Usporedba mogućnosti dizajna korisničkih sučelja za Android pomoću XML-a i Jetpack Composea

Godek, Tomislav

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Tomislav Godek

USPOREDBA MOGUĆNOSTI DIZAJNA KORISNIČKIH SUČELJA ZA ANDROID POMOĆU XML-A I JETPACK COMPOSEA

DIPLOMSKI RAD

Varaždin, 2022.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Tomislav Godek

Matični broj: 48779

Studij: Baze podataka i baze znanja

USPOREDBA MOGUĆNOSTI DIZAJNA KOSNIČKIH SUČELJA ZA ANDROID POMOĆU XML-A I JETPACK COMPOSEA

DIPLOMSKI RAD

Mentor:

Izv. prof. dr. sc. Zlatko Stapić

Varaždin, studeni 2022.

Tomislav Godek

Izjava o izvornosti

Izjavljujem da je moj diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

Novi i moderni set alata za razvoj Android korisničkog sučelja naziva se *Jetpack Compose*. Razvijen je s ciljem pojednostavljenja i ubrzanja izgradnje korisničkih sučelja. Ipak, pitanje je u kojoj mjeri ova nova tehnologija može zamijeniti sve mogućnosti dosadašnje tehnologije opisa korisničkog sučelja pomoću XML jezika. Cilj ovog diplomskog rada teorijski je opisati i praktično prikazati mogućnosti tehnologije *Jetpack Composa* te definirati kriterije i usporediti spomenutu tehnologiju s XML dizajnom.

U samom početku ovog rada opisan je pojam korisničkog sučelja i prikazan način kako kreirati dobra korisnička sučelja. Putem različitih primjera objasnili smo koje sve pred procese i korake moramo proći prije negoli uopće krenemo izrađivati korisnička sučelja za ciljanu skupinu korisnika. Nakon teorijskog opisa korisničkoga sučelja u radu je prikazan način kako kreirati korisnička sučelja za Android putem XML-a i *Jetpack Composea*. Opisani su osnovni koncepti obiju tehnologija i prikazani primjeri najčešće korištenih elementa korisničkog sučelja. Za praktični dio rada kreirane su dvije verzije aplikacije, jedna izrađena u XML-u, a druga putem *Jetpack Composea.* Kako bi znali koja tehnologija omogućava brži i jednostavniji razvoj mobilnih aplikaciji uspoređeni su različiti aspekti aplikacije. Tako je u praktičnom dijelu uspoređen način kreiranja složene liste elementa te glavne navigacijske komponente. Zatim je uspoređena lakoća ponovne iskoristivi određenih UI elemenata te je na kraju prikazan način kreiranja teme i stila aplikacije.

Ključne riječi: android, razvoj mobilnih aplikacija, *Jetpack Compose*, dizajn korisničkih sučelja

Sadržaj

1. Uvod

Danas smo u interakciji sa softverom u gotovo svakom aspektu našeg svakodnevnog života: posao, slobodno vrijeme, komunikacija, kupovina, učenje itd. Popis uređaja i stvari s „pametnim“ softverom i internetskom vezom sve više raste. Tako danas imamo automobile, pametne zvučnike, televizore, satove i domove koji sadrže određene „pametne“ značajke kako bi korisniku što više olakšali svakodnevni život i pružali što više relevantnih informacija. Za većinu svjetskog stanovništva pametni telefon je i dalje glavno sredstvo komunikacije i interakcije s različitim virtualnim i stvarnim objektima. Globalno gledano, više od polovice svjetskog stanovništva ima pristup internetu i gotovo je nezamislivo vidjeti osobu bez pametnog telefona. Veličine i vrste zaslona variraju, a eksplozija sučelja koja su prvenstveno gesta ili glas konstantno evoluiraju. Konačno, softver postaje moćniji, predvidljiviji, sposobniji ponuditi pametnije uvide i djelovati neovisnije. Jednom riječju, postaje sve više poput nas.

Dizajn sučelja, kao i sve ostalo, mijenja se kako bi išao u korak s vremenom. Sa sve većim zahtjevima dolazi i potrebna za kreiranje sve složenijih korisnička sučelja. Kako bi razvojni inženjeri mogli isporučiti što kvalitetniji proizvod, potreban je set alata koji će omogućiti izradu korisničkih sučelja koja su istovremeno dinamična, jednostavna za implementirati i lagana za izmijeniti. Kako ovaj diplomski rad obrađuje mobilnu platformu za Android operativni sustav, istraženi su sustavi koji će omogućiti kreiranje modernih korisničkih sučelja za navedenu platformu. Jedan od tih alata koji omogućuje kreiranje nativnih korisničkih sučelja za Android na vrlo reaktivan i jednostavan način zove se *Jetpack Compose*. Osmišljen je od strane *Googlea*, a prva stabilna verzija objavljena je u srpnju 2021. godine. Ovim radom želi se usporediti dosadašnji razvoj korisničkih sučelja za Android u XML-u (eng. *Extensible Markup Language*) s novim načinom koji pruža *Jetpack Compose*. Glavni cilj ovog diplomskog rada teorijski je opisati i praktično pokazati mogućnosti tehnologije *Jetpack Composea* te definirati kriterije i usporediti spomenutu tehnologiju s XML dizajnom. Način na koji će se to postići je kreiranjem mobilne aplikacije koja će sadržavati osnovne elemente koja svaka aplikacija danas sadrži.

Istraživanjem najpopularnijih mobilnih aplikacija trenutno na tržištu poput *Facebook-a*, *Instagram-a*, *Netflix-a, Twitter-a,* zaključio sam da gotovo svaka mobilna aplikacija sadrži prikaz određene liste sadržaja kao i prikaz detalja same liste. Jedan od vrlo popularnih komponenata u aplikaciji koji vrlo dobro iskorištava prikaz liste je pretraživanje određenog sadržaja. Korisniku se u takvim scenarijima uglavnom nudi polje za unos teksta i aplikacija će korisniku vratiti rezultat u obliku uređene liste sadržaja. Svaka aplikacija također sadrži određeno centralo mjesto, uglavnom na samom početnom zaslonu, a pruža pregled najbitnijih elemenata u obliku kartica koje predstavljaju sažeti prikaz detalja određenog sadržaja.

Naravno, nezaboravni dio svake aplikacije je glavna navigacijska komponenta koja se danas u velikom postotku aplikacija prikazuje kao lista ikona na donjem djelu ekrana.

Rezultat svega navedenog je izrada mobilne aplikacije pod nazivom *Sofa Time* kao praktični dio izrade diplomskog rada. Aplikacija će korisnicima omogućiti pretraživanje i praćenje njihovih najomiljenijih tv emisija preko intuitivnog i jednostavnog korisničkog sučelja. Kako bi usporediti implementaciju starog načina kreiranja sučelja za Android i novog načina, implementirane su dvije verzije aplikacije. Prva verzija sadrži implementaciju korisničkog sučelja putem XML-a što je u trenutku pisanja ovog rada i dalje najpopularniji izbor prilikom izrade korisničkih sučelja mobilnih aplikacija koje se pokreću na Android operativnom sustavu. Druga verzija aplikacije u potpunosti je implementirana uz pomoć *Jetpack Composea*. Kroz implementaciju aplikacije prikazani su najosnovniji UI (eng. *User Interface*) elementi koje nudi ovaj set alata.

Motivaciju za pisanje ovog rada dobio sam nakon testiranja *Jetpack Composea* prilikom izlaska prve stabilne verzije. Pozitivno me iznenadio deklarativni pristup izradi korisničkih sučelja naspram starog imperativnog pristupa što je uvelike smanjivo vrijeme potrebno za implementaciju UI elemenata. Određeni elementi poput navigacijske komponente u trenutku pisanja ovog rada sadrži ograničene mogućnosti te ovim radom želim predstaviti sve prednosti i mane koje nosi sa sobom *Googleov* novi set alata. Ovaj rad namijenjen je svima koji se dvoume između prelaska iz XML-a u *Jetpack Compose* ili žele istražiti mogućnosti oba alata i na kraju odlučiti za sebe što je po njima najisplativiji način izrade nativnih korisničkih sučelja za Android.

Kompletni sadržaj ovog diplomskog rada podijeljen je u šest poglavlja. Uvodno poglavlje opisuje svrhu i cilj diplomskog rada te sadrži kratak opis cijelog rada. Drugo poglavlje sadrži opis metoda i tehnika koji su primijenjeni u ovom radu. U trećem poglavlju objašnjen je pojam korisničkog sučelja i načina kako kreirati dobra korisnička sučelja. Četvrtim poglavljem prikazali smo načine kreiranja korisničkih sučelja uz pomoć XML-a i *Jetpack Composea*. U prvom dijelu četvrtog poglavlja ukratko je objašnjen sam koncept *Material Designa* i koje benefite pruža. Zatim je opisan stari *View* sustav u Androidu i objašnjen životni ciklus aplikacije. Dan je detaljan uvid u kreiranje hijerarhije XML resursa i načina kako ih povezati s aktivnostima i fragmentima. Drugi dio poglavlja pokriva deklarativni set alata zvan *Jetpack Compose* u kojemu su predstavljeni ključni elementi na koje se oslanja navedeni skup alata. Preko primjera prikazane su osnovne *Composable* metode i načini kako putem *Modifier* parametra promijeniti izgled *Composable* elemenata. U petom poglavlju putem praktičnog rada uspoređen je način kreiranja najčešće korištenih UI komponenti poput liste koja prikazuje različiti tip sadržaja i donje navigacijske trake (eng. *Bottom Navigation Bar*). Također, ovim poglavljem je uspoređena lakoća ponovno iskoristivih UI elementa te kreiranja stila i teme aplikacije. Zadnje

poglavlje sadrži zaključak, odnosno sažeti opis cijelog rada i autorov komentar na temu diplomskog rada.

2. Metode i tehnike rada

Budući da ovaj seminarski rad pokriva tehnologiju koja je u vrijeme pisanja rada relativno nova i u vrlo aktivnom razvoju, najviše je korištena metoda istraživanja putem interneta za teorijske cjeline koje se odnose na *Jetpack Compose*. Metoda pregleda literature često je korištena za cjeline koje su vezane za razvoj aplikacija za Android putem XML-a, te prilikom izrade poglavlja vezanih za teoriju korisničkog sučelja. Metodom analize sastavljen je popis funkcionalnosti koje sadrži programsko rješenje ovog diplomskog rada. Sva poglavlja vezana uz opis programskog rješenja izrađena su uz pomoć tehnike programiranja i dizajna programskog rješenja. Programski kôd napisan je pomoću metode analize postojećih programskih rješenja napisanih u Kotlinu, te analizom *Googleove* službene dokumentacije za Android.

Za izradu ovog diplomskog rada korišten je program za uređivanje teksta naziva *Microsoft Word*. Alat se koristio za pisanje same teorije te opis cjelokupnog praktičnog rada. U sklopu rada izrađeno je programsko rješenje naziva *Sofa Time* napisano u programskom jeziku Kotlin. Za samu izradu aplikacije korišten je *Android Studio* verzije 2021.2.1, službeno razvojno okruženje za *Googleov* operativni sustav Android. Aplikacija je razvijenu i testirana na emulatoru te fizičkog uređaju.

3. Korisnička sučelja

Kada razmišljate o pojmu *korisničko sučelje* što Vam je prva asocijacija? Često ćete dobiti odgovor da je to sve što diramo po mobilnom uređaju ili „klikamo“ po web stranici. Možemo zaključiti da se radi o svemu što korisnik vidi na mobilnom uređaju ili računalu, a služi za interakciju između korisnika i programa.

Što sve spada u korisničko sučelje, na koje načine možemo ostvariti komunikaciju s objektima za koje smatramo da spadaju u korisnička sučelja i kako kreirati dobra korisnička sučelja opisano je u ovom poglavlju.

3.1. Što su korisnička sučelja?

Korisnička sučelja su pristupne točke na kojima korisnici ostvaruju interakciju s dizajnom (Tidwell i ostali, 2019). Dizajn korisničkog sučelja je proces koji dizajneri koriste za izradu sučelja u softveru ili računalnim uređajima, fokusirajući se na izgled ili stil. Dizajneri nastoje stvoriti sučelja koja će korisnicima biti laka za korištenje i koja će biti ugodna, te zahtijevaju minimalan napor od strane korisnika kako bi se dobio maksimalni željeni rezultat (Interaction Design Foundation, 2022).

Tipično korisnička sučelja promatramo kao grafička korisnička sučelja, ali mogu biti i u drugom obliku kao npr. sučelja upravljanja glasom. Time možemo korisničko sučelje podijeliti u slojeve interakcije koji privlače ljudska osjetila (vid, dodir, sluh) (Interaction Design Foundation, 2022). Oni uključuju i ulazne uređaje kao što su tipkovnica, miš, mikrofon, zaslon osjetljiv na dodir, skener otiska prsta i kamera, te izlazne uređaje kao što su monitori, zvučnici i pisači. U ovom radu naviše ćemo se posvetiti zaslonima osjetljivim na dodir budući da se tema odnosi isključivo na mobilne uređaje, a isto tako zasloni će ostati još dugo kao glavni dizajn korisničkog sučelja na mobilnim uređajima. Vjerujem da će ih biti još više, samo u različitim veličinama i oblicima. Zapravo, složenost onoga što trebamo prikazati na tim zaslonima sve više i više raste.

U sljedećoj cjelini dan je odgovor na pitanje kako kreirati dobra korisnička sučelja za korisnike i na što sve trebamo pripaziti prije negoli uopće krenemo kreirati sama korisnička sučelja.

3.2. Kako kreirati dobra korisnička sučelja za korisnike

Trenutno, dizajneri i programeri imaju veliki izbor i pristup raznim alatima za izradu i dizajn kvalitetnog softvera. Svijet dizajna i softvera razvio se u pristup sustavima, komponentama i modulima. Počinjanje od nule do dizajniranja ili kodiranja nečeg potpuno novog više nije norma. Postoje brojni setovi alata koji omogućuju stvaranje sučelja temeljenih na zaslonu a istodobno omogućuju odlično prilagođavanje različitim veličinama zaslona. Ove biblioteke komponenti treba promatrati kao način da se brzo dođe do osnovne baze dizajna korisničkog sučelja. (Tidwell i ostali, 2019).

Prema Tidwell-u i ostalima u knjizi „*Designing Interfaces 3rd Edition",* prije samom odabira alata i kreiranja korisničkog sučelja postoji četiri neophodna koraka koja svaki dizajner mora ispuniti:

1. mora razumjeti ljude koji će koristiti naše sučelje
2. mora razumjeti ciljeve korisnika i razumjeti što žele postići putem korisničkog sučelja
3. mora okarakterizirati vrste ljudi koji će koristiti naš dizajn
4. mora prepoznati obrasce ljudskog ponašanja

Kroz ovo poglavlje objasniti ćemo što svaki od koraka zapravo znači te putem primjera predstaviti rješenja za svaki korak.

Dobro korisničko sučelje ne započinje s ubacivanjem slika, definiranjem boja ili pak kreiranjem osnovnih UI elemenata. Počinje s razumijevanjem ljudi što je ujedno i prvi korak u dizajniranju korisničkih sučelja za ljude. Potrebno je saznati kako oni razmišljaju, zbog čega koriste određeni dio programa i na koji način stupaju u interakciju s promatranim objektom (Tidwell i ostali, 2019). Što više znamo o ljudima koji koriste našu aplikaciju i što više suosjećamo s njima, učinkovitije možemo dizajnirati za njih. Softver je, naposljetku, samo sredstvo za postizanje cilja za ljude koji ga koriste.

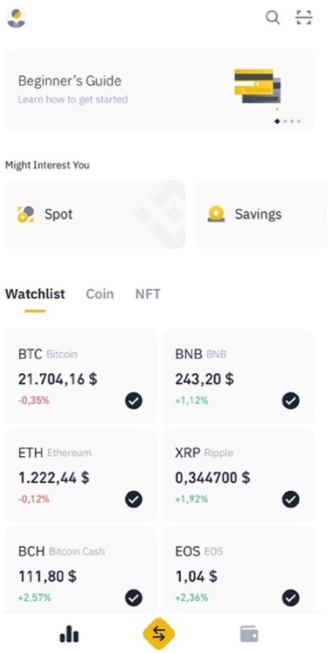
Ne postoje stroga pravila za kreiranje dobrog korisničkog sučelja, ali postoje određene strukture koje nam mogu pomoći u razumijevanju dizajna za ljude. Prvi korak u dizajniranju za ljude je razumijevanje ljudskog konteksta za namjenu našeg dizajna. Dizajn interakcije počinje definiranjem i razumijevanjem ljudi koji će koristiti naš dizajn. Konkretnije, potrebno je utemeljiti svoje odluke o dizajnu ovisno o korisnikovom poznavanju relevantnih tema ciljanog dizajna te njihovom stupnju vještine rada sa softverom (Tidwell i ostali, 2019).

Prema Tidwell-u i ostalima u knjizi „*Designing Interfaces 3rd Edition"* postoji poznata izreka u području dizajna a to je „*You are not the user",* odnosno, vi niste korisnik. Često se dogodi da dizajneri toliko zaglave u vlastitom svijetu da ponekad zaborave tko zapravo koristi

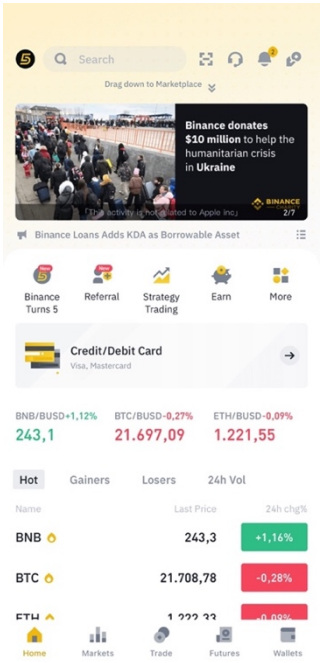
njihov proizvod. Želimo predstaviti podatke u svojoj aplikaciji na način koji će korisnici lako razumjeti. Svaki put kada netko koristi aplikaciju ili bilo koji digitalni proizvod, on „vodi razgovor sa strojem“. Kao dizajner korisničkog sučelja mi skriptiramo taj razgovor ili barem definiramo njegove uvjete. Ako namjeravamo skriptirati razgovor, trebali bi smo što bolje razumjeti ljudsku stranu. Koji su motivi i namjere korisnika? Koji „rječnik" riječi, ikona i gesta korisnik očekuje da koristi? Kako aplikacija može postaviti očekivanja na odgovarajući način za korisnika? Sve su to pitanja koja si moramo postaviti ako želimo imati dobro korisničko sučelje (Tidwell i ostali, 2019).

Dobro korisničko sučelje omogućuje novim korisnicima jednostavno korištenje aplikacije putem *onboarding* procesa ili prikaza jednostavnog seta funkcionalnosti, dok naprednim korisnicima pruža složeniji set alata i funkcionalnosti. Kao dizajner korisničkog sučelja moramo pronaći balans. Početnicima mora pomoći da nauče koristiti aplikaciju dok čestim korisnicima omogućuje nesmetano obavljanje poslova (Babich, 2019).

Izvrstan primjer takvog ponašanja predstavlja aplikacija za trgovanje kripto-valutama zvana *Binance* koja na vrlo efektivan način prikazuje sadržaj ovisno o preferencijama korisnika. Naime kada korisnik preuzme aplikaciju i registrira se, prezentira se zaslon prikazan na slici 1. Ako je korisnik koristio već duže vrijeme aplikaciju i želi koristiti puni potencijal aplikacije može u postavkama aplikacije jednim klikom prebaciti na prikaz za profesionalce čiji dizajn možemo primijetiti na slici 2.



Slika 1: *Binance* - prikaz ekrana za početnike



Slika 2: *Binance* - prikaz ekrana za profesionalce

Time se aplikacija „transformirala“ u *Pro* verziju s naprednijim funkcionalnosti i dodatnim sadržajem. Prema mojem mišljenju ovakav način prezentacije korisničkog sučelja predstavlja odličan način kako novom korisniku prezentirati sadržaj na što jednostavniji način, a u isto vrijeme *Pro* korisniku omogućuje nesmetano korištenje svih naprednih mogućnosti koje aplikacija nudi.

Sljedeći važni korak u izradi dobrog korisničkog sučelja je razumjeti ciljeve korisnika (Tidwell i ostali, 2019). Ciljevi korisnika mogu biti različiti: pronalaženje neke činjenice ili predmeta, žele nešto naučiti, žele provesti neku transakciju, stvoriti nešto ili se jednostavno zabavljati.

Bitan korak u dizajniranju sučelja je naučiti što njegovi korisnici stvarno pokušavaju postići. Ispunjavanje obrasca, na primjer, gotovo nikada nije cilj sam po sebi . Ljudi to čine samo zato što pokušavaju kupiti nešto na internetu, obnoviti vozačku dozvolu ili instalirati softver. Izvode neku vrstu transakcije. Ako postoji način da završe transakciju, a da korisnik uopće ne prođe kroz taj obrazac, potrebno je u potpunosti ukloniti taj postupak. To dovodi korisnika bliže njegovom cilju, uz manje vremena i truda koji utroši (Babich, 2019).

Prođimo kroz jedan jednostavan scenarij. Korisnik pristupa nekoj aplikaciji koja omogućuje rezerviranje avionske karte. U aplikaciji utipkava ime željenog grada kojeg želi posjetiti sa svojom obitelji i pokušava pronaći cijene avionskih karata za razne datume. Uči iz onoga što pronađe, ali njegov cilj nije samo pregledavanje i istraživanje različitih opcija. Njegov cilj je zapravo transakcija: kupiti avionske karte. Opet, mogao je to učiniti preko drugih aplikacija koje pružaju sličnu uslugu ili preko telefona s turističkim agentom uživo. Po čemu je ova aplikacija bolja od onih drugih opcija? Je li brža? Jednostavnija za korištenje? Ima veću vjerojatnost da pronađe bolju ponudu?

Uz sve navedeno nije dovoljno promatrati korisnika kao bezličnog entiteta koji ima niz jednostavnih slučajeva korištenja s jednim usmjerenim zadatkom (Tidwell i ostali, 2019). Da bi dobro izradili dizajn, moramo uzeti u obzir mnoge “mekše” čimbenike: očekivanja, sklonosti, društveni kontekst, različita uvjerenja. Svi ovi čimbenici mogu utjecati na dizajn aplikacije ili stranice. Među tim mekšim čimbenicima možemo pronaći kritičnu značajku ili faktor dizajna koji našu aplikaciju čini privlačnijom i uspješnijom (Tidwell i ostali, 2019).

Treći korak u dizajniranju za ljude te uvjet da bi uopće započeli dizajn je okarakterizirati vrste ljudi koji će koristiti naš dizajn (uključujući upravo spomenute mekše faktore) (Tidwell i ostali, 2019). Svaki korisnik je jedinstven. Ono što jednoj osobi pada teško, drugoj neće. Trik je u tome da shvatimo što je općenito istina o našim korisnicima, a to znači naučiti o dovoljnom broju pojedinačnih korisnika kako bismo izdvojili uobičajene obrasce ponašanja od onih koji odskaču.

Ova faza otkrivanja korisnika potrošit će vrijeme i resurse na početku ciklusa dizajna, pogotovo ako nemamo pojma tko je naša publika i zašto bi mogli koristiti naše dizajne. Dobra praksa je na početku kreirati koncept i pokazati ga ciljanoj skupini ljudi. Na temelju povratnih informacija znati ćemo gdje se nalaze potencijalne promjene u dizajnu korisničkog sučelja. Budući da ćemo dobiti ovu povratnu informaciju vrlo rano, puno je lakše napraviti izmjene u tom trenutku, umjesto da čekamo za promjene dok se aplikacija nalazi u završnim fazama. Važno je da budemo otvoreni za povratne informacije i otvoreni da ih što više tražimo. Neki od načina da otkrijemo ciljane korisnike našeg dizajna je preko intervjua, anketa i kreiranju persona (Tidwell i ostali, 2019).

Četvrti i zadnji element dobrog korisničkog dizajna je prepoznavanje obrazaca ljudskog ponašanja, percepcije i razmišljanja koji su relevantni za dizajniranje sučelja (Tidwell i ostali, 2019). Sučelje koje dobro podržava određene obrasce ponašanja pomoći će korisnicima da postignu svoje ciljeve daleko učinkovitije od sučelja koja ih ne podržavaju. Prikazati ćemo neke od najbitnijih obrazaca, a jedan od njih je „sigurno pretraživanje“. On omogućuje korisniku da bezbrižno istražuje programski paket bez da se izgubi u njemu. Određeni dizajn će zadovoljiti taj obrazac ako korisnik bez poteškoća može proći kroz cijelu aplikaciju u što je uključeno lako navigiranje i intuitivno reagiranje na određene akcije (Tidwell i ostali, 2019).

Sljedeći važan obrazac je „trenutačno zadovoljstvo“. Ako korisnik počne koristiti aplikaciju i doživi "uspješno iskustvo" (eng. *Successful experience*) unutar prvih nekoliko sekundi, to je zadovoljstvo! Veća je vjerojatnost da će ta osoba nastaviti koristiti naš proizvod, čak i ako kasnije postane teže. Osjećati će se sigurnije u primjeni aplikacije te sigurniji u sebe. Aplikacije treba biti dizajnirana tako da u svakom trenutku informira korisnike o tome što se događa putem odgovarajućih povratnih informacija u razumnom vremenskom roku (Nielsen, 2020).

Među važnijim obrascima spada još obrazac pod nazivom „zadovoljavajuće“ (eng. *Satisfying*). Javlja se kada su ljudi spremni prihvatiti "dovoljno dobro" umjesto "najbolje" ako učenje svih alternativa može koštati vremena ili truda. To znači da prilikom dizajna korisničkog sučelja koristimo kratke i jasne opise, koristimo jako uočljive elemente na koje želimo da korisnici kliknu i sučelje bude jednostavno za navigirati jer korisnici uvijek traže prvu stvar koja bi mogla raditi. Postoji još mnogo različitih obrazaca koje nećemo u ovom radu detaljno opisivati ali su izvrsno opisane u knjizi „*Designing Interfaces 3rd Edition“* autora Tidwell-a i ostalih.

Većina principa dizajna korisničkog sučelja navedenih u ovom poglavlju zasigurno će biti primjenjiva prilikom izrade bilo kojeg korisničkog sučelja. Dan je opis na sve što trebamo pripaziti prilikom odabira i izrade korisničkog sučelja, a veliki dio dobrog korisničkog sučelja proizlazi iz poznavanja samih ljudi koji će koristiti našu aplikaciju.

U sljedećem poglavlju opisan je način kreiranja korisničkih sučelja uz pomoć XML-a i *Jetpack Composea*. Opisani su osnovni koncepti svake tehnologije i prikazani primjeri kreiranja osnovnih elemenata korisničkog sučelja.

4. Kreiranje korisničkih sučelja u XML-u i *Jetpack Composeu*

Android je trenutno najpopularniji operacijski sustav za mobilne uređaje. Njegov najveći konkurent je *Apple-ov* iOS koji trenutno zauzima oko 27% tržišta (StatCounter, 2022). U 2022. godini Android i dalje vrlo dominantna mobilnom platformom s gotovo 73% udjela u svjetskom tržištu mobilnih uređaja (StatCounter, 2022).

Najbolji način za razvoj aplikacija koje podržavaju Android je korištenje *Android Studioa*, službenog IDE-a za razvoj mobilnih aplikacija ciljanih korištenju na Android operativnom sustavu. *Android Studio* temelji se na IntelliJ IDEA-u, a uključuje skup uređivača kôda, UI alata i predložaka (Toporov, 2013). Također uključuje Android SDK (eng. *Android Software Development Kit*) koji je potreban za razvoj svih Android aplikacija. Android SDK uključuje Android izvorne datoteke i kompajler koji se koristi za kompiliranje kôda u format koji Android OS „razumije“ (Griffiths & Griffiths, 2021).

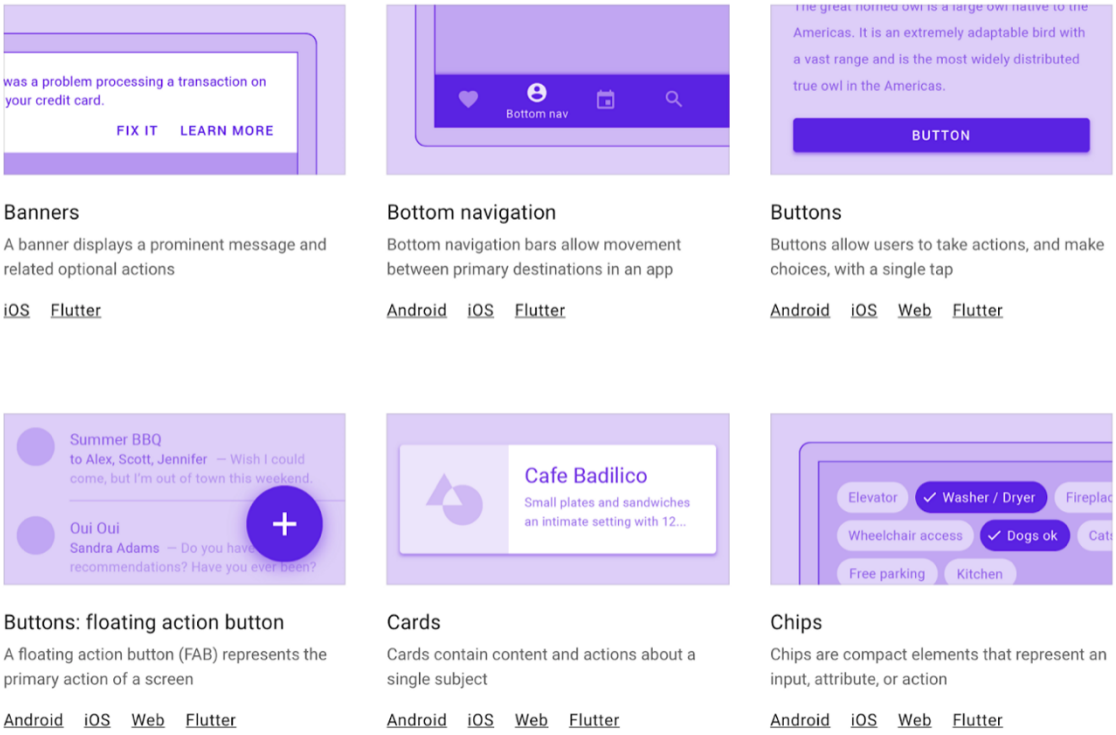
Ovo poglavlje započinje kratkim uvodom u *Googleov* preporučeni način izgradnje korisničkih sučelja prema smjernica zvanim *Material Design*. Ukratko je objašnjen sam koncept *Material Designa* i koje benefite pruža. Glavni fokus ovog poglavlja postavljen je na opis i usporedbu tradicionalnog načina kreiranja korisničkih sučelja za Android uz pomoć XML-a te novog načina uz pomoć *Jetpack Composea*. Prvo je opisan stari *View* sustav u Androidu i objašnjen životni ciklus aplikacije. Dan je detaljan uvid u kreiranju hijerarhije XML resursa i načina kako ih povezati s aktivnostima i fragmentima. Drugi dio poglavlja pokriva deklarativni set alata zvan *Jetpack Compose* u kojemu su predstavljeni ključni elementi na koje se oslanja navedeni skup alata. Detaljno je razrađen princip rada *Composable* metoda i koje su najbolje prakse prilikom izrade UI elemenata u *Jetpack Composeu*. Na kraju cjeline prikazani su osnovni *Composable* elementi i načini kako im promijeniti izgled i ponašanje putem *Modifier* atributa, najvažnijeg atributa svake *Composable* metode.

4.1. *Material Design*

Material Design sveobuhvatan je vodič za vizualni dizajn i interakciju na svim platformama i uređajima (*Material Design*, 2022). Komponente *Material Designa* dizajnerima i programerima nude način implementacije posebnog *Material* dizajna u njihovu aplikaciju. Razvijen od strane glavnog tima inženjera i dizajnera korisničkog sučelja u *Googleu,* ove komponente omogućuju pouzdan tijek razvoja za izradu lijepih i funkcionalnih aplikacija (Figma, 2022). Već dugi niz godina *Material Design* predstavlja smjernice koje pružaju programerima izradu konzistentnost i usklađenih korisnička sučelja, a istovremeno ostavlja dovoljno slobode u izradi personaliziranih dizajna.

Tim iz Googlea koji je razvio *Material Design* preporučuje da aplikacije slijede smjernice materijalnog dizajna kako bi se osiguralo da aplikacije pružaju konzistentno korisničko iskustvo i time uzorci naučeni u jednoj aplikaciji mogu se lako prenijeti na drugu (Chapman, 2022). Komponente materijalnog dizajna pružaju stilizirane verzije standardnih elemenata korisničkog sučelja što programerima znatno smanjuje vrijeme potrebno za izgradnju navedenih elemenata. Tako na primjer postoje stilizirane verzije gumba, alatnih traka, potvrdnih okvira i mnogo drugih komponenti. Primjeri takvih komponenata prikazani su na slici 3. Posebna korist u korištenju *Material Designa* je što nudi dodatne komponente naspram uobičajenih komponenti koje nudi Android. Neki od tih komponenti su birač datuma (eng. *Date Picker*), *čipovi (*eng. *Chips*) i birač vremena (eng. *Time Picker*).

Kreiranje tema na temelju *Material Design* smjernica je način prilagodbe dizajna kako bi održali konzistentnost aplikacije i prilagodili stil aplikacije prema našoj marki. Prema smjernicama svaka aplikacija mora sadržavati definirane boje, tipografiju i oblike (*Material Design*, 2022). Izmjena stila tih komponenti automatski će se odraziti na promjenu stila komponenti koje koristimo u aplikaciji a dio su *Material Design* sustava. Tako na primjer izmjenom glavne boje (eng. *Primary color*) definirane unutar *Material Design* teme, promijeniti ćemo sve boje komponenti koje koriste navedeno boju na jedan ili drugi način. Konfiguriranjem boja, tipografije i oblika možemo dobiti kompletan sustav dizajna za svoju marku. Više o *Material Designu* i kako ga koristi u izradi aplikacija objašnjeno je cjelinama koji se odnose na praktični dio ovog rada.



Slika 3: *Material Design* komponente (Izvor: Material Design, 2022)

4.2. *View* sustav

U ovoj cjelini objašnjen je *View* sustav, osnovni gradivni blok za izgradnju korisničkog sučelja za Android koji se koristi više od jednog desetljeća za izgradnju raznih vrsta korisničkog sučelja (Android Developers, 2022c). Sam pogled (eng. *View*) predstavlja pravokutno područje na zaslonu uređaja, a osnovna klasa zadužena za stvaranje interaktivnih pogleda zove se *View*. (Android Developers, 2022b). Postoji mnogo specijaliziranih podklasa *View* objekta koji djeluju kao kontrole ili omogućuju prikaz teksta, slike ili nekog drugog sadržaja.

Podklasa *ViewGroup* osnovna je klasa za izradu rasporeda pogleda, a predstavlja nevidljive spremnike koji objedinjuju elemente korisničkog sučelja ili druge grupe rasporeda pogleda u jednu cjelinu (Griffiths & Griffiths, 2021). *ViewGroup* objekti mogu biti jedan od mnogih tipova rasporeda pogleda koji pružaju određenu strukturu rasporeda svojih podređenih elemenata. Primjer najčešće korištenih rasporeda pogleda su *LinearLayout*, *RelativeLayout* i *ConstraintLayout*. *ViewGroup* objekte možemo shvatiti kao „roditelje“ koji sadrže određeni broj elemenata „djece“, a raspoređuju ih unutar nekog prozora (ili zaslona) u stablo podređenih elemenata. Elemente korisničkog sučelja i raspored pogleda možemo kreirati direktno iz kôda ili definiranjem „stabla pogleda“ u jednoj ili više XML datoteka.

Android nudi nekoliko pred definiranih elemenata korisničkog sučelja. Tako u *View* sustavu na primjer postoji *TextView* za prikaz običnog teksta na ekranu, *Button* za prikaz tipke te *ImageView* za prikaz slike na ekranu. Svakom od tih komponenti može se promijeniti specifični atribut poput pozadinske boje, visine, širine te mnogo drugih atributa.

Kako bi shvatili kako kreirati *View* objekte, potrebno je prvo razumjeti aktivnosti i fragmente koji se zapravo brinu o kreiranju prozora unutar kojeg će nacrtati i postaviti navedene elemente korisničkog sučelja pozicionirane unutar nekog rasporeda pogleda.

4.2.1. Aktivnosti i fragmenti

Svaka nativna aplikacija u *Androidu* uključuje jednu ili više aktivnosti. *Activity* je posebna klasa koja kontrolira ponašanje aplikacije i odlučuje kako će prezentirati korisniku sadržaj (Griffiths & Griffiths, 2021). Drugim riječima, odgovorna je za stvaranje prozora unutar kojeg se prezentiraju komponente korisničkog sučelja. Iako se aktivnosti često prikazuju korisniku kao prozori preko cijelog zaslona, mogu se koristiti i na druge načine: kao plutajući prozori, način rada s više prozora ili ugrađeni u druge prozore (Android Developers, 2022b). Većina aplikacija sadrži više zaslona, što znači da obuhvaćaju više aktivnosti. Obično je jedna aktivnost u aplikaciji navedena kao glavna aktivnost, a predstavlja prvi zaslon koji se pojavljuje kada korisnik pokrene aplikaciju. Svaka aktivnost može pokrenuti drugu aktivnost kako bi

izvršila različite radnje. Za primjer uzmimo aplikaciju pomoću koje možemo pregledavati nadolazeće filmove. Glavna aktivnost aplikacije može sadržavati listu nadolazećih filmova. Pritiskom na neki od filmova, glavna aktivnost poziva drugu aktivnost unutar koje su prikazani detalji filma.

Zadnjih nekoliko godina u svijetu Androida naginje se sve više takozvanoj arhitekturi zasnovanoj na uporabi jedne aktivnosti (eng. *Single Activity Architecture*). Iz samog naziva možemo pretpostaviti o čemu je riječ - uporaba jedne aktivnosti kroz cijelu aplikaciju. U određenim slučajevima potrebno je imati više od jedne aktivnosti zbog određenih tehničkih razloga. Glavni fokus navedene arhitekture nije ograničiti razvojne inženjere na samo jednu aktivnost, već na uporabu što manjeg broja aktivnosti unutar aplikacije (Ghosh, 2021). Budući da smo ranije definirali kako jedna aktivnost uglavnom predstavlja jedan prozor aplikacije unutar kojeg iscrtava UI elemente, postavlja se pitanje kako prikazati ostale prozore. Prateći navedenu arhitekturu, aktivnost unutar aplikacije definiramo kao spremnik kojega „punimo“ s fragmentima.

Fragment predstavlja ponašanje ili dio korisničkog sučelja u aktivnosti (Chugh, 2022a). Možemo kombinirati više fragmenata u jednoj aktivnosti kako bi izgradili korisničko sučelje s više okna. Fragmente često promatramo kao modularna pod-razina aktivnosti koja ima vlastiti životni ciklus i prima vlastite ulazne događaje (Griffiths & Griffiths, 2021). Također, fragmenti ne mogu živjeti sami za sebe - moraju biti domaćini aktivnosti ili nekog drugog fragmenta. Hijerarhija pogleda fragmenta postaje dio hijerarhije pogleda „roditelja“ (aktivnost ili fragment). Mnoge moderne biblioteke specifične za Android, dizajnirane su da rade prvenstvo s fragmentima, a među najpopularnijima spadaju *Navigation* i *BottomNavigationView* o kojima će više biti riječi u poglavljima koji se odnose na praktični dio rada.

Aktivnosti su idealno mjesto za postavljanje globalnih elemenata oko korisničkog sučelja naše aplikacije kao što je navigacija ladica (eng. *Navigation Drawer*). S druge strane, fragmenti su posebno korisni kada želimo uvesti modularnost i mogućnost ponovne upotrebe korisničkog sučelja unutar naše aplikacije. Time su prikladniji za definiranje i upravljanje korisničkim sučeljem jednog zaslona ili samo dijela zaslona.

4.2.1.1. Životni ciklus

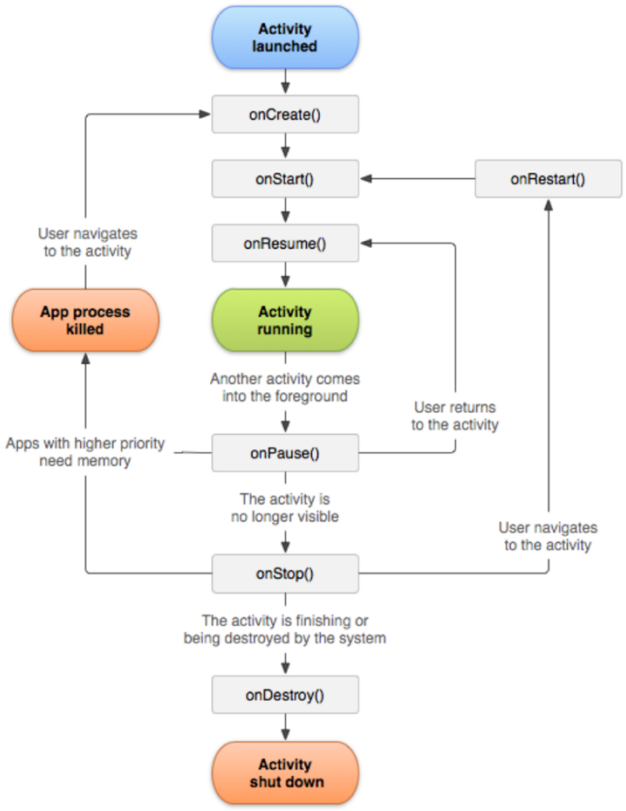
Svaka Android aplikacija ima određen životni ciklus i stanja kroz koje prolazi od trenutka kada korisnik otvori aplikaciju, pa sve do kada izađe iz aplikacije. Za razliku od programskih paradigmi u kojima se aplikacije pokreću metodom *main(),* Android pokreće aplikaciju u instanci *Activity* pozivanjem specifičnih metoda koje odgovaraju određenim fazama njegovog životnog ciklusa (Bogode, 2021).

Klasa *Activity* pruža brojne povratne pozive (eng. *Callback*) koje obavještavaju aktivnost da se stanje promijenilo (stvara, zaustavlja, nastavlja aktivnost ili uništava proces u kojem se aktivnost nalazi) (Griffiths & Griffiths, 2021).

Postoji šest temeljnih povratnih poziva koji omogućuje reagiranje na pojedina stanja unutar jednog životnog ciklusa aktivnosti, a to su: *onCreate(), onStart(), onResume(), onPause(), onStop() i onDestroy() (Android Developers, 2022e).* Sustav poziva svaki od ovih povratnih poziva kada aktivnost ulazi u novo stanje. Slika 4. grafički prikazuje pojednostavljeni prikaz životnog ciklusa aktivnosti.

Povratni poziv *onCreate()* mora se implementirati, a pokreće se kada sustav prvi put kreira aktivnost. U *onCreate()* metodi prikladno je postaviti osnovnu logiku pokretanja aplikacije koja bi se trebala dogoditi samo jednom tijekom cijelog trajanja aktivnosti. Ovdje obično pozivamo *setContentView(int)* metodu kojoj proslijedimo XML resurs koji definira izgled korisničkog sučelja. Unutar XML resursa možemo definirati je jedan ili više rasporeda pogleda unutar kojeg se nalaze međusobno povezani elementi korisničkog sučelja.

Posljednji poziv prije same dealokacije aktivnosti je *onDestroy()* i obično se implementira kako bi osigurali da su svi resursi aktivnosti oslobođeni kada se aktivnost ili proces koji je sadrži „uništi“ (Bogode, 2021).

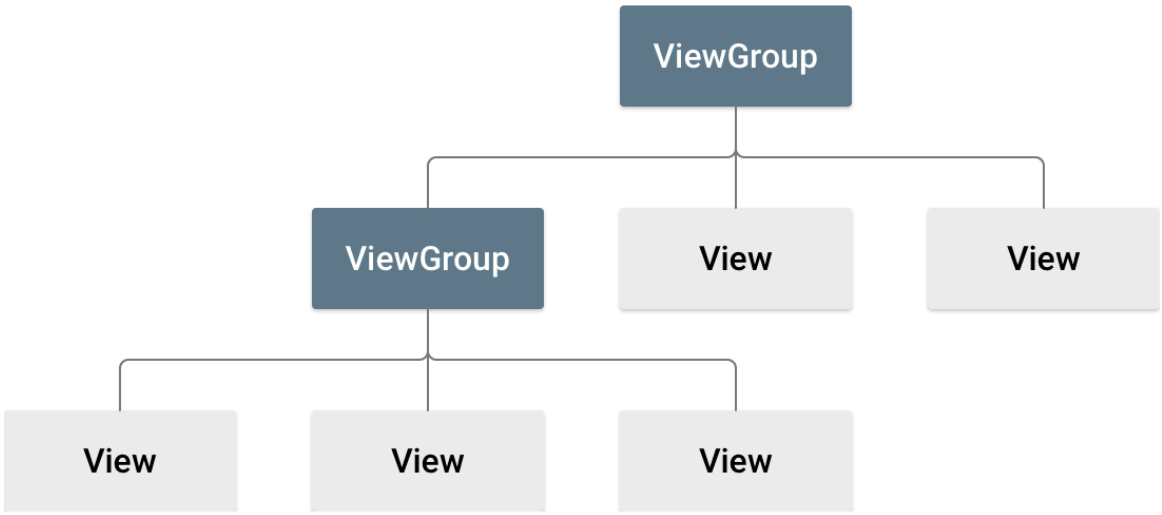


Slika 4: Životni ciklus aktivnosti (Izvor: Android Developers, 2022e)

U ovoj cjelini opisane su osnovne klase koje se brinu o prikazu prozora nad kojim aplikacija iscrtava svoj sadržaj. Kako kreirati sadržaj koji želimo prikazati unutar aktivnosti ili fragmenta objašnjeno je u sljedećoj cjelini pod nazivom XML.

4.2.2. XML

XML definira strukturu korisničkog sučelja u aplikaciji kao što je aktivnost. Svi elementi unutar XML dokumenta izgrađeni su pomoću hijerarhije elementa korisničkog sučelja i rasporeda pogleda (Okolie, 2019). Na slici 5. možemo vidjeti vizualni prikaz hijerarhije rasporeda pogleda i elemenata korisničkog sučelja. Element korisničkog sučelja obično predstavlja nešto što korisnik može vidjeti i s čime može komunicirati, dok je raspored pogleda nevidljivi spremnik koji objedinjuje elemente korisničkog sučelja i druge rasporede pogleda.



Slika 5: Hijerarhija *View* i *ViewGroupa* (Izvor: Android Developers, 2022c.)

Najpopularniji način kreiranja rasporeda pogleda i UI elemenata u *View* sustavu je putem XML-a uz pomoć posebnom XML vokabulara, gdje jedna oznaka odgovara jednoj podklasi elementa korisničkog sučelja ili rasporeda pogleda. Tradicionalni pristup izradi korisničkih sučelja za Android sastoji se od definiranja stabla komponenti i njihova izmjena tijekom izvođenja programa. Iako se to može učiniti potpuno programski, preferirani način je stvaranje korisničkog sučelja putem XML datoteka, a njihovo ažuriranje, odnosno promjena izgleda programskim putem. Time omogućujemo odvajanje prezentacijskog dijela aplikacije od dijela koji kontrolira njezino ponašanje. Također, definiranje strukture prikaza UI elemenata putem XML datoteka olakšava pružanje različitih izgleda za različite veličine i orijentacije zaslona (Android Developers, 2022c).

Kao developeri imamo veliku fleksibilnost upotrebe jedne ili obje metode za izradu korisničkog sučelja aplikacije. Prema navedenome možemo kreirati strukturu izgleda unutar XML-a, a za vrijeme izvršavanja aplikacije dinamički mijenjati svojstva samih elemenata ovisno o trenutnom stanju aplikacije.

Važno je napomenuti da svaka XML datoteka izgleda mora sadržavati točno jedan korijenski element koji mora biti podklasa *View* ili *ViewGroup* objekta (Okolie, 2019). Nakon što definiramo korijenski element, možemo dodati objekte izgleda ili elemente korisničkog sučelja kao podređene elemente kako bi postupno izgradili hijerarhiju prikaza. Pogledajmo sljedeći isječak XML kôda u kojem je prikazan jednostavan primjer definiranja jednog rasporeda pogleda unutar kojeg se nalaze tri elementa korisničkog sučelja.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:gravity="top"
    android:padding="20dp"
    android:orientation="vertical"
    android:layout_height="match_parent">

    <TextView
        android:text="Trending Now"
        android:layout_width="wrap_content"
        android:textAppearance="@style/TextAppearance.AppCompat.Large"
        android:layout_height="wrap_content" />

    <Space
        android:layout_width="0dp"
        android:layout_height="10dp"/>

    <androidx.cardview.widget.CardView
        android:layout_width="match_parent"
        android:layout_height="150dp" />

</LinearLayout>
```

Isječak kôda 1: Prikaz jednostavnog XML dokumenta

U primjeru iznad korijenski element je *LinearLayout*, a omogućuje nam poravnati elemente u jednom smjeru, ovisno o zadanoj orijentaciji (okomito ili vertikalno) koju određujemo putem sljedećeg atributa:

android:orientation="vertical".

Unutar samog izgleda nalaze se tri elementa korisničkog sučelja. Prvi je *TextView*, a omogućuje prikaz jednostavnog teksta. Uz pomoć atributa:

```
android:text="Trending Now"
```

zadajemo tekst koji će se prikazati na zaslonu ekrana kada aktivnost ili fragment iscrta elemente unutar *onCreate()* metode. Svaki *View* i *ViewGroup* podržava vlastiti niz XML atributa. Neki su atributi specifični za određeni element (kao što je atribut *text* za *TextView*), dok su neki zajednički svim *View* objektima, jer su naslijeđeni od korijenske *View* klase kao što su širina i visina te ID elementa (Android Developers, 2022c).

Element poput *Space*-a koristimo kada želimo dodati razmak između elemenata i time stvoriti vizualnu hijerarhiju u našem dizajnu. Osim samog *Space* elementa, razmak možemo dodati uz pomoć margina (eng. *Margin*) i podstave (eng. *Padding*) koji su dostupni kao atributi samih elementa.

Posljednji element korisničkog sučelja iz priloženog primjera ima nešto duži naziv za razliku od ostalih. Dolazi s prefiksom *androidx* što označava skup biblioteka koje će zamijeniti sve originalne Android biblioteke koje se više ne održavaju (Android Developers, 2022b). Radi se o ekstenzijskoj biblioteci koje pružaju dodane mogućnosti nad već postojećim elementima. Osim nadogradnje osnovnih elemenata, *androidx* pruža razne dodatne funkcionalnosti koje omogućuju brži razvoj korisničkog sučelja (Ghita, 2022). Tako navedena biblioteka sadrži komponentu zvanu *CardView* koja se koristi za prikaz bilo koje vrste podataka pružanjem izgleda zaobljenog kuta zajedno s određenom elevacijom (eng. *Elevation*). Tijekom izrade praktičnog rada *CardView* je često korišten kako bi podigli razinu konzistentnosti i ujednačenosti unutar aplikacije.

Općenito govoreći, izgledi su odgovorni uglavnom za dimenzioniranje i pozicioniranje svoje djece. Iako mogu imati vizualni prikaz (npr. boju pozadine ili obrub), obično nemaju interakciju s korisnikom. *ScrollView* je jedan od iznimaka tog pravila, jer omogućuje interakciju s korisnikom putem *swipe* gesta. s druge strane, elementi korisničkog sučelja (popust tipki, potvrdnih okvira i tekstualnih polja) ne samo da omogućuju interakciju korisnika – to je njihova svrha (Künneth, 2022).

Kada smo kreirali i povezali sve potrebne elemente u XML datoteci, možemo ju „ugraditi“ u bilo koju aktivnost ili fragment čiji je postupak opisan u sljedećoj cjelini.

4.2.3. Učitavanje XML resursa

U većini slučajeva, kada kreiramo XML datoteke, kreirano ih za jedan cijeli ekran. Takve datoteke predstavljaju resurse rasporeda pogleda koje možemo prikazati unutar neke aktivnosti ili fragmenta. Najbolji trenutak kada želimo spojiti XML s aktivnosti je unutar *onCreate()* poziva, budući da je to prvi poziv unutar životnog ciklusa aktivnosti. Pozivom *setContentView()* metode možemo učitati određeni resurs i prikazati njegove elemente kada će aktivnost biti kreirana (Künneth, 2022).

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    binding = ActivityMainBinding.inflate(layoutInflater)
    setContentView(binding.root)
```

Isječak kôda 2: Učitavanje XML unutar aktivnosti

U primjeru programskog kôda iznad koristimo se značajkom zvanom „povezivanje prikaza“ (eng. *View binding*). Navedena značajka omogućuje lakše upravljanje i interakciju s izgledima i elementima korisničkog sučelja definiranim u XML-u (Genç, 2022). Jednom kada omogućimo povezivanje prikaza unutar nekog modula, biblioteka generira klasu za svaku XML datoteku prikaza rasporeda unutar tog modula. Svaka generirana klasa sadrži izravne reference na sve izglede i elemente korisničkog sučelja koji imaju definiran ID, kao i na korijenski element za jedan XML dokument prikaza rasporeda. (Android Developers, 2022b).

Postoji drugi, stariji način kako dohvatiti referencu prema određenom elementu korisničkog sučelja, a to je pozivanjem metode *findViewById* kojoj moramo proslijediti ID traženog elementa.

```
findViewById<ImageView>(R.id.tv_show_poster)
```

Isječak kôda 3: Dohvaćanje reference elementa korisničkog sučelja

Svaki element korisničkog sučelja može imati ID koji označava jedinstveni identitet unutar hijerarhije prikaza elemenata. U XML datoteci rasporeda taj ID definiramo kao niz znakova putem *id* atributa (Griffiths & Griffiths, 2021).

Preko *findViewById* metode imamo globalni pristup bilo kojem elementu korisničkog sučelja unutar naše aplikacije, što može dovesti do neočekivanog ponašanja ako dohvaćamo reference elemenata koji ne pripadaju hijerarhiji prikaza korijenskog elementa unutar trenutne aktivnosti ili fragmenta (Android Developers, 2022c).

Značajka „povezivanja prikaza“ omogućuje puno sigurniji pristup referencama elementima korisničkog sučelja i rasporedima pogleda budući da navedena značajka generira

klase koje sadrže reference prema elementima koji se nalaze unutar jednog XML dokumenta. Isto tako prilikom svake promjene XML dokumenta klase se ponovno generiraju s ažuriranim referencama. Da bi omogućili značajku, *viewBinding* opcija mora biti postavljena na *true* u datoteci *build.gradle* na razini modula (Genç, 2022). Način kako postaviti navedenu opciju prikazano je u sljedećem isječku kôda.

```
buildFeatures {
    viewBinding = true
}
```

Isječak kôda 4: Postavljanje značajke povezivanja prikaza

U ovom poglavlju prikazani su načini kako povezati aktivnosti i fragmente s elemenata korisničkog sučelja definiranim u XML-u. U sljedećem poglavlju prikazan je način kako napraviti dinamične promjene nad elementima korisničkog sučelja tijekom izvršavanja aplikacije.

4.2.4. Promjena korisničkog sučelja

U prethodnom poglavlju predstavljena je značajka koja omogućuje pristup referencama elemenata definiranih unutar neke XML datoteke rasporeda pogleda. Budući da imamo direktni pristup *View* objektu, time imamo direktni pristup svih njihovim metodama i atributima.

U isječku kôda pod brojem 5. promijenili smo vidljivost elementa ovisno o određenom stanju *loading* varijable. Pozitivno stanje *loading* varijable odnosi se na stanje u kojem dohvaćamo sadržaj s nekog web servisa. Kada je dohvaćanje sadržaja dovršeno, *loading* stanje prelazi u stanje *false*. Kako bi obavijestili korisnika da trenutno učitavamo sadržaj, možemo prikazati traku napretka (eng. *Progress bar*). Referencu prema cijeloj hijerarhiji rasporeda elementa možemo pohraniti u lokalnu varijablu *binding* putem koje imamo pristup *visibility* atributu trake napretka. Taj atribut odlučuje o tome hoće li element biti vidljiv na ekranu ili ne, te u slučaju kada dohvaćamo sadržaj želimo prikazati traku napretka i sakriti bilo koji drugi sadržaj na ekranu. Kada je sadržaj dohvaćen, pomoću *bindinga* ponovno pristupimo traki napretka i promijenimo atribut *visibility* na *View.GONE,* što će kompletno sakriti prikaz elementa te kao takvi neće više sudjelovati u mjerenju tijekom sljedećeg iscrtavanja.

```
if (loading) {
    binding.progressBar.visibility = View.VISIBLE
    binding.layoutStackView.visibility = View.INVISIBLE
} else {
    binding.layoutStackView.apply {
        alpha = 0f
        visibility = View.VISIBLE
        animate()
            .alpha(1f)
            .setDuration(

resources.getInteger(android.R.integer.config_mediumAnimTime).toLong(
)
            )
            .setListener(null)
    }
    binding.progressBar.visibility = View.GONE
}
```

Isječak kôda 5: Primjer promjene atributa UI elemenata

U primjeru iznad prikazano je kako promijeniti izgled elementa korisničkog sučelja prilikom prvog iscrtavanja elementa na ekranu budući da se kôd iz isječka nalazi unutar *onCreateView()* metode. Postavlja se pitanje kako reagirati na akcije kada, npr. korisnik klikne na određeni gumb ili unese određeni tekst. Pogledajmo u sljedećem isječku kôda kako reagirati na promjene kada korisnik unosi tekst u polje za unos teksta (potrebno je imati na umu da se sljedeći isječak kôda nalazi unutar *onCreateView()* metode tako da se može izvršiti kada je fragment postavio sve svoje elemente):

```
binding.etSearch.doOnTextChanged { text, _, _, _ ->
    text?.let {
        if (it.isNotEmpty()) {
            binding.editTxtSearch.endIconDrawable =
ResourcesCompat.getDrawable(
                resources,
                R.drawable.ic_cancel,
                resources.newTheme()
            )
            viewModel.searchState.value = it.toString()
        } else {
            binding.editTxtSearch.endIconDrawable = null
        }
    }
}
```

Isječak kôda 6: Primjer reagiranje na unos teksta

U isječku kôda iznad pristupamo *EditText* instanci putem *binding* varijable. Element *EditText* posebna je podkasa *View* objekta koja pruža mogućnost unosa teksta preko unaprijed definiranog polja za unos. Kada dohvatimo instancu *EditText* objekta unutar aktivnosti ili fragmenta, imamo pristupiti *lambda* metodi pod nazivom *doOnTextChanged*. Blok funkcije će

se izvršiti svaki puta kada korisnik unese ili izbriše tekst unutar polja za unos teksta. U našem primjeru, kada korisnik počne pisati tekst, želimo prikazati ikonu na čiji klik možemo izbrisati uneseni tekst. *EditText* objekt sadrži atribut koji nam omogućuje postavljanje ikone na sami kraj okvira a zove se *endIconDrawable*. Sve što moramo učiniti kako bi prikazali ikonu je proslijediti ID željenog XML resursa.

Koncept promjene korisničkog sučelja predstavljen u prethodna dva primjera zovemo imperativni, zbog toga što svaku promjenu korisničkog sučelja vršimo izmjenom atributa komponente korisničkog sučelja (Griffiths & Griffiths, 2021). Kao što možemo vidjeti u navedenim primjerima, ovo radi prilično dobro kada je u pitanju manja aplikacija. Što više elemenata korisničkog sučelja imamo, to će biti zahtjevnije pratiti i ažurirati promjene unutar aplikacije.

4.2.5. Često korišteni XML izgledi

U prethodnom poglavlju veliki fokus bio je na elementima korisničkog sučelja. U ovom poglavlju prikazani su i objašnjeni najčešće korišteni XML rasporedi pogleda. Svaki tip pogleda pruža jedinstven način prezentacije ugniježđenih elemenata. Iako je moguće ugnijezditi jedan ili više rasporeda pogleda unutar drugog kako bi postigli određeni dizajn korisničkog sučelja, trebali bi nastojati da hijerarhija tih rasporeda pogleda bude što manja (Android Developers, 2022c). Kada imamo vrlo malo slojeva rasporeda pogleda u pojedinom XML dizajnu, aktivnost (ili fragment) će brže iscrtati i prikazati potrebne elemente te će samim time aplikacija imati bolje performanse (široka hijerarhija prikaza bolja je od duboke hijerarhije pogleda) (Allison, 2022).

Svaki put kada definiramo neki raspored pogleda u XML-u, moramo odraditi minimalno tri stvari (Griffiths & Griffiths, 2021):

- odrediti vrstu rasporeda pogleda
- popuniti izgled određenim elementima korisničkog sučelja ili drugim rasporedima pogleda
- postaviti širinu i visinu samog izgleda

Tri najkorištenija izgleda prema vlastitom iskustvu su: *LinearLayout*, *RelativeLayout* i *ConstraintLayout. LinearLayout* organizira elemente duž jedne linije. Pomoću atributa *android:orientation* možemo postaviti ugniježđene elemente u horizontalni ili vertikalni red (Chugh, 2022b). Isto tako, izgled će stvoriti traku za pomicanje ako duljina prozora premašuje

duljinu zaslona. Na slici 6. možemo vidjeti kako će se elementi unutar *LinearLayouta* rasporediti s orijentacijom postavljenom na vertikalni prikaz.

Slika 6: Ilustracija *Linear Layouta*

RelativeLayout postavlja elemente na temelju njihovih međusobnih odnosa i odnosa s nadređenim izgledom. Korištenjem *RelativeLayouta* možemo postaviti pogled tako da bude pozicioniran lijevo, desno, ispod ili iznad ovisno o susjednim elementima (Android Developers, 2022c). Time, ovaj izgled omogućuje fleksibilno postavljanje podređenih elemenata ovisno o drugim elementima (npr. element X ispod elementa Y) ili u odnosu na nadređenog (npr. poravnan na dno u odnosu na nadređeni izgled). Ako ne postavimo niti jedno ograničenje na podređeni element, izgled će takve elemente pozicionirati u gornji lijevi kut. Na slici 7. možemo vidjeti jedan primjer rasporeda elemenata uz pomoć *RelativeLayouta*.

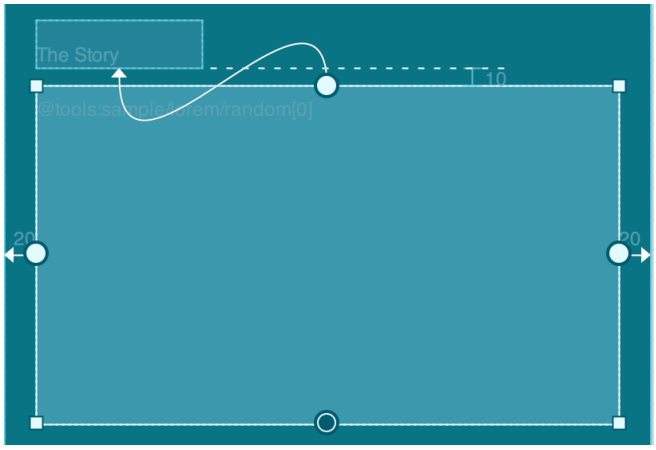
Slika 7: Ilustracija *Relative Layouta*

ConstraintLayout je od svih izgleda najsloženiji te istovremeno najfleksibilniji što se tiče pozicioniranja svojih podređenih elemenata. Omogućuje nam stvaranje kompleksnih prikaza s plosnatom hijerarhijom izgleda (Allison, 2022). To znači da unutar *ConstraintLayouta,* u većini slučajeva nema potrebe za dodatnim ugniježđenim izgledima što čini ovaj izgled vrlo dobrim za performanse. Sličan je *Relative Layoutu* po tome što su svi elementi raspoređeni u skladu s odnosima između susjednih elemenata, ali je fleksibilniji od *RelativeLayouta* i lakši za korištenje kada se koristi zajedno s Android Studio *Layout Editorom* (Android Developers, 2022c). Na slici 8. prikazan je jedan kompleksniji raspored elemenata koji na vrlo lak način možemo postaviti korištenjem *ConstraintLayouta.*

Kako bi element u *ConstraintLayoutu* bio validan, mora imati barem jedno vodoravno i jedno okomito ograničenje. Svako ograničenje (eng. *Constraint*) predstavlja vezu ili poravnanje s drugim pogledom, nadređenim izgledom ili nevidljivom smjernicom (Ilicic, 2017).

Slika 8: Ilustracija *Constraint Layouta*

Primjer postavljenih ograničenja u *Layout Editoru* je prikazan je na slici 9. Iz navedenog primjera možemo primijetiti da svako ograničenje može imati margine kako bi dodali razmak između elemenata ili okvira izgleda. Svako ograničenje počinje s *app:layout_constraint*, a možemo ih definirati nad svim podređenim elementima *ConstraintLayouta*. *(Allison, 2022)*.



Slika 9: Prikaz *Layout Editora* u *Android Studiou*

Lanci (eng. *Chains*) su specifična vrsta ograničenja koja nam omogućuju dijeljenje prostora između elemenata i kontrolu raspoloživog prostora podijeljenog između njih (Ilicic, 2017). Da bi stvorili lanac, moramo odabrati elemente koje želimo povezati zajedno, a zatim odabrati *'Centriraj vodoravno'* (eng. *Center Horizontally*) za stvaranje vodoravnog lanca ili *'Centriraj okomito'* (eng. *Center Vertically*) za stvaranje vertikalnog lanca uz pomoć *Layout Editora*.

Postoje tri moguća načina organizacije lanca: *spread*, *spread_inside* i *packed (R., 2017)*. *Spread* je zadani način, a pozicionira prikaze u lancu tako da su jednako udaljeni unutar dostupnog prostora. *Spread inside* je vrlo sličan *spreadu*, a razlika je u tome što prvo „pričvršćuje“ najudaljenije elemente u lancu na vanjske rubove, a zatim postavlja preostale elemente u lancu tako da su jednako udaljeni unutar dostupnog prostora. Zadnji način organizacije lanca je *packed* koji će „upakirati*“*, odnosno spojiti elemente u jednu grupu, nakon čega će centrirati grupu unutar dostupnog prostora (Allison, 2022).

Ovime smo završili prikaz *View* sustava i dizajn korisničkog sučelja putem XML-a. U sljedećoj cjelini prikazan je i objašnjen deklarativni pristup kreiranju korisničkog sučelja za Android putem *Jetpack Composea*.

4.3. *Jetpack Compose*

Jetpack Compose je *Googleov* najnoviji set alata koji omogućuje kreiranje korisničkog sučelja na vrlo lak i intuitivan način korištenjem samo Kotlina što u potpunosti eliminira potrebu za XML-om (Künneth, 2022). Kako bi bolje razumjeli *Jetpack Compose*, ova cjelina počinje s uvodom u *Composable* metode koje predstavljaju osnovu ovog seta alata. Predstavljen je način kako putem metoda možemo opisati izgled cijelog ekrana i svih njegovih komponenti. Do kraja ove cjeline uspostavit ćemo temeljno razumijevanje o tome što su *Composable* metode i kako se koriste. Pokriti ćemo što je rekompozicija (eng. *Recomposition*) i vidjeti koliko je bitna prilikom prikaza elementa korisničkog sučelje na samom ekranu. U ovom poglavlju također su opisane osnovne *Composable* metode i načini kako ih možemo iskoristiti za kreiranje raznih dizajna.

4.3.1. Ključna načela *Composable* metoda

Jetpack Compose drastično mijenja način pisanja korisničkih sučelja u Androidu. Elemente korisničkog sučelja više nije moguće kreirati putem XML-a, što znači da postoji samo jedan način kreiranja UI elemenata, a to je isključivo programski, putem takozvanih *Composable* metoda.

Composable metodu kreiramo kao bilo koju drugu metodu u Kotlinu, ali s malom razlikom, a to je da ju moramo označiti s *Composable* anotacijom kako bi *Compose* kompajler (eng. *Compiler*) bio obaviješten da pretvori podatke u elemente korisničkog sučelja i prikaže ih na zaslonu ekrana (Arriola, 2022). Primjer takve metode prikazan je u sljedećem isječku kôda.

```
@Composable
fun EmptyListMessage(
    message: String
) { ... }
```

Isječak kôda 7: Primjer *Composable* metode

Postoje određena svojstva koja svaka *Composable* metoda mora poštivati. Tako metoda označena s @*Composable* anotacijom uglavnom vraća *Unit* tip podatka, što možemo interpretirati kao da ne vraća ništa budući da je odgovora smo za prikaz elemenata na zaslonu ekrana. Za parametar mogu prihvatiti bilo kakav tip podatka te isto tako mogu prihvatiti druge *Composable* metode kao jedan od parametra, što ih čini vrlo fleksibilnim i lakim za ponovnu

uporabu na različitim mjestima u aplikaciji. Preporučeno je da svaka metoda poprima paskal (eng. *Pascal*) tip imenovanja (Ghita, 2022). To znači da metoda mora početi s velikim slovom, dok su ostali znakovi napisani malim slovima. Ako se ime sastoji više od jedne riječi, svaka riječ slijedi ovo pravilo.

Budući da Kotlin ima podršku za *top-level* metode (metode definirani direktno unutar datoteke, bez potrebe da se nalaze unutar neke klase) dobra praksa je kreirati *Composable* metode kao *top-level* kako bi bile što fleksibilne za ponovno korištenje.

U starom *View* sustavu kada je bilo potrebno promijeniti prikaz određenog UI elementa, bilo to putem korisnikove interakcije ili putem određene promjene unutar aplikacije, morali smo prvo pronaći željeni element i nakon toga inicirati promjene putem izmjene atributa elementa. Takva promjena odradila se uglavnom pozivanjem metode *findViewById()* ili putem „povezivanja pogleda“ (eng. *View Binding*) koje su zapravo prolazile kroz cijelo stablo elemenata korisničkog sučelja određene hijerarhije prikaza, sve dok ne bi pronašle traženi element (Android Developers, 2022f). Takvo ručno manipuliranje elementima korisničkog sučelja povećava vjerojatnost za pogrešku. Ako se dio podataka prikazuje na više mjesta, lako je zaboraviti ažurirati jedan od prikaza koji ih prikazuje. Isto tako vrlo je lako stvoriti neželjena ponašanja kada istovremeno želimo ažurirati više elemenata odjednom. Općenito, složenost održavanja takvog prikaza korisničkog sučelja raste s brojem elemenata koji zahtijevaju ažuriranje (Richardson, 2020).

S *Jetpack Composeom*, predstavljen je deklarativni pristup kreiranju i ažuriranju korisničkih sučelja. Takav pristup funkcionira tako da se inicira zahtjev za osvježavanje cijelog zaslona od nule, a zatim primjenjuju samo promjene za one elemente kojima je stvarno potrebno promijeniti izgled (Trengrove, 2022). Ovim pristupom izbjegava se složenost ručnog ažuriranja hijerarhije prikaza korisničkih elemenata. Jedan izazov s osvježavanjem cijelog zaslona je taj što je potencijalno skupo što se tiče vremena potrebnog za osvježavanje ekrana i računalne snage. Kako bi *Compose* riješio taj problem, tijekom rekompozicije (eng. *Recomposition*) provjerava koje dijelove korisničkog sučelja treba ponovno iscrtati (Künneth, 2022).

4.3.2. Rekompozicija

Kao što je već navedeno, u imperativnom, odnosno starom načinu, komponentu korisničkog sučelja ažuriramo promjenom njegovog atributa. U *Jetpack Composeu*, promjenu radimo tako da pozovemo istu metodu s različitim podacima, što će uzrokovati ponovno iscrtavanje samo onih komponenti korisničkog sučelja definiranih unutar te metode (Künneth, 2022). Takav način promjene zove se rekompozicija. Prilikom rekompozicije *Compose* će preskočiti sve one *Composable* metode čije ulazne vrijednosti, odnosno parametri nisu promijenjeni (Trengrove, 2022). Preskakanjem svih metoda koje nemaju promijenjene parametre, *Compose* može na vrlo učinkovit način ažurirati korisničko sučelje ekrana. Važno je napomenuti da elemente korisničkog sučelje u *Composeu* možemo promijeniti samo putem rekompozicije (Ghita, 2022). Tako je životni ciklus *Composable* metoda vrlo jednostavan i sastoji se od tri stanja (Android Developers, 2022f):

- ulazak u kompoziciju
- kompozicija
- izlazak iz kompozicije

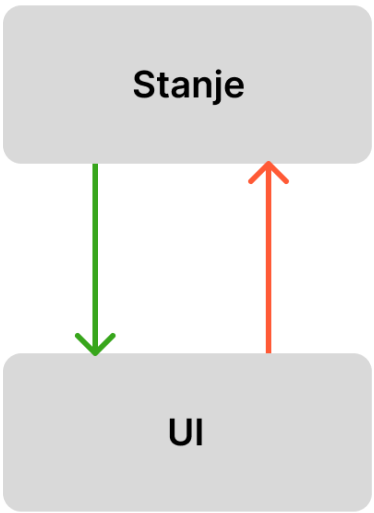
Usporedimo li životni ciklus sa starim *View* sustavom možemo primijetiti da je životni ciklus u *Composeu* značajno jednostavniji.

Rekompoziciju obično pokrećemo promjenom *State<T>* objekta (Guerrero, 2022). *Compose* prati sve takve objekte i poziva sve *Composable* metode u kompoziciji koji ovise o određenom *State<T>* objektu. Kako bi bolje predočili kako ažurirati određenu komponentu korisničkog sučelja priložen je sljedeći isječak kôda u kojem je prikazano kako promijeniti boju i tip ikone na korisnikov zahtjev.

```
IconButton(
    onClick = {
        if (isAdded)
viewModel.onEvent(TvShowDetailEvent.RemoveFromWatchlist)
        else viewModel.onEvent(TvShowDetailEvent.AddedToWatchList)
    }
) {
    Icon(
        tint = if (isAdded) MaterialTheme.colors.primary
        else MaterialTheme.colors.textPrimary,
        painter = painterResource(
            id =
            if (isAdded) R.drawable.ic_circle_filled_add
            else R.drawable.ic_circle_outlined_add
        ),
        modifier = Modifier.size(40.dp),
        contentDescription = "Add to WatchList"
    ) }
```

Isječak kôda 8: Rekompozicija UI elementa

U primjeru isječka kôda pod brojem 8. možemo primijetiti *Composable* metodu *IconButton()*. To je jedna od mnogih standardnih *Composable* metoda koje pruža *Compose* biblioteka, a omogućuje prikaz tipke u obliku ikone. Metoda kao parametar sadrži lambda metodu *onClick* putem koje reagiramo na korisnikov pritisak. U *lambda* bloku metode možemo postaviti razne uvjete putem kojih određujemo što će se dalje dogoditi. U našem primjeru šaljemo određeni „događaj“ *ViewModel* klasi koja je odgovora za rukovanje bilo kakvih UI događaja. Takav princip zove se jednosmjerni protok podataka (eng. *Uniderectional* data *flow*) (Android Developers, 2022f). Prema tom modelu *ViewModel* klasa pruža podatke *Composable* metodi najviše razine. Ta metoda koristi podatke dobivene od *ViewModela* za opisivanje korisničkog sučelja pozivanjem drugih *Composable* metoda i prosljeđuje odgovarajuće podatke kroz hijerarhiju *Composable* elemenata. Kada je korisnik u interakciji s UI elementima, pokreću se događaji kao što je *onClick*. Ti događaji obavještavaju *Composable* metodu putem promjene stanja u aplikaciji. Kada se stanje promijeni, metoda koja prima izmijenjene podatke ponovno će nacrtati elemente korisničkog sučelja (Mendoza, 2021). To znači da nisu samo podaci jednosmjerni nego i događaji, samo na suprotne načine. Zajedno s *ViewModelom*, koji predstavlja naš jedini izvor podataka, smanjili smo broj neočekivan ažuriranja na minimum budući da ažuriranje iniciramo s centralnog mjesta. Na slici 10. prikazan je grafički prikaz jednosmjernog protoka podataka.



Slika 10: Jednosmjerni protok podataka (izvor: Arriola, 2022)

Prateći princip jednosmjernog protoka podataka, *IconButton() Composable* metodi proslijeđena je *isAdded* vrijednost koja predstavlja određeno stanje. Na temelju tog stanja *Icon() Composable* metodi možemo odrediti boju i tip ikone tako što postavilo željenu boju putem *tint* atributa i željeni resurs putem *painter* atributa. Iz isječka kôda možemo primijetiti

koliko je *Compose* API (eng. *Application Programming Interface*) jednostavan i intuitivan za korištenje. Kada bi isto to htjeli napraviti preko starog *View* sustava trebali bi prvo dohvatiti referencu traženog *View* objekta i zatim promijeniti željeni atribut elementa za što bi trebali puno više kôda i vremena. *Compose* API predstavlja vrlo reaktivan pristup ažuriranju elementa gdje su promjene inicirane automatski putem promjene podatka.

U sljedećem poglavlju prikazane su osnovne *Compose* komponente i opisani načini kako ih pravilno koristiti.

4.3.3. Osnovne *Compose* komponente

Kako bi što brže i lakše kreirali korisnička sučelja, *Compose* nudi nekoliko gotovih komponenti koje omogućuju definiranje raznih specijaliziranih izgleda. Uz standardni set komponenti, *Compose* sadrži set *Material Design* komponenta i rasporeda prikaza koji su dostupni kao *Composable* metode (Android Developers, 2022d). U nastavku ovog poglavlja opisani su najkorišteniji *Compose* elementi.

Najosnovniji *Compose* elementi su redovi (eng. *Rows*) i stupci (eng. *Columns*) koji služe kao vrsta *LinearLayout Viewa* u *Jetpack Composeu* (Majnerić, 2022). Svaki od tih komponenti sadrži atribute za osnovno pozicioniranje i raspored podređenih elemenata (npr. *verticalAlignment* i *horizontalArrangement* za redove).

Raspored (eng. *Arrangment*) odnosi se na raspoređivanje elemenata duž glavne osi (u slučaju reda, glavna os je X ili vodoravna, a u slučaju stupca glavna je os Y ili okomita) (Künneth, 2022). Kada želimo postaviti horizontalni raspored imamo pristup sljedećim vrstama rasporeda (Zhukovich, 2021):

- početak (eng. *Start*)
- kraj (eng. *End*)
- centrirano (eng. *Center*)
- jednak raspored (eng. *SpaceEvenly*)
- razmak između (eng. *SpaceBetween*)
- razmak okolo (eng. *SpaceAround*)
- razmak od (eng. *SpaceBy*)

U slučajevima kada želimo postaviti vertikalni raspored, tada imamo pristup sljedećim vrstama rasporeda (Zhukovich, 2021):

- iznad (eng. *Top*)
- ispod (eng. *Bottom*)
- centrirano (eng. *Center*)
- jednak raspored (eng. *SpaceEvenly*)
- razmak između (eng. *SpaceBetween*)
- razmak okolo (eng. *SpaceAround*)
- razmak od (eng. *SpaceBy*)

Uz raspored, moguće je postaviti i poravnanje (eng. *Alignment*), putem kojega možemo elemente centrirati ili poravnati na jedan od vanjskih rubova reda ili stupca.

```
Column(
    verticalArrangement = Arrangement.spacedBy(10.dp),
    horizontalAlignment = Alignment.CenterHorizontally
) {
    Image(
        painterResource(R.drawable.ic_sad_face),
        contentDescription = null,
        modifier = Modifier.requiredSize(70.dp),
        colorFilter =
ColorFilter.tint(MaterialTheme.colors.textPrimary)
    )
    Text(
        text = message,
        style = MaterialTheme.typography.h3,
        color = MaterialTheme.colors.textPrimary,
        textAlign = TextAlign.Center
    )
}
```

Isječak kôda 9: Primjer *Column() Composable* metode

U programskog isječku pod brojem 9. prikazan je primjer stupca koji sadrži dvije *Composable* metode. Kao atribut prima *verticalArrangement* pomoću kojeg možemo definirati vertikalnu poziciju elemenata. U našem primjeru postavili smo vertikalni razmak od 10 točkica po inču (eng. *Dot per Inch*). Osim dodavanja razmaka, mogli smo elemente rasporediti da imaju jednak razmak ili dodatni jedan od ostalih opcija koji utječu na vertikalno pozicioniranje elemenata. Atribut *horizontalAlignment* omogućuje horizontalno pozicioniranje elemenata. Tako su u priloženom isječku kôda elementi horizontalno centrirani putem atributa *Alignment.CenterHorizontally*. Kako bi rasporedili elemente u obliku stupca, *Composable* metoda *Column()* kao zadnji atribut sadrži *lamba* metodu tipa:

```
content: @Composable ColumnScope.() -> Unit
```

kojoj možemo proslijediti druge *Composable* metode. Prikazana *lambda* metoda je tipa *ColumnScope* i specifična je po tome što omogućuje raspoređivanje podređenih *Composable* metoda u obliku stupca. Ako ne definiramo raspored i poravnanje za stupac, svi podređeni elementi biti će raspoređeni okomito na vrhu i poravnati vodoravno prema lijevom rubu.

Sve navedeno za *Column() Composable* vrijedi jednako i za *Row() Composable* metodu, samo što umjesto *ColumnScope* lambda metode sadrži *RowScope* lambdu metodu kao posljednji atribut preko kojega možemo dodati druge *Composable* metode i rasporediti ih u obliku reda. Ako ne definiramo raspored i poravnanje za red, svi podređeni elementi bit će raspoređeni okomito na vrhu i poravnati vodoravno prema lijevom rubu.

```
Row(
    modifier = Modifier.fillMaxWidth(),
    horizontalArrangement = Arrangement.SpaceBetween,
    verticalAlignment = Alignment.Top,
) {
    Text(
        modifier = Modifier.fillMaxWidth(0.80f),
        color = MaterialTheme.colors.textPrimary,
        style = MaterialTheme.typography.h1,
        text = headerModel.title
    )
    IconButton(
        modifier = Modifier.offset(x = ((-3).dp)),
        onClick = { … }
    )
```

Isječak kôda 10: Primjer *Row() Composable* metode

U programskog isječku pod brojem 10. prikazan je primjer *Row() Composable* metode koja sadrži dva elementa koji su maksimalno udaljeni jedan od drugoga i poravnati na gornji rub retka.

Pomoću *Column()* i *Row() Composable* metoda prikazali smo kako rasporediti elemente duž x i y osi. Postavlja se pitanje kako pozicionirati elemente jedan iznad drugog duž z-osi. Tu dolazi u spas *Box() Composable* metoda koja rješava upravo navedeni problem. Za razliku od *Column()* i *Row() Composable metoda*, *Box()* ima samo *contentAligment* atribut koji omogućuje pozicionirane svoje „djece“ (Ghita, 2022). Ako ništa ne proslijedimo *contentAligment* atributu, *Box() Composable* će pozicionirati sve svoje komponente na gornji desni rub. U isječku kôda pod brojem 11. centrirali smo podređene elemente na donji rub nadređene *Composable* metode. Kada bi usporedili *Box()* s *View* sustavom tada bi *Relative Layout* bio najbliži ekvivalent prema načinu rasporeda svojih podređenih elemenata.

```
Box(
    modifier = Modifier
        .fillMaxSize()
        .padding(15.dp),
    contentAlignment = Alignment.BottomCenter
) { … }
```

Isječak kôda 11: Primjer *Box() Composable* metode

Column(), Row() i *Box() Composable* metode predstavljaju osnovni način kako pozicionirati elemente korisničkog sučelja. Kombinirajući navedene metode možemo na vrlo lak način stvoriti kompleksna korisnička sučelja. Ipak, svaki od tih *Composable* metoda nema previše koristi ako im ne proslijedimo *Composable* metode koje predstavljaju elemente korisničkog sučelja. U nastavku su prikazane najkorištenije *Composable* komponente koje predstavljaju pojedina korisnička sučelja.

Često u aplikacijama želimo prikazati običan tekst. *Compose* nudi jednostavan način prikaza teksta na ekranu, a postižemo ga pozivanjem *Text() Composable* metode. U isječku kôda pod brojem 12. prikazan je primjer navedene metode. Iz isječka možemo primijetiti da je moguće postaviti razna svojstva teksta poput boje, stila te samog sadržaja koji će se prikazati. Postoji još mnogo drugih atributa koje je moguće postaviti poput maksimalnog broja linija, veličine fonta, a sve što moramo učiti je unutar konstruktora metode prosjedili ispravni tip podatka kako bi se željena konfiguracija *Composable* elementa primijenila prilikom rekompozicije.

```
Text(
    modifier = Modifier.fillMaxWidth(0.80f),
    color = MaterialTheme.colors.textPrimary,
    style = MaterialTheme.typography.h1,
    text = headerModel.title
)
```

Isječak kôda 12: Primjer kreiranja teksta u *Composeu*

Kada želimo prikazati sliku pomoću *Composea*, sve što moramo napraviti je pozvati *Image() Composable* metodu kojoj proslijedimo određeni tip slike (Ghita, 2022). Slika koju proslijedimo može biti jedna od sljedeća tri tipa (Künneth, 2022):

- Resurs tipa *Drawable*
- bit mapa (eng. *Bitmap*)
- resurs tipa *imageVector*

```
Image(
    painterResource(
    id = R.drawable.ic_circle_filled_add),
    contentDescription = "Add button icon"
)
```

Isječak kôda 13: Primjer kreiranja slike u *Jetpack Composeu*

U isječku kôda pod brojem 13. proslijedili smo *Image() Composable* metodi *Drawable* resurs putem *painterResourse()* metode koja prima za parametar ID nekog lokalnog resursa unutar našeg projekta. Uz sliku, *Composable* metodi *Image()* možemo proslijediti opis slike preko *contentDescription* atributa kako bi povećali pristupačnost naše aplikacije ljudima s poteškoćama u vidu tako što će zvučnom porukom korisniku objasniti što predstavlja prikazana slika (Android Developers, 2022a). Naravno, preduvjet takvog ponašanja je da korisnik ima na svojem mobilnom uređaju uključen jedan od dostupnih servisa pristupačnosti poput *TalkBacka*.

Budući da su tipke neizbježni element korisničkog sučelja gotovo svake aplikacije, *Compose* sadrži dva tipa *Composable* metoda koje omogućuju prikaz tipke. Prva je *Button() Composable* metoda preko koje možemo prikazati tipku s željenim sadržajem. Jedan od parametra *Button() Composable* metode je *content()* lambda metoda sa sljedećom strukturom:

```
content: @Composable RowScope.() -> Unit.
```

Kao što možemo primijetiti, ima istu strukturu kao *Row() Composable* metoda koju smo prethodno objasnili. Time možemo zaključiti da će *Button()* metoda rasporediti svoje elemente horizontalno, jedan pokraj drugoga. U isječku kôda 14., preko *Button()* metode kreirali smo tipku koja sadrži ikonu i tekst, pozicionirane jednu kraj druge. Budući da *Button() Composable* metoda kao parametar sadrži *content()* metodu tipa *RowScope(),* imamo mogućnost definirati poravnanje i poziciju elementa kao što bi to učinili unutar *Row() Composable* metode.

```
Button(onClick = { … }) {
    Image(
        painterResource(
        id = R.drawable.ic_circle_filled_add),
        contentDescription = "Add button icon"
    )
    Text(text = "Add to Watchlist")
}
…

IconButton(
    modifier = Modifier.size(40.dp),
    onClick = { … }
) {
```

```
        Icon(imageVector = Icons.Default.Add)
    }
```

Isječak kôda 14: Primjer kreiranja različitih tipki u *Jetpack Composeu*

U slučajevima kada želimo prikazati samo ikonu, a da istovremeno imamo pristup istim funkcionalnostima koje ima tipka, postoji *IconButton() Composable* metoda koja kreirana upravo za takav scenarij. U isječku kôda pod brojem 14. možemo primijetiti *IconButton() Composable* metodu koja sadrži podređeni element *Icon()* putem kojeg možemo prikazati ikonu kroz *imageVector* atribut.

Ovime smo pokrili osnovne *Compose* elemente čijom kombinacijom možemo izraditi vrlo kompleksna korisnička sučelja. Istražili smo načine postavljanja željenog rasporeda elemenata putem *Row(), Column()* i *Box() Composable* metoda i koje sve mogućnosti poravnanja i pozicioniranja podređenih elemenata posjeduju. Isto tako prikazani su načini kako putem *Jetpack Composea* kreirati najčešće korištene elemente kao što su tipka, slika i tekst.

Ono što nije pokriveno u ovom poglavlju je način kako modificirati *Compose* elemente, odnosno kako na primjer postaviti veličinu slike ili obojati pozadinu tipke. U prikazanim isječcima kôda možemo primijetiti da je često korišten *Modifier* atribut. Kako putem tog atributa možemo promijeniti stil i ponašanje elementa te zašto je jedan od najvažnijih atributa bilo koje *Composable* metode objašnjeno je u sljedećem poglavlju.

4.3.4. Manipuliranje izgledom *Compose* komponenta

Sve *Composable* metode sadrže atribut *Modifier*, a omogućuje nam postavili određeni stil elementa ili rasporeda prikaza, poput veličine prikaza, podstave, boje pozadine, i mnogo drugih stvari (Castillo, 2022). *Modifier* također omogućuje *Composable* metodama proširiti osnovni skup funkcionalnosti dodavanjem obrade korisničkih zahtjeva putem *lambda* metoda. Tako elemente možemo učiniti „klikabilnim“ jednostavnim dodavanjem *clickable lambda* bloka putem *Modifier* atributa (Künneth, 2022). Način kako to možemo postići prikazan je u isječku kôda pod brojem 15.

Među najviše korištene modifikatore *Modifier* atributa spadaju *.fillMaxSize(), .fillMaxWidth()* & *.fillMaxHeight()*. Sva tri modifikatora sadrže zadanu vrijednost od 1.0, koja označava da će 100% veličine / širine / visine našeg zaslona biti ispunjeno elementom koji sadrži navedeni modifikator (Majnerić, 2022). Ako dodijelimo neku drugu vrijednost koja je manja od 1.0, poput 0.8, element nad kojim je postavljen modifikator zauzeti će 80% dostupnog prostora. Tijekom izrade praktičnog rada, navedeni modifikatori, koristili su se u gotovo 90% slučajeva.

```
Box(
    modifier = modifier
        .fillMaxWidth()
           .clip(MaterialTheme.shapes.small)
        .background(SemiDarkGray)
        .padding(vertical = 4.dp, horizontal = 8.dp),
        .clickable { ... }
)
```

Isječak kôda 15: Primjer upotrebe *Modifier* atributa

Iz isječka kôda pod brojem 15. primjećujemo da modifikatore možemo ulančati. *Composable* metodi *Box()* dodali smo ukupno pet modifikatora. Prvo smo odredili da će element zauzeti cijelu širinu roditelja putem *.fillmaxwidth()* modifikatora. Zatim smo preko *.clip(), .background() i .padding()* modifikatora nad elementom dodali blago zaobljene rubove, obojali pozadinu u tamno sivu i odredili horizontalnu i vertikalnu podstavu. Na kraju smo kroz *.clickable lambda* metodu omogućili da element bude klikabilan što nam omogućuje reagiranje na korisnikovu interakciju s elementom.

Ulančavanjem samo nekoliko modifikatora običnu *Box() Composable* metodu u potpunosti smo izmijenili. Ovime smo prikazali koliko je *Modifier* zapravo moćan i jednostavan za korištenje. Važno je primijetiti da se modifikatori nad elementom primjenjuju sekvencijalno od prvoga prema posljednjem (Android Developers, 2022f). Promjenom rasporeda pozivanja modifikatora u određenim situacijama možemo dobiti različita ponašanja, te je stoga vrlo bitno pripaziti na sami redoslijed modifikatora. *Compose* API sadrži jako veliki broj modifikatora, te je u ovom radu prikazan samo osnovni koncept njihovog korištenja. Cijela lista nalazi se na sljedećem linku: https://developer.android.com/jetpack/compose/modifiers-list.

Ovim poglavljem predstavljen je osnovni princip *Jetpack Composea*. Prikazani su temeljni koncepti *Composable* metoda i načina kako ih kreirati. Velikim dijelom usredotočili smo se na to kako ažurirati korisničko sučelje kroz rekompoziciju i objasnili kako putem jednosmjernog protoka podataka smanjiti broj nepredvidljivih promjena UI elemenata. Putem raznih primjera predstavljeni su osnovni *Compose* elementi i načini kako promijeniti njihov izgled putem *Modifier* atributa. U nastavku ovog rada slijedi praktični dio rada u kojem su prikazane dvije verzije iste aplikacije, jedna kreirana putem starog *View* sustava i druga kreirana putem *Jetpack Composea*. Putem primjera izrađena je detaljna analiza ovih dviju tehnologija, a na kraju je izneseno mišljene o samom *Compose* alatu te koje sve novosti i promjene je pridonio Android sustavu u odnosu na stari *View* sustav.

5. Praktični rad

Ovom cjelinom žele se prikazati mogućnosti *Jetpack Composea* i usporediti s XML-om putem praktičnog primjera. Kako bi mogli usporediti ove dvije tehnologije, izrađene su dvije verzije aplikacije. U prvoj verziji, aplikacija je u potpunosti kreirana putem starog *View* sustava. Svaki ekran i komponenta korisničkog sučelja kreirana je u XML-u i prikazana na ekranu preko aktivnosti ili fragmenta. Druga verzija aplikacije implementirana je putem *Jetpack Composea*. Svaki element korisničkog sučelja predstavljen je putem *Composable* metoda i prikazan na ekranu putem rekompozicije.

Kroz ovu cjelinu detaljno su uspoređeni načini kreiranja najčešće korištenih komponenti u *Jetpack Composeu* i XML-u. Tako je prvo uspoređen način kreiranja i popunjavanja liste elementa. Objašnjen je postupak kako u *View* sustavu i *Jetpack Composeu* izraditi listu koja može primiti različite tipove podataka i prema tim tipovima prikazati ispravne UI elemente. Zatim je uspoređen način kreiranja donje navigacijske trake i objašnjen sustav navigacije iz jednog ekrana na drugi. Nakon toga predstavljen je način kako izraditi elemente za ponovno korištenje na više mjesta u aplikaciji i istodobno objašnjeno zašto je *Jetpack Compose* u tom segmentu znatno bolja opcija u odnosu na XML. Na kraju ovog poglavlja uspoređen je način kreiranja teme između XML-a i *Jetpack Composea*.

Kako bi imali bolju predodžbu što želimo uopće usporediti kroz praktičan dio, u sljedećem poglavlju objašnjena je domena i namjena aplikacije te ukratko opisane same funkcionalnosti aplikacije.

5.1. Opis funkcionalnosti aplikacije

Mobilna aplikacija kreirana za potrebe usporedbe XML-a i *Jetpack Composea* zove se *Sofa Time*. Glavna namjena aplikacije je omogućiti korisniku praćenje njemu najdražih tv serija ili serija koje trenutno prati dodavajući ih u osobnu listu praćenih serija. Putem sustava za pretraživanje korisnik može na vrlo brz i lak način pronaći traženu tv seriju, te istodobno može pregledati popis najpopularnijih i trenutno aktualnih tv serija. U bilo kojem trenutno korisnik ima opciju izbrisati tv seriju iz liste praćenih serija i kao takva se neće više prikazati u njegovom popisu.

5.2. Arhitektura *Sofa Time* aplikacije

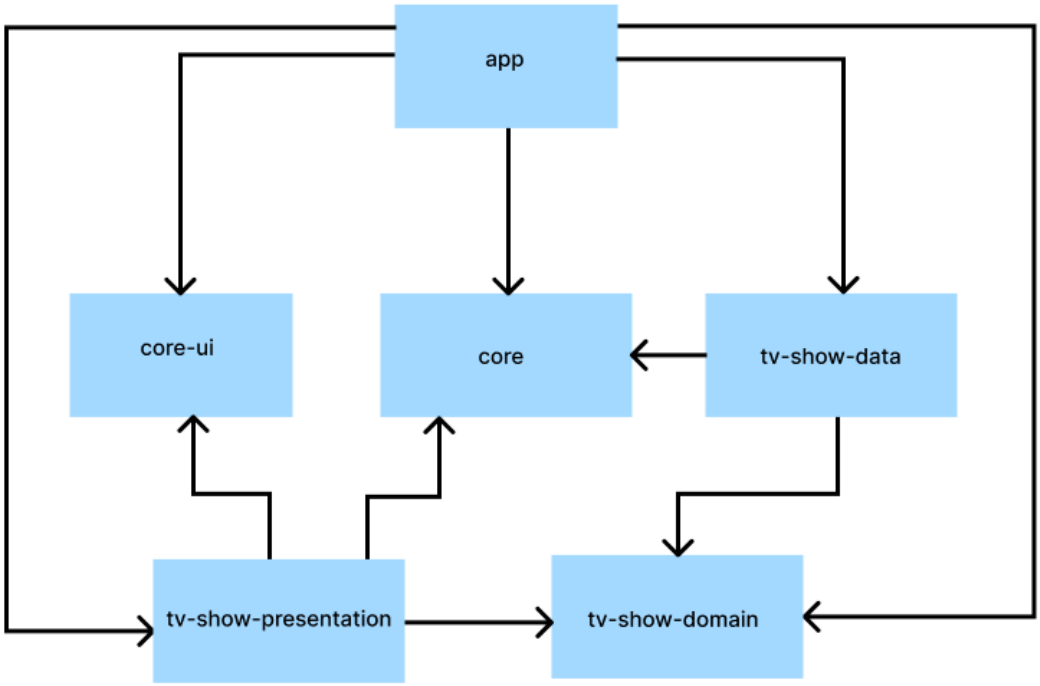
Kako bi što brže izradili dvije aplikacije koje imaju identični set funkcionalnosti, a razlikuju se samo po implementaciji korisničkog sučelja, odnosno prezentacijskog sloja, cijeli projekt je razvijen modularno. Prema tom principu, projekt je razdijeljen u module. Svaki od modula ima točno određenu zadaću i sadrži određeni skup funkcionalnosti. Cilj modularizacije ovog projekta bio je kreirati što veći broj modula koji se mogu dijeliti između aplikacija. Istodobno, modularizacijom možemo kompleksni sustav podijeliti na manje i lakše podsustave, smanjujući time složenost projektiranja i održavanja velikog sustava. U projektu možemo imati jedan ili više modula, a jedan modul može koristiti drugi modul tako što postavimo ovisnost na njega.

Uz pomoć *Android Studioa* možemo kreirati različite tipove modula. U ovom radu fokusirati ćemo se na sljedeća tri tipa modula:

- Android aplikacija (eng. *Android Application*)
- Android biblioteka (eng. *Android Library*)
- Java/Kotlin biblioteka (eng. *Java/Kotlin Library*).

Android aplikacija predstavlja naš *app* modul i sadrži sve potrebne module i biblioteke za normalno izvršavanje aplikacije. U slučaju kada želimo kreirati modul koji posjeduje ovisnosti prema android bibliotekama tada moramo odabrati Android biblioteku kao ciljani modul. Unutar takvog modula često definiramo prezentacijski sloj aplikacije, poput UI komponenata i ekrana koje želimo prikazati.

Java/Kotlin biblioteka namijenjena je za slučajeve kada želimo kreirati modul koji će sadržavati „čisti“ kôd u Kotlinu ili Javi. Takve module često koristimo za definiranje domenskog sloja aplikacije koji sadrži domenske klase i modele te ekstenzijske metode (eng. *Extension Functions*) koje često koristimo unutar projekta.



Slika 11: Arhitektura *Sofa Time* aplikacije

Na slici 11. prikazana je arhitektura *Sofa Time* aplikacije kreirane uz pomoć *Jetpack Composea*. XML verzija aplikacije prati istu strukturu uz izuzetak da je *tv-show-presentation* modul dio samog *app* modula. Iz svakog modula izlaze i/ili ulaze strelice. One predstavljaju ovisnost jednog modula u odnosu na drugi. Tako možemo primijetiti da *app* modul ima ovisnost na sve ostale module. Razlog tome je što *app* modul predstavlja glavni modul aplikacije. Unutar njega se nalazi glavna i jedina aktivnost aplikacije unutar koje se prikazuju svi ostali ekrani. Kako bi imali bolju predodžbu pojedinog modula, u nastavku je kreirana tablica pomoću koje možemo saznati tip modula i kratki opis njegove namjene.

Iz tablice 1. možemo saznati da su *core* i *tv-show-domain* tipa Java/Kotlin biblioteke. *Core* modul sadrži osnovne poslovne klase, servise i ekstenzijske metode za često korištene tipove podataka. Budući da se u aplikaciji na većini ekrana prikazuje određeni format slike, u *core* modulu se nalazi klasa koja definira sve moguće veličine slika i servis odgovoran za kreiranje ispravne putanje i formata slike. Modul *tv-show-domain* sadrži domenske modele i klase potrebne za prikaz entiteta koje predstavljaju tv serije. Moduli *core* i *tv-show-domain* su dijeljeni moduli između aplikacija.

Modul *tv-show-data* također je dijeljeni modul između aplikacija, a sadrži podatkovni sloj aplikacije. Unutar tog modula objedinjena je cijela logika vezana za dohvat podataka s udaljenog servisa i pohrana podataka u lokalnu bazu.

Modul *app* se između *Jetpack Compose* i XML verzije aplikacije razlikuje u tome što je u XML verziji uz navigaciju i glavne aktivnosti dodana cijela prezentacijska logika aplikacije.

Razlog tome je korištenje *Jetpack Navigation Component* biblioteke koja je dizajnirana više za monolitnu arhitekturu aplikacije zbog čega su svi XML resursi dodani direktno u *app* modul projekta. U *Jetpack Composeu* svaka se *Composable* metoda može na vrlo lak način pozvati iz bilo kojeg modula (ovdje se misli na bilo koji modul tipa Android biblioteka) unutar projekta pa je zbog toga moguće imati poseban modul koji sadrži sve *Composable* metode koje služe za prikaz ekrana i elemenata korisničkog sučelja vezanih za funkcionalnost prikaza tv serija.

Preostali modul koji još nije opisan je *core-ui*, a sadrži često korištene UI elemente poput prikaza kada je lista tv serija prazna ili prikaza u slučaju kada se dogodi pogreška prilikom dohvata podataka s udaljenog servisa.

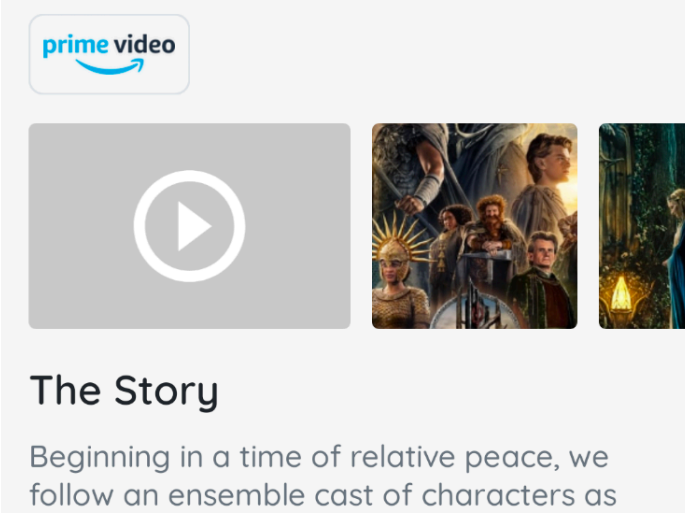
Tablica 1: Opis modula *Sofa Time* aplikacije

Modul	Tip modula	Opis
app (*Jetpack Compose*)	Android aplikacija	Sadrži glavnu aktivnost aplikacije i navigaciju
app (XML)	Android aplikacija	Sadrži glavnu aktivnost aplikacije, navigaciju i cijelu prezentacijsku logiku aplikacije
core	Java/Kotlin biblioteka	Sadrži osnovne poslovne modeli i klase, ekstenzije klasa i pomoćne metode
core-ui	Android biblioteka	Sadrži temu aplikacije, često korištene UI elemente
tv-show-data	Android biblioteka	Podatkovni sloj aplikacije, sadrži mrežne pozive prema vanjskom servisu i lokalno pohranu podatka
tv-show-domain	Java/Kotlin biblioteka	Sadrži domenske modele i klase za prikaz tv-serija
tv-show-presentation	Android biblioteka	Prezentacijski sloj aplikacije, sadrži UI elemente i prikaze ekrana

Nakon opisane namjene i funkcionalnosti *Sofa Time* aplikacije te prikazane arhitekture u nastavku slijedi usporedba različitih UI komponenata aplikacije. Prvo ćemo prikazati i usporediti način kreiranja liste sa složenim tipovima podataka u XML-u i *Jetpack Composeu.* Zatim slijedi usporedba način kreiranja donje navigacijske trake i općenito same navigacije između ekrana. Nakon usporedbe implementacija navigacijskih komponenti slijedi usporedba mogućnosti ponovne primjene UI elemenata između tehnologija. Ovdje ćemo vidjeti koliko je fleksibilan XML i *Jetpack Compose* oko kreiranja elemenata koje možemo na lak način uključiti u određeni dizajn. Na kraju je prikazana usporedba kreiranja teme aplikacije.

5.3. Usporedba prikaza liste elemenata

Gotovo svaka aplikacija sadrži funkcionalnost prikaz određene liste sadržaja. Često je taj sadržaj složen i dinamičan. U ovom poglavlju želimo usporediti kreiranje jedne takve liste koja ima mogućnost prikazati različite tipove podataka. Na slici 12. prikazana je lista koju želimo prikazati. Iz slike možemo primijetiti da se radi o horizontalnom prikazu elemenata s time da se na prvoj poziciji liste nalazi element koji će korisnika odvesti na web mjesto s prikazom najave tv serije u obliku videa. Ostali elementi predstavljaju prikaze postera tv serije.



Slika 12: Prikaz horizontalne liste elemenata

5.3.1. Implementacija liste u XML-u

Prvo ćemo prikazati implementaciju u XML-u. Kreiranje liste elemenata u XML sastoji se od tri koraka:

- Dodati *RecyclerView* putem XML-a
- kreirati adaptere (eng. *Adapters*)
- učitati podatke u listu unutar aktivnosti ili fragmenta

Najčešći način kreiranja liste elemenata u XML-u je putem *RecyclerView* komponente koja nam omogućuje učinkovito prikazivanje velikih lista podataka tako što putem adaptera proslijedimo podatke koje želimo prikazati. U isječku kôda pod brojem 16. prikazan je primjer kako dodati *RecyclerView* unutar *ConstraintLayout* rasporeda pogleda.

```
<androidx.constraintlayout.widget.ConstraintLayout
    android:id="@+id/layoutStackView"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginBottom="40dp">
...

      <androidx.recyclerview.widget.RecyclerView
          android:id="@+id/recyclerViewPosterSection"
          android:layout_width="0dp"
          android:layout_height="wrap_content"
          android:layout_marginStart="20dp"
          android:layout_marginTop="16dp"
          app:layout_constraintEnd_toEndOf="parent"
          app:layout_constraintStart_toStartOf="parent"
          app:layout_constraintTop_toBottomOf="@id/recyclerViewNetwork" />

...

</androidx.constraintlayout.widget.ConstraintLayout>
```

Isječak kôda 16: Primjer dodavanja *RecyclerViewa* u XML-u

Iz prikazanog primjera primjećujemo da *RecyclerView* elementu možemo dodati margine i definirati visinu i širinu prostora koji će zauzeti. Postavljajući širinu na nula, *RecyclerView* će se proširiti kroz cijelu širinu roditelja.

Nakon što smo dodali *RecyclerView* i pozicionirali ga unutar rasporeda pogleda, sljedeći korak je definirati kako će izgledati pojedini element unutar liste. Takav element kreiramo kao i svaki drugi XML element. Primjer kreiranja prikaza postera nalazi se u isječku kôda pod brojem 17.

```
      <com.google.android.material.imageview.ShapeableImageView
      xmlns:android="http://schemas.android.com/apk/res/android"
          xmlns:app="http://schemas.android.com/apk/res-auto"
          xmlns:tools="http://schemas.android.com/tools"
          android:id="@+id/imageViewTvShowPoster"
          android:layout_width="115dp"
          android:layout_height="115dp"
          android:scaleType="centerCrop"
          app:layout_constraintBottom_toBottomOf="parent"
          app:layout_constraintEnd_toEndOf="parent"
          app:layout_constraintStart_toStartOf="parent"
          app:layout_constraintTop_toTopOf="parent"
          app:shapeAppearanceOverlay="@style/roundedCornerShapeSmall"
          tools:src="@tools:sample/avatars">

      </com.google.android.material.imageview.ShapeableImageView>
```

Isječak kôda 17: XML – Primjer izrade *poster* komponente

Kada smo kreirali sve potrebne prikaze u XML-u, sljedeći korak je kreirati adaptera koji će nam omogućiti povezati podatke s XML resursima. Za povezivanje podataka s *RecylerViewom* postoji posebna adapter klasa zvana *RecylerView.Adapter.* Tipično se kreira vlastita klasa koja će implementirati potrebne metode *RecylerView.Adapter* klase.

Za svaki tip elementa koji želimo prikazati u listi moramo definirati vlastiti objekt vlasnika pogleda (eng. *ViewHolder*). Prilikom kreiranja vlasnika pogleda ne popunjavamo ga još s potrebnim podacima. U isječku koda pod brojem 18. možemo vidjeti da smo unutar *onCreateViewHolder()* metode kreirali vlasnik pogleda za svaki tip podatka.

```
override fun onCreateViewHolder(
    parent: ViewGroup,
    viewType: Int
): RecyclerView.ViewHolder {
    if (viewType == TYPE_TRAILER && posterSectionState.trailer != null) {
        val binding = ItemTrailerPlaceholderBinding.inflate(
            LayoutInflater.from(parent.context),
            parent,
            false
        )
        return TrailerViewHolder(binding)
    } else {
        val binding =
            ItemTvShowPosterBinding.inflate(
                LayoutInflater.from(parent.context),
                parent,
                false
            )
        return PostersViewHolder(binding)
    }
}
```

Isječak kôda 18: Primjer implementacije *onCreateViewHolder()* metode

Nakon što kreiramo vlasnika pogleda, putem *RecyclerView*-a ga možemo povezuje sa potrebnim podacima. To radimo unutar *onBindViewHolder()* metode. Budući da želimo prikazati dva različita tipa elementa moramo povezati podatke sa svakim tipom vlasnika pogleda. U isječku kôda pod brojem 19. povezali smo *trailer* tip podatka s *TrailerViewHolderom* te listu postera s *PostersViewHolder* vlasnikom pogleda. Svaki od vlasnika pogleda sadrži metodu *bind()* koja je odgovorna za popunjavanje određenog *View* objekta s pripadajućim podacima.

```
override fun onBindViewHolder(
    holder: RecyclerView.ViewHolder,
    position: Int
) {
    if (holder is TrailerViewHolder) {
        holder.bind(posterSectionState.trailer)
    } else if (holder is PostersViewHolder) {
        holder.bind(posterSectionState.posters[position])
    }
}
```

Isječak kôda 19: Primjer implementacije *onBindViewHolder()* metode

Kako bi *RecyclerView* znao koliko elemenata treba prikazati, potrebno je implementirati *getItemCount()* metodu kojoj moramo proslijediti broj elemenata u listi. Isječak kôda ispod prikazuje primjer implementacije *getItemCount()* metode.

```
override fun getItemCount(): Int {
    return posterSectionState.posters.size
}
```

Isječak kôda 20: Primjer implementacije *getItemCount()* metode

Nakon implementacije adaptera, sve što trebamo napraviti kako bi prikazali listu elemenata je unutar aktivnosti ili fragmenta spojiti *RecyclerView* objekt s adapterom.

```
private suspend fun handlePosterSectionDisplay() {
    viewModel.posterSectionState.collect { posterSection ->
        val postersAdapter = PostersAdapter(posterSection) { url ->
openTrailerInWebView(url) }
        binding.recyclerViewPosterSection.apply {
            adapter = postersAdapter
            layoutManager = LinearLayoutManager(
                requireContext(),
                LinearLayoutManager.HORIZONTAL,
                false
            )
            setHasFixedSize(true)
        }
    }
}
```

Isječak kôda 21: Primjer prikaza liste putem *RecyclerViewa* u fragmentu

Kako bi popunili listu s podacima u fragmentu, prvo moramo dohvatiti referencu prema *RecyclerView* objektu. Nakon toga, sve što moramo učiniti je adapter atributu *RecyclerView* objekta proslijediti adapter koji smo u prethodnom primjeru kreirali. Kako bi odredili raspored prikaza liste i orijentaciju, moramo kreirati upravitelja izgleda (eng. *Layout Manager*). Postoje različiti tipova upravitelja izgleda koji omogućuju različite rasporede poput prikaza u obliku

polja (eng. *Grid*). U našem slučaju želimo prikazati elemente linearno jedan ispod drugog. Takav prikaz možemo postići putem *LinearLayoutManager()* klase kojoj moramo odrediti orijentaciju koja je u našem slučaju horizontalna. Kako bi optimizirali performanse liste možemo dodati *setHasFixedSize(true)* u slučaju kada znamo da se prikazana lista neće mijenjati, odnosno ako se pozicija liste neće mijenjati nakon početnog učitavanja podataka.

Ovime smo završili prikaz implementacije liste u XML-u. U sljedećoj cjelini prikazan je način implementacije liste koja prikazuje različite vrste elemenata u *Jetpack Composeu*.

5.3.2. Implementacija liste u *Jetpack Composeu*

U prethodnom primjeru možemo primijetiti da se implementacija liste u XML-u sastojala od nekoliko bitnih koraka. Morali smo kreirati dizajn u XML-u, zatim kreirati adaptera te nakon toga *RecycerView* i adapter povezati unutar fragmenta. U *Jetpack Composeu* implementacija je puno jednostavnija. Za razliku od XML-a, u *Jetpack Composeu* postoje posebne *Composable* metode za horizontalni i vertikalni prikaz. Kada želimo prikazati listu horizontalno tada koristimo *LazyRow(),* a u suprotnom, za vertikalni prikaz koristimo *LazyColumn().* U isječku kôda ispod prikazan je primjer implementacije horizontalne liste putem *LazyRow() Composable* metode.

```
@Composable
fun VideoAndImageSection(
    modifier: Modifier = Modifier,
    imageLoader: ImageLoader,
    video: Video?,
    posters: List<Poster>,
    onPlayVideoClick: (String) -> Unit
) {
    LazyRow(
        modifier = modifier.fillMaxWidth(),
        horizontalArrangement = Arrangement.spacedBy(12.dp)
    ) {
        video?.let { video ->
            item {
                VideoItem(video, onPlayVideoClick)
            }
        }
        items(posters) { poster ->
            ImageItem(
                imageLoader = imageLoader,
                poster = poster,
            )
        }
    }
}
```

Isječak kôda 22: Primjer prikaza liste putem *LazyRow()* metode

U teorijskom dijelu *Jetpack Composea* objasnili smo *Row() Composable* element i sve njegove mogućnosti. *LazyRow()* je nadogradnja postojeće *Composable* metode i zapravo predstavlja ekvivalent *RecyclerView* elementa s horizontalnom orijentacijom u *Jetpack Composeu*. Na isti način *LazyColumn()* predstavlja ekvivalent *RecyclerView* elementa s vertikalnom orijentacijom.

Kako bi popunili *LazyRow()* ili *LazyColumn()* s podacima, postoje posebne *Composable* metode pomoću kojih možemo prikazati element ili listu elementa. Prema isječku kôda pod brojem 22. možemo primijetiti *item()* i *items() Composable* metode. Putem *item()* metode možemo prikazati jedan element, dok uz pomoć *items()* metode možemo prikazati listu elemenata. Važno je napomenuti da je redoslijed pozivanja ovih metoda vrlo bitan jer će se elementi dodavati onim redoslijed kojim su te metode pozvane. Tako će se u našem primjeru prvo prikazati element koji predstavlja video najavu serije, a nakon toga prikazati će se lista prikaza postera tv serije.

Za svaki od prikaza kreirana je posebna *Composable* metoda i opisuje kako pojedini element treba izgledati. Tako u prikazanom primjeru *VideoItem()* predstavlja element za prikaz video najave tv serije, dok *ImageItem()* predstavlja pojedini prikaz postera tv serije.

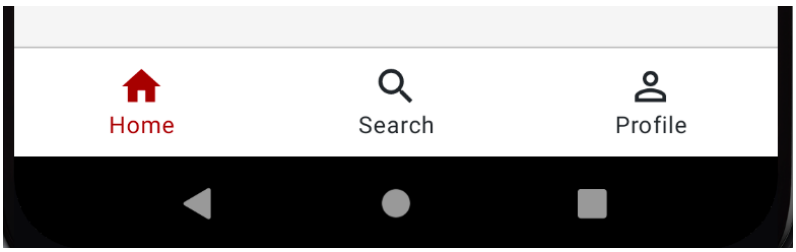
Raspored i poravnanje podređenih elemenata možemo mijenjati na isti način kao što to činimo za *Row()* i *Column() Composable* metode. Ako pogledamo isječak kôda pod brojem 22. možemo primijetiti da smo elemente rasporedili tako što smo dodali razmak od 12 točaka po inču između svakog elementa u listi.

Iz prikazanog primjera možemo primijetiti koliko je *Jetpack Compose* olakšao kreiranje lista i njihovo popunjavanje različitim elementima ovisno o tipu podatka. Isto tako smanjili smo vrijeme potrebno za izradu složenih lista budući da vrlo lako možemo dodati nove elemente unutar *LazyRow()* ili *LazyColumn().* Sve što moramo napraviti kako bi u postojeću listu dodali novi element je pozvati *item()* ili *items() Composable* metodu kojoj proslijedimo neku vlastitu *Composable* metodu.

5.4. Usporedba kreiranja navigacijske komponente

Većina današnjih aplikacija sadrži jedan oblik glavne navigacije. Često je ta navigacija predstavljena putem ikona na donjem dijelu ekrana. Na slici 13. prikazan je primjer donje navigacijske trake. Prva ikona u primjeru predstavlja početni ekran aplikacije na kojem se nalazi lista tv serija koje su trenutno u trendu. Element *Search* predstavlja ekran na kojem je moguće pretraživati tv serije, a na elementu *Profile* nalazi se popis svih trenutno praćenih tv serija od strane korisnika. Iz prikaza možemo zaključiti da postoje ukupno tri glavna odredišta u našoj aplikaciji, gdje *Home* predstavlja početno i glavno odredište.

U nastavku ove cjeline uspoređen je način kreiranja donje navigacijske trake s tri odredišta u XML-u i *Jetpack Composeu* te prikazan način navigacije iz jednog ekrana na drugi.



Slika 13: Prikaz glavne navigacijske komponente *Sofa Time* aplikacije

5.4.1. Kreiranje donje navigacijske trake u XML-u

U XML-u postoji posebna biblioteka kreirana od strane *Googlea* koja omogućuje rukovanje navigacijom unutar aplikacije. Zove je *Jetpack Navigation Component* i dio je *Jetpack* paketa biblioteka. Kreiranje donje navigacijske trake kao na slici 13. vrlo je lako postići koristeći navedenu biblioteku. Budući da se takva komponenta mora stalno prikazivati na dnu prilikom promjene ekrana, moramo ju postaviti unutar glavne aktivnosti. Da bi stvorili donju navigacijsku traku, prvo moramo definirati traku u XML datoteci glavne aktivnosti. U isječku kôda pod brojem 23. možemo vidjeti primjer kreiranja navedene navigacijske trake.

```
<com.google.android.material.bottomnavigation.BottomNavigationView
    android:id="@+id/bottom_nav"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_alignParentBottom="true"
    android:background="?android:attr/windowBackground"
    app:itemIconTint="@color/color_selector_bottom_nav_item"
    app:itemTextColor="@color/color_selector_bottom_nav_item"
    app:labelVisibilityMode="labeled"
    app:layout_constraintBottom_toBottomOf="parent"
    app:menu="@menu/drawer_bottom_nav_menu" />
```

Isječak kôda 23: Postavljanje navigacijske komponente u XML-u

U prikazanom primjeru potrebno je obratiti pozornost na *app:menu* atribut. To je posebni atribut *BottomNavigationView* resursa putem kojega postavljano *menu* XML resurs navigacijske trake. Takav XML resurs nalazi se unutar *menu* mape projekta, a primjer kako definirati navedeni resurs s tri elementa prikazan je u isječku kôda pod brojem 24. Svakom od elementa možemo odrediti id, ikonu i naziv. Važno je primijetiti da id *item* resursa mora odgovarati id-u fragmenta kojega želimo prikazati kada korisnik klikne na element. Time će *Jetpack Navigation Component* biblioteka znati koji fragment mora prikazati.

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">

    <item
        android:id="@+id/tvShowHomeFragment"
        android:icon="@drawable/ic_home"
        android:title="Home" />

    <item
        android:id="@+id/tvShowSearchFragment"
        android:icon="@drawable/ic_search"
        android:title="Search" />

    <item
        android:id="@+id/tvShowTrackedFragment"
        android:icon="@drawable/ic_person"
        android:title="Profile" />

</menu>
```

Isječak kôda 24: Primjer kreiranja *menu* resursa

Nakon što smo kreirali sve potrebne XML resurse, sve što moramo napraviti kako bi mogli „kretati“ između ekrana putem donje navigacijske trake je pozvati *setupWithNavController()* metodu unutar glavne aktivnosti. Navedenoj metodi moramo proslijediti *navController* objekt koji je zadužen za navigaciju unutar cijele aplikacije. Primjer povezivanja *navControllera* s *BottomNavigationView* resursom prikazan je u isječku kôda pod brojem 25.

```
binding.bottomNav.setupWithNavController(navController)
```

Isječak kôda 25: Postavljanje navigacijske komponente u aktivnosti

Metoda *setupWithNavController()* će automatski promijeniti ekran kada odaberemo element unutar donje navigacijske trake. Također, pozivom navedene metode *navController* će se sam brinuti oko promjeni boja elemenata donje navigacijske trake i upravljanjem stoga navigacije.

Kao što možemo primijetiti, XML pruža vrlo lak način kreiranja donje navigacijske komponente, a putem *Jetpack Navigation Component* biblioteke pruža automatiziran način navigacije između ekrana koji su vezani za donju navigacijsku komponentu. U sljedećoj cjelini prikazan je primjer kako putem *Jetpack Composea* možemo kreirati donju navigacijsku komponentu i postići prikazano ponašanje.

5.4.2. Kreiranje donje navigacijske trake u *Jetpack Composeu*

U *Jetpack Composeu* postoji posebna *Composable* metoda koja omogućuje prikaz donje navigacijske trake. Zove se *BottomNavigation()* te kao svakoj drugoj *Composable* metodi možemo proslijediti druge *Composable* metode koje će u ovom slučaju biti navigacijski elementi.

Kada kreiramo *BottomNavigation() Composable* metodu bez podređenih elemenata prikazati će se prazni okvir na donjem dijelu ekrana. Kako bi vidjeli navigacijske elemente moramo dodati *BottomNavigationItem() Composable* metodu kao podređeni element *BottomNavigation()* komponente. Ta metoda omogućuje prikaz i pozicioniranje elemenata unutar *BottomNavigation()* komponente. Kako bi olakšali kreiranje navigacijskih elemenata u projektu je kreirana posebna klasa zvana *BottomNavItem* putem koje su kreirani svi navigacijski elementi koji se nalaze u donjoj navigacijskoj traci. U isječku kôda pod brojem 26. prikaza je struktura navedene klase.

```
sealed class BottomNavItem(
        val title: Int,
        val icon: ImageVector,
        val route: String
)
```

Isječak kôda 26: Prikaz *BottomNavItem* klase

Nakon što smo kreirali listu *BottomNavItem()* elemenata, potrebno je proslijediti tu listu *BottomNavigation() Composable* metodi i za svaki element u listi pozvati *BottomNavigationItem()* metodu. U isječku kôda pod brojem 27. možemo primijetiti da je kreirana ekstenzijska metoda za *RowScope Composable* tip. Razlog tome je što *BottomNavigation()* raspoređuje elemente u obliku reda čime sadrži *content() lamba* metodu tipa *RowScope.* Unutar ekstenzijske metode pozovemo *BottomNavigationItem()* metodu kojoj prosjedimo potrebne informacije. Budući da prikazana ekstenzijska metoda kao parametar prima *BottomNavItem* tip podataka možemo na vrlo lak način dodati ikonu i tekst navigacijskom elementu koji će se prikazati u donjoj navigacijskoj traci.

```
@Composable
private fun RowScope.AddItem(
    screen: BottomNavItem,
    currentDestination: String?,
    navController: NavHostController
) {
    BottomNavigationItem(
        selectedContentColor = MaterialTheme.colors.primary,
        unselectedContentColor = MaterialTheme.colors.NavigationForeground,
        label = {
            Text(text = stringResource(screen.title))
        },
        icon = {
            Icon(imageVector = screen.icon, contentDescription =
"Navigation icon")
        },
        selected = currentDestination == screen.route,
        onClick = {
                navController.navigate(screen.route)
                …
            } )
```

Isječak kôda 27: Primjer implementacije *BottomNavigationItem Composable* metode

Poput XML-a, *Jetpack Compose* također za navigaciju koristi *navController* objekt koji je odgovoran za navigaciju između ekrana i upravljanje navigacijskog stoga. Putem *navigate()* metode možemo „navigirati“ na određeni ekran tako što proslijedimo metodi rutu ekrana u obliku niza znakova.

Budući da je donja navigacijska traka jedna od *Material Design* komponenti, pozicioniranje takvog elementa možemo odraditi putem *Scaffold() Composable* metode. Ta metoda omogućuje na vrlo lak način dodavanje i pozicioniranje standardnih *Material Design* komponenti kao što je donja navigacijska traka. Sve što trebamo napraviti je kreirati *Scaffold()* metodu i unutar *bottomBar lambda* metode pozvati *Composable* metodu unutar koje smo definirali *BottomNavigation()* komponentu. Osim donje navigacijske trake, putem *Scaffold()*

Composable metode imamo mogućnost dodati *Material Design* komponente poput gornje alatne trake (eng. *TopAppBar*), plutajuće akcijske tipke (eng. *Floating Action Button*) i mnogo drugih elemenata.

```
Scaffold(
    bottomBar = {
        AnimatedVisibility(
            visible = bottomBarState.value,
            enter = slideInVertically(initialOffsetY = { it }),
            exit = slideOutVertically(targetOffsetY = { it }),
            content = {
                BottomBar(
                    navItems = bottomNavItems,
                    navController = navController
                )
            }
        )
    },
    modifier = Modifier.fillMaxSize(),
    scaffoldState = scaffoldState
) { … }
```

Isječak kôda 28: Prikaz *Scaffold() Composable* metode

Iz prikazanih primjera možemo primijetiti da je u XML-u i *Jetpack Composeu* vrlo lako implementirati donju navigacijsku traku. U XML-u putem *Jetpack Navigation Component* biblioteke vrlo lako možemo povezati donju navigacijsku traku s ostatkom navigacije, dok putem *Composea* moramo unutar *Scaffold()* metode objediniti navedenu komponentu. Sama navigacija u *Jetpack Composeu* u trenutku pisanja ovog rada ne sadrži kompletni set funkcionalnosti te ima određena ograničenja poput mogućnosti prosljeđivanja samo primitivnih podataka (tekst, brojevi, *Boolean* vrijednosti, itd.) između ekrana.

U sljedećem poglavlju uspoređena je mogućnost ponovne iskoristivosti često korištenih UI komponenti. Ovdje se misli na kreiranje elementa korisničkog sučelja kojeg možemo lako uključiti u određeni dio aplikacije i time smanjiti potrebu za dupliciranjem kôda.

5.5. Usporedba mogućnosti ponovnog korištenja UI komponenti

Često postoji slučaj kada neki element korisničkog sučelja prikazujemo na više mjesta u aplikaciji. Kako bi izbjegli dupliciranje kôda, takav element kreiramo na jednom mjestu unutar našeg projekta i po potrebi koristimo u ostalim dijelovima aplikacije. U *Sofa Time* aplikaciji potrebno je prikazati ikonu i tekst prikazanu na slici 14. u slučajevima kada je određena lista tv serija prazna. Budući da u aplikaciji na više mjesta želimo prikazati takav element, u nastavku ove cjeline objašnjen je postupak kako u XML-u i *Jetpack Composeu* element prikazana na slici 14. pozvati na različitim mjestima u aplikaciji bez potrebe za ponovnom implementacijom.

Slika 14: Prikaz u slučaju prazne liste

Budući da u *View* sustavu većinu rasporeda pogleda i elemenata korisničkog sučelja kreiramo putem XML-a, postavlja se pitanje kako „ugraditi“ neki XML element u postojeći raspored pogleda. Jedan od najčešćih načina kako to možemo postići je korištenjem posebne XML oznake zvane *<include/>.* Navedenoj oznaci potrebno je putem *layout* atributa proslijediti korijenski identitet XML resursa kojega želimo prikazati.

Osim običnih pogleda, <*include/*> oznaci možemo proslijediti cijele rasporede pogleda što nam omogućuje ponovno korištenje vrlo složenih komponenti. Kako bi definirali koliki prostor će zauzeti element dodan preko <*include/*> oznake, potrebno je odrediti širinu i visinu elementa putem *android:layout_width* i *android:layout_height* atributa.

```
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingTop="10dp"
    tools:context=".presentation.tvShowSearch.TvShowSearchFragment">
…

<include
    android:id="@+id/emptyListMessage"
    layout="@layout/item_empty_list_message"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:visibility="gone"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="@id/guidelineTrailing"
    app:layout_constraintStart_toStartOf="@id/guidelineLeading"
    app:layout_constraintTop_toTopOf="parent" />

…

</androidx.constraintlayout.widget.ConstraintLayout>
```

Isječak kôda 29: Primjer korištenja <*include/*> oznake

U isječku kôda pod brojem 29. putem <*include/*> oznake proslijedili smo kompletni raspored pogleda kako bi mogli prikazati dizajn sa slike 14. Kako bi promijeniti atribute proslijeđenog prikaza možemo dodati id <*include/*> oznaci. Putem tog id-a možemo lako pristupiti ostalim elementima unutar fragmenta ili aktivnosti i tako promijeniti svojstva elementima.

U *Jetpack Composeu* nije potrebno dodati posebne oznake ili anotacije kako bi iskoristili određenu *Composable* metodu unutar druge. Svaka *Composable* metoda može sadržavati druge *Composable* metode kao podređene elemente gradeći time cijelo stablo ugniježdenih *Composable* metoda. Time je *Jetpack Compose* daleko fleksibilniji od XML-a u slučajevima kada želimo ponovno iskoristiti UI elemente na različitim mjestima unutar aplikacije.

Kako bi ponovno iskoristili određenu *Composable* metodu u *Jetpack Composeu*, sve što trebamo napraviti je kreirati metodu s željenim parametrima i označiti ju s @*Composable*

anotacijom. U isječku kôda pod brojem 30. kreirali smo *Composable* metodu koju želimo pozvati svaki puta kada dohvatimo praznu listu tv serija. Metoda sadrži *message* parametar koji određuje tekst koji će se prikazati preko *Text() Composable* metode. Tako u različitim listama možemo na vrlo lak način prikazati drugačiji tip poruke.

```
@Composable
fun EmptyListMessage(
    message: String
) {
    Box(
        modifier = Modifier
            .fillMaxSize()
            .padding(horizontal = 25.dp),
        contentAlignment = Alignment.Center
    ) {
        Column(
            verticalArrangement = Arrangement.spacedBy(10.dp),
            horizontalAlignment = Alignment.CenterHorizontally
        ) {
            …
            Text(
                text = message,
                style = MaterialTheme.typography.h3,
                color = MaterialTheme.colors.textPrimary,
                textAlign = TextAlign.Center
            )

        }
    }
}
```

Isječak kôda 30: Primjer izrade vlastite *Composable* metode za ponovnu uporabu

Nakon što smo kreirali *EmptyListMessage() Composable* metodu sa svim potrebnim elementima, možemo ju pozvati unutar bilo koje druge *Composable* metode. U isječku kôda pod brojem 31. primijetimo kako na temelju određenih uvjeta želimo prikazati *EmptyListMessage() Composable* metodu. Iz primjera možemo zaključiti koliko je zapravo lagano stvarati dinamične prikaze u *Jetpack Composeu* koristeći samo uvjete i parametre *Composable* metoda.

```
if (trackedTvShows.isEmpty() && !isLoading) {
    EmptyListMessage(message = "You are not tracking any Tv Show at the
moment")
}
```

Isječak kôda 31: Primjer prikaza *Composable* metode na temelju određenih uvjeta

Putem parametra, *Composable* metodama možemo znatno povećati ponovnu upotrebu budući da za svaki novi poziv metode imamo mogućnost proslijediti drugačiji podatke

i time prikazati različit sadržaj na različitim mjestima u aplikaciji. Važno je napomenuti da osim tipova podataka, *Composable* metodama možemo proslijediti i *lambda* metode što ih čini dodatno fleksibilnijima.

5.6. Usporedba kreiranja teme u XML-u i *Jetpack Composeu*

Kako bi uskladili stil elemenata korisničkog sučelja unutar cijele aplikacije, tim iz *Googlea* je kreirao *Material Design* smjernice koje nam omogućuju kreirati standardizirane elemente korisničkog sučelja. Takvi elementi imaju predodređen stil koji je definiram putem tema u projektu.

Temu u starom *View* sustavu kreiramo kao XML resurs. U isječku kôda pod brojem 32. prikazan je način definiranja teme *Sofa Time* aplikacije. Putem posebno imenovanih resursa kao što je *colorPrimary* možemo odrediti kako ćemo obojati komponentu koja posjeduje navedeni atribut. Kada definiramo temu aplikacije, sve komponente u aplikaciji će poprimiti definirali stil. Uz temu postoji individualni način postavljanja izgleda elementa, a to je putem *<style/>* oznake. Ta oznaka predstavlja skup atributa koji specificiraju izgled nekog pogleda. Stilom možemo odrediti atribute kao što su boja fonta, veličina fonta, boja pozadine i još mnogo toga.

```
<style name="Theme.SofaTimeXml"
parent="Theme.MaterialComponents.DayNight.NoActionBar">

    <item name="colorPrimary">@color/medium_red</item>
    <item name="colorPrimaryVariant">@color/dark_red</item>
    <item name="colorOnPrimary">@color/snow_white</item>

    <item name="colorSecondary">@color/star_yellow</item>
    <item name="colorOnSecondary">@color/black</item>

    <item name="colorTextPrimary">@color/semi_dark_gray</item>
    <item name="colorTextPrimaryDim">@color/medium_gray</item>
      ...
</style>
```

Isječak kôda 32: Definiranje teme aplikacije u XML-u

Pojedinom XML elementu možemo promijeniti stil putem određenih atributa. Tako *TextView* XML element sadrži *textColor* atribut putem kojega možemo izmijeniti boju teksta koja će se prikazati. Kako bi izmijeniti font i stil teksta koristimo *textAppearance* atribut kojemu

proslijedimo jedan od definiranih stila fontova unutar našeg projekta. U isječku kôda pod brojem 33. prikazan je primjer kako *TextView* XML elementu promijeniti boju teksta i tip fonta.

```
<TextView
    android:id="@+id/textViewTvShowTitle"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textAlignment="center"
    android: ="@style/TextBody1"
    android:textColor="?colorTextPrimary"
    tools:text="Game of Thrones" />
```

Isječak kôda 33: Primjer promjene stila *TextView* XML elementa

Ovime smo ukratko prikazali kako elementima korisničkog sučelja u XML-u promijeniti stil. U nastavku ove cjeline prikazano je kako putem *Jetpack Composea* postaviti temu i stil *Composable* metodama.

Temu aplikacije u *Jetpack Compose* kreiramo putem posebne *Composable* metode zvane *MaterialTheme().* Toj *Composable* metodi moramo proslijediti skup boja, fontova i oblika kroz *colors*, *typography* i *shapes* parametre. U isječku kôda pod brojem 34. prikazan je primjer kreiranja *MaterialTheme() Composable* metode.

```
MaterialTheme(
    colors = colors,
    typography = Typography,
    shapes = Shapes,
    content = content
)
```

Isječak kôda 34: Primjer definiranja teme u *Jetpack Composeu*

Putem navedene metode svaka komponenta korisničkog sučelja (poput teksta i tipki) će u aplikaciji poprimiti vizualnih stil i ponašanje definirano putem *colors* parametra.

Ponekad želimo ručno promijenili stil određene *Composable* metode. To možemo učiniti putem posebnih parametra *Composable* metoda gdje svaka *Composable* metoda sadrži vlastiti set mogućih promjena stilova. U isječku kôda pod brojem 35. promijenili smo boju teksta i stil fonta *Text() Composable* metode. Možemo primijetiti da smo boju i font dodali putem *MaterialTheme* objekta. Taj objekt objedinjuje sve *Material Design* komponente naše aplikacije poput boje, tipografije i oblika te omogućuje jednostavan pristup tip komponentama.

```
Text(
    text = story,
    style = MaterialTheme.typography.body2,
    color = MaterialTheme.colors.textPrimary,
)
```

Isječak kôda 35: Primjer promjene stila *Text() Composable* metode

Ponekad imamo potrebu proširiti osnovnu *Material Design* temu. *Jetpack Compose* nam omogućuje na vrlo lak način dodati npr. nove tipove boja. Tako na primjer putem *MaterialTheme* objekta želimo imati pristup posebnoj boji koja će nam služiti za postavljanje boje teksta. Najjednostavniji način kako dodati novu boju je putem ekstenzijske metode nad *Colors* objektom. U isječku kôda pod brojem 36. prikazan je primjer kreiranja *textPrimary* boje čime smo prošili set *MaterialTheme* novom bojom. Na isti način možemo proširiti tipografiju i oblike *MaterialTheme* objekta.

```
val Colors.textPrimary: Color get() = if (isLight) SemiDarkGray else
Color.White
```

Isječak kôda 36: Primjer dodavanja nove *MaterialTheme* boje

Iz prikazanih primjera možemo zaključiti da u XML-u i *Jetpack Compose* na vrlo slične načine možemo konfigurirati temu i stil elemenata korisničkog sučelja. Putem obje tehnologije u mogućnosti smo postaviti osnovni skup boja, tipografije i oblika te na vrlo lak način primijeniti željeni stil nad određenim elementom. *Jetpack Compose* je u aspektu proširenja osnovne teme nešto fleksibilniji od XML-a budući da putem ekstenzijskih metoda možemo vrlo lako dodati nove boje, tipografiju i oblike te time kreirati vrlo bogati skup stilova.

6. Zaključak

Ovim diplomskim radom usporedili smo načine kreiranja korisničkog sučelja u XML-u i *Jetpack Composeu*. Kroz teorijski dio opisali smo što su to korisnička sučelja te kako kreirati dobra korisnička sučelja. Zatim smo objasnili osnovne koncepte XML-a i *Jetpack Composea*. Za svaku tehnologiju prikazali smo životni ciklus i načine kako kreirati osnovne elemente korisničkog sučelja. Putem praktičnih primjera u radu mogli smo primijetiti kako u starom *View* sustavu postoji jasna podjela između kreiranja korisničkog sučelja i definiranja ponašanja UI elemenata. Svaki pogled i raspored pogleda kreirali smo unutar određene XML datoteke. Unutar tog dokumenta definirali smo samo kako određeni element mora izgledati kada se prikaže na zaslonu ekrana. Ponašanje i izmjenu elementa dodali smo programski unutar fragmenta ili aktivnosti. Kod *Jetpack Composea* situacija je puno drugačija. Cijela logika prikaza i izmjene UI elementa definirana je isključivo programskim putem kroz *Composable* metode. Kada bi mogli navesti prednosti jedne tehnologije u odnosu na drugu tada bi *Jetpack Composea* definitivno imao prednost u odnosu na XML kada je u pitanju lakoća kreiranja ponovno iskoristivih UI elemenata. Razlog tome je što svaka *Composable* metoda može sadržavati drugu *Composable* metodu. Također kreiranje složenih prikaza poput liste s različitim tipovima podatka je znatno jednostavnije u *Jetpack Composeu* budući da nije potrebno za svaku listu kreirati adaptere kao što je to slučaj s XML-om. Situacija u kojoj XML ima trenutno veću primjenu u odnosu na *Jetpack Compose* je prilikom prosljeđivanja podataka s jednog ekrana na drugi budući da osim primitivnih podataka u XML-u možemo proslijediti i složene tipove podataka, što je u trenutku pisanja ovog rada nemoguće u *Jetpack Composeu*.

Jetpack Compose verzija *Sofa Time* aplikacija izrađena je putem *Jetpack Compose* verzije 1.2.0. Prema mojem mišljenju navedena verzija sadrži apsolutno sve potrebne značajke za kreiranje modernih mobilnih aplikacija. Aplikacija izrađena u *Jetpack Compose* sadrži sve UI elemente kao i verzija aplikacije u XML-u s time da je za implementaciju *Jetpack Compose* verzije bilo potrebno uložiti manje vremena budući da je kreiranje korisničkog sučelja bilo puno jednostavnije za implementirati u odnosu na XML verziju.

Programska rješenja *Sofa Time* aplikacije izrađena su uz pomoć *Android Studio* razvojnog okruženja i javno su dostupna na *Githubu* putem sljedećih linkova:

- https://github.com/mangata-labs/sofa-time
- https://github.com/mangata-labs/sofa-time-xml

Popis literature

Allison, M. (2022). *Welcome to ConstraintLayout.com*. https://constraintlayout.com

Android Developers. (2022a). *Accessibility in Compose*. https://developer.android.com/jetpack/compose/accessibility

Android Developers. (2022b). *App architecture*. https://developer.android.com/topic/architecture/intro

Android Developers. (2022c). *Layouts*. https://developer.android.com/develop/ui/views/layout/declaring-layout

Android Developers. (2022d). *Material Components and layouts*. https://developer.android.com/jetpack/compose/layouts/material

Android Developers. (2022e). *The activity lifecycle*. https://developer.android.com/guide/components/activities/activity-lifecycle

Android Developers. (2022f). *Thinking in Compose*. https://developer.android.com/jetpack/compose/mental-model

Arriola, C. (2022). *Thinking in Compose*. https://medium.com/androiddevelopers/thinking-in-compose-c4ef150bb7cf

Babich, N. (2019). *The 4 Golden Rules of UI Design*. https://xd.adobe.com/ideas/process/ui-design/4-golden-rules-ui-design/

Bogode, S. (2021). *Understand the Activity Lifecycle*. https://openclassrooms.com/en/courses/4661936-develop-your-first-android-application/4679976-understand-the-activity-lifecycle

Castillo, J. (2022). *Composed modifiers in Jetpack Compose*. https://jorgecastillo.dev/composed-modifiers-in-jetpack-compose

Chapman, C. (2022). *Why Use Material Design? Weighing the Pros and Cons*. https://www.toptal.com/designers/ui/why-use-material-design

Chugh, A. (2022a). *Android Fragment Lifecycle*. https://www.digitalocean.com/community/tutorials/android-fragment-lifecycle

Chugh, A. (2022b). *Android Layout—LinearLayout, RelativeLayout*. https://www.digitalocean.com/community/tutorials/android-layout-linearlayout-relativelayout

Figma. (2022). *Google's Material Design: A design system with mass adoption*. https://www.designsystems.com/googles-material-design-a-design-system-with-mass-adoption/

Genç, M. (2022). *Android View Binding using in Activity, Fragment, Dialog, View, RecyclerViewHolder*. https://innovance.com.tr/android-view-binding-using-in-activity-fragment-dialog-view-recyclerviewholder/

Ghita, C. (2022). *Kickstart Modern Android Development with Jetpack and Kotlin*. Packt Publishing.

Ghosh, S. (2021). *Why Single Activity Architecture is preferred Over Multiple Activity Architecture in Android App Development?* https://www.webskittersacademy.in/why-single-activity-architecture-android-app-development/

Griffiths, D., & Griffiths, D. (2021). *Head First Android Development, 3rd Edition*. O'Reilly Media.

Guerrero, R. (2022). *Managing State in Jetpack Compose*. https://www.raywenderlich.com/30172122-managing-state-in-jetpack-compose

Ilicic, M. (2017). *ConstraintLayout in practice*. https://blog.undabot.com/constraintlayout-in-practice-b3e47df7a618

Interaction Design Foundation. (2022). *User Interface Design*. https://www.interaction-design.org/literature/topics/ui-design

Künneth, T. (2022). *Android UI Development with Jetpack Compose*. Packt Publishing.

Majnerić, S. (2022). *The Fundamentals of Jetpack Compose*. https://arsfutura.com/magazine/the-fundamentals-of-jetpack-compose/

Material Design. (2022). https://material.io/design/introduction

Mendoza, J. (2021). *Unidirectional Data Flow for Android UIs*. https://medium.com/swlh/unidirectional-data-flow-in-for-android-uis-5edd1386f40d

Nielsen, J. (2020). *10 Usability Heuristics for User Interface Design*. https://www.nngroup.com/articles/ten-usability-heuristics/

Okolie, U. (2019). *Introducing XML*. https://medium.com/@urluchy/introducing-xml-6587b91bf49d

R., M. (2017). *ConstraintLayout Chains*. https://medium.com/@nomanr/constraintlayout-chains-4f3b58ea15bb

Richardson, L. (2020). *Understanding Jetpack Compose—Part 1 of 2*. https://medium.com/androiddevelopers/understanding-jetpack-compose-part-1-of-2-ca316fe39050

StatCounter. (2022). *Mobile Operating System Market Share Worldwide*. https://gs.statcounter.com/os-market-share/mobile/worldwide

Tidwell, J., Brewer, C., & Valencia, A. (2019). *Designing Interfaces, 3rd Edition*. O'Reilly Media, Inc.

Toporov, E. (2013). *IntelliJ IDEA is the base for Android Studio, the new IDE for Android developers*. https://blog.jetbrains.com/blog/2013/05/15/intellij-idea-is-the-base-for-android-studio-the-new-ide-for-android-developers/

Trengrove, B. (2022). *Jetpack Compose: Debugging Recomposition*. https://medium.com/androiddevelopers/jetpack-compose-debugging-recomposition-bfcf4a6f8d37

Zhukovich, A. (2021). *Jetpack Compose: Layouts*. https://alexzh.com/jetpack-compose-layouts/

Popis slika

Popis tablica

Popis kôdova