Primjena paradigme čiste arhitekture pri razvoju mobilnih aplikacija za iOS

Martan, Dino

Master's thesis / Diplomski rad

2022

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:211:356524

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

Download date / Datum preuzimanja: **2024-03-28**

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

Dino Martan

Primjena paradigme čiste arhitekture pri razvoju mobilnih aplikacija za iOS

DIPLOMSKI RAD

Varaždin, 2022.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Dino Martan

JMBAG: 0016123598

Studij: Informacijsko i programsko inženjerstvo

Primjena paradigme čiste arhitekture pri razvoju mobilnih aplikacija za iOS

DIPLOMSKI RAD

Mentor:

Izv. prof. dr. sc. Zlatko Stapić

Varaždin, rujan 2022.

Dino Martan

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

Mobilne aplikacije postaju sve veće i kompleksnije. Razvoj treba biti brz, a nerijetko je potrebno raditi izmjene postojećih značajki aplikacija. Kako bi se to omogućilo, potrebno je razdvojiti komponente (korisničko sučelje, navigacija i poslovna logika). Te komponente je zatim potrebno povezati. Povezivanjem tih komponenti bave se arhitekture softvera. Arhitekture utječu na kvalitetu i održivost kôda, a time i konačnog proizvoda, odnosno aplikacije. Jedna od arhitektura je čista arhitektura. Cilj ovog rada je prikazati paradigmu čiste arhitekture i njenu primjenu kod razvoja iOS aplikacija što će biti postignuto teorijskom podlogom u prvom dijelu rada te praktičnim primjerom u drugom dijelu rada.

Ključne riječi: čista arhitektura; mobile; iOS; swift; razvoj mobilnih aplikacija

Sadržaj

Sadržaj v
1. Uvod 1
2. Metode i tehnike rada 3
3. Arhitektura 4
3.1. SOLID principi 5
3.2. Dobra arhitektura 14
3.3. Čista arhitektura 15
4. iOS arhitekture 18
4.1. MVC arhitektura 19
4.2. MVP arhitektura 20
4.3. MVVM arhitektura 21
4.4. VIPER arhitektura 22
4.5. VIP arhitektura 23
4.6. Testiranje u iOS arhitekturama 25
4.6.1. MVC jedinični testovi 30
4.6.2. MVP jedinični testovi 32
4.6.3. MVVM jedinični testovi 37
5. Usporedba iOS arhitektura 42
6. Praktičan dio 45
6.1. Baza podataka 57
6.2. Implementacija 58
6.2.1. Servisi 63
6.2.2. Repozitoriji 65
6.2.3. Navigacija 67
6.2.4. Scene 70

6.3. Jedinični testovi....80
7. Zaključak....94
Popis literature....95
Popis slika....97
Popis tablica....99
Popis programskih kôdova....100
Prilozi....102

1. Uvod

IT sektor je brzorastući i stalno se mijenja. Dolaze nove tehnologije, *hardware* iz godine u godinu postaje jači i brži i IT postaje nezamjenjiv dio u mnogim aspektima poslovanja. Primjerice, ERP sustavi u poduzećima pružaju veliku brzinu i jednostavnost kod praćenja informacija unutar poduzeća. Sve to uzrokuje neprestan rast ponude, ali i potražnje. Mobilni uređaji su u posljednjih desetak godina sveprisutni i pojedinci se sve više oslanjaju na njih. Kako za njima raste potražnja, tako nastaje i prostor za sve većim brojem mobilnih aplikacija putem kojih poduzeća mogu doprijeti do sve većeg broja ljudi. Sva ta brzina i nagli rast dovode do potrebe za sve bržom isporukom mobilnih aplikacija i tu je potrebno obratiti pozornost na kvalitetu koja vrlo često pada s brzinom.

Dvije su velike platforme za izradu mobilnih aplikacija: Android i iOS. U ovom radu će pažnja biti usmjerena na iOS platformu. Razvoj iOS aplikacija, ali i mobilnih aplikacija generalno, predstavlja specifičan izazov. S obzirom na veličinu ekrana mobilnih uređaja, nije moguće na jednom mjestu prikazati veliku količinu informacija. To često dovodi do potrebe za više ekrana. Većina aplikacija služi za interakciju s korisnikom. Na početku se prikaže neki sadržaj, zatim korisnik odradi neku akciju (najčešće dodir gumba) te aplikacija putem mrežnog zahtjeva dohvati nove podatke koje opet prikaže korisniku. Naručitelji aplikacija često traže da te interakcije budu „žive“, odnosno da postoje animacije kako korisničko iskustvo ne bi bilo dosadno. To je tek pojednostavljen primjer korištenja aplikacije, ali i iz njega se mogu izdvojiti neke osnovne komponente koje su potrebne za izradu iOS aplikacija. Za početak, potrebno je korisničko sučelje koje se sastoji od ekrana odnosno elemenata na ekranu (gumb, tekst, slika…). Animacije mogu biti dio korisničkog sučelja. Spomenuto je i da je potrebno više ekrana, dakle još jedna komponenta je navigacija između ekrana. Kako bi aplikacija dohvatila podatke preko mreže, potrebna je i mrežna komponenta. Naravno, ti podatci ne mogu biti besmisleni za što se brine komponenta poslovne logike koja najčešće stoji između korisničkog sučelja i mrežne komponente.

Iz ovakvog jednostavnog, teoretskog primjera aplikacije, izdvojene su četiri komponente: korisničko sučelje, navigacija, poslovna logika i mrežna logika. No, ako malo bolje razmislimo, ove komponente se mogu primijeniti i na većinu postojećih aplikacija. Više komponenti koje međusobno ovise jedna o drugoj nameću vrlo važno pitanje: na koji način je potrebno povezati te komponente? Zadaću povezivanja osnovnih komponenti ima arhitektura sustava. Jedna od arhitektura je čista arhitektura. Koncept čiste arhitekture nastoji što je više moguće razdvojiti pojedine komponente kako bi razvoj, a posebice kasnije održavanje, bilo jednostavno i predvidivo.

Ovaj se rad sastoji od ukupno četiri glavna poglavlja u razradi teme. U prvom poglavlju bit će napravljen osvrt na sam pojam arhitektura, što neku arhitekturu čini dobrom, a što ne te će biti predstavljen pojam čiste arhitekture. U drugom poglavlju će biti napravljen pregled najkorištenijih arhitektura i razvoju iOS aplikacija te osvrt na testiranja unutar tih arhitektura. U trećem poglavlju će biti prikazana usporedba arhitektura iz prethodnog poglavlja te njihove prednosti i mane. Četvrto poglavlje obuhvaća praktični dio rada. Praktični dio je iOS mobilna aplikacija „*Connect*“ koja će biti razvijena koristeći čistu arhitekturu. Kroz primjer će biti dane upute za postavljanja arhitekture te njena upotreba. Uz samu aplikaciju će biti izrađeni i jedinični testovi koji pokrivaju glavne dijelove aplikacije, odnosno arhitekture.

2. Metode i tehnike rada

Ovaj diplomski rad podijeljen je u dva dijela: teorijski i praktični dio. Za izradu teorijskog dijela rada je korišteno nekoliko metoda. S obzirom na to da je pojam arhitekture softvera poprilično širok, metodom klasifikacije će biti podijeljen u više dijelova. Na temelju više izvora bit će napravljen pregled različitih arhitektura korištenjem analize i sinteze. Svaka od arhitektura, zajedno sa ostalim temeljnim pojmovima rada, bit će opisana deskriptivnom metodom. S obzirom na to da postoji više različitih arhitektura koje se koriste kod razvoja iOS aplikacija, važno je napraviti usporedbu između njih. Usporedba će biti provedena koristeći komparativnu analizu. S obzirom na to da ovaj rad prikazuje dio stavova, zaključaka i spoznaja drugih autora, koristi se i metoda kompilacije. Sve navedene metode odabrane su s ciljem opisivanja i predstavljanja teoretskog predstavljanja pojma arhitekture te pregled arhitektura koje se koriste u izradi iOS aplikacija.

Glavni dio literature je knjiga R. C. Martina „Clean Architecture“ s obzirom na to da autor u toj knjizi predstavlja čistu arhitekturu koja je tema ovog rada. Uz to, korišteni su i ostali izvori (pretežito internet članci) te službene dokumentacije pojedinih platformi.

Za izradu praktičnog dijela korišten je programski jezik *Swift* te *Xcode* programsko okruženje (13.4.1). Za testiranje je korišten simulator (koji je u sklopu *Xcode* okruženja) te fizički uređaji iPhone 7 i iPhone 13. Za verzioniranje je korišten GitHub repozitorij dostupan na https://github.com/dinoMartan/diplomskiRad te GitHub Desktop aplikacija. Za izradu *backend* dijela potrebnog za funkcioniranje aplikacije korišten je *Firebase*. U praktičnom dijelu korištene su biblioteke:

- *SnapKit* [1] – biblioteka za postavljanje ograničenja između elemenata korisničkog sučelja
- *KeychainSwift* [2] – biblioteka za spremanje i dohvaćanje podataka u i iz *Keychaina*
- *Kingfisher* [3] – biblioteka za spremanje i pred memoriranje fotografija s Interneta
- *IQKeyboardManager* [4] – biblioteka za automatsko prilagođavanje ekrana tipkovnici
- *WSTagsField* [5] – biblioteka koja sadrži tekstualno polje u kojem je omogućeno dodavanje *tagova*
- *Firebase* [6]*, FirebaseAuth* [7]*, FirebaseFirestore* [8], *FirebaseFirestoreSwift* [9] i *FirebaseStorage* [10] – biblioteke za pristup različitim značajkama *Firebasea*

3. Arhitektura

U modernom svijetu, stalno smo okruženi građevinama. Od najmanjih kao što je kiosk ili manjih do velikih nebodera. Znamo da je većina građevina izgrađena od betona, stakla i čelika. No, činjenica da su građene od gotovo istih bazičnih komponenti ne znači da izgledaju jednako i da imaju istu namjenu. Neke zgrade imali prekrasan vanjski izgled, neke su izgrađene s naumom da mogu podnijeti jake potrese, treće pak mogu biti više od ostalih. Sve te razlike najčešće pripisujemo arhitekturi građevine.

Sličnu paralelu možemo povući i s izgradnjom odnosno razvojem softvera. No, u ovom slučaju, nemamo više bazičnih elemenata kao što su beton i čelik, već je najbazičniji element opet softver. Iz tog razloga, prema C. Martinu [11], softver može kreirati bazične elemente po želji. Problem nastaje u tome što je potrebno pažljivo kreirati bazične elemente te ih zatim pažljivo koristiti u daljnjem razvoju.

Prema C. Martinu [11], glavna zadaća softvera je da bude lako izmjenjiv. Iako se lako prepustiti tezi da je glavna zadaća softvera da izvršava zadani zadatak, to može biti problematično na duge staze. Nerijetko se desi da naručitelj softvera u početku zaželi brz i funkcionalan softver i nakon nekog vremena počne dodavati ili izbacivati funkcionalnosti. Ako developer ne teži tome da softver bude izmjenjiv, to dovodi do dugih i mukotrpnih izmjena koje koštaju puno resursa (vremena i novaca).

Do sad su spominjani bazični elementi. U građevini, to mogu biti cigle dok u razvoju softvera to mogu biti klase. Autor navodi kako sama arhitektura nije dovoljna jer, u slučaju da cigle, odnosno bazični elementi nisu ispravni, zgrada ili softver neće ispasti kako je planirano. Iz tog je razloga potrebno uvesti i neka pravila odnosno principe za izradu bazičnih elemenata prije pravila za izradu arhitektura. Ti se principi u literaturi često označavaju kao principi dizajna. Primjerice, Chauhan [13] navodi četiri takva principa: SOLID (zapravo skup principa), DRY (eng. *Don't Repeat Yourself*), KISS (eng. *Keep it simple, Stupid!*) te YAGNI (eng. *You ain't gonna need it*) principe. Martins [14] pak navodi deset principa koji su predstavljeni više kao pravila i upute (primjerice Povećaj ponovno korištenje). C. Martin [11] pak opet navodi SOLID principe. SOLID principi se najčešće spominju u literaturi principa dizajna softvera. Iz tog razloga će u ovom radu biti korišteni kao temelj izrade bazičnih elemenata. Nakon što su bazični elementi izrađeni po nekim pravilima i možemo računati na njihovu ispravnost i funkcionalnost, te elemente možemo međusobno povezati što je posao arhitekture.

Prema D. Garlanu [12], arhitektura je zapravo most između zahtjeva i same implementacije. S obzirom na to, autor navodi da arhitektura direktno utječe na izgled

implementacije odnosno razvoj softvera. Prije svega, utječe na jasnoću odnosno razumijevanje sustava. To postiže apstrakcijom tih sustava u oblik koji je razumljiv i jednostavan. Arhitektura također uvelike utječe i na ponovno korištenje elemenata. Tu se može raditi o jednostavnim klasa, ali i o podsustavima ili bibliotekama. Što je sustav veći, to je ponovno korištenje značajnije jer se smanjuje potreba za velikim izmjenama i pojednostavljuje se testiranje. Time arhitektura utječe i na evoluciju sustava. Arhitektura također daje nacrt za međusobnu povezanost različitih komponenti sustava, određuje granice apstrakcije i definira raspodjelu odgovornosti. Arhitektura utječe i na neke druge aspekte razvoja koji nisu nužno vezani uz sam programski kôd. S obzirom na to da je arhitektura zapravo skup pravila, ona omogućava sistematičnu analizu pridržavanja tih pravila. Ona može ukazivati na potencijalne probleme te pokazati je li ispravna za zadani projekt prije početka same implementacije.

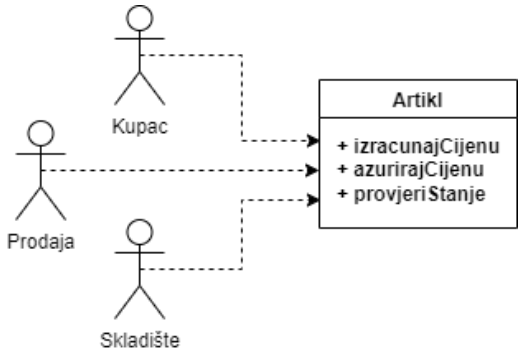
Arhitektura je vrlo važan dio razvoja softvera, ali sama arhitektura nije dovoljna. Potrebno je da svi elementi unutar arhitekture budu ispravni i slijede neka pravila. U nastavku će biti napravljen pregled SOLID principa koji definiraju kako pojedini elementi trebaju izgledati. Nakon toga će biti dan pregled karakteristika prema kojima je moguće odrediti je li neka arhitektura dobra. Na kraju će biti dan teorijski pregled jedne od mnogobrojnih arhitektura, čiste arhitekture.

3.1. SOLID principi

Bazični elementi u softveru su kreirani iz softvera te je zadaća developera da kreira svoje bazične elemente. U kreiranju tih elemenata (klasa) potrebno se držati nekih principa i tu uvelike pomažu SOLID principi spomenuti ranije. Oni omogućavaju promjene, jednostavni su za razumjeti i mogu se primjenjivati na različitim sustavima. U nastavku će biti napravljen pregled SOLID principa te primjeri u kojima se poštuju ili ne poštuju ti principi.

1) SRP (*Single Responsibility Principle*) – svaka klasa može imati samo jedan razlog za promjenu, odnosno klasa treba biti odgovorna za samo jednu grupu korisnika.

   Kako bi se razumjelo što to točno znači, najlakše je uzeti primjer u kojem se princip ne poštuje. Na slici 1 je prikazan upravo takav primjer.



Slika 1 Single Responsibility Principle

Na ovom primjeru vidimo da klasa `Artikl` ima tri metode, `izracunajCijenu()` koju koriste kupci, `azurirajCijenu()` koju koristi prodaja i `provjeriStanje()` koju koristi skladište. Ovo je ne poštivanje SRP-a jer se može desiti da je potrebno da kupac također može provjeriti stanje, ali s nekim drugim parametrima. To bi dovelo do izmjene te metode kako bi odgovaralo kupcima, ali u tom slučaju skladište gubi svoje informacije. Jedno od rješenja je kreiranje zasebnih klasa, primjerice `KupacArtikl`, `ProdajaArtikl` i `SkladisteArtikl`.

Ovaj se princip vrlo često tumači kao „klasa treba imati samo jednu odgovornost“. To je krivo tumačenje ovog principa i to se tumačenje može primijeniti na metode unutar klase, ali ne i na samu klasu. Prema K. Arori [15], ovaj princip zapravo govori da klasa treba biti odgovorna za jednu operaciju.

2) OCP (*Open-Closed Principle*) – dodavanje ponašanja klase mora biti moguće dodavanjem novog kôda, a ne izmjenom postojećeg.

Ovaj princip se u programskom jeziku *Swift* najčešće postiže korištenjem sučelja koji se u *Swiftu* nazivaju protokoli (eng. *Protocol*). Uzmimo za primjer klasu `Automobil` prikazanu na programskom kôdu 1. Uzmimo i da ta klasa ima neke atribute, primjerice potrošnju i kapacitet goriva.

```
class Automobil {
    let potrosnja: Float
    let kapacitetGoriva: Float

    init(potrosnja: Float, kapacitetGoriva: Float) {
        self.potrosnja = potrosnja
        self.kapacitetGoriva = kapacitetGoriva
    }
}
```

Programski kôd 1 Primjer klase Automobil

Uzmimo i da imamo klasu `Utrka` prikazanu na programskom kôdu 2. U toj klasi se nalazi lista objekata tipa `Automobil`. Na ovaj način, u utrkama mogu sudjelovati samo objekti tog tipa.

```
class Utrka {
    let trkaci: [Automobil]

    init(trkaci: [Automobil]) {
        self.trkaci = trkaci
    }

    func dajPoredak() -> [Automobili] {
        // logika po kojoj se na temelju potrošnje
        // i kapaciteta određuje poredak
    }
}
```

Programski kôd 2 Primjer klase Utrka

Razmotrimo idući scenarij. U utrku je iz nekog razloga potrebno dodati i osobu opisanu klasom `Osoba` prikazanu na programskom kôdu 9.

```
class Osoba {
    let kondicija: Float
    let umor: Float


    init(kondicija: Float, umor: Float) {
        self.kondicija = kondicija
        self.umor = umor
    }
}
```

Programski kôd 3 Primjer klase Osoba

Tu nastaje problem. Kako bi objekt `Osoba` dodali u utrku, bilo bi potrebno mijenjati ili klasu `Osoba` (tako da nasljeđuje `Automobil` što pak opet nema smisla) ili klasu `Utrka`. Daleko bolji način koji Swift omogućava je izrada protokola. Kreirajmo protokol `Trkac` prikazan na programskom kôdu 4. Njime možemo definirati što želimo da trkač ima.

```
protocol Trkac {
    var rezultat: Float
}
```

Programski kôd 4 Primjer protokola Trkac

`Trkac` sadrži varijablu rezultat. Kako bi klase `Automobil` i `Osoba` postali trkači, dovoljno je da implementiraju taj protokol. Isto tako, klasa `Utrka` umjesto liste objekata tipa `Automobil` treba koristiti listu `Trkac` objekata. Klase mogu implementirati protokole u ekstenzijama što je vidljivo na programskom kôdu 5.

```
extension Automobil: Trkac {
    var rezultat: Float {
        potrosnjaGoriva * kapacitet
    }
}
extension Osoba: Trkac {
    var rezultat: Float {
        kondicija * umor
    }
}
class Utrka {
    let trkaci: [Trkac]

    init(trkaci: [Automobil]) {
        self.trkaci = trkaci
    }


    func dajPoredak() -> [Trkac] {
        // implementacija
    }
}
```

Programski kôd 5 Primjer protokola Trkac – proširenje

Time dobivamo da klase `Automobil` i `Osoba` implementiraju taj protokol. Svaka od tih klasa sada može dati rezultat i time sudjelovati u utrci. Klasa `Utrka` sada sadrži listu

tipa `Trkac` što znači da u utrci može sudjelovati bilo koja klasa koja implementira taj protokol. Time se poštuje *Open-Closed* princip, odnosno nove značajke ne utječu na postojeće.

3) LSP (*Liskov Substitution Principle*) – ako je S podtip od T, tada objekti tipa T mogu biti zamijenjeni objektima tipa S.
Kao i za prethodne principe, najjednostavnije je pokazati slučaj u kojem se ovaj princip krši. Kao primjer možemo uzeti Martinov primjer [11]. Uzmimo klasu `Pravokutnik` prikazanu na programskom kôdu 6 koja sadrži atribute širina i visina. Uz to, u klasi je i metoda za izračun površine.

```
class Pravokutnik {
    var sirina: Int
    var visina: Int

    init(sirina: Int, visina: Int) {
        self.sirina = sirina
        self.visina = visina
    }
    func povrsina() -> Int {
        sirina * visina
    }
}
```

Programski kôd 6 Primjer klase Pravokutnik (prema [11], 79)

Kreirajmo nakon toga klasu `Kvadrat` prikazanu na programskom kôdu 7 koja nasljeđuje `Pravokutnik`, ali uz jednu modifikaciju. Za kvadrat znamo da ima stranice iste duljine. Prema tome, bilo kada se mijenja širina ili visina, drugu stranicu također postavljamo na istu duljinu.

```
class Kvadrat: Pravokutnik {
    override var sirina: Int {
        didSet {
            super.visina = visina
        }
    }
    override var visina: Int {
```

```
            didSet {
                super.sirina = duzina
            }
        }
    }
```

Programski kôd 7 Primjer klase Kvadrat

Od metode za izračun površine pravokutnika očekujemo da izračuna točnu površinu. Uzmemo li objekt tipa `Pravokutnik` sa širinom i visinom duljina 5 i 2, površina izračunata tom metodom iznosi 10, što je i očekivano. No, u slučaju da kreiramo objekt tipa `Kvadrat` sa širinom i visinom duljina 5 i 5 te izmijenimo primjerice širinu na 6, površina izračunata metodom će iznositi 36 jer se izmijenila i duljina visinom. Time se ne poštuje ovaj princip jer objekt potklase ne može adekvatno zamijeniti objekt natklase.

Kao i za prethodni princip, rješenje ovog problema u Swiftu također mogu biti protokoli. Kreiramo li protokol `Povrsina` i njime zadamo da sve klase (i strukture) koje implementiraju taj protokol moraju implementirati metodu za izračun površine, možemo biti sigurni da će iznos površina biti točan i za kvadrat i za bilo koji drugi pravokutnik.

4) ISP (*Interface Segregation Principle*) – potrebno je izbjegavati ovisnosti o onome što se ne koristi.

   Ovo je vrlo jednostavan princip. Cilj je razdvajanje sučelja/protokola tako da sadrže atribute i metode koje klase ili strukture trebaju implementirati, odnosno da ne sadrže ništa što klasa ne treba. Uzmimo za primjer protokol `Vozilo` prikazanog na programskom kôdu 8. Želimo da taj protokol sadrži metode za različite vrste vozila. Primjerice, želimo metodu za najveću brzinu na cesti, najveću brzinu na vodi i potrošnju goriva.

```
protocol Vozilo {
    func najvecaBrzinaNaCesti() -> Float
    func najvecaBrzinaNaVodi() -> Float
    func potrosnjaGoriva() -> Float
}
```

Programski kôd 8 Primjer protokola Vozilo

Nakon što imamo definiran protokol, možemo kreirati klase koje taj protokol nasljeđuju. Primjerice, vozila su automobili i brodovi. Time nastaju klase `Vozilo` i `Brod` prikazane na programskom kôdu 9.

```
class Automobil: Vozilo {
    func najvecaBrzinaNaCesti() -> Float {
        100
    }
    func najvecaBrzinaNaVodi() -> Float {
        // Nije potrebno
    }
    func potrosnjaGoriva() -> Float {
        8
    }
}
class Brod: Vozilo {
    func najvecaBrzinaNaCesti() -> Float {
        // Nije potrebno
    }
    func najvecaBrzinaNaVodi() -> Float {
        25
    }
    func potrosnjaGoriva() -> Float {
        15
    }
}
```

Programski kôd 9 Primjer klasa Automobil i Brod

Već sada vidimo problem. `Automobil` mora implementirati metodu za najveću brzinu na vodi, a ona mu ne treba jer automobil ne može putovati po vodi. Isto tako `Brod` mora implementirati metodu za najveću brzinu na cesti. Ovo je pojednostavljen primjer, ali je lako zaključiti da, kako sve više različitih klasa implementira taj protokol, izmjena tog protokola povlači izmjene u svim tim klasama. Primjerice, želimo li dodati novu metodu za izračun potrošnje struje u protokol `Vozilo`, tada će i klasa `Automobil` i klasa `Brod` morati implementirati tu metodu.

Rješenje je zapravo vrlo jednostavno. Protokol `Vozilo` treba razdvojiti u više protokola. Primjerice, možemo kreirati po jedan protokol za svaku od te tri metode (npr. `CestovnoVozilo`, `MorskoVozilo` i `PotrosnjaVozila`). Tada će klasa `Automobil` implementirati protokole `CestovnoVozilo` i `PotrosnjaVozila`, a klasa Brod `MorskoVozilo` i `PotrosnjaVozila`.

5) DIP (*Dependency Inversion Principle*) – klase ne smiju ovisiti o konkretnim implementacijama već o apstrakcijama.

   U Swiftu, apstrakcije su protokoli. Prema tome, sve klase koje imaju ovisnost, trebaju ovisiti o protokolima.

Ovih se pet principa treba držati kod kreiranja klasa, odnosno bazičnih elemenata. Nakon što su izgrađeni bazični dijelovi (imamo ciglu, beton i staklo u građevinskom primjeru), potrebno ih je povezati u najmanje samostojeće komponente. Komponente mogu biti međusobno povezane. Na primjeru razvoja iOS aplikacija, te komponente možemo nazvati scene. Nakon što konačno imamo klase i iz njih kreirane scene, možemo se posvetiti arhitekturi. No, od velikog broja arhitektura, potrebno se odlučiti za jednu. Moguće je i da jedna arhitektura na jednom projektu neće biti idealna za drugi projekt. Tu se nameće pitanje kako prepoznati dobru arhitekturu. U nastavku će biti napravljen pregled na koji način se neka arhitektura može ocijeniti i što arhitektura mora zadovoljiti kako bi mogli reći da je ona dobra i kako bi mogli biti sigurni u odabir arhitekture.

3.2. Dobra arhitektura

Glavni problem koji arhitektura nastoji riješiti prema [11] je struktura, odnosno pronalazak načina na koji je najlakše i najbrže moguće izgraditi sustav kojeg je moguće održavati. Tu spada raspodjela na komponente te način na koji te komponente međusobno komuniciraju. Arhitektura je zapravo set pravila po kojima se kreira struktura kôda kako bi postigli neki željeni učinak.

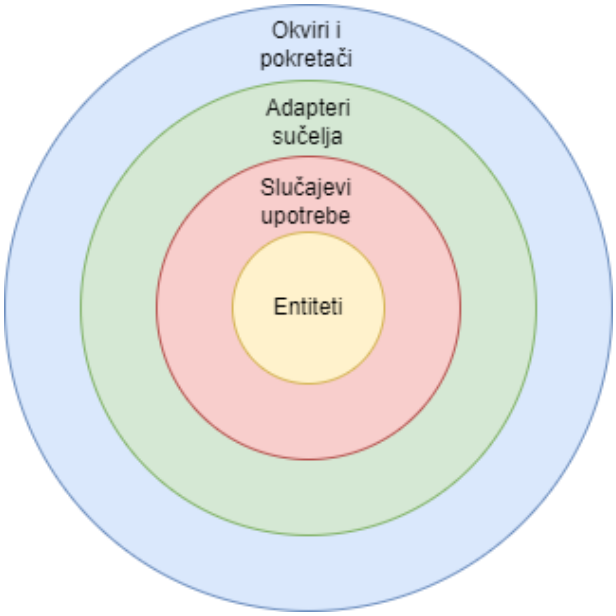
S obzirom na to da pod arhitekturu softvera spada toliko odluka i odgovornosti, postavlja se pitanje na koji način možemo odrediti je li neka arhitektura dobra ili ne. Prema C. Martinu [11], možemo sagledati više pokazatelja koje arhitektura mora podržavati:

1) Slučajevi upotrebe (eng. *uses cases*) sustava – arhitektura mora podržati zahtjeve sustava.
2) Razvoj sustava – u slučaju da na projektu radi više timova ili veći broj developera, arhitektura mora omogućiti neovisan razvoj pojedinih dijelova sustava kako bi se izbjegli konflikti ili duplikacija kôda.
3) *Deployment* sustava – *deployment* sustava ne smije ovisiti o velikom broju točno podešenih konfiguracijskih datoteka i skripti već treba biti moguće trenutno *deployati* sustav.
4) Jednostavnost izmjene sustava – kao što je i ranije navedeno, jednostavnost izmjena je jedna od glavnih značajki u razvoju softvera, pa tako i arhitektura.
5) Razdvajanje slojeva – kako bi sustav bio lako izmjenjiv, potrebno je razdvojiti korisničko sučelje, poslovnu logiku i pomoćne komponente.
6) Testiranje – arhitektura mora omogućavati jedinično testiranje.

Arhitektura ima mnogo. Neke su specifične za tehnologiju, a neke univerzalne. U ovom poglavlju su definirani neki pokazatelji koji mogu biti indikacija je li arhitektura dobra ili nije. U nastavku ovog poglavlja će biti definirana jedna od arhitektura, čista arhitektura.

3.3. Čista arhitektura

Nakon definiranja prava i obaveza arhitektura, vrijeme je za neke konkretne prijedloge. Prema C. Martinu [11], čista arhitektura je arhitektura koja poštuje pravilo o ovisnosti (eng. *The Dependency Rule*). To pravilo glasi „Ovisnosti izvornog kôda mogu pokazivati samo prema unutra, prema politikama višeg sloja“ (eng. *„Source code dependencies must point only inward, toward higher-lever policies“*). Grafički prikaz čiste arhitekture nalazi se na slici 2.



Slika 2 Čista arhitektura (prema [11], str. 203)

Primijenimo li pravilo o ovisnosti na ovu sliku, ono što se nalazi na unutarnjem krugu ne smije znati za ono što se nalazi na vanjskom krugu. Isto tako naziv (bilo naziv metode, klase, varijable i sl.) u unutarnjem krugu ne smije spominjati naziv bilo čega iz vanjskog kruga. Kao što je vidljivo iz grafičkog prikaza, arhitektura se sastoji od nekoliko slojeva. To su entiteti, slučajevi upotrebe, adapteri sučelja te okviri i pokretači. U nastavku će biti pojašnjeni pojedini dijelovi te njihove uloge u arhitekturi.

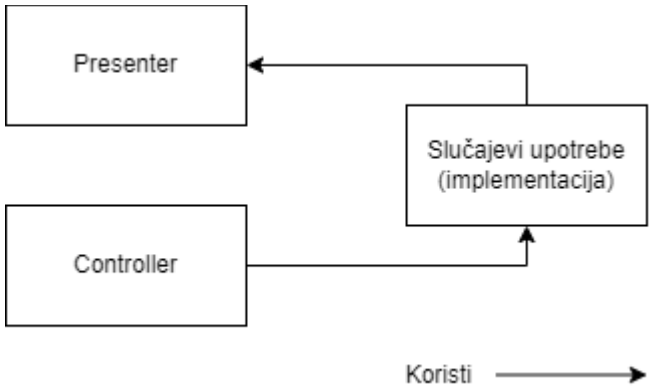
Entiteti predstavljaju kritična poslovna pravila kroz cijelo poduzeće. To mogu biti objekti sa zadanim metodama ili pak samo modeli podataka. Isti su za sve aplikacije tog poduzeća, neovisno o tehnologiji.

U sloj **slučajeva upotrebe** (eng. *Use Case*) spadaju poslovna pravila specifična za sustav tj. aplikaciju. Promjene u ovom sloju ne bi trebale utjecati na entitete, bazu podataka, korisničko sučelje ni vanjske okvire. Bilo kakve promjene poslovnih zahtjeva će uzrokovati promjene na ovom sloju. Česta implementacija ovog sloja su *Interactori*.

Adapteri sučelja (eng. *Interface Adapters*) podrazumijevaju konverzija podataka iz oblika koji najviše odgovara entitetima i slučajevima upotrebe u oblik koji odgovara korisničkom sučelju ili nekom drugom obliku prikaza podataka. Tu spada cijela MVC arhitektura (o njoj će biti riječi nešto kasnije) koja se sastoji od *ViewModela*, *View*a i *Controller*a zajedno sa *Presenterom*. Njegova je dužnost biti između MVC-a i *Use Casea*. Važno je napomenuti da *Presenter* sve podatke koje dobije od *Interactora* treba pretvoriti u *ViewModel* koji se sastoji od *Stringova* i *flagova*. Taj sloj komunicira s *Use Case* slojem tako da od njega zatraži neku radnju. *Use Case* zatim odradi poslovni zahtjev te vraća odgovor *Interface Adapter* sloju.

Okviri i *pokretači* se najčešće sastoje od baza podataka ili pak nekog vanjskog okvira odnosno biblioteka. U većini slučajeva nije potrebno mijenjati taj kôd.

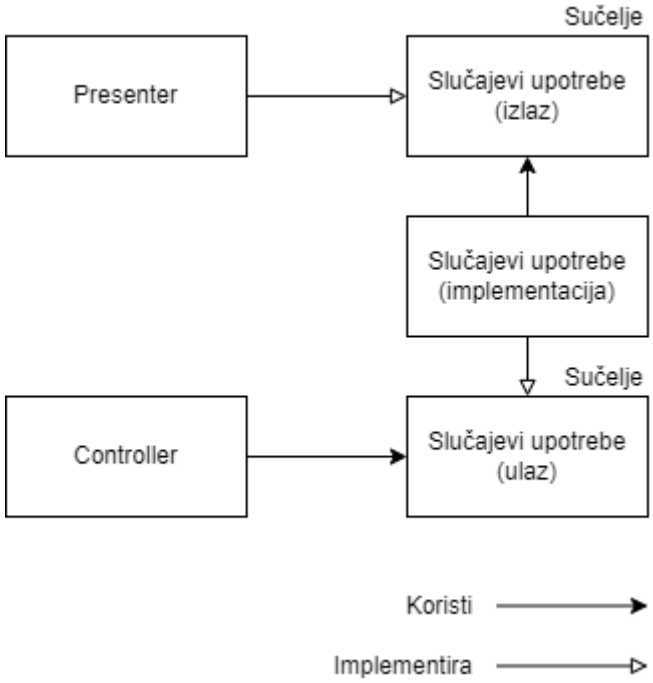
S obzirom na to da su na slici 2 zadana četiri kruga, mogla bi nastati teza kako su oni uvijek potrebni, no prema C. Martinu [11], autoru koncepta čiste arhitekture, sami krugovi nisu toliko bitni. Njih može biti i više prema potrebi. Jedino je pravilo da se arhitektura mora držati *Dependency Rule*-a. No, držati se ovog pravila i nije toliko trivijalno. Razmotrimo slučaj prikazan na slici 3. Na ovoj je slici prikazano **prelaženje granica**.



Slika 3 Čista arhitektura – prelaženje granica (prema [11], 203)

Ranije je navedeno kako su *Controller* i *Presenter* dio adaptera sučelja. Kontrola prolazi iz adaptera sučelja u slučajeve upotrebe i nazad u adaptere sučelja. Takav pristup krši

Dependency Rule. Taj problem je moguće riješiti uvođenjem sučelja ulaz i izlaz. Razmotrimo idući scenarij na slici 4.



Slika 4 Čista arhitektura - sučelja za ulaz i izlaz (prema [11], 203)

Pretpostavimo da interakcija korisnika (primjerice pritisak gumba) dolazi u *Controller*. Pomoću sučelja za ulaz slučajeva upotrebe, *Controller* (odnosno sloj adaptera sučelja) šalje podatke u sloj slučajeva upotrebe. Taj sloj zatim obavlja neku poslovnu funkciju. Nakon obrade i obavljanja poslovnog zahtjeva, putem sučelja za izlaz slučajeva upotrebe se podatci šalju ponovo u sloj adaptera sučelja (u ovom slučaju *Presenteru*). Ovaj pristup ne krši *Dependency rule* jer slojevi ne znaju jedni za druge već samo znaju za sučelja.

U ovom je poglavlju napravljen pregled čiste arhitekture. Glavna značajka ove arhitekture je jednosmjeran tok podataka kroz više slojeva od kojih svaki ima zadanu ulogu. Ako postoji potreba za dvosmjernom komunikacijom, ona mora biti omogućena sučeljima, a ne direktnom dvosmjernom vezom između slojeva. U idućem će poglavlju biti prikazan pregled najkorištenijih arhitektura kod izrade iOS aplikacija. Dvije arhitekture imaju uporište u čistoj arhitekturi.

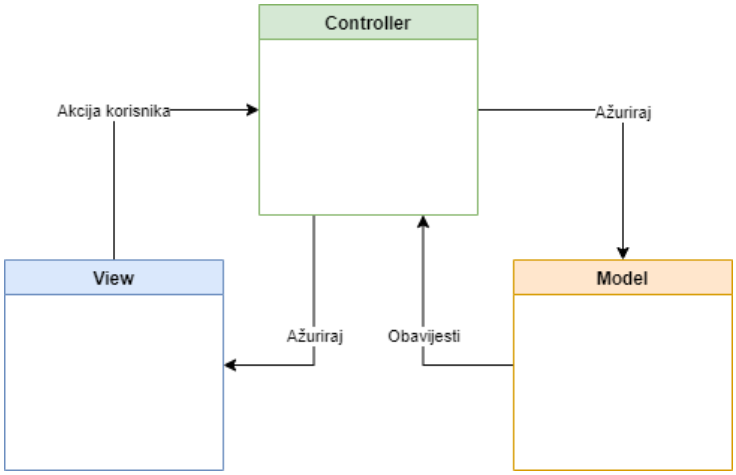
4. iOS arhitekture

Do sada su pokriveni osnovni elementi za izradu aplikacija (klase) i pojam arhitekture. U ovom poglavlju će biti opisane arhitekture koje se koriste u izradi iOS aplikacija. S obzirom na to da većina arhitektura nije vezana za platformu već je apstraktni koncept, potrebno je arhitekture prilagoditi platformama. O svakoj arhitekturi će biti navedene prednosti i mane. Arhitekture koje će biti obrađene su MVC, MVP, MVVM, VIPER i VIP. Valja napomenuti da od svih navedenih arhitektura u ovom poglavlju, jedino su VIPER i VIP arhitekture kreirane specifično za razvoj iOS aplikacija i obje imaju uporište u čistoj arhitekturi. Ostale arhitekture potječu iz drugih grana razvoja softvera, ali se primjenjuju i u razvoju iOS aplikacija.

Ako *Googlamo „Apple* [*bilo koja od arhitektura koje će biti obrađene u ovom poglavlju*] *architecture*“, samo za MVC arhitekturu ćemo dobiti rezultat koji vodi na službenu dokumentaciju Applea. Time možemo zaključiti da je MVC jedina arhitektura koju Apple zagovara. To potkrepljuju i Appleove radionice na kojima sam prisustvovao i na kojima se demonstracije uvijek rade koristeći MVC arhitekturu. Osim toga, glavne komponente *Foundations frameworka* (Appleova biblioteka koja sadrži osnovne elemente za izradu iOS aplikacija) odgovaraju upravo MVC arhitekturi. Za razliku od iOS-a, na službenoj dokumentaciji Androida [16] postoji cijela sekcija o arhitekturi aplikacija. No, bez obzira na ne postojanje službene dokumentacije o arhitekturama kod razvoja iOS aplikacija, arhitekture drugih tehnologija se mogu prilagoditi za razvoj iOS aplikacija. Osim toga, developeri koji imaju potrebu za boljim arhitekturama smišljaju i prilagođavaju nove arhitekture specifično za izradu iOS aplikacija. U nastavku će biti napravljen pregled najkorištenijih arhitektura u izradi iOS aplikacija.

4.1. MVC arhitektura

Prema Appleovoj dokumentaciji [17], MVC (eng. *Model-View-Controller*) je arhitektura koja se sastoji od tri ključna elementa: *Model*, *View* i *Controller*. Na slici 5 je prikazan tok akcija između tih elemenata.



Slika 5 MVC arhitektura (prema [17])

Model predstavlja podatke koji su specifični za aplikaciju. Model je u dvosmjernoj komunikaciji s *Controllerom*.

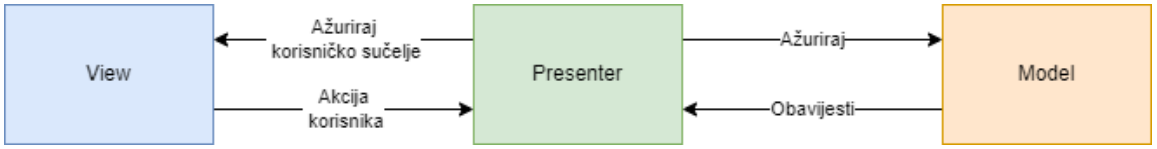
View je objekt korisničkog sučelja. On se sastoji od podataka koji se prikazuju korisniku (bile to tekstualne oznake, gumbovi, slike ili nešto drugo). *View* je također zadužen za preuzimanje korisnikovih akcija kao što je normalan pritisak na gumb, dug pritisak na gumb, *swipe* akcija i sl. *View* prima podatke u obliku *Modela* koje treba prikazati od *Controllera*, a njemu prosljeđuje korisnikove akcije. Ovdje ponovno možemo uočiti spominjanu dvosmjernu komunikaciju.

Controller je središnji dio ove arhitekture. On komunicira s preostala dva elementa dvosmjernom komunikacijom. *Controller* određuje na koji *View* objekt šalje koje podatke, reagira na akcije koje mu šalju *Viewovi*, brine se o životnom ciklusu i obavlja poslovnu logiku.

Kao što možemo vidjeti, *Controller* je centralni modul koji radi sve važne zadaće dok su *View* i *Model* kao potpora koja ne obavlja nikakve važne zadaće. Za male aplikacije s malim brojem funkcionalnosti to nije nužno veliki problem, no postane li aplikacija veća/velika, *Controller* postane centar za sva zbivanja čime postaje masivan te se MVC ponekad interpretira kao *Massive-View-Controller.*

4.2. MVP arhitektura

MVP (eng. *Model-View-Presenter*) je arhitektura koja nastoji riješiti problem velikih *Controllera* u MVC arhitekturi. Prema slici 6, vidimo da su *View* i *Controller* odvojeni zasebno te da je centar arhitekture *Presenter* koji radi s *Modelima*.



Slika 6 MVP arhitektura (prema [18])

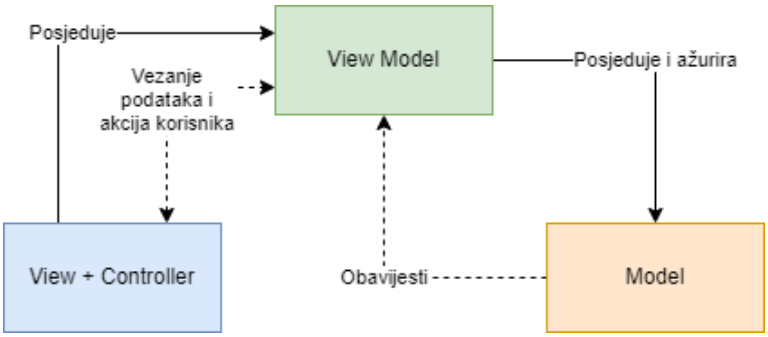
Model predstavlja podatke poslovne logike, kao i u MVC arhitekturi. On je u dvosmjernoj komunikaciji s *Presenterom*.

View se sastoji od *Viewa* koji čini elemente korisničkog sučelja (gumbovi, tekstualne oznake i sl) i *Controllera* koji upravlja tim *Viewom* (jedan *Controller* može upravljati s više *Viewova*, o tome će biti riječi nešto kasnije).

Presenter je središnji dio ove arhitekture. On prima zahtjeve tj. akcije korisnika od Controllera, obavlja svu poslovnu logiku te vraća podatke *Controlleru* koje je potrebno prikazati na *Viewu* tako da poziva metode *Viewa*. Slično kao i kod MVC-a, i u ovoj arhitekturi postoji dio koji ima potencijal biti velik i težak za održavati, a to je u ovom slučaju *Presenter*.

4.3. MVVM arhitektura

MVVM (eng. *Model-View-ViewModel*) je arhitektura koju je 80-ih objavio *Smalltalk* kao pokušaj rješavanja problema velikih *Controllera* u MVC-u. 2005. godine Microsoft je objavio adaptaciju MVVM arhitekture za aplikacije s korisničkim sučeljem na svojim stranicama dokumentacije [19]. Ova arhitektura podsjeća na MVP kao što je vidljivo iz slike 7. *ViewModel* zamjenjuje *Presenter* iz MVP arhitekture i obavlja vrlo sličnu funkciju. No, postoji jedna bitna razlika, a to je da u ovoj arhitekturi *ViewModel* ne poziva metode *Viewa* kako bi *View* prikazao nove podatke već *ViewModel* obavještava *View* da je došlo do promjene podataka. Ti podatci mogu biti u jednostavnijem obliku od domenskih modela. To se može činiti kao mala razlika, ali to omogućava neovisnost *ViewModela* od *Viewa*. Tehnička realizacija može biti koristeći obrazac delegata ili pak koristeći reaktivne biblioteke (npr. Appleov *Combine* ili *RxSwift*).



Slika 7 MVVM arhitektura (prema [20])

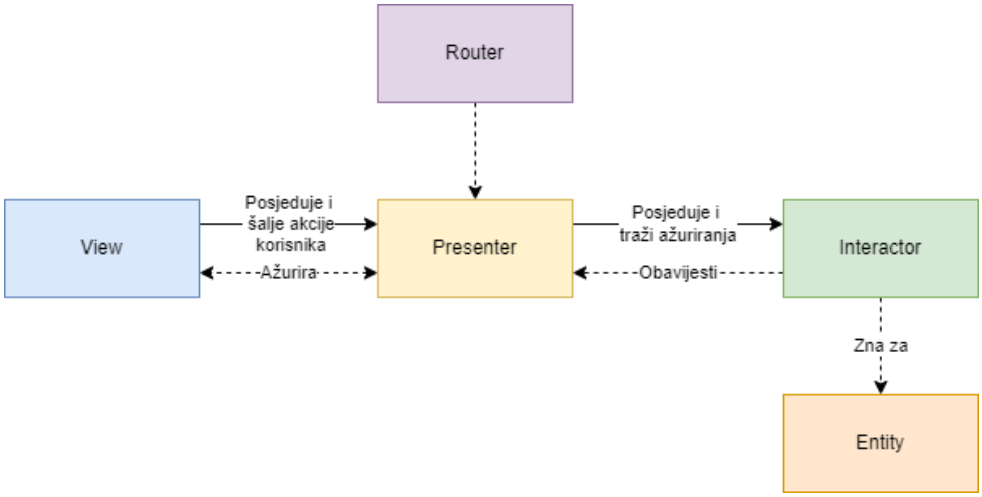
Model, isto kao i u MVP-u predstavlja podatke. On ne ovisi o korisničkom sučelju i sadrži stanja te se samo prikazuje na korisničkom sučelju.

View se, ponovno kao i u MVP-u, odnosi na samo korisničko sučelje (tekstualne oznake, gumbove i sl.) i uključuje *Controller*. U MVP-u, *View* šalje *ViewModelu* akcije korisnika (pozivajući odgovarajuće metode) te osluškuje promjene podataka. Kada *ViewModel* obavijesti *View* o promjeni podataka, *View* može prikazati te podatke samostalno.

ViewModel je zadužen za formatiranje *Modela* u podatke koje *View* može prikazati. On je također zadužen za primanje akcija od *Viewa*. Time se u *ViewModel* odvaja poslovna logika od korisničkog sučelja i modela. *ViewModel* komunicira s *Viewom* i *Modelom* dvosmjernom komunikacijom.

4.4. VIPER arhitektura

VIPER (eng. *View-Interactor-Presenter-Entity-Router*) je arhitektura koja potječe iz Čiste arhitekture R. C. Martina. Sastoji se od pet glavnih slojeva i nastoji razdvojiti odgovornosti što je više moguće. Na slici 8 je prikazana vizualna reprezentacija ove arhitekture.



Slika 8 VIPER arhitektura (prema [21])

View je, kao i u dosadašnjim arhitekturama samo korisničko sučelje i uključuje *Controller* (dakle mogli bi reći *View and Controller-Interactor-Presenter-Entity-Router*). *View* prima akcije korisnika i prosljeđuje ih preko *Controllera* u *Presenter*.

Interactor je dio koji je zaslužen za poslovnu logiku. Od *Presentera* prima zahtjev te koristeći *Entitete* obavlja neku poslovnu funkciju. Nakon obrade *Entiteta*, vraća odgovor *Presenteru*.

Presenter je zadužen za primanje korisnikovih akcija iz *Controllera*, pozivanje *Interactora* da obavi neku poslovnu funkciju te vraćanje podataka koje je potrebno prikazati *Controlleru*. Osim toga, *Controller* može tražiti od *Presentera* da se prikaže neki drugi ekran/scena. *Presenter* u tom slučaju od *Routera* traži izmjenu scene.

Entity je, kao što je u čistoj arhitekturi, apstraktan model koji nije jedinstven za aplikaciju već za cijelo poduzeće.

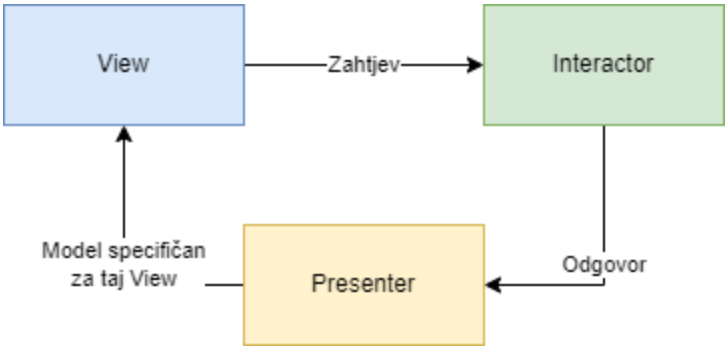
Router je sloj koji omogućava navigaciju. Za razliku od dosadašnjih arhitektura koje ne uzimaju u obzir navigaciju. Navigacija je vrlo koristan i važan dio mobilne aplikacije i poseban,

testabilan sloj je neophodan za rast aplikacije. U VIPER-u, *Router* instancira i prikazuje iduću scenu.

VIPER je relativno složena arhitektura koja odvaja poslovnu logiku od korisničkog sučelja. Iako potiče iz Čiste arhitekture R. C. Martina, ova arhitektura koristi dvosmjernu komunikaciju između *Viewa* i *Presentera* te između *Presentera* i *Interactora* što nije u skladu s *Dependency Ruleom*. Osim toga, *Presenter* može postati vrlo velik jer je on u središtu arhitekture i komunicira s *Viewom*, *Interactorom* i *Routerom*.

4.5. VIP arhitektura

Čistu arhitekturu R. C. Martina je Raymond Law prilagodio za iOS i Mac projekte te kreirao VIP (eng. *View-Interactor-Presenter*) odnosno *Clean Swift* arhitekturu [22]. Iako se iz naziva čini da je to arhitektura koja je smanjenja verzija VIPER-a, to nije točno jer naziv ne govori sve o arhitekturi. VIP samo označava jednosmjerno kretanja podataka vidljivo iz slike 9.



Slika 9 VIP arhitektura (prema [23])

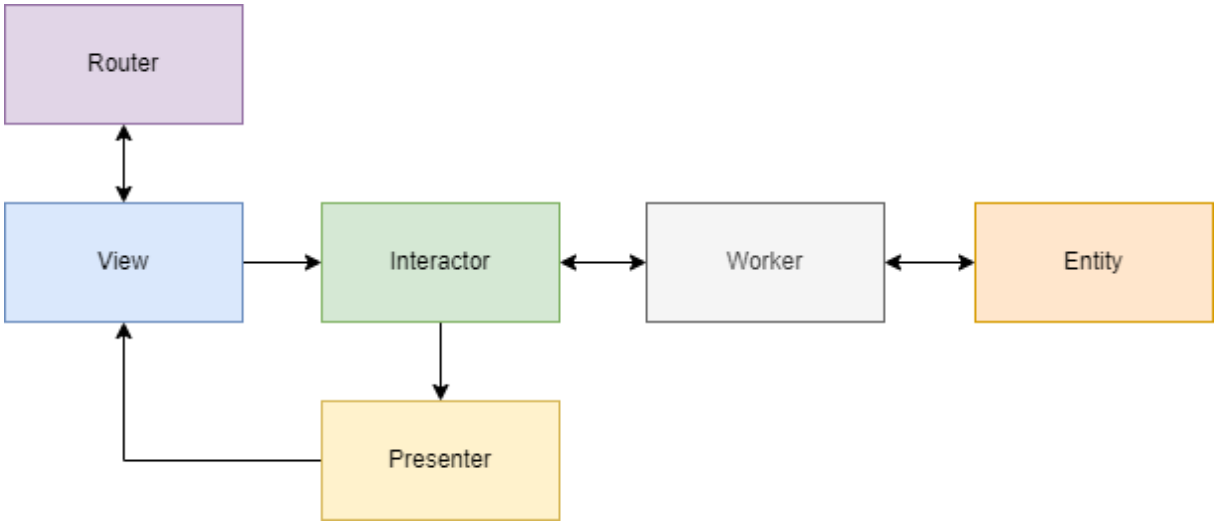
View je, kao i u svim dosadašnjim arhitekturama, samo korisničko sučelje i u VIP arhitekturi uključuje i *Controller*.

Interactor je u središtu arhitekture. On prima akcije korisnika od *Viewa* preko *Controllera*. Zatim implementira poslovnu logiku i nakon obrade podataka u obliku *Modela*, šalje ih *Presenteru*.

Presenter dobiva podatke od *Interactora* koje *View* ne može prikazati pa ih iz tog razloga *Presenter* oblikuje u *ViewModel* odnosno u model koji *View* može prikazati. *ViewModel* se najčešće sastoji od *Stringova* i *flagova*. Kreirani *ViewModel Presenter* šalje *Controlleru* koji pak postavlja *View*.

Router je zadužen za navigaciju. *Controller* sadrži *Router* te preko njega upravlja navigacijom.

Proširimo li VIP na cijeli *Clean Swift*, dobivamo konačan pregled svih komponenti ove arhitekture vidljiv na slici 10. *Interactor* također može imati posebne *Workere* u koje se odvaja neka poslovna logika. Oni mogu pristupati i raditi s entitetima.



Slika 10 Clean Swift arhitektura (prema [24])

Nakon primljene akcije, *View* preko *Controllera* šalje zahtjev (eng. *Request) Interactoru*. *Interactor* pomoću *Workera* obavlja poslovne funkcije koristeći *Entity* te odgovor (eng. *Response)* šalje *Presenteru*. *Presenter* zatim kreira model specifičan za to korisničko sučelje (*View Model)* koji šalje *Controlleru* koji na temelju toga može poslati podatke u *View* i/ili pozvati *Router* za navigaciju na drugu scenu.

Od svih do sada nabrojanih arhitektura, ovo je jedina arhitektura koja poštuje *Dependency Rule*. Kako je već spomenuto, mogućnost izmjene je jedna od glavnih zadaća arhitektura. Tu mogućnost olakšavaju i podupiru testovi. Iduće će se poglavlje baviti mogućnostima testiranja u iOS arhitekturama navedenim u ovom poglavlju.

4.6. Testiranje u iOS arhitekturama

Do sada su bile navedene najkorištenije arhitekture u razvoju iOS mobilnih aplikacija. Testiranje je vrlo važan dio svakog razvoja softvera pa tako i mobilnih aplikacija za iOS. Rast projekta, uvođenje novih ljudi na projekt, održavanje kroz vremenski period. Sve su to faktori koji povećavaju mogućnost da, ako netko nešto promjeni u kôdu, postoji šansa da neki od prijašnjih dijelova neće raditi ispravno. To mogu biti sitnice kao što su krivi font na nekom od ekrana do pogrešne bankovne transakcije ili nešto gore. U oba slučaja klijent, a ni korisnici neće biti zadovoljni. To će dovesti do dugotrajnog *debuggiranja* i gubljenja vremena i resursa na nešto što se vrlo vjerojatno moglo izbjeći pisanjem testova.

Xcode podržava pisanje jediničnih testova (eng. *unit tests*) te testova korisničkog sučelja (eng. *UI tests*). Valja napomenuti da ono što Xcode i Apple smatraju UI testovima, zapravo i nisu UI testovi već integracijski testovi. U tim testovima nije moguće pristupiti objektima kao što je npr. `UILabel` direktno već ti testovi rade na principu crne kutije koju predstavlja `XCUIApplication`. Moguće je samo u kôdu postaviti referencu na labelu te u testovima tražiti od `XCUIApplicationa` da pristupi toj labeli. Time je onemogućeno postavljanje bilo kakvih *mockova* što može dovesti do toga da, ako početni dio testa padne, drugi se dio koji ovisi o prvom dijelu ne testira što je zapravo integracijski test. Uzmimo za primjeri idući scenarij. Aplikacija ima ekran za prijavu na kojem je potrebno unijeti korisničko ime i lozinku za prijavu. Ako su podatci točni (što se provjerava putem API poziva), otvara se idući ekran, primjerice početni ekran, a ako su podatci netočni, prikazuje se greška. Ako želimo testirati korisničko sučelje, uređaj koji izvršava testove mora biti spojen na Internet (s obzirom na to da je potrebno odraditi API poziv) te u bazi podataka moraju postojati testni podatci jer će u protivnom testovi korisničkog sučelja pasti. Testiranje samo početnog ekrana također nije moguće bez testiranja ekrana za prijavu. Prema [25], jedno od rješenja je kreiranje *mock* servera što nije trivijalno za implementirati. Druga mogućnost je dodavanje posebnih provjera unutar samih metoda koje izvršavaju mrežne zahtjeve. To je vrlo loše s obzirom na to da se u produkcijski kôd dodaje kôd za testiranje i prema autoru članka, ta bi se metoda trebala izbjegavati. S obzirom na takav pristup testiranju korisničkog sučelja, arhitekture ne mogu utjecati na izgled i izvođenje tih testova. Iz tog razloga će u aplikaciji koja prati ovaj rad biti izrađeni samo jedinični testovi.

Jedinični testovi se kreiraju u klasama koje nasljeđuju `XCTestCase`. Svaka testna klasa može nadjačati (eng. *override*) `setupWithError()` i `tearDownWithError()` metode. Te se metode pokreću prije svakog izvođenja jediničnog testa za prvu te na kraju izvođenja jediničnog testa za drugu metodu. U njima se postavljaju *mockovi* te klase i pomoćne

klase koje se koriste pri testovima čime se omogućuje početno stanje svakom testu. Kako bi *Xcode* prepoznao test, potrebno je započeti naziv testne metode sa *test*. Za provjeru ispravnosti se koriste `XCTAssert` metode.

S obzirom na to da pisanje velikog broja jediničnih testova u *Xcodeu* može dovesti do testnih metoda sa velikim, nerazumljivim nazivima, jedno od rješenja je korištenje vanjskih biblioteka *Quick & Nimble*. One omogućavaju preglednije pisanje testova. Isto tako, ručno pisanje *mockova* može uzeti dosta vremena i kod izmjena kôda koji se testira, tražiti izmjenu i *mockova*. Korištenje vanjske biblioteke primjerice *Mockingbird* podosta olakšava taj proces. U praktičnom dijelu rada ipak neće biti korištena nijedna biblioteka za pisanje jediničnih testova s obzirom na to da se radi o relativno malom broju testova koje je moguće jednostavno kreirati i održavati bez korištenja biblioteka.

Za razliku od testova korisničkog sučelja, arhitektura uvelike utječe na mogućnost jediničnog testiranja. U nastavku će biti napravljen osvrt na već spomenute arhitekture i mogućnosti pisanja jediničnih testova u razvoju iOS mobilnih aplikacija. Za prikaz testiranja u pojedinim arhitekturama bit će dani primjeri jediničnih testova. Oni će obuhvaćati testiranja pojedinih dijelova arhitektura. No, prije samih testova, potrebno je kreirati nešto što će ti testovi testirati. To će biti izvedeno na način da će svaka arhitektura implementirati istu, jednostavnu funkcionalnost. Na primjer, uzmimo da aplikacija treba sadržavati jedan ekran prikazan na slici 11. Na tom ekranu neka se nalazi gumb. Pritiskom gumba, potrebno je koristiti API (https://dog-api.kinduff.com/api/facts) putem kojeg se dohvaća po jedna činjenica o psima. Tu je činjenicu zatim potrebno prikazati na ekranu iznad gumba.

Slika 11 Korisničko sučelje aplikacije za jedinične testove

Različite arhitekture mogu ipak imati neke zajedničke komponente. Ti su dijelovi prikazani na programskom kôdu 10. Zajedničke komponente čine korisničko sučelje (`CommonView`) koje sadrži labelu i gumb, zatim model odgovora API-a (`APIResponse`), domenski model koji predstavlja činjenicu (`Fact`) te repozitorij (`Repository`) koji poziva API.

```
class CommonView: UIView {
    private let label: UILabel = {
        let label = UILabel()
        label.textAlignment = .center
        label.numberOfLines = 0
        return label
    }()

    private lazy var button: UIButton = {
```

```
        let button = UIButton()
        button.setTitle("Dohvati info", for: .normal)
        button.addGestureRecognizer(UITapGestureRecognizer(target: self,
action: #selector(buttonTap)))
        return button
    }()

    var didTapButton: (() -> Void)?
…
}

struct APIResponse: Decodable {
    let facts: [String]?
}

struct Fact: Equatable {
    let text: String
}

protocol RepositoryProtocol {
    func getDogFacts(_ completion: @escaping ((Result<APIResponse,
AFError>) -> Void))
}

class Repository: RepositoryProtocol {
    func getDogFacts(_ completion: @escaping ((Result<APIResponse,
AFError>) -> Void)) {
        AF.request("https://dog-
api.kinduff.com/api/facts").responseDecodable(of: APIResponse.self) {
response in
            completion(response.result)
        }
    }
}
```

Programski kôd 10 Jedinični testovi - zajedničke komponente

Osim ovih zajedničkih komponenti, i jedinični testovi arhitektura mogu imati zajedničke dijelove. U ovom primjeru, to se odnosi na dva zajednička *mocka* prikazana programskim kôdom 11.

```
class RepositoryMock: RepositoryProtocol {
    private let dataMock = DataMock()

    var error: AFError?
    var getDogFactsCalled = false
    var getDogFactsCounter = 0

    func getDogFacts(_ completion: @escaping ((Result<APIResponse,
AFError>) -> Void)) {
        getDogFactsCalled = true
        getDogFactsCounter += 1

        guard let error = error else {
            completion(.success(dataMock.getAPIResponse()))
            return
        }
        completion(.failure(error))
    }
}

struct DataMock {
    var facts: [String]? = ["test fact"]

    func getAPIResponse() -> APIResponse {
        APIResponse(facts: facts)
    }

    func getFact() -> Fact {
        Fact(text: facts?.first ?? "")
    }
}
```

Programski kôd 11 Jedinični testovi - zajednički *mockovi*

Mockovi služe za provjeravanje je li neka metoda pozvana te s kojim je podatcima pozvana. Više riječi o *mockovima* bit će u poglavlju 6.3.

Nakon zajedničkih komponenti, na red dolaze pojedine arhitekture. U nastavku će biti napravljen prikaz implementacije spomenute funkcionalnosti u različitim arhitekturama te pripadajući jedinični testovi.

4.6.1. MVC jedinični testovi

Ova je arhitektura korisna za brz razvoj manjih aplikacija. Najveća prednost joj je i najveća mana, a to je da su korisničko sučelje, njegova kontrola i poslovna logika, sve na jednom mjestu. Kako raste `ViewController`, on postaje nepregledan i može se desiti da se sve veći broj metoda ponavlja ili se treba izdvajati u posebne klase. To se isto odražava na testove. S obzirom na to da je prioritet testirati poslovnu logiku, a ona je u `ViewControlleru`, potrebno je pisati testove za cijeli `ViewController`. To je još i dobar slučaj s obzirom na to da postoji mogućnost i da poslovnu logiku uopće nije moguće testirati. U slučaju da postoji ista poslovna logika na primjerice dva ekrana i raspisana je u dvije različite metode u dva različita `ViewControllera`, potrebno je tu logiku dva puta testirati i u slučaju promjene na jednom mjestu, promjena neće biti izvršena na drugom. Tako će prvi test pasti, ali drugi proći što nije idealan slučaj.

Pogledajmo primjer ranije spomenute aplikacije. Potrebno je koristeći `Repository` dohvatiti činjenicu o psu na dodir gumba i prikazati je. Implementacija tog *Controllera* se nalazi na programskom kôdu 12.

```
class MVCController: UIViewController {
    private let commonView: CommonView
    private let repository: RepositoryProtocol

    init(repository: RepositoryProtocol) {
        self.repository = repository
        commonView = CommonView()
        super.init(nibName: nil, bundle: nil)
    }

    required init?(coder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }

    override func loadView() {
        super.loadView()
```

```
        view = commonView
        setupButtonAction()
    }

    private func setupButtonAction() {
        commonView.didTapButton = { [unowned self] in
            self.getFacts()
        }
    }
}

extension MVCController {
    private func getFacts() {
        repository.getDogFacts { [weak self] result in
            guard case .success(let apiResponse) = result else { return }
            let text = apiResponse.facts?.first ?? ""
            self?.commonView.updateLabelWith(text)
        }
    }
}
```

Programski kôd 12 Jedinični testovi – MVCController

Vidljivo je da poslovnu i prezentacijsku logiku obavlja *Controller*. Na pritisak gumba, *View* putem `didTapButton` metode obavještava *Controller* da je pritisnut gumb. *Controller* zatim koristeći repozitorij dohvaća činjenicu i poziva metodu *Viewa* `updateLabelWith` kako bi se prikazala ta činjenica.

Nakon što imamo neku funkcionalnost, potrebno je tu funkcionalnost testirati. No, tu dolazimo do problema ove arhitekture. Kako testirati poslovnu logiku? Znamo da je poslovna logika dohvaćanje činjenice o psima putem repozitorija. Također znamo da se ona poziva tek na dodir gumba. S obzirom na to da je `commonView` privatna varijabla, a `getFacts()` privatna metoda, njima iz testova ne možemo pristupiti. Prema tome, poslovnu logiku nije moguće testirati. Loše rješenje bi bilo učiniti i varijablu i metodu javnom.

Na ovaj način se MVC arhitekturom kreiraju male i jednostavne aplikacije. Jedinično testiranje je vrlo nepraktično ili nemoguće. U idućem poglavlju će na sličan način biti prikazana mogućnost jediničnog testiranja koristeći MVP arhitekturu.

4.6.2. MVP jedinični testovi

MVP omogućava svakako bolju pokrivenost testovima od MVC-a. S obzirom na to da je centar arhitekture *Presenter*, ako veličina projekta odnosno funkcionalnosti pojedine scene postane velik, veličina testnih klasa će rasti s veličinom *Presentera* odnosno i više s obzirom na to da najčešće nije dovoljno napisati jedan test za jednu metodu. Isto tako, u slučaju bilo kakvih promjena *Modela* koji su dio poslovne logike, bit će potrebno izmijeniti i testove koji se odnose na vraćanje podatka iz *Presentera* u *Controller* odnosno *View*. Uzmimo za primjer ponovno ranije zadanu funkcionalnost, ali ovaj puta implementiranu koristeći MVP arhitekturu. Promotrimo za početak implementaciju `MVPControllera` prikazanog programskim kôdom 13. On sadrži *Presenter* te korisničko sučelje. Također implementira `MVPControllerProtocol`. Pritiskom na gumb, *Controller* poziva metodu *Presentera* koji obavlja poslovnu logiku.

```
protocol MVPControllerProtocol: AnyObject {
    func setLabel(label: String)
}

class MVPController: UIViewController {
    private let commonView: CommonView
    private var mvpPresenter: MVPPresenterProtocol

    init(mvpPresenter: MVPPresenterProtocol) {
        self.mvpPresenter = mvpPresenter
        commonView = CommonView()
        super.init(nibName: nil, bundle: nil)
        self.mvpPresenter.mvpController = self
    }

    required init?(coder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }

    override func loadView() {
        super.loadView()
        view = commonView
        setupButtonAction()
    }
```

```
    private func setupButtonAction() {
        commonView.didTapButton = { [unowned self] in
            self.mvpPresenter.getDogFact()
        }
    }
}

extension MVPController: MVPControllerProtocol {
    func setLabel(label: String) {
        commonView.updateLabelWith(label)
    }
}
```

Programski kôd 13 Jedinični testovi – MVPController

Osim Controllera, ova arhitektura sadrži i Presenter prikazan programskim kôdom 14. On pomoću repozitorija dohvaća podatke. Osim toga, sadrži referencu na *Controller* te ima pristup svim metodama `MVPControllerProtocola`.

```
protocol MVPPresenterProtocol {
    func getDogFact()
    var mvpController: MVPControllerProtocol? { get set }
}

class MVPPresenter: MVPPresenterProtocol {
    private let repository: RepositoryProtocol

    unowned var mvpController: MVPControllerProtocol?

    init(repository: RepositoryProtocol) {
        self.repository = repository
    }

    func getDogFact() {
        repository.getDogFacts { [weak self] result in
            guard case .success(let apiResponse) = result else { return }
            let fact = Fact(text: apiResponse.facts?.first ?? "")
```

```
            self?.mvpController?.setLabel(label: fact.text)
        }
    }
}
```

Programski kôd 14 Jedinični testovi - MVPPresenter

Pristup svim metodama protokola *Controllera* je dosta problematično. Time se onemogućava ponovno korištenje *Presentera* te mu se daje pristup metodama kojima ne bi nužno trebao imati pristup. Jedno rješenje tog problema bi bilo kreiranje drugog protokola *Controllera* koji bi se odnosio samo na interakciju s *Presenterom*.

Sada na red dolazi jedinično testiranje. Za razliku od MVC arhitekture, u ovoj arhitekturi je ipak moguće testirati poslovnu logiku. To je moguće zato što je poslovna logika odvojena u *Presenter* čije su metode definirane protokolom (dakle javne su) i jer on koristi `RepositoryProtocol` što omogućava korištenje *mockova* za provjeru jesu li pozvane ispravne metode. Osim zajedničkih *mockova*, ovdje će nam trebati i *mock* `MVPControllera` koji je prikazan na programskom kôdu 15.

```
class MVPControllerMock: MVPControllerProtocol {
    var didSetLabelCalled = false
    var didSetLabelCounter = 0
    var label: String?

    func setLabel(label: String) {
        didSetLabelCalled = true
        didSetLabelCounter += 1
        self.label = label
    }
}
```

Programski kôd 15 Jedinični testovi – MVPControllerMock

Na programskom kôdu 16 je prikazana implementacija jediničnih testova za `MVPPresenter` iz primjera.

```
class MVPPresenterTests: XCTestCase {
    private var sut: MVPPresenter!
    private var repositoryMock: RepositoryMock!
    private var mvcControllerMock: MVPControllerMock!
    private var dataMock: DataMock!

    override func setUpWithError() throws {
        repositoryMock = RepositoryMock()
        mvcControllerMock = MVPControllerMock()
        sut = MVPPresenter(repository: repositoryMock)
        sut.mvpController = mvcControllerMock
        dataMock = DataMock()
    }

    override func tearDownWithError() throws {
        mvcControllerMock = nil
        repositoryMock = nil
        sut = nil
        dataMock = nil
    }
}

// MARK: getDogFact() tests
extension MVPPresenterTests {
    func testGetDogFact_WhenCalled_ShouldCallRepositoryGetDogFacts() {
        // When
        sut.getDogFact()

        // Then
        XCTAssertTrue(repositoryMock.getDogFactsCalled)
        XCTAssertEqual(repositoryMock.getDogFactsCounter, 1)
    }

    func
testGetDogFact_WhenCalled_OnSuccess_ShouldCallMVPControllerSetLabel() {
        // Given
        let expectedFact = dataMock.getFact().text

        // When
        sut.getDogFact()
```

```
        // Then
        XCTAssertTrue(mvcControllerMock.didSetLabelCalled)
        XCTAssertEqual(mvcControllerMock.didSetLabelCounter, 1)
        XCTAssertEqual(mvcControllerMock.label, expectedFact)
    }

    func testGetDogFact_WhenCalled_OnFailure_ShouldNotCallPresenterOutputDidGetDogFactWithFact() {
        // Given
        repositoryMock.error = AFError.explicitlyCancelled

        // When
        sut.getDogFact()

        // Then
        XCTAssertFalse(mvcControllerMock.didSetLabelCalled)
    }
}
```

Programski kôd 16 Jedinični testovi - MVPPresenter testovi

Za metodu `getDogFact()` kreirana su tri testa. U prvom se testu provjerava je li pozvana ispravna metoda repozitorija. U drugom testu se provjerava je li u slučaju uspješnog dohvaćanja činjenice pozvana ispravna metoda `MVCControllera`. U zadnjem se testu provjerava je li u slučaju neuspješnog dohvaćanja činjenice metoda `MVCControllera` ostala ne pozvana.

Ova arhitektura omogućava testiranje poslovne logike. Mana joj je povezivanje *Controllera* i *Presentera* čime *Presenter* postaje neiskoristiv na drugim mjestima. Taj problem rješava MVVM arhitektura čiji će primjer biti dan u idućem poglavlju.

4.6.3. MVVM jedinični testovi

U ovoj arhitekturi ključan dio su *ViewModeli* koji sadrže svu poslovnu logiku. Iz tog je razloga njih važno testirati. Kako oni često koriste neke druge klase kao što su API klase, bitno je voditi računa o tome da te klase implementiraju protokole. Tako je moguće za njih kreirati *mockove* te osigurati da ostale klase ne utječu na ponašanje *ViewModela*. Ako se desi da padne test *ViewModela*, znamo da je greška u njemu i da ne moramo provjeravati prateće klase. Njih je potrebno testirati posebno koristeći *mockove* ako je potrebno. S obzirom na to da se u *ViewModel* klase odvaja poslovna logika koja se može razlikovati od ekrana do ekrana, čest je slučaj da jedan ekran ima vlastiti *ViewModel* i eventualno koristi neki od zajedničkih *ViewModela*. To dovodi do toga da su *ViewModel* klase relativno male, odnosno implementiraju relativno mali broj metoda zbog čega je testiranje puno preglednije. Osim toga, MVVM rješava problem MVP arhitekture na način da se u *ViewModel* ne šalje referenca *Controllera* već *Controller* reaktivno može dobivati podatke iz *ViewModela*. To se postiže korištenjem reaktivnih biblioteka ili u jednostavnijem slučaju, korištenjem *closurea* što će biti prikazano na primjeru. Pogledajmo primjer `MVVMControllera` prikazanog programskim kôdom 17. *Contoller* sadrži korisničko sučelje i *ViewModel* što je vrlo slično MVP primjeru. No, u konstruktoru se ne kreira referenca *Controllera* na *ViewModel* već se u metodu `bindData()` *Controller* „spaja“ na obavijesti o promjenama jer se `bindDogFact` *closure ViewModela* poziva svaki put kad se dohvati nova činjenica.

```
class MVVMController: UIViewController {
    private let commonView: CommonView
    private var mvvmViewModel: MVVMViewModelProtocol

    init(mvvmViewModel: MVVMViewModelProtocol) {
        self.mvvmViewModel = mvvmViewModel
        commonView = CommonView()
        super.init(nibName: nil, bundle: nil)
    }

    required init?(coder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }

    override func loadView() {
        super.loadView()
```

```
        view = commonView
        setupButtonAction()
        bindData()
    }

    private func setupButtonAction() {
        commonView.didTapButton = { [unowned self] in
            self.mvvmViewModel.getDogFact()
        }
    }

    private func bindData() {
        mvvmViewModel.bindDogFact = { [weak self] fact in
            self?.commonView.updateLabelWith(fact)
        }
    }
}
```

Programski kôd 17 Jedinični testovi – MVVMController

Implementacija *ViewModela* vidljiva je na programskom kôdu 18. `MVVMViewModelProtocol` sadrži sve ulaze i izlaze *ViewModela*. `bindDogFact` je *closure* koji se poziva nakon uspješnog dohvaćanja činjenice s API-a.

```
protocol MVVMViewModelProtocol {
    func getDogFact()
    var bindDogFact: ((_ fact: String) -> Void)? { get set }
}

class MVVMViewModel: MVVMViewModelProtocol {
    private let repository: RepositoryProtocol

    var bindDogFact: ((String) -> Void)?

    init(repository: RepositoryProtocol) {
        self.repository = repository
    }

    func getDogFact() {
```

```
        repository.getDogFacts { [weak self] result in
            guard case .success(let apiResponse) = result else { return }
            let fact = apiResponse.facts?.first ?? ""
            self?.bindDogFact?(fact)
        }
    }
}
```

Programski kôd 18 Jedinični testovi – MVVMViewModel

Sva poslovna logika je u MVVM-u odvojena u *ViewModel* koji putem *closurea* ili neke već spomenute reaktivne biblioteke može obavijestiti *Controller* o promjeni šaljući jednostavan oblik podataka. Na programskom kôdu 19 prikazan je primjer testiranja `MVVMViewModela`. Možemo primijetiti da se koriste samo zajednički *mockovi*. Slično kao i kod primjera testiranja *Presentera* u MVP arhitekturi, testira se metoda `getDogFact()`. Tu metodu testiraju tri testa. Prvi test testira je li pozvana ispravna metoda repozitorija. Drugi test provjerava je li na uspješno dohvaćanje činjenice pozvan/ažuriran `bindDogFact` *closure*. Treći test provjerava je li taj *closure* ostao ne pozvan u slučaju neuspješnog dohvaćanja činjenice.

```
class MVVMViewModelTests: XCTestCase {
    private var sut: MVVMViewModel!
    private var repositoryMock: RepositoryMock!
    private var dataMock: DataMock!

    override func setUpWithError() throws {
        repositoryMock = RepositoryMock()
        sut = MVVMViewModel(repository: repositoryMock)
        dataMock = DataMock()
    }

    override func tearDownWithError() throws {
        repositoryMock = nil
        sut = nil
        dataMock = nil
    }
}

// MARK: getDogFact() tests
```

```
extension MVVMViewModelTests {
    func testGetDogFact_WhenCalled_ShouldCallRepositoryGetDogFacts() {
        // When
        sut.getDogFact()

        // Then
        XCTAssertTrue(repositoryMock.getDogFactsCalled)
        XCTAssertEqual(repositoryMock.getDogFactsCounter, 1)
    }

    func testGetDogFact_WhenCalled_OnSuccess_ShouldCallBindDogFact() {
        // Given
        let expectedFact = dataMock.getFact().text
        let expectation = expectation(description: "expectation")
        sut.bindDogFact = { fact in
            guard fact == expectedFact else { return }
            expectation.fulfill()
        }

        // When
        sut.getDogFact()

        // Then
        wait(for: [expectation], timeout: 5)
    }

    func testGetDogFact_WhenCalled_OnFailure_ShouldNotCallBindDogFact() {
        // Given
        repositoryMock.error = AFError.explicitlyCancelled
        sut.bindDogFact = { fact in
            XCTFail()
        }

        // When
        sut.getDogFact()
    }
}
```

Programski kôd 19 Jedinični testovi - MVVMViewModel testovi

Na ovaj se način testira poslovna logika u MVVM arhitekturi. *ViewModel* je moguće koristiti na više mjesta s obzirom na to da ne ovisi o ni jednom *Controlleru* već svaka klasa koja koristi *ViewModel* može od njega tražiti neku akciju i osluškivati promjene podataka. Biblioteke kao što je *RxSwift* pružaju još više mogućnosti za osluškivanje promjena, ali s obzirom na jednostavnost primjera, ovdje nisu korištene.

Preostale su još dvije arhitekture: VIP i VIPER. Ove su arhitekture po mnogočemu vrlo slične, osim u toku podataka. Detaljan primjer testiranja u VIP arhitekturi bit će prikazan u praktičnom primjeru u poglavlju 6.3. Ukratko, u toj se arhitekturi mogu testirati *Interactor*, *Presenter* i *Router* uz korištenje *mockova*. *Interactor* predstavlja poslovnu logiku, *Presenter* prezentacijsku logiku, a *Router* navigacijsku logiku. Slično tome, u VIPER arhitekturi te također mogu testirati *Interactor*, *Presenter* i *Router*. Jedina je razlika u tome što *Presenter* ima ulogu primanja akcija od *Controllera* te prosljeđivanja podataka iz *Interactora* u *Controller* (dvosmjerna komunikacija).

Jedinično testiranje je potrebno u bilo kojoj arhitekturi. MVC je najgora arhitektura u pogledu testiranja. MVP i MVVM su donekle u istom rangu srednje dobrih po pitanju testiranja dok su VIPER i VIP odlične. Scholichinu i sur. [26] su napravili istraživanje na temu performansi testiranja u MVC, MVP, MVVM i VIPER arhitekturama. Prema tom istraživanju, na temelju vremena izvođenja testova, veličine testnih klasa, izmjenjivosti i performansama (memorija i procesorska snaga), najbolje arhitekture za testiranje su MVVM i VIPER.

5. Usporedba iOS arhitektura

U poglavlju 3.2 je opisano na koji način možemo odrediti je li neka arhitektura dobra. U ovom će poglavlju biti napravljena analiza do sada spomenutih arhitektura u izradi iOS aplikacija. Na tablici 1 napravljena je usporedba arhitektura na temelju pokazatelja dobrih arhitektura opisanih u poglavlju 3.2. Na temelju dosadašnjih opisa arhitektura, za svaki od pokazatelja je dodijeljena jedna od tri oznaka: **+**, **+/-** ili **-**.

Tablica 1 Usporedba arhitektura

Arhitektura	Slučajevi upotrebe (1)	Razvoj sustava (2)	*Deployment* sustava (3)	Jednostavnost izmjene sustava (4)	Razdvajanje slojeva (5)	Testiranje (6)
MVC	**+**	**-**	**+**	**-**	**-**	**-**
MVP	**+**	**+/-**	**+**	**+/-**	**+/-**	**+/-**
MVVM	**+**	**+/-**	**+**	**+**	**+/-**	**+/-**
VIPER	**+**	**+**	**+**	**+**	**+**	**+**
VIP	**+**	**+**	**+**	**+**	**+**	**+**

Sve spomenute arhitekture su u stanju obaviti poslovnu funkciju. *Deployment* sustava u razvoju iOS aplikacija najviše ovisi o pomoćnim datotekama koje kod kreiranja projekta generira *Xcode* i te datoteke ne ovise o arhitekturi.

MVC – iz tablice je vidljivo da ovu arhitekturu valja izbjegavati što je više moguće. Iznimke mogu biti vrlo male aplikacije (od svega nekoliko ekrana/scena) s malim brojem funkcionalnosti. No, i s tim valja biti oprezan jer je uvijek lako misliti da „već ću prebaciti projekt na drugu arhitekturu“ i to se ne desi i nastanu veliki problemi. Testiranje je vrlo limitirano jer je sva poslovna logika u *Controlleru*. *Contoller* brzo postane ogroman i kako projekt raste, izmjene su sve teže i problematičnije. Paralelan rad više developera na ovoj arhitekturi je znatno otežan, odnosno u nekim slučajevima i nemoguć.

MVP - Iako daleko bolja od MVC arhitekture, ova arhitektura ostavlja otvoren problem što *View* koristi modele iz poslovne logike/entitete. To donekle onemogućava ponovno korištenje i bilo kakva izmjena na modelima poslovne logike zahtjeva izmjenu i na razini *Viewa*

za razliku od primjerice MVVM arhitekture koja, u slučaju izmjene entiteta treba samo izmijeniti pretvorbu tih modela u *ViewModel*. Testiranje je svakako lakše i bolje naspram MVC arhitekture. Testirati se može poslovna logika koja je odvojena u *Presenteru*. Rad u timu je donekle olakšan premda svaki član tima mora raditi na svojoj sceni kako se ne bi stvorili konflikti. S obzirom na to da *Presenter* sadrži i poslovnu i prezentacijsku logiku, *Presenter* može postati velik i nepregledan.

MVVM – arhitektura koja rješava nedostatak MVP arhitekture. Odvajanje modela koji koristi *View* u poseban model daje jednostavnu mogućnost izmjene. Testiranje je također znatno bolje od MVC-a i nešto bolje od MVP-a s obzirom na *ViewModel*. Loša strana arhitekture je, slično kao i u MVP-u, veličina centralnog modula koji obavlja poslovnu logiku i konverziju poslovnih modela u modele razumljive *Viewu*.

VIPER – relativno kompleksna arhitektura koja se bazira na Čistoj arhitekturi R. C. Martina. Odlična je za veće projekte na kojima radi veći broj ljudi jer više ljudi može raditi na istoj sceni, ali na različitim modulima. Testiranje je odlično jer se testovima mogu pokriti svi slojevi. Negativna strana je potencijalno velik *Presenter* s obzirom na to da je on zadužen za komunikaciju i ja *Controllerom* i sa *Interactorom*. Ne spada u čistu arhitekturu zbog dvosmjerne komunikacije.

VIP (*Clean Swift*) – arhitektura izrađena kao implementacija Čiste arhitekture u iOS razvoju. Karakterizira je modularnost, razdvojenost slojeva i mogućnost pokrivanja testovima. Negativna strana je relativna kompleksnost i veći broj datoteka koje su potrebne za implementaciju.

Arhitektura je osnova svakog projekta. Zato je vrlo važno dobro razmotriti poslovne zahtjeve, resurse i vremenske okvire. Za izradu prototipa ili neke vrlo male aplikacije kao što je primjerice praćenje bilješki, MVC arhitektura bi mogla biti dobar izbor. Zamka leži u tome da je lako uočiti na jednostavnoj aplikaciji nešto što može bolje, nešto što bi se moglo dodati novo. Tako aplikacija počne rasti i sve negativne strane te arhitekture postaju sve izraženije. Bilo bi idealno u jednom trenutku odlučiti da ta arhitektura nije više zadovoljavajuća i da je treba promijeniti, ali to zahtjeva velike promjene koje mogu biti dugotrajne i skupe pa se od te ideje odustane.

MVP je arhitektura koja se može koristiti za male i srednje projekte jer je pokriće testovima i razdvajanje odgovornosti dovoljno dobro. Ipak, bilo bi je dobro izbjegavati jer iduće arhitekture daju dosta više prednosti.

MVVM arhitektura se također može koristiti na manjim i srednjim projektima. Prednosti su joj slične kao i MVP uz dodatnu prednost odvajanja modela za *View*. Time se osigurava prezentacijska logika.

VIPER arhitektura je relativno kompleksna arhitektura. Pruža odličnu skalabilnost i jednostavnost izmjene te odlično pokrivanje testovima. S druge strane, za srednje, a ponajviše za male projekte je često pretjerana, odnosno nije potrebna.

VIP je arhitektura slična VIPER-u. Najveća je razlika jednosmjeran tok podataka što omogućava najbolje pokrivanje testovima i podjelu odgovornosti. Idealna je, kao i VIPER za srednje i veće projekte dok za male projekte može biti nepotrebna.

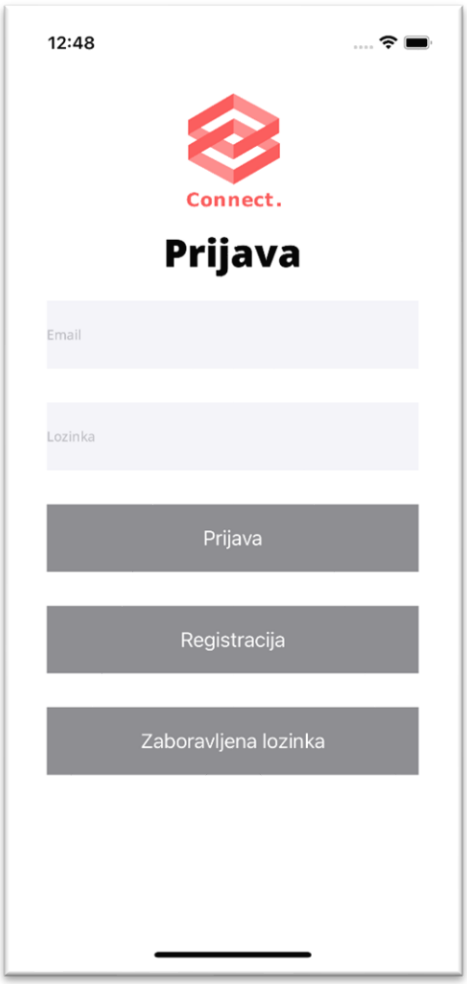
S obzirom na temu ovog rada, od svih arhitektura, pravilo čiste arhitekture zadovoljava jedino VIP arhitektura. Iz tog je razloga ona odabirana za izradu praktičnog primjera. U idućem poglavlju će biti napravljen pregled implementacije aplikacije *Connect* s čistom arhitekturom. Bit će prikazan detaljan izgled scene te jedinično testiranje te scene.

6. Praktičan dio

Do sada su u radu bile opisane različite arhitekture kod izrade iOS aplikacija, njihove prednosti i mane te mogućnosti testiranja. S obzirom na to da je tema rada primjena paradigme čiste arhitekture, u ovom će poglavlju biti prikazano upravo to. U ranijem poglavlju je navedeno kako je za praktičan dio odabrana VIP arhitektura, odnosno *Clean Swift* arhitektura. U ovom poglavlju bit će opisan praktičan primjer – izrada iOS aplikacije koristeći upravu tu arhitekturu.

Svrha ovo praktičnog dijela je prikazati način korištenja te prednosti i mane čiste arhitekture. Značajke aplikacije su također smišljene s tom namjerom. Cilj aplikacije je omogućiti studentima lakši pronalazak suradnika na projektima. Primjerice, student koji želi izraditi iOS aplikaciju nerijetko nailazi na problem izrade *backenda*. Isto tako, student koji želi izraditi *backend* aplikaciju, nailazi na problem *frontenda*, bilo to web ili mobilna aplikacija. Ova aplikacija omogućava kreiranje, ažuriranje, brisanje i pretragu različitih projekata. Korisnik može kreirati projekt, unijeti naziv i kratki opis te navesti što traži i što ima u obliku *tagova*. Primjerice, može se kreirati projekt „Izrada aplikacije za restoran“, opisa „Aplikacija je namijenjena izradi narudžbi i računa za restorane“ s *tagovima* potrebno je „*backend, web frontend*“ i *tagovima* postoji „*dizajn, ios“*. Drugi korisnici mogu pretraživati projekte po *tagovima*. Ako korisnik naiđe na projekt koji mu je interesantan, može poslati poruku vlasniku tog projekta unutar aplikacije. U aplikaciji je omogućen pregled svih poruka korisnika kao i izmjene podataka korisnika. Ovo su ukratko značajke aplikacije, a u nastavku će biti detaljnije opisano koje su sve značajke na pojedinim ekranima.

Aplikacija se sastoji od devet ekrana/scena. Kod pokretanja aplikacije, otvara se ekran za prijavu prikazan na slici 12.

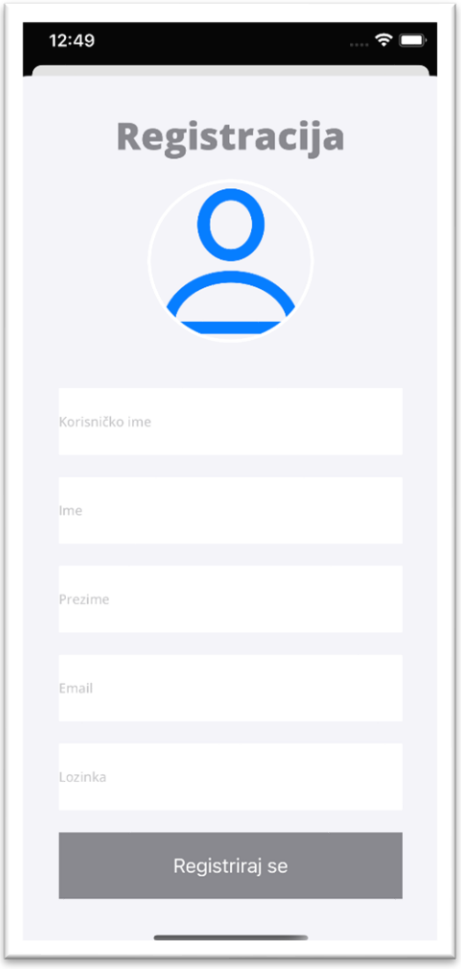


Slika 12 Prijava

Na tom ekranu se nalaze dva tekstualna polja i tri gumba. Korisnik može unijeti email i lozinku i pritisnuti gumb za prijavu kako bi se prijavio u aplikaciju (otvara se početni ekran nakon uspješne prijave). Klikom na gumb zaboravljena lozinka, otvara se tekstualno polje prikazano na slici 13.

Slika 13 Zaboravljena lozinka

U ovo polje korisnik može unijeti svoj email na koji će mu *Firebase* poslati poveznicu za oporavak lozinke. Zadnji je gumb registracije. Odabirom tog gumba, otvara se idući ekran za registraciju prikazan na slici 14.



Slika 14 Registracija

Na ovom ekranu korisnik može unijeti sve potrebne podatke uključujući i fotografiju. Pritiskom na gumb Registriraj se, korisnik se registrira te se nakon uspješne registracije otvara početni ekran koji je prikazan na slici 15.

Slika 15 Početna

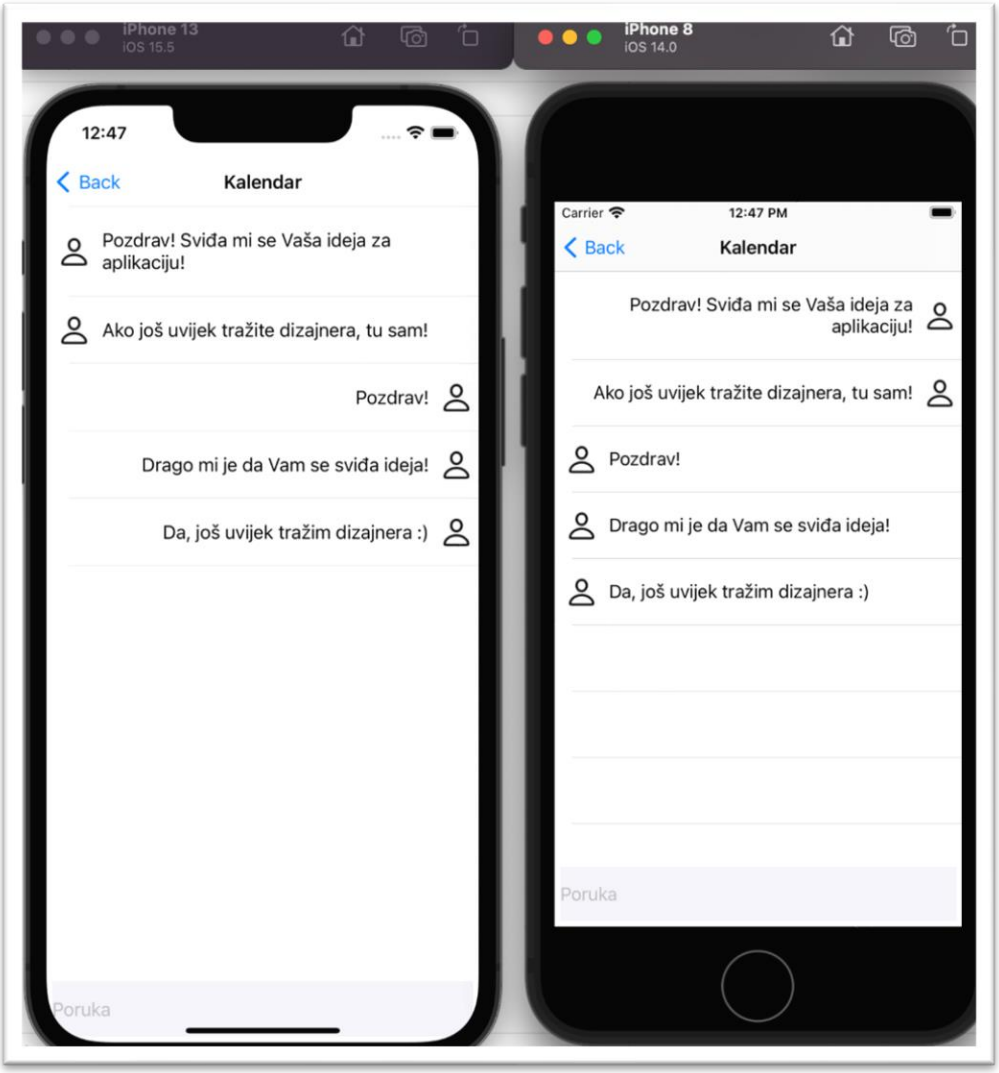
Otvaranjem početnog ekrana, na dnu je vidljiva navigacija. Ona se sastoji od ekrana Moji projekti, Početna, Razgovori i Profil. Na početnom ekranu se nalazi popis projekta. Korisnik može vidjeti naziv te *tagove* koji označavaju što vlasnik projekta traži. Osim toga, na vrhu ekrana se nalazi tražilica. Koristeći tražilicu, korisnik može filtrirati projekte po *tagovima*.

Pritiskom na bilo koji od projekata iz popisa na početnom ekranu, otvara se ekran s detaljima projekta prikazan na slici 16.

Slika 16 Detalji projekta

Na ovom ekranu su prikazani svi podatci o projektu: naziv projekta, podatci o vlasniku, opis te svi *tagovi*. Na tom ekranu se još nalazi i gumb za slanje poruke. Pritiskom na taj gumb, otvara se ekran detalja razgovora.

Ekran detalja razgovora prikazan je na slici 17. Na slici su dva različita simulatora koji su prijavljeni s različitim profilima.



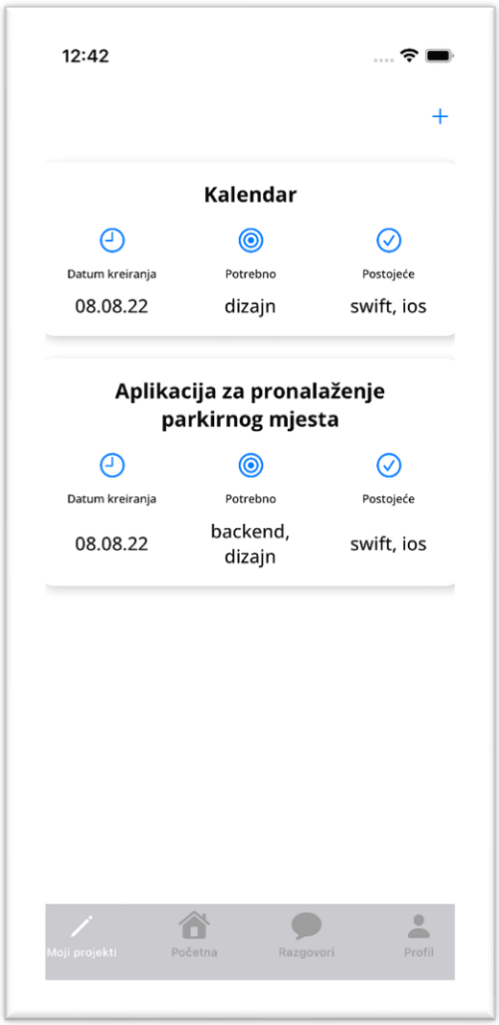
Slika 17 Detalji razgovora

Poruke trenutnog korisnika su centrirane desno, a poruke drugog korisnika lijevo. Ovaj ekran se ažurira u realnom vremenu. Omogućuje korisnicima komunikaciju o pojedinom projektu unutar aplikacije. Osim putem detalja projekta, do ekrana detalja razgovora se može doći i putem ekrana Razgovori prikazanog na slici 18 koji je dostupan u navigaciji.

Slika 18 Razgovori

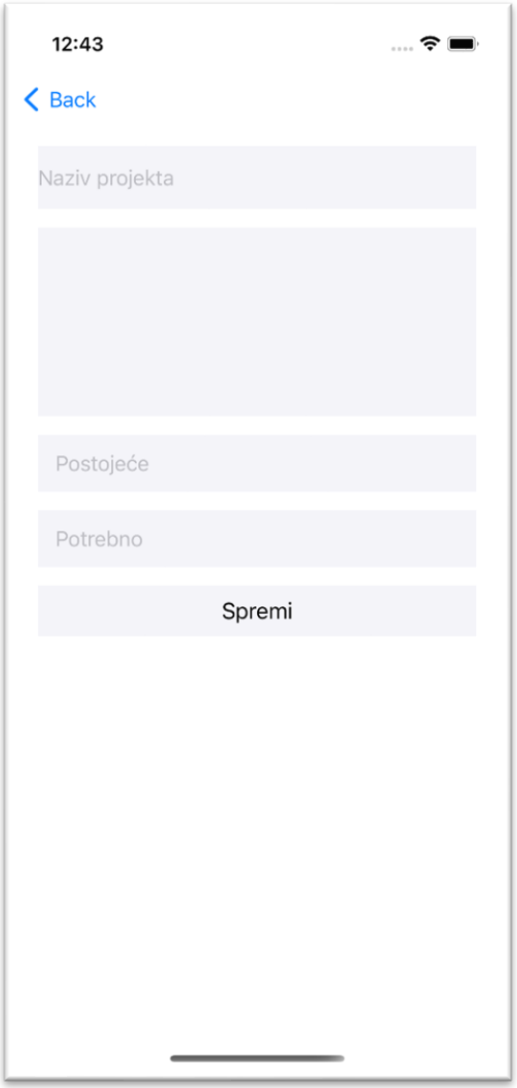
Na ovom ekranu se nalazi popis svih razgovora trenutno prijavljenog korisnika s ostalim korisnicima. Za svaki je razgovor prikazana profilna slika korisnika s kojim se razgovora, njegovo ime te zadnja poruka. Pritiskom na razgovor, otvara se ekran detalja razgovora.

Ekran Moji projekti koji se otvara iz navigacije, a prikazan je na slici 19, sadrži popis svih projekata prijavljenog korisnika te prikazuje informacije naziv projekta, datum objave te *tagove*. Povlačenjem retka moguće je obrisati projekt uz prikaz obavijesti.

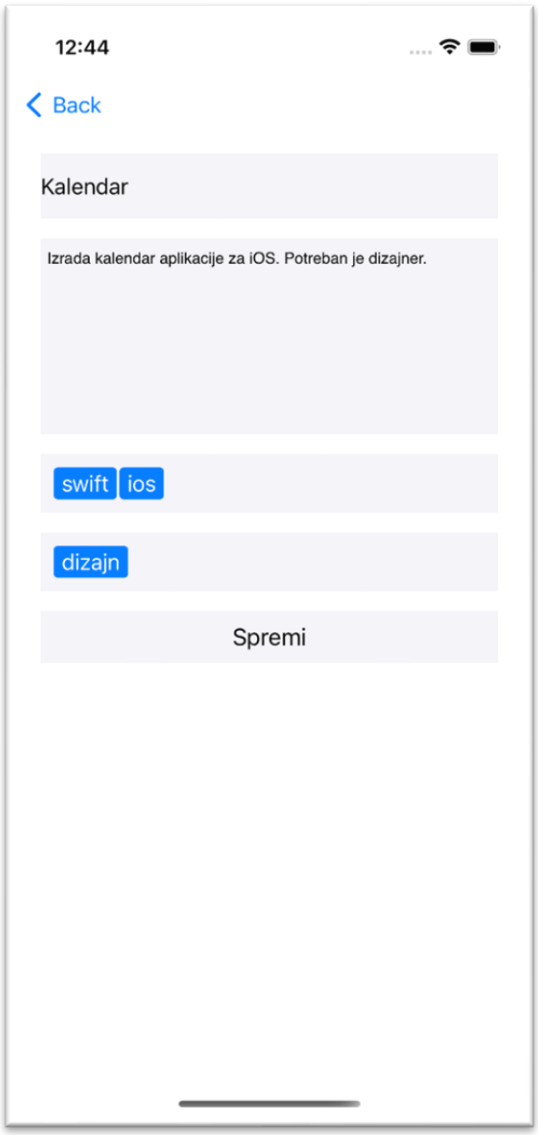


Slika 19 Moji projekti

U gornjem desnom uglu nalazi se gumb +. On otvara ekran za kreiranje i ažuriranje projekta što je prikazano na slici 20. Isti se ekran otvara ako korisnik klikne na bilo koji od projekata iz popisa na ekranu Moji projekti prikazanog na slici 21. Na njemu korisnik unosi naziv, opis te *tagove* potrebne za projekt. Klikom na gumb Spremi, u slučaju kreiranja novog projekta, on se kreira, a u slučaju ažuriranja, ažurira.



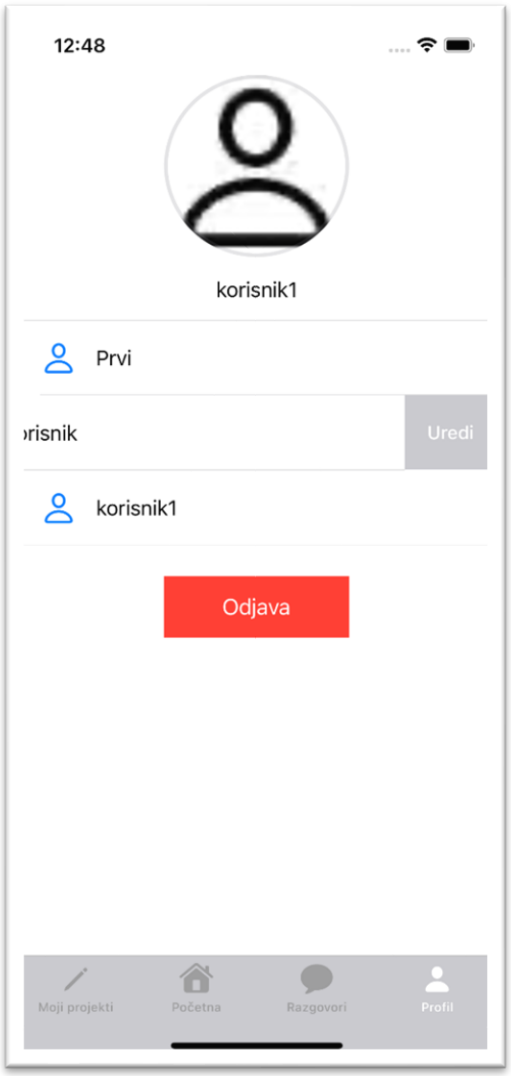
Slika 20 Kreiranje projekta



Slika 21 Uređivanje projekta

Razgovori je također ekran navigacije. Na njemu se nalazi popis svih razgovora korisnika s drugim korisnicima i ažurira se u realnom vremenu. Odabirom bilo kojeg od razgovora, otvara se ekran Razgovor (isti ekran koji se otvara kod odabira slanja poruke na ekranu detalji projekta). Ovdje korisnik vidi sve poruke između sebe i drugog korisnika. Ovaj se ekran podatcima osvježava također u realnom vremenu.

Putem navigacije dostupan je i Profil ekran prikazan na slici 22.



Slika 22 Profil

Na tom ekranu korisnik može vidjeti svoje podatke te ih izmijeniti povlačenjem retka. Klikom na ikonu se otvara prozor preko kojega korisnik može izmijeniti svoju fotografiju.

Ovo je opis aplikacije i njenih značajki sa stajališta korisnika. U nastavku će biti napravljen pregled baze podataka te konkretna implementacija.

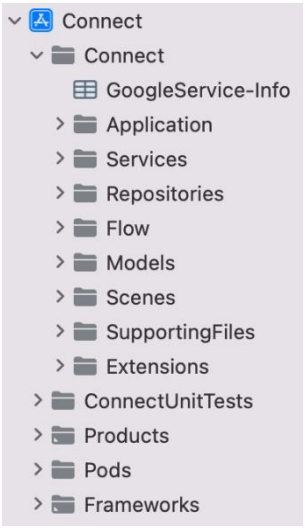
6.1. Baza podataka

Za bazu podataka odabran je *Firebase*. *Firebase* se sastoji od nekoliko dijelova od kojih su za potrebe ovog rada korišteni *Firebase Authentication, Firebase Database*, *Firebase Storage* te *Firebase Functions*. *Firebase Authentication* korišten je za kreiranje korisnika koristeći email i lozinku, prijavu korisnika te ponovno postavljanje lozinke. *Firebase Database* je korišten za bazu podataka. S obzirom na strukturu tog servisa, koristi se NoSQL baza podataka. U *Firebaseu* se podatci spremaju unutar kolekcija. U Kolekcije se spremaju dokumenti. Dokumenti sadrže polja koja mogu biti datum, tekst, lista, mapa, *boolean*, geografska lokacija ili referenca na neki drugi dokument. Postoji više načina na koje je moguće sistematizirati podatke u NoSQL bazi podataka. S obzirom na to da je cilj ovog rada prikazati korištenje arhitekture u iOS-u, baza podataka nije u potpunosti optimizirana. U bazi su tako spremljeni duplikati podataka (primjerice kolekcija *Users* sadrži podatke o korisnicima, a ti podatci su dupli cirani unutar dokumenata kolekcije *Projects*). Ovo je potencijalan problem, ali rješenje je u korištenju *Firebase Functions*-a. Pomoću njih moguće je presresti svaki poziv (primjerice ažuriranje podataka za dokument unutar kolekcije *Users*) te dodati ažuriranja podataka o tom korisniku na svim ostalim mjestima u bazi. Za spremanje fotografija koristi se *Firebase Storage*.

6.2. Implementacija

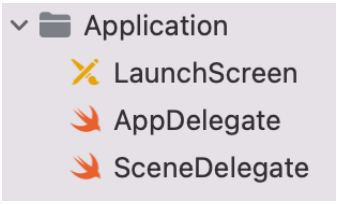
Do sada je bilo opisano koja je svrha aplikacije te *backend* koji podržava ovu aplikaciju. U nastavku će biti napravljen detaljan pregled same implementacije koristeći *Clean Swift* arhitekturu. U nadolazećim poglavljima bit će prikazane implementacije samo nekih od klasa kako bi se čitatelju stvorila jasna slika na koji način se implementira ova arhitektura. Cijela implementacija je dostupna na https://github.com/dinoMartan/diplomskiRad.

Za početak valja napraviti organizaciju datoteka unutar samog projekta. Na slici 23 vidljiva je ta organizacija. Za početak će biti opisana struktura datoteka i generalna uloga pojedinih dijelova, a kasnije će biti opisana i prikazana implementacija pojedinih klasa.



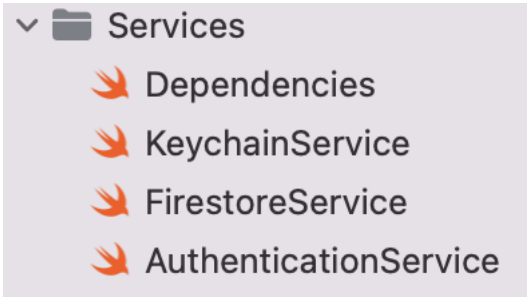
Slika 23 Organizacija datoteka – pregled

Na slici 23 možemo prvo uočiti *GoogleService-Info*. To je datoteka koju izdaje *Firebase* i koja se stavlja u projekt kako bi mogao komunicirati s tim servisom. Nakon toga se nalazi *Application* mapa prikazana na slici 24.



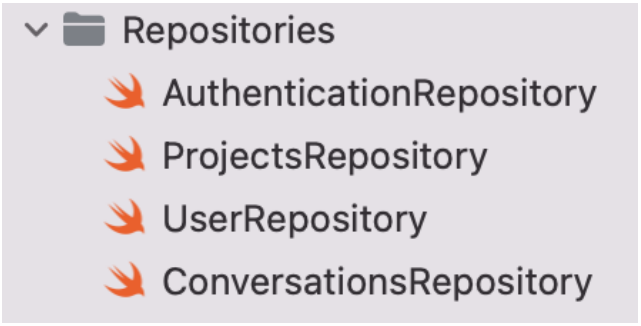
Slika 24 Organizacija datoteka – aplikacija

Unutar nje su smješteni *LaunchScreen* (izgled ekrana koji se prikazuje dok se aplikacija učitava) te *AppDelegate* i *SceneDelegate* klase. Te se klase pokreću kod pokretanja aplikacije i između ostalog, omogućavaju postavljanje početnog ekrana i instanciranje objekata koji se koriste kroz cijelu aplikaciju. Nakon toga, imamo *Services* mapu prikazanu na slici 25.



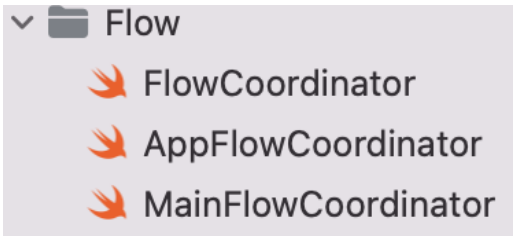
Slika 25 Organizacija datoteka – servisi

Ona sadrži *FirestoreServices* koji implementira metode koje koriste vanjsku biblioteku *Firebasea*. Mapa također sadrži i *KeychainServices* koji pak implementira metode za sigurno lokalno spremanje podataka kao što su ID korisnika koristeći vanjsku biblioteku *KeychainSwift*. Zadnji servis je *AuthenticationService* koji koristi *FirebaseAuth* biblioteku i koristi se za prijavu i kreiranje korisnika. U mapi se još nalazi i datoteka *Dependencies* koja zapravo na jednom mjestu sadrži *FirestoreServices*, *KeychainServices* i *AuthenticationService*. Iduća je mapa *Repositories* prikazana na slici 26.



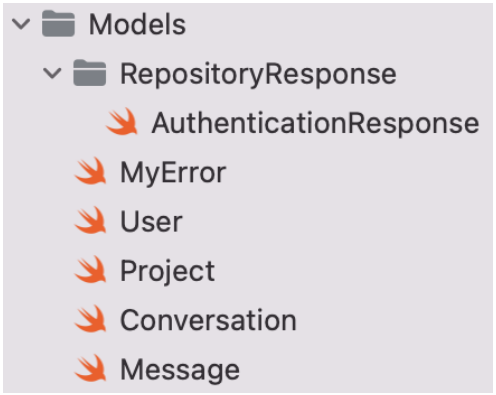
Slika 26 Organizacija datoteka - repozitoriji

Ona sadrži sve klase koje koriste *FirestoreService* kako bi pozivale *Firebase* API. Repozitoriji čine sloj između servisa i poslovne logike. *Flow* mapa prikazana na slici 27 sadrži glavne elemente koordinatora (eng. *Coordinator*).



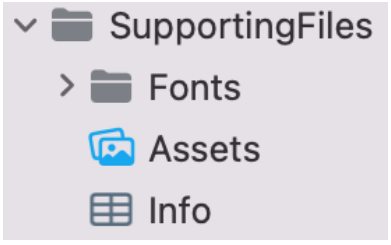
Slika 27 Organizacija datoteka – tok

Koordinatori su sa zaduženi za navigaciju između scena te oni instanciraju sve potrebne elemente scena. *Models* mapa prikazana na slici 28 sadrži sve potrebne modele.



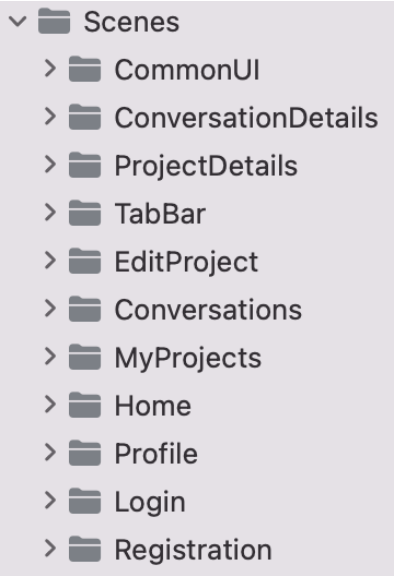
Slika 28 Organizacija datoteka – modeli

S obzirom na to da se u ovoj konkretnoj aplikaciji *backend* gotovo pa implementira unutar same aplikacije, koriste se isti modeli i za *backend* i za samu aplikaciju. U slučaju da bi bili korišteni neki drugi API servisi, bilo bi poželjno kreirati posebne modele za podatke koje vraća API te ih u repozitorij klasama mapirati u modele aplikacije. Iduća je mapa *SupportingFiles* prikazana na slici 29.



Slika 29 Organizacija datoteka – pomoćne datoteke

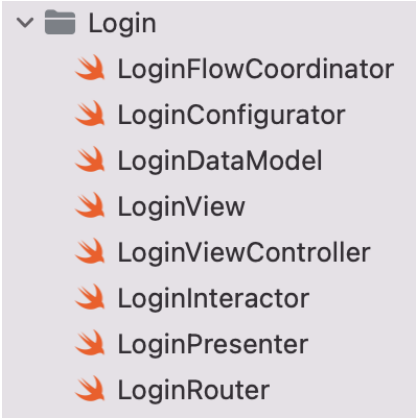
Ona sadrži popratne elemente aplikacije kao što su fontovi i fotografije. U ovoj aplikaciji koristi se *OpenSans* font. U *Extensions* mapi se nalaze ekstenzije. To su najčešće ekstenzije klasa koje pruža *Swift*. I za kraj, najvažniji dio iz pogleda arhitekture, *Scenes* mapa prikazana na slici 30.



Slika 30 Organizacija datoteka – scene

Unutar ove mape se nalaze mape za devet scena, mapa za zajedničke elemente korisničkog sučelja (primjerice gumb) te *TabBar* mapa koja sadrži *TabBar* koji sam po sebi nije scena odnosno nema svoje korisničko sučelje, ali služi za navigaciju i instanciranje scena prikazanih na traci navigacije u aplikaciji (Moji projekti, Početna, Razgovori i Profil scene).

Arhitektura nalaže razdvajanje ekrana jer se najčešće za različite ekrane koriste različiti podatci i poslovna logika. U ovoj aplikaciji je devet ekrana koji su podijeljeni u devet scena. Organizacija jedne scene (scene prijave) prikazan je na slici 31.



Slika 31 Organizacija datoteka – scena Prijava

Svaka scena se sastoji od `FlowCoordinator`, `Configurator`, `DataModel`, `View`, `ViewController`, `Interactor`, `Presenter` i `Router` klase. Ponekad je potrebno dodati još pokoju klasu. Najčešće su to klase koje sadrže elemente korisničkog sučelja (primjerice redak tablice). S obzirom na to da se radi o relativno velikom broju datoteka koje je potrebno izraditi za svaku scenu, *Xcode* pruža mogućnost korištenja predložaka (eng. *Templates*). Predlošci omogućavaju kreiranje datoteka po pre definiranom izgledu što može ubrzati proces kreiranja nove scene.

Do sada je u ovom poglavlju napravljen pregled što aplikacija treba raditi i na koji način su klase organizirane u smislu strukture datoteka. U nastavku slijedi detaljan pregled pojedinih dijelova te prikaz implementacija. Za početak će biti napravljen pregled servisa te implementacija jednog od servisa. Nakon toga će na isti način biti predstavljeni repozitoriji te implementacija navigacije. Nakon toga će biti napravljen detaljan pregled jedne scene. To uključuje implementaciju svih pojedinačnih klasa scene. Na kraju će na sličan način biti prikazani jedinični testovi. S obzirom na to da su servisi međusobno slični u smislu načina implementacije, kao što su i repozitoriji međusobno slični, a posebice scene, u nadolazećim poglavljima neće biti prikazane implementacije svih klasa, ali su dostupne u izvornom kôdu.

6.2.1. Servisi

Servise možemo gledati kao most između aplikacije i vanjskih biblioteka. Oni sadrže i implementiraju protokole. Unutar klase `Dependencies` su sadržani svi servisi aplikacije. S obzirom na to da se radi o klasama, prosljeđuje se samo referenca što je optimalno za memoriju (izbjegava se instanciranje objekata svaki put kad je potreban servis).

Uzmimo za primjer servis `KeychainService` prikazan na programskom kôdu 20. Unutar njega se nalazi protokol `KeychainServiceProtocol` te klasa `KeychainService` koja implementira taj protokol.

```
import Foundation
import KeychainSwift

protocol KeychainServiceProtocol {
    func getUserLoggedIn() -> Bool
    func setUserLoggedIn(_ userLoggedIn: Bool)

    func getUserId() -> String?
    func setUserId(_ userId: String?)
}

class KeychainService: KeychainServiceProtocol {
    private let keychainSwift: KeychainSwift

    private enum Key: String {
        case userLoggedIn
        case userId
    }

    init() {
        keychainSwift = KeychainSwift()
    }
}

extension KeychainService {
    func getUserLoggedIn() -> Bool {
```

```
        keychainSwift.getBool(Key.userLoggedIn.rawValue) ?? false
    }

    func setUserLoggedIn(_ userLoggedIn: Bool) {
        keychainSwift.set(userLoggedIn, forKey: Key.userLoggedIn.rawValue)
    }
}

extension KeychainService {
    func getUserId() -> String? {
        keychainSwift.get(Key.userId.rawValue)
    }

    func setUserId(_ userId: String?) {
        guard let userId = userId else { return }
        keychainSwift.set(userId, forKey: Key.userId.rawValue)
    }
}
```

Programski kôd 20 KeychainService

Klasa `KeychainService` sadrži klasu `KeychainSwift` koja je dio vanjske biblioteke. Vidimo da su implementirane sve metode protokola i sve implementacije koriste `KeychainSwift`. U slučaju da autori biblioteke prestanu održavati tu biblioteku ili iz bilo kojeg razloga odlučimo da nam ova biblioteka više nije najbolji izbor, bilo bi potrebno samo u ovom servisu zamijeniti `KeychainSwift` biblioteku s nekom drugom (ili klasom vlastite implementacije). Time je osigurana neovisnost ostatka aplikacije od biblioteke jer se biblioteka koristi samo unutar servisa. Svi ostali dijelovi aplikacije imaju pristup samom servisu koji se u slučaju izmjene biblioteke, ne mijenja.

Vrlo slični su i ostali servisi. Servis `FirestoreService` implementira `FirestoreServiceProtocol`. Ovaj servis koristi dvije biblioteke, `FirbaseFireStore` i `FirebaseStorage`. `AuthenticationService` implementira `AuthenticationServiceProtocol` te koristi `FirebaseAuth` biblioteku.

`KeychainService` servis koriste klase koje implementiraju poslovnu logiku dok `AuthenticationService` i `FirebaseService` koriste samo klase repozitorija. Servise mogu koristiti *Interactor* klase ili repozitoriji.

6.2.2. Repozitoriji

Repozitorij klase se nalaze između servisa koji komuniciraju s bazom podataka i poslovne logike. U ovoj aplikaciji klase repozitorija koriste `AuthenticationService` i `FirebaseService`. Po izgledu su repozitoriji vrlo slični servisima. Njihovu implementaciju, kao i kod servisa, definiraju protokoli. No, glavna je razlika u tome što repozitoriji koriste servise, a ne vanjske biblioteke. O važnostima servisa bilo je riječi u ranijem poglavlju. U repozitorijima su definirane putanje do dokumenata u bazi te modeli koje sloj poslovne logike može očekivati. Uzmimo za primjer `ProjectsRepository` prikazan na programskom kôdu 21.

```
import Foundation

protocol ProjectsRepositoryProtocol {
    func getProject(projectId: String, completion: @escaping
((Result<Project, MyError>) -> Void))
    func setProject(project: Project, completion: @escaping ((Result<Void,
MyError>) -> Void))
    func deleteProject(projectId: String, completion: @escaping
((Result<Void, MyError>) -> Void))
    func getAllProjects(completion: @escaping ((Result<[Project], MyError>)
-> Void))
    func getProjectsWithNeedFor(_ need: String, completion: @escaping
((Result<[Project], MyError>) -> Void))
    func getProjectsForUser(_ userId: String, completion: @escaping
((Result<[Project], MyError>) -> Void))
}

class ProjectsRepository: ProjectsRepositoryProtocol {
    private let firestoreService: FirestoreServiceProtocol

    init(firestoreService: FirestoreServiceProtocol) {
        self.firestoreService = firestoreService
    }

    deinit {
        print("deinit \(self)")
```

```
    }

    func getProject(projectId: String, completion: @escaping
((Result<Project, MyError>) -> Void)) {
        firestoreService.getDocument(documentPath: "projects/" + projectId,
completion: completion)
    }
    …
}
```

Programski kôd 21 ProjectsRepository

On implementira `ProjectsRepositoryProtocol` koji definira sve potrebne radnje nad projektima. To uključuje kreiranje projekata, dohvaćanje svih projekata, dohvaćanje projekata po željenim *tagovima,* dohvaćanje projekata za specifičnog korisnika te brisanje projekata. U programskom kôdu iznad je prikazana implementacija jedne od metoda, metode `getProjects()`. Na isti način su implementirane i ostale metode (uz razliku u metodama servisa koje pozivaju i vrijednostima putanje dokumenata). Metode pozivaju servis s putanjom do dokumenta. U slučaju izmjene baze podataka (primjerice potrebno je koristiti neki novi API), bilo bi potrebno dodati još jedan sloj, primjerice API sloj između servisa i repozitorija koji bi putem servisa dohvaćao podatke u obliku modela baze podataka. Repozitorij bi u tom slučaju koristio taj API sloj i mapirao modele baze podataka i modele aplikacije. No, zbog specifičnosti baze podataka u ovoj aplikaciji, taj dio nije potreban.

Preostali repozitoriji su `AuthenticationRepository`, `UserRepository` te `ConversationsRepository`. Oni isto tako koriste servise i vraćaju modele aplikacije.

6.2.3. Navigacija

Nakon što imamo servise i repozitorije koji omogućavaju komunikaciju s bazom podataka ili bibliotekama, potrebno je odlučiti na koji način će svi elementi biti međusobno povezani. Čista arhitektura nalaže da *Router* bude zadužen za instanciranje novih scena odnosno navigaciju. S obzirom na to da *ViewController* sadrži *Router* koji bi onda sadržavao idući *ViewController*, to izgleda dosta nepraktično jer se kreira scena u sceni. Iz tog razloga je u ovom radu korišten koordinator *(eng. Coordinator*) uzorak. Koordinator je zadužen za instanciranje svih komponenti scena te kreiranje koordinatora drugih scena. Konkretno u ovoj aplikaciju se koristi `FlowCoordinator` protokol prikazan programskom kôdom 22.

```
import UIKit

protocol FlowCoordinator: AnyObject {
    var childFlowCoordinators: [FlowCoordinator] { get set }
    var dependencies: DependenciesProtocol { get set }
    var rootViewController: UINavigationController { get set }
    func start()
}

extension FlowCoordinator {
    func addChildFlowCoordinator(_ flowCoordinator: FlowCoordinator) {
        for element in childFlowCoordinators {
            if element === flowCoordinator { return }
        }
        childFlowCoordinators.append(flowCoordinator)
    }

    func removeChildFlowCoordinator(_ flowCoordinator: FlowCoordinator?) {
        guard childFlowCoordinators.isEmpty == false, let coordinator =
flowCoordinator else { return }

        for (index, element) in childFlowCoordinators.enumerated() {
            if element === coordinator {
                childFlowCoordinators.remove(at: index)
                break
            }
        }
```

```
    }
}
```

Programski kôd 22 FlowCoordinator

U njemu je zadan popis drugih koordinatora, referenca na ovisnosti, početni kontroler te `start()` metoda. Uz to, `FlowCoordinator` protokol ima definirane metode za dodavanje i brisanje koordinatora iz popisa koordinatora. Koordinator je klasa koja je zadužena za navigaciju. Ona instancira *Controller* i, koristeći statičnu metodu konfiguratora, sve ostale elemente scene. *Coordinator* također implementira protokol izlaza *Routera*. Na primjeru scene prijave to je `LoginRouterOutput` protokol. Temelju tog izlaza može kreirati novi *Coordinator* i prikazati iduću scenu spremajući taj novi *Coordinator* u listu `childFlowCoordinators` ili, ako je sam dijete nekog *Coordinatora,* može tražiti od svog roditelja *Coordinatora* da ga zatvori i obriše iz svoje liste. Promotrimo li `LoginFlowCoordinator`, prikazan na programskom kôdu 23, vidjet ćemo da je taj koordinator zadužen točno za to.

```
import UIKit

class LoginFlowCoordinator: FlowCoordinator {
    var childFlowCoordinators = [FlowCoordinator]()
    var rootViewController: UINavigationController
    internal var dependencies: DependenciesProtocol

    weak var delegate: LoginFlowCoordinatorDelegate?

    init(rootViewController: UINavigationController, dependencies:
DependenciesProtocol) {
        self.rootViewController = rootViewController
        self.dependencies = dependencies
    }

    func start() {
        startLoginFlow()
    }

    deinit {
```

```
        print("deinit \(self)")
    }
}

extension LoginFlowCoordinator {
    private func startLoginFlow() {
        let authenticationRepository = AuthenticationRepository()
        let loginViewController = LoginViewController()
        LoginConfigurator.configureModule(
            loginRouterOutput: self,
            viewController: loginViewController,
            keychainService: dependencies.keychainService,
            authenticationRepository: authenticationRepository)
        rootViewController.setViewControllers([loginViewController],
animated: true)
    }
}

extension LoginFlowCoordinator: LoginRouterOutput {
    func showRegistration() {
        delegate?.showRegistration()
    }

    func shouldClose() {
        delegate?.shouldRemoveFlowCoordinator(self)
    }

    func showMainFlow() {
        delegate?.showMainFlow()
    }
}
```

Programski kôd 23 LoginFlowCoordinator

Ovaj koordinator instancira potrebne repozitorij klase koristeći servise iz svoje `Dependencies` klase, kreira `LoginViewController` i `AuthenticationRepository` te koristi `LoginConfigurator` za povezivanje svih elemenata. Također implementira i `LoginRouterOutput` protokol te je time delegat `LoginRouteru`. Sam koordinator je dijete

`LoginFlowCoordinatora` koji mu je delegat. Na ovaj način instanciranje kontrolera i ostalih dijelova nije unutar *Routera* i neovisno je o implementaciji scena. Svaka scena sadrži svoj koordinator koji može prikazati bilo koju drugu scenu.

6.2.4. Scene

Od svih dijelova, ovo je najvažniji dio arhitekture i zapravo arhitektura najviše određuje izgled ovog dijela. Čista arhitektura nalaže kružni tok podataka. Iz korisničkog sučelja odnosno *Controllera* u sloj poslovne logike (*Interactor*), iz njega u *Presenter* koji formatira i šalje podatke u *View* koji ih može prikazati. Značajke odnosno funkcionalnosti možemo tretirati kao akcije. Te akcije se sastoje od tri dijela: zahtjev (eng. *Request*) koji pokreće akciju (*Controller -> Interactor*), zatim odgovora (eng. *Response*) koji nastaje nakon obrade podataka i poslovne logike (*Interactor -> Presenter*) i na kraju modela razumljivog za korisničko sučelje (eng. *View Model*) (*Presenter -> Controller*). Na ovaj način organizirani podatci daju jasnu sliku što se događa u sceni. Također se pruža mogućnost jednostavne izmjene. Pogledajmo primjer modela podataka `Login` prikazanog programskim kôdom 24.

```
import Foundation

struct Login { }

// MARK: LoginAction
extension Login {
    struct LoginAction {
        struct Request {
            let email: String
            let password: String
        }

        struct Response: Equatable {
            struct Success: Equatable { }

            struct Failure: Equatable {
                let myError: MyError?
            }
        }
```

```
        struct ViewModel {
            struct Success: Equatable { }

            struct Failure: Equatable {
                let myError: MyError?
            }
        }
    }
}

// MARK: ForgottenPasswordAction
extension Login {
    struct ForgottenPasswordAction {
        struct Request {
            let email: String
        }

        struct Response: Equatable {
            struct Success: Equatable { }

            struct Failure: Equatable {
                let myError: MyError?
            }
        }

        struct ViewModel {
            struct Success: Equatable {
                let title: String
                let message: String
            }

            struct Failure: Equatable {
                let myError: MyError?
            }
        }
    }
}
```

Programski kôd 24 Login

Vidimo da na ovoj sceni postoje dvije akcije: akcija za prijavu (`LoginAction`) te akcija za zaboravljenu lozinku (`ForgottenPasswordAction`). Akcija za prijavu započinje zahtjevom u kojem su email i lozinka. Rezultat poslovne logike se pak sastoji od dva dijela: uspjeh (`Success`) i neuspjeh (`Failure`). Valja napomenuti da se za neuspjeh odnosno grešku koristi model `MyError` zato što je u ekstenziji *Controllera* definirana metoda koja prima taj model pa svaki *Controller* može na isti način procesirati grešku. U slučaju dodavanja novih vrsta grešaka za koje je potreban drugi način procesiranja, nije potrebno mijenjati svaki *Controller* već samo ekstenziju. Podatci pripremljeni za korisničko sučelje se također sastoje od dva dijela, uspjeha i neuspjeha. Na vrlo sličan način je organiziran i tok podataka za akciju zaboravljene lozinke. Iz korisničkog sučelja se šalje email, rezultat poslovne logike su uspjeh ili neuspjeh. *Presenter* u slučaju uspjeha priprema naslov i poruku koju šalje *Controlleru* odnosno *Viewu* koju on prikazuje.

Organizacija podataka je prvi dio scene. Sada kada znamo koje su akcije i koji se podatci očekuju, možemo krenuti na implementaciju ostalih dijelova. Nastavimo promatrati scenu prijave. Za početak ćemo uzeti korisničko sučelje odnosno par *Controller* i *View*. Na primjeru scene prijave, `LoginView` sadrži samo elemente korisničkog sučelja. Tu se nalazi slika, naslov, dva polja za unos teksta te tri gumba. Uz to, tu su i akcije za pritisak gumba. Svi elementi korisničkog sučelja su raspoređeni koristeći *SnapKit* biblioteku. Korisničko sučelje može biti izrađeno programskim putem ili pak putem vizualnog editora (u obliku *Storyboarda*). Idući je *Controller* odnosno `LoginViewController` čija je implementacija prikazana na programskom kôdu 25. On sadrži `LoginView`, `LoginInteractorProtocol` te `LoginRouterProtocol`. O druga dva dijela će biti riječi malo kasnije.

```
import UIKit

protocol LoginPresenterOutput: AnyObject {
    func presenter(didSucceedLogin viewModel:
Login.LoginAction.ViewModel.Success)
    func presenter(didFailLogin viewModel:
Login.LoginAction.ViewModel.Failure)
    func presenter(didSucceedForgottenPassword viewModel:
Login.ForgottenPasswordAction.ViewModel.Success)
    func presenter(didFailForgottenPassword viewModel:
Login.ForgottenPasswordAction.ViewModel.Failure)
}
```

```
class LoginViewController: UIViewController {
    var loginView: LoginView?
    var interactor: LoginInteractorProtocol?
    var router: LoginRouterProtocol?

    override func loadView() {
        super.loadView()
        self.view = loginView
    }

    override func viewDidLoad() {
        super.viewDidLoad()
        setupInteractions()
    }

    deinit {
        router?.shouldClose()
        print("deinit \(self)")
    }
}


extension LoginViewController {
    private func setupInteractions() {
        loginView?.loginButtonTapInteraction = { [weak self] in
            guard let email =
self?.loginView?.emailTextFieldView.textField.text,
                  let password =
self?.loginView?.passwordTextFieldView.textField.text
            else { return }
            self?.interactor?.loginUser(request:
Login.LoginAction.Request(email: email,

password: password))
        }

        loginView?.registerButtonTapInteraction = { [weak self] in
            self?.router?.showRegistration()
        }
```

```
        loginView?.forgottenPasswordButtonTapInteraction = { [weak self] in
            self?.handleForgottenPasswordButtonTap()
        }
    }

    private func handleForgottenPasswordButtonTap() {
        showAlertControllerWithTextField(title: "Zaboravljena lozinka",
                                         message: nil,
                                         placeholder: "Email") { [weak
self] email in
            guard let email = email else { return }
            self?.interactor?.forgottenPassword(request:
Login.ForgottenPasswordAction.Request(email: email))
        }
    }
}

extension LoginViewController: LoginPresenterOutput {
    func presenter(didSucceedLogin viewModel:
Login.LoginAction.ViewModel.Success) {
        router?.showMainFlow()
    }

    func presenter(didFailLogin viewModel:
Login.LoginAction.ViewModel.Failure) {
        showMyErrorAlert(viewModel.myError)
    }

    func presenter(didSucceedForgottenPassword viewModel:
Login.ForgottenPasswordAction.ViewModel.Success) {
        showAlert(title: viewModel.title,
                  message: viewModel.message)
    }

    func presenter(didFailForgottenPassword viewModel:
Login.ForgottenPasswordAction.ViewModel.Failure) {
        showMyErrorAlert(viewModel.myError)
    }
}
```

Programski kôd 25 LoginViewController

`LoginViewController` postavlja reakcije na pritiske gumbova koji se nalaze na `LoginViewu`. S obzirom na to da sa *Controllerom* počinje i završava krug toka podataka u čistoj arhitekturi, on je zadužen za započinjanje i završavanje tog kruga. Akcija započinje primjerice pritiskom gumba prijava. Zatim *Controller* kreira zahtjev za tu akciju koji je prema modelu podataka tipa `Login.LoginAction.Request`. S tim zahtjevom poziva metodu `LoginInteractora loginUser()`. Završetak kruga su podatci koje *Presenter* pošalje *Controlleru*. Oni su definirani protokolom (u ovom slučaju to je `LoginPresenterOutput` protokol). S obzirom na to da svaka od akcija u ovoj sceni može biti uspješna ili neuspješna, `LoginPresenterOutput` protokol sadrži po dvije metode za svaku akciju. Svaka od tih metoda prima poseban `ViewModel`. `LoginViewController` implementira taj protokol i na temelju tih podataka u obliku `ViewModela` može ažurirati korisničko sučelje, pozvati `LoginRouter` ili pak započeti neku drugu akciju. U ovom primjeru, ako je prijava uspješna, `LoginViewController` traži od `LoginRoutera` da prikaže glavni tok aplikacije. U slučaju da je akcija izmjene lozinke uspješna se prikazuje obavijest s naslovom i podnaslovom. Ako je bilo koja od akcija neuspješna, prikazuje se greška u obliku obavijesti.

Interactor obavlja svu poslovnu logiku. *Interactor* je definiran protokolom. On može koristiti servise i repozitorije te dohvaća modele aplikacije koje prosljeđuje *Presenteru* unutar modela odgovora. Sve njegove metode uvijek primaju parametar zahtjeva. Pogledajmo `LoginInteractor` prikazan programskim kôdom 26.

```
import Foundation

protocol LoginInteractorProtocol: AnyObject {
    func loginUser(request: Login.LoginAction.Request)
    func forgottenPassword(request: Login.ForgottenPasswordAction.Request)
}

class LoginInteractor: LoginInteractorProtocol {
    var presenter: LoginPresenterProtocol?

    private let keychainService: KeychainServiceProtocol
    private let authenticationRepository: AuthenticationRepositoryProtocol
```

```
    init(keychainService: KeychainServiceProtocol,
authenticationRepository: AuthenticationRepositoryProtocol) {
        self.keychainService = keychainService
        self.authenticationRepository = authenticationRepository
    }

    deinit {
        print("deinit \(self)")
    }
}

extension LoginInteractor {
    func loginUser(request: Login.LoginAction.Request) {
        authenticationRepository.signInUser(email: request.email, password:
request.password) { [weak self] result in
            switch result {
            case .success(let authenticationResponse):
                self?.keychainService.setUserLoggedIn(true)
                self?.keychainService.setUserId(authenticationResponse.user
Id)
                self?.presenter?.interactor(didSucceedLogin:
Login.LoginAction.Response.Success())
            case .failure(let myError):
                self?.presenter?.interactor(didFailLogin:
Login.LoginAction.Response.Failure(myError: myError))
            }
        }
    }

    func forgottenPassword(request: Login.ForgottenPasswordAction.Request)
{
        authenticationRepository.sendResetPasswordEmail(email:
request.email) { [weak self] result in
            switch result {
            case .success(_):
                self?.presenter?.interactor(didSucceedForgottenPassword:
Login.ForgottenPasswordAction.Response.Success())
            case .failure(let myError):
```

```
                self?.presenter?.interactor(didFailForgottenPassword:
Login.ForgottenPasswordAction.Response.Failure(myError: myError))
            }
        }
    }
}
```

Programski kôd 26 LoginInteractor

Akcija prijave korisnika rekli smo započinje tako da `LoginViewController` kreira zahtjev `Login.LoginUserAction.Request` te s tim zahtjevom poziva metodu `LoginInteractora loginUser()`. `LoginInteractor` u toj metodu pak poziva repozitorij i na temelju odgovora repozitorija, poziva metode `LoginPresentera` i eventualno servisa. Kod poziva metoda `LoginPresentera`, prvo kreira objekt odgovora koji je za uspješno obavljen zahtjev u ovom primjeru `Login.LoginUserAction.Response.Success` te s njime poziva metodu `LoginPresentera`.

Presenter je također definiran protokolom. On mapira odnosno pretvara odgovor koji dobije od *Interactora* u model koji je razumljiv korisničkom sučelju: *ViewModel*. On sadrži referencu na *Controller* u obliku delegata. Pogledajmo primjer `LoginPresentera` prikazanog na programskom kôdu 27.

```
import Foundation

protocol LoginPresenterProtocol: AnyObject {
    func interactor(didSucceedLogin response:
Login.LoginAction.Response.Success)
    func interactor(didFailLogin response:
Login.LoginAction.Response.Failure)
    func interactor(didSucceedForgottenPassword response:
Login.ForgottenPasswordAction.Response.Success)
    func interactor(didFailForgottenPassword response:
Login.ForgottenPasswordAction.Response.Failure)
}

class LoginPresenter: LoginPresenterProtocol {
    weak var viewController: LoginPresenterOutput?

    deinit {
        print("deinit \(self)")
    }

    func interactor(didSucceedLogin response:
Login.LoginAction.Response.Success) {
        viewController?.presenter(didSucceedLogin:
Login.LoginAction.ViewModel.Success())
    }

    func interactor(didSucceedForgottenPassword response:
Login.ForgottenPasswordAction.Response.Success) {
        let title = "Email poslan"
        let message = "Email za reset lozinke je poslan na Vašu email
adresu"
        viewController?.presenter(didSucceedForgottenPassword:
Login.ForgottenPasswordAction.ViewModel.Success(title: title,

                                    message: message))
    }
```

```
    func interactor(didFailLogin response:
Login.LoginAction.Response.Failure) {
        viewController?.presenter(didFailLogin:
Login.LoginAction.ViewModel.Failure(myError: response.myError))
    }

    func interactor(didFailForgottenPassword response:
Login.ForgottenPasswordAction.Response.Failure) {
        viewController?.presenter(didFailForgottenPassword:
Login.ForgottenPasswordAction.ViewModel.Failure(myError: response.myError))
    }
}
```

Programski kôd 27 LoginPresenter

On sadrži delegat tipa `LoginPresenterOutput` koji je definiran na `LoginViewControlleru`. Pogledamo li primjer, specifično metodu *`interactor(didSucceedLogin…)`*, vidimo da `LoginPresenter` prima odgovor te kreira poseban model definiran kao `Login.LoginUserAction.ViewModel.Success` te njega šalje delegatu, odnosno `LoginViewControlleru`.

Time je zatvoren krug toka podataka. Na ovaj način, podatci putuju intuitivno i predvidivo što ovelike olakšava bilo kakve izmjene. Na primjer, ako želimo da se nakon uspješne prijave prikaže poruka, dovoljno je u model `Login.LoginAction.ViewModel.Success` dodati *String*, u `LoginPresenteru` implementirati taj novi model i u `LoginViewControlleru` prikazati tekst iz *ViewModela*.

Ranije je spomenuto da *Controller* sadrži još i *Router*. *Router* je definiran protokolom i služi komunikaciji scene s navigacijom, odnosno koordinatorom. Putem njega *Controller* može od koordinatora tražiti da prikaže novu scenu ili poslati podatke u neku prethodnu.

Na kraju je tu konfigurator. Ovo zapravo nije dio arhitekture i sav kôd konfiguratora može biti unutar *Controllera*. No, s obzirom na to da se radi o instanciranju i međusobnom spajanju elemenata, u ovom je primjeru to odvojeno. Sadrži se od jedne statične metode. Iz primjera se mogu vidjeti veze između pojedinih elemenata.

Na ovaj su način kreirane sve scene u ovom praktičnom primjeru. Model podataka koji kružnim tokom prolazi kroz različite slojeve arhitekture su ključan dio koji nastoji očuvati *Dependency Rule*. Kako projekt raste, uvode se novi ljudi, često postaje problematično uvođenje na projekte. No, ako se kroz projekt koristi ovakav pristup, bilo tko tko je upoznat

samo s arhitekturom, ali ne i radom aplikacije, može brzo shvatiti što se u aplikaciji događa. Za to je prvenstveno zaslužan model podataka specifičan za svaku scenu kojim se odvajaju akcije. Do sada je već više puta spominjana jednostavnost izmjene korištenjem ove arhitekture. Osim jednostavnosti izmjene, arhitektura omogućava i sigurnost izmjene u obliku jediničnih testova. S obzirom na to da je većina elemenata arhitekture definirana protokolima, ti se elementi mogu *mockati*. Uzmemo li to u obzir i činjenicu da su elementi raslojeni odnosno raspodjela odgovornosti je precizna, za sigurnost izmjena se možemo osloniti na jedinične testove. U idućem će poglavlju biti napravljen pregled implementacije jediničnih testova u *Clean Swift* arhitekturi. Kao i s implementacijom aplikacije, bit će prikazani testovi za jedan od repozitorija te testovi *Interactora, Presentera* i *Routera* za jednu scenu.

6.3. Jedinični testovi

Važnost jediničnih testova raste s kompleksnošću aplikacije. S obzirom na to da je ova arhitektura namijenjena kompleksnijim aplikacijama, bilo bi idealno da podržava dobro jedinično testiranje. U nastavku ćemo vidjeti da je to svakako tako. Prvo će biti prikazano kreiranje *mockova*, a zatim jedinično testiranje repozitorija. Nakon toga će biti prikazano jedinično testiranje *Interactora, Presentera* i *Routera*.

Mockovi se mogu implementirati na više načina i mogu biti različiti. Primjerice *mockovi* klasa koje implementiraju protokole su zapravo isto klase koje implementiraju te protokole. U ovom praktičnom radu to se odnosi na sve *mockove* servisa, repozitorija, *Presentera* i *ViewControllera*, odnosno protokola koji on implementira kao izlaz iz *Presentera*. Implementacija tih *mockova* može biti izvedena na više načina, a u nastavku će biti prikazana implementacija u ovom praktičnom primjeru.

Već je spomenuto da *mockovi* služe tome da sa sigurnošću možemo biti sigurni da klasa koju *mockamo* radi točno ono što smo naveli u *mocku* i tako kod testiranja klasa koje koriste taj *mock* možemo biti sigurni da ako test padne, nije problem u klasi koju testirana klasa koristi. Od *mockova* očekujemo da možemo saznati nekoliko ključnih podataka, a to su je li neka metoda pozvana, koliko puta je pozvana, s kojim je parametrima pozvana te toj metodi zadajemo što ona vraća. Pogledajmo primjer `FirestoreServiceMocka` prikazanog ispod na programskom kôdu 28.

```
import Foundation
@testable import Connect

class FirestoreServiceMock: FirestoreServiceProtocol {
    var myError: MyError?
    var expectedResponse: Codable?

    var getDocumentCalled = false
    var getDocumentCounter = 0
    var getDocumentDocumentPath: String?

    func getDocument<T: Codable>(documentPath: String, completion:
@escaping ((Result<T, MyError>) -> Void)) {
        getDocumentCalled = true
        getDocumentCounter += 1
        self.getDocumentDocumentPath = documentPath

        handleCompletion(completion)
    }

    private func handleCompletion<T: Codable>(_ completion: @escaping
((Result<T, MyError>) -> Void)) {
        guard let myError = myError else {
            guard let expectedResponse = expectedResponse else {
                return
            }
            completion(.success(expectedResponse as! T))
            return
        }
        completion(.failure(myError))
    }
  …
}
```

Programski kôd 28 FirestoreServiceMock

Na početku su definirane dvije varijable, `myError` i `expectedResponse`. S obzirom na to da se radi o *mocku*, svaki test koji koristi taj *mock* ima mogućnost odlučivanja što će taj *mock*

vratiti. Primjerice, ako je potrebno da *mock* vrati grešku, u testu će se postaviti `myError` varijabla. Ako je pak potrebno da *mock* vrati uspjeh s nekim određenim tipom, u testu će se postaviti `expectedResponse` varijabla. Korištenje tih dviju varijabli je definirano u implementaciji metoda *mocka*.

Pogledajmo metodu `getDocument`. Ta se metoda poziva s parametrom `documentPath` te `completionom`. Za tu su metodu kreirane tri varijable: `getDocumentCalled`, `getDocumentCounter` i `getDocumentDocumentPath` koje su na početku definirane sa *false*, 0 i *nil*. Na poziv metode se te varijable postavljaju na *true*, += 1 i na `documentPath` atribut metode. Nakon toga slijedi implementacija logike izlaza metode. Ako je u testu postavljen `myError`, metoda poziva `completion` neuspjeh s tom varijablom. Ako je pak postavljen `expectedResponse`, metoda poziva `completion` uspjeh s tom varijablom. Ako niti jedna od tih varijabli nije postavljena, `completion` se ne poziva. S obzirom na to da se ta logika koristi u svakoj metodi u ovom *mock,* odvojena je u posebnu metodu radi preglednosti. Isto tak su postavljene i ostale metode u *mocku* ovog servisa te implementacija nije prikazana na primjeru zbog redundantnosti. Na ovaj način su implementirani *mockovi* svih klasa koje implementiraju neki protokol.

Druga vrsta *mockova* u ovom praktičnom dijelu su *mockovi* podataka. S obzirom na to da podatci, odnosno strukture ne implementiraju protokole, te je *mockove* potrebno „ručno“ izraditi, odnosno instancirati objekte tih modela. U ovoj aplikaciji *mockovi* podataka za testiranje se nalaze unutar `DataMock` strukture prikazanog na programskom kôdu 29.

```
@testable import Connect
import Foundation

struct DataMock {
    // User
    var userId: String? = "id"
    var userUsername: String? = "username"
    var userFirstName: String? = "first name"
    var userLastName: String? = "last name"
    var userEmail: String? = "email@email.com"
    var userProfileImage: String? = "www.google.com"

    // Authentication
    var authenticationUserId: String? = "user id"
```

```
    // Project
    var projectId: String? = "project id"
    var projectTitle: String? = "project title"
    var projectCreatedAt: Date? = Date(timeIntervalSince1970: 12345)
    var projectDescription: String? = "project description"
    var projectHaveTags: [String]? = ["have1", "have2"]
    var projectNeedTags: [String]? = ["need1", "need2"]
}

extension DataMock {
    func getUser() -> User {
        User(id: userId,
             username: userUsername,
             firstName: userFirstName,
             lastName: userLastName,
             email: userEmail,
             profileImage: userProfileImage)
    }

    func getAuthenticationResponse() -> AuthenticationResponse {
        AuthenticationResponse(userId: authenticationUserId)
    }
}
extension DataMock {
    func getProject() -> Project {
        Project(id: projectId,
                title: projectTitle,
                createdAt: projectCreatedAt,
                description: projectDescription,
                haveTags: projectHaveTags,
                needTags: projectNeedTags,
                owner: getUser().getUserNested())
    }
}...
```

Programski kôd 29 DataMock

Za početak se predefiniraju vrijednosti atributa svih struktura, a zatim se pomoću metoda mogu dohvatiti pojedini objekti. `DataMock` koriste testovi kako bi se izbjeglo višestruko ponavljanje kôda instanciranja objekata. Na manjim aplikacijama to i nije veliki problem, ali pojavljuje li se neki model na primjerice 20 ekrana, ako dođe do izmjene tog modela, potrebno je i mijenjati *mock* na minimalno 20 mjesta, dok na ovaj način samo na jednom.

Nakon što imamo pripremljene *mockove*, možemo se za početak posvetiti jediničnom testiranju. Kako bi mogli testirati repozitorij, potrebno je prvo provjeriti što taj repozitorij koristi i kreirati *mockove*. Uzmimo za primjer `UserRepository`. On implementira dvije metode protokola i koristi `FirestoreService`. Iz toga znamo da nam je za jedinične testove potreban `FirestoreServiceMock`. Unutar testne klase, klasu koju testiramo označavamo kao sustav pod testiranjem (eng. *system under test – sut*). `UserRepositoryProtocol` sadrži dvije metode. Pogledajmo `getUser` metodu. Ona se poziva s parametrom `userId` te `completionom`. Njena implementacija poziva metodu `FirestoreServicea getDocument` s parametrom `documentPath` koji je `„users/" + userId`. Za testiranje ove metode, koristeći `FirestoreServiceMock`, možemo provjeriti je li pozvana točna metoda servisa, koliko je puta pozvana i poziva li se `completion` na uspjeh ili neuspjeh s točnim podatcima. Ako to sve implementiramo u testovima, dobivamo `UserRepositoryTests` prikazan ispod programskim kôdom 30.

```
import Foundation
@testable import Connect
import XCTest

class UserRepositoryTests: XCTestCase {
    private var sut: UserRepository!
    private var firestoreServiceMock: FirestoreServiceMock!
    private var dataMock: DataMock!

    override func setUpWithError() throws {
        firestoreServiceMock = FirestoreServiceMock()
        sut = UserRepository(firestoreService: firestoreServiceMock)
        dataMock = DataMock()
    }

    override func tearDownWithError() throws {
        firestoreServiceMock = nil
```

```
        sut = nil
        dataMock = nil
    }
}

// MARK: getUser(userId: String, completion: @escaping ((Result<User,
MyError>) -> Void)) tests
extension UserRepositoryTests {
    func
testGetUser_WhenCalledWithUserId_ShouldCallFirestoreServiceGetDocumentWithD
ocumentPath() {
        // Given
        let userId = "user id"
        let expectedDocumentPath = "users/" + userId

        // When
        sut.getUser(userId: userId) { _ in
            //
        }

        // Then
        XCTAssertTrue(firestoreServiceMock.getDocumentCalled)
        XCTAssertEqual(firestoreServiceMock.getDocumentCounter, 1)
        XCTAssertEqual(firestoreServiceMock.getDocumentDocumentPath,
expectedDocumentPath)
    }

    func
testGetUser_WhenCalledWithUserIdOnSuccess_ShouldCallCompletionWithUser() {
        // Given
        let expectation = expectation(description: "expectation")
        let userId = dataMock.userId ?? ""
        let expectedResponse = dataMock.getUser()
        firestoreServiceMock.expectedResponse = expectedResponse

        // When
        sut.getUser(userId: userId) { result in
            guard case .success(let user) = result, user.id == userId else
{
                return
```

```
            }
            expectation.fulfill()
        }

        // Then
        wait(for: [expectation], timeout: 5)
    }

    func
testGetUser_WhenCalledWithUserIdOnFailure_ShouldCallCompletionMyError() {
        // Given
        let expectation = expectation(description: "expectation")
        let userId = dataMock.userId ?? ""
        let myError = MyError(type: .firestoreFailed, message: nil)
        firestoreServiceMock.myError = myError

        // When
        sut.getUser(userId: userId) { result in
            guard case .failure(let error) = result, myError == error else
{
                return
            }
            expectation.fulfill()
        }

        // Then
        wait(for: [expectation], timeout: 5)
    }
}
```

Programski kôd 30 UsersRepositoryTests

Na početku se definira `sut` te potrebni *mockovi*. U metodama `XCTestCase` klase `setupWithError` (metoda koja se poziva prije izvršavanja svakog testa) se svi ti objekti instanciraju dok se u `tearDownWithError` (metoda koja se poziva nakon svakog završenog testa) deinicijaliziraju kako bi bili sigurni da prethodni test nema utjecaj na trenutni. U prvom testu se provjerava poziva li točna metoda servisa jednom s očekivanim `documentPathom`. U drugom testu, koristeći `DataMock`, servisu se postavlja očekivani odgovor te se provjerava jesu li podatci ispravni te poziva li se `completion`. U trećem testu se na isti način provjerava

ponašanje u slučaju greške. Prvo se servisu postavlja greška čime se osigurava da u tom testu servis vraća točno tu grešku te se nakon toga provjerava poziva li testirana metoda `completion` neuspjeh s tom greškom. Na ovaj način se mogu testirati i ostale metode repozitorija.

Na isti način možemo testirati i sve ostale repozitorije. Ako repozitorij koristi više servisa, za njih na isti način kao što je ranije opisano možemo dodati *mockove*.

Nakon testiranih repozitorija, možemo testirati pojedine komponente scena. U ovom primjeru će biti prikazano testiranje *Interactor* i *Presenter* klasa, konkretno `LoginInteractor` i `LoginPresenter` klasa. S obzirom na to da arhitektura nalaže kružan tok podataka, za početak nam i u testovima trebaju ti podatci. S obzirom na definiran model podataka za *Login* scenu, jednostavno je izraditi i *mock* tih podataka. Time dobivamo `LoginDataModelMock`. Ovdje su postavljeni svi očekivani podatci koji sudjeluju u *Login* sceni. Za svaku akciju postoje zahtjev, odgovor te `ViewModel`. Ako modeli podataka pojedinih scena koriste neki od modela iz domene aplikacije, može se koristiti `DataMock` kako bi se opet izbjegla potreba za nepotrebnim kreiranjem *mockova.*

S pripremljenim modelima podataka, možemo krenuti u testiranje `LoginInteractora` i `LoginPresentera`. *Interactor* u svakoj metodi prima zahtjev i šalje odgovor, a *Presenter* u svakoj metodi prima odgovor i šalje *ViewModel* što i odgovara *mocku* podataka.

Pogledajmo prvo *`LoginInteractor`* čiji je programski kôd prikazan ranije te što je sve moguće testirati. Za početak je vidljivo da ta klasa koristi `KeychainServiceProtocol` i `AuthenticationRepositoryProtocol` te sadrži `LoginPresenterProtocol`. Kao i za repozitorij, potrebno je kreirati *mockove* za te tri klase kako bi mogli biti sigurni da `LoginInteractor` koji se testira poziva sve potrebne metode klasa koje koristi. Time dobivamo `KeychainServiceMock`, `AuthenticationRepositoryMock` te `LoginViewControllerMock` koji zapravo implementira `LoginPresenterOutput` protokol.

`LoginInteractorProtocol` sadrži dvije metode, `loginUser()` i `forgottenPassword()`. Svaka od njih prima pripadajući zahtjev. Potrebno je testirati te dvije metode. U testovima se provjerava pozivaju li metode sve što trebaju s potrebnim podatcima te vraćaju li odgovor s točnim podatcima. Nakon ove analize, dobivamo `LoginInteractorTests` prikazan na programskom kôdu 31.

```
@testable import Connect
import XCTest

class LoginInteractorTests: XCTestCase {
    private var sut: LoginInteractor!
    private var loginPresenterMock: LoginPresenterMock!
    private var keychainServiceMock: KeychainServiceMock!
    private var authenticationRepositoryMock: AuthenticationRepositoryMock!
    private var loginDataModelMock: LoginDataModelMock!
    private var dataMock: DataMock!

    override func setUpWithError() throws {
        loginDataModelMock = LoginDataModelMock()
        keychainServiceMock = KeychainServiceMock()
        authenticationRepositoryMock = AuthenticationRepositoryMock()
        sut = LoginInteractor(keychainService: keychainServiceMock,
                              authenticationRepository:
authenticationRepositoryMock)
        loginPresenterMock = LoginPresenterMock()
        sut.presenter = loginPresenterMock
        dataMock = DataMock()
    }

    override func tearDownWithError() throws {
        sut = nil
        loginPresenterMock = nil
        keychainServiceMock = nil
        authenticationRepositoryMock = nil
        loginDataModelMock = nil
        dataMock = nil
    }
}

// MARK: LoginUserAction tests
extension LoginInteractorTests {
    func
testLoginUserAction_WhenCalledWithRequest_ShouldCallAuthenticationRepositor
ySignInUserWithRequestData() {
        // Given
        let request = loginDataModelMock.loginAction.request
```

```
        // When
        sut.loginUser(request: request)

        // Then
        XCTAssertTrue(authenticationRepositoryMock.signInUserCalled)
        XCTAssertEqual(authenticationRepositoryMock.signInUserCounter, 1)
        XCTAssertEqual(authenticationRepositoryMock.email, request.email)
        XCTAssertEqual(authenticationRepositoryMock.password,
request.password)
    }

    func
testLoginUserAction_WhenCalledWithRequestOnSuccess_ShouldCallKeychainServic
eSetUserLoggedInWiithTrue() {
        // Given
        let request = loginDataModelMock.loginAction.request

        // When
        sut.loginUser(request: request)

        // Then
        XCTAssertTrue(keychainServiceMock.getUserLoggedIn())
    }

    func
testLoginUserAction_WhenCalledWithRequestOnSuccess_ShouldCallKeychainServic
eSetUserIdWithUserIdFromResponse() {
        // Given
        let request = loginDataModelMock.loginAction.request
        let expectedUserId = dataMock.authenticationUserId

        // When
        sut.loginUser(request: request)

        // Then
        XCTAssertEqual(expectedUserId, keychainServiceMock.getUserId())
    }
```

```
    func
testLoginUserAction_WhenCalledWithRequestOnSuccess_ShouldCallPresenterDidSu
cceedLoginWithResponseSuccess() {
        // Given
        let request = loginDataModelMock.loginAction.request
        let expectedResponse =
loginDataModelMock.loginAction.responseSuccess

        // When
        sut.loginUser(request: request)

        // Then
        XCTAssertTrue(loginPresenterMock.didSucceedLoginCalled)
        XCTAssertEqual(loginPresenterMock.didSucceedLoginCounter, 1)
        XCTAssertEqual(loginPresenterMock.didSucceedLoginResponse,
expectedResponse)
    }

    func
testLoginUserAction_WhenCalledWithRequestOnError_ShouldCallPresenterDidFail
LoginWithResponseFailure() {
        // Given
        let myError = LoginDataModelMock.myError
        authenticationRepositoryMock.myError = myError
        let request = loginDataModelMock.loginAction.request
        let expectedResponse =
loginDataModelMock.loginAction.responseFailure

        // When
        sut.loginUser(request: request)

        // Then
        XCTAssertTrue(loginPresenterMock.didFailLoginCalled)
        XCTAssertEqual(loginPresenterMock.didFailLoginCounter, 1)
        XCTAssertEqual(loginPresenterMock.didFailLoginResponse,
expectedResponse)
    }
}
```

Programski kôd 31 LoginInteractorTests

Za *loginUser*() metodu vidimo da poziva metodu AuthenticationRepositorya *signInUser*(). Na uspjeh (*success*) ona vraća AuthenticationResponse model, a metoda koju testiramo poziva KeychainService metode setUserLoggedIn(true) i setUserId(authenticationResponse.userid) te poziva metodu *presentera*. Na neuspjeh (*failure)* se iz repozitorija vraća MyError, a metoda koju testiramo poziva metodu *presentera*. To je pet ponašanja koja se očekuju od pozivanja te metode pa će za svako od tih očekivanih ponašanja biti kreiran jedinični test.

Za početak se u prvom testu testira poziva li se metoda signInUser() AuthenticationRepositorya. Kako bi pozvali metodu loginUser() u LoginInteractoru, potreban je zahtjev koji je ranije kreiran u LoginDataModelMocku. Na poziv loginUser() metode, koristeći AuthenticationRepositoryMock provjeravamo je li točna metoda repozitorija pozvana s točnim podatcima.

Nakon toga, testiraju sve tri očekivana ponašanja u slučaju uspjeha repozitorija. Koristi se isti zahtjev kao i u prijašnjem testu.

U drugom testu koristeći KeychainServiceMock provjerava se je li postavljeno da je korisnik prijavljen.

Treći test provjerava poziva li se na uspjeh metoda KeychainServicea setUserId()s točnim podatcima.

U četvrtom ovom testu provjeravamo poziva li se na uspjeh repozitorija metoda LoginPresentera didSucceedLogin() s točnim podatcima. Iz LoginDataModelMocka možemo uzeti očekivani odgovor te na poziv loginUser() metode klase koju testiramo provjeriti poziva li se metoda *presentera* s očekivanim odgovorom.

Za kraj se u petom testira poziva li se točna metoda *presentera* s točnim podatcima na neuspjeh repozitorija. Slično kao i u prethodnom testu, koriste se zahtjev i očekivani odgovor iz LoginDataModelMocka. Jedina je razlika što se greška nadodaje i na AuthenticationRepositoryMock s obzirom na način na koji je on postavljen (ako greška nije zadana u testu, metoda repozitorija će vratiti uspjeh).

Time su testirana sva očekivana ponašanja metode LoginInteractora. Na isti način se testiraju i ostale metode te one neće biti prikazane ovdje, ali su dostupne u izvornom kôdu.

Za kraj ostaje testiranje *Presentera*, odnosno konkretno primjer testiranja LoginPresentera čija je implementacija prikazana ranije. Kao i do sada, za početak su potrebni *mockovi* i plan što je potrebno testirati. Jedina referenca koju LoginPresenter ima

je delegat koji implementira `LoginPresenterOutput` protokol, a to je `LoginViewController`. Time dobivamo `LoginViewControllerMock` koji implementira samo `LoginPresenterOutput` protokol. `LoginPresenter` sadrži četiri metode. Svaka od njih prima odgovor te šalje `ViewModel`. S obzirom na isto ponašanje, bit će prikazan test samo jedne metode. Implementacija `LoginPresenterTests` klase je prikazan na programskom kôdu 32.

```
@testable import Connect
import XCTest

class LoginPresenterTests: XCTestCase {
    private var sut: LoginPresenter!
    private var loginViewControllerMock: LoginViewControllerMock!
    private var loginDataModelMock: LoginDataModelMock!

    override func setUpWithError() throws {
        loginDataModelMock = LoginDataModelMock()
        loginViewControllerMock = LoginViewControllerMock()
        sut = LoginPresenter()
        sut.viewController = loginViewControllerMock
    }

    override func tearDownWithError() throws {
        loginDataModelMock = nil
        loginViewControllerMock = nil
        sut = nil
    }
}

// MARK: LoginAction tests
extension LoginPresenterTests {
    func
testDidSucceedLogin_WhenCalledWithResponseSuccess_ShouldCallViewControllerD
idSucceedLoginWithViewModelSuccess() {
        // Given
        let response = loginDataModelMock.loginAction.responseSuccess
        let expectedViewModel =
loginDataModelMock.loginAction.viewModelSuccess
```

```
        // When
        sut.interactor(didSucceedLogin: response)

        // Then
        XCTAssertTrue(loginViewControllerMock.didSucceedLoginCalled)
        XCTAssertEqual(loginViewControllerMock.didSucceedLoginCounter, 1)
        XCTAssertEqual(loginViewControllerMock.didSucceedLoginViewModel,
expectedViewModel)
    }
}
```

Programski kôd 32 LoginPresenterTests

Uzmimo prvu metodu `LoginPresentera, interactor(didSucceedLogin response: …)` koja prima odgovor tipa `Login.LoginAction.Response.Success` te šalje `ViewModel` tipa `Login.LoginAction.ViewModel.Success`. *Mock* tih podataka imamo u `LoginDataModelMocku`. U testu tako možemo upotrijebiti te *mock* podatke kao odgovor i očekivani *ViewModel*. Pozivom metode koju testiramo koristeći `LoginViewControllerMock` možemo provjeriti je li pozvana točna metoda `LoginPresenterOutputa` te je li pozvana s točnim `ViewModelom`.

Na isti način se testiraju i preostale metode *Presentera*.

Time su jediničnim testovima pokriveni repozitoriji, poslovna logika scena te prezentacijska logika scena. Jedinični testovi *Interactora* i *Presentera* različitih scena su vrlo slični prikazanima. Potrebno je kreirati *mock* podataka scene te *mockove* svih klasa koje testirane klase koriste. S obzirom na to da za svaku od scena treba izraditi barem 7 ili više datoteka, korištenje *Xcode* predložaka (eng. *templates*) je jedna od mogućnosti za olakšano kreiranje datoteka za scene.

7. Zaključak

Arhitekture su važan dio razvoja softvera pa tako i iOS aplikacija. U ovom radu opisane su najčešće korištene arhitekture u izradi iOS aplikacija, njihove prednosti i mane te mogućnost testiranja u pojedinim arhitekturama kroz komparativnu analizu, analizu i sintezu, klasifikaciju, deskripciju te kompilaciju. Od svih navedenih arhitektura, u ovom radu posebnu pažnju ima čista arhitektura. *Dependency Rule* koji je temelj čiste arhitekture zahtjeva jednosmjeran tok podataka za razliku od ostalih spomenutih arhitektura gdje je tok podataka dvosmjeran. Iako je ova arhitektura vrlo robusna i omogućava izradu velikih aplikacija, ima i nekih mana. Prije svega, tu je velik broj datoteka potrebnih za samo jednu scenu. S obzirom na to da ova arhitektura nije toliko često korištena, može predstaviti problem developerima koji se prvi puta susreću s njom na nekom projektu. Za srednje ili manje aplikacije, može se koristiti primjerice MVVM arhitektura koja se ne sastoji od velikog broja datoteka i testiranje je relativno dobro. Od svih arhitektura, idealno bi bilo izbjegavati MVC i MVP arhitekture. Jedina iznimka gdje bi te arhitekture bile preporučljive su vrlo male aplikacije (od svega par ekrana) od kojih se ne očekuje daljnji rast.

U praktičnom dijelu prikazano je korištenje čiste arhitekture na primjeru iOS aplikacija. Aplikacije je izrađena u *Xcode* programskom okruženju. Izvorni kôd aplikacije dostupan je na https://github.com/dinoMartan/diplomskiRad. Aplikacija se sastoji od devet ekrana odnosno scena. Svaka od tih scena se sastoji od modela podataka, korisničkog sučelja, poslovne logike i prezentacijske logike. Jasno definiran kružni tok podataka omogućava jasnu podjelu odgovornosti. To pak omogućava da na jednoj sceni može raditi i više developera istovremeno. Nakon zadanog modela podataka, jedan developer može raditi na korisničkom sučelju jer zna koje podatke očekuje i koje podatke treba slati poslovnoj logici, drugi developer može raditi na implementaciji poslovne logike jer zna koje podatke očekuje i koje podatke treba poslati dalje. Treći developer može raditi na prezentacijskoj logici, odnosno pretvarati očekivane podatke iz poslovne logike u podatke koje očekuje korisničko sučelje. Nakon spajanja svih dijelova scene, neće biti konflikata jer niti jedan developer ne dira klase na kojima rade drugi. Osim toga, poslovna logika i prezentacijska logika mogu biti pokrivene jediničnim testovima.

Izrada softvera se brzo mijenja. Često izlaze nove biblioteke, okruženja pa čak i programski jezici. No, od kad postoji izrada softvera, postoji i potreba za njegovom organizacijom. Arhitekture softvera imaju upravo tu ulogu. Zanemarivanje arhitektura može izgledati kao dobar način za uštedjeti vrijeme tijekom razvoja, ali to može dovesti do problema kasnije zbog nemogućnosti izmjene i slabe ili nikakve pokrivenosti testovima.

Popis literature

[1] „SnapKit“. Pristupljeno: 25. svibanj 2022. [Na internetu]. Dostupno na: https://cocoapods.org/pods/SnapKit

[2] „KeychainSwift“. Pristupljeno: 25. svibanj 2022. [Na internetu]. Dostupno na: https://cocoapods.org/pods/KeychainSwift

[3] „Kingfisher“. Pristupljeno: 25. svibanj 2022. [Na internetu]. Dostupno na: https://cocoapods.org/pods/Kingfisher

[4] „IQKeyboardManager“. Pristupljeno: 25. svibanj 2022. [Na internetu]. Dostupno na: https://cocoapods.org/pods/IQKeyboardManager

[5] „WSTagsField“. Pristupljeno: 25. svibanj 2022. [Na internetu]. Dostupno na: https://cocoapods.org/pods/WSTagsField

[6] „Firebase“. Google. Pristupljeno: 25. svibanj 2022. [Na internetu]. Dostupno na: https://cocoapods.org/pods/Firebase

[7] „FirebaseAuth“. Google. Pristupljeno: 25. svibanj 2022. [Na internetu]. Dostupno na: https://cocoapods.org/pods/FirebaseAuth

[8] „FirebaseFirestore“. Google. Pristupljeno: 25. svibanj 2022. [Na internetu]. Dostupno na: https://cocoapods.org/pods/FirebaseFirestore

[9] „FirebaseFirestoreSwift“. Google. Pristupljeno: 25. svibanj 2022. [Na internetu]. Dostupno na: https://cocoapods.org/pods/FirebaseFirestoreSwift

[10] „FirebaseStorage“. Google. Pristupljeno: 25. svibanj 2022. [Na internetu]. Dostupno na: https://cocoapods.org/pods/FirebaseStorage

[11] R. C. Martin, *Clean Architecture*. Pearson Education Inc., 2018.

[12] D. Garlan, „Software Architecture“, Pristupljeno: 09. kolovoz 2022. [Na internetu]. Dostupno na: http://www.cs.cmu.edu/afs/cs.cmu.edu/project/able/ftp/encycSE2001/encyclopedia-dist.pdf

[13] S. Chauhan, „Different Types of Software Design Principles“. https://www.dotnettricks.com/learn/designpatterns/different-types-of-software-design-principles (pristupljeno 09. lipanj 2022.).

[14] S. Martins, „10 Design Principles in Software Engineering“. https://betterprogramming.pub/10-design-principles-in-software-engineering-f88647cf5a07 (pristupljeno 09. lipanj 2022.).

[15] G. Kumar Arora, *SOLID Principles Succinctly*. Morrisville, NC 27560: Syncfusion, 2016. Pristupljeno: 09. srpanj 2022. [Na internetu]. Dostupno na: http://www.freedownloads247.com/UploadedFiles/8-2017/4223/solidprinciplessuccinctly.pdf

[16] „App architecture“. Android. Pristupljeno: 15. svibanj 2022. [Na internetu]. Dostupno na: https://developer.android.com/topic/architecture/intro

[17] „Model-View-Controller“. Apple. Pristupljeno: 15. svibanj 2022. [Na internetu]. Dostupno na: https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html

[18] S. El Oulladi, „iOS Swift : MVP Architecture“. https://saad-eloulladi.medium.com/ios-swift-mvp-architecture-pattern-a2b0c2d310a3 (pristupljeno 15. svibanj 2022.).

[19] „Introduction to Model/View/ViewModel pattern for building WPF apps“. Microsoft. Pristupljeno: 15. svibanj 2022. [Na internetu]. Dostupno na: https://docs.microsoft.com/hr-hr/archive/blogs/johngossman/introduction-to-modelviewviewmodel-pattern-for-building-wpf-apps

[20] M. Aljamea i M. Alkandari, „MMVMi: A Validation Model for MVC and MVVM Design Patterns in iOS Applications“, 2018, Pristupljeno: 19. svibanj 2022. [Na internetu]. Dostupno na: http://www.iaeng.org/IJCS/issues_v45/issue_3/IJCS_45_3_03.pdf?ref=https://githubhelp.com

[21] B. Orlov, „iOS Architecture Patterns“, 2015. https://medium.com/ios-os-x-development/ios-architecture-patterns-ecba4c38de52 (pristupljeno 19. svibanj 2022.).

[22] R. Law, „Clean Swift“. Pristupljeno: 05. veljača 2022. [Na internetu]. Dostupno na: https://clean-swift.com/

[23] D. Vrzan, „Getting Started With the VIP Clean Architecture Pattern“, 2022. https://www.raywenderlich.com/29416318-getting-started-with-the-vip-clean-architecture-pattern (pristupljeno 20. svibanj 2022.).

[24] D. Atanasov, „Introducing Clean Swift Architecture (VIP)“, 2017. https://medium.com/hackernoon/introducing-clean-swift-architecture-vip-770a639ad7bf (pristupljeno 15. svibanj 2022.).

[25] Z. Moola, „Mocking services for UI testing in Swift“. https://medium.com/dvt-engineering/mocking-services-for-ui-testing-in-swift-ff66dd1b1934 (pristupljeno 30. srpanj 2022.).

[26] F. Sholichin, M. Adham bin Isa, S. Abd Halim, i M. Firdaus bin Harun, „Review of ios architectural pattern for testability, modifiability, and performance quality“, *J. Theor. Appl. Inf. Technol.*, sv. 97, kol. 2019, [Na internetu]. Dostupno na: http://www.jatit.org/volumes/Vol97No15/3Vol97No15.pdf

Popis slika

Slika 1 Single Responsibility Principle 6
Slika 2 Čista arhitektura (prema [11], str. 203) 15
Slika 3 Čista arhitektura – prelaženje granica (prema [11], 203) 16
Slika 4 Čista arhitektura - sučelja za ulaz i izlaz (prema [11], 203) 17
Slika 5 MVC arhitektura (prema [17]) 19
Slika 6 MVP arhitektura (prema [18]) 20
Slika 7 MVVM arhitektura (prema [20]) 21
Slika 8 VIPER arhitektura (prema [21]) 22
Slika 9 VIP arhitektura (prema [23]) 23
Slika 10 Clean Swift arhitektura (prema [24]) 24
Slika 11 Korisničko sučelje aplikacije za jedinične testove 27
Slika 12 Prijava 46
Slika 13 Zaboravljena lozinka 47
Slika 14 Registracija 48
Slika 15 Početna 49
Slika 16 Detalji projekta 50
Slika 17 Detalji razgovora 51
Slika 18 Razgovori 52
Slika 19 Moji projekti 53
Slika 20 Kreiranje projekta 54
Slika 21 Uređivanje projekta 55
Slika 22 Profil 56
Slika 23 Organizacija datoteka – pregled 58
Slika 24 Organizacija datoteka – aplikacija 58
Slika 25 Organizacija datoteka – servisi 59

Slika 26 Organizacija datoteka - repozitoriji 59
Slika 27 Organizacija datoteka – tok.....60
Slika 28 Organizacija datoteka – modeli.....60
Slika 29 Organizacija datoteka – pomoćne datoteke 60
Slika 30 Organizacija datoteka – scene.....61
Slika 31 Organizacija datoteka – scena Prijava 62

Popis tablica

Tablica 1 Usporedba arhitektura.....42

Popis programskih kôdova

Programski kôd 1 Primjer klase Automobil 7

Programski kôd 2 Primjer klase Utrka..... 7

Programski kôd 3 Primjer klase Osoba..... 8

Programski kôd 4 Primjer protokola Trkac..... 9

Programski kôd 5 Primjer protokola Trkac – proširenje..... 9

Programski kôd 6 Primjer klase Pravokutnik (prema [11], 79).....10

Programski kôd 7 Primjer klase Kvadrat.....11

Programski kôd 8 Primjer protokola Vozilo 11

Programski kôd 9 Primjer klasa Automobil i Brod 12

Programski kôd 10 Jedinični testovi - zajedničke komponente 28

Programski kôd 11 Jedinični testovi - zajednički mockovi.....29

Programski kôd 12 Jedinični testovi – MVCController.....31

Programski kôd 13 Jedinični testovi – MVPController.....33

Programski kôd 14 Jedinični testovi - MVPPresenter 34

Programski kôd 15 Jedinični testovi – MVPControllerMock 34

Programski kôd 16 Jedinični testovi - MVPPresenter testovi 36

Programski kôd 17 Jedinični testovi – MVVMController.....38

Programski kôd 18 Jedinični testovi – MVVMViewModel.....39

Programski kôd 19 Jedinični testovi - MVVMViewModel testovi 40

Programski kôd 20 KeychainService 64

Programski kôd 21 ProjectsRepository.....66

Programski kôd 22 FlowCoordinator.....68

Programski kôd 23 LoginFlowCoordinator.....69

Programski kôd 24 Login.....71

Programski kôd 25 LoginViewController.....74

Programski kôd 26 LoginInteractor 77
Programski kôd 27 LoginPresenter 79
Programski kôd 28 FirestoreServiceMock 81
Programski kôd 29 DataMock 83
Programski kôd 30 UsersRepositoryTests 86
Programski kôd 31 LoginInteractorTests 90
Programski kôd 32 LoginPresenterTests 93

Prilozi

1) Programski kôd aplikacije *Connect* - https://github.com/dinoMartan/diplomskiRad