Usporedba VIPER i MVVM arhitektura za razvoj iOS aplikacija

Bunoza, Domagoj

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

Sveučilišni studij

USPOREDBA VIPER I MVVM ARHITEKTURA ZA RAZVOJ iOS APLIKACIJA

Diplomski rad

Domagoj Bunoza

Osijek, 2023.

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

Obrazac D1: Obrazac za imenovanje Povjerenstva za diplomski ispit

Osijek, 08.07.2023.

Odboru za završne i diplomske ispite

Imenovanje Povjerenstva za diplomski ispit

Ime i prezime Pristupnika:	Domagoj Bunoza
Studij, smjer:	Diplomski sveučilišni studij Računarstvo
Mat. br. Pristupnika, godina upisa:	D-1190R, 07.10.2021.
OIB studenta:	02615020561
Mentor:	izv. prof. dr. sc. Josip Balen
Sumentor:	,
Sumentor iz tvrtke:	Josip Juhasz
Predsjednik Povjerenstva:	prof. dr. sc. Krešimir Nenadić
Član Povjerenstva 1:	izv. prof. dr. sc. Josip Balen
Član Povjerenstva 2:	Matej Arlović, mag. ing. comp.
Naslov diplomskog rada:	Usporedba VIPER i MVVM arhitektura za razvoj iOS aplikacija
Znanstvena grana diplomskog rada:	**Programsko inženjerstvo (zn. polje računarstvo)**
Zadatak diplomskog rada:	U diplomskom radu biti će provedena detaljna analiza dviju najpopularnijih arhitektura za razvoj iOS aplikacija: VIPER i MVVM. U radu će biti prvo provedena teorijska analiza i usporedba VIPER i MVVM arhitektura te njihovih prednosti i nedostataka, što prethodi prikazu implementacije pojedine arhitekture. Prikaz implementacija bit će demonstriran na primjeru značajke Consent Management (Upravljanje suglasnostima) u sklopu aplikacije za elektroničke kartone pacijenata. Nakon prikaza implementacija bit će prikazan i obrazložen postupak migracije s VIPER na MVVM arhitekturu te provedena diskusija o upotrebi pojedine arhitekture. Sumentor: Josip Juhasz Tema rezervirana za: Domagoj Bunoza
Prijedlog ocjene pismenog dijela ispita (diplomskog rada):	Izvrstan (5)
Kratko obrazloženje ocjene prema Kriterijima za ocjenjivanje završnih i diplomskih radova:	Primjena znanja stečenih na fakultetu: 3 bod/boda Postignuti rezultati u odnosu na složenost zadatka: 3 bod/boda Jasnoća pismenog izražavanja: 3 bod/boda Razina samostalnosti: 3 razina
Datum prijedloga ocjene od strane mentora:	08.07.2023.
Potvrda mentora o predaji konačne verzije rada:	*Mentor elektronički potpisao predaju konačne verzije.* Datum:

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

IZJAVA O ORIGINALNOSTI RADA

Osijek, 31.08.2023.

Ime i prezime studenta:	Domagoj Bunoza
Studij:	Diplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	D-1190R, 07.10.2021.
Turnitin podudaranje [%]:	4

Ovom izjavom izjavljujem da je rad pod nazivom**: Usporedba VIPER i MVVM arhitektura za razvoj iOS aplikacija**

izrađen pod vodstvom mentora izv. prof. dr. sc. Josip Balen

i sumentora ,

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

Potpis studenta:

SADRŽAJ

1. UVOD

Arhitekturni obrasci ključni su za stvaranje aplikacija, kako na *iOS* platformi, tako i na ostalima, jer nude metodičan način upravljanja i organiziranja koda. Arhitekturni obrasci pružaju najbolje prakse i principe koji pomažu razvojnim inženjerima u stvaranju skalabilnih, održivih i modularnih aplikacija. Razvojni inženjeri implementacijom arhitekturnih obrazaca mogu učinkovito upravljati svojom složenošću koda, povećati mogućnost ponovne upotrebe koda i promovirati suradnju unutar razvojnih timova. Nekoliko dobro poznatih arhitekturnih obrazaca koji se često primjenjuju u razvoju mobilnih aplikacija za operacijski sustav *iOS* su *MVVM* (*Model-View-ViewModel*), *MVP* (*Model-View-Presenter*), *VIPER* (*View-Interactor-Presenter-Entity-Router*) i *MVC* (*Model-View-Controller*). Ovi obrasci idealni su za različite zahtjeve projekata i razvojne okolnosti jer svaki od njih posjeduje različite kvalitete i prednosti.

U drugom poglavlju ovog diplomskog rada objašnjeni su najčešći arhitekturni obrasci korišteni u razvoju mobilnih aplikacija za operacijski sustav *iOS*, a to su *MVC, MVP, MVVM* i *VIPER*. Za svaki arhitekturni obrazac prikazan je dijagram korištenja komponenata vlastitog arhitekturnog obrasca, potkrijepljen objašnjenjem i prednostima. Općenite smjernice za razvoj aplikacija za operacijski sustav *iOS*, smjernice za ljudsko učenje, značajke programskog jezika *Swift*, mogućnosti integriranog razvojnog okruženja *Xcode*, testiranje *iOS* aplikacija i korištenje distribucijske platforme *App Store*, objašnjeno je u trećem poglavlju. Četvrto poglavlje sadrži prikaz implementacije značajke za upravljanje suglasnostima mobilne aplikacije za elektroničke kartone pacijenata koristeći arhitekturni obrazac *MVVM*, a zatim i usporedbu s ekvivalentnom implementacijom koristeći arhitekturni obrazac *VIPER*. Peto poglavlje objašnjava tri koraka kojima je moguće s arhitekturnog obrasca *VIPER* prijeći na arhitekturni obrazac *MVVM*.

2. PREGLED AKTUALNIH ARHITEKTURNIH OBRAZACA ZA RAZVOJ iOS APLIKACIJA

Arhitekturni obrazac je skup pravila, tehnika i obrazaca koji prikazuju kako se razvoj mobilnih aplikacija treba odvijati. Taj skup pravila omogućava stvaranje logičnog i cjelovitog proizvoda koji se ne suprotstavlja zahtjevima klijenta niti industrijskom standardu. Budući da pružaju metodu strukturiranja koda i poboljšanja mogućnosti održavanja i skalabilnosti, arhitekturni obrasci ključni su za stvaranje iOS aplikacija. U zajednici razvojnih inženjera za iOS koristi se nekoliko arhitekturnih obrazaca, a odabir arhitekturnog obrasca ovisi o mnoštvu faktora, od kojih su neki publika koja se želi doseći, platforma na kojoj će aplikacija biti razvijena, značajke i funkcionalnosti koje su potrebne te vrijeme i novac koji su alocirani za izradu aplikacije, uz vještine razvojnog tima. *Apple*-ov sustav nudi razvojnim inženjerima smjernice o tome kako razviti *iOS* mobilnu arhitekturu temeljenu na *MVC* modelu, iako *iOS* razvojni inženjeri nisu ograničeni samo na jedan arhitekturni obrazac [1].

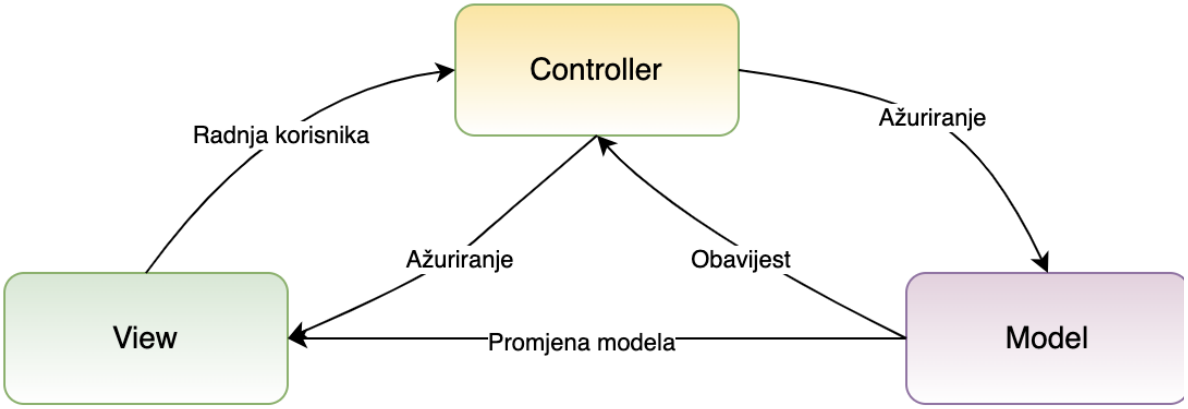
Mnoštvo aplikacija razvijeno je bez ikakve arhitekture ili bez obzira na ikakve industrijske standarde. Manjak arhitekture može uzrokovati dulji i skuplji razvojni proces, nemogućnost održavanja (posebno ako se dogodi promjena tima razvojnih inženjera), teška nadogradivost i skalabilnost, otežano testiranje i sklonost pogreškama. Dobar arhitekturni obrazac za mobilne aplikacije trebao bi provoditi principe razvoja programskih rješenja: *KISS, DRY* i *SOLID*. Također, jasno definiran arhitekturni obrazac podupire fleksibilnost i agilne metodologije razvoja, što čini testiranje učinkovitijim i čini buduće održavanje jednostavnijim i manje sklonim pogreškama. Dobar arhitekturni obrazac nije specifičan za platformu, već bi trebao biti primjenjiv i za nativna i hibridna rješenja kada su u pitanju mobilne aplikacije [2].

Važno je napomenuti da ne postoji savršeni univerzalni arhitekturni obrazac, ali odabirom ispravnog arhitekturnog obrasca, moguće je povećati brzinu tima razvojnih inženjera, poboljšati kvalitetu koda i ubrzati potencijalne nadogradnje i promjene. Osim toga, odabir „ispravnog“ arhitekturnog obrasca dobra arhitektura koda neće biti zajamčena. Za svaki arhitekturni obrazac potrebno je raspodijeliti kod na više manjih dijelova koji su međusobno neovisni koliko je moguće, a aplikaciju je poželjno podijeliti na više potrebnih modula. Također poželjno je korištenje ubrizgavanja ovisnosti (eng. *dependency injection*) [3].

Arhitekturni obrasci ne moraju biti strogo definirani u smislu da se po jednoj aplikacije smije koristiti isključivo jedan arhitekturni obrazac. Arhitekturni obrasci mogu se implementirati na razini cijele aplikacije, na razini modula, ili čak na razini dijela modula. U različitim modulima moguće je implementirati različite arhitekturne obrasce.

2.1. Arhitekturni obrazac MVC

Jedan od najpopularnijih arhitekturnih obrazaca u razvoju *iOS* aplikacija je *Model-View-Controller* (*MVC*). Nudi metodičan način raspodjele koda, podjele problema i poboljšanja skalabilnosti i mogućnosti održavanja *iOS* aplikacija [4]. Shematski prikaz *MVC* arhitekture prikazan je na slici 2.1. Jedna od glavnih mana ovog arhitekturnog obrasca je uska povezanost svih komponenata.



Sl. 2.1. *MVC* arhitektura

U *MVC*-u *Model* predstavlja podatke i većinu poslovne logike aplikacije. Uključuje interakcije s vanjskim servisima ili bazama podataka, strukturama podataka i algoritmima. Bavi se izračunima, implementacijom poslovnih pravila i upravljanjem integritetom podataka. Komponentu *View* simboliziraju elementi korisničkog sučelja (UI). Fokusira se na davanje informacija korisniku za pregled i dobivanje povratnih informacija od njih. U *MVC*-u, *View* prati izmjene *Model*-a i po potrebi prilagođava njegov prikaz. Osim toga, *Controller*-u prenosi radnje korisnika na obradu. Komponenta *Controller* služi kao posrednik. Dobiva korisnički unos iz *View*-a, tumači ga i zatim komunicira s *Model*-om prema potrebi. Na temelju korisničkih radnji, *Controller* modificira *Model*

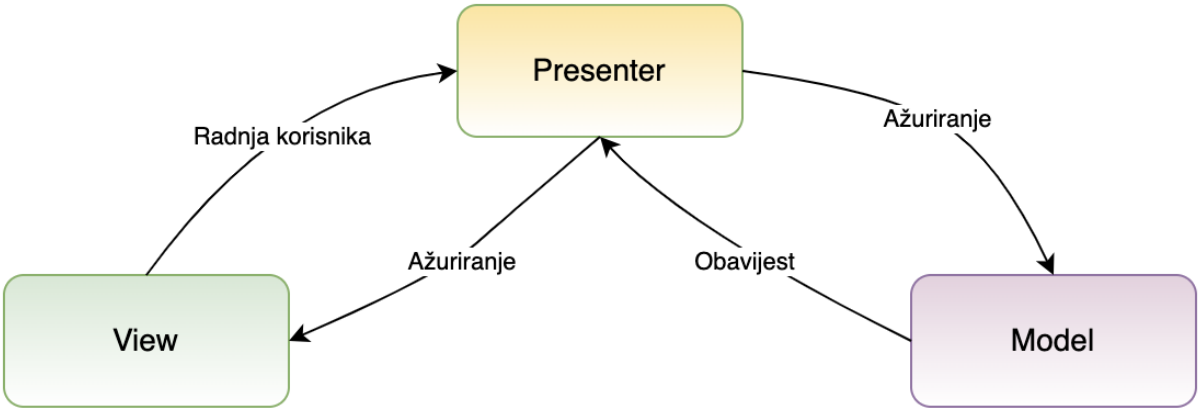
i ažurira pregled svih promjena. Isto tako, prima informacije iz modela i daje ih *View*-u kako bi ih mogao prikazati.

Prednosti *MVC* arhitekture:

- Razdvajanje odgovornosti: *MVC* omogućuje jasnu podjelu između slojeva podataka, poslovne logike i korisničkog sučelja. Bolja organizacija koda, mogućnost održavanja i testiranja omogućeni su ovim odvajanjem.
- Ponovno korištenje koda: odvajanjem *Model*-a, *View*-a i *Controller*-a, razvojni inženjeri mogu ponovno koristiti dijelove u nekoliko značajki ili čak u zasebnim projektima.

2.2. Arhitekturni obrazac MVP

Arhitekturni obrazac *MVP* do nedavno bio je jedan od vodećih arhitekturnih obrazaca u razvoju *iOS* aplikacija, zbog toga što je zadržao jednostavnost arhitekturnog obrasca *MVC*, a rješava problem uske povezanosti razdvajanjem odgovornosti. Na slici 2.2. može se vidjeti dijagram koji predstavlja interakcije komponenata *MVP* arhitekturnog obrasca.



Sl. 2.2. *MVP* arhitektura

Budući da ne želimo da komponenta *View* posjeduje instancu komponente *Model*, instanca komponente *Model* sadržana je unutar komponente *Presenter*. Prilikom interakcije korisnika s komponentom *View*, radnjom rukuje *Presenter*, koji mijenja *Model*. Promjenom komponente

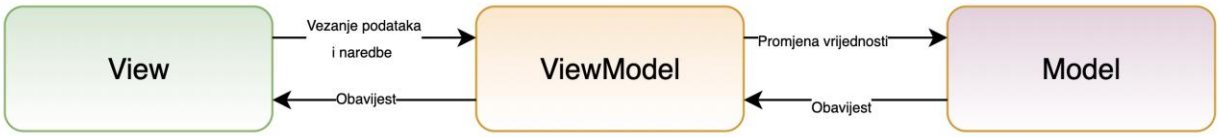
Model, *Presenter* dobiva obavijest, te *Presenter* izvodi potrebne promjene nad komponentom *View*.

Prednosti *MVP* arhitekture:

- Razdvajanje odgovornosti: *MVP* promiče jasno odvajanje između korisničkog sučelja i komponente koja rukuje podacima. Ovo odvajanje omogućuje bolju mogućnost održavanja, testiranje i čitljivost koda.
- Jednostavnije testiranje: ovim arhitekturnim obrascem pojednostavljuje se jedinično testiranje (eng. *unit testing*) jer je poslovna logika odvojena od sloja korisničkog sučelja.

2.3. Arhitekturni obrazac MVVM

Model-View-ViewModel (*MVVM*) prilično je omiljen jer pomaže u razdvajanju problema, potiče mogućnost održavanja koda i poboljšava mogućnost testiranja. Slično kao i *MVP* arhitekturni obrazac, no postoji razlika u dinamici rada komponente koja sadrži poslovnu logiku, što se može vidjeti na slici 2.3. [5].



Sl. 2.3. *MVVM* arhitektura

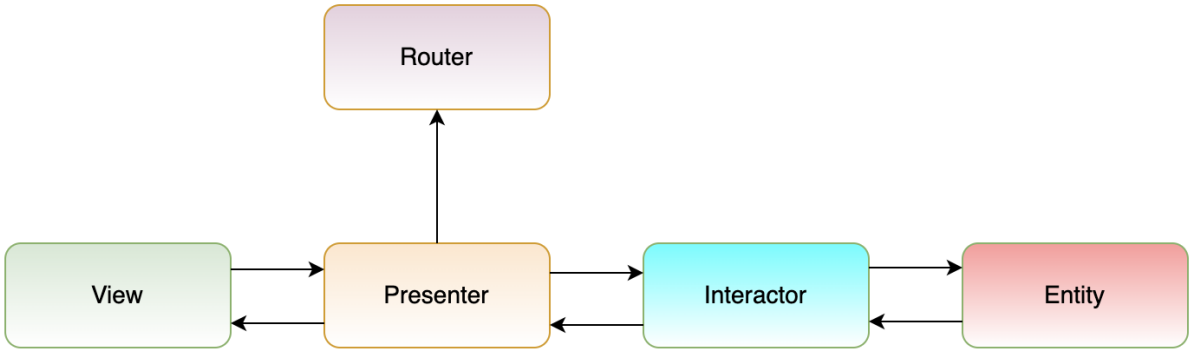
Komponenta *Model* enkapsulira podatkovne strukture, algoritme i interakcije s vanjskim uslugama ili bazama podataka. Komponenta *Model* odgovorna je za upravljanje integritetom podataka, izvođenjem izračuna i implementacijom poslovnih pravila. Komponenta *View* upravlja elementima korisničkog sučelja i njihovim prikazivanjem. Usredotočena je na prikazivanje informacija korisniku i primanje korisničkog unosa. U *MVVM*, komponenta *View* promatra promjene u *ViewModel*-u i ažurira svoj prikaz u skladu s tim. Također prenosi radnje korisnika na *ViewModel* za obradu. *ViewModel* djeluje kao posrednik između *View*-a i *Model*-a. Otkriva podatke i naredbe koje *View* može promatrati. *ViewModel* dohvaća podatke iz *Model*-a, oblikuje ih, nakon čega *View* reagira na promjene unutar *ViewModela*-a, prilagođujući svoj prikaz u skladu s promjenama.

Prednosti *MVVM* arhitekture:

- Modularnost koda: primjena *MVVM* arhitekture vodi do raslojavanja mobilne aplikacije, gdje svaki sloj ima vlastitu odgovornost. Postoji jasna granica između korisničkog sučelja, poslovne logike i dohvaćanja podataka.
- Vezanje podataka (eng. *dana binding*): *MVVM* često koristi radne okvire za vezanje podataka ili *data binding* kao što su *RxSwift* ili *Combine* za uspostavljanje reaktivnog protoka podataka između *View* i *ViewModel* komponenti. Ovo pojednostavljuje ažuriranje elemenata korisničkog sučelja i smanjuje potrebu za ručnom sinkronizacijom podataka.
- Mogućnost testiranja: *MVVM* poboljšava mogućnost testiranja jer je poslovna logika odvojena od korisničkog sučelja. *ViewModel* se može testirati neovisno o korisničkom sučelju, što omogućuje sveobuhvatno jedinično testiranje.
- Recikliranje koda: odvajanjem *View*-a od logike u *ViewModel*-u, obje se komponente mogu ponovno koristiti u više slučaja ili čak u različitim projektima, što promiče smanjenje razvojnog napora.

2.4. Arhitekturni obrazac VIPER

VIPER je arhitekturni obrazac kao i *MVC* ili *MVVM*, ali dodatno odvaja kod na individualne odgovornosti. *MVC* motivira razvojne inženjere da svu poslovnu logiku pohrane u potklasu *UIViewController*-a. *VIPER*, kao i *MVVM* prije njega, nastoji riješiti ovaj problem. U odnosu na navedene arhitekture, *VIPER* odvaja još dvije odgovornosti čime nastaju dvije komponente, kao što je vidljivo na slici 2.4.



Sl. 2.4. *VIPER* arhitektura

Komponenta *View p*redstavlja korisničko sučelje. Za ovu komponentu najčešće se koristi odgovarajuća *SwiftUI* komponenta *View*. Komponenta *Interactor* sadrži poslovnu logiku i izvodi potrebne operacije nad podacima. Komunicira s vanjskim uslugama, dohvaća podatke i izvodi potrebne izračune ili transformacije. Odgovorna je za obradu poslovnih podataka i ažuriranje *Presenter*-a rezultatom. Komponenta *Presenter* djeluje kao posrednik između *View*-a i *Interactor*-a. Prima radnje korisnika iz *View*-a, obrađuje ih i u skladu s tim komunicira s *Interactor*-om. *Presenter* dohvaća podatke iz *Interactor*-a, oblikuje ih i daje *View*-u za prikaz. Komponenta *Entity* predstavlja podatkovne objekte koji se koriste u aplikaciji. Može biti jednostavan podatkovni model ili prikaz kompleksnog objekta domene. Komponenta *Router* upravlja navigacijom između zaslona. Obično se sastoji od metoda kojima je povratni tip implementacija *View* komponente.

Prednosti *VIPER* arhitekture:

- Modularnost: *VIPER* promovira visoko modularnu strukturu, omogućujući jednostavno dodavanje ili uklanjanje komponenti. Svaka komponenta ima određenu odgovornost, što olakšava razumijevanje, održavanje i proširenje aplikacije.
- Mogućnost testiranja poboljšana je *VIPER* dizajnom budući da se svaka komponenta može testirati pojedinačno. S obzirom da je dohvaćanje podataka sadržano u *Interactor*-u, a logika je smještena u *Presenter*-u, upravljanje *unit testingom* postaje još jednostavnije.
- Skalabilnost: *VIPER* podržava skalabilnost *iOS* aplikacija. S modularnom strukturom i jasnim odvajanjem odgovornosti, razvojni inženjeri mogu dodavati nove značajke ili module bez većeg utjecaja na postojeću bazu koda.
- Suradnja: *VIPER* omogućuje kolaborativni razvoj omogućujući programerima da rade na različitim komponentama istovremeno, budući da se svaka komponenta može razvijati neovisno.

3. RAZVOJ iOS APLIKACIJA

Brojni postupci i industrijski standardi trebaju se slijediti prilikom izrade *iOS* aplikacije kako bi se zajamčila njezina visoka kvaliteta i jednostavnost upotrebe. Postoje uobičajene prakse i načela kojih se *iOS* programeri pridržavaju, iako točan proces razvoja može varirati ovisno o potrebama projekta i preferencijama pojedinca. Definiranje ideje i konceptualizacija aplikacije početna je faza u izradi *iOS* aplikacije. Potrebno je identificirati ciljnu publiku, razumjeti njihove zahtjeve te razmisliti o značajkama i funkcijama kako bi se zadovoljile te potrebe. Nakon što se uspostavi koncept aplikacije, slijedi prikupljanje specifičnih potreba. Da bi se opisala funkcionalnost aplikacije, korisničko sučelje i cjelokupno korisničko iskustvo podrazumijeva provođenje istraživanja tržišta, ispitivanje usporedivih aplikacija i razgovor s klijentima. Završetkom istraživanja zahtjeva, počinje faza dizajna. Cilj faze dizajna je razviti korisničko sučelje za aplikaciju koje je jednostavno za korištenje i vizualno privlačno. *Apple's Human Interface Guidelines,* smjernice i preporuke za dizajn, često se slijede tijekom razvoja *iOS* aplikacija. *Wireframes, mockups* i prototipovi često se izrađuju pomoću softvera za dizajn poput *Adobe XD* ili *Sketch* [6].

Prije početka razvoja ključno je isplanirati tehničku arhitekturu aplikacije. Odabir arhitekturnog obrasca, kao što je *MVC, MVP, MVVM* ili *VIPER*, odabir odgovarajućih okvira i biblioteka, te definiranje ukupne strukture projekta. *iOS* aplikacije obično se izrađuju pomoću *Swifta*, programskog jezika koji je razvio *Apple*. Pisanje koda za implementaciju značajki, funkcionalnosti i elemenata korisničkog sučelja aplikacije dio je faze razvoja. Kodiranje, otklanjanje pogrešaka i testiranje često se obavljaju u *XCode*-u, službenom integriranom razvojnom okruženju (eng. *IDE*) za razvoj aplikacija za *iOS*.

Testiranje je ključni korak za verifikaciju i validaciju funkcionalnosti, performansi i korisničkog iskustva aplikacije. Sadrži niz tehnika testiranja, uključujući testiranje prihvatljivosti korisnika, integracijsko testiranje i *unit testing*. Automatizirano testiranje često se provodi pomoću programa kao što su *XCTest* i *Appium*, a ručno testiranje provodi se na stvarnim uređajima i simulatorima. Aplikacija se može objaviti u *App Store*-u nakon što prođe opsežno testiranje i dobije odobrenje. To podrazumijeva postavljanje *Apple*-ovog računa razvojnog programera, pripremu programa za objavu i pridržavanje *Apple*-ovih pravila za snimke zaslona, metapodatke i potpisivanje koda. Nakon toga, aplikacija prolazi kroz *Apple*-ovu proceduru odobrenja prije nego što bude dostupna za preuzimanje u *App Store*-u. Ažuriranje i održavanje aplikacije potrebni su za rješavanje pogrešaka, dodavanje novih značajki i osiguravanje kompatibilnosti aplikacije s najnovijim

verzijama *iOS*-a nakon objavljivanja. Kvaliteta i zadovoljstvo korisnika softvera poboljšavaju se rutinskim ažuriranjem, analizom povratnih informacija korisnika i praćenjem performansi.

3.1. Operacijski sustav iOS

iOS mobilni je operacijski sustav koji je razvio *Apple Inc*. Posebno je dizajniran za *Appleov* mobilni uređaj, *iPhone*. Pruža bogato i intuitivno korisničko sučelje sa širokim rasponom značajki i funkcionalnosti. Elegantno sučelje prilagođeno korisniku karakterizira njegova jednostavnost i lakoća korištenja. Koristi geste, interakcije temeljene na dodiru i intuitivne elemente navigacije kako bi pružio besprijekorno korisničko iskustvo. *iOS* platforma može se pohvaliti robusnim i opsežnim *App Store*-om koji broji milijune aplikacija u raznim kategorijama. Korisnici mogu jednostavno preuzeti i instalirati aplikacije izravno iz *App Store*-a na svoje *iOS* uređaje, poboljšavajući funkcionalnost i svestranost svojih uređaja.

3.2. Smjernice za ljudsko sučelje

Human Interface Guildelines (*HIG*) skup su načela dizajna i preporuka koje daje *Apple* za razvoj korisničkih sučelja na svojim platformama, uključujući *iOS, macOS, watchOS* i *tvOS*. Ove smjernice imaju za cilj osigurati dosljedno, upotrebljivo i kohezivno korisničko iskustvo na svim *Apple* uređajima. *HIG* razvojnim inženjerima pruža najbolje prakse i načela dizajna za stvaranje intuitivnih i vizualno privlačnih korisničkih sučelja. U *HIG*-u nalaze se i načela vizualnog dizajna, uključujući učinkovitu upotrebu boja, tipografije, ikona i slika. Naglašava se važnost stvaranja vizualno privlačnog korisničkog sučelja koje je u skladu s *Apple*-ovom estetikom. Sadrži preporuke za korištenje hijerarhijskih navigacijskih struktura, pružanje jasnih navigacijskih znakova i osiguravanje dosljednog postavljanja uobičajenih elemenata korisničkog sučelja poput gumba i izbornika.

HIG naglašava važnost dizajniranja uključivih i pristupačnih korisničkih sučelja. Pruža smjernice za podržavanje značajki pristupačnosti, kao što su *VoiceOver, Dynamic* Type i pomoćne tehnologije. *HIG* potiče razvojne inženjere da svoje aplikacije učine pristupačnima korisnicima s invaliditetom i da razmotre pristupačnost tijekom procesa dizajna i razvoja [7].

3.3. Programski jezik Swift

Swift je programski jezik za *iOS, macOS, watchOS, tvOS* i *Linux* aplikacije. Stvorio ga je *Apple* 2014. godine, te uz potporu jedne od najutjecajnijih tehnoloških tvrtki na svijetu, *Swift* postaje dominantan jezik za razvoj *iOS*-a i šire. Kreatori *Swift*-a spoznali su da, kako bi se izgradio definirajući programski jezik, tehnologija mora biti otvorena za sve. Dakle, unutar svojih sedam godina postojanja, Swift je stekao veliku podršku zajednice i obilje alata trećih strana. Njegova sintaksa potiče pisanje čistog i dosljednog koda koja se ponekad čak može činiti strogom. Swift pruža zaštitne mjere za sprječavanje pogrešaka i poboljšanje čitljivosti. Swift je realiziran imajući performanse u vidu: kao što je navedeno na službenoj stranici *Apple*-a, *Swift* je do 2,6 puta brži od *Objective-C* i 8,4 puta brži od *Python*-a, čime opravdava svoje ime. Budući da je rangiran kao superiorniji u odnosu na *Objective-C, Swift* je postavljen na 19. mjesto među najpopularnijim programskim jezicima 2023. (dok je *Objective-C* na 24. mjestu) i na 12. mjesto među najomiljenijim jezicima, prema [8].

3.4. Integrirano razvojno okruženje Xcode

Xcode je integrirano razvojno okruženje (*IDE*) koje je stvorio *Apple* za razvoj softverskih aplikacija za *iOS, iPadOS, macOS, watchOS, tvOS* i šire. Uključuje skup alata koje razvojni inženjeri mogu koristiti za pisanje koda, uklanjanje pogrešaka i testiranje softvera, kao i alate za upravljanje projektnim datotekama i resursima. Od svog prvog izdanja 2003. godine, *Xcode* je prešao dug put, a najnovije stabilno izdanje je verzija 14.3, izdana 30. ožujka 2023. *XCode* podržava veliki izbor programskih jezika. To uključuje *C, C++, Objective-C, Objective-C++, Java, AppleScript, Python, Ruby, ResEdit* i *Swift*, s različitim modelima programiranja, uključujući ali ne ograničavajući se na *Cocoa* i *Java* [9].

3.5. Testiranje iOS aplikacija

Za testiranje *iOS* aplikacija, razvojni inženjeri i testeri koriste kombinaciju alata i okvira koji su posebno dizajnirani za testiranje *iOS* aplikacija. *Xcode, Apple*-ovo integrirano razvojno okruženje (*IDE*), pruža ugrađene mogućnosti testiranja za *iOS* aplikacije. Uključuje *XCTest, Apple*-ov radni okvir za testiranje. *Xcode* omogućuje razvojnim inženjerima stvaranje, pokretanje i analizu testova unutar *IDE*-a. *XCTest* je *Apple*-ov službeni radni okvir za testiranje za *iOS* aplikacije. Podržava pisanje i izvođenje *unit* testova i testova korisničkog sučelja koristeći *Swift* ili *Objective-C*. *XCTest*

pruža različite *API*-je za utvrđivanje očekivanih rezultata, interakciju s elementima korisničkog sučelja aplikacije i mjerenje izvedbe. *Appium* je *open-source* okvir za testiranje na više platformi koji omogućuje automatizirano testiranje *iOS*, *Android* i *web* aplikacija. Koristi standardne *API*-je za automatizaciju i podržava više programskih jezika, uključujući *Java*, *JavaScript* i *Python*. *Appium* omogućuje pisanje testova koji su u interakciji s elementima korisničkog sučelja aplikacije i simuliraju radnje korisnika [10].

3.6. Distribucijska platforma App Store

App Store je internetsko tržište koje je razvilo i kojim upravlja *Apple*. To je platforma za digitalnu distribuciju posebno dizajnirana za *iOS* uređaje, uključujući *iPhone*, *iPad* i *iPod Touch*. *App Store* korisnicima omogućuje otkrivanje, preuzimanje i instaliranje širokog spektra aplikacija razvijenih za *Apple*-ov ekosustav. *App Store* je dom za milijune aplikacija u raznim kategorijama, uključujući igre, aplikacije za produktivnost, zabavu, obrazovanje itd. Nudi raznolik izbor aplikacija koje zadovoljavaju različite interese i potrebe korisnika. Moguće je preuzimanje besplatnih aplikacija i kupnju aplikacija koje se plaćaju izravno sa svojih *iOS* uređaja. Također podržava kupnju unutar aplikacija, koje korisnicima omogućuju isključivanje dodatnih značajki, sadržaja ili virtualnih dobara unutar aplikacije. Korisnici mogu ocjenjivati i recenzirati aplikacije na *App Store*-u, dajući povratne informacije i uvid u svoje iskustvo s aplikacijama. Ocjene i recenzije aplikacija pomažu korisnicima da donesu informirane odluke i daju razvojnim inženjerima vrijedne povratne informacije za poboljšanje svojih aplikacija [11].

4. IMPLEMENTACIJE MVVM I VIPER ARHITEKTURNIH OBRAZACA NA PRIMJERU ZNAČAJKE UPRAVLJANJE SUGLASNOSTIMA

Implementacije navedenih arhitektura prikazane su na primjeru funkcionalnosti aplikacije za elektroničke kartone pacijenata – upravljanje suglasnostima (eng. *Consent Management)*. Ideja funkcionalnosti upravljanje suglasnostima pruža korisniku na izbor korištenje nekih od ostalih funkcionalnosti aplikacije za elektroničke kartone pacijenata. Početna vrijednost suglasnosti je negativna, što znači da prilikom prvog ulaska u aplikaciju korisnik nema uključene određene funkcionalnosti. U ovom primjeru, funkcionalnosti koje zahtijevaju suglasnost su upravljanje lijekovima (eng. *Medication Management*) i vremenska crta (eng. *Timeline*).

4.1. Korištene tehnologije i programski alati

U ovom potpoglavlju prikazane su korištene tehnologije i programski alati koji su bili potrebni za razvoj značajke upravljanje suglasnostima. Navedeni su: uređaj za razvojni proces, integrirano razvojno okruženje, programski jezik i radni okviri te uređaj korišten za testiranje i ispravljanje grešaka.

4.1.1. Uređaj za razvojni proces

Apple je stvorio situaciju u kojoj se samo *Xcode* može koristiti za izradu i potpisivanje koda *iOS* aplikacija. *Xcode* je dostupan samo za *macOS*. Dakle, za jednostavno pokretanje projekta vezanog za *iOS* razvoj potreban nam je *Mac*. Nažalost, imati ažuran *Mac* dostupan svima ili cijelom timu nije jednostavno, isplativo niti održivo, zbog čega su se pojavile određene metode pokretanja *Xcode*-a na računalima koje nisu *Mac*. Praktični dio ovog diplomskog rada izveden je na računalu *MacBook Pro*, proizveden 2019. godine. Za vrijeme implementacije značajke *Upravljanje suglasnostima*, računalo je koristilo *macOS* verziju 13, pod imenom *Ventura*.

4.1.2. Integrirano razvojno okruženje

Korišteno integrirano razvojno okruženje je *Xcode*, razvijen od strane *Apple Inc*. Verzija *Xcode*-a korištena za razvoj navedene značajke je 14.2, koja je puštena u javnost 13. prosinca 2022. *Xcode* je dizajniran posebno za izradu softverskih aplikacija za *Apple*-ove platforme. Razvojnim

inženjerima pruža alate i radne okvire koji su optimizirani za jedinstvene značajke i mogućnosti. Redovito se ažurira najnovijim tehnologijama i značajkama iz *Apple*-a, uključujući nove radne okvire, *API*-je i alate. Korištenjem *Xcode*-a razvojni inženjeri mogu osigurati da su njihove aplikacije ažurirane s najnovijim značajkama platforme.

4.1.3. Programski jezik i radni okviri

Programski jezik korišten za realizaciju značajke *Upravljanje suglasnostima* je *Swift,* verzija 5.7. Ova verzija uvodi još jedan veliki skup promjena i poboljšanja jezika, uključujući snažne značajke kao što su regularni izrazi, olakšanje razvojnim inženjerima kao što je skraćena sintaksa i mnoštvo poboljšanja dosljednosti u vezi ključnih riječi programskog jezika. Radni okvir korišten za implementaciju korisničkog sučelja je *SwiftUI*. *SwiftUI* je alat koji omogućuje deklarativnu izradu aplikacija. Drugim riječima, kažemo *SwiftUI*-ju kako želimo da naše korisničko sučelje izgleda i funkcionira, a samom radnom okviru prepuštamo odgovornost kako to radi i kako to implementirati kada korisnik s njim komunicira. U usporedbi s imperativnim korisničkim sučeljem, koje su *iOS* razvojni inženjeri koristili prije *iOS*-a 13, deklarativno korisničko sučelje je razumljivije. Često mijenjamo izgled i funkcionalnost korisničkog sučelja ovisno o tome što se događa. Na primjer, u imperativnom korisničkom sučelju, mogli bismo stvoriti funkciju koja se poziva kada se klikne gumb da pozove vrijednost i prikaže oznaku na korisničkom sučelju unutar funkcije. Imperativni način rada uzrokuje razne vrste problema, od kojih se većina vrti oko stanja. Moramo pratiti u kakvom je stanju aplikacija i osigurati da naše korisničko sučelje ispravno odražava to stanje. Deklarativni način rada, s druge strane, omogućuje nam da istovremeno informiramo *iOS* o svakom potencijalnom stanju naše aplikacije. Kada smo prijavljeni, možemo prikazati poruku dobrodošlice, a kada smo odjavljeni, možemo prikazati gumb za prijavu. Ručni prijelaz između ta dva stanja imperativna je metoda rada [12] [13].

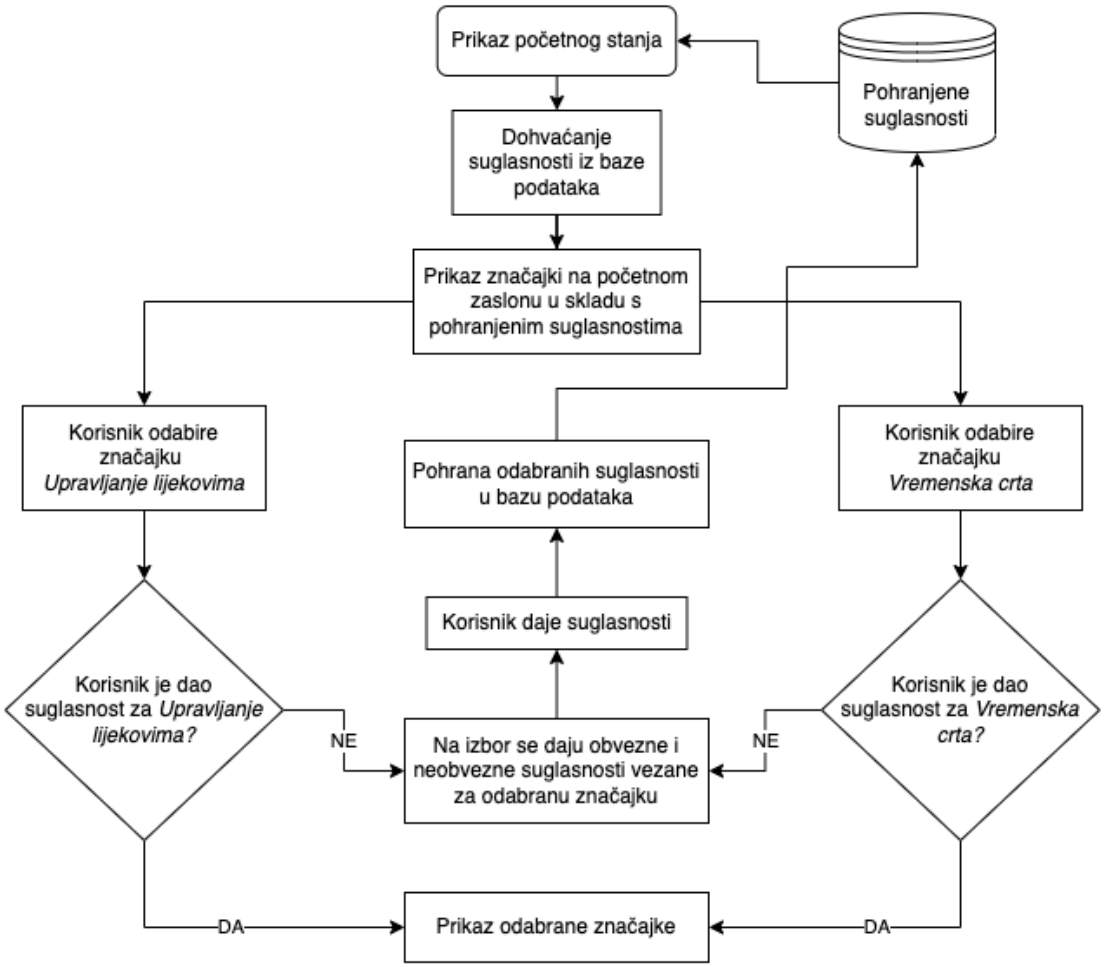
Combine radni okvir korišten je za asinkrono procesiranje vrijednosti. *Combine* je deklarativni *API* za rukovanje promjenama vrijednosti tijekom vremena. Kod za rad s komponentama kao što su delegati, upozorenja, mjerači vremena, *completion* blokovi i *callbacks* mogu se objediniti i učiniti jednostavnijim uz pomoć ovog radnog okvira. *RXSwift,* reaktivni okvir treće strane, već je neko vrijeme dostupan na *iOS*-u, ali *Apple* je 2019. godine najavio vlastiti na *WWDC*-u (*WorldWide Developers Conference*) [14] [15].

4.1.4. Uređaj korišten za testiranje i ispravljanje grešaka

Za testiranje i ispravljanje grešaka korišten je *iPhone* 12, s operacijskim sustavom *iOS* 16. Osim fizičkog uređaja, u velikoj mjeri korišten je i simulator *iOS* uređaja različitih modela i verzija sustava. Simulator je u sklopu integriranog razvojnog okruženja *Xcode*.

4.2. Značajka za upravljanje suglasnostima

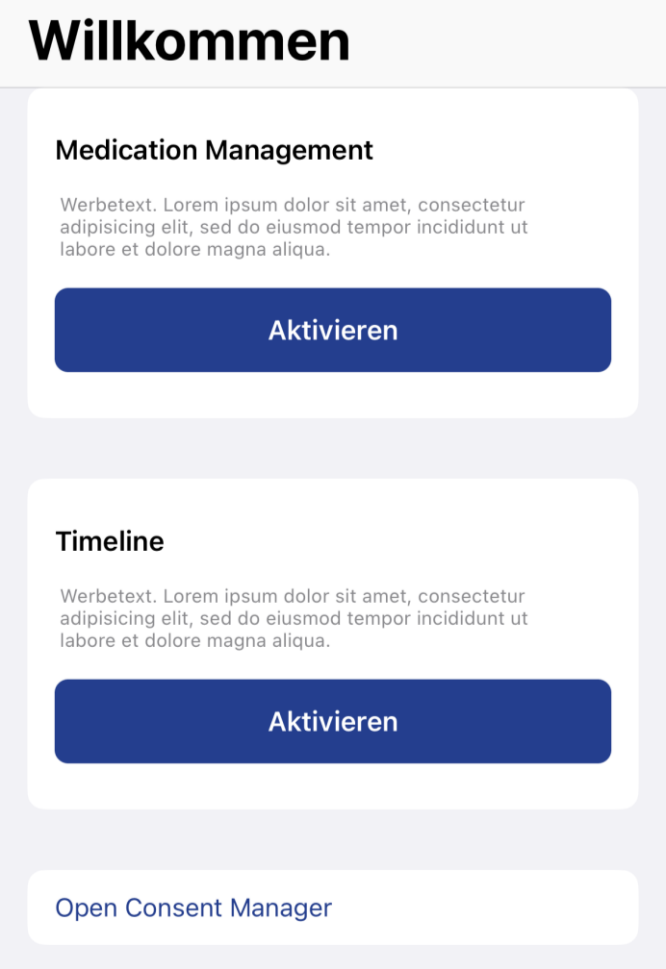
Upravljanje suglasnostima značajka je aplikacije za elektroničke kartone pacijenata, kojom korisnik može upravljati svojim podacima tako da daje ili poništava svoju suglasnost za određene funkcionalnosti aplikacije koje koriste osobne podatke. Funkcionalnosti aplikacije kojima su potrebne suglasnosti su upravljanje suglasnostima i vremenska crta. Na slici 4.1. prikazan je dijagram tijeka korištenja značajke upravljanja suglasnostima.



Sl. 4.1. *Dijagram tijeka značajke za upravljanje suglasnostima*

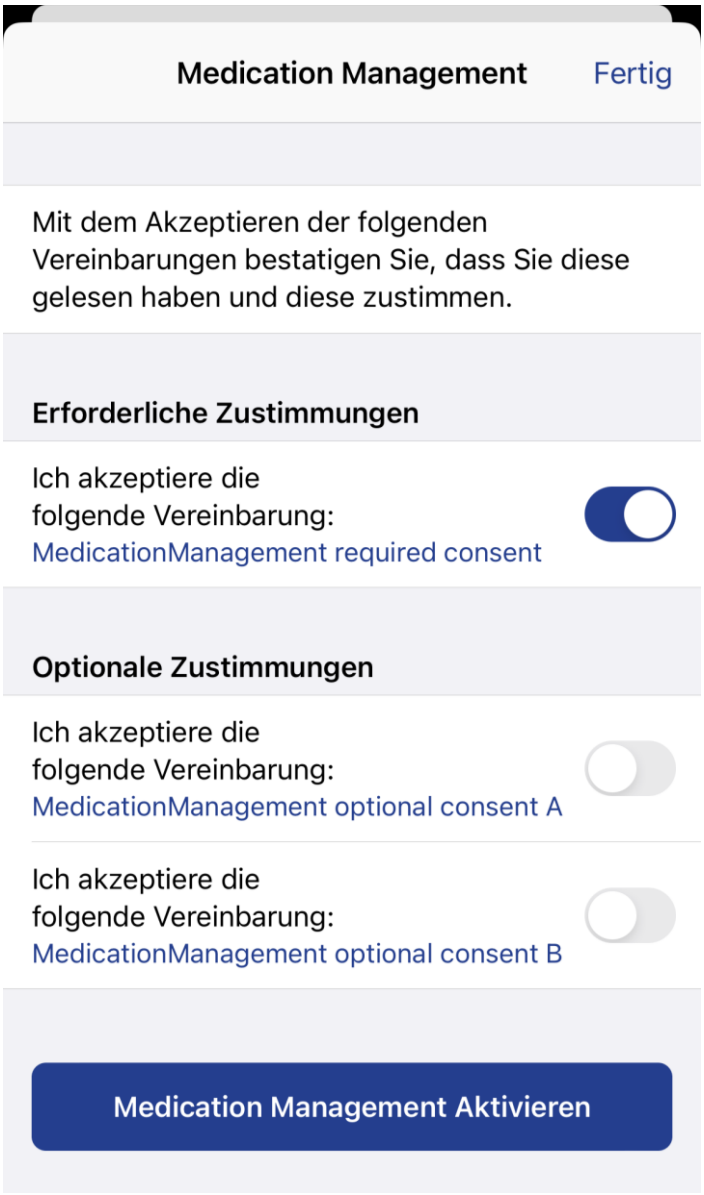
4.2.1. Prikaz tijeka značajke

Prilikom pokretanja aplikacije, učitava se početno stanje, što znači stanje bez suglasnosti. Prilikom pojavljivanja korisničkog sučelja, iz baze podataka dohvaćaju se pohranjene suglasnosti. Na slici 4.2. može se vidjeti stanje bez danih suglasnosti.



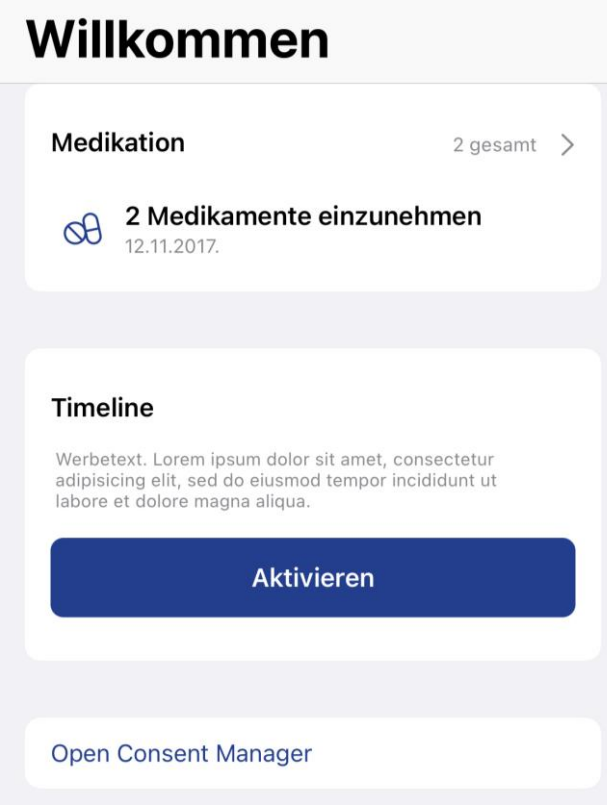
Sl. 4.2. *Stanje bez suglasnosti*

Korisnik se nalazi na početnom zaslonu, s početnim stanjem – bez danih suglasnosti. Odabirom jednog od dva gumba, korisnik može pokrenuti željenu značajku. U slučaju da korisnik odabere prvu značajku, upravljanje lijekovima, prikazuje se zaslon s opcijama koje omogućuju korisniku davanje suglasnosti za odabranu značajku, kao na slici 4.3. U slučaju ove značajke, postoji jedna obavezna suglasnost i dvije neobvezne. Budući da je gumb za nastavak onemogućen, nije moguće nastaviti dok se ne da suglasnost za željenu značajku ili dok se korisnik ne vrati korak unazad. Prilikom davanja suglasnosti, omogućuje se tipka za aktivaciju značajke, čime se pohranjuje dana suglasnost.

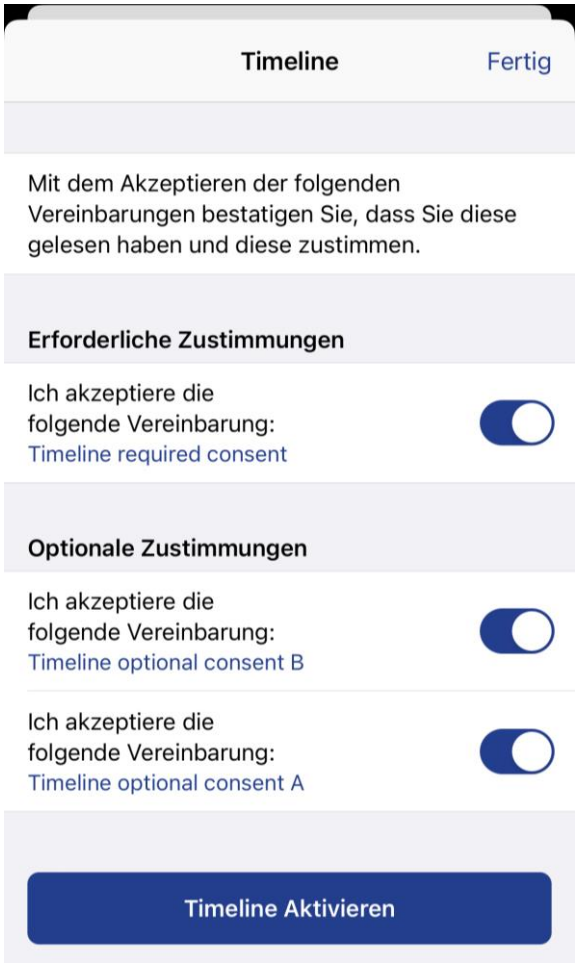


Sl. 4.3. *Zaslon za davanje suglasnosti za značajku upravljanja lijekovima*

Prilikom aktivacije značajke, aplikacija vodi korisnika na početni zaslon, s tim da će se ovaj put prilikom dohvaćanja suglasnosti, dohvatiti dana suglasnost za značajku upravljanja lijekovima, te će biti vidljiv zaslon kao na slici 4.4. Budući da je suglasnost za upravljanje lijekovima dana, umjesto odjeljka za aktivaciju, prikazuje se personalizirani odjeljak, koji je povezan s logikom značajke. Moguće je aktivirati značajku vremenske crte, čime se pojavljuje zaslon kao na slici 4.5. Na novom zaslonu moguće je dati suglasnost za značajku vremenske crte. Za svrhu primjera, korisnik odlučuje dati sve moguće suglasnosti: dvije neobvezne i jednu obveznu, čime se omogućuje gumb za aktivaciju značajke.

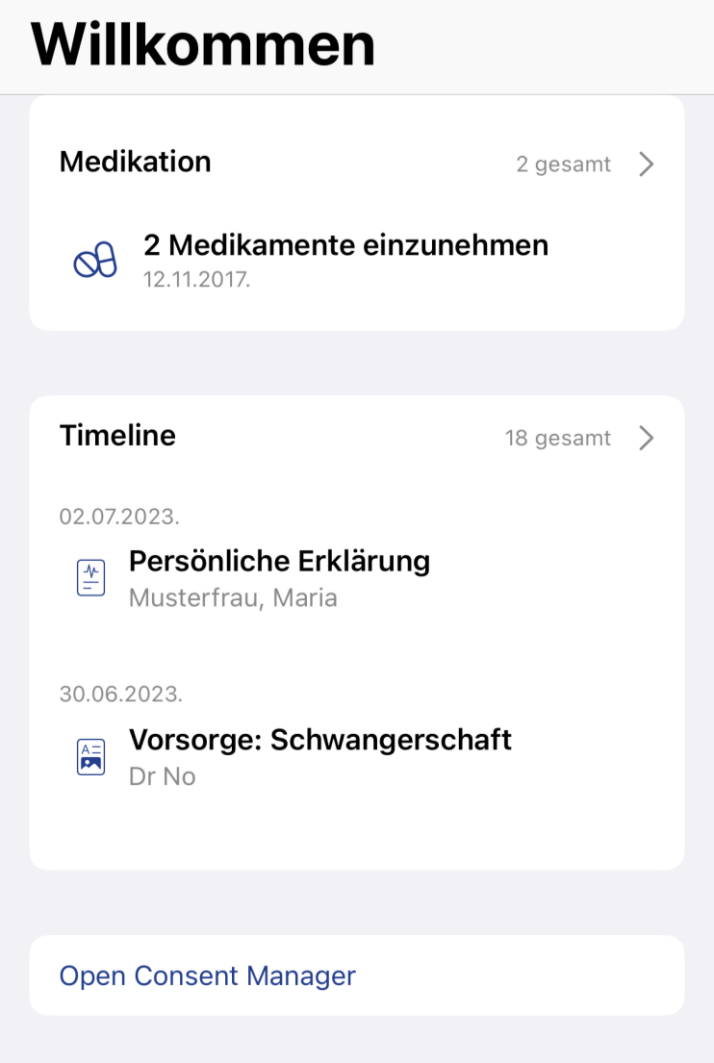


Sl. 4.4. *Početni zaslon s danom suglasnosti za upravljanje lijekovima*



Sl. 4.5. *Zaslon za davanje suglasnosti za značajku vremenske crte*

Odabirom aktivacije značajke, korisnik se vraća na početni zaslon. Ovog puta, obje značajke su aktivirane, jer su dohvaćene pohranjene suglasnosti iz baze podataka. Tako se može vidjeti personalizirana značajka upravljanja lijekovima, ali ovog puta i personalizirana značajka vremenske crte. Značajka vremenske crte podsjeća korisnika na pretrage koje su zakazane i prikazuje prethodno obavljene pretrage. Na slici 4.6. može se vidjeti primjer podsjetnika za neke od pretraga. Na dnu zaslona može se vidjeti *Open Consent Manager*, koji otvara prikaz svih danih suglasnosti, uz mogućnost odabira suglasnosti i poništavanja suglasnosti.



Sl. 4.6. *Početni zaslon s danim suglasnostima*

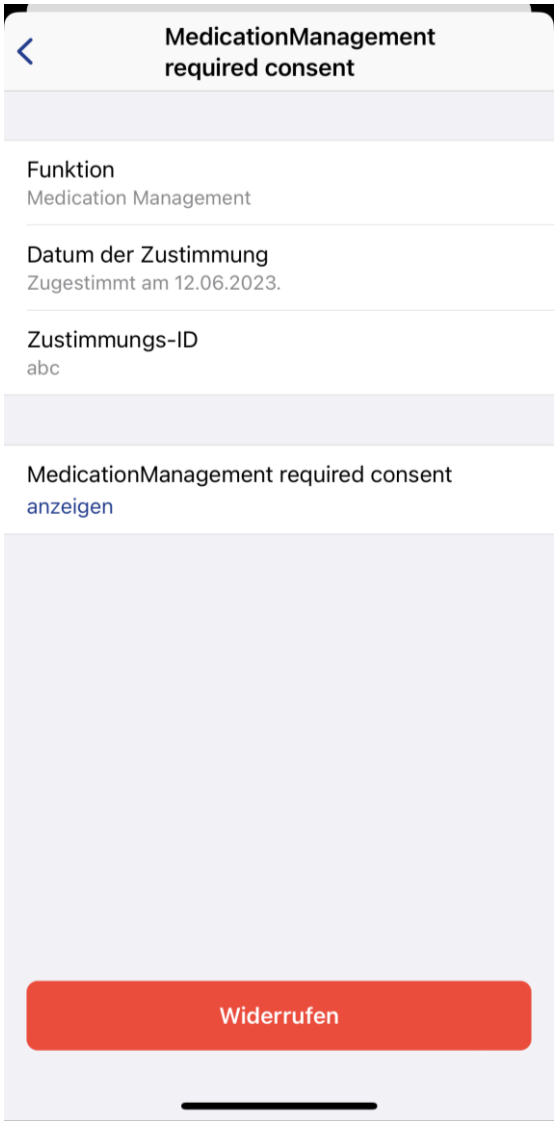
Ako korisnik odluči pregledati dane suglasnosti, može to učiniti pritiskom na gumb *Open Consent Manager*. U tom slučaju pokazuje se popis svih danih suglasnosti, kategoriziranih prema značajki na koju se odnose. Na suglasnostima se može vidjeti koja je suglasnost obavezna i koja nije. Također, može se vidjeti koja je suglasnost dana i koja nije. Ako je suglasnost dana, vidljiv je i datum na koji je određena suglasnost dana, kao na slici 4.7. Osim mogućnosti odabira suglasnosti i njihovog pregleda, moguće je vratiti se na početni zaslon pritiskom na gumb postavljen u gornjem desnom uglu, prateći dijagram tijeka sa slike 4.1.

Sl. 4.7. *Prikaz suglasnosti*

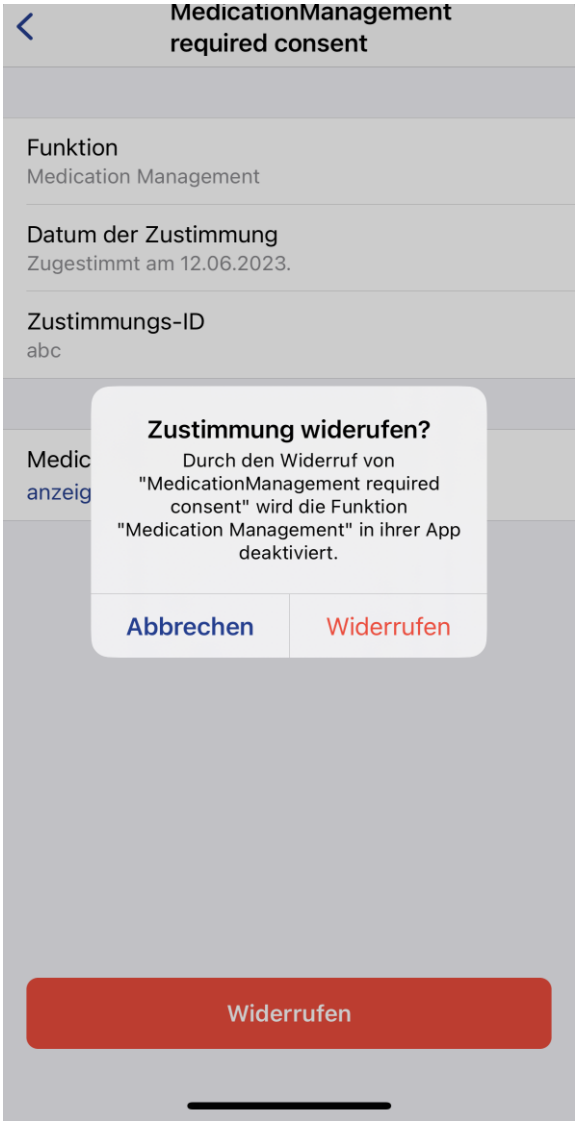
Odabirom jedne od suglasnosti prikazuje se zaslon na slici 4.8. Prikazuje naziv značajke na koju se suglasnost odnosi, datum davanja suglasnosti i *ID* suglasnosti koji je jedinstven za pojedinu značajku. Na dnu postoji crveni gumb koji simbolizira destruktivnu radnju u *iOS* sustavu, a služi za ukidanje prethodno dane suglasnosti.

Pritiskom na destruktivni gumb, pojavljuje se zaslon sa slike 4.9. Zaslon ne zauzima cijelu površinu uređaja, već radi na principu skočnog prozora. Sastoji se do naslova, poruke, opcije prekida i destruktivne opcije. Opcija prekida plave je boje i sve što radi je uklanja skočni prozor bez interakcije s bazom podataka. Destruktivna opcija briše suglasnost iz baze podataka.

Ako je suglasnost koje je ukinuta obvezna, automatski se ukidaju i neobvezne suglasnosti. Ako je suglasnost neobvezna, korisnik ostaje na istom zaslonu. Na isti princip radi i ukidanje suglasnosti za ostale značajke.

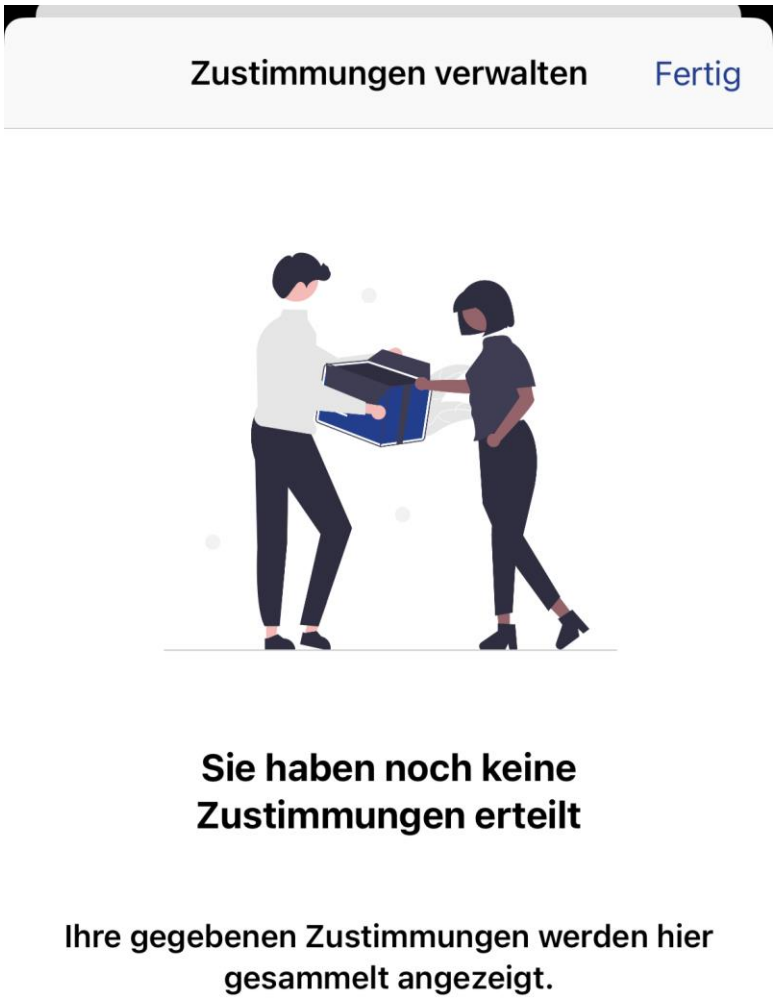


Sl. 4.8. *Detalji odabrane suglasnosti*



Sl. 4.9. *Detalji odabrane suglasnosti*

Ako se otvori upravljanje suglasnostima u stanju aplikacije kad niti jedna suglasnost nije dana, prikazuje se zaslon sa slike 4.10.

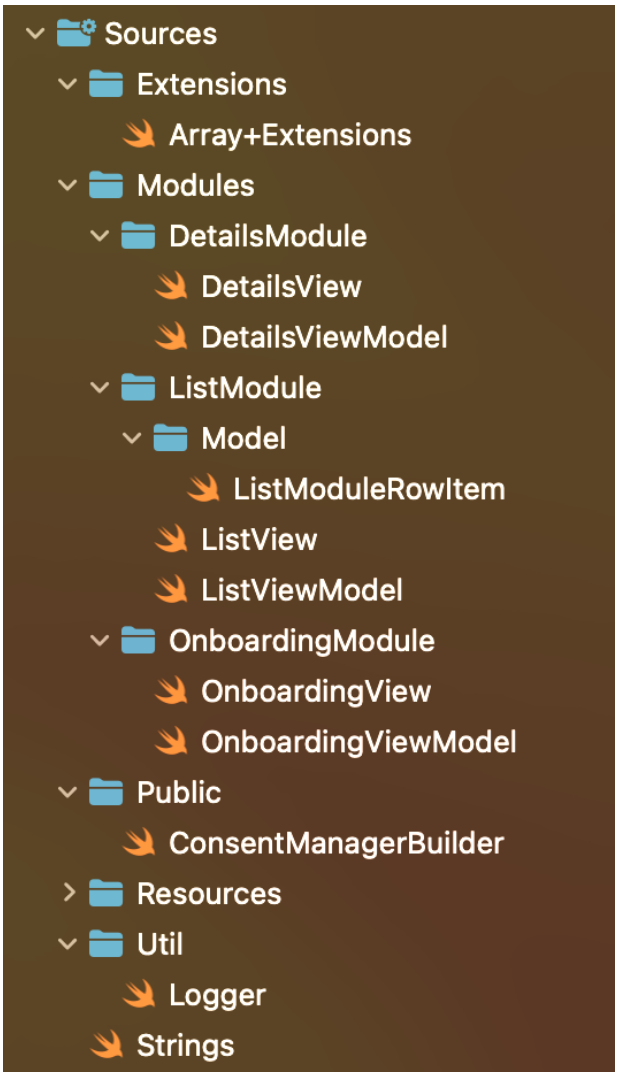


Sl. 4.10. *Prikaz praznog stanja*

Navedeni zaslon prikazuje se u takozvanom praznom stanju ili *empty state*. Prikazuje se poruka koja govori da nema danih suglasnosti, uz prikladnu ilustraciju. Korisniku se nudi opcija prekida tijeka u gornjem desnom uglu.

4.3. Implementacija MVVM arhitekturnog obrasca na primjeru značajke za upravljanje suglasnostima

Implementacija određene funkcionalnosti imajući u vidu *MVVM* arhitekturu podrazumijeva određen način imenovanja datoteka i direktorija. Na slici 4.11. mogu se vidjeti datoteke podijeljene po semantički značajnim cjelinama.



Sl. 4.11. *Prikaz MVVM arhitekture na primjeru značajke upravljanja suglasnostima*

Ishodišni direktorij naziva se *Source*, a sadrži direktorije *Extensions, Modules, Public, Resources, Util* i datoteku *Strings*. Direktorij *Extensions* sadrži jednu datoteku pod nazivom *Array+Extensions*, a služi za proširivanje funkcionalnosti standardne klase *Array*. Direktorij pod nazivom *Modules* sadrži tri direktorija, tj. modula. Moduli su cjeline koje su semantički odvojene

u svrhu čitljivosti koda. U ovom primjeru postoji tri modula: *DetailsModule, ListModule* i *OnboardingModule*. *Public* direktorij sadrži točku ulaska u značajku, i njena odgovornost je vjerodostojno prikazivanje stanja u kojoj se aplikacija nalazi. Unutar direktorija *Util* nalazi se jedna datoteka *Logger* koja ima za cilj pojednostavljivanje programskim inženjerima uklanjanje grešaka i ispisivanje željenih vrijednosti u terminalu. Time inženjeri mogu steći uvid u potencijalne ili tekuće probleme u aplikaciji ili značajki. Direktorij *Resources* i datoteka *Sources* povezani su zato što su stvoreni automatiziranom skriptom koja dohvaća *Stringove* s udaljene baze podataka.

4.3.1. Modul za ukrcavanje

Navedena tri modula najbitniji su dio kako arhitekture, tako i same značajke. Module se može shvatiti kao određene dijelove značajke ili određene slučajeve tijeka aplikacije ili značajke do kojih korisnik može doći. Prateći dijagram tijeka sa slike 4.1., možemo vidjeti da je prvo stanje početno stanje, čemu je ekvivalent ove značajke odjeljak s gumbom *Open Consents Manager* kao na slici 4.2. Ova značajka je specifična po tome što neke od ostalih značajki ovise o njoj. Tako možemo inicirati *OnboardingModule* ili modul za ukrcavanje odabirom aktivacije značajke upravljanja lijekovima. Odabirom značajke prikazuje se zaslon kao na slici 4.3., te se prikazuje *OnboardingView*. Konstruktor strukture *OnboardingView* može se vidjeti u programskom kodu 4.1.

```
public init(
    logic: ConsentManagementLogic,
    featureCode: Int
) {
    _viewModel = StateObject(
        wrappedValue: OnboardingViewModel(
            logic: logic,
            code: featureCode
        )
    )
}
```

Programski kod 4.1. *Prikaz konstruktora strukture OnboardingView*

Budući da se navedena struktura može prikazati i aktivacijom značajke upravljanja lijekovima i aktivacijom značajke vremenske crte, bilo je potrebno definirati koje stanje će kada poprimiti ova struktura. Taj problem riješen je uvođenjem parametra *featureCode* u parametarskom konstruktoru, te dodjeljivanjem jedinstvenog broja objema značajkama koje ovise o danim suglasnostima. Unutar tijela konstruktora, može se vidjeti kako se varijabli *viewModel* dodjeljuje vrijednost. Budući da je *viewModel* deklariran uz *PropertyWrapper* tipa *StateObject,* kao što je vidljivo u programskom kodu 4.2., ako se želi dodati vrijednost koja je tipa stanja, dodaje se tako da se varijabli s prefiksom „_“ pridodaje vrijednost novog *StateObject*-a, kojem se vrijednost dodjeljuje preko njegovog konstruktora. U ovom slučaju vrijednost koja se želi dodijeliti je *OnboardingViewModel*. Prema imenu navedene klase da se zaključiti da se radi o arhitekturi *MVVM*. *ViewModel*-u se daje logika aplikacije, budući da će se u *viewModel*-u događati poslovna logika aplikacije, te je potrebno dohvaćanje udaljenih podataka.

```
@StateObject var viewModel: OnboardingViewModel
```

Programski kod 4.2. *Prikaz deklaracije varijable viewModel*

Glavni dio korisničkog sučelja ovog modula vidljiv je u programskom kodu 4.3. Prikaz ovog programskog koda na djelu može se vidjeti na slici 4.5. Glavni element ovog sučelja je *List* koji sadrži jedan odjeljak (*Section*), dva odvojena niza prekidača (*Toggle*) i odjeljka s gumbom za aktivaciju značajke. U prvom odjeljku korisniku se jednostavnim tekstualnim elementom daje do znanja kako se davanjem suglasnosti slaže s uvjetima korištenja. Nakon toga nalaze se dva odjeljka, jedan s popisom obveznih suglasnosti i pripadajućim prekidačem, drugi isto tako samo za neobvezne suglasnosti.

```
List {
    Section {
        Text(ConsentManagement.consentApprovalInfo)
            .textStyle(.listText)
    }
    .rowStyle(.color(.clear))
    .listRowSeparator(.hidden)

    if !viewModel.isLoading {
        listConsents(
            title: ConsentManagement.requiredConsent,
            consents: viewModel.requiredConsents
        )

        listConsents(
            title: ConsentManagement.optionalConsent,
            consents: viewModel.optionalConsents
        )

        Section {
            EmptyView()
        } footer: {
            createActivateButton()
        }
    }
}
.listStyle(.grouped)
.onAppear {
    viewModel.onAppear()
}
```

Programski kod 4.3. *Glavni dio korisničkog sučelja OnboardingView*

Programsko rješenje prikaza dvaju odjeljka s različitima vrstama suglasnosti može se vidjeti u programskom kodu 4.4. Može se vidjeti *Property Wrapper* neposredno iznad funkcije koji znači da funkcija vraća tip *View*. Kako bi omogućilo ponovno korištenje funkcije, ona prima parametar *title* i *consents*. Na ovaj način, ista funkcija može se koristiti za prikaz i obveznih i neobveznih suglasnosti, jednostavnom promjenom proslijeđenih argumenata, kao što se može vidjeti na prethodnoj slici. U ovoj funkciji postavlja se inicijalna vrijednost prekidača i definira se tekst koji stoji uz prekidač. Posljednji element na ovom zaslonu je prikazan u programskom kodu 4.5.

```
@ViewBuilder
private func listConsents(
    title: String, consents: [CMConsent]
) -> some View {
    Section {
        ForEach(consents) { item in
            if let url = URL(string: item.externalLink) {
                let isToggleOn = Binding<Bool>(
                    get: { item.state.isActive },
                    set: { _ in viewModel.changeState(of: item) }
                )

                Toggle(
                    isOn: isToggleOn,
                    label: {
                        BasicRow {
                            Text(ConsentManagement.consentApproval)
                                .textStyle(.listText)
                        } detail: {
                            Link(item.name, destination: url)
                                .buttonStyle(.secondary(style: .inline))
                        }
                    }
                )
                .toggleStyle(.brandedSwitch)
            }
        }
    } header: {
        Text(title)
            .textStyle(.headerTitle)
    }
}
```

Programski kod 4.4. *Prikaz niza suglasnosti s pripadajućim prekidačima*

```
@ViewBuilder
private func createActivateButton() -> some View {
    Button {
        viewModel.storeConsents()
        dismiss()
    } label: {
        Text(viewModel.activateButtonTitle)
    }
    .buttonStyle(.primary)
    .disabled(viewModel.requiredGiven)
}
```

Programski kod 4.5. *Prikaz implementacije gumba za aktivaciju značajke*

CreateActivateButton funkcija je koja definira izgled i ponašanje gumba koji daje mogućnost aktivacije značajke. Posljednja linija koda u tijelu funkcije označava omogućenost gumba, a može se vidjeti da to ovisi o varijabli *requiredGiven* koja se nalazi u *viewModel*-u. Pritiskom na gumb, izvodi se funkcija *viewModel*-a pod nazivom *storeConsents*, koja pohranjuje dane suglasnosti u bazu podatka. Nakon toga, ovaj dio korisničkog sučelja uklanja se sa zaslona.

Unutar klase *OnboardingViewModel* osim nekolicine varijabli kojima se dinamički dodaje vrijednost u programskom kodu 4.6., nalaze se funkcije poslovne logike.

```
var requiredConsents: [CMConsent] {
    consents.filter { !$0.isOptional }
}

var optionalConsents: [CMConsent] {
    consents.filter(\.isOptional)
}

var requiredGiven: Bool {
    !requiredConsents.allSatisfy(\.state.isActive)
}

var navBarTitle: String {
    requiredConsents.first?.category ?? ""
}

var activateButtonTitle: String {
    navBarTitle + " " + ConsentManagement.activate
}
```

Programski kod 4.6. *Prikaz computed variables u viewModelu*

Computed variables mogu se shvatiti kao posebne *getter* funkcije kojima se daje semantički značajan naziv, kojima je potrebno odrediti povratni tip.

Funkcija za dohvaćanje podataka poziva se prilikom pojavljivanja *OnboardingView* strukture na zaslonu, a njena definicija može se vidjeti u programskom kodu 4.7.

```swift
private func loadConsents() {
    isLoading = true
    logic
        .getConsentsByCode(code: code)
        .receive(on: DispatchQueue.main)
        .sink(
            receiveCompletion: { [weak self] completion in
                self?.isLoading = false
                if case let .failure(error) = completion {
                    CMLogger.error(message: String(describing: error))
                }
            },
            receiveValue: { [weak self] consents in
                self?.consents = consents
            }
        )
        .store(in: &cancellables)
}
```

Programski kod 4.7. *Prikaz učitavanja suglasnosti*

Prije početka dohvaćanja podataka, varijabla *isLoading* postavlja se na vrijednost *true*, jer *View* komponenta promatra navedenu varijablu. Promjenom vrijednosti varijable, zaslon će prikazati prikladniji zaslon tijekom učitavanja podataka. Dohvaćanje podataka počinje pozivom funkcije *getConsentsByCode*, koja vraća suglasnosti relevantne za određenu značajku unutar tipa *AnyPublisher*. Taj tip podataka omogućuje operacije iz radnog okvira *Combine*. Funkcijom *receive* definira se vrsta *Queue*-a na kojem će se podaci primiti. U ovom slučaju, podaci se primaju na glavnom *Queue*-u, jer ažuriranja elemenata korisničkog sučelja s pozadinske niti nisu dozvoljena i mogu dovesti do neželjenog ponašanja. Nakon definiranja *Queue*-a, definira se radnja koja će se izvršiti prilikom završetka dohvaćanja, te je moguće iskoristiti varijablu *completion*. Koristeći *completion*, može se provjeriti uspješnost zadane radnje, te adekvatno reagirati na istu. Još jedna *lambda* funkcija koja je prikazana je *receiveValue*, pomoću koje je moguće izvoditi operacije nad podacima koji su primljeni iz baze podataka. U ovom slučaju ne izvode se operacije nad primljenim podacima, već se suglasnosti pohranjuju u varijablu unutar *OnboardingViewModel*-a. Posljednja linija određuje gdje se želi pohraniti ova specifična pretplata ili *subscription*.

4.3.2. Modul popisa

Modul popisa dio je značajke za upravljanje suglasnostima čija je zadaća prikaz svake pojedinačne suglasnosti u obliku popisa, gdje se odabirom jednog elementa popisa, korisnik vodi na modul s detaljima. Na slici 4.7. može se vidjeti modul popisa izgleda unutar aplikacije. Prikazuje se niz suglasnosti, kategoriziranih prema značajki na koju se odnose, u ovom slučaju su to upravljanje lijekovima i vremenska crta. Podaci koji trebaju biti prikazani nalaze se u *ViewModel* komponenti modula popisa, u varijabli pod nazivom *manageConsentsItems*. U programskom kodu 4.8. vidi se provjera postojanja vrijednosti varijable *manageConsentsItems*. Budući da je navedena varijabla deklarirana kao *Optional*, što je karakteristično znakom upitnika nakon imena tipa varijable, prije korištenja varijable potrebno je ustanoviti da varijabla ima vrijednost. Provjera vrijednosti vrši se pomoću *if let* izraza, koji ujedno ispituje sadrži li varijabla s desne strane vrijednost, i pridodaje novoj varijabli *items* vrijednost provjereno postojeće vrijednosti varijable. Ako varijabla *manageConsentsItems* sadrži vrijednost, slijedi daljnja provjera. Budući da je navedena varijabla tipa polje, potrebno je provjeriti postoji li koji element unutar popisa i reagirati na adekvatan način. Provjera postojanja elemenata unutar polja vrši se pomoću ugrađene funkcije *isEmpty* čiji je povratni tip *Bool*. Ako je polje prazno, prikazuje se novi zaslon pod nazivom *EmptyScreenLayout*, koji prikazuje adekvatan naslov, ilustraciju i popratnu poruku.

```
if let items = viewModel.manageConsentsItems {
    if items.isEmpty {
        EmptyScreenLayout(
            title: ConsentManagement.emptyListTitle,
            text: ConsentManagement.emptyListText
        )
```

Programski kod 4.8. *Provjera postojanja vrijednosti varijable i elemenata unutar popisa*

U programskom kodu 4.9. može se vidjeti rješenje za prikaz odjeljaka. Svaki odjeljak sadrži niz suglasnost koji se odnose na značajku trenutnog odjeljka. Za svaku suglasnost trenutnog odjeljka stvara se *BasicDetailRow*, struktura koja prikazuje u jednom retku naziv suglasnosti, a odmah ispod toga prikazuje stanje davanja dotične suglasnosti manjim fontom i svjetlijom bojom. *BasicDetailRow* omotan je pomoću omotača *NavigationWrapper*, koji omogućuje prikazivanje idućeg potrebnog zaslona. Destinacija svakog retka određena je tako da se pritiskom na *BasicDetailRow* postavlja vrijednost željene suglasnosti unutar *ViewModel* komponente, a zatim

se kao destinacija određuje *DetailsView* struktura, kojoj se prosljeđuju potrebni argumenti. Prilikom napuštanja *DetailsView* strukture potrebno je osvježiti podatke što se događa u *onDisappear* funkciji. Svakom promjenom varijable *id* unutar varijable *consent* prikazuje se različita nova *DetailsView* struktura. Kao podnaslov odjeljka postavlja se ime značajke za koju se prikazuju suglasnosti, što je vidljivo u argumentu *header*.

```
    Section {
        ForEach(section.items) { item in
            NavigationWrapper(
                content: {
                    Button(
                        action: {
                            viewModel.consent = item
                        },
                        label: {
                            BasicDetailRow(
                                text: item.name,
                                details: viewModel.getValidFromString(for: item)
                            )
                        }
                    )
                },
                destination: {
                    DetailsView(
                        viewmodel: DetailsViewModel(
                            selectedConsent: item,
                            logic: viewModel.logic,
                            dismiss: { viewModel.consent = nil }
                        )
                    )
                    .onDisappear {
                        viewModel.consent = nil
                        viewModel.onAppear()
                    }
                },
                showView: .constant(viewModel.consent?.id == item.id)
            )
        }
    } header: {
        Text(section.items[0].category)
            .textStyle(.headerTitle)
    }
}
```

Programski kod 4.9. *Prikaz odjeljaka suglasnosti*

U programskom kodu 4.10. mogu se vidjeti različite varijable unutar *ViewModel* komponente na čiju promjenu reagira struktura korisničkog sučelja. Varijabla *manageConsentsItems* tipa je polja strukture *Model* komponente *ListModuleRowItem,* koja je deklarirana kao *Optional*, što znači da je moguće da varijabla u nekom trenutku nema vrijednost.

```
@Published var consent: CMConsent?
@Published var manageConsentsItems: [ListModuleRowItem]?
@Published var isLoading = true
```

Programski kod 4.10. *Varijable unutar ViewModel komponente*

Privatna funkcija *getConsentsForActiveFeatures* u programskom kodu 4.11. prikazuje način dohvaćanja značajki koje su trenutno aktivne, što znači da je za njih dana barem obavezna suglasnost.

```
private func getConsentsForActiveFeatures() {
    isLoading = true
    logic
        .getConsentsForActiveFeatures()
        .receive(on: DispatchQueue.main)
        .sink(
            receiveCompletion: { [weak self] completion in
                self?.isLoading = false
                if case let .failure(error) = completion {
                    CMLogger.error(message: String(describing: error))
                }
            },
            receiveValue: { [weak self] consents in
                self?.manageConsentsItems = consents.groupConsentsByCategory()
            }
        )
        .store(in: &cancellables)
}
```

Programski kod 4.11. *Dohvaćanje aktivnih suglasnosti*

Za dohvaćanje podataka koristi se *Combine* radni okvir. Funkcijom *receive* naređuje se dohvaćanje na glavnoj niti, što je standardna praksa za dohvaćanje podataka koji utječu na korisničko sučelje. Prilikom završetka dohvaćanja podataka, varijabla *isLoading* postavlja se na vrijednost *false*. Ako se dogodi pogreška pri dohvaćanju, ista se ispisuje u konzoli kako bi razvojni inženjer imao uvid u razlog zašto je došlo do pogreške. Ako ne dođe do pogreške, dobivena vrijednost se postavlja za

vrijednost varijable *manageConsentsItems* unutar *ViewModel* komponente. Zadnja linija koja sadrži funkciju *store* omogućuje pohranu navedenog promatranja rezultata dohvaćanja.

Komponenta *Model* modula popisa prikazana je programskim kodom 4.12. Struktura *ListModuleRowItem* implementira protokole *Identifiable* i *Equatable*, što znači da svaka instanca navedene strukture sadrži jedinstveni identifikator, te da je omogućena usporedba jednakosti dvaju instanci tipa *ListModuleRowItem*.

```
struct ListModuleRowItem: Identifiable, Equatable {
    var id: Int
    var items: [CMConsent]
}
```

Programski kod 4.12. *Model komponenta popis modula*

Osim identifikatora, model sadrži i polje tipa *CMConsent*. Navedena klasa predstavlja model jedne specifične suglasnosti koja sadrži mnoštvo varijabli, od kojih su neke identifikator suglasnosti, ime suglasnosti, kategorija suglasnosti i varijabla koja govori je li određena suglasnost obvezna ili neobvezna. Osim toga, postoji varijabla *validFrom*, koja je tipa *Optional Date*. Budući da neka suglasnost može i ne mora biti dana, ima smisla postaviti tip ove varijable kao *Optional*, jer ako suglasnost nije dana, varijabla *validFrom* ne može imati vrijednost, tj. vrijednost navedene varijable bit će *nil*. Također, *CMConsent* ima varijablu *externalLink*, koji sadrži adresu *web* stranice. Pohranjena *web* stranica ima zadaću korisniku pobliže objasniti nužnost dotične suglasnosti. Jedna od bitnijih varijabli klase koja označava suglasnost je *state*, koja je tipa *ConsentState*. *ConsentState* također je model ove značajke, a njegova vrijednost može biti jedan od dva slučaja. Prvi slučaj je *GIVEN*, koji znači da je korisnik u nekom trenutku korištenja aplikacije dao svoju suglasnost za određenu značajku. Drugi slučaj je *WITHHELD*, a ako suglasnost ima ovu vrijednost varijable *state*, moguća su dva slučaja, da korisnik nikada nije dao suglasnost za određenu značajku, ili da je korisnik dao suglasnost za neku značajku i u određenom trenutku nakon davanja suglasnosti ju je povukao za određenu značajku.

4.3.3. Modul detalja

Modul detalja je modul koji prikazuje informacije o jednoj specifičnoj suglasnosti. Dizajn modula detalja može se vidjeti na slici 4.8. Ovaj dio značajke prikazuje funkciju ili ime značajke na koju se suglasnost odnosi, što je na navedenoj slici značajka upravljanja lijekovima. Može se vidjeti kako je datum davanja suglasnosti pohranjen u standardom europskom formatu, gdje prve dvije znamenke simboliziraju dan, druge dvije mjesec, i zadnje četiri godinu davanja suglasnosti, odvojene točkom. Kao što se na slici može vidjeti u naslovu u navigacijskoj traci, radi se o obveznoj suglasnosti, što znači da značajka upravljanja lijekovima neće moći biti korištena bez navedene suglasnosti. Treća stavka na zaslonu znači identifikator koji je jedinstven za određenu značajku. Odabirom zadnjeg reda popisa moguće je preusmjeriti korisnika na vanjsku poveznicu, koja mu može dati dodatna pojašnjenja zašto i kako se koriste korisnički podaci. Na dnu se nalazi crveni gumb koji simbolizira destruktivnu radnju, a pritiskom na njega korisniku se daje mogućnost povlačenja dane suglasnosti.

U programskom kodu 4.13. nalazi se funkcija odgovorna za prikaz detalja suglasnosti koja se nalazi u *View* komponenti *MVVM* arhitekturnog obrasca. Izlazni proizvod ove funkcije podijeljen je u dva odjeljka. Prvi odjeljak sastoji se od tri elementa tipa *BasicDetailRow*, koji prikazuje glavni tekst i detaljni tekst, a on je svjetlije boje i manje veličine. Prvi takav element prikazuje ime značajke na koju se suglasnost odnosi, a druga dva elementa prikazuju se samo ako je dana suglasnost za određenu značajku. Drugi element prikazuje datum davanja suglasnosti dok treći prikazuje jedinstveni identifikacijski kod unutar iste značajke. Drugi odjeljak prikazuje tekst koji je moguće odabrati, čime se pokreće pretraživač s otvorenom odgovarajućom *web* stranicom. Pretraživač se pokreće prilikom promjene *showingSafari* varijable iz vrijednosti *false* u vrijednost *true*, što nam govori da se radi o varijabli tipa *Bool*. Ugrađenom funkcijom *toggle* možemo promijeniti vrijednost varijable tipa *Bool* u *true* ako je trenutna vrijednost *false,* i u *false* ako je trenutna vrijednost *true*. Na kraju, pomoću modifikatorskih funkcija *buttonStyle* i *listStyle* postavlja se željeni stil prikaza gumba i popisa. Budući da se u ovom odjeljku ne radi o tipu *BasicDetailRow*, potrebno je ručno postavljanje vrijednosti koja određuje udaljenost elemenata unutar odjeljka, koja je u ovom slučaju jednaka 4 i način poravnanja koji je u ovom slučaju postavljen na poravnanje u odnosu na lijevi rub.

```
@ViewBuilder
private func createList() -> some View {
    List {
        Section {
            BasicDetailRow(
                text: ConsentManagement.function,
                details: viewModel.consent.category
            )

            if viewModel.consent.state.isActive {
                BasicDetailRow(
                    text: ConsentManagement.dateOfApproval,
                    details: viewModel.validFrom
                )

                BasicDetailRow(
                    text: ConsentManagement.consentId,
                    details: viewModel.consent.id.consentCode
                )
            }
        }

        Section {
            VStack(alignment: .leading, spacing: 4) {
                Text(viewModel.consent.name)
                    .textStyle(.listText)

                Button {
                    showingSafari.toggle()
                } label: {
                    Text(ConsentManagement.show)
                }
                .buttonStyle(.secondary(style: .inline))
            }
        }
    }
    .listStyle(.grouped)
}
```

Programski kod 4.13. *Prikaz detalja određene suglasnosti*

Deklarativna definicija destruktivnog gumba nalazi se u programskom kodu 4.14., unutar funkcije *createButton*.

```swift
@ViewBuilder
private func createButton() -> some View {
    if viewModel.consent.state.isActive {
        Button(role: .destructive) {
            self.showingAlert = true
        } label: {
            Text(ConsentManagement.revoke)
        }
        .buttonStyle(.primary)
        .padding()

    } else {
        Button {
            viewModel.giveConsent()
        } label: {
            Text(ConsentManagement.agree)
        }
        .buttonStyle(.primary)
        .padding()
    }
}
```

Programski kod 4.14. *Prikaz definicije destruktivnog gumba*

Ovisno o aktivnosti suglasnosti, tj. ovisno o tome je li korisnik dao suglasnost, prikazuje se odgovarajući gumb. Ako suglasnost nije dana, prikazuje se običan gumb koji omogućuje korisniku davanje suglasnosti, a ako je suglasnost dana, prikazuje se destruktivni gumb, koji daje mogućnost povlačenje suglasnosti. U programskom kodu 4.15. prikazana je definicija skočnog prozora.

```swift
private func renderAlert() -> Alert {
    Alert(
        title: Text(ConsentManagement.alertTitle),
        message: Text(viewModel.alertMessage),
        primaryButton: .cancel(
            Text(Common.cancel)
        ),
        secondaryButton: .destructive(
            Text(ConsentManagement.revoke),
            action: { viewModel.revokeConsent() }
        )
    )
}
```

Programski kod 4.15. *Prikaz definicije skočnog prozora*

U funkciji *renderAlert* prikazana je definicija skočnog prozora koji se prikazuje prilikom pritiska na destruktivni gumb. Izgled skočnog prozora može se vidjeti na slici 4.9. Sastoji se od naslova, poruke i dva gumba. Jedan gumb je standardnog dizajna, koji poništava radnju, a drugi je destruktivan i služi kao potvrda povlačenja suglasnosti.

Programski kod 4.16. prikazuje logiku davanja suglasnosti na strani *ViewModel* komponente. Nad instancom logike poziva se funkcija *activateConsents*, kojoj se predaje jedinstveni identifikator suglasnosti koja se želi aktivirati. Određuje se glavna nit na kojoj se želi primiti rezultat, te u *sink* bloku definira se radnja koja se želi izvršiti prilikom završetka dohvaćanja i radnja koja se želi izvršiti nad primljenim podacima. Završetak dohvaćanja uspoređuje se s neuspjehom, te ako je rezultat istinit, znači da dohvaćanje nije uspjelo i ispisuje se odgovarajuća poruka u konzoli.

```
func giveConsent() {
    logic
        .activateConsents([consent.id])
        .receive(on: DispatchQueue.main)
        .sink(
            receiveCompletion: { completion in
                if case let .failure(error) = completion {
                    CMLogger.error(message: String(describing: error))
                }
            },
            receiveValue: { [weak self] consents in
                guard let self = self else {
                    return
                }
                self.consent = consents.first {
                    $0.id == self.consent.id
                } ?? self.consent
            }
        )
        .store(in: &cancellables)
}
```

Programski kod 4.16. *Prikaz logike davanja suglasnosti*

Budući da dohvaćene suglasnosti sadrže i trenutnu suglasnost s danom suglasnosti, dobivene suglasnosti filtriraju se s obzirom na jedinstveni identifikator, te se filtrirana vrijednost postavlja za vrijednost trenutne suglasnosti unutar *ViewModel*-a.

Na sličan način funkcionira i povlačenje suglasnosti. Nad logikom se poziva funkcija *revokeConsent*, kojoj se prosljeđuje jedinstveni identifikator suglasnosti, kao u programskom kodu 4.17. Na isti način kao i u prošlom primjeru postavlja se nit na kojoj se odvija dohvaćanje podataka, a jedina razlika je u tome što provjerava je li suglasnost obvezna.

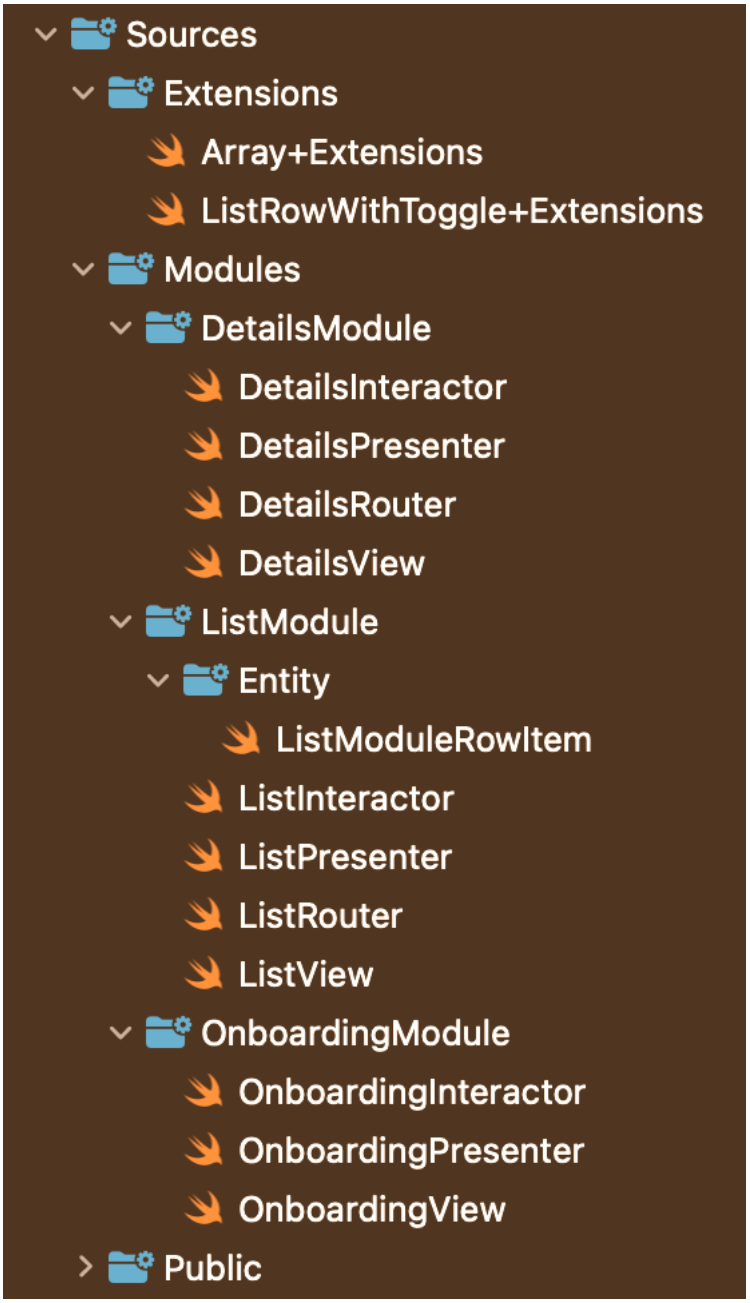
```
func revokeConsent() {
    logic
        .revokeConsent(id: consent.id)
        .receive(on: DispatchQueue.main)
        .sink(
            receiveCompletion: { completion in
                if case let .failure(error) = completion {
                    CMLogger.error(message: String(describing: error))
                }
            },
            receiveValue: { [weak self] consents in
                guard let self = self else {
                    return
                }
                if !self.consent.isOptional {
                    self.dismiss()
                } else {
                    self.consent = consents.first {
                        $0.id == self.consent.id
                    } ?? self.consent
                }
            }
        )
        .store(in: &cancellables)
}
```

Programski kod 4.17. *Prikaz logike ukidanja suglasnosti*

Provjera obveznosti suglasnosti provjerava se zato što korisnik ne bi trebao moći ostati na zaslonu s detaljima suglasnosti nakon povlačenja suglasnosti koja je obvezna. Jedini način da korisnik da suglasnost je da pokuša pristupiti značajki za koju je potrebna suglasnost. Ako suglasnost nije obvezna, korisnik nakon ukidanja iste ostaje na zaslonu, no ako je ukinuta suglasnost obvezna, korisnika se vraća na popis suglasnosti i automatski se ažurira popis danih suglasnosti.

4.4. Implementacija VIPER arhitekturnog obrasca na primjeru značajke za upravljanje suglasnostima u usporedbi s MVVM arhitekturnim obrascem

VIPER arhitekturni obrazac semantički je prilično sličan *MVVM*-u, što se na prvi pogled možda i ne bi moglo reći. Organizacija koda *VIPER* arhitekture može se vidjeti na slici 4.12.



Sl. 4.12. *Prikaz VIPER arhitekture na primjeru značajke upravljanja suglasnostima*

Prva činjenica koja se da opaziti promatranjem navedene slike je da se ovdje nalazi značajno više datoteka nego u *MVVM* arhitekturnom obrascu, što je očekivano budući da se *MVVM* arhitekturni obrazac sastoji od tri komponente, a *VIPER* od pet.

4.4.1. Modul za ukrcavanje

Modul za ukrcavanje u implementaciji *VIPER* arhitekturnog obrasca sastoji se od komponenti *View, Presenter* i *Interactor*. *View* komponenta ima jednake zadaće i odgovornosti kao što je navedeno u poglavlju 4.3.1. *Presenter* u *VIPER* arhitekturnom obrascu preuzima dio odgovornosti *ViewModel*-a iz *MVVM* arhitekturnog obrasca. U programskom kodu 4.18. može se vidjeti primjer dohvaćanja svih suglasnosti.

```
func loadConsents() {
    isLoading = true
    interactor.getConsentsByCode(code: code)
        .sink(receiveCompletion: { [weak self] completion in
            self?.isLoading = false
            if case let .failure(error) = completion {
                CMLogger.error(message: String(describing: error))
            }
        }, receiveValue: { [weak self] consents in
            self?.consents = consents
        })
        .store(in: &cancellables)
}
```

Programski kod 4.18. *Prikaz dohvaćanja suglasnosti u VIPER arhitekturnom obrascu*

U funkciji *loadConsents* može se vidjeti razlika implementacije dohvaćanja suglasnosti u *VIPER* i *MVVM* arhitekturim obrascima. Umjesto pozivanja funkcije *getConsentsByCode* direktno nad instancom logike, ovdje se za to koristi *Interactor* komponenta arhitekturnog obrasca *VIPER*. Prilikom završetka dohvaćanja, varijabli koja predstavlja učitavanje podataka daje se vrijednost *false*, u slučaju pogreške ista se ispisuje u konzolu kako bi razvojni inženjer znao gdje i zašto je došlo do pogreške. Dohvaćene suglasnosti postavljaju se za trenutnu vrijednost suglasnosti unutar *Presenter*-a, što se može shvatiti kao ažuriranje podataka. Cijela navedena rutina pohranjuje se u obliku *Cancellable* tipa koji znači određenu vrstu promatranja.

Sličan ovom je i način pohranjivanja suglasnosti, koji se vidi u programskom kodu 4.19. Umjesto nad instancom logike, funkcija *activateConsents* odvija se na instancom komponente *Interactor*. Prilikom završetka pohranjivanja suglasnosti, provjerava se je li došlo do pogreške, te se ista ispisuje u konzolu.

```
func storeConsents() {
    interactor.activateConsents(consents)
        .sink(receiveCompletion: { completion in
            if case let .failure(error) = completion {
                CMLogger.error(message: String(describing: error))
            }
        }, receiveValue: { _ in })
        .store(in: &cancellables)
}
```

Programski kod 4.19. *Prikaz pohranjivanja suglasnosti u VIPER arhitekturnom obrascu*

Funkcije unutar komponente *Interactor* mogu se vidjeti u programskom kodu 4.20. Pozivaju se nad instancom logike koja više nije sadržana u *ViewModel*-u nego u komponenti *Interactor*. Kao što se može vidjeti u funkciji *getConsentByCode*, vraća se samo vrijednost iz instance logike, za razliku u funkciji *activateConsents*, u kojoj se osim interakcije s logikom događa i filtriranje suglasnosti s obzirom na varijablu *isActive*, što znači da se pohranjuju suglasnosti koje su promijenjene u komponenti *Presenter*.

```
func getConsentsByCode(
    code: Int
) -> AnyPublisher<[CMConsent], CMError> {
    logic.getConsentsByCode(code: code)
}

func activateConsents(
    _ consents: [CMConsent]
) -> AnyPublisher<[CMConsent], CMError> {
    let ids = consents
        .filter(\.state.isActive)
        .map(\.id)

    return logic.activateConsents(ids)
}
```

Programski kod 4.20. *Prikaz pohranjivanja suglasnosti u komponenti Interactor*

4.4.2. Modul popisa

Slično kao i u prethodnom modulu, komponenta *Presenter* ne poziva funkciju izravno nad instancom logike nego nad instancom komponente *Interactor*. Zadaća modula popisa je prikazivanje suglasnosti koje je korisnik već pristao dati prilikom pokušaja ulaska u značajku za koju je potrebno dati suglasnost. Kako bi se suglasnosti prikazale, potrebno ih je dohvatiti iz baze podataka, što se može vidjeti u programskom kodu 4.21.

```
func getConsentsForActiveFeatures() {
    isLoading = true
    interactor.getConsentsForActiveFeatures()
        .sink(receiveCompletion: { [weak self] completion in
            self?.isLoading = false
            if case let .failure(error) = completion {
                CMLogger.error(message: String(describing: error))
            }
        }, receiveValue: { [weak self] consents in
            self?.manageConsentsItems = consents.groupConsentsByCategory()
        })
        .store(in: &cancellables)
}
```

Programski kod 4.21. *Prikaz dohvaćanja suglasnosti u komponenti Presenter*

Modul za ukrcavanje koristi se pozivanjem funkcije *getConsentsByCode* za dohvaćanje trenutno potrebne suglasnosti za koju se odvija ukrcavanje, a modul popisa koristi se funkcijom *getConsentsForActiveFeatures* budući da je potrebno dohvaćanje svih suglasnosti koje su trenutno omogućene. Slično kao i u prethodnim interakcijama s komponentom *Interactor*, prilikom završetka dohvaćanja provjerava se je li došlo do pogreške, a dohvaćene suglasnost grupiraju se po kategorijama i sortiraju se tako da se prva pojavljuje suglasnost koja je obvezna, a zatim se pojavljuje neobvezna suglasnost. Ovaj poredak pohranjuje se pomoću klase *ListModuleRowItem* koja simbolizira komponentu *Entity* arhitekturnog obrasca *VIPER*. Komponenta *Entity* arhitekturnog obrasca *VIPER* ekvivalentna je komponenti *Model* arhitekturnog obrasca *MVVM*. Jedna od glavnih razlika između dva navedena arhitekturna obrasca je pojava komponente *Router* sadržane unutar komponente *Presenter*. Odgovornost komponente *Router* je prikazivanje drugog zaslona ili takozvana navigacija korisnika kroz aplikaciju. U programskom kodu 4.22. može se vidjeti definicija komponente *Router* modula popisa.

```
class ListRouter {
    func makeDetailsView(
        selectedConsent: CMConsent,
        logic: ConsentManagementLogic,
        dismiss: @escaping () -> Void
    ) -> some View {
        DetailsView(
            presenter: DetailsPresenter(
                selectedConsent: selectedConsent,
                interactor: DetailsInteractor(
                    logic: logic
                ),
                dismiss: dismiss
            )
        )
    }
}
```

Programski kod 4.22. *Prikaz definicije komponente Router modula popisa*

Komponenta *Router* obično se imenuje prema modulu u kojem se nalazi, tako je i dotični *Router* imenovan *ListRouter*. Klasa se sastoji od jedne funkcije koja treba voditi na modul detalja pa se zbog toga zove *makeDetailsView*, jer se obično funkcije komponente *Router* nazivaju prema destinaciji kojoj vode uz prefiks *make, show* ili *render*, što je u ovom kontekstu semantički jednako. Funkcija prima tri argumenta. Budući da funkcija vodi na zaslon s detaljima, potrebno je definirati detalje o kojoj suglasnosti se radi, tako da je prvi argument specifična suglasnost. Drugi argument je instanca logike koja će se proslijediti komponenti *Interactor* u novonastalom modulu, a treći argument je radnja koja se odvija prilikom povlačenja suglasnosti.

4.4.3. Modul detalja

Modul detalja koji implementira arhitekturni obrazac *VIPER* razlikuje se od implementacije arhitekturnog obrasca *MVVM* u jednakoj mjeri kao i implementacije modula popisa. Najbitnija stavka koju treba spomenuti je komponenta *Router*, imenovana *DetailsRouter*, budući da se nalazi unutar modula detalja. Definicija *DetailsRouter*-a može se vidjeti u programskom kodu 4.23.

```
class DetailsRouter {
    func makeSafariView(
        url: URL
    ) -> some View {
        SafariView(url: url)
    }
}
```

Programski kod 4.23. *Prikaz definicije komponente Router modula detalja*

Ova klasa sadrži jednu funkciju čija je odgovornost prikazati *SafariView*, odnosno pokrenuti pretraživač s otvorenom željenom *web* stranicom. Iz tog razloga potrebno je funkciji proslijediti argument tipa *URL*, kako bi *SafariView* bio u stanju prikazati željenu *web* stranicu. U programskom kodu 4.24. može se vidjeti primjer korištenja funkcije komponente *Router*.

```
@ViewBuilder
func safariBuilder() -> some View {
    if let url = url {
        router.makeSafariView(url: url)
    }
}
```

Programski kod 4.24. *Prikaz korištenja funkcije komponente Router*

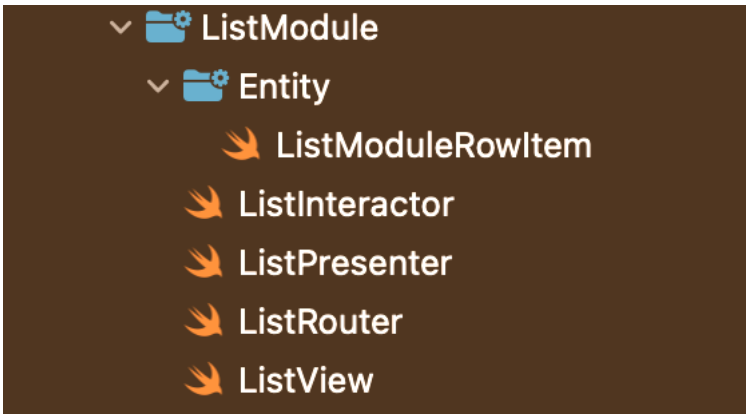
Funkcija *safariBuilder* unutar komponente *Presenter* prikazuje jedan način korištenja komponente *Router*. Pomoću izraza *if let* moguće je osigurati postojanje vrijednosti varijable *url*, čime izoliramo jedan faktor koji može utjecati na pogrešku prilikom učitavanja *web* stranice. Unutar tijela *if let* izraza nad instancom komponente *Router* poziva se funkcija *makeSafariView* te se kao argument predaje već provjerena (eng. *safely unwrapped*) vrijednost varijable *url*. Ako ne dođe do nikakvih ostalih pogrešaka, korisnik bi prilikom poziva ove funkcije trebao vidjeti prikazan polazni pretraživač operacijskog sustava *iOS, Safari*, koji otvara poveznicu koja je proslijeđena funkciji unutar varijable *url.*

5. PRIJELAZ S ARHITEKTURNOG OBRASCA VIPER NA MVVM NA PRIMJERU ZNAČAJKE UPRAVLJANJA SUGLASNOSTIMA

U ovom poglavlju bit će predstavljen način implementacije arhitekturnog obrasca *MVVM* u slučaju da je programskim kodom već implementiran arhitekturni obrazac *VIPER*. Način prelaska s arhitekturnog obrasca *VIPER* na *MVVM* bit će prikazan u koracima, na primjeru modula popisa značajke za upravljanje suglasnostima, budući da sadrži sve komponente arhitekturnog obrasca *VIPER*: *View*, *Interactor*, *Presenter*, *Entity* i *Router*, za razliku od ostalih navedenih modula. U nastavku slijede koraci potrebni za migraciju s arhitekturnog obrasca *VIPER* na arhitekturni obrazac *MVVM*.

5.1. Analiza postojećih komponenata arhitekturnog obrasca VIPER

U prvom koraku potrebno je analizirati postojeće datoteke u arhitekturnom obrascu *VIPER*, što se može napraviti uz pomoć slike 5.1. Na navedenoj slici jasno se može vidjeti svaka komponenta arhitekturnog obrasca *VIPER*: *View*, *Presenter*, *Router*, *Interactor* i *Entity*, uz odgovarajući prefiks modula. Unutar ovog koraka također je potrebno uočiti međuovisnosti navedenih komponenata. Iako se obično prati obrazac u kojem komponenta *View* sadrži komponente *Presenter* i *Router*, *Presenter* sadrži komponentu *Interactor*, a *Interactor* raspolaže komponentom *Entity*, nerijetko se koriste modifikacije koje su potrebne za specifičnosti nekog projekta.



Sl. 5.1 *Prikaz arhitekturnog obrasca VIPER na primjeru modula popisa*

5.2. Uvođenje komponente ViewModel

Komponenta *ViewModel* srž je arhitekturnog obrasca *MVVM*. Budući da arhitekturni obrazac *VIPER* ne sadrži komponentu *ViewModel*, potrebno je uvesti izmjene koje će omogućiti uvođenje iste. Budući da je u ovom slučaju udaljeni izvor podataka simboliziran instancom logike, sadržaj komponente *Interactor*, koji je vidljiv u programskom kodu 5.1., potrebno je refaktorirati tako da se pripoji komponenti *Presenter*, što će činiti komponentu *ViewModel* arhitekturnog obrasca *MVVM*.

```
class ListInteractor {
    let logic: ConsentManagementLogic

    init(
        logic: ConsentManagementLogic
    ) {
        self.logic = logic
    }

    func getConsentsForActiveFeatures(
    ) -> AnyPublisher<[CMConsent], CMError> {
        logic.getConsentsForActiveFeatures()
    }
}
```

Programski kod 5.1. *Prikaz komponente Interactor VIPER arhitekturnog obrasca*

Programski kod 5.2. prikazuje funkciju u kojoj se vidi kombinacija komponente *Interactor* i komponente *Presenter*. Može se vidjeti dohvaćanje podataka iz instance logike, te pohranjivanje prikazanog tijeka događaja unutar komponente *ViewModel*. Svojstva komponente *ViewModel* mogu se vidjeti u programskom kodu 5.3. Destinacija navedenog pohranjivanja tijeka događaja je varijable *cancellables*, koja predstavlja skup promatranja *ViewModel*-a nad komponentom *Model*. U slučaju ovog modula komponenta *Model* ne zahtijeva refaktoriranje, već se može koristiti i u arhitekturnom obrascu *MVVM*. Osim varijable za pohranu promatranja, postoje i *Published* varijable koje služe za vezanje podataka (eng. *data binding*) s komponentom *View*.

```
private func getConsentsForActiveFeatures() {
    isLoading = true
    logic
        .getConsentsForActiveFeatures()
        .receive(on: DispatchQueue.main)
        .sink(
            receiveCompletion: { [weak self] completion in
                self?.isLoading = false
                if case let .failure(error) = completion {
                    CMLogger.error(message: String(describing: error))
                }
            },
            receiveValue: { [weak self] consents in
                self?.manageConsentsItems = consents.groupConsentsByCategory()
            }
        )
        .store(in: &cancellables)
}
```

Programski kod 5.2. *Prikaz funkcije koja kombinira funkcionalnosti komponente Interactor i Presenter*

```
private var cancellables: Set<AnyCancellable> = .init()
private let logic: ConsentManagementLogic

@Published var consent: CMConsent?
@Published var manageConsentsItems: [ListModuleRowItem]?
@Published var isLoading = true
```

Programski kod 5.3. *Prikaz svojstava komponente ViewModel arhitekturnog obrasca MVVM*

5.3. Refaktoriranje komponente View

Komponenta *View* zahtijeva refaktoriranje zato što u arhitekturnom obrascu *VIPER* navedena komponenta ima jedinu zadaću prikaz već pripremljenih podataka iz komponente *Presenter*. U arhitekturnom obrascu *MVVM*, komponenta *View* ponaša se nešto drugačije. Budući da je za razvoj značajke upravljanja suglasnostima korišten radni okvir *SwiftUI*, navigacija će se također odvijati u *View* komponenti. U slučaju da se ne koristi navedeni radni okvir, sličan učinak može se postići korištenjem strukturnog oblikovnog obrasca *koordinator*. Smisao refaktoriranja komponente V*iew,* između ostalog je spajanje komponenata *View* i *Router* u jednu komponentu. U programskim kodovima 5.4. i 5.5. može se vidjeti definicija komponente *Router* i njezina funkcionalnost. Smisao navedene komponente i jedina odgovornost je prikaz druge određene *View*

komponente. U programskom kodu 5.6. prikazan je način pripreme komponente *Router* za korištenje od strane komponente *View*.

```
class ListRouter {
    func makeDetailsView(
        selectedConsent: CMConsent,
        logic: ConsentManagementLogic,
        dismiss: @escaping () -> Void
    ) -> some View {
        DetailsView(
            presenter: DetailsPresenter(
                selectedConsent: selectedConsent,
                interactor: DetailsInteractor(
                    logic: logic
                ),
                dismiss: dismiss
            )
        )
    }
}
```

Programski kod 5.4. *Prikaz komponente Router i njene funkcionalnosti*

```
destination: {
    presenter.detailsBuilder(for: item, dismiss: { consent = nil })
        .onDisappear {
            consent = nil
            presenter.onAppear()
        }
},
```

Programski kod 5.5. *Prikaz korištenja komponente Router u komponenti View*

```
func detailsBuilder(
    for item: CMConsent,
    dismiss: @escaping () -> Void
) -> some View {
    router.makeDetailsView(
        selectedConsent: item,
        logic: interactor.logic,
        dismiss: dismiss
    )
}
```

Programski kod 5.6. *Prikaz korištenja komponente Router u komponenti Presenter*

Arhitekturni obrazac *MVVM* ovo može pojednostaviti, posebice uz korištenje *SwiftUI* radnog okvira, kao što je prikazano u programskim kodovima 5.7. i 5.8.

```
var body: some View {
    NavigationView {
        ZStack {
            createContent()
        }
        .navigationTitle(ConsentManagement.manageConsents)
        .navigationBarTitleDisplayMode(.inline)
        .onAppear {
            viewModel.onAppear()
        }
        .toolbar {
            ToolbarButton(.text(Common.done)) {
                dismiss()
            }
        }
    }
}
```

Programski kod 5.7. *Prikaz navigacijskog pogleda*

```
NavigationWrapper(
    content: {
        Button(
            action: {
                viewModel.consent = item
            },
            label: {
                BasicDetailRow(
                    text: item.name,
                    details: viewModel.getValidFromString(for: item)
                )
            }
        )
    },
    destination: {
        DetailsView(
            viewmodel: DetailsViewModel(
                selectedConsent: item,
                logic: viewModel.logic,
                dismiss: { viewModel.consent = nil }
            )
        )
        .onDisappear {
            viewModel.consent = nil
            viewModel.onAppear()
        }
    },
    showView: .constant(viewModel.consent?.id == item.id)
)
```

Programski kod 5.8. *Prikaz navigacijskog omotača*

Umjesto korištenja funkcije komponente *Router*, koristi se ugrađeni element radnog okvira *SwiftUI Navigation View*, uz koji se jednostavno može definirati navigacija. Navigacija se obično definira pomoću drugog elementa *NavigationLink* ili navigacijska veza, čija je odgovornost povezivanje trenutnog zaslona s idućim kada korisnik vrši interakciju s navigacijskom vezom. U ovom slučaju koristi se navigacijski omotač koji prima tri argumenta. Prvi je sadržaj elementa koji je prikazan na prvom zaslonu i za koji treba omogućiti interakciju s korisnikom prilikom dodira. Drugi argument je destinacija, što znači zaslon koji se korisniku treba prikazati prilikom interakcije s navigacijskom vezom. Zadnji argument koji se predaje je izraz ili varijabla koja je tipa *Bool*. U slučaju da je izraz ili varijabla navedenog tipa jednaka *true*, prikazat će se zaslon definiran argumentom *destination*.

Zadnji dio refaktoriranja komponente *View* je vezanje podataka. U programskom kodu 5.9. nalazi se jednostavan primjer vezanja podataka sadržan u *View* komponenti. Funkcija *createContent* nalazi se unutar *body* varijable, tako da se poziva prilikom prikazivanja dotične komponente *View*. Pozivanjem ove funkcije provjerava se postojanje instance komponente *Model* unutar komponente *ViewModel*, ako podaci ne postoje, prikazuje se zaslon praznog stanja, ako podaci postoje, prikazuje se element *SwiftUI* radnog okvira *List*, koji prikazuje modele postojeće u *ViewModel*-u.

```
@ViewBuilder
private func createContent() -> some View {
    if let items = viewModel.manageConsentsItems {
        if items.isEmpty {
            EmptyScreenLayout(
                title: ConsentManagement.emptyListTitle,
                text: ConsentManagement.emptyListText
            )
        } else {
            List(items) { section in
```

Programski kod 5.9. *Prikaz vezanja podataka u komponenti View*

Ovim deklarativnim načinom rada funkcija *createContent* poziva se svaki put promjenom vrijednosti varijable *manageConsentItems*, jer je označena oznakom *Published* unutar komponente *ViewModel*. Ovim jednostavnim primjerom pokazano je kako deklarativni način definiranja korisničkog sučelja i vezanje podataka funkcioniraju unutar arhitekturnog obrasca *MVVM*.

6. ZAKLJUČAK

Arhitekturni obrasci su obrasci koji omogućuju razvojnim inženjerima razvoj skalabilnih, modularnih i održivih aplikacija. Arhitekturni obrazac *MVVM* prikazuje da bi, osim komponenata *View* i *Model*, trebao postojati sloj između – komponenta *ViewModel*, ali ne govori kako se komponenta *ViewModel* stvara, niti kako se podaci dohvaćaju - nisu sve odgovornosti jasno definirane. Otvoren je i može se implementirati na mnogo različitih načina. S druge strane, arhitekturni obrazac *VIPER* vrlo je specifična softverska arhitektura. Sadrži slojeve s vlastitim odgovornostima i manje je otvorena za promjene. Kad je riječ o odabiru jednog ili drugog, nema najboljeg univerzalnog rješenja. Ako ste u dugoročnom projektu s dobro definiranim zahtjevima i namjeravate ponovno koristiti komponente, arhitekturni obrazac *VIPER* definitivno je bolja opcija. Jasnije odvajanje odgovornosti poboljšava mogućnost testiranja i ponovne upotrebe. Ako izrađujete prototipove ili se radi o kraćem projektu bez potrebe za ponovnim korištenjem komponenti, arhitekturni obrazac *MVVM* bolje odgovara. S arhitekturnim obrascem *VIPER* moguće je pojavljivanje potrebe za stvaranjem puno klasa i protokola za male odgovornosti. Arhitekturni obrazac *MVVM* općenito proizvodi puno manje koda zbog ne tako jasnog odvajanja odgovornosti i može izbjeći neke dodatne troškove koje bi stvorio arhitekturni obrazac *VIPER*. Kod bi svakako bio jednostavniji što se tiče implementacije, a i dalje bi bio jednostavan za testiranje i održavanje.

LITERATURA

[1] The Ultimate Guide to Mobile Application Architecture [online], Scand, 2022, dostupno na: https://scand.com/company/blog/the-ultimate-guide-to-mobile-application-architecture/. [23. lipanj 2023].

[2] R. Ranjan, The Mobile App Architecture Guide for 2023 [online], Net Solutions, 2022, dostupno na: https://www.netsolutions.com/insights/mobile-app-architecture-guide/. [23. lipanj 2023].

[3] R. Cacheaux, Which Architecture is Right for Me? [online], Kodeco, 2023, dostupno na: https://www.kodeco.com/books/advanced-ios-app-architecture/v3.0/chapters/2-which-architecture-is-right-for-me. [5. srpanj 2023].

[4] F. Laso-Marsetti, Model-View-Controller (MVC) in iOS – A Modern Approach [online], 2019 dostupno na: https://www.kodeco.com/1000705-model-view-controller-mvc-in-ios-a-modern-approach. [23. lipanj 2023].

[5] D. Indrawan, D. S. Kusumo i S. Y. Puspitasari, Analysis of the implementation of MVVM architecture pattern on performance of iOS mobilebased applications, JIPI (Jurnal Ilmiah Penelitian dan Pembelajaran Informatika), svez. 8, br. 1, pp. 59-65, 2022.

[6] K. W. Tracy, Mobile Application Development Experiences on Apple's iOS and Android OS, IEEE Potentials, svez. 31, br. 4, pp. 30-34, 2012.

[7] F. Nayebi, J.-M. Desharnais i A. Abran, An Expert-based Framework for Evaluating iOS Application Usability, u Joint Conference of the 23rd International Workshop on Software Measurement, Ankara, 2013.

[8] TIOBE Index for June 2023 [online], TIOBE, dostupno na: https://www.tiobe.com/tiobe-index/. [24. lipanj 2023].

[9] E. K. Ekren, What Is Xcode and How to Use It?, Netguru [online], 2022, dostupno na: https://www.netguru.com/blog/what-is-xcode-and-how-to-use-it. [24. lipanj 2023].

[10] C. Tozzi, A Beginner's Guide to iOS App Testing [online], SweetCode, 2022. dostupno na: https://sweetcode.io/a-beginners-guide-to-ios-app-testing/. [24. lipanj 2023].

[11] W. Kelton, Apple App Store [online], Investopedia, 2023, dostupno na: https://www.investopedia.com/terms/a/apple-app-store.asp. [24. lipanj 2023.].

[12] P. Hudson, What is SwiftUI? [online], Hacking with Swift, 2023, dostupno na: https://www.hackingwithswift.com/quick-start/swiftui/what-is-swiftui. [24. lipanj 2023].

[13] J. Varma, u SwiftUI for Absolute Beginners: Program Controls and Views for iPhone, iPad, and Mac Apps, Apress, 2019, pp. 1-6.

[14] A. Rai, Combine Framework in Swift [online], Medium, 2020, dostupno na: ://medium.com/@anuj.rai2489/combine-framework-in-swift-b730ccde131. [24. lipanj 2023].

[15] D. Yuristiawan, Reactive way of Mobile App Development using iOS Combine Framework [online], Halodoc, 2022, dostupno na: https://blogs.halodoc.io/reactive-way-of-mobile-app-development-using-ios-combine-framework/#:~:text=Combine%20is%20a%20powerful%20framework,easy%20to%20read%20and%20maintain.. [24. lipanj 2023].

SAŽETAK

Arhitekturni obrasci temelj su razvoja skalabilnih, nadogradivih i lako održivih aplikacija. Cilj ovog diplomskog rada bio je usporediti dva popularna arhitekturna obrasca, istaknuti njihove prednosti i prikazati prijelaz s jednog arhitekturnog obrasca na drugi. U početku rada prikazane su četiri najpopularnije arhitekture koje su trenutno korištene u razvoju *iOS* aplikacija, a potkrijepljene su dijagramima i navođenjem prednosti i specifičnosti. Zatim, objašnjeni su alati, programski jezik, integrirano razvojno okruženje, operacijski sustav *iOS*, industrijski standard smjernica za ljudsko sučelje, metodologija testiranja *iOS* aplikacija i distribucijska platforma. Nakon toga, demonstrirana je implementacija trenutno najpopularnijeg arhitekturnog obrasca *MVVM* na primjeru značajke upravljanja suglasnostima aplikacije za elektroničke kartone pacijenata, nakon čega je ista uspoređena s implementacijom arhitekturnog obrasca *VIPER*. Na kraju rada nalazi se popis koraka uz objašnjenja kako prijeći s arhitekturnog obrasca *VIPER* na arhitekturni obrazac *MVVM*.

Ključne riječi: arhitekturni obrazac, *iOS*, migracija, *MVVM*, *VIPER*

ABSTRACT

Comparison between MVVM and VIPER architectures in iOS application development

Architectural patterns are foundation of development of scalable, upgradable, and easily maintainable applications. The goal of this master's thesis was to compare two popular architectural patterns, point out their advantages and show migration from one architectural pattern to the other. In the beginning of this paper, four most popular architectural patterns in *iOS* development are shown, joined by diagrams, and stating advantages and characteristics. Furthermore, tools, programming language, integrated development environment, *iOS* operating system, industrial standard of human interface guidelines, *iOS* application testing methodology and distribution platform are explained. Then, implementation of the currently most popular architectural pattern *MVVM* is demonstrated using example of consent management feature of electronic patient records application, after which it is compared to implementation of *VIPER* architectural pattern. Finally, there is a list of steps accompanied with elaborations on how to migrate from *VIPER* architectural pattern to *MVVM* architectural pattern.

Key words: architectural pattern, *iOS*, migration, *MVVM*, *VIPER*

ŽIVOTOPIS

Domagoj Bunoza rođen je 7. veljače 2000. godine u Našicama, Hrvatska, a živi u Đakovu. Nakon završetka Osnovne škole Josipa Antuna Ćolnića u Đakovu, upisuje opći smjer Gimnazije Antuna Gustava Matoša. Gimnaziju završava 2018. godine i upisuje sveučilišni preddiplomski studij računarstva na Fakultetu elektrotehnike, računarstva i informacijskih tehnologija u Osijeku. Završetkom preddiplomskog studija, na istom fakultetu upisuje diplomski studij programskog inženjerstva. Od kvaliteta ističe znanje engleskog jezika, sposobnost rada pod pritiskom, upornost i odgovornost. Posjeduje znanje programskih alata, odnosno tehnologija: iOS, Swift, Python, C, Javu, C++, C# i Microsoft Office paketa.