (key === 'visitAddress.postOffice' && !this.customerBasicDataEditForm.get('postalAddress.postOffice.postalCode')!.value && !this.customerBasicDataEditForm.get('postalA
```

Kuvio 37: Rivinvaihtoa kaipaava koodirivi

7.2.2 Työmaat-lomake

Luokkaa worksite-edit.component.html katselmoitaessa löytyi pari seikkaa, joita voisi parantaa. Ensimmäisenä toolbar-koodin muotoilu kuviossa 38 voitaisiin yhtenäistää käyttäen uutta riviä korostetussa kohdassa, koska muotoilu on erilaista muissa luokissa.

```
1  <p-toolbar>
2      <div class="ui-toolbar-group-left">
3          <edit-default-buttons
4              (onSaveButtonClickAction)="saveWorksite()"
5              (onRemoveButtonClickAction)="remove()" [removeButtonDisabled]="!worksiteForm.valid"
6              [hideNewButton]="true"
7              [hideCopyButton]="true">
8          </edit-default-buttons>
9      </div>
10 </p-toolbar>
```

Kuvio 38: Toolbar-koodi worksite-edit.component.html -luokassa

Eroavaisuus ei ole suuri, kuten voidaan havaita esimerkiksi nostetusta kortit-luokan toolbar-koodista kuviossa 39, mutta luettavuus on parempi. Tämä on kuitenkin makuasia, nappien asettelu vierekkäin toimii luettavuuden osalta myös hyvin, mutta olisi syytä käyttää vain toista muotoilua, ei molempia.

```
<p-toolbar>
    <div class="ui-toolbar-group-left">
        <edit-default-buttons
            newButtonTextTranslationKey="card.new"
            (onNewButtonClickAction)="newCard()"
            [newButtonDisabled]="!cardEditForm.controls.id.value"
            (onSaveButtonClickAction)="saveCard()"
            [saveButtonDisabled]="!cardEditForm.valid"
            (onRemoveButtonClickAction)="remove()"
            [removeButtonDisabled]="!cardEditForm.valid"
            [hideCopyButton]="true">
        </edit-default-buttons>
    </div>
</p-toolbar>
```

Kuvio 39: Toolbar-koodi card-edit.component.html -luokassa

Lisäksi toistuu aiemmin katselmoinnissa havaittu rakenteellinen tyyli, jossa form-tag on haudattu toisen lohkon sisään, kuten havainnollistetaan kuviossa 40. Kyseisessä luokassa on kuitenkin käytössä CSS-tiedosto, joka käyttää annettua id:tä, joten rakenteen muuttaminen vaatii enemmän vaivannäköä, mikäli se on mahdollista.

```
<div id="worksiteEditPanel">
    <form [formGroup]="worksiteForm">
        <div class="p-grid">…
        </div>
    </form>
</div>
```

Kuvio 40: HTML-rakenne card-edit.component.html -luokassa

Katselmoitaessa TypeScript-luokkaa worksite-edit.component.ts löytyi kaksi riviä, joiden pituus oli turhan pitkä ja rivinvaihto olisi ollut luettavuuden vuoksi paikallaan. Lisäksi yksi luokan sisäinen parametri oli määritetty julkiseksi, joka aiheutti sekaannusta siitä, käytetäänkö sitä myös luokan ulkopuolella. Kyseinen parametri on havainnollistettu kuviossa 41 korostetusti.

```
    private customerId: string = '';
    worksiteForm: FormGroup = <FormGroup>{};
    cachedConfirmTranslations: any = {};
    worksiteNameValid: boolean = false;
    searchPostOfficeResults: PostOffice[] = [];
    searchContactPersonResults: ContactPerson[] = [];
```

Kuvio 41: Julkinen parametri luokassa worksite-edit.component.ts

7.2.3 Kortit-lomake

Ainoa katselmoinnissa (HTML) esiin noussut seikka luokassa card-edit.component.html oli ei-välttämätön nimen antaminen autoComplete-kentälle, vastaavalla tavalla kuin kuviossa 32. Luokassa card-edit.component.ts TypeScript-koodissa havaittiin yksi turha import sekä yksi liian pitkä koodirivi, joka kaipaa rivinvaihtoa luettavuuden parantamiseksi.

8 Tutkimuksellisen kehittämistyön tulokset

Tässä kappaleessa esitellään tutkimuksesta saadut tulokset haastattelun, havainnoinnin ja koodikatselmoinnin osalta.

8.1 Haastattelu

Tutkimukseen kuuluneessa puolistrukturoidussa teemahaastattelussa, joka järjestettiin yrityksen tiloissa 1.6.2020, saatiin arvokasta taustatietoa järjestelmästä, sen historiasta ja nykytilasta järjestelmän kehittäjältä ja Leanvay Oy:n osakkaalta, Chao Fengiltä. Haastattelussa selvitettiin haastateltavan taustoja ja saatiin selville, että Feng on koulutukseltaan tekniikan kandidaatti ja hänellä on alan työkokemusta kuuden vuoden ajalta. Haastateltavan taustatiedot koettiin tarpeellisiksi selvittää, jotta voitiin varmistua henkilön asiantuntijuuden tasosta tutkimukseen liittyen.

Haastattelussa selvitettiin myös LogiPlanin käyttötarkoitusta, asiakasmääriä ja historiaa. Tarkempaa tietoa näistä on kappaleessa 2.1. Kyseisessä kappaleessa kerrotaan myös syistä, miksi on päädytty vaihtamaan sovelluskehystä. Kappaleessa 5 on nostettu esiin haastattelusta saatuja vastauksia liittyen ohjelmakoodin parantamiseen ja mitä etuja tällä saavutetaan. Haastattelusta ja epävirallisista keskusteluista saatu tieto yrityksen sisäisistä ohjelmointikäytännöistä auttoi tutkimuksen tekemisessä, sillä saatiin tietoa siitä, minkälaista koodia halutaan yrityksessä tuottaa.

Edellä mainittujen haastattelutuloksien perusteella pystyttiin suorittamaan koodikatselmointi mahdollisimman tehokkaasti, sillä tunnistettiin hyödyt koodin parantamisen takana sekä luokat, joista koodia tulisi tutkia. Haastattelukysymykset löytyvät kokonaisuudessaan liitteestä 2.

8.2 Havainnointi

Osallistuvan havainnoinnin osalta, varsinaista päivitystyötä Wicketistä Angulariin tehtäessä, saatiin tuloksia siitä, miten koodista saadaan toimivaa. Jotta havainnointi oli mahdollisimman tehokasta, tutkija työskenteli tutkimusaikana enimmäkseen LogiPlan-järjestelmän front end -

koodin parissa. Havainnoinnin edetessä kävi selväksi, että käytetyt ohjelmointikäytännöt Leanvay Oy:ssä olivat lähtökohtaisesti hyvällä tasolla reaktiivisten lomakkeiden käytön osalta. Hyvät ohjelmointitavat on otettu huomioon hyvällä nimeämisellä, turhan toiston välttämisellä, selkeällä logiikalla ja asianmukaisella muotoilulla. Havainnointiin pohjautuen tuotettu koodi Leanvay Oy:ssä on kurinalaisesti toteutettua ja havaitut puutteet hyviin ohjelmointikäytäntöihin liittyen olivat hyvin vähäisiä.

Havainnoinnin aikana, useiden eri luokkien kohdalla, hyödynnettiin Angular-ohjelmointi tyyliopasta (Google b, 2020) josta löytyy suosituksia koodin kirjoittamiseen Angularia käytettäessä. Useiden eri luokkien kohdalla työskennellessä pyrittiin pitämään huoli siitä, että suosituksia on noudatettu. Havainnoinnin perusteella näin on toimittu kautta linjan ja esimerkiksi seuraavista tyylioppaan (Google b, 2020) suosittelemista tyyleistä on pidetty kiinni:

-Pienet funktiot (helpompi testata ja lukea)

-Yhtenäinen nimeäminen (auttavat löytämään etsityn koodin nopeammin)

-Testiluokkien nimeämiskäytäntö (helppo tunnistaa testeiksi)

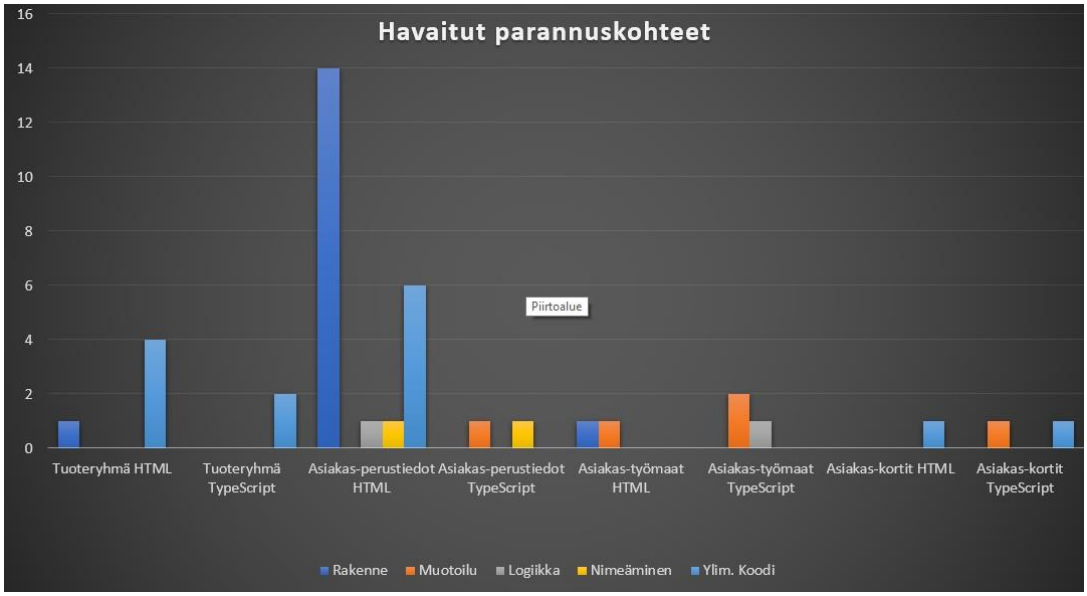
-Toiston välttäminen (vähentää koodin määrää)

-Projektin kansiorakenne (helpompi kehittää projektin kasvaessa)

Testit lomakkeisiin liittyen Leanvay Oy:ssä on toteutettu asianmukaisella tavalla ja lomakkeiden toimivuus vahvistettu ensin testaamalla ne sekä Unit-testeillä sekä e2e-testeillä front end -puolella. Back end -ohjelmoinnin valmistuttua JUnit- ja integraatiotesteineen luodut sivut viedään demoympäristöön, jossa niiden toiminnallisuus testataan ihmisten toimesta, ennen kuin tuotettu koodi viedään tuotantoympäristöön asiakkaiden käytettäväksi.

8.3 Koodikatselmointi

Osana tutkimusta suoritetun koodikatselmoinnin havaituista parannusehdotuksista suurin osa löytyi koodin HTML-puolelta, kuten voidaan havaita koodikatselmoinnin tuloksia kuvaavasta kuviosta 42. Kaaviossa on merkitty havaitut parannuskohteet määrittäin, eli sama asia toistuessaan useammin kuin kerran on merkitty kaavioon indikoimaan parannuskohteiden määrää luokittain. Parannuskohteet on eritelty sanallisesti kappaleessa 7.



Kuvio 42: Kaavio koodikatselmoinnissa ilmi tulleista parannusehdotuksista

Katselmoinnissa havaituista huomautuksen aiheista yksikään ei ollut kriittinen, vaan parannusehdotukset koskivat lähinnä rakenteellisuutta, muotoilua ja ylimääräisen koodin siivoamista. Koodi oli täysin toimivaa, mutta etenkin isossa projektissa jokainen koodirivi, josta päästään eroon on kuitenkin etu, joten tutkimuksen tuloksia voidaan pitää hyödyllisinä.

Uudenlaisia tapoja toteuttaa koodia reaktiivisiin lomakkeisiin liittyen ei tutkimuksessa löydetty, vaan tulokset osoittavat, että käytössä olevat ohjelmointikäytännöt noudattavat hyviä käytäntöjä yleisesti ottaen, pois lukien havaitut pienet parannuskohteet. Kyseiset parannuskohteet eivät vaadi suuria muutoksia ohjelmointikäytäntöihin, korkeintaan hieman lisätarkkuutta yhtenäisyyden ylläpitämiseksi katselmointivaiheessa, joka toteutetaan aina uutta koodia julkaistaessa, kehittäjän kollegan toimesta.

9 Kehitysehdotukset

Reaktiivisiin lomakkeisiin liittyvät HTML-koodit olisi hyvä yhtenäistää, sillä näissä havaittiin tutkimuksessa eniten poikkeavuuksia toisistaan. HTML-koodi oli silti selkeää, toimivaa ja helposti luettavaa, mutta jo pienellä siivoamisella saadaan aikaan hyötyä isoissa projekteissa, kuten LogiPlan. Vaikka siivottaisiin vain pari riviä koodia per luokka, vaikutus kertautuu äkkiä, mikäli luokkia on satoja tai tuhansia. Lisäksi koodin luettavuus paranee entisestään, mikäli se on mahdollisimman yhtenäistä eri luokissa. Tämä helpottaa esimerkiksi uuden kehittäjän projektiin tuomista ja mahdollisten virheiden löytämistä. Varsinaiseen koodin tuottamisprosessiin ei tässä tutkimuksessa löydetty syitä tehdä muutoksia, mutta katselmointivaiheessa voitaisiin

hyödyntää esimerkiksi yhteistä tarkistuslistaa, jotta poikkeavuudet ja virheet havaittaisiin suurimmalla todennäköisyydellä.

Yleinen vastaan tuleva rakenteellinen seikka oli HTML-rakenne, joka alkoi monesti joko maintagilla tai div:inä. Osassa tapauksista tämä oli turha rakenne, mutta osassa tarvittiin lomakkeelle parent-lohko käytettävän CSS-tiedoston vuoksi, pääasiassa jotta saatiin sivusta skrollaava isoissa lomakkeissa. Tämän rakenteen voisi mahdollisesti miettiä uusiksi, jotta voidaan käyttää form-tagia aloittavana myös enemmän tilaa vievissä lomakkeissa, yhtenäisyyden parantamiseksi.

HTML-puolella ilmaantui myös validointeja, jotka olisi parempi toteuttaa TypeScript-koodin puolella, jota suositellaankin jatkossa tehtäväksi tämän opinnäytetyön perusteella. Täten saadaan koodista edelleen yhtenäisempää ja nopeammin testattavaa, sillä testit voi ajaa Unit-testeillä eikä hitaammilla e2e-testeillä.

Koodissa esiintyvien pienimuotoisten muotoiluseikkojen, kuten turhien välilyöntien, liian pitkien rivien ja käyttämättömien importtien osalta suositetaan käyttöönotettavaksi automaattista muotoilu-pluginia (esimerkiksi Prettier) Visual Studioon, jota yrityksessä käytetään front end- kehityksessä. Muotoilutyökalun voi valjastaa toimimaan esimerkiksi siten, että se aktivoituu ja siivoaa koodiin luokkaa tallennettaessa. Ohjelmoijien näkemys ja mieltymykset tosin voivat olla eroavaisia koodin asettelun osalta, joten olisi tärkeää yhteisesti miettiä ja päättää minkälaista muotoilua ja työkalua halutaan käyttää, jotta kaikki ovat tyytyväisiä. Pienienkin muotoiluerojen poistaminen parantaa koodin luettavuutta entisestään, vaikka se on jo nyt hyvällä tasolla.

10 Yhteenveto ja johtopäätökset

Tämä opinnäytetyö keskittyi Angularin reaktiivisiin lomakkeisiin liittyvän ohjelmakoodin tutkimiseen ja ohjelmointikäytäntöjen mahdolliseen parantamiseen. Helposti luettava koodi helpottaa ohjelmistoyrityksien toimintaa vähentäen koodin lukemiseen, testaamiseen ja korjaamiseen käytettävää aikaa. Tästä syystä pyrittiin varmistamaan ja parantamaan ohjelmointikäytäntöjä käyttäen laadullisina menetelminä tutkijan omaa havainnointia, koodikatselmointia ja järjestelmän kehittäjän haastattelua.

Tutkimuksen tulokset osoittavat, että Angularin reaktiivisiin lomakkeisiin liittyvä koodi oli asiakasyrityksessä hyvin toteutettua, hyvät ohjelmointikäytännöt on huomioitu eikä kriittisiä parannusehdotuksia löytynyt. Pieniä parannuskohteita tutkimuksessa onnistuttiin löytämään ja näistä raportoitiin asiakasyritykselle, jossa otettiin tutkimustulokset positiivisesti vastaan.

Tutkimuksessa esiin nousseista ohjelmointikäytäntöihin liittyvistä seikoista merkittävimpinä voidaan pitää HTML-koodin puolella olleita parannuskohteita, joiden yhtenäistäminen parantaisi jo entisestään hyvällä tasolla olevaa ohjelmakoodia luettavuuden osalta. Ohjelmointikäytäntöjen osalta ei ole tarvetta suurille muutoksille, mutta toimintaa voidaan tehostaa entisestään kehitysehdotuksissa mainituin tavoin.

Ohjelmakoodi on yrityksessä puhdasta ja helposti luettavaa. Esitettyjen parannusehdotusten, kuten muotoiluseikkoihin puuttumisen automatisoidulla muotoilutyökalulla ohjelmointiympäristössä, avulla saadaan pidettyä koodi jatkossakin yhtä siistinä, vähentäen mahdollisuuksia inhimillisten virheiden syntyyn. Pieni lisätarkkuus ei olisi pahitteeksi koodia tuotettaessa ja katselmoidessa, jotta saadaan pienetkin virheet karsittua pois.

Hyvien ohjelmointitapojen osalta tutkimus osoittaa, että niiden noudattaminen on hyvällä mallilla reaktiivisiin lomakkeisiin liittyvässä koodissa. Ylimääräinen koodi on minimaalista ja nimeämiset on toteutettu selkeästi sekä luokkien, muuttujien ja funktioiden osalta. Muotoilu noudattaa myös hyvin hyviä ohjelmointitapoja, joka edesauttaa hyvän nimeämisen ohella koodin tulkitsemista.

Reliabiliteetti tutkimuksessa pyrittiin varmistamaan suorittamalla koodikatselmointi mahdollisimman huolellisesti ja vahvistamalla katselmoinnin löydökset kokeneemman järjestelmäkehittäjän kanssa, joka kävi myös tutkitut luokat läpi ja vahvisti löydetyt parannuskohteet oleellisiksi tutkimuksen kannalta. Hän myös varmisti, ettei tutkijalta jäänyt parannuskohteita löytämättä.

Tutkimuksen validiteetti ilmeni havainnoinnin osalta siten, että tutkija keskittyi opinnäytetyön aiheen mukaisesti Angularin reaktiivisia lomakkeita toteuttavan koodin työstämiseen ja tutkimiseen tutkimuksen osalta. Validiteettia vahvistettiin esittelemällä tutkimuksessa esille nousseet parannuskohteet järjestelmän kokeneelle kehittäjälle, joka vahvisti tulokset.

Opinnäytetyön tekeminen toimi opettavaisena kokemuksena, sillä tehty tutkimus auttoi kehittämään tutkijan taitoa tunnistaa hyviä ohjelmointikäytäntöjä koodissa ja noudattaa niitä myös itse koodia tuotettaessa.

Lähteet

Painetut

Anaya, M. 2018. Clean Code in Python. Birmingham, Yhdistynyt kuningaskunta: Packt Publishing.

Coremans, C. 2015. HTML: A Beginner's Tutorial. Quebec, Kanada: Brainy Software.

Eskola, J. & Suoranta, J. 1998. Johdatus laadulliseen tutkimukseen. Tampere: Vastapaino.

Gupta, G. 2013. Mastering HTML5 Forms. Birmingham, Yhdistynyt kuningaskunta: Packt Publishing.

Hirsjärvi, S. & Hurme, H. 2015. Tutkimushaastattelu: Teemahaastattelun teoria ja käytäntö. Helsinki: Gaudeamus.

Leavitt, A. 2018. Becoming a software engineer. New York, Yhdysvallat: The Rosen Publishing Group.

Martin, R. 2009. Clean code: a handbook of agile software craftsmanship. New jersey, Yhdysvallat: Prentice Hall.

McFedries, P. 2018. Web Coding & Development All-in-one. New Jersey, Yhdysvallat: John Wiley & Sons.

Mohammed, Z. 2019. Angular Projects: Build Nine Real-World Applications from Scratch Using Angular 8 and TypeScript. Yhdistynyt kuningaskunta: Packt Publishing.

Padolsey, J. 2020. Clean Code in JavaScript. Birmingham, Yhdistynyt kuningaskunta: Packt Publishing.

Rawat, S. 2014. Getting Started with Review Board. Birmingham, Yhdistynyt kuningaskunta: Packt Publishing.

Shofner, M. 2018. Web developer. New York, Yhdysvallat: The Rosen Publishing Group.

Vilkka, H. 2015. Tutki ja kehitä. 4.Painos. Jyväskylä: PS-kustannus.

Westfall, L. 2009. The certified software quality engineer handbook. Wisconsin, Yhdysvallat: ASQ Quality Press.

Zhu, Y. 2017. Failure-Modes-Based Software Reading. Cham, Sveitsi: SpringerBriefs in Computer Science.

Sähköiset

Baker, M. 2009. What is a Software Framework? And why should you like 'em? Viitattu: 23.4.2020. https://www.cimetrix.com/blog/bid/22339/what-is-a-software-framework-and-why-should-you-like-em

Bodner, H. 2018. 4 Types Of Code Reviews Any Professional Developer Should Know About. Viitattu: 25.6.2019. https://dzone.com/articles/4-types-of-code-reviews-any-professional-developer

Fluin, S. 2020. Version 9 of Angular Now Available — Project Ivy has arrived! Viitattu: 6.5.2020. https://blog.angular.io/version-9-of-angular-now-available-project-ivy-has-arrived-23c97b63cfa3

Google a. 2020. Introduction to Angular concepts. Viitattu: 20.4.2020. https://angular.io/guide/architecture#introduction-to-angular-concepts

Google b. 2020. Angular coding style guide. Viitattu: 16.6.2020. https://angular.io/guide/styleguide

Leanvay. 2020. Tuote. Viitattu: 21.4.2020. https://leanvay.com/#features

Pat research. 2020. Apache Wicket. Viitattu: 7.5.2020. https://www.predictiveanalyticstoday.com/apache-wicket/

Ryan. 2019. What about Angular? Angular JS history through the years (2009-2019). Viitattu: 6.5.2020. https://www.ryadel.com/en/angular-angularjs-history-through-years-2009-2019/

Saaranen-Kauppinen, A. & Puusniekka, A. 2006. KvaliMOTV - Menetelmäopetuksen tietovaranto: Validiteetti. Viitattu: 11.6.2020. https://www.fsd.tuni.fi/menetelmaopetus/kvali/L3_3_1.html

Sanyal, N. 2019. Know Everything About Angular 8 Template Driven Forms. Viitattu: 22.4.2020. https://dev.to/nileshsanyal/know-everything-about-angular-8-template-driven-forms-4gof

Shah, J. 2019. Reactive Forms in Angular. Viitattu: 21.4.2020. https://medium.com/@jinalshah999/reactive-forms-in-angular-a46af57c5f36

Thomerson, J. 2012. Tutorial – Apache Wicket: The Fun Web Framework. Viitattu: 4.5.2020. https://jaxenter.com/tutorial-apache-wicket-the-fun-web-framework-104503.html

Tilastokeskus. Validiteetti. Viitattu 11.6.2020. https://www.stat.fi/meta/kas/validiteetti.html

Vilkka, H. 2006. Tutki ja havannoi.pdf. Luettu 29.4.2020. http://hanna.vilkka.fi/wpcontent/uploads/2014/02/Tutki-ja-havainnoi.pdf

Vuorinen, C. 2014. What is Clean Code and why should you care? Viitattu: 5.5.2020. https://cvuorinen.net/2014/04/what-is-clean-code-and-why-should-you-care/

Wozniewicz, B. 2019. The Difference Between a Framework and a Library. Viitattu: 30.4.2020. https://www.freecodecamp.org/news/the-difference-between-a-framework-and-a-library-bd133054023f/

Julkaisemattomat

Feng, C. 2020. Järjestelmäkehittäjän haastattelu LogiPlan-toiminnanohjausjärjestelmästä. 06.06.2020. Leanvay Oy. Vantaa

Kuviot

Liitteet

Liite 1: Listaus LogiPlan-toiminnanohjausjärjestelmän lomakkeita sisältävistä sivuista

- Tilausten hallinta
    - -Tilaukset
    - -Työmaatilaukset
    - -Päiväajot
    - -Maksutilaukset
- Kuljetussuunnittelu
- Tolppajono
    - -Autojen hallinta
    - -Hallinta
    - -Jonohistoria
    - -Kahvilanäkymä
- Laskutus
    - -Esilaskutus
    - -Hyväksytyt laskut
    - -Laskulistaus
    - -Esitilitys
    - -Hyväksytyt tilitykset
    - -Polttoaineet
    - -Vuokrat ja maksut
- Seuranta
    - -Ajoneuvoseuranta
    - -Tilausseuranta
    - -Työntekijätapahtumat
    - -Työajat
    - -Korjauspyynnöt
    - -Viestit
- Varastojen hallinta
- Raportit
- Hinnastot
    - -Myyntihinnastot
    - -Myyntisopimukset
    - -Ostohinnastot
    - -Ostosopimukset
    - -Maksuhinnastot
    - -Alueet
    - -Sopimusryhmät
- Perusrekisteri

-Asiakkaat
-Asiakasluokat
-Alihankkijat
-Ajoneuvot
-Ajoneuvojen lisävarusteet
-Tuotteet
-Tuoteryhmät
-Tekstipohjat
-Kohteet

- Asetukset
  -Työpoikkeamatyypit
  -Postitoimipaikat
  -Maat
  -Hintaluokat
  -Kustannuspaikat
  -Viitenumerosarjat
  -Tilit
  -Käyttäjät
  -Yritykset
  -Parametrit
  -Omat luokitukset

Liite 2: Haastattelukysymykset

Tausta

1. Haastateltavan nimi?
2. Haastateltavan koulutus?
3. Työkokemus alalta?
4. Asema Leanvay Oy:ssä?

Tutkimuskohteen (LogiPlan) Kuvaus

1. Milloin luotu?
2. LogiPlanin käyttötarkoitus?
3. Paljonko asiakkaita/käyttäjiä?

Sovelluskehyksen valinta (Angular vs. Wicket)

1. Mistä syystä päätettiin vaihtaa sovelluskehystä?
2. Harkittiinko muita sovelluskehyksiä ennen Angularin valitsemista?

Lomakkeet LogiPlanissa

1. Mihin kaikkeen LogiPlanissa käytetään lomakkeita?
2. Miksi ne ovat tarpeellisia?
3. Kuinka paljon niitä on arviolta?
4. Miten monimutkaisia ne ovat pahimmillaan?
5. Koodin parantamiset edut?

Koodikatselmointi

1. Mitkä luokat mukaan (Reaktiivisia lomakkeita tutkittaessa)?

2. Onko koodia eritoten katselmoitu?

3. Jos on, niin millä metodeilla?