

Vývoj iOS aplikace pro odhad porozity vzorků pneumatik

Bc. Dominik Vanduch

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. Dominik Vanduch
Osobní číslo: A23417
Studijní program: N0613A140022 Informační technologie
Specializace: Kybernetická bezpečnost
Forma studia: Kombinovaná
Téma práce: Vývoj iOS aplikace pro odhad porozity vzorků pneumatik
Téma práce anglicky: Development of an iOS Application for Estimating Tire Sample Porosity

Zásady pro vypracování

- Popište základní principy a problematiku vývoje pro mobilní platformy od společnosti Apple, včetně nástrojů pro vývoj aplikací využívajících zabudovanou kameru zařízení a zpracování pořízených snímků.
- Vyhledejte a porovnejte aplikace řešící obdobné případy užití jako jsou měření fyzikálních vlastností pomocí mobilních aplikací a použití mobilních aplikací pro kontrolu kvality ve výrobě.
- Sestavte funkcionální a nefunkcionální požadavky na aplikaci a vytvořte návrh aplikace.
- Implementujte aplikaci a popište její klíčové části.
- Otestujte vytvořenou aplikaci a zhodnoťte možnosti využití mobilních zařízení Apple pro odhad porozity vzorků pneumatik.

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

Seznam doporučené literatury:

1. MARQUES, Oge. Image Processing and Computer Vision in iOS. Springer, 2020. ISBN 978-3-030-54030-2.
2. Apple Developer Documentation. Apple Developer [online]. Apple, 2024 [cit. 2024-10-12]. Dostupné z: <https://developer.apple.com/documentation/>
3. Swift. Documentation [online]. Apple, 2024 [cit. 2024-10-12]. Dostupné z: <https://www.swift.org/documentation/>
4. SwiftUI. Apple Developer [online]. Apple, 2024 [cit. 2024-10-12]. Dostupné z: <https://developer.apple.com/documentation/swiftui/>
5. Vision. Apple Developer [online]. Apple, 2024 [cit. 2024-10-12]. Dostupné z: <https://developer.apple.com/documentation/vision/>
6. Core ML. Apple Developer [online]. Apple, 2024 [cit. 2024-10-12]. Dostupné z: <https://developer.apple.com/documentation/coreml/>
7. RAYWENDERLICH TUTORIAL TEAM, Alexis GALLAGHER, Matthijs HOLLEMANS, Audrey TAM a Chris LAPOLLO. Machine Learning by Tutorials (Second Edition): Beginning Machine Learning for Apple and iOS. 2nd ed. Razeware, 2021. ISBN 978-1942878933.
8. SHARMA, Mukesh. IOS Development with SwiftUI. BPB Publications, 2022. ISBN 939103098X.

Vedoucí diplomové práce: **Ing. Radek Vala, 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;
- při tvorbě této práce jsem použil nástroj generativního modelu AI Claude Sonnet (<https://claude.ai>) za účelem jazykové úpravy textu, zlepšení stylistiky a vyhledávání zdrojů. Po použití tohoto nástroje jsem provedl kontrolu obsahu a přebírám za něj plnou odpovědnost.

Ve Zlíně, dne

.....
podpis autora

ABSTRAKT

Diplomová práce se zabývá vývojem mobilní aplikace pro objektivní odhad porozity vzorků pneumatik využívající platformu iOS. V teoretické části práce jsou popsány mobilní platformy společnosti Apple, principy vývoje aplikací pro operační systém iOS, nástroje a knihovny používané pro vývoj a problematika porozity materiálů. Praktická část zahrnuje analýzu existujících mobilních aplikací, specifikaci prototypové aplikace, její návrh, implementaci, validační testování a zhodnocení vytvořeného prototypu. Vytvořená aplikace využívá algoritmy k analýze obrazu za účelem automatické detekce a kvantifikace porozity z fotografií řezů vzorků pneumatik. Součástí práce je zhodnocení dosažených výsledků, identifikace limitů řešení a možných budoucích směrů rozvoje aplikací využívaných v oblasti automatizované detekce a kvantifikace porozity materiálů pomocí mobilních technologií.

Klíčová slova: mobilní technologie, iOS, porozita, zpracování obrazu, digitalizace

ABSTRACT

This master's thesis focuses on the development of a mobile application for the objective estimation of porosity in tire samples using the iOS platform. The theoretical part describes Apple's mobile platforms, principles of application development for iOS, tools and libraries used in the development process, and the concept of material porosity. The practical section encompasses the analysis of existing mobile applications, prototype application specification, architectural design, implementation, validation testing, and evaluation of the developed prototype. The developed application employs image analysis algorithms for automatic detection and quantification of porosity from photographs of tire sample cross-sections. The thesis includes assessment of achieved results, identification of solution limitations, and exploration of potential future development directions for applications employed in automated detection and quantification of material porosity using mobile technologies.

Keywords: mobile technology, iOS, porosity, image processing, digitization

Rád bych vyjádřil své poděkování vedoucímu této diplomové práce panu Ing. Radku Valovi, Ph.D. za odborné vedení, věnovaný čas a cenné rady během celého procesu vzniku této práce.

Zvláštní poděkování náleží prof. Jorge Lacayo-Pinedovi za jeho podporu, expertní konzultace a motivaci při realizaci tohoto projektu.

V neposlední řadě děkuji všem osobám z mého okolí, které mě během této etapy studia podporovaly a povzbuzovaly.

OBSAH

ÚVOD.....	12
I TEORETICKÁ ČÁST	13
1 MOBILNÍ PLATFORMY SPOLEČNOSTI APPLE	14
1.1 IOS	14
1.2 IPADOS.....	14
1.3 WATCHOS	15
1.4 PROPOJENÍ PLATFORMEM	15
1.5 APP STORE.....	16
2 MOŽNOSTI VÝVOJE APLIKACÍ PRO MOBILNÍ PLATFORMY.....	17
2.1 NATIVNÍ VÝVOJ.....	17
2.2 MULTIPLATFORMNÍ VÝVOJ	18
2.3 HYBRIDNÍ VÝVOJ.....	19
2.4 SHRUTÍ MOŽNOSTÍ VÝVOJE	21
3 VÝVOJ APLIKACÍ PRO PLATFORMU IOS.....	22
3.1 SWIFT	22
3.1.1 HELLO, WORLD!	23
3.1.2 PROMĚNNÉ A KONSTANTY	24
3.1.3 ŘÍZENÍ TOKU PROGRAMU	25
3.1.4 FUNKCE.....	26
3.1.5 OBJEKTY A TŘÍDY	27
3.1.6 VÝČTOVÉ TYPY A STRUKTURY	28
3.1.7 ASYNCHRONNÍ PROGRAMOVÁNÍ	29
3.1.8 PROTOKOLY A ROZŠÍŘENÍ.....	30
3.1.9 ZPRACOVÁNÍ CHYB.....	31
3.1.10 GENERIKA.....	32
3.2 SWIFTUI.....	33
3.2.1 KOMPOZICE.....	33
3.2.2 STAV A DATOVÝ TOK	34
3.2.3 MODIFIKÁTORY	36
3.2.4 NAVIGACE.....	37
3.2.5 VZTAH SWIFTUI S FRAMEWORKEM UIKIT	38
3.3 FOUNDATION.....	39
3.4 SWIFTDATA.....	39
3.4.1 KONTEJNERY A KONTEXT	40
3.4.2 DATOVÝ MODEL.....	40
3.4.3 VZTAHY	41
3.4.4 PRÁCE S DATY.....	42
3.4.5 SROVNÁNÍ S CORE DATA	42
3.5 CORE IMAGE	43
3.5.1 CICONTEXT	43
3.5.2 CIIMAGE.....	44
3.5.3 CIFILTER	44
3.6 XCODE.....	44
4 POROZITA	46

II PRAKTICKÁ ČÁST	48
5 ANALÝZA EXISTUJÍCÍCH MOBILNÍCH APLIKACÍ.....	49
5.1 MOBILNÍ APLIKACE PRO MĚŘENÍ FYZIKÁLNÍCH VLASTNOSTÍ.....	49
5.2 MOBILNÍ APLIKACE PRO KONTROLU KVALITY VE VÝROBĚ	50
5.3 MOBILNÍ APLIKACE PRO MĚŘENÍ POROZITY MATERIÁLŮ	51
5.4 SHRUTÍ ANALÝZY EXISTUJÍCÍCH ŘEŠENÍ.....	51
6 SPECIFIKACE APLIKACE	52
6.1 CÍLE APLIKACE.....	52
6.2 POPIS APLIKACE.....	53
6.3 FUNKCIONÁLNÍ POŽADAVKY	54
6.4 NEFUNKCIONÁLNÍ POŽADAVKY	55
7 NÁVRH APLIKACE.....	57
7.1 POSTUP A METODIKY	57
7.2 NÁVRH DRÁTĚNÝCH MODELŮ	57
7.2.1 HLAVNÍ OBRAZOVKY ŘÍZENÉ STRUKTUROU TAB VIEW	58
7.2.2 VOLBA ZDROJE VSTUPNÍ FOTOGRAFIE	59
7.2.3 PŘIDÁNÍ ZÁZNAMU	60
7.2.4 VYTVOŘENÍ NOVÉ SMĚSI NEBO VÝROBNÍ LINKY	62
7.2.5 OBRAZOVKA DETAILU ZÁZNAMU	62
7.3 NÁVRH DATOVÉHO MODELU	64
7.3.1 RECORD.....	65
7.3.2 COMPOUND	65
7.3.3 EXTRUSION LINE	66
7.3.4 VZTAHY MEZI ENTITAMI	66
8 VÝVOJ APLIKACE.....	67
8.1 POSTUPY A METODIKY	67
8.2 VYTVOŘENÍ PROJEKTU	67
8.3 KONFIGURACE APLIKACE.....	70
8.4 ARCHITEKTURA APLIKACE A STRUKTURA PROJEKTU	72
8.5 DEFINICE DATOVÉHO MODELU A PERZISTENCE DAT	74
8.5.1 INICIALIZACE DATOVÉHO KONTEJNERU	74
8.5.2 MODEL ZÁZNAMU ANALÝZY	75
8.5.3 MODEL SMĚSI MATERIÁLU	77
8.5.4 MODEL VÝROBNÍ LINKY	78
8.5.5 VZTAHY MEZI ENTITAMI	78
8.6 VYTVOŘENÉ UŽIVATELSKÉ ROZHRAŇÍ.....	78
8.6.1 HLAVNÍ NAVIGACE.....	79
8.6.2 DOMOVSKÁ OBRAZOVKA.....	80
8.6.3 OBRAZOVKY PRO NAHRÁNÍ VSTUPNÍ FOTOGRAFIE.....	81
8.6.4 VYTVOŘENÍ ZÁZNAMU O MĚŘENÍ POROZITY	83
8.6.5 PŘIDÁNÍ DOPLŇUJÍCÍCH INFORMACÍ K ZÁZNAMU.....	84
8.6.6 DETAIL ZÁZNAMU A SDÍLENÍ VÝSLEDKŮ MĚŘENÍ	85
8.7 KLÍČOVÉ ČÁSTI IMPLEMENTACE.....	86
8.7.1 VÝBĚR OBLASTI Z FOTOGRAFIE K ANALÝZE POROZITY.....	87
8.7.2 ANALÝZA VYBRANÉ OBLASTI A VÝPOČET POROZITY	92
8.7.3 EXPORT ZÁZNAMŮ DO FORMÁTU CSV A JEJICH SDÍLENÍ	95

9	VALIDAČNÍ TESTOVÁNÍ APLIKACE.....	99
9.1	METODIKA TESTOVÁNÍ.....	99
9.2	VÝSLEDKY TESTOVÁNÍ.....	100
9.3	SHRNUTÍ TESTOVÁNÍ.....	102
10	ZHODNOCENÍ VYTVOŘENÉ APLIKACE	107
10.1	PŘEDNOSTI VYTVOŘENÉ APLIKACE	107
10.2	LIMITY VYTVOŘENÉ APLIKACE	107
10.3	MOŽNÉ SMĚRY BUDOUCÍHO ROZVOJE APLIKACE	108
10.4	SHRNUTÍ HODNOCENÍ VYVINUTÉ APLIKACE	108
	ZÁVĚR	110
	SEZNAM POUŽITÉ LITERATURY.....	111
	SEZNAM OBRÁZKŮ	116
	SEZNAM TABULEK.....	118
	SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK	119

ÚVOD

Porozita materiálu je při výrobě pneumatik kritickým parametrem ovlivňujícím kvalitu, bezpečnost a životnost výrobků. Vysoká míra porozity v gumové směsi představuje riziko pro kvalitu finálního produktu, protože vytváří slabá místa ve struktuře a nepříznivě ovlivňuje jeho funkční vlastnosti. Tradiční metody hodnocení porozity jsou založeny na subjektivním vizuálním posouzení zkušenými pracovníky, nebo na časově a finančně náročných laboratorních analýzách vyžadujících specializované vybavení. Tyto přístupy však neumožňují okamžitou a přesnou zpětnou vazbu přímo ve výrobním procesu. Z tohoto důvodu je diplomová práce zaměřena na vývoj nástroje v podobě mobilní iOS aplikace, který by umožnil rychlý, levný a objektivní odhad porozity vzorků pneumatik.

V teoretické části jsou představeny mobilní platformy společnosti Apple. Důraz je kladen také na jejich vzájemné propojení a distribuční mechanismy. V dalších částech jsou popsány přístupy k vývoji mobilních aplikací a jejich vzájemné porovnání. Následně je popsán vývoj pro platformu iOS včetně klíčových nástrojů a knihoven. Součástí teoretické části je také charakteristika vývojového prostředí Xcode a vymezení pojmu porozity materiálu.

V praktické části je provedena analýza existujících mobilních aplikací pro měření fyzikálních vlastností, kontrolu kvality a detekci porozity materiálů. Cílem je identifikovat současný stav řešení a příležitosti pro nové inovativní přístupy. Následně je popsán kompletní proces vytvoření funkčního prototypu aplikace, který zahrnuje jeho specifikaci včetně definování funkcionálních a nefunkcionálních požadavků, návrh drátěných modelů a datového modelu, implementaci klíčových komponent systému a validační testování. Na základě provedených testů je v závěru praktické části provedeno zhodnocení vytvořené aplikace, kde jsou popsány dosažené výsledky, identifikovány limity řešení a navrženy směry možného budoucího rozvoje.

Výstupy práce přibližují problematiku využití iOS aplikací k odhadu a kvantifikaci porozity vzorků pneumatik při výrobě.

I TEORETICKÁ ČÁST

1 MOBILNÍ PLATFORMY SPOLEČNOSTI APPLE

Společnost Apple vyvíjí specializované operační systémy pro různé kategorie mobilních zařízení. Každá platforma je optimalizována pro specifické využití daného typu zařízení. Jednotlivé systémy jsou navrženy s důrazem na vzájemnou kompatibilitu a sdílení funkcionalit. V následujících podkapitolách jsou popsány jednotlivé mobilní operační systémy společnosti Apple, jejich specifické vlastnosti a vzájemné propojení, včetně distribuční platformy App Store.

1.1 iOS

Systém iOS je operační systém vyvíjený společností Apple pro zařízení iPhone a dříve také iPad (do roku 2019) a iPod Touch (do ukončení výroby v roce 2022). [1][49] Systém byl představen v roce 2007 společně s prvním iPhone. Od té doby prošel kontinuálním vývojem, zejména v oblastech uživatelského rozhraní, bezpečnostních mechanismů a aplikačního ekosystému. Na rozdíl od systému Android, který podporuje širokou škálu zařízení různých výrobců, představuje iOS uzavřený systém optimalizovaný výhradně pro hardware Apple, což zajišťuje vysokou stabilitu a bezpečnost platformy. [1] Vývoj aplikací pro iOS je realizován primárně v programovacích jazycích Swift nebo Objective-C s využitím vývojového prostředí Xcode, přičemž distribuce aplikací podléhá přísným požadavkům App Store. [2] Kromě základního dotykového ovládání zahrnuje iOS pokročilé funkcionality jako hlasového asistenta Siri, multitaskingové možnosti nebo systémové profily Focus pro správu notifikací. [1] Integrace s cloudovými službami umožňuje aplikacím efektivní správu dat a využití výpočetních zdrojů vzdálených serverů. [2] Celosvětový podíl iOS na trhu mobilních operačních systémů dosáhl v roce 2023 hodnoty 19,9 %, což pro společnost Apple představuje víceleté maximum. [1]

1.2 iPadOS

Systém iPadOS je operační systém společnosti Apple představený v roce 2019, který vychází z iOS, ale je určený výhradně pro tablety iPad a rozšiřuje možnosti této platformy o funkcionalitu optimalizovanou pro větší displeje a pokročilejší způsoby interakce. Systém přináší pokročilé multitaskingové schopnosti prostřednictvím funkcí Split View pro současné zobrazení dvou aplikací vedle sebe a Slide Over umožňující práci s dodatečnou aplikací v plovoucím okně, což významně zvyšuje produktivitu při práci s tabletem. Zásadní komponentou iPadOS je vylepšená správa souborů realizovaná prostřednictvím aplikace

Files, která poskytuje uživatelům přístup k hierarchické struktuře souborového systému podobně jako v desktopových operačních systémech. Platforma zahrnuje nativní podporu externích vstupních zařízení včetně myši, klávesnic a stylusu Apple Pencil, což umožňuje využití iPadu pro profesionální účely a kreativní práci s vysokou mírou přesnosti. Funkce Sidecar rozšiřuje možnosti iPadOS o schopnost fungovat jako externí monitor pro zařízení s macOS, čímž zvyšuje integraci a využitelnost v rámci ekosystému Apple. Safari prohlížeč v iPadOS je optimalizován pro načítání desktopových verzí webových stránek a obsahuje pokročilého správce stahování, který přibližuje uživatelskou zkušenost k desktopovým standardům. Architektura iPadOS zachovává bezpečnostní a vývojářské principy iOS při současném přizpůsobení specifickým požadavkům tabletové platformy, což vytváří konzistentní vývojové prostředí napříč mobilními zařízeními Apple. [3]

1.3 watchOS

Systém watchOS je operační systém společnosti Apple určený pro chytré hodinky Apple Watch. Na trh byl poprvé uveden v roce 2015 jako specializovaná platforma optimalizovaná pro nositelná zařízení s malými displeji a specifickými způsoby interakce. Systém je navržen s důrazem na hlubokou integraci s ostatními zařízeními Apple. Platforma watchOS se od svého vzniku kontinuálně vyvíjela směrem k rozšiřování možností v oblasti zdravotního monitorování a zaznamenávání fitness aktivit. Zahrnuje pokročilé funkce jako sledování srdečního tepu, detekci pádů, analýzu spánkových vzorců a monitorování různých typů fyzických aktivit prostřednictvím integrovaných senzorů. Nejnovější iterace systému rozšiřují možnosti personalizace uživatelského rozhraní včetně upravitelných ciferníků a vylepšených nástrojů pro sledování sportovních výkonů a zdravotních parametrů s možností dlouhodobého sledování trendů. Architektura watchOS podporuje širokou škálu aplikací zahrnující jak předinstalované systémové aplikace, tak aplikace třetích stran, přičemž využívá hardwarové senzory Apple Watch pro sběr a analýzu biometrických dat v reálném čase. Integrace s klíčovými službami iOS včetně Apple Pay pro bezkontaktní platby, Siri pro hlasové ovládání a systému notifikací, transformuje Apple Watch z pouhého příslušenství na samostatné funkční zařízení schopné vykonávat mnoho úloh nezávisle na připojeném telefonu. [4]

1.4 Propojení platforem

Propojení jednotlivých platforem společnosti Apple představuje jeden z esenciálních faktorů, který odlišuje její produkty od konkurence. Základními technologiemi této integrace

jsou služby Apple ID pro jednotnou autentifikaci, iCloud pro cloudovou synchronizaci dat a funkcionality Continuity, která umožňuje bezproblémové pokračování aktivit mezi zařízeními bez ztráty kontextu práce nebo komunikace. Například funkce Handoff umožňuje pokračovat v práci na jiném zařízení přesně tam, kde uživatel skončil. Funkce Universal Clipboard podporuje kopírování a vkládání obsahu mezi zařízeními v reálném čase. Další prvky, jako je možnost odemykání Mac zařízení pomocí Apple Watch nebo automatické přepínání připojených AirPods sluchátek mezi zařízeními, zvyšují komfort a uživatelský zážitek. Tyto mechanismy propojení nejen zvyšují efektivitu práce, ale také posilují loajalitu uživatelů tím, že činí přechod k zařízením jiných značek složitější. Apple tak skrze tuto strategii nejen zlepšuje uživatelský komfort, ale zároveň upevňuje svou tržní pozici a odlišuje se od konkurence. [5]

1.5 App Store

App Store je digitální distribuční platforma od společnosti Apple, která slouží k distribuci aplikací pro zařízení s operačními systémy iOS, iPadOS, watchOS, tvOS a macOS. Platforma byla spuštěna v roce 2008 a v současnosti (tedy v roce 2025) nabízí více než 2,2 milionu aplikací. Mezi nimi se nachází jak aplikace vyvinuté samotnou společností Apple, jako jsou například Keynote nebo GarageBand, tak populární aplikace třetích stran, například od společností Meta či Microsoft. Aplikace v App Store mohou být zdarma, obsahovat jednorázové platby nebo nabízet funkce prostřednictvím předplatného či nákupů uvnitř aplikací (anglicky *in-app purchases*). Kromě běžného stahování aplikací App Store nabízí tematické kolekce doporučených aplikací, denní články a recenze v sekci „Dnes“. Důležitou částí je také herní služba Apple Arcade, která nabízí neomezené hraní vybraných mobilních her za měsíční poplatek. App Store je zároveň místem, kde Apple nejen distribuuje aplikace, ale i ovlivňuje kvalitu aplikací prostřednictvím zpětné vazby od uživatelů a pravidelných aktualizací pravidel pro vývojáře. [6]

2 MOŽNOSTI VÝVOJE APLIKACÍ PRO MOBILNÍ PLATFORMY

Vývoj mobilních aplikací představuje komplexní oblast, která nabízí různé přístupy k vytváření aplikací pro mobilní zařízení. V současné době je možné rozlišit tři základní metodiky vývoje mobilních aplikací, z nichž každá má své specifické charakteristiky, výhody a omezení. Tyto přístupy se liší především v používaných technologiích, programovacích jazycích, možnostech přístupu k funkcím zařízení a celkové architektuře řešení.

První přístup představuje nativní vývoj, který využívá platformově specifické programovací jazyky a nástroje pro vytváření aplikací optimalizovaných pro konkrétní operační systém. Druhým přístupem je multiplatformní vývoj, který umožňuje sdílení kódu mezi různými platformami při zachování nativních vlastností. Třetí možností je hybridní vývoj, který kombinuje webové technologie s nativními komponenty. [14]

Následující podkapitoly detailně popisují jednotlivé přístupy, jejich charakteristické vlastnosti, výhody a omezení. Závěrečné podkapitoly poskytují komparativní analýzu těchto přístupů a shrnutí klíčových aspektů pro rozhodování při výběru vhodné metodiky pro konkrétní projekt.

2.1 Nativní vývoj

Nativní vývoj mobilních aplikací představuje přístup, při kterém jsou aplikace vytvářeny od základu s využitím platformově specifických programovacích jazyků a vývojových nástrojů. Tento přístup vyžaduje od vývojářů používání různých technologií pro každou cílovou platformu – například pro Android se využívají jazyky Java nebo Kotlin, zatímco pro iOS se používají Swift nebo Objective-C. [15]

Charakteristickou vlastností nativního vývoje je přímé využívání nativních knihoven, API (anglická zkratka pro *Application Programming Interface*) a SDK (anglická zkratka pro *Software Development Kit*) konkrétní platformy. Tento přístup umožňuje vývojářům plně využít všechny možnosti a funkce cílové platformy, včetně přístupu ke všem hardwarovým komponentám zařízení, jako jsou kamera, mikrofón, akcelerometr, GPS nebo souborový systém. [14]

Nativní aplikace dosahují vysokého výkonu, protože je jejich kód spouštěn přímo na cílové platformě bez nutnosti použití převodních nebo interpretačních vrstev. Tento přístup umožňuje vytváření intuitivních uživatelských rozhraní, která plně respektují

designové konvence konkrétní platformy a poskytují uživatelům konzistentní prostředí shodné s ostatními nativními aplikacemi na zařízení. [14][15] Významnou výhodou je přímý přístup k nejnovějším funkcím operačního systému, kdy vývojáři mohou okamžitě implementovat nové funkce po jejich vydání bez čekání na podporu ze strany multiplatformních frameworků. Nativní aplikace nabízejí robustní offline funkcionalitu, která umožňuje práci s lokálním úložištěm a databázemi, což je důležité pro uživatele v oblastech s omezeným internetovým připojením. Knihovny pro nativní platformy poskytují vývojářům pokročilé nástroje pro implementaci bezpečnostních funkcí, jako je šifrování souborů nebo detekce podvodů, přičemž proces ladění kódu je efektivnější díky přímému přístupu k systémovým nástrojům. Nativní vývoj je dále nezbytný pro vysoce komplexní aplikace vyžadující náročné zpracování dat nebo pokročilé grafické funkce, jako jsou herní aplikace s 3D animacemi. [14]

Nativní vývoj však přináší významná omezení v podobě nemožnosti opětovného použití kódu mezi platformami, což vyžaduje vytvoření samostatných verzí aplikace pro každou platformu. Tato skutečnost vede k vyšším nákladům na vývoj a nutnosti udržovat více vývojových týmů se specializovanými dovednostmi pro jednotlivé platformy. Další nevýhodou je delší doba uvedení na trh, protože je nutné projít celým vývojovým cyklem pro každou platformu zvlášť. Správa většího týmu vývojářů s různými specializacemi může být náročná na koordinaci a komunikaci, což dále zvyšuje komplexnost projektového řízení. [14][15]

Nativní aplikace jsou dostupné výhradně prostřednictvím zabudovaných distribučních platforem, což je vystavuje specifickým pravidlům a omezením těchto služeb. Každá aktualizace musí projít schvalovacím procesem, který může trvat od jednoho dne až po dva týdny. [14][17]

2.2 Multiplatformní vývoj

Multiplatformní vývoj, označovaný také jako *cross-platform* development, představuje přístup k vytváření mobilních aplikací, které jsou kompatibilní s více operačními systémy současně. Namísto vytváření samostatných aplikací pro iOS a Android mohou vývojáři sdílet některé nebo i téměř všechny části zdrojového kódu mezi různými platformami, čímž zajišťují konzistentní funkcionalitu napříč všemi cílovými platformami. [15]

Charakteristickou vlastností multiplatformního vývoje je využití specializovaných frameworků, které poskytují nástroje, knihovny a komponenty uživatelského rozhraní

umožňující vytvoření aplikací pro více platform ze společného kódu. Na trhu je dostupných několik populárních frameworků, mezi které patří Flutter, React Native, Kotlin Multiplatform, Ionic, .NET MAUI nebo NativeScript. Tyto frameworky se liší podle používaných nástrojů, programovacích jazyků, podporovaných platform, možností znovupoužití kódu a dalších faktorů. [15][16]

Multiplatformní vývoj přináší výrazné benefity v oblasti efektivity a ekonomické výhodnosti prostřednictvím možnosti sdílení zdrojového kódu mezi různými operačními systémy bez nutnosti vytváření samostatných aplikací pro každou platformu. Vývojáři mohou využít jednotný kód pro iOS i Android, což významně urychluje vývojový proces díky redukcí množství kódu, který je třeba napsat a otestovat. Tento přístup představuje nákladově efektivní řešení zejména pro startupy a společnosti s omezenými rozpočty, protože umožňuje pokrytí více platform při současném snížení celkových vývojových nákladů. Moderní multiplatformní technologie nabízejí flexibilní sdílení kódu, které umožňuje vývojářům využívat výhody nativního programování při zachování možnosti opětovného použití komponent napříč platformami. Implementace těchto technologií může přilákat nové vývojáře a zvýšit motivaci vývojářů díky práci s inovativními nástroji a řešením nových výzev v oblasti vývoje aplikací. [15]

Multiplatformní přístup však čelí několika technickým a praktickým omezením, která mohou ovlivnit kvalitu výsledného produktu a efektivitu vývojového procesu. Výkonnostní charakteristiky multiplatformních aplikací mohou zaostávat za nativními řešeními. Přístup k platformově specifickým API a nativním funkcím mobilních zařízení vyžaduje dodatečné úsilí a může být technicky náročnější než u nativního vývoje. Konzistence uživatelského rozhraní může být u frameworků umožňujících sdílení UI (anglická zkratka pro *User Interface*) komponent omezená, přičemž aplikace mohou působit méně nativně ve srovnání s aplikacemi vyvinutými specificky pro danou platformu. Proces najímání kvalifikovaných vývojářů představuje výzvu, protože na pracovním trhu je dostupných méně profesionálů se zkušenostmi s multiplatformními frameworky ve srovnání s vývojáři specializovanými na jednotlivé platformy. [15]

2.3 Hybridní vývoj

Hybridní vývoj mobilních aplikací představuje kombinaci nativního a webového přístupu. Jádro aplikace je v tomto případě napsáno pomocí webových technologií jako je HTML, CSS a JavaScript, které jsou následně zabaleny do nativní aplikace. Prostřednictvím

specifických knihoven mohou tyto aplikace získat plný přístup k funkcím mobilního zařízení. [14][17]

Charakteristickou vlastností hybridního přístupu je využití vestavěného prohlížeče v rámci nativní aplikace. Systém iOS využívá komponentu *WKWebView*, zatímco Android využívá *WebView* pro zobrazení webové aplikace. Kód je následně zabalen do nativní aplikace pomocí řešení jako Apache Cordova nebo Ionic Capacitor. [17]

Hybridní vývoj aplikací přináší významné výhody v oblasti efektivity a ekonomické výhodnosti díky možnosti sdílení kódu, což umožňuje vývojářům vytvářet aplikace rychle a nákladově efektivně pro více platforem současně. [14] Tento přístup zajišťuje rychlé uvedení na trh a dosažení širokého publika napříč různými platformami. K tomu využívá webové technologie HTML, CSS a JavaScript zapouzdřené v nativní aplikační obálce. [14] [17] Prostřednictvím systému rozšíření mohou hybridní aplikace získat plný přístup k funkcím mobilního zařízení, čímž překonávají omezení čistě webových aplikací a dosahují parity s nativními aplikacemi v oblasti funkčnosti. [17] Offline funkcionality představuje další výhodu hybridních řešení, která uživatelům umožňuje využívat aplikaci i bez internetového připojení. [14] Podpora multiplatformního vývoje s možností opětovného použití kódu činí hybridní přístup atraktivním pro organizace, které potřebují rychle vytvořit a spustit minimální produkt pro pilotní projekty, protože poskytuje rychlejší, jednodušší a cenově dostupnější řešení. [14]

Tento přístup však čelí několika technickým omezením, která mohou negativně ovlivnit uživatelskou zkušenost a celkový výkon aplikace. Ve srovnání s nativními aplikacemi závislost na vestavěném prohlížeči často vede k nižšímu výkonu a může způsobovat nekonzistentní uživatelská rozhraní napříč různými systémy. [14][17] Přístup k hardwarovým funkcím zařízení je omezen a vyžaduje implementaci specifických rozšíření, což může komplikovat vývojový proces. [14] Bezpečnostní konfigurace představují dodatečnou komplexitu, kterou je nutné řešit při vývoji hybridních aplikací. [14] Uživatelské rozhraní musí být znovu vytvořeno pomocí specializovaných knihoven, protože běžné aplikační UI komponenty neexistují nativně ve webovém prostředí, což může vést k tomu, že aplikace nefunguje zcela správně nebo působí méně nativně. [17]

2.4 Shrnutí možností vývoje

Výběr vhodného přístupu k vývoji mobilních aplikací představuje strategické rozhodnutí, které má dlouhodobý dopad na úspěch projektu. Každý z analyzovaných přístupů – nativní, multiplatformní a hybridní – nabízí jedinečné výhody a má svá specifická omezení.

Nativní vývoj zůstává nejvhodnější volbou pro komplexní aplikace vyžadující maximální výkon, plný přístup k hardwarovým funkcím a nejlepší uživatelské prostředí. Je ideální pro projekty s dostatečnými zdroji a týmy se specializovanými dovednostmi pro jednotlivé platformy. [14][15]

Multiplatformní vývoj představuje kompromis mezi efektivitou vývoje a kvalitou výsledného produktu. Moderní multiplatformní technologie se neustále zlepšují a umožňují vytváření aplikací s téměř nativním výkonem při současném sdílení značné části kódu mezi platformami. [15]

Hybridní přístup je vhodný především pro rychlé prototypování, pilotní projekty nebo aplikace s omezenou životností. Ačkoli může vykazovat horší výkon, nabízí rychlé a nákladově efektivní řešení pro nenáročné mobilní aplikace. [14][17]

Rozhodnutí by mělo být založeno na pečlivém vyhodnocení projektových požadavků, dostupných zdrojů, odborných znalostí týmu a dlouhodobých cílů organizace. Neexistuje univerzální řešení – nejlepší přístup závisí na specifických potřebách každého projektu. [14][15]

3 VÝVOJ APLIKACÍ PRO PLATFORMU IOS

Při vývoji nativních aplikací pro mobilní platformy společnosti Apple existuje celá řada nástrojů a knihoven, které umožňují efektivní tvorbu, testování a publikaci softwaru. Tato sada nástrojů zahrnuje nejen vývojové prostředí a programovací jazyky, ale i knihovny a API, jež usnadňují integraci různých hardwarových funkcí zařízení, jako jsou například kamera nebo senzory. [11]

Základními předpoklady pro vývoj aplikací pro iOS je splnění několika požadavků: použití zařízení Mac s operačním systémem macOS, instalace vývojového prostředí Xcode a aktivní účet v programu Apple Developer.

V následujících podkapitolách jsou podrobně představeny hlavní nástroje pro vývoj iOS aplikací, konkrétně programovací jazyk Swift, deklarativní framework SwiftUI pro tvorbu uživatelských rozhraní, Foundation framework poskytující základní systémové knihovny, SwiftData pro správu persistentních dat, Core Image framework pro zpracování a analýzu obrazových dat a vývojové prostředí Xcode sloužící jakožto integrovaný nástroj pro celý vývojový cyklus.

3.1 Swift

Swift je moderní programovací jazyk, který byl navržen s důrazem na bezpečnost, výkonnost a jednoduchost použití. Jedná se o univerzální jazyk, který je vhodný pro širokou škálu aplikací od systémového programování přes mobilní a desktopové aplikace až po cloudové služby. [20] Swift se vyznačuje kompilovanou povahou a statickým typováním, což zajišťuje kontrolu datových typů již v době kompilace a přispívá k celkové bezpečnosti kódu. [19]

Jazyk Swift byl oficiálně představen společností Apple v roce 2014 jako alternativa k dosud používanému jazyku Objective-C pro vývoj aplikací na Apple platformách. [19] Navzdory tomu, že byl původně vyvinut pro zařízení Apple, je Swift open-source projekt dostupný pod licencí Apache 2.0, což podporuje jeho používání a vývoj napříč různými platformami. [20]

Swift je založen na moderních principech návrhu programovacích jazyků a kombinuje zkušenosti z desetiletí vývoje softwaru. Jazyk využívá kompilátor z LLVM sady nástrojů, který transformuje Swift kód do optimalizovaného strojového kódu a maximalizuje využití moderního hardwaru. Syntaxe jazyka je navržena tak, aby byla čistá a výrazná, přičemž vývojáři nemusí používat středníky na konci příkazů. Swift podporuje automatickou správu

paměti prostřednictvím deterministického počítání referencí, což minimalizuje využití paměti bez režie spojené s její správou. [12]

Mezi hlavní charakteristiky Swift jazyka patří bezpečnost designu, kdy jsou proměnné vždy inicializovány před použitím, pole a celá čísla jsou kontrolována na přetečení a potenciální souběžné přístupy k datům mohou být identifikovány již v době kompilace. Jazyk implementuje inovativní koncept *optionals* pro bezpečné zacházení s hodnotami *nil*, což eliminuje celou kategorii chyb za běhu programu typických pro jiné programovací jazyky. Swift také podporuje inferenci typů, což činí kód čitelnějším a méně náchylným k chybám, zatímco moduly eliminují potřebu hlavičkových souborů a poskytují jmenné prostory. [12]

V následujících podkapitolách jsou popsány základní koncepty jazyka Swift.

3.1.1 Hello, World!

Hello, World! je tradičním prvním krokem programování, který se používá k ověření základní funkčnosti vývojového prostředí a k seznámení s novým programovacím jazykem. Historicky se jedná o jednoduchý program, který zobrazí na obrazovce textový řetězec „Hello, World!“. Tento příklad má původ v 70. letech 20. století, kdy Brian Kernighan použil podobný příklad v knize *The C Programming Language* k ukázce jednoduchosti syntaxe a možností jazyka C. Postupně se stal standardem, který provází programátory při prvních krocích v každém novém jazyce. [21]

V kontextu programovacího jazyka Swift představuje program *Hello, World!* velmi jednoduchou strukturu, protože jak je možno pozorovat na obrázku číslo 1, k jeho sestavení stačí pouze jediný řádek kódu.

```
1 print("Hello, World!")
```

Obrázek 1 Program *Hello, World!* v jazyce Swift

Swift nevyžaduje explicitní deklaraci hlavní metody, jako například *main* v jiných programovacích jazycích. Veškerý kód, který je napsán na globální úrovni slouží automaticky jako vstupní bod programu. Další zajímavostí jazyka Swift, která přispívá k jeho jednoduchosti, je absence nutnosti importovat speciální knihovny pro základní operace jako je výpis textu. Stejně tak není vyžadováno ani používání středníků na konci příkazů, což ještě více zvyšuje čitelnost a přehlednost výsledného kódu. [13]

3.1.2 Proměnné a konstanty

V programovacím jazyce Swift se pro práci s daty používají dva základní typy identifikátorů: proměnné a konstanty. Proměnné jsou deklarovány pomocí klíčového slova *var*, zatímco konstanty se vytvářejí pomocí klíčového slova *let*. Konstanty se vyznačují tím, že jejich hodnota musí být přiřazena právě jednou během životního cyklu programu. Tato hodnota nemusí být známa v době kompilace, ale jakmile je konstantě přiřazena, nelze ji již změnit. Konstanty tak umožňují pojmenovat hodnotu, která je určena jednou, ale následně používána na více místech v kódu. Ve srovnání s konstantami proměnné umožňují opakovanou změnu své hodnoty během běhu programu. [13]

Swift disponuje pokročilým systémem inference typů, který umožňuje kompilátoru automaticky odvodit typ proměnné nebo konstanty na základě přiřazené hodnoty. Pokud je například při inicializaci poskytnuta celočíselná hodnota, kompilátor automaticky odvozuje typ *Integer*. Tato funkcionality eliminuje potřebu explicitního specifikování typu ve většině případů. [13]

V situacích, kdy počáteční hodnota neposkytuje dostatečné informace pro inferenci typu, nebo když není k dispozici žádná počáteční hodnota, je nutné typ specifikovat explicitně. To se provádí zápisem typu za názvem proměnné nebo konstanty, odděleným dvojtečkou. Explicitní specifikace typu je také vyžadována při vytváření prázdných polí, kde není k dispozici kontextová informace o typu. [13]

Důležitým principem jazyka Swift je absence implicitní konverze typů. Pokud je potřeba převést hodnotu na jiný typ, musí se vytvořit explicitní instance požadovaného typu. Toto pravidlo přispívá k bezpečnosti kódu a předchází neočekávaným chybám způsobeným automatickou změnou typu. [13]

Pro začlenění hodnot do řetězců poskytuje Swift interpolaci řetězců pomocí syntaxe „`()`“. Tato funkcionality umožňuje vložení hodnoty proměnné nebo výsledku výpočtu přímo do řetězce znaků, což výrazně zjednodušuje konstrukci složitějších textových výstupů. [13]

Obrázek číslo 2 obsahuje základní ukázkou práce s proměnnými a konstantami.

```
1 // Vytvoření proměnné a konstanty
2 var myVariable = 10
3 let myConstant = 100
4
5 // Změna hodnoty proměnné
6 myVariable = 11
7
8 // Explicitní specifikace typu
9 let explicitDouble: Double = 3
10
11 // Interpolace řetězců
12 let name = "Swift"
13 let greeting = "Hello, \(name)!"
```

Obrázek 2 Proměnné a konstanty v jazyce Swift

3.1.3 Řízení toku programu

Řízení toku programu představuje základní mechanismus pro určování pořadí vykonávání příkazů v aplikaci na základě podmínek a opakovaných operací. Swift poskytuje standardní konstrukty pro řízení toku zahrnující podmíněné příkazy, cykly a mechanismy pro zpracování chybějících hodnot. Syntaxe těchto konstruktů nevyžaduje kulaté závorky okolo podmínky, zatímco složené závorky okolo těla příkazu jsou povinné. [13]

Podmíněné příkazy umožňují vykonání různých částí kódu na základě splnění určitých kritérií. Příkaz *if* vyžaduje booleovský výraz jako podmínku, což eliminuje běžné programátorské chyby způsobené neočekávanou konverzí hodnot. Specifickou funkcionalitou Swiftu je kombinace *if* a *let* pro práci s volitelnými hodnotami (anglicky *optionals*), které mohou obsahovat buď skutečnou hodnotu, nebo *nil* reprezentující absenci hodnoty. Tento mechanismus (anglicky zvaný *optional binding*) umožňuje bezpečné „rozbalení“ hodnoty pouze v případě, že není prázdná, čímž se předchází chybám způsobeným přístupem k neexistujícím datům. Pro zpracování chybějících hodnot poskytuje Swift operátor „??“, který umožňuje specifikovat výchozí hodnotu použitou v případě absence původní hodnoty. [13]

Příkaz *switch* představuje pokročilý mechanismus pro větvení programu podporující různé datové typy a komplexní srovnávací operace, včetně vzorů s podmínkami pomocí klíčového slova *where*. Na rozdíl od mnoha jiných programovacích jazyků je vykonávání jednotlivých případů automaticky ukončeno po jejich zpracování bez potřeby explicitního klíčového slova *break*, což snižuje riziko nechtěného vykonání ostatních případů. [13]

Cykly umožňují opakované vykonávání bloku kódu podle specifikovaných pravidel. Cyklus *for-in* iteruje přes kolekce dat a pro slovníky umožňuje současné použití párů názvů

pro klíč a hodnotu. Navíc podporuje ignorování nepotřebných hodnot pomocí podtržítka. Příkazy *while* a *repeat-while* opakují blok kódu na základě podmínky, kde *repeat-while* testuje podmínku až po prvním provedení, což zaručuje alespoň jedno vykonání cyklu. Pro numerické sekvence Swift poskytuje operátory rozsahu „..*<*“ (neobsahuje horní hranici) a „..*<=*“ (zahrnuje obě hranice), které zjednodušují práci s indexy a iteracemi v cyklech. [13]

Praktické využití těchto konstruktů je zobrazeno na obrázku číslo 3, který ukazuje implementaci podmíněných příkazů, práci s volitelnými hodnotami, switch konstrukce s pokročilými vzory a iteraci pomocí *for-in* cyklu.

```
1 // Podmíněný příkaz if
2 let score = 10
3 if score > 5 {
4     print("OK")
5 } else {
6     print("Fail")
7 }
8
9 // Práce s optionals
10 var optionalName: String? = "Dominik"
11 if let name = optionalName {
12     print("Hello, \(name)")
13 }
14
15 // Switch příkaz
16 let fruit = "banana"
17 switch fruit {
18 case "banana":
19     print("Yellow fruit")
20 case let x where x.contains("berry"):
21     print("It's a \(x)")
22 default:
23     print("Unknown fruit")
24 }
25
26 // For-in cyklus
27 for i in 0..<7 {
28     print("Number: \(i)")
29 }
```

Obrázek 3 Řízení toku programu v jazyce Swift

3.1.4 Funkce

Funkce představují základní prvek jazyka Swift sloužící k organizaci kódu do opakovaně použitelných bloků, což umožňuje lepší strukturu programů, opakovanou využitelnost logiky a jednodušší ladění aplikací. Ve Swiftu se funkce deklarují pomocí klíčového slova *func* následovaného názvem funkce, seznamem parametrů a případně návratovým typem odděleným symbolem „*->*“, přičemž návratová hodnota se určuje příkazem *return*. Funkce jsou ve Swiftu takzvanými „first-class citizens“, což znamená, že s nimi lze zacházet

jako s běžnými hodnotami – mohou být přiřazeny do proměnné, předány jako argument jiné funkci nebo vráceny jako výsledek, což umožňuje vytváření vysoce modulárních a dynamických programů. Jazyk podporuje vnořené funkce s přístupem k proměnným vnější funkce a funkcionální styl programování prostřednictvím funkcí vyššího řádu, jako je metoda *map*, která aplikuje funkci na každý prvek kolekce. [13]

Praktickou implementaci základní funkce pro sčítání dvou číselných hodnot s ukázkou využití návratové hodnoty lze vidět na obrázku 4.

```
1 func add(a: Int, b: Int) -> Int {  
2     return a + b  
3 }  
4  
5 let result = add(a: 2, b: 3)  
6 // result obsahuje hodnotu 5
```

Obrázek 4 Deklarace funkce v jazyce Swift

3.1.5 Objekty a třídy

Objektově orientované programování v jazyce Swift využívá třídy jako šablony pro vytváření objektů, které reprezentují entity s vlastnostmi a chováním. Třída se definuje pomocí klíčového slova *class* následovaného názvem třídy, přičemž uvnitř těla třídy se deklarují vlastnosti (anglicky *properties*) pro uchování stavu objektu a metody (anglicky *methods*) pro definování jeho chování. Vlastnosti se zapisují stejně jako běžné proměnné nebo konstanty, zatímco metody jsou obdobné funkcím, ale jsou definovány v kontextu konkrétní třídy. [13]

Instance třídy se vytváří přidáním kulatých závorek za její název. Přístup k vlastnostem a metodám objektu se realizuje pomocí tečkové notace. Významnou komponentou tříd je inicializátor *init*, který slouží k nastavení počátečního stavu objektu při jeho vytvoření, kde pomocí klíčového slova *self* lze odlišit parametry inicializátoru od vlastností objektu. Každá vlastnost musí mít buď definovanou výchozí hodnotu, nebo být explicitně nastavena v inicializátoru, což zajišťuje konzistentní inicializaci objektů. [13]

Swift podporuje dědičnost tříd, což umožňuje vytváření podtříd, které rozšiřují nebo upravují funkcionality nadřazené třídy. Dědičnost se označuje dvojtečkou za názvem podtřídy s uvedením názvu nadřazené třídy. Pokud podtřída přepisuje metodu zděděnou z nadřazené třídy, je nutné použít klíčové slovo *override*, přičemž kompilátor kontroluje správnost přepsání a v případě chyby vyvolá upozornění. [13]

Praktickou implementaci základní třídy s vlastnostmi, metodami a vytvořením instance lze vidět na obrázku číslo 5.

```
1 class Shape {  
2     var numberOfSides = 0  
3  
4     func simpleDescription() -> String {  
5         return "A shape with \(numberOfSides) sides."  
6     }  
7 }  
8  
9 let shape = Shape()  
10 shape.numberOfSides = 3
```

Obrázek 5 Deklarace třídy a její použití v jazyce Swift

3.1.6 Výčtové typy a struktury

Výčtové typy a struktury představují základní nástroje pro modelování dat v jazyce Swift, přičemž každá konstrukce má své specifické vlastnosti a vhodné scénáře použití. Výčtové typy (anglicky *enum*) umožňují definování skupiny souvisejících hodnot, přičemž každému případu lze přiřadit buď pevnou hodnotu (například číslo nebo řetězec), nebo s ním spojit dodatečná data specifická pro konkrétní výskyt výčtového typu. Swift výčty podporují také přidávání metod, což umožňuje jejich rozšíření o chování související s konkrétním typem, a jsou běžně používány například při reprezentaci stavů síťové komunikace nebo výsledků výpočtů. [13]

Struktury (anglicky *struct*) jsou podobné třídám, ale na rozdíl od nich jsou předávány hodnotou, nikoliv referencí, což znamená, že přiřazení struktury vytváří její kopii. Struktury jsou vhodné pro jednoduché datové typy bez sdíleného stavu, jako jsou souřadnice nebo rozměry, a mohou obsahovat vlastnosti, metody, inicializátory i plnit požadavky definované protokoly. [13]

Kombinace výčtů a struktur umožňuje vytváření složitějších modelů s vysokou čitelností a bezpečností typů. To je možné vidět na obrázku číslo 6, který ukazuje příklad výčtového typu pro teplotní jednotky a struktury teploměru.

```
1 enum TemperatureUnit {
2     case celsius, fahrenheit
3 }
4
5 struct Thermometer {
6     var value: Double
7     var unit: TemperatureUnit
8
9     func description() -> String {
10         return "Temperature: \(value) \(unit)"
11     }
12 }
13
14 let reading = Thermometer(value: 22.5, unit: .celsius)
```

Obrázek 6 Výčtové typy a struktury v jazyce Swift

3.1.7 Asynchronní programování

Asynchronní programování představuje klíčový koncept v moderním vývoji aplikací, který umožňuje efektivní zpracování časově náročných úloh bez zablokování hlavního vlákna aplikace. Ve vývoji pro platformy Apple je asynchronní programování nezbytné zejména při síťové komunikaci, přístupu k souborům, zpracování obrázků nebo jiných operacích vyžadujících delší čas na dokončení. [13]

Funkce označené klíčovým slovem *async* mohou být pozastaveny a později obnoveny. Při jejich volání se používá klíčové slovo *await*, které označuje bod potenciálního čekání na výsledek operace. Swift umožňuje spuštění více asynchronních operací současně pomocí konstrukce *async let*, která deklaruje paralelní úlohy s možností pozdějšího získání jejich výsledků prostřednictvím *await*. Pro komplexnější scénáře nabízí jazyk strukturu *TaskGroup* pro souběžné provádění více úloh a typ *Task* pro spuštění asynchronního kódu ze synchronního prostředí. [13]

Z hlediska bezpečné správy stavu zavádí Swift koncept *actor*, který rozšiřuje objektově orientované programování o bezpečné řízení přístupu k proměnným v asynchronním prostředí. *Actor* funguje podobně jako třída, ale automaticky zajišťuje ochranu svého vnitřního stavu před souběžnými přístupy více vláken, čímž eliminuje potřebu ruční synchronizace. [13]

Praktické využití asynchronních funkcí s klíčovými slovy *async* a *await* je ukázáno na obrázku číslo 7.

```
1 func fetchUserID(from server: String) async → Int {  
2     return 123123123  
3 }  
4  
5 func fetchUsername(from server: String) async → String {  
6     let userID = await fetchUserID(from: server)  
7     return "User \(userID)"  
8 }  
9  
10 Task {  
11     let name = await fetchUsername(from: "main")  
12     print("Hello, \(name)")  
13 }
```

Obrázek 7 Asynchronní programování v jazyce Swift

3.1.8 Protokoly a rozšíření

Protokoly představují formální definici rozhraní, kterému musí vyhovovat všechny typy implementující daný protokol. Pomocí klíčového slova *protocol* lze specifikovat požadavky na vlastnosti a metody bez nutnosti konkrétní implementace. Klíčové slovo *mutating* označuje metody, které mohou měnit stav hodnotových typů jako jsou struktury a výčtové typy. Metody tříd tento modifikátor nevyžadují, protože instance tříd lze vždy upravovat bez omezení. [13]

Rozšíření (anglicky *extension*) umožňují obohacení existujících typů o nové funkce, vlastnosti i implementaci protokolů bez změny původního zdrojového kódu. Tento mechanismus lze aplikovat například na nativní typ *Int* nebo typy importované z externích knihoven, čímž se rozšiřují jejich možnosti podle potřeb konkrétní aplikace. Kombinace protokolů a rozšíření přispívá k modulární a znovupoužitelné architektuře, kdy různé datové typy mohou sdílet společné rozhraní bez duplicity kódu. Polymorfni práce s hodnotami protokolového typu zajišťuje přístup pouze k metodám uvedeným v protokolu. [13]

Praktické využití protokolů a rozšíření je ukázáno na obrázku číslo 8.

```
1 protocol Describable {
2     var description: String { get }
3 }
4
5 class Car: Describable {
6     let brand: String
7     let model: String
8
9     init(brand: String, model: String) {
10         self.brand = brand
11         self.model = model
12     }
13
14     var description: String {
15         return "\(brand) \(model)"
16     }
17 }
18
19 extension Int: Describable {
20     var description: String {
21         return "Číslo: \(self)"
22     }
23 }
```

Obrázek 8 Protokol a rozšíření v jazyce Swift

3.1.9 Zpracování chyb

Mechanismus zpracování chyb v programovacím jazyce Swift poskytuje strukturovaný přístup k řešení neočekávaných situací, které mohou nastat při běhu aplikace. Systém umožňuje předávání chybových stavů zpět k volající entitě za účelem jejich dalšího zpracování. [13]

Chyby jsou v prostředí Swift reprezentovány prostřednictvím libovolného typu, který implementuje protokol *Error*. Funkce schopné generovat chybové stavy jsou označovány klíčovým slovem *throws*. K vyvolání chyby slouží klíčové slovo *throw*. Volání takových funkcí vyžaduje použití konstrukce *do-catch*, kde je před volání funkce umístěno klíčové slovo *try* a v bloku *catch* jsou implementovány mechanismy pro zachycení chyb. Zachycení může být realizováno obecně pro všechny typy chyb, nebo selektivně podle konkrétních kategorií chybových stavů. [13]

Selektivní zachytávání představuje výhodu v podobě možnosti definovat specifické reakce na jednotlivé typy chyb, čímž se eliminuje nutnost rozlišování všech případů v rámci jednoho obecného bloku. Alternativní přístup představuje využití operátoru „try?“, který v případě vzniku chyby vrací hodnotu *nil* namísto očekávané hodnoty, čímž transformuje výraz na takzvaný *optional*. Pro zajištění provedení úklidových operací nezávisle na výskytu chybového stavu slouží klauzule *defer*. Kód obsažený v této klauzuli je vždy vykonán

bezprostředně před ukončením funkce. Tento přístup umožňuje tvorbu strukturovaného, čitelného a bezpečného kódu, který minimalizuje riziko neošetřených výjimek a zajišťuje konzistentní stav aplikace. [13]

Praktická ukázka zpracování chyb je znázorněna na obrázku číslo 9.

```
1 enum FileError: Error {
2     case notFound
3     case accessDenied
4 }
5
6 func openFile(_ name: String) throws -> String {
7     if name == "missing.txt" {
8         throw FileError.notFound
9     }
10    return "File opened: \(name)"
11 }
12
13 do {
14     let result = try openFile("data.txt")
15     print(result)
16 } catch FileError.notFound {
17     print("File not found")
18 } catch {
19     print("Other error")
20 }
21
22 let file = try? openFile("missing.txt")
23 print(file ?? "Failed to open")
```

Obrázek 9 Zpracování chyb v jazyce Swift

3.1.10 Generika

Generika představuje mechanismus, který umožňuje definovat funkce, typy a datové struktury způsobem, aby mohly pracovat s různými datovými typy, aniž by bylo nutné duplikovat kód pro každý konkrétní typ zvlášť. Tento přístup eliminuje potřebu vytváření samostatných implementací pro každý datový typ při zachování stejné funkcionality. [13]

Obecné typy se v deklaraci označují pomocí hranatých závorek s identifikátorem typu, typicky ve formátu „<T>“, kde T reprezentuje libovolný typ, který bude specifikován až při použití. Systém generik ve Swiftu umožňuje definovat omezení (anglicky *constraints*), která musí použitý typ splňovat, například implementaci konkrétního protokolu jako *Equatable*. Tato omezení se specifikují pomocí klíčového slova *where* umístěného před tělem funkce nebo za deklarací typu. [13]

Implementace těchto konceptů přináší významné výhody v podobě vyšší míry znovupoužitelnosti kódu a flexibility při jeho rozšiřování. Současně je zachována typová

bezpečnost, neboť kompilátor kontroluje správnost typů již v době překladu, čímž se předchází chybám, které by se jinak projeví až za běhu aplikace.

Na obrázku číslo 10 je uvedena ukázka generické funkce pro výměnu hodnot, která demonstruje princip definice a použití parametrického typu.

```
1 func swap<T>(_ a: inout T, _ b: inout T) {  
2     let temp = a  
3     a = b  
4     b = temp  
5 }
```

Obrázek 10 Ukázka generické funkce v jazyce Swift

3.2 SwiftUI

SwiftUI je deklarativní framework pro vytváření uživatelských rozhraní v programovacím jazyce Swift, představený společností Apple na vývojářské konferenci WWDC 2019. Na rozdíl od imperativního přístupu, kde jsou definovány konkrétní kroky popisující způsob vykreslení a umístění jednotlivých prvků, umožňuje SwiftUI vývojářům deklarativně specifikovat, jaký má být výsledný vzhled rozhraní. Systém následně automaticky zajišťuje detaily vykreslování a reakce na změny stavu aplikace. Díky této deklarativní povaze je výsledný kód obvykle stručnější a přehlednější, neboť vývojář definuje strukturu a vzájemné vztahy prvků namísto specifikace každého jednotlivého kroku pro jejich konstrukci a úpravy. [23][24]

SwiftUI současně poskytuje jednotnou abstraktní vrstvu pro všechny platformy Apple, což umožňuje sdílení značné části kódu napříč různými zařízeními. Framework lze bez komplikací kombinovat s existujícími frameworky UIKit a AppKit, což umožňuje využívání osvědčených komponent již zavedených frameworků bez nutnosti kompletní migrace existujícího kódu. [22]

Následující podkapitoly jsou zaměřeny na klíčové koncepty a mechanismy SwiftUI frameworku.

3.2.1 Kompozice

SwiftUI je založen na konceptu *view*, který představuje základní stavební prvek uživatelského rozhraní. *View* může být jakýkoliv vizuální element – od jednoduchého textu nebo tlačítka až po komplexní obrazovku aplikace. Každé *view* ve SwiftUI je definováno

jako struktura implementující protokol *View*, která vyžaduje pouze jednu povinnou vlastnost – *body* vracející popis toho, co má být zobrazeno. [25][26]

Základním principem SwiftUI je kompozice *view*, která umožňuje vytváření složitých uživatelských rozhraní postupným skládáním jednodušších komponent. Tento přístup rozděluje komplexní obrazovky na menší, logicky oddělené části, které lze opakovaně používat napříč aplikací. Výsledkem je modulární architektura, kde každá komponenta je srozumitelná, snadno udržitelná a testovatelná. Při potřebě změny vzhledu nebo chování komponenty stačí provést úpravu na jednom místě a SwiftUI automaticky propaguje změny do všech instancí této komponenty v aplikaci. Tento přístup minimalizuje duplikaci kódu a zjednodušuje údržbu aplikace. [26]

Pro organizaci a uspořádání *views* poskytuje SwiftUI kontejnery jako *VStack* pro vertikální uspořádání, *HStack* pro horizontální rozmístění a *ZStack* pro překrývání prvků. Každé jednotlivé *view* může obsahovat vlastní logiku, stylování i interaktivní chování. [26]

Na obrázku číslo 11 je možné vidět definici *view* s textem a jeho umístění do vertikálního kontejneru.

```
1 struct SimpleText: View {
2     var text: String
3     var body: some View {
4         Text(text)
5         .font(.headline)
6         .padding()
7     }
8 }
9
10 struct ContentView: View {
11     var body: some View {
12         VStack(spacing: 8) {
13             SimpleText(text: "Hello")
14             SimpleText(text: "World")
15         }
16     }
17 }
```

Obrázek 11 Kompozice *views* ve SwiftUI

3.2.2 Stav a datový tok

Moderní uživatelská rozhraní musí reagovat na změny dat a aktualizovat zobrazené informace v reálném čase. SwiftUI řeší tuto problematiku prostřednictvím systému reaktivního stavu, který automaticky sleduje změny dat a zajišťuje odpovídající aktualizaci rozhraní bez nutnosti manuálního zásahu vývojáře. [27]

Klíčovým mechanismem pro správu stavu jsou *property wrappers* – speciální atributy, které rozšiřují chování běžných vlastností o dodatečnou funkcionalitu. *Property wrapper* se označuje symbolem „@“ před názvem a automaticky obaluje původní vlastnost logikou pro sledování změn, validaci nebo jiné specifické chování.

Pro správu dat v jednotlivých komponentách slouží *property wrapper* *@State*, který umožňuje ukládání a sledování lokálních hodnot v rámci konkrétního *view*. Když je potřeba předat data mezi nadřazenou a podřazenou komponentou, využívá se mechanismus *Binding*, který vytváří propojení umožňující změny dat v obou směrech. [27]

Pro komplexnější scénáře, kde je nutné sdílet data napříč více komponentami aplikace, se implementuje třída splňující požadavky protokolu *ObservableObject*. Vlastnosti této třídy, které mají být sledovány, jsou označovány atributem *@Published*, čímž se zajistí automatická notifikace všech závislých *view* při změně hodnoty. Pro přístup k těmto sdíleným datům se využívá *@ObservedObject* nebo *@StateObject* podle kontextu použití. [27]

V situacích, kdy je potřeba zpřístupnit stav v nesouvisejících částech aplikace bez explicitního předávání parametrů, se používá *@EnvironmentObject*. Tento mechanismus umožňuje vložení objektu do prostředí aplikace a jeho následné využití v libovolné komponentě hierarchie *view*. [27]

Pro optimální výkon je důležité minimalizovat složitost stavových proměnných a vyhnout se komplexním výpočtům přímo v těle *view*, aby nedocházelo k nadměrnému překreslování a následnému zpomalení aplikace. [27]

Jednoduchý příklad správy stavu je možné vidět na obrázku číslo 12.

```
1 class Counter: ObservableObject {
2     @Published var count = 0
3 }
4
5 struct ContentView: View {
6     @StateObject private var counter = Counter()
7     @State private var step = 1
8
9     var body: some View {
10         VStack(spacing: 12) {
11             Stepper("Step: \(step)", value: $step, in: 1...5)
12             Button("Increment") {
13                 counter.count += step
14             }
15             Text("Count: \(counter.count)")
16         }
17         .padding()
18     }
19 }
```

Obrázek 12 Správa stavu ve SwiftUI

3.2.3 Modifikátory

Modifikátory představují způsob, jak ve SwiftUI upravovat vzhled a chování komponent uživatelského rozhraní. Jedná se o speciální funkce, které se aplikují na *view* pomocí tečkové notace a umožňují postupné přidávání různých úprav bez nutnosti vytváření složitých podtříd nebo manuální manipulace s rozložením. [28]

Princip fungování modifikátorů je založen na funkcionálním přístupu – každý modifikátor přijme původní *view* jako vstup, aplikuje na něj specifickou změnu a vrátí nový, upravený *view*. Tyto úpravy mohou zahrnovat vizuální vlastnosti jako velikost písma, barvu textu, odsazení, barvu pozadí nebo zaoblení rohů, ale také aspekty chování jako reakce na dotyky nebo animace. [28]

Modifikátory lze řetězit za sebou, což umožňuje kombinovat více úprav v logickém pořadí. Například *view* s textem může být postupně upraveno přidáním barvy, odsazení a pozadí jedním plynulým řetězcem volání. Díky deklarativnímu charakteru SwiftUI jsou všechny aplikované transformace přímo viditelné v kódu, což významně zvyšuje čitelnost a snižuje pravděpodobnost chyb při úpravách rozhraní. [28]

Modifikátory také podporují interaktivní prvky a animace prostřednictvím specializovaných funkcí jako *onTapGesture* pro reakci na dotyky nebo *rotationEffect* pro transformace, čímž umožňují snadnou tvorbu dynamických a responzivních uživatelských rozhraní. [28]

Ukázku použití modifikátorů lze vidět na obrázku číslo 13.

```
1 struct ContentView: View {  
2     var body: some View {  
3         Text("Tap Me")  
4             .font(.headline)  
5             .padding(12)  
6             .background(Color.blue)  
7             .foregroundColor(.white)  
8             .cornerRadius(8)  
9             .onTapGesture {  
10                 print("Text tapped")  
11             }  
12     }  
13 }
```

Obrázek 13 Modifikátory ve SwiftUI

3.2.4 Navigace

Navigace představuje významný aspekt moderních mobilních aplikací, který umožňuje uživatelům efektivně se pohybovat mezi různými obrazovkami a sekcemi aplikace. SwiftUI poskytuje komplexní navigační systém postavený na současných návrhových principech, který dokáže řešit jak jednoduché přechody, tak složité víceúrovňové navigační struktury. [31]

Základem navigačního systému SwiftUI jsou navigační kontejnery. Jedná se o specializované komponenty, které definují způsob organizace a prezentace obsahu aplikace. Hlavní navigační komponenty zahrnují *NavigationStack* pro vytváření hierarchických struktur *views* uspořádaných do zásobníku, *NavigationSplitView* pro zobrazení obsahu v rozdělených sloupcích (typicky pro větší obrazovky) a *TabView* pro přepínání mezi paralelními sekcemi aplikace prostřednictvím záložek. Vzhled a chování těchto kontejnerů se upravuje pomocí modifikátorů, například *navigationSplitViewStyle* pro styl rozděleného zobrazení nebo *navigationTitle* pro nastavení titulku v navigační liště. [31]

Pro vytváření přechodů mezi jednotlivými pohledy slouží *NavigationLink* komponenta, která představuje interaktivní prvek umožňující navigaci. SwiftUI rozlišuje dva základní způsoby implementace navigačních odkazů. První přístup umožňuje přímé vložení cílového pohledu, což je vhodné pro jednoduché případy. Druhý přístup pracuje s hodnotami, které jsou následně převáděny na konkrétní pohledy – tento způsob poskytuje větší kontrolu nad navigačním procesem a umožňuje aplikaci aktivně sledovat a ovlivňovat navigační stav. [32]

Správa navigačního stavu může být buď automatická, nebo může být převzata vývojářem pro pokročilejší scénáře. Automatická správa je dostačující pro základní navigační případy, zatímco manuální přístup umožňuje programové řízení navigace,

monitorování změn stavu a implementaci pokročilých funkcí jako je obnovení stavu aplikace po jejím restartu. [32]

3.2.5 Vztah SwiftUI s frameworkem UIKit

Framework UIKit je původní systém pro vytváření uživatelských rozhraní na platformě iOS, který byl představen v roce 2008. UIKit byl vyvinut v programovacím jazyce Objective-C a navržen tak, aby poskytl vývojářům kompletní sadu komponent a nástrojů pro zpracování uživatelských vstupů, reakce na dotykové ovládání a řízení celkového toku aplikace. Významnou součástí vývoje UIKit aplikací je *Interface Builder* integrovaný do vývojového prostředí Xcode, který umožňuje vizuální navrhování obrazovek prostřednictvím takzvaných *storyboards*, neboli grafických reprezentací sekvencí obrazovek a přechodů mezi nimi. UIKit aplikace jsou strukturovány podle principu oddělení logiky od *view*, kde *view controllers* zprostředkovávají komunikaci mezi datovým modelem a uživatelským rozhraním. [29]

Vztah mezi tradičním frameworkem UIKit a moderním SwiftUI je charakterizován vzájemnou kompatibilitou a možnostmi postupné migrace. Zatímco SwiftUI, představený v roce 2019, využívá deklarativní přístup zaměřený na popis výsledného stavu rozhraní, UIKit vyžaduje imperativní řízení jednotlivých kroků pro vykreslení a aktualizaci komponent. Pro bezproblémovou integraci poskytuje Apple specializované nástroje: třídu *UIHostingController* pro vkládání SwiftUI pohledů do existující hierarchie UIKit *view controllers*, a protokoly *UIViewRepresentable* a *UIViewControllerRepresentable* pro obalení UIKit komponent do SwiftUI prostředí. [29] Tento mechanismus podporuje postupné nahrazování komplexních UIKit částí novým deklarativním kódem bez nutnosti kompletního přepisu aplikace.

Při porovnání obou frameworků je nutné zohlednit několik klíčových charakteristik. UIKit představuje zavedené a prověřené řešení s rozsáhlou knihovnou komponent – od základních tlačítek až po složité kolekce a tabulky. Jeho hlavní výhodou je široká podpora starších verzí iOS a postupný vývoj zahrnující pokročilé funkce jako *Auto Layout* pro responzivní rozložení, dynamické velikosti písma nebo *drag-and-drop* interakce. Tato dlouhodobá evoluce má za následek rozsáhlou dokumentaci a aktivní vývojářskou komunitu. Zároveň však UIKit často vyžaduje psaní obsáhlejšího kódu pro řízení stavu a interakcí, což může vést k vyšší složitosti údržby při růstu velikosti projektu. [29]

SwiftUI nabízí kontrastní přístup založený na významném snížení objemu psaného kódu díky deklarativnímu stylu a vestavěnému reaktivnímu modelu, který automaticky aktualizuje rozhraní při změnách dat. Vývojáři mohou využívat funkci *live preview* v prostředí Xcode, která umožňuje sledování úprav kódu v reálném čase a významně urychluje proces návrhu a testování uživatelského rozhraní. Mezi omezení SwiftUI patří požadavek na verzi operačního systému ve verzi iOS 13, což omezuje kompatibilitu se staršími zařízeními. Méně rozvinutá knihovna předpřipravených komponent ve srovnání s UIKit může vyžadovat vytváření vlastních řešení tam, kde UIKit nabízí hotové komponenty. [30]

3.3 Foundation

Foundation je základní framework společnosti Apple, který poskytuje důležitou vrstvu funkcionality pro aplikace a frameworky na všech Apple platformách. Tento framework podporuje všechny hlavní platformy včetně iOS, iPadOS, macOS, tvOS, visionOS a watchOS. Účelem Foundation frameworku je poskytnutí univerzální sady základních nástrojů pro práci s daty, texty, čísly a další základní operace. Framework definuje třídy, protokoly a datové typy, které jsou používány napříč většinou knihoven a frameworků společnosti Apple, čímž zajišťuje konzistentní přístup k základním funkcionalitám. [33]

3.4 SwiftData

SwiftData je moderní framework pro správu perzistentních dat v aplikacích určených pro Apple platformy, který byl uveden na trh s iOS 17.0 a odpovídajícími verzemi souvisejících operačních systémů pro ostatní platformy společnosti. Framework kombinuje prověřenou technologii perzistence z původního frameworku Core Data s pokročilými možnostmi programovacího jazyka Swift, zejména využitím maker a asynchronního programování. To vývojářům umožňuje implementovat perzistentní úložiště dat s minimálním objemem kódu a bez nutnosti externích závislostí. Klíčovou výhodou SwiftData je jeho jednoduchost použití. K transformaci běžné třídy na perzistentní model stačí přidat makro `@Model`. Významnou předností je také hluboká integrace se SwiftUI frameworkem, která umožňuje deklarativní přístup k definici datového modelu a poskytuje efektivní mechanismy pro zobrazování a reaktivní aktualizaci dat v uživatelském rozhraní při změnách v úložišti. [34] [35]

V následujících podkapitolách jsou popsány hlavní aspekty frameworku SwiftData včetně kontejnerů a kontextu, datového modelu, vztahů mezi entitami, praktické práce s daty a srovnání s frameworkem Core Data.

3.4.1 Kontejnery a kontext

Architektura SwiftData je postavena na dvou základních komponentách, které společně zajišťují správu perzistentních dat v aplikaci. Tyto komponenty tvoří most mezi objektovým modelem aplikace a fyzickým úložištěm dat. [37]

ModelContainer představuje hlavního správce datového úložiště, který zodpovídá za propojení mezi datovými modely definovanými vývojářem a jejich fyzickým uložením na zařízení. Kontejner zajišťuje koherentní a efektivní provádění všech operací čtení a zápisu podle schématu poskytnutého vývojářem. Významnou funkcionalitou je automatická podpora synchronizace prostřednictvím frameworku CloudKit, která umožňuje sdílení dat mezi zařízeními uživatele. Při změnách datového schématu kontejner automaticky provádí migrace dat. Složitější transformace lze řídit pomocí předem definovaných migračních plánů. [36] [37]

ModelContext funguje jako pracovní prostor pro manipulaci s datovými objekty při běhu aplikace. Kontext udržuje objekty a jejich změny v operační paměti. Změny jsou fyzicky uloženy až po explicitním volání metody *save*. Tento přístup umožňuje optimalizaci výkonu prostřednictvím hromadného zpracování změn a podporuje pokročilé funkce jako historie změn s možností vrácení zpět nebo vypnutí automatického ukládání. Kontexty jsou vázány na vlákno, ve kterém byly vytvořeny, a jsou zpřístupněny uživatelskému rozhraní skrze prostředí SwiftUI. [36] [37]

3.4.2 Datový model

Datový model definuje strukturu, typ a způsob zpracování dat uchovávaných aplikací. Model je implementován jako běžná Swift třída označená speciálním makrem *@Model*, které automaticky zajišťuje propojení s perzistentním úložištěm. Díky tomuto makru je eliminována nutnost manuálního řešení operací ukládání a načítání dat – veškerá perzistence je řešena automaticky na pozadí. [38]

Modely označené makrem *@Model* se bezproblémově integrují s uživatelským rozhraním vytvořeným v SwiftUI. Systém automaticky sleduje změny vlastností objektů a v reálném čase aktualizuje odpovídající části uživatelského rozhraní, čímž se eliminuje

nutnost manuálního sledování změn nebo psaní dodatečného kódu pro zobrazování aktualizací. Framework také optimalizuje výkon seskupováním většího množství změn do jedné operace uložení, což současně zvyšuje bezpečnost a konzistenci dat. [38]

Datové modely podporují definici složitějších struktur včetně vztahů mezi objekty a validačních pravidel, například požadavek na jedinečnost určitých hodnot. To umožňuje vytváření komplexních datových architektur bez nutnosti implementace rozsáhlého pomocného kódu. [38]

3.4.3 Vztahy

Vztahy ve SwiftData reprezentují asociace mezi různými datovými modely, které umožňují modelování vzájemných propojení odpovídajících reálným scénářům – například vztah mezi studenty a školami nebo mezi autory a knihami. Framework SwiftData automaticky spravuje konzistenci těchto vztahů a zajišťuje, že data zůstávají vždy v platném a koherentním stavu napříč celou aplikací. [39]

Framework rozlišuje dva způsoby definice vztahů – odvozené a explicitní. Odvozené vztahy jsou automaticky odvozeny ze struktury datového modelu bez nutnosti dodatečné konfigurace. SwiftData je aplikuje pouze v případech, kdy lze garantovat bezpečnost operací. Například u povinných vlastností objektu se odvozování neprovádí, aby se předešlo situacím, kdy by nastavení hodnoty *nil* způsobilo pád aplikace. Explicitní vztahy se definují prostřednictvím makra *@Relationship*, které umožňuje přesné určení charakteru vazby a pravidel pro její chování. Odpovědnost za validitu modelů v tomto případě přebírá vývojář. [39]

Typy vztahů odpovídají standardním kardinalitám používaným v relačních databázích:

- **One-To-One** – každá instance jednoho modelu je spojena s právě jednou instancí druhého modelu,
- **One-To-Many** – jedna instance nadřazeného modelu může být asociována s množinou instancí podřízených modelů,
- **Many-To-One** – více instancí podřízených modelů odkazuje na jednu instanci nadřazeného modelu,
- **Many-To-Many** – obousměrné asociace mezi instancemi obou modelů bez hierarchického uspořádání. [40]

3.4.4 Práce s daty

Manipulace s daty ve SwiftData je založena na deklarativním přístupu, který umožňuje reaktivní propojení datových zdrojů s uživatelským rozhraním. Integrace se SwiftUI je realizována prostřednictvím specializovaných *property wrappers*, které automatizují správu dat a jejich zobrazování. [41]

Pro přístup k datovému kontextu se využívá *property wrapper* `@Environment`, který zpřístupňuje *ModelContext* pro ukládání, sledování a správu změn datových modelů v rámci konkrétního *view*. Načítání dat zajišťuje *property wrapper* `@Query`, který umožňuje deklarativní definici dotazů a automatické zobrazení kolekcí objektů přímo ve *view*, přičemž se stará o reaktivní aktualizace při změnách dat. [41]

Proces úpravy dat následuje přímočarý vzorec: objekty jsou vytvořeny nebo modifikovány prostřednictvím standardních Swift operací, následně se změny potvrdí voláním metody `context.save`. SwiftUI framework poté automaticky zajistí překreslení všech závislých *view* podle aktuálního stavu dat, čímž se udržuje synchronizace mezi datovým modelem a uživatelským rozhraním bez nutnosti explicitního řízení ze strany vývojáře. [41]

Praktickou ukázkou práce s daty lze vidět na obrázku číslo 14.

```
1 import SwiftUI
2 import SwiftData
3
4 struct ExampleView: View {
5     @Environment(\.modelContext) private var context
6     @Query(sort: \.name) private var items: [Item]
7
8     var body: some View {
9         List(items) { item in
10             Text(item.name)
11         }
12         .onAppear {
13             let newItem = Item(name: "Example")
14             context.insert(newItem)
15             try? context.save()
16         }
17     }
18 }
```

Obrázek 14 Práce s daty ve SwiftUI

3.4.5 Srovnání s Core Data

Framework Core Data je dlouhodobě využívaný objektově-relační mapovací framework (anglicky *ORM*, *Object-Relational Mapper*) s přibližně dvacetiletou historií vývoje, který nabízí komplexní funkcionalitu pro perzistenci. SwiftData byl ve srovnání představen jako

modernější alternativa postavená nad technologickými základy Core Data, která upřednostňuje jednoduchost použití a bezproblémovou integraci se Swift a SwiftUI frameworky prostřednictvím intuitivního API využívajícího makra a *property wrappers*. Zatímco Core Data poskytuje širší spektrum pokročilých funkcí a je vhodný pro komplexní aplikace s náročnými datovými požadavky, SwiftData se zaměřuje na vývojářskou efektivitu a rychlost implementace za cenu omezenější funkčnosti. Volba mezi oběma frameworky by měla vycházet ze specifických požadavků projektu. SwiftData je ideální pro nové aplikace s moderní architekturou, kde je prioritou jednoduchost a rychlý vývoj, zatímco Core Data zůstává preferovanou volbou pro projekty vyžadující pokročilé datové operace a maximální kontrolu nad perzistencí. [42] [43]

3.5 Core Image

Core Image je framework pro zpracování a analýzu obrazů vyvinutý společností Apple. Framework byl představen v roce 2005 v rámci macOS 10.4 Tiger jako součást QuartzCore frameworku. Hlavním cílem frameworku bylo umožnit výkonné, nedestruktivní zpracování obrazů s aplikací filtrů a transformací v téměř reálném čase přímo na úrovni grafického jádra systému. Architektura frameworku zajišťuje vysokou efektivitu prostřednictvím GPU akcelerace využívající technologii Metal, s možností využití CPU při nedostupnosti hardwarové akcelerace. Core Image je integrován s dalšími Apple frameworky jako AVFoundation nebo UIKit, což umožňuje bezproblémové zpracování snímků a videa přímo v aplikacích. [44]

Následující podkapitoly představují hlavní komponenty frameworku, a to vykreslovací kontext *CImageContext* zajišťující finální renderování upravených obrazů, třídu *CImage* reprezentující obrazová data a komponentu *CIFilter* určenou pro aplikaci efektů a transformací.

3.5.1 CImageContext

CImageContext představuje vykreslovací jádro frameworku Core Image, který převádí instrukce pro zpracování obrazu na finální pixelová data. Zatímco *CImage* a *CIFilter* definují pouze návod na zpracování, *CImageContext* provádí skutečné výpočty a vykreslení, nejčastěji do formátu *CGImage* pro zobrazení nebo do *CVPixelBuffer* pro video zpracování. Kontext může využívat GPU akceleraci prostřednictvím Metal nebo OpenGL pro vysoký výkon, případně CPU pro zajištění kompatibility. Z důvodu náročnosti vytváření nové

instance *CImageContext* je doporučeno jeho opakované použití, protože časté vytváření nových kontextů může výrazně snížit výkon aplikace, zejména při zpracování v reálném čase. [44]

3.5.2 CImage

CImage funguje jako virtuální reprezentace obrazu, která místo skutečných pixelových dat ukládá pouze instrukce popisující způsob vytvoření finálního obrazu. Tento přístup implementuje princip „líné evaluace“ (anglicky *lazy evaluation*) – skutečné vykreslení probíhá až při explicitním požadavku prostřednictvím *CImageContext*, což optimalizuje využití paměti a výpočetních zdrojů. *CImage* lze vytvořit z různých zdrojů včetně existujících obrazových formátů (*UIImage*, *CGImage*), souborů nebo přímo z pixelových dat. Všechny operace jsou nedestruktivní a zachovávají původní data beze změny. [44]

3.5.3 CFilter

CFilter představuje základní nástroj pro aplikaci efektů a transformací na obrazová data, který přijímá jeden nebo více *CImage* objektů jako vstup a produkuje nový *CImage* jako výstup. Filter nemodifikuje původní data, ale pouze aktualizuje instrukce pro zpracování, což umožňuje nedestruktivní editaci a efektivní řetězení více filtrů za sebou pro vytváření komplexních efektů. Core Image poskytuje rozsáhlou knihovnu předpřipravených filtrů kategorizovaných podle typu efektu (úpravy barev, rozmazání, zkreslení). Vývojáři mohou vytvářet vlastní filtry kombinováním existujících nebo implementací vlastních výpočetních jader pro specializované zpracování. [44]

3.6 Xcode

Xcode představuje oficiální integrované vývojové prostředí (anglicky *Integrated Development Environment* – zkráceně *IDE*) vyvíjené společností Apple určené pro vytváření aplikací pro všechny Apple platformy. Současně se jedná o jediný oficiálně podporovaný způsob, jak vyvíjet a distribuovat aplikace skrze App Store. Prostředí vzniklo v roce 2003 jako nástupce Project Builder, vývojového nástroje původně vyvinutého společností NeXT pro operační systém NeXTSTEP, který byl po akvizici NeXT společností Apple v polovině 90. let postupně přepracován do současné podoby. Xcode v současnosti zastává klíčovou roli v procesu vývoje aplikací pro Apple platformy, neboť představuje jediný oficiálně podporovaný způsob kompilace a distribuce aplikací prostřednictvím App Store. [45]

Architektura Xcode integruje textový editor, nástroje pro ladění, systémy správy závislostí a distribuční mechanismy do jednoho celku. Prostředí implementuje prediktivní

dokončování kódu využívající lokálně spuštěný model strojového učení trénovaný specificky pro jazyk Swift a knihovny od Apple. Funkcionalita Xcode zahrnuje integraci se systémy správy verzí, nástroje pro organizaci kódu a centralizovanou správu lokalizačních prostředků. Významnou částí Xcode je jeho integrace s frameworkem SwiftUI prostřednictvím *Previews*, která umožňuje interaktivní vývoj s náhledem uživatelského rozhraní v reálném čase. Ladící a testovací nástroje poskytují pokročilé diagnostické možnosti včetně breakpointů, krokování kódu a strukturované konzole pro výstup ladících informací. Tyto komponenty společně tvoří komplexní vývojovou platformu pokrývající celý životní cyklus vývoje aplikací od návrhu po distribuci. [45][46]

4 POROZITA

Porozita představuje základní fyzikální charakteristiku materiálu, která kvantifikuje míru prázdných prostor v jeho struktuře. Definuje se jako poměr objemu pórů k celkovému objemu materiálu a vyjadřuje se obvykle v procentech. Póry mohou být vyplněny plynem, kapalinou nebo jinou pevnou látkou. Jejich přítomnost a distribuce významně ovlivňují mechanické, tepelné a další vlastnosti materiálu. [7]

Póry v materiálech se vyskytují v širokém spektru velikostí, které například federace IUPAC (*International Union of Pure and Applied Chemistry*) charakterizuje do tří základních kategorií podle jejich rozměrů. Mikropóry dosahují velikosti menší než 2 nanometry, mezopóry se pohybují v rozmezí 2 až 50 nanometrů a makropóry překračují velikost 50 nanometrů. Kromě rozměrové charakteristiky je důležitá také prostorová distribuce pórů, která zahrnuje jak povrchové póry nacházející se na rozhraní materiálu s okolním prostředím, tak vnitřní póry umístěné uvnitř materiálové struktury. Klíčovým aspektem pórové struktury je dále konektivita, která určuje, zda jsou póry vzájemně propojené a tvoří propojenou síť umožňující pohyb tekutin, nebo zda existují jako izolované dutiny bez vzájemného spojení. [8]

Porozita významně ovlivňuje fyzikální a mechanické vlastnosti materiálů. Materiály s vysokou porozitou vykazují nižší hustotu a hmotnost ve srovnání s materiály s nízkou porozitou, které jsou charakteristické vyšší hustotou. Z mechanického hlediska vysoká porozita často vede ke snížení pevnosti a tuhosti materiálu v důsledku strukturálního oslabení způsobeného přítomností pórů. Porozita dále ovlivňuje tepelné vlastnosti – materiály s vysokou porozitou typicky vykazují nižší tepelnou vodivost, což může být výhodné pro tepelně izolační aplikace. Podobně jsou ovlivněny akustické vlastnosti, kde porézní struktury mohou vykazovat odlišnou absorpční kapacitu zvuku. Absorpční schopnosti materiálů pro kapaliny a plyny jsou přímo úměrné jejich porozitě, neboť póry poskytují prostor pro zadržení těchto látek. [7]

Pro kvantifikaci porozity se využívá spektrum analytických metod přizpůsobených typu materiálu a požadované přesnosti měření. Jednou ze základních metod je Archimédova metoda standardizovaná v normách DIN EN 623-2 a ASTM D792, která je vhodná pro stanovení porozity větší než 1 %. Postup zahrnuje sušení vzorku, odstranění vzduchu z pórů pod sníženým tlakem a následné ponoření do vody. Získané hodnoty umožňují výpočet skutečné hustoty pevného materiálu, objemové hustoty vzorku a porozity. [9] Mezi

pokročilejší metody patří například mikroskopické techniky, které poskytují informace o povrchu materiálů s vysokým rozlišením. Fyzikální adsorpce plynů, nejčastěji dusíku nebo oxidu uhličitého, umožňuje charakterizaci specifického povrchu a distribuce velikosti pórů. Rtuťová měření porozity využívají penetraci rtuti do pórové struktury pod tlakem a poskytují informace o pórech v rozsahu velikostí od 900 mikrometrů až po 4 nanometry. [8]

II PRAKTICKÁ ČÁST

5 ANALÝZA EXISTUJÍCÍCH MOBILNÍCH APLIKACÍ

V této kapitole je popsána analýza současného stavu mobilních aplikací využívaných v oblasti měření fyzikálních vlastností materiálů, kontroly kvality a porozity materiálu. Cílem této analýzy je získat komplexní přehled o existujících řešeních na trhu před zahájením návrhu a vývoje vlastní prototypové aplikace, což umožňuje identifikovat silné a slabé stránky současných přístupů, identifikovat mezery v dostupné funkcionalitě a definovat příležitosti pro inovativní řešení.

Metodika analýzy spočívala v systematickém vyhledávání aplikací na platformách App Store a Google Play prostřednictvím relevantních klíčových slov odpovídajících jednotlivým oblastem dané problematiky. Analýza využívala verze těchto platform v anglickém jazyce s využitím následujících klíčových slov: *measure, physical values, ar measure, photo measure, quality, quality control, manufacturing, porosity, porosity measurement, porosity analysis*.

5.1 Mobilní aplikace pro měření fyzikálních vlastností

Mobilní aplikace pro měření fyzikálních vlastností představují v současnosti rychle se rozvíjející segment, který využívá pokročilých senzorů a technologií implementovaných v moderních mobilních zařízeních. Tyto aplikace nabízejí alternativní přístup k tradičním měřicím nástrojům a umožňují provádět široké spektrum fyzikálních měření přímo prostřednictvím mobilního telefonu nebo tabletu.

Pro pochopení současného stavu a identifikaci klíčových funkcionalit těchto aplikací byl analyzován jeden příklad z kategorie zabudovaných aplikací a jeden příklad ze specializovaných aplikací třetích stran.

Pro první kategorii, tedy zabudované aplikace operačních systémů, byla vybrána aplikace **Measure** pro platformu iOS. Aplikace díky rozšířené realitě umožňuje provádět přesná měření přímo prostřednictvím mobilního telefonu. Základní funkcionalita umožňuje měření délky, šířky a výšky reálných objektů prostřednictvím kamery zařízení. Na zařízeních vybavených LiDAR skenerem jsou poskytovány pokročilé funkce včetně vertikálních a hranových vodících linií pro zvýšení přesnosti měření. Mezi specifické možnosti aplikace patří automatická detekce obdélníkových objektů s výpočtem jejich rozměrů a ploch, měření výšky osob a vedení historie provedených měření. [51]

Druhou kategorii reprezentují specializované aplikace třetích stran dostupné prostřednictvím distribučních platforem, z nichž byla pro analýzu vybrána aplikace s názvem **phyphox** vyvinutá RWTH Aachen University. Aplikace poskytuje komplexní přístup k využití senzorů mobilních zařízení pro fyzikální experimenty a měření. Prostřednictvím aplikace je umožněn přímý přístup k širokému spektru senzorů včetně akcelerometru, magnetometru, gyroskopu, barometrického čidla a GPS modulu. Aplikace obsahuje přednastavené experimenty pro okamžité použití a pokročilé funkce jako vzdálené ovládání prostřednictvím webového rozhraní nebo možnost vytváření vlastních experimentálních protokolů. [52][53]

Na základě provedené analýzy lze konstatovat, že mobilní aplikace pro měření fyzikálních vlastností představují standardní výbavu současných mobilních systémů s širokým potenciálem využití v oblasti měření.

5.2 Mobilní aplikace pro kontrolu kvality ve výrobě

Kontrola kvality představuje kritický aspekt výrobních procesů. Při analýze mobilních aplikací zaměřených na tuto problematiku bylo identifikováno značné množství řešení s různými specifickými zaměřenými a funkcionalitami. Spektrum dostupných aplikací se liší v závislosti na jejich cílové skupině a specifických požadavcích jednotlivých odvětví. Byla identifikována kategorie aplikací, které jsou vyvinuty pro konkrétní firemní procesy nebo organizace, přičemž zůstávají veřejně dostupné prostřednictvím standardních distribučních platforem.

Mezi konkrétní příklady aplikací této kategorie patří **Infor LN Quality Inspections**, která umožňuje zaznamenávání výsledků inspekčních testů přímo v místě kontroly včetně výrobních hal a skladů, podporuje offline režim a připojení fotografické dokumentace. [54] Další aplikací je **NEOGAGE Quality Control**, která je zaměřená na posuzování kvality produktů a dodávek s funkcemi pro evidenci produktů, plánování inspekcí, vytváření personalizovaných inspekčních karet, správu uživatelských oprávnění a vedení statistik provedených kontrol. [55] Při kontrole dostupnosti bylo zjištěno, že aplikace od Neogage není v České republice prostřednictvím App Store dostupná, ačkoliv oficiální stránky aplikace tuto skutečnost nezmiňují.

Výsledky analýzy potvrzují, že mobilní aplikace pro kontrolu kvality představují rozvinutou kategorii s širokou škálou specializovaných řešení přizpůsobených specifickým potřebám výrobních procesů a firemních požadavků.

5.3 Mobilní aplikace pro měření porozity materiálů

Analýza mobilních aplikací zaměřených na problematiku měření porozity materiálů neodhalila veřejně dostupné aplikace, které by se přímo zabývaly výpočtem nebo analýzou porozity materiálů.

Je však nutné vzít v úvahu, že specializované aplikace tohoto typu mohou být distribuovány prostřednictvím alternativních kanálů, které nejsou veřejně přístupné. U iOS platformy se jedná především o distribuci prostřednictvím *TestFlight* pro interní testování organizací, *Enterprise Distribution Program* určený pro vnitřní použití ve firmách s více než 100 zaměstnanci, nebo *Custom Apps for Business* distribuované přímo konkrétním organizacím prostřednictvím *Apple Business Manager* programu. Tyto distribuční metody umožňují vývoj a nasazení specializovaných aplikací mimo veřejný App Store, což může vysvětlovat absenci podobných řešení ve veřejně dostupném prostoru. [50] Absence veřejně dostupných aplikací zaměřených na porozitu materiálů může představovat potenciální tržní příležitost.

5.4 Shrnutí analýzy existujících řešení

Provedená analýza existujících mobilních aplikací odhalila tři klíčové poznatky pro vývoj prototypové aplikace pro detekci porozity pneumatik. Aplikace zaměřené na měření fyzikálních vlastností představují rozvinutou kategorii s širokým zastoupením na mobilních platformách, což naznačuje dostupnost vhodných vývojových nástrojů, knihoven a dokumentace pro implementaci měřicích funkcionalit. Aplikace pro kontrolu kvality ve výrobě tvoří specializovanější segment často přizpůsobený specifickým firemním procesům. Jejich přístup k evidenci, správě a analýze výrobních dat může sloužit jako inspirace pro návrh prototypové aplikace. Absence veřejně dostupných aplikací zaměřených na porozitu materiálů představuje prostor pro vývoj inovativního řešení a potvrzuje potenciální přínos vyvinutého prototypového řešení pro oblast detekce porozity v gumárenském průmyslu.

6 SPECIFIKACE APLIKACE

Porozita materiálů je v řadě průmyslových aplikací, včetně výroby pneumatik v gumárenském průmyslu, jedním z klíčových ukazatelů kvality finálního produktu. Přítomnost nežádoucí porozity v materiálu pneumatiky může negativně ovlivnit mechanické vlastnosti, životnost a také bezpečnost výrobku. V současné průmyslové praxi se pro kvantifikaci porozity uplatňují převážně dva přístupy: subjektivní vizuální hodnocení realizované zkušenými odborníky nebo laboratorní analytické metody, které jsou spojeny se značnými časovými a finančními nároky. Tyto metody mají tedy své zjevné nedostatky, a proto je nezbytné hledat nové, efektivnější řešení.

Hlavním cílem následujících kapitol praktické části práce je navržení, implementování, testování a zhodnocení prototypu specializované mobilní aplikace pro platformu iOS, která využije vestavěnou kameru mobilních zařízení Apple a algoritmy pro analýzu obrazu k odhadu porozity vzorků pneumatik. Návrh aplikace je založen na moderních technologiích společnosti Apple, konkrétně programovacím jazyku Swift, frameworku SwiftUI pro definici uživatelského rozhraní a knihovně SwiftData pro správu a perzistenci dat. Tím je zajištěna vysoká kompatibilita s platformou iOS a současně je umožněna efektivní údržba a flexibilní rozšiřitelnost řešení do budoucna.

6.1 Cíle aplikace

Cílem vývoje aplikace je vytvoření moderního nástroje pro přesný odhad porozity vzorků pneumatik, který nahradí současné neefektivní metody založené na subjektivním hodnocení pracovníků nebo časově a finančně náročných laboratorních analýzách. Aplikace má být navržena jako komplexní řešení umožňující nejen samotnou analýzu porozity z fotografií řezů pneumatik prostřednictvím zpracování obrazu, ale také systematickou správu všech provedených měření včetně možnosti doplnění relevantních metadat. Nedílnou součástí aplikace je také zavedení funkce exportování výsledků měření do strukturovaného formátu CSV (anglická zkratka pro *Comma-Separated Values*), která umožní snadnou integraci získaných dat do stávajících výrobních systémů, následné zpracování nebo archivaci. Uživatelské rozhraní má být implementováno v souladu s designovými standardy tak, aby byla zajištěna intuitivní obsluha, responzivní chování a snadná budoucí rozšiřitelnost aplikace.

6.2 Popis aplikace

Aplikace pro odhad porozity vzorků pneumatik umožňuje uživatelům, zejména technikům kontroly kvality, rychle a efektivně analyzovat fotografie řezů vzorků vytlačované směsi pořízené přímo ve výrobním procesu. Uživatel má možnost buď přímo pořídit novou fotografii vzorku pomocí vestavěné kamery zařízení, nebo použít již existující snímek uložený v zařízení nebo cloudovém úložišti. Po načtení fotografie uživatel definuje oblast zájmu výběrem oblasti řezu vzorku pneumatiky, čímž se zajistí analýza relevantní části fotografie. Aplikace poté pomocí algoritmů vyhodnotí porozitu vzorku pneumatiky.

Výsledkem provedené analýzy je výpočet procentuálního zastoupení pórů vzhledem k celkové ploše materiálu v rámci definované analyzované oblasti. Tento výsledek lze přímo interpretovat jako kvantitativní odhad míry porozity daného vzorku, což je v průmyslové i laboratorní praxi považováno za klíčový parametr při hodnocení strukturální kvality materiálů. Porozita zásadně ovlivňuje mechanické, tepelně-izolační a další fyzikální vlastnosti materiálů. Její správné určení je tudíž nezbytné pro zajištění požadované úrovně jakosti výrobků. [7]

S ohledem na praktické využití výsledků analýzy v reálných výrobních procesech je žádoucí, aby vyvíjená aplikace byla vybavena intuitivním a uživatelsky přívětivým rozhraním. Rozhraní musí umožňovat nejen snadné ovládání, ale rovněž rychlou a jednoznačnou interpretaci získaných dat. V tomto kontextu je požadováno, aby prezentace výsledků byla přehledná, strukturovaná a dostatečně vizuálně jasná, čímž se významně usnadní rychlé a správné rozhodování o kvalitě analyzovaného výrobku přímo v rámci výrobních nebo kontrolních procesů.

Výstupem každé jednotlivé analýzy jsou čitelné numerické údaje, které poskytují přesnou informaci o porozitě v rámci analyzované oblasti. Tyto údaje jsou dále během analýzy doplněny o vizuální reprezentaci v podobě analyzovaného snímku, na němž jsou detekované póry jasně a výrazně označeny. Toto grafické zvýraznění slouží nejen ke zvýšení transparentnosti celého procesu hodnocení, ale i k usnadnění ověření správnosti identifikace a kvantifikace porozity. Výsledky každé analýzy lze následně exportovat do souboru ve formátu CSV, a to buď pro jednotlivý záznam, nebo seskupeně podle konkrétní směsi či výrobní linky. Tato výstupní data usnadňují sdílení dat, jejich dlouhodobou archivaci nebo propojení s jinými nástroji používanými v rámci řízení kvality ve výrobním podniku.

Celý postup od získání obrazu přes samotnou analýzu až po interpretaci výsledků je navržen s ohledem na minimalizaci časové náročnosti a maximalizaci opakovatelnosti výsledků při současném zachování vysoké míry přesnosti a spolehlivosti. Důraz je kladen zejména na to, aby proces byl co nejméně závislý na subjektivním posouzení operátora a aby bylo možné jednotlivé kroky metodiky snadno standardizovat a validovat v různých výrobních podmínkách.

6.3 Funkcionální požadavky

V tabulce číslo 1 jsou popsány funkcionální požadavky na aplikaci. Každý požadavek je formulován tak, aby jeho splnění bylo jednoznačně ověřitelné v průběhu testování aplikace.

Tabulka 1 Funkcionální požadavky na aplikaci

ID	Popis požadavku
F01	Aplikace umožní pořízení nové fotografie pomocí vestavěné kamery zařízení.
F02	Aplikace umožní načtení existující fotografie ze zařízení nebo cloudového úložiště.
F03	Uživatel může vybrat oblast zájmu (řez pneumatiky) na načtené fotografii.
F04	Aplikace automaticky analyzuje vybranou oblast a detekuje póry (porozitu).
F05	Aplikace vypočítá procentuální zastoupení porozity vůči celkové ploše materiálu.
F06	Výsledky analýzy jsou zobrazeny numericky s přesností na tři desetinná místa.
F07	Na analyzovaném snímku jsou detekované bubliny vizuálně označeny.
F08	Uživatel může k analýze doplnit informace o použitém materiálu.
F09	Uživatel může k analýze doplnit informace o použité výrobní lince.
F10	Uživatel může k analýze doplnit libovolnou textovou poznámku.
F11	Každá provedená analýza je automaticky uložena do databáze aplikace.
F12	Domovská obrazovka zobrazuje seznam všech provedených analýz.
F13	Uživatel může vyhledávat v seznamu analýz podle klíčových slov a tím filtrovat analýzy například podle datumu pořízení, materiálu, výrobní linky nebo poznámky.
F14	Uživatel může zobrazit detail jednotlivé analýzy se všemi uloženými údaji.
F15	Uživatel může exportovat data jednotlivého záznamu do CSV formátu.
F16	Uživatel může exportovat všechna data pro konkrétní směs do CSV formátu.
F17	Uživatel může exportovat všechna data pro konkrétní výrobní linku do CSV formátu.
F18	Uživatel může smazat vybranou analýzu z aplikace.

6.4 Nefunkcionální požadavky

V tabulce číslo 2 jsou popsány nefunkcionální požadavky na aplikaci. Každý požadavek je formulován tak, aby jeho splnění bylo jednoznačně ověřitelné v průběhu testování aplikace.

Tabulka 2 Nefunkcionální požadavky na aplikaci

ID	Popis požadavku
NF01	Aplikace je v anglickém jazyce.
NF02	Analýza jedné fotografie je dokončena do 5 sekund na aktuální 16. generaci iPhone zařízení.
NF03	Aplikace funguje offline bez nutnosti internetového připojení.
NF04	Velikost aplikace nepřesáhne 100 MB po instalaci.
NF05	Exportované CSV soubory jsou kompatibilní s programy Apple Numbers, Microsoft Excel a Google Sheets.
NF06	Aplikace je responzivní a reaguje na uživatelské akce do 2 sekund.
NF07	Aplikace je optimalizována pro různé velikosti obrazovek zařízení iPhone.
NF08	Aplikace nepotřebuje externí závislosti nebo síťové služby pro základní fungování.
NF09	Přesnost detekce porozity má za optimálních světelných podmínek odchylku maximálně ± 1 % oproti referenčnímu laboratornímu měření.

7 NÁVRH APLIKACE

Tato kapitola je zaměřena na návrh aplikace jako celku. V úvodu kapitoly je nejprve objasněno, jaké postupy a metodiky byly při návrhu aplikace zvoleny a jaký význam měly v rámci dosažení cílů práce. Návrh aplikace je následně rozdělen do dvou hlavních oblastí, a to na návrh drátěných modelů uživatelského rozhraní a návrh datového modelu, který zabezpečuje správu perzistentních dat.

V rámci návrhu drátěných modelů bylo cílem vytvořit intuitivní a přehledné uživatelské rozhraní, které by umožnilo uživatelům snadno a rychle provádět všechny požadované operace s minimálním počtem interakcí a bez zbytečných kroků navíc. Návrh byl zaměřen na nalezení co nejefektivnějšího způsobu, jak uspořádat jednotlivé funkce a obrazovky aplikace tak, aby byla zachována logická návaznost pracovních postupů, aby uživatelé nebyli zatěžováni složitými navigačními strukturami a aby byla minimalizována pravděpodobnost chybného použití. Výsledný návrh drátěných modelů byl výsledkem několika iterací, během nichž byly analyzovány různé scénáře použití a hledána optimální řešení z pohledu UX (anglická zkratka pro *User Experience*, česky uživatelský prožitek).

V úvodních fázích projektu bylo nezbytné definovat nejen vizuální strukturu aplikace, ale také její funkční logiku a datové struktury, které budou zajišťovat správu a ukládání všech relevantních informací. Vzhledem k tomu, že aplikace pracuje s obrazovými daty, bylo nutné navrhnout vhodné strategie pro jejich správu a manipulaci tak, aby byla zachována kvalita fotografií při současném efektivním využití úložného prostoru zařízení.

7.1 Postup a metodiky

Pro návrh uživatelského rozhraní a celkovou strukturu aplikace byl zvolen přístup prototypování pomocí drátěných modelů (anglicky *wireframes*). Tento postup umožnil nejen rychlé ověření konceptu a usnadnil následnou implementaci, ale také poskytl jasný základ pro identifikaci hlavních entit systému a vztahů mezi nimi. Analýza drátěných modelů tak přímo přispěla k návrhu datového modelu aplikace a definování požadavků na jednotlivé komponenty řešení.

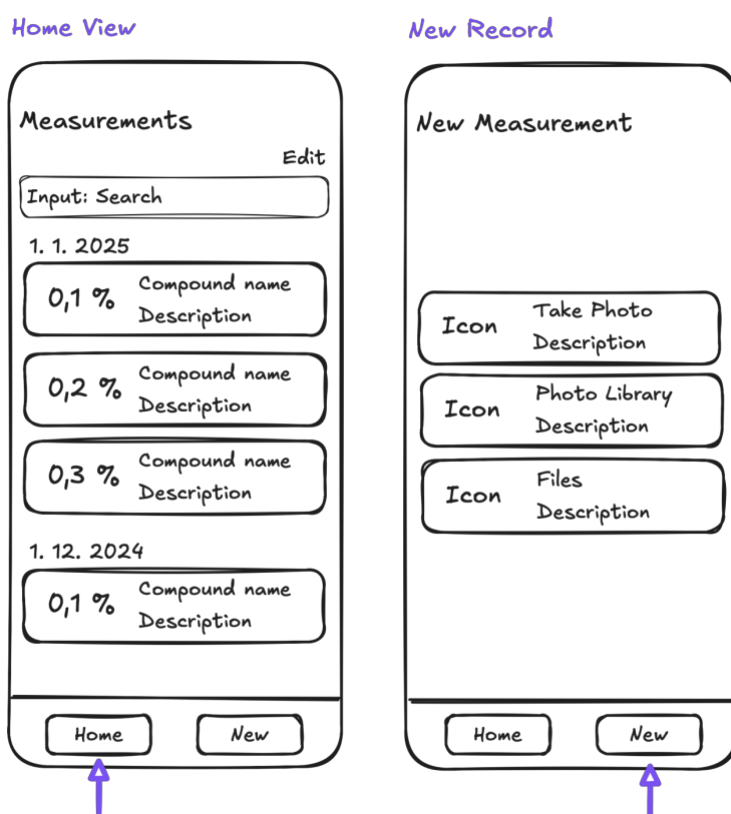
7.2 Návrh drátěných modelů

V rámci návrhu aplikace byla zpracována sada drátěných modelů, které slouží jako vizuální návrh jednotlivých obrazovek aplikace. Zároveň představují logiku a postupné kroky interakce uživatele se systémem. Cílem této kapitoly je detailně popsat jednotlivé obrazovky

a jejich vzájemné návaznosti tak, aby bylo možné jasně komunikovat strukturu uživatelského rozhraní.

7.2.1 Hlavní obrazovky řízené strukturou Tab View

Základní navigace v aplikaci je realizována prostřednictvím spodní lišty (anglicky *Tab View*), která obsahuje dvě hlavní obrazovky: domovskou obrazovku a obrazovku pro přidání nového záznamu. Tyto hlavní obrazovky je možné vidět na obrázku číslo 15.



Obrázek 15 Návrhy hlavních obrazovek aplikace

V levé části obrázku číslo 15 je domovská obrazovka (označena *Home View*), která je uživateli zobrazena ihned po spuštění aplikace. Na této obrazovce se nachází seznam všech uložených měření. Záznamy jsou seskupeny podle data vytvoření. Každý záznam obsahuje hodnotu vypočítané porozity, název použité směsi a volitelný popis. Navíc je zde naznačena možnost filtrování a vyhledávání v seznamu podle data vytvoření, poznámky, použité směsi nebo výrobní linky. Součástí této obrazovky je také tlačítko *Edit*, které umožňuje uživateli odstranit existující záznamy. Domovská obrazovka představuje hlavní rozcestník aplikace a poskytuje rychlý přístup ke všem historickým údajům.

V pravé části obrázku číslo 15 se pod označením *New Measurement* nachází obrazovka pro přidání nového záznamu, která umožňuje uživateli zahájit proces přidání nové fotografie vzorku pneumatiky do aplikace. Je zde zobrazena trojice karet, z nichž každá reprezentuje jinou možnost zdroje vstupní fotografie:

- pořízení nové fotografie zabudovanou kamerou zařízení,
- výběr existující fotografie z galerie,
- výběr existující fotografie ze souborového systému.

Tato jednoduchá a přehledná struktura zlepšuje UX při procesu rozhodování uživatele.

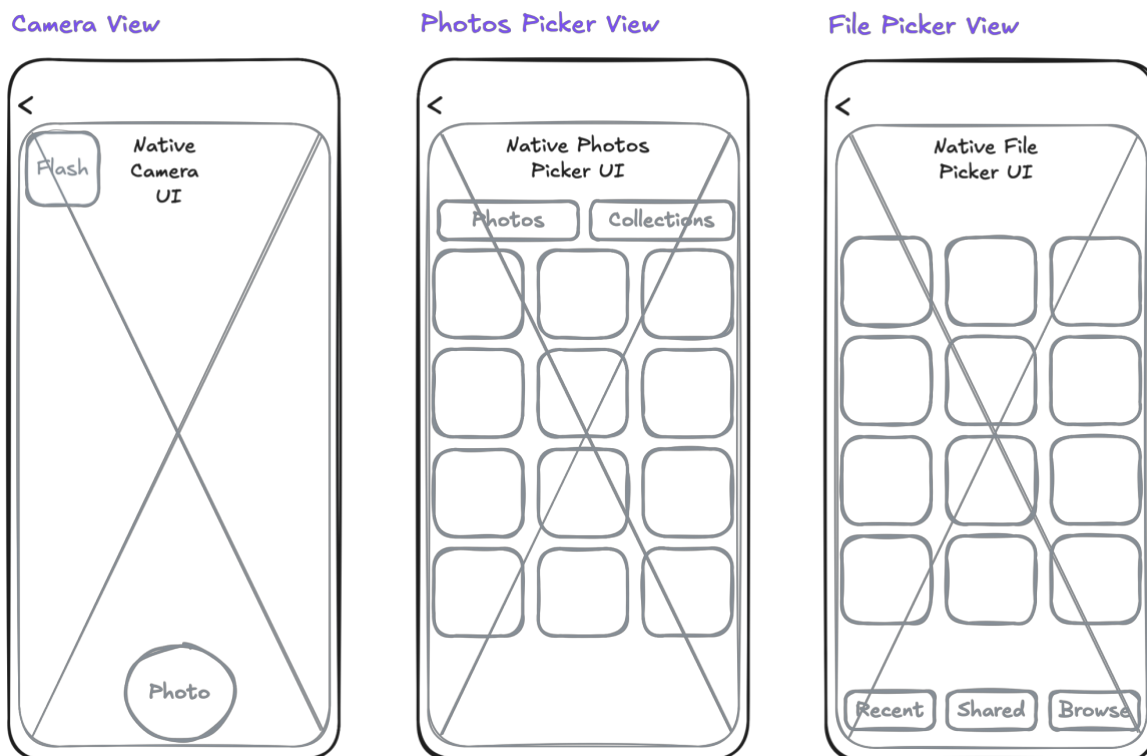
7.2.2 Volba zdroje vstupní fotografie

V závislosti na zdroji vstupní fotografie pro vytvoření nového záznamu je následně uživateli zobrazeno příslušné rozhraní. Toto rozhraní může nabývat jedné ze tří podob. Každá odpovídá jinému způsobu pořízení nebo výběru fotografie:

- zabudované systémové uživatelské rozhraní fotoaparátu,
- zabudované systémové uživatelské rozhraní pro výběr fotografie z galerie,
- zabudované systémové uživatelské rozhraní pro výběr fotografie ze souborového systému.

Všechna tato rozhraní jsou integrována pomocí zabudovaných komponent systému iOS a jsou vhodně nakonfigurována tak, aby splňovala specifické potřeby aplikace. V případě rozhraní fotoaparátu je funkcionality omezena výhradně na pořízení fotografie (pořízení videozáznamu není umožněno). Tento přístup zajišťuje, že uživatel nebude vyrušen nabídkou nerelevantních funkcí a celý proces pořízení vstupního snímku bude maximálně efektivní. Rozhraní pro výběr existující fotografie z galerie a ze souborového systému jsou rovněž přizpůsobena potřebám aplikace. Konkrétně jsou nakonfigurována tak, aby uživateli zobrazovala pouze relevantní soubory vhodné pro analýzu, přičemž je prováděno filtrování podle typu souboru. Tím je zajištěno, že nebude možné omylem vybrat nekompatibilní soubor.

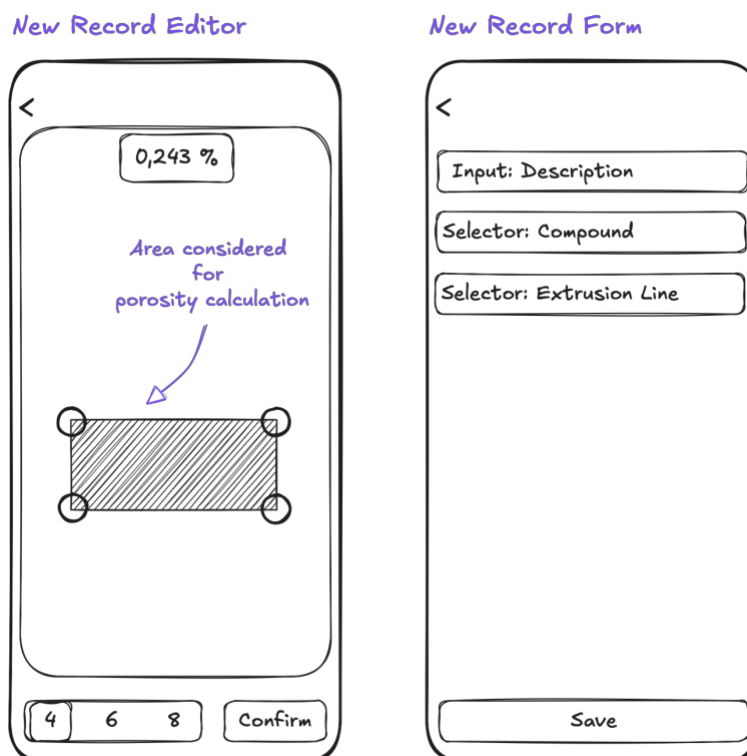
Jelikož všechna tato rozhraní využívají standardní zabudovaná rozhraní systému iOS, jejich chování odpovídá očekávání uživatelů, což zajišťuje konzistentní uživatelský zážitek. Návrh všech tří variant výběru zdroje vstupní fotografie je znázorněn na obrázku číslo 16.



Obrázek 16 Návrhy obrazovek pro zvolení vstupní fotografie

7.2.3 Přidání záznamu

Po výběru nebo pořízení fotografie je uživatel přesměrován do procesu přidání nového záznamu měření, který se skládá ze dvou na sebe navazujících kroků. Tyto kroky jsou znázorněny na obrázku číslo 17.



Obrázek 17 Návrhy obrazovek pro vytvoření nového záznamu

Prvním krokem je obrazovka sloužící pro výběr relevantní části obrazu, na které je současně provedena analýza porozity. Návrh této obrazovky je součástí obrázku číslo 17 v jeho levé části. Uživatel má možnost ohraničit oblast zájmu pomocí čtyř, šesti nebo osmi bodů. Body lze volně přesouvat tažením prstem. Výchozí nastavení jsou čtyři body. Vybraný obdélník je vizuálně zobrazen přes fotografii, přičemž vyplněná oblast reprezentuje část snímku, která bude analyzována. Po dokončení výběru klikne uživatel na tlačítko s textem *Confirm* a tím je přesunut na další krok.

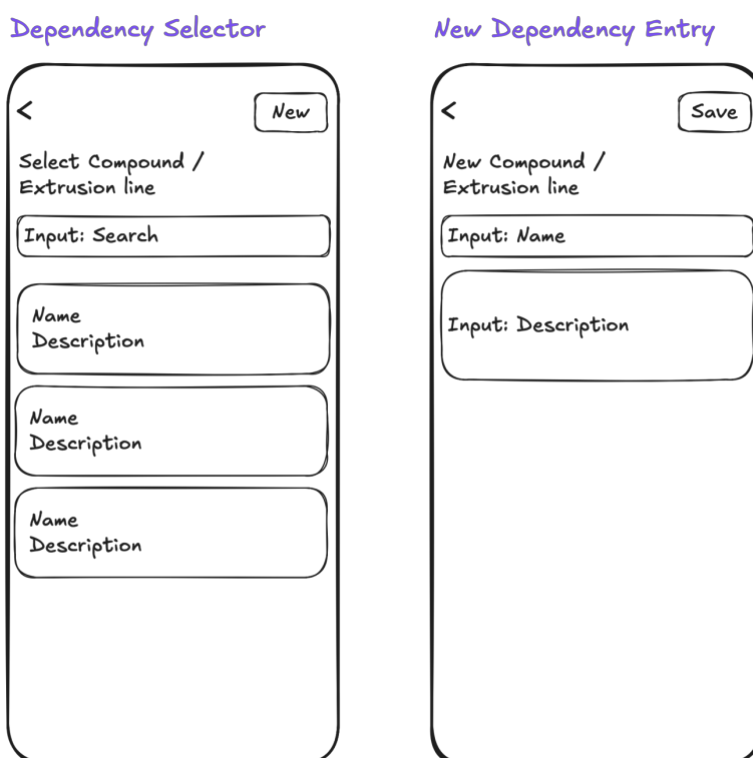
V druhém kroku, jehož návrh je zobrazen v pravé části obrázku číslo Obrázek 17, je uživatel vyzván k doplnění dodatečných informací o měření. Návrh této obrazovky je složen z:

- textového pole pro přidání poznámky,
- volby směsi ze seznamu existujících (případně vytvoření nové),
- volby výrobní linky ze seznamu existujících (případně vytvoření nové),
- tlačítko pro uložení nového záznamu.

Formulář je navržen s důrazem na jednoduchost a přehlednost, aby proces přidání nového záznamu byl co nejjednodušší a nejrychlejší.

7.2.4 Vytvoření nové směsi nebo výrobní linky

Při vyplňování formuláře nového záznamu je uživateli umožněno nejen vybrat existující směs či výrobní linku, ale také vytvořit směs nebo výrobní linku zcela novou. Tento proces je pro směsi i pro výrobní linky totožný a je znázorněn na obrázku číslo 18.



Obrázek 18 Návrh obrazovky pro vytvoření doplňujících datových struktur

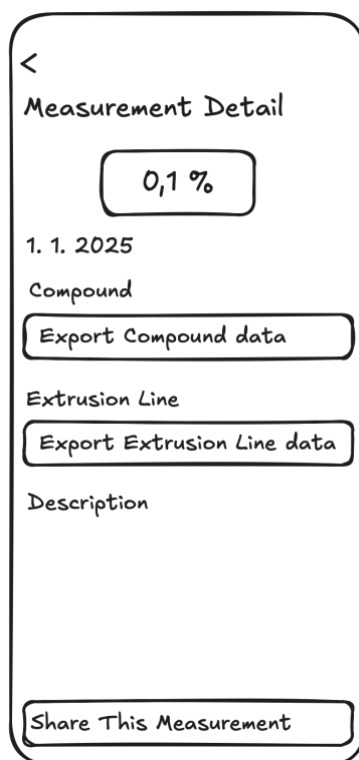
Po stisknutí selektoru směsi nebo výrobní linky je uživateli zobrazen seznam existujících položek, včetně možnosti v nich vyhledávat pomocí textového pole v horní části obrazovky. Návrh tohoto rozhraní je v levé části obrázku číslo 18. Každá položka je zobrazena s názvem a popisem a uživatel výběr provede stisknutím požadované karty.

Pokud požadovaná položka v seznamu neexistuje, může uživatel vytvořit novou pomocí tlačítka *New*, které otevře jednoduchý formulář s poli pro název a popis. Po uložení nové položky je automaticky provedeno její zvolení a návrat na předchozí obrazovku. Tím je zajištěna plynulost celého procesu bez nutnosti složitého přecházení mezi obrazovkami.

7.2.5 Obrazovka detailu záznamu

Každý vytvořený záznam lze v aplikaci zobrazit v samostatné detailní obrazovce, která uživateli poskytuje kompletní přehled o daném měření. Tato obrazovka představuje významný prvek aplikace, protože kombinuje veškerá podstatná data vztahující se ke konkrétnímu záznamu a současně umožňuje jejich efektivní správu, sdílení a rozšířenou analýzu. Přístup k detailu je dostupný výběrem konkrétního záznamu ze seznamu na domovské obrazovce. Návrh uživatelského rozhraní této obrazovky je znázorněn na obrázku číslo 19.

Record Detail



Obrázek 19 Návrh obrazovky
pro zobrazení detailního
pohledu na existující
záznam

Po otevření detailu záznamu jsou uživateli zobrazeny všechny relevantní informace, které byly při jeho vytvoření uloženy. Mezi tyto informace patří vypočítaná hodnota porozity, datum vytvoření měření, název použité směsi, název výrobní linky a popis.

Další funkcionalitou dostupnou v rámci detailu záznamu je možnost exportovat naměřené hodnoty. Pro exportování má uživatel tři možnosti, které rozhodují o kontextu exportovaných dat:

- exportování všech dat pro danou směs,
- exportování všech dat pro danou výrobní linku,
- exportování dat aktuálně zvoleného měření.

Implementace funkcionality pro exportování dat představuje významný prvek pro zajištění flexibility a interoperability systému. Možnost výběru různých kontextů exportu umožňuje uživatelům kontrolovat, jaká data budou exportována, a přizpůsobit tak export konkrétním analytickým potřebám. Export lze díky tomu efektivně integrovat s externími systémy a zvýšit praktickou využitelnost aplikace ve výrobním prostředí.

7.3 Návrh datového modelu

Proces návrhu datového modelu navazuje na předchozí návrh drátěných modelů, díky kterému bylo možné přesně vymezit jednotlivé obrazovky aplikace a jejich vzájemné propojení. Na základě vizualizace hlavních uživatelských scénářů bylo možné identifikovat klíčové objekty systému a stanovit odpovídající datové struktury.

V této podkapitole je představen návrh datového modelu aplikace. Je popsána struktura datového modelu včetně identifikace jednotlivých entit, jejich atributů a vzájemných vztahů, a je objasněna volba konkrétního uspořádání modelu s ohledem na požadavky systému a efektivitu správy dat. Tento návrh tvoří základní model, zachycující, jak budou data v rámci aplikace strukturována, ukládána a dále zpracovávána.

Aby bylo možné aplikaci efektivně využívat, je nezbytné navrhnout datový model, který splní následující požadavky:

- musí podporovat uložení všech provedených měření,
- musí umožnit přiřazení výrobních specifikací k jednotlivým záznamům, konkrétně identifikaci použité směsi a výrobní linky,
- musí umožnit efektivní vyhledávání a filtrování záznamů na domovské obrazovce podle zvolených kritérií,
- musí umožnit rozšíření o další doplňující informace, například datum a čas pořízení fotografie či uživatelské poznámky,
- musí zajistit konzistenci a integritu dat při editaci a mazání záznamů.

Na základě analýzy požadavků a návrhu drátěných modelů byly identifikovány tři hlavní entity: *Record*, *Compound* a *Extrusion Line*. Každá z těchto entit plní specifické role, které jsou podrobněji popsány v následujících podkapitolách.

7.3.1 Record

Record představuje jednotlivý záznam o provedené analýze fotografie. Obsahuje klíčové informace o fotografii, vypočtené hodnotě porozity a vazby na další entity. Je navržen tak, aby uchovával následující data:

- *id* – unikátní identifikátor zajišťující jednoznačnost každého záznamu,
- *dateCreated* – datum a čas vytvoření záznamu,
- *note* – volitelná poznámka uživatele vztahující se k dané analýze,
- *compound* – odkaz na entitu *Compound*, reprezentující použitou směs,
- *extrusionLine* – odkaz na entitu *ExtrusionLine*, reprezentující výrobní linku,
- *porosityValue* – výsledná hodnota porozity vyjádřená jako desetinné číslo,
- *originalImageData* – binární data původní pořízené fotografie,
- *croppedImageData* – binární data fotografie po oříznutí, tedy oblasti, která byla analyzována,
- *verticesRaw* – binární reprezentace souřadnic bodů použitých k definici vybrané oblasti v obraze,
- *verticesCount* – počet bodů použitých při výběru oblasti (4, 6 nebo 8).

Tato struktura umožňuje nejen uložení samotných výsledků analýz, ale také zachování všech podpůrných informací důležitých pro jejich zpracování a interpretaci.

7.3.2 Compound

Compound reprezentuje druh směsi, ze které je daný vzorek pneumatiky vyroben. Entita je navržena tak, aby obsahovala:

- *id* – unikátní identifikátor zajišťující jednoznačnost každého záznamu,
- *dateCreated* – datum vytvoření záznamu o směsi,
- *name* – název směsi,
- *note* – popis nebo poznámka vztahující se ke směsi.

Popsaná struktura umožňuje například seskupení výsledků měření podle použitých směsí, což je následně umožňuje snadno analyzovat a porovnávat.

7.3.3 Extrusion Line

Extrusion Line slouží k evidenci výrobních linek, na kterých byly vzorky pneumatik vyrobeny. Každá výrobní linka má přiřazeny následující údaje:

- *id* – unikátní identifikátor zajišťující jednoznačnost každého záznamu,
- *dateCreated* – datum vytvoření záznamu o lince,
- *name* – název výrobní linky,
- *note* – poznámka nebo popis dané výrobní linky.

Podobně jako u entity *Compound* je i zde cílem umožnit kategorizaci výsledků a identifikaci souvislostí mezi výrobními podmínkami a dosaženou porozitou.

7.3.4 Vztahy mezi entitami

Vztahy mezi jednotlivými entitami jsou navrženy s ohledem na jednoduchost i flexibilitu datového modelu. Každý *Record* může mít přiřazen jeden *Compound* a jednu *Extrusion Line*. Vazby mezi *Record* a *Compound* a mezi *Record* a *Extrusion Line* jsou definovány jako volitelné, což umožňuje zaznamenávat i takové analýzy, kde nejsou všechny výrobní údaje dostupné.

Takto navržená struktura poskytuje pevný základ pro snadnou správu dat, efektivní implementaci funkcí jako je filtrování výsledků podle směsi nebo výrobní linky, a zároveň umožňuje budoucí rozšíření aplikace o další funkce bez zásadního zásahu do existujícího modelu.

8 VÝVOJ APLIKACE

V této kapitole je popsán proces implementace iOS aplikace. Postupně jsou popsány použité metodiky, vytvoření a konfigurace projektu, architektura řešení, implementace datového modelu, uživatelského rozhraní a klíčových funkcionalit aplikace, které zahrnují analýzu porozity a export výsledků.

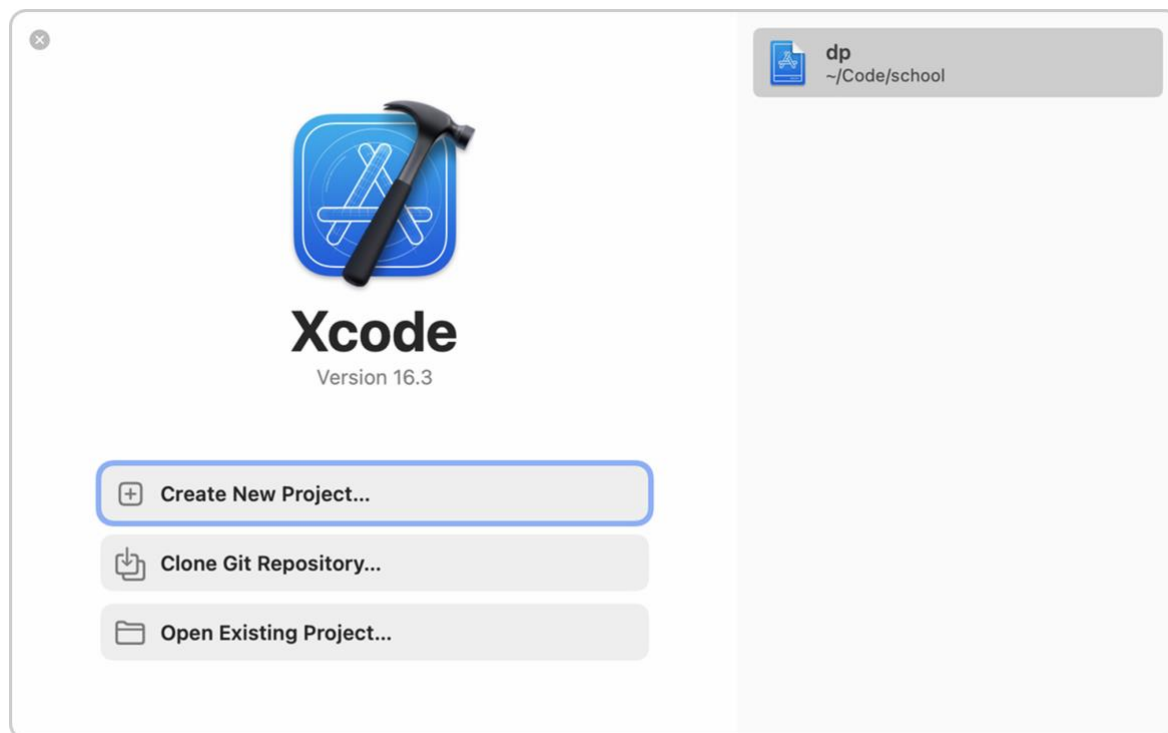
8.1 Postupy a metodiky

Vývoj aplikace byl realizován iterativním způsobem v souladu s principy agilního přístupu. Konkrétně se jednalo o přístup inspirovaný metodikou inkrementálního vývoje. Inkrementální vývoj je založen na principu, že se nejdříve tvoří počáteční implementace, předkládá se uživatelům ke komentování a vyvíjí se přes několik verzí, dokud nevznikne odpovídající systém. [56] Vývoj byl rozdělen na jednotlivé logické celky (například návrh datového modelu, tvorba jednotlivých obrazovek, implementace algoritmu pro analýzu snímků a další). Každý celek byl samostatně navržen, implementován a následně validován.

8.2 Vytvoření projektu

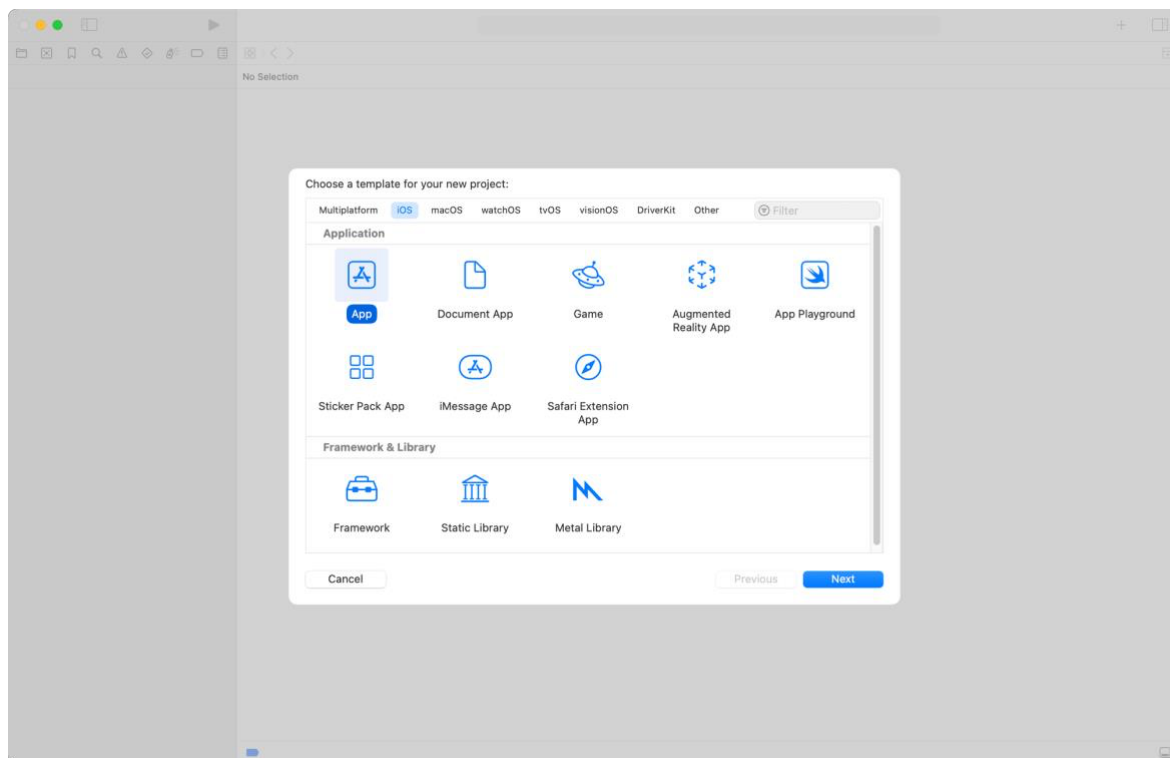
V této podkapitole je popsán postup vytvoření nového projektu v prostředí Xcode pro účely implementace prototypové aplikace. Proces je popsán tak, aby zahrnoval všechny důležité volby, které byly při vytváření projektu zvoleny.

Vytvoření projektu je realizováno vývojovým prostředím Xcode. Po jeho spuštění je zobrazena úvodní obrazovka, na které je pro vytvoření nového projektu nutné zvolit možnost „Create New Project“, jak je zachyceno na obrázku číslo 20. Tato volba slouží k inicializaci zcela nového projektu prostředím Xcode.



Obrázek 20 Úvodní obrazovka aplikace Xcode

Po kliknutí na tlačítko pro vytvoření nového projektu je uživateli zobrazeno dialogové okno s výzvou „Choose a template for your new project“. V rámci tohoto okna je nezbytné vybrat v sekci „Application“ šablonu s názvem „App“, která je určena pro tvorbu běžných aplikací pro platformu iOS. Tato volba je znázorněna na obrázku číslo 21. Šablona „App“ poskytuje výchozí konfiguraci projektu s podporou technologií SwiftUI a SwiftData.

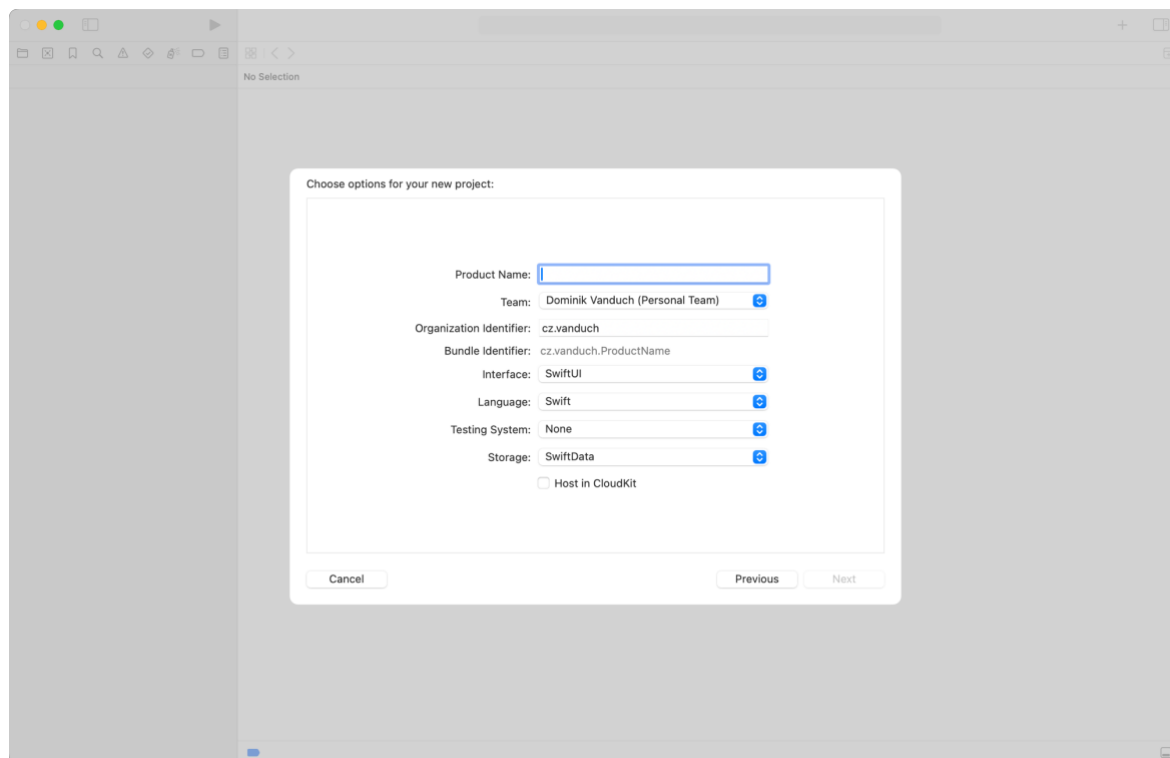


Obrázek 21 Volba šablony v prostředí Xcode

Po potvrzení volby šablony se zobrazí dialogové okno s nadpisem „Choose options for your new project“, kde je nutné specifikovat základní parametry projektu. Pro účely vyvíjené aplikace byly vyplněny následující specifikace projektu:

- *Product Name* – název aplikace pro identifikaci projektu,
- *Team* – vývojářský účet pro podepisování aplikace,
- *Organization Identifier* – identifikátor ve formátu reverzní domény pro jedinečnou identifikaci,
- *Interface* – zvolen framework *SwiftUI*,
- *Language* – zvolen jazyk *Swift*,
- *Testing System* – *None*, neboť pro účely prototypové aplikace nebylo plánováno zapojení testovacího frameworku,
- *Storage* – zvolen framework *SwiftData* pro nativní perzistenci dat,
- *Host in CloudKit* – nezaškrtnuto, protože cloudové ukládání není požadováno.

Obrazovka pro nastavení těchto možností je znázorněna na obrázku číslo 22.



Obrázek 22 Konfigurace možností šablony projektu v Xcode

Po potvrzení těchto nastavení je uživatel vyzván k uložení projektu do souborového systému počítače. V této fázi je důležité zvolit vhodné umístění pro uložení projektu. Pro tento projekt byla zvolena možnost „Create Git repository on my Mac“. Tímto krokem je inicializován lokální Git repozitář, který umožnil efektivní verzování kódu během vývoje. Tento krok není zobrazen na žádném z obrázků, ale jeho provedení je důležité zmínit, neboť verzovací systémy jako je Git tvoří nedílnou součást moderního vývojového procesu.

Po uložení projektu a potvrzení všech voleb byla v prostředí Xcode automaticky vygenerována základní struktura projektu dle zvolené šablony. Projekt po tomto kroku obsahuje předgenerované soubory, které umožňují okamžitě zahájit vývoj aplikace.

8.3 Konfigurace aplikace

Před implementací funkcionality aplikace bylo nutné provést základní konfiguraci projektu. Každý iOS projekt je definován prostřednictvím souboru s příponou *xcodeproj*, který obsahuje metadata o projektu, včetně nastavení build konfigurace, definice cílových zařízení a dalších projektových parametrů. Součástí této konfigurace je také soubor *Info.plist*, který slouží jako informační seznam vlastností aplikace a definuje klíčová metadata, jako je identifikátor balíčku, verze aplikace nebo požadovaná oprávnění systému.

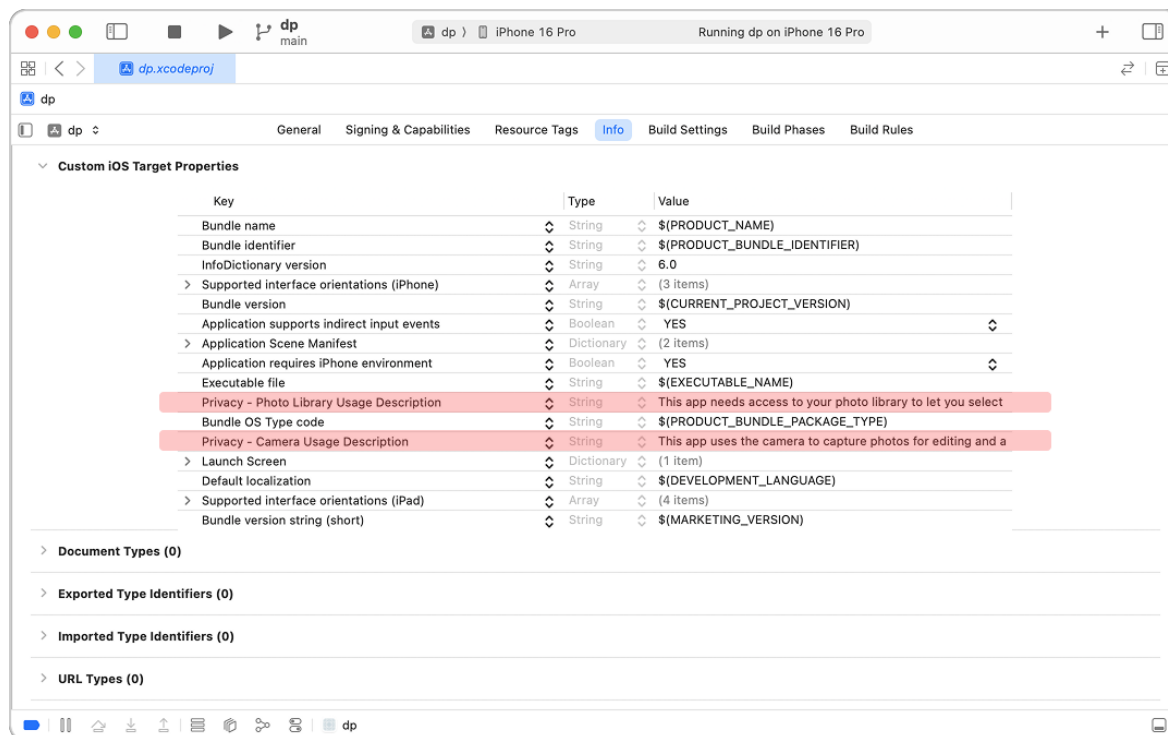
Vzhledem k povaze aplikace, která vyžaduje přístup k fotografiím uloženým v zařízení a možnost pořizování nových snímků prostřednictvím kamery, bylo nezbytné do tohoto konfiguračního souboru přidat specifická oprávnění. Konkrétně byly přidány klíče:

- *Privacy – Photo Library Usage Description,*
- *Privacy – Camera Usage Description.*

Tyto klíče slouží k definování textových popisů, které jsou uživateli zobrazeny v systémových dialogových oknech při prvním pokusu o přístup k příslušným funkcím zařízení. Jelikož systém iOS vyžaduje explicitní souhlas uživatele před udělením přístupu k citlivým datům nebo funkcionalitám, jsou tyto popisy nezbytné pro informování uživatele o účelu požadovaného přístupu.

Aplikace také využívá možnost výběru souborů ze systému, avšak tento přístup je realizován prostřednictvím *UIDocumentPickerViewController*, který poskytuje zabezpečený přístup k souborům mimo kontejner aplikace. Jelikož je tento výběr prováděn prostřednictvím systémového rozhraní, které automaticky zajišťuje bezpečnostní omezení a uživatelské oprávnění, není nutné do konfigurace aplikace přidávat dodatečná oprávnění pro přístup k souborovému systému. [47]

Výsledná konfigurace těchto oprávnění je zachycena na obrázku číslo 23, kde jsou v prostředí Xcode viditelná přidaná nastavení pro přístup ke galerii a kameře zařízení.



Obrázek 23 Přidání oprávnění pro přístup ke galerii a fotoaparátu zařízení

8.4 Architektura aplikace a struktura projektu

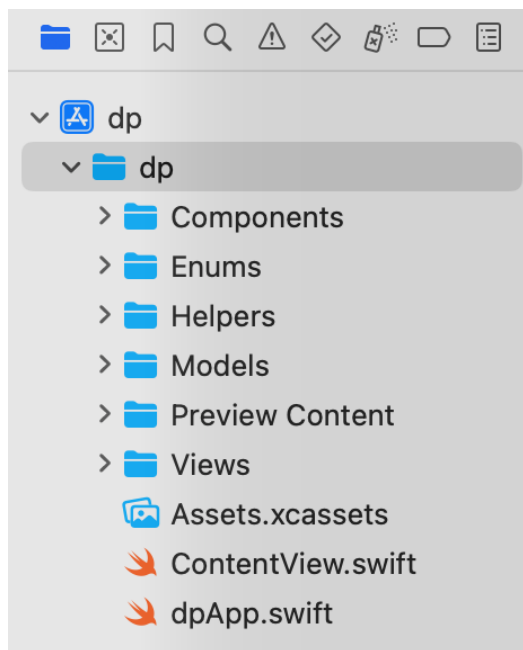
Pro vývoj aplikace byla zvolena architektura **Model-View** (dále jen MV), která představuje minimalistický přístup k návrhu iOS aplikací v prostředí SwiftUI. Tato architektura je charakterizována přímou vazbou mezi datovým modelem a prezentační vrstvou, přičemž je plně využíváno reaktivních vlastností frameworku SwiftUI. V rámci návrhové vzoru MV jsou definovány dvě základní komponenty – *Model* a *View*.

Model reprezentuje datovou vrstvu aplikace a obsahuje veškerou aplikační logiku spojenou se zpracováním dat, jejich ukládáním a manipulací. *View* představuje prezentační vrstvu, která je zodpovědná za zobrazení dat uživateli a zpracování uživatelských interakcí. Propojení těchto dvou vrstev je realizováno prostřednictvím *property wrappers* frameworku SwiftUI, jako jsou například *@StateObject*, *@ObservedObject* a *@Published*, které zajišťují automatickou aktualizaci uživatelského rozhraní při změnách v datovém modelu. [48]

Mezi hlavní výhody zvoleného přístupu patří jednoduchost implementace, minimální množství potřebného kódu pro dosažení požadované funkcionality a přirozená kompatibilita s deklarativní charakteristikou SwiftUI. Architektura MV je optimální pro aplikace menšího až středního rozsahu, kde není vyžadována komplexní separace business logiky do samostatné vrstvy. Na druhou stranu je třeba zmínit omezení v podobě snížené testovatelnosti jednotlivých komponent a potenciální problémy se škálovatelností při růstu

aplikace. [48] Při zohlednění těchto charakteristik se architektura MV jeví jako vhodná volba pro účely vyvíjené prototypové aplikace, kde rychlost implementace a jednoduchost struktury převažují nad požadavky na dlouhodobou škálovatelnost a rozsáhlou testovatelnost.

Konkrétní implementace architektury je reflektována ve struktuře projektu, která je znázorněna na obrázku číslo 24.



Obrázek 24 Struktura projektu prototypové aplikace

Hlavní struktura projektu je tvořena následujícími prvky:

- *Models* – složka obsahující datové modely aplikace využívající framework SwiftData,
- *Views* – složka obsahující prezentační komponenty využívající framework SwiftUI,
- *Components* – složka obsahující znovupoužitelné UI komponenty sdílené napříč aplikací využívající framework SwiftUI,
- *Helpers* – složka obsahující pomocné třídy a funkce,
- *Enums* – složka obsahující výčtové typy používané v aplikaci,
- *Preview Content* – složka obsahující testovací data a zdrojové soubory pro náhledy v Xcode,
- *Assets.xcassets* – soubor s grafickými zdroji aplikace,

- *dpApp.swift* – hlavní soubor obsahující vstupní bod aplikace,
- *ContentView.swift* – soubor definující základní strukturu aplikace.

Tato struktura projektu umožňuje efektivní organizaci kódu při zachování principů MV architektury, kde je jasně oddělena datová vrstva od prezentační vrstvy, přičemž je zachována jednoduchost a přehlednost celého řešení.

8.5 Definice datového modelu a perzistence dat

Datová vrstva aplikace byla implementována s využitím frameworku SwiftData, který zajišťuje perzistenci dat v místním úložišti zařízení. Inicializace datového modelu a konfigurace úložiště jsou realizovány při spuštění aplikace prostřednictvím hlavní struktury aplikace, která definuje globální kontejner pro správu datových objektů.

8.5.1 Inicializace datového kontejneru

Konfigurace datového úložiště je provedena v rámci hlavní struktury aplikace a tato implementace je znázorněna na obrázku číslo 25.

```
1 @main
2 struct dpApp: App {
3     var sharedModelContainer: ModelContainer = {
4         let schema = Schema([
5             Record.self, Compound.self, ExtrusionLine.self,
6         ])
7         let modelConfiguration = ModelConfiguration(
8             schema: schema,
9             isStoredInMemoryOnly: false
10        )
11
12        do {
13            return try ModelContainer(
14                for: schema,
15                configurations: [modelConfiguration]
16            )
17        } catch {
18            fatalError("Could not create ModelContainer: \(error)")
19        }
20    }()
21
22    var body: some Scene {
23        WindowGroup {
24            ContentView()
25        }
26        .modelContainer(sharedModelContainer)
27    }
28 }
```

Obrázek 25 Implementace datového kontejneru

Datový kontejner je vytvořen jako sdílený objekt typu *ModelContainer*, který slouží jako centrální bod pro správu všech datových operací v aplikaci. *Schema* definuje všechny entity, které mají být spravovány frameworkem *SwiftData*, konkrétně třídy *Record*, *Compound* a *ExtrusionLine*.

Konfigurace modelu je nastavena tak, aby data byla perzistentně ukládána do místního úložiště zařízení, nikoli pouze do paměti. Toto nastavení zajišťuje, že všechny záznamy zůstanou zachovány i po ukončení aplikace. V případě chyby při vytváření kontejneru je implementováno kritické selhání aplikace s odpovídající chybovou hláškou, což umožňuje identifikaci problémů při ladění.

Vytvořený kontejner je následně připojen k hlavnímu oknu aplikace prostřednictvím modifikátoru *modelContainer*, který zajišťuje dostupnost datového kontextu ve všech potomcích hierarchie *views*.

8.5.2 Model záznamu analýzy

Hlavní datovou entitou aplikace je třída *Record*, která reprezentuje jednotlivý záznam analýzy porozity. Implementace této třídy je zobrazena na obrázku číslo 26, kde je patrná struktura datového modelu s využitím *SwiftData* frameworku pro perzistenci dat.

```
1 @Model
2 final class Record {
3     var id: UUID
4     var dateCreated: Date
5     var note: String?
6     var porosityValue: Double
7
8     @Relationship(deleteRule: .nullify, inverse: \Compound.records)
9     var compound: Compound?
10    @Relationship(deleteRule: .nullify, inverse: \ExtrusionLine.records)
11    var extrusionLine: ExtrusionLine?
12
13    var verticesRaw: Data?
14    var verticesCount: Int = 4
15
16    //...
17 }
```

Obrázek 26 Vlastnosti hlavní třídy *Record*

Třída obsahuje základní identifikační údaje včetně unikátního identifikátoru a časového razítka vytvoření. Samotný výsledek analýzy je uložen v proměnné *porosityValue* jako desetinné číslo reprezentující vypočtenou porozitu. Volitelné vlastnosti umožňují přiřazení textové poznámky, vazby na směs materiálu (*compound*) a identifikaci výrobní linky

(*extrusionLine*). Tyto vztahy jsou definovány s pravidlem *nullify*, což znamená, že při odstranění souvisejících entit se vazba pouze zruší bez ovlivnění samotného záznamu. K uložení geometrických dat analyzované oblasti slouží vlastnosti *verticesRaw* a *verticesCount*, které uchovávají informace o oblasti výběru na fotografii.

Zvláštní pozornost byla věnována implementaci ukládání obrazových dat, které představují kritickou část datového modelu vzhledem k velikosti fotografií. Původní i ořezaná fotografie jsou ukládány jako binární data s využitím atributu *externalStorage*, který zajišťuje, že framework SwiftData ukládá tyto objekty mimo hlavní databázový soubor. Tento přístup významně optimalizuje výkon aplikace a snižuje velikost hlavního datového úložiště, což je důležité zejména při práci s větším množstvím fotografií. Pro transparentní práci s obrazovými daty jsou implementovány odvozené vlastnosti *originalImage* a *croppedImage*, které automaticky zajišťují konverzi mezi binárními daty a objekty typu *UIImage*. Při přiřazení obrázku do těchto vlastností je automaticky provedena konverze do formátu PNG prostřednictvím metody *pngData*, zatímco při čtení je využita inicializace objektu *UIImage* z uložených binárních dat. Praktická implementace této funkcionality je zobrazena na obrázku číslo 27.

```
1 @Model
2 final class Record {
3     // ...
4
5     @Attribute(.externalStorage) var originalImageData: Data?
6     @Attribute(.externalStorage) var croppedImageData: Data?
7
8     var originalImage: UIImage? {
9         get { originalImageData.flatMap(UIImage.init(data:)) }
10        set { originalImageData = newValue?.pngData() }
11    }
12
13    var croppedImage: UIImage? {
14        get { croppedImageData.flatMap(UIImage.init(data:)) }
15        set { croppedImageData = newValue?.pngData() }
16    }
17
18    // ...
19 }
```

Obrázek 27 Implementace ukládání obrázků v aplikaci do perzistentního úložiště

Vzhledem k tomu, že framework SwiftData neposkytuje přímou podporu pro datový typ *CGPoint*, bylo implementováno řešení založené na serializaci do JSON formátu. Souřadnice vrcholů, definujících analyzovanou oblast na fotografii, jsou ukládány v binární

podobě jako JSON data v atributu *verticesRaw*. Pro zajištění transparentního přístupu k těmto datům je implementovaná odvozená vlastnost *vertices*, která automaticky zajišťuje serializaci a deserializaci souřadnic prostřednictvím pomocné datové struktury *PointDTO*. Tato struktura funguje jako přenosový objekt umožňující konverzi mezi nativními objekty *CGPoint* a jejich JSON reprezentací.

Implementace pomocné struktury *PointDTO* je zobrazena na obrázku číslo 28.

```
1 private struct PointDTO: Codable {  
2     var x: Double  
3     var y: Double  
4     init(_ p: CGPoint) {  
5         x = p.x  
6         y = p.y  
7     }  
8     var cg: CGPoint { .init(x: x, y: y) }  
9 }
```

Obrázek 28 Implementace pomocné struktury PointDTO

8.5.3 Model směsi materiálu

Třída *Compound* reprezentuje směs materiálu používanou při výrobě pneumatiky a její implementace je znázorněna na obrázku 29.

```
1 @Model  
2 final class Compound {  
3     var id: UUID  
4     var dateCreated: Date  
5     var name: String  
6     var note: String  
7  
8     var records: [Record] = []  
9     //...  
10 }
```

Obrázek 29 Implementace třídy pro ukládání směsi

Tato entita slouží například ke kategorizaci záznamů analýz podle typu použitého materiálu a umožňuje snadné filtrování a vyhledávání v historii měření. Struktura třídy je relativně jednoduchá a obsahuje základní identifikační údaje včetně unikátního identifikátoru, časového razítka vytvoření, názvu směsi a volitelné poznámky. Název směsi slouží jako primární identifikátor pro uživatele, zatímco poznámka umožňuje uložení dodatečných informací o vlastnostech nebo složení směsi. Důležitou součástí modelu je také vztah

k záznamům měření prostřednictvím pole *records*, které udržuje odkazy na všechny záznamy používající danou směs materiálu.

8.5.4 Model výrobní linky

Třída *ExtrusionLine*, uchovávající informaci o použité výrobní lince při výrobě pneumatiky, má identickou implementaci jako třída pro ukládání směsi. Tato konzistence v návrhu zajišťuje jednotné chování při práci s oběma typy referenčních dat. Tato entita umožňuje například kategorizaci záznamů podle místa výroby a poskytuje další dimenzi pro organizaci a filtrování dat.

8.5.5 Vztahy mezi entitami

Datový model aplikace implementuje obousměrné vztahy typu „jeden k mnoha“ mezi hlavními entitami, což vytváří hierarchickou strukturu umožňující efektivní organizaci a kategorizaci dat. Tento přístup umožňuje, aby jedna směs (*Compound*) nebo jedna výrobní linka (*ExtrusionLine*) byly asociovány s libovolným počtem záznamů analýz, zatímco každý záznam může být spojen maximálně s jednou směsí a jednou výrobní linkou.

Technická implementace využívá pokročilé funkcionality SwiftData frameworku prostřednictvím anotace *@Relationship*. Parametr *deleteRule* nastavený na *nullify* definuje chování při mazání souvisejících entit – při odstranění směsi nebo výrobní linky se odpovídající reference v záznamech analýz automaticky nastaví na *nil*, čímž se zachová integrita záznamu bez nutnosti jeho odstranění. Parametr *inverse* zajišťuje obousměrnost vztahu tím, že explicitně definuje opačnou stranu vazby, což umožňuje SwiftData automaticky synchronizovat změny v obou směrech.

Praktický význam této architektury spočívá v možnosti efektivního filtrování a kategorizace dat. Uživatel může snadno zobrazit všechny analýzy provedené s konkrétní směsí nebo na specifické výrobní lince, zatímco systém automaticky udržuje konzistenci dat bez nutnosti manuální synchronizace. Tato implementace také podporuje budoucí rozšíření aplikace o pokročilé analytické funkce založené na historických datech seskupených podle směsi nebo výrobních procesů.

8.6 Vytvořené uživatelské rozhraní

V této podkapitole je popsáno vytvořené uživatelské rozhraní aplikace a představeny jeho klíčové části.

8.6.1 Hlavní navigace

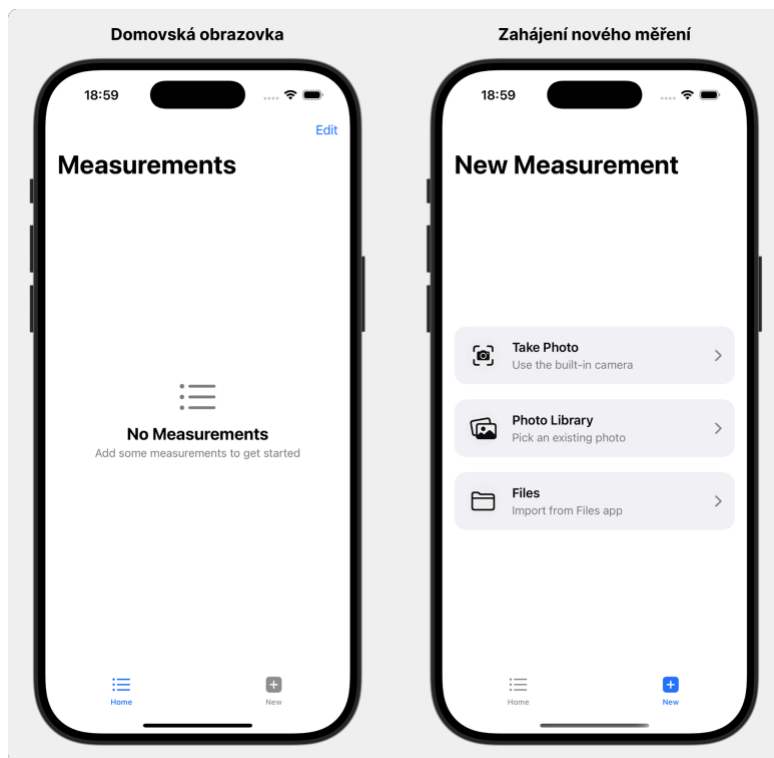
Hlavní navigační struktura aplikace byla implementována prostřednictvím komponenty *TabView* frameworku SwiftUI, která poskytuje standardní rozhraní pro přepínání mezi primárními sekcemi aplikace. Navigační systém je rozdělen do dvou hlavních záložek umístěných v dolní části obrazovky. Každá záložka reprezentuje odlišnou oblast aplikace.

Domovská obrazovka, zobrazená na levé straně obrázku číslo 30, představuje výchozí pohled aplikace po jejím spuštění. Poskytuje kompletní přehled všech provedených měření porozity v podobě přehledného seznamu. Pokud aplikace neobsahuje žádná uložená měření, zobrazí se uživateli prázdná obrazovka s informačním textem „No Measurements“ doplněným popisem „Add some measurements to get started“. Tento přístup zajišťuje, aby uživatel vždy pochopil aktuální stav aplikace a věděl, jaké kroky má podniknout pro zahájení práce s aplikací. Pravý horní roh obrazovky obsahuje tlačítko *Edit*, které aktivuje editační režim umožňující správu existujících záznamů. Tato funkcionality odpovídá standardnímu chování seznamů na iOS platformě, což zajišťuje intuitivní uživatelskou zkušenost v souladu s očekáváními uživatelů.

Druhá záložka je zobrazena na pravé straně obrázku číslo 30 a představuje vstupní bod pro vytvoření nového měření porozity. Tato sekce obsahuje tři možnosti pro získání vstupní fotografie:

- *Take Photo* – možnost pořízení nové fotografie prostřednictvím zabudované kamery,
- *Photo Library* – výběr existující fotografie z knihovny fotografií,
- *Files* – importování fotografie ze souborového systému.

Každá z těchto možností je reprezentována samostatnou kartou s příslušnou ikonou a popisným textem, přičemž na pravé straně každého řádku je umístěna šipka indikující možnost navigace do další úrovně rozhraní.



Obrázek 30 Hlavní navigace v aplikaci

Tato navigační struktura odpovídá konvencím platformy iOS a poskytuje intuitivní způsob organizace funkcionalit aplikace. Rozdělení na zobrazení existujících dat a vytváření nových záznamů zajišťuje logické oddělení různých uživatelských úkolů.

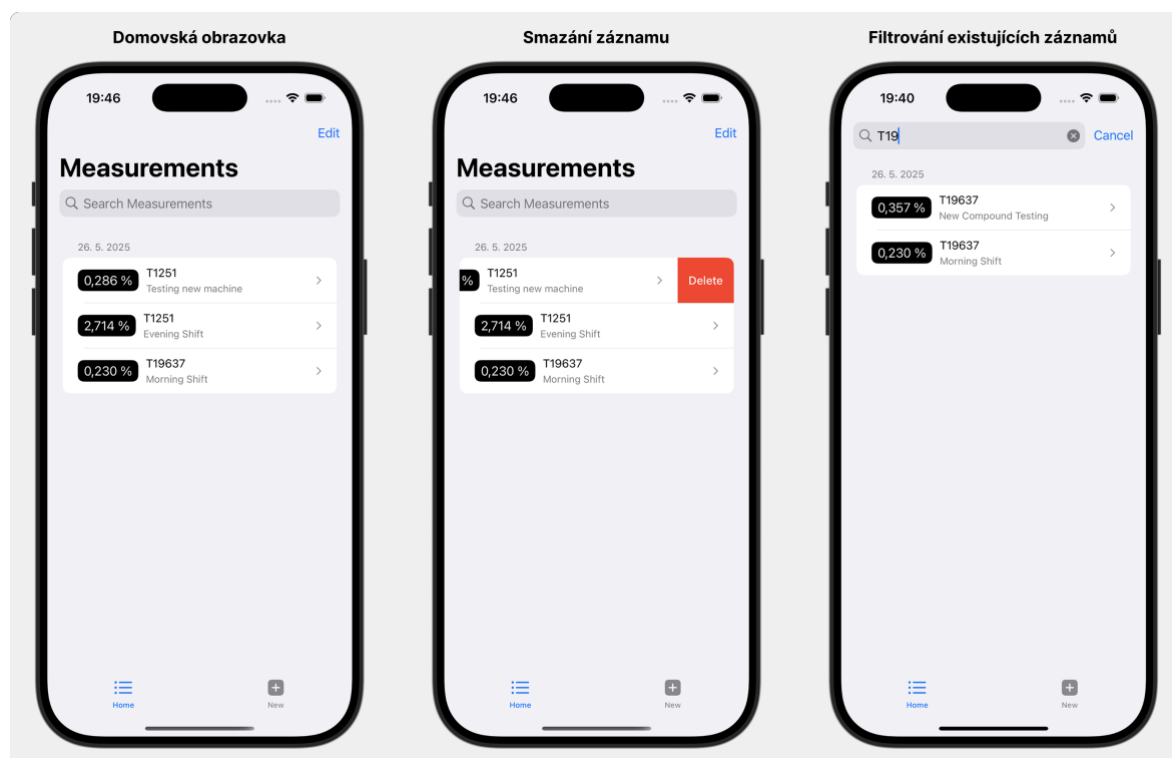
8.6.2 Domovská obrazovka

Domovská obrazovka aplikace představuje centrální bod pro správu a přehled všech provedených měření porozity. Implementace této komponenty byla realizována prostřednictvím *NavigationStack* a *List* komponenty frameworku SwiftUI. Data jsou získávána pomocí *@Query* makra ze SwiftData frameworku s automatickým řazením podle data vytvoření v sestupném pořadí.

V levé části obrázku číslo 31 lze vidět zobrazení záznamů měření na domovské obrazovce. Záznamy jsou zde organizovány do chronologické struktury seskupené podle jednotlivých dnů. Každý den je reprezentován samostatnou sekcí s hlavičkou obsahující datum. Jednotlivé záznamy v rámci každého dne obsahují vypočtenou hodnotu porozity zobrazenou v procentech, identifikační název použité směsi a doplňující poznámku pro dané měření. Šipka na jednotlivých záznamech naznačuje možnost přechodu na detailní zobrazení konkrétního měření.

Prostřední část obrázku číslo 31 ukazuje implementaci standardního iOS chování pro mazání záznamů prostřednictvím gesta. Po tažení prstu z pravé strany do levé dojde k zobrazení červeného tlačítka *Delete*, které umožňuje odstranění konkrétního záznamu z databáze. Jedná se o běžnou funkcionalitu napříč platformou iOS a její implementaci zprostředkovává ve SwiftUI modifikátor *onDelete*.

Pravá část obrázku číslo 31 zobrazuje implementaci vyhledávací a filtrovací funkcionality. V horní části obrazovky je umístěno vyhledávací pole. Vyhledávání probíhá v reálném čase napříč všemi atributy záznamu včetně data vytvoření, poznámek, názvu směsi a výrobní linky. Filtrovaná data jsou zobrazena ve stejné struktuře jako kompletní seznam a zachovávají chronologické seskupení podle dnů. V zobrazovaném příkladu je použit filtr „T19“ a na hlavní obrazovce jsou zobrazeny pouze záznamy odpovídající tomuto kritériu.

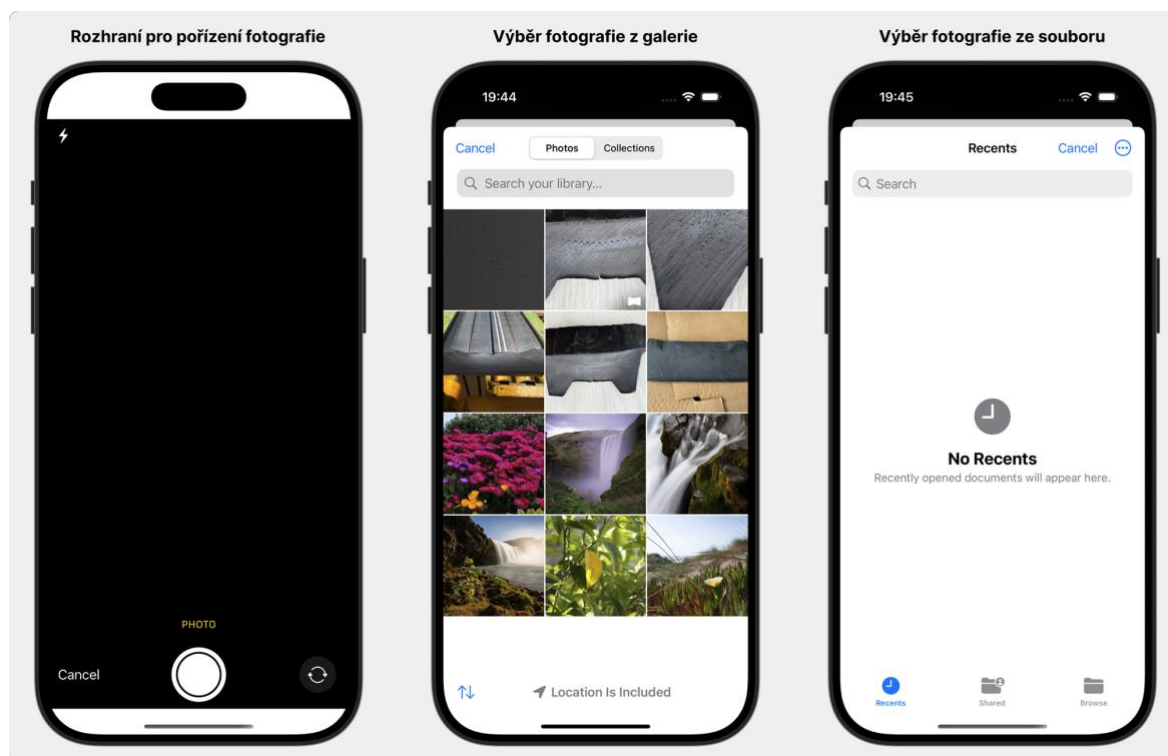


Obrázek 31 Domovská obrazovka aplikace a její funkce

8.6.3 Obrazovky pro nahrání vstupní fotografie

Funkcionalita získání vstupní fotografie pro analýzu porozity byla implementována prostřednictvím tří různých metod, které využívají standardní systémová rozhraní platformy iOS. Jak je znázorněno na příloženém obrázku číslo 32, uživatel má k dispozici pořízení

nové fotografie pomocí zabudované kamery, výběr existující fotografie z galerie a importování souboru ze souborového systému zařízení.



Obrázek 32 Nahrání fotografie do aplikace

Pro implementaci uvedených funkcionalit bylo nutné integrovat framework SwiftUI s komponentami UIKit. V levé části obrázku číslo 32 je znázorněno nativní rozhraní kamery iOS. Toto rozhraní je charakterizováno centrálním tlačítkem pro pořízení snímku a ovládacími prvky umístěnými ve spodní části obrazovky. Navíc bylo zachováno ovládání blesku pro optimalizaci snímků při slabém osvětlení. Rozhraní je automaticky konfigurováno pouze pro pořizování fotografií, čímž je vyloučena možnost nahrávání videí nebo jiných nekompatibilních formátů.

Prostřední část obrázku číslo 32 zobrazuje standardní rozhraní galerie fotografií iOS, které uživatelům umožňuje procházet a vybírat z uložených obrázků. Podobně jako tomu bylo u fotoaparátu, obsah galerie je filtrován tak, aby zobrazoval pouze fotografie, čímž eliminuje možnost výběru videí nebo jiných mediálních souborů.

Pravá část obrázku číslo 32 zobrazuje rozhraní aplikace Files pro importování fotografií ze souborového systému zařízení. Tato implementace využívá UIKit komponentu s explicitně definovaným typem obsahu *UTType.image*, což zajišťuje striktní filtrování a zobrazení pouze obrazových souborů podporovaných formátů.

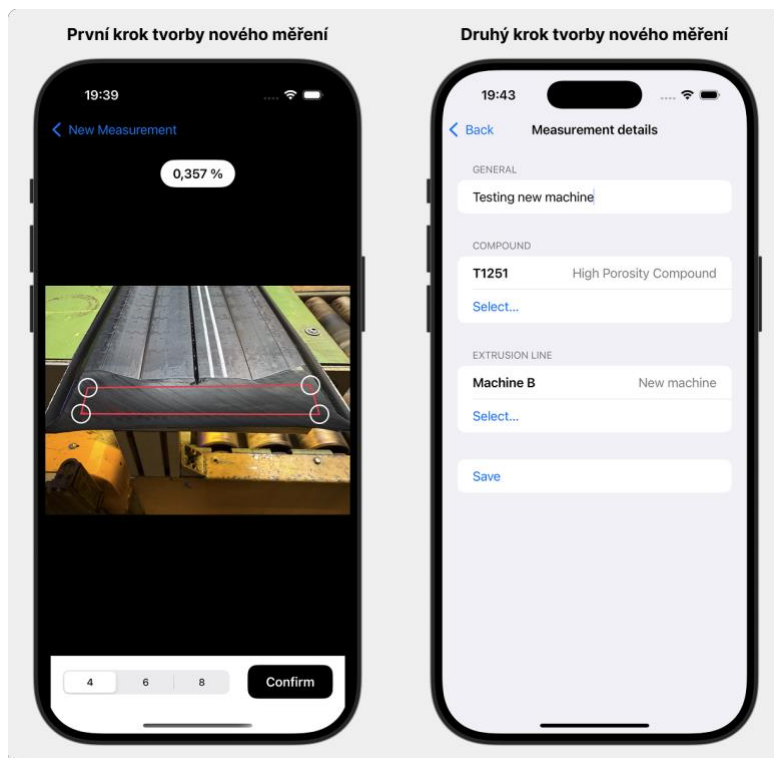
Po úspěšném výběru nebo pořízení fotografie je obraz uložen do vnitřního stavu aplikace.

8.6.4 Vytvoření záznamu o měření porozity

Aplikace byla navržena tak, aby byl uživatel po nahrání fotografie veden dvěma kroky: výběrem oblasti fotografie pro analýzu a zadáním souvisejících údajů s daným měřením. Účelem je nabídnout prostředí umožňující odhad porozity vzorku pneumatiky na základě zvolené části snímku.

První krok, jehož rozhraní je zobrazeno na levé straně obrázku číslo 33, umožňuje definici oblasti k analýze na nahrané fotografii prostřednictvím čtyř ohraničujících bodů. Aplikace podporuje také složitější geometrické tvary prostřednictvím přepnutí na model se šesti či osmi body podle potřeb konkrétní analýzy. Po každé úpravě pozice jednotlivých bodů se automaticky spustí výpočet porozity a výsledek se zobrazí v horní části obrazovky. Pixels, u kterých byla detekována porozita, jsou zvýrazněny červenou barvou, což poskytuje vizuální zpětnou vazbu o průběhu analýzy a umožňuje ověření správnosti výsledků. Potvrzením tlačítka *Confirm* se výřez uloží a uživatel je přesměrován do druhého kroku.

Druhý krok procesu, ilustrovaný na pravé straně obrázku číslo 33, zpřístupňuje formulář pro zadání doplňujících informací k provedenému měření. Formulář obsahuje textové pole pro popis měření následované dvěma sekcemi určenými pro výběr směsi materiálu a výrobní linky. Výběr v těchto sekcích je realizován prostřednictvím tlačítka *Select*, které otevírá seznam existujících položek a současně umožňuje vytvoření nových entit podle aktuálních potřeb. Závěrečné tlačítko *Save* zajišťuje uložení kompletního záznamu do databáze, čímž jsou všechny relevantní informace o měření systematicky zachyceny.



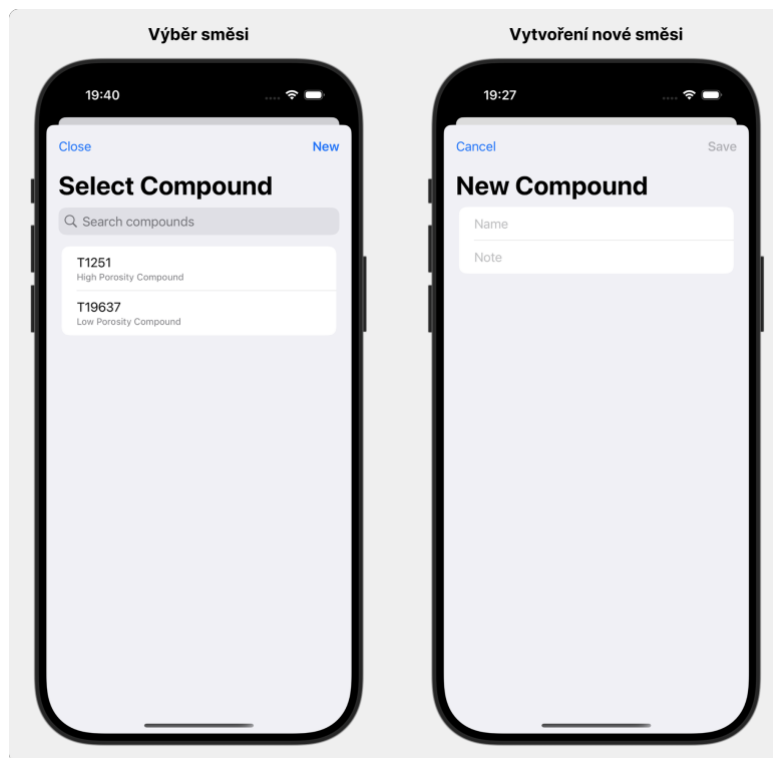
Obrázek 33 Vytvoření nového záznamu měření

8.6.5 Přidání doplňujících informací k záznamu

Pro rozšíření záznamu o použitou směs či výrobní linku je použito dvoustupňové rozhraní, které bylo navrženo tak, aby bylo možné buď vybrat existující položku ze seznamu, nebo vytvořit novou položku s požadovanými atributy. Uživatelské rozhraní se pro zajištění konzistentního uživatelské zážitku shoduje pro směs i výrobní linku.

První krok procesu, znázorněný na levé straně obrázku číslo 34, představuje obrazovku pro výběr existující položky ze seznamu dostupných možností. Horní část obrazovky obsahuje vyhledávací pole umožňující rychlé filtrování seznamu podle zadaného textu. Pod ním se nachází přehled dostupných položek, kde je u každé zobrazen název a doplňující popis pro snadnější identifikaci. Pokud požadovaná položka v seznamu neexistuje, může uživatel vytvořit novou pomocí stisknutí tlačítka *New* umístěného v horní části obrazovky.

Stisk tlačítka *New* otevírá formulář pro zadání údajů nové položky, jak ukazuje pravá strana obrázku číslo 34. Formulář se skládá ze dvou textových polí určených pro zadání názvu a poznámky k nové položce. Po kompletním vyplnění obou polí lze novou položku uložit prostřednictvím tlačítka *Save*. Systém následně automaticky vybere nově vytvořenou položku pro přiřazení k aktuálnímu záznamu, čímž se eliminuje nutnost dalších kroků ze strany uživatele.



Obrázek 34 Přidání doplňujících informací k záznamu

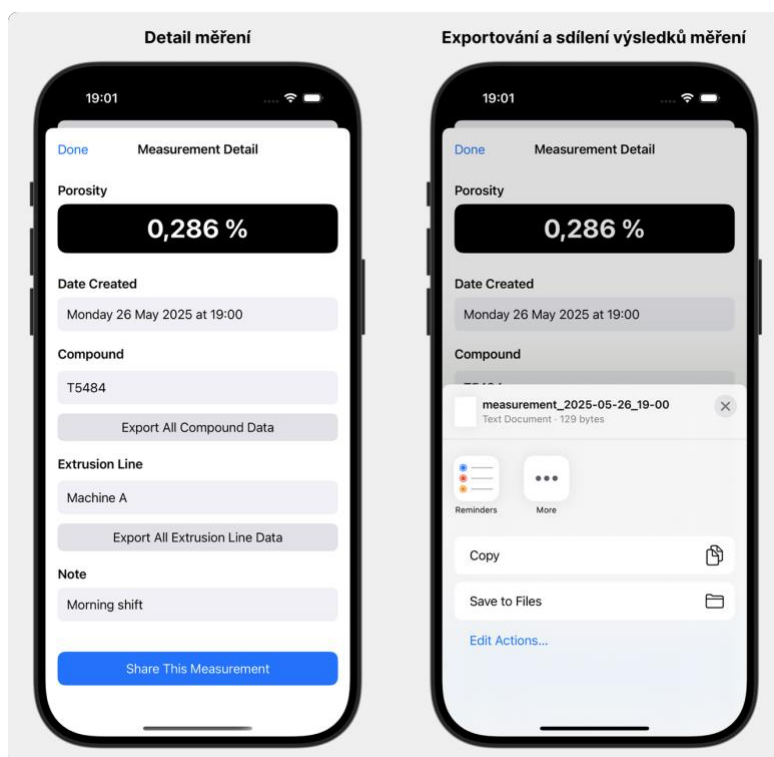
8.6.6 Detail záznamu a sdílení výsledků měření

Pro zobrazení detailu konkrétního měření se po výběru záznamu na domovské obrazovce otevře detailní pohled, který umožňuje prohlížení výsledků analýzy porozity a souvisejících metadat. Rozhraní tohoto detailního pohledu, zachycené na levé straně obrázku číslo 35, poskytuje uživateli přehledný přístup ke všem relevantním informacím o vybraném měření. V horní části obrazovky se nachází hodnota porozity. Pod touto hlavní informací jsou dále uvedeny doplňující informace k měření, včetně data a času pořízení záznamu, názvu použité směsi, výrobní linky a případné poznámky.

U směsi a výrobní linky je navíc k dispozici tlačítko umožňující export všech záznamů, které s daným prvkem souvisí. Po jeho stisknutí je vygenerován soubor ve formátu CSV, který obsahuje všechna měření spojená s danou směsí nebo výrobní linkou. Tento export umožňuje snadno získat širší pohled na vývoj porozity v čase v rámci konkrétního materiálu či výrobního zařízení. Tato funkce nachází využití zejména při hledání trendů, zpětném dohledávání problémových šarží nebo vyhodnocování kvality výroby.

Posledním prvkem na obrazovce je tlačítko sloužící ke sdílení právě zobrazeného záznamu. Po jeho stisknutí je uživateli nabídnuta možnost exportu jediného měření, a to opět ve formátu CSV. Exportování a následné sdílení probíhá prostřednictvím systémového

dialogu, který je zobrazen v pravé části obrázku číslo 35 a umožňuje odeslat soubor e-mailem, uložit jej do souborového systému zařízení nebo využít jiné dostupné aplikace (například poznámky, chatovací aplikace nebo cloudové úložiště). Možnost exportování dat představuje důležitý nástroj pro flexibilní práci s daty. Uživatel není nucen exportovat celý datový objem, ale může zvolit datový set dle aktuálních potřeb – ať už se jedná o analýzu konkrétního měření, celkové hodnocení kvality směsi, nebo srovnání výkonu jednotlivých výrobních linek. Tím je dosaženo vysoké míry přizpůsobení požadavkům reálného provozu.



Obrázek 35 Detail měření, export a sdílení výsledků

8.7 Klíčové části implementace

Tato podkapitola je zaměřena na popis klíčových komponent implementace, které tvoří hlavní funkcionalitu aplikace. Jsou představeny tři zásadní části, které bezprostředně souvisejí s uživatelskou interakcí, zpracováním obrazu a správou výstupních dat.

Nejprve je popsán mechanismus výběru požadované oblasti z pořízené nebo importované fotografie. Tato funkcionalita je realizována pomocí obrazovky pro manipulaci s ohraničujícími body, jenž uživateli umožňuje přesně vymežit oblast zájmu, ve které je prováděna analýza porozity. Výběr oblasti je esenciálním krokem, který podmiňuje správnost a spolehlivost výpočtu porozity.

V další podkapitole je popsán samotný proces analýzy obrazu, včetně algoritmického výpočtu odhadu porozity na základě vybrané oblasti. Text pokračuje vysvětlením způsobu zpracování pixelových dat, identifikace porézních oblastí a výpočtu relativního zastoupení porozity vůči celkovému objemu materiálu ve zvoleném výřezu. Je uvedeno, jakým způsobem je využita knihovna Core Image a jak byla optimalizována přesnost detekce.

Závěrečná podkapitola se věnuje implementaci exportování výsledků měření do formátu CSV a možnostem jejich sdílení. Uživatel má možnost exportovat jednotlivé záznamy i seskupené soubory dat podle zvolené směsi nebo výrobní linky. Je popsán proces serializace dat a jejich distribuce s ohledem na udržení integrity informací a snadnou integraci do stávajících výrobních a kontrolních procesů.

8.7.1 Výběr oblasti z fotografie k analýze porozity

Obrazovka pro výběr oblasti z fotografie k analýze porozity představuje základní komponentu aplikace. Tato funkcionality je implementována prostřednictvím struktury *NewRecordEditorView*, která kombinuje interaktivní uživatelské rozhraní s pokročilými algoritmy pro zpracování obrazu.

Hlavní stav komponenty je spravován pomocí dekorátorů *@State* a *@Bindable*, které zajišťují automatickou synchronizaci mezi datovým modelem a uživatelským rozhraním. Významným prvkem je pole kontrolních bodů typu *ControlPoint*, které definují vrcholy polygonu ohraničujícího analyzovanou oblast. Definice základních stavových proměnných pro práci s hraničními body je znázorněna na obrázku číslo 36.

```
1 @Bindable var record: Record
2
3 @State private var porosity: Double?
4 @State private var porosityMask: UIImage?
5 @State private var isCalculating = false
6 @State private var points: [ControlPoint] = []
7 @State private var selectedID: UUID?
8 @State private var selectedVertices = 4 {
9     didSet { regeneratePoints() }
10 }
11
12 private let allowedCounts = [4, 6, 8]
```

Obrázek 36 Definice stavových proměnných použitých k výběru oblasti fotografie k analýze porozity

Inicializace obrazovky probíhá na základě existujícího záznamu měření. Ohraničující body jsou buď načteny z uložených dat, nebo vygenerovány ve výchozí konfiguraci. Proces

generování výchozích bodů a jejich polohy je implementován prostřednictvím statické metody *defaultPoints*, která vytváří geometricky pravidelné rozložení vrcholů podle zvoleného počtu.

Rozhraní obrazovky je strukturováno jako vrstvená kompozice několika vizuálních komponent. Implementace této kompozice je naznačena na obrázku číslo 37. Základní vrstvu tvoří původní fotografie vzorku pneumatiky, která je zobrazena s využitím modifikátoru *aspectRatio* pro zachování správných proporcí. Nad touto vrstvou je umístěn takzvaný overlay (česky překrývající vrstva) obsahující interaktivní elementy pro manipulaci s kontrolními body. Vizuální zpětná vazba je doplněna o vrstvu *porosityMask*, která zvýrazňuje detekované póry v analyzované oblasti. Tato vrstva je vykreslena se sníženou průhledností, což umožňuje současné zobrazení původního obrazu i výsledků analýzy. Kombinace těchto vizuálních prvků poskytuje uživateli komplexní představu o kvalitě a přesnosti prováděné analýzy.

```
1 ZStack {
2   Image(uiImage: img)
3     .resizable()
4     .aspectRatio(contentMode: .fit)
5
6   if let mask = porosityMask {
7     Image(uiImage: mask)
8       .resizable()
9       .aspectRatio(contentMode: .fit)
10      .blendMode(.screen)
11      .opacity(0.6)
12   }
13
14   CropOverlay(
15     size: geo.size,
16     onDragEnd: recalcPorosity,
17     points: $points,
18     selectedID: $selectedID
19   )
20 }
```

Obrázek 37 Vrstvená struktura pro zobrazení interaktivních bodů a detekované porozity nad obrázkem

Interaktivní vrstva s ohraničujícími body je realizována prostřednictvím *CropOverlay* struktury, která kombinuje Canvas API pro vykreslování polygonu se správcí gest pro manipulaci s jednotlivými vrcholy. Implementaci této vrstvy je možné vidět na obrázku číslo 38. Canvas komponenta využívá Path API pro vytvoření uzavřené křivky spojující všechny ohraničující body. Polygon je vykreslen s kontrastní růžovou barvou pro lepší viditelnost na tmavém pozadí materiálu.

```

1 private struct CropOverlay: View {
2     let size: CGSize
3     var onDragEnd: () → Void
4
5     @Binding var points: [ControlPoint]
6     @Binding var selectedID: UUID?
7
8     var body: some View {
9         Canvas { ctx, canvasSize in
10             guard points.count ≥ 2 else { return }
11             var path = Path()
12             path.move(to: points[0].point * canvasSize)
13             points.dropFirst().forEach {
14                 path.addLine(to: $0.point * canvasSize)
15             }
16             path.closeSubpath()
17             ctx.stroke(path, with: .color(.pink), lineWidth: 2)
18         }
19         .overlay(handles)
20         .contentShape(Rectangle())
21     }
22
23     @ViewBuilder private var handles: some View {
24         ForEach($points) { $pt in
25             Circle()
26                 .stroke(Color.white, lineWidth: 2)
27                 .frame(width: 24, height: 24)
28                 .position(pt.point * size)
29                 .gesture(
30                     DragGesture()
31                         .onChanged { value in
32                             pt.point = clamp(value.location / size)
33                             selectedID = pt.id
34                         }
35                         .onEnded { _ in onDragEnd() }
36                 )
37         }
38     }
39     // ...
40 }

```

Obrázek 38 Interaktivní vrstva pro výběr požadované sekce k analýze porozity

Každý ohraničující bod je reprezentován jako interaktivní kruhový element, který reaguje na dotyková gesta uživatele. Implementaci ohraničujících bodů je možné vidět na obrázku číslo 39. Ovládání bodů gesty tažením umožňuje jejich snadné přemísťování v rámci hranic fotografie. Souřadnice každého bodu jsou automaticky normalizovány do rozmezí 0 až 1 pro zachování konzistence napříč různými velikostmi obrazu.

```

1 Circle()
2   .stroke(Color.white, lineWidth: 2)
3   .frame(width: 24, height: 24)
4   .gesture(
5     DragGesture()
6       .onChange { value in
7         pt.point = clamp(value.location / size)
8         selectedID = pt.id
9       }
10    .onEnded { _ in onDragEnd() }
11  )

```

Obrázek 39 Implementace logiky pro vykreslení a chování ohraničujících bodů

Proces výpočtu porozity je spuštěn automaticky při každé změně pozice bodů. Implementace využívá asynchronní zpracování prostřednictvím Task API, které zajišťuje, že náročné výpočty neblokují hlavní vlákno uživatelského rozhraní. Výsledek analýzy je zobrazen v reálném čase pomocí *PorosityPill* komponenty, která poskytuje okamžitou zpětnou vazbu o kvalitě vybrané oblasti. Přidáním stavové proměnné *isCalculating* lze v uživatelském rozhraní signalizovat uživateli stav, kdy probíhá výpočet porozity, čímž dochází ke zlepšení uživatelského zážitku. Realizaci asynchronního výpočtu porozity lze pozorovat na obrázku číslo 40.

```

1 private func recalcPorosity() {
2   guard let img = record.originalImage else { return }
3   let crop = img.cropped(using: points.map { $0.point * img.size })
4
5   isCalculating = true
6   Task.detached(priority: .userInitiated) {
7     let value = PorosityCalculator.porosity(in: target) ?? 0
8     await MainActor.run {
9       porosity = value
10      isCalculating = false
11    }
12  }
13 }

```

Obrázek 40 Implementace asynchronního výpočtu porozity

Funkčnost ořezávání fotografie je implementována rozšířením třídy *UIImage*, které přidává metodu *cropped*. Tato metoda využívá Core Graphics framework pro vytvoření nového obrazu obsahujícího pouze oblast definovanou ohraničujícími body. Proces zahrnuje vytvoření maskovací vrstvy z hraničních bodů a následné vykreslení původního obrazu do nového kontextu s aplikovanou maskou. Tato implementace je naznačena na obrázku číslo 41.

```

1 extension UIImage {
2     func cropped(
3         using polygon: [CGPoint],
4         backgroundColor: UIColor = .white
5     ) → UIImage? {
6         guard polygon.count ≥ 3 else { return nil }
7
8         let fmt = UIGraphicsImageRendererFormat.default()
9         fmt.opaque = true
10
11         return UIGraphicsImageRenderer(size: size, format: fmt).image { ctx in
12             backgroundColor.setFill()
13             ctx.fill(CGRect(origin: .zero, size: size))
14             let path = createPath(from: polygon)
15             ctx.cgContext.addPath(path.cgPath)
16             ctx.cgContext.clip()
17             draw(at: .zero)
18         }
19     }
20
21     private func createPath(from points: [CGPoint]) → UIBezierPath {
22         let path = UIBezierPath()
23         guard !points.isEmpty else { return path }
24
25         path.move(to: points[0])
26         points.dropFirst().forEach { path.addLine(to: $0) }
27         path.close()
28
29         return path
30     }
31 }

```

Obrázek 41 Rozšíření třídy UIImage

Ukládání stavu interakcí mezi uživatelem a aplikací je provedeno prostřednictvím metody *saveCrop*, která je zobrazena na obrázku číslo 42. Tato metoda aktualizuje datový model záznamu s definovanými parametry. Kromě souřadnic kontrolních bodů jsou uloženy také ořezaný obraz a vypočtená hodnota porozity, což zajišťuje konzistenci dat pro další zpracování a exportování.

```

1 private func saveCrop() {
2     guard let img = record.originalImage,
3           let cg = img.cgImage
4     else { return }
5
6     let size = CGSize(width: cg.width, height: cg.height)
7     let pixelPoints = points.map { $0.point * size }
8
9     record.vertices = points.map(\.point)
10    record.verticesCount = selectedVertices
11    record.croppedImage = img.cropped(using: pixelPoints)
12    record.porosityValue = porosity ?? record.porosityValue
13 }

```

Obrázek 42 Ukládání stavu interakcí mezi uživatelem a aplikací

Celková implementace obrazovky pro výběr relevantní části fotografie k analýze představuje spojení interaktivního a responzivního designu s pokročilými technikami zpracování obrazu. Reaktivní architektura SwiftUI frameworku umožňuje vytvoření responsivního uživatelského rozhraní, které poskytuje okamžitou zpětnou vazbu a intuitivní ovládání pro přesnou definici analyzovaných oblastí.

8.7.2 Analýza vybrané oblasti a výpočet porozity

Algoritmus pro výpočet porozity vzorků pneumatik představuje další klíčovou funkcionalitu vytvořené aplikace. Proces analýzy je rozdělen do několika na sebe navazujících kroků, které umožňují přesné určení poměru pórů materiálu vůči celkovému objemu analyzovaného materiálu.

Základním principem algoritmu je převod barevného snímku na binární masku, která jednoznačně rozlišuje mezi materiálem pneumatiky a póry. Binární reprezentace je nutná z důvodu potřeby jednoznačné klasifikace každého pixelu buď jako materiál nebo jako pór, což umožňuje následné kvantitativní vyhodnocení. Tento převod je realizován prostřednictvím prahování, které klasifikuje jednotlivé pixely na základě jejich intenzity jasu. Prahování je vhodnou metodou pro tuto aplikaci, protože póry v materiálu pneumatiky se projevují jako tmavší oblasti v důsledku stínu způsobeného absencí materiálu, což vytváří přirozený kontrast vůči světlejšímu základnímu materiálu. Tmavé oblasti odpovídající pórům jsou identifikovány jako pixely s intenzitou pod definovanou prahovou hodnotou, zatímco světlejší oblasti představují samotný materiál pneumatiky. Hlavní funkce pro výpočet porozity je zobrazena v ukázce kódu na obrázku číslo 43.

```
1 static func porosity(  
2     in image: UIImage,  
3     threshold: CGFloat = 0.03,  
4     downsampleTo targetWidth: CGFloat = 2048  
5 ) -> Double? {  
6     guard  
7         let (binary, totalPixels) = prepareBinaryMask(  
8             image: image,  
9             threshold: threshold,  
10            downsampleTo: targetWidth  
11        )  
12     else { return nil }  
13  
14     let bubbleCount = binary.filter { $0 == 0 }.count  
15     return Double(bubbleCount) / Double(totalPixels)  
16 }
```

Obrázek 43 Hlavní funkce pro výpočet porozity

Před samotnou analýzou je vstupní obraz upraven za účelem optimalizace výpočetní náročnosti. Konkrétně se jedná o změnu velikosti, protože moderní mobilní zařízení pořizují fotografie s vysokým rozlišením, které by při přímém zpracování způsobovalo nadměrnou spotřebu paměti a prodloužení doby výpočtu bez významného přínosu k přesnosti výsledků.

Dalším důležitým krokem je konverze na odstíny šedi, která je realizována z důvodu eliminace barevných informací. Ty nejsou pro identifikaci pórů relevantní a mohou vnášet do analýzy nežádoucí šum. Barevné informace mohou být ovlivněny osvětlením, nastavením fotoaparátu nebo vlastnostmi materiálu, což by mohlo vést k nekonzistentním výsledkům. Převod vstupního snímku na černobílý (anglicky *grayscale*) umožňuje algoritmu pracovat výhradně s rozdíly v intenzitě jasu, které jsou klíčové pro rozlišení mezi materiálem a póry. Konverze je realizována prostřednictvím frameworku Core Image, konkrétně filtru *CIColorControls* s parametrem saturace nastaveným na nulovou hodnotu. Proces konverze je zobrazen v ukázce kódu na obrázku číslo 44.

```
1 let ciGray = CIImage(cgImage: cgScaled)
2   .applyingFilter(
3     "CIColorControls",
4     parameters: [kCIInputSaturationKey: 0]
5   )
```

Obrázek 44 Aplikování černobílého filtru na vstupní snímek

Prahování představuje kritický krok v procesu analýzy, jehož cílem je transformace spojitých hodnot jasu na diskrétní binární reprezentaci. Implementace této funkcionality využívá opět framework Core Image, konkrétně filtr *CIColorThreshold*, který transformuje snímek v odstínech šedi na binární reprezentaci. Pro potřeby prototypové aplikace bylo provedeno experimentální ověření různých prahových hodnot za účelem optimalizace detekce porozity. Na základě testování v konkrétních světelných podmínkách, ve kterých byly realizovány validační testy, byla jako optimální určena prahová hodnota 0,03. Tato hodnota umožňuje spolehlivé rozlišení mezi materiálem pneumatiky a póry na základě charakteristických rozdílů v intenzitě jasu těchto oblastí. Pixely s intenzitou pod prahovou hodnotou jsou klasifikovány jako póry a označeny hodnotou 0, zatímco pixely představující materiál nabývají hodnoty 255. Volba této konkrétní prahové hodnoty reflektuje podmínky, jako jsou optimální osvětlení nebo kontrastní poměry v řezech pneumatik.

Výpočet porozity je založen na statistické analýze binární masky, která umožňuje kvantifikaci poměru porézních oblastí k celkovému objemu materiálu. Algoritmus počítá

počet pixelů odpovídajících pórům a tento počet vztahuje k celkovému počtu pixelů v analyzované oblasti. Výsledná hodnota porozity je vyjádřena jako desetinné číslo v rozsahu od 0 do 1, kde hodnota 0 reprezentuje materiál bez pórů a hodnota 1 by teoreticky odpovídala oblasti složené výhradně z pórů – což lze interpretovat jako chybu v měření. Tato reprezentace umožňuje snadné porovnávání mezi různými vzorky a poskytuje intuitivní interpretaci výsledků pro uživatele aplikace. Implementace přípravy binární masky je uvedena v ukázce kódu na obrázku číslo 45.

```

1 private static func prepareBinaryMask(
2     image: UIImage,
3     threshold: CGFloat,
4     downsampleTo targetWidth: CGFloat
5 ) -> ([UInt8], Int)? {
6     guard
7         let cgInput = image.cgImage,
8         let cgScaled = cgInput.resized(toWidth: targetWidth),
9         let threshFilter = CIFilter(name: "CIColorThreshold")
10    else { return nil }
11
12    let ciGray = CIImage(cgImage: cgScaled)
13        .applyingFilter(
14            "CIColorControls",
15            parameters: [kCIInputSaturationKey: 0]
16        )
17    threshFilter.setValue(ciGray, forKey: kCIInputImageKey)
18    threshFilter.setValue(threshold, forKey: "inputThreshold")
19    guard
20        let ciOut = threshFilter.outputImage,
21        let cgOut = ciContext.createCGImage(ciOut, from: ciOut.extent),
22        let data = cgOut.dataProvider?.data as Data?
23    else { return nil }
24
25    return ([UInt8](data), cgOut.width * cgOut.height)
26 }

```

Obrázek 45 Příprava binární masky pro výpočet porozity

Pro vizualizace výsledků analýzy je implementována funkce pro generování barevné masky, která překrývá původní snímek s identifikovanými póry. Tato vizualizace je důležitá pro validaci výsledků algoritmu a umožňuje uživateli ověřit, zda byly póry správně identifikovány. Póry jsou zvýrazněny červenou barvou pomocí filtru *CIFalseColor*, což uživateli umožňuje vizuálně ověřit správnost detekce. Tato funkcionalita slouží nejen pro kontrolu výsledků, ale také pro případné ladění parametrů algoritmu v závislosti na specifických charakteristikách analyzovaných vzorků.

Implementace využívá výhradně nativní knihovny iOS platformy, konkrétně Core Image framework pro zpracování obrazu a Core Graphics pro manipulaci s bitmapovými daty. Volba nativních technologií je motivována požadavky na optimální výkon,

spolehlivost a plnou integraci s operačním systémem iOS. Nativní přístup také zajišťuje efektivní využití hardwarových akcelérátorů dostupných v zařízeních Apple, což je kritické pro zpracování obrazu v reálném čase. Algoritmus je navržen tak, aby minimalizoval spotřebu paměti prostřednictvím efektivního zpracování dat po jednotlivých krocích bez nutnosti uchovávání mezivýsledků, což je důležité pro stabilní provoz na mobilních zařízeních s omezenými zdroji.

Přesnost výpočtu porozity je ovlivněna několika faktory, mezi které patří kvalita vstupního snímku, osvětlení při pořizování fotografie a vhodnost zvolené prahové hodnoty. Algoritmus zároveň obsahuje mechanismy pro zacházení s chybovými stavy, včetně neplatných vstupních dat nebo selhání při zpracování obrazu.

8.7.3 Export záznamů do formátu CSV a jejich sdílení

Export naměřených dat do standardizovaného formátu představuje poslední představenou klíčovou funkcionalitu, sloužící k integraci aplikace do existujících průmyslových procesů a systémů řízení kvality. Implementace exportování dat do formátu CSV je motivována potřebou zajistit interoperabilitu s externími systémy pro analýzu dat, reporting a dlouhodobé archivování výsledků měření porozity. Formát CSV byl zvolen z důvodu své univerzálnosti, jednoduchosti zpracování a široké podpory napříč různými softwarovými platformami.

Architektura funkcionality je rozdělena do dvou hlavních komponent, které zajišťují oddělení logiky generování CSV obsahu od mechanismu sdílení dat. Toto rozdělení umožňuje flexibilní použití funkcí pro export dat v různých kontextech aplikace a usnadňuje budoucí rozšíření funkcionality. První komponenta je odpovědná za transformaci datových struktur aplikace do formátu CSV, zatímco druhá komponenta zajišťuje prezentaci exportovaných dat uživateli prostřednictvím nativních sdílecích mechanismů platformy iOS.

Generování CSV obsahu je implementováno prostřednictvím struktury *CSVHelper*, která poskytuje metody pro export jednotlivých záznamů i kolekcí záznamů. Metoda *exportRecord* zastřešuje export jednotlivého záznamu a je zobrazena v ukázce kódu na obrázku číslo 46.

```
1 static func exportRecord(_ record: Record) -> String {
2     let headers = "Date Created, Porosity Value, Compound, Extrusion Line, Note"
3     let dateFormatter = DateFormatter()
4     dateFormatter.dateFormat = ".medium"
5     dateFormatter.timeStyle = .short
6
7     let row = [
8         escapeCsvValue(dateFormatter.string(from: record.dateCreated)),
9         String(record.porosityValue),
10        escapeCsvValue(record.compound?.name ?? ""),
11        escapeCsvValue(record.extrusionLine?.name ?? ""),
12        escapeCsvValue(record.note ?? ""),
13    ].joined(separator: ",")
14
15    return "\(headers)\n\(row)"
16 }
```

Obrázek 46 Export jednotlivého záznamu do formátu CSV

Strukturování exportovaných dat reflektuje všechny stěžejní informace uložené v datovém modelu aplikace. Hlavička CSV souboru obsahuje pět sloupců odpovídajících datu vytvoření záznamu, hodnotě porozity, názvu směsi, označení výrobní linky a textové poznámce.

Export více záznamů současně je implementován prostřednictvím metody *exportRecords*, která zpracovává kolekce záznamů a generuje jednotný CSV soubor se všemi relevantními daty. Tento přístup je efektivnější pro hromadné operace a umožňuje uživatelům exportovat kompletní datové sady. Implementace hromadného exportu je zobrazena v ukázce kódu na obrázku číslo 47.

```
1 static func exportRecords(_ records: [Record]) -> String {
2     let headers = "Date Created, Porosity Value, Compound, Extrusion Line, Note"
3     let dateFormatter = DateFormatter()
4     dateFormatter.dateFormat = ".medium"
5     dateFormatter.timeStyle = .short
6
7     let rows = records.map { record in
8         [
9             escapeCsvValue(dateFormatter.string(from: record.dateCreated)),
10            String(record.porosityValue),
11            escapeCsvValue(record.compound?.name ?? ""),
12            escapeCsvValue(record.extrusionLine?.name ?? ""),
13            escapeCsvValue(record.note ?? ""),
14        ].joined(separator: ",")
15    }
16
17    return ([headers] + rows).joined(separator: "\n")
18 }
```

Obrázek 47 Export kolekce záznamů do formátu CSV

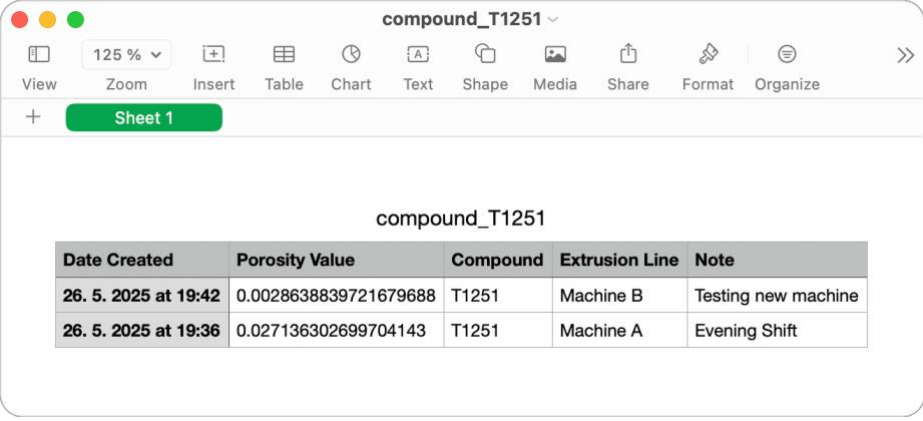
Sdílení exportovaných dat je realizováno pomocí struktury *ShareManager*, která využívá nativní *UIActivityViewController* pro prezentaci možností sdílení uživateli. Tento přístup zajišťuje konzistentní uživatelský zážitek s ostatními iOS aplikacemi a poskytuje široké spektrum možností pro distribuci dat včetně e-mailu, cloudových služeb a přímého přenosu do jiných aplikací. Využití nativního mechanismu také automaticky respektuje uživatelské preference a nastavení systému.

Proces sdílení je realizován prostřednictvím několika na sebe navazujících kroků, které zajišťují vytvoření a následné sdílení CSV souboru. Nejprve je vytvořena cesta k dočasnému souboru v cache adresáři aplikace, který je vhodný pro dočasné ukládání souborů. Následuje zápis CSV obsahu do souboru. Po úspěšném uložení dat se vytvoří instance *UIActivityViewController*, která představuje standardní iOS rozhraní pro sdílení souborů prostřednictvím různých aplikací a služeb. Tato komponenta je následně prezentována uživateli, což umožňuje výběr způsobu sdílení podle individuálních potřeb. Celá implementace zahrnuje také ošetření chybových stavů prostřednictvím *try-catch* bloku, který zachytí případné problémy při zápisu souboru a informuje uživatele o chybě. Implementace funkcionality sdílení je zobrazena na obrázku číslo 48.

```
1 struct ShareManager {
2     static func shareCSV(content: String, fileName: String = "export") {
3         //...
4
5         let cachesDir = FileManager.default.urls(
6             for: .cachesDirectory,
7             in: .userDomainMask
8         ).first!
9         let fileURL = cachesDir.appendingPathComponent("\(fileName).csv")
10
11         do {
12             try content.write(to: fileURL, atomically: true, encoding: .utf8)
13             let activityVC = UIActivityViewController(
14                 activityItems: [fileURL],
15                 applicationActivities: nil
16             )
17
18             //...
19
20             presentingViewController.present(activityVC, animated: true)
21         } catch {
22             print("Error writing CSV file: \(error)")
23         }
24     }
25 }
```

Obrázek 48 Implementace sdílení CSV souborů

Na ukázce v příloženém obrázku číslo 49 je zobrazené, jak se exportovaná data zobrazují po otevření v aplikaci Numbers. Tabulkový formát přehledně strukturuje všechny stěžejní informace o měření porozity do jasně definovaných sloupců. Každý řádek představuje samostatný záznam s kompletními metadaty včetně času vytvoření, naměřené hodnoty, identifikace materiálu a výrobní linky. Tato standardizovaná struktura dat umožňuje uživatelům okamžitě pokračovat v analýze výsledků například pomocí pokročilých funkcí tabulkového procesoru, jako jsou výpočty statistických ukazatelů, vytváření grafů pro vizualizaci trendů nebo filtrování dat podle specifických kritérií. Díky této interoperabilitě aplikace plní svůj hlavní cíl integrace do existujících průmyslových procesů, kde naměřená data mohou být bezproblémově začleněna do širších systémů řízení kvality a reportingu bez nutnosti manuální transformace nebo přepisu informací.



Date Created	Porosity Value	Compound	Extrusion Line	Note
26. 5. 2025 at 19:42	0.0028638839721679688	T1251	Machine B	Testing new machine
26. 5. 2025 at 19:36	0.027136302699704143	T1251	Machine A	Evening Shift

Obrázek 49 Náhled na exportovaná data směsi T1251 v aplikaci Numbers od Apple

9 VALIDAČNÍ TESTOVÁNÍ APLIKACE

Devátá kapitola se zabývá validačním testováním vytvořeného prototypu mobilní aplikace, jehož cílem je ověřit, zda aplikace splňuje definované cíle a funkcionální i nefunkcionální požadavky. V následujících podkapitolách jsou popsány testovací metodiky, které byly použity k validaci splnění stanovených požadavků na aplikaci. Poté jsou představeny výstupy tohoto testování. Závěr kapitoly shrnuje získané poznatky z testovacího procesu a jejich dopad na celkové hodnocení aplikace jako náhrady současných metod odhadu porozity pneumatik. Testovací proces byl navržen tak, aby poskytl objektivní hodnocení schopnosti aplikace plnit svou primární funkci v průmyslových podmínkách a aby identifikoval případné oblasti pro budoucí vylepšení.

9.1 Metodika testování

Validační testování vyvinuté aplikace bylo realizováno prostřednictvím systematického přístupu založeného na ověření splnění předem definovaných funkcionálních a nefunkcionálních požadavků. Testovací proces byl navržen tak, aby poskytl objektivní hodnocení schopnosti aplikace fungovat v reálných výrobních podmínkách a aby identifikoval případné nedostatky nebo oblasti vyžadující optimalizaci.

Pro účely testování bylo vybráno zařízení iPhone 16 Pro, které reprezentuje současnou generaci mobilních zařízení Apple a splňuje technické specifikace uvedené v nefunkcionálním požadavku NF02. Tento konkrétní model byl zvolen z důvodu jeho dostupnosti autorovi této práce, pokročilými fotografickými schopnostmi, výpočetním výkonem a dostupností nejnovějších funkcí platformy iOS, což zajistilo testování aplikace v optimálních podmínkách.

Testování funkcionálních požadavků bylo provedeno metodou, při které byla ověřována korektnost jednotlivých funkcí aplikace bez znalosti vnitřní implementace. Každý funkcionální požadavek byl testován samostatně prostřednictvím definovaných testovacích scénářů. Výsledky byly hodnoceny binárně jako splněný nebo nesplněný.

Hodnocení nefunkcionálních požadavků vyžadovalo komplexnější přístup zahrnující měření výkonu, analýzu uživatelské zkušenosti a ověření přesnosti algoritmů detekce porozity. Výsledek byl následně taktéž hodnocen jako splněný nebo nesplněný. Měření rychlosti odezvy aplikace bylo realizováno pomocí vestavěných nástrojů pro profilování

výkonu ve vývojovém prostředí Xcode, což umožnilo přesné stanovení doby trvání jednotlivých operací.

Zásadním aspektem testovací metodiky bylo ověření funkčnosti aplikace při různých světelných podmínkách, které odpovídají reálnému prostředí průmyslové výroby pneumatik. Testování bylo provedeno za kontrolovaného umělého osvětlení pomocí LED světel s nastavitelnou intenzitou a teplotou, za přirozeného denního světla a za smíšeného osvětlení kombinujícího umělé a přirozené světelné zdroje. Tato variabilita světelných podmínek byla nezbytná pro ověření robustnosti algoritmů zpracování obrazu a jejich schopnosti adaptace na různé fotografické situace vyskytující se ve výrobní praxi.

Pro zajištění reprezentativního hodnocení uživatelské zkušenosti byla sestavena testovací skupina skládající se ze tří uživatelů s různou úrovní odborných znalostí a zkušeností s porozitou při výrobě pneumatik. První uživatel disponoval kompletní znalostí aplikace včetně detailů její implementace a reprezentoval tak vývojářský pohled na funkcionalitu systému. Druhý uživatel měl rozsáhlé znalosti v oblasti kontroly kvality pneumatik a problematiky porozity materiálů, avšak bez hlubokého porozumění technické implementaci aplikace, čímž napodoboval zkušeného pracovníka výroby s doménovou expertízou. Třetí uživatel byl bez předchozích zkušeností s podobnými aplikacemi nebo s problematikou porozity, který byl pouze základně instruován k používání aplikace. Tato poslední role byla záměrně zvolena tak, aby reprezentovala nového zaměstnance ve výrobě, který ještě nemá vybudované doménové znalosti, ale potřebuje aplikaci efektivně využívat.

Každý testovací uživatel provedl standardizovanou sadu úkolů zahrnující pořízení fotografií vzorků pneumatik, výběr oblasti zájmu, analýzu porozity a export výsledků do CSV formátu. Výsledky jednotlivých uživatelů byly následně porovnány s referenčními hodnotami získanými laboratorními metodami, což umožnilo kvantifikovat přesnost aplikace a identifikovat případné rozdíly v použitelnosti mezi uživateli s různou úrovní odborných znalostí.

9.2 Výsledky testování

V tabulce číslo 3 jsou uvedeny výsledky testování funkcionálních požadavků. Každý požadavek byl hodnocen binárně jako splněný nebo nesplněný na základě úspěšného provedení definovaných testovacích scénářů.

Tabulka 3 Výsledky testování funkcionálních požadavků na aplikaci

ID	Popis požadavku	Výsledek
F01	Aplikace umožní pořízení nové fotografie pomocí vestavěné kamery zařízení.	Splněn
F02	Aplikace umožní načtení existující fotografie ze zařízení nebo cloudového úložiště.	Splněn
F03	Uživatel může vybrat oblast zájmu (řez pneumatiky) na načtené fotografii.	Splněn
F04	Aplikace automaticky analyzuje vybranou oblast a detekuje póry (porozitu).	Splněn
F05	Aplikace vypočítá procentuální zastoupení porozity vůči celkové ploše materiálu.	Splněn
F06	Výsledky analýzy jsou zobrazeny numericky s přesností na tři desetinná místa.	Splněn
F07	Na analyzovaném snímku jsou detekované bubliny vizuálně označeny.	Splněn
F08	Uživatel může k analýze doplnit informace o použitém materiálu.	Splněn
F09	Uživatel může k analýze doplnit informace o použité výrobní lince.	Splněn
F10	Uživatel může k analýze doplnit libovolnou textovou poznámku.	Splněn
F11	Každá provedená analýza je automaticky uložena do databáze aplikace.	Splněn
F12	Domovská obrazovka zobrazuje seznam všech provedených analýz.	Splněn
F13	Uživatel může vyhledávat v seznamu analýz podle klíčových slov a tím filtrovat analýzy například podle datumu pořízení, materiálu, výrobní linky nebo poznámky.	Splněn
F14	Uživatel může zobrazit detail jednotlivé analýzy se všemi uloženými údaji.	Splněn
F15	Uživatel může exportovat data jednotlivého záznamu do CSV formátu.	Splněn
F16	Uživatel může exportovat všechna data pro konkrétní směs do CSV formátu.	Splněn
F17	Uživatel může exportovat všechna data pro konkrétní výrobní linku do CSV formátu.	Splněn
F18	Uživatel může smazat vybranou analýzu z aplikace.	Splněn

V tabulce číslo 4 jsou uvedeny výsledky testování nefunkčních požadavků.

Tabulka 4 Nefunkční požadavky na aplikaci

ID	Popis požadavku	Výsledek
NF01	Aplikace je v anglickém jazyce.	Splněn
NF02	Analýza jedné fotografie je dokončena do 5 sekund na aktuální 16. generaci iPhone zařízení.	Splněn
NF03	Aplikace funguje offline bez nutnosti internetového připojení.	Splněn
NF04	Velikost aplikace nepřesáhne 100 MB po instalaci.	Splněn
NF05	Exportované CSV soubory jsou kompatibilní s programy Apple Numbers, Microsoft Excel a Google Sheets.	Splněn
NF06	Aplikace je responzivní a reaguje na uživatelské akce do 0,5 sekundy.	Splněn
NF07	Aplikace je optimalizována pro různé velikosti obrazovek zařízení iPhone.	Splněn
NF08	Aplikace nepotřebuje externí závislosti nebo síťové služby pro základní fungování.	Splněn
NF09	Přesnost detekce porozity má za optimálních světelných podmínek odchylku maximálně ± 1 % oproti referenčnímu laboratornímu měření.	Splněn

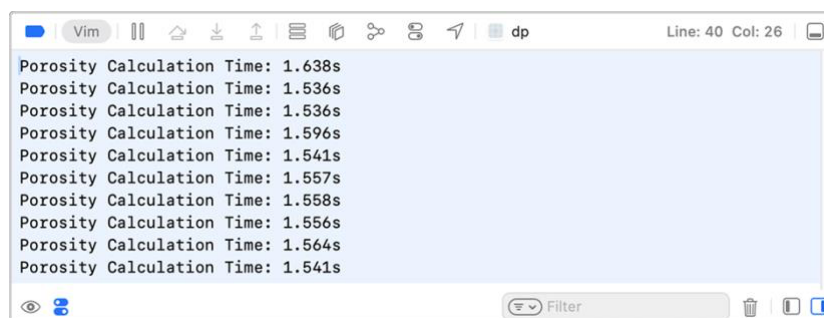
9.3 Shrnutí testování

Validační testování vyvinuté mobilní aplikace pro detekci porozity pneumatik bylo provedeno na základě předem definovaných funkčních a nefunkčních požadavků. Testovací proces byl realizován prostřednictvím standardizované metodiky zahrnující tři testovací subjekty s různou úrovní odborných znalostí a zkušeností v problematice kontroly kvality pneumatik.

Výsledky testování potvrdily splnění všech požadavků u všech tří testovacích subjektů. Funkcionalita aplikace byla ověřena napříč celým spektrem definovaných scénářů použití, přičemž nebyly identifikovány žádné defekty nebo nedostatky v implementaci základních funkcí systému. Testovací subjekty úspěšně realizovaly všechny požadované operace včetně pořizování fotografií, výběru oblasti zájmu, provedení analýzy porozity, správy dat a exportu výsledků do formátu CSV.

Hodnocení některých nefunkčních požadavků vyžadovalo detailnější analýzu měřitelných charakteristik aplikace. Například požadavek NF02 týkající se rychlosti analýzy byl ověřen pomocí přesného měření času prostřednictvím funkce v kódu

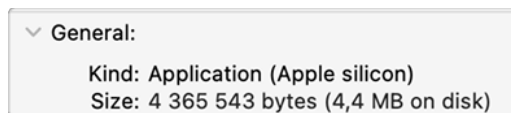
CFAbsoluteTimeGetCurrent, jejíž výstupy je možné vidět v prostředí Xcode na obrázku číslo 50.



Obrázek 50 Výstupy měření doby výpočtu porozity

Na základě desíti měření byla stanovena průměrná doba výpočtu analýzy jedné fotografie na 1,56 sekundy při testování na simulátoru zařízení iPhone 16 Pro. Tento výsledek výrazně překonává stanovený požadavek maximální doby pěti sekund a potvrzuje efektivnost implementovaných algoritmů zpracování obrazu.

Velikost aplikace byla na zařízení 4,4 MB po instalaci, což splňuje požadavek NF04 stanovující maximální limit 100 MB. Tato kompaktní velikost aplikace zajišťuje rychlou instalaci a minimální nároky na úložný prostor mobilních zařízení. Výstup informačního dialogu zobrazujícího informace o aplikaci je možné vidět na obrázku číslo 51.

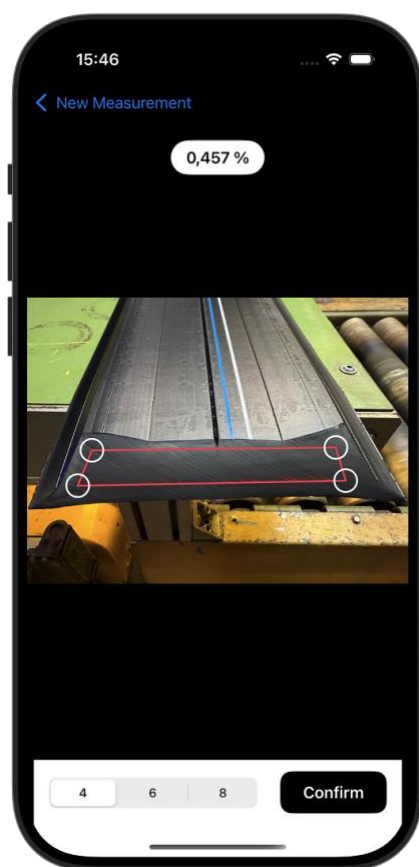


Obrázek 51 Velikost aplikace po instalaci

Kritickým aspektem hodnocení byla validace přesnosti detekce porozity definovaná v požadavku NF09. Testovací subjekty byly instruovány k pořizování fotografií kolmo vůči řezu vzorku se zaostřením na oblast řezu. Za optimálních podmínek neměl žádný z testovacích subjektů obtíže s dosažením požadované přesnosti měření. Významným pozorováním bylo zjištění, že všechny tři kategorie uživatelů s různou úrovní zkušeností dosahovaly srovnatelných výsledků, což dokumentuje robustnost aplikace a její nezávislost na odborných znalostech operátora.

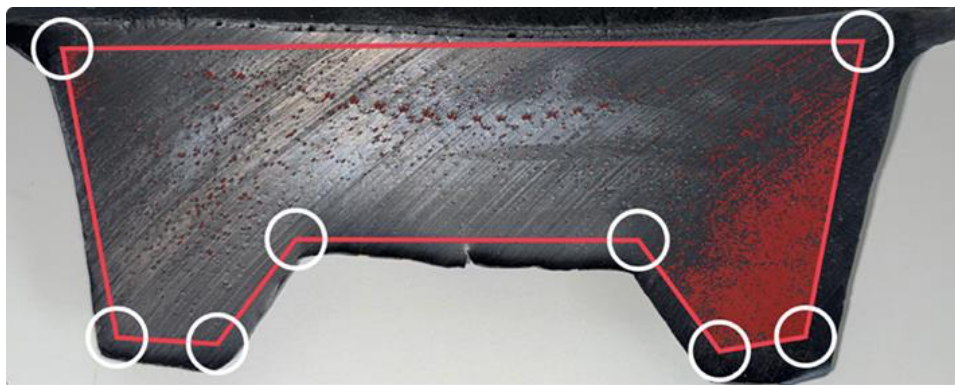
Úspěšné splnění požadavku NF09 je zachyceno při měření vzorku za optimálních podmínek na obrázku číslo 52, kde aplikace naměřila porozitu 0,457 %. Laboratorní referenční metody pro tentýž vzorek stanovily porozitu na hodnotě 0,3724 %. Tato odchylka představuje rozdíl přibližně 0,08 %, což spadá do tolerance očekávané u porovnání

povrchové optické analýzy s objemovými laboratorními technikami a potvrzuje praktickou použitelnost aplikace pro detekci porozity pneumatik.



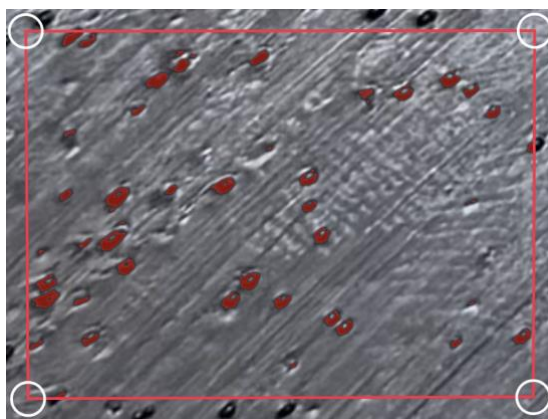
Obrázek 52 Správné měření porozity aplikací

Následná analýza faktorů ovlivňujících přesnost měření odhalila několik kritických aspektů. Světelné podmínky byly identifikovány jako zásadní parametr ovlivňující kvalitu detekce porozity. Optimální výsledky byly dosaženy za podmínek přirozeného intenzivního denního světla, zatímco využití běžně dostupného umělého osvětlení, včetně stolních lamp s nastavitelnou silou a teplotou světla, vedlo k neuspokojivému poklesu přesnosti měření. Podobně neuspokojivé výsledky byly zaznamenány při použití vestavěného blesku mobilního zařízení, který způsoboval lokální osvětlení vzorku s nerovnoměrným rozložením intenzity světla v kruhovém tvaru. Na obrázku číslo 53 lze vidět, jak použití blesku způsobuje nerovnoměrné osvětlení vzorku, což vede k nedostatečné přesnosti výpočtu porozity.



Obrázek 53 Nedostačující použití zabudovaného blesku zařízení k přisvětlení vzorku

Při kombinaci kvalitního osvětlení a materiálů s velmi vysokou porozitou se občas projevil specifický jev, kdy se světlo odrazilo ze dna velkých pórů, jak je možné pozorovat na obrázku číslo 54. Tyto odrazy způsobovaly, že algoritmus nesprávně vyhodnotil dané oblasti jako materiál místo porozity, což vedlo k systematickému podhodnocení skutečné porozity vzorku. Tento efekt se však vyskytoval pouze u méně běžných vzorků s vysokou porozitou a velkými póry.



Obrázek 54 Vliv velkých pórů na výpočet porozity

Kvalita řezu vzorku představovala druhý významný faktor ovlivňující přesnost analýzy. Přítomnost mechanických defektů, nečistot, kapalin nebo stop po řezacím nástroji na povrchu vzorku způsobovala vznik tmavých oblastí, které byly algoritmem nesprávně klasifikovány jako porozita. Tento jev vedl k systematickému nadhodnocení skutečné míry porozity a představuje nutnost standardizace postupů přípravy vzorků pro zajištění spolehlivých výsledků měření.

Validace výsledků aplikace byla realizována porovnáním s referenčními hodnotami získanými laboratorními metodami. Během tohoto procesu byla identifikována principiální

limity optické metody spočívající v tom, že poměr pórů na povrchu řezu nemusí nutně odpovídat celkové objemové porozitě vzorku. Byly pozorovány případy, kdy vizuální inspekce řezu neodhalila přítomnost porozity, zatímco laboratorní analýza prokázala porozitu na úrovni $\pm 0,5$ %. Toto pozorování ukazuje, že výsledná odchylka měření není způsobena pouze chybou algoritmu aplikace, ale také limitací povrchové optické analýzy ve srovnání s laboratorními technikami.

Celkové hodnocení testovacího procesu potvrdilo funkčnost a použitelnost aplikace pro detekci porozity pneumatik. Aplikace splnila definované požadavky a prokázala schopnost poskytovat přesné a opakovatelné výsledky za předpokladu dodržení optimálních podmínek měření. Pro zaručení optimálních výsledků se ukázalo jako vhodné volit prahovou hodnotu pro výpočet porozity s ohledem na konkrétní světelné podmínky při měření. Identifikované limity související se světelnými podmínkami a kvalitou vzorků představují oblasti pro budoucí vylepšení a optimalizaci systému pro zaručení stabilních výsledků v různých světelných podmínkách.

10 ZHODNOCENÍ VYTVOŘENÉ APLIKACE

Realizované validační testování prototypu mobilní aplikace poskytlo komplexní přehled o funkčnosti, výkonnosti a praktické použitelnosti vyvinutého řešení pro detekci porozity pneumatik. Výsledky testování prokázaly schopnost aplikace fungovat jako alternativa současných metod hodnocení porozity materiálů v gumárenském průmyslu a potvrdily splnění všech definovaných požadavků.

10.1 Přednosti vytvořené aplikace

Během testovacího procesu byly identifikovány významné přednosti vyvinutého řešení. Za optimálních světelných podmínek bylo dosaženo požadované přesnosti detekce porozity oproti referenčním laboratorním metodám. Průměrná doba zpracování jedné fotografie byla naměřena na úrovni 1,56 sekundy, což představuje zásadní zrychlení oproti současným laboratorním metodám, kde získání výsledků může v důsledku firemních procesů trvat i několik dní. Uživatelské rozhraní bylo testovacími uživateli hodnoceno jako intuitivní a responzivní, přičemž doba odezvy na uživatelské akce nepřesahovala stanovený limit dvou sekund. Kompaktní velikost aplikace 4,4 MB zajišťuje minimální nároky na úložný prostor mobilních zařízení a umožňuje rychlou distribuci řešení. Offline funkcionality eliminuje závislost na síťovém připojení během měřicích procesů, což je výhodné pro provozní podmínky. Implementovaná funkcionality exportování dat ve formátu CSV byla ověřena jako plně kompatibilní se standardními tabulkovými procesory, což usnadňuje integraci výsledků analýz do stávajících výrobních systémů a podporuje dlouhodobou archivaci dat.

10.2 Limity vytvořené aplikace

Testovací proces zároveň odhalil několik limitací ovlivňujících praktickou použitelnost aplikace v různých provozních podmínkách. Jako kritický faktor byly identifikovány světelné podmínky, kdy optimální výsledky byly dosaženy pouze za přirozeného intenzivního denního světla. Využití běžně dostupného domácího či kancelářského umělého osvětlení nebo vestavěného blesku mobilního zařízení vedlo k neuspokojivému poklesu přesnosti měření. Pro nasazení ve výrobních podmínkách je proto nezbytné zajistit vhodný světelný zdroj a následně provést kalibraci algoritmu pro tyto specifické podmínky.

Kvalita řezu vzorku byla identifikována jako druhý významný limitující faktor. Přítomnost mechanických defektů, nečistot nebo stop po řezacím nástroji způsobovala

systematické nadhodnocení porozity. Pro zvýšení přesnosti analýzy je proto nutné ze vzorků před samotným měřením odstranit nečistoty a kapaliny, pokud je tak technicky možné učinit.

Během validačního procesu byly dále zjištěny také limity optické metody detekce porozity. Hlavní omezení spočívá v analýze pouze povrchové porozity na řezu vzorku oproti objemové porozitě stanovované laboratorními technikami. Během testování byly zaznamenány případy, kdy se optická metoda výrazně odlišovala od výstupů laboratorní analýzy. Tato skutečnost vyplývá z nemožnosti optické metody detekovat póry, které nejsou přítomny na analyzovaném povrchu řezu.

10.3 Možné směry budoucího rozvoje aplikace

Na základě získaných poznatků z testovacího procesu lze identifikovat čtyři perspektivní směry budoucího rozvoje aplikace. Prvním směrem je vývoj adaptivních algoritmů schopných kompenzovat různé světelné podmínky nebo případná integrace specializovaných externích světelných zdrojů s kontrolovanými parametry osvětlení do procesu analýzy. Druhým směrem rozvoje je rozšíření funkcionality o automatickou detekci kvality řezu vzorku, což by umožnilo eliminaci defektů nalezených v řezu nebo jejich kompletní vyřazení pro účely posuzování porozity. Třetí oblastí je implementace kalibračních mechanismů pro různé typy světelných podmínek, materiálů a směsí, která by mohla vést ke zvýšení přesnosti měření bez nutnosti modifikace zdrojového kódu aplikace. Čtvrtým možným směrem je vývoj algoritmů pro odhad objemové porozity na základě analýzy více řezů stejné směsi nebo implementace strojového učení pro zlepšení přesnosti detekce za různých podmínek.

10.4 Shrnutí hodnocení vyvinuté aplikace

Vyvinutý prototyp mobilní aplikace představuje funkční řešení pro detekci porozity pneumatik, které využívá potenciál moderních mobilních technologií platformy iOS. Během validačního testování byla prokázána schopnost aplikace dosahovat výsledků srovnatelných se zavedenými laboratorními metodami. Oproti současným přístupům založených na subjektivní evaluaci zkušených pracovníků nebo časově a finančně náročných laboratorních

analýzách bylo dosaženo objektivního a rychlého vyhodnocení porozity materiálů.

Praktický přínos vyvinutého řešení spočívá v představení problematiky detekce porozity v gumárenském průmyslu a vyvinutí konkrétního technologického řešení v podobě

prototypové mobilní aplikace pro platformu iOS. Realizovaná aplikace úspěšně prokázala spojení přesnosti měření s okamžitou dostupností výsledků, čímž překonává hlavní omezení současných přístupů. Vyvinuté řešení tak vytváří předpoklady pro zásadní zefektivnění procesů kontroly kvality ve výrobě pneumatik prostřednictvím objektivního a rychlého vyhodnocení porozity materiálů. Současně byly identifikovány limity prototypové aplikace a definovány perspektivní směry možného budoucího rozvoje. Tato práce tak poskytuje jak funkční prototyp, tak i ucelený základ pro další rozvoj aplikací využívaných v oblasti automatizované detekce a kvantifikace porozity pomocí mobilních technologií.

ZÁVĚR

Diplomová práce se zabývala problematikou využití mobilních technologií pro objektivní odhad porozity vzorků pneumatik. Práce byla rozdělena na část teoretickou a praktickou.

V teoretické části byly popsány technologie a nástroje potřebné pro vývoj specializovaných iOS aplikací. Dále byly představeny mobilní platformy společnosti Apple s důrazem na operační systém iOS a jeho možnosti v oblasti zpracování obrazu. Detailně byl popsán programovací jazyk Swift, framework SwiftUI pro tvorbu uživatelského rozhraní a technologie SwiftData pro správu perzistentních dat. Zvláštní pozornost byla věnována knihovně Core Image a jejímu využití pro analýzu fotografií. V závěru teoretické části byl vymezen pojem porozita a jeho význam.

V praktické části byla nejprve provedena analýza existujících mobilních aplikací, která odhalila prostor pro vývoj inovativního řešení v oblasti detekce porozity. Následně byl popsán kompletní proces vývoje funkčního prototypu aplikace zahrnující specifikaci cílů a požadavků, návrh drátěných modelů a datového modelu, implementaci včetně vytvoření projektu, konfigurace aplikace, architektury systému, uživatelského rozhraní a popsání klíčových částí aplikace. Validační testování prokázalo splnění definovaných cílů a požadavků aplikace. Byly identifikovány limity související především se světelnými podmínkami a kvalitou vzorků. Poslední kapitola praktické části obsahuje hodnocení vytvořeného prototypu a jsou v ní identifikovány možné budoucí směry vývoje.

Výsledkem práce je plně funkční iOS aplikace, která úspěšně představila možnosti využití mobilních zařízení pro průmyslovou kontrolu kvality při výrobě pneumatik a poskytuje tak unikátní využitelné řešení v této oblasti. Teoretické poznatky z oblasti vývoje iOS aplikací a zpracování obrazu byly aplikovány do praktického řešení reálného průmyslového problému. Aplikace prokázala schopnost poskytovat rychlé a objektivní výsledky měření porozity. Byly identifikovány také limity související především se světelnými podmínkami a kvalitou vzorků. Pro budoucí rozvoj bylo identifikováno několik klíčových oblastí.

SEZNAM POUŽITÉ LITERATURY

- [1] POSEY, Brien. *Apple iOS*. Online. TechTarget. 2025. Dostupné z: <https://www.techtarget.com/searchmobilecomputing/definition/iOS>. [cit. 2025-02-10].
- [2] *What is iOS app development?* Online. IBM. 2025. Dostupné z: <https://www.ibm.com/think/topics/ios-app-development>. [cit. 2025-02-10].
- [3] VOLVOIKAR, Palash. Online. What is iPadOS? Everything you need to know about Apple's iPad operating system. *ANDROID AUTHORITY*. 2021. Dostupné z: <https://www.androidauthority.com/what-is-ipad-os-3072887/>. [cit. 2025-02-10].
- [4] LEFEBVRE, Rob. What Is watchOS? Online. *Lifewire*. 2024. Dostupné z: <https://www.lifewire.com/what-is-watchos-4690550>. [cit. 2025-03-04].
- [5] Apple Ecosystem Explained 2024. Online *Tech Journeyman*. 2023. Dostupné z: <https://techjourneyman.com/blog/apple-ecosystem-explained-2024/>. [cit. 2025-03-04].
- [6] App Store. Online. *Appleinsider*. 2025 Dostupné z: <https://appleinsider.com/inside/app-store> [cit. 2025-03-05].
- [7] POROZITA. Online. *ATRIA CZ*. 2025 Dostupné z: <https://www.atria-europe.com/cs/slovník-pojmu/porozita/> [cit. 2025-03-08].
- [8] How to measure the materials porosity. Online. *ATRIA*. 2025 Dostupné z: <https://atriainnovation.com/en/blog/how-to-measure-the-materials-porosity/> [cit. 2025-03-12].
- [9] Density and Porosity Measurements of Solid Materials. Online. *Anderson Materials Evaluation, Inc*. 2024 Dostupné z: <https://andersonmaterials.com/density-and-porosity-measurements-of-solid-materials/> [cit. 2025-03-12].
- [10] Understanding Porosity and Density. Online. *WGNHS*. 2024 Dostupné z: <https://home.wgnhs.wisc.edu/water/wisconsin-aquifers/understanding-porosity-density/#main> [cit. 2025-03-12].
- [11] What Is iOS Development? Online. *BrainStation*. 2025 Dostupné z: <https://brainstation.io/career-guides/what-is-ios-development> [cit. 2025-03-18].
- [12] Swift. Online. *Apple Developer*. 2025 Dostupné z: <https://developer.apple.com/swift> [cit. 2025-03-18].

- [13] A Swift Tour. Online. *Swift*. 2025 Dostupné z: <https://docs.swift.org/swift-book/documentation/the-swift-programming-language/guidedtour/> [cit. 2025-03-18].
- [14] Native, hybrid, or cross-platform apps? Online. *Microsoft*. 2025 Dostupné z: <https://www.microsoft.com/en-us/power-platform/products/power-apps/topics/app-development/native-vs-cross-platform-apps> [cit. 2025-03-25].
- [15] Cross-platform and native app development: How do you choose? Online. *Kotlin Multiplatform*. 2025 Dostupné z: <https://www.jetbrains.com/help/kotlin-multiplatform-dev/native-and-cross-platform.html> [cit. 2025-03-25].
- [16] The Six Most Popular Cross-Platform App Development Frameworks. Online. *Kotlin Multiplatform*. 2025 Dostupné z: <https://www.jetbrains.com/help/kotlin-multiplatform-dev/cross-platform-frameworks.html> [cit. 2025-03-25].
- [17] GRIFFITH, Chris. What is Hybrid Mobile App Development? Online. *Ionic*. 2025 Dostupné z: <https://ionic.io/resources/articles/what-is-hybrid-app-development> [cit. 2025-03-25].
- [18] What is iOS app development? Online. *IBM*. 2025 Dostupné z: <https://www.ibm.com/think/topics/ios-app-development> [cit. 2025-03-26].
- [19] LETTIER, Mariel. Swift iOS Programming Language. *Capicua* [online]. 2025. Dostupné z: <https://www.capicua.com/blog/swift-programming-language>. [cit. 2025-03-26].
- [20] About Swift. Online. *Swift*. 2025 Dostupné z: <https://www.swift.org/about/> [cit. 2025-03-26].
- [21] *The History of Hello World: A Brief Overview*. Online. *CodeInterview*. 2024. Dostupné z: <https://codeinterview.io/blog/the-history-of-hello-world-a-brief-overview/>. [cit. 2025-03-28].
- [22] *SwiftUI*. Online. *Apple Developer*. 2025. Dostupné z: <https://developer.apple.com/xcode/swiftui/>. [cit. 2025-04-11].
- [23] CHING, Chris. *What is SwiftUI?* Online. *Codewithchris*. 2024. Dostupné z: <https://codewithchris.com/what-is-swiftui/>. [cit. 2025-04-12].

- [24] HUDSON, Paul. *What is SwiftUI?* Online. HACKING WITH SWIFT. 2024. Dostupné z: <https://www.hackingwithswift.com/quick-start/swiftui/what-is-swiftui>. [cit. 2025-04-13].
- [25] HUDSON, Paul. *View composition*. Online. HACKING WITH SWIFT. 2023. Dostupné z: <https://www.hackingwithswift.com/books/ios-swiftui/view-composition>. [cit. 2025-04-17].
- [26] *Using View Composition in SwiftUI*. Online. Kodeco. 2025. Dostupné z: <https://www.kodeco.com/books/swiftui-cookbook/v1.0/chapters/9-using-view-composition-in-swiftui>. [cit. 2025-04-26].
- [27] *Best Practices for State Management in SwiftUI*. Online. Kodeco. 2025. Dostupné z: <https://www.kodeco.com/books/swiftui-cookbook/v1.0/chapters/8-best-practices-for-state-management-in-swiftui>. [cit. 2025-04-26].
- [28] MAGARE, Darshan. *A Quick Guide To SwiftUI Modifiers*. Online. IT Path Solutions. 2023. Dostupné z: <https://www.itpathsolutions.com/a-quick-guide-to-swiftui-modifiers/>. [cit. 2025-04-26].
- [29] NDUNGU, Dedan. *SwiftUI vs. UIKit: What is the best choice for building an iOS user interface in 2024?* Online. Sendbird. 2025. Dostupné z: <https://sendbird.com/developer/tutorials/swiftui-vs-uikit>. [cit. 2025-04-27].
- [30] *SwiftUI vs UIKit: A Comparison*. Online. Alibaba Cloud. 2024. Dostupné z: <https://www.alibabacloud.com/tech-news/a/swift/gv9vmn0hm0-swiftui-vs-uikit-a-comparison>. [cit. 2025-04-27].
- [31] *Navigation*. Online. Apple Developer. 2025. Dostupné z: <https://developer.apple.com/documentation/swiftui/navigation>. [cit. 2025-04-28].
- [32] *Understanding the navigation stack*. Online. Apple Developer. 2025. Dostupné z: <https://developer.apple.com/documentation/swiftui/understanding-the-composition-of-navigation-stack>. [cit. 2025-04-29].
- [33] *Foundation*. Online. Apple Developer. 2025. Dostupné z: <https://developer.apple.com/documentation/foundation/>. [cit. 2025-04-30].
- [34] HUDSON, Paul. *What is SwiftData?* Online. HACKING WITH SWIFT. 2023. Dostupné z: <https://www.hackingwithswift.com/quick-start/swiftdata/what-is-swiftdata>. [cit. 2025-04-30].

- [35] *SwiftData*. Online. Apple Developer. 2025. Dostupné z: <https://developer.apple.com/documentation/swiftdata>. [cit. 2025-04-30].
- [36] *ModelContainer*. Online. Apple Developer. 2025. Dostupné z: <https://developer.apple.com/documentation/swiftdata/modelcontainer>. [cit. 2025-04-30].
- [37] HUDSON, Paul. *What's the difference between ModelContainer, ModelContext, and ModelConfiguration?* Online. Apple Developer. 2023. Dostupné z: <https://www.hackingwithswift.com/quick-start/swiftdata/whats-the-difference-between-modelcontainer-modelcontext-and-modelconfiguration>. [cit. 2025-04-30].
- [38] HUDSON, Paul. *How to define SwiftData models using the @Model macro*. Online. HACKING WITH SWIFT. 2023. Dostupné z: <https://www.hackingwithswift.com/quick-start/swiftdata/how-to-define-swiftdata-models-using-the-model-macro>. [cit. 2025-05-06].
- [39] HUDSON, Paul. *Inferred vs explicit relationships*. Online. HACKING WITH SWIFT. 2023. Dostupné z: <https://www.hackingwithswift.com/quick-start/swiftdata/inferred-vs-explicit-relationships>. [cit. 2025-05-06].
- [40] YADAV, Ashish. *Swift Data — Relationship — Part 2*. Online. Medium. 2023. Dostupné z: https://medium.com/@ashishyadav_16301/swift-data-relationship-2ce95520c2f4. [cit. 2025-05-06].
- [41] TUBAGA, Joash. *A Beginner's Ultimate Guide to SwiftData*. Online. Codewithchris. 2024. Dostupné z: <https://codewithchris.com/swift-data/>. [cit. 2025-05-06].
- [42] BADAR, Ehtisham. *SwiftData vs CoreData?* Online. Medium. 2023. Dostupné z: <https://medium.com/@ehtisham.badar/swiftdata-vs-coredata-54f6b8c47cbd>. [cit. 2025-05-06].
- [43] HUDSON, Paul. *SwiftData vs Core Data*. Online. Medium. 2023. Dostupné z: <https://www.hackingwithswift.com/quick-start/swiftdata/swiftdata-vs-core-data>. [cit. 2025-05-06].
- [44] VITTORIOSI, Pasquale. *Getting started with Core Image*. Online. Create with Swift. 2025. Dostupné z: <https://www.createwithswift.com/getting-started-with-core-image/>. [cit. 2025-05-09].

- [45] BOHON, Cory. *Xcode Through the Years*. Online. MartianCraft. 2022. Dostupné z: <https://martiancraft.com/blog/2022/01/xcode-through-the-years/>. [cit. 2025-05-10].
- [46] *Xcode*. Online. Apple Developer. 2025. Dostupné z: <https://developer.apple.com/xcode/>. [cit. 2025-05-13].
- [47] *Providing access to directories*. Online. Apple Developer. 2025. Dostupné z: <https://developer.apple.com/documentation/uikit/providing-access-to-directories>. [cit. 2025-05-15].
- [48] DORADO, Juan. *SwiftUI Architecture: A Complete Guide to the MV Pattern Approach*. Online. Swifty Journey. 2024. Dostupné z: <https://www.swiftyjourney.com/swiftui-architecture-a-complete-guide-to-the-mv-pattern-approach>. [cit. 2025-05-16].
- [49] COSTELLO, Sam. *Everything You Need to Know About the iPod Touch*. Online. Lifewire. 2021. Dostupné z: <https://www.lifewire.com/ipod-touch-everything-to-know-1999772>. [cit. 2025-05-16].
- [50] BECK, Austin. *Choosing the Right Distribution Method for Your iOS Applications*. Online. Medium. 2021. Dostupné z: <https://medium.com/geekculture/choosing-the-right-distribution-method-for-your-ios-applications-b260bddd4a12>. [cit. 2025-05-19].
- [51] *Use the Measure app on your iPhone, iPad, or iPod touch*. Online. Apple. 2025. Dostupné z: <https://support.apple.com/en-us/102468>. [cit. 2025-05-22].
- [52] *Phypox*. Online. App Store Preview. 2025. Dostupné z: <https://apps.apple.com/us/app/phyphox/id1127319693>. [cit. 2025-05-23].
- [53] *Phypox*. Online. Google Play. 2025. Dostupné z: https://play.google.com/store/apps/details?id=de.rwth_aachen.phyphox&hl=en. [cit. 2025-05-23].
- [54] *Infor LN Quality Inspections*. Online. Google Play. 2025. Dostupné z: <https://play.google.com/store/apps/details?id=com.infor.LN.QualityInspections>. [cit. 2025-05-23].
- [55] *NEOGAGE Quality Control*. Online. NEOGAGE Quality Control. 2025. Dostupné z: <https://qc.neogage.com/en/#features>. [cit. 2025-05-23].
- [56] SOMMERVILLE, Ian. *Softwarové inženýrství*. Brno: Computer Press, 2013. ISBN 978-80-251-3826-7

SEZNAM OBRÁZKŮ

Obrázek 1 Program <i>Hello, World!</i> v jazyce Swift	23
Obrázek 2 Proměnné a konstanty v jazyce Swift.....	25
Obrázek 3 Řízení toku programu v jazyce Swift	26
Obrázek 4 Deklarace funkce v jazyce Swift	27
Obrázek 5 Deklarace třídy a její použití v jazyce Swift	28
Obrázek 6 Výčtové typy a struktury v jazyce Swift	29
Obrázek 7 Asynchronní programování v jazyce Swift	30
Obrázek 8 Protokol a rozšíření v jazyce Swift.....	31
Obrázek 9 Zpracování chyb v jazyce Swift	32
Obrázek 10 Ukázka generické funkce v jazyce Swift.....	33
Obrázek 11 Kompozice <i>views</i> ve SwiftUI	34
Obrázek 12 Správa stavu ve SwiftUI	36
Obrázek 13 Modifikátory ve SwiftUI	37
Obrázek 14 Práce s daty ve SwiftUI	42
Obrázek 15 Návrhy hlavních obrazovek aplikace	58
Obrázek 16 Návrhy obrazovek pro zvolení vstupní fotografie	60
Obrázek 17 Návrhy obrazovek pro vytvoření nového záznamu.....	61
Obrázek 18 Návrh obrazovky pro vytvoření doplňujících datových struktur	62
Obrázek 19 Návrh obrazovky pro zobrazení detailního pohledu na existující záznam....	63
Obrázek 20 Úvodní obrazovka aplikace Xcode.....	68
Obrázek 21 Volba šablony v prostředí Xcode	69
Obrázek 22 Konfigurace možností šablony projektu v Xcode	70
Obrázek 23 Přidání oprávnění pro přístup ke galerii a fotoaparátu zařízení	72
Obrázek 24 Struktura projektu prototypové aplikace	73
Obrázek 25 Implementace datového kontejneru.....	74
Obrázek 26 Vlastnosti hlavní třídy Record.....	75
Obrázek 27 Implementace ukládání obrázků v aplikaci do perzistentního úložiště	76
Obrázek 28 Implementace pomocné struktury PointDTO.....	77
Obrázek 29 Implementace třídy pro ukládání směsí.....	77
Obrázek 30 Hlavní navigace v aplikaci	80
Obrázek 31 Domovská obrazovka aplikace a její funkce	81
Obrázek 32 Nahrání fotografie do aplikace	82
Obrázek 33 Vytvoření nového záznamu měření.....	84
Obrázek 34 Přidání doplňujících informací k záznamu.....	85

Obrázek 35 Detail měření, export a sdílení výsledků	86
Obrázek 36 Definice stavových proměnných použitých k výběru oblasti fotografie k analýze porozity	87
Obrázek 37 Vrstvená struktura pro zobrazení interaktivních bodů a detekované porozity nad obrázkem	88
Obrázek 38 Interaktivní vrstva pro výběr požadované sekce k analýze porozity	89
Obrázek 39 Implementace logiky pro vykreslení a chování ohraničujících bodů	90
Obrázek 40 Implementace asynchronního výpočtu porozity	90
Obrázek 41 Rozšíření třídy UIImage	91
Obrázek 42 Ukládání stavu interakcí mezi uživatelem a aplikací	91
Obrázek 43 Hlavní funkce pro výpočet porozity	92
Obrázek 44 Aplikování černobílého filtru na vstupní snímek	93
Obrázek 45 Příprava binární masky pro výpočet porozity	94
Obrázek 46 Export jednotlivého záznamu do formátu CSV	96
Obrázek 47 Export kolekce záznamů do formátu CSV	96
Obrázek 48 Implementace sdílení CSV souborů	97
Obrázek 49 Náhled na exportovaná data směsi T1251 v aplikaci Numbers od Apple	98
Obrázek 50 Výstupy měření doby výpočtu porozity	103
Obrázek 51 Velikost aplikace po instalaci	103
Obrázek 52 Správné měření porozity aplikací	104
Obrázek 53 Nedostačující použití zabudovaného blesku zařízení k přisvícení vzorku	105
Obrázek 54 Vliv velkých pórů na výpočet porozity	105

SEZNAM TABULEK

Tabulka 1 Funkcionální požadavky na aplikaci.....	55
Tabulka 2 Nefunkcionální požadavky na aplikaci.....	56
Tabulka 3 Výsledky testování funkcionálních požadavků na aplikaci	101
Tabulka 4 Nefunkcionální požadavky na aplikaci	102

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

API	Application Programming Interface
ASTM	American Society for Testing and Materials
CPU	Central Processing Unit
CSS	Cascading Style Sheets
CSV	Comma-Separated Values
DIN	Deutsches Institut für Normung
DTO	Data Transfer Object
EN	European Norm
GPS	Global Positioning System
GPU	Graphics Processing Unit
HTML	HyperText Markup Language
ID	Identifier
IDE	Integrated Development Environment
IUPAC	International Union of Pure and Applied Chemistry
JSON	JavaScript Object Notation
LiDAR	Light Detection and Ranging
LLVM	Low Level Virtual Machine
MB	Megabyte
MV	Model-View
ORM	Object-Relational Mapping
PNG	Portable Network Graphics
SDK	Software Development Kit
UI	User Interface
UX	User Experience
WWDC	Worldwide Developers Conferenc