Lauri Riikonen

Android-sovelluksen migraatio Jetpack Composeen

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

17.3.2024

Tiivistelmä

Tekijä:	Lauri Riikonen
Otsikko:	Android-sovelluksen migraatio Jetpack Composeen
Sivumäärä:	37 sivua
Aika:	17.3.2024
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Mobile Solutions
Ohjaajat:	Lehtori Peter Hjort

Insinöörityössä toteutettiin Android-sovelluksen migraatio käyttämään Googlen suositeltuja Android-sovelluskehitysohjeistuksia. Olemassa olevan sovelluksen käyttöliittymä muunnettiin hyödyntämään modernia deklaratiivista Android-ohjelmistokehystä nimeltä Jetpack Compose.

Työssä perehdyttiin niihin relevantteihin vaiheisiin, jotka mahdollistavat migraation toteuttamisen. Näitä vaiheita olivat mm. yleisen ymmärryksen hankkiminen Jetpack Composen toiminnasta. Sen lisäksi perehdyttiin myös siihen, mitä tarkoitetaan design systemsillä, kun puhutaan erilaisten sovellusten käyttöliittymäkehityksestä.

Työssä käytiin lävitse niitä erinäisiä vaiheita, joita tulee toteuttaa itse sovelluksen muuntamisvaiheessa. Näihin sisältyi mm. yksittäisten ruutujen muuntaminen moderniin muotoon. Sen lisäksi käsiteltiin myös sovelluksen navigointilogiikan muuttamista.

Lopputulemana saatiin päivitetty sovellus, josta saatiin poistettua merkittävä määrä koodia, kun erittäin suuri osa XML-tiedostoista poistettiin kokonaan sovelluksesta. Poistamatta jääneet XML-tiedostot toimivat vain resursseina sovelluksessa. Tarvittava koodin määrä saatiin huomattavasti pienemmäksi ja luettavammaksi Jetpack Composen deklaratiivisen luonteen vuoksi.

Edut, joita sovelluksen migraatio toi, liittyvät sovelluksen tulevaisuudessa tehtävään kehitykseen ja ylläpitoon. Koska Compose on Android-kehityksessä keskeisessä asemassa ja teknologia, johon Google panostaa vahvasti, on sovelluksen tulevaisuus teknologisesta näkökulmasta turvattuna, koska merkittävät deprekaatiot (ohjelmointirajapintojen vanhentumiset) eivät ole todennäköisiä Composelle lähitulevaisuudessa.

Avainsanat:	Android, Jetpack Compose, Design systems, Sovelluksen Migraatio

Abstract

Author:	Lauri Riikonen
Title:	Android application migration to Jetpack Compose
Number of Pages:	37 pages
Date:	17 March 2023
Degree:	Bachelor of Engineering
Degree Programme:	Information Technology
Professional Major:	Mobile Solutions
Supervisors:	Peter Hjort, Lecturer

In the thesis, an Android application migration was implemented to adhere to Google's recommended Android app development guidelines, utilizing a modern declarative Android software framework called Jetpack Compose.

The work involved familiarizing with the relevant phases that enable the implementation of migration in the initial phase. These phases included gaining a general understanding of Jetpack Compose's functionality. Additionally, the concept of design systems in the context of various application user interface development was explored.

The work also covered various steps to be implemented during the application transformation phase. These included converting individual screens into modern format, as well as addressing the transformation of the application's navigation logic.

As a result, an updated application was obtained, removing a significant amount of code, as a large portion of XML files were completely removed from the application, except for some individual files that served only as resources. The amount of required code was also significantly reduced and made more readable due to Jetpack Compose's declarative nature.

Other advantages brought by the application migration are related to future development and maintenance of the application. Since Compose holds a relevant position in Android development and is a technology heavily invested in by Google, the future of the application is secured from a technological perspective, as significant deprecations are unlikely for Compose in the near future.

Keywords: Android, Jetpack Compose, Design systems, Application migration

Sisällys

1 Johdanto

Insinöörityö tehtiin eräälle finanssialan toimijalle, ja Android-sovellus, jota työssä muokattiin, on eräänlainen tunnistautumissovellus. Sovelluksen avulla toimijan asiakkaat voivat toteuttaa erilaisia tunnistautumista vaativia prosesseja, kuten verkkopankkiin tunnistautumista tai korttimaksujen vahvennuksia.

Työssä itsessään ei pyritä muuttamaan sovelluksen liiketoimintalogiikkaa, vaan työn tavoitteena on päivittää sovelluksen käyttöliittymäkerros vastaamaan modernin Android-kehityksen suosituksia. Työssä XML-perusteinen (*Extensible Markup Language)* implementaatio tullaan poistamaan ja korvaamaan Googlen uudella suosittamalla ohjelmistokehyksellä nimeltä Jetpack Compose. XML:n toimintaperiaate perustuu tiedon rakenteelliseen kuvaamiseen merkintäkielen hierarkiassa, kun taas Jetpack Compose tarjoaa modernin lähestymistavan käyttöliittymien luomiseen suoraan Kotlin-koodissa deklaratiivisesti. Näin mahdollistetaan sovelluksen tehokkaampi kehitys ja parempi ylläpito myös tulevaisuudessa.

Työn aikana tullaan tutustumaan niihin erinäisiin vaiheisiin, joita tulee suorittaa, jotta sovelluksen migraatio voidaan mahdollistaa. Työssä tullaan perehtymään mm. siihen, kuinka Jetpack Compose toimii pääpiirteittäin ja mitä tarkoitetaan käsitteellä design systems (suunnittelujärjestelmä, eräänlainen komponenttikirjasto) ja kuinka se liittyy käyttöliittymäkehitykseen. Työssä tullaan paneutumaan myös siihen, kuinka design systemsin mukaisia ohjeistuksia tullaan käyttämään, kun tehdään Android-kehitystä käyttäen Jetpack Composea.

Työn edetessä etsitään myös ratkaisuja siihen, kuinka itse sovellukselle tullaan toteuttamaan migraatio. Seikkoja, joita tullaan selvittämään ovat esimerkiksi, kuinka Compose voidaan ottaa käyttöön jo olemassa olevassa sovelluksessa, jossa sitä ei tueta. Ratkaisuja tullaan etsimään myös muihin yleisiin ongelmiin, kuten siihen, miten näkymät käytännössä kirjoitetaan ja kuinka design

systemsiä hyödynnetään sovelluksen migraation yhteydessä ja kuinka sovelluksen navigointi tullaan toteuttamaan modernein standardein.

2 Android käyttöliittymäkehitys

2.1 Vanha kehitystapa

Historiallisesti katsottuna Androidin käyttöliittymä-kehitys on perustunut imperatiiviselle ohjelmointiparadigmalle, mikä tarkoittaa erilaisten käyttöliittymä-pienoisohjelmien eli widgettien esittämistä sen mukaan, kuinka ne on määritelty niin kutsuttuun View-hierarkiaan [1]. Tyypillisesti näiden komponenttien päivitys tapahtui kutsumalla erilaisia funktioita, kuten findViewById() tai button.setText() jotka muokkaavat esimerkiksi widgetin kuvaa tai tekstisisältöä XML-perusteisessa sovelluksessa.

View'ien manuaalinen päivittäminen on kuitenkin virhealtista, ja siinä saattavat yhdistyä ohjelmoijan huolimattomuusvirheet sekä erilaiset virheilmoitukset, jotka syntyvät, kun käyttöliittymä-päivitykset ovat määrittelemättömästä syystä konfliktissa keskenään.

XML-kehitystapa kärsii myös siitä, että Googlen luomat uudet Android-kehitykseen liittyvät ominaisuudet eivät päädy enää XML-perusteiselle alustalle tai päätyminen tapahtuu rajoitetun tuen kera.

Potentiaalisia ongelmia voivat myös olla XML-tyylin deprekoituminen, jolla tarkoitetaan ohjelmiston merkitsemistä vanhentuneeksi ja että sen käyttöä ei enää suositella. Tämä tarkoittaa, että vaikka ominaisuus tai toiminto voi toimia nykyisessä versiossa, sitä ei enää kehitetä, ja se saatetaan poistaa tulevista versioista tulevaisuudessa, vaikkei se kovin todennäköistä olekaan.

2.2 Kehitys käyttäen Jetpack Composea

Jetpack Compose on Googlen ja ohjelmistokehittäjien keskuudessa erittäin tunnetun yrityksen JetBrainsin (mm. IntelliJ Idean, WebStormin sekä Kotlin-kielen luoja) yhdessä kehittämä moderni ohjelmistokehys.

Mobiilikehityksessä on jo jonkin aikaa kuljettu kohti deklaratiivista ohjelmointiparadigmaa, jonka suuntaa alettiin viitoittamaan ensimmäisen kerran jo useampia vuosia sitten web-kehityksessä Metan luoman React-kirjaston yhteydessä [2]. Mobiilikehityksessä alustariippumattomat teknologiat kuten React Native [3] ja Flutter [4] omaksuivat tämän paradigman ensimmäisenä, mutta sittemmin sekä Apple SwiftUI:lla [5] että nyt myös Google Jetpack Composella [6] ovat siirtymässä tähän paradigmaan.

Composella on useita keskeisiä etuja, jos verrataan aiempaan käyttöliittymä-kehitystapaan, jossa XML-tiedostojen käyttö oli erittäin merkitsevässä asemassa ja joka samalla toi omat haasteensa, jotka osaltaan ovat vaikuttaneet siihen, että Composen kaltainen ratkaisu on tullut tarpeelliseksi. Tutkielma keskittyy pääasiassa Composella tehtävään kehitykseen, mutta esimerkkejä XML-tiedostoja käyttävään tapaan ilmenee myös melko runsaasti kontekstin luomista varten.

Huomionarvoista on myös mainita, että Compose toimii vain Kotlin-ohjelmointikielen kanssa, vaikka Android-kehityksessä Java on historiallisesti ollut myös erittäin merkityksellisessä asemassa. Kotlin-kielen tuomat modernit ominaisuudet ja Kotlinin Googlelta saama status suositeltuna Android-alustan kehityskielenä ovat vaikuttaneet asiaan.

Kun Android ensimmäisen kerran esiteltiin yli kymmenen vuotta sitten, sovellusten rakenteet olivat täysin erilaiset ja ne olivat helpommin rakennettavissa, koska silloiset yksinkertaiset sovellukset vaativat vain XML-tiedoston, joka liitetään Android-maailmassa tunnettuun Activity-komponenttiin. Tuolloin myös laitteiden määrä, jota tarvitsi tukea, oli huomattavasti vähäisempi. Nykyisin on kuitenkin useita erilaisia seikkoja, joita täytyy ottaa huomioon

Android-sovelluskehityksessä kuten esimerkiksi erilaisten näyttöjen koot tai pikseleiden tiheys. [7.]

Ajan myötä, kun laitteiden määrä ja tuettavien ominaisuuksien määrä kasvoi, paine toisenlaisen ratkaisun löytämiseen nousi suureksi, kun vanha imperatiivinen ohjelmointiparadigma ei enää ollut tehokas ratkaisu [7]. Tämän lisäksi jotkin kriittiset APIt (sovellusrajapinta, application interface, API) ja niiden päivittämisen sitominen uusien Android-versioiden julkaisemiseen ei ollut enää tehokas ratkaisu [8]. Tämän seurauksena kehitettiin Jetpack Compose. Se on Androidin uusi moderni ohjelmistokehys, joka tuo useita etuja Android-kehitykseen käyttäen lähestymistapana deklaratiivista ohjelmointiparadigmaa.

Composen etuina voidaan ensimmäisenä mainita selkeästi vähentynyt koodin tarve, joka voi joissain tapauksissa olla moninkertaisesti pienempi kuin vastaavan toiminnallisuuden saavuttaminen XML-tiedostoja käyttävään käyttöliittymäkehitystapaan verrattuna.

Toinen selkeä etu on Composen intuitiivisuus. Composen intuitiivisuus perustuu jo edellä mainitulle deklaratiiviselle ohjelmointiparadigmalle, joka käytännössä tarkoittaa pienten UI-komponenttien luomista, joita on helppo ylläpitää, päivittää ja uudelleen käyttää sovellusten kehityksessä.

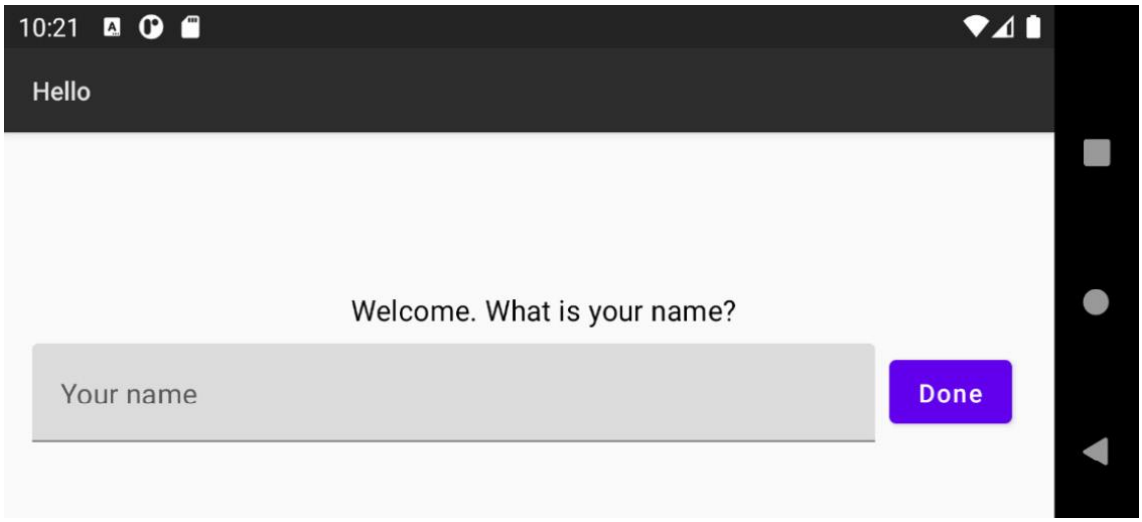
Compose on myös täysin yhteensopiva jo olemassa olevien koodikantojen kanssa, joten sen käyttöönottaminen ei vaadi suuria toimenpiteitä ja näin ollen mahdollistaa matalan kynnyksen käyttöönoton Composesta kiinnostuneille Android-kehittäjille. [9.]

2.3 Käyttöliittymän luonnin perusteet

Composen deklaratiivinen toiminnallisuus perustuu siihen, että käyttöliittymä rakennetaan aina uudelleen, kun jokin sen osa vaatii päivittämistä. Ongelmat, joita tämä lähestymistapa saattaa kuitenkin aiheuttaa, liittyvät esimerkiksi uudelleen rakennuksen nopeuteen tai esimerkiksi laskentatehon ja akun käytön mahdollisiin komplikaatioihin. Compose osaa kuitenkin älykkäästi päättää, mitkä

osat käyttöliittymästä tulee uudelleen renderöidä. Tätä prosessia kutsutaan uudelleen suorittamiseksi (Recomposition). [10.]

Kuvassa 1 havainnollistetaan yksinkertaisella esimerkillä, kuinka käyttöliittymärakennus toimii luomalla pieniä composable-funktioita, jotka esittävät jotakin tiettyä osaa käyttöliittymästä.



Kuva 1. Yksinkertainen Compose UI. [7.]

Yllä oleva käyttöliittymä koostuu kolmesta eri composable-funktiosta. Käyttöliittymän osat eriteltyinä: Funktio, joka esittää tekstiä (”Welcome...”). Funktio, joka esittää tekstikenttää, johon käyttäjä voi syöttää tekstiä (”Your name”). Funktio, joka esittää painiketta (”Done”), joka reagoi käyttäjän painallukseen. [7.]

Koodissa composable-funktiot tulee määritellä @Composable-annotaatiolla, jolla ne tunnistetaan nimenomaan käyttöliittymää luoviksi funktioiksi (esimerkkikoodi 1).

```
@Composable
fun Welcome() {
  Text(
    text = stringResource(id = R.string.welcome),
    style = MaterialTheme.typography.subtitle1
  )
}
```

Esimerkkikoodi 1. Yksinkertaisen tekstin palauttava composable-funktio.

Tämä Welcome-funktio palauttaa tekstiä (Text), joka on Compose-kirjastoon sisään rakennettu funktio, ja sen käyttöönotto ei vaadi muuta kuin oikean paketin tuontia tiedostoon, jota sillä hetkellä käsitellään. Funktion parametrit määrittelevät, minkälaisen tekstin ja tyylin Compose luo näytettävälle tekstille siinä vaiheessa, kun funktio renderöidään näytölle. Muunlaiset komponentit toimivat samalla periaatteella, mutta niiden rakenne ja parametrit ovat erilaiset. [7.]

Muita mahdollisia muunnettavia composable-funktion ominaisuuksia ovat esimerkiksi fonttikoko, tekstin väri tai rivien korkeus. Näiden lisäksi muita hieman erilaisia ominaisuuksia ovat esimerkiksi kulmien pyöreys, jos kyseessä on vaikkapa neliötä esittävä composable-funktio, jonka kulmat on haluttu pyöristää.

2.4 Komponentin uudelleen suorittaminen

Kuten luvussa 2.3 mainittiin, Composen tapa päivittää käyttöliittymää perustuu komponenttien uudelleen suorittamiselle, jossa UI rakennetaan uudelleen, kun jotakin käyttöliittymän osaa tarvitsee päivittää. Käyttöliittymän kulloinenkin rakenne perustuu senhetkiselle datalle, jota composable-funktioille on syötetty. Esimerkiksi jos Text composable -funktion text-parametrin arvo muuttuu, aiheuttaa tämä komponentin uudelleen suorittamisen uuden datan mukaisesti.

Compose mahdollistaa myös ehdollisen renderöinnin esimerkiksi tavallisen Kotlin-koodin mukaisten if- tai when-ehtolausekkeiden avulla. Näitä voidaan hyödyntää mm. tilanteissa, joissa halutaan määritellä, mikä tietty composable-funktio halutaan renderöidä tai vaikkapa jos ohjelmoija haluaa antaa ehdollisia parametrejä composable-funktioille. [7.]

Seuraavat seikat on syytä pitää mielessä kun komponentteja uudelleen suoritetaan:

Composable-funktiot voidaan suorittaa satunnaisessa järjestyksessä:

```
     @Composable
     fun ButtonRow() {
       MyFancyNavigation{
          StartScreen()
           MiddleScreen()
           EndScreen()
    }
}
```

Esimerkiksi tässä tapauksessa ei voida olettaa, että nämä composable-funktiot suoritettaisiin niiden koodissa kirjoitetussa järjestyksessä. Tästä syystä esimerkiksi StartScreen() ei voi alustaa globaalia muuttujaa, jota MiddleScreen() tai EndScreen() hyödyntäisi omissa funktioissaan. Useita composable-funktiota voidaan suorittaa myös samanaikaisesti.

Komponenttien uudelleen suorittaminen ohittaa mahdollisimman monta composable- ja lambda-funktiota. Tämä tarkoittaa yksinkertaisesti sitä, että Compose pyrkii päivittämään vain ne asiaankuuluvat käyttöliittymän osat, jotka tarvitsevat uudelleen suorittamista.

Komponenttien uudelleen suorittaminen on ns. optimistinen, mikä tarkoittaa, että Compose olettaa komponenttien suorittamisen päättyvän, ennen kuin funktioille syötetään uutta dataa. Jos kuitenkin tapahtuu niin, että data muuttuu ennen kuin komponenttien uudelleen suorittamien on saatu loppuun, saattaa uudelleen suorittaminen peruuntua ja alkaa alusta uudella datalla. Optimistinen uudelleen suorittaminen on kuitenkin se tapa, johon ohjelmoijan olisi syytä pyrkiä. [10.]

Composable-funktioita voidaan suorittaa todella useasti, esimerkiksi animointien yhteydessä [10]. Näissä tapauksissa komponentin uudelleen suorittaminen tapahtuu esimerkiksi, kun animoidaan vaikkapa jonkin komponentin alkuperäistä korkeutta, leveyttä tai vaikkapa väriä uuteen arvoon.

2.5 Tilan hallinta

Composeen liittyy myös tärkeä seikka, jota ei olla käsitelty aiemmin, jotta Composen perusidean ymmärtäminen olisi helpompaa. Tämä konsepti on niin kutsuttu state eli tila Compose-sovelluksissa. Tila vastaa mitä tahansa arvoa, kuten muuttujan arvoa tai sisäänrakennetun tietokannan tilaa, ja on tosiasiallisesti vastuussa komponentin uudelleen suorittamisen käynnistämisestä [11].

Yleisesti käytetty muuttuja on nimeltään MutableState. Tämäntyyppisen muuttujan arvon muuttaminen aiheuttaa komponenttien uudelleen suorittamisen [12]. Myös toisenlaisia muuttujia tilan hallintaan löytyy, kuten LiveDatan observointi tai Flowien kerääminen, ja näiden muuttujien arvojen perusteella komponenttien uudelleen suorittamisen käynnistäminen [11].

Jotta muuttujan arvo säilyy muistissa uudelleen suorittamisten välillä, tarvitaan vielä toinenkin hyödynnettävä ominaisuus. Tätä varten on olemassa niin kutsuttu remember-API. Arvo, jonka remember saa, tallennetaan muistiin ja palautetaan uudelleen suorittamisen yhteydessä eli tosiasiassa remember vastaa arvon tallentamisesta. Yhdistämällä MutableStaten ja remember-APIn tarjoama toiminnallisuus saadaan aikaiseksi toimiva tilan hallinta composable-funktioiden muuttujille.

Seuraavassa ovat tavat, joilla voidaan alustaa muuttujat hyödyntäen MutableStatea ja remember-APIa:

- val mutableState = remember { mutableStateOf(default) }
- var value by remember { mutableStateOf(default) }
- val (value, setValue) = remember { mutableStateOf(default)} [11.]

Tilan hallinnassa nousevat usein esille erilaiset tilanteet, joissa tietyn tilan arvo kannattaa hyvän käytännön mukaisesti nostaa mahdolliselle toiselle tasolle (state hoisting). Tällaiset tilanteet voivat liittyä joko käyttöliittymälogiikkaan tai liiketoimintalogiikkaan. Yleinen paikka, jossa voidaan hallita tilaa, on esimerkiksi ViewModel-luokka [13].

ViewModel on luokka, jolla mallinnetaan käyttöliittymässä näytettävää dataa, ja tästä syystä luokka onkin saanut nimen *ViewModel.* ViewModelin mahdollisiin tehtäviin kuuluu esimerkiksi:

1. käyttöliittymän muuttujien arvojen alustaminen, hallinta ja säilyttäminen
2. datan pyytäminen erilaisista lähteistä, kuten palvelimilta
3. datan käsittely sellaiseen muotoon, että se voidaan esittää käyttöliittymässä
4. käyttäjän toimintoihin reagoiminen ja tilojen asianmukainen muuttaminen. [14.]

ViewModelia ei tallennetta compositionin osana, joten siellä pidetyt arvot eivät muutu komponenttien uudelleen suorittamisen yhteydessä. Näin saavutetaan vastaava toiminnallisuus kuin MutableStaten ja rememberin käyttöllä. Tyypillisesti ViewModel-luokkaa käytetään, kun halutaan, että composable-funktiot pääsevät käsiksi sovelluksen liiketoimintalogiikkaan.

On sovelluskohtaista, minkälainen lähestymistapa tilan mahdolliselle nostamiselle otetaan käyttöön. [13.]

3 Design systems käyttöliittymäkehityksessä

3.1 Design systems yleisesti

Kun yritykset kasvavat, ne joutuvat ratkaisemaan entistä monimutkaisempia käyttäjäongelmia useilla eri alustoilla, kuten mobiilisovelluksissa ja verkkosivustoilla. Tämä johtaa yleensä siihen, että tiimit keskittyvät tiettyihin tuotteen osa-alueisiin, kuten etusivuun, käyttäjätilien hallintaan tai jonkinlaiseen hakutoimintoon. Jokainen yksittäinen tiimi kehittää omia laatuvaatimuksiaan, jotta tiimit voisivat työskennellä nopeammin. Tämä kuitenkin yleisesti heikentää eri alustojen yhtenäistä designia ja saattaa johtaa siihen, että tuotteiden

eroavaisuudet huomataan vasta siinä vaiheessa, kun niitä tarkastellaan yhdessä. [15.]

Ongelma muodostuu myös siitä, että eri sidosryhmillä on omat osuutensa hoidettavana, jotta voidaan saada aikaan miellyttävä käyttäjäkokemus tuotteelle. Ryhmistä voidaan mainita seuraavat: designerit, frontend-kehittäjät sekä tuotepäällikkö.

Designereiden eli sunnittelijoiden, tehtäviin kuuluu esimerkiksi kuvankaappauksien ottaminen ja erilaisten artboardien jatkuva jakaminen edes takaisin, ja niiden samanaikainen päivittäminen. Työssään designerit joutuvat myös selittämään samoja valintoja ja ratkaisuja kerta toisensa jälkeen muille sidosryhmille. Tämän lisäksi tarvittavien tiedostojen etsiminen muuttuu haastavaksi ja lisäksi pitäisi myös muistaa kuka työskenteli minkäkin asian parissa. Ongelma, joka nousee myös esille designereilla, on tunne siitä, kuinka he eivät todellisuudessa kykene vaikuttamaan tuotteen etenemissuunnitelmaan.

Frontend-kehittäjien tehtäviin kuuluu samanlaisten koodiratkaisujen kehittäminen erilaisiin konteksteihin. He yrittävät antaa palautetta design-tiimille, mutta usein tuntuu siltä, että osapuolet eivät ymmärrä toisiaan. Frontend-kehittäjillä ei ole pääsyä design-tiedostoihin, eikä heillä ole ymmärrystä, minkä takia design-tiimi on päätynyt ratkaisuihinsa. Omassa työssään Frontend-kehittäjät havaitsevat epäjohdonmukaisuuksia designissa, ja koodista tulee sotkuista.

Tuotepäälliköllä on paljon painetta sekä muilta sidosryhmiltä että käyttäjiltä. Hänen tehtäviinsä kuuluu mm. käyttäjien tarpeiden ymmärtäminen sekä vision luominen tuotteelle. Tuotepäällikön ongelma koostuu siitä, että käytännössä tuntuu siltä, että ei ole tarpeeksi designereita ja kehittäjiä, jotta kaikki tarvittava työ saataisiin tehtyä. Ongelmaksi muodostuu myös se, että syntyy tunne siitä, kuinka design-tiimin toivottaisiin antavan enemmän palautetta tuotteen etenemissuunnitelmasta ja näkymästä, mutta se fakta tiedostetaan, ettei heillä ole aikaa sille. [16.]

Tämän ongelman ratkaisua varten monet firmat ovatkin usein päätyneet sellaiseen ratkaisuun, jossa pyritään suunnittelemaan design ja luomaan sen pohjalta uudelleen käytettäviä komponentteja. Tätä modulaarista rakennustapaa on totuttu kutsumaan nimellä design systems.

Käytännössä desing systemsillä tarkoitetaan uudelleen käytettäviä käyttöliittymän osia, jotka jaetaan tiimien välillä ja jotka kootaan yhteen tuotteiden rakentamista varten. Ohjeistus komponenttien käyttöä varten kirjataan dokumenttiin, joka sisältää määritelmät siitä, kuinka komponentteja tulisi käyttää. Tämän lisäksi design systemsiin kuuluu taustalla olevat design-periaatteet, säännöt sekä ohjenuorat, joilla tiimit voivat luoda koherentteja käyttäjäkokemuksia. [15.]

3.2 Komponentin luominen

Myös Android-sovellusten on tarpeen seurata edellä mainitun kaltaisia määritelmiä, oli kyse sitten XML-perusteisesta tai Jetpack Compose -perusteisesta komponentista. Seuraavassa tullaankin käymään lävitse hieman tarkemmin, kuinka yksittäinen komponentti voidaan rakentaa Composea käyttäen seuraten design-tiimin asettamia vaatimuksia.

Esimerkiksi valikoitunut komponentti on hyvin yksinkertainen, mutta se havainnollistaa hyvin sitä, kuinka Compose-perusteiset komponentit voidaan luoda varsin pienellä määrällä koodia, joka on samalla myös erittäin helposti ymmärrettävissä, eikä sisällä useita tiedostoja, jotka todennäköisesti tarvittaisiin vastaavanlaisen XML-komponentin luomista varten.

Composen ehdottomia etuja onkin se, kuinka vaivatonta composable-funktioiden uudelleenkäyttö on, ja se korostuu myös silloin, kun luodaan uudelleen käytettäviä design systemsin mukaisia komponentteja. Hyvä lähestymistapa onkin luoda tietynlainen perusrakenne komponenttia esittävälle composable-funktiolle, jota voidaan käyttää uudelleen toisten komponenttien luomista varten.

Esimerkkikoodissa 2 luodaan perusrakenne tietynlaiselle painikkeelle, jota voidaan hyödyntää toisten composable-funktioiden toimesta.

```
@Composable
private fun BaseButton(
          modifier: Modifier,
          onButtonClicked: () -> Unit,
          text: String,
          enabled: Boolean = true,
          style: TextStyle,
          contentPadding: Dp,
          textColor: Color = Color.Unspecified,
          backgroundColor: Color,
          disabledBackgroundColor: Color,
          rippleTheme: RippleTheme
){
CompositionLocalProvider(LocalRippleTheme provides rippleTheme) {
     Button(
          modifier = Modifier,
          onClick = { onButtonClicked() },
          enabled = enabled,
          colors = ButtonDefaults.buttonColors(
               backgroundColor = backgroundColor,
               disabledBackgroundColor = disabledBackgroundColor
),
shape = RectangleShape,
contentPadding = PaddingValues(
horizontal = contentPadding),
elevation = ButtonDefaults.elevation(
          defaultElevation = 0.dp,
          pressedElevation = 0.dp,
          disabledElevation = 0.dp
)
) {
     Text(
     text = text,
     style = style,
     color = textColor,
     maxLines = 1,
     overFlow = TextOverFlow.Ellipsis
)
}}}
```

Esimerkkikoodi 2. Painiketta esittävä BaseButton composable-funktio, jota on helppo uudelleenkäyttää toisten vastaavien komponenttien luontia varten.

Luodun komponentin perusta on hyvin selkeä ymmärtää, ja ei ole mielekästä käydä kaikkia sen parametreja erikseen lävitse, koska ne ovat hyvin loogisesti ymmärrettävissä ilman kummempaa tarkennusta. Huomioimisen arvoista on kuitenkin mainita se, että kaikilla parametreilla pyritään tyydyttämään design systemsin mukaiset edellytykset, kuten esimerkiksi painikkeeseen sisältyvän tekstin väri tai painikkeen taustaväri. Kuitenkin komponentissa on pari sellaista

toiminnallisuutta, jotka eivät ole itsestään selviä, ja toiminnallisuuksia onkin hyvä hieman tarkentaa, koska toiminnallisuudet ovat oleellisia tekijöitä kehittämiselle, kun käytetään Composea.

Ensimmäinen näistä on Modifier. Modifier on nimensä mukaisesti ominaisuus, jonka avulla voidaan muokata composable-funktiota erilaisilla tavoilla. Muokattavia ominaisuuksia voivat olla esimerkiksi komponentin koko, käyttäytyminen ja yleinen ulkoasu. Näiden lisäksi modifierin avulla voidaan komponentille lisätä erilaisia saavutettavuuteen (accessibility) liittyviä ominaisuuksia ja prosessoida käyttäjän syötettä sekä lisätä komponentin vuorovaikutusta, kuten tekemällä komponentti napautettavaksi, vieritettäväksi, vedettäväksi tai vaikkapa zoomattavaksi. Modifierit ovat standardin mukaisia Kotlin-luokkia. [17.]

Toinen oleellinen seikka on komponentin koodin sisältä löytyvä Button(), joka itseasiassa on Jetpack Compose -ohjelmistokehyksen sisään valmiiksi rakennettu komponentti, jonka ominaisuuksia on helppo muokata. Todellisuudessa tämä itse luotu peruskomponentti jo itsessään perustaa suuren osan toiminnallisuudestaan Composeen sisäänrakennetun komponentin toiminnallisuudelle, eikä tässä tapauksessa olekaan tarvetta ruveta luomaan tyhjästä täysin uutta omaa painiketta esittävää composable-funktiota.

Lähestymistavasta, jossa käytetään jo Composeen sisäänrakennettua funktiota on olemassa muitakin käytännön etuja, koska monet hieman enemmän aikaa vaativat asiat ovat jo valmiiksi huomioituna, kuten saavutettavuuteen liittyvät seikat. Tästä voidaan esimerkkinä antaa se, että komponentti on fokusoitavissa sekä sen lisäksi tunnistettavissa painikkeeksi erilaisten näytönlukijoiden kuten Talkbackin toimesta [18].

Kun peruskomponentti painikkeelle on olemassa, sen pohjalta on helppoa rakentaa uusia komponentteja design systemsin vaatimusten perusteella. Esimerkiksi painikkeista voikin olla vaatimuksia, jotka edellyttävät eri värisiä sekä kokoisia painikkeita. Painikkeiden toteuttaminen onkin helppoa, kun luodaan esimerkiksi *primääri* tai *brändi*- painike, jotka eivät vaihtele muulta kuin

värityksiensä osalta. Sen lisäksi primääri tai brändi painikkeista voi olla esimerkiksi pieniä, keskikokoisia tai suuria variaatioita.

Uusi komponentti voidaan luoda nimettynä esimerkiksi BrandButtoniksi, joka kutsuu BaseButton() composable-funktiota, ja antaa sille designin systemsin mukaiset parametrit, kuten painikkeen taustavärin sekä korkeuden ja leveyden jo edellä mainitun modifier-parametrin kanssa. Näin ollen saadaan luotua komponentti, jota on erittäin helppo uudelleen käyttää ja mukauttaa, kunhan BrandButton() itsessään jätetään sellaiseksi, että sille annettavat parametrit ovat muokattavissa.

Hyvä lähestymistapa onkin käyttää Kotlin-kielen ominaisuutta, joka mahdollistaa sen, että funktion parametreille voidaan antaa tietyt oletusarvot, joiden perusteella komponentti luodaan [19]. Näin ollen esimerkiksi BrandButton() voi hyödyntää design systemsin mukaista väritystä oletus-parametrina samalla mahdollistaen täysin erilaisen värin syöttämistä, jos ikinä sattuu tulemaan tilanne, jossa tarve erilaisen värin tukemiselle on tarpeellista.

Kun BrandButton() on luotu edellä kuvatun mukaisesti mahdollisimman joustavaksi, pakollisiksi parametreiksi jäävät pelkästään nappulassa näytettävä merkkijono (String) text sekä painikkeen painalluksen yhteydessä suoritettava funktio onButtonClicked. Nämä on jätettävä pakolliseksi siitä luonnollisesta syystä, että ne ovat täysin kontekstisidonnaisia. Joissain tapauksissa nappulan painalluksesta esimerkiksi navigoidaan sovelluksen toiseen näkymään tai jossain toisessa yhteydessä nappulan painallus taas käynnistää tapahtumaketjun, jolla haetaan tietoa vaikkapa internetistä tai laitteen sisäisestä tietokannasta. Molemmissa tapauksissa sekä esitettävän tekstin että painalluksen on oltava tarkasti määriteltävissä, missä ikinä sitä päädytäänkään käyttämään tulevaisuudessa.

Lopulta päädytään seuraavanlaiseen komponentin alustukseen, joka voidaan ottaa käyttöön Composea tukevassa ympäristössä:

```
BrandButton(
     onButtonClicked = {/* TODO: Tee jotain painalluksesta! */},
     text = ”Design systems button”
)
```

Esimerkkikoodi 3. Composable-funktio, jolla lisätään nappula sovellukseen.

4 Sovelluksen migraatio

4.1 Sovellus ennen migraatiota

Tässä sovelluksessa, jolle ei olla vielä toteutettu migraatiota, käytetään valmiita design systemsin mukaisia komponentteja, joilla on ennalta määritelty XML-rakenteensa. Nämä komponentit on luotu erillisessä projektissa, ja ne integroidaan osaksi tätä sovellusta omassa osamoduulissaan. Myös luodut Compose-perusteiset komponentit tulevat tämän osamoduulin mukana, mutta migraatiota edeltävässä sovelluksen versiossa Composea ei vielä hyödynnetä, vaan Compose otetaan asteittain käyttöön migraation edetessä.

Käyttämällä design systemsissä määriteltyjä komponentteja varmistetaan, että design systemsin mukaiset ohjeistukset toteutuvat ja käyttöliittymän ulkoasu seuraa varmasti määriteltyjä spesifikaatioita. Tämän lisäksi UI-kehitys myös nopeutuu, kun komponentteja ei tarvitse luoda alusta asti, vaan riittää, että komponenteille antaa esimerkiksi pelkkiä parametreja, joiden mukaisesti komponentti renderöidään ruudulle.

Osamoduulin tuonti projektiin on varsin helppoa, ja sitä varten voidaan hyödyntää Git-versionhallintatyökalua. Komento, jota käytetään, on seuraavanlainen:

```
git submodule add example.com/design_systems
```

Esimerkkikoodi 4. Git-komento osamoduulinen lisäämiselle projektiin.

Tämän komennon tuloksena syntyy .gitmodules-tiedosto, joka toimii asetustiedostona. Tämä tiedosto tallentaa yhteyden projektin URL-osoitteen ja

paikallisen alikansion välillä [1]. Kun design systems -osamoduuli on alustettu, on komponenttien käyttäminen helppoa ja osamoduulin sisältämiin komponentteihin voidaan viitata suoraan sekä Kotlin- että XML-tiedostoissa antamalla oikea tiedostopolku.

Osamoduuleiden kanssa työskentelyä varten on olemassa muitakin komentoja, kuten esimerkkikoodissa 5 esitetään:

```
git submodule init
git submodule update
```

Esimerkkikoodi 5. Git-komento osamoduulin alustukselle ja päivittämiselle.

Ensimmäinen näistä alustaa paikallisen konfiguraatiotiedoston, jos kehittäjä on käyttänyt esimerkiksi ensin git clone-komentoa noutaakseen jonkin projektin lähdekoodin paikalliselle työskentely-ympäristölle. Tämän jälkeen on kuitenkin käytettävä myös jälkimmäistä komentoa, jotta kaikki osamoduulin data noudetaan ja vaihdetaan osamoduuli-projekti osoittamaan oikeaan git-referenssiin.

Jälkimmäinen on myös hyödyllinen, jos vaihtaa git-haaroja pääprojektin välillä. Tällöin myös osamoduulit on päivitettävä komennolla, jotta niiden git-referenssi vastaa sitä referenssiä, joka on määritelty sen hetkiselle aktiiviselle projektihaaralle. [20.]

Sovellus, jonka migraatio tullaan toteuttamaan, on alun perin rakennettu hyödyntäen vanhaa UI-rakennustapaa, jonka peruskomponentteja ovat Activity-luokat, Fragment-luokat sekä näihin sidotut XML-tiedostot. Sovelluksessa jokainen fragmentti vastaa yhtä näkymää, ja näkymän erilaiset UI-komponentit kuten painikkeet, tekstikentät ja erilaiset listat sekä niiden poikkeavat ominaisuudet, kuten leveydet pituudet, fonttikoot ja muut vastaavat attribuutit ovat määriteltynä XML-tiedostossa, joka kuvaa näkymän käyttöliittymää.

Fragmenttien yksi päätehtävistä on tuottaa sovellukselle lisää modulaarisuutta ja parantaa näkymien uudelleen käytettävyyttä. Tällaisten asioiden

hallitseminen onkin yksinkertaisempaa fragmenttien kautta kuin hallitsemisen toteuttaminen erillistä Activity-luokkaa käyttäen jokaiselle näkymälle. [7.]

Fragmentit voidaan ottaa käyttöön erittäin helpolla tavalla kuten esimerkkikoodissa 6 määritellään:

```
dependencies {
    def fragment_version = "1.5.7"

    // Java language implementation
    implementation "androidx.fragment:fragment:$fragment_version"
    // Kotlin
    implementation "androidx.fragment:fragment-ktx:$fragment_version"
}
```

Esimerkkikoodi 6. Tuodaan fragment-kirjasto app-tason build.gradle-tiedostoon.

Kun fragment-kirjasto on tuotuna projektiin, on mahdollista luoda uusia fragment luokkia, ja hyödyntää fragmenttien ominaisuuksia kuten esimerkkikoodissa 7 määritellään:

```
class ExampleFragment: Fragment(R.layout.example_fragment){
     override fun onViewCreated(
               view: View,
               savedInstanceState: Bundle?) {
               super.onViewCreated(view, savedInstanceState)

               // XML Tiedostossa määriteltyihin komponentteihin
            voidaan hankkia referenssi täällä

               }
}
```

Esimerkkikoodi 7. Fragment-perusteinen luokka.

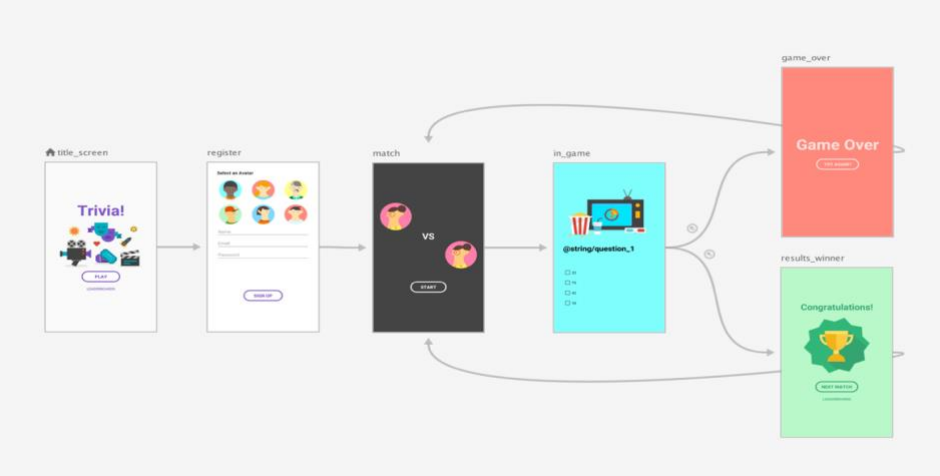
Esimerkissä luodaan fragment-luokka, jolle syötetään konstruktori-parametrina XML-tiedosto-resurssi, joka toimii layouttina näkymälle. Luotu luokka perii fragment-luokan ominaisuudet, jonka myötä on tarve sille, että korvataan (override) fragment-luokan tarvittavat metodit kuten onViewCreated, jonka jälkeen näkymän logiikka voidaan kirjoittaa fragmenttiin [21]. Tämä logiikkaa sisältää perustoimintoja kuten XML-perusteisten id-referenssien paikantamista ja tämän jälkeen esimerkiksi nappulan painamiseen reagoimista tai tekstikentän syöttöön reagoimista.

Sovellus, jolle migraatio toteutetaan, käyttää vain yhtä activity-luokkaa, jonka tehtävänä on toimia ns. isäntänä fragmenteille lisätyn FragmentContainerView-komponentin kautta. Tätä kautta tapahtuu mm. eri fragmenttien esittäminen käyttäjälle [22].

Sovellus hyödyntää yleisesti käytettyä Jetpack navigation -kirjastoa, joka helpottaa applikaation navigoinnin implementoinnissa. Käytännössä navigoinnilla tarkoitetaan vuorovaikutusta, joka mahdollistaa käyttäjien siirtymisen näkymästä toiseen applikaatiossa.

Navigaatio-komponentti koostuu kolmesta keskeisestä osasta, joita ovat: Navigation graph, NavHost ja NavController.

Navigation graph on XML-resurssi, joka sisältää kaiken navigaatioon liittyvän tiedon, tarkoittaen kaikkia yksittäisiä näkymiä applikaatiossa, joita kutsutaan kohteiksi. Tämän lisäksi navigation graph sisältää myös mahdolliset reitit, joita käyttäjä voi käyttää navigoidessaan applikaatiossa kuten kuvassa 2 havainnollistetaan. [23.]



Kuva 2. Navigation graph. [24.]

NavHost on tyhjä container, joka näyttää tietoa navigation-kaavioista. Navigaatio-komponentti sisältää default NavHost-implementaation nimeltä NavHostFragment, joka sisältää fragmentin kohteet. [23.]

```
<androidx.fragment.app.FragmentContainerView
     android:id="@+id/nav_host_fragment"
     android:name="androidx.navigation.fragment.NavHostFragment"
     android:layout_width="0dp"
     android:layout_height="0dp"
     app:layout_constraintLeft_toLeftOf="parent"
     app:layout_constraintRight_toRightOf="parent"
     app:layout_constraintTop_toTopOf="parent"
     app:layout_constraintBottom_toBottomOf="parent"
     app:defaultNavHost="true"
     app:navGraph="@navigation/nav_graph" />
```

Esimerkkikoodi 8. NavHostFragment lisättynä sovelluksen MainActivity-luokkaan.

NavController on objekti, joka hallinnoi sovelluksen navigointia NavHostista käsin. NavControllerin vastuulla on myös navigaatiokohteiden vaihtaminen NavHostissa, kun käyttäjät siirtyvät näkymästä toiseen applikaatiossa [23]. NavControllerin paikallistaminen tapahtuu kuten esimerkkikoodi 9 havainnollistaa.

```
Fragment.findNavController()
Activity.findNavController()
View.findNavController()
```

Esimerkkikoodi 9. Metodit, joilla NavController voidaan löytää riippuen kontekstista.

4.2 Sovelluksen arkkitehtuuri

Tyypillisessä Android-sovelluksessa on useita sovelluskomponentteja, kuten esimerkiksi aktiviteetteja ja fragmentteja. Nämä sovelluskomponentit määritellään sovelluksen ns. manifestissa. Android-käyttöjärjestelmä käyttää tätä tiedostoa päättääkseen, miten sovellus integroidaan osaksi laitteen kokonaiskäyttökokemusta. Ottaen huomioon, että tyypillisessä Android-sovelluksessa voi olla useita komponentteja ja käyttäjät vuorovaikuttavat usein useiden sovellusten kanssa lyhyen ajanjakson aikana, sovellusten on sopeuduttava erilaisiin käyttäjäohjattuihin työnkuluihin ja tehtäviin.

Koska mobiililaitteet ovat resurssirajoitteisia, voi käyttöjärjestelmä voi milloin tahansa sulkea joitakin sovellusprosesseja tehdäkseen tilaa uusille. Tässä ympäristössä sovelluksen komponentit voivat käynnistyä yksittäin ja

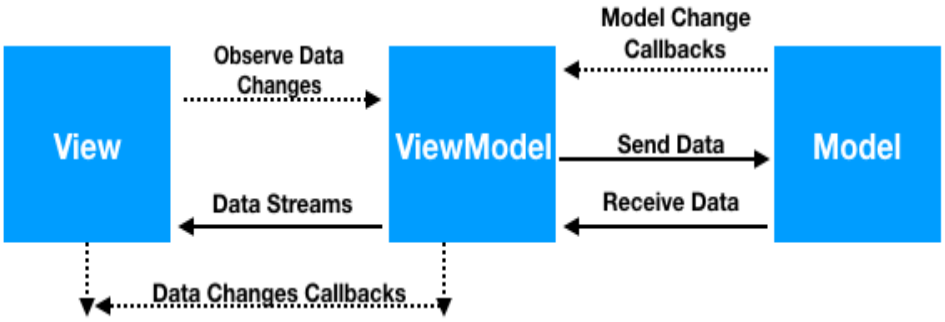
epäjärjestyksessä, ja käyttöjärjestelmä tai käyttäjä voi tuhota ne milloin tahansa. Koska nämä tapahtumat eivät ole kehittäjän hallinnassa, ei ole suotavaa tallentaa tai säilyttää sovellusdataa tai tilaa sovelluksen komponenteissa, eikä sovelluskomponenttien tulisi olla riippuvaisia toisistaan.

Sovelluksen koon kasvaessa tietynkaltaisen arkkitehtuurin seuraaminen on tärkeätä. Se varmistaa applikaation kestävyyden, turvallisen laajentamisen ja sen lisäksi tekee applikaatiosta helpommin testattavissa olevan.

Lähtökohtaisesti tämä tarkoittaa sitä, että applikaatioilla tulisi olla vähintään kaksi eri kerrosta: käyttöliittymä-kerros, joka esittää applikaation dataa näytöllä sekä tämän lisäksi datakerros, joka sisältää applikaation liiketoimintalogiikan. Näiden lisäksi voidaan käyttää myös domain-kerrosta, joka yksinkertaistaa vuorovaikutusta käyttöliittymäkerroksen ja datakerroksen välillä. [25.]

Sovelluksessa käytetty ohjelmistoarkkitehtuuri on MVVM, ja sitä ei tulla muuttamaan migraation yhteydessä. Tässä arkkitehtuurissa M (Model) tarkoittaa datakerrosta, V (View) käyttöliittymäkerrosta ja VM (ViewModel) domainkerrosta.

ViewModel-luokan toiminnallisuutta kuvailtiin jo aiemmin luvussa 2.5, ja tässä sovelluksessa ViewModel-luokan vastuualueisiin kuuluvatkin hyvin pitkälti asiat, joita jo aikaisemmin lueteltiin sen yleisiksi tehtäviksi. Kuvassa 3 havainnollistetaan MVVM-arkkitehtuurin mukaisten kerrosten välistä vuorovaikutusta.



Kuva 3. MVVM-arkkitehtuuri kaaviomuodossa [26.].

MVVM-arkkitehtuuri on yleisesti käytetty Jetpack Composen kanssa, ja tämä on syy, minkä takia sovelluksen arkkitehtuuria ei ole syytä ruveta muuttamaan, vaikka Composen kanssa on olemassa muitakin yhteensopiva arkkitehtuureita, kuten esimerkiksi MVI (Model, View, Intent).

Jos sovelluksen MVVM-arkkitehtuuria haluttaisiin kuitenkin ruveta muuttamaan, olisi tässä vaiheessa otollisinta, koska koodia muutetaan useasta paikasta ja domainkerros itsessään saa myös joitakin pienehköjä muutoksia riippuen siitä, kuinka paljon erilaisia toiminnallisuuksia tietyllä sovelluksen osalla tai erinäisellä näkymällä on.

Muita Android-kehityksessä yleisesti (vaikkakin vanhempia) käytettyjä arkkitehtuureja ovat mm. MVP (Model, View, Presenter) - ja MVC (Model, View, Controller) -arkkitehtuurit. Näissä on kuitenkin omat ongelmansa, joiden vuoksi ne eivät välttämättä ole erityisen miellyttäviä ratkaisuja käyttöönotettavaksi Composen kanssa.

Esimerkiksi MVP:n peruslogiikka, jossa Viewin ja Presenterin toiminta on sidottu vahvasti erilaisten Viewin ja Presenterin välisten rajapintojen (Interface) kutsujen perusteella toteutettavaan toimintaan, ei ole lähtökohtaisesti sellainen, joka on yhteensopiva deklaratiivisen ohjelmointiparadigman kanssa, jota Jetpack Compose edustaa.

4.3 Yksittäisen näkymän muuntaminen Composable-muotoon

Android-sovelluskehityksessä, kun siirrytään käyttämään Jetpack Composea fragmenttien sijasta, ensimmäinen askel on päästä eroon fragmentteihin liittyvistä XML-komponenteista ja mahdollisesti myös itse XML-tiedostoista.

On olemassa useampia tapoja, kuinka se voidaan toteuttaa. Ensimmäinen mahdollisuus on käyttää XML-tiedostoihin upotettuja ComposeView'jä. Tämä on yksinkertainen tapa, jolla voidaan mahdollistaa sekä XML- ja Compose komponenttien käyttäminen samanaikaisesti. ComposeView voidaan ottaa käyttöön kutsumalla sen setContent-metodia.

Esimerkkikoodissa 10 esitellään, kuinka upottaa ComposeView XML-tiedostoon. prosessissa itsessään ei ole mitään tavallisesta poikkeavaa, ja ComposeView'n asettaminen tapahtuu samalla tavalla kuin minkä tahansa muunkin komponentin asettaminen XML-tiedostoon.

```
<androidx.compose.ui.platform.ComposeView
      android:id="@+id/compose_view"
      android:layout_width="match_parent"
      android:layout_height="match_parent" />
```

Esimerkkikoodi 10. Yksittäinen ComposeView XML-tiedostossa.

Tämän jälkeen XML-tiedostoon sidotusta fragmentistä voidaan asettaa compose-sisältö ComposeView'iin. Se tapahtuu ensin hankkimalla referenssi XML-tiedostossa määriteltyyn id:hen. Sen jälkeen käytetään apply-scope-funktiota viitattuun ComposeView'iin, jonka jälkeen kutsutaan setContent-metodia, jonka kautta taas päästään Compose-maailmaan ja voidaan asettaa Compose sisältö komponentin sisälle esimerkkikoodi 11 mukaisesti. [27.]

```
_binding = FragmentExampleBinding.inflate(inflater, container, false)
val view = binding.root

binding.composeView.apply {
setViewCompositionStrategy(ViewCompositionStrategy.DisposeOnViewTreeLi
fecycleDestroyed)
            setContent {
                // In compose world
                MaterialTheme {
                    Text("Hello Compose!")
```

```
                }
            }
        }
```

Esimerkkikoodi 11. ComposeViewin asettaminen yhdelle komponentille XML-tiedostoon fragment-perusteisessa luokassa.

Tapa, jota tämän sovelluksen migraatiossa käytetään, ei kuitenkaan sisällä ComposeViewi'en käyttöä yksittäisiä komponentteja varten, koska sovellus ei sisällä sellaisia näkymiä, joissa olisi tarpeellista hyödyntää sekä XML-perusteisia ja Compose-perusteisia komponentteja samanaikaisesti. Sen sijaan ComposeView luodaan koodissa, ja siihen asetetaan esimerkkikoodin 12 mukaisesti kaikki tietyn ruudun Compose-perusteiset käyttöliittymäkomponentit.

```
class WelcomeFragment: Fragment() {
     override fun onCreateView(
          inflater: LayoutInflater,
          container: ViewGroup?,
          savedInstanceState: Bundle?
): View {
     return ComposeView(requireContext()).apply {

     setViewCompositionStrategy(ViewCompositionStrategy.DisposeOnViewTr
eeLifecycleDestroyed)
     setContent {
               // Tänne pelkästään Compose perusteinen koodi
          }
     }
}
```

Esimerkkikoodi 12. ComposeView'n asettaminen koko layoutin käsittäväksi fragment-luokassa.

Esimerkki on sovelluksen ensimmäisestä näkymästä, joka esitetään käyttäjälle ja on toimiva ratkaisu, jotta saadaan fragmentin alustus toteutettua niin, että se tukee Composea heti tavalla, jossa koodi voidaan kirjoittaa suoraan fragmenttiin ilman suurempia vaivoja. Yllä olevassa esimerkissä ei tarvitse luoda erillisiä ComposeViewe'jä, joihin täytyy etsiä id-referenssejä ja sitä kautta vasta mahdollistaa Compose-koodin kirjoittaminen. Tässä vaiheessa myös olemassa oleva fragmenttiin sidottu XML-tiedosto voidaan poistaa, koska sille ei ole enää käyttöä.

Toteutuksessa on kuitenkin yksi ongelma, joka on hyvä huomioida jo aikaisessa vaiheessa. Se liittyy hyviin ohjelmointikonventioihin, ja tässä esimerkissä

ongelmaksi nousee samankaltaisen fragment-luokan luomisen jatkuva tarve. Koska vastaavaa rakennetta tullaan hyödyntämään käytännössä kaikkialla sovelluksessa, on hyvä hyödyntää luokkien abstraktiota, jota havainnollistetaan esimerkkikoodissa 13. Näin vähennetään tarvittavan koodin määrää ja koodista saadaan ylipäänsä helpommin luettavaa, ymmärrettävämpää sekä mahdollistetaan koodin helpompi ylläpito.

Luotava Kotlin-ohjelmointikielen mukainen abstract class näyttää seuraavalta [28.]:

```
abstract class ComposeContentFragment: Fragment() {

     @Composable
     abstract fun onSetComposeContent()

     // Vastaava alustus kuin edeltävän esimerkin onCreateViewissä

     setContent {
          onSetComposeContent()
     }
}
```

Esimerkkikoodi 13. Abstract-luokan implementointi Compose-perusteisten luokkien suoraviivaisempaa käyttöönottamista varten.

Näin ollen, jos jo aiemmin mainittu WelcomeFragment muunnetaan luotavaksi esimerkkikoodi 14 mukaisesti:

```
class WelcomeFragment: ComposeContentFragment() {
     @Composable
     override fun onSetComposeContent(){
          // Tänne compose perusteinen koodi
     }
}
```

Esimerkkikoodi 14. Luotuun abstract-luokkaan perustuva fragment-luokan implementaatio.

Saadaan vähennettyä tarvittavan koodin määrää oleellisesti, ja koodi muuntuu myös erittäin luettavaksi.

Seuraava askel sen jälkeen, kun peruslogiikka sovelluksen fragmenttien luontia varten on valmis, onkin itse Compose-koodin kirjoittaminen kyseiseen fragmenttiin. Tässä vaiheessa ruvetaankin jo hyödyntämään niitä design

systems-osamoduulin mukana tulevia komponentteja, jotka perustuvat Compose-implementaatioille, ja niiden käyttöönottaminen onnistuu käyttämällä import-komentoa ja antamalla esimerkkikoodin 15 mukainen oikea tiedostopolku siihen osamoduulin osioon, jossa kyseinen komponentti sijaitsee.

```
import designsystems. . .composeComponents.BrandButton
```

Esimerkkikoodi 15. Tuodaan tietty painikekomponentti design systems -osamoduulista käytettäväksi pääprojektin ympäristöön.

Kun komponentin tiedostopolku on määriteltynä käsiteltävään tiedostoon, on sen käyttäminen erittäin helppoa, ja sitä voidaan käyttää kuin mitä vain muutakin composable-funktiota.

Aiemmin luvussa 3.2 luodun BrandButton() komponentin kohdalla pakollisina syötettävinä parametreina tulee antaa vain teksti, joka näytetään painikkeessa, sekä toiminto, joka suoritetaan, kun käyttäjä painaa kyseistä painiketta kuten esimerkkikoodissa 16 havainnollistetaan. Tässä tapauksessa nappulan painaminen kutsuu NavControlleria luvussa 4.1 mainitulla tavalla ja navigoi sovelluksen seuraavaan näkymään. Vastaavaa navigointitapaa käytetään myös sovelluksen muissa näkymissä.

```
BrandButton(
     onButtonClicked = { findNavController().navigate(…) },
     text = stringResource(id = R.string.btn_start)
)
```

Esimerkkikoodi 16. Design systemsin mukaisen painikkeen käyttö, kun se on tuotuna pääprojektiin.

Koska tämä WelcomeFragment on toiminnallisuudeltaan erittäin yksinkertainen, ja se ei käsitä muuta sisältöä kuin vain tekstiä sovelluksen esittelyä varten sekä nappulan, jolla käyttäjä voi siirtyä seuraavaan näkymään, sitä voidaan käyttää hyvänä esimerkkinä myös toista perusrakennetta varten, joka toistuu useissa näkymissä hieman eri muodossa.

Komponentti, jota tullaa hyödyntämään on Jetpack Compose -ohjelmistokehykseen sisäänrakennettu composable-funktio nimeltä Scaffold.

Tämän komponentin perusideana on toimia eräänlaisena perustavanlaatuisena standardoituna lähtökohtana käyttöliittymän rakentamista varten, jossa noudatetaan Material Designin (Googlen luoma eräänlainen design-kieli) mukaisia ohjeistuksia [29]. Scaffoldin perusominaisuuksiin kuuluu esimerkiksi Floating action buttonin, TopBarin tai BottomBarin oikeanlainen asettelu ruudulle, niin että ohjelmoijan ei tarvitse muuta kuin antaa edellä mainittujen UI-osien composable-funktiot parametreina, ja Scaffold asettelee ne oikeille paikoilleen ruudulla, kuten esimerkkikoodissa 17 näytetään.

Lisäksi Scaffoldille tulee antaa parametrinä myös Composable-sisältö, joka esitetään myös ruudulla muiden komponenttien lisäksi. [30.]

```
@Composable
fun ScaffoldExample() {
    Scaffold(
        topBar = {},
        bottomBar = {},
        floatingActionButton = {},
        content = {
          Column() {
               Text(…)
               BrandButton(…)
          }
     }
)}
```

Esimerkkikoodi 17. Scaffold, parametreista vain content on pakollinen syöttää.

Yleensä content-osion sisältö niin sanotusti kääritään (wrap) vielä oman composable-funktionsa sisään kuten Composen sisäänrakennettuun Column()-funktioon, jota tässäkin esimerkissä tullaan pian hyödyntämään. Column on ensimmäinen yleisistä Composen komponenteista, joita käytetään UI:n luomista varten. Toinen näistä on Row(). Nimiensä mukaisesti ne asettelevat järjestyksessä komponentit sisällensä joko pysty- tai vaakasuuntaan.

Tyypillisesti Columnille annetaan vielä joitakin parametrejä, jotta se vastaa tarpeisiin halutulla tavalla. Yleinen parametri, joka tulee antaa, on modifier-parametri. Modifierillä voidaan muokata composable-funktioita monin tavoin kuten todettiin jo aiemmin luvussa 3.2.

Yleiset Columnille annettavat muokkaukset modifierin kautta ovat esimerkiksi komponentin koon pakottaminen kattamaan kaikki mahdollinen tila ruudulla ja aktivoimaan Columnin pystysuuntainen vierittäminen, kuten esimerkkikoodi 18 näyttää. Vastaavanlaiset ominaisuudet esiintyvät hyvin monissa sovelluksen eri näkymissä toistuvasti.

```
Column(
     Modifier = Modifier
          .fillMaxSize()
          .verticallScroll(rememberScrollState())
     ){
          Text(...)
          BrandButton(...)
     }
)
```

Esimerkkikoodi 18. Column composable-funktio, jolle on syötettynä modifier-parametri.

4.4 Sovelluksen navigaation muuntaminen Composable-muotoon

Kun sovelluksen migraatiossa on päästy siihen pisteeseen, että kaikki sovelluksen näkymät on muunnettu tukemaan Composea ja kaikki tarvittavat ruutu kohtaiset fragmentteihin sidotut XML-tiedostot on poistettu sovelluksesta, edetään vaiheeseen, jossa sovelluksen navigointilogiikkaa ruvetaan muuntamaan Composea tukevaksi.

Prosessi itsessään on melko laaja, ja sen seurauksena jälleen muutetaan logiikka jo edellä kuvaillusta XML-perusteisesta implementaatiosta kohti pelkästään Compose-ohjelmistokehyksen mukaista toteutustapaa. Myös Compose-implementaatio edellyttää useita erinäisiä komponentteja, jotta sovellukselle saadaan luotua luotettava ja vankka navigointilogiikka, joka mahdollistaa saumattoman siirtymisen näkymästä toiseen.

Jotta compose-navigointia voidaan alkaa hyödyntämään, on ensiksi kyseinen Compose Navigation- kirjasto tuotava osaksi sovelluksen app-tason

build.gradle-tiedoston dependencies-osiota esimerkkikoodi 19 mukaisella tavalla:

```
implementation "androidx.navigation:navigation-compose:$nav_version"
```

Esimerkkikoodi 19. Tuodaan Compose Navigation -kirjasto osaksi sovellusta.

Tämän jälkeen tarvitaan seuraavat edellä määritellyn kirjaston tuomat ominaisuudet, jotta sovelluksen navigointilogiikkaa voidaan alkaa muokkaamaan. Rakenteeltaan, nimeämiseltään ja logiikaltaan ominaisuudet ovat hyvin samankaltaisia XML-perusteisen navigoinnin kanssa.

1. NavController on avainasemassa puhuttaessa navigoimisesta compose-kontekstissa. Se vastaa navigaation hallinnasta sovelluksessa, pitäen sisällään mm. NavGraphin, sekä tarjoaa metodeita, jotta mahdollistetaan siirtymimen navigaatiokaaviossa näkymästä toiseen. Sen tehtäviin kuuluu esimerkiksi pitää muistissa ne sovelluksen näkymät, joihin on navigoitu aiemmin. [31.]

2. NavHost. Compose-kontekstissa on tarve ottaa käyttöön Navhost composable-funktio, johon luodaan navigaatiokaavio sisältäen ne erinäiset navigointikohteet, johon sovelluksessa voidaan siirtyä. [24.]

3. Navigointikohteet (destinations), voidaan määritellä omissa funktioissaan, jotka kirjoitetaan seuraavankaltaisella tavalla: *composable(“route”)*. Nämä ovat niitä kohteita, joita Navhost hyödyntää siihen kirjoitetussa kaaviossa. [32.]

NavControllerin perusedellytys on se, että se luodaan hyvin aikaisessa vaiheessa ja sijoitetaan compose-hierarkiassa mahdollisimman korkealle, jotta hierarkiassa seuraavilla komponenteilla on myös pääsy NavControlleriin ja sen tuomiin navigointiominaisuuksiin. Tämän sovelluksen tapauksessa NavController tuodaan koodiin jo MainActivityssä, josta koodin suoritus alkaa, ja näin ollen MainActivty on se ainut Activity-luokka, jonka päälle koko Compose-

implementaatio rakennetaan. NavController alustetaan esimerkkikoodin 20 mukaisesti:

```
val navController = rememberNavController()
```

Esimerkkikoodi 20. Sovelluksen compose-perusteinen NavController.

Seuraavaksi tarvitaan ne kohteet, jotka tullaan lisäämään NavHostin navigointikaaviota varten. Esimerkkikoodi 21 on esimerkki siitä, kuinka tällainen navigointikohde voidaan luoda:

```
composable(
     route = "welcome",
) {
          WelcomeScreen()
}
```

Esimerkkikoodi 21. NavHostin navigointikaaviolle syötettävä kohde.

Tässä yhteydessä jo olemassa olevien näkymien implementaatiota tullaan muokkaamaan tuntuvasti. Toistaiseksi näkymien logiikka on perustunut edelleen fragment-luokkien tuomalle toiminnallisuudelle, mutta nyt kun siirrytään muuntamaan navigaatiota Compose-perusteiseksi, fragmentit eivät ole enää validi vaihtoehto näkymien esittämistä varten.

Esimerkkikoodista 21 erottuu kohta WelcomeScreen(), joka on käytännössä se kohta implementaatiota, joka vastaa jo aiemmin luotua ja esiteltyä WelcomeFragmenttiä. Tässä tapauksessa tämä WelcomeScreen() kuitenkin esittää omaa composable-fuktiotaan, joka syötetään tämän juuri luodun navigointikohteen content trailing lambdan sisältönä [33].

Todellisuudessa tämän content-lambdan mukana voi syöttää mitä vain compose-sisältöä, mutta koodin luettavuuden vuoksi on suositeltavaa, että yksittäisen ruudun näkymä kirjoitetaan omaan composable-funktioonsa [32]. Tämä sovellus seuraa samanlaista konventiota joka paikassa, ja fragment-jälkiliite muuntuu käytännössä Screen-nimeämiskonvention mukaiseksi.

Sisällöltään luotu WelcomeScreen() vastaa hyvin pitkälti sitä, mitä WelcomeFragment sisälsi. Nyt kuitenkin mahdollistetaan se, ettei näkymä ole enää mitenkään sidoksissa Fragment-implementaatioihin. Näkymä luodaan kuten mikä vain composable-funktio käyttämällä @Composable-annotaatiota ja sen jälkeen kirjoittamalla funktion nimi ja sen tarvitsemat parametrit, kuten navigointilogiikka, joka tullaan määrittelemään tarkemmin NavHostin-navigointikaaviossa.

Tämän jälkeen, kun NavController ja navigointikohteet on luotuna, tarvitaan NavHost, johon kirjoitetaan navigointikaavio sisältäen kaikki sovelluksen mahdolliset navigointikohteet. Ennen tätä on kuitenkin vielä mahdollista hieman parantaa edellä näytettyä esimerkkikoodia 21 Kotlin-kielen ominaisuuksien avulla. Ensimmäinen näistä on extension-funktiot. Niiden avulla luokan ominaisuuksia ja metodeja voidaan laajentaa ilman, että on tarvetta periä itse luokkaa [34].

Luettavuus saadaan paremmaksi, kun kohde kirjoitetaankin suoraan osaksi NavGraphBuilder-luokkaa esimerkkikoodin 22 mukaisella tavalla:

```
NavGraphBuilder.welcomeComposable(
     route: String,
     navigateToTosScreen: () -> Unit
){
     composable(
          route = route
     ) {
          WelcomeScreen( /**/ )
     }
}
```

Esimerkkikoodi 22. Paranneltu Welcome-navigointikohteen implementaatio.

Koska navigointikohteet vaativat aina route-parametrin, on mahdollista hyödyntää myös Kotlinin sealed class -ominaisuutta, jolla mahdollistetaan rajattu perintä jonkin luokan hierarkialle [35].

Näin ollen voidaan luoda esimerkkikoodin 23 mukainen ”Screen” sealed-luokka johon navigointikohteet määritellään kootusti ja ne alustetaan sopivan route-arvon kanssa:

```
sealed class Screen(val route: String) {
     object Welcome: Screen(route = "welcome")
     object TermsOfService: Screen(route = "tos")

     // Muut Screen luokan hierarkiaan kuuluvat objektit, eli kaikki
     sovelluksen näkymät
}
```

Esimerkkikoodi 23. Navigointikohteet koottuna sealed-luokkaan.

Kun esimerkkikoodien 22 ja 23 mukaiset parannukset ovat tehtynä, NavHost voidaan alustaa esimerkkikoodin 24 mukaisella tavalla:

```
NavHost(
     navController = navController,
     startDestination = startDestination
     ) {
          welcomeComposable(
               route = Screen.Welcome.route,
               navigateToTosScreen = {
                    navcontroller.navigate(Screen.TermsOfService.route)
               }
          )

          termsOfServiceComposable(
               // Terms of service-navigointikohteen parametrit
          )

          // Sovelluksen muut navigointikohteet
     }
)
```

Esimerkkikoodi 24. NavHost-implementaatio sisältäen kaavion ja sen navigointikohteet.

NavHost composable-funktion vastuulle kuuluu kuitenkin muutakin kuin pelkästään navigointikaavion alustaminen. Sille on syötettävä parametreina muutakin tietoa. NavHost vaatii jo aiemmin alustetun NavControllerin, jonka kautta tapahtuu navigointi navigointikaaviossa määriteltyjen navigaatiokohteiden välillä ja muiden navigaatiotoiminnallisuuksien hyödyntäminen kuten navigaatiopinon viimeisimpien kohteiden poistaminen (pop).

NavHostille on myös annettava startDestination parametrina. Se on navigointikohde, joka nimensä mukaisesti toimii sovelluksen aloitusnäkymänä. Tämä arvo voi muuttua esimerkiksi sen perusteella, kuinka käyttäjä on käyttänyt sovellusta. Jos käyttäjä on esimerkiksi kirjautunut sisään sovellukseen

tunnuksillaan, startDestination saa eri arvon jonkin aiemmin suoritetun logiikan perusteella, eikä käyttäjää pakoteta aina aloittamaan esimerkiksi jo nähdystä welcomeComposablesta.

Tässä vaiheessa sovelluksen migraatio onkin jo lähellä valmista ja jäljellä ei ole muuta kuin mahdollisen testiautomaation ja saavutettavuusongelmien korjaaminen. Ongelmat juontavat juurensa siitä, että vaikka migraation jälkeinen lopputulema onkin ulkonäkymältään samanlainen kuin XML-toteutus, on sen implementaatio koodin puolesta muuttunut huomattavasti.

Esimerkiksi testiautomaatio saattaa hajota, koska sen toiminnallisuus usein perustuu siihen, että luoduilla komponenteilla on jonkinlainen id, joilla ne tunnistetaan näkymästä. Tällainen on XML-elementeillä lähes aina, mutta composable-funktioilla ei ole vastaavanlaista attribuuttia.

Saavutettavuus taas saattaa muuttua, koska jotkin toiminnallisuudet saattavat toimia huonommin tai paremmin Composen kanssa.

Kun sovelluksen perustoiminnallisuus on olemassa, voidaan myös hioa joitakin sovelluksen ominaisuuksia osia paremmaksi, esimerkiksi lisäämällä animaatioita sovellukseen. Yksinkertainen paikka, jonne animaatioiden lisääminen on vaivatonta, on sovelluksen navigointi. Käytännössä se tarkoittaa sitä, että aina kun siirrytään ruudusta toiseen, siirtyminen esitetään jonkinlaisen animaation kanssa.

Toteutus vaatii sitä, että tietylle navigointikohteelle annetaan enterTransition ja exitTransition-parametrit. Mahdollisia animaatioita ovat esimerkiksi ruudun liukuminen johonkin suuntaan, niin pysty- tai vaakasuunnassa tai näitä sekoittaen [36]. Tyypillinen toteutus, joka valikoitui kuitenkin usein käytettäväksi tässä sovelluksessa, on varsin hillitty. Tarkoittaen sitä, että ruutu liukuu oikealta näkymään ja poistuu vastaavasti vasemmalle, kun uusi näkymä tulee taas näkymään oikealta. Animaatiolle voidaan antaa myös jokin aika, kuinka kauan toteutus kestää. Tätä havainnollistetaan esimerkkikoodilla 25.

```
composable(
          route = “menu”,
          enterTransition = {
               slideIntoContainer(
                    towards = Left,
                    animationSpec = tween(
                         durationMillis = 500
               )
     },
          exitTransition = {
               slideOutOfContainer(
                    towards = Left,
                    animationSpec = tween(
                         durationMillis = 500
               )

     }
)
```

Esimerkkikoodi 25. Navigointikohteessa määritellyt enter ja exit transitiot.

5 Lopuksi

Sovelluksen migraation toteuttamisen myötä havaittiin monia positiivisia lopputulemia. Ensinnäkin tarvittavan koodin määrä vähentyi merkittävästi, kun valtava määrä XML-tiedostoja saatiin poistettua kokonaan ja muutenkin koodin logiikkaa saatiin yleisesti yksinkertaistettua. Myös edellytykset kehitykselle tulevaisuudessa parantuivat merkittävästi, koska kaikki uudet ominaisuudet voidaan kirjoittaa suoraan hyödyntäen Composea.

Esimerkiksi compose-perusteisen tunnistautumisruudun esittäminen käyttäjälle on erittäin suoraviivaista ja erityyppisiä käyttäjän tunnistautumisvaatimuksia on helppo implementoida Composen avulla.

Migraation seurauksena myös koodin luettavuus ja edellytykset ylläpidolle parantuivat huomattavasti. Compose on Googlen modernien suositusten mukainen ohjelmistokehys, jonka kehittämiseen panostetaan paljon Googlen toimesta. Näin ollen on hyvin epätodennäköistä, että lähitulevaisuudessa tulisi tapahtumaan suuria deprekaatioita, jotka vaikuttaisivat migrattuun sovellukseen.

Migraatio tarjoaa vakaan perustan sovelluksen jatkokehitykselle ja varmistaa sen ajantasaisuuden ja yhteensopivuuden tulevien Android-päivitysten kanssa, mikä ei olisi lainkaan taattua, jos olisi pitäydytty vanhassa XML-perusteisessa implementaatiossa.

Lähteet

1 View. Verkkoaineisto. Google Developers. <https://developer.android.com/reference/android/view/View>. Luettu 26.11.2022.

2 React. Verkkoaineisto. React. <https://react.dev/>. Luettu 16.3.2024.

3 React Native. Verkkoainesto. React Native. <https://reactnative.dev/>. Luettu 16.3.2024.

4 Flutter – Build for any screen. Verkkoaineisto. Flutter. <https://flutter.dev/>. Luettu 16.3.2024.

5 SwifttUI. Verkkoaineisto. Apple Developer. <https://developer.apple.com/xcode/swiftui/>. Luettu 16.3.2024.

6 Build better apps faster with Jetpack Compose. Verkkoaineisto. Google Developers. <https://developer.android.com/jetpack/compose>. Luettu 16.3.2024.

7 Künneth, Thomas. 2022. Android UI Development with Jetpack Compose. Birmingham: Packt Publishing.

8 Gaba, Vinya. 2021. A Practical Introduction to Jetpack Compose Android Apps. Verkkoaineisto. CODE Magazine. <https://www.codemag.com/Article/2105061/A- Practical-Introduction-to-Jetpack-Compose-Android-Apps>. Päivitetty 31.8.2022. Luettu 8.9.2022.

9 Why adopt Compose. Verkkoaineisto. Google Developers. <https://developer.android.com/jetpack/compose/why-adopt>. Luettu 25.9.2022.

10 Thinking in Compose. Verkkoaineisto. Google Developers. <https://developer.android.com/jetpack/compose/mental-model>. Luettu 26.11.2022.

11 State and Jetpack Compose. Verkkoaineisto. Google Developers. <https://developer.android.com/jetpack/compose/state>. Luettu 28.11.2022.

12 Arriola, Chris. 2022. Composable Functions. Verkkoaineisto. <https://medium.com/androiddevelopers/composable-functions-a505ab20b523>. Luettu 28.11.2022.

13 Where to hoist state. Verkkoaineisto. Google Developers. <https://developer.android.com/jetpack/compose/state-hoisting>. Luettu 28.11.2022.

14 Ghita, Catalin. 2022. Kickstart Modern Android Development with Jetpack and Kotlin. Birmingham: Packt Publishing.

15 Perez-Cruz, Yesenia. 2019. Expressive Design Systems. New York: A Book Apart.

16 Vesselov, Sarrah & Taurie Davis. 2019. Building Design Systems: Unify User Experiences through a Shared Design Language. Dade City, Portland: Apress.

17 Compose modifiers. Verkkoaineisto. Google Developers. <https://developer.android.com/jetpack/compose/modifiers>. Luettu 9.3.2024.

18 Accessibility in Compose. Verkkoaineisto. Google Developers. <https://developer.android.com/jetpack/compose/accessibility>. Luettu 9.3.2024.

19 Functions. Verkkoaineisto. <https://kotlinlang.org/docs/functions.html#default-arguments>. Luettu 10.3.2024.

20 7.11 Git Tools - Submodules. Verkkoaineisto. <https://git-scm.com/book/en/v2/Git-Tools-Submodules>. Luettu 5.8.2023.

21 Fragments. Verkkoaineisto. Google Developers. <https://developer.android.com/guide/fragments>. Luettu 5.8.2023.

22 Create a fragment. Verkkoaineisto. Google Developers. <https://developer.android.com/guide/fragments/create>. Luettu 5.8.2023.

23 Navigation. Verkkoaineisto. Google Developers. <https://developer.android.com/guide/navigation>. Luettu 5.8.2023.

24 Design navigation graphs. Verkkoaineisto. Google Developers. <https://developer.android.com/guide/navigation/design/design-graph>. Luettu 5.8.2023.

25 Guide to app architecture. Verkkoaineisto. Google Developers. <https://developer.android.com/topic/architecture>. Luettu 6.9.2023.

26 Android MVVM Design Pattern. Verkkoaineisto. <https://journaldev.nyc3.digitaloceanspaces.com/2018/04/android-mvvm-pattern.png>. Luettu 6.9.2023.

27 Using Compose in Views. Verkkoaineisto. Google Developers. <https://developer.android.com/jetpack/compose/migrate/interoperability-apis/compose-in-views>. Luettu 6.9.2023.

28 Classes. Verkkoaineisto. <https://kotlinlang.org/docs/classes.html#abstract-classes>. Luettu 10.3.2024.

29 Material Design. Material Design. Verkkoaineisto. <https://m3.material.io/>. Luettu 10.3.2024.

30 Scaffold. Verkkoaineisto. Google Developers. <https://developer.android.com/jetpack/compose/components/scaffold>. Luettu 3.3.2024.

31 Create a navigation controller. Verkkoaineisto. Google Developers. <https://developer.android.com/guide/navigation/navcontroller>. Luettu 10.3.2024.

32 Navigate to a destination. Verkkoaineisto. Google Developers. <https://developer.android.com/guide/navigation/use-graph/navigate#navigate-example>. Luettu 10.3.2024.

33 Higher-order functions and lambdas. Verkkoaineisto. <https://kotlinlang.org/docs/lambdas.html#passing-trailing-lambdas>. Luettu 10.3.2024.

34 Extensions. Verkkoaineisto. <https://kotlinlang.org/docs/extensions.html>. Luettu 10.3.2024.

35 Sealed classes and interfaces. Verkkoaineisto. <https://kotlinlang.org/docs/sealed-classes.html>. Luettu 10.3.2024.

36 Quick guide to animations in Compose. Verkkoaineisto. Google Developers. <https://developer.android.com/jetpack/compose/animation/quick-guide#animate-whilst>. Luettu 16.3.2024.