

УНИВЕРЗИТЕТ У БЕОГРАДУ
МАТЕМАТИЧКИ ФАКУЛТЕТ



Марко Вельковић

КРЕИРАЊЕ ВИЦЕТА У ПРОГРАМСКОМ
ЈЕЗИКУ SWIFT

мастер рад

Београд, 2022.

Ментор:

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

Чланови комисије:

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

др Мирко Спасић, доцент
Универзитет у Београду, Математички факултет

Датум одбране: 15. јануар 2022.

Наслов мастер рада: Креирање виџета у програмском језику Swift

Резиме: Употреба програмског језика *Swift* за израду виџета у *iOS* оперативном систему

Кључне речи: виџет, *Swift*, *iOS*, *Apple*, програмски језик, програмирање

Садржај

1 Увод	1
2 Програмски језик Swift	2
2.1 Настанак, развој и карактеристике	2
2.2 Основни концепти	3
2.3 Напредни концепти	16
2.4 Особине	27
2.5 Xcode	32
2.6 SwiftUI	37
3 Улога и развој вицета	47
3.1 Основно	47
3.2 Развој вицета	48
3.3 Дизајн вицета	55
4 Опис апликације	59
5 Закључак	82
Библиографија	83

Глава 1

Увод

Глава 2

Програмски језик Swift

Swift је модеран програмски језик, првенствено намењен развоју апликација на платформама компаније *Apple* (*iOS*, *iPadOS*, *macOS*, *tvOS* и *watchOS*). Настао је 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 Латнер¹ (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, бесплатно упутство за коришћење језика „The Swift Programming Language” [6] и званична веб страница програмског језика [7]. 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* [8] проглашен за омиљени програмски језик 2015. године, док је 2016. године заузео друго место у тој категорији. Децем-

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

ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

бра 2015. године изворни кôд језика, подржане библиотеке, дибагер и менаџер пакета постали су отвореног кода, под лиценцом *Apache 2.0*, доступни на *GitHub*-у [3].

На годишњој конференцији *WWDC* 2016. године представљена је апликација за уређај *iPad* под називом *Swift Playgrounds* [4], 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* [5] које омогућава декларативно програмирање апликација за све *Apple* платформе. У време писања рада последња званична верзија језика је *Swift 5.6*.



Слика 2.1: *Swift* лого

Swift је моћан и интуитиван језик за програмирање апликација намењених *Apple* платформама. Писање кода у *Swift*-у је забавно и лако, синтакса је веома концизна, али у исто време веома изражajна. 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 Основни концепти

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 [7].

Основе програмског језика *Swift*

Swift подржава основне типове променљивих, *Int* за целобројне вредности, *Float* и *Double* за бројеве са основом 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*, након чега следи име константе и њена иницијализација. Декларисање променљиве се постиже употребом кључне речи *var*. Када се декларише променљива 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 inicializaciju
5 var raspolozivoNovca = 1000
6
7 // Deklarisanje promenljive koriscenjem anotacije
8 var brojPorcija: Int
```

Пример кода 2.1: *Декларисање променљивих и константи*

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

```
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 променљиве*

Торка се користи за груписање вредности било ког типа и једна торка може садржати вредности различитих типова. У примеру 2.3 - *Торка* могу

ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

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)
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 означенено 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, "Cena proizvoda ne moze biti negativna
4 ") //true
5 assert(cenaDrugogProizvoda >= 0, "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

ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

- Префиксни унарни оператори (на пример, '-' за бројевне вредности и '!' за логичке вредности)
- Постфиксни унарни оператори (на пример, '?' и '!' који се користе над опционаним 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. При- мер примене тернарног оператора као и приказ блока кода условног гранања (које је објашњено у делу 2.2 - Контрола тока) који једна линија са тернарним оператором може заменити 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*

ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

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

- Оператор 'nil-сједињавања' ('??') је бинарни оператор који се користи над опционим променљивима. 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 pseudRandomBroj = Int.random(in: 1...100)  
3 if pseudRandomBroj % 2 == 0 {  
4     a = 10  
5 }  
6 let b = a ?? 5
```

Пример кода 2.6: *nil-сједињавање*

- Оператори распона

- Затворен распон (*a...b*), распон од 'a' до 'b' укључујући обе вредности.
- Полу-отворен распон (*a..*), распон од 'a' до 'b' укључујући само вредност 'a'.
- Распони једне стране [*a...*], распон од 'a' и надаље докле год је то могуће (ову врсту оператора треба користити опрезно).

Карактери и стрингови

Стринг је низ карактера, као што је „Здраво, светe”. У *Swift*-у се стрингови представљају помоћу класе *String*, која омогућава брз, ефикасан и *Unicode*-компабилан начин рада са текстом. Да би се вредност неке променљиве или израза уметнула у стринг користи се операција уметања, односно интерполације. Уметање променљиве или израза у стринг се постиже њиховим уписивањем између обрнуте косе црте након чега следи отворена заграда, па променљива или израз и на крају затворена заграда. Креирање и операције са стринговима (надовезивање, рад са карактерима и уметање) биће приказане кроз пример 2.7 - *Операције нај сћринговима*.

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

Пример кода 2.7: *Операције нај сиричноговима*

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", "
3     Paradajz", "Hleb", "Cezar premaz"]
4 var brojSastojaka = recept.sastojci.count
5
6 if brojSastojaka < 6 {
    print("Nisu svi sastojci nabavljeni")
```

ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

```
7 }
8     else if brojSastojaka > 6 {
9         print("Broj sastojaka je veci nego u receptu, ali samo napred
10        eksperimentisi")
11    }
12    else {
13        print("Broj sastojaka je odgovarajuci")
14    }
```

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")
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). 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 ко-

ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

лекције и њихове индексе и променом елемента колекције на одговарајућем индексу. 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 концроле цикла 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..
```

Пример кода 2.10: *Наредба концроле цикла 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 концроле цикла while*.

```
1 let nasumicniBrojevi = [3, 12, 5, 18, 11, 99]
```

ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

```
2 var i = 0
3
4 while i < nasumicniBrojevi.count, nasumicniBrojevi[i] < 15 {
5     print("Broj \(nasumicniBrojevi[i]) je manji od 15")
6     i += 1
7 }
8
9 let nasumicniBroj = nasumicniBrojevi[2] // 5
10
11 repeat {
12     print("Zdravo, svete!")
13 } 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 - *Догаџи 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: Догаџи наредбама контроле тока

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,

ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

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: 2.13 - *Поштис функције*, док је потпис параметара функције приказан у примеру: 2.14 - *Поштис ћарамећара функције*.

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

Пример кода 2.13: *Поштис функције*

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

Пример кода 2.14: *Поштис ћарамећара функције*

Као што је већ наведено, параметри функције могу бити именовани и неименовани. Приликом позивања функције са именованим параметрима потребно је навести лабеле параметара уз конкретну вредност. Приликом дефинисања функције, 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 2.15 - *Дефинисање и ћозивање функције са ћарамећаром*, док пример 2.16 - *Дефинисање и ћозивање функције са ћовраћном 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 2.15: *Дефинисање и ћозивање функције са ћарамећаром*

```
1 func izracunajCenu(_ proizvodi[String: Int]) -> Int {
2     var ukupnno = 0
```

ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

```
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 2.16: *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 2.17 - *Дефинисање и позивање функције са параметрима са подразумеваним вредносћима.*

```
1 func ispisiSastojkeSaDvaParametra(sastojci: [String], ispisati
2     ispisatiCeloIme: Bool = true) {
3     for sastojak in sastojci {
4         if ispisatiCeloIme {
5             print(sastojak)
6         } else {
7             print(sastojak.prefix(3))
8         }
9     }
10 }
11 let sastojci = ["Jaja", "Sira"]
12 ispisiSastojkeSaDvaParametra(sastojci: sastojci, ispisati: false)
13 ispisiSastojkeSaDvaParametra(sastojci: sastojci)
14 // Параметар 'ispisati' се имати вредност 'true'
```

Пример кода 2.17: *Дефинисање и позивање функције са параметрима са подразумеваним вредносћима*

Опционе променљиве и рад са њима

Опционе променљиве се користе у ситуацијама када није сигурно да ли ће променљива имати неку вредност, да би се избегло приступање таквој променљивој јер би дошло до грешке у раду програма (када променљива нема

ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

вредност њена подразумевана вредност је *nil* и са њом се мора пажљиво рукувати).

Када се дефинише опциона променљива експлицитно се наводи ког је типа након чега следи знак '?' и након тога се може, а не мора, извршити иницијализација. Уколико се променљива иницијалзијује без експлицитног навођења опционог типа, неопходно је да израз којим се иницијализује променљива буде опционог типа, у супротном променљивој не би био додељен опциона тип и она не би могла да се користи као опциона променљива. Пример дефинисања опционе променљиве уз иницијализацију и експлицитно навођење опционог типа, као и два начина иницијализације без навођења типа може се видети у делу 2.18 - *Дефинисање оцијоне променљиве*.

```
1 // Opciona promenljiva tipa 'Int?'
2 var opciona: Int? = 42
3 // Promenljiva tipa 'String'
4 var brojUOblikuStringa = "55"
5 // Opciona promenljiva tipa 'Int?'
6 var konvertovaniBroj = Int(brojUOblikuStringa)
```

Пример кода 2.18: *Дефинисање оцијоне променљиве*

У неким ситуацијама не може се радити са опционим променљивима, на пример када се прослеђују као параметри функције која очекује конкретну вредност, па се опциона променљива 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 променљиве и руковање са *nil* вредношћу.

Први начин је коришћењем условног гранања (*if* или *guard*) приказан у примеру 2.19 - *Одмотавање оцијоне променљиве коришћењем услова*, а други начин задавањем подразумеване вредности односно коришћењем оператора *if*-сједињавања (енг. *nil coalescing*) приказан у примеру 2.20 - *Одмотавање оцијоне променљиве 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
9 // tipa Int? dok funkcija ocekuje parametar tipa Int
```

ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

```
9
10 // 1. nacin koriscenjem if-a
11 if let raspakovaniBroj = opcioniBroj {
12     saberiDvaBroja(raspakovaniBroj, broj) // 'raspakovaniBroj' je
13     tipa Int
14 }
15 else {
16     print("Prvi broj nema vrednost, ne moze se sabrati")
17 }
18
19 // 2. nacin koriscenjem if-a
20 if opcioniBroj != nil {
21     saberiDvaBroja(opcioniBroj!, broj)
22 }
23 else {
24     print("Prvi broj nema vrednost, ne moze se sabrati")
25 }
26
27 // 3. nacin koriscenjem guard-a
28 guard opcioniBroj != nil else {
29     print("Prvi broj nema vrednost, ne moze se sabrati")
30     return
31 }
32 saberiDvaBroja(opcioniBroj!, broj)
```

Пример кода 2.19: *Одмештаавање оиционе 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 2.20: *Одмештаавање оиционе променљиве задавањем подразумеване вредности*

2.3 Напредни концепти

Колекције

Swift дефинише три примарна типа колекција: низове, скупове и речнике. Сва три типа су дефинисана као генеричке² колекције. Уколико се дефинисана колекција додели променљивој, 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, на различитим индексима. Конкретан пример дефинисања, иницијализације и управљања подацима низа може се видети у коду 2.21 - *Раг 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 inicialna 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() {
11     if index < 3 {
12         koraci[index] = korak
13     }
14     else {
15         koraci.append(korak)
16     }
17 }
18 // Brisanje prvog clana niza, ukoliko niz nije prazan
19 if !koraci.isEmpty {
20     koraci.remove(at: 0)
21 }
```

Пример кода 2.21: *Раг са низовима*

Скупови су колекције које не гарантују чување редоследа елемената и у којима један елемент може да се појави највише једанпут. Тип елемента скупа

²Генерички код омогућава писање флексибилних и поновно искористивих функција и типова; помоћу њих се избегава дуплирање кода

ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

мора бити могуће кодирати³ (енг. *hashable*). Рад са скуповима приказан је у примеру 2.22 - *Pag sa skupovima*.

```
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  
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
```

³Тип који се може кодирати мора имати дефинисану функцију за одређивање *hash* вредности за сваку инстанцу, два елемента могу имати исту hash вредност ако су једнаки

Пример кода 2.22: *Pag sa 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 примеру 2.23 - *Pag sa речницима*.

```
1 // Kreiranje recnika tipa [Int : String]
2 var tipoviHTTPStatusa: [Int : String] = [:]
3
4 // Dodeljivanje recnika promenljivoj 'tipoviHTTPStatusa',
5 tipoviHTTPStatusa = [200: "OK", 201: "Resurs je kreiran", 202: "
6     Zahtev je prihvacen"]
7
8 // Dodavanje novog elementa ukoliko ne postoji, odnosno promena
9     postojeceg
10 tipoviHTTPStatusa[404] = "Stranica nije pronadjena"
11
12 // Brisanje elementa iz recnika
13 if let izbrisanaVrednost = tipoviHTTPStatusa.removeValue(forKey:
14     201) {
15     print("Vrednost izbrisana iz recnika: \(izbrisanaVrednost)")
16 }
17
18 // Razlicite vrste iteracija kroz recnik
19 for kod in tipoviHTTPStatusa.keys {
20     print(kod)
21 }
22
23
24 for status in tipoviHTTPStatusa.values {
25     print(status)
26 }
```

Пример кода 2.23: *Pag sa речницима*

Генеричке функције и затворења

Као и други објектно оријентисани програмски језици и *Swift* пружа могућност дефинисања генеричких функција⁴ 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 2.24 - *Дефинисање и позивање генеричке функције са више повратних вредносћи*.

```
1 // Definisanje generickie 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 nizBrojeva = [5, 12, -4, 19, -99]
13 let minimumIMaksimum = minMax(niz: nizBrojeva)
14 let minimum = minimumIMaksimum?.min
15 let maksimum = minimumIMaksimum?.max
```

Пример кода 2.24: *Дефинисање и позивање генеричке функције са више повратних вредносћи*

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 2.25 - *Дефинисање и позивање функције са променљивим параметрима*.

```
1 func zameniDvaParametra<T>(prvi: inout T, drugi: inout T) {
```

⁴Генеричка функција је функција са генеричким параметрима или генеричким повратним вредностима. Генерички тип може представљати више различитих типова одједном који задовољавају одређене услове дефинисане од стране програмера

ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

```
2     var pomocna = prvi
3     prvi = drugi
4     drugi = pomocna
5 }
6
7 // Prosledjeni parametri moraju biti promenljive
8 var prviString = "Ja sam prvi"
9 var drugiString = "Ja sam drugi"
10
11 print(prviString + ", " + drugiString)
12 // Ja sam prvi, Ja sam drugi
13
14 zameniDvaParametra(prvi: &prviString, drugi: &drugiString)
15 print(prviString + ", " + drugiString)
16 // Ja sam drugi, Ja sam prvi
```

Пример кода 2.25: *Дефинисање и јозивање 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 изражажности. Синтакса израза затворења приказана је у примеру 2.26 - *Синтакса израза затворења*. На примеру 2.27 - *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 2.26: *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
```

ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

```
9 // Koriscenjem zatvorenja
10 sortiraniBrojevi = nasumicniBrojevi.sorted(by: { (broj1: Int,
11     broj2: Int) -> Bool in
12     return broj1 < broj2
13 })
14
15 // Bez eksplicitnog navodjenja tipa parametra
16 sortiraniBrojevi = nasumicniBrojevi.sorted(by: {broj1, broj2 in
17     return broj1 < broj2
18 })
19
20 // Kada u zatvorenju postoji samo jedna naredba, nije potrebno
21 // navodjenje kljucne reci 'return', povratna vrednost bice
22 // vrednost izvrseranja te naredbe
23 sortiraniBrojevi = nasumicniBrojevi.sorted(by: {broj1, broj2 in
24     broj1 < broj2
25 })
26
27 // Swift omogucava i kratka imena parametara, za pruzanje
28 // izrazajnije sintakse
29 sortiraniBrojevi = nasumicniBrojevi.sorted(by: {
30     $0 < $1
31 })
32
33 // Kada koristimo tipove za koje je vec definisano ponasanje
34 // prilikom poredjenja, mozemo proslediti samo kako zelimo da
35 // sortiramo clanove niza
36 sortiraniBrojevi = nasumicniBrojevi.sorted(by: <)
```

Пример кода 2.27: 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 2.28 - 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 {
```

ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

```
6         completion(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 2.28: Редно заштаворење

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 инстанцу класе као и 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*), имплементација протокола. Функционалности које поседују само класе су: наслеђивање друге класе, провера типа инстанце у времену извршавања програма, деиницијализација.

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 (енг. dot syntax) може се видети у примеру 2.29 - *Дефинисање класе и структуре*.

```
1 // Definisanje strukture
```

ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

```
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 glavniIzgled = 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
25 // svojstva
25 let maliSiviOkvir = OkvirPozadine(visina: 50, sirina: 50, boja: .
    gray)
```

Пример кода 2.29: *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 2.30 - *Лењо својство*.

```
1 struct UcitavanjeFajla {
2     var imeFajla = "recepti.txt"
3 }
4
```

ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

```
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 ,
15 // 'UcitavanjeFajla'
15 print(menadzer.ucitavanje.imeFajla)
```

Пример кода 2.30: *Ленъ својство*

За разлику од својства и лењих својства 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 2.31 - *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) {
```

ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

```
16         self.ukupnoVremeSpremanja = self.vremeKuvanja +
17             novoVremePripreme
18     }
19
20     var ukupnaCena: Double {
21         return potrebneNamirnice.reduce(0) {
22             $0 + $1.cena * $1.kolicina
23         }
24     }
25
26     var lazanje = Recept(potrebneNamirnice: [Namirnica(ime: "Kore",
27         cena: 250, kolicina: 1), Namirnica(ime: "Mleveno meso", cena:
28         500, kolicina: 0.75), Namirnica(ime: "Sos", cena: 100, kolicina
29         : 1)], vremeKuvanja: 25, ukupnoVremeSpremanja: 70)
30
31     print("Ukupna cena recepta: \(lazanje.ukupnaCena)")
32     // Ukupna cena recepta: 725
33     print("Vreme pripreme recepta: \(lazanje.vremePripreme)")
34     // Vreme pripreme recepta: 45
35     print("Staro ukupno vreme pripreme: \(lazanje.ukupnoVremeSpremanja
36         )")
37     // Staro ukupno vreme pripreme: 70
38     lazanje.vremePripreme = 30 // Umesto 45
39     print("Novo ukupno vreme pripreme: \(lazanje.ukupnoVremeSpremanja
40         )")
41     // Novo ukupno vreme pripreme: 55
```

Пример кода 2.31: Рачунајућа својсвава

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

ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

посматрача приказан је у делу 2.32 - *Посматрачи својсћива*.

```
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 2.32: *Посматрачи својсћива*

Методе су функције које су везане за одређени тип класе, структуре или набрања (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 2.33 - *Методе*.

```
1 class Recept {  
2     var ime: String  
3  
4     // Koriscenjem kljucne reci 'self', naglasava se pristupanje  
5     // svojstvu/metodi klase  
6     init(ime: String) {  
7         self.ime = ime  
8     }  
9  
10    // Metod koji ispisuje svojstvo 'ime'  
11    func ispisiIme() {  
12        print(ime)  
13    }  
14  
15    // Metod koji menja svojstvo 'ime', parametrom 'ime'  
16    func promeniIme(novo ime: String) {
```

ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

```
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 2.33: *Мешавине*

Као и у свим објектно оријентисаним језицима, и у *Swift*-у постоји наслеђивање класа. Класа која наследи другу класу наслеђује сва њена својства и методе које нису дефинисане као приватне и може их мењати, односно преписати (енг. *override*). Свака класа која не наслеђује ниједну другу класу назива се основна класа. Пример наслеђивања класе може се видети у делу 2.34 - *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 2.34: *Наслеђивање класа*

2.4 Особине

Програмски језик *Swift* је креиран са намером да буде безбедан, модеран и ефикасан. Детаљан опис ових као и других важних особина дат је у наставку

поглавља.

Модеран језик

Swift је настао као резултат најновијих истраживања програмских језика. Именовани параметри су изражажни у једноставној синтакси што код чини лаким за читање и разумевање. 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 код чистијим и отпорнијим на грешке. Меморијом се управља аутоматски, коришћењем аутоматског бројача референци (енг. *Automatic Reference Counting, ARC*).

ARC се користи за праћење и управљање меморијом коју апликација користи. Бројач референци води рачуна само о инстанцима 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 својства, 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 2.35 - *Слаба референца*.

```
1 protocol NoviReceptDelegate : AnyObject {
2     func receptJeKreiran(_ viewController: NoviReceptController, _ 
3         recept: Recept)
4 }
```

ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

```
5  class NoviReceptController() : UIViewController {
6      weak delegate: NoviReceptDelegate?
7      ...
8      func sacuvajRecept(_ recept: Recept) {
9          ...
10         self.delegate?.receptJeKreiran(self, recept)
11     }
12 }
13 class ReceptController : UIViewController {
14     var novRecept = NoviReceptController()
15     override viewDidLoad() {
16         ...
17         self.novRecept.delegate = self
18     }
19     ...
20 }
21 extension ReceptController : NoviReceptDelegate {
22     func receptJeKreiran(_ viewController: NoviReceptController, -
23     recept: Recept) {
24 }
```

Пример кода 2.35: Слаба референца

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 типова (класе, структуре)*, док се истовремено 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*. Ограниччење код екstenзије 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) {
```

ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

```
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: *Нагођања посебнојећет шића (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: *Нагођања Swift класе*

Безбедан начин програмирања

У току развијања језика уложени су огромни напори да би он био што безбеднији. Променљиве су увек иницијализоване, низови и целобројне променљиве се увек проверавају да не би дошло до прекорачења, меморијом се управља аутоматски и много других карактеристика. Још једна безбедносна одлика је да *Swift* објекти подразумевано никада не могу бити *nil*. *Swift* компајлер ће спречити покушај да се направи или искористи *nil* објекат, избацивањем грешке у време превођења програма. Међутим, постоје случајеви

ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

када је коришћење вредности *nil* валидно. За те случајеве користе се опционе променљиве.

Ефикасно извршавање

Swift је наследник програмских језика *C* и *Objective-C* и као такав од почетка је дизајниран да буде концизан и ефикасан. Коришћењем технологије *LLVM* компајлера, *Swift* код се трансформише у оптимизовани изворни код који извлачи највише из модерног хардвера. Синтакса и стандардна библиотека су такође направљени тако да учине да се најочитији начин кодирања извршава најбоље без обзира на ком је уређају програм покренут.

Одличан језик за почетнике

Swift је дизајниран тако да може бити свачији први програмски језик. У циљу подучавања *Apple* је направио бесплатан наставни план и програм који може свако користити [2]. Најбоља апликација за почетнике је *Swift Playgrounds* [4], апликација у почетку намењена уређајима *iPad*, развијена од стране *Apple*-а. Касније је иста апликација развијена и за употребу на уређајима са оперативним системом *macOS*.

Отвореног кода

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 код 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 података
- *Xcode* подршка за *Swift Playgrounds*

Package manager

Swift package manager је вишеплатформски 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 података су такође писани у *Swift*-у, чинећи конфигурацију циљаних извршних датотека и управљање зависностима међу пакетима веома једноставним.

Компатибилан са *Objective-C*-ом

Цела апликација може бити написана у *Swift*-у, или се *Swift* може користити за додавање нових функционалности у већ постојећи програм. *Swift* и *Objective-C* могу узајамно 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 развијаоцима софтвера (енг. *software developers*) [1], конкретно на адресама *Importing Objective-C into Swift* и *Importing Swift into Objective-C*.

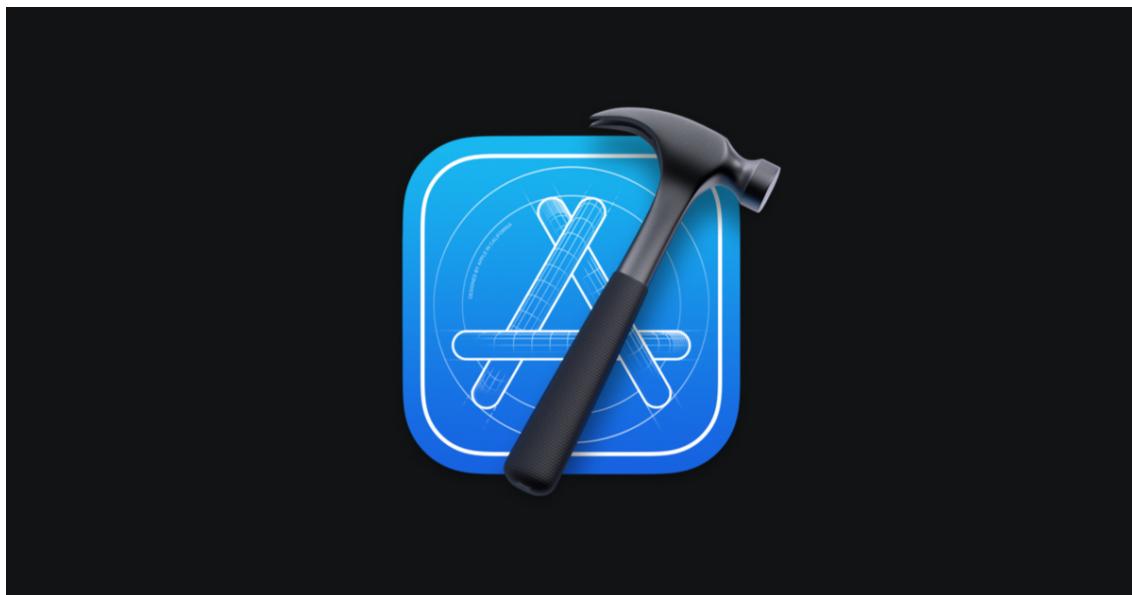
2.5 Xcode

Xcode је интегрисано развојно окружење (ИРО) развијено од стране компаније *Apple*, намењено оперативном систему *macOS*. Софтвер је бесплатан и могуће је преузети га на *Mac App Store*⁵.

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 *Xcode13*. *Xcode* укључује алат командне линије (енг. *Command Line Tools*, *CLT*) који омогућава *UNIX* стил развоја софтвера помоћу терминала. На слици 2.2 може се видети званични лого ИРО-а *Xcode13*.

⁵Платформа која служи за дигиталну дистрибуцију апликација намењених оперативном систему *macOS*



Слика 2.2: Званични лоѓот на Xcode 13

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

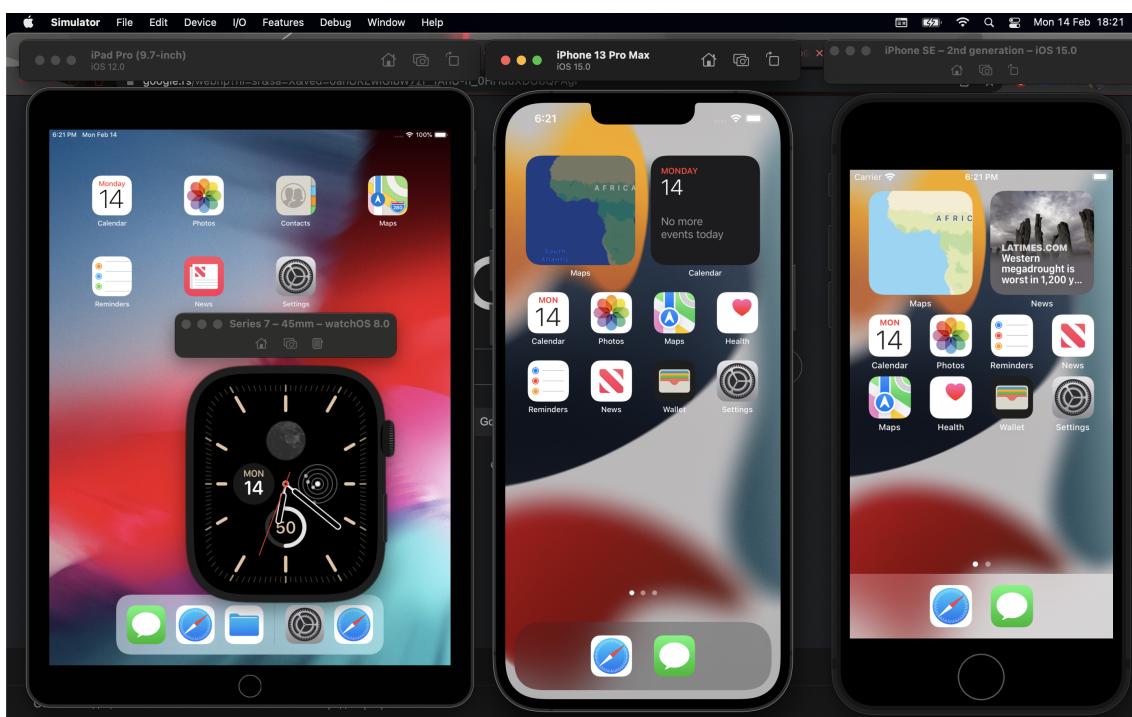
Симулатор

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).

Као што је већ истакнуто, 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 *Apple* платформе (*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

ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

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-a*, 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

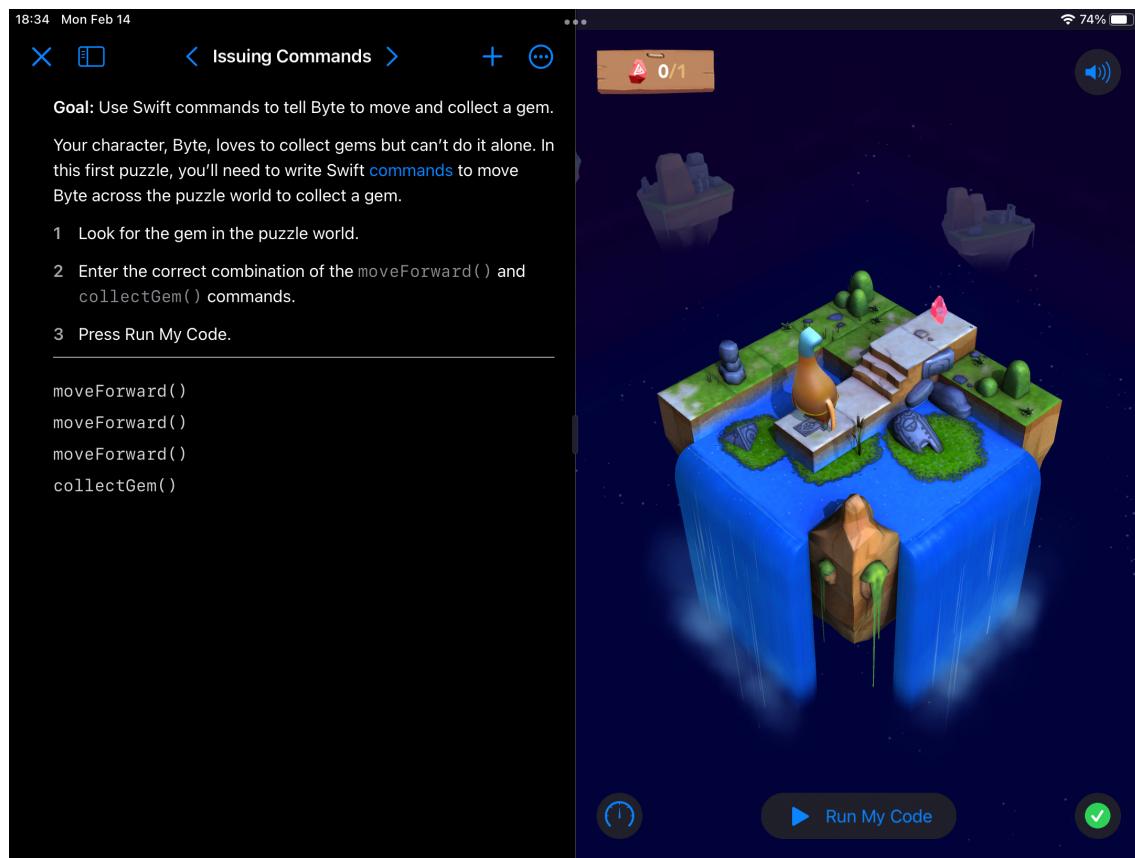
ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

- 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

Swift Playgrounds

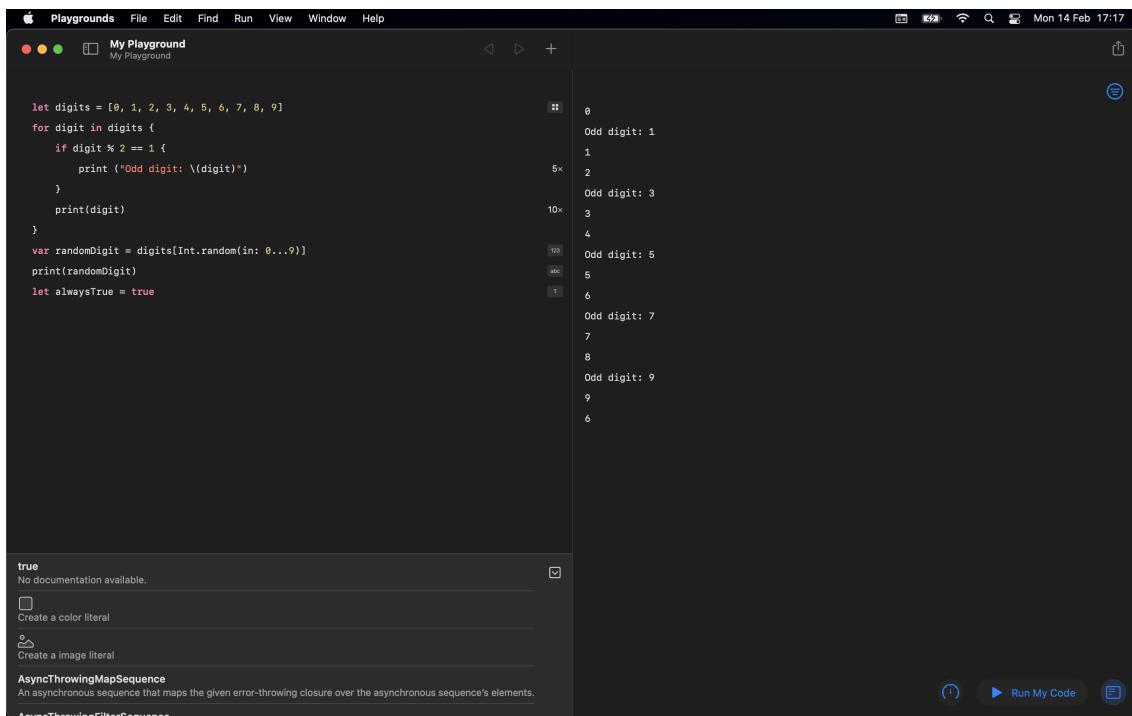
Попут апликације *Swift Playgrounds iPad* чији је један од пројеката намењен почетницима призан на слици 2.4, однедавно постоји апликација *Swift Playgrounds* за оперативни систем *macOS* која је одлична за почетнике, али и за искусније програмере који желе да испробају део кода или се само мало забаве. 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.

ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT



Слика 2.4: Приказ айлайкације Swift Playgrounds на iPad-у

ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT



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

2.6 SwiftUI

Део развоја програмског језика *Swift* је усмерен ка поједностављивању процеса израде корисничког интерфејса и увођењу декларативне синтаксе у језик. У том контексту настало је радно окружење *SwiftUI* које одликује могућност брзог креирања концизних и ефикасних решења.

Уопштењо

SwiftUI је радно окружење које служи за израду апликација са графичким интерфејсом погодним за све *Apple* платформе, користећи моћ програмског језика *Swift* са што мање кода. Омогућава креирање разноврсних апликација уз само један скуп алата и програмског интерфејса апликације (енг. *Application Programming Interface, API*).

Основна структура

У склопу овог радног окружења добија се велики број погледа (енг. *views*), контрола и распоредних структура (енг. *layout structures*) који олакшавају процес израде корисничког интерфејса апликације. Уз то садржи и алате за управљање током података од модела до погледа и контролера, које корисник види и може интераговати са њима преко додира, гестова и других типова улазних података у апликацији који се обрађују помоћу обрађивача догађаја.

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.

Карakteristike

Основна карактеристика која издваја радно окружење *SwiftUI* од радног окружења *UIKit* 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 што једноставније опишу понашање корисничког интерфејса. код је много једноставнији за читање и разумевање као и за писање, чиме је обезбеђена значјана уштеда времена приликом писања новог кода и одржавања већ постојећег. Пример кода у *SwiftUI*-у приказан је у делу 2.38 - Пример *SwiftUI* кода. Модификатор *@State* биће објашњен у делу 2.6 - Стане и ток података.

```
1 // Ucitavanje SwiftUI radnog okruzenja
2 import SwiftUI
3
4 // Kreiranje strukture koja ce sadrzati glavni pogled
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
14          in
15              // Prikaz slike
16              Image(recept.slika)
17              // Definisanje vertikalnog skupa elemenata
18              VStack(alignment: .leading) {
19                  // Prikaz teksta
20                  Text(recept.ime)
21                  // Prikaz teksta sive boje
22                  Text(recept.vremePripreme)
23                      .color(.gray)
24              }
25      }
26 }
```

Пример кода 2.38: 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 да брине о томе. Ово се постиже променљивим стањима и везивањем, чиме се подаци везују за конкретне погледе. Тиме се остварује једини извор истине⁶ (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:

- *State* - Омогућава локално управљање стањем корисничког интерфејса, пример 2.39 - *Омоћачи података - State*. Када је променљива означена као *State*, другом погледу се мора проследити са префиксом '\$' уколико се жели омогућити промена њене вредности

⁶Једини извор истине је начин структуирања информационих модела и шеме података тако да се сваки податак обрађује и мења на само једном месту

ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

- *BindableObject* - Користећи омотач својства *ObservedObject* може се приступити спољашњој референци на модел података који имплементира *ObservableObject* протокол. Уколико је променљива смештена у спољашње окружење, може јој се приступити користећи омотач својства *EnvironmentObject*. Иницијализација посматрајућег (енг. *observable*) објекта директно у погледу постиже се коришћењем модификатора *StateObject*
- *Binding* - Користи се за дељење референце на једини извор истине, пример 2.40 - *Омотачи њога/шака - Binding*
- *Environment* - Подаци сачувани у *Environment-y* се могу делити кроз целу апликацију, пример 2.41 - *Омотачи њога/шака - 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 - *Различићи омотачи њога/шака.*

Data Flow Primitives

	Source of Truth	Derived Value
Read-only	Constant	Property
Read-write	@State BindableObject	@Binding

Слика 2.6: *Различићи омотачи њога/шака*

ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

```
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
9             // oznacava da li je recept medju omiljenima (puna
10            // zvezdica - jeste, prazna - nije)
11            OmiljenRecept(daLije: $daLijeOmiljen)
12        }
13    }
14}
```

Пример кода 2.39: *Омощачи ћогаџака - State*

```
1 struct Recept: View {
2     var recept: ReceptPodatak
3     // Promenljiva 'daLijeOmiljen' je definisana u jednom pogledu, a
4     // moze se menjati u drugom
5     @Binding var daLijeOmiljen: Bool
6
7     var body: some View {
8         Button(action: {
9             // Akcija dugmeta koja menja promenljivu 'daLijeOmiljen',
10            self.daLijeOmiljen.toggle()
11        }) {
12            // Provera promenljive 'daLijeOmiljen' i prikaz
13            // odgovarajuce slike
14            Image(systemName: daLijeOmiljen ? "star.fill" : "star.
15                empty")
16        }
17    }
18}
```

Пример кода 2.40: *Омощачи ћогаџака - Binding*

```
1 struct Kulinarstvo_widgetEntryView : View {
2     var entry: Provider.Entry
3
4     // Cita podatke za 'widgetFamily' iz okruzenja aplikacije i smesta
5     // 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
10     odgovarajuci widget
11     switch widgetFamily {
12         case .systemSmall:
13             RecipeView(recipe: entry.recipe)
14                 .widgetURL(entry.recipe.url)
15         case .systemMedium:
16             RecipeMediumView(recipe: entry.recipe, ingredients: entry.
17                 recipe.ingredients.count > 3 ? Array(entry.recipe.
18                     ingredients.dropLast(entry.recipe.ingredients.count -
19                         3)) : entry.recipe.ingredients)
20         default:
21             Text(" ")
22     }
23 }
```

Пример кода 2.41: *Омощачи ћога ћака - 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 најбоље бити показана на једном конкретном примеру: Форма за пријављивање на одређени сајт, креирање вертикалног скупа елемената, хоризонтално и вертикално центрираних у том скупу, скуп се састоји 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 пона-

ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

вљање поступка за дугме.

За разлику од *UIKit-a*, *SwiftUI* се базира на декларативном начину програмирања и коришћењем радног окружења. *SwiftUI* је доволно 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* и *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. Једноставно се изјасни 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 - *Xcode* - преглед уживо.

Уколико се сагледају архитектуре образаца, може се приметити да се *UIKit* првенствено базира на *MVC*⁷ обрасцу, док *SwiftUI* користи *MVVM*⁸ образац. За заинтересоване читаоце, постоји могућност комбиновања ова два радна окружења и коришћење *SwiftUI*-а унутар *UIKit* кода, или обратно. Ова тема се оставља читаоцима да сами истраже како се то може постићи.

Xcode - преглед уживо

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 пре свега у могућности брзог прегледа измена и то без поновног обнављања (енг. *rebuilding*) апликације, поготово уколико се ради на дода-

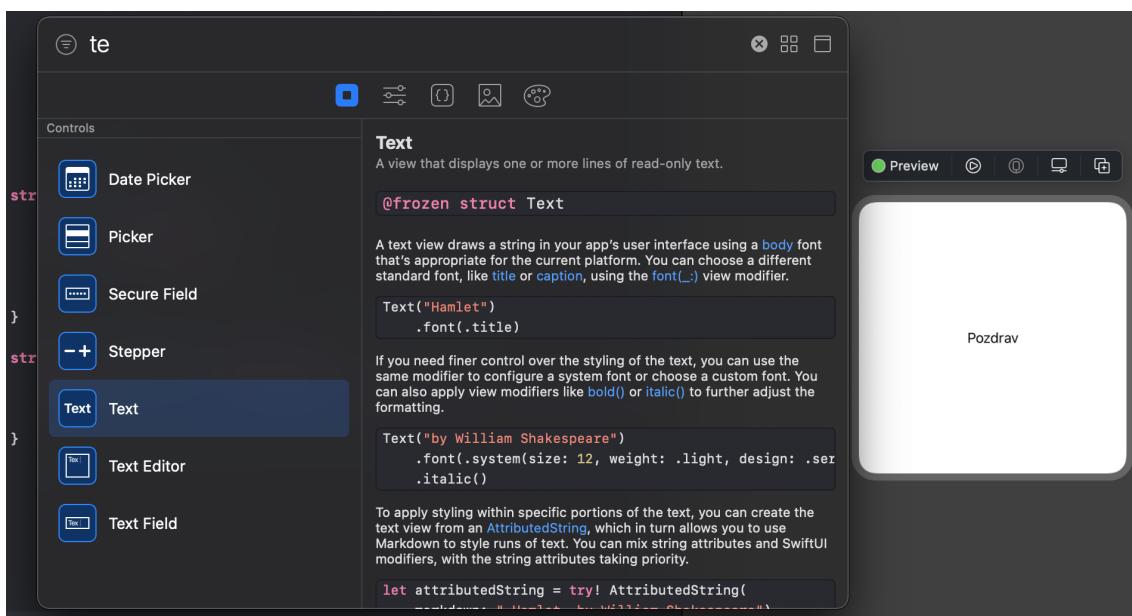
⁷Архитектурни образац Модел-Поглед-Контролер (енг. *Model-View-Controller*) који се заснива на подели на три целине, модел - структура података, поглед - приказ података у корисничком окружењу, контролор - управљање моделом

⁸Архитектурни 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

ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

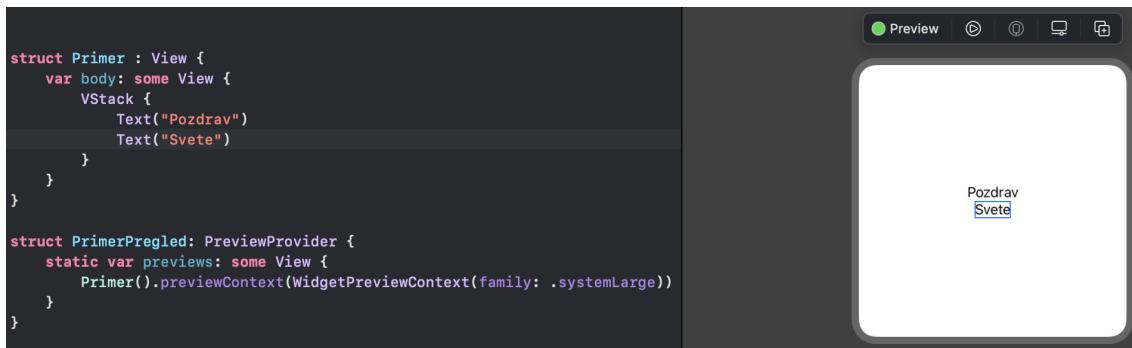
вању или измени погледа који се налази дубоко унутар навигације апликације и за који је потребно више кликова и/или превлачења да би се до њега дошло.

Преглед уживо помаже да се и у *SwiftUI*-у користи метод превлачења и пуштања за креирање корисничког интерфејса, који се разликује од претходног који је коришћен унутар *Storyboard-a*, јер сваки елемент се превлачи у део где се пише код и када се испусти тај елемент постаје део кода. Избор графичких елемената може се видети на слици 2.7, док је код програма и приказ уживо након испуштања графичког елемента *Text* приказан на слици 2.8.



Слика 2.7: Приказ графичких елемената

ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT



Слика 2.8: кôд ћрограма након исyушиштава елеменtа

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 2.42 - *Xcode - uređaj uživo*.

```
1
2     struct PlaceholderView : View {
3         var body : some View {
4             Kulinarstvo_widgetEntryView(entry: SimpleEntry(date: Date
5                 (), configuration: ConfigurationIntent(), recipe:
6                 RecipeModel.testData[0]))
7         }
8
9     // Struktura u kojoj se konfigurise prikaz uzivo
10    struct Kulinarstvo_widget_Previews: PreviewProvider {
11        static var previews: some View {
12            // Grupisanje vise pogleda
13            Group {
14                // Prikaz malog widget-a sa prvim elementom iz liste
15                Kulinarstvo_widgetEntryView(entry: SimpleEntry(date:
16                    Date(), configuration: ConfigurationIntent(),
17                    recipe: RecipeModel.testData[0]))
18                    .previewContext(WidgetPreviewContext(family: .
19                        systemSmall))
20
21            // Prikaz srednjeg widget-a sa skrivenim sadrzajem
22            PlaceholderView()
23        }
24    }
25}
```

ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

```
19         .previewContext(WidgetPreviewContext(family: .  
20                         systemMedium))  
21             .redacted(reason: .placeholder)  
22     }  
23 }
```

Пример кода 2.42: *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. Као што је виђено у примеру кода изнад, 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, виште величина вицета, истих погледа са светлом и тамном бојом позадине, погледа на различитим језицима...

За оне који желе да истраже виште о овој теми, препорука је да одгледају два одлична клипа са *Apple*-ове конференције за програмере из 2019. и 2020. године респективно. Клипови су: '*Mastering Xcode Previews*' и '*Structure your app for SwiftUI previews*'.

Глава 3

Улога и развој виџета

Виџет, 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 виџета у оперативном систему *Android*, који су присутни више од десет година, виџети на Apple платформама су уведени 2020. године, тако да је и сама технологија која подржава њихово креирање и даље у активном развоју.

3.1 Основно

Виџет на уређајима са *Apple* платформом узима један од кључних делова апликације за коју је развијен и приказе га крајњим корисницима тамо где ће га 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 (величина 2x2 места на почетном екрану *iPhone*-а), средњих (2x4) и великих (4x4), а од верзије оперативног система *iPadOS15* екстра великих (4x8) (само за *iPad* уређаје) виџета.

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 испоручити више типова виџета исте величине). Унутар галерије такође постоји опција за измену виџета у којој корисници могу да контролишу и мењају своје виџете и тиме их прилагоде себи, али само уколико је у току конструисања виџета од стране програмера то омогућено. Више речи о овоме биће у делу 3.2 - Развој

вицета.

На оперативним системима *iOS* и *iPadOS* галерија има могућност додања паметних гомила (енг. *smart stack*), које могу садржати до 10 различитих вицета исте величине. Паметна гомила у једном тренутку приказује један од вицета који се налазе у њој. Корисник може сам да мења који ће вицет бити приказан једноставним померањем (енг. *scrolling*). Временом, паметна гомила може научити који вицет корисник ставља на почетак гомиле у току дана (или недеље) и сама мењати примарне вицете у одређеном тренутку (на пример, након гашења аларма прво се приказује вицет са временском прогнозом, па најновије вести, стање у саобраћају...)

Siri асистент¹ може и сам додати вицете у паметну гомилу уколико претпостави да постоји неки вицет који би кориснику био користан. Након тога корисник сам одлучује да ли жели да новододати вицет остане у паметној гомили или не.

WidgetKit

WidgetKit је радно окружење које уз *widget API* из *SwiftUI*-а служи за израду вицета, од његовог изгледа, преко временског ажурирања па све до омогућавања конфигурације вицета од стране крајњих корисника и управљања паметном гомилом приликом ротације вицета од стране система. Још једна могућност коју ово радно окружење пружа је повезивање апликације и самог вицета, што омогућава кориснику да отвори апликацију притиском на вицет и аутоматски оде на одговарајући поглед из вицета када жели да види детаљније податке. Мора се обратити пажња код оваквог начина комуникације јер вицет не би смео да служи само као пречица за покретање апликације, више о томе биће објашњено у делу 3.3 - Дизајн вицета.

3.2 Развој вицета

Вицет је ништа друго до заправо само један *SwiftUI* поглед. Вицети су тренутно једини део оперативних система *Apple* платформа који у потпуности морају бити написани коришћењем радног окружења *SwiftUI*. *Apple* је отпочетка развоја вицета имао на уму овакву идеју, због начина приказивања и

¹Интелигентни лични асистент на уређајима са *Apple* платформом

ГЛАВА 3. УЛОГА И РАЗВОЈ ВИЦЕТА

само повременог ажурирања података, као и немогућности корисничке интеракције са самим вицетима (осим једноставног клика којим се отвара одређени део апликације).

Додавање вицет додатка апликацији

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

Кораци за креирање вицет 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 ради на пројекту
4. Уколико вицет подржава конфигурацију од стране корисника, штиклирати поље *Include Configuration Intent*
5. Кликнути на дугме *Finish*

Додавање детаља конфигурације

Као што је већ напоменуто, шаблон вицет додатка пружа иницијалну имплементацију вицета која имплементира *Widget* протокол. Два могућа начина конфигурације вицета су статичка (енг. *StaticConfiguration*) и конфигурација са сврхом (енг. *IntentConfiguration*).

Статичка конфигурација се користи за вицете који немају параметре који могу бити конфигурисани од стране корисника (на пример, системска апликација *Screen time* која води статистику о времену проведеном на одговарајућем уређају). Конфигурација са сврхом се користи за вицете чији одређени параметри могу бити конфигурисани од стране корисника (на пример, системска апликација за временску прогнозу где корисник може наместити одређени

ГЛАВА 3. УЛОГА И РАЗВОЈ ВИЦЕТА

град за који жели да добија податке). Ова конфигурација ће бити укључена и конфигурациони фајл ће бити додат уколико програмер приликом додања вицета додатка штиклира поље *Include Configuration Intent*.

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

- Тип (енг. *Kind*), стринг који идентификује вицет, требао би да казује шта вицет представља
- Снабдевач (енг. *Provider*), објекат класе која имплементира протокол *TimelineProvider* и кроз временску линију коју производи одређује у ком тренутку ће вицет бити поново изрендерован и нови подаци бити приказани. Више о овом протоколу и свеукупној причи о временој линији 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 вицета
- Прилагођена сврха (енг. *Custom Intent*), фајл који дефинише параметре које корисник може мењати и прилагођавати себи, више о овоме у делу: 3.2 - *Intent*

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

```
1 struct KulinarstvoSecondWidget: Widget {
2     @Environment(\.colorScheme) var colorScheme
3
4     let kind: String = "KulinarstvoSlasnoIEfikasnoSecondWidget"
5
6     var body: some WidgetConfiguration {
7         StaticConfiguration(kind: kind, provider: SecondProvider()
8             ) { entry in
9                 KulinarstvoSecondWidgetEntryView(entry: entry)
10            }
11            .configurationDisplayName("Recept na klik")
```

ГЛАВА 3. УЛОГА И РАЗВОЈ ВИЦЕТА

```
11     .description("Dodaj svoje omiljene recepte na pocetni  
12         ekran")  
13     .supportedFamilies([.systemLarge])  
14 }
```

Пример кода 3.1: *вицећ - 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 *widget* 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 за приказ виџета треба да буду приказани и одређени подаци, а подаци нису пристигли са серверске стране, 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 - *вицећ - placeholder* може се видети креирање погледа чувара места (енг. *placeholder view*) коришћењем статичких података који су увек доступни, и конкретан приказ тог чувара места у прегледу уживо са сакривеним подацима (енг. *redacted data*) - приказ како ће корисник видети виџет док не пристигну конкретни подаци у неким ситуацијама.

```
1 struct PlaceholderView : View {  
2     var body : some View {  
3         Kulinarstvo_widgetEntryView(  
4             entry: SimpleEntry(date: Date(), configuration:  
5                 ConfigurationIntent(), recipe: Datafeed.shared.  
6                 favRecipes[0], parameterToShow: MainParameter.  
7                 Sastojci.rawValue))  
8     }  
9 }
```

ГЛАВА 3. УЛОГА И РАЗВОЈ ВИЦЕТА

```
7
8     struct Kulinarstvo_widget_Previews: PreviewProvider {
9         static var previews: some View {
10            PlaceholderView()
11                .previewContext(WidgetPreviewContext(family: .
12                                systemLarge))
13                .redacted(reason: .placeholder)
14        }
15    }
```

Пример кода 3.2: *вишет - placeholder*

Када пристигну подаци са сервера, снабдевач добија обавештење, сакупља реалне податке и приказује вицет са њима. Након што корисник дода вицет на почетни екран и буде приказан иницијални снимак изгледа вицета, *WidgetKit* позива функцију *getTimeline* из провајдера, чиме захтева временску линију.

Intent

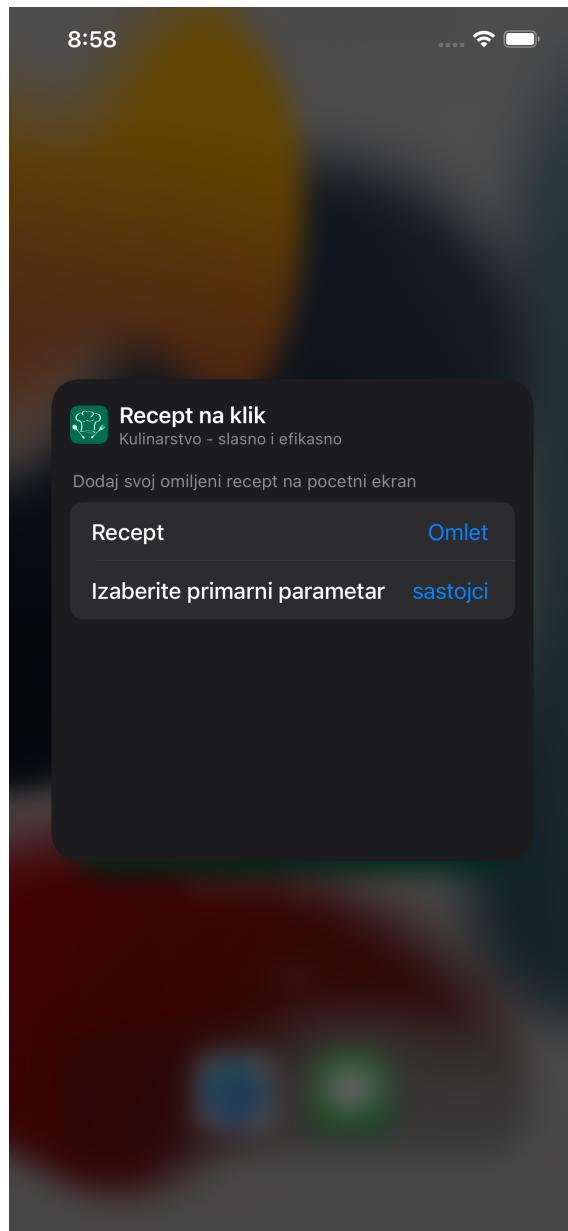
Вицети представљају погледе који не интерагују са корисницима, односно не подржавају интерактивне елементе, као што су поглед *Scroll* и дугме *Switch*. Једна врста интеракције корисника са вицетом постиже се омогућавањем конфигурације вицета од стране корисника коришћењем конфигурације *Intent*, у којој се наводе сви параметри које корисник може да промени (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
- Коришћење протокола *IntentTimelineProvider* уместо протокола *TimelineProvider* као снабдевача временске линије, да би конфигурација параметара од стране корисника била сачувана у уносима временске линије
- Уколико параметри зависе од динамичких података потребно је имплементирати екstenзију *Intent*-а

На слици 3.1 се може видети како изгледа конфигурација једног вицета са два параметра, први за избор рецепта који ће у вицету бити приказан и избор примарног параметра уз опис рецепта (састојци или припрема).

ГЛАВА 3. УЛОГА И РАЗВОЈ ВИЦЕТА



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, и може се конфигурисати који део апликације ће бити приказан кориснику у зависности од елемента унутар виџета на који је кликнуо. Свим величинама виџета може бити додат моди-

ГЛАВА 3. УЛОГА И РАЗВОЈ ВИЦЕТА

модификатор `widgetURL(_ :)`, којим се одређује у који део апликације ће корисник бити одведен када кликне на вицет.

За све величине вицета, осим малих, може се користити и веза (енг. *Link*) која се додаје једном погледу унутар вицета, којом је одређено место у апликацији које ће бити отворено (на пример, један вицет средње величине који садржи листу са 3 рецепата, сваки елемент листе има везу која води ка детаљној страни о рецепту који тај елемент представља). Иако вицет користи везе унутар својих елемената, може користити и модификатор `widgetURL(_ :)`. Овај модификатор ће бити активиран уколико корисник кликне на поглед унутар вицета који нема дефинисану везу и биће отворена апликација без додатне навигације од стране вицета. 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:
10                     true)
11                     .widgetURL(entry.recipe.url)
12             case .systemMedium:
13                 Link(destination: entry.recipe.url ?? URL(
14                     fileURLWithPath: ""))
15                     RecipeMediumView(recipe: entry.recipe, listName:
16                         entry.parameterToShow)
17             case .systemLarge:
18                 Link(destination: entry.recipe.url ?? URL(
19                     fileURLWithPath: ""))
20                     RecipeLargeView(recipe: entry.recipe,
21                         mainParameter: entry.parameterToShow)
22             default:
23                 Text("")
24         }
25     }
26 }
```

Пример кода 3.3: *Везе унущар вишећа*

Више виџета у једном проширењу

Уколико постоји потреба за коришћењем више различитих типова виџета у једном проширењу, то се може лако постићи уз само пар измена главног дела проширења, означеног анотацијом `@main`. Уместо протокола `Widget` главна структура мора имплементирати протокол `WidgetBundle`. Тело структуре сада имплементира протокол `Widget` и додата је анотација `@WidgetBundleBuilder`. Приказ употребе више типова виџета у једном проширењу може се видети у примеру 3.4 - *Више вијећа у једном проширењу*.

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

Пример кода 3.4: *Више вијећа у једном проширењу*

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 дозволио 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)

ГЛАВА 3. УЛОГА И РАЗВОЈ ВИЦЕТА

и да буду временски и кориснички релевантни. Први корак у дизајну вицета је избор дела апликације који ће тај вицет представљати.

Свака величина вицета 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 вицета приказују исте податке, али истовремено приликом додавања нових података се мора водити рачуна о почетној идеји, односно делу апликације које тај вицет треба да представља. Уколико не постоји довољна количина података за веће вицете одређене величине могу бити искључене из понуде кориснику.

Вицет не би смео да служи само као пречица за покретање апликације. Корисници очекују од сваког вицета 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).

Ажурирни подаци

Да би вицети могли да пружају корисне и прецизне информације морају бити ажурирани. Вицети не подржавају ажурирање у реалном времену, а и сам систем може ограничити ажурирање вицета у зависности од корисничког понашања и интеракције са њим, па се мора пронаћи начин на који ће подаци у вицету увек бити релевантни.

Потребно је пронаћи оптимално време за ажурирање података у вицету, узимајући у обзир колико се сами подаци које вицет 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, вицет 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 вицет треба да омогући кориснику конфигурабилност како би могао да пружи релевантне информације (на пример, књига коју корисник тренутно чита и његов прогрес у апликацији *Apple Books*), док поједини вицети могу то изоставити (на пример, најновије вести). Уколико

ГЛАВА 3. УЛОГА И РАЗВОЈ ВИЦЕТА

је вицет конфигурабилан, потребно је да подешавања буду једноставна и да се не захтева превише информација од корисника. Кориснички интерфејс за измену вицета је унапред одређен и исти за све вицете, као што је показано у делу 3.2 - *Intent*.

Дизајн прилагођен свима

Вицети треба да буду јарких боја како би се истицали на екрану, али истовремено и јасно видљив текст како би корисник могао да види све потребне информације након откључавања или непосредно пре закључавања уређаја. Вицет 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 вицета који се може боље искористити.

Количина информација која ће бити приказана у вицету мора бити оптимална. Уколико се прикаже премало информација вицет неће имати превелики значај за кориснике, док превише информација на мало простора отежава читање и разумевање података.

Једна од ставки која се не сме заборавити у данашње време је израда дизајна вицета за обе врсте боја системске позадине (светле и тамне). Дизајн вицета 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 недостатаکа.

Apple саветује да се никад не користи фонт текста мањи од 11 поена². Коришћење мањег фонта би корисницима знатно отежало употребу вицета. Увек треба користити званичне елементе за приказ текста, како би се омогућила склабилност као и системско читање текста.

Пажњу треба обратити на дизајн прегледа вицета унутар галерије, за све типове и величине који вицет подржава, као и приказ чвара 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 вицету потребно је да имају исту функционалност јер би у

²Apple-ов израз за „број који треба уписати у поље”, универзална мера у дизајну на Apple платформама

ГЛАВА 3. УЛОГА И РАЗВОЈ ВИЏЕТА

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

Опис 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 вицет који је имплементиран у склопу апликације може послужити као подсетник за потребне састојке или кораке припреме и истовремено бити употребљен за предлагање наредних оброка.

Рад апликације ће бити приказан упоредо на два симулатора, *iPhone 13 Pro Max* са тамном бојом позадине (у наставку означен бројем 1) и *iPhone SE (2nd generation)* са светлом бојом позадине (у наставку означен бројем 2). Оба симулатора покрећу оперативни систем *iOS 15.0*.

Почетни экран

На почетном экрану 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(е).

Рецепти су приказани унутар табеле, сваки рецепт је презентован именом, временом припреме у минутима и сликом. Приказ почетног екрана може се видети на сликама 4.1 и 4.2.

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”,

ГЛАВА 4. ОПИС АПЛИКАЦИЈЕ



Слика 4.1: Почекани екран - 1

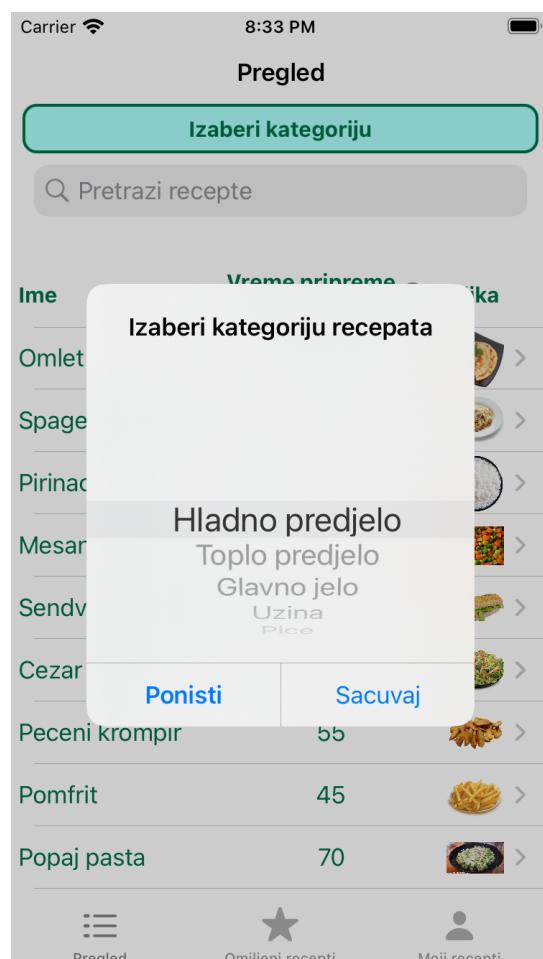


Слика 4.2: Почекани екран - 2

ГЛАВА 4. ОПИС АПЛИКАЦИЈЕ



Слика 4.3: Кашеторија - 1



Слика 4.4: Кашеторија - 2

, „Салата“ и „Хлеба“. Приказ избора категорије налази се на сликама 4.3 и 4.4, док је на сликама 4.5 и 4.6 приказан изглед екрана када је једна категорија изабрана (конкретно категорија „Главно јело“).

Поред избора категорије, корисник може филтрирати приказане рецепате претрагом по имениу 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 сликама 4.7 и 4.8.

Рецепте на почетној страни корисник може и сортирати, опадајуће или

ГЛАВА 4. ОПИС АПЛИКАЦИЈЕ

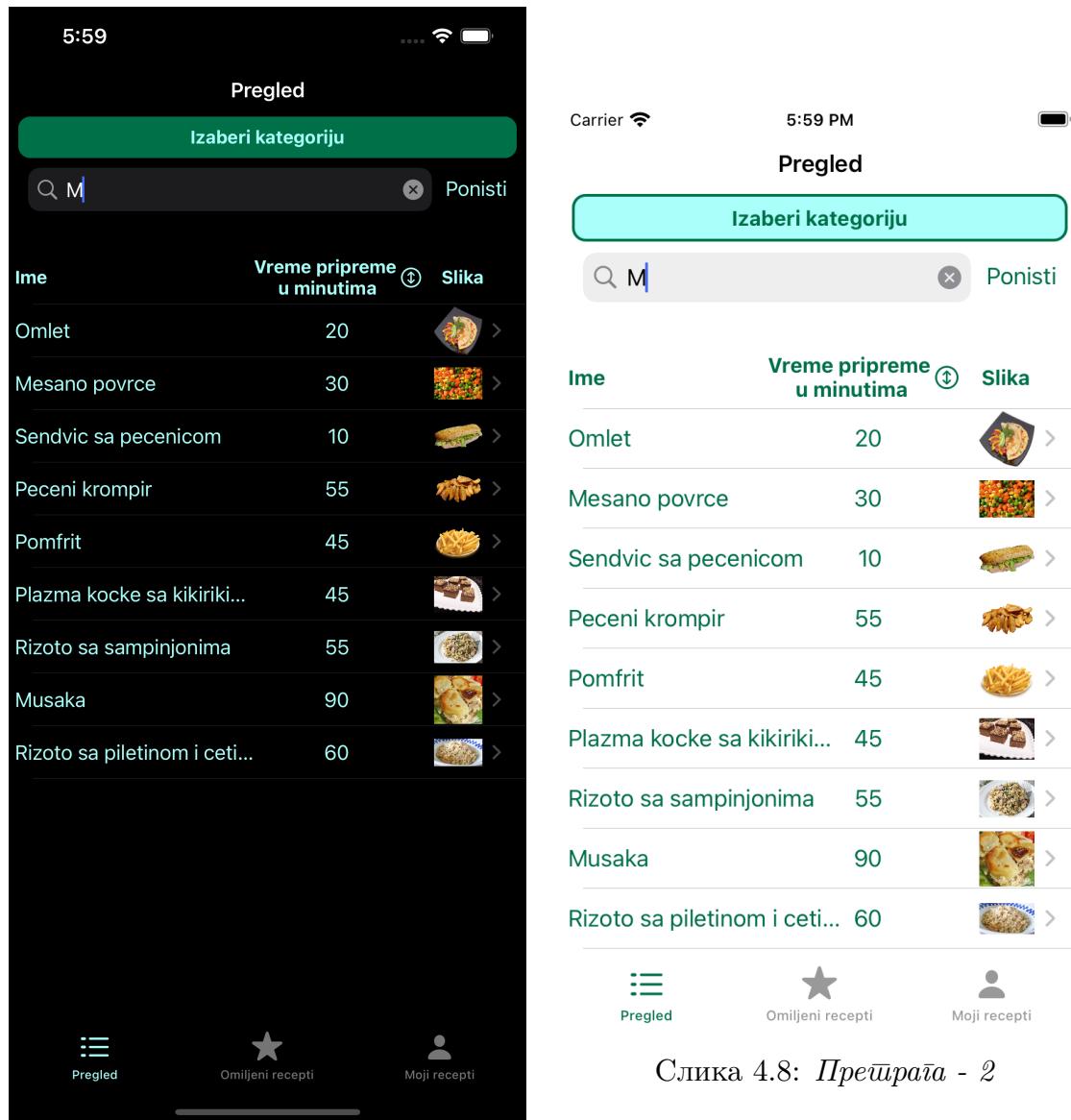


Слика 4.5: Изабрана кашејорија - 1



Слика 4.6: Изабрана кашејорија - 2

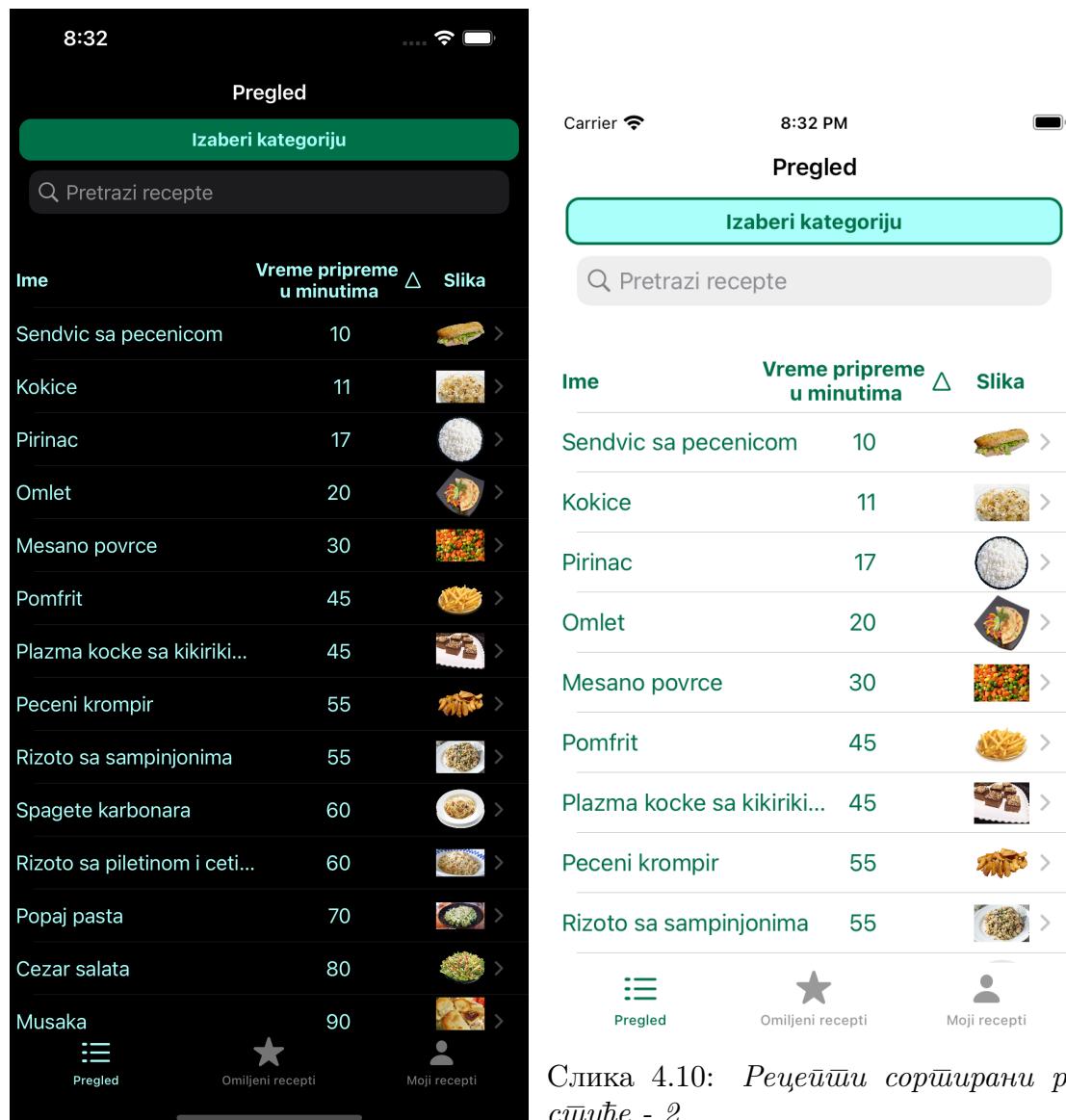
ГЛАВА 4. ОПИС АПЛИКАЦИЈЕ



Слика 4.7: Пређураћа - 1

Слика 4.8: Пређураћа - 2

ГЛАВА 4. ОПИС АПЛИКАЦИЈЕ



Слика 4.9: Рецепти су сортирани по врему припреме - 1

Слика 4.10: Рецепти су сортирани по називу - 2

растуће по укупном времену потребном за њихово спремање (припрема и кување). 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) који се спрема најбрже од понуђених. Сортирани рецепти у растућем редоследу су приказани на сликама 4.9 и 4.10.

ГЛАВА 4. ОПИС АПЛИКАЦИЈЕ

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. Слике 4.11 и 4.12 представљају изглед дела „Омиљени рецепти”.

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 и да манипулише, више о овоме биће објашњено у делу 4 - Измена и брисање постојећег рецепта. Још једна могућност која се пружа кориснику на овој страни је додавање новог рецепта, што ће детаљно бити објашњено у делу 4 - Креирање новог рецепта. Сликама 4.13 и 4.14 приказан је изглед странице „Моји рецепти”.

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, могућност повећавања и смањивања броја особа за које је рецепт предвиђен (истовремено ће бити промењена количина састојака као и време припреме), списак састојака и корака припреме, могућност додавања и брисања рецепта из листе омиљених, и уколико је приказани рецепт креиран од стране тренутног корисника, може га изменити или обрисати. Детаљан приказ рецепта за цезар салату приказан је на slikama 4.15 и 4.16 за четири особе односно на slikama 4.17 и 4.18 за шест особа.

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 slikama 4.19 и 4.20.

ГЛАВА 4. ОПИС АПЛИКАЦИЈЕ



Слика 4.11: *Омиљени рецети - 1*



Слика 4.12: *Омиљени рецети - 2*

ГЛАВА 4. ОПИС АПЛИКАЦИЈЕ

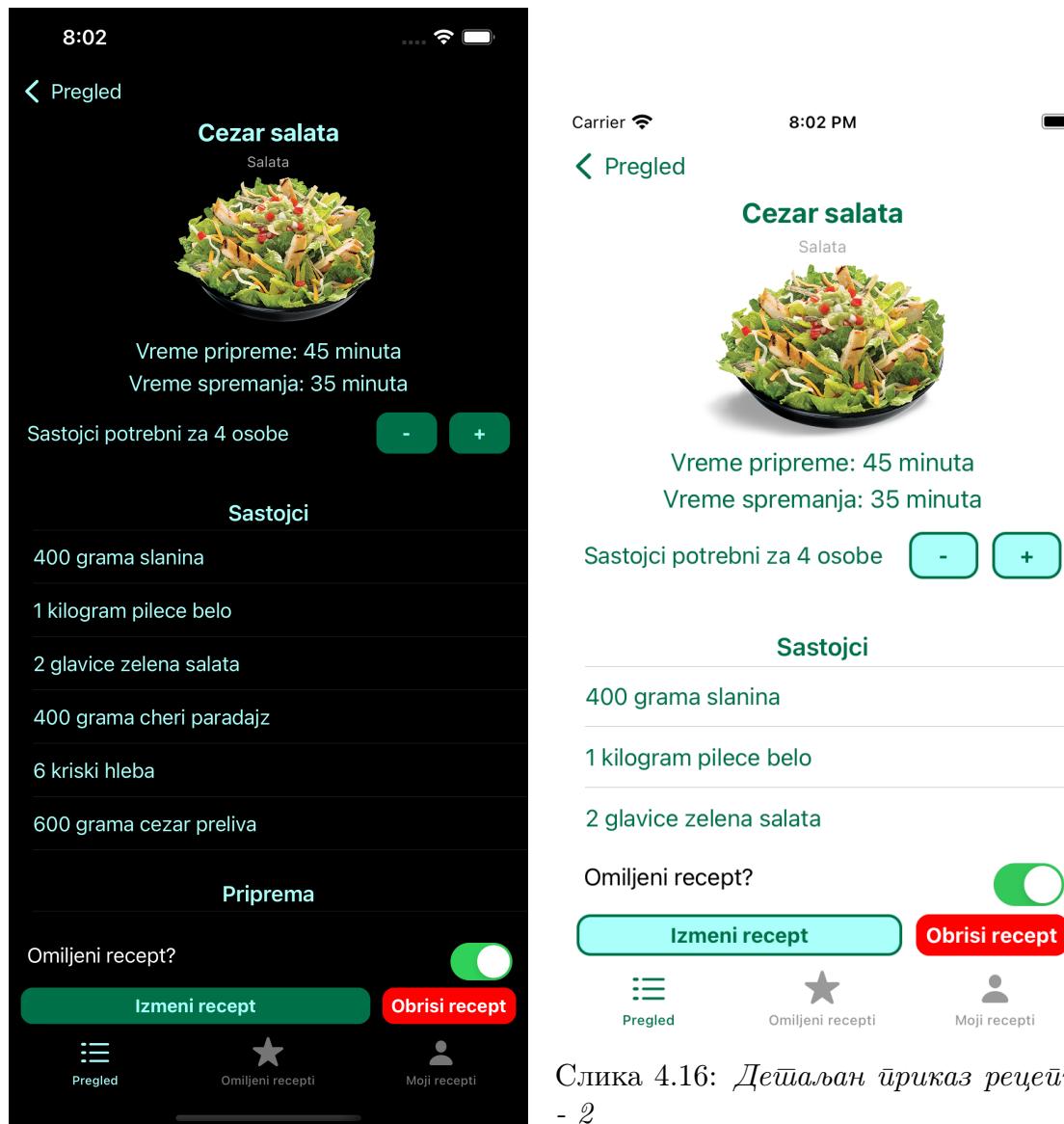


Слика 4.13: *Moju rečepćinu - 1*



Слика 4.14: *Moju rečepćinu - 2*

ГЛАВА 4. ОПИС АПЛИКАЦИЈЕ

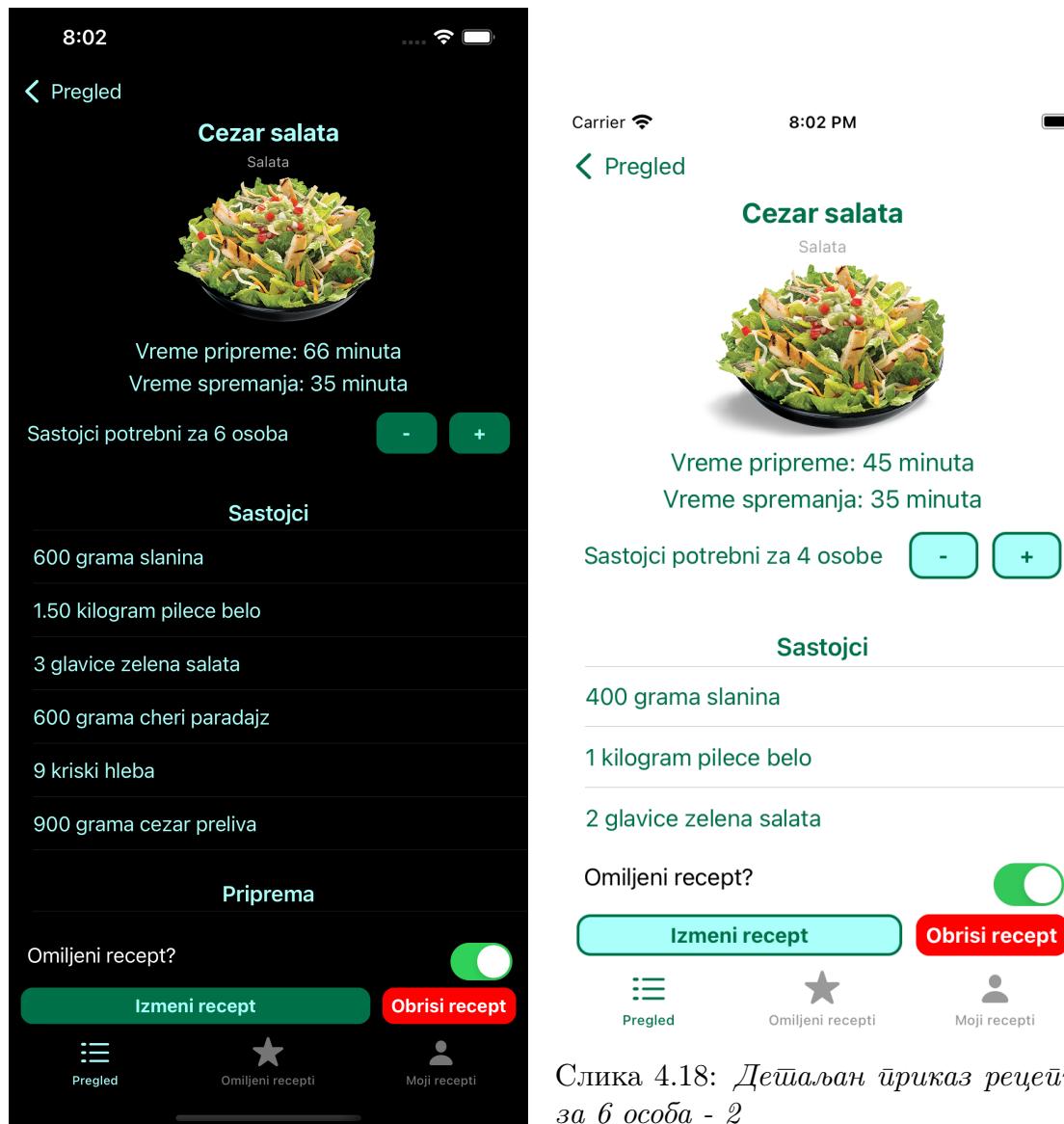


Слика 4.15: Дештањан јриказ рецета

- 1

Слика 4.16: Дештањан јриказ рецета
- 2

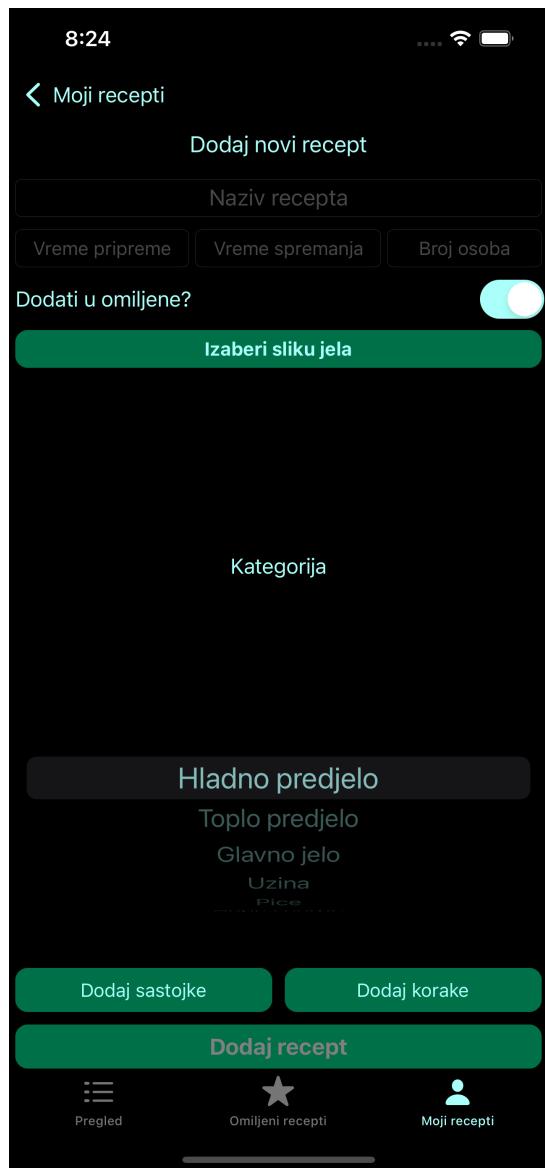
ГЛАВА 4. ОПИС АПЛИКАЦИЈЕ



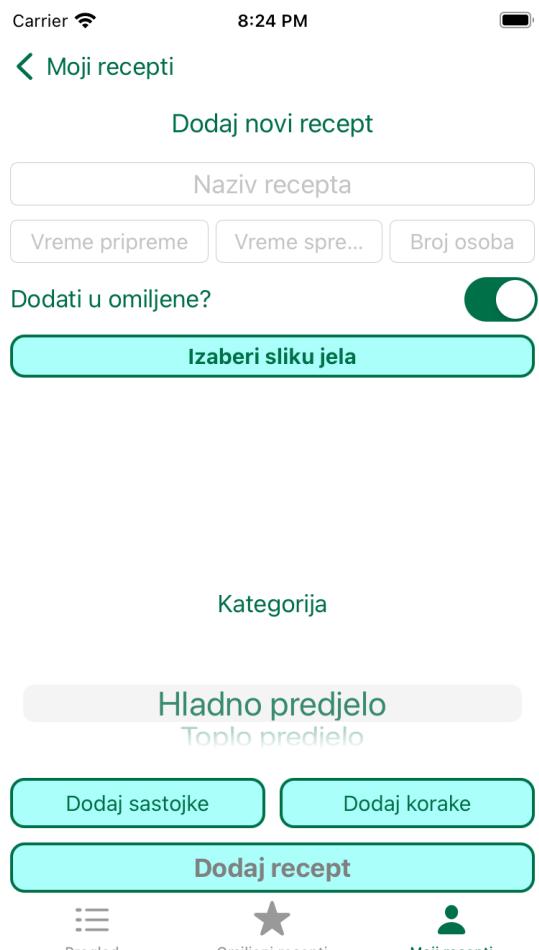
Слика 4.17: Дештањан јриказ рецептија за 6 особа - 1

Слика 4.18: Дештањан јриказ рецептија за 6 особа - 2

ГЛАВА 4. ОПИС АПЛИКАЦИЈЕ



Слика 4.19: *Нов рецепт* - 1

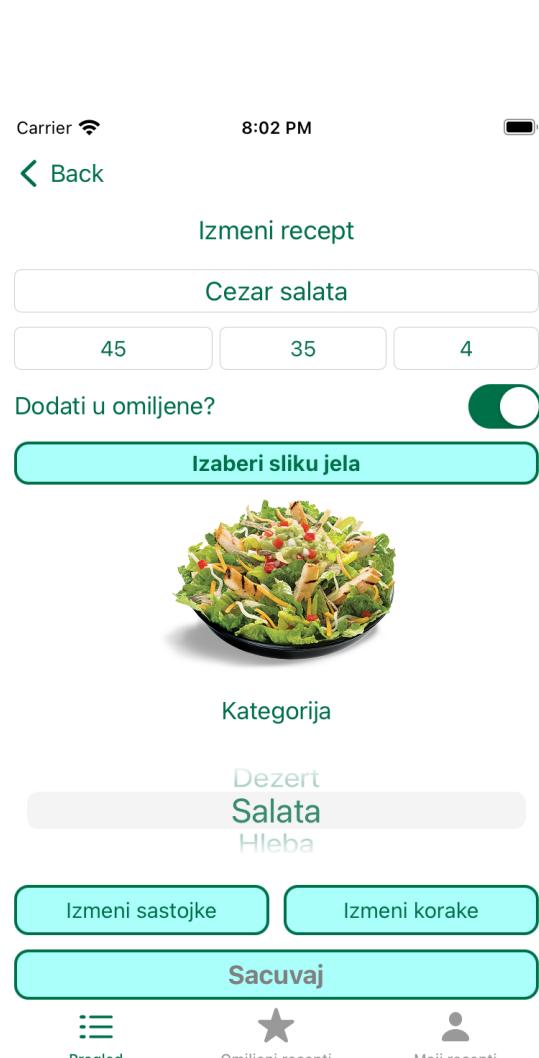


Слика 4.20: *Нов рецепт* - 2

ГЛАВА 4. ОПИС АПЛИКАЦИЈЕ



Слика 4.21: Измена рецепта - 1



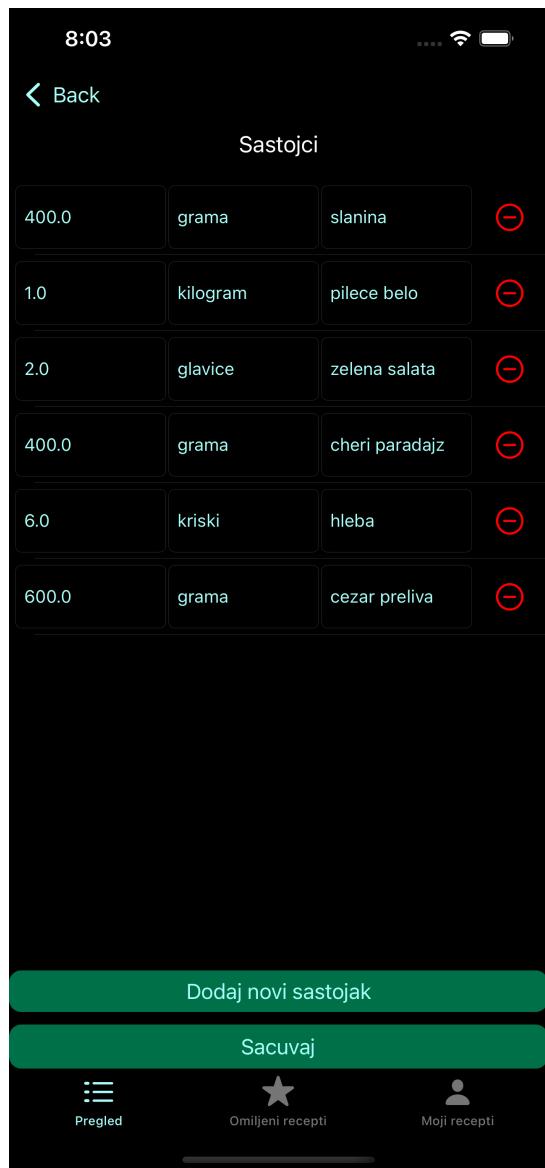
Слика 4.22: Измена рецепта - 2

Измена и брисање постојећег рецепта

Корисник има могућност промене свих рецепата које је додао. Приказ изгледа погледа измене рецепта налази се на сликама 4.21 и 4.22. Постоји могућност измене и брисања састојака 4.23 и 4.24, додавања нових састојака 4.25 и 4.26, измене и брисања корака припреме 4.27 и 4.28 и додавања нових корака припреме 4.29 и 4.30.

Уколико корисник жели може и обрисати рецепте које је додао, кликом на дугме „Обриши рецепт“ унутар детаљног приказа рецепта, након чега ће му

ГЛАВА 4. ОПИС АПЛИКАЦИЈЕ

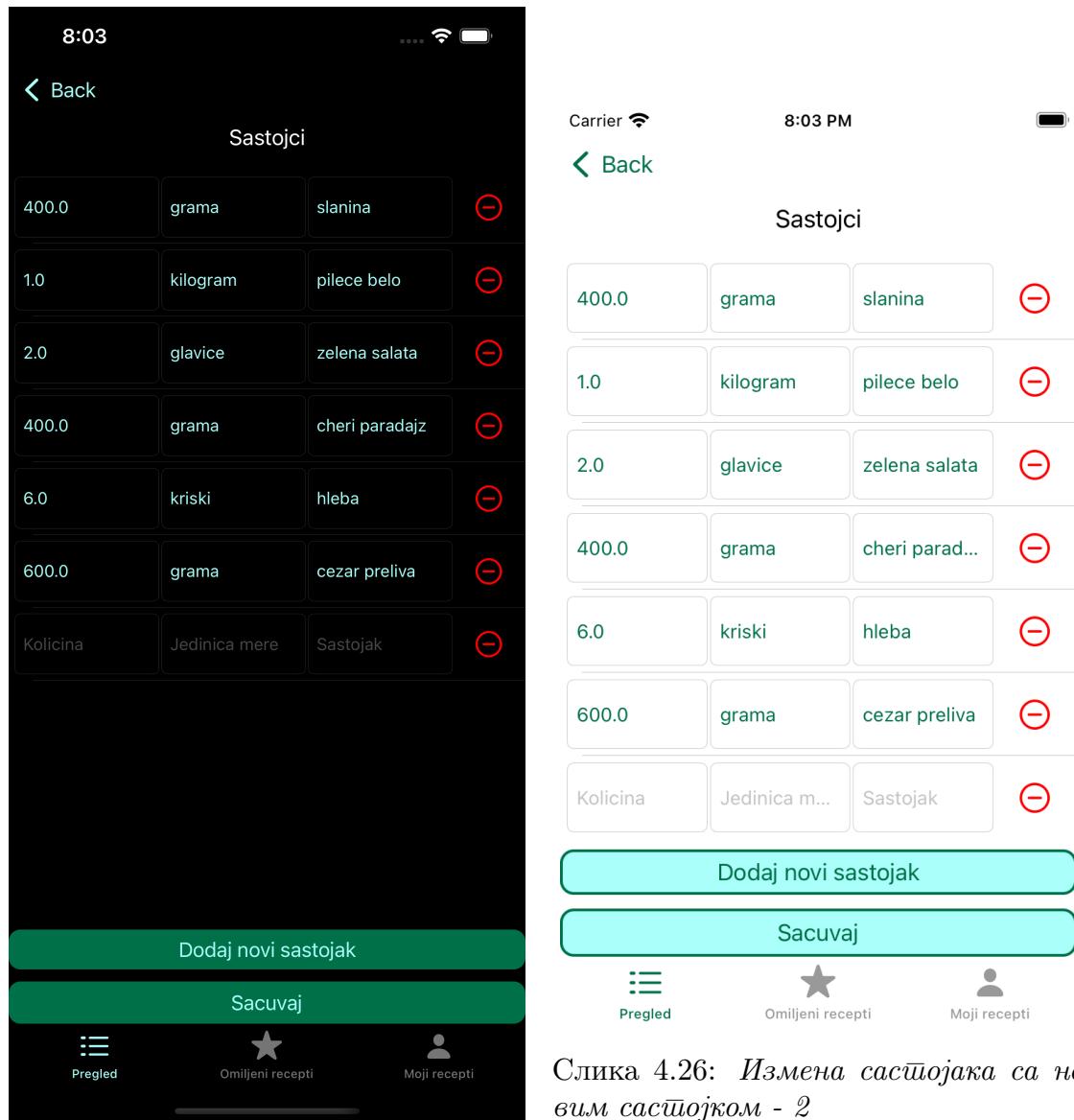


Слика 4.23: Измена састојојака - 1



Слика 4.24: Измена састојојака - 2

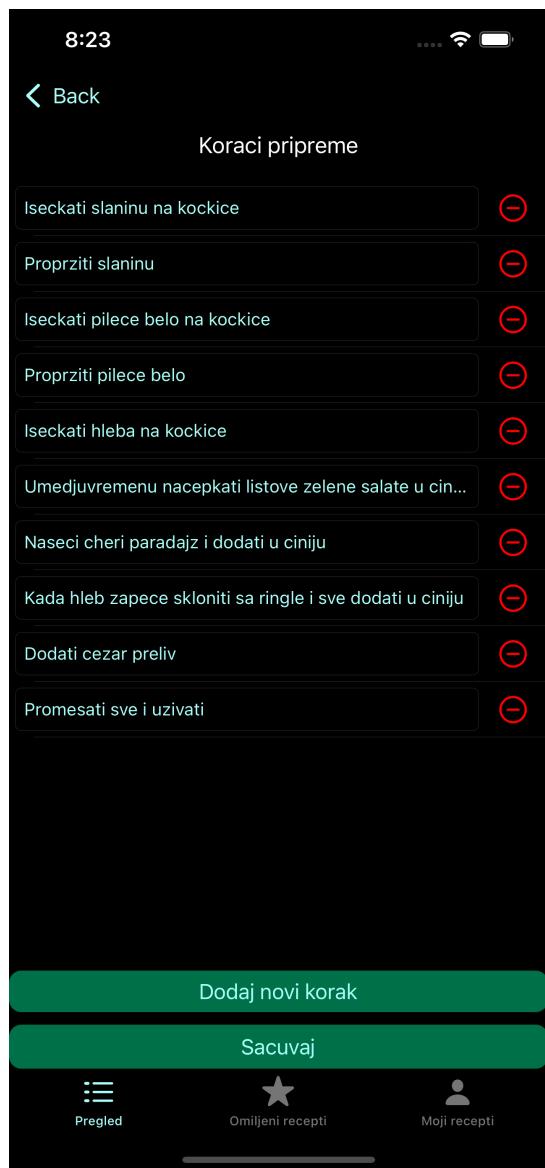
ГЛАВА 4. ОПИС АПЛИКАЦИЈЕ



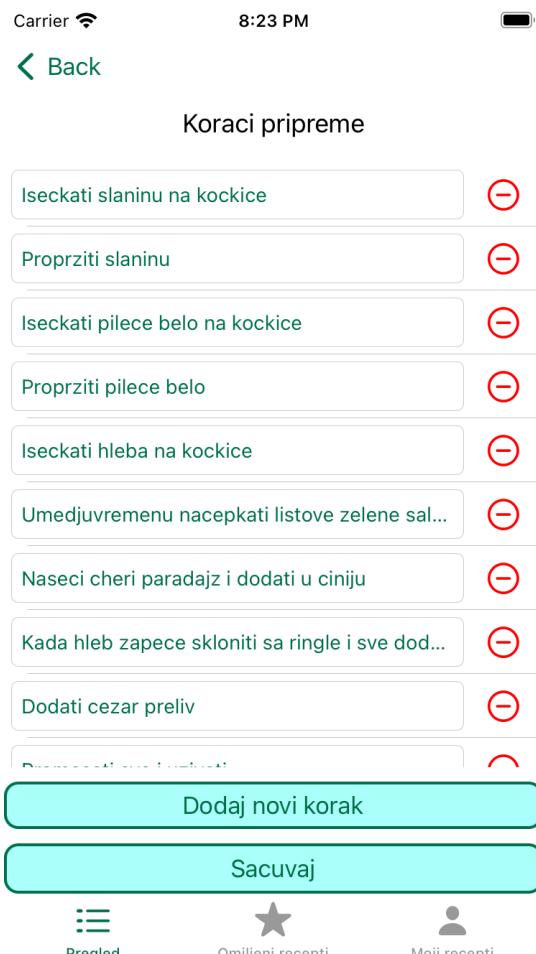
Слика 4.25: Измена састојака са новим састојком - 1

Слика 4.26: Измена састојака са новим састојком - 2

ГЛАВА 4. ОПИС АПЛИКАЦИЈЕ

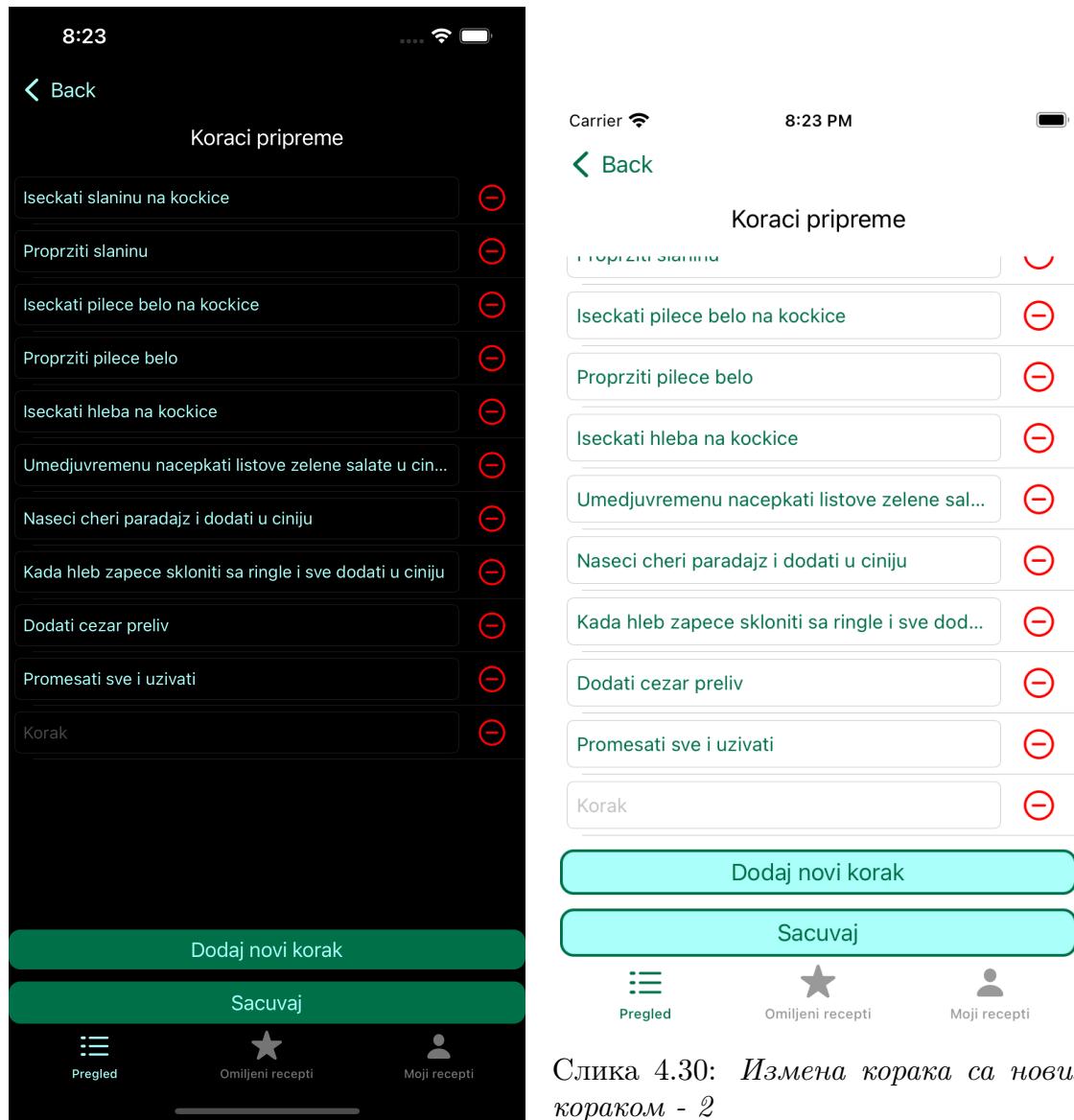


Слика 4.27: Измена корака - 1



Слика 4.28: Измена корака - 2

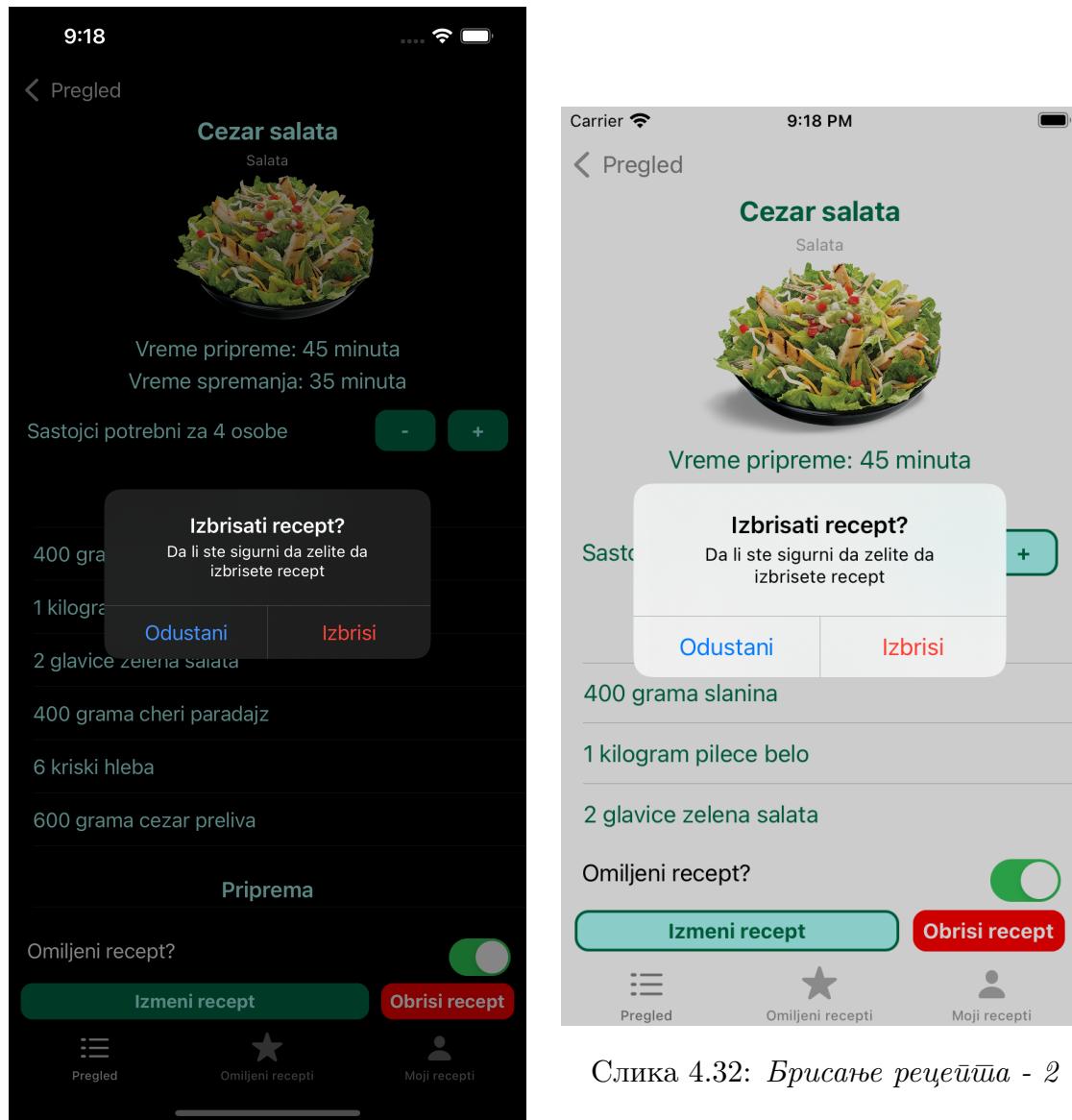
ГЛАВА 4. ОПИС АПЛИКАЦИЈЕ



Слика 4.30: Измена корака са новим кораком - 2

Слика 4.29: Измена корака са новим кораком - 1

ГЛАВА 4. ОПИС АПЛИКАЦИЈЕ



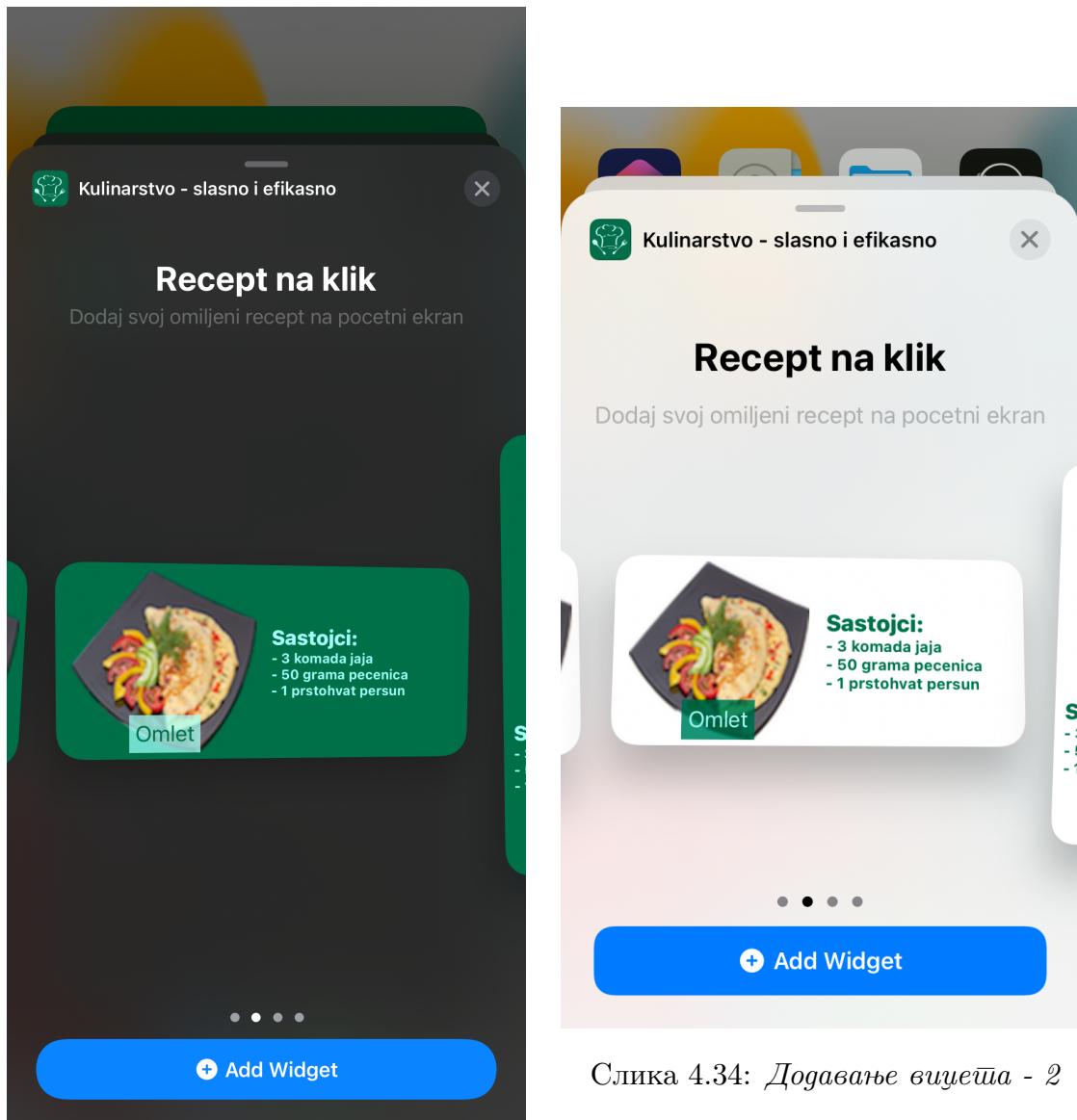
Слика 4.31: Брисање рецептија - 1

бити приказано упозорење у којем ће моћи да потврди брисање рецепта или да од њега одустане. Приказ упозорења приликом брисања рецепта налази се на сликама 4.31 и 4.32.

Приказ вицета

Корисник може додати вицет на почетни екран из вицет галерије, где се налазе оба типа вицета ове апликације (први тип у све три величине,

ГЛАВА 4. ОПИС АПЛИКАЦИЈЕ



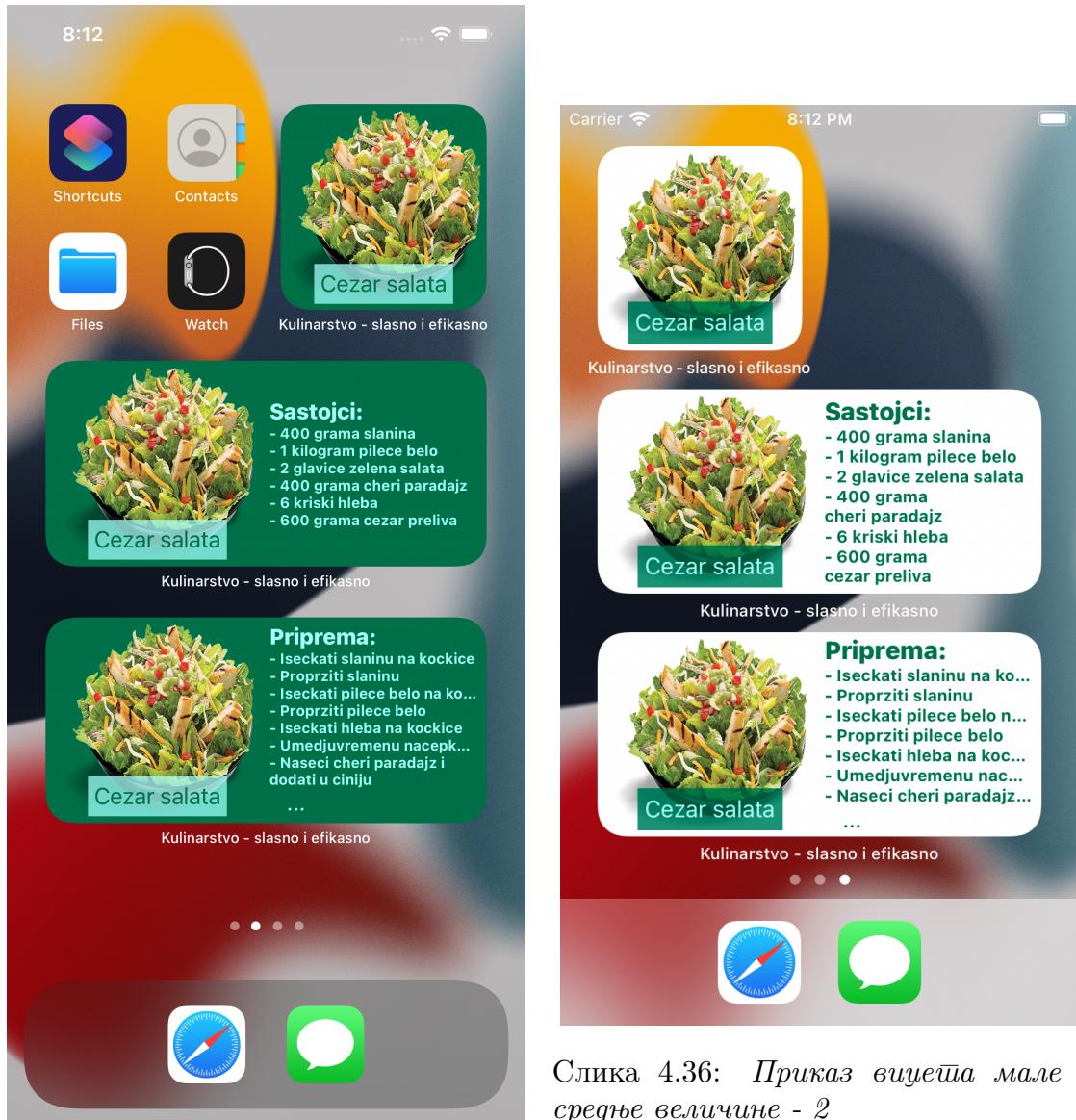
Слика 4.33: Додавање виџета - 1

Слика 4.34: Додавање виџета - 2

док је други представљен само у великој величини) који ће детаљније бити објашњени у наставку. Приказ додавања виџета из галерије представљен је на сликама 4.33 и 4.34.

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

ГЛАВА 4. ОПИС АПЛИКАЦИЈЕ



Слика 4.36: Приказ вијећа мале и средње величине - 2

Слика 4.35: Приказ вијећа мале и средње величине - 1

рецепта, параметар који је конфигурабилан од стране корисника. Приказ малог и средњег виџета може се видети на сликама 4.35 и 4.36.

Први тип виџета у великој величини приказује рецепт са листом састојака и корака припреме, који су као код средњег виџета 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). Велики виџет првог типа приказан је на сликама 4.37 и

ГЛАВА 4. ОПИС АПЛИКАЦИЈЕ



Слика 4.37: Приказ првој вишећа велике величине - 1

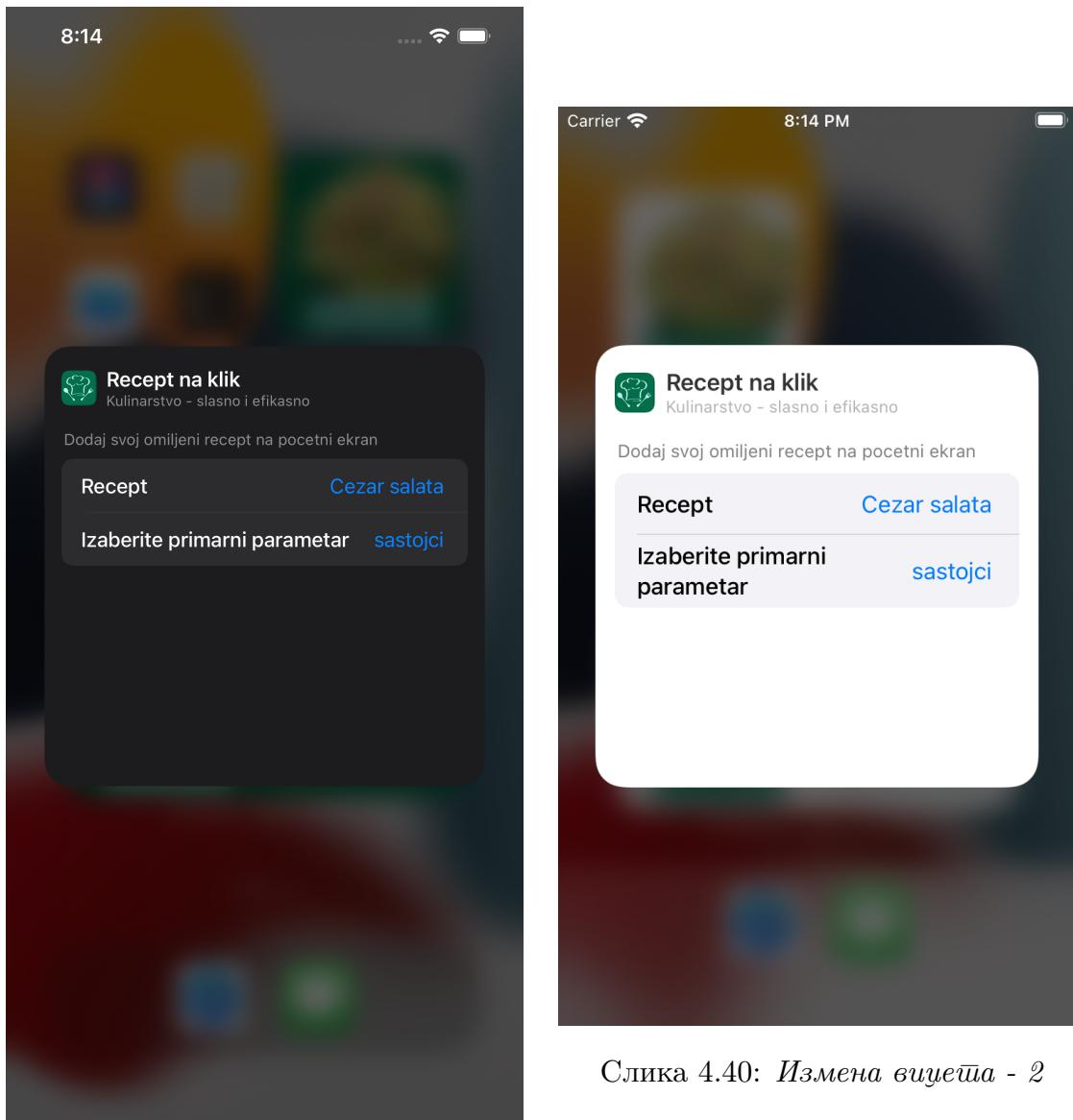


Слика 4.38: Приказ првој вишећа велике величине - 2

4.38.

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 параметра који одређује која ће листа бити примарна (листа састојака или листа корака припреме). Изглед екрана приликом конфигурације средњег виџета

ГЛАВА 4. ОПИС АПЛИКАЦИЈЕ

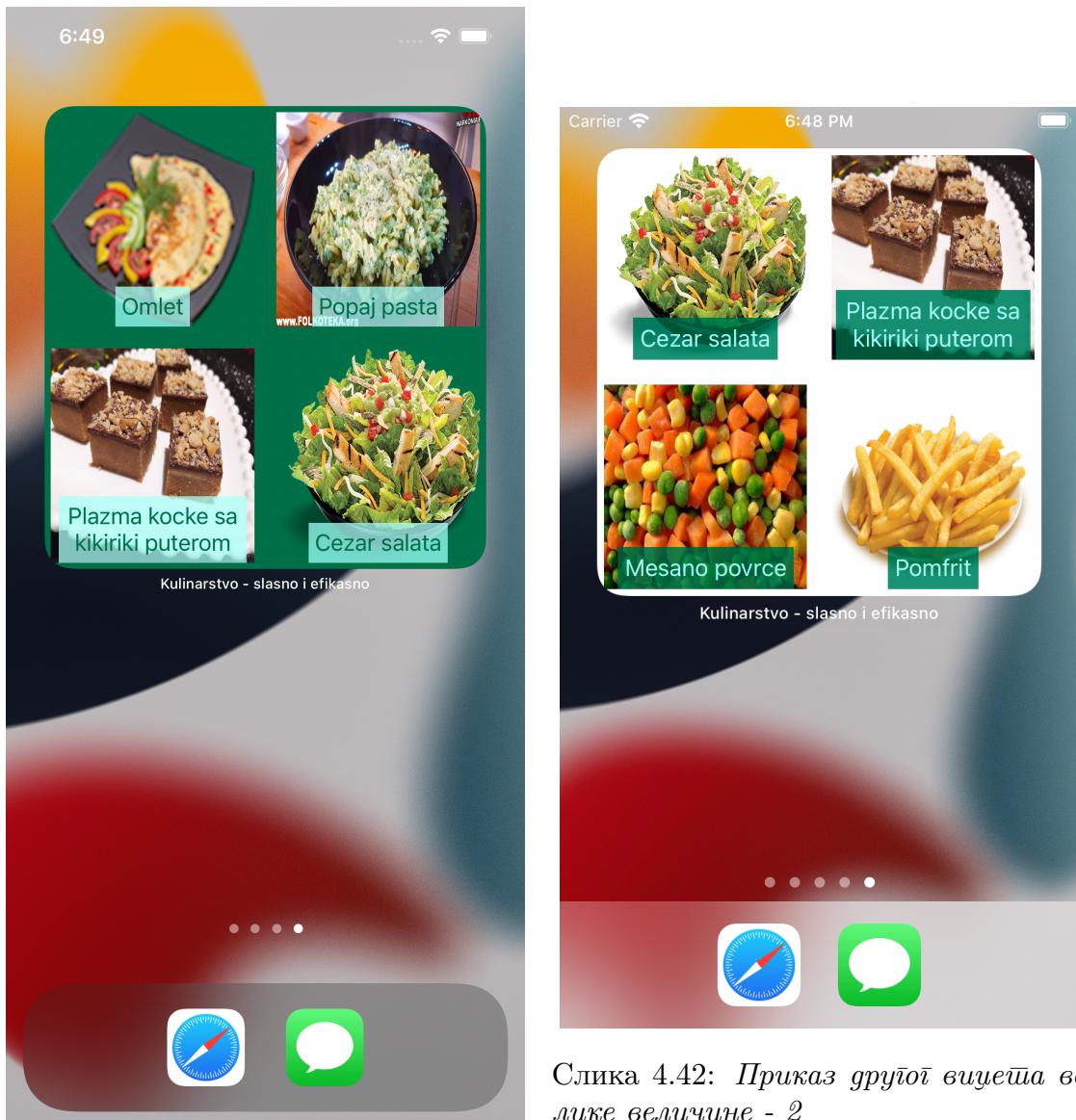


Слика 4.39: Измена вијећа - 1

приказан је на сликама 4.39 и 4.40.

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, који су статички конфигурисани и поновно се учитавају свака 24 сата и тако кориснику предлажу шта би могао да спрема тог дана. Приказани рецепти се налазе у листи омиљених рецепата корисника, тако да је задовољство корисника иза-

ГЛАВА 4. ОПИС АПЛИКАЦИЈЕ



Слика 4.41: Приказ другог вијешта велике величине - 1

Слика 4.42: Приказ другог вијешта велике величине - 2

браним рецептима загарантовано. Приказ другог типа виџета може се видети на сликама 4.41 и 4.42.

Глава 5

Закључак

Библиографија

- [1] Apple Inc. Apple Developer. on-line at: <https://developer.apple.com/swift/>.
- [2] Apple Inc. Swift Education. on-line at: <https://www.apple.com/education/k12/teaching-code/>.
- [3] Apple Inc. Swift on GitHub. on-line at: <https://github.com/apple/swift>.
- [4] Apple Inc. Swift Playground. on-line at: <https://www.apple.com/swift/playgrounds/>.
- [5] Apple Inc. SwiftUI. on-line at: <https://developer.apple.com/xcode/swiftui/>.
- [6] Apple Inc. The swift programming language (swift 5.5), 2014.
- [7] Apple Inc. Swift.org, 2021. on-line at: <https://www.swift.org/>.
- [8] StackOverflow. Stack overflow. on-line at: <https://stackoverflow.com/>.