Izrada i testiranje mobilne aplikacije za pregled preostalog broja pacijenata u redomatskom redu

Šarić, Valentin

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

Sveučilišni studij

Izrada i testiranje mobilne aplikacije za pregled preostalog broja pacijenata u redomatskom redu

Diplomski rad

Valentin Šarić

Osijek, 2019.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Obrazac D1: Obrazac za imenovanje Povjerenstva za obranu diplomskog rada

Osijek, 24.09.2019.

Odboru za završne i diplomske ispite

Imenovanje Povjerenstva za obranu diplomskog rada

Ime i prezime studenta:	Valentin Šarić
Studij, smjer:	Diplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	D-942R, 19.09.2018.
OIB studenta:	71982821286
Mentor:	Prof.dr.sc. Goran Martinović
Sumentor:	
Sumentor iz tvrtke:	Matko Čeme, mag.ing.comp.
Predsjednik Povjerenstva:	Izv. prof. dr. sc. Krešimir Nenadić
Član Povjerenstva:	Izv. prof. dr. sc. Alfonzo Baumgartner
Naslov diplomskog rada:	Izrada i testiranje mobilne aplikacije za pregled preostalog broja pacijenata u redomatskom redu
Znanstvena grana rada:	**Programsko inženjerstvo (zn. polje računarstvo)**
Zadatak diplomskog rada:	U radu treba napraviti modernu mobilnu aplikaciju za pregled broja pacijenata u redomatskom redu u zdravstvenoj ustanovi gdje već postoji redomat tvrtke IN2. Redni broj bi se uzimao na redomatu, a pacijent nakon što dobije redomatski listić koji na sebi ima broj i barkod treba ga moći unijeti u aplikaciju ručnim unosom znakova s listića koji označavaju redomatski broj ili skeniranjem barkoda s redomatskog listića. Nakon unosa redomatskog broja korisniku, u aplikaciji se treba prikazati trenutni redomatski broj koji je na redu kod prijema, odnosno koliko još redomatskih brojeva ima ispred njega, te mu poslati odgovarajuću obavijest kad se približi njegov redomatski broj. Također, u radu je potrebno opisati testne alate prikladne za testiranje mobilne aplikacije, te napraviti testni plan.
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: 2 bod/boda Razina samostalnosti: 3 razina
Datum prijedloga ocjene mentora:	24.09.2019.

Potpis mentora za predaju konačne verzije rada u Studentsku službu pri završetku studija:	Potpis:
	Datum:

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

IZJAVA O ORIGINALNOSTI RADA

Osijek, 28.10.2019.

Ime i prezime studenta:	Valentin Šarić
Studij:	Diplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	D-942R, 19.09.2018.
Ephorus podudaranje [%]:	5

Ovom izjavom izjavljujem da je rad pod nazivom**: Izrada i testiranje mobilne aplikacije za pregled preostalog broja pacijenata u redomatskom redu**

izrađen pod vodstvom mentora Prof.dr.sc. Goran Martinović

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

Problemi redova i čekanja temelje se na određivanju optimalnog broja mjesta za usluživanje, kako bi se smanjilo vrijeme i troškovi čekanja. Ako je u nekom redu broj mjesta za usluživanje korisnika premali, troškovi će biti niski, ali će vrijeme čekanja biti predugo. Ako je pak broj mjesta za usluživanje prevelik, vrijeme čekanja bit će malo, ali će se pojaviti gubitci zbog neiskorištenosti mjesta za usluživanje. Kada su u pitanju bolnice, predugo čekanje može dovesti do katastrofalnih posljedica. Kako bi se to izbjeglo, potrebno je broj liječnika prilagoditi broju pacijenata. Jedno od rješenja ovoga problema je uvođenje redomata u hodnike bolnica i formiranje redova pomoću automatiziranog sustava.

Cilj ovog rada je izrada i testiranje mobilne aplikacije za pregled preostalog broja pacijenata u redomatskom redu. Rad je nastao iz suradnje Fakulteta elektrotehnike i računarstva u Osijeku i tvrtke IN2. Aplikacija je dio složenijeg sustava i služi za prikaz stanja u redovima bolnice. Korisnik ima mogućnost odabira bolnice ručno ili skeniranjem QR koda kako bi dobio uvid u trenutno stanje u redovima te bolnice. Također, ima mogućnost unosa broja s listića redomata te tako dobiva uvid u trenutnu poziciju broja u redu. Aplikacija je pisana u programskom jeziku Swift za iOS operacijski sustav, a arhitektura aplikacije ostvarena je po uzoru na *CLEAN* arhitekturu. Uzorak dizajna za korisničko sučelje je *MVVM* pri čemu su se poštovali *SOLID* principi razvoja programa. Uporabom reaktivnog načina programiranja riješeni su problemi kontinuiranog dohvaćanja podataka kako bi u svakom trenutku bilo prikazano stvarno stanje u redovima. Aplikacija je razvijena razvojem programske podrške vođenog testiranjem pa je programski kod i svaka funkcionalnost aplikacije detaljno testirana. Nakon izrade aplikacije testirane su performanse i provedeno je ručno testiranje prema prethodno napisanom testnom scenariju.

U poglavlju 2 opisani su problemi raspoređivanja, redova i čekanja kao i neka rješenja ovih problema. Poglavlje 3 opisuje zahtjeve koje aplikacija mora ispuniti i potrebnu opremu i tehnologije za ostvarenje tih zahtjeva. Načini testiranja programske podrške opisani su u poglavlju 4. U poglavlju 5 detaljno je opisan proces razvoja programske podrške vođen testiranjem. Poglavlje 6 sastoji se od opisa programskog rješenja mobilne aplikacije, korištenih tehnologija i detaljnog opisa arhitekture aplikacije. U 7. poglavlju prikazani su primjeri nekih od testova napisanih za vrijeme izrade aplikacije, dok su u poglavlju 8 prikazani rezultati testiranja i krajnji izgled pojedinih aktivnosti aplikacije.

1.1. Opis zadatka

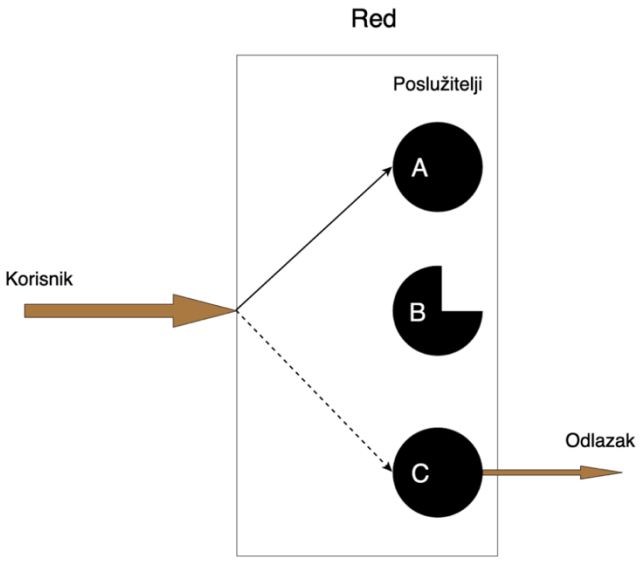
U radu treba napraviti modernu mobilnu aplikaciju za pregled broja pacijenata u redomatskom redu u zdravstvenoj ustanovi gdje već postoji redomat tvrtke IN2. Redni broj bi se uzimao na redomatu, a pacijent nakon što dobije redomatski listić koji na sebi ima broj i barkod treba ga moći unijeti u aplikaciju ručnim unosom znakova s listića koji označavaju redomatski broj ili skeniranjem barkoda s redomatskog listića. Nakon unosa redomatskog broja korisniku, u aplikaciji se treba prikazati trenutni redomatski broj koji je na redu kod prijema, odnosno koliko još redomatskih brojeva ima ispred njega, te mu poslati odgovarajuću obavijest kad se približi njegov redomatski broj. Također, u radu je potrebno opisati testne alate prikladne za testiranje mobilne aplikacije, te napraviti testni plan, scenarij, napisati testove, te prikladno testirati napravljenu aplikaciju. Student nije obvezan raditi na poslužiteljskoj strani pripreme podataka, nego samo pozvati podatke s odgovarajuće adrese.

2. PROBLEMI REDOVA, ČEKANJA I POSLUŽIVANJA

Red se može promatrati kao „crnu kutiju“. Na primjer, red pacijenata u bolnici - ako se promatra jednog pacijenta, on prvo ulazi u red, zatim u redu čeka i određeno vrijeme je potrebno da iz njega izađe. Dakle, red se u tom kontekstu može promatrati kao crna kutija koja ima ulaz i izlaz. Slika 2.1 prikazuje shemu reda kao crne kutije.

Slika 2.1. *Red kao crna kutija*

Ipak, red nije obična „crna kutija“ i poznato je što se u njemu treba odvijati. Postoje različiti algoritmi za upravljanje redovima. Jedan od njih, koji je najčešće spominjan u literaturi [1], je red s tri poslužitelja i jednim korisnikom koji je prikazan na slici 2.2.



Slika 2.2. *Red s tri poslužitelja*

Za prethodni primjer s pacijentom vrijedi sljedeće: ako postoje tri liječnika koji ga mogu primiti, prvi liječnik je slobodan, drugi liječnik je trenutno zauzet pregledom drugog pacijenta, a treći liječnik upravo završava pregled trećega pacijenta, tada će promatrani pacijent ići na pregled kod prvog liječnika, a treći liječnik prihvaća sljedećeg pacijenta koji se pojavi.

2.1. Modeli čekanja i teorija čekanja

Teorija čekanja je matematičko proučavanje linija čekanja ili redova čekanja [1]. Teorija omogućuje matematičku analizu nekoliko povezanih procesa, uključujući dolazak u red, čekanje u redu i posluživanje poslužitelja na prednjem dijelu čekanja [1]. Teorija dopušta izvođenje i izračunavanje nekoliko mjera učinkovitosti, uključujući prosječno vrijeme čekanja u redu čekanja ili sustava, očekivani broj čekanja ili prijema usluge i vjerojatnost susreta sa sustavom u određenim stanjima poput praznog, punog i dostupnog poslužitelj ili poslužitelja za koji je potrebno neko vrijeme da može poslužiti [1]. Najjednostavniji oblik modela redova temelji se na procesu rođenja i smrti, gdje proces rođenja opisuje vrijeme dolazaka korisnika u red, a proces smrti opisuje uslugu ili vrijeme zadržavanja u redu čekanja [2]. Za teoriju čekanja, ako je moguće, bilo bi prikladno raditi s raspodjelom vjerojatnosti koje pokazuju svojstvo pamćenja, jer to uobičajeno pojednostavljuje matematiku koja je uključena. Svojstvo bez pamćenja često se označava kao Markovsko svojstvo, a postupak s njim naziva se Markov proces, što znači da je vjerojatnost raspodjele budućih stanja s obzirom na sadašnje stanje i sva prošla stanja, ovisna samo o sadašnjem stanju, a ne o bilo kojem prethodnom stanju [2]. Radi toga, modeli čekanja često se moderiraju kao Poissonovi procesi pomoću eksponencijalne podjele. Pretpostavimo da je vrijeme međuprostora opisano eksponencijalnom raspodjelom s parametrom λ (intenzitet prometa), a vrijeme zadržavanja opisano eksponencijalnom raspodjelom s parametrom μ. Tada se prolazno ponašanje sustava čekanja izražava:

$$pi'(t) = \lambda\, pi - 1\,(t) + (\lambda + \mu)\, pi\,(t) + \mu\, pi + 1\,(t) \qquad (2\text{-}1)$$

pri čemu je $p_i'(t)$ derivacija za $p_i(t)$, što je vjerojatnost da će *i* u sustavu čekanja postojati u vremenu *t*. Sustav je opisan kao funkcija vremena i može se riješiti kada znamo početnu vrijednost u vremenu 0. Pretpostavimo da sustav postigne statističku ravnotežu. Tada je rješenje neovisno o početnim vrijednostima. Uz to, on ima ravnotežu između dolaska i usluga što podrazumijeva $\lambda / \mu < 1$. Tada je $pi'(t) = 0$.

Uvrštavanjem $pi(t) = pi$, dobivamo:

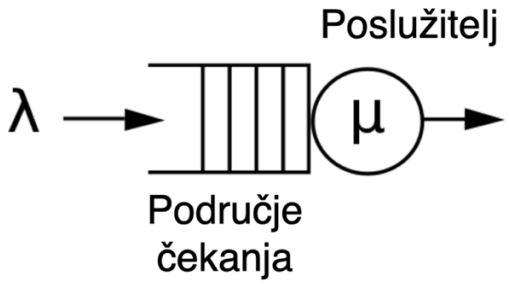
$$(\lambda + \mu)pi = \lambda pi - 1 + \mu pi + 1 \qquad (2\text{-}2)$$

Ovaj sustav čekanja označen je $M/M/1$ i predstavlja eksponencijalno vrijeme dolaska i vrijeme zadržavanja uz jednog poslužitelja. Klasifikacija sustava čekanja slijedi Kendellovu definiciju [1].

2.2. Kendallova notacija

Kandell je predložio opisivanje modela čekanja koristeći tri faktora $A/S/c$ gdje A označava vrijeme između dolaska na red, S distribuciju vremena usluge i c broj poslužitelja na čvoru [1]. Od tada je proširena na $A\ /\ S\ /\ c\ /\ K\ /\ N\ /\ D$ gdje je K kapacitet reda čekanja, N je broj stanovnika koji se trebaju služiti, a D je disciplina u redu čekanja [2], [3]. Kada posljednja tri parametra nisu navedena (npr. $M/M/1$ red), pretpostavlja se $K = \infty, N = \infty\ i\ D\ =\ FIFO$ [4]

U Kendallovoj notaciji: M označava Markova ili pamćenje i znači da se dolasci događaju u skladu s Poissonovim postupkom; D označava determinirane poslove odnosno poslove koji dolaze u red koji zahtijevaju fiksnu količinu usluge; k opisuje broj poslužitelja na čvoru čekanja ($k\ =\ 1, 2, ...$). Ako na čvoru ima više poslova nego na poslužiteljima, onda će se poslovi čekati na uslugu. Slika 2.3 prikazuje Kendallov model opisa jodnoga čvora reda $M/M/1$ [4].



Slika 2.3. *Kendlov model opisa jednoga čvora reda M/M/1*

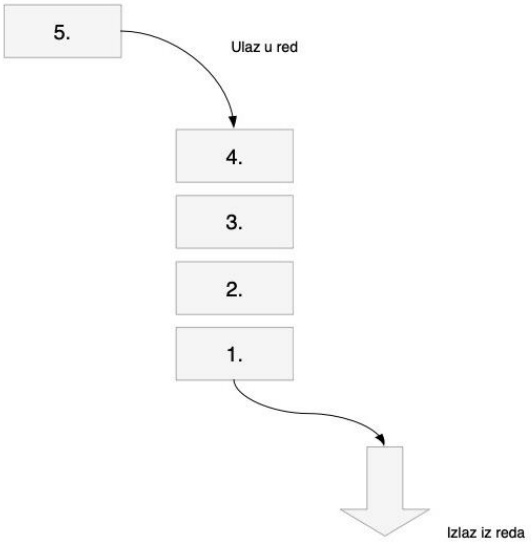
2.3. Načini raspoređivanja

Postoje različiti načini raspoređivanja korisnika u red, a neki od njih su algoritmi FIFO, LIFO, raspoređivanje po prioritetu, SJN.

2.3.1. FIFO način raspoređivanja

FIFO je skraćenica za „prvi ulazi, prvi izlazi“ (engl. *first in, first out*) metodu za organiziranje i manipuliranje među spremnikom podataka, gdje se najprije obrađuje najstariji (prvi) unos ili 'glava' reda čekanja. Analogno je obradi reda čekanja s ponašanjem FCFS algoritma što je skraćenica za „Prvi koji dođe, bit će prvi poslužen“ (engl. *first come, first served*): gdje ljudi napuštaju red u redoslijedu kojim dolaze. FCFS je ujedno i žargonski izraz za algoritam za planiranje operativnog sustava FIFO, koji svakom vremenu procesne centralne jedinice (CPU)

daje redoslijedom kojim se traži. Suprotno FIFO-u je LIFO što predstavlja „Posljednji koji je došao, prvi će izaći“ (engl. *last in, first out*) [5]. Na slici 2.4 prikazan je način rada FIFO algoritma.

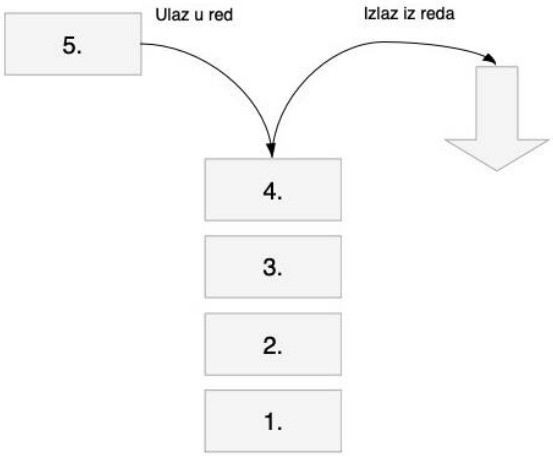


Slika 2.4. *Prikaz algoritma FIFO [5]*

2.3.2. LIFO način raspoređivanja (stog)

LIFO je poznati koncept raspoređivanja zadataka koji se naziva Stog. Redoslijed iz kojeg elementi izlaze iz stoga stvara alternativni naziv LIFO (zadnji ulazi, prvi izlazi) [5] . Rad ovog algoritma može se lako opisati na skupu fizičkih predmeta složenih jedan na drugi. Zadnji predmet koji je složen na skup predmeta, prvi se mora skinuti kako bi došli do sljedećeg predmeta. Stavljanje predmeta na ovakav skup definirano je kao operacija „gurni“ (engl. *push*), a skidanje predmeta sa skupa kao operacija „skini“ (engl. *pop*).

Promatrane kao linearna struktura podataka, operacije guranja i skidanja događaju se samo na jednom kraju strukture, koji se naziva vrhom skupa. To omogućuje implementaciju skupa kao povezan popis s pokazivačem na gornji element. Niz se može implementirati s ograničenim kapacitetom. Ako je skup pun i ne sadrži dovoljno prostora da prihvati element koji se gura, smatra se da je skup u stanju prepunjenosti [5]. Na slici 2.4 prikazan je način rada LIFO algoritma



Slika 2.5. *Prikaz algoritma LIFO[5]*

2.3.3. Raspoređivanje po prioritetu

Raspoređivanje po prioritetu temelji se na raspodijeli korisnika i njihovom usluživanju ovisno o njihovom prioritetu. Korisnici s visokim prioritetom prvi se poslužuju. Postoje dvije vrste redova prioriteta, neprimjenjivi, gdje se posao usluživanja ne može prekinuti jednom kada započne nad jednim korisnikom i preventivni, gdje se usluživanje jednog korisnika može prekinuti ako se pojavi korisnik s većim prioritetom [5]. Implementacija ovakvog algoritma u bolnicama, a pogotovo na hitnim prijemima na automatiziran način bila bi od velikog značaja i jedno od mogućih poboljšanja sustava opisanog u ovome radu upravo je implementacija ovog algoritma [5].

2.3.4. SJF način raspoređivanja

SJF je skraćenica za „prvo najkraći posao“ (engl. *shortest job first*), a predstavlja algoritam raspoređivanja koji zadatke raspoređuje prema njihovom trajanju, tako da se zadatci s najkraćim trajanjem izvođenja prvi izvode [6] . Ovaj algoritam povoljan je zbog svoje jednostavnosti i jer umanjuje prosječno vrijeme koje svaki proces mora čekati dok njegova izvedba ne bude dovršena. Međutim, potencijalno ima problem „gladovanja“ [6] za procese kojima treba dulje vremena da se izvrše, ako se stalno dodaju novi procesi s kratkim vremenom izvođenja. Rješenje ovog problema ostvaruje se tehnikom koja se zove starenje [6], a njenom primjenom vodi se računa o tome koliko pojedini proces čeka na izvođenje.

Sljedeći nedostatak korištenja SJN algoritma je što ukupno vrijeme izvršenja posla mora biti poznato prije izvršenja. Iako je nemoguće točno predvidjeti vrijeme izvršenja, za njegovu procjenu može se upotrijebiti nekoliko metoda, poput ponderiranog prosjeka prethodnih vremena izvršenja [6].

Iako su u aplikaciji izrađenoj u sklopu ovoga rada operacije nad redovima jednostavne, važno je uzeti u obzir problematiku raspoređivanja i teorije čekanja radi smjera u kojemu će se aplikacija kasnije razvijati i potencijalnih unaprjeđenja, a time i problema s kojima bi se kasnije moglo susresti.

3. ZAHTJEVI NA APLIKACIJU I IDEJNO RJEŠENJE

Ova aplikacija zamišljena je kao krajnji proizvod koji će korisnici koristiti kako bi dobili informacije o tome koliko je u određenoj bolnici ljudi u pojedinim redovima. Aplikacija je dio većeg sustava koji se sastoji od Poslužitelja koji je zadužen za raspoređivanje pacijenata u pojedinim bolnicama i REST API-a koji služi kao izvor podataka koji se dohvaćaju i prikazuju unutar aplikacije.

Korisnik aplikacije najprije treba odabrati bolnicu, a nakon toga dobiva uvid u stanje svih redova u bolnici te ima mogućnost pretraživanja redova. Također, korisnik može u aplikaciju unijeti broj i naziv reda s listića dobivenog iz redomata te dobiti uvid u broj pacijenata koji se nalaze ispred njega u redu. Korisnik također ima mogućnost skeniranja QR koda umjesto ručnog odabira bolnice. Kada je QR kod skeniran, u aplikaciju će se automatski unijeti podatci s listića i otvoriti informacije vezane za broj s listića. Ako su unutar QR koda samo podatci vezani za bolnicu, tada se u aplikaciju unosi samo taj podatak i korisnik treba ručno unijeti red i broj ako želi dobiti specifične informacije za taj broj. Podatci unutar aplikacije osvježavaju se svake 3 sekunde, tako da korisnik ima precizan uvid u trenutno stanje u redovima. Aplikacija je usko vezana uz internetsku vezu i radi samo ako je spojena na nju budući da ovisi o REST API-u. Potpuno je dinamična te prikazuje podatke koji se nalaze na API-u dok god je struktura podataka zadovoljena. Time je omogućeno dodavanje novih bolnica i redova od strane poslužitelja u bilo kojem trenutku što će se uspješno manifestirati bez potrebe promjena unutar iOS aplikacije.

Osnovne funkcionalnost aplikacije:

- Uvod korisnika u korištenje aplikacije (samo kod prvog pokretanja)
- Dohvaćanje popisa ponuđenih bolnica i odabir bolnice
- Učitavanje podataka s QR-koda
- Kontinuirano dohvaćanje informacija o redovima unutar odabrane bolnice
- Pretraživanje redova prema korisnikovom unosu
- Kontinuirani Prikaz podataka vezanih za specifični red i broj ako su uneseni
- Prikaz poruka vezanih za iznimke i pogreške
- Točan prikaz na svim trenutno podržanim veličinama zaslona

3.1. Uvod korisnika u korištenje aplikacije

Kod izrade i dizajniranja aplikacije najvažnije je osigurati dobro korisničko iskustvo, no kakvo će ono biti ako korisnik ne razumije što aplikacija radi i što mu omogućava? Zato je potrebno korisnika na neki način uvesti u korištenje aplikacije. Danas je veoma popularno kreiranje uvodnog zaslona (engl. *on boarding screen*) s ciljem da se korisnika u početku nauči kako aplikaciju treba koristiti. Ovaj zaslon najčešće se prikazuje samo prvi puta kada je aplikacija pokrenuta, a nakon toga se više ne prikazuje i obično se sastoji od slika svih zaslona aplikacije uz pojašnjenje ponašanja unutar pojedinog zaslona. U ovoj aplikaciji, uvodni zaslon realiziran je upravo tako, sastoji se od slika zaslona unutar aplikacije, korisnik može klizati prstom kako bi prešao na sljedeću točku uvoda te se na kraju pokreće aplikacija.

3.2. Dohvaćanje popisa ponuđenih bolnica

Budući da je aplikacija namijenjena za korištenje u više bolnica, popis bolnica dohvaća se s REST API-a i korisnik mora odabrati bolnicu kako bi mogao pristupiti informacijama o redovima vezano za istu. Ova funkcionalnost je povezana s dohvaćanjem podataka budući da s obzirom na odabir bolnice šaljemo zahtjev na određenu rutu API-a. Aplikacija je napravljena dinamično pa se tako u bilo kojem trenutku može promijeniti broj bolnica na API-u te će se to manifestirati i u aplikaciji, važno je samo da se poštuje struktura podataka transportnog formata što je u ovom slučaju JSON.

3.3. Učitavanje podataka s QR koda

Ako se korisnik trenutno nalazi u bolnici i uzeo je listić iz redomata, tada ima mogućnost skenirati QR kod koji se nalazi na listiću te tako izbjeći potrebu ručnog odabira bolnice. Bolnice, odnosno redomati u bolnicama, ne moraju imati mogućnost ispisa QR koda na pojedine listiće nego je moguće korisniku olakšati odabir bolnice tako da je na nekom mjestu u bolnici zalijepljen QR kod. Ako redomat ima mogućnost ispisa QR koda na pojedine listiće, tada korisnik ne mora ručno unositi niti ime reda niti broj koji je dobio, jer će se sve ovo automatski unijeti u tom slučaju i korisnik će odmah dobiti uvid u informacije vezane za njegov broj.

3.4. Kontinuirano dohvaćanje informacija o redovima

Budući da se korisnicima pokazuje stanje u redovima, potrebno je kontinuirano dohvaćati podatke s poslužitelja kako bi se dobio točan uvid u stanje redova. Dakle podatci se dohvaćaju svake 3 sekunde, a izazovi koje ovakav način prikaza podataka nosi nisu zanemarivi. Ako dođe do pogreške, izgubi se internetska veza ili se dogode neki problemi s poslužiteljem – korisnika je

potrebno o tome obavijestiti bez čudnih ponašanja aplikacije. Ovo također znači da se podatci dohvaćaju na asinkron način, odnosno ni u kojem trenutku sučelje aplikacije ne smije biti blokirano dok se čekaju podatci s poslužitelja. Kako bi ovu asinkronost bilo lakše realizirati, u aplikaciji se koristi reaktivan način programiranja pomoću reaktivnih ekstenzija (RX) o kojima će više riječi biti kasnije.

3.5. Pretraživanje redova prema korisnikovom unosu

Korisnik ima mogućnost pregleda stanja u redovima bolnice čak i ako se ne nalazi u bolnici. U tom slučaju može vidjeti koliki je red i donijeti odluku o svome odlasku u bolnicu ili ranije javiti o svom dolasku ako se radi o hitnom slučaju, a u redu je gužva. Budući da u bolnici može biti veliki broj redova, korisnik ima mogućnost pretraživanja redova kako bi brzo pronašao red za traženi odjel. Ova mogućnost također donosi određene implementacijske izazove budući da se podatci konstantno osvježavaju, svako dohvaćanje podataka potrebno je filtrirati ako je korisnik unio tekst za pretragu.

3.6. Kontinuirani prikaz podataka vezanih za specifični red i broj

Aplikacija mora kontinuirano prikazivati trenutno stanje u redu za specifičan uneseni broj. Ova funkcionalnost nosi iste izazove kao i prethodna, obje funkcionalnosti dijele isti izvor podataka i razlika je samo u filtriranju i prikazu podataka. Također, potrebno je omogućiti unos podataka koji su dohvaćeni s QR koda kod inicijalnog prikazivanja ovoga zaslona.

3.7. Prikaz poruka o iznimkama i pogreškama

Ako za vrijeme dohvaćanja podataka dođe do nekakve pogreške i podatci se ne osvježe, korisnika je o tome potrebno obavijestiti kako bi znao da trenutno prikazano stanje ne odražava stvarno stanje u redu, također kako bi mogao poduzeti određene akcije ako problem uzrokovan postavkama na njegovom uređaju. Ovo je realizirano pomoću prikaza s porukom na dnu zaslona kada do iznimke dođe. Implementirano je dinamično prikazivanje i micanje prikaza s iznimkama. Također ako korisnik pretražuje unosom imena koje ne postoji u popisu redova za navedenu bolnicu, prikaže se zaslon koji o tome obavještava korisnika. Kao i za sve ostale dinamičke značajke, implementaciju ove specifikacije znatno olakšava rad s reaktivnim ekstenzijama.

3.8. Točan prikaz na svim podržanim inačicama uređaja

Kako bi korisničko iskustvo bilo što bolje, aplikacija mora djelovati jednako na svim veličinama zaslona iOS uređaja koji su trenutno aktualni i u obje orijentacije. Možda se čini kako je ova funkcionalnost uobičajena, no veliki broj popularnih aplikacija ne podržava obje orijentacije i ovo je zapravo važna poslovna odluka budući da može donijeti komplikacije ako se aplikacija bude razvijala i rasla. Dizajn aplikacije je posve jednostavan i čist, fokus je stavljen na funkcionalnost aplikacije i kvalitetu koda i krajnjeg proizvoda.

3.9. Potrebna oprema i alati korišteni za izradu aplikacije

Budući da se radi o prirodnoj iOS aplikaciji, oprema potrebna za izradu sastoji se od računala koje podržava instalaciju alata za izradu prirodnih iOS aplikacija u ovom slučaju to je MacBook Pro (Retina, 13-inch, Early 2015). Alat u kojemu je aplikacija rađena je xCode inačica 10.3 (10G8). Također, zbog testiranja aplikacije i funkcionalnosti skeniranja QR koda, potreban je mobilni uređaj na kojemu se nalazi iOS operacijski sustav i koji podržava skeniranje QR koda. Korišten je iPhone 8. REST API kao izvor podataka realiziran je od strane tvrtke IN2.

Programski jezik u kojemu je aplikacija napisana je Swift, a zbog jednostavnije manipulacije asinkronim događajima aplikacija se jako oslanja na reaktivne ekstenzije odnosno biblioteke rxSwift i rxCocoa. Također kako bi reaktivni kod mogao biti testiran koristi se biblioteka rxTest.

Tehnologije, njihova uloga i implementacija bit će detaljnije objašnjeni kasnije u radu.

4. TESTIRANJE PROGRAMSKE PODRŠKE

Testiranje programske podrške je istraživanje provedeno kako bi se dionicima pružile informacije o kvaliteti proizvoda ili usluge koja se testira [7]. Testiranje programske podrške također može pružiti objektivan, neovisan pogled na proizvod kako bi se poduzeću omogućilo da cijeni i razumije rizike implementacije programske podrške. Tehnike testiranja uključuju proces izvršenja programa ili aplikacije s namjerom pronalaženja programskih grešaka (grešaka ili drugih nedostataka) i provjere da li je proizvod prikladan za korištenje [7].

Testiranje programske podrške uključuje izvršavanje programske komponente ili komponente sustava kako bi se procijenilo jedno ili više svojstava od interesa navedenih u [7]. Općenito, ta svojstva ukazuju na opseg u kojem se komponenta ili sustav testira:

- udovoljava zahtjevima koji su usmjeravali njegov dizajn i razvoj,
- odgovara na sve vrste ulaza
- obavlja svoje funkcije u prihvatljivom vremenu,
- dovoljno je upotrebljiva,
- može se instalirati i izvoditi u predviđenom okruženju
- postiže opći rezultat koji njegovi dionici žele.

Budući da je broj mogućih testova čak i za jednostavne programske komponente praktički beskonačan, svako testiranje programa koristi neku strategiju za odabir testova koji su izvedivi za raspoloživo vrijeme i resurse [7]. Kao rezultat, testiranje programa obično (ali ne isključivo) pokušava izvršiti program ili aplikaciju s namjerom pronalaženja grešaka ili drugih nedostataka. Posao testiranja je iterativan proces, jer kada je jedna greška ispravljena, može osvijetliti druge, dublje pogreške ili čak stvoriti nove [7].

Testiranje programske podrške može pružiti objektivne, neovisne informacije o kvaliteti programske podrške i riziku njegovog neuspjeha korisnicima ili sponzorima.

Testiranje programske podrške može se provesti čim postoji izvršni program (čak i ako je djelomično dovršen). Opći pristup razvoju programa često određuje kada i kako se provodi testiranje. Na primjer, u faznom procesu većina se testiranja događa nakon što su zahtjevi sustava definirani i zatim implementirani u programe koji se mogu testirati. Nasuprot tome, pod agilnim pristupom, zahtjevi, programiranje i testiranje često se obavljaju istovremeno [7]. Iako testiranje može odrediti ispravnost programske podrške pod pretpostavkom nekih specifičnih hipoteza ono

ne može identificirati sve nedostatke unutar programa [8]. Umjesto toga, ono pruža kritiku ili usporedbu koja uspoređuje stanje i ponašanje proizvoda s predviđenim ponašanjima - principima ili mehanizmima kojima netko može prepoznati problem [8]. Ta ponašanja mogu uključivati (ali nisu ograničena na) specifikacije, ugovore, usporedive proizvode, prethodne verzije istog proizvoda, zaključke o namjeravanoj ili očekivanoj svrsi, očekivanja korisnika ili kupaca, relevantne standarde, primjenjive zakone ili druge kriterije.

Primarna svrha testiranja je otkrivanje kvarova programske podrške kako bi se otkrile i ispravile greške. Testiranje ne može utvrditi da proizvod funkcionira ispravno u svim uvjetima, već samo da ne funkcionira ispravno pod određenim uvjetima [7]. Opseg testiranja programske podrške često uključuje ispitivanje koda, kao i izvršavanje tog koda u različitim okruženjima i uvjetima, kao i ispitivanje aspekata koda: radi li ono što treba i čini ono što treba. U trenutnoj kulturi razvoja programska organizacija za testiranje može biti odvojena od razvojnog tima. Postoje različite uloge članova testnog tima. Informacije dobivene testiranjem programske podrške mogu se koristiti za ispravljanje procesa razvoja programske podrške.

Svaki proizvod ima ciljanu publiku. Primjerice, publika za videoigre potpuno se razlikuje od publike nekog bankarskog programa. Stoga, kada organizacija razvija ili na neki drugi način ulaže u proizvod, može procijeniti hoće li proizvod biti prihvatljiv krajnjim korisnicima, njegovoj ciljanoj publici, kupcima i drugim dionicima. Testiranje programske podrške pomaže procesu pokušaja izrade ove procjene [7].

4.1. Pristupi i vrste testiranja

Postoje različiti pristupi i vrste testiranja. Testiranje prema pristupima i vrstama možemo podijeliti na statičko, dinamičko, testiranje bijele kutije, testiranje crne kutije, testiranje sučelja komponenti i vizualno testiranje.

4.1.1. Statički i dinamički pristup testiranju

Postoji mnogo pristupa u testiranju programske podrške. Recenzije, prolazi ili inspekcije nazivaju se statičkim testiranjem [9], dok se izvršavanje programiranog koda s danim skupom testnih slučajeva naziva dinamičkim testiranjem [9].

Statično testiranje je često implicitno, kada programski alati / uređivači teksta provjeravaju strukturu izvornog koda ili programski prevoditelji provjeravaju sintaksu i protok podataka kao

statičku analizu programa [9]. Dinamičko testiranje odvija se kada se program pokrene [10]. Dinamičko testiranje može početi prije 100% završetka programa kako bi se testirali pojedini dijelovi koda i primijenili na diskretne funkcije ili module [10].

Statička ispitivanja uključuju verifikaciju, dok dinamičko testiranje uključuje i provjeru valjanosti/validaciju.

Pasivno testiranje znači provjeru ponašanja sustava bez ikakve interakcije s proizvodom. Suprotno aktivnom testiranju, testeri ne pružaju nikakve testne podatke, ali gledaju sistemske dnevnike i tragove. Oni koriste obrasce i proučavaju specifično ponašanje kako bi donijeli neke vrste odluka. To se odnosi na izvanmrežnu provjeru i analizu dnevnika [10].

4.1.2. Testiranje bijele kutije (engl. *white-box testing*)

Testiranje bijele kutije (poznato i kao testiranje s jasnim okvirom, testiranje staklene kutije, transparentno testiranje okvira i strukturno testiranje) provjerava unutarnje strukture ili rad programa, za razliku od funkcionalnosti koja je izložena krajnjem korisniku [7]. U testiranju bijele kutije, interna perspektiva sustava (izvorni kod), kao i vještine programiranja, koriste se za dizajniranje testnih slučajeva. Ispitivač bira ulaze za provođenje staza kroz kod i određuje odgovarajuće izlaze [8].

Dok se testiranje bijelih kutija može primijeniti na razini jedinice, integracije i sustava u procesu testiranja programske podrške, to se obično radi na razini jedinice (engl. *unit testing*) [8]. Ono može testirati staze unutar jedinice, putanje između jedinica tijekom integracije i između podsustava tijekom testa na razini sustava. Iako ova metoda dizajna testa može otkriti mnoge pogreške ili probleme, možda neće otkriti neispunjene dijelove specifikacije ili nedostajuće zahtjeve [8].

Tehnike koje se koriste u ispitivanju bijelih kutija prema [8] i [9] uključuju:

- API testiranje - testiranje aplikacije pomoću javnih i privatnih API-ja (sučelja aplikacijskog programiranja)
- Pokrivenost kodom - izrada testova koji zadovoljavaju neke kriterije pokrivenosti kôdom (npr., Dizajner testiranja može izraditi testove kako bi se svi izrazi u programu izvršili barem jednom)

- Metode ubrizgavanja pogrešaka - namjerno uvođenje kvarova kako bi se ocijenila učinkovitost i strategija testiranja
- Metode ispitivanja mutacije
- Metode statičkog ispitivanja

Alati za pokrivenost kodom mogu procijeniti cjelovitost paketa za testiranje koji je stvoren bilo kojom metodom, uključujući ispitivanje crne kutije. To programerskom timu omogućuje pregled dijelova sustava koji se rijetko testiraju i osigurava da su testirane najvažnije funkcije [10]. Pokrivenost kodom kao programska metrika može se prema [10], [11] i [12] prijaviti kao postotak za:

- Pokrivenost funkcija, koja izvještava o izvršenim funkcijama
- Pokrivenost izvješća, koja izvješćuje o broju izvršenih redaka za dovršenje testa
- Pokrivenost odluke, koja izvještava o tome jesu li izvršene i granice istine i laži određenog testa

Pokrivenost koda od 100% osigurava da se sve staze koda ili grane (u smislu protoka kontrole) izvršavaju barem jednom. To je korisno u osiguravanju ispravne funkcionalnosti, ali nije dostatno jer isti kod može obraditi različite ulaze ispravno ili netočno. Pseudo-testirane funkcije i metode su one koje su pokrivene, ali nisu specificirane [13] (moguće je ukloniti njihovo tijelo bez prekida bilo kojeg test slučaja).

4.1.3. Testiranje crne kutije (engl. *black-box testing*)

Testiranje crne kutije (poznato i kao funkcionalno testiranje) tretira program kao "crnu kutiju", ispitujući funkcionalnost bez ikakvog znanja o internoj implementaciji, a da ne vidi izvorni kod. Ispitivači su samo svjesni onoga što program treba raditi, a ne kako to radi [15]. Metode ispitivanja crnih kutija prema [14],[15] i [16] uključuju: analiza graničnih vrijednosti, testiranje svih parova, tablice prijelaza stanja, testiranje tablica odlučivanja, testiranje temeljeno na modelu, testiranje slučajeva upotrebe, istražna ispitivanja i testiranje na temelju specifikacija.

Testiranje na temelju specifikacija ima za cilj testirati funkcionalnost programske podrške u skladu s primjenjivim zahtjevima [17]. Ova razina testiranja obično zahtijeva da se testeru daju detaljni testni slučajevi, koji zatim mogu jednostavno provjeriti je li za dani ulaz, izlazna vrijednost (ili ponašanje) ili "je" ili "nije" ista kao i očekivana vrijednost naveden u testnom slučaju. Testni slučajevi temelje se na specifikacijama i zahtjevima, tj. na tome što bi aplikacija trebala raditi [17].

Ona koristi vanjske opise programske podrške, uključujući specifikacije, zahtjeve i nacrte za izvođenje testnih slučajeva. Ovi testovi mogu biti funkcionalni ili nefunkcionalni, iako su obično funkcionalni.

Testiranje na temelju specifikacija može biti potrebno kako bi se osigurala ispravna funkcionalnost, ali ona nije dovoljna za zaštitu od složenih ili visokorizičnih situacija [18].

Jedna od prednosti tehnike crne kutije jest ta što nije potrebno znanje programiranja. Što god bi programeri mogli imati, tester vjerojatno ima drugačiji skup i može naglasiti različita područja funkcionalnosti. S druge strane, za testiranje crne kutije je rečeno da je "kao šetnja u mračnom labirintu bez svjetiljke" [19]. Budući da ne ispituju izvorni kod, postoje situacije kada tester napiše mnoge testne slučajeve provjeravajući nešto što je moglo biti testirano samo jednim testnim slučajem ili neke dijelove programa ne testira.

Ova metoda testiranja može se primijeniti na sve razine testiranja programske podrške: testiranje jedinica (engl. *unit-testing*), integracijsko testiranje (engl. *integration-testing*) i testiranje sustava [20].

4.1.4. Testiranje sučelja komponenti

Testiranje sučelja komponenti je varijacija testiranja crne kutije, s naglaskom na vrijednosti podataka izvan samo povezanih radnji komponente podsustava [21]. Praksa testiranja sučelja komponenti može se koristiti za provjeru rukovanja podacima koji se prenose između različitih jedinica ili komponenti podsustava, osim testiranja potpune integracije između tih jedinica. Podaci koji se prosljeđuju mogu se smatrati "paketima poruka", a raspon ili tipovi podataka mogu se provjeriti, za podatke generirane iz jedne jedinice, i testirati na valjanost prije nego što se proslijede u drugu jedinicu [22][23]. Jedna od mogućnosti za testiranje sučelja je da se odvojena datoteka dnevnika pohranjuje, a često se zapisuje vremenska oznaka kako bi se omogućila analiza tisuća slučajeva prijenosa podataka između jedinica danima ili tjednima. Testovi mogu uključivati provjeru rukovanja nekim ekstremnim vrijednostima podataka, dok se druge varijable sučelja prosljeđuju kao normalne vrijednosti. Neobične vrijednosti podataka u sučelju mogu objasniti neočekivane performanse u sljedećoj jedinici.

4.1.5. Vizualno testiranje

Cilj vizualnog testiranja je pružiti programerima mogućnost da ispitaju što se događalo u trenutku kvara programske podrške [24], [25] prikazivanjem podataka na takav način da programer može lako pronaći informacije koje zahtijeva, a informacije su jasno izražene.

U središtu vizualnog testiranja je ideja da prikazivanje nekog problema (ili neuspjeh testa), a ne samo njegovo opisivanje, uvelike povećava jasnoću i razumijevanje. Vizualno testiranje, dakle, zahtijeva snimanje cijelog testnog procesa - hvatanje svega što se događa na testnom sustavu u video formatu [25]. Izlazni videozapisi dopunjeni su ulazom u stvarnom vremenu putem web-kamere za sliku-u-slici i zvučnog komentara s mikrofona.

Vizualno ispitivanje pruža brojne prednosti. Kvaliteta komunikacije se drastično povećava, jer testeri mogu pokazati problem (i događaje koji su do njega doveli) za izvođača, za razliku od toga da ga samo opisuje [25], a potreba za ponovnim neuspjehom testiranja prestat će postojati u mnogim slučajevima. Razvojni inženjer će imati sve dokaze koji su mu potrebni za neuspjeh testiranja i može se umjesto toga usredotočiti na uzrok greške i način na koji ga treba popraviti [25].

„Ad-hoc“ testiranje i istraživačko testiranje važne su metodologije za provjeru integriteta programske podrške, jer zahtijevaju manje vremena za pripremu za implementaciju, dok se važne greške mogu brzo pronaći [26]. U „ad-hoc“ testiranju, gdje se testiranje odvija improviziranim načinom, sposobnost ispitivača da osnuju testiranje dokumentiranih metoda i zatim improviziraju varijacije tih testova može rezultirati rigoroznijim ispitivanjem grešaka. Međutim, ako se ne održi stroga dokumentacija postupaka, jedno od ograničenja „ad-hoc„ testiranja je nemogućnost reproduciranja uočenih nepravilnosti [26].

4.2. Razine testiranja

Testiranje se može podijeliti i prema razinama, najdublja razina testiranja su testovi jedinica, zatim slijede testovi integracije koji testiraju povezanost između jedinica i za kraj testovi sustava koji testiraju rad cijelog sustava.

4.2.1. Testiranje jedinica

Testiranje jedinica odnosi se na testove koji provjeravaju funkcionalnost određenog dijela koda, obično na razini funkcije. U objektno orijentiranom okruženju, to je obično na razini klase, a minimalni testovi jedinica uključuju konstruktore i destruktore. [27]

Ove vrste testova obično pišu programeri dok rade na kodu, kako bi osigurali da određena funkcija radi kako se očekuje. Jedna funkcija može imati višestruke testove, za hvatanje kutnih slučajeva (engl. *edge cases*) ili drugih grana u kodu. Jedinično testiranje ne može provjeriti funkcionalnost programske podrške, već se koristi kako bi se osiguralo da građevni blokovi programske podrške rade neovisno jedan o drugom [27]. Testiranje jedinica ima za cilj eliminirati greške u konstrukciji prije nego što se kod promovira u dodatno testiranje; ova strategija ima za cilj povećati kvalitetu dobivenog programa kao i učinkovitost cjelokupnog procesa razvoja.

Ovisno o očekivanjima organizacije u razvoju programske podrške, testiranje jedinica može uključivati statičku analizu koda, analizu toka podataka, analizu metrike, analizu pokrivenosti kodom i druge prakse testiranja programske podrške [27].

4.2.2. Testiranje integracije

Testiranje integracije je bilo koja vrsta testiranja programske podrške koja nastoji provjeriti sučelja između komponenti programske podrške [28]. Komponente programske podrške mogu se integrirati na iterativan način ili sve zajedno ("veliki prasak").

Testiranje integracije radi otkrivanja grešaka u sučeljima i interakcije između integriranih komponenti (modula). Postupno veće skupine testiranih softverskih komponenti koje odgovaraju elementima arhitektonskog dizajna integrirane su i testirane sve dok program ne funkcionira kao sustav [28].

4.2.3. Testiranje sustava

Testiranje sustava testira potpuno integrirani sustav kako bi se provjerilo zadovoljava li sustav njegove zahtjeve [28]. Na primjer, test sustava može uključivati testiranje sučelja za prijavljivanje korisnika, zatim stvaranje i uređivanje unosa, te slanje ili ispisivanje rezultata, obradom sažetaka ili brisanjem (ili arhiviranjem) unosa, a zatim odjavom.

5. PRISTUP RAZVIJANJA PROGRAMSKE PODRŠKE VOĐEN TESTIRANJEM

Pristupu razvijanja programske podrške vođen testiranjem (engl. *test-driven development*) proces je razvoja softvera koji se oslanja na ponavljanje vrlo kratkog razvojnog ciklusa: zahtjevi se pretvaraju u vrlo specifične testne slučajeve, a zatim se softver poboljšava tako da testovi prođu. Kod ovog pristupa nije dozvoljena implementacija određenih funkcionalnosti ako nije testirana i ako ne prolaze svi ostali testovi unutar aplikacije, što ulijeva povjerenje programeru i jamči da sve radi kako bi trebalo i nakon dodavanja nove funkcionalnosti.

Američki softverski inženjer Kent Beck, koji je zaslužan za to što je razvio ili „ponovno otkrio“ tehniku, prema [29] izjavio je 2003. godine da pristup razvijanja programske podrške vođenog testiranjem potiče jednostavan dizajn i ulijeva povjerenje i sigurnost. Razvoj vođen testom povezan je s konceptima programiranja kojemu prethodi pisanje testova što je zapravo jedan koncept ekstremnog programiranja [30]. Ovakva metodologija programiranja prvi puta je uvedena 1999. godine no tek u novije vrijeme je postala popularna i zapravo zaživjela [31].

5.1. Stil programiranja razvijanjem programske podrške vođenim testovima

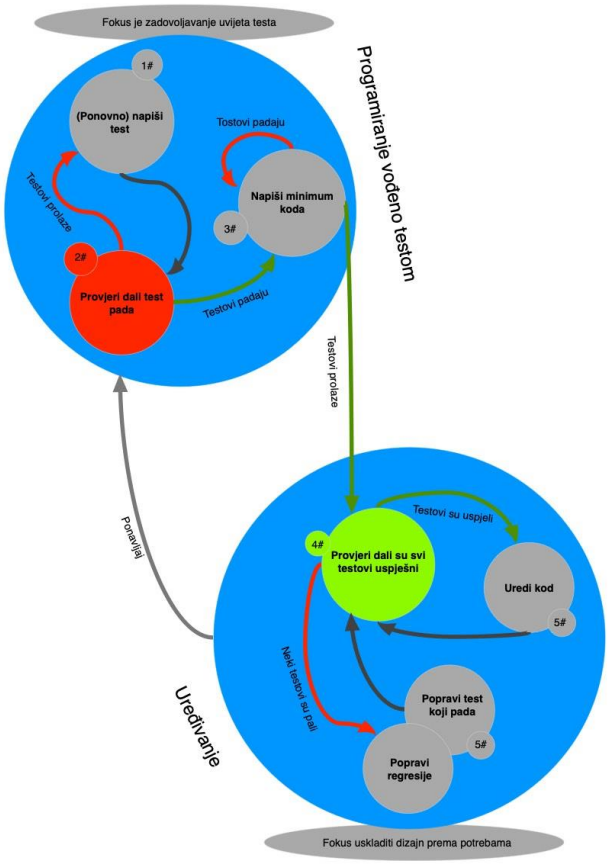
Različiti su aspekti razvijanja programske podrške vođenog testiranjem, na primjer načela "neka bude jednostavno, glupo" (KISS) i "Neće ti trebati" (YAGNI). Fokusiranjem na pisanje samo koda potrebnog da testovi prođu, dizajn aplikacije je često čišći i jasniji nego što se to postiže drugim metodama [31]. Da bi se implementirao kod prema nekom od uzoraka dizajna, testovi se pišu tako da produkcijski kod poštuje taj uzorak dizajna. Na kraju, kod možda neće savršeno zadovoljavati odabrani uzorak dizanja i možda će biti znatno jednostavniji no dok god svi testovi prolaze to je u redu. To također pomaže programeru da misli samo na ono što je zaista bitno u određenom trenutku.

Testove treba pisati prije funkcionalnosti koja se testira. Ovakav pristup pomaže osigurati da je aplikacija napisana na način pogodan za jednostavno testiranje, jer programeri moraju razmotriti kako testirati aplikaciju od samog početka, a ne dodavati testove kasnije [31]. Također osigurava pisanje testova za svaku značajku. Uz to, pisanje testova prvo vodi dubljem i ranijem razumijevanju zahtjeva proizvoda, osigurava učinkovitost testnog koda i održava stalnu usredotočenost na kvalitetu softvera.

Svaki test u početku mora padati. To osigurava da test stvarno radi i može detektirati pogrešku. Tek nakon što se ovo utvrdi, može se implementirati temeljna funkcionalnost. Ovakav način programiranja često se opisuje principom "crveno / zeleno / uredi", gdje crveno znači neuspjeh testa, a zeleno znači da test prolazi [31]. Pristupu razvijanja programske podrške vođen testiranjem temelji se na neprestanom ponavljanju ovih koraka. Prvo se napiše test koji ne prolazi, zatim se napiše kod kako bi test prošao i na kraju se uredi i testni i produkcijski kod. Kreiranje uspješnog koda koji zadovoljava test u svakoj fazi ulijeva zadovoljstvo programeru, povećava samopouzdanje i povećava produktivnost [31]. Navedeni princip je pojednostavljen i pravi ciklus razvijanja programske podrške vođenog testiranjem je nešto složeniji.

5.2. Ciklus razvijanja programske podrške vođenog testiranjem

Ciklus razvijanja programske podrške vođenog testiranjem sastoji se od dodavanja testa, pokretanja svih testova, pisanja koda potrebnog da test prođe, ponovnog pokretanja svih testova te modificiranja i dorade. Ovi koraci stalno se ponavljaju dok funkcionalnost nije potpuno implementirana. Ciklusi razvijanja programske podrške vođenog testiranjem prikazani su na slici 5.1.



Slika 5.1. *Prikaz ciklusa pristupa razvijanja programske podrške vođenog testiranjem [32]*

1. Dodavanje testa

U razvoju vođenom testom, svaka nova značajka započinje pisanjem testa. Prvo se napiše test koji će definirati funkciju ili poboljšanja funkcije koja bi trebala biti vrlo štura. Da bi napisao test, programer mora jasno razumjeti specifikacije i zahtjeve značajke koju implementira. Nakon što je potpuno jasno koji su korisnički zahtjevi i mogući scenariji, programer započinje s pisanjem testova. Ovo je najvažnija razlika između pristupa razvijanja programske podrške vođenog testiranjem i programiranja prilikom kojega se testovi cjelina pišu kasnije nakon implementacije funkcionalnosti: programer je usredotočen na korisničke zahtjeve prije pisanja koda, što je suptilna, ali važna razlika. Na ovaj način programer dobiva puno bolju sliku o tome što je potrebno implementirati, kao i je li nešto slično već implementirano i može se iskoristiti. [30]

2. Pokretanje svih testova i provjera dali novi test pada

Ovaj korak osigurava da novi testni slučaj ispravno radi, odnosno da novi test pada ako testirana funkcionalnost nije implementirana. Budući da je moguće da u trenutku pisanja novog testa u aplikaciji već imamo zahtjevne i složene funkcionalnosti važno je na početku utvrditi da one ne zadovoljavaju uvjete ovog testa te se tako izbjegavaju lažni pozitivni testovi pa se programer može potpuno pouzdati u novo dodani test. [30]

3. Pisanje tek onoliko koda koliko je potrebno da test prođe

Sljedeći korak je napisati kod tako da test prolazi, no u ovom koraku nije potrebno da cijela testirana funkcionalnost bude implementirana, naprotiv, potrebno je napisati apsolutni minimum koda koji je dovoljan da test prođe. [30]

4. Pokretanje svih testova

Pokretanjem svih testova programer provjerava je li i dalje sve u redu, odnosno, je li novo dodani kod uzrokovao promjenu u ponašanju u ostatku koda i time uzrokovao padanje nekog drugog testa. Ovaj korak daje dodatnu sigurnost u novo dodani test i kod. Ako neki od testova padne, programer ispravlja kod dok svi testovi ne prođu i tek tada prelazi na sljedeći korak. [30]

5. Modificiranje i dorada koda

Rastuća baza koda mora se redovito čistiti tijekom korištenja razvijanja programske podrške vođenog testiranjem. Novi se kod može premjestiti s mjesta gdje je bio samo da bi zadovoljio test na neko logičnije mjesto. Svako ponavljanje koda mora biti uklonjeno. Nazivi objekata, klase, modula, varijabli i metoda trebaju jasno predstaviti njihovu trenutnu svrhu i upotrebu jer se dodaje dodatna funkcionalnost. Kako se dodaju značajke, tijela metoda mogu se povećati, kao i objekti. U ovakvim okolnostima najbolje je funkcije i objekte „razbiti“ u više manjih jedinica i pažljivo i precizno ih imenovati radi poboljšanja čitljivosti i održivosti, što će kasnije postati sve korisnije u životnom ciklusu softvera. Hijerarhije nasljeđivanja mogu se preurediti da bi bile logičnije i korisnije, a možda imale koristi od poznatih uzoraka dizajna. Postoje posebne i opće smjernice za uređivanje i stvaranje čistog koda [33][34]. Neprekidnim ponovnim pokretanjem testnih slučajeva u svakoj fazi uređivanja, programer može biti uvjeren da postupak ne mijenja bilo koju postojeću funkcionalnost. Koncept uklanjanja duplikata važan je aspekt svakog softverskog dizajna. U ovom se koraku, provodi uklanjanje duplikata između testnog koda i produkcijskog koda - na primjer, čarobni brojevi ili tekstualni podatci koji se ponavljaju u oba slučaja samo kako bi test prošao u koraku 3.

6. Ponavljanje

Počevši s još jednim novim testom, ciklus se zatim ponavlja kako bi se poboljšala funkcionalnost. Veličina koraka uvijek treba biti mala, sa samo 1 do 10 izmjena između svakog testnog postupka. Ako novi kod ne zadovoljava novi test ili drugi testovi neočekivano padaju, programer bi trebao zanemariti novo dodano ponašanje i vratiti se na stanje kada je sve radilo, na ovaj način se uklanja pretjerano trošenje vremena na otklanjanje problema. Kontinuirana integracija pomaže pružanjem kontrolnih točaka na koje se programer može vratiti. Pri korištenju vanjskih biblioteka nije potrebno testiranje njihovog koda i za to je odgovoran njihov autor, osim ako postoji opravdan razlog za sumnju da u kodu biblioteke postoje greške, no tada je bolje ne koristiti tu biblioteku. [34]

5.3. Najbolje prakse prilikom razvijanja programske podrške vođenog testiranjem

Najbolje prakse koje bi pojedinac mogao slijediti bile bi odvajanje zajedničke logike postavljanja i uništavanja objekta kojeg se testira u zasebne metode kako bi za vrijeme cijelog životnog ciklusa testirane komponente fokus bio samo na rezultate potrebne za prolazak testa i kako bi bilo lakše dizajnirati vremenski ovisne testove tako da ih je moguće pokrenuti u okruženju koje nije u

stvarnom vremenu [34]. Uobičajena praksa da se dopušta 5-10 posto testova koji kasne s izvršavanjem(testovi osjetljivi na vrijeme) smanjuje potencijalni broj lažnih negativnih testova [34]. Također se predlaže da se testni kod tretira s istim poštovanjem kao proizvodni kod. Testni kod mora ispravno raditi i za pozitivne i za negativne slučajeve, dugo trajati te biti čitljiv.

5.4. Loše prakse prilikom razvijanja programske podrške vođenog testiranjem

Testni slučajevi nikako ne bi trebali ovisiti o stanju unutar sustava koje je izmijenjeno u nekome od prethodnih testova, odnosno testovi nad jedinicama bi se uvijek trebali pokretati u jasnim i pred definiranim uvjetima [35]. Također, testni slučajevi ne bi trebali ovisiti jedan o drugom, najbolji pokazatelj da ova praksa nije zadovoljena je ako je kod testiranja komponente važan poredak izvršavanja testova. Ne treba se provoditi testiranje performansi izvršavanja i vremena izvršavanja [35]. Loša praksa je kreiranje testnih slučajeva koji testiraju više stvari, tako zvanih "sveznajućih" slučajeva, u tom slučaju ovakve testne slučajeve teško je održavati i nejasno je zašto je došlo do pada i koje je točno ponašanje uzrokovalo pad testa. Testiranje implementacijskih detalja i testovi koji se sporo pokreću također su prakse koje je potrebno izbjegavati.

5.5. Prednosti razvijanja programske podrške vođenog testiranjem

Iako se razvijanjem programske podrške vođenim testiranjem piše više testova, programeri su obično puno produktivniji [35]. Koristeći pristup razvijanja programske podrške vođenog testiranjem smanjuje se korištenje programa za otklanjanje pogrešaka (engl. *debugger*). Kada testovi počnu padati, uz korištenje sustava za praćenje inačica koda, jednostavno se vrati na zadnju verziju u kojoj su testovi prolazili. Ovakav način je često produktivniji od procesa otklanjanja pogrešaka [36]. Pristupu razvijanja programske podrške vođenog testiranjem nudi više od jednostavne provjere ispravnosti i može znatno poboljšati dizajn programa [37]. Ako se programer najprije mora usredotočiti na kreiranje testnih slučajeva, on tada mora zamisliti kako klijenti koriste funkcionalnost i to zapravo jesu testni slučajevi. Pristup razvijanja programske podrške vođen testiranjem nudi mogućnost razbijanja problema u manje cjeline. To omogućava programeru da se usredotoči na zadatak na kojemu trenutno radi, jer je prvi cilj proći test. Iznimni slučajevi i postupanje s pogreškama ne razmatraju se u početku, a testovi za stvaranje ovih vanjskih okolnosti provode se odvojeno. Pristup razvijanja programske podrške vođen testiranjem povećava razinu sigurnosti tako što je sav pisani kod pokriven barem jednim testom. To cijelome timu daje veće povjerenje u kod ali i veće samopouzdanje kada je potrebno uvesti nove

funkcionalnosti. Iako je istina da je potrebno više koda s pristupom razvijanja programske podrške vođenim testiranjem, nego kada se on ne koristi, zbog testiranja manjih cjelina, ukupno vrijeme implementacije koda moglo bi biti kraće na temelju modela Müllera i Padberga [38]. Veliki broj testova pomaže ograničiti broj nedostataka u kodu. Česta pokretanje testova i od rane faze razvijanja programa pomažu uvidjeti nedostatke u razvojnom ciklusu, sprječavajući ih da postanu globalni i skupi problemi. Uklanjanje nedostataka u ranom postupku obično izbjegava dugotrajne i naporne uklanjanje pogrešaka kasnije u projektu.

Pristup razvijanja programske podrške vođen testiranjem može dovesti do modularnog, fleksibilnog i proširivog koda. Do ovog se efekta često dolazi zato što metodologija zahtijeva da programeri softvera misle na male jedinice koje se mogu samostalno napisati, testirati i integrirati. To dovodi do manjih, više usredotočenih klasa, koje ovise o apstraktnim metodama [35]. Upotreba dizajn uzorka lažnog objekta (engl. *mock*) također pridonosi razdvajanju koda na manje cjeline, jer ovaj uzorak zahtijeva da se kod napiše tako da se moduli mogu lako prebacivati između modela za testiranje i "stvarnih" inačica za uporabu.

Budući da se piše samo onoliko koda koliko je potrebno da test prođe, automatizirani testovi obično pokrivaju svaki put koda [36]. Kao rezultat, automatizirani testovi koji proizlaze iz pristupa razvijanja programske podrške vođenog testiranjem su obično jako temeljiti u otkrivanju neočekivane promjene u ponašanju koda. Ovo otkriva probleme koji mogu nastati tamo gdje promjena kasnije u razvojnom ciklusu neočekivano mijenja druge funkcionalnosti.

5.6. Ograničenja pristupa razvijanja programske podrške vođenog testiranjem

Pristupu razvijanja programske podrške vođenog testiranjem ne provodi dovoljno ispitivanja u situacijama kada su potrebni potpuno funkcionalni testovi za utvrđivanje uspjeha ili neuspjeha, zbog toga što se zasniva na testiranju manjih jedinica koda. Primjeri za to su korisnička sučelja, programi koji rade s bazama podataka i neki koji ovise o određenim mrežnim konfiguracijama. Pristup razvijanja programske podrške vođen testiranjem potiče programere da pišu minimalnu količinu koda u takvim modulima i da logiku koja se može testirati pišu dijelovima odvojenim od ovih modula, ovo se postiže korištenjem lažnih objekata koji predstavljaju „svijet“ izvan promatranog testiranog modula. [39]

Testove jedinica (engl. *unit test*) kreirane u okruženju u kojem se radi razvijanje programske podrške vođeno testiranjem obično kreira programer koji piše kod koji se testira. Stoga testovi mogu biti prilagođeni kodu: ako, primjerice, programer ne shvati da se moraju provjeriti određeni ulazni parametri, najvjerojatnije niti test ni kod neće provjeriti te parametre. Drugi primjer: ako programer pogrešno protumači zahtjeve za modul koji razvija, kod i testni slučajevi radit će pogrešno odnosno testirat će se zapravo pogrešno ponašanje. Zbog toga će testovi proći, što daje lažni osjećaj ispravnosti implementacije. [39]

Podrška uprave unutar organizacije u kojoj je programer zaposlen je neophodna. Bez cijele organizacije koja vjeruje da će pristup razvijanja programske podrške vođen testiranjem poboljšati proizvod, uprava može osjetiti kako je trošenje vremena na pisanje testova beskorisno[40]. Veliki broj testova koji su zadovoljeni može donijeti lažan osjećaj sigurnosti, što rezultira manjim brojem dodatnih aktivnosti testiranja softvera, poput integracijskog testiranja i testiranja sukladnosti. Testovi postaju dio projekta za održavanje. Loše napisani testovi i sami su skloni kvaru i skupo ih je održavati. Postoji rizik da se testovi koji redovito generiraju lažne pogreške zanemaruju, tako da kad dođe do stvarnog kvara, neće biti otkriven.

5.7. Lažni objekti i integracijski testovi kod razvijanja programske podrške vođenog testiranjem

Testovi jedinica (engl. *unit test*) su tako nazvani, jer svaki testira jednu jedinicu koda. Složeni modul može imati tisuću testova jedinica, a jednostavan modul može imati samo deset. Testovi koji se koriste za razvijanje programske podrške vođeno testiranjem nikada ne bi trebali prijeći granice procesa u programu, a kamoli mrežne veze. Tako se uvode kašnjenja zbog kojih se testovi polako izvode i onemogućavaju programerima da pokreću sve testove. Uvođenje ovisnosti o vanjskim modulima ili podacima također pretvara testove jedinica u testove integracije. Ako se jedan modul ne ponaša pravilno unutar lanca modula povezanih integracijskim testovima tada nije baš najjasnije koji je modul uzrok problema i kako je do njega došlo.

Kad se kod koji se razvija oslanja na bazu podataka, web uslugu ili bilo koji drugi vanjski proces ili uslugu, provođenje razdvajanja u manje dijelove koji su izolirani također je prilika i motiv za kreiranje modularnog koda koji se lako testira i moguće ga je koristiti na više mjesta odnosno nije specifičan za trenutnu implementaciju.

Prema [40] dva koraka su potrebna kako bi se ovo ostvarilo:

- Kad god je u konačnom dizajnu potreban vanjski pristup, treba definirati sučelje koje opisuje dostupni pristup. Ovo je također opisano principom inverzije ovisnosti (engl. *dependency inversion principle*) što je jedan od 5 principa koji čine poznate SOLID dizajn principe u objektno orijentiranom programiranju.
- Sučelje treba implementirati na dva načina, od kojih jedan stvarno pristupa vanjskom procesu, a drugi je lažni (engl. *mock*). Lažni objekti trebaju bilježiti zahtjeve kako bi se nad njima radile provjere ispravnog ponašanja.

Lažne objektne metode koje vraćaju podatke, lažirajući podatke koji bi zapravo dolazili iz baze podataka ili s poslužitelja, mogu pomoći procesu testiranja tako što uvijek vraćaju iste, realne podatke na koje se testovi mogu osloniti [40]. Također se mogu postaviti u unaprijed definirane načine grešaka, tako da se rutine upravljanja pogreškama mogu razviti i pouzdano ispitati. U načinu pogreške, metoda može vratiti nevažeći, nepotpuni ili nulti odgovor ili može vratiti iznimku. Budući da kod koji se testira pomoću umjetnih objekata zahtjeva zamjenu stvarnih objekata umjetnima, odnosno stvarne ovisnosti (engl. *dependencies*) umjetnima, za to se koristi puni potencijal principa inverzije ovisnosti odnosno primjena tog principa u vidu ubrizgavanja ovisnosti (engl. *dependency injection*). Umjetni objekt se pomoću ubrizgavanja ovisnosti preda objektu kojega testiramo. Dakle stvarna baza podataka ili drugi vanjski kod nikada se ne testira prilikom razvijanja programske podrške vođenog testiranjem. Kako bi se izbjegle pogreške koje bi mogle proizići iz toga, potrebni su drugi testovi koji omogućuju testiranje upravljačkog koda s "stvarnim" implementacijama lažnih objekata [40]. Za to se koriste integracijski testovi koji se bitno razlikuju od testova jedinica. Manje ih je, i oni se moraju izvoditi rjeđe od testova jedinica. Zbog načina na koji testiranje radi i same prirode testiranja stvari s integracijskim testovima su nešto kompliciranije. Integracijski testovi koji mijenjaju bilo koji vanjski modul, bazu podataka ili sl. uvijek trebaju biti pažljivo osmišljeni uzimajući u obzir početno i konačno stanje datoteka ili baze podataka, čak i ako bilo koji test ne uspije. To se prema [40] često postiže nekom kombinacijom sljedećih tehnika:

- *TearDown* metoda koja je sastavni dio mnogih okvira testiranja.
- *Try, catch, finally* metodama za hvatanje iznimki
- Kreiranje kopije baze podataka i postavljanje baze u stanje kopije nakon svakog testa
- Inicijalizacija baze s početnim stanje prije svakog testa

5.8. Načini testiranja aplikacije za prikaz preostalog broja pacijenata u redomatskom redu

Aplikacija je rađena prema pristupu razvijanja programske podrške vođenog testiranjem principima i poštujući pravila i metodologije pristupa razvijanja programske podrške vođenog testiranjem. Budući da je aplikacija rađena na ovaj način, gotovo cijela aplikacija je testirana i možda je bolje navesti stvari koje nisu testirane. Kao što je ranije navedeno, kod izrade aplikacije pristupom razvijanja programske podrške vođenog testiranjem ne testira se kod vanjskih biblioteka kao ni kod korisničkog sučelja, tako da su to stvari koje u ovoj aplikaciji nisu testirane. Također nije testiran kod koji je usko povezan uz neke sistemske operacije kao što je kod za skeniranje QR koda, ovo je testirano od strane Apple-a i nema razloga za dodatnim testiranjem. Dodatni izazov u testiranju predstavlja reaktivni način programiranja i reaktivnog koda. Također, unutar aplikacije su napisani i integracijski testovi za određene slučajeve za koje je to bilo sigurno napraviti. Testirane su performanse aplikacije s alatima dostupnima unutar IDE-a što uključuje testiranje opterećenja procesora i potencijalnog curenja memorije (engl. *memory leak*). Aplikacija je testirana i ručno nakon što je završena, a za to su napisani testni scenariji prema kojima je rađeno testiranje.

6. PROGRAMSKO RJEŠENJE MOBILNE APLIKACIJE

Aplikacija je napisana u programskom jeziku Swift u prirodnom okruženju Xcode. Xcode je integrirano razvojno okruženje (IDE) za macOS koji sadrži paket alata za razvoj softvera koji je Apple razvio za macOS, iOS, iPadOS, watchOS i tvOS. Prvo izdanje bilo je 2003 godine, a najnovije stabilno izdanje je inačica 10.3 i dostupno je putem Mac App Store-a besplatno za korisnike macOS Mojave.

6.1. Programski jezik Swift

Swift je programski jezik za opću namjenu koji je Apple Inc. razvio za iOS, iPadOS, macOS, watchOS, tvOS, Linux i z/OS. Swift je dizajniran za rad s Appleovim Cocoa i CocoaTouch bibliotekama i velikim dijelom postojećeg Objective-C koda napisanog za Apple proizvode [41]. Izgrađen je s okvirom za pred-obradu otvorenog koda LLVM i uključen je u Xcode od verzije 6, objavljene 2014. Na Apple platformama, koristi Objective-C pokretačku mašinu (enlg. *runtime*) što omogućava C, Objective-C, C++ i Swift kod za pokretanje unutar jednog programa. Apple je prilikom kreiranja ovog programskog jezika imao cilj da Swift podržava mnoge temeljne koncepte povezane s Objective-C programskim jezikom, osobito dinamičku otpremu (engl. *dynamic dispatch*), široko rasprostranjeno dinamičko povezivanje(engl. *late binding*), proširivo programiranje i slične značajke, ali na "sigurniji" način, olakšavajući uhvatiti programske greške [42]. Swift ima značajke koje se bave nekim uobičajenim programskim pogreškama kao što je preusmjeravanje nul pokazivača (engl. *null pointer*) i pruža sintaktički šećer (engl. *syntactic sugar*) kako bi se izbjegle piramide propasti (engl. *pyramid of doom*). Swift podržava koncept proširivanja protokola, sustava proširivanja koji se može primijeniti na tipove, strukture i klase, što Apple promovira kao stvarnu promjenu u programskim paradigmama koje nazivaju „protokolarno orijentirano programiranje“ [43].

6.2. ReactiveX i RxSwift

Kao što je ranije navedeno, aplikacija se snažno oslanja na reaktivnim ekstenzijama radi jednostavnijeg upravljanja asinkronim događajima. RxSwift dio je ReactiveX biblioteke koja podržava većinu popularnih programskih jezika kao što su: Java, Scala, C#, C++, Clojure, JavaScript, Python, Groovy, JRuby i puno drugih. ReactiveX služi za kreiranje asinkronih programa a temelji se na događajima koristeći se vidljivim nizovima podataka. Ova biblioteka zapravo proširuje uzorak promatrača (engl. *observable pattern*) kako bi podržao nizove podataka i događaja i uz to omogućava njihovo povezivanje, manipulaciju i transformiranje pomoću

ugrađenih operatora. Operatori omogućuju da se na deklarativan način kreiraju nizovi događaja a da pritom otklanjaju poteškoće vezanu za operacije nad nitima i druge kompleksne radnje povezane s istovremenosti. Ova biblioteka temelji se na tokovima događa (engl. *observalbes*), oni nam omogućavaju da tretiramo tok asinkronih podataka kao obična polja podataka koja koristimo kod normalnog programiranja. Na ovaj način izbjegavamo kreiranje mreže *callback* funkcija što znatno smanjuje veličinu koda i njegovu kompleksnost a time i mogućnost nastanka grešaka. Poznato je da manje koda znači i manju vjerojatnost za pojavljivanje grešaka unutar koda. Tehnike poput Java *futuresa* jednostavno je koristiti za jednu razinu asinkronog izvršavanja, ali problemi započinju ako se počnu ugnježđivati. [44]

Features funkcije je teško koristiti za optimalno sastavljanje uvjetnih asinkronih tokova izvršavanja (ili nemoguće, jer latencije svakog zahtjeva variraju u vrijeme izvođenja). Daleko od toga da to nije izvedivo, ali brzo postaje komplicirano (i time se povećava vjerojatnost pogreške) ili se počnu primjenjivati metode blokiranja i čekanja rezultata što zapravo ugrožava svojstva asinkronog ponašanja. ReactiveX tokovi događaja podržavaju ne samo emitiranje pojedinih skalarnih vrijednosti (kao što to čine *features* funkcije), već i nizove vrijednosti ili čak beskonačne tokove podataka. U ovom radu, primjer beskonačnog toka podataka je tok događaja koji svakih 3 sekunde dohvaća podatke i to radi dok god je aplikacija uključena. Tok događaja je zapravo apstrakcija koja može biti korištena za sve ove slučajeve. [44]

6.2.1. ReactiveX operatori

ReactiveX pruža kolekciju operatora s kojima je moguće filtrirati, odabrati, transformirati, kombinirati i sastaviti promatrane podatke. Svaka implementacija ReactiveX-a za određeni programski jezik implementira skup operatora. Iako se implementacije dosta poklapaju, postoje i neki operatori koji su implementirani samo u određenim programskim jezicima. Također, svaka implementacija nastoji imenovati svoje operatore da sliče onima sličnih metoda koje su već poznate iz drugih konteksta u tom jeziku. Većina operatora djeluje na tok događaja i vraća novi tok događaja. To omogućuje primjenu ovih operatora jedan za drugim u lancu. Svaki operator u lancu mijenja stanje koje proizlazi iz prethodnog operatora. [45]

Kod drugih dizajn obrasci, poput obrasca graditelja (engl. *builder pattern*), u kojem različite metode određene klase djeluju na objekt iste klase modificirajući taj objekt. Ovi obrasci također omogućuju povezivanje metoda slično kao i s uzorkom promatrača. No, iako kod uzorka graditelja,

redoslijed po kojem se metode pojavljuju u lancu obično nije bitan, s operaterima toka događaja je redoslijed itekako bitan. [45]

Lanac operatora nad tokom događaja ne djeluje nezavisno na originalni tok događaja koji je prvi u lancu, nego svaki operator mijenja izlaz prethodnog operatora odnosno radi operacije nad njim. [45]

6.2.2. Operatori RxSwift biblioteke

Operatori se općenito dijele na osam većih cjelina a to su:

- Operatori za kreaciju toka događaja
- Operatori za transformiranje toka događaja
- Operatori za filtriranje toka događaja
- Operatori za kombiniranje toka događaja
- Operatori za rješavanje pogrešaka
- Operatori za manipulaciju s tokovima događaja
- Uvjetni i Boolean operatori
- Matematički operatori
- Operatori za povezivanje
- Operatori za pretvaranje toka događaja

Budući da postoji veliki broj operatora za ReactiveX biblioteku, to programeru daje veliku fleksibilnost a k tome je i dokumentacija detaljna, dobro opisana i na različite načine objašnjava rad određenih operatora.

6.2.3. Primjer vladanja nekih od RxSwift operatora

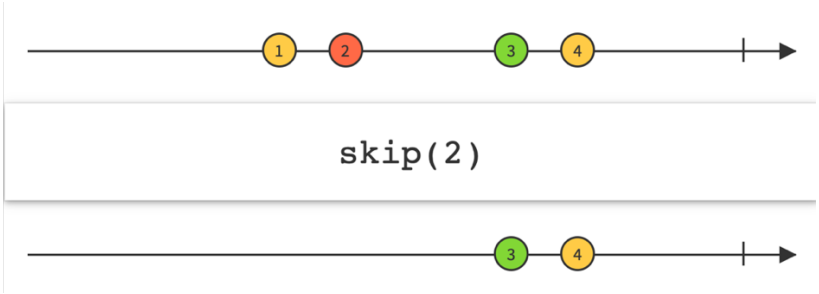
Budući da je RxSwift bitan dio aplikacije neki od operatora korištenih u aplikaciji bit će detaljnije opisani u nastavku. Također, testiranje reaktivnog koda provodi se na specifičan način pomoću RxTest biblioteke što je također znatno utjecalo na tijek izrade aplikacije i bit će detaljnije objašnjeno u nastavku.

U aplikaciji za pregled stanja u redovima neki od korištenih operatora su *Interval*, *FlatMap*, *Just* i *Skip*. Dokumentacija ReactivX biblioteke vladanja operatora opisuje između ostaloga i pomoću *Marble* dijagrama, što je zapravo inovativan i jako dobar način za opisivanje funkcionalnosti

operatora, a dijagrami pokazuju kako se točno ponaša tok događaja nakon što se na njega primjeni neki od operatora.

Operator *Skip*

Skip operator ima jednostavnu funkciju a to je da, kao što to njegovo ime govori, preskoči određeni broj evenata i emitira evente nakon njih [45]. Budući da im je funkcionalnost poprilično jednostavna, može poslužiti kao dobar primjer za početnike u reaktivnom programiranju prilikom objašnjavanja načina rada operatora ali i čitanja iz *marble* dijagrama. Na slici 6.1 prikazan je *marble* dijagram za opis funkcionalnosti operatora *skip*.

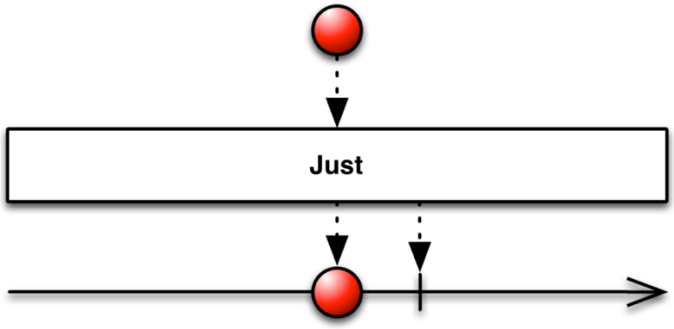


Slika 6.1. *Prikaz marble dijagrama skip operatora[45]*

U aplikaciji rađenoj u sklopu ovoga rada, ovaj operator je korišten za izbjegavanje inicijalnog događaja releja koji šalje zahtjeve na poslužitelj, a zbog prirode njegove implementacije sadrži prazan inicijalni događaj.

Operator just

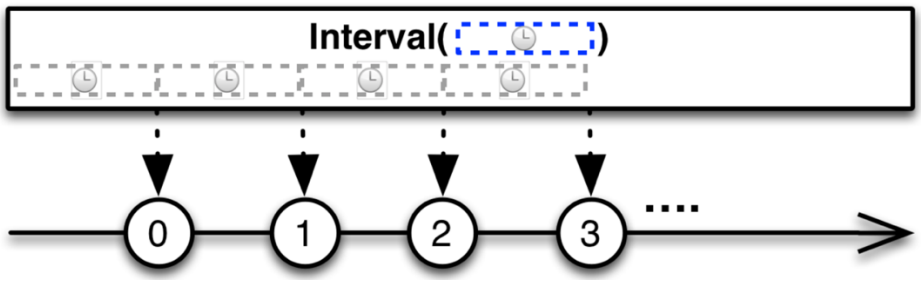
Just operator služi za pretvaranje bilo kojeg podatka u toku događaja koji emitira taj podatak [45]. Ovaj operator je poprilično jednostavan, ali i često korišten u aplikacijama koje se oslanjaju na reaktivno programiranje. On je najjednostavniji način za pretvoriti neku vrijednost u tok događaja, a budući da operatori mogu raditi različite operacije, ali samo nad tipom toka događaja, jasno je zašto je ovaj operator toliko koristan. Na slici 6.2 prikazan je *marble* dijagram za operator *just*.



Slika 6.2. *Prikaz marble dijagrama just operatora[45]*

Interval operator

Operator *interval* vraća tok događaja koji emitira beskonačan tok brojčanih vrijednosti s rastućim redoslijedom, a pri tome programer može odabrati vremenski razmak između dva emitiranja [45]. Na slici 6.3 prikazan je *marble* dijagram operatora *interval*.

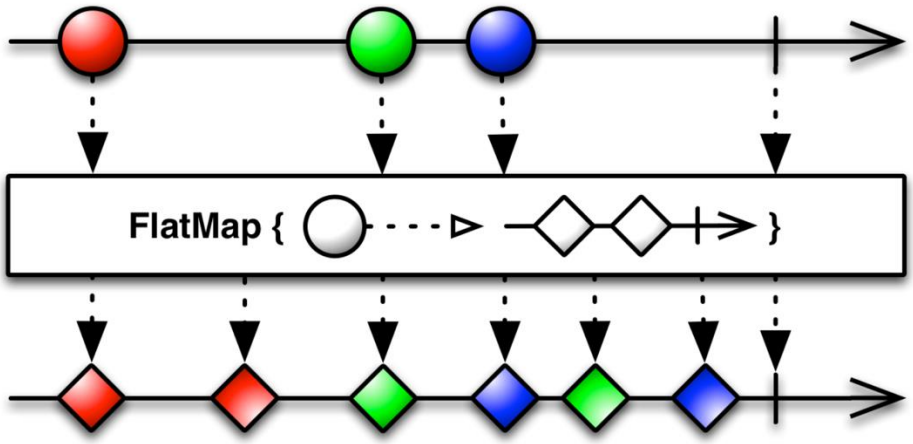


Slika 6.3. *Prikaz marble dijagrama interval operatora[45]*

Ovaj operator u aplikaciji je korišten kako bi se zahtjevi na poslužitelj slali svakih 3 sekunde, a to je ostvareno u kombinaciji sa sljedećim operatorom koji će biti objašnjen, a to je operator *flatMap*.

FlatMap operator

Operator *flatMap* transformira tok događaja primjenjujući funkciju koju programer definira na svaku stavku koju emitira izvor toka događaja, pri čemu ta funkcija vraća toka događaja koji i sam emitira stavke [45]. *FlatMap* zatim spaja emitirane tokove događaja, emitirajući ove spojene rezultate kao jedan tok događaja. Na slici 6.4 prikazan je *marble* dijagram *flatMap* operatora.

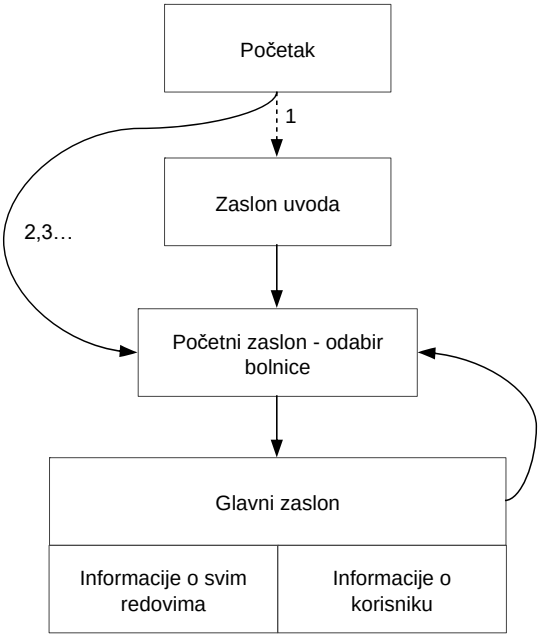


Slika 6.4. *Prikaz marble dijagrama flatMap operatora[45]*

U aplikaciji je korištena modificirana inačica ovog operatora *FlatMapLatest* čije se vladanje razlikuje jedino po tome što kod *FlatMapLatest* operatora rezultanti tok događaja emitira samo evente zadnjega toka događaja koji je se obrađuje, a to je upravo ono što je potrebno u ovom slučaju budući da se radi zahtjev na poslužitelj.

6.3. Dijagram toka aplikacije

Nakon što korisnik prvi puta uključi aplikaciju, pokaže mu se uvodni zaslon s uputama o korištenju aplikacije. Ovaj zaslon pokazuje se samo prvi puta kada je aplikacija pokrenuta nakon instalacije. Nakon zaslona uvoda, slijedi početni zaslon na kojemu je potrebno odabrati bolnicu, na ovom zaslonu korisnik ima mogućnost ručnog odabira bolnice ili automatskog tako da skenira QR kod. Na slici 6.5 prikazan je dijagram toka aplikacije za pregled preostalog broja pacijenata u redomatskom redu.



Slika 6.5. *Dijagram toka aplikacije*

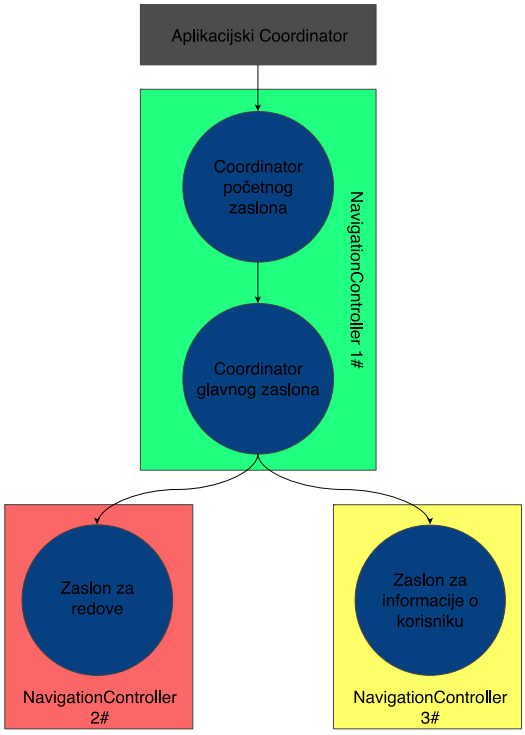
Jednom kada je bolnica odabrana ili QR kod skeniran, otvara se glavni zaslon koji se sastoji od dva pod zaslona. Jedan zaslon prikazuje informacije u redovima bolnice dok drugi prikazuje informacije o korisniku odnosno specifičnom broju unutar nekoga reda. Korisnik se s glavnoga zaslona u bilo kojemu trenutku može vratiti na zaslon odabira bolnice.

6.4. Arhitektura aplikacije

Arhitektura aplikacije napravljena je po uzoru na načela *CLEAN* arhitekture koju je predstavio Robert C. Martin poznatiji pod nadimkom Uncle Bob, koja je objavljena u njegovom blog postu 2012. godine pod nazivom *The Clean Architecture* [46]. No, nije strogo vezana uz *CLEAN* arhitekturu nego je proizašla iz razvijanja programske podrške vođenog testiranjem. Dio sustava zadužen za korisničko sučelje rađen je po uzoru na *MVVM* (engl. *Model,View,View-Model*) arhitekturu koju su osmislili Microsoftovi arhitekti Ken Cooper i Ted Peters i njome su posebno pojednostavili programiranje korisničkih sučelja usmjereno na događaje. Ova arhitektura bila je sasvim logičan odabir budući da se koristi RxSwift kroz cijelu aplikaciju. Za navigaciju i kreiranje zaslona korišten je dizajn obrazac koordinatora koji je predstavljen 2015. godine a osmislio ga je Soroush Khanlou i predstavio u blog postu pod nazivom *The Coordinator* [47]. Ovaj dizajn obrazac specifičan je za iOS aplikacije i rješava probleme kreiranja *ViewControllera* koji su zapravo temeljne komponente svake iOS aplikacije.

6.4.1. Dizajn obrazac koordinatora

Koordinator obrazac nastao je kako bi riješio problem kreiranja zaslona. Kreiranje zaslona odnosno glavnih komponenti unutar iOS aplikacije, *ViewControllera* ranije se radilo tako da bi jedan *ViewController* kreirao drugi i tako bi nastao graf *ViewControllera*. Problem u takvom načinu kreiranja *ViewControllera* je što se programer često zatekne u situaciji da iz *ViewController-a* „djeteta“ mora reći *ViewControlleru* „roditelju“ da napravi određenu akciju. Prema tome, „dijete“ je svjesno svog roditelja i ima pristup njegovim akcijama. U stvarnom životu, „dijete“ nikada ne bi trebalo govoriti „roditelju“ što da radi, a u programiranju „dijete“ ne bi trebalo znati niti tko mu je roditelj niti kako je ono nastalo odnosno tko ga je kreirao. Ovo je slikoviti prikaz problema s kojim su se programeri često susretali prije nego li je predstavljen Koordinator uzorak. Na slici 6.6 prikazan je graf koordinatora aplikacije za pregled preostalog broja pacijenata u redomatskom redu.



Slika 6.6. *Graf Koordinatora u aplikaciji za prikaz preostalog broja pacijenata*

Ovaj dizajn uzorak sastoji se od Koordinator-a za svaki zaslon, odnosno Koordinator inicijalizira svaki *ViewController*, ali i sve ostale komponente korisničkog sučelja kao na primjer *ViewModel*. Svaki Koordinator može posjedovati polje „djece“ Koordinatora, i tako se kreira graf Koordinatora, koji predstavlja navigaciju kroz aplikaciju. Ovim uzorkom je navigacija aplikacije

znatno preglednija i pojednostavljena, a važna odgovornost je uklonjena iz *ViewModel*-a što je korak ka rješavanju problema ogromnih *ViewControllera* koji imaju previše odgovornosti. Implementacija Koordinatora je prilično jednostavna, potrebno je kreirati protokol koji ima instancu *NavigationController-a*, zatim polje Koordinatora djece i na kraju funkciju start koju je potrebno implementirati. Na slici 6.7 prikazan je protokol za implementaciju koordinatora.

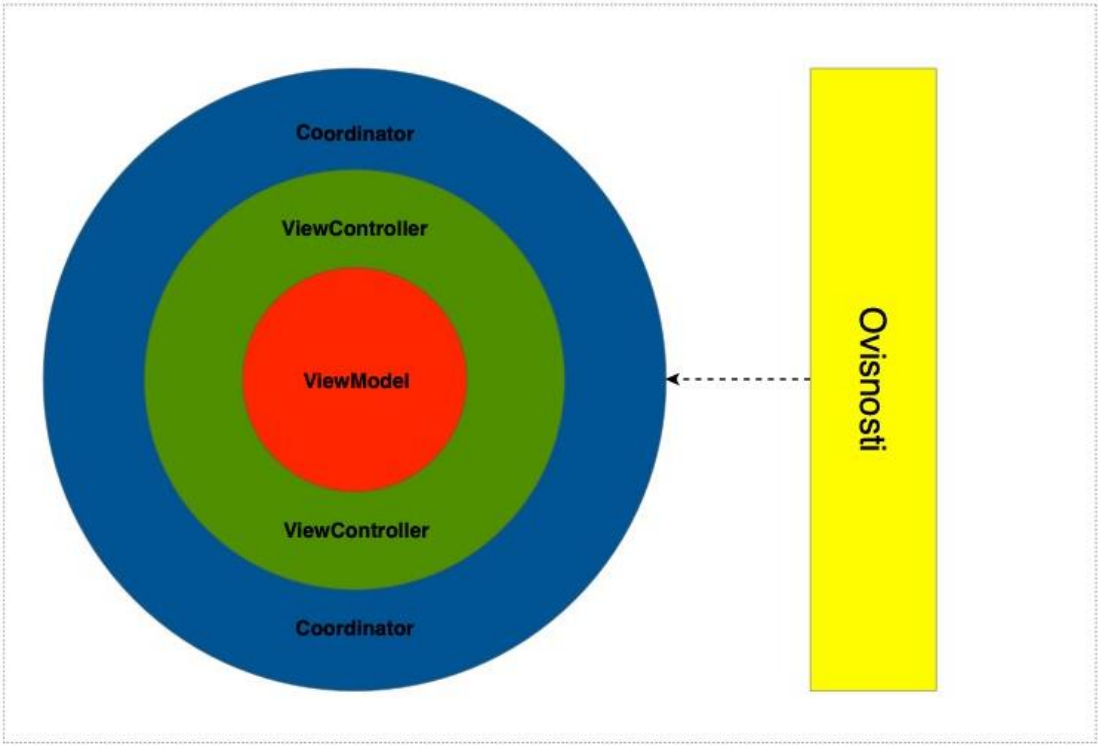
```
protocol Coordinator: class{
    var childCoordinators: [Coordinator] {get set}
    var presenter: UINavigationController {get}
    func start()
}
```

Slika 6.7. *Prikaz koordinator protokola*

Unutar funkcije start obično se postavlja *ViewController* na navigacijski stog *NavigationController*-a te tako prikazuje na zaslonu. Unutar Koordinatora moguće je kreirati Koordinatore „djecu“, nužno ih je spremiti u polje „djece“ kako bi „roditelj„ imao povezanost s „djetetom“, drugim riječima, ako ne dodamo Koordinator „dijete“ u polje unutar „roditelja“, tada smo prekinuli graf navigacije. Nakon što je „dijete“ gotovo s radom, „roditelj“ ga briše iz svoga polja i sustav ga čisti odnosno briše iz memorije.

6.4.2. Arhitektura vezana za korisničko sučelje

Arhitektura korisničkog sučelja sastoji se od koordinatora kao vanjske komponente koja kreira ostale komponente, zatim *ViewControllera*, i *ViewModel-a* koji je sadržan unutar *ViewControllera*. Na slici 6.8 prikazana je arhitektura korisničkog sučelja.



Slika 6.8. *Prikaz arhitekture korisničkog sučelja*

Komponenta u sredini, *ViewModel* zadužena je za svu poslovnu logiku aplikacije. *ViewController* je usko povezana sa sustavom, posjeduje metode životnoga ciklusa i reagira na korisnikove interakcije s aplikacijom. Ova komponenta ne smije sadržavati nikakvu poslovnu logiku i ona samo obavještava *ViewModel* o korisničkim interakcijama, zatim *ViewModel* na te interakcije nekako reagira, izvrši određene operacije i odašilje signal o tome što se treba dogoditi. Ove signale *ViewController* sluša i na njih reagira. Ovakva arhitektura osigurava lako testiranje komponenti. Koordinator kreira *ViewController* i *ViewModel* koji je smješten unutar *ViewControllera*, također Koordinator pomoću ubrizgavanja ovisnosti (engl. *dependency injectiona*) predaje *ViewControlleru* i *ViewModelu* njihove ovisnosti i oni su tako izolirani od ostatka sustava. Testiranje je jako jednostavno budući da sve komponente zapravo ovise o apstrakcijama pa se svaka od njih lako može promatrati kao izolirana jedinica s ulazima i izlazima u vidu protokola. Na ovaj način, lako je kreirati lažne ovisnosti te na njima promatrati i testirati ponašanje svake od navedenih komponenti. Jedino komponenta koju je teže testirati je *ViewController*, no budući da je to komponenta koja je usko vezana uz sustav i korisničko sučelje, ona se zapravo niti ne testira detaljno nego onoliko koliko je moguće u pristupu razvijanja programske podrške vođenog testiranjem. Ta komponenta mogla bi se u potpunosti testirati jedino pomoću UI testova, no oni se

obično ne svrstavaju u razvijanje programske podrške vođeno testiranjem, nego se u svrhu povećavanja kvalitete pišu kasnije.

6.4.3. Implementacija *ViewModela*

ViewModel je komponenta u kojoj je sadržana potpuna poslovna logika aplikacije i nužno je da je u potpunosti prilagođena za testiranje odnosno da se svaki tijek i vladanje mogu lako testirati. Kod razvijanja programske podrške vođenog testiranjem, programer je zapravo prisiljen kreirati komponente na ovakav način s obzirom na to da prvo piše testove, no programer si mora predočiti konkretnu implementaciju i prije početka pisanja testova kako bi znao na koji način treba napisati test. Zato je potrebno imati na umu arhitekturu kojoj se teži za vrijeme pisanja testova. Arhitektura koja je odabrana u ovom radu je *MVVM*, a *ViewModel* ima jasno definirane ulaze izlaze i ovisnosti koje su zapravo strukture. Na slici 6.8 prikazan je dio implementacije *ViewModel*-a za zaslon s informacijama o redovima.

```
class QueueScreenViewModel: ViewModel{

    struct Dependecies{
        weak var dataPublisherDelegate: QueueDataPublisher?
        var repository: QueueRepositoryProtocol
    }

    struct Input{
        let tableDataPublisher: ReplaySubject<()>
        let initDataPublisher: ReplaySubject<()>
        let userInteraction: PublishSubject<UserInteractionType>
    }

    struct Output{
        var disposables: [Disposable]
        var tableData: [SectionModel<SectionType, CellModel>]
        var notifyView: ReplaySubject<NotifyViewType>
    }
```

Slika 6.9. *Prikaz struktura unutar ViewModela za zaslon sa stanjem u redovima*

Klasa ovisi o protokolu za repozitorij i *QueueDataPublisher* koji delegira podatke svake tri sekunde s roditeljskog *ViewModela*. Ulazi u klasu su događaji koje emitira *ViewController*, a izlazi su polje podataka potrebnih za prikaz na zaslonu *tableData*, polje *disposable* tipova podataka koje je potrebno predati *ViewControlleru* kako bi se tokovi podataka oslobodili iz memorije nakon što *ViewController* završi s radom i na kraju varijablu *notifyView* koja obavještava *ViewController* o tome što se treba napraviti sljedeće. *ReplaySubject* i *PublishSubject* su specifični tipovi tokova podataka sadržani u bibliotekama RxSwift i RxCocoa koji mogu odašiljati i prihvaćati podatke. Razlika između njih je u tome što *ReplaySubject* ima spremnik u kojemu je spremljen određeni

broj događaja i ako se neka komponenta pretplati na događaje nakon što su se određeni događaji već izvršili, tada će *ReplaySubject* emitirati zadnjih x događanja, pri čemu je x veličina spremnika. Važno je primijetiti da se sve interakcije korisnika šalju kroz jedan tok podataka, ovo je realizirano tako da je svaki događaj zapravo jedan od slučajeva *UserInteractionType* enumeracije. Slučajevi te enumeracije mogu sadržavati određene podatke. Tako na primjer, *ViewController* može obavijestiti *ViewModel* da se dogodila interakcija korisnika čiji je tip „promjena teksta“ i sadrži podatke „novi tekst“. Na slici 6.10 prikazana je enumeracija za tipove korisničkih interakcija.

```
import Foundation
enum UserInteractionType{
    case inputTextChange(to: String)
    case userEnteredNumber(queue: String, number: Int)
}
```

Slika 6.10. *Prikaz enumeracije tipova korisničkih interakcija*

Također svi događaji kojima *ViewModel* obavještava *ViewController* o promjenama koje je potrebno napraviti na korisničkom sučelju, emitiraju se na jednom toku podataka pomoću slučajeva enumeracije *NotifyView*. Na primjer, *ViewModel* može obavijestiti *ViewController* s tipom „pokaži učitavač“ i *bool* vrijednosti false što znači da je potrebno ugasiti učitavač (engl. *loader*). Slika 6.11 prikazuje enumerator obavijesti prema *ViewControlleru*.

```
import Foundation
enum NotifyViewType: Equatable{
    case showErrorState(error: ResponseError)
    case dataHasChanged
    case showLoader(_: Bool)
    case noNumber
    case yourTurn
}
```

Slika 6.11. *Prikaz enumeratora obavijesti za ViewController*

Ovakva implementacija komunikacije između komponenti ne samo da smanjuje veličinu koda, nego i omogućava lakše testiranje redoslijeda kojim su se događaji pojavljivali.

Funkcija koja povezuje ulazne tokove podataka s izlaznim tokovima je funkcija *transfom*. Ona je ključ za uspješnu komunikaciju između *ViewControllera* i *ViewModela*. Poziva ju *ViewController* i predaje ulazne tokove podataka, a prima izlazne tokove podataka *ViewModela*. U polje „disposable“ tipova podataka dodaju se pretplate *ViewModel*-a te se to polje u *ViewControlleru*

oslobađa nakon završetka njegova rada. Na slici 6.12 prikazana je implementacija funkcije *transform*.

```
var input: Input!
var output: Output!
let scheduler: SchedulerType
let dependencies: Dependecies
var filterText = ""
var cachedData: [SectionModel<SectionType, CellModel>] = []

@discardableResult func transform(input: QueueScreenViewModel.Input) -> QueueScreenViewModel.Output {
    let output = Output(disposables: [], tableData: [], notifyView: ReplaySubject<NotifyViewType>.create(bufferSize: 5))
    self.input = input
    self.output = output
    self.output.disposables.append(bindDataDisposable(publisher: input.tableDataPublisher))
    self.output.disposables.append(createUserInteractionDisposable(publisher: input.userInteraction))
    self.output.disposables.append(getInitialDataDisposable(publisher: input.initDataPublisher))
    return output
}
```

Slika 6.12. *Prikaz implementacije funkcije transform*

Kada se dogodi neka interakcija korisnika, *ViewController* pomoću instance Input strukture šalje događaj na određeni ulazni tok podataka, *ViewModel* tada izvrši potrebnu radnju i rezultat šalje pomoću instance Output strukture na neki izlazni tok podataka i na kraju *ViewController* pokaže korisniku rezultat.

6.4.4. Implementacija ViewControllera

Iz perspektive *ViewControllera* bitno je istaknuti pozivanje metode *transform ViewModel*-a kao i kreiranje vreće za „disposable“ tipove koja služi za oslobađanje resursa nakon završetka tokova podatka. Također sve UI komponente pisane su u kodu i nije korišten pomoćni alat pomoću kojega je moguće povući i ispustiti (engl. *drag and drop*) pojedine UI komponente. Na ovaj način postignuta je dodatna fleksibilnost budući da korištenjem pomoćnog alata Xcode generira kod za komponente UI-a i to radi u ne baš čitljivom xml formatu, pa je prilikom spajanja koda, ako se radi u timu, znatno otežano rješavanje eventualnih konflikata. Na slici 6.13 prikazana je implementacija UI komponente pisanjem koda.

```
class PersonalInfoViewController: UIViewController, ShowErrorView{
    let viewModel: PersonalInfoViewModel
    let disposeBag = DisposeBag()

    //MARK: UI
    let rowNumberInputView: RowNumberInputView = {
        let rowNumberView = RowNumberInputView()
        rowNumberView.translatesAutoresizingMaskIntoConstraints = false
        return rowNumberView
    }()
```

Slika 6.13. *Prikaz implementacije UI komponente pisanjem swift koda*

Kada se UI komponente implementiraju na ovakav način, osim kreiranja komponente potrebno je komponentu dodati u hijerarhiju UI komponenti, također, potrebno je napisati kod koji komponentu smješta na određeno mjesto na zaslonu. U iOS programiranju komponente se smještaju na zaslonu tako da se napišu međusobne ovisnosti između komponenti te sustav na osnovu njih može izračunati kako zaslon treba izgledati. Tehnologija koja to omogućuje naziva se „Auto-Layout“ a odnosi između komponenti postavljaju se pomoću tako zvanih „ograničenja“ (engl. *constraints*). Kod za pisanje ograničenja često je robustan i redundantan pa postoje određene biblioteke koje mogu pomoći da se taj kod svede na minimum postižući isto ponašanje. U ovoj aplikaciji za pisanje ograničenja koristi se biblioteka *SnapKit*. Slika 6.14 prikazuje dodavanje UI komponenti u hijerarhiju elemenata, te postavljanje ograničenja UI komponenti pomoću biblioteke *SnapKit*.

```
func setupUI(){
    self.view.backgroundColor = .white
    self.view.addSubview(rowNumberInputView)
    self.view.addSubview(resultView)
    self.view.addSubview(imageView)
    setupConstraints()
}

func setupConstraints(){
    rowNumberInputView.snp.makeConstraints { (maker) in
        maker.leading.trailing.equalToSuperview()
        maker.top.equalToSuperview().offset(navigationController?.navigationBar.frame.height ?? 60)
    }

    imageView.snp.makeConstraints { (maker) in
        maker.centerX.equalToSuperview()
        maker.top.equalTo(rowNumberInputView.snp.bottom).offset(23)
        maker.height.width.equalTo(100)
    }

    resultView.snp.makeConstraints { (maker) in
        maker.top.equalTo(imageView.snp.bottom).offset(20)
        maker.leading.trailing.equalToSuperview().inset(27)
    }
}
```

Slika 6.14. *Prikaz dodavanja UI elementa u hijerarhiju elemenata i postavljanja ograničenja*

Funkcija *ViewControllera bindViewModel* zaslužna je za povezivanje *ViewControllera* i *ViewModela* pozivajući funkciju *transform*. Izlaz *ViewModela* sprema se u varijablu, svaki *disposable*“ tip iz polja se stavi u vreću *disposable* tipova koja služi za oslobađanje memorije nakon završetka rada *ViewControllera*. Slika 6.15 prikazuje povezivanje događaja *ViewControllera* i *ViewModela*

```
func bindViewModel(){
    let output = self.viewModel.transform(input: PersonalInfoViewModel.Input(dataPublisher: ReplaySubject<()>.create(bufferSize: 1),
        initDataPublisher: ReplaySubject<()>.create(bufferSize: 1), userInteraction: PublishSubject<UserInteractionType>()))
    output.disposables.forEach { (disposable) in
        disposable.disposed(by: disposeBag)
    }
    subscribeToEvents(of: output)
}
```

Slika 6.15. *Prikaz povezivanja događaja ViewControllera i ViewModela*

Na kraju metode *bindViewModel* poziva se metoda *subscribeToEvents* kojoj se predaje output kako bi se *ViewController* pretplatio na događaje koje će *ViewModel* emitirati. Na slici 6.16 prikazan je način na koji *ViewController* sluša događaje koje emitira *ViewModel* u toku podataka *notifyView*. Također prikazan je način na koji se pomoću RxSwifta mogu mijenjati niti na kojima će se izvršavati određeni dio koda, a to je pomoću operatora *subscribeOn* i *observeOn*.

```
func subscribeToEvents(of output: PersonalInfoViewModel.Output){
    output.notifyView
        .observeOn(MainScheduler.instance)
        .subscribeOn(ConcurrentDispatchQueueScheduler.init(qos: .background))
        .subscribe(onNext: {[unowned self] (notifyType) in
            switch notifyType{
            case .dataHasChanged:
                self.handleDataChange()
            case .showErrorState(let error):
                self.showErrorView(true, errorText: error.errorText)
            case .noNumber:
                self.noNumber()
            default:
                return
            }
        }).disposed(by: disposeBag)
    viewModel.input.dataPublisher.onNext(())
    viewModel.input.initDataPublisher.onNext(())
}
```

Slika 6.16. *Pretplaćivanja ViewControllera na tok izlaza ViewModela*

SubsribeOn operator govori na kojoj niti izvršavanja će se odvijati manipulacija događajima, dok operator *observeOn* govori na kojoj niti će biti emitirani rezultati manipulacije događajima. U ovom slučaju, događaji se izvršavaju na pozadinskoj niti kako se ne bi blokiralo korisničko sučelje, a rezultati događaja se manifestiraju na glavnoj niti izvođenja programa.

6.4.5. Implementacija kontinuiranog dohvaćanja podataka

Ova funkcionalnost je najvažniji dio aplikacije i sa sobom nosi određene izazove. Potrebno je kontinuirano slati zahtjeve, čiji će odgovor dolaziti kasnije i u trenutku kada se pojavi potrebno je osvježiti korisničko sučelje. Budući da podatci trebaju biti prikazana na dva zaslona, zaslonu za prikaz stanja redova i zaslonu za prikaz informacija o pojedinom redu, koji se trebaju dinamično osvježavati novim sadržajem, potrebno je osmisliti način kako oba zaslona obavijestiti o dolasku novih podataka a da se pri tome misli na uštedu resursa i sa strane mobilne aplikacije ali i sa strane poslužitelja. Veliki broj poslanih zahtjeva mogao bi preopteretiti poslužitelj, a i usporiti aplikaciju. Umjesto da svaki zaslon šalje zahtjeve svake tri sekunde, ovo je najbolje realizirati da se pozivi i dohvaćanja podataka odvijaju na jednome mjestu, te se nakon svakog dohvaćanja podataka obavijeste potrebni zasloni o tome. Također, potrebno je napraviti rješenje koje bi odgovaralo korištenoj arhitekturi i bilo lako za testiranje.

Kao rješenje u aplikaciji za pregled preostalog broja pacijenata u redu, kreiran je protokol koji opisuje potrebno vladanje objekta koji ga implementira. Objekt je zadužen za slanje i dohvaćanje podataka i emitiranje dohvaćenih podataka u jednom toku podataka. Slika 6.17 prikazuje protokol za dohvaćanje podataka i proširenja metode za pretplatu na tok podataka.

```
protocol QueueRequestWorker: class{
    var requestPublisher: BehaviorRelay<Result<QueueResponse,ResponseError>> { get }
    var timer: Observable<Int> { get }
    func subscribeToDisposable(action: @escaping (()->Observable<Result<QueueResponse,ResponseError>>)) -> Disposable
}

extension QueueRequestWorker{
    func subscribeToDisposable(action: @escaping (()->Observable<Result<QueueResponse,ResponseError>>)) -> Disposable{
        return timer.flatMapLatest { (_) -> Observable<Result<QueueResponse,ResponseError>> in
            return action().catchError({_ in return Observable.just(Result<QueueResponse, ResponseError>.failure(.networkError))})
            }
            .bind(to: requestPublisher)
    }
}
```

Slika 6.17. *Prikaz protokola za dohvaćanje podataka i proširenja metode za pretplatu*

Klasa koja implementira ovaj protokol mora imati instancu releja za emitiranje podatka, ovo je tok događaja na koji će se pretplatiti svatko tko želi biti obaviješten o dohvaćanju novih podataka. Na ovom primjeru, vidljiva je ranije spomenutu kombinaciju operatora ^i operatora interval. Operator interval primjenjuje se na *timer* toku podataka i emitira konstante cjelobrojne vrijednosti a operator *flatMap* u ovom slučaju pretvara te vrijednosti u određenu akciju koju korisnik klase preda funkciji. Klase koje ovise o ovoj implementaciji neće ovisiti o konkretnom tipu, nego o apstrakciji budući da se radi o protokolu, te će tako i testiranje biti olakšano.

6.4.6. Implementacija protokola za dohvaćanje podataka

Objekt koji implementira ovaj protokol, zahvaljujući proširenju protokola ne treba implementirati funkciju za pretplatu. Slika 6.18 prikazuje implementaciju protokola za kontinuirano dohvaćanje podataka .

```
class QueueRequestWorkerImpl: QueueRequestWorker{
    var requestPublisher: BehaviorRelay<Result<QueueResponse, ResponseError>>

    typealias ResponseType = QueueResponse
    var timer: Observable<Int>

    init(scheduler: SchedulerType = ConcurrentDispatchQueueScheduler.init(qos: .background), timeInterval: Int = 2500) {
        timer = Observable<Int>.interval(RxTimeInterval.milliseconds(timeInterval), scheduler: scheduler)
        requestPublisher = BehaviorRelay<Result<QueueResponse, ResponseError>>(value: Result.success(QueueResponse(queueStatuses: [])))
    }
}
```

Slika 6.18. *Implementacija protokola za kontinuirano dohvaćanje podataka*

Korištenje objekta za dohvaćanje podataka svodi se na pozivanje metode protokola kojom se preplaćuje na tok podataka i predavanje akcije koju je potrebno izvršavati u određenim vremenskim intervalima. Bitno je samo da akcija koju se predaje u obliku *closure* funkcije, vraća definirani povratni tip. Dakle pomoću ovakve implementacije moguće je dohvaćati podatke iz različitih izvora kao na primjer baze podatka, memorije uređaja ili interneta. Slika 6.19 prikazuje preplate na tok podataka unutar *HomeViewModel* klase.

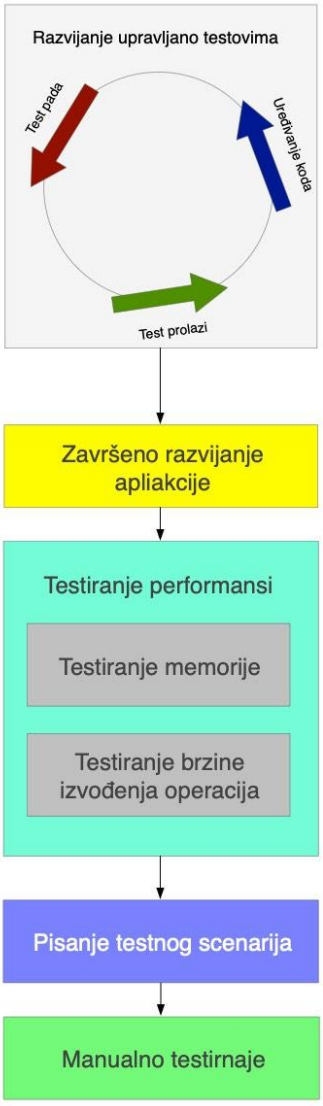
```
func createWorkerDisposable() -> Disposable{
    return self.dependencies.worker.subscribeToDisposable {[weak self] () -> Observable<Result<QueueResponse, ResponseError>> in
        guard let safeSelf = self else{return Observable.just(Result<QueueResponse, ResponseError>.failure(ResponseError.networkError))}
        return safeSelf.dependencies.repository.getQueueResponseObservable()
        }
}

func subscribeOnWorker() -> Disposable{
    return dependencies.worker.requestPublisher.bind(to: dataPublisher)
}
```

Slika 6.19. *Prikaz pretplate na tok podataka unutar HomeViewModel klase*

U aplikaciji za pregled rasporeda pacijenata u redu, pretplata na tok događaja koji se kontinuirano dohvaćaju napravljena je u klasi *HomeViewModel* te se zatim šalje preko *HomeViewCoordinatora* njegovoj „djeci“ koordinatorima koji podatke predaju svojim *ViewModel* klasama. Na ovaj način dohvaćanje podataka radi se samo na jednom mjestu, ovisnosti su „ubrizgane“ principom ubrizgavanja ovisnosti (engl. *dependency injection*) dakle poštuje se princip inverzije ovisnosti (D u SOLID principima) i testiranje je lako kao što će biti prikazano u nastavku.

7. TESTIRANJE APLIKACIJE ZA PREGLED PREOSTALOG BROJA PACIJENATA U REDOMATSKOM REDU

Aplikacija je testirana tijekom cijeloga razvoja i sastoji se od 72 jedinična testa, budući da je rađena procesom razvijanja programske podrške vođenog testiranjem, gotovo svi elementi aplikacije su testirani jediničnim testiranjem (engl. *unit test*), neke komponente testirane su i integracijskim testiranjem, testirane su performanse aplikacije pomoću Xcode „instruments“ alata te je na kraju provedeno ručno testiranje aplikacije. U ovom poglavlju bit će prikazani samo neki od testova iz aplikacije koji su pogodni za lakše objašnjavanje bitnih koncepata. Na slici 7.1 prikazan je dijagram toka testiranja u aplikaciji.



Slika 7.1. *Prikaz dijagrama toka testiranja aplikacije*

7.1. Testovi jedinica unutar aplikacije

Programiranje aplikacije započinje pisanjem testa jedinice (engl. *unit test*), a način na koji se oni pišu najbolje je prikazati primjerom. Za testiranje iOS aplikacija u sklopu alata Xcode dolazi i biblioteka XCTest za pisanje testova jedinica. Glavna klasa za definiranje testnih slučajeva i testnih funkcija naziva se *XCTestCase* i u njoj je potrebno pisati testove jedinica. Na slici 7.2 prikazan je primjer jednog jednostavnog testa jedinice.

```
func testHomeCoordinator_init_firstChildCoordinatorIsOfRightType() {
    let sut = makeSUT()

    sut.start()

    XCTAssertTrue(sut.childCoordinators.first is QueuesScreenCoordinator, "first child coordinator is of wrong type")
}
```

Slika 7.2. *Primjer Unit testa unutar aplikacije*

Svaka testna funkcija mora započinjati ključnom riječi „test“ kako bi ju Xcode mogao prepoznati, a vrijednosti ili izrazi koje želimo provjeravati pišu se unutar *Test Assertions* metoda. Postoji više vrsta *Test Assertion* funkcija koje provjeravaju različite stvari a na slici 7.1. prikazana je funkcija koja provjerava dali je *bool* rezultat uvjeta koji joj je predan jednak istini (engl. *true*). Iako primjer iznad možda djeluje jednostavno, puno stvari se zapravo događa u tom dijelu koda, počevši od imenovanja testa, pokretanja metode za inicijalizaciju, pa do kreiranja objekta koji se testira i na kraju ispitivanja uvjeta.

7.1.1. Imenovanje testa

Testovi se obično imenuju tako da se prvo napiše komponenta koja se testira, u ovom slučaju je to *HomeCoordinator*, zatim se napiše ponašanje koje uzrokuje vladanje koje želimo testirati a to je na prethodnom primjeru inicijalizacija objekta, na kraju se napiše očekivani rezultat to je u ovom slučaju da je koordinator dijete pravog tipa.

7.1.2. Struktura tijela testa

Tijelo unutar testa podijeljeno je u tri dijela, prema konvenciji - dano, kada, onda (engl. *Given, when, then*) prvi dio predstavlja objekte, varijable i ostale komponente koje su dane testu na korištenje. U drugom djelu implementiraju se određeni događaji koji dovode do stanja objekta koje želimo testirati, a u trećem dijelu pišu se provjere. Na primjeru koda sa slike 7.1. pomoću funkcije *makeSUT*, kreira se objekt koji testiramo, *sut* je skraćenica za sustav koji se testira (engl. *system*

under test). Na taj se način simulira cijeli životni ciklus objekta i spriječeno je dijeljenje stanja objekta između više testova. Ovakvom implementacijom poštuju se dobre prakse testiranja koje su opisane u poglavlju 5. Budući da je objekt koji se testira u ovom primjeru koordinator, u drugom dijelu testa pokrećemo njegovu funkciju *start*, kako bi postavili njegov *NavigationController* na stog i inicijalizirali „djecu“ koordinatore. U trećem dijelu, provjeravamo dali je objekt prvog koordinatora djeteta pravog tipa i time je test završen.

7.2. RxTest i funkcija za kreiranje objekta koji se testira

Testiranje reaktivnog koda dosta se razlikuje od uobičajenog budući da je potrebno asinkrono ponašanje pretvoriti u sinkrono jer se testovi sinkrono izvode. Biblioteka RxTest služi za rješavanje ovih problema uvođenjem *TestScheduler* klase. *TestScheduler* ima mogućnosti kreiranja umjetnih objekata na koje možemo povezati tokove podataka i onda na njima promatrati i testirati ponašanje koda. Svaka klasa koja je pisana na reaktivan način, treba osigurati način da joj se preda testni raspoređivač (engl. *scheduler*) ako ju je potrebno testirati. Najbolji način da se to postigne je pomoću ubrizgavanja ovisnosti predati instancu raspoređivača kojim će se promatrati tokovi podataka. Slika 7.3 prikazuje ubrizgavanje raspoređivača u *ViewModel*.

```
init(scheduler: SchedulerType = ConcurrentDispatchQueueScheduler.init(qos: .background), dependencies: Dependecies) {
    self.dependencies = dependencies
    self.scheduler = scheduler
}
```

Slika 7.3. *Prikaz ubrizgavanja raspoređivača u ViewModel*

Također, moguće je postaviti podrazumijevanu vrijednost za raspoređivač što je u ovom slučaju pozadinski raspoređivač, na ovaj način izbjegavamo potrebu za predavanjem objekta raspoređivača prilikom inicijalizacije *ViewModela* u produkcijskom kodu, no imamo mogućnost predati *TestScheduler* objektu *ViewModela* u testnom kodu. Slika 7.4 prikazuje Prikaz globalne varijable testne klase i funkcije *setUp*.

```
class QueueScreenViewModelTests: XCTestCase {
    var scheduler: TestScheduler!
    var disposeBag: DisposeBag!
    var publisherSubjectMock: ReplaySubject<Result<QueueResponse, ResponseError>>!
    var dataPublisherMock: QueueDataPublisher!
    var mockDataObserver: TestableObserver<Result<QueueResponse, ResponseError>>!

    override func setUp() {
        scheduler = TestScheduler(initialClock: 0)
        disposeBag = DisposeBag()
        publisherSubjectMock = ReplaySubject<Result<QueueResponse, ResponseError>>.create(bufferSize: 1)
        dataPublisherMock = MockManager.getMockDataPublisher(publisher: publisherSubjectMock)
        mockDataObserver = scheduler.createObserver(Result<QueueResponse,ResponseError>.self)
        scheduler.start()
    }
```

Slika 7.4. *Prikaz globalnih varijabli testne klase i funkcije setUp*

U metodi *setUp* moguće je postaviti inicijalna stanja globalnih varijabli, a ona se automatski poziva prije svake testne funkcije, ovime se izbjegava dijeljenje zajedničkih stanja varijabli između više testova. Također postoji i funkcija *tearDown* koja se automatski poziva nakon izvršavanja svake testne funkcije, ona služi za čišćenje stanja varijabli ali u ovom radu, primijenjena je na zanimljiv način koji će biti opisan u nastavku. Na slici 7.5. prikazan je dio koda za kreiranje lažnih promatrača, na koje se tokovi događaja mogu povezati i na kojima se onda promatra ponašanje i događaji tog toka.

```
//MARK: helpers
func makeSUT(with dataPublisherMock: QueueDataPublisher?) -> QueueScreenViewModel{
    let sut = QueueScreenViewModel(scheduler: scheduler, dependencies: QueueScreenViewModel.Dependecies(dataPublisherDelegate:
        dataPublisherMock, repository: MockManager.createQueueRepositoryMock()))
    self.weakSUT = sut
    let output = sut.transform(input: QueueScreenViewModel.Input(tableDataPublisher: ReplaySubject<()>.create(bufferSize: 1),
        initDataPublisher: ReplaySubject<()>.create(bufferSize: 1), userInteraction: PublishSubject<UserInteractionType>()))

    output.disposables.forEach { (disposable) in
        disposable.disposed(by: disposeBag)
    }
    sut.dependencies.dataPublisherDelegate?.dataPublisher.bind(to: mockDataObserver).disposed(by: disposeBag)
    sut.input.initDataPublisher.onNext(())

    return sut
}
```

Slika 7.5. *Prikaz funkcije za kreiranja objekta za testiranje*

U funkciji za kreiranje objekta za testiranje, potrebno je inicijalizirati objekt, predati mu testni raspoređivač i lažne instance (engl. *mock*) ostalih ovisnosti kako bi se objekt u potpunosti izolirao a nad lažnim instancama vršilo testiranje njima pripadnog ponašanja. Na slici 7.3. također je vidljiv način na koji je testni raspoređivač predan instanci objekta za testiranje. U funkciji je prikazano kreiranje umjetnog promatrača *mockDataOvposlužitelj* pomoću *TestScheduler* objekta koji se povezuje na *dataPublisher* delegat i sprema događaje koje on emitira. Varijabla *weakSUT* koju sadrži objekt koji testiramo, služi za provjeru dali se nakon svih operacija objekt uspješno

oslobodio iz memorije, ova varijabla ima slabu (engl. „weak“) referencu na objekt koji testiramo i nakon svakog testa trebala bi imati vrijednost „nil“. Ako se objekt uspješno uklonio iz memorije znači da nemamo curenja memorije (engl. *memory leak*) za vrijeme životnog ciklusa objekta u izolaciji. Budući da nakon svakog testa varijabla *weakSUT* treba imati vrijednost *nil*, ovo se provjerava u ranije spomenutoj *tearDown* funkciji čiji je prikaz na slici 7.6.

```
override func tearDown() {
    XCTAssertNil(weakSUT)
}

weak var weakSUT: QueueScreenViewModel?
```

Slika 7.6. *Primjer provjere dali je testirani objekt nakon testa uspješno uklonjen iz memorije*

7.3. Primjer testiranja reaktivnog koda

Testovi pisani za reaktivni kod ne razlikuju se strukturom niti imenovanjem onima za sinkroni kod. Kod njihovog pisanja programer poštuje ista pravila i jedina razlika je u tome što koristi dodatni alat kako bi asinkrono ponašanje mogao promatrati kao sinkrono. Slika 7.7 prikazuje primjer testiranja reaktivnog koda aplikacije.

```
func testViewModel_whenPublisherIsNil_showErrorMessage(){
    let sut = makeSUT(with: nil)
    let notifyViewMock = scheduler.createObserver(NotifyViewType.self)
    sut.output.notifyView.bind(to: notifyViewMock).disposed(by: disposeBag)

    sut.input.tableDataPublisher.onNext(())
    publisherSubjectMock.onNext(Result<QueueResponse,ResponseError>.success(QueueResponse.init(queueStatuses: [])))

    XCTAssertEqual(notifyViewMock.events.last!.value.element, NotifyViewType.showErrorState(error: .publisherDelegateIsNil))
}
```

Slika 7.7. *Primjer testiranja reaktivnog koda aplikacije*

Zahvaljujući dobroj strukturi i imenovanju, iz koda sa slike 7.5. lako je iščitati funkcionalnost i svrhu ovoga testa. Testira se prikaz poruke pogreške u slučaju kada je *dataPublisherDelegate* jednak *nil*. Najprije se kreira objekt za testiranje, predaje mu se *nil* vrijednost kao *dataPublisherDelegate*, kreira se lažni tok događaja pomoću *TestSchedulera* i poveže se s tokom događaja za obavijesti prema *ViewConrtolleru*, pokrene se emitiranje podataka i na kraju se na lažnom objektu promatraču, koji snima događaje toka podataka za obavijesti prema *ViewControlleru*, provjerava je li uspješno prikazana poruka o pogrešci.

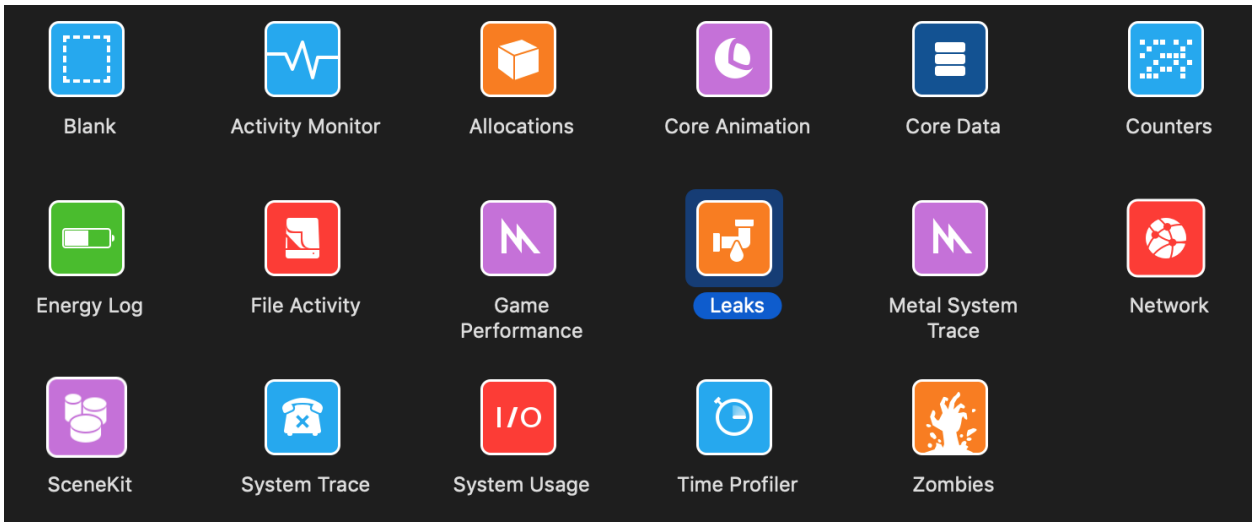
Kako bi se aplikacija uspješno izradila pristupom razvijanja programske podrške vođenog testiranjem, važno je imati dobar uvid u zahtjeve i funkcionalnosti koje je potrebno implementirati

i moći predvidjeti implementaciju prije nego što je to obično potrebno, a te sposobnosti dolaze iskustvom i vježbom. Možda u početku, ovakav način razvijanja aplikacija djeluje komplicirano no on ima višestruke prednosti kao što je navedeno u poglavlju 5.5. Isplati se uložiti dodatni napor u početku kako bi na kraju rezultat bio nesumnjivo bolji.

7.4. Testiranje performansi aplikacije

Performanse aplikacije testirane su pomoću alata Xcode Instruments koji programerima daje uvid u stanje memorije, opterećenja procesora i ostale potrebne informacije. Ovaj alat, od velike je pomoći programerima i jako je dobro realiziran, za razliku od alata za testiranje performansi nekih drugih operacijskih sustava, iOS je sustav koji je moguće precizno testirati i tako maksimalno optimizirati aplikaciju.

Postoji više načina testiranja performansi koje nudi Xcode instruments alat, a u ovoj aplikaciji testirano je curenje memorije pomoću alata *Leaks* i brzine izvršavanja operacija pomoću alata *Time Profiler*. Slika 7.7 prikazuje alate za testiranje performansi unutar xCode alata.



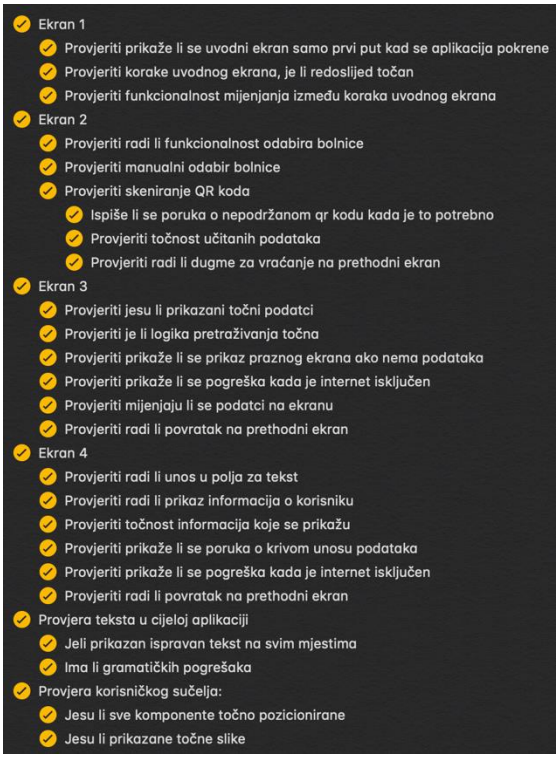
Slika 7.7. *Prikaz alata za testiranje performansi unutar xCode alata*

Alat *Leaks* daje uvid u generalno korištenje memorije, provjerava curenja memorije i daje statistiku alokacije objekata, te povijest adresa memorije za sve alocirane objekte kao i one koji nisu uklonjeni iz memorije nakon korištenja odnosno objekata koji uzrokuju curenje memorije. Korištenjem ovoga alata u aplikaciji za pregled preostalog broja pacijenata, prepoznata su mjesta na kojima je dolazilo do curenja memorije. Nakon ispravka koda, curenja memorije su uklonjena i osigurana je stabilnost.

Također, testirano je vrijeme potrebno da se izvrše određene operacije pomoću alata *Time Profiler*. Ovaj alat, snima operacije koje se odvijaju u aplikaciji i daje uvid o njihovom trajanju. Alat prikazuje ukupno korištenje procesora u određenom trenutku kao i opterećenost pojedinih niti izvršavanja.

7.5. Ručno testiranje i testni scenarij

Nakon što je programer zadovoljan s implementacijom, sljedeća faza u procesu izrade aplikacije je testiranje funkcionalnosti „gotovog“ proizvoda. Za ovakvo testiranje obično postoje posebni odjeli u poduzećima koji se bave osiguravanjem kvalitete proizvoda. U aplikaciji za pregled preostalog broja pacijenata, provedeno je ručno testiranje po testnom scenariju koji je prethodno izrađen. Slika 7.10 prikazuje testni scenarij izrađen za testiranje aplikacije.



Slika 7.10. *Prikaz testnog scenarija*

Prvi korak kod ručnog testiranja je napisati testni scenarij, odnosno stvari koje je potrebno testirati u aplikaciji. Potrebno je detaljno pročitati dokumentaciju projekta, proučiti sve funkcionalnosti i korake korisnika kojima se izvršavaju određene akcije. Zatim je potrebno proučiti dizajn aplikacije i potencijalne radnje koje bi mogle dovesti do problema. Testiranje treba biti strogo i treba mi se pristupati kritički. Dobro je da aplikaciju ručno testira osoba koja ju nije programirala upravo iz ranije navedenih razloga, to je u većini slučajeva zadovoljeno budući da obično postoji poseban

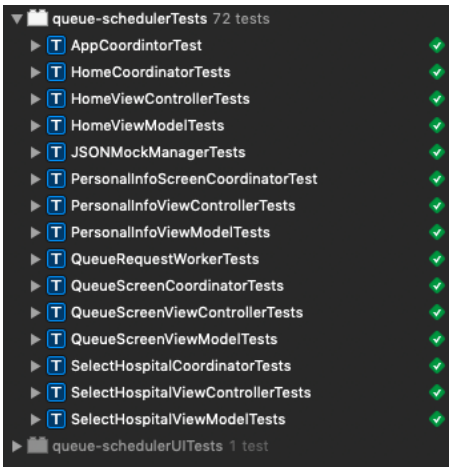
odjel za testiranje aplikacija. Na primjeru testnog scenarija sa slike 7.1. funkcionalnosti koje se testiraju podijeljene su prema zaslonima na kojima se nalaze. Također, posljednja dva testna slučaja potrebno je provesti na svim zaslonima. Aplikacija se testira tako da se instalira nova inačica aplikacije na uređaj te prati napisani testni scenarij. To je potrebno napraviti na svim uređajima na kojima je aplikacija podržana, u ovom slučaju, svi uređaji koji koriste iOS verziju 10 i noviji. Proces testiranja nakon programiranja osigurava kvalitetu proizvoda. Danas je nezamislivo objavit aplikaciju koja nije prošla kroz nekoliko iteracija testiranja. Testiranjem performansi dobiva se uvid u ponašanje i brzinu aplikacije na svim podržanim uređajima i moguće je uvidjeti određene poteškoće koje nisu na prvi pogled vidljive i možda nisu povezane uz logiku aplikacije. Pozitivno korisničko iskustvo je cilj kojemu svaki proizvođač teži, a kvaliteta aplikacije je najbolji način da se taj cilj ostvari.

8. REZULTATI TESTIRANJA I GRAFIČKI PRIKAZ SVIH ZASLONA APLIKACIJE

U aplikaciji za pregled preostalog broja pacijenata testiranje se odvijalo nad svakim dijelom koda aplikacije jer su se prvo pisali testovi zahvaljujući pristupu razvijanja programske podrške vođenog testovima. Testirane su performanse aplikacije i na kraju je provedeno ručno testiranje prema testnom scenariju. Rezultati testiranja su odlični i aplikacija radi u skladu s očekivanjima.

8.1. Rezultati razvijanja programske podrške vođenog testiranjem

Aplikacija sadrži 72 testa jedinica, a s obzirom na kompleksnost aplikacije i njezinu veličinu ovaj broj je zadovoljavajući. Treba uzeti u obzir da se u nekim slučajevima radilo više provjera unutar jedne testne funkcije te da se neki dijelovi koda ne testiraju ili su već testirani u sklopu neke vanjske biblioteke. Slika 8.1 prikazuje rezultate i broj jediničnih testova provedenih u aplikaciji.



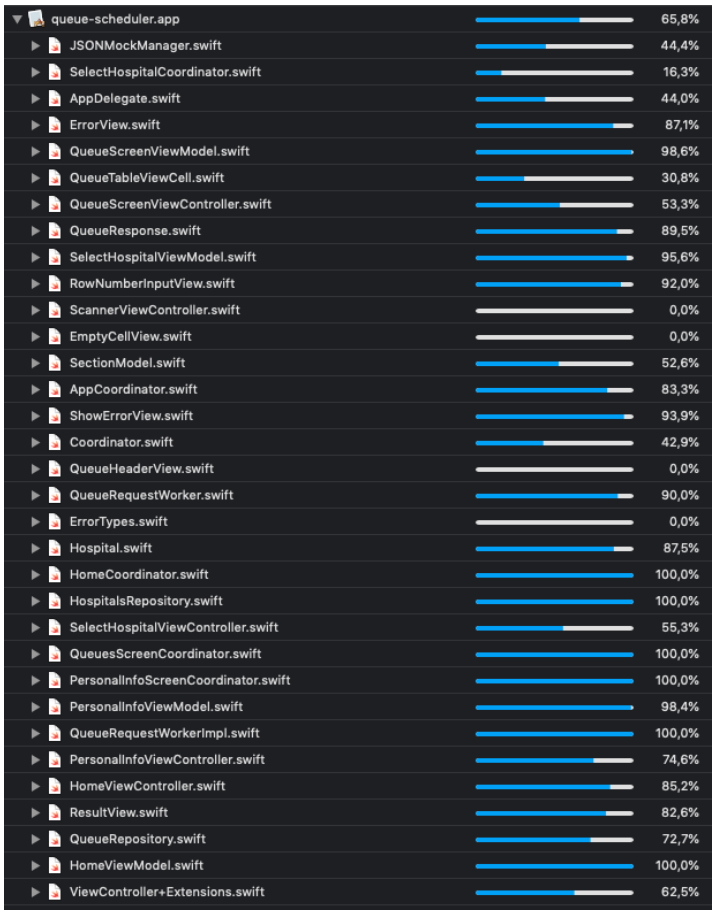
Slika 8.1. *Prikaz broja testova i testnih datoteka aplikacije*

Pokrivenost koda testovima je 65.8% a razlog tome je što komponente korisničkog sučelja nisu testiranje kao ni implementacija usko povezana za sustav. Pokrivenost testova računa se tako da se prate tokovi programa kroz koje je sustav prošao prilikom pokretanja testova, pa ovaj podatak ne govori o tome koliko je zapravo aplikacija dobro testirana, nego samo koliki postotak koda je „dotaknut“ testovima. Na slici 8.2 prikazana je ukupna pokrivenost koda aplikacije.

Slika 8.2. *Prikaz ukupne pokrivenosti koda testovima*

Nešto bolji uvid u kvalitetu provedenog testiranja možemo dobit ako promatramo pojedinačne klase i funkcije koje su pokrivene testovima. Na ovaj način možemo eliminirati stvari koje nije potrebno testirati i dobiti nešto realniju sliku o kvaliteti, no i dalje je ovaj broj praktički beskoristan i nije temelj za bilo kakve prosudbe.

Slika 8.3 prikazuje kako klase u kojima je sadržan najveći dio logike a to su *ViewModel*-i ujedno i najbolje pokrivene testovima, ovo je logično budući da je cilj arhitekture bila mogućnost u potpunosti testirati *ViewModel*.



Slika 8.3. *Prikaz pokrivenosti koda po datotekama programa*

Datoteke koje su slabije pokrivene testovima su one koje sadrže „čisti“ UI kod i one u kojima su sadržani samo modeli podataka, dakle strukture, enumeratori i ostali. Tako na primjer, kod unutar datoteke *ErrorTypes* uopće nije pokriven testovima, no to je i očekivano budući da je ovo datoteka koja sadrži samo jedan enumeratori.

Dakle pokrivenost koda nije pretjerano značajna informacija u ovom kontekstu i ovakvom arhitekturom - no postoje slučajevi u kojima može biti izrazito korisna. Na primjer, ako prilikom razvijanja programske podrške vođenog testiranjem primijetimo da neki dio koda nije pokriven testovima, ovo nam može biti signal da nismo provjerili sve slučajeve koje je potrebno testirati. Često su dijelovi koda koji nisu pokriveni testovima zapravo stanja u koja program nikada ne bi trebao doći no možda su zadržana zbog potencijalnog razvijanja aplikacije i sigurnosti u budućnosti. Dio koda koji aplikacija nikada ne pokreće se naziva „mrtvi kod“. Na slici 8.4. prikazan je primjer mrtvoga koda aplikacije za pregled stanja pacijenata.

```
func createUserInteractionDisposable(publisher: PublishSubject<UserInteractionType>) -> Disposable{
    return publisher.subscribe(onNext: {[unowned self] (interactionType) in
        switch interactionType{
        case .inputTextChange(let text):
            self.filterText = text
            self.handleSections(for: .success(self.cachedData))
        default:
            return
        }
    })
}
```

Slika 8.4. *Primjer „mrtvog koda“*

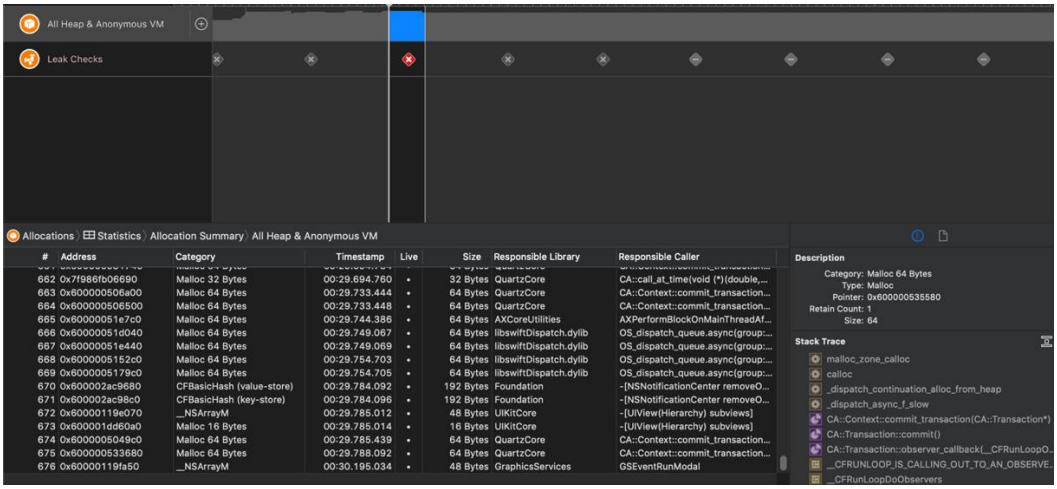
Enumerator *UserInteractionType* sastoji se od dva tipa interakcije korisnika, no na zaslonu za prikaz stanja u svim redovima koristi se samo jedan tip interakcije korisnika a to je promjena prilikom pretraživanja. Budući da se koristi isti enumerator za interakciju korisnika na ovom zaslonu kao i na zaslonu za specifične informacije o korisniku (što možda i nije najbolje rješenje), ovaj dio koda nikada neće biti izvršen, barem ne dok se ne uvede još neka moguća interakcija korisnika na ovom zaslonu. Zato je ovaj kod „mrtav kod“ i potrebno ga je prepraviti. Ovo je samo jedan od primjera kako testovi i pokrivenost testovima mogu pomoći prilikom razvijanja programske podrške vođenog testiranjem.

Rezultati testiranja ne mogu se zorno predočiti brojkama, oni se očituju za vrijeme razvijanja programske podrške vođenog testiranjem i u budućnosti nakon što je aplikacija napravljena i dodaju se nove funkcionalnosti. Dobro napisani testovi služe kao dokumentacija i pomažu kod uključivanja novih programera u izradu projekta, budući da je jasno što se određenim dijelom koda željelo postići. Također, testovi daju veliku sigurnost i samopouzdanje programeru, lako je dodavati nove izmjene, programeri su produktivniji i zadovoljniji te su pod manjim stresom budući da je smanjena vjerojatnost nepredviđenih situacija. Kod pristupa razvijanja programske podrške vođenog testiranjem često se dogodi situacija da testovi ukažu na neku nelogičnost znatno prije nego što je funkcionalnost implementirana, ovo daje mogućnost da se o problemu pomnije razmisli prije nego što se „izgubi“ u implementaciji i na kraju nerijetko obriše cijela funkcionalnost. Razdvajanje funkcionalnosti u male cjeline olakšava način na koji se nosi s kompleksnim implementacijama i povećava kvalitetu budući da su svi napori usmjereni na to da se maleni dio nekog kompleksnog vladanja napravi što kvalitetnije.

8.2. Rezultati testiranja performansi i ručnog testiranja

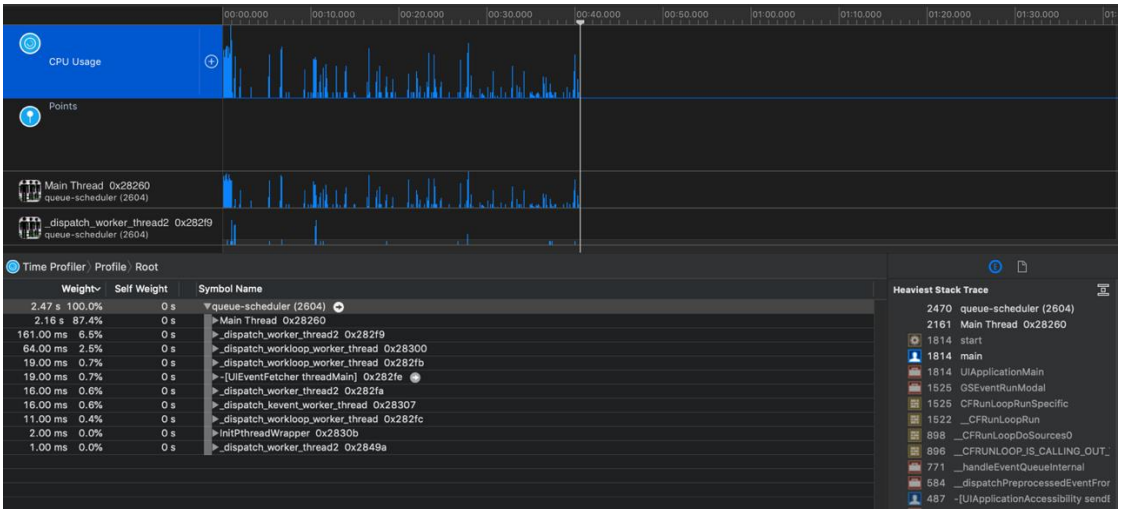
Testiranje performansi pomaže da se utvrde određene nepravilnosti kada je u pitanju brzina i fluidnost aplikacije, no i potencijalni problemi u logici kao što su beskonačne petlje i „čvrste“

(engl. *strong*) reference na objekte, koje onda uzrokuju curenje memorije. Na slici 8.5 prikazani su rezultati testiranja memorije aplikacije pomoću alata Leaks.



Slika 8.5. *Prikaz rezultata testiranja performansi aplikacije pomoću alata Leaks*

Na slici 8.6 prikazani su rezultati testiranja brzine izvođenja operacija aplikacije pomoću programskog alata *Time Profiler*.



Slika 8.6. *Prikaz rezultata brzine izvođenja operacija pomoću alata Time Profiler*

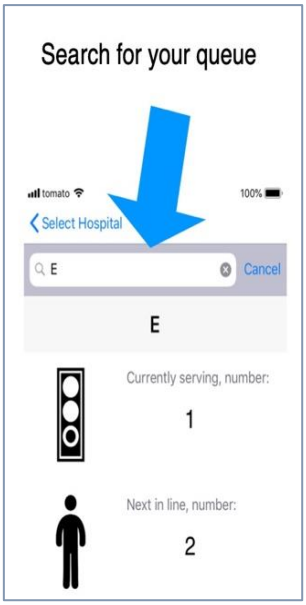
Na slici 8.6. vidljivo je da se sve operacije unutar aplikacije za pregled broja pacijenata odvijaju brzo. Jedino je za vrijeme pokretanja aplikacije vrijeme određenih operacija nešto dulje, no to je uzrokovano operacijskim sustavom i na to programer ne može utjecati.

U aplikaciji za pregled rasporeda pacijenata nisu uočeni problemi prilikom testiranja performansi, no radi se o aplikaciji s malim brojem funkcionalnosti. Budući da aplikacija ima potencijal da naraste na nešto puno veće potrebno je postaviti dobre temelje i testirati performanse od samoga početka.

Ručno testiranje proizvoda je zadnji korak u izradi aplikacije i radi se u više iteracija. Nakon što se problemi uočeni manualnim testiranjem isprave, potrebno je ponovno testirati aplikaciju i taj proces ponavljati dok kvaliteta aplikacije nije na zadovoljavajućoj razini. Dobra stvar je također što se prilikom ručnog testiranja aplikaciju promatra iz perspektive korisnika i moguće je uvidjeti neke nelogičnosti vezane za dizajn i korisničko iskustvo. Ručno testiranje u aplikaciji za pregled rasporeda pacijenata je rezultiralo aplikacijom koja ispunjava sve uvijete postavljene kod planiranja izrade a najbolji pokazatelj rezultata testiranja je krajnji proizvod.

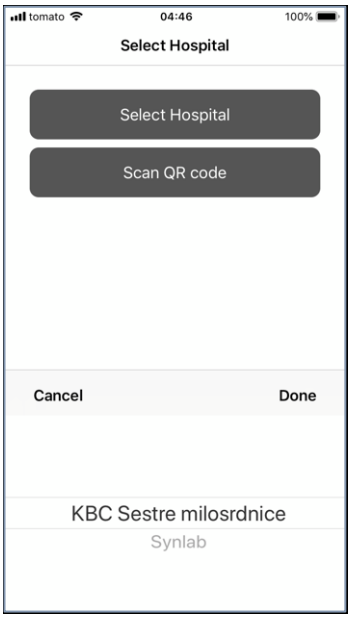
8.3. Grafički prikaz svih zaslona aplikacije

Na slici 8.7 prikazan je zaslon uvoda. Uloga ovoga zaslona je pokazati korisniku način na koji se aplikacija koristi.



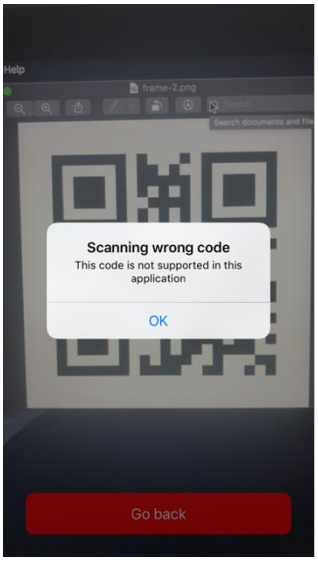
Slika 8.7. *Uvodni zaslon*

Slika 8.8 prikazuje zaslon aplikacije odabir bolnice.



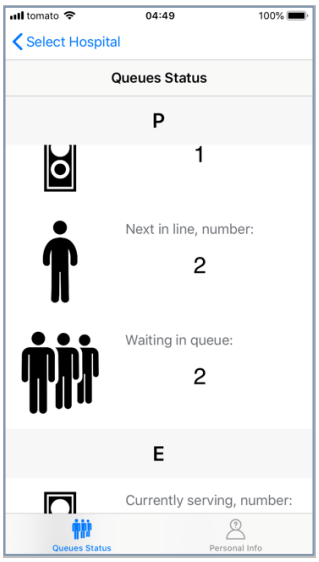
Slika 3.8. *Prikaz zaslona za odabir bolnice*

Na slici 8.9 prikazan je zaslon za skeniranje QR koda uz poruku pogreške koja se prikaže ako skenirani kod nije podržan za ovu aplikaciju.



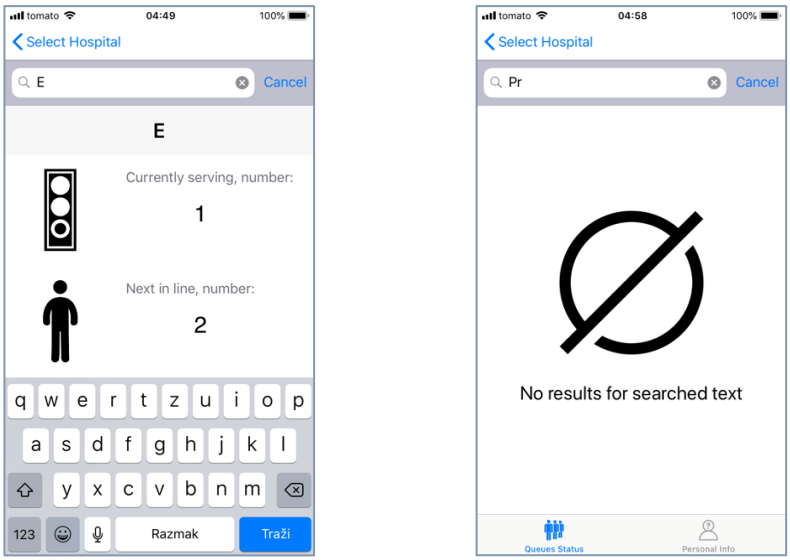
Slika 8.9. *Prikaz zaslona za skeniranje QR koda*

Na slici 8.10 prikazan je zaslon za prikaz stanja redova u bolnici.



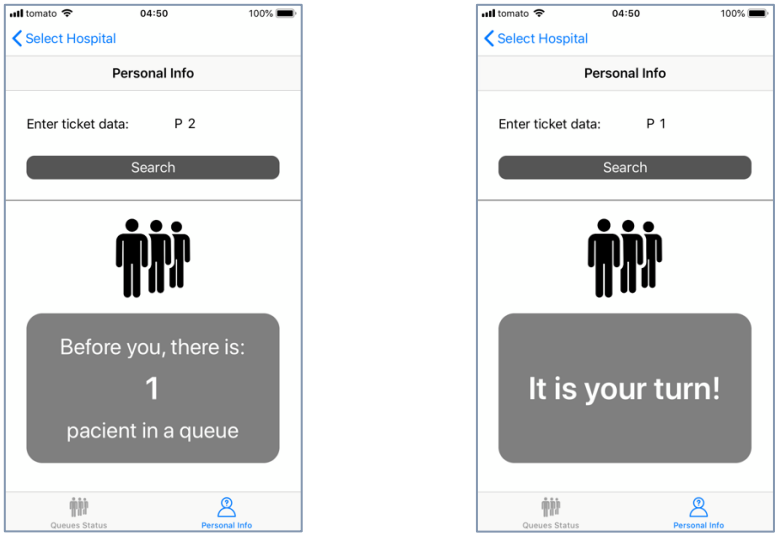
Slika 8.10. *Zaslon s prikazom stanja u redovima*

Slika 8.11 prikazuje isti zaslon kao i slika 8.10 no uz prikaz polja za pretragu redova, također prikazano je stanje zaslona koje se prikaže ako se pretražuje ime reda koje ne postoji.



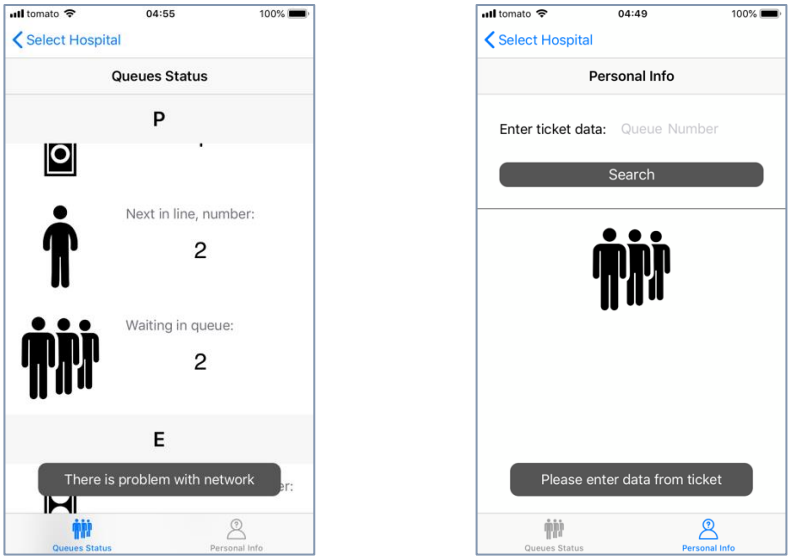
Slika 8.11. *Pretraživanje redova prema korisnikovom unosu*

Na slici 8.12 prikazan je zaslon za prikaz informacija o konkretnom broju nekoga reda.



Slika 8.12. *Zaslon s prikazom podataka za specifičan broj*

Slika 8.13 prikazuje stanje zaslona kada se prikazuju određene obavijesti, na ovome primjeru to su obavijest o problemu s internetskom mrežom te obavijest o tome da je potrebno unijeti podatke s listića kako bi se prokazale informacije za određeni broj.



Slika 8.13. *Prikaz zaslona s obavijestima*

Na slikama 8.14 i 8.15 prikazani su zasloni na većemu uređaju, u ovom slučaju iPad iz 2018 godine.

Slike 8.14. *Prikaz zaslona na uređaju iPad 2018 (Landscape mode)*

Slike 8.15. *Prikaz zaslona na uređaju iPad 2018 (Portret mode)*

9. ZAKLJUČAK

U ovome diplomskom radu razvijena je aplikacija za pregled preostalog broja pacijenata u redomatskom redu. Cilj aplikacije je kontinuirano dohvaćati podatke s poslužitelja za bolnicu koju korisnik odredi ručnim unosom ili skeniranjem QR koda. Podatke je potrebno prikazati korisniku, omogućiti mu pretragu prema imenima redova i unos broja s listića kako bi dobio detaljnije informacije vezano za taj broj.

Aplikacija je razvijena korištenjem razvoja programske podrške pokretanog testovima, te je stoga gotovo cijeli programski kod detaljno testiran pomoću testova jedinica. Nakon što je postupak razvijanja programske podrške vođen testovima završen, aplikacija je testirana testovima performansi i ručno prema testnom scenariju. Rezultati testova pokazuju stabilnost i kvalitetu programskog rješenja. Arhitektura aplikacije rađena je po uzoru na *CLEAN* arhitekturu poštujući *SOLID* principe za izradu programa. Arhitektura dijela korisničkog sučelje koja se koristila u izradi aplikacije je *MVVM* uz korištenje dizajn obrasca koordinatora za navigaciju. Ovakav pristup izradi jamči kvalitetu i robusnost. Također, aplikacija je pogodna za daljnju nadogradnju i može poslužiti kao temelj za gradnju proizvoda puno većega obujma. Nedostatak ovog pristupa je nešto dulje vrijeme izrade u slučaju da programer nije dovoljno iskusan. Potrebno je puno iskustva da bi izrada aplikacije ovim načelima bila vremenski učinkovita.

LITERATURA

[1] L. A. Miller, A Problem of Standards of Service, Applied Queueing Theory, Macmillan International Higher Education, New York, 2016.

[2] Taha, A. Hamdy, Operations research: an introduction, Prentice Hall, 2003

[3] P Sen, Rathindra, Operations Research: Algorithms And Applications, PHI Learning Pvt. Ltd., 2009

[4] N. Gautam, Queueing Theory, Operations Research and Management Science Handbook, Operations Research Series, CRC Press, 2016

[5] M. Harchol-Balter, Scheduling: Non-Preemptive, Size-Based Policies. Performance Modeling and Design of Computer Systems, Cambridge University Press, 2013

[6] A. S. Tanenbaum, Modern Operating Systems (3. izdanje). Pearson Education, 2015

[7] Kaner, Cem, Exploratory Testing, Quality Assurance Institute Worldwide, Annual Software Testing Conference, 2013.

[8] Leitner, Andreas, Ciupa, Ilinca, Oriol, Manuel, Meyer, Bertrand, Fiva, Arno, Contract Driven Development = Test Driven Development – Writing Test Cases, Conference: Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September, 2007

[9] D. Graham, E. Van Veenendaal, I. Evans, Foundations of Software Testing. Cengage Learning, Thomson, 2006

[10] W.L. Oberkampf, C.J. Roy, Verification and Validation in Scientific Computing, Cambridge University Press, 2010

[11] D. Lee, A.N. Netravali, K.K. Sabnani, B. Sugla, A. John, Passive testing and applications to network management, Springer Science & Business Media, 2009

[12] R. Black, Pragmatic Software Testing: Becoming an Effective and Efficient Test Professional, John Wiley & Sons, 2016

[13] Vera-Pérez, O Luis; Danglot, Benjamin; Monperrus, Martin; Baudry, Benoit, A comprehensive study of pseudo-tested methods, Empirical Software Engineering, Springer Verlag, 2018 pp 1-31

[14] R Patton, Software Testing (2nd ed.). Indianapolis: Sams Publishing, John Wiley & Sons, 2011

[15] M.G. Limaye, Software Testing, Tata McGraw-Hill Education, Tata McGraw-Hill Education, 2009

[16] K.A. Saleh, Software Engineering. J. Ross Publishing, 2009

[17] G. T. Laycock, The Theory and Practice of Specification Based Software Testing (PDF) Department of Computer Science, University of Sheffield, 1992

[18] J Bach, Risk and Requirements-Based Testing, Computer magazine pp. 113-114, vol. 32, 1999

[19] R. Savenkov, "How to Become a Software Tester. Roman Savenkov Consulting,", p. 386. ,2008

[20] P. Ammann, J. Offutt, Introduction to Software Testing, Cambridge University Press, 2008

[21] A.P. Mathur, Foundations of Software Testing, Pearson Education India, 2013

[22] A.J. Clapp, S.F. Stanten, W.W. Peng, D.R. Wallace, D. A. Cerino, R. J. Dziegiel Jr, „Software Quality Control, Error, Analysis", William Andrew, 1995

[23] P.A. Mathur, Foundations of Software Testing, Pearson Education India, 2013

[24] J Lönnberg, Visual testing of software (PDF), Helsinki Universit of Technology, 2003

[25] R. Chima, "Visual testing". TEST Magazine, July 24, 2012. Retrieved January 13, 2012.

[26] W.E. Lewis, Software Testing and Continuous Quality Improvement, CRC Press, 2000

[27] R. V. Binder, Testing Object-Oriented Systems: Objects, Patterns, and Tools, Science Press, 2003

[28] IEEE (1990). IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries, Opseg 610,Izd. 1990 iz ANSI / IEEE Std, IEEE, 1990

[29] R.K. Gupta, H. Prajapati, H. Singh, Test-Driven JavaScript Development

[30] K. Beck, Test Driven Development: By Example, Addison-Wesley Professional; 1 edition , November 18, 2002

[31] Newkirk, JW and Vorontsov, AA, Test-Driven Development in Microsoft .NET, Microsoft Press, 2004

[32] Xarawn, TDD Global Lifecycle.png pristupano (posjećeno 26.8.2019)

[33] K. Beck, C Andres, Extreme Programming Explained: Embrace Change (XP Series), Addison-Wesley Professional; 2 edition, November 17, 2004

[34] J. Langr, T. Ottinger, Agile in a Flash, The Pragmatic Programmers, 2011

[35] H. Erdogmus, M. Torchiano, Agile Processes in Software Engineering and Extreme Programming: 13th International Conference, 2012

[36] N. L. Stepping, Through the Looking Glass: Test-Driven Game Development, članak, 2005

[37] H. Mayr, Projekt Engineering Ingenieurmässige Softwareentwicklung in Projektgruppen, Fachbuchverlag Leipzig im Carl Hanser Verlag, 2001

[38] Müller, M. Matthias Padberg, Frank, About the Return on Investment of Test-Driven Development, ICSE-Workshop on Economics-Driven Software Engineering Research, 2003

[39] A. Hunter Are Unit Tests Overused? Simple-talk.com (posjećeno 12.07.2019)

[40] S. Loughran, Testing (PDF). HP Laboratories.

[41] Apple Inc. , Xcode on the Mac App Store. (posjećeno 12.07.2019)

[42] Apple Inc. , The Swift Linux Port, Swift.org. (posjećeno 12.07.2019)

[43] Apple Inc. ,Protocol-oriented Programming in Swift. YouTube. (posjećeno 12.07.2019)

[44] ReactiveX dokumentacija, http://reactivex.io (posjećeno 17.08.2019)

[45] ReactiveX dokumentacija, „Operatori“, http://reactivex.io/documentation/operators.html, (posjećeno 17.08.2019)

[46] R. C. Martin, The Clean Code, https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html, (posjećeno 20.08.2019)

[47] S. Khanlou, The Coordinator, http://khanlou.com/2015/01/the-coordinator/, (posjećeno 20.08.2019)

SAŽETAK

U ovome radu izrađena je mobilna aplikacija za pregled preostalog broja pacijenata u redomatskom redu. Aplikacija prikazuje stanje u redovima bolnice koju korisnik može odabrati ili učitati pomoću QR koda. Također, korisnik može unijeti broj s listića iz redomata i tako dobiti informacije o trenutnom stanju za točno taj broj. Aplikacija je napravljena po uzoru na CLEAN arhitekturu i obrazac dizajna MVVM za dijelove korisničkog sučelja. Aplikacija je rađena kroz razvoj programske podrške vođen testiranjem na primjeru pa su tako provedeni testovi jedinica gotovo cijeloga koda aplikacije, a nakon izrade provedeno je testiranje performansi i ručno testiranje prema testnom scenariju. Ako programer nije uvježban i tek započinje s razvojem programske podrške vođenim testiranjem, vrijeme za pisanje aplikacije bit će znatno duže od onoga uobičajenoga. Testovi performansi pokazali su da je aplikacija stabilna i fluidna i da ne postoje curenja memorije niti procesi koji preopterećuju procesor. Kvaliteta aplikacije na kraju je provjerena ručnim testiranjem po predloženom testnom scenariju koje je pokazalo da aplikacija izvršava sve zadane funkcionalnosti bez pogrešaka.

Ključne riječi: iOS, mobilna aplikacija, razvoj programske podrške vođen testiranjem ,redomatski red, testiranje programske podrške.

DEVELOPMENT AND TESTING OF MOBILE APPLICATION FOR REMAINIG NUMBER OF PATIENTS REVIEW IN QUEUING SYSTEM

ABSTRACT

In this paper the goal was to create mobile application to view the remaining number of patients in a row. The application displays the status in the hospital rows that the user can select or load using a QR code. Also, the user can enter a number from the list from the ticket dispenser and thus obtain information about the current status for that exact number, the data is continuously retrieved from the server and the constant status in the rows is displayed at all times. The application is modeled on the CLEAN architecture and design form MVVM for parts of the user interface. The application was developed through an approach of test-driven software development, so unit tests of almost the entire application code were performed, and after development, manual testing was preformed according to the test scenario, and performance testing was preformed too. The approach to developing test-driven software has proven to be excellent and the only downside is the time it takes to learn how to apply it properly. If the programmer is not trained and is just starting to develop test-driven software, the time to write an application will be much longer than usual, but once the principle is well learned and practiced, it produces great results and it is definitely worth starting with this method of software development. Performance tests have shown that the application is stable and fluid, there are no memory leaks or processes that overload the processor. The quality of the application was eventually verified by manual testing according to a test scenario, which showed that the application performs all given functionalities without errors.

Key words: iOS, mobile application, test driven development , redomatski red, testiranje programske podrške.

ŽIVOTOPIS

Valentin Šarić rođen je 14. veljače 1994. godine u Požegi, Republika Hrvatska. Stanuje u Pleternici, na adresi Matije Antuna Relkovića 22. Godine 2000. započinje osnovnoškolsko obrazovanje u OŠ fra Kaje Adžića u Pleternici te je izrazito aktivan u Nogometnom klubu Slavija Pleternica. Upisuje srednju Tehničku školu u Požegi, smjer računarstvo 2008. godine i počinje se uz nogomet baviti glazbom (gitara). Nakon završetka srednje škole, upisuje Stručni studij informatike na Elektrotehničkom fakultetu u Osijeku 2012. godine i nastavlja sa nogometnom karijerom u Olimpiji iz Osijeka, a studij završava 2016. godine kada upisuje razlikovnu godinu kako bi stekao pravo na upis diplomskog studija. Diplomski studij računarstva upisuje 2017. godine, a krajem 2018. godine pohađa Android praksu u tvrtki Plava tvornica u kojoj trenutno radi kao iOS programer. Od ostalih znanja i vještina posjeduje znanje engleskog jezika, principe objektno orijentiranog programiranja, posebice programske jezike Javu i Swift, te platforme Android i iOS. Posjeduje vozačku dozvolu B kategorije.

PRILOZI

Prilog 1. Dokument rada

Prilog 2. Pdf rada

Prilog 3. Programski kod mobilne aplikacije