

Moderní metodologie při vývoji komplexních mobilních aplikací ve SwiftUI

Bc. Alexandr Čížek

Diplomová práce
2025



Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně

Fakulta aplikované informatiky

Ústav informatiky a umělé inteligence

Akademický rok: 2024/2025

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	Bc. Alexandr Čížek
Osobní číslo:	A23425
Studijní program:	N0613A140022 Informační technologie
Specializace:	Softwarové inženýrství
Forma studia:	Kombinovaná
Téma práce:	Moderní metodologie při vývoji komplexních mobilních aplikací ve SwiftUI
Téma práce anglicky:	Modern Methodology in the Development of Complex Mobile Applications in SwiftUI

Zásady pro vypracování

- Prostudujte aktuální trendy a metodiky vývoje aplikací na platformě iOS, se zaměřením na SwiftUI a agilní přístup k vývoji. Zvláštní důraz věnujte návrhovým vzorům a principům čistého kódu (Clean Code).
- Vypracujte přehled doporučených metodik pro správu vývojového cyklu, včetně tvorby architektury aplikace, dokumentace, strategie verzování a distribuce aplikace, s ohledem na požadavky moderního iOS vývoje.
- Navrhněte optimální strukturu aplikace a komponent s důrazem na modulární přístup, čitelnost a rozšiřitelnost kódu, pomocí praktických příkladů z reálného vývoje iOS aplikací.
- Analyzujte postupy testování a interní kontroly kvality, se zaměřením na jednotkové testy, testování uživatelského rozhraní, blackbox testování a strategii verzování v rámci agilního vývoje.
- Zhodnoťte legislativní a standardní požadavky pro distribuci aplikací na platformě iOS, včetně pravidel App Store a požadavků na ochranu osobních údajů a navrhněte postupy, jak tyto požadavky splnit v rámci vývojového cyklu.

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam doporučené literatury:

1. BELLO, Antonio; MOREFIELD, Bill; REICHEL, Sarah a TAM, Audrey. SwiftUI by Tutorials: Declarative App Development on the Apple Ecosystem. Fifth Edition. Kodeco, 2023. ISBN 1950325881.
2. LODI, Gio. Test-Driven Development in Swift: Compile Better Code with XCTest and TDD. Apress, 2021. ISBN 1484270010.
3. LAYTON, Mark C.; OSTERMILLER, Steven J. a KYNASTON, Dean J. Agile Project Management For Dummies. 3rd Edition. For Dummies, 2020. ISBN 1119676991.
4. HUDSON, Paul. Testing Swift. Online. Hacking with Swift, c2024. Dostupné z: <https://www.hackingwithswift.com/books/testing>. [cit. 2024-10-24].
5. VILMART, Florent; DE SIMONE, Sergio a SCALZO, Giordano. Hands-On Design Patterns with Swift. Packt Publishing, 2018. ISBN 1789135567.

Vedoucí diplomové práce: **Ing. Bc. Pavel Vařacha, Ph.D.**
Ústav informatiky a umělé inteligence

Datum zadání diplomové práce: **27. října 2024**
Termín odevzdání diplomové práce: **2. června 2025**



L.S.

doc. Ing. Jiří Vojtěšek, Ph.D. v.r.
děkan

prof. Mgr. Roman Jašek, Ph.D., DBA v.r.
ředitel ústavu

Ve Zlíně dne 17. listopadu 2024

PROHLÁŠENÍ AUTORA DIPLOMOVÉ PRÁCE

Beru na vědomí, že

- odevzdáním diplomové práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb., v platném znění bez ohledu na výsledek obhajoby;
- diplomová práce bude uložena v elektronické podobě v univerzitním informačním systému a bude dostupná k nahlédnutí;
- jedno vyhotovení diplomové práce v listinné podobě bude ponecháno Univerzitě Tomáše Bati ve Zlíně k uložení;
- na moji diplomovou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- podle § 60 odst. 1 autorského zákona má Univerzita Tomáše Bati ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- podle § 60 odst. 2 a 3 mohu užít své dílo – diplomovou práci – nebo poskytnout licenci k jejímu využití jen s předchozím písemným souhlasem Univerzity Tomáše Bati ve Zlíně, která je oprávněna v takovém případě ode mne požadovat přiměřený příspěvek na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše);
- pokud bylo k vypracování diplomové práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tj. k nekomerčnímu využití), nelze výsledky diplomové práce využít ke komerčním účelům;
- pokud je výstupem diplomové práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá; neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji, že

- jsem na diplomové práci pracoval(a) samostatně a použitou literaturu jsem řádně citoval(a); v případě publikace výsledků budu uveden(a) jako spoluautor;
- odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou obsahově totožné.

Ve Zlíně, dne

.....
podpis autora

ABSTRAKT

Tato diplomová práce se zabývá moderními metodologiemi a přístupy při vývoji komplexních mobilních aplikací pro platformu iOS se zaměřením na framework SwiftUI. V teoretické části jsou popsány klíčové principy návrhových vzorů, testování, clean code, správy verzí a agilního vývoje. Praktická část demonstruje aplikaci těchto principů na reálné mobilní aplikaci Votap, která je dostupná v App Store a umožňuje uživatelům anonymně hodnotit politiky v reálném čase. Důraz je kladen na použití architektury MVVM, správnou organizaci projektu, testovatelnost, dokumentaci pomocí nástroje DocC a zajištění souladu s požadavky legislativy včetně GDPR.

Klíčová slova: SwiftUI, iOS, MVVM, Git, Clean Code, Testování, Agilní vývoj, Votap

ABSTRACT

This thesis explores modern methodologies and approaches in the development of complex mobile applications for the iOS platform, with a focus on the SwiftUI framework. The theoretical section discusses key concepts such as design patterns, testing, clean code principles, version control, and agile development. The practical part demonstrates the application of these principles on a real-world mobile application called Votap, available on the App Store, which allows users to anonymously rate politicians in real time. Emphasis is placed on the use of the MVVM architecture, proper project structure, testability, documentation using DocC, and compliance with privacy legislation such as the GDPR.

Keywords: SwiftUI, iOS, MVVM, Git, Clean Code, Testing, Agile Development, Votap

OBSAH

ÚVOD.....	13
I TEORETICKÁ ČÁST	14
1 NÁSTROJE PRO VÝVOJ	15
1.1 SWIFT	15
1.2 PŘECHOD Z OBJECTIVE-C NA SWIFT	17
1.3 UIKIT.....	18
1.4 SWIFTUI.....	19
1.5 SROVNÁNÍ UIKIT A SWIFTUI	20
1.6 XCODE	21
1.7 PROGRAM APPLE DEVELOPER.....	22
2 NÁVRHOVÝ VZOR MVVM.....	24
2.1 MODEL	24
2.2 VIEW	24
2.3 VIEW MODEL	24
3 PRINCIPY SOLID	26
3.1 PRINCIP JEDNÉ ODPOVĚDNOSTI.....	26
3.2 PRINCIP OTEVŘENOSTI A UZAVŘENOSTI.....	26
3.3 LISKOVŮV PRINCIP ZAMĚNITELNOSTI	26
3.4 PRINCIP ODDĚLENÍ ROZHRAŇÍ	27
3.5 PRINCIP OBRÁCENÍ ZÁVISLOSTÍ	27
4 CLEAN CODE	28
4.1 ČITELNOST A SROZUMITELNOST	28
4.2 FUNKCE A JEJICH VELIKOST	28
4.3 KÓD BEZ DUPLICIT	29
4.4 SMYSLUPLNÉ KOMENTÁŘE A DOKUMENTACE.....	29
4.5 KONZISTENCE A STANDARDY	29
4.6 REFAKTORING A UDRŽOVÁNÍ KÓDU	29
4.7 TESTOVATELNOST A STABILITA.....	30
5 AGILNÍ VÝVOJ.....	31
5.1 ZÁKLADNÍ HODNOTY.....	31
5.2 VÝVOJOVÝ CYKLUS.....	31
5.3 METODIKY	32
6 VERZOVÁNÍ A GIT	34

6.1	ZÁKLADNÍ FUNKCE SYSTÉMU GIT.....	34
6.2	GITHUB.....	35
6.3	STRATEGIE VĚTVENÍ.....	35
6.4	SLUČOVÁNÍ VĚTVÍ.....	36
7	TESTOVÁNÍ.....	37
7.1	FUNKCIONÁLNÍ A NEFUNKCIONÁLNÍ TESTOVÁNÍ	37
7.2	WHITE-BOX A BLACK-BOX TESTOVÁNÍ.....	37
7.3	HLAVNÍ ÚROVNĚ TESTOVÁNÍ.....	37
7.4	FRAMEWORK XCTEST.....	38
7.5	TESTFLIGHT	39
8	DOKUMENTACE.....	40
8.1	ZÁKLADNÍ ROZDĚLENÍ.....	40
8.2	TVORBA DOKUMENTACE V XCODE.....	41
9	POŽADAVKY PLATFORMY APP STORE A LEGISLATIVA.....	42
9.1	POKYNY PRO KONTROLU APLIKACÍ.....	42
9.2	OSOBNÍ ÚDAJE UŽIVATELŮ	43
9.3	OBECNÉ NAŘÍZENÍ O OCHRANĚ OSOBNÍCH ÚDAJŮ	44
II	PRAKTICKÁ ČÁST.....	45
10	ÚVOD K APLIKACI VOTAP	46
10.1	CÍL A ZAMĚŘENÍ APLIKACE.....	46
10.2	FUNKCIONALITA A KLÍČOVÉ VLASTNOSTI.....	47
11	ARCHITEKTURA APLIKACE	50
11.1	IMPLEMENTACE VZORU MVVM.....	50
11.1.1	Model.....	50
11.1.2	View	51
11.1.3	View model	52
11.2	SERVICE TŘÍDY	53
11.3	DATOVÉ REPOZITÁŘE	54
11.4	DEPENDENCY INJECTION VE VIEW MODELECH	55
11.5	ZNOVUPOUŽITELNÉ VIEW KOMPONENTY.....	56
11.6	POMOCNÉ NÁSTROJE.....	57
11.7	ASSETS	58
12	APLIKACE PRINCIPŮ CLEAN CODE A SOLID.....	60
12.1	UPLATNĚNÍ PRINCIPŮ SOLID	60
12.1.1	Princip jedné odpovědnosti	60
12.1.2	Princip otevřenosti a uzavřenosti	61

12.1.3	Liskovův princip substituce	62
12.1.4	Princip oddělení rozhraní	64
12.1.5	Princip obrácení závislostí	65
12.2	POMOCNÉ METODY	66
12.3	PSÁNÍ KOMENTÁŘŮ	67
12.4	PROCES REFAKTORINGU	68
13	STRUKTURA PROJEKTU	71
13.1	ORGANIZACE A POJMENOVÁNÍ SOUBORŮ	71
13.2	MODULARIZACE A ODDĚLENÍ DOMÉN	74
13.3	PRÁCE S KONFIGURACÍ A PROSTŘEDÍM	75
13.4	PROJEKTOVÁ KULTURA A NÁSTROJ SWIFTLINT	76
14	DOKUMENTACE.....	78
14.1	Použití DocC v XCODE	78
14.2	UKÁZKA STRUKTUROVANÉ DOKUMENTACE	78
14.3	UDRŽOVÁNÍ A AKTUALIZACE	80
15	VERZOVÁNÍ.....	82
15.1	KLONOVÁNÍ REPOZITÁŘE	82
15.2	INTEGRACE SYSTÉMU GIT V XCODE.....	83
15.3	VĚTVENÍ A PULL REQUESTS	84
16	TESTOVÁNÍ	86
16.1	UNIT TESTY	86
16.1.1	Struktura a pojmenování	86
16.1.2	View modely	87
16.1.3	Mock objekty	88
16.1.4	Asynchronní metody	89
16.2	UI TESTY VE SWIFTUI	90
16.3	NÁVRH BLACK-BOX TESTŮ	91
16.4	ALPHA A BETA TESTOVÁNÍ POMOCÍ TESTFLIGHT	92
17	AGILNÍ VÝVOJOVÝ NÁVRH PRO TÝM	94
17.1	ROZDĚLENÍ ROLÍ V TÝMU	94
17.1.1	Product manager.....	94
17.1.2	UI/UX designer	95
17.1.3	Vývojář.....	95
17.1.4	Tester.....	95
17.2	PRÁCE VE SPRINTech A PLÁNOVÁNÍ.....	95
17.3	STRATEGIE VERZOVÁNÍ	96
17.4	SPOLUPRÁCE POMOCÍ NÁSTROJŮ	97
17.5	PRŮBĚH JEDNÉ KONKRÉTNÍ ITERACE	98

17.5.1	Komunikace product managera se stakeholdery	99
17.5.2	Příprava backlogu a plánování sprintu	99
17.5.3	Vývoj a implementace během sprintu	100
17.5.4	Testování v rámci sprintu	101
17.5.5	Průběžná komunikace a aktualizace úkolů.....	102
17.5.6	Závěr sprintu a retrospektiva	103
18	DISTRIBUCE A LEGISLATIVA.....	105
18.1	PROCES NASAZENÍ APLIKACE NA APP STORE.....	105
18.1.1	Nahrávání aplikace na App Store Connect.....	105
18.1.2	Schvalovací proces	106
18.1.3	Vydávání aktualizací	106
18.2	PRÁCE S OSOBNÍMI ÚDAJI A OCHRANA SOUKROMÍ	107
18.2.1	Shromažďovaná data	107
18.2.2	Smazání účtu a všech osobních údajů	107
18.2.3	Požadavky GDPR.....	107
19	SHRNUTÍ.....	109
19.1	ZHODNOCENÍ PŘÍSTUPU	109
19.2	MOŽNOSTI ROZŠÍŘENÍ APLIKACE	109
19.3	DOPORUČENÍ PRO TÝMOVÝ VÝVOJ	110
ZÁVĚR.....		111
SEZNAM POUŽITÉ LITERATURY		113
SEZNAM OBRÁZKŮ		116
SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK		118

ÚVOD

Vývoj moderních mobilních aplikací je stále komplexnější disciplínou, která vyžaduje znalost nejen programovacích jazyků a frameworků (softwarový rámec, který poskytuje předpřipravenou strukturu a nástroje pro vývoj aplikací), ale také správné architektury, principů udržitelného kódu, týmové spolupráce a efektivních metod testování. Cílem této práce je propojit teoretické poznatky s jejich praktickou aplikací na konkrétním příkladu – vývoji mobilní aplikace Votap pro platformu iOS ve SwiftUI.

Práce nejprve představuje základní technologie a metodologie používané při profesionálním vývoji mobilních aplikací, včetně principů jako MVVM (architektonický vzor pro oddělení datové logiky, uživatelského rozhraní a jejich propojení), SOLID (soubor pěti základních principů objektově orientovaného návrhu), Clean code (čistý, čitelný a snadno udržitelný kód), agilní vývoj a strategie verzování. Následně je demonstrováno, jak lze tyto principy uplatnit v reálném projektu včetně konkrétní implementace architektury, testování, dokumentace a procesů týmové spolupráce. Výsledkem je ucelený pohled na to, jak efektivně navrhnout, vyvíjet, spravovat a rozšiřovat komplexní mobilní aplikaci v souladu s moderními standardy pro vývoj ve SwiftUI.

I TEORETICKÁ ČÁST

1 NÁSTROJE PRO VÝVOJ

Vývoj mobilních aplikací pro platformu iOS se v průběhu let výrazně změnil, a to nejen díky technologickým inovacím společnosti Apple, ale také kvůli měnícím se potřebám vývojářů a rostoucím požadavkům na moderní aplikace. S nástupem nových technologií a frameworků se změnil nejen způsob, jakým se aplikace vyvíjejí, ale i jejich samotná architektura, výkon a uživatelská přívětivost. Apple dlouhodobě klade důraz na ekosystém svých nástrojů a poskytuje vývojářům vše potřebné pro efektivní tvorbu aplikací, od programovacích jazyků až po vývojová prostředí a nástroje pro distribuci softwaru.

Tato kapitola popisuje klíčové technologie a nástroje, které jsou v současnosti nezbytné pro vývoj iOS aplikací. Úvodní část se věnuje historickému vývoji s cílem objasnit, jaké změny přinesly jednotlivé generace softwarových nástrojů a z jakých důvodů Apple přešel od jazyka Objective-C k modernějšímu jazyku Swift. Následuje detailní rozbor jazyka Swift a frameworku SwiftUI, které dnes hrají zásadní roli v tvorbě aplikací, včetně srovnání se starším přístupem pomocí UIKit.

Dále je věnována pozornost vývojovému prostředí Xcode, které je základním nástrojem při tvorbě iOS aplikací, a také významu vývojářského účtu Apple Developer, který je nezbytný pro testování a distribuci aplikací prostřednictvím App Store. Popsané technologie a nástroje představují základ moderního vývoje pro iOS a jejich důkladné porozumění je zásadní pro tvorbu efektivních, bezpečných a výkonných aplikací.

1.1 Swift

Swift je moderní programovací jazyk vyvinutý společností Apple pro vývoj aplikací na iOS, macOS, watchOS a tvOS. Byl navržen s důrazem na bezpečnost, rychlost a snadnou čitelnost kódu. Od svého uvedení v roce 2014 prošel značným vývojem a stal se klíčovou součástí moderního vývoje aplikací pro Apple platformy.

Swift kombinuje vlastnosti objektově orientovaného, funkcionálního, imperativního a blokově strukturovaného programování, což vývojářům umožňuje psát čistý, efektivní a výkonný kód. Využívá moderní koncepty, jako jsou „closures” (podobné lambda funkcím), rozšířená práce s typy (tuples, type inference), protokoly a pokročilá správa paměti prostřednictvím automatického počítání referencí (ARC).

Jedním z klíčových rysů jazyka Swift je jeho bezpečnost. Díky přísné typové kontrole a práci s volitelnými hodnotami (optionals) eliminuje běžné chyby, jako jsou dereference

nulových ukazatelů nebo přetečení polí. Kromě toho jeho syntaxe podporuje pokročilé kontrolní mechanismy, včetně klíčových slov jako „do“, „guard“, „defer“ a „repeat“, které umožňují efektivnější a bezpečnější řízení toku programu.

Velkou výhodou jazyka Swift je jeho otevřenost – od roku 2015 je dostupný jako open-source projekt. Komunita vývojářů se aktivně podílí na jeho vylepšování a rozšiřování, čímž se stává čím dál více univerzálním jazykem.

Swift je navržen tak, aby byl co nejvíce přístupný začátečníkům. Díky jednoduché a intuitivní syntaxi je vhodný jako první jazyk pro nové programátory, a zároveň nabízí dostatečně pokročilé funkce pro profesionální vývojáře. Apple podporuje jeho výuku prostřednictvím nástrojů, jako je Swift Playgrounds, který umožňuje interaktivní učení a rychlé testování kódu.

Swift přináší mnoho funkcí, které ho činí výkonným a flexibilním jazykem. Closures, podobné lambda funkcím v jiných jazycích, umožňují efektivní práci s anonymními funkcemi. Tuples a vícenásobné návratové hodnoty umožňují funkcím vracet více hodnot současně, čímž odstraňují nutnost vytvářet složité datové struktury. Swift také podporuje iteraci přes kolekce, což usnadňuje práci s datovými strukturami prostřednictvím „for-in“ smyček.

Další důležitou vlastností jazyka Swift jsou struktury a třídy, které umožňují použití metod, rozšíření (extensions) a protokolů, čímž zvyšují modularitu kódu. Swift také obsahuje funkcionální programovací vzory, jako jsou „map“, „filter“ a „reduce“, které zjednodušují operace nad kolekcemi.

Pro zajištění bezpečnosti a robustnosti kódu Swift nabízí pokročilé zpracování chyb pomocí klíčových slov „do“, „try“ a „catch“. Další klíčovou funkcí jsou volitelné hodnoty (optionals), které pomáhají předcházet chybám spojeným s „nil“ hodnotami. Swift dále usnadňuje vývoj díky odvozování typů, což znamená, že kompilátor může automaticky odvozovat datové typy proměnných, což vede k přehlednějšímu a stručnějšímu kódu.

Správa paměti v jazyce je založena na mechanismu automatického počítání referencí (Automatic Reference Counting). Tento systém sleduje počet referencí na konkrétní objekt a jakmile již není objekt potřebný, automaticky uvolní jeho paměť. Swift zajišťuje automatickou správu paměti, což znamená, že programátor se nemusí starat o ruční alokaci a dealokaci objektů. Tento mechanismus pomáhá předcházet únikům paměti, protože pokud je ARC správně používáno, minimalizuje riziko nechtěného držení objektů v paměti. Kromě

toho zvyšuje výkon aplikací, protože efektivní správa paměti optimalizuje využití systémových zdrojů a přispívá k plynulosti běhu aplikace. Jedním z hlavních úskalí ARC jsou tzv. „silné“ referenční cykly. Pokud dva objekty drží silné reference na sebe navzájem, ARC je nikdy neuvolní, což vede k únikům paměti. Tento problém lze řešit použitím tzv. „slabých“ referencí (weak) nebo „nevlastních“ referencí (unowned), které umožňují přerušování cyklu a správné uvolnění paměti. [1] [2]

1.2 Přejít z Objective-C na Swift

Objective-C byl dlouhá léta hlavním programovacím jazykem pro vývoj aplikací v ekosystému Apple. Tento jazyk vznikl v 80. letech jako nadstavba jazyka C a díky své objektově orientované povaze umožňoval efektivní vývoj aplikací pro macOS a později i pro iOS. V roce 2014 však Apple představil nový jazyk Swift, který byl navržen tak, aby odstranil některé nevýhody Objective-C a zjednodušil vývoj aplikací. Přechod od Objective-C k jazyku Swift znamenal zásadní změnu nejen v syntaxi, ale i v přístupu k bezpečnosti, správě paměti a celkové efektivitě vývoje.

Jedním z hlavních rozdílů mezi těmito jazyky je jejich syntaktická struktura. Objective-C využívá zastaralou syntaxi, což se projevuje zejména v použití složitých konstrukcí, například při volání metod pomocí hranatých závorek. Naproti tomu Swift nabízí moderní, čitelnější a kompaktnější syntaxi, která se více podobá jazykům jako Python či JavaScript. Díky tomu je Swift snazší na naučení a umožňuje rychlejší vývoj bez zbytečné složitosti.

Dalším významným rozdílem je způsob deklarace tříd a souborová struktura. V Objective-C musí být každá třída definována ve dvou souborech: v hlavičkovém souboru s příponou „h“ a v implementačním souboru s příponou „m“. Toto rozdělení je historicky dané a vychází z jazyka C. Naproti tomu Swift kombinuje deklaraci a implementaci do jednoho souboru, což zjednodušuje správu kódu a urychluje vývoj. Tímto způsobem odpadá nutnost manuální synchronizace mezi hlavičkovým a implementačním souborem, což je častý zdroj chyb v Objective-C.

Přejít na Swift přinesl také vylepšení v oblasti bezpečnosti a stability kódu. Objective-C umožňuje práci s ukazateli, což s sebou přináší riziko neplatných referencí a následných pádů aplikace. Swift naopak pracuje s bezpečnějšími mechanismy, jako jsou optionals, které nutí programátora explicitně řešit případy, kdy hodnota může být nulová. Tím se snižuje pravděpodobnost chyb způsobených neočekávanými hodnotami a aplikace jsou celkově stabilnější.

I přes všechny výhody jazyka Swift je Objective-C v určitých situacích stále relevantní. Mnoho starších aplikací a rozsáhlých kódových základů je napsáno v Objective-C, a přepisování do jazyka Swift by bylo nákladné a časově náročné. Navíc Objective-C nabízí přímou interoperabilitu s jazykem C a C++, což je výhodné pro vývojáře, kteří pracují s těmito technologiemi. Přesto je trend jasný – většina nových iOS aplikací se vyvíjí v jazyce Swift a Apple ho aktivně podporuje. Díky pravidelným aktualizacím a novým technologiím, jako je SwiftUI pro tvorbu uživatelského rozhraní, se Swift stává stále populárnější volbou pro vývoj mobilních aplikací. [3] [4]

1.3 UIKit

UIKit je jedním ze základních frameworků společnosti Apple určených pro tvorbu uživatelského rozhraní na zařízeních s iOS. Přestože v roce 2019 získal na popularitě SwiftUI, UIKit zůstává široce využívaným nástrojem díky své dlouhé historii, stabilitě a rozsáhlým možnostem. Mnoho aplikací v App Store stále využívá UIKit, a proto je jeho znalost klíčová pro každého, kdo se chce stát profesionálním iOS vývojářem. I v projektech založených na SwiftUI se často setkáváme s potřebou integrace nebo úprav komponent vytvořených v UIKit.

Framework UIKit je součástí sady CocoaTouch, která byla představena spolu s prvním oficiálním vydáním iOS (tehdy iPhoneOS) v roce 2008. Již předtím byl UIKit interně používán v prvních dvou verzích iOS. Na rozdíl od SwiftUI, který byl od začátku navržen se zaměřením na Swift, byl UIKit původně vyvíjen v jazyce Objective-C. I když dnes většina vývojářů preferuje Swift, UIKit je stále úzce spjat s Objective-C, což znamená, že při práci s tímto frameworkem se často setkáme s kódem v tomto starším jazyce.

Při vytváření projektu s UIKit narazíme na několik klíčových komponent, které tvoří jeho strukturu. Třída „AppDelegate” slouží jako vstupní bod aplikace, zatímco „SceneDelegate” se stará o správu obrazovek a změny stavů aplikace. Uživatelské rozhraní lze vytvářet vizuálně pomocí tzv. „Storyboardů”, kde je možné navrhovat pohledy (View) a nastavovat jejich vlastnosti. Každý pohled je řízen pomocí tzv. „View controlleru”, který spravuje jeho životní cyklus a logiku. Tento přístup je úzce spojen s architekturou Model-View-Controller (MVC), kde Model zpracovává data aplikace, View zajišťuje jejich vizuální zobrazení a Controller slouží jako spojovací článek mezi nimi, reaguje na uživatelské interakce a aktualizuje data podle potřeby.

Jednou z výhod UIKit je možnost tvorby rozhraní pomocí Storyboardů, kde lze jednoduše přidávat prvky, jako jsou tlačítka nebo text, a následně je propojit s controllerem. Tímto způsobem lze snadno reagovat na uživatelské akce a měnit vlastnosti prvků dynamicky.

Díky své bohaté funkcionalitě a prověřené stabilitě představuje UIKit i nadále významný nástroj pro vývoj iOS aplikací. Nabízí kompletní prostředí pro tvorbu uživatelských rozhraní, zpracování dotykových gest a interakci mezi aplikací, uživatelem a systémem. Přestože se Apple stále více zaměřuje na SwiftUI jako na modernější alternativu, UIKit zůstává nepostradatelným frameworkem, jehož znalost je důležitá pro každého vývojáře, který se chce pohybovat ve světě iOS aplikací. [5] [6]

1.4 SwiftUI

SwiftUI je relativně nový framework společnosti Apple, který byl poprvé představen v roce 2019 spolu s verzí iOS 13. Přestože zpočátku obsahoval určitá omezení, která zpomalila jeho adopci, postupně se vyvinul do zralého nástroje vhodného pro většinu vývojářů zaměřených na novější verze iOS. Apple vytvořil SwiftUI s cílem výrazně zjednodušit a zpřístupnit vývoj uživatelského rozhraní napříč svými platformami, tedy nejen pro iOS, ale i pro macOS, tvOS a watchOS.

Zásadním rysem SwiftUI je jeho deklarativní přístup k tvorbě uživatelských rozhraní. Místo manuálního nastavování a správy komponent, jako tomu bylo u tradičních nástrojů, zde vývojář pouze definuje, jak má uživatelské rozhraní vypadat a chovat se. Když se změní stav aplikace, SwiftUI automaticky provede potřebné aktualizace v uživatelském rozhraní, aniž by vývojář musel manuálně řídit jednotlivé změny. Tento přístup vede k přehlednějšímu a méně chybovému kódu.

SwiftUI je integrovaný s jazykem Swift, což znamená, že využívá jeho moderní vlastnosti, jakými jsou například generika, protokoly nebo funkcionální programování. Apple musel nejprve rozšířit možnosti samotného jazyka Swift, aby bylo možné SwiftUI vůbec vytvořit. Od uvedení jazyka Swift v roce 2014 prošel tento jazyk řadou zásadních změn a vývojáři, kteří s ním pracovali od jeho počátků, si dobře pamatují složité migrace mezi jednotlivými verzemi.

Další klíčovou výhodou SwiftUI je jeho podpora moderních funkcí, jako jsou animace, gesta nebo vlastní uživatelské komponenty. Také přináší vestavěnou podporu adaptivních

rozvržení komponent, což vývojářům usnadňuje vytváření aplikací, které se přizpůsobí různým velikostem obrazovek a uživatelským potřebám. Díky funkci „Live Preview“ je možné okamžitě vidět výsledky změn v kódu bez nutnosti opakovaného kompilování a spouštění aplikace, což výrazně zrychluje vývojový proces.

Přestože SwiftUI přináší mnoho výhod, má i své nevýhody. Vzhledem k tomu, že jde stále o poměrně nový framework, není jeho podpora ve starších verzích iOS ideální a některé funkce jsou dostupné pouze v nejnovějších verzích systému. Kromě toho komunita kolem SwiftUI ještě není tak rozsáhlá jako u starších technologií, což může znamenat obtížnější hledání řešení problémů.

Navzdory omezením Apple jednoznačně směřuje k tomu, aby se SwiftUI stal budoucností vývoje uživatelských rozhraní pro všechny jeho platformy. S každou novou verzí iOS se framework rozšiřuje a získává nové možnosti, což naznačuje, že jeho význam bude v budoucnu jen růst. Vývojáři, kteří se nyní naučí SwiftUI, tak získají výhodu při tvorbě moderních a efektivních aplikací, které budou plně využívat nejnovější technologie společnosti Apple. [7]

1.5 Srovnání UIKit a SwiftUI

SwiftUI a UIKit jsou dva hlavní frameworky pro tvorbu uživatelských rozhraní v ekosystému Apple. Každý z nich přistupuje k vývoji UI odlišným způsobem a jejich výběr závisí na konkrétních požadavcích projektu.

UIKit umožňuje vytvářet uživatelské rozhraní jak programově, tak pomocí tzv. „Interface Builderu“ v Xcode. Interface Builder nabízí vizuální nástroje pro návrh rozhraní pomocí přetahování prvků. SwiftUI tento přístup opouští a spoléhá na Live Preview, který umožňuje vizuální kontrolu uživatelského rozhraní (UI) přímo při psaní kódu. Změny se okamžitě zobrazují v náhledu, což zrychluje iteraci návrhu. UIKit je imperativní framework, kde vývojář musí explicitně definovat, kdy a jak se má UI aktualizovat. Naproti tomu SwiftUI využívá deklarativní přístup, kde se definují pouze stavy a framework sám provádí aktualizace na základě změn ve stavovém modelu aplikace. Tento koncept může být zpočátku náročnější na pochopení pro vývojáře přecházející z UIKit.

UIKit má dlouhou historii a rozsáhlou komunitu, což znamená, že existuje mnoho dostupné dokumentace a řešení běžných problémů. Naproti tomu SwiftUI je novější a komunita kolem něj stále roste, což může znamenat omezenější dostupnost zdrojů a podpory.

SwiftUI vyžaduje minimálně iOS 13, což může být omezení pro aplikace, které potřebují podporovat starší verze. UIKit naproti tomu umožňuje podporu až do iOS 9, což jej činí vhodnějším pro aplikace s širším dosahem. Rychlost vývoje SwiftUI umožňuje rychlejší vývoj díky menšímu množství kódu potřebnému k dosažení stejného výsledku jako v UIKit. Live Preview navíc zrychluje iteraci návrhu. Nicméně SwiftUI zatím neobsahuje všechny možnosti UIKit, což může znamenat nutnost použití rozhraní „UIViewRepresentable” pro zpětnou integraci chybějících funkcí.

SwiftUI nabízí vestavěnou podporu animací a přechodů, které jsou snadno implementovatelné a vyžadují méně kódu než v UIKit, kde je nutné animace definovat manuálně. SwiftUI je navržen jako multiplatformní framework, což znamená, že stejný kód lze snadněji sdílet mezi iOS, macOS, watchOS a tvOS. UIKit je primárně zaměřen na iOS a vyžaduje samostatné implementace pro ostatní platformy.

Oba frameworky lze kombinovat. UIKit prvky lze vkládat do SwiftUI pomocí rozhraní „UIViewRepresentable” a SwiftUI komponenty lze vkládat do UIKit přes rozhraní „UIHostingController”. Tato interoperabilita však přináší určitou složitost, proto je obvykle preferováno použití pouze jednoho z těchto frameworků.

Volba frameworku pro nový projekt Výběr mezi SwiftUI a UIKit závisí na konkrétním projektu. UIKit je stabilnější a vhodnější pro aplikace s širokou zpětnou kompatibilitou a bohatými funkcemi. SwiftUI je modernější, umožňuje rychlejší vývoj a je perspektivní pro budoucnost, ale zatím má určitá omezení. Apple jasně naznačuje, že SwiftUI je budoucností vývoje UI na jeho platformách. I když UIKit je momentálně stále dominantní, vývojáři by se měli s SwiftUI seznámit, protože jeho význam bude s každou novou verzí iOS růst.

1.6 Xcode

Xcode je integrované vývojové prostředí (IDE) od společnosti Apple, které je klíčovým nástrojem pro tvorbu aplikací na Apple platformách. Od svého prvního vydání v roce 2003 se neustále vyvíjí a dnes patří mezi nejdůležitější nástroje pro vývojáře, kteří chtějí pracovat v ekosystému Apple. Podporuje především programovací jazyky Swift a Objective-C, ale lze v něm pracovat i s dalšími, jako jsou C nebo C++. Jeho největší výhodou je nejen široká dostupnost, ale i rozsáhlá podpora ze strany společnosti Apple, která zajišťuje snadnou integraci s jeho technologiemi a službami.

Xcode umožňuje efektivní vývoj a správu aplikací díky svému výkonnému editoru, interaktivním náhledům uživatelského rozhraní a pokročilým nástrojům pro ladění a analýzu výkonu. Vývojáři mohou využívat nástroj Xcode Cloud, který usnadňuje spolupráci a automatizaci procesů, a díky podpoře moderních frameworků, jako je SwiftUI, mohou snadno vytvářet intuitivní a vizuálně atraktivní aplikace. Důležitou vlastností Xcode je i jeho schopnost plynule přenášet aplikace mezi různými architekturami společnosti Apple, což vývojářům umožňuje snadno optimalizovat své projekty pro nejnovější zařízení.

Další výhodou Xcode je jeho dostupnost zdarma. Apple jej poskytuje bezplatně, což otevírá dveře vývoji aplikací i těm, kteří si nemohou dovolit investovat do drahých vývojových nástrojů. To motivuje širší komunitu vývojářů k tvorbě aplikací pro Apple ekosystém, čímž se rozšiřuje nabídka dostupných aplikací pro uživatele. Ačkoliv existují alternativní vývojová prostředí třetích stran, Xcode zůstává oficiálním a nejspolehlivějším řešením, které zajišťuje nejlepší kompatibilitu s Apple technologiemi a usnadňuje distribuci aplikací přes App Store.

Pro práci s Xcode je nutné vlastnit Mac s dostatečným výkonem, neboť tento nástroj není určen pro Windows. Přesto existují způsoby, jak jej na Windows provozovat, například přes virtuální stroje, ovšem tyto metody nejsou oficiálně podporovány a často přinášejí technické komplikace. Alternativně lze využít multiplatformní frameworky jako React Native nebo Xamarin, které umožňují vývoj iOS aplikací i na jiných operačních systémech. [8]

1.7 Program Apple Developer

Program Apple Developer je nezbytný pro každého, kdo chce distribuovat aplikace pro platformy společnosti Apple. Tento účet umožňuje přístup k nástrojům a službám potřebným pro vývoj, testování a publikaci aplikací na App Store. Registrace probíhá přes web Apple Developer a je rozdělena do několika kroků.

Nejdříve je nutné mít Apple ID s aktivovaným dvoufaktorovým ověřením. Po přihlášení do portálu Apple Developer lze zahájit registraci do programu, kde je třeba rozhodnout, zda se zaregistrovat jako individuální vývojář, nebo jako firma.

Individuální účet je vhodný pro jednotlivce a malé podnikatele, kteří chtějí publikovat aplikace pod svým jménem. Pokud však aplikace patří společnosti nebo organizaci, je nutné zvolit firemní registraci. Ta vyžaduje ověření právního subjektu pomocí čísla D-UN-S, které

přiděluje agentura Dun & Bradstreet. Tento krok slouží k ověření identity organizace a jejího právního postavení.

Po vyplnění údajů a přečtení licenční smlouvy následuje platba členského poplatku, který činí 99 USD ročně. Některé neziskové organizace, vzdělávací instituce nebo vládní subjekty mohou požádat o zproštění tohoto poplatku. Po úspěšné registraci a potvrzení od lze spravovat aplikace přes platformu App Store Connect, kde je možné nahrávat nové verze aplikací, spravovat metadata a monitorovat výkonnost aplikací na App Store.

Program Apple Developer poskytuje široké možnosti nejen pro vývoj aplikací, ale i pro testování a distribuci. Díky přístupu k nástrojům jako TestFlight lze snadno sdílet beta verze aplikací s testery. Integrace s technologií Xcode Cloud poté umožňuje automatizovat procesy vývoje. Vytvoření a správa účtu je tak klíčovým krokem pro každého, kdo chce vyvíjet a publikovat aplikace na App Store. [9] [10]

2 NÁVRHOVÝ VZOR MVVM

V prostředí SwiftUI je jedním z nejčastěji používaných architektonických vzorů tzv. MVVM (Model–View–ViewModel). Tento vzor je oblíbený především díky své přehlednosti a schopnosti oddělit jednotlivé vrstvy aplikace. Právě tato separace zodpovědností pomáhá vytvářet udržitelný, testovatelný a rozšiřitelný kód, což je u moderních mobilních aplikací zásadní. Na druhou stranu je třeba počítat s tím, že MVVM může být zbytečně složité pro velmi jednoduché aplikace a jeho principy vyžadují určitou míru pochopení složitějších konceptů jazyka Swift. Základem MVVM architektury je myšlenka, že jednotlivé části aplikace by se měly soustředit pouze na svou konkrétní roli.

2.1 Model

Vrstva modelu představuje základní stavební kámen každé aplikace. Obsahuje veškerá data, struktury a pravidla, která určují, jak se s daty zachází. V ideálním případě je model úplně nezávislý na uživatelském rozhraní – vůbec netuší, že nějaké UI (user interface) existuje. Může jít o jednoduché datové struktury, nebo složitější třídy, které zahrnují validace, výpočty nebo přístup k externím zdrojům (např. síťové volání, ukládání do databáze atd.). Účelem vrstvy je poskytovat spolehlivá a konzistentní data aplikaci bez ohledu na to, jak jsou zobrazena nebo jak s nimi uživatel interaguje.

2.2 View

View je část aplikace, kterou vidí a ovládá uživatel. Ve SwiftUI je View deklarativní, což znamená, že popisujeme, co má být na obrazovce, nikoliv jak se má jednotlivý prvek vykreslit nebo aktualizovat. View zobrazuje data, která mu předává tzv. View model – nikdy by nemělo obsahovat žádnou business logiku ani přímý přístup k modelu. Pokud uživatel provede nějakou akci (například stiskne tlačítko, nebo zadá text), View předá tuto událost zpět do View modelu, který ji zpracuje a v případě potřeby aktualizuje model nebo samotný View model. Ve SwiftUI se ke sledování změn používají tzv. „property wrappery” jako „@ObservedObject”, „@StateObject” nebo „@EnvironmentObject”, které umožňují efektivně reagovat na změny v datech a udržovat synchronizaci s View modelem.

2.3 View model

View model je srdcem MVVM architektury. Je to prostředník mezi modelem (daty) a View (uživatelským rozhraním). Na jedné straně komunikuje s modelem – načítá, upravuje a

aktualizuje data – a na druhé straně připravuje tato data do formy, kterou může View snadno zobrazit. Ve SwiftUI je View model typicky třída implementující rozhraní „ObservableObject“. Tím pádem může používat vlastnosti s anotací „@Published“, což umožňuje automaticky informovat View o změnách. Pokud dojde ke změně (např. nová položka v seznamu), View model aktualizuje odpovídající proměnnou, a View, které ji sleduje se automaticky překreslí.

View model má navíc na starost obsluhu uživatelských akcí – například stisk tlačítka, které spustí metodu pro přidání nové položky nebo aktualizaci stavu. To vše se děje bez přímého zásahu do modelu nebo View. Výhodou tohoto přístupu je, že logika zůstává oddělená od uživatelského rozhraní, což zvyšuje testovatelnost i udržitelnost kódu. [11] [12]

3 PRINCIPY SOLID

Principy SOLID představují pět základních pravidel objektově orientovaného návrhu, která byla formulována Robertem C. Martinem v roce 2000 v jeho práci „Design Principles and Design Patterns“. Později byl tento koncept rozšířen Michaelem Feathersem, který vytvořil akronym SOLID. Během posledních dvaceti let tyto principy zásadně ovlivnily způsob, jakým programátoři přistupují k návrhu softwaru, a přispěly k vývoji udržitelnějšího a lépe rozšiřitelného kódu.

Hlavním cílem principů SOLID je usnadnit vývoj softwaru tak, aby byl přehledný, snadno udržovatelný a flexibilní vůči budoucím změnám. Tím se minimalizuje složitost aplikací při jejich růstu a vývojáři se vyhýbají problémům, které by mohly vzniknout při špatně navržené architektuře. Dodržování těchto principů usnadňuje také refaktorování kódu a podporuje agilní vývoj softwaru.

3.1 Princip jedné odpovědnosti

Princip jediné odpovědnosti (Single Responsibility Principle – SRP) říká, že třída by měla mít pouze jednu odpovědnost, což znamená, že by měla mít pouze jeden důvod pro změnu. Tento princip pomáhá vytvářet kód, který je snadněji testovatelný, má nižší závislosti a je lépe organizovaný. Například, pokud existuje třída pro výpočet plochy různých tvarů, jako je čtverec a kruh, tato třída by neměla také obsahovat logiku pro výstup dat (např. ve formátu JSON), protože by tím porušila princip SRP. Místo toho by měla být vytvořena samostatná třída pro generování výstupu.

3.2 Princip otevřenosti a uzavřenosti

Princip otevřenosti a uzavřenosti (Open/Closed Principle – OCP) vyžaduje, aby třídy byly otevřeny pro rozšíření, ale uzavřeny pro modifikace. To znamená, že lze přidat novou funkcionalitu do systému bez změny již existujícího kódu. Takovým způsobem lze přidávat nové vlastnosti a funkce bez rizika porušení existujícího chování systému, což zajišťuje stabilitu aplikace během jejího vývoje.

3.3 Liskovův princip zaměnitelnosti

Podle Liskovového principu zaměnitelnosti (Liskov Substitution Principle – LSP) by měly být odvozené třídy nebo moduly plně zaměnitelné za své základní třídy bez ovlivnění chování programu. Když nahradíme třídu její podtřídou, aplikace by měla stále fungovat

správně. Tento princip zaručuje, že podtřídy nebudou mít chování, které by mohlo porušit očekávání původní třídy, což napomáhá udržitelnosti a rozšiřitelnosti kódu.

3.4 Princip oddělení rozhraní

Princip oddělení rozhraní (Interface Segregation Principle – ISP) tvrdí, že by klienti neměli být nuceni implementovat rozhraní, které nevyužívají. Místo jednoho velkého rozhraní, které by bylo odpovědné za mnoho různých funkcí, je lepší vytvořit několik menších a konkrétnějších rozhraní. Tento přístup zajišťuje, že třídy a moduly budou mít pouze metody, které skutečně potřebují, což usnadní údržbu a minimalizuje zbytečné závislosti.

3.5 Princip obrácení závislostí

Princip obrácení závislostí (Dependency Inversion Principle – DIP) vyžaduje, aby moduly závisely na abstrakcích (např. rozhraních), nikoliv na konkrétních implementacích. To znamená, že vysokoúrovňové třídy by neměly záviset na nízko úrovněvých implementacích, ale na abstraktních kontraktech, které mohou být implementovány různými způsoby. Tento přístup zajišťuje flexibilitu a umožňuje snadnou výměnu implementací (např. změna typu databáze), aniž by bylo nutné měnit samotnou aplikaci. [13] [14]

4 CLEAN CODE

Pojem clean code, tedy „čistý kód“, označuje styl psaní kódu, který je přehledný, srozumitelný, snadno čitelný a zároveň dobře udržitelný. Tento přístup není jen otázkou osobního stylu nebo estetiky, ale představuje jeden ze základních pilířů profesionálního vývoje softwaru. Cílem je psát kód nejen pro počítače, ale především pro lidi – pro kolegy v týmu i pro sebe sama v budoucnu. Jedním z nejvýznamnějších autorů, kteří tento přístup formulovali a rozšířili, je Robert C. Martin, známý také jako „Uncle Bob“. Ve své knize Clean Code: A Handbook of Agile Software Craftsmanship z roku 2008 popisuje principy, návyky a konkrétní doporučení pro psaní kvalitního kódu. Jedna z jeho klíčových myšlenek vystihuje podstatu celého konceptu:

*„Poměr času stráveného čtením a psaním kódu je více než 10:1.
Neustále čteme starý kód, abychom mohli napsat nový.“*

– Robert C. Martin [15]

Tento fakt vysvětluje, proč je důležité klást důraz na čitelnost a strukturu kódu. Čistý kód zvyšuje nejen kvalitu výsledného softwaru, ale i efektivitu vývojového týmu, zrychluje orientaci v projektu a usnadňuje údržbu a rozšiřování aplikace v budoucnu. V následujících podkapitolách se zaměříme na hlavní principy a doporučení, které pomáhají psát čistý a udržitelný kód.

4.1 Čitelnost a srozumitelnost

Základní vlastností čistého kódu je jeho čitelnost. Kód by měl být psán primárně pro lidi, kteří s ním budou pracovat – ať už se jedná o kolegy v týmu nebo o samotného autora po několika měsících. V praxi to znamená používat smysluplná pojmenování proměnných, funkcí a tříd, která jasně odráží jejich účel a použití. Namísto zkratk, nejasných zkrácenin nebo generických názvů se doporučuje volit popisné a jednoznačné názvy.

4.2 Funkce a jejich velikost

Funkce by měly být krátké a dělat jednu konkrétní věc. Tento princip vychází ze známého pravidla „jedna funkce – jeden účel“. Takto navržené funkce jsou snadno pochopitelné, opakovaně použitelné a dobře testovatelné. Pokud funkce začne plnit více rolí najednou, ztrácí se její přehlednost a roste riziko chyb. V takových případech je vhodné rozdělit

funkčnost do menších částí a každou z nich oddělit do samostatné metody nebo pomocné struktury.

4.3 Kód bez duplicit

Jedním z hlavních principů čistého kódu je tzv. „DRY” (Don’t Repeat Yourself). Každý kousek logiky by měl být v kódu reprezentován pouze jednou. Duplicitní části nejen zbytečně zvětšují objem kódu, ale ztěžují i jeho údržbu – změna na jednom místě vyžaduje identickou úpravu i jinde. Pokud se stejná logika vyskytuje na více místech, je vhodné ji vyčlenit do společné funkce, modulu nebo objektu.

4.4 Smysluplné komentáře a dokumentace

Komentáře by měly doplňovat informace, které nejsou zřejmé ze samotného kódu. Neměly by však nahrazovat špatně pojmenované proměnné nebo funkce. Ideální je, když kód hovoří sám za sebe a komentáře slouží pouze k vysvětlení záměru nebo upozornění na výjimky. Zastaralé nebo zbytečné komentáře mohou být spíše na škodu – matou čtenáře a vytvářejí falešný kontext.

4.5 Konzistence a standardy

Důležitým aspektem čistého kódu je dodržování konzistentního stylu – od pojmenovávání až po strukturu souborů a odsazování. Jazyk Swift má své idiomatické zápisy, které podporují stručnost, ale nikdy by neměly být upřednostňovány na úkor srozumitelnosti. Používání interních standardů nebo komunitních konvencí (např. Swift API Design Guidelines) přispívá k jednotnému stylu napříč celým projektem a zvyšuje přehlednost.

4.6 Refaktoring a udržování kódu

Psaní čistého kódu není jednorázová činnost, ale dlouhodobý proces. Refaktoring – tedy úprava struktury kódu bez změny jeho chování – by měl být přirozenou součástí vývojového cyklu. Díky průběžnému zjednodušování a zlepšování existujícího kódu lze předejít technickým chybám, zamezit degradaci kvality projektu a připravit kód na budoucí rozšiřování.

4.7 Testovatelnost a stabilita

Dobře strukturovaný a čistý kód bývá zpravidla i snadněji testovatelný. Oddělené funkce, minimální závislosti a jednoznačná logika umožňují psát cílené jednotkové (unit) testy, které ověřují funkčnost aplikace. Testování je tak nedílnou součástí procesu psaní čistého kódu – zajišťuje, že budoucí změny nenaruší existující chování. [16]

5 AGILNÍ VÝVOJ

Agilní metodologie představuje přístup k vývoji softwaru, který klade důraz na flexibilitu, pragmatičnost a postupné dodávání funkčních částí aplikace. Tento přístup často vyžaduje kulturní změnu ve firmách, jelikož přenáší pozornost z kompletního produktu na jednotlivé dílčí části a jejich kvalitu.

Výhody agilního vývoje spočívají především v lepší schopnosti přizpůsobit se měnícím se podmínkám, zvýšení efektivity týmové spolupráce a kontinuálním testování v průběhu celého vývoje. Tím je možné dříve identifikovat chyby a provádět nezbytné úpravy. Agilní přístup tak napomáhá doručit kvalitní software s vysokou přidanou hodnotou pro byznys.

5.1 Základní hodnoty

Agilní Manifest vznikl v roce 2001 jako výsledek setkání 17 softwarových vývojářů. Vymezuje čtyři základní hodnoty:

- Jednotlivci a jejich interakce před procesy a nástroji – vývoj řídí lidé, nikoliv nástroje či metodiky.
- Fungující software před rozsáhlou dokumentací – dokumentace není opomíjena, ale její rozsah je omezen na nezbytné minimum.
- Spolupráce se zákazníkem před smluvním vyjednáváním – zákazník se aktivně účastní vývoje a poskytuje průběžnou zpětnou vazbu.
- Reakce na změnu před dodržováním plánu – změny nejsou vnímány jako hrozba, ale jako příležitost k vylepšení výsledku.

Manifest je dále rozvinut dvanácti principy, mezi které patří například důraz na včasné a kontinuální dodávání hodnoty, podpora týmů, přijetí změn i v pozdních fázích vývoje, měření pokroku pomocí fungujícího softwaru nebo neustálé zlepšování.

5.2 Vývojový cyklus

Agilní vývojový cyklus představuje opakující se, iterativní přístup k vývoji softwaru, kde je cílem postupně doručovat funkční části aplikace s důrazem na neustálou zpětnou vazbu a možnost rychle reagovat na změny. Skládá se z následujících fází:

- Koncept – prvním krokem je identifikace obchodní příležitosti a zhodnocení, zda má smysl projekt realizovat. Zvažuje se technická proveditelnost, ekonomická

náročnost, přínos pro uživatele a odhad potřebného času a zdrojů. Výstupem je prioritizace projektů na základě jejich hodnoty a proveditelnosti.

- Zahájení – v této fázi se sestavuje vývojový tým, přidělují se role (např. Scrum Master, Product owner), zajišťuje se financování a dochází k prvním konzultacím se zákazníkem. Definují se základní požadavky a vzniklý časový harmonogram. Výsledkem je jasně daný plán, co a kdy má být dodáno.
- Konstrukce a iterace – klíčová fáze, ve které vzniká funkční software. Vývoj probíhá ve sprintech – krátkých časových úsecích, obvykle 2–4 týdny. Na začátku sprintu tým stanoví cíle a priority. Na konci každé iterace by měl vzniknout fungující a testovaný produktový přírůstek. Každý sprint obvykle zahrnuje analýzu a definování požadavků, vývoj funkcí, testování a získání zpětné vazby od stakeholderů. Po každé iteraci dochází k revizi, retrospektivě a plánování dalšího sprintu.
- Vydání – po dokončení několika iterací následuje fáze finálního testování, odstraňování zbývajících chyb a finalizace dokumentace. Je připravena produkční verze softwaru, která je následně nasazena k reálnému použití. Důležitou roli zde hraje kvalitní komunikace s koncovými uživateli.
- Provoz – po nasazení aplikace je potřeba zajišťovat její hladký chod. To zahrnuje monitoring, řešení incidentů, poskytování technické podpory a zaškolení uživatelů.
- Ukončení – v poslední fázi dochází k ukončení podpory dané verze nebo celého systému. Důvodem může být technologické zastarání, změna obchodní strategie nebo přechod na novější řešení. Zahrnuje oznámení uživatelům, archivaci dat a případnou migraci do nového systému.

5.3 Metodiky

Agilní vývoj zahrnuje několik osvědčených přístupů, které sdílejí společné principy (např. iterativní vývoj, důraz na spolupráci a adaptaci), ale liší se způsobem organizace práce a důrazy:

- Scrum – nejrozšířenější agilní framework, založený na práci v krátkých časových úsecích (sprintech), s důrazem na týmové role (Scrum Master, Product owner) a pravidelné ceremonie (stand-upy, retrospektivy, plánování sprintu).

- Lean – zaměřuje se na efektivitu, minimalizaci plýtvání a doručování hodnoty zákazníkovi v co nejmenších dávkách. Podporuje týmovou autonomii a průběžné zlepšování.
- Extreme Programming (XP) – klade důraz na technickou kvalitu, časté vydávání verzí, úzkou spolupráci se zákazníkem a praktiky jako párové programování nebo testování před psaním kódu (TDD).
- Crystal – sada lehkých metodik, které se přizpůsobují konkrétnímu kontextu projektu (velikost týmu, kritičnost systému). Klíčová je komunikace a minimální byrokracie.
- DSDM (Dynamic Systems Development Method) – Zdůrazňuje byznys potřeby, rychlé dodávky a kontrolu kvality. Pracuje s prioritizací požadavků (např. metodou MoSCoW).
- FDD (Feature Driven Development) – Zaměřený na vývoj konkrétních funkcí podle předem připraveného doménového modelu. Vhodný pro větší týmy a rozsáhlejší systémy. [17]

6 VERZOVÁNÍ A GIT

Verzování (version control) je klíčová softwarová praktika, která umožňuje sledovat a spravovat změny v kódu a dalších souborech během vývoje aplikace. Každá změna je zaznamenána, což týmu umožňuje vracet se ke starším verzím, sledovat historii úprav a jednoduše řešit konflikty mezi různými vývojáři. Verzování tak funguje jako pojistka proti ztrátě nebo poškození kódu a podporuje bezpečné experimentování.

Nejrozšířenějším systémem pro verzování je Git – distribuovaný systém, který umožňuje každému vývojáři pracovat s plnou kopií celého repozitáře včetně historie. To podporuje offline práci, flexibilní větvení (branching) a slučování změn (merging). Git díky své rychlosti, spolehlivosti a otevřenosti představuje standard ve vývoji softwaru všech velikostí – od malých projektů až po rozsáhlé enterprise systémy. [18]

6.1 Základní funkce systému Git

Git je rozšířený nástroj pro správu verzí, který vývojářům umožňuje sledovat změny v kódu, efektivně spolupracovat v týmu a bezpečně experimentovat bez rizika narušení hlavní větve projektu. Práce se systémem Git obvykle začíná inicializací repozitáře pomocí příkazu „git init“, čímž vznikne prostředí, kde lze sledovat změny v souborech. Když vývojář vytvoří nebo upraví soubor, systém si této změny všimne, ale nezačne ji sledovat, dokud není přidána do tzv. „staging area“ příkazem „git add“. Následně je možné vytvořit tzv. „commit“ pomocí příkazu „git commit“, což je konkrétní záznam změn s komentářem, který se stává součástí historie projektu.

Pro oddělený vývoj nové funkce či opravy se běžně používají tzv. „větvě“ (branches) – nové větve se vytvářejí pomocí „git checkout“, čímž se otevře nový směr vývoje, který je nezávislý na hlavní větvi (s pojmenováním např. main nebo master). Po dokončení práce na větvi lze změny odeslat na vzdálený server (například GitHub) příkazem „git push“ čímž se ostatní členové týmu dostanou ke stejnému kódu. Změny se pak typicky navrhuje ke sloučení prostřednictvím tzv. „pull requestu“ (PR), což je žádost o revizi a schválení kódu před jeho začleněním zpět do hlavní větve. Jakmile je PR schválen, dojde ke sloučení větví příkazem „git merge“ a výsledný kód je dostupný v hlavní větvi projektu.

Aby měl vývojář vždy aktuální verzi kódu, může kdykoliv stáhnout nové změny pomocí „git pull“. Historii změn v repozitáři si lze prohlédnout přes „git log“, kde jsou přehledně zobrazeny všechny změny s informací o autorovi, datu a popisu změny. Díky

těmto nástrojům Git zajišťuje nejen technické zázemí pro týmovou spolupráci, ale také přehlednou a dohledatelnou historii celého vývoje. [19]

6.2 GitHub

GitHub je platforma postavená na technologii Git, která vývojářům umožňuje sdílet své repozitáře online, spolupracovat na kódu a využívat nástroje pro řízení vývoje. Nabízí uživatelsky přívětivé webové rozhraní, ale i pokročilé nástroje. GitHub umožňuje transparentní spolupráci napříč týmy, podporuje open-source komunitu a je také využíván jako online portfolio pro ukázkou vývojářské práce. I když GitHub není k použití systému Git nutný, stal se nejrozšířenější platformou svého druhu a neodmyslitelnou součástí moderního vývoje aplikací. Vedle platformy GitHub existují i další alternativy, jako GitLab, Bitbucket nebo Azure DevOps, které rovněž staví na systému Git. [20]

6.3 Strategie větvení

Strategie větvení v systému Git slouží k efektivní organizaci práce v týmu a řízení vývoje softwaru. Jednou z tradičních a dříve hojně využívaných metod je tzv. „Gitflow“, což je strukturovaný model navržený pro projekty s plánovanými vydáními.

V Gitflow se používají dvě hlavní větve: „main“, která uchovává stabilní, produkční verze aplikace, a „develop“, která slouží jako integrační větev pro probíhající vývoj. Veškeré nové funkce se vyvíjejí v oddělených „feature“ větvích, které jsou vytvářeny z větve develop a po dokončení do něj opět sloučeny. Feature větve by nikdy neměly být slučovány přímo do hlavní větve main.

Když je připraveno vydání, vzniká z větve develop větev s označením „release“, která slouží k finalizaci verze – opravám chyb, dokumentaci nebo testování. Jakmile je release připraven, je sloučen do větve main a zároveň zpětně i do develop, aby se případné opravy dostaly i do budoucího vývoje. Pro urgentní opravy produkce se v Gitflow používají tzv. „hotfix“ větve, které jako jediné vznikají přímo z větve main. Tato strategie tak umožňuje paralelně pracovat na nových funkcích i opravách, aniž by došlo k narušení vydání.

Gitflow definuje i pojmenování větví, čímž přispívá k přehlednosti projektu. Celkově nabízí jasně strukturovaný přístup k řízení verzí, ale v prostředí častého nasazování může být příliš těžkopádný. Moderní týmy dnes častěji volí jednodušší přístupy jako trunk-based development, kdy se většina vývoje odehrává na jedné hlavní větvi s krátkodobými feature větvemi. Výběr strategie záleží na velikosti týmu, typu projektu a míře automatizace –

důležité je, aby byla konzistentní a podporovala plynulý vývoj bez zbytečných konfliktů. [21]

6.4 Slučování větví

Při vývoji v systému Git se dříve či později nastane situace, kdy je potřeba sloučit změny z jedné větve do druhé. Git a GitHub nabízejí několik metod, jak tento „merge” provést, přičemž každá má své výhody a specifika. Výběr správné strategie ovlivňuje přehlednost historie změn, plynulost spolupráce i způsob řešení konfliktů.

Nejběžnější metodou je klasické slučování pomocí tzv. „merge commitu”. Tato varianta zachovává veškeré commity z vývojové větve a zároveň přidává nový commit, který reprezentuje samotné sloučení. Výsledkem je detailní historie vývoje, která ukazuje, jak se změny postupně vyvíjely. Tento přístup je vhodný, pokud je cílem zachovat úplnou a transparentní historii vývoje, včetně jednotlivých kroků a iterací.

Alternativou je tzv. „squash merge”, který při sloučení zkomprimuje všechny změny z feature větve do jediného commitu. Tím se výsledná historie výrazně zjednoduší a zpřehlední, což je výhodné zejména u větví s mnoha drobnými nebo průběžnými změnami. Tento přístup se často využívá tam, kde je důležitá čistota hlavní větve a kde samotný vývojový proces není nutné explicitně archivovat. Je však třeba počítat s tím, že tento přístup maže informaci o jednotlivých autorech i průběhu vývoje.

Třetí možností je „rebase and merge”, kdy se všechny jednotlivé commity z vývojové větve přenesou do hlavní větve jako samostatné záznamy, ale bez přidání merge commitu. Výsledkem je lineární historie, která působí čistě a přehledně, protože eliminuje rozvětvení. Tento přístup ale přepisuje identitu commitů, což může komplikovat práci s změnami nebo historií.

Každá z těchto metod má své výhody i nevýhody a neexistuje univerzálně nejlepší volba. Důležité je, aby tým měl sjednocený přístup a rozuměl tomu, jak zvolená strategie ovlivní historii a správu kódu v projektu. [22]

7 TESTOVÁNÍ

Testování aplikace je nezbytnou součástí vývoje. Jeho cílem je ověřit, že software odpovídá požadavkům, funguje podle očekávání a je dostatečně spolehlivý pro produkční nasazení. Testování pomáhá identifikovat chyby v rané fázi, snižovat rizika a zajišťovat vysokou kvalitu výsledného produktu. V praxi tak software prochází sérií testů, které mohou být manuální i automatizované, a zahrnují různé úrovně, typy a přístupy.

V praktické části se zaměříme především na jednotkové (unit) a testy uživatelského rozhraní (UI), které představují klíčové prvky funkčního testování. Dále se budeme věnovat tzv. „black-box” testování, které ověřuje funkčnost systému z pohledu koncového uživatele bez znalosti vnitřní implementace. Součástí bude také využití nástrojů jako je TestFlight – oficiální platforma od společnosti Apple pro distribuci testovacích verzí aplikací, která usnadňuje testování v reálném prostředí na zařízeních uživatelů.

7.1 Funkcionální a nefunkcionální testování

Testování se obecně dělí na funkcionální a nefunkcionální. Funkcionální testování ověřuje, zda software plní to, co má – tedy správnou implementaci požadavků. Sem patří např. unit testy, integrační testy nebo testy přijetí uživatelem (UAT). Oproti tomu nefunkcionální testování se zaměřuje na vlastnosti jako výkonnost, bezpečnost, použitelnost nebo kompatibilitu. Je důležité si uvědomit, že kvalitní software není jen o funkcionalitě, ale i o tom, jak se chová za různých podmínek a jak dobře zvládá zátěž, škálování nebo útoky.

7.2 White-box a black-box testování

Z pohledu testovacího přístupu rozlišujeme tzv. „black-box” a „white-box” testování. U black-box testování se testuje bez znalosti vnitřní struktury kódu – testující posuzuje pouze vstupy a výstupy, tedy jak se aplikace chová z pohledu uživatele. Naopak white-box testování využívá znalost vnitřní implementace a zaměřuje se na pokrytí kódu, větví, cyklů apod. Kombinací těchto přístupů vzniká také „grey-box” testování, které částečně využívá interní znalost systému a umožňuje efektivně testovat i bez úplného přístupu ke kódu.

7.3 Hlavní úrovně testování

Testování softwaru lze členit také podle jednotlivých úrovní, které odrážejí fáze, v nichž se software nachází, a míru jeho složitosti. Na nejnižší úrovni se nachází jednotkové testování (unit testing), které se zaměřuje na testování jednotlivých komponent nebo funkcí v izolaci.

Cílem je ověřit, že každý samostatný prvek systému funguje správně. Tento typ testování je často automatizovaný a bývá součástí práce samotných vývojářů.

Další úroveň představuje integrační testování (integration testing), které ověřuje, zda jednotlivé části systému mezi sebou správně komunikují. Testují se zde například rozhraní mezi moduly, datové toky nebo komunikace přes API (rozhraní pro komunikaci mezi různými softwarovými částmi nebo systémy, např. aplikací a serverem).

Následuje systémové testování (system testing), které posuzuje aplikaci jako celek. Cílem je zjistit, zda všechny části společně fungují v souladu s očekáváním. Tento typ testování často zahrnuje i prvky nefunkčního testování, jako je výkonnost nebo bezpečnost.

Nejvyšší úroveň je testování přijetí (acceptance testing), které ověřuje, zda software splňuje obchodní cíle a očekávání zadavatele. Tento typ testu provádějí většinou koncoví uživatelé, klienti nebo stakeholderi a jeho výsledkem je rozhodnutí, zda je produkt připraven pro nasazení do produkce. [23] [24]

7.4 Framework XCTest

XCTest je oficiální testovací framework od společnosti Apple, který je určený pro psaní a spouštění testů v rámci Xcode projektů. Umožňuje vývojářům ověřit, že jejich kód funguje správně, že jednotlivé komponenty spolu správně spolupracují a že výkon aplikace zůstává v přijatelných mezích. Jeho integrace s Xcode umožňuje snadnou práci s aplikací v rámci vývoje, včetně vizualizace výsledků testů a automatického spouštění testů při každé změně kódu. Pomocí XCTest lze vytvářet různé druhy testů – od jednotkových testů (unit tests) přes UI testy, až po výkonnostní testy (performance tests).

Jednotkové testy slouží k ověřování chování jednotlivých funkcí, metod nebo tříd. Testují logiku v izolaci od zbytku aplikace, což zajišťuje rychlé provádění a snadnou detekci chyb. Tyto testy jsou typicky psány přímo vývojáři a běží automatizovaně.

Framework XCTest umožňuje také testování uživatelského rozhraní pomocí nástroje „XCUIAutomation“. Vývojáři mohou simulovat uživatelské interakce – například stisknutí tlačítka nebo psaní do textových polí – a ověřit, že aplikace reaguje tak, jak má. To je klíčové pro ověření správného fungování uživatelského toku a interakce v reálných scénářích.

Další důležitou oblastí je měření výkonu pomocí tzv. „performance tests“. Tyto testy sledují, jak dlouho trvá vykonání určitého kusu kódu, a umožňují porovnávat výsledky s

předchozími verzemi. Pokud dojde k výraznému zpomalení, test selže. Pomáhá to odhalit regresi výkonu v čase.

Základním nástrojem každého testu ve frameworku XCTest jsou tzv. „asserty” – výrazy, které ověřují, že nějaká podmínka platí. Pokud assert selže, test je označen jako neúspěšný. [25]

7.5 TestFlight

TestFlight je oficiální nástroj od společnosti Apple, který usnadňuje testování aplikací před jejich zveřejněním v App Store. Umožňuje vývojářům sdílet testovací verze svých aplikací s interními i externími testery, získávat zpětnou vazbu a odhalit případné chyby dříve, než se aplikace dostane ke koncovým uživatelům.

V jistém smyslu se dá říci, že TestFlight slouží jako nástroj pro testování přijetí (acceptance testing) — ověřuje se, jak aplikace funguje v reálných podmínkách, na různých zařízeních a v rukou skutečných uživatelů. Testeři mohou nahlásit chyby, posílat zpětnou vazbu, sdílet snímky obrazovky a přispívat tak k finálním úpravám před vydáním.

Vývojáři mohou přes TestFlight distribuovat různé „buildy” (konkrétní verze aplikace vytvořená pro testování před finálním vydáním), testovat současně více verzí a rozdělit testery do skupin podle potřeby – například podle typu zařízení nebo konkrétní funkce, kterou mají ověřit. K dispozici je také veřejný odkaz, přes který se může k testování přihlásit široká veřejnost, nebo lze zvát testery cíleně e-mailem.

Zpětná vazba je sbírána přímo v platformě App Store Connect, kde lze sledovat statistiky, jako počty instalací nebo crash reporty. Díky tomu se TestFlight stává důležitým nástrojem pro ladění kvality, uživatelského zážitku (UX) i funkčnosti aplikace v reálném prostředí — tedy přesně v duchu kvalitního akceptačního testování. [26]

8 DOKUMENTACE

Dokumentace kódu je nedílnou součástí profesionálního vývoje softwaru, která přispívá nejen ke snadnějšímu porozumění implementace, ale také ke zlepšení spolupráce v týmu a k dlouhodobé udržitelnosti projektu. I když se často klade důraz na psaní tzv. „selfdocumenting code” – tedy takového kódu, který je natolik srozumitelný, že nevyžaduje doplňující komentáře – ve větších projektech a složitějších systémech tato snaha sama o sobě nestačí.

Dokumentace slouží jako most mezi tím, co je v kódu napsané, a tím, co měl vývojář skutečně na mysli. Umožňuje zachytit kontext rozhodnutí, popsat architekturu systému, vysvětlit specifické chování nebo upozornit na výjimky a okrajové případy. Je také neocenitelným nástrojem při zaškolování nových členů týmu, kteří se díky dobře vedené dokumentaci mohou rychleji zorientovat ve struktuře projektu, bez nutnosti pročítat tisíce řádků kódu a odhadovat záměry původních autorů.

8.1 Základní rozdělení

Dokumentaci lze rozdělit do několika úrovní podle cílové skupiny a účelu. Interní dokumentace je určena pro vývojáře uvnitř organizace a zahrnuje jak komentáře přímo ve zdrojovém kódu (inline dokumentace), tak například designové dokumenty nebo popisy vývojových procesů. Naproti tomu externí dokumentace je určena pro vývojáře mimo organizaci – typickým příkladem je veřejně dostupná dokumentace k API, knihovnám nebo SDK (sada nástrojů a knihoven pro vývoj aplikací pro konkrétní platformu).

Zajímavou kategorií je tzv. „high-level” dokumentace, která se zaměřuje na popis architektury systému, rozhodovacích procesů a celkového kontextu. Naopak „low-level” dokumentace se věnuje detailům jednotlivých funkcí, tříd nebo proměnných. Obě úrovně však mají své limity – ta první často postrádá konkrétnost, zatímco ta druhá nenabízí širší kontext. Právě proto se v poslední době stále více prosazuje koncept tzv. „walkthrough” dokumentace, která propojuje obě perspektivy a krok za krokem provádí čtenáře klíčovými částmi kódu v jejich vzájemných souvislostech.

Jednou z největších výzev v oblasti dokumentace je její udržitelnost v čase. Jak se mění samotný kód, dokumentace často zůstává v původním stavu, což může vést ke zmatení nebo dokonce k chybám. Klíčem je tedy psát dokumentaci způsobem, který umožňuje její

jednoduchou aktualizaci – ať už pomocí verzovacích systémů, nebo nástrojů, které detekují změny v kódu a upozorní na možný nesoulad.

Další výzvou bývá rozdílná úroveň znalostí mezi čtenáři dokumentace. To, co je samozřejmé pro zkušeného vývojáře, může být pro juniora naprosto nejasné. Vhodným řešením bývá členění dokumentace podle úrovně zkušeností – například pomocí sekcí, poznámek pod čarou nebo odkazy na doplňující vysvětlení.

A v neposlední řadě – dobře napsaná dokumentace šetří čas. Nejen týmový čas, ale i ten vlastní. I když se může zdát, že dokumentace zdržuje, z dlouhodobého hlediska zrychluje vývoj, usnadňuje debugování (proces hledání a odstraňování chyb v programu) a přispívá k vyšší kvalitě kódu. Vývojáři se tak mohou lépe soustředit na psaní nových funkcí. [27]

8.2 Tvorba dokumentace v Xcode

Pro dokumentaci kódu v jazyce Swift je k dispozici nástroj přímo od společnosti Apple se zkratkou DocC (Documentation Compiler), který umožňuje vývojářům vytvářet profesionální a bohatě strukturovanou dokumentaci přímo ze zdrojového kódu. DocC je plně integrovaný do Xcode a dokáže generovat přehledné referenční stránky, výukové tutoriály i podrobné články o architektuře a použití knihoven či balíčků. Výhodou je, že dokumentace vzniká přímo při práci na kódu a je přirozenou součástí vývojového procesu, včetně podpory nástrojů jako „Quick Help” nebo automatického doplňování kódu.

DocC využívá vlastní variantu značkovacího jazyka „Markdown”, která je přizpůsobena specifickým potřebám vývojářské dokumentace. Umožňuje například vkládat odkazy na symboly v kódu, vytvářet seznamy, zvýrazňovat ukázky kódu, přidávat kontextové poznámky nebo strukturovat obsah pomocí direktiv. Díky tomu lze dokumentaci snadno organizovat a rozšiřovat – od jednoduchých popisků funkcí až po interaktivní návody, které vývojářům pomáhají pochopit, jak API používat v praxi.

Jakmile je dokumentace připravená, lze ji vygenerovat pomocí nabídky „Build Documentation” přímo v Xcode. Výsledkem je tzv. „DocC Archive”, který lze zobrazit v náhledu v Xcode, publikovat pro tým nebo nasadit jako webovou dokumentaci. [28] [29]

9 POŽADAVKY PLATFORMY APP STORE A LEGISLATIVA

Bezpečnost a dodržování legislativních pravidel patří mezi klíčové oblasti, které musí každý vývojář zohlednit při tvorbě mobilní aplikace. Nejde přitom pouze o ochranu uživatelských dat, ale i o respektování pravidel a doporučení stanovených platformou, na které má být aplikace distribuována. Tyto požadavky často definují nejen to, jak má být aplikace navržena z technického hlediska, ale i jak se má chovat z pohledu etiky, transparentnosti a práv uživatelů.

Platformy jako App Store nebo Google Play stanovují poměrně přísná pravidla pro zveřejnění aplikace. Vývojáři musí zajistit, že aplikace nebude porušovat žádná bezpečnostní omezení, bude transparentní ve zpracování dat a bude respektovat práva koncových uživatelů. To zahrnuje např. nutnost poskytovat informace o typech sbíraných dat nebo získávat výslovný souhlas s jejich využitím.

9.1 Pokyny pro kontrolu aplikací

Pokyny pro kontrolu aplikací (App Review Guidelines) představují soubor pravidel, kterými se musí řídit všechny aplikace určené pro distribuci přes App Store. Tato pravidla určují, jak má aplikace fungovat, jaké informace může sbírat, jak má vypadat uživatelské rozhraní a také jaké business modely jsou povoleny. Cílem je zajistit bezpečnost, soukromí, konzistenci a kvalitu napříč celou platformou. Před schválením každé nové verze aplikace prochází tzv. „App Review” procesem, tedy ruční kontrolou, při které se hodnotí soulad s pravidly.

App Review Guidelines jsou poměrně rozsáhlé a rozdělené do několika tematických oblastí – bezpečnost, výkon, business modely, design a právní soulad. Pro vývojáře je zásadní nejen tato pravidla znát, ale i je průběžně sledovat, protože se pravidelně aktualizují v reakci na vývoj technologií i společenské potřeby. Pro představu, jak mohou být některé z pravidel formulovány a jaký mají dopad na vývoj, je užitečné uvést několik konkrétních příkladů:

Pravidlo 1.1 zakazuje aplikace s obsahem, který by mohl být považován za urážlivý, znepokojivý nebo nevhodný – např. násilí, diskriminace nebo pornografie. Cílem je ochránit uživatele před nevhodným obsahem a udržet App Store jako bezpečné prostředí.

Pravidlo 2.1 říká, že aplikace musí být plně funkční a připravená pro reálné použití. Není povoleno nahrávat nedokončené prototypy nebo „placeholder” obsah (dočasně vložený

nehotový nebo zástupný obsah). Vývojář musí během procesu schvalování zajistit přístup k plné funkcionalitě (např. přes demo účet).

Pravidlo 4.2.2 výslovně uvádí, že aplikace nesmí sloužit primárně jako marketingový materiál, katalog, sbírka odkazů nebo jednoduchý wrapper (obalová aplikace, která pouze zobrazuje obsah webu bez vlastní funkčnosti či přidané hodnoty). Aplikace musí mít smysluplnou a interaktivní funkcionalitu, která přináší přidanou hodnotu. Toto pravidlo často způsobuje odmítnutí jednoduše převedených firemních webů do aplikace bez přidané hodnoty.

Dodržení těchto pravidel je nutnou podmínkou pro publikaci aplikace. Je proto důležité již při návrhu aplikace tyto požadavky zohlednit a vyhnout se tak zbytečnému zdržení či zamítnutí během procesu schvalování. [30]

9.2 Osobní údaje uživatelů

V rámci procesu publikování aplikace na App Store je dnes nedílnou součástí také deklarace způsobu nakládání s osobními údaji uživatelů. Apple klade na ochranu soukromí velký důraz a snaží se, aby uživatelé ještě před stažením aplikace věděli, jaká data o nich budou sbírána, k čemu budou využita a zda jsou spojena s jejich identitou. Tyto informace jsou prezentovány přímo na produktové stránce aplikace formou tzv. „Privacy Nutrition Labels” – jednoduchých a srozumitelných „štítků“ s přehledem sbíraných dat a jejich využitím.

Vývojáři mají povinnost poskytnout tyto informace nejen za svou aplikaci, ale také za všechny třetí strany (např. SDK, analytické a reklamní nástroje), jejichž kód aplikace obsahuje. Sbíraná data se musí kategorizovat podle typu (např. lokalizační údaje, kontaktní informace, obsah od uživatele, diagnostika atd.), podle způsobu využití (např. personalizace obsahu, analytika, reklamní účely, základní funkčnost aplikace) a podle toho, zda jsou tato data spojena s konkrétním uživatelem nebo zařízením.

Apple zároveň upozorňuje, že i data, která nejsou určena k marketingovým účelům nebo jsou sbírána pouze zřídka, mohou stále podléhat povinnosti zveřejnění – výjimky existují, ale jsou velmi úzce definovány. V poslední době byla pravidla rozšířena například o povinnost zveřejnit informace i o třetích stranách pomocí tzv. „privacy manifests” a nově jsou také podporovány podpisy SDK, které zvyšují transparentnost softwarového řetězce.

Celý tento systém má za cíl posílit důvěru uživatelů a zároveň přimět vývojáře k větší odpovědnosti při práci s osobními údaji. [31]

9.3 Obecné nařízení o ochraně osobních údajů

V oblasti ochrany osobních údajů hraje GDPR (Obecné nařízení o ochraně osobních údajů) zásadní roli i při vývoji mobilních aplikací. Toto nařízení se vztahuje na všechny vývojáře, kteří zpracovávají osobní data uživatelů v EU – bez ohledu na to, zda sami sídlí v Evropě. Pokud je aplikace nabízena uživatelům z EU, musí být s GDPR v souladu.

Za osobní údaje se přitom považuje široká škála informací – od jména nebo e-mailu až po IP adresu, historii vyhledávání nebo geolokaci. Z pohledu GDPR je pak „zpracováním“ každá operace s těmito daty: sběr, uchování, sdílení, analýza atd. Klíčovým pravidlem je, že pro každý akt zpracování musí mít vývojář tzv. právní základ, např. souhlas uživatele, plnění smlouvy nebo oprávněný zájem.

Souhlas musí být vždy informovaný, dobrovolný, specifický a odvolatelný. Nestací tedy pouze obecná formulace nebo „opt-out“. V praxi se tento princip odráží například při zobrazování cílené reklamy, přístupu k citlivým funkcím zařízení (kamera, mikrofon), nebo při integraci analytických nástrojů. Vývojář musí uživateli předem jasně sdělit, jaká data budou sbírána, k čemu budou použita a jak je možné jejich zpracování odmítnout.

S tím souvisí i již zmíněné Privacy Nutrition Labels, které Apple vyžaduje při publikování aplikací na App Store. Pokud uživatel svá data poskytne, má podle GDPR právo být informován, získat přístup ke svým údajům, opravit je, požadovat jejich výmaz, omezit zpracování nebo data přenést jinam. [32]

II PRAKTICKÁ ČÁST

10 ÚVOD K APLIKACI VOTAP

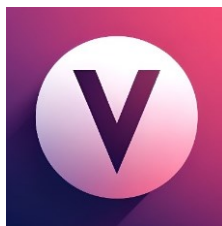
Pro praktickou demonstraci principů a metod popsaných v teoretické části jsem zvolil svou vlastní iOS aplikaci Votap, která používá SwiftUI a Firebase a je dostupná na platformě App Store. Votap umožňuje uživatelům hlasovat o politických osobnostech v reálném čase a sledovat aktuální trendy a vývoj veřejného mínění. Aplikace cílí na jednoduché a intuitivní uživatelské rozhraní, rychlou zpětnou vazbu a moderní přístup ke zpracování dat.

Tato část se zaměří na to, jak byly teoretické koncepty aplikovány při vývoji této aplikace v praxi. Bude ukázána implementace architektonického vzoru MVVM, přístup k psaní čistého, modulárního a udržitelného kódu podle zásad Clean code a principů SOLID. Tato část popisuje také strukturu projektu, organizaci souborů, způsob verzování pomocí systému Git, přístup k testování včetně asynchronních operací, práci s dependency injection, a v neposlední řadě i využití nástroje DocC pro dokumentaci kódu.

Součástí bude také návrh, jak by bylo možné vývoj této aplikace řídit v rámci agilního týmu, a to včetně doporučení ohledně sprintů, rozdělení rolí, nástrojů pro kolaboraci a vhodného workflow. Tím se praktická část pokusí propojit vývojové praktiky s reálným týmovým prostředím a procesy.

10.1 Cíl a zaměření aplikace

Cílem aplikace Votap je vytvořit centralizovanou platformu pro politické dění, která propojuje občany a politiky v reálném čase. Uživatelé mohou prostřednictvím jednoduchého a přehledného rozhraní hlasovat o aktuálních tématech, vyjadřovat podporu nebo nesouhlas s kroky konkrétních politiků a sledovat trendy veřejného mínění bez zpoždění. Aplikace má sloužit jako rychlý a přímý kanál zpětné vazby — politici tak získávají okamžitý přehled o náladě společnosti, což může vést k větší odpovědnosti. Smyslem je eliminovat pasivní čekání na další volební období a podpořit průběžný dialog mezi veřejností a zastupiteli. Votap představuje krok směrem k modernější a transparentnější demokracii, kde technologie zjednodušují a zefektivňují komunikaci i rozhodovací procesy.

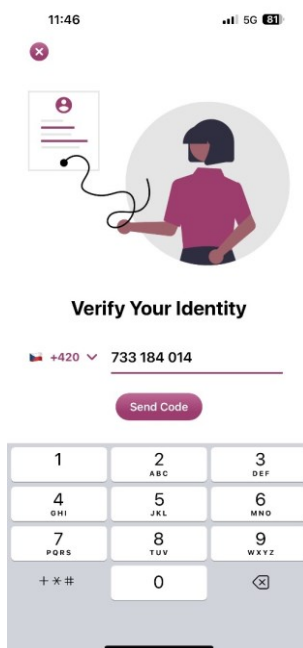


Obrázek 1 Logo aplikace Votap

(zdroj: vlastní)

10.2 Funkcionalita a klíčové vlastnosti

Aplikace Votap je navržena tak, aby umožnila občanům aktivně se zapojit do politického dění prostřednictvím jednoduchého, rychlého a zároveň bezpečného hlasování. Každý uživatel si vytvoří anonymní účet pomocí ověření telefonního čísla, na které je odeslán jednorázový ověřovací kód. Při registraci zadá také své PSČ, což umožňuje určit jeho přibližnou polohu a regionální příslušnost bez nutnosti zadávat konkrétní adresu – důraz je kladen na zachování soukromí.

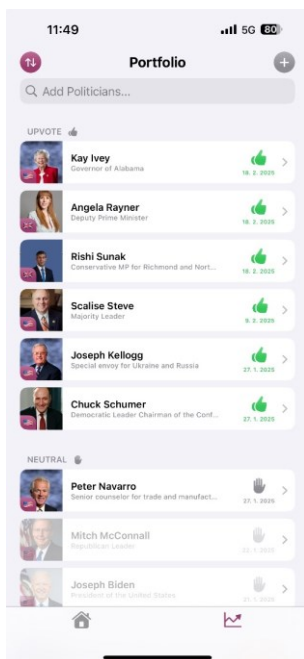


Obrázek 2 Přihlašování v aplikaci Votap

(zdroj: vlastní)

Po přihlášení může uživatel vyhledávat politiky, kteří jsou aktivní ve veřejném prostoru, a přidat si je do vlastního portfolia. Hlasování probíhá velmi jednoduše – uživatel si může zvolit, zda konkrétního politika podporuje (upvote), odmítá (downvote) nebo je neutrální. Výsledky hlasování jsou ukládány a umožňují později sledovat změny v

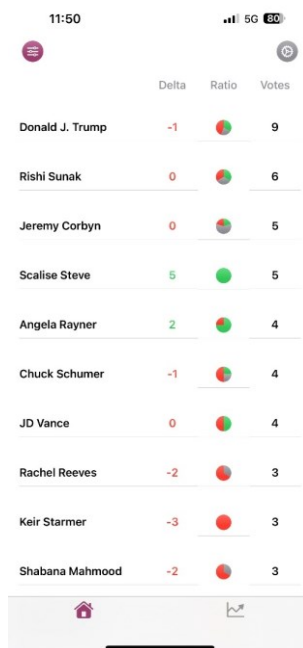
uživatelově názoru nebo vývoj v čase. Politici, na které uživatel hlasoval, zůstávají dostupní v přehledném seznamu pro snadný přístup a další interakce.



Obrázek 3 Portfolio v aplikaci Votap

(zdroj: vlastní)

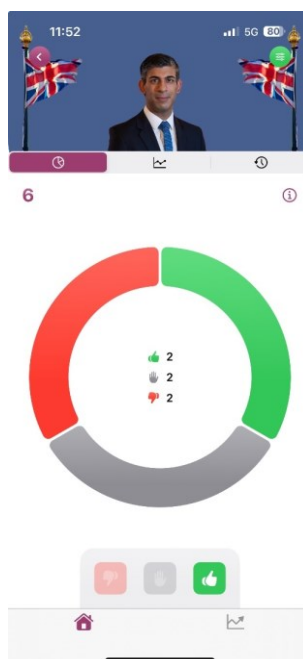
Aplikace dále nabízí přehledné žebříčky politiků, které zobrazují míru zájmu uživatelů. Politici jsou v žebříčku řazeni podle celkového počtu hlasů, přičemž je možné výsledky filtrovat podle země působení daného politika. Jednou z klíčových metrik je tzv. “delta” – rozdíl mezi počtem pozitivních (upvote) a negativních (downvote) hlasů – která pomáhá lépe vystihnout celkové naladění veřejnosti vůči konkrétním osobnostem.



Obrázek 4 Žebříčky v aplikaci Votap

(zdroj: vlastní)

Každý politik má v aplikaci také svůj vlastní profil s detailními statistikami, kde je možné sledovat historický vývoj hlasování, časové trendy podpory i geografické rozložení hlasů. Uživatel tak vidí, jaký názor na daného politika převládá u lidí z různých zemí.



Obrázek 5 Profil politika v aplikaci Votap

(zdroj: vlastní)

11 ARCHITEKTURA APLIKACE

Architektura aplikace byla navržena s důrazem na čistotu kódu, modularitu a udržitelnost při dlouhodobém vývoji. Klíčovým prvkem je využití návrhového vzoru MVVM, který zajišťuje jasné oddělení zodpovědností mezi jednotlivými vrstvami aplikace a umožňuje efektivní testování a opakované použití komponent. Dále jsou implementovány tzv. „service” třídy pro práci se síťovými požadavky, datové repozitáře pro správu dat a přístup k nim, a vzor dependency injection, který pomáhá udržovat volnou vazbu mezi třídami a usnadňuje testování pomocí tzv. „mock” objektů. Významnou roli hrají také znovupoužitelné View komponenty, které zajišťují konzistentní vzhled aplikace.

11.1 Implementace vzoru MVVM

Návrhový vzor MVVM slouží k oddělení prezentační logiky od uživatelského rozhraní. Tento přístup zvyšuje přehlednost, usnadňuje testování a podporuje opakované použití kódu napříč aplikací. Následující části popisují konkrétní roli jednotlivých vrstev MVVM v aplikaci.

11.1.1 Model

Modelová vrstva v architektuře MVVM slouží k reprezentaci aplikačních dat a jejich struktury. V aplikaci Votap se pracuje s rozsáhlým množstvím modelů, které pokrývají různá data — od uživatelských informací až po veřejně dostupné údaje o politicích. Následující ukázka se zaměří na konkrétní datový model reprezentující politika, který je typickým příkladem rozdělení dat na společná a specifická podle země. Toto rozdělení podporuje modularitu a přehlednost datové vrstvy.

```
struct Politician: Identifiable, Codable, ReadableModel {  
    @DocumentID var id: String?  
    let countryCode: String  
    let lastUpdated: Date  
    let common: PoliticianCommonData  
    let countrySpecific: PoliticianCountrySpecificData
```

Obrázek 6 Implementace vrstvy model

(zdroj: vlastní)

Model obsahuje základní atributy jako kód země, čas poslední aktualizace, a dále dvě struktury – „common” pro data společná napříč zeměmi (například jméno, funkce, fotka) a „countrySpecific” pro lokální detaily (například název instituce, regionální zařazení).

Tento přístup zajišťuje, že model je snadno rozšiřitelný a dobře připravený pro podporu politiků z různých zemí.

Důležitým aspektem je také přizpůsobení modelu pro různé zdroje dat. V aplikaci se používá např. jedna varianta modelu přizpůsobená pro práci s cloudovou databází Firebase Firestore (využívá anotaci `@DocumentID`) a jiná pro lokální cache pomocí frameworku SwiftData. To umožňuje efektivní práci s daty v různých kontextech bez narušení celkové architektury aplikace. MVVM tímto způsobem podporuje oddělení logiky zpracování dat od jejich prezentace i perzistence.

Z architektonického hlediska navíc není chybou rozdělit modely na dvě skupiny – tzv. „DTO” objekty (Data Transfer Objects) určené čistě pro mapování dat ze zdrojů, a tzv. doménové modely, které se používají dál v rámci aplikace (např. ve View modelech). Mezi objekty lze pak mít převodní metody, které zajistí správnou transformaci a izolaci datové logiky od zbytku aplikace.

11.1.2 View

Vrstva View tvoří vizuální komponenty aplikace, které uživatel vidí a se kterými interaguje. Ve SwiftUI jsou tyto komponenty strukturami, které implementují protokol s názvem „View”. Základní požadavek tohoto protokolu je implementace proměnné „body”, která deklarativně popisuje uživatelské rozhraní dané komponenty.

```
struct MainView: View {
    @StateObject var portfolio = PortfolioViewModel()
    @StateObject var network = NetworkMonitor()
    @EnvironmentObject var user: UserViewModel
    @EnvironmentObject var snackbar: SnackbarViewModel
    @AppStorage("showPopupMessages") var showPopupMessages = true

    var body: some View {
        ZStack {
            switch network.status {
            case .loading:
                EmptyView()
            }
        }
    }
}
```

Obrázek 7 Implementace vrstvy View

(zdroj: vlastní)

View je v architektuře MVVM napojeno na svůj View model, ze kterého čerpá data a přes který spouští uživatelské akce. Jelikož je View model v jazyce Swift implementován jako třída (class), jedná se o referenční typ – to znamená, že místo kopírování objektu lze předávat dál jeho sdílenou instanci. Díky tomu může více View komponent sdílet jeden View model a pracovat se stejnými daty, což je užitečné zejména pro uchování konzistentního stavu napříč aplikací.

Existují dva běžné způsoby, jak View získá instanci View modelu, přičemž oba pracují s referenčním typem (třídou). Prvním je použití anotace „@StateObject“, která se hodí nejen tehdy, když View instanci View modelu vytváří, ale i v případě, kdy ji přebírá od nadřazené komponenty prostřednictvím svého konstrukturu. V tomto případě je reference předána explicitně a podřízené View s ní dále pracuje, aniž by vytvářelo novou instanci. Druhou možností je využití anotace „@EnvironmentObject“, kdy je instance View modelu vložena do tzv. prostředí (environment) v nadřazené komponentě. Podřízené View si pak tuto instanci samo načte pomocí anotace bez potřeby konstrukturu. Obě varianty umožňují pracovat s jednou sdílenou instancí View modelu, která uchovává data a stav aplikace, a to napříč různými částmi uživatelského rozhraní.

View poté přistupuje k veřejným proměnným a metodám View modelu, aniž by muselo znát jeho interní implementaci. Tím je zachován princip separace odpovědností – View pouze zobrazuje stav a o veškerou logiku se stará View model.

11.1.3 View model

View model v architektuře MVVM slouží jako prostředník mezi View (uživatelským rozhraním) a modelem (daty aplikace) a využívá přitom tzv. „observer pattern“. Tento návrhový vzor spočívá v tom, že objekty (pozorovatelé) se „přihlásí“ k odběru změn jiného objektu (subjektu), a jakmile dojde ke změně, jsou automaticky upozorněny. Ve SwiftUI se tato reakce na změny odehrává automaticky díky anotaci „@Published“, která je používána pro proměnné ve View modelu – jakmile se hodnota takové proměnné změní, příslušné View se automaticky znovu překreslí. View model musí být v takovém případě označen protokolem „ObservableObject“.

```
final class SettingsViewModel: ObservableObject {
    enum LoadingState {
        case loading, ready
    }

    @Published var showLogoutAlert = false
    @Published var loadingState: LoadingState = .ready
    @Published var alertError: AlertError?
    private let authService: AuthService

    init(
        authService: AuthService = FirebaseAuthService()
    ) {
        self.authService = authService
    }

    func signOut() {
```

Obrázek 8 Implementace vrstvy View model

(zdroj: vlastní)

Ve View modelu bývá běžné mít v konstruktoru závislosti na různých service třídách, které zajišťují například autentizaci, práci s databází nebo síťovou komunikaci. Tyto závislosti lze předávat jako parametry, což je základní princip vzoru dependency injection. Detailněji se mu budeme věnovat v samostatné části.

Struktura View modelu obvykle zahrnuje privátní metody, které slouží k internímu zpracování logiky a nejsou přístupné z View, a veřejné metody, které naopak View volá jako reakci na uživatelské interakce – například při stisknutí tlačítka. Tímto způsobem View model zachytává události z UI, zpracuje je, a případně aktualizuje proměnné, čímž se aktualizuje i samotné uživatelské rozhraní.

Je také běžné, že třídy View modelů jsou označovány klíčovým slovem `final`. To znamená, že nemohou být děděny, což má několik výhod – mimo jiné to umožňuje kompilátoru optimalizovat výkon a zároveň zamezuje nechtěnému přepisování metod, což je důležité pro předvídatelnost a stabilitu logiky v aplikaci.

11.2 Service třídy

V aplikaci jsou tzv. „service“ třídy zodpovědné za komunikaci s externími systémy, jako je databáze, síť nebo autentizace. Vzhledem k tomu, že tyto interakce probíhají většinou mimo hlavní vlákno a mohou trvat delší dobu, jsou metody těchto tříd běžně implementovány jako

asynchronní. Hlavním cílem service vrstev je oddělit datovou logiku od zbytku aplikace a zpřehlednit tak View modely, které pak pouze využívají tyto služby a neřeší samotné technické detaily implementace.

```
final class FirestoreService<T: Codable> {  
    private let db = Firestore.firestore()  
}  
  
extension FirestoreService where T: ReadableModel {  
    func read(from path: String) async throws -> T? {  
        let document = try await db.document(path).getDocument()  
  
        guard document.exists else {  
            return nil  
        }  
  
        return try document.data(as: T.self)  
    }  
}
```

Obrázek 9 Service třída pro databázi Firestore

(zdroj: vlastní)

Příkladem takové service třídy může být generická třída „FirestoreService”, která zajišťuje práci s databází Firestore. Generika umožňuje práci s libovolným typem, který splňuje podmínku „Codable”, což znamená, že jej lze snadno zakódovat a dekodovat při práci s daty. Navíc lze pomocí vlastních protokolů omezit, jaké operace jsou pro daný model povolené – například „ReadableModel” je uživatelsky definovaný protokol, který slouží k označení, že daný typ podporuje pouze čtení z databáze. Díky tomu je možné přesně řídit, které modely mohou být pouze načítány (například z Firestore) a které mají mít přístup i k operacím jako je zápis, aktualizace nebo mazání. Tento přístup zvyšuje typovou bezpečnost a umožňuje předcházet chybám už na úrovni kompilace.

Díky tomuto přístupu je možné mít například model politika, který bude pouze „readonly”, což zabrání jeho úpravám nebo mazání z databáze. Tento princip úzce souvisí s tzv. „CRUD” operacemi (Create, Read, Update, Delete), které definují základní sadu akcí pro práci s daty.

11.3 Datové repozitáře

V aplikaci Votap jsou repozitáře navrženy tak, aby uměly pracovat s různými typy úložišť – například se vzdáleným serverem (Firestore) a lokální cache (SwiftData). Každý repozitář

tak poskytuje jednotné rozhraní pro práci s daty, bez ohledu na to, odkud konkrétně pocházejí. View modely díky tomu nemusí řešit, jak a odkud se data získávají, a jednoduše volají například metodu „getPolitician”.

```
final class PoliticianRepository {  
    private let firestore = FirestoreService<Politician>()  
    private let swiftData = SwiftDataService<CachedPolitician>()
```

Obrázek 10 Datový repozitář pro data o politicích

(zdroj: vlastní)

Typickým příkladem je načítání profilů politika. Nejprve se zkusí vyhledat v lokální cache, a pokud tam není dostupný, provede se požadavek na Firestore. Pokud je výsledek úspěšný, data se uloží do cache pro budoucí použití. Tento přístup zlepšuje výkon aplikace a zároveň snižuje potřebu opakovaných síťových dotazů.

11.4 Dependency injection ve View modelech

V ideálním případě by View modely neměly být závislé na konkrétních implementacích služeb, ale měly by místo toho pracovat s abstrakcemi – v jazyce Swift typicky ve formě protokolů. To umožňuje využít princip dependency injection, tedy vkládání závislostí zvenčí, což výrazně zvyšuje flexibilitu a testovatelnost kódu.

Díky tomu je možné například ve výchozím prostředí (produkční aplikace) předat View modelu reálnou službu, která komunikuje s databází Firestore, zatímco v prostředí pro testování lze předat tzv. „mock” třídu – tedy zjednodušenou testovací verzi, která simuluje chování skutečné služby, ale bez reálné komunikace se serverem. Mock třídy mohou například vracet předem připravená data nebo sledovat, jaké metody byly volány.

To je obzvlášť důležité při unit testování, protože testy by měly být rychlé, spolehlivé a izolované. Pokud by testy závisely na skutečné síťové vrstvě, stávaly by se nestabilními a pomalými – a v případě nedostupného serveru by dokonce mohly zcela selhat. Používání protokolů a dependency injection tak přináší čistší architekturu a zároveň usnadňuje vývoj kvalitních a udržitelných testů.

```
init(  
    authService: AuthService = FirebaseAuthService()  
) {  
    self.authService = authService  
}
```

Obrázek 11 Dependency injection v konstruktoru View modelu

(zdroj: vlastní)

Tento View model využívá službu pro autentizaci uživatelů, přičemž není pevně závislý na konkrétní implementaci, ale pouze na protokolu „AuthService”. V běžném provozu se používá výchozí implementace „FirebaseAuthService”, ale díky konstruktoru lze při vytvoření View modelu předat jinou, například testovací mock třídu.

11.5 Znovupoužitelné View komponenty

Znovupoužitelné View komponenty ve SwiftUI slouží k oddělení opakovaně využívaných částí uživatelského rozhraní do samostatných struktur. Typickým přístupem je vytvoření vlastního View, které obsahuje proměnné určitého datového typu – například „String”, „Int”, nebo jakýkoli jiný typ – avšak tyto proměnné nejsou v komponentě inicializovány přímo, ale pouze deklarovány. To znamená, že při vytváření instance komponenty je nutné hodnoty těchto proměnných předat pomocí konstrukturu.

```
struct AsyncImageView: View {  
    let urlString: String?  
    let descriptor: String?  
  
    var body: some View {
```

Obrázek 12 Deklarace znovupoužitelné View komponenty

(zdroj: vlastní)

Takové komponenty jsou univerzální a mohou být použity v různých částech aplikace bez nutnosti duplikace kódu. Stačí pouze předat jiný vstup (například URL obrázku nebo text) a komponenta se podle toho vykreslí. Tento princip výrazně zvyšuje čistotu kódu – pokud je třeba změnit například způsob vykreslování obrázku, stačí tuto změnu provést na jediném místě, a projeví se všude, kde je komponenta použita. Díky tomu lze rychle reagovat na změny v designu nebo logice bez rizika chyb způsobených nekonzistencí mezi více kopiemi téhož kódu.

```
AsyncImageView(  
    urlString: politician.imageUrl,  
    descriptor: politician.displayName  
)
```

Obrázek 13 Použití znovupoužitelné View komponenty

(zdroj: vlastní)

11.6 Pomocné nástroje

Sekce „utilities” (česky často překládána jako pomocné nástroje nebo pomocný kód) představuje část projektu, kde jsou soustředěny obecné pomocné funkce a nástroje, které nejsou specifické pro jednotlivé komponenty View nebo doménovou logiku. Slouží k usnadnění práce napříč aplikací a podporují princip znovupoužitelnosti kódu.

Jedním z nejběžnějších prvků v této sekci jsou tzv. „extensions” – rozšíření existujících nebo vlastních typů o nové metody a vlastnosti. Tyto rozšíření jsou organizovány přehledně, obvykle ve formátu názvu souboru ve tvaru „Typ+Popis.swift”, například „String+MatchesRegex.swift”, což pomáhá rychlé orientaci. Toto rozšíření ověřuje, zda řetězec odpovídá zadanému regulárnímu výrazu.

```
extension String {  
    func matchesRegex(_ regex: String) -> Bool {  
        self.range(  
            of: "^\\(regex)$",  
            options: .regularExpression  
        ) != nil  
    }  
}
```

Obrázek 14 Rozšíření typu String pro podporu systému Regex

(zdroj: vlastní)

Pomocí rozšíření lze nejen přidávat nové funkce, ale také implementovat protokoly pro dané typy. To umožňuje například začlenění typů do určitého rozhraní, což je užitečné při práci s generickými třídami nebo při testování. Pomocné nástroje tedy poskytují důležité stavební bloky pro čistou a udržitelnou architekturu aplikace.

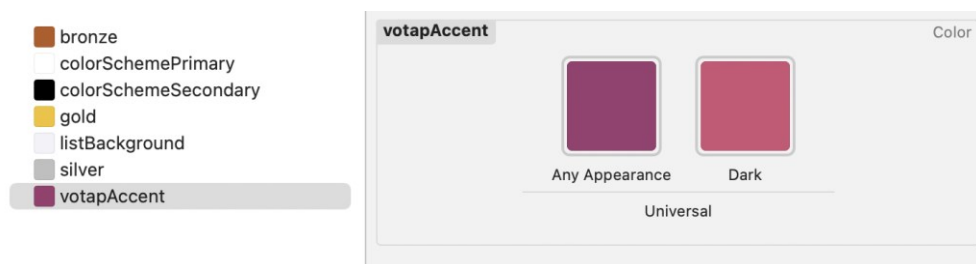
```
extension VoteViewModel: Equatable {
    static func == (lhs: VoteViewModel, rhs: VoteViewModel) -> Bool {
        lhs.id == rhs.id
    }
}
```

Obrázek 15 Rozšíření pro podporu rozhraní Equatable

(zdroj: vlastní)

11.7 Assets

V rámci aplikace jsou veškeré statické zdroje – jako jsou barvy, obrázky, ilustrace nebo datové soubory – součástí širší kategorie „Assets“. Barvy a obrázky jsou spravovány pomocí tzv. „Asset katalogů“, což je speciální struktura v Xcode projektu určená pro centrální správu vizuálních prvků. Barvy jsou definovány s podporou světlého i tmavého režimu, což umožňuje jejich jednoduchou úpravu na jednom místě bez zásahu do kódu. Obrázky a ilustrace jsou rovněž organizovány v katalogu a podporují různé velikosti a škálování pro různá zařízení.



Obrázek 16 Definice vlastních barev v Xcode

(zdroj: vlastní)

Kromě vizuálních prvků zde patří i další podpůrné soubory, které nejsou součástí katalogů, ale jsou uloženy ve složkách projektu. Příkladem je soubor „countries.json“, který obsahuje klíčová metadata o jednotlivých zemích. Pro každou zemi je zde uvedeno například její zobrazované jméno, telefonní předvolba, maska telefonního čísla, regulární výraz pro ověření PSČ, příklady platných poštovních kódů a vlajka reprezentovaná pomocí emoji. Tento soubor slouží jako jednotný zdroj dat pro práci s lokalizací, validací vstupů a filtrováním podle zemí napříč aplikací.

```
"US":{
  "displayName":"United States",
  "phonePrefix":"1",
  "phoneNumberMask": "### ###-####",
  "postalCodeRegex": "\\d{5}([\\-]\\d{4})?",
  "postalCodeExamples": [
    "12345",
    "54321",
    "12345-6789",
    "54321-4321",
    "98765-4321"
  ],
  "flag": "🇺🇸"
},
```

Obrázek 17 Data o USA ze souboru countries.json

(zdroj: vlastní)

12 APLIKACE PRINCIPŮ CLEAN CODE A SOLID

Tato kapitola ukazuje, jak jsou principy clean code a SOLID prakticky aplikovány v návrhu a implementaci aplikace Votap. Na konkrétních příkladech bude demonstrováno, jak tyto principy přispívají k přehlednosti, rozšiřitelnosti, udržitelnosti a testovatelnosti celého řešení. Každý princip bude ilustrován příkladem z reálného kódu s vysvětlením jeho přínosu i možných důsledků jeho porušení.

12.1 Uplatnění principů SOLID

V této kapitole bude ukázáno, jak lze principy SOLID uplatnit v rámci vývoje aplikace ve SwiftUI – a to jak při návrhu service tříd, modelů a View modelů, tak i v celkové architektuře projektu. Každý princip bude ilustrován konkrétní ukázkou z reálného projektu, spolu s vysvětlením správného i problematického použití.

12.1.1 Princip jedné odpovědnosti

Princip říká, že třída by měla vykonávat jen jednu úlohu a neměla by kombinovat více oblastí zodpovědnosti, které spolu přímo nesouvisí.

```
final class NetworkMonitor: ObservableObject {
    enum ConnectionStatus {
        case loading, connected, disconnected
    }

    private let networkMonitor = NWPathMonitor()
    private let workerQueue = DispatchQueue(label: "Monitor")
    @Published var status: ConnectionStatus = .loading

    init() {
        networkMonitor.pathUpdateHandler = { path in
        }
        networkMonitor.start(queue: workerQueue)
    }
}
```

Obrázek 18 Aplikace principu jedné odpovědnosti

(zdroj: vlastní)

V případě třídy „NetworkMonitor” je tento princip dodržen. Jejím jediným účelem je sledovat stav síťového připojení zařízení a publikovat aktuální stav do proměnné „status”,

kterou mohou následně využívat komponenty závislé na síťové dostupnosti. Veškerá logika se soustředí výhradně na monitoring připojení, zpracovávání výsledků a aktualizaci stavu.

Naopak porušení tohoto principu by nastalo například ve chvíli, kdyby tato třída kromě sledování připojení také ukládala data do databáze, zobrazovala uživatelské upozornění nebo odesílala logy na server. Tím by se dostala do zodpovědnosti více domén (UI, data, síť) a každá změna v jedné oblasti by potenciálně narušila ostatní – výsledkem by byl kód hůře čitelný, méně udržitelný a náchylnější k chybám.

12.1.2 Princip otevřenosti a uzavřenosti

Princip říká, že moduly, třídy nebo funkce by měly být otevřené pro rozšíření, ale uzavřené pro změny – tedy že bychom měli umožnit přidání nového chování bez nutnosti zasahovat do stávajícího kódu.

```
final class SwiftDataService<T: PersistentModel> where T.ID == String {
    private let modelContext: ModelContext

    init() {
        ...
    }

    private func saveContext() throws {
        ...
    }

    func printAll() {
        ...
    }
}

extension SwiftDataService where T: ReadableModel {
    func read(where predicate: Predicate<T>) throws -> T? {
        ...
    }
}

extension SwiftDataService where T: WritableModel {
    func create(_ object: T) throws {
        ...
    }

    func update(_ object: T, with updates: (T) -> Void) throws {
        ...
    }
    func delete(_ object: T) throws {
        ...
    }
}
```

Obrázek 19 Aplikace principu otevřenosti a uzavřenosti

(zdroj: vlastní)

Typickým příkladem dodržení tohoto principu je generická třída „SwiftDataService“, která poskytuje základní funkcionalitu pro práci s daty. Namísto toho, aby se do ní přidávaly nové metody přímo, rozšiřuje se pomocí extension, které se aktivují pouze pro typy odpovídající určitým protokolům:

Naopak porušením principu by bylo, kdyby se každá nová funkcionalita (např. zápis nebo mazání) přidávala přímo úpravou původní třídy. Takový přístup by vedl k častým zásahům do existujícího kódu, vyššímu riziku chyb a horší udržitelnosti.

12.1.3 Liskovův princip substituce

Princip říká, že objekty odvozené od nějaké základní třídy by měly být plně zaměnitelné za objekty této třídy, aniž by došlo ke změně správného chování programu. Jinými slovy, pokud kód očekává instanci typu A, měl by být schopen bez problémů pracovat i s instancí typu B, která z A dědí.

V následujícím příkladu se jedná o hypotetickou situaci, protože aktuální architektura aplikace Votap staví primárně na kompozici a protokolech namísto dědičnosti. Nicméně pro účely ilustračního vysvětlení principu lze uvažovat následující konstrukci:

```
class Notification {
    let message: String
    init(message: String) {
        self.message = message
    }

    func display() {
        print("Notification: \(message)")
    }
}

class NewVoteNotification: Notification {
    let politicianName: String
    init(message: String, politicianName: String) {
        self.politicianName = politicianName
        super.init(message: message)
    }

    override func display() {
        print("New vote on \(politicianName): \(message)")
    }
}

class AlertNotification: Notification {
    override func display() {
        print("⚠ Alert: \(message)")
    }
}
```

Obrázek 20 Aplikace dědičnosti v jazyce Swift

(zdroj: vlastní)

Třídy „NewVoteNotification” a „AlertNotification” dědí ze základní třídy „Notification”, přičemž každá implementuje vlastní variantu metody „display”. V praxi pak lze tyto třídy bezpečně používat společně v kolekci:

```

let notifications: [Notification] = [
    NewVoteNotification(
        message: "User voted up",
        politicianName: "Jane Doe"
    ),
    AlertNotification(
        message: "System maintenance tonight"
    )
]

for notification in notifications {
    notification.display()
}

```

Obrázek 21 Aplikace Liskovového principu substituce

(zdroj: vlastní)

Díky tomu, že všechny objekty v poli splňují podmínky Liskovova principu, fungují korektně jako instance základní třídy „Notification”.

Tento mechanismus je v jazyce Swift přirozeně zajištěn skrze podporu objektově orientovaného programování. Vývojář tak nemusí pro dodržení principu provádět žádné speciální kroky – jazyk sám zajistí, že každá podtřída může být použita tam, kde se očekává její nadřazený typ, a to bez dalšího zásahu.

Swift navíc podporuje tzv. „type casting”, což je způsob, jak bezpečně rozlišit konkrétní typ objektu v hierarchii tříd. Například pomocí klíčového slova „as” lze zjistit, zda je daná instance určité podtřidy, a podle toho s ní dále pracovat.

12.1.4 Princip oddělení rozhraní

Princip říká, že žádná třída by neměla být nucena implementovat rozhraní, která nevyužívá. Jinými slovy – místo jednoho velkého protokolu je vhodnější vytvořit více menších, specifických protokolů.

```

protocol AuthService {
    func signOut(completion: @escaping (Result<Void, AuthError>) -> Void)
    func deleteUser(completion: @escaping (Result<Void, AuthError>) -> Void)
}

```

Obrázek 22 Aplikace principu oddělení rozhraní

(zdroj: vlastní)

Příklad správného řešení v rámci aplikace je použití protokolu „AuthService”, který definuje dvě samostatné funkce: „signOut” a „deleteUser”. Tento návrh je v pořádku, pokud všechny typy, které protokol implementují, skutečně obě funkce potřebují.

Pokud by však v systému existoval typ, který má umožňovat například pouze odhlášení uživatele, musel by přesto implementovat i metodu „deleteUser”, přestože pro něj nemá smysl. Takové chování by bylo v rozporu s principem oddělení rozhraní, protože nutí třídu implementovat zbytečnou funkcionalitu.

12.1.5 Princip obrácení závislostí

Tento princip je posledním z pěti zásad SOLID. V jeho jádru stojí myšlenka, že vysokoúrovňové moduly (například View modely) by neměly být závislé na konkrétních implementacích nízko úrovňových tříd (např. služeb pro práci s daty), ale měly by záviset na abstrakcích – nejčastěji ve formě protokolů. Zároveň by ani nízko úrovňové komponenty neměly být závislé na vysokoúrovňových, obě vrstvy mají být odděleny pomocí společného rozhraní.

```
final class SettingsViewModel: ObservableObject {
    private let authService: AuthService

    init(
        authService: AuthService = FirebaseAuthService()
    ) {
        self.authService = authService
    }

    func signOut() {
        authService.signOut { result in
            ...
        }
    }

    func deleteUser() {
        ...
    }
}
```

Obrázek 23 Aplikace principu obrácení závislostí

(zdroj: vlastní)

Tento princip je dobře aplikován v následujícím View modelu, který očekává v konstruktoru instanci služby typu „AuthService”, což je protokol definující požadované

metody. Konkrétní třída „FirebaseAuthService”, která tento protokol implementuje, je zde předána jako výchozí hodnota parametru.

Špatným řešením by bylo, pokud by View model uvnitř sebe přímo vytvářel instanci „FirebaseAuthService”, čímž by vznikla pevná vazba na konkrétní implementaci. Takový přístup by znemožnil snadné testování nebo výměnu logiky bez zásahu do samotného View modelu a porušoval by princip obrácení závislostí. Použitím protokolu a dependency injection se tomu dá efektivně předejít.

12.2 Pomocné metody

Použití pomocných metod je běžnou a doporučovanou praxí při psaní čitelného a udržitelného kódu. Jde o to, že určité části logiky, které se v rámci třídy opakují nebo by zbytečně zatěžovaly hlavní metody, jsou vyčleněny do samostatných privátních metod. Tím se zvyšuje čitelnost kódu, zamezuje se duplicitě a usnadňuje se případné testování nebo refaktoring.

```
final class PoliticianRepository {  
    private let firestore = FirestoreService<Politician>()  
    private let swiftData = SwiftDataService<CachedPolitician>()  
  
    func getPolitician(for id: String) async throws -> Politician? {  
        ...  
  
        try cachePolitician(serverPolitician)  
  
        return serverPolitician  
    }  
  
    private func cachePolitician(_ politician: Politician) throws {  
        ...  
    }  
}
```

Obrázek 24 Použití pomocné metody v repozitáři

(zdroj: vlastní)

Příkladem může být metoda „cachePolitician” v repozitáři „PoliticianRepository”, která slouží k uložení objektu politika do paměti cache. Tato metoda by mohla být snadno zapsána přímo do metody „getPolitician”, ale oddělení do vlastní pomocné funkce zajišťuje lepší strukturu, opakovanou použitelnost a jednodušší orientaci v kódu. Díky tomu je hlavní metoda přehlednější a zaměřuje se čistě na logiku rozhodování, odkud data načíst.

12.3 Psaní komentářů

Komentáře by neměly sloužit k popisu toho, co kód dělá – to by mělo být patrné z jeho struktury a pojmenování. Místo toho by měly vysvětlovat proč kód dělá to, co dělá, případně za jakých podmínek k něčemu dochází. Dobrý komentář přináší hodnotu navíc, například když vysvětluje nečekané chování, rozhodnutí nebo důvod, proč byla daná implementace zvolena.

Na následujícím příkladu je ukázána metoda z aplikace pro přesun prvku v rámci pole. Porovnány jsou dvě verze: první s užitečnými komentáři, druhá se zbytečnými popisy, které jen duplikují to, co je již zřejmé z kódu samotného.

```
extension Array where Element: Equatable {  
  /// Přesune zadaný prvek na novou pozici v rámci pole.  
  /// Pokud prvek není nalezen nebo je nový index neplatný, neprovede se žádná změna.  
  mutating func move(_ item: Element, to newIndex: Int) {  
    // Zabráníme přístupu mimo rozsah pole  
    guard newIndex >= 0 && newIndex < self.count else {  
      return  
    }  
  
    // Vyhledání aktuální pozice prvku v poli  
    if let currentIndex = self.firstIndex(of: item) {  
      // Pokud je prvek již na cílové pozici, není třeba s ním hýbat  
      guard currentIndex != newIndex else { return }  
  
      self.remove(at: currentIndex)  
    }  
  
    self.insert(item, at: newIndex)  
  }  
}
```

Obrázek 25 Příklad správného použití komentářů v kódu

(zdroj: vlastní)

Tato verze doplňuje komentáře pouze tam, kde skutečně rozšiřují srozumitelnost kódu – popisují záměr a důvody jednotlivých kroků. Díky tomu je kód čitelnější, zejména pro vývojáře, kteří s ním přijdou do kontaktu později.

```
// Rozšíření typu Array
extension Array where Element: Equatable {
  // Tato metoda slouží k přesunutí položky na nový index
  mutating func move(_ item: Element, to newIndex: Int) {
    // Zkontroluj, jestli je index v rozsahu
    guard newIndex >= 0 && newIndex < self.count else {
      return
    }

    // Najdi aktuální index položky
    if let currentIndex = self.firstIndex(of: item) {
      // Pokud jsou stejné, tak skonči
      guard currentIndex != newIndex else { return }

      // Odstraň položku
      self.remove(at: currentIndex)
    }

    // Vlož položku
    self.insert(item, at: newIndex)
  }
}
```

Obrázek 26 Příklad chybného použití komentářů v kódu

(zdroj: vlastní)

Komentáře v této variantě také popisují to, co je již zřejmé ze syntaxe a názvů metod. Nepřinášejí žádnou přidanou hodnotu a mohou být vnímány jako šum. Příliš podrobné nebo redundantní komentáře mohou dokonce snižovat čitelnost a ztěžovat údržbu, protože se mohou časem dostat do nesouladu s tím, co kód skutečně dělá.

12.4 Proces refaktoringu

Refaktoring je důležitou součástí vývoje softwaru, která pomáhá zlepšovat kvalitu a čitelnost kódu, aniž by docházelo ke změně jeho funkčnosti. Často se jedná o proces postupných, malých změn, které vedou k lepší udržitelnosti aplikace. V praxi se může stát, že při rychlém vývoji vznikne část kódu, která sice správně plní svůj účel, ale z hlediska struktury a čitelnosti není ideální. Takový kód může být obtížné pochopit, upravit nebo znovu použít.

```
extension Int {  
    var shortString: String {  
        if self >= 1_000_000 {  
            return String(Double(self) / 1_000_000) + "M"  
        } else if self >= 1_000 {  
            return String(Double(self) / 1_000) + "K"  
        } else {  
            return String(self)  
        }  
    }  
}
```

Obrázek 27 Ukázka kódu před refaktoringem

(zdroj: vlastní)

V následujícím příkladu je ukázána původní, z hlediska čistého kódu méně vhodná implementace funkce, která formátuje číselné hodnoty do zkráceného zápisu (například 1.2K nebo 3.4M). Ačkoliv tato verze technicky funguje, její čitelnost, opakovatelnost a rozšiřitelnost jsou omezené. V kódu se objevuje nepřehledná podmínková logika typu „if-else“, chybí kontrola výstupního formátu – čísla se zobrazují i s nadbytečnými desetinnými místy, a jsou zde použita tzv. „magická čísla“ (1000, 1000000), jejichž význam není jasný na první pohled. Také samotné sestavování řetězce probíhá pomocí spojování textu s číslem přes operátor „+“, což není z hlediska přehlednosti ideální. Název proměnné „shortString“ navíc neříká jednoznačně, co přesně daná funkce vrací.

```
// Formátování čísel s příponami (např. 1.2K nebo 3.4M)
extension Int {
    var suffixFormat: String {
        switch abs(self) {
            case 1_000..<1_000_000:
                // Hodnoty mezi tisícem a milionem formátujeme s příponou K
                return format(divisor: 1_000, suffix: "K")
            case 1_000_000...:
                // Hodnoty nad milion formátujeme s příponou M
                return format(divisor: 1_000_000, suffix: "M")
            default:
                // Hodnoty menší než tisíc zobrazujeme bez úpravy
                return String(self)
        }
    }
}

/// Pomocná metoda pro výpočet formátovaného řetězce s příponou
private func format(divisor: Double, suffix: String) -> String {
    let value = Double(self) / divisor
    return String(format: "%.1f%@", value, suffix)
}
}
```

Obrázek 28 Ukázka kódu po refaktoringu

(zdroj: vlastní)

Tato verze představuje ukázkový příklad aplikace refaktoringu v praxi. Proměnná nyní nese název „suffixFormat“, který přesněji vystihuje její účel. Oproti původní implementaci, která měla veškerou logiku obsaženou v jedné funkci, je kód nově rozdělen do menších a lépe srozumitelných částí. Cílem této úpravy bylo zvýšit čitelnost, opakovatelnost a snadnější údržbu.

Jednou z hlavních změn je oddělení zodpovědností. V původní verzi probíhal výpočet hodnoty i její formátování přímo uvnitř matematického výrazu, čímž docházelo k opakování stejné logiky s různými parametry. V nové verzi je opakující se výpočet přesunut do pomocné metody „format“ která má na starosti převod čísel na řetězec s odpovídající příponou. Díky tomu je každá část kódu zodpovědná pouze za jediný úkol, čímž se naplňuje princip jedné odpovědnosti.

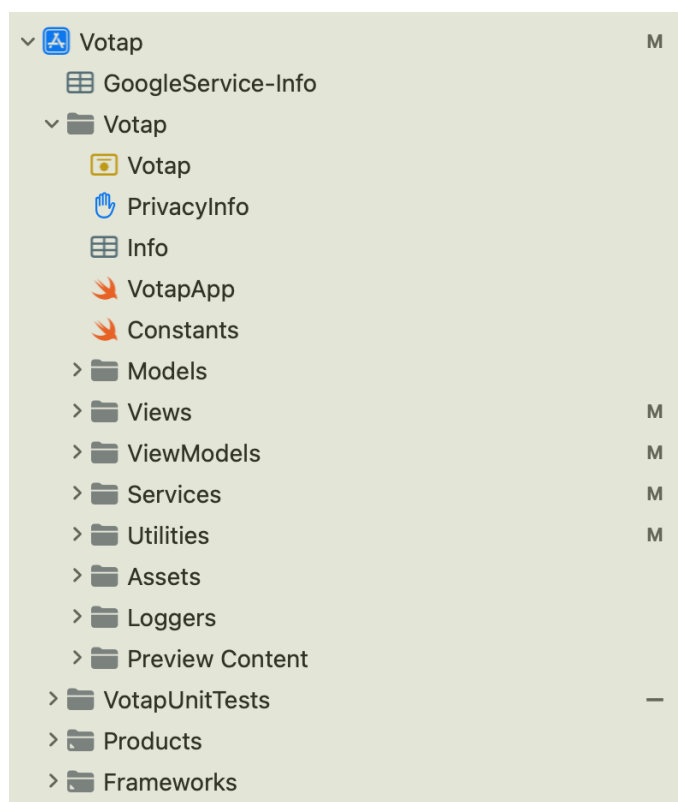
Výsledný kód je tak nejen výrazně čitelnější, ale i připravený na budoucí rozšíření – například o formátování hodnot v miliardách nebo trilionech. Přidání nové větve do „switch“ výrazu je díky centralizované logice formátování snadné a nevyžaduje duplikaci stávajícího kódu. Kromě toho je celý kód opatřen smysluplnými komentáři.

13 STRUKTURA PROJEKTU

Dobře navržená struktura projektu představuje základní kámen úspěšného a udržitelného vývoje aplikace. Jasná organizace složek, konzistentní pojmenovávání a respektování týmových konvencí umožňují nejen rychlejší orientaci v kódu, ale také efektivnější spolupráci více vývojářů. Tato kapitola se věnuje tomu, jakým způsobem je projekt Votap rozdělen do jednotlivých částí, jaké konvence byly při vývoji dodržovány a jaké osvědčené postupy mohou být užitečné při škálování nebo dlouhodobé údržbě aplikace.

13.1 Organizace a pojmenování souborů

Na obrázku níže je znázorněna struktura projektu v prostředí Xcode. Projekt je přehledně rozdělen do jednotlivých složek podle funkcionality a architektonických vrstev, což výrazně přispívá k lepší čitelnosti a udržitelnosti kódu.



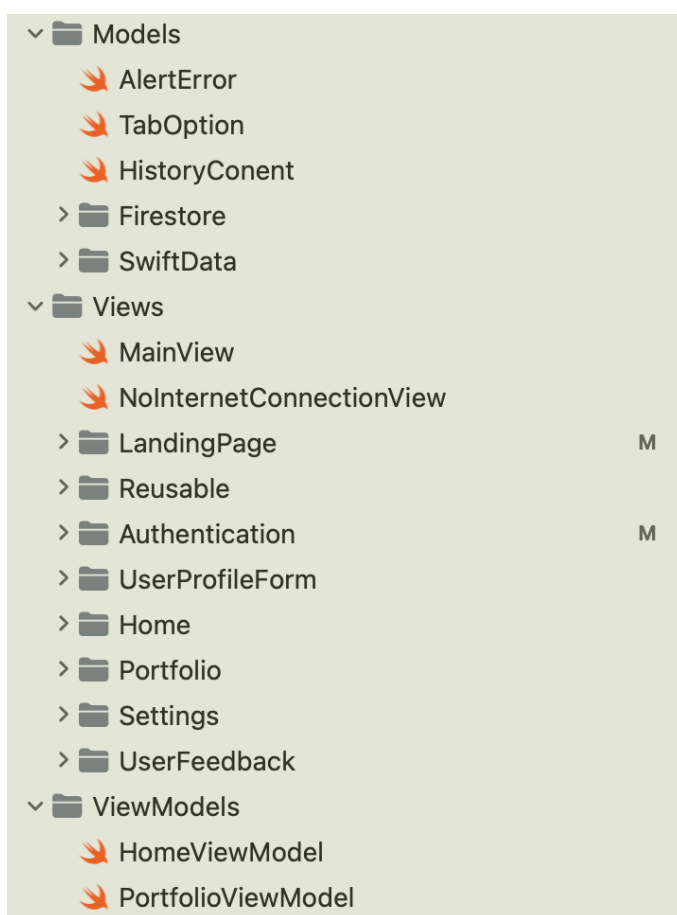
Obrázek 29 Celková souborová struktura aplikace Votap

(zdroj: vlastní)

Hlavní soubor „VotapApp” obsahuje vstupní bod aplikace označený pomocí atributu „@main”. V rámci SwiftUI aplikací tento soubor definuje výchozí konfiguraci aplikace.

Obvykle se zde vytváří kořenový View a inicializují se základní komponenty (například prostředí pro „@EnvironmentObject“).

Soubor „Constants“ slouží jako centrální místo pro uložení konstant, které se opakovaně používají v různých částech aplikace. Příkladem může být slovník podporovaných zemí, základní URL cesty, identifikátory pro analytiku nebo obecné řetězce. Centralizace těchto hodnot snižuje riziko chyb způsobených duplikací nebo nejednotností a usnadňuje jejich údržbu (změna na jednom místě se automaticky projeví ve všech částech aplikace).



Obrázek 30 Souborová struktura MVVM v aplikaci Votap

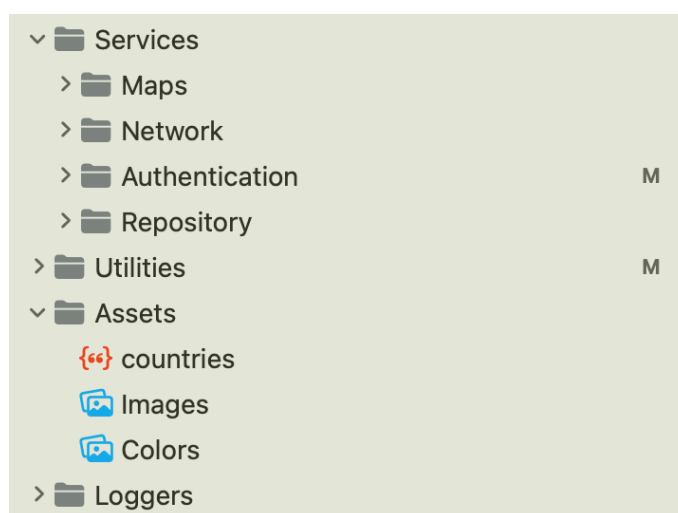
(zdroj: vlastní)

Struktura projektu Votap je organizována podle architektonického vzoru MVVM, tedy do složek „Models“, „Views“ a „ViewModels“. Tento způsob rozdělení podporuje oddělení zodpovědností jednotlivých vrstev aplikace a přispívá k lepší přehlednosti a rozšiřitelnosti kódu. Při pojmenovávání souborů je dodržována jednotná konvence, která z názvu ihned napoví, do které vrstvy soubor patří.

Ve složce Models se nacházejí datové struktury reprezentující stavové nebo doménové objekty aplikace. Modely by měly být vždy pojmenovány jako podstatná jména – například „TabOption“, „AlertError“ nebo „HistoryContent“, protože reprezentují entity. Dále lze tyto modely třídit do podsložek podle jejich zaměření nebo původu dat, jako například Firestore a SwiftData, což pomáhá oddělit modely používané pro různé zdroje dat (např. síťová data vs. lokální cache).

Složka Views obsahuje uživatelské rozhraní aplikace. Typicky zde najdeme soubor „MainView“, který představuje výchozí bod z pohledu UI – například obsahuje navigační strukturu aplikace. Jednotlivé části aplikace (např. přihlašování, nastavení, domovská stránka) jsou dále rozděleny do tematických složek. Tento způsob členění do podadresářů zvyšuje přehlednost v projektech s větším množstvím obrazovek. Každý soubor ve složce končí slovem „View“, aby bylo na první pohled jasné, že se jedná o vizuální komponentu.

Do složky ViewModels jsou pak umístěny všechny View modely – logické vrstvy zodpovědné za transformaci dat a obsluhu interakcí. Zde se důsledně dodržuje konvence, že každý soubor končí příponou „ViewModel“, což odpovídá jejich účelu. View modely jsou většinou vázané ke konkrétní obrazovce a jejich pojmenování odpovídá názvu komponenty, se kterou spolupracují – např. „SettingsViewModel“, „PortfolioViewModel“, „LandingPageViewModel“.



Obrázek 31 Zbytek souborové struktury v aplikaci Votap

(zdroj: vlastní)

Kromě složek odpovídajících architektuře MVVM obsahuje projekt i další klíčové části, které slouží k podpoře aplikační logiky a infrastruktury.

Ve složce „Services“ se nachází jednotlivé servisní třídy, které zajišťují konkrétní funkcionality, jako je síťová komunikace (Network), autentizace uživatele (Authentication), práce s mapovými podklady (Maps) nebo datové repozitáře (Repository).

Složka „Utilities“ slouží jako prostor pro různé pomocné nástroje, přičemž v tomto projektu se zde nacházejí především extensions – tedy rozšíření standardních nebo vlastních typů o další užitečné metody.

V „Assets“ jsou uloženy veškeré multimediální a podpůrné zdroje. Konkrétně se zde nachází dva asset katalogy – „Images“ a „Colors“. Obrázky a ilustrace jsou spravovány centrálně v katalogu „Images“, zatímco barevná paleta aplikace, včetně podpory pro světlý a tmavý režim, je definována v katalogu „Colors“. Mimo katalogy se zde nachází i samostatný soubor „countries.json“, který obsahuje doplňující data o státech.

Nakonec soubory ve složce „Loggers“ slouží k logování (zaznamenávání důležitých událostí a chování aplikace během běhu). V aktuálním stavu obsahuje pouze „ConsoleLogger“ (logování do konzole), ale její struktura umožňuje budoucí rozšíření – například o logování do souboru, do vzdáleného serveru nebo o integrované analytické nástroje.

Celková organizace projektu tak reflektuje nejen architekturu a komponentovou strukturu aplikace, ale i potřebu oddělit podpůrné nástroje a zdroje do samostatných a přehledně členěných celků.

13.2 Modularizace a oddělení domén

V aktuální verzi aplikace Votap není zatím využita modularizace ani explicitní oddělení domén, jelikož rozsah projektu to v této fázi nevyžaduje. Nicméně s rostoucí složitostí aplikace a počtem funkcionalit může být modularizace velmi užitečná – zejména pro zajištění přehlednosti, znovupoužitelnosti kódu, rychlejší kompilace a efektivní spolupráce většího týmu.

Modularizace v Xcode znamená rozdělení projektu do samostatných modulů – tedy dílčích logických celků, které mají jasně definované odpovědnosti a rozhraní. Tyto moduly mohou být implementovány jako Swift Package, frameworky, případně dynamické knihovny. Každý modul pak může být samostatně testován, verzován a opakovaně použit v různých částech projektu nebo i napříč více aplikacemi.

V aplikaci Votap by bylo možné v budoucnu oddělit například celý subsystém práce s daty do samostatného modulu. Tento modul by zahrnoval veškeré repozitáře, servisní třídy zajišťující komunikaci s databázemi Firestore a SwiftData, stejně jako odpovídající modely.

Další oblastí vhodnou pro modularizaci je autentizace. Ta by mohla být zabalena do samostatného modulu obsahujícího implementaci rozhraní „AuthService” s celou logikou spojenou s přihlašováním uživatelů, správou jejich účtů nebo mazáním profilu.

Třetím příkladem je oddělení hlavních UI komponent. Tyto komponenty, jako například „AsyncImageView”, vlastní tlačítka, notifikace nebo další opakovaně používané vizuální prvky, mohou být zpracovány jako samostatný modul, který se využívá napříč celou aplikací bez závislosti na konkrétní doménové logice.

Každý z těchto modulů by pak měl své vlastní rozhraní (pomocí protokolů), které by bylo dostupné ostatním částem aplikace, aniž by se musely zabývat interní implementací. Například „PoliticianRepository” by mohl být deklarován jako protokol v hlavní aplikaci a samotná implementace – včetně komunikace s Firestore a cache – by mohla být v modulu „DataRepository”.

Takový přístup by výrazně přispěl k udržitelnosti aplikace v případě jejího dalšího růstu, například pokud by se rozšířila o nový frontend (část aplikace, kterou vidí a používá uživatel, např. uživatelské rozhraní a interakce v mobilní aplikaci), nebo pokud by se vyvíjelo více verzí aplikace s různými funkcemi pro různé trhy.

13.3 Práce s konfigurací a prostředím

Při vývoji aplikací je velmi důležité oddělit jednotlivá prostředí, zejména produkční a testovací. Důvodem je bezpečnost a stabilita systému – testování nad reálnými daty v produkčním prostředí může vést k nechtěným změnám či únikům dat. Navíc v testovacím prostředí je běžné simulovat chování systémů, které v ostrém provozu fungují jinak – například síťové výpadky, offline režimy, latenci apod.

Pro vývoj aplikace Votap se využívá služba Firebase Emulator, což je lokální simulace služeb Firebase. Emulator umožňuje vývojářům testovat databázi, autentizaci i další části systému bez připojení k reálným serverům Firebase. Díky tomu je možné testovat funkčnost aplikace a interakce se vzdálenými službami, aniž by docházelo ke změnám v produkční databázi. Emulator je vhodný zejména pro vývoj a unit/integration testy.

Pro přepínání mezi jednotlivými prostředími je aplikace nakonfigurována pomocí tzv. preprocesorových maker. Tato makra slouží k rozlišení chování aplikace v závislosti na aktivní konfiguraci sestavení.

```
#if FIREBASE_EMULATOR
setupFirebaseEmulator()
#elseif DEBUG
ConsoleLogger.info("Testing app in debug mode.")
#endif
```

Obrázek 32 Použití preprocesorových maker

(zdroj: vlastní)

Například při zapnutí konfigurace „FIREBASE_EMULATOR” se v aplikaci provede volání funkce „setupFirebaseEmulator”, která nastaví spojení s lokálním serverem. V jiném případě, například v „DEBUG” režimu, se provede běžná inicializace nebo se pouze vypíše log pro účely vývoje.

Tento přístup umožňuje vývojářům snadno přepínat mezi jednotlivými prostředími bez nutnosti ručních úprav ve více částech kódu. V prostředí Xcode je možné definovat tzv. „build konfigurace”, jako „Debug”, „Release”, nebo nově vytvořenou konfiguraci „Firebase Emulator”. Každé z těchto konfigurací lze přiřadit specifický kompilační „flag”, což je návěští, které lze následně použít v kódu pomocí preprocesorových direktiv jako „#if”. Díky tomu se při sestavování aplikace automaticky aktivuje příslušná část kódu odpovídající zvolené konfiguraci.

Tímto způsobem lze například zajistit, že v „Emulator” režimu se inicializuje připojení k lokální instanci Firebase, zatímco v „Debug” režimu proběhne standardní testování aplikace bez produkčního prostředí. V „Release” režimu pak aplikace používá ostré služby a připojení. Tento způsob práce výrazně zvyšuje bezpečnost, snižuje riziko nechtěné práce s reálnými daty a zároveň zjednodušuje přepínání mezi prostředími pro různé účely vývoje a testování.

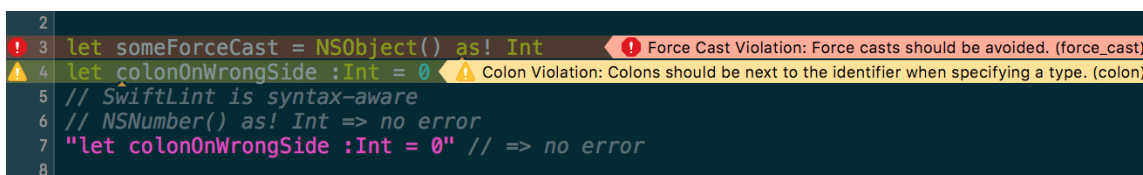
13.4 Projektová kultura a nástroj SwiftLint

Projektová kultura představuje soubor pravidel, zvyklostí a konvencí, které vývojový tým dodržuje při psaní a správě zdrojového kódu. Jejím cílem je zajistit konzistenci, čitelnost a snadnou údržbu celého projektu – zejména pokud na aplikaci pracuje více vývojářů. Pokud

každý programuje trochu jiným způsobem, může dojít k chaotickému kódu, zbytečným chybám a zpomalení vývoje. Jedním z klíčových nástrojů, který pomáhá udržet tým na jedné línii, je SwiftLint.

Jde o nástroj třetí strany (není oficiálně vytvořen společností Apple), který provádí statickou analýzu kódu v jazyce Swift. Na základě předem definovaných pravidel kontroluje, zda kód odpovídá stanoveným konvencím a standardům. Pomocí jednoduché konfigurace lze například nastavit, že každá proměnná musí mít název delší než dva znaky, že mezi funkcemi musí být určitý počet prázdných řádků nebo že názvy tříd musí začínat velkým písmenem.

Tato pravidla je možné upravit nebo doplnit podle potřeb konkrétního projektu nebo týmu – konfigurace se provádí v souboru „swiftlint.yml“. Nástroj lze integrovat přímo do procesu buildování aplikace v Xcode, takže před každým sestavením proběhne kontrola kódu. Výsledky analýzy se mohou zobrazit jako „warning“ (upozornění, které vývojáře informuje, ale nebrání spuštění) nebo jako „error“, který zabrání kompilaci, dokud není problém vyřešen.



```
2
3 let someForceCast = NSObject() as! Int
4 let colonOnWrongSide :Int = 0
5 // SwiftLint is syntax-aware
6 // NSNumber() as! Int => no error
7 "let colonOnWrongSide :Int = 0" // => no error
8
```

Obrázek 33 Ukázka nástroje SwiftLint

(zdroj: <https://github.com/realm/SwiftLint>)

První chyba (zvýrazněná červeně) upozorňuje na použití vynuceného přetypování, což je v jazyce Swift považováno za rizikovou praxi, protože při neúspěchu může vést k pádu aplikace za běhu.

Druhé varování (žluté) se týká nevhodného umístění dvojtečky při definici typu. SwiftLint v tomto případě hlásí varování s vysvětlením, že dvojtečka by měla být přímo za názvem proměnné bez mezery. I když se jedná o čistě stylistickou záležitost, dodržování těchto pravidel přispívá k jednotnému a profesionálnímu vzhledu celého kódu.

Nástroj SwiftLint tímto způsobem přirozeně učí vývojáře lepším návykům a pomáhá udržet vysokou kvalitu kódu napříč celým týmem i v dlouhodobém horizontu.

14 DOKUMENTACE

Dokumentace hraje klíčovou roli při dlouhodobé udržitelnosti a rozšiřování softwarových projektů. V prostředí Xcode lze pro tento účel efektivně využít nástroj DocC, který umožňuje vytvářet přehlednou, strukturovanou a propojenou dokumentaci přímo z anotací v kódu. Výhodou tohoto přístupu je, že dokumentace zůstává v těsné blízkosti implementace a může být snadno aktualizována při změnách ve zdrojovém kódu. V následujících podkapitolách bude popsáno, jak se dokumentace pomocí DocC vytváří, jak může vypadat výsledná forma a jak zajistit její aktuálnost v průběhu vývoje.

14.1 Použití DocC v Xcode

Použití nástroje DocC v Xcode spočívá v psaní speciálních dokumentačních komentářů přímo do zdrojového kódu. Tyto komentáře se nejčastěji přidávají na úroveň tříd, struktur, metod nebo protokolů. Zatímco u pohledů (Views) se často uvádí jen stručné popisy, u metod je běžné detailně komentovat vstupní parametry, návratové hodnoty a účel samotné funkce. Z těchto komentářů lze následně vygenerovat formální dokumentaci v tzv. „Apple stylu“, tedy ve formátu, který je známý například z oficiální dokumentace jazyka Swift, SwiftUI nebo Foundation frameworku. Tento výstup může obsahovat jak čistý text, tak i formátovaný kód, přehledné seznamy, poznámky nebo hypertextové odkazy na další entity v rámci projektu – jako jsou funkce, objekty, enumerace, proměnné a další.

14.2 Ukázka strukturované dokumentace

Zde je ukázka použití strukturované dokumentace ve formátu DocC, která je zapsána přímo nad deklarací metody. Komentář popisuje, co metoda dělá, jaký očekává vstup, co vrací a obsahuje také poznámku a příklad použití.

```
extension String {  
    /// Checks whether the string fully matches the provided regular expression pattern.  
    ///  
    /// This method uses regular expression matching to determine whether the entire string  
    /// conforms exactly to the given regex pattern. It anchors the pattern automatically with `^` and `$`  
    /// to ensure that partial matches are not considered valid.  
    ///  
    /// - Parameter regex: A `String` representing the regular expression to match against.  
    /// - Returns: `true` if the entire string matches the regular expression, otherwise `false`.  
    ///  
    /// - Note: This method anchors the pattern at both the start and end of the string.  
    /// If you want partial matches, consider using `range(of:options:)` without anchoring.  
    ///  
    /// ### Example  
    /// ```swift  
    /// let email = "example@email.com"  
    /// let regex = "[A-Z0-9a-z._%+-]+@[A-Za-z0-9.-]+\\.[A-Za-z]{2,64}"  
    /// let isValid = email.matchesRegex(regex)  
    /// print(isValid) // true  
    /// ```  
    func matchesRegex(_ regex: String) -> Bool {  
        return self.range(of: "^\\(regex)$", options: .regularExpression) != nil  
    }  
}
```

Obrázek 34 Ukázka dokumentačních komentářů

(zdroj: vlastní)

Výsledkem této dokumentace je automaticky generovaný výpis v Xcode. Zobrazí se také při použití nástroje „Quick Help”, ale také může být součástí exportované dokumentace v podobě webové stránky. Následující obrázek ukazuje výslednou podobu této dokumentace.

```
func matchesRegex(_ regex: String) -> Bool
```

Parameters

regex

A String representing the regular expression to match against.

Return Value

true if the entire string matches the regular expression, otherwise false.

Discussion

This method uses regular expression matching to determine whether the entire string conforms exactly to the given regex pattern. It anchors the pattern automatically with `^` and `$` to ensure that partial matches are not considered valid.

Note

This method anchors the pattern at both the start and end of the string. If you want partial matches, consider using `range(of:options:)` without anchoring.

Example

```
let email = "example@email.com"
let regex = "[A-Z0-9a-z._%+-]+@[A-Za-z0-9.-]+\\.[A-Za-z]{2,64}"
let isValid = email.matchesRegex(regex)
print(isValid) // true
```

Obrázek 35 Ukázka výsledné dokumentace

(zdroj: vlastní)

14.3 Udržování a aktualizace

Udržování a aktualizace dokumentace je zásadní součástí vývoje softwaru, zejména v týmech a u projektů s delší životností. Pokud se dokumentace nevyvíjí společně s kódem, pak i velmi rychle přestává plnit svou funkci. Může to vést k tomu, že vývojáři budou uváděni v omyl, použijí rozhraní nesprávným způsobem nebo se zpomalí zaškolování nových členů týmu. Každá změna v implementaci – ať už se jedná o úpravu názvu metody, parametrů, návratového typu nebo samotné logiky – by měla být vždy doprovázena odpovídající aktualizací dokumentace. Ideálně by tak úprava dokumentace měla být přirozenou součástí procesu refaktoringu, a to jako jeho závěrečný krok. Pokud k tomu nedojde, může dokumentace začít popisovat chování, které už ve skutečnosti neexistuje, nebo naopak opomíjí nové vlastnosti a funkce.

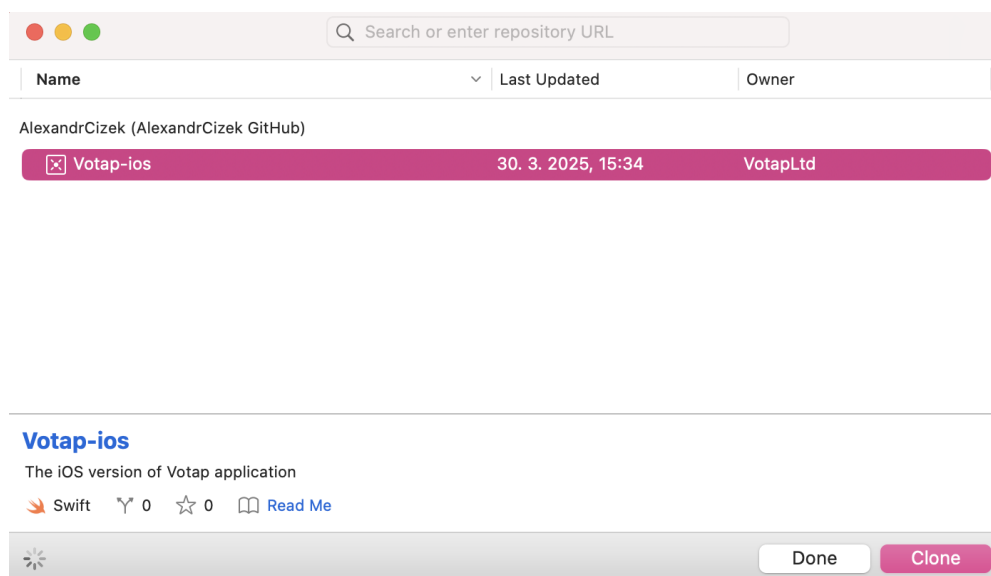
V praxi je proto výhodné nastavit pravidla a nástroje, které vývoj dokumentace podporují. Lze využívat například nástroj SwiftLint nebo kontinuální integrační procesy (CI), které automaticky kontrolují přítomnost dokumentačních komentářů u veřejných částí rozhraní. Dále se osvědčuje použití šablon pro komentáře, které zajistí konzistenci napříč celým projektem. Dobrým zvykem je také zahrnout kontrolu dokumentace do procesu revize kódu – například při vytváření pull requestu by měl reviewer ověřit, že spolu s úpravou kódu došlo i k adekvátní úpravě dokumentace. Díky těmto návykům lze udržet projektovou dokumentaci aktuální, přehlednou, a především užitečnou pro další vývoj.

15 VERZOVÁNÍ

Správa verzí je nedílnou součástí vývoje každé moderní aplikace a v případě iOS vývoje se nejčastěji využívá systém Git. Ten slouží k verzování zdrojového kódu, sledování změn, možnosti návratu ke starším verzím a zejména pro spolupráci více vývojářů. V rámci vývoje aplikace Votap je Git používán v kombinaci s hostingovou službou GitHub, která poskytuje webové rozhraní pro správu repozitářů.

15.1 Klonování repozitáře

Po úvodním nastavení verzovacího systému Git a vytvoření repozitáře (například na platformě GitHub) je dalším krokem tzv. „klonování“ (vytvoření kopie) projektu v Xcode. Tento proces začíná volbou možnosti „Clone from Git repository“, kde uživatel zadá URL adresu repozitáře.



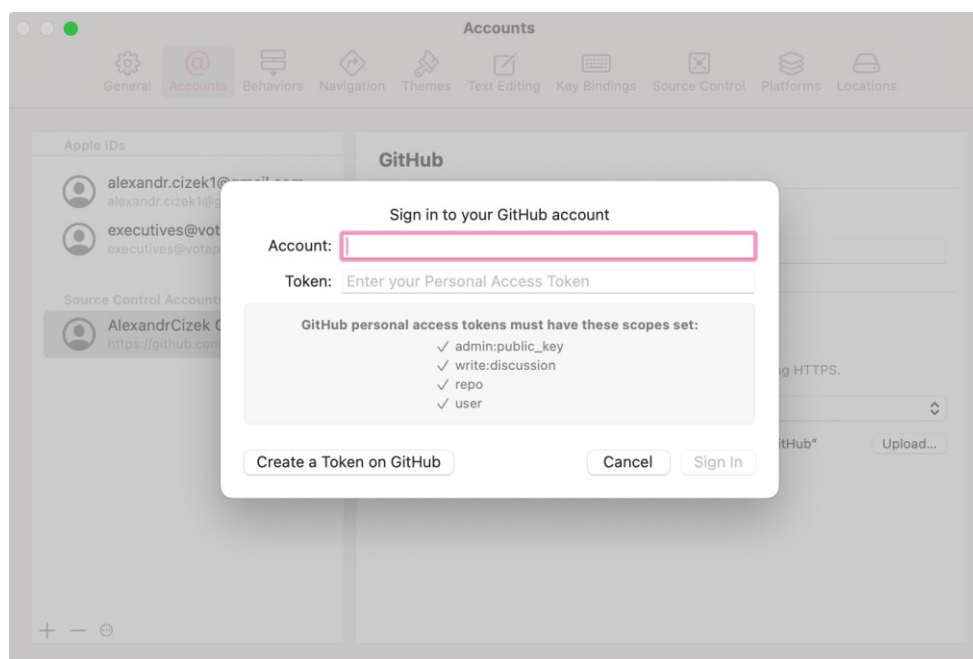
Obrázek 36 Klonování repozitáře v Xcode

(zdroj: vlastní)

Na obrázku je vidět dialogové okno v Xcode, ve kterém je dostupný vzdálený repozitář „Votap-ios“. Zde je možné vybrat konkrétní repozitář a také zvolit, se kterou větví chce vývojář pracovat. Po stisknutí tlačítka „Clone“ se projekt stáhne na lokální disk a je připraven k dalšímu vývoji přímo v prostředí Xcode. Tento krok je nezbytný zejména v týmové spolupráci, kdy více vývojářů pracuje na stejném kódu v různých větvích a pravidelně změny sdílí prostřednictvím verzovacího systému.

15.2 Integrace systému Git v Xcode

Apple nabízí přímou integraci systému Git v prostředí Xcode, což vývojářům umožňuje spravovat verzování bez nutnosti přecházet do příkazové řádky. Prvním krokem při integraci systému Git v prostředí Xcode je přidání uživatelského účtu ke vzdálenému repozitáři. Na obrázku je zachyceno okno pro přihlášení ke GitHub účtu, které se nachází v nastavení Xcode pod záložkou „Accounts“.

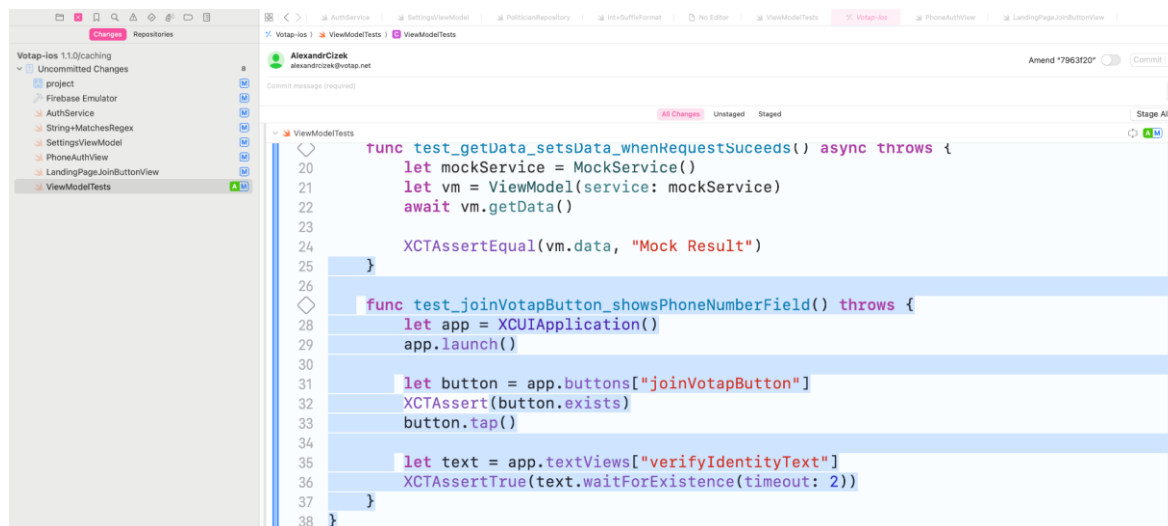


Obrázek 37 Přihlášení k účtu GitHub v Xcode

(zdroj: vlastní)

Pro přihlášení je nutné vyplnit název účtu a tzv. „Personal Access Token“ (unikátní přístupový kód vygenerovaný uživatelem, který slouží k ověření identity místo klasického hesla a umožňuje bezpečný přístup k API nebo službám jako GitHub). Tento token se vygeneruje přímo na platformě GitHub a je potřeba jej následně zkopírovat a vložit do Xcode.

Na dalším obrázku je vidět prostředí Xcode s otevřeným „Source Control“ navigátorem, který slouží k vizuálnímu zobrazení změn v rámci repozitáře. V levé části okna jsou vypsané jednotlivé soubory, které byly změněny (M), přidány (A) nebo smazány (D). Tyto změny jsou seskupené pod konkrétní větví a čekají na potvrzení.



Obrázek 38 Source Control panel v Xcode

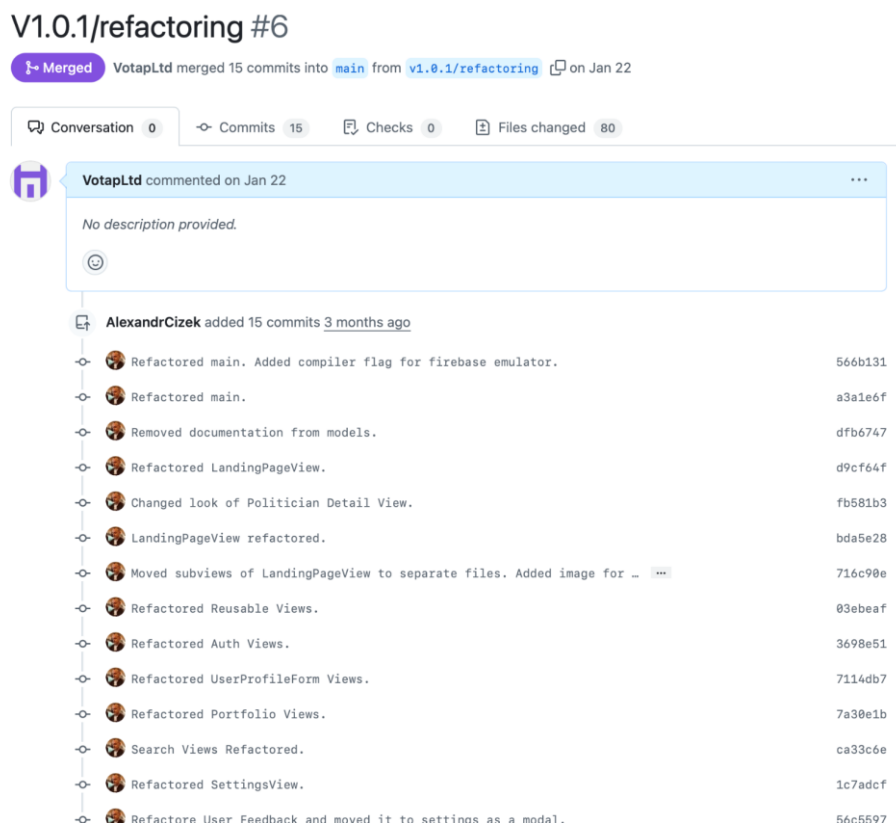
(zdroj: vlastní)

V horní části lze zadat popis (commit message), který by měl stručně vystihovat, co bylo v dané sadě změn provedeno. Po potvrzení změn (stiskem commit) je možné je také odeslat na vzdálený server (push), odkud je uvidí i ostatní členové týmu.

Ačkoliv je možné všechny tyto operace provádět i přes příkazový řádek (přes příkazy jako git add, git commit, git push a další) nebo nástroje jako Git Bash, integrace verzovacího systému přímo v Xcode přináší výhodu přehledného uživatelského rozhraní, které celý proces zjednodušuje a urychluje. Výsledkem je ale vždy stejný výstup – historie verzí kódu, která je synchronizovaná s repozitářem na platformě GitHub nebo jiné službě.

15.3 Větvení a pull requests

Na dalším obrázku je vidět proces práce s větvemi a pull requesty v systému GitHub, který je nedílnou součástí moderního týmového vývoje. Typickým scénářem je, že pro novou funkci nebo úpravu kódu vývojář nejprve vytvoří vlastní větev, na které pracuje odděleně od hlavní verze aplikace. Tato separace umožňuje nezávislý vývoj více funkcí paralelně bez rizika, že si vývojáři navzájem přepíšou nebo naruší práci.



Obrázek 39 Pull request v platformě GitHub

(zdroj: vlastní)

Po dokončení práce ve větvi a vytvoření jednotlivých commitů přichází na řadu pull request. Jedná se o žádost o začlenění změn do jiné větve – např. do hlavní větve projektu. Tento krok nebývá automatický, ale naopak je příležitostí k tzv. „code review“, tedy revizi kódu jiným členem týmu. Ten zkontroluje kvalitu, funkčnost, čitelnost i soulad s projektovými konvencemi. Cílem tohoto procesu je zajistit, aby se do hlavní větve dostával pouze ověřený a udržitelný kód.

Součástí pull requestu je i komentář autora, přehled změněných souborů, počet commitů a stav testů. Po schválení dochází ke sloučení změn – a tím se daná větev většinou uzavírá. Výsledkem je přehledný vývojový proces s jasným sledem změn a vyšší celkovou kvalitou kódu. Tímto způsobem lze zajistit systematickou práci celého týmu i v rozsáhlejších projektech.

16 TESTOVÁNÍ

Testování je nedílnou součástí moderního vývoje softwaru a hraje klíčovou roli při zajišťování kvality, spolehlivosti a udržitelnosti aplikace. V rámci vývoje iOS aplikací pomocí Swift a SwiftUI existuje několik úrovní a přístupů k testování, které pokrývají jak jednotlivé funkční části aplikace, tak i celé uživatelské rozhraní. V této kapitole bude představena sada technik a nástrojů používaných k testování aplikace Votap.

Následující část se zaměří především na unit testy, které ověřují logiku View modelů, včetně práce s asynchronními metodami a závislostmi řešenými pomocí dependency injection a mock tříd. Dále se podíváme na testování uživatelského rozhraní ve SwiftUI a návrh black-box testovacích scénářů, které simulují reálné chování uživatele bez ohledu na vnitřní implementaci. V neposlední řadě bude popsán i proces testování prostřednictvím platformy TestFlight, která umožňuje distribuci aplikace uživatelům ve fázi alpha a beta verze, ještě před jejím finálním zveřejněním v App Store.

16.1 Unit testy

V této části se zaměříme na použití nástroje XCTest, který slouží k psaní unit testů v jazyce Swift. Na praktických ukázkách si představíme testování jednotlivých komponent aplikace, zejména View modelů, využití mock objektů a práci s asynchronními metodami. Cílem je ukázat, jak testovat funkčnost kódu izolovaně a efektivně.

16.1.1 Struktura a pojmenování

V testovacích třídách v jazyce Swift, které implementují rozhraní „XCTestCase”, se dodržuje určité strukturální rozvržení. Každá taková třída obsahuje metody „setUpWithError” a „tearDownWithError”, které se spouští před a po každém jednotlivém testu. Samotné testovací metody pak začínají klíčovým slovem „test” a nesmí mít žádné parametry. Příklad základní struktury může vypadat následovně:

```
final class ViewModelTests: XCTestCase {
    override fun setUpWithError() throws {
        // This method is called before the invocation of each
        test method in the class.
    }

    override fun tearDownWithError() throws {
        // This method is called after the invocation of each test
        method in the class.
    }

    fun test_funcName_condition_expectedResult() throws {

    }
}
```

Obrázek 40 Struktura třídy pro unit testování v XCTest

(zdroj: vlastní)

Při pojmenovávání jednotlivých testů je doporučeno používat konvenci, která je přehledná pro účel každého testu. Nejčastěji se používá struktura ve formátu „test_nazevTestovaneFunkce_podminka_ocekavanyVysledek”.

Například „test_signOut_setsAlertError_whenSignOutFails” jasně popisuje, co se testuje, za jaké situace, a jaký má být výsledek. Takové názvy pomáhají rychle pochopit chování testu bez nutnosti nahlížet do samotné implementace.

16.1.2 View modely

Testování View modelů je v rámci unit testů klíčové, protože tato vrstva zpracovává logiku aplikace a propojuje ji s uživatelským rozhraním. Testy se zaměřují především na konstruktory, kde ověřujeme, zda se po vytvoření instance správně nastaví výchozí hodnoty jednotlivých vlastností. Dále testujeme veřejné metody, které reagují na interakce uživatele nebo komunikují se službami – například načtení dat nebo zpracování akcí. Naopak privátní metody není možné testovat přímo, protože nejsou přístupné mimo třídu. Tyto části logiky by proto měly být ověřeny nepřímo prostřednictvím veřejných metod, které je využívají.

Pokud by View modely měly přímé závislosti na konkrétních service třídách místo využití protokolu (rozhraní), nebylo by možné je efektivně testovat. Při spuštění unit testu by totiž bylo nutné používat reálné služby, které mohou provádět operace jako zápis do databáze nebo volání sítě. To je ale pro unit testy nežádoucí – testy by byly pomalé, nestabilní a závislé na dostupnosti vnějších systémů.

16.1.3 Mock objekty

Příkladem řešení této situace je vytvoření protokolu „Service”, který definuje rozhraní pro získání dat. Tento protokol pak implementují dvě třídy – reálná „FirestoreService”, která by v produkční aplikaci komunikovala například se serverem, a „MockService”, která vrací předem připravený výsledek pouze pro účely testování. View model pak není závislý na konkrétní třídě, ale pouze na protokolu, což umožňuje během testů předat jinou implementaci – právě mock.

```
protocol Service {  
    func getData() -> String  
}  
  
class FirestoreService: Service {  
    func getData() -> String {  
        // real server request  
        return "Real Result"  
    }  
}  
  
class MockService: Service {  
    func getData() -> String {  
        // mock result for unit testing  
        return "Mock Result"  
    }  
}  
  
class ViewModel: ObservableObject {  
    @Published var data = ""  
    private let service: Service  
  
    init(service: Service) {  
        self.service = service  
    }  
  
    func getData() {  
        data = service.getData()  
    }  
}
```

Obrázek 41 Použití rozhraní ve View modelu

(zdroj: vlastní)

Na základě výše uvedené architektury je možné velmi jednoduše vytvořit unit test. Během testu použijeme instanci „MockService”, která simuluje očekávané chování. Díky tomu máme plnou kontrolu nad vstupy i výstupy a testujeme čistě logiku View modelu.

```
func test_getData_setsData_whenRequestSucceeds() throws {  
    let mockService = MockService()  
    let vm = ViewModel(service: mockService)  
    vm.getData()  
  
    XCTAssertEqual(vm.data, "Mock Result")  
}
```

Obrázek 42 Příklad unit testu pro View model

(zdroj: vlastní)

Test zajišťuje, že po zavolání metody „getData“ View model správně uloží hodnotu získanou z poskytnuté služby (v tomto případě MockService). Výhodou tohoto přístupu je predikovatelnost, jednoduchost a izolovanost testu – není potřeba žádná konfigurace backendu ani připojení k internetu. Takový test je spolehlivý a může být spouštěn opakovaně s konzistentními výsledky.

16.1.4 Asynchronní metody

Při testování asynchronních metod vzniká problém – pokud by se ve View modelu zavolala asynchronní metoda a následně okamžitě provedly testovací assert funkce, existuje riziko, že tyto funkce proběhnou ještě předtím, než metoda skutečně dokončí svou práci. To by mohlo vést k chybnému vyhodnocení testu, i když je samotná logika v pořádku. Proto je nutné při testování asynchronních operací použít vhodné mechanismy pro počkání na dokončení.

Pokud metoda „getData“ ve View modelu pracuje s asynchronní službou – například získává data ze serveru – je potřeba upravit její implementaci tak, aby podporovala „async“. Následně je třeba test také přizpůsobit, aby počkal na dokončení volání a správně vyhodnotil výsledek.

```
func test_getData_setsData_whenRequestSucceeds() async throws {  
    let mockService = MockService()  
    let vm = ViewModel(service: mockService)  
    await vm.getData()  
  
    XCTAssertEqual(vm.data, "Mock Result")  
}
```

Obrázek 43 Unit test s asynchronní metodou

(zdroj: vlastní)

16.2 UI testy ve SwiftUI

UI testy slouží k automatizovanému ověření, že uživatelské rozhraní aplikace funguje podle očekávání. Simulují reálné interakce uživatele – například stisknutí tlačítka, psaní do textových polí nebo navigaci mezi obrazovkami – a kontrolují, zda aplikace na tyto akce správně reaguje.

I přesto, že UI testy nejsou v praxi tak často používány, mohou přinést významnou přidanou hodnotu – obzvlášť u aplikací, kde je důležitá plynulá interakce uživatele a stabilita uživatelského rozhraní.

Následující test je ukázkou základního UI testu v aplikaci Votap, jehož cílem je ověřit, že se po kliknutí na tlačítko „Join Votap” zobrazí nová obrazovka s textem „Verify Your Identity”. Aby bylo možné tyto prvky v rámci testu spolehlivě identifikovat, musí být opatřeny tzv. „identifikátory pro přístup” pomocí metody „accessibilityIdentifier”. Tyto identifikátory nejsou viditelné pro uživatele, ale slouží právě pro účely testování.

```
Button("Join Votap") {  
    }  
    .accessibilityIdentifier("joinVotapButton")  
  
Text("Verify Your Identity")  
    .accessibilityIdentifier("verifyIdentityText")
```

Obrázek 44 Identifikátory pro UI testy ve SwiftUI

(zdroj: vlastní)

V samotném UI testu se pak vytváří instance aplikace pomocí objektu „XCUIApplication”, která se poté spouští. Následuje vyhledání tlačítka podle jeho identifikátoru a ověření, že prvek skutečně existuje. Po kliknutí tlačítka se očekává, že se zobrazí nový prvek – komponenta s textem. Protože zobrazení nové obrazovky nemusí být okamžité, používá se metoda „waitForExistence”, která dává systému čas (v tomto případě 2 sekundy), aby mohl požadovaný prvek načíst.

```

func test_joinVotapButton_showsVerifyIdentityText() throws {
    let app = XCUIApplication()
    app.launch()

    let button = app.buttons["joinVotapButton"]
    XCTAssert(button.exists)
    button.tap()

    let text = app.textViews["verifyIdentityText"]
    XCTAssertTrue(text.waitForExistence(timeout: 2))
}

```

Obrázek 45 Příklad UI testu

(zdroj: vlastní)

Je třeba dodat, že tvorba a údržba UI testů je časově náročnější než u unit testů. Jsou závislé na konkrétním vzhledu a struktuře rozhraní, a každá větší změna v UI může znamenat i úpravu testů. Proto se obvykle doporučuje UI testy používat pro tzv. „kritické cesty“ aplikace, kde je stabilita obzvlášť důležitá (například přihlášení, registrace, dokončení hlasování apod.).

16.3 Návrh black-box testů

V rámci vývoje aplikace Votap je součástí každého nového vydání revize a případná aktualizace sady black-box testů, které slouží k ověření správného chování aplikace z pohledu koncového uživatele. Tyto testy jsou navrhovány bez znalosti vnitřní implementace – tedy pouze na základě očekávaného vstupu a výstupu. Hlavní myšlenkou black-box testování je neřešit, jak daná funkce uvnitř funguje, ale pouze ověřit, že na zadané vstupy reaguje správným způsobem.

Tests v0				
Tr ID	Tr Title	Tr Description	Tr Steps	Tr Expected Results
AUTH_001	Valid Phone Number Input	Verify that entering a valid phone number (+16505551234) displays a loading indicator and then navigates to the screen to enter a 6-digit verification code.	1. Open the app and navigate to the phone number input screen. 2. Enter the valid phone number +16505551234. 3. Press the "Continue" or "Next" button (or whatever action is required to submit the phone number).	1. A loading indicator (such as a spinner or progress bar) should appear immediately after submitting the phone number. 2. After a few seconds (depending on the network speed or predefined loading time), the loading indicator should disappear. 3. The app should navigate to the screen that prompts the user to enter a 6-digit verification code.
AUTH_002	Invalid Phone Number Input	Verify that entering an invalid (non-existing) phone number shows a loading indicator, returns to the initial screen, displays an alert that the number is invalid, and clears the phone number text field.	1. Open the app and navigate to the phone number input screen. 2. Enter an invalid phone number (e.g., +1234567890). 3. Press the "Continue" or "Next" button (or the appropriate action button to submit the phone number).	1. A loading indicator should appear immediately after submitting the phone number. 2. After a few seconds, the loading indicator disappears. 3. The app returns to the initial screen where the phone number is entered. 4. An alert is displayed, stating that the phone number is invalid. 5. The phone number text field is cleared and ready for new input.
AUTH_003	Invalid Format for Verification Code Input	Verify that when entering a verification code in an invalid format (e.g., fewer than 6 digits, more than 6 digits, non-numeric characters), the "Submit" button remains grayed out and inactive until a valid 6-digit numeric code is entered.	1. Open the app and navigate to the phone number input screen. 2. Enter a phone number in an invalid format: • Example 1: abc123def (contains alphabetic characters). • Example 2: 123 (too few digits).	1. The "Continue" or "Next" button remains grayed out and inactive. 2. The app does not display a loading indicator or attempt to proceed to the next screen. 3. The user is unable to submit the phone number until it is corrected to a valid format.
AUTH_004	Change Number After Valid Phone Number Input	Verify that after successfully entering a valid phone number and reaching the verification screen, selecting "Change Number" navigates back to the phone number input screen with the phone number field cleared.	1. Follow all steps in Test Case AUTH_001 to reach the screen where the 6-digit verification code is entered. 2. On the verification code screen, tap the "Change Number" button.	1. The app navigates back to the initial phone number input screen. 2. The phone number text field is cleared and ready for new input.

Obrázek 46 Návrh black-box testů

(zdroj: vlastní)

Na přiloženém obrázku je zachycena tabulka s návrhem testů. Každý řádek představuje jeden konkrétní testovací případ, který obsahuje základní popis testu, podrobné kroky k jeho provedení a očekávaný výsledek. Tabulka dále obsahuje informace s datem testování, výsledku a jménu testera. Takto vytvořená struktura umožňuje testy snadno revidovat, verzovat a rozšiřovat o další položky – například o předpoklady testu (pre-conditions), stav po provedení (post-conditions), případně prioritu nebo předpokládaný dopad chyby.

Typickým prostředím pro provádění těchto testů jsou simulátory (emulátory zařízení), případně reálná zařízení s předprodukční konfigurací – tedy nikoli s ostrou produkční databází, ale s reálnými službami (například Firebase), které jsou ale oddělené od produkčních dat.

16.4 Alpha a beta testování pomocí TestFlight

V rámci distribuce testovacích verzí aplikací pro platformu iOS je standardním nástrojem služba TestFlight, která vývojářům umožňuje snadno sdílet aplikaci s testery před oficiálním vydáním v App Store. TestFlight je přímo propojen s vývojovým účtem Apple Developer a umožňuje testování interními testery a externími uživateli.

Při testování aplikací se často rozlišují dvě fáze. Alpha testování představuje interní testování aplikace ve velmi raném stadiu vývoje. Provádí jej obvykle samotní vývojáři nebo blízcí členové týmu. Cílem je ověřit základní funkčnost a stabilitu hlavních částí aplikace ještě předtím, než je vystavena širšímu publiku. V rámci TestFlight to znamená přidání tzv. „internal testers“, kteří mají přístup ihned po nahrání nové verze do App Store Connect.

Beta testování pak následuje po fázi alpha a zahrnuje širší skupinu uživatelů – často uživatele z cílové skupiny, kteří testují aplikaci v reálných podmínkách. Tito testéři jsou pozváni prostřednictvím soukromého odkazu a jejich zpětná vazba je cenným zdrojem při ladění chyb, UI/UX problémů nebo testování výkonu před zveřejněním aplikace.

V praxi se tedy aplikační balíček vygenerovaný v Xcode nahrává do App Store Connect, kde je možné jej následně označit jako build určený k testování. Pro alpha testování se přidají interní Apple ID účty členů týmu, pro beta testování se vytvoří odkaz pro sdílení. Každý tester obdrží pozvánku do aplikace TestFlight, kterou si stáhne zdarma z App Store a kde může testovanou verzi aplikace stáhnout, používat a zasílat zpětnou vazbu přímo vývojářům.



Obrázek 47 Distribuce aplikace na App Store Connect

(zdroj: vlastní)

Tímto způsobem je možné včas odhalit chyby, které by se jinak mohly objevit až v produkčním nasazení, a zároveň získat cenný vhled do uživatelského vnímání aplikace ještě před jejím oficiálním spuštěním.

17 AGILNÍ VÝVOJOVÝ NÁVRH PRO TÝM

V první verzi aplikace Votap probíhal veškerý vývoj pouze v rámci jednoho vývojáře, a tedy nebylo nutné nastavovat týmový proces ani organizační strukturu vývoje. Celý cyklus návrhu, implementace, testování i nasazení byl realizován individuálně. Se zapojením dalších členů však vzniká potřeba zavést udržitelný vývojový proces, který umožní efektivní spolupráci v týmu, rychlejší iteraci a přehlednost v rámci správy kódu i vydávání nových verzí.

Aplikace zahrnuje například anonymní autentizaci uživatelů pomocí telefonního čísla, zpracování geografické polohy podle poštovního směrovacího čísla, práci s reálnými daty politiků uloženými v databázi a jejich interaktivní vizualizaci. Návrh dále vychází z hypotetického scénáře, v němž má tým k dispozici produktového manažera, který komunikuje se stakeholderem – v tomto případě investorem a zakladatelem projektu – a zajišťuje přenos priorit, zpětné vazby a celkového směřování produktu.

Cílem této části je nastavit jasné rozdělení rolí, navrhnout způsob správy verzí, definovat práci ve sprintech a přiblížit způsoby spolupráce v týmu. Tento návrh reflektuje potřeby a výzvy typické pro moderní vývoj mobilních aplikací a může být snadno přizpůsoben reálnému týmu. Je však důležité zdůraznit, že se jedná o jeden z možných přístupů, nikoli o univerzální řešení vhodné pro každý projekt.

17.1 Rozdělení rolí v týmu

Následující kapitola se věnuje rozdělení rolí v rámci agilního vývojového týmu, jak by mohl být navržen pro vývoj aplikace Votap. Popisuje jednotlivé zodpovědnosti a přínosy každé role v kontextu spolupráce a vývoje.

17.1.1 Product manager

Product manager (PM) je prostředníkem mezi vývojovým týmem a stakeholderem (v tomto případě investorem nebo zakladatelem). Je odpovědný za celkovou vizi produktu, plánování a prioritizaci úkolů. Vytváří a spravuje tzv. „backlog“, což je seznam všech funkcí, zlepšení, oprav chyb a technických úkolů, které je třeba provést. PM rozhoduje, které úkoly se dostanou do nejbližšího sprintu, a během vývoje zajišťuje, že práce týmu odpovídá potřebám uživatelů a cílům projektu. V případě aplikace Votap by např. definoval, jak má fungovat hlasovací mechanismus, jaká data jsou nejdůležitější zobrazit na detailu politika nebo jak má vypadat onboarding nového uživatele.

17.1.2 UI/UX designer

Designer připravuje návrhy obrazovek a interakcí podle požadavků PM a zpětné vazby od uživatelů. Vytváří prototypy jednotlivých částí aplikace – například úvodní přihlašovací flow (zadání čísla, ověření pomocí kódu, výběr lokality), rozhraní pro hlasování (UI tlačítek, zpětná vazba po hlasu), žebříčky politiků nebo stránku s detailem politika. Připravuje různé verze návrhů, které jsou následně validovány s týmem a případně testovány s uživateli.

17.1.3 Vývojář

Vývojáři implementují jednotlivé části aplikace na základě návrhů designéra a priorit stanovených PM. Každý může mít specializaci dle části aplikace – např. autentizace a správa profilu, hlasování a portfolio, žebříčky a veřejná data. Vývojáři také spolupracují na návrhu architektury, struktury projektu a kódu, přičemž dbají na principy clean code, MVVM a SOLID. Každý vývojář vytváří samostatné větve v systému Git, pravidelně provádí commity a pull requesty, které jsou následně zkontrolovány ostatními (code review). Vývojáři také připravují jednotkové testy a podílí se na integraci s backendem a konfigurací testovacích prostředí.

17.1.4 Tester

Tester má na starosti kontrolu funkčnosti aplikace z pohledu uživatele. Vytváří testovací scénáře (včetně black-box testů) podle toho, co bylo ve sprintu vyvinuto, a ověřuje, zda aplikace funguje správně i po nasazení nových verzí. V případě aplikace Votap testuje například celý přihlašovací flow, odesílání hlasů, správné zobrazení žebříčků podle zemí a validaci vstupních údajů (např. PSČ). Také připravuje a spravuje testování přes TestFlight (včetně rozlišení mezi alfa a beta verzí).

17.2 Práce ve sprintech a plánování

Pro řízení vývoje aplikace Votap volíme agilní metodologii Scrum, která je jednou z nejrozšířenějších forem agilního řízení softwarových projektů. Scrum je vhodný zejména tam, kde se požadavky na produkt mohou v průběhu vývoje měnit, a kde je potřeba rychle reagovat na zpětnou vazbu od uživatelů nebo stakeholderů. Aplikace Votap, zaměřená na aktuální politické dění a okamžitou odezvu uživatelů, takový dynamický přístup vyžaduje.

Vývoj bude probíhat v opakujících se cyklech, tzv. sprintech, které budou mít délku dva týdny. Každý sprint začíná plánovacím setkáním, kde tým společně s product

managerem vybere z backlogu konkrétní úkoly (user stories), které chce v daném sprintu realizovat. Úkoly budou vybírány na základě jejich priority a odhadované náročnosti. Každá user story by měla být dobře popsána, ideálně tak, aby obsahovala jasná kritéria dokončení.

Na začátku sprintu si tým stanoví cíle a rozdělí si jednotlivé úkoly. Během sprintu se každý den koná krátká schůzka (daily scrum), kde členové týmu shrnou, na čem pracují, s čím případně potřebují pomoci, a co plánují udělat dále.

Po ukončení sprintu probíhá tzv. „sprint review“, kde tým ukáže hotovou funkcionalitu product managerovi a stakeholderovi (v našem případě zakladateli aplikace), a dále „sprint retrospective“, kde si tým interně zhodnotí, co se povedlo, co je třeba zlepšit a co bude dělat jinak v dalším sprintu.

17.3 Strategie verzování

V rámci vývoje aplikace Votap je použit Git jako hlavní verzovací systém a GitHub jako platforma pro správu repozitáře a týmovou spolupráci.

Celková strategie vychází z tzv. „Feature Branch Workflow“, přičemž jsou do procesu integrovány některé prvky klasického Gitflow, upravené tak, aby vyhovovaly menšímu a rychlému týmu. Struktura repozitáře bude postavena na dvou hlavních větvích. První je větev „main“, která bude reprezentovat stabilní a produkčně připravenou verzi aplikace. Druhou klíčovou větví je „develop“, která bude sloužit jako vývojová větev, do které se budou slučovat všechny nové funkcionality a opravy po jejich schválení v rámci code review.

Každá nová funkcionalita, nebo větší úprava bude vyvíjena v samostatné větví, která bude vždy založena z aktuální verze větve develop. Pro pojmenování větví se bude používat jednotný formát odpovídající typu změny, například „feature/nazev-funkce“ nebo „bugfix/nazev-opravy“, což přispěje k přehlednosti celého repozitáře. Vývojář si tak vždy nejprve vytvoří vlastní větev, ve které pracuje a do které pravidelně ukládá své změny formou commitů. Popisy commitů by měly být stručné, ale výstižné, například „feat: přidání hlasování“ nebo „fix: oprava validace telefonního čísla“.

Po dokončení funkcionality vytvoří vývojář pull request zpět do větve develop. Každý pull request by měl být zkontrolován jiným členem týmu (vývojářem), který provede kontrolu kódu, upozorní na případné nedostatky a ověří správnost a kvalitu implementace.

Schválený pull request se následně sloučí do develop a příslušná feature větev se smaže, aby se udržovala čistota v seznamu větví.

Správa verzí bude založena na principu sémantického verzování (Semantic Versioning), kde číslo verze má formát ve tvaru „major.minor.patch”, například 1.3.2. Zvýšení major verze znamená změnu, která není zpětně kompatibilní, minor verze značí přidání nové funkcionality zpětně kompatibilním způsobem a patch verze označuje opravu chyby bez přidání nové funkcionality. Na konci každého sprintu, po dokončení testování a stabilizace kódu, se z větve develop vytvoří release větev (release/1.0.0). Tato větev bude po otestování sloučena do main a zároveň zpět do develop, aby byly zachovány všechny změny. Všechny vydání budou označeny pomocí Git „tagů”, aby bylo možné snadno sledovat historii verzí.

Vývojáři budou v průběhu vývoje pravidelně aktualizovat své feature větve o změny z develop, aby minimalizovali riziko konfliktů při slučování. Všechny konflikty musí být vyřešeny lokálně před vytvořením pull requestu.

Výsledkem tohoto procesu je, že kód ve větvi develop bude vždy v relativně stabilním stavu a bude připraven k nasazení, zatímco větev main bude obsahovat pouze ověřený, produkční kód. Strategii verzování tak pevně propojujeme s agilním vývojem ve sprintech, kde každý sprint končí stabilní a vydanou verzí aplikace připravenou k testování v TestFlight nebo k nasazení na App Store.

17.4 Spolupráce pomocí nástrojů

Efektivní týmová spolupráce při vývoji aplikace, jako je Votap, vyžaduje vhodnou podporu pomocí moderních nástrojů, které usnadňují řízení práce, komunikaci i samotný vývojový a testovací proces. Správně nastavené nástroje pomáhají udržet přehled o projektu, snižují riziko chyb a umožňují rychleji reagovat na změny požadavků.

Pro správu úkolů a řízení vývoje bude využíván nástroj Jira. Jde o robustní systém určený pro plánování sprintů, evidenci backlogu, přiřazování úkolů jednotlivým členům týmu a sledování jejich postupu. Každý úkol v Jira obsahuje podrobný popis, prioritu, stav a komentáře, což umožňuje transparentní řízení práce a snadné přizpůsobení se agilnímu vývoji. Typicky zde vznikají již zmíněné user stories (uživatelské příběhy) nebo technické úkoly, které jsou rozděleny do jednotlivých sprintů podle aktuálních priorit projektu.

Pro každodenní komunikaci v týmu bude sloužit platforma Slack. Slack je komunikační platforma, která umožňuje rychlé zasílání zpráv, tvorbu tematických kanálů (např. #development, #testing, #design) a integraci s dalšími nástroji, jako je GitHub nebo Jira. Díky nástroji Slack má tým možnost okamžitě reagovat na změny, diskutovat o problémech v reálném čase a zůstat v neustálém kontaktu, což je klíčové zejména při distribuovaném nebo plně vzdáleném vývoji.

Zdrojový kód aplikace je verzován pomocí platformy GitHub, který již byl popsán v předchozí části.

Pro automatizaci sestavování (buildování) aplikace, spouštění testů a doručování testovacích verzí bude využit Xcode Cloud. Jedná se o službu přímo od Apple, která je integrována s Xcode a umožňuje nastavit tzv. CI/CD pipeline (Continuous Integration/Continuous Deployment). CI znamená, že při každé změně v repozitáři (např. schválení pull requestu) se automaticky spustí sestavení aplikace a unit testy, čímž se rychle odhalí chyby. CD následně umožňuje automatické doručování funkčních verzí testerům nebo na TestFlight. V našem případě Xcode Cloud zajistí, že se každá verze před spuštěním testování automaticky sestaví, otestuje a odešle interním testerům bez potřeby ručních zásahů.

Pro distribuci alfa a beta verzí aplikace testerům využijeme TestFlight, který umožňuje snadné sdílení verzí aplikace s interními i externími uživateli, sběr zpětné vazby a odhalování chyb před veřejným vydáním aplikace.

Díky propojení těchto nástrojů – Jira pro řízení práce, Slack pro komunikaci, GitHub pro správu kódu, Xcode Cloud pro automatizaci buildů a TestFlight pro testování – je možné vytvořit ucelený a efektivní vývojový proces. Tento proces podporuje agilní přístup, umožňuje častou integraci změn, včasné testování a zvyšuje kvalitu celkového výstupu.

17.5 Průběh jedné konkrétní iterace

V této části bude detailně popsán návrh jedné iterace (sprintu) v rámci agilního vývoje aplikace Votap. Cílem je ilustrovat, jak by mohl prakticky vypadat celý proces od prvotního zadání úkolů až po dokončení sprintu, včetně rozdělení práce v týmu, použití nástrojů a způsobu spolupráce.

Je důležité zdůraznit, že se jedná pouze o návrh vycházející ze specifik aplikace Votap a dříve popsaných rolí a metodologie. V praxi je vždy nutné takový proces flexibilně přizpůsobit konkrétním potřebám týmu, zkušenostem členů i aktuálnímu stavu projektu.

Tento příklad slouží primárně pro demonstraci možného postupu a ukázání logické návaznosti jednotlivých činností v rámci jednoho sprintu.

17.5.1 Komunikace product managera se stakeholdery

Hlavním cílem této fáze je získat jasné zadání, porozumět prioritám produktu a přetavit je do konkrétní podoby, se kterou může vývojový tým dále pracovat. Na začátku projektu aplikace Votap bylo cílem umožnit uživatelům rychlé a anonymní vytvoření účtu, založené na ověření telefonního čísla pomocí SMS kódu, a poskytnout možnost hlasovat pro politiky v reálném čase.

PM tedy vede úvodní schůzku, kde společně se stakeholderem definuje klíčové funkce, které musí být součástí první verze (MVP – Minimum Viable Product). Z této schůzky vzniká první produktový backlog, který obsahuje například tyto základní požadavky:

- Uživatel může zadat své telefonní číslo a obdržet SMS ověřovací kód.
- Po ověření čísla zadá své PSČ a na základě něj se určí země uživatele.
- Uživatel si může prohlížet seznam politiků v jeho zemi.
- Uživatel může hlasovat (upvote/downvote/neutral) pro politiky.

Kromě toho stakeholder vyjádří i některé důležité preference, například:

- Aplikace musí být jednoduchá na použití a rychlá.
- Důraz na anonymitu – žádné ukládání osobních dat mimo to, co je nezbytné.
- První zaměření na trhy USA a Velká Británie.

Tyto informace PM zaznamená a připraví tak podklady pro plánování sprintu. Jeho úkolem je přenést vizi stakeholdera do technického a vývojového jazyka, srozumitelného pro celý tým. PM zároveň stanoví předběžné priority – například, že ověřování telefonního čísla a vytvoření účtu je naprostým základem a musí být dokončeno dříve než funkce související s prohlížením politiků.

17.5.2 Příprava backlogu a plánování sprintu

Po úvodní komunikaci Product Managera se stakeholderem následuje interní příprava sprintu. Product Manager nyní spolupracuje s celým týmem na rozpracování produktového backlogu do konkrétních uživatelských scénářů.

Každá user story stručně popisuje, co má uživatel možnost v aplikaci udělat, proč je to důležité a jak poznáme, že je požadavek správně splněn (akceptační kritéria). V případě aplikace Votap by například v rámci prvního sprintu vznikly následující user stories:

- Jako uživatel chci ověřit své telefonní číslo pomocí kódu zasláního SMS, abych si mohl vytvořit účet.
- Jako uživatel chci zadat své PSČ a tím uvést, ze které země pocházím, abych mohl správně hlasovat.
- Jako uživatel chci vidět seznam politiků v mé zemi, abych mohl rozhodnout, koho podpořit.

PM zapisuje tyto požadavky do nástroje Jira, kde zároveň nastavuje jejich prioritu, odhaduje náročnost (například pomocí tzv. „story pointů”) a přiřazuje odpovědnosti členům týmu. Vývojáři se v této fázi aktivně účastní diskuzí, aby bylo možné správně odhadnout reálnou obtížnost jednotlivých úkolů.

Komunikace v týmu probíhá také pomocí nástroje Slack, kde si členové vyjasňují detaily k technickému řešení, architektuře či designu obrazovek. UI/UX designer mezitím připravuje první návrhy přihlašovacího procesu a obrazovek pro zadávání PSČ a seznam politiků, které následně sdílí s vývojáři. Tester začíná navrhovat testovací scénáře pro ověření základní funkčnosti (například validace telefonního čísla).

Plánování sprintu probíhá na tzv. „sprint planning meetingu”, který trvá přibližně 2–3 hodiny a na jehož konci má tým jasně definovaný seznam úkolů na nejbližší dvoutýdenní sprint. Po uzavření plánování je sprint zahájen a každý člen týmu přesně ví, na čem pracuje, jaký je účel jednotlivých úkolů a jakým způsobem bude jeho práce následně testována a nasazena.

17.5.3 Vývoj a implementace během sprintu

Po plánovací fázi se tým přesouvá k samotné realizaci úkolů naplánovaných pro aktuální sprint. Každý člen týmu si přebírá konkrétní user stories podle rozdělení z plánovacího setkání a začíná na nich pracovat.

Vývojáři vytvářejí pro každou funkci samostatné větve v GitHub repozitáři, aby oddělili svou práci a minimalizovali konflikty při slučování změn. Například při implementaci funkce anonymního přihlášení uživatele přes telefonní číslo vznikne větev s názvem „feature/phone-authentication”, kde vývojář postupně vytváří potřebné obrazovky

(zadání čísla, ověření kódu) a propojuje je s Firebase autentizací. V průběhu vývoje přidává commity, které jasně popisují jednotlivé části implementace.

Po dokončení funkce vývojář otevře pull request, čímž dává ostatním v týmu najevo, že jeho práce je připravena k revizi. Každý pull request je navázán na příslušnou položku v Jira, kde se průběžně aktualizuje stav user story. Pokud je potřeba, vývojář na základě zpětné vazby upraví kód a požádá o nové schválení.

Paralelně s tím UI designér připravuje návrhy dalších obrazovek, například nový způsob zobrazení seznamu politiků ve výběrové zemi. PM průběžně sleduje postup v Jira, konzultuje s vývojáři případné problémy a aktualizuje backlog podle priorit nebo zpětné vazby od stakeholdera.

Důležité je, že práce jednotlivých členů týmu probíhá paralelně, ale koordinovaně. Například tester může začít připravovat testovací scénáře ihned po dokončení prvních funkčních prototypů bez nutnosti čekat na uzavření celého sprintu. Výsledkem této fáze je funkční, integrovaná a částečně otestovaná verze aplikace, která je připravena na finální kolo interního testování a případné ladění v posledních dnech sprintu.

17.5.4 Testování v rámci sprintu

Během sprintu probíhá testování nově implementovaných funkcionalit paralelně s jejich vývojem. V týmu má tester na starosti ověřování, zda jednotlivé user stories splňují akceptační kritéria, ale také základní pokrytí automatizovanými testy tam, kde je to účelné.

V případě aplikace Votap by například po dokončení implementace funkcionality ověření telefonního čísla v rámci registrace probíhalo testování několika způsoby. Funkční testování by ověřilo, že po zadání validního telefonního čísla a následném zadání správného ověřovacího kódu dojde k úspěšnému vytvoření účtu a přesměrování uživatele na vyplnění PSČ. Pokud je zadáno neplatné číslo, aplikace musí korektně zobrazit chybovou hlášku. Zároveň vývojář připraví jednoduchý UI test pomocí Xcode, který simuluje kliknutí na tlačítko „Join Votap“, vyplnění telefonního čísla a ověření, že se objeví pole pro zadání kódu. Tyto testy pomáhají rychle odhalit chyby v navigaci nebo chybějící prvky.

Současně je View model použitý v procesu registrace pokryt unit testy, které testují například logiku validace telefonního čísla, správné nastavení stavů načítání nebo správné vyvolání autentizační služby a reakce na úspěch či chybu. Po úspěšném interním testování

je build aplikace pomocí CI/CD pipeline nahrán do TestFlight, kde probíhá další fáze testování na reálných zařízeních.

Organizace úkolů je řízena pomocí nástroje Jira, kde každá user story mění svůj stav podle průběhu: například „In Progress” → „Ready for Testing” → „Done”. Tester také využívá Slack pro rychlou komunikaci v případě nalezených chyb nebo nejasností.

Pro ilustraci lze uvést konkrétní případ testování funkce ověření telefonního čísla. User story by například zněla následujícím způsobem: „Jako nový uživatel chci být schopen ověřit své telefonní číslo, abych mohl vytvořit anonymní účet.” Pro ověření správné implementace byla vytvořena například tato unit testovací metoda:

```
func test_validatePhoneNumber_validNumber_returnsTrue() {  
    let viewModel = AuthenticationViewModel()  
    let result = viewModel.validatePhoneNumber(  
        "+420 123 456 789"  
    )  
    XCTAssertTrue(result)  
}
```

Obrázek 48 Unit test na základě user story

(zdroj: vlastní)

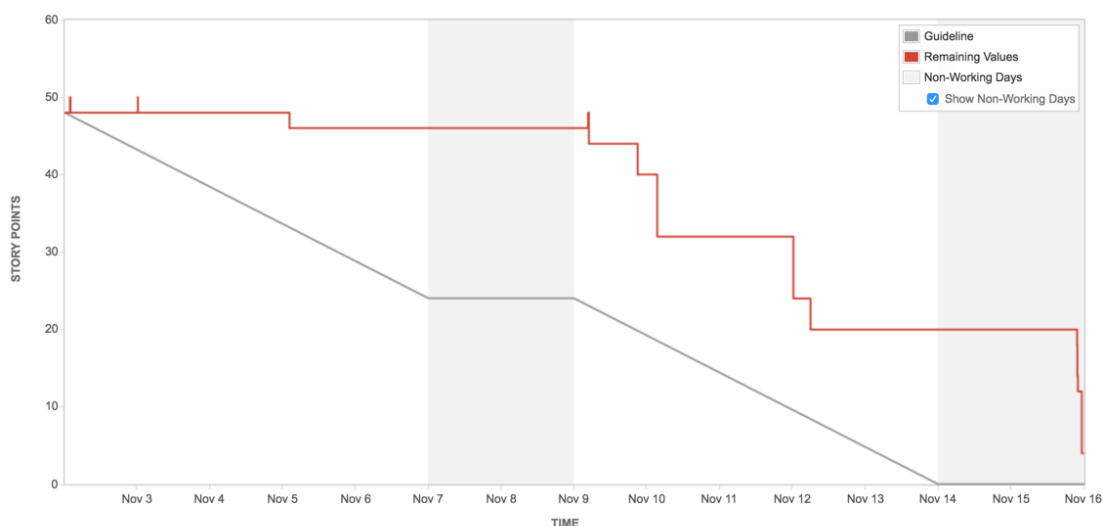
17.5.5 Průběžná komunikace a aktualizace úkolů

Během sprintu probíhá komunikace v týmu primárně přes Slack, kde má tým vytvořené specializované kanály pro různé oblasti práce – například kanál pro denní statusy, pro hlášení problémů, pro diskuze o návrzích designu či pro testování. Členové týmu každý den krátce reportují svůj pokrok, naráží-li na překážky, otevrou diskuzi s ostatními. Pokud například vývojář narazí při implementaci formuláře pro zadání telefonního čísla na problémy s validací formátu PSČ podle zadané země, ihned otevře diskuzi, aby tým UX návrhářů potvrdil, jaké přípustné formáty se mají akceptovat. Pokud problém nelze rychle vyřešit asynchronně, domluví se krátký operativní videohovor.

Jakmile se v průběhu sprintu objeví nutnost upravit existující úkol nebo přidat nový, aktualizuje se stav v Jira. Například tester zjistí, že po odeslání telefonního čísla nedojde v některých případech ke správnému přesměrování na ověřovací obrazovku. V takovém případě vytvoří nový ticket a popíše konkrétní kroky vedoucí k chybě, přiřadí ho odpovědnému vývojáři a označí jej jako „Bug“. Vývojář chybu opraví, vytvoří nový commit

s odpovídajícím komentářem (např. Fix: Phone number verification flow bug), otevře pull request a následně aktualizuje stav úkolu.

Postup sprintu tým se sleduje prostřednictvím tzv. „burndown“ grafu, který se automaticky generuje v Jira. Tento graf ukazuje, kolik práce (v počtu story points) zbývá do konce sprintu, a jak by se ideálně měla práce postupně snižovat. Pokud například sprint začíná s celkovým objemem 80 story points a po týdně je hotových pouze 20, graf vizuálně upozorní, že tempo plnění je nedostatečné. Díky tomu má product manager i celý tým včasné varování, že je třeba například přehodnotit priority, přerozdělit úkoly nebo upravit rozsah sprintu.



Obrázek 49 Ukázka burndown grafu

(zdroj: <https://confluence.atlassian.com/jirasoftwareserver/burndown-chart-938845620.html>)

17.5.6 Závěr sprintu a retrospektiva

Po uzavření sprintu se tým schází na retrospektivu, která probíhá formou online meetingu. Cílem retrospektivy je zjistit, co se během sprintu povedlo, co nefungovalo ideálně a jaké změny lze provést pro další iteraci. Vývojáři například zhodnotí, že příprava a správa pull requestů fungovala velmi dobře díky dodržování jasných pravidel. Na druhou stranu zjistí, že při implementaci formulářů došlo ke zpoždění kvůli nejasnostem v UX návrzích – tým se proto dohodne, že návrhy budou napříště schváleny s přesnějším zadáním ještě před zahájením vývoje.

Na závěr retrospektivy product manager shrne hlavní poznatky a vytvoří v akční body pro zlepšení. Například přidání povinného checklistu k pull requestům (kontrola názvu větve, přidání screenshotu k výsledku, potvrzení ručního testování) nebo zavedení nového kanálu na pro okamžité připomínky k designu. Všichni členové se tak aktivně podílí na neustálém zlepšování procesů a prostředí, což vede k vyšší kvalitě výstupů v dalších iteracích.

18 DISTRIBUCE A LEGISLATIVA

Publikace mobilní aplikace a její správa v reálném provozu vyžaduje nejen technickou přípravu a procesní nastavení, ale také dodržení legislativních požadavků na ochranu osobních údajů. Tato kapitola shrnuje proces nasazení aplikace Votap na App Store, práci s osobními údaji uživatelů, zejména v kontextu GDPR.

18.1 Proces nasazení aplikace na App Store

Pro úspěšné zveřejnění aplikace na App Store je nutné projít několika kroky, které zahrnují nejen technickou přípravu aplikace, ale i splnění požadavků společnosti Apple na obsah a kvalitu. Proces zahrnuje vytvoření správného buildu v Xcode, nastavení metadat v App Store Connect, nahrání aplikace, a následné schválení revizním týmem Apple. Tento proces je potřeba opakovat také při každé aktualizaci aplikace.

18.1.1 Nahrávání aplikace na App Store Connect

Před samotným nahráním aplikace je třeba splnit několik důležitých požadavků, mezi nimiž hrají klíčovou roli tzv. „Privacy Nutrition Labels”. Tyto informace slouží k transparentnímu informování uživatelů o tom, jaká data aplikace shromažďuje, k čemu jsou používána a zda jsou nějakým způsobem propojena s identitou uživatele. V moderním procesu publikace aplikací na App Store je vyžadováno, aby byly tyto údaje pravdivě a detailně uvedeny.

V našem případě jsou tyto informace definovány přímo v souboru „PrivacyInfo.plist”, který je součástí projektu v Xcode. Tento soubor obsahuje strukturovaná metadata popisující jednotlivé typy dat, například informace o kontaktech, poloze, identifikátorech zařízení, nebo jakým způsobem jsou data používána (například pro analytiku, personalizaci, nebo třetím stranám). U aplikace Votap jsme uvedli například sběr telefonního čísla a poštovního směrovacího čísla, přičemž další geografické údaje jsou odvozovány z veřejně dostupných databází.

Jakmile je aplikace připravena, je třeba vytvořit archivovaný build v Xcode. Nejprve je nutné v nastavení projektu upravit číslo verze (version) a číslo buildu (build number) tak, aby odpovídaly nové verzi aplikace. Správné nastavení certifikátu a profilu v sekci „Signing & Capabilities” zajišťuje, že aplikace bude digitálně podepsána a bude ji možné úspěšně odeslat.

Archivace samotná probíhá přes nabídku „Product-Archive”, což vytvoří soubor, který lze následně odeslat do App Store Connect.

Po odeslání se build objeví v App Store Connect, kde je nutné jej přiřadit k nové verzi aplikace. Zde také vyplňujeme metadata aplikace — název, popis, klíčová slova, podporované jazyky, screenshoty z aplikace na různých zařízeních a odkazy na zásady ochrany osobních údajů.

18.1.2 Schvalovací proces

Po nahrání buildu a přiřazení všech potřebných metadat aplikace nastává fáze schvalovacího procesu. Tento proces obvykle probíhá poměrně rychle — podle našich zkušeností bývá aplikace zkontrolována během přibližně 48 hodin. Je však potřeba počítat s tím, že první publikace nové aplikace trvá zpravidla déle než schvalování běžných aktualizací, které bývá výrazně rychlejší.

Dobrou strategií, zejména v případě dalších verzí, je nahrávat častější a menší aktualizace. Ty totiž obvykle vyžadují méně důkladné posouzení, což může celý schvalovací proces urychlit. Všechny aplikace a jejich změny jsou hodnoceny podle oficiálních pravidel App Store Review Guidelines, která definují, co je v aplikaci povoleno a co je považováno za nepřípustné chování.

V případě zamítnutí aplikace obdrží vývojář zprávu s konkrétní specifikací, která část pravidel byla porušena. Je pak nutné provést odpovídající úpravy v aplikaci, vytvořit nový build a celý proces odeslání ke schválení opakovat. Apple také nabízí možnost v případě nejasností rezervovat si termín videohovoru se zástupcem schvalovacího týmu přímo v App Store Connect. Tento krok může výrazně urychlit proces objasnění problémů a předejít opakovanému zamítnutí. Apple také nabízí možnost v případě nejasností rezervovat si termín videohovoru se zástupcem schvalovacího týmu přímo v App Store Connect. Tento krok může výrazně urychlit proces objasnění problémů a předejít opakovanému zamítnutí.

18.1.3 Vydávání aktualizací

Po úspěšném schválení a zveřejnění první verze aplikace na App Store přichází na řadu pravidelné vydávání aktualizací. Aktualizace umožňují nejen opravovat chyby a zvyšovat stabilitu aplikace, ale také postupně přidávat nové funkce či vylepšovat uživatelský zážitek.

Proces vydávání aktualizací probíhá velmi podobně jako u první verze – každá nová verze musí mít vlastní unikátní číslo verze a build number. V Xcode je nutné tyto hodnoty

před nahráním aktualizace správně nastavit. Aktualizovaný build je poté opět nahrán přes App Store Connect, kde je potřeba doplnit nové informace, jako jsou změny v popisu aplikace nebo novinky v rámci aktualizace (release notes).

18.2 Práce s osobními údaji a ochrana soukromí

Ochrana osobních údajů je při vývoji a distribuci mobilních aplikací klíčovým tématem. V rámci aplikace Votap je kladen důraz na transparentní zpracování údajů, minimalizaci sběru dat a respektování práv uživatelů v souladu s platnou legislativou, zejména GDPR. V následujících částech bude popsáno, jak aplikace pracuje s citlivými informacemi, jaké mechanismy ochrany jsou implementovány a jaká opatření byla přijata pro splnění legislativních požadavků.

18.2.1 Shromažďovaná data

Aplikace Votap pracuje pouze s nejnutnějšími osobními údaji, konkrétně telefonním číslem a poštovním směrovacím číslem (PSC), které slouží k ověření identity a geografickému zařazení uživatele. Neuchovávají se žádné konkrétní adresy ani jiné citlivé informace — PSC je využito k určení oblasti pro správné přiřazení hlasů. Zároveň jsou uchovávány anonymní záznamy o tom, kdy a jak uživatel hlasoval, což umožňuje analyzovat vývoj preferencí v čase. Tento přístup zachovává anonymitu, ale poskytuje dostatek informací pro fungování aplikace.

18.2.2 Smazání účtu a všech osobních údajů

Aplikace Votap umožňuje uživatelům kdykoli trvale smazat svůj účet včetně všech osobních údajů. Tuto možnost lze využít přímo v nastavení aplikace, kde je k dispozici funkce pro odstranění účtu. Při potvrzení požadavku na smazání je uživatelský účet odstraněn z autentizační služby Firebase (včetně telefonního čísla) a zároveň jsou odstraněna všechna související data uložená v databázi, například informace o hlasování a uložené preference.

Tento přístup je v souladu s požadavky na ochranu osobních údajů a umožňuje uživatelům plně uplatnit své právo být zapomenut. Proces smazání probíhá automatizovaně bez nutnosti kontaktovat podporu, čímž se zvyšuje komfort a bezpečí uživatelů.

18.2.3 Požadavky GDPR

Aby aplikace Votap splňovala požadavky vyplývající z GDPR, je nutné dbát na několik zásadních oblastí, které se týkají sběru, uchovávání a zpracování osobních údajů.

Prvním klíčovým požadavkem je informování uživatele o tom, jaká data jsou sbírána a jak s nimi bude nakládáno. V případě aplikace Votap je tato povinnost naplněna prostřednictvím zveřejněné „Privacy Policy”, která je dostupná jak přímo v aplikaci, tak na stránce aplikace v App Store. Dále je aplikace navržena tak, aby plnila povinnost umožnit uživateli smazat všechna osobní data – takzvané právo být zapomenut.

S ohledem na budoucí vývoj je potřeba počítat také s povinností umožnit uživateli exportovat své osobní údaje – takzvané právo na přístup k údajům. V současné verzi aplikace tato funkce ještě není implementována, ale do budoucna je plánováno přidat možnost, aby si uživatel mohl jedním kliknutím vygenerovat a stáhnout soubor s informacemi, které o něm systém eviduje.

Aplikace rovněž využívá nástroje pro analytiku, konkrétně Firebase Analytics, což přináší povinnost transparentně informovat uživatele o tom, že jejich chování v aplikaci může být sledováno za účelem zlepšení služeb a analýzy chování uživatelů. Tato skutečnost je opět uvedena v Privacy Policy, kde je uživatelům vysvětlen účel sledování i to, že data jsou agregována a anonymizována. Dodržování těchto principů je klíčové pro udržení důvěry uživatelů a pro minimalizaci rizika regulatorních postihů při působení na trzích Evropské unie.

19 SHRnutí

V praktické části práce byla podrobně popsána architektura, struktura, metodologie vývoje a návrh týmové spolupráce při vývoji aplikace Votap. Byla představena nejen samotná implementace klíčových principů jako MVVM, SOLID či clean code, ale také konkrétní nástroje a procesy využívané při správě verzí, testování a agilním řízení vývoje.

V rámci hypotetického návrhu týmové spolupráce byl podrobně rozpracován celý první sprint včetně příkladů konkrétních úkolů, nástrojů, které by členové týmu využívali, a možných problémů, které by bylo potřeba v průběhu řešit. Všechny návrhy vycházely z reálných potřeb a specifikací samotné aplikace a z praktických zkušeností získaných při jejím samostatném vývoji.

19.1 Zhodnocení přístupu

Zvolený přístup, který kombinuje moderní architektonické principy s důrazem na čitelný a testovatelný kód, se ukázal jako velmi efektivní pro vývoj aplikace. Použití verzovacího systému Git, správné rozdělení odpovědností ve vrstvě MVVM, důsledné testování a návrh procesů zaměřených na spolupráci v týmu umožňuje projekt snadno rozšiřovat, udržovat a adaptovat na nové požadavky. Důležité je také zmínit, že otevřená komunikace v rámci týmu a důsledné plánování výrazně přispívají k vyšší kvalitě výsledného produktu a ke spokojenosti všech zúčastněných.

19.2 Možnosti rozšíření aplikace

Aplikace Votap má velký potenciál k dalšímu rozvoji. V budoucích verzích by bylo možné zavést například možnost hlasování o aktuálních událostech, konkrétních krocích politiků nebo vybraných tématech (například nové zákony, vládní reformy apod.).

Další přirozenou oblastí rozšíření je zahrnutí politických stran a umožnění uživatelům hodnotit nejen jednotlivé politiky, ale i celé subjekty jako celek. Zajímavou funkcionalitou by také mohlo být zavedení sociální vrstvy – například sledování přátel, zobrazování anonymizovaných názorů ostatních uživatelů v rámci dané země, nebo diskusní fóra pro výměnu názorů. Technologicky by bylo možné rozšířit aplikaci například o chytrá upozornění (push notifikace) při důležitých změnách v hodnocení politiků, nebo o sekci s personalizovanými novinkami na základě oblíbených politiků a témat.

19.3 Doporučení pro týmový vývoj

Při práci na podobných projektech je klíčové zajistit, aby v týmu panovala otevřená a upřímná komunikace. Členové týmu by měli mít možnost bez obav diskutovat i o zdánlivě drobných problémech a nedorozuměních. I malé procesní nedostatky, pokud nejsou řešeny včas, mohou v pozdějších fázích vývoje přerůst v zásadní komplikace a zpomalit celý projekt.

Důležitou součástí týmové práce je také pravidelné poskytování zpětné vazby. Ať už při retrospektivách na konci sprintu, nebo průběžně během každodenní spolupráce, otevřená komunikace přispívá k rychlému odstraňování překážek a posilování důvěry v týmu. Zároveň je vhodné věnovat pozornost i dokumentaci – nejen ke kódu, ale také k procesům a dohodám uvnitř týmu. Udržování aktuální a přehledné dokumentace usnadňuje adaptaci nových členů a zjednodušuje údržbu aplikace v delším časovém horizontu.

Zkušenosti ukazují, že úspěšné týmy neustále reflektují a upravují své pracovní postupy na základě reálných zkušeností z vývoje. Pravidelné hodnocení a vylepšování vnitřních procesů pomáhá předcházet opakování chyb a přispívá k efektivnějšímu a udržitelnějšímu rozvoji aplikace. Kromě technických dovedností je proto vhodné rozvíjet také měkké dovednosti, jako je týmová spolupráce, schopnost efektivní komunikace a řešení konfliktů. Dodržováním těchto zásad lze výrazně zvýšit efektivitu vývoje, zlepšit kvalitu výsledného produktu a zajistit, že práce na aplikaci bude dlouhodobě udržitelná a příjemná pro všechny členy týmu.

ZÁVĚR

Tato práce se zaměřila na moderní metodologie a principy používané při vývoji mobilních aplikací se zaměřením na SwiftUI. V teoretické části byly popsány klíčové technologie, architektury, principy čistého kódu, správného testování, verzování a dokumentace, které přispívají k udržitelnému a efektivnímu vývoji.

V praktické části byla představena aplikace Votap, reálný projekt publikovaný na App Store, který slouží jako ukázka aplikace těchto principů v praxi. Byla detailně popsána její architektura postavená na vzoru MVVM, způsob práce s daty pomocí repozitářů a service tříd, použití asynchronních metod, základy testování a příprava aplikace pro produkční prostředí včetně publikace. Součástí práce byl také návrh agilního vývoje v týmovém prostředí, včetně rozdělení rolí, práce ve sprintech, strategie verzování, použití nástrojů pro týmovou spolupráci a příklad konkrétního průběhu iterace.

Výsledkem je komplexní shrnutí teoretických základů i praktických postupů, které mohou sloužit jako vodítko pro jednotlivce i menší týmy při vývoji moderních mobilních aplikací. Práce zároveň ukazuje, jak důležité je propojení teorie s praxí a neustálé přizpůsobování procesů konkrétním projektovým potřebám.

SEZNAM POUŽITÉ LITERATURY

- [1] Swift – Apple Developer. Online. Apple. C2025. Dostupné z: <https://developer.apple.com/swift/>. [cit. 2025-05-06].
- [2] Swift Programming Language. Online. GeeksforGeeks. 2023. Dostupné z: <https://www.geeksforgeeks.org/swift-programming-language/>. [cit. 2025-05-06].
- [3] POPKO, Aleksander. Objective-C vs Swift: iOS Comparison. Online. Netguru. 2024. Dostupné z: <https://www.netguru.com/blog/objective-c-vs-swift>. [cit. 2025-05-06].
- [4] JALLI, Arturri. What Is Objective C? Online. Builtin. 2022. Dostupné z: <https://builtin.com/software-engineering-perspectives/objective-c>. [cit. 2025-05-06].
- [5] What is UIKit and what is it for? Online. Swhosting. 2024. Dostupné z: <https://www.swhosting.com/en/blog/what-is-uikit-and-what-is-it-for>. [cit. 2025-05-06].
- [6] L, Jeroen. UIKit vs. SwiftUI: How to Choose the Right Framework for Your App. Online. Getstream. 2023. Dostupné z: <https://getstream.io/blog/uikit-vs-swiftui/>. [cit. 2025-05-06].
- [7] What is SwiftUI? Online. HowDev. C2025. Dostupné z: <https://how.dev/answers/whatis-swiftui>. [cit. 2025-05-06].
- [8] KURINNA, Olha. What is Xcode? Everything you need to know. Online. Apptension. 2023. Dostupné z: <https://www.apptension.com/blog-posts/what-is-xcode>. [cit. 2025-05-06].
- [9] Become a member – Apple Developer Program. Online. Apple. C2025. Dostupné z: <https://developer.apple.com/programs/enroll/>. [cit. 2025-05-06].
- [10] ALSHAREEF, Amr. How to Create an Apple Developer Account and Enroll in the Program. Online. Buildfire. 2025. Dostupné z: <https://learn.buildfire.com/en/articles/477266-how-to-create-an-apple-developer-account-and-enroll-in-the-program>. [cit. 2025-05-06].
- [11] Complete Guide to MVVM in SwiftUI. Online. Swift Anytime. 2024. Dostupné z: <https://www.swiftanytime.com/blog/mvvm-in-swiftui>. [cit. 2025-05-06].

- [12] Implement MVVM Architecture in SwiftUI. Online. Kodeco. C2025. Dostupné z: <https://www.kodeco.com/books/swiftui-cookbook/v1.0/chapters/10-implement-mvvmarchitecture-in-swiftui>. [cit. 2025-05-06].
- [13] OLORUNTOBA, Samuel a WALIA, Anish Singh. SOLID: The First 5 Principles of Object Oriented Design. Online. DigitalOcean. 2024. Dostupné z: <https://www.digitalocean.com/community/conceptual-articles/s-o-l-i-d-the-first-five-principlesof-object-oriented-design> <https://www.baeldung.com/solid-principles>. [cit. 2025-05-06].
- [14] MILLINGTON, Sam. A Solid Guide to SOLID Principles. Online. Baeldung. 2025. Dostupné z: <https://www.baeldung.com/solid-principles>. [cit. 2025-05-06].
- [15] SATRASALA, Esh. Writing Clean Code With Swift — 7 Essential Tips. Online. Medium. 2019. Dostupné z: <https://medium.com/@EshwarSatrasala/writing-clean-codewith-swift-7-essential-tips-7e18d9ca5590>. [cit. 2025-05-06].
- [16] What Is Clean Code? A Guide to Principles and Best Practices. Online. Codacy. 2023. Dostupné z: <https://blog.codacy.com/what-is-clean-code>. [cit. 2025-05-06].
- [17] ROBINSON, Scott; BRUSH, Kate a SILVERTHORNE, Valerie. What is Agile software development? Online. TechTarget. 2024. Dostupné z: <https://www.techtarget.com/searchsoftwarequality/definition/agile-software-development>. [cit. 2025-05-06].
- [18] What is version control? Online. GitLab. C2025. Dostupné z: <https://about.gitlab.com/topics/version-control/>. [cit. 2025-05-06].
- [19] An Intro to Git and GitHub for Beginners. Online. HubSpot. 2020. Dostupné z: <https://product.hubspot.com/blog/git-and-github-tutorial-for-beginners>. [cit. 2025-05-06].
- [20] What Is GitHub and Why Should You Use It? Online. Coursera. 2025. Dostupné z: <https://www.coursera.org/articles/what-is-git>. [cit. 2025-05-06].
- [21] Gitflow workflow. Online. Atlassian. C2025. Dostupné z: <https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow>. [cit. 2025-05-06].

- [22] About merge methods on GitHub. Online. GitHub. C2025. Dostupné z: <https://docs.github.com/en/repositories/configuring-branches-and-merges-in-your-repository/configuring-pull-request-merges/about-merge-methods-on-github>. [cit. 2025-05-06].
- [23] What is Software Testing? Definition, Types, and Tools. Online. Katalon. 2025. Dostupné z: <https://katalon.com/resources-center/blog/software-testing>. [cit. 2025-05-06].
- [24] BERGA, Kristina. Software Testing 101: Definition, Types & Everything Else. Online. TestDevLab. 2024. Dostupné z: <https://www.testdevlab.com/blog/softwaretesting-101-definition-types-everything-else>. [cit. 2025-05-06].
- [25] XCTest – Apple Developer Documentation. Online. Apple. C2025. Dostupné z: <https://developer.apple.com/documentation/xctest>. [cit. 2025-05-06].
- [26] TestFlight - Apple Developer. Online. Apple. C2025. Dostupné z: <https://developer.apple.com/testflight/>. [cit. 2025-05-06].
- [27] Code documentation: Benefits, challenges, and tips for success. Online. Swimm. C2025. Dostupné z: <https://swimm.io/learn/code-documentation/code-documentationbenefits-challenges-and-tips-for-success>. [cit. 2025-05-06].
- [28] Documenting apps, frameworks, and packages – Apple Developer Documentation. Online. Apple. C2025. Dostupné z: <https://developer.apple.com/documentation/xcode/documenting-apps-frameworks-and-packages>. [cit. 2025-05-06].
- [29] DocC. Online. Swift. C2025. Dostupné z: <https://www.swift.org/documentation/docc/>. [cit. 2025-05-06].
- [30] App Review Guidelines - Apple Developer. Online. Apple. C2025. Dostupné z: <https://developer.apple.com/app-store/review/guidelines/>. [cit. 2025-05-06].
- [31] App Privacy Details – Apple Developer. Online. Apple. C2025. Dostupné z: <https://developer.apple.com/app-store/app-privacy-details/>. [cit. 2025-05-06].
- [32] BATEMAN, Robert. GDPR and Mobile Apps. Online. TermsFeed. 2025. Dostupné z: <https://www.termsfeed.com/blog/gdpr-mobile-apps/>. [cit. 2025-05-06].

SEZNAM OBRÁZKŮ

Obrázek 1 Logo aplikace Votap.....	47
Obrázek 2 Přihlašování v aplikaci Votap.....	47
Obrázek 3 Portfolio v aplikaci Votap	48
Obrázek 4 Žebříčky v aplikaci Votap	49
Obrázek 5 Profil politika v aplikaci Votap	49
Obrázek 6 Implementace vrstvy model	50
Obrázek 7 Implementace vrstvy View.....	51
Obrázek 8 Implementace vrstvy View model.....	53
Obrázek 9 Service třída pro databázi Firestore	54
Obrázek 10 Datový repozitář pro data o politicích	55
Obrázek 11 Dependency injection v konstruktoru View modelu	56
Obrázek 12 Deklarace znovupoužitelné View komponenty	56
Obrázek 13 Použití znovupoužitelné View komponenty.....	57
Obrázek 14 Rozšíření typu String pro podporu systému Regex	57
Obrázek 15 Rozšíření pro podporu rozhraní Equatable.....	58
Obrázek 16 Definice vlastních barev v Xcode.....	58
Obrázek 17 Data o USA ze souboru countries.json.....	59
Obrázek 18 Aplikace principu jedné odpovědnosti	60
Obrázek 19 Aplikace principu otevřenosti a uzavřenosti	61
Obrázek 20 Aplikace dědičnosti v jazyce Swift	63
Obrázek 21 Aplikace Liskovového principu substituce	64
Obrázek 22 Aplikace principu oddělení rozhraní	64
Obrázek 23 Aplikace principu obrácení závislostí	65
Obrázek 24 Použití pomocné metody v repozitáři.....	66
Obrázek 25 Příklad správného použití komentářů v kódu.....	67
Obrázek 26 Příklad chybného použití komentářů v kódu.....	68
Obrázek 27 Ukázka kódu před refaktoringem	69
Obrázek 28 Ukázka kódu po refaktoringu	70
Obrázek 29 Celková souborová struktura aplikace Votap.....	71
Obrázek 30 Souborová struktura MVVM v aplikaci Votap	72
Obrázek 31 Zbytek souborové struktury v aplikaci Votap	73
Obrázek 32 Použití preprocesorových maker	76
Obrázek 33 Ukázka nástroje SwiftLint.....	77
Obrázek 34 Ukázka dokumentačních komentářů	79

Obrázek 35 Ukázka výsledné dokumentace	80
Obrázek 36 Klonování repozitáře v Xcode.....	82
Obrázek 37 Přihlášení k účtu GitHub v Xcode.....	83
Obrázek 38 Source Control panel v Xcode	84
Obrázek 39 Pull request v platformě GitHub	85
Obrázek 40 Struktura třídy pro unit testování v XCTest.....	87
Obrázek 41 Použití rozhraní ve View modelu.....	88
Obrázek 42 Příklad unit testu pro View model.....	89
Obrázek 43 Unit test s asynchronní metodou	89
Obrázek 44 Identifikátory pro UI testy ve SwiftUI	90
Obrázek 45 Příklad UI testu.....	91
Obrázek 46 Návrh black-box testů	91
Obrázek 47 Distribuce aplikace na App Store Connect	93
Obrázek 48 Unit test na základě user story.....	102
Obrázek 49 Ukázka burndown grafu	103

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

API	Programové rozhraní pro komunikaci mezi aplikacemi (Application Programming Interface)
ARC	Automatická správa paměti v jazyce Swift (Automatic Reference Counting)
CI/CD	Automatizace sestavení, testování a nasazení (Continuous Integration/Continuous Deployment)
CRUD	Základní operace s daty: vytvoření, čtení, úprava, smazání (Create, Read, Update, Delete)
DI	Vkládání závislostí – způsob, jak objektům poskytovat jejich závislosti (Dependency Injection)
DIP	Princip obrácení závislostí – moduly by měly záviset na abstrakcích, ne konkrétních třídách (Dependency Inversion Principle)
DocC	Systém pro generování vývojářské dokumentace v prostředí Xcode
DRY	Zásada, že by se žádná informace neměla v kódu opakovat (Don't Repeat Yourself)
DSDM	Agilní metodika pro vývoj softwaru zaměřená na rychlé dodání (Dynamic Systems Development Method)
DTO	Objekt sloužící pro přenos dat mezi vrstvami systému (Data Transfer Object)
FDD	Vývoj řízený funkcionalitou – agilní metodika zaměřená na konkrétní funkce (Feature-Driven Development)
GDPR	Obecné nařízení EU o ochraně osobních údajů (General Data Protection Regulation)
Git	Distribuovaný systém pro správu verzí zdrojového kódu
IDE	Integrované vývojové prostředí (např. Xcode)
iOS	Mobilní operační systém společnosti Apple

ISP	Princip oddělení rozhraní – rozhraní by měla být malá a specifická (Interface Segregation Principle)
JSON	Formát pro výměnu dat čitelný pro člověka i stroj (JavaScript Object Notation)
LSP	Liskovův princip zaměnitelnosti – podtřídy musí být plně zaměnitelné za nadřazené třídy (Liskov Substitution Principle)
MVC	Architektonický návrhový vzor Model–View–Controller
MVVM	Architektonický návrhový vzor Model–View–ViewModel
MVP	Nejmenší životaschopný produkt – první funkční verze s minimem klíčových funkcí (Minimum Viable Product)
OCP	Princip otevřenosti a uzavřenosti – třídy by měly být otevřené pro rozšíření, ale uzavřené pro modifikaci (Open/Closed Principle)
PM	Produktový manažer – osoba zodpovědná za řízení vývoje produktu
PR	Návrh na sloučení změn v Git repozitáři, často spojený s code review (pull request)
SDK	Sada nástrojů pro vývoj softwaru (Software Development Kit)
SOLID	Soubor pěti zásad pro návrh objektově orientovaného softwaru
SRP	Princip jediné odpovědnosti – každá třída by měla mít pouze jeden důvod ke změně (Single Responsibility Principle)
SwiftUI	Deklarativní framework od společnosti Apple pro tvorbu uživatelského rozhraní
TDD	Vývoj řízený testy – testy se píšou ještě před samotnou implementací (Test-Driven Development)
UAT	Uživatelské akceptační testování – fáze, kdy koncoví uživatelé ověřují, zda aplikace splňuje požadavky

UI	Uživatelské rozhraní (User Interface)
UIKit	Framework společnosti Apple pro tvorbu uživatelského rozhraní pomocí imperativního přístupu
URL	Adresa zdroje na internetu (Uniform Resource Locator)
UX	Celkový dojem uživatele při práci s aplikací (User Experience)
XP	Extrémní programování – agilní vývojová metodika zaměřená na kvalitu a přizpůsobivost změnám