iOS aplikacija logičke igre memory

Lovretić, Luka

Master's thesis / Diplomski rad

2020

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **Josip Juraj Strossmayer University of Osijek, Faculty of Electrical Engineering, Computer Science and Information Technology Osijek / Sveučilište Josipa Jurja Strossmayera u Osijeku, Fakultet elektrotehnike, računarstva i informacijskih tehnologija Osijek**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:200:678924

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-05-12**

Repository / Repozitorij:

Faculty of Electrical Engineering, Computer Science and Information Technology Osijek

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

Sveučilišni studij

iOS APLIKACIJA LOGIČKE IGRE „MEMORY“

Diplomski rad

Luka Lovretić

Osijek, 2019.

Sadržaj

1. UVOD ... 1

1.1. Zadatak diplomskoga rada ... 1

2. LOGIČKA IGRA „MEMORY“ ... 2

3. KORIŠTENE TEHNOLOGIJE U IZRADI APLIKACIJE ... 3

3.1. iOS i Swift ... 3

3.2. Xcode ... 6

3.3. iOS Simulator ... 8

3.4. Reaktivno programiranje ... 8

3.5. Biblioteke ... 9

3.6. Model – View – ViewModel ... 11

3.7 Koordinatori ... 12

4. STRUKTURA APLIKACIJE ... 14

4.1. Korisničko sučelje aplikacije ... 14

4.2. Model kartice ... 15

4.3. Model generiranja kartica ... 15

4.4. Ćelije ... 17

4.5. Upravljač ... 18

4.6. ViewModel ... 20

4.7. MPCManager ... 23

5. RAD APLIKACIJE ... 26

5.1. Logička igra ... 26

6. ZAKLJUČAK ... 31

7. LITERATURA ... 32

8. SAŽETAK ... 33

ABSTRACT ... 34

9. ŽIVOTOPIS ... 35

10. PRILOZI 10.1. Prilog P10.1. – Kompletan programski kod ... 36

1. UVOD

Uporaba pametnih mobilnih uređaja postala je ljudska svakodnevnica. Današnji pametni mobilni uređaji su integrirani u gotovo sve aspekte ljudskog života. Kako u medicini, tako i u industrijskom sektoru, pa i u zabavi i opuštanju. Postoje dva najpoznatija operativna sustava za pametne mobilne uređaje. iOS u vlasništvu firme Apple Inc. i Android čiji je vlasnik firma Google Inc…

Zadatak diplomskog rada je izrada iOS aplikacije za igranje logičke igre memory. Logička igra treba imati mogućnosti igranje igre jednog igrača ili igranja u više igrača. Također, korisnik može odabrati između tri ponuđene teme i dva broja parova. Tijekom izrade aplikacije korištena su većinom znanja stečena tijekom obrazovanja na stručnoj praksi za vrijeme 2. godine diplomskog studija. Cilj aplikacije je korisniku omogućiti opuštanje uz vježbanje kognitivnih sposobnosti.

Aplikacija je napisana u Swift programskom jeziku koristeći MVVM arhitekturu gdje je navigacija implementirana pomoću koordinatora. Prvo će se upoznati pravila i objasniti beneficije igranja logičke igre memory. Nakon toga će se objasniti tehnologije i biblioteke korištene za izradu rada, to jest objasnit će teorijsku podlogu operativnog sustava, programskog jezika, korištenih biblioteka i arhitekture i slično. Zatim će biti prikazana i objašnjenja struktura aplikacije te programski kod pomoću kojeg su se implementirala rješenja za glavne probleme u aplikaciji. Na kraju će biti prikazan grafički slijed aplikacije logičke igre memory.

Zasluge za sve kartične ikone pripadaju Icon Pond-u s web stranice www.falticon.com.

1.1. Zadatak diplomskoga rada

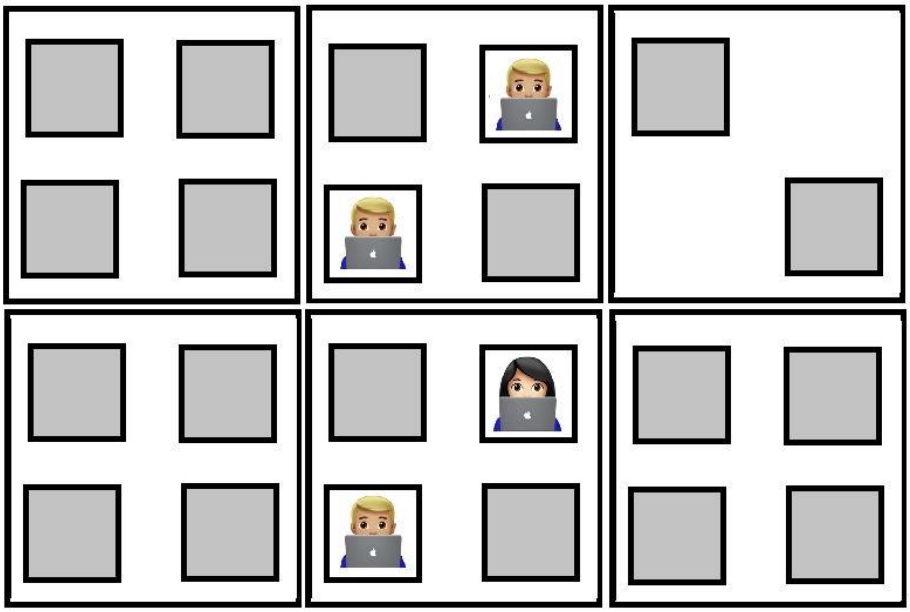
U teorijskom dijelu rada potrebni je proučiti i opisati tehnologije za izradu mobilnih aplikacija za iOS platformu. U praktičnom dijelu rada potrebno je izraditi mobilnu aplikaciju logičke igre memory koja ima opcije igranja igre jednog igrača ili igre s više igrača, napraviti sustav za praćenje rezultata te napraviti nekoliko tema za pozadinu kartica igre.

2. LOGIČKA IGRA „MEMORY“

Logička igra „Memory“ je kartaška igra za jednog ili više igrača. Pravila igre su jednostavna za shvatiti, pa je igra iznimno popularna u mlađim generacijama. Igra se može igrati s bilo kojim špilom karata, a danas ima i veliki broj različitih komercijalnih setova igre s mnoštvom različitih tema.

Na početku igre, karte se postavljaju u redove s licem prema dolje, tako da svaki red ima jednak broj karata. Igrač na potezu, okreće dvije karte. Ako su te karte par, igrač ih osvaja i ima pravo na još jedan potez. Ako one nisu par, okreće ih licem prema dolje na početnu poziciju te predaje potez sljedećem igraču.

Igra završava kada su svi parovi karata osvojeni ili ako igrači predaju igru. Cilj igre je složiti više parova karata od protivnika i na taj način pobijediti u igri. Na slici 2.1. možemo vidjeti jednostavan primjer.



Sl.2.1. *Primjer logičke igre „Memory“*

Igranje logičke igre „Memory“ ima veliki broj beneficija, kao što su: poboljšanje sposobnosti za pronalaženje sličnosti i različitosti između predmeta, koncentracije, vokabulara, vizualnog pamćenja te sposobnosti za klasificiranja predmeta po sličnosti i povećanje kratkotrajnog pamćenja te pozornosti na pojedinosti.

3. KORIŠTENE TEHNOLOGIJE U IZRADI APLIKACIJE

Poglavlje govori o tehnologijama korištenim u izradi aplikacije. Kako bi se razvijale aplikacije za *iOS* uređaje, potrebno je koristiti *Appleovo* sklopovlje i programsku podršku.

3.1. iOS i Swift

iOS je operacijski sustav namijenjen mobilnim uređajima koje proizvodi tvrtka *Apple*. Operacijski sustav je pušten za javnost 2007. godine pod nazivom „*iPhone OS 1*“. Izumom i puštanjem u upotrebu tableta *iPad*, operacijski sustav službeno mijenja ime u „*iOS“*. Svake godine, a tako i ove, platforma se nadograđuje i predstavlja u novoj inačici, pa će se i ove jeseni, 2019., u upotrebu pustiti nova inačica *iOS* 13.

Operacijski sustav se može podijeliti na četiri sloja: *Core OS*, *Core Services*, *Media* i *Cocoa Touch*. Prvi sloj je pisan u *Unix* programskom jeziku, dok ostala tri sloja koriste *Swift* ili *Objective-C* programski jezik. U ovom radu smo koristili *Swift* programski jezik.

Swift je noviji programski jezik pušten u javnost 2014. godine koji se koristi za razvoj *OS X, iOS, tvOS i watchOS* aplikacija. Razvijanje *Swifta* započeo je Chris Lattner korištenjem i unaprjeđivanjem ideja iz drugih programskih jezika kao što su: *Objective-C*, *Ruby*, *Python* i mnogi drugi. [1]

Programski jezik je osmišljen na način da se kompleksne ideje izražavaju na jasan i sažet način. Jednostavnost, brzina i lakoća savladavanja programskog jezika *Swift*, razlozi su što sve veći broj studenata i inženjera posvećuju učenju i razvijanju svojih *Swift* vještina.

Programski jezik *Swift* ima svoje inačice osnovnih tipova, kao što je *Int* za cjelobrojne iznose, *Double* i *Float* za decimalne izraze, *Bool* za vrijednosti koje mogu biti istina ili laž te *String* za tekstualne podatke. *Swift* također ima i kolekcijske tipove, a to su *Array* za čuvanje podataka u polju, *Set* za čuvanje unikatnih podataka u polju i *Dictionary* za čuvanje podataka u polju gdje se određena vrijednost pohranjuje pod određenim ključem.

Osim osnovnih tipova, *Swift* omogućuje još neke tipove kao što su *Tuple* i *Optional*. *Tuple* omogućuje programeru stvaranje i dijeljenje svojih prilagođenih tipova, te jedan takav primjer možemo vidjeti na programskom kodu 3.1.1. *Tuple* se sastoji od podataka kojima ćemo prikazati igru i od podataka koji se koriste za prikaz broja uparenih kartica.

```
public typealias GameDataInit = (collectionData: [CollectionItem<Any,Card>], optionsData: OptionsDataSource)
```

Programski kod 3.1.1. *Primjer Tuple-a*

Optional tip nam govori da vrijednost objekta može imati određenu vrijednosti, no i da ta vrijednost može biti *nil*, to jest da vrijednost ne postoji. *Optional*-i su puno sigurniji od nil pokazivača u Objective-C programskom jeziku te omogućuju neke od najvažnijih značajki programskog jezika *Swift*. [2]

Protocol ili protokol je skup metoda, varijabli i drugih svojstava pomoću kojih se definira izvršavanje nekakvog zadatka. Protokole mogu nasljeđivati klase, strukture i slično te se pomoću njih osigurava da će odrađena klasa imati sposobnost izvršiti određeni zadatak, jer će postojati greška sve dok nisu implementirane sve vrijednosti, metode, odnosno svojstva postavljena u protokolu. Na programskom kodu 3.1.2. vidimo jedan primjer protokola i njegovu implementaciju. Pomoću protokola iz primjera postavljamo koji je igrač na potezu u igri u više igrača.

```
public protocol TurnDelegate: class{
    func itIsYourTurn(_ value: Bool)
}

extension GameViewController: TurnDelegate{
    func itIsYourTurn(_ value: Bool) {
        viewModel.yourTurn = value
        self.viewModel.output.userInteractionPublisher.onNext(value)
    }
}
```

Programski kod 3.1.2. *Primjer protokola i njegove implementacije*

Jedno od najvažnijih svojstava *Swifta* su *Extensions* ili ekstenzije. Pomoću ekstenzija možemo proširiti i prilagoditi postojeće funkcionalnosti svojim klasama, strukturama, enumeracijama ili protokolima. Na programskom kodu 3.1.3. imamo primjer gdje upravljač u ekstenziji nasljeđuje dva protokola pomoću kojih se postavlja izvor podataka za tablični prikaz.

```
extension HostGameViewController: UITableViewDelegate, UITableViewDataSource{
    func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
        return viewModel.output.players.count
    }

    func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
        let cell: PlayerTableViewCell = tableView.dequeue(for: indexPath)
        cell.configureCell(with: viewModel.output.players[indexPath.row].displayName)
        return cell
    }
```

Programski kod 3.1.3. *Primjer nasljeđivanja u ekstenziji*

Kako bi upravljač naslijedio protokole, potrebno je implementirati funkcije koje zahtijevaju ti protokoli. Tako na ovom primjeru vidimo implementaciju funkcije za broj redova unutar tablice

te za postavljanje ćelija u tablici. Osim što u funkcijama možemo naslijediti funkcionalnosti nekih drugih klasa ili protokola, u ekstenzijama možemo implementirati i novu funkcionalnost za željeni objekt. Na programskom kodu 3.1.4. možemo vidjeti primjer jedne takve ekstenzije.

```
extension GameViewModel{
    public func starNewGame(){
        cards = nil
        input.loadDataSubject.onNext(true)
    }

    public func quitGame(){
        if multiPlayerGame{
            MPCManager.shared.sendData(gameData: QuitData(didQuit: true))
        }
    }
```

Programski kod 3.1.4. *Primjer dodavanja novih funkcionalnosti u ekstenziji*

Pomoću *Generic-a* pišu se funkcije i strukture koje se mogu iskoristiti u više navrata. Primjer primjene *Generic-a* vidimo prilikom slanja i primanja podataka u igri s više igrača na programskom kodu 3.1.5. Kako bi se izbjeglo dupliciranje programskoga koda, napravljene su funkcije koje kao parametar primaju generički tip. Taj generički tip mora nasljeđivati protokol *Codebale* kako bi omogućili kodiranje i dekodiranje u i iz JSON objekta. Ukoliko se radi o primanju nekakvog podataka, potrebno je raščlaniti taj objekt u određeni model. Kako bi se funkcija mogla koristiti za sva raščlanjivanja potrebno joj postaviti da vraća generički tip podatka.

```
    public func sendData<T: Codable>(gameData: T){
        do{
            let dataToSend = try JSONEncoder().encode(gameData)
            try session.send(dataToSend,
                             toPeers: session.connectedPeers,
                             with: .reliable)
        }
        catch let error{
            print("There was an error sending game data: \(error.localizedDescription)")
        }
    }

    private func decodeRecivedData<T: Codable>(jsonData: Data) -> T?{
        let object: T?
        do {
            object = try JSONDecoder().decode(T.self, from: jsonData)
        }catch let error {
            debugPrint("Error while parsing data from server. Received dataClassType: \(T.self). More info:
\(error.localizedDescription)")
            object = nil
        }
        return object
    }
```

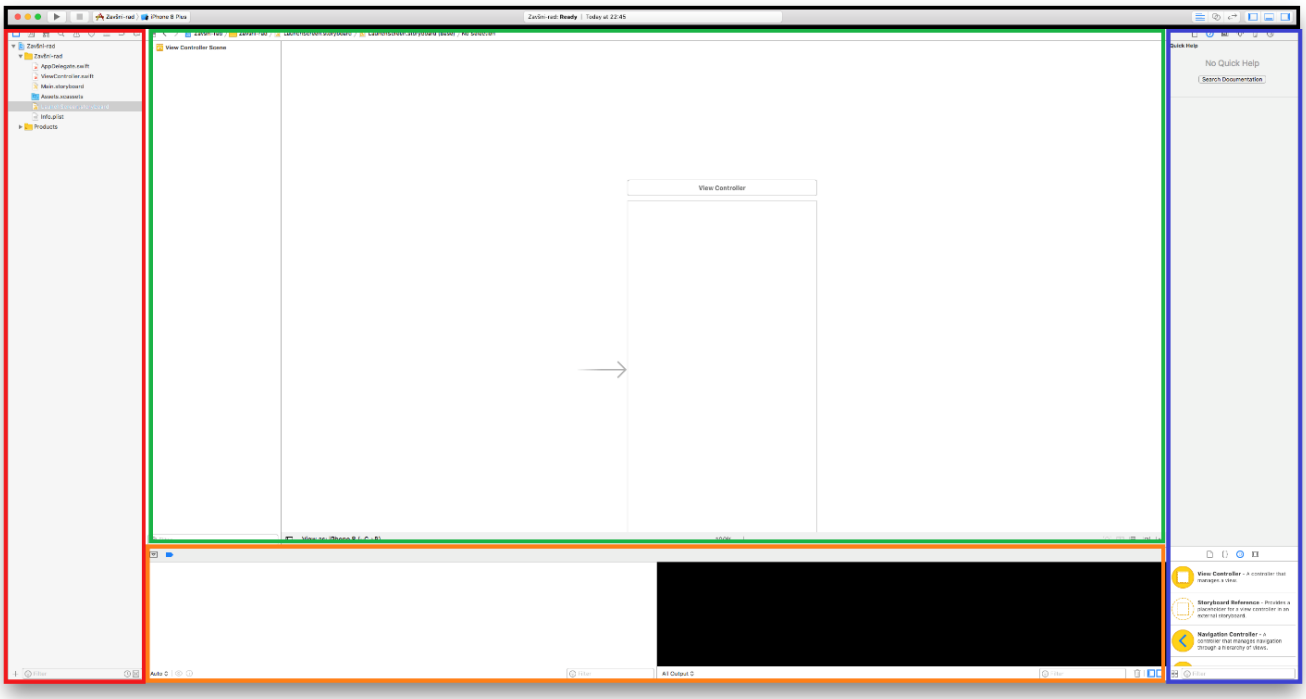
Programski kod 3.1.5. *Primjer uporabe Generic-a*

3.2. Xcode

Xcode je integrirano razvojno okruženje za razvijanje aplikacija za sve *Apple* platforme. *Xcode* se ne naplaćuje te je dostupan na svim *Mac* uređajima. Sadrži prevoditelj sljedeće generacija, LLVM prevoditelj, koji osim što kompilira, neprestano procjenjuje napisani kod, pronalazi sintaktičke i logičke greške u kodu te traži na koji način se mogu ispraviti postojeće greške.

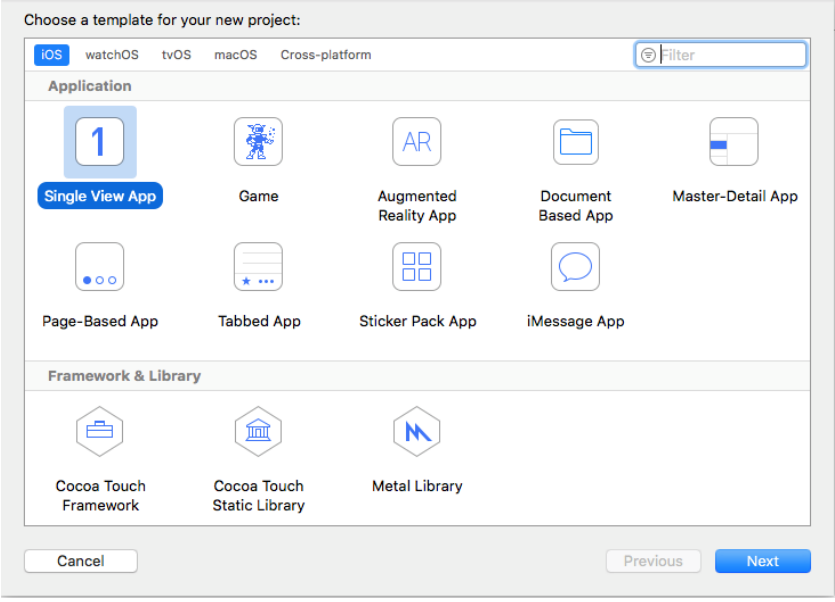
Na slici 3.2.1 možemo vidjeti kako izgleda radni prozor u *Xcodu*. Crnim okvirom je označena alatna traka u kojoj se nalaze gumbi za pokretanje i zaustavljanje aplikacije, izbornik za postavljanje sheme, izbornik za postavljanje uređaja na kojem će se aplikacija pokretati, te gumbi za postavljanje priređivača odnosno radnog prostora. Crvenom bojom je označen navigacijski dio zaslona. Unutar njega se nalaze sve klase i svi popratni dokumenti. Zelenom bojom je označen priređivački prostor, odnosno prostor u kojem razvojni inženjer trenutno radi. Plavom bojom je označen prostor u kojem se nalaze usluge i podrške kojima se razvojni inženjer može služiti prilikom razvoja aplikacije. Narančastom bojom je označeno sučelje za nadgledanje grešaka koje se automatski pokazuje pokretanjem aplikacije.

Svaki razvojni inženjer može prilagođavati izgleda radnog prostora kako bi imao što bolji ugođaj i pregled u pisanju programskog koda.



Sl.3.2.1. *Xcode* radni prozor

Xcode pruža različite predloške za najčešće tipova aplikacija kao što su *Master-detail app, Document Based App* i mnoge druge. Na slici 3.2.2. možemo vidjeti za koje platforme *Xcode* nudi predloške, te koji su to predlošci za *iOS*. Za ostvarenje ovog rada, korišten je *Single View App* predložak.



Sl.3.2.2. Predlošci za *iOS* uređaje

Xcode ima ugrađen graditelj korisničkog sučelja koji se naziva *Storyboard*. Unutar *Storyboarda* programer iz knjižnice s objektima izabire koji mu objekti trebaju te ih jednostavno povlačenjem i ispuštanjem miša postavlja na zaslon. Povezivanje koda i korisničkog sučelja se također obavlja u samo par klikova miša. Osim korištenjem *Storyboard-a*, korisničko sučelje se može izgraditi pisanjem programskog koda.

Kako bi testirali aplikaciju na fizičkom uređaju, potrebno je imati plaćeni račun na https://developer.apple.com. Zasluge za račun pripadaju tvrtki Plava tvornica d.o.o. Na račun možemo dodati aplikaciju te generirati certifikate za potpisivanje aplikacije. Razlikujemo certifikate kojima potpisujemo aplikaciju u razvoju, aplikaciju koja se testira te aplikacija koja se preuzima iz trgovine.

Uz Xcode dolazi još par programa kao što su *Instruments* (služi za optimizaciju aplikacija), *iOS Simulator* (služi za testiranje aplikacije prilikom razvijanja), *Accessibility Inspector* (služi za postavljanje govorne pomoći – za slijepe osobe) i slično.

3.3. iOS Simulator

IOS simulator je alat koji služi kako bi pokrenuli i testirali aplikaciju na našem računalu. Simulator omogućuje simulaciju za većinu *Apple* uređaja te podržava opcije koje ti uređaji imaju.

Simulator ima svoje prednosti i nedostatke. Koristan je za simuliranje i testiranje korisničkog sučelja koje se može obavljati interakcijom miša, međutim za simuliranje i testiranje složenijih programskih rješenja postaje nedovoljan.

Cijela aplikacija ovog diplomskog rada je simulirana u *iOS Simulatoru* i simulator je mogao i uspio odraditi sve potrebne zahtjeve korisnika. Aplikacija je također testirana na fizičkim uređajima različitih dimenzija.

Sl.3.3.1. *iOS simulator*

3.4. Reaktivno programiranje

Reaktivno programiranje je način programiranja u kojem jedna komponenta odašilje protoke podataka dok se druga komponenta na njih pretplaćuje i sluša promjenu vrijednosti unutar toka te ima odgovarajuću reakciju na nju. Reaktivno programiranje je uvedeno iz razloga što mobilni uređaji nisu dovoljno snažni za kompleksno računanje i teške poslove, pa dolazi do smrzavanja mobilnih aplikacija i slično. Protok podataka postoji sve dok se ne pošalje događaj *greške* ili događaj koji javlja da su poslani svi podaci. Cijelo mobilno reaktivno programiranje se svodi na:

Observable – predstavlja protok podataka. Podaci se mogu odašiljati periodično ili samo jedanput, ovisno o postavkama protoka podataka. Pomoću njih dohvaćamo i obrađujemo podatke koje prikazujemo.

Observer – predstavlja promatrača protoka podataka. Svaki promatrač se može pretplatiti na protok podataka. Svaki puta kada se odašilje podatak, promatrač u svojoj pretplati može izvršiti nekakvu akciju ili manipulaciju nad tim podacima.

Scheduler – pomoću njega postavljamo koja akcija će se izvršavati na kojem *Threadu*. Tako da se sva promatranja protoka podataka izvršavaju na *Main threadu*, dok se sve pretplate izvršavaju na *Background threadu*.

3.5. Biblioteke

Korištenjem biblioteka, razvojni inženjer si olakšava i ubrzava rješavanje programskih problema. Svaka biblioteka sadrži različite klase i objekte kojima inženjer ima pristup. Također manipulacijom koda unutar klasa ili objekata, biblioteke se mogu prilagoditi za neku novu vrstu implementacije. *Apple* pruža veliki broj biblioteka koji pomažu u pisanju programa, dok su neke od njih: *WebKit, ARKit, WatchKit, MapKi, ...* Za ostvarenje ovog rada koristili smo *foundation, UIKit*, *MultipeerConnectivity* i *RxSwift* biblioteke.

Biblioteka *foundation* nam pruža osnovni sloj funkcionalisti za aplikacije, to jest omogućuje nam sortiranje, filtriranje, izračunavanje datuma i vremena i druge operacije.

UIKit je biblioteka koji sadrži veliki broj značajki potrebnih za infrastrukturu *iOS* ili *tvOS* aplikacija. [1]

Neke od ključnih značajki su:

Sučelje za stvaranje i upravljanje – mogućnost stvaranja i organiziranja polja za pisanje, boje, fonta i sličnih funkcija.

Upravljanje životom aplikacije – mogućnost upravljanja radom aplikacije u situacijama kao što su pokretanje i zatvaranje aplikacije, odlazak u pozadinu i slično.

Upravljanje događajima – uspostavljanje odnosa između aplikacije i korisnika aplikacije, npr. dodir zaslona od strane korisnika.

Višezadaćnost – mogućnost obavljanja više događaja u isto vrijeme.

Funkcija kopiranja, rezanja i zaljepljivanja – daje mogućnost kopiranja, rezanja i zaljepljivanja aplikaciji.

Zaštita podataka – omogućuje zaštitu podataka pomoću enkripcije podataka.

Bežični ispis – omogućuje bežični ispis dokumenata.

Prepoznavanje dodirnih gesta – omogućuje korisniku korištenje aplikacija pomoću dodirnih gesta.

Pristup – omogućuje aplikaciji pristup korisnikovim podacima kao što su kalendar, kontakti i slično.

MultipeerConnectivity je biblioteka koja omogućuje povezivanje dva ili više uređaja putem *WiFi* mreže ili *Bluetooth* mreže. Omogućuje prijenos podatak između uređaja. Kako bi omogućili komunikaciju potrebno je implementirati nekoliko tipova objekata: [2]

- *MCSession* – objekt koji omogućuje komunikaciju između uparenih uređaja. Uređaj domaćin stvara sesiju te povezuje svaki uređaj koji prihvati poziv za povezivanje. Objekt također čuva set svih ID-eva uparenih uređaja.

- *MCNearbyServiceAdvertiser*- objekt koji govori uređajima u blizina da su se voljni spojiti na sesiju određenog tipa. Objekt koristi lokalni partnerski objekt kako bi pružio informacije koje će identificirati uređaj i njegovog korisnika ostalim uređajima u neposrednoj blizini.

- *MCAdvertiseAssistant* – objekt koji ima sličnu funkcionalnost kao i *MCNearbyServiceAdvetiser*. Glavana razlika je ta što ovaj objekt pruža korisničko sučelje za prihvaćanje, odnosno odbijanje, pozivnica za spajanje na sesiju.

-*MCNearbyServiceBrowser* – objekt koji omogućuje programatsko traženja uređaja koji imaju aplikaciju koja podržava spajanje na sesiju određenog tipa.

- *MCBrowserViewController* – objekt koji pruža korisničko sučelje za prikaz uređaja u neposrednoj blizini s kojima se korisnik može povezati.

- *MCPeerID* – objekt koji pruža unikatnu identifikaciju uređaju.

RxSwift biblioteka je jedna od inačica *Reactive extensions*-a bilbioteke koja omogućuje reaktivno programiranje s programskim jezikom *Swift*. RxSwift je generička apstrakcija

programiranja izražena sučeljem *Observable<Element>*.[3] Neke od funkcionalnosti koje se koriste u aplikaciji su:

- *Observables* – pomoću ove funkcionalnosti odašilju se podaci, to jest stvaraju se asinkroni tokovi podataka.

- *Subscribe* – pomoću ove funkcionalnosti se pretplaćuje na asinkrone protoke podataka. Ukoliko se odašilje nekakva vrijednost, u pretplati se dobije njena vrijednost te se s njom može dalje manipulirati ili pozvati akciju kojoj je ta vrijednost potrebna.

- *Dispose* – pomoću ove funkcionalnosti možemo ručno otkazati pretplatu, to jest ne moramo čekati zadnji događaj i grešku u protoku.

- *Subject* – subjekti u isto vrijeme omogućuju ponašanje promatrača i promatranog protok podataka. Razlikujemo više tipova subjekata kao što su *Publish* i *Replay*. Publish subject započinje odašiljanje s praznom vrijednošću i svoje vrijednosti odašilje samo pretplatnicima. Replay subject se inicijalizira s određenom veličinom međuspremnika, te će podatke do veličine međuspremnika odašiljati novim pretplatnicima.

RxSwift ima mnoštvo operatora pomoću kojih se može odraditi pretvorba podataka. Neki od njih su *map, filter, flatMap* i slično.

3.6. Model – View – ViewModel

Model-View-ViewModel je način dizajna programske arhitekture. Arhitektura programske podrške je plan koji opisuje skup aspekata i odluka koji su od iznimne važnosti za projekt. Podrazumijeva uzimanje u obzir sve vrste zahtjeva, organizaciju sustava, međusobnu komunikaciju između dijelova sustava, koji bi se rizici trebali razmatrati i drugo. Također uzima u obzir vanjske aktivnosti, ako one postoje.

Glavna značajka MVVM arhitekture je potpuna odvojenost poslovne logike od logike za postavljanje pogleda. Unutar *ViewModel-a* se obavlja sva manipulacija nad podacima, tako da *View* ne zna ništa o podacima osim njihovog prikaza na korisničkom sučelju. Na taj način smo omogućili posebno testiranje poslovne logike i logike pogleda.

Kao što vidimo iz naziva, arhitektura se sastoji od tri dijela: [5]

- ViewModel – sadržava poslovnu logiku aplikacije i sastoji se od različitih programskih dijelova ovisno o složenosti aplikacije.
- Pogled (*View*) – sadržava kod za prikaz vizualnih elemenata korisničkog sučelja i reakcije na korisničke akcije. Sastoji se od protokola ili *evenata* koji sadržavaju metode korisnikovih akcija i *ViewController* koji je odgovoran za materijalizaciju i prikazivanje komponenti korisničkog sučelja te otkrivanje događaja koje prosljeđuje *viewModel-u*.
- Model – sadržava podatke aplikacije. Najčešće se radi od klasama ili strukturama.

Na slici 3.5.1. grafički je prikazana MVVM šablona.

Sl.3.5.1. MVVM

Svaki složeniji *ViewModel* sastoji se od *Ulaza, Izlaza* i *Ovisnosti*. Ulaze čine svi događaji koji korisnik može komunicirati s aplikacijom. Izlaze čine sve vrijednosti koje javljaju upravljaču da su podaci spremni za korištenje. Ovisnosti čine sve vrijednosti koje su potrebne da bi se napravio odgovarajući izlaz za određeni ulaz.

MVVM arhitektura ima prednosti pri otklanjaju pogrešaka, olakšava praćenje podrijetlo događaja, olakšava testiranje te izbjegava „skrivene“ lance događaja jer im nedostaje logika. Jedna od mana ove arhitekture je smanjenje brzine razvoja programa jer implementacija *viewModel* zahtjeva dodatan posao te zahtjeva pravljenje posebnih datoteka za pojedinu komponentu.

3.7 Koordinatori

Koordinator obraz predstavio je Soroush Khanlou u svome blogu i na konferenciji *NSSpain*. [3] Obraz obuhvaća cijelu navigacijsku logiku u *iOS* aplikacijama. Kako bi upravljač imao zadaću upravljati jedino s prikazom različitih pogleda, uvodi se obrazac koordinator. Koordinator je odgovoran za stvaranje upravljača, njihov prikaz i njihove brisanje. Time upravljači postanu izolirani i nevidljivi jedni drugome te se mogu lako koristiti u više navrata.

Zbog svoje jednostavnosti, za obrazac nije potrebno koristiti nikakve dodatne biblioteke. Unutar aplikacije postoji klasa *AppCoordinator*. U toj klasi započinje navigacija aplikacije.

Klasa je napravljena kako bi se smanjila klasa *AppDelegate*. Programski kod 3.7.1 prikazuje klasu *AppCoordinator*.

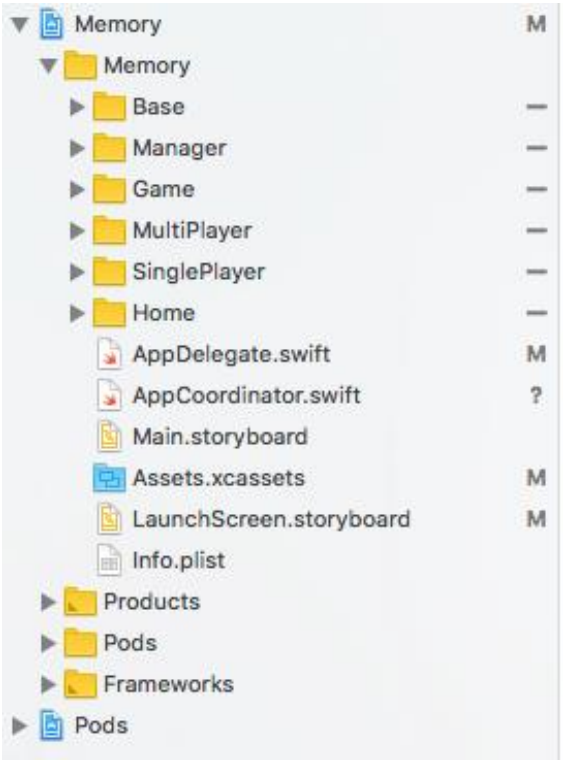
```
class AppCoordinator: Coordinator{

    var childCoordinators: [Coordinator] = []
    let window: UIWindow
    //root navigation controller
    var presenter: UINavigationController

    init(window: UIWindow) {
        self.window = window
        presenter = UINavigationController()

    }

    func start() {
        window.rootViewController = presenter
        window.makeKeyAndVisible()
        presenter.isNavigationBarHidden = true
        let homeCoordinator = HomeCoordinator(presenter: presenter)
        addChildCoordinator(childCoordinator: homeCoordinator)
        homeCoordinator.start()
    }
}
```

Programski kod 3.7.1. AppCoordinator

Svaki koordinator u aplikaciji nasljeđuje protokol *Coordinator*. Protokol zahtjeva implementaciju *childCoordinatora*, funkcije *start* i varijable *presenter*. Funkcija *start* u svakom koordinatoru ima istu zadaću, a to je prikazivanje upravljača koji se nalazi unutar njega. Svaki koordinator ima polje dijete koordinatora, dok svako dijete koordinator ima poveznicu na roditelja. *Presenter* predstavlja korijen navigacije naše aplikacije. Preko te varijable se postavljaju upravljači na stog, odnosno brišu upravljači sa stoga.

4. STRUKTURA APLIKACIJE

Aplikacija je pisana MVVM arhitekturom uz korištenje koordinatora zbog toga ima više datoteka. Na slici 4.1. možemo vidjeti strukturu. Svaka funkcionalnost je odvojena u zasebnu datoteku, te svaka funkcionalnost ima svoj upravljač, koordinator i viewModel. Osim funkcionalnosti postoji *Manager* koji nam služi za postavljanje komunikacije između uređaja i bazična datoteka u kojoj su implementirane klase i funkcionalnosti koje se koriste u cijeloj aplikaciji.



Sl.4.1. Struktura aplikacije

U *AppDelegatu* se nalaze životne metode aplikacije, u *AppCoordinatoru* se nalazi koordinator aplikacije pomoću kojeg se započinje navigacije u aplikaciji te u *LaunchScreen.storyboard*-u se nalazi *Splash screen* aplikacije.

4.1. Korisničko sučelje aplikacije

Aplikacija razvijena u diplomskom radu ima podržava sve dimenzije iPhone uređaja. Također ima podršku korištenja aplikacije u uspravnom, odnosno portretnom načinu rada.

Korisničko sučelje aplikacija je razvijeno pisanjem programskog koda. Aplikacija ima pet zaslona, a to su:

- *zaslon za izbor igre za jednog ili više igrača*

- *zaslon za postavljanje opcija igre za jednog igrača*
- *zaslon za pridruživanje u igru s više igrača*
- *zaslon za postavljanje opcija igre u više igrača*
- *zaslon igre*

U radu će se razraditi samo funkcionalnosti i korisničko sučelje koje su vezane za igru.

Za izradu zaslona igre potrebno je dodati *UICollectionView*. U njemu će se nalaziti kartice za igru. Također treba postaviti ćeliju za *UIColletionView*. Unutar ćelija se postavljaju dva *UIImageView* koje predstavljaju našu karticu. Jedan pogled predstavlja poleđinu kartice, dok drugi predstavlja sliku kartice. Osim toga, napravljen je prilagođen pogled u kojem se prikazuje broj spojenih kartica, vrijeme trajanja igre, te *UIButton-i* za početak nove igre i izlazak iz trenutne igre.

4.2. Model kartice

Model kartice je jednostavna klasa nazvana *Card* gdje joj dodjeljujemo varijable: ime, je li okrenuta, podudara li se, treba li se maknuti iz igre i treba li se okrenuti u prvotno stanje. Osim spomenutih varijabli, model nasljeđuje *typealias Codable* (protokole *Decodable* i *Encodable*) kako bi omogućili primanje i slanje kartice prilikom igre u više igrača.

```
public class Card: Codable {
    var imageName = ""
    var isFlipped = false
    var isMatched = false
    var needsRemoving = false
    var needsFilliping = false
}
```

Programski kod 4.2.1. Model kartice

4.3. Model generiranja kartica

Kako bi napravili bazu kartica koje će se koristiti, potrebno je generirati polje u koje će se spremiti kartice u parovima uzimajući u obzir da parovi moraju biti unikatni. Za svaku temu postoje 18 različitih slika, te generiranjem kartica treba se uzeti u obzir da se slike ne smiju ponavljati. Pomoću funkcije *uniqueRandoms(numberOfRandoms: Int) -> [Int]* dobijemo polje cjelobrojnih brojeva pomoću kojih ćemo dobiti polje kartica u funkciji *createPairOfCards(for theme: Themes, with randomNumbers: [Int]) -> [Card]*. Implementaciju funcija možemo vidjeti na programskom kodu 4.3.1. U tom polju parovi se nalaze jedan pored drugoga, te će takvi biti postavljeni na pogled. To znači da će igra biti iznimno lagana, to jest neigriva.

```
private func createPairOfCards(for theme: Themes, with randomNumbers: [Int]) -> [Card]{
        var pairOfCards = [Card]()
        for randomNumber in randomNumbers{
            let cardOne = Card()
            cardOne.imageName = "\(theme.getThemeCardTitle(cardNumber: randomNumber))"
            pairOfCards.append(cardOne)
            let cardTwo = Card()
            cardTwo.imageName = "\(theme.getThemeCardTitle(cardNumber: randomNumber))"
            pairOfCards.append(cardTwo)
        }
        return pairOfCards
    }

func uniqueRandoms(numberOfRandoms: Int) -> [Int] {
    var uniqueNumbers = Set<Int>()
    while uniqueNumbers.count < numberOfRandoms {
        uniqueNumbers.insert(Int(arc4random_uniform(18) + 1))
    }
    return Array(uniqueNumbers)
}
```

Programski kod 4.3.1 Metoda za stvaranje polja unikatnih parova

Kako bi to spriječili, potrebno je napraviti novo polje u koje će se nasumično spremati vrijednosti kartica što možemo vidjeti na programskom kodu 4.3.2.

```
private func createCardsArray(with generatedCardsArray: [Card]) -> [Card]{
        var generatedCards = generatedCardsArray
        var cardsToAdd = [Card]()
        for _ in 1...generatedCards.count {
            let randomOrder = Int(arc4random_uniform(UInt32(generatedCards.count)))
            cardsToAdd.append(generatedCards[randomOrder])
            generatedCards.remove(at: randomOrder)
        }
        return cardsToAdd
}
```

Programski kod 4.3.2. Metoda za stvaranje polja kartica [6]

Funkcija prima polje unikatnih kartica te ih nasumično postavi unutar polja kako bi bilo moguće igrati igru.

Unutar funkcije *grabCards()* radimo oba zahtjeva. Prvo pomoću nasumično generiranih brojeva pravimo parove kartica u *generatedCardsArray* polje. Zatim prolazimo kroz to polje, nasumično uzimamo vrijednosti, te ih spremamo u novo polje *cards*.

```
public func grabCards(for theme: Themes, with numberOfPairs: NumberOfPairs) -> Observable<[Card]>{
        var cards = [Card]()
        var generatedCardsArray = [Card]()
        let randomNumbers = uniqueRandoms(numberOfRandoms: numberOfPairs.numberOfParisValue)
        generatedCardsArray += createPairOfCards(for: theme, with: randomNumbers)
        cards += createCardsArray(with: generatedCardsArray)
```

```
        return Observable.just(cards)
    }
```

Sl.4.3.3. Repositorij za generiranje kartica

4.4. Ćelije

Unutar ćelije smo definirali dva *UIImageView-a* koji predstavljaju našu karticu. Zatim pišemo funkcije za postavljanje kartica, okretanje kartica, okretanje kartica nazad i brisanje kartica s pogleda.

```
public func configureCell(for card: Card){
        frontImageView.image = UIImage(named: card.imageName)
        //Keep track of the card that gets passed in
        if card.needsRemoving {
            remove()
            return
        }
        if card.isMatched == true {
            //if the card has been matched make image views invisible
            remove()
            return
        } else {
            //if the card hasnt been matched make image views visable
            backImageView.alpha = 1
            frontImageView.alpha = 1
        }
        //determine if the card is in a flipped up state or flipped down state
        if card.isFlipped == true {
            flipCard()
        } else {
            if card.needsFilliping  {
                flipCardBack()
                return
            }
            flipCardBack()
        }
    }

    public func flipCard(){
        UIView.transition(from: backImageView,
                          to: frontImageView,
                          duration: 0.3, options: [.transitionFlipFromLeft, .showHideTransitionViews], completion: nil)
    }

    public func flipCardBack(){
        UIView.transition(from: self.frontImageView,
                          to: self.backImageView,
                          duration: 0.3, options: [.transitionFlipFromRight, .showHideTransitionViews], completion: nil)
    }

    public func remove(){
        backImageView.alpha = 0
```

```
        UIView.animate(withDuration: 0.3, delay: 5, options: .curveEaseOut, animations: {
                self.frontImageView.alpha = 0
        }, completion: nil)
}
```

Sl.4.4.1. Manipuliranje karticama

4.5. Upravljač

Kako bi aplikacija bila što manje memorijski zahtjevna, koristi se isti upravljač za igru u više igrača i za igru kada korisnik igra sam. Zaslon igre sadrži jedan upravljač u kojem se nalaze metode pomoću kojih postavljamo *UICollectionView*. Postavlja se broj sekcija, postavljaju se kartice u ćelije, odnosno postavlja se izvor podataka za *UICollectionView*, veličina pojedine ćelije, te se definira što će se dogoditi kada korisnik odabere jednu od kartica. Kod implementacije prikazan je na programskom kodu 4.5.1.

Potrebno je odrediti *Ulaze* za koje je potrebno odrediti odgovarajuće *Izlaze*. Jedan od takvih ulaza smo vidjeli na prethodnom programskom kodu pri definiranju akcije na korisnikov izbor kartice.

```
extension GameViewController: UICollectionViewDelegate, UICollectionViewDataSource,
UICollectionViewDelegateFlowLayout{
    func collectionView(_ collectionView: UICollectionView, numberOfItemsInSection section: Int) -> Int {
        return viewModel.output.gameData.collectionData.count
    }

    func collectionView(_ collectionView: UICollectionView, cellForItemAt indexPath: IndexPath) -> UICollectionViewCell {
        let cell: CardCollectionViewCell = collectionView.dequeue(for: indexPath)
        let item = viewModel.output.gameData.collectionData[indexPath.row].data
        cell.configureCell(for: item)
        return cell
    }

    func collectionView(_ collectionView: UICollectionView, layout collectionViewLayout: UICollectionViewLayout,
sizeForItemAt indexPath: IndexPath) -> CGSize {
        let itemsCount = viewModel.output.gameData.collectionData.count
        if itemsCount == 16 {
            return CGSize(width: UIScreen.main.bounds.width/4.5,
                    height: UIScreen.main.bounds.width/(4.5*0.66))
        }else{
            return CGSize(width: UIScreen.main.bounds.width/6.5,
                    height: UIScreen.main.bounds.width/(6.5*0.66))
        }
    }

    func collectionView(_ collectionView: UICollectionView, didSelectItemAt indexPath: IndexPath) {
        input.cardPickPublisher.onNext(indexPath.row)
    }
}
```

Programski kod 4.5.1. Korištenje *UIColletionViewa*

Nakon što smo odredili *Ulaze* potrebno je pozvati funkciju iz *ViewModel-a* pomoću koje se podaci obrađuju u podatke koje *Upravljač* zna prikazati. *Ulazi* su i korisnikova komunikacija s *UIButton-ima* za izlaz iz igre i za početak nove igre.

Na programskom kodu 4.5.2. vidimo programski kod koji prikazuje jednu takvu pretvorbu. Naš izlaz ima varijablu *Disposables*. Unutar varijable su pohranjene sve vrijednosti čijim rezultatom *Upravljač* raspolaže te se rezultati moraju *zbrinuti*.

```
private let input = GameViewModel.Input(loadDataSubject: ReplaySubject.create(bufferSize: 1),
                                        cardPickPublisher: PublishSubject())

    private func initalizeVM(){
        let output = viewModel.transform(input: input)
        output.disposables.forEach { (disposable) in
            disposable.disposed(by: disposeBag)
        }
        initializeRefreshDriver(refreshObservable: viewModel.output.refreshView)

        output.userInteractionPublisher
            .observeOn(MainScheduler.instance)
            .subscribeOn(ConcurrentDispatchQueueScheduler(qos: .background))
            .subscribe(onNext: { [unowned self](isEnable) in
                self.collectionView.isUserInteractionEnabled = isEnable
            })
            .disposed(by: disposeBag)

        output.optionDataRefresh
            .observeOn(MainScheduler.instance)
            .subscribeOn(ConcurrentDispatchQueueScheduler(qos: .background))
            .subscribe(onNext: { [unowned self](_) in
                self.optionsView.configureView(for: self.viewModel.output.gameData.optionsData,
                                               itemsCount: self.viewModel.output.gameData.collectionData.count)
            })
            .disposed(by: disposeBag)
}
```

Programski kod 4.5.2. Inicijalizacija pretvorbe podataka

Izlaz se sastoji i od *Subjekata* koji odašilju *događaje* koje upravljač koristi za postavljanje *pogleda*. Programski kod na programskom kodu 4.5.2 prikazuje dva takva subjekta. Jedan subjekt služi za postavljanje broja uparenih kartica, dok drugi omogućuje korisnikovu komunikaciju s karticama.

4.6. ViewModel

ViewModel je klasa pomoću koje se dohvaćaju podaci, obrađuju podaci te javljaju upravljaču da može prikazati podatke, odnosno to je klasa u kojoj se definira sva poslovna logika naše igre. Kako bi aplikacija bila što manje memorijski zahtjevna, koristi se isti viewModel za igru u više igrača i za igru kada korisnik igra sam.

Funkcija za prvotno postavljanje pogleda, prikazana je na programskom kodu 4.6.1. Funkcija kao parametar prima subjekt koji se okida kada je potrebno postaviti sve kartice za igru, a kao izlaz daje inicijaliziran zaslon s ispravnim podacima.

```
private func initializeData(for dataSubject: ReplaySubject<Bool>) -> Disposable{
        return dataSubject.flatMap({ [unowned self](_) -> Observable<[Card]> in
            if let cards = self.dependecies.cards{
                self.output.userInteractionPublisher.onNext(self.yourTurn)
                return Observable.just(cards)
            }
            return self.dependecies.gameRepository.grabCards(for: self.dependecies.theme, with:
self.dependecies.numberOfPairs)
        }).map({[unowned self] (cards) -> GameDataInit in
            self.resetPlayersScore()
            let collectionItems = self.createCollectionData(for: cards)
            let optionsItems = self.createOptionsData()
            return GameDataInit(collectionData: collectionItems, optionsData: optionsItems)
        })
            .observeOn(MainScheduler.instance)
            .subscribeOn(ConcurrentDispatchQueueScheduler(qos: .background))
            .subscribe(onNext: { [unowned self](gameData) in
                self.output.gameData = gameData
                self.checkIfYouNeedToSendGameData()
                self.output.refreshView.onNext(.complete)
            })
}
```

Programski kod 4.6.1. Inicijalizacija podataka

Razlikujemo dvije inicijalizacije podataka. Jedna inicijalizacija koristi repozitorij za pravljenje kartica. Takva inicijalizacija se koristi u slučajevima kada korisnik odabere igru u kojoj igra sam te u slučaju kada je korisnik domaćin igre u više igrača. Ukoliko se korisnik priključuje u igru, već ima polje kartica koje treba prikazati (dobije ga od igrača domaćina), pa nema potrebu stvarati nove kartice.

Prilikom inicijalizacije podataka, potrebno je pretvoriti podatke iz repozitorija u podatke koje će upravljač znati prikazati. Funkcije *createCollectionData(for cards: [Card]) -> [CollectionItem<Any,Card>]* i *createOptionsData() -> OptionsDataSource* za rezultat imaju polje kartica koje je izvor podataka za *CollectionView*, odnosno izvor podataka za postavljanje

komponenti koje prikazuju broj spojenih kartica. U pretplati odašiljemo događaj da su podaci spremni za korištenje, pohranjujemo podatke za prikaz u polje i provjeravamo je li potrebno slati podatke igre.

Svaka korisnikova interakcija s zaslonom se odašilje preko ulaza za zaslon. Tako da postoje tri različita protoka podataka koje korisnik za vrijeme igre može započeti. Jedan je započinjanje nove igre, drugi je završetak trenutne igre i treći je odabir željene kartice. Na programskom kodu 4.6.2. možemo vidjeti programski kod koji se poziva kada korisnik dodirne jednu od kartica. Unutar koda je postavljena cijela logika igre. Postavljamo pomoćnu varijablu koja će reći je li to prva okrenuta kartica i na temelju te varijable će se pozivati određene akcije u pretplati.

```
private func initializeCardPickObservable(cardPickPublisher: PublishSubject<Int>) -> Disposable{
    var isFirstIndex: Bool = false
    return cardPickPublisher.map({[unowned self](index) -> ([CollectionItem<Any,Card>], Int) in
        let card = self.output.gameData.collectionData[index].data
        isFirstIndex = self.checkIfFirstCardIsFliped(for: card, with: index)
        self.checkIfMoveDataNeedsToBeSend(for: index)
        return (self.output.gameData.collectionData, index)
    }).observeOn(MainScheduler.instance)
        .subscribeOn(ConcurrentDispatchQueueScheduler(qos: .background))
        .subscribe(onNext: { [unowned self](data) in
            if isFirstIndex{
                self.output.refreshView.onNext(.reloadRows(indexPaths: [IndexPath(row: data.1, section: 0)]))
            }else{
                self.output.refreshView.onNext(.reloadRows(indexPaths: [IndexPath(row: data.1, section: 0)]))
                self.output.userInteractionPublisher.onNext(false)
                self.checkForMatches(for: data.1)
            }
        })
}
```

Programski kod 4.6.2. Korisnikova komunikacija s karticom

Prilikom pretvorbe podataka potrebno je provjeriti trebaju li se podaci slati na server i je li okrenuta prva ili druga karta. Pomoću metode *checkIfFirstCardIsFillped(for card: Card, with index: Int) -> Bool*.

```
private func checkIfFirstCardIsFliped(for card: Card, with index: Int) -> Bool{
    if card.isFlipped == false {
        card.isFlipped = true
        self.output.gameData.collectionData[index].data = card
        if self.firstFlippedCardIndex == nil{
            self.firstFlippedCardIndex = index
        }else {
            return false
        }
```

```
        }
        return true
    }
```

Programski kod 4.6.3. Ispitivanje koja je kartica okrenuta

Funkcija kao parametre prima okrenutu karticu i indeks na kojoj se nalazi ta kartica u polju, a kao rezultat vraća bool vrijednost. Ukoliko je globalna varijabla za utvrđivanje je li okrenuta prva ili druga karta, vraćamo rezultat. Također, unutar funkcije se mijenja status kartice i kartica s promijenjenim statusom se pohranjuje u polje na isto mjesto. Funkcija *checkIfMoveDataNeedsToBeSend(for index: Int)* provjerava radi li se o igri u više igrača i je li korisnikov potez te na temelju tih informacija odlučuje o slanju podataka.

U pretplati imamo dva moguća slijeda akcija. U prvom slučaju, okrenuta kartica je prva kartica koju je korisnik okrenuo. U tom slučaju potrebno je javiti upravljaču da osvježi karticu na mjestu kako bi se prikazala slika kartice. U drugom slučaju okrenuta kartica je druga kartica koju je korisnik okrenuo. Tada je potrebno provjeriti podudaraju li se kartice ili ne, što programski kod 4.6.4. prikazuje.

```
private func checkForMatches(for secondFillpedCardIndex: Int){
        let cardOne = self.output.gameData.collectionData[firstFlippedCardIndex!].data
        let cardTwo = self.output.gameData.collectionData[secondFillpedCardIndex].data

        if cardOne.imageName == cardTwo.imageName {
            cardsMatched(cardOne: cardOne,
                         cardTwo: cardTwo,
                         firstCardIndex: firstFlippedCardIndex!,
                         secondCardIndex: secondFillpedCardIndex)
            checkIfTheGameEnded()
            changeOptionsData()
        }else{
            cardsDontMatch(cardOne: cardOne,
                           cardTwo: cardTwo,
                           firstCardIndex: firstFlippedCardIndex!,
                           secondCardIndex: secondFillpedCardIndex)

        }
        finishMove(with: secondFillpedCardIndex)
    }
```

Programski kod 4.6.2. Uparivanje kartica

Funkcija *checkForMatches(for scondFlippedCardIndex: Int)* provjerava podudaraju li se kartice te na temelju rezultata radi odgovarajuće akcije. Ukoliko se kartice podudaraju potrebno je promijeniti njihov status i naznačiti da se kartice trebaju maknuti iz pogleda, provjeriti jesu li

sve kartice uparene te promijeniti broj spojenih parova za određenog igrača. Ukoliko se kartice ne podudaraju, potrebno im je promijeniti status i provjeriti je li potrebno promijeniti koji je igrač na potezu. Na kraju funkcije potrebno je završiti potez, odnosno javiti pogledu da osvježi podatke i time okrenuti kartice (ukoliko nisu uparene) ili ih ukloniti iz pogleda (ukoliko su kartice uparene) te je potrebno globalnoj varijabli za utvrđivanje je li okrenuta prva ili druga kartica pridružiti vrijednost nil.

4.7. MPCManager

MPCManager služi kao pomoćna klasa za upravljanje komunikacije između uređaja. Klasa je napravljena kao *Singelton*, kako bi postojala samo jedna instanca menadžera u aplikaciji. Programski kod 4.7.1. prikazuje inicijalizator menadžera. Kao što je već napisano u teoretskom djelu, *MultipeerConnectivity* biblioteka zahtjeva da se implementiraju određeni objekti te da se postave određeni protokoli. U initu možemo vidjeti njihovu implementaciju.

```
public override init(){
    peerID = MCPeerID(displayName: UIDevice.current.name)
    session = MCSession(peer: peerID, securityIdentity: nil, encryptionPreference: .none)
    advetiser = MCNearbyServiceAdvertiser(peer: peerID, discoveryInfo: nil, serviceType: serviceType)
    browser = MCNearbyServiceBrowser(peer: peerID, serviceType: serviceType)
    super.init()
    advetiser.delegate = self
    browser.delegate = self
    session.delegate = self
}
```

Programski kod 4.7.1. Init menadžera

Za komunikaciju između uređaja koristimo protokole koji se nalaze u *MultipeerConnectivity* biblioteci. Za praćenje status povezanosti uređaja i za primanje podataka unutar sesije koristimo *MCSessionDelegate*. Programski kod 4.7.2. prikazuje implementaciju funkcija koje nam to omogućuju.

```
extension MPCManager: MCSessionDelegate{
  public func session(_ session: MCSession,
                      peer peerID: MCPeerID,
                      didChange state: MCSessionState) {
    switch state{
    case MCSessionState.connected:
      print("Connected to session: \(session)")
      DispatchQueue.main.async{[unowned self] in
        self.managerDelegate?.connectedWithPeer()
      }
```

```
        case MCSessionState.connecting:
            print("Connecting to session: \(session)")

        default:
            print("Did not connect to session: \(session)")
        }
    }

    public func session(_ session: MCSession,
                        didReceive data: Data,
                        fromPeer peerID: MCPeerID) {
        DispatchQueue.main.async{[unowned self] in
            if let decodedData: GameData = self.decodeRecivedData(jsonData: data){
                self.joinDelegate?.joinGame(theme: decodedData.theme, numberOfPairs: decodedData.numberOfPairs, cards:
decodedData.cards)
            } else if let decodedData: MoveData = self.decodeRecivedData(jsonData: data){
                self.moveDelegate?.cardPickPublisher.onNext(decodedData.index)
            }else if let decodedData: TurnData = self.decodeRecivedData(jsonData: data){
                self.turnDelegate?.itIsYourTurn(decodedData.yourTurn)
            }else if let _: QuitData = self.decodeRecivedData(jsonData: data){
                self.quiteDelegate?.quitGame()
            }
        }
    }
```

Sl.4.7.2. MCSessionDelegate

Sesija može biti u jedno od tri stanja, a to su: *spojeno, u procesu spajanja* i *nije spojeno*. Kada je sesija u stanju spojeno, implementacijom protokola šalje se informacija o uparenosti uređaja. Implementirali smo funkciju koja prima podatke koje je poslao drugi uređaj. Ovisno o klasi u koju su se raščlanili primljeni podaci, poziva se određeni protokol.

Drugi protokol koji se implementira je *MCNearbyServiceAdvertiserDelegate*. Protokol služi kako bi mogli napraviti akciju kada korisnik dobije pozivnicu s uređaja domaćina. Implementaciju možemo vidjeti na programskom kodu 4.7.3.

```
extension MPCManager: MCNearbyServiceAdvertiserDelegate{
    public func advertiser(_ advertiser: MCNearbyServiceAdvertiser,
                           didReceiveInvitationFromPeer peerID: MCPeerID,
                           withContext context: Data?,
                           invitationHandler: @escaping (Bool, MCSession?) -> Void) {
        NSLog("%@", "didReceiveInvitationFromPeer \(peerID)")
        self.invitationHandler = invitationHandler
        joinDelegate?.invitationWasReceived(fromPeer: peerID.displayName)
    }
}
```

Programski kod 4.7.2. *MCNearbyServiceAdvertiserDelegate*

Zadnji implementirani protokol služi kao pomoć pri pronalasku svih uređaja s kojima se može uređaj domaćin spojiti, odnosno ukoliko se izgubi povezanost s nekim uređajem.

5. RAD APLIKACIJE

5.1. Logička igra

Prilikom pokretanja aplikacija na zaslonu uređaja pokazuje se *Splash screen* koji se sastoji od pozadinske boje i imena aplikacije. Nakon što se aplikacija učitala, korisniku se prikazuje izbornik za odabir načina igre, to jest hoće li igrati igru u više igrača ili će igrati sam. Također, to je prvi zaslon s kojim korisnik može komunicirati. Prikaz ovih zaslona možemo vidjeti na slici 5.1.1.

Sl.5.1.1. Prvi pogled na aplikaciju

Ako korisnik odabere da igra sam, otvara mu se zaslon na kojem mora izabrati opcije igre. Korisnik bira broj parova koje mora spojiti, te temu igre. Nakon odabira može započeti s igrom. Ukoliko korisnik odabere igru u više igrača, pokreće se izbornik u kojem se korisnik može priključiti u igru, te mogućnost odabrati da on bude domaćin igre. Ako se odluči za domaćinstvo, otvara mu se zaslon s opcijama igre, te popisom svih mogućih uređaja protiv kojih korisnik može igrati.

Sl.5.1.2. Prikaz različitih izbornika aplikacije

Na prvom zaslonu na slici 5.1.2. vidimo da je *button* za početak igre onesposobljen dok korisnik ne izabere opcije igre, odnosno dok ne izabere temu i broj parova. Aplikacija ima tri teme kartica, a to su: *obrazovna*, *znamenitosti* i *kupovina* koje se mogu prikazivati u osam ili osamnaest parova. Kada je riječ o igri u više igrača, umjesto da je element onesposobljen, korisniku se prikaže upozorenje kada pokuša pozvati drugi uređaj na povezivanje, što možemo vidjeti na slici 5.1.3. Na slici također možemo vidjeti kako izgleda poziv za spajanje na sesiju.

Sl.5.1.4. Prikaz pozivnice i upozorenja da korisnik treba odabrati opcije igre

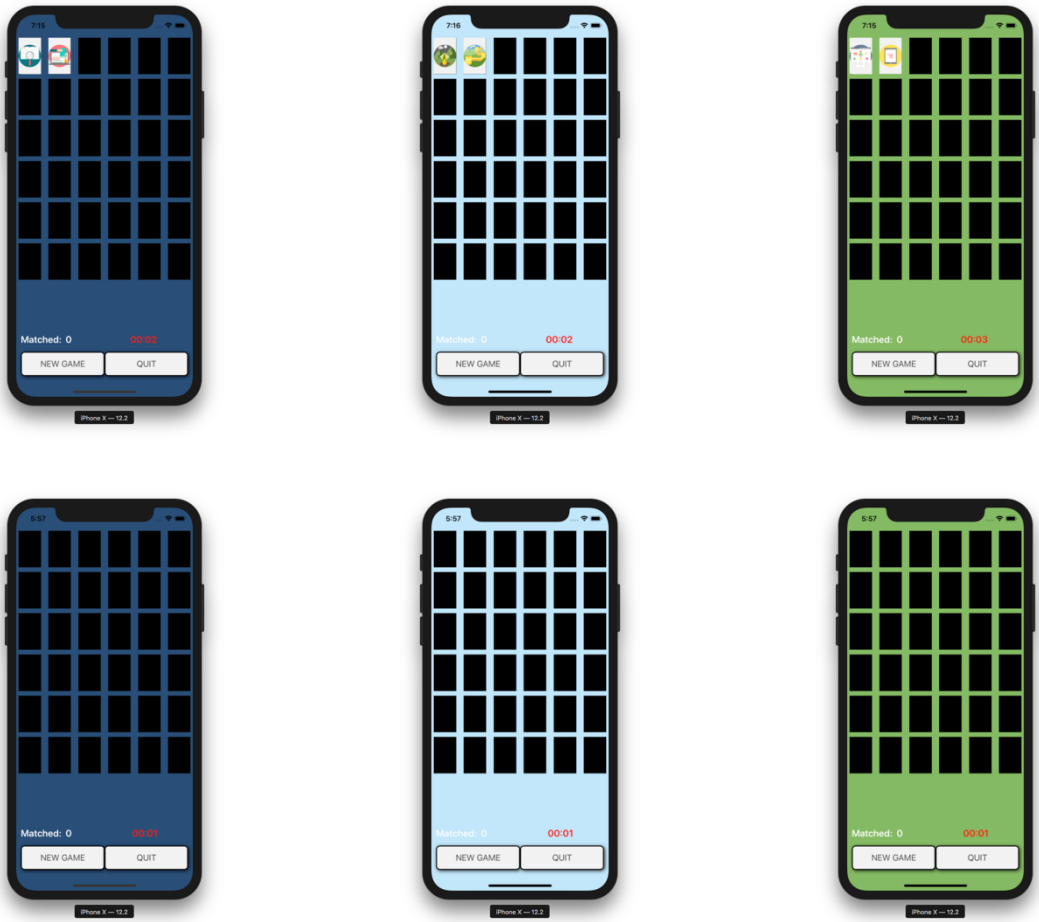
Kada korisnik, odnosno korisnici, pokrenu igru, prikazuje se *collectionView* u kojem su nasumično postavljene kartice s pogledom prema dolje, što možemo vidjeti na slici 5.1.3. Aplikacija podržava isključivo portretski način rada kako bi se izbjegli problemi koji bi korisnici imali ukoliko bi se podržavao pejzažni način rada.

Sl.5.1.4. Prikaz različitih tema logičke igre

Korisnik dodirom na karticu okreće ju i prikazuje kartičnu sliku, odnosno postavlja karticu s pogledom prema gore, što vidimo na slici 5.1.4.

Sl.5.1.5. Korisnikov prvi potez

Na slici 5.1.5. vidimo primjer u kojem korisnik, odabirom druge kartice, nije pronašao jednak par. Aplikacija u tome slučaju okreće obje kartice prema dolje.



Sl.5.1.6. Kartice nisu jednak

Na slici 5.1.4. možemo vidjeti primjer u kojem je korisnik ponašao jednak par kartica. U tome slučaju će aplikacija ukloniti te kartice iz igre.

Sl.5.1.7. Uspješno sparen par kartica

Kao što je prikazano na svim slikama, u svakom trenutku možemo započeti s novom igrom te izaći iz trenutne igre. Također možemo pratiti koliko smo parova uparili i koliko je vremena proteklo od početka igre.

6. ZAKLJUČAK

Uspješnom izradom diplomskog rada izrađena je mobilan aplikacija za igranje logičke igre memory s različitim mogućnostima kao što su igranje igre u više igrača, odabir teme igre i druge. Aplikacija je napisana u programskom jeziku Swift za operativni sustav iOS te je iz toga razloga dostupna isključivo korisnicima koji posjeduju Apple pametni uređaj. Logička igra memory je jednostavna logička igra pomoću koje se jačaju kognitivne sposobnosti igrača.

Za izradu aplikacije korištene su najnovije tehnologije koje su nam omogućile brzu i jednostavnu izradu aplikacije. Pomoću koordinatora upravljamo navigacijom aplikacije, odnosno upravljamo koji će se zaslon prikazati uzevši u obzir korisnikovu komunikaciju s korisničkim sučeljem. MVVM arhitektura nam je omogućila da razdvojimo poslovnu logiku od logike za postavljanje korisničkog sučelja. Takvim načinom programiranja omogućili smo testiranje poslovne logike neovisno o logici za postavljanje korisničkog sučelja te na jednostavniji način možemo izdvojiti programski kod koji će se koristiti više puta u aplikaciji. Reaktivni način programiranja omogućio nam je da napravimo protoke podataka na koje se mogu pretplatiti svi kojima su ti podaci potrebni za realizaciju zaslona.

Model kartice, repozitorij za dohvaćanje kartica te logika igre su najbitnije komponente ove aplikacije. Model kartice govori pogledu u kojem je stanju kartica te kako se ona treba prikazati korisniku. U repozitoriju pravimo polje nasumično poredanih kartica koje korisnik mora spojiti. Logika igre je poslovna logika koja se nalazi u viewModelu igre. U njemu se nalazi kod za manipulaciju modela kartice uzevši u obzir korisnikovu akciju. Kako bi omogućili korisniku igranje igre protiv korisnika u neposrednoj blizini koristimo biblioteku MultipeerConnectivity. Biblioteka nam omogućuje stvaranje sesije na koju se mogu povezati uređaji te komunikaciju između povezanih uređaja.

Kako bi aplikacija bila cjelovitija i funkcionalnija, odnosno kako bi bila za širu primjenu, potrebno je napraviti neke promjene. Najjednostavnija i najveća nadogradnja bi bila u području dizajna korisničkog sučelja. Kao poboljšanje funkcionalnosti ove aplikacije može se proširiti model kartice s brojačem koliko je kartica puta okrenuta što bi omogućilo postavljanje naprednijeg sustava za praćenje rezultata. Kako bi omogućili igranje igre preko interneta potrebno je implementirati drugu biblioteku za povezivanje kao što je GameCenter.

7. LITERATURA

[1] Raywenderlich.com tutorial team: Swift Apprentice; Drugo izdanje, autorsko pravo © 2016 Razeware LLC., 2016.

[2] https://developer.apple.com/ (08.09.2019.)

[3] Raywenderlich.com tutorial team: RxSwift: Reactive Programming with Swift, autorsko pravo © 2017 Razeware LLC., 2017.

[4] https://medium.com (08.09.2019.)

[5] https://www.raywenderlich.com (08.09.2019.)

[6] http://stackoverflow.com/ (08.09.2019.)

[7] M.Kiš: Englesko - Hrvatski Hrvatsko - Engleski Informatički rječnik; Drugo izdanje, 2002.

8. SAŽETAK

Tema ovog diplomskog rada je izrada mobilne aplikacije logičke igre memory. Glavni zadatak aplikacije je igranje igre u više igrača ili protiv samoga sebe. Aplikacija je napravljena za iOS operacijski sustava pisana programskim jezikom Swift.

Ovaj rad sastoji se od dva dijela. Prvi dio opisuje korištene tehnologije i daje njihova teorijska objašnjenja koja daju odgovor na pitanje iz kojeg razloga su korištene. Drugi dio daje odgovor na pitanje kako aplikacija radi, kako ide slijed aplikacije i opisuje njenu implementaciju i korištenje.

Aplikacija je uspješno razvijena korištenjem najnovijih tehnologija kao što su MVVM, koordinatori, reaktivno programiranje i slično.

Ključne riječi: iOS platforma, logička igra, Swift, MVVM, reaktivno programiranje

ABSTRACT

iOS APPLICATION OF THE MEMORY LOGIC GAME

The theme of this thesis is the development of a mobile application of the logical game memory. The main task of the application is to play a multiplayer or a singleplayer game. The application is made for iOS operating systems written in the Swift programming language.

This paper consists of two parts. The first part describes the technologies used and gives their theoretical explanations that explains why they were used. The second part explains the question of how the application works, describes the flow of the application and describes its implementation and use.

The application has been successfully developed using the latest technologies such as MVVM, coordinators, reactive programming and the like.

Key words: iOS platform, logic game, Swift, MVVM, reactive programming

9. ŽIVOTOPIS

Luka Lovretić rođen je 15.07.1991. u Osijeku. Osnovnu školu je pohađao u Osnovnoj školi Mladost u Osijeku. Završio je Isusovačku klasičnu gimnaziju s pravom javnosti u Osijeku. Nakon završene srednje škole upisuje Preddiplomski studij računarstva na Elektrotehničkom fakultetu Osijek i završava ga 2015. godine. Po završetku preddiplomskoga studija, upisuje diplomski studij na Fakultetu elektrotehnike, računarstva i informacijskih tehnologija. Izvrsno govori engleski jezik te se služi njemačkim jezikom. Poznaje rad u Microsoft Office alatima, Apple iWork alatima te se služi programskim jezicima kao što su C, C++, Swift i Objektno usmjerenim nadskupom programskog jezika (Objective - C).

10. PRILOZI

10.1. Prilog P10.1. – Kompletan programski kod

- Nalazi se na CD-u i https://github.com/llovretic/Memory