FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Využití praktik OOP a FP pro vývoj iOS mobilní aplikace pro Ace Volleyball Academy
Student:	Adam Procházka
Vedoucí:	Ing. Marek Suchánek, Ph.D. et Ph.D.
Studijní program:	Informatika
Obor / specializace:	Teoretická informatika
Katedra:	Katedra teoretické informatiky
Platnost zadání:	do konce letního semestru 2024/2025

Pokyny pro vypracování

Ace Volleyball Academy (ace-vba.com) pořádá volejbalové tréninky, turnaje a soustředění. Jedná se o významnou českou organizaci v oblasti volejbalu se stovkami aktivních členů. V rámci existující webové aplikace mohou členové spravovat svůj profil, hlásit se na akce a sledovat své statistiky (úrovně). Pro snadné použití z mobilních zařízení by bylo vhodné vytvořit mobilní aplikaci po vzoru komerční Volley World App. Cílem této práce je vyvinout mobilní aplikaci pro iOS zařízení v multi-paradigmatickém jazyce Swift s vhodným využitím praktik objektově-orientovaného a funkcionálního programování za účelem dosažení efektivity, snadné rozšiřitelnosti a udržitelnosti.

- Analyzujte doménu organizování volejbalových tréninků a turnajů s využitím konceptuálního modelování. Zaměřte se na fungování Ace Volleyball Academy, ale zohledněte i obecné fungování podobných organizací.
- Proveďte stručnou rešerši existujících řešení, v případě potřeby použijte řešení i pro jiné sporty než je volejbal či obecná řešení pro sportovní organizace a pořádání turnajů či jiných obdobných akcí.
- Sestavte katalog oprioritizovaných požadavků a případy užití pro vlastní mobilní aplikaci.
- Popište možnosti vývoje mobilních aplikací ve Swift, zaměřte se na doporučené vzory, architektury a další postupy jak strukturovat zdrojový kód pro zlepšení rozšiřitelnosti, udržitelnosti a efektivity. Zohledněte možnosti kombinace aplikování funkcionálního a objektově-orientovaného programování díky tomu, že se jedná o multiparadigmatický

Elektronicky schválil/a doc. Ing. Jan Janoušek, Ph.D. dne 16. listopadu 2023 v Praze.

jazyk.
- Navrhněte vlastní mobilní aplikaci a implementujte její prototyp s využitím vhodných návrhových vzorů, praktik FP, a dalších. V rámci textu práce diskutujte odlišné přístupy, jejich výhody a nevýhody v případě této aplikace. Do návrhu a implementace dle potřeby zahrňte také případně nutná rozšíření existujícího backendu webové aplikace.
- Zhodnoťte vývoj ve Swift a výslednou mobilní aplikaci v kontextu využití OOP/FP praktik a vlivu na udržitelnost aplikace.

Elektronicky schválil/a doc. Ing. Jan Janoušek, Ph.D. dne 16. listopadu 2023 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Využití praktik OOP a FP pro vývoj iOS mobilní aplikace pro Ace Volleyball Academy

Adam Procházka

Katedra teoretické informatiky
Vedoucí práce: Ing. Marek Suchánek, Ph.D. et Ph.D.

16. května 2024

Poděkování

Rád bych poděkoval mému vedoucímu práce, panu Ing. Marku Suchánkovi, Ph.D. et Ph.D, za jeho cenné rady, připomínky a ochotu věnovat mi svůj čas. Dále bych rád poděkoval všem členům mé rodiny za jejich trpělivost a podporu, kterou mi po celou dobu studia poskytovali. Rád bych také poděkoval svým kamarádům za to, že mi byli v tento čas nablízku a podporovali mě.

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. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu) licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 16. května 2024

České vysoké učení technické v Praze
Fakulta informačních technologií

Odkaz na tuto práci

Procházka, Adam. *Využití praktik OOP a FP pro vývoj iOS mobilní aplikace pro Ace Volleyball Academy.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2024.

Abstrakt

Tato bakalářská práce se zabývá vývojem mobilní aplikace pro platformu iOS určené pro organizaci volejbalových tréninků a turnajů v multi-paradigmatickém jazyce Swift. Aplikace umožňuje uživatelům sledovat svůj profil, volejbalové dovednosti a nadcházející události. Aplikace trenérům umožňuje měnit hráčské dovednosti a organizátorům spravovat hráče v události a jejich status.

V práci je analyzována doména a provedena rešerše existujících řešení. Na základě analýzy a rešerše byl sestaven katalog požadavků a navržena architektura aplikace. Implementace zahrnuje objektově orientované i funkcionální programovací přístupy a návrhové vzory. Aplikace byla navržena pro volejbalový klub Ace Volleyball Academy, do budoucna se plánuje její rozšíření o další funkce a vydání na AppStore.

Klíčová slova Swift, SwiftUI, iOS, mobilní aplikace, volejbal, objektově orientované programování, funcionální programování, návrhové vzory

Abstract

This bachelor thesis deals with the development of a mobile application for the iOS platform designed to organize volleyball training and tournaments in the multi-paradigm language Swift. The app allows users to track their profile, volleyball skills and upcoming events. For coaches, the app allows players to edit skills and organizers to manage participants in an event and their status.

In this thesis, the domain is analyzed, and research on existing solutions is conducted. Based on the analysis and research, a catalogue of requirements has been compiled, and the application architecture has been designed. The implementation includes object oriented and functional programming approaches and design patterns. The application was designed for the Ace Volleyball Academy, and future plans are to extend it with additional features and release it on the AppStore.

Keywords Swift, SwiftUI, iOS, mobile application, volleyball, object oriented programming, functional programming, design patterns

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Aktivní život a s ním spojený sport je součástí životů mnohých z nás. Není vždy jednoduché sportovat sám či bez speciálního vybavení. Proto jsou zde sportovní kluby, které poskytují možnost trávit čas s lidmi se stejným zájmem a účastnit se pravidelných tréninků, her či turnajů nebo soustředění. Sportovní kluby existují pro všechny různé sporty a sportovní aktivity. Mezi nimi je i Ace Volleyball Academy, volejbalový sportovní klub čítající stovky aktivních členů a zaměřující se na zpřístupnění volejbalu pro hráče všech úrovní.

Mobilní telefony a mobilní aplikace jsou již mnoho let neoddělitelnou součástí každého z nás. Usnadňují a zrychlují běžné denní činnosti, dávají nám tak více času na aktivity, které jsou pro nás důležité. Před mobilními aplikacemi byly webové aplikace a ty mobilní z nich často vycházejí.

Programování se jako mnoho ostatních věcí řídí styly a pravidly. Těm v programování říkámé programovací paradigmata a jsou spjaté se samotným programovaním již od jeho počátku. Paradigmata nám dávají pravidla, podle kterých se musíme řídit a naoplátku za to dostáváme lepší kód a informace o něm. Tato práce se zabývá využitím praktik dvou z těchto paradigmat: objektově-orientovaného a funkcionálního programování. To první vychází z chápání světa okolo nás v objektech. Dává nám výhody jako objekty, zapouzdření, polymorfismus či dědičnost. To funkcionální naopak vychází z výpočetního modelu lambda kalkulus, který se používaná při zkoumání funkcí. To nám říká, že program může být proveden pouze aplikováním a skládáním funkcemi. Dává nám funkce vyššího řádu či neměnnost dat, která se hojně využívá při souběžných výpočtech na současných procesorech.

V současné době existuje webová aplikace (ace-vba.com), která uživatelům umožňuje spravovat svůj profil, hlásit se na akce a sledovat své statistiky (úrovně). Cílem této práce je navrhnout mobilní aplikaci pro snadnou interakci se sportovním klubem. Tato aplikace má být navržena v multi-paradigmatickém programovacím jazyce Swift za vhodného využití praktik OOP a FP za účelem dosažení efektivity, snadné rozšiřitelnosti a udržitelnosti.

Kapitola 1

Cíle práce

Hlavním cílem této práce je vyvinutí mobilní aplikace pro platformu iOS v multiparadigmatickém jazyce Swift za použití praktik objektově-orientovaného a funkcionálního programování.

Dílčími cíly jsou:

1. Analyzovat doménu organizování volejbalových tréninků a turnajů s využitím konceptuálního modelování. Analýza má být zaměřena na fungování Ace Volleyball Academy, ale zároveň má zohledňovat i obecné fungování podobných organizací.

2. Provést rešerši existujících řešení mobilních aplikací. V případě potřeby lze zahrnout i řešení pro jiné sporty než je volejbal či obecná řešení pro sportovní organizace a pořádání turnajů či jiných obdobných akcí.

3. Sestavit katalog prioritních požadavků a případů užití pro vlastní mobilní aplikaci.

4. Popsat možnosti vývoje mobilních aplikací v jazyce Swift. Tento cíl se zaměřuje na doporučené vzory, architektury a další postupy jak strukturovat zdrojový kód pro zlepšení rozšiřitelnosti, udržitelnosti a efektivity. Díky tomu, že Swift je multiparadigmatický jazyk, má tento cíl zohledňovat možnosti kombinace aplikování funkcionálního a objektově-orientovaného programování při vývoji mobilní aplikace.

Kapitola 2

Analýza domény

V této kapitole se zaměříme na doménu organizování sportovních akcí v rámci volejbalového sportovního klubu. Hlavní důřaz je kladen na fungování Ace Volleyball Academy, pro kterou je navrhována výsledná mobilní aplikace.

Autor je několikaletý hráč volejbalu, proto informace v této kapitole pochází jak z jeho hlavy, tak i z odborné literatury [5, 6, 7] a z konzultací s trenéry a hráči volejbalu.

2.1 Volejbal

Tato práce se zabývá návrhem aplikace pro volejbalový klub. Je proto na místě si ještě před samotným doménovým modelem něco říci o volejbale, jelikož se od toho odvíjí návrh aplikace.

Klasický (šestkový) volejbal je míčový sport. Jeho pravidla se pro mezinárodní soutěže odvíjí od pravidel [6] od Mezinárodní volejbalové federace (FIVB[1]), pravidla českého volejbalu [7] se s drobnými odlišnostmi od těch mezinárodních řídí těmi od České volejbalové federace (ČVS[2]).

Volejbal se hraje na obdélníkovém hřišti, které je rozdělelené sítí na dva čtverce. V těchto čtvercích stojí 6 a 6 hráčů tvořící jednotlivé týmy. Obě strany hřiště jsou vzhledem k síti pomyslně rozděleny na 6 částí, každá vymezující jednoho hráče. Hráče můžeme dále rozdělit na 3 vpředu a 3 vzadu. Tím se 6 hráčů rozdělí na 3 přední a 3 zadní. Na hřišti je dále speciální příčná čára ve vzdálenosti tří metrů od sítě na obou stranách hřiště. Pro hráče zadní skupiny platí pravidlo, že nesmějí útočit z výskoku před touto čarou.

Nebudu zacházet příliš hluboko do pravidel volejbalu a zároveň je zjednoduším (pro plné znění se odkazuji se na ty mezinárodní [6], ze kterých také v následujícím textu čerpám), ale řekneme si nějaké základy, aby běžný čtenář porozumněl, o co se jedná. Charakteristická věc pro volejbal je, že dotyk země míčem je chyba. Hráči mají až 3 dotyky míče mezi sebou, poslední z nich musí směřovat míč přes sít na soupeřovo hřiště, ti pak mají také až 3 dotyky, aby vrátili míč zpět atd.

Podobně jako v jiných sportech se volejbalový zápas hraje na sety, skládající se z výměn (nebo také rozeher). Jeden z týmů je vždy podávající a druhý

[1] Celým názvem Fédération Internationale de Volleyball, `https://fivb.com`

[2] Celým názvem Český volejbalový svaz, `https://www.cvf.cz`

přijímající. Podává tým, který právě vyhrál výměnu a tím i získal bod. Podává hráč v postavení vpravo vzadu. Rozehra trvá do chyby jednoho z týmů a bod získává tým, který neudělal chybu. Jak již bylo řečeno, hráči dodržují postavení na hřišti. To musí být dodrženo vždy předtím, než se podávající hráč dotkne balónu a uvede ho do hry. Poté je již pohyb hráčů libovolný. Pro popsání změn postavení u hráčů, tzv. rotace, v průběhu hry si musíme nejdříve popsat dva druhy získaných bodů. První druh, stejnojmenně pojmenovaný jako „bod“, se děje při vlastním podání a následném vyhrání výměny. V tomto případě se rotace neuskutečňuje a postavení hráčů zůstává stejné. Naopak je tomu u „ztráty“, tedy u bodu vyhraném po soupeřově podání. Pokud dojde ke ztrátě, tak dochází k rotaci hráčů. Hráči ve směru hodinových ručiček se o jedno postavení otočí a tím přichází nový hráč na podání.

Seznámíme se s následujícímy pojmy pro typy úderů při volejbalu:

1. příjem a příhra – první úder týmu s cílem následné náhry;
2. náhra – většinou druhý úder týmu s cílem následného útoku;
3. útok — poslední úder týmu s cílem zisku bodu nebo znepříjemnění soupeřovy pozice.

Standardní postup při výměně je hraní na výše zmiňované tři údery. Jak příjem, tak příhra jsou první údery ve výměně. Příjem je speciální forma příhry, kdy přihráváme míč z podání. Další první údery s cílem následné náhry jsou již příhry.

Ve volejbale existuje mnoho systémů hry, strategií a taktik. Nebudeme se zabývat vším (odkazuji na [5]), ale podíváme se na pro nás zajímavé dva systémy hry z hlediska výkonnostní úrovně hráčů. Pro tyto systémy se totiž liší organizace sportovních akcí. Tyto dva systémy jsme pojmenovali následovně:

- nespecializovaný herní systém,
- specializovaný herní systém.

2.1.1 Nespecializovaný herní systém

V tomto herním systému hráči nemají určenou pozici a zjednodušeně „hrají tam, kde stojí“, neboli role hráče se odvíjí od pozice, kde se na hřišti nachází. Systém je zpravidla následující: hráč u sítě uprostřed nahrává (případně u sítě vpravo), hráči u sítě po stranách útočí, zadní hráči přijímají a přihrávají.

Tento systém hry není náročný na sehrannost týmu. Proto je oblíbený zejména u začátečníků a nově zformovaných týmů. Zároveň jak hráči v průběhu hry rotují, tak si každý hráč zahraje na každé pozici a může si tak všechny pozice vyzkoušet.

2.1.2 Specializovaný systém hry

Ve specializovaném systému hry se hraje na tzv. pozice. V této souvislosti rozumíme pojmem pozice roli pro každého z hráčů na hřišti. Pozice pro hráče jednak určuje, co má na hřišti dělat, ale také, kde má stát. Každý hráč má předem určenou svou pozici, na které hraje po celou dobu zápasu. Každá pozice, s výjimkou speciální obranné (libero, viz dále), má dvě fáze: pokud je hráč

v přední části hřiště, a pokud je hráč v zadní části hřiště. Pokud zadní hráč (s výjimkou výše zmíněného libera) vyskočí zpoza tří-metrové čáry, může před dopadem útočit i nad úrovní sítě nad přední částí hřiště, což by jinak z přední řady nemohl. Být blíže síti je při útoku obecně lepší z důvodů např.: kratší doby letu míče, lepšího úhlu pro prudší úder a lepší viditelnosti bránicích hráčů. Toto pravidlo nám dává možnost hrát na více útočníků a hrát tento systém hry.

Hlavní motivací pro zavedení tohoto sytému je větší počet útočících hráčů na hřišti, a více útočníku nám dává větší možnost pro uhrání míče. Specializovaný systém hry můžeme rozdělit podle počtu útočníků na systémy nejčastěji 4-2 (4 útočníci) nebo 5-1 (5 útočníků) [5]. Systém 4-2 je pro nás zajímavý pro smíšená družstva, kde hrají 4 muži a 2 ženy, naopak systém 5-1 je nejčastěji používaný v mužských a ženských kategoriích.

Každý hráč má na hřišti vymezené místo, kde může stát. Toto pravidlo platí před uvedením míče do hry. Od té chvíle je již pohyb hráčů na hřišti libovolný. Hráči se po příjmu vymění a dostanou se na místo své pozice. Tím se docílí toho, že každá výměna je stejná z hlediska postavení pozic na hřišti. To napomáhá k sehrání týmu a zlepšení se hráčů ve hraní za jednu pozici.

Tento systém je náročnější, jelikož vyžaduje, aby hráči znali jednak svou pozici, ale také pozice spoluhráčů, aby se dokázali ve hře měnit. Zastoupenost pozic je v týmu daná, proto sestavení hry a sehnání potřebných hráčů je také naročnější než u předchozího systému, jelikož hráči se zpravidla soustředí na jednu až dvě pozice.

2.1.2.1 Pozice

Jak bylo výše zmíněno, pro tento herní systém je specifické hraní na pozice. Každá pozice má ve hře svou roli a své postavení. V následujícím textu si přiblížíme v současné době (a zároveň i v AVA) používané role:

- smečař (OH),
- nahravač (S),
- blokař (MB),
- univerzál (OP),
- libero (L).

Výše uvedené role se používají v systému 5-1, a to v následujícím složení: nahravač, univerzál, 2 smečaři, 2 blokaři a libero. Rozdíl oproti systému 4-2 je absence univerzála za druhého nahravače (nahravačku).

Na příjmu stojí tři hráči: dva smečaři a libero. Na síti (v přední části hřiště) útočí hráči: smečař (vlevo), blokař (uprostřed) a univerzál (vpravo, pokud je přední). Ze zadní části útočí druhý smečař (uprostřed) a univerzál (vpravo, pokud je zadní). Nahravač je role, která primárně ani neútočí ani nepřijímá. Naopak jak z názvu vyplývá, tak nahrává, tedy má dotek míče mezi příjmem a útokem. Libero je speciální role, která nepodává a jde do hry za blokaře do zadní části hřiště. Když by z důvodu rotace hráčů mělo jít libero dopředu, tak se opět mění za předtím vystřídaného blokaře. Jeho hlavní úkol je obrana. Co se blokování týče, je zde hlavním hráčem blokař, ale ostatní přední hráči se

k němu přidávají (smečař, nahravač či univerzál). Blok je aktivita hráče blízko sítě, kdy se snaží odrazit příchozí míč od soupeře zpět do jeho pole. [6]

Pokud se jedna pozice vyskytuje v týmu dvakrát jako například smečař nebo blokař, stojí na hřišti naproti sobě tedy jeden je vždy přední a druhý zadní. To platí i pro dvojici pozic nahravač–univerzál.

2.2 Doménový model

Doménový model (viz 2.1) se skládá z těchto objektů:

- sportovní akce – událost pořádaná sportovním klubem,
- hráč – běžný člen klubu,
- organizátor – osoba pověřená organizací sportovní akce,
- trenér – osoba vedoucí trénink,
- úroveň – označení výkonnosti hráčů,
- místo konání – označení místa konání sportovní akce.

V následujícím textu se na tyto objekty blíže podíváme.

2.2.1 Sportovní akce

Jedna z náplní sportovních klubů je pořádání sportovních akcí. V klasickém sportovním klubu se jedná hlavně o tréninky, ale také o turnaje a soutěže pro různé skupiny lidí a kategorií.

Na sportovní akce se přihlašují hráči. Obecně jsou sportovní akce určeny konkrétním místem konání (sportovní hala), časem konání a skupinou lidí, pro kterou je akce určena. Skupiny lidí specifikujeme různými kritérii (např.: pohlaví, pozice, úroveň, ...). Tato kritéria se často dávají do názvů akcí jako např.: trénink pro muže výkonnostní kategorie A, turnaj pro rodiče s dětmi nebo kemp pro ženy U20 (ženy ve věku do 20 let). Rozdělení hráčů je v rámci AVA řešeno pomocí vykonnostních úrovní. Každý hráč a každná akce je ohodnocena úrovní: číslem od 1 až do 8. Hráči do 5. úrovně hrají nespecializovaným volejbalovým systémem a jsou hodnoceni pro všechny své dovednosti. Hráči od 5. úrovně hrají systémem na posty a jsou hodnoceni pro své dovednosti na jednotlivých pozicích.

V rámci Ace Volleyball Academy se v současné době vyskytují následující typy sportovních událostí: trénink (training), tréninková hra (game), přípravná akademie (preparation academy) a turnaj (cup).

2.2.1.1 Trénink

Trénink je sportovní akce, na které je přítomný trenér. Ten jednak trénink organizuje, ale také se stará o jeho náplň zadáváním cvičení, která se hráči snaží co nejlépe plnit a zlepšovat tak své dovednosti.

Trénink se zpravidla zaměřuje na nácvik izolovaných herních situací a herních činností.

2.2.1.2 Hra

Tréninková hra je další ze sportovních akcí. Na rozdíl od tréninku nevyžaduje přítomnost trenéra, ale podobně jako trénink slouží k zlepšení herních dovedností.

Hlavní náplň této sportovní akce je nesoutěžní nácvik hry volejbalu, herních situací a připravení tak hráčů na soutěžní turnaje.

2.2.1.3 Akademie

Tréninky se konají zpravidla v týdnu a jsou určeny pro jednu úroveň hráčů. Akademie je rozšířený trénink pro více úrovní o větším počtu hráčů (a trenérů) a kurtů, která se kvůli náročnějším časovým možnostem koná o víkendu. Podobnou akci bychom našli u běžných volejbalových klubu pod názvem jednodenní soustředění, či kemp.

U akademie je časté, že se zaměřuje na jednotlivou herní činnost, situaci nebo na trénink hraní za určitou pozici.

2.2.1.4 Turnaj

Hráči se v rámci týmu mohou přihlásit na turnaj, aby změřili síly s dalšími hráči a týmy podobných dovednostních kategorií. Trvání turnaje je kvůli časové náročnosti v měřítku dnů (jeden či více).

Pro hráče bez týmů se pořádají turnaje, kde se hráči nepřihlašují v rámci týmu, ale přihlásí se jako hráč a na začátku turnaje jsou hráči do týmů rozděleni.

2.2.2 Hráč

Hráčem je běžný člen sportovního klubu, který hraje, tedy účastní se sportovních událostí. Každý hráč je přiřazen do výkonnostní kategorie, podle které se může hlásit na sportovní akce, pokud splňuje všechna její kritéria.

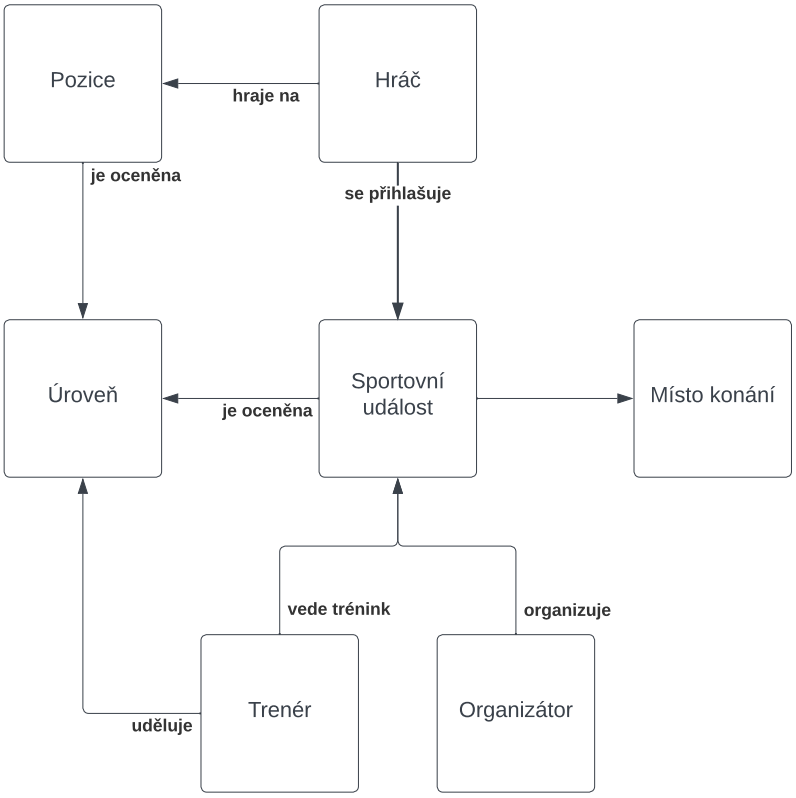
2.2.3 Trenér

Trenér je osoba oprávněná pro vedení tréninku. Trenér by měl vědět, jak se daný sport hraje a měl by umět tyto dovednosti skrze trénink předat dále.

Podle předvedených dovedností na trénincích trenér uděluje hráčům úrovně a sleduje jejich vývoj.

2.2.4 Organizátor

Organizátor je osoba oprávněna pro organizování sportovních akcí. Organizace v tomto smyslu znamená otevření sportovního objektu a šaten; připravení hřiště na hru; zapsání prezence hráčů a vybrání peněz za danou sportovní akci. Pokud se jedná o malou sportovní akci jako je trénink či hra, je přítomen pouze jeden organizátor. V případě tréninku může trenér zastávat i funkci organizátora nebo jím může být jeden z hráčů. Naopak rozsáhlejší akce jako turnaj vyžadují přítomnost více organizátorů, kteří spravují mimo jiné i rozpis zápasů.



Obrázek 2.1: Doménový model

Kapitola 3

Rešerše existujících řešení

V této kapitole se podíváme na existující řešení mobilních aplikací pro správu uživatelů a organizování akcí. Webových a mobilních aplikací zabývajících se touto problematikou je na online trhu nepřeberné množství a v rozsahu této práce není možné alanyzovat všechna.

V následujícím textu se zaměříme na tyto konkrétní řešení: česká mobilní a webová aplikace Týmuj, která byla vybrána z důvodu její popularity a funkčností pro správu sportovního týmu; rezervační systém iSport, který byl vybrán kvůli jeho popularitě mezi českými sportovními areály a celosvětová aplikace Volley World, která byla vybrána kvůli jejímu zaměření na správu volejbalového klubu.

3.1 Týmuj

Platforma Týmuj nabízí webovou a mobilní aplikaci (viz snímek obrazovky 3.1) pro správu sportovního týmu. Není rozdíl v používání mobilní nebo webové aplikace, obě mají stejné funkcionality a data jsou mezi nimi synchronizována. Poté co si uživatel vytvoří svůj účet, je odkázán na připojení se k nějakému týmu nebo jeho založení. Pro vytvoření týmu je potřeba zadat jméno týmu, vybrat sport, město a měnu pro platby. Připojení dalších uživatelů do týmu se řeší zasláním pozvánek formou URL. Používání této aplikace se odvíjí od používání týmu, který představuje základní organizační jednotku. V týmu mají uživatelé 3 role:

- **Hráč** – uživatel se základnímy právy v týmu, který se může účastnit již vytvořených akcí nebo psát do chatu.
- **Správce** – uživatel, který má více práv jak běžný hráč a může přidávat a odebírat hráče, vytvářet a spravovat události a udělovat role správce.
- **Majitel** – uživatel, který tým vytvořil a má všechna práva jako hráč a správce a dále jako jediný možnost smazání týmu.

Aplikace automaticky spravuje členům týmu docházku na události, tato informace je ve výchozím nastavení běžně viditelná, ale toto chování je možné změnit v nastavení soukromí společně s viditelností kontaktních údajů.

Správci mají možnost vytvářet v rámci týmu jednorázové a opakované události. Ty se členům týmu chronologicky zobrazí v jejich aplikaci. V rámci aplikace existují dva typy událostí:

- **Událost** – běžná sportovní událost.
- **Zápas** – událost představující zápas proti jinému týmu.

Pro vytvoření události je nutné zadat název události v případě události či jméno soupeřícího týmu v případě zápas; dále datum a čas; členy, kteří mají být na událost pozváni; a nepovinně vyplnit poznámku, počet míst, místo a zaslání upozornění. Pokud je hráč pozvaný na událost, nachází se ve vztahu k této události v jednom ze 4 stavů:

- **Ve frontě** – hráč je ve frontě na událost, pokud je vyplněna kapacita události a tato kapacita je již obsazená.
- **Jdu** – hráč zareagoval na událost a chce se jí účastnit.
- **Nevím** – hráč buď ještě nezareagoval nebo se nerozhodl, jestli se chce události účastnit.
- **Nejdu** – hráč zareagoval na událost a neúčastní se jí.

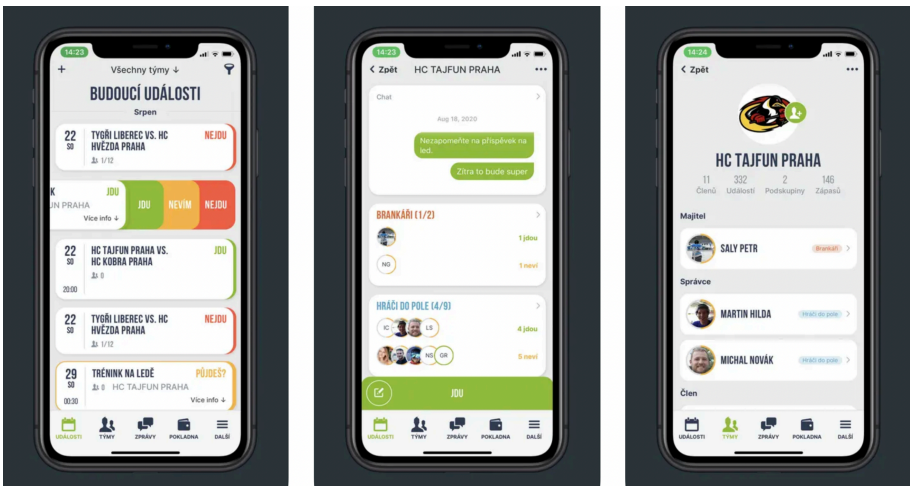
Tyto reakce lze sledovat jako docházku u několika předešlých událostí. Dále má každý možnost psaní zpráv mezi sebou, to se děje na 3 místech: v týmovém chatu – místu, kam může kdokoli psát a zobrazuje se všem členům týmu; v chatu události – místo pro posílání zpráv mezi členy události; na nástěnce – místo, kam může psát pouze správce nebo majitel týmu. Nástěnka se zobrazuje vedle nadcházejících událostí, lze zde posílat dokumenty či vytvářet ankety. V rámci týmu je také možné vytvářet a evidovat platby.

Aplikace Týmuj funguje v bezplatné a zpoplatněné prémiové verzi. Ve verzi zdarma jsou členové týmu omezeni počtem účastníků týmu a počtem událostí, nemohou zasílat notifikace na zprávy, nemají přístupnou galerii a soubory. Prémiová verze tyto funkcionality zpřístupňuje. Její cena se odvíjí od počtu spravovaných týmů v současné době pro maximálně 4 týmy. [8]

3.2 iSport systém

iSport (viz snímek obrazovky3.2) je český rezervační systém s více jak 700 zákazníky. Pro majitele klubu iSport system vytvoří vlastní webovou aplikaci s unikátní URL. Webová aplikace jde po domluvě přizpůsobit potřebám klubu. Je zde možnost i naimplementování specifické funkcionality. iSport systém nabízí svým uživatelům možnost inzerovat události v přehledných rozvrzích v několika záložkách. Události je možné třídit podle lektora, sportu, data, či majitelem vytvořených štítků. Každá rezervace spadá do jedné ze tří následujících kategorií:

- **Individuální rezervace** se používají pro jednorázové zarezervování sportoviště (jako například volejbalové haly). Pro sportoviště lze vypsat volné hodiny v daném časovém pásmu i je cenově odlišit.



Obrázek 3.1: Snímky obrazovky mobilní aplikace Týmuj [1]

- **Skupinová lekce** je jednorázová událost, která navíc obsahuje i kapacitu a případně i lektora. Na jednu skupinovou lekci se tedy může přihlásit více uživatelů. Každá lekce má libovolnou kapacitu a může obsahovat i místa pro náhradníky nebo minimální kapacitu pro konání.
- **Uzavřený kurz** se používá pro inzerci opakující se lekce. Pokud se uživatel přihlásí na kurz, automaticky se přihlašuje na několik souvisejících lekcí na které se nejde jednotlivě přihlašovat.

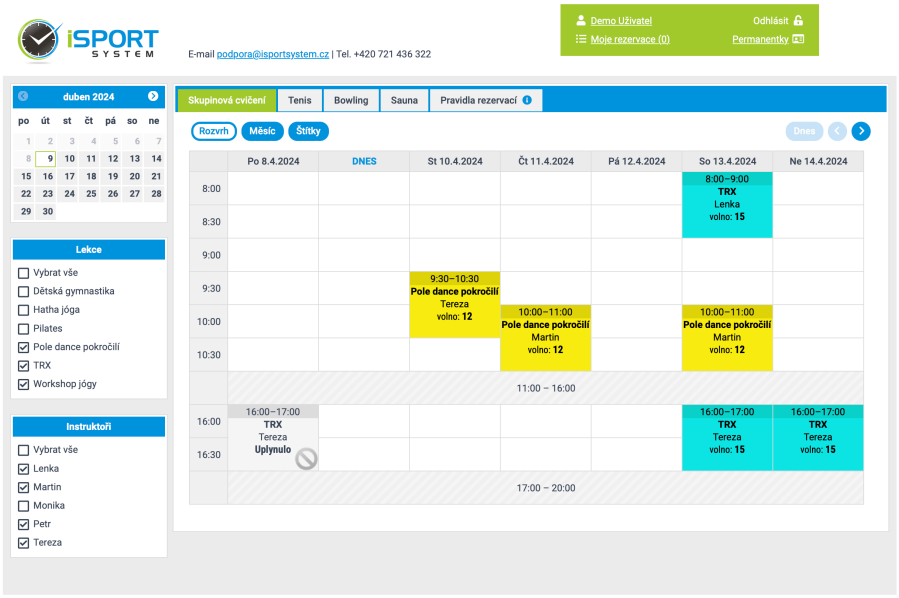
. V rámci rezervací lze zakoupit i doplňující služby (půjčení balónu nebo sítě).

Pro přidání běžných uživatelů do systému stačí, aby se na vytvořeném webu registrovali. Po registraci je jim zpřístupněn kalendář s událostmi, na které se mohou přihlašovat. V rámci svého profilu mohou uživatelé spravovat své rezervace a své konto. Registrované uživatele lze přiřadit do skupin a pro tyto skupiny lze definovat podmínky rezervací včetně jejich ceny. Pro uživatele lze nabízet i předplatné na určitou dobu platnosti a spárovat toto předpolatné se skupinami uživatelů.

Individuální rezervace lze v systému speciálně označit a rozlišit je na tři typy:

- **událost s možností rezervace** – představuje běžnou událost, na kterou se dá přihlásit;
- **událost bez rezervace** – představuje událost na kterou se nedá přihlásit a blokuje tak sportoviště pro mimořádný účel;
- **událost s výhradním právem** – na tuto událost se mohou přihlašovat pouze uživatelé patřící do skupiny, pro kterou se tato událost koná.

iSport systém nenabízí bezplatnou verzi. Pro vytvoření vlastního rezervačního systému je potřeba si vybrat z jedné z čtyř plánů, které se liší cenou, počtem nabízených sportovišť, dostupnými funkcionality a nabízenou podporou. Zpoplatněné je dále grafické přizpůsobení či implementace vlastní funkcionality. [9]



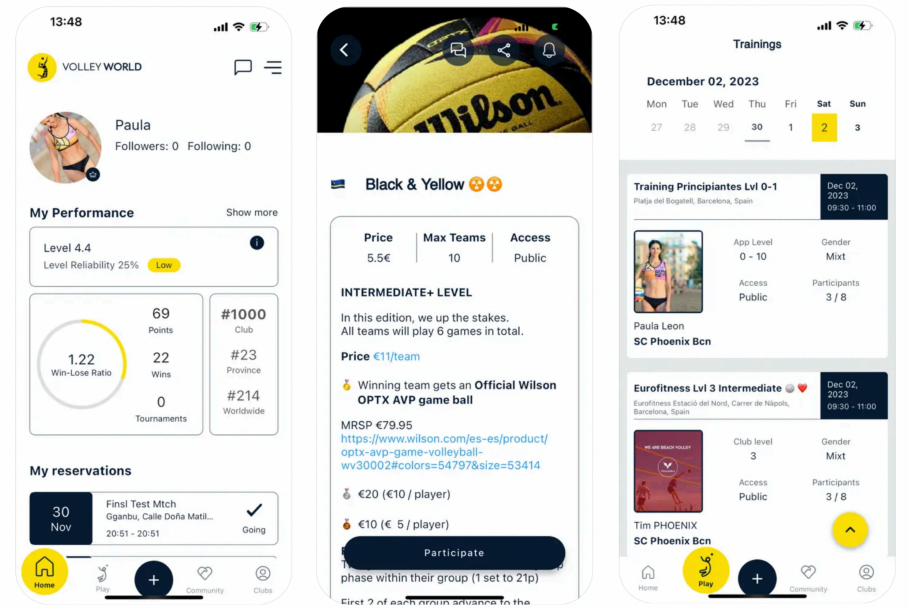
Obrázek 3.2: Snímek obrazovky demo rezervačního systému iSport [2]

3.3 Volley World

Zatímco předchozí dvě řešení jsou univerzálně použitelné pro libovolné sporty, aplikace Volley World (viz snímky obrazovky3.3) se inzeruje jako specificky vytvořená pro volejbal a volejbalové kluby. Mobilní aplikace nabízí pokročilou správu uživatelů, systém pro placení přímo integrovaný v aplikaci, správu rezervací, organizaci turnajů, nástroje pro komunikaci a propojení s uživateli.

Aplikace nabízí vytvoření vlastního klubu po vyplnění údajů jako název, typ (beach volejbalový, šestkový či oba), lokace a případně i odkaz na sociální sítě. V rámci klubu může majitel přidávat a odebírat členy nebo vytvářet události typu: hra, přátelský zápas, trénink, turnaj, volejbalový kemp a sociální událost. Každá událost obsahuje následující informace:

- název události;
- typ zápasů – zápas typu 2 na 2, 3 na 3 nebo 6 a 6;
- pohlaví nebo smíšené pohlaví;
- datum a čas události;
- organizátor;
- popis – upřesnění události;
- veřejnost – zda se jedná o veřejnou či soukromou událost;
- cena a zda se startovné vybírá přímo v aplikaci;
- úroveň události podle klubové úrovně hráčů.



Obrázek 3.3: Snímky obrazovky mobilní aplikace Volley World [3]

Při častém vytváření událostí lze využít vzor a nastavit tak údaje automaticky. Každá událost má tři seznamy přihlášených uživatelů: účastnící se, neúčastnící se a na čekací listině. Organizátor má možnost přihlášené hráče z události odstranit, potvrdit jim platbu, či je přeřadit do jiného seznamu účastníků. Aplikace nabízí velice pokročilý systém organizace turnajů. Uživatelé se přihlásí na turnaj po registraci týmu, splnění vstupních podmínek jako úrovně hráčů a zaplacení vstupních registračních poplatků. Organizátoři si mohou vybrat zda se bude jednat o jednorázovou událost či se skóre bude počítat do sezónní ligy. Aplikace může automaticky vytvořit kvalifikační turnaj. Podle bodů lze hráče nasadit do vyrovnaných skupin a automaticky i vytvořit eliminační fázi. Aplikace nabízí různé vyřazovací systémy jako: na jednu nebo dvě prohry, hra každý s každým či nasazení všech hráčů do jedné skupiny. Skóre zápasu lze zadávat přímo do aplikace, kde se dá i sledovat vývoj turnaje. Uživatelé mohou po vytvoření účtu jednoduše zažádat o přiřazení do klubu výběrem ze seznamu podle názvu případně dle blízkosti podle polohy. Majitel klubu poté může tento požadavek potvrdit či odstranit. Na domovké obrazovce uživatelé vidí klubem nabízené události na které se mohou registrovat. Na svém profilu uživatel sleduje své body, výhry a prohry v zápasech, popis svých silných a slabých stránek, svoji úroveň a docházku na události.

Co se komunikace s uživateli týče, nabízí aplikace uvítací emaily pro nové členy a notifikace v telefonu při změnách v události nebo zaslání zprávy organizátorem pro všechny členy události.

Aplikace nabízí verzi zdarma pro běžné uživatele a členy klubů nebo několik předplatných pro organizátory a kluby. Předplatné přidávající kapacity pro organizaci událostí a další funkcionality jako např.: online databáze uživatelů, obchod s klubovými předměty nebo možnost exportovat data klubu. Jedno z předplatných je i speciálně pro vytvoření klubu. [10]

3.4 Srovnání

Všechny výše zmíněné řešení nám dávají nástroj pro vytváření událostí a správu uživatelů. Analýzou těchto aplikací dostáváme nadhled do požadavků na navrhovanou aplikaci a jejích případů užití. V následujícím textu si srovnáme správu událostí a uživatelů.

V aplikaci Týmuj (3.1) bychom museli vytvořit tým a v tomto týmu vytvářet události. Pro volejbalový klub by bylo vhodné oddělit jednotlivá soutěžní družstva a jejich události. Pro každou skupinu by tak bylo potřeba vytvořit jeden tým. Počet těchto týmu je však aplikací omezen, tedy pro početný klub je použití této aplikace nevhodné. Zároveň aplikace nepodporuje sledování rozsáhlejších statistik u hráčů a není zaměřena na volejbal. Naopak se jednoduše používá a její využití je vhodné v rámci jedné skupiny.

Aplikace iSport (3.2) nám dává robustní systém pro pronájem sportovišť a správu rezervací. Oproti Týmuj zde nejsme omezeni počtem skupin. Systém se intuitivně ovládá a je přehledný, což je ale ovlivněno i vyšší cenou. Pro oddělení skupin uživatelů lze využít již zabudované funkce pro oddělení skupiny a nabízet tak události omezenému počtu uživatelů. Využití by si našly jak jednorázové rezervace pro přátelské hry nebo jednorázové tréninky, tak dlouhodobé lekce pro pravidelné tréninky.

Největší podobnost s navrhovanou aplikací nacházíme v aplikaci World Volley (3.3), která nabízí hotové řešení jak pro organizování různých typů událostí, tak pro správu uživatelů a monitorování jejich statistik, které se přímo týkají volejbalu. U aplikace naopak chybí reference o klubech, které tyto aplikací používají a není tak zřejmá její spolehlivost.

Kapitola 4

Katalog požadavků a případy užití

V této kapitole budou popsány požadavky na navrhovanou mobilní aplikaci a její případy užití. Požadavky vycházejí z analýzy existujících řešení v předchozí kapitole a také přímo ze zadání práce.

4.1 Požadavky

Požadavky budeme dělit na funkční a nefunkční [11]:

- **Funkční požadavky** – požadavky, které nám určují, jak se má výsledný produkt chovat.
- **Nefunkční požadavky** – požadavky, které nám neurčují chování produktu, ale blíže produkt specifikují.

U požadavků budeme udržovat stejnou strukturu. U každého požadavku budeme zaznamenávat tyto informace:

- **Název** – stručný a výstižný název pro jednoduchou identifikaci požadavku.
- **Popis** – detailnější popis požadavku.
- **Priorita** – priorita požadavku.

U rozsáhlejších požadavků budeme vytvářet požadavky nižší úrovně. Priorita požadavků se bude řídit podle kategorií metody MoSCoW [12]:

- **Must have** – požadavek, který výsledný produkt musí splňovat a bez jehož splnění produkt nedává smysl.
- **Should have** – požadavek, který by měl být splněn ve výsledném produku, pokud je to možné.
- **Could have** – požadavek, který by výsledný produkt mohl splňovat, ale jeho splnění není nutné.
- **Won't have** – požadavek, který výsledný produkt v této verzi nebude splňovat, ale je vhodné jej evidovat.

4.1.1 Funkční požadavky

FP1 – Správa uživatele

FP1.1 – Přihlášení uživatele

Uživateli bude po vyplnění přihlašovacích údajů umožněno se přihlásit do mobilní aplikace.

Priorita: **Must have**

FP1.2 – Odhlášení uživatele

Uživatel se může z aplikace odhlásit.

Priorita: **Must have**

FP1.3 – Registrace uživatele

Uživatel se může po vyplnění osobních údajů registrovat do sportovního klubu Ace Volleyball Academy a vytvořit si tak uživatelský účet. Pokud se uživatel registruje v současné webové aplikaci, mělo by být ve výsledné mobilní aplikaci možné se do tohoto účtu přihlásit.

Priorita: **Could have**

FP2 – Zobrazení událostí

FP2.1 – Výpis událostí

Aplikace bude vypisovat seznam událostí, který bude obsahovat o každé události základní informace.

Priorita: **Must have**

FP2.2 – Detail události

Pro každou událost bude možné si zobrazit její detail obsahující rošířené informace o události, účastnících se hráčích a informace o pořadateli události.

Priorita: **Should have**

FP3 – Přihlášení a odhlášení z události

Uživatelům bude umožněno se přihlašovat na vypsané události. Pokud se hráč přihlásí na událost, bude mu umožněno se z ní i odhlásit.

Priorita: **Must have**

FP4 – Správa uživatele v události

FP4.1 – Čekací fronta

U událostí se bude zaznamenávat jejich kapacita. Pokud počet přihlášených uživatelů naplní kapacitu události, budou další uživatelé přemístěni do čekací fronty. Uživatele bude možné mezi čekací frontou a hlavní listinou přesouvat.

Priorita: **Should have**

FP4.2 – Status uživatele

U každého uživatele v události se bude sledovat jeho status. Tento status bude sloužit organizátorům k tomu, aby věděli, zda uživatel fyzicky přišel na událost a jestli za událost zaplatil. Status uživatele bude mít tři hodnoty:

1. „Going“ – Hráč se chce účastnit události.
2. „Checked-in“ – Hráč přišel na událost.
3. „Checked-out“ – Hráč zaplatil za událost.

Priorita: **Could have**

FP4.3 – Vyhození uživatele

Správcům bude umožněno vyhodit uživatele z události, jestli tam nebude patřit a nebude ani vhodné ho umístit na čekací listinu.

Priorita: **Should have**

FP5 – Uživatelský profil

Pro každého uživatele bude možné si zobrazit jeho uživatelský profil, který bude obsahovat profilovou fotku, veřejné osobní údaje, sekci s událostmi a zobrazit si hráčské úrovně.

Priorita: **Should have**

FP6 – Hráčské úrovně

U uživatelů budeme sledovat jejich hráčské úrovně, které budou reprezentovat jejich dovednosti ve volejbale za určité pozice. Jednotlivé úroveně budou označeny číslem a postupem v úrovni. Tyto úrovně bude možné uživatelům přiřazovat, zvyšovat je či snižovat je a upravovat postup v úrovni.

Priorita: **Should have**

FP7 – Využití existujícího API

Mobilní aplikace bude po vzoru té webové komunikovat s existujícím API pro získávání dat o událostech a uživatelích.

Priorita: **Should have**

FP8 – Správa událostí

Správcům aplikace bude umožněno provádět operace spojené se správou událostí. Po zadání údajů spojených s událostí bude možné vytvořit novou událost. Události bude možné editovat a také mazat.

Priorita: **Could have**

FP9 – Notifikace

Z mobilní aplikace budou uživatelům chodit systémové notifikace ohledně nových událostí, změn v přihlášených událostech a změn spojených s jejich uživatelským profilem.

Priorita: **Won’t have**

4.1.2 Nefunkční požadavky

NP1 – Framework SwiftUI

Pro tvorbu uživatelského rozhraní aplikace se bude využívat framework SwiftUI.

Priorita: **Should have**

NP2 – Vzhled po vzoru webové aplikace

Vzhled mobilní aplikace by měl vycházet z existující webové aplikace. To usnadní uživatelům používat obě aplikace současně.

Priorita: **Should have**

NP3 – Využití OOP, FP a návrhových vzorů

Aplikace bude vhodnou formou kombinovat praktiky OOP, FP a návrhové vzory. Tento požadavek plyne ze zadání práce.

Priorita: **Must have**

NP4 – Indikátor procesu na pozadí

Pokud bude aplikace provádět proces na pozadí, který trvá delší dobu nebo asynchroně s uživatelským rozhraním, bude o tom aplikace informovat zobrazením indikátoru načítání.

Priorita: **Could have**

NP5 – Adaptivní barevné schéma aplikace

Aplikace bude obsahovat více barevných schémat, která definují barevné prvky aplikace. Barevné schéma se bude adaptivně měnit podle módu ve kterém se aktuálně zařízení nachází.

Priorita: **Could have**

4.2 Případy užití

Případy užití nám dávají do kontextu funkční požadavky přímo s používaním samotné aplikace. U případů užití sledujeme jejich hlavní, případně vedlejší scénáře, které nám ukazují, jak bude možné s výslednou aplikací pracovat.

UC1: Příhlášení uživatele

Hlavní scénář: uživatel se pomocí emailu a hesla úspěšně přihlásí do aplikace.

1. Uživatel spustí aplikaci a zobrazí se mu obrazovka pro přihlášení.
2. Uživatel vyplní email a heslo do formuláře.
3. Uživatel klikne na tlačítko „Log in“.
4. Uživatele se podařilo ověřit a úspěšně se přihlásil do aplikace.

Vedlejší scénář: uživatel se pokusí přihlásit, ale zadá neplatné údaje.

Začátek: *3. bod hlavního scénáře*

4. Uživatele se nepodaří ověřit.
5. Zobrazí se chybová hláška o neplatných uživatelských údajích.
6. Uživatel klikne na tlačítko „Try again“.
7. Uživatel má možnost znovu zadat přihlašovací údaje.

UC2: Zobrazení vlastního profilu

Hlavní scénář: uživatel si zobrazí vlastní profil a informace na něm uvedené.

Předpoklad: *úspěšný UC1.*

1. Uživatel zmáčkne tlačítko „My Profile“ v dolní části aplikace.
2. Uživateli se zobrazí jeho stránka s uživatelským profilem.
3. Uživatel se pomocí posunutí obrazovky podívá na sekce svého profilu: profilová fotka, veřejné osobní údaje, hráčské úrovně a události.

UC3: Odhlášení uživatele

Hlavní scénář: Uživatel se odhlásí z aplikace.

Začátek: *konec UC2.*

1. Uživatel klikne na tlačítko „Log out“.
2. Uživatel je odhlášen z aplikace.
3. Uživatel je vyzván k přihlášení se do aplikace.

UC4: Zobrazení nadcházejících událostí

Hlavní scénář: uživatel si zobrazí dostupné nadcházející události.

Předpoklad: *úspěšný UC1.*

1. Uživatel zmáčkne tlačítko „Events“ v dolní části aplikace.
2. Uživatel je přesměrován na novou obrazovku.
3. Uživateli se zobrazí seznam událostí.
4. Uživatel se pomocí posunutí obrazovky podívá na nadcházející události.

UC5: Zobrazení detailu události

Hlavní scénář: uživatel si zobrazí detail události ze seznamu událostí.

Začátek: *konec UC4.*

1. Uživatel si vybere událost, jejíž detail si chce zobrazit.
2. Uživatel klikne na obrázek nebo na jméno vybrané události.
3. Uživatel je přesměrován na obrazovku s detailem události.
4. Uživatel si posunem obrazovky prohlédne údaje o události: místo konání, cena, datum a čas, informace o organizátorovi, počet přihlášených uživatelů a počet uživatelů na čekacích na událost.

Vedlejší scénář: uživatel si zobrazí detail události ze svého profilu.

Začátek: *konec UC2.*

1. Uživatel se posunutím obrazovky dostane do sekce „Events“.
2. Uživatel si vybere událost, jejíž detail si chce zobrazit.
3. Uživatel klikne na vybranou událost.

Pokračování: *3. bod hlavního scénáře.*

UC6: Přihlášení na událost

Hlavní scénář: uživatel se přihlásí na událost s nevyčerpanou kapacitou.

Začátek: *konec UC4 nebo konec UC5.*

1. Uživatel klikne na tlačítko „Join“.
2. Nápis na tlačítku se změní na „Leave“.
3. Uživatel je zapsaný na hlavním seznamu události.

Vedlejší scénář: uživatel se přihlásí na čekací listinu události, která má vyčerpanou kapacitu.

Začátek: *2. bod hlavního scénáře.*

3. Uživatel je zapsaný na seznamu čekacích hráčů.

UC7: Odhlášení z události

Hlavní scénář: uživatel se odhlásí z dříve přihlášené události, kterou najde na svém profilu.

Začátek: *konec UC4 nebo konec UC5.*

1. Uživatel klikne na tlačítko „Leave“.
2. Nápis tlačítka se změní na „Join“.
3. Uživatel je odhlášen z události a smazán se seznamu účastníků.

UC8: Zobrazení hráčů účastnících se události

Hlavní scénář: uživatel si zobrazí seznam hráčů účastnících se události a seznam čekajících hráčů.

Začátek: *konec UC4 nebo konec UC5.*

1. Uživatel se posunutím obrazovky dostane do sekce s hráči.
2. Uživatel klikne na tlačítko „Players“.
3. Uživatel je přesměrován na novou obrazovku.
4. Uživateli se zobrazí seznam hráčů účasnících se události.
5. Uživatel se posunutím obrazovky vlevo dostane do seznamu čekajících hráčů.

UC9: Změna statusu u hráče účastnícího se události

Hlavní scénář: organizátor změní status u hráče účastnícího se události pomocí explicitního zadání statusu.

Začátek: *4. bod UC8.*

1. Uživatel si posunem obrazovky ze seznamu vybere hráče, kterému chce změnit status.
2. Uživatel klikne na tlačítko s třemi tečkami.
3. Uživatel vybere jednu z akcí:

   a) Uživatel klikne na tlačítko s nápisem „Check-in Player“.
   b) Uživatel klikne na tlačítko s nápisem „Check-out Player“.
4. Status u hráče se změní podle zmáčknutého tlačítka na *Checked-in* nebo *Checked-out*. Pokud status uživatele byl stejný, jako zmáčknuté tlačítko, jeho status se změní na *Going*.

Alternativní scénář: organizátor změní status u hráče účastnícího se události pomocí kliknutí na ukazatel statusu.

Začátek: *1. bod hlavního scénáře.*

2. Uživatel klikne na ukazatel statusu hráče.
3. Status hráče se posune podle aktuální hodnoty:
   a) Z „Going“ na „Checked-in“.
   b) Z „Checked-in“ na „Checked-out“.
   c) Z „Checked-out“ na „Going“.

UC10: Přesunutí hráče z nebo na čekací listinu

Hlavní scénář: organizátor přesune hráče mezi seznamy hráčů u události.

Začátek: *4. nebo 5. bod UC8.*

1. Uživatel si posunem obrazovky ze seznamu vybere hráče, kterého chce přesunout na druhý seznam.
2. Uživatel klikne na tlačítko s třemi tečkami.
3. Uživatel vybere možnost „Change List“.
4. Vybranný uživatel je přesunut mezi seznamy hráčů, tedy: jestliže byl hráč na seznamu čekajících hráčů, nyní se bude nacházet na seznamu účastních se hráčů a naopak.

UC11: Vyhození hráče z události

Hlavní scénář: organizátor přesune hráče mezi seznamy hráčů u události.

Začátek: *4. nebo 5. bod UC8.*

1. Uživatel si posunem obrazovky ze seznamu vybere hráče, kterého chce přesunout na druhý seznam.
2. Uživatel klikne na tlačítko s třemi tečkami.
3. Uživatel vybere možnost „Kick Player“.
4. Vybraný hráč je odstraněn z události a vymazán ze seznamu.

UC12: Zobrazení profilu uživatele

Hlavní scénář: uživatel si zobrazí profil hráče z události.

Začátek: *4. nebo 5. bod UC8.*

1. Uživatel si posunem obrazovky ze seznamu vybere hráče, jehož profil si chce zobrazit.
2. Uživatel klikne na profilový obrázek nebo na jméno uživatele.

3. Uživatel je přesměrován na novou obrazovku s profilem vybraného uživatele.

Vedlejší scénář: uživatel si zobrazí profil organizátora události.

Začátek: *konec UC5*.

1. Uživatel se posunem obrazovky dostane do sekce o organizátorovi.
2. Uživatel klikne na tlačítko s profilovým obrázkem a jménem organizátora.
3. Uživatel je přesměrován na novou obrazovku s profilem organizátora.

UC13: Změna úrovně u hráče

Hlavní scénář: Trenér hráči upraví úroveň nebo postup v úrovni a potvrdí změny.

Začátek: *konec UC12*.

1. Uživatel se posunem obrazovky dostane do sekce s údaji o uživateli.
2. Uživatel klikne na tlačítko „Actions“.
3. Zobrazí se nabídka s akcemi, které může uživatel provést.
4. Uživatel klikne na možnost „Edit Levels“.
5. Uživatel je přesměrován na novou obrazovku s úrovněmi.
6. Uživatel změní úroveň pomocí výběru jedné z hodnot ve vyskakovacím okně.
7. Uživatel změní hodnotu postupu v úrovni pomocí posunutí posuvníku.
8. Uživatel změny potvrdí zmáčknutím tlačítka „Update Levels“.
9. Změny v úrovních jsou uloženy.

Vedlejší scénář: Trenér hráči upraví úroveň nebo postup v úrovni, ale změny nepotvrdí.

Začátek: *7. bod hlavního scénáře*.

8. Uživatel změny nepotvrdí zmáčknutím tlačítka „Revert All“.
9. Aktuálně provedené změny v úrovních jsou smazány.

Funkčními požadavky jsme si stanovili, jaké funkčnosti chceme, aby aplikace uměla. Případy užití nám dávájí reálné scénáře, jak se aplikace bude používat krok za krokem. Pokrytí jednotlivých případů užití funkčními požadavky můžeme vidět v tabulce 4.1. V této tabulce si také můžeme zkontrolovat, zda jsme pokryli všchny FP, které jsme plánovali využít. V tabulce můžeme vidět, že FP1.3, FP8 a FP9 nejsou pokryty ani v jenom případu užití. Priorita u *FP1.3: Registrace uživatele* je stanovena na *Could have* a stejně tak tomu je i u *FP8: Správa událostí*. Priorita u *FP9 – Notifikace* je stanovena na *Won't have*. Tyto požadavky jsme neplánovali pokrýt a z jejich priorit vidíme, že jejich pokrytí není nutné, aby výsledný produkt dával smysl. Bylo ale vhodné je pro úplnost uvést. Pokryli jsme tedy všechny požadavky, které plánujeme naplnit.

UC	1	2	3	4	5	6	7	8	9	10	11	12	13
FP1.1	X												
FP1.2			X										
FP1.3													
FP2.1				X									
FP2.2					X			X	X	X	X	X	
FP3						X	X						
FP4.1						X		X		X			
FP4.2									X				
FP4.3											X		
FP5		X			X							X	X
FP6													X
FP7	X	X	X	X		X	X		X	X	X		X
FP8													
FP9													

Tabulka 4.1: Tabulka pokrytí funkčních požadavků v případech užití

Kapitola 5

Vývoj v jazyce Swift s využitím OOP a FP

V této kapitole se zaměříme na multi-paradigmatický jazyk Swift a jeho vlastnosti. Dále se budeme zabývat programovacími paradigmaty a návrhovými vzory.

5.1 Jazyk Swift a jeho vlastnosti

Swift je moderní, multi-paradigmatický programovací jazyk vytvořený společností Apple pro vývoj na platformách jako jsou MacOS (operační systém pro počítače Mac), iOS (operační systém pro telefony iPhone), watchOS (OS pro hodinky Apple Watch) a tvOS (OS pro televize Apple TV). Swift je vyvíjen open-source, je volně dostupný a obdržuje pravidelné aktualizace. [13]

Swift patří mezi moderní programovací jazyky a obsahuje mnoho funcionalit, které pomáhají programátorům psát kód. Například tento jazyk inicializuje proměnné před jejich použitím, dále kontroluje, jestli programátor nepřistupuje mimo meze pole nebo v jazyce existuje datový typ, který slouží k práci s nulovým ukazatelem a při zavedení nových proměnných lze jejich typ odvodit z kontextu. [14]

5.2 Objektově orientované programování ve Swiftu

Objektově orientované programování (OOP) je prvním programovacím paradigmatem, který se budeme v této kapitole zabývat. OOP se zabývá halvně třídou a objektem, ale uvedeme si i rozšiřující pojmy jako struktura nebo protokol (z jazyka Swift). Zmíníme se i o konceptech z OOP jako je zapouzdření, dědičnost nebo polymorfismus.

5.2.1 Objekt

Objekt je základní pojem z OOP. Objektem nazveme entitu, která má datový stav a kolekci operací, které nad ní můžeme provádět. Stav objektu, který je okolí skrytý, definují jeho proměnné, které často nazýváme členské proměnné.

Operace objektu potom nazýváme metody. Objekt nemůžeme vytvořit sám o sobě, ale potřebujeme vzor, kterým je pro něj třída. [15]

5.2.2 Třída

Třída nám slouží jako vzor pro tvorbu objektů. Při výrobě objektů říkáme, že objekt je instancí třídy. Třída má definované stejné operace a proměnné jako od ní vytvořený objekt. Zatímco v objektu můžeme najít proměnné, se kterými můžeme přímo pracovat, u třídy definujeme hypotetické proměnné, které se inicializují až při tvorbě objektu. [15, 16] Ukázku, jak můžeme vytvořit třídu v jazyce Swift najdeme ve výpise kódu 1:

```
1  class MojeTřída {
2      var mojeProměnná: String
3
4      init(mojeProměnná: String) {
5             self.mojeProměnná = mojeProměnná
6      }
7
8      func mojeFunkce() {
9
10     }
11 }
```

Ukázka kódu 1: Definice třídy v jazyce Swift, převzato z [16]

5.2.3 Struktura

Struktura je podobně jako třída vzor pro tvorbu objektů. Stejně jako třída i struktura nám seskupuje nějakou funcionalitu ve formě operací a stavu. Rozdíly mezi těmito dvěma koncepty nacházíme přímo v programovacích jazycích. V jazyce Swift najdeme tyto rozdíly [16, 17]:

- **Konstruktor**: třídy i struktury musí mít konstruktor, u struktury si ale můžeme nechat vygenerovat výchozí výchozí kontruktor, který inicializuje všechny členské proměnné.
- **Destruktor**: pouze u třídy můžeme definovat vlastní destruktor.
- **Typ**: třída je *reference type*, zatímco struktura je *value type*. To znamená, že struktury se kopírují při změně.
- **Dědičnost**: pouze třídy mohou dědit.

V jazyce Swift máme dvě skupiny datových typů podle jejich chování při přiřazování [18]:

- **Typ předávaný referencí** („Reference type“ – při přizování vždy předáme referenci na sdílenou instanci.
- **Typ předávaný hodnotou** („Value type“) – při přiřazování se vždy vytvoří kopie.

Ukázku, jak můžeme vytvořit strukturu v jazyce Swift můžeme vidět ve výpise kódu: 2

```
struct MojeStruktura {
    var mojeProměnná: String

    func mojeFunkce() {

    }
}
```

Ukázka kódu 2: Definice struktury v jazyce Swift, převzato z [16]

5.2.4 Protokol

Protokol je koncept z jazyka Swift, ale jeho obdoby najdeme i v jiných programovacích jazycích (např. abstraktní třída v jazyce C++). Definuje nám vzor s metodami, proměnnými a dalšími požadavky pro určitou funkcionalitu nebo úkol. Protokol může být splňován různými typy včetně třídy a struktury a tím poskytovat implementaci těchto požadavků. Říkáme, že pokud nějaký typ splňuje všechny požadavky udáváné v protokolu, k tomuto protokolu konformuje. Zároveň v protokolech můžeme definovat přístupnost atributů. [16, 19] To můžeme vidět u atributu `mojeProměnná` ve výpise kódu 3, kde zároveň můžeme vidět, jak se protokol vytváří v jazyce Swift.

```
protocol MůjProtokol {
    var mojeProměnná: String { get set }

    func mojeFunkce()
}

struct MojeStruktura: MůjProtokol {
    var mojeProměnná: String = "Ahoj světe!"

    func mojeFunkce() {
        print(mojeProměnná)
    }
}
```

Ukázka kódu 3: Definice protokolu a konformace k protokolu v jazyce Swift

5.2.5 Rozšíření

V jazyce Swift lze přidávat další metody, konstruktory, metody nebo atributy do již definovaných tříd, struktur nebo tyto třídy a struktury konformovat k protokolu bez modifikace jejich zdrojového kódu. Toho docílíme pomocí klíčového slova `extension`. [16] Ve výpise 4 můžeme vidět rozšíření existující

struktury `String` o novou metodo `pozpátku`, která vrací řetezec pozpátku. Díky rozšiřitelnosti můžeme tuto metodu rovnou zavolat na nově vytvořeném řetězci.

```
1  extension String {
2      func pozpátku() -> String {
3          var výsledek = ""
4          for písmeno in self {
5              výsledek = String(písmeno) + výsledek
6          }
7          return výsledek
8      }
9  }
10
11 print("Ahoj světe!".pozpátku())
12 > !etěvs johA
```

Ukázka kódu 4: Rozšíření struktury `String` v jazyce Swift, převzato z [16]

5.2.6 Zapouzdření

Zapouzdření je jedním z konceptů z OOP. Jak již bylo řečeno, třídy v sobě definují metody a proměnné, které definují jejich vlastnosti. Metody a proměnné jsou zapouzdřeny uvnitř těchto tříd. To znamená, že vidíme pouze to, co daná třída umí, aniž bychom se museli zabývat tím, jak to dělá. Na druhou stranu, se k datům třídy ani dostat nemůžeme a nemůžeme tak narušit jejich bezpečnost. [20]

5.2.7 Dědičnost

Dalším z konceptů OOP je dědičnost. Každá třída nám definuje nějakou sadu vlastností. Díky dědičnosti pak můžeme vytvářet třídy, jejichž vlastnosti jsou od této třídy odvozené. Odvozená třída (potomek) tak dědí všechny vlastnosti od této třídy (rodič). Tyto zděděné vlastnosti poté potomek může využívat nebo dále upravovat a také přidávat vlasnosti nové. Dědičnost nám také dává možnost znovu využít již napsaný kód napříč třídami, aniž bychom museli kód psát znovu. [15, 16, 20] Ukázku dědění třídy v jazyce Swift si ukážeme na následujícím výpisu kódu 5.

5.2.8 Polymorfismus

Polymorfismus je další z konceptů z OOP. Dává nám možnost možnost pracovat s objekty s jedním rozhraním nezávisle na jejich opravdovém typu. Na každém z těchto objektů můžeme přistoupit k dané operaci a každý objekt si ji řeší sám. To nám dává možnost se na objekty dívat rovnocenně. [16, 20] Příklad si ukážeme na rozšířeném výpisu kódu 6, který navazuje na předchozí příklad 5.

```
class Zvíře {
    func udělejZvuk() {

    }
}

class Pes: Zvíře {
    func udělejZvuk() {
        print("Haf, haf!")
    }
}
```

Ukázka kódu 5: Ukázka třídní dědičnosti v jazyce Swift

```
class Kočka: Zvíře {
    func udělejZvuk() {
        print("Mňau, mňau!")
    }
}

let zvířata: [Zvíře] = [Pes(), Kočka()]
for zvíře in zvířata {
    zvíře.udělejZvuk()
}
> Haf haf!
> Mňau mňau!
```

Ukázka kódu 6: Ukázka polymorfismu v jazyce Swift

5.3 Funkcionální programování ve Swiftu

Funkcionální programování (FP) je druhé programovací paradigma, kterým se budeme v této kapitole zabývat. Ve FP je všechen kód tvořený funcemi (ukázka syntaxe a definice funkce viz 7 a 8). Tyto funkce dokážeme volat s dalšími funkcemi (funkce vyšších řádů), řetězit je za sebe nebo vracet jako hodnoty (first-class objekty). Toto paradigma je také oblíbené pro absenci vedlejších efektů. Volání funkce nemá jiný význam než výpočet jejího výsledku. Neexistují zde přiřazovací operátory nebo proměnné. Jakmile se v programu vyskytne nějaká hodnota, již se nezmění. Tím se eliminuje velká řada chyb a také dovoluje libovnolné pořadí spuštění programu, jelikož žádná funkce nemá vedlejší efekt. [21]

```
func <jméno funkce>(<parametry>) -> <návratový typ> {
    <příkazy>
}
```

Ukázka kódu 7: Syntax pro definici funkce v jazyce Swift

Ve výpise kódu 8 můžeme vidět návratovou hodnotu `Void`, neboli funkce nemá návratovou hodnotu. Pro úplnost ukázky je tam tato hodnota uvedena, ale v tomto případě ji není potřeba uvádět.

```
1 func ahojSvěte() -> Void {
2     print("Ahoj světe!")
3 }
```

Ukázka kódu 8: Definice funkce v jazyce Swift

5.3.1 Generika

Generika (nebo také parametrický pomymorfismus) je dalším z konceptů využívaných v mnoha PJ. Dává nám možnost u funkcí a datových typů abstrahovat jejich konkrétní typ za generický typ `<T>` (např. u `List<T>`) a tento typ určit až později. Tento generický typ můžeme dále rekurzit (např. `List<List<T>>`). [22] Tento koncept nám dovoluje nepsat zbytečný kód navíc pro mnoho datových typů, když se mění pouze signatura funkce a její tělo zůstává stejné. Toho budeme využívat u funkcí vyšších řádů a budeme si je uvádět v jejich generické podobě. [23]

5.3.2 Lambda funkce a uzávěrky

Jestliže nechceme využívat definice funkcí, které můžeme volat z mnoha míst programu podle jejich jména, můžeme využít něco, čemu říkáme anonymní funkce, případně lambda funkce v populárních PJ nebo uzávěrka (z aj. closure) v jazyce Swift. Lambda funkce nám dovolují zabalit nějaký blok kódu do bezejmenných celků, které často využijeme pouze na jednom místě v programu. Typicky je v programu využíváme při ošetřování chyb nebo jako argument nebo návratovou hodnotu pro funkce vyšších řádů. Syntax pro vytvoření uzávěrky v jazyce Swift můžeme vidět ve výpise kódu 9.

```
1 { (<parametry>) -> <návratový typ> in
2     <příkazy>
3 }
```

Ukázka kódu 9: Syntax uzávěrky v jazyce Swift

V jazyce swift se uzávěrky také chovají jako běžný datový typ (first-class objekt) a můžeme je přiřazovat do proměnných a s těmi dále pracovat. [22] To můžeme vidět na příkladě 10.

```
1 let sčítání = { (Int, Int) -> Int in
2     return x + y
3 }
```

Ukázka kódu 10: Příklad uzávěrky v jazyce Swift

5.3.3 Funkce vyšších řádů

Ve funcionálním programování jsou funkce brány jsou každý jiný datový typ, to znamená, že funkce můžeme předávat dalším funkcím v parametru, vracet je z funkcí jako návratový typ, přiřazovat je proměnným nebo je ukládat do kolekcí. Tyto funkce nazýváme funkce vyšších řádů. [23]

Definici funkce, která očekává předání jiné funkce v parametru můžeme vidět ve výpise 11. Tato funkce `apply` vezme číselný argument a předá ho funkci v parametru, která očekává číselný argument. Hodnotu z této funkce poté vrátí.

```
1 func apply(f: (Int) -> Int, x: Int) -> Int {
2     return f(x)
3 }
```

Ukázka kódu 11: Definice funkce s funkcí jako parametrem v jazyce Swift

Vrácení funkce jako návratový typ můžeme vidět ve výpise kódu 12. V tomto výpise můžeme vidět definici funkce, která bere jeden číselný argument a vrací funkci, která bere další číselný argument a oba argumenty vynásobí.

```
1 func násobení(x: Int) -> (Int) -> Int {
2     return { y in
3         return x * y
4     }
5 }
```

Ukázka kódu 12: Definice funkce, která vrací funkci jako návratovou hodnotu v jazyce Swift

Mezi tři základní funkce vyšších řádů patří map, filter a reduce, které nám poskytují neiterativní zpracování nějaké sekvence dat. Používání těchto funkcí je často spjaté s předáváním lambda funkcí v argumentech. [22]

Používání každé z těchto funcí si ukážeme na následující jednoduché sekvenci čísel 13:

```
1 let values = [1, 2, 3, 4, 5]
```

Ukázka kódu 13: Sekvence čísel v jazyce Swift

5.3.3.1 Map

Mapování je proces, při kterém aplikujeme sekvenci argumentů na jednu funkci, která nám postupně vrátí sekvenci výsledků. [22]

V ukázce 14 můžeme vidět generickou definici této funkce a příklad jejího volání s dříve definovanými hodnotami a uzávěrkou, která pro libovolné číslo vrátí jeho dvojnásobek. Tato funkce bere dva argumenty: pole prvků, které se má mapovat a mapovační funkci, kterou na toto pole aplikuje. Ve výpise

můžeme dále vidět výsledek tohoto volání, čímž je stejně dlouhá sekvence čísel, ale s každým prvkem vynásobeným dvěma.

```
1  func map<T, U>(xs: [T], f: T -> U) -> [U] {
2      var result: [U] = []
3      for x in xs {
4          result.append(f(x))
5      }
6      return result
7  }
8
9  print(map(values, {x in x * 2})
10 > [2, 4, 6, 8, 10]
```

Ukázka kódu 14: Funkce `map` v jazyce Swift, převzato z [23]

5.3.3.2 Filter

Filtrování je proces, při kterém aplikujeme danou binární funkci na sekvenci argumentů. Výsledná sekvence se skládá z argumentů, pro které funkce vrátila pravdivou hodnotu. [22]

V ukázce 15 můžeme vidět generickou definici této funkce a příklad jejího volání pro stejnou kolekci čísel jako v předchozím příkladě. Druhým parametrem volání je uzávěrka, která nám vrací pravdivou hodnotu pro čísla větší jak 2.

```
1  func filter<T>(xs: [T], f: T -> Bool) -> [T] {
2      var result: [T] = []
3      for x in xs {
4          if f(x) { result.append(x) }
5      }
6      return result
7  }
8
9  print(filter(values, {x in x > 2})
10 > [3, 4, 5]
```

Ukázka kódu 15: Funkce `filter` v jazyce Swift, převzato z [23]

5.3.3.3 Reduce

Redukce je proces, při kterém aplikujeme danou funkci na sekvenci argumentů a postupně skádáme hodnoty až dostáváme pouze jednu hodnotu. [22]

V ukázce 16 můžeme vidět generickou definice této funkce s třemi parametry: pole prvků, které se má zredukovat; výchozí hodnota; redukční funkce, která bere dva argumenty typů `R` a `A` a vrací hodnotu typu `R`. V příkladě voláme funkci `reduce` opět se stejnou kolekcí jako v předchozích příkladech, nyní však

tuto funkci využíváme pro sečtení prvků v poli. Výchozí hodnota pro sčítání je tak 0 a vytvořená uzávěrka nám pro dvě hodnoty vrací jejich součet.

```
1  func reduce<A, R>(arr: [A], init: R, f: (R, A) -> R) -> R {
2      var result = init
3      for a in arr {
4          result = f(result, a)
5      }
6      return result
7  }
8
9  print(reduce(values, 0 {x, y in x + y})
10 > 15
```

Ukázka kódu 16: Funkce `reduce` v jazyce Swift, převzato z [23]

5.3.3.4 Skládání funkcí dohromady

Ve jazyce Swift máme předchozí funkce definovány i jako metody u generických typů. Můžeme je proto za sebe řetězit a vytvářet tak relativně krátký kód pro nejednoduché úlohy. V ukázce 17 můžeme vidět další konstrukty z jazyka Swift. Prvním je koncová uzávěrka (trailing closure v aj.) tedy možnost napsat uzávěrku jako blok kódu za volanou funkci místo psaní do argumentu, jestliže se jedná o poslední argument a dolarová notace, která zastupuje argumenty funkce. Číslo za dolarem označuje index argumentu ve volání (např. `$0` zastupuje první argument). [24]

V ukázce 17 můžeme vidět zřetězení funkcí `map`, `filter` a `reduce` na jednoduchém příkladě se stejnými hodnotami, které nejprve namapujeme na jejich dvojnásobnou hodnotu, poté vyfiltrujeme ty, které jsou větší jak 5 a nakonec sekvenci zredukujeme násobením.

```
1  print(values.map { $0 * 2 }
2              .filter { $0 < 5 }
3              .reduce(1) { $0 * $1 })
4  > 8
```

Ukázka kódu 17: Skládání funkcí v jazyce Swift, převzato z [24]

5.3.4 Výčtový typ

Výčtové typy nám dovolují seskupit související hodnoty dohromady a používat je bezpečně co se týče typových chyb. Oproti ostatním jazykům jako C nebo Java, kde je možné pouze pracovat s celočíselnými hodnotami, můžeme v jazyce Swift používat i řetězce nebo desetinná čísla. U výčtových typů můžeme, podobně jako u tříd nebo struktur, také definovat metody. K hodnotám ve výčtu můžeme asociovat jinou hodnotu nebo jí hodnotu přiřadit. [16] Na výpise kódu 18 můžeme možnosti výčtu hodnot.

```
enum Hodnoty {
    case žádná
    case litry(Double)
    case sekundy(Int)
}
```

Ukázka kódu 18: Výčtový typ v jazyce Swift, převzato z [24]

5.3.5 Pattern matching

Pattern matching je jedna z možností, jak pracovat s výčtovými typy. Dovoluje nám testovat proměnnou na základě nějakého vzoru nebo druhu ve výčtu a podle výsledku strukturovat program vyvarováním se chyb s datovým typem. S případě porovnávání na výčtový typ nás kompilátor nutí uvést cestu pro každý typ z výčtu, aby se nevyskytla chyba při porovnávání. Pattern matching není omezený jen na výčtové typy, ale můžeme ho používat i pro `n-tice` nebo typ `Optional`. [24]

V ukázce 19 můžeme vidět rozšíření výčtového typu `Values` o novou metodu `printValue`, která nám na základě druhu z výčtu vypíše text na konzoli. Pattern matching je v jazyce Swift prováděn pomocí klíčových slov `swich` a `case` a můžeme si povšimout, že u případu `.liters` a *.seconds* si namapujeme i přidruženou hodnotu, kterou poté využijeme při vypisování.

```
enum Values {
    // ...

    func vypišHodnotu() {
        switch self {
            case .žádná:
                print("Žádná")
            case .litry(let l):
                print("Litrů:", l)
            case .sekundy(let s):
                print("Sekund:", s)
        }
    }
}

Values.sekundy(60). vypišHodnotu()
> Sekund: 60
```

Ukázka kódu 19: Pattern matching v jazyce Swift, převzato z [24]

5.3.6 Optional

Typ `Optional` nám indikuje, že hodnota může nebo nemusí být přítomná. V jazyce Swift ji definujeme pomocí symbolu `?`. Tento typ se používá tam, kde nevíme, zda bude nějaký výsledek a chceme to bezpečně indikovat. Zjistit, zda

je hodnota přítomna a přistoupit k ní pak můžeme pomocí pattern matchingu; operátoru `!`, který vynutí její rozbalení, ale může vyhodit chybu, pokud hodnota není přítomna; pomocí konstruktu `if let`, který přiřadí hodnotu do nové proměnné a zanoří se do `if` větve, jestliže je hodnota přítomná, nebo spadne do dobrovolně definované `else` větve, pokud je hodnota nepřítomna. Využití tohoto konstruktu můžeme vidět na výpise 20.

```
let možnýŘetězec: String? = "Obsahuje řetězec!"
if let řetězec = možnýŘetězec {
    print(řetězec)
} else {
    print("Neobsahuje řetězec!")
}
> Obsahuje řetězec!
```

Ukázka kódu 20: Typ `Optional` v jazyce Swift

5.4 Multiparadigmatický přístup ve Swiftu

Jelikož je Swift multiparadigmatický programovací jazyk, můžeme tyto přístupy kombinovat a využívat podle potřeby. Nabízí se tedy kód strukturovat a zpouzdřovat objektově do tříd a implementavat prvky z FP a tyto funkce pak řetězit.

Navážeme na ukázku řetězení funkcí 17 a ukážeme si, jak vypadá část `Array<T>` s metodou `map` z dokumentace jazyka [25] na výpise 21. U prvního argumentu funkce `map` se setkáváme s dvojím pojmenováním argumentů, kde první argument (ten pro volání funkce) nemá jméno (`_`), tedy při volání této funkce nemusíme specifikovat jméno argumentu a druhé jméno argumentu slouží pro práci s argumentem uvnitř metody. Tato metoda nám vrací nové pole prvků generického typu `T`, tedy původní pole neměníme a jelikož je návratový typ pole, můžeme dále řetězit další volání funkcí vyššího řádu.

```
struct Array<Element> {
    // ...

    func map<T>(_ transform: (Self.Element) throws -> T)
    ↪  rethrows -> [T] { ... }
}
```

Ukázka kódu 21: Struktura `Array` s metodou `map` v jazyce Swift, převzato z [25]

5.5 Návrhové vzory ve Swiftu

Návrhové vzory jsou součástí návrhu konzistentního a dobře udržitelného softwaru. Cílí na běžné problémy a dávají nám ověřené řešení. Jejich používání může zrychlit proces vývoje a díky jejich popularitě mezi vývojáři vede i k rychlejšímu pochopení kódu.

Návrhové vzory budeme rozdělovat (a běžně se tak také rozdělují) do tří kategorií:

- **„Creational patterns“** *(vytvářející)* – návrhové vzory, které se zabývají vytvářením objektů.
- **„Structural patterns“** *(strukturální)* – návrhové vzory, které se zabývají kompozicí objektů.
- **„Behavioral patterns“** *(behaviorální – týkající se chování)* – návrhové vzory, které se zabývají interakcí mezi objekty.

Kniha [26] celkově definuje 23 návrhových vzorů. Z důvodu rozsahu podkapitoly není možné do podrobna rozebrat každý z nich, proto si z každé z těchto kategorií ukážeme 3 až 5 vzorů, které jsou podle autora nejužitečnější. U těchto vzorů si představíme jeho následující 4 části [26]:

1. **Jméno vzoru**: pro identifikaci a rychlý popis návrhového vzoru budeme používat jméno tvořené jedním až dvěma slovy.
2. **Problém**: problém nám popisuje, kdy tento tento návrhový vzor můžeme použít.
3. **Řešení**: řešení nám popisuje, jak výše zmíněný problém tento návrhový vzor řeší.
4. **Důsledky**: důsledky nám shrnují použití tohoto návrhového vzoru, jeho výhody a nevýhody.

Na závěr kapitoly se podíváme na přímé použití některých z níže jmenovaných návrhových vzorů a podíváme se na jejich implementaci v jazyce Swift. Zde budeme čerpat hlavně z knihy v současné době o nejaktuálnější verzi jazyka Swift [16].

5.5.1 Vytvářející návrhové vzory

Tato kategorie návrhových vzorů se zabývá problémy s vytvářením objektů. Do této kategorie patří tyto návrhové vzory [16, 26, 27]:

Stavitel („Builder“)

Problém: Potřebujeme vytvořit třídu se složitou konfigurací a v aplikaci nechceme mít volně dostupné výchozí inicializační hodnoty.

Řešení: Umístíme logiku vytváření složitějších objektů do speciální třídy. Do této třídy také uložíme výchozí hodnoty pro konstrukci objektu.

Důsledky: Díky tomuto NV umožníme volající komponentě vytvářet objekty bez nutnosti vědět příliš mnoho konfiguračních informací a výchozích hodnotách o daném objektu.

Tovární metoda („Factory method“)

Problém: Máme několik tříd, které implementují stejný protokol nebo dědí od stejné třídy a chceme přenechat logiku, jakou instanci vytvořit na někom jiném.

Řešení: Pro vytváření jednotlivých instancí těchto tříd vytvoříme metodu, která je bude vytvářet. Logiku pro vytváření instancí schováme do této metody před komponentou, která tuto metodu bude volat.

Důsledky: Schováme logiku na vytváření těchto objektů a volající komponenta neví, která z implementací byla vybrána.

Abstraktní továrna („Abstract factory")

Problém: V komponentě potřebujeme používat několik kompatibilních objektů, ale nepotřebujeme, aby komponenta věděla, jaké typy objektů to jsou nebo jaké objekty spolu dokáží spolupracovat.

Řešení: Tento návrhový vzor nám poskytuje rozhraní pro vytváření sady souvisejících objektů bez specifikace jejich konkrétního typu. Dělá to tak skrze abstraktní továrnu, tedy abstraktní třídu s metodami pro vytváření jednotlivých objektů z dané skupiny. Tato třída má několik implementací podle dané sady objektů.

Důsledky: Díky tomuto NV jsme schopni zapouzdřit výtváření objektů do jedné třídy a jednoduše tak vyměnit celou skupinu objektů.

Prototyp („Prototype")

Problém: Potřebujeme vytvořit novou instanci objektu se stejnou konfigurací.

Řešení: Třídě, kterou chceme kopírovat, nově umožníme vytvářet kopie podle existujících instancí. Často se to dělá vytvořením protokolu s metodou `clone`, nám vrací zkopírovaný objekt stejného typu.

Důsledky: Díky tomuto přístupu jsme schopni schovat vytváření instance před volající komponentou do stejné třídy. Také nám umožňuje neopakovat kód pro inicializaci objektů.

Jedináček („Singleton")

Problém: Potřebujeme zajistit, aby se používala pouze jedna instance dané třídy.

Řešení: U této třídy, říkejme jí jedináček, si vytvoříme nový atribut, který bude držet sdílenou instanci. Konkstruktor této třídy bude nově privátní, tedy jedináčka bude možné vytvořit pouze zevnitř této třídy. Ta bude zároveň poskytovat možnost pro získání přístupu k této sdílené instanci. Ukázku můžeme vidět ve výpise 22.

Důsledky: Díky tomuto přístupu si můžeme být jisti, že se v programu nachází pouze jedna instance dané třídy.

5.5.2 Strukturální návrhové vzory

Tato kategorie návrhových vzorů se zabývá tím, jak můžeme třídy kombinovat dohromady, aby se s nimi lépe pracovalo nebo abychom schovali nějakou jejich vnitřní logiku. Z této kategorie jsem vybral následující návrhové vzory [16, 26, 27]:

```swift
class MůjJedináček {
    static let sdílenáInstance = MůjJedináček()
    var data: Int = 0
    private init() {}
}

var jedináček1 = MůjJedináček.sdílenáInstance
var jedináček2 = MůjJedináček.sdílenáInstance
jedináček1.data = 1

print(jedináček1.data == jedináček2.data)
> true
```

Ukázka kódu 22: Adaptace návrhového vzoru *jedináček* v jazyce Swift, převzato z [16]

Adaptér („Adapter“)

Problém: Máme dvě komponenty s nekompatibilním rozhraním (mnohdy jedna z nich je komponenta z externí knihovny) a chceme s těmito komponentami pracovat.

Řešení: Zde vytvoříme novou třídu, která nám bude mapovat funkcionalitu z jedné komponenty na druhou.

Důsledky: Díky tomuto přístupu můžeme zakomponovat kód z externí knihovny do naší aplikace, aniž bychom museli (nebo mohli) měnit externí kód. Na druhou stranu se složitost kódu se zvyšuje. V případě, že můžeme měnit rozhraní komponenty, je její změna jednodušší a doporučenný přístup.

Fasáda („Façade“)

Problém: Máme několik rozhraní nebo složité rozhraní, se kterým se těžko pracuje.

Řešení: Vytvoříme nové, jednodušší rozhraní, které bude tyto rozhraní sjednocovat a bude poskytovat jednoduché metody pro práci s nimi. Ukázku můžeme vidět ve výpise kódu 23

Důsledky: Nově vytvořené rozhraní kombinujeme předchozí složitě se používající rozhraní a přebírá logiku pro jejich operaci. Nově nabízené metody také zapouzdřují a zjednodušují použití těchto rozhraní v komponentách.

Dekorátor („Decorator“)

Problém: Potřebujeme upravit chování objektu, aniž bychom měnili danou třídu nebo komponenty, které jí používají.

Řešení: Původní třídu obalíme do nové třídy, kterou nazveme základní dekorátor, která dědí od původní třídy a deleguje operace na ní. Od tohoto dekorátoru podědíme nové dekorátory podle konkrétních funkcionalit, které potřebujeme implementovat. Instanci původní třídy zabalíme

do dekorátorů, které k jednotlivým voláním metod přidávají nové funkcionality a zároveň volají i ostatní dekorátory a vytvářejí tak zásobník dekorátorů.

Důsledky: Tento přístup můžeme kombinovat a vytvářet tak rozsáhlejší efekty bey nutnosti úpravy zdrojových tříd. Tento přístup nám také zjednodušuje vytváření podskupin funkcionalit, které potřebujeme zahrnout. Naopak v případě, kdy můžeme upravovat zdrojové třídy je tento přístup zbytečný a komplikovaný.

```
1  class MojeFasáda {
2      var komponenta1: Komponenta1
3      var komponenta2: Komponenta2
4
5      func jednodušší0perace() {
6          komponenta1.složitáOperace1()
7          komponenta2.složitáOperace2()
8      }
9
10     init(komponenta1: Komponenta1 = Komponenta1(), komponenta2:
       ↪  Komponenta2 = Komponenta2()) {
11         self.komponenta1 = komponenta1
12         self.komponenta2 = komponenta2
13     }
14 }
15
16 class Komponenta1 {
17     func složitáOperace1() {}
18     init() {}
19 }
20
21 class Komponenta2 {
22     func složitáOperace2() {}
23     init() {}
24 }
```

Ukázka kódu 23: Adaptace návrhového vzoru *fasáda* v jazyce Swift, převzato z [16]

5.5.3 Behaviorální návrhové vzory

Tato kategorie návrhových vzorů se zabývá algoritmy, předáním odpovědnosti mezi objekty a jejich vzájemnou komunikací. Z této kategorie jsem vybral následující návrhové vzory [16, 26, 27]:

Příkaz („Command“)

Problém: Máme několik možných příkazů a typicky až za běhu programu se rozhodujeme, který z nich to má být.

Řešení: Vytvoříme si nový protokol pro všechny příkazy a jednotlivé příkazy zabalíme do nových tříd, které budou z tohoto protokolu dědit a budou schopné provést daný příkaz. Ukázku můžeme vidět na výpise kódu 24.

Důsledky: Můžeme měnit, který příkaz se vykoná za běhu programu a daný příkaz zapouzdříme do nové třídy.

Strategie („Strategy")

Problém: Podobně jako u předchozího návrhového vzoru potřebujeme měnit implementaci, kterou použijeme za běhu programu. V tomto případě měníme algoritmus, který se bude provádět.

Řešení: Vytvoříme si nový protokol, který bude obsluhovat dané algoritmy. V původním místě volání algoritmů pak budeme volat předanou instanci strategie.

Důsledky: Tento přístup nám umožňuje zaměňovat algoritmy za běhu programu. Zároveň nám také umožňuje zapouzdřit detaily volání algoritmu do nové třídy.

Řetěz odpovědnosti („Chain of responsibility")

Problém: Máme několik objektů, které mohou vykonat (vzít odpovědnost za) nějaký úkol, ale chceme zajistit, aby pouze jeden z nich tento úkol vykonal.

Řešení: Vytvoříme řetěz odpovědnosti mezi těmito objekty. Každý objekt zjistí, jestli dokáže vyřešit daný úkol, pokud ne, předá zodpovědnost dalšímu objektu, jinak úkol vykoná.

Důsledky: Tento přístup je vhodný, pokud pouze jeden z objektů má vykonat daný úkol. Objekty za sebe můžeme poskládat v preferenční sekvenci. Jednoduše můžeme řetěz přeskládat, zmenšit nebo zvětšit. Jeho využití se naopak nehodí, pokud chceme zvolit, který z objektů má daný úkol vykonat.

Návštěvník („Visitor")

Problém: Nad kolekcí různých objektů (často se společným předkem) chceme provést nějakou operaci, ale do existujícího kódu těchto objektů nechceme nebo nemůžeme ve velkém zasahovat.

Řešení: Operaci delegujeme na novou třídu, která pro dané typy objektů bude poskytovat metody pro provedení této operace v závislosti na typu objektu. Do původního objektu pouze přidáme metodu pro přijetí návštěvníka, ve které objekt na návštěvníkovi zavolá odpovídající metodu. Kolekci objektů poté projdeme a u každého zavoláme nově vytvořenou metodu pro přijetí návštěvníka s návštěvníkem obsahujícím chtěnou operaci.

Důsledky: Díky tomuto návrhovému vzoru můžeme provádět nové algoritmy nad kolekcemi různých objektů, aniž bychom museli přidávat velké množství nového kódu do stávajících objektů. Tento přístup se naopak nedoporučuje, pokud je kolekce objektů stejného typu nebo pokud můžeme přidávat kód přímo do třídy.

Šablonová metoda („Template method“)

Problém: Máme algoritmus pro různé typy objektů se společnými kroky a kroky, které závisí na konkrétním objektu.

Řešení: Vytvoříme třídu s šablonovou metodou, která bude mít pevné kroky definované a místo závislých kroků bude volat abstraktní metody, které jednotlivé implementace této třídy doimplementují. Případně lze místo metod použít uzávěrky.

Důsledky: Tento přístup nám dovoluje přenechat implementaci určitých kroků algoritmu na jiné třídě nebo kroky specifikovat předáním dané funkce. Tento přístup se naopak správně nepoužívá, pokud přenecháváme celou implementaci na nějaké podtřídě nebo předané funkci.

```swift
protocol Příkaz {
    func spusť()
}

struct PříkazJedna: Příkaz {
    func spusť() {
        print("Spouštím příkaz jedna...")
    }
}

struct PříkazDva: Příkaz {
    func spusť() {
        print("Spouštím příkaz dva...")
    }
}

struct MojeStruktura {
    var příkaz: Příkaz

    func operace() {
        //...
        příkaz.spusť()
    }
}

var mojeStruktura = MojeStruktura(příkaz: PříkazJedna())
mojeStruktura.operace()
print("Výměna příkazu...")
mojeStruktura.příkaz = PříkazDva()
mojeStruktura.operace()
> Spouštím příkaz jedna...
> Výměna příkazu...
> Spouštím příkaz dva...
```

Ukázka kódu 24: Adaptace návrhového vzoru *příkaz* v jazyce Swift, převzato z [16]

Kapitola 6

Návrh mobilní aplikace

V této kapitole se zaměříme na návrh mobilní aplikace. Popíšeme si aktuálně používáné architektury, jejich výhody a nevýhody a vybereme jednu konkrétní z nich. Ve druhé části kapitoly si navrhneme datovou část aplikace. V poslední části si poté rozvrhneme uživatelské rozhraní aplikace do jednotlivých obrazovek a popíšeme si logiku mezi nimi.

6.1 Architektura

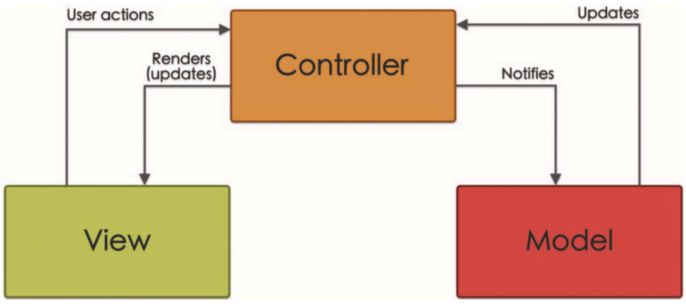
Architektura nám specifikuje strukturu výsledného softwaru, její komponenty a komunikaci mezi nimi. Mezi nejznámější a nejpoužívanější architektury patří Model-View-Controller (MVC), Model-View-ViewModel (MVVM) a Model-View-Presenter (MVP). Tyto architektury si zde blíže přiblížíme. Mezi další možné architektury pužitelné pro vývoj mobilní aplikace patří dále například.: View–Interactor–Presenter–Entity–Router (VIPER), View–Interactor–Presenter (VIP) nebo čistá architektura. [4]

6.1.1 Model-View-Controller

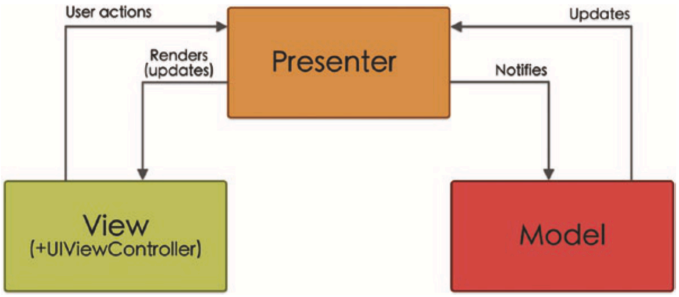
Nejprve se podíváme na architekturu MVC. Její původní verze se datuje k 70. letům minulého století. Tuto verzi si představovat nebudeme, naopak se rovnou podíváme na verzi od společnosti Apple, která odstraňuje komunikaci mezi View a Modelem (schéma nové verze architektury viz 6.1). Tento krok byl udělán hlavně kvůli znovupoužitelnosti jednotlivých komponent ve vývoji.

Tato architektura má tři komponenty jak název napovídá. View se skládá z částí, které uživatel může vidět a s kterými může interagovat. Controller je komponenta mezi View a Modelem, která reaguje na události z View a interpretuje je na Model. Model nám představuje byznysovou logiku aplikace a je zodpovědný za manipulaci s daty. Nazpátek pak Model vrací aktualizace na Controller a ten upozorňuje View.

Výhody této architektury spočívají v její jednoduchosti, což souvisí i s potřebou méně kódu oproti ostatním architekturám pro její implementaci a její komponenty mají jasně definovanou odpovědnost. Mezi nevýhody naopak patří vysoká propojenost mezi View a Controllerem, kdy nám tyto dvě komponenty často splývají do jedné. [4]



Obrázek 6.1: Schéma architektury MVC od společnosti Apple, převzato z [4]



Obrázek 6.2: Schéma architektury MVP, převzato z [4]

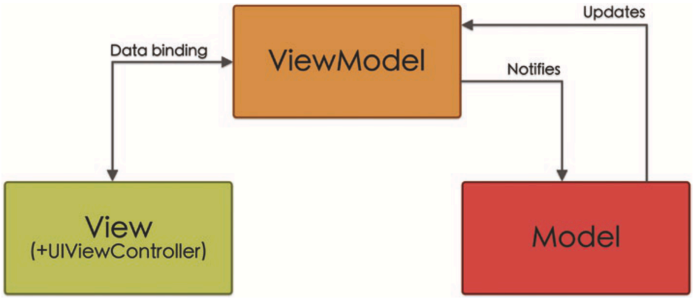
6.1.2 Model-View-Presenter

Druhá architektura, na kterou se zaměříme, je MVP (schéma můžeme vidět na obrázku 6.2. Je to architektura, která se nám odvíjí od MVC někdy v 90. letech. MVP nám představuje novou vrstvu zvanou Presenter. Ten má na starosti zpracovávání žádosti od View, delegovaci žádostí na patřičné operace v Modelu a při vzniklé změně v Modelu o tom upozorňuje View. Model zde chápeme ve stejném smyslu jako u MVC architektury. Naproti tomu View nám obsahuje jak komponenty z předchovího View, tak i komponenty z předchozího Controlleru. Díky nové vrstvě, zde ale View a Controller nepotřebují obsahovat tolik kódu a Controller zde slouží pro přesměrování požadavků, koordinaci mezi vrstvami a spravování navigace v aplikaci.

Architektura MVP je s novou vrstvou o něco komplikovanější než MVC. S tím nám ale umožňuje lépe rozdělit povinnosti jednotlivých vrstev a i je lépe testovat. S tím že se jedná o komplikovanější architekturu se její použití nevyplácí na malých projektech. [4]

6.1.3 Model-View-ViewModel

Architektura MVVM (schéma můžeme vidět na obrázku 6.3) se objevuje okolo roku 2005 jako následník architektury MVP. Úloha vrstvy Model zůstává stejná jako v předchozích dvou představených architekturách. View vrstva obsahuje pouze prvky uživatelského rozhraní a neobsahuje logiku. Tu máme v nové vrstvě ViewModel, která nahrazuje předchozí Presenter vrstvu. ViewModel, podobně



Obrázek 6.3: Schéma architektury MVVM, převzato z [4]

jako Presenter nebo Controller má na starosti logiku aplikace. ViewModel je vlastněn vrstvou View a je s ní spojen skrze tzv. datové propojení *(data binding)*. Tím je možné automaticky aktualizovat View, pokud dojde ke změně v Modelové vrstvě. Zároveň nám tím ale ViewModel vždy představuje to, co vidíme ve View vrstvě a slouží tak k přenosu dat mezi View a Modelem.

Tato architektura nám představuje novou komponentu, která je nově plně zodpovědná za transformaci dat mezi dalšími dvěma vrstvami. To nám usnadňuje udržitelnost a správné rozdělení funkcionalit mezi komponenty. Problém se může nacházet ve správné implementaci datového propojení. Použití externích knihoven nám zvětšuje velikost aplikace. Na druhou stranu ve SwiftUI, frameworku pro tvorbu UI pro jazyk Swift, existuje pozorovací systém, který tento problém řeší. Využití této architektury se tedy hodí společně se SwiftUI. Kód ve ViewModelech bývá často podobný a v rámci projektu i duplicitní.

6.1.4 Výběr architektury

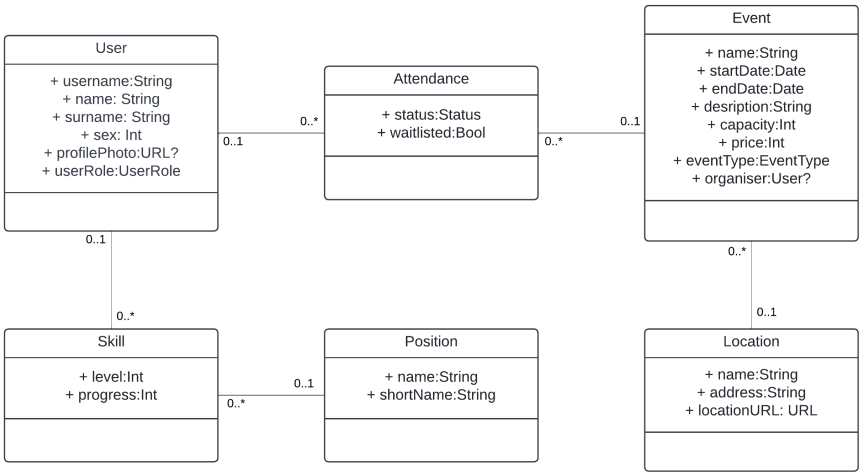
Autor vybral architekturu MVVM, kvůli její dobré udržitelnosti, jednoduchému rozdělení kódu mezi komponenty a zároveň dobrému použití s frameworkem SwiftUI, které plyne z *NP3: Využití OOP, FP a návrhových vzorů* v kalatogu požadavků.

6.1.5 Využití existujícího API

V rámci Modelové vrstvy bude podle funkčního požadavku využíváno existující API, které používá i webová aplikace. Toto API se bude využívat podle funkčních požadavků pro přihlašování, získávání dat o událostech a uživatelích, příhlašování a odhlašování uživatelů na události, upravování statusu u uživatelů účastnících se událostí a spravování hráčských úrovní.

6.2 Návrh datových tříd

Na základě analýzy domény a požadavků na aplikaci si navrhneme třídy reprezentující data v naší aplikaci. Výsledný datový model můžeme vidět na obrázku 6.4.



Obrázek 6.4: Datový model

6.2.1 Uživatel

U uživatele (*User*) si musíme pamatovat jeho jméno a příjmení, abychom je mohli zobrazovat v aplikaci stejně tak jako profilového obrázek, který budeme zobrazovat na profilu a v události. Uživatelů bude více typů podle oprávnění v aplikaci: administrátor, organizátor, trenér a hráč.

6.2.2 Dovednost

Hráč může mít několik dovedností (*Skill*), ale každá dovednost je spjatá vždy s maximálně jedním hráčem. U dovednosti sledujeme její úroveň a postup v úrovni.

6.2.3 Pozice

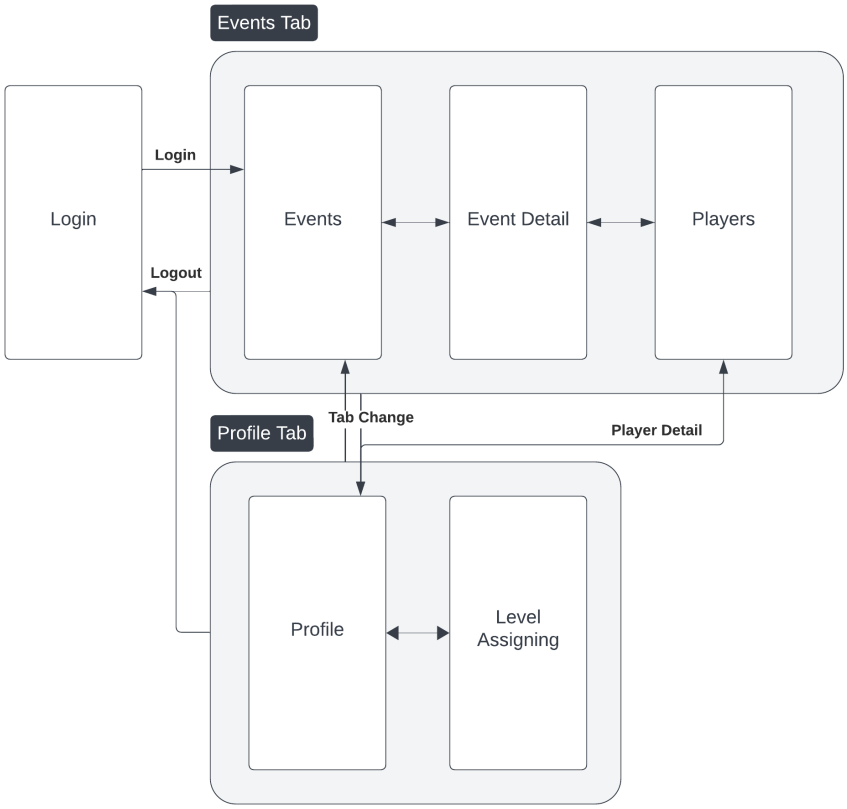
Dovednost je spjatá s pozicí (*Position*), pro kterou je udělována. Pozice nám představuje volejbalovou pozici jak jsou popsány v analýze domény (smečař, blokař, libero, ...). Tyto pozice mají svůj celý název a zkrácený název (ten vychází z anglického názvosloví pro).

6.2.4 Událost

U události (*Event*) si potřebujeme pamatovat její jméno, datum a čas začátku a konce a kapacitu pro krátké zobrazení. Pro detailní zobrazození události dále potřebujeme vědět cenu, podrobnosti a organizátora. Událostí bude více typů: trénink, hra, turnaj a soustředění.

6.2.5 Lokace

Událost může být spjatá s lokací (*Location*). Ta nám představuje halu, kde se daná událost bude konat. U lokaci si musíme pamatovat její jméno, adresu



Obrázek 6.5: Schéma obrazovek aplikace

a pro rychlé určení pozice i její URL pomocí souřadnic.

6.2.6 Účast na události

Vztah mezi uživatelem a událostí nám vyjadřuje třída „účast na události“ (*Attendance*). Z funkčních požadavků o statusu v události nám vyplývá atribut *status* a jeho možné hodnoty: *going*, *checked-in*, *checked-out*. Z *FP4.1: Čekací fronta* nám poté vyplývá atribut *waitlisted*, který nám bude určovat, jestli je hráč na čekací listině.

6.3 Uživatelské rozhraní

V této sekci si navrhneme rozložní aplikace na jednotlivé obrazovky a logiku mezi nimi.

Aplikace bude začínat na obrazovce *Login*, kde bude probíhat přihlášení uživatele. Po úspešném přihlášení se obrazovka změní na *Events*, kde budou vypsané nadcházející události a drobné informace o nich. Po kliknutí na vybranou událost se obrazovka přepne na *Event Detail*. Tato obrazovka bude obsahovat podrobné informace o události jako jsou datum, čas začátku a konce, cena,

popis události, informace o pořadateli a odkaz na přihlášené hráče. Kliknutím na tento odkaz uživatele odkáže na obrazovku *Players*. Na této obrazovce budou dva seznamy hráčů: seznam hráčů účastnících se události a seznam hráčů na čekací listině. Po kliknutí na libovolného hráče se uživatel dostane do obrazovky *Profile*, kde najde detail profilovu daného uživatele. Tato obrazovka bude obsahovat tři sekce: informace o uživateli jako jméno a profilový obrázek; sekci o uživatelských herních dovednostech a sekci o minulých uživatelských událostech. Obrazovka *Level Assigning* bude přístupná uživatelům typu *Trenér* z profilové obrazovky daného hráče. Na této obrazovce budou zobrazeny jednotlivé dovednosti daného hráče a bude možné je upravit.

Uživatelské rozhraní aplikace bude dále obsahovat dvě záložky: *Events Tab* a *Profile Tab*. Mezi těmito záložkami bude možné překlikávat a dostat se tak ze stránky s událostmi do svého profilu a naopak. Výsledné schéma můžeme vidět na obrázku 6.5.

Kapitola 7

Implementace s využitím OOP a FP

V této kapitole se podíváme na proces implementace výsledné mobilní aplikace na základě návrhu v předchozí kapitole. Nejprve se zaměříme na `SwiftUI`, framework pro tvorbu uživatelského rozhraní. Poté si specifikujeme existující API a ukážeme si jeho používání. Dále si implementujeme grafické rozhraní, ukážeme si provázání obrazovek s procesy v aplikaci. Na závěr si ukážeme využití OOP, FP a návrhových vzorů přímo v implementaci aplikace.

7.1 SwiftUI

Z nefunkčního požadavku vychází použití frameworku `SwiftUI` [28] pro tvorbu uživatelského rozhraní. Jedná se o deklarativní framework, tedy píšeme, co cheme, aby jednotlivé obrazovky obsahovaly, ne to jak se to má provést. Použití tohoto frameworku je úzce spojeno s protokolem `View`, který nám představuje grafický prvek, který uživatel uvidí. Konstrukci vlastního `View` můžeme vidět na výpise 25, kde atribut `body` nám představuje vyzobrazený obsah. V tomto případě se jedná o text *„Ahoj světe!“*.

```
1 struct MojeView: View {
2     var body: some View {
3         Text("Ahoj světe!")
4     }
5 }
```

Ukázka kódu 25: Konformace k protokolu `View`, převzato z [29]

Organizaci těchto `View` zařídíme pomocí speciálních struktur jako `HStack` (horizontální rozložení), `VStack` (vertikální rozložení) nebo `ScrollView` (`View`, které mohu posouvat), které také konformují k protokolu `View`, tedy mohu je kombinovat a řetězit. Těmto strukturám předáme do konstrukturu uzávěrku s `View`, které chceme vyzobrazit. Tyto struktury dále v konstruktoru berou další nepovinné parametry pro bližší specifikaci rozložení jako vzdálenost mezi jednotlivými grafickými prvky nebo zarovnání.

Protokol `View` má definované metody pro jeho modifikaci. Tyto metody obalují `View` do struktur, které konformují k protokolu `ViewModifier`. Mezi

nejpoužívanější metody, které modifikují `View` patří například: `padding` (mezera mezi ostatními `View`), `frame` (ohraničení) nebo `background` (pozadí).

Strukturu a výchozí bod aplikace definujeme strukturou, která konformuje k protokolu `App`, do kterého vkládáme jednotlivá naše vytvořená `View`.

7.2 Využití API

Jak vyplývá z funčních požadavků, jako zdroj dat bude využíváno API, které využívá i existující webová aplikace[3]. Jeho přesné použití budeme specifikovat v této podkapitole. Pro práci s API jsme si navrhli protokol `APIServicing` 26.

```
protocol APIServicing {
    func getEvents() async throws -> [Event]
    func getEventDetail(eventID: Event.ID) async throws -> Event
    func joinEvent(eventID: Event.ID, userID: User.ID) async
    ↪ throws -> Event
    func leaveEvent(eventID: Event.ID, userID: User.ID) async
    ↪ throws -> Event
    func getPositions() async throws -> [Position]
    func getLevels() async throws -> [Level]
    func getToken() async throws -> Token
    func postLogin(token: Token, email: String, password:
    ↪ String) async throws -> Void
    func getUserSession() async throws -> UserSession
    func getUser(userID: User.ID) async throws -> User
    func updateEvent(eventID: Event.ID, userID: User.ID, status:
    ↪ Status?, waitListed: Bool?) async throws -> Event
    func updateSkills(userID: User.ID, skills: [Skill]) async
    ↪ throws -> User
}
```

Ukázka kódu 26: Protokol `APIServicing` pro práci s existujícím API

Většina metod vychází z funkčních požadavků. Poněkud složitější je přihlašování a s ním spojené vytvoření relace s API. To probíhá ve více krocích:

1. **Získání tokenu**: prvním krokem je získání tokenu, který bude součástí dalších žádostí. Tento krok představuje metoda `getToken`.
2. **Zaslání přihlašovacích údajů**: druhým krokem je zaslání uživatelského emailu a hesla společně s tokenem z předchozího kroku. Pokud tento krok proběhne úspěšně, získáme zabezpečené *Cookie*, kterým se budeme v nadcházejících žádostech autentizovat. Tento krok představuje metoda `postLogin`.
3. **Získání uživatelské relace**: třetím krokem je získání uživatelské relace. Jedná se o objekt, který obsahuje `userID`, tedy unikátní číslo reprezentující uživatele a `userRole` – role, kterou uživatel reprezentuje. Tento krok představuje metoda `getUserSession`.

[3] Ace Volleyball Academy, `https://ace-vba.com`

(4.) **Získání uživatele**: dalším krokem, který už nepatří přímo k přihlášení, ale navazuje na něj, je získání objektu `User`, který získáme pomocí `userID` získaného v předchozím kroku a zavolání metody `getUser`.

Ukazovat si celou implementaci komunikace s API by bylo zbytečně zdlouhavé. Ukážeme si ale průběh na jednom z použití: získání událostí. Nejprve potřebujeme vědět URL endpointu, který máme zavolat. Pro jednoduché použití v kódu si tyto endpointy staticky uložíme jako rozšíření struktury `URL`, která představuje URL v jazyce Swift 27.

```
1 extension URL {
2     //další API endpointy...
3
4     static let getEvents = URL(string:
      ↪  "<base-API-address>/api/events")!
5 }
```

Ukázka kódu 27: Protokol `APIServicing` pro práci s existujícím API

S využitím tohoto rozšíření si ukážeme implementaci části protokolu `APIServicing`. Ve výpise kódu 28 můžeme vidět třídu `APIService` konformující k protokolu `APIServicing`. V konstruktoru třídá dostává objekt typu `URLSession`, který zprostředkovává metodu `data` pro asynchronní získávání dat. Jako parametr tato metoda bere objekt typu `URLRequest`, který v jazyce Swift představuje dotaz na URL. V metodě `getEvents` pro získání událostí voláme parametrickou privátní metodu `fetchData`, která vrací typ `T`, od kterého potřebujeme, aby konformoval k protokolu `Decodable`, což je protokol umožňující vytvoření objektu z dat, v našem případě ve formátu JSON. Metoda `fetchData` nejprve vytvoří objekt `URLRequest`, kterému předá svůj parametr `url` představující API endpoint. Dále nastaví metodu žádosti na `GET` a zavolá metodu `data` na objektu `session` s vytvořenou žádostí. Tato metoda nám vrátí objekt typu `Data`, který v posledním kroku předáme objektu `JSONDecoder` do metody `decode`, která nám zkonstruuje námi žádaný objekt, v tomto případě pole událostí.

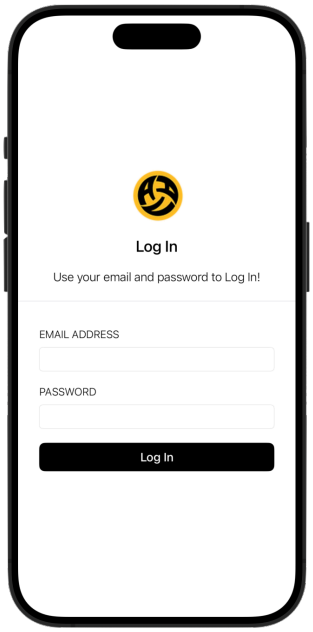
Z důvodu složitosti některých operací si podle návrhového vzoru *fasáda* navrhneme třídu *APIFacade*, která bude poskytovat jednodušší rozhraní pro komunikaci s API. Její implementaci pro metodu `login` spojenou s přihlášením uživatele můžeme vidět ve výpise 29. Použití fasády nám v tomto případě dovoluje schovat logiku komunikace s API pro přihlášení před ostatními komponentami aplikace, které budou potřebovat využívat přihlášení (obrazovka *Login*).

7.3 Grafické rozhraní a procesy

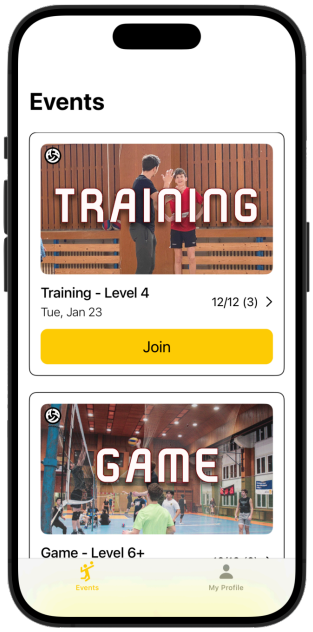
Grafické rozhraní aplikace sestává ze 6 obrazovek. Při spuštění aplikace je uživatel vyzván k přihlášení, poté je mu odemčen samotný obsah aplikace s dvěma záložkami: *události* a *můj profil*. Z obrazovky *události* se kliknutím na událost uživatel přepne na *detail události*. Z *detailu události* se kliknutím tlačítko *hráči* uživateli zobrazí dva seznamy hráčů přihlášených na událost. Přepnutím

```swift
final class APIService: JSONAPIServicing {
    var session: URLSession

    init(session: URLSession) {
        self.session = session
    }

    //implementace ostatních metod...

    private func fetchData<T: Decodable> (url: URL) async throws -> T {
        var request = URLRequest(url: url)
        request.httpMethod = "GET"
        let (data, _) = try await self.session.data(for: request)
        return try JSONDecoder().decode(T.self, from: data)
    }

    func getEvents() async throws -> [Event] {
        try await fetchData(url: .getEvents)
    }
}
```

Ukázka kódu 28: Konformace třídy `APIService` k protokolu `APIServicing` a implementace metody `getEvents`



Obrázek 7.1: Přihlašovací obrazovka



Obrázek 7.2: Obrazovka s událostmi

```
final class APIFacade {
    var api: JSONAPIServicing

    init(api: JSONAPIServicing = JSONAPIService()) {
        self.api = api
    }

    //ostatní metody...

    func login(email: String, password: String) async ->
    ↪ UserSession? {
        do {
            let token = try await api.fetchToken()
            try await api.postLogin(token: token, email: email,
            ↪ password: password)
            return try await api.fetchUserSession()
        } catch {
            print("ERROR:", error)
            return nil
        }
    }
}
```

Ukázka kódu 29: Fasáda `APIFacade` nad `APIService`

záložky nebo kliknutím na hráče v seznamu je zobrazena obrazovka s uživatelským profilem odkud se lze přepnout na obrazovku s úpravou dovedností.

7.3.1 Přihlášení

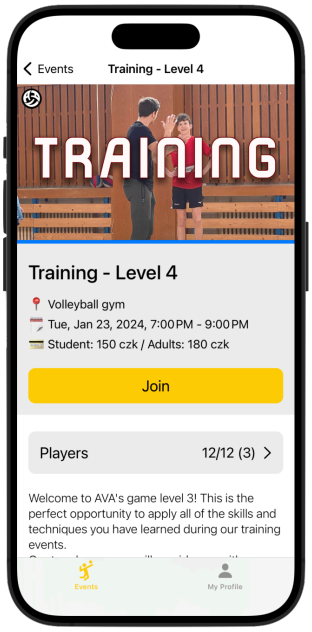
Přihlašovací obrazovka 7.1 obsahuje dvě kolonky: jednu pro zadání emailu a druhou pro zadání hesla. Pro odeslání přihlašovacích údajů je na obrazovce tlačítko „Log In“, které volá metodu `login` na `APIFacade`. Při neúspěšném přihlášení je uživateli ukázáno vyskakovací okno, které uživateli oznámí chybu a vyzve ho na opětovné zadání údajů. V opačném případě se obrazovka s přihlášením zavře a uživateli se ukáže obrazovka s událostmi.

7.3.2 Události

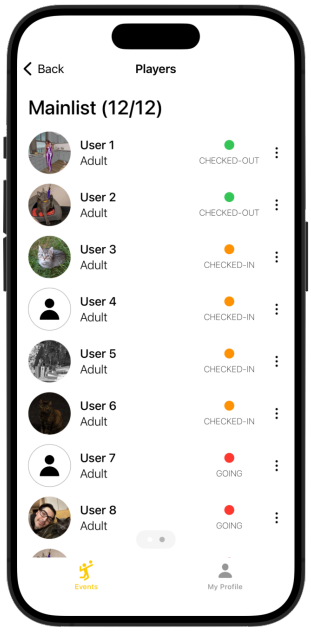
Obrazovka s událostmi 7.2 obsahuje posouvatelný seznam náhledů k událostem, kde každý prvek obsahuje fotku událostí, její jméno, datum, počet hráčů a tlačítko pro přímé přihlášení. Fotka události slouží i jako tlačítko pro zobrazení obrazovky s detailem události.

7.3.3 Detail události

Detail události 7.3 poskytuje podrobnější informace k dané události. Obrazovka obsahuje fotku události, její jméno, místo konání, datum a čas, informace



Obrázek 7.3: Detail události



Obrázek 7.4: Obrazovka s hráči

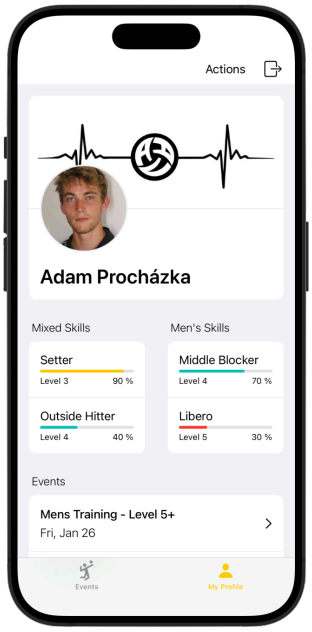
o ceně. Stejně jako v předchozí obrazovce i tato obsahuje tlačítko pro přihlášení na událost. Při posunutí obrazovky se uživatel dostane do podrobnostech o události, za kterou následuje sekce o organizátorovi. Pro zobrazení hráčů, kteří jou přihlášeni na událost, musí uživatel kliknout na tlačítko ve středu obrazovky s nápisem *Players* a počtem hráčů.

7.3.4 Hráči v události

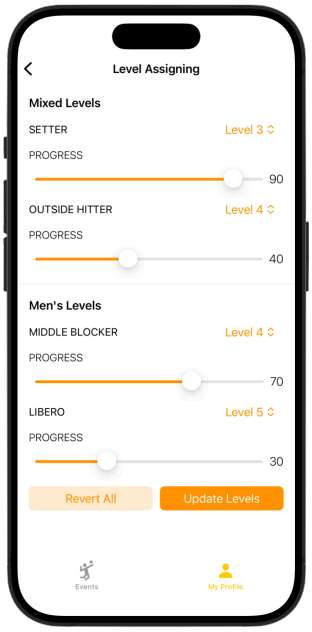
Obrazovka s hráči 7.4, je rozdělena na dvě části mezi kterými se jde přesouvat posunem obrazovky doleva a zpět doprava. Indikátor, na které obrazovce se uživatel nachází, je umístěn v dolní části obrazovky podobně jako v nativních iOS aplikacích. První obrazovka, označená *Mainlist*, obsahuje hráče, kteří se budou účastnit události. U každého z těchto hráčů se v seznamu ukazuje i jejich status, který mohou organizátoři měnit. Po kliknutí na tlačítko s třemi tečkami je organizátorovi zobrazena nabídka s tlačítkami pro manipulaci s hráči v události jako jsou: změna seznamu, změna statusu a vyhození z události. Při kliknutí na profilovou fotku nebo na jméno je uživatel přesměrován na obrazovku s profilem daného hráče.

7.3.5 Uživatelský profil

Uživatelský profil 7.5 obsahuje informace ohledně daného uživatele. Na tuto obrazovku se uživatel může navigovat kliknutím na tlačítko *My Profile* v dolní části aplikace, což mu zobrazí jeho profil nebo z obrazovky, která byla popisována jako předchozí, což uživateli ukáže profil daného hráče.



Obrázek 7.5: Profilová obrazovka



Obrázek 7.6: Přiřazování dovedností

Tato obrazovka je rozdělena na tři sekce: první obsahuje informace o uživateli, tedy profilovou fotku a jméno; druhý obsahuje informace o uživatelských dovednostech, pro každou dovednost je to její úroveň a postup v úrovni; třetí sekce je seznam událostí, na které je uživatel přihlášen.

V horní části obrazovky se nachází dvě tlačítka. První slouží pro odhlášení uživatele, po jeho zmáčknutí jsou smazána uživatelská data a zabezpečené *Cookie* a uživatel je přesměrován na přihlašovací obrazovku. Druhé tlačítko, s nápisem *Actions* je určeno pro provádění akcí nad uživatelem. V této fázi aplikace pro trenéry zobrazí tlačítko *Edit Levels*, které slouží pro úpravu dovedností hráče v nové obrazovce.

7.3.6 Hráčské dovednosti

Obrazovka s hráčskými dovednostmi 7.6 obsahuje všechny dovednosti, které má uživatel přiřazené. Dovednosti jsou rozděleny do dvou kategorií: smíšené a nesmíšené. Pro každou z dovedností je na obrazovce zobrazeno tlačítko s úrovní, které umožňuje vybrat úroveň pro dovednost z dostupných úrovní podle API. Zároveň pro každou dovednost je možné vybrat postup v úrovni pomocí posuvníku, který umožňuje pouze hodnoty od 0 do 100 po 10. Po změně hodnot v dovednostech může trenér změny uložit tlačítkem *Update Levels* nebo změny zahodit zmáčknutím tlačítka *Revert All*.

7.4 Využití OOP, FP a návrhových vzorů

V této podkapitole se podíváme na použití vhodných návrhových vzorů, praktik FP a OOP pro implementaci jednotlivých komponent mobilní aplikace podle provedené analýzy.

7.4.1 Správce souborů Cookies

Jak již bylo řečeno výše, autentizace při komunikaci s API funguje na základě výměny souborů *Cookie*. Při přihlašování se tyto soubory automaticky ukládají do sdílené třídní instance úložiště `HTTPCookieStorage`. Při odhlášení potřebujeme tyto soubory smazat, aby z API nešlo získat uživatelskou relaci. Pro tyto účely jsme navrhli protokol `CookieManaging` a jeho implementaci `CookieManager` 30. Právě v implementaci metody `deleteCookies` můžeme vidět využití funkce vyššího řádu `forEach` nad kolekcí `HTTPCookie`. Této funkci předáváme uzávěrku, která pro dané `Cookie` volá metodu `deleteCookie`, tedy efektivně smažeme všechny prvky v kolekci. Tato kolekce je typu `Optional`, proto si můžeme všimnout symbolu `?`, který nám zajišťuje, že funkce `forEach` se zavolá pouze pokud je přítomná hodnota kolekce.

```
protocol CookieManaging {
    func deleteCookies() -> Void
}

final class CookieManager: CookieManaging {
    private var storage: HTTPCookieStorage

    init(storage: HTTPCookieStorage = .shared) {
        self.storage = storage
    }

    func deleteCookies() {
        self.storage.cookies?.forEach {
            storage.deleteCookie($0)
        }
    }
}
```

Ukázka kódu 30: Správce souborů `Cookie`

7.4.2 API

Výše je již zmíněno využití fasády pro komunikaci s externím API. Tuto fasádu si teď doplníme o výše předvedený `CookieManager` a zároveň si implementujeme návrhový vzor *jedináček*. Díky přidání `CookieManager` do fasády můžeme přidat novou metodu `logout` a mít tak metody spojené s přihlašováním na jednom místě. Tato metoda bude pouze delegovat volání jak můžeme vidět v ukázce 31. Jelikož v aplikaci nepotřebujeme více instancí této třídy, tak si ji

zabalíme do návrhového vzoru *jedináček*. V ukázce můžeme vidět implementaci privátního konstrukturu, který volá pouze sdílená instance.

```
final class APIFacade {
    private var api: APIServicing
    private var cookies: CookieManaging

    static let shared: APIFacade = .init()

    private init(api: APIServicing = APIService(), cookies:
    ↪ CookieManaging = CookieManager()) {
        self.api = api
        self.cookies = cookies
    }

    func logout() {
        self.cookies.deleteCookies()
    }

    //ostatní metody...
}
```

Ukázka kódu 31: Rozšíření `APIFacade` o `CookieManager`

7.4.3 Indikátor načítání

Na indikátoru načítání si ukážeme předávání uzávěrek jako funkcí vyššího řádu do SwiftUI struktur. Ukážeme si to na obrazovce 7.1, kde pokud uživatel klikne na tlačítko *Log In*, částečně se ztmaví obrazovka a zobrazí se indikátor načítání.

Ve výpise 32 můžeme vidět implementaci této obrazovky pomocí struktury `LoginView`, která konformuje k protokolu `View`. Indikátor načítání implementujeme jako `ZStack` (struktura, která předané `Views` skládá na sebe), kterému v uzávěrce předáme `contentView`, `View` které představuje obsah dané obrazovky, a `loadingOverlayView` představující překrytí obrazovky v době načítání. To je podmíněno probíhajícím načítáním dat na pozadí (proměnná `isLoading` na `ViewModelu`).

7.4.4 Rozdělení hráčů na seznamy

Jak víme z obrazovky 7.4, jednotlivé hráče přihlášené na událost musíme rozdělit na hlavní a čekací seznam. Proto ve výpise 33 můžeme vidět rozšíření struktury `Event` o nové dva atributy `mainlist` a `waitlist`. Tyto atributy volají privátní metodu `filteredUsers`, která nám schovává logiku filtrování uživatelů za použití parametru `waitlisted` a do aplikace tak zpřístupňujeme pouze dva výše zmíněné atributy. V metodě `filteredUsers` můžeme vidět použití funkce (v tomto případě metody) vyššího řádu `filter`, definovanou nad strukturou `Array` v jazyce Swift. Této funkci (metodě) v uzávěrce s jedním parametrem (`user`) předáváme logiku filtrování uživatelů, v tomto případě

```
struct LoginView: View {
    @Bindable var viewModel: LoginViewModel

    var body: some View {
        ZStack {
            contentView

            if viewModel.isLoading {
                loadingOverlayView
            }
        }
    }

    //další atributy...
}
```

Ukázka kódu 32: Indikátor načítání v `LoginView`

ponecháváme uživatele, jejichž hodnota atributu `waitlisted` se shoduje s parametrem metody.

```
extension Event {
    private func filteredUsers(waitlisted: Bool) -> [Attendance]
    ↪ {
        self.users.filter { user in
            user.waitlisted == waitlisted
        }
    }

    var mainlist: [Attendance] {
        filteredUsers(waitlisted: false)
    }

    var waitlist: [Attendance] {
        filteredUsers(waitlisted: true)
    }
}
```

Ukázka kódu 33: Rozdělení hráčů na jednotlivé seznamy

7.4.5 Filtrování a řazení událostí

V obrazovce 7.2 chceme zobrazovat nadcházející události a seřazené podle data, aby uživatel nemusel dlouze posouvat seznamem skrze náhodnou permutaci všech událostí v systému. Události získáváme dotazem na API, který nám vrací všechny události v databázi a ještě neseřazené. Pro naše účely jak vidíme v 34 můžeme modifikovat získáné události v metodě `fetchEvents` třídy

`EventsListViewModel`. Nad získanými událostmi můžeme zřetězit funkce vyšších řádů `filter` a `sorted` s patřičnými uzávěrkami jako jejich parametr. Funkci `filter` dáme za argmuent uzávěrku, která nám pro danou událost vrací pravdu, jestliže událost ještě nezačala. Funkci `sorted` předáváme uzávěrku, která bere dva argumenty a vrací pravdu, jestliže čas první události je dříve než čas druhé události. Zřetězení těchto dvou funkcí nám docílí vyfiltrování minulých událostí a jejich seřazení podle času začátku. Můžeme si všimnout i obalovacího protokolu `@Observable`, díky kterému můžeme jednoduše provázat data z `ViewModelu` se spjatým `View`.

```
@Observable
final class EventsListViewModel {
    //metody, konstruktor a atributy...

    func fetchEvents() async {
        defer { isLoading = false }
        isLoading = true

        do {
            events = try await api.fetchEvents()
                .filter { event in
                    event.startDate.timeIntervalSinceNow > 0
                }
                .sorted { e1, e2 in
                   e1.startDate.timeIntervalSinceReferenceDate <
                   ↪  e2.startDate.timeIntervalSinceReferenceDate
                }
        } catch {
            print("[ERROR]:", error)
        }
    }
}
```

Ukázka kódu 34: Filtrování a řazení událostí

7.4.6 Mapování dovedností

Na obrazovce 7.6 můžeme uživateli upravit jeho dovednosti. Pokud uživatel změny potvrdí, zavolá se metoda na `ViewModelu`, která asynchronně volá metodu `updateSkills` na `APIService` s modifikovanou kolekcí dovedností. Tuto metodu vidíme v ukázce 35. V této metodě nejdříve definujeme struktury potřebné do těla žádosti: `UpdateSkillsBody` obaluje kolekci dovedností `SkillData`. Pro získání návratové hodnoty z žádosti poté definujeme strukturu `UpdateSkillsResponse` s jediným atributem na který se mapuje odpověď z API. Jak víme, tak v parametru metody dostáváme typ `[Skill]` a do těla žádosti potřebujeme typ `[SkillData]`. Proto si v `SkillData` vytvoříme konstruktor, který bere typ `Skill` a můžeme použít funkci vyššího řádu `map`, která nám z pole dovedností vytvoří pole pro použití v žádosti pomocí jedné uzávěrky. API očekává, že dostane pole dovedností pod jménem `positionSkills`,

my ho máme pojmenované `skillsData`, proto si musíme dodefinovat speciální výčtový typ `CodingKeys`, který spravuje názvy atributů v zakódovaných datech. Nyní již tuto strukturu můžeme zakódovat, abychom ji mohli použít do těla žádosti. Ve zbytku metody nastavujeme u žádosti její typ, hlavičku a tělo a poté žádost odesíláme a získáváme její odpověď, kterou dekódujeme na výše definovanou strukturu, jejíž status poté vracíme.

7.4.7 Spravování hráčů v události

V této sekci si ukážeme jak v aplikaci předáváme funkce jako funkcionalitu. Obrazovka 7.4 představuje struktura `PlayersListView`, ta obsahuje kolekci uživatelů přihlášených na danou událost, které rozděluje do dvou seznamů podle čekací listiny. Pro zobrazení jednotlivých uživatelů v seznamu slouží struktura `PlayersRowView`, jejíž `ViewModel` vidíme v ukázce 36. Veškerá logika pro spravování hráčů je tedy nechána ve struktuře `PlayersListView` a jednotlivé `PlayersRowView` volají pouze funkce předané jejich `ViewModelu` a díky tomu můžeme odlehčit strukturu `PlayersRowView`.

7.4.8 Autorizace

Na obrazovce s uživatelským profilem 7.5 chceme podle uživatelské role zobrazovat a schovávat tlačítko pro úpravu dovedností. Podobně na obrazovce se seznamy hráčů 7.4 chceme tlačítko pro úpravu statusu hráčů zobrazit pouze pro určité role uživatelů. Jelikož zobrazení těchto částí záleží na uživatelské roli, rozšíříme si strukturu `UserRole` 37 o dvě nové metody `canAdjustPlayerStatus`, řídící zobrazování tlačítka na úpravu statusu, a `canAdjustPlayerSkills`, řídící zobrazování tlačítka na úpravu uživatelských dovedností. Logiku tohoto rozhodování tedy zapouzdříme přímo do struktury, která o tom rozhoduje. Jelikož ve výchozím nastavení chceme zabránit zobrazení ovládacích prvků, tak v těle těchto metod najdeme `switch`, který defaultně zakazuje jejich zobrazení a musíme explicitně určit pro jaké role chceme zobrazování povolit.

```swift
func updateSkills(skills: [Skill]) async throws -> String {
    struct SkillData: Encodable {
        let id: Int
        let progress: Int
        let level: Int
        let shortName: String
        let category: Int
        let name: String

        init(skill: Skill) {
            self.id = skill.id
            self.progress = skill.progress
            self.level = skill.level
            self.shortName = skill.position?.shortName ?? "NA"
            self.category = 0
            self.name = skill.position?.name ?? "Beginner"
        }
    }

    struct UpdateSkillsBody: Encodable {
        let skillsData: [SkillData]

        enum CodingKeys: String, CodingKey {
            case skillsData = "positionSkills"
        }
    }

    struct UpdateSkillsResponse: Decodable {
        let state: String
    }

    let body = UpdateSkillsBody(skillsData: skills.map {
        SkillData(skill: $0)
    })

    var request = URLRequest(url: .updateSkills)
    request.httpMethod = "PATCH"
    request.setValue("application/json", forHTTPHeaderField:
    ↪  "Content-Type")
    request.httpBody = try JSONEncoder().encode(body)

    let (data, _) = try await self.session.data(for: request)
    let response = try
    ↪  JSONDecoder().decode(UpdateSkillsResponse.self, from:
    ↪  data)
    return response.state
}
```

Ukázka kódu 35: Mapování dovedností

```
@Observable
final class PlayersRowViewModel {
    var onToggleWaitlistClicked: () -> Void
    var onCheckInPlayerClicked: () -> Void
    var onCheckOutPlayerClicked: () -> Void
    var onKickPlayerClicked: () -> Void
    var onStatusButtonClicked: () -> Void
    var isThreeDotsMenuPresented = false

    //konstruktor vynechán
}
```

Ukázka kódu 36: Předávání funkcí do `PlayerListViewModel`

```
extension UserRole {
    var canAdjustPlayerStatus: Bool {
        switch self {
        case .owner, .admin, .organizer:
            return true
        default:
            //.user
            return false
        }
    }

    var canAdjustPlayerSkills: Bool {
        switch self {
        case .owner, .admin, .coach:
            return true
        default:
            //.user
            return false
        }
    }
}
```

Ukázka kódu 37: Autorizace podle `UserRole`

Kapitola 8

Zhodnocení výsledků

V této kapitole si zhodnotíme vývoj v jazyce Swift s využitím SwiftUI a výslednou mobilní aplikaci v kontextu využití praktik OOP a FP a jejich vlivu na udržitelnost aplikace. V poslední sekci se zaměříme na současný stav a další možný rozvoj aplikace.

Výsledkem této práce je analýza, návrh a následná imlementace mobilní aplikace za využití praktik OOP a FP. V minulé kapitole jsme si ukázali, jak byly při implementaci využity praktiky jako návrhové vzory, funkce vyšších řádů s uzávěrkami, návrhy protokolů, struktury konformující k těmto protokolům, rozšiření objektů a jiné.

8.1 Shrnutí funkcionalit řešení

Aplikace byla navržena pro členy volejbalového klubu Ace Volleyball Academy. Uživatelům umožňuje základní funkce jako přihlášení, odhlášení a sledování svých úrovní ve volejbalových dovednostech na svém profilu. Aplikace umožňuje zobrazovat nadcházející události a jejich podrobnosti včetně zobrazení příhlášených hráčů s možností zobrazení jejich profilu. Aplikace obsahuje funkcionality specifické podle uživatelské role. Pro organizátory aplikace umožňuje spravovat hráče v události a měnit jejich statusu. Pro trenéry aplikace umoňuje hráčům měnit jejich úrovně ve volejbalových dovednostech.

Vzhled i funkcionality aplikace vychází z části z rešerše existujících řešení. Inspiraci od aplikace Týmuj můžeme vidět v návrhu obrazovky s nadcházejícími událostmi a implementaci čekací fronty na události. U aplikace Volley World jsme se inspirovali detailem události a profilem uživatele, který umožňuje sledovat základní statistiky, úrovně a rezervace. Oproti těmto aplikacím ale naše aplikace umožňuje sledovat specifické dovednosti u hráčů, které jsou přímo spojené s volejbalovými pozicemi. Naše aplikace také slouží přímo pro účely Ace Volleyball Academy, tedy nelze v ní vytvářet soukromé skupiny ani události.

8.2 Využití FP a OOP

Jedna z hlavních praktik využívaných v OOP je objektově-orientovaný návrh, enkapsulace třídních dat a enkapsulace logiky do třídních metod. V této mo-

bilní aplikaci je většina procesů spojena s akcemi prováděnými na obrazovkách, jejichž logiku řeší v naší vybrané architektuře příslušný `ViewModel`. Pro modelaci jednotlivých entit (událost, uživatel, ...) poté využíváme struktury, abychom zabalili data do ucelených celků. Při výběru mezi třídou a strukturou pro toto modelování dat jsme v našem případě zvolili struktury, jelikož data bereme z externího zdroje a jejich identitu máme definovanou atributem (v našem případě atribut `id`), zároveň veškeré UI v aplikaci píšeme ve SwiftUI, tedy v jazyce Swift a nepotřebujeme podporu Objective-c [30]. Toto rozhodnutí vychází i z doporučení ve výše zmiňovaném článku. Struktury od sebe sice nemohou dědit, ale mohou konformovat k protokolům. Modelování chování podle protokolů a jejich dědičnosti je zároveň i doporučovaný způsob oproti třídám a třídní dědičnosti.

Přístupy FP, se kterými jsme se v této práci setkali, byly v jazyce Swift nativně podporovány formou uzávěrek, které ve Swiftu fungují jako first-class objekty a kolekce v jazyce Swift nativně podporovují většinu nejpoužívanějších funkcí vyššího řádu (`map`, `filter`, `reduce`, `forEach`, `contains`, ...). Při implementaci se nestalo, že by bylo potřeba využít metodu nad kolekcí, která by nebyla již nativně podporována. Pokud by se však stalo, že by bylo potřeba využít funkci nad určitou kolekcí, je možné si specifickou funkci doimplementovat a zakomponovat ji do původní kolekce díky rozšíření (`extension`).

Oba přístupy se vzájemně nevylučují, tedy můžeme je v jazyce Swift kombinovat. To ostatně také děláme v této aplikaci. Kód seskupujeme do struktur často navržených podle známých návrhových vzorů, konformujících k navrženým protokolům nebo obsahující funkcionality podle vybrané architektury. Do těchto struktur můžeme předávat funkce, které opouštějí jejich blok vytvoření a jsou volány později například pro předání informace struktuře nad ní. Jak je zmíněno výše, nad kolekcemi struktur jsou nativně podporované nejběžnější funkce vyšších řádu a v případě potřeby lze jednoduše přidat vlastní.

Aplikace je v současném návrhu dobře udržitelná. Příklad pro toto tvrzení si ukážeme na výměně implementace třídy komunikující s API. V současném návrhu by změna API znamenala napsání nové třídy konformující k protokolu `APIServicing`. Tuto implementaci bychom potom nahradili ve fasádě `APIFacade` za současnou implementaci (to můžeme udělat, jelikož ve fasádě pracujeme s protokolem a obě třídy k tomuto protokolu konformují) a volání ve fasádě tak budou nově delegovány na novou implementaci v celé aplikaci.

8.3 Budoucí rozvoj

Aplikace v současné chvíli splňuje všechny požadavky tak jak jsou využívány v případech užití, tyto případy užití aplikace naplňuje a je připravena na používání uživateli.

Nejbližší budoucí rozvoj aplikace popisují v současném stavu neimplementované požadavky: *FP1.3 – Registrace uživatele (could have)*, *FP8 – Správa událostí (could have)*, *FP9 – Notifikace (won't have)*. Podíváme se, jaké změny bychom v aplikaci museli udělat, abychom implementovali první zmíněný požadavek a tím také demonstrujeme její udržitelnost. V prvním kroku bychom navrhli novou obrazovku pro získání dat o novém uživateli. Poté bychom rozšířili protokol `APIServicing` o novou metodu pro registraci uživatele a implementovali bychom tuto metodu podle specifikací API. V tomto kroku bychom

také rozšířil strukturu `URL` o nový endpoint API. Tuto metodu bychom napojili na nějaký UI prvek na obrazovce, který by volal metodu na `ViewModelu` pro komunikaci s API. V posledním kroku bychom tuto obrazovku zapojili nějakou logikou mezi ostatní obrazovky. Podle provedené analýzy existujících řešení, bychom tuto obrazovku zobrazovali jako alternativní možnost vedle přihlášení. Po registraci bychom uživatele znovu vyzvali k přihlášení.

Závěr

Hlavním výsledkem této bakalářské práce je vyvinutí mobilní aplikace pro platformu iOS v multi-paradigmatickém jazyce Swift za využití praktik objektově-orientovaného a funkcionálního programování.

V rámci této práce byla analyzována doména organizování volejbalových tréninků a turnajů s využitím konceptuálního modelování. Po analýze domény byla provedena rešerše existujících řešení aplikací zabývajících se organizací sportovních událostí a spravováním uživatelů. Na základě provedené rešerše byl sestaven katalog oprioritizovaných požadavků a případy užití pro výslednou mobilní aplikaci. Před návrhem mobilní aplikace byly popsány možnosti vývoje v jazyce Swift s využitím praktik objektově-orientovaného a funcionálního programování a návrhových vzorů. V rámci návrhu aplikace byla vybrána architektura, byly navrženy datové třídy a uživatelské rozhraní aplikace. V posledním kroku byla provedena implementace s využitím vhodných návrhových vzorů, praktik funkcionálního a objektově-orientovaného programování a byl zhodnocen jejich vliv na udržitelnost aplikace.

Aplikace byla navržena pro členy volejbalového klubu Ace Volleyball Academy. Uživatelům umožňuje základní funkce jako přihlášení, odhlášení a sledování úrovní ve volejbalových dovednostech na svém profilu. Aplikace umožňuje zobrazovat nadcházející událostí a jejich podrobnosti včetně zobrazení účastnících se hráčů. Pro organizátory událostí aplikace umožňuje sledovat u hráčů jejich status a tento status také mohou jednoduše měnit. Pro trenéry poté aplikace poskytuje možnosti jak hráčům měnit úrovně ve volejbalových dovednostech.

Do budoucna autor práce plánuje mobilní aplikaci vydat na AppStore, aby si ji užívatelé mohli stáhnout a využívat. S autorem projektu AVA by pak autor chtěl mobilní aplikaci dále rozšiřovat o nové funkcionality a vylepšení. Autor by rád vyvinul i mobilní aplikaci pro uživatele na platformě Android.

Literatura

[1] EventServices: Týmuj. [online], 2024, [cit. 01-04-2024]. Dostupné z: https://apps.apple.com/cz/app/týmuj/id1218860308

[2] iSport: Demo rezervační systém. [online], 2024, [cit. 01-04-2024]. Dostupné z: https://demo.isportsystem.cz

[3] Knoegel, T.: Volley World - Play Volleyball. [online], 2024, [cit. 01-04-2024]. Dostupné z: https://apps.apple.com/us/app/volley-world-play-volleyball/id1537110255

[4] García, R. F.: *iOS Architecture Patterns: MVC, MVP, MVVM, VIPER, and VIP in Swift*. Apress, 2023, ISBN 978-1484290682.

[5] Císař, V.: *Volejbal*. Sport (Grada), Grada, 2005, ISBN 9788024705026.

[6] FIVB: Official Volleyball Rules 2021-2024. [online], 2021, [cit. 01-03-2024]. Dostupné z: https://www.fivb.com/-/media/2023/volleyball/files/fivb-volleyball_rules_2021_2024.pdf

[7] ČVS: Pravidla volejbalu. [online], 2021, [cit. 01-03-2024]. Dostupné z: https://www.cvf.cz/dokumenty/download/05_Pravidla/5-02_Volejbal/pravidla_volejbalu_2021-2024.pdf

[8] EventServices: Týmuj.cz. [online], 2024, [cit. 01-04-2024]. Dostupné z: https://www.tymuj.cz

[9] iSport System: Rezervační systém iSport. [online], 2024, [cit. 01-04-2024]. Dostupné z: https://www.isportsystem.cz

[10] Knögel, T.: Volley World App. [online], 2024, [cit. 01-04-2024]. Dostupné z: https://www.volleyworldapp.com/en

[11] Wiegers, K. E.; Beatty, J.: *Software requirements*. Pearson Education, 2013, ISBN 978-0-7356-7966-5.

[12] International Institute of Business Analysis: *A Guide to the Business Analysis Body of Knowledge*. Lightning Source Inc, 2009, ISBN 978-0-9811292-1-1.

[13] Apple Inc.: Swift. [online], 2024, [cit. 01-03-2024]. Dostupné z: https://developer.apple.com/swift

[14] Apple Inc.: The Swift Programming Language: About Swift. [online], 2024, [cit. 01-03-2024]. Dostupné z: https://docs.swift.org/swift-book/documentation/the-swift-programming-language/aboutswift

[15] Wegner, P.: Concepts and paradigms of object-oriented programming. *ACM Sigplan Oops Messenger*, ročník 1, č. 1, 1990: s. 7–87. Dostupné z: https://dl.acm.org/doi/pdf/10.1145/382192.383004

[16] Hoffman, J.: *Mastering Swift 5.3: Upgrade your knowledge and become an expert in the latest version of the Swift programming language*. Packt Publishing Ltd, 2020, ISBN 978-1-80056-215-8.

[17] Apple Inc.: The Swift Programming Language: Structures and Classes. [online], 2024, [cit. 23-04-2024]. Dostupné z: https://docs.swift.org/swift-book/documentation/the-swift-programming-language/classesandstructures

[18] Apple Inc.: Value and reference types in Swift. [online], 2024, [cit. 23-04-2024]. Dostupné z: https://www.swift.org/documentation/articles/value-and-reference-types.html

[19] Apple Inc.: The Swift Programming Language: Protocols. [online], 2024, [cit. 23-04-2024]. Dostupné z: https://docs.swift.org/swift-book/documentation/the-swift-programming-language/protocols

[20] Nwokoro, I.; Okonkwo, O.; Alo, U.; aj.: Object-Oriented Programming and Software Development Paradigm. *Academia Letters*, 09 2021, doi: 10.20935/AL3443.

[21] Hughes, J.: Why Functional Programming Matters. *The Computer Journal*, ročník 32, č. 2, 01 1989: s. 98–107, ISSN 0010-4620, doi:10.1093/comjnl/32.2.98. Dostupné z: https://doi.org/10.1093/comjnl/32.2.98

[22] Bonev, S.; Galletly, J.: Functional programming features supported by Kotlin and Swift. *Electrotechnica & Electronica (E+ E)*, ročník 54, 2019. Dostupné z: https://epluse.ceec.bg/wp-content/uploads/2019/11/20190506-02.pdf

[23] Eidhof, C.; Kugler, F.; Swierstra, W.: *Functional Programming in Swift*. Florian Kugler, 2014, ISBN 978-3-00-048005-8.

[24] Nayebi, F.: *Swift Functional Programming*. Packt Publishing Ltd, 2017, ISBN 978-1-78728-450-0.

[25] Apple Inc.: Apple Developer Documentation – Array. [online], 2024, [cit. 25-04-2024]. Dostupné z: https://developer.apple.com/documentation/swift/array

[26] Gamma, E.; Helm, R.; Johnson, R.; aj.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994, ISBN 978-0201633610.

[27] Freeman, A.: *Pro Design Patterns in Swift*. Apress, 2015, ISBN 978-1484203958.

[28] Apple Inc.: Apple Developer Documentation – SwiftUI. [online], 2024, [cit. 03-05-2024]. Dostupné z: `https://developer.apple.com/documentation/swiftui`

[29] Apple Inc.: Apple Developer Documentation – SwiftUI: View. [online], 2024, [cit. 03-05-2024]. Dostupné z: `https://developer.apple.com/documentation/swiftui/view`

[30] Apple Inc.: Apple Developer – Choosing Between Structures and Classes. [online], 2024, [cit. 10-05-2024]. Dostupné z: `https://developer.apple.com/documentation/swift/choosing-between-structures-and-classes`

Příloha A

Seznam použitých zkratek

API Application Programming Interface

AVA Ace Volleyball Academy

ČVS Český volejbalový svaz

FIVB Fédération Internationale de Volleyball

FP Funkcionální programování

FP Funkční požadavek

JSON JavaScript Object Notation

L Libero

MB Middle Blocker

MVC Model-View-Controller

MVP Model-View-Presenter

MVVM Model-View-ViewModel

NP Nefunkční požadavek

OH Outside Hitter

OOP Objektově-orientované programování

OP Opposite

OS Operační systém

S Setter

UC Use Case

UI User Interface

URL Uniform Resource Locator

VIP View-Interactor-Presenter

VIPER View-Interactor-Presenter-Entity-Repository

Příloha **B**

Obsah přiloženého média

- `readme.md` ..... návod k obsahu adresáře
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF