FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání diplomové práce

Název:	Aplikace pro Apple Watch určená pro bezpečnější pilotování dronů
Student:	Bc. Petr Dušek
Vedoucí:	Ing. Lukáš Brchl
Studijní program:	Informatika
Obor / specializace:	Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2022/2023

Pokyny pro vypracování

Při létání s dronem vidí pilot pouze pozici svého dronu v kontrolní aplikaci. Nemá však přehled o tom, jestli se v okolí nepohybují další drony nebo letadla. Proto nyní existují aplikace třetích stran, která data o dalších letounech sdružují a poskytují je ostatním. Cílem práce je návrh a implementace aplikace pro Apple Watch, která bude sloužit pilotům dronů pro jejich bezpečnější pohyb napříč vzdušným prostorem. Aplikace by měla poskytovat kontextuální "flight radar" ostatních dronů a bude je upozorňovat na potenciální hrozby formou notifikací.

- Proveďte rešerši existujících aplikací pro chytré hodinky, která jsou relevantní k danému tématu.
- Na základě rešerše navrhněte vhodné uživatelské rozhraní cílové aplikace.
- Navrhněte taktéž architekturu dané aplikace, její způsob komunikace s okolním světem a aplikaci implementujte.
- Aplikaci otestujte s reálnými piloty dronu a získejte od nich zpětnou vazbu.
- Proveďte zhodnocení dosažených výsledků a navrhněte budoucí rozšíření.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 10. února 2022 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Aplikace pro Apple Watch určená pro bezpečnější pilotování dronů

Bc. Petr Dušek

Katedra softwarového inženýrství
Vedoucí práce: Ing. Lukáš Brchl

1. května 2022

Poděkování

Chtěl bych tímto poděkovat vedoucímu Ing. Lukášovi Brchlovi za skvělý přístup, cenné rady a připomínky během tvorby této práce. Dále bych rád poděkoval Ing. Mariánovi Hlaváčovi za pomoc při implementaci a vysvětlení funkcí webové služby, která byla v rámci práce použita. Nakonec bych chtěl poděkovat celé své rodině a přítelkyni za finanční, psychickou i morální podporu po celou dobu studia na vysoké škole.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. Dále prohlašuji, že jsem s Českým vysokým učením technickým v Praze uzavřel licenční smlouvu o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona. Tato skutečnost nemá vliv na ust. § 47b zákona č. 111/1998 Sb., o vysokých školách, ve znění pozdějších předpisů.

V Praze dne 1. května 2022 . . . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií

Odkaz na tuto práci

Dušek, Petr. *Aplikace pro Apple Watch určená pro bezpečnější pilotování dronů.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2022.

Abstrakt

Tato diplomová práce popisuje návrh a implementaci aplikace pro chytré hodinky Apple Watch. Tato aplikace bude sloužit pilotům dronů k lepší orientaci ve vzdušném prostoru. Součástí práce je jednak rešerše podobných aplikací na systém *WatchOS*, jednak specifikace požadavků, případů užití a návrh grafického rozhraní. Taktéž je popsáno testování grafického rozhraní s účastníky, na jehož základě vznikla výsledná aplikace. Součástí realizace je návrh architektury aplikace pro systém *WatchOS*, komunikace aplikace s přidruženou mobilní aplikací, popis webové služby využívané aplikací a také popis implementačních detailů. Nakonec je celé řešení otestováno s několika účastníky pohybujícími se mimo obor letectví a taktéž s účastníky, kteří v letectví pracují.

Klíčová slova vývoj aplikací pro chytré hodinky, Apple Watch, WatchOS, SwiftUI, MVVM architektura, čistá architektura, WebSocket, Flutter

Abstract

This thesis describes analysis, design and implementation of application for Apple Watch smart watches. This application will help drone pilots with better orientation in airspace. The thesis also includes research of other similar applications for WatchOS operating system, requirements and use cases specification and design proposal. Design proposal was tested with several testers, who helped with final appearance of application. Realization part includes architectural design proposal for *WatchOS*, communication with associated mobile application, describtion of web service and implementation details. The whole solution is tested with several testers and also with people who work in flight industry.

Keywords smart watch app development, Apple Watch, WatchOS, SwiftUI, MVVM architecture, clean architecture, WebSocket, Flutter

Obsah

Seznam obrázků

Seznam tabulek

Úvod

V dnešní době se pilotování dronů stává čím dál populárnějším, a to nejen mezi profesionály, ale i laiky, kteří využívají drony pro pořizování fotek či videí na dovolené, nebo při sportování. To ovšem přináší množství rizik, kterým se snažíme předejít. Jedná se například o monitorování vzdušného prostoru a informování pilota o hrozbách, letových zónách a okolních letadlech.

Při létání s dronem uživatelé většinou využívají mobilní aplikace třetích stran, ve kterých mohou sledovat živý přenos z jejich zařízení. V danou chvíli však nemohou používat zároveň jiné aplikace, jako je například aplikace *Dronetag*, která slouží právě k zobrazování letadel ve vzdušném prostoru a umožňuje uživatelům mít přehled o tom, co se v jejich okolí děje.

Lidé také v dnešní době začínají čím dál více používat chytrá nositelná zařízení, jako například chytré hodinky. Tato zařízení by bylo možno využít jako rozšíření mobilní aplikace a nabídnout uživatelům dodatečný pohled na to, co se ve chvíli, kdy létají s drony děje ve vzdušném prostoru.

Další motivací pro vznik této aplikace jsou nová pravidla pro drony, která by měla přijít v platnost. Jedná se o dálkovou identifikaci dronů, jež bude povinnou výbavou u všech nových dronů s identifikačním štítkem třídy dronu C (C0-C6) [3]. Kategorie dronů jsou podrobněji popsané na stránkách Úřadu pro civilní letectví, viz [4]. Původně měla být dálková identifikace povinná pro lety ve specifické kategorii od 1. 7. 2022, tento termín sice vypadá, že se nestihne, avšak víme, že v budoucnu tato legislativa vyjde v platnost. Společnost *Dronetag s.r.o.* tedy vyvíjí zařízení, která zmíněnou dálkovou identifikaci implementují, a uživatelé si mohou tato zařízení připnout na svůj dron, a rozšířit je o tuto funkcionalitu.

Vzniklá aplikace by měla s těmito zařízeními umět komunikovat a poskytovat uživatelům získané informace o poloze svého, nebo ostatních dronů. Navíc díky tomu, že se jedná o aplikaci na chytré hodinky, nebude muset uživatel zavírat mobilní aplikaci, ve které sleduje živý přenos z kamery svého dronu. Bude si tak moct zobrazit radar na svých hodinkách a rychleji se zorientovat

v prostoru v případě hrozícího nebezpečí.

Cíl práce

Cílem této práce je zanalyzovat dostupné aplikace pro chytré hodinky a provést rešerši existujících aplikací, které se zabývají podobnou problematikou.

Následně na základě této rešerše navrhnout uživatelské rozhraní cílové aplikace a tento návrh patřičně otestovat s několika účastníky.

Dalším cílem je navrhnout architekturu dané aplikace a způsob, kterým bude získávat data o letadlech, která bude uživatelům zobrazovat.

Hlavní součásti bude také samotná implementace aplikace pro chytré hodinky Apple Watch, jež bude sloužit pilotům dronů pro bezpečnější pohyb ve vzdušném prostoru a bude je upozorňovat na potenciální hrozby formou notifikací.

Nakonec bude potřeba vzniklé řešení otestovat a to jak s lidmi, kteří nemají s létáním zkušenosti, tak i s lidmi, kteří se pohybují v oboru letectví a mají tak bližší pohled do této problematiky.

Kapitola 1

Rešerše aplikací pro Apple Watch

Chytré hodinky jsou stále poměrně novým zařízením na trhu a zájem o ně se každým rokem zvyšuje. Podle článku [5] se kvůli pandemii covid-19 v minulém roce zrychlil růst o 12 % a udává, že za rok 2021 se počet lidí vlastnících chytré hodinky zvýšil o téměř 30 %. Ve Spojených státech amerických zůstávají nejpopulárnějšími hodinkami na trhu Apple Watch od firmy *Apple Inc.*, které představují 63 % prodejů v posledním kvartálu roku 2021.

Dle [5] tři ze čtyř kupujících využívají chytré hodinky ke sledování své kondice nebo zdraví. Převážně se jedná o sledování počtu kroků, spánku a celkové aktivity za den.

Na hodinkách Apple Watch je možné mít aplikaci buď to přidruženou k mobilní aplikaci, nebo si můžeme od verze watchOS 6 stahovat aplikace samo-

Obrázek 1.1: Prvky na hlavní ploše, zdroj: [1]

statné, nezávislé na mobilní aplikaci [6]. Vytvořená aplikace také může podporovat tvorbu widgetů přímo na hlavní plochu hodinek, jak můžeme vidět na obrázku 1.1. Společnost *Apple Inc.* nedovoluje stahování aplikací jinou než oficiální cestou, a to přes svůj obchod *App Store*, který nabízí mnoho různých aplikací, z nichž se převážná většina zabývá sledováním zdraví, cvičení, monitorováním spánku, ale narazíme i na jednoduché hry, zápisníky a další zajímavé kategorie. Vzhledem k omezené velikosti displeje a limitací ovládání těchto zařízení je ovšem nabídka výrazně menší, než je tomu například u mobilních telefonů.

Jako vývojáři musíme pečlivě zvážit, zda má aplikace pro tuto platformu smysl, a také si dát pozor při jejím návrhu a tvorbě designu. Hlavním cílem je, aby byla aplikace pro uživatele srozumitelná a jednoduchá na ovládání.

Abychom se inspirovali a zjistili, které prvky se nejčastěji u těchto typů aplikací používají, provedli jsme rešerši. Na jejím základě bylo vybráno několik aplikací, které by nám pomohly porozumět aktuálnímu trhu. Nejzajímavějšími aplikacemi byly následující: *CARROT* [7], *Radar Game* [8], *Flightradar24* [9] a *PlaneWatcher* [10].

1.1 CARROT

Nejdříve se podíváme na aplikaci CARROT, která se sice zabývá předpovědí počasí, ale obsahuje množství zajímavých prvků a poslouží tak k lepšímu porozumění, co vše je možné na chytrých hodinkách vytvářet. Díky tomu jsme tuto aplikaci vybrali do rešerše i přesto, že nemá s letectvím nic společného. Zajímavými prvky jsou hlavně obrazovka výběru nebo onboarding. Snímky obrazovek jsou vidět na obrázku 1.2.

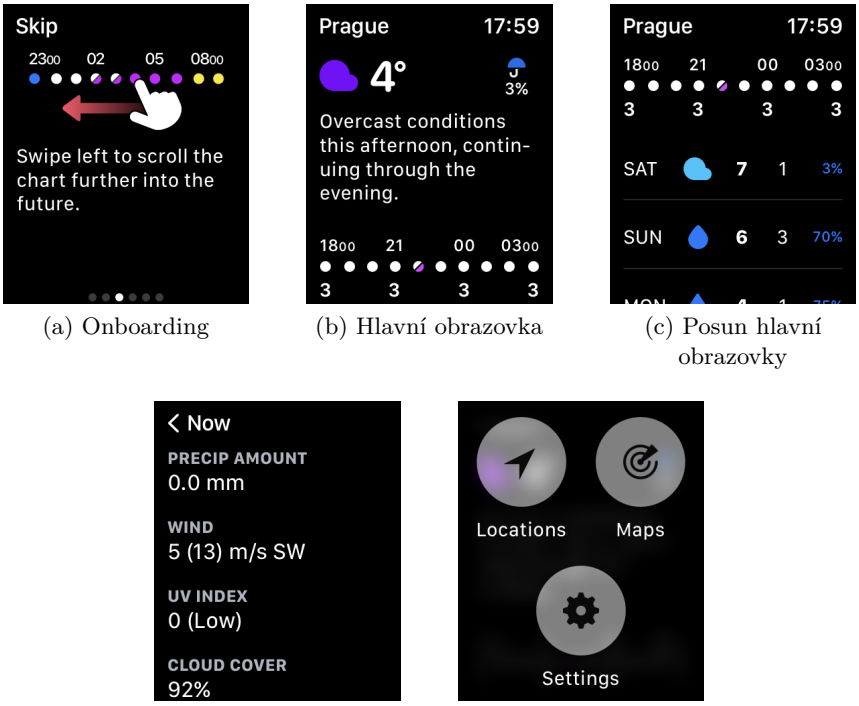
Při prvním spuštění vidíme onboarding. Jedná se o obrazovku, která uživateli krátce vysvětlí, jak má s aplikací pracovat. U jiných aplikací pro chytré hodinky jsme se zatím s tímto prvkem nesetkali, ovšem pokud máme v aplikaci nějaká gesta, která nemusí uživatele napadnout, je to určitě chytré rešení.

Hlavní obrazovka ukazuje počasí na aktuálním místě, kde se nacházíme. Pomocí digitální korunky [1] nebo pomocí dotyku se můžeme pohybovat směrem nahoru nebo dolů a zobrazovat tak předpověď pro následující dny. Máme také možnost na jednotlivé prvky klikat pomocí dotykového ovládání a zobrazit si detail s podrobnějšími informacemi. Vidíme zde například rychlost větru, UV index, pocitovou teplotu a další informace spojené s počasím.

Zajímavým prvkem je zobrazení nabídky, které spustíme delším podržením prstu na obrazovce viz 1.2e. Co bychom jako uživatelé uvítali, je zmínka o této funkcionalitě, jelikož se o ní nedozvíme ani při onboardingu. Z nabídky si můžeme změnit výběr lokace, zobrazit mapu (mapa je dostupná pouze v premium verzi), nebo se přesunout do nastavení aplikace.

[1] jedná se o hardwarový prvek, nacházející se na straně chytrých hodinek

CARROT také poskytuje několik widgetů, které si uživatel může zobrazit přímo na hlavní ploše svých chytrých hodinek, uvidí tak aktuální informace o počasí při pouhém pohledu na hodinky.



(a) Onboarding (b) Hlavní obrazovka (c) Posun hlavní obrazovky

(d) Detail (e) Nabídka

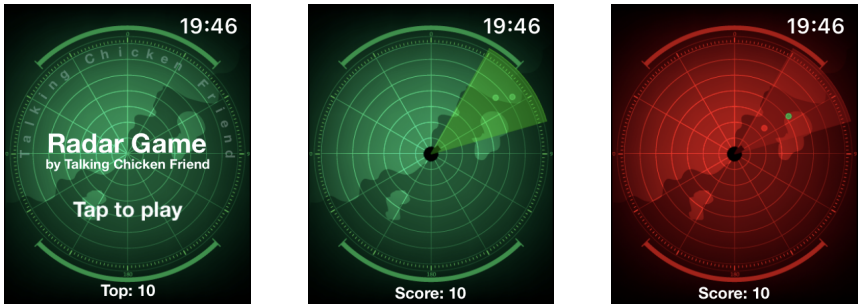
Obrázek 1.2: Aplikace CARROT

1.2 Radar Game

Radar Game je jedna z her dostupných pro *Apple Watch*. Hlavním důvodem, proč byla vybrána v rámci rešerše, je prvek radaru, který bude obsažen i v rámci vyvíjené aplikace. Hra má pouze dvě scény: hlavní menu a scénu, ve které hrajeme hru, snímky pořízené z aplikace vidíme na obrázku 1.3.

Hlavní scéna pouze zobrazuje pořadí v žebříčku a po kliknutí začínáme hrát. Herní systém je velice přímočarý. Vidíme radar, který má ve středu dělo, okolo něhož létají prvky, které se přibližují. Úkolem je sestřelit prvky dříve, než se dostanou do středu. Jakmile se dostanou do středu, hra končí.

Na radaru vždy vidíme pouze prvky ve směru, kterým je kanón natočen. Směr pohledu můžeme měnit pomocí digitální korunky a při klepnutí prstem střílíme. Jakmile je některý prvek v blízkosti středu, radar zčervená a upozorní tak uživatele na hrozící nebezpečí.



(a) Menu (b) Normální stav (c) Zobrazení nebezpečí

Obrázek 1.3: Aplikace Radar Game

1.3 FlightRadar24

FlightRadar24 je aplikace, která umožňuje sledovat probíhající lety a získávat o nich informace. Při spuštění vidíme hlavní menu, kde máme na výběr ze dvou možností: zobrazení letů v okolí a sledování konkrétního letu.

Pokud si vybereme lety v okolí, aplikace nám zobrazí seznam, který na prvním místě obsahuje tlačítko pro zobrazení mapy a následně seznam letů. Položka s letem nese informace o vzdálenosti od uživatele, výšce letadla, rychlosti a další, jak vidíme na obrázku 1.4b

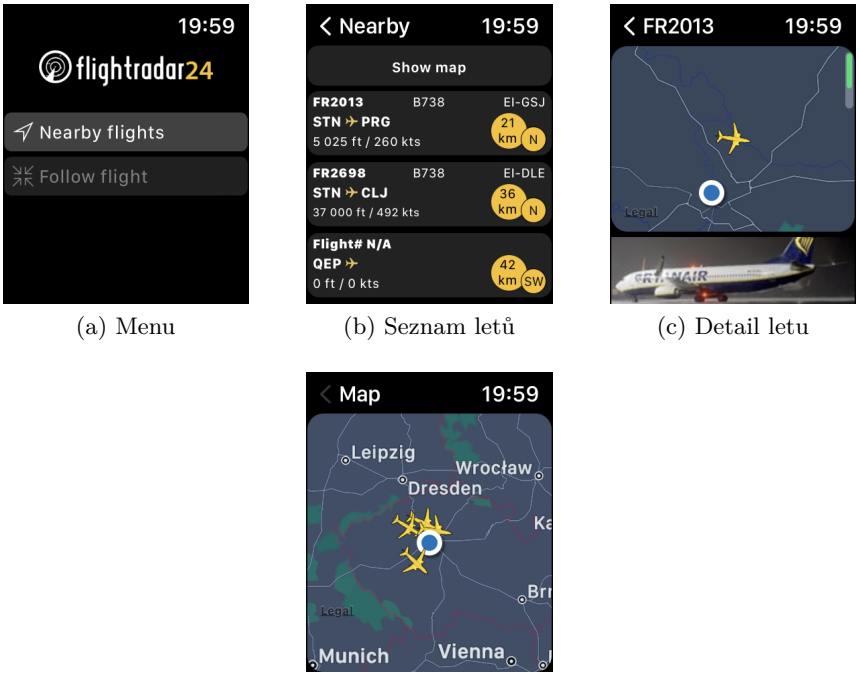
Po kliknutí na konkrétní let se nám otevře detail, ve kterém zabírá většinu plochy mapa zobrazující pozici letadla a pozici uživatele. Pod mapou vidíme obrázek s typem letadla a jeho název. Naopak pokud uživatel klikne na položku mapy v seznamu, vidí svou pozici a všechna letadla v okolí.

Druhou možností z hlavní nabídky je sledování určitého letu. Ovšem tuto možnost nelze pomocí hodinek plně ovládat a po vybrání nás aplikace pouze informuje o tom, že pro sledování musíme let vybrat na telefonu.

Aplikace nejspíše plní svůj účel, na druhou stranu bychom uvítali možnost posouvat se po mapě, přibližovat ji, nastavit si například filtry a různé jednotky vzdálenosti. Tyto možnosti ovšem v aplikaci nejsou. Dále také nevidíme, jakým směrem se uživatel dívá, a nevíme tedy přesně, jakým směrem bychom mohli letadlo na obloze spatřit. Snímky pořízené z aplikace vidíme na obrázku 1.4.

1.4 PlaneWatcher

PlaneWatcher je placená aplikace, která je dostupná samostatně pouze pro chytré hodinky *Apple Watch*. Hlavní funkcí aplikace je funkce radaru, na kterém vidíme pozici uživatele, okolní letiště a různé typy letadel pohybující se v okolí. Aplikace nezobrazuje pouze letadla, ale také například helikoptéry a balóny.



(a) Menu (b) Seznam letů (c) Detail letu

(d) Mapa

Obrázek 1.4: Aplikace Flightradar24

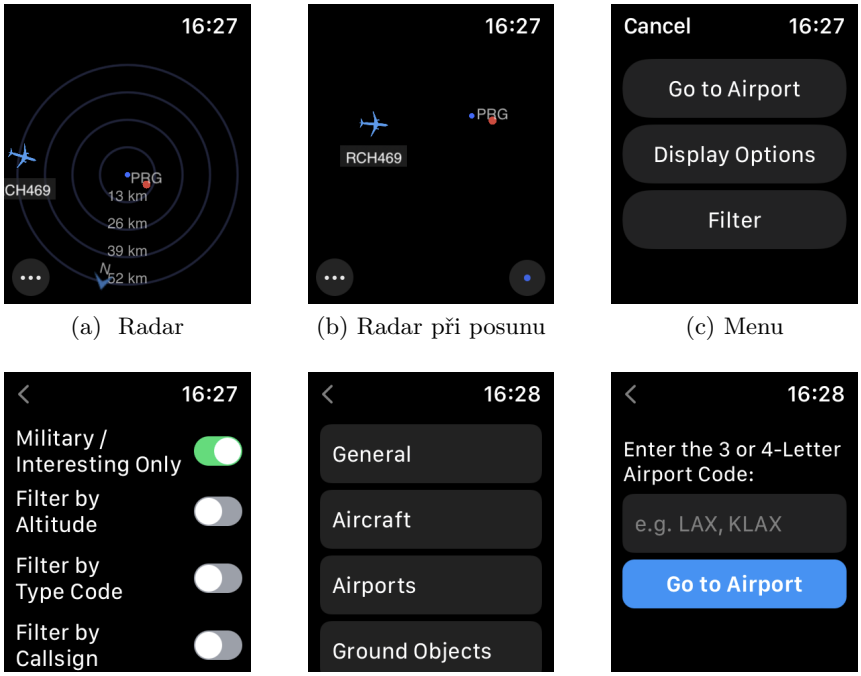
Ve výchozí pozici je uživatel zobrazen ve středu radaru. Pomocí digitální korunky na straně chytrých hodinek můžeme měnit vzdálenost mezi jednotlivými kruhy, které jsou vidět na obrázku 1.5a. Další možností je posun po radaru. Pomocí dotykového ovládání je možné posunout radar a vidět tak letouny, které jsou více vzdálené. V tomto případě zmizí kruhy zobrazující vzdálenost a na pravém dolním okraji obrazovky se ukáže tlačítko pro vycentrování zpět na uživatele.

Jednotlivá letadla zobrazená na radaru máme možnost vybrat pomocí dotykového ovládání tím, že na ně klikneme. Po kliknutí se nám otevře detail zařízení s dodatečnými informacemi o poloze, rychlosti, názvu letadla atd. V případě, že se na vybraném místě nachází více letadel, aplikace nám zobrazí seznam všech letadel na této pozici, ze kterého si následně vybereme to, které chceme zobrazit.

V levé části obrazovky vidíme tlačítko, které při vybrání vede na obrazovku menu. Menu obsahuje tři prvky: přesun na letiště, volby zobrazení a filtry.

Přesun na letiště zobrazí obrazovku, na které uživatel vyplní kód letiště a aplikace mu následně zobrazí radar vycentrovaný na zadané letiště.

Volby zobrazení umožňují skrýt si určité prvky na radaru, jedná se například o zobrazení popisků letounů, zobrazení kruhů vzdáleností, změnu použité jednotky pro vzdálenost atd. Můžeme si také vybrat, zda chceme zobrazovat



(a) Radar (b) Radar při posunu (c) Menu

(d) Filtry (e) Volby zobrazení (f) Přesun na letiště

Obrázek 1.5: Aplikace PlaneWatcher

pouze letadla, nebo i další zmíněné typy letounů.

Scéna s filtry nám naopak umožňuje navolit si parametry pro maximální výšku, filtrovat letouny podle kódu nebo podle registrace. Veškeré zadávání textu probíhá buďto přes hlasové ovládání, nebo pomocí klávesnice na mobilním telefonu.

Můžeme také letadlo sledovat. Tato akce zajistí to, že se nám vybrané letadlo vycentruje a je vždy ve středu radaru.

Tato aplikace je oproti aplikaci *Flightradar24* použitelnější, nabízí velké množství filtrů a její ovládání je velice intuitivní. Má také velmi dobré hodnocení v obchodě *App Store*, což může být dané i tím, že se jedná o jedinou opravdu použitelnou aplikaci tohoto typu.

Některé chování je ovšem lehce matoucí, a to například mizení kruhů při posunu radaru. Uživatel při posunu ztrácí pojem o tom, kde se nachází, jelikož pozadí je pouze tmavé a můžeme tak lehce ztratit orientaci. Naopak oproti zmíněnému *Flightradaru24* vidíme například i indikaci světových stran a radar se dokáže i otáčet směrem, kterým se díváme. Snímky z aplikace vidíme na obrázku 1.5

Kapitola 2

Analýza a návrh

Tato kapitola se zabývá analýzou a návrhem zmíněné aplikace pro *Apple Watch*. Analýza zahrnuje definice požadavků a popis případů užití. Je také zahrnutý návrh uživatelského rozhraní a jeho otestování, které bylo provedeno s několika vybranými uživateli.

Aplikace pro chytré hodinky většinou přináší rozšíření mobilní aplikace a snaží se uživateli zobrazovat zjednodušený pohled. V aktuální chvíli aplikace *Dronetag* poskytuje velké množství funkcí a uživatel pomocí této aplikace dokáže ovládat veškeré funkce, aniž by musel využít webové aplikace. Na chytrých hodinkách chceme uživateli poskytnout pouze část funkcí, kterými mobilní aplikace disponuje, jelikož je ovládání na takto malém displeji limitující. Je tedy očekávané, že složitější a zdlouhavější akce uživatel vykoná v mobilní aplikaci a chytré hodinky mu budou sloužit ve chvíli, kdy například poletí se svým dronem a bude potřebovat mít povědomí o tom, co se děje okolo něj.

Nejdůležitější částí by měl být radar, který bude zobrazovat letadla v okolí uživatele. Aplikace by měla být přehledná a na první pohled by mělo být jasné, co které prvky zobrazené na radaru znamenají. U aplikací pro chytré hodinky se neočekává, že by v nich uživatel trávil velké množství času, a spíše se snaží, aby předané informace byly srozumitelné a rychle pochopitelné.

2.1 Definice požadavků

Jednou z hlavních částí analýzy je sestavení požadavků. Požadavky jsou definovány podle funkcí, které si musíme stanovit a vymezit tím hranice systému. Se společností *Dronetag s.r.o.* jsme na základě požadavků jejich uživatelů sestavili následující seznam funkčních a nefunkčních požadavků, které by měla finální aplikace pokrýt.

2.1.1 Funkční požadavky

Funkční požadavky přímo popisují chování aplikace.

- **F1: Přihlášení**

  Aplikace bude umožňovat přihlásit uživatele.

- **F2: Odhlášení**

  Aplikace bude schopná odhlásit uživatele.

- **F3: Onboarding**

  Aplikace bude při prvním spuštění zobrazovat onboarding s tipy ohledně dostupných funkcí.

- **F4: Pozice uživatele**

  Aplikace bude zobrazovat pozici uživatele.

- **F5: Pozice dronu**

  Pokud bude probíhat let s některým z dronů, které uživatel vlastní, aplikace bude zobrazovat pozici tohoto dronu.

- **F6: Letová zóna**

  Při probíhajícím letu bude aplikace zobrazovat hranice letové zóny pro probíhající let.

- **F7: Okolní letadla**

  Aplikace bude zobrazovat veškerá okolní letadla, o kterých získá informace z API.

- **F8: Signalizace ohrožujícího letadla**

  Aplikace bude signalizovat letadla v nebezpečné vzdálenosti od uživatelova dronu. Signalizace bude vizuální, zvuková a také pomocí vibrací.

- **F9: Notifikace**

  Aplikace bude informovat o hrozícím nebezpečí formou notifikací.

- **F10: Změna centrování**

  Aplikace bude umožňovat přepínat centrování mezi pozicí dronu a pozicí uživatele.

- **F11: Zobrazení měřítka**

  Aplikace bude schopna zobrazovat nastavené měřítko radaru.

- **F12: Nastavení měřítka**

  Aplikace bude umožňovat měnit si nastavené měřítko z předem definovaných hodnot.

- **F13: Pohyb po radaru**

  Uživatel bude schopný se pohybovat po radaru pomocí dotykového ovládání.

- **F14: Zobrazení informací o letadle**

  Uživatel si bude moct zobrazit detailní informace o vybraném letadle.

- **F15: Úprava nastavení**

  Uživatel bude schopen nastavit zobrazení vizuálních prvků, případně povolit nebo zakázat některá výchozí chování.

2.1.2 Nefunkční požadavky

Nefunkční požadavky určují omezení kladená na systém a mají zásadní dopad na návrh architektury.

- **N1: Aplikace pro chytré hodinky**

  Aplikace bude implementována jako nativní aplikace běžící na operačním systému *WatchOS*.

- **N2: Podpora velikostí displeje**

  Aplikace by měla podporovat všechny dostupné velikosti displeje u hodinek *Apple Watch*.

- **N3: Programovací jazyk**

  Aplikace by měla být napsána v programovacím jazyce *Swift*.

- **N4: Propojení s mobilním zařízením**

  Aplikace bude schopná získávat potřebná data z přidružené mobilní aplikace.

- **N5: Získání dat z webové služby**

  Aplikace bude schopná získávat data z webové služby pomocí Representational State Transfer (REST) API nebo protokolu *Websocket*.

- **N6: Monitoring chyb**

  Aplikace bude posílat a ukládat data o pádech za běhu.

- **N7: Verze systému**

  Aplikace by měla podporovat operační systém WatchOS verze 8 a vyšší.

- **N8: Jazyk aplikace**

  Aplikace by měla být v anglickém jazyce.

2.2 Případy užití

Případy užití slouží k detailnější specifikaci funkčních požadavků. Funkční požadavky se typicky rozkládají na několik případů užití. Všechny funkční požadavky jsou pokryty, viz tabulky 2.1 a 2.2. Podrobný popis případů užití je popsán níže nebo je vidět ve formě diagramu na obrázku 2.1.

- **UC1: Přihlásit se**

  Umožňuje uživateli získat údaje z přidružené mobilní aplikace nebo se přihlásit nezávisle na mobilní aplikaci.

  Hlavní scénář

  1. Případ užití začíná, jestliže je uživatel přihlášen v přidružené mobilní aplikaci.
  2. Uživatel otevře aplikaci na chytrých hodinkách.
  3. Aplikace na pozadí získá údaje o přihlášeném uživateli z přidružené mobilní aplikace.

  Alternativní scénář 1

  1. Případ užití začíná ve chvíli, kdy uživatel není přihlášen na mobilní aplikaci.
  2. Nepřihlášený uživatel otevře aplikaci na chytrých hodinkách.
  3. Aplikace se na pozadí pokusí získat údaje o přihlášeném uživateli.
  4. Požadavek na data o přihlášeném uživateli selže.
  5. Uživatel se následně přihlásí na mobilní aplikaci a stiskne tlačítko *Retry*.
  6. Aplikace získá údaje o aktuálním uživateli.

  Alternativní scénář 2

  1. Případ užití začíná ve chvíli, kdy má uživatel vypnutou synchronizaci s mobilní aplikací.
  2. Nepřihlášený uživatel otevře aplikaci na chytrých hodinkách.
  3. Aplikace uživateli zobrazí obrazovku s přihlášením.
  4. Uživatel zadá uživatelské jméno a heslo.
  5. Uživatel klikne na tlačítko *Sign in*.
  6. Aplikace získá data z API a přihlásí uživatele.

- **UC2: Odhlásit se**

  Přihlášený uživatel se může odhlásit z aplikace.

  1. Případ užití začíná, jestliže je uživatel přihlášen a má zobrazenou obrazovku s radarem.
  2. Uživatel podrží prst na obrazovce.
  3. Systém zobrazí uživateli obrazovku s profilem.
  4. Uživatel vybere položku nastavení.
  5. Systém zobrazí uživateli obrazovku nastavení.
  6. Uživatel klikne na tlačítko *Log out*.
  7. Systém uživatele odhlásí.

- **UC3: Zobrazit uživatele chytrých hodinek**

  Aplikace zobrazí pozici uživatele na radaru [2].

  Hlavní scénář

  1. Případ užití začíná, jestliže je uživatel přihlášen.
  2. Uživatel otevře aplikaci.
  3. Aplikace zobrazí uživateli radar, na kterém je viditelný prvek indikující jeho pozici.

  Alternativní scénář

  1. Případ užití začíná, jestliže je uživatel přihlášen a nemá povolená práva k poloze.
  2. Uživatel otevře aplikaci.
  3. Aplikace zobrazí uživateli dialog informující ho tom, že je nutné povolit pozici v nastavení.
  4. Uživatel přejde do nastavení a povolí aplikaci získání lokace.
  5. Systém skryje dialog a zobrazí uživateli radar s jeho pozicí.

[2] jedná se pouze o pozici aktuálního uživatele aplikace, ostatní uživatelé se nezobrazují

- **UC4: Zobrazit dron**

  Aplikace zobrazí pozici vlastního dronu v případě, že probíhá let s některým ze zařízení, která patří uživateli.

  1. Případ užití začíná, jestliže je uživatel přihlášen a probíhá let s některým z jeho zařízení.
  2. Uživatel otevře aplikaci.
  3. Systém zobrazí radar.
  4. Radar zobrazuje prvek indikující pozici dronu.

- **UC5: Zobrazit okolní letadla**

  Aplikace zobrazí pozice okolních letadel [3].

  1. Případ užití začíná, jestliže je uživatel přihlášen a v jeho okolí létají cizí letadla.
  2. Uživatel otevře aplikaci.
  3. Systém zobrazí radar.
  4. Radar indikuje veškerá letadla v okolí získaná z REST API.

- **UC6: Zobrazit indikaci hrozícího nebezpečí**

  Systém indikuje hrozící nebezpečí v případě přiblížení okolních letadel k vlastnímu zařízení na definovanou vzdálenost.

  1. Případ užití začíná, jestliže je uživatel přihlášen, letí se svým dronem a v jeho okolí létají cizí letadla.
  2. Uživatel otevře aplikaci.
  3. Radar indikuje veškeré letadla v okolí získaná z REST API.
  4. V případě, že se některý z okolních letadel přiblíží k pozici vlastního dronu na vzdálenost nižší, než je vzdálenost definovaná uživatelem, aplikace zvýrazní tento dron pomocí barevné animace a začne vydávat zvuky a vibrace.

[3] v tuto chvíli se jedná pouze o drony, které spadají do kategorie letadel

- **UC7: Zobrazit letovou zónu**

  Aplikace zobrazí letovou zónu pro probíhající let.

  1. Přihlášený uživatel, který letí s dronem, otevře aplikaci.
  2. Aplikace zobrazí pozici a hranice letové zóny pro probíhající let.

- **UC8: Zobrazit měřítko**

  Aplikace zobrazí aktuální měřítko nastavené uživatelem.

  1. Přihlášený uživatel otevře aplikaci.
  2. Aplikace zobrazí radar, na kterém jsou viditelné kruhy indikující vzdálenost.
  3. Systém informuje uživatele o vzdálenosti mezi jednotlivými kruhy podle aktuální hodnoty měřítka.

- **UC9: Změnit měřítko**

  Umožňuje uživateli měnit aktuální měřítko radaru.

  1. Přihlášený uživatel otevře aplikaci na hlavní obrazovce s radarem.
  2. Přihlášený uživatel pomocí digitální korunky změní měřítko.
  3. Systém aktualizuje hodnoty vzdálenosti mezi kruhy a přepočítá pozice zobrazených prvků.

- **UC10: Přepnout centrování**

  Umožňuje uživateli přepínat centrování mezi uživatelem a vlastním dronem.

  Hlavní scénář

  1. Případ užití začíná ve chvíli, kdy je uživatel přihlášen, letí s dronem a v minulosti si zobrazil onboarding.
  2. Přihlášený uživatel otevře aplikaci.
  3. Aplikace zobrazí hlavní obrazovku s radarem.
  4. Přihlášený uživatel stiskne tlačítko pro vycentrování na dron.
  5. Radar vycentruje na dron a sleduje jeho pozici.

  Alternativní scénář

  1. Případ užití začíná ve chvíli, kdy je uživatel přihlášen a má otevřený radar vycentrovaný na pozici dronu.
  2. Přihlášený uživatel stiskne tlačítko pro vycentrování na uživatele.
  3. Radar vycentruje na pozici uživatele a sleduje jeho pozici.

- **UC11: Posunout radar**

  Uživatel je schopen posouvat se po radaru pomocí dotykového ovládání posunem prstu po obrazovce.

  1. Přihlášený uživatel otevře aplikaci na hlavní obrazovce s radarem.
  2. Aplikace je vycentrována na uživatele.
  3. Uživatel se pomocí dotykového ovládání posune prstem po radaru.
  4. Aplikace zruší centrování na uživatele a posune pozici radaru.

- **UC12: Zobrazit detail dronu**

  Systém umožňuje otevřít si detail vybraného letadla a zobrazit dodatečné informace týkající se vybraného prvku.

  1. Případ užití začíná ve chvíli, kdy je uživatel přihlášen a na radaru vidí drony v okolí.
  2. Uživatel pomocí dotykového ovládání klikne na jeden z dronů zobrazených na obrazovce.
  3. Aplikace otevře detail vybraného dronu a zobrazí uživateli dodatečné informace o jeho poloze, směru letu a dalších potřebné charakteristiky.

- **UC13: Zobrazit notifikace**

  Aplikace bude zobrazovat notifikace v případě hrozícího nebezpečí.

  1. Případ užití začíná ve chvíli, kdy je uživatel přihlášen a má povolené notifikace.
  2. V případě hrozícího nebezpečí aplikaci informuje uživatele pomocí push notifikace.

- **UC14: Přepnout synchronizaci**

  Uživatel bude moci zapnout nebo vypnout synchronizaci přihlášeného uživatele z přidružené mobilní aplikace.

  1. Případ užití začíná ve chvíli, kdy je uživatel přihlášen.
  2. Uživatel otevře aplikaci a přejde na obrazovku nastavení.
  3. V nastavení si kliknutím přepne synchronizaci dle svých preferencí.

- **UC15: Skrýt/zobrazit vizuální prvky**

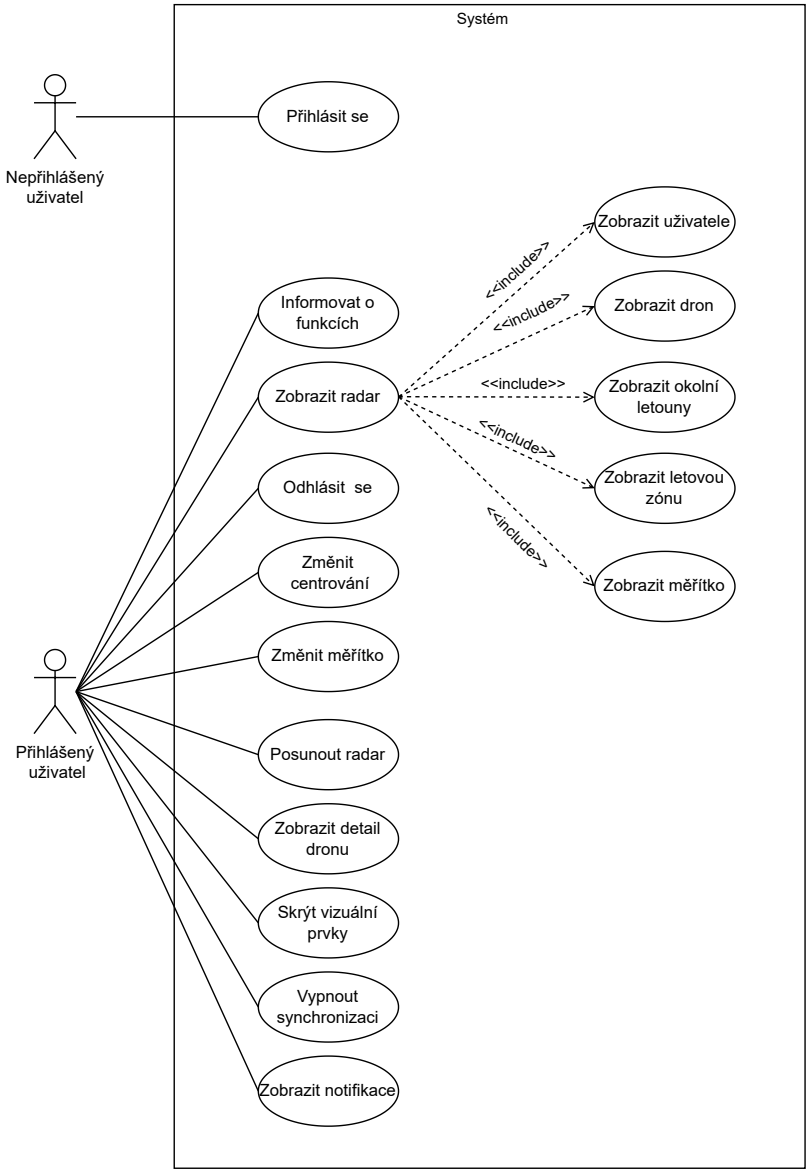
  Aplikace umožňuje skrývat vizuální prvky na radaru podle preferencí uživatele.

  1. Případ užití začíná ve chvíli, kdy je uživatel přihlášen.
  2. Uživatel otevře aplikaci a přejde pomocí podržení prstu na obrazovce do profilu.
  3. Na obrazovce profilu si uživatel vybere položku *Visuals*.
  4. Na obrazovce s vizuály si dle svých preferencí nastaví viditelnost prvků.

- **UC16: Informovat o funkcích**

  Aplikace bude při spuštění informovat uživatele o všech důležitých funkcích, které poskytuje.

  1. Případ užití začíná ve chvíli, kdy je uživatel přihlášen a v minulosti neotevřel aplikaci.
  2. Uživatel otevře aplikaci.
  3. Systém uživateli zobrazí posloupnost tipů týkajících se funkcí, které může uživatel využívat.
  4. Uživatel postupně projde veškeré tipy a klikne na tlačítko *Finish*
  5. Systém zobrazí uživateli obrazovku radaru.



Obrázek 2.1: Diagram případů užití

Tabulka 2.1: Tabulka pokrytí funkčních požadavků – část 1

	UC1	UC2	UC3	UC4	UC5	UC6	UC7	UC8
F1	X							
F2		X						
F4			X					
F5				X				
F6							X	
F7					X			
F8						X		
F11								X

Tabulka 2.2: Tabulka pokrytí funkčních požadavků – část 2

	UC9	UC10	UC11	UC12	UC13	UC14	UC15	UC16
F3								X
F9					X			
F10		X						
F12	X							
F13			X					
F14				X				
F15						X	X	

2.3 Prototypy

Součástí analýzy byl prvotní návrh User Interface (UI), u kterého bylo nutné dbát na zažité konvence, aby uživatelé, kteří jsou zvyklí na systém *WatchOS*, i uživatelé, kteří s tímto systémem nemají žádné zkušenosti, dokázali aplikaci intuitivně používat a jednoduše dosáhli svých cílů.

Jednou z limitací při návrhu byla velikost displeje. Jelikož jsme se snažili pokrýt co největší škálu zařízení, bylo potřeba volit jednotlivé prvky tak, abychom zobrazovali veškerá důležitá data a ta se zároveň vešla na plochu displeje.

Tato sekce obsahuje popis jednotlivých prototypů, které vznikly před samotným vývojem aplikace. V rámci návrhu vznikly dva typy prototypů. Jeden **statický**, druhý **posouvatelný**. Důvod vzniku těchto dvou rozdílných prototypů byl zjistit, jestli se dokáží uživatelé po ploše radaru pohybovat i přesto, že mají na chytrých hodinkách výrazně menší využitelnou plochu, než je tomu například u mobilního telefonu. Statický prototyp neumožňuje uživateli pohyb ani změnu centrování a dovoluje jen pozorovat dění a zjišťovat si případné informace. Statický prototyp tedy nepokrývá některé předchozí funkční požadavky.

Statický prototyp

Statický prototyp uživateli nedovoluje pohybovat se po mapě ani nijak posouvat. Návrhy jsou dostupné na obrázcích 2.2 a 2.4. Dělí se na dva stavy: **stav, kdy dron neletí**, a **stav, kdy letí**.

Stav, kdy dron neletí

Ve stavu, kdy dron neletí, vidí uživatel pouze svou pozici a cizí drony v okolí, viz 2.4a. Radar je centrován na pozici uživatele. Pro zjištění dodatečných informací máme možnost posunout obrazovku doleva a zobrazit si tak seznam dronů, které vidíme na radaru, viz 2.4c. Při kliknutí na některou položku v seznamu přejde systém na obrazovku s detailními informacemi o příslušném zařízení. Tyto informace obsahují: Global Navigation Satellite System (GNSS) souřadnice, výšku zařízení, rychlost letu, název zařízení a další, viz 2.4d.

Stav, kdy dron letí

Ve stavu, kdy dron letí, vidíme uprostřed radaru svůj dron. Radar se při pohybu dronu vždy vycentruje na jeho polohu a sleduje takto jeho pozici. Také můžeme vidět pozici uživatele, pokud se nachází v dostatečné blízkosti, a cizí drony v okolí. Pokud by byl uživatel ve větší vzdálenosti, indikujeme jeho pozici pomocí modrého zbarvení na okraji obrazovky. Radar nás zároveň varuje, pokud je některý z cizích dronů v nebezpečné blízkosti našeho dronu, viz 2.4b. Indikace probíhá pomocí červeného zvýraznění nebezpečného dronu.

V obou stavech má uživatel možnost změnit si maximální vzdálenost a přiblížení pomocí digitální korunky, která se nachází na straně chytrých hodinek. Nastavená vzdálenost je viditelná v pravém horním rohu aplikace. V ani jednom případě nedáváme uživateli možnost klikat na prvky nacházející se na radaru a veškerá interakce s prvky probíhá pouze pomocí seznamu dronů.

Posouvatelný prototyp

Posouvatelný prototyp se oproti statickému liší hned v několika aspektech. Na rozdíl od statického prototypu můžeme klikat přímo na prvky, které zobrazuje radar. Taktéž se můžeme pomocí dotykového ovládání posouvat po mapě a vidět tak drony, které se nachází ve větším okruhu našeho dronu. Prototyp má opět dva stavy: **stav, kdy dron neletí**, a **stav, kdy letí**. Návrh je vidět na obrázcích 2.3 a 2.5.

Stav, kdy dron neletí

Ve stavu, kdy dron neletí, vidíme uprostřed obrazovky pozici uživatele a okolní drony viz 2.5a. Navíc zde máme možnost se po mapě posouvat a objevit tak drony nacházející se ve větším okruhu. Pro vycentrování zpět na pozici

uživatele slouží tlačítko nacházející se na dolní pravé straně obrazovky. Pokud si chce uživatel zobrazit dodatečné informace, musí pomocí dotykového ovládání kliknout na prvek, který chce vybrat, a aplikace mu zobrazí dodatečné informace o tomto zařízení.

Stav, kdy dron letí

Ve stavu, kdy dron letí, vidíme uprostřed obrazovky pozici dronu viz 2.5c. Radar se opět centruje na náš dron a sleduje jeho pohyb. Opět vidíme indikaci nebezpečných dronů v okolí, které by nás mohly ohrožovat – 2.5b a také pozici uživatele. Oproti předchozímu stavu máme na spodní straně další tlačítko pro centrování obrazovky na náš dron. Tlačítko je aktivní pouze v případě, že se uživatel posune po mapě a změní tak výchozí nastavení.

V obou stavech máme opět možnost měnit si maximální vzdálenost pomocí digitální korunky.

2.3.1 Lo-fi prototypy

Prvním krokem návrhu uživatelského rozhraní bylo vytvoření Low Fidelity (lo-fi) prototypů. Lo-fi prototypování [11] umožňuje rychle a za minimální náklady vyzkoušet naše nápady a návrhy. Většinou se jedná o sadu papírových náčrtů. Nevýhodou lo-fi prototypování je, že nedokážeme simulovat komplexnější interakce nebo animace. Také je vyžadována větší představivost účastníků.

Vzhledem k těmto nevýhodám a kvůli získání podrobnější zpětné vazby posloužili lo-fi prototypy pouze k prvotní představě výsledné aplikace a následné tvorbě Hight Fidelity (hi-fi) prototypů. Lo-fi prototypy můžeme vidět na obrázcích 2.2 a 2.3.

U návrhu posouvatelného prototypu se liší pouze obrazovka radaru, která dovoluje pohybovat se po obrazovce a klikat na prvky. Obrazovka detailu dronu zůstává totožná.

(a) Seznam dronů (b) Radar (c) Detail dronu

Obrázek 2.2: Lo-fi prototyp – statický

(a) Radar

Obrázek 2.3: Lo-fi prototyp – posouvatelný

2.3.2 Hi-fi prototypy

Hi-fi prototyp oproti předchozímu lo-fi prototypu je více detailní. Je navíc klikatelný a umožňuje uživateli s návrhem přímo interagovat. Díky tomu dokážeme objevit více nedostatků a účastníci si výslednou aplikaci dokáží lépe představit.

Hi-fi prototypování [12] má ovšem také své nevýhody. Stojí nás větší množství času, je daleko časově náročnější provádět v nich úpravy a účastníci testování se nemusí cítit dobře při kritizování nedostatků, jelikož se prototyp tváří jako hotová aplikace.

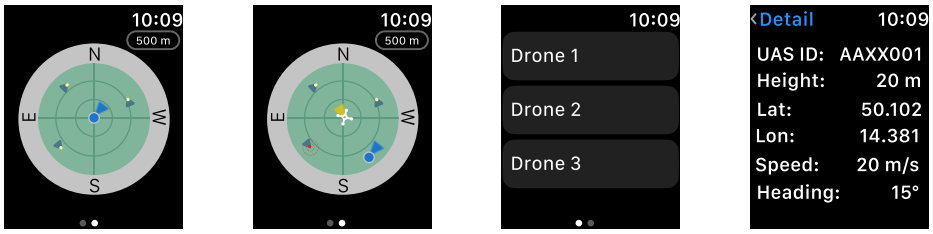
První návrh tohoto prototypu je velmi minimalistický, jelikož hlavním cílem bylo otestovat preference účastníků mezi statickou a posouvatelnou verzí. Návrhy lze vidět na obrázcích 2.4 a 2.5.

Pro účely testování byly hi-fi prototypy nejprve navrženy v programu *Figma* a následně implementovány jako jednoduchá aplikace spustitelná na reálném zařízení. Díky tomu jsme mohli testování provádět přímo na chytrých hodinkách *Apple Watch 3. generace* a uživatelé tak měli představu o tom, jak se systém bude chovat.

2.4 Testování prototypů

Testování prototypů s jednotlivými testery probíhalo v kanceláři společnosti *Dronetag s.r.o.* Před testováním byly prototypy nainstalovány na *Apple Watch 3. generace* se 38mm displejem. Vzhledem k tomu, že se jedná o nejmenší dostupné *Apple Watch* hodinky, mohlo dojít k určitému zkreslení testování. Ovšem výsledná aplikace bude dostupná i pro uživatele s tímto zařízením, a tedy je dobré otestovat použitelnost i na tomto zařízení.

Všichni účastníci tohoto testování měli základní znalosti ohledně dronů a jsou technicky zaměření.



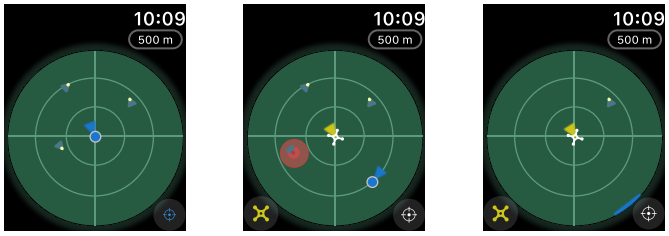
(a) Radar

(b) Radar - indikace nebezpečí

(c) Seznam dronů

(d) Detail dronu

Obrázek 2.4: Hi-fi prototyp – statický



(a) Radar – vycentrování na uživatele

(b) Radar – indikace nebezpečí

(c) Radar vycentrování na dron

Obrázek 2.5: Hi-fi prototyp – posouvatelný

Testovací scénáře

Každému z testerů byly postupně spouštěny výsledné prototypy a zadány následující úkoly:

Statický prototyp:

1. Dron neletí
   a) Najděte drony 1, 2 a 3.
   b) Zobrazte jejich detail.
2. Dron letí
   a) Podívejte se na radar. Jakým směrem musíme letět k uživateli?
   b) Pokuste se zobrazit uživatele na radaru.

Posouvatelný prototyp:

1. Dron neletí
    a) Otevřete detail dronu 1 a 2.
    b) Vycentrujte zpátky na pozici uživatele.
    c) Oddalte mapu a zjistěte informace o dronu 3.
2. Dron letí
    a) Otevřete detail dronu 1, 2 a 3.
    b) Vycentrujte zpátky na svůj dron.
    c) Zvětšete vzdálenost na 1500 metrů a otevřete detail dronu 1.

Tester č. 1

Tester měl zkušenosti se zařízeními Apple a v minulosti používal hodinky Apple Watch.

Statický prototyp:

1. Dron neletí
    a) Testera napadlo měnit si vzdálenost pomocí digitální korunky.
    b) Pro zobrazení detailu se nejdříve snažil klikat na drony na mapě. Poté ho napadlo přejít na levou záložku a otevřít si dron ze seznamu.
2. Dron letí
    a) Nejdříve mu nebylo jasné, k čemu slouží modrá indikace. Poté mu došlo, že se jedná o pozici uživatele.
    b) Uživatele zobrazil pomocí oddálení.

Posouvatelný prototyp:

1. Dron neletí
    a) Uživatel si všimnul, že nemá na výběr stránky jako u předchozího prototypu. Následně si pomocí klikání otevřel detail.
    b) Vycentrovat na uživatele se podařilo bez problému.
    c) Taktéž bez problému.
2. Dron letí
    * Body a), b) provedl účastník bez problému.
    c) Uživatel oddálil a následně vybral ze seznamu dron 1.

Nápady a poznatky:

1. U druhého prototypu se mu líbilo, že může klikat přímo na prvky, které vidí na radaru.

2. Pomohla by animace přiblížení, která by dala větší kontext tomu, aby uživatel věděl, co se děje.

3. Jak se bude aplikace chovat, kdybych letěl s více drony najednou?

Tester č. 2

Tester neměl žádné předchozí zkušenosti s chytrými hodinkami.

Statický prototyp:

1. Dron neletí

    a) Bez pomoci by nedokázal najít třetí dron. Nenapadlo ho použít digitální korunku.

    b) Detail dronu se podařilo otevřít bez problému.

2. Dron letí

    a) Prvek indikující směr uživatele mu připadal intuitivní.

    b) Uživatele zobrazil pomocí změny měřítka.

Posouvatelný prototyp:

1. Dron neletí

    * Body a), b) provedl přímočaře.

    c) Mapu se podařilo oddálit a zobrazit třetí dron. Uživateli se zdálo zvláštní chování při změně měřítka, což nejspíše zapříčinil samotný prototyp, který neměl podobu finální aplikace.

2. Dron letí

    a) Bez problému.

    b) Nebyla mu jasná ikona na tlačítku představující dron.

    c) Uživatel si změnil vzdálenost a úspěšně se mu podařilo otevřít detail prvního dronu.

Nápady a poznatky:

1. Testerovi vadilo, že se hodinky stále zhasínaly při kratším nepoužívání.
2. Uvítal by, kdyby bylo možno klikat na prvek zobrazující aktuální měřítko.
3. Při kliknutí na více dronů najednou by nemusela aplikace zobrazovat klikatelný čtverec. Stačilo by rovnou zobrazit seznam dronů, ze kterého uživatel může vybrat.
4. Zvážil by nepoužívat kolečko u radaru a dalších prvků, aby došlo k lepšímu využití místa displeje.
5. Spíše by se přikláněl k posouvatelnému prototypu.

Tester č. 3

Tester uvedl, že nemá žádné zkušenosti s chytrými hodinkami.

Statický prototyp:

1. Dron neletí
   a) Tester nedokázal najít třetí dron. Nevšiml si rádiusu v horní části obrazovky. Po nápovědě si dokázal pomocí digitální korunky změnit radius a najít tak třetí dron.
   b) Detail dronu se podařilo otevřít bez problému.
2. Dron letí
   a) Prvek indikující směr uživatele nebyl příliš pochopitelný, připomínal rybník.
   b) Uživatele zobrazil pomocí změny měřítka.

Posouvatelný prototyp:

1. Dron neletí

    a) Tester nejdříve hledal možnost zobrazení seznamu jako u prvního prototypu. Následně kliknul na drony a zobrazil detail.

    b) Pomocí tlačítka vycentroval zpět na pozici uživatele.

    c) Bez problému.

2. Dron letí

    * Neuvedené body a), b) provedl bez problému.

    c) Tester si změnil vzdálenost a kliknul na seskupení dronů. Bez problému vybral dron ze seznamu.

Nápady a poznatky:

1. Radius v horní části obrazovky by měl zobrazovat, v jaké jednotce se pohybujeme.

2. Hodinky jsou velmi malé, ale chybí například globální minimapa zobrazující celou scénu.

3. Tester se přikláněl k posouvatelnému prototypu.

Tester č. 4

Tester vlastní hodinky Apple Watch a používá na nich různé aplikace při sportu.

Statický prototyp:

1. Dron neletí

    a) Bez problému.

    b) Detail dronu se podařilo otevřít bez problému.

2. Dron letí

    a) Prvek indikující uživatele na radaru se mu zdál srozumitelný.

    b) Uživatele zobrazil bez problému.

Posouvatelný prototyp:

1. Dron neletí
    a) Tester si pomocí kliknutí na drony zobrazil jejich detail.
    b) Úspěšně se podařilo vycentrovat zpátky na pozici uživatele.
    c) Bez problému.
2. Dron letí
    a) Postup účastníkovi připadal srozumitelný.

Nápady a poznatky:

1. V případě hrozby kolize by mohly hodinky zavibrovat.
2. Čísla dronů jsou příliš malá a nemusí být čitelná.
3. Posouvatelný prototyp byl pro testera použitelnější.
4. Bylo by dobré využít větší plochu displeje než zobrazovat radar v kruhu.
5. Zvážil by také zvětšení prvků pro centrování, aby ikony byly viditelnější.
6. U hodinek, které mají kompas, by bylo dobré měnit orientaci radaru podle směru pohledu uživatele.

Tester č. 5

Tester neměl žádné předchozí zkušenosti s *Apple Watch* ani jinými chytrými hodinkami.

Statický prototyp:

1. Dron neletí
    a) Bez problému našel drony 1 a 2, u třetího dronu potřeboval napovědět ohledně změny měřítka pomocí digitální korunky.
    b) Detail dronů otevřel bez problému.
2. Dron letí
    a) Pomocí prvku indikujícího uživatele zjistil jeho směr.
    b) Bez problému zobrazil uživatele pomocí digitální korunky.

Posouvatelný prototyp:

1. Dron neletí

    a) Zpočátku nedokázal přijít na to, jakým způsobem otevřit detail dronů. Následně zkusil prstem klepnout na dron.

Ostatní úkoly zvládl vyřešit bez problému.

Nápady a poznatky:

1. Bylo by dobré mít na radaru prvek zobrazující pozici dronu, aby uživatel nemusel vždy centrovat na dron.

2. Pokud by byla mapa větší, raději by se po ní pohyboval. V opačném případě by mu stačil statický prototyp.

Tester č. 6

Tester měl zkušenosti s konkurenčními chytrými hodinkami. S hodinkami Apple Watch nikdy předtím nepracoval.

Statický prototyp:

1. Dron neletí

    * Oba dva body provedl bez problému.

2. Dron letí

    a) Směr uživatele našel v pořádku.

    b) Uživatele zobrazil pomocí digitální korunky.

Posouvatelný prototyp:

1. Dron neletí

    * Všechny body provedl bez problému.

2. Dron letí

    a) Bez problému.

    b) Pomocí levého tlačítka vycentroval na dron.

    c) Bez problému.

Nápady a poznatky:

1. Může být problém drony nakliknout v případě, že budou létat vysokou rychlostí.
2. Tlačítka pro centrování by mohla být nad sebou. Uvedl, že by mohl očekávat jinou funkcionalitu.
3. Spíše by využil variantu s posouvací mapou.
4. V případě, že se některý objekt nachází za viditelnou hranicí radaru, by uvítal zobrazit nějakou indikaci, aby uživatel věděl, že se nachází nějaký objekt a má možnost se k němu posunout.

2.4.1 Shrnutí

Účastníci, kteří měli již zkušenosti s používáním *Apple Watch*, měli menší problém s provedením jednotlivých úkolů. Ostatním účastníkům se po krátké nápovědě podařilo taktéž bez problému úkoly vykonat.

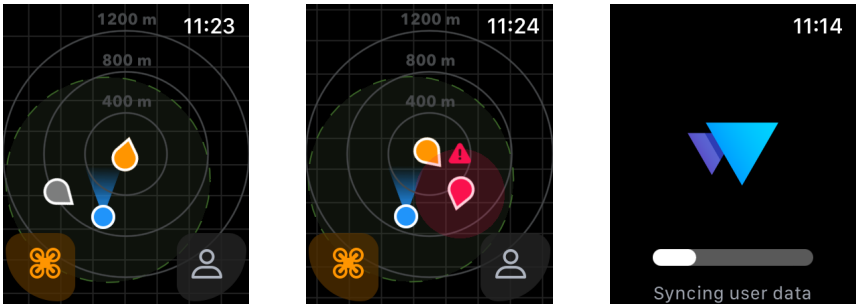
Posouvatelný prototyp byl na základě odezvy většiny účastníků posouzen jako lepší. Nápady a poznatky z každého testování byly dále uváženy při tvorbě návrhu výsledné aplikace.

2.5 Úpravy na základě testování

Díky výsledkům získaným z předchozího testování bylo v návrhu radaru provedeno množství úprav a návrh byl poté rozšířen o další potřebné obrazovky. V první řadě jsme z odpovědí vyhodnotili nejkritičtější místa návrhu, která by mohla později ztížit uživatelům práci s aplikací.

Jako nejvýznamnější problémy byly vyhodnoceny tyto:

1. Příliš malá plocha radaru.
2. Příliš malá tlačítka pro změnu centrování.
3. Chybějící jednotka u aktuálního přiblížení.
4. Chybějící indikace vlastního a ohrožujícího dronu v případě, kdy se nachází mimo plochu radaru.



(a) Radar (b) Radar – indikace nebezpečí (c) Úvodní obrazovka

Obrázek 2.6: Finální vzhled aplikace

Plocha radaru

Jelikož samotný radar je tou nejdůležitější funkcí celé aplikace, rozhodli jsme se i na základě výsledků testování využít pro jeho zobrazení celou plochu chytrých hodinek.

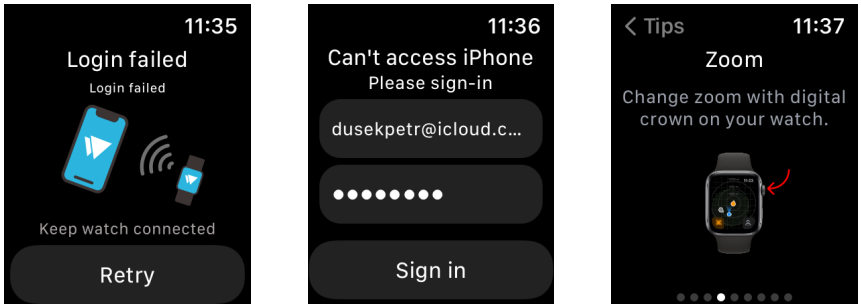
Tlačítka pro centrování byla upravená tak, aby překrývala plochu radaru. Dále byly upraveny tvary tlačítek a místo kruhového tvaru byl zvolen tvar připomínající trsátko. Tím jsme dosáhli větší úspory místa a tlačítka mohla být zvětšena.

Jednou z dalších úprav radaru bylo i odstranění prvku zobrazujícího aktuálně nastavené měřítko. Tento prvek v předchozím návrhu zabíral velké množství místa a popis uvnitř navíc nebyl dobře čitelný. Místo toho radar zobrazuje tři kruhy určující vzdálenost. Tyto kruhy nazveme **distanční kruhy**. Vzdálenost mezi distančními kruhy je dána odpovídající vzdáleností, vepsanou u každého kruhu. Kruhy mají mezi sebou vždy stejnou vzdálenost. Tedy jestliže první kruh má napsanou vzdálenost 20 metrů, další kruh znamená vzdálenost 40 metrů a poslední kruh 60 metrů.

Z důvodu pohybu prvků po ploše radaru bylo také přidáno pozadí. Díky pozadí je mnohem jasněji viditelné, který prvek se v aktuální chvíli pohybuje, a který prvek naopak ne.

Dále byla přidána postranní indikace vlastního a ohrožujícího dronu v případě, kdy se nachází mimo plochu radaru. Tímto uživatel zjistí, kterým směrem by se měl po radaru posunout, aby jednodušeji objevil hledaný prvek.

Finální podobu můžeme vidět na obrázcích 2.6a a 2.6b.



(a) Selhání přihlášení (b) Přihlášení (c) Onboarding

Obrázek 2.7: Finální vzhled aplikace

Úvodní obrazovka

Každá mobilní aplikace by měla obsahovat úvodní obrazovku. Většinou se jedná o jednoduchou obrazovku s logem samotné aplikace. Na této obrazovce by mělo dojít k získání potřebných dat a informací nutných ke běhu aplikace.

U chytrých hodinek sice není úvodní obrazovka striktně vyžadována, ovšem tato konkrétní aplikace potřebuje získat při spuštění informace o přihlášeném uživateli. Hlavním zdrojem informací o přihlášeném uživateli je přidružená mobilní aplikace. Z přidružené aplikace se systém při spuštění snaží získat data o tomto uživateli.

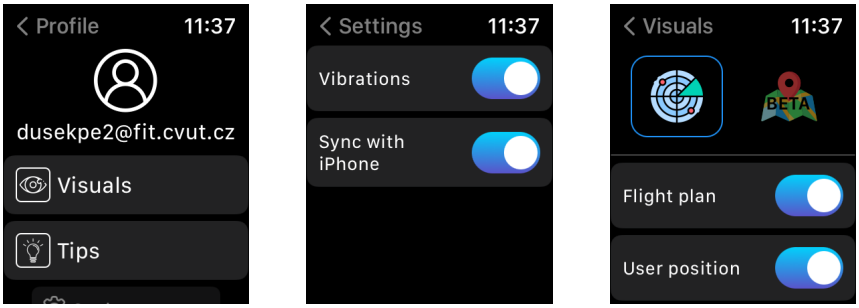
Úvodní obrazovka tedy obsahuje logo aplikace, pod kterým vidíme prvek informující o stavu synchronizace uživatele, viz obrázek 2.6c. Na této obrazovce probíhá na pozadí získávání potřebných dat o přihlášeném uživateli. Získávání těchto informací však může v některých případech selhat.

Selhání přihlášení

Jak již bylo zmíněno, mohou nastat situace, při kterých se systému nepodaří získat informace o uživateli. V takovém případě systém uživatele přesměruje na obrazovku informující o chybě, která nastala. Uživatel zde má pouze jednu možnost, kterou je tlačítko *Retry*. Po stisknutí se systém pokusí opět získat data z mobilní aplikace. V případě, že požadavek opět několikrát selže, je uživatel automaticky přesměrován na obrazovku přihlášení, viz obrázek 2.7a.

Přihlášení

V situaci, kdy dojde k chybám při získávání dat z mobilní aplikace, nebo v případě vypnuté synchronizace zobrazuje systém obrazovku přihlášení. Tato obrazovka dovoluje přihlásit se nezávisle na mobilní aplikaci zadáním registrované e-mailové adresy a příslušného hesla, jak vidíme na obrázku 2.7b.



(a) Profil (b) Obecné nastavení (c) Nastavení vizuálů

Obrázek 2.8: Finální vzhled aplikace

Onboarding

Onboarding informuje uživatele při prvním zapnutí aplikace o funkcích, které aplikace umožňuje. Jelikož se jedná o první setkání uživatele s aplikací, měly by být vysvětleny veškeré důležité funkce pro to, aby uživatel dokázal aplikaci používat. Ukázku jedné z částí vidíme na obrázku 2.7c.

Profil

U mobilních aplikací je zvykem zobrazovat obrazovku profilu, na které máme většinou přístup k nastavení. Převzali jsme tedy toto chování a přidali obrazovku profilu i pro tuto aplikaci.

Profil zde slouží k informování o aktuální verzi, přihlášeném uživateli a je také mezikrokem k přechodu na další obrazovky: nastavení vizuálů, zobrazení tipů (onboarding) a obecného nastavení, viz obrázek 2.8a.

Nastavení vizuálů

Nastavení vizuálů dovoluje uživateli personalizovat si aplikaci. Slouží převážně k aktivaci či deaktivaci vizuálních prvků, jako jsou například: zobrazení letové zóny, zobrazení pozice uživatele, postranní indikace atd. Finální podobu vidíme na obrázku 2.8c

Obecné nastavení

Poslední obrazovkou je obecné nastavení. Obecné nastavení dovoluje vypnutí synchronizace s mobilním telefonem nebo vypnutí vibrací. Ukázka vzhledu je na obrázku 2.8b.

Kapitola 3

Realizace

Tato kapitola se zabývá popisem realizace aplikace pro chytré hodinky. Součástí realizace je i návrh architektury, kterou se aplikace řídí.

Realizace dále pokrývá tvorbu grafického rozhraní a získávání dat prostřednictvím poskytnutého REST API, které nebylo součástí vývoje, jelikož společnost *Dronetag s.r.o.* má v tuto chvíli již implementovanou webovou službu, která poskytuje data dalším vývojářům, a také mobilní aplikaci *Dronetag*, navázanou na toto API. Aplikace pro chytré hodinky využívá stejného API, avšak nejedná se pouze o rozšíření mobilní aplikace. Stávající kód nebylo možné použít z důvodu, že mobilní aplikace je napsaná v jazyce *Flutter*, který v tuto chvíli nepodporuje vývoj aplikací pro chytré hodinky.

Taktéž nebylo vhodné využít volání funkcí mobilní aplikace, jelikož by mohlo docházet k chybám v rámci dostupnosti mobilní aplikace, což by způsobovalo výpadky systému na chytrých hodinkách. I kvůli možnému osamostatnění vzniklé aplikace v budoucnu bylo vhodnější implementovat veškerou logiku nezávisle.

V tuto chvíli aplikace využívá mobilní aplikaci pouze k získání dat přihlášeného uživatele, ovšem je připravena i na případ, kdy mobilní aplikace nereaguje a chytrý telefon není v blízkosti nebo neodpovídá. V tomto případě je ovšem nutné mít hodinky připojené k internetu nebo mobilní síti, jelikož primárním zdrojem je právě zmíněné REST API.

Implementace byla provedena ve vývojovém prostředí *XCode* za použitím programovacího jazyka *Swift*. Jedná se o programovací jazyk určený k implementaci aplikací pro platformy společnosti *Apple Inc.* Jedná se o mobilní zařízení, chytré hodinky, ale podporuje i tvorbu aplikací pro systém *MacOS*.

3.1 Architektura

Při vývoji softwaru je jednou z nejdůležitějších fází navržení architektury. Architektura hraje významnou roli v případě udržitelnosti systému a pozdější orientaci v kódu. Chceme vytvořit aplikaci, která bude do budoucna jednoduše rozšiřitelná a v níž se další programátoři dokáží vyznat. Cílem architektury je udržet komplexitu aplikace na takové úrovni, aby bylo jednoduché ji později spravovat. U menších projektů sice volba architektury nehraje tak významnou roli jako u složitějších a robustnějších projektů, nicméně správným návrhem architektury dokážeme v budoucnu ušetřit velké množství času.

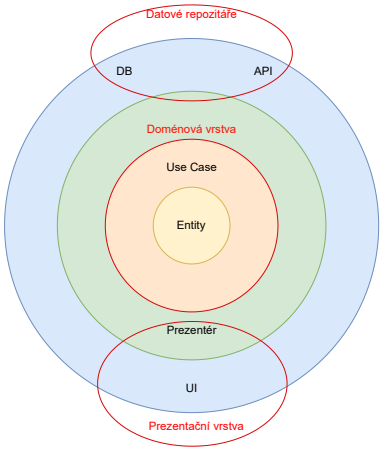
Princip, kterým bychom se měli při návrhu řídit, je nazýván SOLID [13], [14]. Jedná se o akronym počátečních písmen pěti principů:

- **S** – Single Responsibility Principle – Princip jedné odpovědnosti.
  Každá třída má právě jednu zodpovědnost
- **O** – Open–Closed Principle – Princip otevřenosti a uzavřenosti.
  Třídy by měly být otevřené pro rozšiřování, ale uzavřené pro změny.
- **L** – Liskov Substitution Principle – Liskovův princip zaměnitelnosti
  Podtřídy by měly být zaměnitelné s jejich bázovými třídami.
- **I** – Interface Segregation Principle – Princip oddělení rozhraní
  Více specifických rozhraní je lepší než jedno univerzální rozhraní.
- **D** – Dependency Inversion Principle – Princip obrácení závislostí
  Závislost na abstrakcích, nikoliv na implementacích.

Nyní se podívejme na obrázek 3.1. Jednotlivé kruhy reprezentují různé úrovně softwaru. Středový kruh je nejvíce abstraktní a vnější kruh je naopak nejvíce konkrétní, toto nazýváme **Princip abstrakce**.

Princip abstrakce říká, že vnitřní kruhy by měly obsahovat byznys logiku a kruhy vnější naopak implementační detaily [15]. Vnější kruh reprezentuje konkrétní mechanismy, které jsou specifické pro danou platformu, jako například síťovou vrstvu nebo přístup k databázi. Čím více se pohybujeme po kruzích do středu, tím je vrstva abstraktnější. Vnitřní kruh je nejvíce abstraktní, a jak již bylo řečeno, obsahuje byznys logiku. Ta nezávisí na dané platformě ani na knihovně, kterou používáme.

Důležité je zachovat pravidlo, které říká, že bychom neměli mít závislosti z vnitřních vrstev do vrstev vnějších. Tento princip souvisí s čistou architekturou, která je popsána v následující podsekci.



Obrázek 3.1: Graf – čistá architektura

3.1.1 Čistá architektura

Čistá architektura [16] se snaží rozdělit aplikaci na několik vrstev. Jedná se o princip Separation of Concerns (SoC) [17]. Tento princip rozděluje program na několik odlišných vrstev, které jsou si vzájemně disjunktní.

Existují různé názory, kolik vrstev by čistá architektura měla mít, ale není definovaný žádný striktní počet.

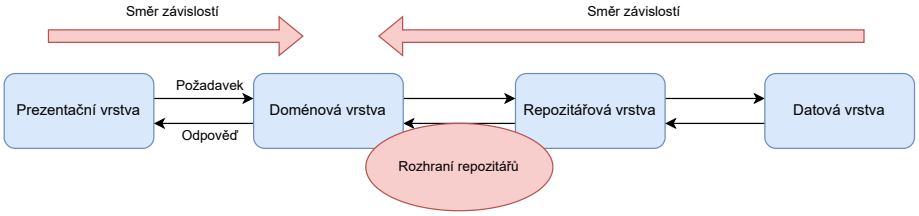
Zvolili jsme architekturu, která sestává ze čtyř vrstev:

- **Datová vrstva**

  Slouží k získávání dat z různých datových zdrojů. Data mohou být získávána buďto z lokální databáze, nebo ze vzdálených systémů, jako například REST API, nebo z připojeného zařízení. Konkrétní implementace je popsána v sekci 3.1.3

- **Repozitářová vrstva**

  Repozitáře se starají o datové operace. Slouží jako jediný zdroj pravdy a mají na starosti doručení relevantních dat. Repozitář je propojený s datovou vrstvou a rozhoduje, ze kterého konkrétního zdroje bude data získávat, nebo naopak ukládat. Repozitář je v podstatě prostředník mezi datovou a doménovou vrstvou. Podrobnější popis nalezneme v sekci 3.1.4



Obrázek 3.2: Čistá architektura – závislost vrstev

- **Doménová vrstva**

  Doménová vrstva obsahuje byznysovou logiku. Jedná se o entity, případy užití a rozhraní repozitářů. Tato vrstva může být potencionálně přepoužita i v jiných projektech, jelikož nezávisí na konkrétních implementacích. Doménová vrstva by neměla obsahovat žádné části jiné vrstvy. Podrobnější popis následuje v sekci 3.1.5.

- **Prezentační vrstva**

  Obsahuje uživatelské rozhraní a potřebná data získává z doménové vrstvy. Prezentační vrstvu můžeme navíc rozdělit na komponenty podle zvolené architektury. Více v sekci 3.1.6

Pro lepší představu lze komunikace mezi zmíněnými vrstvami vidět na obrázku 3.2.

3.1.2 Použité technologie

Abychom se lépe orientovali v následujících popisech implementačních detailů jednotlivých vrstev, je potřeba si nejprve vysvětlit technologie, které byly v tomto projektu využity.

Dependency injection (DI) – Swinject

DI je návrhový vzor, který slouží pro snížení závislostí mezi jednotlivými částmi systému. *Swinject* [18] je framework, který pomáhá tento vzor implementovat v jazyce *Swift*, a můžeme tak rozdělit naši aplikaci na volně vázané komponenty, které mohou být jednodušeji testovány a vyvíjeny.

Ačkoliv lze implementovat DI bez použití knihoven třetích stran, *Swinject* poskytuje jednoduché řešení závislostí, i v případě zvyšující se komplexity kódu.

O závislosti ve vytvořené aplikaci se stará třída `ContainerBuilder`, která používá vzor `Builder`. Tato třída je volána vždy při spuštění aplikace funkcí *buildContainer()*. V této funkci jsou postupně vytvořeny kontejnery pro jednotlivé vrstvy, jež jsme zmínili v podsekci 3.1.1.

```
import Swinject

class ContainerBuilder {

    // MARK: - Build
    static func buildContainer() -> Container {
        var container = Container(parent: nil,
                                  defaultObjectScope: .transient)

        container = registerDataSourceLayer(to: container)
        container = registerRepositoryLayer(to: container)
        container = registerDomainLayer(to: container)

        return registerPresentationLayer(to: container)
    }

    // MARK: - Registrations
    static func registerDataSourceLayer(
        to container: Container
    ) -> Container {
        let container = Container(parent: container,
                                  defaultObjectScope: .container)
        ...

        return container
    }

    ...

    static func registerPresentationLayer(
        to container: Container
    ) -> Container {
        let container = Container(parent: container,
                                  defaultObjectScope: .transient)

        ...

        return container
    }
}
```

Ukázka kódu 3.1: Třída ContainerBuilder

Funkce *registerDataSourceLayer()* registruje třídy v datové vrstvě, funkce *registerRepositoryLayer()* naopak třídy v repozitářové vrstvě a tak dále.

Swinject umožňuje také tvorbu dílčích kontejnerů. Vidíme, že v každé funkci je vytvářený nový kontejner, kterému při inicializaci předáváme rodičovský kontejner a rámec objektu.

Hierarchie kontejnerů odpovídá stromové struktuře. Všechny služby registrované u rodiče jsou dostupné i všem jeho potomkům.

Dále zde máme různé typy rámců: `transient`, `graph`, `container` a `weak`. Pro nás jsou důležité dva z těchto typů, a to `container` a `transient`:

- `container`

  Jedná se o typ rámce, kdy je vytvořena jen jediná instance objektu. Ta je následně sdílena dalším kontejnerům. Jinými slovy se jedná o *Singleton*.

- `transient`

  Vytvořená instance není sdílena. Jinými slovy kontejner vždy vytváří novou instanci objektu.

Vidíme, že u všech vrstev kromě vrstvy prezentační definujeme rámec `container`, jelikož se jedná o *Singletony*. U prezentační vrstvy je použitý rámec `transient`, jelikož každý prezentační modul musí být vytvářený znovu. Aplikace může například zobrazovat dva stejné prezentační moduly na jiných místech aplikace, kdyby se jednalo o *Singleton*, byl by jeden z těchto modulů při zobrazení již nainicializovaný a nemusel by zobrazovat správná data.

Samotná registrace je triviální. V jazyce *Swift* používáme pro definování rozhraní klíčové slovo `protocol`. Každý datový zdroj má tedy své rozhraní a k němu minimálně jednu implementaci, která tomuto rozhraní odpovídá. Pokud máme tedy například rozhraní `SecuredStorage` a jeho implementaci `SecuredStorageImpl`, registraci do kontejneru provedeme takto:

```
static func registerSecuredStorage(to container: Container) {
    container.register(SecuredStorage.self) { _ in
        return SecuredStorageImpl()
    }
}
```

Ukázka kódu 3.2: Registrace třídy bez závislostí ve *Swinject* frameworku

Ve chvíli, kdy je nějaká třída navázána na toto rozhraní, předáme jí referenci pomocí funkce *resolve()*.

```
static func registerSomeRepository(to container: Container) {
    container.register(SomeRepository.self) { r in
        SomeRepositoryImpl(
            securedStorage: r.resolve(SecuredStorage.self)!,
            keyValueStorage: r.resolve(KeyValueStorage.self)!,
            coreDataStorage: r.resolve(CoreDataStorage.self)!
            ...
        )
    }
}
```

Ukázka kódu 3.3: Registrace třídy se závislostmi ve *Swinject*

Stinsen

Framework *SwiftUI*, který bude více popsaný níže v podsekci 3.1.6, obsahuje v tuto chvíli několik problémů. Jedním z nich je přesměrování. Přesměrování se implementuje pomocí struktury `NavigationLink`. Ta má podobné chování jako obyčejné tlačítko, s tím rozdílem, že výsledkem je přesměrování na jinou obrazovku.

To ovšem přináší problém takový, že by tímto způsobem každé `View` muselo vědět o všech možných cestách, kam může nastat přesměrování. Jedná se o porušení **principu jedné odpovědnosti**, popsaného na začátku této sekce, a nastává problém těsného spojení.

Existuje pár způsobů, jak lze alespoň částečně oddělit přesměrování ze samotného **View**, například definováním přechodů v rámci třídy `ViewModel`. Robustnější a jednodušší řešení však nabízí knihovna třetích stran *Stinsen* [19]. s *Stinsen* umožňuje velice elegantně oddělit logiku přesměrování mezi obrazovkami a odstraňuje nutnost definování struktury `NavigationLink` u implementace *SwiftUI View*. Třída, která se stará o přesměrování, musí mít definované tzv. kořenové *View* a všechny další cesty je potřeba definovat operátorem `@Route`, kterému navíc předáme styl přechodu.

Styly přechodů mohou nabývat těchto typů:

- `modal`
- `push`
- `fullscreen`

Příklad použití vypadá následovně:

```
final class Coordinator: NavigationCoordinatable {
    let stack = NavigationStack(initial: \Coordinator.start)

    @Root var start = makeStart
    @Route(.modal) var forgotPassword = makeForgotPassword
    @Route(.push) var registration = makeRegistration

    func makeRegistration() -> RegistrationCoordinator {
        return RegistrationCoordinator()
    }

    @ViewBuilder func makeForgotPassword() -> some View {
        ForgotPasswordScreen()
    }

    @ViewBuilder func makeStart() -> some View {
        LoginScreen()
    }
}
```

Ukázka kódu 3.4: Stinsen – koordinátor

Výhodou této knihovny je, že je napsaná čistě ve frameworku *SwiftUI* a je jí možné používat napříč všemi platformami, které *SwiftUI* podporují.

Detailnější implementace přesměrování je popsána v navazující sekci 3.1.6

Alamofire

Další z knihoven třetích stran použitých v této práci je knihovna *Alamofire* [20]. *Alamofire* je určená ke psaní HTTP požadavků a činí kód oproti nativní implementaci přehlednějším.

Jedná se o jednu z nejpoužívanějších knihoven vývojáři v rámci jazyka *Swift*.

Reaktivní programování – Combine

Reaktivní programování je deklarativní programovací paradigma, které se soustředí na datové toky [21]. Je to jedna z cest jak psát asynchronní kód.

Donedávna museli programátoři v jazyce *Swift* využívat pro reaktivní programování knihovny třetích stran, jako například *RxSwift* [22]. Avšak s příchodem *SwiftUI* představila společnost *Apple Inc.* vlastní reaktivní framework *Combine* [23].

Combine poskytuje deklarativní rozhraní pro zpracování hodnot v průběhu času. Tyto hodnoty mohou reprezentovat více druhů asynchronních událostí.

Combine deklaruje dva základní typy:

- `Publisher` – vydavatel

  Je typ rozhraní, které může poskytovat posloupnost hodnot v průběhu času. Vydavatelé vydávají hodnoty pouze tehdy, když o to odběratelé požádají.

- `Subscriber` – odběratel

  Na konci řetězce vydavatelů je odběratel, jenž přijímá hodnoty poskytnuté vydavatelem. Odběratel má pod kontrolou, jak rychle přijímá události od vydavatelů, ke kterým je připojen.

Combine má dva vestavěné vydavatele:

- `assign`

  Přiřazuje výstup vydavatele proměnné nebo objektu.

- `sink`

  Zavoláním `sink` nad vydavatelem přiřazujeme odběratele.

Dále můžeme nad odběratelem kdykoliv zavolat funkci *cancel()*, která ukončí přijímání událostí a uvolní alokované prostředky. Ukončení událostí lze řešit i pomocí objektu `AnyCancellable`. Instance objektu `AnyCancellable` automaticky volá funkci *onCancel()* při deinicializaci.

Nad vydavatelem můžeme aplikovat množství operátorů [24]. Zmíníme několik nejpoužívanějších:

- `map`

  Transformuje všechny prvky získané od vydavatele na prvky definované v uzávěru (closure).

- `merge`

  Spojuje hodnoty od více vydavatelů do jednoho datového toku.

- `flatMap`

  Transformuje prvky z předchozího vydavatele na nového vydavatele.

- `filter`

  Odflitruje pouze prvky, které splňují danou podmínku.

Příklad použití:

```
private var cancellables = Set<AnyCancellable>()

dronesRepository.getMyDrone()
    .map { $0?.name ?? "" }
    .removeDuplicates()
    .flatMap { [unowned self] serialNumber in
        return getFlightZone(serialNumber: serialNumber)
    }
    .sink(receiveValue: { [weak self] flightZone in
        // Do something
    })
    .store(in: &cancellables)
```

Ukázka kódu 3.5: Příklad použití frameworku *Combine*

3.1.3 Datová vrstva

Datová vrstva obsahuje definice rozhraní a jejich konkrétní implementaci. Aplikace je navázána na čtyři datové zdroje, kterými jsou:

- **REST API**
- **lokální databáze**
- **WebSocket**
- **přidružená mobilní aplikace Dronetag**

Pro každý tento datový zdroj je definované rozhraní. Podrobnější popis datových zdrojů a jejich napojení je detailněji popsán v sekci 3.3

3.1.4 Repozitářová vrstva

Repozitářová vrstva obsahuje implementace repozitářů. Každý repozitář má na starosti poskytování dat, která spolu souvisí.

Konkrétními repozitáři jsou:

- `UserSessionRepositoryImpl`

  Stará se o přihlášení uživatele, získávání tokenů potřebných pro přihlášení, autorizace webové služby a stažení informací o přihlášeném uživateli.

- `UserLocationRepositoryImpl`

  Stará se o získání GNSS lokace uživatele a směru, kterým se uživatel otáčí.

- `SettingsRepositoryImpl`

  Tento repozitář se stará o nastavení a jeho ukládání. Jde například o zapínání vibrací, přepínání vybraného typu radaru nebo přepínání synchronizace s telefonem.

- `DronesRepositoryImpl`

  Tento repozitář lze považovat za jeden z nejdůležitějších. Díky němu získává aplikace pozici svého i ostatních letadel, letové zóny a detailní informace o letadlech. Repozitář získává data z REST API nebo přes protokol *Websocket*.

- `MockedDronesRepositoryImpl`

  Tento repozitář má stejné rozhraní jako předchozí repozitář. Rozdíl od předchozího repozitáře je, že neposkytuje reálná data, ale pouze data předem definovaná. Díky tomuto repozitáři je jednodušší provádět například jednotkové testy, jelikož víme, jaká data máme očekávat.

- `VisualsRepositoryImpl`

  Stará se o nastavování, ukládání a získávání preferovaných vizuálních prvků, které chce uživatel vidět na radaru, respektive na mapě.

- `VersionRepositoryImpl`

  Stará se o poskytování aktuální verze aplikace. Jedná se zatím o velmi jednoduchý repozitář, ovšem mohl by být v budoucnu více rozšířen a to je důvod, proč tento repozitář vzniknul samostatně.

3.1.5 Doménová vrstva

Jak již bylo řečeno, doménová vrstva obsahuje **doménové entity**, **případy užití** a **definice rozhraní** pro repozitáře, které byly zmíněny v předchozí podsekci.

Nejdříve je potřeba si vysvětlit pojem *případ užití*, jelikož jsme tento pojem již používali v kapitole **Analýza a návrh** 2. V tomto případě pojem znamená něco trochu jiného a budeme pro něj používat anglický název – **UseCase**.

Případ užití v předchozí kapitole znamenal popis akcí a kroků, které definují interakci mezi účastníkem a systémem. Tyto akce a kroky vedou k dosažení nějakého cíle a pokrývají funkční požadavky.

Na rozdíl od toho **UseCase** v doménové vrstvě popisuje článek [25] tak, že se jedná o objekt, který má jednu nebo více funkcí, implementující specifický problém, například přihlášení.

Z pohledu samotné implementace `UseCase` rozhraní poskytuje navenek pouze jednu funkci, kterou je funkce *use()*. `ViewModel`, který tento `UseCase` volá, nepotřebuje znát přesnou implementaci, zajímá ho jen správný výsledek.

Tímto způsobem dokážeme odstranit velkou část logiky ze samotného `ViewModelu`, a navíc pokud bychom měli více `ViewModelů` potřebujících stejnou logiku, nemusíme ji implementovat vícekrát a stačí nám zavolat jeden jediný `UseCase`.

Doménová vrstva je prostředníkem mezi vrstvou prezentační a datovou. Nemělo by se tedy stát, že by byla některá třída z prezentační vrstvy závislá na třídě z datové vrstvy.

Konkrétními `UseCase` jsou:

- `GetFirstScreenAfterSplashScreenUseCase`

  Obstarává správné zobrazení obrazovky, která následuje po úvodní obrazovce.

- `GetFirstScreenAfterLoginScreenUseCase`

  Obstarává, která obrazovka se má zobrazit po přihlášení.

- `GetDefaultCenteredTypeUseCase`

  Stará se o to, na který prvek má být radar ve výchozím stavu vycentrován.

- `GetPossibleScrollValuesForRadarUseCase`

  Obstarává hodnoty, mezi kterými si uživatel může měnit měřítko radaru.

- `GetValidAccessTokenUseCase`

  Zajišťuje získání validního tokenu pro přihlášeného uživatele. Token je platný pouze po nějakou dobu a v případě, že vyprší, je potřeba, aby ho systém obnovil.

Je dobré zamyslet se nad tím, jestli pro nějakou logiku dává smysl vytvářet `UseCase`. Kdybychom vytvářeli `UseCase` pro veškerou jednoduchou logiku, aplikace by se mohla stát nepřehlednou.

3.1.6 Prezentační vrstva

Prezentační vrstva obsahuje uživatelské rozhraní. Při vývoji *iOS* aplikací definujeme uživatelské rozhraní pomocí tříd *UIViewController* nebo nově pomocí *SwiftUI View*. Pohledy (`View`) by se měli soustředit pouze na zobrazení grafických prvků. Jsou koordinovány buďto použitím *Controlleru*, *Presenteru*, nebo *ViewModelu*. To ovšem závisí na zvoleném architektonickém vzoru prezentační vrstvy.

Nejčastěji používané vzory u prezentační vrstvy jsou tyto:

- **Model-View-Controller (MVC)**
- **Model-View-Presenter (MVP)**
- **Model-View-ViewModel (MVVM)**

MVC

Architektura MVC [26] se dělí na tři komponenty. Tyto tři komponenty se nazývají **Model**, **View** a **Controller**.

MVC obecně definuje vrstvy tak, že **Model** reprezentuje vnitřní logiku aplikace a je zodpovědný za ukládání a načítání dat. **View** je zodpovědné za vykreslování uživatelského rozhraní a interakci s uživatelem a **Controller** je komponenta, která se stará o tok událostí mezi **View** a **Modelem**.

MVC u iOS aplikací

Architektura MVC byla donedávna u iOS aplikací vnímána jako oficiální architektura. Hlavním frameworkem pro tvorbu grafického rozhraní byl framework *UIKit* [27], který je poslední dobou nahrazován frameworkem *SwiftUI* [28]. *SwiftUI* framework byl poprvé vydán v roce 2019 a je v tuto chvíli stále v raném vývoji. Tudíž pro složitější grafická rozhraní se stále neobejdeme bez použití frameworku *UIKit*. Ovšem pro jednoduché aplikace je jeho použití dostatečné.

Implementace MVC se liší v závislosti na použití zmíněných frameworků:

- **UIKit**

  Definice MVC se u frameworku *UIKit* oproti obecné definici liší. **Model** se podle definice nemění, **View** je reprezentováno třídou `UIView`, která je zodpovědná za vykreslování grafického rozhraní a ve chvíli, kdy uživatel provede akci, informuje o tom svázanou třídu `UIViewController`. `UIViewController` je něco jako spojení **View** a **Controlleru** dohromady – budeme jej dále nazývat **ViewController**. **ViewController** přijí-má uživatelské akce z **View** a reaguje na ně zavoláním příslušné funkce **Modelu**. Následně po vykonání funkce **Modelu** provede aktualizaci **View** a restartuje cyklus. Vazbu mezi třídami lze vidět na obrázku 3.3.

  Hlavním problémem této architektury je však oddělitelnost **View** od **ViewControlleru**, jelikož každý **ViewController** si drží pevnou vazbu a **View** a dochází tedy k těsné vazbě, která snižuje znovupoužitelnost.

- **SwiftUI**

  U frameworku *SwiftUI* je naopak standardní architekturou MVVM. K tomu se ale dostaneme až v další části 3.1.6.

  Přesto, že *SwiftUI* nemá žádný **ViewController**, existují způsoby, kterými lze jeho chování simulovat. Z důvodu, že se nejedná o standardní přístup, nebudeme tento způsob v rámci této práce zmiňovat.

Obrázek 3.3: UIKit – architektura MVC

Obrázek 3.4: UIKit – architektura MVP

Model-View-Presenter

Architektura MVP [29] je alternativou k MVC. Skládá se z komponent **Model**, **View** a **Presenter**.

Model opět reprezentuje datovou vrstvu. Je zodpovědný za byznysovou logiku a komunikaci s databází a síťovou vrstvou. **View** reprezentuje uživatelské rozhraní a informuje **Presenter** o akcích uživatele. **Presenter** načítá data z **Modelu**, aplikuje logiku uživatelského rozhraní a spravuje, co se má na **View** zobrazit.

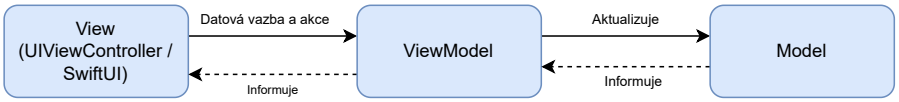
MVP u iOS aplikací

MVP u iOS aplikací se snaží vylepšit nedostatky architektury MVC. Propojení mezi komponentami vidíme na obrázku 3.4.

Na první pohled se může zdát, že architektura MVP je totožná s architekturou MVC, jen s jinými názvy komponent. Tak tomu ovšem není. U MVP je samotný **ViewController** vnímán jako **View**. To znamená, že **ViewController** zahrnuje pouze kód související s **View** a veškerá další logika je implementována v **Presenteru**.

Díky tomu můžeme velice jednoduše oddělit uživatelské rozhraní od zbytku komponent a zbavíme se tak těsné vazby. **View** je závislé pouze na rozhraní **Presenteru** a jeho implementace je tak odstíněna.

Použitím této architektury izolujeme životní cyklus **View** a je jednodušší v této architektuře psát například jednotkové testy. Na druhou stranu má i tato architektura své nevýhody. **Presenter** se může časem nafukovat a má tak příliš mnoho práce, jelikož se stará o chod celého **View**. Tím porušujeme zmíněný *princip jedné odpovědnosti*.



Obrázek 3.5: Swift – architektura MVVM

Model-View-ViewModel

MVVM je architektura, kde **Model** představuje veškerou logiku a data, se kterými aplikace pracuje. **View** je vrstva uživatelského rozhraní a **ViewModel** spojuje obě předchozí vrstvy. **ViewModel** tedy získává data z **Modelu** a informuje **View** o změnách. Propojení mezi komponentami je viditelné na obrázku 3.5.

MVVM u iOS aplikací

Architektura MVVM [30] má opět společné rysy s předchozí architekturou MVP. **ViewController** opět reprezentuje pouze uživatelské rozhraní. **ViewModel** získává informace z **ViewControlleru**, zpracovává je a posílá zpět. **Model** zůstává stejný jako u předchozích architektur.

Máme zatím popis MVVM pouze ve frameworku *UIKit*. Jak je tomu ale v případě *SwiftUI*? *SwiftUI* je přímo dělané pro architekturu MVVM [31]. Zbavujeme se nutnosti použití třídy `UIViewController` a dokážeme tedy oddělit samotné **View** od zbytku komponent. V případě, že **View** nezávisí na vnějším stavu, hrají roli **ViewModelu** lokální proměnné, označené jako `@State`, které poskytují vazbu pro obnovení uživatelského rozhraní, při každé změně stavu. Pro složitější scénáře roli **ViewModelu** představuje externí třída rozšiřující protokol `ObservableObject`.

MVC vs MVP vs MVVM

Zmínili jsme několik nejpoužívanějších architektur. Kterou z nich ale vybrat? MVP i MVVM jsou deriváty MVC. Klíčovým rozdílem od MVC je závislost, kterou mezi sebou mají jednotlivé vrstvy, a jak pevně jsou spolu svázané.

U MVC má **View** přímý přístup k **Modelům**. Vystavování kompletních **Modelů** může mít ovšem za následek ztrátu výkonu. Navíc tak **View** vystavujujeme data, která pro něj nejsou relevantní. MVVM se snaží tomuto předejít.

U MVP je role **Controlleru** nahrazena **Presenterem**. **Presentery** stojí na stejné úrovni jako **View**, odposlouchávají události nad **View** i **Modelem** a zprostředkovávají akce mezi nimi. Narozdíl od MVVM zde není mechanismus pro vazbu **View** na **ViewModely**. Místo toho spoléháme na to, že každé **View** implementuje rozhraní umožňující **Presenteru** interagovat s **View**.

MVVM nám umožňuje vytvořit specifický **Model** pro **View**, který obsahuje pouze informace relevantní pro **View**. Nemusíme tak dávat **View** přístup k celému **Modelu**. Na rozdíl od **Presenteru** u architektury MVP není vyžadováno, aby **ViewModel** odkazoval na **View**. **View** se může vázat na vlastnosti **ViewModelu**, který nazpět vystavuje data obsažená v **Modelech**.

Jelikož budeme chtít pro tvorbu UI využívat převážně framework *SwiftUI*, který je na architekturu MVVM připravený, a zároveň je u této architektury jednodušší testovatelnost kódu a lepší udržitelnost, zdá se tato architektura být dobrou volbou.

WatchOS

Doposud architektura prezentační vrstvy odkazovala pouze k *iOS* aplikacím. Nicméně tato práce se zabývá tvorbou systém pro *WatchOS*. Důvodem toho, že jsme se doposud zabývali pouze systémem *iOS*, je jejich podobnost.

Framework *UIKit* používá třídu `UIViewController`; tu při vyvíjení aplikací na systém *WatchOS* nenalezneme. Existuje zde však ekvivalent, a to třída `WKInterfaceController`. `WKInterfaceController` plní stejný účel jako třída `UIViewController` a lze na ní tedy aplikovat totožné architektury. Zatímco framework *UIKit* na systému *iOS* dovoluje vytvářet grafické rozhraní dvěma způsoby – pomocí storyboardů nebo čistě pomocí kódu, systém *WatchOS* ve frameworku *WatchKit* (ekvivalent frameworku *UIKit* pro systém *WatchOS*) vyžaduje definici grafického rozhraní čistě pomocí storyboardů.

Storyboardy jsou vizuálním nástrojem pro tvorbu grafického rozhraní a přechodů mezi nimi [32]. Pomocí tohoto nástroje dokážeme vytvořit celé grafické prostředí pouze pomocí kurzoru myši a potřebné prvky si stačí pouze přetáhnout na plátno a připnout je buďto ke straně obrazovky, nebo k dalším prvkům. Storyboardy mají ovšem své nevýhody a mezi programátory je tvorba grafického rozhraní tímto způsobem vnímána spíše negativně z několika důvodů:

- **Přepoužitelnost** – Ve chvíli, kdy chceme storybardy kopírovat nebo přesouvat, musíme je přesunout i s **ViewControllery**, které jsou na ně navázány. Jinými slovy nemůžeme přepoužít samotný **ViewController**, jelikož má vazbu na daný Storyboard a s ním spojené funkce.
- **Datový tok** – V mnoha situacích potřebujeme přesouvat data mezi více **ViewControllery**. Storyboardy se dokáží postarat o přechod mezi **ViewControllery**, ale ne o tok dat mezi nimi. Cílové **ViewControllery** tedy musí být nakonfigurovány s kódem, přepisujícím vizuální reprezentaci.
- **Verzování** – Pokud používáme některý z verzovacích nástrojů, jako je například *Github*, změny ve storyboardech jsou těžko čitelné. Pro ostatní členy týmu je obtížné vyznat se v úpravách, které byly provedeny.

Naštěstí společnost *Apple Inc.* přišla v roce 2019 s frameworkem **SwiftUI**. **SwiftUI** používá deklarativní syntax a tvorba grafického rozhraní se stává mnohem jednodušší. Systém *WatchOS* je také použitím **SwiftUI** daleko robustnější a umožňuje vytvářet rozsáhlejší aplikace, než tomu bylo u **Storyboardů**. Dokonce i v dokumentaci frameworku *WatchKit* se dočteme, že bychom měli pro tvorbu aplikací na systém *WatchOS* silně zvážit použití *SwiftUI*. Na druhou stranu s sebou *SwiftUI* přináší také několik problémů, které bylo potřeba vyřešit. Jedním z nich je přesměrování mezi obrazovkami.

Přesměrování mezi obrazovkami – Routování

Přesměrování mezi obrazovkami nebo také **routování** se zaměřuje na oddělení logiky přesunu mezi obrazovkami do samostatných komponent.

Hlavním důvodem je to, abychom zachovali *princip jedné odpovědnosti*. V aplikacích pro chytré telefony, nebo pro chytré hodinky je přesun na další obrazovku velmi častý. Je to převážně kvůli velikosti displeje, jelikož máme omezený prostor pro zobrazení veškerých vizuálních prvků a neměli bychom uživatele zahlcovat tolika informacemi najednou.

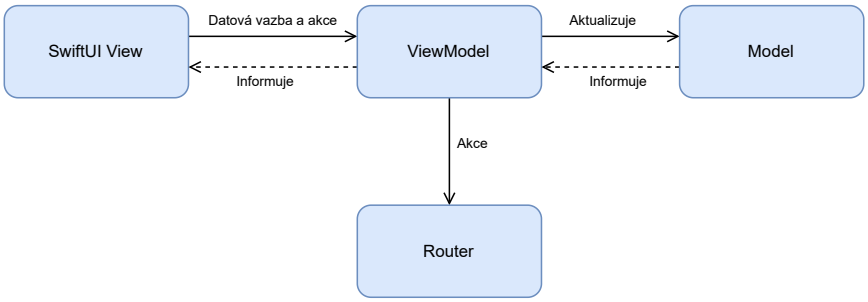
Abychom se zbavili logiky přechodu z **View**, je možné tuto logiku nechat na **ViewModelu**. Lepším řešením je ovšem použít **ViewModel** pouze jako prostředníka, který ve chvíli, kdy má nastat přechod, zavolá funkci jiné komponenty. Tuto komponentu nazveme **router**.

Router má na starosti právě přechod mezi obrazovkami a obsahuje logiku nutnou pro to, aby dokázal rozhodnout, kam přechod nastane. Abychom ušetřili **ViewModel** od další logiky, vystaví **router** pouze rozhraní, které má funkce pojmenované tak, aby obsahovaly název akce, například *onButtonTap()*. **ViewModel** při každém kliknutí na příslušné tlačítko zavolá funkci **routeru** *onButtonTap()* a **router** se již sám rozhodne, co se v tuto chvíli stane.

Abychom si dokázali lépe představit spojení architektury MVVM a **router**, vidíme jejich vazbu na obrázku 3.6.

Jelikož bude mít aplikace velké množství prezentačních modulů, které budou mít vždy vlastní **router**, nebylo by dobrým řešením, abychom v každém **routeru** implementovali logiku přechodů zvlášť. To by vedlo do stavu, kde by naše aplikace obsahovala velké množství duplicit. Z toho důvodu provedeme ještě jednu úpravu, a to takovou, že přidáme třídu `MainRouter`.

`MainRouter` bude vědět o všech možných cestách a stavech, do kterých se v aplikaci bude možné dostat. Každý samostatný **router** tedy nebude muset znát logiku přesměrování, ale zavolá pouze příslušnou funkci třídy `MainRouter`, vystavenou v jejím rozhraní. Tímto se zbavíme duplicit mezi jednotlivými **routery**. Třída `MainRouteru` bude tedy hlavní ovládací komponentou celé aplikace, jelikož bude rozhodovat o tom, která obrazovka se zobrazí jako první.



Obrázek 3.6: Architektura MVVM s komponentou Router

V předchozí podsekci 3.1.2 jsme zmínili knihovnu *Stinsen*. Ta nám nyní umožní oddělit logiku přechodu z **View** přímo do třídy `MainRouter`, která může mít například takovéto rozhraní:

```
protocol MainRouter: AnyObject {

    func onCancel(_ callback: (() -> ())?)
    func showSignInScreen()
    func showDroneDetailScreen()
}
```

Ukázka kódu 3.6: Rozhraní třídy MainRouter

Dále si `MainRouter` drží informaci o třídě `ContainerBuilder`, kterou jsme zmínili v podsekci *Swinject* 3.1.2. Tudíž při každém přechodu dokáže pomocí DI získat instanci prezentačního modulu a ten zobrazit. Ukázka části implementace třídy `MainRouter` lze vidět na následující ukázce 3.7:

```
final class MainRouterImpl: NavigationCoordinatable {

    // MARK: - Properties
    let stack = NavigationStack(initial: \MainRouterImpl.screen)
    let container: Container!

    // MARK: - Routes
    @Root var screen = splashScreenView
    @Root var signInScreen = signInScreenView
    @Route(.push) var droneDetailScreen = droneDetailScreenView
    .
    .
}

// MARK: - View builders
private extension MainRouterImpl {

    @ViewBuilder func signInScreenView() -> some View {
        container.resolve(SignInScreenView.self)!
    }
}

// MARK: Protocol conformance
extension MainRouterImpl: MainRouter {

    func showSignInScreen() {
        root(\.signInScreen)
    }
}
```

Ukázka kódu 3.7: Implementace třídy MainRouter

3.1.7 Výsledná podoba architektury

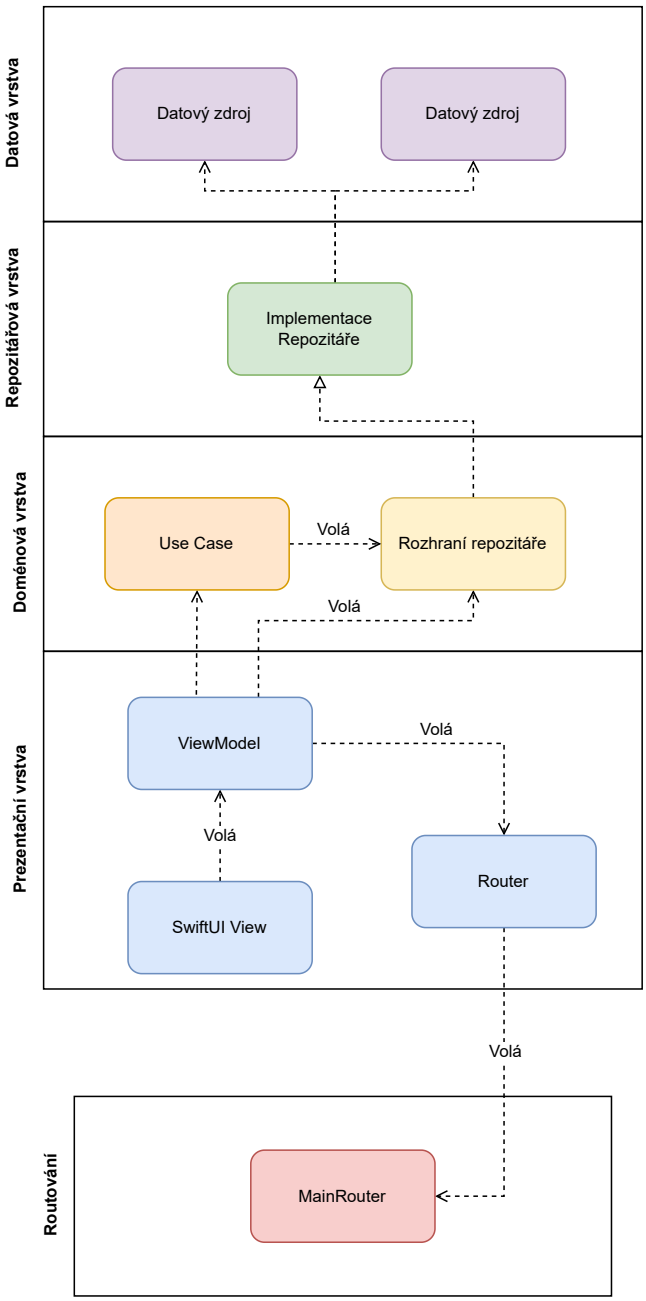
Nyní bychom již měli mít představu o všech vrstvách a částech architektury. Posledním krokem je pospojovat je všechny do jednoho celku a představit si celkovou komunikaci v rámci těchto vrstev.

Nejvhodnější formou vizualizace je diagram 3.7. Vidíme, že prezentační vrstva odpovídá architektuře MVVM, kde má třída `ViewModel` vazbu na doménovou vrstvu a na `Router`, který volá funkce poskytnuté hlavní třídou `MainRouter`.

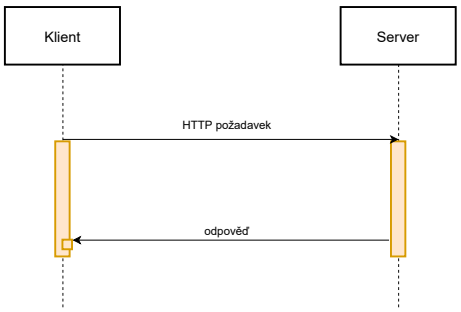
`MainRouter` má povědomí o veškerých možných přechodech mezi obrazovkami a cesta, která v něm není definována nelze uskutečnit.

`ViewModel` může mít vazbu na větší množství repozitářů nebo na několik `UseCase` tříd. `UseCase` navíc může používat opět několik repozitářů.

Rozhraní pro repozitář implementují třídy v repozitářové vrstvě a ty dále mohou využívat více datových zdrojů.



Obrázek 3.7: Architektura – komunikace všech vrstev dohromady



Obrázek 3.8: HTTP – Komunikace mezi klientem a serverem

3.2 Popis API

Implementace API nebyla součástí této práce. Pro pochopení kontextu je ale dobré si vysvětlit, která API jsou dostupná a jaké jsou endpointy, kterých aplikace pro chytré hodinky využívá.

Společnost *Dronetag s.r.o.* poskytuje vývojářům dva druhy API, viz [33]:

- **Dronetag API**

  Jedná se o primární API a považuje se za centrálního poskytovatele dat. Tato služba se zaměřuje na poskytování statických dat. Jedná se například historii letů, registrovaná zařízení atd.

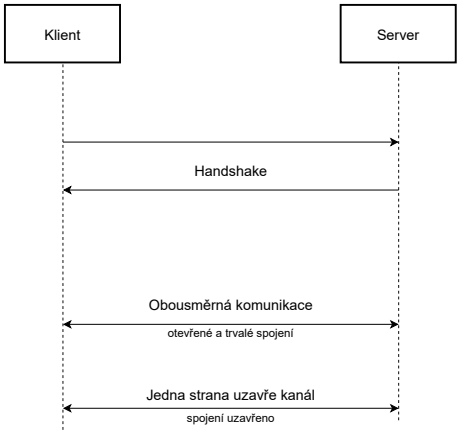
- **Dronetag Live Service API**

  Toto API oproti předchozímu poskytuje data v reálném čase. Neposkytuje však pouze standardní HTTP API, ale také umožňuje použití protokolu *WebSocket* pomocí knihovny *Socket.IO*. Prostřednictvím tohoto API jsme schopni přijímat telemetrická data nebo aktuální stav zařízení.

Websocket

Komunikace mezi mobilní aplikací a webovou službou je většinou řešené prostřednictvím protokolu *HTTP*, kde klient (mobilní aplikace) odešle *HTTP* požadavek na server, server požadavek zpracuje, odpoví klientovi a uzavře spojení. Server bez požadavku nemá možnost posílat cokoliv klientovi. Diagram můžeme vidět na obrázku 3.8.

U jiných typů aplikací jako například u chatovacích aplikací potřebujeme, abychom získávali data v reálném čase. *WebSocket* umožňuje, aby klientská strana otevřela spojení se serverem a udržovala ho [34]. *WebSocket* je Transmission Control Protocol (TCP) spojení mezi klientem a serverem, které umožňuje



Obrázek 3.9: WebSocket – Komunikace mezi klientem a serverem

plně duplexní komunikaci [35]. To znamená, že data mohou být přenášena oběma směry současně, viz obrázek 3.9.

WebSocket může poskytovat několik kanálů, ze kterých může klient odposlouchávat, nebo naopak posílat data zpět serveru.

3.2.1 Dronetag API

Nejprve se podíváme na primární REST API. Požadavky, které bude aplikace posílat, bychom mohli rozdělit do těchto skupin:

- **přihlášení**
- **profil uživatele**
- **detail letu**
- **registrace zařízení pro posílání push notifikací**

Přihlášení

Nejdříve si popíšeme přihlášení, jelikož bez něj není možné provádět v aplikaci žádné další akce. Přihlásit se můžeme několika způsoby, prvním způsobem je, že aplikace získá obnovovací token (**refresh token**) z přidružené mobilní aplikace, v tom případě potřebuje získat přístupový token (**access token**).

Access token má vždy pouze krátkodobou platnost v řádu desítek minut. Je tedy nutné server požádat o token ve chvíli, kdy jeho platnost vyprší a aplikace potřebuje odeslat některý z požadavků.

K tomu slouží endpoint:

- *POST /auth/jwt/refresh/*

  V těle tohoto endpointu posíláme obdržený **refresh token**. Server nám v odpovědi vrací **access token**, který si aplikace ukládá do lokální databáze do doby, než opět vyprší.

Dalším způsobem přihlášení je pomocí e-mailové adresy a hesla. K tomu slouží endpoint:

- *POST /auth/jwt/token/*

  V těle tohoto endpointu posíláme e-mailovou adresu a heslo. Server nám v odpovědi vrátí jak **refresh token**, tak **access token**. Oba tyto tokeny si aplikace ukládá opět do lokální databáze.

Profil uživatele

Ve chvíli, kdy je aplikace přihlášena, je potřeba zjistit informace o aktuálně přihlášeném uživateli. K tomu slouží endpoint:

- *GET /users/me*

  Tomuto endpointu v hlavičce předáváme validní **access token** a jako odpověď získáme od serveru objekt typu *JSON*, který může vypadat takto [4]:

```
    {
    "id": "7f74af8a-2b3c-4039-83c6-e76ea4028bb0",
    "email": "dusekpetr@icloud.com",
    "is_verified": true,
    "full_name": "Petr Dusek",
    "phone_number": null,
    "uas_operator_id": null,
    "country": "CZ",
    "registered": "2022-03-26T21:34:33.035232Z",
    "default_aircraft": null,
    "owned_devices": [
        {
            "id": "f4f57d11-98ef-4f8f-8207-987e95bd5f6b",
            "serial_number": "APWA0004",
            "name": "Dronetag Mocker",
            "type": "dronetag-mocker"
        }
    ],
    "preferences": { ... },
    "flight_hours_this_month": 0.0
}
```

[4] Některé položky objektu byly kvůli velikosti záměrně vynechány.

Detail letu

Posledním volaným endpointem je:

- *GET /flights/current-flight/{sn}*

  Tento endpoint se používá k získání detailních informacích o probíhajícím letu. Server v odpovědi posílá objekt typu *JSON* obsahující: datum zahájení letu, datum plánovaného ukončení letu, ID letu nebo například informace o letové zóně pro toto zařízení.

Registrace zařízení pro posílání push notifikací

Aby mohl server posílat push notifikace na určitá zařízení, je potřeba tato zařízení registrovat. K tomu slouží endpoint:

- *POST /fcm*

  Tomuto endpointu posáláme v hlavičce validní **access token** a v těle informace o názvu, ID, typu zařízení a registračního id získaného z platformy *Firebase* [36].

3.2.2 Dronetag Live Service API

Toto API narozdíl od předchozího slouží k poskytování dat v reálném čase. Je možné samozřejmě využít REST API, ale vzhledek k tomu, že potřebujeme data v reálném čase, musela by aplikace v nějakém cyklu posílat neustále požadavky na server. To ovšem není ideální řešení. Místo toho tedy využijeme protokolu *Websocket*, který je implementovaný pomocí knihovny *Socket.IO*.

Websocket

Služba se nachází na adrese *https://live.dronetag.app/api/v2/* a aplikace ji využívá k odposlouchání těchto událostí:

- `authenticate`

  Tuto událost posílá klient serveru a pomocí ní se autentizuje. V těle zprávy musí klient poslat přístupový token. Server odpovídá buďto chybou, nebo potvrzením.

- `telemetry`

  Tento kanál slouží pro přijímání telemetrických dat. Uživatel nemusí být pro odposlouchávání těchto událostí autentizovaný, ale může tak přijít o některá soukromá data. Data jsou posílána ve formátu *JSON* jako pole, které může vypadat následovně:

```
[ { "time": "2022-04-10T14:41:30.432000+00:00",
    "location": {
        "latitude": 50.103708028399815,
        "longitude": 14.388017134645914
    },
    "altitude": 97.41424122766924,
    "pressure": 1001.308,
    "altitude_pressure": 99.9,
    "height": -2.469999999999999,
    "velocity": {
        "x": -3.749, "y": -9.198, "z": -0.113
    },
    "uas_id": "AAXX0001" } ]
```

- `flight`

  Posledním využitým kanálem je kanál `flight`, který klienta informuje o změně letu. Může se jednat o zahájení letu, nebo naopak o ukončení. Data jsou opět ve formátu *JSON* a vypadají následovně:

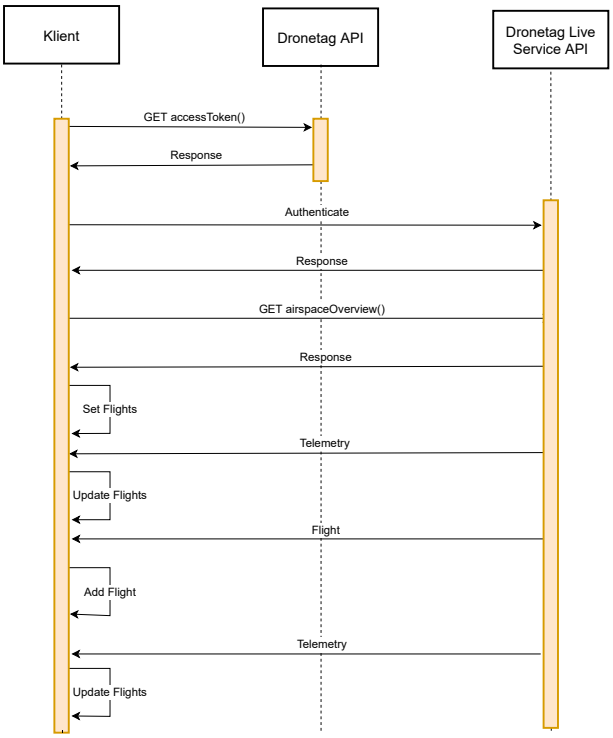
```
{
"flight_id": "797f89e9-7993-45fc-b0b7-1fddcf2d6729",
"uas_id": "AAXX0001",
"update": "started",
"takeoff_lat": 50.103997,
"takeoff_lon": 14.3884453,
"visibility": "public"
}
```

REST API

Jediný endpoint, který využijeme z REST API této služby, je endpoint:

- *GET /airspace/overview*

  Tento endpoint vrací přehled letového prostoru. Obsahuje aktuálně letící letadla s jejich příslušnými telemetrickými údaji. Data mají totožnou podobu jako u události `telemetry`.



Obrázek 3.10: Komunikace klienta se serverem při aktualizaci letů.

3.2.3 Komunikace s API při získávání telemetrických dat

Pro získávání a aktualizace telemetrických dat musí aplikace komunikovat s oběma těmito typy API. Diagram pro komunikaci popisuje obrázek 3.10.

Klient nejdříve získá **access token** z **Dronetag API**, následně autentizuje session pro *WebSocket* a pošle požadavek pro získání přehledu letového prostoru. Z tohoto přehledu si klient uloží seznam letů a následně odposlouchává kanály `telemetry` a `flight`. Jesliže přijde v kanálu `telemetry` aktualizace pro zařízení, které má již uložené v seznamu letů, aktualizuje příslušný let. Pokud se ovšem v seznamu letů nenachází, není možné ho aktualizovat. V případě, že z kanálu `flight` obdrží status pro nějaký nový let, případně status pro ukončení některého zahájeného letu, přidá ho do uloženého seznamu, respektive odebere. Následně opět poslouchá na obou těchto kanálech pro případnou aktualizaci letů.

Tímto způsobem probíhá vždy komunikace mezi těmito službami. Pokud aplikace přechází z pozadí do popředí, je vždy nejprve potřeba získat přehled letového prostoru a až potom tyto lety aktualizovat.

3.3 Datové zdroje

Tato sekce popisuje konkrétněji implementaci datových zdrojů ve vzniklé aplikaci. Jsou popsány použité typy databází, propojení s mobilní aplikací a rozhraní pro komunikaci s webovou službou.

3.3.1 Lokální databáze

Aplikace k ukládání dat používá více typů databází:

- **Keychain**
- **UserDefaults**
- **Core Data**

Keychain

Slouží k bezpečnému uložení dat uživatele [37]. Jde například o hesla, certifikáty nebo kryptografické klíče. Data v této databázi zůstávají uložená i po smazání aplikace. Aplikace má pro tento typ databáze definované rozhraní `SecuredStorage`, kterému odpovídá jeho implementace `SecuredStorageImpl`.

Rozhraní vypadá následovně:

```
protocol SecuredStorage {

    func save(string: String, key: KeychainKey)
    func save(data: Data?, key: KeychainKey)
    func loadString(key: KeychainKey) -> String?
    func loadData(key: KeychainKey) -> Data?
    func delete(key: KeychainKey)
    func deleteAll(_ except: [KeychainKey])
}
```

Ukázka kódu 3.8: Rozhraní pro Keychain – SecuredStorage

Je patrné, že se jedná o databázi typu **klíč–hodnota**, kde klíč je reprezentován výčtovým typem *KeychainKey*. Ukládat můžeme jak text, tak i složitější datové objekty.

UserDefaults

Souží k ukládání výchozích dat uživatele [38]. Tato databáze je opět typu **klíč–hodnota**. Slouží převážně pro ukládání hodnot, které si uživatel v aplikaci nastaví. Při smazání aplikace se automaticky maže i tato databáze. Rozhraním pro tento typ databáze je protokol `KeyValueStorage` s odpovídající implementací `KeyValueStorageImpl`.

Rohzraní vypadá takto:

```
protocol KeyValueStorage {

    func save<T>(object: T, key: DefaultsKey)
    func get(key: DefaultsKey) -> Any?
    func get<T>(key: DefaultsKey, defaultValue: T) -> T
    func remove(key: DefaultsKey)
    func removeAll(_ except: [DefaultsKey]?)
}
```

Ukázka kódu 3.9: Rozhraní pro UserDefaults – KeyValueStorage

Core Data

Nejedná se přímo o databázi, ale o framework určený ke správě grafu objektů [39]. Graf objektů je v podstatě kolekce vzájemně propojených objektů. Pomocí *Core Data* můžeme tento graf ukládat na disk a taktéž máme možnost v grafu vyhledávat. Ve výsledku je tedy možno tento framework použít i jako databáze a lze pomocí něj ukládat objekty a závisloti mezi nimi.

Samozřejmě pro programovací jazyk *Swift* existují další alternativy jako například *SQLite* nebo *RealmSwift*. Tyto alternativy jsou ovšem implementované jako knihovny třetích stran a v tuto chvíli se nám nepodařilo zprovoznit je na systému *WatchOS*.

```
import Combine
import CoreData

protocol CoreDataStorage {

    func save<T: NSManagedObject>(_ entityType: T.Type,
                                  responseModel: Any)

    func get<T: NSManagedObject>(_ entityType: T.Type,
                                 id: String
                                ) -> AnyPublisher<T?, Never>

    func getAll<T: NSManagedObject>(_ entityType: T.Type
                                    ) -> AnyPublisher<[T], Never>

    func deleteAll<T: NSManagedObject>(_ entityType: T.Type)
}
```

Ukázka kódu 3.10: Rozhraní pro Core Data – CoreDataStorage

Výše uvedené rozhraní navíc využívá framework *Combine* a můžeme tedy asynchronně pozorovat změny objektů.

Core Data slouží zatím pouze k ukládání profilu uživatele, jelikož má vazby na další objekty jako například uživatelem vlastněná zařízení. Tyto vazby mezi objekty by bylo složité implementovat u předchozích databází typu **klíč–hodnota**

3.3.2 Mobilní aplikace

Jedním z datových zdrojů je také přidružená mobilní aplikace. V případě, že má uživatel nainstalovanou mobilní aplikaci *Dronetag* a je v ní přihlášen, aplikace na chytrých hodinkách se s ní pokusí spojit a získat **refresh token**.

Ke komunikaci mezi těmito dvěma zařízeními slouží framework *Watch Connectivity* [40].

Watch Connectivity

Watch Connectivity implementuje obousměrnou komunikaci mezi aplikací pro *iOS* a její spárovanou aplikací *WatchOS*. Je možné si tedy mezi těmito dvěma zařízeními vyměňovat zprávy.

Aplikace si mohou mezi sebou posílat zprávy, u kterých se očekává odpověď druhé strany. Spárovaná aplikace na systému *WatchOS* tedy vyšle požadavek a očekává odpověď obsahující **refresh token**.

Při implementaci jsme narazili na problém, kdy se může stát, že jedna ze zmíněných stran nekomunikuje. Je to nejspíše chyba tohoto frameworku, která bude v budoucnu opravena. V tuto chvíli je ovšem potřeba uživateli dát možnost přihlásit se i nezávisle na mobilní aplikaci.

Komunikace s aplikací ve frameworku Flutter

Další omezení přináší implementace přidružené aplikace ve frameworku *Flutter*. Vzhledem k tomu, že se nejedná o nativní implementaci, je potřeba najít řešení, jak spustit kód v přidružené aplikaci.

Flutter však umožňuje volat spefické funkce, závislé na cílové platformě. Tato volání se provádí pomocí třídy `MethodChannel`. V nativní implementaci aplikace vytvoříme tuto třídu se specifkým jménem a následně nad ní vyvoláme metodu:

```
let channel = FlutterMethodChannel(name: "watch_connectivity",
                                   binaryMessenger: messenger)
channel.invokeMethod(type.rawValue, arguments: arguments)
```

Ukázka kódu 3.11: Vyvolání metody ve Flutter z nativní aplikace

Poté je potřeba na straně *Flutter* implementace odchytit toto volání a zpětně vrátit odpověď, která se provolá na straně nativní implementace.

Podrobnější informace můžeme zjistit zde [41]

3.3.3 REST API

Pro posílání požadavků a získávání dat z REST API slouží rozhraní `Networking`:

```
import Combine

protocol Networking {

    func requestModel<T>(
        _ modelType: T.Type,
        spec: RequestSpec
    ) -> AnyPublisher<T, Error> where T: Codable

    func sendRequestWithEmptyResponse(
        _ spec: RequestSpec) -> AnyPublisher<Void, Error>
}
```

Ukázka kódu 3.12: Rozhraní pro REST API

Rozhraní poskytuje dvě funkce, první slouží k poslání požadavku a získání odpovědi, která odpovídá typu posílaném v prvním parametru. Druhá funkce slouží pouze k odeslání požadavku ve chvíli, kdy nás nezajímá odpověď serveru.

RequestSpec

Vidíme, že funkce vyžadují parametr typu `RequestSpec`. Jedná se o protokol, který umožňuje jednodušeji definovat parametry konkrétního požadavku, jako je adresa serveru, cesta k endpointu, metoda, hlavičky atd.

Protokol vypadá takto:

```
protocol RequestSpec {

    var baseUrl: String { get }
    var path: String { get }
    var method: RequestMethod { get }
    var headers: [String: String] { get }
    var queryParams: [String: String] { get }
    var bodyParams: [String: Any] { get }
}
```

Ukázka kódu 3.13: Protokol RequestSpec

3.3.4 WebSocket

Posledním datovým zdrojem je **Dronetag Live Service API**, pro které má aplikace definované rozhraní `LiveServiceDataSource`. Implementace tohoto rozhraní – `LiveServiceDataSource` – využívá knihovnu *Socket.IO* pro programovací jazyk *Swift* [42].

Rozhraní vypadá následovně:

```
protocol LiveServiceDataSource {

    func authorize(
        accessToken: String
    ) -> AnyPublisher<Bool, LiveServiceError>

    func listenForEvents<T>(
        _ modelType: T.Type,
        event: LiveServiceEvent
    ) -> AnyPublisher<T, Never> where T: Codable

    func connect()
    func disconnect()
}
```

3.4 Implementace radaru

Dostáváme se k části popisující samotnou implementaci radaru. Jedná se o hlavní komponentu celé aplikace, jelikož předpokládáme, že většinu času budou uživatelé trávit na této obrazovce.

Ve druhé kapitole jsme si definovali veškeré funkční požadavky, které by měla aplikace splňovat. Většina těchto požadavků se vztahuje právě k radaru.

Implementace radaru byla poněkud obtížnější. Nejdříve jsme zkoušeli radar implementovat čistě pomocí *SwiftUI* frameworku. Narazili jsme ovšem na omezení v podobě posouvatelného *View*, jelikož ve chvíli, kdy umožníme posouvání vertikálním i horizontálním směrem zároveň, nelze vynutit vycentrování na zvolený prvek. Taktéž je obtížnější vkládat objekty na určité pozice. Z těchto důvodů bylo použití *SwiftUI* frameworku pro funkce radaru vyhodnoceno jako nevhodné. Naštěstí existuje framework *SpriteKit*, který je pro tento daný problém vhodnější.

SpriteKit

Framework *SpriteKit* [43] je primárně určený pro tvorbu 2D her. Umožňuje vytvářet animace, detekovat kolize objektů, generovat světelné efekty atd.

My sice nechceme vytvářet hru, ale funkce tohoto frameworku jsou ideální pro vytvoření scény radaru a renderování vizuálních prvků reprezentujících letadla. *SpriteKit* má několik základních prvků [44]:

- `SKScene`

  Veškeré prvky zobrazené pomocí tohoto frameworku se zobrazují pomocí scény, což je instance objektu `SKScene`. Scénu v aplikaci reprezentuje třída `RadarScene`.

- `SKNode`

  Abychom mohli zobrazovat grafické objekty, máme k dispozici uzlový objekt. *SpriteKit* poskytuje několik typů těchto objektů, které jsou zděděné z objektu `SKNode`.

- `SKSpriteNode`

  Objekt reprezentující obrázek nebo barvu.

- `SKShapeNode`

  Objekt reprezentující matematický tvar, který může být vyplněný barvou.

- `SKLabelNode`

  Objekt vykreslující text.

- `SKCameraNode`

  Objekt reprezentující kameru. Tímto objektem definujeme část scény, která je viditelná.

Objekt `SKScene` má také několik základních funkcí, které nás informují o životním cyklu objektu a jež je nutné si představit:

- *sceneDidLoad()*

  Informuje nás ve chvíli, kdy je scéna prezentována.

- *didSimulatePhysics()*

  Informuje nás, abychom po provedení fyzikálních simulací provedli dodatečnou potřebnou logiku.

Prvky do scény vkládáme pomocí funkce *addChild()*. Veškeré grafické objekty jsou vloženy při zavolání funkce *sceneDidLoad()*, ve které navíc proběhne prvotní nastavení všech grafických objektů. Funkce *didSimulatePhysics()* je volána při každém překreslení scény. Slouží tedy pro nastavení pozice kamery, v případě, že sledujeme pohyb našeho dronu nebo uživatele. Dále pro simulaci nekonečného pozadí, a zobrazení postranních indikací.

Třída `RadarScene` má svůj vlastní **ViewModel**, kterým je třída `RadarScreenViewModel`. Ta poskytuje scéně několik `publisherů`, obsahujících informace o pozicích grafických prvků, jako jsou pozice vlastního dronu, pozice uživatele, umístění letové zóny atd. Při zobrazení scény jsou tito vydavatelé odebíráni a aktualizace grafických prvků tedy probíhá asynchronně.

Framework *SwiftUI* umožňuje vložení *SpriteKit* scény a je tedy možné zachovat použití *SwiftUI* i pro samotný radar.

Simulace nekonečného pozadí

Jelikož se prvky posouvají po scéně radaru, je potřeba nějak rozlišit, který z prvků se pohybuje. Toho docílíme přidáním pozadí scény.

Problémem je, že scéna je nekonečná, a jelikož vzdálenost mezi prvky může být v některých případech veliká, vyžaduje to i vytvoření nekonečného pozadí, což není prakticky možné.

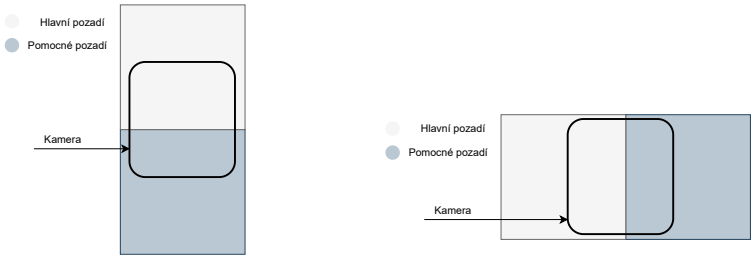
Triviálním řešením je vytváření nových grafických objektů reprezentujících dodatečná pozadí v případě, že se kamera dostane mimo hlavní pozadí. To by ovšem nebylo příliš efektivní a aplikace by tímto způsobem začala v nějakou chvíli využívat velké množství dostupných prostředků. To by mohlo v extrémním případě vyvolat pád aplikace.

Daleko lepším řešením je nekonečné pozadí pouze simulovat a vykreslovat pomocné pozadí pouze ve chvíli, kdy se kamera dostane hlavní mimo pozadí.

Mohou nastat tři případy:

- Kamera se nachází mimo vertikální plochu pozadí
- Kamera se nachází mimo horizontální plochu pozadí
- Kamera se nachází mimo horizontální i vertikální plochu pozadí

V případě, že ze kamera nachází pouze mimo vertikální nebo horizontální plochu, vykreslíme pouze jedno pomocné pozadí. Pomocné pozadí tak vyplní prázdnou plochu, nad kterou přesahuje kamera, viz obrázky 3.11.

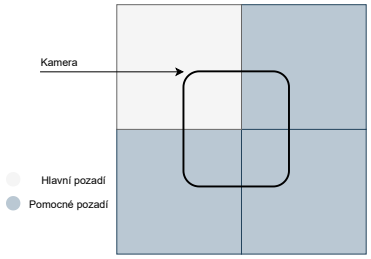


(a) Kamera mimo vertikální plochu

(b) Kamera mimo horizontální plochu

Obrázek 3.11: Vykreslení jednoho pomocného pozadí

V případě, že se kamera nachází mimo vertikální a zároveň i horizontální plochu, je potřeba vykreslit navíc pomocné rohové pozadí, které vidíme na obrázku 3.12.



Obrázek 3.12: Vykreslení více pomocných pozadí

Tímto způsobem vždy vykreslujeme maximálně tři pomocná pozadí, a to jen v případě, že se kamera nachází ve zmíněných hraničních pozicích.

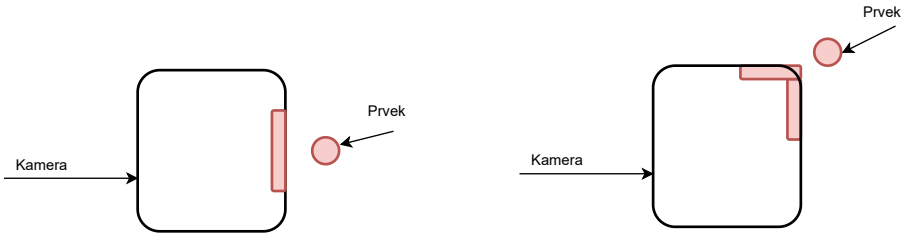
Jakmile se kamera vzdálí od hlavního pozadí o větší vzdálenost, než je plocha jednoho pozadí, hlavní pozadí přesuneme a postup opakujeme.

Postranní indikace

Při pohybu po scéně radaru mohou nastat situace, kdy nevidíme svůj dron, nebezpečné drony nebo pozici uživatele. V tuto chvíli chceme uživatele navést, kterým směrem se má posunout, aby se jednodušeji dostal ke hledanému prvku.

Mohou nastat tyto situace, zobrazené také na obrázku 3.13:

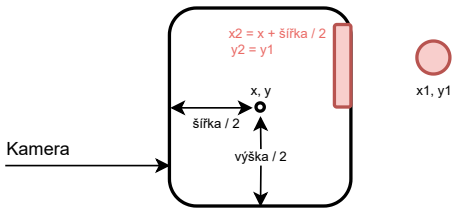
- Prvek se nachází mimo vertikální nebo horizontální plochu kamery.
- Prvek se nachází mimo vertikální a zároveň horizontální plochu kamery.



(a) Horizontální indikace (b) Rohová indikace

Obrázek 3.13: Indikace prvku mimo záběr kamery

Jestliže se prvek nachází pouze mimo vertikální nebo horizontální plochu kamery, je potřeba nejdříve vypočítat, o jakou stranu se jedná. Následně z velikosti plochy kamery a její pozice vypočítáme pozice hranic kamery. Výpočet je složitější, jelikož musíme nejprve rozhodnout, zda se jedná o kladné, nebo záporné hodnoty pro souřadnice na ose X i pro souřadnice na ose Y. Pokud se prvek nachází mimo vertikální i horizontální plochu zároveň, zobrazí se obě tyto indikace najednou. V tu chvíli se indikace překrývají a působí jako rohová indikace. Zjednodušený výpočet vidíme na obrázku 3.14.



Obrázek 3.14: Zjednodušený výpočet postranní indikace

Získání lokace uživatele

Pro získání lokace uživatele slouží v jazyce *Swift* framework *Core Location* [45]. Díky němu můžeme požádat o přístup k lokaci a získávat tak aktuální data o pozici, na které se zařízení nachází.

Systém *WatchOS* žádá uživatele při prvním požadavku o lokaci povolení. Zavoláním funkce *requestWhenInUseAuthorization()* je zobrazen systémový dialog, ve kterém potvrdíme, případně zakážeme přístup k lokaci.

Na systému *WatchOS* je toto chování poněkud odlišné od systému *iOS*. Pokud je aplikace na chytrých hodinkách instalována samostatně, zobrazí se dialog přímo na zařízení. Ovšem pouze jednou. Pokud tedy přístup k lokaci zakážeme, měli bychom mít možnost ho opět povolit v nastavení, kde následně systém neumožňuje výběr žádné z položek. Nejspíše se jedná o chybu v systému, která bude v budoucnu opravena.

Tím, že aplikaci zatím nebudeme poskytovat samostatně, ale pouze jako doplněk k mobilní aplikaci, je chování odlišné. V tomto případě se systémový dialog na hodinkách nezobrazuje a očekává se povolení lokace v přidružené mobilní aplikaci.

Toto chování není z pohledu User Experience (UX) přívětivé, jelikož uživateli nedokážeme otevřít nastavení v mobilní aplikaci z chytrých hodinek. V tuto chvíli je jediným řešením uživatele informovat o cestě k nastavení v systému *iOS* a navést ho k tomu, aby si povolení zapnul.

Přepočet GNSS souřadnic

Z REST API nám přicházejí informace o souřadnicích jednotlivých prvků. Tyto souřadnice získává aplikace v podobě GNSS souřadnic. Ovšem při umístění těchto prvků do 2D prostoru je potřeba tyto souřadnice přepočítat.

Abychom získali absolutní pozice prvků v 2D prostoru, musíme si nejdříve určit nějaký centrální bod. Tímto bodem je vždy počáteční pozice uživatele, kterou aplikace získá při prvním zobrazení radaru.

Pozice uživatele je také získávána v podobě GNSS souřadnic a je tedy potřeba vypočítat absolutní pozici v metrech.

Co znamenají GNSS souřadnice?

Abychom přepočet souřadnic blíže pochopili, je potřeba si nejdříve vysvětlit, co GNSS souřadnice znamenají. Pro lepší představu se podívejme na obrázek 3.15.

Je patrné, že souřadnice na ose Y budou odpovídat zeměpisné šířce. Souřadnice na ose X odpovídají zeměpisné délce, ale pouze ve chvíli, kdy se souřadnice nachází na rovníku. V ostatních případech potřebujeme roznásobit úhel u zeměpisné šířky se zeměpisnou délkou.

Souřadnice na ose X vypočteme následovně (*longitude* je zeměpisná délka a *latitude* zeměpisná šířka):

$$x = longitude * \cos(latitude * \pi/180)$$

Pro přepočítání zeměpisné šířky na délku v metrech použijeme vzoreček pro vypočítání délky oblouku:

$$l = \alpha * r$$

Poloměr země k pólům odpovídá přibližně 6356 km, úhel α je zadaný v radiánech a je roven zeměpisné šířce. Vzdálenost od rovníku k pólu vypočteme dosazením poloměru země k pólům a maximální zeměpisné šířce, která odpovídá 90 stupňům, což je přibližně 1.5707 radiánů.

$$l = 1,57079632679 * 6356$$
$$l = 9983,98145308$$

Vydělením této hodnoty 90 stupni získáme přibližnou vzdálenost pro jeden stupeň zeměpisné šířky:

$$9983,98145308/90 \approx 111km$$

Stejným postupem vypočteme vzdálenost v metrech pro osu X, s rozdílem poloměru země, který je přibližně 6371 km. Vypočtením získáme přibližnou hodnotu jednoho stupně, která odpovídá 111 km.

Tímto postupem získáme absolutní hodnotu souřadnic X a Y pro veškeré prvky umístěné na radaru.

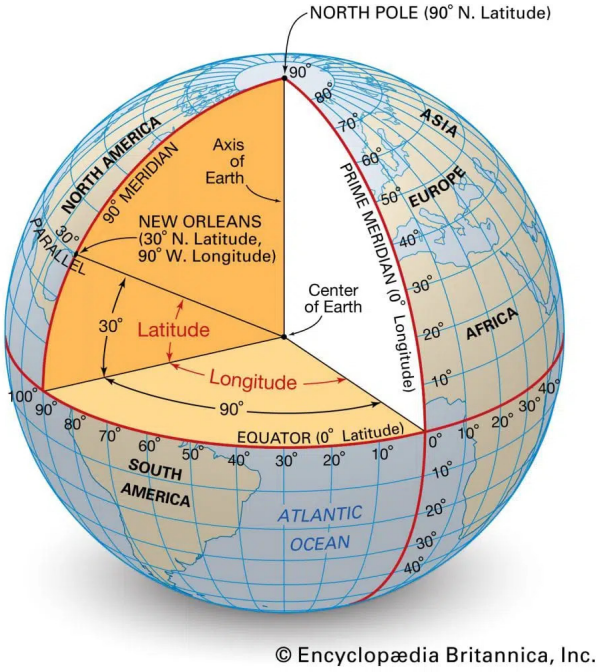
Jelikož přepočtené souřadnice nabývají vysokých hodnot, odečítáme od každé souřadnice souřadnici pro centrální bod, kterým je již zmíněná pozice uživatele. Pozice uživatele je tedy vždy umístěna nejdříve na pozici 0,0 až do chvíle, než se uživatel pohne.

Zobrazení letové zóny

Poslední z důležitých funkcí radaru je zobrazení letové zóny. Letovou zónu si uživatel může buďto přímo navolit při plánování letu, nebo se v ostatních případech vytvoří základní letová zóna automaticky. Automatickou zónu tvoří vždy kruh o poloměru jednoho kilometru z místa vzletu.

Vzhledem k tomu, že si uživatel může letovou zónu navolit čistě podle sebe, nemůže aplikace počítat pouze se základními tvary této zóny. Hranice zóny dostává aplikace v podobně pole GNSS souřadnic. Nejdříve je tedy potřeba tyto souřadnice přepočítat pomocí vzorce z minulé podsekce.

Následně pro vykreslení zóny použijeme třídu *UIBezierPath*, která v jazyce *Swift* slouží pro vykreslování vlastních obrazců.



Obrázek 3.15: GNSS souřadnice, zdroj: [2]

UIBezierPath má tři základní funkce:

- *move(to:)*

  Přesuneme se na pozici danou parametrem, bez toho, abychom kreslili.

- *addLine(to:)*

  Vykreslí rovnou čáru mezi aktuální pozicí a pozicí zadanou v parametru.

- *close()*

  Uzavře cesty vykresleného objektu.

Pomocí těchto funkcí je již vykreslení zóny triviální. Na začátku se vždy přesuneme pomocí funkce *move(to:)* na první souřadnici zóny a následně vykreslíme rovnou čáru mezi dalšími body. Ukončíme funkcí *close()*.

3.5 Implementace mapy

Abychom uživatele nenutili k používání pouze jednoho podkladu, byla aplikace rozšířena i o mapový podklad. Mapový podklad může přijít vhod ve chvíli, kdy pilot letí nad obydlenou oblastí a chce znát podrobnější přehled o budovách nebo pozicích, na kterých se jeho dron nachází.

Mapa v systému WatchOS

Systém *WatchOS* donedávna neumožňoval implementovat mapu, s příchodem frameworku *SwiftUI* to je však již možné. Oproti mapě, kterou lze implementovat na zařízení *iOS* pomocí frameworku *UIKit*, je funkcionalita však omezenější a zatím nejsou dostupná rozšíření pro složitější účely.

Jedním z omezení je nemožnost vykreslování obrazců přímo do mapy. Tuto funkcionalitu je možno obejít vytvořením anotace místa, která by měla potřebný tvar. Bylo to také jedno z řešení, kterým jsme implementovali letovou zónu do mapy, ale následně jsme tuto funkci zakázali, jelikož nastával jeden důležitý problém.

Mapa se v případě, že má na jednom místě umístěných více prvků, rozhoduje sama o tom, který prvek zobrazí na vrchu. Hlavním kritériem je vertikální pozice prvku a toto kritérium bohužel zatím nelze změnit. Ve chvíli, kdy se tedy nějaký prvek nachází svou vertikální pozicí nad zobrazenou zónou, není možné na něj klikat a zároveň je překrytý. Toto chování poněkud znemožňuje využití mapy a z toho důvodu bylo vykreslení zóny v tuto chvíli na mapě zakázáno.

Umístění prvků do mapy

Umístění prvku do mapy je oproti implementaci radaru značně jednodušší, jelikož mapa umožňuje vkládat prvky přímo podle jejich GNSS souřadnice a není tedy potřeba tyto souřadnice nijak přepočítávat.

Přepínání centrování

Přepínání centrování opět není nijak složité. Jedním z parametrů při inicializaci třídy reprezentující mapu je vykreslovaný region. Tento region má definovanou středovou pozici a při změně této pozice se taktéž posouvá kamera zobrazující mapu. Při každém posunu prvku, který chceme sledovat, stačí pouze změnit tento region.

Volba mezi mapou a radarem

Mezi mapovým podkladem a radarem se může uživatel rozhodnout sám v nastavení. Ve výchozí pozici je vždy zobrazován radar, jelikož mapa má v tuto chvíli stále omezené funkce, a je tedy považována zatím spíše za jakýsi prototyp.

Abychom nekopírovali vzhled tlačítek pro centrování a stále zachovali principy dobrého návrhu, jsou mapa i radar implementovány jako samostatné komponenty, které jsou následně při výběru předány hlavní obrazovce.

Hlavní obrazovka má za úkol autentizovat použité webové služby a změnu centrování, kterou následně deleguje vybrané komponentě, kterou je buď to radar, nebo mapa.

3.6 Podpůrné obrazovky

Tato sekce se zabývá popisem dalších obrazovek, které rozšiřují aplikaci a poskytují uživateli dodatečné funkce. Jedná se o nezbytnou část aplikace, jelikož je potřeba řešit například přihlášení, nastavení atd.

Úvodní obrazovka

Úvodní obrazovka slouží k získání obnovovacího tokenu uživatele, přihlášeného v přidružené mobilní aplikaci. Při získávání tohoto tokenu mohou nastat čtyři různé stavy:

1. Mobilní aplikace není dostupná a aplikace na systému *WatchOS* zatím nemá žádný obnovovací token.
2. Mobilní aplikace není dostupná, ale aplikace na systému *WatchOS* již v minulosti získala obnovovací token.
3. Mobilní aplikace je dostupná, ale uživatel v ní není přihlášen.
4. Mobilní aplikace je dostupná a uživatel se v ní již přihlásil.

Pokud nastanou stavy 1 nebo 2, je uživatel přesměrován na obrazovku, která ho informuje o nastalé chybě. Dále je možné pokračovat kliknutím na tlačítko *Retry*, které spustí opětovný požadavek pro získání obnovovacího tokenu. Jestliže je požadavek spuštěn po třetí, nastane automatické přesměrování na obrazovku přihlášení.

Pokud nastanou stavy 2 či 4, je provedeno přesměrování buďto na obrazovku informující o všech funkcích aplikace, které má uživatel dostupné, nebo na hlavní obrazovku obsahující radar, případně mapu. Podklad závisí na předchozí volbě uživatele.

Přihlášení

Jestliže si uživatel v nastavení vypne synchronizaci s přidruženou mobilní aplikací nebo nastane jedna z výše zmíněných chyb, je odkázán na obrazovku samostatného přihlášení.

Přihlášení je možné provést zadáním uživatelského jména a hesla. Jelikož psaní textu na chytrých hodinkách je v tuto chvíli omezené a nemáme k dispozici standardní klávesnici, je možné provést zadání údajů ze zařízení *iOS*.

Profil

Profil je jednou z podpůrných obrazovek a informuje uživatele o jeho e-mailu, verzi aplikace a nabízí mu možnost odhlášení. Dále také funguje jako prostředník k přechodu na další obrazovky, jako jsou například obecná nastavení nebo nastavení vizuálních prvků.

Nastavení vizuálních prvků

Nastavení vizuálních prvků je jedna z položek obrazovky profilu. Jsou zde nastavení pro různá chování aplikace tak, aby si uživatel mohl aplikaci přizpůsobit dle svých vlastních preferencí.

Zapnutí nebo vypnutí prvků je provoláváno pomocí frameworku *Combine* a to asynchronně. Ve chvíli, kdy uživatel změní některé z nastavení a vrátí se na hlavní obrazovku zobrazující buďto radar, nebo mapu, jsou mu zobrazeny pouze vizuální prvky, které si sám nastavil.

3.7 Notifikace

Cílem práce bylo také upozorňovat uživatele na potenciální hrozby formou notifikací. Notifikace jsou několika typů:

- vizuální indikace,
- zvuková indikace,
- indikace pomocí vibrací,
- push notifikace.

Vizuální indikace

Vizuální indikace probíhá pomocí animací, které se spustí ve chvíli, kdy se některé z cizích letadel ocitne v uživatelem definované blízkosti.

Zvuková indikace a indikace pomocí vibrací

Zvuková indikace je na systému *WatchOS* řešená triviálně. Jedná se o napsání pouze tohoto řádku kódu:

```
WKInterfaceDevice.current().play(.notification)
```

Tímto způsobem se spouští jak zvuková indikace, tak i indikace pomocí vibrací. Tato indikace nastává ve stejnou chvíli jako vizuální indikace a její chování lze volitelně nastavit.

Push notifikace

Push notifikace jsou řešení pomocí platformy *Firebase* [36]. Platforma *Firebase* poskytuje velké množství funkcí, jedná se například o logování pádů aplikace, poskytování statistik o počtu uživatelů nebo například posílání push notifikací. *Firebase* je možné přidat do projektu jako rozšiřující knihovnu. Zavoláním funkce *FirebaseApp.configure()* při inicializaci projektu vytvoříme novou instanci a zahájíme tak i logování chyb aplikace.

Pro zasílání push notifikací je nejprve potřeba získat potřebná práva. O to se stará třída `UNUSerNotificationCenter`. Při spuštění aplikace je tedy potřeba implementovat následující kód:

```
let authOptions: UNAuthorizationOptions = [.badge, .sound]
UNUserNotificationCenter.current().requestAuthorization(
    options: authOptions,
    completionHandler: { _, _ in }
)
```

Ukázka kódu 3.14: Vyžádání práv pro zasílání push notifikací

Dále je v aplikaci implementovaná třída `NotificationHandler`, která obsluhuje notifikace. Tato třída je přiřazená jako delegát pro notifikace chodící z platformy *Firebase* a je také delegátem pro systémové notifikace. Přiřazení těchto delegátů v implementaci vypadá takto:

```
Messaging.messaging().delegate = self
UNUserNotificationCenter.current().delegate = self

WKExtension.shared().registerForRemoteNotifications()
```

Ukázka kódu 3.15: Přiřazení delegátů pro push notifikace

Následující funkce, kterou provolává zmíněný delegát, registruje příslušné zařízení na server, jenž následně posílá push notifikace tomuto zařízení:

```
func messaging(_ messaging: Messaging,
    didReceiveRegistrationToken fcmToken: String?) {
    guard let fcmToken = fcmToken else { return }
    securedStorage.save(string: fcmToken, key: .fcmToken)
    userSessionRepository.setFcm(fcmToken)
}
```

Ukázka kódu 3.16: Registrace push notifikací na server

Aplikace si také ukládá obdržený token z platformy *Firebase*, který by dále mohla využít.

Důvodem, proč aplikace na systému *WatchOS* přijímá push notifikace samostatně, je to, že pokud bychom zařízení samostatně neregistrovali, *Apple* ekosystém by se rozhodoval sám, na které zařízení bude notifikace posílat.

Cílové zařízení, na kterém uživatel obdrží notifikace, je ekosystémem vybíráno podle toho, které zařízení aktuálně používá. Pokud tedy pracuje na svém mobilním telefonu, obdrží notifikace na něm. V opačném případě pokud je displej mobilního telefonu zhasnutý, obdrží notifikaci na chytrých hodinkách. To by ovšem způsobovalo posílání notifikací ve většině případů pouze na mobilní telefon, jelikož uživatelé dronů při létání používají většinou různé aplikace třetích stran, určené pro sledování záznamu z kamery jejich dronu. Je tedy potřeba posílat notifikace samostatně na systém *WatchOS*.

Kapitola 4

Testování

Tato kapitola se zabývá popisem testování vzniklé aplikace. Testování probíhalo několika způsoby. Pro nezbytné a hlavní funkce systémy vznikly jednotkové testy, které v tuto chvíli mají pouze minimální pokrytí. Následně bylo hotové řešení otestováno s účastníky, kterým byla aplikace představena, a byly jim zapůjčeny chytré hodinky s nainstalovanou aplikací, na nichž si mohli vše potřebné vyzkoušet. Toto uživatelské testování bylo rozděleno na dvě fáze, které jsou podrobněji popsané v sekcích níže.

4.1 Jednotkové testy

Jednotkový test je způsob testování, kdy testujeme pokud možno co nejmenší část kódu, kterou lze v systému logicky izolovat. Ve většině programovacích jazyků je to funkce, podprogram, metoda nebo vlastnost [46]. Důležité je, aby testovaná jednotka byla izolovaná.

Testovaná jednotka může být v podstatě cokoliv – řádek kódu, metoda nebo celá třída. Čím menší testovaná jednotka, tím lépe. Jednotkové testy pomáhají lépe odhalit chyby v systému.

V jazyce *Swift* definujeme třídu pro testovací případ tak, že dědí od třídy `XCTestCase` a každá funkce, která reprezentuje jednotkový test, musí začínat slovem `test`. Vývojové prostředí *Xcode* poté dokáže tyto testy odhalit a spustit.

Aplikace má v tuto chvíli pokrytí jednotkovými testy opravdu minimální. Jednotkové testy zahrnují pouze klíčové funkcionality, mezi které patří například přihlášení, získání dat z přidružené mobilní aplikace a otestovaný repozitář, který zajišťuje veškerá data ohledně dronů.

V budoucnu by bylo dobré jednotkové testy rozšířit také pro další části systému a vyhnout se tak situacím, kdy by nová funkcionalita mohla rozbít některou stávající funkční část systému.

4.2 Uživatelské testování

V závěru byly provedeny testy s uživateli, jejichž cílem bylo ověřit použitelnost výsledné aplikace. Uživatelské testování bylo provedeno dvoufázově. V první fázi proběhlo testování s pěti účastníky různého charakteru, kteří buď to měli nějakou zkušenost s drony, nebo naopak s drony v minulosti nepřišli do styku. V druhé fázi proběhlo testování s lidmi, kteří pracují v oboru letectví a mají na tuto problematiku širší pohled.

4.2.1 Testování s různými účastníky

Testování bylo prováděno na místě za účasti pěti testerů. Byla jim představena aplikace, její účel a následně si každý uživatel mohl aplikaci vyzkoušet a dát zpětnou vazbu na jednotlivé funkce a celkové chování.

- **Tester č. 1**
  - Aplikace přestávala vibrovat až po návratu na obrazovku s radarem.
  - Textace u nastavení vizuálních prvků nebyla občas srozumitelná.
  - Mohly by být změněny tvary tlačítek pro centrování tak, aby méně překrývaly obrazovku.
  - Přibližování na radaru se chovalo opačně, než byl uživatel zvyklý.
- **Tester č. 2**
  - Uživateli bylo potřeba podrobněji vysvětlit, co které prvky na radaru znamenají.
  - Vibrace by bylo dobré zrušit ihned při přepnutí nastavení.
  - Aplikace nevibrovala při zobrazení mapy.
  - U popisku „heading“ nebylo srozumitelné, co to znamená.
  - Prvky vizuálních nastavení by mohly obsahovat nápovědu, kde se uživatel dozví, o jaký prvek se jedná.
  - Uživatel by zvážil klikání na krajní indikace, které by ho následně vycentrovalo na příslušný prvek.
  - Místo digitální korunky se snažil opakovaně používat gesta pro přiblížení jako u mobilních zařízení.

- **Tester č. 3**
  - Uživatel měl předchozí zkušenosti s hodinkami *Apple Watch* a dle jeho slov mu připadala aplikace jednoduchá a intuitivní.
  - Neměl žádné problémy s ovládáním ani další nápady na zlepšení.
- **Tester č. 4**
  - Vibrace by si podle jeho slov vypnul, jelikož se mu zdály obtěžující.
  - Některé vysvětlivky u nastavení nejsou z textace srozumitelné.
  - Přibližování na radaru se chovalo opačně, než by očekával.
  - Upřednostnil by mapový podklad před radarem.
  - Aplikace se mu používala velmi dobře.
- **Tester č. 5**
  - Funkce tlačítek pro centrování není zpočátku pochopitelná.
  - Při zvoleném mapovém podkladu není jasné, jak daleko od sebe se prvky nachází.
  - Chyběly stupně u směru pohledu jednotlivých letounů.

Shrnutí

Tato část testování byla velice přínosná, jelikož bylo nalezeno několik problémů, které by mohly uživatelům ztížit práci s aplikací. Všichni účastníci tohoto testování uvedli, že se jim s aplikací pracuje velmi dobře a neměli žádné výrazné problémy s pochopením aplikace ani s jejím ovládáním.

Obrazovka s přehledem funkcionalit při spuštění se ukázala jako velice dobré řešení, které účastníkům usnadnili práci s aplikací. Díky tomu nebylo potřeba žádné další vysvětlení funkcionalit.

Zásadní nedostatky, které byly vyhodnoceny a následně opraveny, jsou:

1. Aplikace vibrovala i po odchodu z radaru.
2. Aplikace nevibrovala vůbec při volbě mapového podkladu.
3. Textace u některých funkcionalit nebyla pochopitelná.
4. Přibližování na radaru se chová opačně, než je zvykem.
5. Prvky vizuálních nastavení by mohly obsahovat nápovědu.

4.2.2 Testování s lidmi z oboru letectví

Ve druhé fázi testování jsme chtěli získat zpětnou vazbu od lidí, kteří se v letectví pohybují. Poptali jsme tři účastníky, se kterými jsme se postupně sešli, předvedli jim vzniklou aplikaci a popsali jim její účel. Zajímala nás jejich reakce a celková zpětná vazba, případně nějaká budoucí rozšíření a funkce, které by oni sami využili v budoucnu.

Profesionální pilot dronů

První testování proběhlo s profesionálním pilotem dronů. Aplikace se mu na první pohled líbila, narazili jsme však na funkcionality, které by rád v budoucnu uvítal, aby se při pohledu na aplikaci dokázal rychleji zorientovat v prostoru.

- **Směrování kamery**

  Účastník uvedl, že by pro orientaci na radaru bylo dobré, aby se kamera otáčela stejným směrem, kterým je otočený vycentrovaný prvek.

- **Zobrazení dodatečných informací přímo na radaru**

  Dále by rád viděl dodatečné informace jako například výšku a váhu letadel nebo také vzdálenost svého letadla od ohrožujících letadel. Pomohlo by mu to se lépe a rychleji zorientovat a zjistit, co se v danou chvíli děje a jestli je tato informace pro něj relevantní.

- **Seznam letadel**

  Při kliknutí na více dronů najednou by také rád věděl, které z letadel je jeho vlastní, a uvítal by jejich setřídění tak, aby na prvních místech byla zobrazena ohrožující letadla. U jednotlivých letadel by dále mohly být informace o váze, vzdálenosti nebo typu a položky by mohly být podbarvené barvou stejně, jako je tomu u radaru.

- **Různé stupně upozornění**

  Účastník uvedl, že by bylo dobré zvolit si dva různé stupně upozornění. Při prvním stupni by aplikace informovala o nebezpečí pouze jednou a šlo by pouze o informování uživatele, že se ve větším okolí jeho dronu pohybuje nějaké další zařízení. Tento první stupeň by však nebyl ohrožující natolik, aby se tím uživatel musel v danou chvíli zabývat.

  Druhý stupeň upozornění by byl naopak kritický a znamenalo by to, že se v okolí pohybuje letadlo, které je už velmi blízko, a uživatel musí rychle jednat, aby nedošlo ke srážce.

Účastník neměl celkově žádné větší problémy s používáním aplikace. Prvotní obrazovka s tipy mu pomohla rychle se zorientovat v celé aplikaci a dokázal bez další pomoci objevit způsob, jakým se dostat do nastavení.

Aplikace by dle jeho názoru mohla v budoucnu sloužit i jako přehledová aplikace pro lidi, kteří chtějí pouze zjistit, co se v jejich okolí pohybuje za letadlo a komu patří. Tato skutečnost by ovšem mohla vést ke zneužívání aplikace lidmi, kteří létání s drony odsuzují, a mohl by jim být předán do rukou nástroj, díky kterému budou na rekreační piloty podávat trestní oznámení například za překročení hranic pozemku. Tomuto bychom se však chtěli spíše vyhnout.

Vedoucí Ústavu letecké dopravy ČVUT

Druhého testování se účastnil vedoucí Ústavu letecké dopravy ČVUT. Kladli jsme otázky, které zazněly v předchozím testování, a také na celkový pohled na aplikaci. Účastník tohoto testování měl opět trochu jiný pohled na celkovou problematiku, tudíž i toto testování bylo velice přínosné.

- **Směrování kamery a světové strany**

  Účastník uvedl, že by se kamera mohla otáčet směrem, kterým je otočený vycentrovaný prvek, aby došlo k rychlejšímu rozeznání, kterým směrem se má uživatel otáčet. Dále při aktuálním stavu by bylo dle jeho slov dobré uživatele informovat o světových stranách. Například zobrazovat vždy na horním okraji displeje informaci, že se jedná o sever. Dále uvedl, že u mapového podkladu mu tato informace nechybí, jelikož v tomto případě je směrování srozumitelné.

  Jedním z nápadů bylo otáčet kameru tak, aby byl uživatelův dron umístěný vždy na severu. Tento nápad by však bylo potřeba otestovat a domyslet celkové chování.

- **Dodatečné informace**

  Aplikace by mohla zobrazovat dodatečné informace o výšce letadel nebo například o směru narušitele. Zde nastává otázka, zda tento směr zobrazovat od polohy uživatelova dronu, nebo naopak od polohy uživatele.

  Vzdálenost mezi spojnicí by nejspíše měla být pouze horizontální, jelikož při započtené výšce by tento údaj mohl být zmatečný.

  Ve stavu, kdy uživatelův dron ohrožuje nějaké jiné letadlo a jsou v těsné blízkosti, by mohla aplikace informovat o rozdílu jejich výšek.

- **Seznam letadel**

  Uvedl, že seznam letadel bude aplikace zobrazovat pouze v krajních případech a je velice malá pravděpodobnost, že by tato situace nastala. Ve chvíli, kdy by opravdu tato situace nastala, mohly by být prvky podbarvené stejnou barvou, jakou jsou obarvené na radaru, a dále by u nich mohla být uvedená výška a hmotnost.

  Samotné řazení prvků nepovažuje za důležité.

- **Různé stupně upozornění**

  Rád by uvítal možnost vypnout si notifikace o ohrožení pro konkrétní prvky na radaru. Ve chvíli, kdy by například létal s někým, o kom ví, nepotřeboval by dostávat upozornění kontinuálně. Také by ohrožující drony mohly být mezi sebou odlišeny různými barvami, aby ve chvíli, kdy má vypnuté upozornění pro drony v okolí, zjistil, že se v blízkosti objevil nový dron, a nejedná se o dron, který tam již byl.

  Funkcionality dvou různých stupňů upozornění si taktéž dokázal představit a byla by pro něj více užitečná, než je tomu v aktuální chvíli, kdy má aplikace pouze jednu hranici pro upozornění.

V tuto chvíli nevidí aplikaci jako nutně potřebnou pro všechny piloty dronů, jelikož dronů létá málo. To ale neznamená, že aplikace nyní nebude mít svou cílovou skupinu. Ve chvíli, kdy by byl vzdušný prostor více přeplněný, by aplikaci uvítal, jelikož by ve chvíli, kdy se v jeho okolí objeví nějaké další zařízení dostal notifikaci a dokázal se lépe zorientovat. V tomto případě by si dle jeho slov aplikaci ihned stáhnul, avšak není uživatelem zařízení od firmy *Apple Inc.*, a tudíž by uvítal toto rozšíření i na ostatní systémy pro chytré hodinky.

Případ, kdy by uživatel aplikaci využíval jen pro svou zvědavost, aby sledoval letadla v okolí, si nedokáže přímo představit. V této situaci by spíše využil mobilní aplikaci.

Prezident české Aliance pro bezpilotní letecký průmysl

Posledním účastníkem testování byl prezident české Aliance pro bezpilotní letecký průmysl. Opět jsme nechali účastníka s aplikací pracovat nejdříve samostatně, vysvětlili jsme mu kontext celé aplikace a následně jsme položili otázky související s předešlými testováními.

- **Upozornění**

  Upozornění o ohrožujícím dronu by mohla brát ohled na to, jakým směrem se ohrožující dron pohybuje. Ve chvíli, kdy by se objevilo zařízení ve velké vzdálenosti a zároveň by toto letadlo letělo směrem pryč, nedává toto upozornění příliš velký smysl.

- **Dodatečné informace**

  Informaci o výšce vnímá jako sekundární. Nevidí tedy nutnost zobrazovat výšku vždy. Podstatná je podle něj horizontální vzdálenost. Hmotnost z jeho pohledu také nehraje příliš velkou roli. Mohou to být informace, které si uživatel sám nastaví, ale nevnímá to jako výchozí nastavení.

- **Směrování kamery**

  Otáčení kamery by mohla být zajímavá funkcionalita do budoucna, ale dle jeho slov by to nemělo být primární nastavení. Uživatel by měl mít možnost si tuto funkci zapnout. Například aplikace *DJI GO* má také vždy sever zobrazený nahoře, tudíž by uživatelé měli být na toto chování zvyklí.

  Dle jeho názoru stačí na radaru zobrazit informaci o severu, kdyby aplikace zobrazovala vlastní dron vždy nahoře, jak bylo zmíněno v předchozím testování, nejspíše by to způsobovalo chaos. Opět se ale mohou najít uživatelé, kterým by tato funkce mohla vyhovovat.

- **Seznam letadel**

  U seznamu letadel by uvítal odlišení prvků podbarvením stejnou barvou, jakou mají prvky na radaru.

  Seznam opět nevnímá jako důležitou funkcionalitu, jelikož nastane u minimálního počtu případů. Uživatel si vždy může přiblížit radar podle své preference a při maximálním přiblížení je pravděpodobnost, že se některá letadla budou nacházet na stejném místě po dlouhou dobu, minimální.

- **Různé stupně upozornění**

  Podle jeho slov by aplikace by aplikace měla mít více stupňů upozornění. Je ale jasné, že mnoho lidí bude chtít notifikace co nejvíce omezit. Myslí si však, že první notifikace by měla být povinná a až druhý stupeň by bylo možné vypnout. Prvotní informaci by měl uživatel vždy dostat nehledě na nastavení, aby o riziku věděl.

  Další notifikace by už měl mít uživatel možnost vypnout nebo potvrdit, že o nebezpečí ví a notifikace posílat nechce.

Cílovými uživateli této aplikace vidí jak profesionální piloty, tak i piloty amatérské. Jelikož by to mohl být jeden z dalších důvodů, proč si zakoupit zařízení od firmy *Dronetag s.r.o.* a upřednostnit ho před konkurencí. Dále si nemyslí, že by aplikaci mohl někdo zneužívat pro monitorování letů v okolí. Jedním z důvodů je, že uživatelé, kteří jsou proti létání s drony, ve většině případů nevlastní chytré hodinky *Apple Watch*.

Aplikaci vnímá do budoucna velice pozitivně. Samozřejmě v tuto chvíli není vzdušný prostor tolik zaplněný, ale v rámci několika let by aplikace mohla být velkým přínosem.

Shrnutí

Tato část testování byla opět velice přínosná. Objevili jsme části aplikace, které je možné dále vylepšovat. Taktéž jsme získali množství nápadů na budoucí funkcionality a potvrdili jsme si, že výsledná aplikace má reálné použití.

Všichni účastníci si dokázali představit využití aplikace v praxi, a i když v tuto chvíli není vzdušný prostor příliš zaplněný, do budoucna by se mohlo jednat o dobrý doplněk k mobilní aplikaci firmy *Dronetag s.r.o.*, která by mohla nalákat více potencionálních uživatelů.

Na základě těchto zpětných vazeb budeme dále zvažovat možná rozšíření aplikace do budoucna, případně upravovat některá výchozí chování.

Kapitola 5

Vydání aplikace do App Store

Poslední kapitola popisuje samotné vydání aplikace do obchodu *App Store*. Proces vydávání aplikací vyžaduje několik náležitostí, které obsahují tvorbu **podepisovacích certifikátů** a vytvoření potřebných **provisioning profilů**.

5.1 Podepisování kódu

Díky podepisování kódu se mohou uživatelé při používání aplikací cítit bezpečněji, neboť si dokáží ověřit vývojáře, kteří aplikaci vytvořili, a to pomocí podepisování kódu [47].

Prvním krokem při podepisování je vytvoření Code Signing Request (CSR), který si musí každý vývojář vytvořit na svém zařízení. CSR musí být následně posláno certifikační autoritě, kterou je firma *Apple Inc. Apple* po přijetí CSR potvrdí identitu vývojáře a vydá certifikát.

CSR si můžeme vytvořit v aplikace **Keychain Access**, která je nainstalovaná na každém *Apple* zařízení.

5.2 Certifikáty

Certifikáty mohou být několika typů, přičemž hlavními dvěma jsou:

- **Vývojový certifikát**

  Slouží k internímu vývoji aplikací. Tyto aplikace mohou být nasazené pouze na interní zařízení registrovaná pod účtem vývojáře.

- **Distribuční certifikát**

  Slouží k vydávání aplikací. Tyto aplikace mohou být následně vydány v obchodě *App Store* na jakékoliv podporované zařízení.

Oba tyto certifikáty si musí každý vývojář vytvořit v portálu **Apple Developer**.

5.3 Identifikátor aplikace

Každá aplikace také potřebuje svůj vlastní identifikátor – **Bundle ID** [48]. U aplikací pro systém *WatchOS* musíme těchto certifikátů vytvořit dvojnásobek oproti *iOS* aplikacím, jelikož *WatchOS* aplikace obsahuje dva cíle:

- **Apple Watch aplikace**

  Slouží pouze ke spuštění aplikace jako takové a obsahuje dodatečné informace, například verzi, identifikátor přidružené aplikace atd.

- **Apple Watch rozšíření**

  Obsahuje veškerou logiku aplikace. Předchozí cíl obsahuje navíc referenci k tomuto cíli.

Pro každý z cílů nemusí být vytvoření pouze jeden identifikátor, ale může jich být více. Většinou definujeme různé identifikátory pro verze vývojové a verze určené k distribuci, jelikož se některé jejich parametry mohou lišit.

Tyto identifikátory musí být vždy unikátní, jinak nemohou být přijaty. Tvar identifikátoru většinou obsahuje název vývojáře a název aplikace, například `com.apple.calculator`.

5.4 Provisioning profily

Provisioning profil obsahuje podepisovací certifikát, identifikátor zařízení a identifikátor aplikace [49]. Narozdíl od vývoje pro *Android* zařízení nemůžeme instalovat aplikace na jakákoliv zařízení a musíme nejdříve aplikace podepsat. Provisioning profil funguje jako propojení mezi zařízením a účtem vývojáře [50].

Každý profil může být také několika typů:

- **Development**

  Slouží k distribuci na reálná zařízení při vývoji. Jedná se o zařízení registrovaná pod účtem vývojáře. Pro spouštění aplikací v simulátoru není potřeba žádný profil.

- **Ad Hoc**

  Slouží k distribuci mimo naši organizaci. Každé zařízení, na které chceme instalovat aplikaci s tímto profilem, musí být však registrované a je zde maximální limit, který určuje 100 zařízení.

- **App Store**

  Profil, který slouží k nahrání aplikace do *App Store*.

5.5 Vydání aplikace

Jakmile máme vytvořené všechny zmíněné certifikáty a profily, můžeme aplikaci archivovat a distribuovat. Distribuce může být řešena dvěma způsoby:

- **manuálně**
- **automatizovaným procesem**

Manuální distribuce

Způsob manuální distribuce je poměrně jednoduchý. Ve vývojovém prostředí *XCode* zvolíme vytvoření archivu a tento archiv následně nahrajeme do portálu *App Store Connect*.

Portál *App Store Connect* slouží k vydávání aplikací. Jakmile je vytvořený archiv nahrán, je potřeba vyplnit dodatečné údaje a počkat na schválení firmou *Apple Inc.* k tomu, aby mohla být aplikace zveřejněna. U *WatchOS* aplikací archiv obsahuje jednak aplikaci na chytré hodinky a jednak přidruženou mobilní aplikaci. Sice máme možnost vydat aplikaci na *WatchOS* samostatně, ale zatím zde není tato možnost úplně separovaná. V případě vydání samostatné aplikace se pouze liší vytvořený archiv.

Automatizovaná distribuce – CI/CD

Automatizovaný proces distribuce je řešený pomocí Continuous Integration (CI)/Continuous Delivery (CD). Existuje několik služeb, který tento proces pro vývoj aplikací na *Apple* zařízení implementují. Jedná se například o **Fastlane** [51] nebo **Codemagic** [52]. Tyto služby dokáží samy o sobě vyřešit proces podepisování a automatizovaně vytvořit archiv aplikace, který následně nahrají do portálu *App Store Connect*. Součástí procesu může být také spouštění jednotkových testů.

Přidružená mobilní aplikace má v tuto chvíli vydávání zautomatizované právě pomocí **Codemagic**. Vzhledem k tomu, že je řešení již funkční, nebylo potřeba většího zásahu. Stačilo pouze správně nakonfigurovat soubory spojené s nastavením *WatchOS* aplikace a vydání se podařilo tímto způsobem vyřešit.

Závěr

Cílem práce bylo zanalyzovat dostupné aplikace pro chytré hodinky a provést rešerši existujících aplikací pro *Apple Watch*. Následně bylo potřeba navrhnout uživatelské rozhraní aplikace, provést analýzu a aplikaci implementovat.

Nejprve byla provedena rešerše, při které bylo objeveno množství zajímavých aplikací pro chytré hodinky *Apple Watch*. Rešerše posloužila jako základ k pochopení aplikací na tento systém a díky tomu mohly být použity zajímavé prvky nebo funkcionality.

Následně jsme přistoupili k analýze aplikace, ve které byly popsány veškeré funkční a nefunkční požadavky a případy užití, které tyto požadavky pokrývají. Díky těmto požadavkům byly vytvořeny dva různé návrhy uživatelského rozhraní, které byly otestovány s několika účastníky. Toto testování posloužilo k potvrzení vlastních nápadů a funkcí aplikace. Po prvotním testování následovala část tvorby finálního návrhu grafického rozhraní, které zahrnovalo zmíněné nápady.

Poté jsme přistoupili k samotné implementaci. Implementace nejprve popisuje podrobný návrh architektury celé aplikace a vysvětlení klíčových pojmů. Při návrhu architektury jsme se inspirovali především architekturami pro systém *iOS*. Architektura byla navržena tak, aby byla pokud možno co nejčistší, a její jednotlivé vrstvy byly co nejlépe oddělené. Část implementace popisuje veškeré problémy, jež bylo v rámci vývoje potřeba řešit. Taktéž je obsažený popis API, které aplikace využívá.

Předposlední částí je testování. Testování bylo provedeno několika způsoby, a to nejdříve s účastníky z různých prostředí a následně také s lidmi, kteří pracují v oboru letectví. Taktéž bylo jednoduše popsáno testování pomocí jednotkových testů.

Poslední kapitola popisuje proces vydání aplikace a jeho nutné kroky. Také jsou stručně popsány typy certifikátů a profilů, které aplikace musí obsahovat, a jejich význam.

Výsledné řešení

Výsledkem této diplomové práce je funkční aplikace pro chytré hodinky *Apple Watch*, která byla velice podrobně otestována. Veškeré odezvy na aplikaci při testování byly pozitivní a můžeme tedy aplikaci považovat za velmi vydařenou, a připravenou k reálnému využití společností *Dronetag s.r.o.*

V aktuální chvíli je poslední verze aplikace nahrána do služby *AppStore Connect* a čeká se na její schválení společností *Apple Inc.* Jakmile bude aplikace schválena, budou si jí moct uživatelé stáhnout zcela zdarma, a začít ji rovnou používat.

Budoucí rozvoj

V části testování jsme získali velké množství informací a nápadů na budoucí rozvoj aplikace. Tyto nápady budou v budoucnu uváženy pro případná rozšíření aplikace, jelikož jsme je sami vyhodnotili jako přínosná. Aplikace by si také do budoucna zasloužila větší pokrytí jednotkových testů a případně i jejich spouštění na CI/CD. V tuto chvíli taktéž chybí lokalizace do různých jazyků, kterou však nemá ani mobilní aplikace, ale byl by to určitě jeden z dobrých přínosů do budoucna.

V blízké době také očekáváme rozšíření mapy ve frameworku *SwiftUI*. V aktuální chvíli nejsou dostupné některé funkce, které naopak framework *UIKit* poskytuje, a není tak možné vykreslit letovou zónu. Jakmile společnost *Apple Inc.* přidá tuto podporu, bude potřeba mapu rozšířit o chybějící funkce, které jsou dostupné u radaru. Jedná se hlavně o zmíněné vykreslení letové zóny. Dále by bylo dobré implementovat otáčení kamery směrem, kterým se dívá uživatel, jelikož se jednalo o nejčastěji zmiňovanou funkci.

Literatura

[1] Complications. *Apple Inc. [online]*, [cit. 2022-02-05]. Dostupné z: https://developer.apple.com/design/human-interface-guidelines/watchos/overview/complications/

[2] Latitude and longitude. *Britannica [online]*, [cit. 2022-04-12]. Dostupné z: https://www.britannica.com/science/latitude

[3] Legislativa a pravidla létání pro dronu. *DronPro*, [cit. 2022-04-22]. Dostupné z: https://dronpro.cz/o-dalkove-identifikaci-dronu-a-nove-legislative-s-lukasem-brchlem-z-dronetag

[4] Jaké jsou požadavky v jednotlivých podkategoriích „otevřené“ kategorie? *Úřad pro civilní letectví*, [cit. 2022-04-22]. Dostupné z: https://www.caa.cz/ufaqs/jake-jsou-pozadavky-v-jednotlivych-podkategoriich-otevrene-kategorie/

[5] Smartwatch popularity continues as we enter the festive season. *Jennifer Chan - elektrotechnický magazín [online]*, 2021, [cit. 2022-02-05]. Dostupné z: https://www.kantar.com/inspiration/technology/smartwatch-popularity-continues-as-we-enter-the-festive-season

[6] Creating Independent watchOS Apps. *Apple Inc. [online]*, [cit. 2022-02-05]. Dostupné z: https://developer.apple.com/documentation/watchkit/creating_independent_watchos_apps

[7] CARROT Weather: Alerts & Radar. *App Store. [software]*, [cit. 2022-04-22]. Dostupné z: https://apps.apple.com/us/app/carrot-weather-alerts-radar/id961390574

[8] Radar Game. *App Store. [software]*, [cit. 2022-04-22]. Dostupné z: https://apps.apple.com/tt/app/radar-game/id1508877005

[9] Flightradar24 — Flight Tracker. *App Store. [software]*, [cit. 2022-04-22]. Dostupné z: https://apps.apple.com/us/app/flightradar24-flight-tracker/id382233851

[10] PlaneWatcher. *App Store. [software]*, [cit. 2022-04-22]. Dostupné z: https://apps.apple.com/us/app/planewatcher/id1535490559

[11] Lo-fi prototypování: kdy se hodí a jak na něj. *DESIGN KISK [online]*, [cit. 2022-02-28]. Dostupné z: https://medium.com/design-kisk/lo-fi-prototypovan%C3%AD-kdy-se-hod%C3%AD-a-jak-na-nej-5a965914ae39

[12] High-fidelity prototyping: What, When, Why and How? *Prototypr [online]*, [cit. 2022-02-28]. Dostupné z: https://blog.prototypr.io/high-fidelity-prototyping-what-when-why-and-how-f5bbde6a7fd4

[13] Návrhové principy: SOLID. *Zdroják.cz [online]*, [cit. 2022-04-02]. Dostupné z: https://zdrojak.cz/clanky/navrhove-principy-solid/

[14] Hassan, S. I.: *Effect of SOLID Design Principles on Quality of Software: An Empirical Assessment*. International Journal of Scientific & Engineering, 6 vydání, ISBN 2229-5517.

[15] Clean Architecture Tutorial for Android: Getting Started. *Raywenderlich.com [online]*, [cit. 2022-04-02]. Dostupné z: https://www.raywenderlich.com/3595916-clean-architecture-tutorial-for-android-getting-started

[16] Clean Architecture and MVVM on iOS. *OLX Group Engineering [online]*, [cit. 2022-02-28]. Dostupné z: https://tech.olx.com/clean-architecture-and-mvvm-on-ios-c9d167d9f5b3

[17] Hürsch, W. L.; Lopes, C. V.: *Separation of Concerns*. 1995.

[18] Swinject. *Swinject [software]*, [cit. 2022-04-05]. Dostupné z: https://github.com/Swinject/Swinject

[19] Stinsen. *Stinsen [software]*, [cit. 2022-04-05]. Dostupné z: https://github.com/rundfunk47/stinsen

[20] Alamofire. *Alamofire [software]*, [cit. 2022-04-05]. Dostupné z: https://github.com/Alamofire/Alamofire

[21] What is Reactive Programming? *Medium [software]*, [cit. 2022-04-09]. Dostupné z: https://medium.com/@kevalpatel2106/what-is-reactive-programming-da37c1611382

[22] Rxswift. *ReactiveX [software]*, [cit. 2022-04-05]. Dostupné z: https://github.com/ReactiveX/RxSwift

[23] Combine. *Apple Inc. [software]*, [cit. 2022-04-05]. Dostupné z: https://developer.apple.com/documentation/combine

[24] Publishers. *Apple Inc. [software]*, [cit. 2022-04-05]. Dostupné z: https://developer.apple.com/documentation/combine/publishers

[25] Martin, R. C.: *Clean Architecture*. Pearson, první vydání, ISBN 978-0134494166.

[26] Model-View-Controller (MVC) Architecture. *John Deacon*, [cit. 2022-04-22]. Dostupné z: http://www.jdl.co.uk/briefings/MVC.pdf

[27] UIKit Documentation. *Apple Inc. [online]*, [cit. 2022-04-03]. Dostupné z: https://developer.apple.com/documentation/uikit

[28] SwiftUI Documentation. *Apple Inc. [online]*, [cit. 2022-04-03]. Dostupné z: https://developer.apple.com/documentation/swiftui/

[29] Android Architecture Patterns Part2: Model-View-Presenter. *Medium [online]*, [cit. 2022-04-03]. Dostupné z: https://medium.com/upday-devs/android-architecture-patterns-part-2-model-view-presenter-8a6faaae14a5

[30] MVVM in iOS swift. *Medium [online]*, [cit. 2022-04-03]. Dostupné z: https://medium.com/@abhilash.mathur1891/mvvm-in-ios-swift-aa1448a66fb4

[31] Clean Architecture for SwiftUI. *Alexey Naumov - Github.com [online]*, [cit. 2022-04-03]. Dostupné z: https://nalexn.github.io/clean-architecture-swiftui/

[32] iOS User interfaces: Storyboards vs. NIBs vs. Custom Code. *Toptal [online]*, [cit. 2022-04-03]. Dostupné z: https://www.toptal.com/ios/ios-user-interfaces-storyboards-vs-nibs-vs-custom-code

[33] Services. *Dronetag s.r.o. [online]*, [cit. 2022-04-10]. Dostupné z: https://help.dronetag.cz/developers/services/

[34] What is Web Socket? *Medium [online]*, [cit. 2022-04-10]. Dostupné z: https://medium.com/geekculture/wht-is-web-socket-c4d8fec64ccd

[35] Lombardi, A.: *WebSocket: Lightweight Client-Server Communications*. O'Reilly Media, Inc., první vydání, ISBN 978-1-449-36927-9.

[36] Firebase. *Firebase [software]*, [cit. 2022-04-13]. Dostupné z: https://firebase.google.com

[37] Keychain Services. *Apple Inc. [online]*, [cit. 2022-04-10]. Dostupné z: https://developer.apple.com/documentation/security/keychain_services

[38] UserDefaults. *Apple Inc. [online]*, [cit. 2022-04-10]. Dostupné z: https://developer.apple.com/documentation/foundation/userdefaults

[39] What Is the Difference Between Core Data and SQLite. *Cocoacasts [software]*, [cit. 2022-04-10]. Dostupné z: https://cocoacasts.com/what-is-the-difference-between-core-data-and-sqlite/

[40] Watch Connectivity. *Apple Inc. [online]*, [cit. 2022-04-10]. Dostupné z: https://developer.apple.com/documentation/watchconnectivity

[41] Writing custom platform-specific code. *Flutter [online]*, [cit. 2022-04-10]. Dostupné z: https://docs.flutter.dev/development/platform-integration/platform-channels

[42] Socket.IO Client Swift. *Socket.IO [software]*, [cit. 2022-04-10]. Dostupné z: https://github.com/socketio/socket.io-client-swift

[43] SpriteKit. *Apple Inc. [software]*, [cit. 2022-04-09]. Dostupné z: https://developer.apple.com/spritekit/

[44] Drawing SpriteKit Content in a View. *Apple Inc. [online]*, [cit. 2022-04-09]. Dostupné z: https://developer.apple.com/documentation/spritekit/drawing_spritekit_content_in_a_view

[45] Core Location. *Apple Inc. [online]*, [cit. 2022-04-10]. Dostupné z: https://developer.apple.com/documentation/corelocation

[46] What Is Unit Testing? *SmartBear Software [online]*, [cit. 2022-04-13]. Dostupné z: https://smartbear.com/learn/automated-testing/what-is-unit-testing/

[47] How to code sign & publish iOS apps. *Nevercode Ltd. [online]*, [cit. 2022-04-22]. Dostupné z: https://blog.codemagic.io/how-to-code-sign-publish-ios-apps/

[48] Bundle IDs. *Apple Inc. [online]*, [cit. 2022-04-22]. Dostupné z: https://developer.apple.com/documentation/appstoreconnectapi/bundle_ids

[49] Profiles. *Apple Inc. [online]*, [cit. 2022-04-22]. Dostupné z: https://developer.apple.com/documentation/appstoreconnectapi/profiles

[50] What is a provisioning profile & code signing in iOS? *Medium [online]*, [cit. 2022-04-22]. Dostupné z: https://abhimuralidharan.medium.com/what-is-a-provisioning-profile-in-ios-77987a7c54c2

[51] Fastlane. *Google, Inc. [software]*, [cit. 2022-04-22]. Dostupné z: `https://fastlane.tools`

[52] Codemagic. *Nevercode Ltd. [software]*, [cit. 2022-04-22]. Dostupné z: `https://codemagic.io/start/`

Příloha A

Seznam použitých zkratek

API Application Programming Interface.

CD Continuous Delivery.

CI Continuous Integration.

CSR Code Signing Request.

DI dependency injection.

GNSS Global Navigation Satellite System.

hi-fi Hight Fidelity.

lo-fi Low Fidelity.

MVC Model-View-Controller.

MVP Model-View-Presenter.

MVVM Model-View-ViewModel.

REST Representational State Transfer.

SoC Separation of Concerns.

TCP Transmission Control Protocol.

UI User Interface.

UX User Experience.

Příloha **B**

Obsah přiložené SD karty

- `readme.txt` ..... stručný popis obsahu SD karty
- `src`
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF