Arhitektura Android aplikacija

Babić, Filip

Undergraduate thesis / Završni rad

2018

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **Josip Juraj Strossmayer University of Osijek, Faculty of Electrical Engineering, Computer Science and Information Technology Osijek / Sveučilište Josipa Jurja Strossmayera u Osijeku, Fakultet elektrotehnike, računarstva i informacijskih tehnologija Osijek**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:200:461983

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-04-19**

Repository / Repozitorij:

Faculty of Electrical Engineering, Computer Science and Information Technology Osijek

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

Sveučilišni studij

ARHITEKTURA ANDROID APLIKACIJA

Završni rad

Filip Babić

Osijek, 2018.

Sadržaj

1. UVOD 3

1.1. Zadatak završnog rada 4

2. ARHITEKTURNI OBRASCI ZA RAZVOJ MOBILNIH APLIKACIJA 5

2.1. Povijest Android operativnog sustava 5

2.2. Programski jezici Java i Kotlin 6

2.3. Dobre i loše prakse pri razvoju programske podrške 6

2.4. Naivan pristup razvoju aplikacije 9

2.5. Model-Pogled-Prezenter arhitekturni obrazac 10

2.6. Model-Pogled-Model Pogleda arhitekturni obrazac 11

2.7. Testiranje aplikacija 13

2.8. Biblioteke potrebne za razvoj aplikacije 13

3. OBLIKOVANJE ANDROID APLIKACIJE 15

3.1. Postavljanje temeljnog dijela projekta 16

3.2. Odabrane ovisnosti 17

3.3. Kreiranje potrebnih usluga 18

3.4. Naivna implementacija 22

3.5. MVP implementacija 25

3.6. MVVM implementacija 29

4. ZAKLJUČAK 36

LITERATURA 37

SAŽETAK 40

ABSTRACT 41

ŽIVOTOPIS 42

PRILOZI.. 43

1. UVOD

Android operacijski sustav je prisutan već gotovo 10 godina. Kroz sve te godine kvaliteta Android aplikacija je dovedena na visoke razine, a svemu tome su zaslužni detaljno razrađen i testiran arhitekturni temelj, moderni alati i ekosustav razvoja. U svijetu strukturiranja koda, od klasičnog pristupa s Model-Pogled-Upravitelj (engl. *Model-View-Controller, MVC*) oblikovnim obrascem do ideja poput arhitekture koja je upravljana porukama (engl. *event-driven architecture*), teško se pronalazi jedino pravo rješenje i izvor istine. Jednako teško se pronalaze pravi alati i biblioteke za razvoj aplikacija kojima je olakšano povezivanje s mrežnim sučeljem, spremanje i korištenje podataka na uređaju, povezivanje korisničkog sučelja s programskom logikom i mnogi drugi procesi svakodnevnog rada s aplikacijama. U ovom završnom radu bit će obrađene teme poput arhitekturnih obrazaca, strukturiranja koda i najboljih praksi pri razvoju programske podrške. Na primjeru jednostavne aplikacije bit će pokazane sve navedene teme.

U drugom poglavlju ovog završnog rada je opisana povijest razvoja Android operativnog sustava, popularnih alata i biblioteka u zajednici te debata o odabiru jezika za razvoj same platforme i aplikacija. Razrađeni su neki od ključnih arhitekturnih principa za razvoj proizvodnih, proširivih i skalabilnih aplikacija, koje se mogu testirati. Uspoređeni su programski jezici Java i Kotlin te su prikazane najbolje prakse koje se koriste u zajednici za razvoj aplikacija, kako su one nastale i tko ih je osmislio. Također je kratko spomenut proces i ideja o testiranju programske podrške. U trećem poglavlju oblikovana je Android aplikacija s točno određenim skupom zahtjeva koji su u industriji razvoja mobilnih aplikacija najčešći. Na primjeru tri principa razvoja programske podrške prikazan je proces zamišljanja i stvaranja koncepta programskog rješenja te njegovo izvršenje. Ukratko su objašnjene korištene biblioteke kao i njihovo djelovanje, a prikazan je i standardan pristup korištenju odabranih biblioteka te njihovo ponašanje u aplikaciji. Osim toga, na svaku inačicu aplikacije dan je osvrt, analizirajući i uspoređujući rješenja za brojne probleme s kojima se programeri svakodnevno susreću. U zadnjem poglavlju vidljiv je zaključak izveden iz usporedbe inačica aplikacije, razvojnih alata i same ideje za aplikaciju. Dodatno je objašnjeno na koji način se projekt može proširiti, poboljšati, uljepšati i kako mu je moguće dati vrijednost za korisničku uporabu. Navedene su dodatne ideje za obogaćivanje projekta s tehničke strane, kako bi budući Android programeri mogli dobili bolju i raznovrsniju podlogu za početak učenja.

1.1. Zadatak završnog rada

U teorijskom dijelu rada potrebno je opisati trenutne trendove u razvoju mobilnih aplikacija za Android platformu. Poseban naglasak staviti na arhitekturu aplikacije i korištene oblikovne obrasce s njihovim prednostima, nedostacima i bibliotekama koji ih podržavaju. U praktičnom dijelu rada programski ostvariti usporedbu arhitekturnih stilova primjenjivih pri razvoju programske podrške za Android platformu, uključujući one koji se ne oslanjaju na oblikovne obrasce te prikazati prednosti i nedostatke pojedinih.

2. ARHITEKTURNI OBRASCI ZA RAZVOJ MOBILNIH APLIKACIJA

U ovom dijelu će se prikazati neki od glavnih obrazaca razvoja mobilnih aplikacija u današnjoj industriji i koje su dobre i loše prakse u razvoju programske podrške. Osim toga će se ukratko opisati povijest Android operativnog sustava, kakav je ekosustav Android zajednice, koji je razlog uspješnosti Android operativnog sustava te koje su glavne značajke istog. Ukratko će se usporediti značajke programskih jezika koji se koriste u razvoju Android aplikacija.

2.1. Povijest Android operativnog sustava

Android operacijski sustav su osnovali Andy Rubin, Rich Miner, Nick Sears i Chris White. Glavne značajke tog operativnog sustava bile su fokus na razvoj platforme u stilu otvorenog koda (engl. *open-source*), omogućivanje detaljnog prilagođavanja sustava, sučelja i njihovog rada svakoj osobi i proizvođaču pojedinačno te širok spektar opcija sustava i uređaja na kojima je isti instaliran [1]. Nakon što ih je Google otkupio u fazi razvojne tvrtke (engl. *startup*), Android se masivno počeo širiti na tržištu, zauzimajući danas 77% cjelokupnog tržišta pametnih telefona [2].

S tehničke strane je utemeljen na inačici Linux operativnog sustava, gdje svaka aplikacija predstavlja jednog korisnika koji je trenutno "prijavljen" u sustav. Aktivne aplikacije su pokrenute u nečemu što se zove Dalvik virtualni stroj (engl. *Dalvik Virtual Machine*, DVM), koji je pak zasnovan na Java virtualnom stroju (engl. *Java Virtual Machine*, JVM). Samim time je vidljivo da su podržani razvojni jezici oni koji se također baziraju na JVM-u, poput Jave, Scala-e, Groovy-ja i Kotlin-a.

Do sada je razvijeno 8 velikih Android verzija, sa svojim podverzijama, od kojih je zadnja, najnovija, Android Pie ili Android 9.0 [3]. Imenovanje Android verzija je napravljeno tako da svaka verzija predstavlja jedno slovo u engleskoj abecedi, počevši od slova "C", gdje su slova "A" i "B" označavali alpha i beta verziju. Osim što predstavljaju slova abecede, svaka verzija, prateći to slovo, predstavlja i jedan slatkiš, poput slova "O" koji predstavlja Oreo keks, ili slova P koji predstavlja pitu (engl. *pie*).

Kao i svaki operacijski sustav, Android ima svoje probleme. Najveći problem trenutno je održavanje regularnog ažuriranja verzija sustava s proizvođačke strane. Naime, obzirom da iza Android sustava stoji ideja otvorenog koda, mnogi proizvođači pametnih telefona su to uzeli zdravo za gotovo, kreirajući svoje, gotovo neprepoznatljive,

inačice Androida. Na taj način mogu ograničiti svoje uređaje s verzijama Androida, zbog čega su kupci prisiljeni često mijenjati svoje uređaje za novije modele.

Zbog tog problema dolazi se do činjenice kako se druga trenutno najnovija verzija Androida (8.0) nalazi na svega 12% uređaja, što najveći problem stvara programerima, koji su primorani podržavati vrlo stare verzije Androida i koristiti ograničen skup mogućnosti iako je sustav, dolaskom svake nove verzije, proširen s brojnim drugim značajkama programske podrške i sklopovlja.

2.2. Programski jezici Java i Kotlin

Iako Android zbog DVM-a podržava sve programske jezike temeljene na JVM-u, trenutno najzastupljeniji programski jezici na Androidu su Java i Kotlin. Oba jezika pružaju objektno-orijentirane paradigme i mogućnosti poput čvrstog, statičnog sustava tipova, brzine i mogućnosti vrlo preciznog rada s višenitnosti, mrežom i bazama podataka. Java godinama bila prvi izbor, ali zbog potrebe za optimizacijom izrade mobilnih aplikacija, lakoćom rada s podacima, proširivosti i zbog pojave modernih jezikovnih značajki poput lambda funkcija, ništavnih tipova (engl. *nullable types*) i tokova podataka (engl. *streams*), Kotlin polako preuzima titulu vodećeg jezika za razvoj Android aplikacija.

Sama Android platforma je pisana u Javi, ali povezivanje s Kotlinom nije problem, jer su Kotlin i Java u potpunosti interoperabilni. To znači da se Java kod može pokretati u Kotlin datotekama i obrnuto, bez dodatnog troška memorije ili vremena. Samim time aplikacije ne moraju biti u potpunosti prepisane u Kotlin, već se mogu prevoditi u koracima.

Najvažnija značajka Kotlina je ta što njegove mogućnosti ne ovise o verziji Jave koja se koristi u projektima, kao ni o verziji Androida koja se podržava. Što je suprotan slučaj od Jave. Naime verzija Jave u projektu ovisi o najmanjoj podržanoj verziji Androida. Nijedna novija verzija, tj Java 8 do Java 10, nije podržana prije Android Nougata (7.0), odnosno svega 43% uređaja podržava nove prevoditelje za JVM.

2.3. Dobre i loše prakse pri razvoju programske podrške

U svakom poslu, tako i u razvoju aplikacija, postoje smjernice koje su zamišljene i kojih bi se svaka osoba trebala držati u svrhu održavanja standarda i kvalitete krajnjeg proizvoda. Iako se govori o apstraktnim stvarima, aplikacijama, i takav proizvod može

biti loše i dobre kvalitete. Ovdje će biti opisane prakse koje pomažu pri održavanju kvalitete razvojnog koda, poput pravilnog imenovanja funkcija i varijabli, razdvajanja koda na logičke strukture, skraćivanja koda i smanjenja broja ponavljanja koda.

Najbitnija stavka kod dobrih praksi je proces i stil imenovanja varijabli i funkcija u kodu. Vrlo čest pristup jest taj da se imena funkcija i varijabli što više skraćuju, pod izlikom da se time štedi vrijeme na pisanje koda. Nažalost, taj pristup je sasvim suprotan od onoga što se stvarno treba raditi. Ako se skraćuju kod i imena do te mjere da iz njih ne može biti razaznana prava funkcionalnost, vrijeme neće biti uštađeno, dapače, u budućnosti kad bude potrebno ponovno posjetiti taj kod, bit će potrebno puno više vremena da se shvati poslovna logika nego što bi bilo da se varijable i funkcije nazovu tako da predstavljaju ono što one rade. Dobar primjer ove rasprave je dan slikom 2.1.

```
public static List<User> filter(List<User> u) {
    List<User> f = new ArrayList<>();

    for (User x : u) {
        if (x.a() < 18) {
            f.add(x);
        }
    }
    return f;
}


public static List<User> filterUnderageUsers(List<User> allUsers) {
    List<User> underage = new ArrayList<>();

    for (User user : allUsers) {
        if (user.getAge() < 18) {
            underage.add(user);
        }
    }
    return underage;
}
```

Slika 2.1: Pravilno imenovanje u kodu

Dana funkcija služi za filtriranje maloljetnih korisnika iz liste svih korisnika. Gledajući prvu inačicu, jasno je kako je kod vrlo kratak. Ali da nije dostupna informacija da ova funkcija filtrira maloljetnike, vrlo teško bi bilo shvatiti njenu ulogu zbog imena varijabli i same funkcije. Jedina vidljiva stvar je broj 18, koji može označavati puno stvari, ne samo maloljetnost.

Sasvim suprotna inačica te funkcije prikazana je na slici 2.1, ispod prve funkcije. Njeno ime je *filterUnderageUsers* čime je odmah objašnjena uloga iste. Nadalje, alocira se varijabla *underage* koja označava maloljetne osobe i prolaskom kroz sve korisnike koje predstavlja parametar *allUsers* traže se oni koji imaju manje od 18 godina. Ovom inačicom funkcije jasno, kratko i čisto je izrečena njena zadaća i nikad neće doći do nedostatka informacija. Time je dokazano kako su kratka imena, koja "štede" vrijeme,

zapravo vrlo nepraktična i uzmu više vremena za razumijevanje nego što bi se potrošilo na pravilno i jasno imenovanje.

Navedeni primjer s imenovanjem, kao i smjernice poput organizacije i strukture koda, u srži prate skup principa, opisanih u [4]. koji se nazivaju SOLID principi razvoja programske podrške. Svako slovo od kojeg se SOLID sastoji predstavlja jednu važnu smjernicu u pisanju koda koja, ako se prati, osigurava čist, proširiv i siguran kod.

Kao primjer, slovo "S" predstavlja princip jedne odgovornosti (engl. *Single Responsibility Principle, SRP*), koji nalaže da svaka logička jedinka u kodu mora biti zadužena samo za jednu stvar. Gledajući prikazanu *filterUnderageUsers* funkciju, njena jedina odgovornost je da filtrira korisnike koji su maloljetni. Unutar iste funkcije se nalazi varijabla *underage* čija je odgovornost držanje svih maloljetnih korisnika i parametar *allUsers* čiji je zadatak držati sve korisnike. Samim time je ta cjelokupna jedinka , funkcija s varijablom i parametrom, prožeta SRP-om.

Ostali principi SOLID-a će biti prikazani kroz ostatak rada i kroz praktični dio gdje će se svi SOLID principi i koristiti za razvoj. Osim SOLID principa, postoji i smjernica čiste arhitekture (engl. *clean architecture*). Te principe i čistu arhitekturu je prvi postavio Robert Cecil Martin u svojim knjigama *Clean Architecture* [5] i *Clean Code* [6]. Osim što je osnivač čiste arhitekture, veliki zagovornik razvoja upogonjenog testovima (engl. *Test Driven Development*), jedan je od osnivača agilnog manifesta (engl. *Agile Manifesto*). Često je zbog svojih inspiracijskih govora, konzultiranja i brojnih knjiga napisanih u svrhu povećavanja standarda aplikacijskog koda, nazivan kao i ujak čistog programiranja, dajući mu nadimak "Uncle Bob".

Uncle Bob je došao na ideju o Clean arhitekturi i SOLID principima kroz svoje osobno iskustvo. Razvijajući godinama proizvodne aplikacije za brojne velike klijente, često je bio primoran raditi za tzv. naslijeđene projekte (engl. *legacy projects*). To su projekti koji postoje već dugo, čiji je kod vrlo loše napisan, neodržavan i na kojima je radilo puno ljudi, svatko naravno koristeći svoj stil pisanja, umjesto jedan zajednički.

Nakon što mu se puno ljudi javilo s problemima vezanim uz projekte nasljedstva i s problemima održavanja novih projekata, on je napisao službeno dokumentaciju u obliku smjernica za razvoj svih vrsta aplikacija. O njemu i njegovom radu je više opisano u [7].

2.4. Naivan pristup razvoju aplikacije

Često, u programerskom žargonu, je korišten izraz “naivno rješenje” ili “naivna implementacija” [8]. Taj pristup rada označava pisanje koda na način gdje se ne očekuje previše promjena ili proširenja, gdje je kod napisan da radi, bez previše mogućnosti mijenjanja. To je u srži vrlo loše, jer su programska podrška, alati, platforme, programski jezici i u širokom smislu tehnologija, stvari koje su najpodložnije brzim promjenama u ovom modernom svijetu.

Zbog toga ime ovom pristupu odgovara, jer je vrlo naivno očekivati od nečega uvijek-mijenjajućeg da se neće promijeniti. Na tehničkoj strani, naivan kod je onaj gdje se sve odgovornosti i sva poslovna logika nalaze na jednom mjestu, bez vidljivo odvojenih struktura. Ako je taj opis primjenjen na arhitekturu aplikacije, on označava takvu organizaciju aplikacije u kojoj organizacije gotovo pa i nema.

U praksi je takav pristup namjerno korišten vrlo rijetko, ali kad se pojavljuje, to gotovo uvijek bude u ista dva slučaja, kod početnika i kod naslijeđenih projekata. Kod početnika se pojavljuje jer početnici nemaju još razvijen smisao ili osjećaj za strukturu, dok se kod projekata nasljedstva pojavljuje zato što su takvi projekti nekoliko godina stari, puno ljudi je radilo na njima pa nikad nije bio održavan jednoznačan stil pisanja koda ili je sam projekt bio razvijan od strane početnika.

Najčešće značajke ovog pristupa su prazne, neiskorištene ili pak zakomentirane funkcije u kodu, funkcije koje broje nekoliko stotina linija koda ili klase koje broje nekoliko tisuća, brojni komentari iznad ili unutar funkcija, koji pokušavaju objasniti na koji način je funkcija izvršena pri pozivu i usko povezivanje klasa (engl. *tight coupling*) [9] gdje, ako se nešto odluči promijeniti, mora se ujedno mijenjati jako puno koda koji je povezan s tim.

Sve navedeno otežava rad na projektu, stvara misaoni teret (engl. *cognitive load*) [10] i može potencijalno prouzrokovati gomilu grešaka u aplikaciji, ako se odluči nešto, što se sasvim ne razumije, promijeniti.

Postoji ipak slučaj gdje je naivna implementacija najbolji izbor. To je kod jednostavnih dijelova aplikacije, za koje se apsolutno zna da nisu promjenjivi ili koji nemaju velike odgovornosti. Primjer je kod zaslona u Android aplikacijama gdje se prikazuju Internet stranice pomoću elementa mrežnog pogleda (engl. *WebView*). Jedina stvar za koju su zaduženi takvi zasloni su prikaz točno određenih stranica. Previše bi troška bilo stvoreno da se takvi zasloni izrađuju prateći neki od arhitekturnih obrazaca.

To naravno ne znači da, dok se pišu, ne treba pratiti dobre prakse navedene u završnom radu, poput dobrog imenovanja funkcija i varijabli i općenito organizacije koda, ali sve više od toga stvara veći trošak nego što je dobit.

2.5. Model-Pogled-Prezenter arhitekturni obrazac

Kao što je već navedeno, naivan pristup je vrlo loš, znajući da će svaka aplikacija barem jednom u budućnosti biti ažurirana ili dorađena. Iz tog razloga su proizašli brojni programski oblikovni obrasci, a naposljetku iz njih i arhitekturni obrasci poput MVC obrasca, koji je pojašnjen u [11]. Trenutno mjesto najzastupljenijeg arhitekturnog obrasca u Androidu zauzima Model-Pogled-Prezenter (engl. *Model-View-Presenter,* MVP) zbog jednostavnosti, prilagodljivosti i brzine postavljanja.

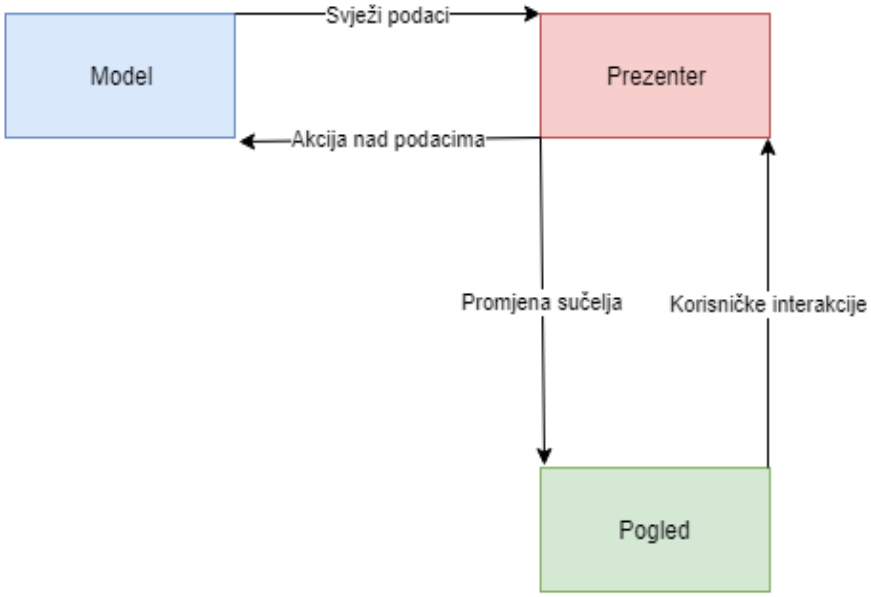
Vrlo je sličan MVC-u, aplikacija je odvojena na tri sloja, na model sloj, sloj pogleda i sloj prezentacije ili poslovne logike. Svaki sloj je precizno definiran, njegove zadaće su usko ograničene kako bi se izbjeglo usko vezanje ovisnosti u aplikaciji, a međusobno su povezani preko slabe reference (engl. *weak reference*) [12], najčešće preko Java ili Kotlin sučelja.

Sloj *modela* je korisniku najdalji i najskriveniji, predstavlja vanjske entitete koji komuniciraju s uslugama poput baze podataka ili mrežnog povezivanja i bavi se pretvaranjem podataka iz jednog tipa u drugi, kako bi sloju koji prikazuje i popunjava sučelje korisniku dao čiste podatke za rad. Sloj *modela* može dodatno biti raščlanjen na podslojeve, granularizirajući strukturu još više.

Sloj *prezentacije* ili sloj *poslovne logike* se bavi povezivanjem korisničkih interakcija unutar aplikacije, poput klikova ili pisanja, sa slojem *modela*, koji dalje dohvaća podatke, pretvara ih u korisne strukture i na kraju vraća sloju *prezentacije* koji ih prosljeđuje sloju *pogleda*. Zadaća sloja *prezentacije* je i provjera primljenih podataka te reagiranje na neispravne podatke i stanja u aplikaciji, šaljući tako zadnjem sloju, sloju *pogleda*, upute koje su mu potrebne za prikazivanje odgovarajućih poruka korisniku.

Sloj *pogleda* ili sloj *platforme* je najuži dio MVP obrasca. On služi isključivo za prikaz sučelja i podataka na sučelju i slanje poruka vezanih uz korisničku interakciju sloju *prezentacije*. Nepisano pravilo je da ovaj sloj mora biti "najgluplji", odnosno da zna najmanje o svojstvima i poslovnoj logici u aplikaciji. Na taj način sloj *pogleda* ovisi samo o platformi i njenim pojedinostima, dok se ostali slojevi mogu bez problema ponovno koristiti u drugim projektima ili drugim dijelovima aplikacije.

Graf koji prikazuje strukturu MVP obrasca i tok podataka između slojeva je dan slikom 2.2, koji je izrađen prema [13].



Slika: 2.2 MVP struktura i tok podataka

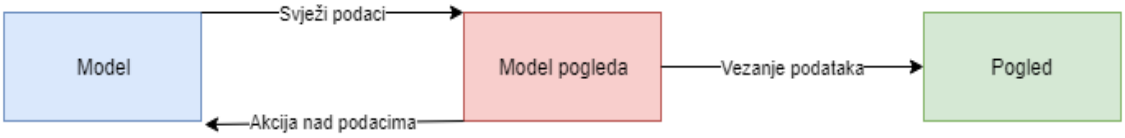
Nakon što se dogodi korisnička interakcija, sloj *prezentacije* prima poruku o istoj. Sloju *modela* se, nakon ispitivanja poruke i podataka, iz sloja *prezentacije* šalje zapovijed koja pokreće traženu akciju (npr. registracija korisnika), pri čijem je završetku prezenter obaviješten, i, s novim podacima, javlja sloju *pogleda* što sljedeće treba biti prikazano. Svaka akcija koju korisnik može poduzeti i svaki korak postavljanja sučelja moraju biti proslijeđeni sloju *prezentacije*, koji odlučuje kako sloj *pogleda* treba nastaviti s radom. Tako se osigurava da se ne može dogoditi ništa što se nije predvidjelo. Detaljnije o MVP obrascu, njegovom nastanku i primjeni u Androidu je opisano u [13].

2.6. Model-Pogled-Model Pogleda arhitekturni obrazac

Paralelno MVP obrascu je bio razvijan i obrazac Model-Pogled-Model Pogleda (engl. *Model-View-ViewModel,* MVVM). Njegova struktura gotovo savršeno prati strukturu MVP-a, ali se oni razlikuju u jednoj vrlo bitnoj stvari. Gledajući sliku 2.3, primjećuje se kako svaki ciklus u MVP obrascu završava tako što sloj *prezentacije* striktno, imperativno, govori sloju *pogleda* što iduće treba prikazati ili se dogoditi.

Problem kod toga je što je, prateći SOLID principe rada, potrebno za svaki komadić podataka koji se prikazuje napisati novu funkciju koja ga treba prikazati. Ako se prikazuje desetak različitih podataka (npr. detaljan opis korisnika), potrebno je i napisati

desetak novih funkcija, što stvara jako puno koda koji se stalno ponavlja (engl. *boilerplate code*) [14]. Na slici 2.3 je vidljivo kako je, u MVVM strukturi, taj problem riješen na jednostavan način.



Slika: 2.3 MVVM struktura i tok podataka

Ne postoji obostrano povezivanje sloja *pogleda* i sloja *modela pogleda,* koji mijenja sloj *prezentacije* po funkciji. U MVVM obrascu, oni su povezani preko vezanja podataka (engl. *Data Binding,* DB), gdje postoji grupirani podatak za svaki zaslon u aplikaciji, koji se zove stanje pogleda (engl. *ViewState,* VS), koji se veže na sloj *pogleda* preko sloja *modela pogleda*.

DB omogućuje da se, na kraju svakog ciklusa, samo promijeni VS i sloj pogleda automatski dobije poruku da se mora ažurirati. Na svaku njegovu promjenu prikazuje se svaki put cijeli komad (engl. *chunk*) podataka odjednom, umjesto da se sloju pogleda zapovijeda pojedinačno za svaki dijelić podatka. Ovakav pristup više nije imperativan, gdje se samo šalju zapovijedi, već je i djelomično, ili čak u potpunosti, reaktivan, jer sloj *pogleda* reagira na promjene i automatski ih bilježi.

Iako MVVM kao takav diktira da je cjelokupni ciklus reaktivan, implementacijski dio je vrlo je teško napraviti tako da stvari poput navigacije budu reaktivne. Navigacija se ne može prikazati kao tip podatka, jer ni sama nije podatak. Zbog toga se MVVM često nadopunjava s dodatnim slojem, slojem *navigacije* kojeg predstavljaju usmjerivači (engl. *router*). Usmjerivači znaju kako upravljati navigacijom, znaju kako se pokreće idući zaslon, čime se dodatno olakšava sloj *pogleda*.

Osim pomoću usmjerivača, navigacija može biti riješena i hibridom MVP i MVVM pristupa, gdje se MVP dio ciklusa brine o porukama i navigaciji, a MVVM dio o toku podataka kroz ciklus i o ažuriranju sloja *pogleda*. Primjena, nastanak i struktura MVVM obrasca su nadalje opisani u [15].

2.7. Testiranje aplikacija

Jedan od ciljeva postavljanja arhitekture u programskoj podršci su testovi. Testovi su oblik pisanja dodatnog koda koji prolazi, u zatvorenom okruženju, kroz prethodno napisane funkcije i provjerava njihovu valjanost. Postoji nekoliko oblika pisanja testova za Android aplikacije, a najpoznatiji su jedinični testovi (engl. *unit tests*) i testovi korisničkog sučelja (engl. *iser interface tests*). Dok jedinični testovi rade testiranje u crnoj kutiji (engl. *black-box testing*), ograničavajući cjelokupan doticaj s aplikacijom na vrlo male jedinice, testovi korisničkog sučelja se izvode na pravim uređajima ili Android emulatorima, testirajući i pokretajući tako cijelu aplikaciju.

Testovi provjeravaju kod na dva načina. Prvi je onaj gdje se testiraju željeno i neželjeno ponašanje aplikacije i određuje da je ishod testa očekivano ponašanje za pojedinačne slučajeve, npr. greška u slučaju nevaljanih podataka ili registracija korisnika za ispravne podatke. Drugi slučaj je onaj kad se već testiran kod mijenja u cijelosti. Ako je kod u srži promijenjen, a testovi i dalje prolaze, to znači da je napisan kod neispravan i kako postoje nepoželjni i predviđeni događaji u aplikaciji.

Nadalje, testovi služe ujedno i kao detaljna dokumentacija aplikacije. U slučaju da aplikacija sadrži vrlo kompleksne funkcije, one se najbolje dokumentiraju kroz testove koji pokrivaju i opisuju svaku putanju izvođenja tih funkcija. Testiranje programske podrške je više objašnjeno u [16], a pristup razvoja upogonjenog testovima je opisan u [17].

2.8. Biblioteke potrebne za razvoj aplikacije

Gotovo uvijek se u razvoju Android aplikacija koriste biblioteke treće strane (engl. *third party libraries*), kojima je postupak razvoja aplikacija uvelike olakšan. Android razvojna platforma je prepuna već spomenutog *boilerplate* koda, zbog čega je trend Android zajednice bio generiranje koda i odvajanje jezičnih tvorevina u biblioteke.

Generiranje koda se vodi filozofijom: “Zašto pisati kod, kad nešto može pisati kod za tebe?” Od živih i datotečnih predložaka (engl. *live and file templates*) do generiranja koda pomoću anotacija, ovaj postupak je zastupljen u brojnim bibliotekama, koje su pak napisane da bi svi koje zanima Android imali više vremena brinuti se o bitnijim stvarima poput čistoće koda i testova. Generiranje koda je doseglo visoke razine, gdje je programerima omogućeno generiranje gotovih projekata sa svega nekoliko klikova mišem.

S druge strane, biblioteke su kamen temeljac Android platforme. Čak i osnovni dijelovi Androida, poput verzije platformskih alata napravljenih za podršku unazad (engl. *backwards compatibility*), su sami po sebi biblioteka. Koristeći Gradle [18] i Maven sustave za izgradnju (engl. *build systems*), moguće je uključiti bilo koju javnu biblioteku, dodavanjem svega jedne linije koda u projekt. To je potaknulo brojne programere da pišu svoje biblioteke, koje će kasnije drugi koristiti. Od biblioteka za elemente korisničkog sučelja, računanje kompleksnih matematičkih funkcija, rad s tokovima podataka, višenitnosti, povezivanje s mrežom i bazom, teško je naći problem za kojeg već nije napisana biblioteka.

U ovom završnom radu će biti korištene biblioteke Butterknife i Kotlin Android Extensions za pametno povezivanje korisničkog sučelja s kodom, Dagger2 za organizaciju ovisnosti u aplikaciji [19], Architecture Components [20] za čuvanje podataka kod promjene orijentacije uređaja, Koin [21] za Kotlin verziju organizacije ovisnosti, Retrofit [22] za povezivanje s mrežom i Room za rad s bazom.

Razlog uvođenja biblioteka u završni rad je taj što su one ključni dio svakodnevnog rada na Android aplikacijama. Ne samo da olakšavaju svaku liniju napisanog koda, već su ujedno i preduvjet za rad na proizvodnim projektima u brojnim tvrtkama i daju uvid u kompleksan životni ciklus Android platforme. Razvoj aplikacija bez ovih rješenja bi bio krajnje mukotrpan i dugotrajan, dok je uz njih vrlo jednostavan.

3. OBLIKOVANJE ANDROID APLIKACIJE

U ovom dijelu završnog rada će biti prikazana stvarna, funkcionalna implementacija prethodno navedenih arhitekturnih obrazaca. Počevši od naivne implementacije, bit će prikazane strukture i povezivanja slojeva i odgovornosti aplikacije na najefikasniji način, vezano uz svaki obrazac. Nadalje će biti dan i osvrt na postavljene arhitekturne obrasce te oblikovne obrasce korištene za preglednije strukturiranje koda. Također će, prije same implementacije, biti opisane najvažnije ovisnosti potrebne za rad aplikacije te kako će se one postaviti unutar projektne strukture.

U aplikaciji koji će biti pisana će biti prikazani popularni, nadolazeći i najbolje ocijenjeni filmovi s usluge *TheMovieDB*, koristeći aplikacijsko programsko sučelje (engl. *Application Programming Interface, API*) koje navedena usluga pruža. Dobiveni rezultati će se spremati u bazu zbog potrebe rada bez mreže. Graf ovisnosi koji pokazuje na koji način će usluge i određeni slojevi biti povezani se nalazi na slici 3.1.

Slika 3.1: Prikaz grafa ovisnosti u aplikaciji

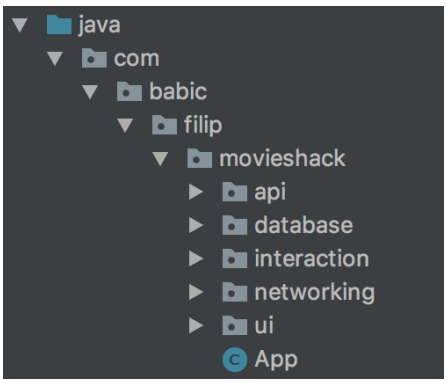
3.1. Postavljanje temeljnog dijela projekta

Prije nego se projekt krene razvijati, potrebno je postaviti osnovne dijelove aplikacije koji će se kasnije ponovno koristiti, smanjujući tako pisanje *boilerplate* koda. Najčešće rješenje je kreiranje objekta s jedinstvenom instancom (eng. *Singleton*) koji predstavlja aplikacijski proces. Unutar aplikacijskog singletona će biti spremljene sve usluge i vanjski entiteti koji su potrebni za rad, štedeći tako procesorsku moć potrebnu za stvaranje entiteta tog tipa.

Svaki projekt dakle počinje s nekim oblikom temelja. Imajući na umu kako se ovaj projekt izvodi u tri različite inačice, najbolje je postaviti što više ovisnosti na način da se mogu koristiti u svakoj inačici, kako promjena arhitekturnog obrasca ili razvojne paradigme ne bi utjecala na njih, već samo na sloj poslovne logike i korisničkog sučelja.

U razvoju aplikacija se dakle najčešće kreće od vanjskog sloja, odnosno s povezivanjem aplikacije s mrežnim sučeljem i bazom podataka. Stavljajući te ovisnosti u apstraktne definicije se osigurava čist i skriven način rada potrebnih usluga, sa strane poslovne logike i samog korisničkog sučelja.

Idejno bi se, prateći ove principe, i te usluge trebale moći zamijeniti, jednako kao i arhitekturni obrazac ili razvojna paradigma, bez da sloj poslovne logike i korisničkog sučelja moraju znati za tu promjenu. Tako će na početku razvoja aplikacije biti kreirani paketi vidljivi na slici 3.2.

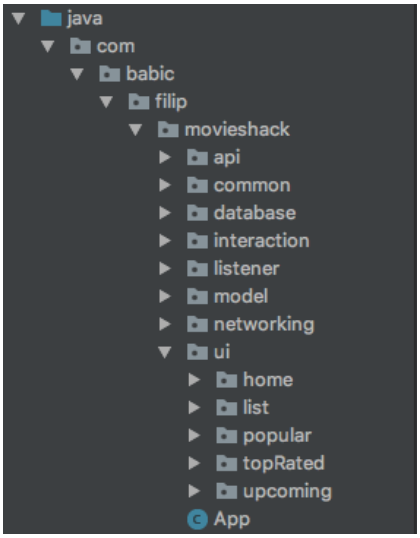


Slika: 3.2 Osnovna struktura projekta

Ključni paketi i slojevi koji će se koristiti kroz sve varijante aplikacije su povezivanje s mrežom - *api*, *networking* i *interaction* paketi sadrže sve što je potrebno za mrežni razgovor. Uz te pakete postoji i "database" paket, koji sadržava sve potrebno za rad s bazom podataka. Nadalje, *App singleton* je klasa u kojoj će proces kreiranja i spremanja tih usluga biti uspostavljen.

App, odnosno klasa aplikacije (engl. *application class*) je klasa koja se pokreće s klikom na ikonicu aplikacije i završava sa završavanjem procesa koji ju pokreće. Idealno mjesto za držanje svih komponenti potrebnih za rad aplikacije. Kako bi se osiguralo da se koristi posebno kreirana klasa App, a ne neka uobičajena Androidova *Application* klasa, potrebno je navesti pod oznakom *name* u Android manifest dokumentu putanju koja pokazuje App klasu.

Uz proširivanje projekta i dovršavanje cjelokupne implementacije, završna struktura će se znatno promijeniti, odnosno velik broj pomoćnih paketa i klasa za obradu podataka će se dodati, a osim pomoćnih klasa će se proširiti i *ui* paket, dobivajući tako dodatni paket za svaki zaslon u aplikaciji. Završna struktura će izgledati kao na slici 3.3.



Slika: 3.3 Konačna struktura proje ta

Vidljivo je kako su dodani pomoćni paketi poput *common* koji sadrže pomoćne (engl. *utility/utils*) klase za rad s podacima i dodatne konstante potrebne za rad aplikacije. Također postoji paket *model* koji drži sve jednostavne Java objekte (engl. *Plain Old Java Object - POJO*) koji modeliraju podatke koji dolaze s mreže i spremaju se u bazu.

3.2. Odabrane ovisnosti

Već je spomenuto koje sve biblioteke će se koristiti u razvoju aplikacije. Ako se otvori *app.gradle* dokument, mogu se vidjeti sve njihove ovisnosti, organizirane po funkcionalnosti na slici 3.4. Iako *app.gradle* dokument sadrži mnogo ovisnosti, neće sve biti korištene u svakoj od inačica aplikacije. Biblioteke za rad s Androidom koje su navedene pod *Support librarie*s su većinom vezane za korisničko sučelje i Android funkcionalnosti. Samo ime govori da *Networking* služi za mrežno povezivanje, a ostatak

biblioteka služi za testiranje, poput JUnit [23] i Mockito [24] biblioteka, provjeravanje postoji li curenja u memoriji (engl. *memory leak*) – *LeakCanary [25]* i za postavljanje grafa ovisnosti - *Dagger* - u idućim iteracijama.

Sve ovisnosti potrebne za MVVM pristup i vezane uz bazu podataka *Room* se nalaze ispod *Architecture components* i *room database* komentara. Sve te ovisnosti su verzionirane pomoću eksternih Gradle varijabli, kako bi mogli jednom izmjenom ažurirati sve ovisnosti vezane uz određenu varijablu. Na taj način se olakšava održavanje ali i ujedno smanjuje vrijeme potrebno za ažuriranje te se smanjuje mogućnost greške pri verzioniranju.

```
implementation fileTree(dir: 'libs', include: ['*.jar'])

// UI
implementation 'com.android.support.constraint:constraint-layout:1.1.0'
implementation 'com.github.ittianyu:BottomNavigationViewEx:1.2.4'
implementation 'com.romandanylyk:pageindicatorview:1.0.0'

// Image loading
implementation "com.github.bumptech.glide:glide:$glide"
kapt "com.github.bumptech.glide:compiler:$glide"

// Support libraries
implementation "com.android.support:appcompat-v7:$supportLib"
implementation "com.android.support:design:$supportLib"
implementation "com.android.support:recyclerview-v7:$supportLib"
implementation "com.android.support:cardview-v7:$supportLib"
implementation "com.android.support:animated-vector-drawable:$supportLib"
implementation "com.android.support:customtabs:$supportLib"

// Networking
implementation "com.squareup.retrofit2:retrofit:$retrofit"
implementation "com.squareup.retrofit2:converter-gson:$retrofit"
implementation "com.squareup.okhttp3:logging-interceptor:$okhttp"

// Dagger
implementation "com.google.dagger:dagger:$dagger"
kapt "com.google.dagger:dagger-compiler:$dagger"
implementation "com.google.dagger:dagger-android:$dagger"
implementation "com.google.dagger:dagger-android-support:$dagger"
kapt "com.google.dagger:dagger-android-processor:$dagger"

// Leakcanary
debugImplementation "com.squareup.leakcanary:leakcanary-android:$leakcanary"
releaseImplementation "com.squareup.leakcanary:leakcanary-android-no-op:$leakcanary"
testImplementation "com.squareup.leakcanary:leakcanary-android-no-op:$leakcanary"

// Test
testImplementation 'junit:junit:4.12'
testImplementation "com.nhaarman:mockito-kotlin:1.5.0"
testImplementation "android.arch.core:core-testing:$architectureComponents"

//Architecture components
implementation "android.arch.lifecycle:extensions:$architectureComponents"
implementation "android.arch.lifecycle:reactivestreams:$architectureComponents"

//room database
implementation "android.arch.persistence.room:runtime:$room"
annotationProcessor "android.arch.persistence.room:compiler:$room"
implementation "android.arch.persistence.room:rxjava2:$room"
```

Slika 3.4: Gradle skripta s ovisnostima u aplikaciji

3.3. Kreiranje potrebnih usluga

Optimalan način pokretanja i postavljanja usluga (engl. *initialization*) je taj da se kreiraju pomoću takozvanih klasa tvornica (engl. *factory*). Zovu se klase tvornice jer svaka ima točno određenu ovisnost koju kreira, a dohvaćanje tih ovisnosti je omogućena cijeloj aplikaciji kroz metode označene *static* ključnom riječju.Na taj način se mogu

pozvati metode bez nužnog kreiranja objekata klasa tvornica, već jednostavnim pozivom na razini klase.

Kako glavninu aplikacije zapravo čini rad s mrežnim sučeljem, prvo će biti kreirana tvornica za vanjski sloj aplikacije koji radi s mrežom radi dohvaćanja podataka s *TheMovieDB* API sučelja. Taj postupak je vidljiv na slici 3.5.

```
public class NetworkingUtils {

    private static final String BASE_URL = "";
    private static MovieApiService movieApiService;

    private static String baseUrl() { return BASE_URL; }

    private static HttpLoggingInterceptor loggingInterceptor() {
        return new HttpLoggingInterceptor().setLevel(HttpLoggingInterceptor.Level.BODY);
    }

    private static OkHttpClient okHttpClient(HttpLoggingInterceptor loggingInterceptor) {
        return new OkHttpClient.Builder()
                .addInterceptor(loggingInterceptor)
                .build();
    }

    private static GsonConverterFactory gsonConverterFactory() {
        return GsonConverterFactory.create();
    }

    private static Retrofit retrofit(GsonConverterFactory gsonConverterFactory, OkHttpClient okHttpClient, String baseUrl) {
        return new Retrofit.Builder()
                .client(okHttpClient)
                .baseUrl(baseUrl)
                .addConverterFactory(gsonConverterFactory)
                .build();
    }

    private static MovieApiService movieApiService() {
        if (movieApiService == null) {
            movieApiService = retrofit(gsonConverterFactory(), okHttpClient(loggingInterceptor()), baseUrl()).create(MovieApiService.class);
        }
        return movieApiService;
    }

    public static MovieInteractor movieInteractor() {
        return new MovieInteractorImpl(movieApiService());
    }
}
```

Slika: 3.5 Tvornica za ovisnosti vezane uz mrežno sučelje

Za rad su potrebni presretači prometa (engl. *interceptor*) kako bi u ispisu bilo vidljivo koji se sve podaci šalju i primaju. Također su potrebni i pretvarači (engl. *converter*) javaskript objektne notacije (engl. *JavaScript Object Notation, JSON*) [26] podataka kako bi se automatiziralo prevođenje (engl. *parsing*) JSON podataka u stvarne Java/Kotlin objekte. Pri samom kraju se kreira *Retrofit* objekt koji će svojom unutarnjom implementacijom izgenerirati sve pozive u *MovieApiService* sučelju.

Isti postupak se primjenjuje za kreiranje tvornica za bazu podataka *Room*. Obzirom da je *Room* odabrana biblioteka za rad s bazom podataka, potrebno je prvo kreirati objekt pristupanja domeni (engl. *Data Access Object, DAO*) [27] koji će biti posrednik između SQL baze koja se generira u pozadini i poslovne logike. To je moguće na način prikazan na slici 3.6.

```
@Dao
public interface MovieDao {

    @Insert
    void addMovie(Movie movie);

    @Update
    void updateMovie(Movie movie);

    @Query("SELECT * FROM movies WHERE type = :type")
    List<Movie> getMoviesByType(String type);

    @Query("SELECT * FROM movies WHERE id = :id")
    Movie getMovieById(String id);
}
```

Slika: 3.6 DAO za rad s bazom podataka

Korištenjem marker anotacija *Dao, Insert, Update* i *Query* dano je do znanja Room bazi što sve je nužno izgenerirati. Na raspolaganju će biti funkcija koja sprema novi element u bazu ili ga ažurira i funkcije koje traže jedan element po identifikatoru ili sve elemente čiji tip odgovara zadanom nizu znakova.

Nakon što je modeliran DAO, potrebno mu je postaviti vrijednost, te zauzeti memoriju za pomoćnu klasu za bazu podataka *DatabaseInterface*. Taj postupak je u cjelosti prikazan na slici 3.7.

```
@Database(entities = {MovieDao.class}, version = 1)
public abstract class DaoProvider extends RoomDatabase {

    private static final String NAME = "movies_database";

    public abstract MovieDao getMovieDao();

    private static DaoProvider instance;

    public static DaoProvider getInstance(Context context) {
        if (instance == null) {
            synchronized (DaoProvider.class) {
                if (instance == null) {
                    instance = Room.databaseBuilder(context, DaoProvider.class, NAME)
                                   .allowMainThreadQueries()
                                   .build();
                }
            }
        }

        return instance;
    }
}
```

Slika: 3.7 Tvornica za sučelje prema bazi podataka

Postupak je jednostavan i sličan onome za *Retrofit* klijent. Radi se provjera postoji li već instanca DAO tvornice, a ako ne postoji, ona se kreira. Nakon što se kreira, moguće je koristiti apstraktnu funkciju *getMovieDao()* kako bi se upogonio posrednik za bazu.

Nakon postavljanja ove dvije ključne stvari, one se spremaju u *App* klasu, te drže na globalnoj razini aplikacije kako bi im se uvijek na jeftin način, što se performansa tiče, moglo pristupiti. Takav oblik povezivanja aplikacije s glavnim ovisnostima je prikazan na slici 3.8.

```
public class App extends Application {

    private static MovieInteractor movieInteractor;
    private static DatabaseInterface databaseInterface;

    @Override
    public void onCreate() {
        super.onCreate();

        movieInteractor = NetworkingUtils.movieInteractor();
        databaseInterface = new DatabaseImpl(DaoProvider.getInstance(this).getMovieDao());
    }

    public static MovieInteractor getMovieInteractor() {
        return movieInteractor;
    }

    public static DatabaseInterface getDatabaseInterface() {
        return databaseInterface;
    }
}
```

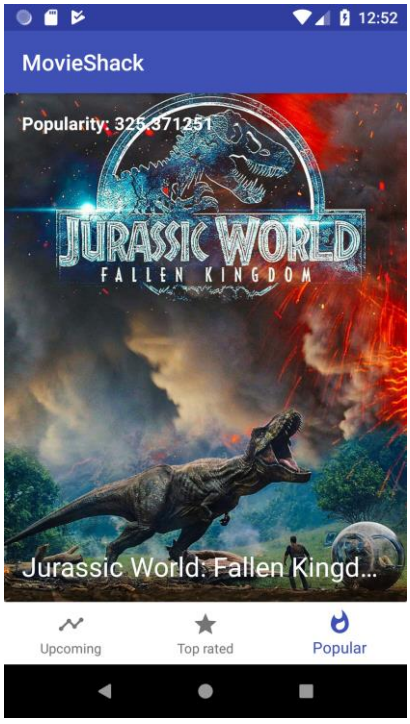
Slika: 3.8 Postavljanje aplikacijskog singletona i glavnih ovisnosti

Obzirom da su te dvije pomoćne klase označene static ključnom riječi, mogu se dohvaćati na klasnoj razini, a one će uvijek biti postavljene i spremne za korištenje. Na taj način se štedi memorija i potrebno vrijeme za pristupanje tim osnovnim dijelovima aplikacije, dok se usput koriste oblikovni obrasci poput tvornice, graditelja (engl. *builder*) i *Singleton* objekta.

Ovi oblikovni obrasci, kao i mnogi drugi koje će biti korišteni, se najbolje objašnjavaju u knjizi *Design Patterns* [28]. Njeni autori su prvi uzeli korisne obrasce u objektno-orijentiranom razvojnom svijetu i prikazali ih u knjizi, dajući tako pismeni izvor smjernica za pametno i čisto strukturiranje koda. Iako je prošlo puno godina nakon izdavanja knjige, ona se još uvijek smatra jednom od najkvalitetnijih i utjecajnijih knjiga u svijetu objektno orijentiranog programiranja. Ipak, zanimljivo je za znati kako većina opisanih uzoraka, u knjizi, je u današnje vrijeme, pojavom modernih jezika, zapravo nepotrebno, jer napredne jezikovne mogućnosti olakšavaju strukturiranje i izgradnju programske podrške, bez potrebne za oblikovnim obrascima.

3.4. Naivna implementacija

Prvotna inačica projekta bit će pisana na naivan način, da bi se bolje pokazalo kako, u početničkoj fazi, strukturirati kod pritom ne gubeći jednostavnost. Ključne stvari koje će biti jasno vidljive u ovoj inačici su manjak organiziranosti koda i odgovornosti i teško vezanje detalja Android platforme s poslovnom logikom. Nakon izgradnje osnovnog zaslona aplikacije i sva tri fragmenta koji će prikazivati različite tipove filmova, dobije se izgled pri pokretanju, kao na slici 3.9.



Slika 3.9: Prikaz zaslona aplikacija s podacima dohvaćenim s mrežnog sučelja

Kako bi se najbolje objasnio naivni pristup u kodu, obradit će se postupak zatraživanja podataka s mrežnog sučelja, spremanje i prikazivanje istih u bazu i na korisničko sučelje. Postupak za prikaz ovog zaslona, točnije fragmenta, jest sljedeći: prvo su kreirani elementi pogleda, nakon toga se stvara poveznica između prikaza podataka i prilagodnika podataka (engl. *adapter*). Kad su sve veze uspostavljene, podaci se zatraže pomoći API sučelja i ovisno o ishodu zahtjeva, ili se prikažu dobiveni podaci ili se prikaže greška korisniku, ovisno o tipu iznimke koju korisnik dobije.

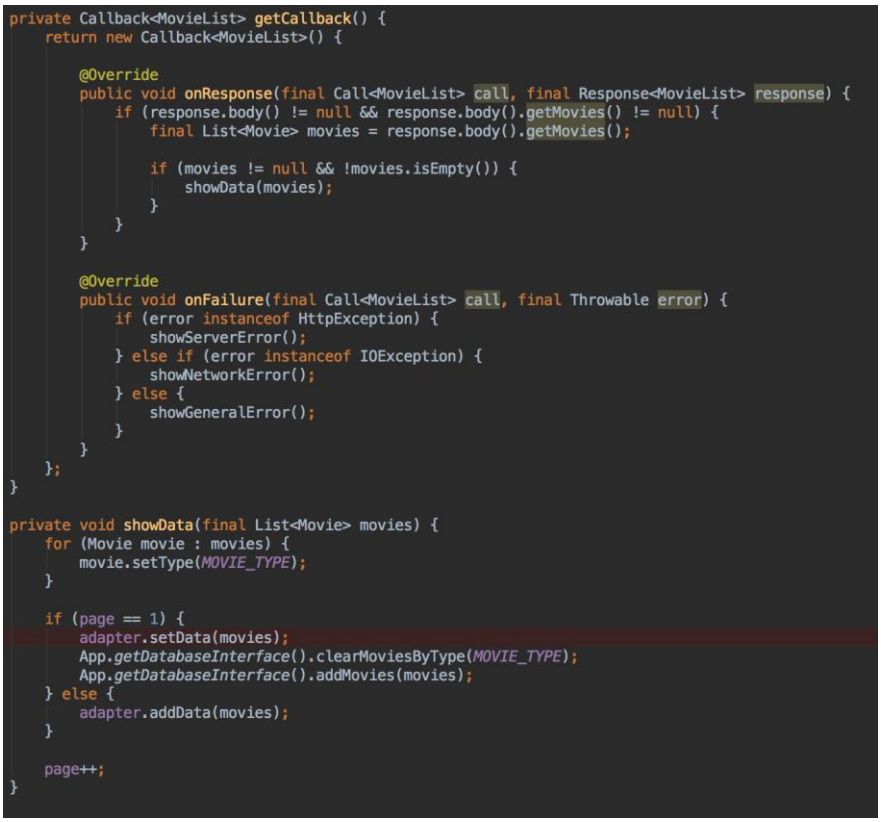
Postavljanje elemenata pogleda i osnovnih poveznica poslovne logike se postiže na način prikazan na slici 3.10.

```
public class PopularMoviesFragment extends Fragment implements RefreshablePage {

    private final MovieAdapter adapter = new MovieAdapter(R.layout.item_movie_popular);

    private int page = 1;

    private static final String MOVIE_TYPE = "POPULAR";

    @Nullable
    @Override
    public View onCreateView(@NonNull final LayoutInflater inflater,
                             @Nullable final ViewGroup container, @Nullable final Bundle savedInstanceState) {
        return inflater.inflate(R.layout.fragment_movies, container,  attachToRoot: false);
    }

    @Override
    public void onViewCreated(@NonNull final View view, @Nullable final Bundle savedInstanceState) {
        super.onViewCreated(view, savedInstanceState);
        initUi(view);

        getMovies();
    }

    private void initUi(View view) {
        RecyclerView movies = view.findViewById(R.id.movies);
        movies.setItemAnimator(new DefaultItemAnimator());
        movies.setLayoutManager(new LinearLayoutManager(getActivity()));
        movies.setAdapter(adapter);
        movies.addOnScrollListener(new LastItemReachedListener(this::getMovies));
    }

    @Override
    public void refresh() {
        page = 1;
        getMovies();
    }

    private void getMovies() {
        if (page == 1 && !NetworkingUtils.hasInternet(getActivity())) {
            final List<Movie> movies = App.getDatabaseInterface().getMoviesByType(MOVIE_TYPE);
            adapter.setData(movies);
        } else {
            App.getMovieInteractor().getMovies(page, MOVIE_TYPE, getCallback());
        }
    }
```

Slika 3.10: Dio klase PopularMoviesFragment vezan za povezivanje elemenata sučelja

Osim što se kroz funkcije životnog ciklusa fragmenta *onCreateView* i *onViewCreated* stvara sučelje sa svim elementima pogleda, to sučelje je potrebno povezati kroz kod radi određivanja funkcionalnosti. Nakon što se pronađe element pogleda *RecyclerView*, potrebno je proslijediti mu adapter, upravljač rasporeda pogleda (engl. *layout manager*) i zbog lijenog učitavanja podataka (engl. *lazy loading*) osluškivač koji sluša geste vertikalnog kretanja liste podataka.

Odmah nakon povezivanja pretvarača i sučelja, podaci se zatraže s prvom početnom stranicom te se poslovna logika vezana uz obradu, spremanje i prikazivanje podataka pokreće. Ovaj postupak je vidljiv na slici 3.11.



Slika 3.11: Dio koda vezan za dohvaćanje podataka s mrežnog sučelja i spremanje u bazu

Nakon što se zatraže podaci, potrebno je proslijediti strukturu poziva natrag (engl. *callback*), koja omogućuje da se nakon dohvaćanja podataka nazad pozove funkcija unutar neke druge klase, kako se dohvaćanje podataka može odvijati u pozadinskoj niti, ne blokirajući tako rad korisničkog sučelja. Kad su podaci spremni, radi se dodatna provjera vezana za stranicu koja se prikazuje, a ako je korisnik na početnoj stranici, ona se sprema u bazu, kako bi se u slučaju nedostatka mrežne veze prva stranica podataka mogla prikazati.

Naivna implementacija je vrlo jednostavna u srži, ne zahtijeva previše strukture i prethodnog razmišljanja za postavljanje. Ujedno je i u tome glavni problem takve implementacije. Ne planirajući nekoliko koraka unaprijed, usko je vezana aktivnost s poslovnom logikom, aplikacijskim *Singletonom* i model slojem. To stvara visoku i tešku povezanost, kod koje je svaka promjena težak vremenski teret.

Također, testiranje gore prikazanog koda je nemoguće, jer implementacija zahtijeva da se testiranje radi putem testova korisničkog sučelja, što zahtijeva rad s pravom bazom podataka i pravim mrežnim sučeljem, tako stvarajući probleme i ne dopuštajući stvaranje lažnih (engl. *mock*) podataka i sučelja, koja se mogu koristiti u kontroliranim uvjetima, umjesto s pravim sučeljima.

Osim toga, povezivanjem svih slojeva aplikacije na jednom mjestu se stvara velika količina programskog koda na svakom elementu Android sučelja, bila to Android aktivnost ili fragment. Kako bi ovaj postupak zadržao svoju jednostavnost, pritom smanjujući kod na elementima sučelja i stvarajući okruženje koje podržava testiranje koda, razvio se MVP princip, koji će biti obrađen sljedeći.

3.5. MVP implementacija

Velik korak dalje ispred naivne implementacije stvara MVP princip strukturiranja koda. On upotpunjuje naivnu implementaciju sa slojevima, dajući tako cjelokupnoj strukturi veći osjećaj smislenosti, organiziranosti i čistoće. Kako bi se kod uopće mogao strukturirati po MVP principu, potrebno je prvo kreirati ugovor (engl. *contract*) između slojeva pogleda i poslovne logike ili prezentacije. Taj ugovor je najčešće prikazan pomoću Kotlin ili Java sučelja (engl. *interface*) i vidljiv je na slici 3.12.

```
public interface PopularMoviesContract {

    interface View {

        void showMovies(List<Movie> movies);

        void addMoreMovies(List<Movie> movies);

        void showServerError();

        void showNetworkError();

        void showGeneralError();
    }

    interface Presenter {

        void setView(View view);

        void getMovies(final boolean hasInternet);

        void refresh(final boolean hasInternet);
    }
}
```

Slika 3.12: Ugovor između pogleda i poslovne logike

Kreiranjem ugovora između pogled-prezenter para na jednom mjestu se postiže puno bolja organiziranost, jer se puno lakše može shvatiti cjelokupna mogućnost i funkcionalnost zaslona koji se prikazuje.

Naravno, potrebno je još i dati stvarnu implementaciju za svaku od funkcija, ali obzirom da je većina koda već gotova, jedini korak je fizički odvojiti kod poslovne logike iz Android specifičnih klasa, u ovom slučaju fragmenta.

Kod koji obuhvaća na jednom mjestu sve ovisnosti potrebne za rad s bazom podataka i mrežnim sučeljem je vidljiv na slici 3.13. Dok je kod za rad s podacima vidljiv na slici 3.14.

```
public class PopularMoviesPresenter implements PopularMoviesContract.Presenter {

    private static final String MOVIE_TYPE = "POPULAR";

    private final MovieInteractor movieInteractor;
    private final DatabaseInterface databaseInterface;
    private PopularMoviesContract.View view;

    private int page = 1;

    @Inject
    public PopularMoviesPresenter(final MovieInteractor movieInteractor, final DatabaseInterface databaseInterface) {
        this.movieInteractor = movieInteractor;
        this.databaseInterface = databaseInterface;
    }

    @Override
    public void setView(final PopularMoviesContract.View view) { this.view = view; }

    @Override
    public void getMovies(final boolean hasInternet) {
        if (page == 1 && !hasInternet) {
            view.showMovies(databaseInterface.getMoviesByType(MOVIE_TYPE));
        } else {
            movieInteractor.getMovies(page, MOVIE_TYPE, getCallback());
        }
    }

    @Override
    public void refresh(final boolean hasInternet) {
        page = 1;
        getMovies(hasInternet);
    }
```

Slika 3.13: Kod poslovne logike, odvojen u klasi prezenter

```
    private Callback<MovieList> getCallback() {
        return new Callback<MovieList>() {
            @Override
            public void onResponse(final Call<MovieList> call, final Response<MovieList> response) {
                if (response.body() != null && response.body().getMovies() != null) {
                    final List<Movie> movies = response.body().getMovies();

                    if (movies != null && !movies.isEmpty()) {
                        showData(movies);
                    }
                }
            }

            @Override
            public void onFailure(final Call<MovieList> call, final Throwable error) {
                if (error instanceof HttpException) {
                    showServerError();
                } else if (error instanceof IOException) {
                    showNetworkError();
                } else {
                    showGeneralError();
                }
            }
        };
    }

    private void showServerError() { view.showServerError(); }
    private void showNetworkError() { view.showNetworkError(); }
    private void showGeneralError() { view.showGeneralError(); }

    private void showData(final List<Movie> movies) {
        for (Movie movie : movies) {
            movie.setType(MOVIE_TYPE);
        }

        if (page == 1) {
            view.showMovies(movies);
            databaseInterface.clearMoviesByType(MOVIE_TYPE);
            databaseInterface.addMovies(movies);
        } else {
            view.addMoreMovies(movies);
        }

        page++;
    }
```

Slika 3.14: Kod vezan za rad s podacima, u klasi prezenter

Kod koji prikazuje greške korisniku u slučaju da nešto pođe po zlu se nalazi u sloju pogleda, kao što je vidljivo na slici 3.15, a popraćen je kodom za postavljanje ovisnosti i povezivanja s prezenterom, kao što je vidljivo na slici 3.16.

```
    @Override
    public void showServerError() {
        Toast.makeText(getActivity(), getString(R.string.server_error), Toast.LENGTH_SHORT).show();
    }

    @Override
    public void showNetworkError() {
        Toast.makeText(getActivity(), getString(R.string.network_error), Toast.LENGTH_SHORT).show();
    }

    @Override
    public void showGeneralError() {
        Toast.makeText(getActivity(), getString(R.string.general_error), Toast.LENGTH_SHORT).show();
    }
}
```

Slika 3.15: Kod za prikazivanje grešaka

Ovdje se ističe u isto vrijeme i prednost i problem MVP pristupa, kod je vrlo čist i kratak, a funkcije su većinom dugačke jednu do nekoliko linija. Time se ostvaruje visoka preglednost i čitljivost koda, ali nažalost se i količina koda zbog brojnih funkcija, anotacija i zagrada dosta povećala.

```
public class PopularMoviesFragment extends Fragment implements RefreshablePage, PopularMoviesContract.View {

    private final MovieAdapter adapter = new MovieAdapter(R.layout.item_movie_popular);

    @Inject
    PopularMoviesContract.Presenter presenter;

    @Nullable
    @Override
    public View onCreateView(@NonNull final LayoutInflater inflater,
                             @Nullable final ViewGroup container, @Nullable final Bundle savedInstanceState) {
        return inflater.inflate(R.layout.fragment_movies, container,  attachToRoot: false);
    }

    @Override
    public void onViewCreated(@NonNull final View view, @Nullable final Bundle savedInstanceState) {
        super.onViewCreated(view, savedInstanceState);
        initUi(view);
        App.component().inject( popularMoviesFragment: this);
        presenter.setView(this);

        presenter.getMovies(NetworkingUtils.hasInternet(getActivity()));
    }

    private void initUi(View view) {
        RecyclerView movies = view.findViewById(R.id.movies);
        movies.setItemAnimator(new DefaultItemAnimator());
        movies.setLayoutManager(new LinearLayoutManager(getActivity()));
        movies.setAdapter(adapter);
        movies.addOnScrollListener(new LastItemReachedListener(() -> presenter.getMovies(NetworkingUtils.hasInternet(getActivity()))));
    }

    @Override
    public void refresh() {
        presenter.refresh(NetworkingUtils.hasInternet(getActivity()));
    }

    @Override
    public void showMovies(final List<Movie> movies) {
        adapter.setData(movies);
    }
```

Slika 3.16: Kod za povezivanje slojeva pogleda i poslovne logike

Također se primjećuje i poseban poziv na App *singleton* klasu koja zove funkciju *inject.* Zajedno s *Inject* anotacijom na prezenteru to pokreće postupak ubrizgavanja ovisnosti o prezenteru pomoću Dagger biblioteke. Kako bi ubrizgavanje ovisnosti bilo omogućeno, potrebno je osigurati graf ovisnosti unutar same Dagger biblioteke. To je

napravljeno na način da je kreirana komponenta (engl. *component*) koja će prosljeđivati ovisnosti unutar aplikacije, prikazana na slici 3.17.

```
@Component(modules = {PresentationModule.class})
public interface AppComponent {

    void inject(App app);

    void inject(PopularMoviesFragment popularMoviesFragment);
}
```

Slika: 3.17: Komponenta potrebna za vezanje ovisnosti u grafu Dagger biblioteke

Unutar gore navedene komponente se nalaze moduli (engl. *module*) koji sadrže metode tvornice za svaku od ovisnosti i njihovih unutarnjih ovisnosti, prikazani na slici 3.18.

```
@Module
@Singleton
public class NetworkingModule {

    private static final String BASE_URL = "http://api.themoviedb.org/";

    @Provides
    public String baseUrl() { return BASE_URL; }

    @Provides
    public HttpLoggingInterceptor loggingInterceptor() { return new HttpLoggingInterceptor().setLevel(HttpLoggingInterceptor.Level.BODY); }

    @Provides
    public OkHttpClient okHttpClient(HttpLoggingInterceptor loggingInterceptor) {
        return new OkHttpClient.Builder()
                .addInterceptor(loggingInterceptor)
                .build();
    }

    @Provides
    public GsonConverterFactory gsonConverterFactory() { return GsonConverterFactory.create(); }

    @Provides
    public Retrofit retrofit(final GsonConverterFactory gsonConverterFactory,
                             final OkHttpClient okHttpClient,
                             final String baseUrl) {
        return new Retrofit.Builder()
                .client(okHttpClient)
                .baseUrl(baseUrl)
                .addConverterFactory(gsonConverterFactory)
                .build();
    }

    @Provides
    public MovieApiService movieApiService(final Retrofit retrofit) { return retrofit.create(MovieApiService.class); }
}
```

Slika 3.18: Modul zaslužan za proizvodnju ovisnosti vezanih uz mrežno sučelje

AppComponent sučelje je komponenta koja prosljeđuje ovisnosti dalje i u njemu se nalazi modul *PresentationModule*. Obzirom da se unutar samog prezentacijskog modula nalaze ostali moduli, oni su vezani i imaju jednak učinak kao da smo ih uključili u komponentu. Funkcije označene s *Provides* anotacijom su tvorničke funkcije, a ako su modul i komponenta označeni istim obujmom (engl. *scope*), poput *Singleton* obujma gore vidljivog, sve funkcije unutar modula će uvijek vraćati isti rezultat, olakšavajući tako sve naredne upite za ovisnostima. Dagger je vrlo važna biblioteka, jer je sad omogućeno unutar cijele aplikacije ponovno korištenje svih navedenih ovisnosti, bez dodatnog memorijskog traga ili potrebne procesorske moći za obradu podataka i pozive funkcija. Jedina negativna strana Dagger biblioteke je što zahtijeva dosta znanja i dokumentacije za postaviti i potrebno je napisati dosta koda kako bi funkcionirao.

MVP princip je uljepšao i organizirao kod potreban za rad fragmenta. Testiranje je omogućeno oslanjanjem na apstrakcije i sučelja umjesto na konkretne klase ili *App* klasu te odvajanjem koda poslovne logike od samog Android dijela ili sloja pogleda. Također, sad je poznato na jednom mjestu koje su sve funkcije vezane za prezenter sloj ili sloj pogleda, otvarajući klasu PopularMoviesContract. U isto vrijeme se kod za postavljanje zaslona gotovo udvostručio a broj klasa potrebnih za rad se povećao s jedne klase na tri.

Iako se te mjere čine kao nešto što se na duže vrijeme ne isplati, ovakva struktura aplikacije daje dodatnu sigurnost i čitljivost i smanjuje vrijeme potrebno za upoznavanje s kodom, znajući točno gdje se koji dio odgovornosti odvija. Nadalje, funkcije su puno kraće i jednostavnije pa je lakše dodavati novu funkcionalnost bez mijenjanja stare i lakše je odrediti izvor problema u slučaju da kod nije ispravan.

3.6. MVVM implementacija

Iako je MVP pristup vrlo čist i kvalitetan, postoje problemi koji su vrlo istaknuti u samom strukturiranju koda i toku podataka. Iako se aplikacija jasno odvaja na slojeve, stvara se velika količina boilerplate koda jer je za svaki podatak koji se prikazuje potrebno kreirati funkciju u ugovoru koja ga prikazuje te dati konkretna implementacija te funkcije na sloju pogleda. Nadalje, svaka metoda koja prikazuje podatke bi se u fazi testiranja trebala pojedinačno testirati, što je dodatan posao.

MVVM princip uzima boilerplate MVP pristupa i dodatan posao koji on nalaže i rješava ga na način da se ne šalje podatak po podatak, već cijelo stanje trenutnog zaslona - objekt tipa *ViewState*. Iako MVVM kao takav ne podrazumijeva nužno samo slušanje podataka na sloju pogleda iz sloja prezentera, već govori o cijeloj reaktivnoj paradigmi, gdje se sve prikazuje kao tok podataka, u ovom poglavlju će biti prikazan najosnovniji oblik MVVM obrasca, te će biti dano dodatno objašnjenje na koje načine se osnovni pristup može unaprijediti.

Osim toga, jezik u kojem će biti implementiran MVVM će biti Kotlin, a sustav ubrizgavanja ovisnosti će biti Koin, čime će se dodatno uštediti na vremenu i kodu potrebnom za povezivanje sučelja i poslovne logike i kreiranju ovisnosti.

Prvi korak je dakle prepisati kod potreban za izgradnju fragmenta i App klase te sustava ubrizgavanja ovisnosti. Postavljanje aplikacijskog *singletona* i svih ovisnosti potrebnih za rad u Kotlinu je napravljeno kao na slici 3.19.

```
class App : Application() {

    companion object {
        lateinit var instance: App
            private set
    }

    override fun onCreate() {
        super.onCreate()
        instance = this

        startKoin(listOf(applicationModule))
    }
}
```

Slika 3.19: Application klasa koja pokreće graf ovisnosti

App klasa se svodi na svega desetak linija koda, na pokretanje Koin biblioteke za ubrizgavanje ovisnosti. Moduli koji su potrebni Koinu za rad su svedeni na kod vidljiv na slici 3.20. Koristeći samo jedan modul, u ovom slučaju globalnu varijablu koja vraća Koin kontekst, moguće je obuhvatiti sve ovisnosti i tvorničke funkcije na jedno mjesto, pritom pišući čist kod koristeći Kotlin lambda funkcije i ekstenzije.

```
private const val BASE_URL = "http://api.themoviedb.org/"

val applicationModule = applicationContext { this: Context

    //application related
    bean { App.instance }
    bean { DaoProvider.getInstance(get()) }
    bean { get<DaoProvider>().movieDao }
    bean { DatabaseImpl(get()) as DatabaseInterface }

    //networking
    bean { BASE_URL }
    bean { HttpLoggingInterceptor().apply { level = HttpLoggingInterceptor.Level.BODY } }
    bean { OkHttpClient.Builder().addInterceptor(get()).build() }
    bean { GsonConverterFactory.create() as Converter.Factory }
    bean { it: ParameterProvider
        Retrofit.Builder()
                .baseUrl(get<String>())
                .client(get())
                .addConverterFactory(get())
                .build()

    }
    bean { get<Retrofit>().create(MovieApiService::class.java) }

    //interaction
    bean { MovieInteractorImpl(get()) as MovieInteractor }

    //presentation
    viewModel { PopularMoviesViewModel(get(), get()) }
}
```

Slika 3.20: Aplikacijski modul koji definira sve tvorničke funkcije za ovisnosti

Umjesto prezentera kao klase koja obuhvaća poslovnu logiku, koristi se klasa *ViewModel* iz biblioteke *Architecture Components* koju je kreirao Google. Ona svojim unutarnjim strukturama i implementacijom osigurava sačuvanje podataka u slučajevima gdje se podaci inače gube, poput promjene orijentacije.

Princip spajanja sloja pogleda i sloja modela pogleda se razlikuje od MVP pristupa jer više ne postoji sučelja za obje strane te relacije, već samo za sloj pogleda.

Model pogleda nema sučelje, već se na njemu funkcije direktno pozivaju, ali se ujedno i omogućuje pristup *ViewState* objektu koji sadrži sve informacije vezane za prikaz sučelja. Samim time je moguće izbaciti sve funkcije koje prikazuju podatke na sučelju, dok se funkcije za navigaciju i prikaz grešaka zadržavaju.

Nakon što se pokrene *ViewModel* varijabla unutar fragmenta, sloj pogleda se odmah pretplati na promjene *ViewState* objekta, pomoću klase *LiveData* također iz biblioteke *Architecture Components,* koja drži uvijek vrijednost zadnje prikazanog podatka, i šalje sve iduće podatke svima koji ju slušaju, samim time osiguravajući da se na svaku promjenu prikazuju svježi podaci. Fragment, nakon prepisivanja u Kotlin izgleda kao na slici 3.21.

```
class PopularMoviesFragment : Fragment(), RefreshablePage, PopularMoviesView {

    private val adapter = MovieAdapter(R.layout.item_movie_popular)
    private val viewModel by viewModel<PopularMoviesViewModel>()

    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?): View? {
        return inflater.inflate(R.layout.fragment_movies, container,  attachToRoot: false)
    }

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)
        initUi()

        viewModel.setView(this)
        viewModel.viewState().observe( owner: this, Observer { it?.run(::showData) })
    }

    private fun showData(moviesViewState: MoviesViewState) {
        adapter.setData(moviesViewState.movies)
    }

    private fun initUi() {
        movies.itemAnimator = DefaultItemAnimator()
        movies.layoutManager = LinearLayoutManager(activity)
        movies.adapter = adapter
        movies.addOnScrollListener(LastItemReachedListener { viewModel.getMovies(NetworkingUtils.hasInternet(activity)) })
    }

    override fun refresh() = viewModel.refresh(NetworkingUtils.hasInternet(activity))

    override fun showServerError() = Toast.makeText(activity, ..., Toast.LENGTH_SHORT).show()
    override fun showNetworkError() = Toast.makeText(activity, ..., Toast.LENGTH_SHORT).show()
    override fun showGeneralError() = Toast.makeText(activity, ..., Toast.LENGTH_SHORT).show()
}
```

Slika 3.21: Sloj pogleda u MVVM inačici i Kotlinu

Uz puno manje koda se postiže jednaka funkcionalnost kao u MVP obrascu i Java programskom jeziku, pokazujući zapravo da je ovo optimalan pristup za razvoj aplikacija, kad su vrijeme razvoja i čistoća koda bitni. Sloj *poslovne logike* je isto skraćen i bolje organiziran. Kod za sloj *poslovne logike* se nalazi na slici 3.22.

```
private const val MOVIE_TYPE = "POPULAR"

class PopularMoviesViewModel(private val databaseInterface: DatabaseInterface,
                             private val movieInteractor: MovieInteractor) : ViewModel() {

    private var page = 1
    private lateinit var view: PopularMoviesView

    private val viewStateData = MutableLiveData<MoviesViewState>()

    fun setView(view: PopularMoviesView) {
        this.view = view
    }

    fun viewState() = viewStateData

    fun getMovies(hasInternet: Boolean) = if (page == 1 && !hasInternet) {
        val data = MoviesViewState(databaseInterface.getMoviesByType(MOVIE_TYPE))
        viewStateData.value = data
    } else {
        movieInteractor.getMovies(page, MOVIE_TYPE, getCallback())
    }

    fun refresh(hasInternet: Boolean) {
        page = 1
        getMovies(hasInternet)
    }

    private fun getCallback(): Callback<MovieList> = object : Callback<MovieList> {
        override fun onResponse(call: Call<MovieList>, response: Response<MovieList>) {
            response.body()?.movies?.run { if (!isEmpty()) showData( movies: this) }
        }

        override fun onFailure(call: Call<MovieList>, error: Throwable) = when (error) {
            is HttpException -> showServerError()
            is IOException -> showNetworkError()
            else -> showGeneralError()
        }
    }
```

Slika 3.22: Sloj poslovne logike, u Kotlinu i MVVM inačici rada

U Kotlinu, svaka struktura kontrole tijeka programa je izraz (engl. *expression*). Koristeći izraze, moguće je skratiti i uljepšati izgled funkcija i velikih provjera, ne gubeći mogućnosti jezika. Ostatak koda koji obrađuje podatke i prikazuje greške se nalazi na slici 3.23.

```
private fun showServerError() = view.showServerError()
private fun showNetworkError() = view.showNetworkError()
private fun showGeneralError() = view.showGeneralError()

private fun showData(newMovies: List<Movie>) {
    newMovies.forEach { it.type = MOVIE_TYPE }

    if (page == 1) {
        databaseInterface.clearMoviesByType(MOVIE_TYPE)
        databaseInterface.addMovies(newMovies)
        viewStateData.value = null
    }
    val data = viewStateData.value ?: MoviesViewState()
    data.copy(movies = data.movies + newMovies)

    viewStateData.value = data

    page++
}
```

Slika 3.23: Kod koji obrađuje podatke i javlja greške sloju pogleda

Vidljivo je kako je cjelokupan kod za postizanje istih rezultata prepolovljen u Kotlinu, a da MVVM pristup slušanja tokova podataka odstranjuje potrebu za velikim brojem funkcija koje prikazuju podatke, umanjujući kod i dalje. Na samom kraju je kod ujedno i čišći, jer je zadaća sloja *pogleda* da prikazuje podatke, a samim time i da odlučuje na koji način će ih prikazati, bez potrebe za brojnim funkcijama koje mu to govore.

Iako je ovo vrlo lagana verzija MVVM obrasca, najbitnija stavka je postavljena, a to je slušanje podataka umjesto zapovjednog načina rada. Koristeći ViewModel i LiveData. Dodatni koraci MVVM obrasca i reaktivnog pristupa bi bili da se cjelokupan sloj *modela* pretvori obuhvati unutar *LiveData* ili *RxJava* struktura podataka, kako bi se uključili mehanizmi poput ponovnih pokušaja za zahtjeve (engl. *retry logic*), automatske provjere i pretvaranje grešaka i podataka u druge strukture za lakši rad i slično. Klasičan MVVM ne govori o referenci na sloj *pogleda* u sloju *modela* pogleda te ovaj pristup kao takav nije čist MVVM. Ovaj pristup je nadopunjeni MVVM, koji uzima neke stavke MVP obrasca. Hibridi poput ovog su vrlo česti na tržištu jer uz male preinake oba principa i spajanje istih se postiže visok standard u razvoju aplikacija i jednostavnost pisanja koda programerima.

3.7. Usporedba inačica i rad aplikacije

Vrlo je bitno, nakon implementacije programskog rješenja, pokazati usporedbu između inačica aplikacije. Obzirom da su uspoređene arhitekture i programski jezici, najbolji način za izraziti metrike je pomoću testova i količine koda.

Tablica 3.1: Razlike u inačicama aplikacije

	Naivna inačica	MVP inačica	MVVM inačica
Broj linija koda	384	477	307
Broj funkcija	129	236	158
Broj klasa	46	74	58
Testabilnost	Ne	Da	Da, uz postavljanje testova

Na tablici je vidljivo kako najmanje klasa postoji u naivnoj implementaciji, što je i očekivano, obzirom da u takvoj implementaciji ne postoje sučelja za apstrakciju između slojeva, jer slojeva kao takvih nema. Testiranje nije moguće jer je kod poslovne logike teško vezan za kod sloja *pogleda*.

Nadalje, zanimljivo je vidjeti kako MVP inačica ima najviše koda u cijelosti. Uzrok tome je činjenica da je MVP inačica samo proširena naivna implementacija. Dodavanjem slojeva i brojnih klasa sučelja se drastično povećava količina koda, ali je ipak korist to što je kod moguće testirati. Testovi u MVP implementaciji su vidljivi na slici 3.24.

```
@RunWith(MockitoJUnitRunner.class)
public class PopularMoviesPresenterTest {

    @Mock
    MovieInteractor movieInteractor;

    @Mock
    DatabaseInterface databaseInterface;

    private PopularMoviesPresenter presenter;

    @Before
    public void setUp() throws Exception {
        presenter = new PopularMoviesPresenter(movieInteractor, databaseInterface);
    }

    @Test
    public void getMoviesFromInternet() {
        presenter.getMovies( hasInternet: true);

        verify(movieInteractor).getMovies(anyInt(), anyString(), any(Callback.class));
        verifyNoMoreInteractions(movieInteractor);
    }
}
```

Slika 3.24: Test za prezenter klasu u MVP inačici

U MVP inačici se ne provjerava tok podataka, niti kakvi podaci se šalju ili primaju, već interakcija između ovisnosti. Obzirom da se nigdje ne drže podaci koji se šalju, ne mogu se provjeriti trenutna stanja podataka koji su spremni za sloj *pogleda,* već samo akcije koju svaka od ovisnosti izvrši.

Daleko najbolja inačica, što se tiče količine koda, jest MVVM. Smanjivanjem broja funkciju za prikazivanje podataka na jednu, koja prima VS objekt sa svim podacima te korištenjem Kotlina se postiže puno manji broj funkcija i linija koda potrebnih za jednako ponašanje. Također, obzirom da se ne koristi generiranje koda, poput ButterKnife-a ili Dagger-a, smanjuje se i broj klasa, te se smanjuju vrijeme potrebno za izgradnju projekta.

Uz svu korist, MVVM povlači jedan problem za sobom, a to je postavljanje testova. Korištenje LiveData struktura iz Android ArchitectureComponent biblioteke zahtijeva dodatan rad kako bi se kod mogao testirati. Ali je, zbog lakoće pisanja testova nakon postavljanja, i količine smanjenog koda, korist puno veća nego što je dodano posla. Test u MVVM inačici se izvršava uz dodatno postavljanje pomoću varijabli pravila (engl. *rule*).

Ta dodatna varijabla pravila omogućuje da se test izvodi bez dodatne pomoći od Android klase *Looper* koja je zadužena za rad s nitima, porukama i izvođenju programa

unutar Android ekosustava. Obzirom da se testovi ne izvršavaju na Android uređaju, a sami *LiveData* objekti se vežu za i zahtijevaju Android sustav, potrebno je "zavarati" test, da se izvršava u stvarnom vremenu, bez oslanjanja na Android. Test, i njegovi ciljevi su vidljivi na slici 3.25.

```
class PopularMoviesViewModelTest {

  private lateinit var viewModel: PopularMoviesViewModel

  private val database = mock<DatabaseInterface>()
  private val interactor = mock<MovieInteractor>()

  @Rule
  @JvmField
  val instantExecutorRule = InstantTaskExecutorRule()

  @Before
  fun setUp() {
    viewModel = PopularMoviesViewModel(database, interactor)
  }

  @Test
  fun `get movies from database`() {
    val movie = Movie( id: "", title: "", description: "",
          numberOfVotes: 0, averageScore: 5f, image: "",
          releaseDate: "", popularity: "500", type: "")

    val moviesResult :List<Movie> = listOf(movie)

    whenever(database.getMoviesByType(any())).thenReturn(moviesResult)

    viewModel.getMovies( hasInternet: false)

    assertNotNull(viewModel.viewStateData.value)
    assertEquals(moviesResult, viewModel.viewStateData.value?.movies)
    assertEquals( expected: false, viewModel.viewStateData.value?.isLoading)
  }
}
```

Slika 3.25: Test s dodatnim postavljanjem, u MVVM inačici

U MVVM implementaciji, obzirom da se podaci za sloj *pogleda* drže u obliku stanja pogleda ili VS, same interakcije između ovisnosti nisu bitne, iako se mogu testirati. Bitnije je provjeriti koje su trenutne vrijednosti stanja pogleda, nakon izvršavanja neke od operacija, u ovom slučaju dohvaćanja podataka iz baze.

4. ZAKLJUČAK

U ovom završnom radu je pokazano na koje sve načine se može strukturirati kod, ovisnosti i kako je sve moguće prenositi podatke i poruke unutar Android aplikacije. Na primjeru jednostavne aplikacije su se pokazali brojni česti korisnički slučajevi u proizvodnim aplikacijama i obrasci razvoja programske podrške vezani za te slučajeve. Također je pokazana bitna razlika u pisanju koda za Android platformu s programskim jezicima Java i Kotlin te zašto je Kotlin optimalno rješenje za sve nove Android pothvate.

Najviše truda je iziskivalo pravilno pisanje koda, kako bi projekt bio proširiv i lagan za korištenje u budućim iteracijama, ali je na kraju dobiven kvalitetan proizvod zahvaljujući SOLID principima rada, čistoj arhitekturi i kreatoru istih - Uncle Bobu. Dodatni zasloni unutar aplikacije kojima bi se aplikacija mogla obogatiti su zasloni poput registracije i prijave korisnika u vlastiti sustav aplikacije, korisnički profil, oglasna ploča gdje bi korisnici mogli razmjenjivati mišljenja o filmovima i drugi.

Mogućnosti kojima bi se ovaj projekt mogao dodatno poboljšati, s tehničke strane, bi bile detaljnije izvršenje povezivanja komponenti aplikacije pomoću biblioteka RxJava i RxKotlin, integracija funkcijske paradigme programiranje pomoću biblioteke Arrow i Kotlin korutina, potpuno pokrivanje aplikacijskog koda s jediničnim testovima i testovima korisničkog sučelja, korištenje modernijih mogućnosti Androida poput strojnog učenja, umjetne inteligencije, proširene i/ili virtualne realnosti i slično.

LITERATURA

[1] Google, Android is for everyone, https://www.android.com/everyone/, pristupljeno: 29. srpanj, 2018.

[2] Statcounter, Mobile Operating System Market Share Worldwide, http://gs.statcounter.com/os-market-share/mobile/worldwide, pristupljeno: 29. srpanj, 2018.

[3] Google, Android Operating System, Google, https://www.android.com/, pristupljeno: 22. lipanj, 2018.

[4] R. C. Martin, Design principles and Design Patterns, objavljeno na: www.objectmentor.com, 2000.

[5] R.C. Martin, Clean Architecture: A Craftsman's Guide to Software Structure and Design, O'Reilly Media, Sebastopol, California, Sjedinjene Američke Države, 2008.

[6] R.C. Martin, Clean Code Handbook Software Craftsmanship, O'Reilly Media, Sebastopol, California, Sjedinjene Američke Države, 2017.

[7] Robert Cecil Martin, InformIt, https://www.informit.com/authors/bio/361a5e70-f1e2-432b-9928-b30b4742ae80, pristupljeno: lipanj, 2018.

[8] What is a naïve method, Computer Science, Stack Exchange, https://cs.stackexchange.com/questions/33914/what-is-a-naive-method, pristupljeno: 29. srpanj, 2018.

[9] The difference between loose coupling and tight coupling in OOP, Stack Exchange, https://stackoverflow.com/questions/2832017/what-is-the-difference-between-loose-coupling-and-tight-coupling-in-the-object-o, pristupljeno: 29. srpanj, 2018.

[10] What is Cognitive Load, The eLearning Coach, 2011., dostupno na: http://theelearningcoach.com/learning/what-is-cognitive-load/, pristupljeno: 29. srpanj, 2018.

[11] Model-View-Controller Pattern, Codecademy, https://www.codecademy.com/articles/mvc, pristupljeno: 22. lipanj, 2018.

[12] U. Barbini, Weak, Soft and Phantom references in Java (and why they matter), DZone, 2017., dostupno na: https://dzone.com/articles/weak-soft-and-phantom-references-in-java-and-why-they-matter, pristupljeno: 29. srpanj, 2018.

[13] F. Cervone, Model-View-Presenter Pattern: Android guidelines, Medium, 2017., dostupno na: https://medium.com/@cervonefrancesco/model-view-presenter-android-guidelines-94970b430ddf, pristupljeno: 22. lipanj, 2018.

[14] What is boilerplate code, Stack Exchange, https://stackoverflow.com/questions/3992199/what-is-boilerplate-code, pristupljeno: 29. srpanj, 2018.
[15] J. Shvarts, Implementing MVVM using LiveData, RxJava, Dagger Android, Medium, 2017., dostupno na: https://proandroiddev.com/mvvm-architecture-using-livedata-rxjava-and-new-dagger-android-injection-639837b1eb6c, pristupljeno: 22. lipanj, 2018.
[16] Atlassian Inc., Software testing, https://www.atlassian.com/software-testing, pristupljeno: 22. lipanj, 2018.
[17] AgileData, Test Driven Development, http://agiledata.org/essays/tdd.html, pristupljeno: 22. lipanj, 2018.
[18] Gradle Inc., Gradle build tool, dostupno na: https://gradle.org/, pristupljeno: 22. lipanj, 2018.
[19] Google (Alphabet Inc.), Dagger2, dostupno na: https://google.github.io/dagger/, pristupljeno: 22. lipanj, 2018.
[20] Google (Alphabet Inc.), Android Architecture Components, dostupno na: https://developer.android.com/topic/libraries/architecture/, pristupljeno: 22. lipanj, 2018.
[21] A. Giuliani, Koin, dostupno na: https://github.com/Ekito/koin, pristupljeno: 22. lipanj, 2018.
[22] Square Inc., Retrofit Android, dostupno na: http://square.github.io/retrofit/, pristupljeno 22. lipanj, 2018.
[23] K.Beck, E.Gamma, D. Saff, M. Clark, JUnit 5, dostupno na: https://junit.org/junit5/, pristupljeno 23. lipanj, 2018.
[24] Otvoren kod, više autora, Mockito, dostupno na: http://site.mockito.org/, pristupljen 23. lipanj, 2018.
[25] Square Inc., LeakCanary, dostupno na: https://github.com/square/leakcanary, pristupljeno 23. lipanj, 2018.
[26] Anoniman autor, JSON, dostupno na: https://www.json.org/, pristupljeno: 23. lipanj, 2018.
[27] Oracle Corporation, Data Access Object, dostupno na: http://www.oracle.com/technetwork/java/dataaccessobject-138824.html, pristupljeno: 23. lipanj 2018.

[28] E. Gamma, R. Helm, R. Johnson, J. Vlissides (*Gang of Four*), Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Boston, Massachusetts, Sjedinjene Američke Države, 1994.

SAŽETAK

Cilj ovog završnog rada je usporediti popularne arhitekturne obrasce za razvoj programske podrške odnosno njihovu primjenu u Androidu, u trenutno najzastupljenijim programskim jezicima za razvoj Android platforme, i dati uvid u prednosti i probleme svakog od obrazaca, pritom koristeći najbolje prakse i oblikovne obrasce za pisanje programskog koda. Rad je detaljno opisao postupke implementacije najpopularnijih arhitekturnih obrazaca u Android zajednici, razliku između istih i benefit čistog pristupa razvoju programske podrške. Osim toga, dan je jasan pregled svrhe svakog od pristupa prema strukturiranju aplikacije te koji problemi pritom nastaju. Za kreiranje projekta je korišteno razvojno okruženje Android Studio, a kod projekta je pisan u programskim jezicima Java i Kotlin.

Ključne riječi: Android, MVP, MVVM, Oblikovni obrasci, SOLID

ABSTRACT

The goal of this thesis is the comparison of popular software development architecture patterns, or precisely their usage in Android, using the most common programming languages for the development on the Android platform, and to give insight into the benefits and issues of each of the processed patterns, whilst using the best practices and design patterns when writing programming code. The tesis gave detailed insight into the process of implementing said patterns, it has shown the difference between each of the patterns and the benefit of a clean development approach, when writing software. Furthermore, it’s been elaborated which use-cases are covered by each of the patterns, what their purpose is and what problems arise during the implementation. To create the project, Android Studio development environment was used, and the code was written in programming languages Java and Kotlin.

Key words: Android, Design patterns, MVP, MVVM, SOLID

ŽIVOTOPIS

Filip Babić rođen je 11.12.1996. u Vinkovcima. Pohađao je osnovnu školu „August Cesarec“, u Ivankovu. Nakon osnovne škole upisao je „Gimnaziju Matije Antuna Reljkovića“, u Vinkovcima, prirodoslovno-matematički smjer. Trenutno studira na „Fakultetu elektrotehnike, računarstva i informacijskih tehnologija Osijek“, u sklopu Sveučilišta Josipa Jurja Strossmayera u Osijeku gdje je upisao preddiplomski studij, smjer računarstvo. Radio je kao student Android developer u tvrtci COBE d.o.o. dvije i pol godine nakon čega prelazi u tvrtku 5 minuta (Five agency), gdje radi kao Android developer. Također je zaposlen kao privatni izvođač radova, pišući i uređivajući Android vodiče za RayWenderlich tim. Nadalje, u sklopu RayWenderlich tima piše knjigu o programskom jeziku Kotlin. Osim toga je aktivan predavač i organizator u sklopu zajednice Osijek Software City.

PRILOZI

1. Arhitektura Android aplikacija u .docx formatu
2. Arhitektura Android aplikacija u .pdf formatu
3. Izvorni kod