Osiguranje kvalitete programske podrške na Android platformi

Žnidarec, Karlo

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA

Sveučilišni studij

OSIGURANJE KVALITETE PROGRAMSKE PODRŠKE NA ANDROID PLATFORMI

Diplomski rad

Karlo Žnidarec

Osijek, 2018.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Obrazac D1: Obrazac za imenovanje Povjerenstva za obranu diplomskog rada

Osijek, 12.07.2018.

Odboru za završne i diplomske ispite

Imenovanje Povjerenstva za obranu diplomskog rada

Ime i prezime studenta:	Karlo Žnidarec
Studij, smjer:	Diplomski sveučilišni studij Elektrotehnika, smjer Komunikacije i informatika'
Mat. br. studenta, godina upisa:	D 1069, 21.09.2017.
OIB studenta:	43406015286
Mentor:	Prof.dr.sc. Goran Martinović
Sumentor:	Dr.sc. Bruno Zorić
Sumentor iz tvrtke:	
Predsjednik Povjerenstva:	Doc.dr.sc. Josip Balen
Član Povjerenstva:	Dr.sc. Bruno Zorić
Naslov diplomskog rada:	Osiguranje kvalitete programske podrške na Android platformi
Znanstvena grana rada:	**Programsko inženjerstvo (zn. polje računarstvo)**
Zadatak diplomskog rada:	U teorijskom dijelu rada potrebno je opisati različite metodologije razvoja programske podrške s naglaskom na one koje se u većoj mjeri oslanjaju na testiranje. Detaljno prikazati pristupe i metode za testiranje programske podrške. Prikazati najčešće korištene pristupe testiranju programske podrške prilikom razvoja aplikacija za Android platformu. U praktičnom dijelu rada ostvariti aplikaciju koja omogućuje lakše snalaženje sudionicima društvenog događanja uz korištenje pristupa i metoda opisanih u teorijskom dijelu rada. (sumentor: dr.sc. Bruno Zorić)
Prijedlog ocjene pismenog dijela ispita (diplomskog rada):	Izvrstan (5)
Kratko obrazloženje ocjene prema Kriterijima za ocjenjivanje završnih i diplomskih radova:	Primjena znanja stečenih na fakultetu: 3 bod/boda Postignuti rezultati u odnosu na složenost zadatka: 3 bod/boda Jasnoća pismenog izražavanja: 3 bod/boda Razina samostalnosti: 3 razina
Datum prijedloga ocjene mentora:	12.07.2018.

Potpis mentora za predaju konačne verzije rada u Studentsku službu pri završetku studija:	Potpis:
	Datum:

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA
I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

IZJAVA O ORIGINALNOSTI RADA

Osijek, 18.07.2018.

Ime i prezime studenta:	Karlo Žnidarec
Studij:	Diplomski sveučilišni studij Elektrotehnika, smjer Komunikacije i informatika'
Mat. br. studenta, godina upisa:	D 1069, 21.09.2017.
Ephorus podudaranje [%]:	4

Ovom izjavom izjavljujem da je rad pod nazivom**: Osiguranje kvalitete programske podrške na Android platformi**

izrađen pod vodstvom mentora Prof.dr.sc. Goran Martinović

i sumentora Dr.sc. Bruno Zorić

moj vlastiti rad i prema mom najboljem znanju ne sadrži prethodno objavljene ili neobjavljene pisane materijale drugih osoba, osim onih koji su izričito priznati navođenjem literature i drugih izvora informacija.
Izjavljujem da je intelektualni sadržaj navedenog rada proizvod mog vlastitog rada, osim u onom dijelu za koji mi je bila potrebna pomoć mentora, sumentora i drugih osoba, a što je izričito navedeno u radu.

Potpis studenta:

SADRŽAJ

1. UVOD

Kvaliteta programske podrške ostvaruje se raznim alatima i tehnologijama koje će biti detaljno opisane u drugom poglavlju. Razvoj programske podrške uglavnom se odvija suradnjom više osoba, svaki programer ostavlja u kodu kojega je napisao neki svoj potpis, a često se događa da jedan programer mora mijenjati kod koji je prvotno napisan od strane nekog drugog programera. Ukoliko takav kod nije pisan po nekim jasno definiranim pravilima, u takvim se situacijama puno vremena troši na čitanje koda i shvaćanje što koji dio koda predstavlja i u kakvim je odnosima s ostatkom koda. Ukoliko svi u razvojnom timu koriste jedinstvena pravila prilikom pisanja koda, uređivanje tuđeg koda usporedivo je s vremenom potrebnim za uređivanje vlastitog koda upravo zbog jasno definiranih pravila pisanja.

Osiguranju kvalitete se u posljednje vrijeme konstantno daje veći značaj, a programski kod koji je napisan kvalitetno i po pravilima jednostavnije je održavati i mijenjati. Namjera je ovog rada opisati relevantne alate i tehnologije koje se koriste kako bi se osigurala kvaliteta programskog rješenja. Opisani alati i tehnologije primijenit će se tijekom razvoja programskog rješenja za pomoć u snalaženju posjetiteljima konferencije.

U drugom poglavlju obradit će se različiti pristupi razvoju programske podrške, detaljnije će se obraditi dva pristupa, metodu vodopada i agilnu metodu, objasniti razlike te iskazati prednosti i mane pojedine metode. Zatim će biti opisan pristup čistog koda, zašto je pisanje koda upotrebom pravila čistog koda važno i kako ono pridonosi ukupnom razvoju programske podrške, bit će prikazani primjeri koji direktno pokazuju koliko je čisti kod pregledniji od koda koji nije pisan po definiranim pravilima. Uz metode čistog koda spomenuti će se oblikovni obrasci koji pružaju već provjerena rješenja na razne probleme s kojima se programeri susreću, navesti će se oblikovni obrasci koji se najčešće koriste u praksi te pružiti primjere za one najkorištenije. Obraditi će se tri najpoznatije arhitekture, odnosno provjerene obrasce prema kojima se postavlja temelje aplikacije. Korištenje ovakvih provjerenih arhitektura tijekom postavljanja projekta rješava česte probleme najčešće vezane uz odnos grafičkog sučelja i poslovne logike programskog rješenja. Za kraj drugog poglavlja ovog rada reći će se nešto više o samom testiranju programskog rješenja, objasniti će se zašto je važno testirati programski kod, navesti razlike između automatskog i ručnog testiranja, koje su prednosti jednog, odnosno drugog, detaljnije će se objasniti različite vrste testova koji se mogu izvoditi na programskom kodu. Kada se opišu različite vrste testova reći će se nešto više o testnim okvirima za automatsko testiranje, detaljnije objasniti za što se pojedini testni okviri koriste te dati primjere pojedinih testova, nakon testnih okvira objasniti će se što je to testiranje kao usluga te kada i kako se takva usluga koristi.

U trećem poglavlju nazvanom „Programsko rješenje za posjetitelje konferencije“ bavit će se konkretnom implementacijom metoda, tehnologija i alata opisanih u drugom poglavlju. Za početak, postavit će se zahtjevi koje razvijeno programsko rješenje mora ispuniti, a zatim će se ukratko objasniti platforme, alate i tehnologije korištene tijekom razvoja programskog rješenja. Iz ovog poglavlja valja spomenuti *Android Architecture Components* skup biblioteka koje su direktno vezane uz arhitekturu *Model – View – ViewModel* koja je korištena tijekom razvoja programskog rješenja. Nakon upoznavanja s korištenim tehnologijama proći će se kroz najvažnije dijelove aplikacije te opisati kako ti dijelovi rade te koje mogućnosti aplikacija pruža krajnjem korisniku. Nakon upoznavanja s aplikacijom, prijeći će se na glavni dio ovog rada u kojem će se definirati što je to kvaliteta programskog rješenja, koja se dva pristupa koriste za utvrđivanje kvalitete, objasnit će se alati koji se koriste za ispitivanje kvalitete programskog rješenja te pružiti uvid u kvalitetu razvijenog programskog rješenja korištenjem upravo alata koji su spomenuti.

U zadnjem poglavlju ovog rada dat će se kratak zaključak na temu osiguranja kvalitete programske podrške, koja je namjera bila tijekom razvijanja programske podrške. Zatim će se osvrnuti na ključna ostvarenja i kako su postignuta u trećem poglavlju rada, navesti prednosti i mane odabranog pristupa te za kraju dati smjernice za budući rad i moguća unaprjeđenja sustava.

2. OSIGURANJE KVALITETE PROGRAMSKE PODRŠKE NA ANDROID PLATFORMI

Razvoj programske podrške nije jednostavan zadatak. Kako bi se osigurali minimalni troškovi razvoja treba, uz dobre programere, dizajnere i ostale članove tima, odabrati i odgovarajuću metodologiju razvoja. Metodologija razvoja predstavlja način pristupa razvoju programske podrške. Razlikuju se tradicionalne, metode orijentirane na tradicionalni pristup procesu razvoja te agilne metode koje su relativno nove, modernije, manje opsežne te su usredotočene na manje jedinice posla, a naglasak drže na vrijednostima i principima umjesto na procesu.

2.1. Metodologije razvoja

Metodologije je moguće podijeliti na one teže i opsežne koje traže puno vremena i discipline jer sadrže mnogo pravila, načina postupanja te dokumentacije, a s druge strane postoje lakše i manje opsežne metodologije. One sadrže tek nekoliko pravila i načina postupanja, stoga su lagane za slijeđenje.

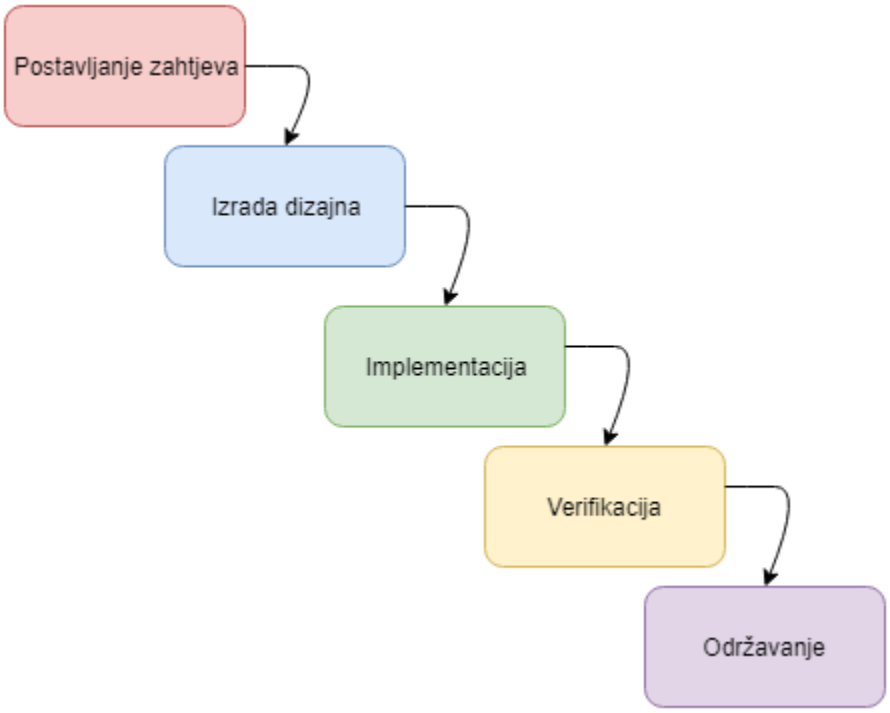
Dvije najpoznatije metode razvoja mobilnih aplikacija su:

1. Metoda vodopada (engl. *Waterfall methodology*)
   - Sekvencijalan proces razvoja, počinje od koncepta, a završava zatvaranjem projekta
2. Agilna metoda (engl. *Agile approach*)
   - Brz i iterativan proces, zadaci su mali i izvode se kratko vrijeme, lako se može prilagoditi promjenama u zahtjevima

2.1.1. Metoda vodopada

Metoda vodopada ili tradicionalna metoda [1] predstavlja linearan pristup razvoju programske podrške na duže vrijeme. Razvoj programske podrške dijeli se na nekoliko koraka koji se odvijaju sekvencijalno, svaki korak ima jasne zahtjeve koji moraju biti završeni prije nego li se može krenuti na sljedeći korak. Na slici 2.1 izrađenoj prema [2] nalazi se grafički prikaz spomenutih koraka tijekom razvoja. Jednom kada je neki korak (faza razvoja) izvršen i potvrđen više nema povratka na njega. Ukoliko dođe do promjene zahtjeva, a zahtjevi su već prihvaćeni i počeo je drugi korak, takav zahtjev se neće razmatrati u trenutnom razvoju odnosno inačici programske podrške. Prednost ovakve metoda je što je moguće jasno definirati rokove te je velika vjerojatnost da će projekt biti završen kada je to i planirano. Metoda vodopada ima nekoliko mana. Prvo je već spomenuto, nema povratka na prethodni korak, čak niti prilikom testiranja ako je

otkriveno nešto što nije pogodno teško je vratiti se u fazu razvoja i to izmijeniti osim ako nije riječ o teškoj pogrešci programa. Druga mana je što nakon zahtjeva nema puno interakcije s naručiteljima projekta te se često može dogoditi da projekt koji je isporučen, iako odgovara zadanim zahtjevima, ne odgovara naručitelju jer to nije ono što je on zamislio. Ovakav se problem može riješiti tako da se naručitelj redovito obavještava o stanju projekta nakon svakog koraka.



Sl. 2.1. Vodopad metoda razvoja

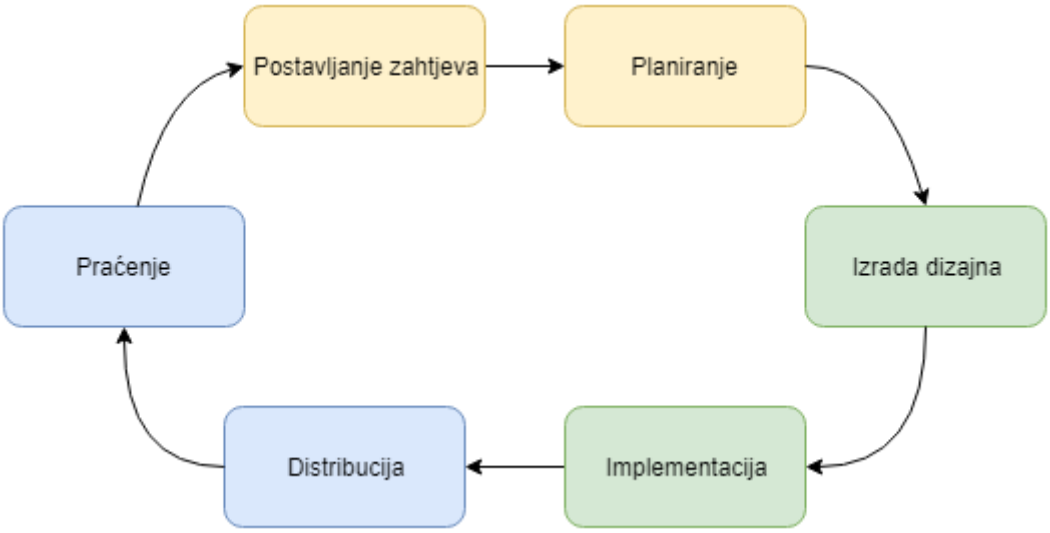
Metoda vodopad ima pet karakterističnih koraka razvoja:

1. Postavljanje zahtjeva
    - U dogovoru sa naručiteljem postavljaju se zahtjevi koje projekt treba zadovoljiti
2. Izrada dizajna
    - Dizajneri izrađuju dizajn aplikacije
3. Implementacija
    - Programeri implementiraju dizajn i razvijaju programsko rješenje.
4. Verifikacija
    - Testeri verificiraju da aplikacija zadovoljava zahtjeve iz prvog koraka.
5. Održavanje
    - Faza održavanja, popravljanje grešaka u programu, itd.

2.1.2. Agilna metoda

Agilne metodologije [3] pokazale su se kao efikasne i od velike pomoći tijekom razvijanja programske podrške. Fokusirane su na fleksibilnost, uključivanje kupca u razvoj, upravljanje

rizikom i konstantnu evaluaciju. Agilne metodologije daju najbolje rezultate jer je kupac uključen tijekom cijelog procesa razvoja te može intervenirati ukoliko vidi nešto što mu se ne sviđa. Ukoliko želi dodati neki novi zahtjev na proizvod, isti se zahtjev može planirati u nekoj od sljedećih faza izrade. Tehnika gdje je razvoj programske podrške podijeljen u više manjih dijelova koji se nazivaju *Scrum*, projektni menadžer dodijeljen je svakom *Scrum*-u, a takav se menadžer naziva *Scrum Master*. Na slici 2.2. napravljenoj prema [4] prikazan je blok dijagram agilne metodologije razvoja.



Sl. 2.2. Agilna metodologija razvoja

Razlikuju se tri glavne faze izrade:

1. Faza prije izrade
    - Uključuje planiranje, okupljanje tima, izradu arhitekture sustava i dizajn općeg modela sustava.
    - Prema slici 2.2., predstavlja prva dva koraka – postavljanje zahtjeva i planiranje
2. Faza izrade – *Sprint*
    - Uključuje izradu programske podrške, izradu dizajna, pregledavanje, testiranje i prilagodbu, najčešće traje 14 dana.
    - Prema slici 2.2. predstavlja druga dva koraka – izradu dizajna i implementaciju.
3. Faza poslije izrade
    - Uključuje konačno testiranje, izradu dokumentacije, implementaciju te završetak projekta.
    - Prema slici 2.2. predstavlja posljednja dva koraka – distribuciju i praćenje.

Postoje različite uloge i odgovornosti članova tima:

1. *Scrum master*

- Osoba koja vodi tim, brine o tome da se uklone sve prepreke te kako bi se pojedini sprint uspješno realizirao.

2. Vlasnik projekta
    - Osoba za koju se projekt izvodi, većinom je to naručitelj, ili osoba koju je on poslao da ga predstavlja, odgovoran je za prenijeti viziju *scrum* timu
3. *Scrum* tim
    - Skupina ljudi koja direktno sudjeluje u izvedbi projekta, programeri, dizajneri, testeri, itd. Odgovorni su za isporučivanje rezultata tijekom sprintova.
4. Dioničari / dionici (engl. *stakeholders*)
    - Svi koji imaju neke veze sa projektom – naručitelji, uprava, krajnji korisnici, itd.

2.2. Čist kod i oblikovni obrasci

U današnje vrijeme razlikuju se dva pristupa programiranju: programiranje i dobro programiranje. Programiranje je proces u kojem je bitno samo da napisani kod izvršava ono za što je predviđen, dok je kod dobrog programiranja, osim funkcionalnosti koda, od izuzetne važnosti kako taj kod izgleda.

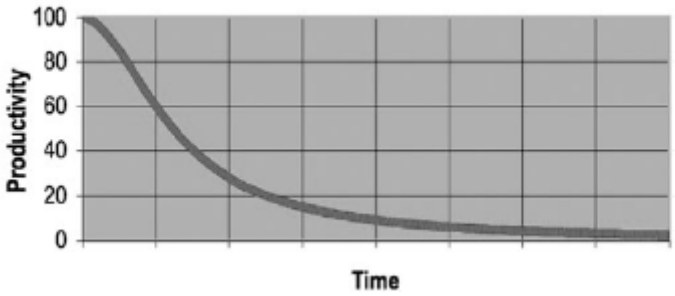
Uz definirana pravila kako kod treba izgledati kojih će se držati svi članovi tima koji rade na određenom projektu, posao uređivanja i održavanja koda će biti uvelike lakši. Situacija u kojoj bi svatko pisao kod onako kako želi, bez definiranih osnovnih pravila predstavljala bi kaotičnu situaciju. Kada bi programer analizirao kod koji on nije pisao, da bi nešto dodao ili popravio neki problem, prvo mora pročitati kod i shvatiti što se događa. Ukoliko ne postoji neki uniformni stil pisanja koda, programer će izgubiti puno vremena na dekodiranje napisanog.

Korištenje unaprijed provjerenih rješenja može uvelike skratiti proces razvoja jer pruža rješenja koja su poznata i provjerena od velikog broja programera. Kao velika prednost ističe se jednostavno održavanje, upravo zato što ih veliki broj ljudi koristi i upoznat je s njima.

2.2.1. Čist kod

Većina programera postavlja si pitanje koliko je zapravo važno da je kod koji napiše čist, odnosno čitljiv te dolaze do zaključka da većina vremena tijekom programiranja odlazi na pisanje koda što nije istina, pogotovo ako se radi o timskom projektu. [5, str.45] Primjer je programer koji mora promijeniti funkcionalnost neke metode, prvo odlazi u modul gdje se metoda nalazi, zatim je potrebno pregledati sve pozive metode koja se želi promijeniti i uvjeriti se da izmjena metode neće utjecati na ostatak koda. Čak i kada je riječ i minimalnim izmjenama potrebno je pročitati podosta starog koda, dolazi se do zaključka da se ne može mijenjati, odnosno dodavati kod ukoliko se ne može razumjeti značenje koda koji ga okružuje.

Loše napisan kod će dugoročno oduzeti više vremena, većina programera bila je u situaciji gdje je bilo potrebno što brže završiti neki projekt te su iz tog razloga zanemarili neke ključne osnove dobrog programiranja misleći da će urediti taj dio koda kada vrijeme bude dopustilo. Problem kod takvog pristupa je što u većini slučajeva nema dovoljno vremena da se taj dio koda uredi u skoroj budućnosti. Kada se nakon dužeg vremena programer koji je pisao taj kod vrati na njega kako bi nešto izmijenio ili čak uredio gubi puno vremena dok shvati što se zapravo događa, ako pak taj dio koda krene uređivati netko tko ga nije pisao gubitak vremena eksponencijalno je veći. Prema [5, str.35] produktivnost tima pada kako se nered u kodu povećava, kako bi povećali produktivnost menadžeri zapošljavaju više ljudi koji će raditi na projektu. Novi ljudi na projektu nisu upoznati sa projektom te se gubi vrijeme koje je potrebno na prilagodbu ljudi na novi projekt, zbog sve većeg pritiska za povećanjem produktivnosti stvara se sve više nereda u kodu, graf ovisnosti produktivnosti i vremena preuzet iz [5, str.35] prikazan je na slici 2.3.



Sl. 2.3. Graf produktivnosti u odnosu na vrijeme

Koliko god malo vremena bilo potrebno je ustrajati u tome da kod bude napisan po nekim osnovnim pravilima, kako bi se uštedjelo vrijeme u budućnosti. Na svoj kod programer treba gledati kao neki potpis koji ostavlja iza sebe, programer je autor koda kojeg napiše i treba mu biti u cilju napraviti što ljepše djelo kojem će se kolege programeri diviti, a ne ismijavati. Jedno od nepisanih pravila programiranja kaže da kod na kojem je programer radio nakon obavljenog posla ne smije biti u gorem stanju nego što ga je programer zatekao.

Karakteristike čistog koda su [6]:

1. Elegantnost
    - Napisani kod treba biti ugodan za čitanje
    - Čitanje takvog koda treba izazvati jednak osjećaj kao što bi to izazvao dobro izgrađen automobil, kuća, lijepo dizajniran stan, itd.
2. Fokusiranost
    - Svaka funkcija, klasa i modul drže se pravila jedinstvene odgovornosti, na njih ne utječu okolne klase, funkcije, odnosno moduli.

3. Pažnja
    - Prilikom pisanja čistog koda potrebno je osim vremena dati i pažnju, čistom kodu netko je dao svoje vrijeme i pažnju čak i na najmanje detalje, odnosno netko je brinuo za napisani kod.
4. Ne sadrži duplikate
    - Ukoliko se jednak kod planira koristi na više mjesta stvaraju se odgovarajuće metode kako bi pojednostavili održavanje takvog koda.
5. Relativno malen broj entiteta
    - Koristiti što manje klasa, metoda, funkcija, itd.

Čist kod piše se upotrebnom raznih pravila, kao što su davanje imena varijabli, metodi ili klasi iz kojeg je odmah vidljivo zašto postoji, što radi i kako je korištena. Ukoliko ime zahtjeva dodatan komentar kako bi se njime objasnila upotreba tada to ime nije dobro definirano. Kao primjer može biti varijabla koja predstavlja broj elemenata liste koji zadovoljavaju neki uvjet, primjer loše imenovanih varijabli za koje je potrebno dodatno komentirati prikazan je u programskom kodu 2.1.

```
int[] list = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int d = 0; // Counter for numbers greater than 4
for (int i : list) {
        if (i > 4) {
            d++;
            System.out.println(i);
        }
}
```

Programski kod 2.1. Primjer loše imenovanih varijabli

Primjer dobro imenovanih varijabli, koje nije potrebno dodatno komentirati vidljiv je u programskom kodu 2.2.

```
int[] listOfNumbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int count = 0;
for (int number : listOfNumbers) {
        if (number > 4) {
            count++;
            System.out.println(number);
        }
}
```

Programski kod 2.2. Primjer dobro imenovanih varijabli

Imena metoda trebala bi sadržavati glagol koji će opisivati što ta metoda radi, neki od primjera:

1. getUser()
    - Dohvaća korisnika.
2. deleteUser()

- Briše korisnika.

3. saveUser()
    - Sprema korisnika.

Jednom kada je odlučen glagol koji će se koristiti za definiranje neke radnje (npr. „*get*“ za dohvaćanje) taj glagol treba se koristi kroz cijeli projekt, korištenje različitih varijanti glagola koji predstavlja istu stvar može dovesti do gubljenja vremena tijekom traženja odgovarajuće metode.

Za razliku od metoda ime klase ne bi trebalo sadržavati glagol, nego imenicu koja opisuje klasu, odnosno objekt, primjeri:

1. Korisnik
2. Knjiga
3. Komentar
4. Osoba

Prilikom pisanja čistog koda veliku pozornost treba posvetiti metodama, već je spomenuto da ime metode mora objašnjavati kakvu radnju metoda obavlja te da treba težiti jedinstvenoj odgovornosti, odnosno što manjem utjecaju okoline na kod unutar metode. Uz prethodno spomenuta svojstva valja spomenuti kako bi funkcije trebale biti što manje, uz ljepši izgled koda ovo svojstvo također će pomoći u nazivanju metoda, primjer je prikazan u programskom kodu 2.3.

```
private handleResponse(Results results) {
        if (results.isEmpty() {
                noResult.setVisibility(View.VISIBLE);
                api.getResults();
        } else {
                noResult.setVisibility(View.GONE);
                if (adapter == null) {
                        adapter = new Adapter(results);
                        listView.setAdapter(adapter);
                } else {
                        adapter.notifyDataSetChanged();
                }
        }
}
```

Programski kod 2.3. Primjer loše napisanog koda

Dan je jednostavan primjer, ali je blok koda velik i treba ga smanjiti koristeći preporuke pisanja čistog koda prema programskom kodu 2.4.

```
private handleResponse(Results results) {
        if (results.isEmpty() {
                showNoResults();
        } else {
                showResults(results);
        }
}
private void showNoResults() {
        noResult.setVisibility(View.VISIBLE);
        api.getResults();
}
private void showResults(Results results) {
        noResult.setVisibility(View.GONE);
        if (adapter == null) {
                adapter = new Adapter(results);
                listView.setAdapter(adapter);
        } else {
                adapter.notifyDataSetChanged();
        }
}
```

Programski kod 2.4. Primjer dobro napisanog koda

Kao što je vidljivo iz primjera metode koje su dane u kod samim imenom objašnjavaju koja je njihova zadaća te ja iznimno lako shvatiti što se događa u kojem koraku, osim čitljivosti ovakav pristup će također pomoći prilikom održavanja koda.

2.2.2. Oblikovni obrasci

Oblikovni obrasci [7] su višekratna rješenja za najčešće probleme koji se javljaju tijekom razvoja programske podrške, mogu ubrzati proces razvoja programske podrške pružajući dokazana rješenja za česte probleme s kojima se programeri susreću. Oblikovni se obrasci dijele na tri osnovne grupe:

1. Obrasci stvaranja (engl. *Creational Design Pattern*)
    - Rješenja za stvaranje klasa, objekata.
2. Strukturni obrasci (engl. *Structural Design Pattern*)
    - Organizacija klasa, objekata.
3. Obrasci ponašanja (engl. *Behavioral Design Pattern*)
    - Pružaju načine na koje objekti i klase mogu međusobno komunicirati.

Obrasci stvaranja

Obrasci stvaranja [8] predstavljaju rješenja za stvaranje klasa i objekata, odvajaju proces instanciranja od ostatka koda. Neki od najkorištenijih obrazaca tijekom razvoja programske podrške za Android su:

1. Graditelj (engl. *Builder*)
2. Tvornica (engl. *Factory*)
3. Jedinstveni objekt (engl. *Singleton*)

Graditelj

Upotrebom Graditelja [9, str.10] pojednostavljuje se proces kreiranja objekta na čist i čitljiv način. Graditelj se koristi kada objekt ima veliki broj parametara koji nisu obavezni, umjesto kreiranja više instanca konstruktora sa različitim parametrima koristi se Graditelj koji na elegantni način rješava takav problem jednostavnim vezanjem statičkih metoda za postavljanje određenog parametra.

Primjer ovakvog obrasca je Graditelj koji se koristi za kreiranje dijaloga prikazan je u programskom kodu 2.5.

```
new AlertDialog.Builder(this)
        .setTitle(„Title“)
        .setMessage(„Message“)
        .setPositiveButton(„Ok“, null)
        .create();
```

Programski kod 2.5. Primjer obrasca stvaranja graditelj

Tvornica

Tvornica se koristi kada postoji osnovna klasa (engl. *Super class*) sa više podklasa te ovisno o zahtjevu potrebno je instancirati odgovarajuću podklasu. Tvornica prebacuje odgovornost kreiranja instance klase sa klijenta na sebe. Osnovu klase može predstavljati sučelje (engl. *Interface*), apstraktna klasa ili normalna klasa, podklase nasljeđuju osnovnu klasu te implementiraju potrebne metode. Kada su kreirane osnovna klasa te njezine podklase može se napisati Tvornica klasu (engl. *Factory class*), koja će na temelju parametara kreirati instancu jedne od podklasa.

Primjeri Tvornica su metode *getInstance()* korištena za instanciranje kalendara za različite vremenske zone i *valueOf()* u klasama kao što su Boolean, Integer, itd.

Jedinstveni objekt

Obrazac jedinstveni objekt se koristi kada je sigurno da nema potrebe za više od jedne instance klase [9, str.17]. Primjer je klasa odgovorna za rad s bazom podataka.

1. Obrazac kreiranja jedinstvenog objekta pruža sljedeće:
2. Osigurava da je samo jedna instanca klase kreirana
3. Pruža globalnu točku pristupa objektu

4. Omogućavanje generiranja više instanca u budućnosti neće utjecati na klijente

Kreiranje jedinstvenog objekta je poprilično jednostavno te se sastoji od kreiranja klase čiji je konstruktor privatan te statičke metode koja vraća instancu objekta. Primjer je prikazan u programskom kodu 2.6.

```
public class BasicSingleton {
        private static BasicSingleton INSTANCE = null;
        private BasicSingleton() {} // disable access to class constructor outside of this class
        public static BasicSingleton getInstance() {
                if  (INSTANCE == null)  {
                        INSTANCE = new BasicSingleton();
                }
                return INSTANCE;
        }
}
```

Programski kod 2.6. Primjer obrasca stvaranja jedinstveni objekt

Strukturni obrasci

Strukturni obrasci dijele se na Strukturne klasne obrasce i Strukturne objektne obrasce, a opisuju načine sastavljanja objekta i klasa radi stvaranja većih struktura.

Neki od strukturnih obrazaca su:

1. Prilagodnik (engl. *Adapter*)
2. Kompozit (engl. *Composite*)
3. Fasada (engl. *Facade*)

Prilagodnik

Prilagodnik [10] se koristi kada postoje dva različita sučelja koja je potrebno povezati, ne mogu se mijenjati, pa se kreira most (engl. *Bridge*) između njih. Dijele se na Prilagodnike za klase i Prilagodnike za objekte, prvi iskorištava svojstvo nasljeđivanja i može prilagođavati samo klase, odnosno ne može se koristi za sučelja, dok drugi kombinira željena svojstva i kreira novu cjelinu, stoga se može koristi za klase i za sučelja. Omogućava da dvije klase ili sučelja međusobno funkcioniraju čak i kada po definiciji nemaju tu mogućnost. Jedan od najčešće korištenih Prilagodnika u razvoju mobilnih aplikacija je Prilagodnik koji oblikuje model u pogled.

Kompozit

Kompozit [11] se koristi kada je potrebno tretirati grupu objekata na sličan način kao jedan objekt. Objekti se slažu u hijerarhiji stabla, a od svaki se čvora može zatražiti da obavi neki zadatak. Sastoji se od Komponente (engl. *Component*) i Lista (engl. *Leaf*), Kompozit može ispod sebe imati druge objekte, dok List nema objekte ispod sebe. Komponenta predstavlja sučelje kojim

klijent vrši interakciju s objektima iz strukture Kompozit. Kada klijent zahtjeva Kompozit zahtjev se predaje od Lista koji obavlja traženu operaciju, a ukoliko klijent zahtjeva List tada mu se on predaje direktno.

Fasada

Fasada[12] predstavlja zajedničko sučelje za upravljanje sučeljima podsustava, definira sučelje više razine kako bi se pojednostavilo korištenje podsustava. Pojednostavljuje pristup kompleksnom sučelju, ali i dalje omogućava kompletan pristup svim metodama podsustava. Ponaša se kao među sloj između klijenta i klase te na taj način razdvaja implementaciju klijenta od podsustava, kao još jedna prednost među sloja javlja se potreba da klijent komunicira samo sa jednim sučeljem umjesto sa većom količinom manjih kompleksnih sučelja.

Obrasci ponašanja

Obrasci ponašanja pomažu definirati komunikaciju između objekata te rješavaju probleme vezane uz algoritme i dodjeljivanje odgovornosti među objektima. Dijele se na obrasce zadužene za ponašanje objekata i obrasce zadužene za ponašanje klasa.

Neki od obrazaca ponašanja su:

1. Naredba (engl. *Command*)
2. Promatrač (engl. *Observer*)
3. Strategija (engl. *Strategy*)

Naredba

Naredba [13] je obrazac koji se koristi kako bi se enkapsulirao zahtjev u objekt, kako bi se omogućilo lakše rukovanje zahtjevom, moguće je stvoriti listu čekanja prema kojoj će se zahtjevi izvršavati. Ovakav obrazac omogućava pohranjivanje promjena stanja u sustavu, pohranjena stanja koriste se u slučaju kvara na sustavu kako bi se isti vratio u zadnje ispravno stanje. Naredba je objekt, pa je stoga jednostavno dodavanje novih Naredbi jednostavnim nasljeđivanjem i proširivanjem.

Promatrač

Promatrač [14] se koristi kada je potrebno stvoriti vezu jedan prema više te kada se želi obavijestiti druge objekte o promjenama ili akcijama glavnog objekta. Glavni objekt naziva se subjekt, dok su ostali objekti promatrači, glavni objekt može imati više promatrača. Subjektu nije bitno kome šalje informacije, on šalje istu informaciju svim promatračima, što predstavlja prednost

jer programer ima minimalan posao kako bi dodao promatrača, ali i manu ukoliko promatrač primi informaciju koja nije namijenjena njemu.

Strategija

Strategija [15] je oblikovni obrazac koji se koristi kada postoji više algoritama za pojedine zadatke, a klijent odlučuje o stvarnoj implementaciji tijekom izvođenja (engl. *Runtime*). Korištenjem ovog obrasca jedna klasa može imati različita ponašanja, ovakva politika pruža alternativu stvaranju podklasa, tako klijenti imaju mogućnost biranja algoritma za izvođenje pojedinog zadatka, a kao mana mora se spomenuti da u takvom slučaju klijent mora biti upoznat sa dostupnim algoritmima, odnosno strategijama.

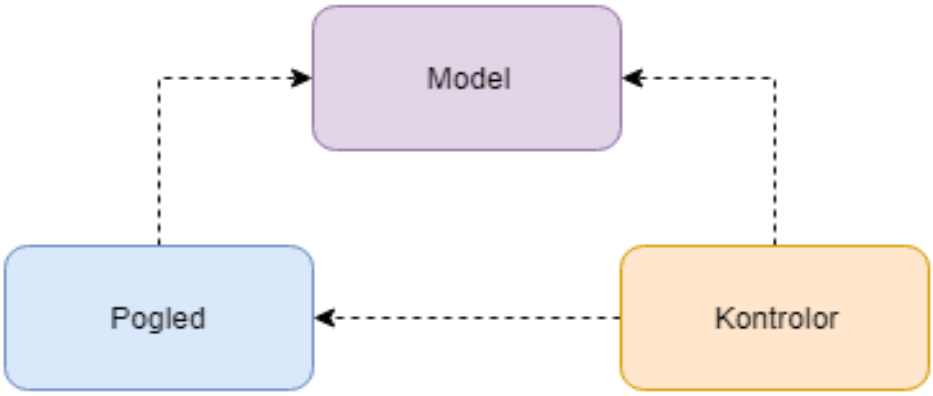
2.2.3. Arhitektura mobilne aplikacije

Za razliku od oblikovnih obrazaca koje je moguće lako izmijeniti tijekom razvijanja programske podrške, arhitektura predstavlja temelje aplikacije te je potrebno odlučiti o obrascu prije nego li počne razvoj. Arhitektura se naravno može promijeniti i tijekom razvoja, ali se ne preporuča. Korištenje provjerene arhitekture donosi mnoge prednosti kao što su testabilnost i jednostavnost pojedinih klasa, odnosno potiče jedinstvenu odgovornost, što olakšava posao programeru kada radi izmjene na takvom dijelu koda. Detaljnije će biti opisane neke od najčešćih arhitektura korištenih tijekom razvoja aplikacija za Android operacijski sustav [16].

Model – pogled – kontrolor

Model – pogled – kontroler (engl. *Model – View – Controller*, MVC) [17] sastoji se od tri osnovna dijela. Prvi je Model koji je odgovoran za opskrbu podacima, upravljanje bazom podataka, mrežnim zahtjevima, itd. Nakon modela dolazi Pogled (engl. *View*) koji predstavlja grafički prikaz podataka iz Modela, a najvažniji dio je Kontrolor (engl. *Controller*) koji sadrži svu bitnu logiku potrebnu za upravljanje zahtjevima korisnika, kao što su upravljanje podacima unutar Modela i obavještavanje Pogleda kada dođe do promjena unutar samog Modela.

Prema slici 2.4. izrađenoj prema [18] vidi se da Kontrolor i Pogled ovise o Modelu, Pogled je odgovoran da podatke iz Modela prikaže korisniku, dok je Kontrolor odgovoran da upravlja tim podacima, odnosno da ih izmjenjuje po potrebi.



Sl. 2.4. MVC arhitektura

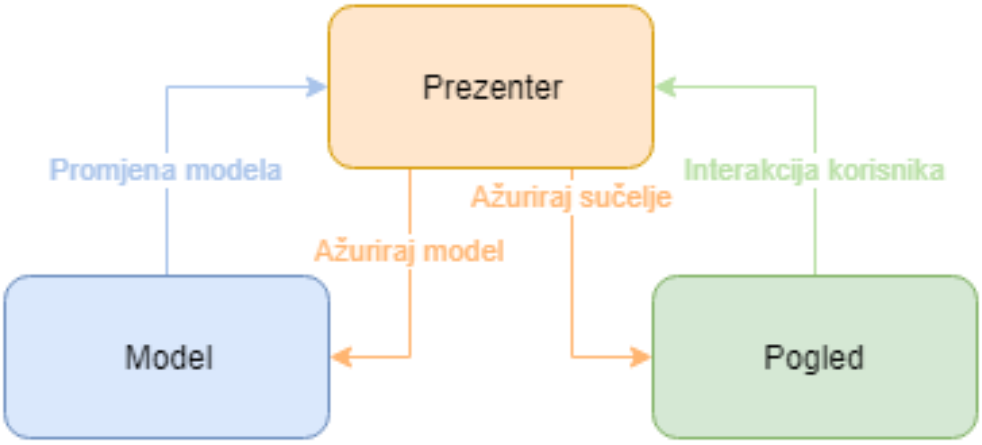
Razlikuju se pasivni i aktivni Model, kod prvog samo Kontrolor ima pravo mijenjati podatke unutar Modela te nakon promjene obavještava Pogled kako je došlo do promjene unutar Modela. Drugi primjer je aktivni Model, gdje Kontrolor nije jedina klasa koja upravlja podacima unutar Modela, stoga je potrebno dodati Promatrače koji će pratiti promjene unutar Modela te obavijestiti Pogled da je došlo do promjene.

Problem ove arhitekture je podjela odgovornosti, Pogled i Kontrolor imaju pristup Modelu i postavlja se pitanje zašto se uopće koristi Kontrolor ako Pogled može direktno komunicirati sa Modelom. Drugi je problem testabilnost, Pogled i Model imaju direktnu referencu jedan prema drugome što značajno otežava pisanje testova.

Model – pogled - prezenter

Model – pogled - prezenter (engl. *Model – View – Presenter*, MVP) [19] jedan je od najčešće korištenih arhitektura prilikom razvoja aplikacija na Android platformi, a njegova arhitektura dijeli Model i Pogled s MVC arhitekturom. Umjesto Kontrolora uvodi Prezenter (engl. *Presenter*), uloga Modela i Pogleda gotovo je jednaka kao i kod MVC arhitekture, ali Prezenter se razlikuje od Kontrolora. Kod MVP arhitekture Prezenter služi kao među sloj između Pogleda i Modela, svaki Pogled ima svoj Prezenter i svaki Prezenter ima Pogled, veza je jedan na jedan, međusobno komuniciraju pomoću sučelja (engl. *Interfaces*).

Prema slici 2.5. izrađenoj prema [20] vidi se da Pogled obavještava Prezenter o interakciji korisnika, Prezenter zatim obavlja potrebne radnje ovisno tipu interakcije te nakon završetka radnje obavještava Pogled ukoliko je došlo do promjene prikaza.



Sl. 2.5. MVP arhitektura

Prednost ovakve arhitekture je što Pogled nikada ne komunicira direktno sa Modelom, uloga Prezentera je predstavljati posrednika između Pogleda i Modela, u ovakvoj arhitekturi Pogled ima trivijalne metode za prikaz i slušanje korisnikovih zahtjeva. Za razliku od MVC arhitekture ovdje je jasno vidljivo da je odgovornost podijeljena i svaki dio MVP strukture se da lako samostalno testirati.

Model – pogled - ViewModel

Model – pogled – *ViewModel* (engl. *Model – View – ViewModel,* MVVM) [21] rješava problem koji se javlja kod MVP arhitekture, naime ako Prezenter pruža podatke, a Pogled ih preuzima i prikazuje, zašto bi pružatelj podataka trebao znati tko će te podatke „trošiti“. Prezenter kao pružatelj ne bi trebao brinuti o tome tko će podatke koristiti, on bi jednostavno trebao pružati podatke, a tko god želi može ih koristiti. Upravo je to princip na kojem radi MVVM arhitektura. MVVM kao i MVP dijeli Model i Pogled čija se implementacija u obje arhitekture gotovo i ne razlikuje.

Prezenter je zamijenjen *ViewModelom,* koji nema vezu jedan naprema jedan prema Pogledu kao što je to imao Prezenter, već se jedan *ViewModel* može koristi u više pogleda, a također jedan Pogled može imati više *ViewModela*. U MVP arhitekturi Prezenter je eksplicitno govorio Pogledu što treba prikazati, dok *ViewModel* pruža tok događaja na koji se Pogledi mogu prijaviti i slušati. *ViewModelu* nije bitno tko sluša on odašilje svima, a velika prednost je što *ViewModel* na drži referencu na Pogled kao što je to slučaj kod Prezentera, samim time smanjuje se broj potrebnih klasa u odnosu na MVP arhitekturu jer se izbacuju nepotrebna sučelja.

Prema slici 2.6. izrađenoj prema [22] vidi se da kao i kod MVP arhitekture Pogled obavještava *ViewModel* o različitim akcijama korisnika, *ViewModel* tada komunicira sa Modelom kako bi ispunio zahtjeve korisnika te nakon odgovora Modela emitira novi događaj Pogledu ili Pogledima. Iako *ViewModel* nema referencu na Pogled, Pogled zahtjeva referencu na *ViewModel*

čime je riješen problem koji je spomenut ranije, pružatelj podataka ne treba znati tko će podatke koristi, dok bi korisnik podataka trebao znati tko pruža podatke, a upravo takav je odnos Pogleda i *ViewModela*.

Sl. 2.6. MVVM arhitektura

2.3. Testiranje koda

Testiranje programskog rješenja predstavlja aktivnost u kojoj se provjerava odgovaraju li stvarni rezultati očekivanim rezultatima te se osigurava programsko rješenje bez mana. Testiranje programskog rješenja pomaže pronaći pogreške, rupe ili zahtjeve koji nisu u skladu sa zahtjevima koji su zadani prije razvoja, testiranje se može izvoditi ručno ili automatski, a najčešće metode testiranja su [23]:

1. Testiranje metodom crna kutija (engl. *Black-Box testing*)
2. Testiranje metodom bijela kutija (engl. *White-box testing*)
3. Nasumično testiranje (engl. *Random Testing/Monkey Testing*)

2.3.1. Metode testiranja programskog rješenja

Testiranje metodom crna kutija

Testiranje crne kutije predstavlja testiranje ponašanja. Osoba koja testira programsko rješenje ne mora biti upoznata sa samom strukturom rješenja, nego testira odziv sustava na dane ulazne varijable. Metoda se naziva testiranje crne kutije jer je to upravo ono što osoba koja testira vidi, odnosno ne vidi na koji način se nešto izvršava, bitno je samo da se izvrši ono što treba. Osoba koja testira programsko rješenje upoznata je sa zahtjevima sustava te se stavlja u ulogu korisnika i provjerava radi li sustav kako je zamišljeno. Testiranje crne kutije može se podijeliti na funkcionalno i nefunkcionalno testiranje, iako se većinom svodi na funkcionalno. Funkcionalno testiranje uključuje testiranje aplikacije na zadane zahtjeve, uključuje razne testove kojima osigurava da se svaki dio programskog rješenja ponaša upravo onako kako je to zamislio projektni menadžer. Nefunkcionalno testiranje uključuje tipove testova koji se fokusiraju na operacijske aspekte programskog rješenja kao što su sigurnost, brzina, kompatibilnost, itd. Prednosti testiranja crne kutije:

1. Testiranje se izvodi iz perspektive korisnika

2. Osoba koja testira ne mora znati programski jezik kojim je programsko rješenje razvijeno
3. Testiranje može izvršiti neovisno tijelo
4. Testovi se mogu dizajnirati odmah nakon specifikacija

Nedostaci testiranja crne kutije:

1. Nije moguće testirati kompletnu aplikaciju
2. Bez jasnih specifikacija teško je dizajnirati testove

Testiranje metodom bijela kutija

Testiranje bijele kutije naziva se još i testiranje strukture, a predstavlja testiranje u kojem je struktura ili komponenta koji se testira poznata osobi koja testira, za razliku od testiranja crne kutije gdje se testirao samo odziv sustava. Naziva se testiranje bijele kutije jer je osobi koja testira sve jasno vidljivo, odnosno testira se unutarnja struktura programskog rješenja.

Prednosti testiranja bijele kutije:

1. Testiranje se može izvršiti i prije završetka grafičkog sučelja
2. Temeljito testiranje aplikacije, mogućnost testiranja svih mogućih ruta

Nedostaci testiranja bijele kutije:

1. Testovi mogu biti iznimno komplicirani stoga je potrebno veliko znanje u programiranju i implementiranom sustavu
2. Održavanje testnih skripti može biti teško ukoliko se implementacija često mijenja

Nasumično testiranje

Nasumično testiranje je testiranje bez planiranja i dokumentacije, testovi se izvode neformalno, nasumično i bez očekivanih rezultata. Osoba koja testira improvizira te proizvoljno bira testne koraka. Metoda se naziva i majmunsko testiranje (engl. *Monkey testing*) upravo iz razloga što ovakvo testiranje oponaša majmuna dok pleše i „udara“ po tipkovnici. Iako je mane pronađene tijekom ovakvog testiranja teže reproducirati, pomaže pronaći mane koje bi bilo nemoguće pronaći oponašanjem normalnog korisničkog ponašanja.

2.3.2. Tipovi testiranja programskog rješenja

Neki od tipova testiranja korištenih u spomenutim metodama testiranja su:

1. Testiranje prihvatljivosti (engl. *Acceptance testing*)

2. Automatsko testiranje (engl. *Automated testing*)
3. Regresijsko testiranje (engl. *Regression testing*)
4. Brzinsko testiranje (engl. *Smoke testing*)

Testiranje prihvatljivosti

Testiranje prihvatljivosti testira stvarne zahtjeve aplikacije, odnosno je li mjera u kojoj su zadani zahtjevi ispunjeni prihvatljiva, dijeli se na testiranje funkcionalnosti, testiranje jednostavnosti korištenja ili oboje. Formalno testiranje vodi računa o korisnikovim potrebama i već spomenutim zahtjevima. Prije razvoja programskog rješenja postavljaju se kriteriji koji moraju biti ostvareni kako bi se implementacija smatrala uspješnom, a upravo se na takve kriterije testira aplikacija tijekom testiranja prihvatljivosti. Testiranje prihvatljivosti predstavlja zadnju stanicu testiranja programskog rješenja, a obično se izvodi metodom testiranja crne kutije, pošto testiranje nerijetko nema striktno definirane procese i slučajeve koje treba provjeriti koristi se i nasumična metoda testiranja.

Automatsko testiranje

Ručno testiranje zahtjeva vrijeme i napor pojedinca kako bi osiguralo ispravnost programskog rješenja. Uz samo testiranje ručni testeri moraju snimiti svoja otkrića kako bi se eventualne mane mogle uspješno popraviti. Automatsko testiranje s druge strane posao testera zamjenjuje sa računalom koje pokreće testne skripte. Iako automatsko testiranje [24] zahtjeva dodatno vrijeme za pisanje testnih skripta i njihovo održavanje uslijed promjena na sustavu, jednom napisani testovi mogu se pokrenuti nebrojen broj puta i izvode se brže nego što bi to mogao izvesti čovjek tehnikom ručnog testiranja. Osim simuliranja korisnika i testiranje grafičkog sučelja, automatsko testiranje također može provjeravati strukturu aplikacije, odnosno bazu podataka i razne Internet usluge.

Regresijsko testiranje

Regresijsko testiranje predstavlja izvođenje automatskih testova nakon promjena na sustavu kako bi se uvjerili da uvedene promjene nisu utjecali na ostatak koda, odnosno da sve radi onako kako je radilo prije promjene. Regresijskim testiranjem osigurava se da neće doći do nazatka aplikacije. Korištenjem Agilne metodologije razvoja gdje se konstantno dodaju nove funkcionalnosti programskom kodu bitno je osigurati da nove funkcionalnosti nisu prouzročile greške kod onih starijih. Većina automatskih testova su regresijski testovi upravo iz razloga što ih je vrlo jednostavno pokrenuti jednom kada su napisani te se često pokreću upravo nakon velikih promjena kako bi se potvrdila ispravnost svih funkcionalnosti za koje je napisan test.

Brzinsko testiranje

Brzinsko testiranje je tip testiranja koje se izvodi odmah nakon izgradnje programskog rješenja kako bi se provjerili ključni elementi napisanog programa. Izvodi se prije svih detaljnijih testova kao što su funkcionalni ili regresijski testovi. Cilj je odmah riješiti probleme do kojih je došlo nekom promjenom kako tim za osiguranje kvalitete ne bi gubio vrijeme na testiranje takve verzije programskog rješenja. Ime je potpuno opisno što znači da se brzinski testovi uistinu obavljaju brzo, ne ide se u dubinu s testiranjem već se testiraju samo ključne funkcionalnosti programskog rješenja. Rezultati brzinskog testiranja koriste se kako bi odlučili hoće li se nastaviti detaljnije testiranje programskog rješenja. Ukoliko programsko rješenje prođe brzinski test prelazi se na detaljnije testiranje, ukoliko ne prođe traži se nova verzija čime se štedi vrijeme jer se isto ne gubi na testiranje programskog rješenja za koje se zna da nije ispravno. Brzinsko testiranje izvodi se ručno ili pomoću automatskih alata, ukoliko do novih verzija dolazi često tada bi se brzinsko testiranje trebalo izvršavati automatski kako bi ubrzali proces razvoja.

2.3.3. Uloga tima za osiguranje kvalitete

Tim za osiguranje kvalitete [25] (engl. *Quality Assurance team*) odgovoran je za olakšavanje testnih procedura. U suradnji sa programerima otkrivaju pogreške i razlike u odnosu na postavljane zahtjeve, osiguravajući ispravnost programskog rješenja prije distribucije. Ulaganje u osiguranje kvalitete nije samo korisno nego je i neophodno zbog što veće konkurencije, stoga je od velike važnosti osigurati kvalitetu kako bi se zadržali trenutn i privukli novi korisnici.

Prednosti uključivanja tima za osiguranje kvalitete u projekt:

1. Pomaže ispunjavaju zahtjeva i očekivanja klijenta
2. Izgrađuje povjerenje kod klijenata te pomaže u pobjeđivanju konkurenata na duže staze
3. Štedi novac detektiranjem i popravljanjem pogrešaka u ranom periodu razvoja
4. Pomaže u postavljaju i održavanju visoke kvalitete, pomiče fokus sa detektiranje na prevenciju problema

Tim za osiguranje kvalitete ne predstavlja samo skupinu testera već osim testiranja aplikacije na pogreške osigurava ispunjenje svih postavljenih zahtjeva, također pomažu tijekom kreiranja intuitivnog korisničkog sučelja. Članovi tima za osiguranje kvalitete izvode ručne i automatske testove, ručni testovi uglavnom se koriste za provjeru grafičkog sučelja nakon čitanja postavljenih zahtjeva te ne zahtijevaju veliko znanje člana tima o samom programiranju i aplikaciji koja se testira. Automatsko testiranje izvodi se pisanjem i izvođenjem testnih skripti. Pisanje testnih skripta oduzima puno vremena, ali moguće ih je pokretati nebrojen broj puta pri čemu se

izbacuje ljudsku grešku, odnosno izvršava sve korake svaki puta na jednak način, dok čovjek može preskočiti neke korake zbog umora ili nezainteresiranosti.

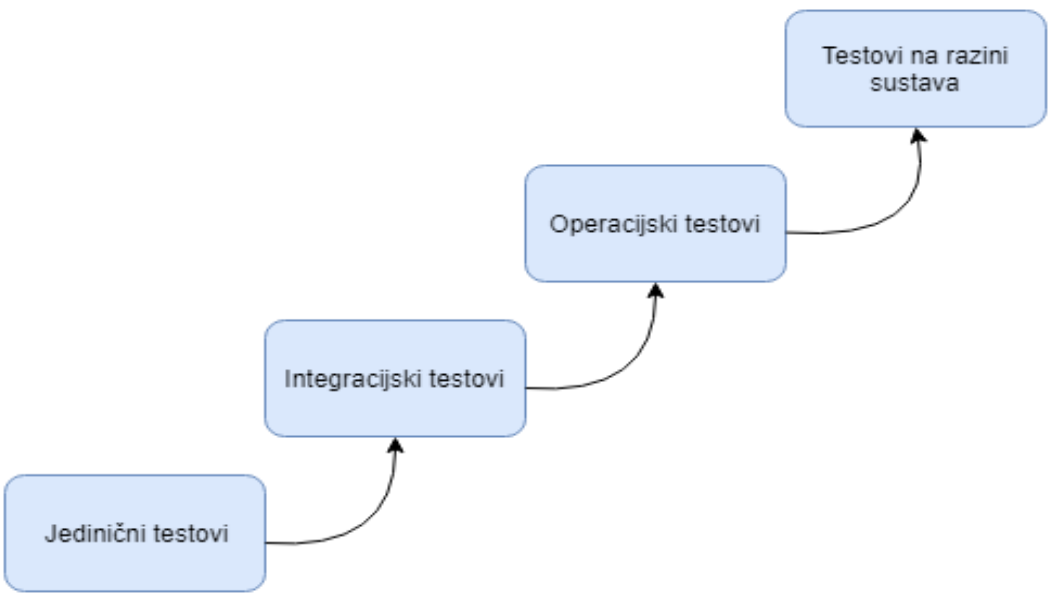
2.3.4. Testiranje Android aplikacije

Kvaliteta Android aplikacije jedan je od ključnih faktora za uspjeh iste. Testiranje aplikacije pomaže prilikom otkrivanja raznih pogrešaka u samoj aplikaciji te također može mjeriti koliko je aplikacija precizna za onom za što je namijenjena. Android Studio [26] dizajniran je kako bi olakšao pisanje testova, a uz minimalne napore moguće je testirati dio koda. Testovi napisani u Android Studiju mogu se pokretati na lokalnom Java virtualnom stroju (engl. *Java Virtual Machine*, JVM) ili na samom uređaju.

Dobro testirana aplikacija mora sadržavati sljedeće testove:

1. Jedinični test (engl. *Unit test*)
   - Testira male dijelove koda
2. Test korisničkog sučelja (engl. *User Interface test*)
   - Testira prikaz korisniku
3. Integracijski test (engl. *Integration test*)
   - Testira komunikaciju između različitih klasa, modula, itd.
4. Operacijski test (engl. *Operational test*)
   - Testira aplikaciju prema zadanim zahtjevima
5. Test na razini sustava (engl. *System test*)
   - Testira cijelu aplikaciju u stvarnom okruženju

Na slici 2.7 izrađenoj prema [27] prikazan je blok dijagram strategije testiranja aplikacije.



Sl. 2.7. Ilustracija strategije testiranja aplikacije

Većina oblika testiranja Android aplikacija zasniva se na [28, str.170] JUnit (engl. *Java Unit*) testnom okviru namijenjenom za pisanje testova u Java programskom jeziku. Generirani kod testira radi li klasa koja se testira onako kako programer očekuje.

Razlikuju se dva tipa testova unutar Android aplikacije:

1. Testovi koji se vrte na Java Virtualnom Stroju
    - Koriste se za testiranje standardnih Java klasa koje ne zahtijevaju Android API
    - Nalaze se na lokaciji – „./src/test/java/“
2. Testovi koji zahtijevaju Android SDK
    - Koriste se za testiranje klasa koje koriste Android API, ovakvi testovi moraju se pokretati na Android uređaju koristeći Android JUnit proširenja.
    - Nalaze se na lokaciji – „./src/androidTest/java“

Slika 2.8. izrađena prema [29] prikazuje piramidu testova koje Android aplikacija treba sadržavati, Jedinični test je temeljni test te bi aplikacija trebala imati najviše Jediničnih testova, nakon njih slijede Integracijski te testovi korisničkog sučelja.

Sl. 2.8. Piramida testova

Jedinično testiranje

Jedinični testovi predstavljaju temelje strategije testiranja. Pokretanjem jediničnih testova lako se može uvjeriti da pojedine komponente aplikacije rade onako kako se to od njih i očekuje. Testiraju male dijelove koda, bilo da je riječ o klasi, metodi ili komponenti, a jedinični testovi se pišu kako bi se provjerilo da je logika testiranog koda dobro napisana, odnosno daje li rezultat kakav se očekuje.

Razlikuju se dva tipa jediničnih testova:

1. Lokalni jedinični testovi (engl. *Local unit tests*)
    - Namijenjeni testiranju logike koja nema interakciju s operacijskim sustavom
2. Instrumentacijski jedinični testovi (engl. *Instrumented unit tests*)
    - Testiraju logiku koja ima interakcije sa Android API-jem te se pokreću na Android uređaju

Primjer lokalnog jediničnog testa prikazan je u programskom kodu 2.7.

```
@Test
public void testAddition() {
        assertEquals(4, 2+2);
}
```

Programski kod 2.7. Primjer lokalnog jediničnog testa

Primjer instrumentacijskog jediničnog testa prikazan je u programskom kodu 2.8.

```
@Test
public void useAppContext() {
        Context context = InstrumentationRegistry.getTargetContext();
        assertEquals(„com.package.name“, context.getPackageName());
}
```

Programski kod 2.8. Primjer instrumentacijskog jediničnog testa

Testiranje korisničkog sučelja

Testiranje korisničkog sučelja važno je kako bi se osiguralo da se korisnici ne susretnu s neočekivanim situacijama te kako bi se osiguralo da se na ekranima korisnika prikazuje upravo ono što se treba prikazati. Najjednostavniji način testiranja korisničkog sučelja je ručno testiranje, ručno testiranje ima svojih prednosti i mana, kao najveća mana je potreba za ljudima koji će temeljito testirati aplikaciju. Za ponovnu izvedbu ručnog testa potrebno je više vremena i također je potreban čovjek koji bi ga izveo, prednost je da niti jedan okvir za testiranje neće moći tako dobro testirati kao što to može testirati čovjek. Nekada dio koda jednostavno nije moguće testirati pomoću automatskih alata za testiranje te je potrebno ručno testiranje.

Postoje razni testni okviri koji simuliraju interakciju korisnika, a najpoznatiji su *Espresso* i *UI Automator* koji se često koriste zajedno. Velika prednost testnih okvira je što se testovi vrte automatski, odnosno nije potreban čovjek, jednom kada su testovi napisani ponovno testiranje izvodi se vrlo brzo jednostavnim pokretanjem testnih skripta. Testiranje korisničkog sučelja sastoji se od niza naredbi koje naređuju računalu da upravlja programskom podrškom upravo na onakav način kako bi to učinio korisnik, potrebno je unijeti ulazne parametre te očekivani rezultat.

Testiranje korisničkog sučelja moguće je izvesti i snimanjem testa, ovakav način je vrlo jednostavan te se sastoji od jednostavnog pritiska na tipku „Snimaj“ (engl. *Record*), zatim se izvode radnje na uređaju koje će test simulirati, dodaju se provjere, itd. Nakon završetka snimanja testa isti se sprema i generira se kod čijim će se pokretanjem simulirati snimljeni sadržaj.

Integracijsko testiranje

Integracijsko testiranje predstavlja nadogradnju na jedinično testiranje, jedinični test izoliran je od vanjskih čimbenika te testira mali dio koda. Integracijski test je nešto veći od jediničnog testa te između ostalog testira međusobnu komunikaciju između više sučelja, uzima u obzir vanjske čimbenike kao što su mreža, baza podataka, vrijeme, niti izvršavanja, itd. Upravo zato što ovisi o vanjskim čimbenicima treba mu duže vrijeme kako bi se izvršio od klasičnog jediničnog testa. Koristi se za testiranje komponenata s kojima korisnici nemaju direktnog doticaja, kao što su Servisi (engl. *Service*) i Pružatelji sadržaja (engl. *Content Providers*).

Operacijsko testiranje

Operacijsko ili funkcionalno testiranje predstavlja visoku razinu testiranja na kompletnost i ispravnost zahtjeva koji su dani aplikaciji. Izrađuju se međusobnom suradnjom klijenata, testera i programera. Najpoznatiji alat za operacijsko testiranje Android aplikacija je *FitNesse*.

Testiranje na razini sustava

Testiranje na razini sustava (engl. *System testing*) [30] provodi se kako bi se uvjerilo da sustav u cjelini radi kako bi trebao, uključujući dizajn aplikacije, učinkovitost te kompatibilnost sa ciljanim uređajima.

Testovi na razini sustava uključuju sljedeće testove:

1. Test grafičkog sučelja (engl. *GUI tests*)
2. Test iskoristivosti (engl. *Usability tests*)
3. Test performansi (engl. *Performance tests*)
4. Test na stresne situacije (engl. *Stress test*)

Od navedenih testova kao najvažniji ističe se test performansi, korisnici žele da aplikacija bude brza i tečna. Test Performansi mjeri performanse pojedinih komponenata Android aplikacije, mogu se koristi alati kao što je *Traceview*, ovakav alat pomaže profilirati performanse testirane aplikacije.

2.4. Automatizacija testiranja i okviri za testiranje

Okviri za testiranje predstavljaju alate za pisanje i izvođenje automatskih testova, sadrže skup uputa za kreiranje i dizajniranje slučajeva za testiranje. Okvir za testiranje odgovoran je za

definiranje formata za izražavanje očekivanih rezultata, kreiranje mehanizma za upravljanje testiranom aplikacijom, izvođenje testova i pružanje izvješća o uspješnosti. Najpoznatiji okviri za testiranje Android aplikacija su Appium, Calabash, Espresso, UIAutomator i Robotium, a njihova svojstva prikazana su u tablici 2.1 izrađenoj prema [31].

Tablica 2.1. Svojstva najpoznatijih okvira za testiranje Android aplikacija

	APPIUM	CALABASH	ESPRESSO	UIAUTOMATOR	ROBOTIUM
ANDROID	✔	✔	✔	✔	✔
IOS	✔	✔	✖	✖	✖
WEB STRANICE	✔ (Android & iOS)	✔ (Android)	✖	Ograničen na x.y klikove	✔ (Android)
SKRIPTNI JEZICI	Gotovo svi	Ruby	Java	Java	Java
ALATI ZA IZRADU TESTOVA	Appium.app	CLI	Hierarhy viewer	UI Automator viewer	
PODRŽANE API VERZIJE	Sve	Sve	8, 10, 15 >=	16 >=	Sve
ZAJEDNICA	Aktivna	Prilično tiha	Google	Google	Suradnici

2.4.1. Appium

Appium [32] je testni okvir koji se koristi za testiranje nativnih, hibridnih te Web aplikacija za Android i iOS, koristi *JSONWireProtocol* kako bi komunicirao sa Android i iOS aplikacijama koristeći Seleniumov WebDriver. Velika prednost Appiuma je što se jedan test napisan za neku platformu može koristi na ostalim platformama, što uvelike smanjuje vrijeme kada se radi o projekt koji za primjer ima mobilnu aplikaciju na iOS i Android platformama, dovoljno je napisati testove za jednu platformu te će se isti moći iskoristiti za drugu.

Programski kod 2.9 prikazuje test za login korisnika preko Web-a, pažnja se usmjerava na prozor „WEBVIEW“, zatim se dohvaćaju pogledi za unos korisničkog imena i lozinke te se nakon uspješnog logina vrši povratak u Nativnu aplikaciju.

```
@Test
public void loginTest() {
        WebDriverWait wait = new WebDriverWait(driver, 10);
        driver.switchTo().window("WEBVIEW");
        WebElement userNameInput = driver.findElement(By.id("input_user_name"));
        wait.until(ExpectedConditions.visibilityOf(userNameInput));
        userNameInput.clear();
        userNameInput.sendKeys("android1@example.com");
        driver.findElement(By.name("password")).sendKeys("password");
        driver.findElement(By.name("login")).click();
        WebElement confirmButton = driver.findElement(By.name("grant"));
        wait.until(ExpectedConditions.visibilityOf(confirmButton));
        confirmButton.click();
        driver.switchTo().window("NATIVE_APP");
}
```

Programski kod 2.9. Primjer Appium testa

2.4.2. Calabash

Calabash [33] je testni okvir koji se koristi za Android, iOS nativne aplikacije te također može testirati hibridne aplikacije. Calabash ima iznimno jednostavnu sintaksu koja omogućava pisanje testova na obje platforme čak i ljudima koji nikada prije nisu programirali. Testovi su napisani u *Cucumber*u te se zatim prebacuju u Robotium tijekom izvođenja. Podržava oko 80 naredbi koje se koriste prirodnim jezikom, a dodavanje novih naredbi moguće je koristeći Ruby ili Javu.

Programski kod 2.10 prikazuje koliko je jednostavno napisati test u Calabashu te je ujedno takav test čitljiv i razumljiv čak i osobama koje nemaju doticaja sa programiranjem.

```
Feature: Login feature
 Scenario: As a valid user I can log into my app
 I wait for text "Welcome"
 Then I press view with id "Login"
 Then I enter text "myUsername" into "login_username"
 Then I enter text "myPassword" into "login_password"
 Then I press view with id "login_button"
 Then I wait for activity "MainActivity"
 Then I see "Welcome myUsername"
```

Programski kod 2.10. Primjer Calabash testa

2.4.3. Espresso

Espresso je namijenjen programerima koji su upoznati sa kodom, kao rezultat toga Espresso testovi su brzi i iznimno pouzdani. Pružaju različite mogućnosti interakcije sa pogledima prikazanim na zaslonu uređaja te vodi računa o sinkronizaciji, odnosno pričekati će sa provjerama sve dok se prikaz ne stabilizira te dok se sve asinkrone operacije ne završe.

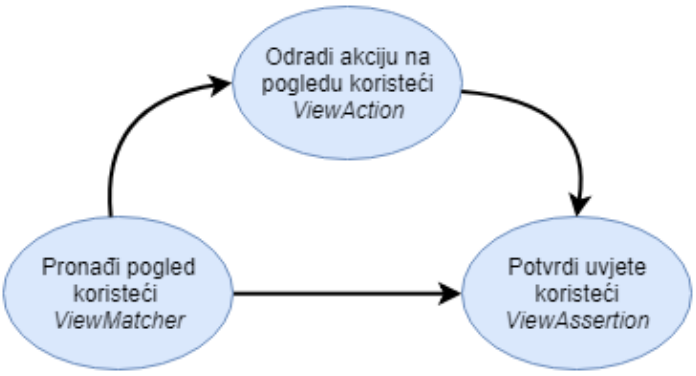
Razlikuje šest tipova anotacija koje se mogu koristiti na metodama unutar testne klase:

1. @BeforeClass
    - Metode sa ovom anotacijom pokreću se na početku i samo jednom
    - Može se koristi za postavljanje preduvjeta koje je potrebno postaviti prije pokretanja testova
2. @Rule
    - Pruža funkcionalno testiranje aktivnosti
    - Omogućava pristup aktivnosti, a preko aktivnosti može se pristupiti resursima itd.
3. @Before
    - Metode sa ovom anotacijom pokreću se prije svakog testa koji se nalazi u testnoj klasi

- Koristi se za postavljanje preduvjeta koje je potrebno postaviti prije svakog testa

4. @Test
    - Metode sa ovom anotacijom pokreću se nakon @Before metode
5. @After
    - Metode sa ovom anotacijom pokreću se nakon svakog testa koji se nalazi u testnoj klasi
    - Može se koristiti za postavljanje vrijednosti u neka početna stanja
6. @AfterClass
    - Metode sa ovom anotacijom pokreću se zadnje i samo jednom
    - Koristi se za izvršavanje metode koja se treba izvršiti nakon što su izvršeni svi testovi

Prema slici 2.9. izrađenoj prema [34] prikazan je proces testiranja koristeći Espresso, prvo je potrebno pronaći pogled nad kojim se želi izvršiti interakcija, zatim se može napraviti provjera ili izvršiti neku od akcija nad pogledom.



Sl. 2.9. Koraci tijekom Espresso testiranja

Prema programskom kodu 2.11. vidljivo je kako se dohvaća pogled sa jedinstvenom oznakom „message“ te se izvršava akciju upisa teksta, zatim se dohvaća pogled s oznakom „button“ te se izvršava akcija klika. Na kraju se dohvaća pogled s oznakom „display“ i provjerava se je li vrijednost koju pogled prikazuje jednaka onoj koja je upisana u pogled s oznakom „message“.

```
@Test
public void testMessage() {
      String MESSAGE = "This is test message";
      Espresso.onView((withId(R.id.message))).perform(ViewActions.typeText(MESSAGE));
      Espresso.onView(withId(R.id.button)).perform(ViewActions.click());
      Espresso.onView(withId(R.id.display)).check(matches(withText((MESSAGE)))));
}
```

Programski kod 2.11. Primjer Espresso testa

2.4.4. UI Automator

Za razliku od Espressa UI Automator je namijenjen svima, odnosno osoba koja ga koristi ne mora poznavati kod koji se testira, već se oslanja isključivo na vidljive UI elemente. Glavna prednost mu je funkcionalno testiranje s više aplikacija, sposobnost testiranja više aplikacija u isto vrijeme te prebacivanje između instaliranih i sistemskih aplikacija. Sadrži koristan alat *UI Automator Viewer* koji se koristi za skeniranje i analiziranje UI komponenta koje su trenutno prikazane na uređaju.

Sastoji se od dva seta API-ja:

1. *UI Automator API*
    - Koristi se za manipuliranje UI komponentama aplikacije
2. *Device State API*
    - Koristi se za izvođenje operacija na samom uređaju (rotacija ekrana, pritisak na neku tipku, itd.):

Prema programskom kodu 2.12 vidljivo je kako je moguće sa UI Automatorom izvršavati operacije nad samim uređajem kao što je pritisak na tipku „Back“ te kako je moguće testirati je li marker prikazan na mapi jednostavnim pretraživanjem objekata koji sadrže ime markera u opisu. Zatim je moguće nad takvim objektom vršiti razne operacije, kao što je u primjeru klik.

```
@Test
public void testMapMarker() {
        UiDevice device = UiDevice.getInstance(getInstrumentation());
        device.pressBack();
        UiObject marker = device.findObject(new UiSelector().descriptionContains(„Marker title“));
        marker.click();
}
```

Programski kod 2.12. Primjer UI Automator testa

2.4.5. Robotium

Robotium [35] je testni okvir za testiranje Android aplikacija, pruža podršku za testiranje Nativnih i hibridnih aplikacija. Zahtjeva minimalno znanje o kodu koji se testira jer baš kao i UI Automator oslanja isključivo na vidljive UI elemente. Testovi se mogu pokretati na emulatoru ili stvarnom uređaju, a koristi se za pisanje funkcionalnih i sistemskih testova. Robotium testovi su jednostavni za pisanje pošto se sve metode nalaze u jednoj klasi (Solo), baš kao i Espresso brine o sinkronizaciji te uglavnom nema potrebe brinuti o kašnjenju, brzo se izvode, automatski pronalazi poglede te samostalno donosi odluke kao što su kada pomaknuti pogled (engl. *Scroll*).

Primjer prikazan u programskom kodu 2.13 prikazuje jednostavan test gdje se uređuje tekst te se zatim prikazuje na zaslonu, kao što se vidi test je jednostavno napisati, a sve potrebne metode za interakciju sa sučeljem dolaze iz klase Solo.

```
public void testTextDisplayed() {
   solo = new Solo(getInstrumentation(), getActivity());
   solo.clickOnText("Other");
   solo.clickOnButton("Edit");
   assertTrue(solo.searchText("Edit Window"));
   solo.enterText(1, "Example text")
   solo.clickOnButton("Save");
   assertTrue(solo.searchText("Changed successfully"));
   solo.clickOnButton("Ok");
   assertTrue(solo.searchText("Example text"));
}
```

Programski kod 2.13. Primjer Robotium testa

2.5. Testiranje kao usluga

Testiranje kao usluga (engl. *Testing as a Service*, TaaS) [36] predstavlja model korištenja vanjskih suradnika prema kojem se testiranje obavlja od strane pružatelja usluge (engl. *Service provider*), a ne od strane vlastitih zaposlenika. TaaS može predstavljati uključivanje konzultanata kako bi pomogli zaposlenicima ili jednostavno prebaciti testiranje određenog dijela koda vanjskim suradnicima. Koristi se kada za testiranje dijela koda nije potrebno veliko znanje o samom dizajnu sustava.

TaaS većinom uključuje usluge kao što su:

1. Regresijsko testiranje (engl. *Regression testing*)
2. Testiranje performansi (engl. *Performance testing*)
3. Testiranje sigurnosti (engl. *Security testing*)
4. Testiranje i praćenje Cloud aplikacija (engl. *Testing and monitoring of cloud-based applications*)

2.6. Osiguranje kvalitete programskog rješenja

Osiguranje kvalitete programskog rješenja (engl. *Software Quality Assurance*) [37] predstavlja skup aktivnosti koje se provode radi osiguravanja kvalitetnog programskog rješenja. Kvaliteta programskog rješenja predstavlja ocjenu programske podrške prema njegovim vanjskim i unutarnjim karakteristikama. Vanjska kvaliteta predstavljaju kako se programska podrška ponaša u realnim situacijama te koliko je koristan korisnicima, unutarnja kvaliteta fokusira se na unutarnje karakteristike programske podrške, a najvažnija karakteristika je kvaliteta napisanog koda. Iako je

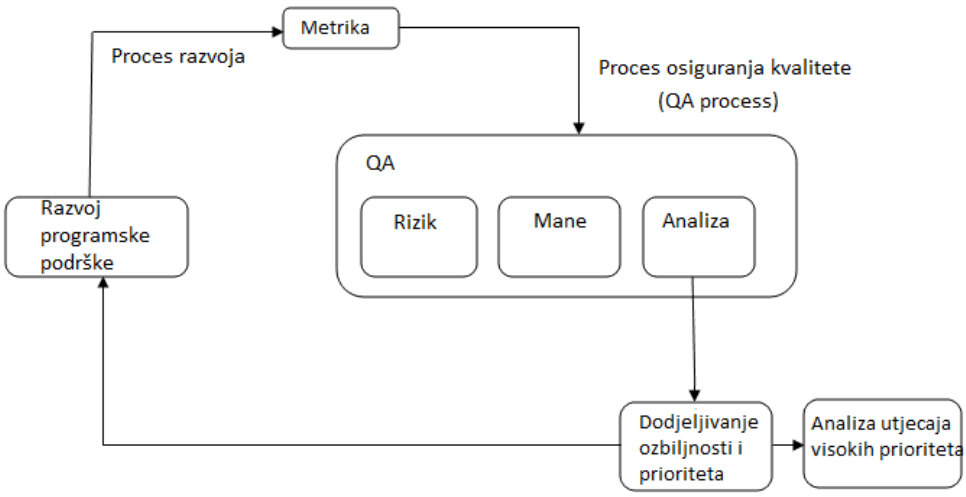
korisnicima važnija vanjska kvaliteta programske podrške ona ovisi o unutarnjoj, jer ukoliko je kod loše napisan teže će biti održavati vanjsku kvalitetu.

2.6.1. Kvaliteta programskog rješenja

Razlikuju se dva pristupa utvrđivanja kvalitete programskog rješenja:

1. Pristup upravljanja manama (engl. *Defect Managment Approach*)
2. Pristup atributima kvalitete (engl. *Quality Attributes Approach*)

Sve što nije onako kako je klijent zamislio naziva se mana u programskoj podršci. Mane se često pojavljuju iz razloga što klijent nije dovoljno dobro definirao zahtjeve ili se jednostavno nije dobro razumio sa programerima što treba napraviti. Najčešće mane koje se pojavljuju zbog lošije komunikacije s klijentom su mane u dizajnu. Osim mana koje nastaju zbog slabih zahtjeva pojavljuju se i mane zbog loše napisanog koda, lošeg upravljanja podacima, itd. Pristup upravljanja manama vodi računa o svakoj mani te joj dodjeljuje stupanj ozbiljnosti. Manja promjena u dizajnu imati će najmanji stupanj ozbiljnosti, dok će pogreška zbog koje programska podrška ne radi ispravno imati najveću ozbiljnost. Uzeći u obzir ozbiljnost mane ona će se riješiti prije ili kasnije od drugih mana, ovakvim se pristupom osigurava rješavanje mana programske podrške prema prioritetima. Na slici 2.10 izrađenoj prema [37] prikazan je blok dijagram pristupa upravljanja mana.



Sl. 2.10. Blok dijagram pristupa upravljanja mana

Pristup atributima kvalitete osigurava da programska podrška zadovoljava dane karakteristike, odnosno tijekom pisanja programskog rješenja nameće uvjete koje takav kod mora zadovoljiti, a fokusira se na šest karakteristika kvalitete:

1. Funkcionalnost (engl. *Functionality*)

- Set funkcija koje pruža programska podrška moraju zadovoljiti svojstva kao što su prikladnost, preciznost, kompatibilnost i sigurnost.

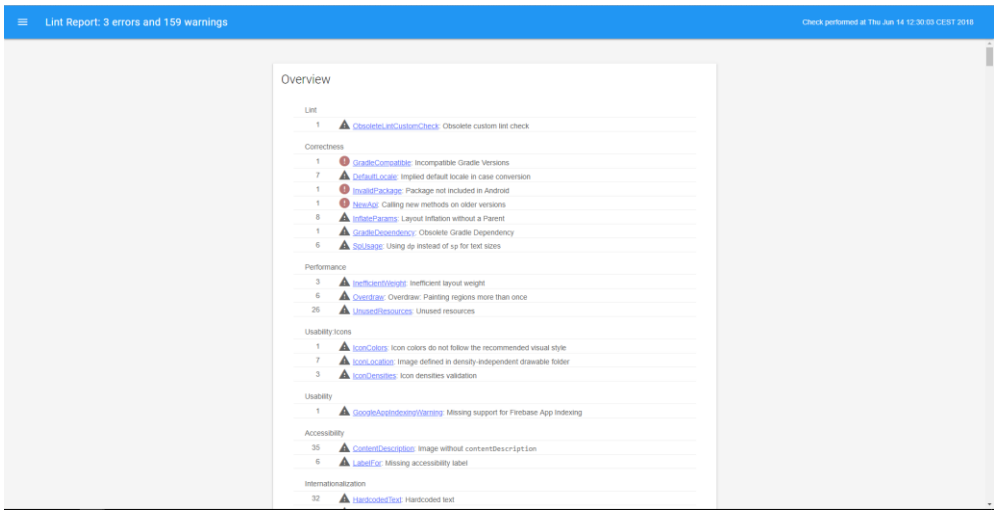
2. Pouzdanost (engl. *Reliability*)
    - Sposobnost programske podrške da ispravno radi u određenim uvjetima određeno vrijeme, može predstavljati i sposobnost programske podrške da preživi isključivanje neke komponente.
3. Korisnost (engl. *Usability*)
    - Predstavlja jednostavnost korištenja funkcija, koliko je lako shvatit što pojedina funkcija predstavlja te koliko vremena treba korisniku da bi shvatio pojedinu funkciju programske podrške.
4. Efikasnost (engl. *Efficiency*)
    - Ovisi o arhitekturi i kvaliteti napisanog koda.
5. Održivost (engl. *Maintainability*)
    - Predstavlja sposobnost identificiranja i popravljanja pogreške u programskoj podršci, a ovisi o čitljivosti i kompleksnosti napisanog koda.
    - Dobru ocjenu održivosti dati će kod koji je lako analizirati, odnosno kod u kojem je lako pronaći pogrešku, kod koji je lako promjenjiv i koji će ostati stabilan nakon promjene te za kraj održivi kod je kod za kojeg je lako pisati testove.
6. Pokretnost (engl. *Portability*)
    - Predstavlja programsku podršku koja se lako prilagođava promjenama u okolini, a odnosi se na jednostavnost instalacije, jednostavnost promjena specifikacija programske podrške i jednostavnost promjene komponente sustava u danom okruženju.

2.6.2. Ispitivanje kvalitete programskog rješenja

Ispitivanje programskog rješenja vrši se rigoroznim testiranjem kako bi se otkrile sve mane programske podrške, uključujući razlike realizacije i zahtjeva klijenta te mane tijekom pisanja programske podrške, greške u radu, itd. Osim ručnog testiranja pripadnika QA tima (engl. *Quality Assurance Team*), postoje razni alati kojima je moguće testirati kvalitetu programskog rješenja. Alati korišteni tijekom razvoja programskog rješenja za pomoć posjetiteljima konferencije su *Lint* izvješće razvojnog okruženje Android Studio te *JaCoCo* alat za mjerenje postotka pokrivenosti koda automatskim testovima.

Razvojno okruženje Android Studio nudi mogućnost generiranja *Lint* izvješća o projektu, *Lint* izvješće analizira kompletan projekt i uspoređuje sa dobrom praksom prilikom razvijanja programskog rješenja. U generiranom izvješću mogu se vidjeti sve mane projekta sa objašnjenjima

zašto je mana štetna za projekt, koliko je ozbiljna prijetnja te daje pomoć prilikom saniranja iste, kako to izgleda može se vidjeti na slici 2.11. Generirano izvješće pronašlo je 3 greške te 159 upozorenja, ovakvo izvješće predstavlja veliku pomoć pri osiguravanju kvalitete programskog rješenja, uspješnim saniranjem svih pogrešaka i upozorenja unaprijediti će se performanse i sigurnost same aplikacije. Ponekad rješavanje svih pogrešaka, a pogotovo upozorenja predstavlja veliki napor, ali njihovo smanjivanje na što manji broj zasigurno će podići kvalitetu programskog rješenja.



Sl. 2.11. Prikaz *Lint* izvješća

JaCoco je alat koji se koristi kao mjerilo dobro testirane aplikacije, aplikacija može imati veliki broj testova, a u isto vrijeme biti lošije testirana od aplikacije jednake veličine s manje testova. Razlog tome je lošija pokrivenost testovima, testovi bi trebali obuhvatiti svaki dio aplikacije. *JaCoCo* izvješće pomaže upravo u segmentu pregleda koji dijelovi aplikacije nisu testirani. Osim postotka koda koji je testiran pruža uvid u pojedine klase i metode te je lako vidjeti koji dio aplikacije je slabo testiran, odnosno što je potrebno testirati kako bi podigli postotak pokrivenosti testovima aplikacije. Kako izgleda izvješće prikazano je na slici 2.12.

app > com.example.karlo.sstconference.modules.chairs

com.example.karlo.sstconference.modules.chairs

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
ChairsViewModel		80%		50%	2	6	3	20	1	5	0	1
ChairsActivity		96%		55%	4	21	4	77	1	16	0	1
ChairsActivity.new Object() {...}		89%		n/a	0	1	0	1	0	1	0	1
Total	25 of 380	93%	5 of 11	54%	6	28	7	97	2	22	0	3

Sl. 2.12. Prikaz *JaCoCo* izvješća

3. PROGRAMSKO RJEŠENJE ZA POSJETITELJE KONFERENCIJE

Razvijeno programsko rješenje za posjetitelje konferencije (u daljnjem tekstu: SST aplikacija) predstavlja primjer dobro napisane aplikacije, čije je komponente lako testirati. Tijekom razvoja korištene su tehnologije opisane u drugom poglavlju, a korišteni alati biti će detaljnije opisani u nastavku

3.1. Zahtjevi na sustav

Prije razvoja SST aplikacije postavljeni su razni zahtjevi koje bi aplikacija trebala zadovoljiti, zahtjevi su vezani za različite funkcionalnosti koje aplikacija mora pružiti krajnjem korisniku.

Tablica 3.1 prikazuje zahtjeve koje aplikacija mora ispuniti vezano za upravljanje korisnicima.

Tablica 3.1. Zahtjevi za upravljanje korisnicima

ID	Prioritet	Opis
		Zahtjevi za upravljanje korisnicima
1	1	Aplikacija mora pružiti različite oblike prijave
2	1	Aplikacija mora pružiti različit prikaz ovisno o obliku prijave

Tablica 3.2 prikazuje zahtjeve koje aplikacija mora ispuniti vezano uz konferencijski program.

Tablica 3.2. Zahtjevi vezani uz konferencijski program

ID	Prioritet	Opis
		Zahtjevi vezani uz konferencijski program
1	1	Aplikacija će pružiti informacije o datumima konferencije
2	1	Aplikacija će sadržavati prikaz svih predavanja s radovima i vremenima
3	2	Aplikacija će pružiti mogućnost pretraživanja radova po ključnim riječima
4	2	Aplikacija će prikazati detalje o izlaganjima najvažnijih predavača

Tablica 3.3 prikazuje zahtjeve koje će aplikacija ispuniti registriranim korisnicima.

Tablica 3.3. Zahtjevi koje će aplikacija ispuniti registriranim korisnicima

ID	Prioritet	Opis
		Zahtjevi koje će aplikacija ispuniti registriranim korisnicima
1	2	Aplikacija će omogućiti pretplatu na pojedina izlaganja
2	3	Aplikacija će omogućiti pretplatu na pauze za kavu i ručak
3	2	Aplikacija će pružiti slanje obavijesti korisniku o pretplaćenim događajima
4	2	Aplikacija će pružiti mogućnost komentiranja raznih izlaganja
5	2	Aplikacija će omogućiti učitavanje slika koje će svi korisnici moći vidjeti

Tablica 3.4 prikazuje zahtjeve za prikaz informacija o regiji i raznim lokacijama koje se nalaze unutar korisnikove okoline.

Tablica 3.4. Zahtjevi za prikaz informacija o regiji i raznim lokacijama

ID	Prioritet	Opis
		Zahtjevi za prikaz informacija o regiji i raznim lokacijama
1	2	Aplikacija će pružiti informacije o fakultetu i sveučilištu
2	2	Aplikacija će pružiti informacije o regiji u kojoj se konferencija održava
3	3	Aplikacija će omogućiti prikaz restorana i kafića na mapi u okolini korisnika
4	3	Aplikacija će omogućiti prikaz muzeja, crkva, biblioteka te zoološkog vrta u okolini korisnika

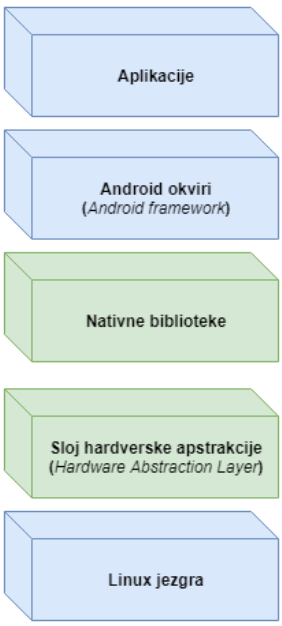
3.2. Opis platformi, alata i tehnologija

SST aplikacija implementira značajan broj alata i tehnologija, neke od najvažnijih kao što su *RxJava*, *Retrofit* te skup biblioteka *Android Architecture Components* bit će detaljnije komentirane u nastavku. Platforme korištene tijekom razvoja SST aplikacije su Android operacijski sustav za koji je sama aplikacija i namijenjena te Firebase Web platforma za pružanje usluga koja između ostalog omogućava programerima brz i jednostavan sustav za pohranu, dohvaćanje te dijeljenje podataka.

3.2.1. Android

Android [38] je operacijski sustav za mobilne telefone, tablete te razne druge uređaje kao što su satovi, televizori, automobili. Android Inc. osnovan je 2003. godine kao tvrtka koja bi se bavila razvojem naprednog operacijskog sustava za digitalne kamere. Kada su osnivači uvidjeli da se potražnja za digitalnim kamerama smanjuje okrenuli su svoje planove prema pametnom telefonu. Google je 2005. godine otkupio tvrtku Android Inc., a projekt razvoja operacijskog sustava naziva se Android otvoreni projekt (engl. *Android Open Source Project*). Android je zasnovan na Linux jezgri koja je u širokoj upotrebi na raznim uređajima kao što su računala, serveri, usmjerivači, televizori, itd.

Android je najpopularniji operacijski sustav na mobilnim uređajima, projekt je otvorenog koda te je vođen od strane Google-a. Google izrađuje svoju verziju Android operacijskog sustava koja je tada korištena i uređivana od strane drugih proizvođača kao što su Samsung, Sony, LG, Huawei te mnogi drugi. Na slici 3.1 izrađenoj prema [39] prikazana je arhitektura Android operacijskog sustava.



Sl. 3.1. Android arhitektura

Zbog svoje prirode otvorenog koda ista verzija Android podrške može izgledati potpuno drugačije na mobitelima različitih proizvođača. Nakon što Google pruži najnoviju verziju programske podrške proizvođači uzimaju programsku podršku te ju prerađuju kako bi je personalizirali. Većinom se mijenja samo grafičko sučelje, a najpoznatija su *Samsung TouchWiz, Sony Xperia* te *Huawei Emotion*.

Uređaji s Android operacijskim sustavom podržavaju razne značajke, neke od najbitnijih su:

1. Slanje poruka
    - Omogućava slanje SMS, MMS, *Android Cloud To Device Messaging* (C2DM) i noviju verziju *Android Google Cloud Messaging* (GCM) servisa za razmjenu poruka putem interneta. Internet pretraživanje
2. Usluge upravljane govorom
    - Postoje razne naredbe koje je moguće izvršiti govorom, naredbe kao što su poziv, slanje poruka i navigacija dostupne su od verzije Android 2.2
3. *Multi-touch*
    - Omogućava podršku za prepoznavanje višestrukog dodira istovremeno.
4. *Multitasking*
    - *Multitasking* između raznih aplikacija sa jedinstvenom alokacijom memorije.
5. Višejezična podrška
6. Pristupačnost
    - U postavkama nalazi se paket usluga pristupačnosti, predstavljaju pomoć ljudima koji imaju problemima s vidom, sluhom, itd.

7. Povezanost
    - Android uređaj može se povezati preko raznih tehnologija kao što su GSM/EDGE, Bluetooth, LTE, CDMA, NFC, WiMAX, itd.
    - Od verzije Android 2.2 moguće je koristiti mobilni uređaj kao mrežno mjesto.
8. Medijska podrška
    - Podrška prema različitim multimedijskim datotekama kao što su H.263, H.264, MP3, MPEG-4, WAV, JPEG, PNG, GIF i mnogi drugi.

Mobilni uređaji sa Android operacijskim sustavom imaju pristup različitim resursima uređaja, na raspolaganju imaju veliki broj senzora kao što su senzor osvjetljenja, senzor udaljenosti, senzor vlage, senzor otkucaja srca, barometar, žiroskop, akcelerometar, magnetometar i mnogi drugi. Ovi se senzori osim obavljanja zadaća koje imaju prema Android operacijskom sustavu mogu koristiti prilikom razvijanja aplikacija.

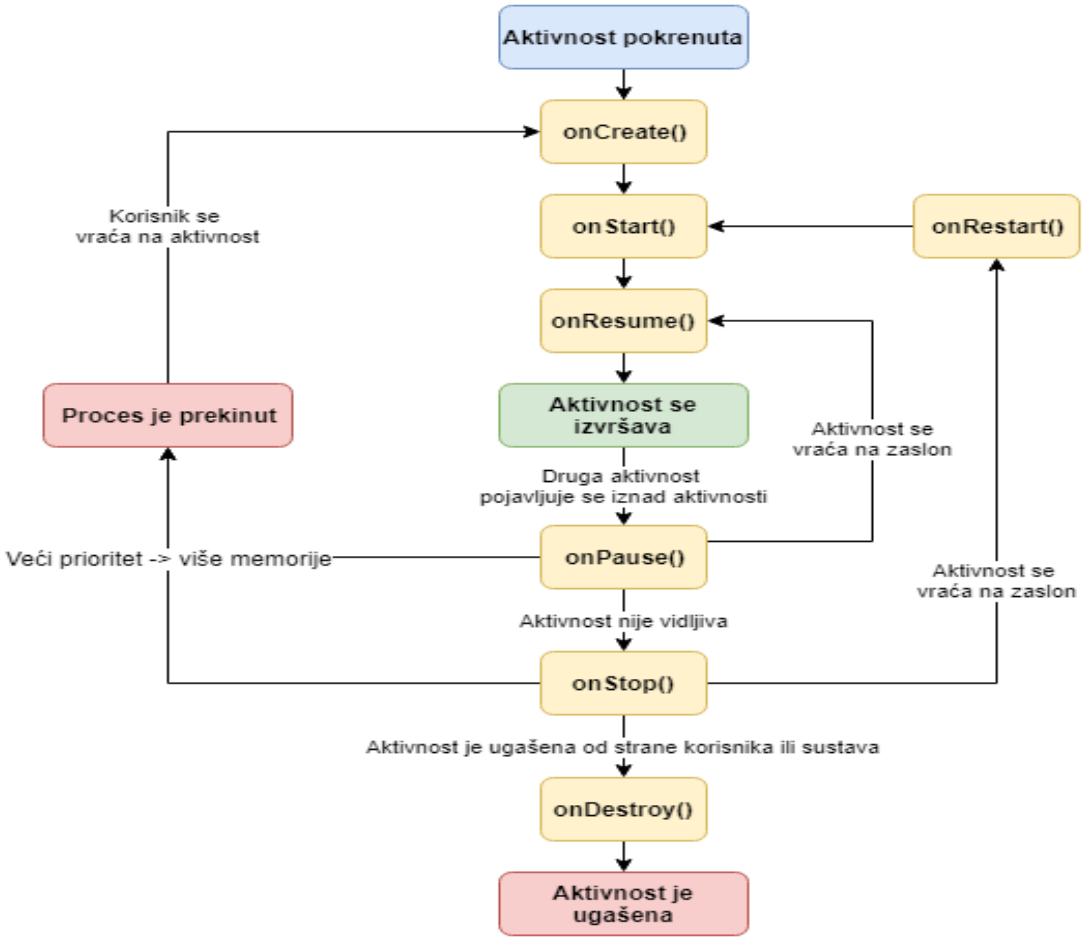
Razvijanje Android aplikacija odvija se u razvojnom okruženju Android Studio koje pruža podršku tijekom razvoja i testiranja aplikacije. Android Studio je IDE (engl. *Integrated Software Aplication*) izgrađen na temelju *IntelliJ IDEA* koji je razvijen od strane tvrtke JetBrains, Android Studio je namijenjen isključivo razvoju Android aplikacija koje se mogu pisati u programskim jezicima Java ili Kotlin. Android Studio sadrži uređivač teksta i Android SDK sa svim potrebnim alatima potrebnima za razvoj, testiranje i objavljivanje Android aplikacije. Android Studio temeljen je na Java programskom jeziku, stoga treba instalirati *Java Development Kit* (JDK), ukoliko već nije instaliran na računalo Android Studio će ga preporučiti tijekom instalacije.

Tijekom razvoja Android mobilne aplikacije treba razlikovati četiri osnovne komponente [40]:

1. Aktivnosti (engl. *Activities*)
    - Odgovorne za prikaz sadržaja korisniku i upravljanje korisnikovim zahtjevima
2. Servisi (engl. *Services*)
    - Izvršavaju pozadinske procese povezane s aplikacijom
3. *Broadcast Receiveri*
    - Odgovorni za komunikaciju s Android operacijskim sustavom i drugim aplikacijama.
4. Pružatelji sadržaja (engl. *Content Providers*)
    - Omogućuju dijeljenje sadržaja s drugim aplikacijama, datoteke se mogu pohraniti u datotečnom sustavu, bazi podataka ili drugdje.

Postoje i dodatne komponente koje se koriste kako bi se upotpunile one osnovne, neke od dodatnih komponenata su Fragmenti, Pogledi, *Intenti*, Resursi i *Manifest*.

Slika 3.2. izrađena prema [41] prikazuje kako izgleda životni ciklus pojedine Aktivnosti, kada korisnik izlazi iz aktivnosti sustav poziva metode koje će rastaviti aktivnost. U nekim slučajevima ovo rastavljanje je samo djelomično i aktivnost nastavlja živjeti u memoriji (primjer je kada se korisnik prebaci na drugu aplikaciju). Ukoliko se korisnik vrati na tu aktivnost ona se nastavlja točno ondje gdje ju je korisnik napustio. Vjerojatnost da će sistem ubiti neki proces i aktivnosti koje se u njemu nalaze ovisi o trenutnom stanju aktivnosti.

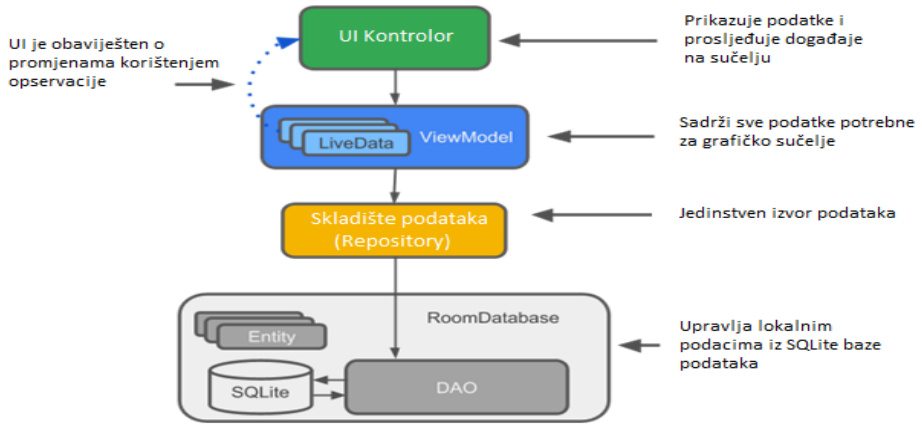


Sl. 3.2 Dijagram životnog ciklusa aktivnosti

3.2.2. Android Architecture Components

Architecture Components [42] naziv je za skup biblioteka koje pomažu pri razvoju robusnih, testabilnih i održivih aplikacija. Skup biblioteka predstavljen je krajem 2017. godine, pružaju jednostavan, fleksibilan i praktičan pristup razvoju aplikacija rješavajući neke poznate probleme kao što su praćenje životnog ciklusa aktivnosti, dijeljenje stanja u kojem se aplikacija nalazi između raznih aktivnosti i fragmenata je iziskivalo puno vremena i pažnje. Iako navedeni problemi nisu nepremostivi predstavljali su trn u oku svakog programera, uvođenjem *Architecture*

Components programeri sada imaju više vremena u razvoju drugih značajki. Na slici 3.3. izrađenoj prema [43] prikazana je arhitektura *Architecture Components*.



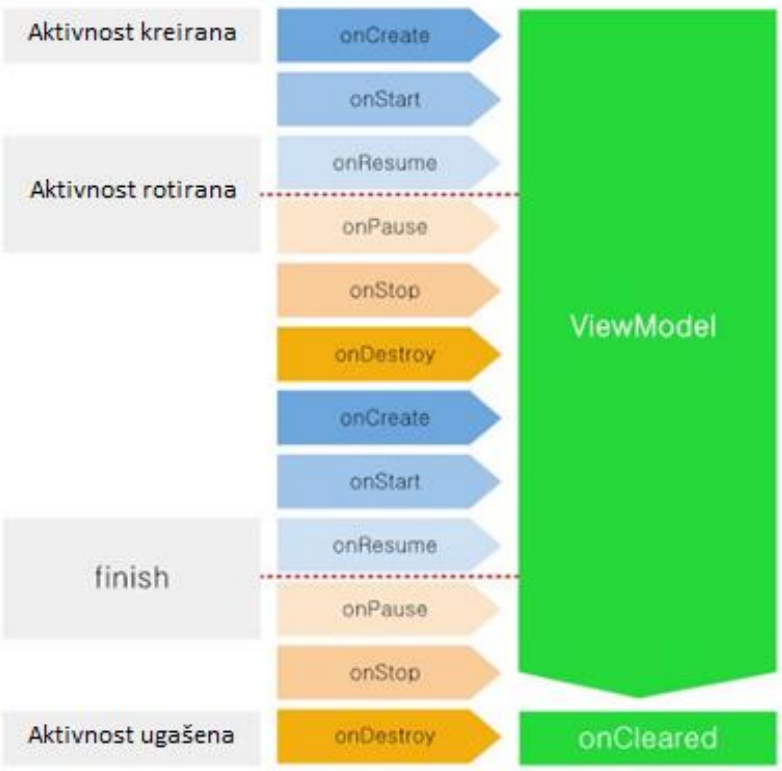
Sl. 3.3. Osnovna forma koja čini *Architecture Components*

Arhitektura *Architecture Components* sastoji se od četiri glavna dijela, sa sljedećim atributima:

1. *Entity*
    - Predstavlja klasu koja ima anotaciju *@Entity* i opisuje tablicu u bazi podataka
2. *SQLite*
    - Baza podataka na mobilnom uređaju
3. *DAO (*engl. *Data Access Object)*
    - Pruža različite SQL upite kao funkcije koje se koriste kada se želi obaviti neka operacija nad bazom podataka.
4. *RoomDatabase*
    - Sloj iznad *SQLite* baze podataka koja rješava probleme pisanja velike količine koda u svrhu upravljanja bazom podataka.
5. Repozitorij (engl. *Repository*)
    - Klasa koja se kreira kako bi se manipuliralo podacima iz više izvora (Internet, baza podataka, itd.)
6. *ViewModel*
    - Služi kao sloj između pružatelja podataka i korisnika (*User Interface*), odgovoran za uklanjanje svih „teških“ zadataka sa aktivnosti i korisničkog sučelja.
    - *ViewModel* ostaje „živ“ nakon promjene orijentacije zaslona.
7. *LiveData*
    - Spremište podataka koje prati princip Pružatelj – Promatrač, drugim riječima na ovakvo se spremište može pretplatiti i pratiti svaku promjenu podataka u njemu.

Architecture Components koristi se zajedno sa oblikovnim obrascem MVVM koji je opisan u poglavlju 2.2.3.3.

Prema slici 3.4 izrađenoj prema [44] vidljivo je kako *ViewModel* ostaje „živ“ kroz sve promjene stanja aktivnosti ili fragmenta te se uništava tek kada se aktivnost ili fragment uništi. Ovakvo svojstvo omogućuje čuvanje podataka sve dok su oni potrebni, primjer je rotacija zaslona koja neće rezultirati ponovnim učitavanjem svog sadržaja jer će sadržaj biti sačuvan u *ViewModelu* iz kojeg će se sadržaj zatim samo dohvatiti.



Sl. 3.4. Životni ciklus *ViewModela*

3.2.3. Retrofit i RxJava

Retrofit

Retrofit je REST (engl. *Representational state transfer*) klijent građen nad OkHttp klijentom dizajniran za Android i Javu. Koristi anotacije kako bi opisao različite HTTP zahtjeve, pruža mogućnost kao što su dinamički parametri unutar URL-a, parametre upita, prilagođena zaglavlja, skidanje i učitavanje datoteka, pisanje lažnih odgovora (engl. *Mocking responses*). Omogućava sinkrone i asinkrone zahtjeve, a samostalno brine o sinkronizaciji, upravljaju nitima, keširanju, učitavanju, itd. Za upravljanje HTTP zahtjevima koristi OkHttp biblioteku koja je obavezna uz Retrofit, omogućuje jedinstveno definiranje krajnjih točaka komunikacije putem HTTP protokola (engl. *endpoint*).

Koristeći Retrofit u kombinaciji sa Gson pretvaračem za JSON ili Xml pretvaračem za XML prilično je lako podatke koji se nalaze na nekom Web servisu u JSON ili XML formatu dohvatiti i pretvoriti u POJO (engl. *Plain Old Java Object*). Retrofit automatski serijalizira JSON podatke koristeći Gson pretvarač i POJO koji mora biti unaprijed definiran po JSON strukturi, takav POJO objekt može sadržavati sve atribute koje ima JSON ili samo one koji se žele dohvatiti.

Za izvršavanje Retrofit zahtjeva potrebna je model klasa koja se koristi za serijalizaciju odgovora, sučelje u kojem je definiran tip HTTP zahtjeva te Retrofit instancu koja se koristi za izvršavanje HTTP zahtjeva definiranog u sučelju. Svako sučelje predstavlja kolekciju različitih API poziva, svaka metoda unutar sučelja koja se želi koristi kao HTTP zahtjev mora sadržavati anotaciju zahtjeva koji obavlja

RxJava.

RxJava moćan je alat koji implementira reaktivno programiranje. Princip reaktivnog programiranja je konzumiranje podataka kako oni dolaze, ostvaruje se odašiljanjem promjena slušateljima. RxJava pruža Java API za asinkrono programiranje sa tokovima podataka koji se mogu promatrati (engl. *Observable streams*). Pruža jednostavno upravljanje višestrukih događaja, pogrešaka i zastoja u emitiranju objekta. Pojednostavljuje korištenje više niti za različite zadatke, česta je praksa da se pružatelj podataka i pretplatnik na te podatke nalaze na drugačijim nitima.

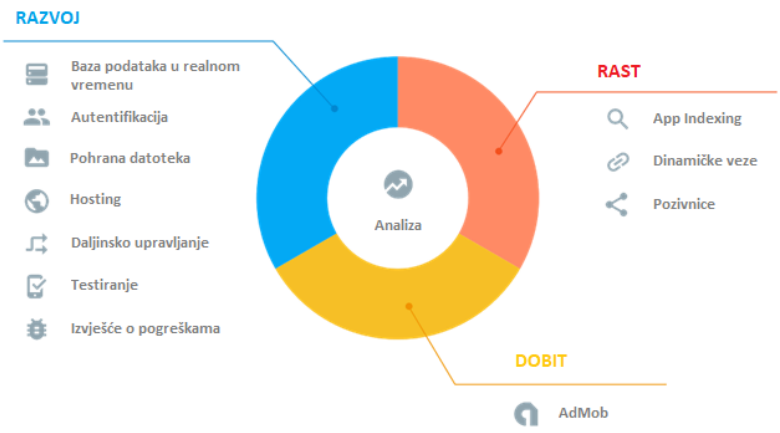
Temelj RxJava koda predstavljaju tri komponente:

1. *Observables*
    - Predstavljaju izvor podataka, pružaju podatke kada se pretplatnik pretplati
    - Može emitirati bilo koji broj podataka uključujući nulu
    - Emitira svim pretplatnicima, ne zanima ga tko su oni
2. Pretplatnik (engl. *Subscribers* / *Observers*)
    - *onNext()* poziva se svaki puta kada promatrani objekt emitira novi podatak
    - *onComplete()* poziva se kada promatrani objekt završi sa emitiranjem podataka
    - *onError()* poziva se ukoliko promatrani objekt završi emitiranje podataka sa pogreškom
3. Set metoda za prilagodbu podataka
    - Različite metode koje omogućuju manipulaciju podacima, kao što je mapiranje u neki drugi objekt, pretvaranje u *Observable,* itd.

3.2.4. Firebase

Firebase [45] je mobilna i Web platforma koja opskrbljuje programere sa mnoštvom alata i servisa kako bi im pomogla u razvijanju kvalitetnih aplikacija, podizanju broja korisnika i kako bi se povećali prihodi od same aplikacije. Firebase je osnovan 2011. godine kao *startup* nazvan Envolve, bio je zamišljen kao API koji je omogućio integraciju usluge online razmjene poruka u stvarnom vremenu. Nakon nekog vremena programeri su počeli koristiti Envolve kako bi sinkronizirali podatke aplikacije, kao što su stanje igre u realnom vremenu (rezultati). Što je otvorilo nove vidike osnivačima Envolvea te su se James Tamplin i Andrew Lee odlučili razdvojiti sustav razmjene poruka i arhitekturu u stvarnom vremenu. U travnju 2012. godine nastao je Firebase kao zasebna tvrtka koja je pružala *Backend-as-a-Service* funkciju u stvarnom vremenu. Google je kupio Firebase 2014. godine te je od tada doživio rapidan rast i posao robusna multifunkcionalna platforma.

Na slici 3.5 izrađenoj prema [46] prikazane su neke od usluga koje nudi Firebase, one se mogu podijeliti u dvije osnovne grupe, a to su razvoj i testiranje aplikacije te rast i uključivanje publike.



Sl. 3.5. Neki od Firebase servisa

Iz prve grupe servisa izdvajamo:

1. Bazu podatka u realnom vremenu (engl. *Realtime Database*)
    - NoSQL baza podataka koja se nalazi u *Cloudu*, podaci se spremaju u JSON formatu radi lakšeg dohvaćanja.
    - Sinkronizacija se vrši unutar milisekunde, što znači da će svi uređaji spojeni na bazu podataka dobit najnovije podatke onog trena kada su oni postavljeni.
2. Pohrana datoteka (engl. *Cloud Storage*)
    - Firebase pruža svojim korisnicima pohranu za video, slike te mnoge druge tipove podataka.

- Ukoliko se učitavanje datoteke prekine u bilo kojem trenutku nastaviti će onda gdje je stalo kada se za to stvore uvjeti.

3. Autentifikacija (engl. *Authentication*)
    - Pruža skup alata za autentifikaciju korisnika emailom ili pomoću servisa kao što su Facebook, Twitter, Google, itd.
4. *Hosting*
    - Firebase hosting nudi sigurnost, brzu isporuku sadržaja te brzo izbacivanje aplikacije u produkciju.
5. Daljinsko upravljanje (engl. *Remote Config*)
    - Pruža mogućnost uređivanja dizajna i ponašanja aplikacije bez potrebe objavljivanja nadogradnje.
6. Testiranje (engl. *Testing*)
    - Test Lab pruža mogućnost testiranja na stvarnim uređajima različitih konfiguracija.
    - Pruža mogućnost Robo testova koji analiziraju kod i izvode testove vezane za korisničko sučelje.
7. Izvješće o pogreškama (engl. *Crash Report*)
    - Šalje potpuno izvješće o pogreški koje će pružiti pomoć pri rješavanju problema, omogućava praćenje fatalnih i onih ne fatalnih pogrešaka, skuplja sve relevantne podatke koje su potrebne za uspješnu dijagnozu pogreške.

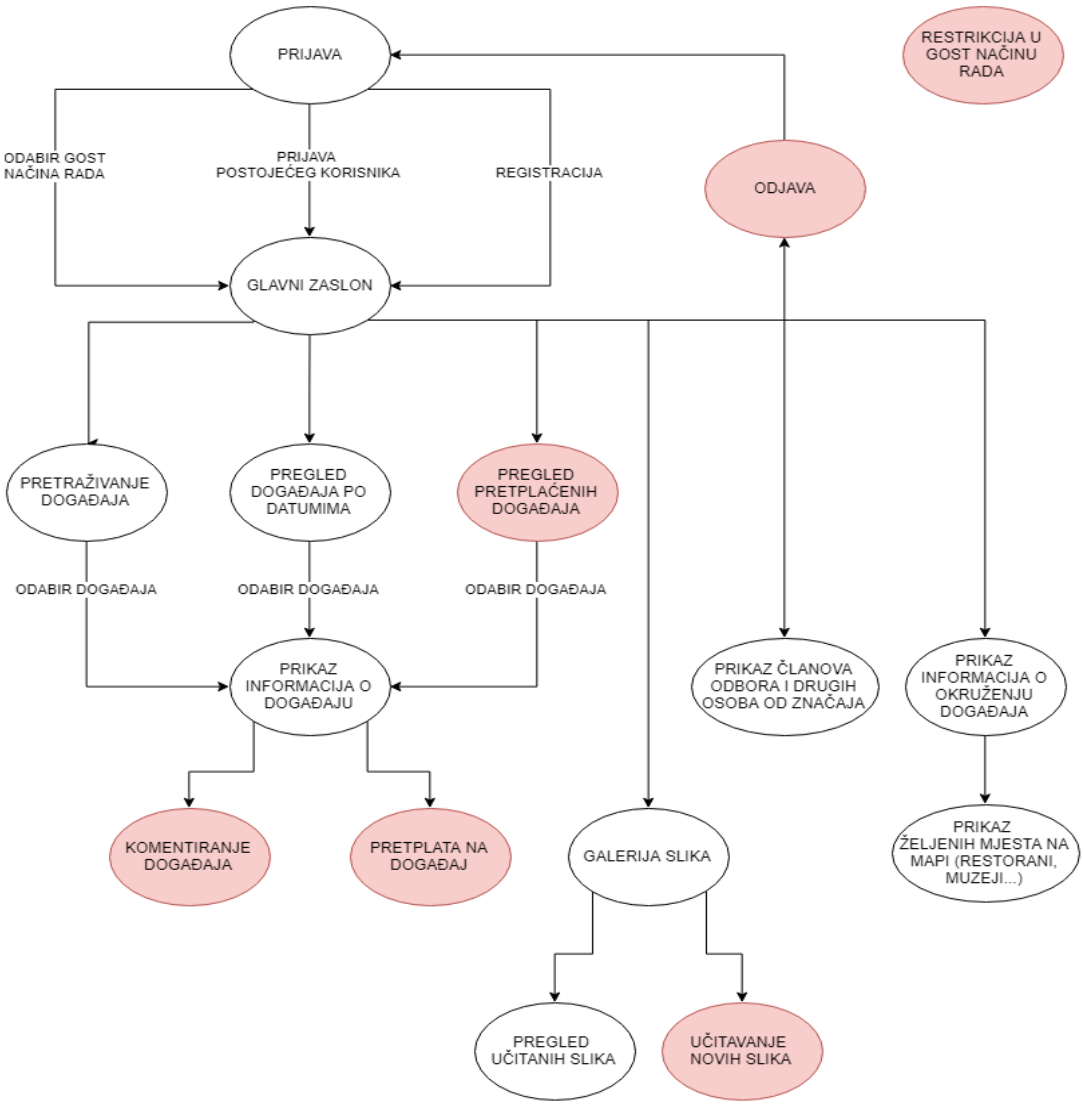
Grupa za rast aplikacije i uključivanje sudionika pruža sljedeće:

1. *App Indexing*
    - Ubacuje aplikaciju u tražilice, ako osoba ima instaliranu aplikaciju pretraživanje nekog pojma relevantnog za aplikaciju pokrenuti će aplikaciju, a ukoliko aplikacija nije instalirana kao prvi rezultat biti će kartica koja omogućava instalaciju aplikacije.
2. Dinamičke veze (engl. *Dynamic Links*)
    - Pruža dinamičke veze koje mogu odvesti korisnika na točno mjestu unutar aplikacije te se mogu mijenjati ovisno o uvjetima tijekom izvedbe koda.
3. Pozivnice (engl. *Firebase Invites*)
    - Pozivnice omogućavaju korisnicima slanje personaliziranih poziva na instalaciju aplikacije, šalju se u obliku emaila ili SMS poruka, pomoću prethodno spomenutih dinamičnih veza.
4. *AdMob*
    - Pojednostavljuje proces ubacivanje reklama u aplikaciju, prati obrazac nativnog formata reklama što znači da će se reklama uklopiti u dizajn aplikacije.

- Integriran sa Firebase analitikom drugim riječima omogućuje praćenje uspješnosti pojedinih reklama.

3.3. Način rada sustava

Sustav se dijeli na nekoliko najvažnijih dijelova, početni zaslon je prijava gdje se pruža mogućnost prijave s postojećim računom, registracije novog računa ili odabir gost načina rada. Nakon početnog zaslona slijedi glavni zaslon koji služi za navigaciju između ostalih dijelova aplikacije kao što su galerija gdje korisnik ima mogućnost pregleda dijeljenih slika s konferencije te učitavanje vlastitih slika. Kao najvažniji dio aplikacije treba spomenuti program konferencije koji korisniku pruža pregled događaja po datumima i vremenima, komentiranje događaja te pretplatu na željene događaje. Valja spomenuti i dio aplikacije zadužen za pružanje informacija o samom okruženju konferencije te prikaz raznih mjesta na mapi. Blok dijagram sustava prikazan je na slici 3.6.

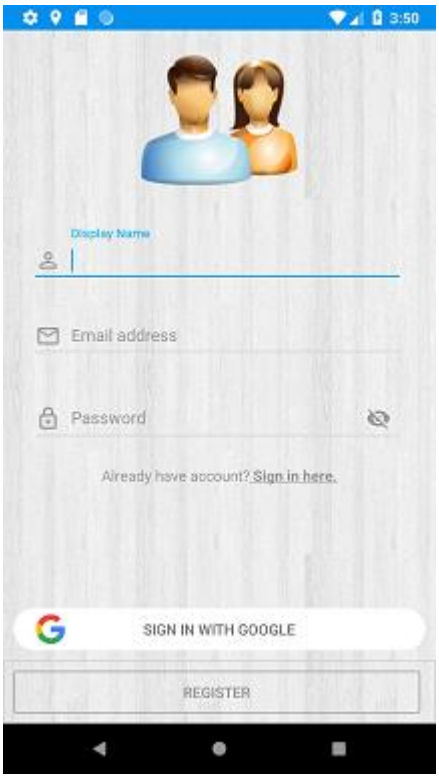


Sl. 3.6. Blok dijagram sustava

3.3.1. Prijava

Kada korisnik instalira aplikaciju prvo što će ga dočekati je prijava. Korisnik ima mogućnost prijave putem postojećeg računa koristeći email i lozinku, prijavu pomoću Google računa, registraciju novog računa te ima mogućnost preskočiti prijavu.

Sl. 3.7. Zaslon za prijavu korisnika



Sl. 3.8. Zaslon za registraciju korisnika

Nakon što je korisnik jednom prošao zaslon s prijavom, bilo to prijavom, registracijom ili odabirom opcije za preskok prijave, sljedeći puta kada uđe u aplikaciju neće vidjeti zaslon prijava nego glavni zaslon aplikacije.

Dio koda zadužen za preusmjeravanje korisnika u gost načinu rada prikazan je unutar programskog koda 3.1.

```
if (EasyPrefs.getGuestMode(this)) {
        AppConfig.USER_LOGGED_IN = false;
        goToHome();
}
```

Programski kod 3.1. Dio koda zadužen za preusmjeravanje korisnika u gost način rada

Provjerava je li korisnik u gost načinu rada dohvaćajući varijablu iz spremljenih svojstava aplikacije te ukoliko je preusmjerava korisnika prema glavnom zaslonu aplikacije.

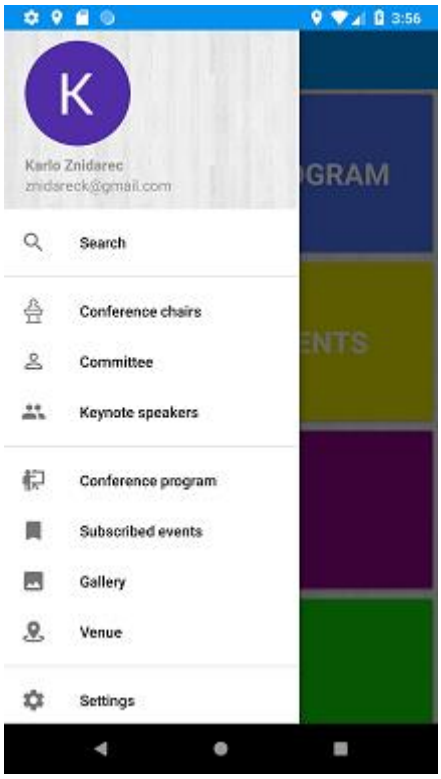
Dio koda zadužen za preusmjeravanje prijavljenog korisnika prikazan je unutar programskog koda 3.2.

```
mViewModel.getUser().observe(this, user -> {
    if (user != null) {
        AppConfig.USER_LOGGED_IN = true;
        EasyPrefs.putGuestMode(this, false);
        goToHome();
    } else {
        findViewById(android.R.id.content).setVisibility(View.VISIBLE);
    }
});
```

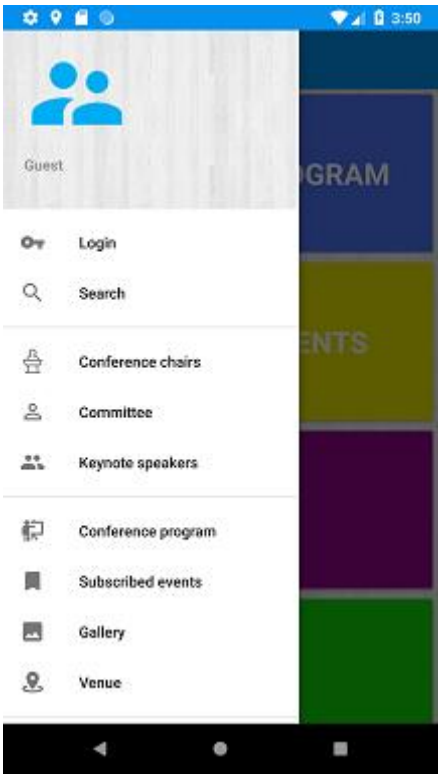
Programski kod 3.2. Dio koda zadužen za preusmjeravanje prijavljenog korisnika

Aktivnost je pretplaćena na metodu *getUser()* iz *ViewModela*, nakon što se postavi vrijednost korisnika unutar *ViewModela* dobit će se odgovor, ukoliko je taj odgovor *null* znači da korisnik nije prijavljen te se prikazuje sadržaj potreban za prijavu. Ukoliko je primljen objekt korisnika koji je validan postavljaju se potrebna polja te se korisnik preusmjerava prema glavnom zaslonu aplikacije.

Ovisno o kakvoj je prijavi riječ imati će opcije prijavljenog korisnika ili gosta:



Sl. 3.9. Navigacija prijavljenog korisnika

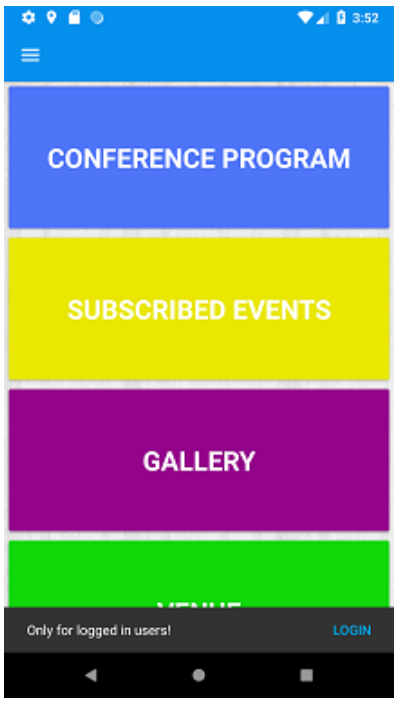


Sl. 3.10. Navigacija korisnika u gost načinu rada

Prijavljeni korisnici imaju svoju sliku, ime za prikaz te email, ukoliko je korisnik izvršio prijavu putem Google računa automatski se povlači slika s tog računa. Korisnik u gost načinu rada neće

moći iskoristi puni spektar mogućnosti aplikacije, kao što su pretplata na događaje, komentiranje događaja, učitavanje slika.

Na slici 3.11. prikazano je kako je korisnik u gost načinu rada pokušao pristupiti sadržaju kojem nema pristup, tada mu se pojavljuje poruka kako nema pravo pristupa te se daje mogućnost odlaska na zaslon sa prijavom.



Sl. 3.11. Restrikcija za korisnike u gost načinu rada

3.3.2. Pretraživanje događaja

Korisnici imaju mogućnost pretrage svih predavanja koja će biti održana na konferenciji po ključnim riječima koja se nalaze u njihovom nazivu.

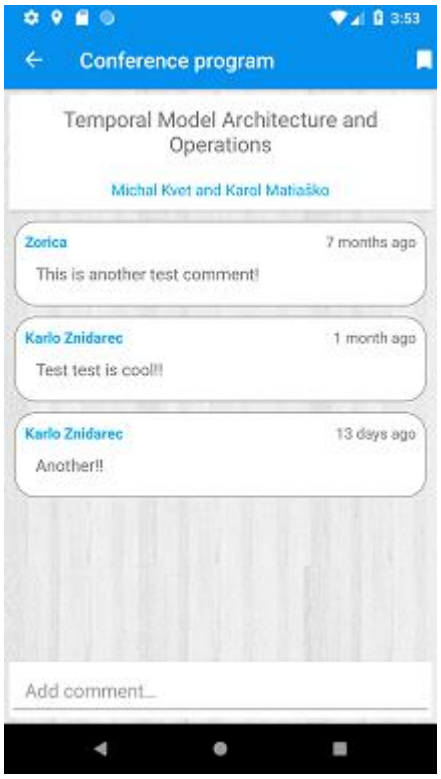
Sl. 3.12. Korisnik pretražuje događaje po ključnoj riječi „temp“

Sl. 3.13. Korisnik ne nailazi na rezultate

Nakon uspješne pretrage korisnik ima mogućnost pretplatiti se ili komentirati događaj.

3.3.3. Pretplata i komentiranje događaja

Prijavljeni korisnici imaju mogućnost pretplate na događaje, bilo da je riječ o nekom predavanju ili stanci za kavu. Korisnici se pretplaćuju na zaslonu sa detaljima pojedinog događaja, na istom se zaslonu nalaze i komentari korisnika ukoliko ih ima. Kao što je prikazano na slici 3.14. prijavljeni korisnici imaju mogućnost komentiranja događaja, korisnik ima mogućnost brisanja svog komentara, a komentirati se mogu svi događaji uključujući stanke za kavu i ručak.



Sl. 3.14. Zaslon pretplaćenog događaja

Sl. 3.15. Zaslon nepretplaćenog događaja

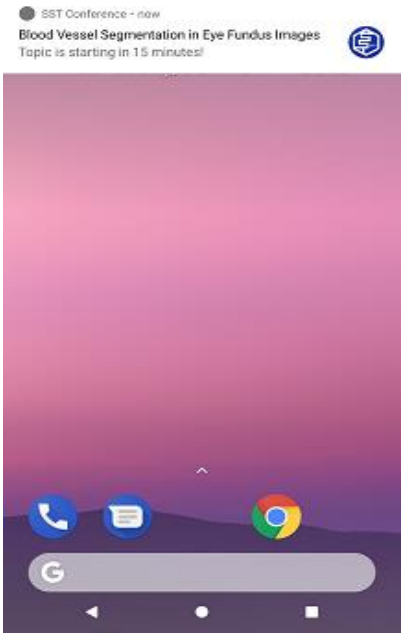
Dio koda zadužen za pretplatu i prekid pretplate na događaj prikazan je unutar programskog koda 3.3.

```
if (isChecked) {
    mViewModel.subscribeToTopic(mTopic);
    Date date = DateUtility.stringToIsoDate(mTrack.getStartDate());
    AlarmUtility.scheduleAlarm(mActivity, DateUtility.getReminderCalendarFromDate(date), mTopic.getId(),
EventAlarmReceiver.class);
} else {
    mViewModel.deleteTopicSubscription(mTopic);
    AlarmUtility.cancelAlarm(mActivity, mTopic.getId(), EventAlarmReceiver.class, null);
}
```

Programski kod 3.3. Dio koda zadužen za upravljanje pretplatom na događaj

Pritiskom na tipku ikonu „*bookmark*“ iz menija ovisno o njenom stanju izvršava se pretplata ili prekid pretplate, ako se korisnik pretplaćuje na temu postavlja se alarm koji će poslati obavijest 15 minuta prije početka događaja koja je vidljiva na slici 3.16. Ako korisnik prekida pretplatu tada se isti alarm gasi.

Svako predavanje ima vrijeme početka, ukoliko se korisnik pretplatio na predavanje i u postavkama je odlučio primati obavijesti aplikacije 15 minuta prije početka predavanja doći će mu obavijest o početku predavanja.



Sl. 3.16. Obavijest o početku predavanja

Kada se korisnik uspješno pretplatio na događaj ima mogućnost primanja obavijesti 15 minuta prije nego li događaj počne, također ima mogućnost pregleda svih pretplaćenih događaja i brisanja pretplate.

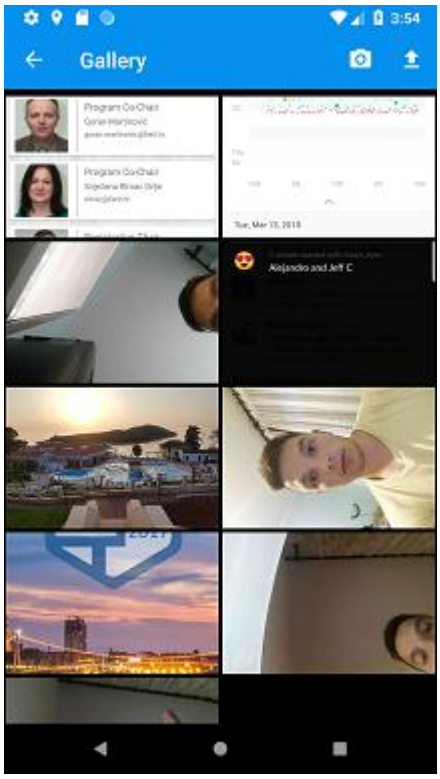
Sl. 3.17. Prikaz svih pretplaćenih događaja

Sl. 3.18. Uklanjanje pretplaćenog događaja

Ukoliko korisnik slučajno obriše pretplaćeni događaj imati će mogućnost u sljedećih par sekundi poništiti brisanje.

3.3.4. Učitavanje i pregled slika

Jedno od svojstva koje pruža aplikacija je galerija slika sa konferencije, prijavljenim korisnicima omogućava se učitavanje slika, dok će čak i oni u gost načinu rada imati mogućnost pregleda slika. Slike se učitavaju na *Firebase Storage* te se u bazu podataka sprema referenca na mjesto datoteke, zatim se pomoću biblioteke *Picasso* ista slika prikazuje u galeriji. *Picasso* omogućava spremanje slika stoga će jednom učitane slike biti dostupne čak i kada korisnik nema pristup internetu.



Sl. 3.19. Prikaz galerije slika

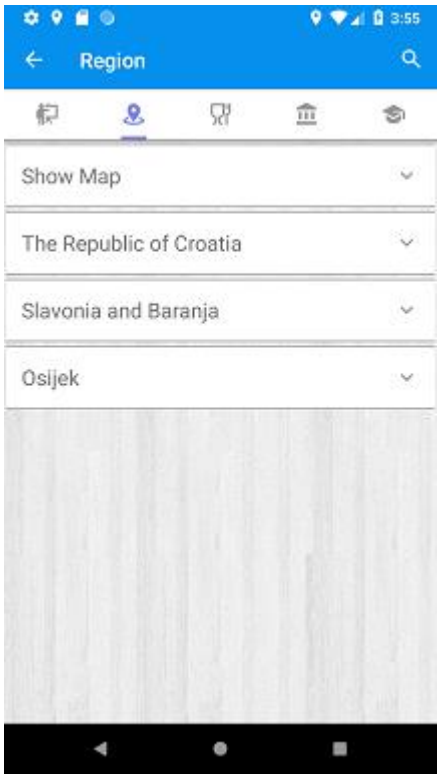
Sl. 3.20. Pregled slika iz galerije

Korisnici u galeriju mogu učitati sliku koja se već nalazi u telefonu ili mogu stvoriti novu koristeći kameru. Ukoliko se proces učitavanja prekine zbog loših uvjeta na mreži isti će se nastaviti kada se uvjeti poprave, osim u slučaju kada je aplikacija ugašena.

3.3.5. Prikaz područja od interesa na mapi

Koristeći lokaciju korisnika aplikacija ima mogućnost prikazivanja raznih područja interesa u korisnikovoj blizini, korisniku će se prikazati zadani sadržaj kao što su informacije o regiji, gradu u kojem se odvija konferencija, hotelu u kojem se drže predavanja, fakultetu te bilo koje druge informacije koje su pružene aplikaciji preko Web servisa.

Na slici 3.21. prikazano je kako izgleda zaslon kartice Regija, korisnik ima mogućnost pregleda zadanog sadržaja kao što su u ovom slučaju informacije o Republici Hrvatskoj, Slavoniji i Baranji te o gradu Osijeku. Na slici 3.22. vidljivo je kako te informacije izgledaju kada je riječ o gradu Osijeku.



Sl. 3.21. Prikaz sadržaja regije

Sl. 3.22. Prikaz informacija o gradu Osijeku

Klikom na „Show Map“ prikazati će se mapa sa označenim mjestima koja su pružena od strane Web servisa ili ukoliko je riječ o karticama „Hrana“ i „Sights“ prikazati će se mjesta koja je korisnik odabrao.

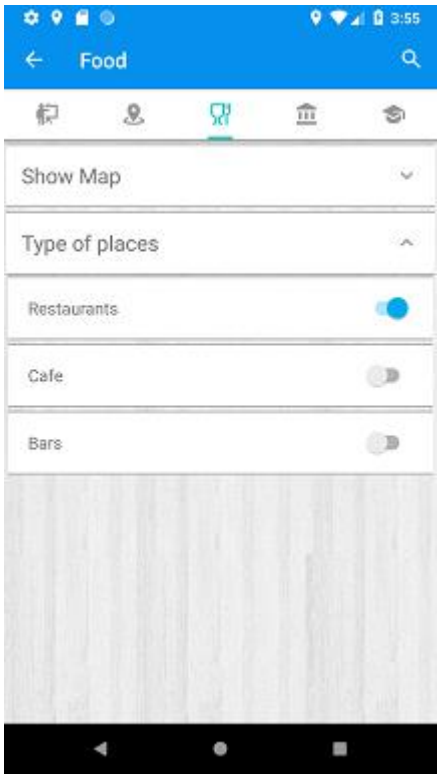
Osim zadanog sadržaja korisnik ima mogućnost pretraživanja restorana, kafića, muzeja, crkva, biblioteka i zooloških vrtova u svojoj blizini. Korisnik također ima mogućnost zadavanja radijusa pretrage, a kako bi koristio ovu opciju mora biti povezan s internetom. Pretraživanje mjesta u okolini izvršava se korištenjem servisa *Google Places*, kako bi aplikacija koristila ovu mogućnost potrebno ju je registrirati na Google Cloud Platformi te prijaviti za *Google Places API*. Za uspješno dohvaćanje obližnjih mjesta potrebna je lokacija na kojoj se želi pretražiti, radijus pretrage, tip objekta koji se pretražuje što je u SST Aplikaciji restoran, kafić, crkva, muzej, biblioteka i zoološki vrt te na kraju najbitnija stavka ovog poziva je API ključ aplikacije koji je generiran prilikom registracije na *Google Places*. Kada su prisutni svi potrebni parametri može se izvršiti upit za pretraživanje.

Sučelje API poziva za dohvaćanje okolnih objekata vidljivo je unutar programskog koda 3.4.

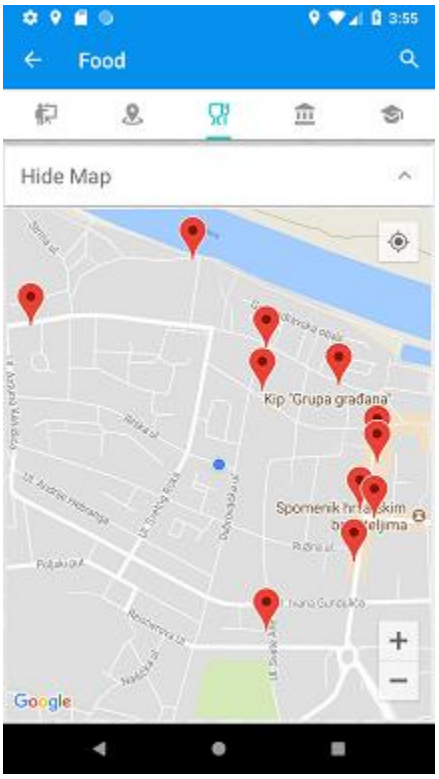
```
@GET("nearbysearch/json?")
Observable<NearbyPlaces> getNearbyPlaces(@Query("location") String location,
                                         @Query("radius") long radius,
                                         @Query("type") String type,
                                         @Query("key") String key);
```

Programski kod 3.4. Sučelja API poziva

Na slici 3.23. vide se opcije koje korisnik može uključiti na kartici „hrana“, korisnik može prikazati bilo koji broj opcija, bilo to jedna, niti jedna ili sve, a kako to izgleda na mapi za opciju restorani vidljivo je na slici 3.24.

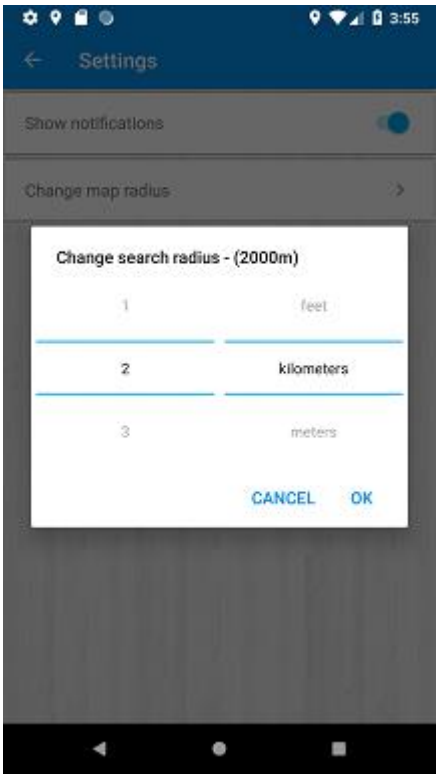


Sl. 3.23. Prikaz sadržaja kartice „hrana“



Sl. 3.24. Prikaz željenog sadržaja na mapi

Kao što je rečeno korisnik ima mogućnost biranja radijusa pretrage unutar koje želi pretražiti mjesta, a kako to izgleda vidljivo je na slici 3.25.



Sl. 3.25. Dijalog za promjenu radijusa pretrage

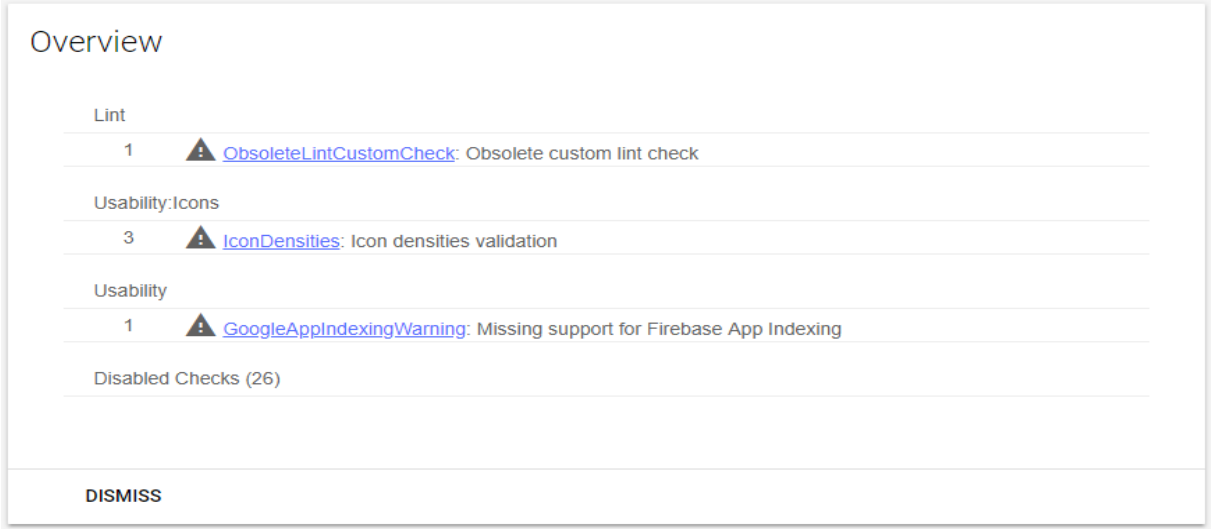
Radijus pretrage može se postaviti u metrima, kilometrima, miljama i *feet,* a Odabrana vrijednost automatski se preračunava u metre jer poslužitelj Google Places tako zahtjeva.

3.4. Osiguranje kvalitete programskog rješenja

Kvaliteta programskog rješenja osigurana je korištenjem metoda kao što su metode čistog koda, oblikovni obrasci te najnovije arhitekture MVVM koja je zajedno sa alatima iz *Architecture Components* riješila poznate probleme te omogućila jednostavno testiranje aplikacije.

3.4.1. Lint izvješće

Na slici 3.26 prikazano je *Lint* izvješće koje je generirano analiziranjem projekta, kao što je vidljivo izvješće nije idealno odnosno nije u potpunosti bez upozorenja, ali rezultat bez pogrešaka i samo 5 upozorenja je svakako zadovoljavajući. Prvo upozorenje odnosi se na biblioteku *ButterKnife* korištenu u projektu, drugo upozorenje vezano je za resurse koji nisu zastupljeni u svim veličinama te posljednje upozorava da nije omogućen *Firebase App Indexing* opisan u poglavlju 3.4.4.



Sl. 3.26. *Lint* izvješće aplikacije

3.4.2. JaCoCo izvješće

O kvaliteti projekta također govori i *JaCoCo* izvješće, kao što je objašnjeno u poglavlju 2.6.2. *JaCoCo* je alat koji se koristi kao mjerilo dobro testirane aplikacije. Aplikacija se sastoji od 74 jedinična testa i 138 instrumentacijska testa koji se pokreću na uređaju, ukupno dajući 212 automatskih testova, a postotak pokrivenosti koda testovima je 84%, što je zadovoljavajući rezultat ako se uzme u obzir da je riječ o postotku pokrivenosti kompletne aplikacije. Ključni dijelovi aplikacije kao što su pružatelji podataka (engl. *Data Source*) pokriveni su u potpunosti, odnosno 100%, dok je pokrivenost većine *ViewModela* iznad 90%. Dio koda nije testiran u potpunosti zbog korištenja *Firebase* servisa, postoje načini s kojima bi se i taj kod testirao, ali zahtjeva velike promjene u implementaciji spomenutog servisa, stoga je ostavljeno za buduće nadogradnje sustava. Na slici 3.27 prikazano je *JaCoCo* izvješće SST aplikacije. Na vrhu izvješća nalaze se paketi koji imaju najviše linija koda koje nisu pokrivene testovima, u slučaju SST aplikacije to su paketi *login* i *gallery* u kojima se koristi spomenuti *Firebase* servis koji ovakvom implementacijom nije moguće testirati. JaCoco izvješće pruža uvid u pokrivenost klasa, odnosno otvaranje klase unutar JaCoCo izvješće rezultirati će točnim prikazom koda koji je testiran, odnosno onog koji nije testiran – kod koji ima zelenu pozadinu je testiran, dok onaj sa crvenom pozadinom nije pokriven niti jednim testom uključenim u JaCoCo izvješće. Na takav način osoba koja piše test može jasno vidjeti koji dio koda treba testirati kako bi se podigao postotak pokrivenosti koda testovima.

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.example.karlo.sstconference.modules.login		79%		57%	40	123	66	310	16	82	0	9
com.example.karlo.sstconference.modules.gallery		75%		48%	35	83	66	251	14	52	0	6
com.example.karlo.sstconference.modules.venue.fragments		82%		46%	52	107	39	266	7	44	0	7
com.example.karlo.sstconference.ui		79%		67%	41	96	56	260	22	66	0	10
net.globulus.easyprefs		60%		n/a	12	39	34	90	12	39	0	1
com.example.karlo.sstconference.models.program		69%		100%	21	68	42	128	21	67	0	9
com.example.karlo.sstconference.modules.program		85%		56%	22	74	27	213	8	54	0	3
com.example.karlo.sstconference.modules.venue		85%		58%	18	85	35	173	8	66	0	5
com.example.karlo.sstconference.modules.program.fragments		93%		66%	39	134	18	345	4	75	1	10
com.example.karlo.sstconference.utility		91%		68%	26	105	27	289	9	70	0	8
com.example.karlo.sstconference.models.keynote		37%		n/a	11	22	32	51	11	22	0	2
com.example.karlo.sstconference.models		68%		75%	17	53	31	99	16	51	0	5
com.example.karlo.sstconference.modules.committee		88%		61%	22	70	22	181	7	47	0	6
com.example.karlo.sstconference.pager		80%		48%	17	36	13	73	2	18	0	4
com.example.karlo.sstconference.models.venue		73%		n/a	15	48	28	92	15	48	0	4
com.example.karlo.sstconference.adapters		92%		81%	13	62	11	162	9	50	0	12
com.example.karlo.sstconference.models.nearbyplaces		54%		n/a	16	34	24	51	16	34	0	4
com.example.karlo.sstconference.models.committee		44%		n/a	7	16	18	33	7	16	0	2
com.example.karlo.sstconference.modules.home		93%		83%	13	67	10	171	8	42	0	4
com.example.karlo.sstconference.modules.search		91%		66%	13	48	11	129	3	32	0	3
com.example.karlo.sstconference.service		74%		25%	7	12	10	46	1	6	0	1
com.example.karlo.sstconference.modules.subscribed		94%		73%	10	56	10	164	2	39	0	4
com.example.karlo.sstconference.modules.keynotespeakers		94%		70%	9	44	9	139	3	31	0	6
com.example.karlo.sstconference.commons		88%		n/a	5	23	7	33	5	23	1	3
com.example.karlo.sstconference.modules.chairs		93%		54%	6	28	7	97	2	22	0	3
Total	2.510 of 16.551	84%	330 of 836	60%	520	1.699	690	4.216	251	1.249	4	164

Sl. 3.27. *JaCoCo* izvješće o pokrivenosti aplikacije testovima

Najvažniji dijelovi aplikacije koje treba testirati su pružatelji podataka, *ViewModeli* ukoliko se koristi MVVM arhitektura, razne algoritme ili proračune ukoliko aplikacija sadrži iste, spremanje i dohvaćanje podataka iz baze podataka, grafičko sučelje, odnosno prikazuje li aplikacija krajnjem korisniku ono što bi trebala. Pošto SST aplikacija ne sadrži kompliciranе algoritme te su najbitniji dijelovi upravo pružatelji podataka i *ViewModeli* oni imaju najveći postotak pokrivenosti te su pokriveni jediničnim testovima. Kao što je već spomenuto postotak pokrivenosti pružatelja podataka je 100%, pružatelji podataka sadrže testove za dohvaćanje, spremanje te brisanje podataka. Testiranje pružatelja podataka vrlo je jednostavno korištenjem alata *Mockito* koji pruža lažne podatke, odnosno nudi mogućnost jasnog definiranja što će pojedini poziv na „lažnoj“ klasi vratiti pozivatelju.

Primjer testiranja prikazan je unutar programskog koda 3.5. Pomoću alata *Mockito* stvorene su lažne klase za dohvaćanje podataka iz baze podataka te za dohvaćanje podataka sa Web servisa te se nakon poziva pružatelju podataka provjerava je li isti pozvao odgovarajuće metode za dohvaćanje podataka iz spomenutih lažnih klasa. Na kraju testa za dohvaćanje podataka provjerava se je li sadržaj koji vraća pružatelj podataka jednak onome koji je definiran kao povratna vrijednost lažnih klasa za dohvaćanje podataka.

```
@Test
public void testGet() {
	List<Image> images = new ArrayList<>();
	for (int i = 0; i < 100; i++) {
		images.add(new Image(i,
		getStringFormat(IMAGE, i)));
	}
	List<Image> apiImages = new ArrayList<>(images);
	Image image = getImage(123);
	apiImages.add(image);

	when(dao.getImages()).thenReturn(Maybe.just(images));
	when(api.getImages()).thenReturn(Observable.just(apiImages));
	when(api.pushImageToServer("123", image)).thenReturn(Completable.complete());

	dataSource.getImages();
	verify(dao).getImages();
	verify(api).getImages();

	dataSource.getImages()
		.flatMap(Observable::fromIterable)
		.distinct(Image::getId)
		.toList()
		.subscribe(imageList -> {
			for (int i = 0; i < imageList.size(); i++) {
				assertEquals(imageList.get(i), apiImages.get(i));
			}
		});
}
```

Programski kod 3.5. Primjer testa za dohvaćanje iz baze podataka

Test spremanja prikazan unutar programskog koda 3.6 provjerava je li pozvana odgovarajuća metoda za spremanje u bazu podataka nakon pozivanja metode za spremanje podatka unutar pružatelja podataka.

```
@Test
public void testSave() {
	Image image = getImage(123);
	when(api.pushImageToServer("123", image)).thenReturn(Completable.complete());
	dataSource.insertOrUpdateImage(image);
	verify(api).pushImageToServer("123", image);

	dataSource.insertOrUpdateImage(image).subscribe(() ->
		verify(dao).insertImage(image));
}
```

Programski kod 3.6. Primjer testa za spremanje u baze podataka

Test brisanja prikazan unutar programskog koda 3.7 testira je li pozvana odgovarajuća metoda za brisanje iz baze podataka nakon pozivanja metode za brisanje podataka unutar pružatelja podataka.

```
@Test
public void testDelete() {
        Image image = getImage(123);

        dataSource.deleteImage(image);
        verify(dao).deleteImage(image);
}
```

Programski kod 3.7. Primjer testa za brisanje iz baze podataka

ViewModel se testira pružajući lažnu klasu pružatelja podataka, zatim se kreira lažnu klasa promatrača te se pretplaćuje na metodu za dohvaćanje podataka iz *ViewModela*, nakon poziva metode promatra se odgovaraju li podaci koje je pretplatnik primio onima koji su definirani kao povratnu vrijednost pružatelja podataka. Primjer koda za testiranje *ViewModela* prikazan je unutar programskog koda 3.8.

```
@Test
public void testGetChairs() {
        List<ConferenceChair> conferenceChairs = new ArrayList<>();
        for (int i = 0; i < 100; i++) {
                conferenceChairs.add(new ConferenceChair(i, TITLE, MAIL, FACILITY, IMAGE, NAME,
                NUMBER));
        }
        Observer observer = mock(Observer.class);
        when(dataSource.getConferenceChairs()).thenReturn(Observable.just(conferenceChairs));
        chairsViewModel.getChairs().observeForever(observer);
        verify(observer).onChanged(conferenceChairs);
}
```

Programski kod 3.8. Primjer testa za testiranje ViewModela

Baza podataka testira se tako što se u nju ubacuju podaci te se zatim isti dohvaćaju i provjeravaju jesu li jednaki ubačenim podacima. Za razliku od prijašnjih primjera jediničnih testova, za potrebe testiranja baze podataka potrebno je napisati jedinični test koji se pokreće na stvarnom Android uređaju. Test za ubacivanje i dohvaćanje objekta iz baze podataka prikazan je unutar programskog koda 3.9.

```
@Test
public void testInsertAndGetOne() {
        mDao.insertConferenceChair(getConferenceChair());
        mDao.getConferenceChairs()
                .toObservable()
                .flatMap(io.reactivex.Observable::fromIterable)
                .firstElement()
                .subscribe(chair -> {
                  assertEquals(chair.getName(), NAME);
                  assertEquals(chair.getChairTitle(), TITLE);
                  assertEquals(chair.getEmail(), MAIL);
                  assertEquals(chair.getFacility(), FACILITY);
                  assertEquals(chair.getPhoneNumber(), NUMBER);
                });
}
```

Programski kod 3.9. Primjer testa baze podataka

Iako sam test prikazan unutar programskog koda 3.9. izgleda kao jedinični test koji se pokreće na JVM-u, razlika je u definiranju *Espresso* pravila koje je potrebno kako bi se kreirala baza podataka koja se testira, primjer takvog pravila prikazan je unutar programskog koda 3.10.

```
@Before
public void initDb() {
        localDatabase = Room.inMemoryDatabaseBuilder(InstrumentationRegistry.getContext(),
        LocalDatabase.class)
                .build();
}
```

Programski kod 3.10. Primjer Espresso pravila

Programski kod 3.11 prikazuje primjer testa za korisničko sučelje koji provjerava je li traka za učitavanje prikazana na korisničkom sučelju kada bi treba biti te se uvjerava da nije prikazana onda kada ne bi trebala biti.

```
@Test
public void testProgressBar() {
        status.postValue(Status.loading(true));
        onView(Matchers.allOf(withId(R.id.progress_bar), isDescendantOfA(withId(R.id.venue_container))))
                .check(matches(isDisplayed()));

        status.postValue(Status.loading(false));
        onView(Matchers.allOf(withId(R.id.progress_bar), isDescendantOfA(withId(R.id.venue_container))))
                .check(matches(not(isDisplayed())));
}
```

Programski kod 3.11. Primjer testa korisničkog sučelja

4. ZAKLJUČAK

Namjera rada bila je razviti aplikaciju za pomoć posjetiteljima konferencije i osigurati kvalitetu razvijene programske podrške. U drugom poglavlju objašnjene su razne tehnike koje se koriste u razvijanju „dobrog“ programskog rješenja, tijekom razvoja korištene su upravo takve tehnike i alati koji su omogućili programski kod koji je lako testirati, koji je lako čitati i shvatiti, koji se lako može mijenjati, nadograđivati i što je najvažnije ovakvo programsko rješenje omogućava jednostavnije održavanje kompletnog sustava. Jedinu manu predstavlja dodatno vrijeme koje je potrebno uložiti tijekom postavljana temelja projekta, pogotovo ako programeri nisu upoznati sa tehnikama koje se koriste, ali kao što je već objašnjeno u radu kompletno vrijeme koje je dodatno uloženo tijekom postavljanja projekta i pisanja koda po nekim pravilima uglavnom se isplati već prvom prilikom kada nešto treba mijenjati u projektu, upravo iz razloga što će takav kod biti neopisivo lakše održavati nego kod koji je pisan bez jasno definiranih pravila.

Kao glavna ostvarenja treba spomenuti *Lint* izvješće razvojnog okruženja Android Studio koje je pokazalo izuzetno zadovoljavajuće rezultate sa samo 5 upozorenja te *JaCoCo* izvješće koje jasno pokazuje ukupnu pokrivenost programskog koda testovima od 84%. Upravo visok postotak pokrivenost govori o osiguranoj kvaliteti jer se jednostavnim pokretanjem testa jednostavno i brzo utvrđuje ispravnost pokrivenog koda.

Sustav je moguće dodatno nadograditi, tijekom razvoja korišten je modularan pristup vanjskim podacima, tako da se razvijeno rješenje može koristi i sa drugačijim podacima, odnosno može se iskoristi za nadolazeće konferencije. Jedna od opcija za iduću verziju programske podrške može biti omogućiti korisnicima međusobno slanje poruka unutar aplikacije, ubacivanje pretplaćenih događaja u korisnikov kalendar te podizanje pokrivenosti testovima na još višu razinu.

LITERATURA

[1] M. Lotz, Waterfall vs. Agile, Segue Technologies, https://www.seguetech.com/waterfall-vs-agile-methodology/, pristupljeno: 23. lipnja 2018.

[2] D. Hughey, The Traditional Waterfall Approach, UMLS, http://www.umsl.edu/~hugheyd/is6840/waterfall.html, pristupljeno: 23.lipnja 2018.

[3] M. Rajput, Agile Methodology for mobile app, Mind Inventory, https://www.mindinventory.com/blog/why-need-agile-methodology-for-mobile-app-development/, pristupljeno: 23. lipnja 2018.

[4] J. Watts-Roy, Kanban vs. Scrum, Number8, https://number8.com/kanban-versus-scrum/, pristupljeno: 23.lipnja 2018.

[5] R.C. Martin, Clean Code, Prentice Hall, Boston, 2009.

[6] S. Gupta, How to write clean code?, Medium, https://medium.com/mindorks/how-to-write-clean-code-lessons-learnt-from-the-clean-code-robert-c-martin-9ffc7aef870c, pristupljeno: 23. lipnja 2018.

[7] P. Szklarska, Design patterns in Android - Builder, Medium, https://medium.com/@pszklarska/android-design-patterns-in-practice-builder-6b044f83e6e9, pristupljeno: 23. lipnja 2018.

[8] P. Szklarska, Design patterns in Android - Adapter, Medium, https://medium.com/@pszklarska/design-patterns-in-android-adapter-875538c343c3, pristupljeno: 23. lipnja 2018.

[9] E. Gamma, R. Helm, R. Johnson i J.M. Vlissides, Design, Addison-Wesley, USA, 1995.

[10] J. Bloch, Effective Java Third Edition, Pearson Education Inc., USA, 2018.

[11] A. Mandliya, Composite Design Pattern in Java, DZone, https://dzone.com/articles/composite-design-pattern-java-0, pristupljeno: 23. lipnja 2018.

[12] M. Goswami, Fascade Design Pattern, Java Code Geeks, https://www.javacodegeeks.com/2012/11/facade-design-pattern-design-standpoint.html, pristupljeno: 23. lipnja 2018.

[13] J. Kulandai, Command Desing Pattern, Java Papers, https://javapapers.com/design-patterns/command-design-pattern/, pristupljeno: 23. lipnja 2018.

[14] P. Szklarska, Design patterns in Android - Observer, Medium, https://medium.com/@pszklarska/design-patterns-in-android-observer-4b7622fa678b, pristupljeno: 23. lipnja 2018.

[15] Design Patterns – Strategy Pattern, Tutorials Point, https://www.tutorialspoint.com/design_pattern/strategy_pattern.htm, pristupljeno: 23. lipnja 2018.

[16] V. Kumar, Android Architecture Patterns, Medium, https://medium.com/@vicky7230/android-architecture-patterns-mv-c-p-vm-4594574eeaa1, pristupljeno: 23. lipnja 2018.

[17] F. Muntenescu, Android Architecture Patterns Part 1: MVC, Medium, https://medium.com/upday-devs/android-architecture-patterns-part-1-model-view-controller-3baecef5f2b6, pristupljeno: 23. lipnja 2018.

[18] Impaler, Building Games Using MVC Pattern, Java Code Geeks, https://www.javacodegeeks.com/2012/02/building-games-using-mvc-pattern.html, pristupljeno: 23.lipnja 2018.

[19] R. Soral, Architectural Guidelines to follow for MVP pattern, Android Pub, https://android.jlelse.eu/architectural-guidelines-to-follow-for-mvp-pattern-in-android-2374848a0157, pristupljeno: 23. lipnja 2018.

[20] A.Wary, Updating Settings Activity With MVP, Fossasia, https://blog.fossasia.org/updating-settings-activity-with-mvp-architecture-in-susi-android/, pristupljeno: 23.lipnja 2018.

[21] F. Muntenescu, Android Architecture Patterns Part 3: MVVM, Medium, https://medium.com/upday-devs/android-architecture-patterns-part-3-model-view-viewmodel-e7eeee76b73b, pristupljeno: 23. lipnja 2018.

[22] A. Ahani, Intro to the MVVM Design Pattern, DZone, https://dzone.com/articles/model-view-viewmodel-design, pristupljeno: 23.lipnja 2018.

[23] J. Sonmez, Software Testing Basics, Usersnap, https://usersnap.com/blog/software-testing-basics/, pristupljeno: 23.lipnja 2018.

[24] What is automated testing?, Smart Bear, https://smartbear.com/learn/automated-testing/what-is-automated-testing/, pristupljeno: 23.lipnja 2018.

[25] J. Marsh, What can a QA team do for you? Upwork, https://www.upwork.com/hiring/development/what-can-a-qa-team-do-for-you/, pristupljeno: 23. lipnja 2018.

[26] Test your app, Developers, https://developer.android.com/studio/test/, pristupljeno: 23. lipnja 2018.

[27] Why Android Testing?, Guru99, https://www.guru99.com/why-android-testing.html, pristupljeno: 23.lipnja 2018.

[28] B.C Zapata, A.H. Ninirola, Testing and Securing Android Studio Applications, Packt Publishing, Birmingham, 2014.

[29] I. Klimov, Android Testing Instructions, My Hexaville, http://myhexaville.com/2017/12/18/android-testing-introduction/, pristupljeno: 23.lipnja 2018.

[30] NguyenDT4, Android Testing Strategy, FPT Tech, https://tech.fpt.com.vn/language/en/getting-started-with-android-testing-chapter-2-android-testing-strategy/, pristupljeno: 23. lipnja 2018.

[31] L. Shao, Top 5 Android Testing Frameworks, BitBar, https://bitbar.com/top-5-android-testing-frameworks-with-examples/, pristupljeno: 23. lipnja 2018.

[32] Introduction to Appium, Appium, http://appium.io/docs/en/about-appium/intro/?lang=en, pristupljeno: 23. lipnja 2018.

[33] Calabash Automation Tool Tutorial, Guru99, https://www.guru99.com/calabash-android-ios-testing.html, pristupljeno: 23. lipnja 2018.

[34] R. Ajesh, Espresso Basics, Medium, https://medium.com/mindorks/android-testing-part-1-espresso-basics-7219b86c862b, pristupljeno: 23.lipnja 2018.

[35] R. Reda, Robotium – Testing Android User Interface, Methods & Tools, http://www.methodsandtools.com/tools/robotium.php, pristupljeno: 23. lipnja 2018.

[36] Software Testing as a Service, Guru99, https://www.guru99.com/what-is-testing-as-a-service-taas.html, pristupljeno: 23. lipnja 2018.

[37] What is software Quality Assurance?, Test Institute, https://www.test-institute.org/What_is_Software_Quality_Assurance.php, pristupljeno: 23. lipnja 2018.

[38] J. Hildenbrand, What is Android?, Android Central, https://www.androidcentral.com/what-android, pristupljeno: 23. lipnja 2018.

[39] Dr. Droid, Android Architecture, Android Hub, http://www.androhub.com/android-architecture/, pristupljeno: 23. lipnja 2018.

[40] Android – Application Components, Tutorials Point, https://www.tutorialspoint.com/android/android_application_components.htm, pristupljeno: 23. lipnja 2018.

[41] Android Life Cycle of Activity, Java Point, https://www.javatpoint.com/android-life-cycle-of-activity, pristupljeno: 23. lipnja 2018.

[42] J. Singh, Android Architecture Components, Android Pub, https://android.jlelse.eu/android-architecture-components-a563027632ce, pristupljeno: 23. lipnja 2018.

[43] Architecture Components, Google Developer Training, https://google-developer-training.gitbooks.io/android-developer-advanced-course-concepts/unit-6-working-with-architecture-components/lesson-14-architecture-components/14-1-c-architecture-components/14-1-c-architecture-components.html#chapterstart, pristupljeno: 23. lipnja 2018.

[44] L. Fujiwara, ViewModels : A Simple Example, Medium, https://medium.com/google-developers/viewmodels-a-simple-example-ed5ac416317e, pristupljeno: 23. lipnja 2018.

[45] J. Lincoln, Everything you need to know about Google Firebase, Ignite Visibiltiy, https://ignitevisibility.com/everything-need-know-google-firebase/, pristupljeno: 23. lipnja 2018.

[46] V. Thakor, Create project in Firebase, Java Query, https://www.javaquery.com/2016/09/how-to-create-project-in-firebase.html, pristupljeno: 23. lipnja 2018.

SAŽETAK

Kao što sama tema diplomskog rada kaže fokus je bio na osiguranju kvalitete programske podrške na Android platformi. Kvaliteta programske podrške ostvaruje se upotrebom metoda kao što su Čisti kod te korištenjem raznih oblikovnih obrazaca koji rješavaju česte probleme tijekom razvoja programske podrške. Osim metoda koje pridonose urednosti programskog koda, važno je odabrati kvalitetnu arhitekturu koja će predstavljati čvrste temelje za kvalitetnu aplikaciju koju je lako testirati. Za osiguranje kvalitete potrebno je testirati programsko rješenje, a u tu svrhu koriste se razni automatski testovi. Primjer osiguranja kvalitete programske podrške prikazan je tijekom izrade programskog rješenja za pomoć posjetiteljima konferencije, programsko rješenje implementira spomenute metode i dobre programske prakse. Odabrana je arhitektura *Model-View-ViewModel* koja trenutno predstavlja arhitekturu koja nudi najviše rješenja za poznate probleme tijekom razvoja aplikacije za Android operacijski sustav. Osim čistog koda kvaliteta se osigurala velikim brojem testova, aplikacija se sastoji od 74 jedinična testa i 138 instrumentacijska testa koji se pokreću na uređaju ukupno dajući 212 automatskih testova i pokriće programskog koda sa testovima od 84%.

Ključne riječi:

Android, Architecture Components, Automatsko testiranje, MVVM, Testni okviri

ABSTRACT

As the topic of this graduate thesis says, focus was on providing quality software support on Android platform. The quality of software is achieved through use of methods such as clean code and various design patterns that solve common problems during the software development. In addition to methods that contribute to the integrity of the software it is important to select a high-quality architecture pattern that will provide a solid foundation for a high-quality and easy-to-test application. For quality assurance it is necessary to test the program code for any faults, for this purpose various automatic tests were written. An example of the quality assurance software support is presented during the development of a program solution for conference visitors, the software solution implements methods and good programming practices. Model-View-ViewModel architecture pattern was selected for project foundation, selected architecture is currently offering the most of all knowing architecture patterns for Android platform. In addition to the clean code the quality is ensured by a large number of tests, the application consists of 74 unit tests and 138 instrumented tests which are driven on the Android device, giving a total of 212 automatic tests and code coverage of 84%.

Keywords:

Android, Architecture Components, Automated testing, MVVM, Testing framework

ŽIVOTOPIS

Karlo Žnidarec je rođen 18.11.1994. u Našicama. Osnovnu školu je upisao 2001. godine u školi Josipa Jurja Strossmayera u Đurđenovcu koju je završio 2009. godine. Nakon završene osnovne upisao je srednju tehničku školu, smjer tehničar za elektroniku u srednjoj školi Isidora Kršnjavog u Našicama iste godine, srednju školu završava 2013. godine. Nakon završene srednje škole upisao je preddiplomski studij smjer elektrotehnika na Elektrotehničkom fakultetu u Osijeku 2013. godine. Nakon završetka preddiplomskog studija 2016. godine postaje Sveučilišni prvostupnik te upisuje diplomski studij smjer mrežne tehnologije na Fakultetu elektrotehnike, računarstva i informacijskih tehnologija u Osijeku. 2018. godine uručeno mu je priznanje za uspješnost u studiranju.

KARLO ŽNIDAREC

_____

PRILOZI

1. Osiguranje kvalitete programske podrške na Android platformi.docx
2. Osiguranje kvalitete programske podrške na Android platformi.pdf
3. Programski kod