



Osaamista
ja oivallusta
tulevaisuuden
tekemiseen

Valtteri Nieminen

Hyvät ohjelmointikäytännöt sovelluksen käyttöliittymän ylläpidossa

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

9.4.2019

Tekijä Otsikko Sivumäärä Aika	Valtteri Nieminen Hyvät ohjelmointikäytänteet sovelluksen käyttöliittymän ylläpidossa 53 sivua 9.4.2019
Tutkinto	Insinööri (AMK)
Tutkinto-ohjelma	Tieto- ja viestintätekniikka
Ammatillinen pääaine	Pelisovellukset
Ohjaajat	Yliopettaja Kari Aaltonen Johtaja Laura Hautsalo
<p>Insinööriyönä tehtiin työtaturmavakuutuksen tarjous- ja siirtojärjestelmä, joka oli verkkosovellus, joka automatisoi vakuutusyhtiöiden välistä kilpailua Suomen lakisääteisiä työtaturma- ja ammattitautivakuutuksia ostavista asiakkaista. Insinööriyön tavoitteena oli noudattaa luettavan ja ylläpidettävän koodaamisen periaatteita sovelluksen ylläpito- ja pienkehitystyössä.</p> <p>Käyttöliittymäkomponenttien koodaamisessa noudatettiin kerrosten löyhää kytkemistä pitämällä rakenne, toiminnallisuus ja ulkoasu omissa tiedostoissaan. Koodi pidettiin tehokkaana toteuttamalla algoritmit yksinkertaisesti ja selkeästi. Uudelleenkäytettävyyttä ja laajennettavuutta edistettiin käyttämällä luokkarakenteita, tekemällä yleiskäyttöisiä aliohjelmia ja välttämällä suoria tietojäsenviittauksia. Koodissa noudatettiin yleisiä kommentointi-, esittely-, muotoilu- ja nimeämiskäytäntöjä.</p> <p>Asiakkaan palaute oli myönteistä, ja ohjelman ylläpitoon osallistuneiden ohjelmoijien yleinen mielipide oli, että tuotettu koodi oli luettavaa ja ylläpidettävää. Hyvä palaute voidaan lukea hyvien käytänteiden mukaisen ylläpidon ansioksi.</p>	
Avainsanat	ohjelmointikäytänteet, ylläpito, käyttöliittymä, JavaScript, Java

Author Title	Valtteri Nieminen Coding Best-Practices in Front End Application Maintenance
Number of Pages Date	53 pages 9 April 2019
Degree	Bachelor of Engineering
Degree Programme	Information and Communication Technology
Professional Major	Game Applications
Instructors	Kari Aaltonen, Principal Lecturer Laura Hautsalo, Manager
<p>The subject of the final year project was the Quote and Transfer System for Workers' Insurance, which was a web application for automating the competition between insurance companies who sell workers' insurances required by the Finnish law. The goal of the project was to follow the principles of writing readable and maintainable code in maintaining the application.</p> <p>The components of the user interface were coded implementing loose coupling of the user interface layers by keeping structural, functional and visual aspects to their respective layers. The code was made effective by implementing algorithms in a simple and straightforward way, and reusable and expandable by using classes whenever possible, making functions general-purpose enough and avoiding referencing data members directly. General conventions were followed regarding comments, declarations, layout and naming.</p> <p>The feedback from the customer was positive. There was a consensus among the developers involved in maintaining the application that the produced code was readable and maintainable. In light of the positive feedback from both the customer and the development team it can be concluded that good coding practices were followed in maintaining the application.</p>	
Keywords	coding best-practices, maintenance, front end, JavaScript, Java

Sisällys

1	Johdanto	1
2	Hyvät ohjelmointikäytännöt	1
2.1	Käyttöliittymän kerrosten löyhä kytkeminen	2
2.2	Ylläpidettävyys	4
2.3	Uudelleenkäytettävyys ja laajennettavuus	4
2.4	Testattavuus	6
2.5	Java-ohjelmointikielen käytännöt	7
2.6	Dokumentointi	16
3	Ohjelman ylläpito	17
3.1	Yleistä	17
3.2	Ohjelma-analyysi	18
3.3	Ylläpidon ongelmia	19
3.4	Ylläpitotyö	21
4	Työtaturmavakuutuksen tarjous- ja siirtojärjestelmän määrittelyt	23
4.1	Taustaa	23
4.2	Projektin tavoitteet ja asiakkaan vaatimukset	23
4.3	Käyttöliittymä	24
4.4	Haasteet	31
5	Työtaturmavakuutuksen tarjous- ja siirtojärjestelmän toteutus	31
5.1	Ohjelman arkkitehtuuri	31
5.2	Angular-ohjelmistokehys	33
5.3	Pienkehitys- ja virheidenkorjaustöitä	34
5.4	Palaute	41
6	Yhteenveto	44
	Lähteet	46

1 Johdanto

Insinööri työ tehtiin osana työtaturmavakuutuksen tarjous- ja siirtojärjestelmän ylläpitotyötä. Tavoitteena oli noudattaa työssä luettavan ja ylläpidettävän koodin kirjoittamisen periaatteita ja käytänteitä. Työhön kuului vikakorjauksia ja pienkehitystä.

Työn tilasi tietotekniikka- ja konsulttiyritys CGI myytäväksi vakuutusyhtiölle. Vakuutusyhtiöön liittyvät tiedot, kuten sen nimi, ovat luottamuksellisia, eikä niitä käsitellä raportissa. Ohjelmisto on tarkoitettu automatisoimaan vakuutusyhtiöiden kilpailua. Työ oli laaja; sovelluksessa oli monia toimintoja, ja siihen liittyi palveluita, joista osasta oli vastuussa toisia tietotekniikkayrityksiä.

Insinööri työn aihe valittiin, koska tilaustyö oli otollinen ylläpitotyön ohjelmointikäytänteiden tarkasteluun. Työn tavoitteena oli auttaa havainnollistamaan laadukkaan ohjelmoinnin periaatteiden merkitystä ja kannustaa ohjelmoijia toteuttamaan niitä.

Luvuissa 2 ja 3 perehdytään ylläpidettävän koodin kirjoittamisen periaatteisiin ja käytänteisiin sekä sovellusten ylläpitoon.

2 Hyvät ohjelmointikäytännöt

Ohjelmoinnille on olemassa käytänteitä eli toimintamalleja, joilla pyritään tuottamaan mahdollisimman laadukasta koodia. Koodin laadulla tarkoitetaan paitsi toimivia algoritmeja myös yleiskäyttöisyyttä ja ylläpidettävyyttä.

Ohjelman ylläpidettävyyden vaaliminen on eduksi yrityksille. Ylläpidettävän ohjelman kustannukset ovat matalammat verrattuna vaikeasti ylläpidettäviin ohjelmiin. Lisäksi ohjelmistokehitys on muuttunut viime aikoina asiakkaiden alkaessa vaatia parempaa laatua, tehoa ja palvelua. Ohjelmistojen elinkaaret ovat lyhentyneet, mikä tuo yrityksille uusia haasteita voiton tuottamiseen. Siksi ylläpidettävä koodi on tärkeää. [Smit ym. 2011; Sambamurthy ym. 2003; Mathiyalkan ym. 2005; Flidner & Vokurka 1997.]

Michael Smit, Barry Gergel, James Hoover ja Eleni Stroulia tutkivat vuonna 2011 ohjelmointikäytänteiden noudattamista ohjelmistoprojekteissa. Tutkimuksessa tarkasteltiin neljää ohjelmistoprojektia. Tutkimus osoitti, että ohjelman koko on suoraan verrannollinen ohjelmointikäytäntörikkeiden määrään ja yleisimpiä rikkeitä olivat dokumentaatio-kommenttien puuttuminen, final-määreen asettamatta jättäminen muuttumattomille muuttujille, kovakoodatut kokonaisluvut ja merkkijonot sekä sulkeiden käyttöön liittyvät rikkeet. Lisäksi tutkimus osoitti, että pienempi kehitystiimi johtaa pienempään määrään rikkeitä. [Smit ym. 2011.]

2.1 Käyttöliittymän kerrosten löyhä kytkeminen

Nicholas Zakas [2012: 67–72] puhuu ylläpidettävän ja luettavan JavaScript-koodin kirjoittamisesta kertovassa kirjassaan käyttöliittymän kerrosten löyhästä kytkemisestä. Käyttöliittymän kerroksilla tarkoitetaan HTML-verkkodokumenttia, sen interaktiiviset toiminnallisuudet sisältävää JavaScript-tiedostoa sekä dokumentin tyyli-informaation sisältävää CSS-tiedostoa. Zakas painottaa, että ylläpidettävyyttä ajatellen näiden kolmen kerroksen yhteydet tulisi rajoittaa vain niiden pakollisiin yhteyksiin, jotta yhteen kerrokseen voitaisiin tehdä muutoksia niin, että muita kerroksia jouduttaisiin muokkaamaan mahdollisimman vähän, tai mieluiten ei lainkaan.

Lisäksi kerroksille ei koskaan tulisi antaa tehtäviä, jotka eivät teoriassa kuulu niille. Tekstiin ja rakenteeseen liittyvät asiat kuuluvat HTML-dokumentille, ulkoasuun liittyvät asiat CSS-tiedostolle ja toiminnallisuuteen liittyvät asiat JavaScript-tiedostolle. Kun koodi seuraa tätä periaatetta, ylläpitäjän etsiessä esimerkiksi syytä jollekin toiminnallisuuteen liittyvään vialle riittää, että hän tutkii JavaScript-tiedostoa eikä hänen tarvitse käydä läpi HTML- ja CSS-tiedostoja. [Zakas 2012: 67–68.]

Jotkin selaimet sallivat CSS-tiedostossa expression-funktion, joka mahdollistaa JavaScript-lauseet CSS-tiedostoissa. Tätä ei suositella edellä mainitusta syystä. HTML-dokumentin ja CSS-tiedoston välinen kommunikaatio tulisi rajoittaa HTML-elementtien class-attribuuttiin. HTML-elementin tyyli-informaatiota voidaan muokata JavaScriptissä elementin style-tietojäsenen kautta (esimerkkikoodi 1), mutta tätä ei suositella.

Huono tapa:

JavaScript-koodissa:
`element.style.color = "red";`

Oikea käytäntö:

CSS-tiedostossa:
`.punainen {
 color: red;
}`

HTML-dokumentissa:
`<element class="punainen">...</element>`

Esimerkkikoodi 1. HTML-dokumentin ja CSS-tiedoston oikeaoppinen kommunikaatio [Zakas 2012: 68].

HTML-kielessä on elementtien attribuuteiksi lisättäviä tapahtumakuuntelijoita, joiden arvoksi voidaan kirjoittaa JavaScript-lauseita. Näistä yleisin on `onclick`, joka tutkii, napsauttaako käyttäjä elementtiä (esimerkkikoodi 2). Yleisyydestään huolimatta näitäkään ei ole ylläpidettävyyden kannalta suositeltavaa käyttää. Parempi käytäntö on käyttää JavaScriptin `add event listener` -funktiota, joka tutkii käyttäjän tekemisiä elementtien suhteen (esimerkkikoodi 3).

HTML-dokumentissa:
`<button onclick="teeJotain()" id="toimintapainike">Napsauta minua</button>`

Esimerkkikoodi 2. Tätä tapaa seurata käyttäjän hiirenpainalluksia ei suositella [Zakas 2012: 70].

JavaScript-tiedostossa:

```
function teeJotain() {
    // Koodia.
}

var painike = dokumentti.getElementById("toimintapainike");
painike.addEventListener("click", teeJotain, false);
```

Esimerkkikoodi 3. Hyvä tapa seurata käyttäjän hiirenpainalluksia [Zakas 2012: 71].

HTML:n `script`-elementtiin voidaan kirjoittaa JavaScript-koodia. Edellä mainituista syistä tätä ei suositella. HTML-merkkauksen käyttö JavaScript-koodissa on niin ikään mahdollista HTML-elementin `innerHTML` -tietojäsenen avulla; senkin käyttöä tulee välttää. [Zakas 2012: 67–72.]

2.2 Ylläpidettävyys

Mika Paulasaari painottaa luettavuuden merkitystä koodin ylläpidettävyydessä vuoden 2018 opinnäytetyössään, joka tutki tapoja edistää koodin laatua käyttöliittymäkehityksessä. Hänen mukaansa kenen tahansa koodia muokkaavan ohjelmoijan tulisi pystyä helposti ymmärtämään olemassa olevaa koodia [Paulasaari 2018: 8]. Koodia kirjoitettaessa on jo alusta lähtien huomioitava koodia tulevaisuudessa muokkaavat ohjelmoijat.

Paulasaari [2018: 8] mainitsee merkityksellisen nimeämiskäytännön yhtenä koodin luettavuuteen vaikuttavana tekijänä. Luokat, muuttujat ja metodit voidaan teoriassa nimetä mielivaltaisesti, mutta niille on selkeyden vuoksi suositeltavaa antaa mahdollisimman kuvaavat nimet. Nimet, joista voidaan helposti päätellä tarkoituksia ja toimintoja, nopeuttavat huomattavasti uuden ohjelmoijan tutustumista ennestään tuntemattomaan koodiin. Lisäksi on tarkempia ohjelmointikielikohtaisia sääntöjä esimerkiksi isoin alkukirjaimin kirjoittamiseen liittyen. Esimerkiksi Java-kielessä on käytäntö, jossa luokkien nimet kirjoitetaan isolla alkukirjaimella ja olioiden ja metodien nimet pienellä. Nimeämiskäytäntö auttaa assosioimaan syntaksin semanttisen tarkoituksen kanssa [Smit ym. 2011: 1].

On suositeltavaa käyttää ohjelmointikielten kommenttisyntaksia tarvittaessa ja kirjoittaa sillä kuvauksia koodista itse lähdekoodin lomaan. Tämä on helpoin tapa esitellä koodia siihen tulevaisuudessa tutustuville ohjelmoijille, ja sitä on suositeltavaa käyttää mahdollisimman usein. [Smit ym. 2011: 1; Kosonen ym. 2005: 51.]

2.3 Uudelleenkäytettävyys ja laajennettavuus

Ylläpitoon liittyy usein ohjelman laajennustöitä. Laajennustöissä koodin uudelleenkäytettävyyden merkitys korostuu. Koodin uudelleenkäytettävyys tarkoittaa eri osien monikäyttöisyyttä. Ilmeisimpänä esimerkkinä oliopohjaisten kielten luokkarakenteen käyttö edistää koodin uudelleenkäytettävyyttä. Hyvä käytäntö on luoda luokka kaikille asioille, joita voidaan pitää erillisinä, esimerkiksi "vakuutus", ja tehdä kaikista asioista, joita voidaan pitää tämän luokan esiintyminä, esimerkiksi "työtaturmavakuutus", tämän luokan olioituja. Jos siis ohjelman joillakin erillisillä osilla on samat ominaisuudet ja toiminnallisuudet, voidaan tehdä luokka, jolle kirjoitetaan nämä ominaisuudet ja toiminnallisuudet kerran.

Näistä erillisistä osista voidaan sitten tehdä luokan ilmentymiä, jotka saavat ominaisuuksensa ja toiminnallisuutensa samalta luokalta. [Hietanen 2003: 19.]

Jos tarvitaan luokka, jolla on samat ominaisuudet kuin olemassa olevalla luokalla ja lisäksi muutama uusi, voidaan luoda aliluokka, joka perii ylläluokan ominaisuudet, jolloin ei ole tarvetta ohjelmoida niitä uudelleen [Kosonen ym. 2005: 137].

Koodin laajennettavuuteen tulee kiinnittää huomiota, kun tehdään aliohjelmia, luokkia ja metodeja. Näitä rakenteita on suositeltavaa ohjelmoida niin, että niitä voidaan käyttää mahdollisimman monipuolisesti. [Peltomäki & Nykänen 2006: 136.] Esimerkiksi usea erilainen lajittelualiohjelma eri tietotyypeille ja järjestyksen suunnille voidaan yhdistää yhdeksi aliohjelmaksi, jonka järjestyksen suunta ja eri tietotyyppien vertailufunktiot määritetään parametreilla (esimerkkikoodi 4).

```
void lajittele1(...); /* Lajittelee kokonaislukuja nousevassa järjestyksessä. */
void lajittele2(...); /* Lajittelee merkkijonoja laskevassa järjestyksessä. */
void lajittele3(...); /* Lajittelee reaalityyppejä laskevassa järjestyksessä. */
```

Edelliset yhdistettynä laajempikäyttöiseen lajittelufunktioon:

```
/* Lajittelee oliotaulun nousevassa tai laskevassa järjestyksessä. */
void sort (
    int *taulu,
    int järjestys,          /* 1 = nouseva, 2 = laskeva. */
    int (*vertaile)(),      /* Tietotyyppikohtainen vertailufunktio. */
    int (*vaihdaPäittäin)() /* Tietotyyppikohtainen päittäinvaihtamis-
                                funktio. */
)
```

Esimerkkikoodi 4. Useiden eri lajittelufunktioiden sijaan voidaan tehdä yksi, jota voidaan käyttää useissa erilaisissa tapauksissa [Harsu 2003: 259].

Aliohjelmalle tai metodille on suositeltavaa antaa vain yksi tehtävä. Mitä enemmän tehtäviä aliohjelmalla on, sitä tarkemmin sen käyttötarkoitus määräytyy ja sen uudelleen käytettävyys kärsii. [Peltomäki & Nykänen 2006: 136.]

Suora viittaaminen olion tietojäseniin sen ulkopuolelta (esimerkkikoodi 5) ei ole suositeltavaa, vaan parempi käytäntö on käyttää olion tietojäsenien lukemiseen ja muokkaamiseen luotuja get- ja set-metodeja (esimerkkikoodi 6). Tällöin jos luokan rakennetta muutetaan jälkeenpäin ja esimerkiksi sen tietojäsenten nimiä muutetaan, ei jouduta muuttamaan jokaista ohjelman kohtaa, jossa tämä tietojäsen luetaan, sillä metodikutsuissa ei käytetä tietojäsenten nimiä. Tämä edistää ohjelman laajennettavuutta. [Kosonen ym. 2005: 120.]

```
tarjous.summa = 40000.0;
tarjous.summa = tarjous.summa * 1.05;
```

Esimerkkikoodi 5. Suora viittaus tarjous-olion summa-tietojäseneen. Suora viittaus ei ole suositeltavaa koodin laajennettavuuden ja ylläpidettävyyden kannalta.

```
tarjous.setSumma(40000.0);
tarjous.setSumma(tarjous.getSumma() * 1.05);
```

Esimerkkikoodi 6. Esimerkkikoodi 5 käyttäen get- ja set-metodeja. Tämä tapa on parempi koodin laajennettavuuden ja ylläpidettävyyden kannalta.

2.4 Testattavuus

Koodia kirjoitettaessa on syytä kiinnittää huomiota sen testattavuuteen. Yksikkötesteissä funktioita testataan yksitellen ja toisistaan erillään. Yksikkötestaus edellyttää, että funktio on kirjoitettu niin, että sen lopputulos riippuu täysin sen parametreina saamista tiedoista ja se tuottaa samoilla parametreilla aina saman lopputuloksen. Esimerkiksi date time -luokan käyttäminen funktiossa on tämän periaatteen vastaista (esimerkkikoodi 7); funktio ei tällöin saisi kaikkea käyttämänsä tietoa parametreina ja se saattaisi tuottaa eri tuloksia testattaessa samoilla parametreilla.

```
public static boolean OnkoVakuutusVoimassa(Vakuutus vakuutus)
{
    Date päivämäärä = new Date();
    if (päivämäärä > vakuutus.getPäätymispäivä()) {
        return false;
    } else {
        return true;
    }
}
```

Esimerkkikoodi 7. Funktio tutkii, onko vakuutus voimassa [Kolodiy 2019].

Yksikkötestauksessa on tärkeää, että funktion kaikki parametrit voidaan luoda keinotekoisesti testausta varten. Lisäksi funktion tulisi tehdä ainoastaan sille annettu tehtävä, eikä päivämäärän tutkiminen ole sen tehtävä. Ongelma ratkaistaan antamalla päivämäärä funktiolle parametrina (esimerkkikoodi 8).

```
public boolean onkoVakuutusVoimassa(Date päivämäärä, Vakuutus vakuutus)
{
    if (päivämäärä > vakuutus.getPäätymispäivä()) {
        return false;
    } else {
        return true;
    }
}
```

Esimerkkikoodi 8. Funktio ei tutki päivämäärää itse, vaan saa sen parametrina.

Toinen ratkaisu olisi luoda päivämäärän tutkimiselle oma funktio, joka voitaisiin korvata testauksessa. Näin yksikkötestiä varten voidaan päivämääräntutkimisfunktion sijaan käyttää funktiota, joka voidaan asettaa palauttamaan mielivaltaisia päivämääriä testausta varten.

Yksikkötestausta ajatellen tulisi ohjelmoijan myös pitää huoli, ettei funktioilla ole sivuvaikutuksia. Funktiota ei voida yksikkötestata täydellisesti, jos se arvojen palauttamisen lisäksi esimerkiksi kutsuu toista funktiota, sillä tämän toisen funktion tulos ei ilmenisi palautuvista arvoista. (Esimerkkikoodi 9.) [Kolodiy 2019.]

```
public boolean onkoVakuutusVoimassa(Date päivämäärä, Vakuutus vakuutus)
{
    if (päivämäärä > vakuutus.getPäättymispäivä()) {
        korvaaVahinko();
        return true;
    } else {
        return false;
    }
}
```

Esimerkkikoodi 9. Korvaa vahinko -funktiota ei pitäisi kutsua Onko vakuutus voimassa -funktion sisällä, sillä tämän toiminnan varmistaminen yksikkötestauksessa on hankalaa.

Globaalit tietojäsenet haittaavat yksikkötestausta. Ohjelman osaa, joka käyttää globaaleja tietojäseniä, ei voida yksikkötestata ilman, että testausta varten joudutaan rakentamaan myös se korkeamman tason ohjelman osa, joka määrittelee globaalit tietojäsenet. [Zakas 2012: 82.]

2.5 Java-ohjelmointikielen käytänteet

Java-ohjelmointikielellä on useita vakiintuneita muun muassa nimeämiseen, kommentointiin ja koodilohkoihin liittyviä käytänteitä. Kehittäjät Peter King, Patrick Naughton, Mike DeMoney, Jonni Kanerva, Kathy Walrath ja Scott Hommel perustelevat ohjelmointikäytänteitä Oraclen verkkoaineistossa muutamalla syyllä:

- Ylläpitopalvelu käsittää 80 prosenttia ohjelmiston kustannuksista sen eliniän aikana.
- Ohjelmiston alkuperäinen tekijä ei lähes koskaan ylläpidä sovellusta sen koko elinkaaren ajan.

- Ohjelmointikäytänteet edistävät ohjelmiston luettavuutta. Insinöörit omaksuvat uutta koodia nopeammin ja perusteellisemmin, kun se noudattaa hyviä ohjelmointikäytänteitä.
- Jos ohjelman lähdekoodia myydään tuotteena, on sen oltava yhtä laadukasta kuin itse ohjelmistonkin. [King ym. 1999: 1.]

Ohjelmointikäytänteitä noudattavaa koodia on helpompi ylläpitää [Smit ym. 2011: 2]. Käytänteet tuntevan ohjelmoijan on helpompaa tulkita uutta koodia, kun se noudattaa käytänteitä. Asetteluun liittyviä käytänteiden noudattamista voidaan helpottaa lähdekoodin tarkastavilla työkalulla. Tämä kohentaa ohjelman ylläpidettävyyttä tutkitusti. [Smit ym. 2011: 10.]

Sisennys

Sisentämiseen tulee käyttää neljää välilyöntiä. Sarkaimet tulee asettaa kahdeksan välilyönnin välein. Yhdellä rivillä tulisi olla enintään 80 merkkiä, sillä monet komentopäätteet eivät pysty käsittelemään pidempiä rivejä. Dokumentaatiossa käytettävissä koodiesimerkeissä tulisi rivin pituus rajoittaa vain 70 merkkiin. Rivinvaihdossa tulisi noudattaa seuraavia periaatteita:

- Riviä vaihdetaan pilkun jälkeen.
- Riviä vaihdetaan ennen operaattoria.
- Riviä vaihdetaan mieluummin ylemmällä sisennyksen tasolla kuin alemmalla (esimerkkikoodi 10).
- Uusi rivi linjataan saman tason lauseen alun kanssa (esimerkkikoodi 11).
- Kaikilla lisäriveillä käytetään kahdeksan välilyönnin sisennystä, jos se on selkeämpää tai jos sivun oikea laita tulee vastaan (esimerkkikoodi 12). [King ym. 1999: 5.]

```
pitkäNimi1 = pitkäNimi2 * (pitkäNimi3 + pitkäNimi4 - pitkäNimi5)
              + 4 * pitkäNimi6; // SUOSITELTAVA TAPA.

pitkäNimi1 = pitkäNimi2 * (pitkäNimi3 + pitkäNimi4
                          - pitkäNimi5) + 4 * pitkäNimi6; // VÄLTETTÄVÄ TAPA.
```

Esimerkkikoodi 10. Sisäkkäisiä lausekkeita sisältävä lause suositellaan jaettavaksi uudelle riville ylemmän tason lausekkeen kohdalta [King ym. 1999: 6].

```
jokuMetodi(pitkäLause1, pitkäLause2, pitkäLause3,
            pitkäLause4, pitkäLause5);

muuttuja = jokuMetodi1(pitkäLause1,
                       jokuMetodi2(pitkäLause2,
                                    pitkäLause3));
```

Esimerkkikoodi 11. Esimerkki Java-kielen rivinvaihtokäytännöstä [King ym. 1999: 5].

```
// SÄÄNTÖJEN MUKAINEN SISENNYS.
jokuMetodi(int argumentti, Object toinenArgumentti, String kolmasArgumentti,
            Object neljäsArgumentti) {
    ...
}

// KÄYTÄ KAHDEKSAN VÄLILYÖNNIN SISENNYSTÄ, KUN SISENNYKSESTÄ TULISI MUUTEN
// LIIAN SYVÄ.
private static synchronized todellaPitkäMetodinimi(int argumentti,
            Object toinenArgumentti, String kolmasArgumentti,
            Object neljäsArgumentti) {
    ...
}
```

Esimerkkikoodi 12. Sellaisissa tapauksissa, joissa sisennyskäytäntö aiheuttaisi liian syvän sisennuksen, käytetään kahdeksan välilyönnin sisennystä kaikille lisäriveille [King ym. 1999: 6].

Ehtolauseiden rivinvaihdoissa kannattaa selkeyden vuoksi käyttää aina poikkeussisennystä, sillä muuten itse ehdonalainen toimenpide saattaa sekoittaa ehtolausekkeeseen (esimerkkikoodi 13).

```
// EHTOLAUSEISSA EI KANNATA KÄYTTÄÄ TÄLLAISTA LISÄRIVISISENNYSTÄ:
if ((ehto1 && ehto2)
    || (ehto3 && ehto4)
    ||!(ehto5 && ehto6)) {
    metodi(); // TÄMÄ LAUSE NÄYTTÄÄ, KUIN SE OLISI OSA EDELLISELLÄ RIVILLÄ
    // LOPPUVAA EHTOLAUSEKETTA.
}

// TÄMÄ ON PAREMPI TAPA SISENTÄÄ EHTOLAUSEKKEEN LISÄRIVIT:
if ((ehto1 && ehto2)
    || (ehto3 && ehto4)
    ||!(ehto5 && ehto6)) {
    metodi();
}
```

Esimerkkikoodi 13. Ehtolauseiden lisärivien sisennys [King ym. 1999: 6].

Kommentit

Java-ohjelmissa käytetään kahta kommenttityyppiä: toteutuskommentteja ja dokumentaatiokommentteja. Toteutuskommentit merkitään kahdella vinoviivalla // rivin alussa tai vinoviivalla ja asteriskilla /* ensimmäisen kommenttirivin alussa ja asteriskilla ja vinoviivilla */ rivin lopussa.

valla */ viimeisen kommenttirivin lopussa. Dokumentaatiokommentit merkitään vinoviivalla ja kahdella asteriskilla /** ensimmäisellä rivillä, välilyönnillä ja asteriskilla " *" kaikilla seuraavilla riveillä ja asteriskilla ja vinoviivalla */ viimeisellä rivillä. Javadoc-työkalulla voidaan muodostaa dokumentaatiokommenteista HTML-tiedostoja.

Toteutuskommentteja käytetään uloskommentointiin (koodilohkojen piilottamiseen kääntäjältä, kun niitä ei haluta poistaa lopullisesti) tai tietyn toteutustavan selittämiseen. Dokumentaatiokommentteja käytetään koodin määrittelyn toteutustapaa yleisemmän tason kuvauksiin.

Kommentteja tulisi käyttää antamaan ohjelmoijalle yleiskuva koodista ja lisätietoa, joka ei käy ilmi itse koodista. Kommenteissa tulisi olla ainoastaan ohjelman lukemiselle ja ymmärtämiselle välttämätöntä tietoa. Esimerkiksi tiedot pakkauksen kääntämisestä tai hakemistoista eivät kuulu kommentteihin.

Epätavalliset toteutustavat on hyvä selventää kommentteilla, mutta ylenpalttinen kommentointi ei ole suositeltavaa ylläpidettävyyden kannalta; kommentteja saatetaan joutua päivittämään, mikä lisää työtaakkaa. Koodia, jota todennäköisesti joudutaan muuttamaan, ei kannata kommentoida. On myös huomattava, että runsaan kommentoinnin tarve saattaa olla merkki huonolaatuisesta koodista. Ennen kommentoimista on mietittävä, voitaisiinko koodi kirjoittaa selkeämmin niin, että sen tarkoitus tulee ilmi ilman, että sitä tarvitsisi erikseen selostaa.

Dokumentaatiokommentit kuvailevat Java-luokkia, -rajapintoja, -konstruktoireita, -metodeita ja -kenttiä. Jokaiselle julkiselle luokalle ja metodille tulee kirjoittaa kuvaus dokumentaatiokommentilla.

Luokkien ja rajapintojen dokumentaatiokommentit ovat sistenttisiä, ja niiden jäsenien dokumentaatiokommentit ovat sisennettyjä. Dokumentaatiokommentin ensimmäinen rivi koostuu vinoviivasta ja kahdesta asteriskista /**, ja kaikki seuraavat rivit viimeisen rivin */-lopetusmerkintää myöten sisennetään yhdellä välilyönnillä, jolloin syntyy pystysuora asteriskien sarake (esimerkkikoodi 14).

```
/**
 * Esimerkki-luokka sisältää...
 */
public class Esimerkki { ...
```

Esimerkkikoodi 14. Dokumentaatiokommentti [King ym. 1999: 9].

Tietoja luokasta, rajapinnasta, muuttujasta tai metodista, jotka eivät sovi dokumentaatioon (kuten toteutukseen liittyviä yksityiskohtia), ei tule kirjoittaa dokumentaatiokommenttiin, vaan välittömästi esittelyn jälkeen kirjoitettavaan lohkokommenttiin tai yksiriviseen kommenttiin. Sen sijaan dokumentaatiokommenttia ei saa kirjoittaa metodi- tai konstruktoresittelylohkon sisäpuolelle, sillä Java liittää dokumentaatiokommentin ensimmäiseen kommentin jälkeiseen esittelyyn. [King ym. 1999: 9.]

Esittelyt

Esittelyt suositellaan [King ym. 1999: 10] kirjoitettavan omille riveilleen, sillä muuttujien kommentointi on tällöin helpompaa. Erityyppisten muuttujien esittely samalla rivillä on kiellettyä. (esimerkkikoodi 15.)

```
int taso; // Sisennystaso.
int koko; // Taulukon koko.

int taso, koko; // Ei suositeltava tapa.
int foo, footaulu[]; // Kielletty tapa.
```

Esimerkkikoodi 15. Esittelyt [King ym. 1999: 10].

Tyyppi voidaan erottaa nimestä joko yhdellä välilyönnillä (esimerkkikoodi 15) tai nimet voidaan linjata (esimerkkikoodi 16).

```
int taso; // Sisennystaso.
int koko; // Taulukon koko.
Object nykyinenKohta; // Taulukon nykyinen valittu kohta.
```

Esimerkkikoodi 16. Linjatut muuttujanimet esittelyissä [King ym. 1999: 10].

Paikallismuuttujat tulisi aina alustaa niiden esittelyn yhteydessä, paitsi jos niiden alkuarvo riippuu jostakin myöhemmin tehtävästä laskutoimituksesta.

Esittelyt tulee aina sijoittaa lohkon alkuun. Muuttujan esittely juuri ennen sille tehtävää toimenpidettä johtaa sekavaan koodiin ja haittaa koodin uudelleenkäytettävyyttä näkyvyalueen sisäpuolella. (Poikkeuksena `for`-silmukkojen indeksit voidaan esitellä `for`-lausekkeessa.)

Esittelyjä, jotka piilottavat ylemmän tason esittelyjä, tulee välttää, sillä tämä mutkistaa ohjelmoimista tarpeettomasti. Esimerkkikoodissa 17 `määrä`-muuttuja esitellään uudelleen luokan metodissa, jolloin instanssimuuttuja `määrä` piilotetaan ja viittaus "`määrä = ...`" ei enää viittaakaan luokan instanssimuuttujaan, vaan metodissa esiteltyyn paikallismuuttujaan. Tässä tapauksessa haluttaessa viitata instanssimuuttujaan se jouduttaisiin tekemään `this`-viitteen avulla: "`this.määrä = ...`". [King ym. 1999: 10–11.]

```
int määrä;
...
metodi() {
    if (ehto) {
        int määrä = 0;
        ...
    }
    ...
}
```

Esimerkkikoodi 17. Vältettävä esittelytapa [King ym. 1999: 11].

Lauseet

Yhtä riviä kohden tulisi olla vain yksi lause esimerkkikoodin 18 kuvaamalla tavalla. Lohkolauseissa, eli lauseissa, joihin liitetään aaltosulkeilla ympäröity joukko lauseita, liitetyt lauseet tulee sisentää yhden tason verran lohkolauseen suhteen. Liitettyjen lauseiden lohkon aloittava aaltosulku suositellaan kirjoitettavan samalle riville kuin lohkolause ja lopettava aaltosulku omalle rivilleen samalle tasolle lohkolauseen kanssa. (Esimerkkikoodi 19.)

```
argumentti1++;           // Oikein.
argumentti2--;           // Oikein.
argumentti3++; argumentti4--; // Väärin.
```

Esimerkkikoodi 18. Jokainen lause tulee kirjoittaa selkeyden vuoksi omalle rivilleen [King ym. 1999: 11].

```
if (ehto) { // Lohkolause.
    lause;   // Liitetyt lauseet.
}
```

Esimerkkikoodi 19. Lohkolauseen asettelu [King ym. 1999: 12].

Ehtolauseissa `else`- ja `else if` -lauseet kirjoitetaan samalle riville edeltävän `if`- tai `else if` -lohkon päättävän aaltosulkeen kanssa (esimerkkikoodi 20). Java-syntaksi sallii

aaltosulkeiden poisjätön, kun lohkolauseeseen liitettäviä lauseita on vain yksi (esimerkkikoodi 21), mutta tämä ei ole suositeltavaa, sillä on tavallista, että lisättäessä uusia lauseita unohdetaan lisätä aaltosulkeet, mikä johtaa virheeseen. [King ym. 1999: 11–13.]

```
if (ehto) {
    lauseet;
} else {
    lauseet;
}

if (ehto) {
    lauseet;
} else if (ehto) {
    lauseet;
} else {
    lauseet;
}
```

Esimerkkikoodi 20. Ehtolauseiden muotoilu [King ym. 1999: 12].

```
if (ehto)
    lause;
```

Esimerkkikoodi 21. Ehtolause on kirjoitettu ilman aaltosulkeita. Edellisessä esimerkkikoodissa näkyvä tapa on suositeltavampi myös vain yhden lauseen sisältäville ehtolauseille. [King ym. 1999: 13.]

Nimeämiskäytännöt

Tunnisteista voidaan päätellä, edustavatko ne esimerkiksi vakiomuuttujia, pakkauksia vai luokkia, kun noudatetaan tarkkoja nimeämiskäytänteitä. Tämä nopeuttaa koodin lukemista oleellisesti.

Pakkauksen nimen etuliitteen on oltava yksi internetin ylätasen verkkotunnuksista tai maatunnuksista pienaakkosin kirjoitettuna (com, edu, net, org, fi jne.). Nimen jälkimmäiset osat noudattavat tyypillisesti yrityskohtaista nimeämiskäytäntöä.

Luokkien ja rajapintojen nimien tulisi olla substantiiveja, ja ne kirjoitetaan isolla alkukirjaimella. Jos nimi koostuu useasta erikseen kirjoitettavasta sanasta, kirjoitetaan sanat yhteen, mutta jokaisen sanan ensimmäinen alkukirjain kirjoitetaan isolla: ”hallinnoitu paneeli” → HallinnoituPaneeli.

Metodien nimien tulisi olla verbejä, ja ne kirjoitetaan pienellä alkukirjaimella. Jos nimi koostuu useasta erikseen kirjoitettavasta sanasta, sanat kirjoitetaan yhteen ja ensimmäistä sanaa seuraavat sanat kirjoitetaan isoin alkukirjaimin: ”juokse nopeasti” → `juokseNopeasti`.

Instanssimuuttujien, luokkamuuttujien ja luokkavakioiden nimet kirjoitetaan pienellä alkukirjaimella, ja monisanaisissa nimissä kaikki sanat kirjoitetaan yhteen ja ensimmäistä sanaa seuraavat sanat kirjoitetaan isoin alkukirjaimin. Muuttujien nimien tulisi olla lyhyitä ja selkeitä. Yhden merkin pituiset nimet ovat sallittuja väliaikaisissa toimenpiteissä, esimerkiksi välitulosten laskemisessa. Vakiintuneita yhden merkin muuttujanimiä ovat konaislukumuuttujille `i`, `j`, `k`, `m` ja `n` sekä merkkimuuttujille `c`, `d` ja `e`.

Staatistien muuttujien nimet kirjoitetaan kokonaan isoilla alkukirjaimilla, ja sanavälejä ilmaistaan alaviivoilla: ”taulukon koko” → `TAULUKON_KOKO`. [King ym. 1999: 15–16.]

Ohjelmointikäytänteitä

Instanssi- ja luokkamuuttujista ei pidä tehdä julkisia ilman hyvää syytä. Useimmiten ainoastaan olio itse lukee ja muokkaa instanssimuuttujiaan, eikä luokan ulkopuolisten tekijöiden tarvitse nähdä niitä. Kuitenkin jos luokkaa käytetään tietorakenteena, on perusteltua tehdä sen muuttujista julkisia.

Staatisiin metodeihin ja muuttujiin ei ole suositeltavaa viitata olion kautta, vaan luokan kautta (esimerkkikoodi 22).

```
luokkametodi();           // Oikein.
Luokka.luokkametodi();    // Oikein.
olio.luokkametodi();      // Väärin.
```

Esimerkkikoodi 22. Viittaus staattiseen metodiin [King ym. 1999: 17].

Numeerisia vakioita ei suositella kovakoodattavan, paitsi arvoilla `-1`, `0` ja `1`, joita usein käytetään esimerkiksi `for`-silmukassa laskurin alkuarvoina.

Saman arvon antamista usealle eri muuttujalle samassa lauseessa on vältettävä, sillä se voi näyttää hämmentävältä (esimerkkikoodi 23).

```

luku1 = luku2 = 9;    // Väärin.
luku1 = 9;            // Oikein.
luku2 = 9;            // Oikein.

```

Esimerkkikoodi 23. Muuttujien arvojen asettaminen [King ym. 1999: 17].

King ym. [1999: 18] suosittelevat käyttämään sulkeita selkeyttämään ehtolausekkeiden laskujärjestystä, vaikka ne eivät teoriassa olisi tarpeen (esimerkkikoodi 24). Heidän mukaansa ohjelmoijilta ei voida odottaa, että he tunsivat lauselogiikan laskujärjestyksen. Toisaalta lauselogiikan laskujärjestys on itse asiassa sama kuin normaalien aritmeettisten operaatioiden laskujärjestys; loogiset operaattorit **tai** sekä **ja** vastaavat yksiselitteisesti aritmeettisiä yhteen- ja kertolaskuoperaattoreita.

```

if (a == b && c == d)
if ((a == b) && (c == d))

```

Esimerkkikoodi 24. Toisella rivillä on kirjoitettu ensimmäisen rivin lauseke sulkeilla, jotka eivät muuta lausekkeen merkitystä. Niillä vain varmistetaan, ettei ohjelmoija vahingossa tulkitse esimerkiksi, että arvoa *a* verrataan arvoon *(b && c)*. Vertailuoperaatiot lasketaan tietenkin ennen loogisia operaatioita. [King ym. 1999: 18.]

Yksinkertainen toimenpide on perusteltua koodata yksinkertaisella tavalla. Yksinkertaisimmille totuusarvotarkistuksille ja valintarakenteille on vakiintuneita kirjoitustapoja, joita on esitetty esimerkkikoodissa 25.

```

if (onkoTotta == true) {
    return true;
} else {
    return false;
}

// Ylempi paremmin:
return onTotta;

if (ehto) {
    return x;
}
return y;

// Ylempi paremmin:
return (ehto ? x : y);

```

Esimerkkikoodi 25. Vakiintuneita tiivistystapoja [King ym. 1999: 18].

Tapauksissa, joissa koodiosuus toimii, vaikka sen ei pitäisi eikä parempaa ratkaisua löydetä, on käytäntönä kirjoittaa koodin yhteyteen kommentti, joka sisältää tekstin "XXX". Tapauksissa, joissa koodiosuus ei toimi ja se tulisi korjata, kirjoitetaan "KORJAAMINUT". [King ym. 1999: 16–18.]

2.6 Dokumentointi

Dokumentointi on keskeisin osa ohjelmistotyötä. Dokumentointiin kuuluu ohjelman käyttöohjeen, testauskuvauksen, ylläpito-ohjeen, määrittelydokumenttien ja arkkitehtuurikuvauksen kirjoittaminen. Testauskuvauksen perusteella ohjelma on kyettävä milloin tahansa testaamaan uudelleen, ja testausta varten tulee valmistaa uudelleenkäytettävää testaustyökalustoa ja -materiaalia. Ylläpito-ohjeessa kuvataan, miten ohjelmaan voidaan lisätä uusia ominaisuuksia ja mitä uusia ominaisuuksia lisättäessä tulee ottaa huomioon.

Kaikista dokumenteista laajin ja yksityiskohtaisin on itse ohjelma. Ohjelman koodin rinnalle on kommentoitu, kuinka määriteltyt tavoitteet on saavutettu ohjelmallisesti. Javadoc-työkalulla voidaan koostaa Java-lähdekoodin dokumentaatiokommenteista HTML-tiedostoja. Dokumentaatiokommenttien @-alkuisia määreitä käytetään erilaisten tietojen välittämiseen Javadocille. Author-määreellä ilmoitetaan luokasta tai metodista vastaava ohjelmoija, ja sitä käytetään luokkakuvauksissa (esimerkkikoodi 26). [Kosonen ym. 2005: 20–21.]

```
/**
 * Vakuutusluokka, joka sisältää vakuutuksille yhteiset piirteet.
 * <p>
 * Esimerkki insinöörityötä varten. Kevät 2019.
 * <p>
 * @author Valtteri Nieminen (valtteri.nieminen@metropolia.fi),
 * Tieto- ja viestintätekniikan tutkinto-ohjelma, Metropolia Ammattikorkeakou-
 * lu.
 */
public class Vakuutus {
```

Esimerkkikoodi 26. Luokkakuvaukset [Laurikkala 2011].

Param-määreillä kuvaillaan metodin parametreja ja return-määreellä sen paluuarvoja (esimerkkikoodi 27).

```
/**
 * Tutkitaan, onko vakuutus voimassa.
 * Verrataan vakuutuksen päättymispäivää nykyhetkeen.
 *
 * @param vakuutus Tutkittava vakuutus.
 * @param tänään Tämän päivän päivämäärä.
 * @return True, jos vakuutus on voimassa ja muuten false.
 */
public boolean onkoVoimassa(Vakuutus vakuutus, Date tänään) {
```

Esimerkkikoodi 27. Metodikuvaus [Laurikkala 2011].

3 Ohjelman ylläpito

3.1 Yleistä

Ylläpito on ohjelman elinkaaren viimeinen vaihe, ja se kestää ohjelman käyttöönotosta sen käytöstä poistoon saakka. Ylläpidossa ohjelmaa muutetaan virheiden korjaamiseksi, laadun parantamiseksi sekä toiminnan muuttamiseksi muuttuneiden vaatimuksien mukaiseksi. [Berns 1984.]

Ohjelmaa joudutaan muuttamaan jälkikäteen muutamasta syystä. Ohjelman käyttäjät löytävät käytännössä poikkeuksetta ohjelmasta virheitä, ja nämä korjataan korjaavassa ylläpidossa. Käyttäjät saattavat ohjelman käyttöönoton jälkeen myös toivoa siihen täydennyksiä ja parannuksia; näiden toiveiden mukaisten muutosten toteuttaminen on täydellistävää ylläpitoa.

Ehkäisevässä ylläpidossa ohjelman rakennetta parannetaan niin, että tulevat ylläpito- ja muutostyöt helpottuvat. [Pressman 1994; Swanson 1976.]

Ohjelman kehitykseen liittyy viisi lainalaisuutta, joita kutsutaan Lehmanin laeiksi:

- 1) Ohjelmaa on päivitettävä, jotta se säilyy käyttökelpoisena.
- 2) Muutokset monimutkaistavat järjestelmää, ja rakenteen ja yksinkertaisuuden säilyttämiseen on nähtävä ylimääräistä vaivaa.
- 3) Ohjelman koko, versioiden julkaisutaajuus ja virheraporttien määrä pysyvät suunnilleen samana versiopäivitysten välillä.
- 4) Ohjelma kehittyy vakionopeudella riippumatta saatavilla olevista resursseista.
- 5) Jokainen versiopäivitys sisältää keskimäärin saman verran muutoksia ohjelmaan.

Ensimmäinen laki johtuu siitä, että reaali maailma muuttuu vääjäämättä, ja esimerkiksi vakuutusjärjestelmässä lait, joihin järjestelmän logiikka perustuu, saattavat muuttua. Lisäksi ohjelmaa käytettäessä siitä löydetään kehityskohteita, joita toivotaan korjattavan.

Toisen lain mukaan järjestelmän muuttuessa sen rakenne rikkoutuu. Tätä voidaan korjata ehkäisevällä ylläpidolla, jossa ohjelman rakennetta parannetaan muuttamatta sen

toimintaa. Tämä aiheuttaa hieman lisäkustannuksia, mutta säästää tulevien ylläpitotoimien kustannuksissa.

Kolmas laki johtuu yritysten haluttomuudesta tehdä ohjelmaan kerralla suuria muutoksia, sillä tämä on riskialtista. Ohjelmaan saattaa tulla paljon virheitä, ja sen toiminta saattaa huonontua. Lisäksi päätökset suurten muutosten tekemiseksi vievät aikaa.

Neljättä lakia ei osata tarkkaan selittää. Lehmanin ja Beladyn tutkimuksen mukaan ohjelmointiprojektin työryhmän kasvattaminen ei kiihdytä ohjelman kehitystä, vaan suuret järjestelmät vaikuttavat olevan jonkinlaisessa kyllästetyssä tilassa.

Viides laki johtuu virheraporttien määrän ja versiopäivityksen muutosten määrän verrannollisuudesta. Jos versiopäivityksessä on paljon muutoksia, siitä seuraa paljon korjattavia virheitä, ja seuraavassa versiopäivityksessä ei voida tehdä paljon muutoksia, sillä joudutaan keskittymään edellisen suuren päivityksen virheiden korjaamiseen. [Lehman & Belady 1985.]

3.2 Ohjelma-analyysi

Ohjelma-analyysi sisältää paitsi ohjelmasta olemassa olevan dokumentaation analysoinnin myös lähdekoodin analysoinnin. Ohjelma-analyysi jaetaan staattiseen ja dynaamiseen analyysiin; staattisessa analyysissä tutkitaan tekstiä ja dynaamisessa ohjelman ajoaikaista käyttäytymistä.

Staattisessa analyysissä tutkitaan ohjelman rakennetta. Staattisessa analyysissä saatetaan esimerkiksi huomata, että koodi koostuu moduuleista, moduulit aliohjelmista ja aliohjelmat edelleen sisäisistä aliohjelmista. Staattisessa analyysissä tarkastellaan myös ohjelman muuttujia ja käytettyjä tyyppejä. Saatetaan selvittää, mitä aliohjelmaa kutsutaan tietystä lauseesta, missä esitellään jokin tyyppi, jota käytetään tietyn muuttujan esittelyssä, mitä tietyn tyyppisiä muuttujia on näkyvissä tietylle aliohjelmalle tai mitä toisia aliohjelmiä tietty aliohjelma kutsuu. Nämä tiedot käyvät ilmi vain lähdekoodista, eikä niitä voida varmasti päätellä ajonaikaisesta käyttäytymisestä. Suurten ohjelmien kohdalla tällaisten tietojen etsimiseen tarvitaan automaattisia työkaluja.

Dynaamisessa analyysissä tarkasteltavaa ohjelman käyttäytymistä voidaan tutkia pitkälti myös staattisen analyysin kautta eli lukemalla koodia. Dynaamisessa analyysissä tutkitaan muun muassa lauseita, jotka suoritetaan tietyn muuttujan arvon tuottamiseksi tai käyttävät tietyn muuttujan arvoa sekä muuttujia, joiden arvoa käytetään laskettaessa tietyn muuttujan arvoa. Näillä tiedoilla voidaan esimerkiksi etsiä aiheuttaja virheeseen, jossa ohjelma tulostaa väärän arvon. Suurten ohjelmien tapauksissa on hyötyä automaattisista työkaluista, jotka esimerkiksi etsivät sellaiset lauseet, jotka suoritetaan tulosarvon tuottamiseksi. [Gopal & Schach 1989.]

Dynaaminen analyysi on erityisen tärkeää oliopohjaisissa kielissä, joissa luodaan ja tuhotaan olioita ajonaikaisesti. Lisäksi sitä tarvitaan valmiiden komponenttien tarkasteluun, kun niiden lähdekoodi ei ole saatavilla. [Systä 2000.]

Instrumentoinnissa lähdekoodiin lisätään erilaisia ajonaikaista käyttäytymistä tutkivia antureita, esimerkiksi laskureita, jotka laskevat tietyn lauseen suorituksia. Instrumentoinnilla voidaan tutkia esimerkiksi olioiden luomista, niiden välisiä operaatiokutsuja ja attribuuttien arvojen muuttumista. [Gergeleit 1994.]

3.3 Ylläpidon ongelmia

Tavallisia ongelmia

Ylläpitotoimissa on kiinnitettävä huomiota siihen, miten koodin tiettyyn kohtaan tehdyt muutokset vaikuttavat muun koodin toimimiseen. Tyypillistä on, että muutostyö aiheuttaa virheen, joka huomataan myöhemmin. Virhettä korjattaessa saattaa syntyä lisää virheitä. Tätä kutsutaan muutostarpeiden väreilyvaikutukseksi.

Ylläpitoa vaikeuttavia tekijöitä ovat koodin huono rakenne, ylläpitäjien riittämättömät tiedot sovelluksesta, riittämätön dokumentaatio ja ylläpitotyön huono maine.

Huonorakenteisen koodin ongelmia ovat pitkät aliohjelmat, tunnusten epäyhtenäinen nimeämiskäytäntö, monimutkaiset moduulit, saavuttamaton koodi, syvät sisäkkäiset valintalauseet, tiukasti kytköksissä olevat moduulit ja irrallisista toisiinsa liittymättömistä

osista koostuvat moduulit. Lisäksi jos ylläpitäjällä ei ole riittävää tietämystä sovelluksesta, ei ylläpitäjä voi saada käsitystä ohjelman toiminnasta edes lukemalla sen lähdekoodia, jos se on huonorakenteista.

Ylläpitäjän on valitettavasti asennoiduttava ohjelmasta olemassa olevaan dokumentaatioon niin, ettei muuhun kannata luottaa kuin lähdekoodiin. Dokumentointi on ohjelmistoprojektien tehtävien tärkeysjärjestyksessä viimeinen tehtävä. Dokumentit jäävät myös usein päivittämättä ylläpitotöiden yhteydessä.

Ylläpitotyötä vieroksutaan; työssä joudutaan tekemisiin tyytymättömän asiakkaan kanssa ja tarkastelemaan toisten kirjoittamaa koodia. Siksi ylläpitotehtävät annetaan useimmiten harjoittelijoille ja muille kokemattomammille ohjelmoijille. Kuitenkin ylläpitotehtävissä tarvittaisiin hyvin monenlaisia ja laaja-alaisia taitoja. Virheiden etsinnässä ja korjaamisessa tarvitaan tietoa koko järjestelmän toiminnasta. Lisäksi jos ylläpitotoimiin kuuluu uuden ominaisuuden toteuttaminen, joudutaan ohjelman määrittely- ja testausvaiheet käymään läpi uudelleen, mikä niin ikään vaatii syvää tietämystä. [Van Vliet 1993.]

Ratkaisuja ongelmiin

Korjaavan ylläpidon kustannuksia vähennetään näkemällä toteuttamisvaiheessa vaivaa koodin luettavuuden ja dokumentaation kattavuuden eteen. Täydellistävän ylläpidon tehtäviä ilmenee vähemmän, kun asiakas osallistuu ohjelmiston analyysi- ja suunnitteluvaiheisiin. Mukauttavalla ylläpidolla ennakoidaan tulevia muutoksia. Lisäksi ylläpitotehtäviä vähennetään välttämällä toistoa koodissa.

Ohjelman ylläpidettävyyttä voidaan parantaa hallinnollisilla toimilla. Voidaan ottaa käyttöön yhtenäiset tavat suunnittelussa, ohjelmoinnissa ja dokumentoinnissa. [Van Vliet 1993.]

3.4 Ylläpitotyö

Muutosvaikutusanalyysi

Ennen muutoksen tekemistä tehdään muutosvaikutusanalyysi, jossa muutoksen vaikutukset muihin järjestelmiin, dokumentteihin, tietokantoihin tai käyttöliittymiin tutkitaan ennakoon. Tietoja saadaan vaatimus- ja suunnitteludokumenteista, jos ne ovat ajantasaisia. Dokumentteja tarkasteltaessa kiinnitetään erityistä huomiota ohjelmien ja moduulien rajapintoihin. Löydettyäessä muutokselle altis ohjelma tai moduuli tulee myös tutkia niiden ulkopuoliset yhteydet.

Muutos saattaa esimerkiksi edellyttää uuden tietokentän lisäämisen, mikä saattaa vaikuttaa tietorakenteisiin, tiedostoihin ja tietokantoihin. Tietoalkioiden kautta muutos voi myös vaikuttaa toisiin järjestelmiin. Kaikki samaa tietokantaa käyttävät järjestelmät joudutaan ehkä muuttamaan.

Muutoksen myötä järjestelmän dokumentaatiokin tulee päivittää. Käyttöliittymän syötteiden formaatit, antamistavat, tulostusmuodot ja raportit voivat myös muuttua. Käyttöohjeetkin saatetaan joutua päivittämään. Selvitetyt muutoksen vaikutukset kirjataan muutospyyntödokumenttiin, ja niiden perusteella laaditaan työmääräarvio, jonka laatimiseen on olemassa erilaisia analyysimenetelmiä ja metriikoita hyödyntäviä systemaattisia tapoja. [Ajila 1995; Arthur 1988.]

Korjaukset

Korjaavaa ylläpitoa tarvitaan, kun ohjelma ei toimi, ohjelma kaatuu, sen tuottamat tulokset eivät ole vaatimusmäärittelyjen mukaisia, ohjelma ei noudata dokumentaatiota tai käyttöohjeet ovat harhaanjohtavat. Korjaukseen liittyy riski, että syntyy uusi virhe; usein korjauksesta syntyy uusi virhe johonkin ohjelman toiseen osaan, mikä jää huomaamatta kehittäjiltä, jos testataan vain ohjelman muutettua osaa.

Kiireellisiä korjauksia ovat sellaiset, jotka kohdistuvat esimerkiksi ihmisen henkeä uhkaaviin ohjelmistovirheisiin, suuria taloudellisia menetyksiä aiheuttaviin virheisiin ja virheisiin, jotka estävät suuren ihmismäärän työnteon. Tällaisissa tapauksissa korjausten

suunnittelu- ja muutosvaikutusanalysointivaiheet sivuutetaan ja edetään suoraan toteutusvaiheeseen. Kun kiireelliset muutokset on tehty, niihin palataan takaisin ja niiden toteutus varmistetaan oikeaksi, ja dokumentaatioita päivitetään tarvittaessa.

Ei-kiireellisiä virheitä ovat esimerkiksi jonkin harvoin tarvittavan komponentin toimintaa haittaavat virheet ja väärin tulostuvat raportit, jos niitä ei tarvita vähään aikaan. Ennen korjauksen aloittamista on syytä varmistua ongelmasta ja siitä, että se on ymmärretty oikein. Tutkitaan, mistä ongelma syntyy; selvitetään, missä ja milloin ongelma ilmenee ja myös missä ja milloin se ei ilmene. [Arthur 1988.]

Virheen etsinnässä tarkastellaan määrittely- ja suunnitteludokumentteja, koodia, syötteitä ja tulosteita. Apuna voidaan käyttää myös virheenetsintäohjelmaa. Aluksi on hyödyllistä etsiä yleisimpiä virhetyppejä; näitä ovat esimerkiksi ohjelman logiikan virheet, kuten puuttuvat vaihtoehdot valintalauseissa ja väärä toistolauseen toistomäärä. Laskennassa on saatettu käyttää väärää operaattoria. Rajapintoihin liittyy usein tiedostojen siirtoon liittyviä virheitä, jos siirtoihin liittyy tietojen muuntamista. Tietokantoihin saattaa liittyä virheitä, jotka seuraavat sen vääränlaisesta päivittämisestä, laitteistoviasta, tai esimerkiksi siitä, että se on voitu palauttaa väärästä varmuuskopiosta.

Pienkehitys

Pienkehitys on uusien ominaisuuksien lisäämistä ohjelmaan. Yleisiä pienkehitystarpeita ovat esimerkiksi joidenkin toimintojen automatisointi ja järjestelmän laajentaminen. Yritysten välinen kilpailu ja lakien ja standardien muutokset voivat myös synnyttää lisäystarpeita.

Ennen muutoksen tekemistä tutkitaan, miten muutos sopii vaatimusmäärittelyyn ja mitä vaatimuksia muutokseen liittyy. Muutospyyntödokumentti käydään läpi, arvioidaan, mitä rajoituksia muutoksella on, ja tehdään työmääräarvio. Lisäyksen vaikutukset muuhun ohjelmaan sekä tietokantoihin, tiedostoihin ja tietorakenteisiin arvioidaan. Muutoksen tekemisessä noudatetaan järjestelmän nimeämiskäytäntöä.

Suunnitteludokumentteja tutkitaan ja selvitetään, mitä aliohjelmaa täytyy lisätä, muuttaa tai poistaa. Järjestelmän dokumentaatio päivitetään.

Ominaisuuksia lisättäessä on kiinnitettävä huomiota järjestelmän rajoituksiin ja siihen, sopiiko lisäys yhteen olemassa olevan ohjelman kanssa. Tehtävä on sitä helpompi mitä paremmin ohjelma on suunniteltu.

4 Työtapaturmavakuutuksen tarjous- ja siirtojärjestelmän määrittelyt

4.1 Taustaa

Työtapaturma- ja ammattitautivakuutus on Suomen lainsäädännöllinen vakuutus, joka korvaa työntekijöille heille työssä, työntekopaikan alueella tai työmatkalla sattuneet tapaturmat ja työnteosta johtuvat sairaudet. Laki velvoittaa työnantajan vakuuttamaan työntekijänsä. [Työtapaturma- ja ammattitautilaki 2015; Vakuuttamisvelvollisuus 2018.]

Vakuutusyhtiöt kilpailevat asiakasyrityksistä, jotka ostavat niiltä työtapaturma- ja ammattitautivakuutuksia. Vakuutusyhtiöt lähettävät asiakkaille tarjouksia, ja asiakkaat irtisanoivat vakuutussopimuksia ja solmivat uusia parhaan tarjouksen tekevien yhtiöiden kanssa. Nämä prosessit ovat moniasteisia ja vaativat runsaasti käsin tehtävää työtä.

Työtapaturmavakuutuksen tarjous- ja siirtojärjestelmä on sovellus, jonka osti vakuutusyhtiö, joka myy lain mukaisia vakuutuksia yrityksille. Sovellus automatisoi edellä mainittuja vakuutusyhtiöiden kilpailuun liittyviä prosesseja. Työ tilattiin, koska yhtiön aiemmin käyttämän työtapaturma- ja ammattitautivakuutustarjouslaskurin käyttämän teknii- kan tukeminen lopetettiin ja laskurin toiminta ei enää ollut varmaa. Aiemman tarjousjärjestelmän käyttämän tietokannan tuki lopetetaan lähitulevaisuudessa, ja aiempi siirtojärjestelmä oli huono. [Työtapaturmavakuutuksen tarjous- ja siirtojärjestelmän projektisuunnitelma 2017.]

4.2 Projektin tavoitteet ja asiakkaan vaatimukset

Projektin tavoitteena oli tehostaa senhetkistä työtapaturmavakuuttamisen tarjous- ja siirto- prosessia niin, ettei tarjoustusta laskiessa jouduttaisi syöttämään samoja tietoja useisiin järjestelmiin, vaan tiedot voitaisiin syöttää yhteen järjestelmään, jossa myös lähetettäisiin

tarjoukset ja siirrettäisiin vakuutukset. Arvioitiin, että prosesseihin kuluva aika saataisiin vähenemään kolmannekseen, kun ne voitaisiin suorittaa yhdessä järjestelmässä usean sijaan.

Sovelluksen piti osata automaattisesti hakea yrityksen tiedot erillisestä asiakastietojärjestelmästä käyttäjän syöttämän yritystunnuksen perusteella. Sovelluksen piti pystyä laskemaan tarjous asiakasyrityksen tilastojen ja yhtiön liiketoimintasääntöjen perusteella.

Yrityksen tilastot tuli pystyä hakemaan napsauttamalla painikkeesta, joka lähettäisi kyselyn erilliseen järjestelmään, joka lähettäisi vastauksen tietyn ajan kuluessa. Samaan tapaan tuli pystyä lähettämään sovelluksesta tarjouskirjeitä asiakkaille ja asiakkaiden vakuutusten irtisanomispyyntöjä muille vakuutusyhtiöille sekä siirtämään vakuutuksia muilta yhtiöiltä, jos asiakas hyväksyi kilpailevan yhtiön tarjouksen. [Työtapaturmavakuutuksen tarjous- ja siirtojärjestelmän projektisuunnitelma 2017.]

4.3 Käyttöliittymä

Työtapaturmavakuutuksen tarjous- ja siirtojärjestelmän käyttöliittymä on niin sanottu tarjousvelho, joka johdattaa käyttäjän täytettävän lomakkeen läpi. Lomakkeen tietoja käytetään vakuutustarjouksen laskemiseen. Laskettu tarjous voidaan sitten lähettää asiakkaalle, ja jos asiakas hyväksyy tarjouksen, vakuutus tulee voimaan automaattisesti. Käyttöliittymän vasemmassa laidassa näkyvässä navigaatiopalkissa näkyy velhon tila (kuva 1).

Haku-sivulla (kuva 1) käyttäjä syöttää hakukenttään yritystunnuksen, jolloin järjestelmä etsii automaattisesti erillisestä asiakastietojärjestelmästä yrityksen nimen, markkina-alu-

eet ja osoitteen. Lisäksi alla näkyy luettelo yritystunnuksella olemassa olevista kesken-eräisistä ja valmiista tarjouksista. Koelaskelma-painike käynnistää tarjousvelhon niin, ettei sitä tallenneta tähän listaan.

Työtapaturmavakuutuksen tarjous- ja siirtojärjestelmä 30.01.2019 Ohje

HAKU

Hae asiakkaan tiedot asiakastietojärjestelmästä tarjouksen/hakemuksen tekemistä varten

ASIAKASNUMERO / Y-TUNNUS / HENKILÖTUNNUS

629834 **HAE** **KOELASKELMA**

LÄHTÖTIEDOT

Kuhmon Peltityö Ky

0629834-6

TYÖNTEKIJÄT

43910, Kattorakenteiden asennus ja kattaminen

YRITTÄJÄT

Tuottajantie 18, 88600, SOTKAMO

LUO UUSI TARJOUS

Valitse joko kokonaan uusi tyhjä tarjouspohja, päivitä aikaisempaa tarjousta tai hyväksy tulostettu tarjous:

Tarjousnumeri	Muutosaika	Tarjouksen tila	Tilan tarkennin	Toiminnot
T51-0002401	21.01.2019 1	Tulostettu tarjous		NÄYTÄ PÄIVITÄ

Kuva 1. Haku-sivu [Työtapaturmavakuutuksen tarjous- ja siirtojärjestelmä 2019].

Lähtötiedot-sivulla (kuva 2) syötetään tarjouksen kieli ja määritetään alkamisvuosi sekä vapaaehtoinen myyjätunnus. Sivulla kerrotaan sovellukselle, onko yritys kunta, seurakunta tai kuntayhtymä. "Siirto / uusi vakuutus / nykyinen asiakas" -kentässä "siirto" tarkoittaa, että tarjous on asiakkaalle, joka on jo ostanut työtapaturmavakuutuksen toiselta yhtiöltä ja jonka asiakkuus halutaan viedä kilpailevalta yhtiöltä. Asiakas siis irtisanoisi vakuutussopimuksensa senhetkisen vakuutusyhtiön kanssa ja solmisi toisen vakuutus-

yhtiön kanssa uuden, jolloin vakuutus ”siirtyisi” vakuutusyhtiöltä toiselle. ”Uudessa vakuutuksessa” asiakkaalle tarjotaan uutta vakuutusta. ”Nykyinen asiakas” valitaan, kun yritys on jo yhtiön asiakas.

Työtaturmavakuutuksen tarjous- ja siirtojärjestelmä 30.01.2019 Ohje

Asiakkaan nimi: Kuhmon Peltityö Ky **Y-tunnus:** 0629834-6

TARJOUKSEN KIELI
Suomi

SIIRTO / UUSI VAKUUTUS / NYKYINEN ASIAKAS
Uusi vakuutus

ONKO KYSEESSÄ KUNTA, SEURAKUNTA TAI KUNTAYHTYMÄ?
Ei

VAKUUTUKSEN ALKAMISVUOSI
2019

MYyjätunnus (täytävä vain jos teet tarjouksen toisen henkilön puolesta)
[input field] **HAE**

MYyjän nimi
[input field]

PERUUTA **SEURAAVA**

Kuva 2. Lähtötiedot-sivu [Työtaturmavakuutuksen tarjous- ja siirtojärjestelmä 2019].

Työntekijät-sivulla (kuva 3) lisätään vakuutettavat työntekijät. Kunkin työntekijän kohdalla täytetään ammattinimike ja palkka ja painetaan Lisää-painiketta. Työntekijöitä voidaan lisätä rajaton määrä.

Työtapaturmavakuutuksen tarjous- ja siirtojärjestelmä 30.01.2019 Ohje

TYÖNTEKIJÄT

Asiakkaan nimi: Kuhmon Peltityö Ky Y-tunnus: 0629834-6 Tarjousnumero: T51-0002498-J

Lisätietoja ammattiluokituksesta ja määrittelmistä löydät LuokitusEksperdistä

KIRJOITA AMMATTINIMIKE TAI KOODI

PALKKASUMMA €

TYHJENNÄ KAIKKI LISÄÄ

Ammattikoodi	Ammattinimike	Palkkasumma	To
25140-001	atk-ohjelmoija	40 000,00 €	

Yhteensä: 40 000,00 €

Kuva 3. Työntekijät-sivu [Työtapaturmavakuutuksen tarjous- ja siirtojärjestelmä 2019].

Yrittäjät-sivulla (kuva 4) lisätään yrittäjän vakuutuksia. Painetaan Lisää-painiketta, joka tuottaa dialogin, johon täytetään vakuutettavan yrittäjän tiedot. Dialogin jälkeen yrittäjä ilmestyy Yrittäjät-sivulla näkyvään listaan.

Työtapaturmavakuutuksen tarjous- ja siirtojärjestelmä 30.01.2019 Ohje

YRITTÄJÄT

Asiakkaan nimi: Kuhmon Peltityö Ky Y-tunnus: 0629834-6 Tarjousnumero: T51-0002498-J

Valitse listalta yrittäjät, joille tarjotaan vapaaehtoisia yrittäjän vakuutuksia (turvaa voidaan tarjota vain YEL-vakuutetuille)

Omistaja	Hetu	Omistusosuus	Toiminnot
No data to display			

Lisää uusi yrittäjä, jolle vakuutusturva halutaan

EDellinen **LISÄÄ** **SEURAAVA**

Kuva 4. Yrittäjät-sivu [Työtapaturmavakuutuksen tarjous- ja siirtojärjestelmä 2019].

Yhteenveto-sivulla (kuva 5) näkyy tarjouksen yhteenveto. Tarjous voidaan tulostaa Tulosta-painikkeesta. Viimeiselle sivulle johtava Seuraava-painike aktivoituu, kun tarjouksen kaikki pakolliset tiedot on täytetty edellisillä sivuilla.

☰

Työtapaturmavakuutuksen
tarjous- ja siirtojärjestelmä

30.01.2019
Ohje

HAKU

LÄHTÖTIEDOT

TYÖNTEKIJÄT

YRITTÄJÄT

YHTEENVETO

HYVÄKSYMINEN

Asiakkaan nimi: Kuhmon Peltityö Ky Y-tunnus: 0629834-6 Tarjousnumero: T51-0002498-J

Yhteenveto valituista turvista sekä hinnoista

Työntekijät		
Vakuutus	Palkkasumma	Vakuutusmaks
▶ Työtapaturmavakuutus		
Työntekijäin ryhmähenkivakuutus		
Vapaa-ajan ryhmävakuutus		
Yksilöllinen vapaa-ajan vakuutus (ei urheilurajoituksia),		
	Yhteensä	

Tuotekortit, avaintietoasiakirjat ja muut myynnintukimateriaalit löydät [täältä](#)


EDELLINEN

TULOSTA

SEURAAVA

Kuva 5. Yhteenveto-sivu [Työtapaturmavakuutuksen tarjous- ja siirtojärjestelmä 2019].

metropolia.fi



Hyväksyminen-sivulla (kuva 6) voidaan valita, mitkä tarjouksen osat hyväksytään. Sivulla täydennetään vakuutuksen siirtoilmoituksen allekirjoittajan nimi sekä muita tietoja. Hyväksy-painikkeesta napsauttamalla tarjous hyväksytään ja vakuutus saatetaan voimaan.

☰

Työtapaturmavakuutuksen
tarjous- ja siirtojärjestelmä

30.01.2019
Ohje

HAKU

LÄHTÖTIEDOT

TYÖNTEKIJÄT

YRITTÄJÄT

YHTEENVETO

HYVÄKSYMINEN

Asiakkaan nimi: Kuhmon Peltityö Ky Y-tunnus: 0629834-6 Tarjousnumero: T51-0002240-7

Valitut vakuutusturvat

☒ TYÖTAPATURMAVAKUUTUS

☒ TYÖNTEKIJÄIN RYHMÄHENKIVAKUUTUS

☒ VAPAA-AJAN RYHMÄVAKUUTUS

☒ YRITTÄJÄN TYÖ- JA VAPAA-AJAN VAKUUTUS (EI URHEILURAJOITUKSIA) , KORHONEN PENTTI (010863-1152)

MAKSUERIEN LKM. JA MAKSUTAPAKOROTUS (TYÖNTEKIJÄT)

1 (0,00 %)

MAKSUERIEN LKM. JA MAKSUTAPAKOROTUS (YRITTÄJÄT)

1 (0,00 %)

SIIRTOILMOITUKSEN ALLEKIRJOITTAJAN NIMI

SIIRTOILMOITUKSEN ALLEKIRJOITUSPÄIVÄ

NYKYINEN VAKUUTUSYHTIÖ

SIIRTOPÄIVÄ/VAKUUTUKSEN ALKAMISPÄIVÄ

YRITTÄJÄN VAKUUTUSTEN ALKAMISAIKA

DOKUMENTOI ALLEKIRJOITETTU TARJOUS

LISÄÄ TIEDOSTO

HUOM! TÄYTETÄÄN VAIN ERIKOISMAKSUISILLE. VALITUN VAIHTOEHDON ENNAKKOMAKSUPROM

MYYNITAPA

Yksintyö

MYYJÄTUNNUS 1

000000

MYYJÄTUNNUS 1. JAKO %


100

EDELLINEN

HYVÄKSY

Kuva 6. Hyväksyminen-sivu [Työtapaturmavakuutuksen tarjous- ja siirtojärjestelmä 2019].

metropolia.fi



4.4 Haasteet

Sovelluksen ylläpitotyöhön liittyi joitakin haasteita. Asioita, joihin tuli perehtyä, oli paljon, sillä järjestelmä oli laaja eikä sen teossa oltu mukana, ennen kuin järjestelmää oli kehitetty jo kaksi vuotta. Alkuperäiset ohjelmoijat olivat kiireisiä, eikä heillä ollut aina aikaa vastata kysymyksiin.

Kehitysympäristön asennus oli monimutkainen prosessi, joka vaati käyttöoikeuksia useisiin palveluihin sekä lukuisia paikallisten asetusten muutoksia. Asennusta varten jouduttiin järjestämään useita noin tunnin tapaamisia eri asiantuntijoiden kanssa. Näissä tapaamisissa ilmeni niin ikään ongelmia, joita asiantuntijat joutuivat selvittämään.

Virheiden etsiminen oli aikaavievää, sillä järjestelmän toteuttamisessa käytetty rakenne ja tekniikka eivät olleet tuttuja ja niihin piti perehtyä huolellisesti sekä staattisen että dynaamisen analyysin kautta, ennen kuin virheen paikantaminen saatettiin aloittaa. Ongelmia tuotti myös erilaisten valmiiden ulkoasukomponenttien dokumentaation huono saatavuus verkossa.

Testiympäristössä taustajärjestelmän yhteys käyttöliittymään oli epävakaa, mikä vaikeutti muutosten testaamista. Jos esimerkiksi tehtiin muutoksia jollakin muulla velhon sivulla kuin Haku-sivulla, ei muutosten testaaminen onnistunut, jos asiakkaan haku Haku-sivulla ei toiminut.

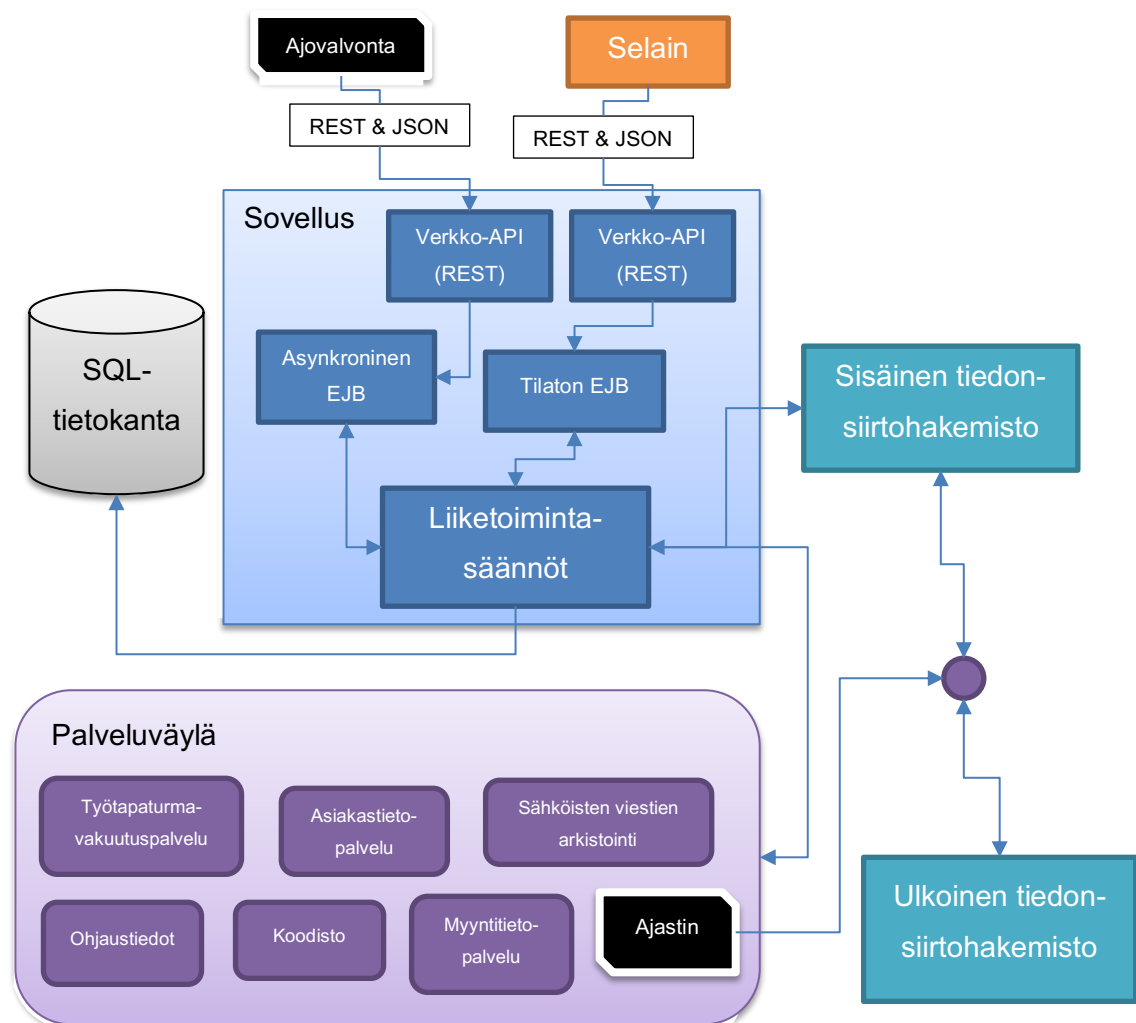
5 Työtapaturmavakuutuksen tarjous- ja siirtojärjestelmän toteutus

5.1 Ohjelman arkkitehtuuri

Työtapaturmavakuutuksen tarjous- ja siirtojärjestelmän arkkitehtuuri on esitetty kuvassa 7. Selaimessa toimiva käyttöliittymä liittyy REST-mallin mukaisilla HTTP-kutsuilla Java-kielellä ohjelmoituun verkko-ohjelmointirajapintaan, joka taas liittyy tilattoman Enterprise JavaBeans -rajapinnan kautta taustajärjestelmään, joka sisältää vakuutusyhtiön liiketoimintasäännöt, joiden perusteella tarjoukset lasketaan. Luodut tarjoukset tallentuvat SQL-tietokantaan.

Käyttöliittymästä lähetetyt irtisanomis- ja tilastopyynnöt tallentuvat sisäiseen siirtohakemistoon, josta ne lähetetään ulkopuolisen yhtiön hallinnoimaan siirtohakemistoon ja jonne niiden vastaukset sitten lähetetään tietyin väliajoin. Näistä eräajoista kirjataan lokeja, joita ajovalvonta valvoo verkko-ohjelmointirajapinnan välityksellä, joka on liitoksissa taustajärjestelmään asynkronisen Enterprise JavaBeans -rajapinnan kautta.

Palveluväylän kautta sovellus hakee tietoja erilaisista palveluista. Nämä palvelut ovat työtaturmavakuutuspalvelu, ohjaustietopalvelu, asiakastietopalvelu, koodistopalvelu, sähköisten viestien arkistointipalvelu ja myyntitietopalvelu. [Työtaturmavakuutuksen tarjous- ja siirtojärjestelmän arkkitehtuurikuvaus 2018.]



Kuva 7. Työtaturmavakuutuksen tarjous- ja siirtojärjestelmän arkkitehtuuri [Työtaturmavakuutuksen tarjous- ja siirtojärjestelmän arkkitehtuurikuvaus 2018].

5.2 Angular-ohjelmistokehys

Angular on Googlen ylläpitämä ohjelmistokehys. Sen keskeinen idea on laajentaa HTML-kieltä niin sanotuilla verkkokomponenteilla, jotka sisältävät uusia toiminnallisuuksia. Nämä komponentit liitetään HTML:n attribuutteihin tai elementteihin. Jokaista komponenttia vastaa malline, ja näiden mallineiden data renderöidään HTML-dokumenttiin kirjoitettavien kaksoisaaltosulkeiden "{ " ja " }" sisään kirjoitetuilla lauseilla. Ohjelmistokehysten keskeisiin ominaisuuksiin kuuluvat kaksisuuntainen datakytkentä, mallinnus, reitittäminen, komponentit ja riippuvuusinjektio. Angularin ohjelmointikieli on TypeScript, joka on Microsoftin JavaScript-laajennus. Angular tarjoaa TypeScript-kirjastoja tavallisten käyttöliittymätoiminnallisuuksien toteuttamiseen. [Chiaretta 2018; Architecture Overview 2019.]

Työtaturmavakuutuksen tarjous- ja siirtojärjestelmän käyttöliittymä on Angularilla rakennettu yhden sivun sovellus. Lisäksi on käytetty kolmannen osapuolen kirjastoja, kuten PrimeNG-kirjastoa. Angularin komponentteja käytetään näkyminä, jotka vaihtuvat käyttäjän toimien mukaan. Esimerkiksi tarjousvelhon sivut ovat omia näkymiään.

Tarjousvelho on toteutettu Arch wizard -kirjastolla. Kirjasto tarjoaa valmiin mallin velhon navigaatiopalkille ja velhon sivuille ja funktiot velhossa navigoimiseen. Navigointipalkin ja sivujen sisällön asettelu ovat muokattavissa.

Tarjousvelhon päivämääräkentät käyttävät PrimeNG-kirjaston kalenterikomponenttia. Päivämääräkentässä on pieni kalenteripainike, jota napsauttamalla saadaan esiin kuu-kausinäkymällinen kalenteri, josta voidaan napsauttaa haluttua päivää. Päivämäärä voidaan myös kirjoittaa kenttään näppäimistöllä. Kalenteriin voidaan asettaa päivämäärän esitysmuoto attribuutilla (esimerkkikoodi 28).

```
<p-calendar name="VakuutuksenAlkamispaiva"
  [(ngModel)]="step6Form.VakuutuksenAlkamispaiva"
  [inputStyleClass]="input" showIcon="true" dateFormat="dd.mm.yy"
  [locale]="calendarLocalizationFi"
  [ngClass]="!offerIsEditable ? 'input--disabled' : ''"
  [disabled]="!offerIsEditable" utc="true">
</p-calendar>
```

Esimerkkikoodi 28. PrimeNG-kalenteri HTML-koodissa [Työtaturmavakuutuksen tarjous- ja siirtojärjestelmän lähdekoodi 2019].

Ohjelmaa voidaan käyttää suomeksi tai ruotsiksi. Tämä on toteutettu Angularin kansainvälistämistyökalulla. Ohjelmassa eri kielillä näytettävä teksti kirjoitetaan eri kielille tarkoitettuihin JSON-tiedostoihin, ja itse HTML-sivuilla näihin teksteihin viitataan niiden tunnuksilla. Globaali kielimuuttuja määrää, minkä JSON-tiedoston teksti näytetään.

Tietokentät, joissa on automaattinen täyttöfunktio, on toteutettu PrimeNG-kirjaston automaattitäyttökomponenteilla. Komponentin parametriksi asetetaan automaattitäyttöön käytettävät sanat sisältävä taulu ja tauluun kohdistettujen hakujen intervalli sekä kriteerit.

5.3 Pienkehitys- ja virheenkorjaustöitä

Vakuutuksen siirron tekeminen mahdolliseksi loppuvuodesta

Lähtötiedot-sivulla (kuva 2) valittiin, liittyikö tarjous siirtoon toiselta yhtiöltä, oliko kyseessä uusi vakuutus vai oliko kyseessä uusi tarjous yritykselle, joka oli jo yhtiön asiakas. Nykyistä vuotta ei olisi pitänyt olla mahdollista valita kesäkuun jälkeen, jos kyseessä oli siirto.

Muutettavia kohteita olivat käyttöliittymä ja taustajärjestelmä. Käyttöliittymään tarvittiin uusi palvelukutsu ja taustajärjestelmään päivämäärätarkistus. Muutosvaikutusanalyysissä todettiin, etteivät muutokset aiheuttaisi lisämuutoksia ohjelman muihin osiin. Dokumentaatio päivitettiin.

Taustajärjestelmään toteutettiin logiikka, jolla määriteltäisiin vakuutukselle valittavissa olevat alkamisvuodet. Vaihtoehdot olivat nykyinen vuosi ja ensi vuosi, paitsi jos tehtiin vakuutuksen siirtoa ja siirtoajankohta olisi heinäkuussa tai myöhemmin, jolloin ensi vuosi olisi ainoa vaihtoehto. Käyttöjärjestelmä ohjelmoitiin hakemaan alkamisvuodet taustajärjestelmästä ja päivittämään valittavissa olevat vuodet joka kerta, kun tarjouksen konteksti muutettaisiin. Näin käyttöliittymän rooli käyttäjän ja ohjelman kommunikaation välineenä ja taustajärjestelmän rooli logiikan haltijana säilyisi.

Dialogi-ikkunan ulkoasuvirhe

Yrittäjät-sivulla (kuva 4) tarjoukseen voitiin lisätä yrittäjän vakuutuksia erillisessä dialogi-ikkunassa. Erityistä päänvaivaa tuotti virhe, jossa käyttöliittymän globaali ulkoasu ei vaikuttanut ikkunaan. Ikkuna ei myöskään reagoinut sen CSS-tiedostoon kirjoitettuihin tyyliohjeisiin. Kun dialogi-ikkunaa verrattiin ohjelman muihin ikkunoihin, jotka toiset ohjelmoijat olivat ohjelmoineet ja joissa ulkoasu toimi, huomattiin, että näiden CSS-tiedostot olivat tyhjiä.

Asiaa tutkittiin useita tunteja niin staattisen kuin dynaamisen analyysin keinoin, kunnes huomattiin, että dialogi-ikkunoiden tyylit määriteltiin TypeScript-koodissa ikkunoiden luontivaiheessa (esimerkkikoodi 29). Toisin kuin kaikki muut komponentit, dialogi-ikkunat saivat ulkoasunsa konstruktorin parametrina. Parametri lisättiin ikkunan konstruktoriin, ja ikkunan ulkoasu korjaantui.

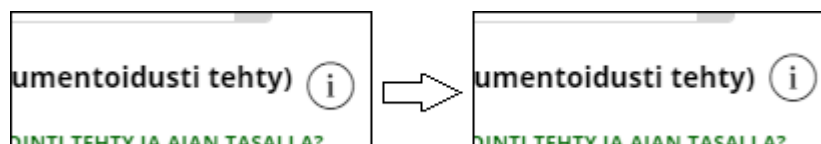
```
let modalInfo = this.modalCtrl.create('ModalInfo', {},
  {cssClass: this.theme});
```

Esimerkkikoodi 29. Dialogi-ikkunan CSS-luokka määritellään sen konstruktorissa [Työtapaturmavakuutuksen tarjous- ja siirtojärjestelmän lähdekoodi 2019].

Virheen paikantamista haittasi poikkeaminen käyttöliittymäkerrosten löyhästä kytkemisestä. Ulkoasuun liittyvä koodi on tyypillisesti kokonaan CSS-tiedostoissa, mutta dialogin tyyli tietoja asetettiin TypeScript-tiedostossa.

Kuvakkeen keskittäminen tekstin suhteen

Työntekijät-sivulla (kuva 3) syötettiin vakuutettavat työntekijät. Sivulta löytyi lisätietokuvake, jota napsauttamalla saatiin esiin lisätietoa työturvallisuuskartoituksesta. Kuvake tuli keskittää sen vieressä näkyvän tekstin kanssa (kuva 8).



Kuva 8. Lisätietokuvakkeen keskittäminen tekstirivin suhteen.

Kuvakkeen löytäminen HTML-koodista vaati vain kevyen staattisen analyysin. Tekstin ja kuvakkeen ympäröivän elementin class-attribuutille asetettiin CSS-luokka, joka keskitti elementit vaakasuunnassa toistensa suhteen CSS flexible box layout -ulkoasumallin avulla. Korjaus oli vaivaton, sillä sivu oli toteutettu noudattaen kerrosten löyhää kytkentää ja muita ylläpidettävän koodin käytänteitä, jolloin etsittävät ominaisuudet löytyivät oletetuista paikoista. Korjauksessa noudatettiin niin ikään löyhän kytkennän periaatetta käyttämällä class-attribuuttia sekä CSS-luokkaa (esimerkkikoodi 30).

CSS-tiedostossa:

```
.center {
    display: flex;
    align-items: center;
}
```

HTML-dokumentissa:

```
<ion-row class="center rowDifferentSectionMargin">
  <strong>
    {{ "WizardQuoteSystem.Step3.OccupationalSafetyAndHealthSurvey"
      | translate }} &#160;
  </strong>
  <ion-icon name="ios-information-circle-outline" class="infoIcon"
    (click)="openModal()"></ion-icon>
</ion-row>
```

Esimerkkikoodi 30. Tietokuvakkeen keskittämisen toteutus [Työtaturmavakuutuksen tarjous- ja siirtojärjestelmän lähdekoodi 2019].

Koelaskelmaominaisuus

Eräs pienkehitystehtävä oli koelaskelmaominaisuuden toteuttaminen. Määrittelyn mukaan Haku-sivulla (kuva 1) Koelaskelma-painikkeesta päästäisiin koelaskelmaan. Kun tarjous luotaisiin ja sen tietoja alettaisiin täyttää, tarjous tallentuisi ykkössivulla näytettävään tarjousten listaan. Koelaskelmaa taas käytettäisiin tarjouksen laskemiseen niin, että se ei tallentuisi listaan.

Painiketta painettaessa ilmestyisi dialogi, johon tulisi syöttää joitakin tietoja, minkä jälkeen velho käynnistyisi tallentamatta tarjousta. Dialogi-ikkunassa piti olla kenttä yrityksen toimialalle ja luottoluokitukselle. Kun kenttään alettaisiin kirjoittaa, tulisi sen näyttää reaaliajassa tekstiä vastaavia toimialoja tai luottoluokituksia (kuva 9).

Kuva 9. Koelaskelma [Työtapaturmavakuutuksen tarjous- ja siirtojärjestelmä 2019].

Muutoksen vaatimusmäärittelyt, suhde yleisiin vaatimusmäärittelyihin ja muutoksen rajoitukset tutkittiin, ja arvioitiin, ettei ominaisuuden lisäykseen liittyisi ongelmia.

Koelaskelmasta luotiin uusi komponentti, joka instantioitaisiin hakusivulla. Koodin kohta, johon instantiointi lisättiin, löytyi vaivattomasti käyttöliittymän koodin loogisen rakenteen ansiosta. Toimiala- ja luottoluokituskenttien reaaliaikahaku toteutettiin lyhyin intervallien suoritettavilla hauilla tauluun, johon oli haettu kaikki toimialat erillisestä tietokannasta. Tämän ansiosta jos tulevaisuudessa syntyisi uusi toimiala, jouduttaisiin muuttamaan ai-noastaan tietokantaa eikä käyttöliittymään tai muihin ohjelman osiin tarvittaisi muutoksia.

Itsetäydentyvien kenttien toiminnallisuutta testattiin aluksi instrumentoinnin keinoin sijoit-tamalla koodiin pysäytyskohtia, joissa ohjelma pysähtyisi. Sitten pystyttiin tutkimaan ajon senhetkisiä muuttujien arvoja. Esimerkiksi toimialalistan toimivuus varmistettiin ensin yrittämällä tallentaa se erääseen muuttujaan, minkä jälkeen sovelluksen ajo pysäytettiin, jolloin muuttujaa päästiin tutkimaan ja nähtäisiin, oliko tietojen hakeminen onnistunut.

Tallenna- ja kumoa muutokset -painikkeet

Sovelluksessa oli mahdollista palata muokkaamaan tarjousta, jos tarjousta ei ollut vielä hyväksytty. Eräässä kehitysvaiheessa tässä jälkimuokkaustilassa tallennus oli mahdol-

lista vain painamalla seuraava-painiketta. Toivottiin täydellistävän ylläpidon toimenpidettä, jossa lisättäisiin uusi tallenna-painike, joka tallentaisi muutokset siirtymättä toiselle sivulle. Lisäksi toivottiin painiketta, jolla voitaisiin kumota muokatessa tehdyt muutokset ja palauttaa tarjouksen tallennetussa versiossa olevat tiedot.

Muutosvaikutusanalyysissä arvioitiin, ettei muutoksella olisi ylimääräisiä vaikutuksia. Uudet painikkeet tulisivat velhon edellinen- ja seuraava-painikkeiden ohelle. Nämä painikkeet eivät olleet velhon yksittäisiä sivuja vastaavissa komponenteissa, vaan kehyskomponentissa, joka oli vastuussa velhon eri sivujen näyttämisestä. Kehyskomponenttiin lisättiin vaaditut painikkeet, jotka näytettäisiin, jos tarjous olisi tallennettu, mutta ei hyväksytty. Tämä tieto löytyisi tarjous-olion tila-jäsenestä. Tallenna-painikkeelle asetettiin funktio, joka tallentaisi tarjouksen, ja kumoa muutokset -painikkeelle funktio, joka hakisi tarjouksen tallennetun version tiedot. Kummatkin funktiot olivat jo olemassa, ja ne oli toteutettu niin, etteivät ne esimerkiksi aiheuttaneet sivuvaikutuksia, joten niitä ei ollut tarpeen muuttaa.

Ohjelman hyvän rakenteen ja laajennettavuuden ansiosta painikkeiden lisääminen oli vaivatonta.

Navigaatiopalkin ulkoasukorjauksia

Navigaatiopalkkiin liittyi ongelma, jossa sen ulkoasu muuttui epäjohdonmukaisesti velhossa navigoitaessa. Määrittelyn mukaan nykyinen ja menneet vaiheet näkyisivät vihreällä ja tulevat vaiheet valkoisena, mutta virheen takia vaiheen väri muuttui punavalkoiseksi, jos käyttäjä etenisi ensin seuraavaan vaiheeseen ja palaisi sitten takaisin edelliseen (kuva 10).



Kuva 10. Navigaatiopalkin odottamaton värienmuutos [Työtapaturmavakuutuksen tarjous- ja siirtojärjestelmä 2019].

Vaiheen olisi pitänyt näkyä vihreänä. Vaiheen punaiseksi värjäävää tyyliasetusta etsittiin niin globaaleista kuin pelkkään navigaatiopalkkiin vaikuttavista CSS-tiedostoista. Lähdekoodista muun muassa etsittiin hakufunktion avulla koodia, joka sisältäisi lauseen `color: red` tai `color: rgb(255, 0, 0)`.

Lopulta paljastui, että värin tuotti navigaatiopalkin luomisessa käytetty Arch wizard -komponentti. Punavalkoinen väri oli yksi komponentin omista oletusulkoasuohjeista, jotka sijaitsivat Angularin komponenttikirjastossa, joita ei ollut sisällytetty hakuun. Palkin oletusulkoasu voitiin korvata kirjoittamalla CSS-ulkoasuohjeita `aw-wizard-navigation-bar-elementille`, mutta ei ollut huomioitu sitä, että komponentti huomioisi myös tilanteet, joissa käyttäjä palasi johonkin vaiheeseen uudestaan. Saatiin selville, että kyseisen tilanteen navigaatiopalkin vaiheen ulkoasua saatettiin muuttaa kohdentamalla ohjeet `steps-indicator-luokan ul-elementeille` ja `done:after-luokan li-elementeille` (esimerkkikoodi 31).

```
aw-wizard-navigation-bar ul.steps-indicator li.editing:after {
  // Step is being revisited
  border-color: $primary-color !important;
  background-color: $primary-color !important;
  color: $secondary-color !important;
}
```

Esimerkkikoodi 31. Navigaatiopalkin ulkoasun korjaus CSS-tyyliohjeessa [Työtapaturmavakuutuksen tarjous- ja siirtojärjestelmän lähdekoodi 2019].

Lisäksi navigaatiopalkki käytti väärää fonttia. Navigaatiopalkin tekstin tyyliasetukset jouduttiin selvittämään kokeilemalla, sillä niitä ei ollut dokumentoitu; tekstin tyyliohjeet tuli kirjoittaa `ul`-, `li`-, `div`- ja `a`-elementeille.

Ongelman paikantaminen oli hankalaa, sillä ulkoasuvirhe ei suoranaisesti johtunut vääristä tyyliohjeista, vaan Angularin navigaatiopalkin vaikeasti jäljitettävistä oletustyyliohjeista, jotka tulivat voimaan tapauksissa, joille ei ollut huomattu asettaa tyyliohjeita. Koska Arch wizardin ulkoasuasetukset oli dokumentoitu huonosti, tarvittiin ehkäisevää ylläpitoa; kaikki navigaatiopalkin tyyliohjeet dokumentoitiin tyyliohjeisiin kommentteilla, jotka kuvasivat eri tyyliohjeiden vaikutuksia, jotta tulevat kehittäjät eivät joutuisi tekemään selvitystyötä uudelleen.

Häiriö tilastovastauksen vastaanottamisessa

Yritystilastopyyntöjen siirtoliikenteessä raportoitiin virheestä, jossa sovellus ei pystynyt käsittelemään erään yrityksen tilastovastausta ja tallentamaan tilastoja tietokantaan. Sovellus sai tilastovastaukset SAS-tietokoneväylältä XML-muodossa. XML on taulukko-muotoisen datan tekstitalennusmuoto. Vastausta tutkittaessa huomattiin, että tilastojen toimittaminen oli estynyt, sillä yrityksellä ei ollut tilastotietoihin liitettäviä sopimuksia (esimerkkikoodi 32).

```

vaksop: 1312847168
<ASNO>
  Asno: 1472543
</ASNO>
<AJOPVM>
  Ajopvm: 18.01.2019
</AJOPVM>
<HAKUPVM>
  Hakupvm: 18.01.2019
</HAKUPVM>
<TILASTOPYYNTOID>
  Tilastopyyntoid: 56162739048044297
</TILASTOPYYNTOID>
<Virhe>
  Virhe: Asiakkaalla ei ole tilastotietoihin liitettäviä
  sopimuksia.
</Virhe>

```

Esimerkkikoodi 32. Sovelluksen vastaanottama XML-sanoma [Työtaturmavakuutuksen tarjous- ja siirtojärjestelmän tilastovastaus 2019].

Kuitenkin sovelluksen tulisi käsitellä tällainen tapaus ilmoittamalla käyttäjälle, etteivät tilastovastauksen kriteerit olleet täyttyneet. Taustajärjestelmän Java-koodista etsittiin kohta, jossa tilastovastaus vastaanotettiin, ja huomattiin, että poikkeuskäsittely tapahtui vain, jos sanomasta löytyi POIKKEUSTIETO-kenttä (esimerkkikoodi 33).

Tilastovastaus-XML-rajapintaluokassa:

```
// POIKKEUSTIETO
@Column(name = "POIKKEUSTIETO")
private String poikkeustieto;

public String getPoikkeustieto() {
    return poikkeustieto;
}
```

Tilastovastauksen käsittelijäluokassa:

```
private static final String label_poikkeustieto = "Poikkeustieto";

private void addVakuutustenTiedot(TilastovastausRaportti Raportti,
    YrityksenVakuutukset yrityksenVakuutukset){

    if (StringUtil.exists(tilastovastausEntity.getPoikkeustieto())){
        raportti.addRivi();
        raportti.addRivi();
        Koodiarvo poikkeustietoKoodiarvo =
            OhjausUtil.haeKoodistoarvoselite(ohjaustiedotPalvelu,
                kieli, KoodistoEnum.POIKKEUSTIETO.name(),
                tilastovastausEntity.getPoikkeustieto());
        raportti.addRivi123(label_poikkeustieto,
            tilastovastausEntity.getPoikkeustieto(),
            poikkeustietoKoodiarvo.getSelite());
        raportti.addRivi12(label_tilastopvm,
            PaivamaaraUtil.getShowDate(
                tilastovastausEntity.getTilastopvm()));
        return;
    }
}
```

Esimerkkikoodi 33. Tilastovastauksen poikkeuskäsittely.

Todettiin, että SAS-palvelun toimittamassa sanomassa olisi pitänyt olla poikkeustietokenttä. SAS-vastaavaa pyydettiin korjaamaan väyläpalvelu. Ongelman paikannus oli vaiivatonta Java-koodin hyvän ja johdonmukaisen rakenteen ja kiitettävän dokumentoinnin ansiosta.

5.4 Palaute

Asiakkaan palaute

Kaiken kaikkiaan asiakas oli insinööriyönä tehtyyn työhön tyytyväinen. Kaikki pyydetty ominaisuudet toteutettiin ja dokumentoitiin. Koodausstandardeja noudatettiin, dokumentteja ylläpidettiin ja asiakkaan asettamat kriteerit kaikille ominaisuuksille täyttyivät.

Asiakas raportoi erääseen pienkehitystyöhön liittyvästä virheestä. Tarjousvelhon yrittäjät-sivun dialogi-ikkunassa (kuva 11) yrittäjää ei voitu lisätä, sillä lisää-painikkeesta napautettaessa ohjelma ilmoitti, ettei omistusosuus-kenttää saisi jättää tyhjäksi.

Henkilötunnus	010180-321A	
	<input type="button" value="Lisää ilman henkilötunnusta"/>	<input type="button" value="Hae"/>
Vakuutettavan nimi (etunimi ensin)	Teijo Teikäläinen	
Omistusosuus	40 %	
YEL-työtulo	23 000 €	<input type="button" value="Hae"/>
Kirjoita ammattinimike tai koodi	Kokki, 51201-007	
Työn osuus	40 %	
Kirjoita ammattinimike tai koodi	Toimistotyöntekijä, 41200-034	
Työn osuus	60 %	
Kirjoita ammattinimike tai koodi	-	
Työn osuus	-	
Vakuutusturva	<input type="text"/>	
<input type="button" value="Peruuta"/>	<input type="button" value="Lisää"/>	

Kuva 11. Yrittäjät-sivun dialogi-ikkuna (Työtapaturmavakuutuksen tarjous- ja siirtojärjestelmän prototyyppi 2018).

Ikkunaan oli aikaisemmin tehty muutos, jossa käyttämätön tietokenttä ("omistusosuus") oli poistettu ikkunasta. Tämä oli tehty vain poistamalla kenttää vastaava osuus HTML-koodista. Ohjelma raportoi puuttuvasta tiedosta, sillä kentän poiston yhteydessä ei ollut huomattu poistaa omistusosuus-kentän tarkistusta. Tarkistus poistettiin, ja yrittäjä voitiin taas lisätä.

Virhe johtui puutteellisesta muutosanalyysistä. Ei ollut otettu huomioon, että kentän poiston myötä poistuisi myös mahdollisuus täyttää lomakkeen kaikki pakolliset tiedot.

Ohjelmoijien mielipide koodin luettavuudesta

Ylläpitoon osallistuneita ohjelmoijia pyydettiin kertomaan mielipiteensä koodin luettavuudesta ja ylläpidettävyydestä. Palaute oli yleisesti ottaen myönteistä. Vastanneet ohjelmoijat halusivat pysyä nimettöminä.

Lähdekoodin luokkien, muuttujien ja metodien nimeämiskäytäntöä pidettiin selkeänä ja hyvänä. Tämä auttoi tiettyjen kohtien etsimisessä koodista ja koodin ymmärtämisessä. Kehityskohtana pidettiin koodin luettavuutta suomea puhumattomien näkökulmasta; jotkin nimet olivat koodissa suomeksi. Vaikka käyttöliittymä oli suunniteltu vain suomea puhuville käyttäjille, olisi lähdekoodissa ollut suotavaa käyttää vain englantia, sillä osa kehittäjistä oli ulkomaalaisia.

Koodin kommentointia pidettiin melko hyvänä. Sitä toivottiin hieman lisää, mutta toisaalta todettiin, että koodi oli niin selkeää, ettei se välttämättä kaivannut kommentointia. Parhaimmillaan koodi onkin niin luettavaa, ettei sitä ole tarpeen selostaa erikseen. Lisäksi koodia joudutaan usein muuttamaan, ja muutoksien yhteydessä koodia kuvaavien kommenttien päivittäminen usein unohdetaan varsinkin, jos kommentteja on ylenpalttisesti.

Luokkien, funktioiden ja komponenttien yleiskäyttöisyyttä pidettiin kiitettävänä. Komponentit ja funktiot oli tehty mahdollisimman yleiskäyttöisiksi. Projektin luonteen vuoksi jouduttiin kuitenkin väistämättä luomaan joitakin funktioita, jotka sopivat vain hyvin erikoistuneisiin tehtäviin. Uusien toiminnallisuuksien luomisesta kerrottiin, että usein helppointa oli aloittaa kopioimalla jokin olemassa oleva komponentti.

Koodia pidettiin luettavana ja ymmärrettävänä. Lukeminen oli joissakin tapauksissa hidasta, jos kommentointi oli vähäistä ja varsinkin, kun kehitystiimiin oli vasta liitytty ja järjestelmästä ei ollut etukäteistietämystä. Kuitenkin lähdekoodi oli kirjoitettu johdonmukaisesti ja tyylikkäästi ja suurimmassa osassa tapauksista koodia alettiin ymmärtää nopeasti.

Tuotettu lähdekoodi oli luettavaa, hyvärakenteista ja yleiskäyttöistä. Työn lopputuloksena voidaan pitää, että koodauksessa noudatettiin onnistuneesti ylläpidettävän ja luettavan koodin käytänteitä.

6 Yhteenveto

Insinööriyössä tehdyssä työtapaturmavakuutuksen tarjous- ja siirtojärjestelmän ylläpito- ja pienkehitystyössä pyrittiin noudattamaan luettavan ja ylläpidettävän koodaamisen periaatteita. Käyttöliittymäkomponenttien koodaamisessa noudatettiin kerrosten löyhää kytkemistä pitämällä käyttöliittymän kerrokset erillään toisistaan ja esimerkiksi injektoidulla CSS-tyyliohjeet class-parametrien kautta yhtenäisen käytännön mukaisesti. Koodi pidettiin tehokkaana toteuttamalla algoritmit yksinkertaisesti ja selkeästi. Uudelleenkäytettävyyttä ja laajennettavuutta edistettiin käyttämällä luokkarakenteita, tekemällä yleiskäyttöisiä aliohjelmia ja välttämällä suoria tietojäsenviittauksia. Koodissa noudatettiin yleisiä kommentointi-, esittely-, muotoilu- ja nimeämiskäytäntöjä.

Ohjelmointikäytännöillä tarkoitetaan toimintamalleja, joilla pyritään tuottamaan mahdollisimman laadukasta koodia. Hyvän ohjelmakoodin piirteitä ovat muun muassa käyttöliittymän kerrosten löyhä kytkeminen, ylläpidettyys ja uudelleenkäytettävyys. Käyttöliittymän kerrosten löyhä kytkeminen tarkoittaa käyttöliittymän kerrosten eli rakenteen (HTML), toiminnallisuuden (komentosarjakoodi) ja ulkoasun (CSS) pitämistä erillään toisistaan. Ylläpidettävyyttä voidaan edistää tekemällä koodista mahdollisimman luettavaa esimerkiksi noudattamalla merkityksellistä nimeämiskäytäntöä ja kommentoimalla koodia riittävästi. Uudelleenkäytettävyyttä edistävät luokkien käyttö, yleiskäyttöiset funktiot tai metodit ja suorien tietojäsenviittausten välttäminen.

Ylläpito on ohjelman elinkaaren viimeinen vaihe, jossa ohjelmaa muutetaan virheiden korjaamiseksi, laadun parantamiseksi ja toiminnan muuttamiseksi muuttuneiden vaatimuksien mukaiseksi. Ylläpitoon kuuluu virheidenkorjausta ja pienkehitystä eli uusien ominaisuuksien toteuttamista. Kun ohjelmasta etsitään virhettä, joudutaan usein tekemään ohjelma-analyysi. Ohjelma-analyysi sisältää staattisen analyysin, joka tarkoittaa koodin lukemista, ja dynaamisen analyysin, joka tarkoittaa ohjelman ajonaikaisen käyttäytymisen tarkastelua. Virheen etsinnässä tarkastellaan määrittely- ja suunnitteludokumentteja, koodia, syötteitä ja tulosteita. Aluksi on hyödyllistä etsiä yleisimpiä virhetyypejä. Pienkehityksessä suunnitteludokumentteja voidaan käyttää apuna, kun etsitään sitä kohtaa koodista, johon uusi ominaisuus toteutetaan. Sekä virheenkorjauksissa että pienkehityksessä ennen muutoksen tekemistä tehdään muutosvaikutusanalyysi, jossa tutkitaan muutoksen vaikutukset muihin järjestelmiin, dokumentteihin, tietokantoihin ja käyttöliittymiin.

Asiakkaan palaute oli myönteistä ja ohjelman ylläpitoon osallistuneiden ohjelmoijien yleinen mielipide oli, että tuotettu koodi oli luettavaa ja ylläpidettävää. Hyvä palaute voidaan lukea hyvien käytänteiden mukaisen ylläpidon ansioksi.

Lähteet

Ajila, Samuel. 1995. Software maintenance: an approach to impact analysis of objects change. *Software — practice and experience*. Vol 25 (10), s. 1155–1181.

Architecture overview. Verkkoaineisto. Google. <<https://angular.io/guide/architecture>>. Luettu 8.4.2019.

Arthur, Lowell. 1988. *Software evolution: the software maintenance challenge*. John Wiley and sons.

Berns, Gerald. 1984. Assessing software maintainability. *Communications of the ACM*. Vol 27 (1), s. 14–23.

Chiaretta, Simone. 2018. *Front-end development with ASP.NET core, Angular, and Bootstrap*. E-kirja. Wrox.

Fliedner, Gene & Vokurka, Robert. 1997. Agility: competitive weapon of the 1990s and beyond? *Production & inventory management*. Vol 38 (3), s. 19–24.

Gergeleit, Martin. 1994. Automatic instrumentation of object-oriented programs. *Gesellschaft für Mathematik und Datenverarbeitung*. No. 826. Sankt Augustin: GMD.

Gopal, Rajeev & Schach, Stephen. 1989. Using automatic program decomposition techniques in software maintenance tools. *Teoksessa Conference on software maintenance*, s. 132–141. IEEE computer society press.

Harsu, Maarit. 2003. *Ohjelmien ylläpito ja uudistaminen*. Talentum.

Hietanen, Päivi. 2003. *C++ ja olio-ohjelmointi*. E-kirja. Docendo.

King, Peter; Naughton, Patrick; DeMoney, Mike; Kanerva, Jonni; Walrath, Kathy & Homme, Scott. 1999. Code conventions for the Java programming language. Verkkoaineisto. Oracle. <<https://www.oracle.com/technetwork/java/codeconventions-150003.pdf>>. Päivitetty 20.4.1999. Luettu 16.2.2019.

Kolodiy, Sergey. Unit tests, how to write testable code and why it matters. Verkkoaineisto. Toptal. <<https://www.toptal.com/qa/how-to-write-testable-code-and-why-it-matters>> Luettu 30.3.2019.

Kosonen, Pekka; Peltomäki, Juha & Silander, Simo. 2005. *Java 2 — ohjelmoinnin peruskirja*. E-kirja. Docendo.

Laurikkala, Jorma. 2011. Javadoc-ohjeet. Verkkoaineisto. Tampereen yliopisto. <<http://www.uta.fi/sis/tkt/oope/harjoitustyo/javadoc.html>>. Luettu 30.3.2019.

Lehman, Meir & Belady, Les. 1985. Program evolution: processes of program change. Academic press professional.

Mathiyalakan, Sathasivam; Ashrafi, Noushin; Zhang, Wen; Waage, Frenck; Kuilboer, Jean-Pierre & Heimann, David. 2005. Defining business agility: an exploratory study. Teoksessa Information resources management conference, s. 848–849. Idea Group Publishing.

Paulasaari, Mika. 2018. Tools for code quality in front-end software development. Master's Thesis. Metropolia University of Applied Sciences. Theseus-tietokanta.

Peltomäki, Juha & Nykänen, Ossi. 2006. Web-selainohjelmointi. E-kirja. Docendo.

Pressman, Roger. 1997. Software engineering: a practitioner's approach. McGraw-Hill.

Sambamurthy, Vallabh; Bharadwaj, Anandhi & Grover, Varun. 2003. Shaping agility through digital options: reconceptualizing the role of information technology in contemporary firms. MIS quarterly. Vol 27 (2), s. 237–263. Management Information Systems Research Center; University of Minnesota.

Smit, Michael; Gergel, Barry; Hoover, James & Stroutila, Eleni. 2011. Maintainability and source code conventions: an analysis of open source projects. University of Alberta, Department of Computing Science. TR11-06. ERA-tietokanta.

Sommerville, Ian. 2001. Software engineering. Addison-Wesley.

Swanson, Burton. 1976. The dimensions of maintenance. Teoksessa The 2nd international conference on software engineering, s. 492–497. IEEE computer society press.

Systä, Tarja. 2000. Static and dynamic reverse engineering techniques for Java software systems. Tohtorin väitöskirja. Tampereen yliopisto. TamPub-tietokanta.

Työtäpaturma- ja ammattitautilaki. 2015. 459/24.4.2015.

Työtäpaturmavakuutuksen tarjous- ja siirtojärjestelmä. 2019. Yrityksen sisäinen dokumentti. CGI.

Työtäpaturmavakuutuksen tarjous- ja siirtojärjestelmän arkkitehtuurikuvaus. 2018. Yrityksen sisäinen dokumentti. CGI.

Työtäpaturmavakuutuksen tarjous- ja siirtojärjestelmän lähdekoodi. 2019. Yrityksen sisäinen dokumentti. CGI.

Työtapaturmavakuutuksen tarjous- ja siirtojärjestelmän projektisuunnitelma. 2017. Yrityksen sisäinen dokumentti. CGI.

Työtapaturmavakuutuksen tarjous- ja siirtojärjestelmän prototyyppi. 2018. Yrityksen sisäinen dokumentti. CGI.

Työtapaturmavakuutuksen tarjous- ja siirtojärjestelmän tilastovastaus. 2019. Yrityksen sisäinen dokumentti. CGI.

Vakuuttamisvelvollisuus. 2018. Verkkoaineisto. Tapaturmavakuutuskeskus.
<<https://www.tvk.fi/tyotapaturma-ja-ammattitautivakuutus/vakuuttaminen/vakuuttamisvelvollisuus/>>. Päivitetty 21.12.2018. Luettu 4.4.2018.

Van Vliet, Hans. 1993. Software engineering: principles and practice. John Wiley and sons.

Zakas, Nicholas. 2012. Maintainable JavaScript: writing readable code. E-kirja. O'Reilly Media.