УНИВЕРЗИТЕТ У БЕОГРАДУ
МАТЕМАТИЧКИ ФАКУЛТЕТ

Марко Вељковић

КРЕИРАЊЕ ВИЦЕТА У ПРОГРАМСКОМ ЈЕЗИКУ SWIFT

мастер рад

Београд, 2022.

Мојој породици и пријатељима на подршци и
менторки на помоћи и саветима

Ментор:

др Милена Вујошевић Јаничић, ванредни професор
Универзитет у Београду, Математички факултет

Чланови комисије:

др Филип Марић, ванредни професор
Универзитет у Београду, Математички факултет

др Мирко Спасић, доцент
Универзитет у Београду, Математички факултет

Датум одбране: јул 2022.

Наслов мастер рада: Креирање виџета у програмском језику Swift

Резиме: Програмски језик *Swift* се користи приликом израде апликација намењених платформама компаније *Apple*. Виџет је део апликације који приказује одабране важне информације на почетном екрану уређаја. Да би се развио виџет потребна је употреба радног окружења *SwiftUI* које користи декларативну синтаксу. Циљ рада је представљање основних особина и карактеристика програмског језика *Swift* и приказ развоја виџета употребом радног окружења *SwiftUI*. Развијена је апликација "Кулинарство — сласно и ефикасно" која представља конкретан пример употребе програмског језика *Swift* приликом развоја апликације за оперативни систем *iOS*. Поред апликације приказано је и креирање виџета који је развијен у склопу ове апликације.

Кључне речи: програмски језици, развој софтвера, *Apple*, *iOS*, *Swift*, виџети

Садржај

Глава 1

Увод

Swift је модеран програмски језик опште намене, настао 2014. године у оквиру компаније *Apple*. Првенствено је намењен развоју апликација на њиховим платформама (*iOS*, *iPadOS*, *macOS*, *tvOS* и *watchOS*), али је део развоја језика усмерен ка поједностављивању процеса израде корисничког интерфејса и увођењу декларативне синтаксе у језик. У том контексту настало је радно окружење (енг. *framework*) *SwiftUI* које одликује могућност брзог креирања концизних и ефикасних решења. Пример примене радног окружења *SwiftUI* је израда видета намењених оперативним системима *iOS*, *iPadOS* и *macOS*.

Видет, као део мобилне апликације, се налази на почетном екрану уређаја (телефона или таблета) и кориснику приказује одабране важне информације из те апликације. За разлику од видета у оперативном систему *Android*, који су присутни више од десет година, видети на платформама компаније *Apple* су уведени 2020. године, тако да је и сама технологија која подржава њихово креирање и даље у активном развоју.

Циљ рада је истраживање и опис карактеристика и могућности које пружа програмски језик *Swift*, промене и побољшања које је донело радно окружење *SwiftUI* и приказ новина насталих увођењем видета (конкретно у оперативном систему *iOS*). У оквиру рада је имплементирана *iOS* апликација о кулинарству „Кулинарство — сласно и ефикасно" коришћењем програмског језика *Swift*, након чега је креиран одговарајући видет помоћу радног окружења *SwiftUI*. Овај видет пружа кориснику помоћ приликом избора и припреме оброка.

У глави 2 представљен је програмски језик *Swift*, описане су његове карактеристике, основни и напредни концепти, и особине које одликују овај

програмски језик. Описано је интегрисано развојно окружење (ИРО) *Xcode* у којем се развијају апликације намењене платформама компаније *Apple*, као и радно окружење *SwiftUI*. Глава 3 описује виџет као део мобилне апликације у оперативном систему *iOS*, развој виџета, као и правила и препоруке приликом његовог дизајнирања. У глави 4 представљена је имплементација и визуелни приказ апликације „Кулинарство — сласно и ефикасно", и виџета који је креиран уз апликацију. Последња, 5. глава представља закључак целокупног рада и приказује даља унапређења апликације.

Глава 2

Програмски језик *Swift*

Swift је модеран програмски језик настао као резултат истоименог пројекта унутар компаније *Apple*, чији је циљ био креирање програмског језика који ће бити сигуран, концизан и ефикасан. Резултати пројекта *Swift* као и унапређења програмског језика *Swift* током година биће приказани у наставку.

2.1 Настанак, развој и карактеристике

Развој програмског језика *Swift* започео је Крис Латнер[1] (енг. *Chris Lattner*) у јулу 2010. године. У јуну 2014. године објављена је прва апликација комплетно написана у *Swift*-у названа *„Apple Worldwide Developers Conference"* (*WWDC*) по истоименој годишњој конференцији информационих технологија компаније *Apple*. На конференцији те године, кроз предавање и интерактивну демонстрацију, представљена је бета верзија језика, бесплатна књига „The Swift Programming Language" [13] и званична веб страница програмског језика [14]. Прва званична верзија језика *Swift* 1.0, постала је доступна 9. септембра 2014. године.

Званично објављивање апликација (на *App Store*-у) писаних у *Swift*-у постало је могуће од верзије програмског језика 2.0. Језик је у оквиру анкете коју организује *Stack Overflow* [16] проглашен за омиљени програмски језик 2015. године, док је 2016. године заузео друго место у тој категорији. Децембра 2015. године изворни кôд језика, подржане библиотеке, дибагер и мена-

[1]Софтверски инжењер најпознатији по развоју технологија *LLVM*, компајлера *Clang* и програмског језика *Swift*.

џер пакета постали су отвореног кода, под лиценцом *Apache* 2.0, доступни на *GitHub*-у [10].

На годишњој конференцији *WWDC* 2016. године представљена је апликација за уређај *iPad* под називом *Swift Playgrounds* [11], која је намењена учењу програмирања у *Swift*-у. Касније је ова апликација развијена и за оперативни систем *macOS*.

Током конференције 2019. године представљено је радно окружење *SwiftUI* [12] које омогућава декларативно програмирање апликација за све платформе компаније *Apple*. У време писања рада последња званична верзија језика је *Swift* 5.6. Званичан лого програмског језика *Swift* приказан је на слици 2.1.

Слика 2.1: *Swift лого*

Swift је моћан и интуитиван језик за програмирање апликација намењених платформама компаније *Apple*. Писање кода у *Swift*-у је забавно и лако, синтакса је веома концизна, али у исто време веома изражајна. Програмски језик *Swift* је безбедан, брз и интерактиван, и као такав погодан за људе који уче основе програмирања. Кôд писан у *Swift*-у се преводи и оптимизује тако да извуче максимум из хардверских компоненти. Детаљнији опис особина и примери кодирања биће дати у наредним поглављима.

2.2 Модерне карактеристике, безбедност и ефикасност

Програмски језик *Swift* је креиран са намером да буде модеран, безбедан и ефикасан. Детаљан опис ових особина као и објашњење зашто је *Swift* одличан програмски језик за почетнике, када је пројекат *Swift* постао пројекат отвореног кода и компатибилност програмског језика *Swift* са *Objective-C*-ом је дат у наставку поглавља.

Модеран језик

Swift је настао као резултат најновијих истраживања програмских језика. Именовани параметри су изражајни (енг. *expressive*) што кôд позивања функција чини лаким за читање и разумевање (као читање реченице). Као и у свим модерним језицима употреба знака тачка-зарез на крају наредби није неопходна и по установљеној конвенцији се не пише. Претпостављање типова променљивих чини кôд чистијим и отпорнијим на грешке. Меморијом се управља аутоматски, коришћењем аутоматског бројача референци (енг. *Automatic Reference Counting, ARC*), који је детаљно објашњен у делу 2.4 - Аутоматски бројач референци.

Безбедан начин програмирања

У току развијања језика уложени су огромни напори да би он био што безбеднији. На пример, променљиве су увек иницијализоване пре употребе, низови и целобројне променљиве се увек проверавају да не би дошло до прекорачења, меморијом се управља аутоматски. Још једна безбедоносна одлика је да *Swift* објекти подразумевано никада не могу бити *nil*[2]. *Swift* компајлер ће спречити покушај да се направи или искористи *nil* објекат, избацивањем грешке у време превођења програма. Међутим, постоје случајеви када је коришћење вредности *nil* валидно. За те случајеве користе се опционе променљиве.

Ефикасно извршавање

Swift је наследник програмских језика *C* і *Objective-C* и као такав од почетка је дизајниран да буде концизан и ефикасан. Коришћењем технологије *LLVM* компајлера, *Swift* кôд се трансформише у оптимизовани извршни кôд који извлачи највише из модерног хардвера.

Друге важне особине програмског језика *Swift*

Swift је дизајниран да буде лак за савладавање, чак и људима који тек почињу да уче програмирање. У циљу подучавања *Apple* је направио бесплатан

[2]У програмском језику *Swift*, *nil* представља специјалну вредност која може бити додељена опционој променљивој и означава стање без вредности (одсуство вредности).

наставни план и програм који може свако користити [9]. Најбоља апликација за почетнике је *Swift Playgrounds* [11], која је у почетку била намењена уређајима *iPad*. Касније је иста апликација развијена и за употребу на уређајима са оперативним системом *macOS*.

Крајем 2015. године програмски језик *Swift* је под лиценцом *Apache* 2.0 постао пројекат отвореног кода. Заједно са пројектом језика, отвореног кода су постале и пратеће библиотеке, дибагер и менаџер пакета. Изворни кôд се налази на *GitHub*-у где је свакоме лако доступан за преузимање и евентуалну дораду и допуну. Пројекат *Swift* се састоји од неколико засебних пројеката:

- *Swift* компајлер,
- стандардна библиотека,
- *Core* библиотека,
- *LLDB* дибагер,
- *Swift* менаџер пакета,
- *Xcode* подршка за *Swift Playgrounds*.

Цела апликација може бити написана у *Swift*-у, или се *Swift* може користити за додавање нових функционалности у већ постојећи програм написан у *Objective-C*-у. *Swift* и *Objective-C* могу истовремено постојати у апликацији, и корисник без проблема може користити делове кода написаног у једном језику унутар другог и обратно, уз само мало додатног подешавања пројекта које се може пронаћи на *Apple*-овом сајту посвећеном програмерима (енг. *software developers*) [3], конкретно на адресама *Importing Objective-C into Swift* [5] и *Importing Swift into Objective-C* [6].

2.3 Основни концепти језика

Swift је наследник програмских језика *C* и *Objective-C*, па је самим тим одређене концепте преузео из ових језика, али истовремено постоје концепти у *Swift*-у који нису присутни у *C*-у и *Objective-C*-у. Објашњење најбитнијих концепата у *Swift*-у дато је у наставку, док се потпуна листа може наћи на званичном сајту програмског језика [14].

Основе програмског језика *Swift*

Swift подржава основне типове промењивих, *Int* за целобројне вредности, *Float* и *Double* за бројеве у покретном зарезу, *Bool* за Булове вредности, *String* за текстуалне вредности, као и три основна типа колекција *Array*, *Set* и *Dictionary* о којима ће бити више речи у делу 2.3 — Колекције.

Поред основних типова који су наслеђени из *Objective-C*-а постоје и неколико ново уведених, као што је *Tuples* (торка) који омогућава креирање груписаних вредности и *Optionals* помоћу којег се рукује *nil* вредношћу на безбедан начин.

Константе се декларишу коришћењем кључне речи *let*[3], након чега следи име константе и њена иницијализација. Декларисање промењиве се постиже употребом кључне речи *var*. Када се декларише промењива може се одмах и иницијализовати или јој може бити додељен тип употребом анотације. Конкретна примена се може видети у примеру 2.1 — *Декларисање променљивих и константи*.

```
1 // Deklarisanje konstante
2 let cenaJela = 200
3
4 // Deklarisanje promenljive uz inicijalizaciju
5 var raspolozivoNovca = 1000
6
7 // Deklarisanje promenljive koriscenjem anotacije
8 var brojPorcija: Int
```

Пример кода 2.1: *Декларисање променљивих и константи*

Целобројне промењиве могу бити написане у облику децималних, бинарних, октални и хексадецималних бројева. Пример дефинисања промењиве са вредношћу броја 25 у свим облицима следи у наставку 2.2 — *Целобројне промењиве*.

```
1 var mojiBroj = 25
2 var binarniBroj = 0b11001     // 25 u binarnom obliku
3 var oktalniBroj = 0o31        // 25 u oktalnom obliku
4 var heksadecimalniBroj = 0x19 // 25 u heksadecimalnom obliku
```

Пример кода 2.2: *Целобројне промењиве*

[3]Појам константе која следи након кључне речи *let* је мало шири и биће објашњен у делу 2.4 - Класе и структуре

Торка се користи за груписање вредности било ког типа и једна торка може садржати вредности различитих типова. У примеру 2.3 — *Торка* могу се видети два начина креирања торке и два начина приступања члановима торке (неименованим и именованим члановима).

```
1  // Definisanje torke (String, Int)
2  let namirnicaSaCenom = ("Jaja 10 komada", 100)
3
4  // Pristupanje clanovima torke
5  let namirnica = namirnicaSaCenom.0
6  let cena = namirnicaSaCenom.1
7
8  // Definisanje torke sa imenovanim clanovima
9  let urednijaNamirnicaSaCenom = (namirnica: "Jaja 10 komada", cena:
       100)
10
11 // Pristupanje imenovanim clanovima torke
12 let urednijaNamirnica = urednijaNamirnicaSaCenom.namirnica
13 let urednijaCena = urednijaNamirnicaSaCenom.cena
```

Пример кода 2.3: *Торка*

Провере испуњености услова коришћењем функције *assert* се дешавају у време извршавања кода. Најчешће се користи за проверу критичног дела кода и уколико тај део кода задовољава услов (вредност израза у *assert*-у је *true*) програм наставља своје извршавање. У супротном извршавање апликације ће бити прекинуто и биће означено место у коду у коме је дошло до прекида програма. Пример примене је приказан у коду 2.4 — *Провере коришћењем функције assert*.

```
1 var cenaPrvogProizvoda = 100
2 var cenaDrugogProizvoda = -100
3 assert(cenaPrvogProizvoda >= 0,
4        "Cena proizvoda ne moze biti negativna") //true
5 assert(cenaDrugogProizvoda >= 0,
6        "Cena proizvoda ne moze biti negativna") //false
```

Пример кода 2.4: *Провере коришћењем функције assert*

Оператори

Као и у већини програмских језика, постоје три основне врсте оператора:

- Унарни оператори
  - префиксни унарни оператори (на пример, '-' за бројевне вредности и '!' за логичке вредности),
  - постфиксни унарни оператори (на пример, '?' и '!' који се користе над опционим променљивима).
- Бинарни оператори
  - оператор доделе '=', који за разлику од језика C, не враћа повратну вредност,
  - аритметички оператори, '+', '-', '*', '/', '%',
  - сложени оператори доделе, '+=', '-=', '*=', '/=',
  - оператори поређења, '==', '!=', '<', '>', '<=', '>='.
- Тернарни оператори
  - једини тернарни оператор који постоји у језику *Swift* је оператор '?:'. Код овог оператора израз са крајње леве стране мора бити типа *Bool*, док израз који се налази у средини оператора мора бити истог типа као израз са крајње десне стране, без ограничења типа. Пример примене тернарног оператора као и приказ како једна линија кода са тернарним оператором може заменити блок кода условног гранања (које је објашњено у делу 2.3 — Контрола тока) приказани су у примеру 2.5 — *Тернарни оператор*.

```
1  var celijaImaSliku = true
2  // Izraz ternarnog operatora
3  let visinaCelije = celijaImaSliku ? 100 : 50
4
5  // Prethodni primer koriscenjem uslovnog grananja
6  var celijaImaSliku = true
7  var visinaCelije: Int
8  if celijaImaSliku {
9      visinaCelije = 100
10 }
11 else {
12     visinaCelije = 50
```

```
13 }
```

Пример кода 2.5: *Тернарни оператор*

Поред основних оператора у *Swift*-у, постоје и специјалне врсте оператора:

- Оператор 'nil-сједињавања' ('??') (енг. *nil coalescing*) је бинарни оператор који се користи над опционим променљивима (опционе променљиве су објашњене у делу 2.3 - Опционе променљиве и рад са њима). Уколико израз са леве стране оператора садржи вредност (није *nil*) та вредност ће бити резултат оператора, док уколико је лева страна оператора једнака *nil*, резултат оператора биће израз са десне стране који не сме бити опционог типа. Пример примене оператора '??' дат је у наставку 2.6 — *nil-сједињавање.*

```
1 var a: Int?
2 let pseudoNasumicanBroj = Int.random(in: 1...100)
3 if pseudoNasumicanBroj % 2 == 0 {
4     a = 10
5 }
6 let b = a ?? 5
```

Пример кода 2.6: *nil-сједињавање*

- Оператори распона, чија је повратна вредност типа генеричке структуре *Range*, структура која представља генерички (објашњено у делу 2.3 - Колекције) интервал вредности
  - затворен распон (a...b), распон од 'a' до 'b' укључујући обе вредности,
  - полу-отворен распон (a..<b), распон од 'a' до 'b' укључујући само вредност 'a',
  - распони једне стране [a...], распон од 'a' и надаље докле год је то могуће.

Карактери и стрингови

Стринг је низ карактера, као што је „Здраво, свете". У *Swift*-у се стрингови представљају помоћу класе *String*, која омогућава брз, ефикасан и *Unicode*-компатибилан начин рада са текстом. Да би се вредност неке променљиве

или израза уметнула у стринг користи се операција уметања, односно интерполације. Уметање промењиве или израза у стринг се постиже њиховим уписивањем између обрнуте косе црте након чега следи отворена заграда, па промењива или израз и на крају затворена заграда. Креирање и операције са стринговима (надовезивање, рад са карактерима и уметање) су приказане кроз пример 2.7 — *Операције над стринговима.*

```
1     var prazanString = ""
2     var drugiPrazanString = String()
3     // Definisanje promenljive opcionog tipa 'String'
4     var treciString: String?
5
6     if !prazanString.isEmpty {
7         prazanString = "Zdravo"
8     }
9
10    // Konkatenacija stringova
11    drugiPrazanString += ", svete"
12
13    // Rad sa karakterima
14    for k in prazanString {
15        print(k)
16    }
17    // Ispisace:
18    // Z
19    // d
20    // r
21    // a
22    // v
23    // o
24
25    // Interpolacija stringova
26    print("\(prazanString)\(drugiPrazanString)!")
27    // Ispisace 'Zdravo, svete!'
```

Пример кода 2.7: *Операције над стринговима*

Контрола тока

Наредбе контроле тока које се користе у *Swift*-у су: *if*, *guard*, *switch* и петље: *for-in* и *while*. *If* наредба приказана је у примеру 2.8 — *If наредба*

контроле тока.

```
1    var recept = Recept("Cezar salata")
2    recept.sastojci = ["Zelena salata", "Pilece grudi", "Slanina",
         "Paradajz", "Hleb", "Cezar premaz"]
3    var brojSastojaka = recept.sastojci.count
4
5    if brojSastojaka < 6 {
6        print("Nisu svi sastojci nabavljeni")
7    }
8    else if brojSastojaka > 6 {
9        print("Broj sastojaka je veci nego u receptu, ali samo napred
             eksperimentisi")
10   }
11   else {
12       print("Broj sastojaka je odgovarajuci")
13   }
```

Пример кода 2.8: *If наредба контроле тока*

Наредба *switch* је слична као у другим програмским језицима, једина битна разлика је да ће се увек извршити тачно један од случајева унутар наредбе, па није потребно експлицитно навођење наредбе *break* након сваког од случајева. Наредба *break* се по конвенцији наводи само када је неки од случајева наредбе *switch* празан, јер сваки случај мора бити извршив (енг. *executable*). Уколико случајевима наредбе *switch* нису обухваћени сви случајеви, на крају наредбе *switch* се мора навести наредба *default* која ће се извршити уколико ниједан од случајева није задовољио услов. Наведена правила приказана су у примеру 2.9 — *Наредба контроле тока switch.*

```
1    enum Zacin {
2        case vegeta, kari, kurkuma, origano, biber
3    }
4    var mojiZacin: Zacin = .vegeta
5
6    switch mojiZacin {
7        case .vegeta:
8            print("Vegeta")
9        case .biber:
10           print("Nije vegeta, nego biber")
11       default:
12           print("Nije vegeta ni biber")
13   }
```

Пример кода 2.9: *Наредба контроле тока switch*

For-in је наредба понављања која се користи за пролаз кроз елементе неке колекције (низа, скупа, речника), колекције су објашњене у делу 2.3 - Колекције. Променљива која се користи за пролаз кроз колекцију је константа и њену вредност није могуће мењати у телу наредбе. Један од начина како се могу мењати елементи колекције (уколико је колекција није константа) је истовременим пролажењем кроз елементе колекције и њихове индексе и променом елемента колекције на одговарајућем индексу. Ово се постиже употребом функције *enumerated()*, која се примењује на низ елемената, а повратна вредност ове функције је низ 2-торки, чији је први члан индекс елемента у низу, а други члан је сам елеменат низа. Још једна могућност *for-in* петље је пролаз кроз задати интервал бројева. Описани начини употребе *for-in* петље могу се видети у примеру 2.10 — *Наредба контроле тока for-in*.

```
1     let sastojci = ["Jaja", "Pecenica", "Maslinovo ulje", "Persun"]
2     // For-in naredba
3     for sastojak in sastojci {
4         print("Potreban sastojak: \(sastojak)")
5     }
6     // For-in naredba nad intervalom
7     for i in 0..<sastojci.count() {
8         print("Potreban sastojak: \(sastojci[i])")
9     }
10
11    let sastojciSaKolicinom = ["Jaja": "3 komada", "Pecenica":
12        "50 grama", "Maslinovo ulje": "Koliko je potrebno da
13        pokrije tiganj", "Persun": "Prstohvat"]
14    // For-in naredba za prolaz kroz recnik
15    for (sastojak, kolicina) in sastojciSaKolicinom {
16        print("Potreban sastojak: \(sastojak), u kolicini:
17              \(kolicina)")
18    }
19
20    var promenljiviElementi = ["Jaja", "Pecenica", "Maslinovo ulje",
          "Persun"]
21    // For-in naredba sa indeksiranjem
22    for (indeks, element) in promenljiviElementi.enumerated() {
23        if element == "Persun" {
24            promenljiviElementi[indeks] = "Origano"
```

```
25          }
26      }
```

Пример кода 2.10: *Наредба контроле тока for-in*

Постоје два типа наредбе понављања *while*. Први тип је наредба *while* која прво проверава да ли је задати услов испуњен и онда извршава једну итерацију тела петље, а други тип је наредба *repeat-while* која прво извршава једну итерацију тела наредбе након чега проверава услов и уколико је он задовољен наставља са следећом итерацијом. Употреба је приказана у примеру 2.11 — *Наредба контроле тока while*.

```
1   let nasumicniBrojevi = [3, 12, 5, 18, 11, 99]
2   var i = 0
3
4   while i < nasumicniBrojevi.count {
5       if nasumicniBrojevi[i] < 15 {
6           print("Broj \(nasumicniBrojevi[i]) je manji od 15")
7       }
8       else if nasumicniBrojevi[i] > 15 {
9           print("Broj \(nasumicniBrojevi[i]) je veci od 15")
10      }
11      else {
12          print("Broj \(nasumicniBrojevi[i]) je jednak 15")
13      }
14      i += 1
15  }
16
17  let nasumicniBroj = nasumicniBrojevi[2] // 5
18
19  repeat {
20      print("Zdravo, svete!")
21  } while nasumicniBroj != 5 // Uvek netacno
```

Пример кода 2.11: *Наредба контроле тока while*

Поред наведених основних наредби контроле тока, постоје додатне помоћне наредбе које се користе заједно са основним наредбама и тиме њихову употребу чине лакшом. Наредба *continue* се користи за прескакање једне итерације унутар петље. *Break* је наредба која прекида извршавање наредбе унутар које се налази, а може се користити унутар случајева наредбе *switch*

као и унутар тела петље. Пример употребе ових наредби приказан је у делу 2.12 — *Додаци наредбама контроле тока.*

```
1    let nizBrojeva = [1, 2, 3, 4, 5, 6, 7, 8, 9]
2    print("Parni brojevi iz niza manji od 7:")
3    for broj in nizBrojeva {
4        if broj % 2 != 0 {
5            continue
6        }
7        if broj > 7 {
8            break
9        }
10       print(broj)
11   }
```

Пример кода 2.12: *Додаци наредбама контроле тока*

Колекције

Swift дефинише три примарна типа колекција: низове, скупове и речнике. Сва три типа су дефинисана као генеричке[4] колекције. Уколико се дефинисана колекција додели промењливој, она се може мењати (додавање, брисање и измена чланова у њој); међутим уколико се она додели некој константи, манипулација њеним члановима неће бити могућа.

Низови се користе за уређено чување елемената истог типа. Један елемент се може појавити у низу више пута, са различитим индексима. Конкретан пример дефинисања, иницијализације и управљања подацима низа може се видети у коду 2.13 — *Рад са низовима.*

```
1    // Definisanje niza sa elementima tipa 'Recept'
2    var recepti: [Recept] = []
3    // Dodavanje novog elementa
4    recepti.append(Recept("Domaca kafa"))
5    // Pristupanje prvom clanu niza
6    var prviRecept = recepti[0]
7    // Kreiranje niza sa 3 inicijalna elementa tipa 'String'
8    var koraci = Array(repeating: "", count: 3)
9    // Foreach petlja kojom prolazimo kroz niz uz pamcenje indeksa
10   for (index, korak) in prviRecept.koraci.enumerated() {
```

[4]Генерички кôд омогућава писање флексибилних и поновно искористивих функција и типова, помоћу којих се избегава дуплирање кода.

```
11          if index < 3 {
12              koraci[index] = korak
13          }
14          else {
15              koraci.append(korak)
16          }
17      }
18      // Brisanje poslednjeg clana niza
19      koraci.popLast()
```

Пример кода 2.13: *Рад са низовима*

Скупови су колекције које не гарантују чување редоследа елемената и у којима један елемент може да се појави највише једанпут. За разлику од већине других програмских језика, елементи унутар *Swift* скупова нису уређени, већ се ефикасност провере постојања елемента гарантује употребом хеш вредности. Тип елемента скупа мора бити могуће кодирати[5] (енг. *hashable*). Рад са скуповима приказан је у примеру 2.14 — *Рад са скуповима*.

```
1       // Kreiranje skupa sa elementima tipa 'String'
2       var sastojci = Set<String>()
3       // Dodavanje novog elementa
4       sastojci.insert("Mlevena kafa")
5
6       // Provera broja elemenata skupa
7       if sastojci.count == 1 {
8           sastojci.insert("Obicna voda")
9       }
10
11      // Provera da li odredjeni element postoji u skupu
12      if sastojci.contains("Secer") {
13          sastojci.remove("Secer")
14      }
15      else {
16          sastojci.insert("Mleko")
17      }
18
19      var dodatniSastojci = Set<String>()
20      dodatniSastojci.insert("Mleko")
21
```

[5]Тип који се може кодирати мора имати дефинисану функцију за одређивање хеш вредности за сваку инстанцу: два елемента могу имати исту хеш вредност ако и само ако су једнаки.

```
22      // Rad sa skupovnim operacijama
23
24      // Unija
25      sastojci.union(dodatniSastojci)
26      // Mlevena kafa, Obicna voda, Mleko
27
28      // Presek
29      sastojci.intersection(dodatniSastojci)
30      // Mleko
31
32      // Razlika
33      sastojci.subtracting(dodatniSastojci)
34      // Mlevena kafa, Obicna voda
35
36      // Simetricna razlika
37      sastojci.symmetricDifference(dodatniSastojci)
38      // Mlevena kafa, Obicna voda
```

Пример кода 2.14: *Рад са скуповима*

Речници се користе за чување скупа парова кључ-вредност, без очувања редоследа. Свака вредност је додељена јединственом кључу, који мора бити погодан за кодирање (енг. *hashable*). Речници се најчешће користе за чување вредности којима је могуће брзо приступити коришћењем одговарајућег кључа. Рад са речницима приказан је у прмеру 2.15 — *Рад са речницима.*

```
1       // Kreiranje recnika tipa [String : String]
2       var kategorijeRecepata: [String : String] = [:]
3
4       // Dodeljivanje recnika promenljivoj 'namirniceSaCenama'
5       kategorijeRecepata = ["Cezar salata": "Salata", "Piletina sa
            cetiri vrste sira": "Glavno jelo", "Omlet": "Toplo predjelo"]
6
7       // Dodavanje novog elementa ukoliko ne postoji, odnosno promena
            postojeceg
8       kategorijeRecepata["Vocni kolac"] = "Dezert"
9
10      // Brisanje elementa iz recnika
11      if let izbrisanaVrednost = kategorijeRecepata.removeValue(forKey:
            "Omlet") {
12          print("Vrednost izbrisana iz recnika: \(izbrisanaVrednost)")
13      }
14
```

```
15      // Razlicite vrste iteracija kroz recnik
16      for recept in kategorijeRecepata.keys {
17          print(recept)
18      }
19
20      for kategorija in kategorijeRecepata.values {
21          print(kategorija)
22      }
23
24      for (recept, kategorija) in kategorijeRecepata {
25          print("\(recept): \(kategorija)")
26      }
```

Пример кода 2.15: *Рад са речницима*

Функције

Функције су самостални блокови кода који представљају логичку целину која остварује одређени задатак. Свака функција је идентификована својим именом које се користи да би се та функција позивала у коду. Поред имена, функција може имати тип повратне вредности (уколико није дефинисан, подразумевани тип је *Void*) и пареметре (именоване или неименоване). Општи потпис дефиниције функције је приказан у примеру: 2.16 — *Потпис функције*, док је потпис параметара функције приказан у примеру: 2.17 — *Потпис параметара функције*.

```
1 func ime_funkcije (parametri_funkcije) {-> tip_povratne_vrednosti}
```

Пример кода 2.16: *Потпис функције*

```
1 {labela_parametra} ime_parametra: tip_parametra {=
    podrazumevana_vrednost}
```

Пример кода 2.17: *Потпис параметара функције*

Параметри фунцкије могу бити именовани и неименовани. Приликом позивања функције са именованим параметрима потребно је навести лабеле параметара уз конкретну вредност. Приликом дефинисања функије, лабела параметра се наводи пре имена параметра, или се наводи карактер '_' за неименоване параметре. Уколико лабела параметра није експлицитно наведена сматраће се да име параметра представља и лабелу. Пример дефинисања и

позивања функције са именованим параметром и без повратног типа представљен је у коду 2.18 — *Дефинисање и позивање функције са параметром*, док пример 2.19 — *Дефинисање и позивање функције са повратном вредношћу* показује дефинисање и позивање функције са неименованим параметром и повратним типом *Int*.

```
1    // Definisanje fukncije
2    func ispisiSastojke(sastojci: [String]) {
3        for sastojak in sastojci {
4            print(sastojak)
5        }
6    }
7
8    let sastojci = ["Jaja", "Sira"]
9    // Pozivanje fukncije
10   ispisiSastojke(sastojci: sastojci)
```

Пример кода 2.18: *Дефинисање и позивање функције са параметром*

```
1    func izracunajCenu(_ proizvodi[String: Int]) -> Int {
2        var ukupnno = 0
3        for (proizvod, cena) in proizvodi {
4            ukupno += cena
5        }
6        return ukupno
7    }
8
9    let proizvodi = ["Jaja": 10, "Sira": 200]
10   let ukupnaCena = izracunajCenu(proizvodi)
```

Пример кода 2.19: *Дефинисање и позивање функције са повратном вредношћу*

Поред лабеле уз параметар може стајати и подразумевана вредност параметра, која ће представљати вредност параметра приликом извршавања тела функције уколико приликом позива функције наведени, опциони, параметар није прослеђен. Пример је приказан у коду 2.20 — *Дефинисање и позивање функције са параметрима са подразумеваним вредностима*.

```
1    func ispisiSastojkeSaDvaParametra(sastojci: [String], ispisati
        ispisatiCeloIme: Bool = true) {
2        for sastojak in sastojci {
3            if ispisatiCeloIme {
4                print(sastojak)
```

```
            }
            else {
                print(sastojak.prefix(3))
            }
        }
    }
    let sastojci = ["Jaja", "Sira"]
    ispisiSastojkeSaDvaParametra(sastojci: sastojci, ispisati: false)
    ispisiSastojkeSaDvaParametra(sastojci: sastojci)
    // Parametar 'ispisati' ce imati vrednost 'true'
```

Пример кода 2.20: *Дефинисање и позивање функције са параметрима са подразумеваним вредностима*

Опционе промењиве и рад са њима

Опционе промењиве се користе у ситуацијама када није сигурно да ли ће промењива имати неку вредност. Када опциона промењива нема вредност, њена подразумевана вредност је *nil* и са њом се мора пажљиво руковати, док употреба промењивих које нису опционог типа није могућа пре њихове иницијализације.

Када се дефинише опциона промењива експлицитно се наводи ког је типа након чега следи знак '?' и након тога се може, а не мора, извршити иницијализација. Уколико се промењива иницијализује без експлицитног навођења опционог типа, неопходно је да израз којим се иницијализује промењива буде опционог типа, у супротном промењивој не би био додељен опциони тип и она не би могла да се користи као опциона промењива. Пример дефинисања опционе промењиве уз иницијализацију и експлицитно навођење опционог типа, као и два начина иницијализације без навођења типа може се видети у делу 2.21 — *Дефинисање опционе промењиве.*

```
    // Opciona promenljiva tipa 'Int?'
    var opciona: Int? = 42
    // Promenljiva tipa 'String'
    var brojUOblikuStringa = "55"
    // Opciona promenljiva tipa 'Int?'
    var konvertovaniBroj = Int(brojUOblikuStringa)
```

Пример кода 2.21: *Дефинисање опционе промењиве*

Опционо уланчавање (eng. Optional chaining) је процес безбедног приступања својствима и методама опционе промењљиве без потребне претходне провере да ли промењљива има вредност. Уколико опциона промењљива има вредност, дохватање својства или позив методе ће успети, док уколико опциона промењљива нема вредност, резултат опционог уланчавања ће такође вратити вредност *nil*. Унутар опционог уланчавања постоји могућност надовезивања више опционих промењљивих. Пример примене опционог уланчавања приказан је у примеру кода 2.22 — *Опционо уланчавање*.

```
1    struct Recept {
2        var ime: String?
3
4        // Vraca ime ispisano velikim slovima ukoliko postoji, u
             suprotnom vraca 'nil'
5        func imeVelikimSlovima() -> String? {
6            return self.ime?.uppercased()
7        }
8    }
9
10   var recept: Recept? = Recept("karbonara")
11
12   // Ispisuje prvo slovo imena recepta ukoliko ono postoji
13   print(recept?.imeVelikimSlovima()?.prefix(1))
14   // 'K'
```

Пример кода 2.22: *Опционо уланчавање*

У неким ситуацијама не може се радити са опционим промењљивима, на пример када се прослеђују као параметри функције која очекује конкретну вредност. У том случају опциона промењљива се мора одмотати и узети вредност која се налази у њој. Да при томе не би дошло до грешке, постоје два начина за безбедно одмотавање опционе промењљиве и руковање са *nil* вредношћу.

Први начин је коришћењем условног гранања (*if* или *guard*) приказан у примеру 2.23 — *Одмотавање опционе промењљиве коришћењем услова*, а други начин задавањем подразумеване вредности односно коришћењем оператора *nil*-сједињавања приказан у примеру 2.24 — *Одмотавање опционе промењљиве задавањем подразумеване вредности*.

```
1    func saberiDvaBroja(_ prvi: Int, _ drugi: Int) -> Int {
2        return prvi+drugi
```

```
3     }
4
5     var opcioniBroj: Int? = 42
6     var broj = 25
7
8     // saberiDvaBroja(opcioniBroj, broj) -> greska, 'opcioniBroj' je
          tipa Int? dok funkcija ocekuje parametar tipa Int
9
10    // 1. nacin koriscenjem if-a
11    if let raspakovaniBroj = opcioniBroj {
12        saberiDvaBroja(raspakovaniBroj, broj) // 'raspakovaniBroj' je
              tipa Int
13    }
14    else {
15        print("Prvi broj nema vrednost, ne moze se sabrati")
16    }
17
18    // 2. nacin koriscenjem if-a
19    if opcioniBroj != nil {
20        saberiDvaBroja(opcioniBroj!, broj)
21    }
22    else {
23        print("Prvi broj nema vrednost, ne moze se sabrati")
24    }
25
26    // 3. nacin koriscenjem guard-a
27    guard opcioniBroj != nil else {
28        print("Prvi broj nema vrednost, ne moze se sabrati")
29        return
30    }
31    saberiDvaBroja(opcioniBroj!, broj)
```

Пример кода 2.23: *Одмотавање опционе променљиве коришћењем услова*

```
1     func saberiDvaBroja(_ prvi: Int, _ drugi: Int) -> Int {
2         return prvi+drugi
3     }
4
5     var opcioniBroj: Int? = 42
6     var broj = 25
7
8     saberiDvaBroja(opcioniBroj ?? 5, broj)
```

Пример кода 2.24: *Одмотавање опционе променљиве задавањем подразумеване вредности*

2.4 Напредни концепти језика

У овом делу ће бити приказани неки од напредних концепата програмског језика *Swift*. Концепти који ће бити представљени су: дефинисање генеричких фунцкија и употреба затворења, као и опис класа и структура у програмском језику *Swift*.

Генеричке функције и затворења

Као и други објектно оријентисани програмски језици и *Swift* пружа могућност дефинисања генеричких функција[6] које се могу користити над различитим конкретним типовима. *Swift* такође омогућава дефинисање функција са две или више повратних вредности, које се враћају као n-торке. Конкретан пример генеричке функције са две повратне вредности приказан је у коду 2.25 — *Дефинисање и позивање генеричке функције са више повратних вредности.*

```
1   // Definisanje generickе funkcije sa dve povratne vrednosti
2   func minMax<T>(niz: [T]) -> (min: T, max: T)? {
3       guard !niz.isEmpty else {
4           return nil
5       }
6       let minimum = niz.min()
7       let maksimum = niz.max()
8
9       return (minimum, maksimum)
10  }
11
12  let nizVremenaPripremeRecepata = [25, 12, 5, 19, 99]
13  let najkraceINajduzeVreme = minMax(niz: nizVremenaPripremeRecepata
        )
14  let najkraceVreme = najkraceINajduzeVreme?.min
15  let najduzeVreme = najkraceINajduzeVreme?.max
```

[6]Генеричка функција је функција са генеричким параметрима или генеричким повратним вредностима. Генерички тип може представљати више различитих типова одједном који задовољавају одређене услове које је програмер дефинисао.

Пример кода 2.25: *Дефинисање и позивање генеричке функције са више повратних вредности*

Прослеђени параметри унутар функција су константе и њихова вредност се не може мењати у телу функције. Да би се омогућило заобилажење овог правила, *Swift* је увео кључну реч *inout* која се наводи приликом дефинисања функције, а пре типа сваког од параметара за које ће бити омогућена промена вредности у телу функције. Још једна промена коју је потребно применити је употреба карактера & приликом прослеђивања променљиве у позиву функције. Употреба овако прослеђених параметара приказана је на примеру генеричке функције која замењује вредности два прослеђена параметра 2.26 — *Дефинисање и позивање функције са променљивим параметрима.*

```
1   func zameniDvaParametra<T>(prvi: inout T, drugi: inout T) {
2       var pomocna = prvi
3       prvi = drugi
4       drugi = pomocna
5   }
6
7   // Prosledjeni parametri moraju biti promenljive
8   var prviRecept = "Cezar salata"
9   var drugiRecept = "Omlet"
10
11  print(prviRecept + ", " + drugiRecept)
12  // Cezar salata, Omlet
13
14  zameniDvaParametra(prvi: &prviRecept, drugi: &drugiRecept)
15  print(prviRecept + ", " + drugiRecept)
16  // Omlet, Cezar salata
```

Пример кода 2.26: *Дефинисање и позивање функције са променљивим параметрима*

Затворења су самостални блокови кода који се могу прослеђивати и користити у коду. Слични су ламбда изразима у другим модерним језицима. Изрази затворења представљају начин за писање затворења у једној линији (енг. *inline*), притом пружајући неколико синтаксних оптимизација у виду кратке форме, разумљивости и изражајности. Синтакса израза затворења приказана је у примеру 2.27 — *Синтакса израза затворења.* На примеру 2.28 — *Израз затворења за сортирање* показана је *Swift* метода *sorted* и како се

једно затворење може написати на неколико начина, од целе функције па све до само једног карактера.

```
1    (parametri) -> tip_povratne_vrednosti in
2        naredbe
```

Пример кода 2.27: *Синтакса израза затворења*

```
1    func uporediBrojeve(_ broj1: Int, _ broj2: Int) -> Bool {
2        return broj1 < broj2
3    }
4
5    let nasumicniBrojevi = [2, 10, 5, 18, 100, -11, -25, 55, 72]
6    // Prosledjivanjem funkcije
7    var sortiraniBrojevi = nasumicniBrojevi.sorted(by: uporediBrojeve)
8
9    // Koriscenjem zatvorenja
10   sortiraniBrojevi = nasumicniBrojevi.sorted(by: { (broj1: Int,
         broj2: Int) -> Bool in
11       return broj1 < broj2
12   })
13
14   // Bez eksplicitnog navodjenja tipa parametra
15   sortiraniBrojevi = nasumicniBrojevi.sorted(by: {broj1, broj2 in
16       return broj1 <  broj2
17   })
18
19   // Kada u zatvorenju postoji samo jedna naredba, nije potrebno
         navodjenje kljucne reci return, povratna vrednost bice vrednost
          izvrsenja te naredbe
20   sortiraniBrojevi = nasumicniBrojevi.sorted(by: {broj1, broj2 in
21       broj1 < broj2
22   })
23
24   // Swift omogucava i kratka imena parametara, radi pruzanja kraceg
          zapisa
25   sortiraniBrojevi = nasumicniBrojevi.sorted(by: {
26       $0 < $1
27   })
28
29   // Kada koristimo tipove za koje je vec definisano ponasanje
         prilikom poredjenja, mozemo proslediti samo kako zelimo da
         sortiramo clanove niza
30   sortiraniBrojevi = nasumicniBrojevi.sorted(by: <)
```

Пример кода 2.28: *Израз затворења за сортирање*

Затворења се могу проследити и као параметри функције. Једино ограничење је да затворење мора ићи као последњи параметар функције. Најчешћи разлог за овакву употребу затворења је сигурност да ће се наредбе у затворењу извршити након што се заврши извршавање функције. Оваква врста затворења назива се репно затворење (енг. *trailing closures*), пример у коду 2.29 — *Репно затворење*.

```
1   func ucitajSliku(sa url: URL, completition: (Image?) -> Void) {
2       if let slika = skini("Omlet.jpg", sa: url) {
3           completition(slika)
4       }
5       else {
6           completition(nil)
7       }
8   }
9
10  ucitajSliku(sa: lokalniUrl) { slika in
11      if let slika = slika {
12          celija.image = slika
13      }
14      else {
15          celija.image.backgroundColor = .gray
16      }
17  }
```

Пример кода 2.29: *Репно затворење*

Класе и структуре

Класе и структуре су конструкције опште намене које имају своја својства и методе. За разлику од већине других програмских језика, класе и структуре у *Swift*-у су сличне по функционалности (биће објашњено у наставку поглавља), па се често за објекат класе и инстанцу структуре користи заједнички назив — инстанца.

У поређењу класа и структура могућности које обе конструкције омогућавају су: дефинисање својстава, дефинисање метода, дефинисање иницијализатора, надограђивање коришћењем проширења (енг. *extensions*), импле-

ментација протокола. Функционалности које поседују само класе су: наслеђивање друге класе, провера типа инстанце у времену извршавања програма, деиницијализација.

Још једна разлика између класа и структура може се видети приликом дефинисања константне (употребом кључне речи *let*) инстанце класе или структуре. Уколико је дефинисана константна инстанца класе, корисник и даље има могућност промене поља те инстанце која нису дефинисана као константна, јер је константна референца инстанце, али не и вредност за коју је та референца везана. Уколико је дефинисана константна инстанца структуре, корисник нема могућност промене ниједног од њених поља.

Уколико неко својство класе или структуре нема унапред дефинисану вредност, оно мора бити:

- део иницијализације, ако је константно,
- део иницијализације или опционог типа, ако је променљиво.

Дефинисање и инстанцирање класе и структуре, као и приступање својствима и методама инстанце коришћењем тачка синтаксе (eng. *dot syntax*) може се видети у примеру 2.30 — *Дефинисање класе и структуре.*

```
1   // Definisanje strukture
2   struct OkvirPozadine {
3       var visina = 0
4       var sirina = 0
5       var boja: UIColor?
6   }
7
8   // Definisanje klase
9   class GlavniIzgled {
10      var okvir = OkvirPozadine()
11      var slika: UIImage?
12      var ponovitiSliku = false
13  }
14
15  // Instanciranje strukture
16  let okvir = OkvirPozadine()
17  // Instanciranje klase
18  let glavniIzgeld = GlavniIzgled()
19
20  // Pristupanje clanovima instance
```

```
21      let okvirGlavnogIzgleda = glavniIzgled.okvir
22      let sirinaOkviraPozadine = okvir.sirina
23
24      // Strukture imaju automatski generisane inicijalizatore za sva
            svojstva
25      let maliSiviOkvir = OkvirPozadine(visina: 50, sirina: 50, boja: .
            gray)
```

Пример кода 2.30: *Дефинисање класе и структуре*

Уколико унутар једне класе постоји инстанца друге класе, инстанцирање друге класе се може одложити док не буде неопходно употребом лењог својства (енг. *lazy property*). Лења својства се користе када инстанцирање класе зависи од других параметара који нису познати у тренутку иницијализације главне класе или када инстанцирање може узети много времена и добро је одложити га док не буде неопходно (можда у неким случајевима не буде уопште искоришћено). Пример употребе лењог својства налази се у коду 2.31 — *Лењо својство*.

```
1       struct UcitavanjeFajla {
2           var imeFajla = "recepti.txt"
3       }
4
5       class MenadzerPodataka {
6           lazy var ucitavanje = UcitavanjeFajla()
7           var podaci: [String] = []
8       }
9
10      var menadzer = MenadzerPodataka()
11      menadzer.podaci.append("Prvi podatak")
12      menadzer.podaci.append("Drugi podatak")
13
14      // Pre izvrsenja f-je 'print', instancira se klasa '
            UcitavanjeFajla'
15      print(menadzer.ucitavanje.imeFajla)
```

Пример кода 2.31: *Лењо својство*

За разлику од својстава и лењих својстава који у себи чувају одређене вредности, рачунајућа својства (енг. *computed properties*) не чувају вредности, већ садрже блок дохватача (енг. *getter*) (који је обавезан) и опционо блок постављача (енг. *setter*) којима се дохватају и постављају вредности других

променљивих. Приликом дефинисања рачунајућег својства, уколико својство садржи и дохватач и постављач потребно је експлицитно навести блокове за оба параметра *get* и *set*, док у случају да садржи само дохватач може се дефинисати само један блок за који ће компајлер закључити да је у питању блок дохватача. У примеру 2.32 — *Рачунајућа својства* приказано је рачунајуће својство *vremePripreme* које садржи и дохватач и постављач, као и рачунајуће својство *ukupnaCena* које има само блок дохватача.

```
1   struct Namirnica {
2       var ime: String
3       var cena: Double
4       var kolicina: Double
5   }
6
7   struct Recept {
8       var potrebneNamirnice: [Namirnica]
9       var vremeKuvanja: Int // U minutima
10      var ukupnoVremeSpremanja: Int // U minutima
11      var vremePripreme: Int {
12          get {
13              return self.ukupnoVremeSpremanja - self.vremeKuvanja
14          }
15          set(novoVremePripreme) {
16              self.ukupnoVremeSpremanja = self.vremeKuvanja +
                    novoVremePripreme
17          }
18      }
19      var ukupnaCena: Double {
20          return potrebneNamirnice.reduce(0) {
21              $0 + $1.cena * $1.kolicina
22          }
23      }
24  }
25
26  var lazanje = Recept(potrebneNamirnice: [Namirnica(ime: "Kore",
        cena: 250, kolicina: 1), Namirnica(ime: "Mleveno meso", cena:
        500, kolicina: 0.75), Namirnica(ime: "Sos", cena: 100, kolicina
        : 1)], vremeKuvanja: 25, ukupnoVremeSpremanja: 70)
27
28  print("Ukupna cena recepta: \(lazanje.ukupnaCena)")
29  // Ukupna cena recepta: 725
30  print("Vreme pripreme recepta: \(lazanje.vremePripreme)")
```

```
31      // Vreme pripreme recepta: 45
32      print("Staro ukupno vreme pripreme:
33            \(lazanje.ukupnoVremeSpremanja)")
34      // Staro ukupno vreme pripreme: 70
35      lazanje.vremePripreme = 30 // Umesto 45
36      print("Novo ukupno vreme pripreme:
37            \(lazanje.ukupnoVremeSpremanja)")
38      // Novo ukupno vreme pripreme: 55
```

Пример кода 2.32: *Рачунајућа својства*

Посматрачи својстава посматрају и реагују на промену вредности својства унутар којег су имплементирани. Посматрачи реагују увек када је нова вредност додељена својству, чак иако је нова вредност једнака старој. Могу се користити унутар кориснички дефинисаних својстава, наслеђених својстава и наслеђених рачунајућих својстава. Посматрачи својстава који се могу дефинисати унутар својства су *willSet*, које се позива пре постављања нове вредности и *didSet*, које се позива након постављања нове вредности својства. Функцији *willSet* се аутоматски прослеђује параметар у којој је смештена нова вредност својства и уколико програмер не наведе експлицитно име параметра, оно ће бити *newValue*. Исто важи и за функцију *didSet* којој се прослеђује параметар старе вредности својства са називом *oldValue*. Пример употребе посматрача приказан је у делу 2.33 — *Посматрачи својства*.

```
1       struct Namirnica {
2           var ime: String
3           var cena: Double {
4               willSet {
5                   print("Nova cena: \(newValue)")
6               }
7               didSet(staraCena) {
8                   if cena < staraCena {
9                       print("Popust na namirnici: \(ime)")
10                  }
11              }
12          }
13      }
14      var mleko = Namirnica(ime: "Mleko", cena: 105.0)
15      mleko.cena = 98
16      // Nova cena: 98
17      // Popust na namirnici: Mleko
```

Пример кода 2.33: *Посматрачи својстава*

Методе су функције које су везане за одређени тип класе, структуре или набрајања[7] (енг. *enumerations*). Методе инстанце су функције које припадају одређеној инстанци и подржавају функционалности те инстанце. Дефинисање метода класе и позивање тих метода над конкретном инстанцом класе приказано је у примеру 2.34 — *Методе*.

```
1   class Recept {
2       var ime: String
3
4       // Koriscenjem kljucne reci 'self', naglasava se pristupanje
            svojstvu/metodi klase
5       init(ime: String) {
6           self.ime = ime
7       }
8
9       // Metod koji ispisuje svojstvo 'ime'
10      func ispisiIme() {
11          print(ime)
12      }
13
14      // Metod koji menja svojstvo 'ime', parametrom 'ime'
15      func promeniIme(novo ime: String) {
16          self.ime = ime
17      }
18  }
19
20  var recept = Recept("Bolonjeze")
21  recept.ispisiIme()
22  // Bolonjeze
23
24  recept.promeniIme(novo: "Karbonara")
25  recept.ispisiIme()
26  // Karbonara
```

Пример кода 2.34: *Методе*

Као и у свим објектно оријентисаним језицима, и у *Swift*-у постоји наслеђивање класа. Класа која наследи другу класу наслеђује сва њена својства и

[7]Набрајање дефинише заједнички тип за групу повезаних вредности и омогућава безбедан рад са тим вредностима у коду.

методе које нису дефинисане као приватне и може их мењати, односно предефинисати (енг. *override*). Свака класа која не наслеђује ниједну другу класу назива се основна класа. Пример наслеђивања класе може се видети у делу 2.35 — *Наслеђивање класа*.

```
1     class Pravougaonik {
2         var sirina = 0
3         var duzina = 0
4
5         func izracunajPovrsinu -> Int {
6             return sirina * duzina
7         }
8     }
9
10    class Kvadrat : Pravougaonik {
11        override func izracunajPovrsinu -> Int {
12            return sirina * sirina
13        }
14    }
```

Пример кода 2.35: *Наслеђивање класа*

Приликом дефинисања класе, конвенција у *Swift*-у је да се за једноставније структуре користи кључна реч *Struct*, а не кључна реч *Class*. Надоградња типа се користи да би се типови одвојили у смислене целине, на пример наслеђивање неке надкласе или имплементација протокола, што се може видети у примеру 2.36 — *Надоградња постојећег типа (класе, структуре)*. Такође се могу надоградити већ постојећи (системски) типови унутар којих ће бити имплементиране нове функције, чиме се елиминише потреба да иста логика буде имплементирана више пута у коду, приказано у примеру 2.37 — *Надоградња Swift класе*. Ограничење код екстензије је да се не могу додавати нова поља и променљиве унутар типа који се надограђује.

```
1  struct Recept {
2      var ime: String
3      var vremePripreme = 30
4      var sastojci: [String] = []
5      var slika: UIImage?
6
7      init(_ name: String) {
8          self.name = name
9      }
```

```
10 }
11
12 var recept = Recept("Omlet")
13
14 extension Recept {
15     func dodajSastojak(_ sastojak: String) {
16         self.sastojci.append(sastojak)
17     }
18 }
19
20 self.recept.dodajSastojak("2 jaja")
```

Пример кода 2.36: *Надоградња постојећег типа (класе, структуре)*

```
1  extension UIImage {
2      func slikaRecepta(_ imeSlike: String) -> UIImage {
3          var image = UIImage(named: imeSlike)
4          var okvirSlike = image.view.frame
5          okvirSlike.width = 50
6          okvirSlike.height = 55
7          image.view.frame = okvirSlike
8          image.backgroundColor = .gray
9          return image
10     }
11 }
12
13 self.recept.slika = UIImage.slikaRecepta(self.recept.ime)
```

Пример кода 2.37: *Надоградња Swift класе*

Аутоматски бројач референци

ARC се користи за праћење и управљање меморијом коју апликација користи. Бројач референци води рачуна само о инстанцама класе, док инстанце структуре и набрајања игнорише јер оне представљају тип вредности, а не референтни тип. Сваки пут када се креира нова инстанца класе, *ARC* одвоји део меморије потребан за смештање информација о инстанци. За сваку инстанцу *ARC* води рачуна о броју својстава, константи и променљивих које реферишу на посматрану инстанцу. *ARC* ће деалоцирати део меморије у којој је смештена инстанца тек када све јаке (енг. *strong*) референце ка њој нестану. Када се инстанца класе додели својству, константи или променљивој

уколико се експлицитно не наведе другачије, аутоматски ће бити креирана јака референца ка инстанци.

Ретка ситуација где програмер мора водити рачуна о управљању меморијом унутар програмског језика *Swift* је избегавање креирања две инстанце неких класа које међусобно држе јаке референце једна ка другој, јер у овој ситуацији долази до формирања циклуса јаких референци (енг. *strong reference cycle*) што онемогућује *ARC* да деалоцира меморију у којој се налазе ове две инстанце (увек ће једна ка другој имати јаку референцу) и долази до цурења меморије. Циклус јаких референци се може спречити употребом *weak* или *unowned* референци које *ARC* не узима у обзир. Пример употребе слабе (енг. *weak*) референце приказан је у коду 2.38 — *Слаба референца*.

```
1   protocol NoviReceptDelegate : AnyObject {
2       func receptJeKreiran(_ viewController: NoviReceptController, _
            recept: Recept)
3   }
4
5   class NoviReceptController() : UIViewController {
6       // Definisanje promenljive sa slabom referencom
7       weak delegate: NoviReceptDelegate?
8       ...
9       func sacuvajRecept(_ recept: Recept) {
10          ...
11          self.delegate?.receptJeKreiran(self, recept)
12      }
13  }
14  class ReceptController : UIViewController {
15      var novRecept = NoviReceptController()
16      override viewDidLoad() {
17          ...
18          self.novRecept.delegate = self
19      }
20      ...
21  }
22  extension ReceptController : NoviReceptDelegate {
23      func receptJeKreiran(_ viewController: NoviReceptController, _
            recept: Recept) {
24      }
25  }
```

Пример кода 2.38: *Слаба референца*

2.5 Развојно окружење *Xcode*

Xcode је интегрисано развојно окружење (ИРО) намењено употреби на оперативном систему *macOS*. Софтвер је бесплатан и могуће је преузети га на *Mac App Store*-у[8] [4], а развија га компанија *Apple*.

Основно

ИРО *Xcode* се користи за развој софтвера намењених оперативним системима *iOS*, *iPadOS*, *watchOS*, *tvOS* и *macOS*. Прва верзија *Xcode*-а објављена је 2003. године, а последња стабилна верзија је *Xcode*13. *Xcode* укључује алат командне линије (енг. *command line tools*) који омогућава развој софтвера помоћу терминала, начин развоја софтвера који одговара стилу развоја на *UNIX*-у. На слици 2.2 може се видети званични лого ИРО-а *Xcode*13.

Слика 2.2: *Званични лого ИРО-а Xcode13*

Xcode се састоји од неколико алата који помажу програмеру приликом развоја апликација за платформе компаније *Apple*, од креирања апликације, преко тестирања и оптимизације, до прослеђивања на *App Store*. Најзначајнији алати који су део *Xcode*-а су симулатор и инструменти.

[8]Платформа која служи за дигиталну дистрибуцију апликација намењених оперативном систему *macOS*.

Симулатор

Симулатор се користи за тестирање апликације у току развоја уколико не постоји могућност употребе физичког уређаја. Тестирање на симулатору у неким ситуацијама може бити и боље јер пружа могућност тестирања апликације на више различитих уређаја (симулатора) одједном (на пример, различите генерације телефона *iPhone*, као и различите верзије оперативног система).

Симулатор је део *Xcode*-а; инсталира се уз њега, а покреће се и понаша као апликација оперативног система *macOS* и омогућава симулацију свих уређаја са платформи компаније *Apple* (*iPhone*, *iPad*, *Apple Watch*, *Apple TV*). Приликом тестирања могуће је и покретање више симулатора за различите платформе да би се тестирала њихова компатибилност, као на пример сарадња апликације на *iPhone*-у и *Apple Watch*-у. Додатне погодности које пружа симулатор су: интеракција са апликацијом коришћењем миша и тастатуре, отклањање неисправности у апликацији, оптимизација графичког приказа. На слици 2.3 може се видети истовремена употреба симулатора телефона *iPhone 13 Pro Max* са оперативним системом *iOS* 15.0 и подешеном тамном бојом приказивања (енг. *dark appearance*), телефона *iPhone SE — 2nd generation* такође са оперативним системом *iOS* 15.0, али светлом бојом приказивања, таблета *iPad Pro (9.7-inch)* са оперативним системом *iOS* 12.0 и паметног сата *Series 7 — 45mm* са оперативним системом *watchOS* 8.0.

Слика 2.3: *Приказ неколико симулатора*

Инструменти

Инструменти су моћан алат, део *Xcode*-а, који служе за анализу перформанси апликације као помоћ при њеном тестирању да би се боље разумело понашање апликације и омогућила додатна оптимизација перформанси. Коришћење инструмената од почетка развијања апликације доприноси раном откривању појединих грешака и олакшава њихово решавање. Неке од функција које инструменти омогућавају су:

- истраживање понашања апликације или процеса,
- испитивање карактеристика специфичних за уређаје као што су *Bluetooth* и *Wi-Fi*,
- профајлирање апликације у симулатору или на физичком уређају,
- анализа перформанси апликације,
- откривање проблема са меморијом,
  - цурење меморије,

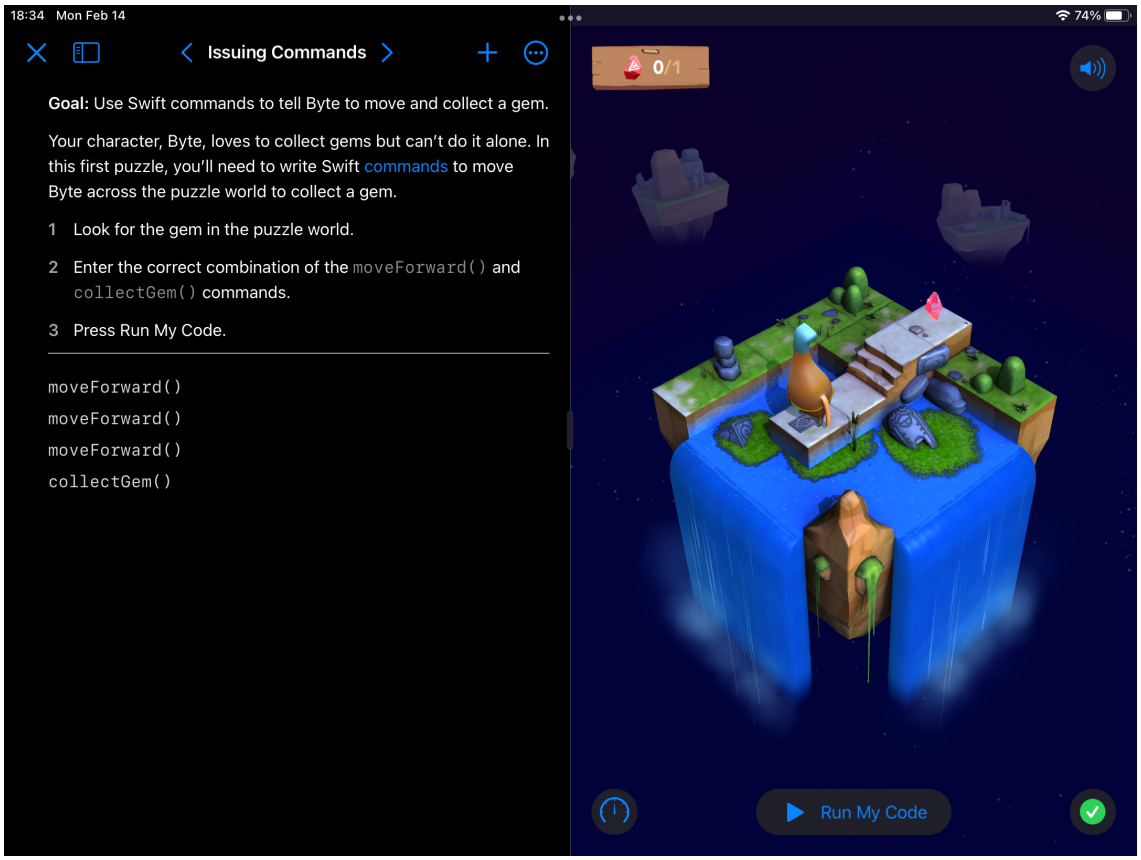
  - напуштена меморија (енг. *abandoned memory*),
  - зомби објекти (деалоцирани објекти који се још увек чувају),

- оптимизовање апликације ради боље енергетске ефикасности.

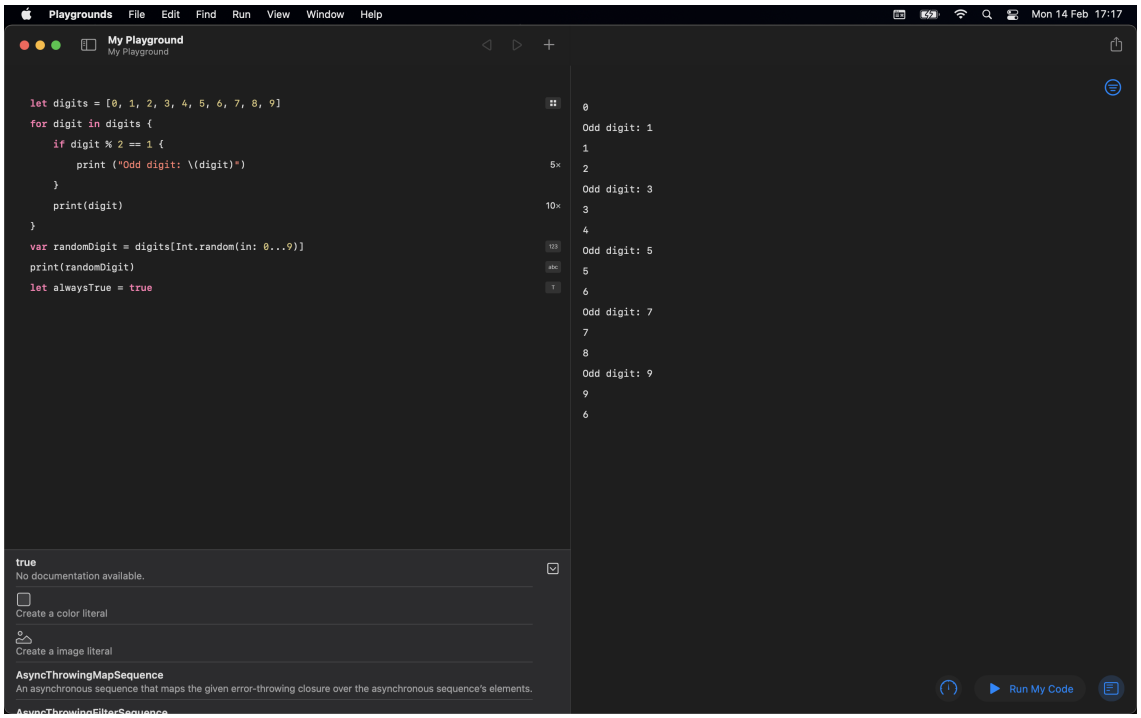
Апликација *Swift Playgrounds*

Попут апликације *Swift Playgrounds iPad* чији је један од пројеката намењен почетницима приказан на слици 2.4, однедавно постоји апликација *Swift Playgrounds* за оперативни систем *macOS* која је одлична за почетнике, али и за искусније програмере који желе да испробају део кода или да се само мало забаве. Резултат извршавања ће бити одмах приказан, као што се може видети на слици 2.5. Када пролази кроз петљу за сваки израз који утиче на рад програма биће исписано колико пута се извршио. Поред променљивих је видљив њихов конкретан тип, а уколико је променљива иницијализована видљива је и њена вредност.

Основни део апликације *Swift Playgrounds*, програм за обраду текста у којем је могуће писање кода у *Swift*-у (*Playground*), је део ИРОа *Xcode*. Да би корисник отворио нови прозор *Playground*-а, потребно је да у отвореној апликацији *Xcode* из траке менија (енг. *menu bar*) изабере *File*, из падајућег менија *New* и из новог падајућег менија *Playground*.



Слика 2.4: *Приказ апликације Swift Playgrounds на iPad-у*



Слика 2.5: *Приказ апликације Swift Playgrounds на macOS-у*

Менаџер пакета

Swift менаџер пакета (енг. *Swift package manager*) је више-платформски алат за израду, покретање, тестирање и груписање *Swift* библиотека и извршних датотека. Помоћу менаџера пакета могу се најлакше поделити библиотеке и изворни кодови. Конфигурација самог менаџера као и сам *Swift* менаџер пакета су такође писани у *Swift*-у, чинећи конфигурацију циљаних извршних датотека и управљање зависностима међу пакетима веома једноставним.

2.6 Окружење за израду апликација са графичким интерфејсом *SwiftUI*

SwiftUI је радно окружење које служи за израду апликација са графичким интерфејсом погодним за све платформе компаније *Apple*, користећи концизну моћ програмског језика *Swift*. Омогућава креирање разноврсних апликација уз само један скуп алата и програмског интерфејса апликације (енг. *application programming interface, API*) [1].

Основна структура

Радно окружење *SwiftUI* пружа велики број погледа (енг. *views*), контрола и распоредних структура (енг. *layout structures*) који олакшавају процес израде корисничког интерфејса апликације. Уз то садржи и алате који управљају током података од модела до погледа и контрола[9], са којима корисник може интераговати. Интеракција се одвија преко додира, гестова и других типова улазних података у апликацији који се потом обрађују помоћу обрађивача догађаја (енг. *event handlers*).

Структура апликације се дефинише помоћу протокола *App* и попуњава се сценама које садрже погледе чији скуп чини кориснички интерфејс апликације. *SwiftUI* омогућава и креирање нових погледа, једини услов је да тај поглед имплементира протокол *View*. Нови поглед се може комбиновати са другим, корисничким или погледима радног окружења, као што су текстуална поља, слике и многи други да би се направили комплекснији погледи који ће бити погодни за све кориснике апликације.

Карактеристике

SwiftUI је настао као наследник радног окружења *UIKit*, а основна карактеристика која издваја радно окружење *SwiftUI* од *UIKit*-а је другачија програмска парадигма, конкретно декларативна синтакса. Више о разликама ова два радна окружења биће описано у поглављу 2.6 — Разлика између радних окружења *SwiftUI* и *UIKit*.

Декларативна синтакса омогућава програмерима да једноставно опишу понашање корисничког интерфејса. Кôд је много једноставнији за читање и разумевање као и за писање, чиме је обезбеђена значајна уштеда времена приликом писања новог кода и одржавања већ постојећег. Пример кода у *SwiftUI*-у приказан је у делу 2.39 — *Пример SwiftUI кода*. Модификатор *@State* биће објашњен у делу 2.6 — Стање и ток података.

```
1 // Ucitavanje SwiftUI radnog okruzenja
2 import SwiftUI
3
4 // Kreiranje strukture koja ce sadrzati glavni pogled
```

[9] Архитектурни образац Модел-Поглед-Контролер (енг. *Model-View-Controller*, MVC) који се заснива на подели на три целине, модел — структура података, поглед — приказ података у корисничком окружењу, контролор — управљање моделом.

```
5  struct Content : View {
6
7      // Definisanje promenljive 'recepti'
8      @State var recepti = RecepModel.listaRecepata
9
10     // Definisanje tela pogleda
11     var body: some View {
12         // Izlistavanje svih recepata kroz listu
13         List(recepti.stavke, action: recepti.izabranaStavka) { recept
               in
14             // Prikaz slike
15             Image(recept.slika)
16             // Definisanje vertikalnog skupa elemenata
17             VStack(alignment: .leading) {
18                 // Prikaz teksta
19                 Text(recept.ime)
20                 // Prikaz teksta sive boje
21                 Text(recept.vremePripreme)
22                     .color(.gray)
23             }
24         }
25     }
26 }
```

Пример кода 2.39: *Пример SwiftUI кода*

Стање и ток података

Декларативно програмирање омогућава да се за погледе вежу одговарајући модели података. Када год се неки од података промени, *SwiftUI* аутоматски поново учита све погледе за који су промењени подаци везани и прикаже их кориснику, тако да програмер не мора бринути о томе. Ово се постиже променљивим стањима и везивањем, чиме се подаци везују за конкретне погледе. Тиме се остварује једини извор истине[10] (енг. *single source of truth, SSoT*) за све податке и олакшава одржавање тачности података у сваком тренутку.

У зависности од конкретне потребе у тренутној ситуацији, постоји више начина за остваривањем јединог извора истине:

[10]Једини извор истине је начин структуирања информационих модела и шеме података тако да се сваки податак обрађује и мења на само једном месту.

- *State* — Омогућава локално управљање стањем корисничког интерфејса, пример 2.40 — *Омотачи података — State*. Када је променљива означена као *State*, другом погледу се мора проследити са префиксом '$' уколико се жели омогућити промена њене вредности.

- *BindableObject* — Користећи омотач својства *ObservedObject* може се приступити спољашњој референци на модел података који имплементира протокол *ObservableObject*. Уколико је променљива смештена у спољашње окружење, може јој се приступити користећи омотач својства *EnvironmentObject*. Инстанцирање посматрајућег (енг. *observable*) објекта директно у погледу постиже се коришћењем модификатора *StateObject*.

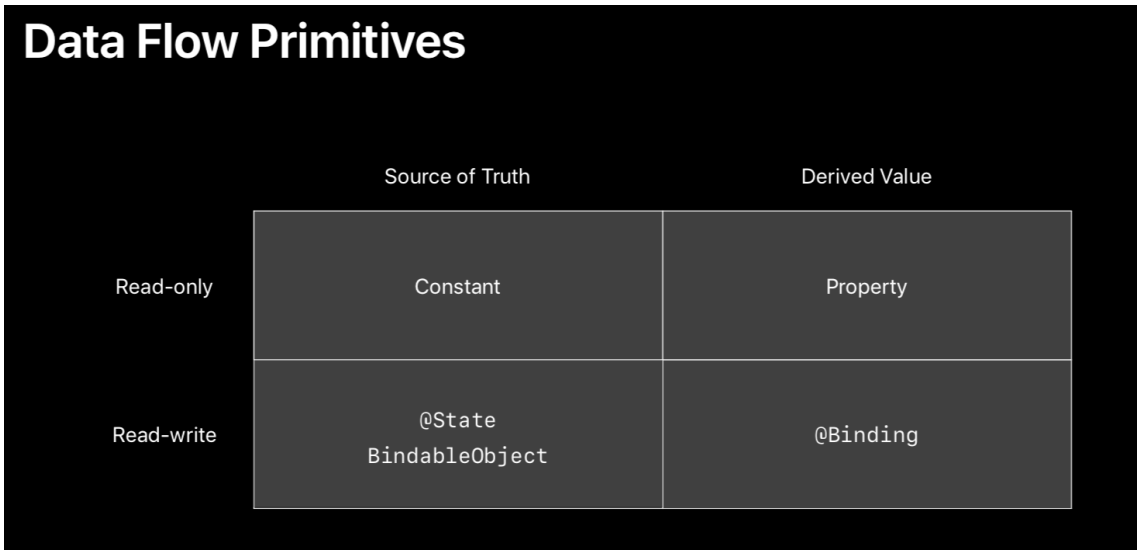
- *Binding* — Користи се за дељење референце на јединu извор истине, пример 2.41 — *Омотачи података — Binding*.

- *Environment* — Подаци сачувани у *Environment-у* се могу делити кроз целу апликацију, пример 2.42 — *Омотачи података — Environment*.

- *PreferenceKey* — Прослеђивање података уз хијерархију погледа, од детета ка родитељу.

- *FetchRequest* — Управљање трајним подацима који се чувају унутар *Core Data*.

Графички приказ модификатора може се видети на слици 2.6 — *Различити омотачи података*.



Слика 2.6: *Различити омотачи података*

```
1   struct Recept: View {
2       var recept: ReceptPodatak
3       @State private var daLiJeOmiljen = false
4
5       var body: some View {
6           VStack {
7               Text(recept.ime)
8               // 'OmiljenRecept' je pogled koji sadrzi zvezdicu koja
                    oznacava da li je recept medju omiljenima (puna
                   zvezdica --- jeste, prazna --- nije)
9               OmiljenRecept(daLiJe: $daLiJeOmiljen)
10          }
11      }
12  }
```

Пример кода 2.40: *Омотачи података — State*

```
1   struct Recept: View {
2   var recept: ReceptPodatak
3   // Promenljiva 'daLiJeOmiljen' je definisana u jednom pogledu, a
       moze se menjati u drugom
4   @Binding var daLiJeOmiljen: Bool
5
6   var body: some View {
7       Button(action: {
8           // Akcija dugmeta koja menja promenljivu 'daLiJeOmiljen'
```

```
9               self.daLiJeOmiljen.toggle()
10          }) {
11              // Provera promenljive 'daLiJeOmiljen' i prikaz
                    odgovarajuce slike
12              Image(systemName: daLiJeOmiljen ? "star.fill" : "star.
                    empty")
13          }
14      }
15  }
```

Пример кода 2.41: *Омотачи података — Binding*

```
1       struct Kulinarstvo_widgetEntryView : View {
2       var entry: Provider.Entry
3
4       // Cita podatke za 'widgetFamily' iz okruzenja aplikacije i smesta
            ih u promenljivu 'widgetFamily'
5       @Environment(\.widgetFamily) var widgetFamily
6
7       @ViewBuilder
8       var body: some View {
9           // U zavisnosti od promenljive 'widgetFamily' prikazuje se
                odgovarajuci widget
10          switch widgetFamily {
11          case .systemSmall:
12              RecipeView(recipe: entry.recipe)
13                  .widgetURL(entry.recipe.url)
14          case .systemMedium:
15              RecipeMediumView(recipe: entry.recipe, ingredients: entry.
                    recipe.ingredients.count > 3 ? Array(entry.recipe.
                    ingredients.dropLast(entry.recipe.ingredients.count -
                    3)) : entry.recipe.ingredients)
16          default:
17              Text("")
18          }
19      }
20  }
```

Пример кода 2.42: *Омотачи података — Environment*

Разлика између радних окружења *SwiftUI* и *UIKit*

UIKit и *SwiftUI* су радна окружења развијена од стране *Apple*-а, која помажу приликом израде корисничког интерфејса апликације. Генерално, највећа разлика између ова два радна окружења је у начину размишљања, како доћи до решења и како то решење касније имплементирати. Ова разлика ће бити показана на једном конкретном примеру. Креирање форме за пријављивање на одређени сајт уз креирање вертикалног скупа елемената, који ће бити хоризонтално и вертикално центрирани у скупу, а скуп ће се састојати од два текстуална поља (корисничко име и лозинка) и једног дугмета (са акцијом провере података).

Са *UIKit*-ом мора се водити рачуна о свим ситним детаљима као што су: креирање вертикалног скупа елемената, његово додавање у главни поглед, креирање текстуалног поља, додавање текстуалног поља у скуп елемената, додавање аутоматског ограничења распореда како би се центрирало текстуално поље, понављање поступка за друго текстуално поље и поновно понављање поступка за дугме.

За разлику од *UIKit*-а, *SwiftUI* се базира на декларативном начину програмирања и коришћењем радног окружења *SwiftUI* је довољно навести груписање два текстуална поља и дугмета у вертиклани скуп елемената и у ком погледу ће се приказати. Све ситне детаље ће радно окружење одрадити само, онако како је то уобичајено (енг. *default*) дефинисано. Наравно, програмер по потреби може и сам променити ове детаље.

Креирање корисничког интерфејса у *UIKit*-у коришћењем само *Swift* кода је веома компликовано, и за веће пројекте готово немогуће. Најчешћи начин израде корисничког интерфејса је коришћењем *Storyboards-a* і *Interface Builder-a*, помоћу којих програмер креира кориснички интерфејс превлачењем, спуштањем и конфигурацијом графичких елемената. У *SwiftUI*-у се кориснички интерфејс изграђује помоћу *Swift* кода. Корисник декларише шта ће бити креирано и радно окружење то уради. Да би процес креирања био бржи и приступачнији, од верзије *Xcode*-а 11, која је изашла у исто време када је представљен *SwiftUI*, постоји могућност прегледа уживо сваког појединачног погледа који је креиран или скупа више погледа одједном. О овоме ће бити више речи у поглављу 2.6 — Преглед уживо — Xcode.

Уколико се сагледају архитектуре образаца, може се приметити да се *UIKit*

првенствено базира на *MVC* обрасцу, док *SwiftUI* користи *MVVM*[11] образац. За заинтересоване читаоце, постоји могућност комбиновања ова два радна окружења и коришћење *SwiftUI*-а унутар *UIKit* кода, или обратно.

Недостаци радног окружења *SwiftUI*

Иако веома моћно радно окружење, сама технологија која га подржава је и даље млада и у активном развоју, па се може наићи на одређене потешкоће приликом употребе *SwiftUI*-а, као што су:

- Проблеми који настају приликом употребе неког од основних елемената, као што је навигација
- Минимална верзија оперативног система мора бити *iOS13*, која је стара свега две године
- Постоје функционалности које су и даље доступне само у *UIKit*-у
- Поједини проблеми могу бити тешки за превазилажење, обзиром да се мали број програмера сусрео са њима
- Заједница још увек није дефинисала смернице за израду сложених корисничких интерфејса
- Радно окружење тренутно пружа мали скуп компоненти корисничког интерфејса

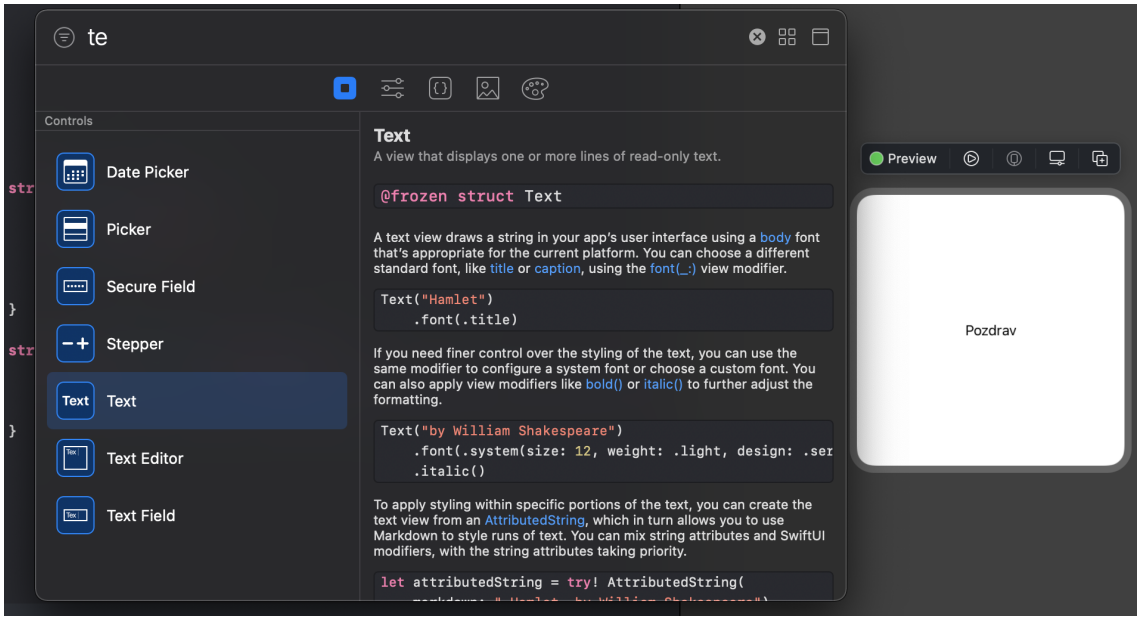
Преглед уживо — Xcode

Са представљањем *SwiftUI*-а, *Apple* је представио и нову верзију њиховог ИРО-а *Xcode11*, у коме је додато својство рада у новом радном окружењу као и могућност прегледа уживо сваког погледа. Предност оваквог начина писања кода је у могућности брзог визуелног прегледа измена без потребе поновне изградње (енг. *rebuilding*) апликације. Ова предност највише долази до изражаја када се ради на додавању или измени погледа који се налази дубоко унутар навигације апликације и за који је потребно више кликова

[11] Архитектурни образац Модел-Поглед-Модел погледа (енг. *Model-View-ViewModel*) који се заснива на подели на три целине, модел — структура података, поглед — приказ података у корисничком окружењу, модел погледа — стање података у моделу.

и/или превлачења (приликом тестирања измена кода) да би се до њега дошло.

Преглед уживо помаже да се у *SwiftUI*-у користи метод превлачења и пуштања (енг. *drag and drop*) за креирање корисничког интерфејса, тако што се сваки елемент превлачи у део где се пише кôд и када се испусти, тај елемент постаје део кода. Избор графичких елемената може се видети на слици 2.7, док је кôд програма и приказ уживо након испуштања графичког елемента *Text* приказан на слици 2.8.



Слика 2.7: *Приказ графичких елемената*

Слика 2.8: *Кôд програма након испуштања елемента*

Да би се омогућило коришћење приказа уживо, инстанца жељеног погледа се смешта унутар тела структуре која имплементира протокол *PreviewProvider*, а која служи за живи приказ погледа или групе погледа. Пример употребе

структуре која имплементира протокол *PreviewProvider* може се видети у делу 2.43 — *Xcode — преглед уживо*.

```
1
2     struct PlaceholderView : View {
3         var body : some View {
4             Kulinarstvo_widgetEntryView(entry: SimpleEntry(date: Date
                  (), configuration: ConfigurationIntent(), recipe:
                  RecipeModel.testData[0]))
5         }
6     }
7
8     // Struktura u kojoj se konfigurise prikaz uzivo
9     struct Kulinarstvo_widget_Previews: PreviewProvider {
10        static var previews: some View {
11            // Grupisanje vise pogleda
12            Group {
13                // Prikaz malog widget-a sa prvim elementom iz liste
14                Kulinarstvo_widgetEntryView(entry: SimpleEntry(date:
                      Date(), configuration: ConfigurationIntent(),
                      recipe: RecipeModel.testData[0]))
15                    .previewContext(WidgetPreviewContext(family: .
                          systemSmall))
16
17                // Prikaz srednjeg widget-a sa skrivenim sadrzajem
18                PlaceholderView()
19                    .previewContext(WidgetPreviewContext(family: .
                          systemMedium))
20                    .redacted(reason: .placeholder)
21            }
22        }
23    }
```

Пример кода 2.43: *Xcode — преглед уживо*

Након сваке измене која се направи у коду који је везан за поглед(е) који се налази у прегледу уживо, *Xcode* ће изнова направити нову верзију и покренути је у прозору за преглед уживо. Преглед уживо не мора приказивати само један поглед, већ се могу груписати различити погледи и сви бити приказани одједном. Предност оваквог приступа је могућност истовременог прегледа старог и новог изгледа погледа, више величина виџета, истих погледа са светлом и тамном бојом позадине, погледа на различитим језицима...

Више на тему прегледа погледа уживо може се наћи у оквиру предавања *Mastering Xcode Previews* [7] и *Structure your app for SwiftUI previews* [8] са *Apple*-ове конференције за програмере из 2019. и 2020. године респективно.

Глава 3

Улога и развој виџета

Виџет на уређајима са платформом компаније *Apple* приказује крајњим корисницима изабрани кључни део апликације. Виџет је за сада једини део апликације који у потпуности мора бити развијен коришћењем окружења *SwiftUI*.

3.1 Основне карактеристике виџета

Виџет се приказује крајњим корисницима тамо где ће се најлакше уочити, на *iPhone*-у и *iPad*-у се може налазити на почетном екрану или у делу *Today View*-а, док се на *Mac* уређајима налази у центру за нотификације. Величина виџета није флексибилна као на *Android* уређајима, па тако постоји могућност креирања малих, средњих и великих, а од верзије оперативног система *iPadOS15* екстра великих (само за *iPad* уређаје) виџета. Величина малог виџета је 2×2 места на почетном екрану *iPhone*-а, величина средњег је 2×4, велики виџет заузима 4×4 места на екрану, док је величина екстра великих виџета 4×8.

Скуп свих тренутно доступних виџета на уређају налази се у галерији виџета (енг. *widget gallery*), која помаже корисницима приликом одабира конкретне величине и типа виџета (једна апликација може имати више типова виџета). Режим за измену виџета унутар галерије омогућава корисницима да контролишу и мењају своје виџете и тиме их прилагоде себи. Измена виџета ће бити доступна корисницима уколико је она омогућена од стране програмера приликом креирања виџета. Више речи о овоме биће у делу 3.2 — Развој виџета.

На оперативним системима *iOS* и *iPadOS* галерија има могућност додавања паметних гомила (енг. *smart stack*), које могу садржати до десет различитих виџета исте величине. Паметна гомила у једном тренутку приказује један од виџета који се налазе у њој. Корисник може сам да мења који ће виџет бити приказан једноставним померањем (енг. *scrolling*). Временом, паметна гомила може научити који виџет корисник ставља на почетак гомиле у току дана (или недеље) и сама мењати примарне виџете у одређеном тренутку (на пример, након гашења аларма прво се приказује виџет са временском прогнозом, па најновије вести, стање у саобраћају).

Siri асистент[1] може и сам додати виџете у паметну гомилу уколико претпостави да постоји неки виџет који би кориснику био користан. Након тога корисник сам одлучује да ли жели да новододати виџет остане у паметној гомили или не.

Радно окружење *WidgetKit*

WidgetKit је радно окружење које уз *widget API* из *SwiftUI*-а служи за израду виџета, од његовог изгледа, преко временског ажурирања па све до омогућавања конфигурације виџета од стране крајњих корисника и управљања паметном гомилом приликом ротације виџета од стране система. Још једна могућност коју ово радно окружење пружа је повезивање апликације и самог виџета, што омогућава кориснику да отвори апликацију притиском на виџет и аутоматски оде на одговарајући поглед из виџета када жели да види детаљније податке. Програмери морају обратити пажњу код оваквог начина комуникације, виџет не би смео да служи само као пречица за покретање апликације јер то може довести до престанка употребе виџета од стране корисника. Лакше је додати саму апликацију на почетни екран која истовремено заузима мање простора него виџет (више о томе биће објашњено у делу 3.3 — Дизајн виџета).

3.2 Развој виџета

Виџет је *SwiftUI* поглед. Виџети су тренутно једини део апликација развијених за платформе компаније *Apple* који у потпуности морају бити написани

[1]Интелигентни лични асистент на уређајима са платформом компаније *Apple*.

коришћењем радног окружења *SwiftUI*. *Apple* је од почетка развоја виџета имао на уму овакву идеју због начина њиховог приказивања, повременог ажурирања података, као и немогућности корисничке интеракције са самим виџетима (осим једноставног клика којим се отвара одређени део апликације).

Додавање виџет додатка апликацији

Додатак апликације (енг. *app extension*) је проширење апликације[2] које пружа додатне функционалности и садржај апликације крајњим корисницима у тренуцима када је не користе. Неки од додатака апликације су: додатак паметног сата (енг. *watch extension*) који омогућава креирање додатка апликације намењеног паметним сатовима компаније *Apple*, додатак центра за обавештења (енг. *notification center extension*) који омогућава прилагођавање изгледа обавештења, виџет додатак (енг. *widget extension*) који омогућава креирање виџета за конкретну апликацију.

Шаблон за виџет додатак креира основне компоненте потребне за његову израду. Унутар овог додатка корисник креира све потребне виџете за своју апликацију, независно од њиховог броја и величине. У одређеним ситуацијама различити типови виџета могу бити одвојени у посебним додацима. Ово се најчешће односи када један тип виџета захтева одређене дозволе од стране корисника, док за други тип оне нису потребне (на пример, приступ тренутној локацији корисника).

Кораци за креирање виџет додатка:

1. Отворити пројекат у *Xcode*-у и изабрати *File -> New -> Target*
2. Из групе *Application Extension*, изабрати *Widget Extension* и кликнути *Next*
3. Унети име додатка и изабрати тим (који може чинити и једна особа) који ради на пројекту
4. Уколико виџет подржава конфигурацију од стране корисника, штиклирати поље *Include Configuration Intent*
5. Кликнути на дугме *Finish*

[2]Обично се састоји од пар фајлова које ИРО аутоматски генерише и групише у посебан фолдер. Унутар фајлова је имплементиран одговарајући шаблон, који корисник допуњује или мења како би додатак одговарао апликацији.

Додавање детаља конфигурације

Шаблон виџет додатка пружа иницијалну поставку програма виџет додатка који имплементира *Widget* протокол. Два могућа начина конфигурације виџета су статичка (*StaticConfiguration*) и конфигурација са сврхом (*IntentConfiguration*).

Статичка конфигурација се користи за виџете који немају параметре који могу бити конфигурисани од стране корисника. Пример виџета који нема конфигурабилне параметре је виџет системске апликације *Screen time* који води статистику о активном времену проведеном на одговарајућем уређају. Конфигурација са сврхом се користи за виџете чији одређени параметри могу бити конфигурисани од стране корисника. Пример виџета са конфигурабилним параметрима је виџет системске апликације за временску прогнозу, где корисник може изабрати одређени град за који жели да добија податке. Ова конфигурација ће бити укључена, и конфигурациони фајл ће бити додат уколико програмер приликом додавања виџет додатка штиклира поље *Include Configuration Intent*.

Да би програмер спровео почетну конфигурацију виџета потребно је да проследи следеће параметре:

Тип (енг. *kind*), стринг који идентификује виџет, требао би да казује шта виџет представља.

Снабдевач (енг. *provider*), објекат класе која имплементира протокол *TimelineProvider* и кроз временску линију коју производи одређује у ком тренутку ће виџет бити поново изрендерован и нови подаци бити приказани. Детаљније објашњење начина функционисања временске линије дато је у делу 3.2 — Временска линија.

Затворење садржаја (енг. *content closure*), затворење које садржи *SwiftUI* поглед и које *WidgetKit* позива када дође време за поновно рендеровање садржаја виџета.

Прилагођена сврха (енг. *custom intent*), фајл који дефинише параметре које корисник може мењати и прилагођавати себи. Израда овог фајла и његова конкретна употреба приликом израде виџета приказана је у делу 3.2 — Конфигурација са сврхом.

Пример почетне статичке конфигурације виџета може се видети у коду 3.1 — *Виџет — почетна конфигурација*. Да би се подесила боја шеме прослеђује се променљива *colorScheme* којом се боја виџета усклађује са системском бојом уређаја. Променљива *kind* служи за јединствену идентификацију типа виџета.

```
1   struct KulinarstvoRandomRecipesWidget: Widget {
2       @Environment(\.colorScheme) var colorScheme
3
4       let kind: String = "
            KulinarstvoSlasnoIEfikasnoRandomRecipesWidget"
5
6       var body: some WidgetConfiguration {
7           StaticConfiguration(kind: kind, provider:
                RandomRecipesProvider()) { entry in
8               KulinarstvoRandomRecipesWidgetEntryView(entry: entry)
9           }
10          .configurationDisplayName("Recept na klik")
11          .description("Dodaj svoje omiljene recepte na pocetni
                ekran")
12          .supportedFamilies([.systemLarge])
13      }
14  }
```

Пример кода 3.1: *Виџет — почетна конфигурација*

Временска линија

Снабдевач временске линије генерише временску линију која се састоји од уноса (енг. *entries*), а сваки унос садржи датум и време када је потребно ажурирати садржај виџета. Када се датум и време из уноса подударе са реалним временом, *WidgetKit* позива затворење садржаја које потом приказује ажуриране податке.

Да би виџет био приказан у виџет галерији, *WidgetKit* захтева од снабдевача преглед снимка (енг. *preview snapshot*). Дохватање прегледа снимка се разрешава коришћењем променљиве *isPreview* којом се проверава да ли снабдевач прегледа снимка шаље тренутни снимак за приказ у галерији или за приказ виџета на почетном екрану. Када је параметар *isPreview* тачан, виџет се приказује у галерији. Уколико унутар виџета треба да буду приказани и одређени подаци који још увек нису учитани, постоје два решења. Могу се приказати подразумевани, унапред одређени подаци, или се могу користити

подаци који чувају место правим подацима (енг. *placeholder*). У примеру 3.2 — *Виџет — placeholder* може се видети креирање погледа чувара места (енг. *placeholder view*) коришћењем статичких података који су увек доступни. Такође је приказана и имплементација чувара места у прегледу уживо са сакривеним подацима (енг. *redacted data*) — приказ како ће корисник видети виџет док не пристигну конкретни подаци у одређеним ситуацијама.

```
1   struct PlaceholderView : View {
2       var body : some View {
3           Kulinarstvo_widgetEntryView(
4               entry: SimpleEntry(date: Date(), configuration:
                    ConfigurationIntent(), recipe: Datafeed.shared.
                    favRecipes[0], parameterToShow: MainParameter.
                    Sastojci.rawValue))
5       }
6   }
7
8   struct Kulinarstvo_widget_Previews: PreviewProvider {
9       static var previews: some View {
10           PlaceholderView()
11              .previewContext(WidgetPreviewContext(family: .
                    systemLarge))
12              .redacted(reason: .placeholder)
13      }
14  }
```

Пример кода 3.2: *Виџет — placeholder*

Када се подаци учитају, снабдевач добија обавештење, сакупља учитане податке и приказује виџет са њима. Након што корисник дода виџет на почетни екран и буде приказан иницијални снимак изгледа виџета, *WidgetKit* позива функцију *getTimeline* из провајдера, чиме захтева временску линију.

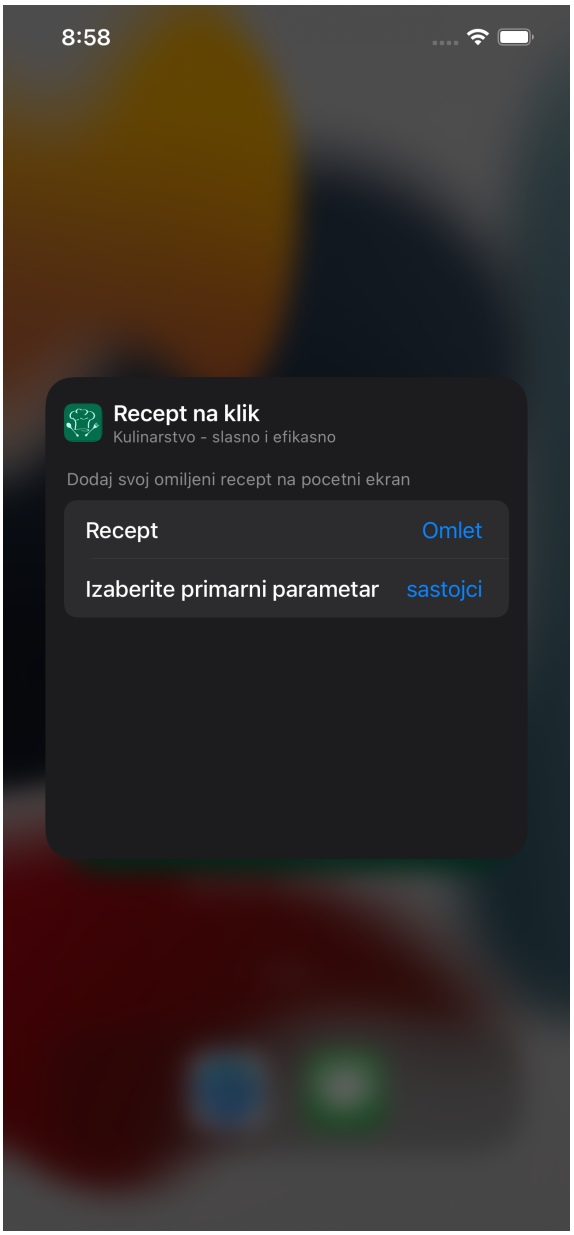
Конфигурација са сврхом

Виџети представљају погледе који не интерагују са корисницима, односно не подржавају интерактивне елементе. Једина интеракција корисника са виџетом постиже се омогућавањем конфигурације виџета од стране корисника употребом фајла *Intent*. У њему се наводе сви параметри које корисник може да промени (као и дозвољене вредности за те параметре).

Да би се додали параметри које корисник може да конфигурише постоје предуслови који се морају испунити:

- додавање дефиниције *Intent*-а који дефинише конфигурабилне параметре,
- структура 'снабдевач временске линије' имплементира протокол *IntentTimelineProvider* уместо протокола *TimelineProvider*, да би конфигурација параметара од стране корисника била сачувана у уносима временске линије,
- уколико параметри зависе од динамичких података потребно је имплементирати екстензију *Intent*-а.

На слици 3.1 се може видети изглед конфигурације виџета који има два параметра, први служи за избор рецепта који ће у виџету бити приказан, а други за избор примарног параметра који ће бити приказан уз опис рецепта (састојци или припрема).



Слика 3.1: *Конфигурација виџета*

Везе унутар виџета

Једини начин директне комуникације између корисника и виџета остварена је везама (енг. *links*) унутар виџета. Када корисник кликне на виџет отвара се апликација којој тај виџет припада, а програмер може конфигурисати који део апликације ће бити приказан кориснику у зависности од елемента унутар виџета на који је кликнуо.

За све величине вицета, осим малих, може се користити веза (енг. *link*) која се додаје једном погледу унутар вицета. Том везом је одређено тачно место у апликацији које ће бити отворено уколико корисник кликне на одговарајући поглед (на пример, виџет који садржи листу са три рецепата и сваки елемент листе има везу која води ка детаљној страни о том рецепту. Када корисник кликне на одговарајући рецепт отвориће се апликација и приказати страницу са детаљним описом рецепта на који је корисник кликнуо). Све величине вицета могу користити модификатор *widgetURL(_:)*. Овај модификатор ће бити активиран уколико корисник кликне на поглед унутар вицета који нема дефинисану везу и биће отворена апликација, уз могућу додатну навигацију уколико је дефинисана приликом додавања модификатора. У примеру кода 3.3 — *Везе унутар вицета* приказана је употреба модификатора *widgetURL(_:)* за мале вицете, као и употреба веза за средње и велике вицете.

```
1     struct Kulinarstvo_widgetEntryView : View {
2         var entry: Provider.Entry
3         @Environment(\.widgetFamily) var widgetFamily
4
5         @ViewBuilder
6         var body: some View {
7             switch widgetFamily {
8             case .systemSmall:
9                 ImageRecipeView(recipe: entry.recipe, isSmallView:
                      true)
10                     .widgetURL(entry.recipe.url)
11            case .systemMedium:
12                Link(destination: entry.recipe.url ?? URL(
                      fileURLWithPath: "")) {
13                    RecipeMediumView(recipe: entry.recipe, listName:
                          entry.parameterToShow)
14                }
15            case .systemLarge:
16                Link(destination: entry.recipe.url ?? URL(
                      fileURLWithPath: "")) {
17                    RecipeLargeView(recipe: entry.recipe,
                          mainParameter: entry.parameterToShow)
18                }
19            default:
20                Text("")
```

```
21              }
22          }
23      }
```

Пример кода 3.3: *Везе унутар виџета*

Више типова виџета у једном виџет додатку

Уколико постоји потреба за коришћењем више различитих типова виџета у једном виџет додатку, то се може постићи уз пар измена главне структуре виџет додатка (структура означена анотацијом *@main*). Уместо протокола *Widget*, главна структура виџет додатка мора имплементирати протокол *WidgetBundle*. Тело структуре треба да имплементира протокол *Widget* и да буде означен анотацијом *@WidgetBundleBuilder*. Приказ употребе више типова виџета у једном виџет додатку може се видети у примеру 3.4 — *Више виџета у једном виџет додатку*.

```
1   @main
2   struct ReceptiWidgets: WidgetBundle {=
3       @WidgetBundleBuilder
4       var body: some Widget {
5           DetaljanPrikazReceptaWidget()
6           ListaRecepataWidget()
7           SpisakZaKupovinuWidget()
8       }
9   }
```

Пример кода 3.4: *Више виџета у једном виџет додатку*

3.3 Дизајн виџета

Главна улога виџета је приказивање садржаја који кориснику пружа корисне информације без покретања апликације. Самим тим подаци морају бити тачни и релевантни за корисника, сам виџет би требао бити конфигурабилан како би кориснику допустио одређену врсту слободе приликом коришћења и дизајниран тако да одговара апликацији којој припада.

Фокус виџета

Подаци које виџет приказује треба да буду минималистички, да одговарају величини виџета (већа величина треба да повлачи и већу количину података) и да буду временски и кориснички релевантни. Први корак у дизајну виџета је избор дела апликације који ће тај виџет представљати.

Свака величина виџета која је омогућена за додавање из галерије треба да садржи одређену количину информација која је пропорционална тој величини. Не сме се дозволити да неколико величина виџета приказују исте податке, али истовремено приликом додавања нових података се мора водити рачуна о почетној идеји, односно делу апликације које тај виџет треба да представља. Уколико не постоји довољна количина података за веће виџете, те величине виџета могу бити искључене из понуде кориснику.

Виџет не би смео да служи само као пречица за покретање апликације. Корисници очекују од сваког виџета да им покаже корисне информације, у супротном неће наићи на добар одзив и истовремено може бити штетно самој апликацији (мањи број корисника, лошија оцена у продавници).

Ажурни подаци

Да би виџети могли да пружају корисне и прецизне информације подаци које приказују морају бити ажурирани. Виџети не подржавају ажурирање података у реалном времену, а и сам оперативни систем може ограничити ажурирање података које виџет приказује у завиности од корисничког понашања и интеракције са њим, па се мора пронаћи начин на који ће подаци у виџету увек бити релевантни.

Потребно је пронаћи оптимално време за ажурирање података у виџету, узимајући у обзир колико се сами подаци које виџет приказује често мењају. Једна корисна информација која се може приказати уз временски зависне податке је поље које ће представљати датум и време када су подаци последњи пут ажурирани. За одређене податке се може искористити помоћ система за одређивање датума и времена (на пример, виџет који приказује време у које ће се огласити аларм, истовремено може приказивати и ажуран податак о томе колико је времена остало до оглашавања аларма).

Конфигурабилност и интеракција

У већини случајева виџет треба да омогући кориснику конфигурабилност како би могао да пружи релевантне информације (на пример, књига коју корисник тренутно чита у апликацији *Apple Books*), док поједини виџети не морају бити конфигурабилни (на пример, најновије вести). Уколико је виџет конфигурабилан, потребно је да подешавања буду једноставна и да се не захтева превише информација од корисника. Кориснички интерфејс за измену виџета је унапред одређен и исти за све виџете, као што је показано у делу 3.2 — Конфигурација са сврхом.

Дизајн прилагођен свима

Виџети треба да буду јарких боја како би се истицали на екрану, али треба да садрже јасно видљив текст како би корисник могао да види све потребне информације након откључавања или непосредно пре закључавања уређаја. Виџет треба прилагодити апликацији коју представља (боје, фонт текста, јединствени елементи...), док истовремено не треба истицати превише елемената који ће указивати на апликацију (лого, име) јер се тиме само непотребно заузима простор унутар виџета који се може боље искористити.

Количина информација која ће бити приказана у виџету мора бити оптимална. Уколико се прикаже премало информација виџет неће имати превелики значај за кориснике, док превише информација на малом простору отежава читање и разумевање података.

Израда дизајна виџета за обе врсте боја системске позадине (светле и тамне) је веома важно како би виџет био прихваћен од стране свих корисника. Дизајн виџета се не сме разликовати од системске боје позадине јер ниједан корисник не жели видети таман текст на светлој боји позадине уколико је изабрао тамну системску боју позадине. Приликом израде обе врсте дизајна може помоћи *Xcode preview* који омогућава истовремено сагледавање оба дизајна, упоређивање и исправљање евентуалних недостатака.

Apple саветује да се никад не користи фонт текста мањи од 11 поена[3]. Коришћење мањег фонта би корисницима знатно отежало употребу виџета. Увек треба користити званичне погледе (погледи дефинисани унутар рад-

[3] *Apple*-ов израз за „број који треба уписати у поље”, универзална мера у дизајну на платформама компаније *Apple*.

ног окружења *SwiftUI*) за приказ текста, како би се омогућила скалабилност текста, као и системско читање текста (читање текста употребом виртуелног асистента).

Пажњу треба обратити на дизајн прегледа виџета унутар галерије, за све типове и величине који виџет подржава, као и приказ чувара места уместо реалних података уколико они нису пристигли на време са сервера и не постоје подразумевани подаци. Уколико се исти елементи налазе у апликацији и истовремено на виџету потребно је да имају исту функционалност јер би у супротном корисници били збуњени.

Потребно је искористити могућност приказа описа виџета у галерији и саставити кратак и јасан опис функционалности виџета. Груписање свих величина једног типа виџета са јединственим описом је погодно корисницима апликације, пре свега због једноставности разумевања коришћења виџета.

Глава 4

Имплементација и визуелни приказ апликације

У овом поглављу биће представљена апликација „Кулинарство — сласно и ефикасно" која прати мастер рад. Апликација треба да помогне корисницима приликом избора и припреме оброка, док виџет који је имплементиран у склопу апликације може послужити као подсетник за потребне састојке или кораке припреме и истовремено бити употребљен за предлагање наредних оброка. Изворни кôд апликације може се наћи на *GitHub*-у [17].

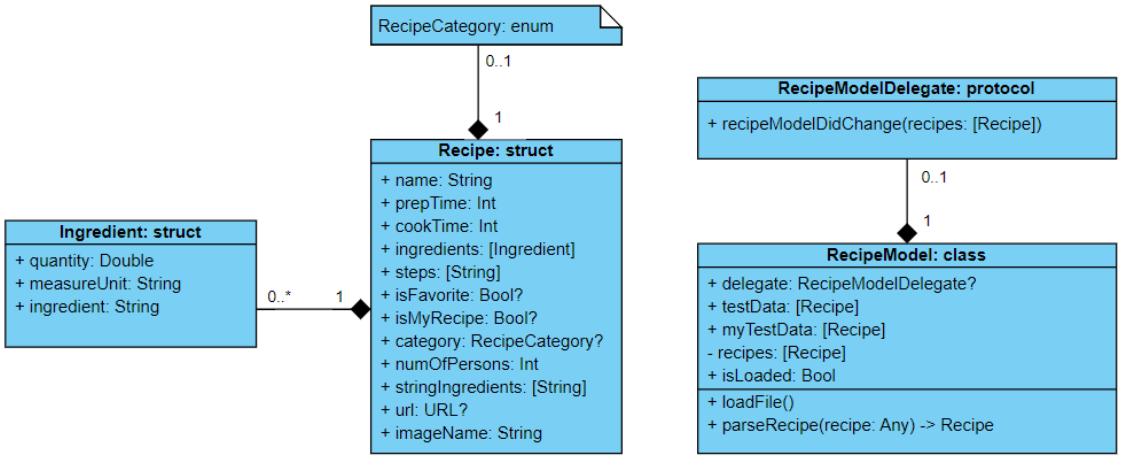
4.1 Карактеристике имплементације

Апликација „Кулинарство — сласно и ефикасно" је развијена у ИРО-у *Xcode*, коришћењем програмског језика *Swift 5.5*. Виџет додатак је креиран употребом радног окружења *SwiftUI*.

Имплементација архитектуре *MVC*

Као архитектурални образац коришћена је архитектура *MVC*, чиме је структура изворног кода подељена у три целине (модел, поглед и контролер). Унутар модела су дефинисане три класе, *Ingredient* којом су описани састојци рецепта, *Recipe* унутар које су дефинисана поља једног рецепта и *RecipeModel* у којој су дефинисане методе које учитавају и парсирају податке

о тренутно доступним рецептима из *JSON*[1] фајла („*RecipesData.json*"). Када се апликација први пут инсталира, биће учитан подразумевани фајл који долази уз апликацију, док ће сваки наредни пут бити учитан локални фајл који ће садржати све промене које је корисник унео (нове рецепте, омиљене рецепте). Дефиниција ових класа је приказана на слици 4.1. Преостале класе које сачињавају модел су *Datafeed* која служи за чување низа рецепата у зависности да ли припадају општем делу, омиљеним рецептима или рецептима које је корисник креирао, и *AppTheme* у којој су дефинисане основне боје апликације за светлу и тамну боју системске позадине.



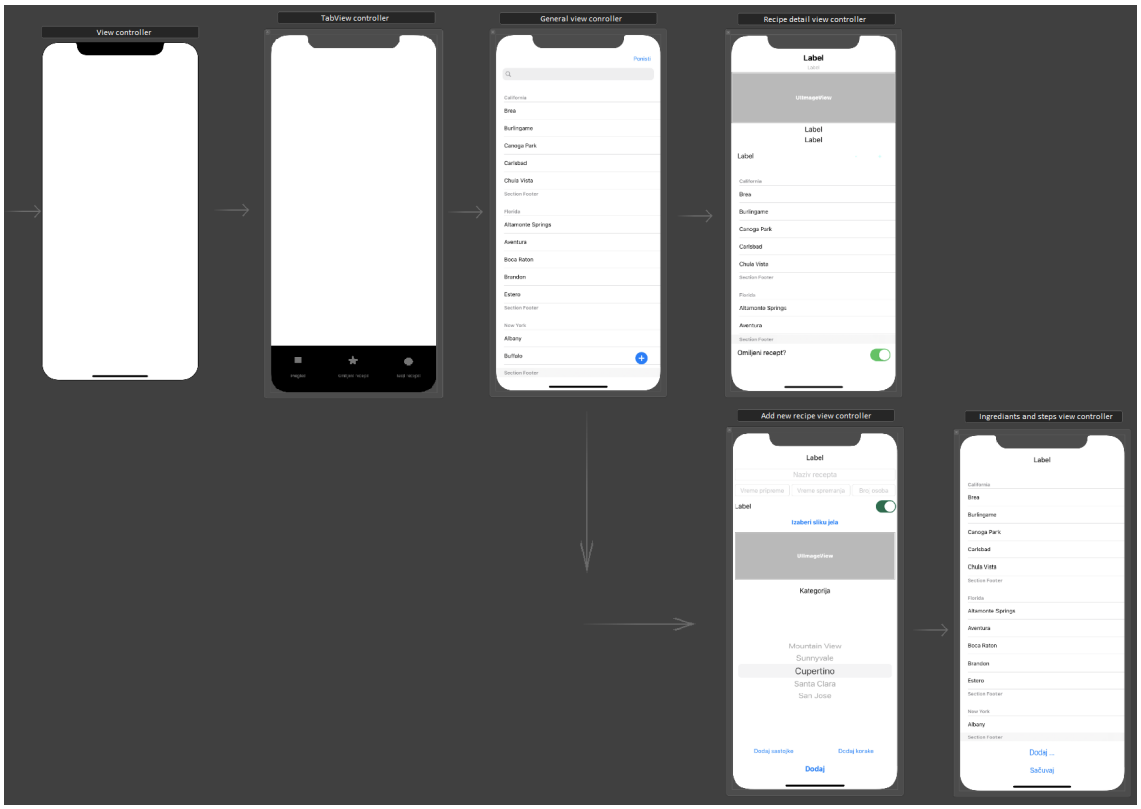
Слика 4.1: *Дијаграм класа модела*

Опис главних и помоћних погледа

Компонента поглед дефинише главне и помоћне погледе који се користе у апликацији. Изглед главних погледа као и њихова повезаност може се видети на слици 4.2. Приликом почетног учитавања апликације приказује се улазни поглед (енг. *entry view*), који је у овом случају празан поглед. Након њега се учитава и приказује главни поглед (*TabBarViewController*) који се састоји из три дела (енг. *tab*). Део са прегледом свих рецепата који се приказује на почетку, део који садржи омиљене рецепте корисника и део са рецептима које је тренутни корисник додао. Уколико корисник кликне на неки од рецепата (у било ком делу) биће му приказан детаљан опис рецепта

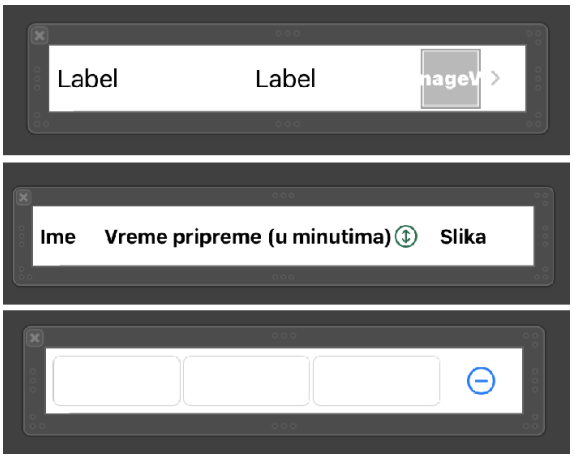
[1] *JSON* (*JavaScript Object Notation*) је текстуални формат дизајниран за размену података који је људима лак за читање и писање, а у програмском коду једноставан за генерисање и парсирање.

(конкретан пример приказа погледа детаљног рецепта може се видети у делу Детаљан приказ рецепта — 4.2). У случају одабира додавања новог рецепта у делу „Моји рецепти" приказаће се поглед *AddNewRecipeViewController* из којег је могуће отворити поглед који приказује додавање састојака и корака припреме рецепта.



Слика 4.2: *Структура и повезаност главних погледа*

Помоћне погледе чине три погледа који дефинишу изглед ћелија (редова у табели). Први поглед дефинише изглед ћелије приказане у главном прозору, која се састоји од две лабеле (име рецепта и време припреме) и слике рецепта. Други поглед приказује изглед заглавља табеле у главном погледу и садржи лабеле за ознаку имена, времена припреме и слике, док се поред лабеле за ознаку времена припреме налази и дугме за сортирање листе рецепата по времену припреме (растуће и опадајуће). Поседњи помоћни поглед дефинише изглед ћелије унутар табеле приликом додавања или измене састојака или корака припреме рецепта од стране корисника. Приказ изгледа ових ћелија може се видети на слици 4.3.



Слика 4.3: *Изглед помоћних погледа*

Имплементација контролера

Сваки главни поглед има упарени контролер који реагује на интеракцију корисника са апликацијом, обрађује догађај и на основу логике која је имплементирана унутар контролера, ажурира модел и/или одговарајући поглед. Главни контролер је *TabBarViewController*, он одређује која инстанца класе *GeneralViewController* ће бити приказана (преглед, омиљени рецепти или моји рецепти). Ове инстанце се разликују на основу два поља *isFavorites* и *isMyRecipes*. Поред ових поља, класа *GeneralViewController* садржи листе са рецептима које се приказују на основу интеракције корисника са апликацијом. Контролер *RecipeDetailViewController* је задужен за поглед који приказује детаљан опис рецепта. Овај контролер омогућава кориснику да повећа или смањи број особа за које ће рецепт бити намењен, чиме ће се аутоматски променити и остали приказани параметри. Количина потребних састојака се мења пропорцијално броју особа, време припреме се мења у односу на фактор времена припреме (променљива која опада или расте логаритамски у односу на тренутни број особа), време кувања се повећава када број особа буде седам и поново када буде петнаест јер се, на пример, време кувања за четири и шест особа не разликује. Унутар погледа корисник може тренутни рецепт додати или избацити из листе омиљених рецепата, док уколико је корисник додао тренутни рецепт, може га изменити или обрисати.

Уколико корисник изабере да дода нови рецепт или измени постојећи, приказаће му се поглед *AddNewRecipeViewController*. Логика која раздваја ова два случаја је имплементирана у контролеру коришћењем променљиве

existingRecipe типа *Recipe?* и у случају додавања новог рецепта има вредност *nil* и сва приказана поља ће бити попуњена чуварима места (енг. *placeholders*), док у случају измене постојећег рецепта променљива *existingRecipe* ће имати вредност инстаце класе *Recipe* чија ће поља садржати информације о тренутном рецепту и оне ће бити приказане приликом измене рецепта. У случају да корисник жели да дода или измени састојке или кораке припреме рецепта, приказаће му се поглед *IngrediantsStepsViewController*. У зависности да ли корисник додаје рецепт или га мења, контролер *IngrediantsStepsView Controller* ће приказати три празна поља за кораке или састојке приликом додавања рецепта, односно тренутне кораке или састојке у случају измене рецепта. Контролер пружа могућност додавања нових корака и састојака, као и брисање или измену постојећих.

Опис имплементације виџета

У пројекту су имплементирана два типа виџета. Први тип је конфигурабилан и омогућава кориснику да промени рецепт који ће бити приказан као и да изабере главни параметар (састојке или кораке припреме), док други тип приказује насумично изабране рецепте из корисникове листе омиљених рецепата.

Први тип виџета имплементира протокол *IntentTimelineProvider*, чиме се омогућава конфигурација виџета од стране корисника. Избор главног параметра је имплементиран унутар фајла *Intent* у којем су кроз набрајање (енг. *enumeration*) одређене могуће вредности параметра (састојци и припрема). Избор рецепта је такође имплементиран кроз *Intent* фајл, а да би се кориснику омогућио динамичан избор рецепта (избор из тренутне листе омиљених рецепата) апликација је проширена додатком *Intent* унутар ког је имплементирана метода која фајлу *Intent* прослеђује корисникову листу омиљених рецепата. Овај тип виџета је имплементиран у све три величине. Мала величина приказује слику и назив рецепта, средња величина додатно приказује састојке или кораке припреме у зависности који је параметар корисник изабрао као примарни. Велики виџет приказује обе листе (састојке и кораке припреме) поред слике и назива рецепта.

Други тип виџета је креиран само у великој величини, имплементира протокол *TimelineProvider* и нема конфигурабилане параметре. Виџет приказује четири различита, насумично изабрана рецепта из корисникове листе омиље-

них рецепата. У случају да корисник има мање од четири рецепта у листи омиљених рецепата, виџет ће приказати подразумевани рецепт.

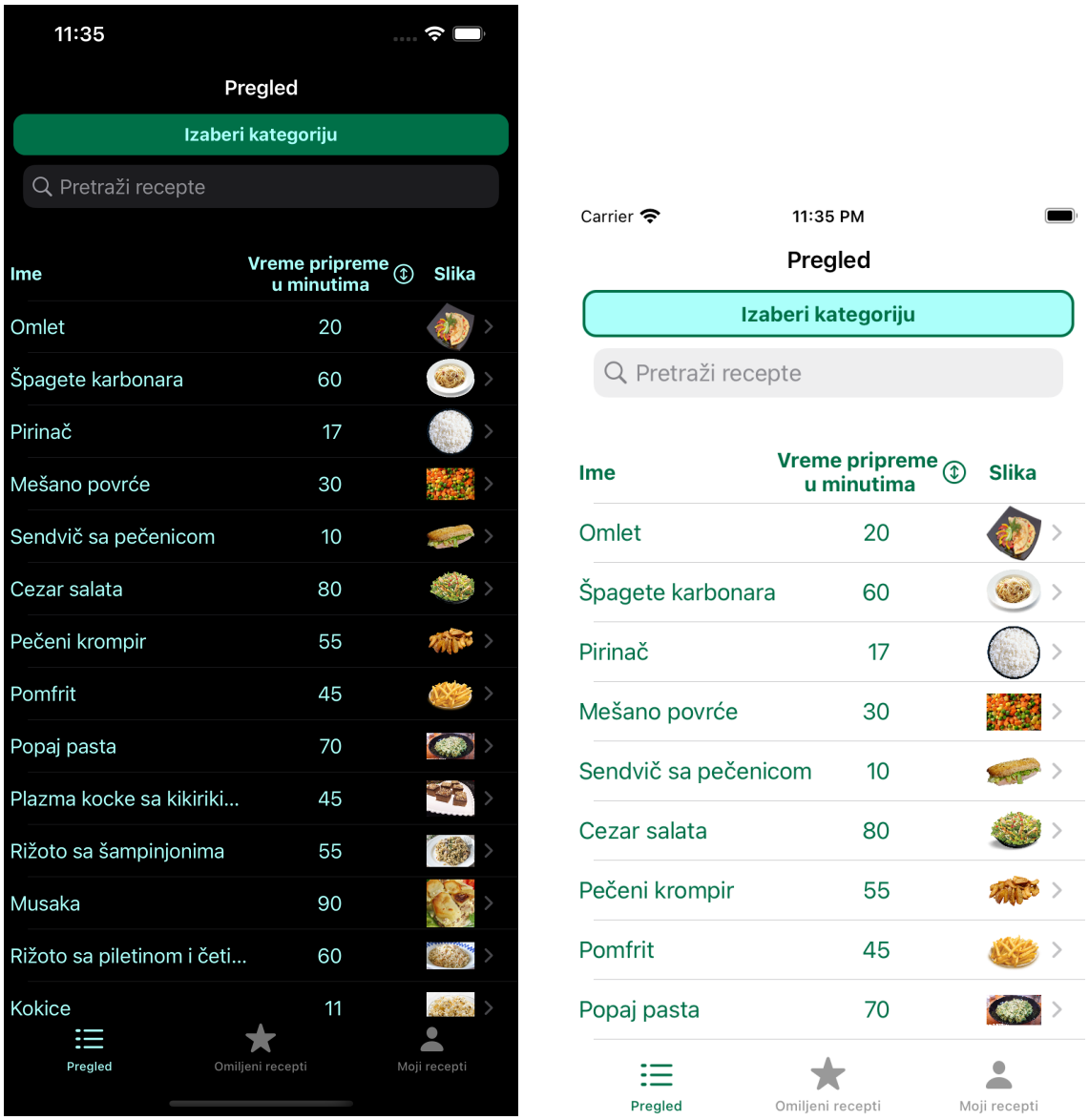
4.2 Визуелни приказ и опис апликације

Рад апликације ће бити приказан упоредо на два симулатора, *iPhone 13 Pro Max* са тамном бојом позадине и *iPhone SE (2nd generation)* са светлом бојом позадине. Оба симулатора покрећу оперативни систем *iOS 15.0*.

Почетни екран

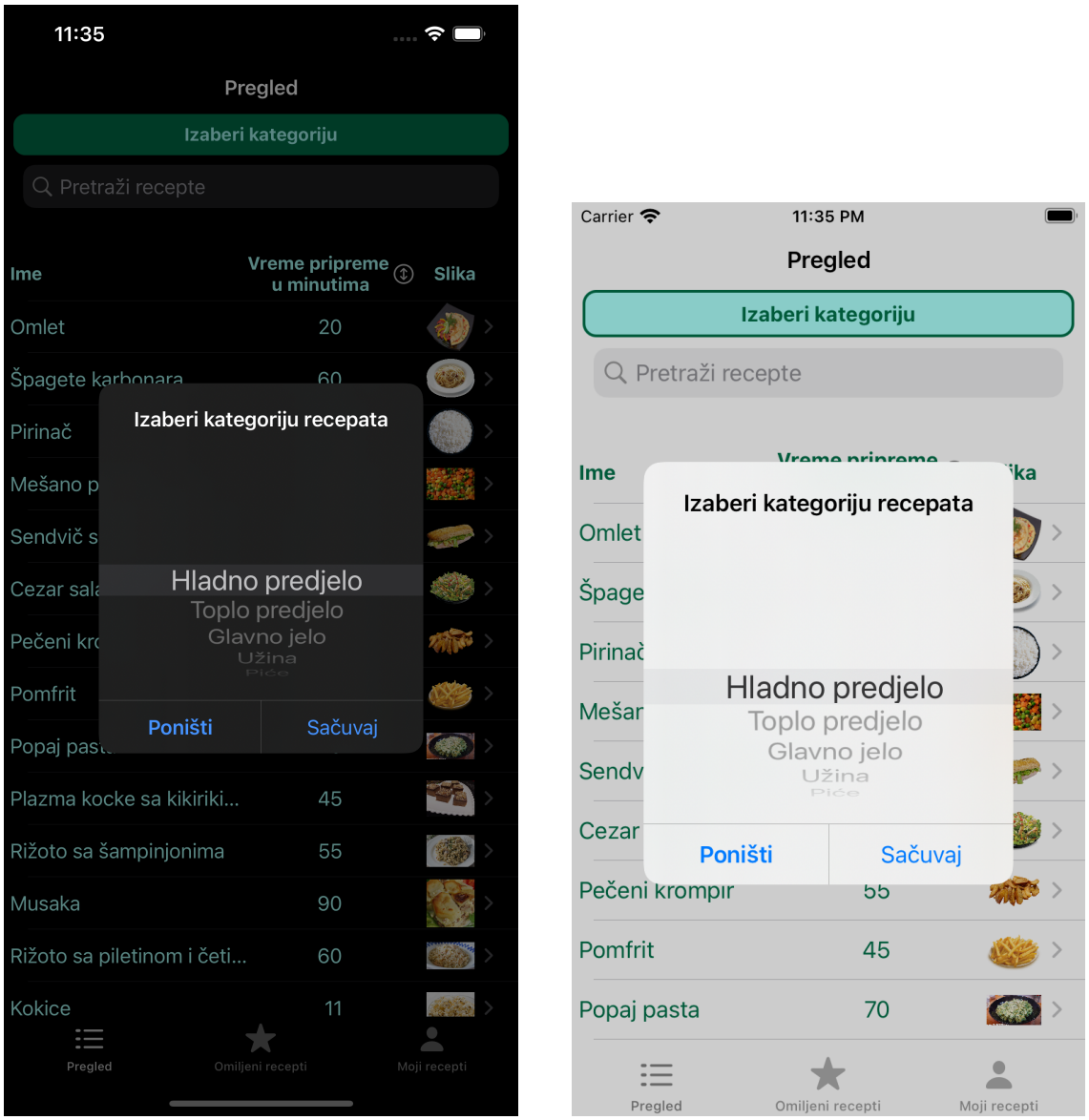
На почетном екрану корисник може видети преглед свих рецепата који су тренутно доступни у апликацији, сортирати рецепте по времену потребном за њихову припрему (растуће и опадајуће), изабрати категорију рецепата који ће бити приказани, претраживати рецепте по именима као и комбиновати ова својства да би у што краћем времену пронашао жељени рецепт (или жељене рецепте).

Рецепти су приказани унутар табеле, сваки рецепт је презентован именом, временом припреме у минутима и сликом. Приказ почетног екрана може се видети на слици 4.4.

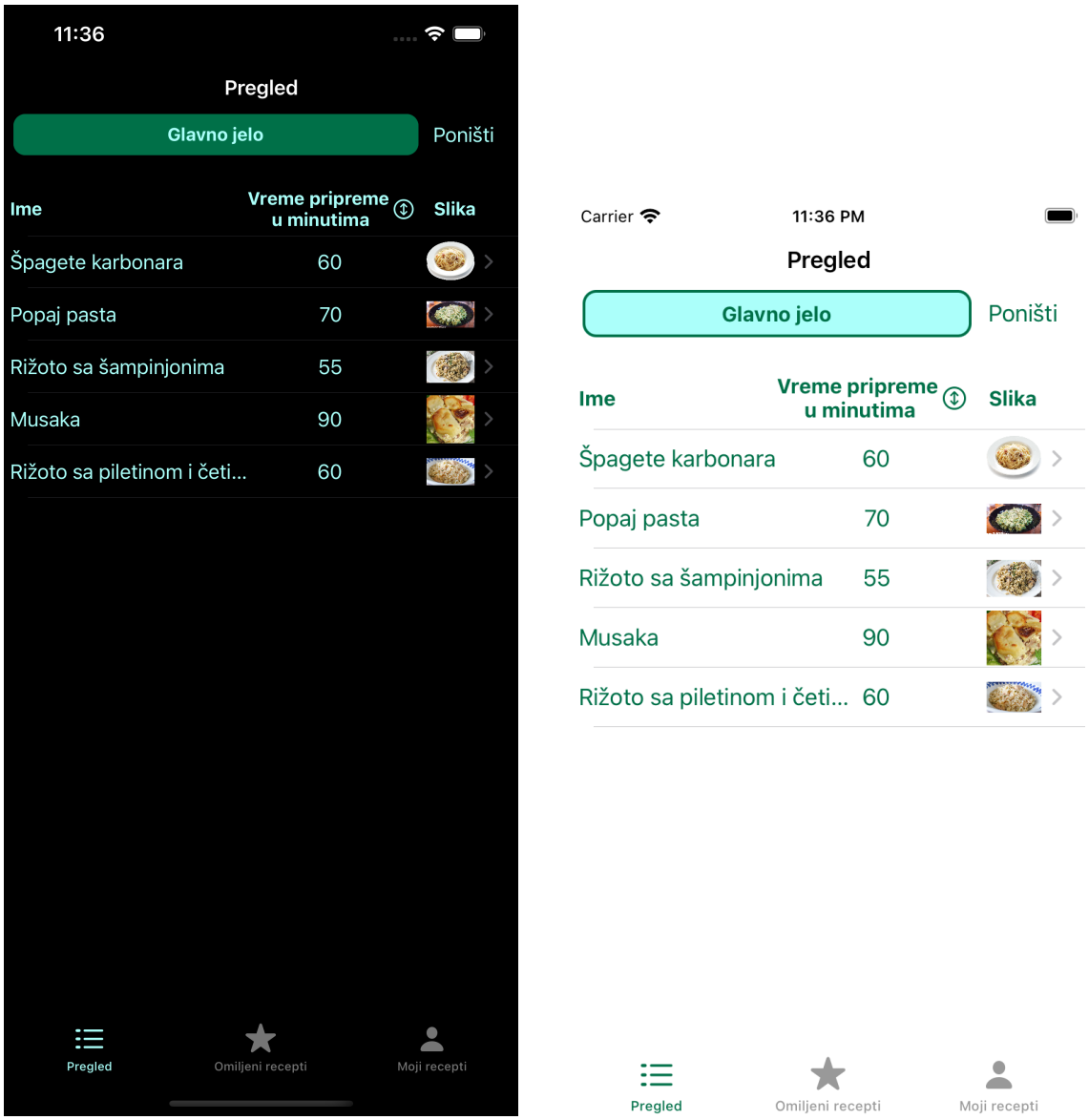


Слика 4.4: *Почетни екран — iPhone 13 (лево) и iPhone SE (десно)*

Корисник може изабрати одређену категорију из које ће му бити приказани рецепти. Међу понуђеним категоријама се налазе „Хладно предјело”, „Топло предјело”, „Главно јело”, „Ужина”, „Пиће”, „Супе и чорбе”, „Дезерт”, „Салата” и „Хлеб”. Приказ избора категорије налази се на слици 4.5, док је на слици 4.6 приказан изглед екрана када је једна категорија изабрана (конкретно категорија „Главно јело”).

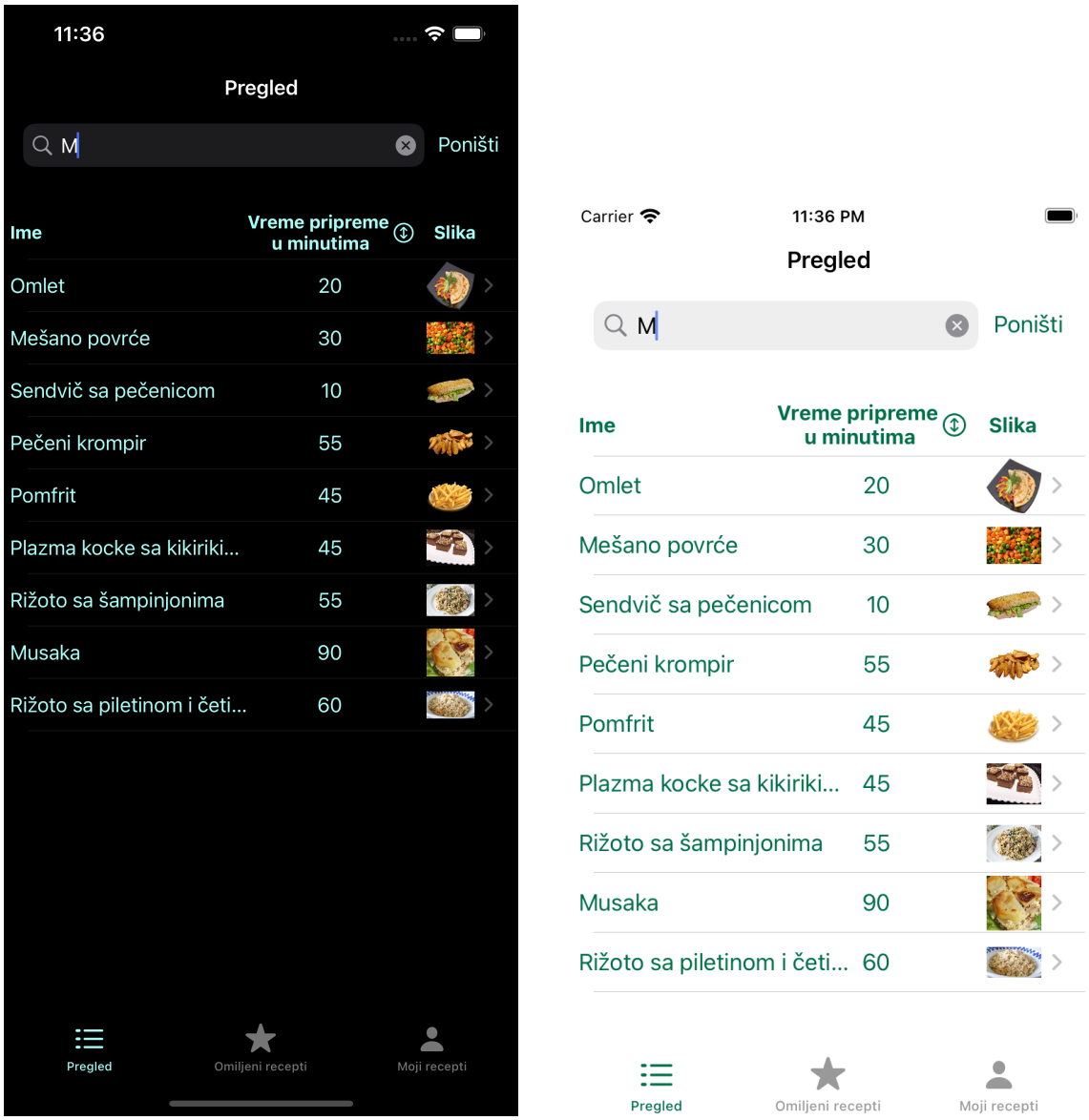


Слика 4.5: *Избор категорије — iPhone 13 (лево) и iPhone SE (десно)*



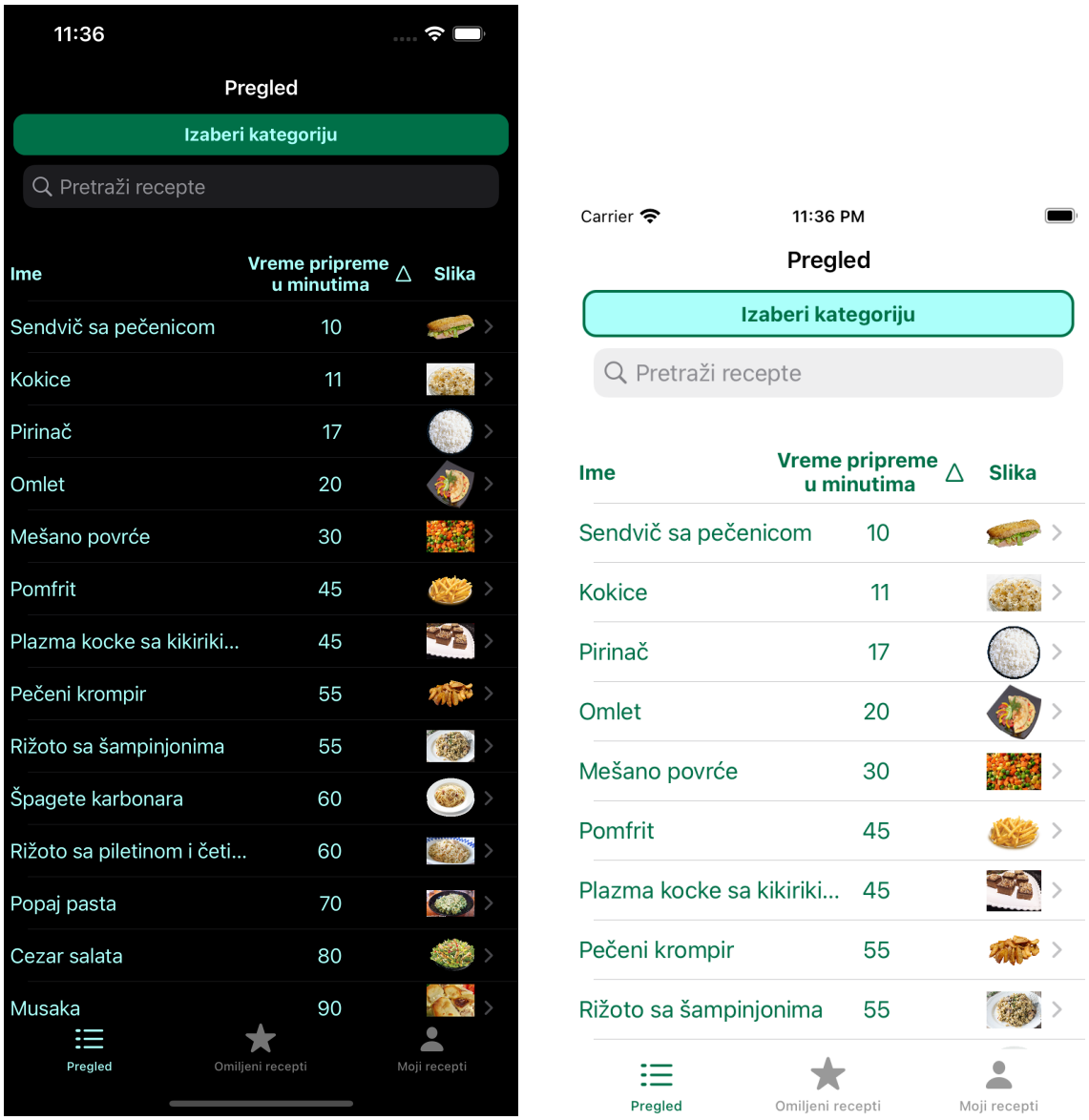
Слика 4.6: *Категорија „Главно јело” — iPhone 13 (лево) и iPhone SE (десно)*

Поред избора категорије, корисник може филтрирати приказане рецепте претрагом по имену и у том случају ће му бити приказани сви рецепти чије име садржи текст који је корисник унео у поље претраге. Почетак претраге приказан је на слици 4.7.



Слика 4.7: *Претрага — iPhone 13 (лево) и iPhone SE (десно)*

Рецепте на почетној страни корисник може и сортирати, опадајуће или растуће по укупном времену потребном за њихову припрему (припрема и спремање). Сортирање се може и комбиновати са избором категорије или претрагом по називу, па тако корисник лако може наћи рецепт из одређене категорије (на пример, из категорије дезерта) за чију припрему је потребно издвојити најмање времена. Сортирани рецепти у растућем редоследу су приказани на слици 4.8.



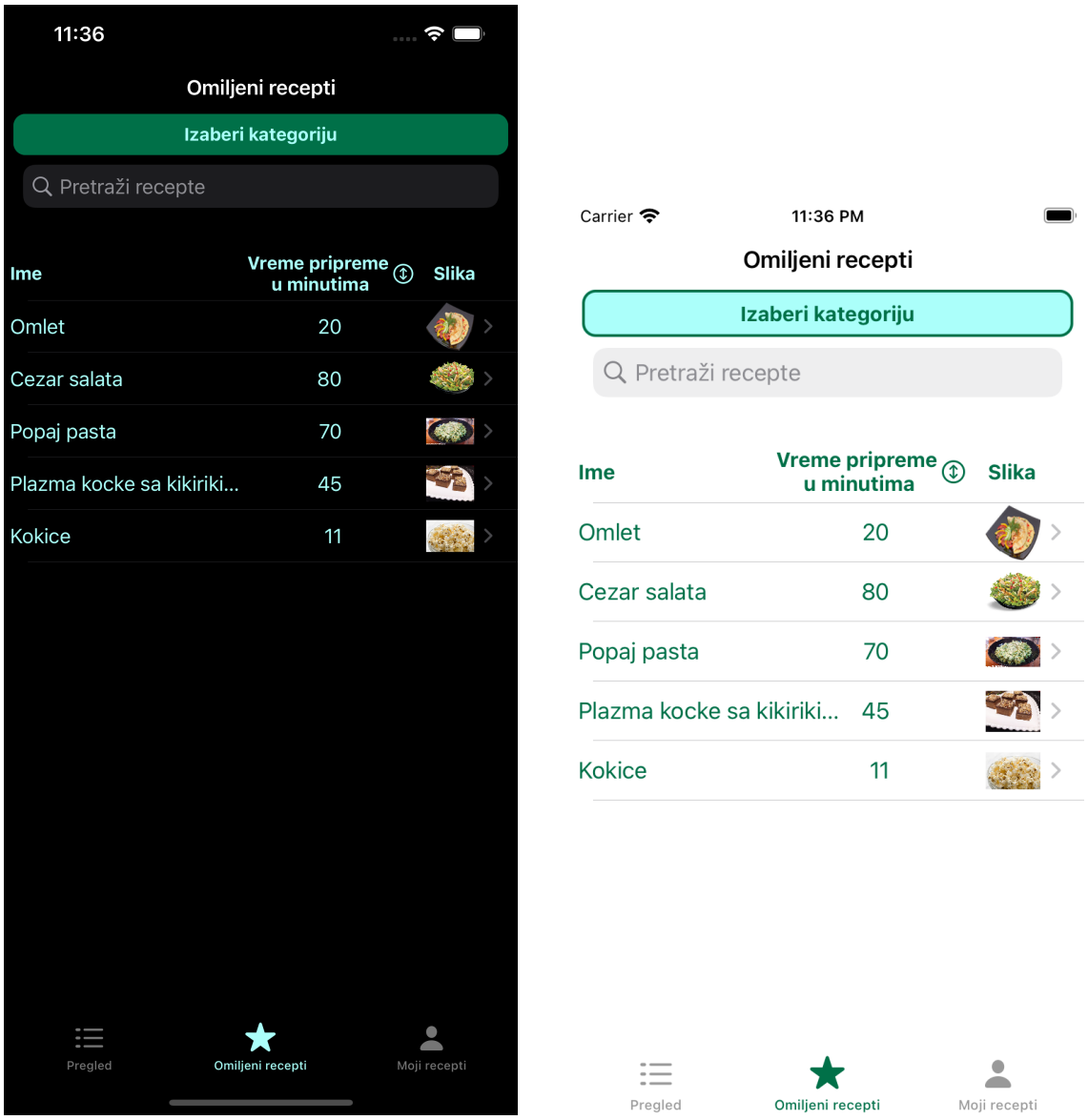
Слика 4.8: *Рецепти сортирани растуће — iPhone 13 (лево) и iPhone SE (десно)*

Омиљени и моји рецепти

Поред дела са прегледом свих рецепата на почетном екрану, кориснику се пружа могућност прегледа дела „Омиљени рецепти" и „Моји рецепти". Све могућности управљања рецептима (избор категорије, претрага по називу и сортирање) које су кориснику биле на располагању у почетном делу, су омогућене и у преостала два главна дела апликације.

„Омиљени рецепти" је део апликације који садржи листу свих рецепата

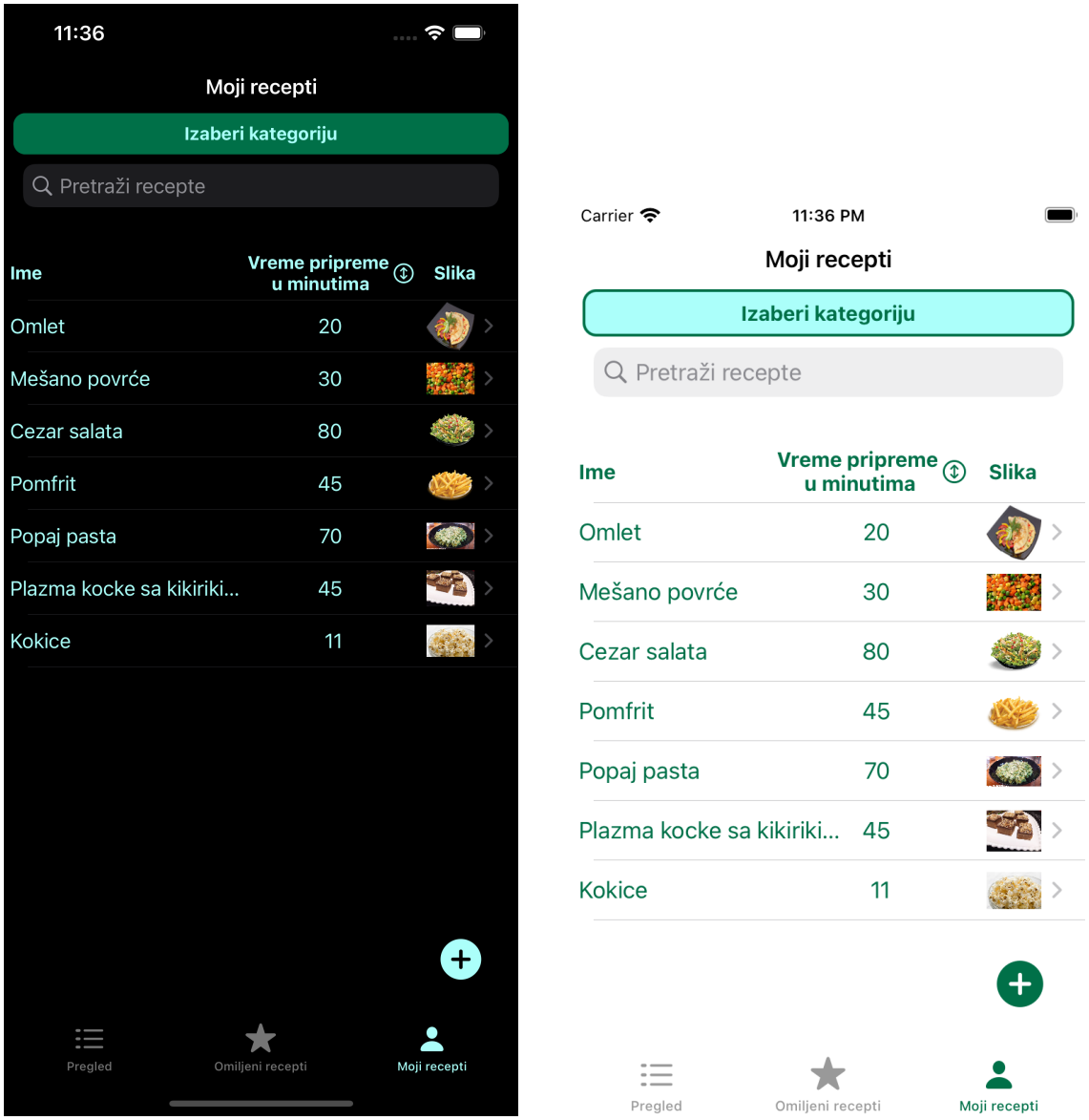
које је корисник означио као омиљене и тиме их издвојио од осталих. Слика 4.9 представља изглед дела „Омиљени рецепти”.



Слика 4.9: *Омиљени рецепти — iPhone 13 (лево) и iPhone SE (десно)*

Када корисник дода свој рецепт, он ће бити приказан у делу „Моји рецепти” у којем ће корисник моћи да види све своје рецепте са којима може манипулисати, више о овоме биће објашњено у делу 4.2 — Измена и брисање постојећег рецепта. Још једна могућност која се пружа кориснику на овој страни је додавање новог рецета, што ће детаљно бити објашњено у делу 4.2 — Креирање новог рецепта. Сликом 4.10 приказан је изглед странице „Моји

рецепти”.

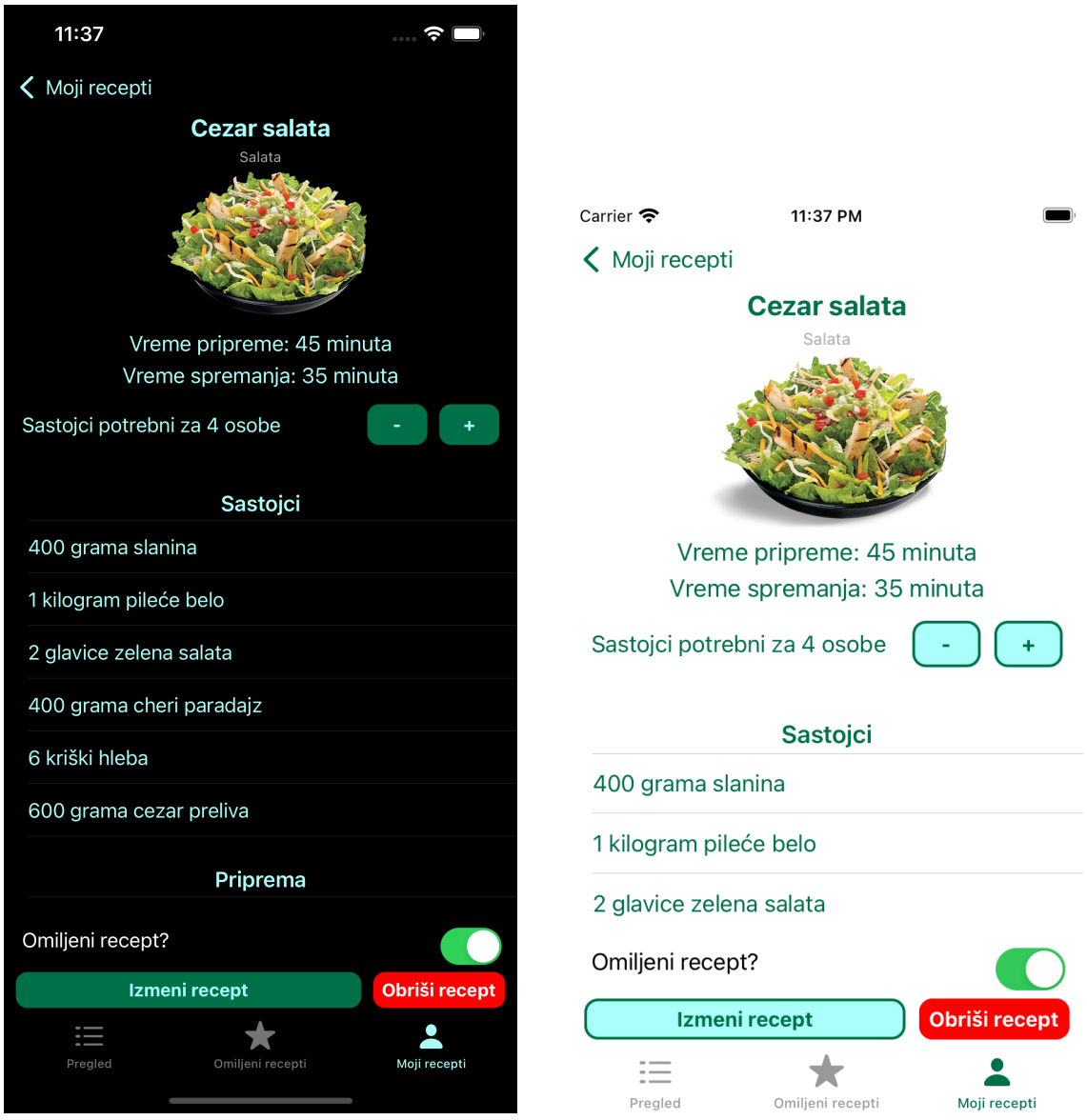


Слика 4.10: *Моји рецепти — iPhone 13 (лево) и iPhone SE (десно)*

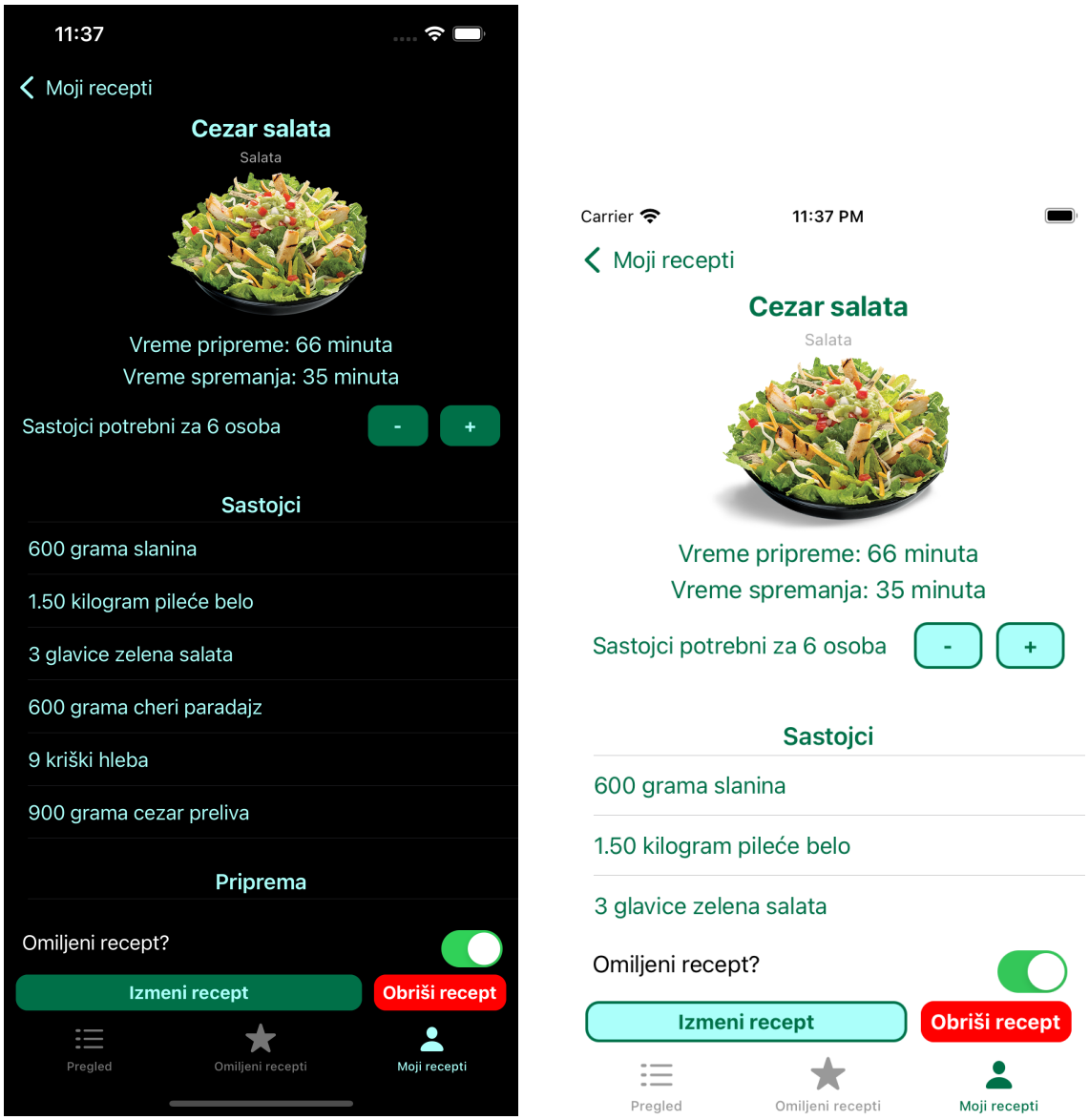
Детаљан приказ рецепта

У детаљном приказу рецепта, кориснику је презентован опис рецепта који се састоји од: назива рецепта, категорије којој рецепт припада, слике рецепта, времена припреме и спремања, броја особа за које је приказана количина састојака намењена, списка састојака и корака припреме, као и могућности додавања и брисања рецепта из листе омиљених рецепата. Кориснику се пружа

могућност повећавања и смањивања броја особа за које је рецепт предвиђен (истовремено ће бити промењена количина састојака као и време припреме). Уколико је приказани рецепт креиран од стране тренутног корисника, приказана су два дугмета за измену и брисање рецепта. Детаљан приказ рецепта за Цезар салату представљен је на слици 4.11 за четири особе, односно на слици 4.12 за шест особа.



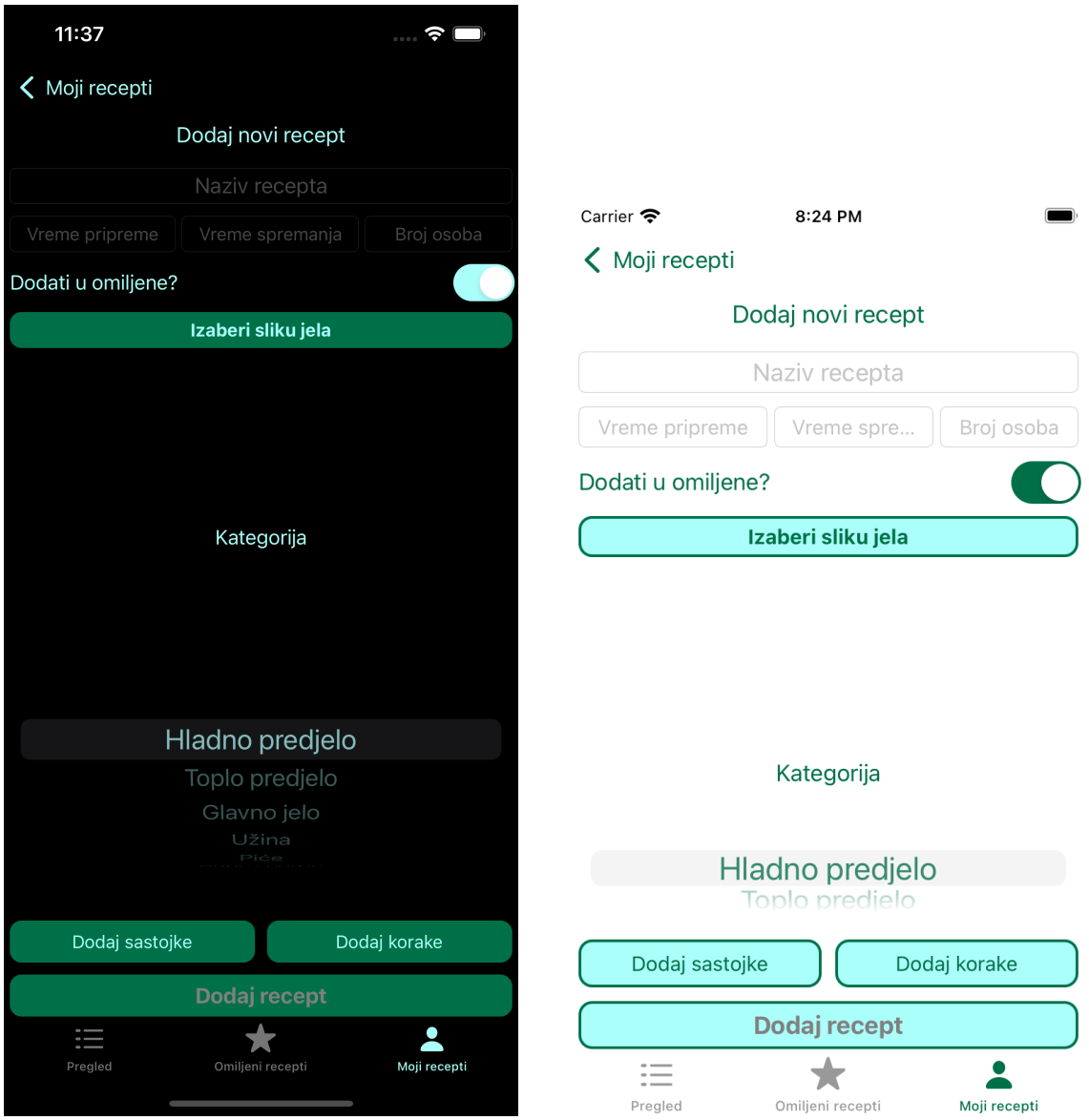
Слика 4.11: *Детаљан приказ рецепта — iPhone 13 (лево) и iPhone SE (десно)*



Слика 4.12: *Детаљан приказ рецепта за 6 особа — iPhone 13 (лево) и iPhone SE (десно)*

Креирање новог рецепта

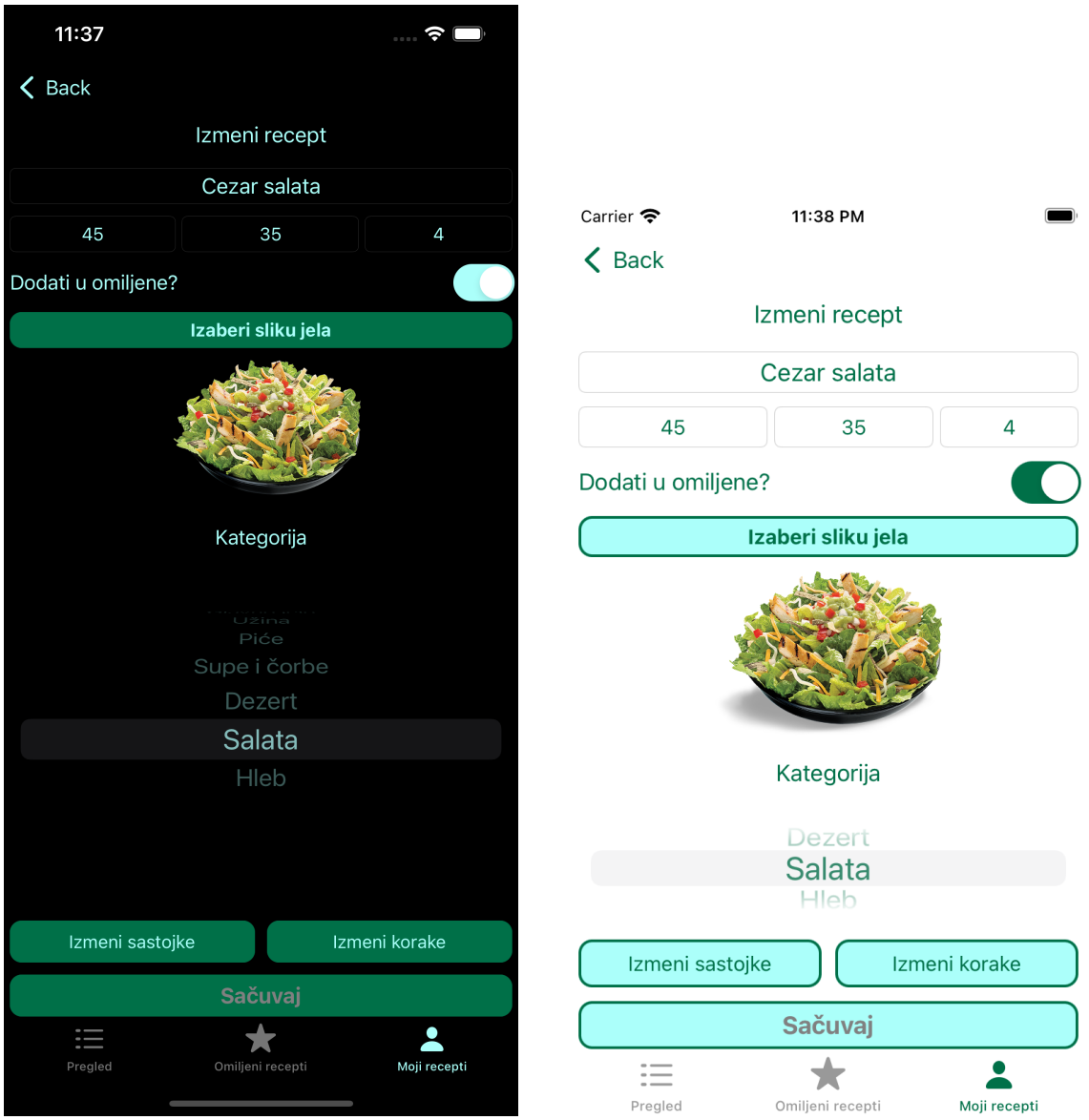
Приликом креирања новог рецепта од корисника се тражи да унесе све потребне информације које су наведене у претходном поглављу. Изглед погледа додавања новог рецепта може се видети на слици 4.13.



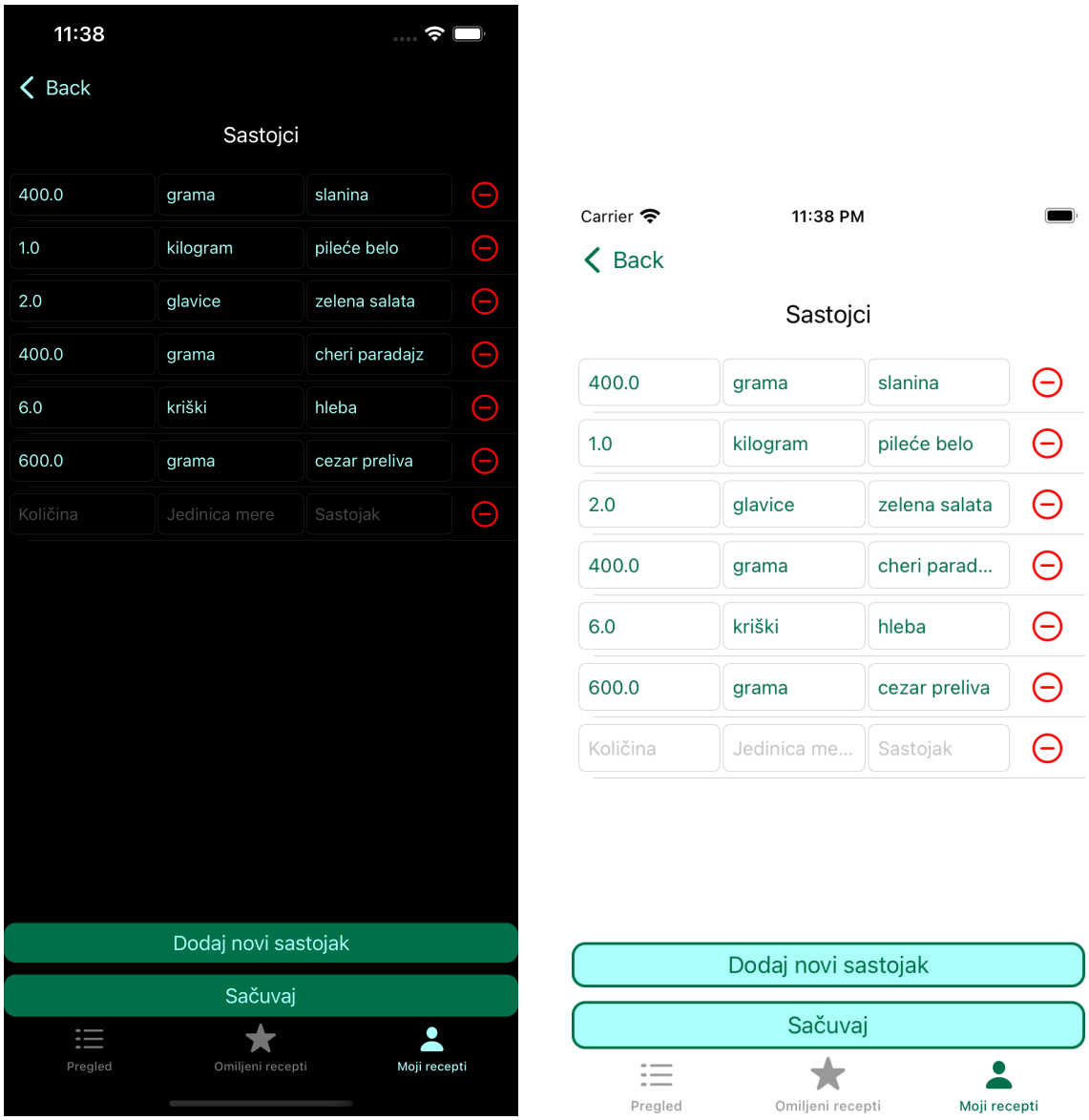
Слика 4.13: *Нов рецепт — iPhone 13 (лево) и iPhone SE (десно)*

Измена и брисање постојећег рецепта

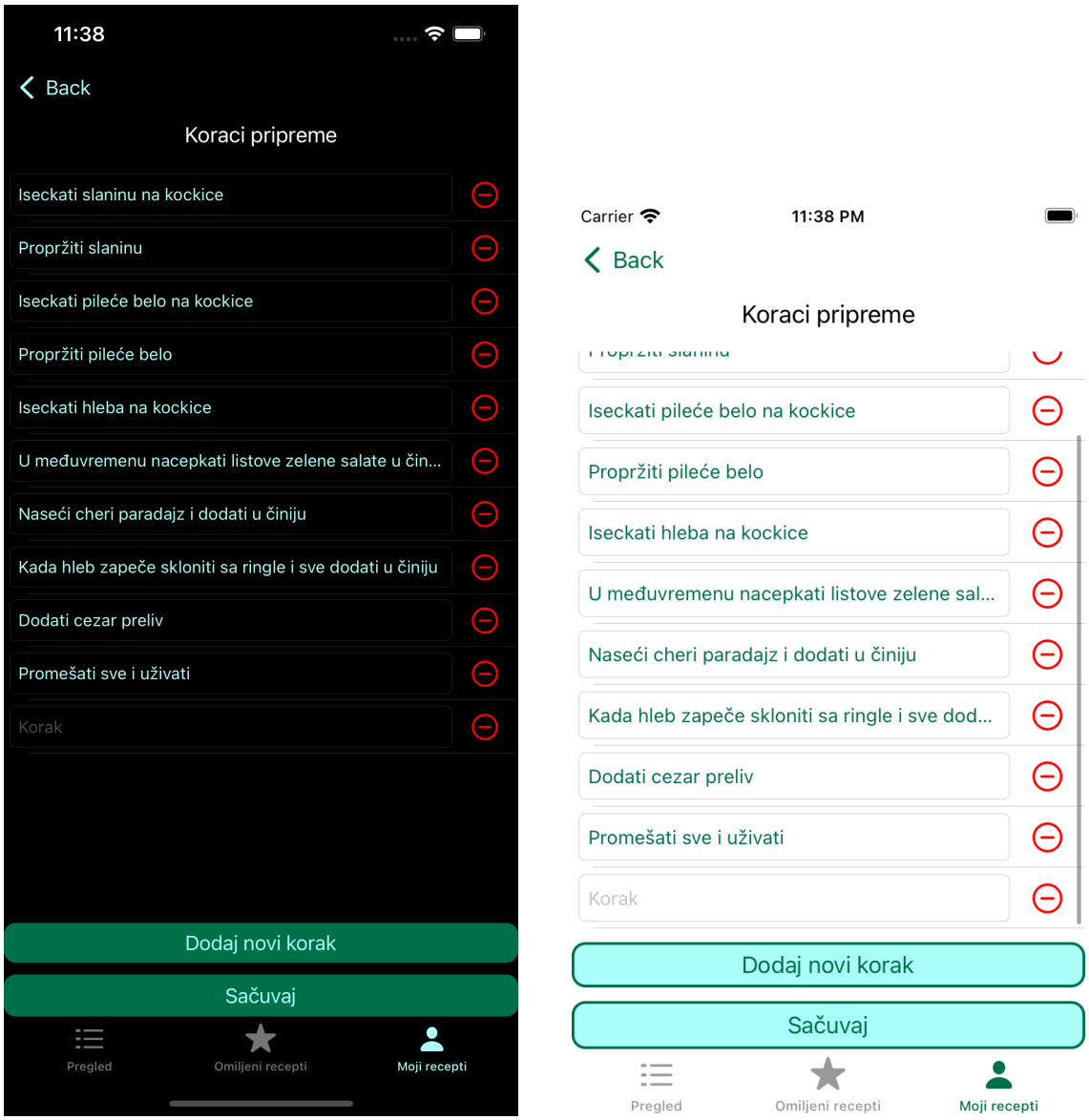
Корисник има могућност промене свих рецепата које је додао. Приказ изгледа погледа измене рецепта налази се на слици 4.14. Постоји могућност измене, брисања и додавања нових састојака (слика 4.15), измене, брисања и додавања нових корака припреме (слика 4.16).



Слика 4.14: *Измена рецепта — iPhone 13 (лево) и iPhone SE (десно)*

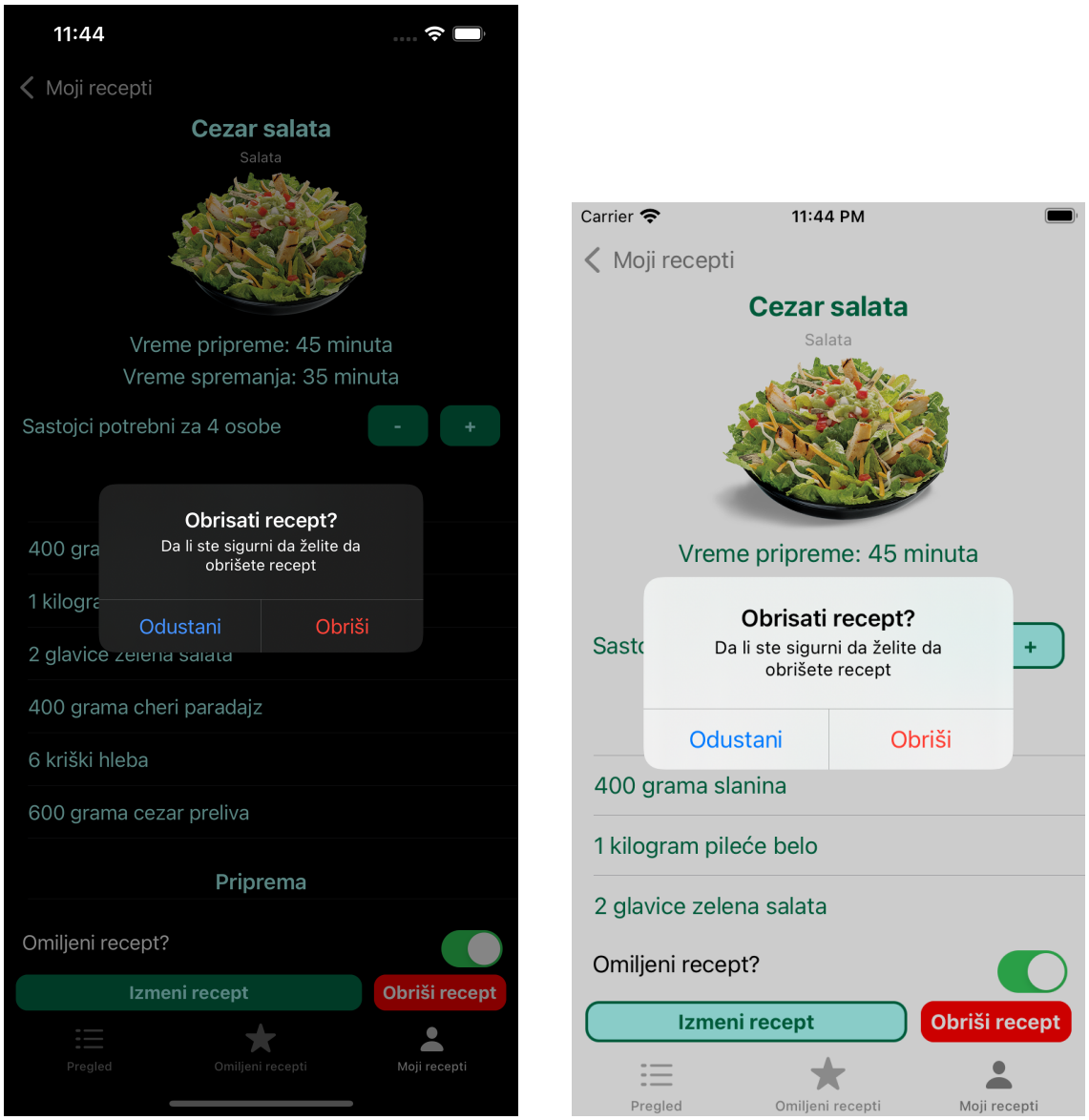


Слика 4.15: *Измена тренутних састојака за припрему рецепта уз додавање новог састојка — iPhone 13 (лево) и iPhone SE (десно)*



Слика 4.16: *Измена тренутних корака за припрему рецепта уз додавање новог корака припреме — iPhone 13 (лево) и iPhone SE (десно)*

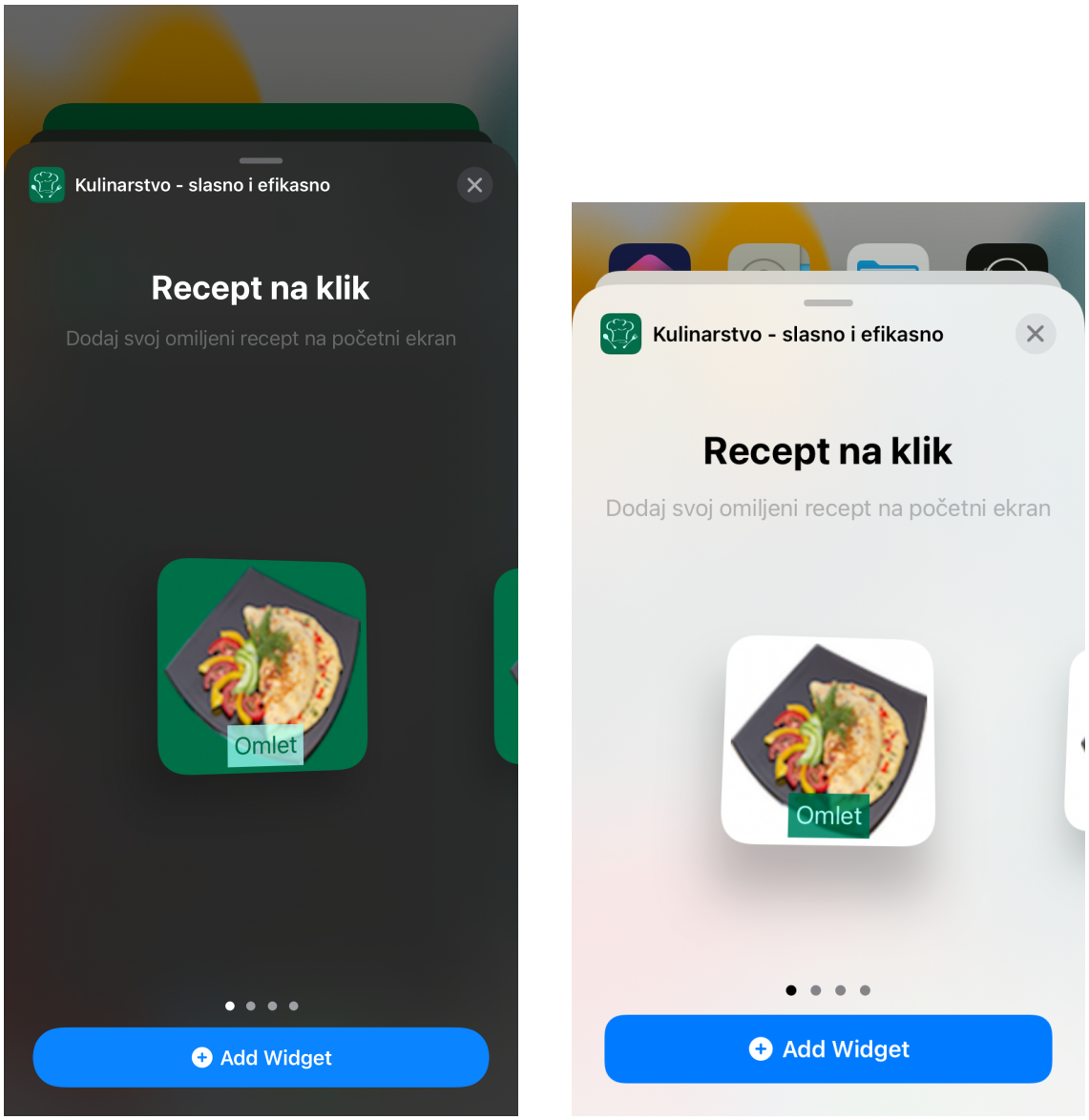
Уколико корисник жели, може и обрисати рецепте које је додао кликом на дугме „Обриши рецепт” унутар детаљног приказа рецепта, након чега ће му бити приказано упозорење у којем ће моћи да потврди брисање рецепта или да од њега одустане. Приказ упозорења приликом брисања рецепта налази се на слици 4.17.



Слика 4.17: *Брисање рецепта — iPhone 13 (лево) и iPhone SE (десно)*

Приказ вицета

Корисник може додати виџет на почетни екран из виџет галерије, где се налазе оба типа вицета ове апликације (први тип у све три величине, док је други доступан само у великој величини) који ће детаљније бити објашњени у наставку. Приказ додавања вицета из галерије представљен је на слици 4.18.



Слика 4.18: *Додавање виџета — iPhone 13 (лево) и iPhone SE (десно)*

Први тип виџета (назван „Рецепт на клик”) доступан је корисницима у све три величине (мала, средња и велика). Мали виџет приказује слику рецепта уз његов назив и може послужити као подсетник кориснику шта је испланирао да спрема или као пречица ка детаљном опису тог рецепта. Средњи виџет је проширење малог виџета који додатно приказује састојке или кораке припреме рецепта, параметар који је конфигурабилан од стране корисника. Приказ малог и средњег виџета може се видети на слици 4.19.

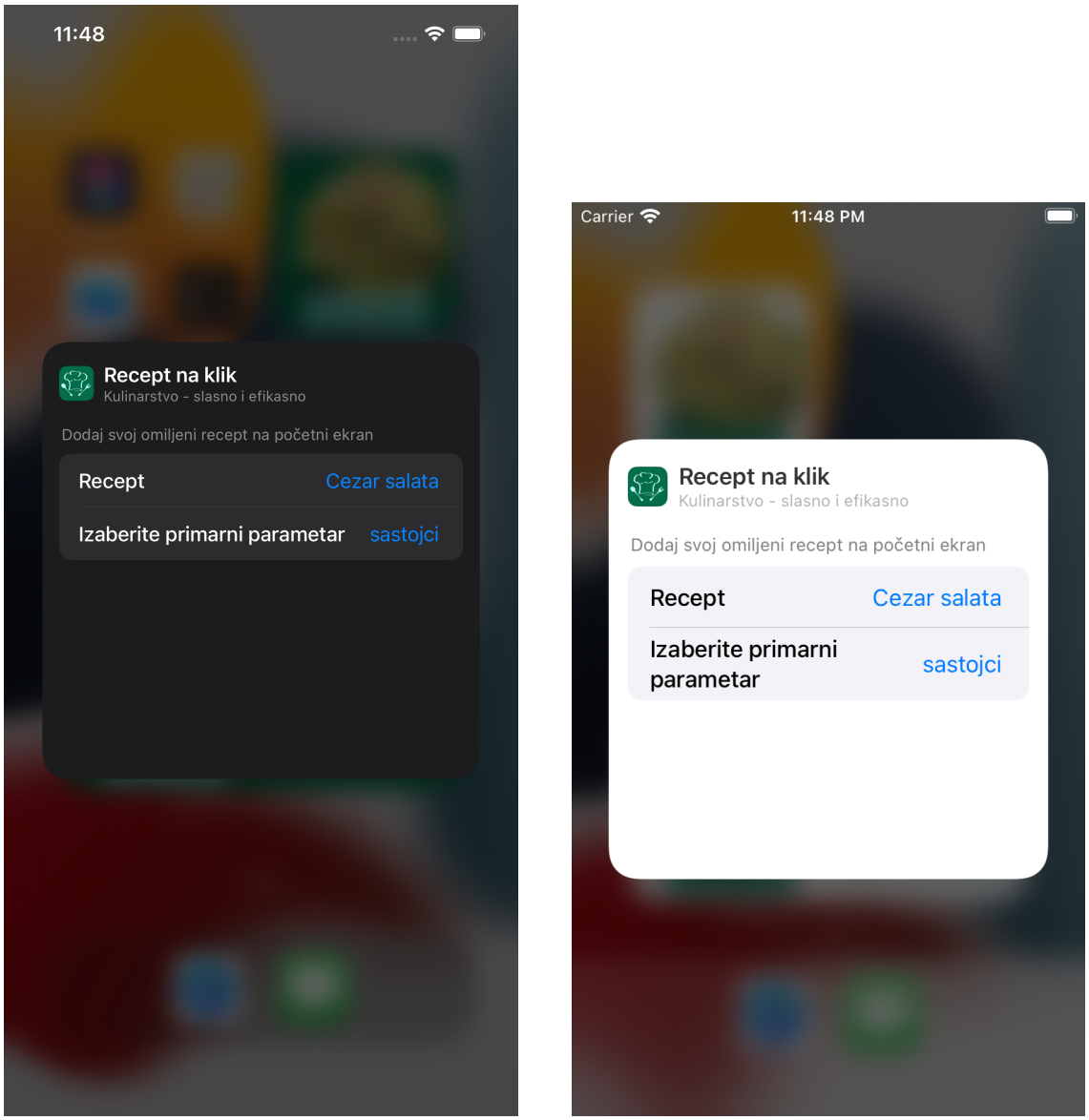
Слика 4.19: *Приказ вицета мале и средње величине — iPhone 13 (лево) и iPhone SE (десно)*

Први тип вицета у великој величини приказује рецепт са листама састојака и корака припреме, који су као код средњег вицета конфигурабилни и корисник може изабрати који ће од параметара бити примаран (приказан у дужој листи), уколико тај параметар испуњава услов — дужина листе мора бити већа од шест елемената (уколико услов није испуњен, секундарни параметар ће бити приказан у дужој листи; док уколико ни секундарна листа не испуњава услов, обе листе ће бити приказане у кратком формату и слика рецепта ће заузети средишњи горњи део вицета). Велики вицет првог типа

приказан је на слици 4.20.

Слика 4.20: *Приказ првог виџета велике величине — iPhone 13 (лево) и iPhone SE (десно)*

Први тип виџета је конфигурабилан, односно корисник може мењати одређене параметре. Код малог виџета може променити рецепт који ће му бити приказан и изабрати неки од рецепата из своје листе омиљених рецепата. Средњи и велики виџет поред избора рецепта омогућавају и одабир примарног параметра (листа састојака или листа корака припреме). Изглед екрана приликом конфигурације средњег виџета приказан је на слици 4.21.



Слика 4.21: *Измена виџета — iPhone 13 (лево) и iPhone SE (десно)*

Други тип виџета се разликује од првог по неколико карактеристика. Други тип је доступан само у великој величини, није га могуће конфигурисати и приказује четири рецепта. Рецепти који су приказани у овом типу виџета представљени су као скуп четири мала виџета првог типа, рецепти су насумично изабрани из корисникове листе омиљених рецепата, статички су конфигурисани и поновно се учитавају свака 24 сата и тако кориснику предлажу шта би могао да спрема тог дана. Приказ другог типа виџета може се видети на слици 4.22.

Слика 4.22: *Приказ другог виџета велике величине — iPhone 13 (лево) и iPhone SE (десно)*

Глава 5

Закључак

У раду је описан програмски језик *Swift*, његови основни и напредни концепти као и најважније особине. Опис језика је употпуњен конкретним примерима, којима су се постепено уводиле могућности које пружа овај програмски језик. Пример употребе програмског језика *Swift* приказан је приликом израде *iOS* апликације „Кулинарство — сласно и ефикасно".

Уз програмски језик *Swift* у раду је представљена и могућност декларативног програмирања употребом радног окружења *SwiftUI*. Конкретан пример употребе овог радног окружења приказан је приликом израде вицета који је имплементиран као додатак апликацији „Кулинарство — сласно и ефикасно".

Описана је имплементација апликације „Кулинарство — сласно и ефикасно" која има за циљ да крајњем кориснику пружи помоћ приликом одабира и припреме жељеног рецепта, уз приказ коришћене архитектуре и описа њених компоненти (модела, погледа и контролера). Апликација је у раду представљена и упоредним снимцима екрана два симулатора са различитим конфигурацијама уз описе њених делова, како би се приказала употреба апликације од стране корисника.

У току израде апликације показано је да је програмски језик *Swift* безбедан и концизан, карактеристике које су у раду наведене као најважније. Радно окружење *SwiftUI* је одличан начин примене декларативног програмирања и знатно скраћује време израде корисничког интерфејса апликације. Једна од запажених мана овог радног окружења приликом израде вицета је сама технологија која је веома млада и која се мења сваког дана, па се може наићи на потешкоће и нелогичности приликом њене употребе, које могу бити незгодне за превазилажење.

План за даље унапређење апликације је имплементација серверске стране апликације засноване на архитектуралном стилу репрезентативног преноса стања (енг. *REpresentational State Transfer*, *REST*) [2]. Поред тога биће додата функционалност креирања корисничких налога и пријављивања на исте употребом протокола *OAuth 2.0* [15].

Библиографија

[1] Edgar Nzokwe Giordano Scalzo. *SwiftUI Cookbook, Second Edition*. Packt Publishing Ltd., 2021.

[2] Lokesh Gupta. REST API. on-line at: `https://restfulapi.net/`.

[3] Apple Inc. Apple Developer. on-line at: `https://developer.apple.com/swift/`.

[4] Apple Inc. Apple Developer - Xcode. on-line at: `https://developer.apple.com/xcode/`.

[5] Apple Inc. Importing Objective-C into Swift. on-line at: `https://developer.apple.com/documentation/swift/imported_c_and_objective-c_apis/importing_objective-c_into_swift`.

[6] Apple Inc. Importing Swift into Objective-C. on-line at: `https://developer.apple.com/documentation/swift/imported_c_and_objective-c_apis/importing_swift_into_objective-c`.

[7] Apple Inc. Mastering Xcode Previews. on-line at: `https://developer.apple.com/videos/play/wwdc2019/233/`.

[8] Apple Inc. Structure your app for SwiftUI previews. on-line at: `https://developer.apple.com/videos/play/wwdc2020/10149/`.

[9] Apple Inc. Swift Education. on-line at: `https://www.apple.com/education/k12/teaching-code/`.

[10] Apple Inc. Swift on GitHub. on-line at: `https://github.com/apple/swift`.

[11] Apple Inc. Swift Playground. on-line at: `https://www.apple.com/swift/playgrounds/`.

[12] Apple Inc. SwiftUI. on-line at: `https://developer.apple.com/xcode/swiftui/`.

[13] Apple Inc. *The Swift Programming Language (Swift 5.6)*. Apple Inc., 2014.

[14] Apple Inc. Swift.org, 2021. on-line at: `https://www.swift.org/`.

[15] Aaron Parecki. OAuth 2.0. on-line at: `https://oauth.net/2/`.

[16] StackOverflow. Stack overflow. on-line at: `https://stackoverflow.com/`.

[17] Marko Veljkovic. Kulinarstvo — slasno i efikasno. on-line at: `https://github.com/bataVeljko/Kulinarstvo-iOS-Widget`.

Биографија аутора

Марко Вељковић рођен је 27.02.1996. у Зајечару. Основну школу и информатички смер Зајечарске гимназије завршио је као носилац Вукове дипломе.

Смер информатика на Математичком факултету Универзитета у Београду уписао је 2015. године, а завршио септембра 2019. године са просечном оценом 8.53. Након тога је уписао мастер студије информатике на истом факултету.

У марту 2020. године почео је са праксом у компанији *Teletrader* као јуниор *iOS* програмер, где је након месец дана добио стално запослење. Тренутно ради у истом тиму као главни програмер на развоју и одржавању апликације *StockMarkets* намењену уређајима *iPad*, и усавршава своје знање у области развоја мобилних апликација за оперативне системе *iOS*, *iPadOS* и *Android*.