VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS

Baigiamasis bakalauro darbas

Veiklos automatizavimas Android sistemoje remiantis įvykiais grįsta architektūra

(Operating automation for Android based on event-driven architecture)

Atliko: 4 kurso, 2 grupės studentės
Evelina Didžiokaitė (parašas)
Violeta Skinderytė (parašas)
Darbo vadovas:
prof. dr. Rimantas Vaicekauskas (parašas)
Recenzentas:
lekt. Jonas Ragaišis (parašas)

Vilnius
2020

Turinys

Įvadas

Pastaraisiais metais įvykiais pagrįsta architektūra pradėjo populiarėti, tačiau vis dar yra daug žmonių susidariusių klaidingą nuomonę, todėl sunku pasakyti, kokios sistemos iš tiesų naudoja įvykiais pagrįstą architektūrą ir kurios tik kažką panašaus. Dėl tokio nesupratimo net įmonės, kurios turi įvykiais pagrįsta architektūra paremtų sistemų, mažai jomis rūpinasi [Cur17]. Šiame darbe bus aprašoma, kas yra įvykiais pagrįsta architektūra ir kaip dažniausiai ji yra vaizduojama sistemose.

Vien iš įvykiais pagristos architektūros pavadinimo galima suprasti, kad pagrindinis šios architektūros elementas yra įvykis, tačiau ši sąvoka nėra vienareikšmė. Programavime įvykiu gali būti laikomas būsenos pokytis ir objektas aprašantis tą pokytį. Net jei įvykis nutiko seniai jo aprašas sistemos darbinėje atmintyje yra išlaikomas tol, kol jis yra aktualus, ir priklausomai nuo struktūros gali būti saugomas žurnale. Toks informacijos dalijimasis leidžia sistemos elementams atsiriboti vieniems nuo kitų ir naudoti sistemą, kurioje yra svarbu sukurti procesą, kuris gali skaityti skelbiamus įvykius, o ne centrinę komponentę, kuri žino apie visus procesus.

Nors įvykiais pagrįsta architektūra praktiniuose įgyvendinimuose retai kada tiksliai atitinka teorinį modelį, tačiau jos turi panašumą į vieną iš dviejų topologijų: mediatoriaus ir tarpininko. Arba sistemoje yra pritaikytas vienas iš trijų įvykiais pagrįstų modeliais: įvykių pranešimai, įvykiais perduodamos būsenos kopijavimas ir įvykių atsekamumas.

Remiantis šia architektūra yra patogu realizuoti automatizacijos sistemas. Automatizacija yra žmogaus darbo keitimas technologiniu, nors tokios sistemos gali turėti platų veiklos krypčių spektrą, šiame darbe bus kalbama apie sistemas dirbančias su mobiliaisiais įrenginiais. Tokios sistemos veikia stebėdamos aplinkos arba vidinius įvykius. Vienas iš tokios sistemos scenarijų yra mobiliojo telefono reakcija į iš anksto nustatyta įvykį, pavyzdžiui, pranešimo atsiuntimas, kai įrenginys yra netoli nustatytos vietos. Šiame darbe aprašoma sistema įgyvendina tokių situacijų automatizavimą Android operacinėje sistemoje. Programos prototipo kūrimui buvo panaudota „Android Studio“ programavimo aplinka, todėl programos aprašymai bus paremti šios aplinkos dokumentacija.

Darbo tikslas buvo išnagrinėti įvykiais pagrįstą architektūrą, ją pritaikyti automatizacijos sistemos kūrimui, patikrinti, kaip tai yra įgyvendinama Android operacinėje sistemoje ir

pabandyti išspręsti užkulisinio veikimo trūkumus atsirandančius panašiose sistemose dėl nuolat besikeičiančios „Android Studio“ dokumentacijos.

Darbo pasidalijimas: Evelina Didžiokaitė parašė antrą ir trečią skyrius bei vartotojo sąsają programos prototipe, Violeta Skinderytė – pirmą skyrių ir servisą, o viską, kas liko, kartu.

1. Įvykiais pagrįsta architektūra

Įvykiais pagrįsta architektūra (angl. *Event-Driven architecture*), trumpiau vadinama EDA, veikia kiekvieną svarbesnį sistemos pokytį suformuluojant į pranešimo formatą, vadinama įvykiu, ir jį paskelbiant visiems sistemos elementams. Tokia informacijos perdavimo forma leidžia komponentams pasiimti ir apdoroti tik tą informaciją, kuri yra aktuali vykdomam procesui, bei išvengti tiesioginio kitų komponentų iškvietimo, kuris reikalauja žinoti pagrindinę proceso informaciją:

- ar procesas jau yra sukurtas,
- ar veikiantis, jei ne – kokią informaciją reikia perduoti norint jį paleisti,
- kokių parametrų reikalauja norimas metodas,
- kaip vadinami metodai ir pats procesas.

Taip atsiribojama nuo sistemos sandaros ir padedama sumažinti tiesioginio bendravimo tarp sistemos elementų būtinybę. Dėl šios savybės atsiranda daug įvykiais pagrįstos architektūros panaudojimo atvejų, kurių vartotojas gali ir nepastebėti – bent minimali architektūros forma yra naudojama daugumoje programavimo kalbų bei aplinkų. Programuotojams geriausiai yra pažįstami įvykiais pagrįstos architektūros pavyzdžiai yra:

- versijų kontrolės sistemos (tokios kaip Git) [Fow17a],
- grafinės vartotojo sąsajos susiejimas su programiniu kodu [Fow17a],
- daugiagijis Java programavimas, kuriame EDA pagrindais slepiasi už asinchroninio veikimo ir metodų *notify* (*notifyAll*) ir *wait*.

Nepaisant tokio plataus architektūros panaudojimo ji tebėra menkai suprantama tarp programuotojų, kurie specialiai tuo nesidomi, todėl norint išvengti nesusipratimų pirmiausia reikia aptarti kertines su įvykiais pagrįsta architektūra susijusias sąvokas.

1.1 Įvykis

Pirmasis žingsnis į įvykiais pagrįstą architektūrą yra suprasti kaip apibrėžiamas įvykis. Intuityviai, įvykis yra bet koks pastebimas pokytis, tačiau programuojant apibrėžimas yra išplečiamas iki sistemos atliekamų veiksmų, kurių informacija gali būti reikalinga daugiau nei

vienam procesui. Todėl įvykiais nelaikomi vidiniai metodai, kurie dalinai įgyvendina proceso funkcionalumą, pavyzdžiui jei proceso funkcija yra išrūšiuoti duomenų sąrašą, įvykiais yra laikoma tik rūšiavimo pradžia ir pabaiga ir nelaikomas elementų perstatymo metodas. Taip pat, įvykiu yra vadinama ir pilnai suformuluota žinutė apibūdinanti pokytį, tačiau jo aprašymo detalumas gali skirtis priklausomai nuo programuotojų pasirinkimo – to paties įvykio aprašymas gali būti „kažkas pasikeitė“ ir „masyvas buvo surūšiuotas didėjimo tvarka“.

Vieno įvykio gyvenimo ciklas susideda iš įvykio generavimo, formatavimo, perdavimo ir sunaudojimo. Įvykių generavimas gali būti atliekamas procesuose, duomenų saugyklose ar išoriniuose įrenginiuose. Kadangi yra daug įvykių šaltinių, jų formatai gali neatitikti sistemos standartų, todėl yra reikalingas papildomas žingsnis, kuriame nesuformatuota informacija yra perskaitoma ir sutvarkoma sistemai priimtinu būdu. Tada informacija yra patalpinama į reikiamus įvykių kanalus, iš kurių įvykį perima suinteresuotos šalys.

Paprastoje įvykiais pagrįstos architektūros sistemoje gali būti tik vienas įvykių kanalas, kuriame yra talpinami visi įvykiai, tačiau sudėtingesnėms yra privalomi keli. Norint išlaikyti informacijos prieinamumą ir saugumą bei palaikyti bendrą sistemos greitį ir tvarką yra patariama kurti specializuotus kanalus, kurie yra prieinami tik su ta sritimi susijusiems elementams. Paskutinis žingsnis yra įvykio sunaudojimas, tai įvyksta kai suinteresuota šalis iš kanalo pasiima įvykį, kad galėtų įgyvendinti tolimesnį sistemos veikimą arba peržiūrėti atliktų veiksmų rezultatus. Priklausomai nuo sistemos, įvykio gavėjas gali būti vienas (tokiu atveju, įvykis iš kanalo pašalinamas iš karto) arba keli (kiekvienas gavėjas nusikopijuoja įvykį kanale palikdami originalą ir po nustatyto laiko tarpo arba kai sistema gauna pranešimą, kad informaciją pasiėmė visos suinteresuotos šalys, įvykis yra pašalinamas).

1.2 Architektūros sandara

Įvykiais pagrįstoje architektūroje įvykiai yra laikomi trigeriais, kurie gali inicijuoti tolesnius vidinius ar išorinius procesus, kurių vykdymas gali būti visiškai nežinomas įvykį sukūrusiam procesui. Toks veikimas leidžia minimizuoti elementų sąryšį ir palaikyti tvarkingą likusios sistemos veiklą netgi išėmus arba pakeitus dalį įvykius apdorojančių procesų. Standartinėje sistemoje norint prijungti naują procesą reikia keisti struktūrą proceso, kurio informacija paveikia naująjį, tačiau EDA leidžia naują procesą prijungti prie reikiamų įvykių kanalų nekeičiant esamos sistemos. Pagrindiniai reikalavimai yra teisingai suformatuoti įvykio

informaciją ir turėti pakankamai įvykių kanalų, kad procesai galėtų greitai rasti jiems reikalingus įvykius. Kaip ir anksčiau minėta formatavimas yra svarbus norint užtikrinti teisingą informacijos nuskaitymą, todėl norint jį praleisti įvykį apdorojantis procesas turėtų būti pritaikytas prie kiekvieno šaltinio, kas reikalautų kiekvieną kartą pridedant naują įvykių generatorių perrašyti kiekvieno elemento apdorojančio jo generuojamus įvykius kodą – atsirastų sąsaja tarp procesorių ir įvykių generatorių. Tačiau norint užtikrinti maksimalų komponentų atsiejimą ir mažą kanalų apkrovą reikia sumažinti atgalinių kreipinių kiekį. Tai įgyvendinama įvykio perduodamų duomenų kiekio padidinimu ir duomenų bazės informacijos kopijų kūrimu. Įgyvendinant šį sprendimą reikia gerai apsvarstyti, kiek informacijos perduoti įvykio aprašyme ir kiek saugoti duomenų bazės kopijoje, priimant šį sprendimą, reikia įvertinti įvykio apdorojimo sudėtingumą ir kreipimosi į duomenų bazę greitį.

Standartinis įvykiais pagrįstos architektūros modelis turi penkias pagrindines dalis [Mic11]:

- metaduomenis yra įvykio trumpas aprašymas, sudarytas iš jo tipo, įvykusio pokyčio patikslinimo, sukūrimo laiko ir apdorojimo taisyklių. Įprastai apdorojimo taisyklės yra saugomos tik įvykių kontrolės įrankiuose (bus aptarti vėliau) ir susiejamos su įvykio objektu pagal jo tipą.

- įvykių procesoriai yra sistemos komponentai apdorojantys gautą informaciją ir ją perduodantys atitinkamiems sistemos funkcionalumo moduliams.

- įvykių perdavimo ir veiksmų kontrolės įrankiai įrašo įvykius į jiems pritaikytus kanalus, stebi jų panaudojimą, kanalų užimtumą ir renka statistinę informaciją apie įvykių apdorojimo greitį, procesų panaudojančių konkretų įvykį skaičių ir kiek kartų įvykis grįžta į kanalus po apdorojimo.

- įvykių perdavimo ir sistemos funkcionalumo sąsaja prižiūri ir užtikrina kanalų prieinamumą bei informacijos perdavimą, formatuoja įvykius ir sukuria terpę, kurioje kontrolės įrankiai gali lengviau stebėti įvykių judėjimą.

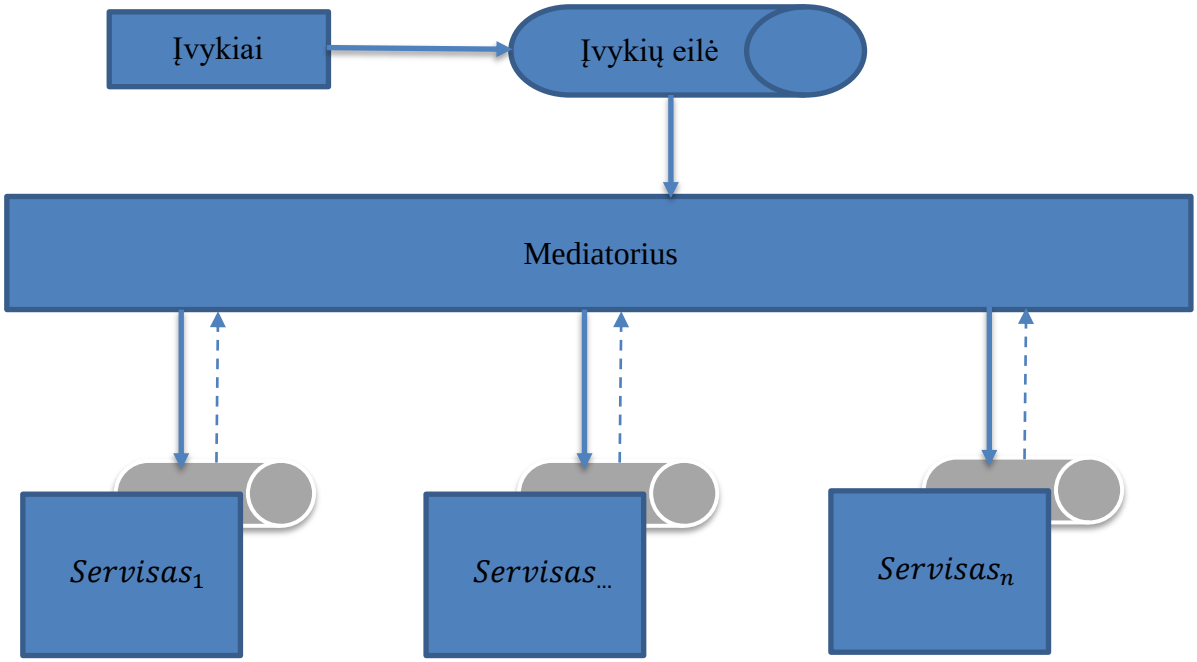
- sistemos resursai yra funkciniai moduliai, informacijos saugyklos, sistemą prižiūrintys darbuotojai ir įvykius generuojantis arba jų galutinius rezultatus apdorojantys elementai.

1.3 Įvykiais pagrįstos architektūros topologijos

Nors dauguma įvykiais pagrįstos architektūros sistemų turi tokias pačias pagrindines dalis, jų išdėstymas labai priklauso nuo pasirinkto įvykių valdymo stiliaus. Standartiniais stiliais, teorijoje dažnai vadinamais topologijomis, yra laikomi mediatoriaus (angl. *mediator*) ir tarpininko (angl. *broker*). Pagrindinis mediatoriaus topologijos bruožas yra centrinis valdymo komponentas, kuris vienintelis turi prieigą prie kelių įvykių kanalų. Tarpininko atveju, centrinio komponento nėra – todėl procesoriams leidžiama tiesiogiai apsikeisti informacija.

1.3.1 Mediatoriaus topologija

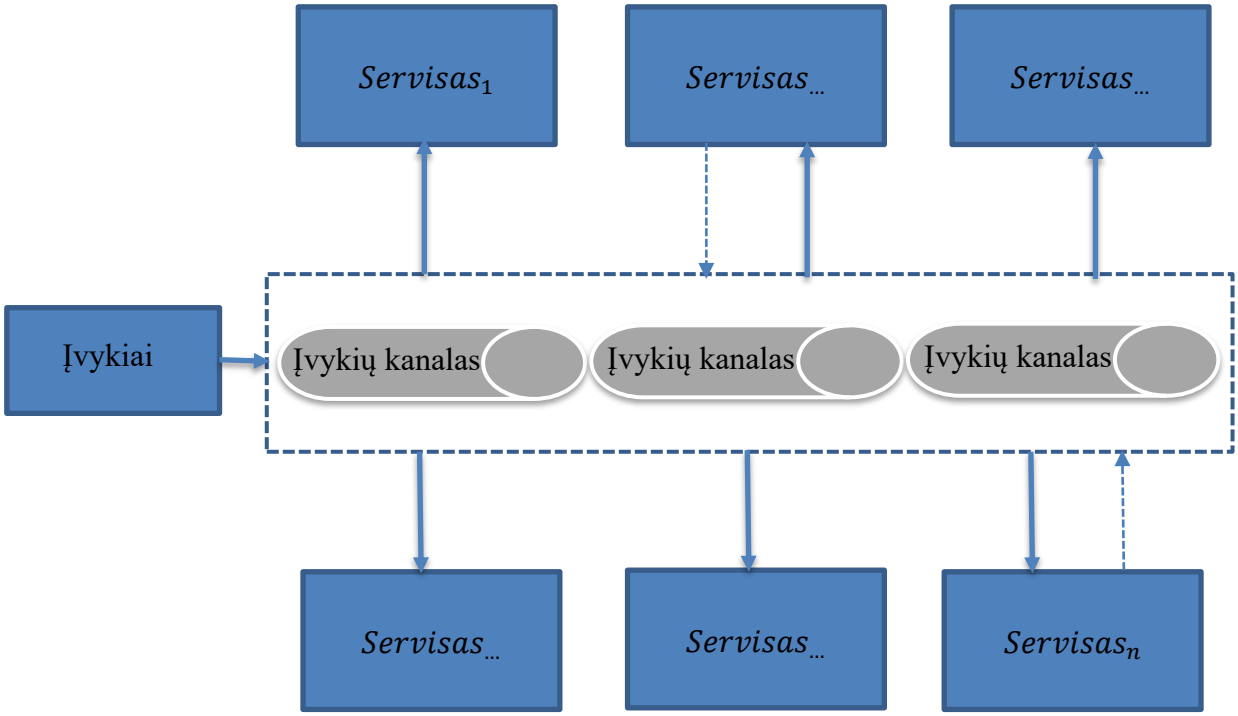
Mediatoriaus topologija skirta sudėtingiems įvykiams, kurie turi labai tikslią informacijos apdorojimo tvarką, todėl privalo būti iš šalies prižiūrima teisinga vykdymo eiga, kas leidžia dalį veiksmų atlikti lygiagrečiai ir dalį – nuosekliai. Ši paskirtis yra matoma ir iš topologijos sandaros, kurios centre yra valdymo elementas – mediatorius. jis gauna pradinį įvykį, kurį suskaido į mažesnius specializuotus įvykius ir perduoda juos procesoriams. Įvykių procesoriai yra antra iš keturių mediatoriaus topologijos dalių, juose yra apdorojami įvykiai ir įgyvendinama reikiama sistemos logika. Gautas atsakymas yra grąžinamas mediatoriui, kuris naują informaciją panaudoja kuriant įvykius priklausančius nuo gauto atsakymo ir tik tada patalpina informaciją į kitą kanalą. Likusios dvi topologijos dalys yra įvykių eilė ir kanalai, eilė yra naudojama mediatoriaus bendravimui su išore, o kanalai – su procesoriais. Pradiniai įvykiai yra patalpinami eilėje iš kurios mediatorius pasiima po vieną įvykį ir jį išskaido taip sukurdamas kelis vykdymo įvykius.



1 schema „Mediatoriaus topologija“ (pagal [Mil18])

1.3.2 Tarpininko topologija

Tarpininko topologija yra pritaikyta įvykiams, kuriems nėra svarbi vykdymo tvarka arba ji gali būti patogiai kontroliuojama pagal pranešimų tipą. Toks išdėstymas leidžiamas tik tada, kai procesai gaunantys tokio pačio tipo pranešimą gali veikti asinchroniškai. Kitaip nei mediatoriaus topologijoje, čia yra tik dvi dalys – tarpininkas ir įvykių procesoriai. Tarpininko tikslas yra talpinti sistemos įvykių kanalus, kad būtų paprastesnė kanalų priežiūra. Iš čia įvykių procesoriai turi pasiimti jiems aktualius įvykius ir apdorotą informaciją grąžinti į tarpininką. Todėl šioje topologijoje (kitaip nei mediatoriaus) yra labai svarbu procesoriuose parašyti teisingą ir greitai veikianti įvykių paėmimo algoritmą.



2 schema „Tarpininko topologija“ (pagal [Mil18])

1.4 Kokie yra realus įgyvendinimai

Nors teorijoje aprašyta įvykiais pagrįsta architektūra yra aiški ir patogi (nors gali būti šiek tiek sudėtinga realizuoti, apie ką pakalbėsime vėliau), bet realiuose sistemose retai pasitaiko pilnas jos įgyvendinimas. Šia architektūra paremtos sistemos įgyvendina bent viena iš trijų struktūrų:

- įvykių pranešimų (angl. *event notification*),
- įvykiais perduodamos būsenos kopijavimas (angl. *event-carried state transfer*)
- įvykių atsekamumas (angl. *event-sourcing*).

Įprastoje sistemoje norint paleisti servisą reikia žinoti jo informaciją, todėl keičiant vieną servisą kitu, jį šalinant, pridedant naują ar tiesiog keičiant iškvietimo informaciją reikia redaguoti ir pradinį procesą, kas didelėse sistemose gali būti sudėtinga. Įvykių pranešimų struktūra yra skirta šios problemos sprendimui – tiesioginis bendravimas tarp sistemos dalių pakeičiamas įvykių kanalu (kanalais). Kai sistemoje įvyksta pokytis, procesas į kanalą paskelbia įvykį ir tęsia savo darbą. Jei yra procesas, kuriam ši informacija aktuali, įvykio informacija nukopijuojama ir apdorojama, nebent yra trūkstamų duomenų – tada įvykio savininkui yra duodama užklausa ir darbas pratęsiamas gavus atsakymą. Šis požiūris yra

panašus į procesų bendravimą komandomis, tačiau čia yra paskelbiamas pokytis ir leidžiama sistemai nuspręsti, ką reikia atlikti vėliau, o dirbant su komandomis – komandos skelbėjas turi žinoti sekantį veiksmą ir jį nurodyti.

Įvykiais perduodamos būsenos kopijavimas gali būti naudojamas tik įgyvendinus įvykių pranešimus, nes jo veiklos principai yra pagrįsti prielaida, kad sistema bendrauja įvykiais. Pagrindinis struktūros tikslas yra sumažinti pradinio proceso gaunamų užklausų kiekį, todėl čia draudžiamos užklausos įvykio kūrėjui. Dėl šio draudimo įvykio pernešamos informacijos kiekis turi smarkiai padidėti, jame turi būti pažymima ne tik kas įvyko, bet ir kokie rezultatai gauti atlikus pirminį veiksmą, su kokia informacija dirbama bei kitą sistemos veikimui būtiną informaciją. Tarkime vartotojas pakeitė informacija saugoma sistemoje, tada įvykyje turi būti parašyta, kurio vartotojo informacija buvo pakeista, kokia buvo originaliai ir kokia yra nauja, bet net tokio pobūdžio pranešimas neužtikrina, kad visais atvejais procesai galės atlikti darbą be patikslinimų. Norint išvengti tokių situacijų kiekvienas procesas turi saugoti visą informaciją, kurios gali prireikti jo veikimui. Įvykiais perduodamos būsenos kopijavimo struktūroje yra stipriai sumažinama pagrindinio proceso apkrova, todėl jo veikimo sutrikimas turi maža poveikį sistemai, tačiau kopijų egzistavimas verčia daugiau dėmesio skirti duomenų vientisumo palaikymui.

Pagrindinis įvykių atsekamumo struktūros tikslas yra saugoti praeities įvykių informaciją, kas leidžia atkurti tiek darbinę, tiek istorinę sistemos būseną. Todėl „ … esminis įvykių atsekamumo testas [...] yra tas, kad bet kuriuo metu galime sunaikinti savo programos būseną ir užtikrintai atstatyti ją iš žurnalo“ (angl. *“... crucial test of event sourcing [...] is that at any time we can blow away our application state and confidently rebuild it from the log.”*) [Fow17a]. Tokios struktūros yra naudojamos daugumoje sistemų, programuotojams gerai pažįstamu, sistemos žurnalų (angl. *log*) pavidalu. Norint gebėti atkurti sistemos būseną yra būtina registruoti kiekvieną pokytį, o įvykiais pagrįstos architektūros atveju kiekvieną paskelbtą įvykį. Įvykių atsekamumas leidžia dirbti be duomenų bazės – visą reikalingą informaciją saugant darbinėje atmintyje, ši savybė yra vadinama atminties vaizdu (angl. *memory image*).

Dėl struktūros pobūdžio įvykių žurnalas yra laikomas tiesos šaltiniu, todėl registravimo procesas turi būti maksimaliai stabilus ir apsaugotas nuo klaidingo įvedimo. Tvarkingo žurnalo turėjimas leidžia atsekti EDA sistemos klaidas ir sutrikimų priežastis. Tai atliekama sistemos kopijai perduodant įvykius, kurie buvo paskelbti prieš iškylant problemai, taip atkuriant būseną,

kai įvyko klaida, ir tada stebėti kiekvieną įvykį. Tačiau yra sistemos atkūrimą apsunkinančių veiksnių. Vienas jų yra žurnalo ilgis – turint didelę ir/arba sudėtingą sistemą įvykių būna labai daug, o norint tiksliai atkurti sistemos būseną reikia paleisti visą žurnalą nuo pradžių. Šios problemos galima išvengti sukuriant sistemos atvaizdus (angl. *screenshot*), kurie išsaugo tarpines sistemos būsenas. Atvaizdai leidžia sumažinti vykdomų įvykių skaičių paimant artimiausią atkuriamai būsenai atvaizdą ir paleidžiant trūkstamus įvykius. Todėl dažnai yra saugomi ir atvaizdai, ir įvykių žurnalas. Taip pat, dažnai pasitaikanti klaida yra manymas, kad kiekvienas sistemos naudotojas (išorinė sistema ar žmogus) turi turėti prieigą prie žurnalo, kas padidina sistemos sudėtingumą. Tačiau šis požiūris yra klaidingas – naudotojai, kuriems nebūtina žinoti informacijos ir neturi gauti prieigos. Nepaisant problemų, kurių sprendimai yra pakankamai aiškus, yra svarbūs klausimai, kuriuos reikia apsvarstyti kuriant tokią sistemą:

1. Kaip elgtis su senų versijų įvykių žurnalais?
2. Ar sistema tikrai turi būti asinchronišką?

Versijavimo klausimas turi kelis pasiūlymus, bet nėra griežtų taisyklių kaip elgtis. Sąlyginai paprastas pasiūlymas yra kurti sistemą, kurios nepaveiktų seni įvykių duomenys, tačiau tokia situacija yra labai reta. Taip pat, yra patarimų saugoti sistemos atvaizdus prieš jai pasikeičiant ir tikrinti žurnalą jei tai yra būtina.

Įvykių atsekamumas daugumai programuotojų asocijuojasi su asinchroninių veikimu, tačiau tai nėra privaloma struktūros dalis. Asinchroniniai įvykiai prideda daug sudėtingumo sistemos kūrimui ir priežiūrai, todėl verta patikrinti jų būtinybę.

1.5 Į ką reikia atsižvelgti kuriant programas pagal EDA

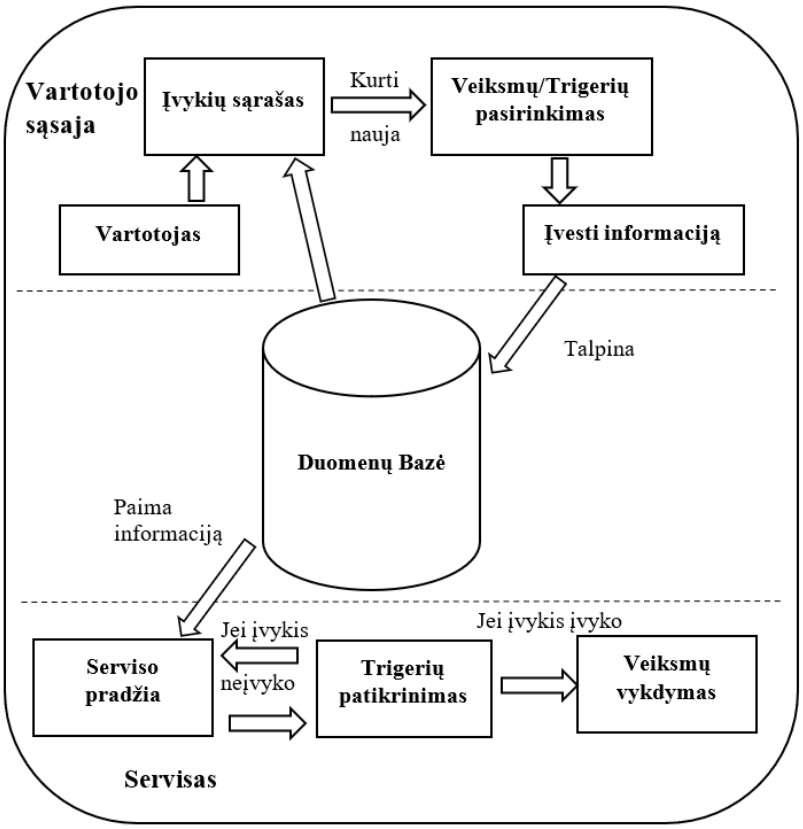
Įvykiais pagrįstos architektūros įgyvendinimas turi sunkumu, kurie yra matomi kuriant pilną įvykiais pagrįstą architektūros sistemą. Šioje architektūroje dalis privalumų yra sudėtingo įgyvendinimo priežastys. Viena jų yra asinchroninis veikimas (pagal pilnos EDA apibrėžimą tai būtina, kuriant dalinę – galima rinktis), kaip jau anksčiau minėta – sukurti asinchroninę sistemą yra sudėtinga. Ją kuriant reikia įvertinti galimas sistemos būsenas ir patikrinti, ar sistema veiks korektiškai visais atvejais. Tačiau sudėtingoje sistemoje numatyti visas būsenas gali būti beveik neįmanoma, todėl tikslingiau yra rasti problematines vietas ir jas apsaugoti nuo išorinio poveikio. Taip pat, sistemos išskirstymas sukuria papildomas funkcijas, kurių veikimą

reikia prižiūrėti, tokias kaip nuotolinis procesų prieinamumas, nutrūkusio ryšio atkūrimas. Šios funkcijos paveikia sistemos programavimo ir testavimo trukmę, nes reikia suprogramuoti atsargines kopijas ir ištestuoti ne tik pačia sistema, bet ir kaip ji veikia, kai bent vienas originalus procesas yra pakeistas atsargine kopija, bei kopijų prisijungimo greitį. Tačiau svarbiausia yra įsitikinti, kad kuriama sistema yra pritaikyta įvykiais pagrįstai architektūrai, pavyzdžiui, jei architektūra verčia nedalijimą transakciją išskaidyti į kelis procesus [Fow17b]. Kuriant sistemą būtinai reikia nustatyti įvykių formatą, kokių formatu bus įrašomas įvykių žurnalas ir patartinas iš anksto nustatyti kaip bus versijuojami įvykiai.

2. Automatizacijos programos veikimo schema

Peržiūrėjus teoriją ir susipažinus su dažniausiomis įvykiais pagrįstos architektūros struktūromis ir įgyvendinimo sunkumais, buvo sudarytas automatizacijos sistemos planas. Automatizacijos sistema leidžia naudotojui paruošti scenarijus, kurie bus vykdomi esant iš anksto nustatytoms sąlygoms, kurios gali būti mygtuko paspaudimas, buvimo vieta, laikas ar kitos priklausomai nuo sistemos galimybių. Čia aprašomos sistemos tikslas yra automatizacijos įgyvendinimui pasitelkti Android mobiliųjų telefonų galimybes.

Šiame darbe aprašomos sistemos vykdymo tvarka yra paprasta ir nereikalaujanti išorinės reguliacijos, todėl akivaizdus sprendimas buvo sistemą kurti pagal tarpininko topologija. Taip pat, tvarkingo mediatoriaus, kuris stebėtų ir rūšiuotų įrenginio sistemos įvykius, įgyvendinimas būtų buvęs panašaus sudėtingumo kaip ir visa likusi programa, kas apkrautų įrenginio sistemą nebūtinu sudėtingumu ir galimai sulėtintų programos veikimą. Iš čia yra matoma, kad tarpininko topologijos pakanka šios sistemos įgyvendinimui, o mediatoriaus buvimas nesukurtu pakankamai naudos, kad papildomas sudėtingumas būtų pateisinamas.



3 Schema „Darbe aprašomos programos schema“

Pradinėje versijoje buvo įgyvendintas labai ribotas trigerių ir veiksmų skaičius, tačiau programos struktūros tai smarkiai nepakeičia. Čia aprašomos sistemos komponentės tarpusavyje bendrauja tik duomenų bazės pagalba – nėra tiesioginių kreipinių tarp pirmosios komponentės (vartotojo sąsajos) ir antrosios (užkulisinio (angl. *background*) serviso). Vartotojo sąsaja įrašo įvestą informaciją į duomenų bazę, o servisas – apdoroja išsaugotą informaciją ir pagal tai atlieka vartotojo nurodytus veiksmus. EDA yra sutelkta į serviso veikimą, ypač trigerių tikrinimo elementą. Todėl didžioji dalis įvykių yra gauti iš įrenginio sistemos per metodus, kurie seka įrenginio būsenos atnaujinimus. Tokie metodai šiame darbe aprašomoje sistemoje yra naudojami buvimo vietos ir įkrovos pokyčių stebėjimui, tačiau jų galima turėti daug daugiau.

Didelė dalis sistemų, su kuriomis susipažinome, patiria sunkumų įgyvendinant EDA, nes jos yra asinchroninės, tačiau šioje sistemoje buvo galima išvengti problemų kylančių su asinchroninėmis, nes dabar įgyvendinta sistema teisingai veikia ir būdama sinchroninė. Norint tai išlaikyti, sistema buvo įgyvendinta taip, kad vienintelis bendras resursas būtų duomenų bazė ir abu komponentai vienu metu nekeistu to paties įrašo. Ši savybė leidžia išvengti klaidų susijusių su asinchroninėmis sistemomis.

Antras, ne ką mažiau pasitaikantis klaidų šaltinis yra sistemos komponenčių prieinamumo užtikrinimas. Tai yra labai svarbu sistemoms, kurios yra patalpintos debesyje, išskirstytos per kelis įrenginius arba apjungia kelias programas tame pačiame įrenginyje. Kadangi aprašoma sistema šiuo metu yra maža, ji yra pritaikyta darbui su vienu įrenginiu ir talpina visą informaciją įrenginio viduje, todėl turi labai mažą tikimybę susidurti su komponenčių neprieinamumu.

Prieš pradedant dirbti ties sistemos įgyvendinimų reikėjo įsitikinti, kad jai tinka EDA. Pagrindinis kriterijus yra, ar architektūra neverčia nedalomų transakcijų padalinti į kelis procesus. Tačiau automatizacijos sistemų pobūdis nereikalauja transakcijų buvimo, todėl sistemoje buvo pasirinkta jų nenaudoti. Plečiant sistemą galėtų atsirasti įvykių, kuriems transakcijos yra būtinos, tačiau šį sprendimą galima atidėti laikui kai bus tiksliai žinoma, ar tokia struktūra reikalinga.

3. Android galimybių analizė ir kodo pavyzdžiai

Sistemos įgyvendinimui buvo pasirinkta „Android Studio" programavimo aplinka, todėl šis skyrius bus paremtas jos dokumentacija ir kitais tiesiogiai su darbu šioje aplinkoje susijusiais šaltiniais.

Prieš pradedant šį darbą buvo išnagrinėtas panašių programų veikimas ir pastebėjus, kad daug iš jų turi trūkumų, buvo nuspręsta sukurti naują sistemą, kurį galėtų paaiškinti trūkumų atsiradimą. Dažniausios klaidos buvo susijusios su užkulisiniu programų veikimu – išjungus aplikaciją, nebeveikdavo nustatyti veiksmai. Todėl buvo nuspręsta šios klaidos priežastį išsiaiškinti pirmiausiai. Atsakymas buvo rastas peržiūrėjus Android veikimo užkulisiuose patarimus „Android Studio" dokumentacijose – paskutinėje versijoje buvo įvestas serviso veikimo limitas, todėl vienintelis būdas išlaikyti šią funkciją buvo sukurti nepašalinamą pranešimą informuojantį vartotoją apie serviso veikimą [And20a].

Taip pat, buvo programų, kurios reikalavo atsisiųsti papildomus to paties kūrėjo servisus, kad teisingai veiktų norima funkciją, bet ne visi nurodydavo, kokį servisą reikia atsisiųsti. Nors tai negali būti laikoma klaida, tačiau toks elgesys gali šiek tiek erzinti, ypač kai reikia atspėti, ko programai trūksta, kad galėtų įvykdyti norimą funkcionalumą.

Kita dažnai pasitaikanti savybė yra sudėtingas arba klaidus vartotojo sąsajos dizainas, kuris gali svyruoti nuo sudėtingo, mažai su technologijomis pažįstamiems žmonėms, iki aiškaus, tik su programavimu susidūrusiems. Nors šiais laikais dauguma turi gerus išmaniųjų technologijų gebėjimus, tačiau negalima pamiršti kitų naudotojų, kurie yra mažiau pažįstami su šia sritimi.

Taip pat, tiriant dabartinę automatizacijos sistemų rinką buvo pastebėta, kad programų, kurios orientuojasi į mobiliųjų telefonų automatizavimą, nėra daug ir dar mažiau iš jų veikia teisingai. Naujesnės aplikacijos daugiau dėmesio skiria socialinių tinklų/elektroninio pašto automatizavimui arba telefono sujungimo su išmaniais namų apyvokos įrenginiais ar automobilių elektroninėmis sistemomis supaprastinimui, todėl buvo nuspręsta bandyti sukurti sistemą, kuri leistų patogiau ir efektyviau naudotis mobiliuoju telefonu.

3.1 Veiksmų/Trigerių analizė

Šioje sistemoje veiksmais yra laikomos vartotojo pasirinktos funkcijos, kurias turi įvykdyti programa, o trigeriais – veiksmų įvykdymo sąlygos. Pavyzdžiui, sakinyje „atvykus į universitetą reikia išjungti telefono garsą", programa pasakymą „atvykus į universitetą" laiko trigeriu, o „išjungti telefono garsą" – veiksmu.

3.1.1 Įgyvendinti veiksmai ir trigeriai

Dabartinėje sistemoje yra tris funkcijos, kurias leidžiama naudoti tik kaip trigerius: data ir laikas, baterijos įkrovos lygis bei buvimo vieta. Taip pat, trys funkcijos, kurios gali būti naudojamos tik kaip veiksmai: trumpųjų žinučių siuntimas, pranešimų skelbimas ir nurodytos programos paleidimas. Galiausiai yra dar trys, kurias galima naudoti kaip trigerį ir kaip veiksmą: garso, šviesumo ir neveiklumo laiko nustatymai.

3.1.1.1 Laiko ir datos trigeris

Laiko ir datos trigerio veikimas yra pakankamai paprastas – sistema periodiškai tikrina, ar esamas laikas ir data yra tokie, kuriuos nustatė vartotojas. Iš vartotojo sąsajos dalies tai yra įgyvendinta panaudojant specializuotus dialogo langus, skirtus laiko (TimePickerDialog.OnTimeSetListener) [And20b] ar datos (DatePickerDialog.OnDateSetListener) [And20c] pasirinkimui. Jie abu automatiškai pakeičia aukštesnio lygio lauko reikšmę perėjus per visus prieš tai buvusio pasirinkimus (pavyzdžiui, perėjus visas vieno mėnesio dienas mėnesio pasirinkimo lauke reikšmė yra padidinama vienetu). Datos pasirinkimo dialogas yra užprogramuotas taip, kad rodytų teisinga dienų skaičių priklausomai nuo metų ir mėnesio.

```
    public void ClickButtonDisplayDate/Time(View view) {
        Date/TimePickerDialog dialog = new Date/TimePickerDialog(…);
        dialog.getWindow().setBackgroundDrawable(…);
        dialog.show();
    }
    private void ClickButtonDisplayDate/Time2() {
        date/timeListener = new Date/TimePickerDialog.OnDate/TimeSetListener() {
            @Override
            public void onDate/TimeSet(…)
{((TextView)findViewById(R.id.idPavadinimas)).setText(String.format("", kint));}
        };
    }
```

Trigerio tikrinimas įvykdytas pasinaudojant įrenginio kalendoriumi, šią informaciją gaunant metodo Calendar.getInstance() pagalba, tada gauta informacija yra sulyginama su vartotojo nustatyta.

```
    date = String.format("%02d/%02d/%d",
Calendar.getInstance(TimeZone.getDefault()).get(Calendar.DAY_OF_MONTH),
(Calendar.getInstance(TimeZone.getDefault()).get(Calendar.MONTH) + 1),
Calendar.getInstance(TimeZone.getDefault()).get(Calendar.YEAR));
    time = String.format("%02d : %02d",
Calendar.getInstance(TimeZone.getDefault()).get(Calendar.HOUR_OF_DAY),
Calendar.getInstance(TimeZone.getDefault()).get(Calendar.MINUTE));
```

3.1.1.2 Baterijos įkrovos trigeris

Baterijos įkrovos lygio trigeris turi dvi dalis, pirmoji leidžia pasirinkti, ar įrenginys yra kraunamas, antroji – koks yra įkrovos lygis procentais. Vartotojo sąsajoje buvo panaudoti vieno pasirinkimo mygtukai (angl. *radio buttons*) pirmajai daliai [And20d] ir paieškos juosta (angl. *seek bar*) antrajai [And20e]. Pasirinkimo mygtukai būna sujungti į sinchronizuota grupe RadioGroup [And20f], kurioje vienas mygtukas visada privalo būti pasirinktas. Ši sąlyga apsaugo nuo klaidingo įvedimo. Paieškos juostos informacija yra nuskaitoma panaudojant getProgress() metodą, kuris grąžina procentinę paieškos juostos reikšmę, ir gauta informacija yra įrašoma į tekstinį lauką šalia juostos.

```
((SeekBar)findViewById(R.id.Phone_Battery_Level_SeekBar)).setOnSeekBarChange
Listener(new SeekBar.OnSeekBarChangeListener() {
        @Override
        public void onProgressChanged(SeekBar seekBar, int progress, boolean
    fromUser) {
            ((TextView)findViewById(R.id.Battery_Level)).setText(progress + " %");
        }
        …
});
```

Servise buvo naudojamas baterijos tvarkytojas (angl. *battery manager*) ir baterijos pokyčių informacijos gavėjas (ACTION_BATTERY_CHANGED) duomenų prieigai [And20g]. Jis yra panaudojamas nustatant telefono krovimo būseną bei baterijos lygį ir mastelį. Krovimo būsena yra sulyginama su vartotojo pasirinkta ir panaudojama nuspręsti, kaip elgtis toliau. Sąlyga laikoma patenkinta, kai esamas baterijos lygis procentais yra nemažesnis už vartotojo pasirinktą (kai įrenginys kraunasi) arba nedidesnis (kai įrenginys nesikrauna). Šiuo atveju lygybė negali būti naudojama dėl tikimybės, kad nebus spėta sureaguoti į naudotojo nurodytą įkrovos lygį.

```
IntentFilter ifilter = new IntentFilter(Intent.ACTION_BATTERY_CHANGED);
Intent batteryStatus = context.registerReceiver(null, ifilter);
int status = batteryStatus.getIntExtra(BatteryManager.EXTRA_STATUS, -1);
boolean isCharging = status == BatteryManager.BATTERY_STATUS_CHARGING ||
                     status == BatteryManager.BATTERY_STATUS_FULL;
int level = batteryStatus.getIntExtra(BatteryManager.EXTRA_LEVEL, -1);
int scale = batteryStatus.getIntExtra(BatteryManager.EXTRA_SCALE, -1);
float batteryProc = level * 100 / (float)scale;
```

3.1.1.3 Vietos trigeris

GPS trigeris lygina buvimo vietą su vartotojo pasirinkta. Jis buvo įgyvendintas naudojant Google APIs siūlomais elementais: žemėlapio fragmentų (angl. *map fragment*) [Goo20a] – vartotojo sąsajai ir Places API [Goo20b]– abiejuose, nes jis leidžia nuskaityti vartotojo pasirinktą vietą žemėlapyje, gauti buvimo vietos atnaujinimus ir paprastai apskaičiuoti atstumą tarp jų.

Vietos pasirinkimo langas buvo perkeltas į iššokantį dialogą, kuriame žemėlapis atvaizduotas panaudojus jau minėtą žemėlapio fragmentą ir Google APIs sugeneruota API raktą, vartotojo patogumui, buvo pažymėta įrenginio buvimo vieta panaudojant Marker klasė. Vartotojo norimos vietos pasirinkimui buvo naudotas OnMapClick metodas ir vieta pažymėta atskiru Marker klasės objektu. Jei naudotojas nori pakeisti savo pasirinkimą jam reikia tiesiog paspausti kitą vietą. Kai vartotojas bus patenkintas savo pasirinkimu yra paspaudžiamas dialogo lango apačioje esantis „Ok“ mygtukas, taip išsaugant paskutinio žymeklio koordinates.

```
GoogleMap.setOnMapClickListener(new GoogleMap.OnMapClickListener() {
    @Override
    public void onMapClick(LatLng coordinate) {
        chosenMarker = GoogleMap.addMarker(new
MarkerOptions().position(coordinate).title("Pasirinkta vieta"));
        GoogleMap.moveCamera(CameraUpdateFactory.newLatLngZoom(coordinate,
15));
    }
});
placesClient.findCurrentPlace(request).addOnSuccessListener((new
OnSuccessListener<FindCurrentPlaceResponse>() {
    @Override
    public void onSuccess(FindCurrentPlaceResponse response) {
        coordinateCurrent =
response.getPlaceLikelihoods().get(0).getPlace().getLatLng();
    }})).addOnFailureListener(new OnFailureListener() {...});
```

Vietos trigerio tikrinimui yra naudojamas programos viduje apibrėžtas vietos artumo dydis, kuris yra sulyginimas su atstumu tarp dabartinės įrenginio buvimo vietos ir naudotojo pasirinktos. Atstumas matuojamas tiesia linija, neatsižvelgiant į kelius tarp šių vietų. Dabartinės buvimo vietos gavimui yra naudojamas vienas iš dviejų būdų: GPS informacijos tiekėjas (GPS_PROVIDER) arba interneto tiekėjas (NETWORK_PROVIDER). Juose prašant nurodyti paskutinę žinomą vietą (metodas getLastKnownLocation). Visada stengiamasi naudoti pirmesni dėl jo patiekiamos informacijos tikslumo.

```
LocationManager locationManager = (LocationManager)
context.getSystemService(Context.LOCATION_SERVICE);
if(Tikrina, ar buvo suteiktas leidimas){//Jei ne, prašo dar kartą}
else {
Location location =
locationManager.getLastKnownLocation(LocationManager.GPS_PROVIDER);
Location locationNetwork =
locationManager.getLastKnownLocation(LocationManager.NETWORK_PROVIDER);
}
```

3.1.1.4 Trumpųjų žinučių siuntimo veiksmas

Trumpųjų žinučių siuntimo veiksmas sukuria žinutę pagal vartotojo įrašytą informaciją ir ją išsiunčia. Informaciją vartotojas įrašo į du redaguojamus tekstinius laukus. Pirmasis iš jų yra skirtas telefono numerio įvedimui ir privalo būti netuščias. Tuo tarpu antrasis, skirtas žinutės tekstui, gali būti tuščias. Šis laukas neapribotas dėl Android sistemos leidimo programiškai siųsti tuščias trumpąsias žinutes, net jei įrenginio žinučių sistema to neleidžia.

Trumposios žinutės siuntimui yra naudojama SMSManager [And20h] klasė, kurioje yra metodas leidžiantis sukurti ir išsiusti trumpąją žinutę metodui perduodant gavėjo telefono numerį ir norimą išsiusti tekstą.

```
SmsManager.getDefault().sendTextMessage(TelefonoReikšmė, null, SMSReikšmė, null,
null);
```

3.1.1.5 Pranešimų skelbimo veiksmas

Pranešimų skelbimo veiksmas skelbia pranešimą (angl. *notification*) su nurodytu tekstu. Vartotojas pranešimo tekstą įrašo į redaguojamą tekstinį lauką, kuris vykdant veiksmą bus įstatytas į pranešimo šabloną ir paskelbtas panaudojant NotificationCompat [And20i] klasę.

```
NotificationCompat.Builder builder = new
NotificationCompat.Builder().ReikalingiDuoemnysPranešimui;
NotificationManagerCompat notificationManager =
NotificationManagerCompat.from();
notificationManager.notify(id++, builder.build());
```

3.1.1.6 Programos paleidimo veiksmas

Programos paleidimo veiksmas automatiškai paleidžia nurodytą programą, kai yra įvykdytos pasirinktos sąlygos. Vartotojas pasirenka programą iš sąrašo, kuris yra užpildytas aplankų tvarkytojo (angl. *package manager*) [And20j] pagalba – jis yra naudojamas ir veiksmo atlikimui. Programų pasirinkimo sąrašui užpildyti buvo panaudotas dialogo langas ir metodas setSingleChoiceItems(), kuriam buvo paduotas sąrašas programų pavadinimų ir jų piktogramų. Šiuo atveju yra saugomas ne programos pavadinimas, o aplankas, kuriame tą programa yra, norint supaprastinti jos radimą įrenginyje.

```
Intent intent =
packageManager.getLaunchIntentForPackage(PasirinktosProgramosReikšmė);
intent.addCategory(Intent.CATEGORY_LAUNCHER);
context.startActivity(intent);
```

3.1.1.7 Garso pakeitimo veiksmas arba trigeris

Garso pakeitimo veiksmas/trigeris nustato vartotojo nurodytą garsumą arba patikrina, ar esamas yra lygus vartotojo pasirinktam. Vartotojo pasirinkimui yra naudojama paieškos juosta ir jos metodas getProgress(), kuris yra paminėtas anksčiau, o įgyvendinimui – garso tvarkytojo (angl. *audio manager*) [And20k] setStreamVolume() metodas veiksmui įgyvendinti ir getStreamVolume() trigeriui.

3.1.1.7 Šviesumo pakeitimo veiksmas arba trigeris

Šviesumo veiksmas nustato vartotojo pasirinktą ekrano šviesumą, o trigeris – tikrina, ar vartotojo nurodytas lygus esamam. Vartotojo sąsajoje yra naudojama paieškos juosta ir jos metodas getProgress(), o servise – sistemos nustatymų klasę Settings.System [And20l], naudojant putInt() veiksmui ir getInt() – trigeriui.

```
Settings.System.putInt(context.getContentResolver(),
Settings.System.SCREEN_BRIGHTNESS_MODE,
Settings.System.SCREEN_BRIGHTNESS_MODE_MANUAL);

Settings.System.putInt(context.getContentResolver(),
Settings.System.SCREEN_BRIGHTNESS, (255 * ŠviesumoReikšmė / 100));
```

3.1.1.8 Neveiklumo laiko veiksmas arba trigeris

Neveiklumo laiko trigeris/veiksmo pasirinkimas vartotojo sąsajoje yra sąrašas, kuriame vartotojas gali pasirinkti ties kokiu neveiklumo laiku/kokį nustatyti vykdant trigerį/veiksmą. Ekrano neveiklumo laiko įvedimui buvo panaudotas dialogo langas užpildytas nustatytų pasirinkimų sąrašu. Šis langas buvo sudarytas naudojant tokį patį metodą kaip ir programos pasirinkimo langas.

Čia yra naudojami tie patys metodai kaip ir šviesume, tik naudojant kitas reikšmes – yra parametras, kuris nurodo, į kurį nustatymą įrašyti nurodyta skaičių.

```
Settings.System.putInt(context.getContentResolver(),
Settings.System.SCREEN_OFF_TIMEOUT, NeveiklumoLaikoReikšmė);
```

3.1.2 Galimi plėtiniai

Dabar įgyvendintos funkcijos nėra vienintelės, kurios gali būti naudojamos tokioje sistemoje, didelė dalis mobilaus įrenginio funkcionalumo gali būti automatizuota, todėl apribojimai atsiranda tik dėl naudingo ir logiško panaudojimo neradimo. Dabar bus pateiktas

veiksmas ir pora trigerių, kuriuos būtų galima panaudoti plečiant sistemą: žadintuvo įjungimas, atėjusių trumpųjų žinučių arba pranešimų šaltinis ir tekstas, mobiliojo įrenginio išorinių mygtukų (garso reguliavimo ar užrakinimo) paspaudimai.

3.1.2.1 Žadintuvo įjungimo veiksmas

Žadintuvo jungimas gali būti naudojamas kaip veiksmas pasitelkiant AlarmClock [And20m] klasę. Ši funkcija leistų nustatyti žadintuvą jei yra įvykdytos nurodytos sąlygos. Pavyzdžiui, žmogui, kuris daug keliauja, skirtingose vietose reikia naudoti skirtingus žadintuvus, todėl norint neužmiršti pakeisti skambėjimo laiko galima nustatyti, kad žadintuvo laikas pasikeistų priklausomai nuo to, kur yra naudotojas.

3.1.2.2 Trumpųjų žinučių ir pranešimų skaitymo trigeris

Trumpųjų žinučių ir pranešimų skaitymą galima laikyti vienu trigeriu, nes trumposios žinutės sukuria santrauką, kuria įrašo į būsenos juostą (angl. *status bar*). „Android Studio“ klasė NotificationListenerService [And20n] leidžia perkaityti pranešimų šaltinį (tai yra programą sukūrusią pranešimą), pavadinimą/siuntėją (trumpųjų žinučių atveju) ir pranešimo tekstą. Todėl atsiranda septyni pranešimų skaitymo variantai:

- skaityti tik šaltinį, pavadinimą/siuntėją arba tekstą
- skaityti poras: šaltinis ir pavadinimas/siuntėjas, šaltinis ir tekstas, pavadinimas/siuntėjas ir tekstas
- skaityti visas tris dalis – šaltinį, pavadinimas/siuntėją ir tekstą,

3.1.2.3 Mobiliojo įrenginio išorinių mygtukų paspaudimų trigeris

Mobiliojo įrenginio išorinių mygtukų paspaudimai gali būti panaudojami programoje, kaip papildoma įvestis panaudojant klasę MediaButtonReceiver [And20o]. Nors dauguma mobiliųjų telefonų dabar turi sistemiškai įgyvendintas panašaus tipo funkcijas, tačiau jų galimybės yra labai ribotos. Šio trigerio įvedimas suteiktų galimybę praplėsti jų panaudojamumą.

3.2 Programinio kodo stabilumas

Dabar greitai vystosi technologijų sritis, todėl yra akivaizdu, kad keičiasi ir Android galimybės, tačiau tai gali apsunkinti programų kūrimą ir funkcionalumo palaikymą. Bandydama prisitaikyti prie pokyčių taip elgiasi ir „Android Studio" programavimo aplinka. Kai kurie jos metodai yra keičiami beveik su kiekviena nauja versija, daugiausiai pokyčių buvo atlikta vartotojo sąsajos funkcionalume. Taip pat, daug pokyčių buvo įvykdyta su GPS apdorojimu. Tačiau dabar, po migracijos į Google APIs siūlomą servisą, galima tikėtis pokyčių mažėjimo.

Kita kodo pokyčių priežastis yra įstatymų pokyčiai, programavimo požiūriu aktualiausi yra susiję su asmens duomenų apsauga. Dėl jų griežtinimo buvo pradėti naudoti vartotojo sutikimai, kurių skaičius ateityje gali didėti, tačiau tai nėra labai aišku.

Paskutinė pokyčių priežastis yra programavimo aplinkos kūrėjų bandymas supaprastinti naudojimąsi sistema. Dėl šios priežasties atsiranda metodai, kurie apjungia esamus, sistemos metodus. Tokie metodai gali būti dažniau keičiami, bet yra vieninteliai nuo kurių pokyčių galima apsisaugoti, naudojant metodus, iš kurių jie buvo sukurti.

4. Automatizacijos programos apžvalga

Automatizacijos sistemų tikslas – bet kurio įrenginio veikimo momentu atpažinti naudotojo iš anksto nustatytą įvykį ir pagal jį įvykdyti nurodytą veiksmą, todėl didelę sistemos dalį sudaro užkulisinis servisas. Šiame darbe aprašyta sistema yra sudaryta iš dviejų dalių – vartotojo sąsajos ir užkulisinio serviso – kurios tarpusavyje nebendrauja ir vienintelis jų sąryšis yra duomenų bazė.

4.1 Vartotojo sąsaja

Vartotojo sąsaja – grafinė programos išraiška skirta supaprastinti vartotojo naudojimosi sistema. Šioje sistemoje su vartotojo sąsaja susietas programinis kodas naudojamas vartotojo įvedamos informacijos paruošimui įvesti į duomenų bazę. Grafiniam atvaizdavimui sukurti buvo naudojamas XML formatas, o apdorojančiam kodui – JAVA programavimo kalba

Vartotojo sąsajos kūrimas buvo pradėtas nuo pagrindinio lango (priedas 1), kuris turi dvi pagrindines dalis: sąrašo elementą paruoštą vartotojo įvestų trigerių ir veiksmų išrašymui ir slankiojantį mygtuką (angl. *floating button*), skirtą naujų įvykių kūrimui. Sąrašo elementas yra pritaikytas tvarkingai išrašyti bet kokios trigeriu bei veiksmų kombinacijos informaciją ir gali reaguoti į paspaudimus atidarydamas langą su detalia informacija apie pasirinktą įvykį.

Detalios informacijos/redagavimo (priedas 2) lange galima keisti norimus duomenis arba įvykį ištrinti. Tokios pačios formos langas yra naudojamas ir sukurti naujam įvykiui, pakeičiant redagavimo mygtuką saugojimu ir trynimo – atšaukimu. Tačiau kuriant naują įvykį atskirame lange reikia pasirinkti trigerius ir veiksmus. Pasirinkimo (priedas 3) ir informacijos įvedimo (priedas 4) langai sujungti panaudojant vaizdo puslapiavimą (angl. *view flipper*), todėl informacijos įvedimo lange yra matomi laukai susiję tik su pasirinktais trigeriais ir veiksmais, kai tuo tarpu redagavime yra matomi visų galimų pasirinkimų informacijos įvedimo laukai ir galima tiesiogiai pakeisti prieš tai buvusių trigerių ir veiksmų pasirinkimą.

Norint užtikrinti sėkmingą programos veikimą buvo prireikta prašyti vartotojo leidimų. Pirmasis buvo skirtas trumpųjų žinučių siuntimui (SEND_SMS).

```
if (ActivityCompat.checkSelfPermission(this, Manifest.permission.SEND_SMS) ==
    PackageManager.PERMISSION_DENIED) {
        ActivityCompat.requestPermissions(this, new
String[]{Manifest.permission.SEND_SMS}, 0);
        if (ActivityCompat.checkSelfPermission(this, Manifest.permission.SEND_SMS)
==
            PackageManager.PERMISSION_DENIED){//---------Patikslinama leidimo
    prašymo priežąstis}
}
```

Antrasis skirtas gauti įrenginio buvimo vietai (ACCESS_FINE_LOCATION). Jie buvo gauti panaudojant sisteminį metodą requestPermission, kuris sugeneruoja dialogo langą prašantį naudotojo patvirtinimo.

```
if (ActivityCompat.shouldShowRequestPermissionRationale((Activity) context,
ACCESS_FINE_LOCATION)) {}
else{
        ActivityCompat.requestPermissions((Activity) context, new
String[]{ACCESS_FINE_LOCATION}, 1);
}
```

Trečiasis leidimas buvo panaudotas sisteminių nustatymų rašymui (WRITE_SETTINGS). Norint jį gauti vartotojas yra nukreipiamas į sistemos parametrų langą. Šis leidimas yra būtinas norint keisti ekrano šviesumą ir neveiklumo laiką.

```
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M &&
!Settings.System.canWrite(getApplicationContext())) {
        Intent intent = new Intent(Settings.ACTION_MANAGE_WRITE_SETTINGS);
        intent.setData(Uri.parse("package:" +
getApplicationContext().getPackageName()));
        startActivity(intent);
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M &&
!Settings.System.canWrite(getApplicationContext())){//---------Patikslinama leidimo
prašymo priežąstis}
}
```

Vartotojo įvesta informacija yra patikrinama ieškant klaidingų įvedimų, tokių kaip tuščias telefono numerio laukas (kai pasirenkamas trigeris susijęs su trumposiomis žinutėmis) arba pasirinkus baterijos įkrovos trigerį nustatyti vieno procento įkrovos lygį. Kai informacija atitinka reikalavimus ir yra patvirtinta, ji yra suformatuojama ir įrašoma į duomenų bazę. Informacija yra saugoma dviejuose lentelėse: vienoje yra parašomas įvykio vardas ir indeksas, antroje – įvykio informacija ir veiksmų bei trigetių tvarka.

4.2 Užkulisinis servisas

Užkulisinis servisas yra užduočių seka vykdoma ant atskiros gijos nei vartotojo sąsaja tam, kad ilgesni skaičiavimai nekliudytų vartotojo sąsajos veikimui. Užkulisinis servisas gali veikti tik kol aplikacija yra aktyvi arba nuolatos priklausomai nuo sistemos poveikių. Servisas buvo sukurtas JAVA programavimo kalbą.

Čia aprašomu atveju buvo pasirinktas nuolatinio veikimo servisas. Jame yra periodiškai perskaitoma duomenų bazė, iš kurios yra gaunami įvykiai, kuriuos reikia atlikti pirmiausia. Jie yra atliekami patikrinant ar kuris nors iš vartotojo įvestų trigerių buvo įgyvendintas. Jei buvo rasta patenkinta sąlyga yra įvykdomi su trigeriu susiję veiksmai ir servisų veikimas prasideda nuo pradžių, kitu atveju serviso veikimas prasideda nuo pradžių iš karto po trigerio patikrinimo.

Norint sukurti savarankišką servisą, reikia naudoti sisteminę funkciją startForeground, kurioje yra įrašomas serviso indeksas ir jo pavadinimas. Ši funkcija buvo neseniai pakeista todėl pirmas bandymas sukurti tvarkingai veikianti servisą buvo nesėkmingas. Taip pat, pagal dabartinius Android standartus, vartotojas turi nuolatos žinoti apie servisų veikimą. Tai užtikrinama sukuriant neištrinama pranešimą.

4.3 Naujų komponentų prijungimas prie kodo

Šiuo metu sukurta sistema yra palyginti nedidelė, todėl yra svarbu sukurti pagrindus, kurie leistu lengvai įtraukti naujas galimybes, tai buvo stengtasi minimizuoti kiekį elementų, kuriuos reikia keisti. Nors sistemos plėtimo metu svarbiausia yra sukurti tvarkingai veikianti funkcionalumą, tačiau vartotojui akivaizdžiausi yra vartotojo sąsajos pokyčiai. Tai verčia sistemą paruošti galimam jos plėtimui, nuo pat jos kūrimo pradžios. Čia aprašoma sistema yra viena tų, kurioms yra vienodai svarbu vartotojo sąsajos ir funkcionalumo įgyvendinimas.

Perėjus per kelis vartotojo sąsajos variantus buvo atrastas toks vartotojo sąsajos kūrimo būdas, kuris pridedant atnaujinimus reikalauja mažiausiai pakeitimų kode. Taigi naujo trigerio/veiksmo įtraukimas susideda iš dviejų dalių:

- sukurti veiksmui pritaikytą grafinio išdėstymo failą, kurį reikia įtraukti į detalios informacijos langą
- sukurti vartotojo įvedamos informacijos metodą

Tačiau vien šito neužtenka, reikia nepamiršti naują komponentą prijungti prie programos serviso. Priklausomai ar naujas komponentas yra veiksmas ar trigeris, yra sukuriamas naujas kodo fragmentas. Jei yra įtraukiamas trigeris reikia sukurti su tuo įvykiu/trigeriu susijusi stebėtoją (angl. *listener*), veiksmo atveju – sukurti metodą, kuris įgyvendintu nurodytą veiksmą ir šį metodą prijungti prie trigerių, taip kad suveikus su šiuo veiksmu vartotojo susietam trigeriui būtų nurodyta užduotis.

Rezultatai ir išvados

Šiuo darbo atlikimo metu susipažinome su įvykiais pagrįsta architektūra, jos topologijomis ir struktūromis. Sukūrėme automatizacijos sistemos schemą paremtą įvykiais pagrįsta architektūra ir jos prototipą įgyvendinome Android operacinėje sistemoje.

Rašant šį darbą buvo pastebėta, kad yra mažai informacijos apie įvykiais pagrįstos architektūros pagrindus, didžioji dalis informacijos kalba jos panaudojimą su minimaliais teoriniais paaiškinimais, kurių dauguma yra specifiniai temai, apie kuria kalbama. Todėl norint įsigilinti į architektūrą reikia ieškoti straipsnių, kurie yra sutelkti į teorinį temos aptarimą.

Darbo metu buvo pastebėta, kad dažniausiai sistemose įvykiais pagrįstą architektūrą būna pritaikyta tik vienoje ar poroje komponenčių ir tik labai retais atvejais – visame veikime. Tokie įgyvendinimai sukuria klaidingą architektūros supratimą ir apsunkina sistemų rašymą bei jų palaikymą.

Planuojant programos prototipą buvo pasirinktas įgyvendinimo būdas, kuris padėtų sumažinti dažniausių architektūros įgyvendinime pasitaikančių klaidų tikimybę. Tai buvo atlikta atsisakius asinchroninio veikimo, nes buvo pastebėta, kad funkcionalumas jo nereikalauja, ir pasirinktas įgyvendinimo formatas leidžiantis išvengti sistemos išskirstyti per kelis įrenginius arba programas.

Šiame darbe aprašyta sistema susideda iš dviejų pagrindinių dalių – vartotojo sąsajos ir serviso – bei duomenų bazės. Joje visas bendravimas tarp komponenčių vyksta tik per duomenų bazę. Programos prototipe buvo panaudoti datos ir laiko, baterijos įkrovos lygio bei buvimo vietos trigeriai, ekrano šviesumo, garsumo ir ekrano neveiklumo laiko funkcijas, kurios gali būti tiek veiksmai tiek trigeriai, bei trumpųjų žinučių siuntimo, pranešimo skelbimo ir programos paleidimo veiksmus. Plečiant programą būtų galima įgyvendinti trumpųjų žinučių ir pranešimų bei mobiliojo įrenginio išorinių mygtukų paspaudimo trigeriai ir žadintuvo įjungimo veiksmas.

Darbo metu buvo įvykdyti planuoti tikslai, įgauta daug žinių apie įvykiais pagrįstą architektūrą bei pagilintos JAVA programavimo kalbos ir Android operacinės sistemos.

Reziumė

In recent years, event-driven architecture has begun to gain popularity, but there are still a lot of misconceptions, so it is hard to say which systems actually use event-driven architecture and which just something similar. Because of these misunderstandings, even companies that have event-driven systems take little care in them. This paper will describe what is an event-driven architecture and how it is commonly represented in systems.

From the name of event-driven architecture alone, it can be understood that the main element of this architecture is an event, but this concept is ambiguous. In programming, an event can be portrayed as a change in system state or an object describing that change. Even if the event occurred a long time ago, its description is retained in the system's working memory if it is still relevant and, depending on the structure, it can be stored in a log. This type of data transfer allows to keep system elements decoupled and use a system in which it is important to create a process that can read the event description, rather than a central component that knows about all the processes.

Although event-driven architecture in practical implementations rarely exactly matches the theoretical model, they have similarities to one of two topologies: mediator and broker. Or, the system employs one of three event-driven patterns: event notification, event-carried state transfer, and event-sourcing.

Based on this architecture, it is convenient to implement automation systems. Automation allows human labor to be changed by the technological one. It can occur in one of two ways, but only one is relevant in this work. The automation system described here is designed to replace user-specified actions with software and is implemented in the Android operating system, while using “Android Studio” programming environment to build the application prototype, so the program descriptions will be based on the documentation of this environment.

The system described in this paper consists of two main parts (the user interface and the service) and the database. In it, all communication between the components takes place only through the database. The system was developed in a way that avoided many errors related to the event-driven architecture.

Šaltiniai

[And20a] Android Studio release notes. https://developer.android.com/studio/releases.

[And20b] Android Developers Documentation. https://developer.android.com/reference/android/app/TimePickerDialog.

[And20c] Android Developers Documentation. https://developer.android.com/reference/android/app/DatePickerDialog.

[And20d] Android Developers Documentation. https://developer.android.com/reference/android/widget/RadioButton.

[And20e] Android Developers Documentation. https://developer.android.com/reference/android/widget/SeekBar.

[And20f] Android Developers Documentation https://developer.android.com/reference/android/widget/RadioGroup.

[And20g] Android Developers Documentation. https://developer.android.com/reference/android/os/BatteryManager.

[And20h] Android Developers Documentation. https://developer.android.com/reference/android/telephony/gsm/SmsManager.

[And20i] Android Developers Documentation. https://developer.android.com/reference/android/support/v4/app/NotificationCompat.

[And20j] Android Developers Documentation. https://developer.android.com/reference/android/content/pm/PackageManager.

[And20k] Android Developers Documentation. https://developer.android.com/reference/android/media/AudioManager.

[And20l] Android Developers Documentation. https://developer.android.com/reference/android/provider/Settings.System.

[And20m] Android Developers Documentation. https://developer.android.com/reference/android/provider/AlarmClock.

[And20n] Android Developers Documentation. https://developer.android.com/reference/android/service/notification/NotificationListenerService.

[And20o] Android Developers Documentation. https://developer.android.com/reference/androidx/media/session/MediaButtonReceiver.

[Cur17] T.C. Currie. Event-Driven Architecture Is the Wave of the Future, https://thenewstack.io/event-driven-architecture-wave-future/?fbclid=IwAR15sm7WPdKFo-yATLt81Okt8TzOK-r_kLd4BaXphDFgSJHNH36DvXIyCh8. 2017.

[Fow17a] Martin Fowler. GOTO 2017 • The Many Meanings of Event-Driven Architecture, https://www.youtube.com/watch?v=STKCRSUsyP0. 2017.

[Fow17b] Martin Fowler. What do you mean by “Event-Driven”?, https://martinfowler.com/articles/201701-event-driven.html?fbclid=IwAR1BzDO6c56zRjMkXeqtTSKXlruaNZXl6XgGGqFBuVbNoSk3T64dA6D4qwo. 2017.

[Goo20a] Google APIs for Android, https://developers.google.com/android/reference/com/google/android/gms/maps/SupportMapFragment.

[Goo20b] Google APIs for Android, https://developers.google.com/places/web-service/intro.

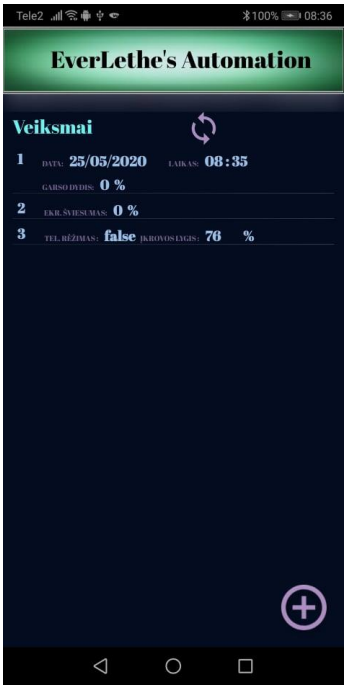
[Mic11] Brenda M. Michelson. Event-Driven Architecture Overview, http://elementallinks.com/el-reports/EventDrivenArchitectureOverview_ElementalLinks_Feb2011.pdf. 0,99 MB, 2011.

[Mil18] Jake Miller. Event-Driven Architecture: A Primer, https://medium.com/high-alpha/event-driven-architecture-a-primer-f636395d0295. 2018.

[Ric15] Mark Richards. Software Architecture Patterns. O‘Reilly Media, Sebastopol (CA), 2015.

Priedai

Priedas 1. Pradinis langas



Priedas 2. Redagavimo langas

Priedas 3. Trigerių/veiksmų pasirinkimo langas

Priedas 4. Informacijos įvedimo langas