Android-arhitektura u Kotlinu

Balint, Tino

Master's thesis / Diplomski rad

2018

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Organization and Informatics / Sveučilište u Zagrebu, Fakultet organizacije i informatike**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:211:942403

Rights / Prava: Attribution-ShareAlike 3.0 Unported/Imenovanje-Dijeli pod istim uvjetima 3.0

Download date / Datum preuzimanja: **2024-04-24**

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Repository / Repozitorij:

Faculty of Organization and Informatics - Digital Repository

zir.nsk.hr

dabar
DIGITALNI AKADEMSKI ARHIVI I REPOZITORIJI

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
V A R A Ž D I N

Tino Balint

ANDROID ARHIKTEKTURA U KOTLINU

DIPLOMSKI RAD

Varaždin, 2018.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Tino Balint

Matični broj: 45332/16-R

Studij: Informacijsko i programsko inženjerstvo

ANDROID ARHITEKTURA U KOTLINU

DIPLOMSKI RAD

Mentor:

Prof. dr. sc. Danijel Radošević

Varaždin, lipanj 2018

Sadržaj

1. Uvod 1
2. Kotlin 2
   2.1. Platforme s kojima *Kotlin* radi 2
   2.2. Promijenjene funkcionalnosti Kotlin-a 3
      2.2.1. Primarni konstruktor 3
      2.2.2. Sekundarni konstruktor 4
      2.2.3. Nasljeđivanje 5
      2.2.4. Nadjačavanje 6
      2.2.5. Sučelja 7
      2.2.6. Statične metode 9
   2.3. Nove funkcionalnosti u *Kotlin-u* 9
      2.3.1. Ekstenzijske funkcije 9
      2.3.2. Podatkovne klase 10
      2.3.3. Destrukturirajuće deklaracije 11
      2.3.4. Funkcije konteksta 12
         2.3.4.1. Funkcija *apply* 13
         2.3.4.2. Funkcija *also* 14
         2.3.4.3. Funkcija *let* 14
         2.3.4.4. Funkcija *run* 15
         2.3.4.5. Funkcija *with* 15
3. Arhitektura *android* aplikacija 16
   3.1. N-slojna arhitektura 16
   3.2. Značajke android arhitekture 18
      3.2.1. Injektor zavisnosti 18
      3.2.2. Biblioteka *Dagger* 20
      3.2.3. Reaktivno programiranje 26
   3.3. *MVP(Model-View-Presenter)* arhitektura 29
      3.3.1. *Model* 30
      3.3.2. *Presenter* 31
      3.3.3. *View* 31
   3.4. *MVC(Model-View-Controller)* arhitektura 32
      3.4.1. *Model* 33
      3.4.2. *Controller* 34
      3.4.3. *View* 34

3.5. MVVM(Model-View-ViewModel) arhitektura 36
3.5.1. *Model* 36
3.5.2. ViewModel 36
3.5.3. *View* 37
3.6. Usporedba arhitektura 38
3.6.1. Testiranje 38
3.6.2. Održavanje 39
3.6.3. Korištenje memorije 40
4. Zaključak 42
5. Popis slika 43
6. Literatura 44

1. Uvod

Kotlin projekt je krenuo 2011. godine kada je tvrtka *JetBrains* započela njegov razvoj. U to vrijeme nije bilo programskih jezika koji obuhvaćaju funkcionalnosti današnjih modernih jezika poput *Swift*-a. Jedini sličan programski jezik je bio *Scala* koja je imala problem dugog kompiliranja koda. 2012. Projekt postaje otvorenog koda pod *Apache 2* licencom. Prva verzija 1.0 izlazi 2016. godine, a najveća prekretnica dogodila se 2017. godine kada je projekt dobio podršku u razvoju i održavanju od strane *Google*-a. (Martin Heller, 2018).

Kotlin je statički napisan programski jezik koji se pokreče pomoću *Java* virtualne mašine zbog čega se može kompilirati u *Java* byte kod. Osim *Jave*, može se kompilirati i u *JavaScript* kod zbog čega se može koristiti i za razvoj više-platformskih aplikacija.

Od *Andorid Studio* verzije 3.0, *Kotlin* je ugrađen kao jedna od primarnih funkcionalnosti gdje korisnik može birati između programiranja u *Kotlin-u* ili *Jave*, ali je moguće programirati i u oba jezika istovremeno sve dok je kod raspoređen u odgovarajuće klase.

Način pisanja koda se razlikuje od dosadašnjeg programiranja u Javi. Iako je objektno orijentirani programski jezik, prvenstveno se piše stilom funkcionalnog jezika koji koristi *lambda* izraze. *Lambda* je anonimna funkcija, odnosno funkcija koja nema identifikator. Često se koristi kao argument koji treba proslijediti, a da se izbjegne pisanje definicije funkcije. Iz tog razloga koristi se samo jednom ili mali broj puta jer ju je sintaktički jednostavnije napisati od imenovanih funkcija. (Britt Ballard, 2018.)

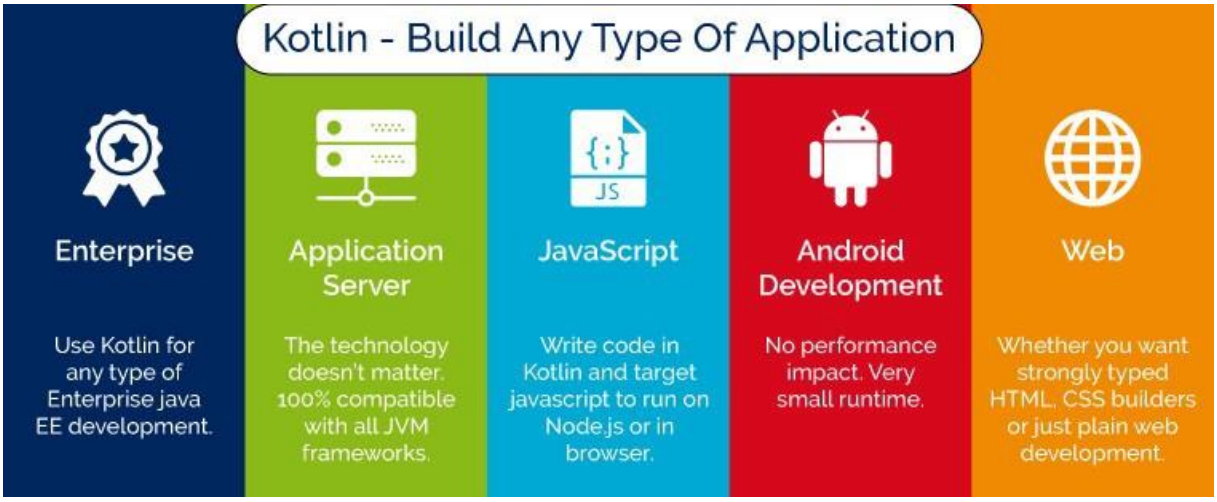
Osim funkcijskog stila pisanja, postoje mnoge prednosti koje Kotlin pruža kako bi pojednostavnio i ubrzao pisanje koda. Takav način rada dobiven je objedinjavanjem jezika poput Scale, Swifta i JavaScripta tijekom svojih godina razvoja. U ovom radu će biti prikazane prednosti korištenja *Kotlin-a* u odnosu na *Javu* te kako se on može iskoristiti za izradu različitih arhitektura za *Android* aplikacije.

2. Kotlin

2.1. Platforme s kojima *Kotlin* radi

Iako je *Kotlin* započeo svoj razvoj 2011. godine, trenutno je jedan od najmodernijih jezika. U posljednjih sedam godina svojeg razvoja, jezik se konstantno razvija i unaprjeđuje novim funkcionalnostima i svojstvima. Zahvaljujući podršci od strane *Google-a*, očekuje se i nastavak tog trenda te sve više programera prelazi s korištenja Jave na Kotlin.

Kotlin je najviše pogodio tržište razvoja Android aplikacija, ali bitno je napomenuti da sve veći broj programera koji Koriste Javu ili JavaScript programski jezik za razvoj web stranica i pozadinskih servisa prelazi na potpuno korištenje Kotlin-a (Adit Microsys, 2016).



Slika 1. Kotlin za razvoj više vrsta aplikacija **(Izvor: Kotlin - Build Any Type Of Application, 2017).**

Slika 1 pokazuje za što se sve *Kotlin može koristiti.* Možemo vidjeti da ga možemo koristiti za izgradnju svih vrsta aplikacija osim onih za *iOS* operacijski sustav. Iako je to trenutno nemoguće, trenutno se radi na verziji koja se zove „Kotlin Native“ pomoću koje bi trebali moći napisati aplikaciju za *android* i *iOS* u isto vrijeme zbog podrške za razvoj više-platformskih aplikacija.

2.2. Promijenjene funkcionalnosti Kotlin-a

Kotlin kao programski jezik uvodi promjene u pisanju objektno orijentiranog koda u odnosu na programske jezike iz „C“ generacije. Promjene se odnosne na sintaksu definiranja sučelja, nasljeđivanja, pisanja konstruktora, ali uvode se i neki noviteti koji olakšavaju razvoj. (Kotlin, 2018).

2.2.1. Primarni konstruktor

U *Kotlin-u* se konstruktori mogu definirati kao primarni i sekundarni. Primarni konstruktor definira se u zaglavlju klase tako da se ispred zaglavlja doda ključna riječ „constructor“ i unutar zagrada navedemo parametre konstruktora.

```
class Osoba constructor(ime : String){}
```

Ukoliko konstruktor ne sadrži anotacije ili modifikatore vidljivosti, moguće je izbaciti ključnu riječ „constructor“ i samo navesti parametre.

```
class Osoba(ime : String){}
```

U oba primjera kreiramo klasu Osoba koja sadrži jedan parametar konstruktora „ime“. Kod definiranja parametara možemo uočiti kako se prvo piše „ime“, a zatim se dodaje tip parametra dodavanjem dvotočke između.

Kada koristimo primarni konstruktor, on ne može sadržavati dodatan kod poput ispisa na ekran. Ukoliko želimo da tijekom poziva konstruktora izvršimo dodatan kod, potrebno je definirati poseban dio klase unutar ključne riječi „init“.

```
class Osoba constructor(ime : String){
    init{
        println("Ime osobe je: $ime")
    }
}
```

Ovakav poziv konstruktora kod inicijalizacije će ispisati i tekst koji sadrži parametar iz konstruktora. Iz navedenog primjera možemo vidjeti da je u Kotlinu moguće koristiti parametre unutar znakovnog niza ukoliko ispred parametra dodamo specijalni znak $.

Česta je praksa proslijediti parametre u klasu kroz konstruktor i zatim ih zapisati u varijable te klase kako bi se te vrijednosti spremile. Ukoliko se koriste varijable istih naziva, koristi se ključna riječ „this“. Taj postupak *Kotlin* radi u pozadini tako da nije potrebno definirati varijable unutar klase već se one navode unutar zaglavlja dodavanjem ključnih riječi *var* ili *val*.

```
class Osoba constructor(val ime:String, var prezime:String) {}
```

Prilikom poziva koda iz primjera, klasa će definirati varijable ime i prezime, te kod poziva konstruktora će te varijable instancirati vrijednostima iz parametara. Ovaj pristup ubrzava pisanje koda jer se sve može napisati u jednoj liniji.

Uočite ključne riječi *val* i *var*. Ukoliko ispred parametra ili varijable napišemo ključnu riječ *val*, to znači da ta varijabla ne može više mijenjati svoju vrijednost unutar navedene klase, dok ako napišemo *var* ona se može mijenjati. Ova svojstva ekvivalentna su korištenju odnosno ne korištenju ključne riječi „final“ u jezicima iz „C“ generacije. Prema smjernicama programiranja, preporučuje se korištenje ključne riječi *val* kada god je to moguće kako bi se izbjegle neočekivane situacije u kojima bi varijabla promijenila svoju vrijednost.

2.2.2. Sekundarni konstruktor

Kako bi definirali sekundarni konstruktor, potrebno je koristiti ključnu riječ „constructor“ unutar same klase.

```
class Osoba {
    constructor(ime: String)
}
```

Ukoliko klasa već sadrži i primarni konstruktor, sekundarni konstruktor mora delegirati primarnom konstruktoru. To je moguće direktno putem primarnog konstruktora ili indirektno putem drugog sekundarnog konstruktora. Delegacija se izvodi pomoću ključne riječi „this“.

```
class Osoba(ime: String) {
    constructor(ime: String, prezime: String) : this(ime)
}
```

Iz primjera možemo vidjeti kako sekundarni konstruktor delegira parametar „ime“ primarnom konstruktoru, a pritom uvodi i novi parametar „prezime“. Kod definiranja novog sekundarnog konstruktora uvijek je potrebno zadovoljiti sve uvjete primarnog konstruktora, odnosno delegirati sve parametre koje primarni konstruktor sadrži.

Ukoliko klasa ne sadrži primarni konstruktor, delegacija se izvršava implicitno. Bitno je napomenuti da se sav kod primarnog konstruktora izvodi prije koda sekundarnog konstruktora stoga će se inicijalizacijski kod unutar „init“ djela primarnog konstruktora izvršiti prije koda definiranog unutar sekundarnog konstruktora.

Ako klasa ne definira niti primarni niti sekundarni konstruktor, automatski će se generirati prazan primarni konstruktor s javnim modifikatorom vidljivosti. Ukoliko ne želimo da klasa ima konstruktor, to možemo napraviti dodavanjem privatnog modifikatora vidljivosti ispred ključne riječi „constructor“.

```
class Osoba private constructor () { }
```

2.2.3. Nasljeđivanje

Svaka klasa u *Kotlin-u* nasljeđuje klasu „Any“. Klasa „Any*“* je nadklasa koja ima metode „equals*()“, „*hasCode*()“* i „toString*()“* te ne sadrži varijable. Nasljeđivanje se izvodi pomoću operatora dvotočke, a da bi bilo moguće potrebno je ispred klase koju želimo naslijediti napisati ključnu riječ *open* koja funkcionira suprotno od ključne riječi „final“, odnosno, omoguće nasljeđivanje. Svaka klasa koja nije specificirana s ključnom riječi „open“ je automatski postavljena kao „final“.

```
open class Osoba constructor(ime : String){}
class Dijete(ime: String) : Osoba(ime) {}
```

Ako podklasa ima primarni konstruktor, ona mora pomoću parametara tog konstruktora inicijalizirati nadklasu kao što je to napravljeno u primjeru. Ukoliko klasa nema primarni konstruktor, mora to napraviti u svakom sekundarnom konstruktoru kojeg posjeduje korištenjem ključne riječi *super*.

```
open class Osoba constructor(ime : String){}

class Dijete : Osoba {
    constructor(ime: String) : super(ime)
}
```

2.2.4. Nadjačavanje

Kao što klase koje nasljeđujemo, tako i metode nadklase koje želimo nadjačati trebaju biti specificirane pomoću ključne riječi „open“.

```
open class Osoba constructor(ime: String) {

    open fun jedi() {}
    fun radi() {}
}

class Dijete : Osoba {
    constructor(ime: String) : super(ime)

    override fun jedi() {}
}
```

U navedenom primjeru, nadklasa sadrži dvije funkcije od kojih je jedna specificirana ključnom riječi „open“, a druga nije. U podklasi je nužno nadjačati metodu koja sadrži ključnu riječ „open“ dok bi u suprotnom, definiranje metode koja nema tu ključnu riječ izazvalo pogrešku od strane kompilatora.

Ukoliko želimo koristiti svojstva ili metode nadklase unutar podklase, možemo to učiniti pomoću ključne riječi *super*.

```
open class Osoba constructor(ime: String) {

    open fun jedi() {
        print("Osoba jede")
```

```
    }

}

class Dijete : Osoba {
    constructor(ime: String) : super(ime){
        super.jedi()
    }

    override fun jedi() {
        print("Dijete jede")
    }
}
```

U navedenom primjeru pozivom metode „super.jedi()“ osiguravamo da će se ispisati tekst "Osoba jede" umjesto teksta "Dijete jede" prilikom inicijalizacije objekta.

Kao što se sav kod primarnog konstruktora izvrši prije nego kod sekundarnog konstruktora, tako se i kod nadklase izvršava prije koda podklase i njenih nadjačanih metoda. Bitno paziti ukoliko nadjačavamo svojstva koja se koriste prilikom inicijalizacije nadklase jer možemo dobiti neočekivani rezultat.

2.2.5. Sučelja

U *Kotlin-u* se sučelja definiraju pomoću ključne riječi „interface“, a implementiraju se na isti način kao što se izvodi i nasljeđivanje, pomoću dvotočke.

```
interface Osoba {
    fun jedi();
}

class Dijete : Osoba {
    override fun jedi() {}
}
```

Metode koje su deklarirane u sučelju su javne i dostupne svima koji koriste to sučelje. Osim metoda moguće je definirati i nepromjenjiva svojstva i varijable. U tom slučaju potrebno ih je nadjačati jer nepromjenjiva svojstva u pozadini imaju definiranu metodu za dohvaćanje tog podatka *get()*.

```
interface Osoba {
    val prezime: String
}

class Dijete : Osoba {
    override val prezime: String
        get() = "Neko prezime"
}
```

Ukoliko klasa implementira više sučelja koji imaju istu metodu, ona mora nadjačati tu metodu. Da bi se specificiralo metodu čijeg sučelja želimo pozvati, potrebno je dodati izraz koji se tvori od ključne riječi *super* i imena klase unutar znakova manje i veće, *super<ime sučelja>.metoda()*.

```
interface Zivotinja {
    fun jedi() {
        print("Zivotinja jede")
    }
}

interface Sisavac{
    fun jedi(){
        print("Sisavac jede")
    }
}
```

```
class Dijete : Zivotinja, Sisavac {
    override fun jedi() {
         super<Zivotinja>.jedi()
    }
}
```

U prethodnom primjeru prilikom poziva metoda *jedi()* nad objektom *Dijete*, ispisati će se **"Zivotinja jede"** jer smo eksplicitno naveli da želimo pozvati metodu od sučelja „Zivotinja“.

2.2.6. Statične metode

U *Kotlin-u* ne postoje statične metode već se preporučuje pozivanje metoda na razini paketa. Ukoliko nam je nužno imati metode koje želimo pozivati bez da kreiramo instancu klase, možemo koristiti objekte koji se zovu „companion object“.

```
class Muskarac {
    companion object {
        const val spol : String = "Muški"
        var ime: String = "Pero"

        fun jedi(){}
    }
}
```

Ukoliko želimo koristiti metodu ili varijabli iz „companion object-a“ potrebno mu je pristupiti preko imena klase. Uočite i ključnu riječ „const“ ispred varijable *spol*. Konstante u *Kotlin-u* nije moguće definirati osim ako se navode unutar *companion object-a*. Da bi pristupili ovim vrijednostima iz druge klase, potrebno im je dodati javni modifikator vidljivosti.

2.3. Nove funkcionalnosti u *Kotlin-u*

2.3.1. Ekstenzijske funkcije

Ekstenzijske funkcije nam omogućavaju da dodamo ili promijenimo već postojeće funkcije neke klase bez da ju moramo naslijediti ili implementirati. Kako bi dodali ekstenzijsku funkciju, potrebno je navesti klasu te ime funkcije nakon točke, „fun“ *ImeKlase.ime funkcije(parametri).*

```
fun String.podjeliZnakovniNiz() {
    this.forEach {
        println(it)
    }
}
```

U danom primjeru definiramo funkciju *podjeliZnakovniNiz* nad već postojećom klasom „String“. Funkcija ne prima parametre nego koristi ključnu riječ *this* kako bi pristupila znakovnom nizu nad kojim je pozvana. Funkcija iterira po svakom znaku u nizu i ispisuje ga u

novi red. Ekstenzijske funkcije najčešće se postavljaju u gornji sloj aplikacije direktno ispod paketa kako bi bile dostupne svim klasama.

```
"Znakovni niz".podjeliZnakovniNiz()
```

Osim definiranja novih funkcija moguće je i izmijeniti postojeće. Ekstenzijske funkcije su vrlo moćne jer se mogu koristiti kao biblioteke nad klasama koje su česte u razvoju aplikacija. Iz tog razloga *Google* je napravio gotovu klasu s ekstenzijskim funkcijama koje se najčešće koriste kako bi bile dostupne svima. Da bi se pristupilo toj klasi u android projektu potrebno je dodati sljedeću liniju „apply plugin: 'kotlin-android-extensions"

Najčešći primjer ekstenzijske funkcije je prikazivanje ili sakrivanje pojedinih elemenata na ekranu.

```
fun View.prikazi(show: Boolean = true) {
    visibility = if (show) VISIBLE else GONE
}

fun View.prikazi(hide: Boolean = true) {
    visibility = if (hide) GONE else VISIBLE
}
```

Korištenjem ekstenzijskih funkcija dovoljno je pozvati metodu nad nekim elementom ekrana, dok bi u protivnom svaki put bilo potrebno postavljati vidljivost elementa.

Ekstenzije moguće je definirati i nad svojstvima, npr. ako želimo svojstvo koja vraća zadnji element u listi definirati ćemo svojstvo „zadnjiElement".

```
val <T> List<T>.lastIndex: Int
    get() = size - 1
```

2.3.2. Podatkovne klase

Prilikom izrade aplikacije, često nam trebaju klase koje služe samo kao modeli nekog objekta iz stvarnog svijeta. Takve klase često ne sadrže dodatne metode nego samo atribute realnih objekata. U *Javi* takve klase se nazivaju *POJO* (eng. *Plain Old Java Object*) obični stari java objekti. Kada koristimo takve klase, potrebne su nam neke ugrađene metode za kopiranje objekata, uspoređivanje ili pretvaranje u znakovni niz.

U *Javi* je takve metode potrebno napisati ručno zbog čega klase od samo 5 atributa znaju biti dugačke i preko 100 linija. Da bi se ubrzalo pisanje takvih klasa, *Kotlin* uvodi podatkovne klase, odnosno klase koje će u pozadini već imati implementirane metode koje su nam potrebne. Podatkovne klase su obične klase koje dodatno koriste ključnu riječ *data.*

```
data class Osoba(val ime: String,
            val prezime: String,
            val oib: Long,
            val spol : String,
            val dob : Int)
```

Ovako napisana klasa u pozadini ima implementirane metode *toString(), equals()* i *copy(), hasCode().* Uočite da klasa ne sadrži vitičaste zagrade niti metode za postavljanje ili dohvaćanje podataka. *Kotlin* svojstva imaju implementirane te metode u pozadini i prilikom pisanja imena svojstva one će se pozivati bez da programer mora brinuti o tome.

2.3.3. Destrukturirajuće deklaracije

Kotlin je jedan od programskih jezika kod kojeg nije potrebno navoditi tip podataka varijable koju inicijaliziramo, osim ako to eksplicitno želimo. Ukoliko definiramo neku varijablu ili svojstvo bez zadavanja tipa podataka i inicijaliziramo ju na neku vrijednost, ona će implicitno poprimiti tip podataka te vrijednosti.

```
val ime = "Tino"
```

U ovom primjeru svojstvo „ime“ će implicitno poprimiti tip podataka „String“. Osim implicitnog postavljanja tipa podataka, *Kotlin* uvodi i destrukturirajuće deklaracije, odnosno ukoliko imamo neki tip podataka koji sadrži više svojstava, npr. „Pair“, možemo ga pohraniti u dekonstruirajuću deklaraciju tako da stavimo proizvoljne nazive svojstava između oblih zagrada.

```
val imePrezime = Pair<String, String>("Tino", "Balint")
val (ime, prezime) = imePrezime
```

Korištenjem dekonstruirajućih deklaracija dobivamo neimenovani objekt s imenovanim parametrima, za razliku od imenovanog objekta kod kojeg ne znamo imena parametara.

Ukoliko bismo željeli pristupiti imenu ili prezimenu preko para „imePrezime“ morali bismo zvati atribute „first“ ili „second“ nad tim objektom. U tom slučaju nije nam vidljivo da li se radi o imenu ili prezimenu ako ne pogledamo dublje u kod. U ovom primjeru ako želimo pristupiti imenu ili prezimenu možemo to napraviti direktno jer smo ih imenovali.

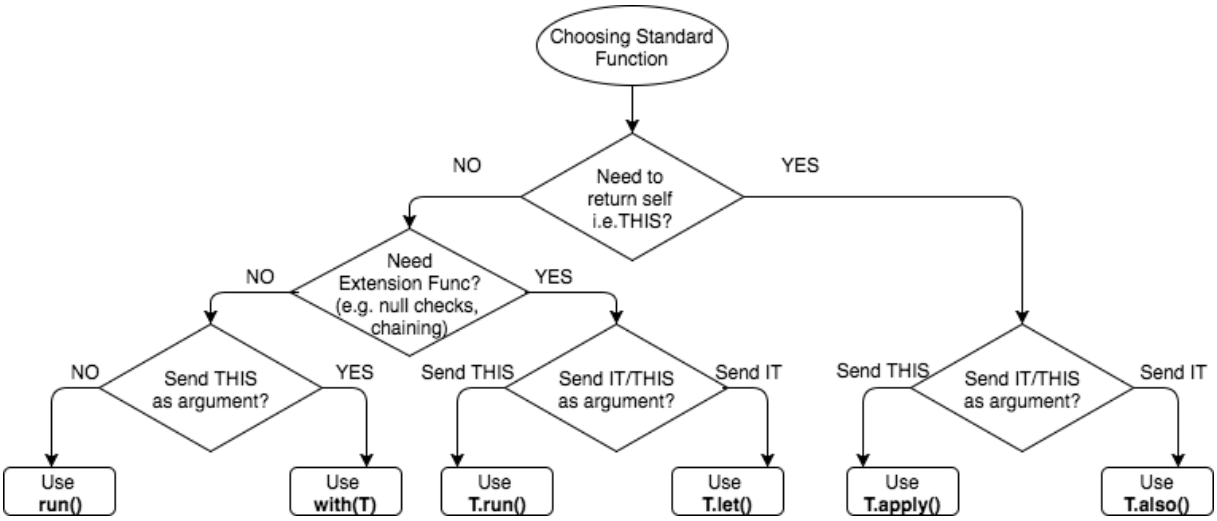
2.3.4. Funkcije konteksta

Funkcije konteksta su funkcije koje koristimo nad bio kojim objektom kako bi pristupili tom objektu na lakši način pod drugačijim kontekstom. Postoji pet takvih funkcija, *let(), apply(), run(), with()* i *also().* Sve funkcije rade na sličan način, ali postoji razlika u kojem slučaju se trebaju koristiti. (김태수, 2017).

Postoje dvije razlike između navedenih funkcija, unutar vitičastih zagrada funkcije objekt postaje *it* ili *this*, odnosno ukoliko želimo pristupiti objektu nad kojim zovemo funkciju ne moramo više pisati njegovo ime već mu pristupamo iz konteksta.

Kada radimo s podacima nekad želimo kao povratnu vrijednost vratiti sam objekt nad kojim radimo dok u drugim slučajevima želimo vratiti neki novi objekt koji će biti rezultat akcija. To su neki od parametara koji odlučuju kada ćemo koristiti neku od navedenih funkcija umjesto druge.

Na slici 2 možemo vidjeti dijagram koji objašnjava kada se koja funkcija treba koristiti.



Slika 2. Dijagram korištenja funkcija konteksta (*Izvor*: Choosing Standard Function, 2017)

2.3.4.1. Funkcija *apply*

Funkcija *apply* je jedna od dvije funkcije koja vraća objekt nad kojim se poziva. Svrha funkcije je da se nad tim objektom rade promjene, ali da nije potrebno stalno pisati ime objekta s kojim radimo već ćemo njemu pristupiti iz konteksta *this*.

Ako npr. želimo promijeniti vidljivost, tekst i boju nekog tekstualnog elementa na ekranu, to ćemo napraviti brže jer naš objekt postaje *this*, odnosno ako želimo zvati metode ili mijenjati svojstva tog objekta, možemo to raditi direktno.

Java

```
textView.setText("Novi text");
textView.setVisibility(VISIBLE);
textView.setColor(BLUE);
```

Kotlin

```
textView.apply {
    this.color = BLUE
    text = "Novi text"
    visibility = VISIBILE
}
```

Iako izgleda kao da nije velika razlika, kada objekt ima duže ime ili želimo promijeniti više svojstava, korištenje funkcije *apply* smanjiti će veliku količinu koda. Svojstvu „color“ eksplicitno pristupamo preko ključne riječi *this*, ali ju možemo izostaviti i ona će se pozivati implicitno.

Uočite kako u K*otlinu* možemo direktno mijenjati svojstava jer imaju ugrađene funkcije za postavljanje i dohvaćanje, dok u *Javi* moramo zvati posebne metode kako bi promijenili stanja objekta.

2.3.4.2. Funkcija *also*

Funkcija *also* je vrlo slična funkciji *apply*. Također ju koristimo kada želimo napraviti promjene nad postojećim objektom i želimo vratiti taj objekt kao rezultat, međutim u ovom slučaju ne pristupamo objektu preko ključne riječi *this* već *it*. To nam može biti korisno ukoliko želimo pristupiti nekom objektu iz klase u kojoj se trenutno nalazimo preko ključne riječi *this* što ne bi smo mogli da smo koristili funkciju *apply*.

Java

```
textView.setText("Novi text");
textView.setVisibility(VISIBLE);
textView.setColor(BLUE);
```

Kotlin

```
textView.also {
    it.color = BLUE
    it.text = "Novi text"
    it.visibility = VISIBILE
}
```

Kao što vidimo na primjeru, kod u *Kotlinu* se povećao jer je sada uvijek potrebno pristupiti objektu preko ključne riječi *it*, ali imamo i dodatnu mogućnost pristupiti objektu klase u kojoj se nalazimo preko ključne riječi *this*.

2.3.4.3. Funkcija *let*

Ukoliko ne želimo vratiti objekt s kojim radimo kao povratnu vrijednost koristimo sve preostale funkcije. Slično kao i kod funkcije *also*, funkcija *let* radi s objektom na način da mu se pristupa preko ključne riječi *it*.

```
osoba.let {
    it.jedi()
    val vrijemeSpavanja = it.spavaj()
    return vrijemeSpavanja
}
```

U danom primjeru pozivamo metode „jedi()“ i „spavaj()“. Metoda spavaj vraća vrijednost koliko dugo je osoba spavala, a tu vrijednost vraćamo kao rezultat funkcije za razliku od prethodno navedenih funkcija gdje se implicitno vraćao objekt nad kojim smo radili.

2.3.4.4. Funkcija *run*

Ukoliko želimo napraviti istu stvar kao kod funkcije *let*, ali želimo koristiti objekt u kontekstu ključne riječi *this*, koristiti ćemo funkciju *run.*

```
osoba.run {
    jedi()
    val vrijemeSpavanja = spavaj()
    return vrijemeSpavanja
}
```

2.3.4.5. Funkcija *with*

Funkcija *with* je skoro identična funkciji *run*. Jedina razlika je da je funkcija *run* ekstenzijska funkcija dok *with* nije. To znači da funkciju *run* možemo pozivati u lancu objekata odnosno nakon što pozovemo jednu funkciju na nju možemo ulančati funkciju *run*.

Kada koristimo funkciju *with* ne dodajemo ju na kraj ulančavanjem već je objekt s kojim želimo raditi argument funkcije.

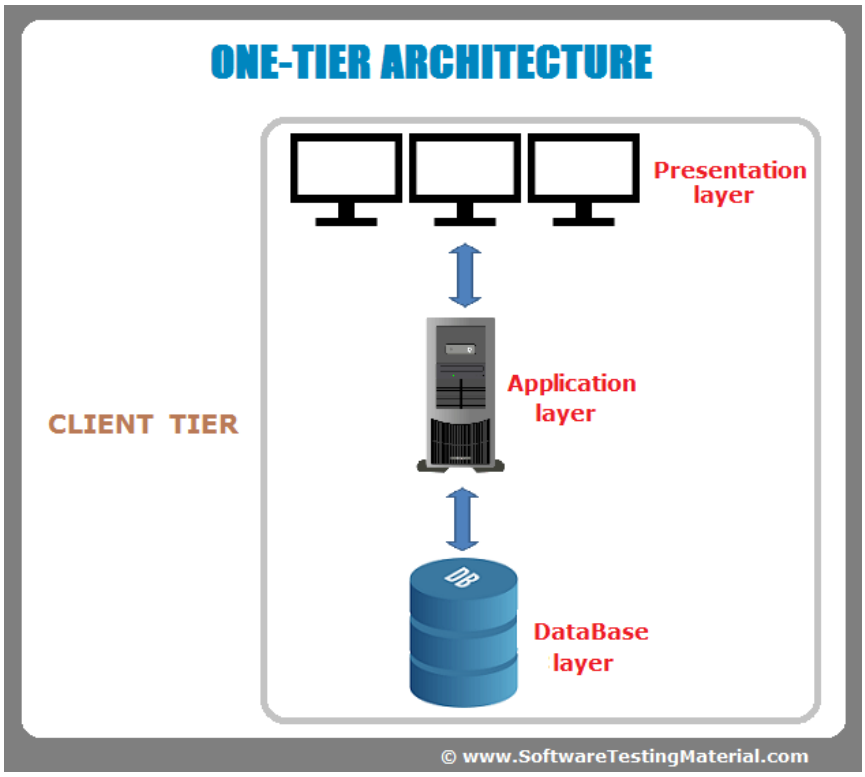
```
with(osoba){
    jedi()
    val vrijemeSpavanja = spavaj()
    return vrijemeSpavanja
}
```

3. Arhitektura *android* aplikacija

3.1. N-slojna arhitektura

Prilikom izrade android aplikacije često se radi s podacima iz različitih izvora. Prije spremanja podataka ili nakon dohvaćanja, podaci se preoblikuju kako bi odgovarali poslovnoj logici aplikacije. Osim poslovne logike, podaci se negdje moraju prikazati ili unijeti od strane korisnika tako da su vidljivi krajnjem korisniku.

Ukoliko sve te korake pišemo na jednom mjestu, dolazi do poteškoća u daljnjem razvoju aplikacije. Prilikom promjene određenog djela aplikacije, programer često mora promijeniti više metoda jer je sve povezano zajedno. Osim potrebe za promjenom velike količine koda, kod je nečitljiv što prouzrokuje teško snalaženje novih programera. Takvu arhitekturu zovemo jednoslojna arhitektura. (Rajkumar, 2017).

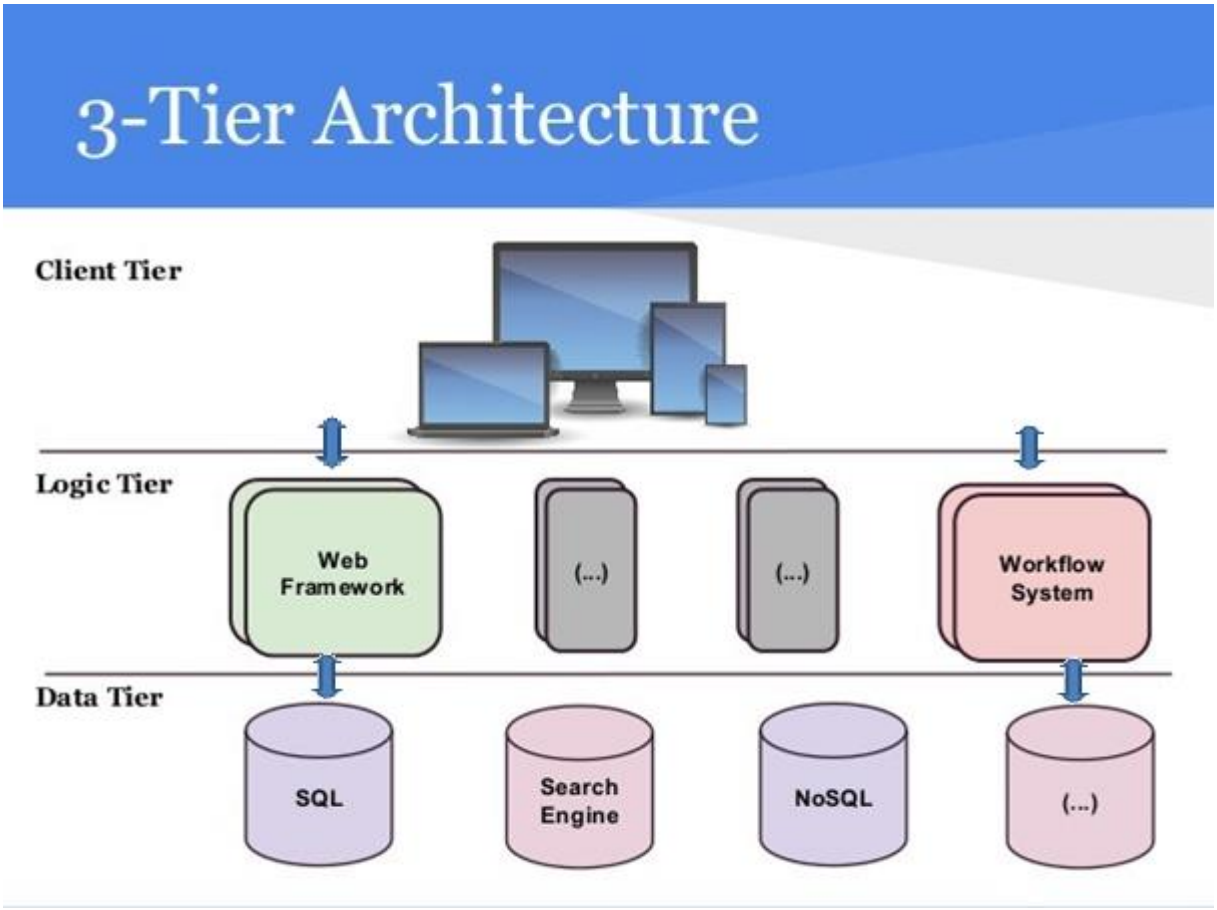


Slika 3. Jednoslojna arhitektura (*Izvor*:* One Tier Architecture, 2017*)

Ako govorimo o profesionalnoj aplikaciji, one sadrže testove za pojedine komponente aplikacije kako bi se dodatno provjerilo da podaci neće doći u neočekivano stanje ili ukoliko je to neizbježno da su takvi slučajevi riješeni od strane programera.

Kako bi se izbjegli i riješili navedeni problemi, potrebno dizajnirati arhitekturu. Osnovna arhitektura bilo koje aplikacije sugerira da se aplikacija podijeli u tri sloja, sloj korisničkog sučelja, sloj poslovne logike i sloj podataka. Takvu arhitekturu zovemo troslojna arhitektura, a sve varijacije postojećih arhitektura u sebi sadrže ovakvu podjelu jer bez njih aplikacija ne može funkcionirati. (Ryan Jentzsch, 2016).

Slika 4 prikazuje protok podataka kroz troslojnu arhitekturu



Slika 4. Troslojna arhitektura (*Izvor: Three Tier Architecture, 2017*)

3.2. Značajke android arhitekture

Većina arhitektura je napravljeno da rade za bilo koju platformu iako postoje specifičnosti unutar svake. Iz tog razloga arhitektura sama po sebi sadrži osnovne komponente dok se specifične komponente nadodaju ovisno o platformi.

Osnovne komponente se uglavnom baziraju na uzrocima dizajna kako bi se olakšalo programiranje i pojednostavilo testiranje.

3.2.1. Injektor zavisnosti

Tijekom programiranja, koristimo objekte različitih klasa koje imaju neku svrhu obrade, spremanja ili slanja podataka. Bilo koji objekt možemo dobiti na dva načina, tako da ga instanciramo unutar klase u kojoj ga želimo koristiti ili da ga instanciramo u nekoj drugoj klasi i proslijedimo preko konstruktora ili neke druge metode.

```
class BusinessLayer{
    var service: ApiService

    constructor() {
        service = ApiService()
    }
}
```

U danom primjeru možemo vidjeti kako smo instancirali objekt koji nam treba unutar konstruktora kao u prvom opisanom načinu. Ako objekt klase „ApiService“ ima parametre konstruktora, sve njegove parametre bi također trebali instancirati unutar konstruktora kako bismo ih prosijedili. Iz tog razloga je preporučeno da se ovakav način instanciranja objekata izbjegava.

Java

```
class BusinessLayer{
    private final ApiService
service;

    BusinessLayer(final
ApiService service){
        this.service =
service;
    }
}
```

Kotlin

```
class BusinessLayer(val
service: ApiService){
}
```

U ovom primjeru objekt koji želimo koristiti dobivamo preko konstruktora i zato nemamo potrebu za instanciranjem njegovih parametara.

Uočite da smo u *Javi* morali napraviti varijablu „service“ i zatim joj dodijeliti vrijednost predanog parametara u konstruktoru. Ista stvar događa se i u *Kotlin* kodu samo što koristimo svojstvo i sve se događa implicitno unutar jedne linije.

Ovakav način instanciranja objekata je preporučljiv kod razvoja aplikacija, međutim ostaje problem gdje taj objekt zapravo nastaje i kako. Prema uzroku dizajna injektora zavisnosti, svi objekti o kojima neka klasa zavisi moraju biti instancirani izvan nje. Mjesto na kojem se takvi objekti instanciraju standardno se zove *object graph* odnosno graf objekata. Takva klasa sadrži metode za instanciranje svih objekata na jednom mjestu i zbog tog razloga svi parametri objekata su također zadovoljeni pozivom tih metoda.

```
class ObjectGraph() {

    private fun provideDatabaseService(): DatabaseService {
        return DatabaseService()
    }

    private fun provideApiService(): ApiService {
        return ApiService()
    }

    private fun provideStudentBusinessLayer() =
            StudentBusinessLayer(provideApiService(),
provideDatabaseService())

    private fun provideProfessorBusinessLayer(apiService:
ApiService) =
            ProfessorBusinessLayer(provideApiService())
}
```

U danom primjeru imamo klasu „ObjectGraph“ koja sadrži metode koje će pružati podatke. Svaka od tih metoda vraća objekt koji želimo dobiti, a ukoliko ima neki parametar, poziva drugu metoda koja pruža taj objekt.

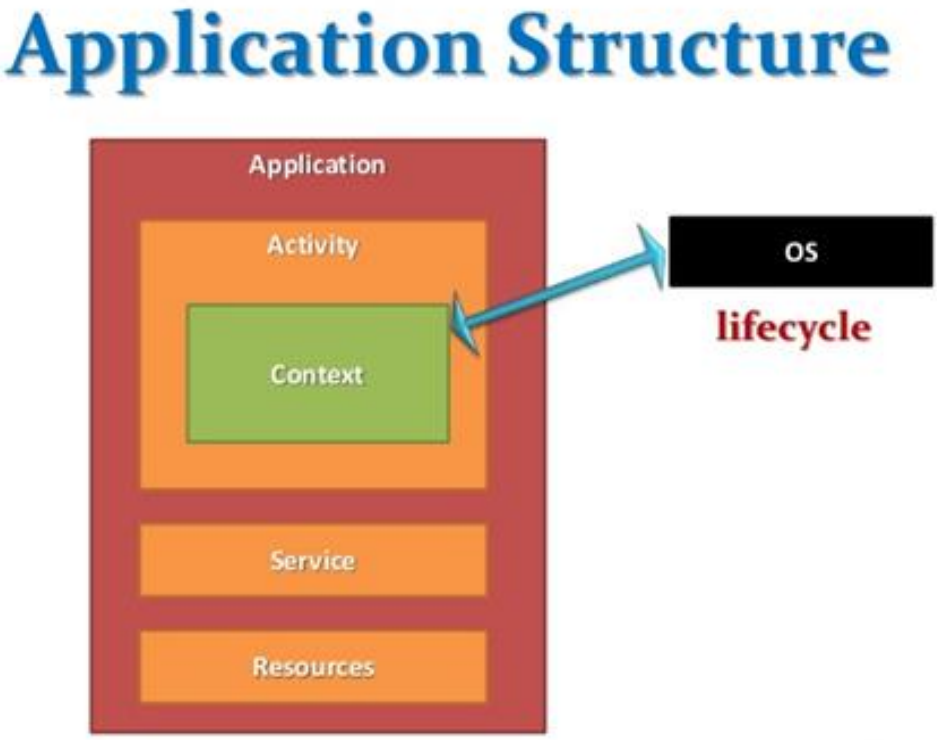
Uočite kako u prve dvije metode imamo naznačen povratni tip podataka i zatim „return“ funkciju koja vraća taj objekt dok u druge dvije metode nemamo naveden ni tip niti one sadrže „return“ funkciju. Ukoliko želimo vratiti objekt povratnog tipa u jednoj liniji, *Kotlin* nam omogućava da stavimo znak pridruživanja „=“ kao što bi smo to napravili kod varijable ili svojstva te zatim pridružimo objekt koji želimo funkciji. U pozadini će povratni tip funkcije biti postavljen na tip objekta kojeg pridružujemo.

3.2.2. Biblioteka *Dagger*

Prateći smjernice za implementaciju uzorka dizajna injektora zavisnosti, izrađuje se klasa *ObjectGraph.* Problem nastaje kada aplikacija postane velika i kada ta klasa ima nekoliko stotina ili tisuća linija. Da bi se to spriječilo, postoje razne biblioteke koje odrađuju taj posao za programera. U programiranju *android* aplikacija, najčešće se koristi biblioteka *Dagger*.

Android aplikacije specifične su jer za vrijeme rada postoje dvije vrste konteksta, aplikacijski kontekst i kontekst aktivnosti. Aplikacijski kontekst postoji cijelo vrijeme dok radi aplikacija i u tom kontekstu želimo pristupati specifičnim android komponentama koje su vezane uz cijelu aplikaciju poput rada sa servisima, rad s bazom podataka, pristupanje *android* resursima i sl.

Kontekst aktivnosti je vezan za jednu aktivnost. *Android* aplikacije podijeljene su na aktivnosti. Svaka aktivnost odnosi se na jednu cjelinu ekrana, npr. ekran za prijavu u sustav. Unutar te aktivnosti možemo imati više ekrana, a svaki od tih ekrana biti će reprezentiran fragmentom. Svaka aktivnost ima svoj kontekst, ali za razliku od aktivnosti aplikacije, kontekst aktivnosti nestaje kada korisnik napusti trenutnu aktivnost. Ukoliko bismo koristili kontekst aktivnosti za pristup bazi podataka ili slanje podataka na server, a u trenutku kada se željena akcija asinkrono odvija u pozadini korisnik napusti trenutnu aktivnost, kontekst bi se izgubio i aplikacija bi se srušila. (Yong Heui Cho, 2015).

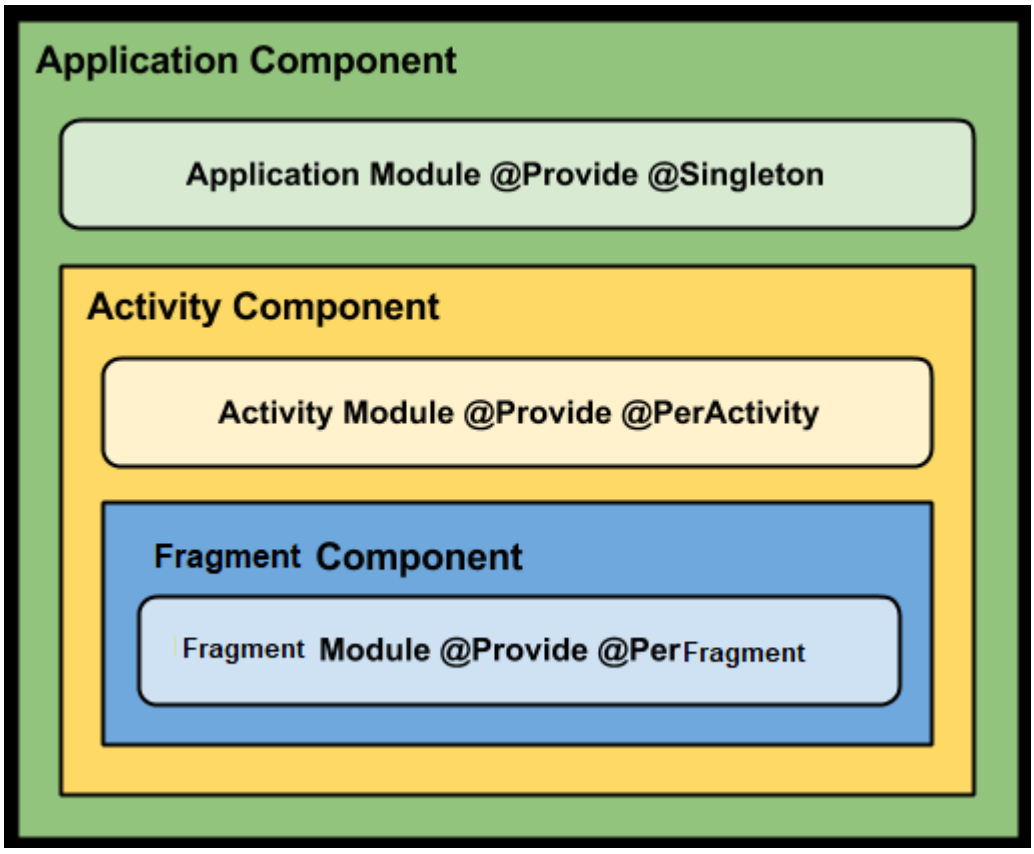


Slika 5. Razine konteksta android aplikacije **(Izvor: Application Structure, 2015)**

Korištenjem biblioteke *Dagger* možemo napraviti vlastitu mrežu grafa objekata koja će biti podijeljena u komponente i module. Komponenta će nam označavati koliko dugo naš objekt treba biti živ.

U našoj arhitekturi koristiti ćemo tri komponente, za aplikaciju, za aktivnost i za fragment. Iako fragment nema svoj kontekst, već koristi kontekst svojeg roditelja, aktivnosti, mi želimo da objekti koje instanciramo za fragment nestanu u isto vrijeme kada nestane i fragment.

Osim komponenti, objekt koji kreiramo moramo svrstati i u modul kako bi se programeri lakše mogli snalaziti u kodu i brže pronaći gdje se instancira koji objekt. Module nazivamo prema svrsi objekta kojeg on instancira. Neke od vrsta modula mogu biti modul podataka, koji radi instancira klase za pohranu i slanje podataka preko servisa ili baze podataka, modul aktivnosti, koji služi za instanciranje objekata različitih aktivnosti i sl.



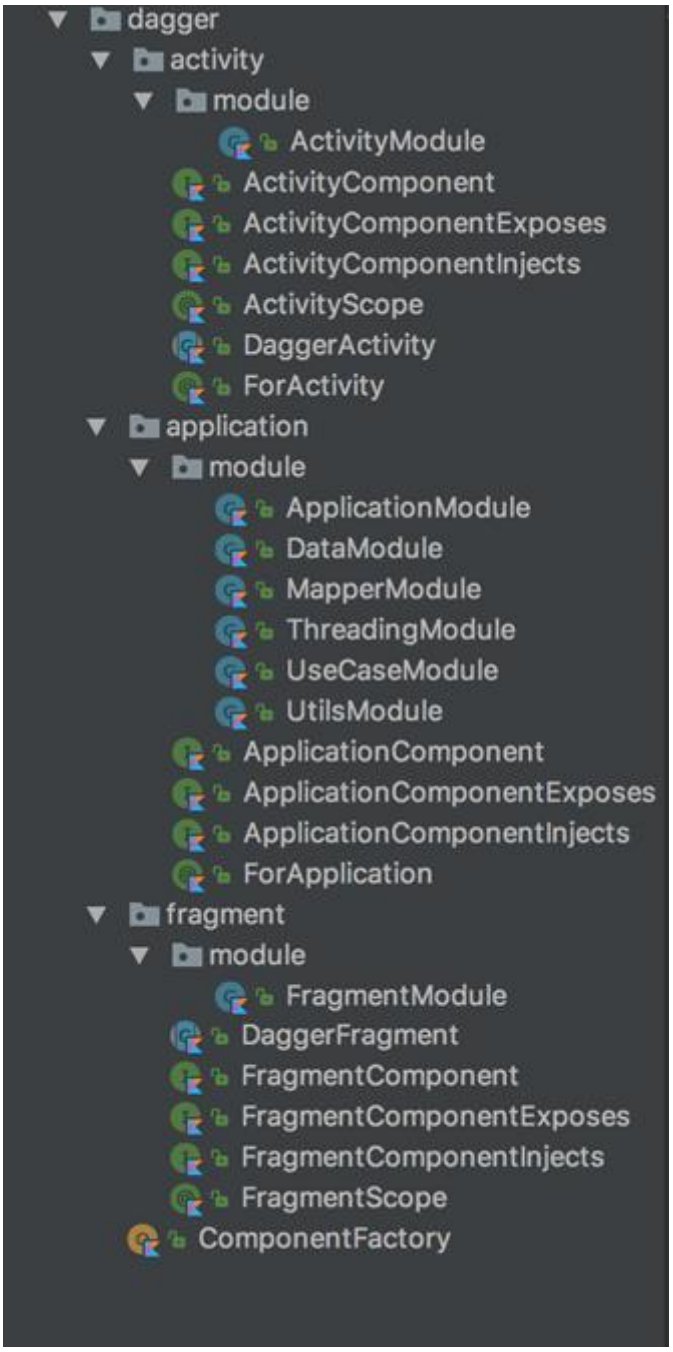
Slika 6. Hijerarhija komponenti i njihovi moduli **(Autorski rad)**

Slika 6. Pokazuje hijerarhiju komponenti. Aplikacijska komponenta je najviša na hijerarhiji i sadrži modul za izradu objekta aplikacije. Uočite anotaciju za instanciranje objekata „Singleton“. To znači da će se svi objekti unutar aplikacijske komponente instancirati samo jednom i taj isti objekt će se koristiti na više mjesta kroz cijelu aplikaciju. Ukoliko želimo objekt koji čita iz baze podataka, nije potrebno da ga svaki puta instanciramo. Na ovaj način uštediti ćemo vrijeme i memoriju prilikom svakog korištenja pojedinog objekta.

Unutar aplikacijske komponente nalazi se komponenta aktivnosti. Komponenta aktivnosti ovisi o aplikacijskoj komponenti. Takva ovisnost je bitna jer bez nje ne bi mogli koristiti objekte koji se instanciraju u aplikacijskoj komponenti unutar aktivnosti. Anotacija za instanciranje objekata modula aktivnosti je „PerActivity“ odnosno svaki puta kada želimo instancirati objekt iz ovog modula on će se instancirati ponovo ukoliko smo promijenili aktivnost ili će se koristiti isti ukoliko smo još uvijek u istoj aktivnosti.

Komponenta fragmenta nalazi se na dnu hijerarhije i ovisi o komponenti aktivnosti. Objekti koji se instanciraju unutar modula fragmenta imaju anotaciju „PerFragment“ i isto kao u prethodnom primjeru, kreirati će se svaki puta kada se promijeni fragment. (Edfora, 2017).

Ova slika prikazuje samo osnovnu hijerarhiju komponenti i njihove module. Dodavanje novih komponenti ovisiti će o aplikaciji koju izrađujemo i njezinoj složenosti. Također, broj modula koji svaka komponenta sadrži bit će veći u realnoj aplikaciji.



Slika 7. Dagger hijerarhija u kodu (*Autorski rad*)

Slika 7 prikazuje kako hijerarhija izgleda u kodu arhitekture koju razvijamo. Možemo vidjeti kako unutar paketa „dagger“ postoje tri paketa za ranije navedene komponente. Komponente aktivnosti i fragmenta trenutno sadrže svaka po jedan modul koji služi za instanciranje objekta aktivnosti odnosno fragmenta. S obzirom na zahtjeve aplikacije moguće je da će nekad biti potrebno dodatni još neki modul, ali većina modula nalazi se unutar aplikacijske komponente.

U aplikacijskoj komponenti, dodatno imamo „dataModule“ odnosno modul podataka koji instancira objekte za rad s podacima, poput rada s bazom podataka. „MapperModule“ služi za instanciranje objekata za mapiranje podataka, odnosno pretvorbu podataka iz jednog modela u drugi. „ThreadingModule“ je modul koji instancira objekte za rad s dretvama, u našem slučaju to su objekti biblioteke *RxJava* koja služi za višedretveno reaktivno programiranje. „UseCaseModule“ je modul koji instancira objekte koji izvršavaju akcije prema odgovarajućim slučajevima korištenja aplikacije. Na kraju se nalazi „UtilsModule“ odnosno modul koji instancira objekte koji sadrže metode koje se često koriste unutar cijele aplikacije. Često se odnose na formatiranje znakovnih nizova, mijenjanje svojstava elemenata na ekranu i sl.

```
@Singleton
@Component(modules = [
                    ApplicationModule::class,
                    ThreadingModule::class,
                    DataModule::class,
                    UseCaseModule::class,
                    MapperModule::class,
                    UtilsModule::class
                    ]
)
interface ApplicationComponent : ApplicationComponentInjects,
ApplicationComponentExposes {

   object Initializer {

        fun init(templateApplication: TemplateApplication): ApplicationComponent =
                DaggerApplicationComponent.builder()
                        .applicationModule(ApplicationModule(templateApplication))
                        .threadingModule(ThreadingModule())
                        .dataModule(DataModule())
                        .useCaseModule(UseCaseModule())
                        .build()

   }
}
```

Primjer iz koda pokazuje kako napisati komponentu pomoću biblioteke *Dagger*. Potrebno je na početku navesti anotacijom „@Singleton“ da ćemo imati samo jednu takvu komponentu u aplikaciji i zatim dodati anotaciju „@Component“ kako bi kompilator znao da se radi o komponenti. Unutar te antoacije potrebno je navesti sve module koje će komponenta sadržavati.

Komponenta se sastoji od sučelja koje ima samo jednu metodu „init“ koja služi za inicijalizaciju komponente u obliku *Builder* uzorka dizajna. Prikazano sučelje nasljeđuje druga dva sučelja, „ApplicationComponentInjects“ i „ApplicationComponentExposes“. Prvo sadrži popis metoda za instanciranje objekata aplikacijske komponente, odnosno instanciranje jedinog objekta aplikacije dok drugo sučelje sadrži popis svih metoda svih modula koje želimo izložiti komponenata niže u hijerarhiji.

```
@Module
class ThreadingModule {

    companion object {

        const val MAIN_SCHEDULER: String = "main_scheduler"

        const val BACKGROUND_SCHEDULER: String = "background_scheduler"
    }

    @Provides
    @Singleton
    @Named(MAIN_SCHEDULER)
    fun provideMainScheduler(): Scheduler = AndroidSchedulers.mainThread()

    @Provides
    @Singleton
    @Named(BACKGROUND_SCHEDULER)
    fun provideBackgroundScheduler(): Scheduler = Schedulers.io()

    interface Exposes {

        @Named(ThreadingModule.MAIN_SCHEDULER)
        fun mainScheduler(): Scheduler

        @Named(ThreadingModule.BACKGROUND_SCHEDULER)
        fun backgroundScheduler(): Scheduler
    }
}
```

Na danom primjeru koda možemo vidjeti kako se definira modul pomoću biblioteke *Dagger*. Svaka klasa koja reprezentira modul mora imati anotaciju „@Module“. Ukoliko želimo napisati metodu odnosno funkciju koja će instancirati neki objekt koji ćemo dalje moći koristiti pomoću injektora zavisnosti, potrebno je tu metodu anotirati s „@Provides“. Anotacija „@Singleton“ određuje da će postojati samo jedna instanca objekta kojeg pružamo unutar cijele aplikacije.

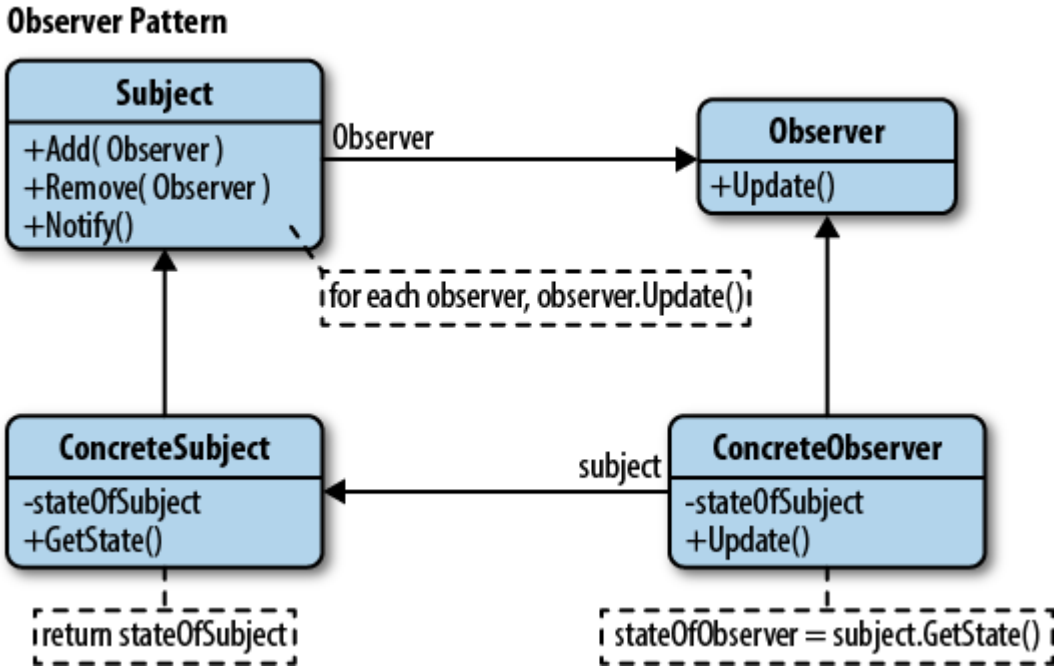
Unutar prikazanog modula imamo dvije metode. Obje metode vraćaju objekt tipa „Scheduler“, odnosno objekt koji unutar biblioteke *RxJava* određuje na kojoj dretvi želimo izvršavati naš kod. Želimo pružati metode za instanciranje dvije vrste takvih objekata, jedan se odnosni na glavnu dretvu, a drugi na pozadinsku dretvu. Budući da su oba objekta istog tipa, biblioteka *Dagger* ne zna koju metodu koristiti za pružanje objekta. Kada postoji takav slučaj, potrebno je koristiti anotaciju „@Named“ unutar koje ćemo navesti ime za naš objekt prema kojem će se objekti istog tipa moći raspoznati.

Na kraju klase nalazi se sučelje „Exposes“ koji služi tome da komponente koje su niže u hijerarhiji imaju pristup metodama koje su navedene u tom sučelju.

3.2.3. Reaktivno programiranje

Tijekom programiranja aplikacije, tijek podataka je često takav da korisnik napravi neku akciju na ekranu koja će promijeniti postojeće podatke u bazi podataka. Ukoliko neki drugi podaci ovise o promijenjenim podacima, potrebno ih je dohvatiti iz baze podataka i prikazati na ekranu. Takav princip rada zahtjeva da se pri svakoj promjeni podataka radi promjena nad drugim podacima ukoliko je to potrebno i zatim radi novi upit nad bazom podataka s promijenjenim podacima.

Jedan način kako bi se izbjegle konstantne provjere da li su se podaci promijenili i radili novi upiti nad bazom podataka da napravimo slušač (eng. *listener*) u obliku *observer* uzorka dizajna. Međutim, kada bi za svaki tok podataka pisali novi uzorak dizajna, programeri bi imali više dodatnog posla i kod bi postao teže čitljiv.



Slika 8. . Observer uzorak dizajna **(Izvor: Observer Pattern, 2018)**

Način kako se rješava takav problem u modernim aplikacijama je da se koristi reaktivno programiranje koje u pozadini radi na principu *observer* uzorka dizajna. Reaktivno programiranje u *Javi* se može postići korištenjem biblioteke *RxJava*, odnosno *RxKotlin*. Budući da se *Kotlin* kod kompilira u *Java byte* kod, te dvije biblioteke su identične.

Reaktivno programiranje radi na način pretplate na izvor podataka. Ukoliko se izvor podataka promijeni, u pretplaćeni protok će se poslati izmijenjeni podaci bez potrebe za provjerom da li su promijenjeni ili eksplicitnog upita prema izvoru podataka.

Prilikom pretplate na izvor podataka možemo specificirati akcije koje će se napraviti nakon što dođu novi podaci, odrediti dretvu na kojoj ćemo slušati za nove podatke i dretvu na koju će se novi podaci poslati. Također, moguće su i akcije kolekcija poput filtriranja i mapiranja podataka. Na kraju, reaktivno programiranje nudi i kontrolu nad ispravljanjem pogrešaka (eng. *Error handling*) kako bi na pravilan način riješili neočekivane situacije.

```
addDisposable(getNewsUseCase.run()
            .subscribeOn(backgroundScheduler)
            .map { it.map { NewsViewModel(it.title, it.description, it.author,
it.imgUrl) } }
            .observeOn(mainThreadScheduler)
            .subscribe(this::onGetNewsSuccess, Throwable::printStackTrace))
}
```

U danom primjeru možemo vidjeti kako izgleda pretplata na izvor podataka pomoću biblioteke *RxJava* odnosno *RxKotlin*. Na početku pozivamo „getNewsUseCase“ koji je zadužen za dohvaćanje vijesti iz izvora podataka. Nakon toga se pomoću metode „subscribeOn“ određuje dretva na kojoj želimo slušati i obraditi podatke prije nego što oni stignu, odnosno određujemo da će to biti pozadinska dretva.

Slijedi metoda „map“ koja mapira podatke iz dobivenog modela podataka u „NewsViewModel“ model podataka. Zadana akcija će se još uvijek odvijati na pozadinskoj dretvi.

Metodom „observeOn“ određujemo da se nakon što se naprave sve ranije akcije želimo prebaciti na glavnu dretvu jer se na njoj moraju prikazivati podaci na ekranu u *android* aplikacijama.

Na kraju stoji metoda „subscribe“ odnosno metoda pretplate. Ona nam govori da dok stignu podaci, želimo ih proslijediti u metodu „onGetNewsSuccess“ odnosno ukoliko dođe do pogreške, želimo ispisati pogrešku na stog.

```
class GetNewsUseCase(private val newsRepository: NewsRepository) :
QueryUseCase<List<Article>> {

   override fun run() = Flowable.just(newsRepository.news())
}
```

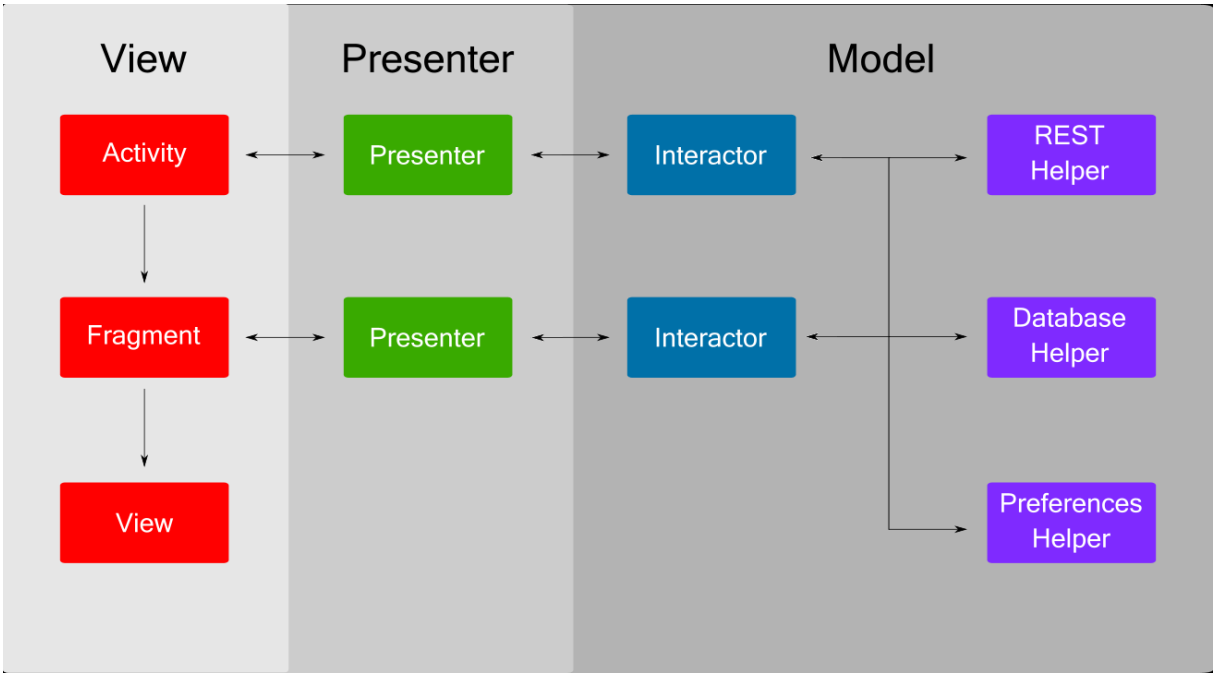
U prethodnom primjeru smo koristili objekt klase „GetNewsUseCase“ kako bi napravili indirektnu vezu s izvorom podataka. U ovom primjeru možemo vidjeti da ta indirektna veza vodi do „newsRepository“ objekta. Taj objekt nam vraća vijesti u obliku liste članaka odnosno „List<Article>“.

Da bi prilikom svake promjene na izvoru podataka mogli dobiti nove objekte odnosno novu listu promijenjenih objekata, potrebno je izvor podataka zamotati u tip „Flowable“ odnosno „Observable“. To je moguće pomoću metode „Flowable.just()“ kao što je prikazano u danom primjeru.

3.3. *MVP(Model-View-Presenter)* arhitektura

Kada se uzmu principi troslojne arhitekture i nadograde s navedenim značajkama *android* arhitekture, dobiva se osnovni način rada svake moderne arhitekture koja se koristi za razvoj *android* aplikacija. Kreiranjem različitih veza između slojeva i dodavanjem dodatnih odgovornosti, može se izgraditi konkretna arhitektura iz one osnovne.

Jedan primjer takve arhitekture je *MVP* koja se zasniva se na tri sloja gdje *view* odgovara sloju korisničkog sučelja, *presenter* odgovara sloju poslovne logike, a *model* odgovara sloju podataka. Ukoliko odemo korak dalje, često se koristi i četvrti sloj *interactor* koji je posrednik između *presenter-a* i *modela*. U tom slučaju *interactor* preuzima odgovornost za poslovnu logiku, a *presenter* prima gotove obrađene podatke. (Yong Heui Cho, 2016),



Slika 9. MVP arhitektura **(Izvor: MVP, 2016)**

Na slici 9 možemo vidjeti princip rada *MVP* arhitekture. Sloj *view* sadrži aktivnosti, fragmente i ostale elemente koji se prikazuju na ekranu. Podaci iz tog sloja mogu biti poslani u *presenter* ukoliko korisnik napravi neku akciju ili mogu doći iz *presenter-a* ukoliko je potrebno prikazati nove podatke na ekranu.

Sloj *presenter* prima unesene podatke iz *view-a* i daje ih *interactor-u* na obradu ili prima obrađene podatke od *interactor-a* i predaje ih u *view* gdje će se oni prikazati.

Podaci unutar *model* sloja mogu doći iz interne baze podataka ili web servisa. Ti podaci se šalju u *interactor* na obradu kako bi odgovarali pojedinim slučajevima korištenja. Suprotno,

već obrađeni podaci mogu doći iz *interactor* sloja i trebaju biti spremljeni u bazu podataka ili poslani preko web servisa.

3.3.1. *Model*

Model sloj reprezentira sloj podataka i kao takav je zadužen za dohvat i slanje podataka iz bilo kojeg izvora. Izvor može biti web servis, interna baza podataka, lokalni *JSON* dokument, *bluetooth* servisi i sl.

Kada sloj primi podatke, oni znaju biti u različitim oblicima ili je moguće da dobivamo suvišne podatke koji nam ne trebaju. Da bi smanjili potrošnju memorije i vrijeme obrade podataka u sloju poslovne logike, potrebno je dobivene podatke mapirati iz modela podataka u takozvane domenske modele koji odgovaraju domeni na koju se odnose.

```
class TemplateClientImpl(private val templateService: TemplateService,
                         private val apiMapper: ApiMapper) : TemplateClient {

    override fun getNews(): Single<List<Article>> {
        return templateService.getNews()
                .map { it -> apiMapper.toArticles(it) }
    }
}
```

Na primjeru iz koda možemo vidjeti kako izgleda poziv web servisa za dohvaćanje podataka o vijestima. Koristi se objekt servisa koji dohvaća podatke i zatim ih mapira u listu članaka. U ovom trenutku imamo podatke spremne za obradu. Podaci se spremaju u repozitorij podataka i zatim se prosljeđuju u sloj poslovne logike *interactor*.

```
class GetNewsUseCase(private val newsRepository: NewsRepository) :
QueryUseCase<List<Article>> {

    override fun run() = Flowable.just(newsRepository.news())
}
```

Dani primjer koda koji je bio prikazan i ranije prikazuje sloj *interactor*. On sadrži klase koje odgovaraju određenom slučaju korištenja, što je u ovom primjeru dohvaćanje vijesti. U manjim aplikacijama se dobiveni podaci direktno prosljeđuju dalje u sloj *presenter*, ali ako imamo potrebu za neku obradu podataka, ona se treba izvršiti u ovom sloju.

3.3.2. *Presenter*

Sloj *presenter* prima obrađene podatke članaka od *interactor-a* i predaje ih u sloj *view* gdje će se prikazati. Budući da postoji obostrana veza između slojeva, tok podataka može ići i u drugom smjeru, odnosno od korisnika prema obradi podataka.

```
addDisposable(getNewsUseCase.run()
            .subscribeOn(backgroundScheduler)
            .map { it.map { NewsViewModel(it.title, it.description, it.author,
it.imgUrl) } }
            .observeOn(mainThreadScheduler)
            .subscribe(this::onGetNewsSuccess, Throwable::printStackTrace))
}
```

U ovom ranije prikazanom primjeru, možemo vidjeti ulogu *presenter-a*. On je taj koji se pretplaćuje na podatke i zatim ih mapira u podatke koji su prilagođeni za sloj *view*.

3.3.3. *View*

View se odnosi na sam ekran, odnosno na ono što korisnik može vidjeti na ekranu. On je odgovoran za korisnikove akcije poput pritiska na gumb, pisanja teksta i sl. Sve podatke koji se prikazuju na ekranu dobiva od *presenter-a* ili od korisnika koji ih unese.

```
class HomeActivity : BaseActivity(), HomeContract.View {

    @Inject
    lateinit var presenter: HomeContract.Presenter

    @Inject
    lateinit var adapter: NewsListAdapter

    lateinit var recyclerView: RecyclerView

    override fun inject(activityComponent: ActivityComponent) {
        activityComponent.inject(this)
    }

    override fun getPresenter(): ScopedPresenter {
        return presenter
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_home)

        recyclerView = activity_home_recycler_view
        initRecyclerView()
        presenter.showNews()
```

```
    }

    private fun initRecyclerView() {
        recyclerView.layoutManager = createLinearLayoutManager()
        recyclerView.adapter = adapter
    }

    private fun createLinearLayoutManager(): LinearLayoutManager {
        return LinearLayoutManager(this, LinearLayoutManager.VERTICAL, false)
    }

    override fun render(newsViewModels: List<NewsViewModel>) {
        adapter.submitList(newsViewModels)
    }
}
```

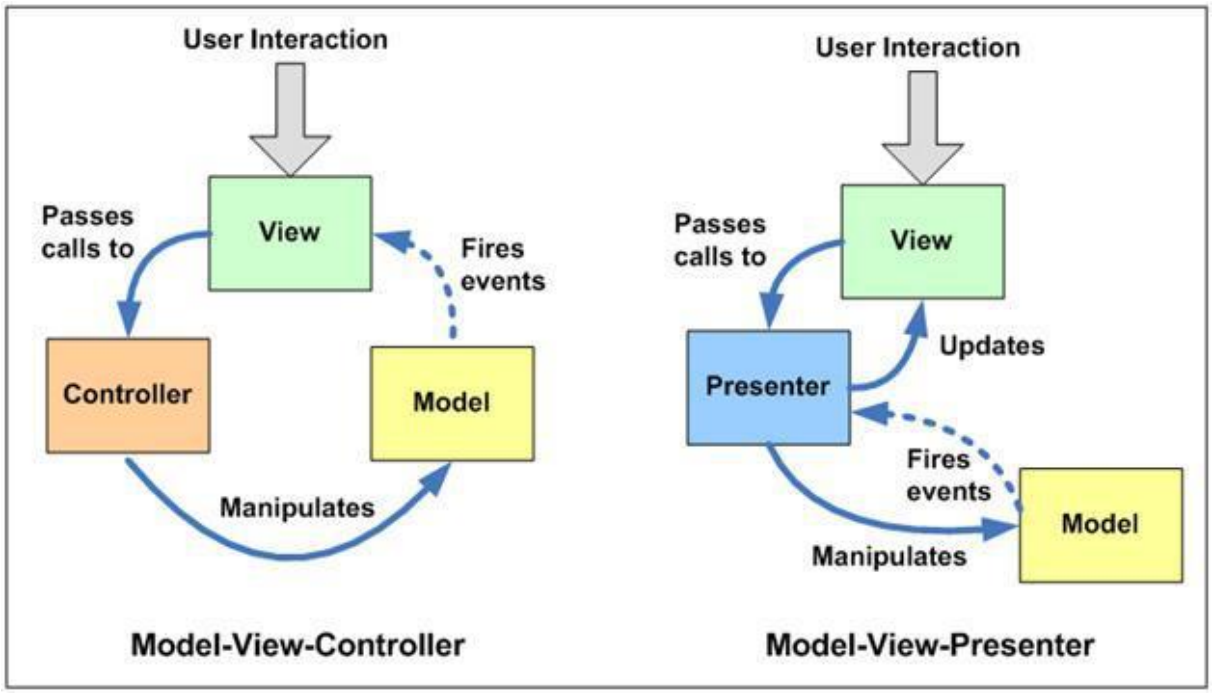
U danom primjeru može se vidjeti kako izgleda kod aktivnosti koja pripada sloju *view*. Većina koda odnosi se na postavljanje elemenata na ekranu poput „recyclerView-a“ i njegovih adaptera. Ono što je bitno za arhitekturu je da *view* sadrži instancu objekta *presenter*. Ukoliko korisnik napravi akciju na ekranu ili želimo postaviti neke početne podatke na ekran, potrebno je pozvati metodu nad tim objektom.

U primjeru možemo vidjeti kako se prilikom kreiranja ekrana poziva metoda „showNews()“ nad objektom „presenter“. Nakon što se vijesti dohvate, *presenter* poziva metodu „render“. *View* nadjačava metodu „render“ u kojoj primljene podatke o vijestima prikazuje na ekranu.

3.4. *MVC(Model-View-Controller)* arhitektura

Jedan od poznatijih uzoraka dizajna za arhitekturu je *MVC*. Najčešće se koristi kod izrade web i desktop aplikacija, dok se za izradu *android* aplikacija koristi samo u nekim dijelovima svijeta.

MVC arhitektura je vrlo slična prethodno objašnjenoj *MVP* arhitekturi, a sadrži i dva ista sloja, *model* i *view* dok je razlika u sloju poslovne logike *controller*. Osim imena, *controller* se razlikuje od *presenter-a* po tome da se u njemu nalazi logika koja odgovara na akcije korisnika. Dok je u *MVP* uzorku dizajna svaki *view* sloj imao svoj *presenter* sloj, u *MVC* uzorku dizajna više *view* slojeva mogu dijeliti zajednički *controller* sloj, a on će odlučivati koji *view* treba trenutno biti aktivan.



Slika 10. Razlike između MVC i MVP arhitektura **(Izvor: MVC vs MVP, 2009)**

Na slici 10 možemo vidjeti glavne razlike između *MVC* i *MVP* arhitekture. U *MVP* arhitekturi, *view* i *model* nisu imali direktnu vezu komunikacije dok je u *MVC* arhitekturi stvar drugačija.

Postoji posebni model prezentacijskoj sloja. *View* se pretplaćuje na promjene tog modela. Kada korisnik napravi akciju na ekranu, akcija se prosljeđuje s *view-a* na *controller* koji zatim radi promjenu nad modelom podataka. Budući da je *view* pretplaćen, on će se indirektnom vezom automatski promijeniti bez da podaci idu nazad preko *controller-a*.

3.4.1. *Model*

Kod *model* sloja ostaje isti kao što je ranije objašnjen u *MVP* arhitekturi. Međutim, njegova svrha više nije da vraća podatke u sloj poslovne logike već da ih predaje direktno na *view*.

3.4.2. *Controller*

Controller ima zadaću odgovoriti na akciju korisnika, po potrebi promijeniti *view* i zatražiti promjenu nad *model-om*. Za razliku od *MVP* arhitekture, *Controller* se ne pretplaćuje na promjene u *model-u*, niti ne predaje nikakve podatke nazad u *view* već samo manipulira trenutnim podacima.

```
override fun changeAuthor(author: String) {
     newsRepository.changeAuthor(author)
}

override fun removeArticle(articleId: Int) {
     newsRepository.removeArticle(articleId)
}


override fun addArticle(article: Article) {
     newsRepository.addArticle(article)
}
```

U danom primjeru možemo vidjeti uobičajene metode za promjenu trenutnih podataka. *Controller* će zatražiti pohranu podataka od *model-a* i izvršiti promjenu prema odgovarajućoj poslovnoj logici ukoliko je to potrebno.

3.4.3. *View*

View se pretplaćuje na podatke koje želi prikazati na ekranu. Na neku korisnikovu akciju, poziva se *controller* koji je odgovoran za nju. *View* ne smije znati što se događa u pozadini, već samo zna da očekuje nove podatke na koje je pretplaćen.

```
class HomeActivity : BaseActivity(), HomeContract.View {

   @Inject
   lateinit var controller: NewsContract.Controller

   @Inject
   lateinit var adapter: NewsListAdapter

   lateinit var recyclerView: RecyclerView

   override fun inject(activityComponent: ActivityComponent) {
       activityComponent.inject(this)
   }
```

```
    override fun getPresenter(): ScopedPresenter {
        return presenter
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_home)

        recyclerView = activity_home_recycler_view
        initRecyclerView()

         addDisposable(getNewsUseCase.run()
                    .subscribeOn(backgroundScheduler)
                    .map { it.map { NewsViewModel(it.title, it.description,
        it.author, it.imgUrl) } }
                    .observeOn(mainThreadScheduler)
                    .subscribe(this::render, Throwable::printStackTrace))
         }
    }

    @OnClick(addButton)
    public fun onAddClicked(){
        controller.addArticle(Article(author.text, article.text, description.text))
    }

    @OnClick(removeButton)
    public fun onRemoveClicked(view : View){
        controller.removeArticle(view.id))
    }

    override fun render(newsViewModels: List<NewsViewModel>) {
        adapter.submitList(newsViewModels)
    }
}
```

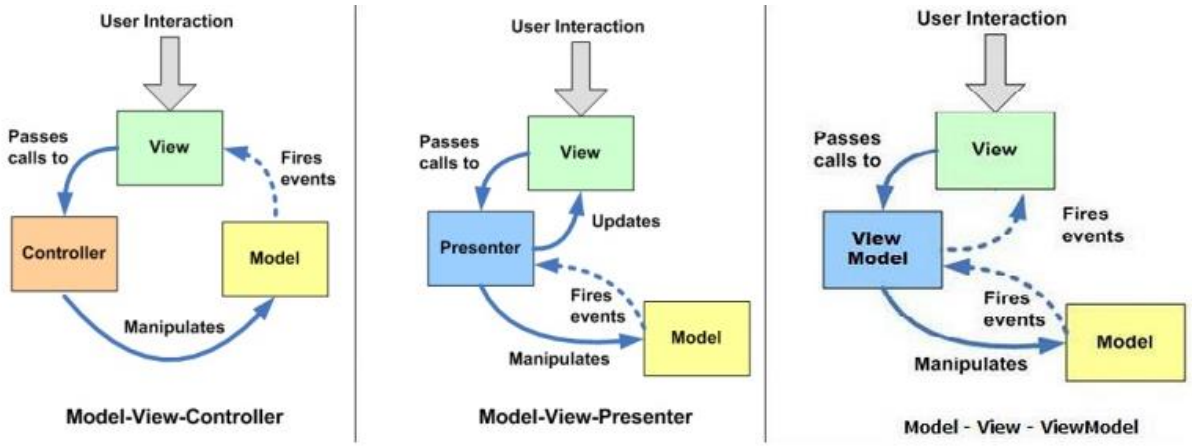
U danom primjeru možemo vidjeti kako izgleda aktivnost unutar arhitekture *MVC*. Aktivnost sadrži instancu objekta „controller“ koji je tipa „NewsContract.Controller“ što znači da će svi ekrani koji prikazuju vijesti u bilo kakvom obliku koristiti taj isti zajednički *controller*.

Unutar metode „onCreate“ umjesto zahtjeva za dohvaćanja podataka možemo vidjeti pretplatu na podatke koja se ranije nalazila u *presenter* sloju. Ukoliko stignu novi podaci, poziva se metoda „render“ koja prikazuje te podatke u listi na ekranu.

Na dnu možemo vidjeti nadjačane metode koje se pozivaju kada korisnik klikne na gumb. U tom slučaju *view* traži od *controller-a* da napravi odgovarajuću akciju.

3.5. *MVVM(Model-View-ViewModel)* arhitektura

MVVM je novija arhitektura u svijetu *android-a* i odnedavno su od strane *Google-a* napravljene pomoćne klase za njenu lakšu implementaciju. Arhitektura je vrlo slična prethodnim dvjema, odnosno sadrži karakteristike od jedne i druge. Prema vezama između slojeva, *MVVM* je identičan *MVP-u*, ali razlikuje se po tome da je sloj gdje se pretplaćuje na podatke *view*, isto kao u *MVC* arhitekturi.



Slika 11. Razlike između MVC, MVP i MVVM arhitektura **(Izvor: MVC vs MVP vs MVVM, 2014)**

3.5.1. *Model*

Kod *model* sloja ostaje isti kao što je ranije objašnjen u *MVP* arhitekturi zbog čega nema potrebe da se ponovo prikazuje.

3.5.2. ViewModel

ViewModel je sličan *presenter* sloju u varijanti *MVP* arhitekture bez *interactora*, odnosno odgovoran je za poslovnu logiku. Razlika je u tome da je *presenter* pretplaćen na izvor podataka i dobivene podatke predaje na *view* dok *viewModel* dohvaća podatke bez pretplate. Na taj način on postaje izvor na koji se treba pretplatiti.

```
class ProjectListViewModel(application: Application, newsRepository:
NewsRepository) : AndroidViewModel(application) {
```

```
  override fun newsFlowable() : Flowable<List<News>>{

      return projectListObservable

  }

   val projectListObservable: Flowable<List<News>>


   init {
       projectListObservable = Flowable.just(newsRepository.news())
   }
}
```

ViewModel prilikom inicijalizacije dohvaća podatke o vijestima iz repozitorija podataka i zatim ih omata kao reaktivni niz podataka kao što je to bilo spomenuto ranije. Nadjačana metoda koja je izložena *view-u*„newsFlowable“ vraća taj reaktivni niz kako bi se na njega moglo pretplatiti.

Cijela klasa nasljeđuje od klase „AndroidViewModel“, odnosno nasljeđuje pomoćnu klasu koju je *Google* napravio. Prilikom programiranja *android* aplikacija, potrebno je paziti na konfiguracijske promjene. To su promjene koje se događaju u posebnim uvjetima kao što je rotiranje ekrana. Prilikom konfiguracijske promjene, aktivnosti se ubijaju i ponovo kreiraju s novim konfiguracijama. Ukoliko dohvaćamo neke početne podatke prilikom pokretanja aktivnosti, svaki put kada se dogodi konfiguracijska promjena ti podaci će se ponovo dohvaćati. Ova pomoćna klasa služi tome da klase koje ju nasljeđuju ne budu uništene prilikom konfiguracijske promjene, odnosno u tom slučaju neće biti potrebno ponovo dohvatiti iste podatke. (Google Developers, 2018).

3.5.3. *View*

Kao i u *MVC* arhitekturi, glavna uloga *view* sloja je da se pretplati na podatke. Razlika je u tome što izvor podataka više nije u *model* već *viewModel* koji sadrži metodu izloženu za pretplatu.

3.6. Usporedba arhitektura

Kada bismo krenuli u izradu *android* aplikacije, morali bismo odabrati neku arhitekturu prema kojoj ćemo raditi. Prilikom odabira, važno je znati koje su prednosti i koji su nedostaci pojedine arhitekture.

3.6.1. Testiranje

Kao što je već ranije napomenuto, testiranje aplikacije je bitno kako bismo izbjegli neugodne situacije u konačnoj verziji. Pisanje testova je brže nego ručno testiranje i jednom kada ih napišemo, ukoliko napravimo promjene, testovi će pokazati da li postoje pogreške prilikom implementacije promjena.

MVP arhitektura je orijentirana na lakoću testiranja. *Presenter* i *model* slojevi nemaju specifične metode vezane za *android* operacijski sustav i iz tog razloga se mogu testirati jediničnim testovima pomoću standardne biblioteke „jUnit“. Ako želimo testirati *view,* potrebno je pisati instrumentacijske testove koji će simulirati korisničke akcije na ekranu. Problem kod *MVP* arhitekture može nastati ukoliko ne koristimo četvrti sloj *interactor* kao pomoćni sloj između *presenter-a* i *model-a*. U tom slučaju je moguće da će se u većim aplikacijama nakupiti previše koda unutar *presentera* što znači ulaganje veće količine vremena na pisanje testnih metoda. *(Kapil Sharma, 2017).*

Kada želimo testirati *MVC* arhitekturu, nailazimo na problem da moramo pisati instrumentacijske testove i za *view* i za *controller*. Budući da je *controller* vezan uz korisničke akcije, to znači da je vezan uz klase koje su specifične za *android* operacijski sustav i kao takve ne mogu biti testirane jediničnim testovima.

MVVM, slično se može testirati slično kao i *MVP*. Razlika je u tome da *MVVM* nema *presenter* zbog čega ima ukupno manje klasa za testiranje. Problem kod *MVVM* arhitekture se nalazi u tome da je teže testirati *view* budući da *xml* datoteke zadužene za izradu izgleda ekrana sadrže podatke u sebi.

Kada se svi podaci uzmu u obzir, može se zaključiti kako je *MVC* arhitektura najteža za testiranje. *MVVM* arhitektura zahtjeva najmanje testnih slučajeva, ali ima problem u testiranju *view* sloja. Iz navedenih razloga, s obzirom na kriterije testiranja, preporučuje se *MVP* arhitektura.

3.6.2. Održavanje

Bitna karakteristika arhitekture je održavanje odnosno da li prilikom izmjene ili dodavanja novih funkcionalnosti moramo mijenjati i stari kod. Održavanje koda ovisi o povezanosti slojeva u arhitekturi, što su više slojevi povezani, to je teže izmijeniti trenutnu ili dodatni novu funkcionalnost. Povezanost znači da će promjena na jednom djelu koda utjecati i na dio koda s kojim je on povezan.

MVP arhitektura ima vrlo dobro odvojene slojeve. Svaki sloj ima svoju svrhu, odnosno slojevi ne ovise jedan o drugome. Kada bismo prilikom kompiliranja izvadili kod iz slojeva *presenter* i *model* i kompilirali prema postavkama kao da *android* komponente uopće ne postoje, kod bi i dalje radio bez pogrešaka.

Budući da u *MVC* arhitekturi postoji povezanost između slojeva *view* i *controller*, to znači da će prilikom promjene u jednom trebati raditi promjene i u drugom. Takva povezanost nam otežava rad na novim funkcionalnostima ili nadogradnji postojećih.

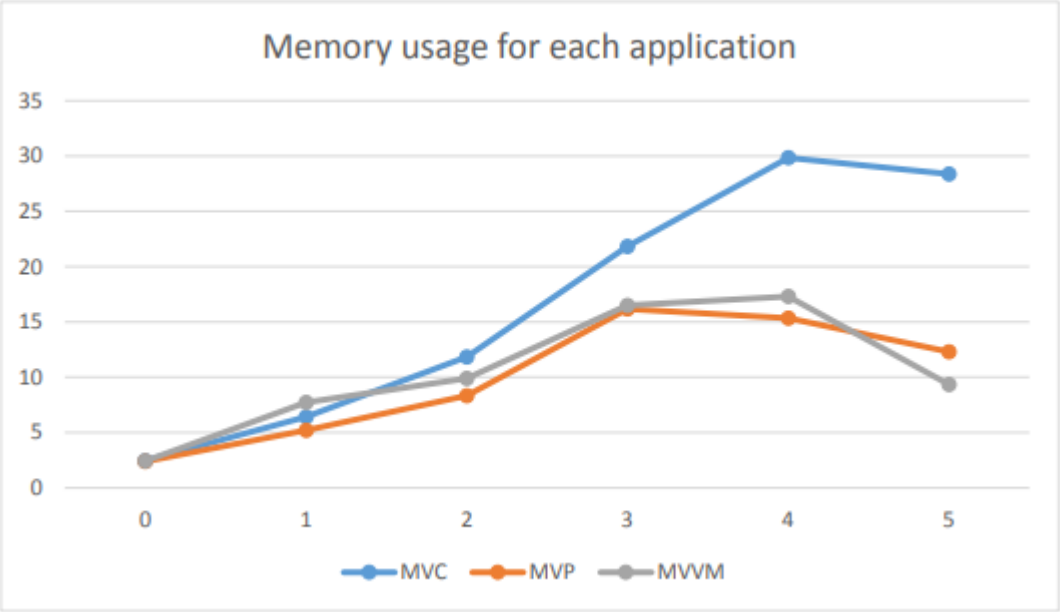
Unutar arhitekture *MVVM* ne postoji povezanost između slojeva, međutim, kao što je bio problem prilikom testiranja, *xml* datoteke u ovoj arhitekturi sadrže podatke. Prilikom dodavanja nove vrste podataka u aplikaciju ili izmjene postojećih, moramo ih ažurirati i u *xml* datotekama.

Zbog navedenih karakteristika, preporučena arhitektura s obzirom na kriterij održavanja koda je *MVP*. *MVVM* arhitektura se nalazi na drugom mjestu jer ažuriranje *xml* datoteka novim podacima je jednostavnije nego ažuriranje koda kroz više slojeva kao što je to potrebno u *MVC* arhitekturi.

3.6.3. Korištenje memorije

Pametni telefoni u današnje vrijeme imaju sve više radne memorije, međutim postoje tržišta gdje se još uvijek koriste stariji modeli i gdje je potrebno paziti na resurse koje aplikacija troši. Neki od takvih tržišta su Indija ili zemlje Afrike. Ako planiramo aplikaciju izbaciti na internacionalno tržište, moramo uzeti u obzir njenu potrošnju resursa.

Svaka arhitektura alocira memoriju prema svojim potrebama i količina alocirane memorije ovisiti će koju arhitekturu smo odabrali. Količinu potrošene memorije pojedine arhitekture možemo mjeriti testiranjem. Korištenje memorije za navedene arhitekture možemo predočiti podacima na slici 12. (Tian Lou, 2016).



Slika 12. . Usporedba alocirane memorije za arhitekture MVC, MVP i MVVM **(Izvor: Tian Lou, 2016, str. 36)**

Na danom grafu, x os se odnosi na broj minuta rada aplikacije, dok se y os odnosi na količinu alocirane memorije u MB. U prvih nekoliko minuta rada aplikacije, sve arhitekture imaju podjednaku potrošnju memorije, ali nakon dužeg rada *MVC* počinje trošiti više dok *MVP* i *MVVM* ostaju podjednake. Međutim, ovaj graf ne pokazuje što se događa kada promijenimo konfiguraciju, npr. rotiramo ekran uređaja.

Ranije je spomenuto kako arhitektura *MVVM* koristi pomoćne klase koje su razvijene od strane *Google-a* koje pomažu da se prilikom promjene konfiguracije podaci ne izgube. Ako uzmemo u obzir navedenu činjenicu, arhitektura *MVVM* bi trošila najmanje memorije u realnom slučaju korištenja aplikacije

Iz navedenih razloga, s obzirom na potrošnju memorije, preporučuje se korištenje *MVVM* arhitekture.

4. Zaključak

Otkako je *Google* najavio da će pružati podršku u razvoju i održavanju jednom od novijih jezika *Kotlin,* njegova popularnost je naglo porasla. *Kotlin* je postao preporučeni jezik prilikom izrade *android* aplikacija i sve više novijih biblioteka podržava razvoj u *Kotlin-u.*

Svojim prednostima u programiranju naspram programskog jezika *Jave*, privlači sve više programera unutar drugih domena programiranja koji rade na drugim platformama. Zbog naglog rasta popularnosti, sve je veća potreba za predloškom po kojem bi programeri trebali raditi prilikom izrade aplikacija.

Predložak izrade aplikacija drugim riječima možemo nazvati i arhitektura jer se arhitektura aplikacije postavlja na početku, prije dodavanja novih funkcionalnosti. Arhitektura aplikacije određuje način na koji ćemo pisati aplikaciju, kako ćemo odvojiti slojeve u aplikaciji, koliko lako će biti testirati aplikaciju ili kako će se voditi njeno održavanje.

Prije nego se krene u izradu aplikacije, programski arhitekt zajedno s programerima mora diskutirati i odabrati arhitekturu koja zadovoljava svojstva i funkcionalnosti aplikacije koja će se izrađivati. Iz tog razloga potrebno je znati prednosti i nedostatke između najpopularnijih i najkorištenijih arhitektura.

U ovome radu uspoređivale su se tri arhitekture, *MVP*, *MVC* i *MVVM,* svaka sa svojim prednostima i nedostacima. Arhitektura *MVC* se pokazala najlošijom prema svim kriterijima, dok su arhitekture *MVP* i *MVVM* podjednake.

Ostaje problem kako se odlučiti između dvije podjednake arhitekture. Odgovor se nalazi u kombinaciji arhitektura, odnosno kako *MVVM* zadržava podatke prilikom promjene konfiguracije, rješenje je da umjesto podataka zadržava čitav *presenter* sloj koji će imati podatke pohranjene u sebi. Na taj način možemo zadržati najbolje karakteristike od obje arhitekture.

5. Popis slika

Slika 1. Kotlin za razvoj više vrsta aplikacija 2
Slika 2. Dijagram korištenja funkcija konteksta 13
Slika 3. Jednoslojna arhitektura 16
Slika 4. Troslojna arhitektura 17
Slika 5. Razine konteksta android aplikacije 21
Slika 6. Hijerarhija komponenti i njihovi moduli 22
Slika 7. Dagger hijerarhija u kodu 23
Slika 8. . Observer uzorak dizajna 27
Slika 9. MVP arhitektura 29
Slika 10. Razlike između MVC i MVP arhitektura 33
Slika 11. Razlike između MVC, MVP i MVVM arhitektura 36
Slika 12. Usporedba alocirane memorije za arhitekture MVC, MVP i MVVM 40

6. Literatura

[1] Britt Ballard, Back to Basics: Anonymous Functions and Closures https://robots.thoughtbot.com/back-to-basics-anonymous-functions-and-closures , dostupno 10.9.2018

[2] Martin Heller, What is Kotlin? The Java alternative explained, https://www.infoworld.com/article/3224868/java/what-is-kotlin-the-java-alternative-explained.html , dostupno 10.9.2018

[3] Adit Microsys, Modern Web Development With Kotlin Programming Language, https://medium.com/@microsysadit/modern-web-development-with-kotlin-programming-language-92651af81266 , dostupno 10.9.2018

[4] Kotlin, Classes and Inheritance, https://kotlinlang.org/docs/reference/classes.html, dostupno 10.9.2018

[5] 김태수, Mastering Kotlin standard functions: run, with, let, also and apply, https://medium.com/@kimtaesoo188/mastering-kotlin-standard-functions-run-with-let-also-and-apply-7fb4492db246 , dostupno 10.9.2018

[6] Ryan Jentzsch, What is the role of a web service in a three tier architecture?, https://www.quora.com/What-is-the-role-of-a-web-service-in-a-three-tier-architecture , dostupno 10.9.2018

[7] Rajkumar, Software Architecture: One-Tier, Two-Tier, Three Tier, N Tier, https://www.softwaretestingmaterial.com/software-architecture/ , dostupno 10.9.2018

[8] Edfora, MVP, Dagger—The Memory Guards of Android, https://blog.edfora.com/mvp-dagger-the-memory-guards-of-android-3b439b8b8159 , dostupno 10.9.2018

[9] Yong Heui Cho, Android – Message, https://www.slideshare.net/YongHeuiCho/android-message , dostupno 10.9.2018

[10] Chris Ripple, Applying MVP in Android, https://www.grapecity.com/en/blogs/applying-mvp-in-android , dostupno 10.9.2018

[11] Google Developers, Handle configuration changes, https://developer.android.com/guide/topics/resources/runtime-changes , dostupno 10.9.2018

[12] Tian Lou, A comparison of Android Native App Architecture MVC, MVP and MVVM, https://pure.tue.nl/ws/files/48628529/Lou_2016.pdf , dostupno 10.9.2018

[13] Kapil Sharma, Cheat Sheet for MVC, MVP, MVVM Android Design Patterns, https://medium.com/@kapil.sharma91812/cheat-sheet-for-mvc-mvp-mvvm-android-design-patterns-b77681ba54b4 , dostupno 10.9.2018

[14] Kotlin - Build Any Type Of Application [slika], https://images.xenonstack.com/blog/Build-Any-Type-Of-Application-Using-Kotlin.jpg , dostupno 10.9.2018

[15] Choosing Standard Function [slika], https://cdn-images-1.medium.com/max/800/1*pLNnrvgvmG6Mdi0Yw3mdPQ.png , dostupno 10.9.2018

[16] One Tier Architecture [slika], https://i0.wp.com/www.softwaretestingmaterial.com/wp-content/uploads/2016/06/one-tier-software-architecture.png?w=629&ssl=1 , dostupno 10.9.2018

[17] Three Tier Architecture [slika], https://i2.wp.com/www.softwaretestingmaterial.com/wp-content/uploads/2016/06/three-tier-software-architecture.png?w=848&ssl=1 , dostupno 10.9.2018

[18] Application Structure [slika], https://image.slidesharecdn.com/5-151008064036-lva1-app6892/95/android-message-3-638.jpg?cb=1477993052 , dostupno 10.9.2018

[19] Observer Pattern [slika], https://www.safaribooksonline.com/library/view/learning-javascript-design/9781449334840/httpatomoreillycomsourceoreillyimages1547801.png , dostupno 10.9.2018

[20] MVP [slika], https://gccontent.blob.core.windows.net/gccontent/blogs/legacy/xuni/2016/05/MVP1.png , dostupno 10.9.2018

[21] MVC vs MVP [slika], https://nirajrules.files.wordpress.com/2009/07/mvcmvp2.jpg , dostupno 10.9.2018

[22] MVC vs MVP vs MVVM [slika], http://2.bp.blogspot.com/-nNwsxDAC3Og/VFOEMQDn4mI/AAAAAAAABDA/G7r3_tLAODM/s1600/image001-757983.png , dostupno 10.9.2018