ELEKTROTEHNIČKI FAKULTET
UNIVERZITET CRNE GORE

Prikupljanje podataka za analizu vektora zaraznih bolesti pomoću Android aplikacije

- specijalistički rad -

Mentor:
Prof. dr Slobodan Đukanović

Kandidat:
Božidar Škrbić

U Podgorici, septembra 2016. godine

ELEKTROTEHNIČKI FAKULTET
UNIVERZITET CRNE GORE

Tip studija: Akademske
Studijski program: Elektronika, telekomunikacije, računari
Smjer: Računari

Kandidat: Božidar Škrbić
Broj indeksa: 3/15

Prikupljanje podataka za analizu vektora zaraznih bolesti pomoću Android aplikacije

Datum izdavanja rada:

_____

Datum predaje rada:

_____

Mentor: ***Prof. dr Slobodan Đukanović***

_____

Kandidat: ***Božidar Škrbić***

_____

IZJAVA O SAMOSTALNOM RADU

Izjavljujem da sam ovaj specijalistički rad uradio samostalno,
uz pomoć mentora i navedene literature.

Kandidat:

_____

Božidar Škrbić

Specijalistički rad je odbranjen dana _____ 2016. godine, sa ocjenom ___ (_____), pred komisijom u sastavu:

1. _____ (predsjednik)
2. _____ (mentor)
3. _____ (član)

SADRŽAJ

1. UVOD **5**

2. LOVĆEN PROJEKAT **7**

3. ANDROID PROGRAMIRANJE **8**

3.1 Razvojna okruženja i programski jezici **8**

3.2 Razvoj Android aplikacije **10**

3.2.1 Aktivnosti 11
3.2.2 Upravljanje događajima 14
3.2.3 Fragmenti i navigacioni drawer 17
3.2.4 Obrada izuzetaka 20
3.2.5 Animacije 23
3.2.6 Višenitno programiranje 27
3.2.7 Komunikacija sa serverom 30

4. VECTOR INSPECT APLIKACIJA **36**

4.1 Aplikacija kao informacioni sistem **36**

4.2 Struktura baze podataka **37**

4.2 Struktura aplikacije **39**

4.2.1 Početni ekran 40
4.2.2 Slanje fotografija 41
4.2.3 Interaktivna mapa 49

5. ZAKLJUČAK **55**

6. LITERATURA **56**

1. UVOD

Android je operativni sistem kreiran od strane Google-a, baziran na Linux-ovom kernelu, koji je svoje postojanje započeo kao operativni sistem isključivo za mobilne telefone i tablet uređaje. Međutim, pristupačnost i prilagodljivost, jedna od najizraženijih osobina ovog operativnog sistema, kao i činjenica da je njegov kôd otvoreno dostupan svima (tzv. „open source“ koncept) dovela je do toga da Android danas nalazimo ne samo u većini mobilnih telefona i tablet računara dostupnih na tržištu već i u nekim uređajima skroz drugačijeg tipa (televizori, satovi, automobili i sl.). Svojom popularnošću Android je otvorio put ka stvaranju mnogobrojnih „pametnih“ uređaja različitih vrsta, a takođe je i sastavni dio velikog broja „internet of things“ rješenja.

Prema podacima iz 2015. godine, preko 54% postojećih mobilnih uređaja koristi Android operativni sistem. Taj procenat je prije samo godinu dana iznosio nešto preko 48%, što govori o tome koliko popularnost ovog operativnog sistema drastično raste iz godine u godinu. Kao što je već spomenuto, Android je „open source“ projekat, te je stoga pogodan brojnim kompanijama koje se bave proizvodnjom mobilnih telefona i tablet uređaja, jer vrlo lako mogu da unesu potrebne izmjene u operativni sistem, kako bi ga prilagodile svom proizvodu. Na taj način nastaju takozvani ROM-ovi. ROM je skraćenica od „read-only memory“, ali u kontekstu Android-a ovaj izraz označava varijantu Android operativnog sistema, zajedno sa svim sistemskim fajlovima i sistemskim aplikacijama koje korisnik ne može da obriše ili na bilo koji način mijenja. Ovo je, doduše, uslovno rečeno, jer se prilično jednostavno može odraditi postupak rutovanja (eng. root – korijen) čime korisnik sebi daje veću kontrolu nad uređajem, između ostalog i mogućnost brisanja i modifikovanja sistemskih aplikacija, veću kontrolu nad hardverskim komponentama telefona, kao i mogućnost instalacije potpuno nove verzije operativnog sistema, odnosno ROM-a. Razvoj ROM-ova nije vezan samo za kompanije koje se bave proizvodnjom Android uređaja, već se tom djelatnošću mogu baviti i nezavisni programerski timovi, što je često i slučaj. Ti tzv. „custom ROMs“ mogu biti ili neke izmjene i poboljšanja već postojećih fabričkih rješenja, bilo u pogledu optimizacija performansi ili izgleda korisničkog interfejsa, ili pak potpuno nova rješenja bazirana na Android-u, kao na primjer CyanogenMod.

Čitava priča o modifikacijama Android operativnog sistema, iako nije tema ovog rada, namjerno je istaknuta u uvodnom dijelu kako bi se na plastičnom primjeru pokazalo koliko je ovaj operativni sistem prilagodljiv i koliko kao takav podstiče čak i individualce i manje programerske timove da iz dana u dan izbacuju nova softverska rješenja i nova poboljšanja čak i u samom operativnom sistemu. Osim toga, ono što je takođe primamljivo i što mnoge programere motiviše da se upuste u razvoj Android aplikacija, sem činjenice da su Android uređaji najbrojniji, jeste i to što je plasiranje gotovog proizvoda vrlo jednostavno i isplativo. Google je još 2008. godine uveo koncept Android marketa (2012. godine preimenovan u Play Store), na kom programeri mogu da otvore nalog i da publikuju svoje kreacije koje na taj način postaju dostupne svim korisnicima Android uređaja. Namjerno je upotrijebljen izraz „isplativo“ budući da se otvaranje developerskog naloga na Play Store-u plaća 25$ jednokratno, što je ogromna prednost u odnosu na, recimo, konkurentski Apple-ov App Store, gdje je za otvaranje i održavanje najjeftinije varijante developerskog naloga potrebno izdvojiti 99$ godišnje. Takođe, razvojno okruženje Android Studio dostupno je za sve operativne sisteme, dok je Apple-ov Xcode dostupan isključivo za Mac OS. Dakle, u poređenju sa iOS-om, drugim po zastupljenosti operativnim sistemom za mobilne uređaje, razvoj aplikacija za Android zahtjeva daleko manje ulaganja.

Cjelokupna uvodna riječ je donekle imala za cilj da opravda razloge zbog kojih je VECTOR INSPECT aplikacija za prikupljanje podataka o vektorima zaraznih bolesti razvijana prvenstveno za Android platformu, budući da je brojnost uređaja i ekonomska isplativost tu igrala presudnu ulogu. Naravno nije isključeno da će u budućnosti postojati i neko rješenje za iOS, ili možda neka cross-platform varijanta, ali to je već priča za neku drugu priliku.

VECTOR INSPECT aplikacija je nastala kao proizvod Lovćen projekta sa ciljem da omogući efikasno sakupljanje podataka o rasprostranjenosti vrsta komaraca na teritoriji Crne Gore. Ideja je da korisnik putem aplikacije fotografiše primijećenog komarca, a da zatim aplikacija te fotografije pošalje na server vezujući uz njih i GPS koordinate i naziv opštine u kojoj su nastale. Na osnovu pristiglih podataka eksperti mogu da vrše identifikaciju primijećenih vrsta i da ih klasifikuju po opštinama, a dalje se ti sirovi podaci mogu iskoristiti za izradu odgovarajućih statistika i analiza sa ciljem predviđanja mogućeg izbijanja i širenja infektivnih bolesti koje ove vrste prenose. U sklopu aplikacije nalazi se i interaktivna mapa kojom se korisnicima prikazuje dio rezultata istraživanja, u prvom redu koje su sve vrste primijećene u odabranoj opštini i kolika je učestanost njihovog javljanja. Zamišljena je i implementacija ankete koja bi omogućila slanje povratnih informacija od strane korisnika i ispitivanje javnog mnjenja čime bi se istraživanje snabdjelo dodatnom količinom podataka za ekspertsku analizu, što bi istu učinilo detaljnijom i efikasnijom. Naravno, upotrebna vrijednost ovakve aplikacije prevazilazi samo jednu vrstu insekata (u ovom slučaju komaraca), tako da se ovo rješenje može u budućnosti upotrijebiti i u nekim drugim projektima kao vrlo efikasan alat za ovakav način prikupljanja podataka.

2. LOVĆEN PROJEKAT

Prije nego se osvrnemo na tehničke detalje koji se tiču Android programiranja i same izrade aplikacije valjalo bi posvetiti i nekoliko pasusa projektu u sklopu kog je VECTOR INSPECT aplikacija nastala. Lovćen projekat je projekat započet u maju 2014. godine sa ciljem vršenja nadzora nad komarcima i bolestima koje oni prenose. Efikasan nadzor nad komarcima i ostalim vektorima je od ključne važnosti za blagovremenu prevenciju i suzbijanje bolesti čiji su oni prenosioci, što je danas od posebnog značaja budući da su infektivne bolesti rastuća prijetnja kako Evropi, tako i čitavom svijetu. Osim nadzora nad prenosiocima patogena, projekat se fokusira i na poboljšanje saradnje i razmjene iskustava sa drugim istraživačkim centrima širom Evrope, trening i usavršavanje mladog naučnog kadra, stvaranje inovativnih metoda istraživanja i širenje relevantnih naučnih informacija iz ove oblasti, kao i nabavku najsavremenije istraživačke opreme i unaprjeđenje postojećih kapaciteta. Sve ovo će za posljedicu imati poboljšanje položaja crnogorskih istraživačkih institucija u odnosu na evropske, kao i unaprjeđenje vještina domaćeg istraživačkog kadra.

Istraživačke aktivnosti i metode Lovćen projekta su u potpunosti usklađene sa aktivnostima i istraživačkim metodama koje se koriste u evropskim istraživačkim centrima. Kako je naglašeno u rezimeu projekta, neke od predloženih tehnika, noviteti čak i na nivou Evropske unije, su sledeće:

1. korišćenje SIT metode za kontrolu brojnosti populacije invazivnih vrsta (eng. **S**terile **I**nsect **T**echnique – tehnika koja podrazumijeva puštanje u prirodu velikog broja sterilisanih muških primjeraka insekata, što za posledicu ima smanjenje brojnosti naredne generacije jedinki) – tehnika do sada primjenjivana samo u Belgiji;
2. implementacija „Priručnika za nadzor nad invazivnim vrstama komaraca u Evropi“;
3. korišćenje mobilnih telefona za nadzor nad invazivnim i domaćim vrstama komaraca – primjer realizacije ove metode je aplikacija VECTOR INSPECT čiji će način funkcionisana i programerska realizacija biti do detalja opisani u narednim poglavljima;
4. razvoj i upotreba novih, netoksičnih, biorazgradivih materijala za kontrolu larvi komaraca;

Sem ovoga, u sklopu projekta se u Crnoj Gori po prvi put obavljaju sljedeća istraživanja:

1. identifikacija prisutnih vrsta komaraca i njihova distribucija (prethodno istraživanje ove vrste je rađeno prije trideset godina i bilo je ograničeno na oblast Durmitora);
2. identifikacija komaraca vektora;
3. utvrđivanje patogena koji prenose komarci;
4. modelovanje uticaja klimatskih promjena na patogene i njihove nosioce;
5. ispitivanje javnog mnjenja na temu prevencije prenosivih bolesti;

Dâ se primijetiti da VECTOR INSPECT aplikacija može naći svoju primjenu u većini nabrojanih stavki i da će kao izuzetno koristan alat intenzivno biti primjenjivana za efikasno prikupljanje podataka bitnih za istraživanje.

Postoji jasna vizija održivosti projekta i nakon njegovog završetka, budući da je oblast istraživanja kojom se on bavi od interesa za državu, u prvom redu za zdravstvo, turizam i održivi razvoj. U planu je da učesnici projekta nastave sa radom na novim projektima. Nastaviće se i sa upotrebom nabavljene opreme u sklopu trening aktivnosti, nastavnog procesa kao i novih naučnih istraživanja. Vrijedi posebno spomenuti u kontekstu ovog rada da će program nadzora korišćenjem VECTOR INSPECT aplikacije biti nastavljen i po okončanju Lovćen projekta.

3. ANDROID PROGRAMIRANJE

3.1 Razvojna okruženja i programski jezici

Dominantno korišćeno okruženje, odnosno IDE (**I**ntegrated **D**evelopment **E**nvironment), za razvoj Android aplikacija je **Android Studio**. Može se donekle reći da je ovo okruženje novitet u svijetu programiranja, budući da postoji tek nekih par godina. Naime, prva demo verzija, pod oznakom 0.1, objavljena je maja 2013. godine. Prva beta verzija, počevši od verzije 0.8, objavljena je u junu 2014. godine, dok je prvo stabilno izdanje, verzija 1.0, dostupna od decembra 2014. godine, nakon čega je Android Studio postao Google-ovo zvanično okruženje za nativni razvoj Android aplikacija, što je do tada bio Eclipse ADT.

Za razliku od Eclipse razvojnog okruženja, koje je prije svega okruženje za rad u Javi, i koje je zahtijevalo instalaciju dodatnog ADT paketa (**A**ndroid **D**evelopment **T**oolbox) kako bi se moglo koristiti i za Android programiranje, Android Studio je razvojno okruženje pravljeno isključivo za razvoj Android aplikacija. U tom smislu je mnogo prilagođeniji za ovu djelatnost, i u velikoj mjeri je stabilniji i brži u odnosu na Eclipse, naročito kada je u pitanju rad na složenijim aplikacijama, što je dovelo do toga da nedugo nakon objavljivanja prve verzije Android Studija, Eclipse ADT bude u potpunosti potisnut. Android Studio uvodi brojne novitete u razvoju Android aplikacija, između ostalih, upotrebu Gradle-a u procesu software build-a (kompajliranje mašinskog koda i formiranje instalacionog paketa) umjesto Apache Ant-a koji je korišćen u Eclipse IDE-u, i u odnosu na koji je Gradle build tehnički napredniji, pregledniji i efikasniji alat (sam Gradle je nastao usavršavanjem Apache Ant-a), integracija Lint i ProGuard alata za provjeru, optimizaciju i maskiranje kôda itd.

Proces razvoja aplikacija u ova dva okruženja može se grubo podijeliti na dva dijela: dizajn i razvoj korisničkog interfejsa i programiranje funkcionalnosti. Za programiranje većeg dijela funkcionalnosti koristi se Java programski jezik. Međutim, valja napomenuti da je Java koja se koristi u Android programiranju donekle modifikovana u odnosu na Javu koja se koristi za razvoj programa na PC računarima. Naravno, sintaksa i osnovne karakteristike su u oba slučaja iste, ali postoje i određene razlike. Prvo, biblioteke i metode koje te biblioteke sadrže se razlikuju. Ovo se, donekle, može i logički zaključiti, jer programer koji radi na razvoju aplikacija za mobilne uređaje (ili bilo koji drugi tip uređaja koji nije standardan PC računar) ima potrebu za dodatnim kontrolama. Tu se prvenstveno misli na biblioteke za upravljanje resursima uređaja, obradom događaja (koja se donekle razlikuje u odnosu na obradu događaja u standardnoj Javi), grafičkim interfejsom i sl**.** Takođe, programi pisani za Android uređaje se kompajliraju na način drugačiji od standardnog. Proces kompajliranja Java source kôda i pokretanja Java programa je, ukratko, sledeći: kompajler na osnovu source kôda kreira bajt kôd čiji format ne zavisi od platforme na kojoj se program pokreće. Bajt kôd se može, ali ne mora, upakovati u *javac.exe* izvršni fajl. Prilikom pokretanja Java programa, prvo se pokreće odgovarajuća virtuelna mašina (koja se razlikuje od platforme do platforme) i koja zatim interpretira bajt kôd u mašinski kôd. Ukoliko je bajt kôd prethodno bio zapakovan u *javac.exe* fajl, pokretanje virtualne mašine i učitavanje bajt kôda će se obaviti automatski. Kada su u pitanju Android aplikacije, postupak počinje na sličan način: source kôd se kompajlira u standardni bajt kôd. Međutim, tu se sva sličnost završava. Dalje se standardni bajt kôd prevodi u Dalvik bajt kôd (fajl sa *.dex* ekstenzijom), koji se zatim pakuje u dobro poznati *.apk* fajl, format u kom se vrši distribuiranje aplikacije. Pokretanje aplikacije započinje izvlačenjem *.dex* iz *.apk* fajla. Dalvik bajt kôd se zatim interpretira u mašinski i aplikacija se pokreće. Dakle, Android operativni sistem posjeduje svoju varijantu virtualne mašine koja se u potpunosti razlikuje od standardne, tako da na Android uređaju nije moguće pokretati Java programe pisane za, na primjer, Windows računar bez prethodne instalacije nekog emulatora. Ovo valja naglasiti jer je osnovni koncept Jave mogućnost pokretanja Java programa nezavisno od platforme, što bi moglo navesti na pomisao da postoji mogućnost da se računarski programi pisani u Javi direktno mogu pokrenuti na Android uređajima. Naravno, predstavljeni opis procesa kompajliranja i

pokretanja Java programa i aplikacija je vrlo grub i dat u kratkim crtama, sa ciljem isticanja suštinskih razlika između Jave koju koristi Android i standardne Oracle-ove Jave. Dakle, na Javu koja se koristi u razvoju Android aplikacija može se gledati kao na poseban Google-ov programski jezik koji je formiran kao nadogradnja na standardnu. Naravno, osoba vična klasičnom Java programiranju bez problema može da savlada i Android programiranje.

Ono što još razlikuje ove dvije varijante Java programskog jezika jeste i način formiranja grafičkog korisničkog interfejsa. Za njegovo definisanje i upravljanje njegovim komponentama računarski Java programi koriste metode iz *javax.swing* i *java.awt* paketa. U Android programiranju, korisnički interfejs se definiše pomoću XML-a, a upravljanje se vrši u Java kôdu pomoću metoda i varijabli iz *android.widget* paketa, gdje se na poseban način XML komponente vezuju sa promjenljivima. O ovome će više biti riječi u narednoj sekciji.

Ono što je, međutim, isto kod obje varijante Java programskog jezika jesu neke osnovne prednosti i nedostaci u odnosu na druge programske jezike. Kao što koncept bajt kôd-a i Java virtualne mašine omogućava izvršavanje standardnih Java programa nezavisno od računarske platforme, tako je i cjelokupan sistem kompajliranja i pokretanja Android aplikacija osmišljen na način kako bi se povećala kompatibilnost aplikacija sa različitim tipovima uređaja na kojima je podignut Android operativni sistem, a koji se po hardverskoj strukturi u nekoj mjeri razlikuju, prvenstveno u pogledu procesorske arhitekture. Ovo je posebno bitno u današnje vrijeme kada Android OS srećemo na nekim, može se reći, neočekivanim mjestima (pametni satovi, pametni televizori, narukvice, naočare, automobili i sl.). Takođe, budući da se aplikacije pokreću na virtuelnoj mašini, njihov rad je odvojen od rada sistemskih procesa, tako da malware-i koje bi aplikacija mogla da sadrži ili pad aplikacije ne mogu da utiču na operativni sistem na kom je virtuelna mašina pokrenuta. Osim toga, Java je sama po sebi moćan programski jezik koji sadrži pregršt zgodnih metoda i biblioteka za rješavanje nekih čestih programerskih problema, od kojih je velika većina dostupna i u Android programskim okruženjima. Tu je i čuveni „garbage collector“, kao i ostali koncepti kojima Java sprječava neke nenamjerne programerske greške koje vrlo lako mogu dovesti do stvaranja raznih nepravilnosti u radu softvera ili drugih neželjenih pojava kao, na primjer, tzv. „curenje memorije“.

Glavni nedostatak Jave u odnosu na neke druge programske jezike jeste brzina izvršavanja programa. Čitavo „petljanje“ sa bajt kôdovima, *.dex* fajlovima i virtuelnim mašinama, pogodnosti koje donosi naplaćuje u vidu vremena izvršavanja. Ovo može biti prilično nezgodan problem kada su u pitanju neke zahtjevne operacije koje mogu značajno usporiti aplikaciju i smanjiti njene performanse, kao, na primjer, obrada slike u realnom vremenu, fizičke simulacije, i slično. U tu svrhu Android Studio ima tzv. Android NDK (**N**ative **D**evelopment **K**it), koji omogućava programeru da dio svoje aplikacije, onaj koji je vremenski posebno zahtjevan, napiše u programskim jezicima C i C++. Budući da se C i C++ ne oslanjaju na virtuelnu mašinu, programi pisani u tim jezicima imaju značajno kraće vrijeme izvršavanja, naročito programi pisani u C-u budući da je on proceduralni, a ne objektno-orjentisani jezik. Međutim, nije preporučljivo da se NDK koristi sem onda kada je to zaista potrebno, budući da ovaj alat unosi dodatnu kompleksnost u izradi aplikacije.

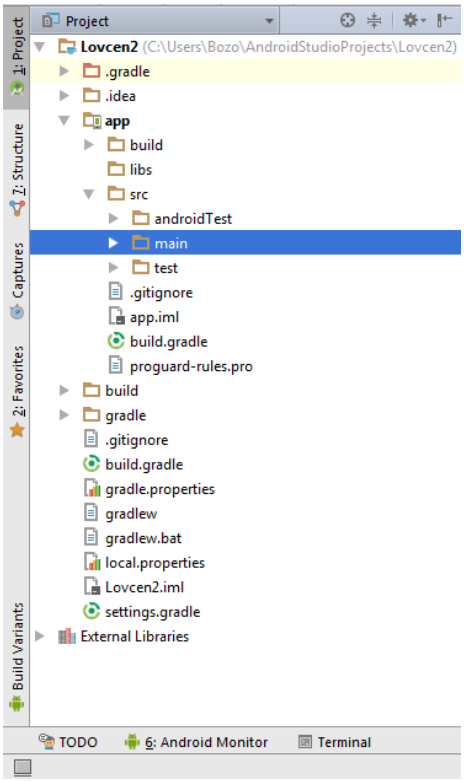
Sve o čemu je do sada govoreno ticalo se tzv. **nativnog programiranja**. Razvoj aplikacija za isključivo jednu platformu odnosno jedan operativni sistem je nativno programiranje. Kôd pisan u okruženju namijenjenom nativnom razvoju aplikacija može se isključivo kompajlirati u oblik u kom se može pokretati samo na jednoj platformi. Razvoj iste aplikacije za neku drugu platformu bi podrazumijevao rad od nule. Nasuprot ovome stoji **cross-platform programiranje**. Kao što sâm naziv sugeriše, source kôd pisan u ovakvim okruženjima se može kompajlirati na više različitih načina, tako da se aplikacija može pokretati na različitim platformama. Dok je jedini korišćeni jezik u nativnom Android programiranju Java, u cross-platform razvoju izbor je znatno širi. Postoje mnogobrojna razvojna okruženja koja omogućavaju razvoj u različitim programskim jezicima: Java, C++, C#, HTML, CSS, JavaScript itd. Pojedina razvojna okruženja koja se koriste za cross-platform razvoj su ona koja smo navikli da koristimo u svrhu razvoja računarskih programa. RAD Studio, koji omogućava razvoj u C++ -u, Xamarin, koji

omogućava razvoj aplikacija u Visual Studiju, upotrebom C# programskog jezika, Codename One dodatak za Eclipse, NetBeans i IntelliJ IDEA, koji proširuje primjenu Jave sa nativnog na cross-platform razvoj, i mnogi drugi. Tu su i razvojna okruženja u kojima je moguće izraditi cross-platform aplikacije koristeći kombinaciju HTML-a, CSS-a i JavaScript-a (slično kao pri izradi web sajtova) kao npr. PhoneGap. Postoji i treća varijanta, tzv. **hibridno programiranje**, kada razvoj aplikacije kombinuje nativne i cross-platform elemente.

Pitanje odabira metode razvoja zavisi prvenstveno od potreba onoga ko zahtjeva razvoj aplikacije. Ukoliko se insistira na glatkom korisničkom interfejsu, animacijama i visokim performansama, onda je bolje raditi nativnu aplikaciju. Ukoliko je ipak bitnije to da aplikacija dopre do što većeg broja korisnika, a da pri tom troškovi razvoja budu minimalni, onda je cross-platform razvoj bolje rješenje. Kao što je već navedeno u prethodnom poglavlju, VECTOR INSPECT aplikacija je rađena isključivo za Android OS, međutim, u zavisnosti od potreba, moguće je proširiti razvoj i na druge platforme.

3.2 Razvoj Android aplikacije

U prethodnoj sekciji smo se kratko dotakli različitih tipova Android aplikacija, različitih razvojnih okruženja u kojima ih programeri mogu stvarati, kao i različitih programskih jezika koji stoje na raspolaganju programerima u zavisnosti, prvenstveno, od odabrane varijante razvoja aplikacija. Na tu temu se, naravno, može još mnogo toga kazati, međutim poenta priče nije bio detaljan pregled različitih razvojnih okruženja i njihove organizacije, već prosto kratak rezime svih mogućnosti koje programeri koji se upuštaju u razvoj Android aplikacija imaju pred sobom. Ipak, nešto više pažnje posvetićemo okruženju u kom je VECTOR INSPECT aplikacija razvijana, i koje je trenutno jedino Google-ovo zvanično okruženje za razvoj aplikacija – Android Studio.



Slika 1. Organizacija foldera u Android Studiju

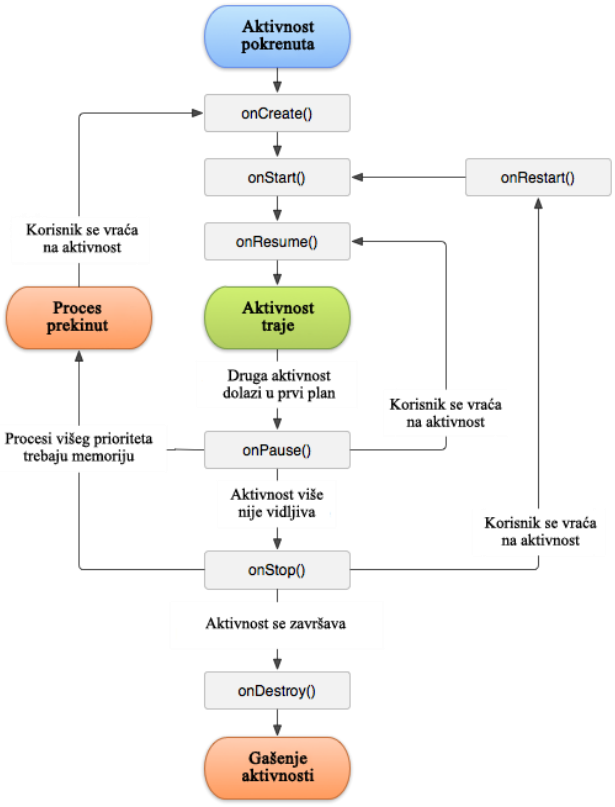
Već su predočene neke od osobina ovog okruženja: bazira se na Google-ovoj verziji Jave, koristi XML za definisanje izgleda korisničkog interfejsa, ima mogućnost programiranja djelova aplikacije u C-u ili C++ -u preko NDK alata itd. Osvrnućemo se potom na neke elemente Android programiranja, stavljajući akcenat na one prisutne u VECTOR INSPECT aplikaciji.

Android Studio koristi prilično složenu strukturu foldera (slika 1) kako bi organizovao sve fajlove iz kojih se kasnije, nakon software build-a, dobija gotov proizvod, odnosno *.apk* fajl. Kontekst cjelokupnog projekta sadržan je u folderu *Lovcen2* koji u Android Studiju ima rang **projekta**. Ovo je sličan koncept kao workspace u Eclipse-u i pripada istom organizacionom nivou, s tom razlikom što se sadržaj ovog foldera ograničava samo na fajlove koji su relevantni za razvoj jedne, konkretne, aplikacije. Sledeći po rangu dolaze **moduli**. Za razliku od projekta, jedna aplikacija može sadržati više modula, koji se po svom tipu mogu razlikovati. Glavni modul je modul ovdje imenovan kao *app*. To je aplikacijski modul (eng. app module) i on čini srž aplikacije. Ovdje se, između ostalog, nalaze sve klase, resursi, manifest fajl (*app/src/main)*, importovane biblioteke (*app/libs*) i *build.gradle* fajl u kom su definisani podaci od značaja za software build, a koji se tiču isključivo ovog modula. Sem aplikacijskog modula postoje i biblioteke (eng. library module) i Google Cloud modul. Library module omogućava da se dio kôda aplikacije koji je sadržan u tom modulu upotrijebi i u nekom drugom projektu importovanjem u vidu

biblioteke, dok Google Cloud modul sadrži rutine i mehanizme kojima se aplikacija može povezati na Google Cloud.

3.2.1 Aktivnosti

Iako je korisno znati način na koji su svi fajlovi organizovani, prilikom razvoja aplikacije programer ne mora previše voditi računa o tome. Ono što je sa programerskog stanovišta najvažnije je struktura same aplikacije. Tu se, prije svega, dotičemo pojma **aktivnosti** (eng. **activity**), kao najbitnijeg elementa. Aktivnost bi se mogla definisati kao strukturna cjelina aplikacije koja se sastoji iz funkcionalnosti definisane u Javi i korisničkog interfejsa definisanog u XML-u koji su međusobno vezani, i koji kao takvi čine osnovni gradivni element svake aplikacije. Java source kôd je smješten u *main/java* folderu, dok se XML kojim se definiše izgled aktivnosti nalazi u *main/res/layout* folderu. Uopšteno, layout-ovi su u Androidu posebni XML resursi kojima se definišu izgledi, bilo aktivnosti, bilo iskačućih prozora, elemenata liste ili spinera i sl. Dio kôda koji se prvi pokreće prilikom startovanja aplikacije je source kôd one aktivnosti koja je označena kao glavna odnosno „launcher“ activity. I tu je, takođe, razlika između Android aplikacija i klasičnih Java programa: dok se pokretanje kôda u Java programima započinje sa metodom *main* koja se nalazi unutar neke klase, u Android-u sve kreće sa odabranom aktivnosti.



Slika 2. Životni ciklus aktivnosti

Prilikom startovanja aktivnosti ona prolazi kroz dobro definisan „životni ciklus“ (slika 2). Prvo se poziva niz metoda, redom *onCreate()*, *onStart()* i *onResume()*, koje se sve mogu preklopiti kako bi programerima omogućile definisanje različitih kôdova koji bi se izvršavali u ovim različitim fazama kreiranja i postojanja aktivnosti. Kada je aktivnost pokrenuta njen interfejs postaje vidljiv na ekranu uređaja, i mehanizam za obradu događaja postaje aktivan (dakle, od ovoga trenutka korisnik može da upravlja aplikacijom preko korisničkog interfejsa). U toku trajanja aktivnosti, mogu se desiti situacije da nova aktivnost bude pozvana i parcijalno ili u potpunosti zakloni prethodnu. Tada se aktivnost ne prekida već se u prvom slučaju pauzira, odnosno stopira u drugom. Aktivnost u sebi sadrži metode *onPause()* i *onStop()* koje se pozivaju nakon pauziranja odnosno stopiranja, a čijim preklapanjem možemo da definišemo kôd koji će se izvršavati u ovim trenucima. Nakon što korisnik ili aplikacija odluči da se vrate na „sklonjenu“ aktivnost, ciklus se vraća nazad na pozivanje *onResume()* metode kada je u pitanju pauziranje, odnosno *onRestart()* pa *onStart()* metode kada je u pitanju stopiranje aktivnosti. U slučajevima kada se zbog memorijskih zahtjeva procesa višeg prioriteta proces koji pripada aktivnosti „ubije“, odnosno prekine, tada se prilikom vraćanja na tu aktivnost ciklus mora vratiti na *onCreate()* metodu. Konačno, u fazi kada se aktivnost završava poziva se *onDestroy()* metoda, koja se takođe može preklopiti kao i sve ostale.

Funkcionalnost aktivnosti se tretira kao standardna Javina klasa koja nastaje nasleđivanjem klase *AppCompatActivity*, koja je dio biblioteke *android.support.v7.app.AppCompatActivity*. Ova biblioteka pored već pomenutih metoda koje prate faze ciklusa aktivnosti, sadrži i druge metode čijim se

preklapanjem mogu programirati različite funkcionalnosti: upravljanje toolbar-om, menijem, drawer-om i sl. Izgled aktivnosti se definiše u odgovarajućem *.xml* fajlu. Svaki *.xml* fajl kao osnovu ima layout element u koji se dalje mogu smještati ili komponente ili novi layout-ovi, a način na koji će se raspoređivati zavisi od tipa osnovnog layout-a. XML layout-ovi obavljaju isti posao kao i layout menadžeri u standardnoj Javi (iako su suštinski totalno različiti budući da se layout menadžeri definišu kao klase). Android Studio posjeduje ugrađeni editor grafičkog interfejsa koji nam omogućava da rasporedimo komponente onako

```
package com.example.bozo.lovcen;

import android.support.v7.app.AppCompatActivity;

import android.view.View;
import android.widget.AdapterView;
import android.widget.ArrayAdapter;
import android.widget.ListView;
import android.widget.TextView;

import java.util.ArrayList;


public class SpeciesList extends AppCompatActivity {

    ListView list;
    ArrayAdapter<String> listAdapter;
    ArrayList<DatabaseRead> listData;

    String municipality;
    ArrayList<DatabaseRead> readData;

    @Override
    protected void onCreate(Bundle savedInstanceState) {


        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_species_list);

        readData = FragmentMapa.readData;
        municipality = getIntent().getExtras().getString("municipality");
        listData = new ArrayList<>();

        for( DatabaseRead r: readData )
            if( r.getMunicipality().equals(municipality) )
                listData.add(r);

        getSupportActionBar().setTitle(R.string.species_activity_title);

        list = (ListView) findViewById(R.id.list);
        listAdapter = new SpeciesListAdapter(this, listData);

        list.setAdapter(listAdapter);

        list.setOnItemClickListener(new AdapterView.OnItemClickListener() {

            @Override
            public void onItemClick(AdapterView<?> parent, View view,
                                                    int position, long id) {

                //Ovdje se definiše reakcija na događaj

            }

        });

    }


}
```

kako želimo, bez direktnog uplitanja u *.xml* fajl. Svu modifikaciju *.xml* fajla editor obavlja samostalno. Naravno, ukoliko želimo da se svaka komponente grafičkog interfejsa savršeno rasporedi na ekranu, nekad je nužno ručno napraviti izmjene budući da editor u rijetkim trenucima ne uspijeva da uradi tačno ono što od njega zahtijevamo. Bilo kako bilo, ova funkcionalnost Android Studija (kao i Eclipse ADT-a) unosi velike olakšice u rad prilikom dizajniranja i programiranja grafičkog interfejsa. Android Studio takođe omogućava i definisanje različitog izgleda aktivnosti u zavisnosti od tipa uređaja, odnosno dimenzija njegovog ekrana ili od njegove orjentacije (portrait ili landscape). Na taj način programeri vrlo lako mogu da savršeno prilagode svoju aplikaciju za različite tipove uređaja, i da pritom svim korisnicima, bez obzira na kakvom uređaju pokreću aplikaciju, omoguće savršen „user experience". Na konkretnom primjeru aktivnosti *SpeciesList*, koja će kasnije biti pomenuta u skopu opisa VECTOR INSPECT aplikacije, to izgleda ovako: prva linija kôda definiše paket kom klasa pripada, kako bi se vezala za dogovarajući projekat. Naredne linije koje počinju ključnom riječju ***import*** se odnose na biblioteke koje klasa importuje, odnosno uvozi. Na sreću, programer o tome ne mora da vodi računa: paket i biblioteke se definišu automatski. Zatim slijedi deklaracija i realizacija klase. Ključna riječ ***extends*** označava da klasa koju upravo kreiramo (*SpeciesList*) nasleđuje klasu *AppCompatActivity*, čime se sugeriše da se radi o aktivnosti a ne o običnoj klasi. Naredne linije odnose se na deklaraciju promjenljivih. Ovdje sintaksa prati standardnu Javinu sintaksu tako da nema potrebe ništa dublje elaborirati. Nakon toga, potrebno je preklopiti metod *onCreate()*. Oznakom *@Override* kompajleru se sugeriše da se radi o preklopljenoj funkciji, čime se eliminiše potreba da on to sam provjerava, te se zahvaljujući toj oznaci vrijeme kompajliranja smanjuje. Njeno dodavanje nije obavezno i ne utiče na ispravno funkcionisanje programa. Metoda *onCreate()* sadrži jedan vrlo koristan argument tipa *Bundle*. Bundle, kako sam naziv sugeriše, je „svežanj" različitih promjenljivih i o tome će biti riječi nešto kasnije. Ovdje je dovoljno reći da ova promjenljiva sadrži podatke o statusu aktivnosti koji se prosleđuju metodi *onCreate()* prilikom njenog pozivanja kako se određeni podaci vezani za aktivnost ne bi izgubili prilikom njenog ponovnog kreiranja (npr. ukoliko uređaj promijeni orjentaciju, ili ukoliko se aktivnost pozove nakon što je sistem usled memorijskih potreba prekinuo tu aktivnost). Dalje se metoda *onCreate()* preklapa kako bi definisali šta to treba da se desi kada se aktivnost kreira.

Prije svega, linija *super.onCreate(...)* poziva „naslijeđeni dio kôda" iz nadklase *AppCompatActivity*, tj. linije kôda koje se nalaze u neprekloplјenoj metodi *onCreate()*. Ovaj poziv je vrlo bitan zbog dodjele konteksta aktivnosti, tako da, iako generalno nije neophodno ovakvo pozivanje kod ostalih metoda, koje možemo u potpunosti preklopiti, u ovom slučaju je ovaj poziv obavezan. Sledeći bitan korak je vezivanje Java klase koja definiše funkcionalnost aktivnosti sa njenim izgledom definisanim odgovarajućim *.xml* fajlom. To se radi metodom *setContentView(...)*, gdje se kao argument prosleđuje ID layout-a koji je definisan unutar *.xml* fajla. Sada se dolazi do pitanja čemu služi prefiks *R.layout?* ***R*** je klasa koja se sastoji od niza ugniježdenih klasa od kojih svaka sadrži podatke o svim ID-jevima nekog od tipova resursa: layout-ovi, komponente layout-ova, animacije, drawable resursi, i ostali. Svaki ID koji definišemo u XML-u definišemo kao string, pri čemu se u nekoj od unutrašnjih klasa *R* klase, zavisno od tipa resursa, automatski dodaje nova promjenljiva čiji će naziv biti taj string koji smo definisali. Ta promjenljiva je uvijek tipa *int* i prilikom inicijalizacije promjenljive automatski će se izvršiti i deklaracija pridruživanjem neke cjelobrojne vrijednosti. Dakle, ID-jevi kojima metode barataju nisu stringovi koje smo mi definisali, već cjelobrojne vrijednosti koje su sa njima povezane. Sada je jasno da je argument *R.layout.activity_species_list* cijeli broj koji jednoznačno definiše resurs tipa layout koji je nazvan „activity_species_list". Na ovaj način se funkcionalnost i grafički interfejs vezuju u jednu cjelinu.

Sada želimo pristupiti i upravljati određenim komponentama koje su definisane u sklopu grafičkog interfejsa. Da bi to bilo moguće potrebno je komponentu deklarisati kao promjenljivu članicu klase odgovarajućeg tipa (tipovi koji se odnose na komponente grafičkog interfejsa definisani su unutar biblioteka *android.widget.**). Tako deklarisana promjenljiva se sada definiše vezivanjem sa svojim parom na grafičkom interfejsu metodom *findViewById(...)* koja kao argument prima cjelobrojni ID komponente koji se definiše u *R* klasi na isti način kao i ID layout-a. Sada je moguće manipulisati komponentom GUI-ja direktno iz kôd-a. U ovom primjeru, komponenta koju smo definisali je tipa ListView, a u sklopu

upravljanja tom komponentom prvo je za nju vezan adapter liste metodom *setAdapter(...)* koja kao argument prima instancu klase koja definiše taj adapter čija je svrha popunjavanje liste odgovarajućim podacima. Zatim je za listu vezan osluškivač metodom *setOnItemClickListener(...)* koji definiše operacije koje će se izvršavati klikom na neki od elemenata liste. Ova metoda i njoj slične dio su mehanizma obrade događaja koji je možda i najbitniji segment Android programiranja, i koji je sastavni dio funkcionalnosti gotovo svake aplikacije.

3.2.2 Upravljanje događajima

Teško je zamisliti neku aplikaciju koja bi se mogla realizovati tako da funkcioniše bez bilo kakve interakcije sa korisnikom, a da pri tom obavlja neku korisnu djelatnost. Gotovo sve aplikacije se projektuju tako da primaju određene ulazne podatke od strane korisnika (preko interaktivnih komponenti koje predstavljaju izvor događaja), i da na osnovu njih vraćaju odgovarajuću informaciju, stvore određenu reakciju na korisničkom interfejsu, izvrše neku operaciju i sl. U Android programiranju kao i u standardnom Java programiranju postoji izuzetno efikasan mehanizam upravljanja događaja kojim se funkcionalnost interaktivnih komponenti može isprogramirati lako i jednostavno. Realizacija upravljanja događaja se, generalno, odvija u tri faze:

1. modelovanje klase koja obavlja funkciju obrađivača događaja (eng. event handler) i koja, kao takva, implementira neki osluškivački interfejs;
2. instanciranje objekta te klase, koji sada obavlja funkciju osluškivača, odnosno, obrađivača događaja;
3. registrovanje osluškivača za GUI komponentu;

Dakle, ideja je da se nakon što korisnik izvrši interakciju sa nekom komponentom grafičkog interfejsa (koja se tretira kao izvor događaja) generiše objekat, tzv. objekat događaja, koji u sebi sadrži referencu na izvor događaja, podatak o tipu događaja i sl. Dalje se taj objekat prosljeđuje osluškivaču, koji na osnovu izvora ili tipa događaja pokreće odgovarajuće rutine kao reakciju na korisničku interakciju.

Iako ništa ne sprječava programera da upravljanje događajima realizuje strogo prateći ovaj postupak, u Android programiranju je ipak najčešća praksa da se ovaj proces skrati na svega jedan korak koji će obuhvatiti sve tri gore navedene faze, upotrebom anonimnih ugniježdenih (unutrašnjih) klasa. Na primjer:

```
public class CameraActivity extends AppCompatActivity {

      Button sendButton;

         ...

       @Override
       protected void onCreate(Bundle savedInstanceState) {

                    ...

              sendButton.setOnTouchListener(new View.OnTouchListener() {

              @Override
                  public boolean onTouch(View v, MotionEvent event) {

                       if (event.getAction() == MotionEvent.ACTION_DOWN)
                          sendButton.setBackgroundResource(R.drawable.button_shape_clicked);

                       ...
```

```
                        ...

                        else if (event.getAction() == MotionEvent.ACTION_UP){
                           sendButton.setBackgroundResource(R.drawable.button_shape);

                           //Generisanje povratnih informacija, reakcije
                                          interfejsa ili pokretanje odgovarajućih operacija

                        }

                        return true;

                    }

                });

                ...

        }

}
```

Ovdje je na komponentu *sendButton* (koja je, kao promjenljiva tipa *Button*, članica klase *CameraActivity* kojom se realizuje funkcionalni dio istoimene aktivnosti) registrovan osluškivač kom je pokretačka interakcija dodir na komponentu. Dakle, ***new*** *View.OnTouchListener(){...}* je instanciranje objekta anonimne klase (koja je, istovremeno, i unutrašnja budući da se nalazi unutar klase *CameraActivity*) koji implementira interfejs *OnTouchListener* (faza 1 i 2). Unutar te klase se realizuje obrada događaja, a taj objekat se kao argument prosljeđuje metodi *setOnTouchListener(...)* koja za komponentu *sendButton* registruje osluškivač, i to baš onaj koji implementira *OnTouchListener* interfejs (faza 3).

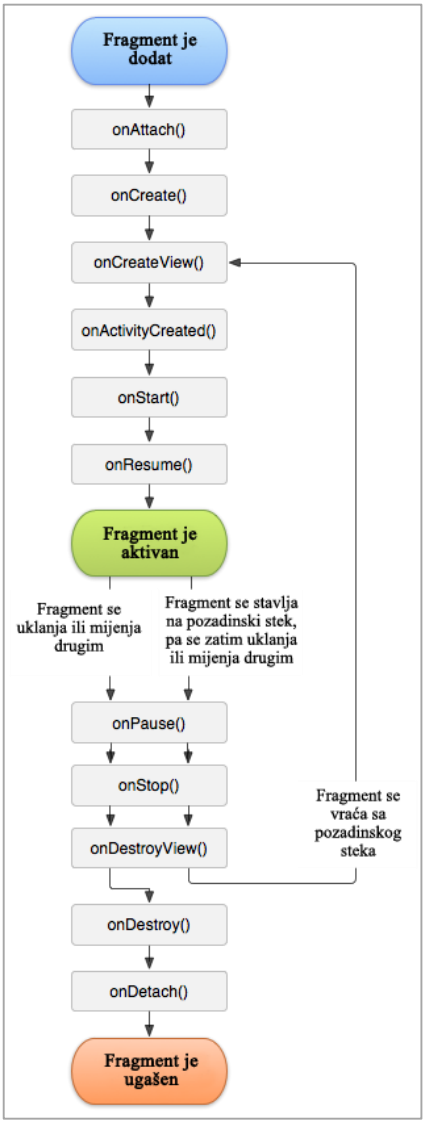
Konkretna realizacija kôda kojim se vrši obrada događaja se radi unutar tzv. **callback metode**. Ovo su metode koje su sadržane unutar osluškivačkih interfejsa i jedine su metode koje ti interfejsi sadrže. Zbog ove osobine u Android programiranju ne postoji potreba za adapterskim klasama. Callback metoda *OnTouchListener* interfejsa je *onTouch(...)*. Ova metoda prima dva argumenta koji se prosljeđuju automatski prilikom generisanja događaja: referencu na komponentu interfejsa koja je izvor događaja, i objekat koji sadrži potpune informacije o događaju. U ovom konkretnom primjeru iskorišćen je podatak o tipu događaja koji se desio nad komponentom *sendButton*. Kada se komponenta dodirne, mijenja se njen izgled, a kad se, zatim, komponenta „pusti“ njen izgled se vraća na prvobitni i započinje se izvršavanje željenih operacija. Na ovaj način je u potpunosti realizovana interakcija sa korisnikom, a mijenjanje izgleda komponente prilikom dodira doprinosi vizuelnom izgledu aplikacije i ostavlja povoljniji utisak na korisnika. Sem *onTouchListener* interfejsa, u Android programiranju postoje i još neki. Neki od najčešće korišćenih prikazani su na slici 3.

Svi navedeni osluškivački interfejsi su sastavni dio biblioteke *android.view.View*, iste one biblioteke koja sadrži i klase kojima se modeluju grafičke komponente u Java kôdu. Sem ovih osluškivača, koji reaguju na korisničku interakciju, postoje i oni koji reaguju na promjene koje nisu izazvane korisničkim djelovanjem. To su osluškivači koji se usko vezuju za neku od funkcionalnosti, i sastavni su dio biblioteka koje sadrže klase i metode kojima se tim funkcionalnostima upravlja. Primjera radi, takvi su osluškivači koji se aktiviraju prilikom završetka animacije, promjene lokacije, završetka pozadinske niti i drugi. Na njih ćemo se kratko osvrnuti u narednim sekcijama i poglavljima.

INTERFEJS	CALLBACK METODA
OnClickListener Osluškivač se aktivira prilikom svakog klika, tj. dodira. Za razliku od *OnTouchListener*-a, koji može da napravi razliku između različitih vrsta dodira i pokreta, osluškivači koji implementiraju ovaj interfejs to ne mogu.	***void onClick(*** *View v* ***)***: *v* – komponenta izvor događaja
OnLongClickListener Osluškivač se aktivira isključivo na dodire koji traju duže od jedne sekunde.	***void onLongClick(*** *View v* ***)***: *v* – komponenta izvor događaja
OnKeyListener Aktiviranje osluškivača se dešava kada je neka od komponenti u fokusu (selektovana), a korisnik pritisne ili pusti hardversko dugme na uređaju.	***boolean onKey(*** *View v, int kk, KeyEvent event* ***)***: *v* – komponenta izvor događaja *kk* – kôd pritisnutog dugmeta *event* – objekat koji sadrži sve informacije o događaju
OnFocusChangeListener Aktivira se prilikom promjene fokusa sa jedne komponente na drugu, tačnije kada se neka od komponenti stavi van fokusa.	***void onFocusChange(*** *View v,* *boolean hasFocus* ***)***: *v* – komponenta izvor događaja *hasFocus* – indikator fokusa komponente
OnCreateContextMenuListener U Android programskim okruženjima moguće je implementirati tzv. „context menu“, koji se otvara prilikom dugog dodira na komponentu. Ovaj osluškivač se aktivira prilikom kreiranja tog menija.	***void onCreateContextMenu(*** *ContextMenu menu,View v, ContextMenu.ContextMenuInfo menuInfo* ***)***: *menu* – kreirani meni *v* – komponenta na koju je izvršen dugi klik *menuInfo* – dodatne informacije o meniju

Slika 3. Tabela najčešće korišćenih osluškivačkih interfejsa

3.2.3 Fragmenti i navigacioni drawer



Slika 4. Životni ciklus fragmenta

Fragment je djelić korisničkog interfejsa koji se vezuje za aktivnost i koji funkcioniše u sklopu nje sa određenom autonomijom, ali koji ne može postojati nezavisno. Grubo gledano, fragmenti imaju istu osnovnu strukturu kao i aktivnosti (*.xml* layout fajl koji definiše grafički interfejs i Java klasa koja definiše funkcionalnost), ali se ne tretiraju kao odvojena aktivnost, već kao podinterfejs koji ima zasebni životni ciklus ali koji opet zavisi od životnog ciklusa aktivnosti za koju je vezan. Java klasa kojom se modeluje fragment mora biti izvedena iz klase *Fragment*. Iako se u praksi obično koriste fragmenti koji imaju grafički interfejs, to nije obaveza. Koriste se, mada znatno rjeđe, i fragmenti bez korisničkog interfejsa koji rade u pozadini aktivnosti izvršavajući određene operacije za njene potrebe, bez posjedovanja sopstvenog korisničkog interfejsa.

Motivacija za uvođenje ovog koncepta nastala je iz potrebe za stvaranjem dinamičnijeg korisničkog interfejsa. Na primjer, ukoliko je potrebno „u hodu“ mijenjati veći dio grafičkog interfejsa, dovoljno je na mjesto jednog fragmenta postaviti drugi, pri tom ne mijenjajući aktivnost, ili, još gore, pojedinačne elemente interfejsa u okviru iste aktivnosti. Na taj način se štedi i vrijeme i memorijski prostor, naročito kada se radi o komplikovanijem interfejsu. Sem toga, fragmenti mogu da posluže i kada se žele razdvojiti i grupisati djelovi korisničkog interfejsa koji sami po sebi čine jednu cjelinu. Na ovaj način se promoviše enkapsulacija i „recikliranje“ kôda. Uzmimo za primjer VECTOR INSPECT aplikaciju: srž aplikacije čine četiri funkcionalnosti – osnovna funkcionalnost za slanje fotografija, interaktivna mapa, anketa, i sekcija sa podacima o projektu. Navigacija kroz aplikaciju, odnosno odabir jedne od ove četiri funkcionalnosti se odvija pomoću navigacionog drawer-a. Ideja je da drawer stalno bude dostupan, kao i meni za promjenu jezika, a da se izgled interfejsa mijenja u zavisnosti od odabrane funkcionalnosti. Ovo je realizovano pomoću jedne aktivnosti koja ima statični dio - toolbar sa dugmadima za pristup drawer-u i meniju za promjenu jezika, i dinamični dio koji poprima drugačiji izgled svaki put kada se odabere druga funkcionalnost. Svaka od funkcionalnosti je realizovana u odvojenom fragmentu, tako da je prilikom odabira dovoljno zamijeniti fragment i dobiti željeni izgled korisničkog interfejsa bez potrebe mijenjanja pojedinačnih elemenata. Takođe, ukoliko bi se ova aplikacija razvijala i za tablet uređaje, gdje je dijagonala ekrana značajno veća, pa je samim tim prilikom dizajniranja interfejsa potrebno rasporediti komponente na drugačiji način, ne bi morali da radimo skroz novi dizajn već bi dovoljno bilo umjesto navigacionog drawer-a u jednom dijelu ekrana postaviti fiksan, stalno prisutan meni, dok bi ostatak ekrana rezervisali za prikaz već dizajniranih fragmenata. Dakle, izgled interfejsa bi se mogao prilagoditi uz minimalnu izmjenu kôda.

Drawer-i su vrlo zgodan način za kretanje kroz aplikaciju, naročito kada je u pitanju aplikacija za mobilne telefone, gdje je, za razliku od tablet uređaja, prostor za planiranje grafičkog interfejsa prilično ograničen. Oni imaju mogućnost da se pozovu po potrebi umjesto da budu stalno prisutni, te na taj način ne zauzimaju dodatni prostor. Budući da se često koriste, Android Studio ima mogućnost automatskog generisanja aktivnosti sa drawer-om. U tom slučaju, dovoljno je već generisani šablon izmijeniti za potrebe aplikacije, i prilagoditi dizajn. Ista „olakšica“ je primijenjena i prilikom razvoja VECTOR INSPECT aplikacije.

Životni ciklus fragmenta je dosta sličan životnom ciklusu aktivnosti. I fragment kao i aktivnost sadrži metode kojima se mogu definisati operacije koje će se obavljati u različitim fazama: *onCreate(), onResume(), onPause(), onStop()*. Sem ovih, fragmenti imaju i metodu *onCreateView()*, koja se poziva onda kada se fragment treba prikazati na ekranu uređaja. Ova metoda kao rezultat vraća objekat tipa *View* (dakle, vizuelna komponenta) koji će se postaviti na naznačeno mjesto u aktivnosti. U slučaju da fragment nema korisnički interfejs, ovu metodu nije potrebno preklapati. Budući da je fragment ugrađen u aktivnost, njegov je životni ciklus zavistan od životnog ciklusa te aktivnosti. Dakle, ukoliko se aktivnost nalazi u *Pause* ili *Stop* fazi, to isto važi i za sve njene fragmente. Ukoliko se aktivnost zaustavlja, zaustavljaju se i svi njeni fragmenti. Fragmentu se dozovljava nezavistan životni ciklus tek kada njegova aktivnost uđe u aktivnu, „running", fazu. S obzirom da su na ovaj način povezani, fragment može u svakom trenutku dobiti referencu na aktivnost unutar koje je sadržan, a isto tako i aktivnost može dobiti referencu na svaki fragment koji se nalazi u njoj. Unutar svake klase koja nasleđuje klasu *Fragment*, može se pozvati metoda *getActivity()* koja će kao rezultat vratiti referencu na aktivnost na koju je taj fragment nakačen. Takođe, i aktivnost može dobiti referencu na fragment preko metode *findFragmentById(...)* koja kao argument prima cjelobrojni ID fragmenta, sadržan u ***R*** klasi, a koja je sastavni dio fragment menadžera. Referenca na fragment menadžer se dobija pomoću metode *getFragmentManager()*. Fragment menadžer je klasa koja je neizostavna u procesu upravljanja fragmentima, i koja, između ostalog, omogućava dodavanje fragmenta, postavljanje jednog fragmenta na mjesto drugog, dobijanje reference na fragment i sl. Takođe, ona posjeduje i metode za upravljanje fragmentskim stekom: metoda *addToBackStack()*, koja je sastavni dio procesa transakcije fragmenta, dodaje fragment na stek (kada se prilikom zamjene fragmenata pozove ova metoda, zamijenjeni fragment se privremeno zaustavlja i stavlja na stek, i u svakom trenutku se može pozvati nazad), *popBackStack()* vraća poslednji fragment sa steka (simulira pritisak na back dugme), *addOnBackStackChangedListener()*, registruje osluškivač koji reaguje na promjene na steku, i sl.

Fragmenti se mogu „prikačiti" za aktivnost na dva načina: definisanjem fragmenta kao komponente *.xml* fajla te aktivnosti, ili unutar kôda Java klase te aktivnosti. Prilikom razvoja VECTOR INSPECT aplikacije korišćen je drugi način, budući da se dinamičko mijenjanje fragmenata (što je u ovom slučaju bilo neophodno jer fragmenti treba da se smjenjuju na osnovu odabrane stavke menija) može obaviti jedino tako. Cjelokupan mehanizam u ovoj aplikaciji je realizovan na sljedeći način:

```
public class MainActivity extends AppCompatActivity
        implements NavigationView.OnNavigationItemSelectedListener {

          Fragment fragment;
          DrawerLayout drawer;

          ...

       @Override
       protected void onCreate(Bundle savedInstanceState) {

             setContentView(R.layout.activity_main);

             drawer = (DrawerLayout) findViewById(R.id.drawer_layout);

             ActionBarDrawerToggle toggle = new ActionBarDrawerToggle(this, drawer, toolbar,
                         R.string.navigation_drawer_open, R.string.navigation_drawer_close){

                  @Override
                  public void onDrawerStateChanged(int newState) {
                      super.onDrawerStateChanged(newState);

                      if( newState == DrawerLayout.STATE_IDLE){

                                        ...
```

```
                                        ...

                                FragmentManager fragmentManager = getSupportFragmentManager();
                                FragmentTransaction transaction =
                                                                fragmentManager.beginTransaction();
                                transaction.setCustomAnimations(R.anim.slide_left,
                                        R.anim.slide_right);

                                transaction.replace(R.id.content_main, fragment);
                                transaction.commit();

                            }

                        }

                };

                Drawer.setDrawerListener(toggle);
                toggle.syncState();

                NavigationView navigationView = (NavigationView)
                                                                findViewById(R.id.nav_view);
                navigationView.setNavigationItemSelectedListener(this);

        }

        @Override
        public boolean onNavigationItemSelected(MenuItem item) {

            int id = item.getItemId();

            if (id == R.id.nav_home) {
                getSupportActionBar().setTitle(R.string.nav_home);
                fragment = new FragmentPocetna();
            } else if (id == R.id.nav_map) {
                getSupportActionBar().setTitle(R.string.nav_map);
                fragment = new FragmentMapa();
            } else if (id == R.id.nav_anketa) {
                getSupportActionBar().setTitle(R.string.nav_anketa);
                fragment = new FragmentAnketa();
            } else if (id == R.id.nav_info) {
                getSupportActionBar().setTitle(R.string.nav_info);
                fragment = new FragmentInfo();
            }

            DrawerLayout drawer = (DrawerLayout) findViewById(R.id.drawer_layout);
            drawer.closeDrawer(GravityCompat.START);

            return true;

        }

}
```

Klasa *MainActivity* je Java kôd istoimene aktivnosti i kao takva nasljeđuje klasu *AppCompatActivity*, ali sa druge strane implementira i interfejs *NavigationView.OnNavigationItemSelectedListener*. Budući da implementira ovaj interfejs, klasa *MainActivity* igra i ulogu osluškivača za navigacioni meni. Callback metoda ovog osluškivača je *onNavigationItemSelected(...)*, i ta metoda je preklopljena kako bi na odgovarajući način reagovala na korisnički odabir neke od stavke iz drawer menija. Ta metoda kao argument prima objekat tipa *MenuItem*, koji joj se automatski prosljeđuje prilikom pozivanja, i koji sadrži sve podatke o odabranoj stavki menija. U našem slučaju, korišćen je samo podatak o ID-ju, na osnovu kojeg možemo da klasifikujemo događaj i isprogramiramo osluškivač da na osnovu toga odreaguje na adekvatan način, u ovom slučaju promjena naslova na toolbar-u preko *getSupportActionBar().setTitle(...)* metode, i instanciranje klase odgovarajućeg fragmenta. Na ovaj način je osluškivač menija pripremljen i preostalo je samo da se izvrši njegova registracija za odgovarajući objekat. U *onCreate()* metodi je, prije svega, layout aktivnosti vezan za njenu Java klasu, kreiran je objekat tipa *DrawerLayout* i vezan za *.xml* fajl koji definiše izgled toga drawer-a, a zatim je kreiran objekat tipa *ActionBarDrawerToggle* čija referenca se čuva u promjenljivu *toggle*. Ova promjenljiva omogućava upravljanje položajem drawer-a (otvaranje i zatvaranje), bilo prevlačenjem prsta preko ekrana, bilo klikom na ikonicu na toolbar-u u

gornjem lijevom uglu. U okviru ove klase nalazi se callback metoda *onDrawerStateChanged(...)* koja se poziva prilikom svake promjene stanja drawer-a. Za ovu priliku je preklopljena tako da se prilikom svakog zatvaranja drawer-a (odnosno, prelaska u idle stanje) vrši zamjena fragmenata. Namjerno je isprogramirano da se izmjena fragmenata vrši kada drawer uđe u idle stanje, a ne paralelno sa njegovim kretanjem, zarad poboljšana performansi i rasterećenja procesora. Kao što je ranije napomenuto, svaka manipulacija sa fragmentima započinje sa fragment menadžerom. Referenca na fragment menadžer date aktivnosti se može dobiti pozivanjem metode *getSupportFragmentManager()*, i ta referenca se čuva u promjenljivu. Zatim se preko objekta tipa *FragmentManager* uzima referenca na objekat tipa *FragmentTransaction*, pozivanjem metode *beginTransaction()* koja je članica klase *FragmentManager*. Iako je, generalno govoreći, *FragmentManager* klasa koja je zadužena za sve tipove upravljanja fragmentima, ipak je klasa *FragmentTransaction* ta koja ima neposrednu odgovornost za dodavanje i smjenjivanje fragmenata, te stoga i nju moramo instancirati. Dalje, metodom *replace(...)* naglašavamo koji fragment želimo da postavimo i u koju komponentu aktivnosti (u tu komponentu će se smjestiti objekat tipa *View* kojeg vraća metoda *onCreateView()*), a zatim metodom *commit()* započinjemo tu transakciju (dodavanje fragmenta, odnosno zamjena). Opciono, metodom *setCustomAnimation(...)* se može dodati ulazna i izlazna animacija za smjenu fragmenata (animacije se definišu kao *.xml* fajlovi), ili se zamijenjeni fragment može dodati na stek metodom *addToBackStack()*. Sada samo preostaje da se objekat *toggle* registruje kao osluškivač za drawer, a cjelokupna klasa *MainActivity* kao osluškivač za meni drawer-a, budući da je ova klasa implementirala odgovarajući osluškivački interfejs. Metodom *syncState()*, omogućeno je da karakteristična animacija drawer dugmeta na toolbar-u bude sinhronizovana sa kretanjima drawer-a. Na ovaj način je realizovan mehanizam koji se može sresti kod velikog broja aplikacija kao osnovni način za kretanje kroz njene funkcionalnosti.

3.2.4 Obrada izuzetaka

Prilikom dizajniranja aplikacije vrlo je bitno da predvidimo sve moguće greške u funkcionisanju koje se mogu desiti kako zbog neočekivanih situacija koje mogu nastati u radu, tako i zbog nepravilnog rukovanja korisnika, što u velikom broju slučajeva može dovesti do „pucanja“ aplikacije. Dakle, bitno je zaštititi aplikaciju i od internih nepravilnosti i propusta u kôdu, kao i od samih korisnika. U tu svrhu u Android programiranju se koristi mehanizam obrade izuzetaka, odlično poznat i široko primjenjivan koncept ne samo u Java programiranju, već i u svim ostalim objektno-orjentisanim programskim jezicima.

Izuzetak je indikator koji ukazuje na problem u izvršavanju nekog programa, koji može nastati iz najrazličitijih razloga. Obrada izuzetaka u Android programiranju se ne razlikuje previše u odnosu na obradu izuzetaka u klasičnom Java programiranju. Dakle, programer ima mogućnost da primijeni već dobro formiran mehanizam koji prilikom nailaska na nepravilnost „baca“ izuzetak, a zatim, bazirano na tipu izuzetka, pokreće odgovarajuće rutine koje imaju za cilj da „izbave“ program iz tog problema. Model na kom se bazira ovaj mehanizam je poznat pod imenom **terminacioni model obrade izuzetaka**. Ovo znači da se izvršavanje programa prekida u tački bacanja izuzetka, i da se kreće sa izvršavanjem rutina kojima se izuzetak obrađuje. Nasuprot ovog modela stoji **nastavljački model** prema kom se izvršavanje programa nastavlja nakon završetka obrade izuzetka. Obrada izuzetaka u Android programiranju se realizuje na sledeći način:

```
try {
    //Dio kôda u kom može nastati problem
} catch (Exception e) {
    //Obrada izuzetka
}
```

Ovo je najjednostavnija moguća implementacija mehanizma za obradu izuzetaka. Njegovu srž čini tzv. *try-catch* blok. Princip je sljedeći: dio kôda za koji smatramo da može da stvori problem u radu se uokviruje u *try* blok. Sada će, umjesto da dođe do pucanja aplikacije ili nekog neočekivanog ponašanja usled nastanka problematične situacije, program prekinuti svoje izvršavanje tačno na onoj liniji kôda gdje se problem desio i pri tom će baciti odgovarajući izuzetak i preći na izvršavanje kôda koji se nalazi unutar *catch* bloka. Nakon toga program nastavlja sa izvršavanjem ostatka kôda koji se nalazi van *try-catch* bloka (sem u slučajevima kada se unutar *catch* bloka prekine izvršavanje kompletnog programa, npr. pozivom metode *System.exit(1)*). Treba naglasiti da se prilikom bacanja izuzetka ne prekida uvijek izvršavanje cijelog programa, već se samo prekida nit u kojoj je došlo do bacanja izuzetka. U specijalnom slučaju kada program sadrži samo jednu nit, bacanje izuzetka značiće i prekidanje izvršavanja cjelokupnog programa. Bačeni izuzetak je objekat tipa *Exception*, i u njemu su smješteni svi podaci o bačenom izuzetku, između ostalih i poruka izuzetka koji je možda i najviše upotrebljavan podatak prilikom njegove obrade. Naime, svaki izuzetak sadrži i svoju jedinstvenu poruku (objekat tipa *String*) u kojoj je opisan problem koji se dogodio. Ova poruka se može prikazati u vidu toast poruke, dodati na log, i sl.

U ovom krajnje jednostavnom primjeru, prikazano je da *catch* blok hvata izuzetak tipa *Exception*, i gotovo da nije pogrešno reći da je ovo i jedini tip izuzetka koji se može baciti. Naime, klasa *Exception*, zajedno sa klasom *Error*, je izvedena iz klase *Throwable* i sve klase direktno ili indirektno izvedene iz nje mogu da se bacaju i hvataju unutar *try-catch* bloka. Osnovna razlika između klasa *Exception* i *Error* je ta što *Exception* obuhvata one izuzetke nakon čije obrade aplikacija može nesmetano nastaviti sa radom, dok *Error* obuhvata fatalne greške u radu programa nakon koje se isti mora zaustaviti (npr. nedostatak memorije ili prepunjenost steka). Kako su izuzeci tipa *Error* izuzeci koje automatski baca virtuelna mašina prilikom nastanka ovakvih situacija, to se programerima savjetuje da prilikom pisanja kôda isključivo programiraju bacanje izuzetaka tipa *Exception*, a da *Error* ostave virtuelnoj mašini (iako razvojno okruženje ne bi javilo grešku ni prilikom bacanja izuzetka tipa *Error*). Iz klase *Exception* direktno je izvedeno preko trideset potklasa (iz kojih je, dalje, izvedeno još potklasa), što omogućava da prilikom bacanja i hvatanja naglasimo o kom tipu izuzetka se radi (npr. *IOException*, *ArithmeticException* i sl.). Pogledajmo sada jedan složeniji primjer:

```
...
try {
    if(ind)
        throw new ArithmeticException(" Dogodio se izuzetak tipa 'ArithmeticException' ");
} catch (InputMismatchException e) {
    //Ovaj blok obrađuje izuzetke tipa 'InputMismatchException'
} catch (ArithmeticException e) {
    //Ovaj blok obrađuje izuzetke tipa 'ArithmeticException'
} finally {
    //Ovaj blok se izvršava u svakom slučaju
}
...
```

Kao što vidimo, bacanje izuzetaka se ne mora nužno dogoditi automatski, već je to moguće i programerski definisati. U ovom primjeru bačen je izuzetak tipa *ArithmeticException* a kao argument konstruktora proslijeđen je objekat tipa *String* koji predstavlja poruku tog izuzetka. Naravno, moglo se definisati bacanje i bilo kog drugog tipa izuzetka. Pri izradi VECTOR INSPECT aplikacije nije se previše komplikovalo sa *try-catch* blokovima, tako da je uvijek kada je to bilo potrebno bacan i hvatan izuzetak tipa *Exception* (tip

koji obuhvata sve podtipove), kom je kao argument prosljeđivana odgovarajuća poruka. Hvatanje izuzetaka se može razdvojiti na više *catch* blokova tako što će svaki od njih biti specijalizovan za hvatanje i obradu određenog tipa izuzetka. Ovdje treba voditi računa da dva *catch* bloka ne mogu da hvataju izuzetke istog tipa, kao i da *catch* blokovi moraju biti poređani tako da se prvo navedu oni koji hvataju izuzetke hijerarhijski nižeg tipa (npr. prvo bi se naveo *catch* blok koji hvata izuzetak tipa *ArithmeticException*, pa tek onda blok koji hvata izuzetak tipa *Exception*) Opciono, za *try-catch* blokom može uslijediti i *finally* blok čiji će se kôd izvršavati bez obzira na to da li je došlo do bacanja izuzetka u *try* bloku. Obrada izuzetaka se može organizovati i po dubini. Drugim riječima rečeno, moguće je postaviti *try* blok unutar drugog *try* bloka. U takvoj hijerarhiji i *catch* blok može da baca izuzetke (što bi bilo korisno u nekim situacijama kada on sam ne može u potpunosti da obradi izuzetak) te se tada hvatanje tog izuzetka obavlja u *catch* bloku koji je vezan za *try* blok višeg nivoa. Takođe, ukoliko kreiramo neku novu metodu koja u datim situacijama može da baci izuzetak, nije potrebno formirati novi try-catch blok unutar te metode već je dovoljno naglasiti prije tijela metode da ona može baciti izuzetak (npr. *void novaMetoda(...) throws Exception {...}*), te će na taj način ovaj izuzetak obrađivati onaj *try-catch* blok kojim bi metoda bila uokvirena prilikom pozivanja. Ovako definisana metoda se ne može pozvati van *try* bloka.

Kao i u standardnom Java programiranju i u Android programiranju postoji podjela izuzetaka na **provjerene** i **neprovjerene**. Neprovjereni izuzeci su oni izuzeci koji najčešće nastaju usled greške u samom programskom kôdu. Na primjer, ukoliko se zbog pogrešno definisanog indeksa pokuša pristupiti nepostojećem elementu niza tada će se baciti *ArrayIndexOutOfBoundsException* izuzetak koji spada u neprovjereni tip. Provjereni izuzeci su oni koji nastaju ne zbog greške u kôdu već iz nekih drugih razloga. Na primjer, ukoliko pokušamo otvoriti nepostojeći fajl doći će do *FileNotFoundException* izuzetka. Dakle, kôd kojim se otvara fajl je skroz korektan, ali se program ipak ne može izvršiti jer ciljani fajl ne postoji. Ovakvi tipovi izuzetaka se tretiraju sa posebnom pažnjom u Java programskom jeziku, te programeru, stoga, nije dozvoljeno da van *try-catch* bloka poziva metode koje su definisane tako da bacaju neki od izuzetaka ovoga tipa:

```
...

try {

    BufferedOutputStream bos = new BufferedOutputStream(new FileOutputStream(imageSave));
    bos.write(imagesForSending.get(i));
    bos.flush();
    bos.close();

} catch (Exception e) {

}

...
```

Dakle, iako je programski kôd unutar *try* bloka sasvim ispravan, i iako unutar *catch* bloka nije definisana nijedna naredba za obradu izuzetaka, ukoliko bi navedene linije kôda bile napisane bez *try-catch* bloka koji ih okružuje, kompajler bi javio grešku budući da konstruktor *FileOutputStream(...)* i metoda *flush()* mogu baciti izuzetke *FileNotFoundException* i *IOException*, respektivno, koji su provjerenog tipa. Na ovaj način, Javini dizajneri su napravili programski jezik koji je sâm sposoban da već u startu preduprijedi neke dosta česte greške, tako što sugeriše programeru na situaciju koja bi mogla da onemogući pravilno izvršavanje njegovog programa, te mu na taj način štedi i vrijeme i trud koji bi inače bio utrošen na lociranje i uklanjanje problema.

3.2.5 Animacije

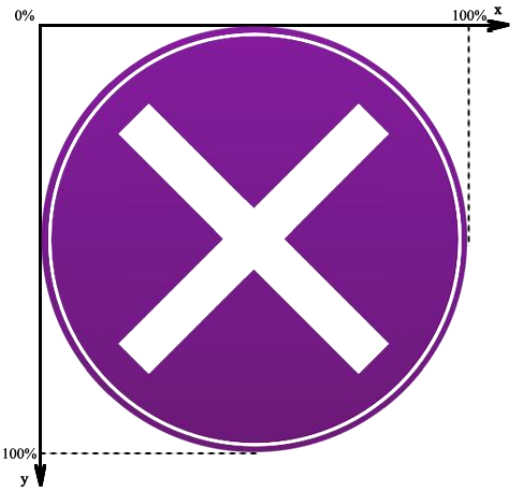
U vrijeme pisanja ovog rada, na Google Play Store-u se nalazilo 2.2 miliona aplikacija, a oko 2 miliona na Apple-ovom App Store-u. Ovaj enormni broj aplikacija koje korisnicima stoje na raspolaganju sugeriše na činjenicu da je veliki broj ideja na osnovu kojih bi se nove aplikacije mogle razvijati već realizovan i da se programeri često odlučuju da unaprjeđuju već postojeća rješenja umjesto da stvaraju nove koncepte. Da bi se neka aplikacija probila u moru ostalih nije dovoljno samo da funkcioniše dobro. Cilj svakog programera koji svoj proizvod pokušava plasirati na tržište jeste natjerati korisnika da se odluči baš za njegovu aplikaciju a ne za neku drugu koja možda funkcioniše podjednako ili približno dobro. Zato se poseban akcenat stavlja na vizuelni izgled aplikacije, prije svega na dizajn i animacije koji aplikaciji podižu kvalitet i čine da na korisnika ostavi izuzetno povoljan utisak. Vodeći se tom logikom, animacije su postale sastavni dio i VECTOR INSPECT aplikacije. Međutim, treba imati na umu da animacije troše određeni dio procesorskih i memorijskih kapaciteta, te stoga, ukoliko se koriste pretjerano ili u neadekvatnom trenutku, mogu umanjiti performanse aplikacije što kod korisnika može izazvati suprotan efekat od onoga koji se animacijama uopšte želio postići.

Animacije se u Android Studiju mogu realizovati na dva načina: definisanjem animacija u XML-u i vezivanjem za odgovarajuću komponentu interfejsa, ili koristeći tzv. property animation sistem. Prvi način obuhvata definisanje animacije unutar *.xml* fajla koji se smješta u poseban folder (*res.anim* folder), a zatim se tom animacijom upravlja odgovarajućim metodama članicama klase *Animation*. Struktura *.xml* fajla animacije je sledeća:

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android">

    <scale
        android:pivotX="50%"
        android:pivotY="50%"
        android:fromXScale="0.0"
        android:fromYScale="0.0"
        android:toXScale="2.0"
        android:toYScale="2.0"
        android:interpolator="@android:anim/accelerate_interpolator"
        android:duration="500"
        />

    <scale
        android:startOffset="500"
        android:pivotX="50%"
        android:pivotY="50%"
        android:fromXScale="1.0"
        android:fromYScale="1.0"
        android:toXScale="0.5"
        android:toYScale="0.5"
        android:interpolator="@android:anim/decelerate_interpolator"
        android:duration="300"
        />

    <rotate
        android:startOffset="800"
        android:pivotX="50%"
        android:pivotY="50%"
        android:fromDegrees="0"
        android:toDegrees="45"
        android:interpolator="@android:anim/linear_interpolator"
        android:duration="300"
        />

    <alpha
        android:startOffset="800"
        android:fromAlpha="1.0"
        android:toAlpha="0.4"
        android:interpolator="@android:anim/linear_interpolator"
        android:duration="300"
        />

</set>
```

Animacija bi se, u suštini, mogla definisati kao postepena promjena nekog od svojstava vizuelnog objekta interfejsa, te se, u skladu s tim, i formira *.xml* fajl. Korjeni element je <*set*> koji definiše višestruku koordinisanu promjenu svojstava. Dakle, ovdje se radi o složenoj animaciji. Svi unutrašnji elementi definišu promjenu tačno jednog od svojstava i to onog čiji naziv korespondira nazivu tag-a. Drugim riječima, svaki od unutrašnjih elemenata čini pojedinačnu animaciju koje udružene u jedan <*set*> elemenat daju složenu animaciju. Svi elementi će se izvršavati istovremeno, sem ukoliko to nije drugačije regulisano njihovim osobinama. Ukoliko bi animacija bila bazirana na promjeni samo jednog od svojstava, *set* elemenat ne bi bio potreban. U ovom slučaju animacija se bazira na promjeni dimenzija, tj. skaliranju, rotaciji i promjeni transparentnosti objekta. Osvrnućemo se sada na jedan od unutrašnjih elemenata (konkretno, drugi po redu <*scale*> elemenat) i sagledati njegove osobine.

Kao što je već rečeno, svaki od elemenata počinje i završava odgovarajućim XML tag-om čiji naziv ukazuje na osobinu objekta na koju taj element utiče. U ovom slučaju je to razmjera. Prva navedena osobina je *startOffset* kojom se definiše koliko želimo da odgodimo početak animacije u odnosu na trenutak kada odgovarajućom metodom u Java kôdu startujemo tu animaciju koju smo prethodno vezali za neki element interfejsa. U ovom slučaju to odgađanje je 500 milisekundi. Osobina *duration* definiše trajanje animacije. Prvi <*scale*> elemenat ima trajanje od 500 milisekundi koje je jednako vrijednosti *startOffset* osobine drugog. Drugim riječima rečeno, startovanje drugog elementa će pričekati 500 milisekundi, odnosno drugi element će početi sa izvršavanjem tek nakon što se prvi element izvrši. Na isti način, treći element „čeka“ da se prvi i drugi izvrše da bi se i on pokrenuo, te je njegov *startOffset* jednak zbiru trajanja prva dva elementa. Četvrti element ima istu vrijednost *startOffset*-a i trajanja kao i treći, što ukazuje na to da se on izvršava istovremeno sa njim. Dakle, ovako definisana animacija odvija se u tri koraka: prvo skaliranje, zatim drugo skaliranje, pa, konačno, istovremeno rotiranje i povećanje transparentnosti.



Slika 5. Koordinatni sistem objekta grafičkog interfejsa

Vratimo se na drugi <*scale*> element. Nakon definisanog odgađanja starta, definisane su osobine *pivotX* i *pivotY*. Ovim dvjema osobinama se određuje fiksna tačka oslonca (pivot) na objektu koja se ne može pomjerati i oko koje se vrše sve promjene, i definiše se onda kada želimo da mijenjamo dimenzije ili rotiramo objekat. Osobina se definiše u procentima u odnosu na koordinatni sistem relativan za objekat za kog je animacija vezana, gdje 0% odgovara koordinatnom početku (gornji, lijevi ugao objekta), dok 100% odgovara maksimalnoj vrijednosti koordinate (širina odnosno visina objekta - slika 5). U slučaju skaliranja, to je tačka od koje, odnosno, ka kojoj se objekat širi ili skuplja, a u slučaju rotacije to je tačka oko koje objekat rotira. Naravno, u slučaju promjene transparentnosti, definisanje ove tačke nema nikakvog smisla. U ovom slučaju za tačku oslonca je uzet centar objekta (50% po *x* i 50% po *y* osi). Zatim se osobinama *fromXScale, toXScale, fromYScale* i *toYScale* određuje od koje do koje vrijednosti će se kretati koeficijent skaliranja objekta. Skaliranje se obavlja odvojeno po *x* i *y* osi i promjena dimenzije po jednom pravcu ni na koji način ne utiče na promjenu dimenzije po drugom. Ukoliko želimo da promijenimo dimenzije objekta, a da pri tom zadržimo njegove proporcije potrebno je sami da definišemo istovjetne stepene skaliranja po objema osama. U ovom primjeru, razmjera objekta je u oba pravca promijenjena sa 1.0 na 0.5, što znači da ova animacija postepeno smanjuje veličinu objekta na pola. Zatim se definiše *interpolator*. Ova osobina definiše način na koji će se vršiti tranzicija sa početnih vrijednosti na krajnje. Drugim riječima, definiše efekat prelaza. U ovom slučaju odabran je „decelerate interpolator“ koji će promjenu svojstva objekta (u ovom slučaju razmjeru) učiniti bržom u početku, a sporijom kako se vrijednost primiče konačnoj (naravno, ovo ne remeti trajanje promjene koje je definisano osobinom *duration*). Dakle, interpolator definiše efekat tranzicije. Vrlo često

se koristi, iako ne u ovom primjeru, i osobina *repeatCount* kojom se definiše broj ponavljanja date animacije.

Nakon što je *.xml* fajl animacije definisan, potrebno je tu animaciju definisati u Java kôdu kao objekat tipa *Animation*, a zatim je vezati za odgovarajući objekat grafičkog interfejsa:

```
Animation xAnimation = AnimationUtils.loadAnimation(getBaseContext(), R.anim.x_anim);
xAnimation.setFillAfter(true);
        ...
x.startAnimation(xAnimation);
        ...
x.clearAnimation();
```

Objekat *xAnimation* se instancira pomoću metode *loadAnimation(...)*, koja je članica klase *AnimationUtils* i kojoj se mora proslijediti kontekst aktivnosti u kojoj se prikazuje animacija kao prvi argument, i *.xml* fajl kojim je animacija definisana, tačnije njen cjelobrojni ID, kao drugi. Kao što vidimo, sem identifikacionih brojeva komponenti interfejsa, *R* klasa sadrži i identifikacione brojeve resursa koji su smješteni u različitim unutrašnjim klasama, u zavisnosti od tipa. Opciono, može se pozvati metoda *Animation.setFillAfter(**true**)*, čime se obezbjeđuje zadržavanje krajnje vrijednosti osobine objekta koja je mijenjana animacijom i nakon što se animacija završi. U suprotnom, izmijenjena osobina objekta bi se vratila na početnu vrijednost po završetku animacije. Ukoliko ne pozovemo ovu metodu, promjenljiva *Animation.fillAfter* ima podrazumijevanu vrijednost ***false***. Pokretanje animacije se vrši metodom *startAnimation(...)*. Nakon završetka animacije, poželjno je nad objektom pozvati metodu *clearAnimation()* koja će „očistiti“ animaciju vezanu za taj objekat, i na taj način rasteretiti RAM. Naravno, sem navedenih, klasa *Animation* sadrži još mnoge metode koje mogu upravljati animacijama, ali ćemo se ovdje zadržati samo na onima koje su pozivane prilikom kodiranja VECTOR INSPECT aplikacije.

Za objekat tipa *Animation* se, opciono, može registrovati osluškivač koji omogućava programiranje obrade događaja za različite faze izvršavanja animacije. Ovaj osluškivač implementira *Animation.AnimationListener* interfejs.

```
xAnimation.setAnimationListener(new Animation.AnimationListener() {

    @Override
    public void onAnimationStart(Animation animation) {
        //Kôd koji se izvršava pri startovanju animacije
    }

    @Override
    public void onAnimationEnd(Animation animation) {
        //Kôd koji se izvršava po završetku animacije
    }

    @Override
    public void onAnimationRepeat(Animation animation) {
        //Kôd koji se izvršava pri ponavljanju animacije
    }

});
```

Interfejs sadrži tri callback metode koje odgovaraju trima fazama: *onAnimationStart(...)* se poziva pri startovanju animacije, *onAnimationEnd(...)* po završetku, a *onAnimationRepeat(...)* prilikom svakog ponavljanja. Sve tri metode kao argument primaju objekat tipa *Animation* koji je ništa drugo nego animacija za koju je osluškivač vezan.

Drugi način animiranja GUI objekata je korišćenjem **property animator**-a. Iako je krajnji efekat naizgled isti, razlike u odnosu na prethodni način postoje. Prije svega, klasa *Animation* može da radi isključivo sa objektima interfejsa čiji je klasni tip izveden iz klase *View*, dok kod property animator-a to nije slučaj. Istini za volju, većina najčešće upotrebljivanih GUI objekata je izvedena iz klase *View*, ali opet nije loše imati na umu ovu prednost property animator-a. Ono što je bitnije naglasiti je način na koji se animacije generišu i prikazuju u ova dva slučaja. U prvom slučaju (upotreba klase *Animation*), animacija se generiše tako što komponenta samo prividno mijenja svoje osobine. Ova činjenica može da stvori velike probleme ukoliko animacija uključuje mijenjanje pozicije komponente. Na primjer, ukoliko animaciju vežemo za dugme, i ta animacija definiše pomjeranje dugmeta preko ekrana, vizuelno ćemo dobiti efekat pomjeranja, ali prostor koji reaguje na klik se neće pomjerati zajedno sa vizuelnom predstavom dugmeta. Kod property animator-a ovo nije slučaj. Još jedna razlika u odnosu na upotrebu *Animation* klase (budući da se možda ne može okarakterisati baš kao prednost) je ta što se animacija može u potpunosti definisati unutar Java kôda, bez potrebe korišćenja XML-a i kreiranja *.xml* resursa. Doduše, animaciju je i u ovom slučaju moguće definisati u XML-u, ali način na koji se to radi će ovom prilikom biti izostavljen iz priče, budući da je prilikom razvijanja VECTOR INSPECT aplikacije animiranje objekata korišćenjem property animator mehanizma realizovano isključivo korišćenjem Java koda. Loša strana ovakvog kreiranja animacija je neophodnost pisanja dosta većeg kôda da bi se željena animacija definisala, tako da nema potrebe koristiti ovaj sistem ukoliko klasa *Animation* može u potpunosti da obavi posao.

Property animator mehanizam je cjelokupan sadržan unutar klase *Animator*. Međutim, u praksi se ova klasa nikada ne instancira, već se to čini sa klasama koje su iz nje izvedene: *ValueAnimator* i iz nje izvedena klasa *ObjectAnimator i AnimatorSet*. Klasa *ValueAnimator* sadrži računske metode kojima određuje vrijednost promjenljive osobine animiranog objekta za svaki frejm animacije na osnovu proslijeđene početne i krajnje vrijednosti. Podrazumijevano, frejm se osvježava na svakih 10 milisekundi. Međutim, da bi se osobina objekta zaista izmijenila pri svakom osvježavanju frejma potrebno je to raditi ručno uz upotrebu odgovarajućih osluškivača koji reaguju na svako ažuriranje vrijednosti. Kako bi se ta komplikacija izbjegla, iz klase *ValueAnimator* izvedena je klasa *ObjectAnimator* koja sem početnih i krajnjih vrijednosti odgovarajuće osobine u obzir uzima i objekat čija se osobina mijenja, te tako sadrži metode kojima se sem računanja vrijednosti prilikom osvježavanja frejma ta vrijednost automatski i

```
...

ObjectAnimator scaleX1 = ObjectAnimator.ofFloat(cloudLayout, "scaleX", 0f, 1.1f);
ObjectAnimator scaleY1 = ObjectAnimator.ofFloat(cloudLayout, "scaleY", 0f, 1.1f);
ObjectAnimator scaleX2 = ObjectAnimator.ofFloat(cloudLayout, "scaleX", 1.1f, 1f);
ObjectAnimator scaleY2 = ObjectAnimator.ofFloat(cloudLayout, "scaleY", 1.1f, 1f);

scaleX1.setDuration(250);
scaleY1.setDuration(250);
scaleX2.setDuration(250);
scaleY2.setDuration(250);

scaleX1.setInterpolator(new AccelerateInterpolator());
scaleY1.setInterpolator(new AccelerateInterpolator());
scaleX2.setInterpolator(new DecelerateInterpolator());
scaleY2.setInterpolator(new DecelerateInterpolator());

AnimatorSet set = new AnimatorSet();
set.play(scaleX1).with(scaleY1);
set.play(scaleX1).before(scaleX2);
set.play(scaleX2).with(scaleY2);

set.start();

...
```

dodjeljuje odgovarajućoj osobini objekta. Zbog ovoga se savjetuje da se uvijek kada je to moguće koristi klasa *ObjectAnimator*. Jedino ograničenje koje ova klasa ima, i zbog kog bi se ipak morala koristiti klasa *ValueAnimator* je nemogućnost mijenjanja željene osobine objekta ukoliko za nju ne postoji odgovarajući

seter. Treća pomenuta klasa koja je izvedena iz klase *Animator* je *AnimatorSet*. Ona sadrži metode kojima se više animacija može iskoordinisati kako bi se dobila neka složena animacija. Dakle, ukoliko napravimo paralelu sa prvim načinom animiranja objekata, instance *ObjectAnimator* klase bi odgovarale unutrašnjim elementima *.xml* fajla, dok bi instanca *AnimatorSet*-a odgovarala korjenom <*set*> elementu.

Objasnimo proces animacije objekta korišćenjem property animator-a na praktičnom primjeru. Svaka osobina koja se želi animirati, tačnije, osobina na čijoj će se promjeni animacija bazirati, se mora definisati kao odvojeni objekat tipa *ObjectAnimator*. Ovi objekti se instanciraju pomoću jedne od tri metode klase *ObjectAnimator*: *ofInt(...), ofFloat(...) i ofObject(...)*. Naziv metode sugeriše na tip vrijednosti koji ima osobina koja se želi mijenjati: cijeli broj, realni broj ili neki proizvoljni objekat. Kao prvi argument se prosleđuje objekat koji se želi animirati (u ovom slučaju *cloudLayout*), drugi argument je naziv osobine koja se želi mijenjati (proslijeđen kao objekat tipa *String*), a poslednja dva argumenta definišu početne i krajnje vrijednosti te osobine. Posebnim metodama se definišu svojstva animacije. Metodom *setDuration(...)* se definiše trajanje animacije u milisekundama koje se prosleđuje kao argument metode, a metodom *setInterpolator(...)* se bira tip interpolatora koji će se koristiti prilikom prelaza od početne ka krajnjoj vrijednosti mijenjane osobine. Kao argument ova metoda prima instancu neke od klasa izvedenih iz klase *Interpolator*. Konačno, višestruke animacije se koordinišu pomoću objekta tipa *AnimatorSet*. Ovaj objekat za tu namjenu sadrži veliki broj metoda, međutim za potrebe ove aplikacije korišćene su dvije: *play(...)* u sklopu sa *with(...)* i *before(...)*. Upotreba ovih metoda je prilično intuitivna: sve tri kao argument primaju animaciju kojom upravljaju, a naziv metode sugeriše na njihovu svrhu. Na primjer, poziv *play(a1).with(a2)* će učiniti da se animacije (tačnije objekti tipa *ObjectAnimator*) startuju zajedno, i sl. Dakle, efekat poziva ovog kôda bi bio sledeći: objekat se od nule uvećava na dimenzije 1.1 put veće od naznačenih, koristeći accelerate interopolator u trajanju od 250 milisekundi, a zatim se smanjuje na originalnu veličinu koristeći decelerate interpolator, takođe u trajanju od 250 milisekundi.

3.2.6 Višenitno programiranje

Prilikom pokretanja aplikacije Android operativni sistem započinje novi proces koji toj aplikaciji dodjeljuje. Sa tačke gledišta procesora, **proces** je instanca jednog programa, koja ima zaseban memorijski prostor i ostale resurse koje ne dijeli sa drugim procesima. U većini slučajeva jedna aplikacija pokreće tačno jedan proces, iako postoje implementacije u kojima glavni proces kreira potprocese koji dijele njegove resurse i čiji životni ciklus zavisi od njegovog. **Nit** je sekvenca programskih instrukcija kojima operativni sistem upravlja i koje definišu jedan proces. Podrazumijevano, proces sadrži jednu, glavnu nit, koja može da pokrene nove niti, što je najčešće i slučaj budući da niti smanjuju „prazan hod“ u izvršavanju aplikacije, te stoga koncept višenitnog programiranja omogućava pisanje izuzetno efikasnih aplikacija. Ovo je naročito značajno kada aplikacija sadrži zahtjevne operacije čije izvršenje iziskuje dosta vremena. Ukoliko bi se te operacije izvršavale na glavnoj niti grafički interfejs bi za svo vrijeme izvršavanja bio zamrznut (stvorio bi se prazan hod) što bi značajno umanjilo kvalitet aplikacije. Sve niti koje sačinjavaju jedan proces dijele resurse (što je i logično, budući da su sve dio istog procesa).

Niti imaju precizno definisan životni ciklus, koji se sastoji iz 4 stanja:

1. **new** – stanje u koje nit ulazi odmah nakon kreiranja i traje sve do njenog pokretanja;
2. **runnable** – stanje u koje nit prelazi nakon pokretanja; ovo je stanje u kom nit izvršava svoj zadatak,
3. **waiting** – stanje u koje nit prelazi kada čeka druge niti višeg prioriteta da završe svoje instrukcije i oslobode procesor;
4. **timed waiting –** stanje čekanja koje za razliku od waiting stanja može imati definisano vrijeme trajanja; prelazak u ovo stanje može se i eksplicitno navesti metodom *Thread.sleep(...)*;

5. **blocked** – stanje u koje nit prelazi nakon pokušaja izvršenja instrukcija koje se u datom trenutku ne mogu izvršiti (na primjer, pokušaj pristupa resursu koji već koristi neka druga nit);
6. **terminated** – konačno stanje u koje nit prelazi po uspješnom ili neuspješnom izvršavanju zadataka;

Za vrijeme trajanja stanja waiting, timed waiting i blocked, nit ne može koristiti procesor čak i ukoliko je on slobodan. Što se tiče runnable stanja, operativni sistem njega razdvaja na dva podstanja: **ready** – stanje kojim nit signalizira da želi da zauzme procesorsko vrijeme i u kom čeka na procesor da započne sa izvršavanjem njenih instrukcija, i **running** – izvršno stanje koje započinje kada procesor krene sa izvršavanjem instrukcija. Sa druge strane, sa aspekta virtualne mašine, ova dva stanja se posmatraju isključivo kao jedinstveno runnable stanje. U slučaju kada se nit nižeg prioriteta nalazi u runnable stanju, a nit višeg prioriteta uđe u ready stanje (dakle, zahtijeva procesorsko vrijeme), tada se izvršenje niti nižeg prioriteta prekida (prelazi u waiting stanje) a procesor se prepušta niti višeg prioriteta. Ovo je tzv. **preemptive scheduling**.

U datom trenutku procesor može da izvršava samo jednu nit (pod uslovom da se radi o procesoru sa jednim jezgrom, bez hyper-threading sistema). Stoga se u sklopu operativnog sistema dizajnira mehanizam raspodjele procesorskog vremena zahvaljujući kojem može uporedo biti aktivno više procesa, na taj način što se svakoj niti naizmjenično dodjeljuje određena porcija procesorskog vremena. Sistem koji to reguliše naziva se **planer niti**. Kriterijum koji planer niti koristi prilikom raspodjele procesorskog vremena je nivo prioriteta niti: veći prioritet znači prednost pri izvršavanju. Ukoliko postoji više niti istog prioriteta, tada se procesorsko vrijeme dodjeljuje naizmjenično, a niti nižeg prioriteta čekaju dok se niti višeg prioriteta u potpunosti ne izvrše. Prioriteti niti se rangiraju vrijednostima od 1 do 10. U Android programiranju prioritet se može definisati statičkim promjenljivima članicama klase *Thread*: *MAX_PRIORITY* (10), *NORM_PRIORITY* (5), *MIN_PRIORITY* (1).

U Javi, a samim tim i u Android programiranju, nit se modeluje klasom *Thread*. Ova klasa se može instancirati na dva načina: standardnim pozivom konstruktora kome se kao argument prosleđuje klasa koja implementira interfejs *Runnable*, ili instanciranjem nove klase koja nasleđuje klasu *Thread*. Prilikom razvoja VECTOR INSPECT aplikacije korišćen je isključivo prvi način, tako da ćemo se osvrnuti samo na njega:

```
...

new Thread(new Runnable(){

    @Override
    public void run() {

        //Kôd koji želimo da se izvršava na drugoj niti

    }

}).start();

...
```

Ovdje su stvari prilično jednostavne. Interfejs *Runnable* ima samo jednu metodu, *run()*, u koju se smješta cjelokupan kôd koji će da se izvršava na drugoj niti. Dakle, prilikom pokretanja niti, poziva se upravo ova metoda. Metodom *start()* nit se pokreće. Sem ovih, klasa *Thread* sadrži još neke metode za upravljanje nitima (na primjer, ranije spomenuta *sleep(...)* metoda). Ovakav način upravljanja nitima se naziva direktnim upravljanjem, i sasvim je u redu koristiti ga ukoliko nije predviđeno kreiranje značajnijeg broja niti. U suprotnom, preporučuje se korišćenje **izvršioca** (eng. **executor**). Izvršilac sve niti smješta u tzv. thread pool i u potpunosti preuzima nadzor i upravljanje nad njima. Pogodnost upotrebe izvršioca je ta što eliminiše potrebu za kreiranjem novih niti svaki put kada se pozove neka instanca klase *Runnable*

(tačnije, pozove njena *run()* metoda), već se u tu svrhu može iskoristiti neka od postojećih niti. Na taj način se rasterećuje procesor i dobija na performansama. Drugim riječima, izvršilac se trudi da optimizuje broj niti tako da se iskoriste sve prednosti koje višenitno programiranje pruža, a da se opet izbjegne zagušenje procesora usled prevelikog broja niti. Ipak, budući da su se prilikom programiranja VECTOR INSPECT aplikacije nove niti uglavnom pokretale upotrebom *AsyncTask*-a, o kome će ubrzo biti riječ, nije bilo potrebe za upotrebom izvršioca. Ipak, budući da je ovo fantastičan način da se programerima uštedi vrijeme koje bi inače bilo potrošeno na pisanje komplikovanih sistema za upravljanje i optimizaciju broja niti, vrijedjelo je na kratko se osvrnuti i na ovaj mehanizam.

Kako bi rasteretili glavnu nit aplikacije (nit korisničkog interfejsa), sem kreiranja nove, pozadinske niti, morali bi voditi računa i o komunikaciji između pozivajuće niti (eng. caller thread) i radne niti (eng. worker thread). Upravo spomenuta *AsyncTask* klasa omogućava programeru da definiše operacije koje radna nit treba da izvršava, bez potrebe da vodi računa o njenom upravljanju i komunikaciji sa pozivajućom niti. Način na koji se ova klasa upotrebljava je sledeći:

```
...

new AsyncTask<Bitmap, String, Integer>(){

    @Override
    protected Integer doInBackground(Bitmap... params) {

         int rezultat;

        Bitmap bmp1 = params[0];
        Bitmap bmp2 = params[1];

        //Izvršavanje kôda na pozadinskoj niti

        //Kada je odgovarajući uslov zadovoljen poziva se:
        publishProgress("Neko obavještenje");

        //Vraća neki rezultat
        return new Integer(rezultat);

    }

    @Override
    protected void onProgressUpdate(String... values) {

       String obavjestenje = values[0];

       //Kôd koji se izvršava na pozivačkoj niti.

    }

    @Override
    protected void onPostExecute(Integer integer) {

       int rezultat = integer.intValue();

       //Kôd koji se izvršava na pozivačkoj niti.

    }

}.execute(bitmap1, bitmap2);

...
```

AsyncTask je apstraktna klasa tako da ne možemo instancirati nju, već isključivo klase izvedene iz nje. Klasa sadrži metode koje se mogu preklapati i kojima se definiše kôd za izvršavanje na pozadinskoj nit, kao i kôd koji će se izvršavati na glavnoj niti u skladu sa obavljenim pozadinskim operacijama. Prilikom kôdiranja VECTOR INSPECT aplikacije forsirano je, prvenstveno radi jednostavnosti kôda i preglednosti, instanciranje unutrašnje, anonimne klase koja nasleđuje klasu *AsyncTask*, kao što je prikazano na primjeru. Sem što je apstraktna, *AsyncTask* je i generička klasa, te se prilikom pisanja zaglavlja izvedene klase pored

naziva *AsyncTask* navode i tipovi generičkih parametara u izlomljenim zagradama. *AsyncTask* ima tri takva parametra: prvi se odnosi na onaj tip podataka koji glavna nit šalje pozadinskoj, drugi označava tip podataka koji će se slati glavnoj niti u toku rada pozadinske kako bi imala uvid u trenutno stanje izvršavanja operacija u pozadinskoj niti, i treći parametar označava tip podataka koji se vraća glavnoj niti kao rezultat konačnog izvršenja neke operacije. Zatim se navodi tijelo izvedene klase, u kom se obavezno preklapa metoda *doInBackground(...)*, a proizvoljno i ostale, te se na kraju metodom *execute(...)* pokreće rad pozadinske niti. Kao argumenti ove metode prosleđuju se promjenljive koje predstavljaju podatke koje glavna nit šalje pozadinskoj na obradu. Bitno je naglasiti da generički parametri moraju biti klasnog tipa, te stoga primitivni tipovi podataka nisu dozvoljeni, već se moraju koristiti odgovarajuće omotačke klase. U ovom projektu u *AsyncTask* klasama generalno su preklapane tri metode: *doInBackground(...)*, *onProgressUpdate(...)* i *onPostExecute(...)*. Prva metoda u svom tijelu sadrži kôd koji se izvršava na pozadinskoj niti, a kao argument joj se prosleđuju isti oni parametri koji su proslijeđeni metodi *execute(...)*, koji su ovdje „upakovani“ u niz, te proslijeđenim podacima možemo pristupiti pristupanjem pojedinačnim elementima niza *params*. U okviru ove metode može se pozivati metoda *publishProgress(...)* koja automatski poziva *onProgressUpdate(...)* metodu kojoj ujedno prosleđuje i svoje argumente, takođe „spakovane“ u niz objekata, kojima se glavnoj niti želi saopštiti neko obavještenje o radu pozadinske niti. U ovom primjeru, kao obavještenje se šalje jedan objekat tipa *String*. Konačno, kada se izvrši cjelokupan kôd namijenjen za pozadinsku nit, *doInBackground(...)* vraća rezultat, poziva se *onPostExecute(...)* metoda, i prosleđuje joj se taj rezultat kao argument. Ovoga puta, rezultat se ne pretvara u niz parametara, već se samo prosleđuje.

Još jedna vrlo bitna klasa kada je riječ o upravljanju nitima je klasa *Handler* koja se najčešće koristi u svrhu odgađanja pokretanja određenog dijela kôda na istoj niti, ili za pokretanje neke sasvim nove niti u željenom trenutku. U kôdu ove aplikacije bilo ju je potrebno koristiti jedino za prvu svrhu. Ovo odgađanje izvršavanja je realizovano metodom *postDelayed(...)*, gdje se kao prvi argument prosleđuje instanca klase koja implementira interfejs *Runnable* (gdje se, unutar *run()* metode smješta kôd čije se izvršavanje želi odgoditi), a kao drugi vrijeme odgađanja u milisekundama.

Naravno, ovo je samo mali dio priče koja bi se mogla ispričati na temu koncepta višenitnog programiranja. Budući da je oblast izuzetno široka, ovdje je stavljen fokus samo na onaj dio koji je upotrijebljen u realizaciji ove aplikacije.

3.2.7 Komunikacija sa serverom

Ključna funkcionalnost VECTOR INSPECT aplikacije je ostvarivanje veze sa web serverima i razmjena podataka. Stoga je od posebne važnosti pravilno upotrijebiti klase i metode koje stoje na raspolaganju za ovu svrhu. Slanje podataka se odvija u dva dijela: prvi, slanje podataka kojima se popunjava baza putem HTTP protokola i drugi, slanje fotografija na FTP server. Detaljniji opis ovog procesa biće iznijet u narednom poglavlju, dok ćemo se ovdje dotaći načina na koji se ova dva protokola implementiraju u program.

Ostvarivanje HTTP konekcije i upotreba protokola za razmjenu podataka se može obaviti na dva načina: upotrebom ugrađene klase *HttpURLConnection*, ili upotrebom neke od eksternih biblioteka koje su specijalizovane za ovu namjenu (na primjer *Volley* biblioteka). U našem slučaju, korišćena su oba načina. I, naravno, da bi se povezivanje na internet uopšte moglo ostvariti neophodno je dodati odgovarajuću dozvolu u *AndroidManifest.xml* fajlu:

```
<uses-permission android:name="android.permission.INTERNET" />
```

Prvi način je korišćen za potrebe testiranja dostupnosti internet konekcije. Cilj testiranja nije bio samo provjera povezanosti uređaja na baznu stanicu (tačnije, da li su Wi-Fi ili paketni prenos podataka uključeni), već da li uređaj zaista može pristupiti nekom udaljenom web serveru. Budući da nema nikakvih potreba za slanjem parametara ili prijemom podataka, upotreba eksterne biblioteke ne bi ni malo olakšala posao. Ostvarivanje veze i razmjena podataka preko interneta spada u one radnje koje opterećuju glavnu

```
...

new AsyncTask<Void, Void, Void>(){

    @Override
    protected Void doInBackground(Void... params) {

        try {

            HttpURLConnection testConnection = (HttpURLConnection) (new
                                        URL("http://www.google.com/")).openConnection();
            testConnection.setRequestProperty("Connection", "close");
            testConnection.setConnectTimeout(2500);

            if (testConnection.getResponseCode() == 200)
                connectionStatus = true;
            else
                connectionStatus = false;

            connectionStatusCheckComplete = true;

        } catch (IOException e) {
            connectionStatus = false;
            connectionStatusCheckComplete = true;
        }

        return null;

    }

}.execute();

...
```

nit, te je stoga praksa da se ostvarivanje i raskidanje konekcije, slanje i preuzimanje podataka i slično, obavlja na pozadinskoj niti. Za tu svrhu, ovdje je upotrijebljen *AsyncTask*. Prije svega, potrebno je napraviti instancu klase *HttpURLConnection*. Ovdje je to urađeno tako što se prvo instancira klasa *URL* (čiji konstruktor kao argument prima objekat tipa *String*, koji predstavlja URL sadržaja kom želimo da pristupimo), pa se zatim nad tim objektom pozove metoda *openConnection()*, čiji će rezultat, kastovan u *HttpURLConnection*, predstavljati objekat ove klase. Zatim se mogu navesti neke osobine konekcije: perzistivnost, maksimalno vrijeme uspostavljanja i sl. U skladu sa HTTP protokolom, na svaku poruku zahtjeva, dobija se poruka odgovora, koja u statusnoj liniji sadrži statusni kôd i poruku. Ovaj kôd iz HTTP poruke odgovora, pod pretpostavkom da je uspješno uspostavljena konekcija sa serverom se može dobiti pozivom metode *getResponseCode()* nad objektom HTTP konekcije. Ukoliko je statusni kôd jednak broju 200 (što je kôd za uspješno ostvaren zahtjev), statička promjenljiva *connectionStatus* se postavlja na vrijednost *true*, čime se signalizira da je internet veza zaista dostupna. U suprotnom *connectionStatus* će uzeti vrijednost *false*. Na taj način, prije svakog slanja podataka provjerom vrijednosti ove promjenljive se može dobiti podatak o dostupnosti veze. Po završetku provjere ažurira se vrijednost još jedne statičke promjenljive koja se prije započinjanja provjere postavlja na *false*. Nakon izvršene provjere njena se vrijednost stavlja na *true*, čime se signalizira da je provjera uspješno izvršena i da je podatak koji možemo dobiti očitavanjem vrijednosti promjenljive *connectionStatus* zaista validan. Metoda *getResponseCode()* baca izuzetak tipa *IOException* u situaciji kada veza sa serverom prethodno nije uspostavljena, te je stoga ovaj kôd uokviren u *try-catch* blok kako bi se u tom slučaju vrijednost *connectionStatus* promjenljive stavila na *false*. Upotreba *try-catch* bloka je obavezna i iz razloga što konstruktor klase *URL* može baciti izuzetak tipa *MalformedURLException* koji spada u provjerene izuzetke. Treba napomenuti i da

HttpURLConnection ne koristi SSL (što će reći da podaci koji se šalju na ovaj način nijesu kriptovani). Međutim, pošto se ovdje radi čisto o testnom zahtjevu koji ne sadrži nikakve parametre zahtjeva, a kamoli kredencijale, upotreba ovog bezbjedonosnog mehanizma bi bila suvišna.

Iako bi se i u slučaju kada dolazi do razmjene podataka preko HTTP-a (slanje parametara zahtjeva, primanje podataka iz baze i ostalo) cjelokupan proces mogao isprogramirati bazirajući se na *HttpURLConnection* klasi, ovdje je ipak odlučeno da se u tu svrhu primijeni eksterna biblioteka: *Volley*. Razlozi ovakve odluke su višestruki. Prije svega, nije potrebno voditi računa o nitima: ova biblioteka sama vrši prebacivanje odgovarajućih operacija na pozadinsku nit, vodi računa o njihovom upravljanju (upravljanje pomoću izvršioca), i ograničava operaciju razmjene podataka na maksimalno 4 niti (dakle, maksimalno izvršavanje četiri zahtjeva istovremeno), čime onemogućava pretjerano opterećenje procesora. Zatim, ne moramo voditi računa o svakom koraku u prijemu i slanju podataka: ne moramo

```
public void imageDataInsert(final String imageName){

    requestQueue = Volley.newRequestQueue(applicationContext);

    StringRequest request = new StringRequest(Request.Method.POST, imageDataInsertUrl,

            new Response.Listener<String>() {
                @Override
                public void onResponse(String response) {

                }
            },

            new Response.ErrorListener() {
                @Override
                public void onErrorResponse(VolleyError error) {

                }
            }){

        @Override
        protected Map<String, String> getParams() throws AuthFailureError {

            Map<String, String> parameters = new HashMap<String, String>();

            parameters.put("imageName", imageName);
            parameters.put("imagePath", imagePath + imageName);

            return parameters;

        }


    };

    requestQueue.add(request);

}
```

implementirati mehanizam za baferovanje i iščitavanje stream-ova, već podatke primamo i šaljemo prostim pozivanjem adekvatnih metoda, dok o svemu ostalom biblioteka sama vodi računa u pozadini.

Volley biblioteka upravlja zahtjevima pomoću tzv. reda zahtjeva (eng. request queue). Ovo je objekat tipa *RequestQueue* koji organizuje HTTP zahtjeve (ovdje predstavljeni kao objekat nekog od četiri tipa: *JsonObjectRequest*, *JsonArrayRequest*, *StringRequest* ili *ImageRequest*) po principu „First In First Out“. Dakle, zahtjev koji se prvi proslijedi redu zahtjeva biće i prvi obrađen. Zahtjev se formira instanciranjem objekta zahtjeva čiji konstruktor prima četiri argumenta:

1. tip zahtjeva: prosleđuje se kao statička promjenljiva kojom se bira jedan od pet tipova zahtjeva – get, post, put, head ili delete; statička promjenljiva se referencira kao *Request.Method.POST* gdje treći dio u nazivu odgovara odabranom metodu;
2. URL na koji se šalje zahtjev: prosleđuje se kao *String* objekat;
3. osluškivač za prijem odgovora: implementira generički interfejs *Response.Listener<T>*;
4. osluškivač za grešku u odgovoru: implementira interfejs *Response.ErrorListener*;

Instanciranje obuhvata i preklapanje metode *getParams()* koja kao rezultat vraća objekat tipa *Map<String, String>* - objekat koji se sastoji od niza uparenih stringova koji predstavljaju parametre POST zahtjeva. Stoga se prvo instancira objekat ovog tipa, a zatim metodom *put(...)* smještamo u njega parove stringova. Prvi string u paru je naziv parametra, dok je drugi njegova vrijednost. Konačno, ovako definisan zahtjev se dodaje u red zahtjeva metodom *add(...)* koja se poziva nad objektom reda zahtjeva i koja kao argument prima formirani zahtjev.

U razmatranom primjeru zahtjev je obuhvatao isključivo slanje parametara, ali ne i prijem odgovora. Tada je dovoljno samo preklopiti metodu *getParams()* bez potrebe ispisivanja bilo kakvog kôda u tijelima callback metoda osluškivača. Takav zahtjev je u VECTOR INSPECT aplikaciji korišćen za upisivanje podataka u bazu. Podaci se šalju kao parametri POST zahtjeva koji su unutar *StringRequest* metode definisani u vidu mape stringova, gdje prvi string predstavlja naziv kolona, a drugi sadržaj za upis u kolonu. O samoj strukturi baze biće više riječi u narednom poglavlju. Međutim, sem ovakvog tipa zahtjeva postojala je potreba i za iščitavanjem podataka iz baze. Format koji je najpodesniji za iščitavanje podataka iz baze je JSON format, koji predstavlja vrlo pregledan način prikaza podataka koji se sastoje od elemenata koje sačinjavaju parovi naziv – vrijednost. Stoga je za ovu svrhu korišćen *JsonObjectRequest* zahtjev umjesto *StringRequest*. Budući da je upit za čitanje podataka iz odgovarajuće tabele formiran unutar *.php* skripte koja je smještena na serveru, nije bilo potrebno slanje nikakvih POST parametara, te stoga nije bilo potrebe preklapati metodu *getParams()*. Dovoljno je samo bilo definisati tijelo callback metode *Response.Listener<JSONObject>* osluškivača, koji sada kao generički parametar prima *JSONObject* umjesto *String*.

```
new Response.Listener<JSONObject>() {

    @Override
    public void onResponse(JSONObject response) {

        try {
            JSONArray speciesJsonArray = response.getJSONArray("species");

            for (int i = 0; i < speciesJsonArray.length(); i++){
                JSONObject speciesJsonObject = speciesJsonArray.getJSONObject(i);

                //Obrada JSON objekta

            }

            status.append("DONE");

        } catch (JSONException e) {
            e.printStackTrace();
        }

    }

}
```

Primljeni JSON objekat, u ovom primjeru, sastoji se od niza podobjekata takođe JSON tipa koji se uzima metodom *getJSONArray(...)* i smješta u promjenljivu tipa *JSONArray*. Struktura primljenog JSON objekta je sledeća: svaki red iz odgovarajuće tabele formira pojedinačni JSON objekat (parovi vrijednosti naziv kolone – sadržaj kolone), a zatim svi objekti formiraju JSON niz koji se smješta u novi JSON objekat koji

predstavlja čitavu tabelu iz koje se podaci iščitavaju. Taj objekat se šalje kao odgovor na zahtjev, i prije bilo kakve obrade, prvo se iz njega „izvlači“ niz podobjekata metodom *getJSONArray(...)* koja kao argument prima naziv toga niza proslijeđen u vidu stringa. Zatim se iz niza uzima svaki pojedinačni objekat nad kojim se vrši obrada koja, između ostalog, obuhvata smještanje očitanih podataka u odgovarajuće kolekcije na način koji je pogodan za dalju manipulaciju. Po završetku obrade cjelokupnog zahtjeva, u statičku promjenljivu *status* se upisuje odgovarajuća poruka čime se signalizira da je učitavanje podataka završeno.

Na ovaj način se u Android programiranju ostvaruje slanje poruka putem HTTP protokola. Međutim, za slanje fajlova, kao što su slike, pogodnije je koristiti FTP protokol. Za realizaciju ovakvog načina slanja fajlova neophodno je koristiti eksterne biblioteke budući da ne postoje ugrađene Javine biblioteke i klase kojima se ovaj protokol može direktno realizovati. Naravno, moguće je pomoću osnovnih elemenata kreirati sopstvenu klasu koja će izvršavati ovu djelatnost, ali nema potrebe trošiti vrijeme na tako nešto. Stoga je u tu svrhu iskorišćena vrlo jednostavna *SimpleFTP* biblioteka. Ta biblioteka ipak nije uvezena na klasičan način, već je za potrebe ove aplikacije njen source kôd prekopiran unutar posebne klase. Razlog za to je bila potreba da se metoda za otvaranje konekcije prepravi dodavanjem određenih

```
...

new AsyncTask<Void, String, Void>(){

    @Override
    protected Void doInBackground(Void... params) {

         try {

             SimpleFTP ftpConnection = new SimpleFTP();

             publishProgress("BEGIN");

             ftpConnection.connect(server, portNum, username, password); //connect() metod
                                                                          baca izuzetak
             ftpConnection.bin();
             ftpConnection.cwd(directory);

             publishProgress("CONNECTED_SUCCESSFULLY");

             int successNumber = 0;

             for (int i=0; i<total; i++ ) {

                 ByteArrayInputStream inputStream = new ByteArrayInputStream(images.get(i));
                 boolean storage = ftpConnection.stor(inputStream, imageNames.get(i) +
                                                                          imageFormat);

                 inputStream.close();

             }

             if (storage)
                 successNumber++;

             ftpConnection.disconnect();

             if (successNumber == total)
                 publishProgress("ALL_SUCCESS");
             else if (successNumber > 0)
                 publishProgress("PARTIAL_SUCCESS");
             else
                 publishProgress("ALL_FAIL");

             ...
```

```
                ...

            } catch (Exception e){
                 publishProgress("ALL_FAIL");
            }

                return null;

            }

        @Override
        protected void onProgressUpdate(String... values) {

             //Modifikovanje korisničkog interfejsa

        }

}.execute();

...
```

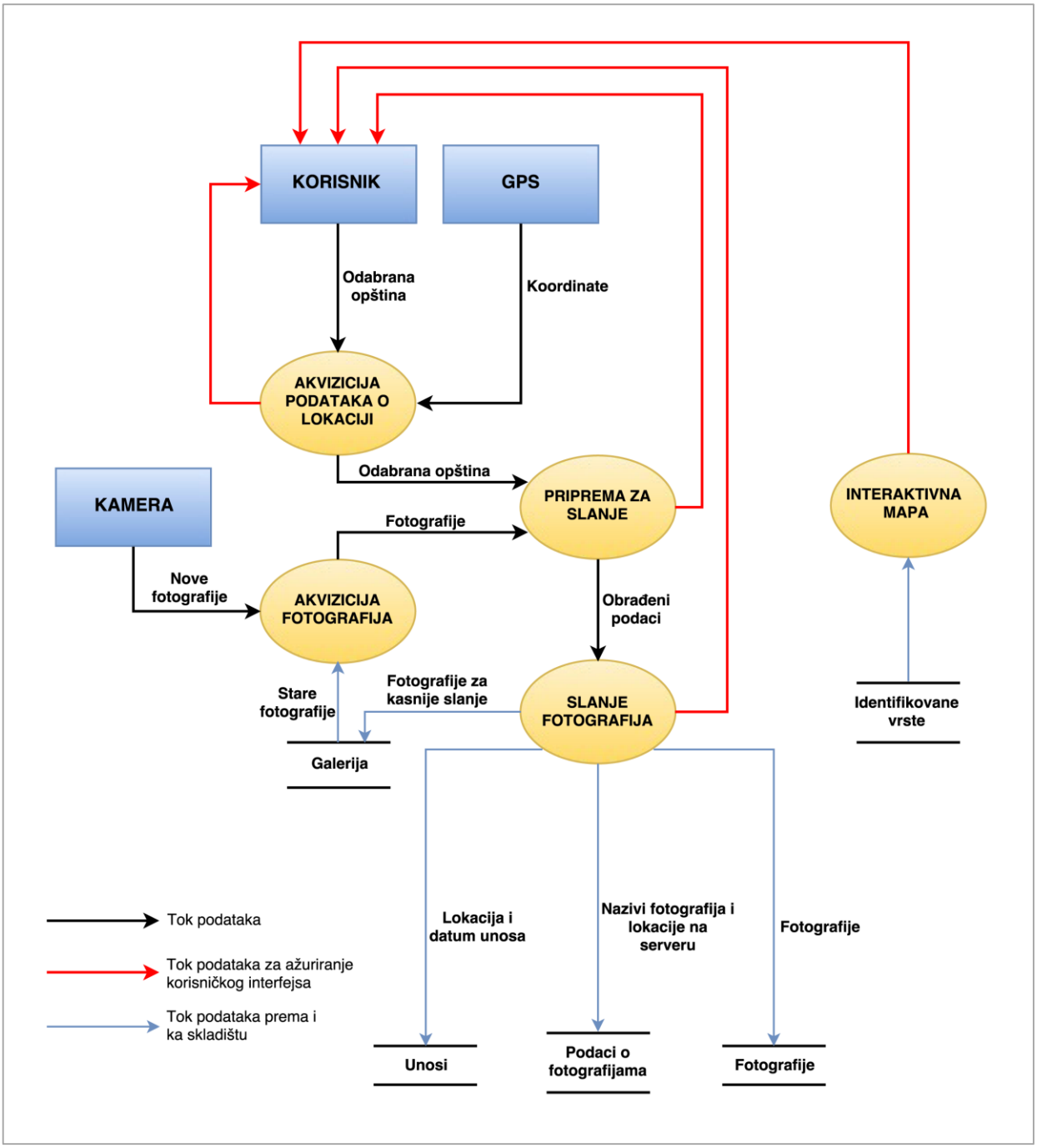
linija kôda, kako bi aplikacija uspješno funkcionisala i sa FileZilla FTP serverom, koji je korišćen za testiranje radne verzije.

Ovo je primjer koji sažima način realizacije slanja slika u VECTOR INSPECT aplikaciji, koji se može generalno primijeniti u bilo kojoj drugoj aplikaciji. *SimpleFTP* klasa se, za razliku od *Volley* klase, ne bavi u potpunosti upravljanjem nitima (sem što se postarala da određene unutrašnje metode učini nitno bezbjednima), tako da je potrebno da se sam programer postara da se otvaranje konekcije i slanje slike obavlja na pozadinskoj niti. Stoga se i ovdje upotrebljava *AsyncTask*. Rad sa FTP protokolom započinje se instanciranjem klase *SimpleFTP* i čuvanjem reference na tu instancu u promjenljivu. Zatim se otvara FTP konekcija pozivajući nad tom promjenljivom metodu *connect(...)* koja kao argumente prima adresu servera, broj porta za ostvarivanje FTP konekcije (ovdje je korišten standardni broj FTP porta – 21), korisničko ime, i šifru. Metoda *bin()* definiše slanje fajla u binarnom modu, a *cwd(...)* definiše direktorijum u kom će se fajl čuvati. Budući da se fajlovi šalju u binarnom modu (što je preporučeno od strane autora biblioteke), sliku je prije slanja potrebno pretvoriti u niz bajtova, a onda taj niz u stream bajtova. U ovom primjeru sve slike su već konvertovane u niz bajtova i ti nizovi su smješteni u kolekciju *images*, odakle se unutar *for* petlje uzima jedna po jedna slika i konvertuje u tok bajtova pozivajući konstruktor *ByteArrayInputStream(...)* i čuvajući referencu na taj tok. Tako formiran objekat koji predstavlja stream bajtova se zajedno sa stringom koji predstavlja naziv fotografije prosleđuje kao argument metodi *stor(...)* koja započinje slanje bajtova i njihovu adekvatnu konverziju i skladištenje na serveru. Ova metoda vraća logički rezultat, tako da je moguće pratiti uspješnost slanja i čuvanja svake pojedinačne fotografije, što je ovdje i učinjeno inkrementiranjem promjenljive *successNumber* nakon svakog uspješnog slanja. Po završetku slanja svakog pojedinačnog fajla, stream se zatvara metodom *close()*, dok se po završetku slanja svih fajlova, zatvara se i FTP konekcija metodom *disconnect()*. Metodom *publishProgress(...)* šalju se izvještaji nakon svake etape slanja, na osnovu kojih se može projektovati sistem koji će korisnika aplikacije na korisničkom intefejsu obavještavati o napredovanju procesa slanja.

4. VECTOR INSPECT APLIKACIJA

4.1 Aplikacija kao informacioni sistem

VECTOR INSPECT aplikacija se zbog svoje strukture i namjene može tretirati kao informacioni sistem, te se njena analiza može obaviti i sa stanovišta strukturne sistemske analize. Prema tome, aplikacija ima precizno definisane interfejse (spoljašnje objekte), ulazne i izlazne tokove podataka, procese i skladišta podataka. Ovdje treba napomenuti da termini „interfejs“ i „proces“ imaju drugačije značenje u kontekstu informacionog sistema u odnosu na programerski kontekst. Čitava aplikacija se, dakle, može predstaviti dijagramom toka podataka:



Slika 6. Dijagram toka podataka aplikacije

Dakle, glavni interfejs je korisnik koji inicira čitav proces slanja i upravlja čitavim informacionim sistemom, i kom se u svakom trenutku šalju povratne informacije u vidu izmjena na grafičkom korisničkom interfejsu. Ostali izvori podataka su i hardverske komponente uređaja, tačnije GPS i kamera. Proces slanja podataka se odvija kao lančano izvršavanje nekoliko procesa. Prvi u nizu je proces akvizicije podataka o lokaciji koji obuhvata određivanje lokacije bilo ručnim odabirom (od strane korisnika) ili automatski na osnovu dobijenih GPS koordinata. Nakon uspješnog određivanja lokacije, započinje se proces akvizicije fotografija, koje opet mogu poticati iz dva izvora: kamere, koja u ovom slučaju predstavlja interfejs od kog informacioni sistem dobija nove fotografije, ili galerije, koja predstavlja lokalno skladište podataka, i iz koje informacioni sistem može dobiti neke od starijih fotografija. Odabir prve fotografije inicira proces pripreme za slanje kom se kao podaci šalju odabrana lokacija i uzeta fotografija. Zatim, ukoliko korisnik želi poslati još fotografija, može se ponoviti proces akvizicije fotografije sa ciljem dodavanja još fotografija u pripremu za slanje. Nakon što korisnik doda sve željene fotografije, vrši se adekvatna obrada podataka u format pogodan za slanje na server, nakon čega se cjelokupni podaci šalju na udaljena skladišta podataka, u sklopu odgovarajućeg procesa. Ta udaljena skladišta su MySQL baza podataka i FTP server. U sklopu baze podataka popunjavaju se dvije tabele: jedna koja sadrži podatke o svakom pojedinačnom unosu (lokacija, koordinate, datum i vrijeme), i druga koja sadrži podatke o svim fotografijama koje su vezane za neki unos (naziv i lokacija na FTP serveru). Same fotografije se smještaju u odgovarajućem direktorijumu na FTP serveru. Paralelno sa ova četiri procesa funkcioniše i peti proces koji je potpuno odvojen od njih. To je proces koji iz baze podataka (udaljeno skladište podataka) uzima podatke o identifikovanim vrstama insekata. Ovi podaci su sadržani u posebnoj tabeli koja se popunjava na osnovu analize pristiglih fotografija i lokacija. Tabela od podataka sadrži naziv vrste i ID unosa na osnovu kog je identifikovana, a formiranjem složenog upita može se formirati i statistika o procentu javljanja svake vrste. Ti podaci se šalju procesu koji na osnovu korisničkog odabira preko interaktivne mape generiše odgovarajući prikaz podataka na korisničkom interfejsu.

4.2 Struktura baze podataka

Za potrebe skladištenja podataka o pristiglim fotografijama, aplikacija je povezana na MySQL bazu koja je smještena na udaljenom serveru. Struktura baze nije previše složena. Sastoji se od dvije tabele, *Entries* i *Photos*, u koje se smještaju podaci o unosu i fotografijama i treće *Species* u kojoj se smještaju podaci o vrstama identifikovanim na osnovu korisničkih unosa. Prve dvije tabele popunjava korisnik slanjem fotografija putem VECTOR INSPECT aplikacije, dok se treća tabela popunjava od strane eksperata koji vrše identifikaciju vrsta na osnovu pristiglih fotografija.

ENTRY_ID	municipality	coordinatePhi	coordinateLambda	date	time
213	DANILOVGRAD	42,5609 N	19,1153 E	08.08.2016	06:22:56
214	CETINJE	42,4014 N	18,8593 E	08.08.2016	06:24:03
215	DANILOVGRAD			08.08.2016	06:25:07
216	BUDVA			08.08.2016	06:27:23
217	PLJEVLJA			08.08.2016	06:28:03
218	BAR	42,0796 N	19,1023 E	08.08.2016	06:29:13
219	BERANE	42,9058 N	20,0105 E	08.08.2016	06:29:58
220	ULCINJ	41,9737 N	19,2947 E	08.08.2016	06:31:37
221	ULCINJ	41,9737 N	19,2947 E	08.08.2016	06:34:58
222	DANILOVGRAD	42,5604 N	19,1144 E	08.08.2016	06:36:56
223	DANILOVGRAD	42,5604 N	19,1144 E	08.08.2016	06:37:18
224	DANILOVGRAD	42,5604 N	19,1144 E	08.08.2016	06:48:45
225	DANILOVGRAD	42,5542 N	19,1108 E	08.08.2016	08:10:31
227	DANILOVGRAD	42,5587 N	19,1232 E	08.08.2016	08:26:43
228	DANILOVGRAD	42,5587 N	19,1232 E	08.08.2016	08:28:02
229	DANILOVGRAD	42,5587 N	19,1232 E	08.08.2016	08:32:05
230	DANILOVGRAD	42.5518 N	19.1150 E	08.08.2016	08:33:46

SPECIES_ID	speciesName	entry_id
1	Culex pipiens	213
2	Culex pipiens	214
3	Aedes albopictus	215
4	Culex pipiens	216
5	Culex pipiens	217
6	Aedes albopictus	218
7	Aedes albopictus	219
8	Aedes aegypti	220
9	Aedes vexanus	221
10	Culex pipiens	222
12	Aedes albopictus	223
13	Aedes aegypti	224
14	Culex pipiens	225

Slika 7. Tabele „Entries“ i „Species“

Glavna tabela za koju su ostale vezane stranim ključevima je tabela *Entries* (slika 7). Kao što sâm naziv sugeriše, ta tabela sadrži osnovne podatke o svakom unosu:

1. **naziv opštine**: ovaj podatak je sadržan u koloni *municipality* koja je tipa *VARCHAR* i sadrži podatak o nazivu opštine sa koje su fotografije poslate;
2. **koordinate**: podatak je sadržan u kolonama *coordinatePhi* i *coordinateLambda* (obje tipa *VARCHAR)* koje označavaju geografsku širinu i dužinu, respektivno, izražene u decimalnom obliku, i odnosi se na koordinate tačke sa koje su fotografije poslate. Na osnovu tih koordinata aplikacija može sama da odredi kojoj opštini ta tačka pripada. Shodno tome, ukoliko su slike poslate nakon automatskog odabira lokacije, svaki unos će sadržati i naziv opštine i koordinate, dok će u slučaju da je lokacija odabrana ručno, unos u tabelu sadržati samo naziv opštine;
3. **datum i vrijeme**: aplikacija prilikom slanja fotografija ujedno šalje i sistemsko vrijeme uređaja na osnovu kog se popunjavaju kolone *date* i *time* (takođe tipa *VARCHAR)* koje sadrže podatke o datumu i tačnom vremenu slanja fotografija;

Radi bolje organizacije redova, kao i radi mogućnosti povezivanja sa ostalim tabelama, tabela *Entries* sadrži i posebnu kolonu *ENTRY_ID* (tipa *INTEGER*), koja sadrži jedinstveni identifikacioni broj svakog unosa, a samim tim i vrste tabele, te stoga obavlja funkciju njenog primarnog ključa. Vrijednost ovog broja se generiše automatskim inkrementiranjem.

PHOTO_ID	photoName	photoPath	entry_id
1	2016080806225601.png	localhost/images/2016080806225601.png	213
2	2016080806225602.png	localhost/images/2016080806225602.png	213
3	2016080806225603.png	localhost/images/2016080806225603.png	213
4	2016080806240301.png	localhost/images/2016080806240301.png	214
5	2016080806240302.png	localhost/images/2016080806240302.png	214
6	2016080806250701.png	localhost/images/2016080806250701.png	215
7	2016080806250702.png	localhost/images/2016080806250702.png	215
8	2016080806272301.png	localhost/images/2016080806272301.png	216
9	2016080806272302.png	localhost/images/2016080806272302.png	216
10	2016080806272303.png	localhost/images/2016080806272303.png	216
11	2016080806272308.png	localhost/images/2016080806272308.png	216
12	2016080806280301.png	localhost/images/2016080806280301.png	217
13	2016080806291301.png	localhost/images/2016080806291301.png	218
14	2016080806291302.png	localhost/images/2016080806291302.png	218
15	2016080806291309.png	localhost/images/2016080806291309.png	218
16	2016080806295801.png	localhost/images/2016080806295801.png	219
17	2016080806313701.png	localhost/images/2016080806313701.png	220
18	2016080806313702.png	localhost/images/2016080806313702.png	220
19	2016080806313703.png	localhost/images/2016080806313703.png	220
20	2016080806313706.png	localhost/images/2016080806313706.png	220
21	2016080806345801.png	localhost/images/2016080806345801.png	221
22	2016080806345802.png	localhost/images/2016080806345802.png	221

Slika 8. Tabela „Photos“

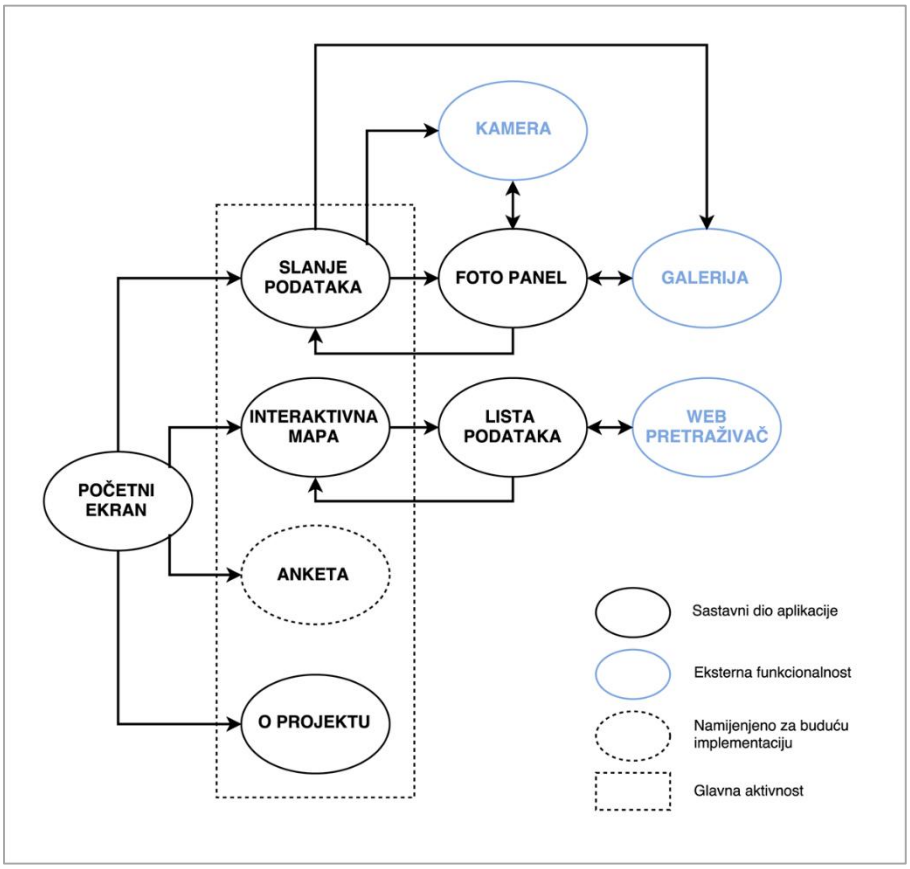
Pristigle fotografije se, kao što je ranije navedeno, čuvaju u odgovarajućem direktorijumu na FTP serveru, jer bi direktno skladištenje u bazu podataka bilo izuzetno nepraktično. Međutim, radi lakšeg praćenja svih pristiglih fotografija, kreirana je tabela *Photos* (slika 8) koja vodi računa o tome gdje je pristigla fotografija sačuvana (što može biti vrlo bitan podatak u slučaju da u međuvremenu dođe do promjene direktorijuma na FTP serveru u kom se čuvaju fotografije) kao i za koji unos se ta fotografija vezuje. Ovi podaci su sadržani unutar dvije kolone, *photoName* i *photoPath*, obje tipa *VARCHAR*. Kao i u tabeli *Entries* i ovdje imamo kolonu koja sadrži jedinstvene identifikacione brojeve svake fotografije. To je kolona sa nazivom *PHOTO_ID*, naravno, tipa *INTEGER*. I njene vrijednosti se takođe generišu sistemom automatske inkrementacije. Drugim riječima, ova kolona služi kao primarni ključ tabele *Photos*. Sem ovih, tabela *Photos* sadrži i kolonu nazvanu *entry_id* koja ukazuje na unos u sklopu kog je data fotografija pristigla. Logično, ova kolona mora biti istog tipa kao i kolona *ENTRY_ID* tabele *Entries* i njene vrijednosti moraju biti iz istog skupa. Dakle, ova kolona ima ulogu stranog ključa tabele *Photos* prema tabeli *Entries*. Primjera radi, prve tri fotografije se vezuju za unos sa ID-jem 213, te ako pogledamo tabelu *Entries* (slika 7), vidjećemo da su te fotografije slikane na

teritoriji opštine Danilovgrad i poslate 8. avgusta 2016. godine u 6 sati, 22 minuta i 56 sekundi. Ono što još treba primijetiti jeste da se nazivi fotografija dodjeljuju po definisanom šablonu:

YYYYMMDDHHMMSSNN

gdje YYYY označava godinu slanja, MM mjesec, DD dan, HH sat, MM minut, SS sekund, a NN redni broj. Na taj način je obezbijeđeno da sve pristigle fotografije imaju različito ime, što je neophodno kako nova fotografija ne bi bila upisana na mjesto neke ranije pristigle sa istim imenom. Tako, na primjer, prve tri fotografije (koje smo maloprije povezali sa odgovarajućim unosom) imaće isti prefiks (*20160808062256*), dok će se međusobno razlikovati po rednom broju (*01*, *02* i *03*).

Na osnovu podataka iz ove dvije tabele, eksperti su u mogućnosti da pregledaju sve unose, vide koje su to fotografije pristigle u sklopu njega, te da identifikuju vrstu insekta koja se na njima nalazi. Nakon uspješno obavljene identifikacije dovoljno je unijeti naziv identifikovane vrste u tabelu *Species* (direktno ili preko neke druge računarske, web ili mobilne aplikacije) kao i unos na osnovu kog je identifikacija obavljena. Stoga tabela *Species* (slika 7) sadrži kolonu sa nazivom vrste (*speciesName*, tipa *VARCHAR*), kao i kolonu sa ID brojem odgovarajućeg unosa (*entry_id*, koji je i u ovom slučaju strani ključ prema tabeli *Entries*). Tabela takođe sadrži i sopstveni primarni ključ, što je u ovom slučaju kolona *SPECIES_ID* koja je, kao što je to slučaj i sa prethodnim primarnim ključevima, tipa *INTEGER* i dobija vrijednosti autoinkrementacijom. Nadovezujući se na prethodni primjer, sada vidimo da se na prve tri fotografije iz tabele *Photos*, koje su fotografisane na teritoriji opštine Danilovgrad u navedeno vrijeme, nalazi vrsta insekta identifikovana kao „Culex Pipiens“. Sem ovih osnovnih podataka, ovako struktuirana baza ostavlja mogućnost dobijanja i nekih složenijih podataka na osnovu dobro napisanih složenih upita. Način na koji je to izvedeno u VECTOR INSPECT aplikaciji biće demonstriran u nastavku rada.



Slika 9. Prikaz strukture aplikacije u vidu grafa. Čvorovi grupisani unutar glavne aktivnosti mogu pozivati jedan drugog u svakom trenutku

4.2 Struktura aplikacije

Kao što je već navedeno, VECTOR INSPECT aplikacija se sastoji od niza tijesno povezanih funkcionalnosti. Stoga ćemo, radi boljeg uvida, cjelokupnu strukturu aplikacije predstaviti pomoću grafa, gdje će svaki čvor odgovarati jednom prikazu na korisničkom interfejsu, koji se mogu grupisati na osnovu funkcionalnosti kojima pripadaju. Zatim će biti predočen detaljan opis svake funkcionalnosti, sa principima njihove realizacije.

4.2.1 Početni ekran

Slika 10. Početni ekran

Prilikom startovanja aplikacije na ekranu uređaja se prikazuje animirani logo Lovćen projekta (slika 10). Naravno, ovaj čvor na grafu se mogao i zaobići, budući da ova aktivnost ne obavlja nikakvu konkretnu funkcionalnost, ali on postoji prvenstveno radi estetike aplikacije. Prateći trendove na osnovu kojih se dosta pažnje posvećuje dizajnu, dodat je tzv. „splash screen“ koji služi kao uvod u aplikaciju, pritom sadržeći vrlo jednostavan ali efektan dizajn, bogat animacijom. Na ovaj način se teži ka stvaranju „eye-candy“ dizajna grafičkog interfejsa koji na korisnike ostavlja vrlo povoljan utisak.

Realizacija ove aktivnosti je prilično jednostavna. Prije svega, u manifest fajlu je bilo potrebno naznačiti da je ovo pokretačka aktivnost dodavanjem tzv. intent filtra unutar elementa *.xml* fajla kojim se definiše ova aktivnost:

```
<intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
```

Zatim je realizovana animacija u XML-u, koja obuhvata animiranje svakog slova pojedinačno. Ovdje je dat način na koji je odrađena animacija za prvo slovo. Kao što vidimo, to je složena animacija definisana pomoću seta koji sadrži dvije translacije. Prva postavlja komponentu za 30% ulijevo (koordinatni sistem je sada relativan u odnosu na čitav ekran, pri čemu se za koordinatni početak uzima unaprijed definisana pozicija komponente), a zatim vrši translaciju udesno za 40% (30% kojim se vraća u početni položaj, tj. koordinatni početak, i još 10%) koristeći accelerate inerpolator. Trajanje ove translacije je 150 milisekundi. Odmah po završetku prve animacije sekvencijalno se izvršava naredna koja sada komponentu pomjera za 10% ulijevo koristeći decelerate interpolator, za isto trajanje. Ovdje je važno napomenuti da se koordinatni početak pomjera nakon svake izvršene translacije definisane unutar jednog seta. Budući da je nakon prve položaj komponente ukupno bio pomjeren za 10% udesno, to će i koordinatni početak za drugu animaciju biti pomjeren za istu vrijednost. Samim tim, budući da druga animacija pomjera komponentu za 10% ulijevo, ta komponenta se vraća na svoj definisani položaj, na kom je bila prije započinjanja animacije. Ovim je postignut svojevrsni „overshoot“ efekat. Svako naredno slovo sadrži istu strukturu animacije, sa drugačijim vrijednostima pojedinih parametara, tačnije početne vrijednosti *x* koordinate u prvoj translaciji (budući da je definisani početni položaj drugačiji, pa je samim tim komponentu potrebno više pomjeriti na samom startu) i *startOffset* koji se podešava tako da animacije svakog slova kreću jedna za drugom bez preklapanja.

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android">

    <translate
        android:fromXDelta="-30%p"
        android:toXDelta="10%p"
        android:interpolator="@android:anim/accelerate_interpolator"
        android:duration="150"
        />

    <translate
        android:fromXDelta="0%p"
        android:toXDelta="-10%p"
        android:interpolator="@android:anim/decelerate_interpolator"
        android:startOffset="150"
        android:duration="150" />

</set>
```

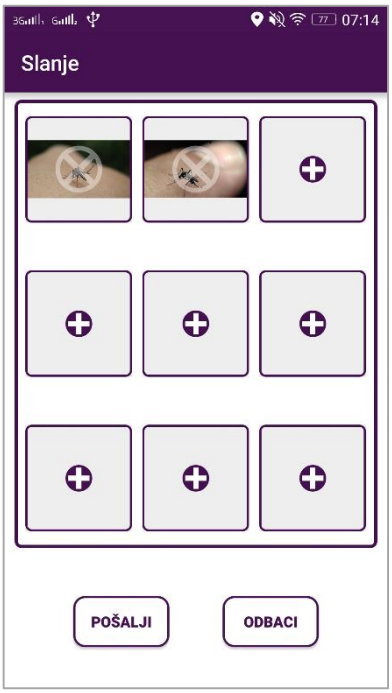
Što se Java kôd-a tiče, početni ekran se realizuje pomoću jedne klase - klase koja definiše tu aktivnost. Prije svega se vidljivost svake komponente stavlja na *INVISIBLE* kako bi se izbjeglo momentalno prikazivanje na ekranu, a zatim se instancira 6 objekata tipa *Animation* (po jedan za svako slovo) vezuvajući za njih odgovarajuće *.xml* fajlove kojima su te animacije definisane nakon čega se one startuju. Za objekat koji reprezentuje animaciju poslednjeg slova se registruje osluškivač animacije. Preklopljene su callback metode *onAnimationStart(...)*, kako bi se vidljivost komponenti postavila na *VISIBLE* prilikom startovanja animacije, i metoda *onAnimationEnd(...)* kako bi se po završetku animacije pozvala naredna aktivnost, a raspustila trenutna. Iako prividno animacija poslednjeg slova započinje poslednja, ona zapravo počinje istovremeno sa svim ostalima jer *startOffset* samo odlaže započinjanje promjene definisanih parametara ali ne i stvarno startovanje animacije. Stoga se osluškivač koji bi reagovao na započinjanje animacije mogao vezati za bilo koju od njih. Međutim, budući da se osluškivač za završetak animacije može vezati jedino za animaciju poslednjeg slova (koja se završava poslednja), onda je taj osluškivač iskorišćen i za svrhe osluškivanja početka animacije.

4.2.2 Slanje fotografija

Slika 11. Interfejs za slanje fotografija

Konačno dolazimo do ključne funkcionalnosti VECTOR INSPECT aplikacije. Odmah nakon završetka animacije početnog ekrana dolazi do prebacivanja sa početne aktivnosti na centralnu aktivnost. Ta aktivnost sadrži četiri fragmenta. Funkcionalnost slanja podataka započinje se unutar prvog, koji je ujedno i podrazumijevani fragmenat koji se „kači“ za aktivnost prilikom svakog njenog pokretanja.

Korisnički interfejs se sastoji iz dva panela (slika 11). Prvi je panel za prikaz koordinata i opštine u kojoj se korisnik nalazi, koji takođe sadrži i radio dugmad za odabir režima određivanja lokacije. Lokacija se može određivati na dva načina: automatski, aktiviranjem GPS-a, dobijanjem koordinata lokacije na kojoj se uređaj nalazi i automatskim određivanjem opštine kojoj ta koordinata pripada, i ručni, koji podrazumijeva ručno biranje opština iz skupa ponuđenih bez potrebe vršenja akvizicije koordinata. Drugi panel sadrži dva dugmeta. Klikom na lijevo dugme poziva se podrazumijevana aplikacija za fotografisanje kojom se vrši akvizicija nove fotografije, dok se klikom na desno dugme otvara galerija iz koje korisnik takođe može da odabere fotografiju za slanje. U oba slučaja fotografija se šalje na panel za slanje koji je realizovan unutar druge aktivnosti, što znači da se nakon obavljenog fotografisanja ili odabira, fotografija smješta u bundle kako bi se prenijela u drugu aktivnost, koja se zatim startuje. Trenutna aktivnost se ne raspušta, već prelazi u *Stop* stanje, te se po završetku cjelokupnog procesa stanja ista vraća u aktivno stanje.



Slika 12. Panel za slanje fotografija

Panel za slanje (slika 12) služi kako bi organizovao fotografije i omogućio višestruko slanje što je vrlo korisna funkcionalnost kada je u pitanju prikupljanje i slanje fotografija za identifikaciju vrsta. Princip funkcionisanja je vrlo jednostavan: prilikom pozivanja kamere ili

galerije iz centralne aktivnosti vrši se odabir ili akvizicija fotografije, nakon čega se poziva panel za slanje kao nova aktivnost, pri čemu se u prvo polje smješta ta fotografija. Slanje dodatnih fotografija se obavlja klikom na + dugme, koje otvara iskačući prozor na kom korisnik može opet odabrati da li sledeću fotografiju želi da uzme iz galerije ili da je sâm fotografiše. Na osnovu odabira, opet se pokreće kamera ili galerija, a dobijena fotografija se smješta u ono polje čije je + dugme bilo kliknuto. Nakon smještanja fotografije u odgovarajuće polje + dugme se zamjenjuje „X“ dugmetom. Klikom na to dugme korisnik može da ukloni fotografiju iz datog polja i da je izuzme iz slanja. Na ovaj način se kreira izuzetno pregledan interfejs za organizaciju slika za slanje koji je veoma lak za upotrebu. Naravno, ne treba posebno spominjati da je cjelokupni proces promjene izgleda interfejsa u skladu sa korisničkim upravljanjem propraćen adekvatnim animacijama (pojavljivanje i nestajanje + i X dugmeta). U dnu panela smještena su dva dugmeta čiji naziv govori o funkciji koju obavljaju: klikom na dugme *Pošalji* započinje se proces slanja, dok klik na dugme *Odbaci* odbacuje sve podatke za slanje i vraća korisnika na centralnu aktivnost, tačnije na početni fragment za odabir lokacije i slanje fotografija.

Ono što se smješta u polja panela je smanjena verzija slike za slanje, takozvani thumbnail. Dakle, nakon odabira slike ili fotografisanja vrše se dva procesa: prvo, kreiranje smanjene bitmape maksimalnih dimenzija 300 x 300 piksela i smještanje iste unutar odgovarajućeg polja panela i drugo, ograničavanje slike na maksimalne dimenzije 2048 x 2048 piksela, kompresovanje u *.png* format, i smještanje u odgovarajuću kolekciju:

```
...

Bitmap tmpBitmap = Bitmap.createScaledBitmap(params[0], width, height, true);

final ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
tmpBitmap.compress(Bitmap.CompressFormat.PNG, 100, outputStream);
byte[] imageBytes = outputStream.toByteArray();

imagesForSending.set(i, imageBytes);
hasImage[i] = true;

...
```

Metoda *createScaledBitmap(...)* svodi bitmapu na nove dimenzije koje se računaju prije poziva ove metode. Dakle, uslov je da najveća dimenzija ne prelazi limit od 2048 piksela (što je uvedeno kao kompromis između kvaliteta slike sa jedne strane i brzine obrade i slanja i minimiziranja memorijskih zahtjeva aplikacije sa druge strane), te se, ukoliko je potrebno, vrši skaliranje ali tako da slika zadrži svoje originalne proporcije. Naravno, ovaj limit se može mijenjati u zavisnosti od potrebe. Budući da rad sa ovolikim bitmapama na pojedinim uređajima zahtjeva veću količinu radne memorije, u manifest fajlu je dodata linija `android:largeHeap="true"` čime se virtuelnoj mašini obezbjeđuje veći memorijski prostor i izbjegava „pucanje“ aplikacije usled njegovog nedostatka. Ukoliko je skaliranje potrebno računaju se nove dimenzije, dok se u suprotnom zadržavaju stare. Ovoj metodi se kao prvi argument prosleđuje originalna bitmapa, a kao drugi i treći predviđene dimenzije. Zatim se bitmapa komprimuje u *.png* format, a bajtovi komprimovane slike se smještaju u unaprijed instancirani izlazni tok podataka. Referenca na te bajtove se čuva u odgovarajućoj promjenljivoj, koja se, konačno, dodaje kao novi element kolekcije koja sadrži nizove bajtova svih slika sa panela za slanje (u ovom primjeru *imagesForSending*). Kolekcija ima devet elemenata, gdje svaki reprezentuje po jedno polje panela. Dakle, slika se tretira kao niz bajtova, šalje se u tom formatu, a na serverskoj strani se primljeni bajtovi smještaju u odgovarajući fajl, čime se slika trajno čuva. Uporedo sa kolekcijom bajtova, ova klasa sadrži i promjenljivu *hasImage* koja sadrži devet logičkih elemenata kojima se signalizira da li na datoj poziciji u kolekciji nizova bajtova stoji validna reprezentacija neke slike. Naime, rečeno je da se klikom na X dugme na postavljenoj fotografiji u nekom od polja ta fotografija briše. To brisanje sem uklanjanja thumbnaila-a ne obuhvata i brisanje elementa iz kolekcije, već samo stavljanje odgovarajućeg elementa u *hasImage* nizu na vrijednost *false*. Prilikom slanja fotografija, prolazi se svaki element kolekcije, te ukoliko na istoj poziciji u *hasImage*-u stoji vrijednost

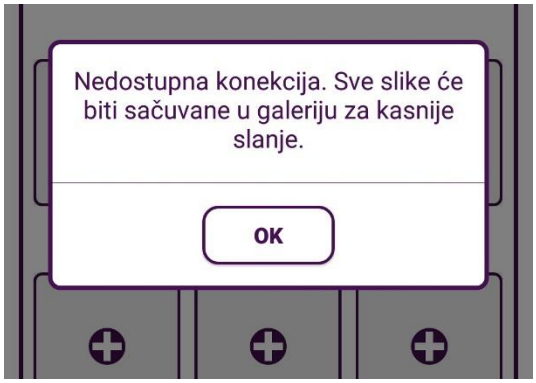
true, tada je niz bajtova validan i on se šalje. Ukoliko, pak, stoji *false*, tada se taj niz preskače budući da je ta fotografija „izbrisana“. Naravno, sve navedene operacije odvijaju se na pozadinskoj niti kako bi glavna nit bila rasterećena.

```
<?php

	if ( $_SERVER["REQUEST_METHOD"] == "POST" ){
		header('Content-Type: text/html; charset=utf-8');
		require 'connection.php';
		makeEntry();
	}

	function makeEntry(){

		global $connect;

		$municipality = $_POST["municipality"];
		$coordinatePhi = $_POST["coordinatePhi"];
		$coordinateLambda = $_POST["coordinateLambda"];
		$date = $_POST["date"];
		$time = $_POST["time"];

		$query =
		  "INSERT INTO entries (municipality, coordinatePhi, coordinateLambda, date, time)
		   VALUES ('$municipality', '$coordinatePhi', '$coordinateLambda', '$date', '$time')";

		$result = mysqli_query($connect, $query);

		if ($result) {
			echo 'success';
		} else {
			echo 'fail';
		}

		mysqli_close($connect);

	}

?>
```

Proces slanja ovako organizovanih fotografija se ne radi unutar klase ove aktivnosti, već je za tu svrhu kreirana posebna klasa, nazvana *DatabaseHandler*, koja, između ostalog, sadrži metode za slanje podataka o slikama, slanje samih slika, brisanje unosa iz baze, kao i metodu za uzimanje podataka iz baze potrebnih za interaktivnu mapu. Prvo se vrši slanje podataka o fotografijama, tj. vrši se unos u glavnu tabelu baze podataka metodom *insertIntoDatabase()* koja je bez argumenata, budući da se podaci za slanje prosleđuju prilikom instanciranja njene klase, kao argumenti konstruktora. Unutar nje se kodira slanje POST zahtjeva HTTP protokola upotrebom *Volley* biblioteke, na način opisan u prethodnom poglavlju. Ovakvim zahtjevom se poziva PHP skripta koja je smještena na serveru i koja uzima parametre iz POST zahtjeva koje koristi za formiranje SQL upita kojim se popunjava odgovarajuća tabela unutar baze podataka. Ovdje nećemo ulaziti u objašnjavanje sintakse PHP jezika, već će biti dovoljno objasniti funkcionisanje skripte. Dakle, skripta za primljeni POST zahtjev uključuje kôd *connection.php* skripte kojim se ostvaruje konekcija na server i bazu podataka i koja sadrži kredencijale potrebne za to. Zatim se poziva funkcija *makeEntry()* koja na početku definiše globalnu varijablu *$connect*, definisanu unutar *connection.php* skripte, kojom se reprezentuje ostvarena konekcija. Zatim se „izvlače“ parametri iz primljenog POST zahtjeva i smještaju se u odgovarajuće promjenljive. Potom se definiše promjenljiva u kojoj se čuva string koji definiše SQL upit kojim se vrši unos parametara u bazu. Dakle, prema upitu, sve vrijednosti POST parametara se upisuju u tabelu *Entries*, i to u kolone sa nazivom *municipality*, *coordinatePhi*, *coordinateLambda*, *date* i *time*. Struktura baze je detaljno opisana u prethodnoj sekciji. Zatim se upit izvršava pozivom funkcije *mysqli_query(...)* kojoj se kao argumenti prosleđuju promjenljiva *$connect*, na osnovu koje uspostavlja konekciju sa bazom, i *$query* koja sadrži upit koji treba izvršiti. Ova funkcija vraća *true* ili *false* rezultat u zavisnosti od uspješnosti izvršavanja upita, te je to iskorišćeno kako bi skripta nakon pozivanja slala adekvatnu poruku odgovora u vidu stringa ('success' u slučaju uspješnog

unosa ili 'fail' u slučaju neuspješnog). Potom se konekcija zatvara pozivom metode *mysqli_close(...)*. String smješten u poruci odgovora se očitava unutar aplikacije (preklapanjem *onResponse(...)* callback metode unutar osluškivača odgovora – sastavni dio *Volley* biblioteke) te se kao takav čuva u promjenljivu članicu *DatabaseHandler* klase, kako bi ovaj podatak bio dostupan i ostalim metodama.

Potom slijedi slanje nizova bajtova. Za te svrhe *DatabaseHandler* sadrži metodu *imageInsert(...)* koja za razliku od gore pomenute metode za slanje podataka o fotografiji sadrži pregršt argumenata koji omogućavaju da se metodi pozvanoj unutar klase foto panela proslijedi kolekcija sa bajtovima, kolekcija sa nazivima fotografija, kao i još neke varijable potrebne za realizaciju obavještavanja korisnika o procesu slanja prikazivanjem odgovarajućih poruka i izvještaja o napretku na korisničkom interfejsu. Uz to, u zaglavlju metode je definisano moguće bacanje izuzetka do kog će doći ukoliko je prethodni upis podataka u bazu bio neuspješan. Stoga prilikom poziva unutar klase aktivnosti panela za slanje ova metoda mora biti uokvirena *try-catch* blokom. Dakle, ukoliko metoda baci izuzetak, preći će se na izvršavanje kôda *catch* bloka unutar kog je realizovano generisanje iskačućeg prozora sa adekvatnim obavještenjem o neuspjelom slanju, kao i čuvanje svih fotografija u galeriju za kasnije slanje. Bitno je napomenuti i da se prije iniciranja slanja fotografija, odnosno pozivanja metode *imageInsert(...)*, provjerava dostupnost konekcije, te se, u slučaju kada je ona nedostupna, korisniku takođe daje obavještenje u vidu iskačućeg prozora a slike se čuvaju na uređaju kako bi bile dostupne za ponovno slanje. Ukoliko je unos u bazu, pak, bio uspješan, započinje se sa iniciranjem FTP konekcije i slanjem svake pojedinačne fotografije, tj. niza bajtova. Ovo je rađeno upotrebom biblioteke SimpleFTP čiji je pregled dat u prethodnom poglavlju. Budući da se ovo slanje obavlja na pozadinskoj niti, tačnije unutar *doInBackground(...)* metode *AsyncTask*-a, realizovano je i publikovanje progresa nakon svake etape povezivanja na server i nakon slanja svake fotografije, opet sa ciljem izbacivanja odgovarajućeg obavještenja na korisnički interfejs. Publikovanje progresa obuhvata pozivanje metode *publishProgress(...)* kojoj se kao argument prosleđuje string kojim se definiše odgovarajuća poruka o progresu na osnovu koje će se vršiti odgovarajući set operacija unutar metode *onProgressUpdate(...)*. Nakon svakog uspješnog slanja pojedinačne slike poziva se i metoda *imageDataInsert(...)* koja vrši unos u tabelu *Photos* na isti način na koji i metoda *insertIntoDatabase()* vrši unos u tabelu *Entries*. Ovoj funkciji se kao argument tipa *String* prosleđuje naziv fotografije sa ekstenzijom formata koji se stavlja unutar POST zahtjeva kao parametar zahtjeva, zajedno sa nazivom direktorijuma u koji je fotografija sačuvana, što je podatak sačuvan u odgovarajućoj promjenljivoj članici klase *DatabaseHandler*. Sem toga, deklarisana je posebna cjelobrojna promjenljiva koja je stavljena u funkciju brojača uspješno poslatih fotografija. Na kraju, po raskidanju FTP konekcije, po poslednji put se publikuje progres sa porukom koja zavisi od vrijednosti tog brojača. Poruka može signalizirati uspješno slanje, kada je vrijednost brojača jednaka broju fotografija, djelimično uspješno slanje, kada je vrijednost brojača manja od broja fotografija, ali veća od nule, i neuspješno slanje, kada je vrijednost brojača jednaka nuli. Na osnovu ovoga se, opet, generiše iskačući prozor na korisničkom interfejsu, a u slučaju neuspjelog slanja fotografije se čuvaju u galeriju, i poziva se metoda *deleteFromDatabase(...)* koja briše poslednji unos u tabelu *Entries*. Na taj način se sprečava mogućnost da se u bazi podataka pojavi neki unos za koji nije vezana ni jedna fotografija. Ovo brisanje se takođe obavlja slanjem HTTP zahtjeva i pokretanjem PHP skripte na serveru, koja je po strukturi identična skripti čiji je kôd ovdje uzet za primjer, s tom razlikom što ona ne uzima nikakve POST parametre, i ima drugačije formulisan SQL upit koji je zadužen za brisanje podataka iz tabele, a ne njihov unos. Iako je prije iniciranja slanja fotografija uspješno izvršen unos podataka u bazu, što sugeriše da je veza sa internetom dobra, ipak je i u ovoj metodi razvijen sistem za detektovanje grešaka prilikom slanja, budući da postoji mogućnost da se internet konekcija naknadno izgubi, ili da nastanu neki problemi u radu FTP servera prije ili u toku slanja. Problemi mogu nastati i prilikom uspostavljanja veze sa FTP serverom, ali, budući da metoda *connect()* baca izuzetak, ovaj scenario je pokriven uokvirivanjem čitavog kôda u *try-catch* blok, i realizacije obrade izuzetka u vidu publikovanja iste one poruke o progresu koja se šalje kada su sve fotografije neuspješno poslate.



Slika 13. Primjer formata iskačućeg prozora

Pomenuto je da je u sklopu slanja fotografija realizovan i mehanizam njihovog čuvanja u slučaju nedostupnosti internet konekcije, neuspješnosti slanja na server i sl. Stoga bi se valjalo osvrnuti i na način na koji je ovo realizovano. U tu svrhu, unutar klase foto panela realizovana je posebna statička metoda. Razlog zbog kog je ona statička je taj što postoji potreba za njenim pozivanjem kako iz klase foto panela, tako i iz klase *DatabaseHandler*. Kako bi izbjegli komplikacije koje bi izazvalo instanciranje klase panela unutar klase *DatabaseHandler* kao jednostavno rješenje nametnulo se proglašavanje ove metode statičkom. Argumenti koji se prosleđuju ovoj metodi jesu kontekst pozivajuće aktivnosti (u ovom slučaju to će uvijek biti aktivnost panela sa fotografijama, ali je zbog statičkog karaktera metode ipak neophodno proslijediti i ovu vrijednost kao argument), kolekcija sa nizovima bajtova, *hasImage* niz logičkih vrijednosti, i prefiks imena fotografija (šalje se samo prefiks budući da se puno ime može kreirati dodavanjem rednog broja prilikom čuvanja). Čuvanje fotografija se takođe obavlja na pozadinskoj niti.

```
...

boolean ind = false;

String savingPath = Environment.getExternalStorageDirectory().getAbsolutePath() + "/LOVCEN";
File dir = new File(savingPath);

if (!dir.exists())
    dir.mkdir();

if (hasImage != null) {

    for (int i = 0; i < 9; i++) {

        if (hasImage[i]) {

            ind = true;

            File imageSave = new File(dir, baseName + "0" + Integer.toString(i) + ".png");

            try {

                BufferedOutputStream bos = new BufferedOutputStream(new
                                                        FileOutputStream(imageSave));
                bos.write(imagesForSending.get(i));
                bos.flush();
                bos.close();

            } catch (Exception e) {

            }

        }

    }

    if (ind)
        MediaScannerConnection.scanFile(context, new String[]{dir.toString()}, null, null);

...
```

Prije svega, neophodno je u manifest fajlu dodati sledeću liniju dozvole:

```
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
```

Zatim se operacija započinje dobijanjem adrese korjenog foldera memorije uređaja na osnovu koje se kreira putanja direktorijuma u koji će se fotografije čuvati.

```
String savingPath = Environment.getExternalStorageDirectory().getAbsolutePath() + "/LOVCEN";
```

Da bi dobili tu putanju prvo se poziva metoda *getExternalStorageDirectory()* koja vraća objekat tipa *File* kojim se reprezentuje korjeni folder. Pozivom metode *getAbsolutePath()* nad tim fajlom dobijamo apsolutnu putanju do njega (dakle, do korjenog foldera) u vidu stringa na koji nadovezujemo relativnu putanju u odnosu na taj korjeni folder do foldera u kom želimo da naša aplikacija čuva fotografije. Na taj način se dobija apsolutna putanja do tog foldera koja se čuva u neku varijablu za kasnije korišćenje. Nakon toga se instancira objekat tipa *File* kom se kao argument konstruktora prosleđuje upravo dobijena apsolutna putanja. Treba napomenuti da instanciranje objekta tipa *File* ne znači nužno otvaranje tog fajla, već se tim objektom dobija apstraktna reprezentacija nekog fajla, u ovom slučaju foldera za smještanje aplikacije. Stoga, da bi taj folder bio zaista kreiran, potrebno je nad tim *File* objektom pozvati metodu *mkdir()* (naravno, prethodno provjerivši da li taj folder već postoji).

U slučaju kada se metoda, budući statička, poziva iz klase foto panela, onda je potrebno koristiti *hasImage* niz, kako bi se poslale samo važeće fotografije iz kolekcije niza bajtova. U slučaju da se poziva iz *DatabaseHandler*-a tada će se ovoj metodi proslijediti nova kolekcija koja sadrži samo fotografije koje treba da se šalju, te stoga nema potrebe za korišćenjem niza *hasImage*, na čije mjesto se pri pozivu ove metode prosleđuje vrijednost *null*. Dakle, selekcija sa uslovom `hasImage != null` služi da razdvoji operacije koje će se vršiti prilikom poziva iz jedne ili druge klase. Kao što vidimo iz primjera, svaka fotografija se čuva zasebno. Prvo se pravi referenca na novi fajl, prosleđujući konstruktoru direktorijum za smještanje i naziv fajla sa ekstenzijom kao argumente. Ova ekstenzija definiše fajl kao sliku komprimovanu u *.png* format. Zatim se otvara izlazni tok podataka prema fajlu, u koji se upisuju bajtovi, nakon čega se tok čisti i zatvara. Na ovaj način fajl je kreiran i sačuvan unutar naznačenog direktorijuma. Međutim, samo čuvanje fotografija ne znači da će one automatski biti prikazane unutar galerije. U tu svrhu je potrebno sugerisati galeriji da postoji direktorijum sa fotografijama koje treba da budu prikazane unutar nje, što odrađuje metoda *scanFile(...)*, članica klase *MediaScannerConnection*. Svrha ove klase jeste upravo omogućavanje skeniranja novih fajlova koji su kreirani ili preuzeti od strane neke aplikacije i njihovog dodavanja i prikazivanja unutar adekvatnog media content provider-a, kao što je galerija. Kao prvi argument se prosleđuje kontekst pozivajuće aktivnosti, a kao drugi niz stringova koji predstavlja sve putanje koje želimo da pokrijemo skeniranjem. Treći argument se odnosi na definisanje tipova fajlova (takođe predstavljenih u vidu niza stringova) za svaku pojedinačnu putanju koju treba uzeti u obzir prilikom skeniranja (ukoliko se proslijedi *null*, kao što je ovdje slučaj, tip fajla se određuje na osnovu ekstenzije), dok se kao četvrti može proslijediti osluškivač koji se aktivira prilikom završenog skeniranja, odnosno proslijediti *null* vrijednost ukoliko nema potrebe za definisanjem osluškivača. Na ovaj način je obezbijeđeno da se sačuvane fotografije pojave u galeriji prilikom njenog prvog narednog otvaranja.

Jedan vrlo interesantan mehanizam implementiran u sklopu ove funkcionalnosti jeste automatska akvizicija lokacije. Ovaj mehanizam podrazumijeva prije svega preuzimanje koordinata sa GPS-a i njihovo prikazivanje, kao i određivanje naziva opštine kojoj tačka sa tom koordinatom pripada. Preuzimanje koordinata je prilično jednostavno i obavlja se pomoću ugrađenih biblioteka za upravljanjem GPS hardverom: *android.location.Location*, *android.location.LocationListener* i *android.location.LocationManager*.

```
...

LocationManager locationManager;
locationManager = (LocationManager) getActivity().getSystemService(Context.LOCATION_SERVICE);

locationListener = new LocationListener() {

    ...
```

```
    ...

    @Override
    public void onLocationChanged(final Location location) {

        if ( isAdded() && radioAuto.isChecked() ) {

            double latitudeCoordinate = location.getLatitude();
            double longitudeCoordinate = location.getLongitude();
            latitude.setText(String.format("%.4f N", latitudeCoordinate));
            longitude.setText(String.format("%.4f E", longitudeCoordinate));

            String municipality = getMunicipality(latitudeCoordinate, longitudeCoordinate);

            if ( !municipality.equals(locationEntry.getText().toString()) )
                locationEntry.setText(municipality);

        }

    }

    @Override
    public void onStatusChanged(String provider, int status, Bundle extras) {

    }

    @Override
    public void onProviderEnabled(String provider) {
        if ( isAdded() && provider.equals(LocationManager.GPS_PROVIDER) )
            radioAuto.performClick();
    }

    @Override
    public void onProviderDisabled(String provider) {
        if ( isAdded() && provider.equals(LocationManager.GPS_PROVIDER) )
            //Prebacivanje na ručni odabir
    }


};

...
```

Da bi se došlo do GPS koordinata, prvo je potrebno dobiti instancu menadžera lokacije. Ova instanca se dobija pozivom metode *getSystemService(Context.LOCATION_SERVICE)* nad instancom aktivnosti. Kako se ovdje radi o fragmentu u kom je funkcionalnost realizovana, ta instanca se može dobiti pozivom metode *getActivity()*. Sada se koordinate preko menadžera lokacije mogu dobiti sa nekoliko linija koda:

```
if (locationManager.isProviderEnabled(LocationManager.GPS_PROVIDER)) {

    locationManager.requestLocationUpdates(LocationManager.GPS_PROVIDER, 1000, 5,
                                                                    locationListener);
    locationManager.requestLocationUpdates(LocationManager.NETWORK_PROVIDER, 1000, 5,
                                                                    locationListener);
}
```

Dakle, ukoliko je GPS provajder dostupan nad *locationManager* objektom se poziva metoda *requestLocationUpdates(...)* kojom se definiše način ažuriranja lokacije. Dostupnost GPS provajdera znači, prosto, da je GPS na telefonu uključen i ne zavisi od dostupnosti ili jačine samog satelitskog signala. Prvi argument ove metode je statička promjenljiva kojom se definiše provajder lokacije, u ovom slučaju GPS hardver. Drugi definiše vrijeme nakon kojeg se lokacija ažurira, a treći minimalna promjena lokacije u metrima koja mora da se dogodi da bi došlo do njenog ažuriranja. Dakle, koordinate se osvježavaju nakon jedne sekunde ili napravljenog pomaka od barem 5 metara. I, konačno, četvrti argument je osluškivač koji će se aktivirati prilikom ažuriranja lokacije i koji će da obavi sve potrebne radnje koje to ažuriranje povlači

za sobom. Ova metoda je pozvana još jednom, sada sa mrežnim provajderom kao provajderom lokacije, tako da su sada za menadžer lokacije vezana dva provajdera. Za razliku od lokacije dobijene sa GPS-a koja je relativno precizna, sa desetak metara greške ili manje, lokacija dobijena od mrežnog provajdera se određuje na osnovu lokacije bazne stanice sa koje uređaj prima telefonski signal, te je stoga ona vrlo gruba i okvirna, ali u slučaju da je satelitski signal u datom trenutku nedostupan, i grubo određivanje koordinata može da posluži sasvim dobro, budući da je naziv opštine podatak koji je nama zaista potreban, a ne precizna lokacija. Naravno, vrlo je bitna stvar definisati dozvolu za dobijanje lokacija unutar manifest fajla. To se radi dodavanjem sledećih dozvola:

```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"   />
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
```

Slika 14. Mapa CG sa obojenim teritorijama opština

Osluškivač lokacije je klasa koja implementira interfejs *android.location.LocationListener* koji sadrži četiri callback metode, prikazane na ovom primjeru. Prva preklopljena metoda je *onLocationChanged(...)* koja se poziva prilikom ažuriranja lokacije i kojoj se prosleđuje argument tipa *Location* u kom su sadržani svi podaci o lokaciji. Na osnovu toga, ukoliko su ispunjeni uslovi (prvi - fragment je zakačen za svoju aktivnost, što se provjerava metodom *isAdded()*, i drugi – odabrano je automatsko određivanje lokacije) očitavaju se koordinate metodama *getLatitude()* i *getLongitude()*, vrši se prikaz istih na grafičkom interfejsu (slika 11), a zatim se na osnovu dobijenih koordinata određuje opština metodom *getMunicipality(...)*, koja, naravno, nije ugrađena, već je realizovana kao metoda članica klase fragmenta koji sadrži početni interfejs. Ukoliko je detektovana opština različita od prethodno određene opštine, vrši se izmjena i tog podatka. Što se ostalih callback metoda tiče, metoda *onStatusChanged(...)* se poziva pri promjeni statusa provajdera (što ovdje nije bilo potrebno te je ta metoda ostala neprekloplјena), a metode *onProviderDisabled(...)* i *onProviderEnabled(...)* se pozivaju kada korisnik sâm promjeni dostupnost provajdera, tačnije pri njegovom isključenju, odnosno uključenju, respektivno. Pri pozivu ovih metoda kao argument im se prosleđuje naziv provajdera kao objekat tipa *String*. Ovdje je definisano da se operacije unutar obje metode izvršavaju jedino ukoliko su ispunjena dva uslova: fragment je pridodat roditeljskoj aktivnosti i provajder usled čije promjene je metoda pozvana je GPS provajder. U slučaju kada se taj provajder aktivira odabir lokacije se prebaca na automatski, dok se u slučaju deaktivacije ovog provajdera definiše kôd kojim se prikazuje odgovarajuće obavještenje na korisničkom interfejsu.

Određivanje koordinata je, manje-više, instanciranje i preklapanje ugrađenih klasa i metoda. Ono što je interesantnije za razmatranje jeste način na koji se na osnovu dobijenih koordinata određuje kojoj opštini na teritoriji Crne Gore ta tačka pripada. Ideja je sledeća: uzeti sliku mape Crne Gore na kojoj je teritorija svake opštine obojena različitom bojom (slika 14), zatim, poznavajući koordinate najekstremnijih tačaka (istok, zapad, sjever i jug), preslikati geografsku koordinatu u poziciju piksela na slici i očitati boju piksela (RGB) na toj poziciji. Na osnovu poklapanja te boje sa nekom od unaprijed dodijeljenih boja opština, dobijamo vrlo preciznu identifikaciju opštine na čijoj se teritoriji tačka sa dobijenim koordinatama nalazi. Ukoliko do poklapanja ne dođe, možemo zaključiti da se radi o tački van teritorije Crne Gore. Upravo ovaj koncept stoji iza realizacije metode *getMunicipality(...)*:

```
public String getMunicipality(double phi, double lambda){

    Bitmap bitmap = BitmapFactory.decodeResource(getResources(), R.drawable.gps_colour);
    double test = lambda;

     ...
```

```
    ...

    StringBuilder municipality = new StringBuilder();

    int bitmapWidth = bitmap.getWidth();
    int bitmapHeight = bitmap.getHeight();

    int ind = -1;

    if (phi > montenegroSouth && phi < montenegroNorth && lambda > montenegroWest &&
                                                                lambda < montenegroEast) {

        int pixelPositionX = (int) ((lambda - montenegroWest) * (bitmapWidth /
                                                        (montenegroEast - montenegroWest)));
        int pixelPositionY = (int) (bitmapHeight - (phi - montenegroSouth) * (bitmapHeight /
                                                    (montenegroNorth - montenegroSouth)));

        int pixel = bitmap.getPixel(pixelPositionX, pixelPositionY);

        ind = Opstina.identify(Color.red(pixel), Color.green(pixel), Color.blue(pixel),
                                                        municipality, MainActivity.opstine);

    }

    if (ind == -1)
        return "VAN TERITORIJE CG";

    return municipality.toString();

}
```

Promjenljive *montenegroSouth*, *montenegroNorth*, *montenegroWest* i *montenegroEast* su unaprijed definisane statičke promjenljive koje sadrže maksimalne i minimalne vrijednosti geografske širine i dužine neke tačke na teritoriji Crne Gore, što je podatak neophodan kako bi se vršilo pravilno preslikavanje na poziciju piksela.

Na ovom primjeru se vidi da je u svrhu identifikacije opštine takođe realizovana još jedna metoda. Naime, svaka opština modelovana je klasom *Opstina* koja kao promjenljive članice sadrži naziv opštine, R, G i B vrijednosti njene boje, kao i resurs ID posebne slike koja je neophodna za realizaciju interaktivne mape, o čemu će biti riječ u narednoj sekciji. Sve opštine se instanciraju pri pokretanju centralne aktivnosti i smještaju u posebnu kolekciju. Sem toga, klasa sadrži i statičku metodu *identify(...)* koja kao argumente prima R, G, i B vrijednosti boje za poređenje, kolekciju objekata tipa *Opstina* koja je unaprijed definisana i koja se formira pri samom startovanju aplikacije, kao i promjenljivu tipa *StringBuilder* koja će se unutar metode modifikovati tako da sadrži naziv identifikovane opštine upisan kao string (ovdje je primijenjena ista logika koja se koristi u C ili C++ jeziku kada koristimo prosleđivanje parametara funkcije po referenci). Ova metoda kao rezultat vraća resurs ID gore pomenute slike koja se koristi u realizaciji interaktivne mape, a koja odgovara identifikovanoj opštinu. Zbog toga naziv opštine nije mogao da se dobije direktno kao rezultat koji metoda vraća, budući da tehnički postoji potreba za vraćanjem dvije vrijednosti. Na ovaj način je u cjelosti opisana prva skupina funkcionalnosti koja se vrti oko slanja fotografija i ostalih podataka na server.

4.2.3 Interaktivna mapa

Jedna od ideja za realizaciju u sklopu VECTOR INSPECT aplikacije jeste korisnički interfejs preko kog bi korisnici u svakom trenutku mogli da imaju uvid u pojedine rezultate istraživanja, u prvom redu spisak identifikovanih vrsta i procenat javljanja na teritoriji odabrane opštine. Kao idealno rješenje se nameće oblikovanje interfejsa u vidu mape (slika 15) koja klikom na neku opštinu generiše odgovarajuće podatke i prikazuje ih korisniku. Ovakvo rješenje, osim što je praktično, je izuzetno primamljivo i sa aspekta dizajna aplikacije.

Slika 15. Interaktivna mapa

Mapa je realizovana unutar posebnog fragmenta na sledeći način: osnovu interfejsa čine dva *ImageView*-a - jedan vidljivi koji sadrži mapu koja se prikazuje korisniku, i drugi koji je u potpunosti transparentan i koji sadrži mapu sa obojenim teritorijama opština. Ovaj *ImageView* je postavljen preko vidljivog kako bi se prilikom dodira aktivirao upravo njegov osluškivač. Ukoliko bi se učinio nevidljivim tako što bi se njegova vidljivost postavila na *INVISIBLE*, njegov osluškivač ne bi reagovao na dodir, te je umjesto toga isti efekat postignut maksimalnim povećanjem transparentnosti komponente. Sam osluškivač je realizovan tako da koordinatu tačke dodira prvo preslika u položaj piksela na slici komponente, zatim uzima podatak o boji toga piksela i na osnovu boje vrši identifikaciju opštine. Dakle, mehanizam je gotovo isti kao i mehanizam detekcije opštine na osnovu GPS koordinate, s tom razlikom što je sada izvor koordinata tačaka dodir na ekran, a ne globalni pozicioni sistem, i što je sada polazni koordinatni sistem postavljen u odnosu na *imageView* komponentu, a ne površinu zemaljske kugle. Takođe, koristi se nešto drugačija obojena mapa (npr. vodene površine se sada ne posmatraju kao dio teritorije neke opštine). Nakon što je opština detektovana, uzima se resurs ID njene slike i ista se postavlja na vidljivu *imageView* komponentu. Ova slika sadrži mapu koja je identična polaznoj sem što je teritorija date opštine istaknuta drugom bojom. Na taj način se prilikom dodira stvara efekat označavanja cjelokupne teritorije opštine kojoj „pripada“ tačka dodira ekrana. Uz to, na mjesto dodira se postavlja oblak sa informacijama i to tako da se vrh špica oblaka nalazi baš na toj tački. Pojavljivanje oblaka je animirano pomoću *ObjectAnimator*-a, a kao pivot animacije je postavljena upravo tačka vrha špica. Oblak je modelovan kao relativni layout koji za pozadinu ima jedan od šest različitih tipova grafičkog prikaza oblaka koji se razlikuju po položaju špica. Tip koji će biti prikazan se bira u zavisnosti od toga da li cjelokupan staje na ekran, u skladu sa gore definisanim odabirom pivot tačke. Animacija je složenog karaktera i stvara svojevrsni iskačući efekat iz tačke dodira (overshoot). Dodirom na oblast van teritorije Crne Gore oblak se zatvara, a mapa Crne Gore se vraća na početnu, neoznačenu.

municipality	speciesName	speciesNumber	speciesTotal
DANILOVGRAD	Culex pipiens	3	6
DANILOVGRAD	Aedes albopictus	2	6
CETINJE	Culex pipiens	1	1
ULCINJ	Aedes vexanus	1	2
BUDVA	Culex pipiens	1	1
ULCINJ	Aedes aegypti	1	2
BERANE	Aedes albopictus	1	1
DANILOVGRAD	Aedes aegypti	1	6
PLJEVLJA	Culex pipiens	1	1
BAR	Aedes albopictus	1	1

Slika 16. Složena tabela

Prilikom pokretanja ovog fragmenta prvo se započinje preuzimanje podataka iz baze, tačnije iz složene tabele, za vrijeme kog se na interfejsu prikazuje animacija. Zapravo, prije svega se provjerava dostupnost internet konekcije na isti način na koji je to rađeno prilikom upisivanja podataka. Jedino ukoliko je konekcija dostupna, započeće se preuzimanje podataka, dok će u suprotnom na interfejsu biti prikazano odgovarajuće obavještenje. Mapa se prikazuje tek nakon što se obavi učitavanje podataka. I u ovom slučaju, upravljanje bazom podataka se obavlja pomoću klase *DatabaseHandler*. Za potrebe preuzimanja podataka kreirana je metoda *readFromDatabase(...)* unutar koje je realizovan HTTP zahtjev, takođe upotrebom *Volley* biblioteke, kojim se pokreće skripta na serveru koja je isprogramirana tako da uzima podatke iz baze, pakuje ih u niz JSON objekata i šalje aplikaciji. Ti podaci su redovi posebne složene tabele pri čemu svaki JSON objekat odgovara njenom jednom redu. Način na koji je ovakav prijem podataka realizovan je opisan u prethodnom poglavlju kada je govoreno o komunikaciji sa serverom. Kako svaki

JSON objekat predstavlja jedan red tabele, to se iz njega može izvući vrijednost pojedinačnih polja u vidu stringova. Ti stringovi se koriste za kreiranje instance objekta klase *DatabaseRead* kojom se modeluje jedan očitani red (varijable članice ove klase predstavljaju po jedno polje iz očitanog reda). Dakle, svaki red tabele se prvo preuzima u vidu JSON objekta koji se zatim pretvara u objekat tipa *DatabaseRead*, i taj objekat se pridružuje kolekciji. Ta kolekcija je, dakle, reprezent tabele koji se vraća kao rezultat izvršavanja ove metode i format u kom aplikacija dobija očitane podatke. Kao argument, ovoj funkciji se prosleđuje objekat tipa *StringBuilder* u koji se upisuje odgovarajuću poruka po završetku preuzimanja podataka. Dakle, i ova metoda vraća dvije vrijednosti, jednu kao rezultat a drugu mijenjanjem proslijeđenog argumenta. Uporedo sa pozivanjem ove metode unutar klase ovog framenta pokreće se pozadinska nit na kojoj će se vršiti stalno ispitivanje, odnosno, polling proslijeđenog *StringBuilder* objekta. Polling obezbjeđuje da se mapa prikaže tek u onom trenutku kada taj objekat promijeni svoju vrijednost, čime se signalizira da su svi podaci preuzeti. Prema tome, preuzimanje podataka se odvija na tri niti: glavna nit koja na grafičkom interfejsu dodaje animaciju za učitavanje, prva pozadinska nit na kojoj se vrši preuzimanje podataka i koja se pokreće automatski sa *Volley* zahtjevom i druga koja je pokrenuta unutar *AsyncTask*-a i koja polling metodom provjerava da li su svi podaci iz baze preuzeti. PHP skripta koja neposredno komunicira sa bazom podataka kodirana je na sledeći način:

```
<?php

	if ( $_SERVER["REQUEST_METHOD"] == "POST" ) {
		include 'connection.php';
		readDatabase();
	}

	function readDatabase(){

		global $connect;

		$query = "SELECT t1.municipality, t1.speciesName, t1.speciesNumber,
t2.speciesTotal FROM (SELECT e.municipality, s.speciesName, count(*)\"speciesNumber\" FROM
species s, entries e WHERE e.entry_id = s.ENTRY_ID GROUP BY e.municipality, s.speciesName
ORDER BY e.municipality) t1, (SELECT e.municipality, count(e.municipality)\"speciesTotal\"
FROM species s, entries e WHERE e.entry_id = s.ENTRY_ID GROUP BY e.municipality ORDER BY
e.municipality) t2 WHERE t1.municipality = t2.municipality ORDER BY t1.speciesNumber DESC";


		$result = mysqli_query($connect, $query);
		$number_of_rows = mysqli_num_rows($result);

		$tmp_array = array();

		if ( $number_of_rows > 0 ) {
			while ( $row = mysqli_fetch_assoc($result) ) {
				$tmp_array[] = $row;
			}
		}

		echo json_encode( array("species" => $tmp_array) );


		mysqli_close($connect);

	}


?>
```

Vidimo da je i ova skripta formirana po sličnom šablonu kao i skripte koje upisuju podatke u bazu, s tim što je u ovom slučaju realizovan upit skroz drugačijeg tipa, a takođe su dodate i linije koda kojima se vrši obrada očitanih podataka i pakovanje u JSON objekte. Kao što je već nagoviješteno, redovi koji se dobijaju ovakvim SQL upitom ne pripadaju ni jednoj od postojećih tabela u bazi, već su to redovi novoformirane

tabele koja nastaje njihovim spajanjem (slika 16). Ta tabela sadrži sve podatke koji su namijenjeni za prikazivanje korisniku u okviru interaktivne mape. Prva kolona *municipality* sadrži nazive opština u kojoj je neka vrsta identifikovana, *speciesName* naziv te vrste, *speciesNumber* broj identifikacija koje se odnose na tu vrstu unutar date opštine, i *speciesTotal* ukupan broj identifikacija svih vrsta na teritoriji te opštine. Dakle, na osnovu ovih podataka dobijenih spajanjem osnovnih tabela, može se automatski napraviti statistika javljanja svake vrste unutar svih opština. Podaci složene tabele su sortirani prema broju javljanja vrsta u okviru jedne opštine, po opadajućem redoslijedu.

Iskačući oblak, pored naziva opštine i podatka o ukupnom broju identifikovanih vrsta sadrži i dugme kojim se prelazi na novu aktivnost unutar koje se prikazuju podaci poslati sa serverske strane. Prilikom poziva te aktivnosti, vrši se filtriranje preuzetih podataka tako da ostanu samo oni podaci, odnosno redovi tabele, koji u svom nazivu imaju naziv odabrane opštine. Zatim se ti podaci šalju adapteru liste koji listu popunjava odgovarajućim podacima, i to nazivom vrste kao i procentom njenog javljanja koji računa na osnovu broja identifikacija date vrste i ukupnog broja identifikacija u okviru date opštine (slika 17). Što je broj identifikacija veći, to će statistika o rasprostranjenosti vrsta biti preciznija.

Da bi elemente liste popunili željenim podacima potrebno je napraviti sopstveni adapter liste. Adapter liste je klasa koja je izvedena iz klase *ArrayAdapter* koja služi za popunjavanje liste odgovarajućim podacima, kao i za definisanje izgleda pojedinačnog elementa. Iako je moguće odmah

```
public class SpeciesListAdapter extends ArrayAdapter {

    private Context context;
    private ArrayList<DatabaseRead> readData;

    public SpeciesListAdapter(Context context, ArrayList<DatabaseRead> readData) {
        super(context, R.layout.listview_layout, readData);
        this.context = context;
        this.readData = readData;
    }

    @Override
    public View getView(int position, View convertView, ViewGroup parent) {

        SpeciesList.ViewHolder viewHolder;

        if (convertView == null) {

           LayoutInflater layoutInflater = (LayoutInflater)
           context.getApplicationContext().getSystemService(Context.LAYOUT_INFLATER_SERVICE);
           convertView = layoutInflater.inflate(R.layout.listview_layout, parent, false);

           viewHolder = new SpeciesList.ViewHolder();
           viewHolder.speciesName = (TextView) convertView.findViewById(R.id.speciesName);
           viewHolder.freq = (TextView) convertView.findViewById(R.id.freq);

           convertView.setTag(viewHolder);

        } else
           viewHolder = (SpeciesList.ViewHolder) convertView.getTag();


        DatabaseRead data = readData.get(position);

        viewHolder.speciesName.setText( data.getSpeciesName() );

        float percentage = Float.parseFloat( data.getSpeciesNumber() ) / Float.parseFloat(
                                                        data.getSpeciesTotal() ) * 100;
        viewHolder.freq.setText( String.format("%.2f", percentage) + " %");

        return convertView;

    }
}
```

instancirati klasu *ArrayAdapter* i tu instancu vezati za listu kao adapter, to ipak nije učinjeno jer ovakav način ograničava mogućnosti dizajniranja komponenti liste, budući da u tom slučaju elemenat liste može da sadrži samo jednu *TextView* komponentu. Generalno je dobra praksa definisati sopstveni adapter liste u vidu nove, izvedene klase. Tri su ključne komponente neophodne za formiranje adaptera liste: izgled pojedinačne stavke liste, koji se definiše u *.xml* fajlu i smješta u *layout* resursnom folderu, podaci kojima će se lista popunjavati i kontekst aktivnosti unutar koje je lista smještena. Podaci za listu i kontekst se unutar klase kojom se definiše adapter čuvaju kao promjenljive članice koje se inicijalizuju unutar konstruktora. Tu je, takođe, neophodno pozvati i konstruktor nadklase kome se kao argumenti prosleđuju ove tri stavke. Najbitnija stvar pri definisanju klase adaptera liste je preklapanje metode *getView(...)*. Ovo je metoda koja se poziva za svaku stavku liste pojedinačno i koja vraća objekat tipa *View* koji će se u listi prikazati kao pojedinačni element. Metoda će se pozvati prilikom prvog prikaza liste ili prilikom skrolovanja, odnosno svaki put kada novi element liste treba da se prikaže na ekranu uređaja, i prilikom poziva će joj se automatski, kao argumenti, proslijediti pozicija elementa liste, referenca na *View* komponentu koja definiše jedan element liste i referencu na objekat tipa *ViewGroup* za koji se ta komponenta „kači“. Kako bi se optimizovale performanse liste, uveden je sistem recikliranja komponenti. Naime, definiše se posebna klasa nazvana *ViewHolder* (ovdje je definisana kao unutrašnja klasa klase aktivnosti liste) koja će u vidu elemenata članica čuvati reference na komponente layout fajla kojim se definiše izgled jednog elementa. Ukoliko objekat *convertView* ne postoji (dakle, proslijeđena je *null* vrijednost), to znači da taj element liste treba prvi put da se prikaže od startovanja aktivnosti, te ga stoga treba kreirati na osnovu definisanog layout fajla. U te svrhe definiše se objekat *LayoutManager* tipa nad kojim se poziva metoda *inflate(...)* kojoj se kao jedan od argumenata prosleđuje ID tog layout fajla i koja će vratiti referencu na objekat tipa *View*. Upravo taj objekat je grafička komponenta koja predstavlja jedan element liste. Takođe, kada se radi o prvom prikazu elementa potrebno je uzeti referencu na sve komponente ovog layout fajla upotrebom standardne metode *findViewById(...)*. Međutim, kako pozivanje ove metode takođe troši procesorsko vrijeme, to se referenca na svaki od elemenata liste čuva unutar posebne klase koja se po pravilu imenuje kao *ViewHolder* i čiji će elementi članovi da čuvaju potrebne reference. U ovom slučaju, ta klasa je definisana kao unutrašnja klasa klase aktivnosti liste koja kao članove sadrži dva objekta tipa *TextView* budući da su to jedine dvije komponente za koje postoji potreba da se referenciraju iz klase adaptera liste. Da bi se *ViewHolder* klasa vezala za dati *View* objekat potrebno ju je prvo instancirati, a onda vezati metodom *setTag()* koja se poziva nad tim *View* objektom. Ukoliko proslijeđeni *convertView* objekat postoji (dakle, elemenat liste je već bio prikazivan), onda nema potrebe da se ponovo prolazi kroz cijelu proceduru kreiranja *View* objekta i referenciranja njegovih komponenti, već je dovoljno iskoristiti već kreiranu komponentu, na taj način što će se pozivom metode *getTag()* nad tim objektom dobiti referenca na njegovu *ViewHolder* klasu, a samim tim i referenca na sve njegove komponente, bez potrebe ponovnog pozivanja *findViewById(...)* metode. Na ovaj način je recikliranjem komponente ostvareno značajno poboljšanje u performansama. Konačno, kada je klasa adaptera liste ovako definisana potrebno ju je u klasi aktivnosti liste instancirati i metodom *setAdapter(...)*, koja se poziva nad objektom liste i kojoj se kao argument prosleđuje ta instanca, vezati za tu listu.

Slika 17. Spisak identifikovanih vrsta za odabranu opštinu

Budući da prosječan korisnik aplikacije nije vrsni poznavalac mnogobrojnih vrsta insekata, došlo se na ideju da se kao funkcionalnost liste realizuje otvaranje Google pretraživača unutar web browser-a i započinjanje pretrage za naziv vrste odabranog insekta prilikom klika na bilo koji elemenat liste. Da bi se to realizovalo potrebno je za listu vezati osluškivač koji reaguje na klik na elemenat liste. To je osluškivač koji implementira interfejs *AdapterView.OnItemClickListener()*, sa callback metodom *onItemClick(...)*,

koji se za listu registruje metodom *setOnItemClickListener(...)*. Budući da je njegova implementacija identična kao i implementacija ostalih tipova osluškivača, nema potrebe govoriti o načinu na koji je realizovan. Dovoljno je, možda, samo reći da je jedan od argumenata njegove callback metode *View* objekat koji reprezentuje element liste na koji je izvršen klik, iz kojeg se, zatim, uzima ime vrste koje je upisano u taj element, tačnije u *TextView* tog elementa, na osnovu kojeg se započinje Google pretraga. Na ovaj način je u potpunosti opisana realizacija funkcionalnosti interaktivne mape.

5. ZAKLJUČAK

Razvoj operativnih sistema za mobilne uređaje učinio je da mobilni telefoni prevaziđu svoju osnovnu namjenu i nađu primjenu u sve širem krugu oblasti. Google je posebno doprinio tom razvoju stvorivši Android operativni sistem čime je dat znatan doprinos u oblikovanju mobilnih telefona u pametne uređaje kakve danas poznajemo. U skladu sa tim, istraživački tim Lovćen projekta oslanja prikupljanje dijela podataka upravo na mobilne telefone koristeći funkcionalnosti aplikacije opisane u ovom radu. Ideja je da se projekat sa naučnih laboratorija i ekspertskih timova proširi na sve građane Crne Gore bez obzira na njihov stepen poznavanja problematike kojom se ovaj projekat bavi, kako bi se sakupila što veća količina podataka neophodnih za analizu prisutnih vrsta komaraca. To je od ključne važnosti za formiranje što kvalitetnije ekspertske analize sa ciljem predviđanja mogućeg izbijanja zaraznih bolesti koje oni prenose, a koje su aktuelna prijetnja kako Evropi, tako i čitavom svijetu. Sa druge strane, upotrebna vrijednost ove aplikacije prevazilazi samo nadzor nad komarcima, te se stoga može primjenjivati u bilo kom istraživanju koje iziskuje ovakav način prikupljanja podataka, što u potpunosti opravdava sav trud uložen prilikom njenog razvoja.

6. LITERATURA

[1] P. Deitel, H. Deitel: „Android How to Program (3rd edition)“;

[2] J. Steele, N. To: „The Android Developer's Cookbook - Building Applications with the Android SDK“;

[3] P. E. Jaroslav: „Projektovanje informacionih sistema“;

[4] developer.android.com

[5] www.stackoverflow.com