Univerzitet u Beogradu

Matematički fakultet

Master rad

Principi programiranja aplikacija za iOS platformu u programskom jeziku Objective-C

Student:

Ivan Milić

Mentor:

dr Filip Marić

Beograd, 2017.

Mentor: **prof. dr Filip Marić**

Matematički fakultet, Univerzitet u Beogradu

Članovi komisije: **prof. dr Saša Malkov**

Matematički fakultet, Univerzitet u Beogradu

prof. dr Miroslav Marić

Matematički fakultet, Univerzitet u Beogradu

Datum odbrane: ______________________________

Principi programiranja aplikacija za iOS platformu u programskom jeziku Objective-C

Apstrakt - Operativni sistem iOS je drugi najpopularniji operativni sistem za mobilne uređaje na svetu sa preko 2,2 miliona aplikacija u prodavnici aplikacija kompanije *Apple* (engl. *App Store*). Osnovni jezik za razvoj aplikacija za iOS je Objective-C. Ovaj rad se bavi predstavljanjem jezika Objective-C, arhitekture operativnog sistema iOS kao i prikazom najčešće korišćenih radnih okvira iOS i koncepata programiranja mobilnih aplikacija za platformu iOS. Takođe, u radu je predstavljena implementacija mobilne aplikacije za prikazivanje podataka o vremenskim uslovima izabrane lokacije. Ova aplikacija razvijena je u cilju prezentovanja upotrebe velikog broja koncepata i radnih okvira za razvoj aplikacija za platformu iOS.

Ključne reči: pametni mobilni telefon, operativni sistem, iOS, aplikacija, razvoj, Objective-C, radni okvir, koncepti

Mobile application development principles for iOS platform in the programming language Objective-C

Abstract - The second most popular mobile operating system globally is iOS operating system featuring over 2.2 million apps in the *Apple*'s App Store. The main programming language used for developing iOS apps is Objective-C. This work focuses on presenting the language Objective-C, the iOS architecture as well as demonstrating the most used iOS frameworks and mobile app development concepts for iOS platform. Also, this work shows the implementation of a mobile app which displays weather data for the selected location. This app is developed in order to demonstrate the use of many concepts and frameworks for developing iOS applications.

Keywords: smartphone, operating system, iOS, application, development, Objective-C, framework, concepts

Sadržaj

1 Uvod 5
2 Operativni sistem iOS 6
2.1 Opis sistema iOS 6
2.2 Arhitektura sistema iOS 6
2.2.1 *Sloj Cocoa Touch* 7
2.2.2 Sloj *Media* 9
2.2.3 Sloj *Core Services* 10
2.2.4 Sloj *Core OS* 13
2.3 Istorijat verzija sistema iOS 14
2.4 Razvojno okruženje 15
3 Programski jezik Objective-C 18
3.1 Objekti i poruke 18
3.2 Klase 21
3.3 Svojstva 23
3.4 Protokoli 24
3.5 Kategorije i ekstenzije 26
3.6 Blokovi 28
4 Radni okvir iOS 29
4.1 Radni okvir *Foundation* 29
4.2 Redovi i tehnologija Grand Central Dispatch 31
4.3 Životni ciklus aplikacije 32
4.4 Upravljanje memorijom 33
4.5 Obrasci projektovanja 34
4.5.1 Obrazac Model-pogled-kontroler 34
4.5.2 Obrazac Unikat 36
4.5.3 Obrazac Fasada 36
4.5.4 Obrazac Dekorater 37
4.5.5 Obrazac Adapter 38
4.5.6 Obrazac Posmatrač 40

4.5.7 Obrazac Podsetnik 42
4.5.8 Obrazac Komanda 45
5 Implementacija aplikacije „WeatherMaster” 48
5.1 Opis aplikacije 48
5.2 Klase sloja model 53
5.3 Klase sloja pogled 53
5.3.1 Pogled DayForecastView 54
5.3.2 *Komponenta aplikacije Storyboard* 56
5.4 Klase sloja kontroler 57
5.4.1 Klasa LoadingViewController 57
5.4.2 Klasa WeatherViewController 59
5.4.3 Klasa SavedLocationsTableViewController 70
6 Zaključak 78
Literatura 79

1 Uvod

U poslednje četiri godine zabeležen je nagli porast prodaje „pametnih” mobilnih telefona. Prema statistikama kompanije Gartner, u 2016. godini prodato je 1,5 milijarde pametnih mobilnih telefona od kojih je preko 216 miliona jedinica proizvela kompanija *Apple* [1].

Mobilni telefoni *Apple* za rad koriste operativni sistem iOS koji zauzima drugo mesto na rang listi popularnosti po procentualnom udelu u tržištu mobilnih operativnih sistema, posle operativnog sistema Android. Uređaji kompanije *Apple* se odlikuju odličnim performansama u radu zbog dobre optimizovanosti operativnog sistema iOS za relativno mali skup uređaja kompanije *Apple*. Kako bi se održala konzistentnost performansi uređaja, ovakav vid optimizacije zahteva samo mala unapređenja performansi hardvera između različitih generacija proizvoda *Apple*. Performanse kao i jednostavnost upotrebe uređaja su glavni razlozi održavanja popularnosti uređaja kompanije *Apple* još od predstavljanja svog prvog pametnog mobilnog telefona *iPhone* u 2007. godini. Zbog svoje popularnosti, porast prodaje mobilnih telefona *Apple* prati opšti porast prodaje mobilnih telefona u svetu.

Porast prodaje mobilnih telefona koji koriste operativni sistem iOS uslovljava porast potražnje za razvijanjem mobilnih aplikacija za platformu iOS što predstavlja motivaciju za izučavanje ovih tehnologija i pisanje ovog rada.

U drugom poglavlju ovog rada biće opisan operativni sistem iOS kroz pregled slojeva arhitekture sistema, radnih okvira (engl. *framework*) i funkcionalnosti ovih slojeva kao i kratki istorijat verzija sistema iOS.

U trećem poglavlju biće predstavljen programski jezik Objective-C kao primarni programski jezik korišćen za razvoj aplikacija za platformu iOS. U ovom poglavlju biće opisane mogućnosti koje Objective-C donosi u odnosu na jezik C čiji je nadskup kao i specifične jezičke strukture ovog programskog jezika.

U četvrtom poglavlju biće prikazane esencijalne klase radnog okvira iOS za razvoj aplikacija, životni ciklus aplikacija, sistem upravljanja memorijom kao i obrasci za projektovanje na koje se radni okvir iOS oslanja u velikoj meri prlikom implementacije aplikacija.

U petom poglavlju biće predstavljena aplikacija „WeatherMaster” koja je razvijena za prikaz upotrebe velikog broja koncepata programiranja mobilnih aplikacija za platformu iOS o kojima je bilo reči u prethodnim poglavljima.

U šestom poglavlju, nakon obrađene teme, na kraju samog rada biće iznet zaključak.

2 Operativni sistem iOS

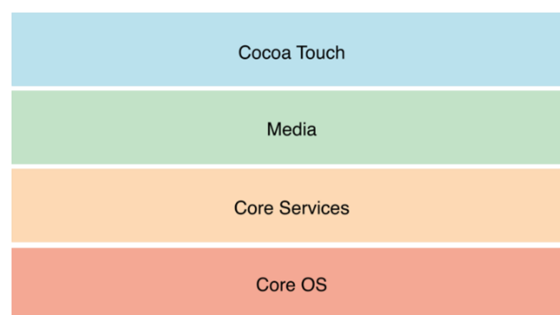
U ovom poglavlju će biti predstavljen operativni sistem iOS, njegova arhitektura i istorijat verzija sistema.

2.1 Opis sistema iOS

iOS je mobilni operativni sistem zatvorenog kôda razvijen od strane kompanije *Apple*. Ovaj operativni sistem je prvobitno razvijen pod nazivom *iPhone OS* za prvi *iPhone* uređaj i predstavljen je na *Macworld* konferenciji 9. Januara 2007. godine. Nakon proširenja upotrebe sistema i na drugim serijama uređaja *Apple* sem *iPhone* uređaja, u junu 2010. godine dolazi do promene naziva sistema u iOS. Danas, iOS se koristi i dalje isključivo na hardveru kompanije *Apple* i to na tri serije uređaja: *iPhone*, *iPad* i *iPodTouch* [2].

2.2 Arhitektura sistema iOS

iOS je baziran na *Unix* operativnom sistemu *Darwin* i kao takav pripada grupi sistema srodnih *Unix*-u. Sistem ima na niskom nivou osnovu napisanu u programskom jeziku C, a klase programskog jezika Objective-C predstavljaju omotač osnove. iOS predstavlja posrednika u komunikaciji između hardvera i aplikacija. Aplikacije ne pristupaju direktno hardveru već komuniciraju sa sistemskim interfejsima koji olakšavaju pisanje aplikacija za uređaje koji imaju različite hardverske sposobnosti. Arhitektura sistema iOS je slojevita arhitektura, sačinjena iz 4 sloja: *Cocoa Touch*, *Media*, *Core Services* i *Core OS* (slika 2.1).



Slika 2.1 Slojevi arhitekture sistema iOS

Sistemski interfejsi iz ovih slojeva su predstavljeni u okviru specijalnih paketa koji se nazivaju „radni okviri”. Ovi paketi sadrže dinamičku deljenu biblioteku i sve resurse potrebne za podršku te biblioteke poput slika, datoteka zaglavlja i pomoćnih aplikacija. Radni okviri višeg nivoa omogućavaju lakše pisanje kôda u odnosu na radne okvire nižeg nivoa jer pružaju objektno-orijentisanu apstrakciju u odnosu na konstrukte nižeg nivoa. Kada određena funkcionalnost sistema nije podržana radnim okvirima višeg nivoa, neophodno je koristiti radne okvire iz nižeg nivoa arhitekture sistema [3].

2.2.1 *Sloj Cocoa Touch*

Sloj *Cocoa Touch*, kao najviši sloj arhitekture sistema iOS, pruža najveći broj funkcionalnosti i radnih okvira koje iOS programer koristi u okviru razvijanja aplikacija.

Neke od ključnih funkcionalnosti koje omogućava ovaj sloj su:

- *App Extension*
    - proširivanje mogućnosti operativnog sistema iOS dodavanjem novih opcija u okviru sistemskog menija za deljenje u drugim aplikacijama; npr. iz sistemske aplikacije za pregled slika, moguće je nad slikom direktno primeniti filter obrade kojim razvijena aplikacija proširuje sistem
- *Handoff*
    - kontinuirano izvršavanje jedne aktivnosti aplikacije na više uređaja ukoliko su prijavljeni istim nalogom; npr. u aplikaciji za pregled Web stranica, korisnik može započeti aktivnost pregleda stranice na jednom uređaju i nastaviti pregled iste stranice na drugom uređaju
- *Document Picker*
    - pristup dokumentima izvan okvira aplikacije; korisnici mogu da pregledaju i menjaju dokumente „iz oblaka” kroz različite aplikacije i time je olakšano deljenje dokumenata između aplikacija
- *AirDrop*
    - mogućnost deljenja podataka i datoteka iz aplikacije sa obližnjim uređajima korišćenjem tehnologije *Bluetooth*
- *TextKit*
    - upravljanje i stilizacija teksta
- *UIKit Dynamics*

    - definisanje animacija i dinamičkih ponašanja vizuelnih elemenata koji simuliraju fiziku stvarnog sveta poput uticaja sile gravitacije, sudara, guranja itd.
- *Multitasking*
    - mogućnost prelaska aplikacije u pozadinski režim izvršavanja nakon pokretanja druge aplikacije; npr. izvršavanje glasovnih instrukcija iz pozadinskog režima rada aplikacije za navigaciju dok korisnik koristi drugu aplikaciju
- *Auto Layout*
    - korišćenje ograničenja za izgrađivanje dinamičkih korisničkih interfejsa koji odgovaraju različitim dimenzijama ekrana (više o ovome u poglavlju 5.3.1)
- *Storyboard*
    - vizuelna reprezentacija korisničkog interfejsa aplikacije koji obuhvata sve ekrane aplikacije i njihove povezanosti (više o ovome u poglavlju 5.3.2)
- *UI State Preservation*
    - čuvanje stanja korisničkog interfejsa aplikacije prilikom prelaska aplikacije u pozadinski režim rada i vraćanje aplikacije na prethodno sačuvano stanje prilikom ponovnog pokretanja aplikacije
- *Apple Push Notification Service*
    - udaljeni (engl. *remote*) servis koji korisniku prosleđuje na uređaj dinamička obaveštenja o aplikaciji čak i kada ona nije pokrenuta; ovaj servis dobija informacije koje se prosleđuju određenim korisnicima preko servera koji je napravljen od strane programera
- *Local Notifications*
    - pravljenje lokalnih obaveštenja u aplikaciji i prosleđivanje obaveštenja sistemu koji upravlja isporučivanjem ovih obaveštenja korisniku kada aplikacija nije pokrenuta
- *Gesture Recognizers*
    - prepoznavanje i reagovanje na gestove korisnika u okviru korisničkog interfejsa aplikacije
- *Standard System View Controllers*
    - korišćenje standardnih sistemskih kontrolera pogleda za obavljanje učestalih aktivnosti; korišćenjem ovih kontrolera pogleda podstiče se održavanje konzistentnog korisničkog iskustva

Radni okviri koji pripadaju ovom sloju su:

- *Address Book UI*
    - pravljenje novih kontakata, menjanje i odabir postojećih kontakata
- *EventKit UI*
    - pregled i menjanje događaja kalendara
- *GameKit*
    - povezivanje sa servisom *Game Center* i deljenje korisnikovih informacija u okviru igre sa ostalim korisnicima
- *iAd*
    - predstavljanje i integracija reklama u okviru korisničkog interfejsa aplikacije
- *MapKit*
    - prikazivanje mape i interakcija sa mapom u okviru aplikacije
- *MessageUI*
    - pravljenje SMS ili *email* poruka iz aplikacije
- *Notification Center*
    - pravljenje dodataka (engl. *widgets*) u okviru centra za obaveštenja operativnog sistema iOS
- *PushKit*
    - podrška za *push* obaveštenja korisniku o dolazećim pozivima u *VoIP* aplikacijama koje rade u pozadinskom režimu; na ovaj način aplikacija ne mora da održava trajnu konekciju sa serverom kada nije aktivna i uređaj troši manje energije
- *UIKit*
    - upravljanje korisničkim interfejsom aplikacije putem arhitekture prozora i pogleda, reagovanje na događaje i model aplikacije neophodan za interakciju sa operativnim sistemom [4]

Neke od navedenih funkcionalnosti i radnih okvira ovog sloja će biti detaljnije objašnjene u narednim poglavljima. U nastavku ovog poglavlja ćemo ukratko opisati ostale slojeve arhitekture sistema iOS.

2.2.2 Sloj *Media*

Na ovom sloju se nalaze komponente u vidu grafičkih, audio i video elemenata. Radni okviri i tehnologije za rad sa ovakvim multimedijalnim sadržajima predstavljeni su u tabeli 2.1 kategorisani po tipu sadržaja kojima upravljaju.

Tabela 2.1 Kategorizacija radnih okvira sloja Media

Grafika	Audio	Video
Core Graphics	**OpenAL**	**AVKit**
Quartz Core	**AV Foundation**	**AV Foundation**
Core Image	**Core Audio**	**Core Media**
Assets Library	**CoreAudioKit**	**Assets Library**
OpenGL ES	**Media Player**	**Media Player**
GLKit		
Metal		
Core Text		
ImageI/O		
Photos/PhotosUI		

Grafički radni okviri omogućavaju programerima da prave aplikacije koje sadrže grafičke interfejse, animacije, komponente za čitanje i pisanje slika i pristup nativnim vizuelnim elementima uređaja. Audio radni okviri omogućavaju puštanje, snimanje i integraciju zvuka u okviru razvijanih aplikacija. Video radni okviri pružaju programerima interfejse za prezentovanje, puštanje, snimanje video snimaka kao i pristup video snimcima već sačuvanim u biblioteci samog uređaja [5].

2.2.3 Sloj *Core Services*

Ovaj sloj sadrži fundamentalne sistemske servise koje koriste aplikacije. Iako se ne koriste svi servisi direktno, mnogi delovi sistema iz viših slojeva arhitekture su izgrađeni nad njima.

Neke od ključnih funkcionalnosti koje omogućava ovaj sloj su:

- *Peer-to-Peer* servisi
    - otkrivanje servisa koje pružaju uređaji u blizini i komunikacija sa tim servisima korišćenjem infrastrukture Wi-Fi tj. bežičnih mreža i tehnologije *Bluetooth*; povezani uređaji bezbedno i direktno prenose poruke, tokove i datoteke bez posredničkih Web servisa
- *iCloud Storage*
    - čuvanje dokumenata i podešavanja aplikacije „u oblaku” čime se omogućava njihov pristup sa svih uređaja bez potrebe za sinhronizacijom ili eksplicitnim prenosom dokumenata
- *Block* konstrukcije

- izdvajanje segmenata kôda i njihovo prosleđivanje funkcijama i metodama u jeziku Objective-C (više o ovome u poglavlju 3.6)
- Zaštita podataka
  - zaštita podataka aplikacije ugrađenom enkripcijom sistema u slučaju zaključavanja uređaja i generisanje ključa za dekripciju prilikom otključavanja uređaja
- Deljenje dokumenata
  - dostupnost određenih dokumenata aplikacije korisniku preko *Documents* direktorijuma i mogućnost ručnog dodavanja dokumenata u isti direktorijum aplikacije koristeći program *iTunes* na računaru
- *Grand Central Dispatch*
  - tehnologija koja pruža funkcije koje omogućavaju konkurentno programiranje aplikacije (više o ovome u poglavlju 4.2)
- Kupovina unutar aplikacije
  - kupovina sadržaja vezanog za aplikaciju unutar aplikacije i izvršavanje finansijskih transakcija koristeći korisnikov nalog pri komunikaciji sa servisom prodavnice aplikacija
- *SQLite*
  - pravljenje lokalne SQL baze podataka aplikacije koja u određenim datotekama aplikacije čuva podatke u vidu tabela i redova baze i omogućava izvršavanje SQL upita nad tim podacima u kôdu
- *XML* podrška
  - pravljenje XML sadržaja i upravljanje njime

Radni okviri koji pripadaju ovom sloju su:

- *Accounts*
  - pravljenje, čuvanje i korišćenje objekata koji enkapsuliraju informacije o korisničkim nalozima kako bi se omogućilo prijavljivanje na određeni nalog samo jednom (engl. *single sign-on*) bez potrebe sa ponavljanjem autorizacionog procesa
- *Ad Support*
  - isporučivanje reklama u aplikacijama putem platforme za reklamiranje kompanije *Apple*
- *CFNetwork*
  - rad sa mrežnim protokolima, korišćenje BSD soketa, rad sa HTTP i HTTPS serverima, rad sa FTP serverima itd.
- *CloudKit*

    - prenos podataka između aplikacije i servisa *iCloud* za čuvanje podataka „u oblaku”
- *Core Data*
    - modeliranje složenog „model” sloja aplikacije iz obrasca Model-Pogled-Kontroler (više o ovome u poglavlju 4.5.7.3)
- *Core Foundation*
    - upravljanje primitivnim strukturama podataka preko skupa interfejsa baziranih na programskom jeziku C
- *Core Location*
    - podaci o trenutnoj lokaciji korisnika aplikacije na osnovu GPS-a i interneta
- *Core Motion*
    - podaci o pokretima uređaja na osnovu ugrađenog žiroskopa i akcelerometra
- *Core Telephony*
    - upravljanje telefonskim podacima uređaja u vidu informacija o mobilnom operateru korisnika
- *Foundation*
    - funkcionalnosti radnog okvira *Core Foundation* u programskom jeziku Objective-C
- *HealthKit*
    - unos i čitanje informacija o zdravlju i aktivnostima korisnika uređaja na osnovu podataka dobijenih preko tehnologije *Bluetooth* od eksternih uređaja poput monitora srčane frekvencije, fitnes narukvica itd.
- *HomeKit*
    - povezivanje, komunikacija i upravljanje „pametnim” uređajima doma korisnika aplikacije poput „pametnih” sijalica, termostata, alarma itd.
- *JavaScript Core*
    - omotači standardnih *JavaScript* objekata za evaluaciju *JavaScript* kôda i parsiranje *JSON* podataka
- *Mobile Core Services*
    - korišćenje uniformnih identifikatora tipova (*UTI*) za opisivanje formata datoteka prilikom prenosa podataka između aplikacija i servisa
- *MultipeerConnectivity*
    - upravljanje *Peer-to-Peer* komunikacijom
- *NewsstandKit*
    - pružanje sadržaja novina i magazina kroz *Newsstand* aplikaciju
- *PassKit*
    - integracija podrške za čuvanje kupona, ulaznica za događaje, kartica za popuste i ostalih elektronskih vaučera u okviru aplikacije

- *Quick Look*
    - mogućnost jednostavnog i brzog prikaza sadržaja čestih tipova dokumenata u aplikaciji poput PDF datoteka, RTF dokumenata, slika, Microsoft Office dokumenata itd.
- *Safari Services*
    - mogućnost dodavanja *URL*-a iz aplikacije u korisnikovu listu za čitanje pregledača Safari
- *Social*
    - podrška za integraciju sa društvenim mrežama pomoću obezbeđenih šablona za pravljenje HTTP zahteva; koristi se u kombinaciji sa *Accounts* radnim okvirom
- *StoreKit*
    - implementiranje kupovine unutar aplikacije
- *SystemConfiguration*
    - pristupanje mrežnim konfiguracijama uređaja, provera aktivnosti mobilnog i Wi-Fi signala kao i određivanje dostupnosti udaljenih servera
- *WebKit*
    - prikazivanje *HTML* sadržaja u korisničkom interfejsu aplikacije [6]

2.2.4 Sloj *Core OS*

Ovaj sloj sadrži radne okvire niskog nivoa koji se ne koriste često u aplikacijama direktno ali su tehnologije viših slojeva izgrađene nad njima.

Radni okviri koji čine ovaj sloj su:

- Accelerate
    - kompleksna izračunavanja u vidu digitalnog procesuiranja signala, funkcija linearne algebre i obrade slika
- Core Bluetooth
    - interakcija sa periferalnim uređajima koji koriste *Bluetooth* tehnologiju niske energije (*BLE)* za komunikaciju sa iOS uređajem
- External Accessory
    - komunikacija sa hardverskim dodatkom uređaja koji se povezuje sa iOS uređajem žičano ili putem tehnologije *Bluetooth*
- Generic Security Services
    - skup funkcija i struktura orijentisanih ka bezbednosti aplikacije
- Local Authentication

    - mogućnost zahtevanja autentikacije korisnika u aplikaciji koristeći senzor za otisak prsta (engl. *Touch ID*)
- Network Extension
    - pravljenje, konfiguracija i kontrolisanje tunela virtualne privatne mreže (*VPN*)
- Security
    - korišćenje javnih i privatnih ključeva, sertifikata i polisa poverenja radi osiguravanja podataka kojima upravlja aplikacija

Sistemski nivo obuhvata okruženje jezgra, *driver*-e i *UNIX* interfejse niskog nivoa operativnog sistema. U okviru *LibSystem* biblioteke nalaze se intefejsi bazirani na programskom jeziku C koji na niskom nivou omogućavaju konkurentnost, umrežavanje, pristup sistemu datoteka, ulazno-izlazne operacije, memorijsku alokaciju kao i razne matematičke proračune.

Operativni sistem iOS je prvobitno dizajniran kao operativni sistem koji pokreće binarne datoteke koristeći 32-bitnu arhitekturu, ali je od sedme verzije iOS-a dodata podrška za 64-bitnu arhitekturu usled čega aplikacije mogu raditi brže dok se svi radni okviri mogu koristiti u 32-bitnim kao i u 64-bitnim aplikacijama [7].

2.3 Istorijat verzija sistema iOS

Na konferenciji kompanije *Apple* pod nazivom *WWDC* (*Worldwide Developers Conference*) svake godine se prezentuje novi softver i tehnologije za programere kao i pregled nove verzije operativnog sistema iOS. U tabeli 2.2 prikazan je kratak istorijat verzija sistema iOS, datum izlaska kao i pregled izmena uvedenih u datoj verziji [8].

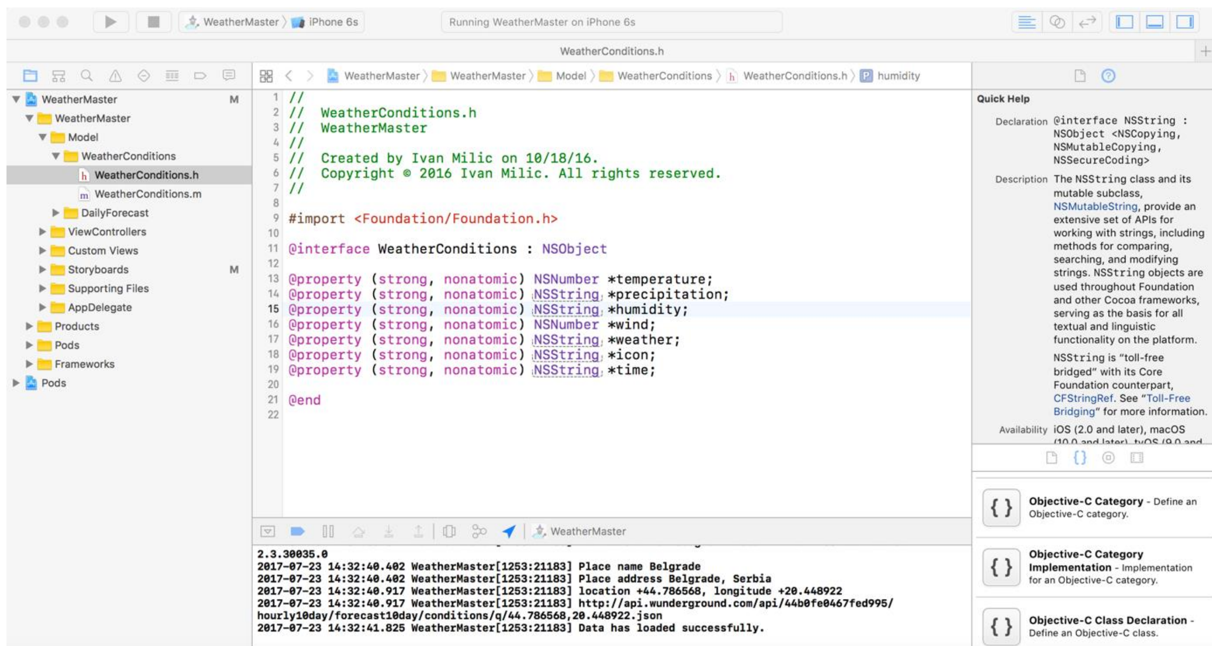
Tabela 2.2 Istorijat iOS verzija

Naziv verzije	Datum izlaska	Pregled izmena
iPhone OS 1	29.06.2007.	osnovne funkcionalnosti, tehnologija reagovanja na dodir, mobilna verzija pregledača *Safari*, *Google* mape
iPhone OS 2	11.07.2008.	prodavnica aplikacija, nativne aplikacije napravljene od strane trećih lica, lokacijski servisi
iPhone OS 3	17.06.2009	funkcije upravljanja tekstom (*cut*, *copy*, *paste*) ograničeno upravljanje glasom, *push* obaveštenja, deljenje internet veze (engl. *tethering*), snimanje video zapisa, kupovina unutar aplikacije
iOS 4	21.06.2010.	promena naziva operativnog sistema iz iPhone OS u iOS, podrška za *multitasking*, direktorijumi na glavnom ekranu uređaja, servis za video pozive *FaceTime*
iOS 5	12.10.2011.	sistem za upravljanje glasovnim komandama *Siri*, prilagođavanje centra za obaveštenja, instaliranje nove verzije sistema iOS bez korišćenja računara, servis *iMessage*, servis *iCloud*
iOS 6	19.09.2012.	servis za mapiranje *Apple Maps*, integracija sa društvenom mrežom *Facebook*, upravljanje platnim karticama, kuponima, vaučerima i kartama kroz *Passbook*
iOS 7	18.09.2013.	potpuni redizajn korisničkog interfejsa sistema iOS, kontrolni centar za upravljanje sistemskim podešavanjima i poboljšanje *multitasking* mogućnosti sistema, interakcija sa senzorom za čitanje otiska prsta (*Touch ID*)
iOS 8	17.09.2014.	pravljenje tastatura i *widget*-a od strane trećih lica, deljenje datoteka između aplikacija i servisa, *Research Kit*, *Health Kit* i*Home Kit* radni okviri, servis *Testflight*
iOS 9	16.09.2015.	unapređivanje sistema *Siri*, muzički servis *Apple Music*, reagovanje na intenzitet pritiska ekrana (engl. *3D Touch*)
iOS 10	13.09.2016.	unapređenje izgleda ekrana zaključanog uređaja, mogućnost razvijanja softvera za servise *Siri*, *iMessages* i *Apple Maps*

2.4 Razvojno okruženje

Integrisano razvojno okruženje (*IDE*) koje se koristi za razvoj i distribuciju aplikacija za operativni sistem iOS naziva se **Xcode** (slika 2.2). Ovaj program može se instalirati samo na operativnom sistemu **macOS** pa je stoga razvoj aplikacija za iOS moguć samo na uređajima kompanije Apple koji podržavaju sistem macOS (Macbook, Macbook Pro, Macbook Air, Mac Pro, Mac Mini i iMac). Osim aplikacija za iOS, u ovom razvojnom okruženju moguć je i razvoj aplikacija za macOS, tvOS i watchOS.

Xcode sadrži glavnu komponentu za uređenje izvornog kôda (engl. *source editor*) sa sintaksnim označavanjem teksta (engl. *syntax highlighting*) i mogućnošću dopunjavanja kôda u toku pisanja (engl. *code completion*).



Slika 2.2 Integrisano razvojno okruženje Xcode

Razvojno okruženje analizira kôd i prikazuje upozorenja i greške na mestima gde se javlja problem u okviru komponente za uređenje izvornog kôda.

Okruženje podržava pisanje aplikacija u programskim jezicima Objective-C kao i Swift pa je kompletna dokumentacija za ova dva jezika ugrađena u Xcode.

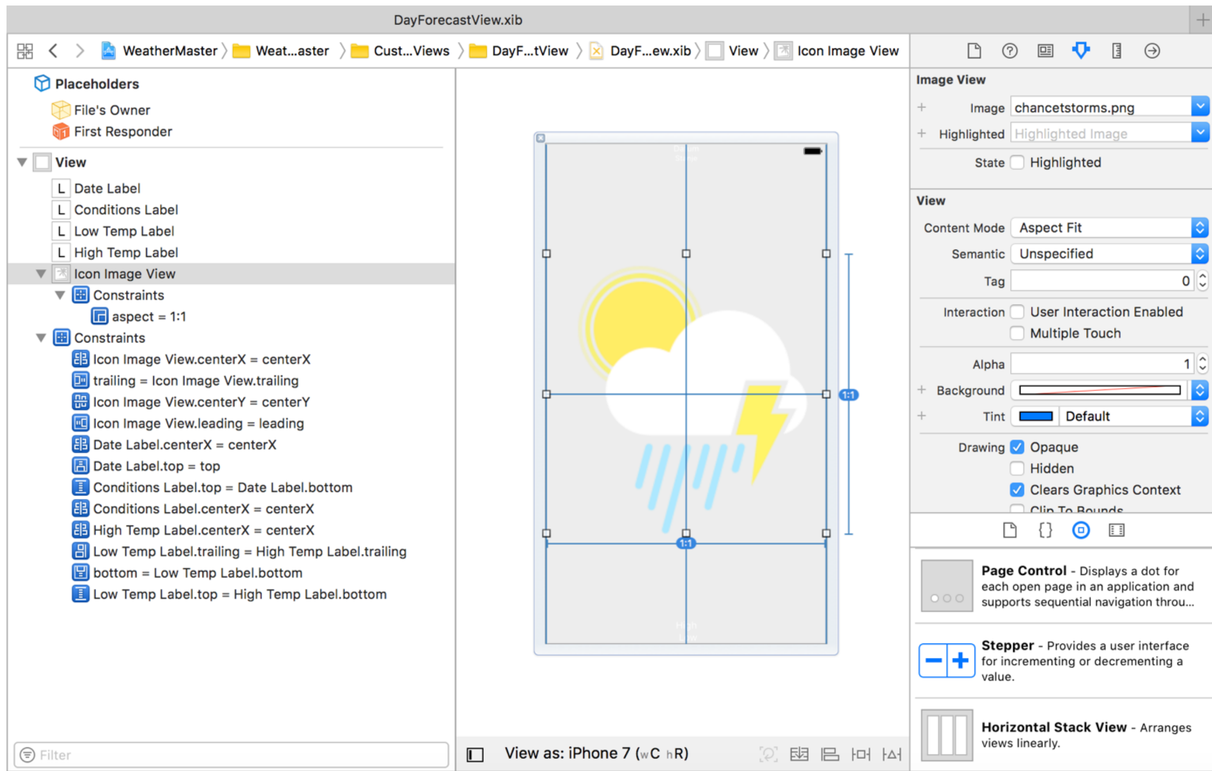
Ugrađenim kompilatorom LLVM vrši se kompilacija kôda nakon čega Xcode omogućava instaliranje, pokretanje i debagovanje aplikacije na povezanim uređajima ili ugrađenom simulatoru. Pre pokretanja simulatora, moguće je odabrati uređaj i verziju operativnog sistema željene simulacije. Namena simulatora je potpuno oponašanje željenog uređaja i mogućnost interakcije programera sa simulatorom u vidu stvarnog uređaja radi efikasnijeg testiranja.

Pomoću ugrađenog alata za debagovanje (engl. *debugger*) i postavljanja tačaka zaustavljanja (engl. *breakpoints*), moguće je čitati i menjati vrednosti promenljivih u toku izvršavanja aplikacije.

Xcode takođe nudi integraciju i punu podršku Subversion i Git sistema kontrole verzija.

U okviru razvojnog okruženja Xcode, ugrađen je alat za izgradnju interfejsa (engl. *Interface Builder*) koji omogućava dizajniranje grafičkog korisničkog interfejsa aplikacije korišćenjem sistema Auto Layout, bez pisanja kôda (slika 2.3). Ovaj alat

takođe omogućava jednostavno vizuelno povezivanje elemenata interfejsa sa odgovarajućim elementima u izvornom kôdu aplikacije.



Slika 2.3 Ugrađeni alat za izgradnju interfejsa

3 Programski jezik Objective-C

Programski jezik Objective-C je objektno-orijentisani programski jezik opšte namene koji je bio primarni programski jezik korišćen za razvoj softvera za iOS i OS X platforme. Danas je ovaj jezik podjednako zastupljen kao i njegov naslednik *Swift*. Tom Love i Brad Cox su početkom 1980-ih godina napravili Objective-C za operativni sistem *NeXTSTEP* od kog potiču iOS i OS X. Objective-C predstavlja nadskup jezika C pružajući mu dinamičnost kao i objektno-orijentisane mogućnosti čija je sintaksa, u vidu slanja poruka objektima, nasleđena od jezika *Smalltalk*. Dinamičnost ovog jezika ogleda se u dinamičkom tipiziranju i dinamičkom uvezivanju što je predstavljeno u narednom poglavlju. Sintaksa jezičkih konstrukcija koje nisu objektno-orijentisane je identična sintaksi ovih konstrukcija u jeziku C, a takođe kôd pisan u jeziku C se može slobodno koristiti u okviru Objective-C datoteka i kompilirati sa Objective-C kompilatorom. U narednim poglavljima biće prikazane mogućnosti koje Objective-C donosi u odnosu na jezik C kao i specifične jezičke strukture ovog programskog jezika [9].

3.1 Objekti i poruke

Nalik strukturama, objekti grupišu podatke. Ti podaci se čuvaju u vidu instancnih promenljivih. Za razliku od struktura, objekat može sadržati i funkcije koje mogu da rade i nad podacima koje objekat sadrži. Te funkcije se nazivaju metode.

Klase opisuju ponašanje i skup atributa koji su zajednički za određeni tip objekata. Ponašanje klase je određeno implementacijom metoda klase, dok su atributi klase određeni instancnim promenljivama. Nasleđivanje je jednostruko odnosno klase u jeziku Objective-C mogu da naslede osobine samo od jedne klase. Natklasa za sve klase je `NSObject`. Kompletna definicija klase sačinjena je iz interfejsa i implementacije klase. **Interfejs** deklariše javna svojstva i metode klase i nalazi se u okviru .h datoteke. U okviru **implementacije** nalazi se kôd koji omogućava rad metoda klase.

U ovom jeziku se ne pozivaju metode već se **prosleđuju poruke** objektima. Cilj poruka se određuje u toku izvršavanja programa. Kod sistema prosleđivanja poruka ne postoji provera tipova i ne postoji garancija da će objekat koji prima poruku zapravo i odgovoriti na poruku. Ukoliko se prosledi poruka kolekciji objekata, odgovoriće samo određeni objekti čija deklaracija sadrži odgovarajuću metodu.

Poruke se najčešće prosleđuju objektima koristeći sintaksu uglastih zagrada (slika 3.1).

```
[tacka ispisiKoordinate];
```

Slika 3.1 Prosleđivanje poruke objektu

Na slici 3.1, sa leve strane nalazi se primalac poruke `tacka` koji predstavlja referencu na objekat. Sa desne strane nalazi se poruka `ispisiKoordinate` koja se šalje primaocu i predstavlja naziv metode objekta primaoca koju objekat treba da izvrši.

Poruka može da sadrži i određeni broj **argumenata** koji predstavljaju parametre metode koja će biti pozvana. Na slici 3.2 metoda `pomnoziKoordinateKoeficijentom`*:* prihvata argument *5*. Treba primetiti da su dve tačke sastavni deo imena metode, kao i da se u naziv metode uključuje i opis navedenog argumenta.

```
[tacka pomnoziKoordinateKoeficijentom:5];
```

Slika 3.2 Slanje poruke sa jednim argumentom

Ime metode koja prihvata više od jednog argumenata se sastoji od više delova. Svaki deo imena metode opisuje odgovarajući argument koji sledi. Svaki deo imena metode se završava sa dve tačke. Pri slanju poruke svaki argument se navodi posle odgovarajućeg dela imena metode.

```
[tacka pomnoziXkoordinatuKoeficijentom:3 iYkoordinatuKoeficijentom:4];
```

Slika 3.3 Slanje poruke sa više argumenata

Na slici 3.3 prikazan je primer slanja poruke sa više argumenata. Poruka se šalje objektu `tacka` koji ima metodu `pomnoziXkoordinatuKoeficijentom: iYkoordinatuKoeficijentom:` koja prihvata dva argumenta u vidu koeficijenata za množenje vrednosti x i y koordinata tačke. Na ovaj način, iz imena metode i poruke koja se šalje objektu, može se zaključiti namena argumenata.

Za pravljenje objekta odgovarajućoj klasi se šalje poruka **`alloc`**. Klasa pravi objekat u memoriji, a zatim se memorijska adresa objekta sačuva u pokazivačkoj promenljivoj. Pokazivač se dalje koristi kao **referenca** na napravljeni objekat. U jeziku Objective-C ne postoji sintaksička razlika između referenci i pokazivača kao što je to slučaj u jeziku C++. Pre upotrebe objekta, potrebno je izvršiti njegovu inicijalizaciju porukom **`init`** (slika 3.4).

```
Tacka *mojaTacka = [Tacka alloc];
[mojaTacka init];
```

Slika 3.4 Pravljenje i inicijalizacija objekta

Pravljenje objekta i njegova inicijalizacija mogu se izvršiti jednom naredbom upotrebom **ugnježdenih poruka** (slika 3.5).

```
Tacka *mojaTacka = [[Tacka alloc] init];
```
Slika 3.5 Pravljenje objekta sa inicijalizacijom pomoću ugnježdenih poruka

Prvo se izvršava slanje poruke iz unutrašnjih zagrada, čime se pravi objekat i dobija njegova adresa, a zatim se tom objektu šalje poruka za inicijalizaciju [10].

Upravljanje memorijom u jeziku Objective-C zasnovano je na **vlasništvu nad objektom** (engl. *object ownership*). Objekat može imati više vlasnika kojima stoji na raspolaganju. Vlasnici mogu da se odreknu objekata, a kada objekat ostane bez ijednog vlasnika, sistem oslobađa memoriju alociranu za njega. Upravljanje memorijom je detaljnije objašnjeno u poglavlju 4.4.

Posebna promenljiva **`self`** u instancnim metodama jezika Objective-C pokazuje na objekat koji upravo izvršava metodu. Ova promenljiva se automatski prosleđuje kao nevidljivi argument svakoj instancnoj metodi. Ključna reč **`super`** može se koristiti isključivo kao primalac poruke i označava izvršavanje implementacije odabrane metode u natklasi.

Ključna reč **`nil`** koristi se za označavanje pokazivača na Objective-C objekte koji nemaju vrednost (nalik konstanti `NULL` za pokazivače u jeziku C). Slanje poruka ovakvim objektima ne dovodi do greške u toku izvršavanja.

Na slici 3.5 prikazan je primer statičkog tipiziranja (engl. *static typing*). Dinamičkim tipiziranjem (engl. ***dynamic typing***) određuje se **tip** objekta u toku izvršavanja (engl. *runtime*), a ne u toku kompilacije (engl. *compile time*) kao kod statičkog tipiziranja. Kada nije poznat tip objekta koji će biti dodeljen promenljivoj, koristi se promenljiva specifičnog tipa **`id`**. Ovoj promenljivoj može biti dodeljen objekat bilo kog tipa. Primer korišćenja `id` tipa za čuvanje podataka je prikazan na slici 3.6 [11].

```
id mojaTacka = [[Tacka alloc] init];
[mojaTacka ispisiKoordinate];
```
Slika 3.6 Primer dinamičkog tipiziranja

Dinamičkim uvezivanjem (engl. ***dynamic binding***) određuje se **metoda** koja se izvršava prilikom slanja poruke objektu tipa `id` bez prethodnog znanja o objektu. Odgovarajuća metoda se određuje u toku izvršavanja. Na slici 3.7 prikazan je primer metode koja kao argument prihvata objekat tipa `id` i šalje poruku tom objektu. Prilikom slanja poruke ovom objektu u toku izvršavanja aplikacije, proverava se postojanje definicije metode `ispisiKoordinate` u objektu i dolazi do greške ukoliko ova metoda nije definisana nad objektom [11].

```
- (void)prikaziInformacije:(id)objekat {
        [objekat ispisiKoordinate];
```

```
}
```

Slika 3.7 Primer dinamičkog uvezivanja

Prednost dinamičkog uvezivanja ogleda se u omogućavanju **polimorfizma**. Na primeru sa slike 3.7, metodi se mogu proslediti instance različitih geometrijskih klasa koje implementiraju ispis koordinata tačaka koje ih definišu. Takođe, olakšano je održavanje kôda. U slučaju uvođenja novih srodnih geometrijskih klasa, neophodno je pisanje novih klasa dok se kôd, koji koristi ove geometrijske klase, ne menja.

Glavni nedostatak dinamičkog uvezivanja predstavlja nemogućnost statičke provere tipova. Kompilator može samo pretpostaviti da je dinamički uvezana metoda definisana u objektu nad kojim se poziva. U slučaju pogrešno napisane metode, dolazi do greške u toku izvršavanja aplikacije kada definicija određene metode nije pronađena u objektu. Takođe, dinamičko uvezivanje metoda se odlikuje lošijim performansama od statičkog uvezivanja metoda usled pretraživanja (engl. *lookup*) adrese metode po imenu.

Na slici 3.7 prikazana je deklaracija metode koja počinje karakterom **-** koji označava metode koje mogu biti pozvane samo nad instancama klasa (poziv metode nad instancom na slici 3.1). Deklaracije metoda koje počinju karakterom **+** označavaju klasne metode kojima nije neophodna instanca i mogu biti pozvane samo nad klasama (poziv metode `alloc` na slici 3.4).

3.2 Klase

Klasa se definiše pomoću dve datoteke:

- Datoteka **zaglavlja** sa ekstenzijom **.h** sadrži **interfejs** klase. U ovoj datoteci se navode deklaracije promenljivih i metoda.
- Datoteka **implementacije** sa ekstenzijom **.m** gde se nalaze implementacije, tj. definicije metoda.

Način definisanja nove klase se može prikazati na primeru jednostavne klase **Tacka**. Na slici 3.8 prikazan je sadržaj datoteke Tacka.h koji predstavlja njen interfejs. Klasa sadrži dve instancne promenljive, koordinate x i y. Prvo se navodi ključna reč `@interface,` zatim sledi ime klase i posle dve tačke, ime natklase. U okviru vitičastih zagrada se navode deklaracije instancnih promenljivih, posle čega se navode deklaracije metoda. Na kraju se navodi ključna reč `@end.` Metode navedene u ovom primeru predstavljaju pristupne metode instancnim promenljivima.

```
@interface Tacka:NSObject
{
    float _x;
    float _y;
}
- (float)x;
- (void)setX:(float)x;
- (float)y;
- (void)setY:(float)y;
@end
```

3.8 Interfejs klase Tacka

Na slici 3.9 prikazan je sadržaj datoteke Tacka.m u kojoj su implementirane prethodno deklarisane metode. Potrebno je prvo uvesti sadržaj datoteke Tacka.h pomoću direktive `#import`. Posle toga se navodi ključna reč `@implementation` i ime klase. U ostatku datoteke se nalaze implementacije metoda, a datoteka se i u ovom slučaju završava ključnom rečju `@end` [12].

Promenljive i metode definisane u datoteci implementacije bez deklaracije u datoteci zaglavlja, nisu vidljive izvan ove datoteke.

```
#import "Tacka.h"

@implementation Tacka

- (float)x {
       return _x;
}

- (void)setX:(float)x {
       _x = x;
}

- (float)y {
       return _y;
}

- (void)setY:(float)y {
       _y = y;
```

```
}
@end
```

3.9 Implementacija klase Tacka

3.3 Svojstva

Rad sa instancnim promenljivama može se olakšati definisanjem svojstava klase (engl. *properties*). Direktivom **@property** kompilator automatski generiše odgovarajuće pristupne metode. Korišćenjem ove direktive, interfejs klase Tacka (slika 3.8) može se definisati kao na slici 3.10.

```
@interface Tacka:NSObject

@property float x;
@property float y;

@end
```

3.10 Interfejs klase Tacka sa svojstvima

Kompilator će sam napraviti odgovarajuće instancne promenljive, kao i pristupne metode. Pri tome, njihovi nazivi odgovaraju nazivima u prethodno prikazanom interfejsu i implementaciji klase Tacka. I dalje je moguće implementirati proizvoljne pristupne metode, ukoliko podrazumevane ne odgovaraju iz nekog razloga.

Prilikom upotrebe klase, za rad sa svojstvima se obično koristi tačkasta (engl. ***dot***) notacija. Pristupanje svojstvima objekta tacka klase Tacka prikazano je na slici 3.11. Ovakav vid pristupanja svojstvima ekvivalentan je kôdu datom u komentarima na istoj slici.

```
float vrednostX = tacka.x;     //[tacka getX]
tacka.x = 0.5;                 //[tacka setX:0.5]
```

3.11 Upotreba dot notacije za pristup svojstvima

Ponašanje podrazumevanih pristupnih metoda se može promeniti pomoću liste **atributa** koja se zadaje u zagradama nakon **@property** direktive. Na primer, pomoću atributa **readonly** može da se zabrani promena vrednosti svojstva, tako što se ne pravi *set* metoda i zabranjuje se dodeljivanje preko *dot* notacije (slika 3.12). Dodeljivanje vrednosti ovoj promenljivoj vrši se pristupanjem instancnoj promenljivoj ovog svojstva pomoću _ karaktera ispred naziva svojstva (_x za svojstvo na slici 3.12).

```
@property(readonly) float x;
```

3.12 *Primer navođenja atributa svojstvima*

Pomoću atributa moguće je odrediti i **atomičnost** svojstva u višenitnom okruženju. Svojstva sa atributom **atomic** su atomična, a takvo ponašanje se postiže zaključavanjem promenljive prilikom pristupa. Tek kada nit koja trenutno pristupa svojstvu završi svoj posao, druga nit može da pristupi tom istom svojstvu. Atomična svojstva ne garantuju sigurno izvršavanje kôda u višenitnim okruženjima (engl. ***thread-safety***) već samo osiguravaju da će pristupne metode svojstava vratiti *validnu* vrednost nezavisno od aktivnosti svojstva na drugim nitima. Odnosno, ukoliko nit A pokušava da pročita vrednost jednog atomičnog svojstva dok niti B i C paralelno postavljaju novu vrednost tom svojstvu, nit A će dobiti bilo koju od tri vrednosti - vrednost pre postavljanja nove ili jednu od novih vrednosti postavljenih od strane niti B i C. Vrednost koju nit A dobija je zbog atomičnosti *validna* odnosno ne može biti različita od navedene tri vrednosti. Navođenjem atributa **`nonatomic`** označavaju se svojstva koja nisu atomična, što u nekim slučajevima može dovesti do efikasnijeg izvršavanja. Ukoliko se ne navede nijedan atribut atomičnosti, svojstvo je atomično.

Ako je cilj da objekat A koji ima neko svojstvo bude vlasnik objekta B koji se dodeli tom svojstvu, navodi se atribut **`strong`**. Tako se garantuje da će objekat B postojati sve dok odgovarajuće svojstvo čuva njegovu referencu. Ovakvo ponašanje je podrazumevano za sva svojstva koja su objektnog tipa. Na taj način je predstavljena jaka veza između glavnog objekta A i objekta B čiju referencu čuva i čiji je vlasnik. Problem sa ovakvom vezom je mogućnost stvaranja uzajamnih jakih veza, tj. **ciklusa** kada objekat B ima takođe jaku referencu na objekat A. U tom slučaju A je vlasnik objekta B, a u isto vreme B je vlasnik objekta A. Tako oba objekta uvek imaju bar jednog vlasnika zbog čega ne mogu biti oslobođeni iz memorije i nagomilavanjem može doći do curenja memorije. Ovo se može sprečiti navođenjem atributa **`weak`** za jedno od svojstava u pomenutom ciklusu, tj. upotrebom slabe veze umesto jedne od jakih veza u ciklusu. U tom slučaju objekat nije vlasnik objekta čiju referencu čuva.

Umesto atributa **`strong`** moguće je koristiti atribut **`copy`**. U tom slučaju, prvo se pravi kopija objekta koji se dodeljuje svojstvu, pa se vlasništvo uspostavlja nad tom kopijom. Na taj način promene na originalnom objektu ne utiču na objekat čija se referenca čuva kao svojstvo [13].

3.4 Protokoli

Pojam protokola odgovara pojmu **interfejsa** u drugim objektno-orijentisanim jezicima, kao što je *Java*. Protokol predstavlja **listu deklaracija** metoda. Neke od

metoda iz te liste su obavezne, a neke opcione. Klasa je u skladu sa datim protokolom ako implementira sve njegove obavezne metode. Na slici 3.13 dat je primer definicije protokola, dok je na slici 3.14 prikazano zaglavlje klase koja je u skladu sa datim protokolom.

```
@protocol Ispisivanje
@required
- (void)ispisi;
@optional
- (void)ispisiULogDatoteku;
@end
```

3.13 *Primer definicije protokola*

Ime protokola se navodi posle direktive **`@protocol`**, obavezne metode ispod **`@required`**, a opicone ispod **`@optional`**. Kod definicije klase koja je u skladu sa datim protokolom, ime protokola se navodi u uglastim zagradama `<...>` posle naziva natklase (slika 3.14).

```
@interface Tacka: NSObject <Ispisivanje>
@property float x;
@property float y;
@end

@implementation Tacka
- (void)ispisiKoordinate {
    NSLog(@"Tacka - X koordinata:%f, Y koordinata:%f", self.x, self.y);
}

// metoda iz protokola
- (void)ispisi {
    [self ispisiKoordinate];
}
@end
```

3.14 *Primer klase koja je u skladu sa protokolom*

Funkcija `NSLog(NSString *format, ...)` često se koristi za ispis poruka u sistemsku konzolu čime se olakšava debagovanje. Prvi argument ove funkcije je format niska sa specifikacijom parametara dok se naredni argumenti ove funkcije ugrađuju u tu nisku.

U slučaju da je klasa u skladu sa više protokola, u okviru uglastih zagrada navodi se lista protokola razdvojenih zarezom, što je prikazano na slici 3.15.

```
@interface Tacka : NSObject <Ispisivanje, Crtanje>
//...
@end
```
3.15 Primer klase koja je u skladu sa više protokola

Iako je u skladu sa protokolom, moguće je da klasa ne implementira neku od opcionih metoda. Prilikom izvršavanja, korisnik objekta te klase može da proveri da li postoji implementacija određene metode. Za proveru se koristi **`respondsToSelector:`** metoda, koji se može pozvati nad bilo kojim objektom. Ova metoda kao argument prihvata odgovarajući selektor koji se može dobiti od imena metode pomoću direktive **`@selector`**. Na slici 3.16 dat je primer upotrebe opcione metode sa prethodno opisanom proverom [14].

```
if ([objekat respondsToSelector:@selector(ispisiULogDatoteku)]) {
  [objekat ispisiULogDatoteku];
}
```
3.16 Provera postojanja opcione metode

3.5 Kategorije i ekstenzije

Pomoću **kategorija** moguće je dodati nove metode postojećim klasama bez potrebe za pravljenjem potklasa. Metode je moguće dodati i kada nam nije dostupan izvorni kôd originalne klase. Na taj način je moguće dodati nove mogućnosti nekoj sistemskoj klasi bez pravljenja nove klase i nasleđivanja. Još jedna primena kategorija jeste i grupisanje metoda neke velike klase u više kategorija i raspoređivanje u više datoteka.

Na slici 3.17 dat je primer deklaracije jedne kategorije. Deklaracija je slična deklaraciji obične klase, a razlika je u tome što se posle naziva klase navodi ime kategorije u zagradama, dok se ime natklase izostavlja.

```
#import "Tacka.h"
@interface Tacka (Pomeranje)
//Deklaracije metoda kategorije
- (void)pomeriXkoordinatu:(float)broj;
- (void)pomeriYkoordinatu:(float)broj;
@end
```
3.17 Deklaracija kategorije

Implementacija kategorije je slična implementaciji klase, i ovde se naziv kategorije navodi posle imena klase u zagradama (slika 3.18). Nazivi datoteka koji čuvaju

deklaraciju i implementaciju kategorije su po konvenciji u obliku *<ime_klase>+<ime_kategorije>* (datoteka `Tacka+Pomeranje.h` na slici 3.17 i datoteka `Tacka+Pomeranje.m` na slici 3.18).

```
#import "Tacka+Pomeranje.h"
@implementation Tacka (Pomeranje)
//Implementacije metoda kategorije
- (void)pomeriXkoordinatu:(float)broj {
    self.x += broj;
}
- (void)pomeriYkoordinatu:(float)broj {
    self.y += broj;
}
@end
```

3.18 Implementacija kategorije

Za razliku od kategorija, **ekstenzije** se mogu dodati samo klasama čiji je izvorni kôd dostupan u vreme kompajliranja, jer se metode ekstenzije implementiraju u okviru implementacije originalne klase. Takođe, ekstenzije mogu da dodaju klasi i nova svojstva i instancne promenljive.

Jedna moguća primena ekstenzija jeste deklarisanje privatnih svojstava klase, tj. svojstava koje samo proširena klasa koristi interno. U tom slučaju ekstenzija se navodi pre same implementacije klase u kojoj se koristi to svojstvo, što je prikazano na slici 3.19. Ekstenzija ima sličan oblik kao kategorija, međutim njeno ime se ne navodi u zagradama pa se ekstenzije nazivaju i anonimne kategorije [15].

```
#import "Tacka.h"
@interface Tacka ()
@property UIColor *boja;
@end

@implementation Tacka
//...
@end
```

3.19 Primer ekstenzije klase

3.6 Blokovi

Blok odgovara pojmu anonimne funkcije ili lambda izraza u drugim jezicima. Blok može da ima argumente i rezultat. Takođe, blok se može proslediti nekoj metodi koja prihvata blok kao argument ili se može sačuvati u odgovarajućoj promenljivoj, slično kao kod pokazivača na funkcije u jeziku C. Na slici 3.20 prikazan je primer bloka koji računa zbir dva cela broja. Struktura bloka je slična funkciji u jeziku C, međutim umesto imena navodi se znak ^.

```
^(float prviBroj, float drugiBroj) {
    return prvi + drugi;
}
```

3.20 *Primer bloka koji sabira dva broja*

Implementacija metode koja prihvata blok sa slike 3.20, prikazana je na slici 3.21. Argument metode je blok koji prihvata dva argumenta tipa **`float`** i vraća vrednost tipa **`float`**. U daljoj implementaciji metode, argument `operacija` koristi se poput imena funkcije.

```
- (Tacka *)tackaPosleObrade:(float (^)(float, float))operacija {
    float xVrednost = operacija(self.x, 3.0);
    float yVrednost = operacija(self.y, 4.0);
    Tacka *novaTacka = [Tacka alloc] init];
    novaTacka.x = xVrednost;
    novaTacka.y = yVrednost;
    return novaTacka;
}
```

3.21 *Primer metode koja ima blok kao argument*

U skladu sa predstavljenim blokom i definicijom metode `tackaPosleObrade:` koja koristi blok, na slici 3.22 prikazan je način prosleđivanja bloka u vidu argumenta metode [16].

```
Tacka *novaTacka =
[tacka tackaPosleObrade:^(float prviBroj, float drugiBroj) {
    return prviBroj + drugiBroj;
}];
```

3.22 *Primer prosleđivanja bloka metodi*

Na ovaj način moguće je proslediti različite operacije za obradu tačke umesto sabiranja.

4 Radni okvir iOS

U ovom poglavlju biće predstavljeni koncepti programiranja aplikacija za platformu iOS, osnove funkcionisanja aplikacija, interakciju aplikacija sa sistemom kao i tehnike dizajniranja univerzalnih korisničkih interfeja.

4.1 Radni okvir *Foundation*

Foundation, kao jedan od glavnih radnih okvira iOS-a, pruža Objective-C jeziku skup esencijalnih klasa za čuvanje podataka, upravljanje niskama, datumima, izuzecima, datotekama itd. Klase ovog okvira imaju unikatan **prefiks `NS`** koji vuče poreklo od naziva operativnog sistema **N**eXT**S**TEP za koji je napravljen programski jezik Objective-C. Postavljanje prefiksa je neophodno kako ne bi došlo do kolizije naziva klasa usled nepostojanja imenskih prostora kao u jeziku C++.

Jezik Objective-C nativno podržava osnovne primitivne tipove podataka iz jezika C (`char`, `int`, `long`, `float`, `double`, `void`) kao i pokazivače, strukture, unije itd. Svi ovi tipovi vrednosti mogu biti sadržani u **`NSValue`** objektu koji je definisan kao jednostavno skladište ovakvih podataka. Za čuvanje svih tipova brojeva (celih brojeva i brojeva u pokretnom zarezu), definisana je klasa **`NSNumber`** (potklasa klase `NSValue`) nad kojom je moguće samo čitanje vrednosti. Klasa **`NSDecimalNumber`** je potklasa klase `NSNumber` i omogućava čuvanje brojeva u pokretnom zarezu sa većom preciznošću. Okvir *Foundation* pruža omotač **`NSInteger`** koji određuje tip celog broja u zavisnosti od arhitekture procesora (`int` na 32-bitnim a `long` na 64-bitnim procesorima). Za čuvanje nizova bajtova u vidu objekata koristi se klasa **`NSData`** koja pruža interfejse za jednostavno upisivanje svojih podataka u datoteke. Tip **`BOOL`** se koristi za čuvanje istinitosnih (engl. *boolean*) vrednosti [17].

Mnogi objekti u okviru *Foundation* imaju varijantu objekta promenljivog (engl. ***mutable***) sadržaja i varijantu objekta nepromenljivog (engl. ***immutable***) sadržaja. Enkapsulirani sadržaj nepromenljivih objekata ne može biti promenjen nakon pravljenja objekata za razliku od promenljivih objekata. Pri radu sa nepromenljivim objektima, jedini način da se „promeni" sadržaj jednog objekta je da se instancira novi objekat i promeni referenca starog objekta da pokazuje na novi objekat.

Klasom **`NSString`** predstavlja se objekat niske koji pruža veliki broj metoda za interakciju sa niskama. Ovo je jedna od klasa čiji sadržaj je nepromenljiv. Klasa **`NSMutableString`** pruža metode za upravljanje promenljivim niskama poput metoda `insertString:atIndex:`, `appendString:` i `deleteCharactersInRange:`.

U jeziku Objective-C, objekti literali se definišu koristeći simbol `@`. Na slici 4.1 prikazan je primer postavljanja prefiksa @ ispred određenih vrednosti čime se inicijalizuje objekat tom vrednošću.

```
NSNumber *brojTacaka = @4;
NSNumber *xKoordinata = @3.14;

NSString *nazivTacke = @"Druga tacka";
```

Slika 4.1 Primeri pravljenja literal objekata

Za čuvanje kolekcija objekata koriste se klase **`NSArray`**, **`NSDictionary`** i **`NSSet`**. Ove klase su nepromenljive i njima odgovaraju varijante promenljivih klasa **`NSMutableArray`**, **`NSMutableDictionary`** i **`NSMutableSet`**. Jedna kolekcija nije vezana za jedan tip objekta i može čuvati objekte različitih tipova.

Klasa `NSArray` pruža metode za upravljanje nizom objekata i često se inicijalizuju literalom niza kao sa slike 4.2. Objekti niza razdvojeni su zarezima u sintaksnoj konstrukciji literal niza `@[objekat1, objekat2, ...]`.

```
NSArray *niz = @[tacka, @"Prva tacka", @42];
```

Slika 4.2 Primer incijalizacije niza

Klasa `NSDictionary` odgovara strukturi mape tj. rečnika iz drugih programskih jezika i predstavlja skup ključ-vrednost parova sa unikatnim ključevima. Ključevi se definišu kao instance `NSString` klase. Na slici 4.3 prikazan je primer inicijalizacije rečnika literalom rečnika umesto pravljenja dva posebna niza ključ objekata i vrednost objekata. Unosi u rečniku, odnosno parovi ključ-vrednost, razdvojeni su zarezima u sintaksnoj konstrukciji literal rečnika `@{par1, par2, ...}` dok su ključ i vrednost u paru razdvojeni karakterom `:`.

```
NSDictionary *recnik = @{
    @"susednaTacka" : tacka,
      @"nazivTacke" : @"Prva tacka",
      @"udaljenost" : @42
};
```

Slika 4.3 Primer inicijalizacije rečnika

Klasa `NSSet` definiše skup različitih objekata. Skupovi se koriste kao alternativa nizovima kada nije važan redosled objekata u kolekciji. Provera pripadnosti elementa je brža kod skupova nego kod nizova.

Promenljive verzije kolekcija objekata imaju mogućnost dodavanja i brisanja elemenata kolekcija u svakom trenutku, dok se elementi nepromenljivih kolekcija utvrđuju prilikom inicijalizacije kolekcija.

4.2 Redovi i tehnologija Grand Central Dispatch

Grand Central Dispatch (**GCD**) je tehnologija dostupna od iOS verzije 4 čija namena je optimizacija podrške aplikacije za sisteme sa multijezgarnim procesorima pomoću paralelizacije zadataka. Ovakvi zadaci se u vidu blokova prosleđuju specifičnim GCD funkcijama. GCD je razvijen kao alternativa za višenitno programiranje čime je fokus programera prebačen sa brige o organizaciji niti na zadatke koje je neophodno izvršiti.

Najčešće korišćena GCD funkcija je `dispatch_async`. Ova funkcija se koristi za pravljenje asinhronih zadataka da bi se izbeglo predugo blokiranje glavne niti prilikom izvršavanja radnji koje zahtevaju duži vremenski period poput preuzimanja podataka sa mreže i obrada velike količine podataka.

Funkcija `dispatch_async` kao parametre prima red (engl. *queue*) i blok bez argumenata i rezultata. Prosleđeni blok predstavlja asinhroni zadatak koji se postavlja na red. Postoji dva tipa redova: serijski i konkurentni. Serijski redovi izvršavaju samo po jedan zadatak u jednom trenutku, dok konkurentni redovi mogu izvršavati više zadataka paralelno. Svi redovi izvršavaju svoje blokove u redosledu *FIFO* (*First In, First Out*).

Moguće je pravljenje redova kao i korišćenje postojećih sistemskih redova serijskog ili konkurentnog tipa.

Postoji jedan serijski sistemski red koji odgovara glavnoj niti (eng. *main thread*) i dobija se funkcijom `dispatch_get_main_queue`. Ovaj red na glavnoj niti se najčešće koristi za ažuriranja grafičkog interfejsa aplikacije i objavljivanja obaveštenja o završenoj obradi.

Postoji četiri konkurentna sistemska reda koji se dobijaju funkcijom `dispatch_get_global_queue`. Ovoj funkciji se prosleđuje identifikator prioriteta i najčešće se koristi za izvršavanje zadataka u pozadini.

Na slici 4.4 prikazan je primer korišćenja ugnježdenih asinhronih GCD funkcija gde se u pozadini izvršava preuzimanje podataka sa mreže na konkurentnom sistemskom redu i nakon toga izvršava ažuriranje grafičkog interfejsa na glavnoj niti serijskog sistemskog reda.

```
dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
    // Preuzimanje podataka sa mreze u pozadini
    // ...
    dispatch_async(dispatch_get_main_queue(), ^{
        // Azuriranje grafickog interfejsa aplikacije na glavnoj niti
```

```
        // nakon preuzimanja podataka
        // ...
    });
});
```

Slika 4.4 Primer korišćenja GCD funkcija i redova

4.3 Životni ciklus aplikacije

Izvršavanje aplikacije započinje od funkcije **`main`** kao u programima pisanim u programskom jeziku C. Razvojno okruženje generiše funkciju `main` u kojoj se poziva funkcija **`UIApplicationMain`** koja inicijalizuje aplikaciju generisanjem fundamentalnih objekata aplkacije, učitavanjem grafičkog korisničkog interfejsa aplikacije i pokretanjem glavne petlje aplikacije. Na slici 4.5 vidimo standardnu *main* funkciju koju generiše razvojno okruženje.

```
#import <UIKit/UIKit.h>
#import "AppDelegate.h"

int main(int argc, char * argv[]) {
    @autoreleasepool {
        return UIApplicationMain(argc, argv, nil,
            NSStringFromClass([AppDelegate class]));
    }
}
```

Slika 4.5 Generisana main funkcija

Jedan od fundamentalnih objekata koje funkcija `UIApplicationMain` generiše je **`UIApplication`**. Ovaj objekat predstavlja centralnu tačku kontrole i koordinacije aplikacija što ostvaruje prosleđivanjem poruka od kontrolnih objekata do odgovarajućih ciljanih objekata. On upravlja glavnom petljom događaja na koje sistem reaguje, poput dodira, promene lokacije, zahteva sa ponovno iscrtavanje pogleda itd. Objekat `UIApplication` radi zajedno sa delegatom aplikacije odnosno objektom **`UIApplicationDelegate`** kojeg pravi programer kao obaveznu komponentu svake aplikacije. Objekat `UIApplicationDelegate` upravlja prelaskom između stanja aplikacije, događajima visokog nivoa i inicijalizacijom aplikacije. Prilikom incijalizacije aplikacije, u ovom objektu se često inicijalizuju globalno raspoloživi podaci pre učitavanja ostalih elemenata. Delegat aplikacije čuva podatak o objektu **`UIWindow`** koji koordinira prezentacijom pogleda na ekranu aplikacije.

Aplikacija se u svakom trenutku nalazi u jednom od 5 mogućih stanja.

U tabeli 4.1 prikazana su stanja i opisi stanja u kojima može biti aplikacija [18]. Aplikacija koja se nalazi u prvom planu je prikazana na ekranu uređaja. Aplikacija koja se nalazi u pozadini nije prikazana na ekranu uređaja, ali se i dalje nalazi u memoriji i moguće je ponovo prikazati aplikaciju bez njenog ponovnog učitavanja.

Tabela 4.1 Stanja aplikacije

Stanje	**Opis**
Nije pokrenuta	Aplikacija nije nikad pokrenuta ili je ugašena od strane sistema
Neaktivna	Aplikacija se nalazi u prvom planu ali trenutno ne prima događaje. Ovo je privremeno stanje za aplikaciju.
Aktivna	Aplikacija se nalazi u prvom planu i prima događaje. Ovo je normalno stanje za aplikacije koje se nalaze u prvom planu.
U pozadini	Aplikacija se nalazi u pozadini i izvršava kôd. Ovo je prelazno stanje u kom se aplikacija nalazi u kratkom vremenskom periodu pre suspendovanja.
Suspendovana	Aplikacija se nalazi u pozadini ali ne izvršava kôd. Sistem automatski prebacuje aplikaciju u ovo stanje i čuva aplikaciju u memoriji dok ne dođe do manjka memorije kada sistem oslobađa suspendovane aplikacije iz memorije da napravi više prostora za aplikacije u prvom planu.

4.4 Upravljanje memorijom

Upravljanje memorijom u iOS aplikacijama se ostvaruje praćenjem vlasništva objekata. **Sistem za brojanje referenci** prati koliko svaki objekat ima vlasnika. Brojač referenci objekta se uvećava kada se utvrdi vlasništvo nad objektom, a umanjuje kada se oslobodi vlasništvo nad objektom. Tek kada brojač dostigne vrednost nula, tada operativni sistem sme da uništi i oslobodi objekat iz memorije.

Pre iOS verzije 4, programeri su bili u obavezi da sami vode računa o broju referenci objekata koristeći sistem **MMR** (*Manual Retain Release*). Ovaj sistem zahteva ručno slanje poruka `retain` i `release` za utvrđivanje i oslobađanje vlasništva nad objektima. Ručni način vođenja računa o memoriji je dovodio do problema poput curenja memorije i visećih pokazivača usled ljudske greške u programiranju. Curenje memorije nastaje usled nagomilavanja neoslobođenih objekata, dok viseći pokazivači nastaju u slučaju oslobađanja objekata preveliki broj puta.

U iOS verziji 4 predstavljen je novi sistem **ARC** (Automatic Reference Counting) za automatsko brojanje referenci objekata. Umesto ručnog slanja poruka za utvrđivanje i oslobađanje vlasništva, ARC sistem automatski dodaje ekvivalente `retain` i `release` poruka u toku kompilacije. Na ovaj način, brojač referenci za sve objekte se

automatski uvećava i umanjuje u toku izvršavanja aplikacije minimizujući verovatnoću za pojavu memorijskih problema. Za razliku od *Garbage Collector* komponente u drugim sistemima, sistem ARC ne procesuira u pozadini što ga čini efikasnijim na sporijim uređajima poput mobilnih telefona. Takođe, sistem ARC nije sposoban da reši **cikluse referenci** o kojima je bilo reči u poglavlju 3.3. Dužnost programera je da prekine ove cikluse korišćenjem slabih (engl. ***weak***) referenci [19].

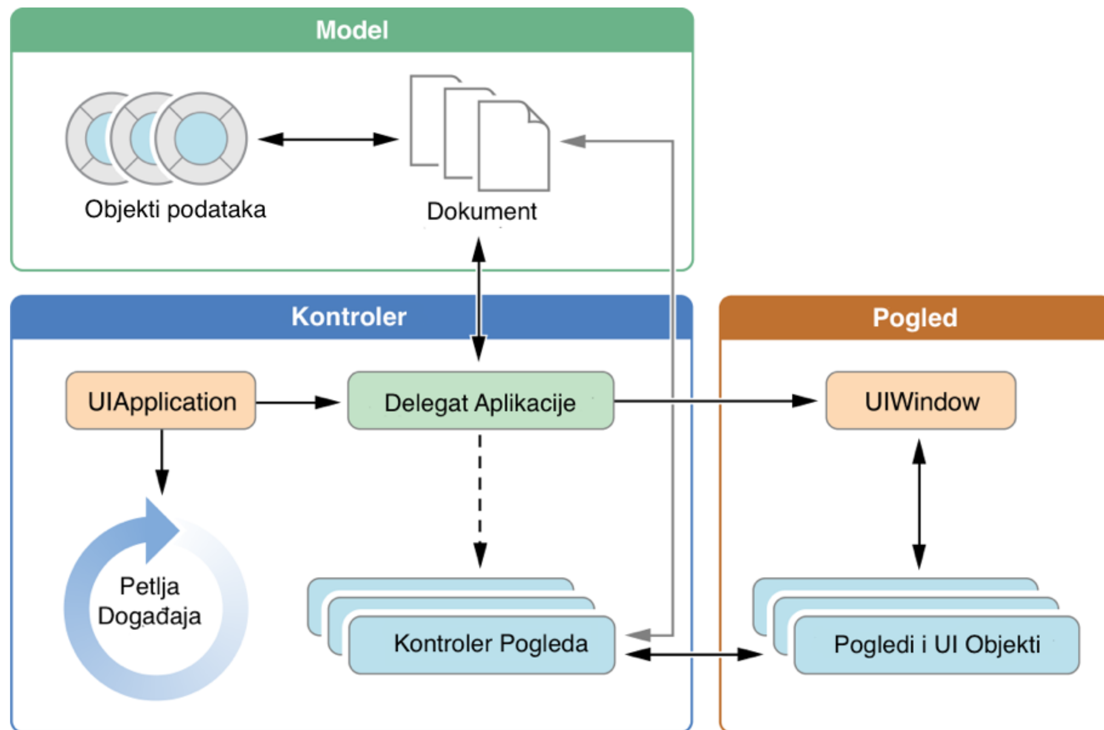
Na prethodnoj slici 4.5 prikazana je upotreba bloka **`@autoreleasepool`** u standardnoj `main` funkciji. Sistem ARC koristi ovaj blok da enkapsulira alokaciju privremenih objekata koje dealocira kada aplikacija stigne sa izvršavanjem do kraja bloka.

4.5 Obrasci projektovanja

Radni okvir iOS oslanja se u velikoj meri na obrasce projektovanja u svojoj implementaciji. Za uspešno pravljenje aplikacije neophodno je dobro razumevanje često korišćenih obrazaca projektovanja. U narednim potpoglavljima biće prikazani najčešći obrasci projektovanja u iOS aplikacijama i načini njihove realizacije. Neki obrasci biće opisani do nivoa ručne implementacije dok će za ostale obrasce biti predstavljene funkcionalnosti iOS-a koje su zasnovane na tim obrascima.

4.5.1 Obrazac Model-pogled-kontroler

Struktura iOS aplikacija organizovana je preko Model-pogled-kontroler (engl. *Model-View-Controller*) arhitekture. Ovo je najčešće korišćeni obrazac prilikom razvoja aplikacija za platformu iOS. Ova troslojna arhitektura razdvaja podatke aplikacije i biznis logiku od vizualne reprezentacije podataka. Na slici 4.6 vidimo odnos sistemskih i napravljenih objekata i njihovu pripadnost slojevima troslojne arhitekture. [18]



Slika 4.6 Ključni objekti u iOS aplikaciji

Sistemski objekti su prikazani u poglavlju 4.1. U narednoj listi biće prikazani tipovi napravljenih objekata od strane programera u okviru slojeva troslojne arhitekture:

- **Sloj Model** je sačinjen iz objekata **podataka** i dokumenata. Ovi objekti čuvaju podatke aplikacije i definišu njihovu manipulaciju.
- **Sloj Pogled** čine **pogledi** i drugi *UI* objekti (objekti korisničkog interfejsa) poput kontrolnih objekata i objekata slojeva. Pogledi su objekti zaduženi za vizuelnu reprezentaciju modela koji iscrtavaju sadržaj u predodređenoj pravougaonoj površini i reaguje na događaje iz te površine. Objekti kontrola implementiraju funkcionalnosti kontrole poput dugmića i tekstualnih polja. Pogledi koriste objekte slojeva za reprezentaciju vizuelnog sadržaja.
- **Sloj Kontroler** je predstavljen objektima **kontrolera pogleda**. Kontroler pogleda je posrednik koji koordinira svim akcijama. On pristupa podacima iz modela koji se prikazuju na pogledima, prati događaje i barata podacima po potrebi. Kontroler pogleda direktno upravlja prezentacijom sadržaja aplikacije i prezentuje jedan pogled zajedno sa njegovom kolekcijom potpogleda.

Model obaveštava Kontroler o izmenama u podacima, a Kontroler vrši izmenu podataka u Pogledu. Pogled obaveštava Kontroler o radnjama korisnika, a Kontroler po potrebi vrši izmene u Modelu. Ovime se postiže veća razdvojenost kôda i povećava ponovna upotrebljivost komponenti.

4.5.2 Obrazac Unikat

Obrazac Unikat (engl. *Singleton*) osigurava da postoji samo jedna instanca određene klase koju više komponenata koriste. Ovaj pristup je čest u radnom okviru iOS i koristi se kada je neophodno da postoji samo jedna instanca određene klase poput deljenog objekta `UIApplication` klase (dobija se porukom `[UIApplication sharedApplication]`), `UIScreen` klase (dobija se porukom `[UIScreen mainScreen]`) itd.

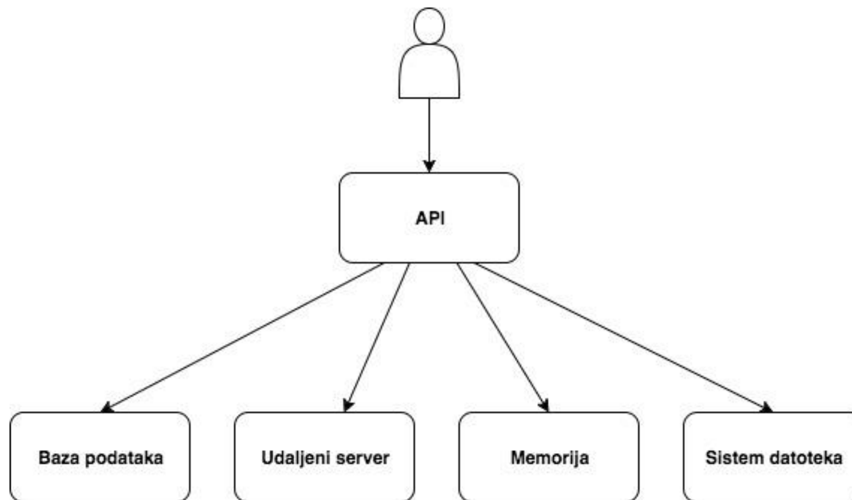
Na slici 4.7 prikazan je primer metode koja implementira obrazac *Singleton* i vraća deljenu instancu objekta `Tacka`. Promenljive sa oznakom **`static`** predstavljaju statičke promenljive koje se prave samo jednom u toku izvršavanja aplikacije i zadržavaju svoju vrednost kroz višestruke pozive metode. Statička promenljiva `_deljenaInstanca` čuva instancu klase `Tacka`. Statička promenljiva `oncePredicate` predstavlja GCD token koji se prosleđuje GCD bloku `dispatch_once` čime se osigurava samo jedno izvršavanje bloka. Prilikom svakog sledećeg poziva funkcije biće vraćena referenca na objekat koji je napravljen u prvom pozivu funkcije u okviru bloka `dispatch_once` kada se jedini put izvršio [20].

```
+ (Tacka *)deljenaInstanca {
    static Tacka *_deljenaInstanca = nil;

    static dispatch_once_t oncePredicate;

    dispatch_once(&oncePredicate, ^{
        _deljenaInstanca = [[MojObjekat alloc] init];
    });
    return _deljenaInstanca;
}
```

Slika 4.7 Primer implementacije Singleton obrasca

4.5.3 Obrazac Fasada

Obrazac Fasada (engl. *Facade*) se ostvaruje tako što se korisniku izlaže jedinstveni *API* koji predstavlja interfejs ka kompleksnom podsistemu sačinjenom od skupa klasa i njihovih *API*-a. Na slici 4.8 prikazan je primer ostvarivanja koncepta obrasca Fasada.



Slika 4.8 Prikaz koncepta obrasca Fasada

Ovaj obrazac je koristan kada komponenta radi sa velikim brojem klasa i podsistema, a želimo da korisnika *API*-a potpuno odvojimo od te kompleksnosti pružajući mu jedinstvenu pristupnu tačku sistema. Klasa Fasade zadržava isti *API* kada se menjaju klase podsistema Fasade. U tom slučaju nije neophodno vršiti nikakve izmene kôda u kojima se koristi *API* čime se smanjuje spregnutost celokupnog sistema [20].

4.5.4 Obrazac Dekorater

Obrazac Dekorater (engl. *Decorator*) predstavlja alternativu pravljenja potklasa tako što se dodaju ponašanja i odgovornosti objektima bez modifikacije njihovog kôda. To se ostvaruje korišćenjem dve implementacije ovog obrasca: **Kategorije** i **Delegacija** [20].

O kategorijama je bilo reči u poglavlju 3.5. Metode kategorija mogu biti izvršene kao normalne metode proširenih klasa. Kategorije pomažu u razdvajanju kôda u sekcije i očuvanju celokupne organizacije kôda.

Delegacija je mehanizam u kom jedan objekat (delegat) pruža informacije i izvršava radnje za drugi objekat. Ovo je veoma bitan obrazac koji se koristi kod većine klasa iz radnog okvira *UIKit* poput: `UITableView`, `UITextView`, `UITextField`, `UIWebView`, `UICollectionView`, `UIScrollView` itd.

Delegacija se u jeziku Objective-C ostvaruje putem protokola o kojima je bilo reči u poglavlju 3.4. Objekat delegat se prilagođava protokolu implementirajući sve obavezne metode protokola. Mehanizam delegiranja se često koristi za omogućavanje ponovne upotrebljivosti pogleda. Objekti pogleda poput tabela se

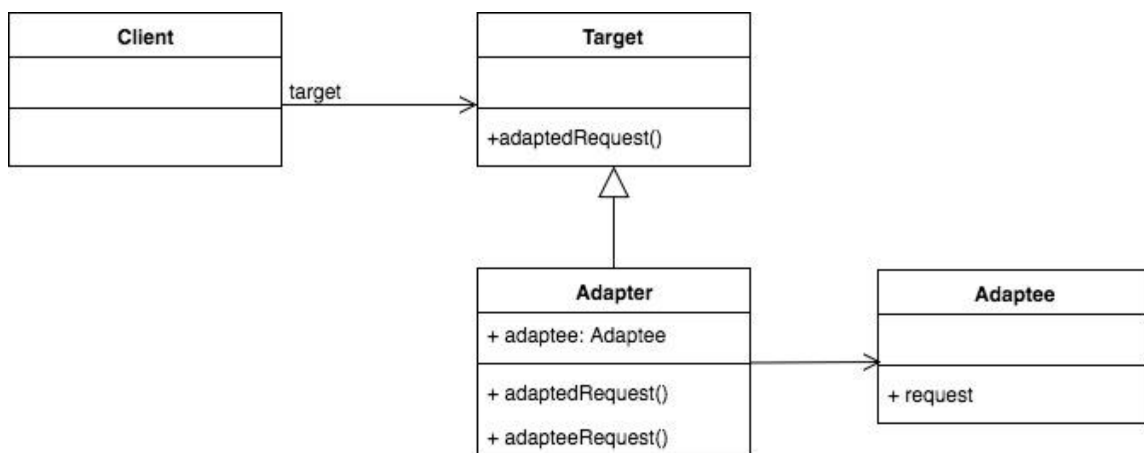
oslanjaju na svoje delegate koji su zaduženi za pružanje informacija o samim sadržajima tih pogleda poput broja ćelija tabele, izgleda ćelija, broja sekcija tabele itd.

4.5.5 Obrazac Adapter

Obrazac Adapter koristi strukturu protokola kako bi omogućio klasama sa nekompatibilnim interfejsima da zajedno rade. Ovaj drugi vid upotrebe protokola se, za razliku od delegacije, koristi za konverziju jednog interfejsa klase u drugi interfejs [20].

U okviru protokola grupisane su metode koje moraju biti implementirane u svim klasama koje druga klasa koristi u okviru jednog posla. Ukoliko klasa ne implementira sve metode iz protokola, neophodno je za nekompatibilnu klasu napraviti objekat Adapter koji implementira metode iz datog protokola čuvajući instancu nekompatiblne klase.

Na narednim slikama biće prikazan primer implementacije obrasca Adapter dok se na slici 4.9 vidi dijagram primera ovog obrasca.



Slika 4.9 Dijagram primera Adapter obrasca

Na slici 4.10 prikazan je primer protokola sa obaveznim metodama koje nekompatibilna klasa mora implementirati.

```
// Target
@protocol Crtanje <NSObject>
// adaptedRequest
- (void)nacrtaj;
@end
```

Slika 4.10 Obavezne metode u okviru protokola

Na slici 4.11 prikazan je primer klase koja mora biti adaptirana.

```
// Adaptee
@interface Tacka : NSObject
// request
- (void)ispisiKoordinate;
@end

@implementation Tacka
- (void)ispisiKoordinate {
    NSLog(@"Tacka - X koordinata:%f, Y koordinata:%f", self.x, self.y);
}
@end
```

Slika 4.11 Klasa koja mora biti adaptirana

Na slici 4.12 prikazan je primer klase koja predstavlja klasu Adapter.

```
// Adapter
@interface GrafickaTacka : NSObject <Crtanje>
// adaptee
@property Tacka *tacka;

// adaptedRequest
- (void)nacrtaj;

// adapteeRequest
- (void)ispisiKoordinate;
@end

@implementation GrafickaTacka
- (void)nacrtaj {
    //...
}
- (void)ispisiKoordinate {
    [_tacka ispisiKoordinate];
}
@end
```

Slika 4.12 Adapter klasa

Na ovaj način je klasa Adaptee indirektno prilagođena protokolu bez izmene u njenom kôdu. Za svaku nekompatibilnu klasu potrebno je napraviti objekat Adapter koji implementira metode protokola Target. Time su usklađeni interfejsi prema

zahtevima klase kojoj su za određeni posao neophodne klase koje implementiraju metode iz protokola.

4.5.6 Obrazac Posmatrač

Obrazac Posmatrač (engl. *Observer*) podstiče dizajn sa niskim stepenom spregnutosti tako što jedan objekat obaveštava drugog objekta o promenama stanja bez potrebe da objekti znaju jedan o drugome. Ovaj obrazac se najčešće koristi da, o promeni svojstva nekog objekta, obavesti sve objekte koji su prijavili interesovanje za određeno svojstvo tog objekta. Ovo se ostvaruje na dva načina u radnom okviru iOS: **Obaveštenja** (engl. *Notifications*) i **Ključ-Vrednost Posmatranje** (engl. *Key-Value Observing)* [20].

4.5.6.1 Obaveštenja

Sistemski centar za obaveštenja funkcioniše po principu prijavljivanja i obaveštavanja (engl. *subscribe-and-publish)*. Objekat izdavač (engl. *publisher)* šalje poruke objektima koji su prijavljeni da osluškuju (engl. *listeners*). U operativnom sistemu iOS postoji veliki broj sistemskih događaja koji šalju obaveštenje aplikaciji poput obaveštenja o promeni stanja aplikacije (`UIApplicationDidEnterBackgroundNotification`), obaveštenja o prikazivanju tastature (`UIKeyboardWillShowNotification`) itd.

Na slici 4.13 prikazan je primer dodavanja posmatrača podrazumevanom centru za obaveštenja. Podrazumevani centar za obaveštenja se dobija kao unikat objekat.

```
[[NSNotificationCenter defaultCenter]
       addObserver:self
         selector:@selector(zavrsenaObradaTacke:)
             name:@"ZavrsenaObradaTackeObavestenje"
           object:nil];
```

Slika 4.13 Dodavanje objekta posmatrača

U ovom primeru, objekat sam sebe dodaje kao posmatrača i kada centar za obaveštenja primi obaveštenje pod nazivom „ZavrsenaObradaTackeObavestenje”, tada se poziva metoda `zavrsenaObradaTacke` koja kao argument prima `NSNotification` objekat.

Na slici 4.14 prikazan je primer slanja obaveštenja u objektu izdavaču.

```
[[NSNotificationCenter defaultCenter]
       postNotificationName:@"ZavrsenaObradaTackeObavestenje"
                     object:self
                   userInfo:@{@"stanje":status}];
```
Slika 4.14 Slanje obaveštenja

Objekat izdavač šalje svojim posmatračima obaveštenje pod nazivom „ZavrsenaObradaTackeObavestenje”, a prosleđuje sebe kao pošiljaoca i rečnik informacija koji u ovom primeru sadrži stanje obrade.

Takođe, neophodno je odjaviti posmatrača iz centra za obaveštenja prilikom dealokacije objekta. Na slici 4.15 prikazan je primer odjave klase od obaveštenja.

```
- (void)dealloc {
    [[NSNotificationCenter defaultCenter] removeObserver:self];
}
```
Slika 4.15 Odjava klase od obaveštenja

Ukoliko se ne izvrši propisna odjava klase od obaveštenja za koje se prijavila, može doći do neželjenog prekida rada aplikacije usled slanja obaveštenja dealociranoj instanci [20].

4.5.6.2 Posmatranje ključa i vrednosti

Kod mehanizma Posmatranja ključa i vrednosti (PKV), nije neophodno ručno slanje obaveštenja iz objekta izdavača već objekti sa posmatranim svojstvom automatski šalju poruku o promenjenoj vrednosti svojstva svim svojim posmatračima. Na slici 4.16 prikazan je primer prijavljivanja zainteresovanosti za svojstvo `boja` objekta `tacka`.

```
[tacka addObserver:self forKeyPath:@"boja" options:0 context:nil];
```
Slika 4.16 Primer prijavljivanja za PKV

U primeru na slici 4.16 posmatrač je `self` odnosno trenutna klasa koja mora imati implementiranu metodu sa slike 4.17.

```
- (void)observeValueForKeyPath:(NSString *)keyPath ofObject:(id)object
       change:(NSDictionary *)change context:(void *)context {

    if ([keyPath isEqualToString:@"boja"]) {
        NSLog(@"Promenjena je vrednost boje tacke!");
    }
```

```
}
```

Slika 4.17 Metoda koja se poziva prilikom promene vrednosti posmatranog svojstva

Metoda sa slike 4.17 se poziva kada god dođe do promene vrednosti posmatranih svojstava. Ovu metodu je neophodno implementirati u svim klasama koje su prijavljene kao posmatrači nekih svojstava [20].

Kao i kod sistemskog centra za obaveštenja, neophodno je odjaviti se od posmatranja svojstava prilikom dealokacije klase kako ne bi došlo do neželjenog prekida rada aplikacije usled slanja poruka dealociranim posmatračima. Primer odjavljivanja od posmatranja svojstva objekta prikazan je na slici 4.18.

```
- (void)dealloc {
    [mojObjekat removeObserver:self forKeyPath:@"boja"];
}
```

Slika 4.18 Primer odjavljivanja od PKV

4.5.7 Obrazac Podsetnik

Obrazac Podsetnik se koristi za implementaciju čuvanja internog stanja objekta bez otkrivanja privatnih podataka objekta. Ovo se u radnom okviru iOS postiže na više načina koji su navedeni u narednim potpoglavljima [20].

4.5.7.1 Sistemske podrazumevane vrednosti

Klasa **`NSUserDefaults`** pruža interfejs za čuvanje podataka u sistemskim podrazumevanim vrednostima. Podrazumevane vrednosti su dostupne u svim delovima kôda kroz globalni unikat objekat `standardUserDefaults` i svi podaci sačuvani u podrazumevanim vrednostima biće dostupni aplikaciji kroz različite instance korišćenja te aplikacije. Podaci će biti dostupni korisniku nakon pokretanja aplikacije iako je bila isključena u prethodnoj instanci korišćenja ili je uređaj bio restartovan. U podrazumevanim vrednostima mogu biti sačuvani objekti brojevi, niske, datumi, nizovi, rečnici i instance klase `NSData`. Kada želimo u podrazumevanim vrednostima da sačuvamo neki drugi tip objekta sem navedenih objekata *Foundation* okvira, tada se najčešće taj objekat arhivira i sačuva u vidu instance klase `NSData`. Na slici 4.19 prikazan je primer čuvanja koordinata tačke u sistemskim podrazumevanim vrednostima koje su dobijene kao unikat objekat slanjem poruke `[NSUserDefaults standardUserDefaults]`.

```
- (void)sacuvajTrenutnoStanje {
    [[NSUserDefaults standardUserDefaults] setFloat:tacka.x
        forKey:@"poslednjaXkoordinata"];
    [[NSUserDefaults standardUserDefaults] setFloat:tacka.y
        forKey:@"poslednjaYkoordinata"];
}
```
Slika 4.19 Primer čuvanja objekta u sistemske podrazumevane vrednosti

Prilikom inicijalizacije sistema, čitaju se sačuvani objekti iz podrazumevanih vrednosti kako bi se podesilo poslednje sačuvano stanje i omogućila konzistentna upotreba aplikacije kroz različite instance korišćenja. Primer učitavanja prethodnog stanja prikazan je na slici 4.20.

```
- (void)ucitajPrethodnoStanje {
    float x = [[NSUserDefaults standardUserDefaults]
        floatForKey:@"poslednjaXkoordinata"];
    float y = [[NSUserDefaults standardUserDefaults]
        floatForKey:@"poslednjaYkoordinata"];
    tacka.x = x;
    tacka.y = y;
    [self nacrtajTacku:tacka];
}
```
Slika 4.20 Primer učitavanja prethodnog stanja iz sistemskih podrazumevanih vrednosti

Sistemske podrazumevane vrednosti predstavljaju listu svojstava (engl. *Property List*) koja je predviđena za čuvanje jednostavnijih podataka. Nije preporučljivo čuvati veliku količinu podataka u podrazumevanim vrednostima. Ukoliko je potrebno sačuvati sva svojstva neke klase, neophodno je iterirati kroz svojstva klase, sačuvati svako svojstvo u listu svojstava i kasnije ponovo napraviti objekat. Međutim, na ovaj način nije moguće sačuvati privatne instancne promenljive koje nisu dostupne spoljašnjim klasama. Iz tog razloga je napravljen mehanizam arhiviranja [20].

4.5.7.2 Arhiviranje

Arhiviranje je mehanizam konverzije celokupnog objekta u tok (engl. *stream*) koji može biti u celosti sačuvan i pročitan bez izlaganja privatnih promenljivih spoljašnjim klasama. Arhiviranje se omogućava implementiranjem metoda protokola `NSCoding`. Na slici 4.21 prikazan je primer implementacija metoda protokola `NSCoding` kako bi se omogućilo arhiviranje klase. Ove metode se pozivaju nad objektom prilikom arhiviranja i dearhiviranja [20].

```
- (void)encodeWithCoder:(NSCoder *)aCoder {
    [aCoder encodeObject:self.x forKey:@"xKoordinata"];
    [aCoder encodeObject:self.y forKey:@"yKoordinata"];
}

- (id)initWithCoder:(NSCoder *)aDecoder {
    self = [super init];

    if (self) {
        _x = [aDecoder decodeObjectForKey:@"xKoordinata"];
        _y = [aDecoder decodeObjectForKey:@"yKoordinata"];
    }
    return self;
}
```

Slika 4.21 Implementacija metoda protokola NSCoding za arhiviranje

Na slici 4.22 prikazan je primer korišćenja klase `NSKeyedArchiver` za arhiviranje objekata u datoteku i korišćenje klase `NSKeyedUnarchiver` za dearhiviranje objekata.

```
- (void)sacuvajTacke {
    NSString *nazivDatoteke = [NSHomeDirectory()
        stringByAppendingString:@"/Documents/tacke.bin"];
    NSData *data = [NSKeyedArchiver archivedDataWithRootObject:tacke];
    [data writeToFile:nazivDatoteke atomically:YES];
}

- (void)procitajTacke {
    NSData *data = [NSData dataWithContentsOfFile:[NSHomeDirectory()
        stringByAppendingString:@"/Documents/tacke.bin"]];
    tacke = [NSKeyedUnarchiver unarchiveObjectWithData:data];
}
```

Slika 4.22 Arhiviranje i dearhiviranje objekata

4.5.7.3 Radni okvir Core Data

Core Data je radni okvir koji definiše arhitekturu za upravljanje grafom objekata i predstavlja vid implementacije obrasca Podsetnik time što omogućava trajnost čuvanja stanja objekata.

U ovoj arhitekturi centralni objekat je kontekst - `NSManagedObjectContext`. Kontekst omogućava dohvatanje, pravljenje i čuvanje objekata definisanih u šemi Core Data.

Ova šema opisuje podatke, kojima radni okvir Core Data upravlja, u vidu entiteta, njihovih svojstava i međusobnih odnosa entiteta. Šema u radnom okviru Core Data je instanca klase `NSManagedObjectModel`. Ova šema se dizajnira na sličan način kao i šema baze podataka.

U steku tehnologija Core Data, sem konteksta (`NSManagedObjectContext`) i modela (`NSManagedObjectModel`), nalaze se i objekti `NSPersistentStoreCoordinator` i `NSPersistentContainer`. Ovi objekti imaju ulogu posrednika između model objekata i eksternih skladišta podataka poput XML datoteka i relacionih baza podataka. Uloga posrednika se ostvaruje mapiranjem podataka iz skladišta sa objektima iz konteksta [21].

4.5.8 Obrazac Komanda

Obrazac Komanda (engl. *Command*) enkapsulira radnju koja može da se prenosi između različitih objekata, sačuva ili dinamički modifikuje. Ovo se ostvaruje u radnom okviru iOS pomoću mehanizama **Izvršavanje** (engl. *Invocation*) i **Cilj-radnja** (engl. *Target-Action*). Ovim mehanizmima se postiže razdvajanje objekata koji prave zahtev za izvršavanjem radnje od objekata koji primaju zahtev i izvršavaju radnju [20].

4.5.8.1 Mehanizam izvršavanja

Mehanizam izvršavanja koristi instancu klase `NSInvocation` za čije pravljenje je neophodan objekat `NSMethodSignature` koji enkapsulira informacije o parametrima i rezultatu metode koju želimo izvršiti. Mehanizam izvršavanja se najčešće koristi kada je potrebno poslati poruku objektu u kasnijem trenutku ili kada je neophodno poslati istu poruku veći broj puta.

Objektu izvršavanja se podešavaju ciljni objekat, selektor metode i parametri metode. Objektu izvršavanja je moguće dinamički promeniti ciljni objekat kao i parametre metode dok se rezultat dobija prilikom prosleđivanja poruke `invoke` objektu tipa `NSInvocation`. Nad ovim objektom je moguće više puta proslediti poruku `invoke` sa različitim kombinacijama ciljnog objekta i parametara [21].

Na slici 4.23 prikazan je primer mehanizma izvršavanja. Parametri metode se podešavaju kao argumenti počevši od indeksa 2 zato što su ciljni objekat i selektor metode argumenti na indeksima 0 i 1. Parametri se prenose preko pokazivača.

```
NSMethodSignature *potpisMetode = [NSMutableArray
    instanceMethodSignatureForSelector:@selector(udaljenostOdTacke:)];

NSInvocation *objekatIzvrsavanja = [NSInvocation
    invocationWithMethodSignature:potpisMetode];

[objekatIzvrsavanja setTarget:tacka];
[objekatIzvrsavanja setSelector:@selector(udaljenostOdTacke:)];
[objekatIzvrsavanja setArgument:&drugaTacka atIndex:2];
[objekatIzvrsavanja invoke];

NSNumber *udaljenost;
[objekatIzvrsavanja getReturnValue:&udaljenost];
```

Slika 4.23 Primer mehanizma izvršavanja

4.5.8.2 Mehanizam Cilj-radnja

Mehanizam Cilj-radnja omogućava kontrolnom objektu (poput dugmeta ili tekstualnog polja) da šalje poruke drugom objektu usled događaja izazvanih interakcijom korisnika nad kontrolnim objektom u okviru grafičkog korisničkog interfejsa. Ove poruke prima ciljani objekat koji je u većini slučajeva kontroler pogleda. Poruka o radnji se određuje preko selektora metode [20].

Na slici 4.24 prikazan je primer registrovanja radnje `pritisnutoDugme` koja se poziva prilikom događaja dodira (`UIControlEventTouchDown`) dugmeta `mojeDugme`. Ciljani objekat u ovom primeru je trenutni kontroler pogleda (`self`) u kome je definisana metoda `pritisnutoDugme`.

```
[mojeDugme addTarget:self
              action:@selector(pritisnutoDugme:)
    forControlEvents:UIControlEventTouchDown];
```

Slika 4.24 Primer registrovanja radnje za događaj dugmeta

Na slici 4.25 prikazana je definicija metode `pritisnutoDugme`.

```
- (IBAction)pritisnutoDugme:(UIButton *)sender {
    NSLog(@"Pritisnuto dugme!");
```

```
    [tacka pomeriXkoordinatu:3.0];
}
```

Slika 4.25 Definicija radnje u mehanizmu Cilj-radnja

`IBAction` je kvalifikator koji označava da se u alatu za izgradnju interfejsa aplikacije može vizuelno spojiti kontrolni objekat sa metodom `pritisnutoDugme` i tako definisati njihovu Cilj-radnja interakciju. Parametar `sender` je definisan kao kontrolni objekat koji šalje poruku.

5 Implementacija aplikacije „WeatherMaster”

U ovom poglavlju biće predstavljena implementacija aplikacije „WeatherMaster” koja prikazuje upotrebu velikog broja koncepata programiranja mobilnih aplikacija za platformu iOS o kojima je bilo reči u teorijskom delu ovog rada. U okviru prikaza implementacije aplikacije, neće biti objašnjavani svi detalji kôda.

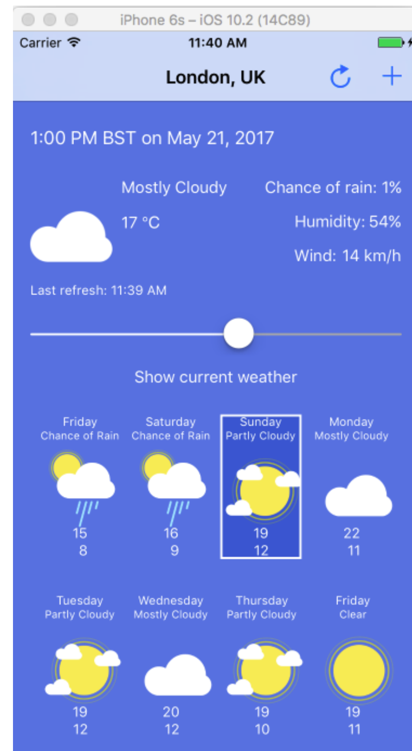
5.1 Opis aplikacije

Aplikacija „WeatherMaster” pruža korisniku informacije o vremenskim uslovima odabrane geografske lokacije i omogućava mu napredni pregled vremenske prognoze za svaki sat u toku odabranog dana. Odabrani dan može biti trenutni dan ili bilo koji od narednih sedam dana. Korisnik može odabrati trenutnu geografsku lokaciju uređaja za prikaz podataka kao i bilo koji grad u svetu. Za dobijanje trenutne geografske lokacije koristi se GPS funkcionalnost uređaja. Aplikacija omogućava čuvanje liste omiljenih geografskih lokacija za brzi pregled vremenskih uslova u tim gradovima. Meteorološke podatke aplikacija dobija od servisa *Weather Underground* koji nudi besplatni pristup podacima uz određena ograničenja.

Prilikom pokretanja aplikacije prikazuje se početni ekran (slika 5.1) koji je ujedno ekran za učitavanje (engl. *loading screen*) meteoroloških podataka poslednje odabrane geografske lokacije. Ukoliko se aplikacija pokreće prvi put, podrazumevano će biti učitani podaci za trenutnu lokaciju. Nakon učitavanja podataka, prikazuje se glavni ekran aplikacije (slika 5.2) sa kog korisnik ima pristup svim učitanim meteorološkim podacima odabrane geografske lokacije. Na slici 5.3 prikazan je glavni ekran aplikacije podeljen u sekcije. Sekcija A predstavlja traku za navigaciju aplikacije na kom se nalazi naziv odabranog grada za prikaz vremenskih uslova kao i dve dugmeta. Levo dugme u navigacionom baru služi za ručno učitavanje novih podataka sa servisa dok desno dugme prikazuje ekran liste sačuvanih lokacija (slika 5.4). U sekciji B prikazani su meteorološki podaci o odabranoj lokaciji za odabrani dan iz sekcije D i odabrani sat iz sekcije C. Prikazani podaci predstavljeni su informacijom o odabranom času i danu za prikaz podataka, stanjem vremenskih prilika, temperaturom, mogućnošću padavina, vlažnošću, jačinom vetra i ikonicom koja predstavlja stanje vremenskih prilika. U sekciji C nalazi se kontrolni objekat klizača (engl. *slider*) kojim možemo odrediti za koji sat u toku odabranog dana (iz sekcije D) želimo da vidimo podatke u sekciji B. Pozicijom klizačem menjamo vrednost odabranog sata pri čemu skroz leva pozicija klizača označava ponoć a skroz desna pozicija 23 časa istog dana. Klizač ima 24 vrednosti osim u slučaju odabira

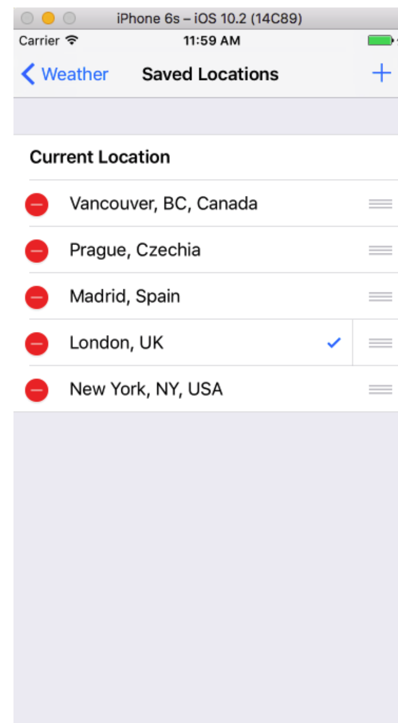
trenutnog dana u sekciji D kada je najmanja vrednost klizača trenutni sat. U sekciji D prikazana je opšta dnevna vremenska prognoza za trenutni dan i narednih 7 dana.

Slika 5.1 Početni ekran



Slika 5.2 Glavni ekran

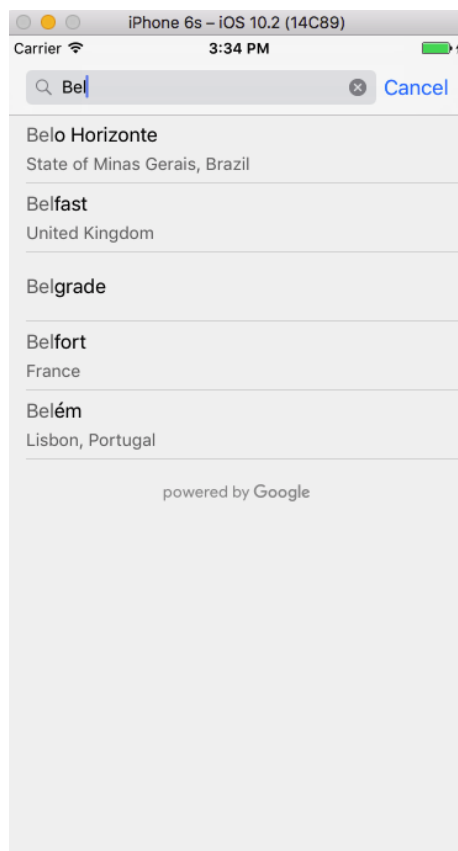
Slika 5.3 Sekcije glavnog ekrana



Slika 5.4 Ekran liste sačuvanih lokacija

Opšta dnevna vremenska prognoza predstavljena je nazivom dana u nedelji, prosečnim stanjem vremenskih uslova tog dana, ikonicom koja predstavlja to stanje kao i maksimalna i minimalna temperatura tog dana. Odabirom dana u sekciji D, klizač se postavlja na najmanju vrednost (pozicija skroz levo) i klizačem se tada za odabrani dan menja sat za koji je prikazana vremenska prognoza u sekciji B. U sekciji C se nalazi i dugme „*Show current weather*" koje ima funkciju vraćanja odabranog dana na trenutni dan i vraćanja klizača na najmanju vrednost (trenutni sat).

Na slici 5.4 prikazan je ekran liste sačuvanih lokacija koji omogućava odabir lokacije za prikaz podataka na glavnom ekranu. Prva stavka na ovoj listi je uvek trenutna lokacija dok se ispod nalaze ostale sačuvane lokacije. Svaka stavka ostalih sačuvanih lokacija ima crveno dugme sa leve strane za brisanje sa liste kao i ručku (engl. *handle*) sa desne strane za menjanje redosleda prikaza lokacija u listi. Stavka koja ima oznaku plave kukice (engl. *checkmark*) pored ručke predstavlja poslednju odabranu lokaciju. Odabirom neke stavke sa liste prikazuje se glavni ekran sa učitanim podacima za novoodabranu lokaciju. U navigacionom baru ekrana liste sačuvanih lokacija imamo sa leve strane dugme za povratak na glavni ekran pri čemu se ponovo učitavaju podaci za poslednju odabranu lokaciju. U ovom baru se sa desne strane nalazi dugme za dodavanje lokacije na listu sačuvanih lokacija. Pritiskom ovog dugmeta otvara se ekran za pretraživanje gradova (slika 5.5).



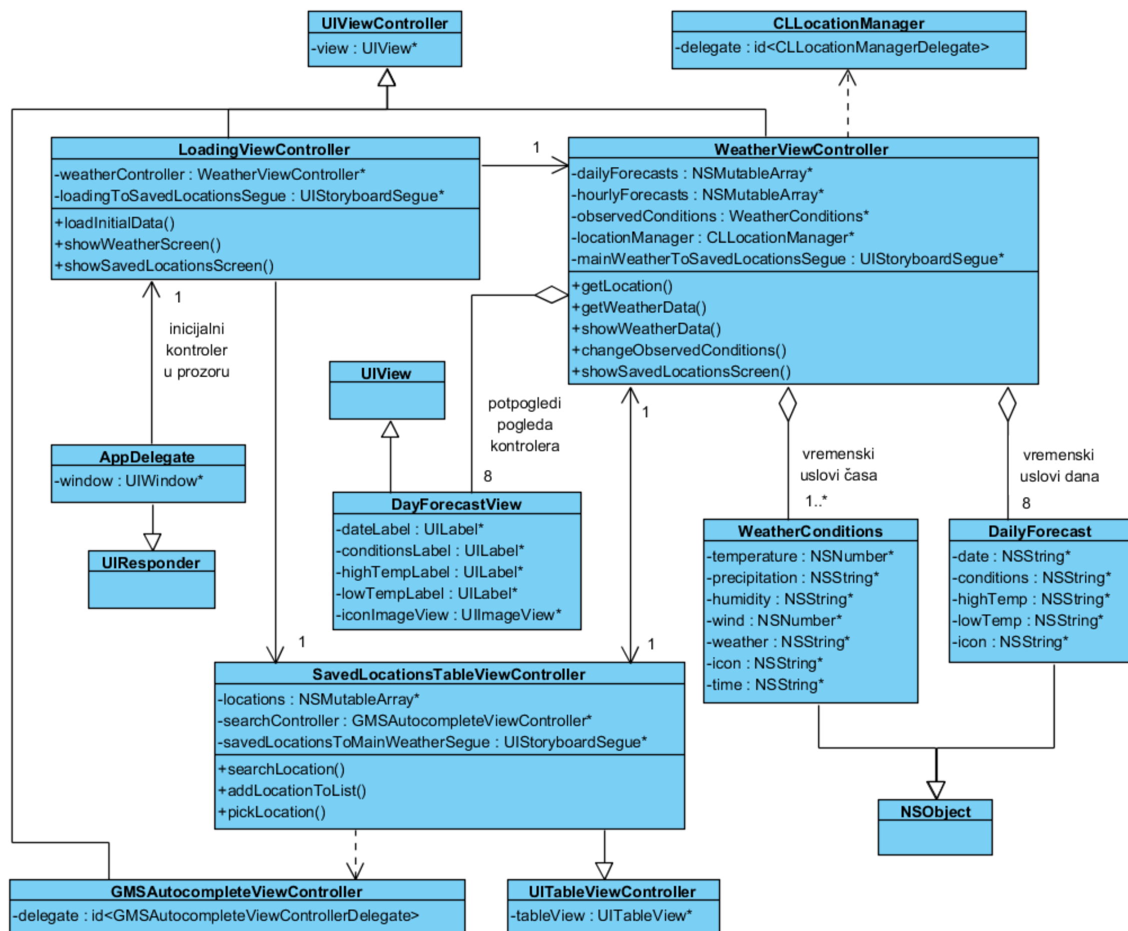
Slika 5.5 Ekran za pretraživanje gradova

Na ekranu za pretraživanje gradova moguće je vršiti pretragu svih gradova sveta sa ponuđenim rezultatima koji dopunjuju trenutno napisani pojam pretrage. Odabirom grada sa ove liste rezultata prikazuje se glavni ekran aplikacije sa učitanim podacima za odabrani grad. Dugme „*Cancel*" predstavlja dugme za povratak na ekran liste sačuvanih lokacija.

Osim ručnog učitavanja novih meteoroloških podataka sa servisa, aplikacija automatski osvežava ove informacije prilikom svakog pokretanja aplikacije ili ponovnog otvaranja već pokrenute aplikacije. Vreme poslednjeg osvežavanja ovih podataka je prikazno u sekciji B (engl. *last refresh*).

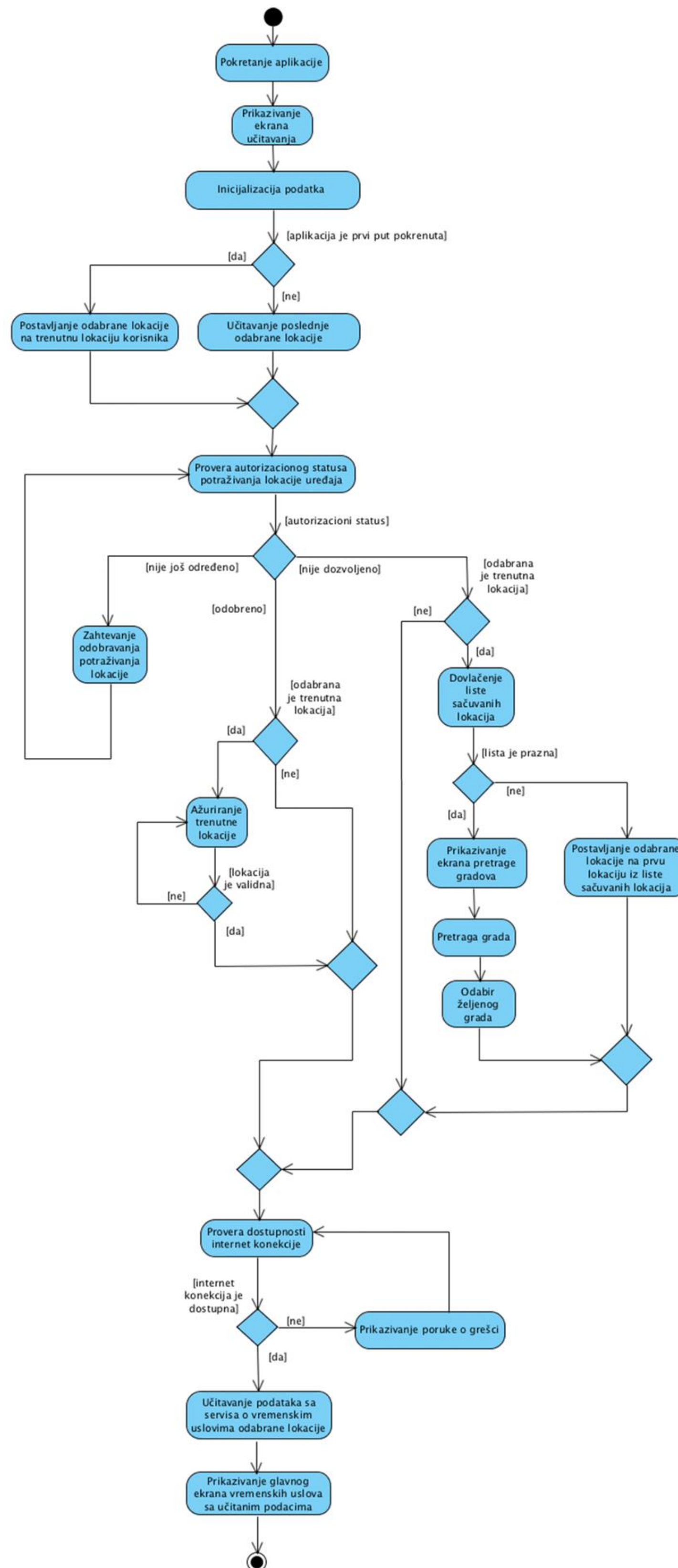
Lista sačuvanih lokacija kao i informacija o poslednjoj odabranoj lokaciji se čuvaju u sistemskim podrazumevanim vrednostima uređaja tako da će biti dostupni korisniku prilikom novog pokretanja aplikacije i nakon što se prethodno aplikacija isključila ili uređaj restartovao.

Na slici 5.6 predstavljen je dijagram klasa ove aplikacije radi boljeg prikaza kompletne strukture aplikacije pre ulaska u detalje same implementacije. Navedeni atributi klasa u dijagramu predstavljaju ključna svojstva ovih klasa, dok operacije u ovom dijagramu predstavljaju uopštenja postojećih metoda u klasama.



Slika 5.6 Dijagram klasa

Na slici 5.7 predstavljen je niz akcija, u vidu dijagrama aktivnosti, koje se izvršavaju radi prikaza podataka o trenutnim vremenskim uslovima, od trenutka pokretanja aplikacije.



Slika 5.7 Dijagram aktivnosti

5.2 Klase sloja model

U model sloju aplikacije nalaze se dve klase.

Klasa **`WeatherConditions`** enkapsulira informacije o vremenskim uslovima za određeni sat jednog dana i njen interfejs je prikazan na slici 5.8.

```
@interface WeatherConditions : NSObject

@property (strong, nonatomic) NSNumber *temperature;
@property (strong, nonatomic) NSString *precipitation;
@property (strong, nonatomic) NSString *humidity;
@property (strong, nonatomic) NSNumber *wind;
@property (strong, nonatomic) NSString *weather;
@property (strong, nonatomic) NSString *icon;
@property (strong, nonatomic) NSString *time;

@end
```

Slika 5.8 Interfejs WeatherConditions klase

Kod velikog broja svojstava u aplikaciji postavljeni su atributi `strong` i `nonatomic`. Atribut `strong` je naveden kako bi objekat svojstva imao bar jednog vlasnika i ne bi bio prerano oslobođen iz memorije. Atribut `nonatomic` je naveden radi efikasnijeg izvršavanja. Ovim svojstvima ne pristupa istovremeno više niti pa im nije neophodna atomičnost.

Klasa **`DailyForecast`** enkapsulira informacije o jednoj dnevnoj vremenskoj prognozi i njen interfejs je prikazan na slici 5.9.

```
@interface DailyForecast : NSObject

@property (strong, nonatomic) NSString *date;
@property (strong, nonatomic) NSString *conditions;
@property (strong, nonatomic) NSString *highTemp;
@property (strong, nonatomic) NSString *lowTemp;
@property (strong, nonatomic) NSString *icon;

@end
```

Slika 5.9 Interfejs DailyForecast klase

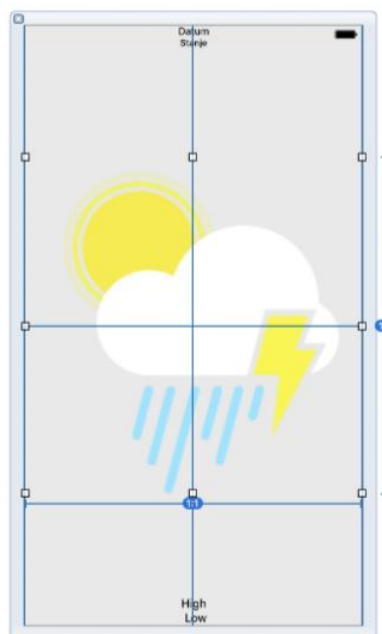
5.3 Klase sloja pogled

Pogled sloj aplikacije predstavljen je ekranima aplikacije odnosno pogledima kojima kontroleri pogleda upravljaju. Ovi pogledi sačinjeni su od velikog broja potpogleda (engl. *subviews*) koji su prisutni na jednom ekranu. Najveći broj ovih potpogleda su ugrađene klase radnog okvira `UIKit` poput labele (`UILabel`), dugmeta (`UIButton`), klizača (`UISlider`), slike (`UIImageView`) i tabele (`UITableView`). Ovo su potklase

`UIView` klase koja definiše pravougaonu površinu na ekranu sa metodama za upravljanje sadržajem te površine.

5.3.1 Pogled DayForecastView

Osim korišćenja pogleda iz radnog okvira `UIKit`, napravljena je klasa `DayForecastView` koja prikazuje podatke o jednoj dnevnoj prognozi. Za klasu pogleda pravi se i datoteka sa ekstenzijom **.xib** koja služi za specifikaciju dizajna pogleda preko alata za izgradnju interfejsa. Na slici 5.10 prikazan je napravljeni pogled `DayForecastView` u alatu za izgradnju interfejsa preko .xib datoteke.



Slika 5.10 Interfejs pogleda DayForecastView

Za izgradnju univerzalnog korisničkog interfejsa, koji odgovara svim dimenzijama ekrana uređaja, koristi se sistem ***Auto Layout***. Ovaj sistem je podrazumevano uključen u alatu za izgradnju interfejsa i omogućava dinamičko računanje veličina i pozicija svih pogleda na osnovu postavljenih ograničenja (engl. *constraints*) nad tim pogledima [22].

Da bi *Auto Layout* sistem uspešno prikazao grafički interfejs neophodno je za svaki element pogleda definisati ograničenja koja definišu horizontalnu i vertikalnu poziciju pogleda u roditeljskom pogledu (engl. *superview*) kao i širinu i visinu pogleda. Postoje različite vrste ograničenja koja mogu definisati poziciju i veličinu pogleda poput definisanja rastojanja od drugih pogleda (gore, dole, levo i desno), poravnanje ivica ili centra sa drugim pogledima, izjednačavanje visine i širine sa drugim pogledima itd. U pogledu `DayForecastView` na slici 5.10 definisana su

naredna ograničenja za pogled slike ikonice: horizontalno i vertikalno centriranje sa roditeljskim pogledom, nepostojanje levog i desnog udaljenja od roditeljskog pogleda, 1:1 odnos (engl. *aspect ratio*) visine i širine. Ovim ograničenjima je *Auto Layout* sistem u mogućnosti da tačno izračuna dimenziju i poziciju pogleda slike dinamički u okviru pogleda `DayForecastView` na osnovu njenih dimenzija i pozicije u pogledu nekog kontrolera pogleda. Za ostale elemente pogleda `DayForecastView` takođe su definisana neophodna ograničenja.

Pored .xib datoteke, za svaki pogled pravi se i .h i .m datoteke. U ovim datotekama nalaze se deklaracije potpogleda kao i metode za učitavanje sadržaja .xib datoteke u interni pogled klase `DayForecastView`. Na slici 5.11 prikazan je interfejs a na slici 5.12 implementacija klase `DayForecastView`. Kôd na slici 5.12 je ručno napisan kako bi se omogućila upotrebljivost ovog pogleda u drugim pogledima aplikacije.

```
@interface DayForecastView : UIView

@property (strong, nonatomic) IBOutlet UIView *view;
@property (weak, nonatomic) IBOutlet UILabel *dateLabel;
@property (weak, nonatomic) IBOutlet UILabel *conditionsLabel;
@property (weak, nonatomic) IBOutlet UILabel *highTempLabel;
@property (weak, nonatomic) IBOutlet UILabel *lowTempLabel;
@property (weak, nonatomic) IBOutlet UIImageView *iconImageView;

@end
```

Slika 5.11 Interfejs klase DayForecastView

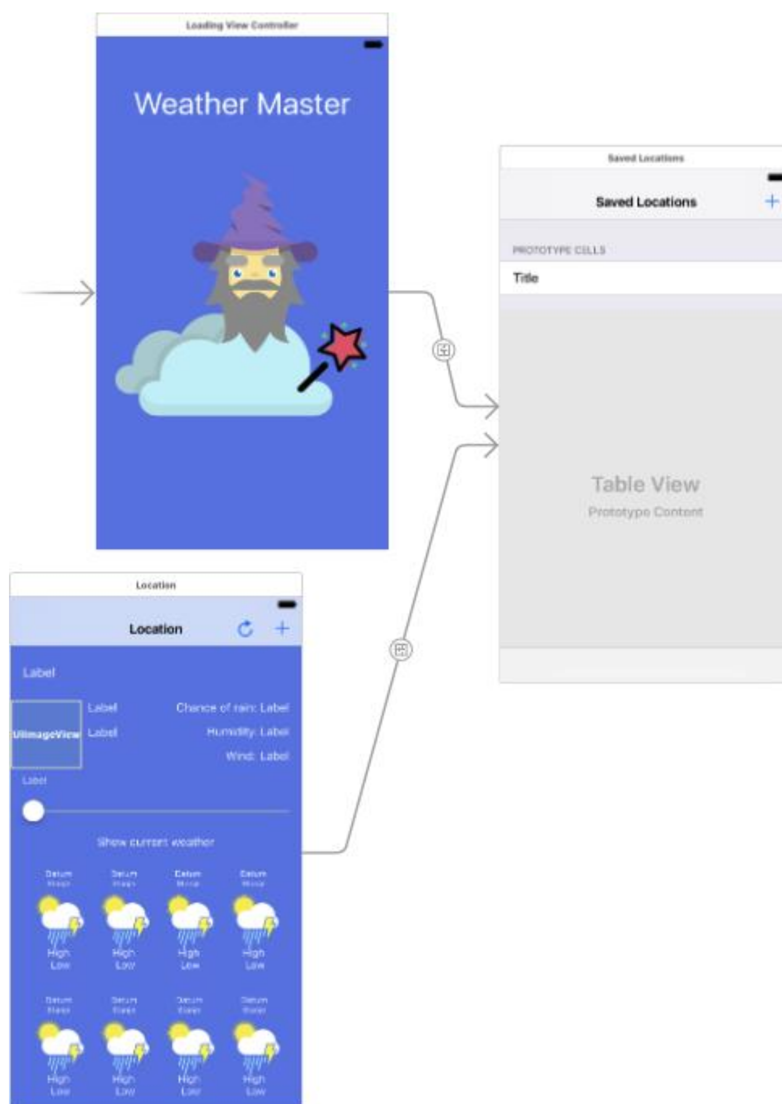
```
@implementation DayForecastView


- (id)initWithCoder:(NSCoder *)aDecoder{
    self = [super initWithCoder:aDecoder];

    if (self) {
        [self xibSetup];
    }
    return self;
}

- (instancetype)initWithFrame:(CGRect)frame {
    self = [super initWithFrame:frame];

    if (self) {
        [self xibSetup];
    }
    return self;
}


- (void)xibSetup {
    [[NSBundle bundleForClass:[self class]] loadNibNamed:NSStringFromClass([self class])
owner:self options:nil];
    [self addSubview:self.view];
    [self.view setTranslatesAutoresizingMaskIntoConstraints:NO];
    [self addConstraints:[NSLayoutConstraint constraintsWithVisualFormat:@"H:|[view]|"
options:0 metrics:nil views:@{@"view":self.view}]];
    [self addConstraints:[NSLayoutConstraint constraintsWithVisualFormat:@"V:|[view]|"
options:0 metrics:nil views:@{@"view":self.view}]];
}
```

```
@end
```

Slika 5.12 Implementacija klase DayForecastView

5.3.2 *Komponenta aplikacije Storyboard*

Svaka iOS aplikacija ima komponentu koja se naziva ***Storyboard***. Ova komponenta služi za specifikaciju dizajna pogleda svih kontrolera pogleda. Svaki pogled kontrolera pogleda predstavlja jedan ekran (engl. *screen*) aplikacije. *Storyboard* komponenta se uređuje kao i .xib datoteka u alatu za izgradnju interfejsa. Kao i kod dizajniranja pojedinačnog pogleda .xib datoteke, koriste se ograničenja u *Storyboard* komponenti za izgradnju dizajna svakog ekrana. *Storyboard* komponente nude i mogućnost definisanja prelaza (engl. ***segue***) između ekrana. Na slici 5.13 prikazan je izgrađeni *Storyboard* aplikacije u alatu za izgradnju interfejsa [23].



Slika 5.13 Storyboard aplikacije

Strelica pored gornjeg ekrana predstavlja ulaznu tačku aplikacije. Ulazna tačka aplikacije je pogled klase **LoadingViewController** definisan kao ekran učitavanja. Ukoliko nije omogućena GPS funkcionalnost uređaja, vrši se prelaz sa početnog pogleda na desni pogled liste sačuvanih lokacija kako bi korisnik odabrao lokaciju sa liste. Ovo je pogled klase **SavedLocationsTableViewController**. Prelaz je, u *Storyboard* komponenti, predstavljen strelicom između dva ekrana. Ukoliko GPS funkcionalnost jeste omogućena, vrši se prelaz na donji glavni ekran aplikacije koji je predstavljen pogledom **WeatherViewController** klase. Prelaz nije definisan u *Storyboard* komponenti zato što se izvršava u samom kôdu. Prelaz sa glavnog ekrana aplikacije na ekran liste sačuvanih lokacije definisan je u *Storyboard* komponenti strelicom i izvršava se automatski prilikom pritiska na taster. Pogled `WeatherViewController` klase, sem standardnih pogleda iz radnog okvira UIKit, sadrži i osam instanci pogleda `DayForecastView` u svojoj donjoj polovini.

5.4 Klase sloja kontroler

Kao što je spomenuto u poglavlju 5.3.2, postoje tri klase kontrolera pogleda: `LoadingViewController`, `WeatherViewController` i `SavedLocationsTableViewController`. Ove klase nasleđuju klasu `UIViewController` i imaju internu instancu `UIView` klase koja predstavlja jedan ekran aplikacije kojim taj kontroler upravlja.

5.4.1 Klasa LoadingViewController

Klasa `LoadingViewController` upravlja ekranom učitavanja.

U ekstenziji ove klase nalazi se instanca `WeatherViewController` klase (slika 5.14).

```
@interface LoadingViewController ()

@property (strong, nonatomic) WeatherViewController *weatherController;

@end
```

Slika 5.14 Ekstenzija LoadingViewController klase

Nakon prikazivanja ovog ekrana, inicijalizuje se instanca `WeatherViewController` klase i njena svojstva. Poslednja odabrana lokacija se čita iz sistemskih podrazumevanih vrednosti i učitavaju se podaci o vremenskim uslovima poslednje odabrane geografske lokacije sa servisa *Weather Underground* (slika 5.15).

```
- (void)viewDidAppear:(BOOL)animated {
    [super viewDidAppear:animated];
```

```
    UIStoryboard *storyboard = [UIStoryboard storyboardWithName:@"Main" bundle: nil];
    self.weatherController = [storyboard
instantiateViewControllerWithIdentifier:@"WeatherViewController"];

    [self.weatherController initProperties];

    // slucaj kada nije odabrana trenutna lokacija u poslednjem koriscenju aplikacije
    if (self.weatherController.locationIndex > 0) {
        [self.weatherController createLocationFromDefaults];
        [self.weatherController changeWeather];

        if (!self.weatherController.isNetworkError) {
            [self.weatherController presentController];
        } else {
            [self showAlert];
        }
    }
}
```

Slika 5.15 Inicijalizacija instance WeatherViewController klase

U slučaju greške usled nepostojanja internet konekcije prikazuje se greška pozivom metode `showAlert` (slika 5.16). Pravi se instanca klase `UIAlertController` iz radnog okvira `UIKit` koja je zadužena za prikaz poruke o grešci. Ova poruka će se prikazivati dok god se ne uspostavi internet konekcija. Instanci klase `UIAlertController` se dodeljuje akcija definisana kao instanca klase `UIAlertAction`. Akcija je predstavljena u vidu dugmeta i radnje koja se izvršava pritiskom na dugme.

```
- (void)showAlert {
    // prikaz greske o nemogucnosti uspostavljanja internet konekcije
    UIAlertController* alert = [UIAlertController
          alertControllerWithTitle:@"Network problem"
          message:@"There was a problem with your network connectivity. Try enabling
cellular data or WiFi."
          preferredStyle:UIAlertControllerStyleAlert];

    UIAlertAction* defaultAction = [UIAlertAction
          actionWithTitle:@"OK"
          style:UIAlertActionStyleDefault
          handler:^(UIAlertAction * action) {
            [self.weatherController changeWeather];

            if (!self.weatherController.isNetworkError) {
                   dispatch_async(dispatch_get_main_queue(), ^{
                            [self.weatherController presentController];
                   });
            } else {
                   // ponavljanje greske dok se konekcija ne uspostavi
                   [self showAlert];
            }
          }];

    [alert addAction:defaultAction];
    [self presentViewController:alert animated:YES completion:nil];
}
```

Slika 5.16 Implementacija metode showAlert

U implementaciji metode `showAlert` prikazana je upotreba GCD funkcije za asinhrono izvršavanje. Svaka interakcija sa korisničkim interfejsom mora biti

izvršena na glavnoj niti aplikacije. Funkcija `dispatch_async`, sa prosleđenim redom na glavnoj niti aplikacije, osigurava izvršavanje na glavnoj niti što je neophodno u slučaju poziva metode za prikaz kontrolera u bloku akcije.

5.4.2 Klasa WeatherViewController

Klasa `WeatherViewController` upravlja glavnim ekranom aplikacije.

U ekstenziji klase definisana su interna svojstva ove klase (slika 5.17).

```
@interface WeatherViewController ()

@property (weak, nonatomic) IBOutlet UILabel *tempLabel;
@property (weak, nonatomic) IBOutlet UILabel *precipLabel;
@property (weak, nonatomic) IBOutlet UILabel *humidityLabel;
@property (weak, nonatomic) IBOutlet UILabel *windLabel;
@property (weak, nonatomic) IBOutlet UILabel *weatherLabel;
@property (weak, nonatomic) IBOutlet UILabel *timeLabel;
@property (weak, nonatomic) IBOutlet UILabel *lastRefreshLabel;
@property (weak, nonatomic) IBOutlet UIImageView *weatherImage;
@property (weak, nonatomic) IBOutlet UISlider *slider;
@property (weak, nonatomic) IBOutlet UIButton *currentWeatherButton;

@property (strong, nonatomic) CLLocationManager *locationManager;
@property (strong, nonatomic) CLLocation *location;
@property BOOL didFindLocation;
@property (strong, nonatomic) NSDate *lastRefresh;
@property BOOL locationEnabled;

@property (strong, nonatomic) WeatherConditions *observedConditions;

@property (strong, nonatomic) NSString *city;
@property (strong, nonatomic) WeatherConditions *currentConditions;

@property (strong, nonatomic) NSArray *dailyForecast;
@property (strong, nonatomic) NSArray *hourlyForecast;

@property (strong, nonatomic) NSMutableArray *dailyForecasts;
@property (strong, nonatomic) NSMutableArray *hourlyForecasts;

@property (strong, nonatomic) NSMutableArray *sliderHours;
@property NSUInteger lastIndex;

@property int hour;
@property int daySelected;

@end
```

Slika 5.17 Ekstenzija klase WeatherViewController

U metodi `initProperties` vrši se inicijalizacija svojstava (slika 5.18).

```
- (void)initProperties {
    self.controllerPresented = NO;

    self.observedConditions = [[WeatherConditions alloc] init];

    self.lastRefresh = [NSDate distantPast];
```

```
    self.sliderHours = [[NSMutableArray alloc] init];
    self.lastIndex = 0;

    self.slider.minimumValue = 0;
    self.slider.continuous = YES;

    self.currentWeatherButton.hidden = YES;

    self.hourlyForecasts = [[NSMutableArray alloc] initWithCapacity:8];
    self.dailyForecasts = [[NSMutableArray alloc] initWithCapacity:8];

    // dovlacenje podataka o poslednjoj izabranoj lokaciji iz sistemskih podrazumevanih
       vrednosti
    NSNumber *locationIndex = [[NSUserDefaults standardUserDefaults]
         objectForKey:@"selectedLocationIndex"];

    if (locationIndex == nil) {

        // slucaj prvog pokretanja aplikacije; podesavanje na trenutnu lokaciju
        self.locationIndex = 0;
        [[NSUserDefaults standardUserDefaults] setInteger:0
                  forKey:@"selectedLocationIndex"];
        [[NSUserDefaults standardUserDefaults] synchronize];

    } else {
        self.locationIndex = [locationIndex intValue];
    }

    // inicijalizacija CoreLocation Manager-a
    self.locationManager = [[CLLocationManager alloc] init];
    self.locationManager.delegate = self;
    self.locationManager.desiredAccuracy = kCLLocationAccuracyKilometer;
}
```

Slika 5.18 Inicijalizacija svojstava

Za dobijanje podataka o trenutnoj lokaciji uređaja pomoću GPS funkcionalnosti, koristi se ***CoreLocation*** radni okvir. Klasa `CLLocationManager` je zadužena za dostavljanje podataka o trenutnoj lokaciji uređaja svom delegatu. Prilikom inicijalizacije `CLLocationManager` klase u `initProperties` metodi, u delegat objektu se poziva metoda sa slike 5.19.

```
- (void)locationManager:(CLLocationManager *)manager
         didChangeAuthorizationStatus:(CLAuthorizationStatus)status {

    // status nije jos odredjen
    if (status == kCLAuthorizationStatusNotDetermined) {
        // zahtev za odobrenje potrazivanja lokacije
        [self.locationManager requestWhenInUseAuthorization];

    // odobreno koriscenje lokacije
    } else if (status == kCLAuthorizationStatusAuthorizedWhenInUse && [CLLocationManager
locationServicesEnabled]) {
        self.didFindLocation = NO;
        self.locationEnabled = YES;

        if (self.locationIndex == 0) {
            // pokretanje potrazivanja lokacije
            [self.locationManager startUpdatingLocation];
        }

    // nije dozvoljeno koriscenje lokacije
    } else if (status == kCLAuthorizationStatusDenied || status ==
kCLAuthorizationStatusRestricted || ![CLLocationManager locationServicesEnabled]) {
```

```
        self.locationEnabled = NO;

        if (self.locationIndex == 0) {
            [self processDisabledLocation];
        }
    }
}
```

Slika 5.19 Promena autorizacionog statusa locationManager-a

Ukoliko nije određena autorizacija korišćenja lokacije uređaja, prvo se zahteva autorizacija metodom `requestWhenInUseAuthorization`. Ukoliko je korišćenje lokacije odobreno, započinje se ažuriranje lokacije u aplikaciji metodom `startUpdatingLocation`. Ukoliko korišćenje lokacije nije odobreno, ovaj slučaj se obrađuje metodom `processDisabledLocation` (slika 5.20). Klasa WeatherViewController implementira protokole `CLLocationManagerDelegate` i `SavedLocationsDelegate` (slika 5.21).

```
- (void)processDisabledLocation {
    // dovlacenje liste sacuvanih lokacija iz sistemskih podrazmevanih vrednosti
    NSArray *savedLocations = [[NSUserDefaults standardUserDefaults]
arrayForKey:@"savedLocations"];

    if ([savedLocations count] > 0) {
        // postavljanje odabrane geografske lokacije na prvu lokaciju iz liste sacuvanih
lokacija
        self.locationIndex = 1;
        [[NSUserDefaults standardUserDefaults] setInteger:1
forKey:@"selectedLocationIndex"];
        [[NSUserDefaults standardUserDefaults] synchronize];

        [self createLocationFromDefaults];
        [self changeWeather];
        if (!self.controllerPresented && !self.isNetworkError) {
            [self presentController];
        }
        if (self.isNetworkError) {
            [self showAlert];
        }

    } else {
        if (!self.controllerPresented) {
            [self presentController];
        }
        // preusmeravanje na ekran liste sacuvanih lokacija u slucaju da je prazna
        [self performSegueWithIdentifier:@"MainWeatherToSavedLocationsSegue" sender:self];
    }
}

- (void)presentController {

    UINavigationController *nav = [[UINavigationController alloc]
         initWithRootViewController:self];
    UIViewController *root = [UIApplication
         sharedApplication].keyWindow.rootViewController;
    [root presentViewController:nav animated:YES completion:nil];

    self.controllerPresented = YES;
}
```

Slika 5.20 Obrađivanje slučaja neodobrene lokacije

```
@interface WeatherViewController : UIViewController <CLLocationManagerDelegate,
SavedLocationsDelegate>
```
Slika 5.21 Navođenje protokola u interfejsu WeatherViewController klase

Nakon pokretanja ažuriranja lokacije u aplikaciji metodom `startUpdatingLocation`, u delegat objektu se poziva metoda sa slike 5.22 prilikom svake promene lokacije.

```
- (void)locationManager:(CLLocationManager *)manager
        didUpdateLocations:(NSArray<CLLocation *> *)locations {

    if (!self.didFindLocation) {

        CLLocation *newLocation = [locations lastObject];

        // provera starosti odredjivanja lokacije da ne bismo koristili kesirane vrednosti
        NSTimeInterval locationAge = -[newLocation.timestamp timeIntervalSinceNow];
        if (locationAge > 60.0) {
            return;
        }

        // provera da li je doslo do nepravilnosti u merenju
        if (newLocation.horizontalAccuracy < 0) {
            return;
        }

        self.didFindLocation = YES;
        // zaustavljanje azuriranja lokacije nakon prvog dobrog odedjivanja
        [manager stopUpdatingLocation];

        self.location = newLocation;
        [self changeWeather];

        if (!self.controllerPresented && !self.isNetworkError) {
            [self presentController];
        }

        if (self.isNetworkError) {
            [self showAlert];
        }
    }
}
```
Slika 5.22 Delegat metoda ažuriranja lokacije

Ukoliko je došlo do greške prilikom ažuriranja lokacije, u delegat objektu se poziva metoda sa slike 5.23.

```
- (void)locationManager:(CLLocationManager *)manager didFailWithError:(NSError *)error {
    NSString *errorDesc = [error localizedDescription];
    NSLog(@"Error finding location: %@", errorDesc);
}
```
Slika 5.23 Delegat metoda greške pri ažuriranju lokacije

Nakon određivanja željene lokacije prikazivanja informacija o vremenskim uslovima, poziva se metoda `changeWeather` koja učitava podatke sa *Weather Undeground* servisa koristeći internet konekciju (slika 5.24).

```
- (void)changeWeather {
    NSError *error;
```

```
    NSString *urlString = [NSString
         stringWithFormat:@"http://api.wunderground.com/api/44b0fe0467fed995/hourly10day/fo
recast10day/conditions/q/%.6f,%.6f.json",
         self.location.coordinate.latitude,
         self.location.coordinate.longitude];

    // ucitavanje podataka sa servisa
    NSData *data = [NSData dataWithContentsOfURL:[NSURL URLWithString:urlString]
         options:NSDataReadingUncached error:&error];

    if (error) {
        NSLog(@"%@", [error localizedDescription]);
        self.isNetworkError = YES;

    } else {
        // popunjavanje pratecih nizova vremenskih uslova na osnovu dobijenih podataka
        NSLog(@"Data has loaded successfully.");
        self.isNetworkError = NO;

        self.lastRefresh = [NSDate date];

        NSDictionary *json = [NSJSONSerialization
                  JSONObjectWithData:data options:kNilOptions error:&error];
        NSDictionary *currentObservation = json[@"current_observation"];
        self.dailyForecast = json[@"forecast"][@"simpleforecast"][@"forecastday"];
        self.hourlyForecast = json[@"hourly_forecast"];

        NSString *currentHour = self.hourlyForecast[0][@"FCTTIME"][@"hour"];
        self.hour = [currentHour intValue];
        [self setHourlyArray];

        [self createDailyForecasts];

        if (self.locationIndex == 0) {
            self.city = [NSString
                  stringWithFormat:@"%@ (Current location)",
                  currentObservation[@"display_location"][@"city"]];
        } else {
            self.city = [[NSUserDefaults standardUserDefaults]
                  arrayForKey:@"savedLocations"][self.locationIndex - 1][@"name"];
        }

        [self createCurrentConditions:currentObservation];

        // podesavanje elemenata grafickog interfejsa na osnovu dobijenih podataka
        [self showCurrentWeather];
    }
}
```

Slika 5.24 Učitavanje podataka sa servisa

Weather Underground servis dostavlja podatke u vidu **.json** datoteke. Na prethodnoj slici 5.24 prikazana je konverzija JSON objekta u standardni rečnik radnog okvira *Foundation*. Iz rečnika se učitavaju podaci za trenutne vremenske uslove, dnevne vremenske prognoze i vremenske prognoze po časovima za odabranu lokaciju.

Na slici 5.25 prikazane su implementacije pomoćnih metoda za pravljenje nizova klasa koje predstavljaju vremenske prognoze za jednu lokaciju u različitim vremenskim intervalima, kao i za prikazivanje povezanih pogleda na glavnom ekranu aplikacije.

```
- (void)setHourlyArray {
```

```
    [self.hourlyForecasts removeAllObjects];

    // popunjavanje niza podataka o vremenskim uslovima svakog sata za 8 dana (danasnji i
    // narednih 7 dana)
    for (int i=0; i<8; i++) {
        NSMutableArray *hourArray;
        // specifican slucaj za danasnji dan
        if (i==0) {
            // podaci za preostale sate danasnjeg dana
            hourArray = [[NSMutableArray alloc] initWithCapacity:(24-self.hour)];
            for (int j=0; j<24-self.hour; j++) {
                [hourArray addObject:[self createHourlyWeatherConditionsForIndex:j]];
            }
        } else {
            // podaci za svih 24 sata narednih 7 dana
            hourArray = [[NSMutableArray alloc] initWithCapacity:24];
            for (int j=0; j<24; j++) {
                int index = 24-self.hour+(i-1)*24+j;
                [hourArray addObject:[self createHourlyWeatherConditionsForIndex:index]];
            }
        }
        [self.hourlyForecasts addObject:hourArray];
    }
}

- (WeatherConditions *)createHourlyWeatherConditionsForIndex:(int)index {
    // pravljenje jednog objekta vremenskih uslova sata
    WeatherConditions *weather = [[WeatherConditions alloc] init];
    weather.temperature = [NSNumber
         numberWithInteger:[self.hourlyForecast[index][@"temp"][@"metric"] integerValue]];
    weather.precipitation = self.hourlyForecast[index][@"pop"];
    weather.humidity = [NSString stringWithFormat:@"%@%%",
         self.hourlyForecast[index][@"humidity"]];
    weather.wind = [NSNumber
         numberWithInteger:[self.hourlyForecast[index][@"wspd"][@"metric"] integerValue]];
    weather.weather = self.hourlyForecast[index][@"condition"];
    weather.icon = [self getIconName:self.hourlyForecast[index]];
    weather.time = self.hourlyForecast[index][@"FCTTIME"][@"pretty"];

    return weather;
}

- (void)createDailyForecasts {
    [self.dailyForecasts removeAllObjects];

    // popunjavanje niza podataka o dnevnim vremenskim uslovima za 8 dana (danasnji i
    // narednih 7 dana)
    for (int i=0; i<8; i++) {
        DailyForecast *daily = [self createDailyForecastForIndex:i];
        [self.dailyForecasts addObject:daily];
    }
    [self setDayViews];
}

- (DailyForecast *)createDailyForecastForIndex:(int)index {
    // pravljenje jednog objekta vremenskih uslova dana
    DailyForecast *daily = [[DailyForecast alloc] init];
    daily.date = self.dailyForecast[index][@"date"][@"weekday"];
    daily.conditions = self.dailyForecast[index][@"conditions"];
    daily.highTemp = self.dailyForecast[index][@"high"][@"celsius"];
    daily.lowTemp = self.dailyForecast[index][@"low"][@"celsius"];
    daily.icon = [self getIconName:self.dailyForecast[index]];

    return daily;
}
```

```
- (void)setDayViews {
    // podesavanje pogleda svih dana na glavnom ekranu aplikacije
    for (int i=0; i<8; i++) {
        DayForecastView *view = [self.view viewWithTag:(100+i)];
        [self setDayView:view forIndex:i];
    }
}

- (void)setDayView:(DayForecastView *)view forIndex:(int)index {
    // podesavanje jednog pogleda dana na glavnom ekranu aplikacije
    DailyForecast *forecast = self.dailyForecasts[index];
    view.dateLabel.text = forecast.date;
    view.conditionsLabel.text = forecast.conditions;
    view.highTempLabel.text = forecast.highTemp;
    view.lowTempLabel.text = forecast.lowTemp;
    view.iconImageView.image = [UIImage imageNamed:forecast.icon];
}

- (void)createCurrentConditions:(NSDictionary *)currentObservation {
    // podesavanje objekta trenutnih vremenskih uslova
    self.currentConditions = [[WeatherConditions alloc] init];
    self.currentConditions.temperature = currentObservation[@"temp_c"];
    self.currentConditions.precipitation = currentObservation[@"precip_1hr_metric"];
    self.currentConditions.humidity = currentObservation[@"relative_humidity"];
    self.currentConditions.wind = currentObservation[@"wind_kph"];
    self.currentConditions.weather = currentObservation[@"weather"];
    self.currentConditions.icon = [self getIconName:currentObservation];
    self.currentConditions.time = @"Current weather";
}

- (void)showCurrentWeather {
    // podesavanje trenutnih vremenskih uslova
    [self setObservedAsConditions:self.currentConditions];
    [self selectDay:0];

    // dugme za prikaz trenutnog vremena se sakriva jer jer trenutno vreme vec prikazano
    self.currentWeatherButton.hidden = YES;

    NSString *dateString = [NSDateFormatter
         localizedStringFromDate:self.lastRefresh
         dateStyle:NSDateFormatterNoStyle
         timeStyle:NSDateFormatterShortStyle];

    self.lastRefreshLabel.text = [NSString
         stringWithFormat:@"Last refresh: %@", dateString];
}

- (void)setObservedAsConditions:(WeatherConditions *)conditions {
    // izabrani vremenski uslovi se postavljaju kao trenutno prikazani
    self.observedConditions = conditions;
    // izabrani vremenski uslovi se prikazuju na glavnom ekranu aplikacije
    [self setLabels];
}

- (void)setLabels {
    // podesavanje svih labela za odabrani prikaz vremenskih uslova
    self.tempLabel.text = [NSString stringWithFormat:@"%g ℃",
         [self.observedConditions.temperature doubleValue]];
    self.precipLabel.text = [NSString stringWithFormat:@"%@%%",
         self.observedConditions.precipitation];
    self.humidityLabel.text = self.observedConditions.humidity;
    self.windLabel.text = [NSString stringWithFormat:@"%g km/h",
         [self.observedConditions.wind doubleValue]];
    self.weatherLabel.text = self.observedConditions.weather;
    self.timeLabel.text = self.observedConditions.time;
    self.weatherImage.image = [UIImage imageNamed:self.observedConditions.icon];
```

```
    self.navigationItem.title = self.city;
}

- (void)selectDay:(int)day {
    // odabir specificnog dana za prikaz vremenskih uslova
    [self setSliderStepsForDay:day];
    [self removeOldHighlight];
    self.daySelected = day;
    [self highlightDay:day];
    self.slider.value = 0;
}

- (void)setSliderStepsForDay:(int)day {
    // podesavanje kontrole klizaca za odabrani dan
    int firstHour;
    if (day == 0) {
        firstHour = self.hour;
    } else {
        firstHour = 0;
    }

    self.slider.maximumValue = 24-firstHour-1;

    [self.sliderHours removeAllObjects];
    for (int i = firstHour; i<24; i++) {
        [self.sliderHours addObject:@(i)];
    }
}

- (void)removeOldHighlight {
    DayForecastView *oldSelected = [self.view viewWithTag:(100+self.daySelected)];
    oldSelected.backgroundColor = [UIColor clearColor];
    oldSelected.layer.borderWidth = 0;
}

- (void)highlightDay:(int)day {
    // oznacavanje odabranog dana
    DayForecastView *newSelected = [self.view viewWithTag:(100+day)];
    newSelected.backgroundColor = [UIColor
         colorWithRed:0.29 green:0.43 blue:0.85 alpha:1.0];
    newSelected.layer.borderWidth = 2;
    newSelected.layer.borderColor = [UIColor whiteColor].CGColor;
}

- (NSString *)getIconName:(NSDictionary *)dict {
    // podesavanje ikonice
    NSString *iconUrl = dict[@"icon_url"];
    NSString *fileName = [[iconUrl componentsSeparatedByString:@"/"] lastObject];
    NSString *icon = [fileName stringByDeletingPathExtension];

    return icon;
}
```

Slika 5.25 Pomoćne metode za pravljenje i prikazivanje vremenskih prognoza

U metodi viewDidLoad, koja se poziva nakon učitavanja pogleda kontrolera, koristi se mehanizam Cilj-radnja za definisanje metode koja se poziva prilikom promene vrednosti klizača (slika 5.26). U toj metodi se određuje čas za poziciju klizača i podešavaju se posmatrani vremenski uslovi za odabrani dan i odabrani čas.

```
(void)viewDidLoad {
    [super viewDidLoad];
```

```
    [self.slider addTarget:self
                    action:@selector(sliderValueChanged:)
          forControlEvents:UIControlEventValueChanged];
}

- (void)sliderValueChanged:(UISlider *)sender {
    // zaokruzivanje pozicije klizaca na celobrojni indeks
    NSUInteger index = (NSUInteger)(self.slider.value + 0.5);

    // simulacija klizaca koji ima stepene umesto kontinualnog klizaca
    [self.slider setValue:index animated:NO];

    if (self.lastIndex != index) {
        self.lastIndex = index;

        // podesavanje posmatranih vremenskih uslova za odabrani dan i sat
        [self setObservedAsConditions:self.hourlyForecasts[self.daySelected][number]];

        // omogucavanje dugmeta za povratak na trenutne vremenske uslove
        self.currentWeatherButton.hidden = NO;
    }
}
```

Slika 5.26 Reagovanje na promenu pozicije klizaca

U metodi `viewWillAppear` postavlja se posmatrač koji metodom `appActivated` reaguje prilikom dobijanja obaveštenja o prelasku aplikacije u aktivno stanje. Ovaj posmatrač se uklanja u metodi `viewWillDisappear`. U metodi `appActivated` automatski se osvežavaju podaci sa servisa ukoliko nisu poslednji put učitani u nekom trenutku u prethodnih 60 sekundi (slika 5.27).

```
- (void)viewWillAppear:(BOOL)animated {
    [super viewWillAppear:animated];

    // postavljanje posmatraca koji reaguje prilikom dobijanja obavestenja o prelasku
    // aplikacije u aktivno stanje
    [[NSNotificationCenter defaultCenter]
         addObserver: self
         selector: @selector(appActivated:)
         name: UIApplicationDidBecomeActiveNotification
         object: nil];
}

- (void)viewWillDisappear:(BOOL)animated {
    [super viewWillDisappear:animated];
    [[NSNotificationCenter defaultCenter] removeObserver:self];
}

- (void)appActivated:(NSNotification *)notif {
    NSTimeInterval secondsBetween = -[self.lastRefresh timeIntervalSinceNow];

    // osvezavanje podataka sa servisa ako je proslo vise od 60 sekundi od poslednjeg
    // osvezavanja
    if (secondsBetween > 60) {
        [self refreshWeather];
    }
}

- (void)refreshWeather {
    // osvezavanje podataka o vremenskim uslovima
    if (self.locationIndex == 0) {
        if (self.locationEnabled) {
            [self startLocation];
        }
```

```
    } else {
        [self createLocationFromDefaults];
        [self changeWeather];
        if (self.isNetworkError) {
            [self showAlert];
        }
    }
}

- (void)startLocation {
    // inicijalizacija upravljaca trenutne lokacije
    if (self.locationManager == nil) {
        self.locationManager = [[CLLocationManager alloc] init];
        self.locationManager.delegate = self;
        self.locationManager.desiredAccuracy = kCLLocationAccuracyKilometer;
    } else {
        self.didFindLocation = NO;
        [self.locationManager startUpdatingLocation];
    }
}

-(void)createLocationFromDefaults {
    // pravljenje objekta lokacije od poslednje odabrane lokacije
    NSDictionary *selectedLocation = [[NSUserDefaults standardUserDefaults]
         arrayForKey:@"savedLocations"][self.locationIndex - 1];
    double lat = [selectedLocation[@"lat"] doubleValue];
    double lon = [selectedLocation[@"lon"] doubleValue];
    self.location = [[CLLocation alloc] initWithLatitude:lat longitude:lon];
}

- (void)showAlert {
    UIAlertController* alert = [UIAlertController
         alertControllerWithTitle:@"Network problem"
         message:@"There was a problem with your network connectivity. Try enabling
cellular data or WiFi."
         preferredStyle:UIAlertControllerStyleAlert];

    UIAlertAction* defaultAction = [UIAlertAction
         actionWithTitle:@"OK"
         style:UIAlertActionStyleDefault
         handler:^(UIAlertAction * action) {}];

    [alert addAction:defaultAction];
    if (self.controllerPresented) {
        [self presentViewController:alert animated:YES completion:nil];
    } else {
        LoadingViewController *loadVC = (LoadingViewController *) [[[[UIApplication
                  sharedApplication] delegate] window] rootViewController];
        [loadVC showAlert];
    }
}
```

Slika 5.27 Automatsko osvežavanje podataka sa servisa

Na slici 5.28 prikazane su metode koje reaguju na događaj pritiska određenog dugmeta u korisničkom interfejsu glavnog ekrana aplikacije. U slučaju pritiska na dugme za ručno osvežavanje podataka poziva se metoda `refreshClicked`. Kada korisnik odabere jedan od osam dana u donjoj polovini glavnog ekrana aplikacije, poziva se metoda `dayTapped`. Pritiskom dugmeta „*Show current weather*", poziva se metoda `currentPressed`.

```
- (IBAction)refreshClicked:(id)sender {
    [self refreshWeather];
```

```
}

- (IBAction)dayTapped:(UITapGestureRecognizer*)sender {
    // odabran dan za prikaz podataka
    UIView *view = sender.view;
    NSInteger tag = view.tag;
    int daySelected = (int)tag - 100;

    if (daySelected != self.daySelected) {
        // prikazuju se podaci o izabranom danu
        [self setObservedAsConditions:self.hourlyForecasts[daySelected][0]];
        [self selectDay:daySelected];
        self.currentWeatherButton.hidden = NO;
    }
}
- (IBAction)currentPressed:(id)sender {
    // odabran prikaz trenutnih vremenskih uslova
    [self showCurrentWeather];
}
```

Slika 5.28 Metode koje reaguju na događaje sa korisničkog interfejsa

Metodom sa slike 5.29 moguće je izvršiti određena podešavanja nad odredištem prelaza između dva kontrolera pre nego što se prelaz zapravo dogodi.

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender {
    if ([[segue identifier] isEqualToString:@"MainWeatherToSavedLocationsSegue"]) {
        // odrediste prelaza izmedju dva kontrolera
        SavedLocationsTableViewController *destVC = [segue destinationViewController];
        destVC.delegate = self;
        destVC.locationIndex = self.locationIndex;
        destVC.locationEnabled = self.locationEnabled;
    }
}
```

Slika 5.29 Podešavanje odredišta prelaza između dva kontrolera

Kao što je već napomenuto, klasa `WeatherViewController` implementira protokol `SavedLocationsDelegate`. Prilikom povratka iz ekrana liste sačuvanih lokacija na glavni ekran aplikacije, kontroler `SavedLocationsTableViewController` poziva metodu `wentBackFromSavedLocations` nad svojim delegatom. Ova metoda implementirana je u klasi `WeatherViewController` i prikazana je na slici 5.30.

```
- (void)wentBackFromSavedLocations {
    self.locationIndex = [[[NSUserDefaults standardUserDefaults]
         objectForKey:@"selectedLocationIndex"] intValue];

    if (self.locationIndex == 0) {

         [self startLocation];
    } else {

        [self createLocationFromDefaults];
        [self changeWeather];

        if (self.isNetworkError) {
            [self showAlert];
        }
    }
}
```

Slika 5.30 Implementacija metode SavedLocationsDelegate protokola

5.4.3 Klasa SavedLocationsTableViewController

Klasa `SavedLocationsTableViewController` upravlja ekranom liste sačuvanih lokacija. Ova klasa je potklasa klase `UITableViewController` koja je specifično namenjena za upravljanje pogledom tabele (`UITableView`) koja ispunjava ceo pogled ovog kontrolera. Klasa `UITableViewController` implementira protokole `UITableViewDelegate` i `UITableViewDataSource` za upravljanje sadržajem pogleda tabele iz kontrolera.

U ekstenziji klase `SavedLocationsTableViewController` nalazi se svojstvo `locations` koje predstavlja promenljivi niz lokacija u tabeli (slika 5.31).

```
@interface SavedLocationsTableViewController ()<GMSAutocompleteViewControllerDelegate>
@property (nonatomic, strong) NSMutableArray *locations;
@end
```

Slika 5.31 Ekstenzija klase SavedLocationsTableViewController

Ova klasa čuva instancu svog delegata koji implementira protokol `SavedLocationsDelegate`. Na slici 5.32 prikazana je deklaracija protokola i interfejs klase `SavedLocationsTableViewController` u datoteci interfejsa klase.

```
@protocol SavedLocationsDelegate <NSObject>
- (void)wentBackFromSavedLocations;
@end

@interface SavedLocationsTableViewController : UITableViewController
@property (nonatomic, weak) id<SavedLocationsDelegate> delegate;
@property int locationIndex;
@property BOOL locationEnabled;
@end
```

Slika 5.32 Interfejs klase SavedLocationsTableViewController i protokol SavedLocationsDelegate

Prilikom povratka iz ekrana liste sačuvanih lokacija u drugi kontroler pogleda, šalje se poruka `wentBackFromSavedLocations` svom delegatu (slika 5.33).

```
- (void)willMoveToParentViewController:(UIViewController *)parent {
    if (![parent isEqual:self.parentViewController]) {
        [self.delegate wentBackFromSavedLocations];
    }
}
```

Slika 5.33 Povratak u drugi kontroler pogleda

U metodi `viewDidLoad` vrši se učitavanje liste sačuvanih lokacija kao i podešavanje pogleda (slika 5.34).

```
- (void)viewDidLoad {
    [super viewDidLoad];

    // ucitavanje prethodno sacuvanih lokacija
    self.locations = [[[NSUserDefaults standardUserDefaults]
         arrayForKey:@"savedLocations"] mutableCopy];

    // podesavanje povratka na prethodni ekran
    if ((self.locations != nil && [self.locations count] > 0) || self.locationEnabled) {
```

```
        self.navigationController.navigationBar.topItem.backBarButtonItem =
            [[UIBarButtonItem alloc]
                initWithTitle:@"Weather"
                style:UIBarButtonItemStylePlain
                target:nil
                action:nil];
    } else {

        self.navigationController.navigationBar.topItem.hidesBackButton = YES;
        self.navigationItem.hidesBackButton = YES;

    }

    self.tableView.editing = YES;
    self.tableView.allowsSelectionDuringEditing = YES;
}
```

Slika 5.34 Učitavanje liste lokacija i podešavanje pogleda

Na slici 5.35 prikazano je reagovanje na prelazak u aktivno stanje aplikacije kada je aktivan ekran liste sačuvanih lokacija. U slučaju isključivanja GPS funkcionalnosti uređaja, prva stavka liste lokacija neće biti trenutna lokacija i neće biti moguće izabrati trenutnu lokaciju dok god se GPS funkcionalnost ne omogući ponovo.

```
- (void)viewWillAppear:(BOOL)animated {
    [super viewWillAppear:animated];

    // postavljanje posmatraca koji reaguje prilikom dobijanja obavestenja o prelasku
    // aplikacije u aktivno stanje
    [[NSNotificationCenter defaultCenter]
         addObserver: self
         selector: @selector(appActivated:)
         name: UIApplicationDidBecomeActiveNotification
         object: nil];
}

- (void)viewWillDisappear:(BOOL)animated {
    [super viewWillDisappear:animated];
    [[NSNotificationCenter defaultCenter] removeObserver:self];
}

- (void)appActivated:(NSNotification *)notif {
    [self checkLocation];
    dispatch_async(dispatch_get_main_queue(), ^{
        [self.tableView reloadData];
    });}

- (void)checkLocation {
    // provera autorizacionog statusa potrazivanja trenutne lokacije
    CLAuthorizationStatus status = [CLLocationManager authorizationStatus];
    if (status == kCLAuthorizationStatusRestricted ||
        status == kCLAuthorizationStatusDenied ||
        ![CLLocationManager locationServicesEnabled]) {

         self.locationEnabled = NO;

    else if (status == kCLAuthorizationStatusAuthorizedWhenInUse &&
             [CLLocationManager locationServicesEnabled]) {

         self.locationEnabled = YES;
    }
}
```

Slika 5.35 Reagovanje na prelazak u aktivno stanje aplikacije na ekranu liste sačuvanih lokacija

Na slici 5.36 prikazane su implementacije određenih metoda oba protokola za upravljanje sadržajem pogleda tabele.

```
// broj sekcija u tabeli
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    return 1;
}

// broj redova u sekcijama
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section {

    if (self.locationEnabled) {
        return [self.locations count] + 1;
    } else {
        return [self.locations count];
    }
}

// izgled celije u tabeli na poziciji enkapsuliranoj u objektu NSIndexPath
- (UITableViewCell *)tableView:(UITableView *)tableView
         cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    // identifikator za ponovnu upotrebljivost instance celije sa drugim sadrzajem
    static NSString *cellIdentifier = @"locationCell";

    // dobijanje vec napravljene celije za prikaz sa izmenjenim sadrzajem
    UITableViewCell *cell = [tableView
         dequeueReusableCellWithIdentifier:cellIdentifier
         forIndexPath:indexPath];

    // pravljenje nove celije ukoliko nije mogla biti dobijena vec napravljena celija
    if (cell == nil) {
        cell = [[UITableViewCell alloc]
                  initWithStyle:UITableViewCellStyleDefault
                  reuseIdentifier:cellIdentifier];
    }

    // podesavanje celije
    if (self.locationEnabled) {
        if (indexPath.row == 0) {
            // postavljanje prve celije u tabeli za trenutnu lokaciju
            cell.textLabel.text = @"Current Location";
            cell.textLabel.font = [UIFont boldSystemFontOfSize:17.0f];
        } else {
            cell.textLabel.text = self.locations[indexPath.row - 1][@"name"];
            cell.textLabel.font = [UIFont systemFontOfSize:17.0f];
        }

        // postavljanje kukice na celiji poslednje izabrane lokacije
        if (indexPath.row == self.locationIndex) {
            cell.accessoryType = UITableViewCellAccessoryCheckmark;
            cell.editingAccessoryType = UITableViewCellAccessoryCheckmark;
        } else {
            cell.accessoryType = UITableViewCellAccessoryNone;
            cell.editingAccessoryType = UITableViewCellAccessoryNone;
        }
    } else {
        // slucaj iskljucene GPS funkcionalnosti
        cell.textLabel.text = self.locations[indexPath.row][@"name"];
        cell.textLabel.font = [UIFont systemFontOfSize:17.0f];

        if (indexPath.row == (self.locationIndex - 1)) {
            cell.accessoryType = UITableViewCellAccessoryCheckmark;
            cell.editingAccessoryType = UITableViewCellAccessoryCheckmark;
        } else {
```

```
            cell.accessoryType = UITableViewCellAccessoryNone;
            cell.editingAccessoryType = UITableViewCellAccessoryNone;
        }
    }

    return cell;
}

// reagovanje na odabir stavke u tabeli
- (void)tableView:(UITableView *)tableView
         didSelectRowAtIndexPath:(NSIndexPath *)indexPath {

    int index = (int) indexPath.row;

    if (self.locationEnabled) {
        [self selectLocationForIndex:index];
    } else {
        [self selectLocationForIndex:(index + 1)];
    }
}

// cuvanje izabrane lokacije i vracanje na prethodni kontroler
- (void)selectLocationForIndex:(int)index {
    self.locationIndex = index;
    [self persistLocationIndex];

    [self.navigationController popViewControllerAnimated:YES];
}

// odredjivanje da li je dozvoljeno vrsiti izmene odredjene stavke u tabeli
- (BOOL)tableView:(UITableView *)tableView canEditRowAtIndexPath:(NSIndexPath *)indexPath {

    if ([tableView numberOfRowsInSection:0] <= 1) {
        // zabranjeno brisanje stavke ukoliko je jedina na listi
        return NO;
    } else {
        if (self.locationEnabled) {
            if (indexPath.row != 0) {
                return YES;
            } else {
                // zabranjeno brisanje stavke trenutne lokacije
                return NO;
            }
        } else {
            return YES;
        }
    }
}

// podrska za izmene u tabeli
- (void)tableView:(UITableView *)tableView
         commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
         forRowAtIndexPath:(NSIndexPath *)indexPath {

    // operacija brisanja celije i azuriranje indeksa izabrane lokacije u listi
    if (editingStyle == UITableViewCellEditingStyleDelete) {

        if (self.locationEnabled) {
            if (indexPath.row <= self.locationIndex) {
                self.locationIndex--;
                [self persistLocationIndex];
            }
        } else {
            if ((indexPath.row + 1) <= self.locationIndex) {
                if (self.locationIndex != 1) {
                    self.locationIndex--;
```

```
                }
                [self persistLocationIndex];
            }
        }

        if (self.locationEnabled) {
            [self.locations removeObjectAtIndex:(indexPath.row - 1)];
        } else {
            [self.locations removeObjectAtIndex:indexPath.row];
        }
        // cuvanje nove liste sacuvanih lokacija u sistemske podrazumevane vrednosti
        [self persistSavedLocations];

        [tableView deleteRowsAtIndexPaths:@[indexPath]
                   withRowAnimation:UITableViewRowAnimationFade];
        [self.tableView reloadData];
    }
}

// odredjivanje da li je dozvoljeno vrsiti premestanje odredjene stavke u tabeli
- (BOOL)tableView:(UITableView *)tableView canMoveRowAtIndexPath:(NSIndexPath *)indexPath {

    if (self.locationEnabled) {
        if (indexPath.row == 0) {
           // zabranjeno premestanje trenutne lokacije (uvek je prva na listi)
            return NO;
        } else {
            return YES;
        }
    } else {
        return YES;
    }
}

// podrska za premestanje u tabeli
- (void)tableView:(UITableView *)tableView
         moveRowAtIndexPath:(NSIndexPath *)fromIndexPath
         toIndexPath:(NSIndexPath *)toIndexPath {

    // operacija premestanja u tabeli i azuriranje indeksa poslednje odabrane geografkse
    // lokacije
    if (self.locationEnabled) {
        NSDictionary *location = self.locations[fromIndexPath.row - 1];
        [self.locations removeObjectAtIndex:fromIndexPath.row - 1];
        [self.locations insertObject:location atIndex:toIndexPath.row - 1];
        [self persistSavedLocations];

        if (fromIndexPath.row == self.locationIndex) {
            self.locationIndex = (int)toIndexPath.row;
            [self persistLocationIndex];
        } else {
            if (toIndexPath.row <= self.locationIndex) {
                self.locationIndex++;
                [self persistLocationIndex];
            }
        }

    } else {
        NSDictionary *location = self.locations[fromIndexPath.row];
        [self.locations removeObjectAtIndex:fromIndexPath.row];
        [self.locations insertObject:location atIndex:toIndexPath.row];
        // cuvanje nove liste sacuvanih lokacija u sistemske podrazumevane vrednosti
        [self persistSavedLocations];

        if ((fromIndexPath.row +1) == self.locationIndex) {
            self.locationIndex = ((int)toIndexPath.row) + 1;
```

```
            [self persistLocationIndex];
        } else {
            if ((toIndexPath.row + 1) <= self.locationIndex) {
                self.locationIndex++;
                [self persistLocationIndex];
            }
        }
    }
}

// upravljanje premestanjem celija
- (NSIndexPath *)tableView:(UITableView *)tableView
         targetIndexPathForMoveFromRowAtIndexPath:(NSIndexPath *)sourceIndexPath
         toProposedIndexPath:(NSIndexPath *)proposedDestinationIndexPath {

    if (self.locationEnabled) {
       // onemogucavanje premestanja celije na prvo mesto (ispred celije trenutne lokacije)
       if (proposedDestinationIndexPath.row == 0) {
            return [NSIndexPath indexPathForRow:1 inSection:0];
       } else {
            return proposedDestinationIndexPath;
       }
    } else {
        return proposedDestinationIndexPath;
    }
}

// pomocna metoda cuvanja liste lokacija u sistemske podrazumevane vrednosti
- (void)persistSavedLocations {
    [[NSUserDefaults standardUserDefaults]
         setObject:self.locations forKey:@"savedLocations"];
    [[NSUserDefaults standardUserDefaults] synchronize];
}

// pomocna metoda cuvanja indeksa poslednje odabrane geografske lokacije u sistemske
   podrazumevane vrednosti
- (void)persistLocationIndex {
    [[NSUserDefaults standardUserDefaults]
         setInteger:self.locationIndex forKey:@"selectedLocationIndex"];
    [[NSUserDefaults standardUserDefaults] synchronize];
}
```

Slika 5.36 Implementacije metoda protokola za upravljanje sadržajem pogleda tabele

Pritiskom na desno dugme u navigacionom baru ekrana liste sačuvanih lokacija, poziva se metoda `addCityButtonClicked` koja prikazuje ekran za pretraživanje gradova (slika 5.37). Ovaj ekran definisan je kao pogled klase `GMSAutocompleteViewController`. Klasa `GMSAutocompleteViewController` je deo biblioteke *„Google Places API for iOS"*. Ova biblioteka je dodata u projekat korišćenjem rukovodioca zavisnosti (engl. *dependency manager*) za Objective-C projekte pod nazivom ***CocoaPods***.

```
- (IBAction)addCityButtonClicked:(id)sender {
    GMSAutocompleteViewController *acController =
         [[GMSAutocompleteViewController alloc] init];
    acController.delegate = self;

    // podesavanje filtera za prikaz samo gradova u padajucoj listi ponudjenih rezultata
    GMSAutocompleteFilter *autoFilter = [[GMSAutocompleteFilter alloc] init];
    autoFilter.type = kGMSPlacesAutocompleteTypeFilterCity;
    acController.autocompleteFilter = autoFilter;
```

```
    [self presentViewController:acController animated:YES completion:nil];
}
```

Slika 5.37 Incijalizacija i prikazavanje instance GMSAutocompleteViewController klase

Klasa `GMSAutocompleteViewController` prikazuje pogled sa tekstualnim poljem za pretragu i padajućom listom ponuđenih rezultata koji dopunjuju pojam pretrage na osnovu zadatih filtera. Instanca ove klase pruža informacije o izabranom rezultatu pretrage slanjem poruka svom delegatu.

Instanca `SavedLocationsTableViewController` klase definisana je kao delegat instance `GMSAutocompleteViewController` klase u metodi sa prethodne slike 5.37. Klasa `SavedLocationsTableViewController` implementira protokol `GMSAutocompleteViewControllerDelegate` i na slici 5.38 prikazane su implementacije obaveznih metoda ovog protokola.

```
// metoda koja se poziva prilikom izbora lokacije iz padajuce liste pretrage
- (void)viewController:(GMSAutocompleteViewController *)viewController
         didAutocompleteWithPlace:(GMSPlace *)place {
    for (NSDictionary *location in self.locations) {
        // povratak u slucaju odabira lokacije koja je vec na listi sacuvanih
        if ([location[@"name"] isEqualToString:place.formattedAddress]) {
            [self dismissViewControllerAnimated:YES completion:nil];
            return;
        }
    }

    // izdvajanje koordinata i naziva izabrane lokacije
    NSNumber *lat = [NSNumber numberWithDouble:place.coordinate.latitude];
    NSNumber *lon = [NSNumber numberWithDouble:place.coordinate.longitude];
    NSDictionary *location = @{
                               @"name" : place.formattedAddress,
                               @"lat" : lat,
                               @"lon" : lon
                               };

    // dodavanje lokacije na listu sacuvanih
    if (self.locations == nil) {
        self.locations = [[NSMutableArray alloc] initWithArray:@[location]];
    } else {
        [self.locations addObject:location];
    }
    [self persistSavedLocations];

    // azuriranje tabele liste sacuvanih lokacija sa novim podacima
    [self.tableView reloadData];

    // odabir nove lokacije
    [self dismissViewControllerAnimated:YES completion:^{

        int index = (int) [self.locations count];
        [self selectLocationForIndex:index];
    }];
}

// metoda koja se poziva prilikom greske u pretrazi
- (void)viewController:(GMSAutocompleteViewController *)viewController
         didFailAutocompleteWithError:(NSError *)error {

    [self dismissViewControllerAnimated:YES completion:nil];
    NSLog(@"Error: %@", [error description]);
```

```
}

// metoda koja se poziva pritiskom na dugme Cancel na ekranu dodavanja nove lokacije
- (void)wasCancelled:(GMSAutocompleteViewController *)viewController {
    [self dismissViewControllerAnimated:YES completion:nil];
}

// prikazivanje indikatora o mreznoj aktivnosti
- (void)didRequestAutocompletePredictions:(GMSAutocompleteViewController *)viewController {
    [UIApplication sharedApplication].networkActivityIndicatorVisible = YES;
}

// sakrivanje indikatora o mreznoj aktivnosti
- (void)didUpdateAutocompletePredictions:(GMSAutocompleteViewController *)viewController {
    [UIApplication sharedApplication].networkActivityIndicatorVisible = NO;
}
```

Slika 5.38 Implementacije metoda GMSAutocompleteViewControllerDelegate protokola

6 Zaključak

Prisustvo konkurencije na tržištu pametnih mobilnih telefona, koje sve više ljudi koristi, dovodi do potrebe za neprekidnim usavršavanjem ovih uređaja. Kompanije koje proizvode mobilne telefone teže ka poboljšanju uređaja unapređivanjem performansi i dodavanjem novih funkcionalnosti ovim uređajima. Kao posledica unapređenja uređaja, često dolazi do izmena u procesu programiranja iOS aplikacija kao i radnih okvira za razvoj. Mobilni telefoni postaju „pametniji" i prilagođavaju se komercijalizaciji novijih tehnologija poput augmentovane realnosti, sistema *Internet of Things* (IoT), češćoj upotrebi veštačke inteligencije i slično.

Glavni doprinos ovog rada je bliže upoznavanje čitalaca sa konceptima i principima programiranja aplikacija za platformu iOS u programskom jeziku Objective-C što je u radu ostvareno kroz prikaz pregleda operativnog sistema iOS, programskog jezika Objective-C i radnih okvira za razvoj aplikacija.

Iako napravljen pre više od 30 godina, programski jezik Objective-C i dalje predstavlja osnovni jezik za razvoj iOS aplikacija. Programski jezik Swift razvijen je sa ciljem da preuzme ulogu primarnog jezika za razvoj iOS aplikacija, međutim Objective-C je i dalje u velikoj meri zastupljen u novim iOS projektima zbog velikog broja postojećih eksternih biblioteka razvijenih za Objective-C projekte. Učenje jezika Objective-C bi svakako trebalo da predstavlja ulaznu tačku u svet programiranja za iOS budućim Swift programerima jer nudi bolje upoznavanje sa samim sistemom i konceptima razvoja aplikacija.

U radu je takođe predstavljena implementacija aplikacije „WeatherMaster". Cilj razvijanja ove aplikacije bio je prikaz primene velikog broja koncepata i radnih okvira za programiranje mobilnih aplikacija za platformu iOS o kojima je bilo reči u prethodnim poglavljima. Motivacija za razvoj aplikacije, koja prikazuje podatke o vremenskoj prognozi, bilo je nepostojanje iOS aplikacije koja na adekvatan način prikazuje podatke o vremenskoj prognozi za specifično odabrani čas u odabranom danu. Može se zaključiti da je cilj razvijanja aplikacije ispunjen, ali svakako postoji prostor za njeno dalje unapređenje.

Literatura

[1] „Statistika prodaje pametnih mobilnih telefona":

http://www.gartner.com/newsroom/id/3609817

[2] „Operativni sistem iOS":

https://en.wikipedia.org/wiki/IOS

[3] „O iOS tehnologijama":

https://developer.apple.com/library/content/documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/

[4] „Sloj Cocoa Touch":

https://developer.apple.com/library/content/documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/iPhoneOSTechnologies/iPhoneOSTechnologies.html#//apple_ref/doc/uid/TP40007898-CH3-SW1

[5] „Sloj Media":

https://developer.apple.com/library/content/documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/MediaLayer/MediaLayer.html

[6] „Sloj Core Services":

https://developer.apple.com/library/content/documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/CoreServicesLayer/CoreServicesLayer.html

[7] „Sloj Core OS":

https://developer.apple.com/library/content/documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/CoreOSLayer/CoreOSLayer.html

[8] „Istorijat verzija sistema iOS":

https://en.wikipedia.org/wiki/IOS_version_history

[9] „Programski jezik Objective-C":

https://en.wikipedia.org/wiki/Objective-C

[10] C. Keur, A. Hillegass and J. Conway, iOS Programming: The Big Nerd Ranch Guide (4th Edition), Atlanta: Big Nerd Ranch, 2014

[11] „Dinamičko uvezivanje i tipiziranje u jeziku Objective-C sa id tipom podataka":

http://www.techotopia.com/index.php/Objective-C_Dynamic_Binding_and_Typing_with_the_id_Type

[12] „Klase u jeziku Objective-C":

http://rypress.com/tutorials/objective-c/classes.html

[13] A. Hillegass and M. Ward, Objective-C Programming: The Big Nerd Ranch Guide (2nd Edition), Atlanta: Big Nerd Ranch, 2013

[14] „Protokoli u jeziku Objective-C":

http://rypress.com/tutorials/objective-c/protocols.html

[15] „Kategorije u jeziku Objective-C":

http://rypress.com/tutorials/objective-c/categories.html

[16] „Blokovi u jeziku Objective-C":

http://rypress.com/tutorials/objective-c/blocks.html

[17] „Radni okvir Foundation":

https://developer.apple.com/reference/foundation?language=objc

[18] „Životni ciklus aplikacije":

https://developer.apple.com/library/content/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/TheAppLifeCycle/TheAppLifeCycle.html

[19] „Upravljanje memorijom":

http://rypress.com/tutorials/objective-c/memory-management

[20] „Obrasci projektovanja iOS aplikacija":
https://www.raywenderlich.com/46988/ios-design-patterns

[21] „Obrasci projektovanja u Cocoa sloju":

https://developer.apple.com/legacy/library/documentation/Cocoa/Conceptual/CocoaFundamentals/CocoaDesignPatterns/CocoaDesignPatterns.html

[22] „Upoznavanje tehnologije Auto Layout”:

https://developer.apple.com/library/content/documentation/UserExperience/Conceptual/AutolayoutPG/index.html#//apple_ref/doc/uid/TP40010853-CH7-SW1

[23] „Komponenta Storyboard”:

https://developer.apple.com/library/content/documentation/General/Conceptual/Devpedia-CocoaApp/Storyboard.html