Teemu Laakso

Tuotekatalogi Nordean Android-sovellukseen

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tietotekniikan koulutusohjelma

Insinöörityö

24.4.2018

Metropolia

Tekijä Otsikko Sivumäärä Aika	Teemu Laakso Tuotekatalogi Nordean Android-sovellukseen 36 sivua 24.4.2018
Tutkinto	insinööri (AMK)
Tutkinto-ohjelma	Tietotekniikan koulutusohjelma
Ammatillinen pääaine	Ohjelmistotekniikka
Ohjaajat	Mobiilitiimin Johtaja Teemu Lepistö Lehtori Juha Kämäri

Insinöörityö tehtiin Nordealle osana Nordean uuden mobiilipankin kehitystä. Työn tavoitteena oli kehittää Nordean tuotteita ja palveluita esittelevä tuotekatalogi Nordean Android-mobiilisovellukseen. Tuotekatalogin tarkoitus on antaa asiakkaalle ajantasaista tietoa Nordean tuotteista ja palveluista sekä tarjota uusi myyntikanava.

Teoriaosuudessa tutustuttiin Android-sovelluskehitykseen ja komponentteihin. Myös toteutuksessa käytetyt kolmannen osapuolen kirjastot ja arkkitehtuurimalli esiteltiin.

Toteutuksessa tuli noudattaa sovelluksessa käytettyä MVP-arkkitehtuuria, joka erottelee käyttöliittymälogiikan sovelluslogiikasta. Tuotekatalogin tuli myös koostua neljästä näkymästä, joista kaikista oli tehty alustava sommitelma. Isossa organisaatiossa toimiminen asetti työn toteutukselle rajoitteita, mutta sain niiden puitteissa työskennellä vapaasti.

Tuotekatalogia varten oli luotu palvelinrajapinta, joka tarjoaa tiedot tuotteista ja kategorioista ja johon tuotekatalogi on yhteydessä. Saatu data mallinnettiin ja siitä poimittiin tarvittavat tiedot eri näkymien luomiseen. Ulkoisia ohjelmointikirjastoja hyödynnettiin tarpeen mukaan. Tuotekatalogista tuli annettujen määrityksien mukainen ja saumaton osa mobiilisovellusta. Toteutuksesta tuli MVP-arkkitehtuurin ansiosta helposti ylläpidettävä kokonaisuus.

Insinöörityölle asetetut tavoitteet saavutettiin, ja tuotekatalogi pääsi osaksi Android-mobiilisovellusta. Sovellus on jo julkaistu Suomessa, ja tuotekatalogi on asiakkaiden käytettävissä.

Avainsanat	Android, Java, tuotekatalogi, MVP

Author Title Number of Pages Date	Teemu Laakso Making product catalog as part of Nordea Android application 36 pages 24 April 2018
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Software Programming
Instructors	Teemu Lepistö, Mobile Team Leader Juha Kämäri, Senior Lecturer

The bachelor's thesis was created for Nordea as part of its new mobile bank development. The objective of the thesis was to develop a product catalog for Nordea's Android mobile application. The purpose of the product catalog is to provide up-to-date information on Nordea's products and services. It will also provide a new sales channel.

Android application development and its components were introduced in theoretical part of the thesis. Third-party libraries and architecture model used in final product was presented.

The implementation had to follow the MVP architecture used in the application, which separates user interface logic from application logic. The product catalog also had to consist of four view that had designs already made. Working in a big organization sets limits to the implementation, but I was able to work freely within them.

Application programming interface that provides information about products and categories, was created for product catalog. Data obtained from API was modeled and used to create different views for product catalog. The product catalog became seamless part of the Android application and it was consistent with the designs. MVP architecture was used to create simple and easily maintained product.

All the objectives set to this bachelor's thesis were achieved and the product store became part of Android application. The application has been released in Finland and the product catalog is already available to customers.

Keywords	Android, Java, product catalog, MVP

Sisällys

Lyhenteet

AOSP	Android Open Source Project. Googlen Android-projekti.
API	Application Programming Interface. Ohjelmointirajapinta.
CSS	Cascading Style Sheets. WWW-dokumenttien tyyliohje.
CTA	Call to action. Käyttöliittymäkomponentti, joka johtaa tuotteen myyntiin.
HAL	Hardware Abstraction Layer. Rajapinta käyttöjärjestelmän ja laitteen välillä.
JSON	JavaScript Object Notation. Avoimen standartin tiedostomuoto tiedonvälitykseen.
MVP	Model – View – Presenter. Ohjelmointiarkkitehtuuri.
OHA	Open Handset Alliance. Avoimia standardeja mobiililaitteille kehittävä yritysyhteisö.
REST	Representational State Transfer. HTTP-protokollaan perustuva arkkitehtuurimalli ohjelmointirajapintojen toteuttamiseen.
SDK	Software development kit. Kokoelma ohjelmistokehitystyökaluja, jotka mahdollistavat sovellusten luonnin tietyllä alustalla.
XML	Extensible Markup Language. Tekstimuotoinen metakieli, joka kuvaa tiedon rakennetta. XML-dokumentti koostuu tekstisisällöstä sekä sille rakenteen ja merkityksen muodostavista elementeistä.

1 Johdanto

Digitaalinen murros on viimevuosina ollut paljon esillä julkisuudessa. Se on ajanut etenkin pankit ahtaalle, kun asiakkaat ovat siirtyneet entistä enemmän käyttämään digitaalisia kanavia perinteisten lähikonttorien sijaan. Vielä 90-luvulla suomalaiset pankit olivat digitalisaation edelläkävijöitä, mutta ovat viimeisen parin vuosikymmen aikana jääneet kilpailijoistaan. Myös uusia pieniä ja ketteriä toimijoita on ilmestynyt alalle.

Tällä vuosikymmenellä suomalaiset pankit ovat tehneet suuria satsauksia palveluidensa digitalisoimiseksi. Kotimaisista pankeista onkin kasvanut varteenotettavia IT-taloja. Useiden muiden digiprojektien lisäksi Nordea on nyt kehittämässä uutta versiota mobiilipankistaan. Uusi sovellus tullaan julkaisemaan kaikissa pohjoismaissa ja sillä tulee olemaan miljoonia käyttäjiä.

Insinöörityön tavoitteena on luoda tuotekatalogi Nordea Mobile -Android-sovellukseen. Tuotekatalogi on eräänlainen digitaalinen kuvasto Nordean tuotteista. Sen tarkoitus on antaa ajantasaista tietoa Nordean tuotteista ja tarjota uusi myyntikanava mobiilisovellukseen.

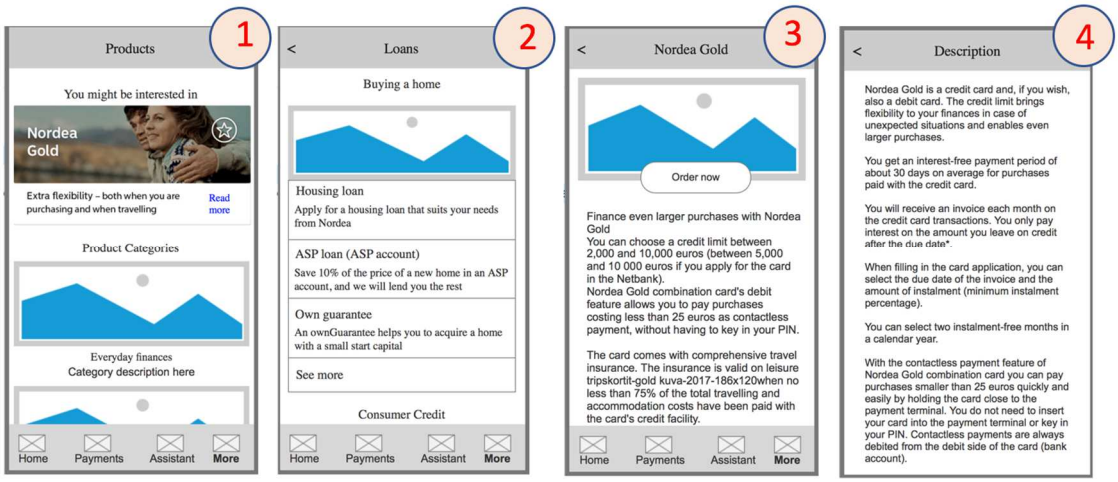
Insinöörityössä määritellään ensin, mitkä ovat tuotekatalogin ominaisuudet ja miltä sen tulee näyttää, jotta se sopii muuhun sovellukseen. Kerron myös hieman taustajärjestelmistä ja sovelluksen arkkitehtuurista. Tämän jälkeen pureudun Android-käyttöjärjestelmän kiehtovaan maailmaan ja kerron myös monista kolmannen osapuolen kirjastoista, joita sovelluksessa on käytetty.

Lopussa lukijalla on selkeä käsitys Android-sovelluksen kehityksestä, käytetyistä komponenteista ja tuotekatalogin toteutuksesta. Insinöörityö tarjoaa lukijalle myös harvinaisen mahdollisuuden nähdä, kuinka sovelluskehitystyötä tehdään yhdessä Pohjoismaiden suurimmista yrityksistä.

2 Määrittelyt ja tavoitteet

2.1 Käyttöliittymä

Tuotekatalogi on osa Nordea Mobile -Android-sovellusta. Sen tulee sopia sovelluksen yleisilmeeseen ja käyttää mahdollisuuksien mukaan sovelluksessa käytettyjä yhteisiä käyttöliittymäkomponentteja. Tuotekatalogissa liikkumisen tulee olla sujuvaa, ja käyttöliittymän tulee olla selkeä.



Kuva 1. Tuotekatalogin käyttöliittymäsuunnitelma.

Käyttöliittymä jakautuu kuvan 1 mukaisesti neljään näkymään:

1) kategorialuetteloon
2) alakategorioihin jaettuun tuoteluetteloon
3) tuotesivuun
4) lisätietosivuun.

Ensimmäisessä näkymässä on listaus tuotekategorioista. Tuotekategoria voi olla esimerkiksi lainat. Tuotekategoriat esitetään kortteina. Kortti sisältää kuvan, kategorian nimen sekä lyhyen kuvauksen. Kategoriakorttia klikkaamalla käyttäjä siirtyy seuraavaan näkymään.

Toisessa näkymässä on valitun tuotekategorian tuotteet lajiteltuna alakategorioittain. Lainat-kategorian eräs alakategoria on kulutusluotto. Myös alakategoriat esitetään kortteina, jotka pitävät sisällään tähän kategoriaan kuuluvat tuotteet. Kaikista tuotteista kerrotaan tuotteen nimi ja lyhyt kuvaus, mutta jokaisen alakategorian ensimmäistä tuotetta

korostetaan myös kuvalla. Listalla näkyy alakategorian kolme ensimmäistä tuotetta. Mikäli tuotteita on enemmän kuin kolme, saa lisää tuotteita ladattua näkymään “See more” -nappia painamalla.

Kolmas näkymä on tuotesivu. Esimerkki kulutusluotosta on Joustoluotto. Näkymä sisältää tuotekuvan ja laajemman kuvauksen tuotteesta. Puoliksi kuvan alareunan päällä on myyntiin tähtäävä painike, jota kutsutaan markkinointitermillä “Call to action” (CTA). CTA:n tarkoitus on johtaa tuotteen myyntiin. Se on koko tuotekatalogin tärkein elementti ja painikkeen tulee olla jatkuvasti näkyvillä. Näkymän lopussa on lista tuotteen lisätiedoista.

Lisätietosivu on tuotekatalogin neljäs näkymä. Tuotteella voi olla useita lisätietosivuja, mm. hinnat ja ehdot. Tässä näkymässä näytetään verkkosivujen data sellaisenaan. Se saattaa sisältää tekstiä, kuvia, taulukkoja ja linkkejä.

2.2 Palvelinrajapinta

Tuotekatalogia varten on tarjolla REST-palvelinrajapinta, joka tarjoaa asiakasohjelmalle tiedot tuotekategorioista ja tuotteista. Tuotekatalogin palvelin hakee tiedot Nordean ulkoisten verkkosivujen omalta palvelimelta. Tällä varmistetaan se, että asiakasohjelman käyttäjä saa varmasti ajantasaisen tiedon tuotteista ja etenkin niiden hinnoista. Tämän vuoksi tuotekatalogi ei vaadi juurikaan ylimääräistä ylläpitoa. Saatu data käydään läpi ja se puhdistetaan ennen tietojen välittämistä. Näin vältetään ylimääräisten haavoittuvuuksien luominen asiakasohjelmaan. Palvelin palauttaa tiedot JSON-muodossa, jonka asiakasohjelma mallintaa.

2.3 Sovelluksen arkkitehtuuri

Koska tuotekatalogi on osa laajempaa kokonaisuutta, tulee sen noudattaa yhteisesti sovittuja periaatteita arkkitehtuurin, käytettyjen komponenttien, koodin muodon ja laadun sekä testauksen suhteen.

- Sovelluksen Android-toteutuksessa käytetään Java-ohjelmointikielen versiota 7. Joitakin Java 8:n ominaisuuksia, kuten lambda-lausekkeita, voidaan käyttää Retrolambda-kirjaston avulla.
- Kirjoitetun koodin tulee noudattaa SOLID-periaatetta.

- Sovelluksessa käytetään Model-View-Presenter-arkkitehtuuria. Tästä on kerrottu tarkemmin luvussa 3.5.
- Pienin tuettu Androidin versio on 5.0 (Lollipop, Api versio 21).

3 Teknologiat

Tässä luvussa käsitellään tarkemmin Android-käyttöliittymää ja työssä käytettyjä komponentteja.

3.1 Android-käyttöjärjestelmä

Android on Googlen kehittämä avoimen lähdekoodin mobiilikäyttöjärjestelmä. Se perustuu Linux-käyttöjärjestelmään ja muihin avoimen lähdekoodin ohjelmistoihin. Se on suunniteltu pääasiassa kosketusnäyttöä hyödyntäville alustoille, kuten puhelimille. Google on kehittänyt Androidista myös omat käyttöjärjestelmät televisioille (Android TV), älykelloille (Wear OS) ja autoille (Android Auto). Androidiin pohjautuvia variaatioita on myös pelikonsoleille, digikameroille sekä tietokoneille. [1.]

3.1.1 Historia

Andy Rubin halusi kehittää käyttäjän paikkatietoja ja internetyhteyttä hyödyntävän käyttöjärjestelmän digikameroille. Tätä varten hän perusti yhteistyökumppaneidensa kanssa Android Inc:n syksyllä 2003. Pian aloittamisensa jälkeen yhtiö kuitenkin huomasi kameroiden tarjoavan liian suppean markkinan yhtiön tavoitteisiin nähden, eivätkä sijoittajat olleet yrityksestä kiinnostuneita. Yhtiö keskittyikin kehittämään Androidista kilpailevaa mobiilikäyttöjärjestelmää silloin markkinoita hallinneita Symbiania ja Microsoftia vastaan. Tässä vaiheessa Google kiinnostui Androidista ja osti yrityksen kesällä 2005. [2.]

Androidia kehitettiin aluksi perinteisille puhelimille, joissa on kiinteät näppäimet. Kaikki muuttui, kun Apple julkaisi ensimmäisen iPhonen vuoden 2007 alussa. Google perui suunnitellun ensimmäisen puhelimen julkaisun ja ryhtyi kehittämään alustastaan paremmin yhteensopivaa kosketusnäytöllisille puhelimille. [3.]

Käyttöjärjestelmän ensimmäinen julkinen beta-versio näki päivänvalonsa marraskuussa 2007 ja ensimmäinen kaupallinen versio, Android 1.0, julkaistiin syyskuussa 2008. Viimeisin merkittävä päivitys Android 8 (Oreo) julkaistiin elokuussa 2017. [1.]

3.1.2 Kehitys

Pärjätäkseen paremmin kilpailussa uutena toimijana alalla Androidista päätettiin tehdä avoimen lähdekoodiin perustuva, jolloin sitä pystytään tarjoamaan laitevalmistajille ilmaiseksi. Tätä varten perustettiin syksyllä 2007 Open Handset Alliance (OHA), joka kehittää kännykkäalan standardeja ja koordinoi Androidin kehitysprojektia: Android Open Source Projectia (AOSP). Tämä mahdollisti Androidin nopean nousun maailman suosituimmaksi matkapuhelinkäyttöjärjestelmäksi. [4; 5.]

Google kehittää seuraava Android-versiota itsenäisesti, kunnes uudet päivitykset ovat julkaisuvalmiita. Tässä vaiheessa käyttöjärjestelmän lähdekoodi julkaistaan AOSP:ssa. Tämän jälkeen kuka tahansa voi käyttää käyttöjärjestelmää tällaisenaan tai jatkokehittää sitä vapaasti haluamaansa suuntaan. Teoriassa tämä mahdollistaa myös Googlen virallisen Androidin kanssa kilpailevan käyttöjärjestelmien syntymisen. Käytännössä näin ei ole kuitenkaan käynyt. [1; 5.]

Googlen Android ei ole vain käyttöjärjestelmä, vaan kokonainen ekosysteemi, joka sisältää Googlen omien sovellusten lisäksi Google API -palvelut. Nämä lisäpalvelut sekä Android-tavaramerkin voi saada kustomoituun Android-versioon vain lisenssillä, jonka myöntämisessä Google on todella tarkka. Lisenssi myönnetään helpoimmin yrityksille, jotka ovat liittyneet OHAaan. Järjestön ehtoihin kuuluu, ettei siihen kuuluva yritys ole mukana kehittämässä tai levittämässä virallisen Androidin kanssa kilpailevaa käyttöjärjestelmää. Näin Google on onnistunut tehokkaasti estämään Androidin pohjalta kehitettyjen käyttöjärjestelmien leviämisen. Tällaisia versioita on laajassa käytössä vain Kiinassa, jossa Googlen palvelut ovat muutenkin osittain kiellettyjä. [5.]

Suuri osa laitetoimittajista muokkaa avoimen lähdekoodin Androidia lisäten siihen omia suljetun lähdekoodin ominaisuuksia, kuten omia sovelluksiaan ja käyttöliittymiään. Vaikka Android onkin avoimen lähdekoodin käyttöjärjestelmä, huomattava määrä siitä on kuitenkin suljetun lähdekoodin alaista. [5.]

Uusi versio Androidista julkaistaan vuosittain. Vain puhtaan Androidin omaavat laitteet, lähinnä Googlen omat puhelinmallit, saavat päivityksen lähes heti. Muut puhelinmallit, joiden käyttöjärjestelmää on muokattu laitekohtaiseksi, joutuvat yleensä odottamaan päivitysten saapumista useita kuukausia, tai sitten ne eivät saa niitä milloinkaan. Laitevalmistajat joutuvat tekemään jokaiselle puhelinmallille oman päivityksensä, sillä käyttöliittymä on vahvasti sidottu laitteistoon. He joutuvatkin tasapainoilemaan uusien mallien kehityksen ja vanhojen mallien ylläpidon kanssa. [1.] Ratkaisuna tähän Google ilmoitti suunnittelevansa Androidin käyttöjärjestelmäkehyksen uudelleen tukemaan nopeampaa ja tehokkaampaa versionpäivitystä laitetoimittajien käyttöjärjestelmäversioille. Projektin nimi on Project Treble. Se esittelee uuden jakelijarajapinnan, joka tarjoaa pääsyn Androidin laitekohtaisiin osiin erottaen ne muusta käyttöjärjestelmäkehyksestä. Tällöin laitetoimittaja voi päivittää uuden Android-version koskematta työlääseen laitekohtaiseen osaan käyttöjärjestelmän lähdekoodista. [6.]

3.1.3 Versionhallinta

Uusin Android-käyttöjärjestelmäversio tarjoaa aina jotain uusia ominaisuuksia. Sovellusta kehittäessä tulee kuitenkin huomioida se, että uusimman käyttöjärjestelmäversion yleistymisessä kestää usein todella kauan. Tämän vuoksi uusimpia ominaisuuksia ei aina kannata ottaa käyttöön, sillä ne eivät välttämättä toimi suurimmalla osasta käyttäjiä. Taulukossa 1 on esitetty Androidin versiojakauma huhtikuussa 2018. Hyvän tavan mukaista on tukea noin 90 % käyttäjistä. [7.]

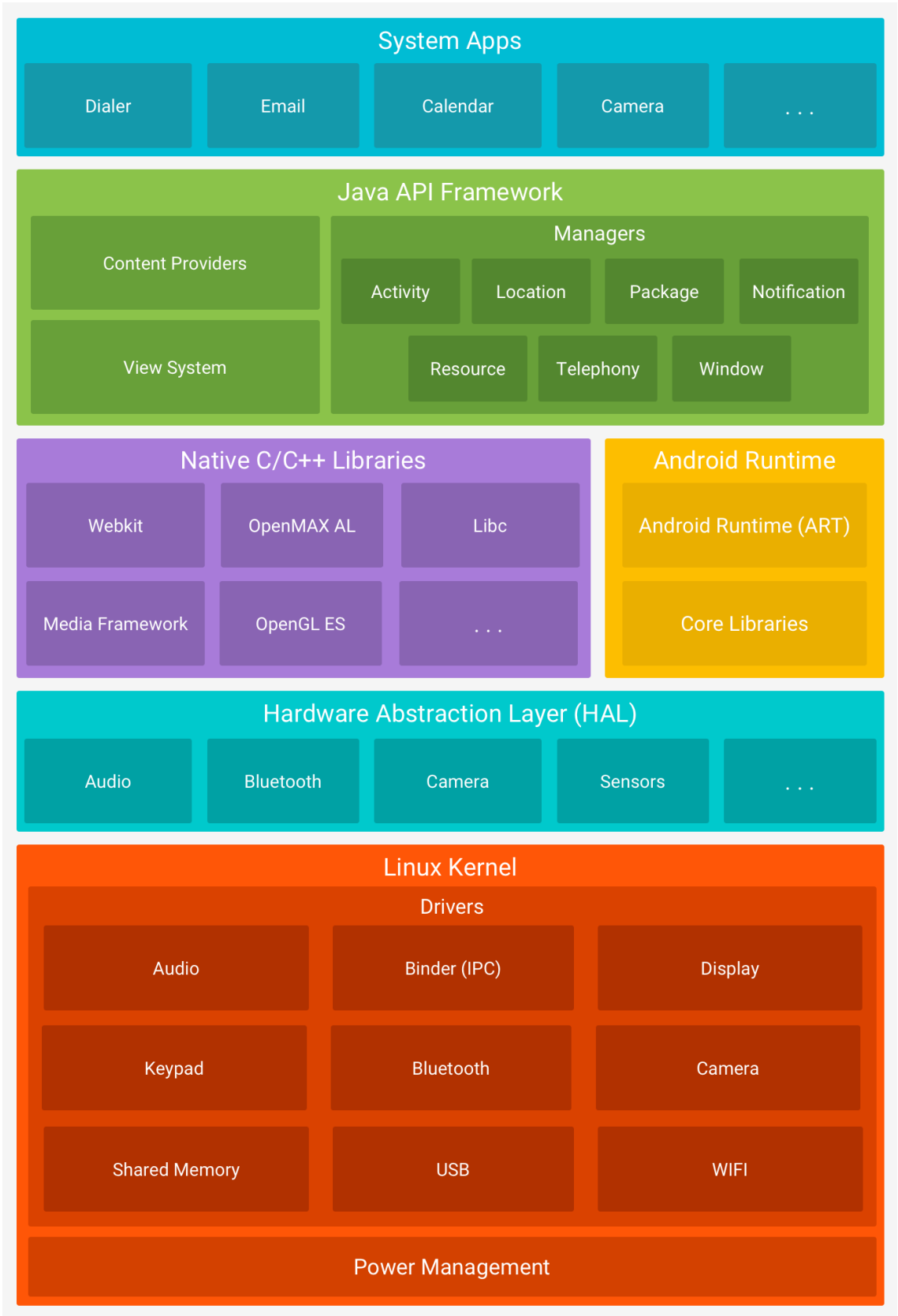
Taulukko 1. Androidin versiojakauma, tilanne huhtikuussa 2018. Taulukosta on poistettu versiot, joilla on alle 0,1 %:n jakauma. [8.]

Versio	**Nimi**	**API**	**Jakauma**
2.3.3 - 2.3.7	Gingerbread	10	0,3 %
4.0.3 - 4.0.4	Ice Cream Sandwich	15	0,4 %
4.1x	Jelly Bean	16	1,7 %
4.2.x		17	2,2 %
4.3		18	0,6 %
4.4	KitKat	19	10,5 %
5.0	Lollipop	21	4,9 %
5.1		22	18,0 %
6.0	Marshmallow	23	26,0 %
7.0	Nougat	24	23,0 %
7.1		25	7,8 %
8.0	Oreo	26	4,1 %
8.1		27	0.5 %

Joidenkin uusien ominaisuuksien käyttäminen vanhoilla laitteilla onnistuu Android Support Library -kirjaston avulla. Näille ominaisuuksille ei tällöin tarvitse tehdä kahta eri toteutusta tukemaan vanhempia laitteita. Joskus tämä ei kuitenkaan ole mahdollista ja vanhoja laitteita varten tulee tehdä toinen toteutus. Tällöin laitteen käyttöjärjestelmäversio tulee tarkistaa ajonaikaisesti. [9.]

3.2 Androidin arkkitehtuuri

Android on Linuxiin perustuva ohjelmistopino. Kuvassa 2 esitellään Android-käyttöjärjestelmän tärkeimmät komponentit.



Kuva 2. Android-ohjelmistopino. [10.]

Androidin pohjana toimii Linuxin käyttöjärjestelmäydin. Se tarjoaa pitkälle kehitetyn ja turvallisen alustan Androidille. Jokainen Android-sovellus pyörii omassa niin kutsutussa hiekkalaatikossaan tarjoten tiettyjä turvallisuusominaisuuksia. Jokainen sovellus on itsenäinen Linux-käyttäjä omine käyttöoikeuksineen. Vain tällä sovelluksella on käyttöoikeus omiin tiedostoihinsa. Jokainen sovellus ajetaan omassa virtuaalisessa ajoympäristössään, eristettynä muista sovelluksista. Sovellukset ajetaan myös omissa Linux-prosesseissaan. Prosessi käynnistetään, kun jotain sovelluksen komponenttia käytetään. Prosessi voidaan myös sulkea, kun sitä ei enää tarvita tai kun muut sovellukset tarvitsevat enemmän muistia. [10; 11.]

Suurin osa nykyisitä laitteista käyttää Linuxin versiota 3.18 tai 4.4, joilla Linux Foundationin pitkäaikainen tuki. Versioon 4.4 asti pitkäaikainen tuki tarkoitti, että sitä tuetaan 2 vuotta julkaisun jälkeen. Google on kuitenkin sopinut Linux Foundationin kanssa, että 4.4:sta alkaen pitkäaikaista tukea jatketaan kuuteen vuoteen asti. [12.]

Käyttöjärjestelmäytimen päällä on laitteiston abstraktiokerros, Hardware Abstraction Layer (HAL). Se määrittelee standartin rajapinnan, jonka avulla korkeamman tason Java ohjelmointirajapintakehys voi kommunikoida laitteiston komponenttien, kuten kameran ja sensorien kanssa. HAL koostuu useista kirjastomoduuleista, josta jokainen on tarkoitettu tietylle komponentille. Oletuksena jokaisella sovelluksella on pääsy vain niihin komponentteihin, joita sovellus käyttää. Jos sovelluksen pitää päästä käsiksi laitteen dataan, esimerkiksi kameraan tai muistikorttiin, on käyttäjältä kysyttävä erikseen lupa. Tämän luvan voi milloin tahansa perua. [10;13.]

Monet Androidin alemman tason komponenteista, kuten HAL ja Android Runtime, on kirjoitettu C- ja C++ ohjelmointikielillä. Tämän vuoksi Androidissa on myös joitakin kirjastoja näille kielille. Sovelluksilla, jotka vaativat C:tä tai C++:aa, on pääsy näihin kirjastoihin Android NDK -työkalulla. [10.]

Android Runtime on Android 4.4:ssä esitelty ajoympäristö, joka tuli vaihtoehtoiseksi ajoympäristöksi Androidin vanhan ajoympäristön Dalvikin rinnalle. Dalvikissa ohjelmakoodi käännetään konekielelle aina ajonaikaisesti (just-in-time compilation), kun taas Android Runtimessa käännös tapahtuu jo sovelluksen asennusvaiheessa (ahead-of-time compilation). Tällöin sovellus ajetaan aina natiivikoodiksi käännetystä versiosta, mikä taas parantaa laitteen suorituskykyä. Koska koodin optimisointi tehdään vain kerran, voidaan siihen käyttää enemmän aikaa ja resursseja kuin aikaisemmin. Tämän vuoksi sovelluk-

sen ensimmäinen käynnistys kestää normaalia pidempään, mutta käyttö on muuten sujuvampaa. Android 5.0:ssa Dalvikista luovuttiin ja Android Runtimesta tuli Androidin ainoa ajoympäristö. [14.]

Java-ohjelmointirajapintakehys, Java API Framework, tarjoaa Androidin alustan kaikki toiminnot Java-rajapintojen avulla. Näitä rajapintoja käytetään Android-sovellusten rakentamiseen. Luettelossa on lyhyt kuvaus tärkeimmistä rakennuspalikoista. [10]

- View Systemiä käytetään sovelluksen käyttöliittymän rakentamiseen. Se sisältää listat, taulukot, painikkeet ja vastaavat komponentit.
- Resource Manager tarjoaa pääsyn resurssitiedostoihin, jotka eivät sisällä koodia. Tällaiset tiedostot sisältävät mm. tekstiä, grafiikkaa ja käyttöliittymän sommitelman.
- Notification Manager tarjoaa sovelluksille mahdollisuuden näyttää hälytyksiä status bar:ssa.
- Activity Manager hallinnoi sovellusten elinkaarta ja tarjoaa yleisen peruutuspinon.
- Content Providerit antavat sovelluksille mahdollisuuden jakaa omaa ja käyttää toisten sovellusten dataa.

Päällimmäisessä kerroksessa ovat itse sovellukset. Käyttöjärjestelmän mukana tulee joitakin yleisimpiä sovelluksia siihen integroituna. Niillä ei ole mitään erikoista asemaa, vaan ne voidaan korvata muilla sovelluksilla. Tällaisia sovelluksia ovat mm. kamera ja puhelusovellukset. [10.]

3.3 Android-sovellus

Android SDK, eli software development kit, tukee virallisesti kolmea ohjelmointikieltä: Javaa, Kotlinia ja C++:aa. Jotkin kolmansien osapuolien kehitystyökalut kuitenkin mahdollistavat Android-sovellusten kirjoittamisen myös muilla ohjelmointikielillä. Virallinen Android SDK yhdessä Javan kanssa on kuitenkin kaikista suosituin yhdistelmä. Tässä luvussa esittelen Android-sovelluksen tärkeimpiä kuvassa 2 esitettyyn Java-ohjelmointirajapintakehykseen kuuluvia komponentteja ja luokkia. [15.]

3.3.1 Manifest

Jokaisen sovelluksen perustana on Android Manifest xml -tiedosto. Siihen on koottu tärkeimmät tiedot, joita Android-rakennustyökalu, Android-käyttöjärjestelmä sekä Google

Play tarvitsevat. Tiedostossa sisältää luettelon kaikista sovelluksen aktiviteeteista, serviceistä, broadcast receivereista ja content providereista. Siinä voidaan myös määritellä, kuinka jotkin sovelluksen osat voi käynnistää. Manifestista ilmenee myös, mitä käyttöoikeuksia sovellus käyttäjältä tarvitsee ja millä laitteilla ja käyttöjärjestelmäversioilla sovellus toimii. [16.]

Esimerkkikoodissa 1 on yksinkertainen esimerkki Android-manifestista. Siinä on ensin määritelty sovelluspakkauksen nimi sekä pienin Android-käyttöjärjestelmäversio, jolla sovellus toimii. Sovellus tarvitsee myös luvan kameran käyttöön. Lupaa kysytään käyttäjältä, kun sovellus käynnistetään tai kun kameraa ollaan käyttämässä. Sovellus sisältää myös yhden aktiviteetin, jolle on asetettu intent filter. Intent filteristä kerron tarkemmin kohdassa Intent. Toimiakseen sovelluksen manifest-tiedoston tulee sisältää manifest-elementin, joka sisältää application-elementin sekä esimerkin mukaisesti jonkin muun elementin sen sisällä. [16.]

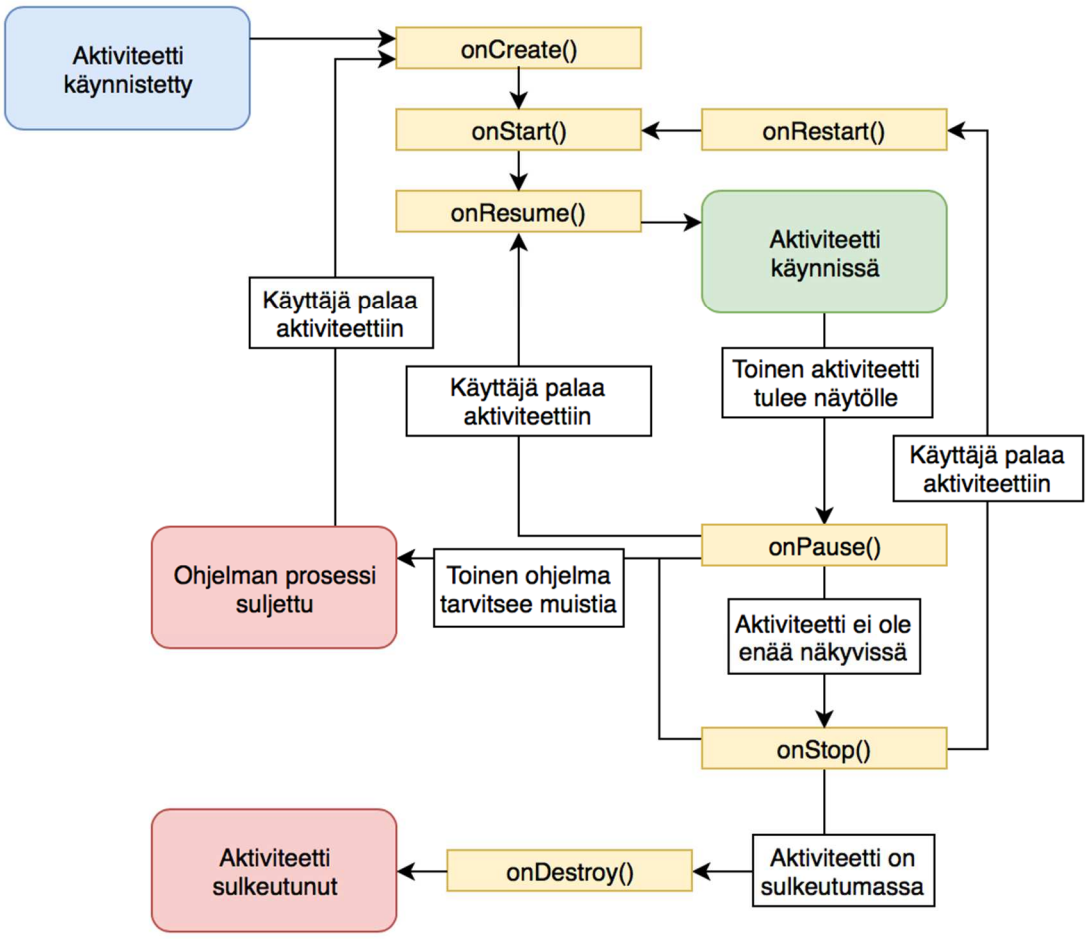
```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
   package="com.example.myapp">
   …
   <uses-sdk android:minSdkVersion=21 android:targetSdkVersion=”27” />
   <uses-permission android:name="android.permission.CAMERA" />
   …
   <application … >
      <activity android:name="com.example.myapp.MainActivity">
         <intent-filter>
            <action android:name="android.intent.action.VIEW"/>
            <category android:name="android.intent.category.DEFAULT"/>
            <data android:mimeType="text/plain"/>
         </intent-filter>
      </activity>
   </application>
   …
</manifest>
```

Esimerkkikoodi 1. AndroidManifest.xml-tiedoston esimerkkisisältö.

3.3.2 Activity

Activityt, eli aktiviteetit, ovat Androidin olennaisimpia rakennuspalikoita. Ne ovat vastuussa käyttöliittymästä, ja näin ollen toimivat vuorovaikutuksessa käyttäjän komentojen kanssa. Ne ovat myös vastuussa liikkumisesta sovelluksen osien ja muiden sovellusten välillä. Sovellus koostuu normaalisti useista aktiviteeteista, joista jokainen toimii itsenäisesti, mutta on samalla vuorovaikutuksessa toistensa kanssa. Sovelluksessa jokainen aktiviteetti voi käynnistää minkä tahansa aktiviteetin sovelluksen sisältä, tai jopa toisesta sovelluksesta. Tämän vuoksi aktiviteettien tulisi olla mahdollisimman riippumattomia toi-

sistaan. Aktiviteetin käynnistyksen toisesta sovelluksesta voi estää sovelluksen manifest-tiedostossa. Tällöin kyseisen aktiviteetin voi käynnistää vain saman sovelluksen aktiviteetista. Jokainen käytössä oleva aktiviteetti tulee esitellä manifest-tiedostossa. Activity Manager kerää aktiviteettejä pinoksi sitä mukaa, kun käyttäjä käynnistää uusia. Kun käyttäjä palaa edelliseen aktiviteettiin, poistaa Acitvity Manager päällimmäisen aktiviteetin pinosta ja näyttää käyttäjälle seuraavan pinosta. [17.]



Kuva 3. Aktiviteetin elämänkaari.

Aktiviteetin tila vaihtuu käyttäjän navigoidessa sovelluksessa sekä muiden sovellusten välillä. Aktiviteetilla on useita callback-metodeita, joiden avulla se tietää tilan vaihtuneen. Aktiviteetti voi olla käynnistymässä, käynnissä, pysähtynyt tai lopetettu. Järjestelmä sulkee aktiviteetin prosessin, mikäli se tarvitsee aktiviteetin käyttämää keskusmuistia aloittaen käyttäjän kannalta vähiten merkityksellisistä prosesseista. [17; 18.]

Kuvassa 3 on esitetty aktiviteetin elämänkaaren eri vaiheet ja missä tilanteessa mitäkin callback-metodia kutsutaan. Ylikirjoittamalla näitä metodeja kehittäjä voi määritellä,

kuinka aktiviteetti reagoi käyttäjän liikkuessa pois ja takaisin aktiviteettiin. Näin voidaan varmistaa sovelluksen toiminta eri tilanteissa. Käyttäjän tekemät muutokset voidaan tallentaa ja palauttaa tilan vaihtuessa. Aktiviteetin kompleksisuudesta riippuu, mitkä metodit kannattaa ylikirjoittaa. Tärkeintä on, että sovellus toimii käyttäjän olettamalla tavalla. [18.]

```
Intent intent = new Intent(this, MyNextActivity.class);
startActivity(intent);
```

Esimerkkikoodi 2. Uuden aktiviteetin käynnistäminen implisiittisen Intent-olion avulla. Aktiviteetin luokkanimi tunnetaan.

Uusi aktiviteetti käynnistetään startActivity()-metodilla, tai startActivityForResult()-metodilla, jos uudesta aktiviteetista halutaan jotain tietoja takaisin. Esimerkkikoodissa 2 on esitetty uuden aktiviteetin käynnistys. Molemmissa tapauksissa tarvitaan Intent-oliota. [18.]

3.3.3 Intent

Intent on abstrakti kuvaus suoritettavasta operaatiosta. Sen avulla aktiviteetti voi antaa käskyjä muille sovelluskomponenteille. Sillä on kolme pääasiallista käyttötarkoitusta. [19.]

- Sitä voidaan käyttää aktiviteetin käynnistykseen.
- Sillä voidaan käynnistää Service-sovelluskomponentti, startService()-metodilla. Tästä komponentista on kerrottu tarkemmin myöhemmin tässä kappaleessa kohdassa Service.
- Sillä voidaan lähettää broadcast viesti, jonka muut sovellukset voivat vastaanottaa.

Kun halutaan käynnistää samaan sovellukseen kuuluva aktiviteetti, jonka luokkanimi tiedetään, käytetään implisiittistä Intent-oliota esimerkkikoodi 2:n tavoin. Intent-olion avulla voi käynnistää myös sovelluksen ulkopuolinen aktiviteetti, jonka nimeä ei tiedetä. Tällöin eksplisiittisen Intent-olion avulla kuvataan käyttöjärjestelmälle haluttu toiminto. Esimerkkikoodi 3:ssa halutaan käynnistää verkkoselain antamalla intentille parametrit Intent.ACTION_VIEW ja näytettävä verkko-osoite. [19.]

```
Intent browserIntent = new Intent(Intent.ACTION_VIEW, url);
startActivity(browserIntent);
```

Esimerkkikoodi 3. Uuden aktiviteetin käynnistäminen eksplisiittisen Intentin avulla. Aktiviteetin luokkanimeä ei tunneta, vaan järjestelmä päättää, mikä sovellus toteuttaa käskyn.

Esimerkkikoodissa 3 kuvatussa tilanteessa aktiviteetti lähettää käyttöjärjestelmälle pyynnön käynnistää sovellus, jolla toiminto voidaan suorittaa. Se käy läpi kaikkien sovellusten manifest-tiedostot ja etsii sieltä aktiviteetteja, joista löytyy intent filter -elementti esimerkkikoodi 1:ssä esitetyllä tavalla. Käyttäjälle näytetään valintadialogi, mikäli tarkoitukseen soveltuvia sovelluksia on useampi. On myös mahdollista, ettei puhelimelle ole yhtään tällaista sovellusta löydy, jolloin uutta aktiviteettia ei voida käynnistää. [19; 20.]

Datan siirtäminen aktiviteettien ja eri sovellusten välillä onnistuu Parcel-luokan avulla. Se on kevyt ja tehokas viestisäiliö, jonka avulla dataa voidaan siirtää dataa laitteen eri osien välillä. Jotta olion data voidaan kirjoittaa ja lukea Parcelista, tulee sen toteuttaa Parcelable-rajapinta. Tämä rajapintaluokan avulla olio pilkotaan palasiksi tallentamista varten ja kootaan myöhemmin muuttujien omien ClassLoader-luokkien avulla. Toinen tapa siirtää dataa on Bundle, joka on käytännössä avainarvopari. Tämä on yleinen tapa siirtää dataa fragmenttien välillä ja myös tällöin siirrettävä data toteuttaa Parcelable-rajapinnan. [20; 21.]

```
public static SubCategoryFragment newInstance(@NonNull Category category) {
    SubCategoryFragment fragment = new SubCategoryFragment();
    Bundle bundle = new Bundle();
    bundle.putParcelable (“category”, category);
    fragment.setArguments(bundle);
    return fragment;
}
```

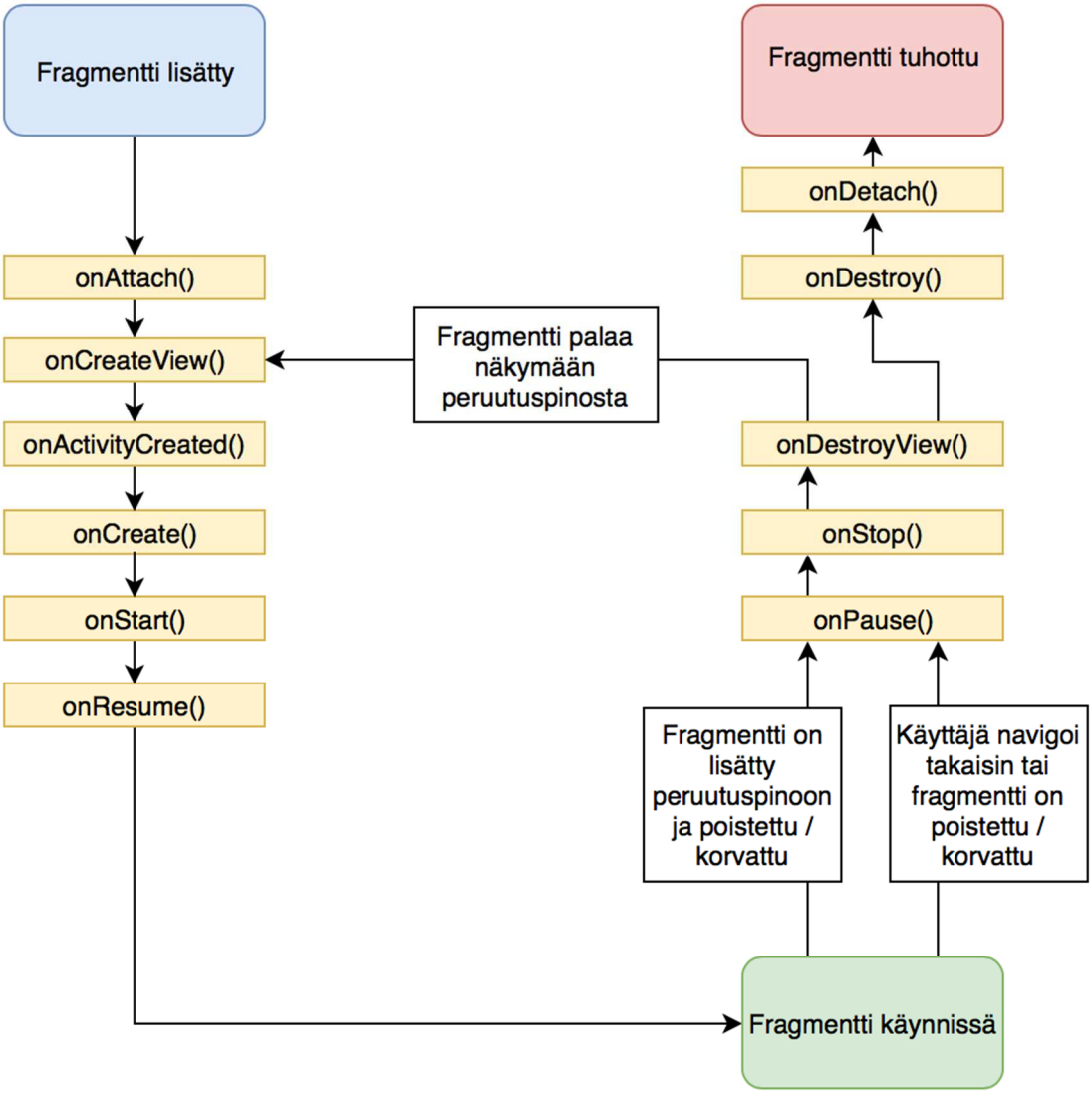
Esimerkkikoodi 4. SubCategoryFragmentin newInstance-metodi, Category-luokka toteuttaa Parcelable-rajapinnan.

Esimerkkikoodissa 4 on kuvattu SubCategoryFragmentin newInstance-metodi, jota kutsutaan, kun fragmentti halutaan käynnistää. Tällöin parametrina annettu Category kirjoitetaan Bundleen ja se asetetaan käynnistettävän fragmentin argumentiksi, johon fragmentti pääsee myöhemmin käsiksi getArguments()-metodilla.

3.3.4 Fragment

Fragmentit edustavat tiettyä käyttöliittymän osaa tai toimintoa aktiviteetin sisällä. Aktiviteetti hallinnoi yleensä useaa fragmenttia, jotka voivat olla ruudulla samanaikaisesti. Fragmenteillä on myös peruutuspino, johon Fragment manager kerää niitä käyttäjän liik-

kuessa sovelluksessa eteenpäin ja josta ne tuodaan taas esiin käyttäjän palatessa edelliseen näkymään. Fragmentti on sidottu omaan aktiviteettiinsa, eikä sitä voi käyttää ilman sitä. Vaikka fragmentilla onkin oma elämänkaari, ei se voi olla ristiriidassa oman aktiviteettinsa elämänkaaren kanssa. Jos aktiviteetti on pysähtynyt tai lopetettu, myös kaikki sen fragmentit ovat samassa tilassa. Fragmentin elämänkaari on esitelty kuvassa 4. [23.]



Kuva 4. Fragmentin elämänkaari.

Tärkeimpiä metodeita ovat onCreate, onCreateView sekä onPause. Yleensä ainakin näeille metodeille tulisi tehdä oma toteutus. OnCreate-metodia kutsutaan, kun systeemi luo fragmentin. Omassa toteutuksessa tulisi alustaa fragmentin tärkeimmät komponentit, jotka halutaan säästää, kun fragmentti pysäytetään. OnCreateView-metodia kutsutaan, kun fragmentti on piirtämässä käyttöliittymää ensimmäisen kerran. Tässä vaiheessa

määritetään, mikä layouttiedosto toimii näkymän juurena. Layouttiedostoista on kerrottu luvussa 3.3.6 Resurssitiedostot. OnPause-metodi indikoi siitä, että käyttäjä on poistumassa fragmentista. Käyttäjä ei enää välttämättä palaa tämän jälkeen fragmenttiin. Jos käyttäjä on tehnyt tässä fragmentissa jotain muutoksia, esimerkiksi muuttanut joitakin asetuksia, ne kannattaa tallentaa viimeistään tässä vaiheessa. [23.]

3.3.5 View

View-luokka on Android-sovelluksen käyttöliittymäkomponenttien perusta. View kattaa nelikulmaisen alueen näkymästä ja on vastuussa oman sisältönsä piirtämisestä ja tapahtumien hallinnoinnista. Se on interaktiivisten käyttöliittymäkomponenttien, kuten painikkeiden ja tekstinäkymien, kantaluokka. Työssä käytettyjä käyttöliittymäkomponentteja ovat muun muassa Button ja TextView. View'n alaluokka ViewGroup toimii kantaluokkana näkymäsäiliöille, jotka voivat sisältää käyttöliittymäkomponentteja tai toisia säiliöitä. Tällaisia säiliöitä joita työssä olen käyttänyt ovat mm. LinearLayout, RecyclerView, CardView ja CoordinatorLayout. [24.]

RecyclerView on suurien ja dynaamisten datasettien esittämiseen käytetty listanäkymä. Se täyttää itsensä layout-managerin avulla syötetyillä käyttöliittymäkomponenteilla. Jokaista lista-alkiota varten luodaan ViewHolder, joka on vastuussa tämän komponentin näyttämisestä näkymässä. ViewHoldereista vastaa Adapter-luokka, joka sitoo niihin tarvittavan datan. RecyclerView luo lista-alkioita vain hiukan enemmän, kuin näkymään mahtuu. Samoja lista-alkioita käytetään uudelle sitä mukaan, kun ne poistuvat näkymästä käyttäjän vierittäessä listaa. Vanhaan lista-alkioon vain syötetään seuraavan alkion data juuri ennen kuin se tulee näkyviin. Tällöin suuriakin listoja voidaan hallinnoida pienellä kapasiteetilla. Jos listassa oleva data muuttuu, voidaan siitä ilmoittaa RecyclerView.Adapter.notify-metodilla. Tällöin adapter sitoo muuttuneiden lista-alkioiden datan uudelleen. [25.]

3.3.6 Resurssitiedostot

Resurssitiedostot ovat erillisiä tiedostoja, joissa sijaitsee Androidin staattinen data. Tiedostot ovat yleensä xml-tiedostoja, ja ne sisältävät tekstiä, kuvia, layout-määrityksiä ja tyylitiedostoja. Layout-tiedostoissa on fragmenttien ja muiden käyttöliittymäkomponenttien määritykset. Androidin näkymä koostuu yhdestä puumaisesta rakenteesta ja sen luomiseen käytetään näitä tiedostoja. Esimerkkikoodissa 5 on alakategorianäkymässä

käytetty layouttiedosto. Se koostuu LinearLayoutista ja sen sisällä olevasta RecyclerViewistä. Tässä tiedostossa on määritetty myös komponenttien koko, taustaväri sekä reunojen väliin jäävä tyhjä tila. Myös niiden arvot on kirjoitettu toisiin resurssitiedostoihin. Jokaiselle komponentille, johon halutaan päästä käsiksi koodista, on annettava yksilöllinen tunnus android:id-atribuutin avulla. [26.]

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingEnd="@dimen/catalog_default_padding"
    android:paddingStart="@dimen/catalog_default_padding"
    android:background="@color/primary_background"
    android:orientation="vertical">

    <android.support.v7.widget.RecyclerView
        android:id="@+id/household_catalog_subcategory_rc_view"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

</LinearLayout>
```

Esimerkkikoodi 5. SubCategoryFragment:n layout-tiedosto

Koska tuotekatalogi julkaistaan kaikissa Pohjoismaissa, tulee sen toimia useilla kielillä. Vaikka suurin osa sisällöstä tuleekin palvelimelta, ovat jotkin tekstit sisällytettävä sovellukseen. Tätä varten on tarjolla lokalisoituja resurssitiedostoja, joissa on käännökset teksteille. Sovelluksessa käytetyt tekstit on sijoitettu tiedostopolkuun res/values/strings.xml. Tiedostosta res/values/<maakoodi>/strings.xml voidaan hakea maakoodin mukaiset käännökset teksteille. Jos jollekin tekstille ei löydy käännöstä, käytetään silloin oletustiedostossa olevaa tekstiä. [27.]

3.4 Ulkoiset kirjastot

Tässä luvussa kerron työssä käytetyistä ulkoisista kirjastoista.

3.4.1 Picasso

Picasso on Square Open Sourcen kehittämä Android-kirjasto, joka yksinkertaistaa kuvien latausta verkosta sovellukseen. Se automatisoi monia Androidin sudenkuoppia, kuten ImageView'n kierrätyksen ja kuvan latauksen peruutuksen adapterissa, monimutkaiset kuvien transformaatiot minimaalisella muistin käytöllä sekä automaattisen kuvien väliaikaisen varastoinnin. [28.]

```
Picasso.get()           // Kutsutaan luokan singleton instanssia
  .load(url)            // kuvan lataus verkosta
  .resize(50, 50)       // kuvan koon määrittäminen
  .centerCrop()         // kuvan keskitys ja rajaus
  .into(imageView)      // kuvan asettaminen näkymään
```

Esimerkkikoodi 6. Kuvan lataaminen, muokkaaminen ja asettaminen näkymään Picasson avulla.

Esimerkkikoodi 6:ssa on havainnollistettu Picasson käytön yksinkertaisuus ja selitetty eri metodien tarkoitus.

3.4.2 GSON

Gson on Googlen kehittämä Java-kirjasto, joka kääntää Java-objektit JSON-muotoon ja toisin päin. Vastaavia kirjastoja löytyy jonkin verran, mutta Gsonissa on joitain hyödyllisiä ominaisuuksia, mitä muissa kirjastoissa ei ole. Ensinnäkin se tukee Javan geneerisiä ominaisuuksia. Lisäksi Java-luokkiin ei tarvitse muista poiketen lisätä Java-annotaatioita, eli huomautuksia, vaan halutun luokan ja sen kaikkien yläluokkien tietokentät ovat automaattisesti mukana käännöksessä. [29.]

```
class EsimerkkiLuokka {
    @SerializedName(“value_1”)
    private int value1 = 1;
    private String value2 = “abc”;
    EsimerkkiLuokka() {
        //tyhjä konstruktori }
}

// JSON:n tekeminen EsimerkkiLuokka oliosta
EsimerkkiLuokka obj = new EsimerkkiLuokka();
Gson gson = new Gson();
String json = gson.toJson(obj);
// ==> saadaan json: {"value_1":1,"value2":"abc"}

// EsimerkkiLuokka olion luominen JSON:n pohjalta
EsimerkkiLuokka obj2 = gson.fromJson(json, EsimerkkiLuokka.class);
// ==> obj2-olio vastaa obj-oliota
```

Esimerkkikoodi 7. Gsonin käyttö JSON <-> olio muunnoksissa. [29.]

Yllä olevassa esimerkkikoodissa 7 on esitelty, kuinka oliosta saadaan JSON ja kuinka JSON:sta luodaan olio. Kuten aikaisemmin jo totesin, Gson ei tarvitse annotaatioita. Kuitenkin niitä voi käyttää esimerkiksi silloin, jos jokin muuttuja halutaan esitellä eri nimellä koodissa, kuin mitä se on JSON-tiedostossa. Jokin kenttä saattaa JSON-tiedostossa olla nimetty tyyliin ”esimerkki_kentta”, mutta koodissa sanat erotellaan alaviivan sijaan isolla kirjaimella ”esimerkkiKentta”. Tällöin voidaan käyttää @SerializedName -annotaatiota, kuten ylläolevan esimerkkikoodi 7:n rivillä 2 on tehty. Muita käytössä olevia annotaatioita ovat @Expose, joka poistaa muunnoksesta merkityt kentät, sekä @Since, joka auttaa versionhallinnassa. [29.]

3.4.3 RxJava

RxJava on reaktiivisen ohjelmoinnin mahdollistava Java-kirjasto. Se tuo asynkronisia ja tapahtumapohjaisia ominaisuuksia sovellukseen. RxJavan pääkomponentteja ovat Observable (datan lähde), Observer eli observoija (reagoiva osapuoli) ja Operator (tietovirran manipuloija). Observer-olio ”tilaa” Observable-luokan olion ja reagoi sen lähettämään dataan. Tähän väliin voi asettaa operator-metodeita muokkaamaan kyseistä dataa. Observer on aina valmiina reagoimaan uuteen dataan sen sijaan, että se kysyy jotain tietoa ja jää odottamaan vastausta. Tällöin sovelluksen toiminta ei keskeydy, vaan tapahtumiin reagoidaan tarpeen tullen. Tämän insinöörityön tapauksessa RxJavaa käytetään taustapalvelukutsuissa. [30.]

3.5 Model-View-Presenter-arkkitehtuuri

Tässä luvussa kerron sovelluksen arkkitehtuurin pohjana käytetystä model–view–presenter-arkkitehtuurimallista.

3.5.1 MVP

MVP on 1990-luvun alussa Applen, IBM:n ja HP:n yhteisyritys Taligentin kehittämä arkkitehtuurimalli, jossa kuvan 32 mukaisesti: [31; 32.]

- Model, kuvaa dataa, joka esitetään käyttöliittymässä. Se vastaa sovelluksen bisneslogiikasta.
- View on käyttöliittymä, joka esittää modelin datan ja ohjaa käyttäjän komennot presentterille. Se vastaa käyttöliittymälogiikasta.
- Presenter tuntee sekä view'n että modelin ja toimii näiden välissä. Se on vastuussa kaikesta ohjelmalogiikasta.

Kuva 5. MVP-arkkitehtuurimallin toimintaperiaate. [32.]

View tuntee sovelluksen käyttöliittymän ja toimintaympäristön. Se näyttää käyttäjälle näkymän presenterin ohjeiden mukaisesti. Listener-metodit, kuten esimerkiksi onClickListener(), ovat view'n vastuulla. Näin view tietää, kun käyttäjä tekee jonkin ennalta määrätyn toiminnon. View kutsuu haluttua presenterin metodia ja jää odottamaan uusia ohjeita presentteriltä. View,n tulee olla mahdollisimman yksinkertainen, joten se tekee vain sen, mitä presenter käskee. Myös latausruudun näyttäminen käyttäjän komennon jälkeen tapahtuu vain, jos presenter niin määrää. [32.]

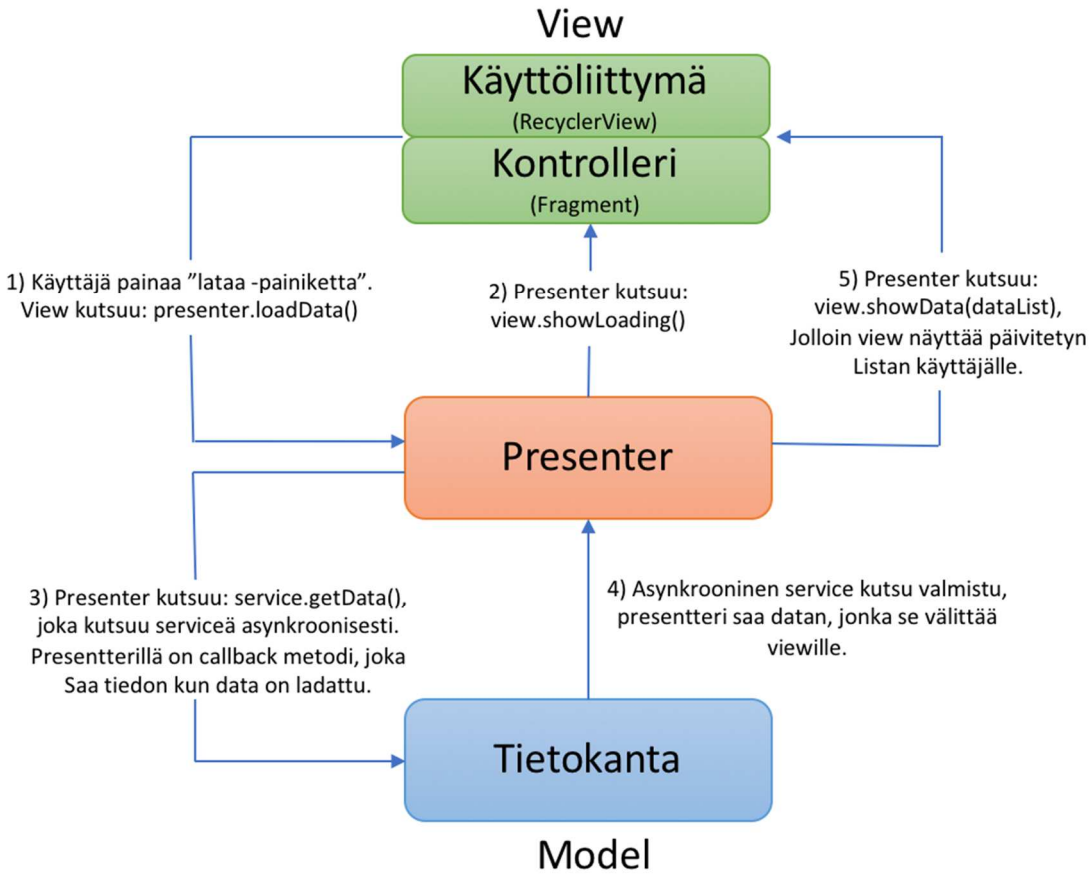
Presenter saa view'ltä käyttäjän antamat komennot. Presenter ei voi kuitenkaan olla vastuussa listener metodeista, sillä muutoin sen tulisi tuntea käyttöliittymä ja toimintaympäristö. Sen sijaan view kutsuu haluttua presenterin metodia käyttäjän komentojen mukaisesti. Presenter kontrolloi view'tä mahdollisimman abstraktisti. [32.]

MVP-arkkitehtuuri pyrkii siihen, että vain view tuntee sovelluksen toimintaympäristön, jolloin presenter ja model voisivat teoriassa toimia myös muissa ympäristöissä. Esimerkiksi jos sovellus on tehty androidille, vain view sisältää androidin omaa koodia. Näin ollen samaa presenteriä ja modelia voidaan käyttää jonkin toisen java-applikaation kanssa ja vain view täytyy tehdä uudelleen. Tämä helpottaa myös presentterin yksikkötestausta, kun testejä ei tarvitse ajaa Android-alustalla. [32.]

3.5.2 Mosby MVP

Mosby MVP on Model-View-Presenter-kirjasto Androidille. Se mahdollistaa modernien android-sovellusten luonnin Model-View-Presenter-arkkitehtuuria hyödyntäen. Mosby tukee myös Kotlin-ohjelmointikieltä, mutta tässä työssä käsitellään vain Java-toteutusta. [33.]

Kuten jo edellisessä luvussa totesin, presenter vastaa kaikesta ohjelmalogiikasta, mutta se ei kuitenkaan toimi näkymän kontrollerina. Androidissa kontrollerin tehtävää hoitaa aktiviteetti tai fragmentti. Presenter on yhteydessä fragmenttiin MvpView-rajapinnan kautta. Presenter ei tiedä näkymästä mitään muuta kuin rajapinnan tarjoamat metodit. Kuvassa 6 on esitetty MVP:n toimintalogiikka Androidilla. Käyttäjä painaa käyttöliittymässä olevaa painiketta. Käyttäjän komentojen kuuntelu on kontrollerin, eli fragmentin vastuulla. Fragmentti kutsuu presenterin loadData-metodia, jolloin presenter pytää fragmenttia näyttämään latausindikaattorin MvpView-rajapinnan kautta. Presenter ei ota kantaa siihen, millainen latausindikaattorin tulee olla, vaan se on fragmentin tehtävä. Presenter lataa pyydetyn datan modelista, eli tietokannasta. Se luo datan perusteella listan olioita ja välittää sen MvpView-rajapinnan kautta fragmentille. Näin presenter käskee view'tä näyttämään listan käyttäjälle, mutta käyttöliittymän päivitys on jälleen fragmentin vastuulla. [32.]



Kuva 6. Esimerkki MVP:n toiminnasta Androidissa, kun käyttäjän käskystä ladataan dataa tietokannasta ja näytetään se listana käyttäjälle. [32.]

Mosby tarjoaa oman presenterin pohjalla käytettävää MvpBasePresentteriä, jossa on heikko viittaus view’n ja näin suojaa sovellusta muistivuodoilta. Tämän vuoksi presenterin tulee aina tarkistaa, onko view liitetty presentteriin isViewAttached-metodilla, ennen kuin se antaa sille komentoja. Vaihtoehtoisesti voi käyttää MvpNullObjectBasePresenteriä, jolloin view’n puuttuessa luodaan tyhjä view ja liitetään se presentteriin. Tällöin presenter ei voi päivittää näkymää, mutta sovellus ei myöskään kaadu view’n puuttumisen vuoksi. Mosby tarjoaa myös MvpActivityn ja MvpFragmentin, jotka molemmat toteuttavat MvpView’n. [32.]

4 Toteutus

Toteutin insinöörityön osana päivätyötäni. Aloittaessani työt Nordeassa it-kehittäjänä liityin tiimiin, joka oli kehittämässä tuotekatalogia Nordean uuteen mobiilisovellukseen.

Projekti oli aloitettu joitakin kuukausia aikaisemmin, ja sovelluksen arkkitehtuuri ja yleisilme oli vielä muovautumisvaiheessa. Tuotekatalogia kehitettiin samanaikaisesti Android- ja iOS-käyttöliittymille. Tiimistä puuttui Android-kehittäjä Scrum Masterin hoitaessa tätä tehtävää. Minulla ei ollut aikaisempaa kokemusta Androidista, vaikkakin Java oli minulle tuttu ohjelmointikieli. Totesin tämän olevan täydellinen aihe insinöörityölleni sen tarjoaman haasteen ja kehitysmahdollisuuksien suhteen.

Olin osa noin kymmenhenkistä Scrum-tiimiä, joka oli osa yli satahenkistä mobiilipankin kehitystiimiä. Sain kuitenkin kädet tekniseen toteutukseen, kunhan vain lopputulos täyttää annetut määritykset ja kirjoittamani koodi on laadukasta. Tarvittaessa sain tukea muilta tiimin jäseniltä ja projektin Android-kehittäjiltä.

Ensin käytin paljon aikaa tutustuakseni sovelluksen, arkkitehtuuriin sekä eri työkaluihin. Tässä vaiheessa tuotekatalogiin oli jo tehty pari näkymää, mutta hyvin pian aloittamiseni jälkeen käyttöliittymä suunniteltiin kokonaan uudelleen. Pääsin siis aloittamaan työn käytännössä katsoen puhtaalta pöydältä, mutta pystyin kuitenkin käyttämään vanhaa toteutusta mallina ja opetteluun. Etenin työn kanssa lineaarisesti ensimmäisestä näkymästä viimeiseen, palaten lopussa vielä hiomaan kaikista työläimpiä osuuksia. Myös joitakin muutoksia suunnitelmiin tuli matkan varrella. Pidin koko ajan myös yksikkötestit ja testiautomaation ajan tasalla. Työkaluista tärkeimmät olivat IntelliJ:in pohjautuva Android-kehitysympäristö Android-Studio sekä Git-versionhallintatyökalu.

4.1 Palvelinrajapinta

Tuotekatalogin on yhteydessä kahteen REST-palvelinrajapintaan: kategoria- ja tuoterajapintaan. Kategoriarajapinta palauttaa JSON-muotoisen listan kategorioista. Kategoria koostuu nimestä, kuvauksesta, taustakuvan osoitteesta ja listasta alakategorioita. Alakategoria sisältää myös nimen, kuvauksen, taustakuvan osoitteesta ja niiden lisäksi listan tuotekuvauksia. Tuotekuvauksesta löytyy tuotteen nimi, lyhyt kuvaus ja taustakuvan osoite. Näiden avulla mallinnetaan määrittelyn mukaisesti kuvassa 1 esitellyt kaksi ensimmäistä näkymää: kategorialuettelo ja alakategorioihin jaettu tuoteluettelo.

Toinen rajapintakutsu, tuoterajapinta, palauttaa halutun tuotteen yksityiskohtaiset tiedot JSON-muodossa. Tuote koostuu nimestä, myyntilauseesta, taustakuvan osoitteesta, listasta call to actioneita sekä sisältöelementeistä. Sisältöelementeillä on otsikko ja sisältö

sekä lista alasisältöelementeistä. Tämän rajapintakutsun avulla mallinnetaan määrittelyn mukaisesti kuvassa 1 esitellyt kaksi viimeistä näkymää: tuotesivu ja lisätietosivu.

Rajapintakutsuja varten luotiin sovellukseen Api-rajapintaluokka, jossa on kaksi get-rajapintakutsua palvelinrajapintaan. Varsinainen yhteys palvelinrajapintaan on hoidettu toisaalla sovelluksessa ja on tämän insinöörityön rajauksen ulkopuolella. Sovelluksen muut osat pääsevät palvelinrajapintaan käsiksi Handler-luokan kautta. Siellä on molemmille rajapinnoille toteutus.

```
public Observable<CatalogCategories> getCategories() {
    return api.getCategories()
            .observeOn(AndroidSchedulers.mainThread());
}
```

Esimerkkikoodi 8. Handler-luokan toteutus kategorioiden noutamiseen.

Esimerkkikoodissa 8 on esitelty Handler-luokan getGategories-metodi, joka kutsuu Api-luokan vastaavaa metodia, joka palauttaa listan kategorioista. Se palauttaa Api-luokan palauttaman Observable-olion, joka käsitellään Androidin pääsäikeessä observeOn-metodin avulla. Vastaus on käsiteltävä pääsäikeessä, jos observoija tekee päivityksiä näkymään. Palvelimelta saatu JSON-muotoinen vastaus käännetään Java-olioiksi GSON:in avulla. Tarkempi kuvaus GSON:in toiminnasta on kerrottu luvussa 3.4.2.

4.2 Model

Kun oli selvillä, kuinka yhteys palvelimeen toimii ja minkälaisia olioita palvelinrajapinta palauttaa, aloitin modelin rakentamisesta. Model, kuten kappaleessa 3.5.1 esitin, kuvaa model-view-presenter-arkkitehtuurissa bisneslogiikkaa. Mallia varten luotiin paketti, joka sisältää kaikki Java-luokat, jotka näkymä esittää. Nämä luokat mallintavat palvelimelta saatavaa JSON-muotoista dataa, joka GSON-kirjaston avulla muutetaan Java-olioiksi. Nämä Java-luokat on kuitenkin ensin rakennettava.

Modelin rakentamiseen on olemassa erilaisia työkaluja, kuten mm. jsonschema2pojo, jotka luovat Java-objekti rakenteen JSON-mallin pohjalta. Tahdoin kuitenkin tehdä tuon työvaiheen käsin, jolloin pääsin syvällisemmin tutustumaan mallinnettavien olioiden rakenteeseen.

```
public class Category implements SelfValidating, Parcelable {
```

```
    @NonNull private final String type;
    @SerializedName("sub_categories")
    @NonNull private List<SubCategory> subCategories;
    @NonNull private final String id;
    @NonNull private final String description;
    @NonNull private final String name;
    @SerializedName("background_image")
    @Nullable private final String backgroundImage;

    protected Category(Parcel in) {
        type = in.readString();
        subCategories = in.readParcelable(SubCategory.class.getClassLoader());
        id = in.readString();
        description = in.readString();
        name = in.readString();
        backgroundImage = in.readString();
    }

    public static final Creator<Category> CREATOR = new Creator<Category>() {
        @Override
        public Category createFromParcel(Parcel in) {
            return new Category(in);
        }

        @Override
        public Category[] newArray(int size) {
            return new Category[size];
        }
    };

    @Override
    public int describeContents() {
        return 0;
    }

    @Override
    public void writeToParcel(Parcel dest, int flags) {
        dest.writeString(type);
        dest.writeParcelable(subCategories);
        dest.writeString(id);
        dest.writeString(description);
        dest.writeString(name);
        dest.writeString(backgroundImage);
    }

    @Override
    public void validate() {
        AssertUtils.assertNonEmpty("type", type);
        AssertUtils.assertNonNull("subCategories", subCategories);
        AssertUtils.assertNonEmpty("id", id);
        AssertUtils.assertNonEmpty("description", description);
        AssertUtils.assertNonEmpty("name", name);
        AssertUtils.assertNullOrNonEmpty("backgroundImage", backgroundImage);

        for (SubCategory subCategory : subCategories) {
            subCategory.validate();
        }
    }
}
```

Esimerkkikoodi 9. Category-luokka, joka mallintaa tuotekategoriaa, on osa MVP-arkkitehtuurin Model-osiota.

Model-luokat ovat tässä insinöörityössä kaikki enemmän tai vähemmän toistensa kaltaisia. Esimerkkikoodissa 9 on esitetty Category-luokan toteutus, josta on siivottu getter- ja

setter-metodit. Se koostuu muutamasta String-muuttujasta ja SubCategory-listasta. SelfValidate on projektissa käytetty rajapinta, jonka validate-metodi tarkistaa olion oikeellisuuden hallitusti pian luomisensa jälkeen. Tämä varmistaa palvelimelta saadun datan on oikeellisuuden, eikä sitä tarvitse enää myöhemmin tarkistaa. Toinen rajapinta, jonka Category-luokka toteuttaa on Parcelable. Sitä tarvitaan, jotta dataa voidaan siirtää tehokkaasti fragmentilta toiselle Bundlen avulla. Tätä varten luokassa tulee olla toteutus datan kirjoittamiseksi Parceliin ja lukemiseksi siitä.

4.3 Kategoriasivu

Tuotekatalogi löytyy sovelluksen alavalikossa Lisää-painikkeen alta. Tuotekatalogilla ei ole omaa aktiviteettia, vaan se toimii samassa aktiviteetissa sovelluksen päänäkymän kanssa. Kategoriasivu on rakennettu Mosby-arkkitehtuurimallin mukaisesti view'sta, presentteristä ja modelista.

View koostuu fragmentista ja view-rajapinnasta, jonka fragmentti toteuttaa ja jonka presenter tuntee. Loin CategoryView-rajapintaluokan, joka on MvpView-rajapinnan alaluokka. CategoryView-rajapinnalla on neljä metodia, jotka on esitetty esimerkkikoodissa 10.

```
public interface CategoryView extends MvpView {
    void updateCatalogItems(@NonNull List<CategoryListItem> itemList);
    void enableWaitingState();
    void disableWaitingState();
    void showError();
}
```

Esimerkkikoodi 10. CategoryView toimii rajapintana CategoryPresentterin ja CategoryFragmentin välillä.

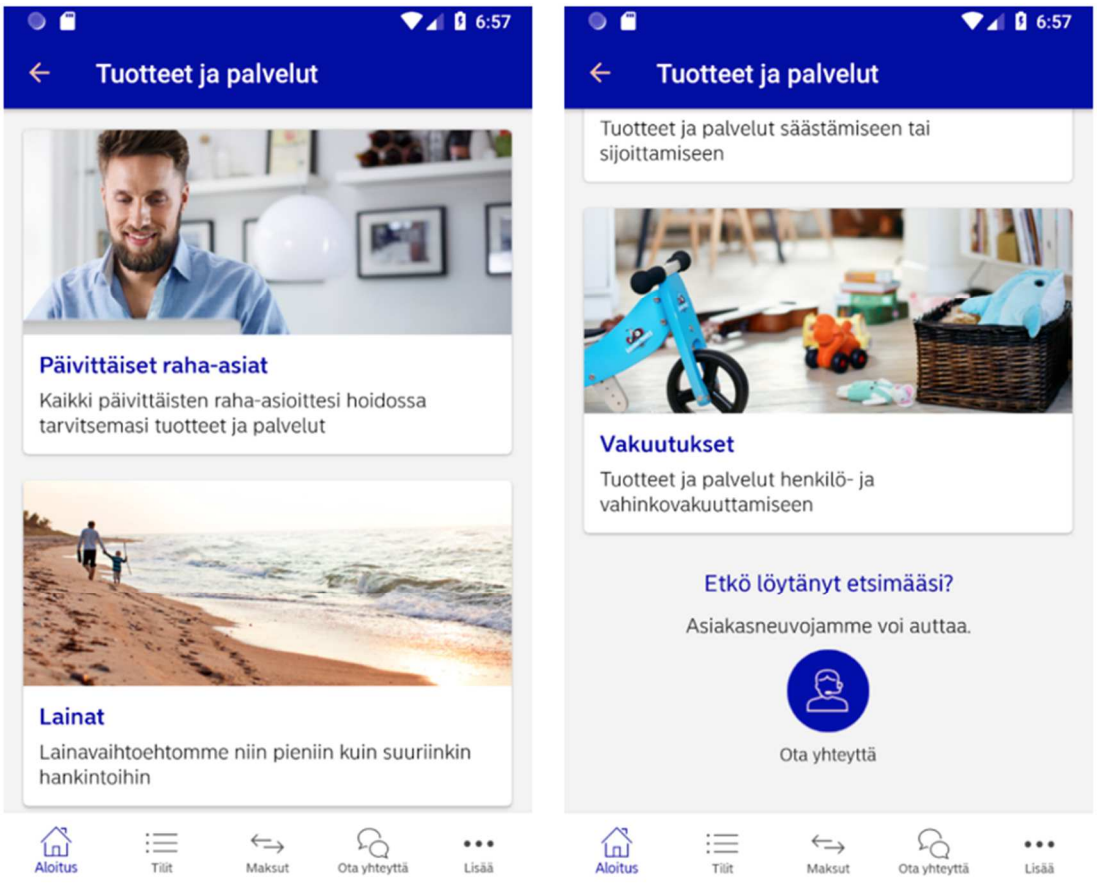
CategoryFragment, joka on MvpFragment-luokan alaluokka, on vastuussa näkymän komponenteista. Näkymää varten on luotu oma category_fragment-resurssitiedosto, jossa on näkymän pohjapiirros xml-muodossa. Näkymä koostuu kolmesta eri komponentista, jotka kaikki peittävät koko näytön. Ne kaikki edustavat eri tilaa, jossa näkymä voi olla. Ensin esillä on latausruutu, jonka jälkeen näytetään sisältösivu tai virhesivu, jos latauksen aikana ilmeni jokin virhe. Sisältö näytetään RecyclerView-komponentista, joka sopii täydellisesti dynaamisen sisällön näyttämiseen. Fragmentin onCreate-metodissa luodaan tätä varten CategoryViewAdpater-olio, joka on vastuussa RecyclerView'n listan sisällöstä.

```
@Override
public CategoryItemViewHolder onCreateViewHolder(@NonNull ViewGroup parent,
int viewType) {
    CategoryItemViewHolder categoryItemViewHolder;
    View v;
    switch (viewType) {
        case CategoryListItem.VIEWTYPE_CATEGORY_ITEM:
            v = LayoutInflater.from(parent.getContext()).inflate(
                    R.layout.category_cardview, parent, false);
            categoryItemViewHolder = new CategoryViewHolder(v);
            break;

        case CategoryListItem.VIEWTYPE_CONTACT_ITEM:
            v = LayoutInflater.from(parent.getContext()).inflate(
                    R.layout.contact_item, parent, false);
            categoryItemViewHolder = new ContactViewHolder(v);
            break;

        default:
            throw new IllegalArgumentException(
             "Unknown view type" + viewType);
    }
    return categoryItemViewHolder;
}

class CategoryViewHolder extends CategoryItemViewHolder<Category> {

    …

    @Override
    void bind(CategoryListItem<Category> item) {
        final Category catalogItem = item.getItem();
        title.setText(item.getItem().getName());
        description.setText(item.getItem().getDescription());
        picasso.get()
                .load(item.getItem().getBackgroundImage())
                .centerInside()
                .fit()
                .into(image);

        itemView.setOnClickListener(v -> {
            for (Listener l : listeners) {
                l.onCategoryItemClicked(catalogItem);
            }
        });
    }
}

public interface Listener {
    void onCategoryItemClicked(@NonNull Category item);

    void onCallToActionClicked(@NonNull CallToAction callToAction);
}
```

Esimerkkikoodi 11. CategoryViewAdapter-luokan onCreateViewHolder-metodi ja CategoryViewHolder-sisäluokka, jota adapter käyttää näkymän luomiseen. CategoryFragment on Listener-rajapinnan toteuttaja ja sen avulla CategoryViewHolder saa yhteyden fragmenttiin.

Esimerkkikoodissa 11 on esitelty CateogryViewAdapter-luokan onCreateViewHolder-metodi, jota RecyclerView kutsuu, kun sille syötetään dataa esitettäväksi. RecyclerView koostuu CategoryListItem-olioista, jotka voivat olla kategorioita tai listan viimeisenä oleva

yhteydenottopainike. Kun kyseessä on kategoria, luo ViewHolder CategoryViewHolder-olion, jonka toteutus on myös esitelty esimerkkikoodissa 11. CategoryViewHolder sisältää ohjeet lista-alkion luomiseen. Tässä tapauksessa lista-alkio koostuu yhdestä kuva- ja kahdesta tekstikomponentista. Luokan konstruktorissa yhdistetään luokan muuttujat annetun layouttiedoston komponentteihin. Bind-metodissa tekstikomponentteihin syötetään haluttu teksti ja kuvakomponenttiin asetetaan kuva Picasson avulla. Tässä luokassa asetetaan myös tarvittavat kuuntelijat lista-alkioon. Niitä varten on luotu Listener-rajapinta, jonka CategoryFragment toteuttaa. Kun käyttäjä painaa lista-alkiota, kutsuu siihen asetettu kuuntelija haluttua fragmentin metodia. OnCategoryItemClicked-metodi käynnistää tuoteluettelon SubCategoryFragmentin ja onCallToActionClicked-metodi käynnistää sovellukseen rakennetun yhteydenotto fragmentin.



Kuva 7. Kategoriasivulla on listaus tuotekategorioista ja listan lopuksi yhteydenottopainike, joka käynnistää uuden aktiviteetin.

Kun näkymä on alustettu, fragmentti luo itselleen presentterin ja ilmoittaa tälle olevansa valmis. CatalgoPresenterin toteutus on yksinkertainen. Kun näkymä on luotu se käynnistää latausnäkymän ja hakee tuotteet palvelimelta. Palvelin palauttaa listan kategori-

oita, joista luodaan CategoryListItemeita RecyclerViewiä varten. Listan viimeiseksi laitetaan yhteydenottopainike, joka on myös CategoryListItem. Tämän jälkeen poistutaan latausnäkymästä ja päivitetään lista getView().updateCategoryItems(categoryListItems)-metodilla. Lopputuloksena on kuvan 7 näkymä. Jos palvelin palauttaa virheen tai tyhjän listan, näytetään virheilmoitus.

4.4 Tuoteluettelo

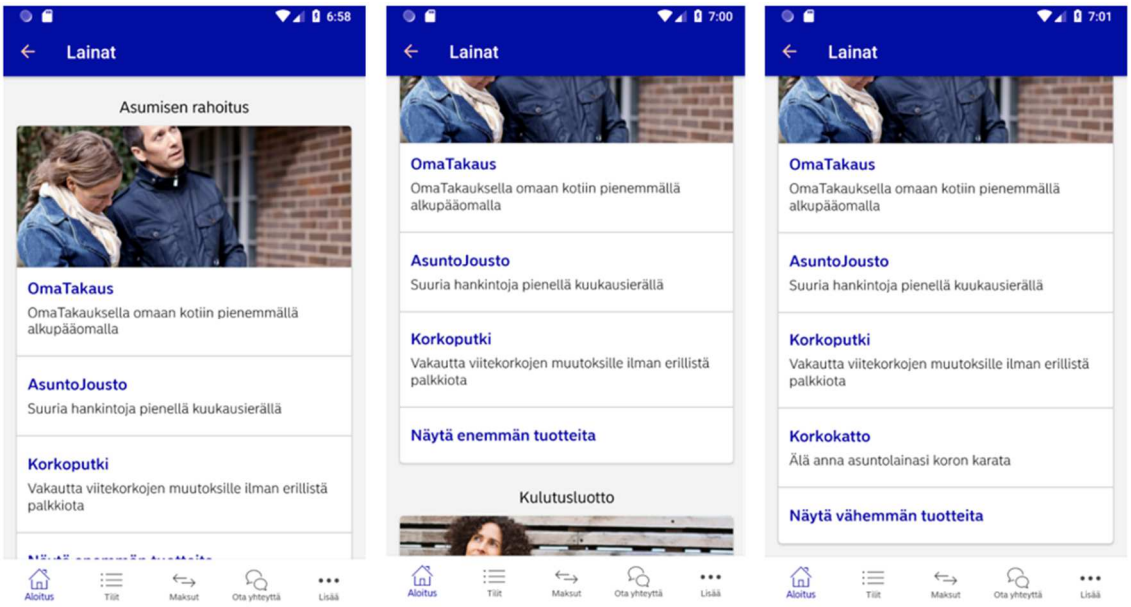
Tuoteluettelo käynnistetään SubCategoryFragmentin newInstance-metdoin avulla. Metodin parametrina annetaan valittu kategoriaolio. Fragmentin createPresenter-metodi välittää tämän olion SubCategoryPresenterille. Presenter luo kategorian tietojen perusteella avainarvo-parin, jossa avain on alakategorian nimi ja arvo on lista tämän alakategorian tuotteista. Avainta käytetään lista-alkiossa, joka näyttää alakategorian nimen TexView'ssä. Tämän alapuolelle listaan tulee CardView-komponentti, jonka sisällä on lueteltu kaikki tämän alakategorian tuotteet.

```
public void loadSubCategoryItems(@Nullable Category selectedCategory) {
    if (selectedCategory != null) {

        boolean moreProducts;
        List<SubCategory> subCategories = selectedCategory.getSubCategories();

        for (SubCategory sc : subCategories) {
            subCategoryListItems.add(new SubCategoryListTitle(
                    sc.getName() != null ? sc.getName() : ""));

            List<SubCategoryListItem> allItemsInCard = new ArrayList<>();
            List<Teaser> products = sc.getProductsInThisSubCategory();

            int listSize = products.size();

            allItemsInCard.add(new SubCategoryListMainProduct(
                    products.get(0)));
            for (int i = 1; i < listSize; i++) {
                allItemsInCard.add(new SubCategoryListProduct(
                        products.get(i)));
            }

            if (listSize > ITEMS) {
                SubCategoryListButton button = new SubCategoryListButton(
                        sc.getName());
                allItemsInCard.add(button);
                moreProducts = true;
            } else {
                moreProducts = false;
            }

            SubCategoryListCard card = new SubCategoryListCard(
                    sc.getId() != null ? sc.getId() : "",
                    allItemsInCard,
                    sc.getName(),
                    moreProducts);
            subCategoryProducts.put(sc.getName(), card);
            subCategoryListItems.add(card);
        }

        SubCategoryListItem contactItem = new SubCategoryContactItem(
                new CallToAction(CallToAction.Type.CONTACT_US);
        subCategoryListItems.add(contactItem);

        getView().updateSubCategoryItems(subCategoryListItems);
    } else {
        getView().showError();
    }
}
```

Esimerkkikoodi 12. SubCategoryPresenter-luokan loadSubCategory-metodi luo avain-arvoparit subCategoryProducts-muuttujaan ja kerää näytettävät lista-alkiot subCategoryListItems-listaan.

Esimerkkikoodi 12:ssa on esitelty avain-arvoparien ja subCategoryListItems-listan luonti. SubCategoryListItems-listan data näytetään fragmentissa. SubCategoryListCard-luokan card-olio sisältää listan kaikista alakategorian tuotteista. Se myös luo tarvittaessa itselleen toisen tuotelistan, jossa näkyy vain 3 ensimmäistä tuotetta. Kun käyttäjä painaa ”Näytä enemmän tuotteita” -painiketta, hakee presenter oikean card-olion avain-arvopari luettelosta alakategorianimen perusteella. Sen jälkeen lataa täyden listan näkymään

sekä päivittää painikkeen tilan, jolloin sen tekstiksi vaihtuu ”Näytä vähemmän tuotteista”. Kuvassa 8 esitelty näkymässä tapahtuva muutos.



Kuva 8. Tuoteluettelossa alakategoria otsikon alla on CardView, jossa ovat tähän kategoriaan kuuluvat tuotteet RecyclerView’n sisällä.

CardViewn sisältö oli toteutettavan omana RecyclerView-listana. Tämän vuoksi adapter-luokkaan tehtiin CardViewHolder-alaluokka, jonka bind-metodissa data sidottiin uuteen RecyclerView-listaan. Myös tällä listalla oli oma adapter-luokka. Kun CardView:n sisällä olevassa datassa tapahtuu muutoksia, on siitä erikseen kerrottava sitä hallinoivalle adapterille notifyItemsInserted- ja notifyItemRemoved-metodeilla.

4.5 Tuotesivu

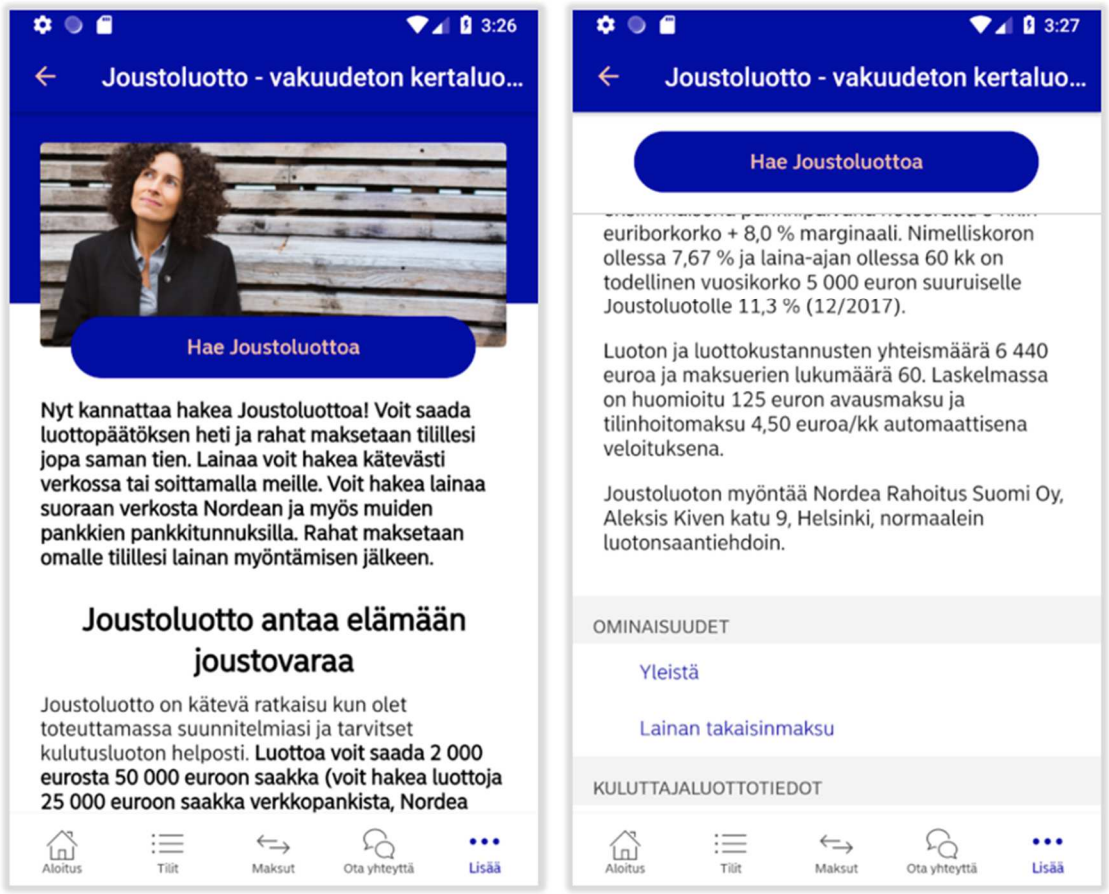
Tämän työn ehdottomasti mielenkiintoisin työvaihe oli tuotesivun työstäminen. Fragmentti käynnistetään jälleen newInstance-metodin avulla ja parametrina annetaan valitun tuotteen id-tunnus. Tämän avulla palvelimelta noudetaan tuotteen tiedot RxJavaa hyödyntäen ja luodaan näkymässä näytettävät oliot esimerkkikoodi 13:n mukaisesti

```
private void loadProductCardItems(@NonNull String productId) {
    getView().enableWaitingState();
    service.getProductItems(productId).subscribe(
          product -> {
              List<ProductCardItem> productCardItems =
                      createProductCardItems(product);
              ProductImageItem imageItem = new ProductCardItemFactory()
                      .createImageItem(getView(), product);
              getView().updateTitle(product.getDisplayName());
              getView().disableWaitingState();
              getView().updateImageItem(imageItem);
              getView().updateProductCardItems(productCardItems);
          },
          error -> {
              getView().disableWaitingState();
              getView().showError();
          }
    );
}
```

Esimerkkikoodi 13. ProductCardPresenterin loadPructCardItems-metodi, jossa palvelimelta haetuista olioista luodaan näkymän päivittämiseen paremmin soveltuvat oliot.

Aiemmasta poiketen tämä näkymä koostuu kahdesta eri osioista. Näkymän ylälaidassa on kuva, jonka päällä on CTA-painike. Painikkeen tarkoitus on johtaa tuotteen myyntiin ja siitä aukeaa tuotteen ostoputki. Ostoputki voi olla yhteydenotto pankkiin, jolloin avataan yhteydenottonäkymä. Se voi olla myös ulkoisilla sivuilla oleva verkkosovellus, jolloin haluttu verkko-osoite avataan selaimella. Selain käynnistetään uudessa aktiviteetissa Intentin ACTION_VIEW ja verkko-osoite parametrien avulla. Ostoputki voi olla myös sovellukseen rakennettu, kuten kuvan 9 esimerkissä. Hae Joustoluottoa-painike käynnistää lainahakemuksen. CTA-painikkeen tyyppi saadaan palvelimelta. Siinä olevan parametrin avulla voidaan tarkistaa, onko sovelluksessa olemassa omaa ostoputkea painikkeelle. Jos näin ei ole, voidaan tilalla käyttää yhteydenottopainiketta.

Näkymän alaosa muodostuu jälleen RecyclerView'stä, jonka ensimmäinen lista-alkio muodostuu WebView-komponentista. Tämän jälkeen tulee nippu TextView-komponentteja, jotka muodostavat ulkoisilla verkkosivuilla olevan välilehtirakenteen. Välilehdillä on otsikko, jonka jälkeen tulee alaotsikoita. Alaotsikoista voidaan avata uusi näkymä, josta kerrotaan luvussa 4.6.



Kuva 9. Tuotesivu ja kelluva CTA-painike.

Näkymä on rakennettu CoordinatorLayoutin sisälle, mikä mahdollistaa näkymän komponenttien käyttäytymisen kustomoinnin. Tässä tapauksessa sitä käytetään CTA-painikkeen kelluttamiseen. Painike on sidottu kuvassa 9 olevan tuotekuvan alalaitaan. Kuvan takana on piilossa valkoinen kotelo, johon painike asettuu kuvan poistuessa näkymästä. Kuva poistuu näkymästä, kun käyttäjä liikkuu siinä alaspäin. Kun käyttäjä liikkuu näkymässä ylös, kuva tulee jälleen esiin, ja painike ottaa oman paikkansa sen alareunassa.

Kun painike oli saatu kellumaan, oli aika keskittyä näkymässä seuravana olevaan elementtiin. Sen sisällä näytetään tuotteen perustiedot ja tärkeimmät myyntiargumentit. Teksti sisältää html-elementtejä, sillä se tulee suoraan ulkoisilta sivuilta. Tämän vuoksi se täytyy näyttää WebView'n sisällä. Sen sisällä olevan tekstin muotoilua varten luotiin css-tyylitiedosto. Tyylitiedosto liitettiin String-muuttujaan, jossa on html-rakenne. Sen body-elementtien väliin liitettiin palvelimelta saatu teksti. Tyylitiedosto on tallennettu raw-resurssikansioon. Tämän vuoksi WebView käyttää tätä kansiota juurenaan esimerkkikoodin 14 mukaisesti.

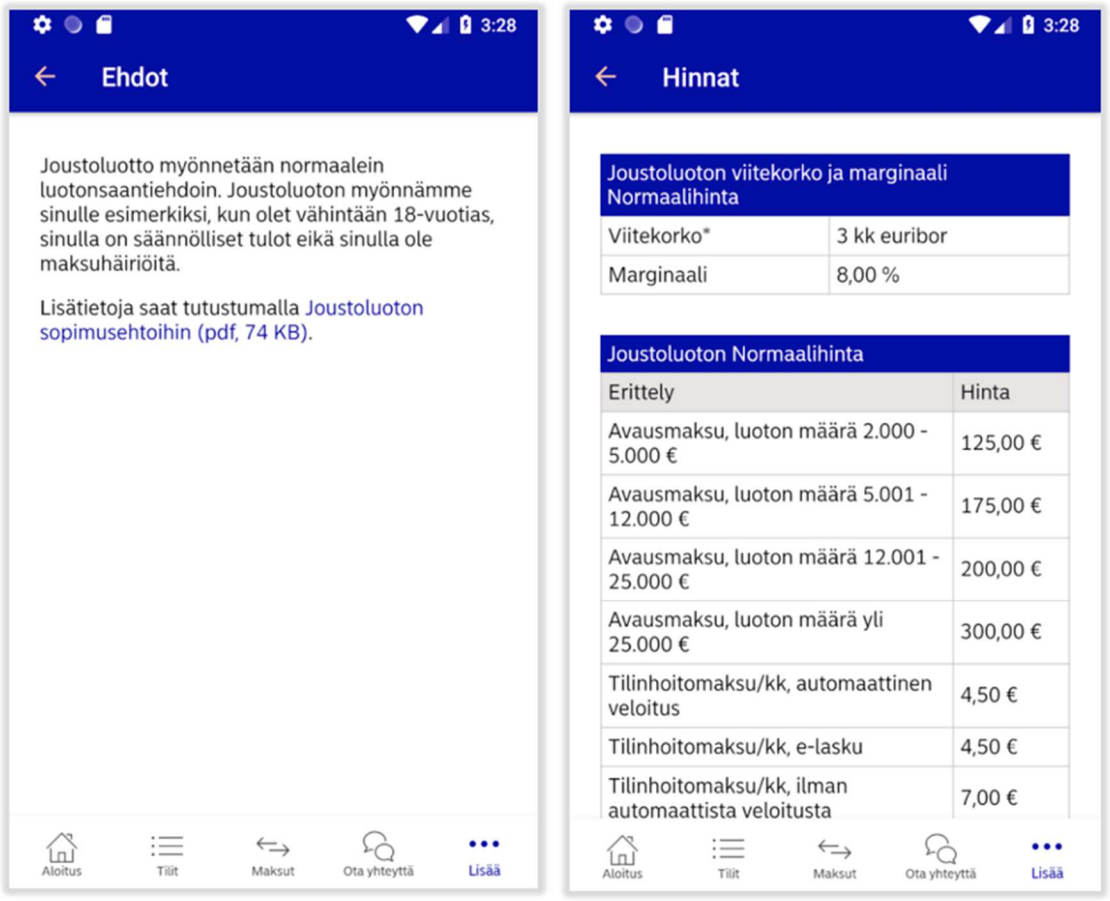
```
webView.loadDataWithBaseURL("file:///android_res/raw",
                            html, "text/html", "utf-8", null);
```

Esimerkkikoodi 14. WebView'n sisällön lataus loadDataWithBaseURL-metodin avulla mahdollistaa tyyli- ja JavaScript-tiedostojen käytön.

RecyclerView'n sisällä oleva WebView aiheuttaa ongelmia, kun käyttäjä palaa lisätietosivulta takaisin tuotesivulle. Jos WebView ei ole ollut näkyvissä, kun käyttäjä on siirtynyt seuraavaan näkymään, ei RecyclerView ole sitä vielä piirtänyt. Kun WebView näkymässä liikuttaessa tulee näkyviin, RecyclerView piirtää sen. WebView'n korkeus on tässä vaiheessa 0 pikseliä. WebView piirtää html-sisällön näkymään samalla kasvattaen omaa korkeuttaan. Käyttäjälle tämä näyttää siltä, että WebView lävähtää kokonaisen näytölle ja käyttöliittymä tuntuu nytkähtävän. Yritin kehittää tähän ongelmaan ratkaisun ottamalla WebView'n korkeuden talteen ja varaamalla sille tila listasta. Tämä ratkaisi ongelman ja käyttöliittymä toimi odotetusti. Tämä toteutus kuitenkin vaati, että WebView olisi latautunut kokonaan, ennen kuin käyttäjä poistuu edelliseen näkymään. Näkymä jäätyi, mikäli käyttäjä poistu siitä ennen kuin WebView oli valmis. Tämä teki toteutuksesta käyttökelvottoman. Tämä virhe huomattiin myöhäisessä vaiheessa, eikä sen kunnolliseen korjaukseen voinut varata aikaa. Tämän vuoksi päädyttiin palauttamaan vanha toteutus sellaisenaan.

4.6 Lisätietosivu

Lisätietosivu on tuotekatalogin yksinkertaisin näkymä. Se koostuu pelkästä WebView-komponentista. Se on niin yksinkertainen, ettei sitä varten ole tarvinnut tehdä omaa presentteriä, eikä sen tarvitse noudatta MVP-arkkitehtuuria. Fragmentille annetaan parametrina näytettävä sisältö html-muodossa. Sen onCreateView-metodissa luetaan sisältö Bundlesta ja asetetaan se esimerkkikoodi 14:sta tavoin html-rakenteen sisälle. Koska tässä näkymässä saattaa olla suuria taulukoita, on niistä tehtävä pieniin näyttöihin mukautuvia JavaScript-kirjastolla. JavaScript-kirjastot on liitettävä html-rakenteeseen ennen sen syöttämistä WebView-näkymään. Joillakin sivuilla saattaa myös olla linkkejä ja pdf-tiedostoja. Näitä varten on luotu oma WebViewClient-luokka, joka antaa WebView'lle ohjeet linkkien käsittelyyn.



Kuva 10. Kaksi esimerkkiä lisätietosivusta. Vasemmassa kuvakaappauksessa pdf-linkki ja oikeassa kuvassa on taulukko.

Kuvassa 10 on esitelty kaksi erilaista lisätietosivua. Lisätietosivujen sisältö vastaa ulkoisilta sivuilta löytyvän välilehden sisältöä. WebView'n ansiosta sivun sisältö voi olla mitä tahansa html-muotoista materiaalia.

5 Yhteenveto

Aloitin insinöörityön tekemisen tutustumalla Android-ohjelmointiin ja Nordean mobiilisovelluksen arkkitehtuuriin. Molemmissa oli todella paljon opittavaa. Olin tehnyt aikaisemmin yhden Android-sovelluksen harjoitusprojektina. Pääsin kuitenkin hyvin pian vauhtiin, koska Java-ohjelmointikieli oli minulle ennestään tuttu. Opin insinöörityön aikana todella paljon Android-sovelluksen kehittämisestä.

Uutta oli myös työskenteleminen näin suuressa projektissa. Aikaisemmat ohjelmointiprojektini olen tehnyt yksin tai hyvin pienessä ryhmässä. Nyt työskentelin yli sadan hengen

projektissa. Isolla kuvalla ei juurikaan ollut vaikutusta päivittäiseen työhöni. Suurin ulkopuolelta tullut muutos oli tuotekatalogin ilmeen uusiminen. Se tuli kuitenkin omalta kannaltani täydelliseen aikaan, sillä pääsin tekemään toteutuksen puhtaalta pöydältä.

Insinöörityön suurimmat haasteet olivat tuoteluettelon sisäkkäiset RecyclerView-listat ja tuotesivun kelluva painike. Molemmat sain toimimaan määritysten mukaisesti. Erityisen ylpeä olen tuotesivun kelluvasta painikkeesta, jota pidän työni hienoimpana yksityiskohtana. Toisaalta tuoteluettelossa tapahtuvat muutokset sisäkkäisissä listoissa vaatisivat hieman lisää hiomista animaatioiden osalta. Myös tuotesivun WebView-komponentti vaatii mielestäni vielä lisää työstämistä.

Insinöörityöni tavoite oli kehittää tuotekatalogi Nordean mobiilisovellukseen. Tämä tavoite saavutettiin määritysten mukaisesti ja tuotekatalogi on saatavilla Nordean mobiilisovelluksen versiossa 3.0 alkaen. Olen lopputulokseen todella tyytyväinen lopputulokseen ja näen tuotekatalogissa paljon mahdollisuuksia myynnin ja asiakaspalvelun kannalta. Tulevaisuudessa yhä suurempi osa tuotteista voidaan avata mobiilisovelluksessa asiakaspalvelun sijaan. Tuotekatalogi tarjoaa tähän valmiin väylän. Tuotteelle voidaan tehdä oma verkkosovellus, joka aukeaa tuotesivulla olevasta CTA:sta. Näin sovellukseen voidaan lisätä ostoputkia, ilman tarvetta sovelluksen päivittämiselle.

Tuotesivulla oleva CTA on tällä hetkellä mielestäni liian kaukana sovelluksen aloitusnäkymästä. Tuotekatalogi on hieman piilossa ja sitä on vaikea löytää. Myös kategoriarakenteen pitäisi olla mielestäni yksinkertaisempi ja asiakkaalle relevantteja tuotteita tulisi korostaa. Uskon, että näin saisimme suuremman käyttöasteen tuotekatalogille ja tätä kautta asiakastyytyväisyys tuotteiden myynti lisääntyisi.

Lähteet

1 Android (operating system). Verkkodokumentti. <https://en.wikipedia.org/wiki/Android_(operating_system)> Luettu 19.3.2018.

2 Android founder: We aimed to make a camera OS. Verkkodokumentti <https://www.pcworld.com/article/2034723/android-founder-we-aimed-to-make-a-camera-os.html> Luettu 19.3.2018.

3 Android Pre-History. Verkkodokumentti. <https://www.androidcentral.com/android-pre-history> Luettu 19.3.2018.

4 Industry Leaders Announce Open Platform for Mobile Devices. Verkkodokumentti. <http://www.openhandsetalliance.com/press_110507.html> Luettu 21.3.2018.

5 Google's iron grip on Android: Controlling open source by any means necessary. Verkkodokumentti. <https://arstechnica.com/gadgets/2013/10/googles-iron-grip-on-android-controlling-open-source-by-any-means-necessary/> Luettu 21.3.2018.

6 Android execs get technical talking updates, Project Treble, Linux and more. Verkkodokumentti. <https://arstechnica.com/gadgets/2013/10/googles-iron-grip-on-android-controlling-open-source-by-any-means-necessary/> Luettu 21.3.2018.

7 Supporting Different Platform Versions. Verkkodokumentti <https://developer.android.com/training/basics/supporting-devices/platforms.html> Luettu 20.4.2018.

8 Dashboards. Verkkodokumentti. <https://developer.android.com/about/dashboards/index.html> Luettu 20.4.2018.

9 Support Library. Verkkodokumentti<https://developer.android.com/topic/libraries/support-library/index.html> Luettu 20.4.2018.

10 Platform Architecture. Verkkodokumentti. <https://developer.android.com/guide/platform/index.html> Luettu 22.3.2018.

11 System and kernel security. Verkkodokumentti. <https://source.android.com/security/overview/kernel-security> Luettu 22.3.2018.

12 Android users rejoice! Linux kernel LTS releases are now good for 6 years. Verkkodokumentti. <https://arstechnica.com/gadgets/2017/09/android-users-rejoice-linux-kernel-lts-releases-are-now-good-for-6-years/> Luettu 22.3.2018.

13 Hardware Abstraction Layer (HAL). Verkkodokumentti. <https://source.android.com/devices/architecture/hal> Luettu 22.3.2018.

14 A Closer Look at Android RunTime (ART) in Android L. Verkkodokumentti. <https://www.anandtech.com/show/8231/a-closer-look-at-android-runtime-art-in-android-l/> Luettu 22.3.2018.

15 Application Fundamentals. Verkkodokumentti. <https://developer.android.com/guide/components/fundamentals.html> Luettu 27.3.2018.

16 App Manifest Overview. Verkkodokumentti. <https://developer.android.com/guide/topics/manifest/manifest-intro.html> Luettu 27.3.2018.

17 Introduction to Activities. Verkkodokumentti. <https://developer.android.com/guide/components/activities/intro-activities.html> Luettu 27.3.2018.

18 The Activity Lifecycle. Verkkodokumentti. <https://developer.android.com/guide/components/activities/activity-lifecycle.html> Luettu 27.3.2018.

19 Intent. Verkkodokumentti. <https://developer.android.com/reference/android/content/Intent.html> Luettu 27.3.2018.

20 Intents and Intent Filters. Verkkodokumentti. <https://developer.android.com/guide/components/intents-filters.html> Luettu 27.3.2018.

21 Parcel. Verkkodokumentti. <https://developer.android.com/reference/android/os/Parcel.html> Luettu 17.4.2018.

22 Parcelables and Bundles. Verkkodokumentti. <https://developer.android.com/guide/components/activities/parcelables-and-bundles.html> Luettu 17.4.2018.

23 Fragments. Verkkodokumentti. <https://developer.android.com/guide/components/fragments.html> Luettu 3.4.2018.

24 View. Verkkodokumentti. <https://developer.android.com/reference/android/view/View.html> Luettu 3.4.218.

25 Create a List with RecyclerView. Verkkodokumentti. <https://developer.android.com/guide/topics/ui/layout/recyclerview.html> Luettu 3.4.2018.

26 App Resources. Verkkodokumentti. <https://developer.android.com/guide/topics/resources/index.html> Luettu 15.4.2018.

27 Localizing with Resources. Verkkodokumentti. <https://developer.android.com/guide/topics/resources/localization.html> Luettu 15.4.2018.

28 Picasso a powerful image downloading and caching library fo Android. Verkkodokumentti. <http://square.github.io/picasso/> Luettu 18.3.2018.

29 Gson User Guide. Verkkodokumentti. <https://github.com/google/gson/blob/master/UserGuide.md> Luettu 4.4.2018.

30 How To Use RxJava. Verkkodokumentti. < https://github.com/ReactiveX/RxJava/wiki/How-To-Use-RxJava> Luettu 6.4.2018.

31 Model-view-presenter. Verkkodokumentti. < https://en.wikipedia.org/wiki/Model–view–presenter> Luettu 20.4.2018.

32 Mosby Model-View-Presenter. Verkkodokumentti. < http://hannesdorfmann.com/mosby/mvp/> Luettu 7.4.2018.

33 Mosby getting started. Verkkodokumentti. < http://hannesdorfmann.com/mosby/getting-started/> Luettu 10.3.2018.