Primerjava načrtovalskih vzorcev programske kode v mobilnih aplikacijah

Vid Ribič, Matevž Pogačnik

Univerza v Ljubljani, Fakulteta za elektrotehniko
E-pošta: ribvid@gmail.com

Comparison of design patterns used in the development of mobile applications

Abstract. *In this paper, we examine some of the more common design patterns used in the development of mobile applications for the iOS. With their help, the application is divided into multiple components, which has certain key advantages. It is easier to reuse user interface (UI) elements and we thus follow the don't repeat yourself (DRY) concept, making the code optimized, more understandable and easier to test. The use of a suitable design pattern quickens development and simplifies bug fixing.*

We also describe the model-view-controller (MVC) architecture, which seems an excellent starting point for most applications. In recent years, however, certain qualms have arisen regarding its suitability for more demanding programs as the use of MVC requires caution in order to avoid the problem of massive view controller.

With the MVP concept, the controller is moved to a part of the view, while man-in-the-middle becomes a new element – the presenter.

Also described is the MVVM pattern, where the view owns the view model, as well as VIPER, which divides individual tasks of the application into five minor components.

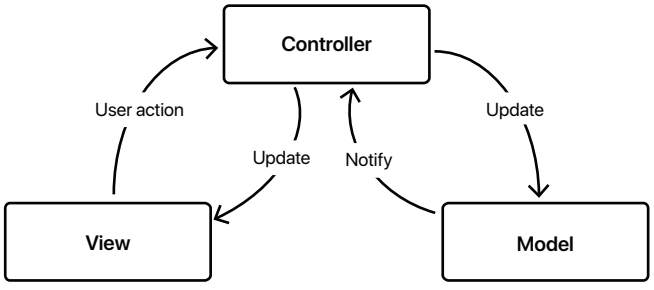
1 Uvod

Pri programiranju že od nekdaj pomembno vlogo igrajo načrtovalski vzorci (angl. design patterns). Z njihovo pomočjo razbijemo programsko kodo na več različnih komponent, vsako s svojo zadolžitvijo. To prinaša nekaj pomembnih prednosti. Modularno napisano kodo je lažje ponovno uporabiti in tako slediti konceptu DRY (angl. don't repeat yourself), s čimer zmanjšamo količino napisane kode in posledično verjetnost napak v njenem delovanju. Pomembna prednost je enostavnejše testiranje, ker ga lahko izvajamo nad vsako komponento posebej. Pri monolitno napisani kodi, ki opravlja več nalog hkrati, je testiranje veliko večji, če sploh rešljiv, izziv. Možnost enostavnega testiranja olajšuje spreminjanje obstoječih delov kode in dodajanje novih funkcionalnosti, prav tako kot boljša preglednost, organiziranost in jasnost, ki jo dosežemo z razbitjem kode na več smiselnih enot. Uporaba arhitekturnih vzorcev rezultira v hitrejšem razvoju, predvsem pa v manj hroščati in bolje optimizirani kodi.

Nič drugače ni pri razvoju programske opreme za mobilne naprave. Sprva so bile mobilne aplikacije enostavne in primitivne, zato so se večinoma uporabljali samo najosnovnejši arhitekturni koncepti, kot je model-pogled-kontroler oziroma MVC (angl. model-view-controller). S kompleksnejšimi in zahtevnejšimi programi so se pojavile težave tega pristopa in začelo se je razmišljati o vpeljavi novih arhitekturnih zasnov. V razvijalskih krogih se zadnja leta čedalje pogosteje zastavlja vprašanje, ali je MVC danes še vedno najprimernejši načrtovalski vzorec ali pa je za sodobno mobilno aplikacijo preveč enostaven, zato bomo v tem članku na kratko predstavili nekaj arhitekturnih idej, primernih za razvoj mobilnih aplikacij za platformo iOS.

2 MVC

Najpreprostejši, najpogostejši in tudi eden izmed najstarejših načrtovalskih vzorcev je model-pogled-kontroler ali MVC. Kratico je leta 1979 prvič uporabil Trygve Reenskaug za programe napisane v Smalltalku [1]. Applova različica, ki jo predstavljamo v nadaljevanju, je prilagojena posebnostim mobilnih aplikacij.



Slika 1: Pri idejni zasnovi MVC je most med pogledom in modelom kontroler.

Arhitektura je trodelna. Na eni strani je model, ki je abstrakten opis vsebine naše aplikacije. Model je popolnoma neodvisen od aplikacijskega ogrodja (angl. framework) [1]. Na drugi strani imamo pogled (angl. view), ki predstavlja vidni del aplikacije, uporabniški vmesnik. Pogled in model nista nikoli v neposredni povezavi, s čimer preprečujemo njuno soodvisnost. To je pomembno, saj lahko le tako poglede ponovno uporabimo tudi z drugimi modeli, kar je pri razvoju mobilnih aplikacij dokaj pogosto. Most

med modelom in pogledom je kontroler (angl. controller). Ta dobi informacijo o spremembi modela in ustrezno posodobi pogled. Ter obratno: ko uporabnik interaktira z aplikacijo in izvede določeno akcijo, kot je pritisk na gumb, lahko kontroler posodobi model, če je to seveda potrebno [2].

Po drugi strani pa to pomeni, da ima kontroler veliko zadolžitev in prav to je ključna težava ter kritika te arhitekturne zasnove. Pri kompleksnih aplikacijah se namreč hitro zgodi, da postane kontroler zelo obsežen, čemur se, kot zapisano v uvodu, skušamo izogniti. Pogled v dokumentacijo razkrije, kakšne zadolžitve lahko dodelimo kontrolerju (izpeljemo ga iz razreda UIViewController) [3]. Med njimi so skrb za pravilen prikaz pogleda, nadzor nad njegovim življenskim ciklom, spremljanje morebitnih rotacij zaslona, zaznava in ustrezna obdelava uporabnikovih akcij, lahko se povezuje s strežnikom in obdeluje odgovore, ki mu jih ta vrne, posluša obvestila (angl. notifications) in tako dalje. Seveda vseh funkcij ni potrebno implementirati, če niso potrebne, kljub temu pa se moramo zavedati, da se lahko hitro ujamemo v past preobsežne kode. Tisti, ki trdijo, da gre za preživet koncept, prav zaradi preobsežnosti kontrolerja MVC ironično pojasnujejo kot "massive view controller". Delno rešitev te težave predstavlja koncept kontrolerjev otrokov (angl. child controllers), na katere lahko prenesemo kakšno izmed nalog in jih nato po potrebi vključujemo v druge kontrolerje. Primer take uporabe bi bila ločena kontrolerja za obravnavo nalaganja vsebine in prikaz napake, ki ju nato lahko vključimo v vse tiste kontrolerje, ki obdelujejo strežniške odgovore [4]. Z uporabo kontrolerjev otrokov se lahko izognemo nevarnosti masivnega kontrolerja, obenem pa jih lahko uporabimo na več različnih mestih in s tem sledimo ideji DRY.

Druga težava kontrolerja, ki je povezan s pogledom in modelom, je oteženo testiranje poslovne logike (angl. business logic). Testiranje sicer je mogoče, vendar zahteva nekaj več dela in truda.

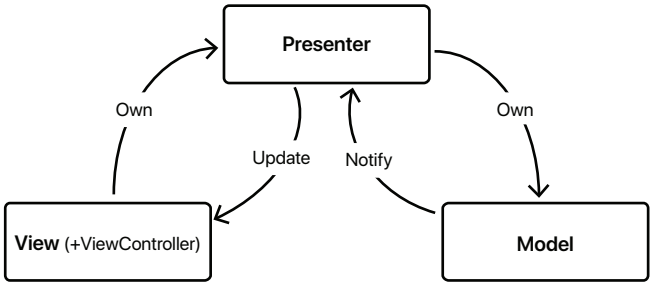
Kljub temu MVC še vedno ostaja pogosta arhitekturna izbira. Prvi razlog je njegova enostavnost – MVC-ja se je najlažje in najhitreje naučiti, hkrati pa v aplikacijo ne vnaša preveliko nepotrebne kode (angl. overheada). MVC je arhitektura, ki jo priporoča in v svojih aplikacijah uporablja Apple. Marsikdo, ki zagovarja uporabo MVC-ja, je prav zaradi tega prepričan, da je aplikacijsko ogrodje napisano z mislijo na to zasnovo (ali vsaj njene neposredne izpeljanke) in so vsi elementi, potrebni za razvoj, že del obstoječih knjižnic.

Četudi se nam zdijo drugi načrtovalski vzorci primernejši od MVC-ja, pa ta velja za osnovo večini ostalih idej. Marsikakšna izmed njih se pravzaprav zdi zelo podobna MVC-ju, zaradi česar se nemalokdo sprašuje, ali ne gre le za drugačen način implementacije istega vzorca in drugačno ime (MV-karkoli, angl. MV-whatever). Masiven kontroler za zagovornike MVC-ja ni odraz slabosti tega načrtovalskega vzorca, pač pa programerja, ki ga ne zna pravilno implementirati [5].

3 MVP

Eden izmed vzorcev, izpeljan iz MVC-ja, je model-pogled-prikazovalnik ali MVP (angl. model-view-presenter). Leta 1996 ga je zasnoval Mike Potel [6]. Ker vsevedni kontroler vodi v obsežno kodo in težavno testiranje, se pri tej zasnovi pomakne v del pogleda; gledano z vidika razredov lahko rečemo, da je poleg razreda UIView sedaj del pogleda tudi UIViewController. Ta sicer še vedno skrbi za UIView, vendar nima več neposredne povezave z modelom, s čimer se izognemo številnim težavam. Prikazovalnik zasede vlogo moža v sredini med modelom in pogledom. Pogled delegira uporabnikove interakcije prikazovalniku, ki vsebuje logiko za njihovo obravnavo. Hkrati je tudi tisti, ki prevzame komunikacijo z modelom, podatke, ki jih od njega prejema, ustrezno pripravi in nato pošlje naprej UIViewControllerju. Prikazovalnik je prav tako kot model neodvisen od aplikacijskega ogrodja UIKit [7].

UIViewController na ta način razbremenimo številnih nalog, hkrati pa tako kot UIView postane nevedni člen arhitekture. S tem ne dosežemo le manj obsežne kode, temveč tudi olajšujemo testiranje, saj lahko sedaj poslovno logiko aplikacije stestiramo neodvisno od UIViewControllerja. Po drugi strani pa MVP prinese nekaj dodatnega dela, saj je potrebno vpeljati nov element, torej prikazovalnik, in ga povezati z ostalimi členi arhitekture.

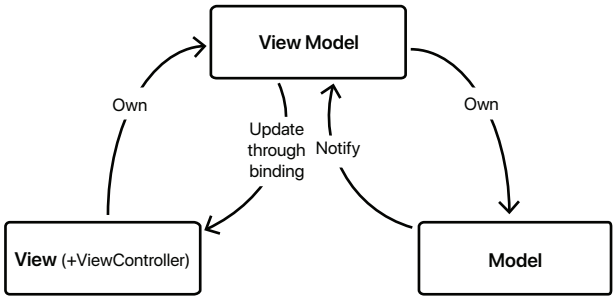


Slika 2: Pri MVP se kontroler pomakne v del pogleda, mož v sredini pa postane prikazovalnik.

4 MVVM

Pri načrtovalski zasnovi MVP ima pogled referenco na prikazovalnik in obratno. Če želimo prikazovalnik razbremeniti povezave s pogledom, lahko uporabimo vzorec model-pogled-pogled model ali MVVM (angl. model-view-view model). Izvorno ga je predstavil Microsoft leta 2005, da bi poenostavil dogodkovno usmerjeno programiranje (angl. event-driven programming) [2].

Še vedno ostaja ideja, da pogled predstavlja vidni del aplikacije in nima možnosti neposredne komunikacije z modelom. Ključna razlika pa je sledeča: pogled si lasti nov nivo, poimenovan pogled model, ta pa si lasti model. Pogled model je zadolžen za hrambo podatkov za pogled, ki si ga lasti. Za primer vzemimo aplikacijo za predvajanje glasbe. Pogled bo narisal seznam skladb, pogled model pa bo hranil informacije o njih (naslov, izvajalca, žanr, dolžino, ...).



Slika 3: Med pogledom in pogledom model vzpostavimo povezavo (angl. binding). Ker iOS nima nativnega pristopa k temu, se veliko razvijalcev odloči za vpeljavo reaktivnosti.

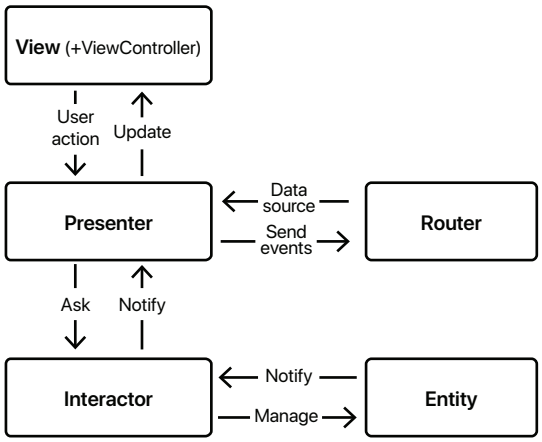
Tok aplikacije je pri MVVM nekoliko drugačen. Postavi se vprašanje, kateri nivo je zadolžen za pridobitev podatkov s strežnika ali baze? Povedano skozi gledišče zgornjega primera: kateri člen arhitekture bo s strežnika pridobil seznam skladb? Pri MVC in MVP sta za to zadolžena kontroler oziroma prikazovalnik, ki nato uporabita model in posodobita pogled. Pri MVVM pa za povezavo s strežnikom oziroma bazo skrbi pogled. Ta pridobi podatke in jih posreduje nivoju pogled model, ki jih sprocesira in pripravi v ustrezni obliki za uporabniški vmesnik ter vrne nazaj pogledu, s katerim je povezan (angl. binding). Težava je, da pri iOS ni pravega in nativnega pristopa za tovrstno povezovanje. Obstajajo sicer mehanizmi, kot so KVO (key-value observing), delegati ali notifikacije, vendar niso tako učinkoviti kot povezovalni mehanizmi pri drugih programskih jezikih. Osnovno aplikacijsko ogrodje ni bilo napisano z mislijo na to načrtovalsko zasnovo. Razvijalci se zato koncepta MVVM najpogosteje lotevajo z reaktivnim pristopom, čeprav ta seveda ni nujen. MVVM še vedno skrbi za razdelitev nalog med različne nivoje, reaktivnost pa za povezovanje (binding).

Čeprav sta pogled in pogled model med seboj tesno povezana, pa je pogled še vedno ločen od modela, s čimer se ohranja osnovna ideja MVC-ja. Hkrati je pri MVVM poslovna logika ločena od kontrolerja in s tem ostaja enostavnost testiranja, ki smo jo vpeljali z MVP-jem. Po drugi strani pa z uvajanjem MVVM-ja prihaja do novih izzivov. Prvo vprašanje, ki bega razvijalce, je razdelitev nalog. Kar nekoliko nenavadno se zdi, da je pogled tisti, ki je zadolžen za povezavo s strežnikom ali bazo, vendar tako narekuje izvorna ideja MVVM-ja. Morda bi bila ta naloga primernejša za pogled model? Primer uporabne vrednosti tega, da je pogled zadolžen za pridobivanje podatkov, je to, da sam ve, kdaj so podatki uspešno preneseni in kdaj je prišlo do napake. Če bi to nalogo preselili na nivo pogled model, bi to pomenilo, da moramo dodati še en nivo abstrakcije, za primer, ko podatkov ni in pogled model vrača pogledu napako.

Druga težava MVVM-ja pri iOS-u je manjko nativnega pristopa k povezovanju. Reaktivnost se sliši kot dobra rešitev, vendar je kompleksna, zahteva več učenja in posledično bolj nagiba našo kodo k možnosti napak. Zavedajoč se prednosti in slabosti tega koncepta, lahko sklenemo, da je MVVM zanimiva in premisleka vredna rešitev, ki pa zaradi svoje zahtevnosti zagotovo ni primerna za vse aplikacije.

5 VIPER

Še korak naprej od MVVM k večji modularnosti je VIPER (view-interactor-presenter-entity-router), ki aplikacijo razdeli na pet delov.



Slika 4: Viper razdeli aplikacijo na pet delov. Čeprav v teoriji dobro razdeli zadolžitve in olajša testiranje, se v praksi izkaže, da je tovrstna abstrakcija nemalokrat prevelika.

Pogled ostane zadolžen za prikaz uporabniškega vmesnika, uporabnikove akcije pa pošlje prikazovalniku. Ta je odgovoren za njihovo obdelavo in je popolnoma neodvisen od uporabniškega vmesnika. Tretji člen je interaktor. Vsebuje vso poslovno logiko, hkrati ima v lasti in upravljanju entiteto (angl. entity), ki je četrti nivo te zasnove, namenjen definiranju strukture za shranjevanje podatkov. Zadnja komponenta je usmerjevalnik (angl. router), ki skrbi za navigacijo na podlagi akcij, prejetih iz prikazovalnika [8].

VIPER razdeli zadolžitve aplikacije na več členov, s čimer še dodatno poenostavi testiranje. Po drugi strani pa uvedba toliko novih razredov aplikacijo zelo razdrobi in podaljšuje učenje ter razumevanje zgradbe in arhitekture programa. Zavedati se moramo tudi velike razlike med osnovno zasnovo MVC in VIPER. Se je res smiselno toliko oddaljiti od koncepta, ki ga zagovarja tisti, ki skrbi za aplikacijsko ogrodje?

6 Sklep

Poleg predstavljenih obstaja še mnogo drugih načrtovalskih zasnov (Redux, Cake, MVC+VS, MAVB, TEA, ...). Postavlja se vprašanje, katera je najprimernejša? Bistveno je zavedanje, da je namen vsake arhitekture programerju olajšati delo, pohitriti programiranje, odpravljanje napak in dodajanje novih funkcionalnosti, poenostaviti testiranje ter zmanjšati obseg kode. Če VIPER ponuja nadrobno razdelitev nalog med posameznimi deli, kar ima nedvomno svoje prednosti, pa je bistveno bolj kompliciran od MVC-ja. Obstaja pomislek, da se arhitekture, kot je VIPER, na papirju zdijo dobra rešitev, v praksi pa prinašajo preveč abstrakcije in posledično zmede. Ključna težava

MVC-ja je nevarnost preobsežnega kontrolerja, vendar to ne sme biti razlog, da se tej idejni zasnovi prehitro odpovemo. Obseg kontrolerja lahko zmanjšamo. Data source in delegate za collection view ali table view, ki sta pogosta elementa uporabniškega vmesnika, lahko ločimo od UIViewControllerja. Iz kontrolerja lahko vselej prestavimo mrežne klice in obdelavo datotek JSON, s čimer med drugim poenostavimo testiranje. Omenili smo tudi kontrolerje otroke, ki jih vključujemo, kjer jih potrebujemo. Težavo pretoka podatkov med posameznimi členi aplikacije lahko učinkovito rešimo z uporabo koordinatorjev. Upoštevajoč možnosti za razbremenitev kontrolerja in izogib njegove preobsežnosti, ugotovimo, da ima vendarle MVC še vedno široko uporabno vrednost. Ker je MVC najenostavnejši in Applov izbrani načrtovalski vzorec za platformo iOS, je najbolje, da se mu odpovemo šele takrat, ko predstavljajo njegove omejitve nepremostljivo oviro.

Literatura

[1] C. Eidhof, M. Gallagher, in F. Kugler, App Architecture, iOS Application Design Patterns in Swift. CreateSpace Independent Publishing Platform, 2018.

[2] A. Harbade, iOS design patterns, iOS design patterns — Part 1 (MVC, MVP, MVVM). Dostopno na: https://medium.com/swlh/ios-design-patterns-a9bd07818129.

[3] Apple, Documentation, UIKit, View Controllers, UIViewController. Dostopno na: https://developer.apple.com/documentation/uikit/uiviewcontroller.

[4] J. Sundell, Using child view controllers as plugins in Swift. Dostopno na: https://www.swiftbysundell.com/posts/using-child-view-controllers-as-plugins-in-swift.

[5] A. Vacić, Much ado about iOS app architecture. Dostopno na: http://aplus.rs/2017/much-ado-about-ios-app-architecture/.

[6] M. Potel, MVP: Model-View-Presenter The Taligent Programming Model for C++ and Java. Taligent, 1996.

[7] I. Agha, A dumb UI is a good UI: Using MVP in iOS with swift. Dostopno na: http://iyadagha.com/using-mvp-ios-swift/.

[8] A. Harbade, iOS design patterns — Part 2 (VIPER). Dostopno na: https://medium.com/@anup.harbade.iosdev/ios-design-patterns-part-2-viper-fa3522b22b6b.