Front-end-ohjelmoijana teleoperaattoreiden monitorointiohjelmia tuottavassa yrityksessä

Henrik Franciscus Leppä

Tämä teos on lisensoitu Creative Commons Nimeä-JaaSamoin 4.0 Kansainvälinen -lisenssillä.

Opinnäytetyö

Tietojenkäsittelyn koulutusohjelma

2016

Tiivistelmä

16.5.2016

Tekijä
Henrik Leppä

Koulutusohjelma
Tietojenkäsittelyn koulutusohjelma

Opinnäytetyön otsikko	**Sivu- ja liite-sivumäärä**
Front-end-ohjelmoijana teleoperaattoreiden monitorointiohjelmia tuottavassa yrityksessä	54 + 0

Opinnäytetyön otsikko englanniksi
Working as a front-end developer in a company that develops monitoring software for telecommunications operators

Tämä portfoliomainen päiväkirjaopinnäytetyö kuvaa ja analysoi verkkokäyttöliittymien ohjelmistokehittäjän työtä teleoperaattoreiden monitorointiohjelmia tuottavassa yrityksessä kymmenen viikon seurantajaksolla, 7.9–13.11.2015. Kehittäjä kirjoittaa jokaisen seurantapäivän alussa, mitä aikoo tehdä päivänä ja kuvailee työtehtäviä. Jokaisen seurantapäivä lopussa kehittäjä kirjoittaa, mitä töitä tuli tehtyä, ja mitä ongelmia tai huomioita päivään liittyy. Jokaisen viikon lopussa kehittäjä syventyy johonkin aiheeseen ja analysoi sitä.

Yritys on Suomessa perustettu, ja nykyään kansainvälinen. Kehittäjä oli työharjoittelussa yrityksessä ennen opinnäytetyötä. Kehittäjä työskentelee tiimissä, joka käyttää scrum-ketteräkehitysmetodia. Front-end-ohjelmoijana kehittäjä tekee tiivistä yhteistyötä muiden kehittäjien sekä käyttäjäkokemustiimin kanssa.

Opinnäytetyön kirjoittamisen aikana kehittäjä oppi ohjelmoinnista muun muassa JavaScriptin ECMAScript 6 -standardin ominaisuuksista ja React-kirjaston käytöstä. Kehittäjä oppi ja kiinnostui enemmän digitaalisesta saavutettavuudesta ja esteettömyydestä, erityisesti WAI-ARIA-esteettömyysmääritelmästä. Kehittäjä oppi paljon kirjoittamisesta, eteenkin miten parhaiten ottaa lukijat huomioon, esimerkiksi typografian, kirjoitustyylin, lyhenteiden käytön, yksityiskohtien, ja erikielisten termien suhteen. Kehittäjä myös oppi paremmin tuottamaan tekstiä. Oppimisprosessin aikana kehittäjälle korostui datan arkistoimisen tärkeys ja menetelmät.

Asiasanat
JavaScript, ketterät menetelmät, ohjelmistoarkkitehtuuri, ohjelmistokehitys, verkko-ohjelmointi

Abstract

16.5.2016

Author
Henrik Leppä

Degree programme
Business Information Technology

Thesis title	**Number of pages and appendix pages**
Working as a front-end developer in a company that produces monitoring software for telecommunications operators	54 + 0

This portfolio-type diary thesis describes and analyses a web interface developer's work in a company that produces monitoring software for telecommunications operators, over a period of ten weeks, between September 7th and November 13th, 2015. The developer plans at the beginning of each day what he is going to do, and describes the tasks. At the end of each day, the developer reflects upon what he accomplished, and what problems and other topics of interest there were during the day. At the end of each week the developer focuses on a relevant subject and analyses it.

The company was founded in Finland, and is now multinational. The developer had his apprenticeship in the company before, and currently works in a team which uses the Scrum agile software development method. As a front-end programmer, the developer works closely with other developers and the user experience team.

During the time of writing this thesis, the developer acquired skills and knowledge of programming with JavaScript ECMAScript 6 features and on usage of the React library. The developer learned and became more interested in digital accessibility, particularly in the WAI-ARIA specification. The developer enhanced his experience of writing, particularly how best to take the readers into consideration, and make better use of the following copywriting features: typography, writing style, use of abbreviations, use of details, and use of foreign terms. The developer also learned techniques for overcoming writer's block. The developer also recognized the importance and methods of data archiving.

Keywords
agile methods, JavaScript, software architecture, software development, web-programming

Sisällys

1 Johdanto 1
1.1 Työtehtävissä vaadittava osaaminen 1
1.2 Tietoperusta 1
1.3 Käsitteitä 3
2 Lähtötilanteen kuvaus 4
2.1 Oman nykyisen työn analyysi 4
2.2 Sidosryhmät työpaikalla 7
2.3 Vuorovaikutustaidot työpaikalla 9
3 Päiväkirjaraportointi 10
3.1 Seurantaviikko 1 10
3.2 Seurantaviikko 2 14
3.3 Seurantaviikko 3 17
3.4 Seurantaviikko 4 21
3.5 Seurantaviikko 5 26
3.6 Seurantaviikko 6 30
3.7 Seurantaviikko 7 34
3.8 Seurantaviikko 8 38
3.9 Seurantaviikko 9 40
3.10 Seurantaviikko 10 44
4 Pohdinta ja päätelmät 49
Lähteet 53

1 Johdanto

Tämä päiväkirjamuotoinen opinnäytetyö kuvaa päivittäistä työtä teleoperaattoreille tukiohjelmia tekevässä yrityksessä, ja analysoi sitä viikoittain. Opinnäytetyön seuranta kohdistuu aikavälille: 7.9–13.11.2015.

Yritys on kansainvälinen, Suomessa vuonna 1986 perustettu, teleoperaattoreille tukiohjelmia tekevä yritys. Yritys työllistää tällä hetkellä noin 600 henkeä. Yrityksen tuotteilla teleoperaattorit voivat esimerkiksi monitoroida, ohjata, ennakoida, ja rahastaa viestintäliikennettä. Yrityksellä on asiakkaita kaikilta mantereilta. Suoritin ammattikorkeakoulun työharjoittelun tässä yrityksessä keväällä 2015.

1.1 Työtehtävissä vaadittava osaaminen

Minut on palkattu yritykseen pääosin front-end-verkko-ohjelmoijaksi, mutta teen myös tarvittaessa back-end-ohjelmointia, ja muita töitä. Front-end-ohjelmoinnissa on tärkeä osata HTML-, JavaScript-, ja CSS-koodauskieliä. Back-end-ohjelmoinnissa on tärkeä osata Java-koodauskieltä. Yrityksen front-end-sovellukset käyttävät perinteisesti Backbone-viitekehystä. Käytämme Mercurial-versionhallintajärjestelmää.

1.2 Tietoperusta

Työtehtäviin tarvittavaan tietoperustaan kuuluu vahva perusohjelmoinnin osaaminen, jota käsitellään esimerkiksi Harold Abelsonin, ja Gerald ja Julie Sussmanin (1984) kirjassa *Structure and Interpretation of Computer Programs (SICP)*. Tätä kirjaa yleensä pidetään funktionaalisen ohjelmoinnin ”Raamattuna”. Kirja opettaa nimettömien funktioiden, eli lambdojen, käytön, ja opettaa jopa miten voi itse tehdä Lisp-ohjelmointikielen tulkin. Kirja on mielestäni joissain kohdissa liian matemaattinen, mutta muuten hyvä.

Vahva JavaScript-ohjelmoinnin ymmärtäminen on tärkeää. Sitä on käsitelty Douglas Crockfordin (2008) kirjassa *JavaScript: The Good Parts*. Kirjan nimi viittaa siihen, että JavaScript on eteenkin historiallisesti pidetty heikkona ja ongelmallisena ohjelmointikielenä, ja näin ollen Crockford esittelee mihin JavaScriptin osiin kannattaa kiinnittää huomiota. Crockford myös esittelee keksimänsä JavaScript Object Notation-, eli JSON-datasiirtoformaatin, jota käytetään hyvin paljon nykyisessä sähköisessä viestinnässä – käytettiin sitten JavaScriptiä tai ei.

Koska front-end-työssä käyttäjän käyttöliittymä muodostuu HTML-koodista, sen oikea käyttö on tärkeää. Käytössä on usein parasta viitata World Wide Web Consortiumin

(W3C) (2014) viralliseen *HTML5*-standardiin, joka on mielestäni hyvin luettava standardiksi. Standardin lukeminen on erityisen tärkeää, koska HTML-elementtien väärinkäyttö saattaa aiheuttaa toimintahäiriöitä sivustossa. Elementtien oikeaoppinen käyttö myös tekee käyttöliittymästä helppokäyttöisemmän, eteenkin käyttäjille, jotka käyttävät pääosin tai pelkästään näppäimistöä, kuten esimerkiksi näkövammaiset.

Koska kehitystiimimme käyttää scrum-ketteräkehitysmetodia, sen tunteminen on tärkeää. Tässä parhaiten auttaa Schwaberin ja Sutherlandin (2013) virallinen scrumin opaskirja, *The Scrum Guide*. Opaskirjasta löytyy myös suomennos Lekman Consultingin (2014) sivuilta. Opaskirja on helppolukuinen, eikä liian pitkä. Pidemmän scrum-kirjan lukeminen ei varmaankaan hyödyttäisi tavallista ohjelmistokehittäjää paljoakaan, koska scrum on suunniteltu niin, että yksityiskohdat saattavat erota eri työympäristöjen ja tiimien välillä.

Näiden teoksien lisäksi on tärkeää säilyttää linkit eri kirjastojen, viitekehysten, ja ohjelmointikielien dokumentaatiosivustoille, koska moderniin ohjelmointiin liittyy niin paljon tietoa, ettei kaikkea ole mahdollista muistaa ulkoa. Itseasiassa ulkoa muistaminen olisi jopa haitallista, koska yksityiskohdat saattavat muuttua paljonkin lyhyessä ajassa. Kehityksessä tulee myös usein haettua tietoa muilta ohjelmistokehittäjiltä, esimerkiksi *Stack Exchange* (Stack Exchange, Inc., 2016) -sivustoista, ja *Mozilla Developer Network (MDN)* (Mozilla Developer Network, 2016) -sivustosta. Myös yleisillä hakukoneilla etsiminen on jatkuvaa.

1.3 Käsitteitä

Automatisoitu testi, koneistettu testi; ohjelman rakennetta tai toimintaa tarkasteleva ohjelma tai komentosarja, joka on asetettu ajettavaksi joka kerta kun ohjelmaan tehdään muutoksia.
Backbone, JavaScript-viitekehys verkkosovellusten tekemiseen.
Back-end, ohjelmiston osa, joka on kauempana käyttäjästä; verkkosivuista puhuttaessa tarkoittaa palvelinpuolta.
Cascading Style Sheets (CSS), koodauskieli, jota pääosin käytetään verkkosivujen sisällön tyylittelemiseen.
Front-end, ohjelmiston osa, joka on lähinnä käyttäjää; verkkosivuista puhuttaessa tarkoittaa osaa, jonka käyttäjä näkee selaimessaan.
HyperText Markup Language (HTML), koodauskieli, jota pääosin käytetään verkkosivujen sisällön jäsentämiseen.
Java, ohjelmointikieli, jota pääosin käytetään palvelimien ohjelmointiin.
JavaScript, ohjelmointikieli, jota pääosin käytetään verkkosivujen selainpuolen ohjelmointiin.
Koodikatselmointi, kehitystyön vaihe, jossa ohjelmoija pyytää muita ohjelmoijia tarkastamaan kirjoittamansa koodin.
Less, dynaaminen tyylikoodauskieli, joka käännetään CSS-koodiksi.
Mercurial, versionhallintajärjestelmä, käytetään koodin säilyttämiseen.
Polyfill, liitännäinen, jonka avulla voi saada uudet toiminnot toimimaan selaimilla, joilla niitä ei vielä ole, joskus käytetään myös sanoja ”shim”, ja ”shiv”.
Program Increment, SAFe-viitekehyksen ajanjakso, joka koostuu useammasta sprintistä.
Scaled Agile Framework (SAFe), ketterän kehityksen viitekehys, jonka on tarkoitus auttaa ketterän kehityksen käyttöä isommissa organisaatioissa.
Scrum, ketterä ohjelmistokehitysmenetelmä.
Scrummaster, scrumtiimin jäsen, joka varmistaa tiimin noudattavan scrumia oikein.
Sprintti, scrumin työskentelyn ajanjakso; yleensä vähintään viikko, ja enintään kuukausi; jotkut käyttävät suomennosta ”pyrähdys”.
Toteutettavuuden osoitus (englanniksi ”Proof of concept”), vajavainen toteutus, jolla arvioidaan jonkin teknologian tai tekniikan toimivuutta.
Tuoteomistaja, scrumtiimin jäsen, joka on vastuussa tiimin tuotteesta, ja tuotteen kehitysjonon hallinnasta; ei ole osa *kehitys*tiimiä.
Yksikkötesti, automatisoitu testi, joka testaa vain yhden luokan, moduulin, tai muun ohjelmakomponentin toimintaa.

2 Lähtötilanteen kuvaus

2.1 Oman nykyisen työn analyysi

Työtehtävät:

- Ohjelmointi
- Automatisoitujen testien kirjoittaminen
  - Yksikkötestien kirjoittaminen
  - Käyttöliittymätestien kirjoittaminen
- Koodin dokumentointi
- Koodikatselmointi
- Tutkimus / toteutettavuuden osoituksien tekeminen
- ”Ad hoc”-testaus
- Päiväpalaveriin osallistuminen
- Sprintin demon suunnittelu
- Sprintin demoon osallistuminen
- Tuotteen kehitysjonon työstöön osallistuminen

Päätyötehtäväni on ohjelmointi, jota teen yrityksen kannettavalla. Suurimman osan ajasta pidän kannettavan telakassa, jolla se on liitetty virtalähteeseen, verkkopiuhaan, ja isompaan näyttöön. Kun olen työpisteelläni, käytän isompaa näyttöä päänäyttönä ja kannettavan omaa näyttöä toissijaisena näyttönä. Kannettavan kanssa käytän langatonta näppäimistöä ja langatonta hiirtä. Kun minun tarvitsee mennä muualle kannettavan kanssa, nostan sen pois telakasta ja otan hiiren mukaan. Työpöytäni on kiinni muiden tiimini jäsenten työpöydissä. Työpöytien välissä ei ole seiniä tai muita esteitä, jotta tiimi voi keskustella työhön liittyvistä asioista mahdollisimman helposti.

Kannettavallani on Windows 7. Vaikka osa yrityksen koodista voi ajaa vain Linuxilla, en ole vielä nähnyt tarpeelliseksi asentaa sitä kannettavalleni, koska sitä tarvitseva koodi on lähinnä palvelimiin liittyvää koodia, ja olen tähän asti lähinnä keskittynyt selainpuoleen. Ohjelmoin pääosin WebStorm-ohjelmistokehitysympäristöllä, käyttäen välillä Eclipse-ympäristöä, kun minun pitää tehdä muutoksia Java-koodiin. Hoidan Mercurialin käytön TortoiseHg-ohjelmalla. Kun teen muutoksia koodiin, testaan toimintaa Mozilla Firefox-, Internet Explorer 11-, ja Google Chrome -selaimilla.

Itse ohjelmoinnin yhteydessä minun usein pitää lisätä automatisoituja testejä kun lisään sovellukseen uusia toimintoja, ja muokata olemassa olevia testejä kun muokkaan toimintoja. Tuotteessamme käytämme JavaScript-yksikkötesteissä Jasmine-kirjastoa, ja selainkäytettävyystesteissä Selenium-viitekehystä.

Koska koodiani tulee käyttämään vähintäänkin kymmenet muut ohjelmoijat, joilla ei ole minuun suoraa yhteyttä, koodin ymmärrettävyys ja dokumentointi on hyvin tärkeää. Dokumentointiin kuuluu: testit, koodiesimerkit, wiki-dokumentaatio, dokumentaatiokommentit (kuten Javadoc ja JSDoc), koodin muut kommentit, ja koodin itsensä luettavuus. Lisäksi, kun saan tehtävän valmiiksi, kirjoitan siitä ratkaisukuvauksen, joka saatetaan liittää seuraavan julkaisun muutoslokiin.

Kun olen tehnyt merkittäviä muutoksia koodin, ja muutoksiin liittyvä toiminto tai korjaus on valmis, teen muutoksistani koodikatselmoinnin Crucible-järjestelmään, ja pyydän yleensä kahta tai useampaa kehitystiimin jäsentä katselmoimaan koodini. Katselmoin itse muiden kehittäjien koodia kun he pyytävät. Koodikatselmoinnissa muut kehittäjät kommentoivat koodia, ja tekevät parannusehdotuksia.

Kerran päivässä, melkein joka päivä, osallistun päiväpalaveriin. päiväpalaveri on noin 5–15 minuutin pituinen, scrummasterin vetämä tapaaminen, jossa kehitystiimin jäsenet kertovat vuorotellen mitä he ovat tehneet viime päiväpalaverin jälkeen, mitä he aikovat tehdä nyt, ja mitä ongelmia heillä on. Päiväpalaverin aikana kaikki kehittäjä seisovat. Koska yksi tiimini jäsenistä työskentelee etänä suurimman osan ajasta, hän yleensä osallistuu päiväpalaveriin Skype-puhelun välityksellä. Päiväpalaveri on tärkeä tiimin jäsenten tiedossa pitämiseen.

Kehitystiimi työskentelee scrumin mukaisesti sprinteissä. Sprintti on muutaman viikon ajanjakso, jonka aikana on tarkoitus saada tietty määrä työtä tehtyä. Lopussa pidetään demo ja sen jälkeen retrospektiivi. Sprintit ovat nykyään kehitystiimissäni kaksi viikkoa. Ne saattoivat aikaisemmin olla 2–3 viikkoa, mutta ne päätettiin muuttaa tasan kahden viikon pituisiksi, jotta kaikkien tiimien sprintit alkavat ja loppuvat samaan aikaan. Yleensä scrumissa sprintin alussa pidetään sprintin suunnittelu, ja sprintin lopussa pidetään retrospektiivi, mutta kehitystiimini päätti pari viikkoa sitten olla pitämättä niitä, koska sprintit suunnitellaan jo ”Program Increment”-jakson suunnittelussa, ja retrospektiivit eivät ole olleet meille hyödyllisiä.

Sprintin lopussa pidetään sprintin demon suunnittelu, jossa tiimi kerääntyy päättämään, mitkä sprintin aikana tehdyt asiat ovat tarpeeksi merkittäviä, että ne kannattaa esittää demossa. Demon suunnittelu pidetään sprintin toiseksi viimeisenä työpäivänä, ja demo itse pidetään sprintin viimeisenä päivänä. Suunnittelun jälkeen demoa varten tehdään diaesitys, joka tehdään HTML-muodossa, jotta se voidaan pitää versionhallinnassa, jotta sitä voi muokata useampi henkilö samaan aikaan. Jäsenet täyttävät omat diansa esitykseen ennen demoa. Demossa tiimin jäsenet esittelevät tekemänsä toiminnot muille tiimin jäsenille, muille tiimeillä, ja muille henkilöille, joita ne kiinnostavat. Demo käydään kokoushuoneessa, jossa on kuvanheitin. Diaesitys toimii demon päärunkona, mutta jäsenet usein esittävät toimintojen toimintaa konsolilla tai selaimilla. Demon aikana pidetään auki myös Skype-tapaaminen, jotta demoon voi osallistua fyysisesti paikallaolemattomatkin, koska kaikki Helsingissä olevat osallistujat eivät mahdu huoneeseen, ja koska jotkin osallistujat ovat toisissa toimistoissa. Tiimin ulkopuoliset osallistuja ovat yleensä ”toimitus ja tuki”-osastola, ”myynti ja markkinointi”-osastolta, tai muista tiimeistä. Esittäjät jakavat ruutunsa Skype-tapaamisen jäsenten kanssa, jotta hekin näkevät. Demomme pääosin vastaavat yleisen scrumin ”sprintin katselmointia”, mutta demossa ei yleensä käydä läpi mitä tehdään ja tapahtuu seuraavaksi, jottei tilaisuus olisi liian pitkä.

Sprintin aikana tuoteomistaja tai scrummaster saattaa järjestää tuotteen kehitysjonon työstämistilaisuuden (josta käytämme vanhahtavaa englanninkielistä termiä ”grooming”). Tuotteen kehitysjonon työstössä kehitystiimi käy läpi tuotteen tulevia toimintoja tai korjauksia. Tiimi ensiksi keskustelee asiasta ja yrittävät saada mahdollisimman selkeän kuvan siitä. Sitten asialle tehdään ”valmiin” määritelmä, joka on yleensä ranskalaisilla viivoilla tehty lyhyt lista, jossa on asian valmiiksi saamiseksi tarvittavat tehtävät. Sitten kehittäjät äänestävät suunnittelupokerin avulla, kuinka paljon työtä he olettavat asiassa olevan. Suunnittelupokerissa jokaisella kehittäjällä on kortit, joissa on Fibonaccin lukujonosta muunnellut arvot ja kysymysmerkki: (?, 0, 1, 2, 3, 5, 8, 13, 20, 40, 100). Yksi piste vastaa suurin piirtein yhtä työpäivää. 8 pistettä vastaa suurin piirtein yhtä sprinttiä, koska sprintissä menee noin yksi päivä demoon, ja yksi päivä muihin asioihin, kuten palavereihin tai koulutuksiin. Pokerissa jokainen kehittäjä valitsee korttinsa, ja sitten kortit paljastetaan samaan aikaan. Asian arvoksi yleensä laitetaan kehittäjien arvioiden keskiarvo, pyöristettynä ylöspäin. Jos kehittäjien arviot eroavat toisistaan merkittävästi, henkilöiltä joilla on pienimmät ja suurimmat arvot kysytään syitä heidän arvioilleen, ja tämän keskustelun jälkeen lopullinen arvio päätetään.

Arvioisin osaamistasoltani olevani taitava suoriutuja. Päätyöalueessani, front-end-ohjelmoinnissa, pystyn tekemään työtehtäväni ilman enempää ohjeistusta. Suurin osa front-

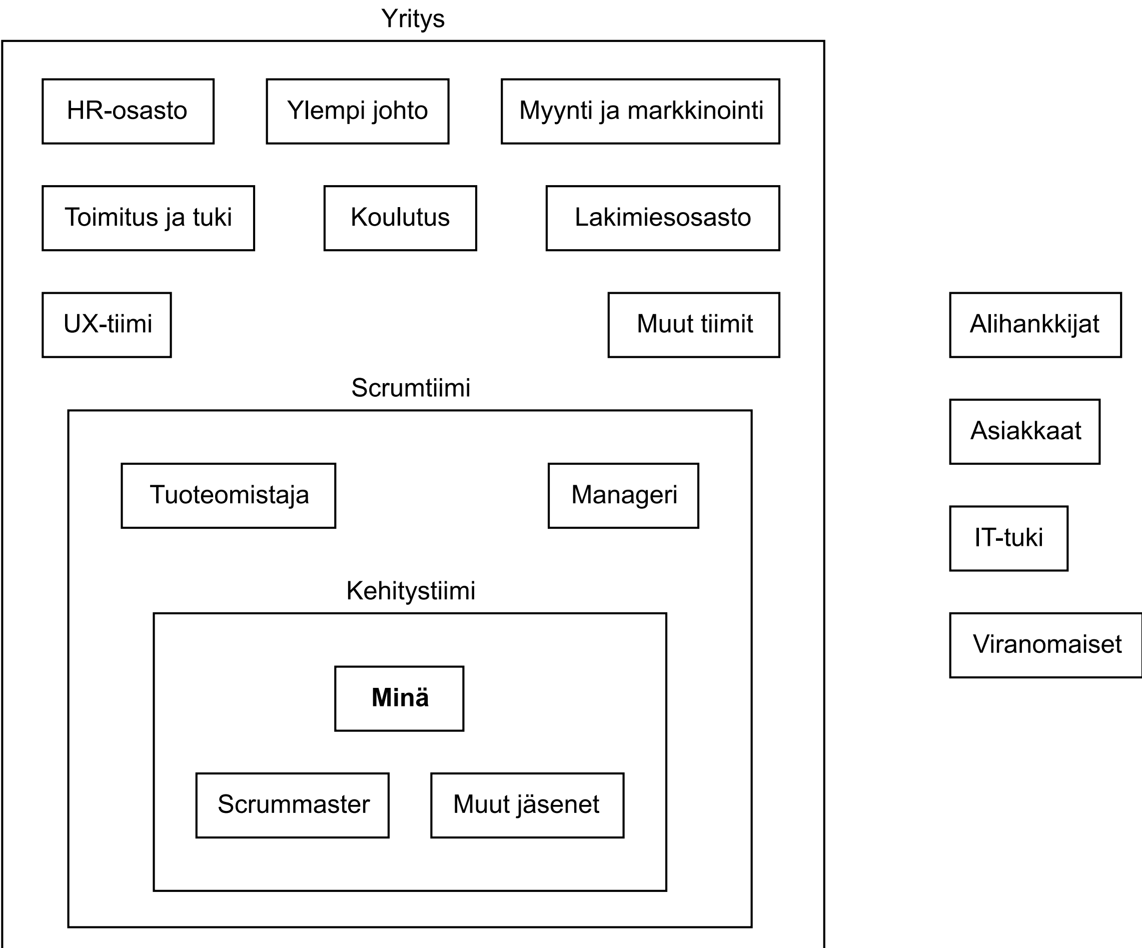
end-kysymyksistäni liittyy yrityksen koodikantaan, eikä käytettyihin ohjelmointikieliin tai kirjastoihin. Itseasiassa, annan yleensä enemmän ohjeistusta muille esimerkiksi JavaScriptistä, kuin itse saan sitä. Lisäksi minua pidetään eräänlaisena tiimin päivämäärien ja lokalisaation käsittelyn erityisosaajana.

Back-end-puolella olen vielä aloitteleva toimija, koska tiedän back-endista hyvin vain Java- ja SQL-kielet, mutta en tiedä paljoakaan muista back-end-asioista, kuten: Liquidbase-tietokantaversionhallintajärjestelmästä, palvelinkomentosarjoista, tai REST-rajapintojen suunnittelusta.

Vaikka olen ollut töissä vain vuoden, ammatillinen kehittymiseni on mielestäni jo hyvällä tasolla. Front-end-ohjelmoinnissa olen mielestäni samalla tasolla kuin muut tiimin jäsenet. Uskon että tämä johtuu pääosin siitä, että olen pari viime vuoden ajan käyttänyt merkittävän osan vapaa-ajastani front-end-ohjelmoimiseen ja siitä lukemiseen. En ole back-end -puolella yhtä kehittynyt kuin muut tiimin jäsenet, mutta mielestäni tämä johtuu pääosin siitä, ettei minua back-end kovin paljoa kiinnosta. Olen mielestäni jo nyt hyvin sisäistänyt kehitysprosessin muut osa-alueet, kuten kommunikaation, dokumentoinnin, ja testauksen. Vaikka se ei liitykään työhöni, haluaisin tällä hetkellä eniten oppia syvemmin C-kieltä, koska suuri osa ohjelmoinnista perustuu siihen, ja sitä käytetään paljon. Minulla on myös pitkä lista eri ohjelmoinnin klassikkokirjoja, jotka haluan lukea kun minulla on aikaa.

2.2 Sidosryhmät työpaikalla

Yrityksessä työskentelen kehitystiimissä (Kuvio 1), jonka vastuulla on yrityksen tuotteiden yhteisen alustan ja siihen liittyvien komponenttien ohjelmoiminen. Kuten on tavallista scrumissa, yksi kehitystiimin jäsenistä on scrummaster, ja kehitystiimin on itsessään osa scrumtiimiä, johon kuuluu myös tuoteomistaja ja manageri.



Kuvio 1. Yrityksen sidosryhmät

Managerini on päävastuullinen tiimin toiminnasta, tuloksesta, ja suunnitelluista tuotejulkaisuista aikatauluineen. Manageri delegoi suuren osan näistä tehtävistä muille, kuten tuoteomistajalle, mutta hän on viimekädessä vastuussa. Manageri myös vastaa työsopimuksista, työterveydestä, lomista, koulutuksista, ynnä muista tiimin jäseniin liittyvistä asioista.

Tuoteomistaja vastaa alustan yhteisen näkemyksen luomisesta ja ylläpitämisessä, koordinoimalla scrumtiimien, UX-tiimin, asiakkaiden, ”myynti ja markkinointi”-osaston, lakimiesosaston, ja ylemmän johdon kanssa.

”User Experience”-, eli käyttäjäkokemus-, eli UX-tiimi, vastaa yrityksen tuotteiden käyttökokemuksen ja ulkoasun suunnittelusta.

Kehitystiimille annettuja työtehtäviä hallinnoidaan yrityksen JIRA-työseurantajärjestelmällä, ja niitä voi kirjata sinne tuoteomistaja, UX-tiimi, muut kehitystiimit, tai kehitystiimin jäsenet itse. Työtehtäviä ei yleensä tule asiakkailta suoraan, kuin kiireellisissä tilanteissa, vaan ne yleensä tulevat tuoteomistajien, ”toimitus ja tuki”-osaston, tai UX-tiimin kautta.

2.3 Vuorovaikutustaidot työpaikalla

Vuorovaikutustaidot ovat työssä tärkeät. Koska työ tehdään kehitystiimissä, jonka jäsenet yhtä lukuun ottamatta istuvat lähekkäin, keskustelutilanteita on joka päivä. Monissa tehtävissä kehittäjät tarvitsevat toistensa apua, koska koodikanta on sen verran suuri ja moninainen, ettei sitä kukaan tunne kokonaan.

Kehitystiimin päiväpalavereissa vuorovaikutustaidot ovat hyvin tärkeät, koska pitää osata jäsentään lyhyesti mitä on tehnyt ja mitä aikoo tehdä. Lisäksi pitää osata kunnioittaa muiden jäsenten puheenvuoroja, ja tunnistaa mitkä asiat vaativat pidempää keskustelua päiväpalaverin jälkeen.

Vuorovaikutustaidot ovat myös tärkeät, kun keskustelee UX-tiimin kanssa, koska vaikka heillä on jotain ohjelmointikokemusta, he eivät ole kovin teknisesti tietoisia. Monissa tilanteissa UX saattaa pyytää jotakin, joka on joko teknisesti liian hankala toteuttaa tai jopa mahdoton. Näissä tilanteissa pitää pystyä keksimään tapa, jolla voi selittää ongelman, ja löytää jokin muu vaihtoehto tai kompromissi. On myös tilanteita, joissa UX-tiimin pyynnöt eivät ole selkeitä, tai he eivät ole määritelleet jotakin vielä ollenkaan. Joissain tilanteissa UX-tiimin jäsenten kanssa tulee käytyä jopa tunninkin mittaisia suullisia keskusteluja.

Suullisten vuorovaikutustaitojen lisäksi tekstilliset vuorovaikutus taidot ovat myös tärkeitä. Kehityksessä tulee keskustelua paljon sähköpostin, Skype-verkkojuttelun, ja kehitysjärjestelmän ja koodikatselmointijärjestelmän kommenttien avulla.

Edellä mainittujen lyhytaikaisten viestien lisäksi pitää myös osata kommunikoida hyvin dokumentoinnin, koodin dokumentointikommenttien, ja koodin itsensä avulla. Näissä kommunikaation muodoissa on erityisen tärkeä olla selkeä, koska niitä tullaan lukemaan silloinkin, kun alkuperäinen ohjelmoija ei enää työskentele yrityksessä, ja tulevaisuuden ohjelmoijat joutuvat selvittämään asioita. Näillä muodoilla hyvin kommunikoiminen on tärkeää silloinkin, kun on ainoa joka työskentelee jonkin komponentin tai ominaisuuden kanssa, koska monet asiat saattavat unohtua hyvinkin nopeasti.

3 Päiväkirjaraportointi

3.1 Seurantaviikko 1

Maanantai 7.9.2015

Läpäisin viime viikolla asiakasprosessi-e-opetuskurssin ensimmäisen osan. Tavoitteeni tänään on läpäistä ainakin sen toinen osa. Kurssin tarkoitus on opettaa kaikille yrityksen työntekijöille pääpiirteittäin miten eri asiakkaisiin liittyvät prosessit etenevät, mitkä roolit niihin osallistuvat, ja mitä dokumentteja niissä luodaan. Kurssi koostuu diasarjasta, jossa on prosessikaavioita, ja niiden selkokielisiä kuvauksia; ja testistä, joka koostuu monivalintakysymyksistä. Testin pystyy ottamaan monta kertaa, ja siinä ei ole aikarajoitusta. Aion myös osallistua päiväpalaveriin.

Puhuin scrummasterin ja toisen kehittäjän kanssa erään näkymän aikojen olemisesta selaimen ajassa, palvelinajan sijaan. Olin aikaisemmin muuttanut yhden UNIX-aikoja käyttävän näkymän käyttämään ISO 8601 -aikoja.

On monia tilanteita, joissa palvelimen pitää lähettää aika selaimelle. Yleensä ohjelmoinnissa aikoja käsitellään olioina, mutta aikoja ei yleensä kuljeteta oliomuodossa. Tällä hetkellä tuotteissamme ajat yleensä kuljetetaan UNIX-millisekuntiajassa. UNIX-muodossa oleva aika on kokonaisluku, joka ilmaisee ajan sekunteina vuoden 1970 alusta Greenwichin ajassa (ilman karkaussekunteja). Millisekunteja käytetään usein sekuntien sijasta, ja joskus käytetään myös muita aikayksiköitä, kuten nanosekunteja, mutta tämä on harvinaisempaa. Esimerkki UNIX-millisekuntiajasta: `1441625222091`.

UNIX-aikojen käyttö aiheuttaa kuitenkin ongelman tuotteessamme. Koska tuotteemme monitoroivat palvelimien toimintaa, ja palvelimien vikoja selvittävät työntekijät usein työskentelevät eri aikavyöhykkeillä, on tärkeää saada aika palvelimen ajassa, jotta aika näkyy kaikille samana. Tämä ei ole kuitenkaan mahdollista UNIX-ajalla, koska siinä ei ole mukana tietoa palvelimen aikavyöhykkeestä. Lisäksi palvelimen aikavyöhykettä ei kannata lähettää erikseen, koska samasta paikasta lähetetyt ajat voivat erota keskusajasta eri määrillä kesäaikajärjestelmän takia.

Jotta aikavyöhyke saadaan lähetettyä ajan mukana, tulee käyttää jotain toista aikaformaattia. Tässä toimii hyvin ISO 8601 -formaatti, joka on laajalti tunnettu ja käytetty kansainvälinen standardi. Esimerkki ISO 8601 -ajasta, joka vastaa aiempaa UNIX-millisekun-

tiaikaa: 2015-09-07 14:27:02.091+03:00. Tämä esimerkkiaika alkaa vuodella, seuraten sitten kuukaudella, päivällä, tunneilla, minuuteilla, sekunneilla, ja millisekunneilla. Viimeinen osa, +03:00, merkitsee ajan olevan kolme tuntia edellä keskusaikaa; tämä on Suomen kesäaika. Suomen talviaika on +02:00. ISO 8601 -formaatissa on mahdollista merkitä aika hiukan eri tavoilla: esimerkiksi päivän ja kellonajan välissä voi olla iso T välin sijaan, ja lukujen väleissä ei tarvitse olla viivoja, pisteitä, ja kaksoispisteitä. Valitsin juuri tämän ISO 8601 -formaatin muodon, koska se oli mielestäni ihmisille kaikista luettavin.

Pääsin e-opetuskurssin toisesta osasta läpi. Osallistuin päiväpalaveriin. Se meni hiukan pitkäksi, koska muut tiimin jäsenet takertuivat tehtäviensä yksityiskohtiin, ja yksi jäsen vastasi Skype-kutsuun hiukan myöhässä.

Tiistai 8.9.2015

Tavoitteeni tänään on osallistua päiväpalaveriin, ja läpäistä asiakasprosessi-e-opetuskurssin kolmas osa. Kolmas osa käsittelee asiakastukiprosessia. Haluan myös mahdollisesti tehdä hiukan ohjelmointia.

Osallistuin päiväpalaveriin. Läpäisin asiakasprosessikoulutuksen. Olin pari viikkoa aikaisemmin työstänyt toteutettavuuden osoitusta WebSocket-menetelmään liittyen. Koulutuksen läpäisyn jälkeen yritin palata työstämään sitä, mutta en päässyt pitkälle, koska se ei toiminut koneellani, ja sen alkuperäinen luoja oli alustan tulevan julkaisun takia liian kiireinen auttamaan minua.

Keskiviikko 9.9.2015

Tavoitteeni tänään on osallistua päiväpalaveriin, ja lisätä käsittely HTML-luokalle, jonka voisi lisätä linkkeihin, jotta niillä voi siirtyä toiseen näkymään vaikka nykyisessä näkymässä olisi tallentamattomia muutoksia.

Yleensä kun yrityksen verkkosivualustan näkymissä on tallentamattomia muutoksia, ja käyttäjä yrittää siirtyä muualle, selain kysyy käyttäjältä haluaako hän jättää muutokset tekemättä ja siirtyä eteenpäin, vai jäädä sivulle. Joissain näkymissä pitää kuitenkin pystyä siirtymään esimerkiksi alinäkymään ilman että selain pysäyttää sen.

Huomasin koko osastolle lähetetyn sähköpostin, jossa ihmeteltiin sitä, että aina näkymiä pois siivotessa, niiden dataolioista poistetaan validointi, mikä on ongelmallista, jos use-

ampi näkymä käyttää yhtä dataoliota samanaikaisesti. Raportoin asian muutostenseurantajärjestelmään. Päiväpalaveria ei pidetty, koska melkein kaikki tiimin jäsenet olivat koulutuksessa.

Huomasin taas, että monissa tilanteissa saattaa olettaa jonkin asian olevan universaalimpi kuin se on. Esimerkiksi saattaa olettaa, että dataolion validoinnin voi poistaa kun sen näkymä siivotaan pois, mutta näin ei voikaan tehdä, koska dataoliota saattaa käyttää jokin toinenkin näkymä.

Torstai 10.9.2015

Kun saavuin työpaikalle, tiimin jäsen, joka on eniten vastuussa alustan julkaisusta, pyysi minua ”ad hoc”-testaamaan julkaisun Internet Explorer 11 ja Google Chrome -selaimilla.

Päiväpalaveria ei taaskaan pidetty koska liian moni tiimin jäsen oli koulutuksessa.

Managerini huomautti tunneista, joka olin erehdyksessä merkannut sairaspoissaoloiksi, koska olen työharjoittelusopimuksella, jossa palkka maksetaan vain tehtyjen tuntien mukaan, ja tuntien määrällä ei ole alarajaa. Korjasimme tunnit oikeiksi tuntiraportointijärjestelmään.

Sain testaukset tehtyä ja löysin muutaman vian. Puhuin kolmen muun kehitystiimin jäsenen kanssa heidän löytämästään viasta, joka aiheutti sen että käyttäjää ei merkattu muokatuksi, jos hänen salasanaa muutettiin, tai hänelle annettiin tai häneltä poistettiin rooleja. Ongelma johtui siitä, että käyttäjän salasana ja roolit eivät olleet samassa oliossa kuin käyttäjän muut tiedot, ja sen takia käyttäjäolio ei nähnyt näitä muutoksia.

Perjantai 11.9.2015

Kun saavuin työpaikalle, tiimin jäsen, joka on eniten vastuussa alustan julkaisusta, pyysi minua vielä tekemään viimeistä testailua.

Testauksessa huomasin, että eräät ikonit, joita piti pystyä kääntämään sivuttain tai ylösalaisin, eivät olleet kääntyneenä. Julkaisusta vastuussa oleva kehittäjä pyysi minua pikaisesti korjaamaan tämän vian.

Vikaa selvittäessä, heti ensimmäiseksi huomasin, että yksi HTML-elementti ei ollut suljettu oikein, joten se jatkui sivun loppuun asti. Aluksi luulin, että vika johtui pelkästään tästä,

mutta kun korjasin elementin sulkemisen, huomasin että vika ei korjaantunut. Minä olin noin tunnin ymmälläni, mistä vika johtui, mutta sitten muistin, että vika oli ilmestynyt vain äskettäin, joten päätin katsoa versionhallinnasta kyseisen koodin historiaa. Historiaa katsoessa huomasin, että toinen kehittäjä oli ylikirjoittanut oikean koodin päivittäessään tiedoston muuta osaa. Keskustellessani julkaisuvastaavan kanssa minulle selvisi, että tiedosto, josta vika löytyi, olikin tiedosto, joka generoidaan automaattisesti komennolla. Koska tämä toinen kehittäjä oli jo lähtenyt viikonlopunviettoon, päätin vain päivittää tiedoston, ja jätin itselleni muistutuksen pyytää tätä toista kehittäjää päivittämään tiedoston generointikomennon.

Päiväpalaveria ei taaskaan pidetty koska liian moni tiimin jäsen oli koulutuksessa.

Viikkoanalyysi

Viikolla käymässäni e-opetuskurssin toisessa osassa opin yrityksen prosesseista, että:

- asiakkaille toimitetuissa projekteissa pitää olla testistrategia mukana tarjouksessa;
- projektimanagerit ja resurssimanagerit suunnittelevat resurssien kysynnän ja tarjonnan keskitetyllä tietojärjestelmällä;
- muutospyynnönhallintaan kuuluu:
  - muutospyynnön teko ja ylös kirjaaminen,
  - sen vaikutuksen arviointi,
  - sen hyväksyminen, ja
  - sen toimeenpano;
- lopullinen vastuu projektista kuuluu sen ohjausjohtokunnalle; ja
- järjestelmäintegraatiotestit ja käyttäjähyväksyntätestaus jätetään aina asiakkaan vastuulle, vaikka yritys tekisi kaiken niihin liittyvän työn.

E-opetuskurssin kolmannessa osassa opin ”incident”- ja ”problem”-sanojen eron asiakastuessa. ”Incident”, eli tapaus, on tilanne jossa palvelu jostain syystä ei toimi joillain tavalla, tai sen laatu on jollain tavalla tarkoitusta huonompi. ”Problem” tarkoittaa tilannetta, jossa monta tapausta tulee samaan aikaan, tai löydetään sellainen, joka saattaa tulevaisuudesta aiheuttaa ongelmia. Opin myös paremmin termien: ”service request”, eli palvelupyynnön; ja ”change request”, eli muutospyynnön käytön. Lisäksi opin miten yritys luokittelee tapahtumien tärkeysjärjestykset, ja mitä eri ”service level agreement”, eli palvelutasosopimuksia, yritys tarjoaa asiakkaille.

Viikolla tekemissäni testailuissa löysin kolme vikaa jotka kaikki johtuivat siitä, ettemme olleet käyttäneet selaimen omaa tapaa tehdä jokin asia, vaan olimme tehneet oman viritelmän, joka ei ollut ottanut huomioon monia asioita jotka on otettu huomioon selaimen omassa menetelmässä.

Ensimmäinen vika oli, että auki olevat alavetovalikkomme eivät pysyneet oikeilla paikoillaan, kun selaimen ikkunan kokoa muutti. Tämä johtui siitä, että auki olevien alavetovalikkomme sijoitetaan ruudulle absoluuttisesti, eikä suhteessa niiden alkuperäiseen paikkaan.

Toisessa viassa, jos yrityksen ristikkokomponentissa liikkui tabulaattorilla, selaimen huomio siirtyi oudolla tavalla: selain ensiksi keskittyi sivun ensimmäisen ristikon otsikkoihin, sitten sivun toisen ristikon otsikkoihin, sitten ensimmäisen ristikon ensimmäiseen soluun, sitten toisen ristikon ensimmäiseen soluun, ja niin edelleen.

Kolmas vika oli, että yrityksen valintaruutu-, alasvetovalikko-, ja valintanappikomponentit eivät olleet kohdistettavissa ollenkaan näppäimistöllä.

Kaksi viimeistä vikaa liittyvät tabulaattorinavigointiin (englanniksi ”tabbing”). Tabulaattorinavigoinnilla kokeneempi käyttäjä pystyy käyttämään ohjelmia nopeammin, kuin jos hän käyttäisi kumpaakin, näppäimistöä ja hiirtä. Lisäksi tabulaattorinavigointi on ainoa tapa käyttää ohjelmia henkilölle, joka ei voi käyttää hiirtä liikunta-, aisti-, tai psyykkisten rajoitteiden vuoksi.

3.2 Seurantaviikko 2

Maanantai 14.9.2015

Kun saavuin työpaikalle, yksi kehitystiimini jäsen pyysi minua kiireellisesti testaamaan ja koodikatselmoimaan hänen tekemä korjaus, joka oli tulossa seuraavaan julkaisuun. Korjaus liittyi alustamme monitorointityökaluun, jossa näkyy käyttäjien tekemät toimenpiteet. Työkalussa toimenpiteistä tehdyt lokit voi rajata ajan mukaan. Työkalussa oli kuitenkin ongelma: jos käyttäjä valitsi aikavälikenttiin sellaiset ajat, että rajauksen alkuaika ja loppuaika olisivat väärinpäin, eli jos hän esimerkiksi laittaisi aikaväliksi 7.10.2014–5.10.2014, työkalun hakukentät lukkiutuisivat, eikä käyttäjä pystyisi muuttamaan rajausta lataamatta näkymää uudelleen.

Toinen kehitystiimini jäsen antoi minulle erään tehtävän, mutta se ei ollut kiireellinen, vaan se annettiin minulle, koska satuin olemaan asiasta tietoisin.

Tavoitteeni tälle päivälle on saada testattua ja koodikatselmoitua työtoverin muutos, ja osallistua päiväpalaveriin.

Osallistuin päiväpalaveriin. Päiväpalaverin paikkaa oli siirretty hiukan, koska scrummaster oli lomalla, joten etätöitä tekevään kehitystiimin jäseneen otettiin yhteyttä toiselta koneelta. Lisäksi tämä jäsen vastasi Skype-kutsuun hiukan myöhässä.

Testasin työtoverini korjauksen, ja se vaikutti toimivan. Katselmoin myös korjauksen koodin, ja ehdotin pari muutosta. Huomautin hänelle koodin kohdasta, jossa hän kirjoitti ohjelman nykyisen tilan komentorivin lokiin, ja joka oli selkeästi jäänyt poistamatta, kun hän oli lisännyt koodin versionhallintaan. Lisäksi hän oli unohtanut poistaa koodinpätkän, jonka hän oli piilottanut kommentoimalla sen pois.

Tiistai 15.9.2015

Olin tämän päivän kotona, koska minulla oli paha olo.

Keskiviikko 16.9.2015

Olin tämänkin päivän kotona.

Torstai 17.9.2015

Tavoitteeni tänään on saada vihdoinkin valmiiksi työ, jonka aloitin edeltävän viikon keskiviikkona: HTML-luokka, jonka avulla voi tehdä linkkejä, jotka eivät varoita tallentamattomista muutoksista niillä navigoidessa. Lisäksi yritän selvittää ongelman, joka estää minua tekemästä Excel-tuntiraportteja, jotka pitäisi lähettää palkanmaksuun. Aion myös osallistua päiväpalaveriin ja sprintin demon suunnitteluun.

Kun tulin töihin, yksi tiimin jäsenistä kysyi, että tuleeko eräs aiemmin tekemäni ominaisuus mukaan seuraavaan julkaisuun. Kerroin hänelle että se ei tule, koska osa sen koodista muutti erään komponentin oletusasetuksia. Tämä oletusasetuksen muutos todennäköisesti aiheuttaa monia muutoksia muiden tiimien projekteihin, joten olin päättänyt jättää se seuraavaan julkaisuun, jotta muilla tiimeillä on tarpeeksi aikaa tehdä tarvittavat muutokset.

Aloin selvittämään työtuntiraportointijärjestelmän ongelmaa, joka esti minua luomasta tuntiraportteja, jotka minun tulee lähettää managerilleni ja palkkalistalle. Näytin ongelman

managerilleni, ja hän ohjeisti minua laittamaan viesti työtuntiraportointijärjestelmän ylläpitäjille. Myöhemmin päivällä sain viestin, että ongelma oli korjattu, ja sain lähetettyä tuntiraportin.

Sain vihdoinkin valmiiksi käsittelyn linkeille, joilla halutaan navigoida kysymättä käyttäjältä haluaako hän tallentaa muutokset. Osallistuin päiväpalaveriin ja sprintin demon suunnitteluun.

Perjantai 18.9.2015

En tullut tänään, koska julkisessa liikenteessä oli katkoja suurmielenosoituksen takia. Yksi tiimini jäsen kävi läpi demodiani puolestani.

Viikkoanalyysi

Erityisesti minulle jäi mieleen viikosta seuraava JavaScript-koodinpätkä:

```
if (status !== 400) {
  self.enableAll(false);
} else {
  /* bad request */
  self.enableAll(true)
}
```

Aluksi ajattelin ehdottaa vain että hän siirtää HTTP-statuskoodin nimen pois kommentista koodiin itseensä:

```
var BAD_REQUEST = 400;
if (status !== BAD_REQUEST) {
  self.enableAll(false);
} else {
  self.enableAll(true);
}
```

Mutta sitten huomasin, että koodin ehdon voi kääntää ja siirtää `enableAll`-metodin argumentiksi:

```
var BAD_REQUEST = 400;
self.enableAll(status === BAD_REQUEST);
```

Minun mielestäni oli erityisen tärkeää siirtää statuskoodin nimi koodiin itseensä, koska uskon vahvasti, että koodin pitäisi ”puhua omasta puolestaan” mahdollisimman paljon. Minulla on tähän kaksi syytä. Ensimmäiseksi, ohjelmointikielet on keksitty, jotta ohjelmia olisi helpompi kirjoittaa ja lukea, joten koodin ei pidä olla pelkästään tietokoneille, vaan myös

ihmisille luettavassa muodossa. Toiseksi, olen huomannut omassa ja muidenkin ohjelmoinnissa, että kommenttien päivitys usein unohtuu koodia päivittäessä, ja silloin koodin ja kommenttien antamat viestit eroavat toisistaan, mikä aiheuttaa sekaannusta ja ongelmia. Mielestäni koodissa voi olla kommentteja, mutta niiden tulisi lähinnä kuvata *miksi* koodin on kirjoitettu, niin kuin se on kirjoitettu, esimerkiksi jos koodissa on käytetty jotain outoa ratkaisua, tavallisen ratkaisun sijaan.

Jeff Atwood (2006), toinen *Stack Overflown* perustajista, pitää luokkien ja muuttujien uudelleennimeämistä yhtenä yleisimpänä refaktorointitehtävänä. Hän myöntää, että hyvä nimeäminen on vaikeaa, mutta sen *pitääkin* olla vaikeaa, koska hyvä nimi kiteyttää tarvittavan tiedon vain pariin sanaan. Hän myös pitää koodin selkeälukuisuutta tärkeänä, koska silloin kehittäjät pitävät koodin kanssa työskentelystä eniten.

Linkkiasiaa selvittäessä opin, että kun HTML-linkkiä napsauttaa, se ensin laukaisee `MouseEvent`-, eli hiiritapahtuman, ja ennen kuin ikkuna/välilehti yrittää navigoida uuteen osoitteeseen, se taas laukaisee `HashChangeEvent`-, eli ristikkomerkkimuutostapahtuman. Koska näillä kahdella tapahtumalla ei ole tietoa toisistaan, minun oli pakko luoda hiiritapahtumakuuntelija, joka merkitsee muistiin että tällaista erityslinkkiä on napsautettu. Sitten minun piti muuttaa ristikkomerkkimuutostapahtuma sellaiseksi, että se päästää navigoinnin läpi, ja samalle poistaa hiiritapahtumassa tehdyn merkinnän.

3.3 Seurantaviikko 3

Maanantai 21.9.2015

Kun tulin töihin, tuoteomistaja kertoi minulle ongelmasta eräässä alustamme näytedemoista. Ongelma on, että kun Internet Explorerilla painoi nappia, jonka tarkoitus oli tuoda esiin varoitusviestin, selain siirtyikin näytedemosovelluksen etusivulle. Aioin korjata tämän ongelman, ja osallistua päiväpalaveriin.

Kun tutkin ongelmaa, huomasin eri selainten käyttäytyvän eri tavalla kun nappia painettiin. Firefoxilla napinpainallus toi varoitusviestin näkyviin, kuten oli tarkoituskin. Google Chromella, kun nappia painoi ensimmäisen kerran, selain lisäsi kysymysmerkin sivun osoitteen loppuun, ja latasi sivun uudestaan. Tämän jälkeen nappi toimi kuten pitikin. Kuten jo mainitsin, Internet Explorerilla napinpainallus vei sovelluksen etusivulle.

Tutkiskeltuani koodia, huomasin napin elementin olevan `<form>`-, eli lomake-elementin sisällä, ja sille ei ollut määritelty `type`-, eli tyyppi-attribuuttia. Muistin aikaisemmin lukeneeni, että `<button>`- eli nappielementin tyypin oletusarvo on `"submit"`, eli ”lomakkeen palautus”. Koska nappi oli lomakkeen sisällä ja se oli tyypiltään lomakkeenpalautusnappi, sitä painaessa Internet Explorer ja Chrome tekivät lomakkeenpalautuksen, joka aiheutti Internet Explorerilla siirtymisen sovelluksen etusivulle, ja Chromella kysymysmerkin lisäämisen sivun osoitteeseen.

Ongelman korjaus oli näin ollen helppo: lisää napille tyyppi, jonka arvo on `"button"`, eli ”nappi”. Näin ollen nappi ei tee mitään muuta kuin mitä sille on määritelty komennoilla. Päätin lisätä saman lomakkeen muillekin napeille tyypit, koska niitä ei ollut määritelty, ja ne olivat kaikki selkeästi tarkoitettu olevan ”nappi”-tyyppiä.

Tiistai 22.9.2015

Olin tämän päivän kotona koska minulla oli paha olo.

Keskiviikko 23.9.2015

Tavoitteeni tänään on osallistua päiväpalaveriin ja käydä läpi yrityksen e-opetuskurssin: ”Testaus ohjelmistokehittäjille, osa 1”. Ensimmäisen osassa käydään läpi testauksen periaatteet, yksikkötestaus, koodikattavuusanalyysi, ja yhtäläisyysosittelu ja raja-arvotestaaminen.

Kurssin mukaan on ohjelmistoinsinöörin vastuu, että koodi toteutetaan, yksikkötestataan, ja yksikköintegraatiotestataan. Lisäksi on erityisen tärkeää, että testit nimetään kuvaavasti, ja ettei testitapauksia ja -tuloksia päällekirjoiteta, jotta niitä voi tutkia myöhemmin. Jotta toteutus on yrityksen standardien mukainen, se pitää olla versionhallinnassa, se pitää olla tarpeeksi kypsä testitiimin järjestelmätestaukseen, ja sille pitää olla toistettavia testejä.

Seuraavaksi kurssissa lueteltiin yleisimpiä staattisia testausmetodeja, kuten staattinen analyysi (esimerkiksi ”data flow”-, eli datavuoanalyysi ja ”control flow”-, eli seurantavuoanalyysi), muodolliset ja vapaamuotoiset koodikatselmoinnit, tekniset katselmoinnit, ”walkthroughs”, eli ohjelman käytön läpikäynnit; ja ”inspection”, eli tarkastelu. Tarkastelun voi tehdä myös automaattisilla työkaluilla, esimerkiksi JavaScript-koodia voi tarkastella JSLint-työkalulla.

Kurssissa lueteltiin myös dynaamisia testausmetodeja, jotka oli jaettu kolmeen ryhmään. Ensimmäinen ryhmä oli koodin rakenteen tutkimiseen pohjautuvat menetelmät: lausekkeiden, valintojen, ehtojen, ja moniehtojen tutkimiset. Toinen ryhmä oli kokemukseen pohjautuvat menetelmät: virheiden arvaaminen, ja "exploratory testing", eli tutkiva testaaminen. Kolmas ryhmä oli määrittelyyn ja laatuvaatimuksiin perustuvat metodit: käyttötapaustestaus, päätöstaulut, tilanmuutostaulut, raja-arvoanalyysi, ja vastaavuusosittelu.

Kurssin osa yksikkötestaamisesta heti painotti, että yksikkötestaaminen on usein huomiomatta jätetty ja väärinymmärretty toiminta, joka voi oikein käytettynä lisätä kehitysprosessin tehokkuutta ja laatua huomattavasti. Yksikkötestauksessa yksiköllä yleensä tarkoitetaan luokkaa tai funktiota, ja niissä koodi ajetaan ja sen lopputulosta verrataan tarkoitettuun lopputulokseen. Kurssi myös painotti, että yksikkötestauksen suoranainen tarkoitus ei ole löytää virheitä, vaan tarkistaa että koodi toimii oikein. Yksikkötestaus ei ota kantaa, miten yksiköt toimivat keskenään, koska tämä kuuluu yksikköintegraatiotestaukseen. Lisäksi yksikkötestit varmistavat, että koodi toimii oikein muutosten jälkeen, ja jos muutokset vaikuttavat testeihin, myös testejä pitää muuttaa. Tällä tavoin yksikkötestejä voi käyttää yksikköjen toimintojen hallintaan ja määrittelyyn, ja näin ollen toimivat myös yksikön dokumentaationa. Jotta yksikkötestit toimisivat dokumentaationa parhaiten, niiden tulee olla luettavia, ja niistä pitäisi selkeästi nähdä mitä koodi tekee.

Kurssiin sisältyi myös osio koodikattavuusanalyysistä, johon tarkoitetuilla työkaluilla, kuten JCov, Cobertura, ja Serenity; voi tarkistaa, millä koodin osilla on yksikkötestejä ja millä ei. Kolme yleisintä kattavuuslaskemismetodia ovat: lausekekattavuus, haarakattavuus, ja reittikattavuus. Lausekekattavuus on näistä yksinkertaisin, koska se vain katsoo, mitkä koodirivit ajetaan testeissä. Haarakattavuus on monimutkaisempi; se katsoo onko kaikki haarat ajettu testeissä. Reittikattavuus on kaikkein monimutkaisin, koska se katsoo onko kaikki haarojen mahdolliset yhdistelmät, eli reitit, testeissä mukana.

Lopuksi ensimmäinen osa syventyi aikaisemmin mainitsemaansa raja-arvoanalyysiin, ja vastaavuusositteluun. Vastaavuusosittelulla tarkoitetaan mahdollisen datan ryhmittelyä. Esimerkiksi aineen olomuodon voi lämpötilan mukaan ryhmitellä kiinteään, nesteeseen, ja kaasuun. Tällöin mahdollisten testitapausten määrä supistetaan hallittavaan määrään, koska koko ryhmän arvoja voi edustaa yhdellä arvolla.

Kurssin mukaan raja-arvoanalyysi on syntynyt havainnosta, että ohjelmien virheet usein syntyvät lähellä raja-arvoja. Koodi saattaa toimia hyvinkin eri tavalla raja-arvojen eri puolilla. Esimerkiksi veden olomuodolle tärkeät raja-arvot ovat 0 ℃, ja +100 ℃. Näin ollen hyvät arvot testaamiseen ovat −1 ℃; +1 ℃; +99 ℃; ja +101 ℃.

Osallistuin päiväpalaveriin. Pääsin e-opetus kurssin ensimmäisestä osasta läpi.

Torstai 24.9.2015

Olin tämän päivän kotona koska minulla oli paha olo.

Perjantai 25.9.2015

Kirjoitin tänä päivänä päiväkirjamerkintöjä, koska olin hiukan jäljessä niissä.

Viikkoanalyysi

Tällä viikolla minulla oli ongelmia saada töitä tehtyä poissaolojen takia. Poissaolojen syy oli se, että olin kotona masentuneena Volkswagen-päästöskandaalista. Tiistaina paljastui, että Volkswagen on huijannut sen dieselautojen typpioksidipäästöissä lisäämällä autojensa moottoreihin huijausohjelman. Ohjelma oli tehty niin, että se tunnisti testauksessa käytetyt olosuhteet, ja komensi silloin moottoria pidättelemään päästöjä. Tämän huijausohjelman avulla Volkswagen on pystynyt myymään noin yksitoista miljoonaa autoa, joiden typpioksidipäästöt ovat jopa 35 kertaa suurempia kuin on laillista.

Mikä tässä tilanteessa minua erityisesti masentaa on että ympäristösäädökset ovat yleensä heikompia kuin mitä tiedeyhteisö suosittelee, ja usein – niin kuin tässä tapauksessa – edes näitä laimennettuja sääntöjä ei noudateta. Tiede asettaa riman tarvittavalle korkeudelle, poliitikot alentavat rimaa, ja yritykset ja valtiot vielä alittavat alennetunkin riman. Tällaisella menettelyllä ympäristötavoitteet jäävät aina puolitiehen, ja esimerkiksi ilmastonmuutoksen annetaan pahentua.

Tämä skandaali vaikutti minuun erityisesti, koska huijauksessa käytettiin tietokoneohjelmaa. Kyseinen ohjelma on niin sanottu ”omisteinen ohjelma”, eli se ei ole avoimen tai edes jaetun lähdekoodin ohjelma. Tämä puolestaan tarkoittaa, että ohjelman koodia ei voi suoraan tutkia; sitä voi tutkia vain seuraamalla sen toimintaa. Software Freedom Law Center-yhdistyksen puheenjohtaja Eben Moglen (2010) vertasi tätä oikeutta tutkia koodia oikeuteen tarkastaa hissin koneisto. Tästä syystä hän tuomitsi omisteisen koodin epäluotettavaksi rakennusmateriaaliksi. Nyt päästöskandaalin johdosta *The New York Timesin* kolumnisti Jim Dwyer (2015) kutsui Moglea ”profeetaksi”. Viitaten *The New York Timesin* artikkeliin, aktivisti David Bollier (2015) sanoi, että Volkswagen ei todennäköisesti olisi huijannut, jos koodi olisi ollut vapaasti tutkittavissa. Hän jopa sanoi että koodi, jota ei voi

tutkia, houkuttelee yrityksiä rikkomaan lakia, koska rikkomusta on vaikea todistaa. Bollier eteenkin harmitteli sitä, että Yhdysvaltain ympäristövirasto EPA *puolustaa* koodin salaisuutta tekijänoikeuslakien nojalla, todennäköisimmin isojen autovalmistajien vaatimuksesta. Bollier vielä peräänkuulutti, että kaikissa turvallisuutta tarvitsevissa laitteissa – kuten lentokoneissa, lääketieteellisissä laitteissa, lukkiutumattomissa jarruissa, ja kaasupolkimien ohjaimissa – koodin pitäisi olla tutkittavissa, jotta tällaiset tapaukset eivät toistuisivat.

Usein, kun ohjelmiston vapautta ajavat järjestöt valittavat omisteisesta ohjelmistosta, heidän näkemystään pidetään vain naiivina idealismina, mutta tällaisista tapauksista näkee, ettei vapaus ole pelkkä aate, vaan sillä on myös käytännön vaikutuksia oikeassa elämässä.

3.4 Seurantaviikko 4

Maanantai 28.9.2015

Tavoitteeni tänään on osallistua päiväpalaveriin ja liittää yksi versionhallintahaarassa olevan toiminnon päähaaraan. Tämä toiminto on 17.9. valmiiksi saamani HTML-luokka, jonka pystyi lisäämään linkkiin, jotta linkki ei pyydä hyväksymään navigointia kun lomakkeeseen on tehty muutoksia. Koska samaa HTML-luokkaa käytetään myös lomake-elementeissä samanlaiseen tarkoitukseen, minun pitää myös päivittää dokumentaatio sanomaan, että luokkaa voi käyttää myös linkeille.

Osallistuin päiväpalaveriin.

Testasin vielä varmuuden vuoksi, että toiminto toimii. Liitin haaran päähaaraan ongelmitta. Päivitin dokumentaation.

Puhuin yhden kehitystiiminjäsenen kanssa `position: sticky`-CSS-säännöstä, jolla saa tehtyä elementin, jolla on tietty staattinen paikka sivulla, mutta kun sivua vierittää, elementti pysyy ruudulla absoluuttisesti määritellyssä paikassa. Sääntö on hyödyllinen, jos haluaa esimerkiksi luoda ylänavigaatiopalkin, joka pysyy aina ruudulla. Erityisesti mietimme, että kannattaako meidän käyttää jQuery-kirjaston liitännäistä tämän efektin luomiseen, vai tätä sääntöä. Koska sääntö ei ole vielä toteutettuna kaikissa selaimissa, jotta sitä voi käyttää, täytyy käyttää Stickyfill-nimistä polyfilliä. Olin hänen kanssaan samaa mieltä, että meidän kannattaa käyttää tätä CSS-sääntöä (polyfillillä) liitännäisen sijasta, koska tulevaisuudessa voi olettaa, että selaimet tulevat tukemaan tätä sääntöä enemmän

ja enemmän. Liitännäisen tuki todennäköisesti loppuu jossain vaiheessa. Lisäksi silloin kun kaikki selaimet tukevat tätä sääntöä, meidän tarvitsee pelkästään poistaa sen polyfill, kun taas liitännäisestä CSS-sääntöön siirtyminen todennäköisesti vaatisi isompia toimenpiteitä.

Tiistai 29.9.2015

Tavoitteeni tänään on osallistua päiväpalaveriin, ja jatkaa ennen julkaisua tehtyjen versionhallintasivuhaarojen liittämistä päähaaraan.

Liitin maanantaina 21.9. tekemäni painikekorjauksen tuotteen versiohallinnan päähaaraan. Osallistuin päiväpalaveriin. Tällä kertaa kehitystiiminjäsen, joka yleensä liittyy Skypen välityksellä, olikin paikalla.

Liitin maanantaina 20.7. tekemäni muutoksen tuotteen versiohallinnan päähaaraan. Muutos liittyi ponnahdusikkunamaiseen komponenttiin, jota käytetään kun lomakkeessa jokin kohta tarvitsee yksityiskohtaista tietoa, mutta tätä tietoa ei haluta pitää näkyvillä koko ajan. Komponentti tyypillisesti tulee esille jollain painikkeella. Aiemmin komponentti oli toteutettu niin, että oletuksena sen pystyi sulkemaan vain painamalla sen sisällä olevaa ”piilota”-painiketta. Muutoksessani vaihdoin komponentin oletuksena piiloutumaan, kun käyttäjä napsauttaa hiirellä jotakin komponentin ulkopuolella. Tämä muutospyyntö tuli UX-tiimiltä.

Muutoksen piti näkyä heti UX-tiimille, mutta samalle emme voineet tehdä muutosta vielä suoraan alustaamme, koska toisen tuotteen julkaisu oli lähellä, ja halusimme välttää ongelmien tuottamista tälle tuotetiimille. Tämän takia minua pyydettiin muokkaamaan ensiksi alustamme esimerkkejä; nämä muutokset laitettiin suoraan päähaaraan. Sitten tein muutoksen komponentin oletusasetukseen ja lisätyn asetuksen poiston, jotka laitoin sivuhaaraan julkaisun jälkeistä aikaa varten.

Keskiviikko 30.9.2015

Tavoitteeni tänään on osallistua päiväpalaveriin ja poistaa eräs erikoinen linkkien käsittely.

Tuotealustallamme oli ennen ongelma, joka tarvitsi erityistä linkkien käsittelyä. Ennen, jos käyttäjä oli tehnyt tallentamattomia muutoksia, ja hän yritti navigoida jollekin ihan toiselle verkkosivustolle, sovellus ei näyttänyt varoitusta tallentamattomista muutoksista, niin kuin sen piti. Tämän ongelman sivuuttamiseksi joku oli keksinyt tehdä käsittelyn, jossa ulkoisen

sivuston osoite laitettiin linkin ristikkomerkkiosaan. Tällöin, kun sovellus huomasi muutoksen ristikkomerkkiosassa, se katsoi, oliko käyttäjällä tallentamattomia muutoksia. Jos oli, sovellus näytti varoituksen ja ehdotti toimenpiteitä. Jos ei, sovellus ohjasi selaimen osoitteeseen ristikkomerkkiosassa. Tietääkseni tätä käsittelyä käytettiin vain sovelluksen ylänavigaatiolinkeissä.

Myöhemmin joku toinen tiimin jäsen korjasi tallentamattomien muutosten varoituksen toimimaan tavallisillakin ulkoisilla linkeillä. Tämän ansiosta pystyin muuttamaan ylänavigaatiolinkit tavallisiksi linkeiksi keskiviikkona 12.08. Koska emme olleet täysin varmoja, että ylänavigaatiolinkit olivat ainoa paikka, jossa käytettiin tätä erikoiskäsittelyä, päätimme poistaa tämän käsittelyn vasta julkaisun jälkeen, jottei se aiheuttaisi ongelmia.

Osallistuin päiväpalaveriin, ja tutkin mitä muutoksia minun tarvitsi tehdä ulkoisten linkkien erikoiskäsittelyn poistamiseksi. Eteenkin kiinnitin huomiota käsittelyn tekevän komponentin Jasmine-yksikkötestien päivittämiseen, koska ne olivat aika kehnosti kirjoitettu. Jasmine käyttää erityistä syntaksia, jossa testit ryhmitellään `describe`-, eli kuvailufunktiokutsuilla, joihin kuvailtavan asian nimi laitetaan ensimmäisenä parametrina. Kuvailtavan asian nimi pitäisi tietysti olla substantiivi. Itse testit ovat `it`-, eli ”se”-funktion kutsuja, ja niissä ensimmäinen parametri on testattava toiminto, joka on verbimuodossa. Tällöin oikeaoppisesti tehdyt testit ovat esimerkiksi muodossa: ”describe Scanner: it can read files”, eli ”kuvaile Scanner: se voi lukea tiedostoja”. Tästä käytännöstä huolimatta, monessa testeistämme oli esimerkiksi substantiiveja ”se”-funktion kutsuissa, mikä tekee testien koodista rumaa, ja vaikeata lukea.

Torstai 1.10.2015

Tavoitteeni tänään on osallistua päiväpalaveriin ja saada valmiiksi ulkoisten linkkien erityiskäsittelyn poisto.

Sain valmiiksi ulkoisten linkkien erityiskäsittelyn poiston. Poistin myös erityiskäsittelyn yksikkötestin, ja siistin hiukan erityiskäsittelyn tekevän komponentin yksikkötestejä. Poistin erityskäsittelyn selittävän kohdan komponentin dokumentaatiosta, ja lisäsin migraatiodokumentaatioon merkinnän, jossa kehotin migraatiota tekevät päivittämään kaikki linkit jotka vielä tarvitsevat erityiskäsittelyä. Koska erityiskäsittelyn poisto oli sen verran iso muutos, ja en ollut täysin varma että tein sen oikein, aloitin muutoksesta koodikatselmoinnin, jonka annoin kahden muun kehitystiimin jäsenen katselmoitavaksi. Yksi katselmoijista sai katselmoinnin valmiiksi, kommentoimatta.

Koska sprintin demo oli suunniteltu perjantaiksi, keskustelin demon suunnittelijan kanssa muutoksista, jotka minä halusin esitellä. Täytin demodiani demoa varten.

Perjantai 2.10.2015

Tavoitteeni tänään on esittää tekemäni muutokset sprintin demossa, osallistua päiväpalaveriin, ja kirjoittaa opinnäytetyötä.

Esitin tekemäni muutokset demossa ongelmitta. päiväpalaveria ei pidetty, koska varascrummaster ei ollut paikalla sen ajankohtana. En oikein pystynyt kirjoittamaan opinnäytetyötä, koska olin väsynyt demon aikaisuuden takia.

Viikkoanalyysi

Viikon kaikki työtehtävät olivat asioita, jotka oli aikaisemmin suunniteltu tehtäväksi, mutta ne pystyttiin tekemään vasta nyt kun toisen tuotteen julkaisu oli valmis. Tiimillämme on yleensäkin ongelmia sprinttien ja julkaisujen kanssa, mutta erityisesti tällä ja viime viikolla. Yksi ongelma on, että tiimi pitämät sprintin demot vievät liikaa aikaa. Demo vie yleensä sen osallistujilta puolet sitä edeltävästä päivästä valmistelulla, ja noin puolet itse demopäivästä. Toinen ongelma on toistuvat julkaisut. Tiimi ei pelkästään työskentele oman julkaisun eteen, vaan myös auttaa kahta muuta tiimiä julkaisuissaan. Julkaisuihin liittyy paljon kiireistä työtä, joka usein pakottaa kehittäjät jättämään sen hetkisen työnsä kesken, ja jatkamaan myöhemmin. Näitä muiden tiimien auttamisia ei yleensä lasketa sprinttien suunnitteluissa, koska ne ovat arvaamattomia.

Näiden ongelmien takia, tiimi on suunnitellut siirtymään käyttämään kanbania. Kanban on scrumin tapaan ketterä kehitysmenetelmä. Kanbanissa ei työskennellä sprinteissä, vaan työskentely on yhtäjaksoista. Kanban keskittyy kanbantauluun (Kuvio 2), jonka avulla seurataan ja ohjataan työskentelyä. Taulu on jaettu sarakkeisiin, jotka edustavat jokaista tarvittavaa työvaihetta. Lisäksi sarakkeet ovat yleensä jaettu alisarakkeisiin: ”työn alla” ja ”valmis siirtymään”. Kun kehittäjä saa yhden tehtävän valmiiksi, hän voi ”vetää” siirrettäväksi valmiin tehtävän seuraavan sarakkeen ”työn alla”-alisarakkeeseen, ja merkitä itsensä tehtävän työstäjäksi. Kun hän on valmis työvaiheen kanssa, hän laittaa tehtävän ”valmis siirtymään”-sarakkeeseen. Tehtäviä lisätään taulun alkupäähän kun niitä tulee, ja tehtäviä poistetaan loppupäästä kun julkaisu tehdään. Taulu voi olla fyysinen taulu, jossa tehtävät ovat muistilappuja; tai sähköinen taulu vaikka yrityksen intranet-sivustolla.

Suunniteltu	Design		Kehitys		Katselmointi	Valmis julkaistavaksi
	Työn alla	Valmis siirtymään	Työn alla	Valmis siirtymään		

Kuvio 2. Esimerkki kanbantaulusta

Siirtyminen kanbaniin vaikuttaa minusta järkevältä ratkaisulta edellä mainituista syistä. Olen kuitenkin hieman huolissani muun organisaation suhtautumisesta siihen. Tämän hetkinen tiimien välinen kommunikointi vaikuttaa olettavan, että kaikki tiimit käyttävät scrumia. Esimerkiksi, ”Program Increment”-jaksojen suunnitteluissa pitää aina määrittää minä sprinttinä jokin toiminto tulee valmiiksi. Tämä ei tietenkään toimi, jos tiimi käyttää kehitys metodia, jossa ei ole sprinttejä.

Yrityksemme on myös siirtymässä ”Scaled Agile Framework (SAFe)”-kehitysviitekehykseen. Sen tarkoitus on helpottaa ketterää kehittämistä suurissa organisaatioissa. Olen kuullut työtovereilta, että SAFe helpottaa tiimien työskentelyä järjestämällä työskentelyn ”julkaisujuniin”, joiden avulla tiimit voivat seurata toistensa aikaansaannosta.

Vaikka en tiedä SAFe:sta vielä paljoakaan, minulla on jo nyt huono tunne siitä. Yksi syy on se, että SAFe:n verkkosivusto, scaledagileframework.com, on suojattu vahvalla käyttöoikeuksien hallintamekanismilla, joka pyrkii estämään tekstin ja kuvien kopioimisen, ja estää myös tyypillisten selaintoimintojen, kuten pikanäppäinten ja pikavalikon käytön. Lisäksi sivuston *About*-sivulla (Leffingwell, 2016) on kielletty kuvien ja tekstin kopioiminen ilman kirjallista lupaa. Vaikka käyttöoikeuksien hallintamekanismit ovat yleisiä musiikissa, DVD-

elokuvissa, peleissä, ja e-kirjoissa; ne ovat lähes ennenkuulumattomia verkkosivustoissa. Toinen syy on se, että yksi Agile Alliancen perustajista, Ken Schwaber (2013), on jyrkästi arvostellut SAFe:a liian prosessipainotteiseksi, eikä tarpeeksi ihmislähtöiseksi; ja verrannut sitä vanhanajan vesiputous- ja RUP-kehitysmenetelmiin.

3.5 Seurantaviikko 5

Maanantai 5.10.2015

Käytin tämän päivän opinnäytetyön kirjoittamiseen. Osallistuin myös päiväpalaveriin.

Tiistai 6.10.2015

Käytin tämän päivän opinnäytetyön kirjoittamiseen kotona.

Keskiviikko 7.10.2015

Tavoitteeni tänään on osallistua päiväpalaveriin, ja jatkaa keskiviikkona 23.9. aloittamaani ”Testaaminen ohjelmoijille”-koulutusta.

Osallistuin päiväpalaveriin, ja pääsin koulutuksesta läpi. Kurssin toisessa osassa käytiin läpi yksikköintegraatiotestaus, testivetoinen kehitys, ”mock”-oliotestaus, ja jatkuva integraatio.

Ensimmäiseksi kurssin toisessa osassa kerrottiin yksikköintegraatiotestauksesta, jota usein kutsutaan pelkäksi integraatiotestaukseksi. Integraatiotesteissä on tärkeä ottaa huomioon muita ulottuvuuksia kuin yksikkötesteissä, kuten: toiminallisuus, suorituskyky, luotettavuus, ja vioista palautuminen. Integraatiotestauksessa voi käyttää kolmea lähestymistapaa:

- **Ylhäältä alas -tapa**: Testaus aloitetaan suurimpien osien vuorovaikutuksista, ja sitten siirrytään pienempiin osiin. Hyödyllinen kun ohjelman ylimalkainen arkkitehtuuri on jo suunniteltu.
- **Alhaalta ylös -tapa**: Testaus aloitetaan pienimpien osien vuorovaikutuksista, ja sitten siirrytään suurempiin osiin. Hyödyllinen kun ohjelma kehitetään ketterästi.
- **”Sandwich”-, eli kerrostamistapa**: Kahden muun tavan yhdistelmä/sekoitelma.

Toiseksi osassa puhuttiin testivetoisesta kehityksestä. Testivetoisessa kehityksessä testit tehdään *ennen* koodia. Testivetoisen kehityksen keksi Kent Beck ja se on erityisesti tarkoitettu lyhyiden jaksojen kehitykseen, joka on yleistä ketterissä ohjelmistokehitystekniikoissa kuten scrumissa, ja extreme programming-, eli XP-metodissa. Testivetoisen kehityksen uskotaan parantavan suunnittelua, rajapintoja, integraatiota, koodin modulaarisuutta, koodin joustavuutta, ja koodin laajennettavuutta.

Testivetoinen kehitys on jaettu kolmeen askeleeseen:

1. **Testin kirjoittaminen**: Tämä vaatii asiakkaan vaatimuksien tuntemisen. Testin tulisi epäonnistua tässä vaiheessa.
2. **Koodin kirjoittaminen**: Koodi kirjoitetaan mahdollisimman pelkistetysti. Koodin tulisi nyt läpäistä testit.
3. **Koodin refaktorointi**: Koska koodi läpäisee testit, refaktorointi ei voi enää rikkoa toiminnollisuutta, koska silloin testit hylkäisivät koodin.

Kolmanneksi osassa puhuttiin ”mock”-oliotestauksesta. Yksikkötestauksessa nimensä mukaisesti halutaan keskittyä pelkästään yhteen testattavaan yksikköön, ja sivuuttaa muut tekijät. Yksiköt kuitenkin usein riippuvat toisista yksiköistä. Tätä varten testauksessa on hyvä luoda ”mock”-, eli ”vale”-olioita, jotka ovat ulkoisesti käyttäytyvät samalla tavalla kuin aidot yksiköt, mutta niissä ei ole aitojen yksiköiden sisäistä logiikkaa. Jotta ”mock”-testaus on mahdollista, metodit täytyy määritellä niin, ettei niissä käytetä tarvittavaa luokkaa. Sen sijaan luodaan rajapinta, jonka luokka täyttää. ”Mock”-olio voi sitten myös käyttää tätä rajapintaa. Kurssi luetteli kolme Java-”mock”-testausviitekehystä: EasyMock, Mockito, ja PowerMock.

Viimeiseksi kurssin toisessa osassa puhuttiin jatkuvasta integraatiosta. Jatkuvassa integraatiossa ohjelmisto käännetään ja testataan automaattisesti, kun koodiin tehdään muutoksia. Tämä vähentää kehittäjien työtä, koska heidän ei tarvitse ajaa testejä ja odottaa tuloksia itse. Lisäksi jatkuva integraatio varmistaa, että ohjelmisto pysyy koko ajan testien mukaisena, ja testejä rikkova koodi huomataan mahdollisimman nopeasti. Testien tulee olla jokseenkin kattavat, jotta jatkuvasta integraatiosta on mahdollisimman paljon hyötyä. Esimerkiksi Jenkins-palvelinohjelmaa käytetään jatkuvassa integraatiossa.

Kehitystiimillämme on hyllyjen päällä iso tietokoneruutu, josta näkyy eri moduulien testien tila. Tämän ansiosta tiimin jäsenet näkevät heti, kun testit epäonnistuvat jossain moduulissa. Lisäksi, kun testit epäonnistuvat, testipalvelin lähettää sähköpostiviestin sille, joka teki muutoksia koodiin viimeiseksi, koska on todennäköistä, että vika johtuu hänen koodistaan.

Torstai 8.10.2015

Tavoitteeni tänään on osallistua päiväpalaveriin ja valmistautua seuraavan päivän Robot Framework -koulutukseen.

Osallistuin päiväpalaveriin ja tutkin Robot Frameworkia alustavasti. Robot Framework on testiautomaatioviitekehys hyväksyntätestaamiseen. Se käyttää niin sanottua ”avainsana-pohjaista” testausmetodia, jota kutsutaan myös taulukko- ja toimintasana -pohjaiseksi testaamiseksi. Avainsana-pohjaisessa testauksessa testitapaukset kirjoitetaan mahdollisimman selkokielisesti taulukoihin. Tämän ansiosta testitapauksia voi käyttää automaattisessa *ja* manuaalisessa testauksessa. Lisäksi näin ollen testitapaukset eivät ole kovin ympäristösidonnaisia, ja muutokset käyttöliittymässä ja käyttöjärjestelmässä ei aiheuta yhtä paljon ongelmia, kuin jos testitapaukset olisi kirjoitettu koodilla.

Perjantai 9.10.2015

Tavoitteeni tänään on osallistua koko päivän mittaiseen Robot Framework -koulutukseen.

Robot Framework -koulutukseen osallistui noin tusina ihmistä joista neljä kuului kehitystiimiini. Koulutusta veti kaksi kouluttajaa Omenia Oy:sta. Koulutus alkoi diaesityksellä, jossa vetäjät kertoivat Robot Frameworkista. Esityksestä opin, että Robot Framework sai alkunsa Nokia Networksissa, joka myös sponsoroi sitä nykyään. Se on kirjoitettu Pythonilla, mutta siitä on myös versiot Java-, ja .NET-ympäristöissä. Lisäksi sille on kirjoitettu kirjastoja Java-, .NET-, Perl-, JavaScript-, ja PHP-ympäristöille. Testitapaukset voi kirjoittaa HTML-, reStructuredText-, ja Gherkin-syntakseilla. Koulutus kuitenkin suositteli käyttämään Robot Frameworkin omaa tekstiformaattia, joka käyttää `.robot`-tiedostopäätettä.

Diaesityksen jälkeen siirryimme tehtävien tekemiseen. Esittäjät jakoivat muistitikut, joissa oli Pythonilla tehty verkkosovellus. Sovellus oli kirjautumissivu, joka johti ”Tervetuloa”-sinulle, jos käyttäjätunnus ja salasana olivat oikeat, ja ”Virhe”-sivulle, jos ne olivat väärät. Sovellus oli tarkoituksella pelkistetty, jotta huomio kiinnittyisi testaamiseen. Esimerkiksi sovellus ei luonut kirjautumiseen istuntoa, tunnuksen ja salasanan tarkastus tehtiin vain JavaScriptilla, ja yhteydessä käytettiin salaamatonta HTTP-yhteyttä, salatun HTTPS-yhteyden sijaan.

Ensimmäisenä tehtävänä vetäjät pyysivät meitä tekemään testitapaus onnistuneelle kirjautumiselle. Testitapauksen tekemisen jälkeen se ajettiin `pybot`-Python-komennolla, joka

luo testeistä raportin ja lokin HTML-muodossa. Niistä näkyy, mitkä testit menivät läpi, ja mitkä testit eivät menneet läpi. Lisäksi lokista pystyy näkemään tarkasti missä kohtaa testi mahdollisesti epäonnistui.

Seuraavaksi vetäjät opettivat avainsanojen, HTML-elementtien paikantimien, muuttujien, testien parametrien, validaation, testien asetus- ja purkamiskomentojen, ja resurssitiedostojen käytöstä. Vetäjät pyysivät tekemään testit kaikille mahdollisille epäonnistuneen kirjautumisen yhdistelmille, ja sitten poistamaan koodista kaikki turha toisto käyttämällä edellä mainittuja tekniikoita.

Viikkoanalyysi

Esimerkki yhdestä koulutuksen testitiedostoista:

```
*** Settings ***
Default Tags    regressio
Resource    kirjautumiset.resource.robot
Test Teardown    Avataan Login Page
Test Template    Kirjautuminen epäonnistuu

*** Variables ***
${VÄÄRÄ KÄYTTÄJÄTUNNUS}    väärä käyttäjätunnus
${VÄÄRÄ SALASANA}    väärä salasana

*** Test Cases ***
Oikea käyttäjä, tyhjä salasana    ${OIKEA KÄYTTÄJÄTUNNUS}    ${EMPTY}
Oikea käyttäjä, väärä salasana    ${OIKEA KÄYTTÄJÄTUNNUS}    ${VÄÄRÄ SALASANA}
Tyhjä käyttäjä, oikea salasana    ${EMPTY}    ${OIKEA SALASANA}
Tyhjä käyttäjä, tyhjä salasana    ${EMPTY}    ${EMPTY}
Tyhjä käyttäjä, väärä salasana    ${EMPTY}    ${VÄÄRÄ SALASANA}
Väärä käyttäjä, oikea salasana    ${VÄÄRÄ KÄYTTÄJÄTUNNUS}    ${OIKEA SALASANA}
Väärä käyttäjä, tyhjä salasana    ${VÄÄRÄ KÄYTTÄJÄTUNNUS}    ${EMPTY}
Väärä käyttäjä, väärä salasana    ${VÄÄRÄ KÄYTTÄJÄTUNNUS}    ${VÄÄRÄ SALASANA}

*** Keywords ***
Kirjautuminen epäonnistuu
    [Arguments]    ${käyttäjätunnus}    ${salasana}
    Syötetään käyttäjätunnus    ${käyttäjätunnus}
    Syötetään salasana    ${salasana}
    Painetaan Login-nappia
    Virhesivu aukeaa
Virhesivu aukeaa
    Title Should Be    Error Page
```

Tämä tiedosto on itseasiassa neljä taulukkoa kirjallisesti formatoituna. Kolmen tähtimerkin välissä olevat tekstit ovat taulukoiden otsikot, ja niiden alla on taulukot itse. Taulukoiden rivit erotellaan vähintään kahdella välilyönnillä, tai yhdellä tabulointimerkillä. Esittäjät kuitenkin suosittelivat käyttämään neljää välilyöntiä, koska sitä määrää käytetään yleensä Python-kielessä.

Esimerkistä huomaa hyvin, että siitä on poistettu kaikki mahdolliset toteutusyksityiskohdat. Jotkin yksityiskohdat, kuten esimerkiksi ”Kirjautuminen epäonnistuu”-avainsanan käyttämät avainsanat, löytyvät kirjautumiset.resource.robot- resurssitiedostosta. Lisäksi ”Virhesivu aukeaa”-avainsanan käyttämästä ”Title Should Be”-avainsanasta näkee, että Robot Frameworkin omatkaan avainsanat eivät ole niin yksityiskohtaisi, että ne aiheuttaisi ongelmia.

3.6 Seurantaviikko 6

Maanantai 12.10.2015

Tavoitteeni tänään on osallistua päiväpalaveriin, selvittää mitä minun kannattaisi tehdä WebSocketin toteutettavuuden osoituksen suhteen, ja muuttaa kahden näkymän ajat palvelimen aikaan.

Kun saavuin työpaikalle, puhuin scrummasterin kanssa WebSocket toteutettavuuden osoitus -tehtävän antamisesti jollekin joka tietää enemmän back-end-puolesta, koska en ollut saanut osoituksen palvelinosaa toimimaan, ja näin ollen tehtävä oli jäänyt paikalleen. Hän hyväksyi tehtävän antamisen toiselle. Keskustelun jälkeen osallistuin päiväpalaveriin.

Päiväpalaverin jälkeen aloin korjaamaan kahden näkymän palvelinaikaongelmaa. Ensimmäisen näkymän korjaaminen sujui aika lailla samalla tavalla kuin aiempien näkymien: päivitin palvelimen palauttamaan ISO 8601 -aikoja, ja päivitin front-endin lukemaan niitä. Minulla oli hiukan ongelmia, koska minun piti tehdä muutoksia projektiin, jota en voinut kääntää, koska projektin pystyy kääntämään vain Linuxilla, ja koneessani ei ole Linuxia. Tämän takia minun piti tehdä muutos niin sanotusti ”sokkona”, eli: (1) tein muutoksen (2) lisäsin sen version hallintaan (3) tarkistin testipalvelimen tilan (4) tein uuden muutoksen, jos kääntäminen tai testit epäonnistuivat (5) yritin uudestaan. Tämä oli hyvin kömpelö tapa työskennellä, mutta koska muutos projektiin oli sen verran pieni, arvioin että näin toimiminen olisi silti huomattavasti nopeampaa kuin Linuxin asentaminen.

Päivän loppua kohden eräs UX-tiimin jäsen pyysi minulta apua JavaScriptin kanssa. Hänellä oli vaikeuksia saada yksi JavaScript-kirjasto toimimaan. Selvitin jonkin aikaa mitä hän yritti tehdä, ja miksi se ei onnistunut. Lopulta huomasin, ettei hän ollut ”kytkenyt” kirjastoa. Usein JavaScript-kirjastoja käyttäessä, tulee ensin lisätä HTML-koodiin elementti johon kirjaston käyttö on tarkoitus kohdistaa, sitten itse kirjasto lisätään sivulle, ja viimeiseksi kirjastoa käsketään tekemään jokin operaatio elementille. Hän oli unohtanut tehdä tämän viimeisen osan, mikä aiheutti sen että mitään ei tapahtunut, eikä edes virheviestiä näkynyt. Selitin hänelle tämän ongelman ja korjasimme koodin, ja hän jatkoi työtään.

Tiistai 13.10.2015

Tavoitteeni tänään on osallistua päiväpalaveriin ja jatkaa kahden näkymän aikojen muuttamista palvelimen aikaan.

Päivitin muutaman komennonsarjani, koska ne käyttivät vanhoja versionumeroita. Osallistuin työhyvinvointikyselyyn, joka koostui viidestä kysymyksestä, joihin pystyi vastamaan: ”samaa mieltä”, ”jokseenkin samaa mieltä”, ”ei osaa sanoa”, ”jokseenkin eri mieltä”, tai ”eri mieltä”. Osallistuin päiväpalaveriin.

Aloin korjaamaan toista näkymää, jonka ajat eivät olleet palvelimen ajassa. Tämä kuitenkin oli ongelmallista, koska näkymä käytti aikoja monimutkaisessa taulukkokomponentissa. Muutaman tunnin ajan yritin selvittää miten taulukkokomponentin voisi muuttaa käyttämään ISO 8601 -aikoja, UNIX-aikojen sijaan, mutta en löytänyt tapaa, koska komponentti käytti minulle tuntematonta kirjastoa, joka oli liian monimutkainen minulle. Koska ajattelin, että joku toinen kehittäjä olisi parempi päivittämään tämän toisen näkymän, jaoin palvelinaikatehtävän kahteen – yksi kummallekin näkymälle. Aloitin koodikatselmoinnin ensimmäisen näkymän korjaukselle.

Keskiviikko 14.10.2015

Käytin tämän päivän päiväkirjamerkintöjen kirjoittamiseen kotona, koska olin niissä jäljessä.

Torstai 15.10.2015

Tavoitteeni tänään on osallistua päiväpalaveriin, osallistua demon suunnitteluun, seurata koodini katselmointeja, ja kirjoittaa kirjoittamatta jääneitä päiväkirjamerkintöjä.

Osallistuin päiväpalaveriin. Osallistuin demon suunnitteluun, jossa päätettiin siirtää esitettävät asiat ensi sprintin demoon, koska niitä oli vain muutama ja niissä ei ollut paljoa näytettävää.

Sain suljettua torstaina 1.10. poistamani koodiin liittyvän koodikatselmoinnin. Muut katselmoijat eivät löytäneet muutoksistani mitään kommentoitavaa.

Perjantai 16.10.2015

Käytin tämän päivän päiväkirjamerkintöjen kirjoittamiseen kotona, koska olin niissä jäljessä, ja koska demo ei poikkeuksellisesti pidetty.

Viikkoanalyysi

Tässä viikkoanalyysissä analysoin Lessin käyttöä tiimissämme. Less on ohjelmointikieli, joka toimii esikääntäjänä CSS-kielelle. Less itse pohjautuu CSS-kieleen: kaikki validi CSS-koodi on validia Less-koodia. Less näin ollen lisää monia toimintoja CSS:n päälle, jotka tekevät siitä dynaamisemman, esimerkiksi: muuttujat, valitsijoiden sisäkkäisyys, mixinit, ja funktiot. Koska Less on esikääntäjä, selaimelle ei lähetetä Less-tiedostoja, vaan sille lähetetään käännetty CSS-tiedosto.

Esimerkki Less koodista, jossa hyödynnetään muuttujaa:

```
@urgent-color: #ff9000;
.error-message {
  color: @urgent-color;
}
.urgent-message {
  color: @urgent-color;
}
```

Käännettäessä Less-koodin `@urgent-color`-muuttujan arvo sijoitetaan itse muuttujan paikalle, jolloin CSS-koodiksi tulee:

```
.error-message {
  color: #ff9000;
}
.urgent-message {
  color: #ff9000;
}
```

Esimerkki Less koodista, jossa hyödynnetään valitsijoiden sisäkkäisyyttä:

```
.site-header {
  .error-message { ... }
  .urgent-message { ... }
}
```

Käännettäessä Less-koodin `.site-header`-luokkavalitsija sijoitetaan kahden muun luokkavalitsijan eteen, jolloin CSS-koodiksi tulee:

```
.site-header .error-message { ... }
.site-header .urgent-message { ... }
```

Less on hyvin hyödyllinen työkalu, koska se poistaa toistoa, ja näin ollen nopeuttaa koodin kirjoittamista, ja helpottaa luettavuutta. Less-muuttujat ovat erityisen hyödyllisiä, koska ne eivät pelkästään vähennä toistoa, vaan ne myös mahdollistavat tarkoituksen lisäämistä koodiin. Edellistä koodiesimerkkiä käyttäen, `#ff9000`-värikoodi kertoo kokeneelle verkko-ohjelmoijalle, että tekstin väri on kirkkaanoranssi, mutta se ei itsessään kerro mitä värillä yritetään viestittää. Värille `@urgent-color`-muuttujanimen antaminen helpottaa viestimään värin tarkoituksen koodissa, ja näin tekee koodista helpomman ymmärtää ja myös jatkokehittää.

Lessin käytössä on kuitenkin haittapuoliakin. Valitsijoiden sisäkkäisyys on hyvin hyödyllinen toiminto, koska se helpottaa monien samankaltaisten valitsijoiden kirjoittamisessa. Tämä kuitenkin usein johtaa siihen, että koodista tulee sekavaa, koska koodiin lisätään hyvin spesifisiä sääntöjä. Jos tyylikoodista tulee hyvin pikkutarkka, siihen muutosten tekeminen tulee vaikeammaksi, koska koodia pitää muuttaa monessa paikassa. Yrityksessämme esimerkkinä tästä pikkutarkkuudesta on käyttöliittymän käänteinen teema. Yrityksen käyttöliittymä on tavallisesti valkotaustainen, mustalla tekstillä. Jotkin tuotteet kuitenkin halusivat käänteisen teeman, jossa käyttöliittymä on tummansinitaustainen, valkoisella tekstillä. Tämä kaksiteemaisuus aiheutti sen, että tyylikoodia oli kaksin kerroin tavalliseen verrattuna. Kun käänteistä teemaa alettiin poistamaan, syntyi ongelmia, koska joissakin näkymissä oli oletettu, että niissä aina käytetään käänteistä teemaa, ja näin ollen niiden tyylit oli justeerattu sitä varten. Tällaiseen tilanteeseen ei todennäköisesti olisi jouduttu, jos olisimme käyttäneet pelkkiä CSS-tyylejä, koska useamman kuin yhden teeman tekeminen pelkillä CSS-tyyleillä olisi todennäköisesti todettu liian hankalaksi.

3.7 Seurantaviikko 7

Maanantai 19.10.2015

Tavoitteeni tänään on osallistua kahden päivän mittaiseen ”Program Increment”-jakson suunnitteluun. Olen osallistunut aikaisemmin jo yhteen tällaiseen suunnitteluun. Suunnitteluun osallistuu kaikki yhteen tuoteperheeseen liittyvät tiimit ja henkilöt. Tiimejä on noin tusina. Suunnittelupäivinä ei pidetä päiväpalavereja.

Suunnittelu alkaa kahden tunnin tilaisuudella, johon kaikkien tiimien jäsenten on tarkoitus osallistua. Tilaisuus pidetään suuressa neuvottelusalissa, jossa on noin 50 henkilöä. Lisäksi tilaisuuteen osallistuu kymmeniä kehittäjiä Skypen välityksellä Oslosta, Kuala Lumpurista, Britanniasta, ja Yhdysvalloista. Tilaisuudessa käydään läpi yrityksen nykytilanne, tuoteperheen visio, tuoteperheen arkkitehtuurin visio, ja suunnittelun konteksti.

Tilaisuuden jälkeen scrummasterit osallistuvat ensimmäiseen scrumien scrum -kokoukseen (englanniksi ”scrum of scrums”), jossa he kertovat toisilleen tiimiensä tilanteen alustavasti, samalla tavalla kuin scrumin päiväpalavereissa.

Kun scrummasterit palaavat, tiimit kerääntyvät eri kokoushuoneisiin, ja suunnittelevat jaksoa. Erityisen tärkeää suunnittelussa on tarvittavien ja saatavilla olevan työmäärän määritteleminen. Tarvittava työmäärä määritellään tuotteen kehitysjonon työstöllä, jossa jokaiselle tarvittavalle tuotteen toiminnolle määritellään käyttäjätarinapisteet (englanniksi ”story points”). Yksi piste vastaa yrityksemme scrumkäytännöissä yhden työpäivän työtä. Yhdessä sprintissä on täyspäiväisellä työntekijällä kahdeksan pistettä, koska sprintit ovat kahden viikon pituisia, ja jokaisessa viikossa on yleensä viisi työpäivää, ja yleensä sprintistä kaksi työpäivää kuluu sprintin suunnitteluun ja sprintin päättämiseen. Työmäärän saatavuus lasketaan työntekijöiden ennustettujen työ- ja lomapäivien mukaan, ja odotettavan työpanoksen mukaan. Esimerkiksi jos työntekijä on osa-aikainen, harjoittelija, tai tekee opinnäytetyötä, työntekijälle lasketaan vähemmän pisteitä. Kun tarvittavat ja saatavat työmäärät on laskettu, tiimit suunnittelevat alustavasti sprintit seuraavaan jaksoon asti. Vertaamalla sprinttien tarvittavien ja saatavien pisteiden määrää toisiinsa pystytään arvioimaan onko tiimillä liian vähän tai liian paljon kehittäjiä.

Tiimien yksittäisten suunnittelujen jälkeen scrummasterit osallistuvat toiseen scrumien scrum -tilaisuuteen, jossa he kertovat toisilleen tiimiensä tilanteen ja suunnitelmat. Tämä jälkeen tuoteomistajat ja arkkitehdit esittävät tiimien suunnitelmat alustavasti ylemmälle johdolle.

Tulin suunnittelutilaisuuteen hiukan myöhässä, koska olin olettanut sen olevan samassa kokoushuoneessa kuin edellinen, ja minulla kesti hiukan löytää oikea paikka. Suunnittelutilaisuus ja tuotteen kehitysjonon työstö onnistuivat ongelmitta, vaikka olin aika väsynyt aikaisen herätyksen takia. Saimme työstettyä noin puolet tuotteen tulevista toiminnoista.

Tiistai 20.10.2015

Tavoitteeni tänään on osallistua ”Program Increment”-jakson suunnittelun toiselle päivälle. Toinen päivä aloitetaan jatkamalla tuotteen kehitysjonon työstöä. Lisäksi tiimit tekevät riippuvuus- ja riskikartoitukset.

Riippuvuuskartoituksessa tiimi miettii, minkä muiden tiimien toiminnat, ja muut tapahtumat liittyvät heidän työhönsä. Lisäksi halutaan tietää, onko mahdollisia tilanteita, joissa tiimin jäisi jumiin, koska jokin toinen tiimi ei ole tehnyt joitain heille kriittistä asiaa.

Riskikartoituksessa tiimi miettii mahdollisia työtä haittaavia tai estäviä riskejä, ja jaottelee ne neljään ryhmään ROAM-menetelmän mukaan. *All About Agile* -sivuston kirjoittajan Kelly Watersin (2012) mukaan ROAM tulee sanoista: ”Resolved”, ”Owned”, ”Accepted”, ja ”Mitigated”:

- **”Resolved”**, eli ”Ratkaistu”, tarkoittaa riskiä, joka ei ole enää ongelma, koska se on vältetty tai siihen on muuten vastattu.
- **”Owned”**, eli ”Omistettu”, tarkoittaa riskiä, joka on annettu jonkun henkilön tai tahon vastuulle.
- **”Accepted”**, eli ”Hyväksytty”, tarkoittaa riskiä, josta on päätetty, ettei sille tehdä mitään.
- **”Mitigated”**, eli ”Lievennetty”, tarkoittaa riskiä, jonka todennäköisyyttä ja/tai vaikutusta on pienennetty.

Riippuvuus- ja riskikartoitusten jälkeen tiimit kerääntyvät taas samaan kokoustilaan, ja esittävät toisilleen suunnitelmansa ja kartoituksensa jaksoon. Muut tiimit ja johto kysyy kysymyksiä ja kommentoi niitä.

Kun kaikki tiimit ovat esittäneet, pidetään jakson arviointiäänestys. Äänestyksessä paikallaolioilta kysytään, kuinka hyvin he odottavat jakson onnistuvan asteikolla 1–5, ja he vastaavat viittaamalla. Jokaiselle arvolle annetut äänet lasketaan ja kerätään myöhempää yhteen liittämistä ja analyysia varten.

Äänestyksen jälkeen pidetään menneen jakson retrospektiivi. Skype-tapaaminen suljetaan tässä vaiheessa, koska retrospektiivi käydään vain paikallisten tiimien välillä. Retrospektiivissä osallistujat jaetaan neljään ryhmään ja he kirjoittavat paperille asioita, jotka aiheuttivat eniten ongelmia jaksossa. Sitten paperit kerätään eteen, ja ongelmat ja niihin liittyvät parannusehdotukset käydään läpi. Lopuksi jokainen osallistuja saa äänestää kahta ongelmaa, joita he eniten haluaisivat korjattavan.

Suunnitteluun osallistuminen onnistui suuremmitta ongelmitta, enkä ollut yhtä ymmällään kuin viime kerralla. En vieläkään oikein osannut antaa mitään kritiikkiä retrospektiivissä, mutta oletan että minulla on ensi kerralla jotain huomautettavaa.

Keskiviikko 21.10.2015

Tavoitteeni tänään on osallistua päiväpalaveriin, kirjoittaa kirjoittamatta jääneitä päiväkirjamerkintöjä, ja alustavasti valmistautua opinnäytetyöseminaariin.

Päiväpalaveria ei pidetty, koska scrummaster oli toisessa palaverissa sille tarkoitettuna aikana, ja kun hän tuli takaisin osa tiimistä oli mennyt lounastauolle. Scrummaster sai tietää palaverissaan, että lokalisaatio onkin tärkeämpi tuotteen lähitulevaisuudelle kuin aiemmin odotettiin. Tämän takia hän pyysi yhtä kehitystiimin jäsentä tekemään toteutettavuuden osoituksen lokalisaatiosta. Puhuin scrummasterin ja toisen kehittäjän kanssa lokalisaatiosta, koska minulle oli noin kuukausi aiemmin annettu tehtäväksi tutkia kansainvälistämisestä ja lokalisaatiosta. Heidän ei tarvinnut kysyä minulta paljoa, koska oli dokumentoinut tutkimukseni hyvin.

Torstai 22.10.2015

Yritin kirjoittaa päiväkirjamerkintöjä kotona, mutta en saanut oikein mitään kirjoitettua, koska en pystynyt keskittymään kirjoittamiseen.

Perjantai 23.10.2015

Tavoitteeni tänään on osallistua päiväpalaveriin, ja kirjoittaa päiväkirjamerkintöjä.

Osallistuin päiväpalaveriin ja kirjoitin päiväkirjamerkintöjä.

Viikkoanalyysi

Käyttöliittymäalustamme käyttää tällä hetkellä pohjanaan Backbone-viitekehystä, jQuery-kirjastoa, ja Handlebars.js-kaavainprosessoria. Näistä työkaluista on kuitenkin suunniteltu luovuttavan, ja siirryttävän käyttämään Facebookin luomia React-kirjastoa ja JSX-ohjelmointikieltä. Tässä viikkoanalyysissä vertaan nykyisiä pääkirjastoja tuleviin kirjastoihin ja analysoin muutosta.

Luin tästä muutoksesta analyysin, jonka oli kirjoittanut erään tuotetiimin front-end-johtaja. Analyysissä hän listasi muutaman syyn olla siirtymättä Reactiin:

- Se on iso muutos arkkitehtuuriin.
- Kehittäjät pitää kouluttaa käyttämään uusia työkaluja.
- Muutossa on todennäköisesti myös tuntemattomia riskejä.

Hän kuitenkin löysi paljon syitä siirtymisen puolesta, lyhennetysti:

- Nykyisessä tavassa on vaikea löytää, mikä näkymä luo minkäkin HTML-koodinpätkän; tämä on helpompaa Reactilla, koska esimerkiksi Google Chromella on liitännäinen, jolla pystyy näkemään React-komponentin nimen HTML-koodista.
- React-komponenttien uudelleenkäyttö on paljon helpompaa kuin Handlebars-kaavojen käyttö. Erityisesti Reactilla on helpompaa käyttää sisäkkäisiä näkymiä.
- Nykyisessä tavassa HTML-luokkia käytetään JavaScript-, ja CSS-koodissa. Näin ollen on vaikea tietää, kumpaan (vai kumpaankin) kyseistä luokkaa käytetään. Reactissa ei tarvitse viitata HTML-luokkiin.
- Backbone-näkymissä pitää kirjoittaa ”siivous”-koodia, jossa oliot, joita ei enää käytetä, piti kytkeä pois käytössä olevista komponenteista, jotta ne eivät jää turhaan muistiin. React hoitaa ”siivouksen” automaattisesti.
- Nykyisessä tavassa, jos esimerkiksi siirtää tekstikenttää lennossa, sen sisältö pitää manuaalisesti kopioida JavaScript-muuttujaan, ja sitten siirtää takaisin tekstikenttäelementtiin, kun siirto on valmis, koska tekstikentän tila katoaa siirrossa. React huolehtii automaattisesti tilan siirtämisestä.
- Koodi on Reactilla ”matemaattisempaa”.

Mielestäni hän kuvaa hyvin osuvasti nykyisen arkkitehtuurin ongelmia. Minullakin on ollut usein vaikeuksia löytää, mihin komponenttiin jokin HTML-koodi kuuluu, ja olen jsokus joutunut käyttämään tunninkin etsimiseen. Minun ei ole ollut ongelmia sisäkkäisten komponenttien käytössä, mutta on mahdollista, että se on helpompaa Reactilla. Minulla on ollut usein vaikeuksia varmistaa käytetäänkö HTML-luokka JavaScript-koodissa, vai CSS-koo-

dissa, vai kummassakin. Tämä on ollut erityisen hankalaa, kun olen miettinyt jonkin luokan poistamista tai uudelleennimeämistä. Voin sanoa kokemuksesta, että ”siivous”-koodimme on hyvin sekavaa, ja siihen muutosten tekeminen hyvin vaikeaa ja aikaa vievää.

Olen törmännyt joihinkin ongelmiin tilallisten elementtien siirtämisessä, mutta en yhtä suuriin kuin hän. Hänen esimerkki oli käyttämämme taulukkokomponentti, jonka elementtien aakkostamisessa joutui siirtämään satoja tilallisia elementtejä. Tällaisessa tilanteessa tilan säilymisen varmistaminen on hyvin hankalaa, ja virheiden tekeminen on hyvin todennäköistä.

Kysyin kehittäjältä mitä hän tarkoitti Reactin ”matemaattisuudella”. Lyhyen keskustelun jälkeen tajusin että hän tarkoitti ”deklaratiivisempaa”, muttei vain muistanut tätä sanaa. Deklaratiivisessa ohjelmoinnissa tietokoneelle kerrotaan *mitä* sen tulee saada aikaiseksi, muttei *miten*. SQL-tietokantakyselyt ovat klassinen esimerkki deklaratiivisesta ohjelmoinnista. Kyselyissä tietokoneelle esimerkiksi kerrotaan *mitä* tietoja halutaan sen hakevan tietokannasta, mutta ei *miten* sen tulee hakea ne. Tässä on kaksi hyvää puolta: (1) Tietokannan käyttäjän ei tarvitse miettiä minkälaisia esimerkiksi kierteitä haku tarvitse. (2) Deklaratiivinen kysely antaa tietokannan tekijälle vapaat kädet päättää kyselyn toteutuksesta, kunhan se vain palauttaa oikean tuloksen; tällöin tietokannan tekijä voi optimoida kyselyiden toimintaa ilman että käyttäjän tarvitsee tehdä mitään näiden optimointien hyödyntämiseen.

Deklaratiivisen ohjelmoinnin vastakohta on imperatiivinen ohjelmointi, jossa tietokoneelle kerrotaan *miten* sen tulee päästä tulokseen. Imperatiivisessa ohjelmoinnissa tietokoneelle annetaan tarkat askelittaiset komentosarjat. Imperatiivinen ohjelmointi on usein yleisempää mitä alempitasoinen ohjelmointikieli on. Lisäksi ohjelmointi on imperatiivisessa yleensä tilallisempaa. Funktionaalinen ohjelmointi usein katsotaan liittyvän vahvasti deklaratiiviseen ohjelmointiin. Viime vuosina funktionaalisen ohjelmoinnin (ja näin ollen deklaratiivisen ohjelmoinnin) suosio on kasvanut. Tämä johtuu aiemmasta funktionaalisen ja deklaratiivisen koodin verrattaisesta hitaudesta ja suuresta resurssien käytöstä. JavaScriptissä on kumpiakin deklaratiivisia ja imperatiivisia piirteitä, ja se on vahvasti funktionaalinen kieli.

3.8 Seurantaviikko 8

Maanantai 26.10.2015

Tavoitteeni tänään on osallistua päiväpalaveriin, ja kirjoittaa päiväkirjamerkintöjä.

Osallistuin päiväpalaveriin ja kirjoitin päiväkirjamerkintöjä.

Tiistai 27.10.2015

Tavoitteeni tänään on osallistua päiväpalaveriin, tuotteen kehitysjonon työstöön, ja kirjoittaa päiväkirjamerkintöjä.

Osallistuin tuotteen kehitysjonon työstöön ja kirjoitin päiväkirjamerkintöjä. Päiväpalaveria ei pidetty, koska se aiottiin pitää työstön jälkeen, mutta se unohtui.

Työstössä käytiin läpi toiminto, jolla tuotteessa pystyisi laittamaan ”muistilappuja” esimerkiksi joihinkin listojen riveihin. ”Muistilaput” toimisivat kommentteina, joilla voisi vaikka varoittaa muita käyttäjiä tekemästä muutoksia riviin, tai selittämään miksi jotkin erikoiset arvot on annettu riville. Muut käyttäjät voisivat myös lisätä kommentteja ”muistilappuihin”. Toiminto päätettiin jakaa kolmeen osaan: käyttöliittymäkomponenttiin itseensä, REST-rajapinnan määritykseen, ja palvelinpuolen ohjelmoimiseen. Osille annettiin omat arvionsa.

Keskiviikko 28.10.2015

Tavoitteeni tänään on osallistua päiväpalaveriin, ja kirjoittaa päiväkirjamerkintöjä.

Osallistuin päiväpalaveriin ja kirjoitin päiväkirjamerkintöjä.

Torstai 29.10.2015

Tavoitteeni tänään on osallistua päiväpalaveriin, demon suunnitteluun, ja kirjoittaa päiväkirjamerkintöjä.

Osallistuin päiväpalaveriin, demon suunnitteluun ja kirjoitin päiväkirjamerkintöjä. Lisäksi sain suljettua tiistaina 13.10. avaamani palvelinaikaan liittyvän koodikatselmoinnin. Ainoa kommentti katselmoinnista oli, että scrummaster halusi lisätä yhden palvelinpuolen testin, joka tarkistaa että ISO 8601 -aikojen muuntaminen toimii oikein.

Perjantai 30.10.2015

Tavoitteeni tänään on osallistua sprintin demoon, päiväpalaveriin, ja valmistella opponointi.

Tulin demoon hiukan myöhässä, koska muistin sen ajan väärin. Demossa käytiin läpi aikakorjaus, jonka sain valmiiksi tiistaina 13.10. Korjaus mainittiin vain ohimennen, koska olin jo tehnyt samanlaisen korjauksen pariin muuhun näkymään. Osallistuin päiväpalaveriin.

Koska henkilö, jonka päiväkirjaopinnäytetyö minun piti opponoida, ei ollut vielä palauttanut työtään, minä en voinut vielä valmistella opponointia. Tämän takia päätin käyttää päivän koulun kannettavani päivittämiseen, koska todennäköisesti tarvitsen kannettavaa seminaarissa, ja kannettavaan tarvittavat päivitykset pystyi tekemään vain koulun verkossa.

Viikkoanalyysi

Viikko meni pääosin opinnäytetyöseminaariin valmistautuessa. Lisäksi minulla oli vaikeuksia keskittyä kirjoittamiseen, osaksi koska en pidä kirjoittamisesta, osaksi koska työpaikkani ja kotini ovat aika mölyisiä, ja osaksi koska minulla oli työhön ja opiskeluun liittymättömät asiat mielessä. Olin aikaisemmin kuullut, että yleisesti työn teko, mutta kirjoittaminen eteenkin, sujuu parhaiten kun tekee sen toimistolla, ja aloittaa työn mahdollisimman nopeasti, välttäen mahdolliset häiriötekijät. Tämän faktan muistaminen ja hyödyntäminen auttoi minua kirjoittamaan monia merkintöjä, jotka olin unohtanut.

3.9 Seurantaviikko 9

Maanantai 2.11.2015

Valmistauduin opinnäytetyöseminaariin ja opponointiin.

Tiistai 3.11.2015

Valmistauduin ja osallistuin opinnäytetyöseminaariin ja opponointiin.

Keskiviikko 4.11.2015

Olin sairaana.

Torstai 5.11.2015

Olin sairaana.

Perjantai 6.11.2015

Olin sairaana.

Viikkoanalyysi

Tässä viikkoanalyysissä analysoin Markdown-merkintäkieltä ja -tiedostoformaattia. Markdown on John Gruberin vuonna 2004 luoma vapaa (ja avoin) formaatti, jota aloitettiin standardoida vuonna 2014, Jeff Atwoodin, John MacFarlanen, ja muiden (2016) *CommonMark*-projektilla. Olen jo aikaisemmin käyttänyt henkilökohtaisesti Markdownia omissa dokumenteissani, mutta viime aikoina tiimimme on miettinyt siirtymistä sen käyttöön dokumentoinnissa – wiki-formaatin sijasta.

Markdown on niin sanottu ”ihmisläheinen” merkintäkieli, mikä tarkoittaa sen merkintätavan näyttävän visuaalisesti luonnolliselta:

```
Paisto-ohjeet
=============

1. Poista pizza pakkauksesta.
   - Voit myös vain leikata paketin reunan auki.
2. Laita mikroaaltouuniin kahdeksi minuutiksi täydellä teholla.
3. Syö!
```

Edellinen koodinpätkä alkaa ”Paisto-ohjeet”-pääotsikolla, joka on merkattu alleviivaamalla otsikon teksti yhtäsuuruusmerkeillä. Sen jälkeen, rivin alkava `1.` aloittaa numerolistan, jossa on kolme kohtaa. Listan ensimmäisen kohdan toisella rivillä oleva sisennetty ranskalainen viiva (`-`) aloittaa listakohdan sisällä uuden, numerottoman alilistan, jolla on vain yksi kohta. Markdown-tiedostot yleisimmin muunnetaan HTML-tiedostoiksi. Edellinen koodi muuntuu koodiksi:

```
<h1>Paisto-ohjeet</h1>
<ol>
  <li>Poista pizza pakkauksesta.
    <ul>
      <li>Voit myös vain leikata paketin reunan auki.</li>
    </ul>
  </li>
  <li>Laita mikroaaltouuniin kahdeksi minuutiksi täydellä teholla.</li>
  <li>Syö!</li>
</ol>
```

Opiskelujeni alkupäässä tein usein sähköiset muistiinpanoni ja tehtäväni Microsoft Word -dokumentteihin. Minua kuitenkin usein ärsyttivät ongelmat formatoinnin kanssa. Välillä dokumenteissa oli ongelmallista, näkymätöntä formatointia. Välillä dokumenttien formatointi ei toiminut oikein, kun tiedoston avasi Wordin toisella versiolla, tai kokonaan toisella ohjelmalla, kuten LibreOfficella. Luulin löytäneeni hyvän ratkaisun käyttämällä Google Docs -tiedostoja, mutta niissä taas on se ongelma, että ne pitävät tiedoston pelkästään Googlen pilvessä, ja näin ollen näitä tiedostoja ei voi siirtää kiintolevylle, ilman muuntamista toiseen tiedostoformaattiin.

Koska nämä dokumenttiformaatit eivät kelvanneet tarkoituksiini, aloin pitämään tekstiäni tavallisissa perustekstitiedostoissa (englanniksi ”plain text files”). Minua kuitenkin häiritsi, ettei minulla oikein ollut mitään hyvää tapaa jäsennellä tekstiä, ja tiesin että perustekstitiedostojen muuntaminen formatoituihin tekstitiedostoihin on työlästä. Tähän ratkaisuna kuitenkin löysin Markdownin, ja sen *CommonMark*-standardin.

Markdown-tiedostojen kirjoittaminen on hyvin helppoa verrattuna muihin merkintäkieliin (kuten HTML-kieleen), ja tiedostot muuntuvat moniksi muiksi tiedostoformaateiksi. Kun Markdown-syntaksin osaa, ainoat ”formatointi”-asiat, joista tarvitsee huolehtia, ovat tiedoston merkistön pitäminen kansainvälisessä UTF-8-muodossa, ja rivien lopetus/erotusmerkkien pitäminen UNIX-muodossa. Nämä asiat ovat helppo ja nopea konfiguroida melkein missä tahansa *modernissa* tekstieditorissa (mutta ei esimerkiksi Microsoftin Notepad-editorissa).

Perustekstitiedostoja voi myös editoida pilvipalveluissa, samalla tavalla kuin Microsoftin ja Googlen tiedostoformaatteja – mutta tämä toiminto löytyy kaikista tuntemistani pilvipalveluista – kun taas yhtiöiden *omia formaatteja* voi editoida vain niiden omissa palveluissaan. Koska tiedostoja ei voi editoida toisissa pilvipalveluissa, ja koska tiedostojen muuntaminen toisiksi ei ole ongelmatonta, voidaan näiden formaattien käyttöä pitää tietynlaisena ”lukkiutumisena” (englanniksi ”vendor lock-in”). Tämä tarkoittaa, että siirtyminen johonkin toiseen pilvipalveluun tulee olemaan hyvin vaivalloista ja työlästä, jos on käyttänyt jompaakumpaa palvelua pitkään ja luonut paljon palvelun formaatin tiedostoja. Sama ”lukkiutuminen” ei kuitenkaan päde Markdowniin, koska se on vapaa standardi. (Vaikka Microsoft pitää Office-tiedostoja vapaana standardina, vapaa ohjelmisto -liike ei yleisesti ole samaa mieltä.) Markdown-tiedostot ovat hyvin tulevaisuudenkestäviä, koska ne pohjautuvat vapaaseen formaattiin, *ja* ne ovat luettavia sellaisinaan.

Markdown on monella tavalla oiva kehitystiimille. Merkintäkielet ovat yleensä liian vaikeita kirjoittaa ei-ohjelmoijille, mutta tämä ei ole ongelma Markdownissa, sen edellä mainitun

ihmisläheisyyden vuoksi. Toisin kuin muut dokumenttiformaatit, Markdown ei ole sidottu yhteen tai kahteen editoriin, vaan sitä voi editoida millä tahansa modernilla perustekstieditorilla. Tämä antaa kehittäjille vapauden käyttää lempieditoriaan. Markdownille löytyy myös editoreja, kuten wereturtlen *ghostwriter*, joka näyttävät esikatseluna tiedostosta tuotettavan HTML-sivun.

Markdown-tiedostot toimivat myös hyvin versiohallinnan kanssa. Esimerkiksi Word-dokumenttien laittaminen versiohallintaan on ongelmallista, koska ne ovat binäärimuotoisia. Tästä johtuen versionhallinta joko pitää tiedostosta vain uusimman version, *tai* se joutuu säilyttämään tiedoston jokaisen version itsenäisenä tiedostona, mikä vie paljon tilaa. Esimerkiksi Word-dokumenteilla *on* oma muutostenseurantamekanismi, mutta se ei toimi versiohallintaohjelmien kanssa, eikä se ole yhtä monipuolinen kuin esimerkiksi Mercurial tai Git.

Markdown-tiedostoissa on myös muitakin hyötyjä. Ne ovat turvallisempia kuin esimerkiksi Word-dokumentit, koska niissä ei voi olla vaarallisia fontteja tai makroja. Markdown-tiedostoissa on *teoriassa* kolme riskiä:

1. **Perustekstitiedostojen haavoittuvuudet**: Näiden hyväksikäyttö käytännön hyökkäyksissä on harvinaista, koska perustekstieditoreja on tuhansia, ja niiden haavoittuvuudet ovat harvinaisia.
2. **Markdown-kääntäjän haavoittuvuudet**: Näiden hyväksikäyttö käytännön hyökkäyksissä on ennenkuulumatonta, todennäköisimmin koska eri Markdown-kääntäjiä on kymmeniä, ja koska Markdown-kääntäjät ovat verrattain yksinkertaisia.
3. **HTML-tiedostojen haavoittuvuudet**: HTML-hyökkäyksen tekeminen Markdown-tiedostolla on hölmöä, koska käyttäjä pystyy näkemään vaarallisen koodin editoidessaan, ja koska on *paljon* helpompaa saada uhri lataamaan HTML-tiedosto, lähettää uhrille HTML-sähköposti, tai tehdä hyökkäys verkkosivulla.

Markdown-tiedostot ovat myös verrattain pieniä, eteenkin silloin, kun dokumentissa olevien kuvien halutaan myös löytyvän samasta kansiosta. Tällöin esimerkiksi Word-dokumentissa kuva on kummassakin – kansiossa, ja tiedostossa itsessään. Markdownin kanssa näin ei käy, koska kuviin voidaan vain viitata, milloin ne eivät ole tiedostossa itsessään sisällä, ja näin ollen ne vievät tilan vain kerran.

3.10 Seurantaviikko 10

Maanantai 9.11.2015

Tavoitteeni tänään on osallistua päiväpalaveriin, ja saada jokin tehtävä jota voisin tehdä tämän viikon.

Luin viime viikon sähköpostit, ja asensin päivityksiä. Osallistuin päiväpalaveriin. Palaverin jälkeen pyysin scrummasterilta jotain tehtävää. Scrummaster antoi tehtäväkseni tehdä uuteen tuotteittemme käyttöliittymään kuuluvan ”tooltip”-, eli ohjetekstikomponentin.

Tuotteessamme on jo ohjetekstikomponentti, mutta sen tyyliä halutaan muuttaa ja siitä halutaan tehdä monipuolisempi. Nykyisen käyttöliittymän ohjetekstit ovat hyvin puhekuplamaisia: ne ovat valkoisia, mustalla tekstillä, harmailla rajalla, ja pyöristetyillä kulmilla. Lisäksi niillä on vahvat varjot. Nykyiset ohjetekstit osoittavat kohteeseensa niiden alalaidassa olevalla 90 asteen kolmiolla. Osoitin on aina alalaidan keskellä.

Uuden käyttöliittymän ohjetekstikomponentti käyttää päinvastaisia värejä: teksti on valkoista tummanharmaalla taustalla. Uusilla ohjeteksteillä ei ole erivärisiä rajoja ja niiden kulmat ovat pyöristetty niin hienojakoisesti, että ohjetekstit ovat melkein täysin nelikulmioita. Uusilla ohjeteksteillä ei ole varjoja. Uudet ohjetekstit myös käyttävät samanlaista kolmiota osoittajana, mutta toisin kuin vanhassa käyttöliittymässä, kolmio voi olla ohjetekstin ala- tai ylälaidassa, ja se voi olla laidan keskellä tai reunoilla. Tämä mahdollistaa ohjetekstin sijoittamisen niin, että se on vähemmän muiden elementtien tiellä. Esimerkiksi, taulukon ylätunnisteen ohjetekstit kannattaa sijoittaa olemaan yläpuolella, kun taas taulukon pohjalla olevien elementtien ohjetekstit kannattaa sijoittaa niiden alle. Osoittimen horisontaalinen sijoittelu taas mahdollistaa sen, että ohjeteksti voidaan pitää aina ruudulla. Esimerkiksi jos ohjeteksti on sivun vasemmassa laidassa, kannattaa osoitin laittaa ohjetekstin vasempaan laitaan, jotta ohjetekstin teksti nojaa oikealle.

Uusiin ohjeteksteihin halutaan myös mahdollisuus lisätä otsikko. UX-tiimi ei kuitenkaan suosittele otsikon käyttöä, koska ohjetekstien pitäisi olla lyhyitä. Tämän takia uusiin ohjeteksteihin lisätään ominaisuus antaa otsikko, mutta ominaisuutta ei ”mainosteta”, jotta sitä käytettäisiin mahdollisimman vähän, eli vain silloin kun ei ole parempia vaihtoehtoja. Uusien toimintojen mahdollistamiseksi, myös ohjetekstien käyttämä kirjasto pitää vaihtaa, koska nykyisessä kirjastossa ei ole kaikkia tarvittavia toimintoja.

Koska uuden ohjetekstikomponentilla ei ollut vielä tehtäväkuvausta päätin tehdä sen. Minulle oli eteenkin tärkeä saada jonkinlainen valmiinmääritelmän tehtyä. Määritelmään kuului kaikki yleiset uuden komponentin vaatimukset:

- komponentin itsensä ohjelmoiminen,
- koodin dokumentaatiokommenttien kirjoittaminen,
- wiki-dokumentaation kirjoittaminen,
- yksikkötestien tekeminen, ja
- komponentin tyylien ja toiminnan tarkastuttaminen UX-tiimillä.

Lisäksi, koska komponentti käyttää ulkoista kirjastoa, kirjastosta tulevat ominaisuudet kannattaa abstraktoida pois, koska silloin komponentin koodin käyttöliittymä ei ole sidonnainen kirjastoon, ja näin ollen kirjaston voi vaihtaa muuttamatta komponentin koodin käyttöliittymää. (Koodi myös tulee katselmoida, mutta tätä ei kirjoiteta valmiinmääritelmään, koska katselmointi tehdään kaikkiin muutoksiin, jotka ovat paria riviä suurempia.)

Kun keräsin tietoa valmiinmääritelmää varten, sain tietää, että toinen tiimin jäsen oli jo tehnyt komponenttia alustavasti: hän oli jo lisännyt tarvittavan kirjaston projektiimme, ja tehnyt lyhyen esimerkin esimerkkimoduuliin.

Tiistai 10.11.2015

Tavoitteeni tänään on osallistua päiväpalaveriin, ja aloittaa uuden käyttöliittymän ohjetekstikomponentin tekeminen.

Aina ennen kuin aloitan tekemään muutoksia, tarkistan että koodi kääntyy, ja että yksikkötestit menevät läpi. Teen tietysti saman tarkistuksen ennen kuin talletan koodin. Teoriassa testaamisen voisi tehdä vain ennen tallettamista, mutta itse ajan testit myös ennen muutoksien tekemistä, koska silloin voin olla varma, että jos testit epäonnistuvat tallettaessa, niin vika on omissa muutoksissa. Jos testejä ei aja aloittaessa, niiden epäonnistuminen tallettaessa saattaa johtua jonkun muun koodista, mutta saattaa olettaa epäonnistumisen olevan omassa koodissa, ja näin ollen saattaa kulua aikaa, kun etsii vikaa omasta koodista, vaikka vika onkin toisen muutoksissa.

Tämän esitestin tuloksena huomasin, että uudenkäyttöliittymän versiohallintahaara ei läpäissyt testejä. Mielenkiintoisinta ongelmassa oli, että testaus antoi virheviestin *ennen kuin* se ehti itse testitapauksiin. Ongelma vaikutti olevan, että testit eivät pystyneet löytämään yhtä toista JavaScript-kirjastoa. Yritin selvittää tätä ongelmaa ja milloin se oli syntynyt toisen kehittäjän ja scrummasterin kanssa. Yritin myös ajoitellen selvittää ongelmaa

yksin. Huomasin, että vaikka keskusmoduulin testit menivät läpi ongelmitta, ongelma vaikutti useampaan käyttöliittymämoduuliin. Näin ongelman pitäisi olla keskusmoduulissa.

Keskiviikko 11.11.2015

Tavoitteeni tänään on osallistua päiväpalaveriin, selvittää testiongelma, ja toivottavasti aloittaa ohjetekstikomponentin työstäminen.

Kerroin scrummasterin kanssa testiongelmasta tuoteomistajalle. Jatkoin ongelman syyn löytämistä. Kävin läpi uuden käyttöliittymän versiohallintahaarassa olevia muutoksia, ja yritin löytää muutoksen, joka rikkoi testit. Samalla toinen ohjelmoija, joka oli työstänyt uutta käyttöliittymää eniten, yritti myös korjata ongelmaa. Tutkiessani huomasin, että ongelma oli syntynyt heti haaran alussa, jossa oli tehty hyvin paljon muutoksia. Nain ollen arvelin, että en tule löytämään ja korjaamaan tarkkaa ongelmaa, ainakaan niin että ehtisin tehdä ohjetekstikomponentin viikon loppuun mennessä. Tämän takia päätin käyttää lopun viikosta opinnäytetyön kirjoittamiseen.

Torstai 12.11.2015

Tavoitteeni tänään on osallistua päiväpalaveriin ja demon suunnitteluun, ja kirjoittaa päiväkirjamerkintöjä.

Tein päiväkirjamerkintöjä. Osallistuin päiväpalaveriin. Tiimin jäsen, joka yleensä liittyy päiväpalaveriin Skypen välityksellä, oli poikkeuksellisesti paikalla. Scrummaster silti yritti soittaa hänelle ennen kuin tajusi hänen olevan paikalla. Scrummaster päätti olla pitämättä demon suunnittelua, koska hän tiesi jo, ettei meillä ollut mitään esitettävää.

Perjantai 13.11.2015

Tavoitteeni tänään on kirjoittaa päiväkirjamerkintöjä ja osallistua päiväpalaveriin.

Kirjoitin päiväkirjamerkintöjä ja osallistuin päiväpalaveriin. Keskustelin scrummasterin ja toisen tiimin jäsenen kanssa JSON-olioiden ominaisuuksien nimeämisestä. Yhdessä palvelinrajapinnassamme käytettiin JSON-olio-ominaisuuksia, joissa sanat oli erotettu väliviivalla, kun taas yleensä JSON-olioissa ominaisuuksien sanat on kirjoitettu camelCase-tyylillä, jossa muiden kuin ensimmäisen sanan ensimmäiset kirjaimet kirjoitetaan isolla, muut kirjaimet kirjoitetaan pienellä, ja sanat liitetään yhteen niin, että lopullisessa sanassa

on vain kirjaimia ja numeroita. Keskustelussa aluksi luulin, että Handlebars.js-mallipohjakieli ei hyväksyisi väliviivoja ominaisuuksien nimissä, mutta keskustelun aikana tekemässäni pikatestissä huomasin, että tämä ei pitänyt paikkansa. Silti suosittelin tiimin jäsentä käyttämään camelCase-tyyliä, koska se on yleinen käytäntö JavaScriptissä ja JSON-tiedostoissa.

Viikkoanalyysi

Yrityksemme on luomassa uutta käyttöliittymäalustaa, ja siirtämässä tuotteemme siihen. Tässä viikkoanalyysissä analysoin sitä.

Olen kuullut muilta kehittäjiltä, että alun perin yrityksen tuotteilla oli jokaisella täysin omat käyttöliittymät, eli niillä ei ollut yhtään yhteistä koodia. Jossain vaiheessa tästä tilanteesta haluttiin päästä eroon, koska ohjelmistokehittäjät huomasivat, että he tekivät paljon päällekkäistä työtä. Lisäksi asiakkaat ja asiakkaiden kanssa työskentelevät työntekijät valittivat käyttöliittymän sirpaloitumisesta: kaikki tuotteet näyttivät hyvin erilaisilta ja toimivat hyvin eri tavoin toisiinsa verrattuna; tuotteiden ”tuntuma” ei ollut samanlainen. Tähän vastauksena muodostettiin käyttöliittymäalustatiimi, ja käyttökokemus-, eli UX-tiimi. Käyttöliittymäalustatiimin tehtävä oli tutkia tuotteita, kerätä niistä usein käytettyjä komponentteja, tehdä komponentista versio, jota kaikki tuotteet voisivat käyttää, ja sitten auttaa muita tuotetiimejä alustan komponentin käyttöönotossa ja käytössä. UX-tiimi taas analysoi tuotteiden silloisia käyttöliittymiä, teki havaintoja niiden käytettävyyden samanlaisuuksista ja eroista, kirjoitti ohjeita käytettävyyden yhtenäistämiseen ja parantamiseen, ja teki tuotteille uusia designeja.

Vaikka nämä käytännöt paransivat kehitystä ja käyttökokemusta huomattavasti, niitä pidettiin puutteellisina, koska niistä huolimatta tuotteet usein käyttävät omia komponenttejaan, ja alustatiimi käyttää liikaa aikaa uusiin komponenttien toimintoihin, joita tarvitsee vain yksi tuote. Tämän takia päätettiin luoda uusi käyttöliittymäalusta. Uusi alusta on alusta alkaen UX:n suunnittelema. Lisäksi on tarkoitus että tulevaisuudessa tuotteiden käyttöliittymät kirjoitetaan kokonaan uudelleen niin, että ne oikeasti käyttävät alustaa pohjanaan. Jotta tämä onnistuisi, UX ja alustatiimi tekevät alustaan valmiita käyttöliittymäpohjia, joista tuotteiden on tarkoitus tehdä käyttöliittymänsä. Tämä varmistaa että tuotteiden käyttökokemus ja ulkonäkö ovat yhtenäiset. Uusi alusta ei tule käyttämään juuri yhtään koodia vanhasta alustasta.

Voi jopa sanoa että vanhalla käyttöliittymäalustalla oli deskriptiivinen eli kuvaileva suhde tuotteisiin, kun taas uudella alustalla on preskriptiivinen eli ohjaileva suhde tuotteisiin, koska vanha alusta kehitettiin *tuotteiden* mukaan, kun taas nyt *tuotteet* tehdään *alustan* mukaan.

4 Pohdinta ja päätelmät

Opinnäytetyön kirjoittamisen aikana ohjelmointitaitoni ovat kehittyneet yleisesti hyvin, muttei valtavasti. En ole ehtinyt opiskella C-kieltä, mutta sen opiskelu kiinnostaa minua vieläkin. JavaScriptistä olen oppinut ja tottunut uusiin ECMAScript 6 -standardin toimintoihin. Eteenkin nuolifunktiosyntaksi on tullut tutummaksi. Lisäksi olen oppinut hiukan React-kirjastosta (kuten viikkoanalyysistä 7 näkyy), mutta minulla on siinä vielä opittavaa, koska en ole vielä päässyt käyttämään sitä tiimin muiden kehittäjien kanssa.

Opinnäytetyön kirjoittamisen aikana olen itsenäisesti opiskellut sähköisestä esteettömyydestä/saavutettavuudesta, ja erityisesti *WAI-ARIA* (*Web Accessibility Initiative – Accessible Rich Internet Applications*) (W3C, 2014) -esteettömyysmääritelmästä. Olen käyttänyt tästä opiskelusta saatua tietoa jo hiukan töissä, mutta tulen todennäköisesti käyttämään sitä enemmän tulevaisuudessa, koska UX on alkanut panostamaan esteettömyyteen.

CSS-tyylittelystä olen oppinut paljon pieniä asioita. Esimerkiksi että kappaleen sisällä huomiota herättävä teksti kannattaa kursivoida, kun taas kappaleen ulkoa huomiota herättävä teksti kannattaa lihavoida. Tämän takia esimerkiksi `<dfn>`-elementin perustyyli kannattaa muuttaa kursiivista lihavoiduksi, koska `<dfn>`-elementin tarkoitus on tuoda esiin, missä jokin termi määritellään ensimmäistä kertaa. Näin ollen verkossa on haluttavampaa, että elementti on lihavoitu, koska silloin se tulee lukijan huomioon, kun hän tähyilee tekstiä. Olen oppinut paljon muutakin typografiasta, kuten m- ja n-ajatusviivojen käytöstä, ja kapiteeli-kirjasintyylistä.

Toinen mainittava CSS-tekniikka on Harry Robertsin (2012) suosittelemat ”näennäisrajoitetut valitsijat”, joka on kommenttityyli, jolla voi ilmaista, että valitsijaa tulee käyttää rajoitetusti, mutta ilman, että sitä oikeasti rajoittaa.

```
.keyboard-key {}      /* Valitsija ei kerro, mille elementille luokka on tarkoitettu! */
kbd.keyboard-key {}    /* Valitsija on liian tarkka, ja voi siksi aiheuttaa ongelmia! */
/*kbd*/.keyboard-key {} /* Näennäisrajoitettu valitsija. */
```

Tämä kommenttityyli on *erittäin* hyödyllinen, koska oikea rajoittaminen lisää valitsijan tarkkuutta (”specificity”), mikä voi aiheuttaa ongelmia muiden tyylien kanssa.

Tarkkaavaisuuteni herpaantui sidosryhmäkuviota (Kuvio 1) tehdessäni, ja eksyin opiskelemaan Scalable Vector Graphics (SVG) -vektorigrafiikkakieltä. Tämä opiskelu ei auttanut minua paljoa opinnäytetyön teossa, mutta siitä tulee todennäköisesti olemaan hyötyä tulevaisuudessa, eteenkin koska SVG-kuvien käyttö vaikuttaa olevan kasvamassa – yrityksessämme ja yleisesti.

Opin paljon kirjoittamisesta yleensä. Opin miten pystyn paremmin erottelemaan tekstissä oman ääneni muiden äänistä. Olen aikaisemmin olettanut, että lainaukset kannattaa tehdä suorina, mutta nyt olen oppinut käyttämään enemmän parafraaseja. Olen oppinut hyvien *verkko*lähteiden etsimistä, ja niistä valitsemista. Verkkolähteissä kannattaa yleensä avata siinä olevat linkit muihin lähteisiin, ja sitten verrata näitä lähteitä ensimmäiseen lähteeseen.

Yksi asia, jonka sisäistin nyt vahvemmin, on että kirjoittaminen tehdään aina jossakin formaatissa ja jollekin kohdeyleisölle, ja että jokaisella formaatilla ja yleisöllä on omat odotuksensa tekstistä, ja näin ollen teksti tyylin pitää vastata. Esimerkiksi Nielsen Norman Groupin Hoa Lorangerin (2014) mukaan verkossa saa *ja kannattaakin* kirjoittaa lauseentynkiä, kirjoittaa luvut numeroina, ja pitää kappaleet lyhyinä – kun taas painetussa tekstissä (kuten esseissä) kannattaa käyttää kokonaisia lauseita, kirjoittaa pienet luvut kirjaimin, ja pitää kappaleet 3–5 virkkeen pituisina. Mielestäni suomalainen perus- ja lukiokoulutus keskittyy liikaa esseekirjoittamiseen, ja näin ollen monelle tulee kuva, että on pelkästään yhdet säännöt, joiden mukaan kirjoitetaan – kun säännöt oikeasti riippuvat formaatista ja yleisöstä. Lisäksi mielestäni pitäisi painottaa *miksi* jossain formaatissa on jokin sääntö, jotteivat säännöt tunnu oppilaista mielivaltaisilta.

Kirjoittamisessa muutenkin kannattaa pitää yleisö mielessä. Yksi esimerkki tästä on lyhenteiden käyttö. Minua itseani usein häiritsee, kun tekstissä käytetään lyhennettä, jota ei ole avattu. Esimerkiksi American Society for Microbiology -yhdistyksen (2016) *mBio*-lehti suosittelee avaamaan lyhenteet kun niitä käyttää ensimmäisen kerran tekstissä. Lehti myös painottaa, että lyhenteiden käyttö tulee olla aina *lukijan* eduksi, *eikä* kirjoittamisen helpottamiseksi. Opin myös lehden ohjeista, että lyhenteiden kanssa kannattaa käyttää apusanoja. Esimerkiksi, en sijaan että puhuisi ”HTML:stä”, puhuu ”HTML-kielestä”, ”HTML-koodista” tai ”HTML-tiedostosta”. Myöhemmin samaan voi viitata käyttämällä pelkkää apusanaa, kuten: ”kieli”, ”koodi”, tai ”tiedosto”.

Toinen tapa ottaa yleisö huomioon, on keskittyä yksityiskohtiin, joita yleisö haluaa tietää, ja välttää yksityiskohtia, joita yleisö ei ymmärrä, tai jotka eivät yleisöä kiinnosta. Minun kykyni katsoa asioita yleisön näkökulmasta on parantunut. Kolmas (eteenkin suomalaisille osoitettu) tapa on välttää turhaa englanninkielisten termien käyttöä, eli anglismeja. Opinnäytetyön kirjoittamisen johdosta pystyn paremmin yhdistämään englanninkieliset sanat niiden suomenkielisiin vastineisiin ja päinvastoin.

Toinen asia, jonka eritysesti huomasin kirjoittaessa, että on parempi kirjoittaa ensin ”puutaheinää”, ja sitten korjata se; *kuin* että odottaisi ”*oikeiden*” sanojen tulemista mieleen. Tällöin, ”puuheinä”-teksti muodostaa kappaleelle rungon, johon sitten voi lisätä tekstiä väliin, ja parannella jo kirjoitettua tekstiä. Tämä on mielestäni hyvä tekniikka, koska on usein helpompaa parannella kuin luoda. Joskus myös ”puuheinä”-teksti tuntuu aluksi kamalalta, mutta kun tekstiä on työstänyt ja katselmoinut jonkin aikaa, se ei enää vaikuta niin pahalta kuin aluksi tunsi. Lisäksi, jos on jo valmiiksi miettinyt jonkin luvun yleiset aiheet ranskalaisin viivoin, on järkevää ensin kirjoitta yksittäiset ”kappaleenaloituslauseet”, ja sitten lisätä kappaleiden teksti niiden ympärille; kuin että yrittäisi kirjoittaa luvun kokonaan alusta loppuun.

Analyysien kirjoittamisesta on ollut jossain määrin hyötyä. Analyysit ovat mielestäni auttaneet opittujen asioiden jäämisessä mieleen. Analyysit myös ovat helpottaneet selittämään asiat paremmin muille, koska kirjoittaessa asiat pitää esittää selkeästi ja kiteytetysti. Lisäksi olen analyyseissä huomannut joitakin asioita, joita en olisi välttämättä huomannut tavallisesti. En esimerkiksi olisi miettinyt Markdownin turvallisuutta, ellen olisi analysoinut sitä.

Datan arkistoiminen on yksi ratkaisumalleista, johon olen kiinnittänyt enemmän huomiota, monesta syystä:

- Pääkotikoneeni kovalevy on viisi vuotta vanha, ja näin ollen epäluotettava.
- Tiedostojen salaavien kiristysohjelmien levitys on kasvussa.
- Olen lukenut monista tapauksista, joissa vanhaan tietoon ei ole päästy käsiksi, tai siihen käsiksi pääseminen on ollut vaikeaa tai kallista.

Datan säilytyksessä kannattaa muistaa Jack Schofieldin (2014) kolme tietojenkäsittelyn lakia:

1. Älä laita dataasi ohjelmaan, [palveluun, tai laitteeseen;] ellet tiedä tarkkaan miten sen saa ulos.
2. Dataasi ei voida katsoa *oikeasti* olevan olemassa, ellei sinulla ole sitä vähintään kahtena kappaleena.
3. Mitä helpompi pääsy sinulla on dataasi, sitä helpompi pääsy on *muilla* dataasi.

Datan säilymisen vuoksi on hyvä pyrkiä pitämään kaikki data vapaissa tiedostoformaateissa, kuten Markdownissa, PDF-arkistotiedostossa (PDF/A), ja OpenDocumenteissa. Omisteisten formaattien käyttöä kannattaa välttää, koska ne ovat joko kokonaan tai osittain sidottu niitä luoviin ohjelmiin, ja näin ollen dataan ei välttämättä pääse enää käsiksi,

jos vaikka ohjelman myynti tai ylläpito lopetetaan tulevaisuudessa. Jos ei ole varmuutta että *sinä* pääset dataasi käsiksi, voi miettiä onko data *oikeasti sinun*.

Opinnäytetyöstä löytyy monia mahdollisia kohteita jatkokehitykselle ja -tutkimukselle.

Kehitystiimini on nyt siirtynyt käyttämään kanbania, ja yritys SAFe:a – niin kuin mainitsin olevan aikomus viikolla 4. Kehitystiimillä on ollut vaikeuksia sopeutua näihin muutoksiin – eteenkin kun on tullut muitakin muutoksia. Kehitystiimistä on lähtenyt muutama jäsen, ja siihen on tullut uusia jäseniä. Kehitystiimin front-end-, ja back-end-ohjelmoijat on jaettu eri tiimeihin, ja UX on liitetty front-end-tiimiin. Kuulun nyt yhdistettyyn front-end/UX-tiimiin, ja näin ollen back-end työn tekeminen pitäisi harvinaistua entisestään. Tiimin tuoteomistaja ja arkkitehti ovat myös vaihtuneet. Näistä kevään muutoksista olisi voinut saada oivaa analyysiä, ja tiimin nykyisestä asettelua voisi myös tutkia.

Tiimillämme ei ole tällä hetkellä aikomuksia käyttää Robot Frameworkia, ja emme vielä tiedä tarkalleen, miten tulemme testaamaan käyttöliittymän. Kun uusi testaamismetodi päätetään, sitä voisi verrata Robot Frameworkiin. Muut tuotetiimit käyttävät vielä Robot Frameworkia.

Uusi käyttöliittymä alkaa olla melkein valmis käyttöön tuotteissa. Kun alusta on saanut ensimmäisen virallisen julkaisunsa, ja kun sitä käytetään useammassa yrityksen tuotteessa, sitä voisi verrata edelliseen alustaan. Lisäksi silloin voisi tutkia, miten alusta on muuttunut lähtösuunnitelmista käyttöönottoon, koska alustaan on tehty muitakin muutoksia kuin aikaisemmin mainitsemani Backbone–React-muutos.

Viikolla 3 analysoimaani Volkswagen-skandaalia puidaan vielä, ja olen varma, että sitä tällä hetkellä tutkii monet organisaatiot, juristit, ja ohjelmistokehittäjät. Skandaalia todennäköisesti tullaan myös tutkimaan paljon tulevaisuudessa, ja toivon että esimerkiksi autojen ja lentokoneiden omisteiseen ohjelmistoon tulisi enemmän läpinäkyvyyttä, jotta tällaiset tapaukset vältettäisiin tulevaisuudessa.

Lähteet

Abelson, H., Sussman, G. & Sussman, J. 1984. *Structure and Interpretation of Computer Programs*. Luettavissa: https://mitpress.mit.edu/sicp/. Luettu: 25.9.2015.

American Society for Microbiology (ASM). 2016. *mBio™ Instructions to Authors - Abbreviations and Conventions*. Luettavissa: http://mbio.asm.org/site/misc/journal-ita_abb.xhtml#02. Luettu: 13.4.2016.

Atwood, J. 2006. *I Shall Call It.. SomethingManager*. Luettavissa: http://blog.codinghorror.com/i-shall-call-it-somethingmanager/. Luettu: 28.10.2015.

Atwood, J., Dumke-von der Ehe, B., Greenspan, D., MacFarlane, J., Marti, V. & Williams, N. 2016. *CommonMark*. Luettavissa: http://commonmark.org/. Luettu: 15.4.2015.

Bollier, D. 2015. *Volkswagen Scandal Confirms the Dangers of Proprietary Code*. Luettavissa: http://www.bollier.org/blog/volkswagen-scandal-confirms-dangers-proprietary-code. Luettu: 23.3.2016.

Crockford, D. 2008. *JavaScript: The Good Parts*. O'Reilly Media / Yahoo Press.

Dwyer, J. 2015. *Volkswagen's Diesel Fraud Makes Critic of Secret Code a Prophet*. Luettavissa: http://www.nytimes.com/2015/09/23/nyregion/volkswagens-diesel-fraud-makes-critic-of-secret-code-a-prophet.html. Luettu: 23.3.2016.

Leffingwell, D. 2016. *About – Scaled Agile Framework*. Luettavissa: http://scaledagileframework.com/about/. Luettu: 1.4.2016.

Lekman Consulting. 2014. *Uusi suomenkielinen Scrum Guide*. Luettavissa: https://lekman.fi/scrumguide/. Luettu: 25.4.2016.

Loranger, H. 2014. *Break Grammar Rules on Websites for Clarity.* Luettavissa: https://www.nngroup.com/articles/break-grammar-rules/. Luettu: 13.4.2016.

Moglen, E. 2010. *When Software is in Everything: Future Liability Nightmares Free Software Helps Avoid*. Luettavissa: https://www.softwarefreedom.org/events/2010/sscl/moglen-software_in_everything-transcript.html. Luettu: 23.3.2016.

Mozilla Developer Network. 2016. *Mozilla Developer Network.* Luettavissa: https://developer.mozilla.org/. Luettu: 12.4.2016.

Roberts, H. 2012. *Quasi-qualified CSS selectors*. Luettavissa: http://csswizardry.com/2012/07/quasi-qualified-css-selectors/. Luettu: 14.4.2016.

Schofield, J. 2014. *Follow Schofield's Three Laws of Computing and avoid disasters*. Luettavissa: http://www.zdnet.com/article/follow-schofields-three-laws-of-computing-and-avoid-disasters/. Luettu: 20.4.2016.

Schwaber, K. 2013. *unSAFe at any speed*. Luettavissa: https://kenschwaber.wordpress.com/2013/08/06/unsafe-at-any-speed/. Luettu: 1.4.2016.

Schwaber, K. & Sutherland, J. 2013. *The Scrum Guide*. Luettavissa: http://www.scrumguides.org/scrum-guide.html. Luettu: 5.4.2016.

Stack Exchange, Inc. 2016. *Stack Exchange.* Luettavissa: http://stackexchange.com/. Luettu: 12.4.2016.

Waters, K. 2012. *Risk Management – How to Stop Risks from Screwing Up Your Projects!* Luettavissa: http://www.allaboutagile.com/risk-management-how-to-stop-risks-from-screwing-up-your-projects/. Luettu: 23.10.2015.

World Wide Web Consortium (W3C). 2014. *Accessible Rich Internet Applications (WAI-ARIA)*. Luettavissa: https://www.w3.org/TR/wai-aria/. Luettu: 14.4.2016.

World Wide Web Consortium (W3C). 2014. *HTML5*. Luettavissa: https://www.w3.org/TR/html5/. Luettu: 4.4.2016.