UNIVERSITATEA DE VEST DIN TIMIȘOARA
FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ
PROGRAMUL DE STUDII DE LICENȚĂ:
INFORMATICĂ APLICATĂ

LUCRARE DE LICENȚĂ

COORDONATOR:
Lect. Dr. Bonchiș Cosmin
Drd. Brîndușescu Alin

ABSOLVENT:
Szasz Karina-Elisabeta

TIMIȘOARA
2020

UNIVERSITATEA DE VEST DIN TIMIȘOARA
FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ
PROGRAMUL DE STUDII DE LICENȚĂ:
INFORMATICĂ APLICATĂ

BENEFICII ADUSE DE NOILE ȘABLOANE ARHITECTURALE FOLOSITE ÎN DEZVOLTAREA APLICAȚIILOR MOBILE ÎNTR-UN MEDIU iOS

COORDONATOR:
Lect. Dr. Bonchiș Cosmin
Drd. Brîndușescu Alin

ABSOLVENT:
Szasz Karina-Elisabeta

TIMIȘOARA
2020

Abstract

MVC[1] and MVVM[2] are two of the most well known architectural design patterns in mobile application development. This thesis aims to analyze, compare and use these architectural patterns in a mobile application build in an iOS[3] environment. The application is written in Swift programming language and has the purpose of getting medical data from a public API[4] available on ORTHANC server.

[1]Model-View-Controller
[2]Model-View-Viewmodel
[3]Mobile operating system developed by Apple
[4]Application programming interface

Introducere

În cadrul universului dezvoltatorilor de software care utilizează programarea orientată pe obiecte, propunearea folosirii șabloanelor de proiectare a aparținut arhitectului englez-american Christopher Alexander cunoscut ca tatăl inițiativei limbajului șabloanelor. Acesta este profesor emerit în arhitectură la Universitatea Berkeley din California, fiind cunoscut în domeniul software prin cartea sa A Pattern Language, care a fost printre primele cărți scrise cu ajutorul tehnologiei hypertext fashion. Hypertext este o metodă de afișare și scriere a unui text, indicat pe un calculator sau un dispozitiv electronic, în care se folosesc referințe care duc către alte surse sau chiar texte, astfel cititorul având libertatea să aleagă modul în care citește, dar având acces și la eventuale explicații. Printre spusele lui regăsim și afirmația: "Fiecare șablon descrie o problemă care apare mereu în domeniul nostru de activitate și indică esența soluției acelei probleme, într-un mod care permite utilizarea soluției de nenumărate ori în contexte diferite". Cu toate că în domeniul programării orientate pe obiecte soluțiile sunt exemplificate prin clase, interfețe, obiecte, etc. nu geamuri, ziduri, acoperișe, în esență un șablon se poate definii astfel: o soluție a problemei date într-un anumit context.

În prezent există numeroase cărți care discută despre șabloane fie ele arhitecturale sau de proiectare, din păcate, într-o proporție majoritară, fiecare carte are același pattern: analizează și explică un sablon în parte. Cu toate acestea, cărțile sunt foarte abundente în informație, fiind explicat în amănunt și cel mai mic detaliu, dar ca să se parcurgă întreaga explicație este nevoie să se aloce un timp mai îndelungat de studiu. Din acest motiv, informațiile ce le ofereau odată cărțile, au fost mutate în mediul online, unde oricine poate accesa orice informație, fara a pierde prea mult timp. Datorită accesări în masă a informațiilor disponibile, acest subiect este în continuă expansiune, lucru care a condus și la abordarea unor noi subiecte de discuție, ca de exemplu compararea mai multor șabloane arhitecturale și sublinierea celui mai eficient șablon.

Așadar datorită mediului online, acum există numeroase abordări asupra subiectului MVC vs MVVM. De cele mai multe ori este aleasă compararea structurilor celor două șabloane încercând astfel să se evidențieze diferențele modului de implementare ale acestora, nu și modul diferit în care acestea lucrează. O altă abordare foarte des întâlnită este transformarea din MVC în MVVM și invers. Ambele abordări sunt bune, dar niciuna dintre ele nu face o comparație exactă bazată pe aceeași aplicație, ci explică modul de utilizare a celor două concepte. Din această cauză, mulți dintre tinerii dezvoltatori sunt derutați, atunci când vine vorba de diferența dintre cele două, care șablon este mai potrivit pentru ceea ce au ei nevoie, ce trebuie să conțină fiecare clasă, cum se fac legăturile între clase și metode, ce se adaugă și ce se elimină atunci când facem conversia din MVC în MVVM, dar și care este scopul acestor șabloane arhitecturale. Una dintre marile probleme ale mediului online este validarea informației, din cauza multitudinii de abordări ale aceluiaș subiect, dar de către autori diferiți, informația devine neclară, utilizatorul ne mai putând să distingă informația corectă de cea incorectă, sau chiar de care informație are nevoie defapt.

Lucrarea abordată este una de sinteză, deoarece are ca scop compararea a doua dintre cele mai cunoscute șabloane arhitecturale folosite în dezvoltarea aplicațiilor mobile într-un mediu iOS. Aceasta este structurată în 4 capitole: Concepte teoretice, Proiectarea și implementarea aplicațiilor, MVC vs MVVM și Concluzii și direcții viitoare.

Primul capitol are rolul de a explica teoretic cele două șabloane arhitecturale, definirea teoretică a componentelor fiecăruia, rolul fiecărei componente dar și al fiecărui șablon, precum și modul de utilizare al acestora. Următorul capitol este destinat construirii aplicației folosind pe rând cele 2 șabloane. În acest capitol se va exemplifica arhitectura aplicației, se va descrie componentele aplicației, vor fi integrat diagrame UML[5], se va explica interfața cu utilizatorul și se vor evidenția beneficiile fiecărui tip de șablon utilizat. Cel de-al treilea capitol reprezintă compararea conceptelor bazată pe aplicațiile realizate cu ajutorul șabloanelor, subliniind avantajele și dezavantajele acestora. Ultimul capitol este reprezentat de concluziile și direcțiile viitoare ale temei.

Pentru a avea o analiză cat mai reală au fost construite două aplicații cu aceeași funcționalitate utilizând pe rând cele două șabloane arhitecturale. Aplicația aleasă pentru exemplificarea fiecărui șablon arhitectural este una relativ simplă, astfel încat conceptele de bază sa fie foarte bine evidențiate.

[5]Unified Modeling Language

Cuprins

Capitolul 1

Concepte teoretice

În prezent, datorită obsesiei dezvoltatorilor software de a-și ușura munca, se poate discuta despre o gamă foarte largă de șabloane arhitecturale. Aceste șabloane arhitecturale au rolul de a facilita munca necesară pentru a structura o aplicație. Principalele șabloane arhitecturale de tipul MV(X) sunt:

- MVC;
- MVP[1];
- MVVM.

Totate șabolanele de tip mv(x) pleacă de la o arhitectura pe 3 nivele, după cum urmează:

1. Models: reprezintă locul unde se află datele.
2. Views: responsabile pentru tot ceea ce vede utilizatorul. În mediul iOS tot ceea ce conține UI la început se regăsește în View.
3. Controller/Presenter/ViewModel: este intermediarul dintre Model și View, având în general rolul de a modifica datele din Model, datorită interacțiunii unui utilizator cu View-ul și reactualizarea View-ului cu modificările petrecute în Model.

1.1 MVC

Model View Controller este un șablon arhitectural, deoarece operațiile care se execută la intrarea și iesirea din program sunt spearate de către logica programului. Șabloanele arhitecturale au un mare rol în ușurarea muncii utilizatorului, dar și al întregii echipe, deoarece datorită SoC[2] navigarea și debuggingul sunt mul mai ușor de realizat fie individual, fie de către întreaga echipă.

SoC este un concept abstract foarte important pe care orice arhitect software ar trebui să îl integreze în proiectele software. Acest concept are ca scop împărțirea unui program în mai multe subsecțiuni, astfel încat fiecare dintre acestea să aibe un rol intuitiv, clar și bine stabilit încă de la început. Putem afirma că SoC stabilește o ierarhie în sistem, astfel încat fiecare parte care contribuie în acesta să își îndeplinească

[1]Model-View-Presenter
[2]Separation of Concerns

rolul cu succes, fiind adaptabilă la schimbări, menținându-se astfel un sistem organizat. În arhitectura programului, SoC este evidențiată prin limite, acestea reprezintă orice tip de constrângere aplicată unei subsecțiuni, care are sarcina de a împărți un set de sarcini ca de exemplu: utlizarea metodelor și a obiectelor, ierarhizarea directoarelor pentru a organiza resursele utilizate în aplicatie, etc. Printre avantajele acestui concept se regăsesc următoarele:

- Inexistența duplicatelor.
- Realizarea debuggingului și al navigării prin proiect sunt mul mai ușor de realizat.
- Sistemul este unul mult mai stabil, datorită mentenanței.
- Datorită seturilor de sarcini individualizate pentru fiecare componentă, acestea sunt mult mai ușor reutilizabile.
- Creșterea mentenației, dar si reutilizarea componentelor poate duce la crearea unui impact major asupra marketingului aplicației.

În MVC-ul tradițional View-ul nu are stari, acesta fiind construit după ce s-au actualizat modificările petrecute în Model. Cel mai bun exemplu este acel moment cand utilizatorul se află pe pagina de Home a unui site, iar acesta dorește să acceseze pagina de contact. Ce se întâmplă mai exact:

- Utilizatorul apasă pe opțiunea Contact din meniul site-ului.
- Site-ul verifică inputul dat de către utilizator.
- Odată percepută comanda, site-ul reactualizează pagina, afișându-se astfel pagina de contact.

Vorbind din perspectiva oricărui dezvoltator de aplicații începător în mediul iOS, la început există multe noțiuni pe care acesta trebuie le stăpânească, cum ar fi: un limbaj de programare nou, folosirea API-urilor, framework-uri noi, precum și cel mai folosit șablon arhitectural Model-View-Controller. Din păcate, atunci când vorbim de programarea în mediul iOS, acestui șablon nu îi este dată prea multă atenție. Deși este un concept foarte comun pentru iOS, fiind neglijat poate cauza diverse probleme pe parcursul dezvoltării aplicației.

Precum s-a evidențiat mai sus, MVC-ul folosit în mediul iOS este tot un șablon arhitectural bazat pe trei componete, cu mici diferențe. Astfel deosebim:

1. Model: este locul unde se află informațiile, obiectele cu care se lucrează, acțiunile de parsare a datelor, codurile specifice rețelelor, manageri. De exemplu avem obiectul Animal, despre care cunoaștem informații despre: tip, înălțime, culoare și mâncare, așadar vom avea un model care conține detaliile obiectului Animal.
2. View: este interfața cu utilizatorul. Aici este locul în care se vor găsi toate informațiile legate de afișare, dar și toate datele transmise de către utilizator(date de intrare sau de ieșire). Clasele aflate aici se pot reutiliza cu ușurință, deoarece nu au în conținut nicio logică specifică pentru domeniu. În cadrul aplicațiilor iOS, în View sunt regăsite toate elementele ce încep cu UI. De exemplu UILabel, este o etichetă, un view care afișează pe ecran un text, aceasta este reutilizabilă, dar și extensibilă.

3. Controller: reprezintă locul în care se realizează legătura dintre primele două componente Model și View, fiind un intermediar între cele două. În esență, aici se înregistrează datele preluate de la utilizator cu ajutorul View-ului, acestea sunt verificate, iar apoi se actualizează Modelul, folosind datele preluate din View. Ideal, este ca această componentă să nu știe efectiv ceea ce se întâmplă în View, în schimb va comunica cu un protocol. Un exemplu foarte bun sunt UITableView-urile, care comunica cu datele lor, cu ajutorul unui protocol numit UITableViewDataSource.

O diagramă care descrie ce fac cele 3 componente ar arăta așa:

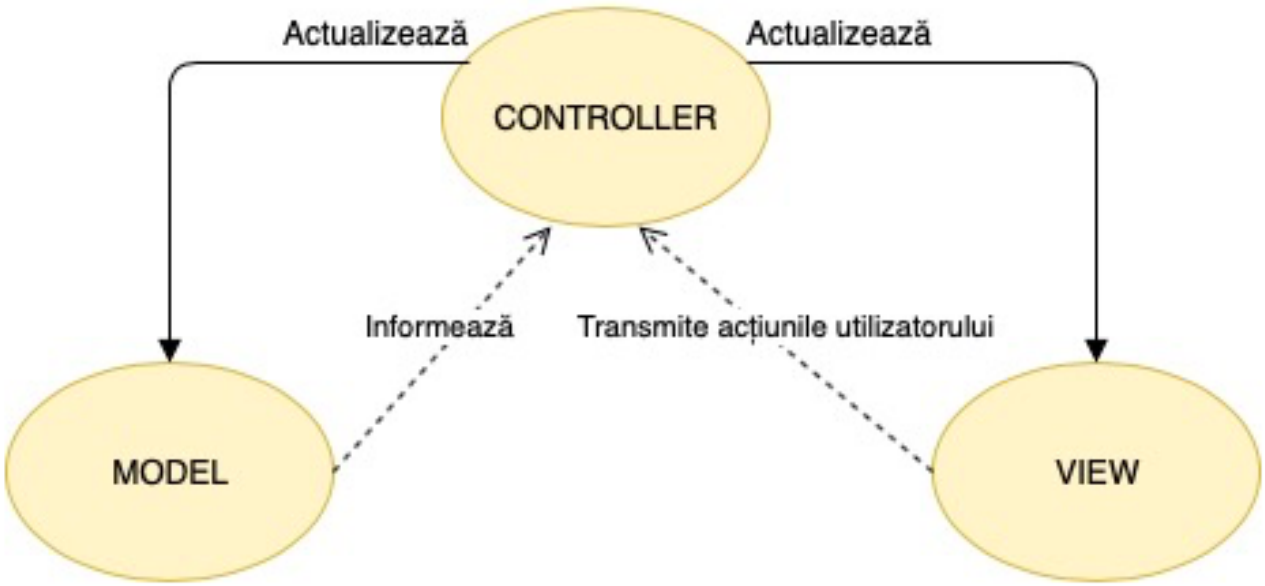

Fig. 1.1 Diagrama MVC

1.1.1 Model-ul (M)

Componenta model este o bază de date a aplicației, deoarece conține datele cu care aplicația va lucra. Acesta mai poate conține:

- Constante. Este ideal să existe un loc, un fișier, în care să existe constante, plasate atât în variabile cât și într-o structură.
- Cod pentru rețea (Network code). Pentru a face mai ușoară abstractizarea conceptelor cererilor de rețea, anteturile Http, răspunsul acestora, dar și gestionarea erorilor, este indicat să se utilizeze o singură clasă pentru întreaga aplicație care să aibe rolul de a comunica în rețea.
- Codul pentru parsarea datelor (Parsing code). Aici, în model, trebuie incluse obiectele care se ocupă cu această acțiune, deoarece partea de networking nu trebuie să știe toate detaliile despre obiectele aflate în model, pentru a reuși să le parseze.

Modelul este foarte bun atunci când vine vorba de testarea programului, deoarece se pot face foarte ușor teste care confirmă logica de business a programului, dar și asupra pieselor mai sensibile ale stratului de model.

1.1.2 View-ul (V)

Atunci când se vorbește despre View, se vorbește despre interacțiunea pe care utilizatorul o are cu aplicația. Astfel, în materie de cod se vor regăsi:

- Clasele ce aparțin UIKit/AppKit.
- Subclasele UIView-ului. Acestea putând fi clasice sau chiar personalizate.
- Animațiile de bază.
- Graficile de bază.

De regulă, componentele View-ului pot fi refolosite, acest lucru nefiind obligatoriu, decât atunci când se lucrează cu mai multe cazuri de utilizare și este necesară o componentă mai generală. Din punctul de vedere al testării, componentele UI au interacțiuni UI sau atribute specifice care funcționează potrivit țintei, astfel permițând doar testarea tranzițiilor.

1.1.3 Controller-ul (C)

Controller-ul reprezintă componenta cea mai puțin reutilizabilă, deoarece conține regulile specifice aplicației. Așadar, unei alte aplicații poate nu va avea sens ceea ce este în aplicația dezvoltată acum. Acesta va defini fluxul de informații regăsite în aplicație, folosindu-se de toate datele regăsite în Model. În Controller se vor regăsi, de obicei, răspunsurile la următoarele posibile întrebări:

- Cum interacționează subcomponentele între ele?
- De câte ori trebuie reîmprospătată aplicația?
- Utilizatorul a apăsat un buton/o celulă/ o imagine, ce trebuie să se întample în continuare?
- Care va fi următorul ecran și care va fi scopul acestuia?

Aceasta este componenta care ar trebui să fie cea mai testată. Datorită faptului că este puternic legată de Model și View este foarte greu de testat, aceasta fiind una din problemele MVC-ului. Controller-ul decide ce se va întâmpla în continuare, este micul motoraș al aplicației, care o face să funcționeze.

1.2 MVVM

Șablonul arhitectural MVVM este unul dintre cele mai noi din seria șabloanelor de tipul MV(X), fiind propus în anul 2005 de către Johnson Gossman, unul dintre arhitecții Silverlight și Microsoft WPF.

Înainte să stabilească pe care șablon să îl utilizeze în crearea aplicației, orice dezvoltator ar trebui să înceapă prin a pune următoarele întrebări:

- Se va colabora cu un alt dezvoltator sau designer, astfel trebuind ca proiectul să fie partajat?

- Cât de importantă este flexibilitatea proiectului?
- Dezvoltarea se dorește a fi una rapidă, simultană?
- Este necesară o testare minuțioasă?
- Se dorește refolosirea componentelor, atât în cadrul proiectului cât și în altele?
- Se vrea ca interfața cu utilizatorul să fie ușor editabilă, fără a reface prea mult codul?

Dacă răspunsurile au fost preponderent pozitive, atunci șablonul arhitectural MVVM, poate fi de ajutor. Acesta este un derivat al șablonului MVC, care urmărește aceeași structură: împărțirea întregului program în trei mari componente. Cele trei componente sunt:

- Model: care la fel ca în cazul șablonului MVC, se ocupă de date.
- View: afișează ceea ce vede utilizatorul; elementele vizuale și controalele de pe ecran.
- ViewModel: este mediatorul dintre View și Model, transformând informațiile primite în valori care se pot afișa cu ajutorul View-ului.

În diagrama următoare se poate observa structura șablonului arhitectural MVVM:

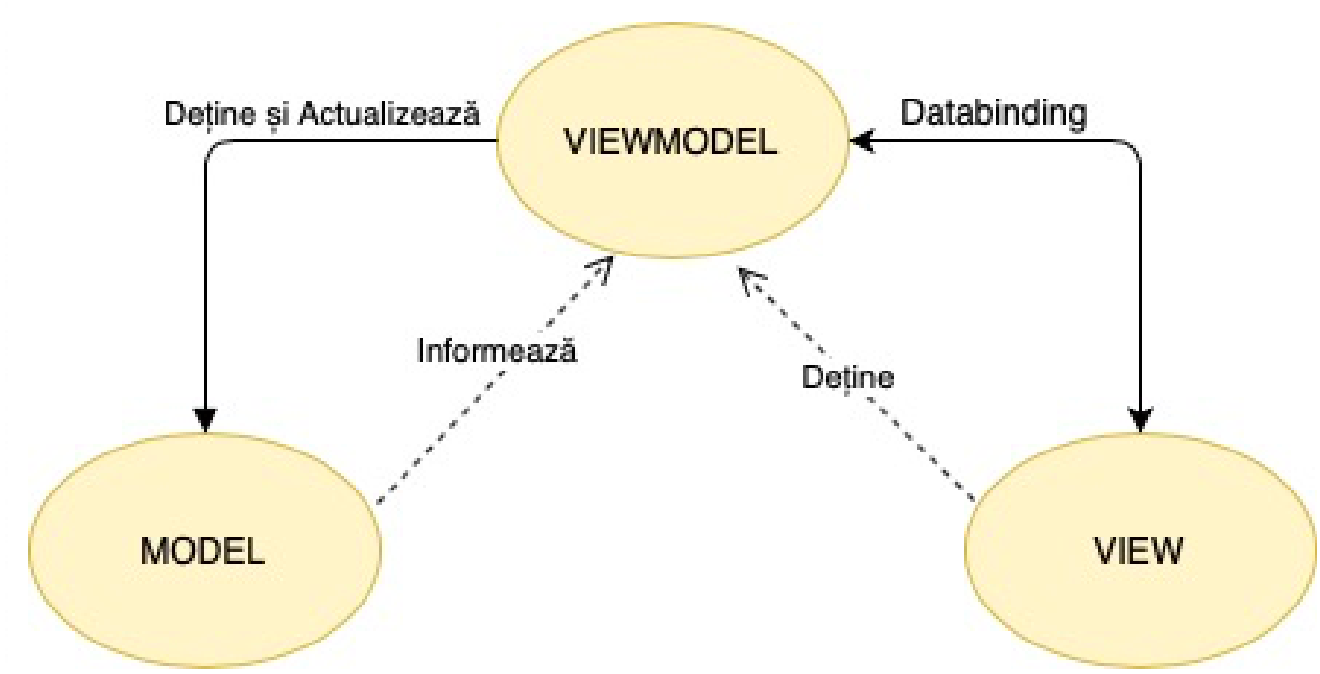

Fig. 1.2 Diagrama MVVM

1.2.1 Model-ul (M)

Modelul reprezintă totalitatea obiectelor și a datelor pe care dezvoltatorul, dar și aplicația le folosește. Un foarte bun exemplu de model este chiar unul dintre contactele din agenda telefonică, acesta conține:

- Numele persoanei.
- Numărul de telefon.
- Adresa de email.
- Domiciliul.

Modelul din MVVM este foarte asemănător cu cel din MVC. Deosebirea componentei Model de celelalte două constă în faptul că acesta deține doar informațiile, nu și serviciile, actiunile sau comportamentele care manipulează aceste informații, care defapt sunt datele cu care se lucrează. Acesta nu este neapărat responsabil pentru:

- Formatarea unui text sau a unei imagini, astfel încât acestea să se încadreze pe ecran.
- Obținerea datelor de pe un server.

Logica aplicației este de preferat să fie separat de această componentă și integrată în alta care operează direct asupra Model-ului.

1.2.2 View-ul (V)

Componenta View este cea mai cunoscută unui utilizator, deoarece este singurul lucru cu care acesta poate interacționa. În traducere, The View înseamnă vedere, așadar această componentă se ocupă cu ceea ce poate vedea și interacționa oricare utilizator. Se poate spune, că aceasta este reprezentarea datelor întregului program.

Un bun exemplu este stocarea pe model a unei date sub forma unui număr de secunde de la miezul nopții din 1 ianuarie 1998(unix time stamp). Utilizatorului, însă nu i se va afișa un lanț de cifre, ci pe ecran va apărea ziua, luna și anul în fusul orar local. Tot aici în vedere se primesc și interacțiunile acesteia cu utilizatorul care mai apoi modifică modelul: apăsarea unui buton, a unei taste, mișcările mouse-ului sau diferite gesturi tactile.

Spre deosebire de View-ul din MVC, aici în MVVM, View-ul este activ, adică nu este complet manipulată de Controller sau Presenter, acesta conține diverse comportamente, legături de date și evenimente care necesită raportarea la Model sau chiar ViewModel. Este responsabilă atunci când vine vorba despre gestionarea propriilor evenimente, nelăsând totul pe seama ViewModel-ului.

View-ul este reprezentat în mare parte de:

- Obiectele din UIView și UiViewController.
- Fișierele .xib și .storyboard

1.2.3 ViewModel-ul (VM)

ViewModel-ul intermediarul, calea de acces prin care datele trec din View în Model sau invers. Este piesa cheie a acestui șablon arhitectural, deoarece face posibilă separarea View-ul de Model. Așadar Model-ul nu conștientizează asupra modului în care data este vizualizată, View-ul păstrează data gata formată, iar ViewModel-ul controlează ceea ce se întâmplă între cei doi. ViewModel-ul ar putea să preia comenzile primite de la utilizator prin View și să le transmită mai departe la Model. Această componentă mai conține și diverse metode sau comenzi care ajută la păstrarea View-ul, modificând Model-ul în urma acțiunilor primite de la View.

1. Model și ViewModel:
    - Model-ul sau alte proprietăți ale acestuia sunt expuse direct de către ViewModel
    - ViewModel-ul poate avea diverse interfețe pentru servicii, configurări de date, pentru a manipula datele.
2. View și ViewModel:
    - Cele două comunică prin apeluri ale unor metode, mesaje, evenimente, proprietăți.
    - Proprietătile ViewModel-ului și Model-ul sunt actualizate datorită View-ului.
    - View-ul conține propriile lui evenimente UI, tranmițându-le prin comenzi ViewModel-ului.

Șablonul arhitectural MVVM este foarte bun atunci când vine vorba despre testare, deoarece este NSObject sau struct, neavând o legătură cu UIKit-ul. Așadar se poate testa mult mai ușor fără să se afecteze UI code-ul.

Capitolul 2

Proiectarea și implementarea aplicațiilor

2.1 Tehnologii folosite

Pentru a fi posibilă realizarea acestei lucrări de licență, a fost necesară dezvoltarea a două aplicații în mediul iOS.

2.1.1 Dezvoltarea aplicațiilor iOS

Sistemul de operare care rulează pe dispozitivele hardware iPhone, iPod Touch, iPad, tvOS, watchOS este sistemul de operare mobil iOS dezvoltat de Apple Inc. Acest sistem de operare mobil este al doilea în lume, fiind clasat exact după Android. Există câteva cerințe care trebuie îndeplinite de către un dezvoltator de aplicații iOS, cu toate acestea trebuie evidențiat faptul că publicul căruia se adresează acest tip de aplicație este unul foarte larg, peste 700 de milioane de utilizatori anual. Un dezvoltator are următoarele beneficii atunci când începe dezvoltarea unei aplicații în mediul iOS:

- Dezvoltarea unei aplicații pentru sute de milioane de utilizatori în întreaga lume.
- Dezvoltarea unei aplicații iOS este realtiv ușoară, de multe ori mai ușoară și clară decât dezvoltarea pentru platforma Android.
- Pentru ca dezvoltarea să fie eficientă s-au pus la dispozitie API-uri și biblioteci extinse.
- Instrumentele dispuse de Xcode vor face ca testarea să fie mult mai simplificată și eficientă.

iOS Software Development Kit (SDK) face posibilă dezvoltarea aplicațiilor în acest mediu, datorită tehnologiilor, limbajelor și ale instumentelor pe care le are în componență. Framework-urile Cocoa Touch sunt printre cele mai esențiale atunci când vorbim de crearea unei aplicații și conțin:

- Intstrumentele UIKit.
- Game Kit.
- Foundation Kit.

- Map Kit.

Toate acestea fac posibilă interacțiunea vocală cu ajutorul SiriKit, explorarea muzicii utilizând MusicKit, manipularea camerei dispozitivului, adăugarea chat-ului iMessage Business, extinderea vizualizării și a ascultării utilizând AirPlay.

2.1.2 Cerințele acestui tip de dezvoltare

Dezvoltarea aplicațiilor în mediul iOS vine la pachet cu anumite cerințe, fără de care aceasta nu ar fi posibilă. Înainte de începerea propriu-zisă a aplicației, fiecare dezvoltator trebuie să posede:

- Un calculator sau un laptop care să aibe sistem de operare macOS (Apple Mac, MacBook Pro, MacBook Air).
- Mediul de dezvoltare integrat (IDE) Xcode trebuie să fie instalat. Xcode rulează doar pe platforma macOS, iar această platformă este disponibilă doar pentru calculatoarele sau laptopurile produse de Apple.
- Un cont Apple Developer activ, pentru care se plătește o taxă anuală, nefiind nevoie să se plătească în cazul în care nu se dorește publicarea aplicației.Dar datorită acestui abonament și doar dacă acesta este activ, există posibilitatea de a se publica aplicația în cadrul Apple App Store. În App Store se regăsesc doar aplicații semnate și publicate de către Xcode.
- Stăpânirea unuia dintre cele două limbaje de programare disponibile pentru dezvoltarea acestor aplicații:
  - Swift.
  - Objective-C.

2.1.3 Limbaje de programare

IDE-ul Xcode poate suporta patru limbaje de programare:

- Swift.
- Objective-C.
- C.
- C++.

Pentru dezvoltarea aplicațiilor doar primele două limbaje de programare pot fi utilizate.

1. Objective-C a fost limbajul de programare prioritar și primar pentru produsele oferite de către Apple, fiind dezvoltat în anul 1980. Derivat din clasicul limbaj de programare C, Objective-C este un limbaj de programare orientat pe obiecte, având și un timp de rulare dinamic. Acesta moștenește sintaxa, instrucțiunile de control și tipurile primitive ale C-ului și adaugă sintaxele pentru definirea metodelor și ale claselor. Oferă o testare dinamică.

2. Swift a devenit noul limbaj de programare oficial al dezvoltării în mediul iOS. Acesta are multe asemănări cu vechiul Objective-C, însă Swift a fost proiectat, astfel încât sintaxa să fie mai simplificată deci mai ușoară și securitatea să fie mai crescută decât în cazul predecesorului său.

Printre diferențele sau asemănările dintre cele două limbaje de programare se enumeră următoarele legate de diferite subiecte:

1. Design.
   - Objective-C a fost din start creat pentru a fi orientat spre trimiterea de mesaje similare limbajului de programare SmalTalk.
   - Swift a fost creat pentru a realiza sistemele de operare pentru Apple.
2. Moștenire.
   - Objective-C nu suportă moștenirea multiplă.
   - Swift nu suportă moștenirea multiplă.
3. Paradigme.
   - Objective-C folosește mesaje ca să apeleze funcționalitățile implementate, precum erau implementate folosind mesageria SmallTalk.
   - Swift declară metode de tipul type-level pentru a apela funcționalitățile implementate.
4. Clase.
   - Atât clasele cât și structurile sunt tratate diferit în Objective-C.
   - În Swift există și clase, structuri, enumerații, toate acestea fiind "first class citizens", adică sunt tratate într-un mod similar cu clasele, atunci când vine vorba de conceptele OOP .
5. Licența.
   - Objective-C este licențiat de General Public License.
   - Swift este open source sub licență Apache.
6. Scriere.
   - Objective-C are o scriere dinamică.
   - Swift are o scriere statică.
7. Operatori de tip Boolean.
   - În Objective-C există YES, NO and BOOLEAN.
   - În Swift există ADEVĂRAT sau FALS.

Cele două aplicații dezvoltate pe rând cu fiecare dintre cele două șabloane arhitecturale propuse au fost scrise în limbajul de programare Swift. Cea mai bună introducere este "Swift is a powerful and intuitive programming language for macOS, iOS, watchOS, tvOS and beyond. Writing Swift code is interactive and fun, the syntax is concise yet expressive, and Swift includes modern features developers love. Swift code is safe by design, yet also produces software that runs lightning-fast.", fiind expusă pe site-ul oficial[1]. Limbajul Swift este unul foarte prietenos cu noii dezvoltatori, fiind un limbaj placut și foarte expresiv la fel ca un limbaj de script. Este foarte rapid, interactiv și sigur care combină perfect noțiunile vechi preluate de la limbajele precedente cu noile tehnologii. Totodata, compilatorul a fost optimizat alături de limbajul care a fost optimizat și el pentru dezvoltare. Clasele majore de erori în programare sunt definite de către Swift prin adaptarea unor noi șabloane de programare.

1. Erorile de tipul out-of-bounds sunt verificate în cadrul indicilor vectorilor.
2. Variabile trebuie inițializate înainte ca acestea să fie declarate.
3. Overflow-ul este verificat pentru întregi.
4. Valorile nule sunt gestionate de către opționale în mod explicit.
5. Gestionarea memoriei se produce automat.

Optimizarea codului a fost făcută tocmai pentru a beneficia de tot ceea ce ține de noile hardware-uri. Pentru a avea cea mai bună performanță, au fost concepute diferite sintaxe și bibleoteci, astfel încât codul să funționeze cât mai rapid și bine. Limbajul de programare Swift este în proiectare de mulți ani, dar este într-o continuă dezvoltare încă și acum.

2.1.4 IDE-ul Xcode

Xcode este mediul de dezvoltare integrat al firmei Apple, fiind folosit pentru crearea aplicațiilor care rulează produsele fabricate de Apple: iPhone, Mac, Macbook Air, MacBook Pro, iPad, Apple Watch). Cu ajutorul instrumentelor pe care IDE-ul Xcode le are, se poate cu ușurință gestiona întregul proces de dezvoltare al aplicației:

- Crearea aplicației.
- Testarea aplicației.
- Optimizarea aplicației.
- Postarea directă a aplicației pe Github.
- Postarea directă a aplicației în App Store.

Acesta poate fi rulat doar pe Mac și poate fi descărcat direct din App Store.

[1]https://developer.apple.com/swift/

2.1.5 DICOM

Atunci când vine vorba despre imagistica medicală, standardul internațional impus pentru procesarea, stocarea, tipărirea, transmiterea, preluarea și afișarea informațiilor legate de imagistica medicală este DICOM[2]. Foarte multe dintre dispozitivele medicale folosesc DICOM. Acesta definește diversele formate pentru imaginile medicale, putând fi astfel schimbate cu datele pentru utilizarea clinică. Domeniile care îl folosesc cel mai mult fiind:

- Radiologie.
- Cardiologie.
- Imagistică.
- Radioterapie.
- Oftalmologie.
- Stomatologie.

2.1.6 ORTHANC

Scopul ORTHANC-ului este de a oferi un server de DICOM autonom și simplu. Datorită acestuia,formatul complex DICOM este ascuns, astfel utilizatorii acestui server se pot concentra pe conținutul acestor fișiere. Funcția majoră a acestuia este transmiterea unui REST API, datorită acestuia, Orthanc poate fi accesat cu ajutorul orcărui limbaj de pe calculator, devenind astfel o bază de date pentru aplicația ce urmează a fi creată. Etichetele imaginilor medicale stocate pot fi accesate, descărcându-le sub forma unui fișier JSON

2.1.7 REST API

RESTful sau REST API (Representational State Transfer) a fost creat tocmai pentru a putea profita de toate protocoalele existente. REST în schimb, se poate utiliza peste toate tipurile de protocoale, folosindu-se de obicei în cadrul API-urilor WEB. Aici, metodele și resursele nu sunt legate de date, REST putând:

- Să gestioneze mai multe tipuri de apeluri.
- Să returneze diferitele formate de date.
- Să schimbe structura astfel încat să fie implementată corect hypermedia.

Datorită flexibilității acestuia, REST poate returna oricare dintre formatele JSON, XML, YAML.

[2]Digital Imaging and Communications in Medicine

Cele șase constrângeri ale REST API-ului, de care trebuie ținut cont atunci când se dorește introducerea lui în proiect:

1. Client-Server
2. Stateless
3. Cache
4. Uniform Interface
5. Layered System
6. Code on Demand

2.2 Prezentarea aplicațiilor

Realizarea celor două aplicații a fost făcută în IDE-ul Xcode, iar ca limbaj de programare a fost folosit Swift. Aplicația are scopul de a prelua informațiile medicale utilizând un API[3] public, oferit de către serverul ORTHANC, care este ca o bază de date pentru aceste aplicații. Cele două aplicații sunt identice având fiecare patru ecrane:

1. Primul ecran: aici apare lista cu toți pacienții.
2. Al doilea ecran: pe acest ecran sunt afișate detalii despre pacientul pe care utilizatorul l-a ales, dar și o listă de analize care poate fi și ea la rândul ei accesată.
3. Al treilea ecran: aici se regăsesc detaliile despre analiza selectată, precum și lista seriilor, care conduce spre ecranul final.
4. Al patrulea ecran: este și ultimul ecran al aplicației, aici găsindu-se detalii despre seria selectată.

[3]Application programming interface

2.3 Aplicația MVC Application

2.3.1 Arhitectura aplicației

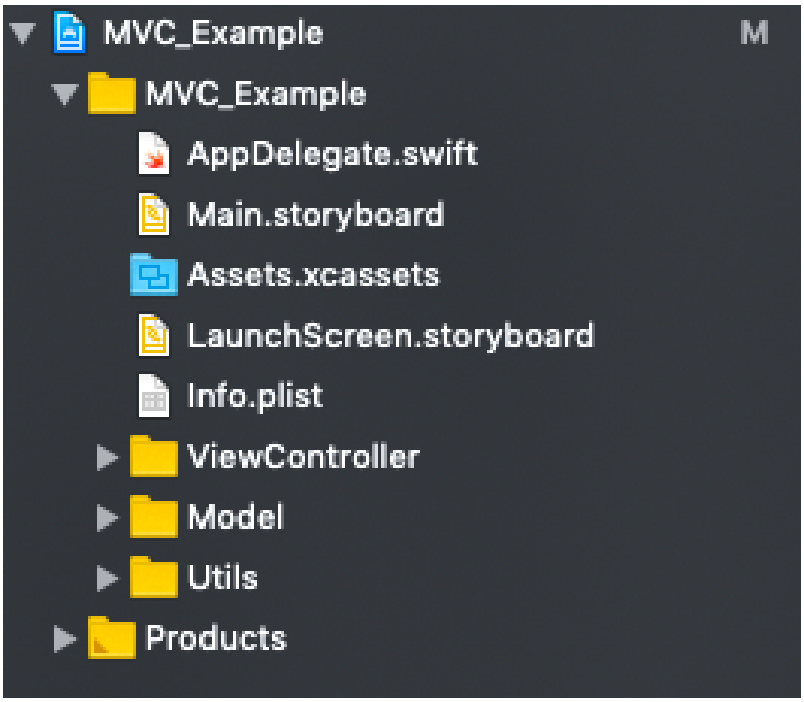

Fig. 2.3.1 Structura aplicației MVC Application.

Aplicația MVC Application este împărțită conform șablonului arhtectural al carui nume îl poartă. Așadar avem următoarele componente:

1. Model: aici regăsindu-se reprezentarea abstractă a datelor..
2. View: este responsabil pentru ceea ce utilizatorul vede pe ecran.
3. ViewController: legătura dintre View și Model se realizează aici.
4. Utils: aici se regăsesc clasele care se ocupă cu partea de networking

2.3.2 Descrierea componentelor

Model-ul are în componeță următoarele clase:

- Patient.
- Study.
- Serie.
- IdModel.

ViewController-ul conține următoarele clase:

- PatientDetailsViewController.
- SeriesDetailsViewController.
- PatientViewController.
- StudyDetailsViewController.

Utils are următoarele clase în componență:

- Constant.
- APIHandlerSeries.
- APIHandlerPatient.
- APIHandlerStudy.

View este reprezentat de Main.storyboard.

2.3.3 Interfața cu utilizatorul

În prezent orice sistem de operare sau aplicație software deține o interfață prin intermediul căreia utilizatorul poate interacționa, realizându-se astfel o comunicare între sistem/aplicație și utilizator. În cadrul dezvoltării aplicațiilor pentru sistemul de operare iOS, interfețele destinate utilizatorilor se realizaează cu ajutorul Storyboard-urilor. Acestea au fost introduse odată cu apariția iOS 5 . În esență, un sotryboard poate fi definit ca "o reprezentare vizuală a unei interfețe cu utilizatorul a unei aplicații iOS.", precum se regăsește pe site-ul oficial[4]. Datorită lor se pot proiecta într-un singur fișier mai multe controller view-uri, permițând totodată și crearea tranzițiilor ce leagă controller view-urile create, formându-se astfel o secvență de scene. Fiecare scenă reprezintă un singur ecran pe care utilizatorul îl poate vedea, acestea fiind alcătuite fiecare dintr-un view controller.

Xcode pune la dispoziție un editor vizual pentru storyboard-uri, tocmai pentru a putea proiecta fiecare ecran al aplicației dezvoltate, adăugând diferite componente: text, butoane, tabele, ș.a.ș.m.d. Pe lângă toate acestea, datorită lor se pot conecta view-uri la obiectele de tipul controller, astfel gestionându-se mult mai ușor informațiile dintre view și controller. Este indicată folosirea storyboard-urilor, deoarece totul se întâmpla în același loc pe aceeași pânză.

În cadrul aplicației MVC Application, tot ceea ce ține de interfață aparține componentei View. Aplicația are patru ecrane precum urmează:

[4]https://developer.apple.com/library/archive/documentation/General/Conceptual/Devpedia-CocoaApp/Storyboard.html

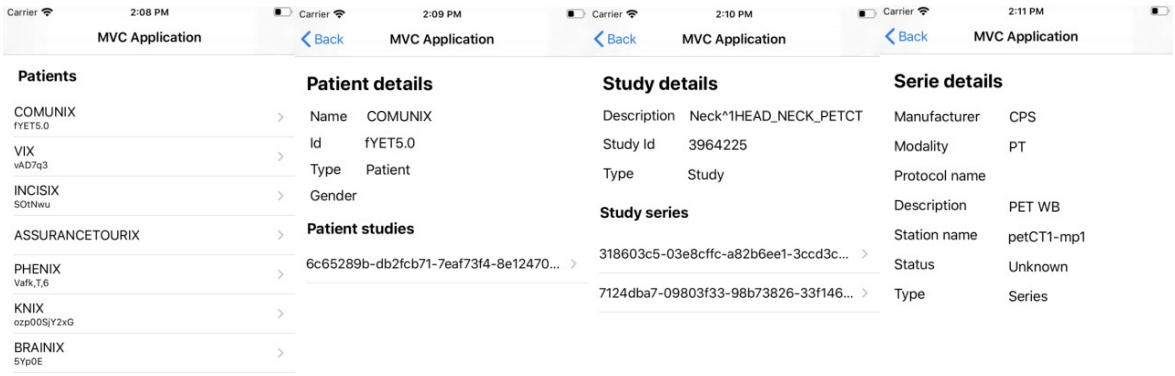

Fig. 2.3.3 Ecranele aplicației MVC Application.

Fiecare ecran are în componență mai multe elemente ce fac posibilă interacțiunea cu utilizatorul după cum urmează:

1. Ecranul principal conține:
    - Un view controller, care oferă funcționalitate și gestionează componentele de tipul application views, toolbars sau navigation bars.
    - Un table view, în cadrul căruia va apărea lista cu toți pacienții.
    - O celulă a table view-ului, care va conține câte un pacient.
    - Conținutul celulei, reprezentat printr-un view și care are două etichete Titel și Subtitle.
    - Un navigator, unde este scris numele aplicației.
    - Diferite constrângeri pentru toate elementele astfel încât fiecare componentă să fie la locul ei.
2. Al doilea Ecran conține:
    - Un view controller.
    - Etichete pentru diverse informații despre pacient.
    - Un table view.
    - O celulă, în cadrul căreia va apărea lista cu analizele medicale ale pacientului selectat.
    - Un navigator .
    - Constrângeri.
3. Al treilea Ecran conține:
    - Un view controller.
    - Etichete aferente informațiilor despre analizele medicale făcute.

- Un table view.
- O celula, unde se vor posta toate seriile analizelor medicale.
- Un navigator.
- Diferite constrângeri.

4. Ultimul ecran conține:
    - Un view controller.
    - Diverse etichete pentru informațiile legate de serii.
    - Un navigator.
    - Constrângeri.

2.3.4 Beneficii

Șablonul arhitecturat Model-View-Controller, cunoscut și ca MVC, este un șablon specializat în proiectarea interefețelor pe calculator. Indiferent de limbajul de programare folosit, MVC va fi mereu un șablon arhitectural excelent. Printre beneficiile acestuia se regăsesc:

- Procesul de dezvoltare devine mult mai rapid. De exemplu dacă pentru realizarea unei aplicații web se lucrează în echipă, atunci există posibilitatea ca unul dintre programatori să lucreze la partea de interfață, în timp ce cel de-al doilea programator poate lucra la dezvoltarea controller-ului și a logicii programului. Astfel aplicația dezvoltată va fi mult mai rapid gătată folosind acest șablon.

- Există posibilitatea ca pentru un singur model să existe mai multe view-uri. În ziua de azi, există o cerere din ce în ce mai mare pentru ceea ce ține de modul de accesare al aplicației dezvoltate, iar datorită separării propuse de MVC, acest lucru este mult mai ușor realizabil.

- Modificarea interfeței nu va afecta întregul program. Atunci când se dorește îmbunătățirea aspectului aplicației de exemplu schimbarea culorilor, textului, aranjării elementelor de pe ecran, modificarea va fi mult mai ușor de realizat, deoarece orice modificare a modelului sau a view-ului nu va afecta întregul program.

- Modelul MVC va returna mereu datele fără a le aplica nicio formatare, așadar oricare dintre componente poate fi utilizată în cadrul oricărei interfețe.

2.4 Aplicația MVVM Application

2.4.1 Arhitectura aplicației

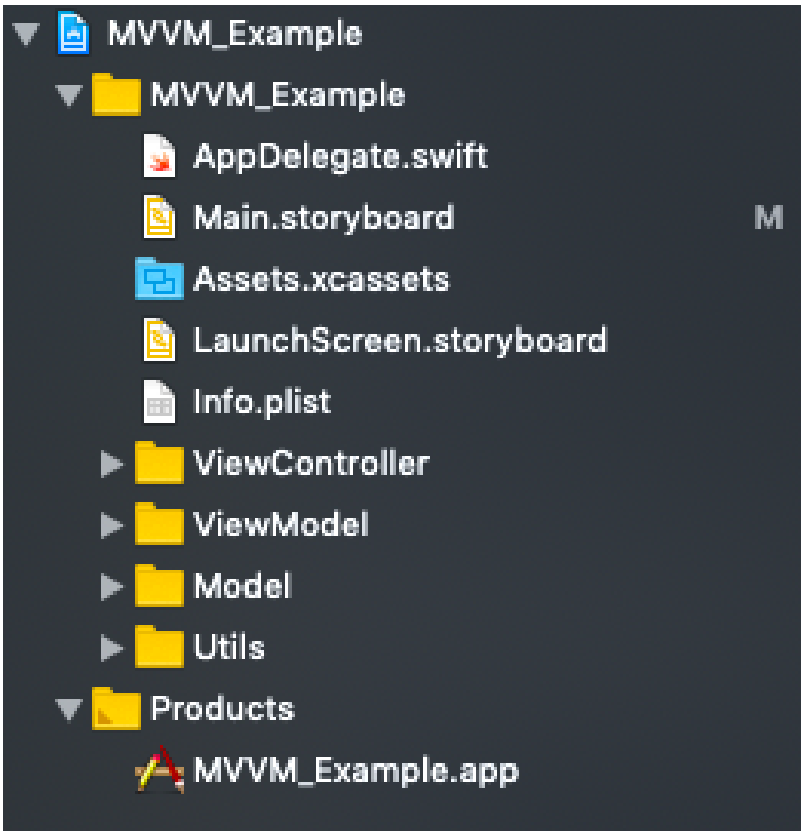

Fig. 2.4.1 Structura aplicației MVVM Application.

Aplicația MVVM Application are structura reprezentativă șablonului arhitectural MVVM, deosebindu-se componentele:

1. Model: locul unde se regăsesc datele.
2. ViewModel: realizează legătura dintre Model și View
3. View: este reprezentat de tot ceea ce vede utilizatorul pe ecran.
4. ViewController: aici se regăsesc acțiunile legate de View.
5. Utils: aici se preiau datele de pe server-ul Orthanc.

2.4.2 Descrierea componentelor

Modelul conține patru clase:

- Patient.
- Study.
- Serie.
- IdModel.

ViewModel-ul are în componență clasele:

- SeriesDetailsViewModel.
- PatientViewModel.
- PatientDetailsViewModel.
- StudyDetailsViewModel.

Main.storyboard ține locul View-ului.

ViewController este alcătuit din următoarele patru clase:

- PatientDetailsViewModel.
- SeriesDetailsViewController.
- PatientViewController.
- StudyDetailsViewController.

Clasele ce alcătuiesc Utils sunt:

- Constant.
- APIHandlerSeries.
- APIHandlerPatient.
- APIHandlerStudy.

2.4.3 Interfața cu utilizatorul

Tocmai, pentru a sublinia faptul că, interfața cu utilizatorul este un concept separat de crearea codului, aplicația MVVM Application are aceași interfață cu utilizatorul precum o are și MVC Application. Pentru aplicația MVVM Application, interfața cu utilizatorul este realizată tot de către componenta View. Aceasta a fost realizată tot în cadrul Main.storyboard, folosindu-se de aceleași componente. Așadar toate componentele ce se pot regăsi în crearea celor patru ecrane sunt:

- View controller.
- Navigator.
- Table view.
- Celule.
- Etichete.
- Constrângeri.

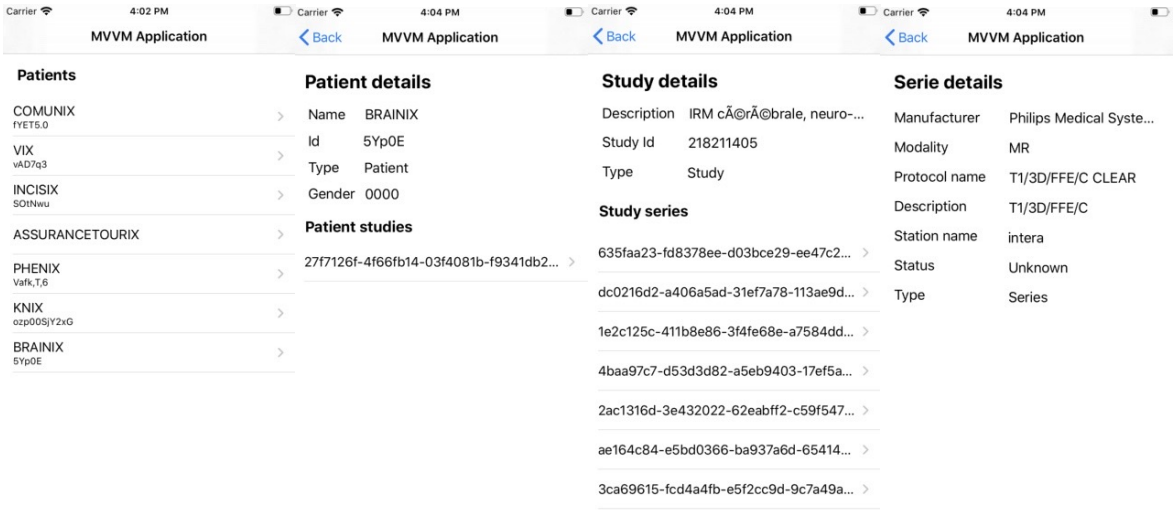

Fig. 2.4.3 Ecranele aplicației MVVM Application.

2.4.4 Beneficii

Șablonul arhitectural Model-View-ViewModel, pe scurt MVVM, este cel mai modern din seria șabloanelor dezvoltate din clasicul MVC, având același obiectiv separarea programului. Cel mai notabil lucru la acesa este despărțirea dintre View și Model, astfel încât dacă trebuie să se modifice model-ul, se va putea face cu ușurință, deoarece nu va afecta view-ul.

Cele mai importante trei lucruri care trebuie menționate atunci când vine vorba despre beneficiile aduse de acest șablon sunt:

- Mentenabilitatea. Separarea clară a tipurilor de cod și a componentelor, duce la o modificare mult mai clară și rapidă.
- Testarea. Datorită acestui șablon, bucățile de cod sunt mult mai granulare, și nu intervine niciun detaliu de tipul UI.
- Reutilizarea. Pentru că MVVM are ca scop evident separarea view-ului de restul programului.

Capitolul 3

MVC vs MVVM

3.1 Structura MVC vs MVVM

Atât aplicația MVC Application cât și MVVM Application, sunt construite după următoarele două diagrame reprezentative celor două șabloane.

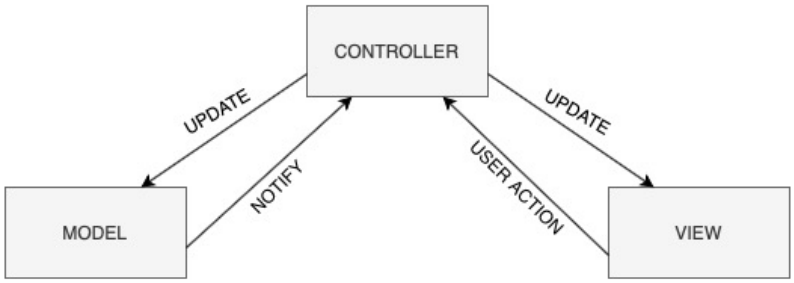

Fig. 3.1.1 Structura MVC în iOS.

Fig. 3.1.2 Structura MVVM în iOS.

Din implementarea celor două aplicații se pot observa următoarele asemănări:

- Clasele ce aparțin Model-ului, Serie, Patient, Study și IdModel sunt identice atât în MVC Application cât și în MVVM Application. Un exemplu de cod pentru aceste clase, deci pentru codul Model-ului este:

```
class Serie {
    var id: String
    var manufacturer: String
    var modality: String
    var protocolName: String
```

```
var description: String
var stationName: String
var status: String
var type: String

init() {
    self.id = ""
    self.manufacturer = ""
    self.modality = ""
    self.protocolName = ""
    self.description = ""
    self.stationName = ""
    self.status = ""
    self.type = ""
}
```

- Pachetul Utils, comun celor două aplicații, conține clasele Constant, APIHandlerSeries, APIHandlerPatient, APIHandlerStudy, clase care se ocupă de preluarea și parsarea datelor. Următoarea secvență de cod are ca scop preluarea și parsarea listei de pacienți, precum și verificarea apariției posibilelor erori fie ele de decodare sau din cauze de networking.

```
func getPatientList(withUrl strUrlP : String, completionBlock :
        @escaping completionBlock){

    if let unwrappedUrlP = URL(string: strUrlP){

        URLSession.shared.dataTask(with: unwrappedUrlP,
        ncompletionHandler: { (data, response, error) in
            var patients = [Patient]()

            if error == nil && data != nil{
                let jsonDecoder = JSONDecoder()
                do {
                    let idModels = try jsonDecoder.decode(
                        [IdModel].self, from: data!)
                    for idModel in idModels {
                        self.getPatient(withUrl:
                                strUrlP+"/"+dModel.id) { (result) in
                                patients.append(result)
                                completionBlock(patients)
                        }
                    }
                } catch {
                    patients = [Patient(id: "Decoding Error")]
                }
            } else {
                patients = [Patient(id: "Network Error")]
            }
```

```
                completionBlock(patients)
            }).resume()
        }
    }
```

Preluarea detaliilor despre pacienți, în cazul de mai sus, se realizează printr-o serie de verificări care poate fi vizualizată prin codul:

```
if let mainDicomTags = json["MainDicomTags"] as? [String: String] {
    if let name = mainDicomTags["PatientName"] {
        patient.name = name
    }
    if let patientId = mainDicomTags["PatientID"] {
        patient.patientId = patientId
    }
    if let gender = mainDicomTags["PatientSex"] {
        patient.gender = gender
    }
}
```

- Tot ceea ce a fost creat în Main.storyboard rămânde valabil și identic în ambele aplicații, având același număr de ecrane, funcționalități și design. Acest lucru reprezentând View-ul.

Diferențele majore dintre cele două se regăsesc atunci când se vorbește despre componenta ce realizează legătura dintre View și Model. Astfel avem:

- Pentru șablonul arhitectural MVC, cea dea treia componentă și practic legătura dintre celelalte două este ViewController. Aici se regăsesc cele patru clase PatientViewController, SeriesDetailsViewController, PatientDetailsViewController, StudyDetailsViewController. În următoarea secvență de cod se poate vizualiza legătura dintre View și Model:

```
class PatientDetailsViewController: UIViewController {

    @IBOutlet weak var tblView: UITableView!
    @IBOutlet weak var nameLabel: UILabel!
    @IBOutlet weak var idLabel: UILabel!
    @IBOutlet weak var typeLabel: UILabel!
    @IBOutlet weak var genLabel: UILabel!

    var patient: Patient = Patient()

    override func viewDidLoad() {
        super.viewDidLoad()

        tblView.tableFooterView = UIView()

        self.tblView.reloadData()
        self.nameLabel.text = patient.name
```

```
        self.idLabel.text = patient.patientId
        self.typeLabel.text = patient.type
        self.genLabel.text = patient.gender
    }
```

- Șablonul arhitectural MVVM are ca și componentă ce reprezintă legătura dintre primele două, ViewModel-ul. Aici sunt cele patru clase PatientDetailsViewModel, PatientViewModel, SeriesDetailsViewModel, StudyDetailsViewModel. Această componentă, am putea spune, că este o reprezentare a ViewController-ului. Dacă ViewController-ul are în componență o etichetă de exemplu, ViewModel-ul ar trebui să conțină o metodă care să o înlocuiască cu textul sub forma unui șir de caractere. Aici se declanșează, se primesc și se trimit toate datele și apelurile.

```
        class PatientDetailsViewModel {

    var patient: Patient = Patient()

    init(data: Patient){
        self.patient = data
    }

    func getPatientName() -> String {
        return patient.name
    }

    func getPatientGender() -> String {
        return patient.gender
    }

    func getPatientId() -> String {
        return patient.patientId
    }

    func getPatientType() -> String {
        return patient.type
    }

    func getNumberOfRowsInSection() -> Int {
        return patient.studies.count
    }

    func getSerieAtIndex(index : Int) -> String {

        let study = patient.studies[index]
        return study
    }

    func getCellData(index : Int) -> String {
        let study = self.getSerieAtIndex(index: index)
```

```
            return study
        }
    }
```

- Componenta View din acest șablon este alcătuită din tot ceea ce conține Main.storyboard plus întregul pachet ViewController, astfel separarea componentelor este mult mai eficientă. În ViewController vor fi configurate view-urile UI. Această componentă nu cunoaște detalii și nu interacționează cu Modelul, în schimb va trece prin ViewModel, cerându-i ViewModel-ului datele de care are nevoie, în formatul gata pentru afișare.

3.2 MVC vs MVVM

Ambele șabloane de proiectare au ca și scop principal separarea View-ului de Model și invers, cu alte cuvinte separarea a ceea ce se vede de ceea ce se întâmplă. Logica aplicației reprezintă gestionarea comunicării dintre o bază de date și interfața cu utilizatorul, asigurându-se că datele au fost procesate, modele s-au actualizat, practic partea de procesare. Aici se includ operații de tipul ștergere, actualizare, formatarea unor tipuri de date. Logica de tip UI reprezintă partea care se ocupă de ceea ce va vedea utilizat.orul, adică interfața aplicației. Aici se pot include acțiunii de tipul animarea componentelor ce se află pe ecran, actualizarea etichetelor, tabelelor, schimbarea culorilor.

Situații în care este folosit șablonului MVC:

- Se presupune că un utilizator va da swipe unui element de tipul to-do, astfel ștergându-l. În acest caz, Controller-ul va trebui să actualizeze informațiile din Model.

- Se presupune că există un temporizator care îl va informa pe utilizator asupra unor detalii de multă vreme nevizualizate. În momentul în care temporizatorul se va declanșa, Model-ul va anunța Controller-ul că în cadrul interfeței este necesară o nouă acțiune. Odată informat, Controller-ul va actualiza View-ul, apărând astfel pe interfață o alertă.

Situații în care este folosit șablonului MVVM:

- Presupunem același prim caz ca mai sus. Este evident că Model-ul trebuie actualizat, așadar ViewController-ul va încerca să facă orice pentru a actualiza View-ul. Însă pentru actualizarea Model-ului, ViewController-ul va comunica cu legătura dintre Model și View, în acest ViewModel, solicitând-ui să actualizeze Model-ul.

- În cazul celui de-al doilea exemplu, de data aceasta ViewModel-ul va putea activa o acțiune în componenta ViewController, care mai apoi va putea actualiza View-ul, deci afișându-se în cele din urmă alerta.

3.2.1 Avantaje și dezavantaje ale șablonului arhitectual MVC

Avantajele MVC:

- Dezvoltarea aplicației este mult mai rapidă.
- Colaborarea și lucrul în echipă este mult mai ușor de realizat.
- Actualizarea se realizează mult mai rapid și ușor.
- Procesul de găsire și rezolvare al erorilor sau al problemelor, care împiedică funcționarea corectă a aplicației, este facil.

Dezavantajele MVC:

- Testarea Viewcomtroller-ului este foarte dificilă.
- Viewcontroller-ul are tendința de adeveni foarte încărcat.
- La început este greu de înteles.
- Asupra metodelor există regul stricte.

3.2.2 Avantaje și dezavantaje ale șablonului arhitectual MVVM

Avantajele MVVM:

- SoC este mult mai bună.
- Testarea este mai eficientă.
- Transparența comunicării dintre componente este mult mai bună

Dezavantajele MVVM:

- De multe ori, în cadrul aplicațiilor cu interfețe de utilizator prea simple MVVM nu este cel mai potrivit.
- Similar, în unele cazuri crearea ViewModel-ului poate fi greu de realizat.
- Din păcate, reutilizarea ViewModel-ului nu este posibilă în majoritatea cazurilor.

Capitolul 4

Concluzii și direcții viitoare

S-au realizat două aplicații utilizând două dintre cele mai noi și utilizate șabloane arhitecturale, în dezvoltarea aplicațiilor în mediul iOS, evidențiindu-se fiecare componentă și rolul acesteia. Cu ajutorul fiecărei aplicații a fost posibilă exemplificare beneficiilor fiecăruia dintre cele două șabloane. Datorită prezentării celor două cazuri posibile în care se utilizează șabloanele, dar și a evidențierii atât a avantajelor cât și a dezavantajelor șablonului MVC respectiv MVVM, s-a obținut o vedere de ansamblu a celor două șabloane arhitecturale.

În viitor, cele două aplicații se vor putea extinde ușor, datorită modului de construcție al acestora, implementându-se noi funcționalități și noi elemente pentru interfața cu utilizatorul.

Bibliografie

[1] Simon NG - Beggining iOS 13 Programming with Swift, Learn how to build a real world app for iPhone and iPad from Scratch, AppCoda Limited, 2019.

[2] Michel Goossens, Frank Mittelbach, and Alexander Samarin. The LATEX Companion. Addison-Wesley, Reading, Massachusetts, 1993.

[3] `http://www.patternlanguage.com/ca/ca.html`

[4] `https://developer.apple.com`

[5] `https://blog.pusher.com/mvvm-ios/`

[6] `https://www.appcoda.com/mvvm-vs-mvc/`

[7] `https://www.orthanc-server.com/index.php`

[8] `https://www.dicomstandard.org/`