Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Automatizálási és Alkalmazott Informatikai Tanszék

Pattogató Bence

Személyes pénzügyi asszisztens alkalmazás fejlesztése

iOS és .Net platformokon

Konzulens

BUDAPEST, 2016

# Bevezetés

A diploma feladatom egy személyes pénzügyi asszisztens szoftverrendszerhez tartozó iOS és webkliens alkalmazás tervezése és fejlesztése. Az alkalmazás tőlem függetlenül más platformokra is implementálásra kerül, ezért nagyon körültekintően kellett megterveznünk minden olyan részét, ami a többi klienst is érinti.

A személyes pénzügyi asszisztens alkalmazás fejlesztésének ötlete saját és egy barátom példájához vezethető vissza. A felvetődő probléma minket érintett a mindennapokban és erre kínálok egy megoldást azoknak, akik szintén szemben találják magukat ezzel.

## A probléma

A probléma: az egyetem mellett aktívan dolgoztam és egyszer elkezdtem utána számolni, hogy mire is költöm el pontosan azt a pénzt amit megkeresek. Elég nehéz és kivitelezhetetlen feladatnak tűnt több hónapra visszamenőleg kielemezni, hogy mire mennyit költöttem, esetleg milyen költségeken tudnék faragni, hogy tudnám átláthatóbbá tenni a pénzügyeimet. Ekkor elkezdtem keresgélni, hogy mások milyen megoldásokat kínálnak a piacon.

Két alapvető ága van a PFM (Personal Financial Management) témakörnek. Az első megközelítés az utólagos elemzésre épít. A feldolgozás menete a következő: egy szoftverbe betápláljuk valamilyen úton a banki számlakivonatainkat (fejlett rendszerek esetén egy egyszerű megoldás, hogy megadjuk e-bankunk hozzáférési adatait, az belép oda és letölti a számára szükséges adatokat) azokat bonyolult és intelligens algoritmusokkal feldolgozza majd készít belőle különböző kimutatásokat. Mivel a készpénzt teljesen automatizáltan nem tudják számon tartani, egyik gyengesége ennek a megközelítésnek, hogy csak az elektronikusan mozgatott pénzeket lehet utólagosan bármilyen módszerrel feldolgozni. Illetve felvet sok biztonság-kritikai kérdése, hogy az adott szoftver fejlesztőjében mennyire bíznak meg a felhasználók, hajlandók-e megadni ilyen szenzitív adatokat.

A másik megközelítés a manuális módszert helyezi a középpontba, azaz a felhasználó felelőssége, hogy a tranzakciókat egyesével bevigye a rendszerbe. Ezt a megoldást már szinte minden létező platformra lefejlesztették, gondolok itt akár egy excel spreadsheet-re, weboldalra, Window/OSX alkalmazásra illetve, természetesen mobil kliensre.

Ennek a módszernek nyilvánvaló hátránya, hogy néminemű idő- és energiaráfordítás szükséges a felhasználótól, hogy minden egyes költését rögzítse a rendszerben. Ellenben, ha egy szoftver annyira kényelmesen és kézreállóan működik, hogy ez kiadásonként pár másodpercen belül elvégezhető, akkor azok a felhasználók, akik számára fontos, hogy teljesen pontos képet kapjanak a pénzügyeik mikéntjéről úgy gondolom, hogy hajlandóak ráfordítani ezt az időt.

Én az utóbbi megoldás kidolgozása mellett döntöttem. Egyik érv, hogy az első megközelítés az egy nagyon komplex rendszer lenne, a másik, hogy engem személyesen különösen érdekel a mobil szoftverfejlesztés, korábban is már foglalkoztam sokat ezzel a témával. Az alkalmazás tervezése és megvalósítása során mindig azt tartottam szem előtt, hogy a felhasználónak hogyan lesz a legkönnyebb, a leggyorsabb és a legegyszerűbb adminisztrálnia egy tranzakciót.

## Feladatkiírás és annak értelmezése

Szemelyes-penzugyi-asszisztens-alkalmazas-Feladatkiiras-1-2

A feladatkiírásból is látszik, hogy a rendszer megvalósítása az nem csak az én feladatom volt, hanem egy másik hallgatóval közösen kaptuk a fenti feladatot. A munka négy nagy részre osztható, ezek a következők:

* Szerveroldali alkalmazás
* iOS mobilalkalmazás
* Android mobilalkalmazás
* Front end webalkalmazás

A felsoroltak közül az én feladatom az iOS mobil-, és frontend webalkalmazás prototípusának megvalósítása, illetve az ezeket backend szoftver tervezése. Az iOS alkalmazáshoz tartozik egy azt kiegészítő, Apple Watch alkalmazás is, ami segíti a felhasználót, hogy még könnyebben és egyszerűbben használhassa a szolgáltatást.

## Előretekintés a dolgozatra

A dolgozat felépítése, a következő: .... ezt a végén egyszerűbb lesz megírni

# A feladat

## Motiváció

## Létező megoldások feltérképezése

## Funckionális specifikáció

Ez is majd később jön ide, van már doksi, amit csak át kell alakítani

# Piackutatás

## Előre tekintés

Feladatom volt, hogy ismerjem meg és mutassam be a témakörben a már létező megoldásokat. iOS alkalmazásokat kerestem az App Store-ban, a következő keresőszavakra:

* PFM,
* financial manager,
* budget tracker,
* budget manager,
* personal finance,
* expense tracker
* expense manager.

Az iOS alkalmazás boltban az ingyenes alkalmazásokat vizsgáltam, mivel az elkészítendő alkalmazás ingyenesen lesz publikálva a tervek szerint, ha arra kerül sor, ezért a fizetősök nem konkurensek egyelőre. Az alkalmazások mindegyikét letöltöttem, átfogóan áttekintettem, működését kipróbáltam és értékeltem az alábbi szempontok szerint. A mellettük szereplő szám azt mutatja, hogy milyen súllyal számítanak bele a végső eredménybe, amit lentebb értékelek majd, az értékkészlet 0 (legrosszabb) – 5 (legjobb):

* Mennyire intuitív az alkalmazás (3)
* Mennyire gyorsan lehet felvenni egy tranzakciót (3)
  + 20.000 Ft autó szervíz
  + 1500 Ft ebéd
* Mennyire átláthatók a múltbeli tranzakciók (2)
* Milyen az első benyomás 2-3 perc használat után (2)
* Mennyire szép az alkalmazás (2)
* Kötelező-e a regisztráció? (1) – véleményem szerint, ha kötelező a regisztráció az első belépésnél, az negatív felhasználói élmény. A pontozásnál úgy értékeltem, hogy mínusz pont, ha kötelező. Ha opcionális, akkor az ötletességet vettem figyelembe, hogy utólag figyelmezteti-e a felhasználót, könnyű-e elérni, stb.

## Megoldások bemutatása és értékelése

A következő listában felsorolom azokat az alkalmazásokat, amelyeket relevánsnak találtam. Ebben a listában röviden szövegesen értékelem és néhány képernyőképet mellékelek az alkalmazásokról, majd utána egy táblázatban összesítem az értékelési szempontok szerint adott pontokat.

### Daily Account Manager (Day book)[[1]](#footnote-1)

Az alkalmazás induláskor iCloud belépést kért, ott tárolja az adatokat. Felhasználói felülete átlagos, tab bar-ral működik a navigáció a képernyők között, mondhatni jól használhatónak. Mind bevételt mind kiadást lehet felvenni, a beviteli folyamat lehetne egyszerűbb. Összességében én nem tovább keresgélnék, ha szeretnék egy jó appot.

### Foreceipt[[2]](#footnote-2)

Az alkalmazás indulásakor Google account-tal kellett bejelentkezni, nem is lehetett más opciót választani, ami nem felhasználóbarát. Az alkalmazást számomra kicsit nehéz volt elkezdeni használni, nem láttam át könnyen. A kezelés szintén tab bar-ral működik, az volt a benyomásom, hogy nem annyira könnyen kezelhető, a felület platfromidegen néhány helyen és sok időbe telik egy tranzakciót felvenni.

### Tosh finance[[3]](#footnote-3)



Indulás után kötelező a bejelentkezés, amit Facebook/Google és hagyományos felhasználónév-jelszó párossal is elvégezhetünk. Egy kiadás felvétele elég egyszerű, gyorsan megvalósítható. Negatívumként annyit tudnék talán említeni erre vonatkozóan, hogy a szám billentyűzet túl színes és a beépítetthez képest fejjel lefelé van, ami elsőre furcsa volt. Az alkalmazás szép, könnyen kezelhető, extra benne, hogy felvehetünk külön számlákat és keretet, hogy még mennyi költhetünk az adott számláról, továbbá lehet különböző formátumokban exportálni.

### Money Mgr.[[4]](#footnote-4)

Felhasználó kezelés nincs az alkalmazásban, iCloud fiókba lehet backup-ot készíteni vagy e-mailben elküldeni magamnak, ezt nem tartom a legkönnyebb útnak. A design iOS 5-6-ra emlékeztet, a felhasználói élmény számomra eddig itt a legalacsonyabb. A tranzakció felvétel sebessége átlagos, lehetne egyszerűbb. Összességében a nagyon elmaradt design és kezelési nehézségek miatt nem tetszett az alkalmazás, nem használnám.

### Quick Money Recorder: Expense Budget Manager[[5]](#footnote-5)

Az alkalmazás viszonylag jó helyen szerepelt az App Store-ban legnagyobb meglepetésemre, mert megnyitva az alkalmazást egy nagyon nem felhasználóbarát, design-ban körülbelül 5 évvel ezelőtti irányelvekhez igazodó képernyő fogadott. Az ikonok, a színek, a gombok formája és elhelyezkedése sem stimmelt, felvenni tranzakciót beletelt pár percbe mire sikerült. Azért is mutatom itt be, mert ez egy jó példa arra, hogy milyen egy alkalmazás, ami a felhasználói igényeket egyáltalán nem elégíti ki. A felhasználó-, és adatkezelés prémium funkció, amit láttam belőle, az az, hogy van több opció is, amik között szerepel Dropbox, Google Drive és iCloud is.

### Buxfer[[6]](#footnote-6)



Induláskor felajánlja az alkalmazás, hogy hozzak létre felhasználói fiókot, akár közösségi profil alapján, de van egy próba lehetőség, hogy beletekintsek az appba, ez nagyon szimpatikus ötlet. Így egy előre adatokkal felöltött képernyőt látok, ahol ki is tudom próbálni a funkciókat, hasznos.

Az kezdeti jó élmény után a tranzakció felvétele sajnos nem hozta a vártakat. A design itt már nekem túl egyszerű volt, ennek ellenére a felvétel kicsit hosszas és további negatívum, hogy az adatokat nem cache-elik, ebből kifolyólag a gombokra tapintva többször is várni kellett, míg letölti azokat, ami nagyon csökkenti a felhasználói élményt.

Bal oldalról elérhető egy menü, amiben nagyon sok opció található, talán egy kicsit túl sok is ha a fenti specifikációt lefedő alkalmazáshoz hasonlítom. Összességében az alkalmazás közepes a többihez képest, kicsit túl bonyolultnak érzem és szerintem több funkciót tettek bele mint amennyire szüksége van egy felhasználónak.

### TapToTrack[[7]](#footnote-7)

Az alkalmazás rögtön a kiadás összesítő nézeten fogad, ahonnan egy-egy, a navigációs sávban elhelyezkedő gombbal lehet átváltani a történeti/chart nézetre illetve a beállításokhoz menni. A tranzakció felvétele kényelmes és gyors, egy apróság kivételével, a leírás mező tapinthatóságát illetően. Összességében letisztult, egyszerű és könnyen használható a felvétel része.

A múltbeli tranzakciók megtekintése véleményem szerint átlagos, a többi alkalmazáshoz képest. Felhasználókezelés tekintetében csak a saját, felhasználónév/jelszavas változatot támogatják.

### Spender[[8]](#footnote-8)

Az alkalmazás nagyon letisztult, modern design-nal rendelkezik. Rögtön a tranzakció rögzítő képernyő fogad induláskor, a kategória kiválasztása is egyszerű és konfigurálható. Az időpont beállítása is kézreáll. Van prémium szolgáltatás, ami extra funkciókat engedélyez, de ezekre nem ereztem szükséget, e nélkül is jól használható. Minimális negatívum, hogy a már felvett tranzakciók visszanézése nekem nem volt elsőre könnyen értelmezhető.

Összességében nagyon egyszerű, letisztult és jól használható alkalmazás, ezt találtam mind közül a legjobban használhatónak.

# Felhasznált technológiák

## Kliens

### iOS

Az iOS alkalmazás fejlesztését választottam feladatomul, több ok miatt is. Három éve foglalkozom iOS szoftverfejlesztéssel, előtte egy évig Android mobil alkalmazásokkal foglalkoztam szabadidőmben / egyetemen. Személyes véleményem, hogy az Apple által képviselt szoftverfejlesztési elvek jóval letisztultabbá, könnyebben átláthatóvá teszik a munkát az iOS rendszerrel, mint legnagyobb vetélytársánál.

### iOS történeti hátter

innen: <https://en.wikipedia.org/wiki/IOS>

## Az alkalmazás felépítése

Az iOS mobil alkalmazás tervezésénél figyelembe vettem az Apple által leírt kezelői felület irányelveket. Ezek vonatkoznak gomb méretektől elkezdve színeken át, betűméretekre, helykihasználásra és sok minden másra, ami mind azt a célt szolgálja, hogy a felhasználók számukra a legkényelmesebb és legkézenfekvőbb módon érjék el a funkciókat.

Az alkalmazásban három fő nézet van, az új kiadás, a kimutatások és a beállítások. A fő nézetek egymás mellett helyezkednek el egy UIPageViewController-ben, amik között akár swype gesztúrával, akár gombnyomással is válthat a felhasználó. Az alkalmazás a középső nézettel indul, hogy a leggyakoribb use-case induláskor rögtön elérhető legyen, az adatbevitel. A fő nézet helyezkedik el középen, tőle balra a kimutatások és jobbra pedig a beállítások képernyő. További három képernyő érhető el az alkalmazásban, ezek a következők:

• Diagram részletes nézet – kimutatások nézetből

• Kategóriák – új kiadások nézetből

• Korábbi kiadások – új kiadás felvétele nézet

# Programozási minták

Az architektúrális tervezési minták használata a programozásban elengedhetetlen. Nagyon sok ilyen mintát használ egy mai szoftverfejlesztő, akár indirekt is. Egy tervezési minta tulajdonképpen egy tervezési sablon, ami egy már ismert, általános probléma megoldására szolgál egy bizonyos kontextusban. Ezt ahhoz tudnám hasonlítani, hogy amikor elkezdenek tervezni egy új autót, akkor az elmúlt sok évtized tapasztalata alapján tudják már, hogy kell gurulnia – erre való a kerék, kell gyorsulnia erre való a belső égésű motor, kell fordulnia erre való a kormánymű, stb. Tehát az autó tervezésnél, ezeket mind alapul veszik és a már kiforrott „váz” egyes elemeit fejlesztik tovább, nem pedig nekiállnak előről.

A programozási minták használata nem csak könnyebbé teszik a szoftverfejlesztők feladatát, hanem egyfajta útmutatás is adnak, hogy egy ismert problémát miként tudunk a legkönnyebben és leghatékonyabban megoldani. Mobil szoftverfejlesztés témakörre korlátozódva kijelenthető, hogy egy jó architektúra a következő előnyökkel ruházza fel a rendszert:

* jól elosztott felelősségek az entitások között, szigorú szabályokkal,
* könnyű és teljes körű tesztelhetőség,
* könnyű és tiszta fejlesztési folyamat valamint jó karbantarthatóság.

## MVP

iOS fejlesztés témakörben egy ismert és sokat vitatott problémakör, hogy az alábbi három feladat feladat kezeléséért milyen entitások felelősek :

* a tárolt adatokhoz való hozzáférés
* a felhasználói felület létrehozása
* az adatok összekötése a felhasználói felülettel
  + adatok megjelenítése a felületen
  + a felületen generált / módosított adatok eljuttatása a tároló réteghez

Az az alapvető megoldás, hogy egy entitás felelőssége legyen megoldani mind a három problémát azt okozná, hogy a kód elburjánzana, nehezen karbantartható és újrafelhasználható lenne. Emellett sok fejtörést okozna a hibakeresés az összeolvadt és el nem különített felelősségű entitásokban. Egyértelmű, hogy a felelősségeket szét kell választani, erre a kérdésre kerestem én a lehető legjobb választ.

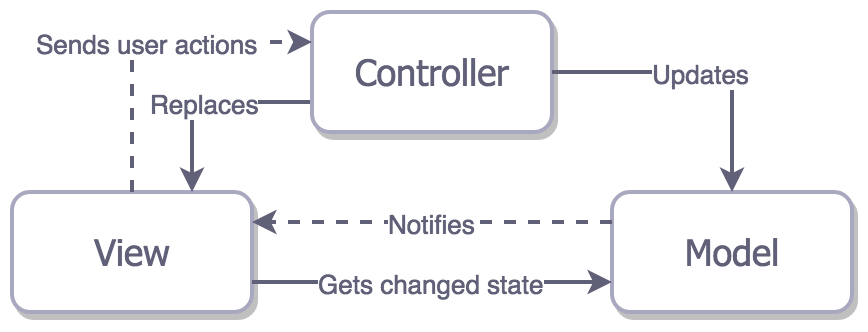
### Az Apple megoldása - MVC

Az Apple javasol a fentebb felvetett problémára egy megoldást, az MVC (Model View Controller) egy általuk módosított változatát.

#### Az eredeti MVC

Az MVC architektúrális minta a rendszer entitásait és fentebb felsorolt felelősségeket három csoportba osztja:

* Model: az adatok eléréséért és kezeléséért felel
* View: a megjelenítési réteg kezelése a felelőssége
* Controller: mediátor a View és a Model között, rajta keresztül éri el a View a Model által kezelt adatokat és az ő felelőssége frissíteni a Model-t ha a View manipulálja az adatokat és fordítva, frissíteni a View-t, ha a Model-ben tárolt adat változott.

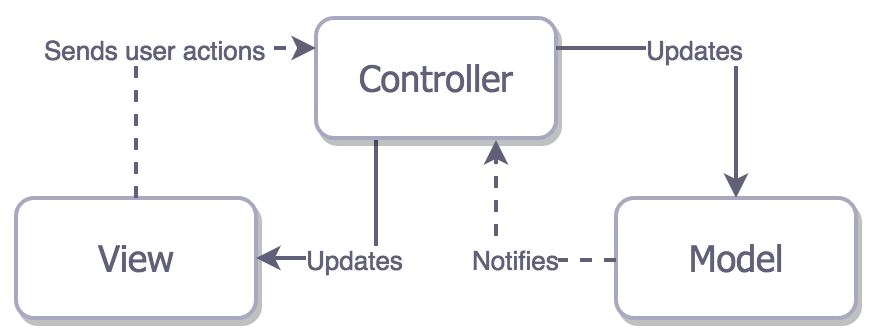


Ebben a mintában a View igazából állapotmentes, tulajdonképpen a Controller létrehozza amikor a Model megváltozott. Ez azt eredményezi, hogy mind a három entitásnak ismernie kell a másik kettőt, ami nagyon nehézzé teszi az újrafelhasználhatóságot fejlesztéskor. Ezért hozta létre az Apple ennek a mintának egy általuk módosított, ebben a környezetben jobban használható változatát.

#### Az Apple féle MVC – teoretikusan

Az Apple által módosított változatban az entitások ugyanazok maradnak, a közöttük történő kommunikáció változik.

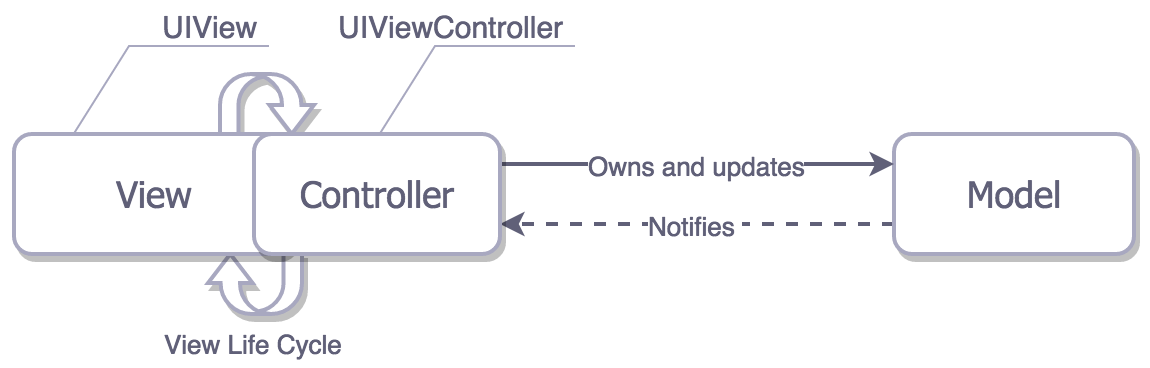
A Controller egy mediátorként viselkedik a View és a Model között, úgy, hogy azoknak ne kelljen ismerniük egymást. Itt a View és a Model újrafelhasználható marad, csak a Controller-lesz az az entitás amit az adott üzleti logika szerint kell módosítanunk és illeszteni a View és Model között. Ez egy elfogadható kompromisszum.



Elméletben ez egy teljesen jól használható és kigondolt minta. A valóság azonban egy kicsit más, amikor valós iOS szoftverfejlesztése és az Apple-féle MVC használatára kerül sor.

#### Az Apple-féle MVC – a gyakorlatban

A Cocoa MVC rövidítést sokan tréfásan Massive View Controller-ként oldják fel, ez nem véletlen. Az Apple-féle MVC arra ösztönözi a fejlesztőt, hogy a Model-t leválasztva a View és Controller entitásokat összemossa a ViewController-ekben azzal, hogy a View-k életciklusa nagyon nagy mértékben van belevonva a ViewControllerbe.



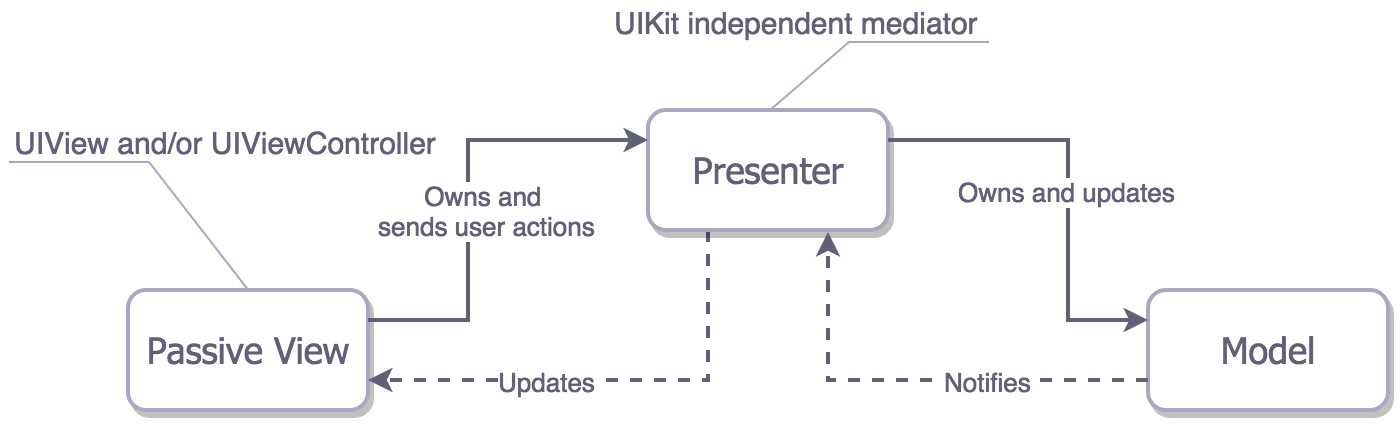
A View-k összes feladata a gyakorlatban kis túlzással arra korlátozódik, hogy eseményeket továbbítsanak a Controller-jük felé. A Controller feladat lesz, hogy a View-t adattal töltse fel, kezelje az eseményeket, hálózati hívásokat menedzseljen, egyszóval majdnem minden.

Az igazi probléma akkor jelentkezik, amikor tesztelésre kerül sor és ott lyukadunk ki, hogy csak a Model-t tudjuk igazából tesztelni, mert az üzleti logika teszteléséhez a View és a Controller összeolvadása miatt szükségünk van a UI elemek renderelésére, az életciklusuk követésére, stb.

Összefoglalva, az Apple által továbbfejlesztett MVC a valóságban elég nehezen tesztelhető kódot eredményez és a felelősségek sem tisztázottak, sok helyen megszegik az elméletben kiosztott felelősségeket, ámbár gyorsan tanulható és kisebb alkalmazásoknál az implementációt meg tudja gyorsítani.

Azonban ha nagyobb volumenű alkalmazást tervezünk, amiben fontos a tesztelhetőség, a karbantarthatóság és a felelősségek elkülönítése, akkor egy másik architektúrális mintára van szükség, mi az MVP mellett döntöttünk.

### MVP – Model View Presenter



Az MVP architektúra a fenti ábrán látható. Nagyon hasonlít arra a mintára, amit az Apple MVC-nek nevez és fentebb olvashattak róla. Viszont van egy nagyon fontos különbség a kettő között, míg az előbbinél a View és a Controller életciklusa és sok más tulajdonsága szinte összeolvadt, itt a Presenter entitás semmit nem tud a View életciklusáról és annak semmilyen működéséről, csakis és kizárólag azért felel, hogy a View-t frissítse a megfelelő Model-ből jövő adatokkal.

A fenti mintával szemben, itt a UIViewControllerek a View szerepét fogják betölteni és nem pedig a Controller-ét, ami magába olvassza a View-t is. Ez a része az az MVP-nek, ami a nagyon jól tesztelhetőséget biztosítja, mivel minden PassiveView – Presenter – Model esemény- / adatáramlás interfészeken keresztül történik, így bármelyik könnyedén lecserélhető (mock-olható). Ennek az ára a fejlesztési idő (kezdetben) jelentős növekedése, mivel mind az adat, mind az események kötését nekünk kell megírnunk, úgy, hogy – mint írtam – minden entitás interfészeken keresztül kommunikáljon a szomszédos rétegekkel.

Egy példán keresztül szemléltetem, hogy hogyan is lehet MVP-t iOS-en, Swift-ben implementálni:

A megjelenítendő példa objektum:

struct Message { // Model

let text: String

}

View entitás interfésze:

protocol MessageView: class {

func setMessage(message: String)

}

Presenter entitás interfésze:

protocol MessageViewPresenter {

init(view: MessageView, message: Message)

func showMessage()

}

Egy Presenter példány, ami megvalósítja a Presenter interfészt:

class MessagePresenter : MessageViewPresenter {

unowned let view: MessageView

let message: Message

required init(view: MessageView, message: Message) {

self.view = view

self.message = message

}

func showMessage() {

let message = "Your message is:" + self.message.text

self.view.setMessage(message)

}

}

A UIViewController, ami az MVP szerinti View entitás, megvalosítja a fent leírt View interfészt:

class MessageViewController : UIViewController, MessageView {

var presenter: MessageViewPresenter!

let showMessageButton = UIButton()

let messageLabel = UILabel()

override func viewDidLoad() {

super.viewDidLoad()

self.showMessageButton.addTarget(self, action: #selector(MessageViewController.didTapButton(\_:)), forControlEvents: .TouchUpInside)

}

func didTapButton(button: UIButton) {

self.presenter.showMessage()

}

func setMessage(message: String) {

self.messageLabel.text = message

}

}

A fent definiált osztályok használatára egy példa:

let model = Person(firstName: "Bence", lastName: "Pattogato")

let view = GreetingViewController()

let presenter = GreetingPresenter(view: view, person: model)

view.presenter = presenter

Egy nyitott kérdés marad az MVP-vel kapcsolatban, mégpedig, hogy kinek felelőssége lesz létrehozni és összerakni a három egymástól teljesen elszeparált entitást. Erre egy bevett módszer, hogy létrehozunk egy alkalmazás szintű, statikus Router osztályt, ami az entitás-hármasok összerakásáért és a nézetről-nézetre való navigálásért fog felelni.

Összességében elmondható a leírt MVP-ről, hogy megfelel azoknak az elvárásoknak amiket az elején támasztottunk egy jó architektúrával szemben. Nagyon könnyen és teljeskörűen tesztelhetőköszönhetően annak, hogy az üzleti logika teljesen le van választva a megjelenítési rétegről. Továbbá a későbbi karbantarthatóságot és rugalmasságot azzal biztosítja, hogy mivel el vannak választva egymástól az entitások - csak interfészeken keresztül ismerik a másikat - így egy konkrét implementációja az adott Model/View/Presenter interfésznek teljes szabadságot ad a fejlesztőnek a változtathatóságban anélkül, hogy a többi entitáson változtatni kellene.

## Singleton

A singleton minta annak a problémának a megoldására szolgál, amikor egy objektumról biztosítani akarjuk, hogy pontosan egy példány fog létezni belőle. A minta használatával elővigyázatosnak kell lenni, mivel saját magáról tárol egy referenciát, így soha nem fog tudni felszabadulni, ezért alaposan megfontolandó, hogy mikor van szükség a használatára. Az alkalmazásunkban az ApplicationRouter osztály singleton, mivel ebből pontosan egy példánynak szabad léteznie. Ez az osztály felel az alkalmazás képernyői közötti navigációért és az MVP minta szerinti entitás hármasok összállításáért.

class ApplicationRouter {

static let sharedInstance = ApplicationRouter()

private init() {

// Egyszeri inicializáló kód

}

}

# Tesztelés

A tesztelés fontossága a szoftverfejlesztésben megkérdőjelezhetetlen. Azt az alapvetést, hogy ami kódot megírunk, az mindig elsőre tökéletesen és hibamentesen fog futni, úgy gondolom bárki aki fejlesztett már szoftvert életében, elveti. Az is tesztelésnek számít már, hogyha lefuttatja a fejlesztő azt a kódot amit megírt és így kipróbálja, hogy megfelelően működik-e.

## Automatizált tesztelés

Ennél vannak ma már sokkal szofisztikáltabb és hatékonyabb módszerek is. Az automatizált tesztelés nagy előnye, nem szükséges emberi beavatkozás, rábízhatjuk egy szoftverre, hogy tesztelje le helyettünk az kérdéses programkódot. Emiatt persze az igénybe vett idő is töredékére csökken, mint a kézi tesztelésnél, amellett, hogy kizárja a tesztelés közbeni hibavétés esélyét. Automatizált tesztek futtatásával tudja a fejlesztő bizonyítani, hogy a megírt kód megfelelően működik, illetve az inkrementális fejlesztés során nem rontott el semmilyen olyan kódrészletet, ami már korábban helyesen működött.

## Tesztelés támogatása XCode-ban

Az XCode alapvetően három különböző tesztelési metodikát támogat.

* Funkcionális teszt, az üzleti logika és a kód funkcionális működésének ellenőrzésére,
* Performancia teszt, a végrehajtási idő mérésére
* Felhasználói felület teszt, a megjelenő felületi elemek és azok közötti interakciók helyességének ellenőrzésére.

A funkcionális és performancia tesztek tulajdonképpen függvények egy-egy működés ellenőrzésére. Minden teszteset felállít egy tesztkörnyezetet (setUp), végrehajtja a megírt teszteket majd felszabadítja a környezetet. A felhasználói feleület tesztek felvételek az alkalmazásból, amiből

A funkcionális tesztelés legelterjedtebb módja a Unit tesztelés. Egy kód unit az a legkisebb tesztelhető kód-egység, ami magában megvalósít egy logikailag ellenőrizhető funkciót. Leggyakoribb használati esetük (mint már említettem), hogy egy időpillanatban az összes addigi funkcióra vannak már írva unit tesztek, mindegyik sikeresen lefut és amikor ezután kódot írunk, akkor ellenőrizzük, hogy továbbra is minden unit teszt sikeres-e, azaz új kód írásával nem rontottunk-e el egy már meglévő működő részt. A tesztesetek eredményét egy előre definiálttal hasonlítja össze, ez dönti el, hogy az sikeres volt-e vagy sem.

## Az XCTest keretrendszer

Az XCode tesztelési keretrendszere a XCTest. Amikor létrehoz a fejlesztő egy új XCode projektet, az rögtön generál hozzá egy teszt targetet is, amit külön tudunk majd futtatni tesztelési célból, így nem kell az alkalmazás minden részét fordítani / futtatni a tesztek ellenőrzéséhez.

Egy teszteset létrehozásához egy osztályt kell létrehozni, ami az XCTestCase ősosztályból származik. Ebben az osztályban minden olyan függvény amit a „test” előtaggal kezdődik, le lesz futtatva és kiértékelve egy unit teszt-ként. Ebben a leszármazott osztályban továbbá kötelezően felül kell írni a következő két metódust, ami minden egyes teszt metódus előtt és után meg fog hívódni.

* setUp() – Ez a metódus hívódik minden „test” előtaggal keződő függvény előtt. Itt kell beállítani a tesztkörnyezetet amennyiben szükség van rá (változókat inicializálni, memória adatbázist létrehozni, stb.).
* tearDown() – Ez a metódus hívódik minden teszt metódus után. Ebben az ellenkezőjét kell elvégezni, felszabadítani azt a tesztkörnyezetet amit a setUp függvényben létrehozunk, illetve a futtatott teszt hatását kitörölni, hogy a következő futására ne legyen hatással.

A tesztek futásának ellenőrzésére az XCTAssert metódus családot kínája a XCTest keretrendszer. Lényege minden metódusnak ezek közül ugyanaz, állítunk egy logikailag igazra vagy hamisra kiértékelhető feltételt, aminek teljesülésétől függ a teszteset sikeressége. A teljesség igénye nélkül néhány példa:

* Sikeres a teszt, ha a (String) name változó nem üres

XCTAssert(!name.isEmpty)

* Sikeres a teszt, ha a person objektum létezik (nem nil)

XCTAssertNotNil(person)

* Ellenőrizhetjük két objektum egyenlőségét (ilyenkor a tömbben tárolt elemeknek meg kell valósítania az Equatable protocot-t, amivel definiálhatjuk, hogy mikor egyenlő két elem) A példában a Person osztály két példánya akkor egyenlő, ha megegyezik az idNumber tulajdonságuk. Mindkét assertation sikeresnek értékelődik ki.

struct Person: Equatable {

var name: String

var motherName: String

var idNumber: String

var age: Int

}

func ==(lhs: Person, rhs: Person) -> Bool {

return lhs.idNumber == rhs.idNumber

}

let steve = Person(name: "Steve", motherName: "Mom", idNumber: "abc123", age: 16)

let patrick = Person(name: "Patrick", motherName: "Mother", idNumber: "abc456", age: 23)

let oldSteve = Person(name: "Old Steve", motherName: "Mom", idNumber: "abc123", age: 46)

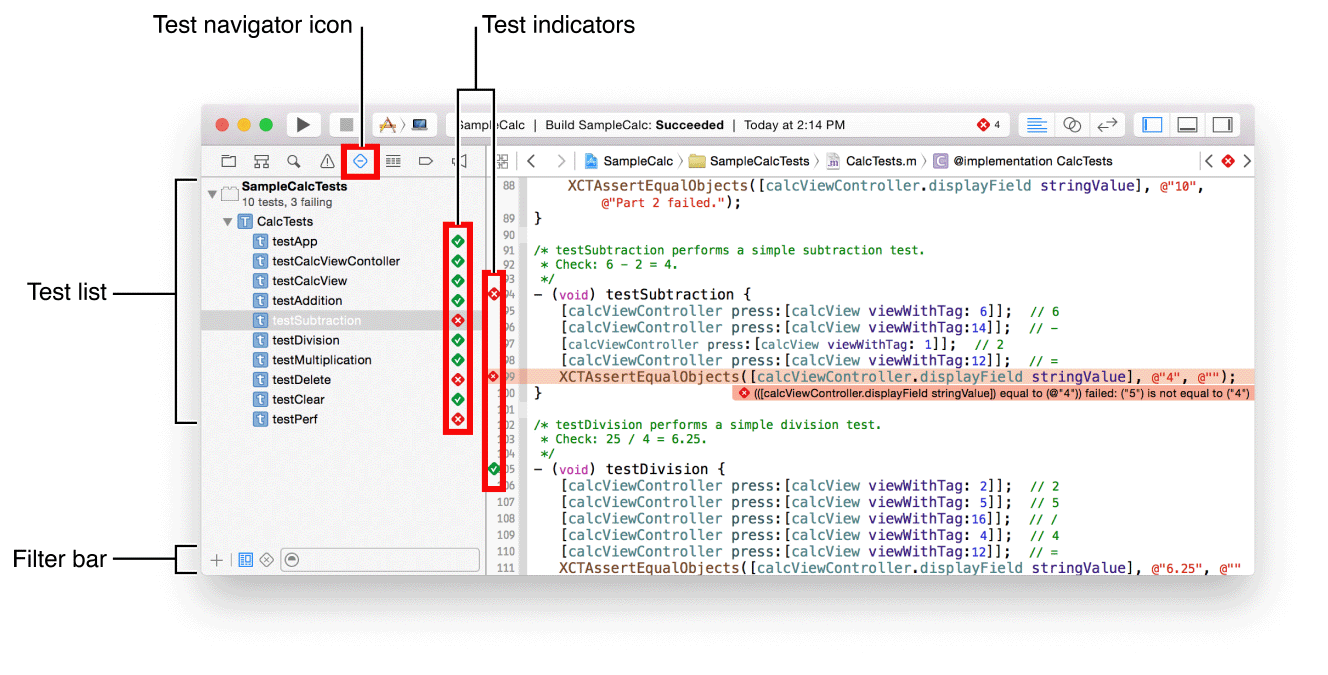
// Sikeres lefutás

XCTAssertNotEqual(steve, patrick)

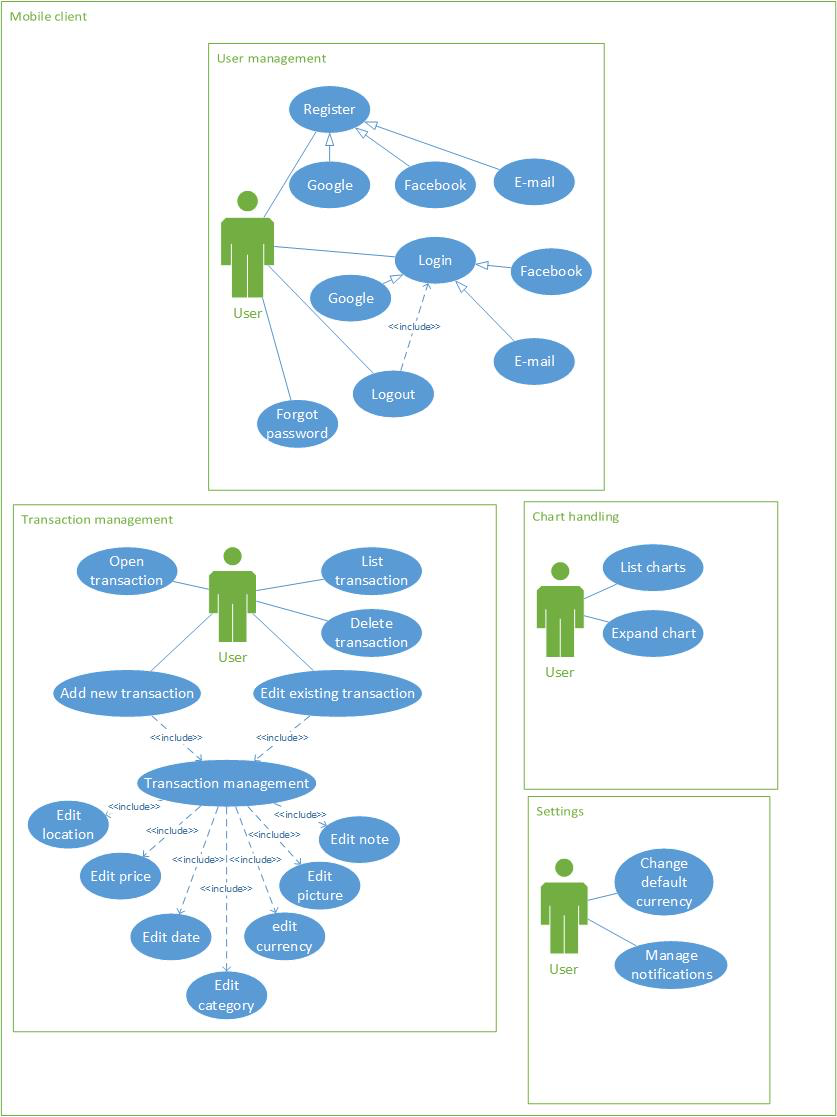
// Sikeres lefutás

XCTAssertEqual(steve, oldSteve)

A tesztelési felület a következő ábrán látható:



A zölddel jelölt tesztek sikeresen futottak le, míg a pirossal jelöltek sikertelenül. Egyesével is elindíthatók a tesztek, ha a zöld/piros ikonra kattint a fejlesztő.

Függelék

Irodalomjegyzék

1. iOS Architecture patterns: <https://medium.com/ios-os-x-development/ios-architecture-patterns-ecba4c38de52#.mkod5pl0e>
2. https://developer.apple.com/library/ios/documentation/ToolsLanguages/Conceptual/Xcode\_Overview/UnitTesting.html

1. https://itunes.apple.com/hu/app/daily-account-manager-income/id1168406615?mt=8 [↑](#footnote-ref-1)
2. https://itunes.apple.com/hu/app/foreceipt-receipt-manager/id721048897?mt=8 [↑](#footnote-ref-2)
3. https://itunes.apple.com/hu/app/toshl-finance-expense-tracker/id921590251?mt=8 [↑](#footnote-ref-3)
4. https://itunes.apple.com/hu/app/money-manager-pro-financial/id465646699?mt=8 [↑](#footnote-ref-4)
5. https://itunes.apple.com/hu/app/quick-money-recorder-expense/id554067740?mt=8 [↑](#footnote-ref-5)
6. https://itunes.apple.com/hu/app/buxfer-expense-tracker-budget/id312260980?mt=8 [↑](#footnote-ref-6)
7. https://itunes.apple.com/hu/app/live-expenses-expense-tracker/id486195690?mt=8 [↑](#footnote-ref-7)
8. https://itunes.apple.com/hu/app/spender-money-management/id637562972?mt=8 [↑](#footnote-ref-8)