TARTU ÜLIKOOL
Arvutiteaduse instituut
Informaatika õppekava

Kristofer Käosaar

Jahitegevust toetava Android-rakenduse moderniseerimine Kotlinis

Bakalaureusetöö (9 EAP)

Juhendaja: Jakob Mass, MSc

Tartu 2021

Jahitegevust toetava Android-rakenduse moderniseerimine Kotlinis

Lühikokkuvõte:

Käesoleva bakalaureusetöö eesmärk on luua uus versioon Traxby OÜ jahipidamise rakendusest Huntloc, mis oleks modernne ja lihtsasti hooldatav. Analüüsitakse Kotlini eeliseid üle Java Androidi rakendust arendades. Töö sisaldab olemasoleva rakenduse analüüsi, Java ja Kotlini võrdlust ja rakenduse arhitektuurse mustri valimise väljaselgitamist. Valminud rakendust võrdleti olemasoleva lahendusega, et selgitada välja, kas täideti töö eesmärk. Selleks analüüsiti mõlemasse rakendusse samade funktsionaalsuste lisamist. Toodi välja lisatud või muudetud klasside, meetodite ja koodi ridade arvud ning arendaja kogemuse kirjeldus ja arendamisele kulunud aeg.

Võtmesõnad:
Traxby OÜ, mobiilirakendus, Java, Kotlin, Android, MVVM

CERCS: P175 Informaatika

Modernizing an hunting oriented Android application in Kotlin

Abstract:

The aim of this bachelor's thesis is to create a new and modern version of an existing hunting oriented Android application for the company Traxby OÜ and to analyze the benefits of using Kotlin over Java in Android development. The thesis contains an analysis of the existing solution, differences between Java and Kotlin, and the difference between adding new functionalities to both the old and new version of the application. Comparison of adding new functionalities was done by measuring development time, lines of code, methods and classes changed or added and taking note of the developer's personal experience.

Keywords:
Traxby OÜ, mobile application, Java, Kotlin, Android, MVVM

CERCS: P175 Informatics

Sisukord

1. Sissejuhatus

Android on Linuxil põhinev operatsioonisüsteem, mis on suunatud puutetundlikutele seadmetele, nagu telefonid, tahvelarvutid ja ka mõned sülearvutid. 2011. aastal sai sellest nutitelefonidel enim kasutatud operatsioonisüsteem ja aastaks 2021 oli Google Play Store'is üle 3 miljardi rakenduse [1,2]. Androidi peamine toetatud programmeerimiskeel on selle avalikustamisest saati olnud Java [3]. Kotlin kuulutati 2017. aastal Androidi ametlikuks programmeerimiskeeleks. Seda on kirjeldatud sisutiheda ja modernse keelena, mille kasutus võimaldab projekti koodiridu vähendada ja koodi kvaliteeti parandada [4].

Huntloc on jahitegevust toetav mobiilirakendus, mis väljastati esmalt 2014. aasta lõpus [5]. Sel ajal pakuti rakenduses teiste jahimeeste reaalajas jälgimist, kaardile kuvatavate objektide loomist ja jagamist ja jälgimisseadmetega ühildumist. Rakenduse eesmärk on teha jahipidamist ohutumaks ja ligipääsetavamaks noortele. Viimasel ajal olid arendajad märganud, et rakenduse hooldamist ja sellele uute funktsionaalsuste arendamist on aeglustanud projekti keeruline struktuur, komponendid mida enam ei toetata ja muud väikesed mured. Kasutuses oli 2017. aastal loodud koodibaas, mis vajas arendajate sõnul uuendamist. Käesoleva bakalaureusetöö eesmärgiks on need probleemid lahendada ja luua rakendusele tervenisti uus koodibaas, mida oleks kergem hooldada ja lihtsam täiendada. Selle tulemuseni jõudmiseks uuritakse, kas Kotlin võib olla Javast parem valik Androidi platvormil rakendust arendades.

Enne uue rakenduse arendamist analüüsitakse Androidi rakendustes kasutatavaid arhitektuurseid mustreid ja häid tavasid. Lisaks antakse ülevaade olemasoleva rakenduse komponentidest ja ülesehitusest, et selgitada välja probleemide allikad. Töö praktiline pool viidi läbi kahes osas. Esmalt arendati uus versioon olemasolevast Huntloci rakendusest. Seejärel võrreldi Huntloci olemasolevasse versiooni ja uude projekti funktsionaalsuste lisamise ajakulu, tekkinud koodi ridade arvu ja arendaja muljet arendamise kogemusest. Võrdlusest selgitatakse välja, kas uue rakenduse arendamine lahendas eelnevalt tõstatatud probleemid ja täitis töö eesmärgi.

Bakalaureusetöö koosneb viiest osast. Järgnevas peatükis antakse ülevaade ettevõtte ja rakenduse taustast, Java ja Kotlini erinevustest ja erinevatest Androidi platvormil kasutatavatest arhitektuursetest mustritest. Teises peatükis antakse ülevaade olemasoleva rakenduse ülesehitusest. Kolmandas peatükis kirjeldatakse uue rakenduse arendamist ning

neljandas peatükis võrreldakse ja analüüsitakse rakendustesse uute funktsionaalsuste lisamist. Lisana on esitatud Huntloci rakenduse funktsionaalsed nõuded (vt. Lisa I).

2. Mõisted ja terminid

Activity - Androidi raamistikus ekraan koos kasutajaliidesega ja elutsükliga, mis on võrreldav akna või raamiga Javas[1].

Fragment - Androidi raamistikus taaskasutatav osa kasutajaliidesest, mis omab enda elutsüklit ja peab eksisteerima *activity* või teise *fragment'i* sees[2].

Andmekiht (ingl *model*) - Osa rakendusest, mis vastutab andmete töötlemise, päringute tegemise ja andmete kohalikku andmebaasi salvestamise eest[3].

Esitluskiht (ing*l view*) - Osa rakendusest, mis vastutab kasutajaliidese loogika eest, tuginedes andmetele ja kasutajate interaktsioonidele.

Baitkood (ingl bytecode) - Kompileeritud klass, mis sisaldab koodi, mida oskab arvuti täita[4]. Baitkoodi on võimalik jooksutada virtuaalmasinal või edasi kompileerida masinkoodiks, mida oskab lugeda seadme protsessor.

Järkjärguline kompileermine (ingl *incremental compilation*) - Tavalise kompileerimise puhul tõlgitakse terve projekti kood baitkoodiks või masinkoodiks, et arvuti oskaks seda lugeda. Järkjärgulise kompileerimise puhul tehakse seda vaid failidega, mida on viimasest kompileerimisest saati muudetud[5].

JSON (ingl *JavaScript Object Notation***) -** Andmeedastuse formaat, mis hoiab tekstina endas organiseerituid andmeid.

Arhitektuurne muster *(ingl architectural pattern)* - Tarkvarasüsteemi struktuuriline korraldus[6].

---

[1] https://developer.android.com/reference/android/app/Activity
[2] https://developer.android.com/guide/fragments
[3] https://developer.android.com/jetpack/guide
[4] https://techterms.com/definition/bytecode
[5] https://kotlinlang.org/docs/gradle.html#incremental-compilation
[6] https://cs.nyu.edu/~jcf/classes/g22.2440-001_sp06/slides/session8/g22_2440_001_c82.pdf

Jetpack - Ametlik Androidi teekide komplekt, mille eesmärk on aidata arendajatel häid tavasid järgida[7]. See sisaldab teeke, mis teevad arendamise ühtlaseks üle kõikide Androidi versioonide.

REST (ingl Representational State Transfer) - Veebiteenuste programmeerimise paradigma, mis on üles ehitatud erinevatelt URI-delt päringute tegemisele[8].

[7] https://developer.android.com/jetpack/androidx/releases/room
[8] https://akit.cyber.ee/term/10843-rest

3. Taust

Selles peatükis kirjeldatakse nii ettevõtte tausta, kui ka Huntloci tarkvara. Antakse ülevaade Kotlini ja Java erinevustest, Androidi rakenduste arhitektuursetest mustritest ja rakenduse arendamise headest tavadest.

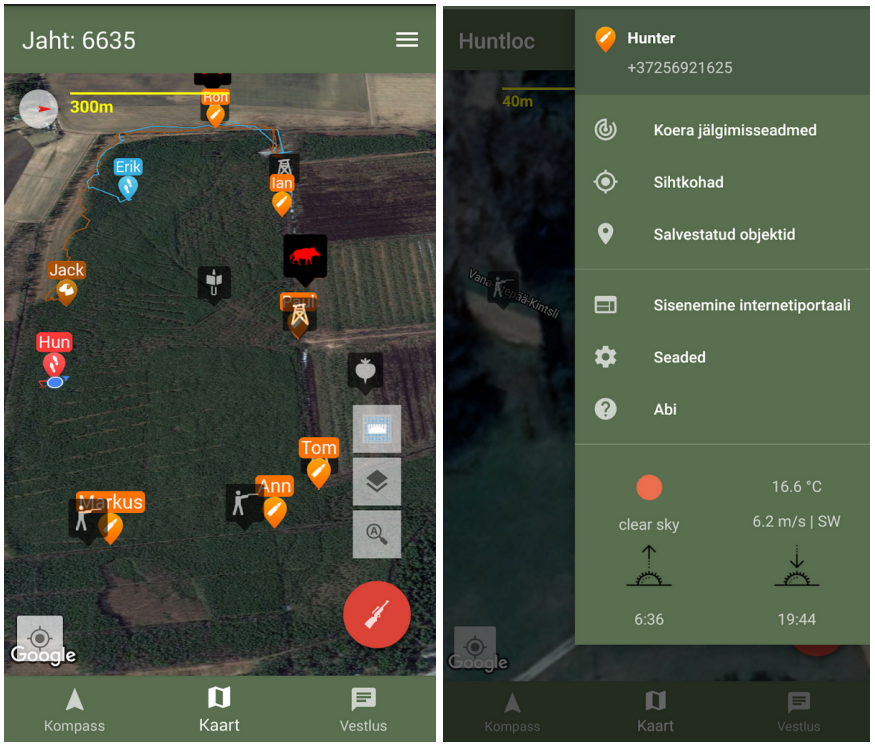
3.1 Ettevõte Traxby OÜ

Traxby OÜ (endiselt Huntloc OÜ) on 2013 aastal Eestis asutatud ettevõte, mis arendab jahimeestele suunatud tarkvara ja mobiilside tehnoloogial põhinevat jälgimisseadet, mis ühildub mobiilirakendusega [5]. Esimene versioon Huntloci rakendusest väljastati 2014. aasta oktoobris [6]. Sel ajal pakuti teiste jahimeeste reaalajas jälgimist, kaardil objektide loomist ja nende teistele kasutajatele jagamist, erinevate kaardikihtide kuvamist ja turul olevate mobiilside tehnoloogial põhinevate jälgimisseadmetega ühildumist. Rakenduse eesmärk oli teha jahipidamist ohutumaks ja noorematele jahihuvilistele ligipääsetavamaks. Peale algset edu keskendus ettevõte oma jälgimisseadmete arendamisele ja müümisele. Põhjuseks oli saada firma kontrolli alla seadmete tugi ja hooldus. Eelnevalt kasutati kõige rohkem Hiina tootja TKSTAR-i seadmeid, mille probleemide tekkimise korral pöördusid kliendid tihti Traxby OÜ poole. Peale seda, kui ettevõtte jälgimislahendus hakkas populaarsust koguma, eemaldati tugi teiste tootjate seadmetele.

Kasutajate baasi laiendamise eesmärgil arendati 2016. aastal Huntloci rakenduse põhjal Liveteam, mis on suunatud meeskondadele, kes soovivad reaalajas ülevaadet meeskonnaliikmete asukohtadest. 2020. aasta alguses avaldati ka Doglo, mis on koduloomade jälgimiseks mõeldud rakendus, milles puudus teistele rakendustele omane kasutajate asukohtade kuvamine ja sündmuse loomine. Doglo arendusprotsessi käigus ilmnesid ka mitmed järgmises alapeatükis kirjeldatud mured. Mobiilirakendusi arendati algselt Traxby OÜ arendusmeeskonna poolt, aga 2017. aastal palgati teine infotehnoloogia ettevõte nende uuendamiseks. Aastast 2019 oli ettevõte jälle rakendusi ise arendanud ja plaanis ka seda edaspidi teha.

3.2 Mobiilirakendus Huntloc

Rakenduse põhifunktsionaalsused on jahimaa-ala haldamine, jahi ajal teiste kasutajate kuvamine ja Traxby OÜ jälgimisseadmega ühildumine. Joonisel 1 on näidatud rakenduses toimuvat jahti, kus on kuvatud kaardi peale jahis osalevad kasutajad, koerad ja jagatud objektid. Kaardil kuvatud osaleja *Jack* on jälgimisseadet kandev koer. Lisaks on joonisel menüü, kus on näha nuppe, mis viivad rakenduse teistele ekraanidele. Läbi rakenduse on võimalik siseneda Huntloci veebiportaali, mida kirjeldatakse järgmises alapeatükis.



Joonis 1. Huntloci rakenduse kaardivaade ja menüü jahi ajal.

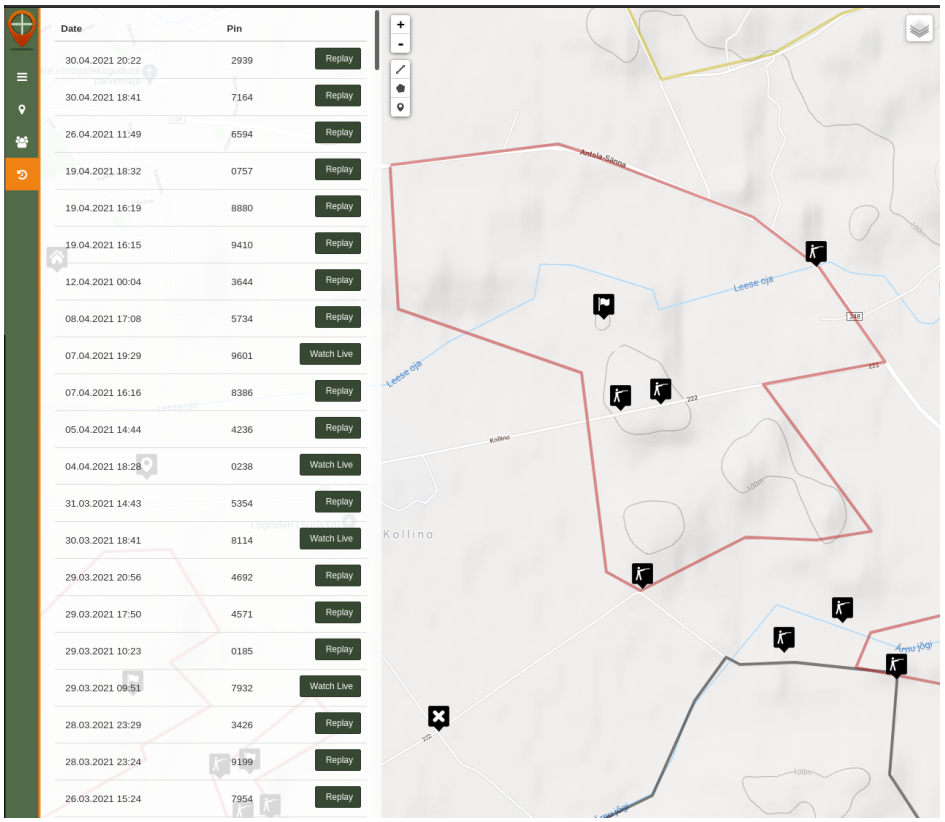
Esimene versioon rakendusest loodi aastal 2014 Traxby OÜ sisese meeskonna poolt Androidi platvormile. Sellest edasiarendatud versioon, koos ka iOS platvormi rakendusega, toodeti aastal 2017 tellimustööna infotehnoloogia ettevõtte FOB Solutions OÜ poolt.

Töös kirjeldatakse ja analüüsitakse täpsemalt Androidi rakenduse uusimat versiooni, seisuga 2020 sügis, ja sellest uue versiooni arendamist. Kirjutamise hetkel kasutas olemasolev lahendus arhitektuurset mustrit ja komponente, mis ei vastanud enam arendusnormidele ja tekitasid arendusprotsessis liigset ajakulu. Põhilised probleemid arendamisel ilmnesid uute funktsionaalsuste lisamisel või andmekihti puudutavate funktsionaalsuste muutmisel.

Rakenduse ülesehitust ja sellega seonduvaid probleeme kirjeldatakse täpsemalt töö neljandas peatükis. Funktsionaalsed nõuded on välja toodud lisas I. Kuna ka teised Traxby OÜ Androidi platvormi rakendused põhinesid Huntloci koodil, siis mõjutasid eelnevalt kirjeldatud mured ka neid. Eelnevatest faktoridest järeldati, et uue rakenduse arendamine oli pikas perspektiivis probleemidele parim lahendus.

3.2.1 Huntloci veebiportaal

Huntloci veebiportaalis on võimalik vaadata kordusi rakenduses toimunud jahtidest ja luua keerulisemaid objekte kaardile, mida kuvatakse peale loomist rakenduses. Joonisel 2 on näha nii nimekirja kasutaja toimunud jahisündmustest kui ka kaardile joonistatud maa-ala piiravat joont.



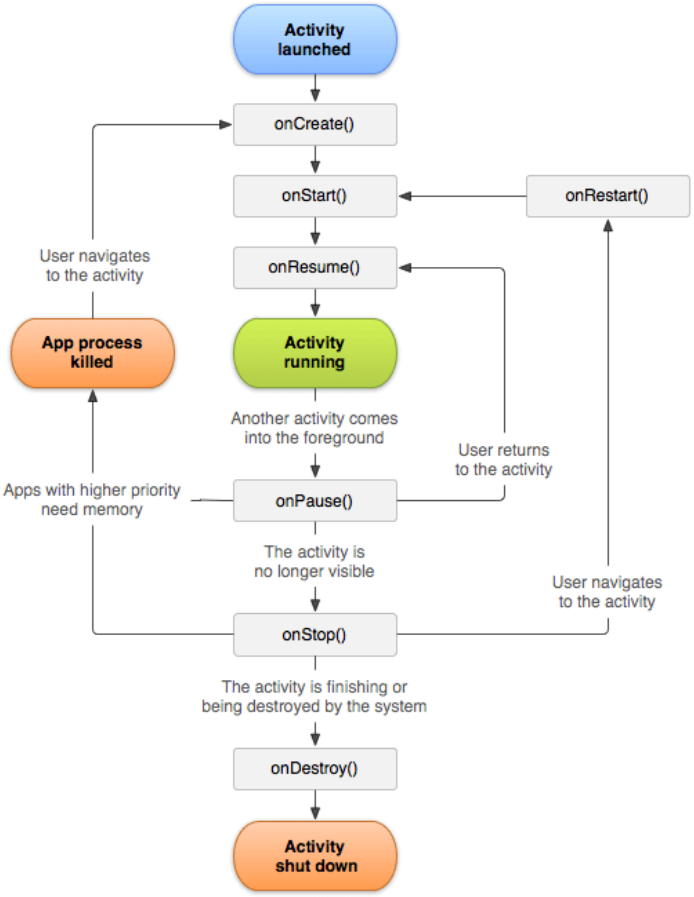
Joonis 2. Huntloci veebiportaali kaardivaade koos kasutaja eelnevate jahisündmuste nimekirjaga.

Portaalile oli enne töö kirjutamist omane meeskonna loomise funktsionaalsus, mis lisati bakalaureusetöö käigus ka Huntloci mobiilirakendusse. Aprill 2021 seisuga oli firmal plaanis meeskondadele funktsionaalsusi juurde lisada ja veebiportaali värskendada, sest sel hetkel oli liikmetel võimalik üksteisega ainult objekte jagada.

3.3 Androidi rakenduste põhikomponendid

Selles alapeatükis kirjeldatakse Androidi rakendustele omaseid komponente, mida mainitakse töös ka edaspidi. Refereeritud on Androidi dokumentatsiooni[9].

Activity[10] on Androidi rakenduse põhiline komponent, mida iseloomustab ekraan koos kasutajaliidesega. Rakenduse töötamiseks peab projektis olema vähemalt üks *activity,* sest see on telefoni jaoks sisenemispunkt rakendusse. Komponendid nagu *activity* omavad elutsüklit (ingl *lifecycle*)[11], mis on rakenduse olekute kogum. Elutsükkel eksisteerib alates komponendi käivitamisest kuni selle sulgemiseni ja muudab olekuid vastavalt kasutaja tegevusele. Ekraani loomise, taustale paneku ja sulgemise jaoks on eraldi olekud ja meetodid, mida süsteem *activity* klassilt välja kutsub (vt. joonis 3).



Joonis 3. Androidi Activity komponendi elutsükli diagramm[11]

[9] https://developer.android.com/docs
[10] https://developer.android.com/reference/android/app/Activity
[11] https://developer.android.com/guide/components/activities/activity-lifecycle

Fragment[12] on taaskasutatav osa rakenduse kasutajaliidesest. Sellel on enda elutsükkel, kuid peab eksisteerima kas *activity* või teise *fragment*'i sees. *Fragment* on suuteline võtma vastu kasutaja sisendit. Erinevalt *activity*'st, võib mitu erinevat *fragment*'i samal ajal töötada. Nendesse mõlemasse luuakse kasutajatele kuvatav pilt kasutades vaateid. Vaade[13] (ingl *view*) on Androidi rakenduse kasutajaliidese põhiline ehitusplokk. See hõlmab ekraanil ristkülikukujulist ala ja vastutab sündmuste haldamise eest. Kõik rakenduse graafilise kasutajaliidese elemendid on vaate alamklassid.

3.4. Java ja Kotlin

Enne uue rakenduse arendamist oli vaja otsustada, kas kirjutada rakendus Javas või kasutada Kotlinit. Selles alapeatükis antakse ülevaade mõlema programmeerimiskeele erisustest ja eelistest nii üldiselt kui ka Androidi platvormi kontekstis.

Java on 1995. aastal avaldatud objektorienteeritud klassipõhine programmeerimiskeel, mida kasutatakse mitmetel platvormidel [7]. Kirjutamise seisuga oli see PYPL indeksi[14] järgi maailmas populaarsuselt teine programmeerimiskeel. Java on saanud laialdaselt kasutatuks turvalisuse, platvormist sõltumatuse ja suure jõudluse poolest [8]. Oma vanuse ja suure kasutajate hulga tõttu on uutel programmeerijatel keelt õppides lihtne leida õppematerjale ja näiteid. Java negatiivsete küljena on tihti välja toodud vaikimisi määratud null-väärtused, mis tekitavad käitusajas null-erindeid [8] [9].

Crisan [10] kirjutab, et Kotlin on 2016. aastal avaldatud programmeerimiskeel, mille looja on ettevõte JetBrains, kes on ka loonud IntelliJ integreeritud arenduskeskkonna. See on modernne ja avatud lähtekoodiga programmeerimiskeel, mis kombineerib nii objektorienteeritud programmeerimise kui ka funktsionaalse programmeerimise tunnusjooni. Kotlin on lihtsuse ja Javaga ühilduvuse tõttu mobiilirakenduste arendajate seas populaarsust kogunud [11].

3.4.1 Võrdlus

Kotlinit on tihtipeale kiidetud lühikese ja sisutiheda koodi tõttu, mis on eriti tähtis suuremate projektide puhul [11][12]. Java klassidele isikupärased *get*-ja *set*- meetodid ja konstruktor

[12] https://developer.android.com/guide/fragments
[13] https://developer.android.com/reference/android/view/View
[14] https://pypl.github.io/PYPL.html

genereeritakse Kotlini andmeklassi loomisel automaatselt. Javas uue klassi loomine on näidatud joonisel 4. Kotlinis tagatakse sama funktsionaalsus palju lühema koodiga (vt. joonis 5).

```
class Dog {

    private String name;
    private String breed;
    private int age;

    public Dog(String name, String breed, int age) {
        this.name = name;
        this.breed = breed;
        this.age = age;

    }

    public String getName() {
        return name;
    }

    public String setName(String name) {
        this.name = name;
    }

    public String getBreed() {
        return breed;
    }

     public String setBreed(String breed) {
        this.breed = breed;
    }

    public int getAge() {
        return age;
    }

     public int setAge(int age) {
        this.age = age;
    }

}
```

Joonis 4. Andmeklass Javas.

```
data class Dog(var name: String, var breed: String, var age: int)
```

Joonis 5. Andmeklass Kotlinis.

Erinevalt Javast ei ole võimalik Kotlini väljadel vaikimisi null-väärtust hoida. Koodis null-väärtuse määramisel või tagastamisel ebaõnnestub kompileerimine. Oliveira, Teixeira ja Eberti uuringus [13] toodi välja, et see oli küsitletud arendajate arvates üks tähtsamaid Kotlini eeliseid üle Java. Javat kasutades tekiks sellises olukorras hoopis nullerind käitusajas siis, kui üritada mingit null-väärtusega muutujat kasutada või tagastada. Kui arendaja siiski soovib Kotlinis muutujale sellise väärtuse määrata, siis on see võimalik *"?"* operaatoriga, kuid see võib tekitada käitusajas nullerindeid nagu Javas [14].

Veel üks Kotlini tähtsatest omadustest on Javaga ühildumine. Mõlemat keelt on võimalik ühes projektis kasutada nii, et arendaja kahe keele kokkusobitamise eest vastutama ei pea. Mõlemad keeled kompileeritakse Java baitkoodiks, mida jooksutab platvorm, millel programmi kasutatakse [10]. See võimaldab Kotlinit kasutada vanematel seadmetel, mis toetavad ainult Javat. Samal põhjusel ei erine suurel määral keelte jõudlused. Korly poolt läbiviidud analüüsist ja testimisest leiti, et kuigi keelte vahel enamjaolt kompileerumiskiirused ei erine, siis on mõned olukorrad, nagu järkjärguline kompileerimine (ingl *incremental compilation*), kus Kotlin on Javast kiirem [15].

Androidi rakendust arendades teeb Kotlin lihtsamaks vaadetele (ingl *view*) ligipääsu. See teeb samuti kirjutatud koodi hulga väiksemaks, mis on suurte või keeruliste vaadete puhul tähtis (vt. tabel 1).

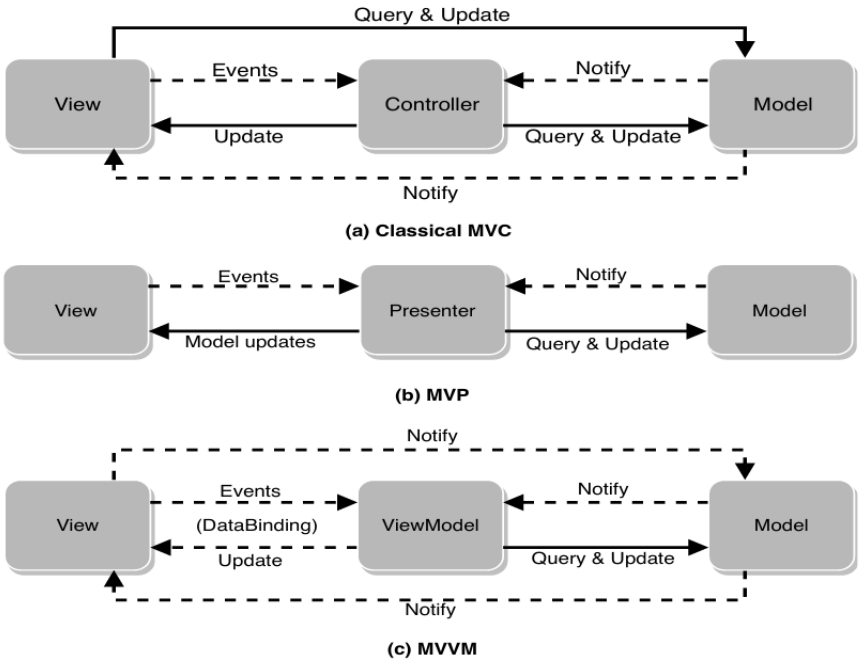
Tabel 1. Ekraanil kuvatava teksti muutmine Javas ja Kotlinis.

Java	Kotlin
TextView text = (TextView) findViewById (R.id.textView); text.setText ("Tere maailm");	textView.setText ("Tere maailm")

Kuigi rakenduse arendamisel Javas oleks võimalik leida rohkem abimaterjale, selgus võrdlusest, et Kotlini kasutamisel on rohkem eeliseid. Kõige tähtsamateks elementideks osutusid keele lühidus andmeklasside kirjutamisel, käitusajas null-erindite vältimine ja lihtsustatud süntaks. Lisaks toetuti uurimustele, mille käigus oli välja tulnud, et Kotlini kasutusele võtmine võib parandada koodi kvaliteeti ja loetavust [4,13].

3.5 Arhitektuursed mustrid Androidi rakendustes

Rakendust luues on vaja mingit viisi projekti koodi organiseerida, et see oleks hooldatav ja ka tulevikus täiendatav. Projekti koodi struktureerimiseks ja modulaarsuse tagamiseks kasutatakse arhitektuurseid mustreid [16]. Selles alapeatükis kirjeldatakse ja võrreldakse kolme populaarsemat Android-rakenduste arendajate poolt kasutatud arhitektuurset mustrit: Model-View-Controller, Model-View-Presenter ja Model-View-ViewModel (vt. joonis 6).



Joonis 6. MVC, MVP ja MVVM'i diagramm [21].

Kõik näiteks toodud mustrit jagavad andmekihti, milles hoitakse äriloogikat, võrguteenuseid ja kohaliku andmebaasi. Lisaks on kõigil kolmel mustril esitluskihti, kus kuvatakse andmekihist tulevat infot ja võetakse vastu kasutaja sisendit. Antud mustreid eristab andmekihi ja kasutajaliidese vahel asuvate kihtide funktsionaalsus ja disain.

3.5.1 Model-View-Controller

Näiteks toodud mustritest kõige vanem on *Model-View-Controller* (MVC), mis kasutab andmekihi ja kasutajaliidese vahel kontrollerit (ingl *controller*), mis reageerib kasutaja sisendile [17]. Enamjaolt kasutatakse kontrollerit, et kasutaja sisendi puhul andmekihist mingit kindlat meetodit välja kutsuda.

MVC arhitektuuri kasutades on kerge andmekihile moodulteste (ingl *unit test)* kirjutada. Hea kohustuste lahususe tõttu (ingl *separation of concerns, SoC*) on ka lihtne mitmel arendajal korraga rakendust arendada [16]. Kuna selles mustris sõltub esitluskiht mõlemast teistest kihist, siis võib esitluskihi loogikat muutes vaja uuendada mitut erinevat klassi, mis vähendab selle mustri kasutamise paindlikkust. Kuna *activity* on tihtipeale seda mustrit kasutades *kontrolleri* rollis, siis on see esitluskihiga väga lähedalt seotud. Seetõttu võivad mõned klassid väga suureks muutuda [16].

3.5.2 Model-View-Presenter

Model-View-Presenter (MVP) oli kirjutamise hetkel üks laialdasemalt kasutatud mustreid Androidi arendajate seas [18]. See on tuletatud MVC-st, aga eraldab äri- ja püsivusloogikat a*ctivity*'st ja *fragment*'ist rohkem. Sellele mustrile on omane esitleja (ingl *presenter)*, mis käitub nagu vahendaja esitluskihi ja andmekihi vahel, kus töödeldakse esitluskihist tulnud sisendit ja andmekihi väljundit. Selle eesmärgiks on vähendada *activity* klassi sees olevat koodi, et kasutajaliidest oleks kergem hooldada ja testida.

MVP-d kasutades eraldatakse esitluskiht rohkem andmekihist kui MVC-s. Seetõttu on efektiivsem kirjutada teste ja peab vähem tähelepanu pöörama erinevate kihtide eraldatusest [19]. Selles mustris on tähtis, et esitluskiht ei teaks mis ülejäänud kihtides toimub. Esitluskiht peaks teavitama esitlejat kasutaja sisendist. Samuti peaks esitleja ütlema esitluskihile, mida kuvada.

3.5.3 Model-View-Viewmodel

Model-View-Viewmodel (MVVM) muster sarnaneb MVP mustriga, kuid erinevalt MVP-st ei ole esitluskihi ja esitleja vahel üks-ühele suhet. Antud arhitektuurne muster liidestab andme- ja esitluskihti kasutades klassi nimega *viewmodel*, mis hoiab esitluskihi andmeid elutsüklist teadlikult. *Viewmodel* pole seotud kindla vaatega ja seetõttu saavad mitu vaadet ühte *viewmodelit* kasutada [19].

Viewmodelis võetakse vastu esitluskihi poolt tulevat infot ja väljastatakse andmekihist andmevooge. Kuna selles mustris on vaade *viewmodel'i* tarbija, siis on väga lihtne muuta rakenduse kasutajaliidest nii, et ülejäänud kihid jäävad samaks. Mai 2021 seisuga oli Google Jetpacki[15] juhendis MVVM soovitatud arhitektuur Androidi rakendustele. Tuginedes eelistele üle teiste arhitektuuride ja Huntloci funktsionaalsetele nõuetele (vt Lisa I), otsustati uue rakenduse arhitektuurina kasutada MVVM-i.

[15] https://developer.android.com/jetpack/guide

3.6 Head tavad Androidi rakenduse arendamisel

Koodi kirjutades ja Androidi rakendust arendades peaks silmas pidama häid tavasid. Heade tavade ja arendusnormide järgimine on tähtis nii arenduse kiiruse kui ka koodi kvaliteedi mõttes.

3.6.1 Head tavad programmeerimisel

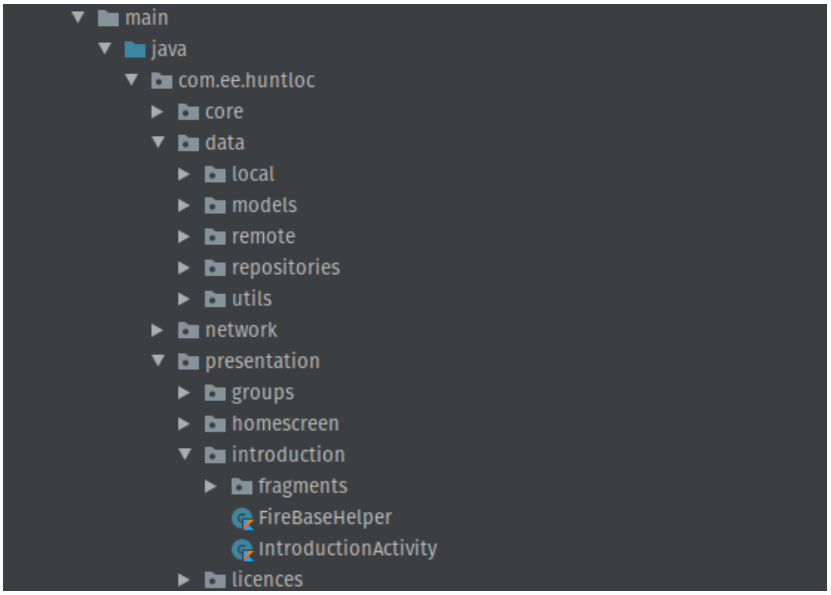
Rakendust arendades vigade ja koodi mahu vähendamiseks tuleks rakendada kohususte lahusust. Kohustuste lahusus seisneb selles, et eraldada mured üksteisest nii, et nad poleks üksteise vahel sidestatud. Eesmärgiks on tagada kood või lahendus, millest on kerge aru saada ja mida on kerge hooldada. [20,21]

Tähtis on ka rakendusliideste turvavõtmete ja teiste konfidentsiaalsete numbrite hoidmine kusagil, kus nad poleks avalikult kättesaadavad. *Futurice'i* Androidi arenduseeskirjades[16] soovitatakse Androidi rakenduse *signing key'd*, mis võimaldab selle *Google Play Store'*i üles panna, hoida privaatses failis, mida ei hoita versioonihaldussüsteemis. Selle võtme leidmisel oleks igal isikul võimalik rakendusest uus versioon *Google Play Store'*i üles panna. Eelnevas Huntloci versioonis oli paraku kirjutamise seisuga *signing key* avalikus failis ja ühtlasi kättesaadav.

3.6.2 Head tavad Androidi kontekstis

Koodi on hea paigutada tunnusjoonte järgi pakettidesse. Eesmärgiks on kapseldus, selgem funktsionaalsuste sõltuvus, liideste piirid ja lihtsam projektisisene navigeerimine. Näiteks on toodud bakalaureusetöö käigus loodud Huntloci Kotlini projekti struktuur, kus on klassid jaotatud paketidesse kihtide, vaadete ja erinevate mudelite kaupa (vt. joonis 7).

[16] https://github.com/futurice/android-best-practices



Joonis 7. Huntloci Kotlini versiooni paketid.

Futurice'i arenduseeskirjade[17] järgi soovitatakse ka võimalikult vähe koodi hoida *activity* klassides. Nendest peaks mõtlema kui konteineritest ja soovitatakse kasutada ka disainimustrit, kus kasutatakse näiteks ühe *fragment*'iga *activity* klassi selle asemel, et ekraani sisu kirjutada selle sisse. Tähtis on ka arhitektuurse mustri kasutamine. See võimaldab muidu koodibaasiga võõral arendajal kiiresti rakendusest aru saada. Lisaks tagatakse ühtlane struktuur tervele rakendusele ja kiirendatakse arendusprotsessi [18].

[17] https://github.com/futurice/android-best-practices

4. Olemasoleva lahenduse analüüs

Selles peatükis antakse ülevaade olemasoleva Huntloci rakenduse arhitektuurist, ülesehitusest ja komponentidest. Tuuakse välja arendamise käigus ilmnenud probleemid ja mured, mida proovitakse töö käigus lahendada.

Rakendusel ei olnud võimalik tuvastada laialdaselt levinud arhitektuurset mustrit. Eelnevas peatükis mainitud esitlus- ja andmekiht olid olemas, aga nende vahel polnud ühendavat osa. Vaadetesse tulevat infot päriti otse andmekihis olevatest *repository* klassidest.

4.1 Vaadete ülesehitus

Ettevõtte sisese dokumentatsiooni puudumise tõttu refereeritakse selles lõigus edaspidi Thebe artiklit [22], kus kirjeldatakse sarnase esitluskihi ülesehitusega rakenduse loomist. Rakenduse kasutajaliides põhineb *Single-Activity* mustril, kus uue *activity* või *fragment*'i loomise asemel kasutatakse Androidi vaateid[18] (ingl *view*). Sellise ülesehituse eesmärk on vältida olukorda, kus tekivad objektid[19] (ingl *object*), millest sõltub suur osa rakendusest. Kuna suur osa Huntloci funktsionaalsetest nõuetest on seotud kaardiga ja seega ühe objektiga, siis oli näha sellise arhitektuuri valimise põhjust.

Rakenduses kasutati navigeerimiseks Square'i poolt arendatud Flow[20] komponenti. Flowi uuendati viimati 2017 aasta oktoobris ja selle kasutamine kuulutati 2020 aasta aprillis mittesoovitatavaks. Vaadetesse sõltuvuste süstimiseks (ingl *dependency injection*, DI) kasutati Daggerit[21]. Selle eesmärgiks oli lihtne ligipääs klassidele ja liidestele, mis vajasid muidu uue isendi loomist (vt. joonis 8).

```
@Inject
TrackerRepository trackerRepository;
@Inject
UserRepository userRepository;
```

Joonis 8. Jälgimisseadme infovaate koodis Daggeri kasutamine.

[18] https://developer.android.com/reference/android/view/View
[19] https://techterms.com/definition/object
[20] https://github.com/square/flow
[21] https://github.com/google/dagger

Laialdasel kasutusel oli RxJava, mis võimaldas rakendusel kasutada nii andme- kui ka esitluskihis asünkroonseid andmevooge. Selle peamiseks eesmärgiks oli uuendada reaalajas jälgimisseadmete ja jahimeeste asukohti kaardil. Vasiliy [23] kirjutas, et pikaajalise hooldatavuse mõttes pole enam soovituslik RxJavat kasutada. Ta lisab, et selle komponentide kasutamine tekitab vajaduse seda terves koodibaasis kasutada, kuna ka moodulitestid peavad sisaldama RxJavale omaseid komponente.

4.2 Andmed

Huntloci rakendusse tulev info pärineb Javal põhinevalt serverilt, mis jookseb Google App Engine pilvandmetöötluse platvormil[22]. Rakendusse tuleb serverist info JSON-i formaadis. Selle töötlemiseks objektideks kasutati GSON-it[23], mis on Google'i poolt loodud JSON-i töötlemise teek. Kuna see ei toetanud automaatselt mitmetasandilisi objekte, siis olid Huntloci andmekihis paljude mudelite jaoks kirjutatud eraldi *JsonSerializer* klass, kus loodi objekt manuaalselt.

Rakenduses kasutati andmete hoiustamiseks Androidi sisseehitatud SQLite[24] andmebaasi ja selle rakendusliidest. SQLite on failipõhine avatud lähtekoodiga andmebaaside teek. Google'i Androidi arendajate dokumentatsioonis soovitatakse tugevalt SQLite rakendusliideste asemel kasutada Room'i püsivusteeki, mis on seda seda andmebaasi lihtsustav abstraktsioonikiht.

4.3 Arutelu

Huntloci rakendus kasutab mitmeid aegunuid komponente. Kuigi RxJavat toetati ka töö kirjutamise hetkel, siis otsustati seda kotlini asünkroonseid töid toetavate komponentide ja *RxJava* liigse keerulisuse tõttu mitte kasutada. Uues rakenduses otsustati kasutada Daggeri ülesannete täitmiseks seda lihtsustavat komponenti Hilt[25]. Suurimaks probleemiks esitus selge arhitektuurse mustri puudumine, mida osutati parandada MVVM-i kasutusele võtmisega.

[22] https://cloud.google.com/appengine
[23] https://github.com/google/gson
[24] https://developer.android.com/training/data-storage/sqlite
[25] https://developer.android.com/training/dependency-injection/hilt-android#hilt-and-dagger

5. Uue rakenduse arendamine

Selles peatükis kirjeldatakse uue Huntloci rakenduse komponente, arhitektuuri ja arendusprotsessi. Lisaks kirjeldatakse tulemuseks saadud rakendust ja mis tähelepanekud arendamise käigus välja tulid.

5.1 Arhitektuur ja keel

Rakendusest arendades otsustati eelmises peatükis välja toodud punktide tõttu kasutada Kotlinit. Kõige tähtsamaks tähelepanekuteks jäid koodi lühidus, null-erindite puudumine ja lihtne ligipääs vaadetele. Lisaks otsustati kasutada eelnevalt kirjeldatud *Model-View-Viewmodel* mustrit, mis eraldas selgelt rakenduses andmekihi ja esitluskihi, mis tegi mõlema poole hooldamise ja arendamise lihtsamaks. See tähendas rakenduse arendajale, et kasutajaliidest või kohalikku andmebaasi arendades ei teki kihtide vahel probleeme. Samuti ei sõltu kuvatavad andmed vaadete elutsükli muutustest, nagu rakenduse taustale minemine, pausile panemine, jms. Huntloci rakendust arendades oli see eriti tähtis, kuna andmete sissevool rakendusse on suur.

5.2 Kasutatud tehnoloogiad

Tuginedes Alcérreca ja Girise artiklite [24,25] otsustati kasutada andmete edastamiseks segu Kotlini *Flow* rakendusliidest ja Androidi *Livedata'st*. Livedata[26] on rakenduse elutsüklist teadlik andmete hoiustamise klass, mis on jälgitava olekuga. See tähendab, et Livedatat jälgivat objekti või meetodit teavitatakse andmete muutumise korral. Kuna see on ka teadlik rakenduse elutsüklist, siis oli seda kõige mõistlikum kasutada esitluskihi ja *viewmodel*i vahel. Flow'i kasutati *Repository* klassis, kus võeti vastu serverist tulevaid andmevooge ja salvestati neid rakendusse. Flow'iga on võimalik tagastada ühest asünkroonsest funktsioonist mitu väärtust, et teavitada rakendust, et hetkel käib andmete laadimine või et andmed on kätte saadud. *Viewmodel* kihis muudetakse *Flow'*iga saadud andmed *Livedata* tüüpi objektiks, et esitluskiht saaks neid jooksvalt uuendada.

Kohaliku andmebaasina kasutati *Roomi*[27], mis valiti lihtsuse ja suure kasutajabaasi tõttu. Lisaks on *Room* osa *Jetpacki*st[28]. Jetpack on ametlikult toetatud *Androidi* teekide komplekt,

[26] https://developer.android.com/topic/libraries/architecture/livedata
[27] https://developer.android.com/training/data-storage/room
[28] https://developer.android.com/jetpack

mille eesmärk on aidata arendajatel jälgida häid tavasid ja võimaldada koodil töötada üle kõigi Androidi versioonide. Rakendust arendades oli ka kasulik Room'i kompileerimise ajal toimuv SQL päringute kontrollimine. Seetõttu polnud võimalik süntaksi-vigadega või ebasobivalt kirjutatud päringuga rakendust kompileerida, mis tegi vigade leidmise lihtsamaks.

5.2.3 Võrguteenused ja andmed

Vanast rakendusest jäeti kasutusse REST klient Retrofit[29]. See teek muutis JSON andmete allalaadimise jaoks vajalike klasside kirjutamiseks lihtsaks ja selgeks. Kasutusse võeti ka HTTP klient OkHttp[30], et lisada igale päringule automaatselt serverile vajalik autentimise muster.

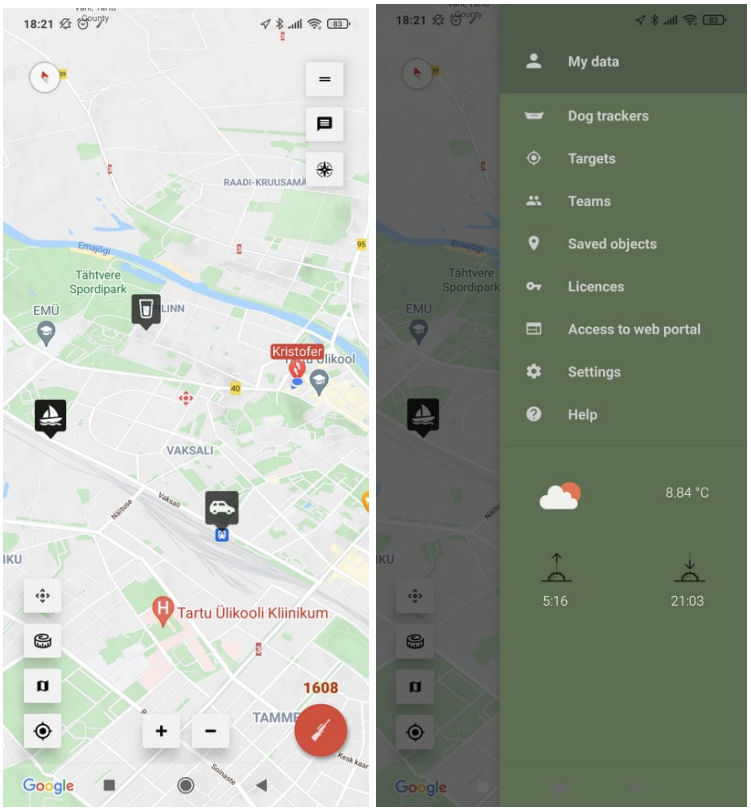
Andmete talitluseks kasutati Moshit[31]. Moshi on kaasaegne JSON-i teek Androidile ja Javale. Seda võib pidada töös eelnevalt kirjeldatud GSON-i järeltulijaks, kuna see on lihtsama rakendusliidese ning arhitektuuriga, kuid täidab sarnaseid eesmärke. Seda peetakse Kotlini-sõbralikuks teegiks, kuna see on varustatud Kotlini-teadlike laiendustega [26].

5.3 Rakenduse ülevaade ja arendusprotsess

Rakenduse põhiosa arendati lõikudes alates Novembrist 2020 kuni Märts 2021. Arenduskeskkonnaks oli Android Studio ja versioonihalduseks kasutati GitHub'i. Rakenduse andmekihi tähtsamad osad said valmis Detsembris ja edaspidi keskenduti rohkem esitluskihile ja kaardile funktsionaalsuste lisamisele. Arendamine viidi läbi üksi töö autori poolt.

Huntloci Kotlini versioon sisaldab kirjutamise seisuga samu funktsionaalsuseid mida Java versioon sisaldab. Toimusid ka mõned muutused kasutajaliidese disainis (vt. joonis 9), aga kuna bakalaureusetöö eesmärgiks oli luua uus koodibaas, siis sellele väga palju tähelepanu ei pööratud.

[29] https://square.github.io/retrofit/
[30] https://square.github.io/okhttp/
[31] https://github.com/square/moshi



Joonis 9. Uue Huntloci rakenduse kaardivaade ja menüü jahi ajal.

Rakenduse arendamist planeerides üritati pingsalt jälgida häid tavasid, et tulevikus arendusprotsessi mugavamaks ja kiiremaks teha. Arendaja perspektiivist oli arhitektuuri muutmine suur edasisamm. Uute esitluskihi osade lisamine oli palju lihtsam, sest vaadete ühendamine rakenduse andmekihiga oli läbi terve projekti ühtlane.

Viewmodel'i kasutamine tuli ootamatult kasuks rakendusele privaatsuspoliitika teksti lisamisel. Google'i privaatsuspoliitika nõuete järgi[32] oli vajalik seda rakenduse kasutuselevõtmisel kuvada. Kuna teksti laadimiseks kulus 3 kuni 4 sekundit, siis loodi *fragment*'i siseselt teksti laadimise asemel *viewmodel*, milles pandi juba asünkroone töö käima enne selle vaateni jõudmist. Tegemist oli rakenduses esimese ekraaniga, kus valitakse esmalt kasutuskeel ja seejärel tutvustakse privaatsuspoliitikat. Kuna veebilehelt teksti laadimist oli juba võimalik alustada keelevaliku ajal, siis järgmise ekraani kuvamise hetkeks oli tekst juba laetud.

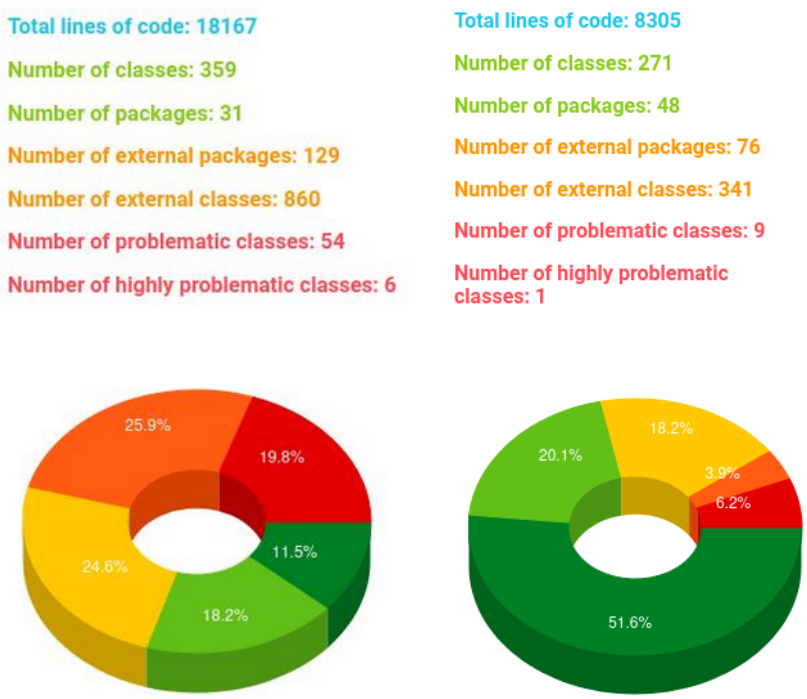
[32] https://support.google.com/googleplay/android-developer/answer/9859455?hl=en

6. Rakenduste võrdlus

Selles peatükis võrreldakse esialgse ja töö käigus valminud rakenduse arhitektuuri, komponente ja viiakse läbi ka jõudluse analüüs. Võrdluse lihtsustamiseks nimetatakse selles peatükis Huntloci rakenduse Java versiooni R1-ks ja Kotlini versiooni R2-ks

6.1 Ülesehitus

R2 taaskasutas mõnda komponenti R1-st, aga suurem osa rakendusest oli uus. Uue arhitektuuri ja Kotlini lühiduse tõttu vähendati koodi ridu rohkem kui poole võrra (vt. joonis 10). Lisaks on näha, et projekt oli struktureeritud nii, et see kasutas funktsionaalsuste eraldamiseks rohkem pakette. Joonisel 10 välja toodud probleemsed klassid (ingl *problematic classes*) on sellised, kus on kas kõrge sidusus teiste failide, klassidega või üleliigselt vastastikmõjusid funktsioonide või klasside vahel.



Joonis 10. CodeMR analüüs R1st (vasakul) ja R2st (paremal).

6.2 Jõudluse võrdlus

Rakenduste võrdluseks vaadati ka erinevusi rakenduse kompileerimisel ja telefonis kasutamisel erinevates olukordades.

6.2.1 Kompileerimine

Koodibaase kompileeriti läbi Android Studio ja Gradle. Tulemuseks võeti 10 kompileerimise keskmine. Esialgse kompileerimise mõõtmise testide vahel tühjendati puhver. R1 kasutas *Gradle* versiooni 5.1.1. R2 kasutas *Gradle* versiooni 6.5. R2 esialgsele kompileerimisele kulus keskmiselt kauem kui rakendusel R1. R2 puhul oli tunduvalt kiirem järkjärguline kompileerimine, mis võib olla kasulik rakendusse muudatuste tegemise testimisel (vt. tabel 2).

Tabel 2. Kompileerimise kiiruse võrdlus mõlemas rakenduses.

	R2	**R1**
Esialgne kompileerimine	~24 sekundit	~18 sekundit
Järkjärguline kompileerimine	~1 sekund	~6 sekundit

6.2.2 Mälu ja protsessori kasutusnäidud

Kasutades Android Studio profileerijat, mis on tööriist reaalajas rakenduse jõudluse testimiseks, viidi läbi lühikesed testid mõlemal rakendusel. Testimiseks kasutati telefoni Xiaomi Mi Note 10 Lite. Katsete tegemise ajal suleti telefonis iga katse vahel kõik võimalikud rakendused. Kaardi kasutamist mõõdeti vaate suurendamise ja vähendamisega ja kaardil ringides liikumisega. Kaardil oli kuvatud kasutaja, üks jälgimisseade ja 10 objekti. Töö käigus arendatud rakendus kasutas mingil määral vähem vahemälu, aga rohkem telefoni protsessorit (vt. tabel 3). Seejärel võrreldi mõlemas rakenduses erinevate vaadete vahel navigeerimist. Selle testi korral olid mõlemad tulemused paremad Kotlini rakenduses, kuigi vahemälu kasutuse vahe oli väike (vt. tabel 4). Mõlemat testi viidi läbi viis korda ja võeti keskmine tulemus.

Tabel 3. Keskmine CPU ja RAMi kasutus rakenduse kaardivaatel navigeerimisel.

	R1	**R2**
Keskmine CPU kasutus	13%	15%
Keskmine RAM kasutus	399 MB	312 MB

Tabel 4. Keskmine CPU ja RAMi kasutus rakenduse vaadete vahel navigeerimisel.

	R1	**R2**
Keskmine CPU kasutus	13%	6%
Keskmine RAM kasutus	330 MB	324 MB

Väiksem CPU kasutus vaadete vahel navigeerimisel rakenduses R2 tekkis ilmselt uuemate komponentide ja arhitektuurse mustri kasutamise tõttu. Kuna rakenduse R1 navigeerimiskomponenti Flow[33] ei oldud mai 2021 seisuga uuendatud 4 aastat, võib eeldada, et seda polnud optimiseeritud uuemate Androidi versioonide jaoks.

6.3 Arendusprotsesside võrdlus

Selles alapeatükis võrreldakse nii R1 kui ka R2 uute funktsionaalsuste lisamist. Põhjuseks oli välja selgitada, kas R2 täitis töö käigus paika pandud eesmärgid. Võrreldi kolme erinevat tüüpi arendamist mõlemas rakenduses. Lisati uus funktsionaalsus tuginedes olemasolevatele komponentidele ja rakenduse arhitektuurile, muudeti olemasolevat funktsionaalsust ja võrreldi uue komponendi lisamist ja kasutamist. Arendamist võrreldi nii kvantitatiivselt kui ka kvalitatiivselt. Loetleti mitu rida koodi lisati, muudeti või kustutati. Samal viisil vaadati ka klasse, faile ja meetodeid. Peale funktsionaalsuse lisamist kirjeldas arendaja isikliku muljet arendamise kogemusest.

Arendusprotsesside võrdluseks vaadati rakenduste andmekihti ja esitluskihti eraldi. Rakenduses R2 liigitati lihtsuse mõttes *viewmodel* klassid esitluskihi alla. Kuna *viewmodeli* eesmärki täitvad meetodid asuvad R1-s samuti esitluskihis, peaks selline võrdlus olema täpsem, kui selle andmekihi alla kategoriseerimine. Funktsionaalsus lisati esmalt rakendusse

[33] https://github.com/square/flow

R2 ja seejärel rakendusse R1. Selle eesmärgiks oli tagada, et arendaja ei tunneks ennast juba selle funktsionaalsuse arendamisel R2-s mugavamalt. Arenduse protsessiks loeti töö kontekstis funkstionaalsuse jaoks põhilise koodi kirjutamist. Testimist, küljendusfailide kirjutamist ja vigade parandamist ei arvestatud ajakulus ega ka teistes mõõdutes

6.3.1 Uued funktsionaalsused

Eesmärgiks võeti kolme funktsionaalsuse lisamine, mis annaksid hea ülevaate arendamise erinevustest mõlemas rakenduses. Võrreldakse funktsionaalsuste lisamise ajakulu, lisatud koodi ridu ja arendaja kogemust.

Suurimaks eesmärgiks võeti Huntloci veebiportaalis oleva tiimde funktsionaalsuse rakendusse lisamine. Portaalis on võimalik määrata kasutajaid meeskondadesse. Meeskondade liikmetel on lihtne ligipääs üksteise telefoninumbritele ja ka teiste kasutajate kaardile loodud objektidele. Selle funktsionaalsuse üleviimine rakendusse oli Huntloci arendusmeeskonnal juba ka varem silmis, kuid selleni polnud ajapuuduse tõttu veel jõutud.

Teiseks funktsionaalsuseks on jahiga liitunud kasutaja kuvamine, kui tal puudub GPS näit. Kirjutamise hetkel polnud rakendustes näha kasutajat, kes liitus jahiga hetkel kui tema telefonil polnud GPS signaali. Otsustati, et selle näitamine tekitaks jahis rohkem selgust,

Kolmandaks funktsionaalsuseks on jälgimisseadme haldusvaates seadme viimase asukoha pildina kuvamine, kasutades Google Maps Static rakendusliidest[34]. Antud rakendusliides tagastab pildi kaardist kohal, mille koordinaadid anti päringuga kaasa. Kirjutamise hetkel oli antud vaates näha ainult seadme viimase asukoha koordinaate, millest kasutajatele kasu ei olnud.

6.3.2 Tiimide nimekirja lisamine

Funktsionaalsuse andmekihi loomine hõlmab endas uute REST teenuste kasutamist, kohalikku andmebaasi mudelite loomist ja sinna salvestamise võimaldamist. Huntloci serveripoolne lahendus kasutas kahte põhilist mudelit. *Team*, kus oli selle loomise aeg , nimi, ja liikmete nimekiri ja *TeamPermission*, mis oli tiimi liikme jaoks loodud objekt, kus oli määratud kasutaja telefoninumber ja tema õigused. Neid kasutati ka rakenduses samas

[34] https://developers.google.com/maps/documentation/maps-static/overview

formaadis. Tabelis 5 on välja toodud andmekihi arendamise kvantiatiivsed mõõdud. Mõningad probleemid tekkisid sellest, et serveri poolne lahendus ei olnud täielikult valmis ja et *Team* klassi ülesehitus erines üsna palju teistest andmeklassidest, mida teenus varem pakkus. Olukord tegi funktsionaalsuse töötamise testimise keerulisemaks ja ajakulu mingil määral suuremaks mõlemas rakenduses.

Tabel 5. Tiimide funktsionaalsuse lisamine (Andmekiht).

	Koodiread	Klassid	Meetodid	Failid	Ajakulu
R1	Lisati: 700	Lisati: 5	Lisati: 62	Lisati: 5	71 minutit
R2	Lisati: 429 Eemaldati 8	Lisati: 6	Lisati: 37	Lisati: 8 Muudeti: 5	39 minutit

Tiimide nimekirja funktsionaalsuse esitluskihi loomine hõlmas endas kahe põhilise vaate loomist. Esmalt loodi nimekiri, kus oli tiime näha. Seejärel loodi individuaalse tiimi liikmete nimekiri, millele sai ligi tiimile nimekirjas peale vajutades. Tiimi loojal ja muutmisõigusega isikul anti võimalus liikmeid kustutada ja nende õigusi muuta. Tabelis 6 on välja toodud esitluskihi arendamise kvantitatiivsed mõõdud

Tabel 6. Tiimide funktsionaalsuse lisamine (Esitluskiht).

	Koodiread	Klassid	Meetodid	Failid	Ajakulu
R1	Lisati: 840	Lisati: 10	Lisati: 56	Lisati: 8	74 minutit
R2	Lisati: 780	Lisati: 7	Lisati: 35	Lisati: 6	58 minutit

Arendaja kogemus rakenduses R2 oli üldjoontes positiivne. Rakenduse lihtsa ja konsistse ülesehituse tõttu oli kerge luua uued mudelid andmekihti ja luua ühendus REST teenusega. Olemasolevaid klasse ja liideseid kasutati mallina, seega läks arendusprotsess kiiresti. Esitluskihi arendus viidi läbi sama põhimõtte järgi.

Rakendust R1 täiendades märkas arendaja, et töö käigus oli vaja aktiivsemalt mõelda ja keskenduda, sest andmekihis kasutatavad tehnoloogiad olid keerulisemad. Rakenduses R1 oli

vajalik JSON-i töötlemiseks kirjutada eraldi keerukas klass, kuna olemasolev lahendus polnud ehitatud pesastatud klasse sisaldavate vastete jaoks. Rakenduses R2 sellist probleemi ei esinenud, sest andmekihis kasutatud Moshi töötleb ka pesastatud klasse sisaldavat JSON-it automaatselt.

6.3.3 Google Maps Static rakendusliidese rakendamine

Enne funktsionaalsuse lisamist, eeldati selle läbiviimiseks suuremat ajakulu. Leiti, et oli efektiivne kasutada Glide'i[35], mis on avatud lähtekoodiga piltide laadimise raamistik. Selle kasutamine tegi mõlemas rakenduses selle funktsionaalsuse koodi lisamise väga lihtsaks. Tabelis 7 on toodud välja funktsionaalsuse lisamise kvantitatiivsed mõõdud.

Tabel 7. Uue rakendusliidese rakendamine jälgimisseadme vaates (Esitluskiht).

	Koodiread	Klassid	Meetodid	Failid	Ajakulu
R1	Lisatud: 51 Eemaldatud: 7	Lisatud: 0	Lisatud: 2	Muudetud: 2	13 minutit
R2	Lisatud: 64 Eemaldatud: 2	Lisatud: 0	Lisatud: 2	Muudetud: 2	14 minutit

Mõlemasse rakendusse funktsionaalsuse lisamine oli arendaja jaoks lihtne. Kaardile vajutamise funktsionaalsust oli lihtsam rakendusse R1 lisada, sest kaardivaatesse oli juba eelnevalt lisatud sarnane funktsionaalsus. Arendaja pani tähele, et kuigi funktsionaalsust oli rakendusse lihtsam lisada, siis anti kaardivaatesse kaasa mahukas *Huntlocation* objekt, mis sisaldas jälgimisseade koordinaate. Aja kokkuhoiu mõttes ja rakenduse arhitektuuri mitte muuta soovides jäeti see lahendus alles. Sellele tuginedes otsustas arendaja rakenduses R2 anda kaardile vajutades andmetena kaasa ainult koordinaadid, et edastatud andmete maht oleks võimalikult väike.

6.3.4 Jahis osaleja kuvamine ilma GPS signaalita

Jahis osaleja kuvamiseks ilma GPS signaalita oli vaja kätte saada rakenduse asukoha haldurist viimane salvestatud asukoht ja saata see läbi REST teenuse serverisse. Tabelis 8 on välja toodud funktsionaalsuse lisamise kvantitatiivsed mõõdud.

[35] https://github.com/bumptech/glide

Tabel 8. Jahis osaleja kuvamine ilma GPS signaalita.

	Koodiread	Klassid	Meetodid	Failid	Ajakulu
R1	Lisatud: 14	Lisatud: 0	Lisatud: 2 Muudetud 3	Muudetud: 2	9 minutit
R2	Lisatud: 7	Lisatud: 0	Lisatud: 0 Muudetud 1	Muudetud: 1	2 minutit

Funktsionaalsuse lisamine rakenduses R2 oli lihtne. Muuta oli vaja ainult meetodit, kus liituti jahiga, kuna asukoht oli juba samas klassis kättesaadav. Seejärel kasutati samat meetodit, mis saatis serverisse kasutaja asukoha tavaolukorras. Rakenduses R1 oli vaja muuta kasutaja asukohta haldavat *LocationService* klassi, kus eelnevalt ei olnud võimalust vanat asukohta lihtsalt kätte saada. Peale sellesse vajaliku meetodi lisamist, oli võimalik asukoht kätte saada ja serverisse edastada.

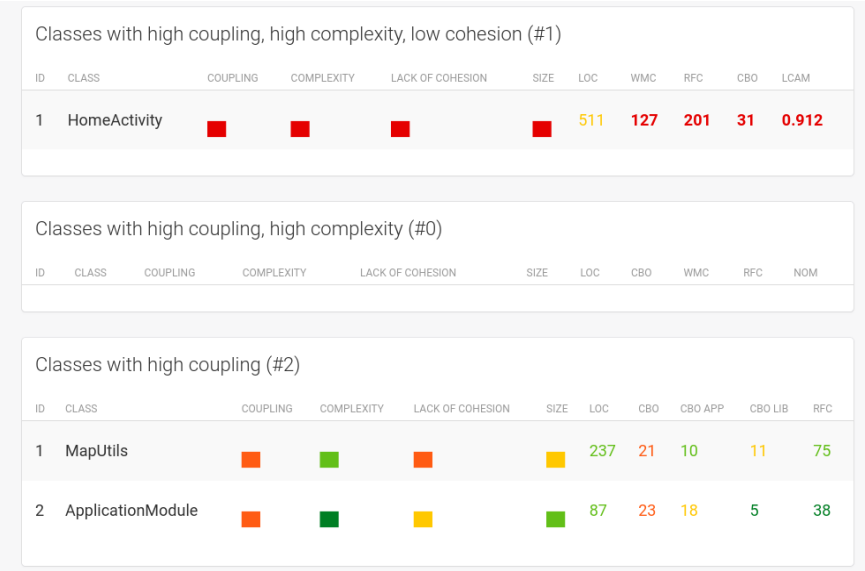
6.4 Arutelu

Rakenduses R2 oli kõige märgatavam erinevus uue koodi lisamisel väiksem ajakulu ja arendaja lihtne arusaam tööprotsessist. Tiimide funktsionaalsust lisades andmekihti säästeti võrreldes rakendusega R1 peaaegu pool ajast. Arendaja kogemusest lähtudes võib järeldada, et rakendusse R2 funktsionaalsuste arendamine oli lihtsam ja säästeti nii aega kui ka energiat, mida oleks saanud rakendada teiste funktsionaalsuste arendamisele.

Funktsionaalsuste lisamise võrdluses peab arvestama, et kuigi arendaja oli juba eelnevalt pädev Javas programmeerimises, arenes ta oskus Kotlinit kasutada töö käigus märgatavalt. Arendaja tundis ennast töö kirjutamise lõpuks mugavamalt Kotlinis ja arenduse ajakulu võis ka sellest sõltuda. Sellega arvestamiseks otsustati, et funktsionaalsus lisatakse ennem rakendusse R2, et arendaja oleks juba enne rakenduses R1 tööle asumist korra funktsionaalsuse lisamise protsessi läbinud.

6.5 Edasiarenduse plaanid

Rakenduse R2 ülesehituses tekkisid arendades mõned probleemid. Kasutades koodi analüüsimise tööriista CodeMR[36], avastati, et kaardi näitamisega ja sellele objektide kuvamisega seotud klassid olid liigse keerukuse ja suurusega (vt. joonis 11).



Joonis 11. CodeMR analüüs koodi kvaliteedist.

Kuna Traxby OÜ-l on soov võtta valminud rakendus kasutusele enne järgmist jahihooaega, siis on plaanis see osa koodist ära parandada ja ümber kirjutada. Lisaks oleks võimalik liigutada rohkem koodi *viewmodel* klassidesse, et esitluskihis oleks võimalikult vähe äriloogikat. Seejärel on planeeritud ka Liveteam ja Doglo rakendused uuendada kasutades töö käigus valminud koodibaasi, seega omab firmale vigade parandamine suurt tähtsust.

[36] https://plugins.jetbrains.com/plugin/10811-codemr

Kokkuvõte

Töö eesmärgiks oli arendada modernne ja paremini hooldatav versioon olemasolevast Huntloci rakendusest. Selleks analüüsiti olemasoleva rakenduse ülesehituse puudujääke ja erinevusi Kotlini ja Java vahel. Jõuti järeldusele, et Kotlinil on piisavalt eeliseid üle Java, et seda projekti keelena kasutada. Kõige tähtsamateks omadusteks osutusid null-erindite puudumine ja kompaktne kood. Androidi rakenduste arhitektuursete mustrite võrdluse tulemusena valiti Model-View-Viewmodel uue rakenduse arhitektuuriks, et tagada eraldatus andmekihi ja esitluskihi vahel.

Arendamine viidi läbi 2020. aasta novembrist kuni 2021. aasta märtsini. Peale rakenduse valmimist võrreldi funktsionaalsuste lisamist nii uute projekti kui ka olemasolevasse rakendusse, et selgitada välja, kas täideti töö eesmärgid. Analüüsides mõlemasse rakendusse funktsionaalsuste lisamist võib saadud andmetest järeldada, et bakalaureusetöö käigus loodud rakenduse arendamisele kulub märkimisväärselt vähem aega ja tekib ka vähem koodi. Arhitektuurse mustri rakendamine tuli arendamise käigus kasuks, lihtsustades nii asünkroonsete töödega seotud funktsionaalsuste lisamist kui ka üldist arendusprotsessi. Töö käigus loodud rakendus vajab enne avalikustamist veel edasist arendamist ja testimist. Traxby OÜ-l on plaanis rakendus laialdaselt kasutusele võtta 2021. aasta sügisel.

Viidatud kirjandus

[1] O'Dea, S. Android: global smartphone OS market share 2011-2018, by quarter, 2020 https://www.statista.com/statistics/236027/global-smartphone-os-market-share-of-android/, (10.12.2020)

[2] Brandom, R. There are now 2.5 billion active Android devices, 2019 https://www.theverge.com/2019/5/7/18528297/google-io-2019-android-devices-play-store-total-number-statistic-keynote (12.12.2020)

[3] Vaughan-Nichols, S. J. A Google Android and Java history lesson, 2011 https://www.zdnet.com/article/a-google-android-and-java-history-lesson/ (12.12.2020)

[4] Flauzino, M., Verissimo, J., Terra, R., Cirilo, E., Durelli, V., Durelli, R. Are you still smelling it?: A comparative study between Java and Kotlin language. *The VII Brazilian Symposium*, 2018 (03.05.2021)

[5] About, Huntloc https://huntloc.com/en/about/ (07.01.2021)

[6] AppBrain'i statistika Huntloci populaarsuse kohta. https://www.appbrain.com/app/huntloc-hunting-app-and-dog-tracking/ee.topgravity.android.huntloc (03.05.2021)

[7] Oracle tutvustus programmeerimiskeelele Java. https://java.com/en/download/help/whatis_java.html (20.02.2021)

[8] Amin, N., Tate, R. Java and Scala's Type Systems are Unsound: The Existential Crisis of Null Pointers. *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications,* 2016 (01.05.2021)

[9] Hovenmeyer, D., Spacco, J., Pugh, W., Evaluating and tuning a static analysis to find null pointer bugs. *PASTE '05: Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, 2005 (01.05.2021)

[10] Crisan, A. A study of Kotlin's: conciseness, safety and interoperability. Kataloonia Polütehnilise Ülikooli arvutiteaduse bakalaureusetöö, 2019 https://upcommons.upc.edu/handle/2117/172871 (02.03.2020)

[11] Ardito, L., Coppola, R., Malnati, G., Torchiano, M. Effectiveness of Kotlin vs. Java in android app development tasks. *Information and Software Technology Volume 127*, 2020 (18.04.2021)

[12] Coppola, R., Ardito, L., Torchiano, M. Characterizing the transition to Kotlin of Android apps: a study on F-Droid, Play Store, and GitHub. *WAMA 2019: Proceedings of the 3rd ACM SIGSOFT International Workshop on App Market Analytics*, 2019 (03.05.2021)

[13] Oliveira, V., Teixeira, L., Ebert, F. On the Adoption of Kotlin on Android Development: A Triangulation Study. *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2020 https://leopoldomt.github.io/assets/pdf/2020-saner.pdf (04.05.2021)
[14] Stürt, R. Null Safety Tutorial in Kotlin: Best Practices, 2019 https://www.raywenderlich.com/436090-null-safety-tutorial-in-kotlin-best-practices (06.04.2021)
[15] Korly, J. Kotlin vs Java: Performance Drill down & Which to choose, 2019 https://medium.com/@johnkorly/kotlin-vs-java-performance-drill-down-which-to-choose-2514bdf91916 (10.04.2021)
[16] Lou, T. A comparison of Android Native App Architecture, MVC, MVP and MVVM. Eindhoveni tehnikaülikooli matemaatika ja arvutiteaduse magistritöö, 2016 https://pure.tue.nl/ws/portalfiles/portal/48628529/Lou_2016.pdf (04.05.2021)
[17] Deacon, J. Model-View-Controller (MVC) Architecture. *Journal of Software Engineering and Applications,* Vol.10 No.12*,* 2009 https://www.rareparts.com/pdf/MVC.pdf (04.05.2021)
[18] Daoudi, A., Moha, N., El-Boussaidi, G., Kpodjedo, S. An Exploratory Study of MVC-based Architectural Patterns in Android Apps. *The 34th ACM/SIGAPP Symposium, 2019* (04.05.2021)
[19] Tripathi, S., Narang, T. Applying Model View View-Model and Layered Architecture for Mobile Applications. *International Conference of International Academy of Physical Sciences (CONIAPS-XVIII)*, 2016 (04.05.2021)
[20] Hürsch, W., Lopes, C. Separation of Concerns. Northeastern University, *Technical report NU-CCS-95-03*, 1995 (03.05.2021)
[21] Gurp, J., Bosch, J. Separation of Concerns: A Case Study, 2002 https://www.researchgate.net/publication/2563874_Separation_of_Concerns_A_Case_Study, (04.05.2021)
[22] Thebe, O. Creating Modular Android Apps with Dagger 2, Flow and Mortar, 2018 https://medium.com/@ozzythebe/creating-modular-android-apps-with-dagger-2-flow-and-mortar-dcea66a8acfd (20.04.2021)
[23] Zukanov, V. What can we learn from the demise of RxJava? 2020 https://www.techyourchance.com/rxjava-lessons-learned/ (21.04.2021)

[24] Alcérreca, J. LiveData with Coroutines and Flow, 2020
https://medium.com/androiddevelopers/livedata-with-coroutines-and-flow-part-i-reactive-uis-b20f676d25d7 (06.04.2021)
[25] Giris, S.F. Using LiveData & Flow in MVVM, 2020
https://proandroiddev.com/using-livedata-flow-in-mvvm-part-i-a98fe06077a0 (06.04.2021)
[26] Beyls, C. Advanced JSON parsing techniques using Moshi and Kotlin, 2018
https://bladecoder.medium.com/advanced-json-parsing-techniques-using-moshi-and-kotlin-daf56a7b963d (06.04.2021)

Lisad

I. Huntloci funktsionaalsete nõuete tabel

Jrk nr	Nimi	Kirjeldus
1	Konto loomine	Uue kasutaja loomine telefoninumbriga ja autentimine
2	Keele vahetamine	Rakenduses kuvatava keele vahetamine
3	Erinevate kaardikihtide kuvamine	Kasutajal võimaldada valida mitme erineva välimusega kaardikihi vahel valida
4	Jälgimisseadme lisamine	Huntloci jälgimisseadme lisamine kas infot sisestades või QR koodi skänneerides
5	Jälgimisseadme haldamine	Jälgimisseadme käivitamine või kinni panemine
6	Lisatud jälgimisseadme kaardil kuvamine	Jälgimisseadme asukoha kuvamine ja uuendamine kaardi peal
7	Jahipoodide kaardil kuvamine	Huntloci seadmeid müüvate poodide kaardil välja toomine
8	Kasutaja enda objektide loomine	Kasutaja võib luua enda objekte, millel on enda ikoon, nimi ja kirjeldus
9	Kasutaja objektide kuvamine	Kasutaja objekte kuvatakse kaardi peal
10	Sihtkoha määramine	Kasutaja saab määrata kaardil sihtkoha ja sellest näidatakse distantsi ja suunda
11	Kompass	Eraldi vaade, kus näidatakse telefoni andurite järgi kompassi
12	Jahipidamine	Teiste kasutajatega ühise *sündmuse* loomine, kus kuvatakse teiste asukohti ja aktiivseid jälgimisseadmeid
13	Kasutaja ligipääsu piiramine sõltuvalt litsentsi olemasolust	Kasutajale lubatakse rohkem funktsionaalsuseid kasutada, kui tal on rakenduse litsents
14	Ilmateade	Rakendus kuvab menüüs kasutaja asukoha ilma, peale vajutades näidatakse ka järgmise kolme päeva ilmateadet
15	Joonlaud kaardi peal	Võimalik ise määrata punkte, mille vahel distantsi mõõta ja ekraanile kuvada

II. Litsents

Lihtlitsents lõputöö reprodutseerimiseks ja üldsusele kättesaadavaks tegemiseks

Mina, Kristofer Käosaar,

1. annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) minu loodud teose Jahitegevust toetava Android-rakenduse moderniseerimine Kotlinis, mille juhendaja on Jakob Mass, reprodutseerimiseks eesmärgiga seda säilitada, sealhulgas lisada digitaalarhiivi DSpace kuni autoriõiguse kehtivuse lõppemiseni.
2. Annan Tartu Ülikoolile loa teha punktis 1 nimetatud teos üldsusele kättesaadavaks Tartu Ülikooli veebikeskkonna, sealhulgas digitaalarhiivi DSpace kaudu Creative Commonsi litsentsiga CC BY NC ND 3.0, mis lubab autorile viidates teost reprodutseerida, levitada ja üldsusele suunata ning keelab luua tuletatud teost ja kasutada teost ärieesmärgil, alates 07.05.2021 kuni autoriõiguse kehtivuse lõppemiseni.
3. Olen teadlik, et punktides 1 ja 2 nimetatud õigused jäävad alles ka autorile.
4. Kinnitan, et lihtlitsentsi andmisega ei riku ma teiste isikute intellektuaalomandi ega isikuandmete kaitse õigusaktidest tulenevaid õigusi.

Kristofer Käosaar
07.05.2021