

Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Dálnoky Bertalan András

Feladat menedzsment alkalmazás fejlesztése iOS platformra

Konzulens

Dr. Blázovics László

BUDAPEST, 2024

Tartalomjegyzék

[Összefoglaló 5](#_Toc181814889)

[Abstract 6](#_Toc181814890)

[1 Bevezetés 7](#_Toc181814891)

[1.1 Motiváció 7](#_Toc181814892)

[1.2 A dolgozat felépítése 9](#_Toc181814893)

[2 A feladat részletes értelmezése 10](#_Toc181814894)

[2.1 Technológiai elvárások 10](#_Toc181814895)

[2.2 Funkcionális elvárások 11](#_Toc181814896)

[3 Felhasznált technológiák 12](#_Toc181814897)

[3.1 SwiftUI 12](#_Toc181814898)

[3.2 Combine 13](#_Toc181814899)

[3.3 iCloud 13](#_Toc181814900)

[3.4 CoreData 14](#_Toc181814901)

[3.5 WidgetKit 15](#_Toc181814902)

[3.6 SwiftLint 16](#_Toc181814903)

[3.7 SwiftGen 17](#_Toc181814904)

[3.8 Resolver 18](#_Toc181814905)

[3.9 Sourcery 19](#_Toc181814906)

[3.10 CocoaPods 19](#_Toc181814907)

[4 Tervezés 20](#_Toc181814908)

[4.1 Architektúra, rétegek 20](#_Toc181814909)

[4.2 Felhasználói felület 24](#_Toc181814910)

[4.2.1 Wireframe készítése 24](#_Toc181814911)

[4.2.2 Design készítése 26](#_Toc181814912)

[5 Megvalósítás 29](#_Toc181814913)

[5.1 Model réteg 29](#_Toc181814914)

[5.2 Service réteg 33](#_Toc181814915)

[5.2.1 SwiftData 33](#_Toc181814916)

[5.2.2 CoreData 34](#_Toc181814917)

[5.3 Navigációs réteg 37](#_Toc181814918)

[5.4 ViewModel réteg 37](#_Toc181814919)

[5.5 View réteg 37](#_Toc181814920)

[5.5.1 iPhone alkalmazás 37](#_Toc181814921)

[5.5.2 iPad alkalmazás 37](#_Toc181814922)

[5.5.3 macOS alkalmazás 37](#_Toc181814923)

[5.5.4 watchOS alkalmazás 37](#_Toc181814924)

[5.6 Widget komponensek 37](#_Toc181814925)

[6 Tesztelés 38](#_Toc181814926)

[6.1 Egység tesztelés 38](#_Toc181814927)

[6.1.1 Utility komponensek tesztelése 40](#_Toc181814928)

[6.1.2 ViewModel réteg tesztelése 41](#_Toc181814929)

[6.2 Felület tesztelés 43](#_Toc181814930)

[6.3 Tesztelési eredmények 44](#_Toc181814931)

[7 Összefoglalás és továbbfejlesztési lehetőségek 48](#_Toc181814932)

[8 Köszönetnyilvánítás 49](#_Toc181814933)

[9 Irodalomjegyzék 50](#_Toc181814934)

Hallgatói nyilatkozat

Alulírott **Dálnoky Bertalan András**, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző, cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2024. 05. 16

...…………………………………………….

Dálnoky Bertalan András

Összefoglaló

A mai rohanó világban, az embereknek egyre több dolgot kell észben tartania nap mint nap. Gyakran előfordul, hogy megfeledkezünk egy olyan dologról, amit meg kellene csinálnunk viszont a sok egyéb tenni való miatt kimegy a fejünkből. Emiatt sokan használnak olyan segédeszközöket, amik segítenek az elvégzendő feladataink számon tartásában például határidő naplókat, táblázatokat, emlékeztetőket, vagy feladat kezelő alkalmazásokat. Ezek az eszközök azonban sokszor nagyon egysíkúak, nehéz módosítani vagy személyre szabni őket, valamint, ha megszokjuk a használatukat, a hátrányaik sok esetben vezethetnek a hatékonyságunk romlásához. Mivel a technológia, már teljes mértékben része a mindennapjainknak, ezeknél az eszközöknél hatékonyabb lehetne egy olyan alkalmazás, ami összefogja az említett eszközök legjobb aspektusait, és kijavítja a legnagyobb hibáikat is.

A diploma munkámban egy olyan többrétegű, az Apple ökoszisztémájában elhelyezkedő multiplatform alkalmazás tervezését és megvalósítását részletezem, amelynek célja, hogy egy ilyen platformot teremtsen. Az alkalmazás ezt, a projektmenedzsmentben gyakran használt Kanban tábla és egy hagyományos naptár ötvözésével kísérli meg, a személyre szabhatóságot is szem előtt tartva.

Az fejlesztés során kiemelt szerepet kapott az, hogy az Apple ökoszisztéma által nyújtott lehetőségéket kihasználva és a lehető legmodernebb technológiák felhasználásával készüljön el az alkalmazás. Végeredményül egy modern szemléletű, bővíthető szoftver készült el, aminek fejlesztése során rengeteg új dolgot sikerült elsajátítanom, illetve lehetőségem volt kipróbálni számos új, vagy általam korábban még nem használt technológiát.

Abstract

In our current fast-paced world, people have more and more things to remember every day. We often forget something we need to do because it slips our minds amidst all the other tasks. As a result, many people use tools to help keep track of their tasks, such as planners, spreadsheets, reminders, or task management applications. However, these tools are often very one-dimensional, difficult to modify or customize, and if we get used to using them, these drawbacks can often lead to a decrease in our efficiency. Since technology is an integral part of our everyday lives, a more effective solution could be an application that combines the best aspects of these tools while also addressing their major flaws.

The focus of this diploma is the design and implementation of a multi-layered, multiplatform application within the Apple ecosystem, capable of creating such a platform. The application attempts this by combining a tool often used in project management, the Kanban board, with the more traditional calendar, while also focusing on customizability.

During the creation of this application, a high emphasis was placed on utilizing the opportunities provided by the Apple ecosystem and incorporating modern technologies as much as possible. The result is a modern, scalable software, and throughout its development, I acquired several new skills and had the opportunity to try out many new technologies, including some that I was previously unfamiliar with.

# Bevezetés

Napjainkban az emberek jelentős része nem tudná elvégezni mindennapi feladatait a telefonja, illetve az azon található különböző szoftverek nélkül. Ez nem is olyan meglepő tény, hiszen a mindennapos technológiai újítások mellett már szinte minden hétköznapi probléma megoldásához létezik egy applikáció.

Nagyon sok olyan alkalmazás létezik már ami egy specifikus feladat elvégzése helyett azt tűzi ki célul, hogy a felhasználók feladatvégzésének, illetve idő beosztásának hatékonyságát növelje. Azonban, ha egy ilyen alkalmazásban hiányoznak bizonyos funkciók, vagy nem tudjuk elegendően személyre szabni a funkciókat, akkor az alkalmazás hatása hatékonyságra elhanyagolható, vagy akár negatív is lehet.

Ezek az alkalmazások azonban általában két nézőpont egyikéből közelítik meg a céljukat. Az egyik megközelítésben a felhasználók egy szimpla tennivalókból álló listát tudnak fenntartani. Ezzel szemben a másik megközelítés inkább a projektmenedzsmentben elterjedt módszereket részesíti előnyben. Mindkét nézőpontnak megvannak a maga előnyei és hátrányai, azonban, ha találnánk a két megközelítés között egy arany középutat, akkor az előnyök még jelentősebbek lehetnének.

## Motiváció

Egyetemi tanulmányaim előrehaladtával egyre biztosabb lettem abban, hogy a legnagyobb akadályt a diákok életében nem az egyes tantárgyak okozta szakmai kihívások jelentik, hanem a szakmai kihívásokkal járó feladatok időbeli beosztása. Ez különösen igaz a mester szakot végző hallgatókra, akik az egyetem mellet akár munkát is vállalnak, szociális életet élnek. Az esetleges gyakori sportolásról vagy egyéb szabadidős tevékenységről és hobbikról pedig még csak szó sem esett. Egy jól átgondolt időbeosztás követése gyakran kevesebb stressz mellett ugyanolyan akadémiai eredményeket tud produkálni, mint a számonkérések előtti napokon keresztül tartó, véget nem érő tanulás.

Már évek óta használok olyan eszközöket, amik segítenek létrehozni, és betartani egy időbeosztást, hogy minden tevékenységemre tudjam megfelelő mennyiségű időt szánni. Azonban minden általam használt eszközből hiányzott valamilyen funkció, ami miatt egy másik eszközzel párhuzamosan voltam kénytelen használni.

Véleményem szerint az egyik leghasznosabb ilyen eszköz a Kanban tábla, aminek funkcionalitását rengeteg alkalmazás nyújtja. Ezekkel azonban az a probléma, hogy az elvégzendő feladatok állnak a középpontban, és az időbeli beosztásra minimális lehetőségünk van.

Egy másik nagyon hasznos eszköz a szimpla határidő napló, vagy a telefonokon megtalálható naptár alkalmazás. Ebben ugyan az időbeli beosztás van a középpontban azonban legtöbb esetben, nehéz egy átfogó képet kapni arról, hogy milyen feladatokat, vagy tennivalókat kell elvégezni az adott napon. Ezen felül ezeket az eszközöket meglehetősen limitáltan lehet személyre szabni, ami nagy mértékben csökkenti a használhatóságukat.

Mivel az említett megoldásokat sok ideig használtam párhuzamosan, egy idő után felmerült bennem az ötlet, hogy milyen hatékony lehetne a két eszköz által nyújtott funkcionalitást egyesítenénk. A jelen dolgozat témáját nyújtó alkalmazás elkészítéséhez az inspirációt az adta, hogy egy ilyen egyesített funkcionalitást nyújtó eszköz után való kutatásom nem járt sikerrel.

Az alkalmazás által támogatott rendszerek kiválasztásakor, több szempont játszott szerepet. Az elsődleges szempontot a személyes tapasztalat és preferencia töltötte be. Mind a platformra való fejlesztői, mind felhasználói szempontból az Apple rendszereiben van a legtöbb tapasztalatom.

Ezen felül a korábban említett eszközökkel való tapasztalatom alapján, éltem azzal a feltételezéssel, hogy a felhasználók több eszközön is használnák az alkalmazást. Az Apple ökoszisztémája lehetővé teszi a különböző platformokat célzó alkalmazások közötti közös kód megosztását, valamint rengeteg népszerű platform részese az ökoszisztémának. Ezekből az okokból kifolyólag az ideális választás az Apple által nyújtott rendszerek voltak.

## A dolgozat felépítése

A dolgozat további fejezeteiben ennek az alkalmazásnak a tervezését és fejlesztését részletezem. A következő szakaszban pontosítom a feladatkiírás részleteit, és kifejtem a megvalósítandó funkciókat. Ezután bemutatom legfontosabb felhasznált technológiákat. A **Tervezés** fejezetben szó lesz a felhasznált architektúráról és a felhasználói felület tervezésének lépéseiről. Ezt követően a **Megvalósítás** fejezetben mutatom be az alkalmazás megvalósításának részleteit, előtérbe helyezve az érdekesebb felmerülő problémákat és megoldásukat, valamint a lényegesebb fejlesztői döntéseket. A **Tesztelés** fejezetben részletezem a szoftver tesztelésénél felmerült akadályokat és a tesztelés eredményeit. Végül összefoglalom az alkalmazás készítése során szerzett tapasztalataimat és az esetleges továbbfejlesztési lehetőségeket.

# A feladat részletes értelmezése

A feladat tehát egy olyan multiplatform alkalmazás megalkotása, ami lehetővé teszi a felhasználóknak a feladataik vagy tennivalóik szervezését és menedzselését, akár több eszközükről is. Ahhoz, hogy a konkrétumok tervezését és a megvalósítást el lehessen kezdeni, tisztában kell lennünk a technológiai követelményekkel és azzal, hogy egészen pontosan milyen funkcionalitást várunk el az alkalmazástól.

Ehhez először is azonosítani kell azokat az aspektusokat, amik elengedhetetlenek az alkalmazás megvalósítása szempontjából, valamint a velük kapcsolatosan elvárt funkciók halmazát. Ezután a technológiai elvárások figyelembevételével el lehet kezdeni a megvalósítás tervezését.

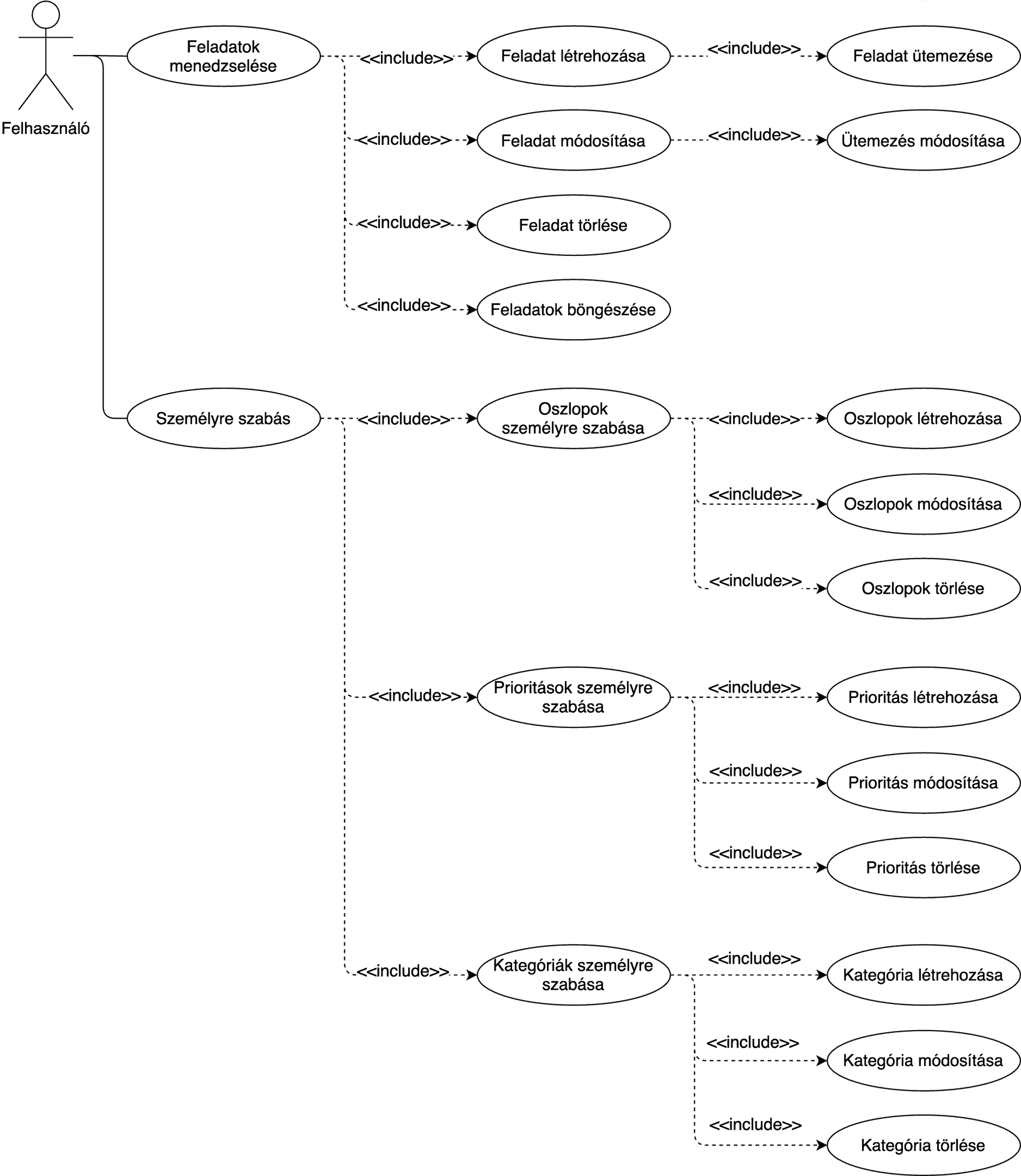
## Technológiai elvárások

A projekt fejlesztése során kiemelt szerepet kapott az, hogy ahol csak lehet ott a lehető legkorszerűbb technológiákat alkalmazzam. Ezek közül a leglényegesebb a *SwiftUI,* ami az Apple platformjain elérhető legmodernebb felhasználói felület (*UI*) fejlesztésére alkalmas keretrendszer. Egy másik fontos, szintén kifejezetten modern technológiai könyvtár, amit alkalmaztam, a *Combine*, ami aszinkron események kezelését teszi lehetővé deklaratív módon. Ezeket az eszközöket részletesebben a következő fejezeben mutatom be.

A felhasznált eszközök mellett fontos elvárás az alkalmazással szemben az, hogy minél több Apple platformon elérhető legyen és a különböző platformok közötti adatszinkronizáció is megvalósuljon. Ezen felül további követelmény, hogy az egyes platformon futó kliensek, használják ki az adott platform adottságait, különös tekintettel a felhasználói felület terén. Az alkalmazás felhasználói élményének növelése érdekében további elvárás, hogy azokon a platformokon, ahol ez lehetséges, a felhasználó *widget* komponensek segítségével is el tudja érni az alkalmazás funkcionalitását.

## Funkcionális elvárások

Az alkalmazás fő aspektusa a feladatok menedzselése egy Kanban táblának megfelelő struktúrán belül. Ez magába foglalja a feladatok létrehozását, módosítását, törlését és böngészését, ismétlődő feladat esetén pedig az ütemezést, illetve az ütemezés módosítását is. Ezen kívül mivel nagy hangsúlyt szeretnénk fektetni a személyre szabhatóságra, a Kanban tábla oszlopai, illetve a feladatoknak adható prioritási szintek, és kategóriák tulajdonságainak teljes mértékben megváltoztathatónak kell lennie a felhasználó által. A megvalósítandó funkciókat az **1. ábra** szemlélteti.



1. ábra: Az alkalmazás funkcióit összefoglaló use-case diagram

# Felhasznált technológiák

A fejlesztéshez felhasznált technológiák kiválasztásánál törekedtem arra, hogy minél újabb és az iparban is használt technológiákat alkalmazzak, illetve, ha volt rá lehetőség, akkor az Apple által nyújtott megoldások mindig prioritást élveztek a harmadik féltől származó eszközökkel szemben.

## SwiftUI

A *SwiftUI* [1] az Apple által fejlesztett keretrendszer, amely a deklaratív *UI* leírást teszi lehetővé.

A deklaratív *UI* készítés az utóbbi évek népszerű trendje a felhasználó közeli technológiák között. Ennek a hátterében az áll, hogy ezzel a megközelítéssel lényegesen kevesebb kód segítségével lehet többé-kevésbe ugyanarra az eredményre jutni, mint az imperatív szemléletű leírókkal. Fejlesztői nézőpontból ez azt jelenti, hogy a felhasználói felület kódja könnyebben olvasható és értelmezhető, valamint nincs szükség a felületek állapotátmeneteinek definiálására, mivel ezt a keretrendszer végzi el.

A *SwiftUI* megjelenése előtt az imperatív *UIKit* [2] könyvtár volt a natív *iOS* fejlesztés, felületkészítésre használt eszköze. A két rendszert összehasonlítva azt tapasztalhatjuk, hogy *SwiftUI* segítségével lényegesen gyorsabb a felhasználói felületek fejlesztése, köszönhetően a korábban említett kisebb mennyiségű kódnak, illetve a *live* *preview* nevű funkciónak. Ez a funkció lehetővé teszi, hogy a fejlesztő környezet egy erre szánt paneljében megtekinthető legyen a kód eredménye anélkül, hogy szimulátoron vagy teszt eszközön futtatni kelljen az alkalmazást.

A *SwiftUI* hátrányai között többnyire olyan aspektusok említhetők, amelyek a technológia frissességének tudhatóak be. Ilyen például, hogy bizonyos beépített felületi elemek személyre szabása meglehetősen nehéz és korlátozott, vagy az, hogy bizonyos gyakran használt komponensek használatához kifejezetten magas a minimum operációs rendszer verziója.

A *SwiftUI* ugyan még mindig fiatalnak mondható, azonban az utóbbi években az ipari projektek körében egyre jobban elterjedt a használata, és mára már szinte minden újonnan induló projektnél erre a keretrendszerre esik a választás.

## Combine

Ahhoz, hogy az alkalmazás zökkenőmentes felhasználói élményt nyújtson, az olyan erőforrásigényes vagy hosszabb ideig futó folyamatokat, mint az adatbázis műveletek vagy a hálózati kommunikáció, aszinkron módon kell megvalósítani. Az Apple két aszinkron kód végrehajtást támogató megoldást fejlesztett ki az utóbbi években.

Az egyik a *Structured* *Concurrency*, amely az *async/await* mintára épít, és segítségével a hagyományos, szinkron metódusok felépítéséhez hasonlóan lehet aszinkron folyamatokat futtatni.

A másik, a reaktív programozási paradigmát megvalósító *Combine* [3] osztálykönyvtár, amellyel kifejező módon lehet adatfolyamokat kiépíteni. Ez a megoldás az úgynevezett *publisher-subscriber* tervezési mintán alapul, aminek lényege, hogy publikáló objektumok üzeneteket tesznek közzé, és ezekre az üzenetekre iratkozhatnak fel a fogyasztók.

A *Combine* használatával lehetőség van több publikálót is láncba kötni, mivel képesek az adatfolyamban felettük álló publikálóktól érkező üzeneteket feldolgozni, és újra publikálni a saját fogyasztóik számára.

A publikálóknak van egy bemeneti és egy kimeneti adatfolyama, a kettő között pedig szabadon lehet transzformálni az adatokat deklaratív metódusok segítségével. Továbbá lehetőség van a folyamban bekövetkező hibák kezelésére, eldobására vagy tovább engedésére is. Ennek köszönhetően a lánc minden elemének csak a közvetlenül előtte álló publikáló által kibocsátott értéket és esetleges hibát kell feldolgoznia. Ezek a tulajdonságok lehetővé teszik a bármilyen hosszú és tetszőlegesen komplex láncok kialakítását.

A könyvtár használatával átláthatóbb és kezelhetőbb az adatfolyam, valamint remekül integrálódik a *SwiftUI* segítségével készített alkalmazásokba, mivel a motorháztető alatt a *SwiftUI* is számos esetben használja a *Combine* megoldásait.

## iCloud

Az *iCloud* [4] az Apple felhőalapú adattároló szolgáltatása, amely lehetővé teszi a felhasználók számára, hogy biztonságosan tárolják és szinkronizálják adataikat a különböző eszközeik között.

Ez a technológia kiválóan alkalmazható, ha az Apple ökoszisztémáján belüli multiplatform alkalmazást szeretnénk létrehozni, hiszen biztosítja az eszközök közötti egyszerű és zökkenőmentes adatmegosztást. Azon kívül, hogy létezik egy *iCloud* szinkronizációra specializált keretrendszer, a *CloudKit* [5], az Apple további keretrendszereiben is támogatva, illetve néhány esetben teljesen integrálva van a technológia használata.

Az *iCloud* szinkronizáció lényege, hogy amikor a felhasználó az egyik eszközén módosít, létrehoz vagy töröl adatokat, akkor erről nem csak egy adatkezelő tranzakció indul, hanem kettő. Az egyik tranzakció lokálisan az adott eszközön végzi el, a másik pedig a felhőben lévő adatokon hajtja végre változtatásokat. Mivel nyilvánvalóan előfordulhat, hogy a változtatás végrehajtásakor nincs az adott eszköznek hálózati kapcsolata, ilyen esetben a tranzakció belekerül egy olyan végrehajtási sorba, ami addig várakozik ameddig nincs hálózati kapcsolat az eszközön. Amint létrejön egy hálózati kapcsolat, a végrehajtási sorban lévő tranzakciókat az operációs rendszer elkezdi elküldeni a felhőben tárolt adatoknak, amíg a két adattároló teljes szinkronban nincs.

Amikor egy tranzakció elér a felhőben lévő adattárhoz, a felhő minden a felhasználó fiókjával azonosított eszköznek küld egy távoli értesítést, a változtatásról, így elérve a teljes szinkronizációt a felhasználó összes eszköze között.

Mivel ilyen módon az *iCloud* csak az úgynevezett *eventual* *consistency* modellnek felel meg, ezért egy olyan alkalmazáshoz, amelyben az adatok időérzékenyek, vagy követelmény a szigorú konzisztencia, ez a felhőszolgáltatás nem alkalmas az adatok tárolására, azonban a jelen dokumentum témájául szolgáló személyes produktivitást növelését célzó alkalmazáshoz remekül felhasználható.

## CoreData

A legtöbb alkalmazásban szükség van az adatok perzisztenciájára, vagyis tartós tárolására, mivel ez biztosítja az adatok hosszú távú megőrzését és hozzáférhetőségét, függetlenül attól, hogy az alkalmazás éppen fut-e vagy sem. Az ilyen jellegű működés nélkül az alkalmazások minden újraindításkor elveszítenék az összes korábban felvett adatot, aminek következményeként rengeteg funkcionalitás nem is lenne lehetséges.

Az Apple több perzisztenciára alkalmas megoldást is kínál. Ezek közül kiemelendő a *CoreData* [6], illetve a rá épülő újabb keretrendszer a *SwiftData* [7]*.* A *CoreData* egy objektum relációs adatkezelő, amely az Apple egyik legkorábbi és ezáltal legkiforrottabb keretrendszere. Lehetővé teszi a magas szintű adatmodellek létrehozását a kapcsolataikkal együtt, valamint segítségével összetett és komplex lekérdezéseket és adatmódosításokat lehet viszonylag könnyedén definiálni és végrehajtani.

Az adatmodellek létrehozására az Apple fejlesztő eszközében az *Xcode* alkalmazásban integrált grafikus felület van biztosítva, így forráskód írása nélkül van lehetőség definiálni a teljes alkalmazás adatmodelljét. Ezen felül működésébe teljesen integrálva van az *iCloud* szinkronizáció támogatása, így minimális konfigurációs lépés elvégzésével a keretrendszerek maguktól el tudják végezni a szinkronizációval járó munka lényegi részét.

Mivel a jelen diplomamunka témájául szolgáló alkalmazásnál egy lényeges technológiai követelmény volt, hogy a lehető legmodernebb technológiákat alkalmazza, ahol lehetséges, ezért külön magyarázatra szorul a tény, hogy a perzisztens adattároláshoz a kevésbé modern *CoreData* keretrendszer lett felhasználva, az újabb *SwiftData* keretrendszerrel szemben. Ennek fejlesztői döntésnek az okai a **Service réteg** fejezetben lesznek kifejtve.

## WidgetKit

A mobiltelefonok és egyéb rendszerint hordozható eszközök nagy előnye, hogy a felhasználók gyakorlatilag bárhol, gyorsan férhetnek hozzá a számukra szükséges információkhoz vagy funkciókhoz. Azonban az tulajdonságot még jobban ki lehet hangsúlyozni az úgynevezett *widget* komponensek felhasználásával.

A *widget* egy olyan általában kisméretű komponens, amit a felhasználók a kezdőképernyőjükön helyezhetnek el és segítségével az adott alkalmazás megnyitása nélkül juthatnak lényeges információkhoz, vagy végezhetnek el egyszerűbb műveleteket. Egy *widget* nemcsak a felhasználói élményt javíthatja, hanem növeli az alkalmazás láthatóságát és használatát is hiszen folyamatosan jelen van a felhasználók képernyőjén.

Ugyan az Apple rendszerein csak az utóbbi években jelentek meg az ilyen jellegű komponensek, azonban már jelentős fejlődésen mentek keresztül. Amellett, hogy az Apple platformjai közül az összes lényegesebb támogatja a használatukat, a fejlesztőknek már lehetősége van interaktív, és a felhasználó által konfigurálható *widget* komponensek létrehozására is.

A *WidgetKit* [8] az Apple által létrehozott keretrendszer, amely lehetővé teszi, hogy a fejlesztők *SwiftUI* használatával készítsenek *widget* komponenseket az alkalmazásaikhoz.

Segítségével hozzá lehet férni az alkalmazás által használt adatokhoz, vagy akár a hálózatról lekért adatokhoz, amiket többféle méretű és konfigurációjú *widget* segítségével van lehetőség megjeleníteni a felhasználó számára.

## SwiftLint

A *SwiftLint* [9] olyan nyílt forráskódú eszköz, ami az Apple platformokon domináns *Swift* programnyelvhez nyújt megoldást a konzisztens kódstílus kikényszerítésére.

Az eszköz fejlesztői által biztosítva van egy alapvető, a közösség által elfogadott szabálykészlet, ami alapján ellenőrzi a fejlesztők által írt kódot, de ezekhez a szabályokhoz továbbiak adhatóak hozzá vagy belőlük szabályok vehetők el egy konfigurációs fájl segítségével. Lehetőség van a szabályok kikapcsolására is , ráadásul ez lokálisan akár egyetlen sor kódra vagy egy egész fájlra vonatkozóan is megtehető, megfelelő tartalmú kód kommentek segítségével.

Az eszköz integrálható a programkód fordításának folyamatába, ezért, ha egy szabály nincs be tartva, az egyből ki is derül a fordítás során. A szabályok megsértése kétfajta hibát okozhat. A súlyosabbnak minősülő stílus hibák a fordítási folyamatot is megszakító *error* jelzést, a kevésbe súlyosak pedig csak figyelmeztető jellegű *warning* jelzést okoznak.

Használatának köszönhetően a kód az egész projektben konzisztensen lesz formázva és könnyebben olvasható, ami megkönnyíti az applikáció karbantartását, és az esetleges új funkciók bevezetését.

## SwiftGen

A *SwiftGen* [10] szintén egy nyílt forráskódú eszköz, ami nagyon népszerű az *iOS* fejlesztők körében. Célja a kódgenerálás erőforrások típusbiztos elérésére. Korábban elsősorban olyan erőforrás típusokhoz volt használva, mint képek, színek, szövegek, illetve betűtípusok, azonban a képekhez és színekhez már létezik az Apple által készített natív kódgenerálási folyamat közvetlenül a fejlesztőeszközbe integrálva. Ettől eltekintve a betűtípusokhoz, és különösen a szöveges erőforrásokhoz továbbra is rendkívül hasznos eszköz.

Legfőbb előnye, hogy már az alkalmazás fordítása közben kiderül, ha egy erőforrás nevét elírta a fejlesztő, vagy nincs hozzáadva a projekthez. A *SwiftGen* használata nélkül, egy ilyen probléma csak a program futása közben derülne ki, ezért sokkal nehezebb lenne megtalálni az okát. Erre példaként szolgál az alábbi kódrészlet és magyarázata.

Text("Example").font(.custom("ProximaNova-Bold", size: 20)) Text("Example").font(Fonts.ProximaNova.bold.font(size: 20))

A fenti két sor kód, ha minden rendben van, ugyanarra az eredményre vezet. Az első sorban egy szöveg segítségél hivatkozunk egy betűtípust tartalmazó fájlra, a második sorban pedig az eszköz által, ugyanehhez a fájlhoz generált kódra hivatkozunk. Ha a hivatkozott fájl törlésre kerül, akkor az első sor kód probléma nélkül lefordul, és a fejlesztőnek nem lesz róla információja, hogy miért nem azt látja a képernyőn, amit elvárt. A második sor ugyanilyen esetben, hibát jelezne a fordítás során, amiből a fejlesztő egyből tudná, hogy a fájl nincs a helyén, és egyből tudja javítani a hibát.

A példában ugyan nem történik súlyosabb probléma, mivel a rendszer a hibás esetben is, az alapértelmezett betűtípussal megjeleníti a szöveget, viszont belátható, hogy egy hasonló esetben, ahol az alkalmazás számára létfontosságú fájl hiányzik, az alkalmazás a hiba megjelenítése nélkül leállna.

Az erőforrásokat, illetve a generált fájl elérési útját és a generáláshoz használt sablon fájlt, egy konfigurációs fájl segítségével lehet definiálni. Az eszköz fejlesztői több előre definiált sablont is a felhasználók rendelkezésére bocsátottak, azonban előfordulhat, hogy ezek nem megfelelőek a céljainknak.

Szerencsére ilyen esetben lehetőség van saját sablon fájlok megírására és felhasználására, a *Stencil* [11] nevű sablon nyelv segítségével. Mivel kódgenerálás futása integrálható a fejlesztett alkalmazás fordítási folyamatába, a konfigurált fájlok minden fordítás során újra generálódnak.

A jelen diplomamunka témáját képező alkalmazás fejlesztése során lehetőségem volt egy olyan teljesen egyedi folyamat kiépítésére is, ami a *Stencil*, illetve a *SwiftGen* fordítási folyamatba való integrálhatóságát kihasználva az Apple legújabb szöveges erőforrásokat tároló fájlformátumából generál típusbiztos erőforrásokat. Ennek a részletei a **Megvalósítás** fejezteben lesznek ismertetve.

A *SwiftGen* használata rengeteg hibakeresési időt megspórolhat a fejlesztőknek, valamint segítségével csökkenthető az applikáció futás idejű hibáinak száma.

## Resolver

A *Resolver* [12] egy nyílt forráskódú, ultrakönnyű eszköz, ami nagyon egyszerű függőség injektálásra ad lehetőséget. A függőség injektálás egy olyan tervezési minta, ami segíti a felelősségek szétválasztását. Segítségével az osztályok kevésbé lesznek kötve egymáshoz, ami növeli a karbantarthatóságot.

A *Resolver* az úgynevezett *Service locator* mintára épül. Ez azt jelenti, hogy egy központi helyen kell regisztrálni azokat az osztályokat, amiket függőségként szeretnénk használni, ahol pedig szükség van rájuk, a *Resolver* az adott osztály típusa alapján ad egy megfelelő példányt.

Az regisztrált osztályok életciklusát lehetőség van befolyásolni a *scope* elnevezésű tulajdonságuk módosítása segítségével. A scope lehetséges értékei közül a leglényegesebbek a *graph*, és az *application*. A *graph* az alapértelmezett érték és ez biztosítja, hogy minden injektálási folyamat egy példányt hozzon létre és a folyamaton belül ez a példány legyen használva. Ezzel szemben az *application* az úgynevezett *Singleton* mintának megfelelő életciklust tesz lehetővé, vagyis az alkalmazás futása során csak egy példány jön létre az első injektálási folyamat során, majd minden ezt követő injektáláskor ez a példány lesz használva.

Mivel a *Resolver* rendkívül jól használható eszköz és a teljesítményre gyakorolt hatása messze felülmúlja a hasonló eszközökét, nagy mértékben javítja a kódbázis fenntarthatóságát és minőségét.

## Sourcery

A *Sourcery* [13] egy olyan nyílt forráskódú eszköz, ami többek között a tesztelést megkönnyítő kód generálását teszi lehetővé a már említett *Stencil* sablonok segítségével.

A megfelelően definiált és generálásra megjelölt protokollok és az általunk megadott sablon fájl mentén képes olyan kód generálására, amivel a tesztek kódjában közvetlenül megadható az elvárt eredménye az egyes műveleteknek. Ez lehetővé teszi, hogy tökéletesen izolálva legyenek azok a funkciók, amiket ténylegesen tesztelni szeretnénk, és ne egy másik osztály esetleges hibás működése okozzon hibát a tesztben.

Természetesen ugyanezt a működést el lehetne érni kézzel írt helyettesítő osztályokkal is, azonban komplexebb osztályok esetén ez nagyobb feladat is lehet, amibe hibák kerülhetnek. A kód generálása emiatt sok időt megspórol, ráadásul a hibák előfordulásának esélye is minimális.

## CocoaPods

A projekt fejlesztéshez felhasznált technológiák, illetve nyílt forráskódú eszközök integrálásához, szükség van egy csomag kezelő rendszerre, ami képes kezelni a külső függőségeket. Az *iOS* fejlesztők körében az egyik legnépszerűbb ilyen eszköz a *CocoaPods* [14].

A *CocoaPods* használatával egy konfigurációs fájl segítségével lehet összegyűjteni, hogy milyen külső függőségekre van a szükség az adott projekten. Ezután a terminálban a megfelelő parancs kiadása után, a rendszer telepíti a kért függőségéket.

Természetesen ahhoz, hogy ez ilyen könnyedén végbe menjen, szükség van arra, hogy a kért függőségek elérhetőek legyenek, úgynevezett *Pod* formában. Mivel a *CocoaPods* az egyik legnépszerűbb csomagkezelő eszköz, ez általában nem okoz problémát a külső eszközök elérésében. Amennyiben az adott eszköz nem érhető el úgy, a forráskódból könnyedén készíthető egy *Pod* a dokumentációban leírt útmutató követésével.

A *CocoaPods* nagy mértékben megkönnyítheti a fejlesztés során felhasznált külső függőségek kezelését, ezért érthető módon rendkívül népszerű és az iparban is gyakran használt eszköz.

# Tervezés

Ebben a fejezetben bemutatom az alkalmazás teljes architektúráját, és az architektúrában megtalálható rétegek felelősségeit, illetve részletezem a felhasználói felület kialakításának lényegesebb lépéseit.

## Architektúra, rétegek

Egy szoftver projekt során az egyik leglényegesebb kérdés, ami már fejlesztési ciklus legelején felmerül, az a szoftver architektúrájának megválasztása. Ez a kérdés különösen fontos hiszen egy rossz architektúra mellett való döntés, rendkívüli módon meg tudja nehezíteni az alkalmazás későbbi új funkciókkal való bővítését, az esetleges hibák javítását, illetve az egész alkalmazás általános karban tartását. Emiatt elengedhetetlen lépés a különböző architektúrák alapos mérlegelése, annak érdekében, hogy megtaláljuk az alkalmazáshoz leginkább megfelelő megoldást.

Egy kliensoldali alkalmazás esetén számos elterjedt architektúra létezik, azonban ezek közül, *iOS* fejlesztésben három lényegesebbet lehet kiemelni. Ezek a *Model-View-Controller* (*MVC*) [15], a *Model-View-ViewModel* (*MVVM*) [16] és a *VIPER* [17]. Mindhárom architektúra arra törekszik, hogy a felhasználói felület kódját a felelősségek jobb elválasztása érdekében, szeparáljuk el az üzleti logikát megvalósító kódtól. Az architektúrák között lévő különbség az elválasztás mértékeben rejlik.

Az *MVC* architektúra három rétegből áll, amelyek közül a *Model* az alkalmazás állapotának tárolásáért felel, míg a *View* a megjelenített felhasználói felületet definiálja. A *Controller* réteg feladata a felhasználói interakciók kezelése, illetve az interakciók következményeként kialakuló állapotváltozások végrehajtása. Bár néhány évvel ezelőtt még ez volt az *Apple* által javasolt architektúra, a gyakran túlzottan hatalmas méretűvé gyarapodó *Controller* osztályok miatt, ma már csak kisebb alkalmazásokhoz érdemes használni.

Ezzel szemben a *MVVM* architektúrában a *Model* rétegmindössze a logikailag összetartozó adatok egységbe zárásáért felelős, míg a *View* réteg feladata ebben az esetben is a felhasználói felületet definiálása. A *ViewModel* réteg felelőssége a felhasználói felület számára biztosítani a megjelenítendő adatokat, továbbá definiálni a felhasználó által végrehajtható műveletek.

A *View* és a *ViewModel* általában az adatkötés mechanizmusán keresztül kommunikálnak egymással, ami azt eredményezi, hogy egy *View* kizárólag egy *ViewModel* példányra támaszkodik, míg a *ViewModel* a *Model* réteghez és a komplexebb logikai műveletekért felelős *Service* réteghez kapcsolódik. Ezáltal egy sokkal lineárisabb, kevesebb rétegközi függőséggel rendelkező architektúrát kapunk.

A *VIPER* architektúra neve, a már említett mintákhoz hasonlóan egy mozaikszó, amely a *View-Interactor-Presenter-Entity-Router* rétegeket jelöli. Ez a megközelítés az *MVVM* architektúra egy tovább gondolt változata. A *View* réteg szerepe a két mintában azonos, valamint az *Entity* réteg is teljesen megfeleltethető az *MVVM* minta *Model* rétegének. A *VIPER* célja a *ViewModel* további felbontása annak logikailag elhatárolható aspektusai mentén. A *Presenter* réteg felel a megjelenítendő adatok kezeléséért, illetve a felhasználói interakciók továbbításáért az *Interactor* felé. Az *Interactor* a kisebb logikai műveleteket hajtja végre és delegálja a komplexebb feladatok végrehajtását az ilyen műveletekért felelős *Service* osztályoknak, valamint biztosítja az ezen műveletek eredményének elérését a *Presenter* réteg számára. A *Router* réteg célja a képernyők közötti navigáció logiájának megvalósítása, amit a *Presenter* réteg kérésére hajt végre.

A *SwiftUI* erőteljesen támogatja az *MVVM* architektúrát, az én alapvető választásom is erre a mintára esett. Mivel azonban mindegyik az alkalmazás szempontjából releváns Apple eszközökön olyan felhasználói felületet szerettem volna alkotni, ami kihasználja az adott eszköz sajátosságait, ezért a különböző platformok között vannak eltérések. Ezen eltérések közül a leglényegesebb az *iPhone* specifikus felhasználói felülethez felhasznált *Coordinator* minta, ami a *VIPER* egyik alapvető elvét illeszti a *MVVM* mintába így létrehozva a végleges *MVVM-C* elnevezésű architektúrát.

A *Coordinator* minta, amelyet az *iOS* fejlesztői közösségben nagy tiszteletnek örvendő *Soroush Khanlou* mutatott be 2015-ben, a *VIPER* architektúra *Router* rétegéhez hasonlóan az alkalmazás navigációjának irányításáért felel. Lényege, hogy a navigációhoz szükséges logikát egy *Coordinator* nevű objektum tárolja, aminek egyetlen felelőssége az alkalmazás képernyői közötti navigáció végrehajtása.

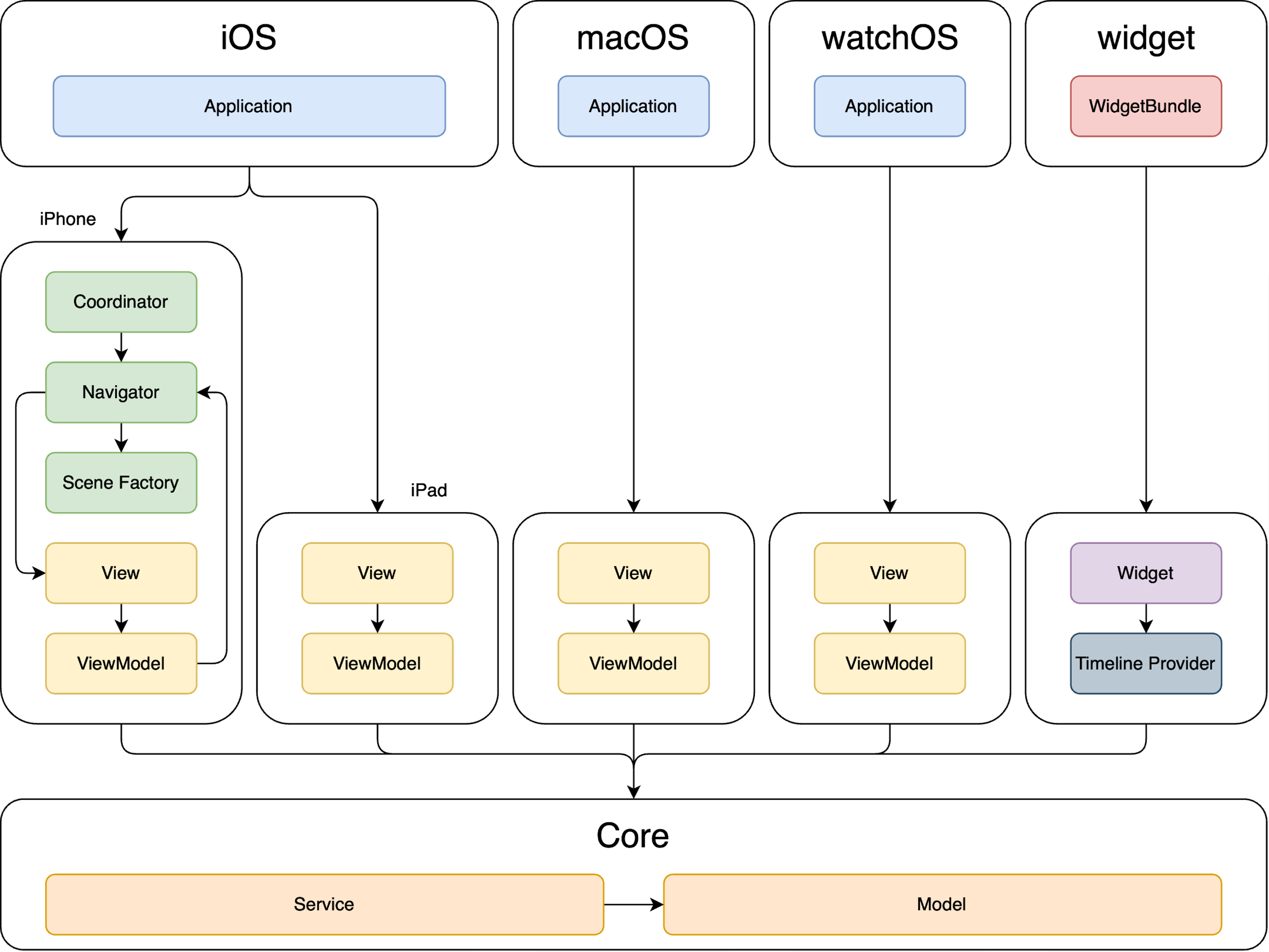
Felhasználásával elérhető, hogy a képernyőknek ne legyen felelőssége az alkalmazásban való navigáció, így kevesebb felelősség hárul rájuk. A minta használata azt is eredményezi, hogy esetleges későbbi fejlesztések során egy teljesen új képernyő létrehozása esetén nem kell lényegesen módosítani a meglévő képernyőket, csak a bővíteni a meglévő *Coordinator* objektumot.

Ennek a mintának az implementációja az Apple korábbi felhasználó felületek készítésére alkalmas keretrendszerében, meglehetősen könnyű feladat volt, annak imperatív jellege miatt. Ezzel szemben az alkalmazásban felhasznált *SwiftUI* egy deklaratív *UI* leíró, ami miatt a navigációs logika alapvető esetben teljesen beágyazódik a felhasználói felület kódjába, így a *Coordinator* minta megvalósítása is több erőfeszítést igényel.

A probléma orvoslásához egy új réteget kell bevezetnünk, ami hidat épít a deklaratív felhasználói felület komponensek és az imperatív kóddal leírt navigációs logika között. Erre több megközelítés is létezik azonban szerencsére a *SwiftUI* 4-es verziójától kezdve elérhető egy *NavigationStack* elnevezésű komponens, aminek segítségével sikerült létrehoznom egy saját megoldást erre a problémára. Ehhez a *NavigationStack* komponensre építve létrehoztam a *Navigator* elnevezésű réteget. Ez a réteg lényegében egy felhasználói felület nélküli komponens, ami megvalósítja a *SwiftUI* *View* protokollját, így képes megjeleníteni bármilyen képernyőt, és a megfelelő logika alapján váltani közöttük.

Mivel az *iPad* és *macOS* specifikus felhasználói felületen az *iPhone* specifikus felhasználói felülethez képest sokkal több terület áll az alkalmazás rendelkezésére, a *watchOS* alkalmazás esetében pedig az interakciók helyett inkább információk nyújtására kerül a fókusz, ezért ezeknél az eszközöknél megvalósítható olyan felhasználói felület, amelyben nincs szükség navigációs logikára és emiatt a *Coordinator* minta használatára sem.

A *widget* komponensek az alkalmazáshoz tartoznak, azonban a helyes működésük eléréséhez az Apple által létrehozott keretek közé vagyunk szorítva, vagyis egy előre meghatározott architektúrába kell illeszkedniük. Ebben az architektúrában a központi egység a *WidgetBundle*, ami tartalmazza az összes elérhető W*idget* komponenst, amik maguk a megjelenítendő felhasználói felületek. Minden egyes *Widget* egy *TimelineProvider* elnevezésű szolgáltatástól kapja a megjelenítendő adatokat. Ezeknek a szolgáltatásoknak azonban valahogyan hozzá kell férnie az alkalmazásban elmentett adatokhoz, különben nem tudnának mit tovább adni a *Widget* számára a megjelenítéshez. Ezért minden *TimelineProvider* hozzáféréssel rendelkezik az alkalmazás *Service* és *Model* rétegéhez, amik segítségével már hozzáférhetnek az alkalmazás releváns adataihoz. Az alkalmazás és hozzá tartozó *widget* komponensek ezek alapján kiépülő architektúráját a **2. ábra** szemlélteti.



2. ábra: Az alkalmazáshoz kialakított végleges architektúra

Az ábrán látható architektúra komponensek közül még két lényegesebbről nem esett szó. Ezek közül a *Service* réteg az alkalmazás működése szempontjából egyik leglényegesebb komponenseket tartalmazza. Fő felelőssége az alkalmazás működéséhez szükséges adatbázis konfigurálása, az adatokon végezhető adatbázis műveletek implementációja, illetve az adatok szinkronizálása az *iCloud* segítségével. Mint látható ez a réteg a *Model* réteggel együtt egy olyan egységet alkot amire minden egyes kliens alkalmazás és minden *widget* komponens támaszkodik, ezért is mondható, hogy ez a két réteg alkotja az alkalmazás magját (*core*).

A másik olyan lényegesebb elem, amiről még nem esett szó, a *SceneFactory.* Ez a komponens a *Coordinator* mintával gyakran együtt előforduló réteg, aminek ahogy arra a neve is, utal az a feladata, hogy inicializálja és felkonfigurálja az alkalmazás képernyőit, ami jelen architektúra esetében a *View* és *ViewModel* objektumok létrehozását és a *Navigator* számára való átadását jelenti.

## Felhasználói felület

Egy esztétikus és jó felhasználói élményt nyújtó design kialakítása nehéz és rendkívül időigényes folyamat, azonban gyorsan elkészíthető vázlatos képernyőtervek segítségével mind a megrendelő mind a fejlesztők láthatják az alapvető elképzeléseket és elvárásokat, amiket az alkalmazás fejlesztés során figyelembe kell venni, így a kliensoldali szoftverek fejlesztésében egy nagyon bevált módszer a képernyő prototípusok készítése.

Amellett, hogy prototípusok könnyen elkészíthetők és változtathatóak, általában megfelelő mennyiségű útmutatást adnak a fejlesztőknek az alapvető funkciók elkészítéséhez, így nem szükséges megvárni a végleges designt, hanem annak készítése folyhat a kezdeti fejlesztéssel párhuzamos ütemben. Mivel a design és az alkalmazásban megtalálható funkcionalitás ideális esetben nem függ egymástól, a végleges design implementációja így lehet az utolsó lépések egyike a projekt lefutásában. Ez több szempontból is előnyős hiszen a prototípusok alapján gyorsan megkezdhető az alkalmazás funkcionalitásának fejlesztése és a design kialakításakor sem kell kompromisszumokat kötnünk az idő hiánya miatt.

Mivel a prototipizálás az iparban is egy remek hatékonyságot mutató módszer, aminek gyakorlatilag nincs negatív velejárója, ezért én is úgy döntöttem, hogy használni fogom. Ez alapján előbb vázlatos képernyőterveket készítettem és ezek segítségével implementáltam az alkalmazás funkcionalitását, a végleges designt pedig csak később alakítottam ki és implementáltam.

### Wireframe készítése

A vázlatos képernyőtervek közül az egyik legalapvetőbb az úgynevezett *wireframe.* Az ilyen terveken általában megjelenik a legtöbb olyan komponens, ami az alkalmazás funkcionalitásához szükséges, egy olyan elrendezésben, ami várhatóan hasonló a végleges design elrendezéséhez.

Véleményem szerint egy *wireframe* akkor igazán nagy segítség a fejlesztőknek, ha nem túl bonyolult, viszont kellően részletes. Ez azért előnyös mert az egyszerűség miatt könnyű megvalósítani kódban és a részletesség miatt vezeti a fejlesztő kezét. Korábban már nekem is volt lehetőségem *wireframe* készítésre ezért ez nem okozott különösebb kihívást, viszont rendkívül nagy segítségemre volt a fejlesztés elkezdésében.

Az alkalmazás szempontjából minden releváns eszközre, minden képernyőhöz készült *wireframe,* mégpedig a *Figma* [18] nevű grafikai design eszköz segítségével*.* Az alábbi képeken az *iPhone,* illetve az *iPad* specifikus képernyőtervekből látható néhány.

|  |  |
| --- | --- |
| A képen elektronika, szöveg, Mobiltelefon, képernyőkép látható  Automatikusan generált leírás  3. ábra: Az iPhone alkalmazás fő képernyőjének terve | A képen képernyőkép, kütyü, Mobiltelefon, Mobileszköz látható  Automatikusan generált leírás  4. ábra: Az iPhone alkalmazás naptárt megjelenítő képernyőjenek terve |
| 5. ábra: Az iPad alkalmazás főképernyőjének terve | |

### Design készítése

Mint ahogy azt már említettem, egy esztétikus és jó felhasználói élményt nyújtó design kialakítása sok időt igényel és nem kevés művészi adottságra is szükség van hozzá. Ilyen téren saját bevallásom szerint meglehetősen kevés tehetségem van, ezért amellett, hogy egy meglehetősen egyszerűnek mondható stílus irányzat mellett döntöttem, a végleges design még így is több iteráción esett át mire meg voltam vele elégedve.

Az általam választott stílus irányzat nem más, mint a *neumorphism.* Ez az irányzat a minimalizmust ötvözi olyan árnyékolási technikákkal, amik olyan hatást keltenek, mintha az egyes felület elemek kiemelkednének a képernyő síkjából vagy belesüllyednének abba.

Ezen felül a design kialakításakor arra is ügyeltem, hogy az alkalmazás témája mind a világos, mind a sötét rendszer témához illeszkedjen, a felhasználó preferenciája szerint. Ezek alapján egy olyan designt sikerült megalkotnom, amivel minden téren meg vagyok elégedve. Az alábbi képeken a bemutatott képernyőterveknek megfelelő elkészült képernyők láthatók az alkalmazásban.

|  |  |
| --- | --- |
| A képen szöveg, Mobiltelefon, kütyü, képernyőkép látható  Automatikusan generált leírás  6. ábra: Az iPhone alkalmazás elkészült főképernyője világos rendszer téma esetén | A képen Mobiltelefon, képernyőkép, kütyü, Mobileszköz látható  Automatikusan generált leírás  7. ábra: Az iPhone alkalmazás elkészült naptárt megjelenítő képernyője világos rendszer téma esetén |

|  |  |
| --- | --- |
| 8. ábra: Az iPhone alkalmazás elkészült főképernyője sötét rendszer téma esetén | A képen elektronika, Mobiltelefon, kütyü, Elektronikus eszköz látható  Automatikusan generált leírás  9. ábra: Az iPhone alkalmazás elkészült naptárt megjelenítő képernyője sötét rendszer téma esetén |
| 10. ábra: Az iPad alkalmazás elkészült főképernyője világos rendszer téma esetén | |

|  |
| --- |
| 11. ábra: Az iPad alkalmazás elkészült főképernyője sötét rendszer téma esetén |

# Megvalósítás

Az architektúra megtervezése, és a képernyőtervek elkészítése után már bele tudtam kezdeni a fejlesztés lényegi részébe. Ebben a fejezetben az architektúra rétegei mentén haladva bemutatom a teljes kódbázis lényegesebb és érdekesebb elemeit.

## Model réteg

A *Model* rétegbe olyan több tulajdonsággal rendelkező objektumok kerültek, amiknek az alkalmazás valamilyen funkcionalitásának szempontjából jelentős szerepe van. A lefektetett funkcionális elvárások mentén ezek az objektumok két főbb csoportba oszthatók. Az egyik csoportot maguk a létrehozható feladatok alkotják, míg a másik csoportban az alkalmazás személyre szabhatósági funkcionalitásához szükséges objektumok szerepelnek.

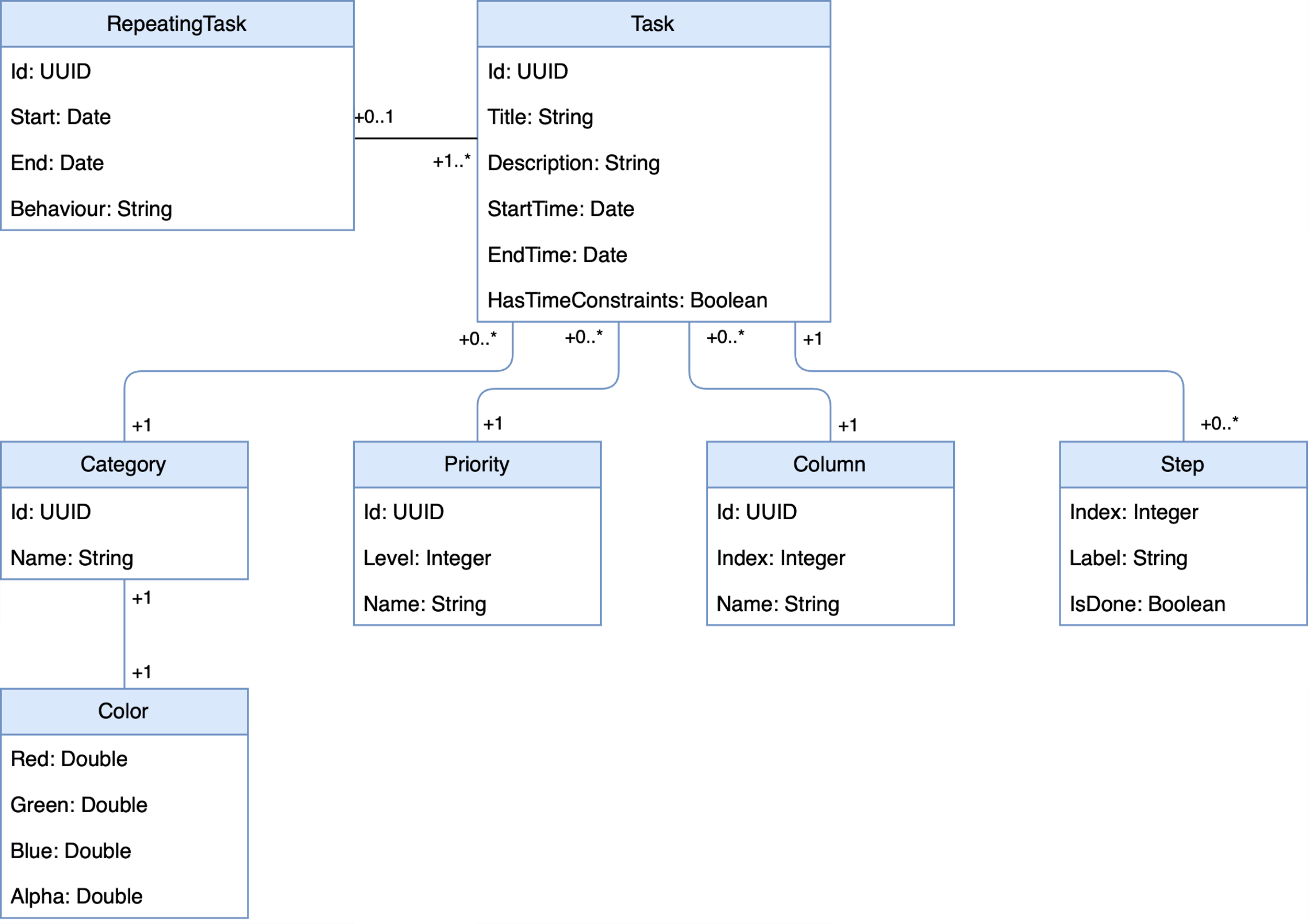
Mivel az alkalmazásban ezektől az objektumoktól perzisztens viselkedést várunk el, a létrehozásukhoz a már bemutatott *CoreData* hívható segítségül. Mint ahogy már említettem az ebben a keretrendszerben kezelt adatmodellekhez, az *Xcode* integrált grafikus felületet biztosít. Ennek a felületnek a segítségével létrehozhatóak az egyes objektumok, minden tulajdonságukkal és más objektumokkal való kapcsolatukkal együtt, majd a létrehozott objektumok alapján a fejlesztői környezet kódot generál számunkra. Az alábbi képen az alkalmazásban használt adatmodell látható a létrehozására használt grafikus felületen.

A képen szöveg, képernyőkép, szoftver, Multimédiás szoftver látható

Automatikusan generált leírás

12. ábra: Az alkalmazásban használt adatmodell az integrált grafikus felületen

Ahogy már említettem a *CoreData* az Apple keretrendszerei közül az egyik legöregebb és emiatt a legkiforrottabb is, azonban ezzel sajnos nem csak előnyök járnak, hanem hátrányok is. Mivel a *SwiftUI* több mint egy évtizeddel a *CoreData* első kiadása után vált elérhetővé, ami szoftverek esetében rengeteg idő, a *CoreData* által generált osztályok használata egy *SwiftUI* segítségével készített alkalmazásban meglehetősen kényelmetlen tud lenni. Szerencsére a *CoreData* által biztosított kódgenerálást át lehet állítani automatikusról manuálisra, aminek következtében az adatmodell alapján való generálás után, a fejlesztőnek lehetősége van módosítani a generált kódon. Így az adatmodell kiegészíthető olyan kényelmi funkcionalitással, amivel már egy *SwiftUI* alapú alkalmazáshoz is alkalmas lesz. Az így kialakított adatmodellt az alábbi ábra szemlélteti.



13. ábra: Az alkalmazás által használt adatmodell

Látható, hogy az adatmodell középpontjában az alkalmazásban létrehozható és menedzselhető feladatokat reprezentáló osztály (*Task*) áll. Minden feladatnak van címe, leírása, tervezett kezdési és befejezési ideje, illetve egy arra mutató értéke, hogy a napon belüli időpontja releváns-e. Ha ez az érték hamis akkor az alkalmazás csak a kezdési dátumot veszi figyelembe, azonban, ha igaz akkor a kezdési dátumot és időt, illetve a befejezési időt is. Ezen felül minden feladatnak van egy referenciája egy kategóriára (*Category*), egy prioritásra (*Priority*), és egy Kanban tábla oszlopra (*Column*).

Az ábrán is látható, hogy a feladatoknak lehet még referenciája feladat lépésekre (*Step*), és ha ismétlődő feladatról van szó akkor az ismétlést szabályozó objektumra (*RepeatingTask*).

Az ismétlődő feladatok létrehozásakor azt szerettem volna elérni, hogy a feladat dátumától eltekintve minden más tulajdonság azonos legyen az ismétlődő feladaton belül, viszont a létrehozás után minden példányt külön, a többi példánytól függetlenül is, meg lehessen változtatni. Emiatt azt a megoldást találtam a legmegfelelőbbnek, hogy egy *RepeatingTask* példány létrehozásánál, egy mintaként kezelt *Task* példány és az ismétlődés paraméterei mentén hozok létre és ütemezek további *Task* példányokat, amiket az ismétlődő feladat egy referencia szintű listában tárolva, gyakorlatilag becsomagol.

Mivel így a kapcsolat inverze miatt az ismétlődő példányoknak is van referenciája a *RepeatingTask*  példányra, könnyedén megoldható, hogy az egyes ismétlődő példányokat megváltoztassuk akár egyesével, akár az összeset egyszerre a közös csomagolón keresztül.

Az ütemezés az osztály kezdési és befejezési dátum tulajdonságai, valamint az ismétlődés szabályát leíró tulajdonsága (*RepeatBehaviour*) alapján történik. Ezekből a tulajdonságokból a két dátum, úgy érzem nem szorul különösebb magyarázatra, viszont a *RepeatBehaviour* már annál inkább. Ezt a tulajdonságot ugyanis akár fel lehetne több egyszerűbb struktúra segítségével is tüntetni az adatmodellben, azonban az egyszerűbb kezelése érdekében úgy döntöttem, hogy a megjelenítési réteg kivételével ez a tulajdonság csak kódolva jelenjen meg az alkalmazásban.

A kódolás első karaktere arra utal, hogy milyen típusú a kódolt szabály, ami lehet napi (D), heti (W), havi (M), éves (Y) vagy egyedi (C) ismétlési szabály. Az első négy érték esetében az adott feladat a típusnak megfelelően, naponta, hetente, havonta, vagy évente fog ismétlődni. Ezekben az esetekben a kódolás további részének már nincs relevanciája, azonban, ha egyedi ismétlési szabályról van szó, akkor igen. Ilyenkor a második karakter az ismétlés alap egységét adja meg, ami lehet nap (D), hét (W), vagy hónap (M), a harmadik karakter pedig az egység frekvenciáját, egy természetes számmal. A kódolás negyedik és egyben utolsó részének csak akkor van relevanciája, ha az ismétlés alap egysége hét vagy hónap. Ekkor ez a rész egy kötőjelekkel elválasztott lista, aminek elemei az alap egységen belül választott napok indexei nullától indulva. Az alábbi kódrészlet egy kódolt ismétlődési szabályra ad példát.

C/W/2/1-2-4

Az itt látható szabály ezek alapján tehát egy egyedi ismétlési szabály, ami minden második héten ismétlődik kedden, szerdán és pénteken.

Felmerülhet a kérdés, hogy mi történik, ha az ismétlési szabály tartalmaz olyan indexet, ami nem minden esetben érvényes. Erre nagyon egyszerű példa lehet egy olyan ütemezés, ami minden hónap 31-én ismétel egy feladatot, mivel nyilvánvalóan nem minden hónap 31 napos. Ez nem is feltétlenül olyan ritka eset, ezért az ütemező algoritmust úgy hoztam létre, hogy az ilyen helyzeteket kezelni tudja az elvárásoknak megfelelően.

A kis kitérő után az adatmodellen tovább haladva a prioritást és a Kanban tábla oszlopait reprezentáló osztályokhoz érünk. Ezeknek az osztályoknak a példányai között fontos a sorrendiség, ezért az alkalmazásban megjelenített nevük mellett egy a sorrendben elfoglalt helyükre utaló tulajdonságot is tartalmaznak.

A kategóriát reprezentáló osztály, a név tulajdonságán kívül rendelkezik egy referenciával, a kategóriához tartozó szín (*Color*) objektumra. A *Color* osztály a hagyományos piros-zöld-kék-átlátszóság (*RGBA*) modell szerint tartalmazza a kategóriák színét. Ennek az osztálynak az ilyen szintű modellezésére csak és kizárólag azért van szükség, mivel a *CoreData* sajnos nem képes a *SwiftUI* által használt beépített szín objektumok elmentésére, így egy olyan formába kell hozni a színeket, amit a *CoreData* is képes eltárolni.

Az adatmodell utolsó eleme a *Step* osztály, ami feladatokon belüli lépéseket reprezentálja. Az adott lépés neve mellett, a lépés sorszámával és azzal jellemezhető, hogy a felhasználó elvégezte-e már.

Az adatmodellben, a *Step* és a *Color* leszámításával, minden osztály rendelkezik egy *Id* elnevezésű tulajdonsággal is, amivel az adott példányok egyértelműen azonosíthatóak, így egyediségük is egyszerűen garantálható.

## Service réteg

A *Service* réteg az alkalmazás egyik leglényegesebb rétege, amely elsősorban az előzőleg bemutatott adatmodell, perzisztens tárolásával kapcsolatos folyamatokat végzi. Emiatt a réteg középpontjában a már bemutatott *CoreData* keretrendszer áll.

### SwiftData

Korábban említettem, hogy ez egy viszonylag régóta létező keretrendszer, aminek van egy újabb alternatívája, mégpedig a közvetlenül ráépülő *SwiftData*. Ezért teljesen jogosan merül fel a kérdés, hogy miért nem a modernebb technológiát alkalmaztam az alkalmazás fejlesztéséhez. Mivel az alkalmazás fejlesztésének kezdete és a *SwiftData* első verziójának kiadása között csak pár hét telt el, ezért eredeti terveim szerint ezt a keretrendszert használtam volna. Azonban miután egy hetet eltöltöttem a vele való kísérletezéssel, úgy ítéltem meg, hogy egy olyan komplexitású alkalmazáshoz, mint a dolgozat témája, egyelőre nem tudom ajánlani. Kiemelném, hogy ez egy rendkívül szubjektív személyes vélemény, ami kizárólag a keretrendszer első verzióját értékeli ezért, nem alkalmas a keretrendszer konkluzív értékelésére.

Az első problémám a *SwiftData* használatával a *SwiftUI* keretrendszerrel való túl szoros csatolás. Mint minden olyan keretrendszernek, ami perzisztens tárolást tesz lehetővé, a *SwiftData* egyik leglényegesebb funkciója a mentett adatok lekérdezése, azonban más keretrendszerekkel ellentétben, a *SwiftData* használatával az erős csatolás miatt, erre gyakorlatilag csak közvetlenül a *View* rétegben van lehetőség. Emiatt használata gyakorlatilag ellehetetleníti bármilyen érdemi architektúra követését, ami egy kis alkalmazásnál nem is feltétlenül probléma, azonban egy komplexebb alkalmazásnál, rengeteg fenntarthatósági kérdés merülhet fel emiatt.

A második nagyobb problémám a keretrendszerrel, hogy túl sokat épít az Apple egyik újabb kódgenerálási módszerére, a *macro* elnevezésű koncepcióra. A *macro* egy olyan technológia, aminek segítségével bizonyos kódrészleteket megjelölve extra funkcionalitást adhatunk nekik a fordító által generált kódon keresztül. Használatával általában rengeteg sablonosnak mondható kód megírása válik feleslegessé, azonban van egy nagyobb hátrányos velejárója is.

Ez pedig az, hogy ha esetlegesen valamit nem jól írtunk meg abban a kódban, amihez a macro kódot generál, akkor a fordító általában nem az általunk megírt kódban jelzi a hibát, hanem egy attól teljesen független hibaüzenettel a generált kódban. A generált kód pedig általában meglehetősen nehezen olvasható, így a kapott hibaüzenetből is szinte lehetetlen következtetni a valódi probléma okára.

A harmadik probléma, ami nagyon nehézzé tette a keretrendszerrel való munkámat az a dokumentációjának hiányossága. Véleményem szerint az egész dokumentáció a minimalizmus hívében íródott, ami egy olyan technológia esetében, amit potenciálisan rengeteg fejlesztő használna, nem a legjobb hozzáállás. Minden a keretrendszerben lévő koncepcióra, a lehető legegyszerűbb példák vannak bemutatva, ami miatt sok esetben csak találgatni lehet, hogy az adott kérdést, hogy lehetne megoldani. Ezt még tovább fokozza a tény, hogy mivel a keretrendszer pár héttel az én kísérletezésem előtt jelent meg, még a fejlesztői közösségben sem gyűlt össze olyan tudás halmaz vele kapcsolatban, amit hasznosnak találtam.

Mivel ezeknek a problémáknak a hatása miatt, rendkívül lassan haladtam, úgy döntöttem, hogy a gyorsabb haladás, és könnyebb fenntarthatóság érdekében ebben az esetben jobb döntés nem a legfrissebb technológia alkalmazása. Tekintve, hogy a döntés meghozatala után, az egy hét kísérletezéssel megalkotott funkcionalitást, a *CoreData* használatával mindössze 4 óra alatt sikerült replikálnom, ezt a döntésemet nem bántam meg.

### CoreData

Mivel az alkalmazás technológiai elvárásai alapján, szeretnénk elérni azt, hogy az elmentett adatokat ne csak az éppen aktuális eszközről érje el a felhasználó, hanem minden megfelelő eszközéről, amihez szükség van az *iCloud* használatára az alkalmazásban. Szerencsére a *CoreData* használatával ez nagyon egyszerű, mivel csak a megfelelő konténer osztályt kell példányosítani, és konfigurálni, ami után az eszközre mentett adatok automatikusan szinkronizálódnak a felhasználó tárhelyébe a felhőben, illetve a többi eszközére is.

Az alkalmazásban a funkcionális követelmények közé tartozik, hogy a Kanban tábla oszlopait, a prioritási szinteket és kategóriákat személyre lehet szabni. Mivel ez nem egy elvárás a felhasználó felé, hanem csak egy lehetőség, ezért az alkalmazásban biztosítva vannak kiindulási értékek.

Ezeket a kiindulási értékeket az alkalmazás az adatbázis kapcsolat felállításakor hozza létre, abban az esetben, ha az adatbázisban nincs a típusoknak megfelelő egyetlen érték sem. Erre az alábbi példakódban lehet példát látni.

private func initiateDefaultPriorities() {

guard let priorities = try? context.fetch(Priority.fetchRequest()),

!priorities.isEmpty else {

**let** low = Priority(context: context)

low.wId = UUID(uuidString: "AC387949-34E7-46FB-9E45")

low.wLevel = 1

low.wName = "Low"

**let** medium = Priority(context: context)

medium.wId = UUID(uuidString: "4B5148CA-60A0-441B-BFD3")

medium.wLevel = 2

medium.wName = "Medium"

**let** high = Priority(context: context)

high.wId = UUID(uuidString: "8925F42B-4C7D-4359-8FF5")

high.wLevel = 3

high.wName = "High"

return

}

}

Ez a metódus az adatbázis kapcsolat felállítása után fut le minden alkalommal. Megfigyelhető, hogy a metódus lényegi része kizárólag akkor fut le, ha a metódus elején végzett adatbázis lekérdezés (*fetch*) eredménye üres. Ha ez nem igaz akkor már vannak megfelelő értékek az adatbázisban, így nincs szükség a hozzáadásukra.

Maguk a kiindulási példányok, a kódba égetett értékek alapján kapják a tulajdonságaikat, még az egyedi azonosítóikat is. Erre azért van szükség, mert az iCloud nem garantálja, hogy minden adatot rögtön frissíteni fog minden eszközön, aminek következtében előfordulhat pillanatnyi inkonzisztencia.

Ilyen eset például, ha a felhasználó egy új eszközére telepíti az alkalmazást, hiszen ilyenkor nem tudjuk garantálni, hogy az iCloud szinkronizáció azelőtt végbe megy, mielőtt az alkalmazás ellenőrzi a kiindulási értékek jelenlétét. Ilyen esetben az alkalmazás újra létrehozná a kiindulási értékeket, majd amint a szinkronizációs ciklus is lefutott a felhasználó a korábbi értékeket is látná. Ezt a problémát úgy oldottam meg, hogy minden lekérdezésnek, része egyben egy olyan metódus is, ami az azonosító alapján képes a duplikált elemek eltávolítására. Ennek a metódusnak viszont csak úgy tudja kiszűrni a duplikált elemeket, ha azok azonosítója megegyezik.

A *Service* réteg a *Combine* segítségével szolgáltatja az adatokat az alkalmazás többi részének. Mint ahogy már a *Combine* bemutatásakor említettem, ez a keretrendszer feliratkozás alapú adatfolyamok kiépítésére használható. Az alkalmazás esetében az adatfolyam forrása a *Service* réteg, amelyben az adatmodell minden eleméhez van legalább egy, esetenként több *CurrentValueSubject* típusú publikáló. Az alkalmazás többi része ezeknek a publikálóknak az üzeneteire iratkozik fel, a megfelelő adatok elérése érdekében. Ezek a publikálók az üzeneteiket az alkalmazás felsőbb rétegeiből, érkező kérésekre teszik közzé a megfelelő adatbázis műveletek elvégzése után. Az alábbi kódrészlet egy ilyen kérést megvalósító metódust mutat be.

func fetchPriorities() {

guard let priorities = try? context.fetch(Priority.fetchRequest()) else {

return

}

self.priorities.send(priorities)

}

Mint látható az adatbázis lekérdezést a *fetch* művelet végzi el, majd az eredményét a *priorities* elnevezésű *CurrentValueSubject* publikáló *send* metódusával küldi el a feliratkozóknak

Az adat lekérdezések mellett a *Service* réteg felelős az összes adatbázis műveletért, úgy, mint a létrehozás, a módosítás és a törlés. Ezen műveletek közül a törlést emelném ki, azon belül is a személyre szabható elemek törlését. Ha a felhasználó egy olyan személyre szabható elemet törölne, ami részese egy az adatbázisban létező feladatnak, vagy az utolsó példány a típusából, akkor az több problémát is okozna, mind a megjelenítés terén, mind az üzleti logikában. Azonban, ha szimplán megtiltjuk az ilyen elemek törlését akkor a felhasználó nem kap róla információt, hogy miért nem történik meg egy olyan művelet, amit el akar végezni.

Ennek a problémának a megoldásához először is létrehoztam az adott hibáknak megfelelő hiba típusokat, és ezekhez tartozó, a felhasználó számára is megjeleníthető részletes hiba leírásokat. Ezután a törlés művelete közben a megfelelő ellenőrzések eredményétől függően, vagy sikeresen végbemegy a törlés művelet vagy a metódus egy olyan hibát dob a hívónak, amit a felhasználó számára is megjeleníthető és informatív. Egy ilyen törlési metódusra ad példát az alábbi kód részlet.

func deleteCategory(\_ category: Category) throws {

if categories.value.count == .one {

throw DataOpError.lastOfKind(type: Category.entityName)

} else if (category.wTasks?.count ?? .zero) > .zero {

throw DataOpError.existingRelationship(type: Category.entityName)

} else {

delete(item: category)

fetchCategories()

}

}

Ahogy látható elsőként azt ellenőrzi a metódus, hogy hány darab, ez esetben kategória van. Amennyiben az éppen törlésre kijelölt kategória az utolsó akkor, egy ennek megfelelő hibát dobunk. Ehhez hasonlóan, az is ellenőrzésre kerül, hogy a törlésre jelölt kategória tartozik-e feladathoz, mivel az ilyenek törlését is meg szeretnénk akadályozni. Ha ezek közül egyik eset sem áll fenn, akkor elvégezzük a törlést, majd annak érdekében, hogy az alkalmazás felsőbb rétegeinek frissek maradjanak az információi, végzünk egy új lekérdezést az adott osztály példányaira.

## Navigációs rétegek

Ezeknek a rétegeknek, ahogy már korábban említettem csak az *iPhone* alkalmazás keretein belül van szerepe. Az itt bemutatott, általam létrehozott megoldás körülbelül három nagyobb iteráción esett át, különböző egyetemi projektjeimben használva, és még mindig vannak ötleteim a fejlesztésével kapcsolatban. Alapját a *NavigationStack* elnevezésű *SwiftUI* nézet adja, aminek használatával létre lehet hozni egy nagyon egyszerű verem alapú navigációs keretrendszert, ami áthidalja a deklaratív felület leírás és az imperatív navigációs logika közötti kapcsolat hiányát.

Ennek a keretrendszernek négy fő komponense van. Az első a *Coordinator*, amely az egész navigációt egybe fogja, és elindítja. A képernyők tényleges megjelenítését és a köztük való váltást, a *Navigator* objektum végzi, ami tartalmazza a *NavigationStack* példányt és magát a navigációs vermet. A megjelenített képernyőt a *Navigator* *SceneFactory* példánya készíti el és konfigurálja, a verembe került értéktől függően. A *SwiftUI* megkötései miatt, a verembe csak bizonyos követelményeknek megfelelő típusok tehetőek, így szükség van egy olyan típusra, ami megfelel ezeknek a követelményeknek és áttudja adni a képernyő konfiguráláshoz szükséges adatokat a SceneFactory számára. Ez a típus pedig egy felsorolt típus (*enum*) ami megvalósítja a negyedik komponenst, vagyis a *Screen* protokollt.

public struct Navigator<Factory: SceneFactory>: NavigatorInput {

@State private var presentationStack = NavigationPath()

public let factory: Factory

public let root: Factory.FlowScreen

public var flowFinished = CurrentValueSubject<Bool, Never>(false)

public var body: some View {

NavigationStack(path: $presentationStack) {

factory.view(for: root, with: self as! Factory.SpecificNavigator)

.navigationDestination(for: Factory.FlowScreen.self) { screen in

factory

.view(for: screen, with: self as! Factory.SpecificNavigator)

}

}

}

}

A fenti kódrészleten a *Navigator* objektum látható, aminek segítségével bármilyen képernyő megjeleníthető. Látható, hogy a keretrendszer legtöbb típusa protokollokon keresztül jelenik meg. Ez azért lényeges mert így a navigáció jelentős része független a projekt többi részétől és ezáltal újra felhasználható más projektekben, minimális erőfeszítés és kód írása árán.

Ahogy látható a *NavigationStack* példány egy kezdő képernyő (*root*) és a verem (*presentationStack)* alapján kéri el a *SceneFactory* példánytól az aktuálisan megjelenítendő képernyőt. A *Navigator* ezen felül rendelkezik egy-egy olyan metódussal is, aminek segítségével értékeket lehet a veremhez adni, illetve abból elvenni. Ezeket a metódusokat hívja meg a *ViewModel* réteg amikor a felhasználó egy olyan műveletet végez, ami navigációval jár.

Felmerülhet a kérdés, hogy ha a verembe egy felsorolt típus értékei kerülnek bele akkor hogyan lehet paramétert átadni az olyan képernyőknek, amiknek szüksége van erre. A választ erre a kérdésre a *Swift* nyelv egyik rendkívül hasznos különlegessége adja, mégpedig az *associated value*, vagyis a hozzárendelt érték. Ez egy olyan sajátossága a nyelvnek, amivel bizonyos felsorolt típusokhoz hozzá lehet rendelni további értékeket, majd a megfelelő helyen felhasználni őket. Segítségével a *ViewModel* réteg képes átadni, a *SceneFactory* pedig képes feldolgozni az egyes képernyők megjelenítéséhez szükséges paramétereket. Az alábbi kódrészleten, ami az *iPhone* alkalmazás képernyőinek megfelelő, felsorolt típust tartalmazza, megfigyelhető az ilyen hozzárendelt értéket kezelni képes felsorolt típus.

public enum ContentScreen: Screen {

case addTask(\_ date: Date)

case calendar

case dashboard

case day(\_: Date)

case settings

case task(\_: UUID)

}

Scene factory pelda + bemutatas

## ViewModel réteg

## View réteg

### iPhone alkalmazás

### iPad alkalmazás

### macOS alkalmazás

### watchOS alkalmazás

## Widget komponensek

# Tesztelés

A szoftverfejlesztésben egy nagyon lényeges folyamat, és az iparban általában a szoftverprojektekkel szemben állított egyik elvárás is, a szoftverek tesztelése. Amellett, hogy maguk a tesztek segíthetnek kimutatni bizonyos hibákat a kódbázisban, a tesztelhetőség megkövetel egy bizonyos szintű minőséget az írt kódtól. Ezáltal nemcsak azt tudjuk garantálni, hogy az egyes tesztelt komponensek hiba nélkül működnek, hanem azok karbantarthatóbbak, könnyebben bővíthetőek, és a bővítések vagy esetleges módosítások miatt előforduló hibák kockázata is kisebb.

Kliensoldali alkalmazások esetén a kód tesztelésének általában két főbb csoportjáról lehet beszélni. Az egyik csoport az egység vagy más néven *unit* tesztelés, melynek során az alkalmazás üzleti logikával kapcsolatos komponenseinek a helyes működését vizsgáljuk. A másik csoport pedig az úgynevezett felület tesztek, amik a felhasználói interakciók által kiváltott képernyőváltozások helyességét tesztelik.

Ebben a fejezetben bemutatom a dolgozat témájául szolgáló alkalmazás egység és automatizált felület tesztelését, a tesztelés nehézségeit és ezek megoldását, illetve a tesztek segítségével elért kódlefedettséget.

## Egység tesztelés

Az alkalmazás fejlesztéséhez az *iOS* fejlesztésben legalapvetőbb fejlesztői környezetet használtam, ami nem más, mint az *Xcode*. Szerencsére ez a fejlesztői környezet teljes támogatást nyújt tesztek készítéséhez, futtatásához és az eredmények elemzéséhez.

Az *iOS* alkalmazásokhoz írt tesztek rengeteg ideig egy *XCTest* elnevezésű keretrendszerre voltak építve, azonban az alkalmazás fejlesztése során az Apple kiadott egy újabb tesztelési keretrendszert, amit kiválóan lehet alkalmazni az egységtesztek írásához. Mivel az alkalmazás fejlesztésében az egyik fő technológiai célként a lehető legmodernebb technológiák alkalmazását tűztem ki célul, ezért logikus döntésnek tartom, hogy az alkalmazás egység tesztelését ennek az új, *Swift Testing* elnevezésű keretrendszernek a használatával végeztem.

A *Swift Testing* keretrendszerben a teszteléshez, csak meg kell jelölni a teszteseteket megvalósító metódusokat az *@Test* annotációval, és a fejlesztői környezet ezután automatikusan felismeri őket, mint tesztek. Ezek a metódusok lehetnek struktúrák, vagy osztályok metódusai, de akár globális metódusok is.

Én azt választottam, hogy a teszteseteket a tesztelt komponensek szerint struktúrákra, illetve osztályokra bontom szét, aminek több előnye is van. Először is, így a tesztek sokkal rendezettebbek, ezért könnyebb navigálni közöttük, valamint így lehetőség van felhasználni a *Swift* *Testing* egyik újdonságát, az *@Suite* annotációt.

Az annotáció használata két fő előnyt nyújt a fejlesztőnek, melyek közül az egyik az alternatív név megadása. Ez azért lehet előnyös mert esetenként a tesztelt komponens és így a hozzá tartozó tesztosztály neve is meglehetősen bonyolult, azonban egy alternatív név segítségével, egy emberi szemmel sokkal olvashatóbb névre cserélhetjük, és a fejlesztői környezet ezt az olvashatóbb nevet fogja megjeleníteni nekünk.

A másik előnye az *@Suite* használatának a címkézés. Lehetőségünk van létrehozni saját címkéket, és a címkékkel megjelölni a tesztosztályokat. Ezután a fejlesztői környezetben lehetőség van a címkék szerint keresni a tesztek között, és akár úgy is futtathatunk teszteket, hogy csak egy adott címkével megjelölt tesztek legyenek végrehajtva.

@Suite("ViewModel tests for task adding screen", .tags(.viewModel))

final class AddTaskViewModelTests {

...

}

A fenti kódrészletben az *@Suite* használatára látható egy példa. Látható, hogy a tesztosztálynak adtam egy gyakorlatilag mondatként olvasható alternatív nevet, illetve megjelöltem az általam létrehozott *viewModel* címkével. A példából nem látszik, ezért fontosnak tartom megjegyezni, hogy egy tesztosztályt több címkével is meg lehet jelölni, ha arra szükség van.

A tesztesetek osztályokra való szétbontásának egy másik nagyon lényeges előnye, hogy így lehetőségünk van a tesztek előtt szükséges konfigurációs lépések, illetve a tesztek után futó állapot tisztító lépesek kiszervezésére a tesztosztály inicializáló, illetve deinicializáló metódusába. Ezek a metódusok minden teszteset előtt, illetve után lefutnak, így garantálni tudjuk, hogy a tesztesetek egymástól függetlenül, izolálva fussanak.

Minden tesztesetet megvalósító metódust a *Given-When-Then* struktúrát követve hoztam létre, vagyis először létrehozzuk a tesztelt komponens kiinduló állapotát (*Given*), majd végrehajtjuk a tesztelendő lépést (*When*), és végül ellenőrizzük, hogy a megfelelő állapotba jutottunk a lépés hatására (*Then*).

Az alkalmazáshoz írt egység tesztek, két csoportba oszthatók. Az egyik csoport a segítő (*utility)* komponensek tesztjei, a másik pedig a *ViewModel* réteghez tartozó tesztek.

### Utility komponensek tesztelése

A segítő komponensek közé az alkalmazás olyan részei tartoznak, amik kifejezetten nem tartoznak egyik, az architektúrában említett réteg közé sem, azonban helyes működésük elengedhetetlen az alkalmazás elvárt viselkedéséhez. Ilyen komponensek például a szöveg manipulációért felelős metódusok, amik nagy szerepet kapnak az alkalmazásban működő szűrési funkciókban. Az alábbi kódrészletben egy ilyen metódushoz tartozó teszt látható.

@Test(

"Case-insensitve contains test",

arguments: zip(["hello", "goodbye"], [true, false])

)

func caseInsensitiveContains(word: String, isContained: Bool) {

let text = "Hello, World!"

let result = text.caseInsensitiveContains(word)

#expect(result == isContained)

}

Ez a teszt egy olyan segéd metódus helyes működését vizsgálja, ami azt ellenőrzi, hogy egy adott szöveg tartalmaz-e egy másik szöveget attól függetlenül, hogy a két szövegben mi a normál, illetve a kapitális betűk összetétele. Ezen a teszten kiválóan megfigyelhető a *Given-When-Then* struktúra felépítése, mivel mindegyik blokkhoz pontosan egy sor kód tartozik a tesztesetben. Először létrehozzuk a szöveget, amiben keresni fogunk, majd meghívjuk a keresést végző metódust és végül az *#expect* makró segítségével ellenőrizzük az eredményt.

Látható, hogy az *@Test* annotáció használatával az egyes teszteseteknek is adhatunk egy alternatív nevet, a tesztosztályokhoz hasonlóan. A példán megfigyelhető a *Swift Testing* egyik másik újdonsága, a paraméterezett teszt metódus is.

Egy paraméterezett teszt metódussal elérhető, hogy egy tesztet több elképzelhető konfigurációban is futtasson a rendszer anélkül, hogy minden konfiguráció számára egy külön tesztesetet kelljen létrehozni. Ehhez, ahogy a kódrészleten is látható a teszt metódusnak létre kell hoznunk paramétereket, amiket az *@Test* annotáció *arguments* paraméterében lehet fel konfigurálni.

Jelen esetben a konfiguráció két értékpárból áll, amiknek az első értéke a szöveg, amit a kiinduló szövegben keresünk, a második érték pedig az elvárt eredménye a keresésnek. Így tehát ez a teszt két külön az adott értékpárnak megfelelő konfigurációban fog lefutni a tesztelés során, azonban csak egy tesztesetet kellet megírnom.

### ViewModel réteg tesztelése

A *ViewModel* réteg az alkalmazás egyik legnagyobb és leglényegesebb rétege, hiszen ez az a réteg, ami a felhasználó számára is közvetlenül látható adatokat szolgáltatja, így tesztelése elengedhetetlen.

Mivel ebben a rétegben az adatok nagy része a *Service* rétegből származik, ezért ahhoz, hogy izolálni tudjuk a teszteket, szükség van úgynevezett helyettesítő (*mock*) osztályokra, amik képesek a *ViewModel* szempontjából helyettesíteni a *Service* réteget, viszont a tesztek szempontjából úgy viselkednek ahogy mi azt a teszteléshez szeretnénk.

Mivel az alkalmazás fejlesztését már a tesztelhetőség szükségességével a fejemben kezdtem meg, ezért minden *Service* osztályt egy olyan protokoll mentén hoztam létre, ami tartalmazza az összes publikus tulajdonságát és metódusát. Így amikor a tesztelésre jutott az idő, a protokollok és a már bemutatott *Sourcery* segítségével nagyon egyszerűen tudtam generálni a teszteléshez megfelelő *mock* osztályokat.

A *ViewModel* rétegnek azon metódusait tudjuk tesztelni, amik publikusak. Ezek a metódusok pedig mind, egy a felhasználó által kiváltott eseményhez, például egy gomb megnyomásához tartoznak. Emiatt maguk a tesztesetek meglehetősen egyszerűek, azonban a mennyiségük kifejezetten nagy. Ez nem is probléma, hiszen komplexebb tesztek esetén nehezebb lenne meghatározni egy esetleges hiba pontos helyét. Az alábbi kódrészlet egy átlagosnak mondható ViewModel réteghez tartozó tesztesetre ad példát.

@Test("Tapped delete button with repeated task")

func didTapDelete\_withRepeated() {

sut.didTapDelete(task: DataServiceInputMock.repeatedTaskMock)

#expect(sut.isAlertPresented)

}

Ez a teszt egy olyan esetet vizsgál, amikor a felhasználó úgy nyom rá a törlés gombra, hogy a törlendő feladat ismétlődik. Ilyenkor az alkalmazásnak fel kell kínálnia a lehetőséget, hogy ne csak az adott feladatot, hanem az összes ismétlődő előfordulását is törölje. Így logikusan a teszt azt ellenőrzi, hogy a felajánlást megjelenítő modális ablak a képernyőn van-e.

Észrevehető, hogy ebben a tesztesetben nincs a *Given* blokknak megfelelő kódrészlet. Ennek az az oka, hogy a *ViewModel* tesztek esetében általában mindig egy közös kiinduló állapotból szeretnénk futtatni minden tesztesetet, így célszerűbbnek találtam, hogy ezt a kiinduló állapotot a tesztosztály inicializáló metódusában állítsam be.

Ilyen és ehhez hasonló jellegű tesztek készültek a *watchOS* kivételével, az összes alkalmazás szempontjából releváns platformon, a *ViewModel* rétegbe tartozó összes osztályhoz. A *watchOS* alkalmazáshoz pedig azért nem készítettem egység teszteket mivel, ahogy már említettem ezen a platformon az alkalmazás fókuszában a gyors információ szolgáltatás van, és így a *ViewModel* réteg nem tartalmaz tesztelhető publikus metódust.

Ezzel az alkalmazáshoz készített egység tesztek bemutatásának végére értem. Felmerülhet a kérdés, hogy az alkalmazás architektúrájában látott rétegek közül miért csak a *ViewModel* réteghez készültek egység tesztek. Ennek elsősorban az az oka, hogy ezen a rétegen kívül, még a *Service* réteg szerepe elég nagy ahhoz, hogy érdemes legyen tesztelni, azonban ez a réteg nagyon szorosan függ a *CoreData* és az *iCloud* helyes működésétől. Emiatt valódi érdeme nem lenne a tesztelésének, hiszen a helyes működése gyakorlatban megegyezik azzal, hogy a felhasznált keretrendszerek helyesen működnek. Ennek az ellenőrzése pedig nem, egy az adott technológiát csak felhasználó kliensalkalmazásnak a felelőssége, hanem magának a keretrendszernek és a fejlesztőinek.

## Felület tesztelés

Amellett, hogy az *Xcode* támogatja az egységtesztek létrehozását és futtatását, van lehetőség automatizált felület tesztek létrehozására és futtatására is. Sajnos, mint általában minden fejlesztői környezet az *Xcode* is rendelkezik korlátokkal és hiányosságokkal, ezért a tesztek eredményének elemzéséhez már érdekesebb saját megoldásokhoz kellett folyamodnom. A felület tesztek elkészítéséhez a már említett *XCTest* keretrendszert használtam mivel a *Swift* *Testing* egyelőre csak az egység tesztelést támogatja.

Az automatizált felület tesztek, az egység tesztekkel ellentétben nem a háttérben futó folyamatok, hanem egy futó alkalmazás használatát szimulálják, előre meghatározott események mentén. Ez az jelenti, hogy a felület tesztek az adott tesztesetek által definiált tevékenységeket, például egy gomb megnyomását próbálják szoftveresen előidézni, majd a kiváltott hatás megfelelőségét ellenőrizni.

Ahhoz, hogy létre tudjuk hozni teszteseteket, ismernünk kell a használt és az ellenőrizendő felület elemek azonosítóját, amik általában a rendszer által vannak generálva. Szerencsére az *Xcode* tartalmaz egy olyan funkciót, amivel a tesztek írása triviálissá válik. Ez a funkció a *Record UI Test* elnevezést kapta, és lehetővé teszi azt, hogy a szimulátoron futó alkalmazásban rögzítsük az adott teszthez tartozó eseményeket, mintha ténylegesen használnánk az alkalmazást, és minden eseményhez a fejlesztői környezet kódot generál számunkra a teszt metódusba. Ezután már csak az elért állapot helyességét ellenőrző kód megírását kell megtenni, és elkészült egy teszteset. Mint minden felület teszt, amit az alkalmazáshoz készítettem, az alábbi kódrészletben látható teszteset is ezzel a módszerrel készült.

func test\_addButton\_whenClicked\_shouldShowPopover() throws {

let windowId = "TaskWiseMac.MacDashboardView-1-AppWindow-1"

let appWindow = app.windows[windowId]

appWindow.buttons["plus.app.fill"].click()

let titleTextfield = appWindow.popovers.textFields["Title"]

\_ = titleTextfield.waitForExistence(timeout: 5)

XCTAssertTrue(titleTextfield.exists)

}

Ez a specifikus teszteset a *macOS* alkalmazás főképernyőjén teszteli, hogy a plusz ikonnal ellátott gomb megnyomásának hatására, láthatóvá válik-e a feladat hozzáadására alkalmas, modális ablak. Mint megfigyelhető ez a teszt is az egységteszteknél már bemutatott *Given*-*When-Then* struktúrát követi. Mivel *macOS* esetén egy alkalmazás több ablakban is meg lehet nyitva, először a megfelelő azonosító segítségével megkeressük a teszteléshez használt ablakot, majd végrehajtjuk a gomb megnyomását szimuláló metódust, végül pedig ellenőrizzük, hogy ennek hatására létezik-e a megfelelő nézet.

Észrevehető, hogy a végleges ellenőrzés előtt, előbb várunk addig amíg a keresett nézet nem létezik, vagy egy általunk megadott időkorlátból ki nem fut a teszt. Erre azért van szükség mert az automatizált felület tesztek általában nagyon érzékenyek és merevek, így néha még olyankor is előfordul, hogy hibát jeleznek amikor minden rendben van. Az ilyen tranziens hibák kiküszöbölésére, én két módszert alkalmaztam, amik közül az első a már említett várakozási idő beállítása.

A másik módszer az elbukó tesztek megismétlését hajtja végre, amit a tesztekhez tartozó konfigurációs fájlban lehet beállítani. Ennek segítségével, ha egy teszt valamilyen okból elbukik, akkor a rendszer addig ismétli a futását ameddig sikeres nem lesz, illetve ameddig el nem éri a beállított maximális ismétlési számot. Ha ennek ellenére van olyan teszt, ami elbukik akkor szinte biztosak lehetünk benne, hogy nem csak tranziens hibáról van szó, hanem valamilyen tényleges probléma van az alkalmazás működésében.

A *ViewModel* réteg egység tesztjeihez hasonlóan *UI* tesztek is, a *watchOS* alkalmazást leszámítva, minden releváns platform, minden képernyőjén, gyakorlatilag az összes felhasználó által végezhető eseményhez készültek.

## Tesztelési eredmények

A szoftverek minőségét általában nem lehet kvantitatív módszerekkel meghatározni, azonban az olyan jellegű mérőszámok közül, amik a szoftvereket jellemzik, a kódlefedettséget használják leggyakrabban valamilyen szintű minősítésre. Ez a szám a tesztek által futtatott, vagyis ellenőrzött kódsorok és a teljes kódbázis kódsorainak aránya, százalékosan kifejezve. Nyilván valóan ebből a számból nem lehet következtetni a szoftver tényleges használhatóságára, vagy a funkcionális követelmények teljesítésére, azonban ad egy garanciát arra, hogy a szoftverben, vagy legalábbis annak a tesztelt részében nincsenek rejtett hibák. Emiatt arra törekedtem, hogy az alkalmazáshoz írt tesztek, elérjenek egy az iparban is már elfogadottnak tekinthető lefedettséget.

A teljes kódbázisra tekintve ennek a meghatározása meglehetősen nehéz feladat, mivel a teszteket egyszerre csak egy operációs rendszeren, és egy készüléken tudjuk futtatni az *Xcode* segítségével. Sajnos arra, hogy egyszerre több operációs rendszeren futtassuk a teszteket jelenleg nem létezik megoldás, azonban egy operációs rendszerhez tartozó több eszközön, még ha egy kicsit körülményesen is, de megoldható a közös tesztelés.

Az alkalmazás teljes tesztelését emiatt minimum két külön teszt folyamat futtatásával lehet elérni. Alapvetően ez a szám három lett volna, ha megelégszek az *iOS* operációs rendszerre készült *iPhone* és *iPad* alkalmazás tesztjeinek külön való futtatásával. Ezeken az eszközökön ugyanaz az operációs rendszer fut, így az alkalmazás maga csak a megjelenítési rétegekben különbözik, viszont ezek pont azok a rétegek, amik a legnagyobb mértékben tesztelve vannak. Így fontosnak tartottam, hogy ha van rá lehetőség, akkor a tesztelésük ugyanannak a folyamatnak a részét képezze.

Erre a fejlesztői környezetben, ahol a legtöbbször futtatom a teszteket, közvetlenül nincs lehetőség, azonban egy kis kutatással sikerült találnom egy módszert, amivel áthidalhatom a problémát. Az *Xcode* ugyanis rendelkezik olyan funkcionalitásokkal, amiket magából a fejlesztői környezetből nem lehet elérni, azonban parancssori műveletekkel igen. Ezek közé a műveletek tartozik a tesztek olyan futtatása, ami egy adott operációs rendszeren belül több eszközre is futtatja a teszteket.

Mivel ez a parancs viszonylag hosszú, és futtatásához minden egyes alkalommal meg kellen nyitni a terminált, valamint az eredmények megtekintéséhez egy újabb hosszú parancsot kellene kiadni, készítettem egy olyan parancssori programot, ami ezeket elvégzi helyettem. Ennek van egy olyan előnye is, hogy az ilyen parancssori programok futását rá lehet kötni egy gyorsbillentyű parancsra a fejlesztői eszközön belül. Így az egyetlen különbség az eredeti teszt futtatás és a több eszközre történő futtatás között, az én szempontomból az, hogy egy másik billentyű kombinációt kell lenyomni az elindításhoz.

#!/bin/sh

echo "Setting environment"

cd ~/Developer/TaskWise/TaskWise

echo "Starting iOS tests"

xcodebuild -workspace TaskWise.xcworkspace -scheme TaskWise \

-destination "name=iPhone 16 Pro" \

-destination "name=iPad Pro 13-inch (M4)" \

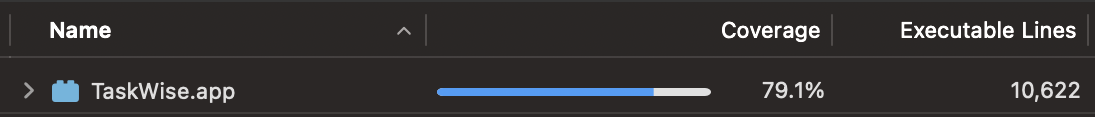
test

echo "Finished testing on iOS, showing results"

open "$PATH\_TO\_LOGS/$(ls -t "$PATH\_TO\_LOGS" | head -n 1)"

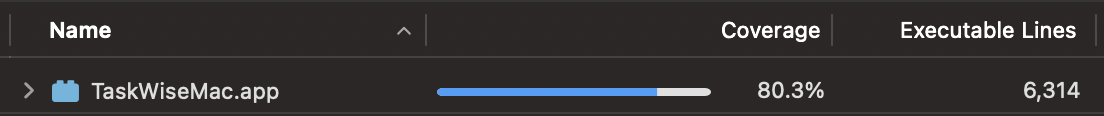
Látható, hogy miután a megnyíló parancssort a megfelelő helyre kell vezérelni, a tesztelést már egy olyan parancs segítségével lehet elindítani, ami közvetlenül az *Xcode* eszközkészletéből dolgozik, azonban itt van lehetőség több teszteszköz megadására is. Az eredményeket a fejlesztői környezetben történő teszteléssel azonos módon egy fájl formájában kapjuk meg, amit az *Xcode* tud értelmezni és emberi olvasásra alkalmas módon megjeleníteni.

Mivel ez a fájl minden teszt futás alkalmával új néven generálódik le, így megnyitása nem lehetséges a fájl nevének programba való beégetésével. A fájl helye a fájlrendszeren belül azonban szerencsére állandó, így ezt a helyet egy környezeti változóként elmentve már meglehet nyitni az adott mappához utoljára hozzáadott fájlt, az utolsó sorban látható parancs segítségével, ami ebben az esetben mindig a generált teszt eredményeket tartalmazó fájl. Az alábbi képen az *iOS* alkalmazás ilyen módszerrel futtatott tesztelésének az eredménye látható.



14. ábra: Az iOS alkalmazáson elért tesztlefedettség

Mivel a több operációs rendszeren való tesztelésre nincs ehhez a módszerhez hasonló megoldás sem, ezért a *macOS* alkalmazás tesztelését az általános módszerrel, közvetlenül a fejlesztői környezetből végeztem. Az alábbi képen ennek az eredménye látható.



15. ábra: A macOS alkalmazáson elért teszt lefedettség

Mint látható mindkét operációs rendszer esetében sikerült egy 80 százalék körüli kódlefedettséget elérnem, ami már az iparban is egy elfogadható értéknek minősül, így megelégedtem az ilyen szintű teszteléssel.

# Összefoglalás és továbbfejlesztési lehetőségek

archivum funkcio

elemzesek-kimutatások

# Köszönetnyilvánítás

Zárásként szeretném megköszönni szüleimnek, Devecseri Anikónak és Dálnoky Bertalannak a folyamatos támogatást, amit tanulmányaim alatt végig nyújtottak.

Illetve szeretném megköszönni nővéremnek Dálnoky Anikónak, a támogatást és lehetőségeket, amik lehetővé tették számomra, hogy *iOS* fejlesztéssel foglalkozzak.

# Irodalomjegyzék

|  |  |
| --- | --- |
| [1] | Apple, [Online]. Available: https://developer.apple.com/documentation/SwiftUI. [Hozzáférés dátuma: 16. május 2024]. |
| [2] | Apple, [Online]. Available: https://developer.apple.com/documentation/uikit. [Hozzáférés dátuma: 23. május 2024]. |
| [3] | Apple, [Online]. Available: https://developer.apple.com/documentation/combine. [Hozzáférés dátuma: 23. május 2024]. |
| [4] | Apple, [Online]. Available: https://developer.apple.com/documentation/foundation/icloud. [Hozzáférés dátuma: 23. május 2024]. |
| [5] | Apple, [Online]. Available: https://developer.apple.com/documentation/cloudkit. [Hozzáférés dátuma: 28. május 2024]. |
| [6] | Apple, [Online]. Available: https://developer.apple.com/documentation/coredata/. [Hozzáférés dátuma: 23. május 2024]. |
| [7] | Apple. [Online]. Available: https://developer.apple.com/documentation/swiftdata. [Hozzáférés dátuma: 28. május 2024]. |
| [8] | Apple, [Online]. Available: https://developer.apple.com/documentation/widgetkit/. [Hozzáférés dátuma: 23. május 2024]. |
| [9] | Realm Inc., [Online]. Available: https://github.com/realm/SwiftLint. [Hozzáférés dátuma: 23. május 2024]. |
| [10] | SwiftGen, [Online]. Available: https://github.com/SwiftGen/SwiftGen. [Hozzáférés dátuma: 23. május 2024]. |
| [11] | K. Fuller. [Online]. Available: https://stencil.fuller.li/en/latest/. [Hozzáférés dátuma: 28. május 2024]. |
| [12] | M. Long. [Online]. Available: https://github.com/hmlongco/Resolver. [Hozzáférés dátuma: 23. május 2024]. |
| [13] | K. Zabłocki. [Online]. Available: https://github.com/krzysztofzablocki/Sourcery. [Hozzáférés dátuma: 23. május 2024]. |
| [14] | CocoaPods, [Online]. Available: https://cocoapods.org. [Hozzáférés dátuma: 23. május 2024]. |
| [15] | F. Laso-Marsetti. [Online]. Available: https://www.kodeco.com/1000705-model-view-controller-mvc-in-ios-a-modern-approach. [Hozzáférés dátuma: 15. október 2024]. |
| [16] | R. Peres. [Online]. Available: https://www.kodeco.com/4161005-%20mvvm-with-combine-tutorial-for-ios. [Hozzáférés dátuma: 15. október 2024]. |
| [17] | M. Katz. [Online]. Available: https://www.kodeco.com/8440907-getting-started-with-the-viper-architecture-pattern. [Hozzáférés dátuma: 15. október 2024]. |
| [18] | Figma, [Online]. Available: https://www.figma.com. [Hozzáférés dátuma: 18. október 2024]. |