Zbornik radova Fakulteta tehničkih nauka, Novi Sad

UDK: 004.9
DOI: https://doi.org/10.24867/24BE22Djordjevic

RAZVOJ IOS APLIKACIJE ZA PRAĆENJE ZDRAVSTVENOG STANJA KORISNIKA UZ UPOTREBU ČISTE ARHITEKTURE I MVVM DIZAJN PATERNA

HEALTH MONITORING IOS APPLICATION DEVELOPMENT USING CLEAN ARCHITECTURE AND MVVM DESIGN PATTERN

Maja Đorđević, *Fakultet tehničkih nauka, Novi Sad*

Oblast – ELEKTROTEHNIKA I RAČUNARSTVO

Kratak sadržaj – *U ovom radu opisano je rešenje implementacije iOS mobilne aplikacije korišćenjem čiste arhitekture i MVVM dizajn paterna. Ovakav pristup pruža skalabilno, pouzdano, testabilno i modularno softversko rešenje. Dato implementirano projektno rešenje je aplikacija za praćenje zdravstvenog stanja korisnika.*

Ključne reči: *mobilne aplikacije, iOS, MVVM dizajn patern, čista arhitektura*

Abstract – *The thesis describes implementation of an iOS mobile application using clean architecture and MVVM design pattern. This approach provides a scalable, reliable, testable and modular software solution. The given implemented project solution is an application for monitoring the user's health condition.*

Keywords: *mobile applications, iOS, MVVM pattern, clean architecture*

1. UVOD

Mobilne aplikacije postale su sastavni deo svakodnevnih aktivnosti i pružaju brz i jednostavan način za pristup informacijama i obavljanje različitih zadataka. Jedna od oblasti u kojoj se mobilne aplikacije sve više razvijaju je oblast zdravstva. Kako bi se osigurao visoki kvalitet, efikasnost i održivost softvera, neophodno je koristiti dobro promišljene arhitekture prilikom razvoja aplikacije. U ovom radu biće predstavljen razvoj mobilne aplikacije *HealthApp* za praćenje zdravstvenog stanja korisnika uz upotrebu modernih pristupa i tehnologija.

Ideja rada jeste realizacija mobilne aplikacije za potrebe praćenja zdravstvenog stanja korisnika, podrška za upozorenje korisnika ukoliko je došlo do kritičnih vrednosti praćenih zdravstvenih parametara, kao i dodatne preporuke za održavanje zdravstvenog stanja na normalnom nivou. Zdravstveni podaci koji se prate unutar aplikacije su: nivo krvnog pritiska, broj otkucaja srca, saturacija kiseonika u krvi, nivo šećera u krvi, brzina disanja, kao i broj pređenih koraka.

Međutim, razvoj takvih aplikacija može biti izazovan jer se moraju uzeti u obzir mnogi faktori kao što su sigurnost, pouzdanost, skalabilnost i performanse.

NAPOMENA:
Ovaj rad proistekao je iz master rada čiji mentor je bila dr Dunja Vrbaški, docent.

Takođe, aplikacija treba da ima mogućnost lakog održavanja i modifikovanja kako bi se prilagodila promenama korisničkih zahteva i potrebama tržišta. Problem koji se rešava u radu jeste razvoj iOS aplikacije za potrebe praćenja zdravstvenog stanja korisnika na jednom mestu, uz korišćenje koncepata čiste arhitekture i MVVM obrasca. Ovakva arhitektura omogućava lakše održavanje, izmenu i testiranje aplikacije. Skaliranje i dodavanje novih funkcionalnosti je takođe olakšano, kao i razdvajanje prikaza korisničkog interfejsa od poslovne logike na jednostavan način. Ove arhitekture pružaju jasnu strukturu i organizaciju koda, što olakšava sam razvoj i održavanje softvera.

2. PROGRAMSKE PARADIGME

Paradigma je jezički nezavisan način programiranja [1]. Ona diktira koje su to programske strukture koje je potrebno koristiti i kada ih je potrebno koristiti. Sve do danas, postojale su tri takve paradigme.

2.1. Struktuirano programiranje

Strukturirano programiranje je programska paradigma koja je usmerena ka poboljšavanju računarskog programa u smislu poboljšavanja njenog kvaliteta, vremena razvoja i povećavanja jasnoće programa. Alat za poboljšavanje računarskog programa se krije u upotrebi konstrukcija kontrole toka, ponavljanja, blokovskih struktura i podrutina. Kod strukturiranog programiranja program se deli na manje module, tako da program postane jednostavan za razumeti.

2.2. Funkcionalno programiranje

Funkcionalno programiranje je stil koji se zasniva na izračunavanju izraza kombinovanjem funkcija. Osnovne aktivnosti su definicaija funkcije, poziv funkcije i kreiranje programa. Program u funkcionalnom programiranju je niz definicija i poziva funkcija. Izvršavanje programa je evaluacija funkcija.

2.3. Objektno orijentisano programiranje

U objektno orijentisanom programiranju polazi se od toga da i podatak i funkcije koje obrađuju taj podatak čine deo neke celine. U ovakvom konceptu takvu celinu posmatramo kao objekat koji ima neka moguća stanja i ponašanja. Stanja predstavljaju vrednosti podataka koji se nalaze u okviru objekata, koji vremenom mogu da se menjaju i nazivaju se polja. Ponašanja predstavljaju pravila menjanja stanja, reakciju na uticaje okoline i

načina uticanja na okolinu. Oni se opisuju funkcijama i nazivaju se metode.

Osnovni principi objektno orijentisanog programiranja su:

1. Apstrakcija
2. Enkapsulacija
3. Polimorfizam
4. Nasleđivanje

3. PRINCIPI DIZAJNA SOFTVERA - SOLID PRINCIPI

Cilj SOLID principa je kreiranje softverskih struktura srednjeg nivoa koje tolerišu promene, jednostavne su za razumevanje i osnova su za komponente koje se mogu koristiti u mnogim softverskim sistemima. Srednji nivoi se odnose na nivoe koji su neposredno iznad nivoa programskog koda i pomažu pri definisanju programskih struktura koje se koriste u modulima i komponentama.

SOLID predstavlja pet osnovnih principa koji pomažu u kreiranju dobre arhitekture softvera [2]. Oni definišu kako se kombinuju funkcije i strukture popdataka u klase i kako bi te klase trebalo kombinovati jednu sa drugom.

3.1. Princip jedinstvene odgovornosti

Princip jedinstvene odgovornosti definiše pravilo da svaka klasa ili modul treba imati jasan i jedinstven razlog za promenu. To znači da svaka komponenta softvera treba biti odgovorna samo za obavljanje jedne specifične funkcionalnosti ili zadatka. Kada se javi potreba za promenom te funkcionalnosti, treba da postoji mogućnost izvršiti tu promenu bez uticaja na ostatak sistema.

3.2. Princip otvorenosti-zatvorenosti

Princip otvorenosti-zatvorenosti podrazumeva da klase trebaju biti otvorene za proširivanje, ali zatvorene za promene. To znači da klase trebaju biti napisane tako da se mogu proširiti, ali da se ne moraju menajti da bi se to postiglo.

3.3. Liskov princip zamene

Princip podrazumeva da klase trebaju biti zamenjive sa izvedenim klasama. Ukoliko se koristi objekt neke klase, on se može zameniti objektom njene podklase bez da dođe do promene u radu programa. To takođe znači da ako se koristi metoda neke klase, metoda podklase se može koristiti na isti način bez promene u radu programa.

3.4. Princip razdvajanja interfejsa

Princip razdvajanja interfejsa se odnosi na to da se interfejsi trebaju podeliti na manje, specijalizovane interfejse koji su konzistentni sa konkretnim potrebama klijenata. Odnosno, trebaju se izbegavati sveobuhvatni interfejsi koji sadrže metode koje nisu potrebne svima koji ih implementiraju.

3.5. Princip inverzije zavisnosti

Princip inverzije zavisnosti se fokusira na smanjenje zavisnosti između komponenti u sistemu, što omogućava lakše održavanje, razvoj i testiranje koda. Prema DIP-u, visokorazvojni moduli, na primer klase, ne bi trebale zavisiti o nižerazvojnim modulima, na primer implementacijama, već bi se trebale oslanjati na apstraktne interfejse ili apstrakcije. Ovaj princip omogućuje da se nižerazvojni moduli menjaju ili zamjenjuju bez da se menjaju visokorazvojni moduli.

4. DRY PRINCIP

DRY (engl. Don't Repeat Yourself) je princip u softverskom inženjerstvu koji se odnosi na smanjenje duplikata koda [3]. Cilj principa DRY je da se smanji količina koda koja se mora menjati kako bi se promenile neke od funkcionalnosti softvera. To se postiže tako što se kod koji se ponavlja preusmerava u jedinstvene metode, klase ili module. Primer kršenja ovog principa je kada se isti kod koristi više puta u različitim delovima programa. To može dovesti do problema održivosti, jer se svaki put kada se kod treba promeniti, mora menjati na svakom mestu gde se koristi. Ako se kod preusmeri u jedinstvenu metodu ili klasu, promene se mogu izvesti na jednom mestu, što čini kod jednostavnijim za održavanje i razumevanje.

5. SOFTVERSKA ARHITEKTURA

Arhitektura softvera je apstraktna struktura softverskog sistema koja predstavlja disciplinu kreiranja takvih struktura i sistema [4]. Svaka struktura sadrži softverske elemente, odnose među njima i svojstva elemenata i relacija. Ona se odnosi na donošenje fundamentalnih strukturalnih izbora. Jednom kada se implementiraju, svaka dodatna promena je skupa.

Softverska arhitektura je važna jer utiče na performanse, sigurnost, skalabilnost, održivost i lakoću razvoja softvera. Ako se softverska arhitektura ne planira i ne implementira ispravno, to može dovesti do problema sa performansama, skaliranjem i održavanjem softvera.

5.1. Karakteristike softverske arhitekture

Softverske arhitekte razdvajaju karakteristike arhitekture u široke kategorije u zavisnosti od funkcionalnosti, učestalosti zahteva, strukture i slično [5]. U tabli su predstavljenje neke od najvažnijih karakteristika koje se obično razmatraju.

Tabela 1. *Osnovne karakteristike softverske arhitekture*

Operativne karakteristike	Strukturalne karakteristike	Međusektorske karakteristike
Dostupnost	Mogućnost konfigurisanja	Pristupačnost
Performanse	Proširivost	Bezbednost
Pouzdanost	Mogućnost podrške	Upotrebljivost
Tolerancija grešaka	Prenosivost	Privatnost
Skalabilnost	Održavanje	Izvodljivost

6. ČISTA ARHITEKTURA

Čista arhitektura je pristup arhitekturi softvera koji se smatra funkcionalnom arhitekturom [1] [6]. Čista arhitektura se fokusira na organizaciju koda tako da se

razdvaja funkcionalnost od implementacije. Ova arhitektura podrazumeva korišćenje SOLID i DRY principa. Čista arhitektura koristi tehnike kao što su inverzija kontorle i ubrizgavanje zavisnosti kako bi se osiguralo da komponente ne budu direktno povezane, već da se komunicira preko jasno definiranih interfejsa. Navedene tehnike omogućavaju da se komponente lako menjaju, testiraju i razvijaju nezavisno jedna od druge.

6.1. Inverzija kontrole

Inverzija kontrole je softverski princip koji se koristi u objektno orijentiranim sistemima. Osnovna ideja principa je da se kontrola nad procesom izvršavanja prebacuje sa klase koja zahteva određene servise na klasu koja te servise pruža.

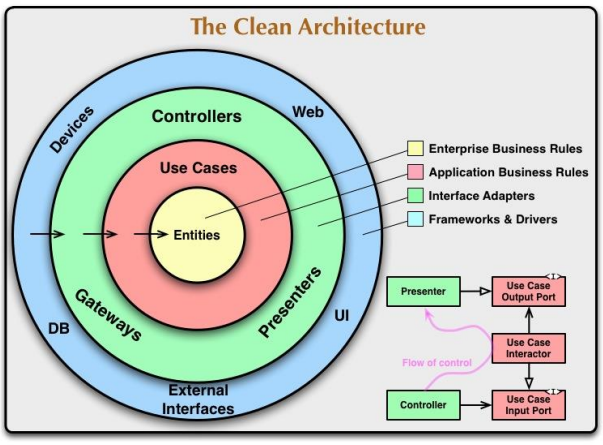
6.2. Ubrizgavanje zavisnosti

Ubrizgavanje zavisnosti je tehnika koja se koristi u objektno orijentiranim sistemima za rešavanje zavisnosti između klasa. Osnovna ideja je da se objekti koji su potrebni nekoj klasi ubace u klasu, umesto da se oni instanciraju unutar klase. To omogućava da se klasa koja koristi druge objekte razdvoji od klase koja ih instancira, što pomaže u održavanju i razvoju softvera.

6.3. Slojevi čiste arhitekture

Čista arhitektura se sastoji iz sledećih slojeva (Slika 1):

1. Prezentacioni sloj: odgovoran za rukovanje korisničkim unosom i prikazivanje informacija korisniku. Obično uključuje korisnički interfejs, kao što je grafički korisnički interfejs ili interfejs komandne linije.
2. Sloj slučaja upotrebe ili aplikativni sloj: sadrži poslovnu logiku i pravila aplikacije. On definiše slučajeve korišćenja sistema i implementira osnovnu funkcionalnost aplikacije.
3. Sloj domena: sadrži modele domena i entitete koji predstavljaju srž problematičnog domena aplikacije.
4. Infrastrukturni sloj: obrađuje spoljne zavisnosti kao što su baze podataka, veb servisi i drugi spoljni sistemi.



Slika 1. *Slojevi čiste arhitekture [6]*

Razdvajanjem koda na ovakve nezavisne sloje, čini ga modularnijim, lakšim za testiranje i održavanje. Takođe, ovaka struktura olakšava dodavanje novih funkcija, menjanje postojećih i zamenu osnovnih tehnologija bez uticaja na ostatak koda.

7. SOFTVERSKI ARHITEKTURALNI OBRASCI

Arhitekturalni obrazac je koncept koji rešava i jasno definiše suštinske i kohezivne elemente arhitekture softvera [7]. Mnoštvo različitih arhitektura može implementirati isti obrazac i takođe deliti iste karakteristike. Često se obrasci definišu kao strogo opisani i opšte dostupni.

Oni predstavljaju nezaobilaznu komponentu razvoja softvera, međutim ne garantuju uvek uspeh projekta sami po sebi. Potrebno ih je integrisati sa dobrim dizajnom i pravilima razvoja, kao i sa primenom drugih praksi za kvalitet softvera, kao što su kontinuirana integracija i testiranje.

Neki od najpoznatijih arhitekturalnih obrazaca su:

- MVC (Model View Controller) obrazac: razdvaja aplikaciju na modele, prikaz i kontrolera tako da su logika aplikacije i interakcija sa korisnikom razdvojene.
- MVVM (Model View View-Model) obrazac: razdvaja aplikaciju na model, prikaz i model prikaza tako da su logika aplikacije i logika prikazivanja podataka nezavisna od interakcije korisnika.
- Mikroservisni obrazac: deli aplikaciju na mnoštvo malih nezavisnih servisa koji rade zajedno kako bi obavili kompleksne zadatke. Obrazac slojevite arhitkture - razdvaja aplikaciju u nekoliko slojeva gde svaki sloj ima specifičnu funkciju.
- Arhitekturalni obrazac vođen događajima: celokupna komunikacija unutar aplikacije se dešava kroz događaje. Događaj se šalje u sistem bus-a jednom kada se izgeneriše.
- Klijent-server obrazac: apikacija se deli na klijent i server deo, u kom klijent šalje zahteve, a server odgovara na te zahteve.

7.1. MVVM arhitekturalni obrazac

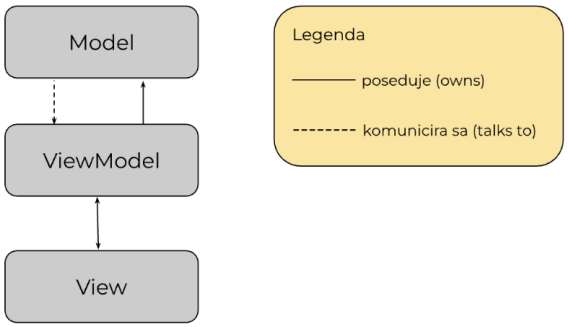
MVVM obrazac je nastao kao evolucija MVC obrasca [8]. Cilj je bio poboljšati MVC obrazac kako bi se omogućilo lakše razvoj aplikacija sa interaktivnim korisničkim interfejsom. MVVM je razvijen od strane Microsoft kompanije i prvi put objavljen 2007. godine. MVVM omogućava razdvojenje logike aplikacije od korisničkog interfejsa što omogućava fleksibilnost, modularnost i testabilnost koda. MVVM obrazac upravo rešava nedostatke MVC obrasca zahvaljujući jačoj modularnosti, boljoj razdvojenosti funkcija i testabilnosti.

7.1.1. Komponente MVVM obrasca

MVVM obrazac se sastoji iz tri jasno definisanih i međusobno povezanih komponenti (Slika 2) u kojima svaka ima zaduženje i obavlja specifične funkcije.

1. Model: predstavlja podatke i logiku aplikacije.
2. Prikaz: koristi se za prikaz podataka i primanje korisničkih akcija. To može uključivati ekrane, forme ili dijaloge.

3. Prikaz modela: komponenta koja povezuje model i prikaz. On se koristi za obradu podataka, izvršavanje logike aplikacije i ažuriranje prikaza.



Slika 2. Komponente *MVVM obrasca*

MVVM obrazac se često koristi u razvoju softvera zbog svojih brojnih prednosti, koje uključuju jasnu raspodelu odgovornosti između komponenata, povećanje produktivnosti, fleksibilnost i podršku testiranju i automatizaciji.

8. OPERATIVNI SISTEM IOS

iOS (iPhone Operating System) je mobilni operativni sistem koji je razvijen od strane Apple kompanije 2007 [9]. godine koji je moguće pokrenuti isključivo na hardverima koji su razvijeni od strane kompanije. Prvobitno je razvijen za operativni sistem iPhone uređaja, a kasnije je prošiten i na ostale uređaje kao što su iPad i iPod Touch.

Arhitektura iOS operativnog sistema sadrži međusloj između aplikacija i hardvera tako da oni ne komuniciraju direktno. Niži slojevi pružaju osnovne usluge, a viši slojevi pružaju korisnički interfejs i sofisticiranu grafiku.

9. RAZVOJNO OKRUŽENJE XCODE

Razvojno okruženje Xcode sadrži pakete alata koji se koriste za razvijanje softvera za iOS, macOS, watchOS i tvOS operativne sisteme. Krajem 2003. godine je izasla prva verzija softvera. Verzija 14.2 je poslednja stabilna verzija koja je objavljena krajem 2022. godine i dostupna je na Mac App Store prodavnici besplatno.

Xcode ima sposobnost da generiše univerzalne binarne datoteke koje omogućavaju softveru da se izvršava na PowerPC, kao i na platformama koje su bazirane na Intel x86 procesorima i koje sadrže i 32-bitni i 64-bitni kod za obe arhitekture. Ovu sposobnost ima zahvaljujući Mach-O izvršnom formatu koji dozvoljava fat binary datoteke koje imaju kod za različite arhitekture.

10. ZAKLJUČAK

U radu je predstavljeno implementaciono rešenje iOS mobilne aplikacije korišćenjem čiste arhitekture i MVVM dizajn obrasca. Čista arhitektura i MVVM dizajn patern pružaju stabilan temelj za razvoj iOS mobilne aplikacije za praćenje zdravlja korisnika. Ovi pristupi omogućavaju organizaciju koda, poboljšavaju održivost i testiranje, olakšavaju integrisanje sa eksternim servisima i pružaju fleksibilnost za buduće promene.

Primena čiste arhitekture omogućuje podelu aplikacije na slojeve odgovornosti. Ovo omogućuje bolje organizovanje koda i jasno definisane granice između komponenti, što olakšava održavanje i proširivanje aplikacije. Takođe, čista arhitektura promoviše nezavisnost od okvira, što olakšava zamenu ili nadogradnju određenih komponenti.

Kombinacija korišćene arhitekture i MVVM dizajn obrasca omogućava laku integrisanost sa različitim servisima i eksternim izvorima podataka, kao što su servisi za praćenje zdravlja korisnika. Jasno odvojeni slojevi i odgovornosti omogućavaju fleksibilnost u promeni ili dodavanju novih servisa bez uticaja na ostatak aplikacije.

11. LITERATURA

[1] Robert C.Martin, “Clean Architecture”, 2020.
[2] Gary McLean Hall, “Adaptive Code : Agile Coding with Design Patterns and SOLID Principles”, 2017.
[3]https://en.wikipedia.org/wiki/Don%27t_repeat_yourself (pristupljeno u martu 2023.)
[4]https://en.wikipedia.org/wiki/Software_architecture (pristupljeno u martu 2023.)
[5]https://www.geeksforgeeks.org/fundamentals-of-software-architecture (pristupljeno u martu 2023.)
[6]https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html (pristupljeno u martu 2023.)
[7 https://en.wikipedia.org/wiki/Architectural_pattern (pristupljeno u martu 2023.)
[8]https://learn.microsoft.com/en-us/dotnet/architecture/maui/mvvm (pristupljeno u martu 2023.)
[9]https://en.wikipedia.org/wiki/IOS (pristupljeno u martu 2023.)

Kratka biografija:

Maja Đorđević rođena je u Ćupriji 1996. godine. Fakultet tehničkih nauka u Novom Sadu upisala je 2015. godine. Završila je osnovne akademske studije na Fakultetu tehničkih nauka 2019. godine na smeru Primenjeno softversko inženjerstvo.
kontakt: majadj@icloud.com