LAPIN AMK
Lapland University of Applied Sciences

ANDROID-ARKKITEHTUURIN JA KARTTAOHJELMAN KEHITYKSEN HAASTEET

Marjasankko-projekti

Mikko Pajula

Opinnäytetyö
Arktiset luonnonvarat ja talous
Tieto- ja viestintätekniikka
Insinööri (AMK)

2019

LAPIN AMK
Lapland University of Applied Sciences

Opinnäytetyön tiivistelmä

Arktiset luonnonvarat ja talous
Tieto- ja viestintätekniikka
Insinööri (AMK)

Tekijä	Mikko Pajula	Vuosi	2019
Ohjaaja(t)	Tauno Tepsa		
Toimeksiantaja	Luonnonvarakeskus		
Työn nimi	Android-arkkitehtuurin ja karttaohjelman kehityksen haasteet		
Sivu- ja liitesivumäärä	58		

Opinnäytetyössä käsitellään Android-mobiiliohjelmaprojektin toteuttamista käyttöliittymän suunnittelusta arkkitehtuurin toteuttamiseen ja yksittäisten käytännön toteutusten ratkaisuihin. Käyttöliittymäsuunnittelun sekä ohjelmistoarkkitehtuurin perusteet käydään läpi. Käyttöliittymän periaatteita pyritään soveltamaan käytäntöön esimerkkien kautta.

Ohjelmistoarkkitehtuurin suunnittelussa kokeillaan olemassa olevien arkkitehtuurimallien soveltamista Android-ympäristössä ja pohditaan ratkaisutapojen toimivuutta. Yksittäisten ominaisuuksien käytäntöjen osalta perustellaan ja suositellaan ratkaisutapoja.

Taustaprojekti ”Marjasankko” toimii käytännön toteutuksen rajapintana, johon suunnitellut ohjelmistoarkkitehtoniset ratkaisut pyritään toteuttamaan. Ohjelman paikkatietopohjaisuus asettaa omat haasteensa mobiiliohjelmalle ja sen arkkitehtuurille sekä käyttöliittymälle. Opinnäytetyössä painotetaan paikkatieto-ominaisuuksien ratkaisuihin ja pohditaan paikkatiedon vaikutusta arkkitehtuuriin sekä käyttöliittymään.

Opinnäytetyön myötä projektiin toteutettiin käyttöliittymä, joka pyrki noudattamaan käyttöliittymäsuunnittelun periaatteita. Yksittäisiä ratkaisuja tullaan käyttämään projektissa. Erityisesti muistivuotojen välttämisen osalta tarvittavat tekniikat ovat jatkuvasti osa ohjelmointityötä. Ohjelmistoarkkitehtuurin osalta projektille löytyi suunta ja komponentit, joilla tullaan etenemään.

Avainsanat Android, ohjelmistoarkkitehtuuri, paikkatietojärjestelmät

LAPIN AMK
Lapland University of Applied Sciences

Abstract of Thesis

Arctic Natural Resources and Economy
Degree Programme in Information and Communication Technology
Bachelor of Engineering

Author Mikko Pajula Year 2019
Supervisor Tauno Tepsa
Commissioned by Natural Resources Institute Finland
Subject of thesis Challenges of Android Architecture and Map Application Development
Number of pages 58

The thesis addressed implementation of an Android mobile software project from the user interface to the software architecture and individual practical solutions to specific tasks. The thesis discussed the basics of user interface construction and software architecture. Principles of the user interface construction were applied using examples.

The existing software architecture principles were applied to the Android environment and the functionality of principles was considered. in case of the individual software properties, solutions were explained and recommended.

The background project “Marjasankko” served as an interface, where principles and the solutions were partly implemented. The project’s nature as a map application with geographic information system base, set its own challenges for software architecture and user interface. Thesis focused on solutions faced in these areas, when implementing a map application.

Alongside with the thesis, user interface was implemented in the background project. The project’s user interface intended to follow principles of user interface design introduced in the thesis. individual software properties were implemented to the process of project development. Particularly the techniques of avoiding memory leaks were implemented. In the field of software architecture, general direction and architecture components were found for the project.

Key words Android, Software architecture, Geographic information systems

SISÄLLYS

ALKUSANAT

Kiitos Lapin Ammattikorkeakoulun pLAB ohjelmistotekniikan laboratorion henkilökunnalle moniulotteisesta tuesta. Ikimuistoiset debug-prosessit testilaitteiden täyttäessä taskuja Lapin luonnon keskellä ja kaupunkiretket sijaintipalvelua testatessa lämmittävät vieläkin mieltä.

Kiitos myös Android-alustalle masokistisesta nautinnosta, jota on kehittäminen tälle alustalle. Muutosvauhti on melkoinen, mutta kaiketi parempaan suuntaan. Kiitos myös SIGSEGV, tiedät, mitä teit.

KÄYTETYT MERKIT JA LYHENTEET

FLUX	Ohjelmistoarkkitehtuuriperiaate, jossa pyritään korjaamaan MVC:n puutteita.
GNSS	Global Navigation Satellite System. Satelliittipaikannusjärjestelmä
Mockup	Käyttöliittymäsuunnitelmasta tehty ohjelma, jossa on ainoastaan graafinen käyttöliittymä ilman ohjelman varsinaista toimintalogiikkaa.
MVC	Model View Controller. Ohjelmistoarkkitehtuuriperiaate, jossa erotetaan toisistaan käyttöliittymä toiminnallisuus sekä tiedon käsittely.

1 JOHDANTO

Ohjelmistoprojektissa mobiiliympäristö toteutusalustana tuo oman haasteensa toteutukselle. Kun tähän lisätään vielä paikkatieto, on toteutus vielä epävarmempi. Kysymyksiä, jotka tulevat esiin projektia aloitettaessa mobiiliympäristössä, on useita. Mitä ratkaisuita tulisi käyttää? Kuinka alustan valmiit ratkaisut toimivat käytännössä? Kuinka alusta rajoittaa perinteisten arkkitehtuurien toteuttamista? Miten projekti on syytä aloittaa?

Mobiilialustan tuomat haasteet ovat tyypillisesti ohjelmakomponenttien elinsyklien (lifecycle) aiheuttaman katkokset ja jatkuvan, myös taustalla tapahtuvan, toiminnan tasapainotus (Android Developers 2018i). Android-käyttöjärjestelmän tapauksessa alustan muuttuminen ajan myötä (Android Developers 2018b) ja laaja laitepohja erilaisine muisti- sekä anturikapasiteetteineen (Android Developers 2018g).

Käyttöliittymän suunnitteluprosessin roolia ei pidä unohtaa, jotta projekti etenisi ilman jatkuvaa uudelleenohjelmointia. Hyvin suunniteltu käyttöliittymä vapauttaa keskittymään muiden teknisten ongelmien ratkaisuihin. Hyvin suunniteltu käyttöliittymämalli löytää ominaisuudet, jotka pitää toteuttaa, jotta ohjelma toimii asiakkaan toivomalla tavalla.

Jatkuva ja pitkäaikainen paikkatiedon kerääminen GNSS-mittauksella on vastoin mobiilialustan vaatimaa virrankäyttöpolitiikkaa (Android Developers 2018b). Alustakohtaiset käyttöliittymäelementit kartassa, kuten reitin piirto tai karttakuvakkeet voivat asettaa rajoituksia esimerkiksi esitettävän tiedon määrään. Mobiilialustalla on oltava tarkkana muistin käytön suhteen (Android Developers 2018b). Valmiit järjestelmät alustassa voivat keventää toteuttamisen määrää, mutta samalla asetetaan mahdollisesti olennainen toiminto kokonaan toisen osapuolen haasteeksi ja lopputulos ei välttämättä vastaa tavoitetta.

Opinnäytetyössä kuvataan haasteiden aiheuttamat suunnitteluvaiheen ratkaisut sekä pohditaan mahdollista toteutustapaa. Arkkitehtoniset rakenteet, käyttöliittymäsuunnittelun periaatteet ja yksittäiset ratkaisut pyritään soveltamaan projektiin ”Marjasankko”. Hankkeen suunnittelu antaa käytännön esimerkin teoriapohjalle.

Käyttöliittymäsuunnittelun periaatteita pyritään soveltamaan suoraan taustaprojektiin, mutta arkkitehtuurin periaatteita kokeillaan ensin erilliseen taustaprojektista irralliseen ohjelmaan. Ohjelman tarkoitus on testata periaatteiden toimivuutta Android-ympäristössä. Samalla ohjelmaan voidaan suunnitella ja ratkaista myös Android-alustalle tapauskohtaisia arkkitehtonisia ratkaisuja, kuten yhden Activity-luokan Fragment-navigaatioarkkitehtuuri.

2 PERIAATTEET

2.1 Käyttöliittymä

2.1.1 Käytettävyys

Ohjelman käyttöliittymän käytettävyyden teoreettiseksi pohjaksi voi alustasta riippumatta valita *heuristisen evaluaation* (Soini 2016, 9). Käytännössä tämä tarkoittaa käytettävyyden arviointia hyväksi havaittujen nyrkkisääntöjen kautta. Heuristinen-sanan merkitys on kuvattuna alla sanakirjasta.

> arvaileva, epäjärjestelmällinen (päättely tms.); keksimiseen perustuva tai johtava (olettamus tms.); nyrkkisääntöjä käyttävä, täydellisyyteen pyrkimätön. Alkujaan sana viittaa keksimiseen, oivaltamiseen ja kokeilemiseen. Nykyisin sitä usein käytetään vain epämääräisesti järjestelmällisyyden vastakohtaa ilmaisevana. Toisaalta se voi tarkoittaa myös sellaisten nyrkkisääntöjen tai kokeilujen soveltamista, joista tiedetään, etteivät ne anna parasta mahdollisuutta vaan (toivottavasti) "riittävän hyvän" ratkaisun. (Suomisanakirja 2018)

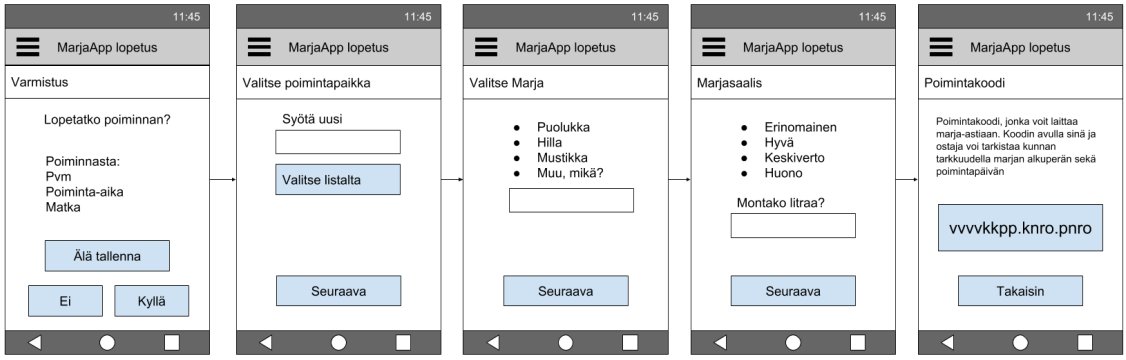
Sanakirjan kuvaus on sikäli hyvä, että siitä välittyy koettu heuristisen evaluaatioprosessin käytännönläheisyys ja iteratiivisuus. Käytettävyyttä suunnitellessa on hyvä käyttää maalaisjärkeä. Heuristiset säännöt antavat hyvän pohjan käytettävyyden käytännön toteutukselle. Jakob Nielsenin kymmenen heuristista sääntöä ovat yksiä tunnetuimmista. (Soini 2016, 9)

1. **Näkyvyys**: Ohjelman tulisi pitää käyttäjä perillä sen tilasta sopivalla palautteella sopivan ajan sisällä.
2. **Yhteensopivuus systeemin ja todellisen maailman välillä**: Ohjelman tulisi ilmaista asiat käyttäjän kielellä, ei järjestelmän kielellä. Ohjelman tulisi myös seurata todellisen maailman käytäntöjä ja välittää informaatiota luonnollisessa järjestyksessä.
3. **Hallitsevuuden ja vapauden tunne käyttäjällä**: Käyttäjällä olisi hyvä olla selkeästi esillä mahdollisuus alkutilaan palaamiseen ja siten poistua epätoivotusta tilasta. Myös helppo paluu edeltävään tilaan olisi hyvä olla.

4. **Jatkuvuus ja standardit**: Käyttäjän ei pitäisi joutua pohtimaan, tarkoittavako samat tilat tai sanat samaa kuin muissa ohjelmissa. Alustakohtaisia käytäntöjä on hyvä noudattaa.
5. **Virheiden ehkäisy**: Hyvää virheilmoitusta parempi on ehkäistä virheet alun perinkin. Käyttöliittymän kohdalla ohjelman voi toteuttaa niin, että virhetilanteisiin joutuu mahdollisimman vähän.
6. **Muistikuormituksen minimoiminen**: Minimoi käyttäjälle tarve muistaa, kuinka käyttöliittymä toimii. Käyttöliittymän toiminnan olisi hyvä olla niin intuitiivinen ja tunnistettava, että muistamista tarvitaan mahdollisimman vähän.
7. **Käytön tehokkuus ja joustavuus**: Käyttöliittymän tulisi tukea tehokäyttäjää, kuitenkin hankaloittamatta aloittelevan käyttäjän käyttökokemusta. Käyttöliittymä voisi täten joustaa ja sopeutua eri käyttäjäryhmiin.
8. **Minimalistinen suunnittelu**: Näkymän ei tulisi näyttää ylimääräistä, joka vie huomion olennaisesta, vaan minimoida sisältö.
9. **Virheistä toipuminen**: Virheviestit tulisi ilmaista käyttäjän ymmärtämällä kielellä, selkeästi ilmaista ongelma ja tarjota ratkaisua.
10. **Ohjeet**: Vaikka käyttöliittymä, joka ei tarvitse ohjetta olisi parempi, on joskus tarpeellista tarjota kattava ohjeistus. Ohjeet tulisi olla helposti saatavilla ja vastata käyttäjän tavoitteisiin sekä tarjota helpot askeleet tavoitteeseen pääsystä. (Nielsen 1994)

2.1.2 Rautalankamallintaminen

Rautalankamallintaminen (Wireframing) tarkoittaa vaihetta, jossa käyttöliittymän näkymistä piirretään ilman estetiikkaa käytettävyysmalli (Kuvio 1). Kun ohjelman tavoiteltavat toiminnot ovat selvillä, voidaan rautalankamallilla suunnitella käytettävyys. Tarkoitus on ei ole suunnitella prototyyppiä, vaan ratkaista käyttökokemuksen ongelmat. Rautalankamallilla suunnitellaan käyttöliittymä niin, että käyttäjät voivat tehdä sillä toimivasti haluamansa asiat. (Barnard 2016)



Kuvio 1. Esimerkki rautalankamallintamisesta viidellä näkymällä ja navigaationuolilla

Rautalankamallisuunnittelulla voidaan hahmottelemalla suunnitella käyttöliittymä vastaamaan ohjelman tavoitteisiin. Tässä vaiheessa ei oteta kantaa ulkonäköön tai ohjelmointiin. (Wood 2014, 54)

2.1.3 Visuaaliset periaatteet

Hyvä lähtökohta käyttöliittymän ulkonäölle on minimalismi, jossa yksinkertaisuudella tehdään kaunista sekä selkeää. Minimalismi terminä kattaa useita merkityksiä ja taiteen aloja, kuten musiikki tai kirjallisuus. Kaikilla näillä aloilla, kuten myös käyttöliittymän suunnittelussa, minimalismin ydinmerkitys on merkityksellisyys ja yksinkertaisuus. Minimalismia käytettäessä tulee toteutettua heuristisiten sääntöjen kahdeksas kohta (Ahtinen & Kokkonen 2000). (Yalanska 2016)

Käyttöliittymissä minimalistisen suunnittelun onnistuneissa esimerkeissä on Googlen etusivu (*http://www.google.fi*), jossa koko käyttöliittymä on suunniteltu selkeän ja yksinkertaisen hakukentän ympärille. Liialliset häiriötekijät poistamalla on kannustettu käyttäjää osallistumaan. Periaate voidaan tiivistää sanontaan ”vähemmän on enemmän”. (Smith 2016)

Minimalistisuuden luonteenomaiset piirteet helpottavat sotkuisen käyttöliittymän välttämistä. Alla on listattu tyypillisiä minimalismin periaatteita. (Yalanska 2016)

- yksinkertaisuus
- selkeys
- ilmaiseva visuaalinen hierarkia

- huomio suhteissa ja asettelussa
- jokaisella elementillä merkitys
- paljon jätettyä vapaata tilaa
- paljon huomiota ydinelementtien yksityiskohtiin
- typografia merkittävänä osana
- merkityksettömien koristelullisten elementtien poistaminen (Yalanska 2016)

2.1.4 Hahmolait

Gestalt, joka on Saksaa, tarkoittaa Suomeksi "hahmoa". Gestalt-teorian tausta on psykologiassa. Se pyrkii selittämään, miten hahmotamme kokonaisuuksia elementeistä. Käyttöliittymäsuunnittelussa on hyvä ottaa huomioon Gestalt-lait. Erityisesti minimalistisessa suunnittelussa, esimerkiksi ryhmittäminen korostuu käyttökokemuksen parantajana. (Laine 2004)

Gestalt-lait kuuluvat seuraavasti:

1. **Samanlaisuus**: Muodoiltaan, kooltaan tai väreiltään samankaltaiset kuviot mielletään yhteensopiviksi.
2. **Läheisyys**: Lähekkäin olevat kuviot mielletään yhteenkuuluviksi. Yhteenkuuluvuutta voi korostaa kiinnittämällä kuviot toisiinsa tai vielä voimistaa asettamalla kuviot limittäin.
3. **Valiomuotoisuus**: Kuviot ymmärretään yksinkertaisempana ja symmetrisempänä kuin ne mahdollisesti ovat. Toisin sanoen kuvioiden kokonaisuus voi muodostaa tarkoituksella tai vahingossa oman kokonaiskuvionsa.
4. **Symmetria**: Mitä symmetrisempi kokonaisuus, sitä enemmän katsoja kiinnittää huomiota kokonaisuuteen, keskittyen vähemmän kuvioihin, jotka muodostavat kokonaisuuden.
5. **Yhteinen liike**: Samaan suuntaan liikkuvat kuviot mielletään samaan ryhmään kuuluviksi.

6. **Yhteenliittyminen**: Toisissaan kiinni olevat kuviot liittyvät toisiinsa. Erona läheisyyteen tällä tarkoitetaan esimerkiksi tapausta, missä kaksi kuviota ovat kiinnitetty toisiinsa viivalla. Tyypillinen esimerkki on välilehti kiinnitettynä taustavärillä sen otsikkoon välilehtivalikossa.
7. **Sulkeutuvuus**: Suljettu tai lähes suljettu viiva muodostaa kuvion. ihminen hahmottaa kokonaisuuden omana kuvionaan, vaikka kokonaisuus ei ole täysin eheä.
8. **Alue**: Laajempi pienempää aluetta ympäröivä alue mielletään taustaksi ja pienempi alue kohteeksi. Alueella voi ryhmittää ja muuntaa alueen kaltaisia muotoja kohteiksi. (Laine 2004)

2.1.5 Mobiilialustan rajoitteet

Perustavanlaatuinen käyttöliittymäsuunnittelun mobiilikohtainen lisähaaste on näytön koko. Muuten erot periaatteissa eivät ole suuria. Mobiiliympäristössä kaikki on kiinni valinnoista. On asioita, mitä täytyy tehdä ja mitä ei pysty tekemään. (Krug 2014, 145)

Tavoitteesta että haluttu informaatio olisi mahdollisimman vähäisten painallusten määrän takana tulee joustaa. Näyttötilan puute johtaa siihen, että tarvitaan enemmän painalluksia, kun tieto on jaettu eri näkymiin pienelle näytölle. Tämä ei ole huono asia, koska se tekee pienen näytön käyttökokemuksesta toimivan. Asian voi ajatella niin, että sormen painallukset korvaavat isommalla näytöllä hiiren liikuttelun ja rullaamisen. Olennaista on muistaa loogiset polut painalluksesta toiseen edetessä. (Krug 2014, 148)

2.1.6 Esteettömyys

Ohjelman esteettömyyden parantaminen lisää sen käytettävyyttä laajemmalle käyttäjäkunnalle. Muun muassa heikkonäköisille, sokeille, kuuroille, heikkokuuloisille, kognitiivisesti haasteellisille ja pysyvästi tai tilapäisesi vammautuneille. Esteettömyysominaisuuksista alustassa esimerkkinä on näyttölukija, joka lukee ääneen tekstisisällöt ohjelmasta. Tässä tapauksessa on hyvä antaa mediasisällölle tekstikuvaus ohjelmassa. (Material design 2019)

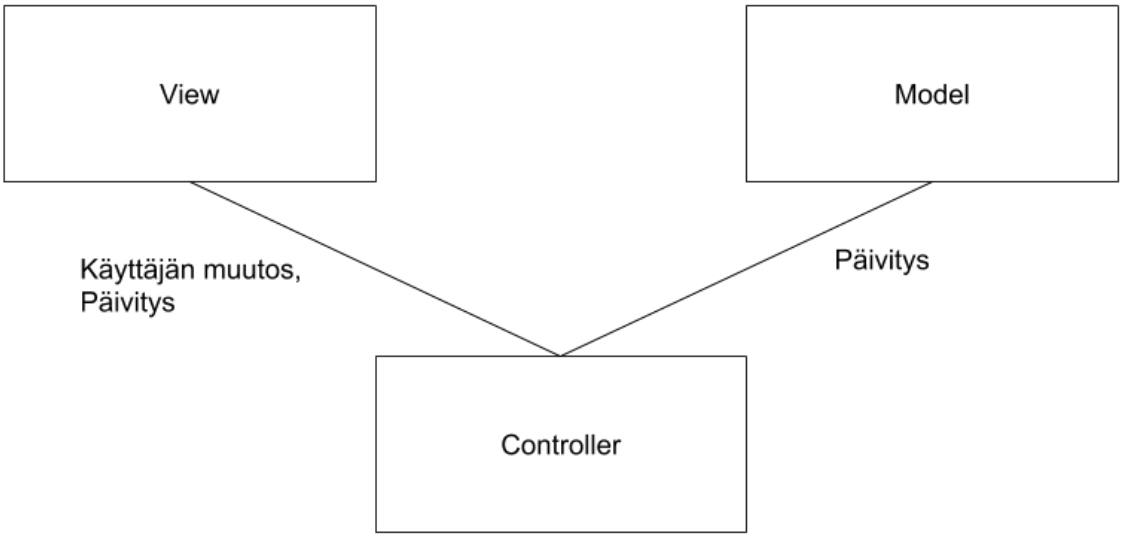
Esteettömyyttä voidaan parantaa lukuisin tavoin. Alla lista huomioitavista asioista.

- Selkeä sisällön asettelu allekkain helpottaa näyttölukijan käyttöä.
- Tekstisisällön tulee olla mahdollisimman luettavaa. Luettavuus parannetaan tekstin osalta fontin valinnalla, kirjainten koolla ja tekstin värillä. Korkea kontrasti tekstin ja taustan välillä parantaa luettavuutta.
- Värisokeiden kohdalla on hyvä esimerkiksi erottaa kyllä ja ei -painikkeet toisistaan punaisen sekä vihreän värin lisäksi ikonilla painikkeissa.
- Tekstiselitykset mediasisältöön, kuten videoihin sekä kuviin näyttölukijaa varten.
- Painikkeet selkeiksi ja riittävän isoiksi, jotta ne olisivat mahdollisimman helposti painettavissa. (Egorina 2017)

2.2 Arkkitehtuuri

2.2.1 Model View Controller

MVC-arkkitehtuurin rakenne on yksinkertainen ja periaate selkeä. Arkkitehtuurin tavoite on eritellä käyttöliittymä ja tietorakenteet ohjelmallisesti toisistaan. erottelu mahdollistaa selkeämmän ja helpomman koodin ylläpidon. MVC:ssa tehdään käyttöliittymä ja tietorakenne erikseen. Ne sidotaan yhteen controller-tasolla, joka ottaa vastaan esimerkiksi tietyn painalluksen signaalin ja ohjaa Model-tason toimimaan halutusti. Controller-taso ottaa myös vastaan tietoa Model-tasolta ja ohjaa käyttöliittymää näyttämään saadun tiedon (Kuvio 2). (Muntenescu 2016)



Kuvio 2. MVC-rakenne (Kononenko 2016)

Model-taso vastaa prosessoinnista ja datanhallinnasta. Model-tasolla hoidetaan tyypillisesti rajapintakutsut ja tietokantayhteydet. Android-alustalla Model on mikä tahansa ohjelmasta riippuvainen datanhallinnan komponentti. Android-alustalla Model voi siis olla pelkkä luokka, jonka luokkamuuttujiin tiedot tallennetaan. (Muntenescu 2016)

View-taso on käyttöliittymätaso. View määrittää käyttöliittymän rakenteen ja ulkonäön. Androidissa tällä tarkoitetaan lähteestä riippuen käyttöliittymän staattisia xml layout -tiedostoja, Fragment- tai Activityluokkia tai erillistä itse tehtyä View-luokkaa, jonne eritellään käyttöliittymän toiminta. Viimeisessä tapauksessa tyypillisesti määritellään Activity ja Fragment controlleriksi (Space-O Technologies 2019). (Muntenescu 2016)

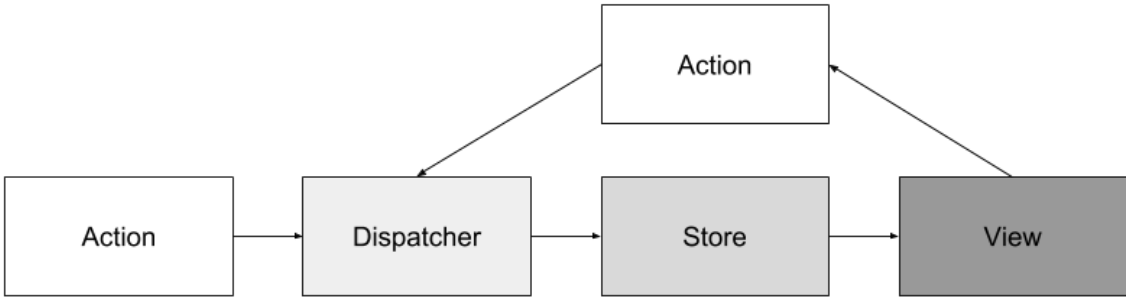
Controller-taso on MVC:ssa logiikkataso, joka välittää käyttöliittymästä käyttäjän komennot Model-tasolle ja takaisin. Controller toimii siis sidosluokkana, joka ainoana sisältää viitteet sekä Model- että Viewtasoille. (Muntenescu 2016)

2.2.2 FLUX

FLUX on arkkitehtuurimalli, jonka Facebook® kehitti käytettäväksi omassa React-kehitysalustassa. React on tarkoitettu verkkosivujen tekemiseen. Flux kehitettin vastaamaan MVC-arkkitehtuurin ongelmiin. Ydinhaaste MVC-arkkitehtuurissa on edestakaisin kulkeva tiedonvälitys luokkien välillä, jota ei ole

kuvauksessa huomioitu. View-luokan toiminta välitetään Controllerissa tarpeellisille Model-luokille ja Model-luokat välittävät controllerille muutokset, joka välittää niitä View-luokille (Kuvio 2). Ohjelman kasvaessa, näiden risteävien tietovirtojen hallinta ja hahmottaminen käyvän hetki hetkellä hankalammaksi. (Clark 2015)

FLUX-arkkitehtuurin kantava ajatus on kehittää MVC-arkkitehtuuria lisäämällä siihen yksisuuntainen tiedonvälitys. Ajatus on, että käyttöliittymänäkymä (View) totetuttaa toiminnon (Action), joka lähetetään keskuslähettimöön (Dispatcher). Lähettimö ohjaa toiminnon oikeille varastoille (Store), jotka tekevät oman tehtävänsä, eli tiedonhallinnan. Varastot, saatuaan päätökseen tehtävänsä, lähettävät muutokset kuhunkin sen tietoa esittävään näkymään (View) (Kuvio 3). (Facebook 2019)



Kuvio 3. Flux arkkitehtuuri (Facebook 2019)

View-luokka FLUX -arkkitehtuurissa on toiminnallisessa kuvauksessaan hieman laajempi kuin MVC -Arkkitehtuurissa. Luokkaa voi kutsua myös Controller-View -luokaksi. Luokka hallitsee muutokset käyttöliittymään ja käyttöliittymän tapahtumat. (Facebook 2019)

Action-luokat ovat lähettimen (Dispatcher) rajapinta käyttöliittymälle. Action-luokka standardisoi toiminnat samaan perusrakenteeseen ja sisältää pyynnön tehdä jokin tietty toimenpide ja siihen tarvittavat tiedot. (Facebook 2019)

Dispatcher-luokka ohjaa Action-luokat niille Store-luokille, joille kukin Action-luokka kuuluu. Dispatcher-luokka siis selkeyttää tiedonvälityksen yhteen kokonaisuuteen. Tällöin tiedon kulkureitit ovat selkeästi määritetty. Dispatcher-luokka pitää huolen tiedonkulun järjestyksestä ja voi myös hallita Action-luokan etenemistä eri Store-luokkiin. Jos esimerkiksi halutaan päivittää jokin Store-luokka ennen toista, sen voi määrittää Dispatcher-luokassa. (Facebook 2019)

Store-luokka pitää sisällään ohjelman datan, tilan ja ohjelmalogiikan. Store ottaa vastaan Action-luokkia, joista se on kiinnostunut, Dispatcher-luokalta. Store-luokka tulee rekisteröidä Dispatcher-luokalle. Tiedon virran järjestyksen ylläpito vaatii, että Store-luokkaan ei tehdä suoria komentoja, vaan komennot lähetetään Action Dispatcher-luokan kautta. Kun Store on käsitellyt Action-luokan komennon, se välittää muutostapahtuman (event) Controller-View-luokalle, joka toimii sille tarpeellisella tavalla, esimerkiksi päivittää muutetun listan käyttöliittymässä. (Facebook 2019)

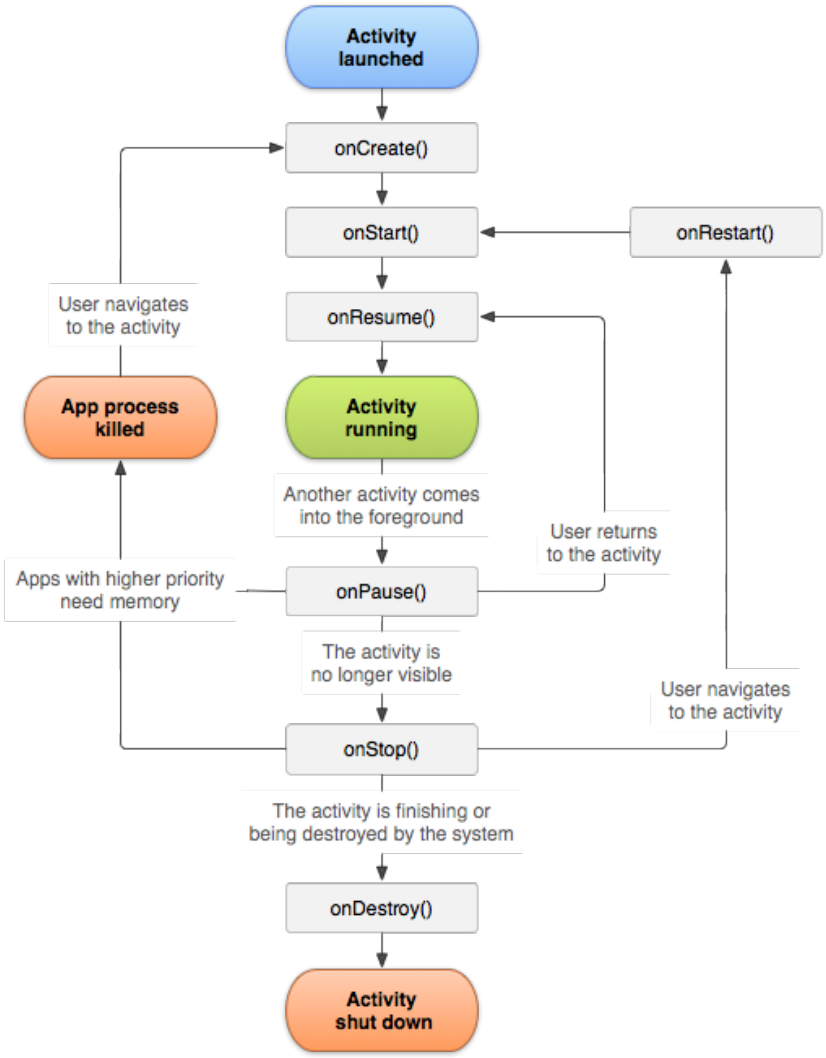
2.3 Android-alusta

2.3.1 Activity

Activity on luokka, joka luodaan aloituspisteeksi käyttäjän interaktiolle. Activitylla luodaan ikkuna, johon käyttöliittymä voidaan rakentaa. Activityn elinkaaressa on neljä tilaa: (Android Developers 2019a)

- Jos Activity on näytöllä näkyvänä päällimmäisenä, se on aktiivinen tai ajossa.
- Jos Activity on näkyvä, mutta ei päällimmäisenä, se on tauolla. Tauolla luokka ja sen muuttujat ovat muistissa, mutta erittäin alhaisen muistin tilanteessa, järjestelmä voi sammuttaa sen, eli poistaa muistista.
- Jos toinen Activity on kokonaan edessä, Activity on pysäytetty. Luokka ja sen muuttujat ovat muistissa, mutta jos muistia tarvitaan muualla, järjestelmä sammuttaa sen.
- Jos Activity on tauolla tai pysäytetty, järjestelmä voi sammuttaa sen. Tällöin jos käyttäjä palaa Activityyn, se luodaan kokonaan uudestaan. (Android Developers 2019a)

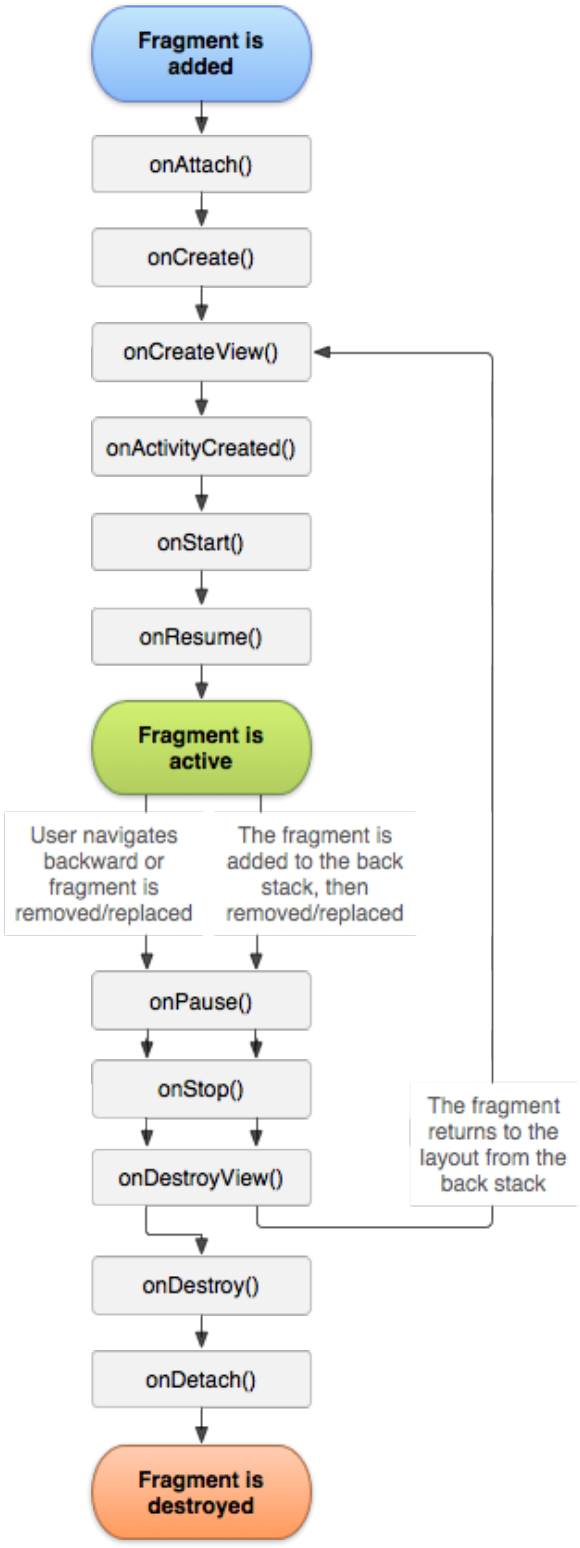
Activityn elinsyklimetodit ovat osa Activity-luokkaa ja joita järjestelmä kutsuu eri tilanteissa (Kuvio 4). Metodit kutsutaan eri tilanteissa käyttöjärjestelmän toimesta. elinsyklimetodien avulla pidetään huoli, että ohjelman toiminta ei aiheuta virhetilanteita, jos käyttäjä laittaa ohjelman taustalle tai järjestelmä sammuttaa sen virransäästösyistä. (Android Developers 2018i)



Kuvio 4. Activity elinsykli (Android Developers 2018i)

2.3.2 Fragment

Fragment on uudelleenkäytettävä käyttöliittymän osaa edustava luokka, jolla on oltava Activity-luokka yläluokkana. Fragmentin elinkaari muistuttaa Activity-luokan vastaavaa (Kuvio 5). Fragment-luokka onkin melkein kuin Activity. Activity-luokalla voi olla useita säiliöitä (Container) näkymässään, johon kuhunkin voidaan luoda Fragment-luokka. Toisin sanoen Fragment-luokkia voi olla useita samaan aikaan näytöllä. Koska Fragment-luokat ovat luokka Activity-luokan alla ja elossa samaan aikaan, niiden väillä voidaan jakaa luokkia, jolloin tiedonvälitys on vapaampaa kuin Activity-luokkien välillä. Activity-luokasta toiseen siirtyessä Activity-luokka sammutetaan ennen seuraavaa Activity-luokkaa ja tiedonvälitystä rajoittuu alustan omiin tapoihin. (Android Developers 2018d)

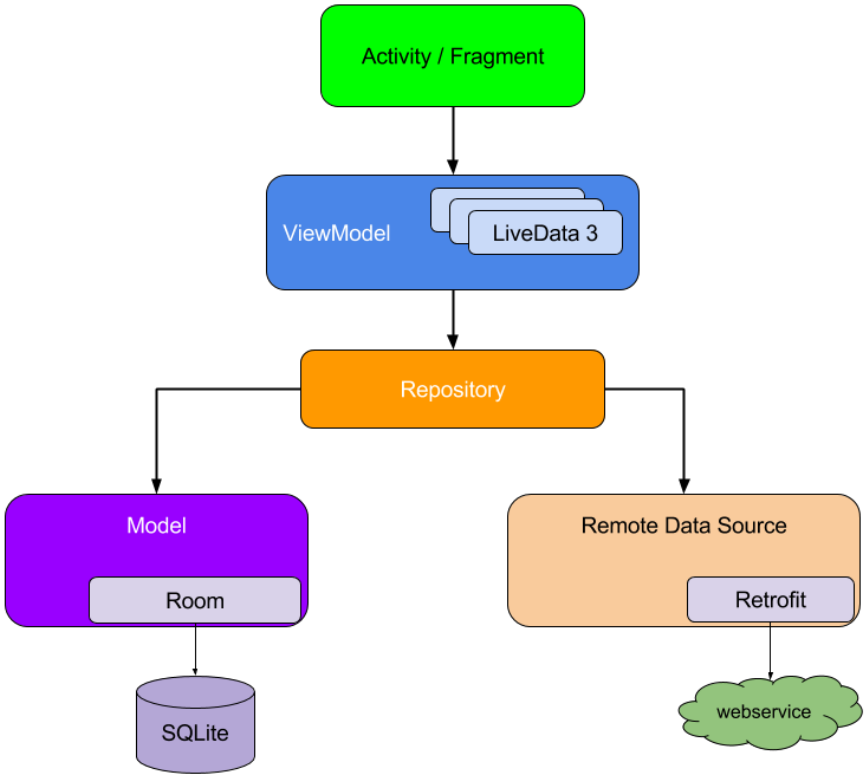


Kuvio 5. Fragment elinsykli (Android Developers 2018d)

2.3.3 Android Jetpack

Toukokuussa 2018 järjestetyssä Google I/O -konferenssissa (Martonik 2018) julkaistiin Android Jetpack (Prasad 2018). Jetpack on kokoelma Android-ohjelmistokomponentteja, joiden tarkoituksena on helpottaa ohjelmistokehitystä alustalle. Komponentit sisältävät vastauksia useisiin Android-alustan arkkitehtuurin haasteisiin. Navigaation toteuttamiseen on tarjottu omat komponentit, kuten myös arkkitehtuurin toteuttamiseen. (Android Developers 2019b)

Suositeltu arkkitehtuuri (Kuvio 6) Jetpack-moduuleissa pyrkii vastaamaan mobiilialustan monimutkaisuudesta seuraaviin haasteisiin: Yksi haaste on useiden eri applikaatiokomponenttien, kuten Activity, Fragment tai Service, välinen kommunikointi sekä navigointi. Toinen klassinen haaste on elinsyklin hallinta.



Kuvio 6. Android jetpack -arkkitehtuurimoduulien suositeltu käyttötapa (Android Developers 2019g)

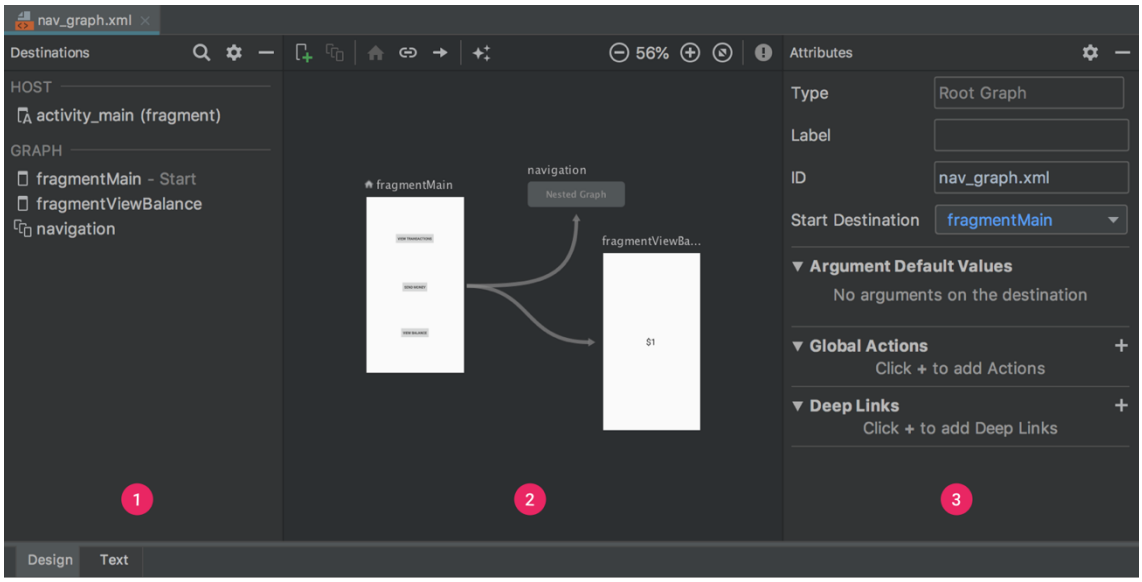
Jetpack-arkkitehtuurikomponentit on kehitetty siten, että eri komponentit vastaavat kukin omasta osa-alueestaan. Tavoitteena on erottaa toiminnallisuus pois Activityn tai Fragmentin sisältä. Näitä luokkia ohjaa käyttöjärjestelmä, joka voi sulkea tai käynnistää ne. Muiden komponenttien on siis kannattavaa olla mahdollisimman vähän riippuvaisia näistä luokista. Toinen olennainen periaate on ohjata Näkymää (View) Model-luokista käsin, jotka ovat riippumattomia View-luokista. Riippumattomuus ja tiedon erottaminen käyttöliittymästä on olennaista kahdesta syystä, jotka on listattu alle. (Android Developers 2019g)

1. Käyttäjät eivät menetä tietoa, jos käyttöjärjestelmä sulkee ohjelman esimerkiksi muistin vapauttamiseksi.
2. Ohjelma jatkaa toimimistaan, vaikka internetyhteys olisi epävarma. (Android Developers 2019g)

Arkkitehtuurisuosituksessa (Kuvio 6) ViewModel-luokka on tarkoitettu hallitsemaan tietoa niin, että siihen ei vaikuta konfiguraatiomuutokset, kuten ruudun kääntäminen. ViewModel-luokka myös tarjoaa rakenteen asynkronisten tietojen hakemiseen ja tavan välttää konfiguraatiomuutoksen seurauksena turhia tiedonhakuja rajapinnasta tai tietokannasta. (Android Developers 2019k)

Repository-luokat tarjoavat selkeän rajapinnan ohjelmalle hakea tietoa. niiden tehtävänä on hallita tiedonhakua ja tallennusta eri tietovarastoihin. Tällöin poistetaan tiedonhallinta eri varastojen välillä pois lopun ohjelman haasteista. (Android Developers 2019g)

Android jetpack navigaatiokomponentit tarjoavat uuden tavan rakentaa Fragment-pohjainen ohjelma. Siihen sisältyy graafinen editori Android Studio -kehitysalustaan (Kuvio 7). Editorilla voi luoda tiedoston, joka sanelee mahdolliset siirtymät sekä säätelee muun muassa taakse-napin toiminnan eri Fragment-näkymistä. (Android Developers 2019h)



Kuvio 7. Navigaatioeditori (Android Developers 2019e)

3 TOTEUTUS

3.1 Käyttöliittymä

3.1.1 Käyttöliittymäperiaatteiden soveltaminen

Taustaprojektin pohjalta ohjelmiston käyttöliittymä lähdetään suunnittelemaan asiakkaan ohjelmalle esittämien ominaisuuksien pohjalta. Taustaprojektissa vaadittavat pääominaisuudet ovat seuraavat:

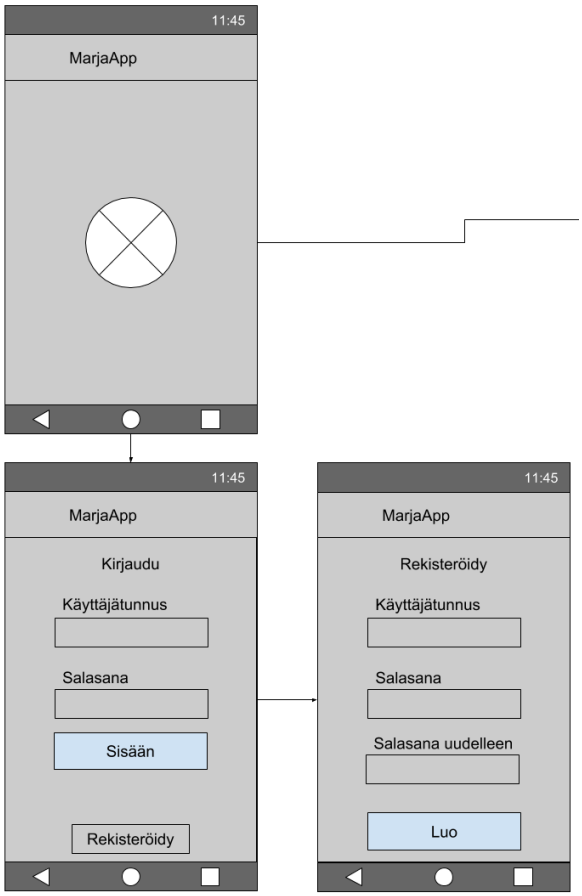
- Ohjelman tarkoitus on tarjota käyttäjälle marjastuspaikkojen sijainnin ja sadon kirjaamista.
- Mahdollisuus jakaa esimerkiksi sukupolvien välillä tietoa marjapaikoista.
- Tarjota ammattilaisille sekä harrastajapoimijoille keino esittää marjasadon alkuperä ilman, että tarkka marjasadon paikka paljastuu
- Mahdollisuus merkitä lähtöpaikka, jotta ohjelmaa voi käyttää tarvittaessa apuna löydettäessä takaisin esimerkiksi autolle.

Käyttöliittymän suunnittelu aloitettiin piirtämällä kynällä ja paperilla rautalankamallihahmotelmia. Rautalankamallissa nähtiin tuleva käyttöliittymä ja päästiin korjaamaan mahdolliset ongelmat, ennen kuin ohjelman valmistaminen on liian pitkällä. Mittavien muutosten tekeminen käyttöliittymään olisi ollut hyvin hankalaa ilman rautalankamallia.

Rautankamallista tehdään Mockup-malli, jossa toteutetaan rautalankamallista käyttöliittymä sen näköiseksi, kun lopullisen ohjelman tarkoitus on näyttää. Mockup-mallilla havainnollistetaan käyttöliittymän visuaalinen tyyli. Mockup helpottaa valitsemaan muun muassa ohjelman värimaailman ja sen avulla voi kysyä palautetta huolehtimatta siitä, että muutosten tekeminen ei vielä tule kalliiksi. (Mkrtchyan 2018)

3.1.2 Rautalankamalli

Ohjelman käyttöliittymän suunnittelu alkaa rautalankamallilla ja mallin kohdalla aloitetaan luontevasti siitä, kun ohjelma käynnistetään. Ohjelma aloittaa toimintansa käynnistyskuvalla noudattaakseen tavanomaista käytäntöä mobiiliohjelmissa. Toinen heuristisen evaluaation periaatteetta ”Jatkuvuus ja standardit” noudattava tapa on heti alkunäkymässä pyytää kirjautumaan tai rekisteröitymään käyttäjätunnuksella sekä salasanalla. Kirjautuminen on tarpeen jakamistoiminnon mahdollistamiseksi. Kirjautumisen myötä käyttäjään voidaan yhdistää käyttäjätunnus. Käyttäjätunnus toimii anonyyminä osoitteena, jolle muut käyttäjät voivat jakaa poimintapaikkoja (Kuvio 8).

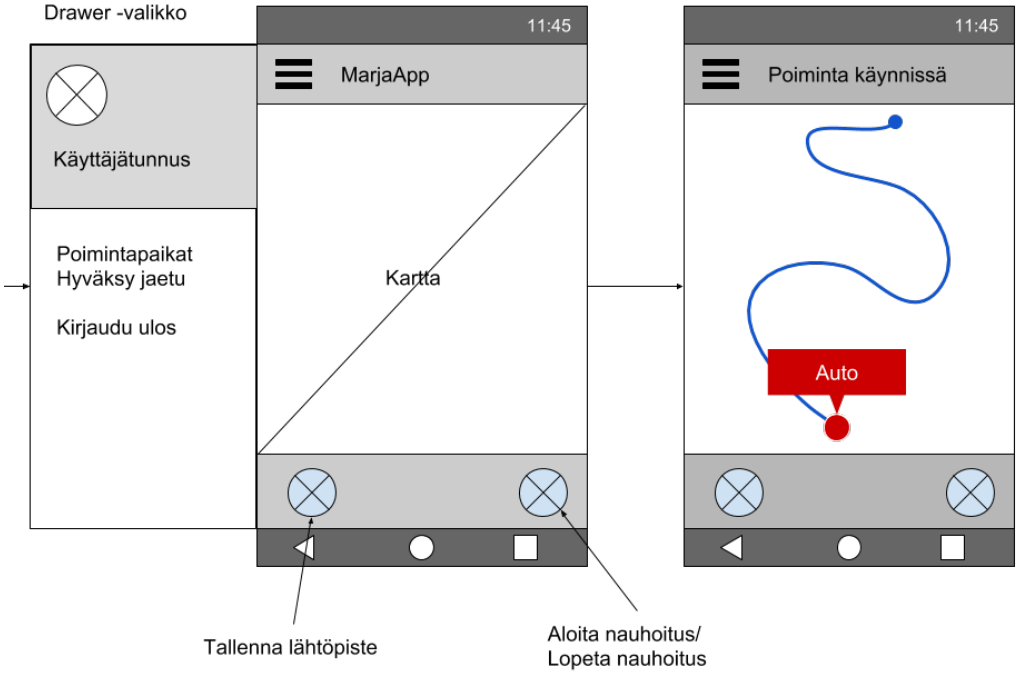


Kuvio 8. Käynnistyskuva, kirjautuminen ja rekisteröityminen

Kirjautumisnäkymässä käytetään hyväksi Gestaltin periaatteita. Läheisyysperiaatteella käyttäjätunnuksen kyselyyn yhdistetään kuvausteksti sekä tekstikenttä

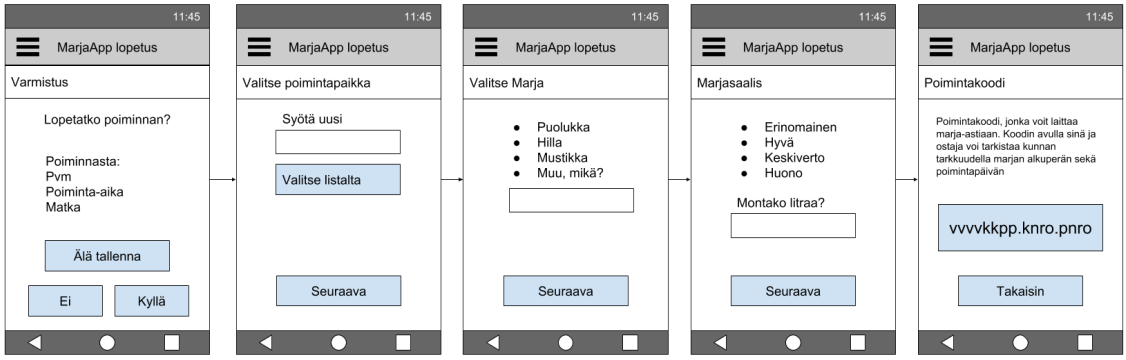
toisiinsa. Samoin tehdään alempana (Kuvio 8), laittamalla kuvausteksti ”salasana” sekä sitä vastaava tekstinsyöttökenttä, johon salasana annetaan, lähekkäin.

Heuristisen evaluaation periaate ”Minimalistinen suunnittelu” on pääperiaatteena ohjelman tärkeimmässä näkymässä (Kuvio 9). Vain olennaiset toiminnot ovat esillä karttanäkymässä, josta käsin voi aloittaa ja lopettaa poimintamatkan. Poimintamatkan lopettamisen ja aloittamisen lisäksi, näkymästä on käyttäjällä mahdollisuus merkitä lähtöpiste poimintamatkalle. Android-alustan jatkuvuuden mukaan valikkona käytössä on vetovalikko. Vetovalikko aktivoidaan joko painamalla symbolista ylävasemmalla tai ”vetämällä” valikko esiin vasemmalta.



Kuvio 9. Kartta ja poimintatila

Poiminnan lopettaminen karttanäkymässä johtaa poiminnan lopettamisnäkymiin (Kuvio 10). Ensimmäisessä näkymässä halutaan varmistaa ”Hallitsevuuden ja vapauden tunne käyttäjällä” -periaate: Mahdollisen vahinkopainalluksen takia kysytään käyttäjältä, haluaako käyttäjä varmasti lopettaa poiminnan.

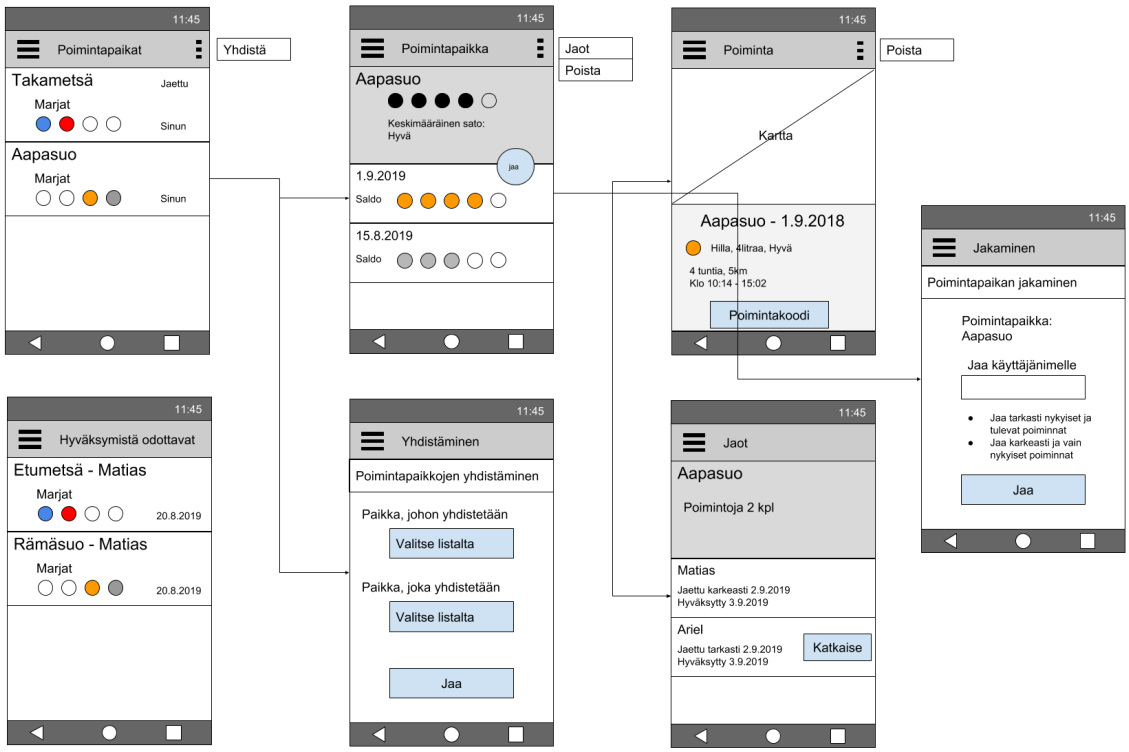


Kuvio 10. Poiminnan lopettamisen malli

Poiminnan lopettaminen vaatii usean näkymän läpikäymisen, mutta sarja kysymyksiä lopettamisen yhteydessä on asiakkaan ominaisuusvaatimus. Kysymyksillä saadaan tilastoitua käyttäjän saama marjasato. Kysymyssarja on jaettu useampaan näkymään selkeyden vuoksi mobiilialustassa, vaikka siitä seuraa painallusten määrän kasvu (Krug 2014, 148).

Viimeinen näkymä kertoo käyttäjälle poimintamatkaa vastaavan poimintakoodin. Koodin rakenne on suunniteltu siten, että se on helppo muistaa ja kirjoittaa marja-astian kylkeen. Vaatimuksena on mahdollisimman lyhyt ja looginen koodi. Koodi muodostuu päivämäärästä, kunnan numerosta ja juoksevasta poimintanumerosta kunnan alueella sinä päivänä.

Vetovalikosta voi katsella poimintoja tai hyväksyä käyttäjälle jaetut poimintapaikat. Poimintalistassa ”Yhteensopivuus systeemin ja todellisen maailman välillä” -periaatteen mukaan poimintapaikasta kuvataan siellä poimitut marjatyypit mieleenpainuvasti marjaa kuvaavalla värillä (Kuvio 11). Esteettömyys huomioidaan siten, että poimitun marjan saa selville myös Poimintamatkan tiedot -näkymässä tekstimuodossa.



Kuvio 11. Poimintapaikkojen katselu, jakaminen sekä jakamisen hallinta

Kustakin poimintapaikasta avautuu tarkemmat tiedot poimintapaikasta, kuten paikan satosaldo. Poimintapaikka koostuu poimintamatkoista, jotka listataan Poimintapaikan tiedot -näkymässä. Samassa näkymässä tarjotaan mahdollisuus jakaa poimintapaikka Android-alustalle tavanomaisen tapaan, pyöreällä painikkeella (Kuvio 11).

Poimintamatkalistasta pääsee poimintamatkan tiedot -näkymään, josta näkee tarkemmat tiedot. Näkymästä pääsee myös katsomaan uudelleen poimintakoodia (Kuvio 11).

Yhdistämisnäkymällä, johon pääsee poimintalistan asetuksista, voidaan yhdistää kaksi poimintapaikkaa (Kuvio 11). Näin käyttäjällä on mahdollisuus järjestää uudelleen poimintapaikat periaatteen ”Hallitsevuuden ja vapauden tunne käyttäjällä” -mukaan.

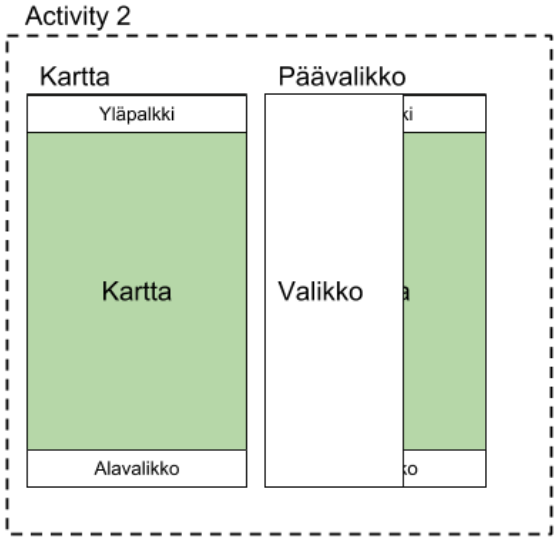
Poimintapaikan tiedoista voi katsoa, miten marjapaikkaa on jaettu. Näin käyttäjä voi kokea hallitsevansa keräämäänsä tietoa (Kuvio 11). Samasta periaatteesta on olemassa jakojen hyväksyntä -näkymä. Käyttäjälle halutaan tarjota oman tietokantansa kontrolli.

3.1.3 Kartan käyttöliittymä

Käyttöliittymä paikkatietopohjaisessa ohjelmassa tarkoittaa tyypillisesti kartan hyväksikäyttöä ja siihen liittyviä käyttöliittymäelementtejä. Ohjelmointialusta, kuten Android tarjoaa karttakäyttöliittymän valmiiksi. Samalla siihen sisältyy muita kartoissa tyypillisiä käyttöliittymäelementtejä, kuten kohteen ponnahdusikkunat tai karttasymbolit (Android Developers 2018e).

Karttanäkymän käyttöliittymä kuvataan alustakohtaisesti. Android alustassa valmiina on tarjolla seuraavia elementtejä kartan ympärille: Lähennys/loitonnuspainikkeet, kompassi, Oma sijainti -painike, tasonvalintapainikkeet, karttatyökalupalkki (Android Developers 2018c).

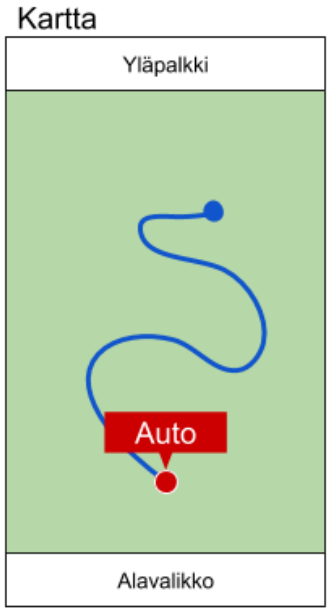
Yleisellä tasolla pääkartan näkymän ympärillä on käyttöliittymäelementteinä yläpalkki ja alavalikko (Kuvio 12). yläpalkista voi avata päävalikon.



Kuvio 12. Activity 2 Pääkartan käyttöliittymäkuvaukset

Android-kartta tarjoaa rajapinnan piirtää pistesijainnille kuvakkeen, jolla voi olla infoikkuna. Myös eri muodot, kuten ympyrät, kuutiot ja viivat ovat tuettuja (Android Developers 2018e).

Karttaelementeistä projektin osalta on tarpeen käyttää viivaa, pistesijaintia ja infoikkunaa (Kuvio 13). Käyttäjän sijainti ja mahdollinen tallennettu auton sijainti merkitään kartalle kuljetun matkan lisäksi.



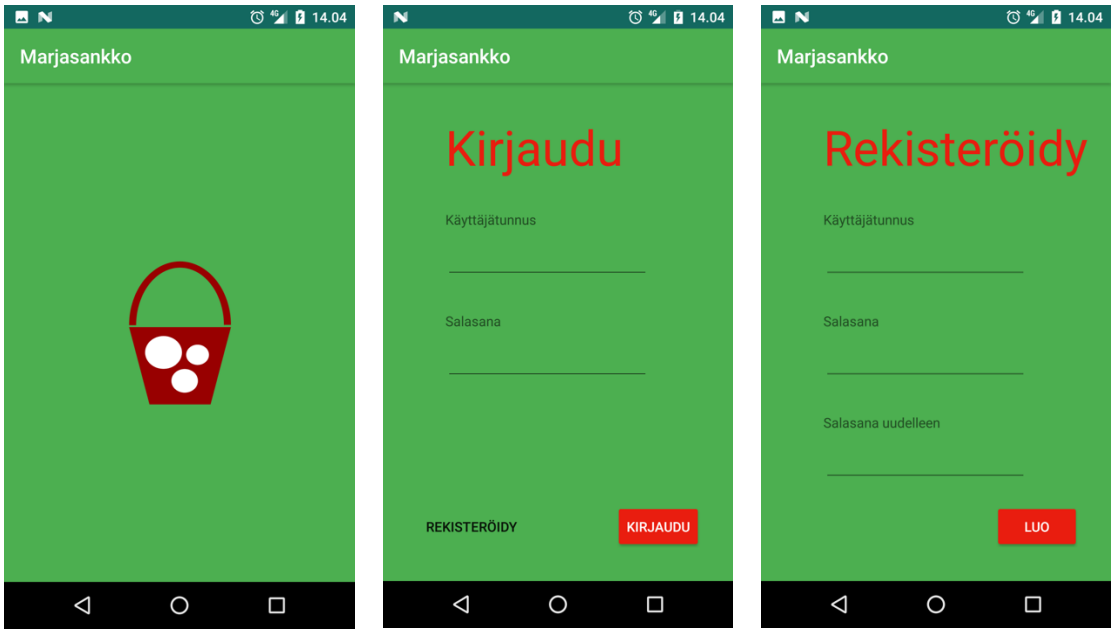
Kuvio 13. Käytettävät kartan käyttöliittymäelementit

Paikkatiedon esittäminen käyttäjälle asettaa haasteen käyttöliittymälle. Koordinaatit ovat numeroina hankalasti hahmotettavissa. Piirtämällä koordinaatit kartalle, saadaan karttakuva. Interaktiivinen digitaalinen kartta on tarpeen alistaa samoille käyttöliittymästandardeille, kuin muutkin käyttöliittymän näkymät ja komponentit (Nivala 2007, 11).

3.1.4 Käyttöliittymämalli

Käyttöliittymämalli tehdään rautalankamallin pohjalta. Malli on toteutettu oikealla visuaalisella tyylillä ja sisällöllä. Mallin tarkoitus on kokeilla rautalankamallia oikealla sisällöllä ja viimeistellä ulkonäkö todellisessa ympäristössä, oikeilla fonteilla, näyttökoolla ja väreillä.

Tarkoitus ei opinnäytetyössä ole edetä taustaprojektin osalta rautalankamallia pidemmälle, mutta teorian soveltamisen esimerkkinä on tehty kolme ensimmäistä näkymää Mockup-malliksi (Kuvio 14). Mockupit tehtiin seuraavista näkymistä ja niiden rautalankamalleista: Käynnistyskuva, kirjautuminen ja rekisteröityminen (Kuvio 8).

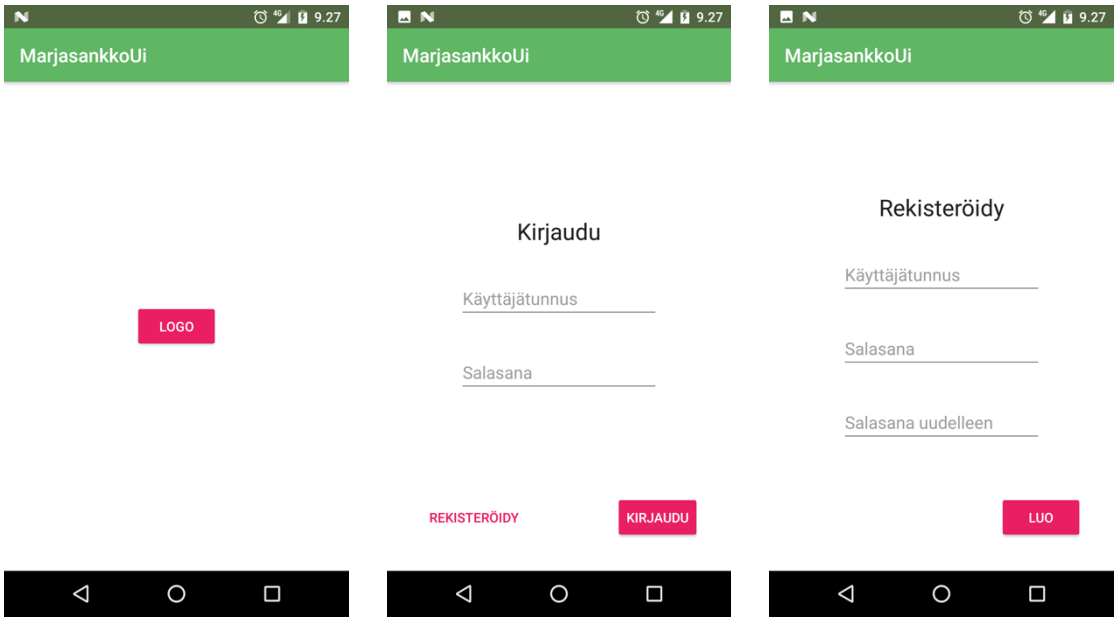


Kuvio 14. Marjasankko-ohjelman kirjautumisnäkymien Mockup-mallit

Käynnistyskuva on toteutuksena yksinkertainen, mutta yksinkertaisuus sisältää paljon graafista suunnittelua. Käynnistyskuvan toteuttaminen vaatii, että pitää suunnitella ohjelman kuvake (logo) ja valita pääväriteema. Käynnistyskuvan Mockup selkeästi esittää tämänkaltaisen mallin hyödyn. Tässä vaiheessa, jos värimaailma tai ikoni eivät täytä käyttöliittymäsuunnittelun periaatteita tai ole kehittäjän mielestä visuaalisesti miellyttäviä, ne ovat suhteellisen helppo huomata ja korjata.

Kirjautumisen sekä rekisteröitymisen Mockup-mallit noudattavat tyypillistä vastaavien näkymien kaavaa, noudattaen täten Jatkuvuus ja standardit -heuristista sääntöä. Värimaailma antaa paljon toivomisen varaa, kuten punaisen otsikkotekstin luettavuus. Toinen ongelma on gestalt-lain ”läheisyys” noudattamatta jättäminen: Käyttäjän syötekenttä ja kuvausteksti eivät ole enemmän lähellä toisiaan kuin muita elementtejä. Taustavärin yli on myös hankalampi lukea tai huomata kuvaustekstejä ja syötekenttiä.

Korjatuissa versioissa (Kuvio 15) syötekentän ja kuvaustekstin läheisyysongelma on korjattu laittamalla kuvausteksti tekstikenttään sisälle vihjeeksi (Hint). Tällöin näkee selkeämmin, mihin tekstikenttään kukin kuvausteksti liittyy, koska kuvaus on tekstikentässä ja marginaalia on muihin tekstikenttiin. Toinen korjaus on Värien kontrasti. Taustaväriksi on valittu valkoinen, jotta sisältö erottuisi paremmin.



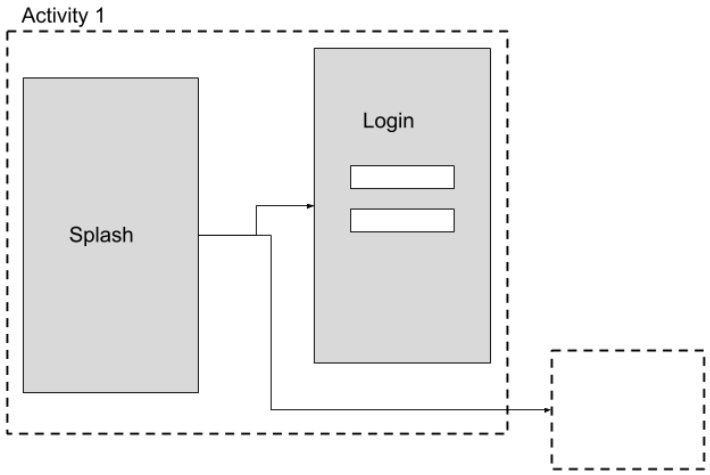
Kuvio 15. Kirjautumisen Mockup-mallin korjatut versiot

3.2 Arkkitehtuuri

3.2.1 Näkymien jakaminen Activity-luokkiin

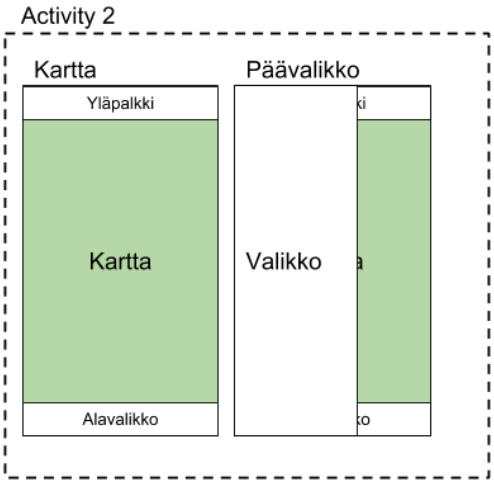
Android-alustassa käyttöliittymänäkymän perusrakenne on luoda Activity-luokka. Näkymä toimii rajapintana käyttöliittymänäkymän ja toiminnallisuuden välillä (Android Developers 2018a). Alustalla on mahdollista tehdä Näkymän sisällä omia alinäkymiä, jotka ovat sidottu luokkaan nimeltä Fragment (Android Developers 2018d). Kuvioissa Activity merkitään katkoviivalla ja sen sisään on piirretty näkymät, jotka ovat sidottu kukin omaan Fragment-luokkaan.

Ohjelman lähdekoodin modulaarisuutta tuetaan käyttöliittymän koodin osalta siten, että ohjelman näkymät pilkotaan Activity-luokkiin. Ensimmäinen Activity käynnistyttyään, ohjaa käyttäjän kesken jääneeseen Activity-luokkaan ja sen näkymään tai hoitaa kirjautumisen. Ensimmäisellä Activity-luokalla on kirjautumisnäkymät (Kuvio 16). Rautalankamallista Activity-luokan näkymät olisivat Käynnistyskuva (splash screen), kirjautuminen (login) ja rekisteröityminen.



Kuvio 16. Activity 1 näkymät ja eteneminen seuraavaan Activity-luokkaan

Pääkartan Activity-luokan ja sen näkymien annetaan keskittyä pääosin kartan toimintoihin (Kuvio 17). Pääkarttanäkymästä voidaan päävalikon sekä alavalikon kautta siirtyä toisiin Activity-luokkiin ja niiden näkymiin.



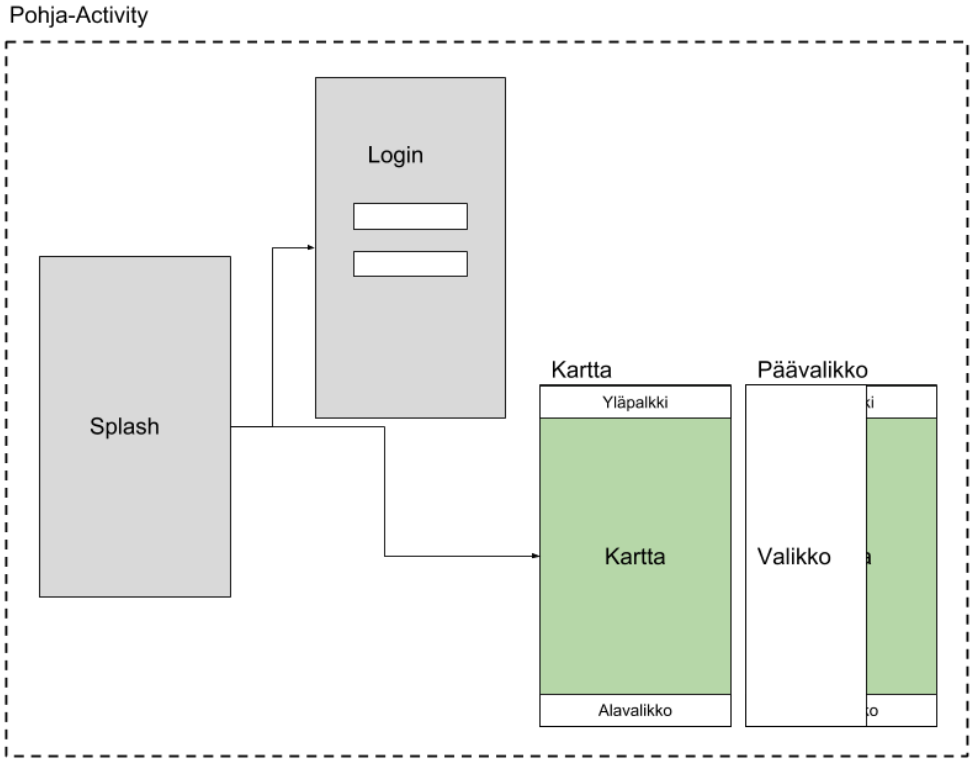
Kuvio 17. Activity 2 Karttanäkymä

Ohjelman toiminta on tarkoitus aloittaa ensimmäisellä Activity-luokalla ja sen ensimmäisellä näkymällä, jossa käyttäjä näkee aluksi logon ja odotuspalkin (Kuvio 16). Taustalla ladataan tarvittavat tiedot rajapinnasta. Jos käyttäjä palaa ohjelmaan käynnistyskuvakkeen kautta tai ohjelma on sammunut kesken, ensimmäinen näkymä ohjaa käyttäjän kesken jääneeseen tilanteeseen muistiin tallennetun tilan perusteella.

Ensimmäisellä käynnistyksellään ensimmäinen Acitivity vaatii käyttäjältä kirjautumisen uuteen tai olemassa olevaan tiliin rajapinnassa. Kirjautumisen jälkeen vaihdetaan toisen Activity-luokan karttanäkymään (Kuvio 17).

3.2.2 Yhden Activity-luokan arkkitehtuuri

Android ohjelmistoprojekti voi koostua useammasta kuin yhdestä Activitystä. Tyypillisesti ohjelmassa, joka on toteutettu usealla Activityllä, Yksi Activity vastaa yhtä käyttöliittymän näkymää (Rabetckiy 2019). Kuitenkin useiden Android Activityjen käyttäminen ei ole niin joustavaa kuin rakentamalla ohjelma yhden Activityn periaatteella. Tällöin yhtä näkymää vastaa yksi Fragment ja kaiken pohjalla on yksi Activity (Kuvio 18). Tämän kaltainen Yhden Activityn ja usean Fragment-luokan rakenne mahdollistaa joustavamman rakenteen, tiedonvälityksen näkymien välillä ja nopeammin toimivan käyttöliittymän. (Hinchman 2018)



Kuvio 18. Näkymät yhdessä pohja-Activityssa

3.2.3 Paikkatiedon asettamat vaatimukset

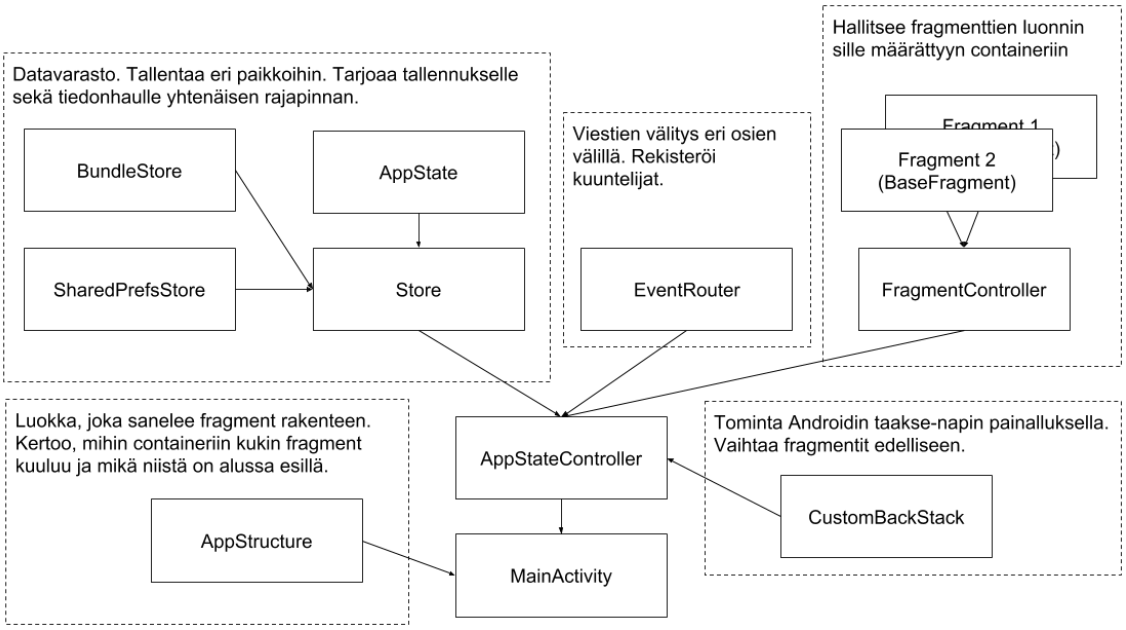
Paikkatiedon haasteet on hyvä tiedostaa mobiiliympäristössä. Ohjelma voidaan suunnitella alusta asti siten, että vakaa toimivuus olisi mahdollisimman taattu. Esimerkiksi karttanäkymä on hyvä luoda oman Activity-luokan alle mahdollisimman usein, koska Activity-luokan vaihtaminen vaikuttaa poistavan kokemuspohjaisesti varmemmin viittaukset paljon muistia vieviin karttakomponentteihin. Kartan käyttämistä on tarpeen rajoittaa, koska kartta on rasite muistin käytölle mobiilialustalla. Kartta esittää kuvia käyttöliittymässä ja kuvat vievät ajomuistia (Arya 2017). Paikkatieto asettaa palvelinrajapinnalle lisähaasteen, jos tavoitteena on saada käyttöön oma taustakartta.

Karttanäkymän mahdollisten haasteiden vuoksi ohjelmassa perusperiaatteena pyritään jakamaan näkymät, jossa käytetään karttaelementtiä, oman Activity-luokan alle. Jakamisesta näin on lisäksi se hyöty, että lähdekoodi on keskittynyt kartan ominaisuuksiin ja muut toiminnallisuudet löytyvät toisten Activity-luokkien alta.

3.2.4 Arkkitehtuurin toteuttaminen Android-alustalla

MVC-arkkitehtuurin kokeilemiseen Android-alustalla käytännön tasolla, ilmenee useita haasteita. Android-alustasta puuttuu paljon sisäisiä MVC-arkkitehtuuria tukevia komponenttejä erityisesti yhden Activity-luokan, usean Fragment-luokan rakenteessa. Tarvittavia toimintoja ovat muun muassa Fragment-näkymien vaihto niin, että sen luonti on mahdollista mahdollisimman yksinkertaisella signaalilla, kuten pelkällä näkymän nimellä. Näin siksi, että fragmentin vaihto esimerkiksi nappia painamalla ei tarvitse kopioida samaa koodia. Toinen syy keskittää Fragment-hallinta on ylläpitää Fragment-historiaa, jotta voidaan toteuttaa takaisinnapin toiminta myös Fragment-luokkien osalta. Alustalla on tarjolla taaksenapin toiminta palatessa edeltävään Activity-luokkaan (Zhao 2017). Fragmenttien tallentaminen jonoon voi aiheuttaa muistivuodon, koska Fragment-luokan elinsykli on sidottu sen Activity-luokkaan, jolloin monen Fragment-luokan ohjelmaan voi jäädä suuri määrä Fragment-luokkia muistiin taustalle (Android Developers 2018d). Ratkaisuna on tuhota Fragment heti seuraavan alta, jolloin sitä ei enää löydy jonosta. Haittana tässä ratkaisussa on, että silloin tarvitaan luoda oma, yllä ehdotettu, takaisinnapin toimintaperiaate, jossa tallennetaan esimerkiksi vain käytyjen Fragment-luokkien nimet omaan listaan.

Opinnäytetyötä varten kokeiltiin arkkitehtuurimallia, joka vastaisi lukuisiin haasteisiin, joita on tullut Android-ohjelmoinnissa vastaan. Tuloksena syntyi kokonaisuus, joka sitoo yhteen useita ratkaisuja käytännön ongelmiin (Kuvio 19). Arkkitehtuuri pyrkii noudattamaan MVC-mallin periaatetta ja on vahvasti inspiroitunut FLUX-mallista.



Kuvio 19. Käytännön kokeiltu MVC-rakenne Android-alustalla

Toteutettu arkkitehtuurimalli pyrkii vastaamaan useisiin eri osa-alueiden haasteisiin, joita Android-alusta tuo arkkitehtuuriin. Osa-alueita ovat Fragment-navigaatio, tiedon varastointi, tiedon välitys ja toiminnallisuuden erottaminen käyttöliittymästä MVC:n periaatteiden mukaan. Alla tarkemmin kuvattuna toteutetun arkkitehtuurimallin periaatteet osa-alueittain.

- Datavarasto on alustariippumaton tiedon tallentamisen ja varastoinnin rakenne. Store-luokka annetaan kullekin Fragment-luokalle (View-Controller), jotka voivat suoraan ajaa luokan metodeja ja siten tallentaa sekä hakea tietoa.
- Viestien välitys (EventRouter) tarjoaa kanavan, jossa kukin luokka voi keskustella toistensa kautta ilman injektiota. Näin tarjotaan interaktiokanava, joka välttää muistivuodot. Sisäänrakennetusti sitä käytetään lähettämällä käsky vaihtaa Fragment tai Activity. Sekä AppStateController, että FragmentController ovat rekisteröityjä kuuntelijoita EventRouter-luokassa. EventRouter annetaan Fragment-luokalle, kun se luodaan.
- AppStructure-luokan ajatus on, että se sanelee ohjelmakohtaisen Fragment-rakenteen. Android-alustassa Activity-luokalla voi olla useita Fragment-luokkia esillä omassa säiliössään (container). AppStructure-

luokka sitoo yhteen kunkin fragment-luokan kuhunkin säiliöön. Tämän sitomisen jälkeen kunkin fragmentin sai käynnistettyä sille kuuluvaan säiliöön pelkällä Fragment-luokan nimellä.

- AppStateController tarjoaa vakioidun rajapinnan, joka on helppo sitoa uuteen Activity-luokkaan. Activity-luokassa sen elinsyklimetodeissa kutsutaan tämän luokan metodeja ja niissä metodeissa on tarvittavat toiminnallisuudet, kuten onStart-metodissa luodaan Fragment näkymään.
- CustomBackStack-luokalle tallennetaan kunkin luodun Fragmentin nimi. joka tallentaa ne jonoon. Taakse-nappia painettaessa BackStack poistaa jonon viimeisimmän nimen ja palauttaa toiseksi viimeisimmän. Nimen Fragment luodaan FragmentController-luokassa näytölle. Jos jono on tyhjä, palauttaa luokka tyhjää, jolloin ajetaan onBackPressed-metodin super.
- FragmentController luo fragment-luokan sille kuuluvaan säiliöön annetun nimen perusteella.

3.2.5 MVC-arkkitehtuuri toteutuksessa

Model on datarakenne ja kokeillussa arkkitehtuurissa sitä edustaa Datavaraston Store-luokka. Ajatuksena on tarjota yhtenäinen rajapinta tiedonhakuun ja tallennukseen, alustariippumattomasti (Lisää kappaleessa ratkaisut, tilamuisti). Tietorakennetta ei kuitenkaan toteutettu yksityiskohtaisesti, kun sen lopullinen muoto on riippuvainen ohjelmasta, jota varten sitä kehitetään. Ainoat tietorakenteeseen tallennetut tiedot kokeillussa arkkitehtuurissa, olivat nykyisen tilan palauttamiseen tarvittavat muuttujat. Lisäksi omatekoinen taakse-napin toiminta tarvitsi tallentaa näkymähistoria, jotta se voi ohjata takaisin edeltävään näkymään.

View on arkkitehtuurissa Fragment-luokkien rooli. FLUX-inspiraationa Fragment muodostaa View-Controller -rakenteen, jossa Controller-ominaisuudet koskevat vain kyseistä näkymää. Tarkemmin toteutuksessa Fragment-luokkaa ei lähdetty erittelemään View ja Controller osiin, koska testirakenteella ei ollut toiminnallisuutta.

Controller-osa on loput luokat. Toisaalta myös FLUX-inspiraatio näkyy tässäkin osassa. Esimerkiksi käyttäjän syöte ei välttämättä kulje Controller-luokan kautta, vaan syötteestä lähetetään Event EventRouter-luokalle, joka ohjaa sen kaikille kyseistä Event-tyyppiä kuunteleville. Nämä voivat vastata lähettämällä oman Eventin EventRouterille, jota View voi kuunnella. Jos Controllerin tarkoitus on MVC-arkkitehtuurissa ohjata dataa Model-luokista View-luokkiin ja takaisin, niin toteutetussa arkkitehtuurissa keskitytään enemmän näkymien luontiin oikeassa järjestyksessä.

3.2.6 Kokemukset toteutuksesta

Arkkitehtuurisen rakenteen toteuttamisen ja testaamisen hyöty on mittava sen tuoman kokemuksen kautta. Toteutettu arkkitehtuuri vastasi useisiin Android-alustan käytännön järjestelyhaasteisiin, mutta käytännön toteuttaminen paljasti toteutuksessa useita haasteita.

Ensimmäinen haaste toteutuksessa on päällekkäiset järjestelmät ja siitä seuraavat arvaamattoman toiminnan riskin kasvu. Luokkia voi toteutuksessa välittää EventRouter-luokan kautta tai normaalisti injektiolla. AppStructure-luokka voi luoda itse annettujen parametrien perusteella Fragment-luokan, mutta myös ottaa vastaan Fragment-luokan parametrina. Päällekkäiset järjestelmät eivät noudata periaatteella KISS (Keep it Simple Stupid). Usea ratkaisu toteutuksessa on ollut vaikeasti perusteltavissa. Hyvät ratkaisut ovat tyypillisesti hyvin perusteltavissa.

3.2.7 Tulevaisuus, muutokset ja Jetpack

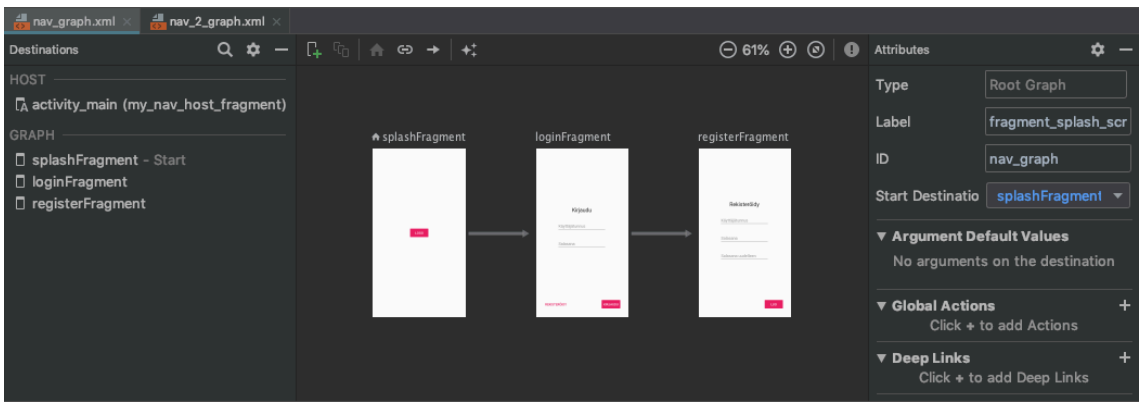
Vuonna 2008 julkaistiin ensimmäinen Android-puhelin (Callaham 2018). Yli kymmen vuotta Android-alusta on jatkuvasti muuttunut (Plesac 2015). Opinnäytetyössä toteutettu arkkitehtuurirakenne pyrkii vastaamaan moniin samoihin haasteisiin, joihin myös vuonna 2018 julkaistu Jetpack-ohjelmistokomponentit pyrkivät vastaamaan. Haasteita olivat esimerkiksi MVC-pohjaisen arkkitehtuurin toteuttaminen alustalle sekä Fragment-navigaatio.

Android-alustan kehityksen nopeutta kuvaa se, että opinnäytetyön tavoitteet, kehittää Android-alustalle projektia varten arkkitehtuuri, syntyivät vuonna 2018. Samana vuonna Jetpack-komponentit julkaistiin. Tutustuminen Jetpack-komponentteihin tapahtui opinnäytetyön arkkitehtuuritoteutuksen jälkeen. Tämän takia opinnäytetyössä toteutettu arkkitehtuuri ei käytä hyväkseen Jetpack-komponentteja.

Kokemukset toteutuksesta -osassa todetaan, että kokeiltu toteutus ei vastannut parhaalla tavalla arkkitehtuurille asetettuihin toiveisiin. Seuraava askel olisikin kokeilla Jetpack-komponentteja sekä noudattaa Jetpack-komponenttien mukana tullutta Android-alustan suositeltua arkkitehtuurirakennetta.

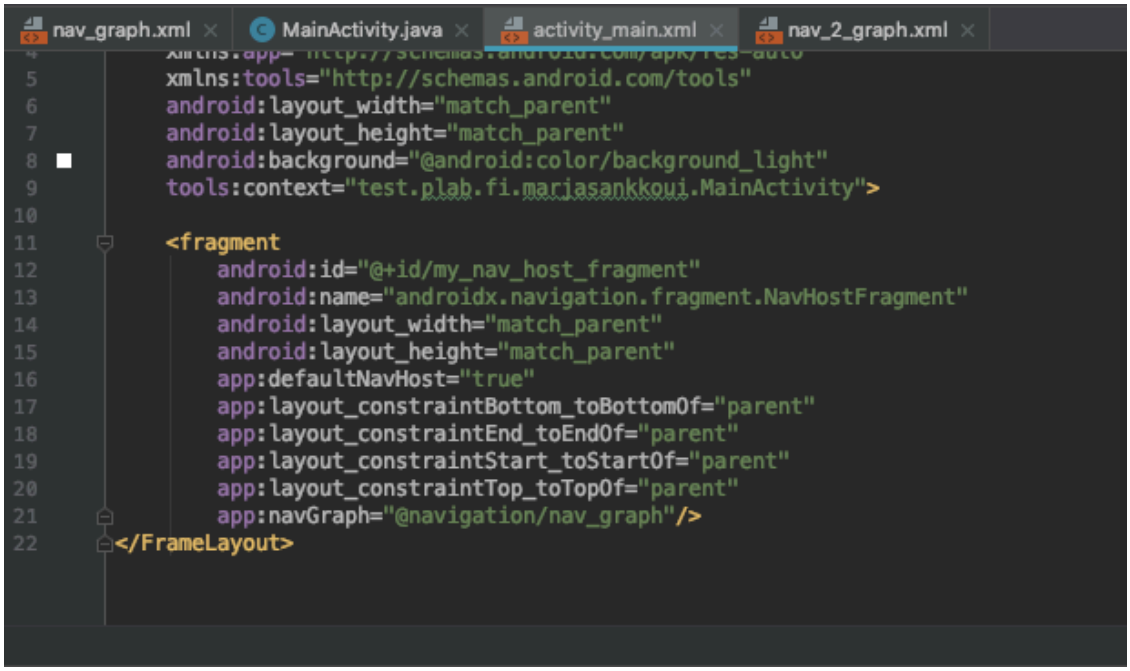
Kokeillusta arkkitehtuurista Store-rakenne on suoraan yhteensopiva repository-rakenteeseen ja olisi käytettävissä lähes suoraan sellaisenaan ilman suurta uudelleenohjelmointia. Kuitenkin tarve Store-rakenteen sisällä tallentaa ja palauttaa tietoja, kun ohjelma kokee konfiguraatiomuutoksen, poistuisi ViewModel-luokan käytön myötä. Tällöin Store-rakenteelle jäisi tehtäväksi vain tarjota selkeä rajapinta ohjelmalle tallentaa ja hakea tietoa taustalla olevista tietolähteistä riippumattomasti. Jetpack-komponenttien tuomasta muutoksesta on alla esimerkki navigaation tapauksessa.

Appstructure-luokkaa korvautuu täysin navigaatioeditorilla (Kuvio 19) tehtävällä XML-tiedostolla. Navigaatioeditori ja sen xml-tiedostoa käyttävät komponentit poistavat kokonaan oman backstack-luokan käytön tarpeen, koska komponenttien sisäinen backstack hallitsee taakse-napin toimintaa Fragmenteilla xml-tiedoston mukaisesti.



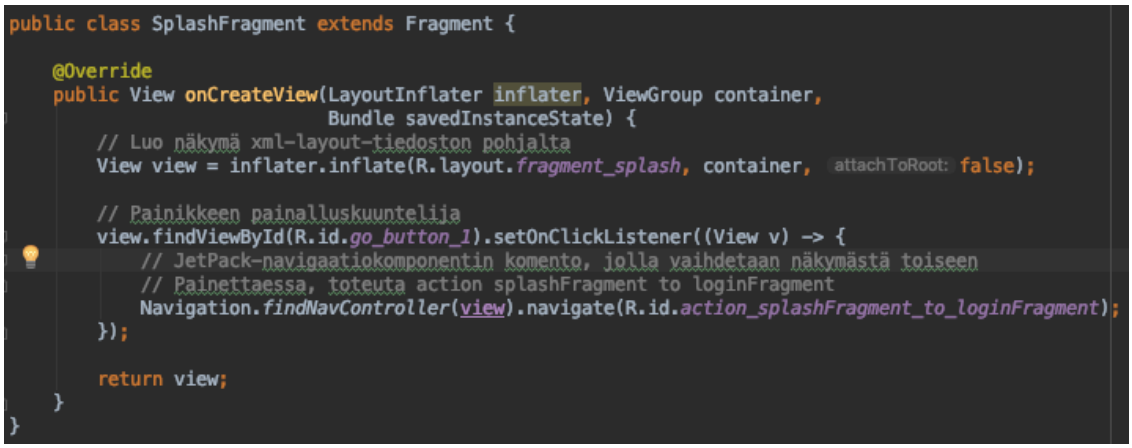
Kuvio 19. Navigaatioeditorilla toteutetut ensimmäiset näkymät

Jetpack-navigaatiokomponenteille kerrotaan navigaatiorakennetiedosto Activity-luokan layout-tiedoston Fragment-säiliössä (Kuvio 20). Fragment-säilö on xml-kuvauskielisen tiedoston fragment-elementti ja sen NavGraph-parametri linkittää navigaatiotiedostoon.



Kuvio 20. Fragment-elementti ja sen navGraph-parametri

Navigaatiotiedosto ei määritä sitä, mikä painike aiheuttaa siirtymän näkymästä toiseen. Se toteutetaan Jetpack-komponenteissa ohjelmallisesti. Napin painalluskuuntelijan ajamassa koodissa ajetaan navigaatiokomponentin metodi ”navigate”, jolle annetaan parametrina XML-navigaatiotiedoston siirtymä, jota kuvaa editorissa nuoli (Kuvio 19) ja lähdekoodissa tunniste (Kuvio 21). FragmentManager-luokka, jolla aikaisemmin tehtiin siirtymät Fragmentista toiseen, on piilotettu taustalle.



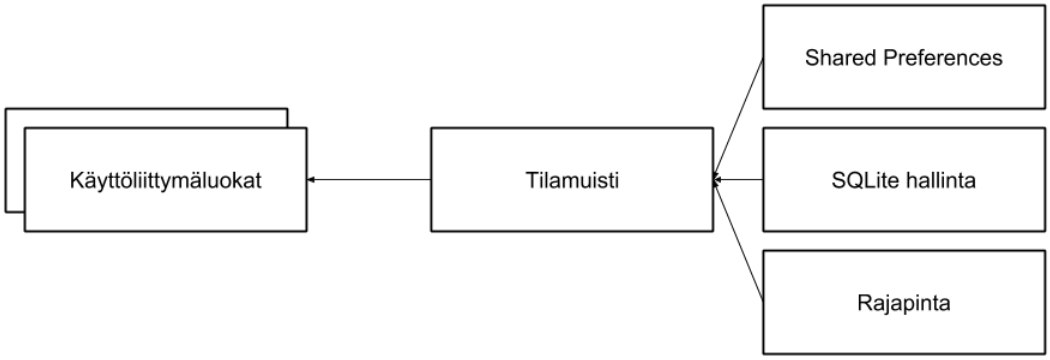
```
public class SplashFragment extends Fragment {

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                             Bundle savedInstanceState) {
        // Luo näkymä xml-layout-tiedoston pohjalta
        View view = inflater.inflate(R.layout.fragment_splash, container,  attachToRoot: false);

        // Painikkeen painalluskuuntelija
        view.findViewById(R.id.go_button_1).setOnClickListener((View v) -> {
            // JetPack-navigaatiokomponentin komento, jolla vaihdetaan näkymästä toiseen
            // Painettaessa, toteuta action splashFragment to loginFragment
            Navigation.findNavController(view).navigate(R.id.action_splashFragment_to_loginFragment);
        });

        return view;
    }
}
```

Kuvio 21. Vaihtaminen näkymästä toiseen ohjelmallisesti JetPack-navigaatio-komponentilla Fragment-luokassa

4 RATKAISUT

4.1 Tilamuisti

Tilamuisti on konsepti, jossa käyttöliittymän tiedonhallinta keskitetään tilamuisti-luokalle. Tiedonhallintarakenne (Kuvio 22) pyrkii keskittämään ohjelman toiminnassa tarvittavat tiedot sekä tilat. Tilamuisti tehdään luokkana, jolta käyttöliittymä hakee tarvitsemaansa tietoa.



Kuvio 22. Alustava tiedonhallintarakenne

Tilamuisti tulee olemaan asynkroninen. Tilamuistiin voi käyttöliittymä myös tallentaa nykyisen tilan, jotta käyttöliittymä voi hakea nämä tiedot takaisin muistista ja jatkaa siitä, mihin jäätiin. Käyttöliittymä voi määrätä, mihin tieto tallennetaan. Vaihtoehtoja on alustavasti kolme.

1. Shared Preferences on Android alustan tarjoama asetustietojen tallennusrajapinta. (Android Developers 2018h)
2. SQLite-tietokanta löytyy Android-alustasta. Tietokantaan on hidasta tallentaa ja lukea, mutta se kykenee tallentamaan paljon informaatiota pysyvästi. (Android Developers 2018f)
3. Rajapinta tarkoittaa verkkopalvelinta, minne tallennetaan käyttäjätiedot. Rajapinta mahdollista myös jakamisominaisuudet eri käyttäjien välillä.

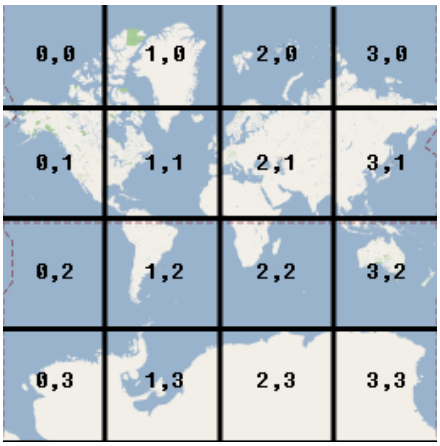
Tilamuistin tarkoituksena on yhtenäistää tietovaraston toiminnallisuus kautta koko ohjelman. Tällöin vältetään mahdollisimman paljon tilannetta, jossa eri koodirivit tekevät samaa asiaa. Toinen tarkoitus on vähentää hakuja tietokantaan,

asetustiedostojen rajapintaan ja verkkopalvelimelle. Tilamuisti jättää haetun tiedon muuttujaan ja palauttaa uudella hakupyynnöllä muistiin jätetyn muuttujan. Tämä säilytystapa palauttaa tiedon muistista aina, kun tieto on asetettu muuttujaan. Muuttuja tyhjennetään, kun sama tieto kirjoitetaan yli ja jos tilamuistiluokka poistetaan kokonaisuutena. Käytännössä siis hakeakseen tietoa, käyttöliittymä luo uuden tilamuistiluokan ja lähettää hakupyynnön tiedolle, joka palautetaan asynkronisesti. Käyttöliittymän ei tarvitse ottaa kantaa, mistä tieto haetaan ja menettelytapa on kaikille muille käyttöliittymänäkymille sama.

Tilamuuttujaan on sisään ohjelmoitu tieto, mistä kutakin tietoa haetaan ja minne tieto tulee tallentaa. Tilamuistiin voi myös luoda ominaisuuden, jolla voidaan lähettää lista tarvittavista tiedoista ja tilamuisti palauttaa kaikki kerralla, kun haut on suoritettu. Haut siis ketjutetaan ja käyttöliittymän tarvitsee tehdä vain yksi hakupyyntö, saadakseen tarvittavat tiedot toimiakseen.

4.2 Taustakartta

Kartta, oli kyseessä verkkosivun kartta tai mobiilisovelluksen kartta, käyttää karttatiilejä. Karttatiilit ovat tapa saada nopeasti kartta käyttäjälle. Karttatiili on kuvatiedosto (tyypillisesti 256x256 pikseliä). Kartta on jaettu neliön muotoisiin kuviin (Kuvio 23), joita haetaan käyttäjän näytölle aina sen mukaan, missä käyttäjän karttanäkymä on. Kukin karttatiili haetaan kolmella muuttujalla: Zoomaustaso, x-koordinaatti ja y-koordinaatti. Karttatiileissä ensimmäinen zoomaustaso on yksi karttaneliö, joka peittää koko kartan. Seuraavassa zoomaustasossa kukin karttaneliö jaetaan neljään karttaneliöön. Karttajako (Kuvio 23) on siis zoomaustaso kolme. X- ja y-koordinaateilla haetaan zoomaustason karttatiili kuvan mukaisesti. Esimerkkinä koordinaatit zoom=3, x=2, y=1 palauttaa karttatiilin, johon kuuluu etelä-Suomi ja lähi-itä. (Google Developers 2019b)



Kuvio 23. Karttatiilikoordinaatit (Google Developers 2019a)

Android-alustassa on mahdollisuus käyttää muitakin taustakarttoja, kuin Googlen omia karttoja. Android -alustan omassa karttarajapinnassa luodaan uusi TileProvider-luokka, joka annetaan karttaluokalle. Annettu TileProvider-luokan metodi getTile() ajetaan aina, kun kartta hakee uutta karttatiiliä. Metodin parametit ovat zoomaustaso x- ja y-koordinaatit. Metodin tulee palauttaa Tile-luokka, joka sisältää karttatiilikuvan bitteinä. TileProvider-luokan tekeminen itse mahdollistaa oman taustakartan tiedostoista tai verkosta. (Google Developers 2019b)

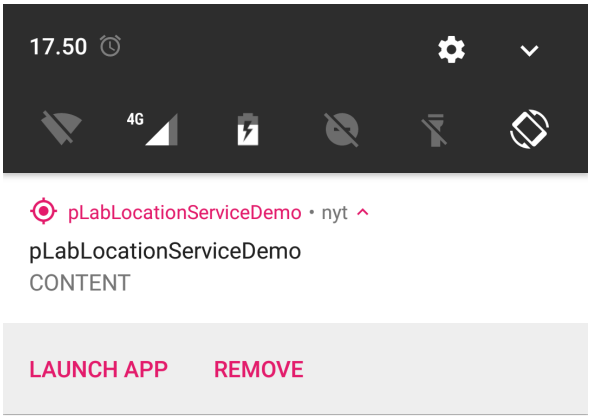
4.3 Sijaintipalvelu

Android-alustan palvelu (Service) on ohjelman komponentti, joka voi suorittaa pitkäaikaista toimintoa taustalla ilman käyttöliittymää. Palvelu voi toimia taustalla, vaikka varsinainen Activity, josta se on käynnistetty, on pysäytettynä. (Android Developers 2019j)

Palveluita Android-alustassa on kolmenlaisia: Taustapapalvelu (Background service), etualapalvelu (Foreground service) ja sidottu palvelu (Bound Service). Taustapalvelu toimii käyttäjältä näkymättömissä ja sen käynnistäneen Activityn ulottumattomissa. (Android Developers 2019j)

Android versio 8 toi suuremmat rajoitukset taustapalveluille. Esimerkiksi versiosta kahdeksan eteenpäin Android-alusta ei salli taustalla olevan ohjelman luoda taustapalvelua. Ohjelman on mahdollista luoda etualapalvelu viiden sekunnin sisällä ohjelman siirryttyä taustalle. (Android Developers 2018b)

Etualapalvelu näkyy käyttäjälle ilmoituskentässä (Kuvio 24), kuten esimerkiksi musiikinsoittopalvelu. Etualapalvelu jatkaa toimintaansa, vaikka ohjelma on taustalla. Sidottu palvelu on taas täysin sen käynnistäneen Activityn ulottuvissa, eli Activitystä käsin voidaan ajaa sidotun taustapalvelun metodeita. (Android Developers 2019j)



Kuvio 24. Etualalla toimiva sijaintipalvelu näkyy ilmoituskentässä

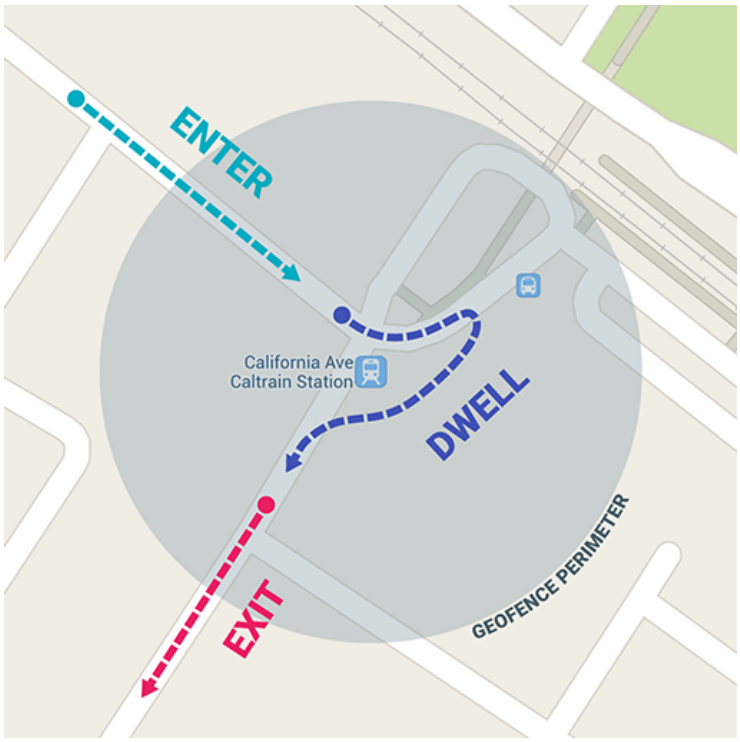
Sijaintipalvelun tapauksessa, palvelu on yleensä tarpeen toteuttaa etualapalveluna. Pelkkänä taustapalveluna käyttöjärjestelmä, Android versiosta kahdeksan eteenpäin, antaa sijainnin vain muutaman kerran tunnissa (Android Developers 2019f). Tyypillisesti sijaintia tarvitaan useammin. Tällöin palvelu tulee toteuttaa etualalla. Etualapalvelun sallitaan toimia vapaammin. Android versiosta kahdeksan eteenpäin alusta on harventanut sijainnin saatavuutta ilman etualalla olevaa palvelua, koska tiheä GNSS-mittaustahti kuluttaa paljon puhelimen akkua. Samasta syystä on ohjelmistokehittäjän syytä käyttää tiheämpää paikannustahtia vain tarvittaessa. (Android Developers 2019i)

Palvelun voi myös toteuttaa niin, että Activityn ollessa aktiivisena, palvelu on sidottu (Bound). Tällöin esimerkiksi kartta voi suoraan palvelulta hakea viimeisimmän sijainnin ja käyttää muita mahdollisia palvelun metodeja. Kun Activity menee taustalle, voidaan palvelu muuttaa etualapalveluksi. (Android Developers 2019j)

4.4 Virtuaaliaidat

Virtuaaliaita (Geofence) on paikkatiedon käsittelytapa, jossa jokin rajattu alue käynnistää tapahtuman. Kun käyttäjän laite GNSS-paikantimen mukaan saapuu

alueen sisään, virtuaaliaitapalvelu ilmoittaa alueen sisään saapumisesta. Jos käyttäjä on alueella tietyn ajan, palvelu ilmoittaa käyttäjän oleskelusta alueen sisällä. Kun käyttäjä poistuu alueelta, palvelu ilmoittaa poistumisesta (Kuvio 25).



Kuvio 25. virtuaaliaidan konsepti (Android Developers 2019d)

Android-alusta tarjoaa valmiin virtuaaliaitarajapinnan. Valmiin virtuaaliaitapalvelun käyttäminen on suotavaa sille sopivissa tilanteissa. Suositeltava minimikoko parhaan tuloksen saavuttamiseksi virtuaaliaidan säteelle on 150-1000 metriä. Tämä johtuu GNSS-paikannuksen tarkkuudesta. (Android Developers 2019d)

Rajoituksena alustan omalla virtuaaliaitapalvelulla on sen mahdollisuus olla huomaamatta virtuaaliaitaa, jos on tarve pienemmälle virtuaaliaidalle kuin suositeltava koko. Syynä on palvelun tarkoituksellisen hidas reaktioaika. Android 8 -versiosta eteenpäin virtuaaliaidan keskimääräinen reaktioaika on muutaman minuutin välein. Tällöin on mahdollista, että pienempi virtuaaliaita ei käynnisty lainkaan, tai käyttäjä joutuu odottamaan paikoillaan minuutteja virtuaaliaidan käynnistymistä. (Android Developers 2019c)

Herkkää reaktioaikaa tarvitsevissa sovelluksissa voi olla tarpeen kehittää itse oma virtuaaliaitapalvelu (Kuvio 26). Android-alustalla on mahdollista saada käyttäjän sijainti muutaman sekunnin välein. Tätä ominaisuutta hyväksikäyttäen on

mahdollista toteuttaa alustan omaa virtuaaliaitapalvelua herkempi palvelu. Olennaisin ominaisuus on ratkaista tapa, kuinka havaita muutostapahtuma.

Alla olevassa ratkaisussa (Kuvio 26) määritetään, onko käyttäjä virtuaaliaidan sisällä. Tämä tehdään laskemalla viimeaikaisten sijaintien keskimääräinen etäisyys virtuaaliaidan keskipisteeseen. Jos keskimääräinen etäisyys on pienempi kuin virtuaaliaidan säde, on käyttäjä alueen sisällä. Jotta virtuaaliaita ei ilmoittaisi jatkuvasti sisällä ja ulkona olemisesta alueen rajalla, algoritmissa on ominaisuus, jossa käyttäjä ilmoitetaan olevan yhä alueen sisällä, jos edeltävä käyttäjän status oli olla aidan sisällä ja keskimääräisen etäisyys keskipisteestä on pienempi kuin virtuaaliaidan säde lisättynä marginaalilla. Ominaisuuden tarkoituksena on, että muutos sisällä olemisesta ulkona olemiseen vaatii kauempana olemista kuin ulkoa sisälle -muutos.

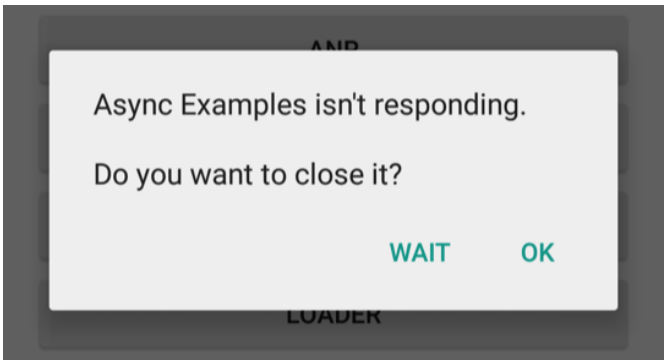
```
/**
 * Algorithm to decide, if current location is inside geofence
 *
 * Currently calculates mean distance in last PREVIOUS_LOCATION_COUNT locations
 * If mean distance is less or equal to radius, user is inside geofence
 *
 * @param fencePoint
 * @param radius
 * @return
 */
private boolean insideGeoFence(LatLng fencePoint, Float radius, boolean previousInside) {
    boolean inside = false;

    // Convert LatLng to Location for distance calculation
    Location center = new Location( provider: "");
    center.setLatitude(fencePoint.latitude);
    center.setLongitude(fencePoint.longitude);

    // Distance Sum init
    float distanceSum = 0;

    // Add disances from fence to previous and current location to sum
    for (Location location : previousLocations) {
        float distance = center.distanceTo(location);
        distanceSum += distance;
    }

    // Divice sum with locations array size
    float mean = distanceSum / previousLocations.size();

    // If mean is less or equal radius, user is inside geofence
    if (mean <= radius) {
        inside = true;
    }

    // If previously inside fence, add margin to geofence radius.
    // Prevent exit, enter repeating when in geofence border area
    if (previousInside && !inside) {
        if (mean <= radius + EXIT_EVENT_MARGIN) {
            Dev.geom(sender,  message: "Inside fence exit margin (radius: "
                    + String.valueOf(radius) + "). mean: "
                    + String.valueOf(mean),  isError: false);
            inside = true;
        }
    }
    return inside;
}
```

Kuvio 26. Algoritmiesimerkki itse toteutetusta virtuaaliaidan metodista, joka kertoo, onko käyttäjä virtuaaliaidan sisällä

4.5 Muistivuotojen välttäminen

Muistivuoto tarkoittaa tilannetta, jossa jokin luokka jää muistiin, vaikka sitä ei enää tarvita. Roskankeräin (Carbage Collection) pyrkii vapauttamaan muistia poistamalla luokat, joihin ei ole viittausta. Kuitenkin, jos viittaus on, Roskankeräin ei voi vapauttaa luokan kohdalla muistia. Tällaisia tilanteita voi syntyä Android-ympäristössä useissa tilanteissa. Muutama pieni muistivuoto ei välttämättä haittaa ohjelman toimintaa, mutta suuremmassa ohjelmassa tai muistia vaativassa

tilanteessa voi seurata ohjelman kaatuminen. Muistivuoto voi myös johtaa ohjelman jumittumiseen ja siitä seuraavaan virheilmoitukseen (Kuvio 27). (Murthy 2018)



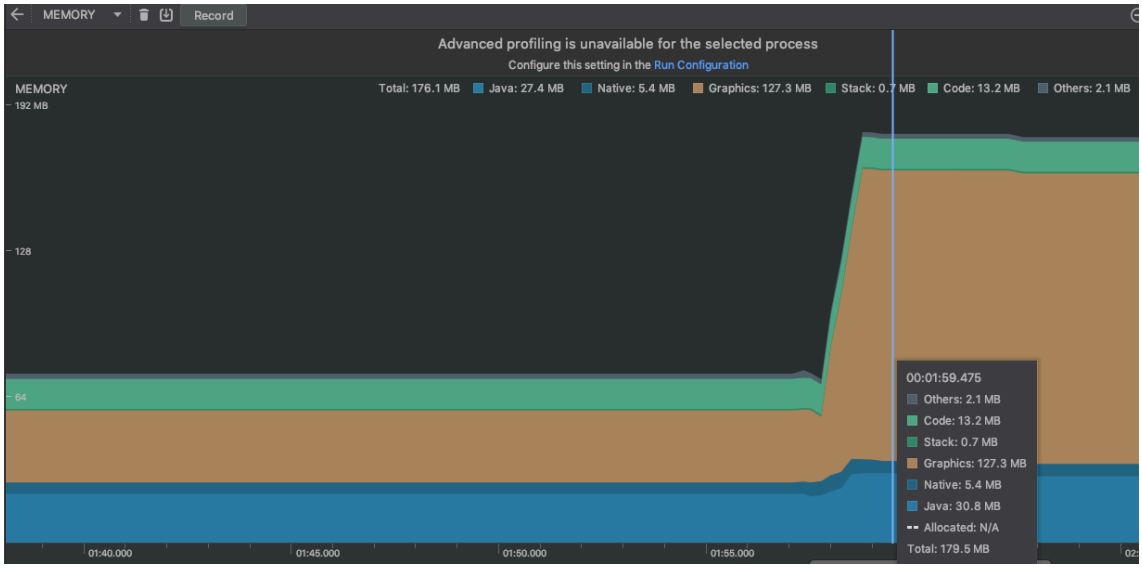
Kuvio 27. Android-alustan virheilmoitus (Tan 2017)

Muistivuotoja voi torjua usealla eri toimenpiteellä. Android-alustalla toimenpiteet ovat pääosin alustariippuvaisia. Alla toimenpiteitä, joilla voi torjua muistivuotoja.

- Muistivuotojen tarkistaminen kolmannen osapuolen ohjelmalla, kuten LeakCanarylla.
- Jos rekisteröi Activity-luokan kuuntelijaksi Receiver-luokalle, on muistettava aina Activityn elinkaaren lopuksi poistaa Activity kuuntelijoista (unregister). Tällöin Activityyn ei jää sen sulkemisen jälkeen viittausta ja roskankeräin voi poistaa sen.
- Activity, View ja Context -luokista ei saa tehdä staattisia.
- Singleton-luokalle, joka tarvitsee Context-luokan, ei tule antaa viittausta Activityyn, vaan esimerkiksi applicationContext. Jos Activity tarvitaan, viittaus tulee poistaa Singletonissa Activityn lopettaessa.
- Luokan sisäisestä luokasta tulee tehdä staattinen. Staattinen sisäluokka ei säilytä viittausta emoluokkaan.
- Asynkroninen AsyncTask-luokka Activity-luokan sisällä tulee muistaa lopettaa Activity-luokkaa lopettaessa, jotta se ei jää taustalle. Sama pätee myös Thread-luokkiin ja muihin asynkronisiin luokkiin.

- Weakreference-luokan käyttäminen viittauksissa luo heikon viittauksen. Weakreferencen kautta viitatessa roskankeräin poistaa luokan muistista, jos jäljellä on vain weakreference-viittauksia. (Murthy 2018)

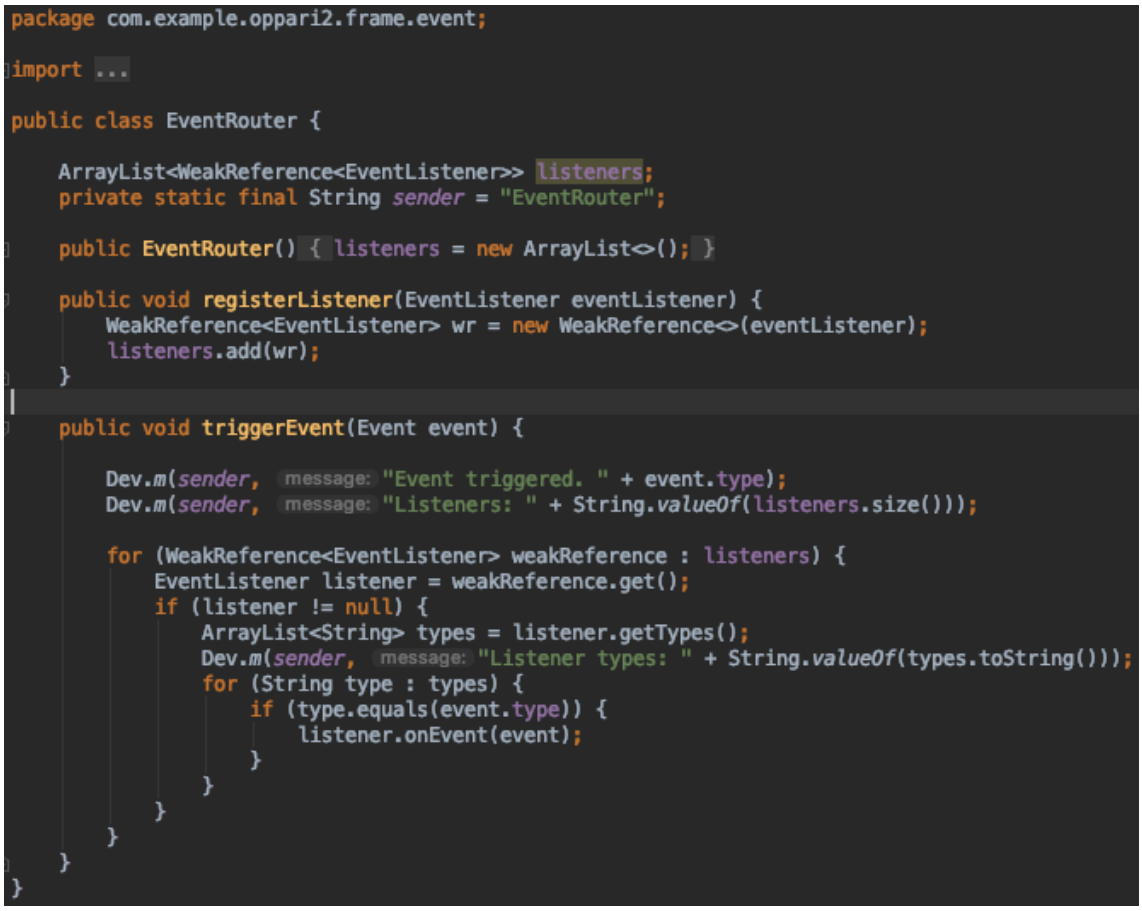
Erityisesti karttaohjelmassa muistivuodot voivat aiheuttaa ongelmia, koska kuvat vievät paljon muistia ja karttanäkymä käyttää paljon muistia esittäessään paljon karttatiilikuvia. Alla olevassa kuviossa näkyy Android Profiler -muistinkäytön seurantatyökalulla muistinkäytön kasvaminen, kun siirrytään karttanäkymään (Kuvio 28). Kuviossa vaaka-akseli on kulunut aika ja pystyakseli on muistin käyttö. Vaaka-akselissa aika etenee siten, että oikea reuna on nykyinen hetki ja etäisyys oikeasta reunasta kertoo, kuinka kauan on tästä hetkestä. Muistivuoto paljon muistia käytävässä ohjelmassa voi helpommin johtaa ohjelman kaatumiseen.



Kuvio 28. Android Profiler -muistinseurantatyökalussa tapahtuva ohjelman muistin käytön muutos siirryttäessä yksinkertaisesta Activitystä kartta-Activityyn

4.5.1 Heikko viittaus

Java-ohjelmointikielessä on neljä tapaa viitata. Strong, soft, weak ja phantom. Opinnäyteyön arkkitehtuuritoteutuksessa on käytössä vain kahta tyyppiä: Strong ja weak. Vahva (Strong) viittaus on normaali viittaus, jossa luokka asetetaan muuttujaksi. Heikko (weak) viittauksessa käytetään WeakReference-luokkaa viittauksen rakentamiseen (Kuvio 29). (Mañas 2016)



```
package com.example.oppari2.frame.event;

import ...

public class EventRouter {

    ArrayList<WeakReference<EventListener>> listeners;
    private static final String sender = "EventRouter";

    public EventRouter() { listeners = new ArrayList<>(); }

    public void registerListener(EventListener eventListener) {
        WeakReference<EventListener> wr = new WeakReference<>(eventListener);
        listeners.add(wr);
    }

    public void triggerEvent(Event event) {

        Dev.m(sender,  message: "Event triggered. " + event.type);
        Dev.m(sender,  message: "Listeners: " + String.valueOf(listeners.size()));

        for (WeakReference<EventListener> weakReference : listeners) {
            EventListener listener = weakReference.get();
            if (listener != null) {
                ArrayList<String> types = listener.getTypes();
                Dev.m(sender,  message: "Listener types: " + String.valueOf(types.toString()));
                for (String type : types) {
                    if (type.equals(event.type)) {
                        listener.onEvent(event);
                    }
                }
            }
        }
    }
}
```

Kuvio 29. Opinnäytetyössä toteutetun arkkitehtuurin EventRouter-luokka, jossa kuuntelijat tallennetaan WeakReference-luokan sisään listalle registerListener-metodissa

Arkkitehtuuritoteutuksessa muistivuotojen kannalta haastava luokka on EventRouter. EventRouter-luokalle pystyy rekisteröimään kuuntelijan, jonka kuuluu implementoida EventListener interface. Jos esimerkiksi Fragment implementoi kuuntelijan ja se on rekisteröity kuuntelijaksi EventRouterille, Fragment ei poistu sulkemisensa jälkeen muistista, koska viittaus siihen on säilynyt EventRouter-luokalla. Ratkaisuja muistivuotoon on kaksi: Ensimmäinen on poistaa viittaus aina, kun sitä ei enää tarvita (unregister). Toinen tapa on toteutettu luokassa, eli jokaiseen kuuntelijaan on heikko viittaus. Tämä on toteutettu listalla Weakreferenence<EventListener> luokkia. Suositeltavaa olisi käyttää molempia tapoja yhtä aikaa.

5 POHDINTA

Käyttöliittymän suunnittelu rautalankamallista Mockup-malliin luo ohjelmistokehittäjän työlle projektin aloittamisessa hyvän pohjan. Käyttöliittymäsuunnittelun merkitystä projektissa on vaikea yliarvioida. Ilman hyvää suunnittelua voi syntyä ohjelma, joka ei vastaa asiakkaan toiveisiin ja on käytettävyydeltään niin sekava, että ohjelman käyttöä ei koeta mielekkääksi. Toisaalta taas hyvä käyttöliittymäsuunnittelu pohjatyönä on ohjelmistokehittäjälle palkitsevaa, kun työaikaa ei kulu liikaa muutoksiin, joita seuraa, kun ensimmäinen versio päätyy asiakkaan käsiin. Käyttöliittymäsuunnittelu rautalankamallilla on yksinkertainen, halpa ja tehokas tapa, joka on lähes poikkeuksetta kannattavaa tehdä. Käyttöliittymäsuunnittelusta on paljon teoriaa ja kirjallisuutta tarjolla, jota pätevät vuosienkin jälkeen peruskohdiltaan.

Opinnäytetyössä lähdettiin pohtimaan sekä kokeilemaan mobiilisovelluksen arkkitehtuuria. Teoria muuntautui käytäntöön tavalla, joka puhuu tekijänsä ajattelutavasta enemmän, kuin lopullisesta totuudesta arkkitehtuurin toteuttamisessa. Toteutettu kokeiluarkkitehtuuri havaittiin käytännössä puutteelliseksi monelta osin, vaikka useat ratkaisut olivat periaatteessa oikeansuuntaisia. Kuitenkin näitä puutteita ei ilman kokeilemista olisi havaittu. Sopivimman arkkitehtuurin löytyminen Android-alustalle läheni usealla oikeansuuntaisella askeleella ja muutamat askeleet tunnistettiin harha-askeleiksi. Kokeileminen kannatti suuresti, mutta antoi vasta suunnan, oikealle tielle kohti projektissa sovellettavaa arkkitehtuuria.

Ratkaisut Android-alustan osa-alueisiin ovat kokemusperäistä osaamista ja yksi tapa toimia. Erityisesti paikkatiedon soveltamisen ratkaisut käytännössä olivat painopisteenä opinnäytetyössä. Näiltä osin tavoite oli kuvata yleisellä tasolla toimiviksi havaittuja ratkaisuja. Ratkaisujen toimivuus perustuu kokemuksiin. Kokemuksia pyrittiin kuvamaan ja perustelemaan ratkaisuja niiden kautta. Ratkaisujen kuvaaminen opinnäytetyössä mahdollisesti ohjaa samojen ongelmia pohtivat ohjelmistokehittäjät lähemmäksi ratkaisua. Paikkatiedon osalta toimivat ratkaisut esimerkiksi sijaintipalvelun osalta ovat tavallista arvokkaampia, koska sijaintipalveluiden maastotestaus on hidasta ja siten kallista. Käytännössä sijainnin muutosten simulointi toimisto-olosuhteissa ei perustoteutuksellaan tuo esille kaikkia

tarvittavia tilanteita, joita syntyy muun muassa erilaisissa liikkumisnopeuksissa, syrjäseuduilla tai erilaisella laitekannalla.

Projektin kannalta opinnäytetyö kerrytti arvokasta tietoa arkkitehtuurin kehittämisestä Android-alustalla. Oman ohjelmistorakenteen toteuttamisen jälkeen osaa esittää tarkemmin kysymykset, joihin ohjelmistoarkkitehtuurin tulisi vastata. Opinnäytetyön valmistamisen aikana tutustuttiin myös Android-alustalle tulleisiin, uusiin ohjelmistoarkkitehtuurikomponentteihin. Nämä komponentit vaikuttivat vastauksilta Android-alustan aukkoihin ohjelmistoarkkitehtuurin saralla.

Opinnäytetyössä ei edetty uusien komponenttien osalta pintaa syvemmälle. Aikataulu ja aiheen rajaus pysäyttivät tältä osin. Uusien Android-alustan arkkitehtuurikomponenttien tarkempi testaus ja käytön kuvaaminen olisi toisen opinnäytetyön aihe. Projektin kohdalla opinnäytetyön loppupäätelmät otettiin käyttöön ja projektin kehittäminen jatkuu uusilla komponenteilla silläkin riskillä, että uudet ja vielä suhteellisen tuntemattomat komponentit tuovat omia haasteita. Kuitenkin Android-alustassa monet ohjelmistokehittämisen hitaudet kumpuavat toimivan arkkitehtuurin puuttumisesta ja siksi riski on todennäköisesti kannattava ottaa.

LÄHTEET

Ahtinen, A. & Kokkonen, A. 2000. Käytettävyyden arviointi. Viitattu 20.3.2019 http://www.sis.uta.fi/ipopp/ipopp2000/AhtinenKokkonen/kaytettavyys_2.html

Android Developers 2018a. Activities. Viitattu 16.12.2018 https://developer.android.com/guide/components/activities/.

Android Developers 2018b. Background Execution Limits. Viitattu 15.12.2018 https://developer.android.com/about/versions/oreo/background.

Android Developers 2018c. Controls and Gestures. Viitattu 15.12.2018 https://developers.google.com/maps/documentation/android-sdk/controls.

Android Developers 2018d. Fragments. Viitattu 16.12.2018 https://developer.android.com/guide/components/fragments.

Android Developers 2018e. Map Objects. Viitattu 15.12.2018 https://developers.google.com/maps/documentation/android-sdk/map.

Android Developers 2018f. Save data using SQLite. Viitattu 15.12.2018 https://developer.android.com/training/data-storage/sqlite.

Android Developers 2018g. Sensors Overview. Viitattu 15.12.2018 https://developer.android.com/guide/topics/sensors/sensors_overview.

Android Developers 2018h. SharedPreferences. Viitattu 15.12.2018 https://developer.android.com/reference/android/content/SharedPreferences.

Android Developers 2018i. Understand the Activity Lifecycle. Viitattu 30.11.2018 https://developer.android.com/guide/components/activities/activity-lifecycle.

Android Developers 2019a. Activity. Viitattu 20.3.2019 https://developer.android.com/reference/android/app/Activity

Android Developers 2019b. Android Jetpack. Viitattu 20.3.2019 https://developer.android.com/jetpack

Android Developers 2019c. Background Location Limits. Viitattu 20.3.2019 https://developer.android.com/about/versions/oreo/background-location-limits

Android Developers 2019d. Create and monitor geofences. Viitattu 20.3.2019 https://developer.android.com/training/location/geofencing

Android Developers 2019e. Get started with the Navigation component. Viitattu 20.3.2019 https://developer.android.com/topic/libraries/architecture/navigation/navigation-implementing

Android Developers 2019f. Get the last known location. Viitattu 20.3.2019 https://developer.android.com/training/location/retrieve-current

Android Developers 2019g. Guide to app architecture. Viitattu 20.3.2019 https://developer.android.com/jetpack/docs/guide

Android Developers 2019h. Navigation. Viitattu 20.3.2019 https://developer.android.com/topic/libraries/architecture/navigation.html

Android Developers 2019i. Optimize location for battery. Viitattu 20.3.2019 https://developer.android.com/guide/topics/location/battery

Android Developers 2019j. Services overview. Viitattu 20.3.2019 https://developer.android.com/guide/components/services

Android Developers 2019k. ViewModel Overview. Viitattu 20.3.2019 https://developer.android.com/topic/libraries/architecture/viewmodel

Arya, R. 2017. How we reduced our Android app's memory footprint by 50%. Viitattu 15.12.2018 https://medium.freecodecamp.org/how-we-reduced-memory-footprint-by-50-in-our-android-app-49efa5c93ad8.

Barnard, L. 2016. Wireframing for beginners. Viitattu 20.3.2019 https://uxmastery.com/wireframing-for-beginners/

Callaham, J. 2018. The history of Android OS: its name, origin and more. Viitattu 20.3.2019 https://www.androidauthority.com/history-android-os-name-789433/

Clark, L. 2015. A cartoon guide to Flux. Viitattu 20.3.2019 https://code-cartoons.com/a-cartoon-guide-to-flux-6157355ab207

Egorina, A. 2017. Accessibility: what you need to know about mobile app development for people with special needs. Viitattu 20.3.2019 https://medium.com/rosberryapps/accessibility-what-you-need-to-know-about-mobile-app-development-for-people-with-special-needs-b52d713f8c95

Facebook 2019. Flux - In Depth Overview. Viitattu 20.3.2019 https://facebook.github.io/flux/docs/in-depth-overview.html

Google Developers 2019a. Map and Tile Coordinates. Viitattu 20.3.2019 https://developers.google.com/maps/documentation/javascript/coordinates

Google Developers 2019b. Tile Overlays. Viitattu 20.3.2019 https://developers.google.com/maps/documentation/android-sdk/tileoverlay

Hinchman, A. 2018. The SingleFragmentActivity Pattern in Android & Kotlin. Viitattu 20.3.2019 https://medium.com/@hinchman_amanda/the-singlefragmentactivity-pattern-in-android-kotlin-ce93385252e5

Kononenko, K. 2016. Model-View-Controller (MVC) Explained Through Ordering Drinks At The Bar. Viitattu 20.3.2019 https://medium.freecodecamp.org/model-view-controller-mvc-explained-through-ordering-drinks-at-the-bar-efcba6255053

Krug, S. 2014. Don't make me think, revisited: A common sense approach to web usability (Third edition.). [San Francisco, CA]: New Riders.

Laine, A. 2004. Hahmolait käytettävyyden parantajina. Viitattu 20.3.2019 http://www.mit.jyu.fi/opetus/opinnayte/LuK/Hahmolait/

Martonik, A. 2018. Google I/O 2018. Viitattu 20.3.2019 https://www.androidcentral.com/google-io-2018/home

Material design 2019. Accessibility. Viitattu 20.3.2019 https://material.io/design/usability/accessibility.html

Mañas, E. 2016. Finally understanding how references work in Android and Java. Viitattu 20.3.2019 https://medium.com/google-developer-experts/finally-understanding-how-references-work-in-android-and-java-26a0d9c92f83

Mkrtchyan, R. 2018. Wireframe, Mockup, Prototype: What is What?. Viitattu 20.3.2019 https://uxplanet.org/wireframe-mockup-prototype-what-is-what-8cf2966e5a8b

Muntenescu, F. 2016. Android Architecture Patterns Part 1: Model-View-Controller. Viitattu 20.3.2019 https://medium.com/upday-devs/android-architecture-patterns-part-1-model-view-controller-3baecef5f2b6

Murthy, A. 2018. 9 ways to avoid memory leaks in Android. Viitattu 20.3.2019 https://android.jlelse.eu/9-ways-to-avoid-memory-leaks-in-android-b6d81648e35e

Nielsen, J. 1994. 10 Usability Heuristics for User Interface Design. Viitattu 20.3.2019 https://www.nngroup.com/articles/ten-usability-heuristics/

Nivala, A. 2007. Usability Perspectives for the Design of interactive maps. Espoo: Suomen geodeettinen laitos. Viitattu 15.12.2018 https://aalto-doc.aalto.fi/bitstream/handle/123456789/2960/isbn9789512289431.pdf?sequence=1&isAllowed=y.

Plesac, Ž. 2015. The past, present and future of Android development. Viitattu 20.3.2019 https://infinum.co/the-capsized-eight/the-past-present-and-future-of-android-development

Prasad, J. 2018. JET, set, PACK, go! Guide to using Android Jetpack Part 1: Overview and Lifecycle-aware Components with code tutorials. Viitattu 20.3.2019 https://android.jlelse.eu/jet-set-pack-go-guide-to-using-android-jetpack-378cf248be00

Rabetckiy, D. 2019. A Single-Activity Android Application. Why not?!. Viitattu 20.3.2019 https://medium.com/rosberryapps/a-single-activity-android-application-why-not-fa2a5458a099

Smith, A. 2016. Less Is Still More: The Importance Of The Minimalist Approach To Web Design. Viitattu 20.3.2019 https://usabilitygeek.com/less-is-more-importance-minimalist-web-design/

Soini, J. 2016. Web-sovellusten käytettävyyden arviointi heuristiikkojen avulla. Oulun Yliopisto. Viitattu 20.3.2019 http://jultika.oulu.fi/files/nbnfioulu-201612103257.pdf

Space-O Technologies 2019. Why Do You Need to Choose MVP Over MVC Android Architectural Pattern?. Viitattu 20.3.2019 https://www.spaceotechnologies.com/mvp-android-architectural-pattern/

Suomisanakirja 2018. Heuristinen. Viitattu 20.3.2019 https://www.suomisanakirja.fi/heuristinen

Tan, F. 2017. Memory Leak Patterns in Android. Viitattu 20.3.2019 https://android.jlelse.eu/memory-leak-patterns-in-android-4741a7fcb570

Wood, D. 2014. Interface design: An introduction to visual communication in UI design. London: Fairchild Books.

Yalanska, M. 2019. Lean and Mean: Power of Minimalism in UI Design. Viitattu 20.3.2019 https://tubikstudio.com/lean-and-mean-power-of-minimalism-in-ui-design/

Zhao, J. 2017. Android Fragment Back Stack Example. Viitattu 20.3.2019 https://www.dev2qa.com/android-fragment-back-stack-example/