Karelia-ammattikorkeakoulu
Tietojenkäsittely (AMK)

TDD ja testausautomaatio osana mobiilisovelluskehitystä

Sauli Karvinen

Opinnäytetyö, joulukuu 2023

www.karelia.fi

OPINNÄYTETYÖ
Joulukuu 2023
Tietojenkäsittelyn koulutus

Tikkarinne 9
80200 JOENSUU
+358 13 260 600 (vaihde)

Tekijä(t)
Sauli Karvinen

Nimeke
TDD ja testausautomaatio osana mobiilisovelluskehitystä

Tiivistelmä

Opinnäytetyö on päiväkirjamallinen ja se käsittelee kirjoittajan ammatillista kehittymistä mobiilikehittäjänä työympäristössä 13 viikon aikana. Pääpaino opinnäytetyössä on testivetoisessa iOS-kehityksessä ja työskentely tapahtuu osana kehitystiimiä ryhmä- ja pariohjelmointina. Opinnäytetyössä käsitellään myös testauksen automatisointia ja arkkitehtuurillisten ratkaisujen vaikutusta testattavuuteen. Opinnäytetyön tavoitteena on kehittyä ammatillisesti mobiilikehittäjänä ja oppia kirjoittamaan rakenteeltaan parempaa ja helpommin ylläpidettävää ohjelmakoodia testausvetoisesti.

Opinnäytetyö koostuu aloitustilanteen kuvauksesta, päivittäisistä päiväkirjamerkinnöistä ja viikoittaisista analyyseista sekä pohdinnasta. Aloitustilanteessa tarjotaan tietoa yrityksestä, toimintatavoista ja opinnäytetyön kirjallisesta pohjasta. Päiväkirjassa kuvaillaan päivittäisiä työtehtäviä ja osaamisen kehittymistä viikkotasolla. Pohdinnassa käsitellään koko kolmentoista viikon ajanjakson aikana tapahtunutta työskentelyä ja kehitystä kokonaisuutena.

Opinnäytetyöni aikana olen päässyt työskentelemään osana kehitystiimiä ja oppinut kirjoittamaan ohjelmakoodia Swift-ohjelmointikielellä noudattaen alalla laajasti hyväksi todettuja toimintatapoja. Olen oppinut heksagonaalisen arkkitehtuurin soveltamista ohjelmakoodissa ja kuinka arkkitehtuurilliset ratkaisut auttavat luomaan testattavampaa ohjelmakoodia. Olen oppinut kirjoittamaan testivetoista ohjelmakoodia iOS-sovelluksille, joka on auttanut ymmärtämään Swift-ohjelmointikieltä ja iOS-sovellusten toimintaa paremmin.

Kieli
suomi

Sivuja 74

Asiasanat
TDD, testausautomaatio, mobiilikehitys, iOS, sovellusarkkitehtuuri

THESIS
December 2023
Degree Programme in Business Information Technology

Tikkarinne 9
80200 JOENSUU
FINLAND
+ 358 13 260 600

Author (s)
Sauli Karvinen

Title
TDD and Test Automation in Mobile Application Development

Abstract

The thesis is presented in diary format and explores the author's professional growth as a mobile developer in a 13-week work setting. The thesis focuses on test-driven iOS development, implemented in collaboration with a development team using group and pair programming. Additionally, the thesis explores test automation and the impact of architectural decisions on testability. The goal of the thesis is to grow professionally as a mobile developer and learn to write structurally better and more maintainable code using test-driven development practices.

The thesis consists of an initial situation overview, daily diary entries, weekly analyses, and reflections. The initial situation section introduces the company, its practices, and the theoretical background. The diary entries detail daily tasks and skill development weekly, while the reflection section evaluates the overall progress over the 13-week period.

Collaboration in the development team over the research period provided valuable experience, enhancing my skills in Swift programming following the best practices of the IT industry. I have learned to utilize the best practices of the hexagonal architecture and how architectural decisions impact the testability of the application. Writing test-driven code for iOS applications has strengthened my understanding of the Swift programming language, and it has also deepened my understanding on how iOS applications work.

Language
Finnish

Pages 74

Keywords
TDD, test automation, mobile development, iOS, software architecture

Sisältö

1 Johdanto 7
 1.1 Aloitustilanne 7
 1.2 Kehittymisen arviointi 8
 1.2 Sidosryhmät ja vuorovaikutus työpaikalla 9
2 Opinnäytetyön tietoperusta 11
 2.1.1 TDD ja testausautomaatio 11
 2.1.2 Puhdas ohjelmakoodi 14
 2.1.3 Sovellusarkkitehtuuri 16
3 Päiväkirjatyöskentely 17
 3.1 Viikko 1 (14.-18.8.2023) 17
 3.2 Viikko 2 (21.-25.8.2023) 22
 3.3 Viikko 3 (28.8.-1.9.2023) 26
 3.4 Viikko 4 (4.-8.9.2023) 31
 3.5 Viikko 5 (11.-15.9.2023) 35
 3.6 Viikko 6 (18.-22.9.2023) 39
 3.7 Viikko 7 (25.-29.9.2023) 43
 3.8 Viikko 8 (2.-6.10.2023) 47
 3.9 Viikko 9 (9.–13.10.2023) 51
 3.10 Viikko 10 (16.–20.10.2023) 55
 3.11 Viikko 11 (23.–27.10.2023) 58
 3.12 Viikko 12 (30.10.-3.11.2023) 62
 3.13 Viikko 13 (13.-17.11.2023) 66
4 Pohdinta 70
Lähteet 74

Sanasto

Protokolla	Interface, rajapinta. Määrittää abstraktit säännöt sille, mitä rajapintaa toteuttavien luokkien täytyy sisältää
Sovellusarkkitehtuuri	Sovelluksen rakenteellisuuden muoto, joka määrittää miten sovelluksen eri osat ovat organisoitu ja ovat vuorovaikutuksessa keskenään.
Hexagonal Architecture	Sovellusarkkitehtuurin muoto, jonka tarkoitus on eriyttää eri sovelluksen osat toisistaan siten, ettei mikään osa ole tiukasti riippuvainen toisesta ja mikä tahansa sovelluksen osa voidaan vaihtaa toiseen ilman suuria muutoksia ohjelmakoodissa. Kutsutaan myös nimellä Ports and Adapters -arkkitehtuuri.
Port	Rajapinta, joka määrittää abstraktin toiminnallisuuden sovelluksen sisään tai ulos tulevan datan ja sovelluksen välillä.
Adapter	Adapteri, joka toteuttaa portin määrittämän toiminnallisuuden sovelluksen sisään tai ulos tulevan datan ja sovelluksen välillä.
TDD	Test Driven Development/Testivetoinen kehitys. Sovelluskehityksen muoto, jossa toiminnallisuudelle kirjoitetaan aina ensin testi, jonka jälkeen kirjoitetaan itse toiminnallisuus läpäisemään testi.
Yksikkötesti	TDD koostuu enimmäkseen yksikkötesteistä. Yksikkötestillä testataan yhtä ohjelman komponenttia.
SUT	Testattava komponentti (System Under Test).

UI-Test	Käyttöliittymän toiminnallisuuden testit. Testaa esimerkiksi sitä, että tapahtuuko näytöllä odotetut muutokset, kun käyttäjä painaa jotakin nappia.
End-to-end testi	Testaa sovelluksen toimintaa alusta loppuun.
iOS	Apple iPhone -älypuhelinten käyttöjärjestelmä.
Swift	Ohjelmointikieli iOS-käyttöjärjestelmälle.
XCode	Kehitysympäristö iOS-käyttöjärjestelmän sovelluksille.
Refaktorointi	Ohjelmakoodin sisäisen toteutuksen muokkaaminen esimerkiksi optimoinnin takia tai luettavuuden parantamiseksi muuttamatta ulkoista toiminnallisuutta.
Test Double	Luokasta luotu jäljennös testejä varten, joka simuloi testattavan luokan toimintaa.
Stub	Jäljennös luokasta, joka palauttaa erilaisia arvoja erilaisilla syötteillä.
Mock	Jäljennös luokasta, jonka avulla voidaan tarkastella testattavan järjestelmän ja järjestelmän kutsumien osien vuorovaikutusta.
Spy	Jäljennös luokasta, joka pitää kirjaa arvoista, joilla luokkaa kutsuttiin testeissä.

1 Johdanto

Tämän alkuanalyysin ensimmäisessä osassa käydään läpi opinnäytetyön aloitustilanne, kirjoittajan työympäristö ja työtehtävät sekä opinnäytetyön keskeiset teemat ja oppimistavoitteet. Toinen osa käsittelee kehittymisen arviointia, jossa arvioidaan kirjoittajan osaamisen tasoa suhteessa työpaikan ja työtehtävien osaamisvaatimuksiin sekä sitä, millä tasolla kirjoittajan ammatillinen kehittyminen on suhteessa kirjoittajan asettamiin kehittymistavoitteisiin. Kolmannessa osassa esitellään työhön liittyvät sidosryhmät ja näiden roolit työympäristössä. Neljännessä osassa kartoitetaan opinnäytetyön tietoperustaa opinnäytetyön teemoista ja avataan olennaisia työhön liittyviä käsitteitä ja termistöä.

1.1 Aloitustilanne

Opinnäytetyö kirjoitetaan päiväkirjamuotoisena työn ohessa. Yritys, jossa työskentelen tuottaa erilaisia lukitus- ja kulunhallintajärjestelmiä ja asiakkaita on ympäri maailman monenlaisissa kohteissa, joissa tarvitaan kyseisen kaltaisia järjestelmiä. Työtehtävissäni työskentelen osana mobiilikehitystiimiä, jossa kehitetään mobiilisovellusta avaimettomien lukituslaitteiden ohjaamista varten. Yrityksessä on kaksi mobiilikehitystiimiä; Android- sekä iOS-sovelluksia kehittävät tiimit. Suoritan opinnäytetyöni enimmäkseen iOS-sovellusta kehittävässä tiimissä. Lisäksi päivittäisiin työtehtäviin kuuluvat sovelluksen testaaminen, kokoukset ja tutkimustyö. Mobiilisovelluksen lisäksi yrityksessä kehitetään myös erillisiä SDK-paketteja sekä Android-, että iOS-käyttöjärjestelmille, joita myydään eteenpäin yrityksille ympäri maailman, jotka voivat luoda omat mobiilisovelluksensa hyödyntäen SDK-pakettien tarjoamaa toiminnallisuutta. Työtehtävät vaativat kykyä työskennellä ryhmässä, tuntemusta ohjelmoinnin perusteista sekä englannin kielen taitoa, koska työkieli on suurimmaksi osaksi englanti. Koen työtehtäväni vaatimustasoltaan hyvin vaativiksi osaamisen tasooni nähden opinnäytetyöni alkuvaiheessa ja tarvitsen paljon työkavereideni opastusta päivittäisessä työskentelyssä.

1.2 Kehittymisen arviointi

Pääpaino opinnäytetyössä on natiivi iOS-sovelluksen kehityksessä osana kehitystiimiä ryhmä- ja pariohjelmointina työskennellen. Tuotetta kehitetään testausvetoisesti (TDD) ja opinnäytetyössä paneudutaan myös testien kirjoittamiseen ja niiden automatisointiin. Opinnäytetyössä sivutaan myös sovelluksen arkkitehtuurillisia ratkaisuja ja sitä, kuinka hyvillä arkkitehtuurisilla ratkaisuilla saadaan tuotettua helpommin testattavaa ja rakenteeltaan parempaa sekä ylläpidettävämpää ohjelmakoodia. Opinnäytetyön alkuvaiheessa koen olevani aloitteleva toimija, koska monet opinnäytetyöhöni liittyvät asiat eivät ole minulle entuudestaan tuttuja ja työtehtävistä suoriutuminen vaatii työkavereideni ohjeistusta. Opinnäytetyöni aikana tavoitteeni on kasvattaa ammatillista osaamistani siten, että kykenen etsimään tietoa käsillä olevista ongelmista ja tämän pohjalta luomaan rakenteeltaan hyviä ja ylläpidettäviä sekä testattavia ratkaisuja perustason ongelmiin ja esittämään omia ratkaisujani tiimille.

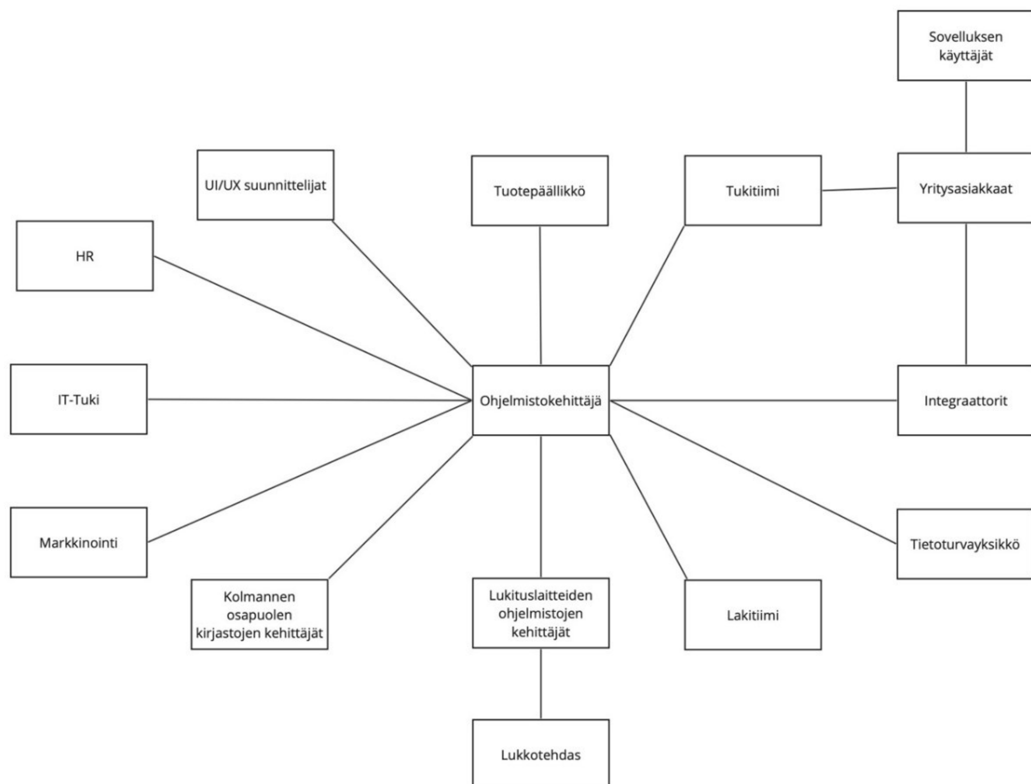
Pari- ja ryhmäohjelmointi on erinomainen tapa kasvattaa tuntemusta tuotteen koodipohjasta, yrityksen työtavoista ja käytänteistä sekä ohjelmoinnista yleisesti. Usean eri ihmisen ideoiden, ratkaisujen ja toteutuksien näkeminen auttaa kehittymään kohti tavoitteita.

Mitä tutummaksi ohjelmakoodi tulee, sitä paremmin ymmärtää sovelluksen arkkitehtuuria, rakennetta, testaustapoja ja nimeämiskäytänteitä. Tämä auttaa minua kasvattamaan itseluottamustani tuotteen kehittäjänä ja lopulta pääsemään lähemmäs oppimistavoitteitani.

Kehittymistä arvioidaan viikkotasolla jokaisen viikon päätteeksi viikkoanalyyseissa. Työjakson aikana tapahtunutta kehitystä verrataan opinnäytetyön alkutilanteeseen laajemmin opinnäytetyön pohdinnassa työn lopussa.

1.2 Sidosryhmät ja vuorovaikutus työpaikalla

Ohjelmistokehittäjät (kuvio 1) vastaavat tuoteperheen kehittämisestä, testaamisesta ja julkaisusta. Tuoteperheen kehitys koostuu mobiilisovelluksesta, backendista ja Mobile SDK -paketeista, joita tarvitaan lukituslaitteiden käyttöön ja ohjaamiseen. Lukituslaitteiden sisäisestä ohjelmistosta vastaavat lukituslaitteiden ohjelmistojen kehittäjät ja mobiilisovellus kommunikoi tämän ohjelmiston kanssa. Lukituslaitteet tulevat tehtaalta, joka sijaitsee samassa toimipisteessä ohjelmistokehityksen kanssa. Ohjelmisto hyödyntää laajasti kolmannen osapuolen kirjastoja, joita kehittävät yrityksen alaisuudessa toimivat kolmannen osapuolen kehittäjät. Mobile SDK -pakettia tarjotaan integraattoreille, jotka hyödyntävät Mobile SDK -pakettia ja backendia omiin sovelluksiinsa.



Kuvio 1. Organisaatiokaavio

Varsinaisia asiakkaita tuotteelle ovat yritysasiakkaat, jotka ostavat yritykseltä lukitusratkaisun. Sovelluksen käyttäjiä ovat yritysasiakkaan työntekijät ja omat asiakkaat, joilla on tarve avata ja sulkea lukituslaitteita. Yritysasiakkaat voivat ostaa Mobile SDK -paketin, backendin ja lukot ja kehittää oman mobiilisovelluksensa näiden päälle, joita yritysasiakkaan asiakkaat käyttävät. Tukitiimi toimii ohjelmistokehittäjien ja yritysasiakkaiden välissä, tuoden ilmi asiakkaiden ongelmia ja ohjeistaen asiakkaita ongelmatilanteissa.

Tuotteen tietoturvallisuus on elintärkeä osa tuoteperheen toimintaa. Tietoturvaan liittyvistä asioista vastaa tietoturvayksikkö, joka ohjaa tietoturvaan liittyvissä asioissa ja valvoo tietoturvan toteutumista. Koska tuote kerää tietoa siitä, kuinka asiakkaat käyttävät tuotetta, jotta voimme saada raportteja esimerkiksi ohjelmassa tapahtuneista virhetilanteista, on meidän noudatettava GDPR-säännöksiä asiakkaiden tietojen keräämisessä. Toimimme yhdessä lakitiimin kanssa ja tarkastamme, että toimimme GDPR-säännösten mukaisesti. Lakitiimi vastaa myös tietosuojalausekkeen sisällöstä ja sen noudattamisesta. Ohjelmistokehittäjän näkökulmasta tietoturvayksikön ja lakitiimin kommunikointi on tärkeää ja samalla opettavaista. Kehitystyössä tulee välillä vastaan haasteellisia kysymyksiä tietoturvasta ja lakipykälistä, jolloin voimme olla yhteydessä näihin tiimeihin. Samalla henkilökohtainen ymmärrys tietoturva- ja lakiasioissa kasvaa.

Sovelluksen käyttöliittymästä ja käyttökokemuksesta vastaavat UI/UX-suunnittelijat, joiden kanssa ohjelmistokehittäjät kommunikoivat näihin liittyvissä asioissa. Yrityksen sisällä toimii oma IT-tukitiimi, joka myöntää ohjelmistokehittäjille työskentelylaitteet sekä ohjeistaa laitteiden asennuksessa ja käyttöönotossa. Ohjelmistokehittäjät osallistuvat toisinaan erilaisiin tapahtumiin, kuten rekrytointitapahtumiin ja näiden tapahtumien käytännön asioista vastaa HR-tiimi. HR-tiimi vastaa ja ohjeistaa myös mahdollisissa matkustuskuluihin liittyvissä asioissa. Välillä tarvitsemme markkinointimateriaalia käyttöömme, jollaisia ovat esimerkiksi kuvat, joita saatamme käyttää sovelluksessa. Markkinointimateriaaleihin liittyvissä asioissa olemme yhteydessä markkinointitiimiin. Tuotepäällikkö kommunikoi kaikkien sidosryhmien kanssa ja välittää sekä tarjoaa tietoa

ohjelmistokehittäjille. Tuotepäällikkö vastaa mitä, milloin ja missä järjestyksessä sovelluksiin rakennetaan.

Ulkoisiin sidosryhmiin kuuluvat yritysasiakkaat, integraattorit ja sovelluksen käyttäjät. Loput sidosryhmistä ovat yrityksen sisäisiä sidosryhmiä.

2 Opinnäytetyön tietoperusta

Opinnäytetyön tietoperusta pohjautuu suurimmaksi osaksi alan kirjallisuuteen ja kirjallisuuden tarjoamiin käytäntöihin. Työpaikassa, jossa opinnäytetyö tuotetaan, sovelletaan laajasti opinnäytetyön lähdekirjallisuudessa tarjottuja käytäntöjä ja toimintatapoja. Ohjelmakoodin rakenteeseen ja testattavuuteen liittyviä asioita käsitellään Robert C. Martinin (2008) kirjoittamassa kirjassa Clean Code: A Handbook of Agile Software Craftmanship. Kirjan ohjeistukset auttavat luomaan rakenteeltaan parempaa ja eriytettyä ohjelmakoodia, jonka tarkoitus on parantaa tuotteen ylläpidettävyyttä sekä lyhyellä, että pitkällä aikavälillä. Sovellusarkkitehtuuria käsitellään Robert C. Martinin (2018) kirjassa Clean Architecture: A Craftsman's Guide to Software Structure and Design ja Vladimir Khorikovin (2020) kirjassa Unit Testing Principles, Practices, and Patterns. Sovellusarkkitehtuurisilla ratkaisuilla sovelluksesta voidaan tehdä paremmin skaalautuva ja muokattava. Yksikkötestauksen tietoperusta pohjautuu enimmäkseen Clean Code ja Unit Testing Principles, Practices, and Patterns kirjoihin, mutta myös muuta alan kirjallisuutta on käytetty yksikkötestauksen tietoperustana.

2.1.1 TDD ja testausautomaatio

Sovelluksen testaukseen on olemassa lukuisia erilaisia keinoja. Sovelluksen toiminnallisuutta voidaan testata käytännössä ohjelmakoodin kirjoittamisen jälkeen ja kokeilla toimiiko tuotettu toiminnallisuus odotetusti. Manuaalinen testaus on kuitenkin hidasta ja todella aikaa vievää eikä manuaalisella testauksella saada välttämättä tehokkaasti kiinni hajonneita osia. Manuaalisella testauksella tulisi

käydä läpi kaikki mahdolliset erilaiset vaihtoehdot, jotka voivat vaikuttaa ohjelman toimintaan ja nämä testit olisi toistettava joka kerta, kun ohjelmakoodia muutetaan tai viimeistään kun ohjelman uutta versiota ajetaan tuotantoon.

Sovelluksella voi olla ulkoinen testaustiimi, joka suorittaa sovelluksen testauksen, tai kehittäjät voivat tehdä testauksen itse. Sovellukselle voidaan myös kirjoittaa testitapauksia, jotka voidaan suorittaa aika ajoin. Tulokset näyttävät joko kaiken edelleen toimivan, tai testien epäonnistuessa tulokset paljastavat muutoksien aiheuttamat ongelmat. Opinnäytetyössä keskitytään enimmäkseen testausvetoiseen kehitykseen ja yksikkötesteihin.

Testausvetoisessa kehityksessä testit kirjoitetaan aina ennen toiminnallisuuden tuottamista. Testausvetoisen kehityksen tarkoituksena on kirjoittaa pienin mahdollinen määrä toimivaa ohjelmakoodia kerrallaan. Testitapausten kasvaessa ohjelmakoodin testauskattavuus voidaan saada verrattain suureksi, joka lisää luottoa ja turvallisuutta ohjelman toimintaan kehitystyön jatkuessa ja ohjelman kasvaessa. Testausvetoisessa kehityksessä kehittäjällä on ensin idea siitä, millainen toiminnallisuus ohjelmaan pitäisi tuottaa. Tämän jälkeen kirjoitetaan testi, joka testaa odotettua lopputulosta. Testivetoisessa kehityksessä testitapaukset ovat yleensä yksikkötestejä, joiden tarkoituksena on testata jonkin yksittäisen funktion, luokan tai komponentin toiminnallisuutta. Kun testi on kirjoitettu, testit ajetaan ensimmäisen kerran. Koska toiminnallisuutta ei ole vielä kirjoitettu, testi ei mene läpi, joka on tässä kohtaa odotettu lopputulos. Vasta epäonnistuneen testitapauksen jälkeen kirjoitetaan toiminnallisuus, joka on valmis silloin kun testitapaus läpäistään. Tämän jälkeen suoritetaan ohjelmakoodin muokkausta (refaktorointia), joka tarkoittaa testitapausten ja toiminnallisuuden laajentamista. Kun toiminnallisuus on valmis, siirrytään kirjoittamaan seuraavaa testitapausta ja toiminnallisuutta. Martin (2008) mainitsee testausvetoiselle kehitykselle kolme lakia, jotka ovat:

1. Tuotantokoodia ei saa kirjoittaa ennen kuin on kirjoittanut epäonnistuneen testitapauksen.

2. Yksikkötestiä kirjoitetaan vain sen verran, kuin testin epäonnistumiseksi on riittävää. Se, että ohjelmakoodi ei käänny, on epäonnistumista.

3. Tuotantokoodia kirjoitetaan vain sen verran, kuin testin läpäisemiseksi on riittävää. (Martin 2008, 122.)

Mitä suuremmaksi ohjelmakoodin testauskattavauus saadaan, sitä enemmän kehittäjä voi luottaa ohjelman toimintaan, kun ohjelmaan tehdään ajan kanssa muutoksia. Ilman testejä ei ole täyttä varmuutta, etteivätkö muutokset ohjelmakoodissa olisi rikkoneet vanhaa toiminnallisuutta. Jotkin ongelmat ilmenevät nopeasti ja ilman testejä kyseiset tapaukset voidaan todeta ennen rikkinäisen ohjelmakoodin pääsemistä tuotantoon ja asiakkaille. Jotkin ongelmat taas piiloutuvat ja ilmenevät vai tietyissä tapauksissa ja ovat riippuvaisia erilaisista olosuhteista tai syötteistä. Tällaisten ongelmien löytäminen ilman testausautomaatiota voi olla todella aikaa vievää ja jopa mahdotonta. Testausautomaatiossa kaikki kirjoitetut testit suoritetaan useita kertoja päivässä, joka mahdollistaa virheiden löytämisen erittäin nopeasti, jolloin ongelmiin voidaan reagoida välittömästi ennen rikkinäisen ohjelmakoodin päätymistä tuotantoon.

Martin mainitsee viisi periaatetta, joita yksikkötestien tulisi noudattaa. Nämä viisi periaatetta muodostavat kirjainlyhenteen F.I.R.S.T. Viisi periaatetta ovat:

1. Testien tulee olla nopeita (Fast)
   Jos testit ovat hitaita, kehittäjät eivät halua ajaa testejä usein. Jos testejä ei ajeta tarpeeksi usein, virheitä ei löydetä tarpeeksi nopeasti, jotta ne voitaisiin korjata helposti. Ajan kanssa ohjelmakoodi alkaa muuttua huonoksi ja vaikeasti ylläpidettäväksi.

2. Testien tulee olla itsenäisiä (Independent)
   Testien ei tulisi olla riippuvaisia toisistaan. Yksi testi ei saisi luoda tarvittavia olosuhteita toiselle testille. Testien tulee olla ajettavissa itsenäisesti ja missä järjestyksessä tahansa. Jos testit ovat riippuvaisia toisistaan,

ensimmäinen epäonnistuva testi aiheuttaa seuraavien testien epäonnistumien, vaikeuttaen virheiden etsintää.

3. Testien tulee olla toistettavia (Repeatable)
   Testien tulisi olla toistettavissa missä tahansa ympäristössä. Testit tulisi pystyä ajamaan tuotantoympäristössä, QA (Quality Assurance) ympäristössä tai omalla kannettavalla tietokoneella vaikkapa junassa matkalla kotiin ilman internet-yhteyttä. Jos testit eivät ole toistettavissa ympäristöstä riippumatta, on olemassa aina tekosyy sille, miksi testit epäonnistuvat. Testit jäävät myös ajamatta aina, kun testien ajamiseen vaadittava testausympäristö ei ole saatavilla.

4. Testien tulee olla itsensä validoivia (Self-Validating)
   Testeillä tulisi olla totuusarvoon perustuva lopputulos. Testit joko läpäistään tai ne epäonnistuvat. Kehittäjän ei tulisi joutua selaamaan lokitiedostoja selvittääkseen, onko testi mennyt läpi vai ei. Jos testit eivät ole itsensä validoivia, testien tulokset voivat muuttua riippuvaisiksi testejä ajavan henkilön tulkinnoista. Lisäksi testitulosten tulkitseminen voi olla todella aikaa vievää.

5. Testien tulee olla ajankohtaisia (Timely)
   Yksikkötestit tulisi kirjoittaa juuri ennen tuotantokoodin kirjoittamista, joka läpäisee juuri kirjoitetun testin. Jos yksikkötestit kirjoitetaan tuotantokoodin kirjoittamisen jälkeen, voi tuotantokoodi muuttua vaikeaksi testata. Tämän seurauksena osan tuotantokoodista saatetaan päättää olevan liian vaikeaa testattavaksi, jolloin testit jäävät kirjoittamatta eikä tuotantokoodia alkujaankaan suunnitella testattavaksi. (Martin 2008, 132.)

2.1.2 Puhdas ohjelmakoodi

Sovelluskehityksessä vähintään 80 prosenttia työstä liittyy jo kirjoitetun ohjelmakoodin ylläpitämiseen (Martin 2008, alkusanat). Sen sijaan, että odotettaisiin

ongelmien ilmaantumista ja niiden jälkikäteen korjaamista, voimme suorittaa päivittäin työssämme huoltotoimenpiteitä ja korjata ohjelman osia ennen kuin ongelmat pääsevät ilmenemään tuotannossa. Kun vanhaa ja jo olemassa olevaa ohjelmakoodia päivitetään uuteen muotoon, ovat olemassa olevat testit mainio apu tarkastellessa säilyykö ohjelman toiminnallisuus ennallaan. Ohjelmakoodia lähdetään uusimaan vähitellen ja testit ajetaan aika ajoin varmistuaksemme, ettei mitään ole mennyt rikki. Ohjelmakoodia voidaan parantaa tekemällä siitä helpommin luettavaa parantamalla funktioiden ja luokkien nimiä, pienentämällä funktioita ja luokkia, jos ne ovat sisällöiltään ja toiminnoiltaan liian laajoja tai eriyttämällä ohjelmakoodia osiin, joista jokaisella on oma tehtävänsä ja parantaa ohjelmakoodin yhtenäisyyttä. Testit tulisi olla kirjoitettu puhtaasti ja helposti luettaviksi. Testikoodin puhtaus on yhtä tärkeää kuin tuotantokoodin puhtaus. Tuotantokoodin muuttuessa, myös testien on usein muututtava. Mitä huonommin organisoitua ja vaikeammin luettavaa testikoodi on, sitä vaikeampaa ja aikaa kuluttavampaa testien muuttaminen on. Testien kirjoittamiseen saattaa kulua huomattavasti enemmän aikaa kuin itse tuotantokoodin kirjoittamiseen, aiheuttaen tuotantoaikojen pidentymistä. Testejä saatetaan alkaa pitää jopa uhkana ja testejä aletaan pitää syyllisenä tuotantoaikojen pidentymiselle.

Martin (2008, 123) mainitsee olleensa osallisena projektissa, jossa edellä mainitun kaltainen tapaus on johtanut yrityksessä testien poistamiseen kokonaan. Ilman testejä tiimi menetti mahdollisuuden reagoida ohjelmakoodin muutoksien aiheuttamiin ongelmiin nopeasti ja ohjelman viallisuus kasvoi vähitellen. Kun vikojen määrä kasvoi, kehitystiimi alkoi pelätä muutosten tekemistä ohjelmakoodiin. Kehitystiimi lopetti ohjelmakoodin siistimisen pelätessään näiden huoltotoimien aiheuttavan enemmän harmia kuin hyötyä. Ohjelmakoodin sekavuus, sovelluksen viat ja tätä myötä asiakkaiden turhautuminen kasvoi jättäen kehitystiimin tuntemaan, että sovelluksen testit olivat pettäneet heidät. Martin kertoo olleensa osallisena myös useissa projekteissa, joissa puhtaasti kirjoitetuilla testeillä ei ole ollut samankaltaisia vaikutuksia ja hän painottaa testikoodin olevan yhtä tärkeää kuin tuotantokoodinkin. Testien puhtaana ja organisoituna pitäminen pitää tuotantokoodin joustavana. Vaikka sovelluksen arkkitehtuuri olisi kuinka joustava tahansa, ilman testejä tämä joustavuus voidaan menettää.

Kattavilla testeillä kehittäjän ei tarvitse pelätä jokaisen ohjelmakoodin muutoksen rikkovan jotakin vanhaa toiminnallisuutta, tehden ohjelmakoodista joustavampaa ja muokattavampaa. Sovelluksen hyvillä arkkitehtuurisilla ratkaisuilla voidaan tehdä ohjelmakoodista testattavaa, kun taas huonoilla arkkitehtuurisilla ratkaisuilla menetetään kyky testata ohjelman kaikkia osia. (Martin 2008.)

2.1.3 Sovellusarkkitehtuuri

Mitä suuremmaksi sovellus ja koodipohja kasvaa, sitä tärkeämpää on pitää kiinni hyvistä arkkitehtuurillisista käytännöistä. Sovellusarkkitehtuurilla tarkoitetaan sovelluksen koodipohjan rakennetta, kuinka ohjelmakoodi ja sen osat ovat jaettu osiin ja kuinka nämä osat kommunikoivat keskenään. Opinnäytetyössä käsitelty arkkitehtuurin muoto on nimeltään Heksagonaalinen arkkitehtuuri, jota kutsutaan myös nimellä Ports and Adapters -arkkitehtuuri. Arkkitehtuuri pohjautuu Separation of Concerns -periaatteeseen, joka saavutetaan jakamalla sovellus kerroksiin, joilla erotetaan sovelluksen looginen toteutus ja ulkoiset elementit toisistaan. Nämä kerrokset yhdistetään toisiinsa porttien ja adaptereiden avulla. Muutokset esimerkiksi sovelluksen käyttöliittymässä tai tietokannassa eivät vaikuta sovelluksen sisempien kerrosten toimintaan. Heksagonaalista arkkitehtuuria noudattava ohjelmakoodi on helposti testattavissa, koska jokaista kerrosta voidaan testata yksittäin, ilman että muut kerrokset vaikuttavat testattavan yksikön toimintaa. (Martin 2018.)

3 Päiväkirjatyöskentely

3.1 Viikko 1 (14.-18.8.2023)

Maanantai 14.8.

Päivän tavoitteena on noutaa IT-toimistosta työkoneeni ja aloittaa asentamaan siihen tarvittavia ohjelmistoja. Lisäksi suunnitelmana oli seurata vierestä sovelluskehityksellisiä tehtäviä. Harjoitteluni perusteella tiesin, että ensimmäisellä viikolla tulee kulumaan paljon aikaa siihen, että saan pääsyn tarvittaviin järjestelmiin. Asensin päivän mittaan sellaisia ohjelmistoja, joihin sain jo pääsyn ja muuten seurasin muiden työskentelyä vierestä ja osallistuin ratkaisujen pohdintaan. Aamulla oli myös mobiilikehittäjien palaveri, jossa käytiin pikaisesti läpi ajankohtaisia asioita ja mahdollisia ratkaisuja. Eräs läpikäytävä asia liittyi Android Mobile SDK-pakettiin. Käsittelimme toiminnallisuutta, jossa erästä prosessia ei

käynnistetä uudestaan, jos aikaisempi prosessi on vielä käynnissä. Lähdimme kirjoittamaan tapaukselle testiä. Testissä täytyisi saada simuloitua käynnissä oleva prosessi, jonka aikana kutsumme funktiota uudestaan ja käynnistysprosessin ei tulisi käynnistyä uudestaan. Käytämme Android-kehitysympäristössä MockK-nimistä testauskirjastoa, jonka avulla on mahdollista kirjoittaa omaa toiminnallisuutta testiluokkiin tai funktioihin testeissä. Loimme testattavaan funktioon pienen viiveen, jonka aikana kutsuimme funktiota uudestaan. Tämä ratkaisu osoittautui toimivaksi ja ajoimme tämän jälkeen kaikki testit varmistuaksemme, että kaikki toimii edelleen. Laitteistoni toimintakuntoon saaminen ei odotetusti valmistunut tämän päivän aikana, mutta pääsin jo seuraamaan vierestä sekä Android- että iOS-tiimien työskentelyä.

Tiistai 15.8.

Pääasiallinen henkilökohtainen tavoitteeni oli tänäänkin saada pääsy tarvittaviin ohjelmistoihin, mutta tarkoitus oli osallistua työskentelyyn iOS-tiimin kanssa.

Käyttäjätunnuksieni aktivoinnin kanssa ilmeni ongelmia, koska olen kirjautunut jo viime syksyn harjoittelussani ollessani erilaisiin palveluihin ja käyttäjätunnukseni on poistettu sittemmin käytöstä. Teimme pariohjelmointina iOS-sovelluksen toiminnallisuutta, jossa tarkastellaan tiettyjen tapahtumien tilaa. Ohjelmakoodissa on tapahtumien kuuntelija, joka kerää näitä tilailmoituksia ja tilan perusteella toimitaan asiaan kuuluvalla tavalla. Kirjoitimme testejä tilan keräämiselle ja pääsin samalla hieman tutustumaan iOS-sovelluksen testaukseen.

Kuten mainittu, Android sovelluksessa on erillinen testauskirjasto, joka helpottaa testitapausten luomista, kun voidaan luoda Mock-luokkia tai funktiota, joiden toiminnallisuutta voidaan simuloida verrattain helposti. iOS-sovelluksessa tämän kaltaista kirjastoa ei ole käytössä ainakaan vielä, joten testien alustaminen vaatii melko paljon itse kirjoitettua testitoiminnallisuutta. Testeissä käytettävät luokat ja funktiot ja niiden tarjoama toiminnallisuus on kirjoitettava itse alusta loppuun. Yksikkötesteissä käytössä ei ole oikeaa lukkoa, jota testit käyttäisivät, vaan tätä varten on kirjoitettu testejä varten luokka, joka simuloi lukon toimintaa.

Testeissä luotu luokka tuotti sattumanvaraisen määrän lukon tiloja, ja lähetti ne testattavalle luokalle testeissä tarkasteltavaksi. Tämän päiväisten testien tarkoitus oli testata lukon eri tilojen käsittelyä. Saimme testit kirjoitettua ja tämän jälkeen muutokset lisättiin versionhallintaan. Vaikka laitteistoni ohjelmistojen asentaminen vei suurimman osan päivästä, pääsin silti seuraamaan vierestä iOS-tiimin toimintaa ja tutustumaan vähitellen tiimin työskentelytapoihin ja ohjelmakoodiin.

Keskiviikko 16.8.

Tavoitteeni oli tänään saada pääsy versionhallintaan ja päästä tutustumaan ohjelmakoodiin. Kloonasin iOS SDK ja mobiilisovellus -projektit versionhallinnasta ja avasin ne XCodella, mutta en juurikaan ehtinyt vielä tutustua projekteihin. Osa päivästä kului edelleen käyttäjätunnusten aktivoinnissa erilaisiin palveluihin, sekä IT-tuen kanssa yhteyttä pitäessä. Seurasin iltapäivällä hieman

vierestä iOS-tiimin työskentelyä. iOS-testeissä on tapauksia, joissa simuloitava luokka tarjoaa sattumanvaraisia luokan tiloja testejä varten. Testi poimii kaikkien mahdollisten tilojen joukosta sattumanvaraisen tilan jokaista testiä varten ja funktiota testatessa tarkastetaan lopuksi, että saatiinko funktiota kutsumalla tämä sama tila. Sattumanvaraisuudella voitaisiin testata toiminnallisuutta erilaisilla arvoilla sen sijaan, että käytettäisiin jotain tiettyä kiinteää arvoa. Jos testi ajetaan vaikkapa sata kertaa ja osa testeistä ei mene läpi, on erittäin mahdollista, että funktio ei toimi kaikilla mahdollisilla arvoilla oikein. Tänään aloin päästä enemmän työskentelyyn kiinni, kun sain pääsyn versionhallintaan. Pääsin myös tutustumaan hieman lisää iOS kehityksen käytänteisiin.

Torstai 17.8.

Tänään tarkoituksena oli kehittää lukituslaitteen ohjelmiston päivityksen mahdollistavaa toiminnallisuutta iOS SDK -pakettiin. Saatuani eilen pääsyn versionhallintaan, pääsin tänään tutkimaan iOS-sovelluksen ohjelmakoodia. Pääsin myös ensimmäistä kertaa kirjoittamaan koodia pariohjelmoinnissa. Toinen kehittäjä oli luonut uuden haaran versionhallintaan, jossa toteutetaan iOS SDK -pakettiin toiminnallisuus lukituslaitteen ohjelmiston päivitykselle. Toiminnallisuuteen sisältyy uuden version lataaminen, tallentaminen muistiin ja ohjelmiston päivittäminen. Lähdimme myös alustamaan jo testitapauksia ja luomaan uusia tyyppejä, portteja ja adaptereita. iOS-tiimillä on tapana lähteä kirjoittamaan testattavaa toiminnallisuutta testaustiedostoon testiluokan perään ja kun testit ja toiminnallisuus on kirjoitettu, siirretään kirjoitetut toiminnallisuudet oikeisiin paikkoihin ohjelmakoodissa. Homma jäi päivän päätteeksi kesken, mutta koska kehitys tapahtuu omassa haarassaan, pystyimme jättämään toteutuksen keskeneräiseksi.

Perjantai 18.8.

Tänään tarkoituksena oli jatkaa lukituslaitteen ohjelmiston päivitykseen liittyviä toiminnallisuuksia. Ryhdyimme luomaan toiminnallisuutta, joka mahdollistaa ladatun ohjelmistopäivityksen lisäämisen muistiin. Pääsin vähitellen palauttamaan mieleeni Ports & Adapters -arkkitehtuurin ideaa, kun loimme alustavaa runkoa toiminnallisuudelle testejä kirjoittaessamme. Testejä kirjoittaessamme kohtasimme haasteita ladatun tiedon tyypin kanssa ja tämän tyypin muuntamisessa ja lisäämisessä muistiin. Testissämme lisäsimme lataamamme tiedoston muistiin ja tämän jälkeen testasimme, onko muistista haettu objekti samanlainen, kuin mitä sinne lisättiin. Tyypit eivät enää vertauksessa kohdanneet, joten ryhdyimme tutkimaan, miksi näin tapahtui. Kun asian tutkimiseen alkoi kulua liikaa aikaa, päätimme keskeyttää pariohjelmoinnin ja ryhtyä selvittämään asiaa omilla tahoillamme. Ymmärrykseni ohjelmakoodin rakenteesta ja toimintatavoista lisääntyi huomattavasti päivän aikana.

Viikkoanalyysi

Ensimmäisellä viikollani kului melko paljon aikaa siihen, että sain tarvittavat ohjelmistot toimintakuntoon. Tämä oli kuitenkin odotettua ja pääsin tästä huolimatta osallistumaan paljon päivittäiseen työskentelyyn muiden kanssa. Ensimmäisen viikon aikana työskentelin enimmäkseen iOS-tiimin kanssa. Tein harjoitteluni edellisenä vuonna samassa yrityksessä ja silloin työskentelin ainoastaan Android-tiimissä, joten iOS-sovelluksen ja SDK:n kehitys on minulle täysin uutta. Testien kirjoittaminen on hieman erilaista, koska iOS-puolella ei ole ulkoisia testauskirjastoja, jonka avulla voitaisiin luoda Mock-objekteja testejä varten. Aloin viikon aikana tutustua tapoihin, kuinka iOS-testeissä luodaan Spy ja SUT -objekteja. Kun tarkoituksenamme oli tiedon muistiin lisäämiseen liittyviä toiminnallisuuksia, loimme Spy-objektin, jonka avulla kykenimme tarkastamaan, oliko lisäämämme tiedot mennyt muistiin, kun ne sinne lisätään. Testeissä luodaan testattava luokka ja tarvittavat simuloitavat riippuvuudet (SUT). Tässä vaiheessa voidaan luoda esimerkiksi Spy-objekti, joka simuloi muistin toimintaa.

Sen sijaan, että testattava luokka kutsuu oikeaa muistia, kutsuu se Spy-objektissa määriteltyä simuloitua toiminnallisuutta. Spy-objekteihin voidaan luoda funktioita tai kenttiä, joiden avulla voidaan tarkastaa esimerkiksi, montako kertaa jotakin funktiota on kutsuttu tai tässä tapauksessa sitä, lisättiinkö muistiin oikeanlaista tietoa.

Ports & Adapters -arkkitehtuuri mahdollistaa luokkien testattavuuden ja sen, että voimme syöttää testattaville luokille haluamamme kaltaisia riippuvuuksia testejä varten. Luokkien tulisi olla riippuvaisia abstraktiosta, eikä tiedosta mitä niille syötetään (Martin 2008, 149). Jokaista luokkaa varten luodaan rajapinta, joka määrittää abstraktilla tasolla sen, millaista toiminnallisuutta luokan täytyy sisältää. Tuotantokoodissa oleva luokka toteuttaa tämän rajapinnan ja tarjoaa toiminnallisuuden vaadituille funktioille. Testitapauksissa luodut testattavat riippuvuudet ja Spy-objektit laajentavat näitä samoja rajapintoja, jolloin testeissä voidaan syöttää testattaville luokille oikean tyyppisiä riippuvuuksia kirjoittaen kuitenkin Spy-objekteille testejä varten erilaisen toiminnallisuuden. Testitapauksessamme loimme uuden portin/rajapinnan, joka tarjoaa abstraktin toiminnallisuuden päivityksen tallentamiselle, sekä adapterin, joka tuottaa toiminnallisuuden portin tarjoamalle funktiolle ja on samalla testauksen kohde. Aluksi adapterin funktion runko on tyhjä, mutta funktiota voidaan kutsua testissä. Adapteri tarvitsee riippuvuutena tietokannan toiminnan tarjoaman luokan. Koska tämä riippuvuus syötetään adapterille ulkoapäin, voimme luoda muistin toiminnallisuuden määrittämää porttia vastaavan Spy-objektin ja syöttää tämän objektin adapterille. Kirjoitimme ensin testin, jossa testattavan funktion täytyy antaa virhe, jos funktion syötteen ovat epäkelpoja. Kun testi ajetaan ensimmäisen kerran, testi epäonnistuu, koska funktion runko on edelleen tyhjä. Tämän jälkeen kirjoitimme funktioon ehdon, joka tarkastaa syötteen ja heittää virheen, jos syöte ei ole kelvollinen. Tämän jälkeen testi ajettiin uudestaan ja nyt testi meni läpi.

Seuraavaksi ryhdyimme hieman parantamaan testiä ja toiminnallisuutta. Halusimme testata, että funktio heittää tietyn tyyppisen virheilmoituksen. Kirjoitimme taas testin ja tämän jälkeen toiminnallisuuden. Tällä tavoin lähdimme luomaan vähitellen uusia testitapauksia kirjoittaen vain niin vähän testiä ja toiminnallisuutta kerrallaan, kuin on tarpeen. Tämä noudattaa suoraan Martinin

mainitsemaa TDD:n kolmea sääntöä (Martin 2008, 122). Kulunut viikko on opettanut luomaan itse testattavien luokkien vaatimia riippuvuuksia, kun apuna ei ole ollut ulkoisia testauskirjastoja tai viitekehyksiä. Pääsin myös palauttamaan mieleen heksagonaalisen arkkitehtuuriin logiikkaa ja sitä, kuinka tätä toteutetaan iOS-sovelluksen ohjelmakoodissa.

Tämän viikon tavoitteissani oli saada työlaitteeni ja ohjelmistoni toimintakuntoon, että pääsen tutustumaan ohjelmakoodiin ja harrastamaan pari- ja ryhmäohjelmointia ja tässä tärkeimmiltä osin onnistuin. En usko, että olisin kuluneella viikolla voinut juurikaan tehdä asioita toisin, koska viikko meni enimmäkseen vierestä seuratessa ja tutustuessa toimintatapoihin. Pääsin myös osallistumaan pohdintaan siitä, kuinka voisimme muuttaa ohjelmakoodin toiminnallisuutta ilman, että rikomme jo käyttäjillä olevaa toiminnallisuutta ennen seuraavaa suurta päivitystä. Kävimme läpi erilaisia vaihtoehtoja emmekä päätyneet vielä tämän suhteen mihinkään toimintamalliin, vaan mietintä jatkunee seuraavalla viikolla.

3.2 Viikko 2 (21.-25.8.2023)

Maanantai 21.8.

Tämän päivän ohjelmassa oli jatkaa lukon ohjelmistopäivityksen mahdollistavan toiminnallisuuden toteuttamista iOS SDK -pakettiin. Kirjoitimme testiä, jossa lisäämme ladatun ohjelmistoversion muistiin. Tämän jälkeen lisäsimme uuden version muistiin samalla ID-numerolla, korvataksemme vanha ohjelmistoversio muistissa. Huomasimme kuitenkin, että käyttämämme muisti ei tue muistissa olevan objektin päivittämistä, vaan meidän täytyi poistaa edellinen objekti ja tämän jälkeen lisätä uusi objekti muistiin. Ryhdyimme samalla luomaan service-luokkaa, joka huolehtii sekä uuden ohjelmistoversion lataamisesta, että sen tallentamisesta muistiin. Vaikka luokka tekeekin kahta asiaa, testattavuus on edelleen helppoa, koska voimme injektoida luokalle portit sekä hakemiseen, että tallentamiseen, testaten näin molempia toiminnallisuuksia erikseen. Opin tänään

uusia asioita käyttämämme muistin toiminnasta ja iOS-tiimin käytänteistä koodin rakenteen suhteen.

Tiistai 22.8.

Tänään jatkoimme ohjelmistopäivityksen toiminnallisuuden kanssa. Tavoitteena on saada luotua suojattu bluetooth-yhteys lukituslaitteen ja puhelimen välille ja tämän jälkeen suorittaa ohjelmistopäivitys. Tätä toimenpidettä ei ole iOS-puolella vielä suoritettu, joten ryhdyimme selvittämään yrityksen dokumentaation kautta tarvittavia toimenpiteitä tämän saavuttamiseksi. Päätimme jakaa tarvittavat työvaiheet osiin ja alkaa kirjoittamaan toiminnallisuuksia testien kautta. Koska tuleva toiminnallisuus ja sen toteuttaminen eivät olleet meille entuudestaan tuttua asiaa, auttoi testien kirjoittaminen pala palalta ennen toteutuksen kirjoittamista hahmottamaan tapahtumaketjun kulkua päivitystä suorittaessa. Kävimme pikaisesti läpi myös testausympäristöä, missä voidaan suorittaa automaattisia end to end -testejä oikeiden mobiililaitteiden kanssa. Syksyn tavoitteena on kehittää tätä testausympäristöä lisää.

Keskiviikko 23.8.

Jatkoimme jälleen ohjelmistopäivityksen kanssa työskentelyä. Tutustuimme dokumentaatioon ja piirsimme kaaviota tapahtumaketjusta. Lopuksi saimme ymmärryksen tapahtumien kulusta ja jaoimme prosessin karkeasti kolmeen osaan: lukituslaitteen valmistelu yhteyttä varten, bluetooth-yhteyden määrittäminen ja lopuksi tapahtumien kuuntelu lukituslaitteelta päivitystä suorittaessa. Kohtasimme ongelmia, kun yritimme luoda testiluokkaa, joka imitoisi lukituslaitetta. Lukituslaitetta imitoivan objektin täytyy toteuttaa protokollaa, joka ei salli instanssien luomista sitä toteuttavilta luokilta. Lyhyen selvittelyn jälkeen jouduimme luomaan Objective-C-ohjelmointikielellä ObjectBuilder-apuluokan, jonka avulla voisimme luoda Mock-objekteja sellaisista luokista, jotka eivät salli

instanssien luomista. Päivän päätteeksi loimme spike-haaran versionhallintaan ja lisäsimme muutokset sinne, koska toiminnallisuuden luominen jäi kesken. Päivän tavoitteena oli lisätä ymmärrystä ohjelmistopäivityksen toteuttamisesta ja pyrkiä jatkamaan toteutuksen luomista. Ymmärrykseni aiheesta lisääntyi merkittävästi ja pääsimme myös jatkamaan työskentelyä.

Torstai 24.8.

Päivän tavoitteena oli saada lisättyä toiminnallisuutta ohjelmistopäivitykseen. Meidän täytyi testata bluetooth-datan liikkumista mobiililaitteen ja lukituslaitteen välillä. Tässä kohtaa testaaminen alkoi muuttua hieman haasteelliseksi, kun tarvitsimme Mockin, joka antaisi meille arvoja lukon tilasta päivityksen aikana. Mockin täytyisi toteuttaa samaa protokollaa kuin tuotantokoodinkin, mutta protokolla ei salli sisäisen datansa muokkaamista, vaan tietoa voi vain lukea. Tutustumalla lisää Applen Core Bluetooth -viitekehykseen, löysin muokattavan version Core Bluetoothin CBService-luokasta, jota voisimme käyttää testeissä. Core Bluetooth vaatii käyttämään delegate-objektia, joka mahdollistaa keskustelun bluetooth-yhteydellä laitteiden välillä. Jättäydyin pois ryhmäohjelmoinnista iltapäivällä ja ryhdyin tutustumaan delegate-malliin saadakseni paremman ymmärryksen siitä, missä järjestyksessä bluetooth-keskustelun funktiokutsut tapahtuvat.

Perjantai 25.8.

Päivän tavoitteena oli parantaa ohjelmakoodin rakennetta selkeämmäksi, joka parantaisi myös koodin testattavuutta. Eilen kirjoittamamme ohjelmakoodi osoittautui verrattain hankalaksi testata, koska jouduimme alustamaan liikaa asioita testejä varten ja luokat tulivat liian riippuvaisiksi toisistaan. Emme voineet testata varsinaista testattavaa luokkaamme ilman, että alustaisimme myös toisen tarvittavan luokan toiminnan testeissämme. Jos toinen luokka muuttuisi, voisi se

hajottaa testit, jolla testaamme toista luokkaa. Loimme näiden kahden luokan välille portin ja adapterin, jonka kanssa luokat kommunikoivat. Saimme näin erotettua luokat toisistaan ja testattua molempia yksittäin. Tutustuin myös Swiftin Continuation-funktioihin ja opin, kuinka voimme luoda asynkronisen funktion sellaisesta funktiosta, joka ei ole asynkroninen. Kutsumme Core Bluetoothin funktiota, joka käynnistää keskustelun oheislaitteen kanssa. Kun lukituslaite vastaa, se kutsuu delegate-objektimme funktiota, joka saa parametreinaan lukituslaitteen lähettämät arvot. Kun tätä funktiota kutsutaan, palautamme continuationilla arvot, jotka saamme delegaten funktion parametrina.

Viikkoraportti

Tällä viikolla pääsin työskentelemään iOS-tiimin kanssa täysipäiväisesti. Tavoitteenamme oli tälläkin viikolla jatkaa ohjelmistopäivityksen mahdollistavan toiminnallisuuden toteuttamista. Toiminnallisuuden toteuttaminen on ollut melko haasteellista ymmärtää ja ryhdyinkin käyttämään aikaa delegate-mallin opiskeluun Swift-ohjelmointikielessä. Delegate mahdollistaa tiedon välittämisen, delegoinnin, kahden objektin välillä, joista toinen on lähde ja toinen delegate. Delegate-malli mahdollistaa monesta yhteen kommunikoinnin objektien välillä. Delegatella voi olla useampi lähde, mutta lähteellä voi olla vain yksi delegate. (Vilmart, Scalzo & De Simone 2018). Apple-SDK ja käyttämämme Core Bluetooth viitekehys käyttää edelleen delegate-objekteja tiedon välitykseen ohjelman ja ohjeislaitteen välillä. iOS SDK -paketissa pyrimme tarjoamaan sen käyttäjille nykyaikaisia asynkronisia async-await funktiota, joita kutsumalla he voivat kommunikoida oheislaitteen kanssa. Core Bluetooth ei kuitenkaan tarjoa vielä async-await funktioita, joilla kommunikoida oheislaitteen kanssa. Käytimme tähän apuna Swiftin tarjoamaa Continuation-tyyppiä, jonka avulla voidaan muuttaa ulkoisen API:n tarjoamat funktiot async-await muotoon. (Giordano & Edgar 2021.) Continuation mahdollistaa myös sen sisällä olevan funktion palauttamisen milloin tahansa, joten pystyimme palauttamaan continuationin sen jälkeen, kun Core Bluetooth kutsuu delegaten funktiota muuttuneilla arvoilla.

Kun lähetämme tietoa oheislaitteelle, laite kutsuu tiedon käsiteltyään delegate-objektia päivittyneillä tiedoilla. Delegaten avulla saamme nämä tiedot omaan ohjelmaamme, jonka jälkeen voimme palauttaa continuationin. Pystyimme testaamaan asynkronisia funktioita käyttämällä avuksi XCTest-viitekehyksen tarjoamaa XCTestExpectation-toiminnallisuutta. Tämän avulla loimme luokasta periytyvän odotteen, jonka on täytyttävä määräämässämme ajassa, tai testi ei mene läpi. Loimme lukituslaitteestamme Spy-objektin, jolla lähetimme delegate-objektille tietoa ja näin simuloimme lukitusaitteen lähettämän kutsun. Lisäksi lisäsimme Spy-objektille pisteen, jossa odote täyttyy ja testin suoritus saa jatkua, jos olemme päässeet sinne asti. Jos asynkroninen funktiomme toimisi oikein ja pääsemme tähän pisteeseen, toimisi funktiomme odotetusti. Tapauksessamme oheislaitteemme voi välittää meille onnistuneessa tapauksessa tietoa tilastaan, tai heittää virheen, jos jotakin meni vikaan. Molemmissa tapauksissa voimme käyttää samaa tapaa funktion testaamiseen. Molempia lopputuloksia varten on luotava omat testitapauksensa; yksi testi testaa virhetilannetta ja toinen testi testaa onnistunutta lopputulosta ja sitä, palauttaako funktiomme oikeanlaista tietoa. Opin viikon aikana paljon uutta delegate-mallista. Delegate-malli on eräs tapa välittää tietoa toisen objektin tilasta toiselle objektille ja tapauksessamme tilan muutos veisi aikaa. Meidän oli löydettävä tapa muuttaa tämä toiminnallisuus Swiftin tarjoamaan async-await muotoon, joka opetti, kuinka voidaan luoda asynkronisia funktioita sellaisista funktioista, joiden valmistumisesta emme saa muuten tietoa ohjelmassamme. Opin myös testaamaan asynkronisia funktioita XCTestExpectation-luokkaa hyväksi käyttäen.

3.3 Viikko 3 (28.8.-1.9.2023)

Maanantai 28.8.

Aamu alkoi kahdella palaverilla. Ensimmäisessä käsiteltiin isompaa kuvaa tuotteesta ja tällä hetkellä tärkeistä asioista. Tämän jälkeen pidimme mobiilitiimien yhteisen palaverin. Palaverin aiheena oli pipeline-tapahtumien yhtenäistäminen eri projektien välillä. Voisimme luoda yhteisen mallin esimerkiksi Android- ja iOS-sovellusten tietyille prosesseille. Näistä pipeline-malleista voitaisiin tehdä

versionumeroituja sen sijaan, että olisi vain yksi pipeline-versio, jota muutetaan tarvittaessa. Näin tietyn projektin voi määrittää käyttämään tiettyä pipeline-versiota, joka helpottaa muutosten tekemistä. Loppupäivän työskentelin iOS-tiimin kanssa. Muutimme toteutustamme lokien kirjaukseen liittyen iOS SDK -paketissa. Muutimme erään luokkamme toteutusta siten, että voimme injektoida sille testeissämme tarvitsemiamme asioita. Tämä auttoi suuresti luokan toteuttamien lopputulosten testaamisessa.

Tiistai 29.8.

Lähetimme eilen iOS-sovelluspäivityksen Applen App Store Connectiin tarkastukseen ennen julkaisua. Otimme uudessa sovellusversiossa käyttöön päivitetyn iOS SDK -paketin, ja tarkoitus oli päivittää sovellus App Storeen. Huomasimme kuitenkin aamulla, ettei tarkastus ollut mennyt läpi. Sovellus ei ollut toiminut oikein eräällä laitteella tietyllä ohjelmistoversiolla. Testasimme sovellusta simulaattoreilla, mutta emme kyenneet toistamaan ongelmaa. Raportoimme takaisin Applelle, ettemme kykene toistamaan ongelmaa simulaattoreilla emmekä kykene testaamaan asiaa laitteella, jonka kanssa he kohtasivat ongelmia. Pyysimme heitä myös hyväksymään lokien keräämisen, jotta voisimme saada mahdollisista virhetilanteista analytiikkaa. Tämän jälkeen jatkoimme edelleen lukituslaitteen ohjelmistopäivityksen kanssa työskentelyä.

Kirjoitimme testejä funktioille, jotka palauttavat continuationin, kun olemme saaneet lukituslaitteelta dataa. Oli tärkeää testata, että alustamme continuationin kerran silloin, kun lähetämme kutsun lukituslaitteelle ja asettaa continuation arvoon nil, kun continuation on palauttanut lukituslaitteelta saamamme datan muistivuotojen välttämiseksi. Testasimme continuation-objektin aloitusarvoa sen jälkeen, kun arvo tulisi asettaa ensimmäisen kerran ja lopuksi sitä, että funktioiden suorittamisen jälkeen arvo on taas nil. Tutustuin myös iltapäivästä tulevan end to end -testausympäristön ohjelmakoodiin. Nykyinen toteutus hakee testattavat elementit niiden sisältämän tekstin perusteella. Tulevaisuudessa pyrimme lisäämään jokaiselle testattavalle elementille yksilöivän tunnisteen, jonka avulla

löytäisimme ne testeissä. Sovelluksen sisältämät tekstit saattavat muuttua hajottaen myös testit, koska elementtejä ei enää testeissä löydy. Tämän vuoksi elementtien paikantaminen yksilöivällä tunnisteella tekee testeistä vähemmän alttiita muutosten aiheuttamille ongelmille.

Keskiviikko 30.8.

Tänään oli tarkoitus saada testilaitteeni käyttökuntoon, jotka ovat puhelin, lukko ja miniboard. Miniboard on piirilevy, jossa on lukon elektroniikka ja johon voi asentaa saman laiteohjelmiston kuin oikeaan lukkoonkin. Sitä voidaan ohjata samalla mobiilisovelluksella kuin lukkoakin. Ohjelmaa kehitettäessä välillä lukituslaite on tarpeen tyhjentää ja asentaa tarvittavia tietoja uudestaan. Varsinaiselle lukituslaitteelle näitä operaatioita ei voi tehdä laisinkaan, joten tähän tarvitsemme miniboardia. Tutustuimme myös viitekehykseen, joka automatisoi monia toistuvasti suoritettavia asioita lukon käyttöön ottamisessa kehitysympäristössä. Iltapäivällä jatkoimme iOS-tiimin kanssa hieman lukituslaitteen ohjelmistopäivityksen kanssa. Muu tiimi oli tehnyt toiminnallisuutta, jolla määrätään bluetooth-yhteyden MTU (maximum transfer unit), joka määrittää sen, kuinka isoissa paketeissa bluetooth siirtää tietoa. Puhelin määrittelee oman MTU-arvonsa ja oheislaite määrittelee omansa. Käyttämämme MTU-arvo on näistä pienempi arvo, koska molemmat laitteet kykenevät sen kokoisiin siirtoihin. Jos pakettia yritettäisiin siirtää isompana kuin mihin toinen laitteista kykenee, ei datan siirtäminen onnistuisi ollenkaan.

Torstai 31.8.

Päivän tavoitteena oli saada miniboardini toimimaan ja opiskella Swift-ohjelmointikieltä. Miniboardin käyttöön ottaminen vaati useita erilaisia toimenpiteitä, joista suurin osa oli automatisoitu erään viitekehyksen toimesta. Harjoittelussa ollessani suurin osa toimenpiteistä täytyi tehdä manuaalisesti, enkä enää

muistanut millaisia toimenpiteitä lukituslaitteen käyttöönottamisen prosessi vaati. Operaatio toimi hyvänä muistin virkistyksenä siitä, kuinka lukituslaite aktivoidaan käyttöön. Loppupäivän opiskelin yleisesti Swift-ohjelmointikieltä ja kirjoitin omia toteutuksia testivetoisesti saadakseni tuntumaa ja rutiinia usein toistettaviin kaavoihin.

Perjantai 1.10.

Ryhdyin tänään ottamaan käyttöön ja tutustumaan end to end -testausympäristöön. Suurin osa päivästä meni pystyttäessä projektia ja päivittäessä dokumentaatiota sitä mukaan, kun ongelmia ilmeni ja niihin ratkaisuja löytyi. Testausympäristö on vielä keskeneräinen projekti, eikä se ole vielä aktiivisessa käytössä ja kehitystyötä tullaan tekemään syksyn mittaan enemmänkin. Toistaiseksi testien ajaminen vaatii verrattain paljon erilaisten skriptien ajamista manuaalisesti ja tietojen syöttämistä käsin sekä skripteihin, että UI-testien toimintaa tarkkailevaan Appium Inspector -sovellukseen. Saimme projektin päällisin puolin asennettua työkaverini kanssa ja sain testit pyörimään testipuhelimessani. Testien ajamisen kanssa ilmeni kuitenkin ongelmia, joita täytyy tutkia tulevaisuudessa. Testiympäristö ei ole vielä kovin vakaa, koska tietoa on kovakoodattava tässä vaiheessa melko paljon. Väärän tiedon syöttäminen missä tahansa prosessin vaiheessa voi aiheuttaa testien epäonnistumisen. Pääsin kuitenkin jo hieman tutustumaan testausympäristön toimintaan ja siihen, millaisia toimenpiteitä vaaditaan projektin pystyttämiseen.

Viikkoanalyysi

Tällä viikolla pääsin vähitellen tutustumaan tulevaan end to end -testausympäristöön. Mobiilisovellusten end to end -testien toteuttaminen ei ole entuudestaan itselleni lainkaan tuttua, joten testaamiseen käytettävät työkalut ja viitekehykset vaativat opiskelua. Tämän viikon tavoitteena olikin alkaa tutustua tähän

testausympäristöön sekä opiskella Swift-ohjelmointikieltä. Testausympäristö on toteutettu yritykselle kolmannen osapuolen toimesta ja projekti kykenee toistaiseksi lähinnä käynnistämään end to end -testausprosessin. Testejä ei ole vielä juurikaan kirjoitettu ja testit ovat toistaiseksi melko helppoja rikkoutumaan. Testausympäristön toiminnan ytimessä on Appium-työkalu. Appium on avoimen lähdekoodin projekti, jolla voidaan automatisoida natiivi, hybridi sekä web-pohjaisten sovellusten käyttöä ja testausta. Appiumia käytetään tyypillisesti iOS- sekä Android-sovellusten toiminnan automatisointiin ja Appium käyttää sisäisesti XCUITest-viitekehystä iOS-sovellusten testaamiseen ja UIAutomator-viitekehystä Android-sovellusten testaamiseen. Appiumin selkeänä etuna on järjestelmäriippumaton yhteensopivuus, joten testejä voidaan kirjoittaa usealle eri käyttöjärjestelmälle käyttäen vain yhtä API:a. Tämä mahdollistaa test platform -projektin konfiguroinnin yhteen projektiin ja samoja testejä voidaan ajaa sekä iOS-, että Android-sovelluksille. (Manikandan 2023.) Projektin pystyttäminen omalle koneelleni ensimmäistä kertaa vei melko paljon aikaa, koska määritettäviä asioita oli paljon ja dokumentaatiot eivät kertoneet aivan kaikkia tehtäviä askeleita. Kuluneella viikolla käytin aikaa myös henkilökohtaisten lukituslaitteideni määrittämiseen sekä opiskeluun. Opiskelin yleisesti Swift-ohjelmointikieltä sekä kirjoitin omia toteutuksia testivetoisesti saadakseni rutiinia ja tuntumaa testien kirjoittamiseen. Yksikkötesteissä toistuu usein sama kaava. Yksikkötesteissä testataan vain yhden yksikön tuottamia lopputuloksia, eikä toimenpiteitä siitä, kuinka tähän lopputulokseen päästiin (Khorikov 2020). Muut yksiköt on erotettava testattavasta yksiköstä, jotta testattava yksikkö ei ole riippuvainen minkään toisen yksikön tietynlaisesta toteutuksesta.

Pohjustin testiluokan, jonka testattava yksikkö olisi adapteri, jonka tehtävä on suorittaa API-kutsu ja vastaanottaa dataa tai heittää virhe, jos haku jostakin syystä epäonnistuu. Loin Ports & Adapters -arkkitehtuurin mukaisen portin, joka tarjoaa funktion datan hakemiselle. Testattava yksikkö saa konstruktorissa parametrina Spy-objektin, joka toteuttaa tämän portin toiminnallisuuden. Koska testissä testataan vain testattavan yksikön toimintaa, voidaan testejä varten syöttää yksikölle itse luotu datan hakemista simuloiva Spy-objekti. Koska Spy toteuttaa portin, on sen toimittava saman kaltaisesti, kuin tuotantokoodinkin

luokan, joka toteuttaa samaa porttia. Riippuvuuksien syöttämistä konstruktorissa kutsutaan riippuvuuksien injektoinniksi. Riippuvuuksien injektointi siirtää luokan vastuita ja toiminnallisuuksia toisten luokkien tehtäväksi, joiden vastuita ovat nämä toiminnallisuudet ovat. (Martin 2008, 157.) Esimerkiksi datan hakeminen on ulkoistettu adapterissa toiselle luokalle, jonka tehtävä on hakea dataa ja palauttaa se adapterille käsiteltäväksi. Viikon aikana tavat, joilla Ports & Adapters -arkkitehtuuria noudatetaan iOS-puolella, tulivat paremmin tutuiksi. Sain myös perustason tietämystä end to end -testausympäristöstä, jonka kanssa tulen toimimaan enemmänkin syksyn mittaan.

3.4 Viikko 4 (4.-8.9.2023)

Maanantai 4.9.

Aamupalaverissa puheenaiheena oli eräs ohjelmakoodin analysointityökalu. Tämä työkalu toimii osana pipelinea ja sen tehtävä on tarkkailla muun muassa testauskattavuutta ja mahdollisia huonoja käytänteitä kirjoitetussa ohjelmakoodissa. Pipeline on prosessi erilaisia automatisoituja toimintoja, jotka vastaavat esimerkiksi testien ajamisesta ja tietyissä määrin ohjelmakoodin rakeenteen oikeellisuuden tarkastamisesta. Vääränlainen rakenne on esimerkiksi liian pitkät koodirivit ilman, että sisältöä on jaettu useammille riveille koodin lukemisen helpottamiseksi tai ohjelmakoodi sisältää ylimääräisiä rivinvaihtoja. Osana testausautomaatioprosessia ajetaan myös arkkitehtuuritestit, jotka vastaavat heksagonaalisen arkkitehtuurin sääntöjen noudattamisesta, joskin arkkitehtuuritestit ovat käytössä toistaiseksi ainoastaan Android-puolella. Pipeline käynnistyy automaattisesti joka kerta, kun ohjelmakoodia pusketaan versionhallintaan. Projektien testauskattavuus olisi tulevaisuudessa tarkoitus saada nostettua 80 prosenttiin eikä pipeline menisi läpi, jos analysointityökalu havaitsee pienemmän testauskattavuuden.

Tiistai 5.9.

Jatkoimme ohjelmistopäivityksen kehityksen kanssa aamupäivällä. Ryhdyimme luomaan toiminnallisuutta, jossa valmistelemme ohjelmistopäivityksen siirtämistä bluetooth-yhteyden yli lukituslaitteelle. Työstettävän toiminnallisuuden toteuttamisen yksityiskohdat olivat melko haasteellisia ymmärtää, koska toiminnallisuuden pohjustaminen vaati työtä, enkä ole aiemmin siirtänyt dataa bluetooth-yhteyden ylitse. Toimin koodin kirjoittajana ryhmäohjelmoitaessa ja sainkin paljon ohjausta ja opetusta. Aion myöhemmin käydä läpi tapahtumaketjua saadakseni vielä selkeämpää ymmärrystä tapahtumien kulusta. Iltapäivällä osallistuin opinnäytetyön ryhmäohjausluennolle ja sain hyviä vinkkejä päiväkirjatyöskentelyyn ja oppimisen reflektointiin.

Keskiviikko 6.9.

Päätimme ryhtyä refaktoroimaan ohjelmakoodiamme hyödyntäen Swift-ohjelmointikielen continuation-ominaisuutta. Continuationin avulla voimme toteuttaa asynkronisen toiminnallisuuden ja odottaa jonkin asian valmistumista ohjelmakoodissamme. Voimme luoda continuationin ja palauttaa sen ohjelmassamme haluamassamme kohdassa. Continuationin palatessa tiedämme, että operaatio on valmistunut. Continuationin avulla voidaan myös palauttaa operaatiosta saatua dataa tai mahdollinen virhe. Nykyinen toteutus teki testaamisesta verrattain monimutkaista ja testeihin alkoi kertyä melko paljon pakollista testien alustamista. Loimme oman protokollan geneerisillä tyypeillä, jonka avulla voimme luoda eri datatyyppejä palauttavia continuationeita. Lisäsimme testattavaan adapteriimme kyvykkyyden ottaa vastaan injektoitava funktio, joka tarjoaa continuation-ominaisuuden testejä varten. Tämän seurauksena pystymme luomaan spy-luokassamme kulloinkin tarvittavan protokollan tarjoamiin funktioihin palautettavat continuationit. Aikaisemmin meidän oli luotava XCTest-viitekehyksen tarjoamia expectation-tiloja, joiden avulla kykenimme lopulta testaamaan, olimmeko koodissamme päässeet johonkin tiettyyn tilaan. Uusi toteutus mahdollisti

täysin näiden tilojen poistamisen ja voimme simuloida paremmin tuotantokoodin toimintaa, selkeyttäen myös testejä huomattavasti.

Torstai 7.9.

Suunnitelmani oli käyttää päivä opiskeluun ja tutustua syvällisemmin ohjelmakoodin rakenteeseen erityisesti lukituslaitteen ohjelmistopäivityksen toiminnallisuuden osalta. Pelkkä ohjelmakoodin selaaminen ei antanut kovinkaan tarkkaan kuvaa kokonaisuudesta, joten päätin luoda tarvittavista porteista ja adaptereista funktioineen kaavion. Kaavio auttoi hahmottamaan kokonaiskuvaa paremmin ja pystyin näkemään funktioiden kutsumisjärjestyksen yhdellä sivulla sen sijaan, että selaisin eri tiedostojen välillä edes takaisin. Tämän jälkeen lähdin toteuttamaan omaa toteutustani seuraavasta tehtävästä asiasta, jota voisin verrata varsinaiseen valmiiseen lopputulokseen myöhemmin, kunhan olemme sen saaneet tehtyä. Lähdin tekemään toteutusta normaalisti testien kautta, joka osaltaan auttoi taas ymmärtämään hieman paremmin funktiokutsujen järjestystä ja mitä portteja tulisi toteuttaa missäkin vaiheessa.

Perjantai 8.9.

Totesimme lukituslaitteen ohjelmistopäivityksen toteutusta tehdessämme, että erillinen versionhallinnan haaramme alkaa paisua liian suureksi, joten päätimme alkaa purkaa prosessia pienempiin osiin. Sen sijaan, että koko ohjelmistopäivitys olisi yksi suuri haara, joka lopulta liitettäisiin versionhallinnan main-haaraan, teemme toteutusta tulevaisuudessa pienemmissä osakokonaisuuksissa. Esimerkiksi itse ohjelmistopäivityksen siirtäminen jaettiin kolmeen osaan. Yksi kokonaisuus olisi prosessin valmistelu. Toisessa osassa hoitaisimme itse tiedon siirtämisen bluetoothin yli ja viimeisessä osassa käsittelisimme siirron jälkeen tapahtuvat asiat. Kykenemme näin rakentamaan toiminnallisuutta eteenpäin pienemmissä selkeissä osissa.

Viikkoanalyysi

Viikon tavoitteena oli käyttää enemmän aikaa asioiden itsenäiseen opiskeluun kuin aikaisemmilla viikoilla. Koodin kirjoittamisen nopean tahdin takia ei asioiden syvällisempään ja sisäistämiseen jää juurikaan aikaa, joten pyrin tutustumaan enemmän jo toteutettuihin asioihin sekä yrittää alkaa muodostaa osittaisia ratkaisuja tuleviin asioihin. Varsinkin ohjelmoitaessa jotakin asiaa, on erilaisten luokkien, protokollien ja niiden funktioiden yhteyden ymmärtäminen kohtalaisen hidasta, koska nämä luokat ja protokollat sijaitsevat eri tiedostoissa ja kokonaiskuvan hahmottamisesta tulee haasteellista. Päätin alkaa luomaan itselleni kaaviota sitä mukaa kun selvitin ohjelmakoodista tapahtumien kulkua. Suuren kokonaisuuden sijaan otin kohteekseni vain yhden toiminnallisuuden, jonka etenemistä seurasin ohjelmakoodissa ja kirjasin ylös portit, protokollat ja adapterit, joita toiminnallisuuden toteuttamisessa käytetään. Tämän jälkeen pyrin kirjoittamaan itse testivetoisesti toiminnallisuutta, jonka tiesin olevan ohjelmassa seuraavana päivänä. Huomasin seuraavana päivänä, että ymmärrykseni koodista oli ilmeisesti kasvanut, koska mielestäni kykenin ottamaan hieman enemmän osaa keskusteluun, kun tutkimme mahdollisia vaihtoehtoja toteutukselle.

Opin viikon aikana myös testien tärkeydestä ohjelmakoodin refaktorointia suoritettaessa. Aloitimme refaktoroida aiemmin kirjoittamaamme koodia parempaan muotoon. Koska aiemmin kirjoittamamme testit testaavat toteutuksen yksityiskohtien sijaan yksikön tuottamaa lopputulosta, pystyimme aloittamaan turvallisesti ohjelmakoodin muokkaamisen. Muutimme ohjelmakoodin rakennetta pienissä osissa ja aika ajoin ajoimme yksikkötestit todetaksemme, ettei mitään mennyt rikki. Jos yksi tai useampi testi ei mennyt läpi, jatkoimme refaktorointia kunnes testit läpäistiin taas onnistuneesti. Uusi toteutuksemme vaati hieman joidenkin testien muuttamista, mutta koska testit muutettiin ennen kuin toteutimme testattavan asian, pystyimme turvallisesti jatkamaan refaktorointiprosessia. Lisäksi loimme uuden protokollan continuationeiden luomiselle tyypin mukaan, jonka avulla voimme luoda uusia continuationeita yhden protokollan pohjalta sen sijaan, että toistaisimme samaa ohjelmakoodia uudestaan ja uudestaan jokaista continuationia luodessa. Testien takia, voimme pitää ohjelmakoodin ja luokat puhtaina ja rakenteeltaan parempana. Refaktoroinnin aikana voimme

esimerkiksi lisätä järjestelmän yhtenäisyyttä, vähentää luokkien riippuvuutta toisistaan, jakaa ohjelmakoodia pienempiin funktioihin ja luokkiin tai selkeyttää luokkien, muuttujien ja funktioiden nimiä. (Martin 2008, 172.) Tämä seuraa Kent Beckin määritelmää yksinkertaisesta suunnittelusta. Määritelmän mukaan ohjelman tulee kyetä ajamaan kaikki testit, ohjelmakoodi ei toista samaa koodia, ohjelmakoodi ilmaisee koodin kirjoittajan tarkoitusta sekä minimoi tarvittavien luokkien ja funktioiden määrän (Beck 2003).

3.5 Viikko 5 (11.-15.9.2023)

Maanantai 11.9.

Otimme käsittelyyn jo aiemmin ilmenneen tapauksen, josta oli luotu issue versionhallintaan. Asia liittyi eräältä integraattorilta tietoon tulleeseen tapaukseen. Saimme tarvittavat muutokset nopeasti tehtyä ja testasimme yksikkötestien lisäksi manuaalisesti, että asia oli tosiaan korjaantunut. Huomasimme yksikkötesteissä ajoittaista testien epäonnistumista, kun ajoimme saman testin 100–1000 kertaa. Näissä tilanteissa Spy-objektimme ei poistunut muistista, vaikka olisi pitänyt. Tutkimme asiaa ja totesimme tämän olevan ongelma vain testiympäristössä asynkronisten testien vuoksi, eikä tuotannossa ongelmaa olisi.

Tiistai 12.9.

Kahdelle seuraavalle päivälle oli ohjelmassa kokoontua yrityksen sisäisten sidosryhmien henkilöstön kanssa ja keskustella ajankohtaisista ja tulevista asioista sekä yrityksen isommasta kuvasta. Toimme esiin esimerkiksi kysymyksiä, huolia ja nykytilannetta ja tämän jälkeen mietimme yksittäin ja ryhmissä mahdollisia ratkaisuja ja ideoita. Käsitellyt aiheet olivat itselleni suurimmaksi osaksi tuntemattomia ja tapahtuma olikin hyvä mahdollisuus kysyä lisää epäselvistä asioista. Tapahtuma auttoi myös valottamaan isompaa kuvaa tuoteperheen tulevaisuuden suunnitelmista ja asioiden yhteyksistä toisiinsa. Eri sidosryhmien

edustajien kasvotusten kokoontuessa keskusteltavaa riitti paljon ja ideoita nousi paljon esiin. Näiden keskustelujen kuunteleminen oli hyvin opettavaista, kun kuulee useita erilaisia ratkaisuehdotelmia samaan ongelmaan.

Keskiviikko 13.9.

Jatkoimme tänään kokoontumista sidosryhmien kanssa. Edellisenä päivänä nostimme yhdessä esiin muutaman erilaisen asian, joita haluaisimme kehittää. Ideoita tuli paljon ja äänestyksen perusteella rajasimme näiden asioiden joukosta muutaman sellaisen asian, jotka koimme tärkeimmiksi tällä hetkellä. Tämän päivän tavoitteena oli alkaa keksiä ryhmissä ideoita näiden asioiden kehittämiseen. Ryhmämme tarkoituksena oli miettiä yrityksen päivittäisiä toimintatapoja ja sitä, voisimmeko kehittää toimintatapoja tehokkaammiksi. Kun rakensimme kokonaiskuvaa esimerkiksi kaikista kokouksista, joihin erilaisia sidosryhmiä osallistuu, totesimme, että kokouksien määrää olisi mahdollista vähentää, jos jotkin kokoukset yhdistettäisiin ja niihin kutsuttaisiin jonkin toisen sidosryhmän edustaja mukaan. Huomasimme myös jonkin verran parannettavaa dokumentaatioissamme ja otimme tehtäväksemme alkaa kehittää dokumentaatiota.

Torstai 14.9.

Huomasimme, että voisimme refaktoroida koodiamme lukituslaitteen ohjelmistopäivityksen toteutuksessa. Jotkin arvot eivät muutu päivitysprosessin aikana tai meidän tarvitsee vain aika ajoin päivittää niitä, eikä meidän tarvitsisi tarjota näitä muuttujia joidenkin funktioiden parametreina. Sen sijaan määritämme nämä arvot prosessin alussa adapterin sisäisinä muuttujina ja tarkastamme ennen niiden käyttöä, että arvot ovat olemassa ja päivitämme arvot tarvittaessa delegatelta saaduilla tiedoilla. Toteutamme mahdollisesti lähitulevaisuudessa erillisen service-luokan, jolloin joudumme mahdollisesti injektoimaan adaptereita tälle service-luokalle, jotta pääsemme käsiksi näihin muuttujiin. Sain lisättyä rutiinia

testien kirjoittamiseen, koska tämän päivän testien kirjoittaminen noudatti hyvin pitkälti samaa kaavaa kuin edellisen toiminnallisuuden testien kirjoittaminen.

Perjantai 15.9.

Ryhdyimme toteuttamaan jo tänään erillistä service-luokkaa ohjelmistopäivityksen lukituslaitteelle siirtämiselle. Tutkittuamme asiaa totesimme, että tulevan service-luokkamme täytyy vastata useasta erilaisesta toiminnallisuudesta. Luokan täytyy olla vastuussa ainakin siirrettävän paketin alustamisesta, sen siirtämisestä lukituslaitteelle ja lopulta lähetyksen viimeistelystä. Tämä prosessi tulisi toistumaan useita kertoja ohjelmistopäivityksen siirron aikana. Emme voi siirtää koko ohjelmistopäivitystä kerralla bluetoothin yli, vaan meidän on siirrettävä ohjelmistopäivitys pienissä paketeissa. Jokaiselle paketille on suoritettava alustaminen, siirtäminen ja lopulta viimeistely. Kun kaikki paketit on siirretty, siirrymme seuraavaan vaiheeseen, joka tulee olemaan koko siirtoprosessin viimeistely. Koska päätimme pilkkoa prosessimme pieniin osiin, keskityimme tänään vain ohjelmistopäivityksen siirtämisen prosessiin.

Viikkoanalyysi

Tiistaina ja keskiviikkona järjestettyjen sidosryhmien kokoontumisten takia, oli tällä viikolla vain kolme päivää, joiden aikana suoritettiin varsinaista ohjelmistokehitystä. Tällä viikolla tavoitteeni oli tutustua lähemmin asynkronisten testien luomiseen ja muistinhallintaan Swift-ohjelmointikielessä. Toteutimme viikon aikana paljon asynkronisia toiminnallisuuksia ja huomasimme testejä ajaessamme ajoittaisia testien epäonnistumisia. Asynkroninen funktio sisältää prosessin, jonka valmistumisessa kestää ennalta määrittelemättömän verran aikaa. Kun testattava luokkamme kutsuu sille injektoidun portin asynkronista funktiota, syntyy kutsuvan ja kutsuttavan luokan välille viittaus. Swift suorittaa muistinhallintaa pitämällä laskuria viittauksien määrästä objekteihin muistissa. Kun

laskurin viittauksien määrä on nolla ja tähän objektiin ei ole enää viittauksia, objekti vapautetaan muistista. (Vilmart, Scalzo & De Simone 2018.) Tilannetta, jossa objektiin ei ole enää viittauksia, mutta laskurin lukema on enemmän kuin nolla, kutsutaan muistivuodoksi. Tämän kaltaisessa tilanteessa objektille ei ole enää käyttöä, mutta sitä ei ole vapautettu muistista. Jokainen testiluokkamme sisältää testin, jossa jokaisen testin päätteeksi tarkastetaan, että sekä testattava luokka, että Spy-objekti on vapautettu muistista. Testeissämme aiheutui ajoittain virheitä, joiden mukaan Spy-objektia ei ollut vapautettu muistista.

Ryhdyin selvittämään, oliko ongelma olemassa vain testeissämme, vai voisiko tuotantokoodimme mahdollisesti aiheuttaa muistivuotoja. Muistivuotojen välttämiseksi tulisi asynkronisissa operaatioissa luoda kahden objektin välille heikko viittaus. Heikon viittauksen voi luoda käyttämällä Swiftin tarjoamaa avainsanaa ‘weak self’, jossa self viittaa testitapauksessa testattavaan yksikköön. Kun viittaus on heikko, tarkastaa Swift automaattisesti, onko testattava luokkamme enää muistissa. Jos luokka ei ole enää muistissa, poistetaan viittaus myös Spy-objektistamme. Jos heikkoa viittausta ei luoda, voi testattava yksikkömme poistua muistista Spy-objektin asynkronisen operaation aikana, jättäen kuitenkin viittauksen Spy-objektista testattavaan yksikköömme, aiheuttaen muistivuodon. XCode mahdollistaa saman testin ajamisen useita kertoja. Ajoin testin useita kertoja tuhannen kerran sarjoissa. Joillakin kerroilla jokainen testiajo päättyi onnistuneesti ja joillakin kerroilla Spy-objekti ei vapautunut muistista. Koska heikon viittauksen luomisen pitäisi mahdollistaa objektien turvallisen muistista vapauttamisen (Vilmart, Scalzo & De Simone 2018.), johtuu testien ajoittainen epäonnistuminen todennäköisesti testausympäristön muistinhallinnasta, eikä tuotantokoodin muistinhallinnan ongelmista. Opin viikon aikana uutta Swift-ohjelmointikielen tavasta hallita muistia ja kuinka voimme testeissä varmistua objektien muistista vapauttamisesta.

3.6 Viikko 6 (18.-22.9.2023)

Maanantai 18.9.

Lukituslaitteen ohjelmistopäivityksen kanssa työskennellessämme kohtasimme ongelman liittyen Data-tyypin käsittelyyn. Yritämme pilkkoa siirrettäviä paketteja alkuperäisestä Data-arraysta käyttäen Swiftin tarjoamia kokoelmien prefix ja suffix -funktioita. Jos siirrettävä pakettikokomme on esimerkiksi 20, voisimme ottaa kokoelmasta 20 ensimmäistä elementtiä käyttäen prefix(20) funktiokutsua, siirtää ne oheislaitteelle ja tämän jälkeen muuttaa olemassa olevaa kokoelmaamme sisältämään loput elementit, joita ei ole vielä siirretty käyttäen suffix(20) funktiokutsua. Jatkaisimme tätä kokoelman pilkkomista niin kauan, kun kokoelmassa riittää siirrettävää dataa. Suffix-funktiokutsu ei muuttanutkaan alkuperäistä kokoelmaa. Meidän täytyisi luoda kopio kokoelmastamme, jota sen sijaan muokkaisimme. Pyrin tutustumaan asiaan lähemmin tällä viikolla ja selvittämään mistä tämä johtuu.

Tiistai 19.9.

Päätimme pilkkoa siirrettävän pakettimme osiin käyttäen Data-objektin subdata-funktiota suffix-funktion sijaan. Suffix-funktion käyttäminen edellytti uuden Data-objektin luomista jokaiselle siirtokerralle, koska suffix-funktio ei mahdollista Data-objektin sisällön kopioimista, toisin kuin subdata-funktio tekee. Ryhdyimme kirjoittamaan testiä paketin siirtämiselle ja päädyimme kovakoodaamaan siirrettävät paketit testeissämme. Testin lopuksi vertaisimme, vastaako siirretyt paketit näitä kovakoodattuja pakettejamme. Totesimme, että voisimme saada testistä luotettavamman, jos pystyisimme kovakoodaamisen sijaan luomaan sattumanvaraisen kokoisen Data-objektin ja lopuksi testaamaan, siirtyikö kaikki paketit perille kuten pitääkin. Loimme testissämme sattumanvaraisen kokoisen Data-objektin. Tämän jälkeen laskimme, montako siirrettävää pakettia tästä objektista saadaan sattumanvaraisella maksimipakettikoolla ja lopuksi vertasimme siirrettyjen pakettien määrää ja sisältöä näihin tietoihin. Ajoimme

varmuuden vuoksi testit läpi sadan kerran toistona, ja testit menivät jokaisella kerralla läpi.

Keskiviikko 20.9.

Tähän asti olemme toteuttaneet ohjelmistopäivityksen toiminnallisuudesta korkeamman tason toteutusta, jossa rakennamme delegate-kutsut ja palautamme delegaten vastaukset continuationeilla käytettäväksi seuraavassa prosessimme vaiheessa. Ryhdyimme tänään käsittelemään siirrettäviä paketteja ja aloimme tutkimaan dokumentaatiota asiaan liittyen. Emme voineet suoraan hyödyntää dokumentaation tarjoamia esimerkkejä, koska jotkin Swift-funktiot olivat vanhentuneet ja ne täytyisi päivittää käyttämään uudempia versioita hieman erilaisella syntaksilla. Päätin alkaa opiskelemaan ja tutustumaan lisää muistinhallintaan Swift-ohjelmointikielellä ymmärtääkseni paremmin prosessin kulkua. Loppupäivästä saimme tietoomme erään ongelmatapauksen, jonka selvittämistä jatkamme huomenna.

Torstai 21.9.

Aloimme selvittämään eilen ilmennyttä ongelmaa. Käytämme toteutuksessa tapahtumien kuunteluun Swiftin tarjoamaa Combine-viitekehystä, joka mahdollistaa erilaisten tapahtumien lähettämisen ja näiden tapahtumien kuuntelun. Huomasimme, että kuuntelemme näitä tapahtumia kahdessa eri paikassa ja toisesta niistä emme saaneet kaikkia tapahtumia. Keskitimme tapahtumien kuuntelun yhteen paikkaan ja saimme asian korjattua. Tämä edellytti hieman refaktorointia ja tämä osoittautui rakenteellisestikin hyväksi ratkaisuksi, koska se selkeytti ohjelmakoodin rakennetta. Kun ryhdyimme lisäämään muutoksiamme versionhallintaan, huomasimme, että jotkin testit eivät menneet läpi pipelinen testausautomaatiossa. Ryhdyimme tutkimaan asiaa tarkemmin ja huomasimme testeissämme pienen virheen asynkronisten toiminnallisuuksien testaamisessa,

kun emme odottaneet erään funktion valmistumista, jonka takia testit eivät menneet läpi jokaisella ajolla. Korjasimme testit ja saimme korjattua ilmenneen ongelman tämän päivän aikana.

Perjantai 22.9.

Päivän tavoitteena oli aloittaa luomaan toteutusta, jolla näytettäisiin kaikki lukolta saadut tilat sovelluksemme näytöllä, kun lukituslaitetta operoidaan. Tähän mennessä lukituslaitteen tiloista on näytetty sovelluksessa vain tilat siitä, onko lukituslaite auki vai kiinni. Näiden lisäksi voisimme näyttää tilat, joissa lukko on aukeamassa tai lukittumassa. Toteutus vaati jonkin verran muutoksia ohjelmakoodin rakenteeseen, emmekä olleet vielä aivan varmoja millaisista muutoksista olisi kysymys. Sen sijaan, että olisimme tällä kertaa alkaneet luoda toiminnallisuutta testivetoisesti, loimme versionhallintaan uuden spike-haaran. Käytämme spike-haaroja kokeellisten asioiden kehittelyyn, kun emme ole vielä täysin varmoja siitä, millainen lopullinen toteutus tulee olemaan tai onko ideamme toteutettavissa. Kun olemme saaneet spike-haarassa asioita kokeiltuamme käsityksen siitä, millainen lopputulos tulisi suurin piirtein olemaan, siirrymme uuteen haaraan, jossa alamme luoda toiminnallisuutta alusta asti testivetoisesti.

Viikkoanalyysi

Olin havainnut jo aiemmin ajoittaista epäonnistumista testeissämme ja hieman tutkinutkin syitä, joista tämä voi johtua. Ongelma on vaikuttanut olevan testiympäristössämme ja muistinhallinnassa asynkronisia testejä ajaessamme. Robert C. Martin mainitsee, ettei testien epäonnistumisia tulisi koskaan jättää huomiotta, vaan syy kannattaisi selvittää ja korjata (Martin 2008, 187). Vaikka testien hajoaminen näyttääkin johtuvan testiympäristöstä eikä tuotannossa ongelmaa pitäisi olla, voi mahdollisesti usein ilmenevät vääränä hälytyksinä pitämämme testien epäonnistumiset peittää alleen oikeita tuotantoon pääseviä

ongelmia, jos testien aiheuttamat väärät hälytykset jätetään huomiotta (Khorikov 2020). Otinkin tällä viikolla tavoitteekseni tutkia, miksi jotkin testit hajoavat ajoittain ja voisiko testejä saada jotenkin muutettua sellaiseen muotoon, että ne menisivät aina läpi.

Testeissämme luomme usein Spy-objektimme sisään kokoelman, joka sisältää erilaisia viestejä, joita voimme vertailla testeissämme. Kun testattava yksikkömme kutsuu toisen luokan funktiota, lisäämme tätä toista luokkaa simuloivan Spy-objektimme funktiossa viestin kokoelmaan, että tätä funktiota on kutsuttu tietyillä arvoilla. Tämän jälkeen voimme tarkistaa, kutsuiko testattava yksikkömme toista luokkaa oikeilla arvoilla. Jokainen testi luo uuden Spy-objektin ja uuden kokoelman viesteistä, johon mahdollisesti useat Spy-objektin funktiot lisäävät viestejä silloin kun niitä on kutsuttu. Jotkin testit antavat ajoittaisia virheitä, jotka viittaavat epäonnistuneeseen muistiviittaukseen, kun kokoelmaan yritetään lisätä viestiä. Jokaisen testin alussa kokoelma lisätään muistiin ja testin lopuksi se poistetaan muistista. Kun useita testejä ajetaan samaan aikaan, alkoi testit antamaan virheitä kokoelman muistiviittauksesta. Veikkaukseni on, että syy tämän taustalla on usean eri testin samanaikaisesti luomat kokoelmat muistissa, joita eri testit yrittävät käsitellä samaan aikaan eri säikeistä.

Aloin miettiä, voisimmeko ajaa testit turvallisesti siten, että jokainen testi odottaa edellisen valmistumista sen sijaan, että testit mahdollisesti kilpailisivat keskenään muistista. Kokeilin lisätä Spy-objektillemme operaatioiden suoritusta hallinnoivan DispatchQueue-jonon. Loin jonon avulla synkronisen operaation, jonka sisällä suoritetaan viestien lisääminen kokoelmaan. Operaation suorittaminen synkronisessa jonossa takaa, että kahta eri operaatiota ei suoriteta samaan aikaan (Vilmart, Scalzo & De Simone 2018). Ajoin epäonnistuneen testin uudestaan tuhat kertaa peräkkäin ja kaikki suoritukset menivät läpi. Seuraavaksi kokeilin, oliko operaatioiden synkronisella suorittamisella vaikutusta testien suoritusnopeuteen. Aiemmin testeissämme oli ollut asetettuna pieniä viiveitä odottamaan asynkronisten operaatioiden päättymistä. Poistettuani viiveet käytöstä koska niitä ei enää tarvittaisi, huomasin testien olevan ehkäpä jopa hieman nopeampia synkronisesti ajettuna kuin manuaalisten viiveiden kanssa. Opin viikon aikana muistinhallinnan lisäksi myös uusia asioita Data-tyypin

käsittelystä Swift-ohjelmointikielellä. Koodipohjan tutummaksi tuleminen on auttanut omien ratkaisujen kokeilemisessa ja mielestäni onnistuin viikon aikana tavoitteessani löytää ongelmia ajoittain hajoavissa testeissä ja kokeilla omia ratkaisujani näiden korjaamiseksi.

3.7 Viikko 7 (25.-29.9.2023)

Maanantai 25.9.

Päivä alkoi viikkopalavereilla, joiden jälkeen ryhdyimme muuttamaan ohjelmakoodimme rakennetta lukituslaitteen ohjelmistopäivityksessä. Jotkin funktiomme alkoivat kasvaa sisäiseltä logiikaltaan hieman liian suuriksi ja monimutkaisiksi, joten lähdimme pilkkomaan toiminnallisuutta pienempiin osiin. Aiemmin meillä oli vain yksi delegate-funktio tiettyjen arvojen muuttumiselle, jonka sisällä käsittelimme useita eri skenaarioita riippuen funktiolle syötettävästä datasta. Päätimme luoda useamman delegate-funktion jokaiselle erilaiselle tapaukselle, jotka jouduimme käsittelemään alkuperäisessä delegate-funktiossamme. Toteutus mahdollisti sekä continuationeiden luomisen, että palauttamisen toiminnallisuuksien liittämisen samoihin tiedostoihin toistensa kanssa, joka lisäsi ohjelmakoodin luettavuutta huomattavasti. Myös testiluokkamme selkeytyivät, kun yksittäiset testit eivät tarvinneet enää niin paljoa pohjustamista kuin ennen.

Tiistai 26.9.

Aloimme aamulla korjata ongelmaa, erään tapahtumaketjun toimintojen prosessointijärjestykseen liittyen. Eräs prosessi voisi tapahtua taustalla, jolloin toinen prosessi voitaisiin suorittaa samaan aikaan. Joissakin tapauksissa prosessien valmistumisen järjestyksellä on kuitenkin merkitystä. Saimme asian nopeasti korjattua luomalla ajamalla toinen prosessi taustalla ja seuraava prosessi laitettiin odottamaan toisen prosessin päättymistä. Jatkoimme iltapäivällä ohjelmistopäivityksen toteuttamista ja saimme viimeisteltyä korkeamman tason

toteutuksen ja delegate-kutsut, joten pääsemme todennäköisesti lähitulevaisuudessa luomaan toteutuksen yksityiskohtia.

Keskiviikko 27.9.

Ryhdyimme tutkimaan erilaisia mahdollisia keinoja tehdä testausympäristöstämme vakaampi asynkronisia testejä suorittaessa. Sen sijaan, että olisimme luoneet Spy-luokalle DispatchQueue-jonon tapahtumille, kokeilimme luoda Spy-luokasta Actorin, joka takaisi luokan kutsumisen eri säikeistä turvallisesti. Tämä ei kuitenkaan toiminut, koska Actor-luokka vaatii kaikkien sen sisältävien funktioiden olevan asynkronisia, mitä kaikki Spy-luokkamme toteuttavien porttien funktiot eivät ole. Asiaa tutkiessamme huomasimme, että analytiikkajärjestelmämme on kirjannut ylös virhetilanteita tästä kyseisestä prosessista aiemmin. Ongelma ei siis olekaan pelkästään testausympäristössämme, vaan tuotantokoodimme voi harvinaisissa tapauksissa aiheuttaa saman ongelman. Päätimme, että asiaa on syytä tutkia lähemmin pikaisesti ja ryhdymme selvittämään asiaa heti huomenna.

Torstai 28.9.

Tarkastelimme tuotantokoodiamme ja huomasimme, että suoritamme erääseen prosessiin kuuluvan asynkronisen operaation taustalla, joka voi aiheuttaa kilpailutilanteen datan päivittämisen kanssa. Päätimme refaktoroida ohjelmakoodiamme ja poistaa tämän taustaoperaation käytöstä ja toteutimme sen sijaan kaikki tarvittavat operaatiot synkronisesti. Koska suoritamme operaatiot synkronisesti, tapahtuvat kaikki kutsut oikeassa järjestyksessä ja seuraava kutsu odottaa edellisen valmistumista, jolloin kilpailutilanteita datan muokkaamisessa ei pääse tapahtumaan. Pääsimme testeissämme eroon sekä operaatioiden viivästyksistä, joita oli luotu simuloimaan keinotekoisesti asynkronisen funktion valmistumisen viivettä. Kaikki testit menivät läpi ja pääsimme luomaan uuden

version sovelluksestamme. Jatkoimme iltapäivästä vielä lukituslaitteen ohjelmistopäivityksen kanssa työskentelyä. Prosessin päätteeksi, pitäisi meidän laskea ja tarkastaa siirretyn datan tarkistussumma, jolla varmistetaan, että siirretty data vastaa alkuperäistä dataa, jota alussa ryhdyimme siirtämään. Tämä vaatii hieman tarkempaa tutustumista aiheeseen ja päätimme jatkaa asiaa seuraavana päivänä.

Perjantai 29.9.

Päätimme käyttää tänään aikaa ohjelmakoodimme refaktorointiin. Totesimme, että voisimme nimetä lukituslaitteen ohjelmistopäivityksessä käytettäviä funktioitamme paremmin, jotta ne kuvaisivat paremmin niiden toimintaa ja sitä, mitä ne palauttavat. Pyrimme myös siirtämään funktioiden sisäistä logiikkaa hieman eri paikkoihin, joissa toiminnan suorittaminen olisi loogisempaa. Lähetämme jokaisessa bluetooth-siirron prosessin vaiheessa delegatella oheislaitteelle komennon. Saamme delegatelta vastauksen, jolloin olemme valmiina aloittamaan kyseisen prosessin. Komennot koostuvat tietyn määrätyn protokollan mukaisista komennoista. Olimme aiemmin nimenneet funktiomme delegatelta saatavan datan mukaisesti, mutta päätimme nimetä funktiomme ja palautustyyppimme komentojen mukaisesti. Näin olisi huomattavasti helpompaa lukea koodia, kun näemme mikä funktio liittyy mihinkin komentoon, koska jokainen komento kuvaa yhtä prosessin osaa.

Viikkoraportti

Edellisellä viikolla opin, ettei hajoavia testejä kannattaisi koskaan jättää huomiotta, joten päätin viikon alussa ottaa puheeksi mahdollisen keinon korjata eräät välillä hajoavat testimme käyttäen DispatchQueue-prosessijonoa. Kun ohjelmassa kutsutaan jotakin lähdettä useasta eri paikasta ja taustaprosessista, saatetaan saada aikaiseksi ilmiö, jota kutsutaan nimellä Race Condition. Lähde

voi olla esimerkiksi luokan sisältämä muuttuja tai tiedosto. Ongelmia ilmene, jos lähde on vain luku -tyyppiä, mutta jos lähdettä sen sijaan kutsutaan useasta eri prosessista ja kohteeseen voidaan kirjoittaa tietoa vähintään yhdestä prosessista, voi ongelmia esiintyä. Ongelma syntyy, kun toinen prosessi voi yrittää käsitellä lähteen tietoa, joka ei vastaa sitä tietoa, jota prosessin tulisi käsitellä, koska tieto on muuttunut toisen prosessin toimesta (Kautsch 2022).

Tietolähde voidaan muuttaa turvalliseksi käsitellä eri paikoista, tekemällä lähteestä säieturvallinen (thread safe). Swift ohjelmointikielessä luokka voidaan muuttaa säieturvalliseksi Actorin avulla. Actor eristää sisäisen tilansa muusta ohjelmasta ja mahdollistaa sisältönsä muokkaamisen synkronisesti. (Kautsch 2022) Tämä takaa luokan turvallisen kutsumisen eri säikeistä, suorittaen jokaisen luokan kutsun yksi kerrallaan sen sijaan, että eri säikeistä samaan aikaan tulevat kutsut yrittäisivät käsitellä luokkaa samaan aikaan. Yritimme luoda Spy-objektistamme Actorin, tavoitteenamme muuttaa kaikki Spy-objektin toiminta synkroniseksi. Tämä ei kuitenkaan toiminut, koska Actor-luokka vaatii kaikkien sen sisältävien funktioiden olevan asynkronisia funktioita. Spy-luokkamme toteuttaa useita eri portteja, joiden kaikki funktiot eivät ole asynkronisia. Tutkiessamme testiympäristöämme ja tuotantokoodiamme huomasimme, että tuotantokoodimme sisältää mahdollisen virhetilanteen. Muutimme toteutusta siten, että poistimme ongelmia aiheuttava taustaprosessin käytöstä, pilkoimme toiminnallisuuden kahteen osaprosessiin ja suoritimme tapahtumat synkronisesti. Tämä korjasi ongelman ja saimme siistittyä myös testausympäristöä. Koska pääsimme poistamaan asynkronisen prosessin poista tuotantokoodista, kykenimme siivoamaan testeistämme pois kaiken asynkronisen toiminnan alustamisen ja tarkistamisen.

Teimme viikon aikana myös paljon ohjelmakoodin refaktorointia ja yleistä parantelua. Saimme funktioiden ja niiden palautustyyppien uudelleen nimeämisillä parannettua ohjelmakoodin luettavuutta huomattavasti. Koska funktion nimen tulisi kertoa miksi se on olemassa, mitä se tekee, mihin sitä käytetään (Martin 2008, 18), nimesimme funktiomme sen mukaan, mihin ohjelmistopäivityksen prosessin komentoihin ja vaiheisiin ne liittyvät sen sijaan, että nimet ja palautustyypit

kuvaisivat delegaten palauttamaa dataa. Tämä oli tarpeellista, koska itse datasta ei selviä lainkaan niin helposti, mihin prosessin vaiheeseen se liittyy. Opin viikon aikana, että hajoavia testejä ei kannata jättää huomiotta, vaikka olettaisikin niiden liittyvän vain testiympäristöön, eikä tuotantokoodiin. Opin ongelmia selvittämällä uusia asioita race condition -tilanteista ja siitä, kuinka tämän kaltaisia kilpailutilanteita voidaan välttää. Opin myös parempia käytänteitä funktioiden, muuttujien ja palautustyyppien nimeämisiin, joka auttaa minua kirjoittamaan luettavampaa ja ylläpidettävämpää ohjelmakoodia.

3.8 Viikko 8 (2.-6.10.2023)

Maanantai 2.10.

Päivä koostui enimmäkseen palavereista ja opinnäytetyön ryhmäohjausluennosta. Muun ajan käytin enimmäkseen heksagonaalisen arkkitehtuurin opiskeluun. Luin Robert C. Martinin Clean Architecture -kirjaa ja vertasin SDK-projektimme rakennetta kirjan sisältämään tietoon. Yritin saada hieman parempaa kuvaa siitä, mitä eri arkkitehtuurin kerroksien tulisi sisältää ja mille kerroksille omat projektimme eri portit ja adapterit sijoittuvat. Iltapäivällä pidimme retrospektiivin, jollaisessa olin ensimmäistä kertaa opinnäytetyöni kirjoittamisen aikana. Retrospektiivissä saimme tuoda esiin asioita mitkä on mielestämme mennyt hyvin, mitkä eivät ole mennyt niin hyvin ja mitä voisimme mahdollisesti muuttaa tai parantaa.

Tiistai 3.10.

Jatkoimme tänään lukituslaitteen ohjelmistopäivityksen toteuttamista. Ryhdyimme toteuttamaan adaptereita, jotka sisältäisivät tarvittavan logiikan prosessin eri osien toteuttamiselle. Päädyimme toteuttamaan melko paljon refaktorointia päivän aikana. Poistimme joitakin portteja ja adaptereita käytöstä ja loimme niistä sen sijaan yksittäisiä apufunktioita. Näille adaptereille ei ollut oikeastaan

tarvetta, koska ne sisälsivät vain yksittäisiä yksinkertaisia funktioita, eivätkä ne oikeastaan adaptoineet mitään eri arkkitehtuurin kerrosten välillä, kuten adapterin heksagonaalisen arkkitehtuurin mukaisesti tulisi lähtökohtaisesti tehdä. Ylimääräiset portit ja adapterit paransivat testattavuutta, mutta vaikeuttivat samalla ohjelmakoodin luettavuutta. Jouduimme testeissämme pohjustamaan testejä hieman enemmän ja rakensimme testien alkuun apufunktioita, joissa pohjustus toteutetaan sen sijaan, että täyttäisimme jokaisen testin alun suurella määrällä toistuvaa koodia.

Keskiviikko 4.10.

Päätimme käyttää tänään kunnolla aikaa suunnitteluun siitä, kuinka jatkamme lukituslaitteen ohjelmistopäivityksen toteuttamista. Ryhdyimme luomaan Miro-sovelluksella kaaviota, joka kuvaisi visuaalisesti, mitä askelia toteutuksemme vaatii. Loimme kaavion myös porteista ja adaptereistamme, sekä niiden yhteyksistä toisiinsa. Päätimme alkaa pitää jokaisen päivän päätteeksi miniretrospektiivejä kehitystiimimme kanssa. Tämä keskustelu kestää noin viidestä kymmeneen minuuttiin ja sen aikana käydään läpi päivän kulkua. Keskustelemme siitä, mikä meni päivän aikana hyvin, mitä opimme, mitä voisimme tehdä paremmin ja asioista, jotka askarruttivat meitä. Totesimme päivän retrospektiivissä, että voisimme käyttää useamminkin aikaa suunnitteluun ja kaavioiden piirtämiseen, koska se auttoi meitä tänään suuresti.

Torstai 5.10.

Aloitimme päivän refaktoroimalla olemassa olevaa tuotantokoodia, koska tarkoituksenamme on luoda saman kaltainen tapahtumien ketju kuin mitä olemme aikaisemmin jo toteuttaneet, mutta haluamme tehdä toteutuksesta paremman. Testejä kirjoittaessamme huomasimme, että tarvitsemme delegate-objektin, joka syötetään kolmannen osapuolen tarjoamalle funktiolle. Testejä varten

meidän tarvitsisi luoda oma Mock-objekti tästä luokasta, joka osoittautui verrattain työlääksi ja päätimme jättää toteutuksen seuraavalle päivälle. Emme olleet täysin ottaneet tätä huomioon suunnitellessamme toiminnallisuutta, joten aiomme huomenna miettiä tapahtumien kulkua uudestaan ja miettiä, kuinka luomme delegate-objektin ja mikä luokka on vastuussa delegaten luomisesta.

Perjantai 6.10.

Suunnittelimme lukituslaitteen ohjelmistopäivityksen siirtoa edeltävää tapahtumaketjua ja erilaisia mahdollisia tapauksia, joita käyttäjä voisi kohdata prosessin aikana. Erilaisia mahdollisia tapauksia on melko paljon, riippuen esimerkiksi siitä, onko käyttäjällä toimivaa internet-yhteyttä tai tapahtuuko bluetooth-yhteyden kanssa joitakin häiriöitä. Päätimme alkaa luoda toteutusta tällä erää onnistuvan prosessin kautta ja käsittelemme mahdolliset ongelmatilanteet myöhemmin. Käytin aikaa myös heksagonaaliseen arkkitehtuuriin tutustumiseen ja luin kirjallisuutta aiheesta. Yritin saada edelleen parempaa kuvaa siitä, mille arkkitehtuurin kerroksille mitkäkin luomamme luokat kuuluvat. Suunnitellessamme luomamme kaaviot auttoivat hahmottamaan kokonaisuutta paremmin.

Viikkoraportti:

Opiskelin tällä viikolla heksagonaalista arkkitehtuuria ja mietin, kuinka sovelluksemme rakenne noudattaa arkkitehtuuria. Tavoitteenani oli lisätä ymmärrystä arkkitehtuurista ja samalla saada parempaa kuvaa siitä, mihin arkkitehtuurin kerroksille mitkäkin luokat kuuluvat. Heksagonaalista arkkitehtuuria kutsutaan myös nimellä Ports & Adapters -arkkitehtuuri. Nimi tulee siitä, että arkkitehtuurilla eriytetään eri osien toiminta toisistaan porttien ja adaptereiden avulla. Arkkitehtuurin mukainen sovellus koostuu Application- ja Domain-kerroksista. Domain-kerros sisältää sovelluksen business-logiikan ja Application-kerros vastaa kommunikoinnista business-logiikan ja ulkomaailman kanssa. Application-

kerroksen ja ulkomaailman välillä käytetään portteja ja adaptereita, joiden avulla erilaiset ulkoiset elementit keskustelevat sovelluksemme kanssa. Ulkoisia elementtejä ovat esimerkiksi business-logiikkaamme käyttävä mobiilisovellus tai tietokanta.

Heksagonaalisessa arkkitehtuurissa Domain-kerroksen ulkopuolella olevat asiat ovat riippuvaisia Domain-kerroksesta, mutta Domain-kerros ei saa olla koskaan riippuvainen Domainin ulkopuolisista asioista. (Martin 2018). Mitä sisemmälle kohti Domainia arkkitehtuurin kerroksissa mennään, sitä korkeatasoisempaa on luokkien sisältö. Arkkitehtuuri nojaa Robert C. Martinin mainitsemaan Dependency Rule -sääntöön, jonka mukaan riippuvuudet saavat osoittaa vain sisäänpäin, eli ulkopuolelta Domain-kerrokseen päin (Martin 2018.). Sisemmät kerrokset eivät saa tietää mitään ulkoisilla kerroksilla tapahtuvista asioista. Esimerkiksi tietokanta yhdistetään sovellukseen portin ja adapterin avulla. Portti tarjoaa abstraktin toteutuksen tarvittaville toiminnoille, esimerkiksi hakemista ja kirjoittamista varten. Adapteri tarjoaa toteutuksen tälle abstraktiolle. Adapterin tehtävä on muuttaa ulkopuolelta tuleva tieto sellaiseen muotoon, jota sovelluksemme seuraava kerros pystyy käsittelemään. Portti ja adapteri mahdollistaa esimerkiksi tietokannan vaihtamisen täysin toiseen tietokantaan ilman, että sovelluksemme sisäistä logiikkaa täytyy muuttaa lainkaan. Tietokannan vaihtuessa sovelluskehittäjän tarvitsee muokata vain adapteria, jolla haetaan ja muokataan ulkopuolelta tuleva tieto sovellukselle sopivaan muotoon.

Yrityksessämme kehitettävään tuotteeseen kuuluu sekä mobiilisovellus, että SDK-paketti, jonka kanssa olen enimmäkseen työskennellyt lukituslaitteen ohjelmistopäivityksen toteutusta kehittäessäni. Mobiilisovellus on ulkoinen elementti heksagonaalisen arkkitehtuurin näkökulmasta. SDK-pakettiamme käyttää myös ulkoiset integraattorit, jotka voivat luoda omat sovelluksensa, jotka kommunikoivat SDK-pakettimme kanssa. Sovellukset ovat riippuvaisia SDK-paketista, mutta SDK-paketin ei tarvitse tietää, kuka sitä käyttää eikä SDK-paketti ole riippuvainen sen käyttäjistä. SDK-paketin Domain-kerroksen ensimmäisellä tasolla sijaitsevat käyttäjätapaukset, jotka tarjoavat hyvin korkeatasoisen toteutuksen toiminnoista, joita SDK-paketin käyttäjät haluavat suorittaa. Domain-kerroksen eri tasoilla käytämme portteja ja adaptereita kommunikoimaan seuraavien

kerrosten kanssa. Viikon aikana sain paremman kuvan siitä, mille kerroksille sovelluksemme eri luokat asettuvat arkkitehtuurillisesti. Opiskelu jatkuu vielä seuraavalla viikolla, koska en ole saanut muodostettua täyttä kokonaiskuvaa aivan kaikkien luokkien vastuista. Koen kyenneeni viikon loppupuolella osallistumaan paremmin keskusteluun suunnitelmia tehdessämme, koska ymmärrykseni sovelluksen rakenteesta oli kasvanut.

3.9 Viikko 9 (9.–13.10.2023)

Maanantai 9.10.

Päätin käyttää tänään aikaa opiskeluun ja sovellusten julkaisuprosessien tutkintaan. Aamun palaverien jälkeen ryhdyin kirjoittamaan sovellusta, jossa toteuttaisin töissä käytettäviä käytänteitä testien kirjoittamisessa sekä porttien, adaptereiden ja service-luokkien luomisessa. Tarkoituksenani oli saada toistoa ja tuntumaa usein toistuvien asioiden toteuttamisessa. Sain viime kuussa sidosryhmien kokoontumisessa tehtäväkseni aloittaa luomaan dokumentaatiota mobiilisovellusten ja SDK-pakettien julkaisemisesta. Tutustuin näiden julkaisuprosesseihin ja pidimme työkavereiden kanssa palaverin, jossa esittelin tekemääni dokumentaation pohjaa. Sain hyviä vinkkejä esimerkiksi rakenteesta ja termistöstä sekä siitä, mikä prosessissa on automatisoitua ja mitä pitää tehdä manuaalisesti. Keskustelen jatkossa Android- ja iOS-tiimien kanssa dokumentaation rakenteesta ja sisällöstä ja muokkaan dokumentaatiota tarkemmaksi tulevaisuudessa. Dokumentaatioiden luominen ei ole itselleni vielä kovin tuttua, joten tehtävä on erittäin opettavainen.

Tiistai 10.10.

Ryhdyimme aamulla tutkimaan erästä esiin noussutta ongelmaa. Android-puolella kyseistä ongelmaa ei ole, joten tutustuimme Androidin ohjelmakoodiin ja huomasimme, että toteutus on hieman erilainen. Päätimme muuttaa

toteutustamme vastaamaan Androidin toteutusta. Toteutus vaati operaatioiden suorittamista hieman eri järjestyksessä kuin tähän mennessä. Kokonaisuudessaan onnistunut prosessi edellyttää useamman yksittäisen prosessin onnistumista ja välissä meidän täytyisi pitää tietoa muistissa, jotta voimme jatkaa prosessia, jos kaikki operaatiot eivät onnistu. Jatkamme tänään kesken jäänyttä toteutusta huomenna.

Keskiviikko 11.10.

Jatkoimme tänään työskentelyä eilen ilmenneeseen ongelmaan liittyen. Prosessimme sisältää useita funktiokutsuja. Osa funktiokutsuista on sellaisia, että niiden epäonnistuessa emme halua jatkaa prosessia. Joidenkin funktiokutsujen epäonnistuminen on kuitenkin hyväksyttävää prosessin lopputuloksen kannalta. Koko prosessin ei esimerkiksi tarvitse epäonnistua, jos jokin lopputuloksen kannalta ei niin kriittinen operaatio aiheuttaa virheen. Opin käyttämään Swiftin optional try -ominaisuutta, jolloin funktiomme ei käsittele tapahtunutta virhettä. Koska epäonnistunut funktiokutsu sisältää toiminnallisuuden, jossa mahdollinen virhetilanne kirjataan tapahtumalokiin, voimme hylätä mahdollisen virheen lopuksi, eikä kyseisen operaation epäonnistuminen lopeta koko prosessiamme virheen tapahtuessa.

Torstai 12.10.

Aloitimme päivän jatkamalla eilen kesken jäänyttä ongelman ratkaisemista. Tehtävänämme oli toteuttaa toiminnallisuus, jossa lähetämme tietoa sovelluksesta ja tämän jälkeen poistamme tietoja muistista. Testeissämme meidän oli luotava Spy-objektillemme kokoelma vastauksia, jotka kertoisivat, onnistuiko lähetys vai ei. Loimme Spy-objektillemme kokoelman Result-tyypin vastauksia. Jokaisella Spy-objektin funktion kutsulla, poistimme ensimmäisen elementin kokoelmasta, jolloin Spy-ohjektimme palauttaisi aina jäljellä olevan kokoelman

tuloksista niin kauan, kuin niitä on jäljellä. Tekemällä näin, pystyimme testaamaan toistuvasti tapahtuvaa toimenpidettä ja saimme jokaiselle toistolle palautettua simuloidun tuloksen kutsumme vastauksista. Saimme toiminnallisuuden toteutuksen valmiiksi ja pystymme julkaisemaan sovelluksesta uuden version, jossa ongelma on korjattu.

Perjantai 13.10.

Ryhdyimme luomaan tänään Use Case -luokkaa lukituslaitteen ohjelmistopäivitykselle. Clean arkkitehtuurin täytyy tukea sovelluksen käyttötarkoitusta ja tämä on arkkitehtuurin ensimmäinen prioriteetti; arkkitehtuurin täytyy tukea käyttötapauksia eli Use Caseja. (Martin 2018.) Use Case -luokat sisältävät jonkin tietyn käyttötapauksen toiminnallisuuden. Tapauksessamme käyttötapaus on ohjelmistopäivitys ja luokkamme nimi kuvaa tätä käyttötapausta. Tälle Use Case -luokalle injektoidaan kaikki adapterit, joita se tarvitsee toteuttaakseen kaiken tarvittavan prosessin toiminnallisuuden. Use Case -luokka sisältää sisäisesti vain portteja, joiden funktioita se kutsuu ja joita luokalle injektoidut adapterit toteuttavat. Use Case -luokkamme sisältävät adapterit olivat esimerkiksi ohjelmistopäivityksen hakeminen ja tallentaminen muistiin, päivitysprosessin suorittaminen ja päivitysprosessin viimeistely. Saimme kirjoitettua päivän aikana Use Case -luokan testeineen valmiiksi.

Viikkoraportti

Viikon tavoitteena oli käyttää enemmän aikaa opiskeluun ja pyrkiä sisäistämään paremmin ohjelmakoodin rakennetta ja Swift-ohjelmointikieltä. Sovellus käyttää melko paljon kolmannen osapuolen kirjastoja, joista joidenkin tarkoitus on ollut itselleni hieman epäselvä. Käytin aikaa dokumentaatioiden lukemiseen ja kaavioiden tulkitsemiseen ja koin tämän erittäin hyödylliseksi. Ohjelmoidessa vauhti on välillä verrattain nopea, eikä kaikkia asioita ehdi välttämättä työn lomassa

sisäistää. Opin myös uusia asioita Swift-ohjelmointikielestä ja koen, että sain lisää itsevarmuutta työskentelyyni ja ohjelmakoodin kirjoittamiseen. Huomasin, että kykenin ottamaan tehokkaammin osaa keskusteluun ja tarjoamaan omia ideoitani erilaisiin tapauksiin, kun kävimme läpi asioita esimerkiksi käyttäjätapausten ja adaptereiden suhteesta, sekä näiden kommunikoinnista kolmansien osapuolien kirjastojen kanssa. Sain viikon aikana myös tuntumaa dokumentaation luomiseen ja ymmärrykseni sovelluksien julkaisuprosesseista kasvoi.

Opin myös erilaisia tapoja käsitellä mahdollisia virhetilanteita ohjelman aikana. Kaikki virheet eivät välttämättä ole sellaisia, että prosessi olisi tarpeen jättää kesken, mutta on tärkeää tietää, jos virhe tapahtui. Kaikki virheet ohjelmassa lisätään lokiin, josta voimme nähdä mahdolliset tapahtuneet virheet prosessin aikana. Pystyimme testaamaan, että prosessi onnistui, vaikka välillä jokin funktio olisi heittänyt virheen laittamalla Spy-objektimme palauttamaan virheen jollekin prosessin funktiokutsulle. Vertasimme testeissämme, että tämän funktiokutsun jälkeiset operaatiot olivat suoritettu onnistuneesti virheestä huolimatta. Testasimme myös lokeista vastaavan luokkamme, että virheen tapahtuessa lokiin on kirjattu tapahtunut virhe. Testejä kirjoittaessani opin paremmin kirjoittamaan pienimmän mahdollisen määrän tuotantokoodia testin läpäisemiseksi. Tuotantokoodia saisi testausvetoisessa kehityksessä kirjoittaa vain sen verran, kuin testin läpäisemiseksi on tarpeen (Martin 2008, 122). Kirjoitimme ensin testin toiminnallisuudelle, jossa suoritamme haluamamme funktiokutsun. Tämän jälkeen lisäämme tuotantokoodiin tämän funktiokutsun ja läpäisemme testin. Tämän jälkeen kirjoitimme testin, jossa epäonnistuva funktiokutsu kirjaa lokiin oikeat viestit. Tässä kohtaa emme luoneet vielä tuotantokoodiin logiikkaa virheen käsittelyyn, vaan lisäsimme vain lokifunktiomme mukaan tuotantokoodiin ja testit läpäistiin. Tämän jälkeen lisäsimme testit onnistuneen funktiokutsun lokien lähettämiselle ja vasta tämän testin kohdalla meidän täytyi lisätä logiikka onnistuneen tai epäonnistuneen tapauksen käsittelylle. Vaikka muutimme tuotantokoodin logiikkaa, jo aiemmin kirjoittamamme testit pitivät huolen siitä, että lisäämämme logiikka ei riko mitään ja ohjelma toimii edelleen oikein.

3.10 Viikko 10 (16.–20.10.2023)

Maanantai 16.10.

Pidimme aamulla palaverin ilmenneestä ajankohtaisesta ongelmasta. Palaverissa pääsimme tekemään yhteistyötä toisen tiimin kanssa, joka tarjosi tärkeää tietoa ongelmasta ja mahdollisista ratkaisukeinoista. Saimme palaverin aikana muodostettua kuvan siitä, kuinka voisimme ratkaista ongelman, ja loimme toteutusta loppupäivän ajan. Ongelmaan liittyy melko paljon erilaisia tapauksia, jotka meidän tulisi ottaa huomioon, jos jotakin menee vikaan prosessin aikana. Loimme päivän aikana Use Case -tason toteutuksen tarvittavalle toiminnallisuudelle ja jatkamme huomenna yksityiskohtien toteuttamista. Opin päivän aikana paljon uutta käyttämiemme kolmansien osapuolten kirjastojen toiminnasta ja siitä, miten näiltä saatavaa tietoa sovelluksessamme käsitellään.

Tiistai 17.10.

Jatkoimme tänään eilisen ongelman ratkaisemista ja saimme viimeisteltyä toiminnallisuuden päivän aikana. Testeissämme Spy-objektimme täytyi toteuttaa useaa eri protokollaa, mutta tarvitsimme testeissämme toteutuksen vain pienelle osalle protokollien vaatimista funktioista. Opin, että protokollille voi luoda oletusarvoisen toteutuksen funktioille, jolloin Spy-objektillemme ei tarvitse lisätä kaikkia funktioita, joita emme tarvitse. Lisäsimme SDK-pakettimme julkisesti jaettaviin funktioihin kaksi eri versiota toteutuksestamme. Toinen käyttää async-await -toteutusta ja toinen completion handler -toteutusta. Integraattorit voivat valita, kumpaa näistä käyttävät. Testasimme molemmat toteutukset ja completion handler -funktioiden tapauksessa meidän täytyi vain tarkastaa funktion tuottama lopputulos funktiomme trailing block -osiossa, kun taas async-await -version tapauksessa funktio palauttaa meille tarkastettavan lopputuloksen.

Keskiviikko 18.10.

Aloimme tänään luoda Use Case -luokkaa erään prosessin tilan tarkastamiselle, sekä toista Use Case -luokkaa tämän tilan nollaamiselle. Kun aktiivinen käyttäjä poistetaan, voi käyttäjä alkaa käyttää sovellusta myöhemmin uudestaan uudella aktivointilinkillä. Kohtasimme hieman ongelmia yrittäessämme testata Continuation-funktioitamme ja päätin alkaa tutkimaan asiaa hieman enemmän omalla ajallani. Opin, että continuation-objektista voi luoda publisher-objektin, jonka tapahtumia voi seurata luomalla Combine-viitekehyksen tarjoaman tapahtumien kuuntelijan. Voisimme ehkäpä testeissämme hyödyntää tätä ominaisuutta ja aion tutustua aiheeseen tarkemmin lähipäivinä. Tutustuin myös geneeristen tyyppien käsittelyyn Swift-ohjelmointikielellä ja opin paljon uusia asioita geneeristen tyyppien käsittelystä.

Torstai 19.10.

Pidimme aamulla tapaamisen Android-tiimin kanssa. Pyrimme yhtenäistämään julkisia integraattoreille näkyviä toiminnallisuuksia ja asioiden nimeämisiä sovellustemme SDK-paketeissa, ja pääsimme jakamaan ideoitamme tiimien kesken. Tapaamisen seurauksena päädyimme uudelleen nimeämään ja hieman muokkaamaan joitakin funktioita. Päivitimme olemassa olevaa julkista dokumentaatiota ja lisäsimme dokumentaation uusille funktioille ja muille toiminnallisuuksille. Opin päivän aikana uusia asioita virallisten dokumentaatioiden luomisesta XCodella ja käytänteitä siitä, mitä asioita ja missä muodossa käyttäjille kannattaa jakaa. Iltapäivällä osallistuin myös tapaamiseen erään kolmannen osapuolen kirjaston kehittäjien kanssa. Pääsimme kysymään meitä askarruttaneita kysymyksiä siitä, mitä heidän funktionsa tekevät sisäisesti ja millaisia virheitä funktiot heittävät, ja saimme paljon hyödyllistä informaatiota kirjaston toiminnasta.

Perjantai 20.10.

Olimme saaneet viikon tärkeimmät asiat jo tehtyä, joten päätimme käyttää aamupäivän asioiden paranteluun. Päivitimme dokumentaatiotamme ja tutkimme erään ulkoisen laadunvarmistuspalvelun-analytiikkaa. Analytiikka tarjosi useita ohjelmakoodin luettavuutta ja ylläpidettävyyttä helpottavia vihjeitä, jotka korjaamalla voimme parantaa koodiamme. Ohjelmakoodimme sisälsi jonkin verran ylimääräistä koodia, jonka pystyimme muuttamaan kompaktimpaan muotoon. Paransimme raporttien perusteella myös virheiden käsittelyämme. Tehtävälistallamme on ollut erään analytiikkatyökalun lisääminen iOS SDK -pakettiimme ja ryhdyimme lisäämään tätä työkalua ohjelmaamme. Työkalun avulla voimme lisätä ohjelmamme kirjaamat lokit myös ulkoiseen järjestelmään, jotta voimme analysoida tapahtumalokejamme tarkemmin. Iltapäivällä osallistuin vielä palaveriin, joka liittyi Android SDK -paketin toimintaan ja jota eräs integraattorimme oli pyytänyt järjestämään.

Viikkoraportti

Viikon päällimmäisenä tavoitteena oli saada korjattua integraattorin raportoima ongelma. Loimme viikon aikana kaavioita tapahtumien kulusta ja mahdollisista ongelmakohdista, jotka voisivat vaikuttaa prosessin kulkuun. Havaitsimme kohdat, joiden toimintaa kehittämällä voimme muuttaa prosessiamme käsittelemään erilaiset virhetilanteet paremmin.

Testaamisen osalta prosessi eteni melko tavanomaisesti. Injektoimme Spy-objektimme SUT-objektillemme ja testasimme virhetilanteet, onnistuvat tapaukset ja että oikeita funktioita kutsutaan. Sovelluksemme uloimmalla kerroksella, jossa lähetämme kutsuja kolmannen osapuolen kirjastolle tarkastaaksemme prosessin tilan, käytimme delegatea. Delegate kertoo meille aika ajoin päivitetyn tilan. Delegaten vastauksien kuunteluun käytimme continuationia, jota yritimme testata käyttäen continuationin tarjoamaa publisher-tapahtumien kuuntelijaa. Testaaminen osoittautui kuitenkin verrattain haasteelliseksi. Huomasin samalla,

että XCode tarjoaa meille analytiikkaa esimerkiksi hitaimmista yksikkötesteistä. Monet näistä testeistä sisälsivät continuationeiden testaamista. Ryhdyin tutkimaan, voisiko testejä nopeuttaa muuttamalla testien toteutusta käyttämään publisher-tapahtumien kuuntelijaa, mutta en ainakaan vielä tällä viikolla löytänyt parempia ratkaisuja. Tutkin alan kirjallisuutta, mutta löytämäni esimerkit eivät tarjonneet tilanteeseen sopivia ratkaisuja Combine-viitekehyksen testaamiseen monisäikeisessä ympäristössä. Ensin vaikutti, että testit toimivat, mutta huomasin testejä useita kertoja ajaessani, että testit hajoavat aika ajoin. Onnistuin puolittamaan testeihin kuluvan ajan, mutta toistaiseksi ratkaisu oli helposti rikkoutuva ja testit hajosivat ajoittain. Jatkan asian tutkimista lähitulevaisuudessa, jos ylimääräistä aikaa asialle löytyy.

Opin viikon aikana ymmärtämään Combine-viitekehyksen toimintaa ja sen käyttöön liittyviä haasteita paremmin. Tutustuin myös käyttämämme laadunvarmistuspalvelun ja analytiikkatyökalun toimintaan ja varsinkin analytiikkatyökalun kanssa tulemme työskentelemään heti ensi viikolla, joten tämän viikon opit tulevat tarpeeseen. Kirjoitimme viikon aikana myös verrattain paljon dokumentaatiota SDK-paketillemme ja opin paljon asioita selkeän ja luettavan dokumentaation luomisesta. Luodessamme SDK-paketin uloimman kerroksen toiminnallisuutta, opin kuinka ja miltä osin jaamme SDK-pakettia julkisesti näkyväksi integraattoreillemme.

3.11 Viikko 11 (23.–27.10.2023)

Maanantai 23.10.

Keskustelimme aamun mobiilikehittäjien palaverissa analytiikkatyökalun lisäämisestä SDK-paketteihimme. Tutkimme, mitä analytiikkaa työkalu lähtökohtaisesti kerää ja kuinka voisimme sopeuttaa työkalua paremmin käyttötarkoituksiimme sopivaksi. Toistaiseksi haluamme kerätä analytiikkaa pelkästään tapahtumista, joita käyttäjä on suorittanut ohjelmalla. Tämä analytiikka auttaa meitä paikantamaan sovelluksessa tapahtuvia virhetilanteita paremmin ja puuttumaan niihin tehokkaammin. Loimme versionhallintaamme spike-haaran ja käytimme

loppupäivän kokeiluun, kuinka voisimme ottaa työkalun käyttöön SDK-paketissamme. Pääsimme päivän päätteeksi hyvään lopputulokseen ja aloitamme huomenna varsinaisen toteutuksen testivetoisesti.

Tiistai 24.10.

Aloimme tänään työskentelemään analytiikkatyökalun parissa. Tarvitsemme työkalun instanssin luodessamme API-avaimen ja mietimme erilaisia vaihtoehtoja avaimen säilytykseen ja käyttämiseen. Emme kuitenkaan onnistuneet luomaan testejämme varten avainta, jota voisimme verrata testeissämme instanssia luodessa. Tämän asian tutkiminen jäi kesken, kun ryhdyimme korjaamaan erästä käsillä olevaa ongelmaa. SDK-paketistamme löytyi päättymätön funktiokutsu, joka kutsui itseään uudestaan päättymättömästi tietyssä käyttötapauksessa. Lisäsimme funktioon tarkastuspisteen, joka lopettaa tämän toiston tietyillä arvoilla ja saimme asian korjattua.

Keskiviikko 25.10.

Lisäsimme tänään analytiikkatyökalun SDK-pakettiimme. Työkalun käyttöön ottaminen vaati API-avaimen käyttämistä ja päätimme, että missä tätä avainta pidämme ja kuinka sitä käytämme. Opin päivän aikana, kuinka API-avaimia voidaan käsitellä turvallisesti ilman, että lisäämme näitä tietoja suoraan ohjelmakoodiimme. Lisäsimme analytiikkatyökalun lokeista vastaavaan luokkaamme, joka on vastuussa kaikkien lokien kirjaamisesta. Lokeista vastaava luokka on osa jokaista luokkaa, jotka suorittavat lokien kirjaamista. Meidän tarvitsi vain injektoida analytiikkatyökalun instanssi lokeista vastaavalle luokalle ja näin saamme lokit analytiikkajärjestelmään kaikkialta, missä lokeja kirjataan.

Torstai 26.10.

Minulla oli aamulla aikaa tutkia erään testiluokan testejä, jotka ovat osoittautuneet melko epävakaiksi viime aikoina. Testeissä oli käytetty asynkronisten operaatioiden testaamisessa keinotekoista viivettä funktion suorituksen ja lopputuloksen vertaamisen välillä. Poistin viiveet käytöstä ja lisäsin sen sijaan testeille XCTest-viitekehyksen tarjoamia XCTestExpectation-luokan funktioita, joiden avulla voimme odottaa, kunnes kaikki asynkroniset operaatiot on suoritettu. Kun kaikki tarvittavat operaatiot on suoritettu, kerromme XCTestExpectation-luokalle, että kaikki on valmista ja voimme siirtyä eteenpäin, jonka jälkeen voimme tarkastella operaation lopputulosta. Ajoin testit läpi useita kertoja ja varmistin, että testit eivät enää hajoa. Esitin havaintoni tiimille ja korjasimme epävakaat testitapaukset.

Perjantai 27.10.

Jatkoimme tänään analytiikkatyökalun lisäämistä SDK-pakettiin. Tavoitteenamme oli saada muutettua analytiikan keräämiseen tarvittavan luvan tila. Käyttäjä voi hyväksyä analytiikkatietojen keräämisen tai käyttäjä voi hylätä tietojen keräämisen. Toistaiseksi tarkoituksemme oli tallentaa annettu arvo muistiin, josta hakisimme tämän tiedon sovelluksen käynnistyksen yhteydessä. Tämän jälkeen käyttäjä voisi muuttaa arvoa aina halutessaan ja tallentaisimme tiedon sekä analytiikkatyökalun tietoihin, että puhelimen muistiin. Teimme mahdolliseksi injektoida funktion analytiikkatyökalun adapterille, jolla voisimme muuttaa tätä arvoa. Injektoimalla funktion, pystymme testaamaan adapteriamme helpommin.

Viikkoraportti

Käytimme valtaosan viikosta analytiikkatyökalun ja erityisesti sen logger-ominaisuuden tutkimiseen, toteutuksen suunnitteluun ja itse toteutukseen. Analytiikan kerääminen auttaa meitä esimerkiksi paikantamaan ohjelmassa tapahtuneita virhetilanteita ja analysoimaan näiden virheiden syitä, jotta voimme parantaa ja kehittää tuotteen toimintaa. Pääsin viikon aikana tutustumaan analytiikkatyökalun toimintaan, joka oli minulle entuudestaan tuntematon järjestelmä. Jatkamme asian tutkimista lähitulevaisuudessa.

Ylimääräisellä ajallani keskityin tutkimaan epävakaita testitapauksia sekä tutkin XCoden tarjoamaa analytiikkaa yksikkötesteistämme. Epävakaissa testeissämme käytimme keinotekoisia viiveitä asynkronisten operaatioiden päättymisen odottamiseen. Keinotekoinen viive oli lyhyt, mutta operaatiot eivät ehtineet valmistua jokaisella testiajolla viiveen aikana. Viivettä voisi varmaankin pidentää, mutta samalla testit hidastuisivat turhan viivästyksen seurauksena, jos operaatiot valmistuvatkin aiemmin. Olimme aiemmin käyttäneet testeissämme XCTestExpectation-luokan wait(), tai asynkronisissa testeissä await fulfillment() funktioita. Ennen asynkronisen operaation suorittamista, voimme määrittää XCTestExpectationin, jonka täyttyminen määritetään asynkronisesti. Voimme määrätä testimme odottamaan tämän odotteen täyttymistä ja suoritamme testiemme vertailut vasta tämän jälkeen. Voimme myös asettaa odotteelle aikarajan, missä ajassa odotteen on täytyttävä tai testi epäonnistuu, jotta testi ei jää jumiin odottamaan odotteen valmistumista (Khorikov 2020.). Poistin testeistä keinotekoiset viiveet ja kutsuin asynkronisten operaatioiden päätteeksi await fulfillment() funktiota. Ajoin testit useita kertoja läpi ja totesin, että testit toimivat 100 % ajasta. Näin voimme suorittaa lopputuloksen vertailun testeissä heti, kun asynkroninen operaatio on päättynyt.

Huomasin, että XCoden testianalytiikka huomioi korjatut funktiot listassa, jossa se näyttää hitaimmat testit. Odotteen lisääminen testiin näyttäisi lisäävän 0.1 sekuntia testin suoritusaikaan, mutta tämä on huomattavasti parempi vaihtoehto, kuin epävakaat testit. Testit ovat edelleen nopeita ajaa, mutta testien epävakaus rikkoisi Martinin mainitsemaa sääntöä, jonka mukaan testien täytyy olla

toistettavissa ympäristöstä riippumatta (Martin 2008, 132). Toista XCoden hitaimpien testien listaamaa testiluokkaa tutkiessani huomasin, että voisimme poistaa testeissä yhden ylimääräisen odotteen pois ilman, että testin lopputulos kärsii. Näissä testeissä saimme testin suoritusajasta poistettua odotteen tuottaman 0.1 sekuntia ja testin suoritusnopeus muuttui 0.2 sekunnista 0.1 sekuntiin. Kokeilin, voisinko nopeuttaa testejä myös kolmannessa testiluokassa, joka listattiin hitaimpien testien joukossa. Tässä testiluokassa käytettiin keinotekoisia viiveitä asynkronisten operaatioiden suoritusta odottaessa. Huomasin kuitenkin, että vaikka sain siistittyä testikoodia kompaktimpaan muotoon, testit itseasiassa hidastuivat käyttäessä odotteita keinotekoisten viiveiden sijaan.

3.12 Viikko 12 (30.10.-3.11.2023)

Maanantai 30.10.

Päivän aikana oli useita palavereita ja tämän takia emme päässeet etenemään kovinkaan paljoa toiminnallisuuksiemme toteutuksissa. Aamulla pidimme viikkopalaverin ja kävimme läpi asioita, joita olimme saaneet edellisen taapamisen jälkeen valmiiksi ja mitä aiomme tehdä tällä viikolla. Tämän jälkeen pidimme mobiilikehittäjien palaverin. Keskustelimme siitä, kuinka voisimme parantaa erilaisiin virhetilanteisiin reagoimista sovelluksissamme monitoroimalla tehokkaammin analytiikkaa sovelluksien virhetilanteista. Iltapäivällä pidimme vielä kuukausittaisen retrospektiivin, jossa kävimme läpi mikä on ollut hyvää, mitä voisimme kehittää ja kuinka voisimme toimia paremmin jatkossa. Palaverien välissä kävimme läpi erilaisia keinoja injektoida analytiikkatyökalun alustamiseen tarvittavat konfiguraatiot SDK-paketissamme, jotta saisimme testattua alustuksen toteutuksen.

Tiistai 31.10.

Päivän tavoitteenamme oli saada luotua yksilöivä support-ID, kun SDK-instanssi luodaan mobiilisovelluksessa ensimmäistä kertaa. Tämä ID tallennettaisiin muistiin ja voisimme liittää tämän ID:n analytiikkatyökalumme lokeihin mukaan, jotta voimme helpommin löytää analyytikan seasta tiettyjen käyttäjien kokemia virhetilanteita. Loimme myös integraattoreille julkisen API:n, jonka avulla he voivat saada suppport-ID:n omiin sovelluksiinsa. Kohtasimme hieman ongelmia tiedon muistiin lisäämisen testaamisen kanssa. Toteutuksemme ei osoittautunut parhaalla mahdollisella tavalla testattavaksi, joten emme voineet testata kaikkia mahdollisia yksityiskohtia. Päätimme, että tutkimme asiaa omilla tahoillamme ja yritämme löytää asiaan paremman toteutuksen. Löysimme paremman keinon ja saimme päivän päätteeksi luotua toimivan lopputuloksen.

Keskiviikko 1.11.

Pidimme aamulla palaverin analytiikkatyökalusta Android-tiimin kanssa. Jaoimme tietoa siitä, kuinka olemme toteuttaneet asian iOS SDK -paketissa, jotta saisimme toteutuksista mahdollisimman yhtenäiset sekä iOS-, että Android-järjestelmillä. Lisäsimme toteutukseemme myös toiminnallisuuden, jolla lisäämme SDK-versionumeron lokeihimme, jotta voimme nähdä, mitä SDK-versiota käyttäjä on käyttänyt. Huomasimme manuaalista testausta suorittaessamme, että voisimme käsitellä tiedon muistiin lisäämistä paremmin. Ilman manuaalista testausta, tämä asia olisi jäänyt toistaiseksi huomiotta. Loimme myös dokumentaation SDK-paketillemme diagnostiikan keräämisen osalta.

Torstai 2.11.

Päivän tavoitteena oli viimeistellä analytiikkatyökalun lisääminen. Viimeinen tehtävä asia oli aikaleiman lisääminen, kun käyttäjä hyväksyy diagnostiikkatietojen keräämisen ja lisätä hyväksynnän tila ja aikaleima muistiin. Kohtasimme testeissämme ensin hieman ongelmia, koska testimme ja toteutuksemme aikaleimat eivät kohdanneet, koska aikaleimat luotiin hieman eri aikoihin. Korjasimme ongelman luomalla aikaleiman automaattisesti luodessamme objektin, jonka lisäisimme muistiin ja näin saimme testeihimme saman aikaleiman kuin tuotantokoodillemmekin. Ryhdyimme päivän päätteeksi vielä jatkamaan pitkästä aikaa lukituslaitteen ohjelmistopäivityksen toteuttamista ja huomasimme, että menetämme yhteyden lukkoomme kesken prosessin. Päätimme jatkaa asian tutkintaa huomenna.

Perjantai 3.11.

Tänään tavoite oli selvittää, miksi lukituslaitteemme sammuu kesken ohjelmistopäivityksen. Käytimme paljon aikaa debuggaamiseen ja löysimme kohdan, jossa lukituslaite sammuu. Emme kuitenkaan saaneet selville, miksi niin tapahtuu. Ryhdyimme tutkimaan dokumentaatiota tarkemmin toiminnallisuuden osalta. Tapahtumien kulun ison kuvan hahmottaminen osoittautui jokseenkin haastavaksi ja päätimme jatkaa asian tutkimista ensi viikolla. Opin kuitenkin päivän aikana paljon hyödyllisiä asioita siitä, kuinka ohjelmakoodia voi debugata tehokkaasti sekä missä järjestyksessä asiat tapahtuvat ohjelmistopäivitystä suorittaessamme.

Viikkoraportti

Tällä viikolla iOS-tiimissä työskenteli neljä henkilöä, joten pystyimme työskentelemään pareina. Itselläni viikon tavoitteena oli saada viimeisteltyä

analytiikkatyökalun lisääminen iOS SDK -pakettiin. Maanantain viikkopalaverissa iOS-tiimin osalta prioriteeteiksi kirjattiin analytiikkatyökalun lisäksi lukituslaitteen ohelmistopäivityksen toiminnallisuuden lisääminen. Ohjelmistopäivityksen toteutus on jäänyt viime viikkoina vähemmälle huomiolle muiden prioriteettien takia. Viikon aikana analytiikka-adapteria luodessamme opin, kuinka voimme luoda luokan konstruktorin siten, että voimme testeissämme injektoida luokan alustamiseen tarvittavia funktioita. Tuotantokoodimme alustaa adapterin automaattisesti tietyillä arvoilla, mutta testeissämme voimme injektoida tarvittavat funktiot konstruktoriin. Injektoimalla voimme testata lopputulosta syöttämillämme arvoilla. Jos luokka alustaisi itsensä sisäisillä arvoilla, emme voi testata yhtä hyvin lopputulosta.

Kun olimme saaneet analytiikkatyökalun lisäämisen valmiiksi, siirryimme takaisin lukituslaitteen ohjelmistopäivityksen toteuttamisen pariin. Halusimme päästä tekemään toteutusta mahdollisimman nopeasti, koska mitä pidempään teemme muita asioita, tietämyksemme ohjelmistopäivityksen toteutuksesta unohtuu enemmän ja enemmän. Jouduimmekin muistelemaan ensimmäiseksi, että mihin tilanteeseen jäimme edellisellä kerralla, kun loimme toteutusta. Lisäsimme testisovellukseemme puhelimen näytölle napin, jolla voisimme käynnistää ohjelmistopäivityksen. Kun käynnistimme päivityksen huomasimme, että lukko sammuu kesken operaation. Selvitimme asiaa ensin työparini kanssa yhdessä. Totesimme kuitenkin, että olisi tehokkaampaa kahdestaan työskentelyn sijaan alkaa tutkia asiaa omilla tahoillamme. Paikansimme kohdan, jossa lukituslaite sammuu. Emme kuitenkaan ymmärtäneet, miksi näin tapahtuu. Voisimme tutustua ensi viikolla Android-sovelluksen ohjelmakoodiin ja yrittää verrata, puuttuuko meiltä jokin vaihe prosessissa.

Opin viikon aikana käyttämään XCoden tarjoamaa debugger-työkalua hyödykseni sovelluksen virhetilanteita paikantaessani. Debugger mahdollistaa olemassa olevien ja muistista poistettujen objektien sisällön tarkastelun (Vilmart, Scalzo & De Simone 2018). Debuggerin avulla voidaan merkata ohjelmakoodiin kohtia, joissa sovellus käytettäessä pysähtyy. Voimme tässä kohdassa tutkia sovelluksen arvoja ja verrata, ovatko ne sellaisia kuten pitääkin. Kohtia merkkaamalla voidaan myös havaita, kutsutaanko jotakin funktiota

lainkaan. Emme voi yksikkötesteillämme täysin todistaa kaikkien toiminnallisuuksiemme toiminnallisuutta yhdessä. Ohjelman manuaalisessa testaamisessa löydettyjen ongelmien paikantamisessa debuggerin käytön osaaminen tehostaa työskentelyäni merkittävästi. Ongelman paikantamisessa myös sovelluksemme tuottamat lokit auttoivat meitä paljon, koska voimme nähdä suoraan konsolista, mitä tapahtumia sovellus kirjaa ylös. Huomasin myös viikon aikana, kuinka ongelmallista voi olla, jos jonkin toiminnallisuuden toteuttamiseen tulee liian pitkä tauko. Asiat ehtivät unohtua nopeasti ja kehittäjä saattaa joutua käyttämään paljon aikaa asioiden muisteluun.

3.13 Viikko 13 (13.-17.11.2023)

Maanantai 13.11.

Olin viime viikon kipeänä ja palasin tänään pitkästä aikaa töihin. Kaksi henkilöä iOS-tiimistä oli tänään kiinni päivystystehtävissä ja ratkomassa asiakkaalla ilmenneitä ongelmia. Kolmas henkilö oli kipeänä, joten työskentelin tänään yksin. Aamun viikkopalaverin jälkeen kävin läpi Slack-keskusteluita viimeisen viikon ajalta ja selvitin, mitä on tapahtunut poissaollessani. Loppupäivän käytin tutustuen ajankohtaiseen ongelmaan. Aikani kului tutustuessa ohjelmakoodiin ja kokeillessa erilaisia ideoita, mutta en saanut konkreettisia ratkaisuja aikaiseksi. Päivä oli kuitenkin kehittävä, koska sain tutustua ajan kanssa ohjelmakoodiin ja kokeilla erilaisia asioita.

Tiistai 14.11.

Tutkin aamulla lukituslaitteen ohjelmistopäivityksen alustamisen toteutusta. Yritin selvittää, että puuttuuko iOS SDK -toteutuksestamme jokin vaihe, joka aiheuttaa lukituslaitteen sammumisen kesken ohjelmistopäivityksen. En kuitenkaan onnistunut toistaiseksi löytämään ratkaisua asiaan. Loppupäivän keskityimme muun tiimin kanssa aiemmin raportoidun ongelman korjaukseen.

Kirjoitimme lähinnä kokeellista spike-toteutusta tänään, jonka toteutuksen tuotantokoodiin aloitamme huomenna. Saamme tiedon delegatelta, kun käsittelemämme prosessi on päättynyt. Prosessin päättymisen jälkeen on meidän muokattava asioita muistissa.

Keskiviikko 15.11.

Pidimme aamulla palaverin alkuviikolla ilmenneestä ongelmasta. Ongelman ratkaisu edellyttää toimia erään prosessin erilaisten lopputulosten käsittelyssä. Pidimme palaverin myös erään kolmannen osapuolen kirjaston kehittäjän kanssa ja saimme tietoa mahdollisista syistä erään integraattorimme ongelmaan, jossa integraattorin sovellus on kaatunut erään prosessin aikana. Saimme hyviä vihjeitä ongelman mahdollisesta juurisyystä, jota voimme tutkia tarkemmin myöhemmin. Emme ehtineet tänään vielä tutustua asiaan, koska tehtävälistallamme on kiireellisempiä asioita tälle viikolle.

Torstai 16.11.

Aloitimme päivän muokkaamalla SDK-pakettimme virheiden käsittelyä. Loimme uuden julkisen virhetyypin, jonka haluamme heittää tietyissä SDK-paketin virhetilanteissa. Haluamme jakaa integraattoreille tietoa tapahtuneesta virheestä, mutta pitää kuitenkin tarkemmat virheilmoitukset ohjelmamme sisällä, ettemme jaa liikaa tietoa ohjelmamme toiminnasta ja virheiden tarkemmista syistä. Liiallisen informaation jakaminen ohjelman toiminnasta voisi aiheuttaa tietoturvariskin. Aloimme iltapäivällä vielä tutkimaan lukituslaitteen ohjelmistopäivitykseen liittyvää ongelmaa. Päätimme kuitenkin, että voisi olla parasta varata tapaaminen erään kolmannen osapuolen kirjaston kehittäjien kanssa ja kysyä heiltä tarkempia tietoja asiaan liittyen. Iltapäivällä ryhdyin vielä luomaan mobiilisovellukselle näkymiä lukituslaitteen ohjelmistopäivitykselle.

Perjantai 17.11.

Opiskelin aamulla SwiftUI-viitekehyksen toimintaan liittyviä asioita ennen kuin ryhdyimme ryhmäohjelmoimaan. Päivän tavoitteena oli alkaa toteuttamaan uudenlaista toteutusta iOS SDK -paketin instanssin luomisessa. Tarkoituksemme on jakaa tiettyjä prosesseja osiin SDK-paketin alustamisessa. Tämän toteuttaminen vaati melko paljon ohjelmakoodin refaktorointia ja suurin osa päivästä kului refaktorointiin. Kirjoittamamme testit paljastivat toteutuksessamme mahdollisen muistivuodon, jonka saimme korjattua välittömästi. Loimme myös uudenlaisen julkisen virheen jonka heitämme, jos SDK-paketin alustaminen epäonnistuu.

Viikkoraportti:

Olin viime viikon kipeänä, joten tämän viikko alkoi selvittämällä, mitä on tapahtunut edellisellä viikolla. Selasin Slackin viestit läpi erityisesti niiden asioiden osalta, joita olin ollut tekemässä ennen sairastumistani. Katselin myös versionhallinnan commit-historiaa ja tutustuin tapahtuneisiin muutoksiin.

Viikon tärkeimpänä tavoitteena oli korjata esiin tullut ongelma. Asiaan liittyen meidän tulisi lähettää ongelmatilanteessa virheitä SDK-paketista. Päätimme luoda uuden virhetyypin, jonka muutamme sovelluksemme Use Case -kerroksessa SDK-paketin käyttäjiä varten toisenlaiseksi virheeksi. Aloitimme kirjoittamalla testit virheen heittämistä varten. Koska toteutuksemme ei heittänyt oikeaa virhettä, testimme eivät menneet tässä kohtaa läpi. Seuraavaksi kirjoitimme toteutuksemme heittämään oikean virheen ja läpäisimme testimme. Jouduimme muuttamaan myös toisen virhetapauksen toisenlaiseksi virheeksi Use Case -kerroksessamme. Kirjoitimme testin ja refaktoroimme toteutuksemme heittämään oikeat virheen erilaisissa tapauksissa riippuen siitä, minkä virheen saamme seuraavalta sovelluskerrokseltamme. Refaktoroimme koodimme käsittelemään tarkemmin virhetilanteet läpäistäksemme testit ja pakotimme testeissämme sovelluksen käsittelemään virheet oikein. Lisäsimme

virheillemme kuvaavat virheilmoitukset, jotka tarjoavat integraattoreille lisää tietoa virheestä. Oli hyvä lisätä virheet myös lokeihimme, jotta saamme tarkempaa tietoa virheistä ohjelman ongelmatilanteissa. (Martin 2008, 106-107.) Emme halua jakaa virheilmoituksia sovelluksen sisäisestä toiminnasta, koska tämä voisi aiheuttaa tietoturvariskin jakamalla liikaa tietoa siitä, kuinka sovellus toimii. Muuttamalla virheilmoitukset yleisempään muotoon, integraattoreiden on helpompi tulkita virhetilanteita. Lisäämällä tarkemmat virhetilanteet lokeihimme, voimme itse nähdä ohjelman sisäiset virheet ja tutkia lokejamme virhetilanteissa.

Kun olimme saaneet virheiden käsittelyn valmiiksi, siirryimme korjaamaan seuraavaa ajankohtaista ongelmaa. Integraattorimme oli kohdannut virhetilanteita SDK-pakettimme kanssa. Meidän täytyi refaktoroida SDK-pakettiamme toimimaan kolmannen osapuolen kirjaston edellyttämällä tavalla. Opin viikon aikana myös testaamaan, missä säikeessä testattavaa toiminnallisuutta suoritetaan. Kutsuimme testattavaa funktiota background-threadista ja tarkastimme, että testattava funktio kutsuu Spy-objektiamme main threadista. Testimme ei mennyt läpi ensin, koska Spy-objektimme funktiota ei suoritettu main-threadissa. Lisäsimme testattavan yksikkömme funktiolle @MainActor merkinnän ja läpäisimme testit. MainActor merkintä varmistaa, että toiminnot funktiossa tai luokassa suoritetaan main-threadissa (Kautsch 2022).

Aloin tutustumaan viikon aikana myös lukituslaitteen ohjelmistopäivityksen näkymien ja toiminnallisuuden luomiseen sovelluksessa. Tutkin sovelluksen näkymien suunnitelmia Figmassa ja aloin luomaan kokeellisia toteutuksia näkymistä. Tämä oli opettavaista, koska en ollut juurikaan keskittynyt sovelluksen kehittämiseen tämän syksyn aikana.

4 Pohdinta

Olen työskennellyt 13 viikkoa mobiilikehitystehtävissä toteuttaen iOS Mobile SDK -pakettia. Tiimini vastuualueeseen kuuluu myös iOS-mobiilisovellus, jonka parissa en juurikaan opinnäytetyöni kirjoittamisen aikana ehtinyt toimia. Prioriteettimme ovat kuluneen 13 viikon aikana olleet lähes täysin iOS SDK -pakettiin liittyvissä asioissa, minkä johdosta mobiilisovelluksen kanssa työskentely on jäänyt lähes täysin toiminnallisuuksien testaamisen asteelle. Opinnäytetyön alussa suunnitelmiini kuului uuden end to end -testausympäristön kanssa työskentely, jonka avulla voitaisiin suorittaa integraatio- ja end to end -testejä. Testausympäristössä on oikeita puhelimia, joilla testataan automatisoidusti sellaisia operaatioita, jotka ovat liian aikaa vieviä yksikkötesteihin, kuten lukituslaitteen ohjelmistopäivitystä. Testausympäristön kehittäminen on kuitenkin jäänyt kuluneen 13 viikon aikana muiden prioriteettien takia vähemmälle huomiolle.

Aloittaessani opinnäytetyöni tekemisen ei minulla ollut juurikaan kokemusta iOS-kehityksestä. Olin suorittanut edellisenä syksynä työharjoittelun samassa yrityksessä eri tiimissä, jossa toteutettiin mobiilisovellusta ja SDK-pakettia Android-sovelluksille. Vaikka pohjimmiltaan molemmissa tiimeissä luodaan saman kaltaisia tuotteita, koin iOS-kehityksen olevan hyvin erilaista Android-kehitykseen verrattuna. Alussa esimerkiksi ohjelmakoodin rakenne ja asioiden nimeämiskäytännöt vaativat paljon sisäistämistä ja opiskelua, koska en ollut tottunut vastaaviin tapoihin tehdä asioita. Myös testien kirjoittaminen erosi paljon Android-kehityksestä. Android-puolella on käytössä ulkoisia testauskirjastoja, jotka auttavat testien kirjoittamisessa ja esimerkiksi Mock-objektien luomisessa. iOS-puolella kuitenkaan tämän kaltaisia kirjastoja ei ole käytössä. Koska testeissämme ei ollut käytössä mitään ulkoisia testauskirjastoja, kirjoitimme kaikki testeihimme liittyvät osat itse. Yksikkötestien kirjoittaminen ilman ulkoisia testaus- tai Mock-kirjastoja auttoi ymmärtämään paremmin, mitä testeissä ja ohjelmakoodissa tapahtuu. Opin luomaan Spy-, Mock- ja Stub-objekteja, joilla voidaan simuloida erilaisten luokkien tai objektien toimintaa, joiden kanssa testattava yksikkömme kommunikoi. Loimme paljon erilaisia protocol defaults -toteutuksia testiympäristöömme, joissa palautimme jotakin yksinkertaista dataa

protokollien määräämillä funktioilla. Näin testeissämme Spy-objektimme ei tarvinnut aina toteuttaa kaikkia tietyn protokollan vaatimia funktioita, vaan pystyimme ylikirjoittamaan vain ne funktiot, joita testeissämme kulloinkin tarvitsimme. Opin myös luomaan erilaisia apuluokkia ja funktioita, joiden avulla voidaan testeissä luoda tarvittavia objekteja jonkinlaisilla oletusarvoilla ilman, että näitä objekteja tarvitsee jokaisessa testissä erikseen luoda itse.

Kehitimme suuren osan 13 viikon jaksosta lukituslaitteen ohjelmistopäivityksen mahdollistavaa toiminnallisuutta. Operaatio tuntui todella vaativalta ja meni useita viikkoja, ennen kuin kykenin kunnolla ottamaan osaa keskusteluun ja muistamaan, mikä luokka on vastuussa mistäkin prosessin osasta. Keskustelin asiasta esimieheni kanssa ja hän näytti minulle kaavion eräästä toiminnallisuudesta, jossa he olivat lokeroineet kaikki portit, adapterit ja muut luokat sen mukaan, mille kerroksille ne arkkitehtuurin näkökulmasta asettuvat. Esimieheni neuvoi myös luomaan itse vastaavia kaavioita, koska ne voivat helpottaa hahmottamaan suuria kokonaisuuksia paremmin. Ryhdyin luomaan itselleni kaaviota ohjelmistopäivityksen toteutuksessa käytettävistä porteista, adaptereista ja muista luokista. Tämä auttoi hahmottamaan kokonaisuutta ja sitä, kuinka eri luokat liittyvät toisiinsa sekä millä kerroksilla tehdään mitäkin asioita. Otin asian puheeksi tiimini kanssa ja työkaverini loikin Miro-sovellukseen selkeän kaavion porteista, adaptereista ja luokista. Tästä opin, että jos ohjelmakoodin lukeminen ja tulkitseminen on vaikeaa, voi visuaalisen kaavion luominen auttaa huomattavasti asioiden yhteyden näkemisessä. Näitä asioita tutkiessani opin ymmärtämään heksagonaalista arkkitehtuuria ja ohjelmamme rakennetta huomattavasti paremmin.

Olen tyytyväinen aihevalintaani koska tiesin, että äkkiä muuttuvat prioriteetit voivat vaikeuttaa johonkin tiettyyn asiaan paneutumista opinnäytetyön kirjoittamisen aikana. Huolimatta siitä, missä työskentelyn prioriteetit milloinkin ovat, oli testivetoinen kehitys aina osa työn tekoa. Prioriteettien muuttuminen on kuitenkin aiheuttanut välillä haasteita viikkotavoitteiden asettamisessa tai niiden saavuttamisessa. Viikkotavoitteita asettaessa oli hyvä tiedostaa, että viikon aikana saattaa päätyä tekemään täysin eri tehtäviä kuin mitä viikon alussa suunnitteli.

Enimmäkseen saimme työskennellä kuitenkin niiden asioiden parissa kuin alun perin suunnittelimmekin ja tavoitteiden asettaminen ja niiden saavuttaminen onnistui. Yrityksessä käytettävät arkkitehtuurilliset ratkaisut mahdollistavat testivetoisen kehityksen ja olen tyytyväinen, että tutkin sovellusarkkitehtuurin vaikutusta testattavuuteen, koska tämä auttoi minua ymmärtämään heksagonaalisen arkkitehtuurin periaatteita paremmin. Näiden arkkitehtuurillisten ratkaisujen noudattaminen, kuten erottamalla ohjelman eri osat ja business-logiikan toisistaan, ovat opettaneet minua kirjoittamaan ylläpidettävämpää ja paremmin skaalautuvaa ohjelmakoodia. Olemme joutuneet miettimään myös melko paljon sitä, kuinka integraattorit käyttävät SDK-pakettiamme ja kuinka teemme sen heille mahdollisimman helpoksi. SDK-paketin julkisten funktioiden nimeäminen ja dokumentaation luominen on opettanut nimeämään asioita selkeämmin ja jakamaan tietoa muille kehittäjille mahdollisimman yksinkertaisesti ja tehokkaasti.

Olen kehittynyt tiimityöskentelyssä ja sisäistänyt paremmin yrityksen toimintatapoja. Alussa minulla oli hieman vaikeuksia tuoda esiin, kuinka voin mitata kehittymistäni pyrkiessäni kohti tavoitteitani. Otin henkilökohtaiseksi tavoitteekseni käyttää aikaa ryhmätyöskentelyn lisäksi myös opiskeluun. Ryhmässä ohjelmoitaessa työtahti oli välillä sen verran nopea, ettei asioiden sisäistämiseen jäänyt työtä tehdessä välttämättä aikaa. Toisaalta ryhmässä ohjelmointi mahdollistaa asioista kysymisen aina kun kysyttävää ilmenee. Joidenkin asioiden sisäistäminen kuitenkin saattaa viedä hieman enemmän aikaa. Näissä tilanteissa saatoin jättäytyä pois ryhmätyöskentelystä ja alkaa yrittää luoda vastaavanlaista toteutusta omassa ohjelmointiympäristössäni. Opiskeltuani syvällisemmin kulloinkin käsillä olevia asioita huomasin kykeneväni ottamaan enemmän osaa keskusteluun ja tarjota enemmän omia ideoitani ratkaisuja miettiessämme. Koen kehittyneeni ammattilaisena ja tiimityöskentelijänä kohti tavoitteitani opinnäytetyöni aikana. Kykenen tulkitsemaan paremmin ohjelmakoodia ja sovelluksen toimintaa itsenäisesti ja etsimään ratkaisuja perustason ongelmiin. Myös opinnäytetyössäni käyttämä kirjallisuus on auttanut asioiden opiskelussa ja ymmärtämisessä. Olen huomannut, että työpaikallani käytetään paljon saman kaltaisia oppeja, joita valitsemissani kirjoissa käsitellään. Kirjallisuus on tarjonnut myös lisätietoa tietyistä aihealueista ja auttanut minua erilaisten ratkaisujen kehittämisessä.

Ryhmäohjelmoitaessa vaihdamme koodia kirjoittavaa henkilöä aika ajoin. Opinnäytetyöni alkuvaiheessa vaihdoimme koodia kirjoittavaa henkilöä noin kahden tunnin välein. Vaikka aluksi minulle oli kehittävämpää katsoa enemmän vierestä, kun muut kirjoittivat koodia, koin muutaman viikon jälkeen kaipaavani enemmän koodin kirjoittamista oppimiseni tueksi. Otin asian puheeksi tiimini kanssa ja päädyimme käyttämään Pomodoro-mallia työn tekemisen rytmittämiseen. Sovimme koodausvuoron kestävän 25 minuuttia, jonka jälkeen pitäisimme viiden minuutin tauon. Tämän jälkeen vaihtaisimme koodin kirjoittajaa ja voisimme vaihtaa vuoroja nopeammissa sykleissä. Ratkaisu oli sekä omastani, että tiimikavereidenkin mielestä toimiva. Koen, että liian pitkään kerralla koodin kirjoittajana oleminen on puuduttavaa siinä missä liian pitkään koodin kirjoittamisen katsominenkin on, ja tämän kaltainen työskentelymalli lisää tehokkuuttani. Myös lyhyet tauot ovat parantaneet jaksamistani työpäivän aikana. Erilaisten työskentelytapojen kokeileminen auttoi hahmottamaan itselleni parhaiten sopivia tapoja tehdä töitä.

Sain työsopimukselleni jatkoa opinnäytetyöni kirjoittamisen aikana ja jatkan yrityksessä samoissa työtehtävissä, kuin mitä tein opinnäytetyöni kirjoittamisen aikana. 13 viikon työskentelyjaksoni aikana oppimani asiat ovat luoneet hyvän pohjan, josta jatkaa työn tekemistä ja ammattilaisena kehittymistä. Kykenen toimimaan osana tiimiä ja osallistumaan keskusteluun ja ratkaisujen kehittämiseen. Kykenen myös tulkitsemaan paremmin yrityksen ohjelmakoodia omaaloitteisesti ja pohtimaan omia ratkaisuja, joita voin ehdottaa ja soveltaa yhdessä tiimini kanssa työskennellessä.

Lähteet

Beck, K. 2003. Test Driven Development. Boston: Addison-Wesley.

Dominik, H. 2022. Test-Driven iOS Development with Swift - Fourth Edition. Birmingham: Packt Publishing Ltd.

Giordano, S. & Edgar, N. 2021. SwiftUI Cookbook - Second Edition. Birmingham: Packt Publishing Ltd.

Kautsch, A. I. 2022. Modern Concurrency on Apple Platforms: Using async/await with Swift. New York: Apress

Khorikov, V. 2020. Unit Testing Principles, Practices, and Patterns. New York: Manning Publications Co.

Manikandan, S. 2023. Test Automation Engineering Handbook. Birmingham: Packt Publishing Ltd.

Martin, R. C. 2008. Clean Code: A Handbook of Agile Software Craftmanship. Boston: Pearson.

Martin, R. C. 2017. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Boston: Pearson.

Vilmart, F., Scalzo, G. & De Simone, S. 2018. Hands-On Design Patterns with Swift. Birmingham: Packt Publishing Ltd.