Moderní metody tvorby nativních multiplatformních mobilních aplikací

Bc. Petr Čápek

Diplomová práce
2014

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
akademický rok: 2013/2014

ZADÁNÍ DIPLOMOVÉ PRÁCE
(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení: **Bc. Petr Čápek**
Osobní číslo: **A12701**
Studijní program: **N3902 Inženýrská informatika**
Studijní obor: **Informační technologie**
Forma studia: **prezenční**

Téma práce: **Moderní metody tvorby nativních multiplatformních mobilních aplikací**

Zásady pro vypracování:

1. **Vypracujte literární rešerši na dané téma.**
2. **Popište vhodné návrhové/architektonické vzory a programovací techniky.**
3. **Analyzujte možnosti tvorby aplikací pomocí frameworků Xamarin.iOS a Xamarin.Android.**
4. **Analyzujte možnosti knihovny MvvmCross.**
5. **Vytvořte případové studie pro řešení typických problémů.**
6. **Demonstrujte případové studie.**

Rozsah diplomové práce:

Rozsah příloh:

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

Seznam odborné literatury:

1. **SHACKLES, Greg. Mobile development with C. 1st ed. Sebastopol, CA: O'Reilly, c2012, xi, 155 p. ISBN 14-493-2023-6.**
2. **GAROFALO, Raffaele. Building enterprise applications with Windows Presentation Foundation and the model view ViewModel Pattern. Sebastopol, Calif: O'Reilly Media, 2011. ISBN 978-073-5650-923.**
3. **OLSON, Scott, John HUNTER, Ben HORGEN a Kenny GOERS. Cross-platform Mobile Development in c. 1st ed. Indianapolis: Wiley Pub., Inc., 2012, p. cm. ISBN 978-1-118-15770-1.**
4. **CHRISTIAN, Nagel, Bill EVJEN, Jay GLYNN, Karli WATSON a Morgan SKINNER. Professional C 2012 and .NET 4.5. New York: Wiley, 2012. ISBN 978-111-8332-122.**
5. **FOWLER, Martin. Patterns of enterprise application architecture. Boston: Addison-Wesley, c2003, xxiv, 533 s. The Addison-Wesley Signature Series. ISBN 978-0-321-12742-6.**

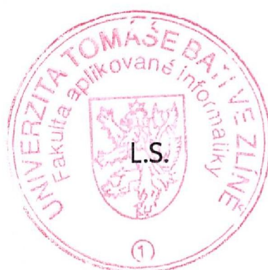
Vedoucí diplomové práce: **Ing. Erik Král**
Ústav počítačových a komunikačních systémů

Datum zadání diplomové práce: **21. února 2014**

Termín odevzdání diplomové práce: **20. května 2014**

Ve Zlíně dne 21. února 2014

prof. Ing. Vladimír Vašek, CSc.
děkan



L.S.

doc. Mgr. Roman Jašek, Ph.D.
ředitel ústavu

Prohlašuji, že

- beru na vědomí, že odevzdáním diplomové/bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že diplomová/bakalářská práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové/bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně a jeden výtisk bude uložen u vedoucího práce;
- byl/a jsem seznámen/a s tím, že na moji diplomovou/bakalářskou 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;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou/bakalářskou 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);
- beru na vědomí, že pokud bylo k vypracování diplomové/bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky diplomové/bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové/bakalářské 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 samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze diplomové/bakalářské práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně ……………………

podpis diplomanta

ABSTRAKT

Práce pojednává o současných multiplatformních technologiích pro tvorbu multiplatformních aplikací se zaměřením na framework Xamarin a framework MVVMCross. Dále se práce zabývá identifikací nejvhodnějších návrhových vzorů pro multiplatformní vývoj mobilních aplikací. Poznatky získané v teoretické části práce jsou poté aplikovány na případovou studii, kterou je původní mobilní aplikace určená pro uchazeče o studium na Univezitě Tomáše Bati ve Zlíně.

Klíčová slova:

Mono, Xamarin, iOS, Android, IoC, MVVMCross, Dependency Injection, .NET, C#

ABSTRACT

This works deals deals with state-of-the art technologies for developing multiplatform mobile applications. It focuses on Xamarin and MVVMCross frameworks and it also focuses on suitable software design patterns for developing mobile applications. The research work done in the theoretical part is the basis of the case study, which is a genuine, working mobile application for those applying to study at The Tomas Bata University in Zlin.

Keywords:

Mono, Xamarin, iOS, Android, IoC, MVVMCross, Dependency Injection, .NET, C#

Zde bych chtěl poděkovat vedoucímu práce Ing. et Ing. Eriku Královi, Ph.D. za odborné vedení, rady a konzultace diplomové práce.

OBSAH

ÚVOD ... **8**
I TEORETICKÁ ČÁST ... **9**
1 MOBILNÍ PLATFORMY ... **10**
1.1 ANDROID ... 11
1.2 IOS ... 12
1.3 WINDOWS PHONE ... 13
2 DRUHY MULTIPLATFORMNÍHO VÝVOJE ... **15**
2.1 NATIVNÍ TVORBA ... 17
2.2 WEBOVÉ APLIKACE ... 17
2.3 „WRITE ONCE, RUN ANYWHERE“ ... 19
2.4 XAMARIN ... 20
2.5 SHRNUTÍ ... 20
3 XAMARIN ... **22**
3.1 MONO ... 24
3.2 XAMARIN.ANDROID ... 26
3.3 XAMARIN.IOS ... 26
3.4 COMPONENT STORE ... 27
3.5 STRATEGIE SDÍLENÍ KÓDU ... 28
3.6 VÝVOJ POD XAMARINEM ... 31
4 NÁVRHOVÉ A ARCHITEKTONICKÉ VZORY ... **37**
4.1 FLUENT API ... 37
4.2 ARCHITEKTURY UŽIVATELSKÉHO ROZHRANÍ ... 38
4.3 DEPENDENCY INJECTION ... 39
4.4 IOC ... 42
4.5 PUBLISH - SUBSCRIBE NÁVRHOVÝ VZOR ... 44
4.6 ASYNCHRONNÍ NÁVRHOVÉ VZORY ... 46
4.6.1 APM ... 47
4.6.2 EAP ... 49
4.6.3 TAP ... 51
5 FRAMEWORK MVVMCROSS ... **54**
5.1 NAVIGACE ... 55
5.2 DATABINDING ... 56
5.3 PLUGINY ... 58
II PRAKTICKÁ ČÁST ... **62**
6 APLIKACE STUDUJ UTB ... **63**
7 CÍLE ... **64**
7.1 SPECIFIKACE APLIKACE ... 64
7.2 NEFUNKČNÍ POŽADAVKY ... 65
7.3 ARCHITEKTONICKÉ CÍLE ... 67
8 NÁVRH UŽIVATELSKÉHO ROZHRANÍ ... **68**

9 APLIKAČNÍ DIAGRAM **70**
9.1 VIEWS 71
9.2 VIEWMODELS 71
9.3 MODELS 72
9.4 SERVICES 72
9.5 PLATFORM DEPENDENT SERVICES 73
10 ROZBOR ARCHITEKTURY **75**
10.1 COMMANDY 75
10.2 IOC 76
10.3 NAVIGACE 80
10.3.1 Přímé předání parametru 81
10.3.2 Sdílená service 82
10.3.3 Zasílání zpráv 83
10.4 VÝSLEDNÁ APLIKACE 84
ZÁVĚR **87**
SEZNAM POUŽITÉ LITERATURY **88**
SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK **91**
SEZNAM OBRÁZKŮ **92**
SEZNAM GRAFŮ **93**
SEZNAM SCHÉMAT **94**
SEZNAM KÓDŮ **95**
SEZNAM PŘÍLOH **97**

ÚVOD

Mobilní platforma je nová a rozvíjející se platforma, a proto i technologie pro vývoj mobilních aplikací se velmi dynamicky vyvíjejí. V průběhu několika posledních let došlo v této oblasti k dramatickým změnám. Z prvotních zařízení, sloužících pouze pro volání a odesílání textových zpráv, se stala univerzální platforma poskytující komplexní služby jako geolokace, bezkontaktní platby, hraní 3D náročných her a další. Také se velice dramaticky změnil trh a výrobci mobilních zařízení. S tím souvisí nástup nových operačních systémů zaměřených speciálně pro mobilní zařízení. S jejich nástupem vznikla potřeba vyvinout nástroje pro tvorbu aplikací pro tyto jednotlivé systémy. Také se objevila snaha vyvinout nástroje, které by umožnily tvorbu univerzálních aplikací, které by bylo možné spouštět na všech operačních systémech a tím ušetřit náklady spojené s vývojem těchto aplikací.

Cílem této práce je porovnat existující technologie pro tvorbu multiplatformních aplikací se zaměřením se na framework Xamarin, který je založena na technologii mono, a také na framework MVVMCross, který rozšiřuje možnosti frameworku Xamarin. Dále se práce bude zabývat identifikací nejvhodnějších návrhových vzorů pro multiplatformní vývoj mobilních aplikací. Poznatky získané v teoretické části práce budou aplikovány na případovou studii, kde bude diskutován jejich přínos.

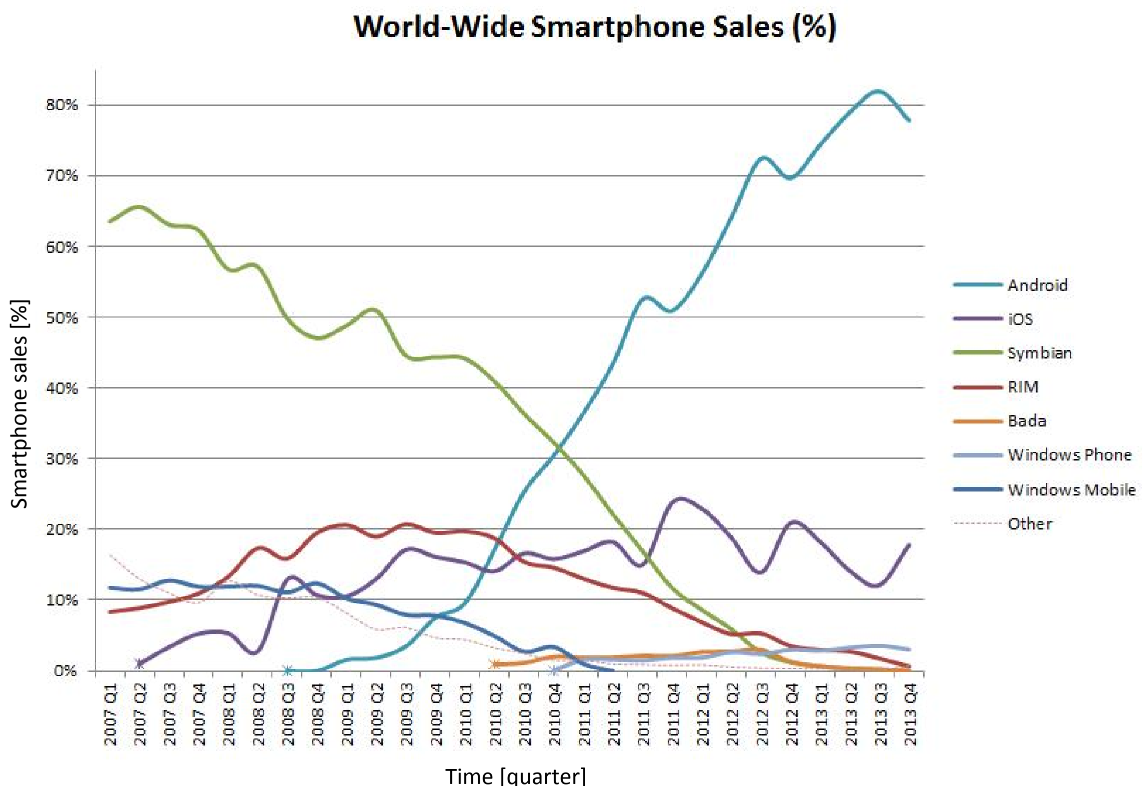
I. TEORETICKÁ ČÁST

1 MOBILNÍ PLATFORMY

Pojmem mobilní platformy můžeme souhrnně označit tzv. chytré telefony (smartphony), tablety, phablety a nejspíše by se tímto pojmem daly označit i další zařízení jako chytré hodinky, google glass a další. V rámci práce se zaměříme především na klasické mobilní zařízení a to jsou telefony a tablety [1].

Mobilní operační systémy se od klasických desktopových OS liší hlavně v tom, že v případě mobilních OS se jedná o systémy přizpůsobené dotykovému ovládání. Dále je v těchto systémech kladen důraz na úsporu v zacházení s dostupnými prostředky. Mobilní OS také často neumožňuje využívat klasický multitasking, jak jsme zvyklí z desktopových OS. Většinou na telefonu sice běží více procesů souběžně, ale uživatel může mít v jednu dobu aktivní pouze jeden program [1], [3].

Každý výrobce mobilního zařízení používá jiný operační systém na daném zařízení. S postupem času se také mění rozložení podílu těchto operačních systémů na trhu.



Graf 1 Podíl mobilních operačních systémů [6]

Z grafu můžeme vidět, že fragmentace napříč mobilními operačními systémy není aktuálně příliš veliká. Aktuálně je nejrozšířenější operační systém Android od Google, druhý nejrozšířenější OS je iOS od firmy Apple a třetí nejrozšířenější je Windows Phone od společnosti Microsoft. V dalších kapitolách si představíme stručně tyto operační systémy.

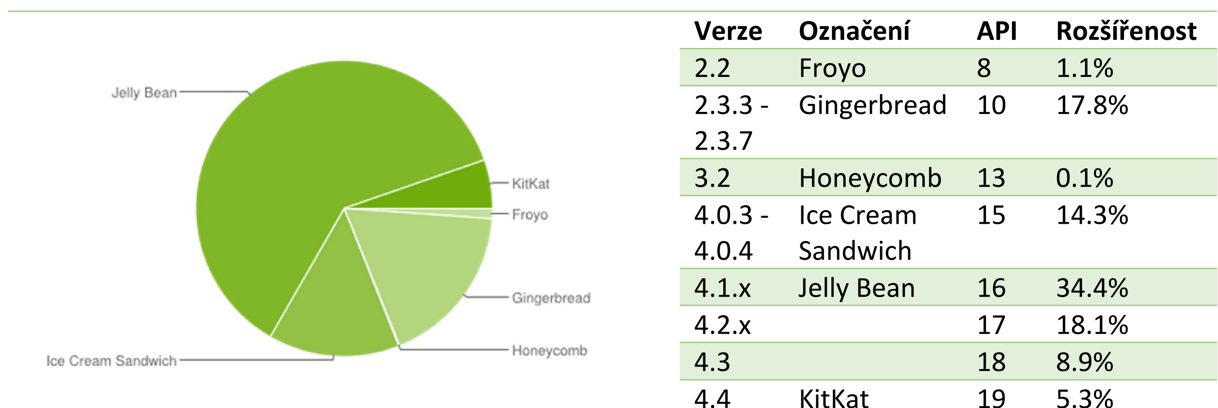
1.1 Android

Nejvíce rozšířeným mobilním operačním systémem je v současnosti Android od společnosti Google. Tento operační systém je založena na Linuxu (nižší vrstva) a Jave (vyšší vrstva).

Android sám o sobě je publikován jako opensource, avšak na koncových zařízení se velice často objevuje s nástavbou, kterou si přidávají samotní výrobci telefonu/tabletu, a tato nástavba již není opensource [7].

Distribuce aplikací do zařízení je možné provést buďto skrze oficiální „GooglePlay“, což je katalog aplikací. Aplikaci do katalogu může přidat každý, kdo vlastní vývojářskou licenci. Aplikace podstoupí schvalovací proces, který je prováděn automatizovaně a aplikace je většinou dostupná do několika hodin od prvotní publikace.

Android existuje v současnosti ve více verzích.



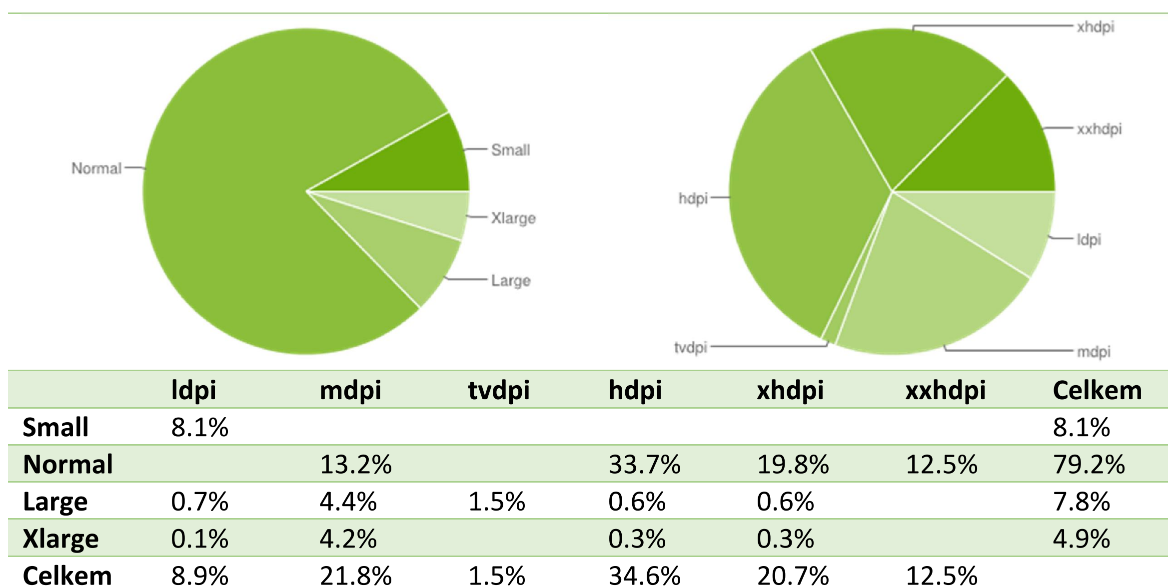
Verze	Označení	API	Rozšířenost
2.2	Froyo	8	1.1%
2.3.3 - 2.3.7	Gingerbread	10	17.8%
3.2	Honeycomb	13	0.1%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	14.3%
4.1.x	Jelly Bean	16	34.4%
4.2.x		17	18.1%
4.3		18	8.9%
4.4	KitKat	19	5.3%

Graf 2 Podíl verzí Androidu [8]

Android byl vytvořen tak, aby fungoval na velmi širokém spektru zařízení. Tato vlastnost umožňuje rozběhnout Android prakticky na jakémkoli zařízení. Ovšem s touto vlastností se váže také problém a tím je fragmentace verzemi Androidu napříč zařízeními. Výrobci sami většinou nevyvíjí moc velké úsilí, aby s příchodem nové verze systému Android provedli update pro starší zařízení. To dává prostor pro neoficiální verze systému Android, kteří se sdružují hlavně okolo fóra XDA-Developers.com a také CianogemMod.com. Na těchto

stránkách mohou uživatelé bezplatně stáhnout neoficiální verze operačního systému Android pro svá zařízení, avšak je třeba počítat s různou úrovní stability takto získaného OS [9], [10].

Další problém platformy Android je obrovská fragmentace zařízení napříč velikostmi obrazovek, jejich rozlišením a také různými poměry stran.



	ldpi	mdpi	tvdpi	hdpi	xhdpi	xxhdpi	Celkem
Small	8.1%						8.1%
Normal		13.2%		33.7%	19.8%	12.5%	79.2%
Large	0.7%	4.4%	1.5%	0.6%	0.6%		7.8%
Xlarge	0.1%	4.2%		0.3%	0.3%		4.9%
Celkem	8.9%	21.8%	1.5%	34.6%	20.7%	12.5%	

Graf 3 Fragmentace velikostí a rozlišení obrazovek [8]

Tento problém pociťují hlavně vývojáři mobilních aplikací, protože je třeba při vývoji brát ohled na různé kombinace rozlišení a velikostí obrazovek při návrhu aplikace.

1.2 iOS

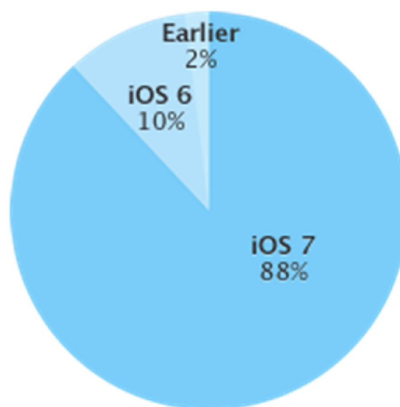
Druhým nejrozšířenějším mobilním operačním systémem je iOS od společnosti Apple. Tento operační systém je z trojice nejpoužívanějších OS nejstarší. Pro vývoj aplikací se používá speciální jazyk ObjectiveC. Systém je dostupný pouze na telefonech/tabletech společnosti Apple [1].

Aplikace jsou do zařízení distribuována skrze Apple AppStore. Na rozdíl od Android OS není možné do iOS nainstalovat aplikaci z jiného zdroje než z AppStore[1]. Pokud jako vývojář

[1] Pomocí neoficiální úpravy tzv. JailBreak je možné prolomit ochranu iOS systému a poté je možné instalovat aplikace z libovolných zdrojů.

chceme umístit svou aplikaci do AppStore, je třeba počítat s přísnějšími podmínkami pro publikaci než v rámci Androidu. Proces schvalování může trvat i týdny a samotnou aplikaci schvaluje reálná osoba na rozdíl od Android GooglePlay.

Z hlediska fragmentace zařízení a SDK je tato platforma velice přívětivá. Většina zařízení používá vždy nejnovější verzi operačního systému a aktuálně existuje pouze 5 různých rozlišení. Reálně stačí počítat pouze se 3 rozlišeními (2 rozlišení pro telefon, 1 pro tablety), protože novější verze zařízení používají většinou 2 násobné rozlišení [12].



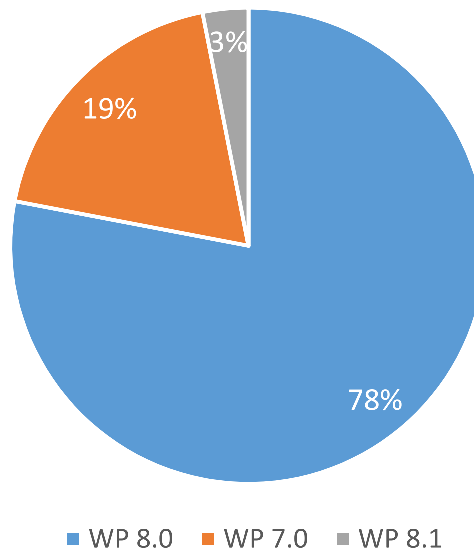
Graf 4 Fragmentace verzí iOS [11]

1.3 Windows Phone

Windows Phone je nejmladší mobilní OS a momentálně okupuje třetí příčku v rozšířenosti mobilních OS. Tento OS nahradil starší Windows Mobile. Pro vývoj aplikací je možné použít kombinace XAML/HTML5 pro frontend a C#/C++/JS pro backend [3][1].

Aplikace jsou do zařízení distribuovány skrze Windows Store. Stejně jako u iOS ani zde není možné instalovat aplikace jiným způsobem[2]. Pokud chceme umístit svou aplikaci do Windows Store, je třeba počítat se schvalovacím procesem, který provádí reálná osoba stejně tak, jako na platformě iOS.

[2] U zařízení podporující SD kartu je možné aplikaci stáhnout z Windows Store a nainstalovat z SD karty do zařízení.



Graf 5 Fragmentace Windows Phone OS [13]

Z hlediska fragmentace verzí OS je na tom platforma Windows Phone o něco hůře než iOS. Společnost Microsoft se rozhodla nasadit novou verzi OS (8.0) bez možnosti upgradu z předchozí verze. Avšak do budoucna je přislíbena snaha zachovat možnost upgradu z nižší verze na vyšší. Z grafu vyplývá, že většina zařízení disponuje nejnovější dostupnou[3] verzí tohoto operačního systému.

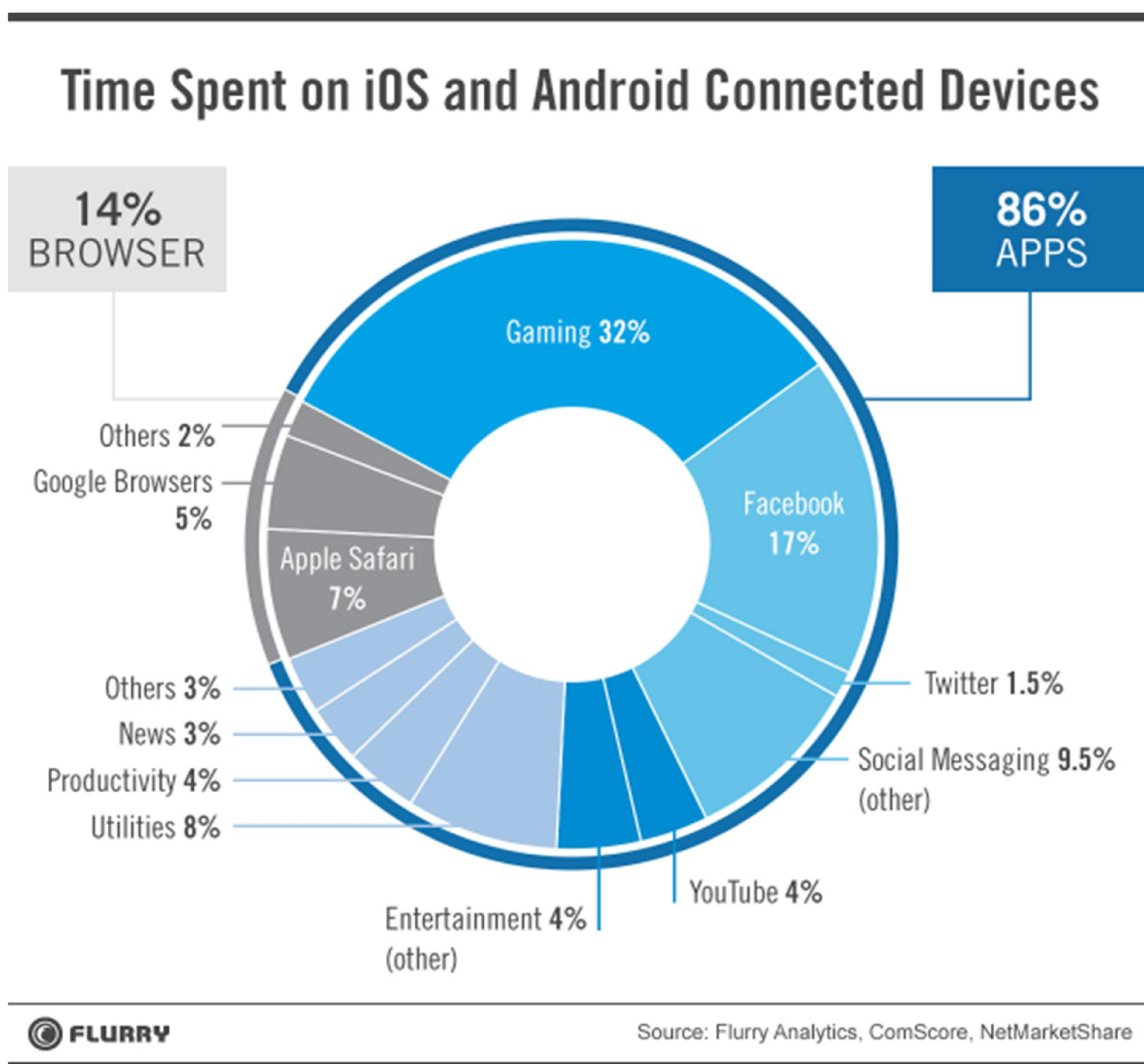
Z hlediska fragmentace rozlišení existují celkem 4 různá rozlišení ve 2 verzích poměrů stran.

[3] V době psaní této práce byla verze 8.1 dostupná pouze pro vývojáře.

2 DRUHY MULTIPLATFORMNÍHO VÝVOJE

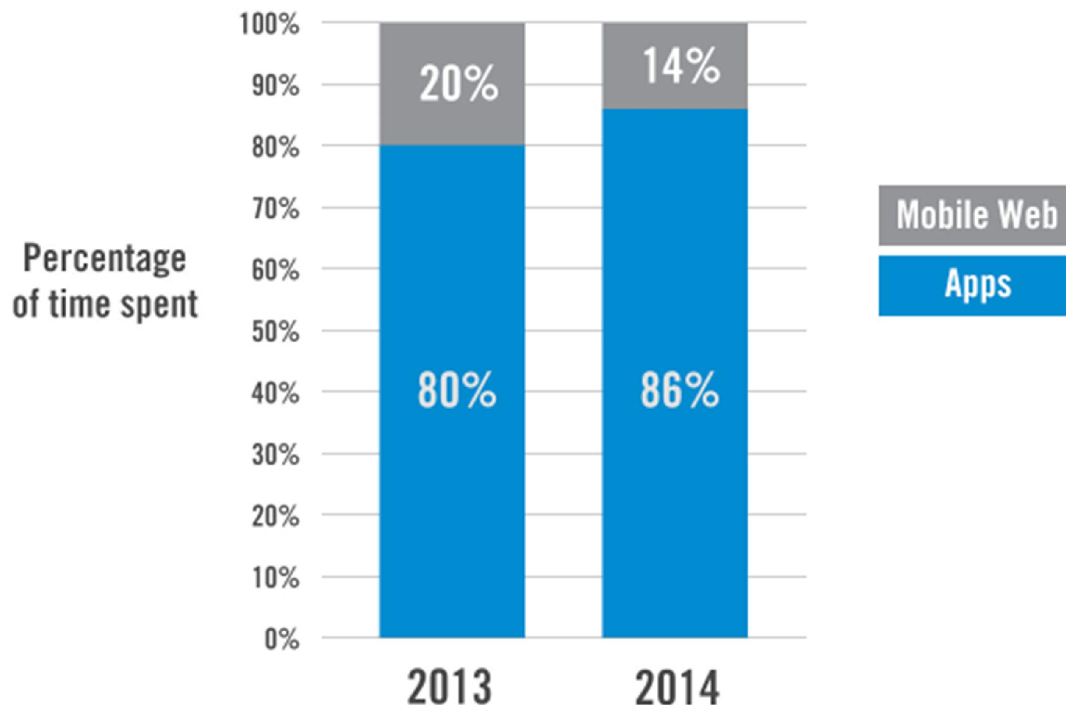
Pokud chceme vyvíjet aplikaci multiplatformě můžeme využít jeden z mnoha dostupných frameworků. Momentálně neexistuje žádný universální framework, který by byl vhodný pro všechny typy aplikací, proto je důležité stanovit si, co od dané aplikace požadujeme a očekáváme a podle toho poté zvolíme vhodný framework [3].

Při výběru frameworku můžeme využít data z následujícího grafu, který zobrazuje rozložení času, který stráví průměrný uživatel při práci s mobilními zařízeními.



Graf 6 Rozložení času stráveného používáním mobilních zařízení pro USA [14]

Z grafu je pro nás podstatné to, že uživatelé stráví 86% času používáním aplikací a 14% používáním webového prohlížeče. Vývoj tohoto trendu mapuje následující graf.



Graf 7 Vývoj trendu mobilní využívání mobilních webů a mobilních aplikací [14]

Z grafu můžeme vidět, že uživatelé mobilních zařízení preferují mobilní aplikace před mobilními weby. Z grafu také vyplývá, že se jedná o dlouhodobý a vzestupný trend[4].

V současnosti existuje na trhu mnoho různých frameworků pro multiplatformní vývoj aplikací. Bohužel není možné popsat všechny frameworky individuálně. Proto si zde popíšeme spíše přístupy jednotlivých frameworků k problematice tvorby multiplatformní aplikace.

[4] Vývoj trendu využívání mobilních webů a mobilních aplikací pro roky 2010 – 2012 je možné najít na http://biznessapps.com/blog/wp-content/uploads/2012/12/mobile-app-tv-consumption.png

2.1 Nativní tvorba

V rámci multiplatformního vývoje se můžeme setkat i s nativní tvorbou aplikace pro každou platformu zvlášť. Tento přístup vychází z toho, že každou platformu vyvíjíme v nativním jazyce a nativním prostředí. Bohužel jednotlivé platformy používají každá svůj jazyk a svoje IDE.

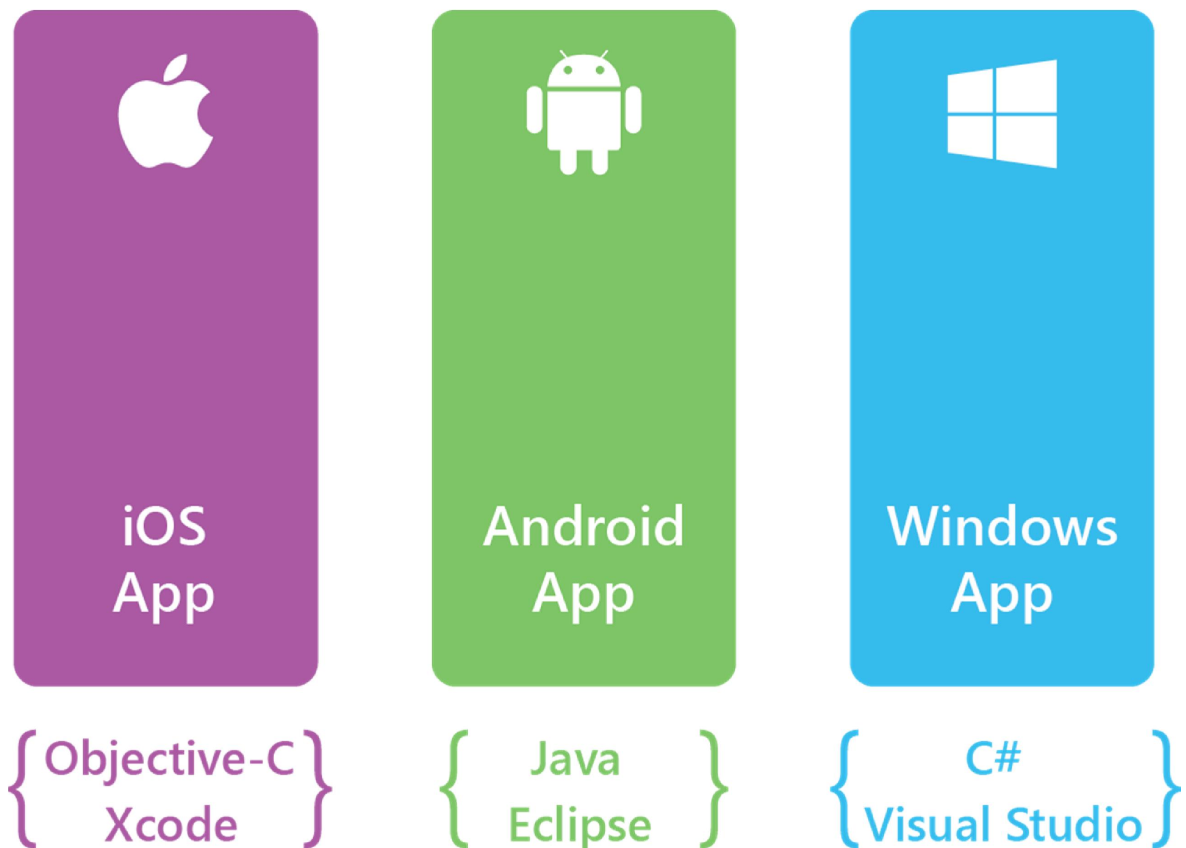


Schéma 1 Jazyky a IDE jednotlivých platforem [16]

Tento přístup si mohou dovolit jen velké společnosti, které vyžadují, aby uživatel aplikaci dostal maximální možné user experince, protože tento přístup pro vývoj aplikací je velice časově a finančně nákladný.

Další nevýhody jsou praktická nemožnost sdílení kódu napříč platformami. Dále také kvůli tomu, že každá platforma používá svůj vlastní jazyk, tento přístup je z hlediska složitosti návrhu a tvorby aplikace velice náročný [18].

2.2 Webové aplikace

Tento přístup vývoje spočívá v tom, že vytvoříme webovou stránku přizpůsobenou speciálně pro mobilní zařízení. Toto přizpůsobení většinou spočívá v tom, že layout stránky je navržen

jako tzv. responzivní design. Někdy také dochází k nahrazení standartních webových stylů pro prvky jako tlačítka a „roletky“ tak, aby vypadaly jako prvky na příslušné platformě. Pokud nedojde k nahrazení těchto webových stylů těmi platformními, většinou dochází alespoň k přizpůsobení těchto prvků pro dotykové ovládání [19], [20].

Obrázek 1 Rozdíl mezi standartní a přizpůsobenou webovou stránkou [15]

Tento přístup vývoje je velice vhodný pro exitující aplikace, pokud již máme plně funkční webovou stránku a v rámci naší aplikace nepotřebujeme využívat specifické prvky daného zařízení jako např. kamera, bluetooth, NFC.

Nevýhodou tohoto přístupu je, že aby uživatel mohl spustit danou aplikaci, musí mít přístup k internetu. Výpočetní výkon takovéto aplikace je velice nízký, protože většina výpočtů je realizována skrze JavaScript.

Typičtí zástupci tohoto přístupu jsou [20]:

- jQuery.Mobile
- Sencha Touch

2.3 „Write once, run anywhere“

Tento přístup by se v překladu dal interpretovat jako „napsat jednou, spouštět všude“. Prakticky se jedná o to, že se celá aplikace vyvine v jednom prostředí a v jednom jazyce, a poté je beze změn distribuována na jednotlivé platformy [17].

Tento přístup se pokouší implementovat mnoho různých frameworků v různých úrovních. Nejznámější a nejrozšířenější implementaci poskytuje framework PhoneGap. Tento framework vykresluje UI vrstvu pomocí nenativních komponent (abstraktní UI). Pomocí tohoto frameworku se vytváří standartní webová mobilní aplikace (HTML5 + JS), která může pomocí PhoneGapu využívat nativní prvky (fotoaparát, NFC, …) daného zařízení. Dále tento framework umožňuje publikovat aplikaci v app storech pro danou platformu jako „nativní“ aplikaci [19].

Takto vytvořená aplikace se po nainstalování tváří jako nativní aplikace. Po spuštění aplikace je otevřen upravený webový prohlížeč, který umožní běh takto vytvořené aplikace.

Jako další zástupce tohoto přístupu je možné označit Appcelerator Titanium, což je framework umožňující napsat jedenkrát UI vrstvu proti Titanium SDK, která je pak přeložena pro každou platformu do nativních UI prvků. Samotný kód je definován pomocí jazyka JavaScript a je překládán také do nativního jazyka platformy. Tento framework řeší některé problémy spojené s využitím frameworků, které vykreslují vrstvu UI nenativně [19].

Hlavní nevýhody tohoto přístupu je, že aplikace má většinou velice špatné user experience. Aplikace běží „pomalu“, nechová se nativně, jak by uživatelé dané platformy očekávali.

Další nevýhodou je tvz. problém nejmenšího společného prvku, který z principu neumožňuje 100% sdílení kódu napříč platformami.

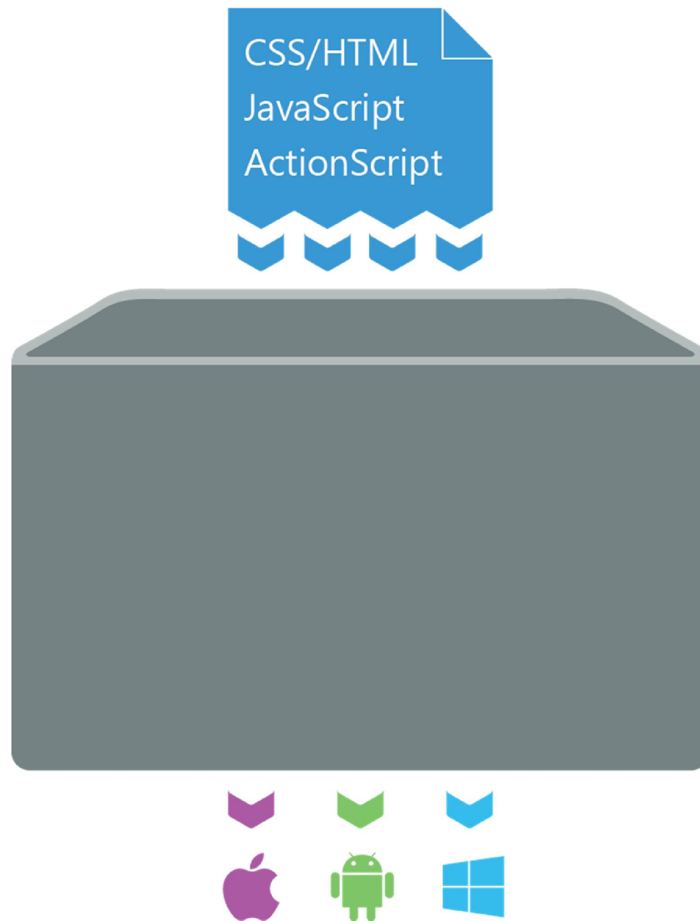


Schéma 2 „Write once, run anywhere“ nebo také „magic box“ [16]

2.4 Xamarin

Způsob multiplatformního vývoje pomocí frameworku Xamarin je natolik odlišný, že se nepodařilo zakomponovat do výše uvedených způsobů vývoje. Pokud bychom měli popsat tento princip jednoduše, tak by se jednalo o sdílené jádro + definice UI vrstvy pro každou platformu zvlášť. Zde je ovšem třeba uvést, že při definici UI vrstvy se Xamarin nepokouší poskytnout žádné společné UI rozhraní, ale poskytuje přesně ty prostředky, které nabízí daná platforma (pouze je poskytuje v jednom jazyku a to C#). Podrobnější rozbor platformy Xamarin bude proveden níže [16].

2.5 Shrnutí

Na základě předchozích odstavců, různých článků a vlastních zkušeností, bylo vytvořeno schéma, které stručně popisuje a charakterizuje předchozí popsané způsoby vývoje. V tomto schématu jsou sledovány hlavně tyto faktory:

- Komplexnost – jak složité je naučit se a vytvořit aplikaci pomocí daného přístupu

- Sdílení kódu – jaké množství stejného kódu je možné použít mezi jednotlivými platformami
- User experience – nejdůležitější vlastnost, která označuje kvalitu vytvořené aplikace
- Vhodnost – pro jaký typ projektů je nejlepší použít daný přístup

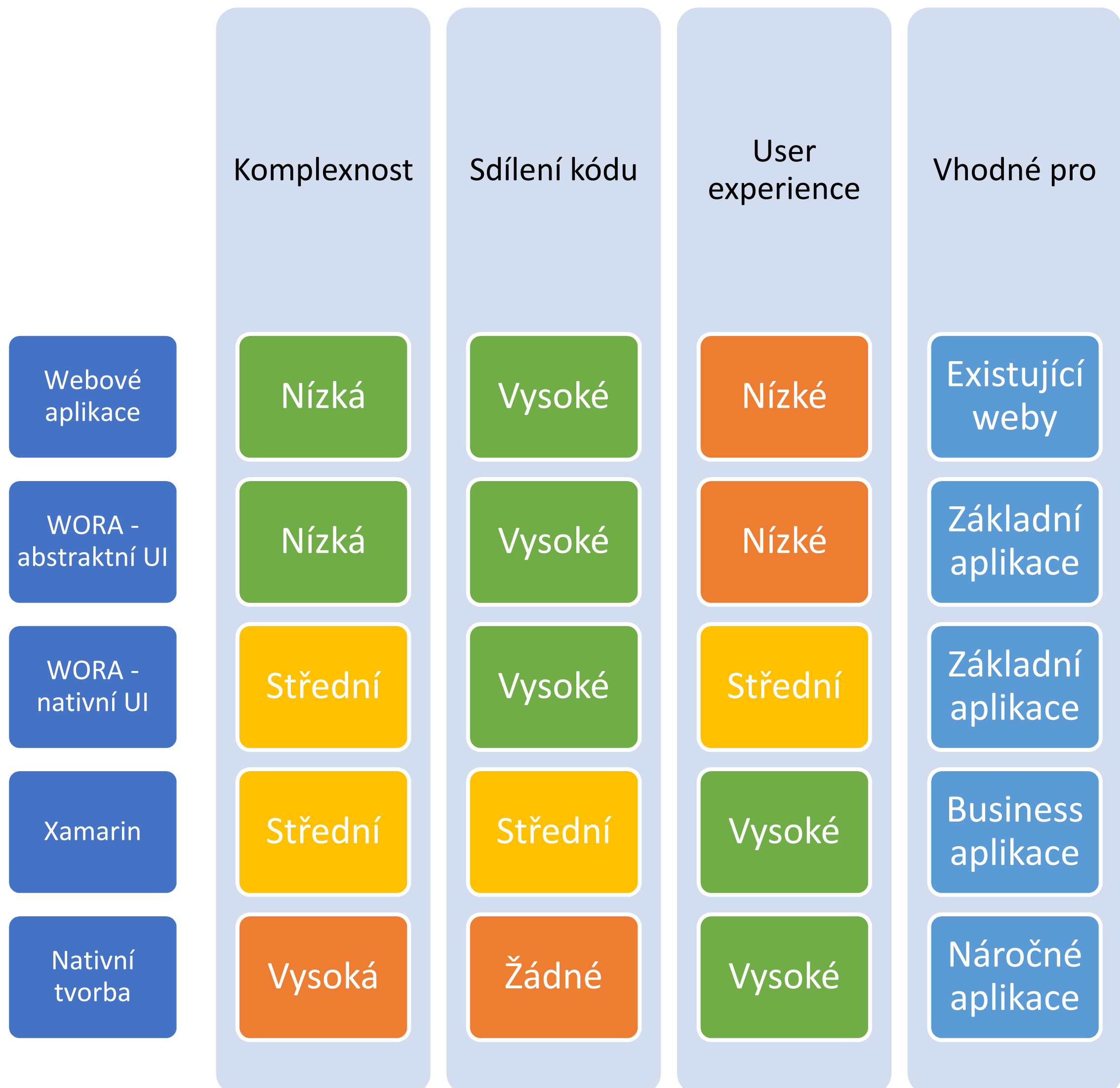


Schéma 3 Shrnutí druhů vývoje[5]

[5] Ve srovnání je rozlišen přístup „Write once, run anywhere“ (WORE) s použitím abstraktního UI (např. PhoneGap) a přístup WORE s použitím nativního UI (např. Appcelerator Titanium).

3 XAMARIN

Xamarin je společnost, kterou založil Miguel de Icaza v roce 2011 spolu s vývojáři, kteří jsou zodpovědní za vznik platforem Mono, MonoTouch a Mono for Android (MonoDroid). Společnost nabízí platformu Xamarin, která umožňuje multiplatformní vývoj aplikací pomocí technologie Mono [16].

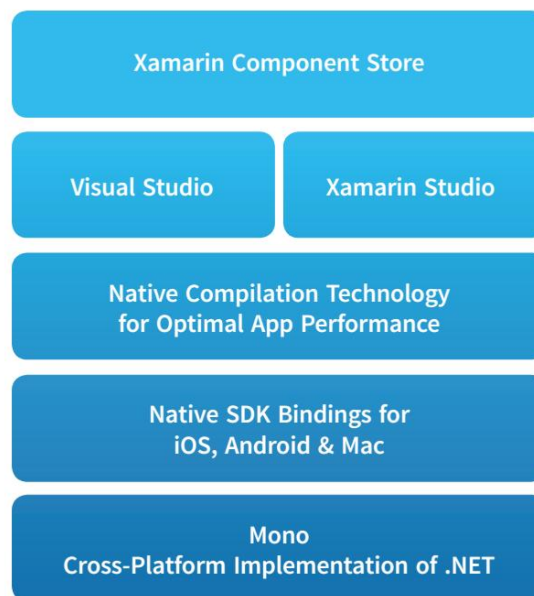


Schéma 4 Xamarin platforma [16]

Xamarin framework je komerční produkt, který existuje ve 4 variantách [21]:

- Free
  - Určeno pro jednotlivce
  - Umožňuje jak vytváření aplikací a jejich nasazení do reálných zařízení, tak jejich publikaci
  - Dodávána s Xamarin Studiem
  - Omezená velikost kódu (max. 32kb kódu)
- Indie (299$/rok)
  - Stejné viz. Free
  - Neomezená velikost kódu
- Business (999$/rok)
  - Stejné viz Indie
  - Určeno pro organizace
  - Integrace Visual Studia
  - Business funkce (WCF, SQLData…)
  - Email Support
- Enterprise (1899$/rok)
  - Stejné viz. Business
  - Kredit 500$ do Component Store

- Vylepšená podpora

Na trhu existuje spousta multiplatformních řešení. Xamarin se snaží jít cestou, kdy je společné jádro aplikace pro každou platformu stejné a definuje se pouze grafické rozhraní (případně platformě závislé prvky) pro každou platformu zvlášť.

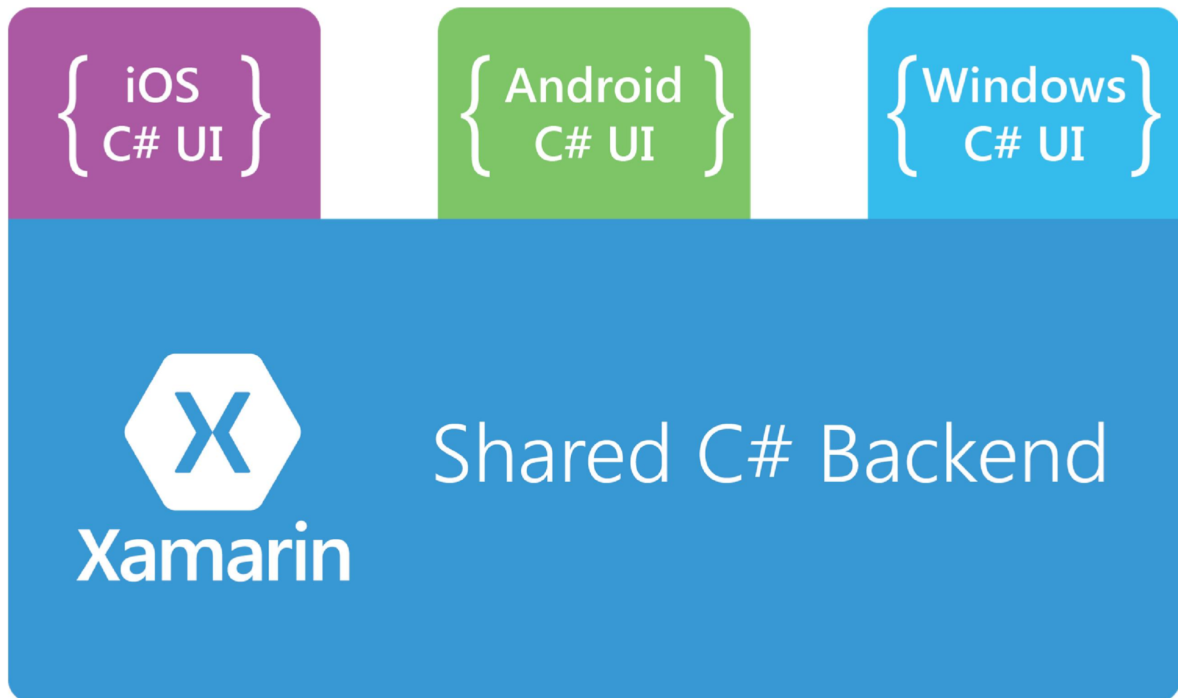


Schéma 5 Princip přístupu Xamarin platformy [16]

Platforma Xamarin je cílena spíše na tzv. business aplikace nebo na náročnější aplikace, kde je více kódu v pozadí, než v samotné prezentační vrstvě. Pokud potřebujeme vytvořit aplikaci, která obsahuje prakticky pouze prezentační vrstvu a minimum logiky a požadavek na prezenční vrstvu je, aby na všech platformách vypadala absolutně stejně, je třeba zvážit, zda nepoužít jiný nástroj, než Xamarin[6].

V opačném případě, kdy velká část aplikace spočívá na řešení aplikační logiky a při správně navržené architektuře aplikace je možné dosáhnout velmi dobrých výsledků[7].

Na stránkách Xamarinu je k dispozici kompletní dokumentace k platformě, krátké příklady pro implementace platformních funkcí v Xamarin prostředí a také vzorové aplikace.

[6] Z hlediska časové náročnosti a složitosti implementace se pro prezentační projekty vyplatí použít frameworky založené na přístupu „Write once, run anywhere“.

[7] V rámci případové studie aplikace iCircuit se dosahovalo sdílení kódu přes 80%.

3.1 Mono

Mono je bezplatný a otevřený projekt (aktuálně vedený firmou Xamarin), který si klade za cíl vytvoření otevřeného C# komplilátor a CLI dle ECMA standartu. Hlavní náplní projektu mono je aby bylo možné spouštět aplikace psané pomocí .NET Frameworku na jiných platformách než Windows [22].

Aktuálně se podařilo implementovat mono pro tyto platformy:

- Windows
- Mac
- iOS
- Android
- Linux
- Spousta dalších jako Raspberry PI, PS3, Wii …

V následujícím schématu je popsána kompatibilita mezi mono a standardním .NET. Červeně jsou označeny doposud neimplementované části, oranžově jsou částečně implementované části.

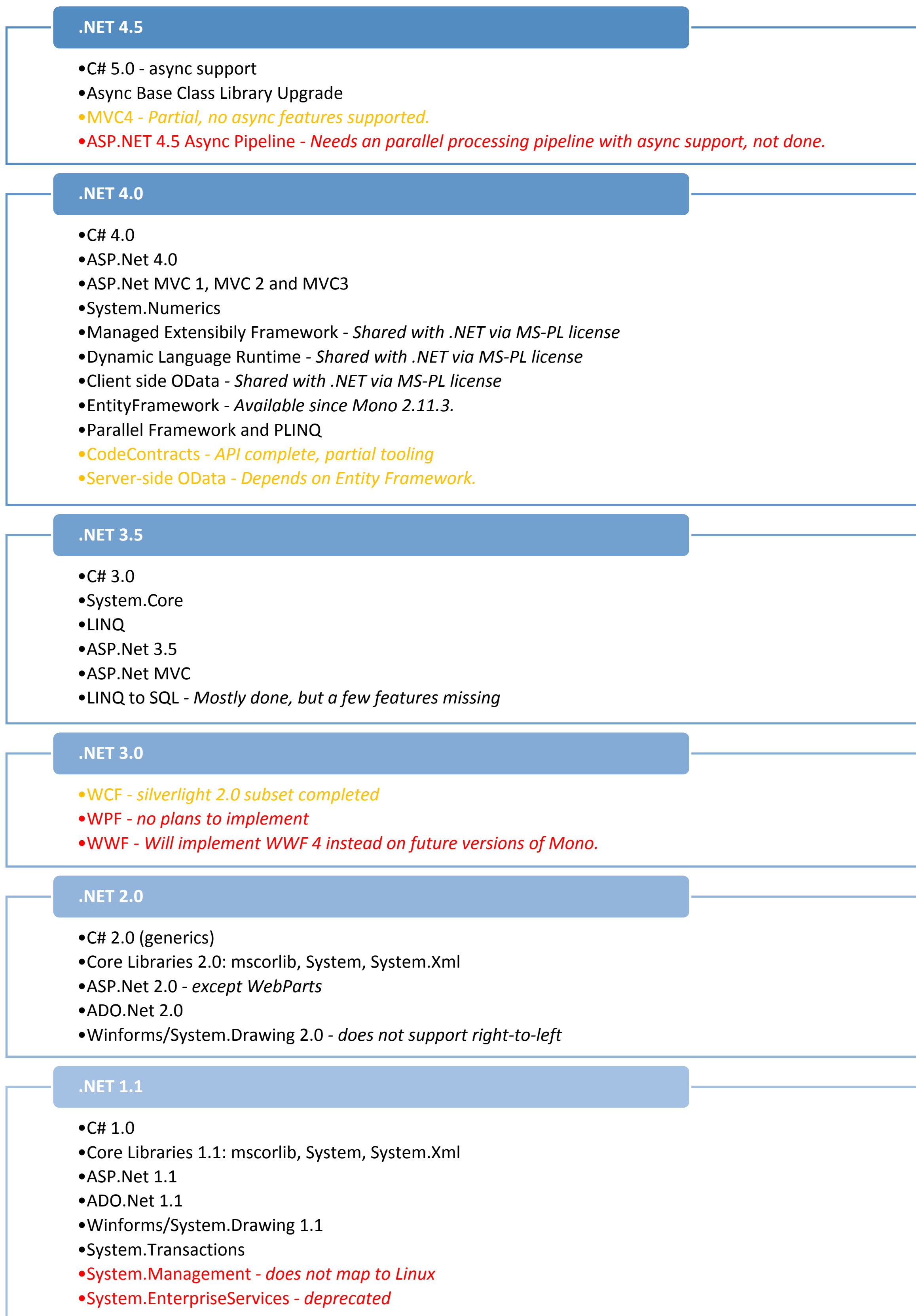


Schéma 6 Kompatibilita mono a .NET [23]

3.2 Xamarin.Android

Xamarin.Android (XA) je framework pro vývoj aplikací pro platformu Android. XA potřebuje pro svoji funkčnost android SDK a příslušné android SDK API. XA si můžeme představit jako .NET wraper nad nativními android knihovnami s určitými úpravami charakteristickými pro jazyk C# (převedení listenerů na eventy, zapouzdření pomocí properties …). Kromě standartních android knihoven poskytuje XA také .NET Base Class Library + další vybrané .NET knihovny. Také je zde možnost použít již existující android knihovny[8].

XA využívá principu JIT (Just In Time) kompilace. To znamená, že C# kód je zkompilován do assemblies a ty jsou uloženy do výstupního APK souboru. V momentě spuštění aplikace dojde k inicializaci mono virtuálního stroje, který bude vykonávat instrukce obsažené v těchto assemblies. Na stejném principu fungují standartní android aplikace s tím rozdílem, že Java je interpretována pomocí Java virtuálního stroje (DALVIK) [21].

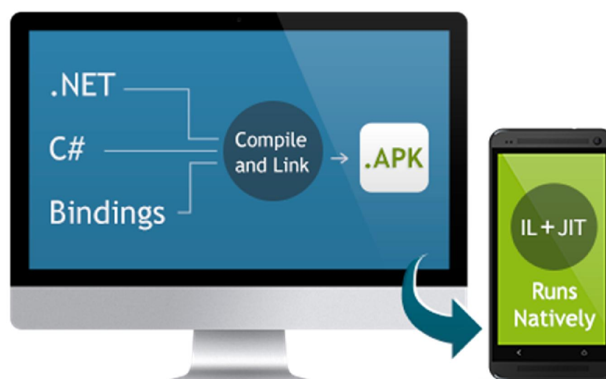


Schéma 7 Princip kompilace Xamarin.Android [16]

3.3 Xamarin.iOS

Xamarin.iOS (XI) je framework který nám umožňuje vytvářet aplikace pro operační systém iOS. XI potřebuje pro svou funkčnost mít dostupný počítač s operačním systémem MacOSX s nainstalovanými developerskými nástroji jako xCode a iOS SDK. Toto omezení znamená, že buďto vyvíjíme v Xamarin Studiu na počítači Apple nebo vyvíjíme ve Visual Studiu,

---

[8] Pomocí Xamarin Java Bindings Library je možné jednoduše převést stávající android knihovnu pro konzumaci v Xamarin.Android.

které je spojeno přes síť s počítačem Apple, na kterém je nainstalován vzdálený Build Host Agent.

V XI stejně jako v XA jsou k dispozici všechny funkce a třídy jako v normálním iOS + další třídy z .NET. Avšak existuje zde umělé omezení, které zakazuje vytvářet dynamický kód za chodu programu, takže na XI na rozdíl od XA není dostupná dynamická generace kódu (hlavně namespace Systém.Emit). S tím souvisí i jiný druh kompilace. Na rozdíl od XA je zde využit princip AOT (Ahead of Time Compilation), což znamená, že veškerý C# kód je přímo převeden do nativního ARM kódu [21].

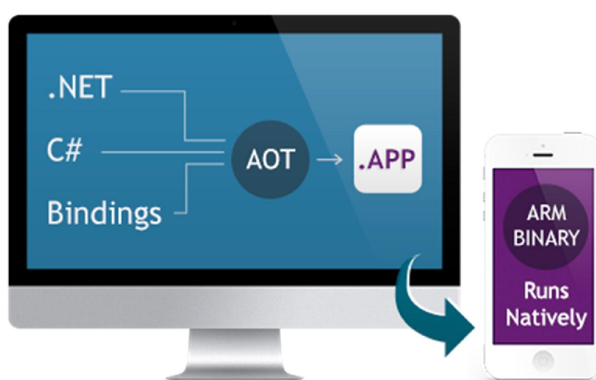
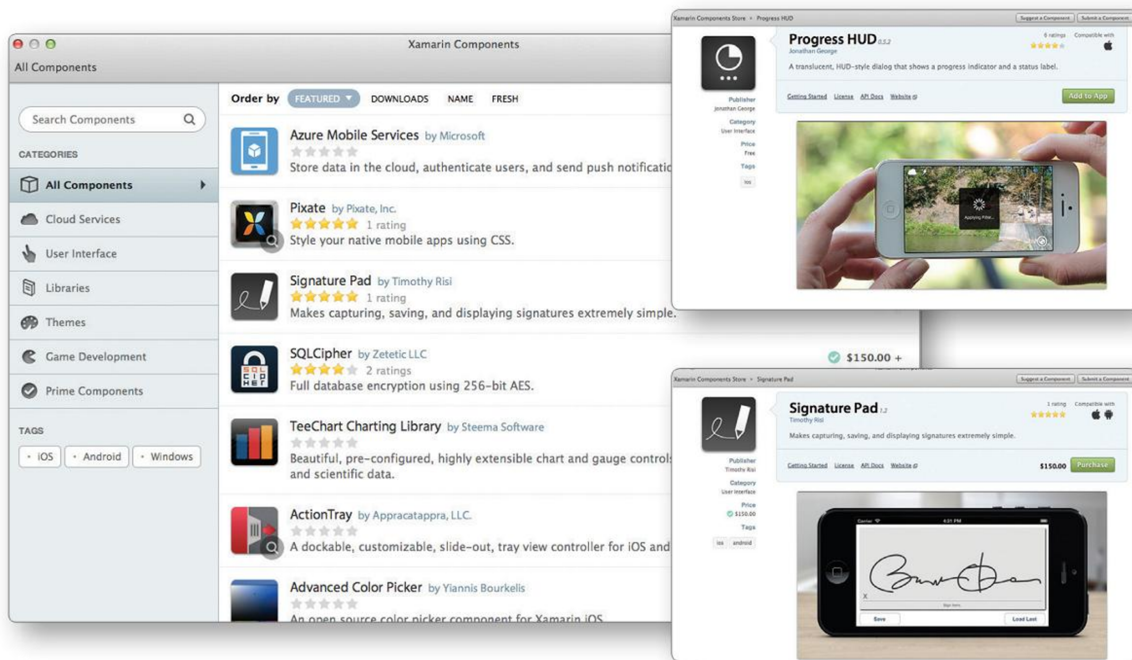


Schéma 8 Princip kompilace Xamarin.iOS [16]

3.4 Component store

Component store je katalog obsahující jak placené tak neplacené UI komponenty a knihovny. Tyto komponenty jsou většinou dostupné na více platforem. Katalog obsahuje jak komponenty společnosti Xamarin, tak také komponenty dalších společností nebo jednotlivců.



Obrázek 2 Xamarin Component Store [16]

Katalog obsahuje seznam kategorií a v něm je možné hledat dle klíčových slov nebo filtrovat dle platforem. Samotná stránka věnovaná komponentě obsahuje stručný popis, delší popis, ukázku implementace, odkaz na stránky výrobce komponenty, hodnocení komponenty a diskuzi ke komponentě [21].

Přístup do komponent store je jak z prostředí Xamarin Studia, tak z Visual Studia. Pokud najdeme vhodnou komponentu, stačí stisknout tlačítko „Add to App“ a komponenta se přidá na váš Xamarin účet, poté se stáhne do počítače a přidá jako reference do projektu.

Kromě nově vytvořených komponent speciálně pro tento katalog, obsahuje katalog i porty populárních knihoven z nativních jazyků do Xamarin platformy.

3.5 Strategie sdílení kódu

Jedním s klíčových vlastností Xamarin frameworku je možnost sdílet kód popisující aplikační logiku mezi jednotlivými platformami. Je možné využít několik strategií sdílení kódu. Mezi základní strategie patří [24]:

Linkování/Klonování

Mezi nejstarší techniky sdílení kódu na platformě Xamarin patří linkování souborů.

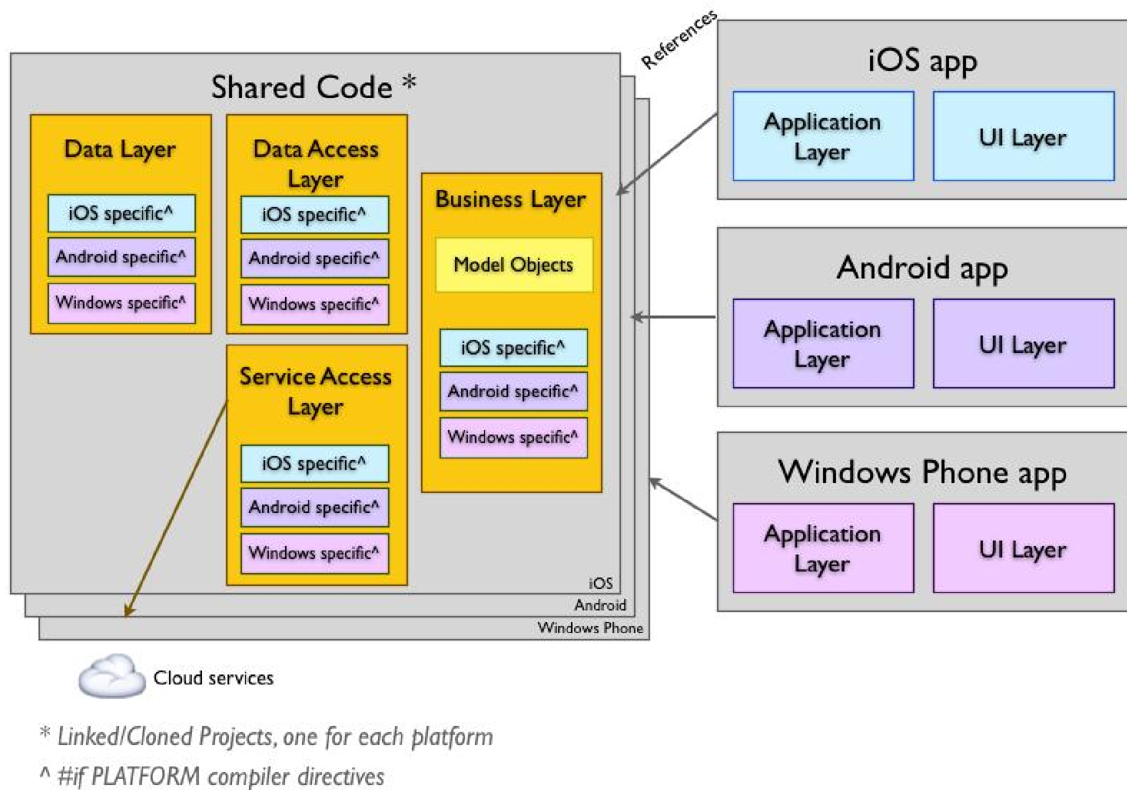


Schéma 9 Linkování/klonování projektů v Xamarinu [24]

Princip této techniky spočívá v tom, že si založíme libovolný projekt a v něm si implementujeme jednotlivé vrstvy projektu. Ty pak pomocí linkování dostaneme do platformních (GUI) projektů. Pro platformě závislé prvky je zde využit podmíněný překlad. Tato technika se objevuje ve více provedeních.

Častější je provedení této techniky za pomocí klonování projektů. Např. pokud máme projekt typu knihovna pro Windows Phone 8, můžeme pomocí nástroje Project Linker vytvořit klony toho projektu pro dané platformy. V našem případě by se jednalo o projekty typu knihovna pro iOS a knihovna pro Android. Tyto knihovny se poté referencují z platformních (GUI) projektů.

Portable Class Library

Momentálně nejpoužívanější varianta sdílení kódu mezi platformami. V této variantě se využívá Xamarin rozšíření standartní Portable Class Library (PCL). Standartní PCL je typ knihovny, který umožňuje vytvářet kód za použití omezeného .NET frameworku pro vybrané platformy. Standartně jsou k dispozici platformy .NET, Windows Phone , Windows Store, Silverlight a Xbox360. Princip PCL je takový, že při vytváření PCL si zvolíme, pro jaké platformy potřebujeme kompatibilitu a podle toho se nám vybere vhodná podmnožina .NET

frameworku, kterou mají implementované všechny vybrané platformy. V praxi to znamená, že čím více platforem zvolíme, tím menší .NET Framework API máme k dispozici [24].

Po nainstalování Xamarin Frameworku nám přibydou do PCL platformy Xamarin.iOS a Xamarin.Android.

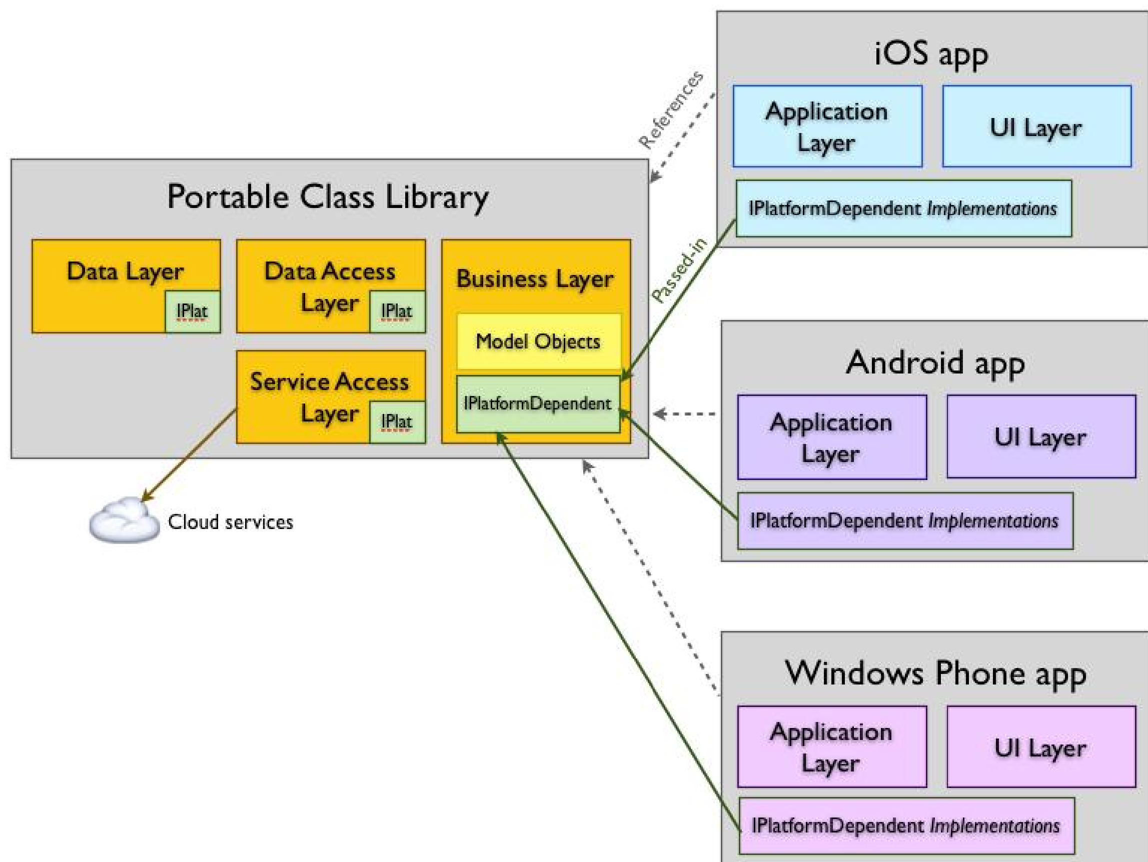


Schéma 10 Sdílení kódu pomocí PCL v Xamarinu [24]

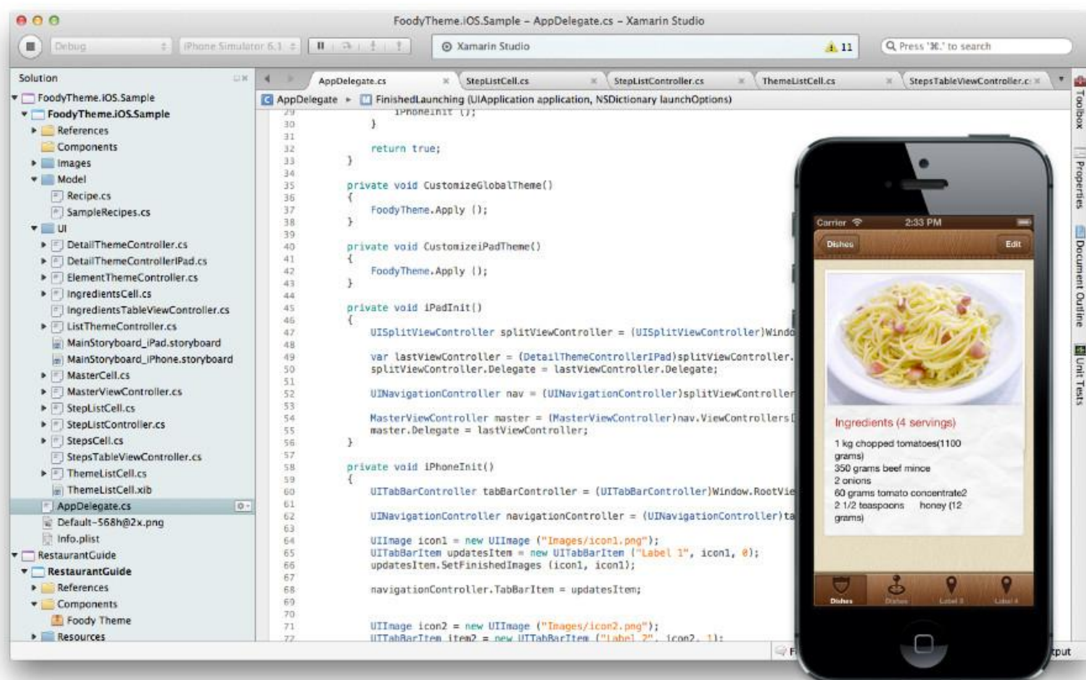
Při použití PCL pro sdílení kódu nám odpadají problémy s podmíněným překladem. Platformě závislá funkcionalita je zde dodána pomocí Dependency Injection[9]. Nevýhodou PCL je, že neobsahuje plné API „velkého" .NET Frameworku. Výhodou PCL je však to, že můžeme využít balíčkovací systém Nuget pro nainstalování knihoven třetích stran pro dodání funkcionality. Momentálně obsahuje systém Nuget spoustu různých knihoven kompatibilních s mono a PCL.

[9] Dependency Injection návrhový vzor založený na rozhraních a jejich implementacích. Podrobněji je popsán v dalších kapitolách.

3.6 Vývoj pod Xamarinem

Xamarin Studio

Pro vývoj na frameworku Xamarin vytvořila společnost Xamarin své vlastní multiplatformní IDE s názvem Xamarin Studio.



Obrázek 3 Xamarin Studio [21]

Toto IDE je dostupné na platformách Windows a MacOS. Na linuxu toto IDE není implementováno, je však možnost využít původní projekt MonoDevelop, který ovšem nenabízí Xamarin knihovny, ale pouze standartní mono knihovny.

Xamarin studio je moderní IDE se spoustou funkcí jako [16]:

- Našeptávač, refaktorizace, hledání
- Designer Adnroid GUI
- Designer iOS (momentálně využívá Xcode Interface Builder, avšak v beta fázi se nachází samostatný designer)
- Integrace s Xamarin Component store
- Podpora vývoje na mobilních zařízeních (debugování/nasazení na simulátorech/zařízeních)
- Podpora pro verzování (Git, SVN, TFS)
- Podpora pluginů
- Podpora Nuget balíčkovacího systému (skrze plugin)
- Využívá stejný systém projektů jako Visual Studio (.sln files)

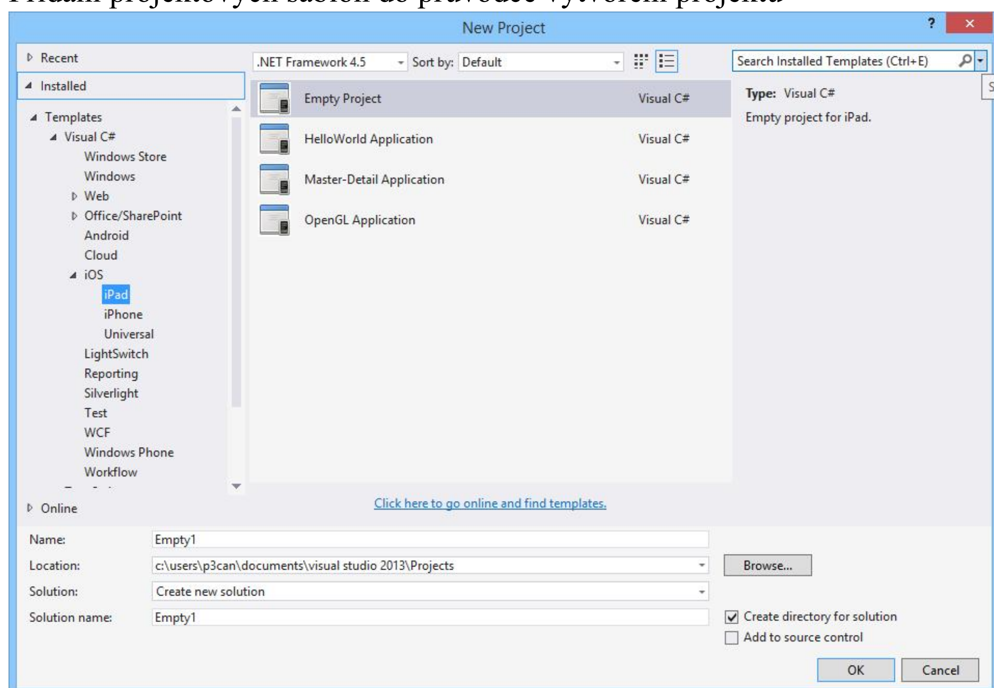
Mezi nedostatky Xamarin studia patří:

- Neumožňuje vývoj pro Windows Phone
- Na platformě Windows neumožňuje vývoj pro iOS

Visual Studio

Společnost Xamarin taktéž umožňuje využívat pro vývoj na její platformě Visual Studio 2010 - 2013. Potřebná funkcionalita přidána následovně [16]:

- Přidání projektových šablon do průvodce vytvoření projektu

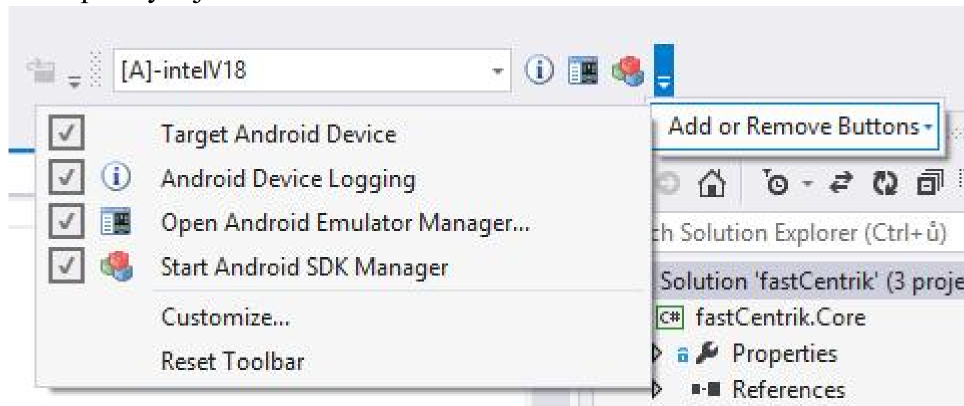


Obrázek 4 Xamarin šablony pro Visual Studio

Po nainstalování frameworku Xamarin dojde k přidání šablon projektů pro Android i iOS. iOS navíc obsahuje podkategorii šablon pro iPhone, iPad nebo univerzální typ aplikací. Hlavní šablony pro jednotlivé platformy jsou:

  - šablony pro aplikaci
  - šablona pro knihovnu
  - šablona pro binding projekt

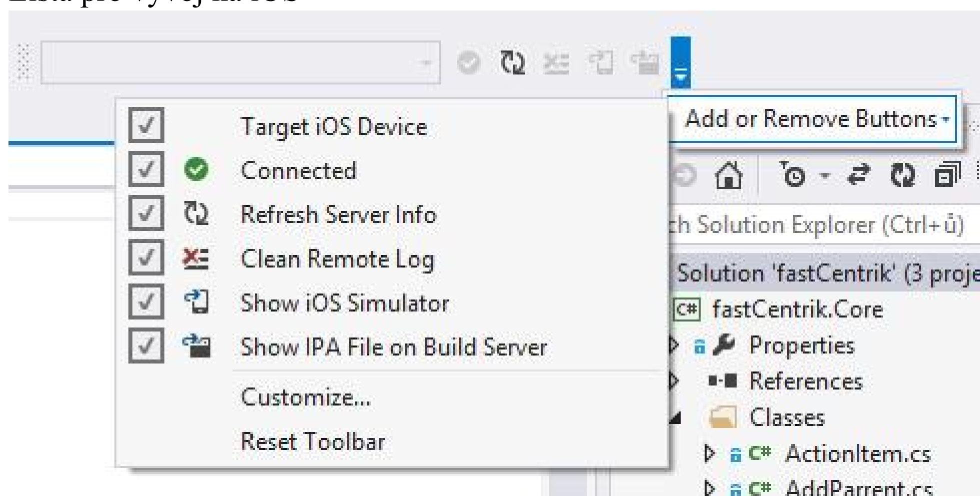
- Lišta pro vývoj na Androidu



Obrázek 5 Android toolbar

Pro zjednodušení vývoje pro Android platformu je do Visual Studia přidán Android Toobar. Pomocí tohoto toolbaru můžeme zvolit, na jakém zařízení/emulátoru chceme debutovat. Dále obsahuje odkazy na Android device logging (obsahuje logy ze zařízení), Emulator manager (obsahuje konfigurace emulátorů) a SDK manager (nástroj pro instalaci Android SDK, Android build tools a jednotlivých Android API)

- Lišta pro vývoj na iOS

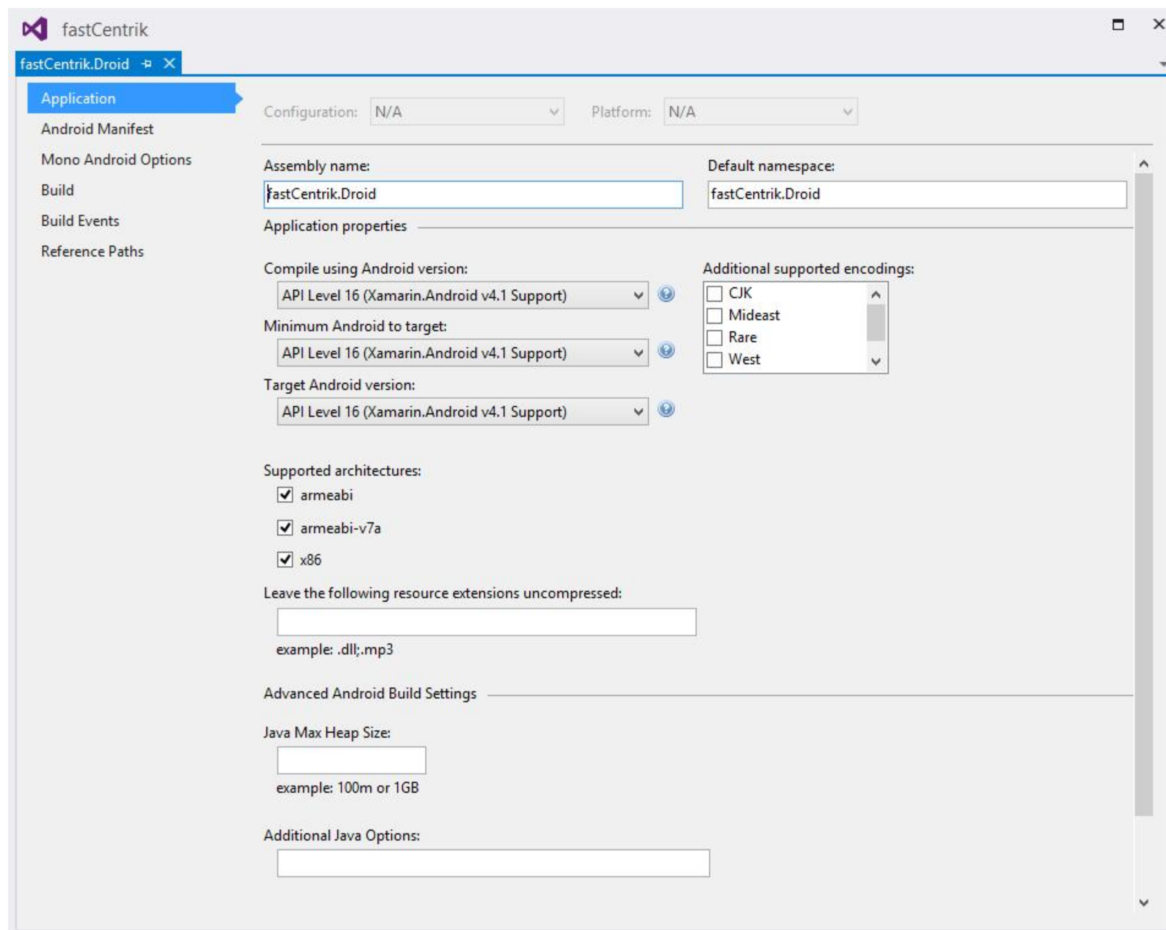


Obrázek 6 iOS toolbar

Pro vývoj pro iOS platformu pomocí Visual Studia byl přidán iOS toolbar. Tento toobar je aktivní pouze tehdy, spojí-li se přes síť se vzdáleným Xamarin build hostem na počítači MAC. Pomocí této lišty můžeme přepínat mezi jednotlivými emulátory nebo zařízeními (pro výběr, zda chceme testovat na zařízení nebo emulátoru byly přidány iPhoneDevice a iPhoneSimulator platformy do Configuration Manageru). Lišta dále obsahuje indikaci spojení mezi Visual Studiem a vzdáleným hostem, obnovení stavu spojení, vyčištění vzdáleného

logu, zobrazení emulátoru na vzdáleném MAC počítači a také zobrazení zkompilovaného souboru aplikace v souborovém systému na MAC počítači.

- Android nastavení projektu

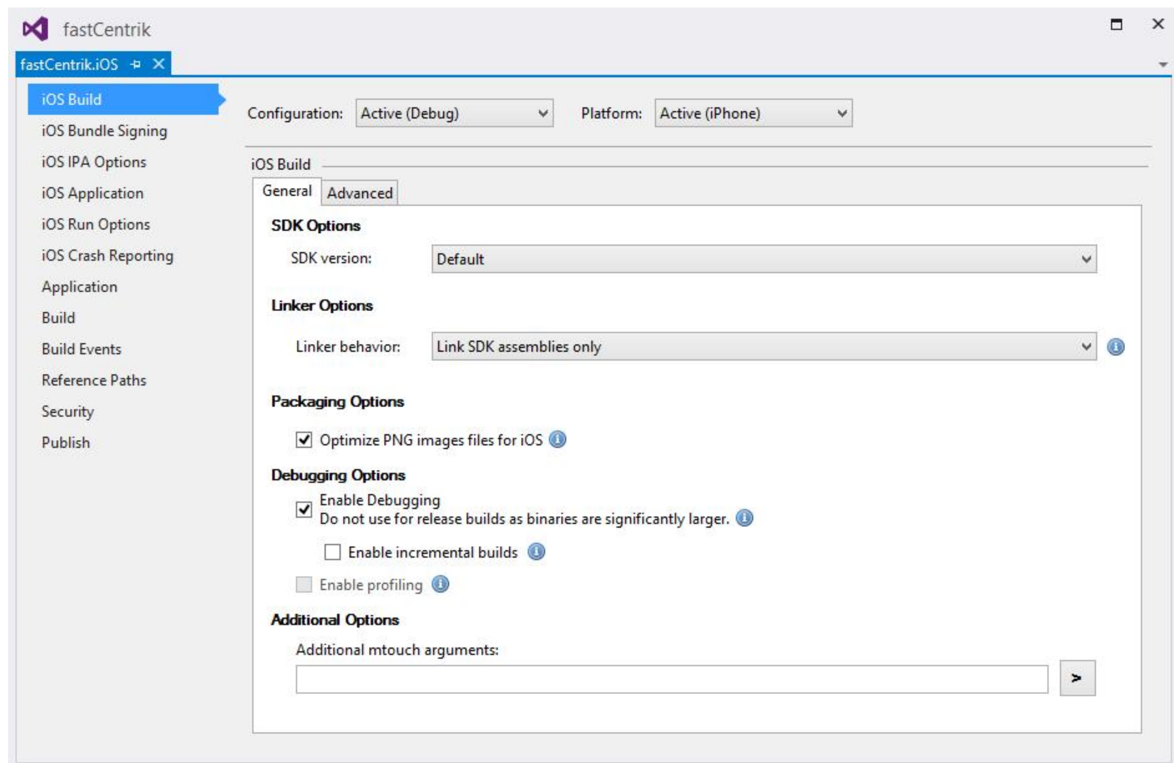


Obrázek 7 Vlastnosti projektu u Xamarin.Android aplikace

Dále je do prostředí Visual Studia přidáno několik záložek do vlastností projektu. Pro Android platformu jsou to 3 záložky (Application, Android manifest, Mono Android Options). Tyto záložky obsahují různá nastavení jako:

- možnost zvolit si cílové Android API
- podporované architektury procesorů
- grafický editor manifestu
- základní informace o aplikaci
- pokročilá debug nastavení

- iOS nastavení projektu



Obrázek 8 Vlastnosti projektu u Xamarin.iOS aplikace

Jak pro Android platformu tak také pro iOS platformu jsou do vlastností projektu přidány některé záložky. Pro iOS je to celkem 6 záložek (iOS Build, iOS Bundle Signing, iOS IPA Options, iOS Application, iOS Run Options, iOS Crash Reporting). Na těchto záložkách nalezneme různá nastavení jako:

- výběr SDK verze aplikace
- podporované architektury procesorů
- pokročilé možnosti buildu
- grafický editor plistu
- základní informace o aplikaci
- pokročilá debug nastavení
- další nastavení

V rámci Visual Studia je možný i vývoj na platformu iOS. Ten je realizován skrze vzdáleného build hosta, který je spuštěn na libovolném Mac počítači s nainstalovaným Xamarinem.

V praxi to funguje tak, že projekt vyvíjíme ve Visual Studiu. V momentě, kdy klikneme na tlačítko debug, na Windowsu je provedena kompilace zdrojových souborů do assemblies. Ty jsou zkopírovány přes síť do vzdáleného build hosta, který provede kompilaci AOT a pošle výslednou aplikaci do emulátoru/zařízení. Dokonce je možné provádět krokování a dynamické vyhodnocování proměnných v debug režimu.

Pro vývoj na android platformou se doporučuje místo normálních ARM images používat X86 images spolu s technologií HAXM[10]. Tímto je mnohonásobně zrychleno testování na emulátoru.

[10] Intel Hardware Accelerated Exucution Manager. Intelem vytvořený virtualizační engine využívající technologie Intel VT pro akceleraci Android systému na hostovaném zařízení.

4 NÁVRHOVÉ A ARCHITEKTONICKÉ VZORY

V rámci vývoje softwaru se můžeme setkat s pojmem návrhový vzor (anglicky design pattern). Jedná se o jakýsi předpis nebo šablonu, jak řešit určité druhy problémů unifikovaným způsobem (tak, aby tomu každý programátor pak rozuměl).

V současné době existuje velké množství návrhových vzorů [2][5]. Jedny z vybraných kategorií mohou být:

- Vytvářející vzory – zabývají se vytvářením objektů
- Strukturální vzory – zabývají se vztahy jednoduchých objektů mezi sebou
- Vzory popisující chování – zabývají se chováním systému
- Více-vlákenné (concurrency patterns) vzory – zabývají se problémy v rámci více-vlákenných systémů

Kromě návrhových vzorů se můžeme setkat i s architektonickými vzory. Ty popisují, jak navrhnout systém jako celek.

V následujících podkapitolách jsou popsané vybrané vhodné vzory a techniky pro vývoj multiplatformních aplikací.

4.1 Fluent API

Flupent API (oficiálně Fluent Interface) je způsob implementaci API tak, aby výsledný kód byl více čitelný. Většinou se pro implementaci využívá zřetězení metod. Samotná implementace se řídí jednoduchými pravidly [5]:

- Kontext je definován skrze návratovou hodnotu volané metody
- Nový kontext je ekvivalentní k původnímu kontextu
- Pro metody nebo vlastnosti nastavující *bool* nebo *enum* datové typy jsou vytvořeny metody pro každou hodnotu zvlášť (*SetWindowVisibility=true* -> *WithWindow*(); *SetWindowVisibility=false* -> *WithoutWindow*())

Pro ilustraci byla vytvořena ukázka normálního kódu.

```
public void NonFluentApi()
        {
            var rectangle = new Rectangle();
            rectangle.SetWidth(20);
            rectangle.SetHeight(40);
            rectangle.ShowBorder = true;
            rectangle.Children = new List<object>();
            rectangle.Children.Add(new Circle(3, 4, 5));
            rectangle.Show();
        }
```

Kód 1 Standartní API

Kód výše byl přepsán do stylu fluent API.

```
public void FluentApi()
        {
            Rectangle
                .Create()
                .WithWidth(20)
                .WithHeight(40)
                .WithBorder()
                .Add(new Circle(3, 4, 5))
                .Show();
        }
```

Kód 2 Fluent API

Použitím fluent API došlo ke zvýšení čitelnosti kódu. Velice často dochází při použití fluent API také ke zkrácení kódu. Toto zkrácení se projeví hlavně u tříd implementující návrhový vzor Immutable, kde při použití standartního API je třeba všechny mezivýsledky ukládat do proměnných.

Mezi hlavní nevýhody fluent API lze zařadit ztížené debuggování a zachytávání vyjímek.

4.2 Architektury uživatelského rozhraní

Martin Fowler definuje pojem architektury grafického uživatelského rozhraní a uvádí následující architektury [32]:

- Forms and Controls,
- Model View Controler,
- Model – View – Presenter (MVP),
- Presentation Model,
- VisualWorks Application Model,
- Humble View.

Martin Fowler později nahradil Model-View-Presenter dvěma architekturami a to Supervising Controller a Passive View [32].

Podrobný popis architekur uživatelských rozhraní je možné najít také zde [33]. Autor především zdůrazňuje rozdíl mezi klasickým vzorem Model – View – Controller, který vznikl v roce 1978 - 1979 a vzorem Model – View – Controller pro webové aplikace, který vznikl na konci devadesátých let a bývá někdy spojován s "Model 2" architekturou [33].

Patforma .NET obsahuje implementace těchto architektur [34]:

- Model-View-Controller (MVC), který se používá pro návrh webových aplikací, především s využitím webového aplikačního frameworku ASP.NET MVC.
- Model-View-ViewModel (MVVM), specializace Presentation modelu pro .NET, který je navržený pro WPF aplikace a využívá jazyku XAML pro tvorbu uživatelského rozhraní. Využívá bindování a commandů. ViewModel nemá přímou reference na View.
- Model-View-Presenter (MVP), které je určené také pro WPF aplikace, ale uživatelské rozhraní vyvolává události (eventy), na které reaguje presenter (kód na pozadí). Presenter má tedy přímou referenci na View.

Nedávno vznikla implementace MVVM pro javascript a HTML5 v podobě knihovny Knockout [35].

4.3 Dependency injection

Návrhový vzor DI (Dependency Injection) je založen na tom, že z kódu odstraňuje pevně vytvořené závislosti a vytváří možnost tyto závislosti měnit při kompilaci nebo za chodu programu. Tato technika je velmi důležitá, protože odstraněním pevně daných vazeb na závislostech nám velmi usnadňuje testování systému. Při vývoji rovněž můžeme reálné závislosti nahradit testovacími a ověřit tak funkčnost systému jako celku [25].

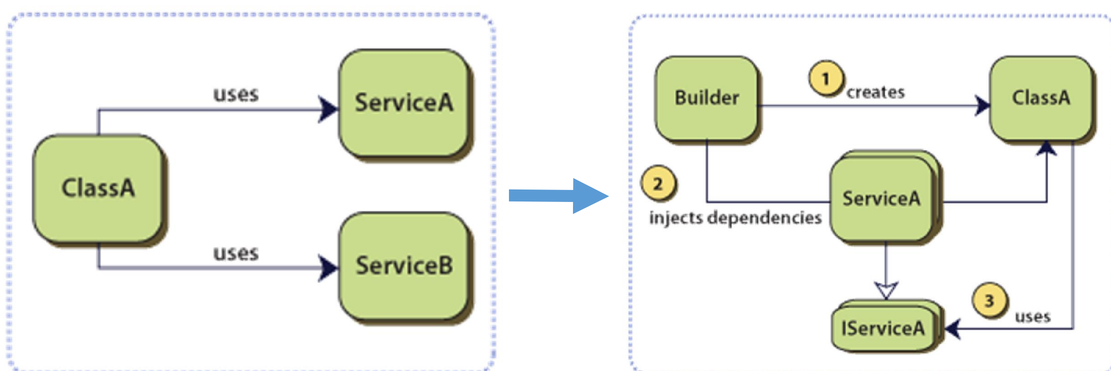


Schéma 11 Princip Dependency injection [25]

Dependency Injection se nejčastěji vyskytuje ve dvou provedeních:

- Constructor injection – závislosti se předávají v konstruktoru dané třídy. Závislosti třídy jsou jasně dané a třída nemůže vzniknout bez daným závislostí.

- Property injection – závislosti se předávají prostřednictvím vlastností dané třídy. Třída je vytvořena bez závislostí a ty jsou dodány až později prostřednictvím setterů.

Zde je ukázková třída před zavedením DI.

```
public class CustomerNotifier
    {
        private List<Customer> Customers { get; set; }
        public void LoadCustomer(string search_pattern)
        {
            using (var connection = new DatabaseContext())
            {
                var result = connection.Search(search_pattern);
                Customers=result;
            }
        }
        public void NotifyDiscount()
        {
            var smssender = new SmsSender();
            foreach (var x in Customers)
            {
                smssender.SendSms(x, "sleva");
            }
        }
    }
```

Kód 3 Vzorová třída před DI

Třída obsahuje 2 metody. Jedna slouží pro načtení zákazníků z databáze a druhá má za úkol rozeslat informační SMS pro vybrané zákazníky. Zde můžeme jasně vidět, že třída obsahuje 2 hlavní závislosti a to závislost na zdroji zákazníků (databáze) a závislost na třídě implementující rozesílání SMS. Tyto dvě napevno zakomponované závislosti zhoršují testovatelnost kódu a také není na první pohled vidět, co vše potřebuje třída pro svoji funkčnost.

DI pomocí constructor injection

Vzorovou třídu upravíme podle constructor injection.

```
public class CustomerNotifierWithConstructorDI
    {
        private IDataRepository DataRepository;
        private ISMSProvider SmsProvider;
        public CustomerNotifierWithConstructorDI(IDataRepository data, ISMSProvi-
der sms)
        {
            DataRepository = data;
            SmsProvider = sms;
        }
        private List<Customer> Customers { get; set; }
        public void LoadCustomer(string search_pattern)
        {
            Customers = DataRepository.Customer(search_pattern);
        }
        public void NotifyDiscount()
        {
            foreach (var x in Customers)
```

```
            {
                SmsProvider.SendSms(x, "sleva");
            }
        }
    }
```

Kód 4 Vzorová implementace constructor DI

Zde hned na první pohled vidíme, jaké závislosti má tato třída. Závislosti jsou přijímány v konstruktoru a bez závislých služeb/komponent není možné třídu použít. Separace závislostí nám nyní umožnila jednodušší testování, protože místo reálné třídy *SmsProvider* můžeme použít její náhradu např. *FakeSmsProvider*, která bude místo rozesílání skutečných sms vypisovat informace do konzole. Také je možné přeložit kód, i když není implementovaná třída databáze (původně třída *DatabaseContext*).

Constructor injection je preferovaná metoda DI. Pokud z nějakého důvodu nemůžeme použít constructor DI, můžeme použít property DI.

DI pomocí property injection

Vzorovou třídu upravíme podle property injection.

```
public class CustomerNotifierWithPropertyDI
    {
        public IDataRepository DataRepository { get; set; }
        public ISMSProvider SmsProvider { get; set; }
        private List<Customer> Customers { get; set; }
        public void LoadCustomer(string search_pattern)
        {
            Customers = DataRepository.Customer(search_pattern);
        }
        public void NotifyDiscount()
        {
            foreach (var x in Customers)
            {
                SmsProvider.SendSms(x, "sleva");
            }
        }
    }
```

Kód 5 Vzorová implementace property DI

Pravidlem u property injection je, že všechny public property, které jsou typem interface a mají veřejný setter budou považovány za závislost.

Výhodou u property DI je, že jde použít za všech okolností na rozdíl od constructor injection, který se např. v ASP.NET aplikacích nedá implementovat.

Nevýhodou je, že závislosti nejsou na první pohled tak viditelné a třídu je možné sestrojit i bez dosazení těchto závislostí.

Pokud bychom měli shrnout hlavní přednosti návrhového vzoru DI, tak je to [25]:

- Separace kódu
- Možnost změny implementace za chodu nebo při překladu
- Možnost změny implementace komponent bez nutnosti změny a překladu kódu.
- Vyznačení závislostí dané třídy
- Lepší možnosti testování
- Tvorba modulárních systémů
- Možnost přeložit kód i bez implementovaných závislostí

4.4 IoC

IoC (Inversion of Control) je technika umožňující sestavení objektů z různých částí podle definovaných pravidel buďto při kompilaci nebo za chodu programu. IoC úzce souvisí s DI a využívá jej [26].

Hlavní částí Ioc je Ioc kontejner. Tento kontejner nám z dodaných částí vytváří hotové instance podle daných pravidel.

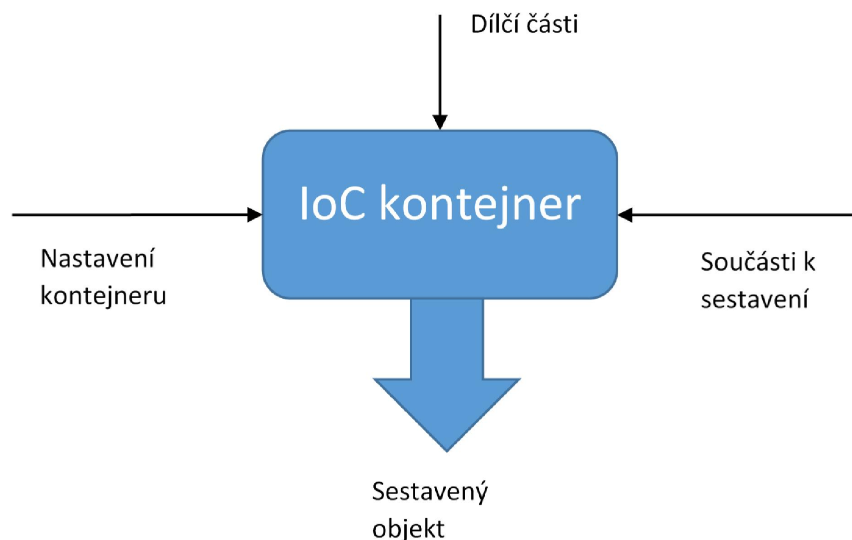


Schéma 12 Princip IoC

Princip techniky IoC spočívá v tom, že vytvoříme instanci IoC kontejneru. Do kontejneru zaregistrujeme dílčí součásti. Poté do kontejneru registrujeme hlavní části, které chceme, aby nám kontejner zkompletoval.

Základní druhy registrací jsou [26], [27]:

- **Registrace typu** – kontejner se při žádosti o vydání tohoto typu bude snažit inicializovat tento typ pomocí defaultního konstruktoru
- **Registrace interface – typ** – k danému interface přiřadíme konkrétní implementaci. Kontejner se poté při žádosti o vydání daného interface pokusí vytvořit instanci daného typu.
- **Registrace interface – typ/typ pomocí delegátu** – kontejneru definujeme způsob, jakým má vrátit daný typ/ interface, skrze delegát nebo labda výraz.

U každé registrace si můžeme nastavit životnost objektu. Základní životnosti objektů jsou:

- **Singleton** – při požadavku o vydání instance typu označeného jako singleton se vytváří vždy pouze jedna instance daného objektu a ta je platná po celou dobu chodu programu.
- **MultiInstance** – při požadavku o vydání instance typu označeného jako multiinstance je při každé žádosti vydána nová instance daného typu.

Kromě základních životností existují i některé další životnosti, které implementují pouze některé kontejnery:

- **PerRequest** – využívá se v rámci WCF. Pro každé nové spojení je vytvořena instance daného typu s životností singleton a při každé žádosti o vydání instance v rámci požadavku se vydává tato stejná instance.
- **PerThread** – pro každé vlákno je vytvořena nová instance daného typu. Při žádosti o vydání instance v rámci vlákna se vydává vždy stejná instance.
- **Scoped** – speciální případ. Umožnuje vymezení sekce. Kdykoli se v této sekci vyskytne požadavek na instanci, v rámci sekce je vždy vrácena stejná instance.

Po registraci všech částí můžeme vyslat požadavek na kontejner, aby nám vydal instanci určitého typu. Kontejner poté vrátí instanci daného s inicializovanými závislostmi. Kontejnery většinou podporují i další nastavení jako např.:

Lazy Initialization – až při prvním požadavku na daný typ je od tohoto typu vytvořena instance. To umožňuje šetřit paměť a nezatěžuje tolik CPU

Intercepting – technika založená na dynamickém generovnání kódu. Slouží k obalování volání metod. Využívá se např. pro logování.

```
public void IocSetup()
{
    var container = TinyIoC.TinyIoCContainer.Current;

    // registrace typu
    container.Register<MessageService>();

    //registrace interface - typ
    container.Register<IMessageService, MessageService>();

    //registrace interface - typ pomocí delegátu
    container.Register<IMessageService>((a,b) => new MessageService());
```

```
            //registrace singletonu
            container.Register<MessageService>().AsSingleton();

            //registrace multiInstance
            container.Register<MessageService>().AsMultiInstance();

            //registrace per request
            container.Register<MessageService>().AsPerRequestSingleton();

            //sestavení objektu
            IMessageService messageService = container.Resolve<IMessageSer-
vice>();
        }
```

Kód 6 Ukázka IoC kontejneru (TinyIoC)

Programátor si většinou IoC kontejner neimplementuje sám, ale využívá již kontejner implementovaný od třetí strany. Na trhu jsou dostupné různé implementace IoC kontejnerů pro různé platformy s různými možnostmi. Mezi nejznámnější patří [27]:

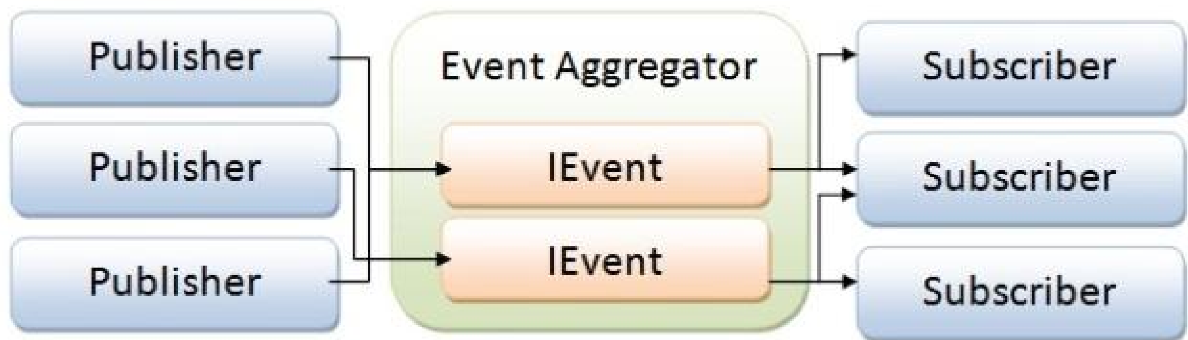
- Windsor Castle
- Unity
- Ninject
- Caliburn
- MEF
- Spring.NET

MvX (použitý v rámci knihovny MVVMCross)

4.5 Publish - subscribe návrhový vzor

Publish – subscribe je návrhový vzor určený pro anonymní rozesílání zpráv. Tento vzor obsahuje 3 objekty [28]:

- Publisher – objekt, který rozesílá zprávy (zdroj zprávy)
- Subscriber – objekt, který se přihlásí k odběru zpráv (cíl zprávy)
- Broker (server, kanál, aggregator) – objekt, který zprostředkovává komunikaci mezi účastníky



Obrázek 9 Publish-subcribe návrhový vzor [28]

Princip fungování je takový, že si nejprve definujeme zprávu, kterou chceme posílat. Poté třída, která chce danou zprávu odchytávat, provede registraci pro odběr daných zpráv skrze třídu broker. Nakonec třída, která chce rozeslat zprávu tak provede skrze metodu třídy broker.

Původně tento návrhový vzor popisoval síťovou topologii pro vysokou škálovatelnost, avšak své uplatnění našel i v rámci architektury MVVM. Zde se používá pro rozesílání zpráv mezi ViewModely. V rámci MVVM v .NET je tento návrhový vzor implementován prostřednictvím třídy *WeakReference*, která umožňuje efektivnější zprávu paměti v systémech MVVM.

```
public class InfoMessage
{
    public string Info { get; set; }
}

public class SamplePublisherViewModel
{
    private void Notify()
    {
        //ziskani instance zprostredkovatele
        IMessenger messanger = Messenger.Default;

        //zaslani zpravy
        messanger.Send<InfoMessage>(new InfoMessage() {Info = "test"});
    }
}

public class SampleSubscriberViewModel
{
    public SampleSubscriberViewModel()
    {
        //ziskani instance zprostredkovatele
        IMessenger messanger = Messenger.Default;

        //registrace pro zpravu
        messanger.Register<InfoMessage>(this, (o) =>
        {
            Console.WriteLine(o.Info);
        });
```

```
        }
    }
```

Kód 7 Implementace publish-subcribe vzoru v MVVLight

Hlavní výhodou toho vzoru jsou [28]:

- Odstranění závislostí mezi zdrojem a příjemcem. Původce zprávy nemusí vůbec nic vědět o tom, kdo je příjemce.
- Škálovatelnost. Při použití tohoto vzoru je možné vytvářet rozsáhlé modulární systémy.
- V rámci .NET použití tohoto vzoru může nahradit klasické eventy, při jejichž neopatrné manipulaci můžou vznikat „memory leaky“.

Mezi nevýhody tohoto vzoru patří:

- Zpožděné doručování zpráv. Zprávy nemusí být doručovány okamžitě, na rozdíl přímého volání.

Nemožnost ověření doručení. Jelikož zprávy do systému pouze vkládáme, nemůžeme si ověřit, zda se některý z posluchačů nenalézá např. v chybovém stavu.

4.6 Asynchronní návrhové vzory

Kromě klasických návrhových vzorů existují i návrhové vzory popisující vytváření a zpracování asynchronních volání funkcí. Rozdíl mezi synchronním a asynchronním volání funkcí popisuje následující grafika.

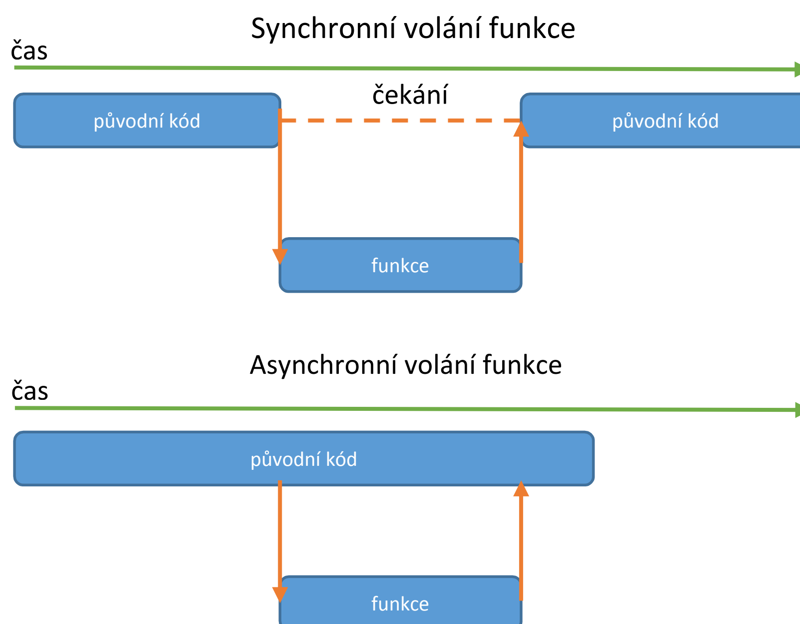


Schéma 13 Synchronní a asynchronní volání

Mezi výhody asynchronního zpracování patří rychlejší zpracování funkcí[11] a v případě GUI aplikací „svižnější“ odezva GUI. Mezi hlavní nevýhody patří složitá implementace.

V rámci desktopových aplikací není povinnost řešit dlouhotrvající funkce asynchronně. Avšak v mobilním prostředí je již nutné spočítat si, jak dlouho bude které volání funkce trvat, protože mobilní platformy nepovolují zablokování UI vlákna. Windows Phone 8 neumožňuje provádět synchronně operace trvající déle než 50ms. Android neumožňuje zablokovat hlavní vlákno na déle než 5s.

Prostředí .NET obsahuje 3 implementace asynchronních návrhových vzorů. V následujících kapitolách jsou jednotlivé vzory rozebrány podrobněji [29].

4.6.1 APM

APM neboli asynchronní programový model je nejstarší asynchronní vzor v rámci .NET. Je založen na dvou metodách ve formátu **Begin***Operation* a **End***Operation* (kde *operation* je název dané funkce) a interface *IAsyncResult*, který vrací metoda **Begin***Operation.* Metoda begin přijímá kromě svých vstupních parametrů také delegát pro callback a je uživatelsky definovaný objekt. Metoda **End***Operation* se volá vždy po skončení operace a vrací nám výsledek této operace [29].

```
public interface IAsyncResult
{
    bool IsCompleted { get; }
    WaitHandle AsyncWaitHandle { get; }
    object AsyncState { get; }
    bool CompletedSynchronously { get; }
}
```

Kód 8 Implementace IAsyncResult

Tento model nabízí 3 možnosti zpracování:

Synchronní volání

Pokud z nějakého důvodu potřebujeme, aby se daná operace vykonala synchronně, je možnost vykonat tuto operaci skrze tento model dvěma způsoby.

- Počkat na dokončení operace zavoláním metody `AsyncWaitHandle.WaitOne()` nad instancí objektu vráceného z **Begin***Operation*

[11] Rychlejším zpracováním se myslí paralelní běh funkcí a projeví pouze u funkcí, které ke svému dokončení potřebují čas, který je větší než čas potřebný pro alokaci a spuštění vedlejšího vlákna, které tento požadavek vyřídí.

- Zavolat **Begin***Operation* a následně **End***Operation*

```
public long GetDataSize(string adress)
{
    HttpWebRequest request = HttpWebRequest.CreateHttp(adress);
    IAsyncResult handle=request.BeginGetResponse(null, null);
    DoWork();
    handle.AsyncWaitHandle.WaitOne();
    WebResponse response=request.EndGetResponse(handle);
    return response.ContentLength;
}

public long GetDataSize2(string adress)
{
    HttpWebRequest request = HttpWebRequest.CreateHttp(adress);
    IAsyncResult handle = request.BeginGetResponse(null, null);
    DoWork();
    WebResponse response = request.EndGetResponse(handle);
    return response.ContentLength;
}
```

Kód 9 Synchronní volání v modelu APM

Cyklické dotazování (pooling)

Po spuštění asynchronní operace se můžeme také cyklicky dotazovat, zda je již operace dokončena.

```
public long GetDataSizeCycle(string adress)
{
    HttpWebRequest request = HttpWebRequest.CreateHttp(adress);
    IAsyncResult handle = request.BeginGetResponse(null, null);
    while (!handle.IsCompleted)
        DoWork();
    WebResponse response = request.EndGetResponse(handle);
    return response.ContentLength;
}
```

Kód 10 Cyklické dotazování v APM

Zavolání po dokončení (callback)

Nejčastějším způsobem zpracování operace je využití callback funkce, která je zavolána po dokončení operace.

```
public long GetDataSizeCallback(string adress)
{
    HttpWebRequest request = HttpWebRequest.CreateHttp(adress);
    request.BeginGetResponse((handle) =>
    {
        WebResponse response = request.EndGetResponse(handle);
    }, null);
    DoWork();
    return 0;
}
```

Kód 11 Callback v APM

Implementování APM modelu je relativně náročný proces. V rámci .NET Frameworku je však možnost volat všechny synchronní metody asynchronně skrze delegáty.

```
public int Sum(int a, int b)
{
    return a + b;
}
public void CallSum()
{
    Func<int,int,int> SumDelegate = Sum;
    IAsyncResult handle = SumDelegate.BeginInvoke(2, 3, null, null);
    DoWork();
    int result = SumDelegate.EndInvoke(handle);
}
```

Kód 12 Asynchronní volání synchronní metody v APM

4.6.2 EAP

EAP neboli eventový asynchronní model je dalším modelem pro řízení asynchronních operací. Model se vyznačuje tím, že třída obsahuje spouštěcí funkci ve formátu *Operation***Async**, která přijímá parametry své standartní parametry + uživatelsky definovaný objekt. Ke každé asynchroní funkci existuje event *Operation***Completed**, který je vyvolán po skončení funkce a kde je možné zjistit výsledek operace. Ke každé asynchronní operaci by měla existovat ještě funkce sloužící pro její zrušení ve formátu *Operation***AsyncCancel**, která nepřijímá žádné parametry a vrací *void [29]*.

U tohoto modelu je důležité registrovat se k eventu dříve, než zavoláme příslušnou metodu, protože občas se může stát, že je metoda provedena synchronně (nebo může dojít ke kontext switch) a nemusíme zachytit její výsledek.

```
SmtpClient smtp = new SmtpClient();
smtp.SendCompleted += (sender, argument) =>
{
    if(argument.Cancelled)
        Console.WriteLine("canceled");
    else if (argument.Error != null)
        Console.WriteLine("error");
    else
        Console.WriteLine("send");
};
smtp.SendAsync(new MailMessage("me@me.cz",adress), null);
smtp.SendAsyncCancel();
```

Kód 13 Volání v EAP

Implementace modelu je jednodušší než v případě APM. Další výhodou je, že na event o dokončení operace se může registrovat více posluchačů z různých tříd.

```
public class Calculator
{
```

```
        public class MyEventArgs : AsyncCompletedEventArgs
        {
            public MyEventArgs(double result, Exception exception, bool can-
celled, object userState)
                : base(exception, cancelled, userState)
            {
                Result = result;
            }

            public double Result { get; private set; }
        }

        public event EventHandler<MyEventArgs> CalculationComplete;
        private Thread background_thread;
        private object lastUserState;

        protected virtual void OnCalculationComplete(MyEventArgs e)
        {
            lastUserState = null;
            EventHandler<MyEventArgs> handler = CalculationComplete;
            if (handler != null) handler(this, e);
        }


        public double Calculate(double a, double b)
        {
            var result = a / b;
            //simulace dlouhotrvajici operace
            Thread.Sleep(1000);
            return result;
        }

        public void CalculateAsync(double a, double b, object userState)
        {
            lastUserState = userState;
            background_thread = new Thread(() =>
            {
                try
                {
                    var result = Calculate(a, b);
                    OnCalculationComplete(new MyEventArgs(result, null, false,
userState));
                }
                catch (Exception ex)
                {
                    OnCalculationComplete(new MyEventArgs(0, ex, false,
userState));
                }

            });
            background_thread.Start();
        }

        public void CalculateAsyncCancel()
        {
            if (background_thread != null && background_thread.IsAlive)
            {
                OnCalculationComplete(new MyEventArgs(0, null, true,
lastUserState));
            }
        }
    }
```

Kód 14 Vzorová implementace EAP

4.6.3 TAP

TAP neboli taskový asynchronní model je nejnovější model pro práci s asynchronními operacemi, který byl uveden s příchodem .NET 4. Model je založen pouze na tom, že obsahuje jednu metodu *Operation***Async**, která přijímá pouze své vstupní parametry a vrací objekt typu *Task* nebo *Task<Result>*. Třídu *Task* můžeme chápat jako rozšíření třídy *Thread* a obsahuje navíc pokročilé metody pro správu vláken jako např. zřetězení volání Tasků, agregace výjimek, plánování spuštění [29].

TAP nabízí více možností volání:

Synchronní volání

Pokud potřebujeme zavolat operaci synchronně, zavoláme na objektu typu *Task* metodu *Wait*() v případě, že nevrací žádná data, nebo property *Result*, která zajistí synchronní běh a vrátí nám přímo výsledek.

```
public long GetDataSizeTask(string adress)
{
    HttpWebRequest request = HttpWebRequest.CreateHttp(adress);
    WebResponse response = request.GetResponseAsync().Result;
    return response.ContentLength;
}
```

Kód 15 Synchronní volání v TAP

Cyklické dotazování (pooling)

Po spuštění asynchronní operace se můžeme také cyklicky dotazovat, zda je již operace dokončena. Cyklické dotazování není obvyklý postup při práci s tímto modelem.

```
public long GetDataSizeTask(string adress)
{
    HttpWebRequest request = HttpWebRequest.CreateHttp(adress);
    var task = request.GetResponseAsync();
    while (!task.IsCompleted)
        DoWork();
    WebResponse response = task.Result;
    return response.ContentLength;
}
```

Kód 16 Cyklické dotazování v TAP

Řetězení operací

Tasky s sebou přináší možnost využít řetězení operací, což velmi usnadňuje práci, pokud potřebujeme sekvenčně volat více operací paralelně. Pokud některá s operací selže, můžeme vyjímku zachytit ve formě agregované vyjímky.

```
        public long GetDataSizeTaskAggregate(string adress)
        {
            HttpWebRequest request = HttpWebRequest.CreateHttp(adress);
            request.GetResponseAsync().ContinueWith(last =>
            {
                return HttpWebRequest.CreateHttp(last.Result.ResponseUri).GetRe-
sponse();
            }).ContinueWith(last =>
            {
                if (last.Exception != null)
                    Console.WriteLine("error behem provadeni");
                else
                    Console.WriteLine(last.Result.ContentLength);
            });
            return 0;
        }
```

Kód 17 Řetězení tasků v TAP

Zpracování pomocí await

Hlavní novinkou .NET 4.5 bylo uvedení klíčových slov *await* a *async*, kde *await* slouží pro konzumaci asynchronních metod a *async* usnadňuje tvorbu asynchronních metod. Klíčové slovo *await* oznámí překladači, že má v daném místě implementovat stavový automat a klíčovým slovem *async* se označují metody obsahující jeden nebo více slov *async*.

```
        public async void GetDataSizeTaskNew(string adress)
        {
            HttpWebRequest request = HttpWebRequest.CreateHttp(adress);
            WebResponse response = await request.GetResponseAsync();
            Console.WriteLine(response.ContentLength);
        }
```

Kód 18 Volání asynchronní metody v TAP pomocí await

Od .NET Frameworku 4.5 je nejrozšířenější model TAP, kde se využívá volání metod pomocí klíčového slova *await*. Toto zpracování asynchronních operací je velmi jednoduché a efektivní. Také s příchodem tohoto frameworku byly dodělány implementace starších modelů APM, EAP do TAP.

Implementace třídy podporující TAP je také velmi jednoduchá (oproti EAP, APM). Při implementaci lze využít třídy *TaskCompletionSource* pro zjednodušení práce.

```
    public class Calculator2
    {
        public double Calculate(double a, double b)
        {
            var result = a / b;
            //simulace dlouhotrvajici operace
            Thread.Sleep(1000);
            return result;
        }

        private TaskCompletionSource<double> tcs;
        public Task<double> CalculateAsync(double a, double b)
        {
```

```
            tcs = new TaskCompletionSource<double>();
            Task.Factory.StartNew(() =>
            {
                try
                {
                    var result = Calculate(a, b);
                    tcs.TrySetResult(result);
                }
                catch (Exception ex)
                {
                    tcs.SetException(ex);
                }
            });
            return tcs.Task;
        }

        public void CalculateAsyncCancel()
        {
            if (tcs != null)
                tcs.SetCanceled();
        }
    }
```

Kód 19 Vzorová implementace TAP[12]

---

[12] Zde demonstrovaná ukázka pouze naznačuje užití třídy *TaskCompetitionSource* pro implementaci Cancelable TAP. V tomto případě není zrušena původní operace a zůstávají tzv. zombie vlákna. Oficiální doporučení pro implementaci Cancelable TAP je použití třídy *CancellationTokenSource* a cyklické ověřování zrušení v metodě *DoWork*.

5 FRAMEWORK MVVMCROSS

MVVMCross je multiplatformní framework usnadňující vytváření aplikací v prostředí Xamarinu. Je založen na architektuře MVVM a z hlediska návrhových vzorů využívá hlavně IoC a Dependency Injection. Jeho autorem je Stuart Slodge. Aktuálně je framework zdarma a funguje jako OpenSource projekt, který má mnoho přispěvovatelů. Původně projekt vznikl jako fork MonoCrossu (od Microsoftu) a převzal další věci z různých frameworků jako MVVMLight a ASP.NET MVC [31].

Cílem frameworku je usnadnit vytváření aplikací pomocí architektury MVVM a dosáhnout separace vrstev Model a ViewModel do PCL[13] a View vrstev pro každou platformu do platformních (GUI) projektů.

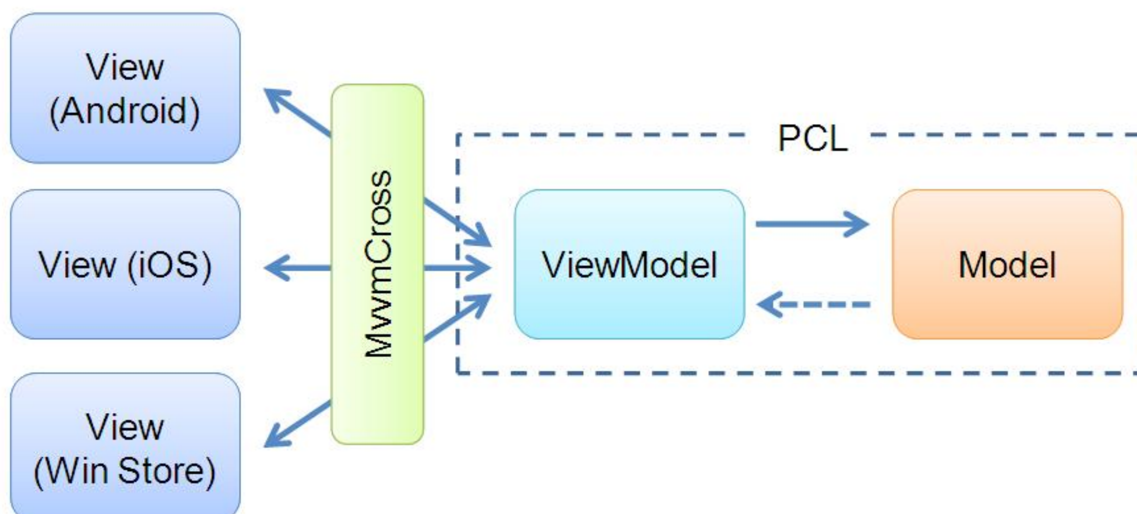


Schéma 14 Princip MVVMCross [30]

MVVMCross aktuálně podporuje tyto platformy [31]:

- Xamarin.iOS
- Xamarin.Android
- Windows Phone
- WPF
- Windows Store
- Mac

[13] Portable Class Library je zvláštní případ .NET Assemblies, která obsahuje platformě přenositelný kód.

Ačkoliv se jedná o velmi nový framework, stihl si okolo sebe vybudovat poměrně slušnou uživatelskou základnu. Také je zde mnoho významných společností, které se rozhodly založit své aplikace na tomto frameworku. Mezi společnosti využívající tento framework patří [36]:

- IBM
- Bosh
- Honeywell
- Nokia

Základní vlastnosti tohoto frameworku jsou rozebrány v následujících kapitolách.

5.1 Navigace

Pomocí MVVMcross můžeme implementovat velice jednoduše „univerzální“ navigaci v rámci naší aplikace. Tato navigace se poté přenese na každou platformu zvlášť, kde je implementována tak, jak je pro danou platformu obvyklé. Standardně je navigace implementována pro jednotlivé platformy takto:

- Pro Windows Phone se naviguje mezi *Page* třídami
- Pro iOS se naviguje mezi *UIViewController* třídami
- Pro Android se naviguje mezi *Activity* třídami

MVVMCross Framework také podporuje navigaci pro tabulární systém navigace (Tabs) pro každou platformu. Pro android existuje také implementace pro navigaci skrze fragmenty.

Systém navigace je založen na návrhovém vzoru ViewModelLocator a implementuje se následujícím způsobem:

1. Nejdříve se definují jednotlivé ViewModely v PCL projektu. Každý ViewModel musí dědit od *MvxViewModel* třídy. Důležité je také, aby každý název této třídy končil příponou ViewModel, protože pak se provede automatická registrace ViewModelu do ViewModelLocatoru.
2. Pro navigaci využijeme metody *ShowViewModel<T>* kde T je daný ViewModel, na který chceme navigovat. Na rozdíl od klasické navigace, kde se většinou naviguje mezi View třídami, zde se naviguje mezi ViewModely. Framework poté prohledá ViewModel locator na dané platformě a nalezne k danému ViewModelu příslušný View, který zobrazí.
3. V GUI projektu definujeme pro každý ViewModel jeden View. Na každé platformě musí View třída dědit od příslušné *MvxView* základní třídy (pro iOS *MvxViewController*, pro Android *MvxActivity*…). Pokud View pojmenujeme stejně jako ViewModel a k názvu View přidáme příponu View, dojde k automatickému spojení View s Viewmodelem na základě jmenné konvence.
4. V PCL projektu ve třídě *App* zaregistrujeme, jaký ViewModel se má spustit po spuštění aplikace pomocí funkce *RegisterAppStart<T>* kde T je třída odvozená od *MvxViewModel*.

Velmi často se stává, že při navigaci potřebujeme předat parametr nebo vlastní objekt. MVVMCross umožňuje při navigaci předávat jak jednoduché parametry, tak instance tříd, které jdou serializovat. Ovšem při předávání celých instancí tříd je třeba dávat pozor na to, že zde dochází k serializaci a poté k deserializaci a tím instance získává jinou adresu. Proto je velmi doporučovaný způsob navigace ten, kdy si předáváme pouze id či jiný identifikátor, přes který poté danou instanci získáme se sdílený service [31].

5.2 Databinding

Nejdůležitější částí MVVMCross frameworku je bindování. Návrhový vzor MVVM původně vznikl pro platformu WPF, kde hrálo bindování velmi důležitou roli. Jednalo se o techniku, při níž docházelo ke spojení vrstvy View s ViewModelem [2]. Původní WPF binding vypadal následovně.

```
        <StackPanel>
            <!-- cely datacontext do property text -->
            <TextBlock Text="{Binding}"/>
            <!-- Property1 z datacontextu do property text -->
            <TextBlock Text="{Binding Property1}"/>
            <!-- Property1 z datacontextu do property text s vyuzitim obousmerne 
vazby -->
            <TextBlock Text="{Binding Property1, Mode=TwoWay}"/>
            <!-- Property1 z datacontextu do property text s vyuzitim prevadece -
->
            <TextBlock Text="{Binding Property1, Converter={StaticResource preva-
dec}}"/>
            <!-- Property1 z datacontextu do property text s vyuzitim string.For-
mat -->
            <TextBlock Text="{Binding Property1, StringFormat=Data:\{0:N1\}}"/>
        </StackPanel>
```

Kód 20 Ukázka WPF bindingu

V této ukázce kódu můžeme vidět nejčastější případy použití bindingu ve WPF. Výhodou bindingu bylo to, že pokud ViewModel implementoval rozhraní *INotifyPropertyChanged*, tak pokud došlo v kódu ke změně hodnoty dané property, bindovací systém informoval o této změně View a došlo k automatické aktualizaci. Pokud bylo bindování nastaveno na mód TwoWay, fungovali notifikaci a naopak tzn., že pokud se změnila hodnota ve View, bindovací systém upravil automaticky hodnotu dané property ve ViewModelu.

Framework MVVMCross umožnil použití modifikovaného systému bindovaní i v prostředí Xamarinu. Systém bindování byl implementován na každou platformu mírně odlišně z důvodu, že každá platforma definuje jiným způsobem vrstvu View [31].

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:local="http://schemas.android.com/apk/res-auto"
    android:orientation="vertical"
```

```
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
  <!-- bindovani celeho datacontextu na property text -->
    <TextView
        local:MvxBind="Text ." />
  <!-- property1 z datacontextu na property text -->
  <TextView
      local:MvxBind="Text property1y" />
  <!-- property1 z datacontextu na property text s vyuzitim obousmerne vazby -->
  <TextView
      local:MvxBind="Text Property1, Mode TwoWay" />
  <!-- property1 z datacontextu na property text s vyuzitim prevadece -->
  <TextView
      local:MvxBind="Text prevadec(Property1)" />
  <!-- property1 z datacontextu na property text s vyuzitim string.Format -->
  <TextView
      local:MvxBind="Text Format('Data:{0:N1}', Property1)" />
```

Kód 21 Ukázka MVVMCross bindingu pro Xamarin.Android

Pro Xamarin.Android je možné bindování definovat přímo v axml souborech. Nejprve je nutné definovat si rozšíření XML (v ukázkovém kódu *local*), pak je již možné pomocí tohoto rozšíření nastavovat vlastnost *MvxBind*, kde první parametr je property na cílovém elementu a druhý parametr oddělený mezerou je zdrojová property na ViewModelu. Pokud chceme bindovat na více property cílového elementu, oddělujeme jednotlivé binding expression středníkem.

Pro platformu Xamarin.iOS je možné definovat bindování pouze v kódu.

```
            var binding = this.CreateBindingSet<LoginView, LoginViewModel>();
            //bindovani celeho datacontextu na property text
            binding.Bind(label).For(o => o.Text).To(o => o);

            //property1 z datacontextu na property text
            binding.Bind(label).For(o => o.Text).To(o => o.Property1);

            //property1 z datacontextu na property text s vyuzitim obousmerne
vazby
            binding.Bind(label).For(o => o.Text).To(o => o).TwoWay();

            //property1 z datacontextu na property text s vyuzitim prevadece
            binding.Bind(label).For(o => o.Text).To(o => o.Property1).WithCon-
version(prevadec,null);

            //property1 z datacontextu na property text s vyuzitim string.Format
(alternativní zápis)
            binding.Bind(label).FullyDescribed("Text Format('Data:{0:N1}', Prope-
rty1)");

            //zapnuti bindingu
            binding.Apply();
```

Kód 22 Ukázka MVVMCross bindingu pro Xamarin.iOS

Bindování se většinou definuje v metodě *ViewDidLoad* třídy typu View. Nejprve je třeba vytvořit si instanci typu *MvxFluentBindingDescriptionSet*. Poté pomocí fluent API skrze

metodu *Bind* vybrat element, na který chceme aplikovat binding, skrze metodu *For* zvolit cílovou property na elementu a nakonec pomocí metody *To* vybrat cílovou property na ViewModelu. Na konec je třeba zavolat metodu *Apply* pro aplikaci definovaných bindingů.

Jelikož MVVMCross binding pracuje nad standartním rozhraním *INotifyPropertyChanged*, je možné použít standartní binding pro platformy WPF a Windows Phone 8. Pokud by však programátor chtěl využít MVVMCross bindování i na platformě WP8, je možné stáhnout si toto bindování ve formě pluginu[14] z balíčkovacího systému Nuget.

Kromě zde uvedených příkladů obsahuje MVVMCross databinding spoustu dalších funkcí jako např. podporu fallback value, podporu binding combinerů, podporu bindování pro eventy, podporu string literal a další. Bohužel probrat důkladně všechny funkce MVVMCross bindování by bylo nad rámec této práce, takže jsou zde uvedeny pouze nejdůležitější a nejpoužívanější funkce.

5.3 Pluginy

Celý framework je navrhnut modulárně. Ne všechna funkcionalita je implementována v základním frameworku. Doplňující funkcionalitu je možné doinstalovat skrze pluginový systém mvvmcrosu. Většinou jsou pluginy umístěny přímo v systému oficiálním balíčkovacím repositáři systému Nuget.

Pluginy fungují na principu dependency injection. V PCL projektu se odkazujeme na příslušný plugin prostřednictvím interface. V GUI projektu pak přidáním téhož pluginu dojde k jeho automatické registraci skrze techniku IoC.

Oddělení některých triviálních funkcí do samostatných pluginů může na první pohled působit nevhodně a zmateně, avšak toto oddělení je z důvodu AOT kompilace na platformě iOS. Bez modulárního systému docházelo v minulých verzích MVVMCrossu k radikálnímu zvětšení výsledného výstupního souboru aplikace (IPA).

Seznam nejpoužívanějších pluginů je zde [31]:

[14] Pro Windows Phone platformu existuje plugin MvvmCross.BindingEx, který přidává podporu MVVMCross bindingu. Pro jeho aktivaci je však třeba upravit třídu setup v GUI projektu pro platformu WP8.

Accelerometr – plugin slouží k získání údajů z akcelerometru. Navržené rozhraní je následující

```
public interface IMvxAccelerometer
    {
        void Start();
        void Stop();
        bool Started { get; }
        MvxAccelerometerReading LastReading { get; }
        event EventHandler<MvxValueEventArgs<MvxAccelerometerReading>> Readin-
gAvailable;
    }
```

Kód 23 Interface pluginu *Accelerometr*

Color – velice užitečný plugin sloužící k multiplatformní manipulaci s barvami. V rámci PCL dostaneme třídu *MvxColor*, ve které pomocí RGB modelu definujeme naši barvu. Na cílové platformě dostaneme převaděč z *MvxColor* do nativní barvy[15] pro danou platformu ve 2 formách:

- Skrze rozšiřující metodu *ToNativeColor* třídy *MvxColor*
- Skrze ValueConverter *NativeColor*, který je možné použít v databindingu

Email – plugin slouží k vyvolání a předvyplnění okna pro zaslání emailové zprávy. Navržené rozhraní je následující

```
public interface IMvxComposeEmailTask
    {
        void ComposeEmail(string to, string cc, string subject, string body, bool
isHtml);
    }
```

Kód 24 Interface pluginu *Email*

File – tento plugin poskytuje platformě nezávislou abstrakci pro práci se souborovým systémem. Systém využívá následující cesty pro dané platformy:

- Android - Context.FilesDir
- iOS - Environment.SpecialFolder.MyDocuments
- WindowsPhone - app-specific isolated storage
- WindowsStore - Windows.Storage.ApplicationData.Current.LocalFolder.Path
- Wpf - Environment.SpecialFolder.ApplicationDataLocation

Podporující operace jsou znázorněny v interface viz. níže.

```
public interface IMvxFileStore
    {
```

[15] Pro Android platformu se *MvxColor* převádí na *Android.Graphics.Color*. Pro Windows platformu se převádí na *SolidColorBrush*. Pro iOS platformu se převádí na *MonoTouch.UIKit.UIColor*.

```
        bool TryReadTextFile(string path, out string contents);
        bool TryReadBinaryFile(string path, out Byte[] contents);
        bool TryReadBinaryFile(string path, Func<Stream, bool> readMethod);
        void WriteFile(string path, string contents);
        void WriteFile(string path, IEnumerable<Byte> contents);
        void WriteFile(string path, Action<Stream> writeMethod);
        bool TryMove(string from, string to, bool deleteExistingTo);
        bool Exists(string path);
        bool FolderExists(string folderPath);
        string PathCombine(string items0, string items1);
        string NativePath(string path);

        void EnsureFolderExists(string folderPath);
        IEnumerable<string> GetFilesIn(string folderPath);
        void DeleteFile(string path);
        void DeleteFolder(string folderPath, bool recursive);
    }
```

Kód 25 Interface pluginu *File*

Location – plugin poskytuje přístup k datům z GPS.

```
public interface IMvxGeoLocationWatcher
    {
        void Start(
            MvxGeoLocationOptions options,
            Action<MvxGeoLocation> success,
            Action<MvxLocationError> error);
        void Stop();
        bool Started { get; }
    }
```

Kód 26 Interface pluginu *Location*

Messanger – implementace message pumpy využívající třídu *WeakReference* pro přihlašování k událostem. Třída poskytuje následující operace:

- Vystavovací metody
    - *Publish*
- Přihlašovací metody
    - *Subscribe*
    - *SubscribeOnMainThread*
    - *SubscribeOnThreadPoolThread*
    - *Unsubscribe*
- Detekční metody
    - *HasSubscriptionsFor*
    - *HasSubscriptionsForTag*
    - *CountSubscriptionsFor*
    - *CountSubscriptionsForTag*
    - *GetSubscriptionTagsFor*
- Čisticí metody
    - *RequestPurge*
    - *RequestPurgeAll*

Při posílání zprávy je třeba, aby zpráva dědila od třídy *MvxMessage*. Při přihlášení je třeba si uchovat token z důvodu garbage collectoru.

PhoneCall – umožnuje zobrazit obrazovku nebo rovnou vytočit dané telefonní číslo. Interface je následující.

```
public interface IMvxPhoneCallTask
{
    void MakePhoneCall(string name, string number);
}
```

Kód 27 Interface pluginu *PhoneCall*

PictureChooser – umožnuje načíst obrázek z fotoaparátu nebo z knihovny daného zařízení. Rozhraní pluginu je následující.

```
public interface IMvxPictureChooserTask
{
    void ChoosePictureFromLibrary(int maxPixelDimension, int percentQuality,
Action<Stream> pictureAvailable, Action assumeCancelled);

    void TakePicture(int maxPixelDimension, int percentQuality,
Action<Stream> pictureAvailable, Action assumeCancelled);
}
```

Kód 28 Interface pluginu *PictureChooser*

WebBrowser – umožňuje spustit prohlížeč a otevřít v něm danou webovou stránku.

```
public interface IMvxWebBrowserTask
{
    void ShowWebPage(string url);
}
```

Kód 29 Interface pluginu *WebBrowser*

II. PRAKTICKÁ ČÁST

6 APLIKACE STUDUJ UTB

V rámci praktické části této práce bude popsána případová studie založená na autorem vytvořené mobilní aplikaci „Studuj UTB“, která je plně funkční a dostupná pro platformy Android, iOS a Windows Phone 8.

Aplikace je dostupná v jednotlivých aplikačních obchodech:

- Pro Android – https://play.google.com/store/apps/details?id=com.UTBShow.Droid
- Pro iOS – https://itunes.apple.com/us/app/studuj-utb/id725109311
- Pro Windows Phone 8 – http://www.windowsphone.com/en-us/store/app/studuj-utb/b23f297e-5c28-4d58-92d8-bbf32d166dc3
- Souhrnně na microsite – http://aplikace.utb.cz/

Aplikace byla vytvořena pomocí moderních metod a technologie použité při její tvorbě jsou popsány v následujících kapitolách.

Aplikace „Studuj UTB“ je mobilní multiplatformní aplikace. Je připravena pro všechny uchazeče o bakalářské, magisterské a doktorské studium. Je možné dozvědět se vše důležité o Univerzitě Tomáše Bati (UTB), nabízených studijních programech, fakultách i zajímavostech ze studentského života ve Zlíně. Prostřednictvím fotografií a vizualizací je možné nahlédnout také do Univerzitního centra, knihovny, vědeckotechnického parku i budoucích budov Univerzity Tomáše Bati.

Dále je možné v aplikaci nalézt přehledy o:

- studijních programů
- dní otevřených dveří
- kolejí a menz UTB
- studentských projektů
- kultury a sportu ve Zlíně

7 CÍLE

Předtím, než začneme programovat, měli bychom si stanovit cíle a požadavky aplikace. Některé specifikace jsou zadané přímo zadavatelem, jiné vyplívají z podstaty aplikace. V následujících kapitolách jsou tyto požadavky roztříděny do kategorií, kde jsou tyto požadavky popsány obecně a pak je popsán vztah vzhledem k aplikaci „Studuj UTB“.

7.1 Specifikace aplikace

Pro grafické zachycení požadavků se v softwarovém inženýrství používají standardizované diagramy, které se označují jako UML diagramy. Pomocí těchto diagramů se zachycují jak požadavky od klienta na aplikaci, tak také business procesy, životnosti objektů a další věci [2].

Pro zachycení specifikací aplikace byl vytvořen UML diagram případů užití. V tomto diagramu jsou zachyceni jednotliví aktéři, kteří budou v rámci aplikace vystupovat. Vlevo jsou umístěni nejčastěji uživatelé aplikace. Uprostřed je definován samotný systém, který musí být ohraničen, aby bylo jasné, co je a co není definováno uvnitř systému. Napravo se většinou umisťují externí systémy, se kterými bude aplikace komunikovat. Uvnitř systému jsou pak definovány tzv. případy užití, což jsou funkce, která bude systém nabízet pro uživatele.

V rámci aplikace „Studuj UTB“ bude figurovat jeden typ uživatele a tím je uchazeč o studiu. Systém poté bude obsahovat různé případy užití. Jejich vazby jsou zachyceny v následujícím diagramu. Systém bude pro svou funkcionalitu využívat dva externí systémy. Prvním je portál STAG, ze kterého budou získávány aktuální informace o studijních programech a oborech. Druhý systém bude sloužit pro odesílání emailů, které budou obsahovat detailní informace o vybraném oboru.

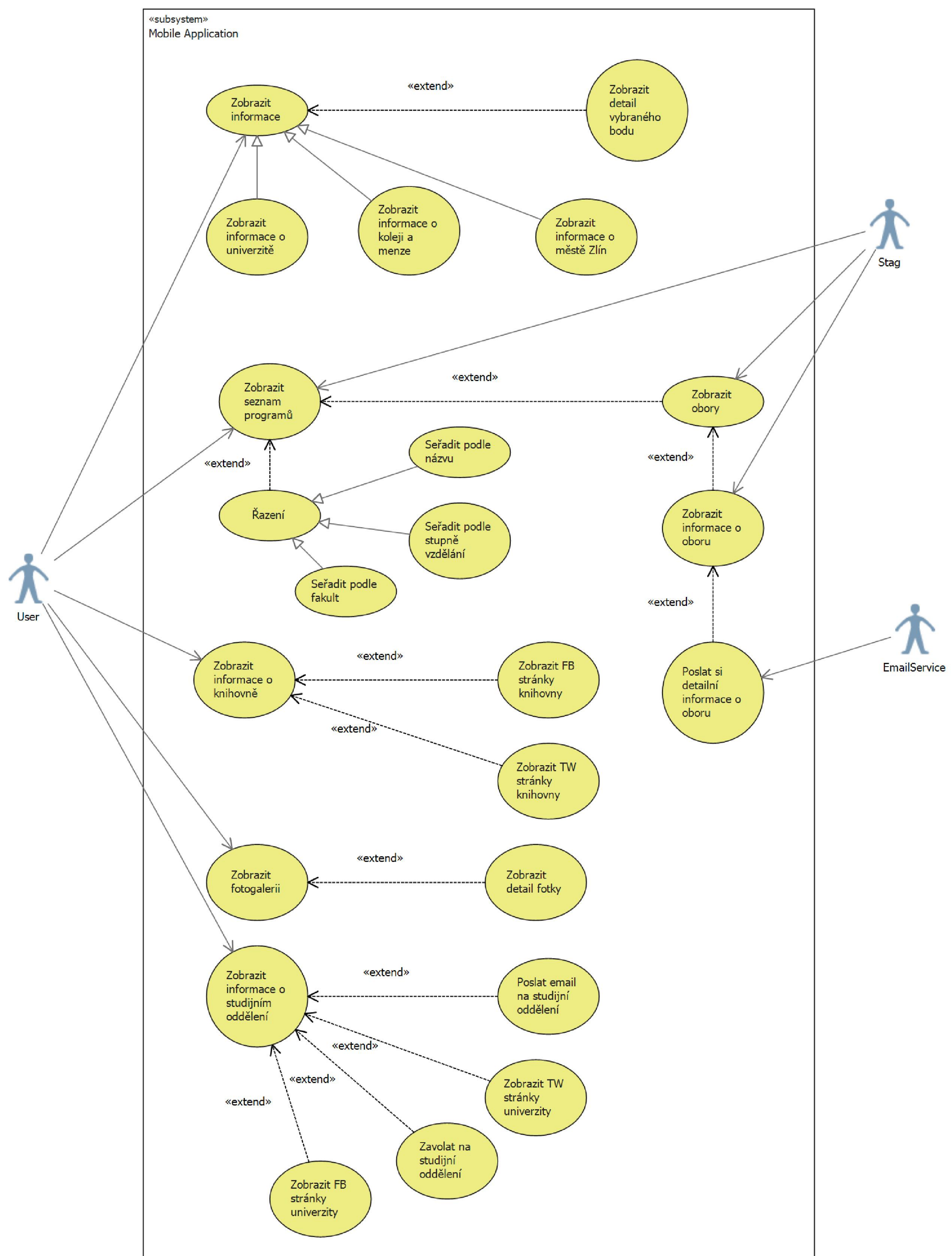


Schéma 15 Diagram případů užití

7.2 Nefunkční požadavky

Nefunkční požadavky je skupina specifikací aplikace, která místo toho, aby definovali to co má aplikace obsahovat, spíše pojednávají o tom, jak by měla být aplikace implementována.

Nejčastější nefunkční požadavky na aplikace jsou bezpečnost, výkon, škálovatelnost. V případě mobilních aplikací obecně platí jiné nefunkční požadavky, než v případě klasických desktopových nebo webových řešení.

V rámci aplikace Studuj UTB byly vytýčeny následující nefunkční požadavky.

- Řídit se podle oficiálních zásad tvorby aplikace pro danou platformu
  - Každá platforma má jiné zásady tvorby aplikace. Nejvíce se tato část projevuje při návrhu uživatelského rozhraní pro danou platformu. Každá platforma má k dispozici jiné prostředky pro realizaci uživatelského rozhraní. Některé jsou společné pro všechny platformy, jiné jsou platformě specifické. Je důležité navrhovat uživatelské rozhraní tak, jak je pro danou platformu zvykem, aby byly uživatelé aplikace schopni naši aplikaci jednoduše a intuitivně používat.
- Vyřešit funkčnost aplikace pro případ nedostupnosti internetu
  - U mobilních aplikací obecně je třeba velmi často řešit problém s nedostupností internetu. Proto musí být aplikace navržena tak, aby se vypořádala úspěšně jak s nedostupností internetu, tak s jeho možnými výpadky v průběhu používání aplikace. V rámci této aplikace je problematika vyřešena tak, že většina obsahu aplikace je statická a nezávislá na internetu. Zbylý obsah se uživateli v případě nedostupnosti internetu nezobrazuje. Zobrazí se hlášení o tom, že daný obsah je dostupný pouze, pokud je telefon připojený k internetu.
- Používat indikátory zaneprázdnění a indikátory stahování dat
  - Pokud aplikace provádí náročné výpočty nebo potřebuje stáhnout data z internetu, je dobré o tom informovat uživatele, aby věděl, že aplikace pracuje. V aplikaci „Studuj UTB" nebudou vykonávány žádné časově náročné výpočty, avšak je třeba zobrazit indikátory stahování dat z internetu, protože data pro seznamy oborů a programů se budou získávat z webových serverů.
- Zpracování vyjímek
  - Aplikace by v případě výskytu chyby neměla zobrazovat uživateli interní hlášení chyb, které jsou pro obyčejné uživatele matoucí. V rámci aplikace „Studuj UTB" byly vytvořeny uživatelsky přívětivá hlášení, která jsou zobrazena, pokud dojde v aplikaci k nějaké chybě.
- Asynchronní zpracování požadavků
  - V rámci mobilních aplikací je nezbytně nutné, aby všechny dlouhotrvající požadavky byly provedeny asynchronně a neblokovaly uživatelské rozhraní. Toto je velký rozdíl oproti desktopovým nebo webovým aplikacím, kde je asynchronní zpracování pouze jako doporučení. V rámci aplikace „Studuj UTB" jsou všechna volání webových služeb prováděna pomocí TAP asynchronního návrhového vzoru popsaného v teoretické části práce.

7.3 Architektonické cíle

V rámci aplikace „Studuj UTB" byl kladen důraz na to, aby byla implementována pomocí moderních technik a metod.

- Dobré user experience a možnost sdílení kódu
  - Pokud chceme, aby měla aplikace dobré user experience, můžeme jako kandidáty na vývojové frameworky vyřadit ty, které se pokoušejí nahradit nativní UI vlastním nenativním UI. Xamarin byl zvolen jako nejvhodnější Framework, protože nabízí nejvyšší možné user experience z dostupných multiplatformních frameworků na trhu a také nabízí možnost sdílení kódu.
- Použití MVVM architektonického vzoru pro návrh aplikace.
  - Návrhový vzor MVVM je nejvhodnější návrhový vzor pro vývoj multiplatformních aplikací pod Xamarinem. Díky nezávislosti mezi ViewModelem a View (z pohledu ViewModelu) se z něj stává ideální vzor pro multiplatformní vývoj.
- Separace kódu do PCL
  - Abychom mohli dosáhnout maximálního možného sdílení kódu, je důležité identifikovat a popsat správně Modely a Viewmodely. Ty pak separujeme do samostatné knihovny typu PCL.
- Redukce „boilerplate" kódu na minimum
  - Pokud máme v rámci aplikace separované vrstvy Model a ViewModel do PCL, můžeme v aplikaci použít Framework MVVMCross, který usnadní vývoj multiplatformní aplikace psané pomocí frameworku Xamarin. MVVMCross také velice výrazně redukuje množství opakujícího se a zbytečného kódu na minimum. Jelikož se jedná o silně modulární Framework, je zde možnost vytvořit si vlastní moduly a ty pak recyklovat ve více projektech.

8 NÁVRH UŽIVATELSKÉHO ROZHRANÍ

Na základě sesbíraných požadavků byl nejprve vytvořen navigační diagram jakožto jednoduchý návrh uživatelského rozhraní. V tomto diagramu jsou zachyceny všechny obrazovky a navigace mezi nimi. Dále je zde nastíněna funkcionalita, která bude na jednotlivých obrazovkách dostupná.

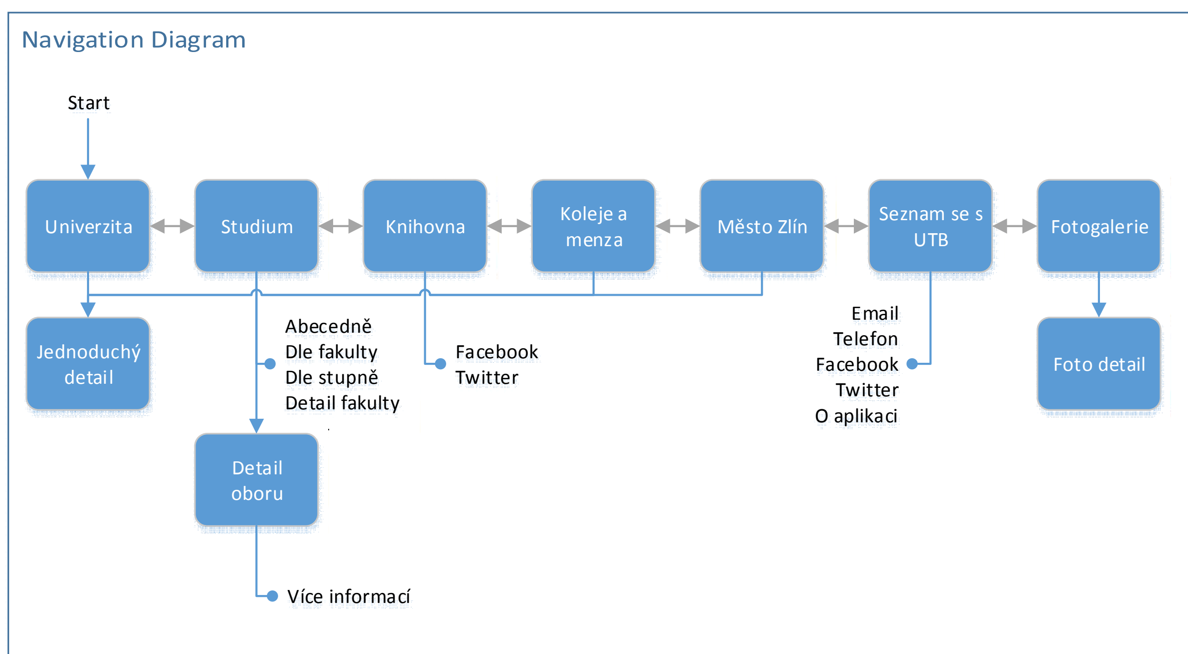


Schéma 16 Navigační diagram

Z diagramu můžeme vidět, že aplikace bude obsahovat 7 hlavních pohledů. Mezi těmito pohledy se bude možné přepínat pomocí gesta „swipe“, pokud to platforma dovolí. Alternativně bude použit záložkový systém (tabs) pro přechod mezi hlavními okny.

Pohledy „univerzita“, „koleje a menza“ a „město Zlín“ budou obsahovat stručný seznam bodů k danému tématu. Po kliknutí na některý z těchto bodů se otevře pohled „jednoduchý detail“, kde se zobrazí podrobnosti k danému bodu.

Pohled „studium“ bude obsahovat seznam oborů. Tento seznam bude možné setřídit podle abecedy, podle fakulty nebo podle stupně vzdělání. Pokud uživatel klepne na logo fakulty, zobrazí se mu informace o dané fakultě. Po klepnutí na obor se otevře nový pohled, kde bude možné vidět detail tohoto oboru. Pokud bude uživatel chtít, bude mít možnost získat více informací formou informačního emailu, který se odešle po stisknutí tlačítka „Více informací“ v pohledu „detail oboru“.

Pohled „knihovna“ bude obsahovat základní informace o knihovně a bude umožnovat přejít na facebookové a twitterové stránky knihovny.

Pohled „seznam se s UTB“ bude obsahovat základní informace o UTB, informace o dnech otevřených dveří, informace o veletrzích, kterých se účastní UTB a kontaktní informace na studijní oddělení. Pokud uživatel klikne na telefonní číslo studijního oddělení, dojde k jeho vytočení. Pokud klikne na emailovou adresu, dojde k zobrazení průvodce odesláním mailu. Dále zde budou tlačítka pro zobrazení facebookové a twitterové stránky university. Pohled také bude obsahovat tlačítko „O Aplikaci“, které zobrazí dialogové okno se stručnými informacemi o verzi aplikace a o jejich tvůrcích.

Poslední pohled „fotogalerie“ bude obsahovat kategoricky seřazené fotografie. Pokud uživatel klepne na některou z fotek, dojde k jejímu zvětšení přes celou obrazovku zařízení.

9 APLIKAČNÍ DIAGRAM

Aplikační diagram se pokouší zachytit složení aplikace „Studuj UTB“. Diagram se pokouší zachytit třídy a kategorizovat je do příslušných vrstev. Diagram je vytvořen pro sdílený projekt Core a neobsahuje třídy z platformních GUI projektů (kromě naznačení vrstvy View).

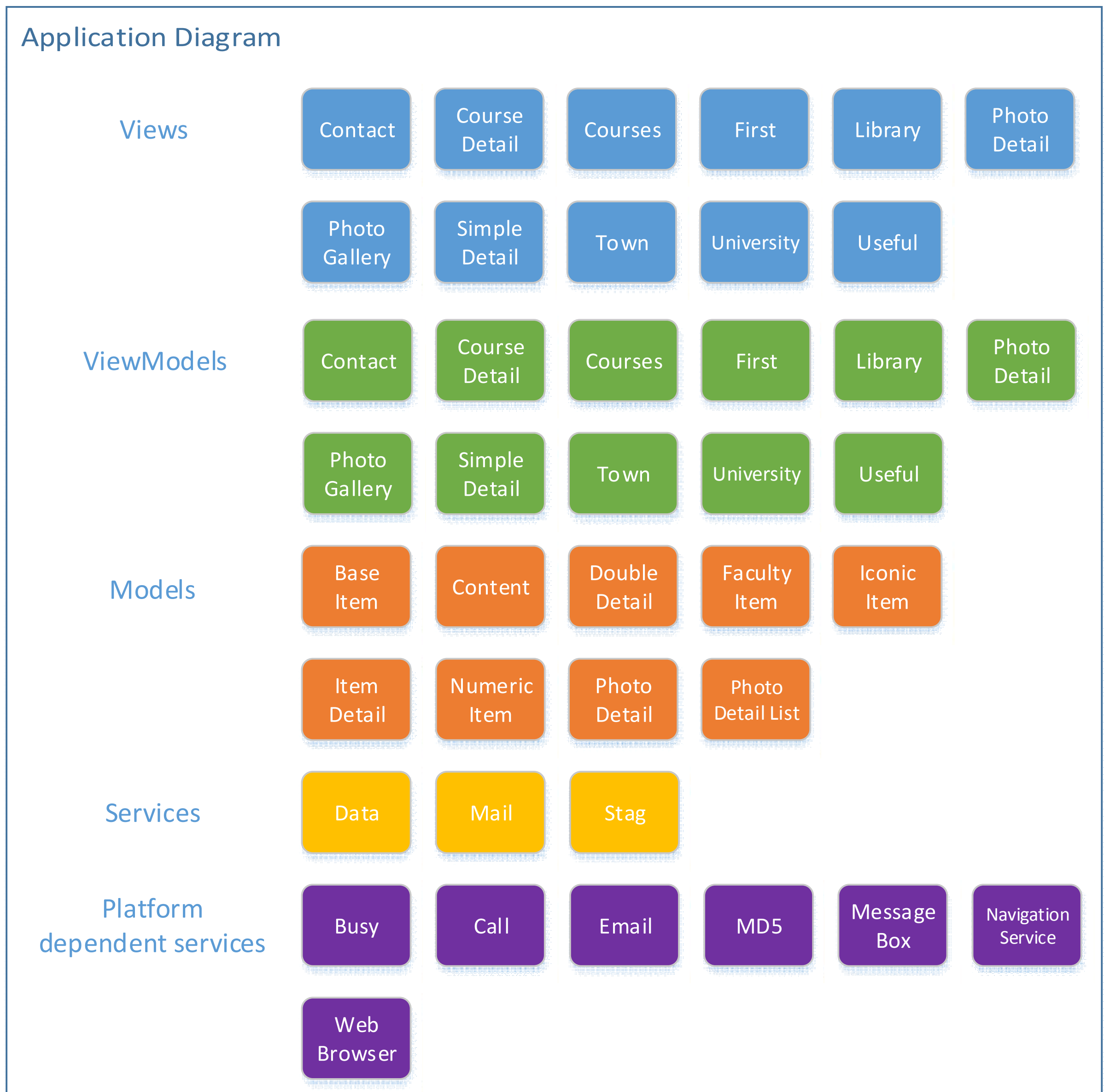


Schéma 17 Aplikační diagram

V diagramu je zachyceno celkem 5 vrstev. Detailní popis jednotlivých vrstev bude uveden v následujících podkapitolách.

9.1 Views

Vrstva označená jako Views obsahuje třídy, které implementují uživatelské rozhraní pro danou platformu. Třídy musí být v ideálním případě 1:1 ku ViewModelům, pokud chceme využít standartní navigační možnosti frameworku MVVMCross. Na každé platformě se definuje vrstva View odlišně.

Na platformě Windows Phone můžeme za tuto vrstvu považovat hlavně soubory typu xaml a jejich code behind soubory. V xaml souborech je definováno uživatelské rozhraní, zatím co v code behind by nemělo být vůbec nic (kromě před generovaného kódu), pokud používáme MVVM architekturu.

Na platformě android definují tuto vrstvu soubory typu axml, které obsahují definice uživatelského rozhraní, a soubory, které implementují třídy odvozené od *Activity*. Třídy odvozené od *Activity* většinou v metodě *OnCreate* nastaví svůj obsah pomocí metody SetContentView na příslušný axml soubor. Také tyto třídy obsahují kód, který se týká implementace uživatelského rozhraní.

Na platformě iOS je možnost definovat Views pomocí dvou způsobů. První způsob je definovat View v xib nebo storyboard souboru a poté k tomuto souboru vytvořit code behind soubor, který bude obsahovat tzv. outlets, což není nic jiného, než reference na prvky uživatelského rozhraní. Druhá varianta je vytvářet celé View jenom pomocí kódu. V rámci aplikace „Studuj UTB“ byla implementována celá vrstva Views pro tuto platformu pouze v kódu.

9.2 ViewModels

Tato vrstva se celá nalézá pouze v Core projektu. Vrstvu tvoří pouze třídy typu ViewModel, které obsahují definice toho, co se má na dané obrazovce objevit (jednoduché property nebo seznamy), nebo jaké akce je možné na dané obrazovce vykonat (Commandy).

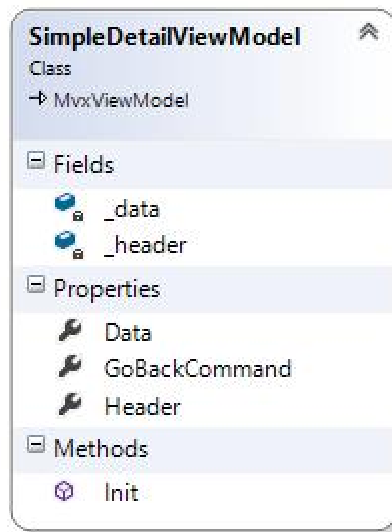


Schéma 18 Diagram třídy pro *SimpleDetailViewModel*

Jako ukázka byla vybrána třída *SimpleDetailViewModel*, která definuje data pro pohled, který se zobrazí po rozkliknutí některého z bodů na obrazovkách „Universita“, „Město Zlín“ nebo „Koleje a menza“

9.3 Models

Tato vrstva se stejně jako předešlá nalézá pouze v Core projektu. V této vrstvě jsou třídy, které popisují strukturu dat.

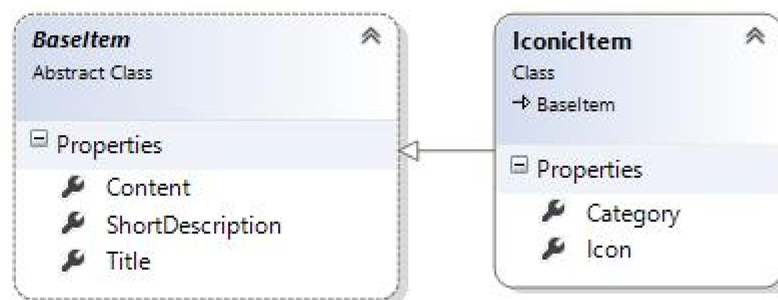


Schéma 19 Diagram třídy pro *IconicItem*

Jako ukázka byla vybrána třída *IconicItem*, která dědí od třídy BaseItem. Třída *IconicItem* popisuje data, která jsou zobrazena v seznamech v pohledu „Koleje a menza“. Jedna instance této třídy odpovídá jednomu řádku v seznamu.

9.4 Services

Tato vrstva je také definována pouze v Core projektu. Obsahuje implementace tříd typu service. Zatím co třídy typu model obsahují převážně strukturu dat a minimum logiky, třídy

typu service obsahují převážně logiku a minimum struktury. Tyto třídy se také používají při sdílení objektů mezi ViewModely.

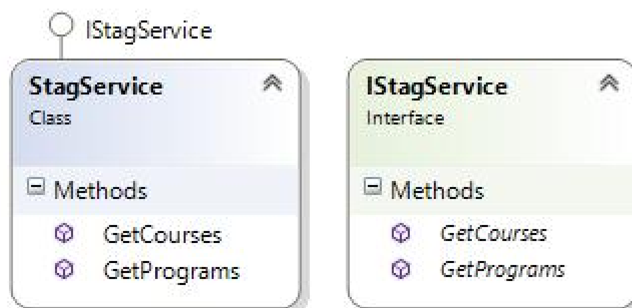


Schéma 20 Diagram třídy pro *StagServices*

Jako ukázka třídy typu service byla vybrána třída *StagService*, která slouží pro získávání dat z externích webových služeb systému STAG. Za povšimnutí stojí, že třída dědí od rozhraní *IStagServices*, které obsahuje ty stejné metody co třida *StagService*. Toto je z důvodu testovatelnosti a zastupitelnosti dané třídy. Při testovaní je možné reálnou třídu *StagService* nahradit třídou *FakeStagService*, která bude implementovat stejné rozhraní a otestovat tak funkcionalitu aplikace i bez připojení k externímu systému STAG. O nahrazení se postará IoC kontejner. Tento princip byl podrobněji popsán v teoretické části práce.

9.5 Platform dependent services

Tato vrstva je implementována ve všech projektech. Obsahuje třídy, které poskytují aplikaci platformě závislou funkcionalitu jako např. volání, gps lokalizace… V projektu Core jsou obsaženy definice rozhraní tak, aby je mohly používat ViewModely. V GUI projektech jsou pak definovány třídy, které implementují tato rozhraní a poskytují je zpět do Core projektu skrze IoC. Tento princip je popsán v teoretické části práce.

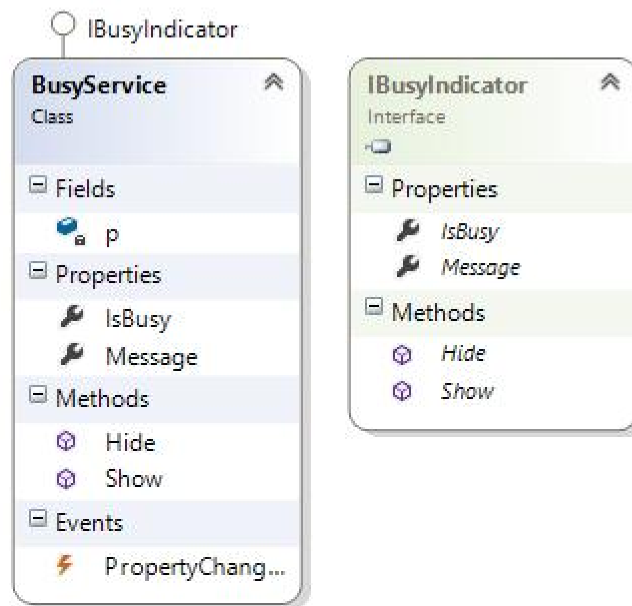


Schéma 21 Diagram třídy pro *BusyService*

Jako ukázka třídy z této vrstvy byla vybrána třída *BusyService*. Diagram zobrazuje její rozhraní a implementaci pro android platformu. Tentokrát je třeba vždy definovat rozhraní a provést implementaci až v platformních GUI projektech. Třída *BusyService* slouží k zobrazení indikátoru činnosti na pozadí při stahování dat z internetu.

10 ROZBOR ARCHITEKTURY

V následujících kapitolách budou rozebrány různé architektonické vlastnosti aplikace „Studuj UTB“.

10.1 Commandy

V rámci MVVM architektury se většinou definují tzv. Commandy. Místo klasických event handlerů v code – behind souborech, které se složitě testovali, se v rámci MVVM převedly tyto akce na Commandy. Command je obecné označení třídy implementující rozhraní *ICommand* [4].

```
public interface ICommand
{
  event EventHandler CanExecuteChanged;
  bool CanExecute(object parameter);
  void Execute(object parameter);
}
```

Kód 30 Rozhraní *ICommand*

Toto rozhraní obsahuje 2 metody a jeden event. Metoda *Execute* obsahuje implementaci toho, co se má provést, když se zavolá tento command. Metoda *CanExecute* obsahuje kód, který vrací informaci o tom, zda může být daný command spuštěn. Event *CanExecuteChanged* vysílá informaci o tom, že došlo ke změně v metodě *CanExecute*.

Toto rozhraní je implementováno na více způsobů. V rámci aplikace je použita implementace z frameworku MVVMCross a to *MvxCommand* (bezparametrická verze) a *MvxCommant*<T> (verze, kde se předává metodě *Execute* parametr typu T). Implementace *MvxCommad* vychází z implementací *DelegateCommand* nebo *RelayCommand*. Jedná se o typ commandů, kde jsou *CanExecute* a *Execute* předávány v konstruktoru třídy jako delegáti.

Pro příklad implementace commandů v rámci aplikace můžeme použít kód z *ContactViewModel*.

```
        public ICommand GoFacebookCommand
        {
            get { return new MvxCommand(() => Mvx.Resolve<IWebBrowserSer-
vice>().ShowPage("https://www.facebook.com/UTBZlin")); }
        }

        public ICommand GoTwitterCommand
        {
            get { return new MvxCommand(() => Mvx.Resolve<IWebBrowserSer-
vice>().ShowPage("https://twitter.com/UniverzitaTBati")); }
        }

        public ICommand EmailCommand
```

```
        {
            get { return new MvxCommand(() => Mvx.Resolve<IMvxComposeEmail-
Task>().ComposeEmail("info@utb.cz", null, null, null, false)); }
        }

        public ICommand CallCommand
        {
            get { return new MvxCommand(() =>
                Mvx.Resolve<IMvxPhoneCallTask>().MakePhone-
Call("UTB","+420576032121")); }
        }
```

Kód 31 Ukázka implementace commandů

V kódu jsou uvedeny 4 ukázky commandů:

- *GoFacebookCommand* – slouží ke spuštění prohlížeče a zobrazení facebookové stránky university
- *GoTwitterCommand* – slouží ke spuštění prohlížeče a zobrazení twitterové stránky university
- *EmailCommand* – slouží ke spuštění průvodce napsáním emailu, který předvyplní adresu university
- *CallCommand* – slouží k inicializaci hovoru na informační oddělení university

Žádný z těchto commandů nemá implementovanou část *CanExecute*, což znamená, že je možné je spouštět kdykoliv.

Výhodou commandů v rámci MVVMCrossu je to, že akce UI vrstvy můžeme implementovat ve sdíleném projektu a použít tento sdílený kód na každé platformě.

10.2 IoC

Pokud chceme vytvářet modulární nebo multiplatformní aplikace, bude nás nejspíše zajímat spojení Depenendency Injection spolu s technikou IoC. Jak již bylo řečeno v teoretické části, základem techniky IoC je tzv. IoC kontejner. Kontejner nám slouží k sestavení částí programu dohromady. U rozsáhlých modulárních systémů ho oceníme zejména proto, že nám umožňuje jednoduše hlídat životnosti objektů. U multiplatformních aplikací oceníme zejména to, že s jeho pomocí můžeme jednoduše používat platformě závislé objekty [27].

V rámci aplikace je využit IoC kontejner dodávaný knihovnou MVVMCross. Kontejner je inicializován po spuštění aplikace a životnost má po celou dobu běhu aplikace. Z pohledu programátora je kontejner zapouzdřen pomocí metod, takže instanci kontejneru nevidí přímo.

MVVMCross kontejner se nazývá *Mvx*. S kontejnerem se nejdříve setkáme v Core projektu.

```
    public class App : Cirrious.MvvmCross.ViewModels.MvxApplication
```

```
    {
        public override void Initialize()
        {
            CreatableTypes()
                .EndingWith("Service")
                .AsInterfaces()
                .RegisterAsLazySingleton();

            CreatableTypes()
                .EndingWith("ViewModel")
                .AsTypes()
                .RegisterAsLazySingleton();

            RegisterAppStart<ViewModels.FirstViewModel>();
        }
    }
```

Kód 32 Plošná registrace do IoC kontejneru

V Core projektu se ve třídě *App* v metodě *Initialize* provede plošná registrace. V rámci aplikace byly plošně zaregistrovány všechny třídy[16] s příponou service. Registrovány byly pomocí svých rozhraní, což znamená, že dané třídy je možné získat pouze přes jejich příslušná rozhraní, nikoli přes jejich typy. Dále je důležité to, že tyto objekty byly registrovány pomocí metody *RegisterAsLazySingleton* což znamená, že objekty jsou registrovány s životností Singleton (po celou dobu běhu programu pouze jedna instance) a vytvářeny jsou až při prvním požadavku (lazy inicializace).

Zajímavostí v aplikaci je, že do IoC kontejneru byly zaregistrovány i všechny ViewModel třídy a to z důvodu jednodušší manipulace s ViewModely při kompozitním zobrazování.

Díky lazy inicializaci servisů (a také lazy inicializaci ViewModelů) můžeme pak v GUI projektech provést registraci platformě závislých tříd.

```
    public class Setup : MvxAndroidSetup
    {
        public Setup(Context applicationContext) : base(applicationContext)
        {

        }

        protected override IMvxApplication CreateApp()
        {
            return new global::UTBShow.Core.App();
        }

        protected override void InitializeFirstChance()
        {
```

[16] Metoda *CreatableTypes*() hledá všechny třídy v daném projektu, které mají public konstruktory a nejsou abstraktní.

```
            DeapExtensions.Binding.Droid.Plugin p = new DeapExtensions.Bin-
ding.Droid.Plugin();
            p.Load();
            Mvx.RegisterSingleton<IBusyIndicator>(new BusyService());
            Mvx.RegisterSingleton<IWebBrowserService>(new WebBrowserMy());
            Mvx.RegisterSingleton<IMessageBox>(new MessageBoxMy());
            Mvx.RegisterSingleton<IMD5>(new MyHash());
            base.InitializeFirstChance();

        }
    }
```

Kód 33 Registrace platformě závislých tříd v Android projektu

```
       public class Setup : MvxTouchSetup
       {
        private UIWindow _window;
        private UIApplicationDelegate _applicationDelegate;
           public static MyPresenter Presenterr;
              public Setup(MvxApplicationDelegate applicationDelegate, UIWindow
window)
            : base(applicationDelegate, window)
              {
                  _window = window;
                  _applicationDelegate = applicationDelegate;
              }

              protected override IMvxApplication CreateApp ()
              {
                     return new Core.App();
              }


        protected override void InitializeFirstChance()
        {
            Mvx.RegisterSingleton<IBusyIndicator>(new BusyService());
            Mvx.RegisterSingleton<IWebBrowserService>(new WebBrowserService());
            Mvx.RegisterSingleton<IMessageBox>(new MyMessageBox());
            Mvx.RegisterSingleton<IMD5>(new MyHash());
            base.InitializeFirstChance();
        }
    }
```

Kód 34 Registrace platformě závislých tříd v iOS projektu

```
    public class Setup : MvxPhoneSetup
    {
        public Setup(PhoneApplicationFrame rootFrame) : base(rootFrame)
        {
        }

        protected override IMvxApplication CreateApp()
        {
            return new Core.App();
        }

        protected override void InitializeFirstChance()
        {
            Mvx.RegisterSingleton<IBusyIndicator>(new BusyService());
```

```
            Mvx.RegisterSingleton<IWebBrowserService>(new WebBrowserService());
            Mvx.RegisterSingleton<IMessageBox>(new MyMessageBox());
            Mvx.RegisterSingleton<IMD5>(new MyHash());
            var builder = new MvxWindowsBindingBuilder();
            builder.DoRegistration();
            base.InitializeFirstChance();
        }
    }
```

Kód 35 Registrace platformě závislých tříd ve WP8 projektu

Na všech třech platformách probíhá registrace ve třídě *Setup*, která dědí od dané platformě závislé třídy *Mvx*Setup*. Samotná registrace je pak provedena v metodě *InitializeFirst-Chance* pomocí metody *RegisterSingleton<T>(instance)*, kde *T* je interface, ke kterému chceme registrovat objekt instance.

Je třeba si uvědomit, že i když se třídy mohou jmenovat stejně, na každé platformě mají jinou implementaci. Jako příklad si můžeme vzít implementace rozhraní *IWebBrowserService*.

```
    public interface IWebBrowserService
    {
        void ShowPage(string www);
    }
```

Kód 36 Definice rozhraní *IWebBrowserService*

```
    public class WebBrowserService:IWebBrowserService
    {
        public void ShowPage(string www)
        {
            WebBrowserTask ta = new WebBrowserTask();
            ta.Uri = new System.Uri(www);
            ta.Show();
        }
    }
```

Kód 37 Implementace rozhraní *IWebBrowserService* na platformě WP8

```
    public class WebBrowserService : IWebBrowserService
    {
        public void ShowPage(string www)
        {
            UIApplication.SharedApplication.OpenUrl(new MonoTouch.Foun-
dation.NSUrl(www));
        }
    }
```

Kód 38 Implementace rozhraní *IWebBrowserService* na platformě iOS

```
    public class WebBrowserMy:IWebBrowserService
    {
        public void ShowPage(string www)
        {
            Intent browserIntent = new Intent(Intent.ActionView, Uri.Parse(www));

            var top = Mvx.Resolve<IMvxAndroidCurrentTopActivity>();
            if (top.Activity == null) return;
```

```
                top.Activity.StartActivity(browserIntent);
            }
        }
```

Kód 39 Implementace rozhraní *IWebBrowserService* na platformě Android

Toto rozhraní slouží k definici kontraktu, který budeme využívat, pokud budeme potřebovat otevřít okno webového prohlížeče na dané platformě. Definici rozhraní provedeme v Core projektu a jednotlivé implementace v GUI projektech. Samotné volání metody pro zobrazení stránky v okně prohlížeče pak vypadá následovně.

```
Mvx.Resolve<IWebBrowserService>().ShowPage("https://twitter.com/UniverzitaT-
Bati"));
```

Kód 40 Ukázka volání objektů z IoC kontejneru

Pomocí *Mvx* se odkážeme na metody MVVMCross IoC kontejneru. Metoda *Resolve<T>* se nám pokusí najít a případně vytvořit instanci třídu, která je registrována k rozhraní *T*.

Metodu *Resolve* většinou voláme v případě, že daný objekt, v jehož kontextu chceme získat referenci na objekt, nebyl vytvořen pomocí IoC kontejneru. Pokud byl daný objekt vytvořen pomocí IoC kontejneru, je vhodnější použít princip Dependency Injection Constructor Injection popsaný v teoretické části práce.

10.3 Navigace

Pokud tvoříme multiplatformní aplikaci, je velice vhodné řešit navigaci v aplikaci unifikovaným způsobem. Jelikož každá platforma implementuje navigaci trochu odlišně, je třeba tento problém vyřešit.

Při použití knihovny MVVMCross můžeme využít její vestavěnou navigaci. Navigace v MVVMCross využívá ViewModelLocatoru a automatických registrací ViewModelů a View podle jmenných konvencí [31].

Jednoduchá navigace bez předávání parametrů může vypadat následovně.

```
        public ICommand DoSimpleNavigation
        {
            get
            {
                return new MvxCommand(() => ShowViewModel<CourseDetailViewMo-
del>());
            }
        }
```

Kód 41 Navigace v kontextu a bez parametrů

Velmi často se stává, že při navigaci potřebujeme předat parametr nebo instanci objektu, které jsou zpracovány na vyžádaném ViewModelu. V rámci architektury MVVM architektury je možné použít 3 základní typy předání parametru při navigaci.

10.3.1 Přímé předání parametru

Framework MVVMCross umožnuje předávat parametry při navigaci přímo. Jedinou podmínkou je, aby předávaný objekt byl serializovatelný. V cílovém ViewModelu se poté tyto objekty získávají v metodě *Init*, kterou přetížíme podle typu a názvu objektu, který si posíláme v navigaci.

```
    public class PhotoDetailViewModel:MvxViewModel
    {

        public void Init(int index,string photolist)
        {
            var json = Mvx.Resolve<IMvxJsonConverter>();
            Items= json.DeserializeObject<PhotoDetailList>(photolist);
            Index = index;

        }
. . .
```

Kód 42 Přímé předání parametru v MVVMCrossu – cílový ViewModel

```
    public class PhotoDetail
    {
        [JsonIgnore]
        [XmlIgnore]
        public PhotoDetailList Parent { get; set; }
        [XmlAttribute]
        public string Title { get; set; }
        [XmlAttribute]
        public string ImageName { get; set; }

        public ICommand GoItemDetailCommand
        {
            get
            {
                return new MvxCommand(() =>
                    {
                        var pos = Parent.IndexOf(this);
                        var json = Mvx.Resolve<IMvxJsonConverter>();
                        var parents = json.SerializeObject(Parent);
                        Mvx.Resolve<INavigationService>().ShowViewModel<PhotoDe-
tailViewModel>(new { index = pos, photolist = parents });
                    });
            }
        }
    }
```

Kód 43 Přímé předání parametru v MVVMCrossu – zdroj navigace

V ukázce je demonstrováno přímé předání parametru při navigaci pokud klepnema na některou z miniatur z fotogalerie (*PhotoGalleryViewModel*). Parametr je definován jako anonymní třída. Zdroj navigace, třída *PhotoDetail*, musí odchytávat navigační kontext přímo z IoC kontejneru, protože nedědí přímo od *MvxViewModel* třídy.

V rámci aplikace „Studuj UTB“ byl použit pouze tento princip navigace.

10.3.2 Sdílená service

Další možnost předávání parametrů v navigaci je využití service, kterou sdílí dané ViewModely. Tento proces umožňuje, na rozdíl od přechozího způsobu, sdílení stejných instancí objektu.

```
public class SharedService
    {
        public string SharedState { get; set; }
    }

    public class OneViewModel:MvxViewModel
    {
        private SharedService SharedService;
        public OneViewModel(SharedService sharedService)
        {
            SharedService = sharedService;
        }

        public ICommand NavigateCommand
        {
            get
            {
                return new MvxCommand(() =>
                {
                    SharedService.SharedState = "from first";
                    ShowViewModel<SecondViewModel>();
                });
            }
        }
    }

    public class SecondViewModel : MvxViewModel
    {
        private SharedService SharedService;
        public SecondViewModel(SharedService sharedService)
        {
            SharedService = sharedService;
        }

        public override void Start()
        {
            if (SharedService.SharedState == "from first")
            {
                //do work
            }
        }
    }
```

Kód 44 Sdílená service v MVVMCrossu

V uvedené ukázce se předává parametr pomocí sdílené service *SharedService*. Tato service musí být zaregistrováná jako singleton. Jednotlivé ViewModely poté dostanou tuto service prostřednictvím Constructor Injection (viz. teoretická část práce).

10.3.3 Zasílání zpráv

Kromě předchozích dvou zmíněných technik, je zde ještě třetí technika předávání parametrů při navigaci. Na rozdíl od předchozích, tato technika je určena pro zpětné zasílání zpráv. Využití této techniky nalezneme především ve scénářích spojených s vnořenými pohledy.

```
    public class OneViewModel : MvxViewModel
    {
        private MvxSubscriptionToken token;
        public OneViewModel(IMvxMessenger messanger)
        {
            token = messanger.Subscribe<MyMessage>(a =>
            {
                /*do work*/
            });
        }

        public ICommand NavigateCommand
        {
            get { return new MvxCommand(() => ShowViewModel<SecondViewModel>());
}
        }
    }

    public class SecondViewModel : MvxViewModel
    {
        private IMvxMessenger messanger;
        public SecondViewModel(IMvxMessenger messanger)
        {
            this.messanger = messanger;
        }

        public ICommand DoneCommand
        {
            get
            {
                return new MvxCommand(() =>
                {
                    messanger.Publish(new MyMessage());
                    Close(this);
                });
            }
        }
    }
```

Kód 45 Zasílání zpráv v MVVMCrossu

V ukázce je demonstrováno použití Messanger pluginu, který je možné doinstalovat pomocí systému Nuget. Při vytvoření *OneViewModel* dojde k registraci pro zprávy typu *MyMessage*.

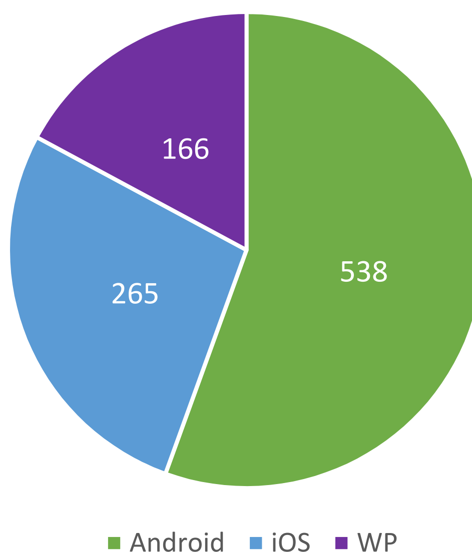
Pokud dojde k navigaci na *SecondViewModel* a zavolání *DoneCommand*, je zaslána zpráva pro předchozí ViewModel a dochází k navigaci zpět pomocí metody *Close(this).*

Tato technika je podrobněji rozepsána v teoretické části práce v kapitole „Publish – subscribe".

10.4 Výsledná aplikace

Pomocí frameworků Xamarin a MVVMCross byla vytvořena aplikace „Studuj UTB". Texty a grafika pro tuto aplikaci byla navržena Martinem Zdražilem z FMK. Samotná implementace aplikace pro tři hlavní mobilní platformy byla provedena autorem této práce.

Aplikace byla velmi pozitivně přijata na všech platformách[17], o čemž svědčí také relativně vysoký počet stažení[18].



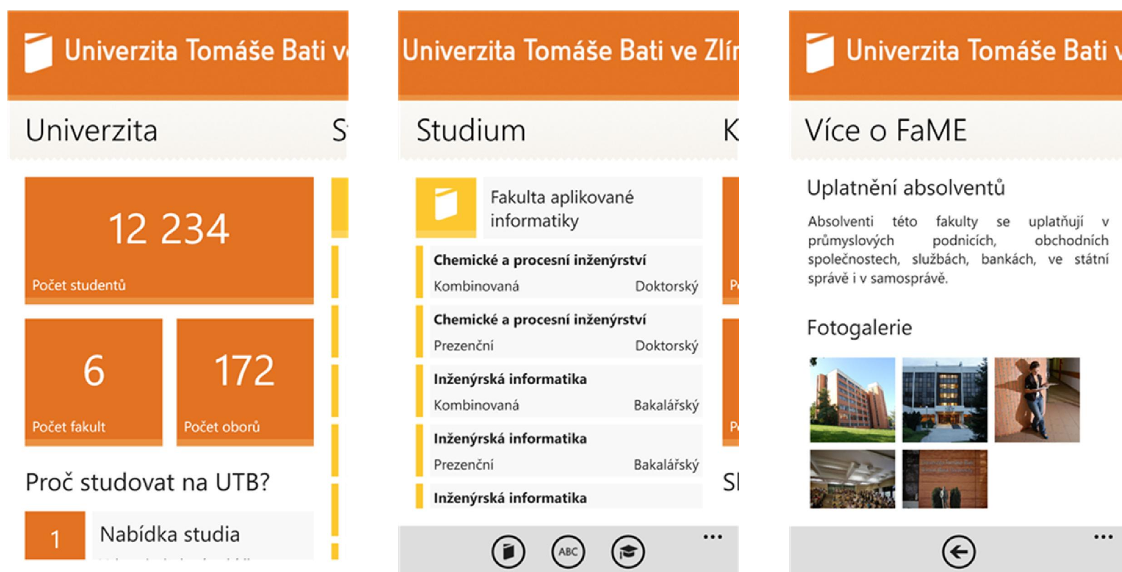
Graf 8 Počty stažení aplikace „Studuj UTB" pro jednotlivé platformy

Výsledná implementace uživatelského rozhraní je demonstrována v následujících odstavcích.

[17] Průměrné hodnocení aplikace pro Android je 4.57/5, pro iOS je 4/4, pro WP 5/5.

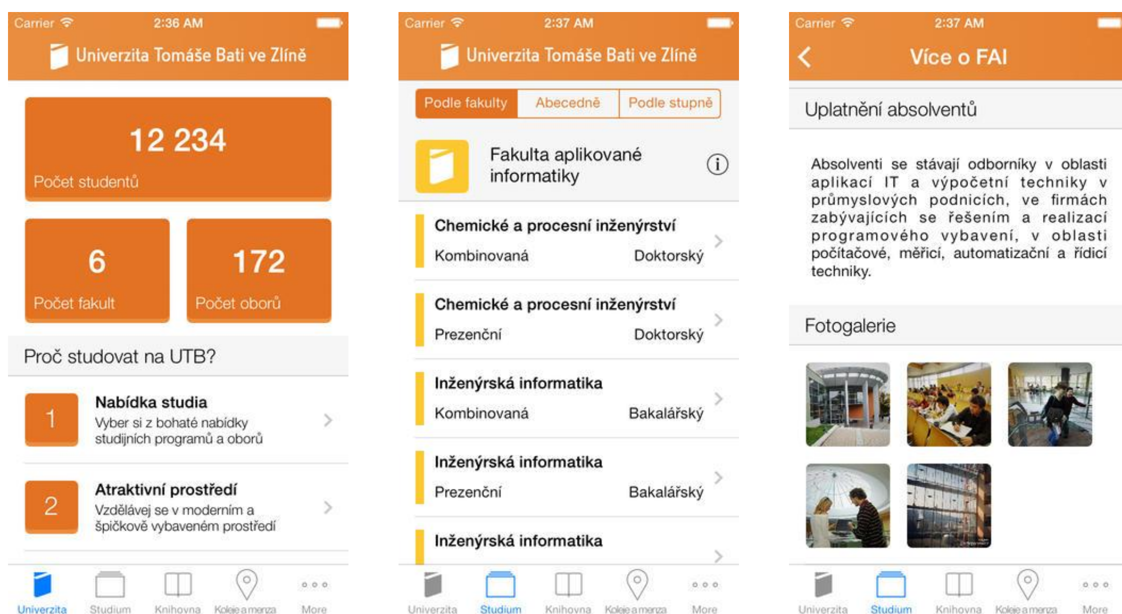
[18] Relativně vysoký počet stažení je brán vzhledem k podobné aplikaci pro platformu Android „Otevřeno na VUT", která má celkově 5 - 10x méně stažení vzhledem ke „Studuj UTB".

Pro platformu WP8 byla navigace uzpůsobena navigačnímu prvku *Pivot*. Ten umožňuje pomocí gesta „swipe“ jednoduše přecházet mezi obrazovkami.



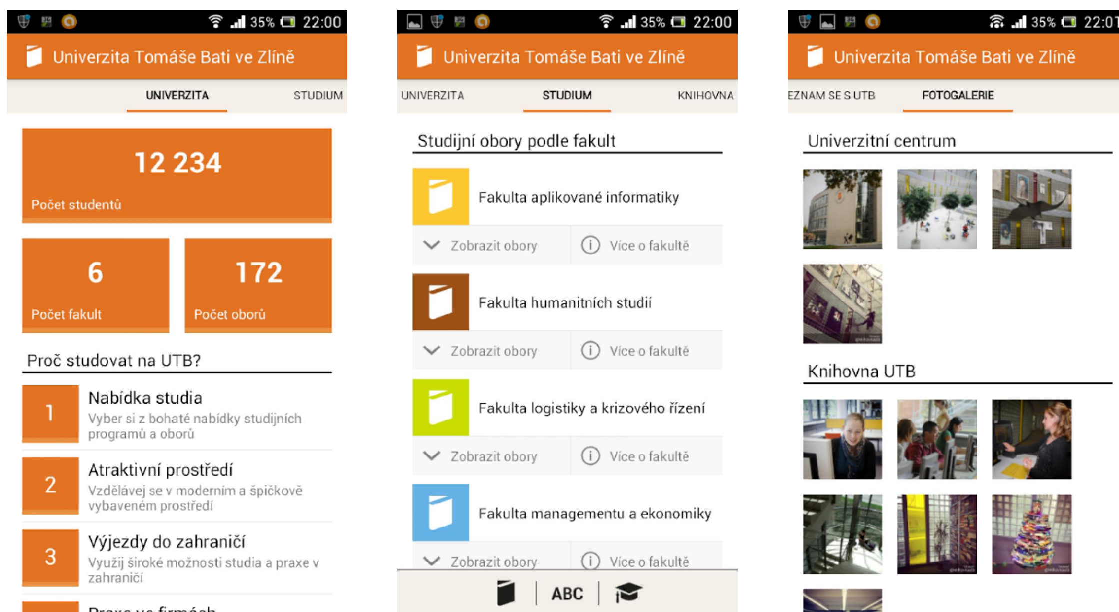
Obrázek 10 Výsledná implementace pro platformu Windows Phone

Pro platformu iOS byla navigace přizpůsobena nativnímu schématu navigace *UITabBar-Controller*. Ten zobrazuje v dolní liště prvních 5 navigačních záložek. Pokud je počet oken větší, jsou agregována do poslední navigační záložky. Navigace pomocí „swipe“ gesta není pro tento prvek dostupná.



Obrázek 11 Výsledná implementace pro platformu iOS

Pro platformu Android byla navigace uzpůsobena pomocí fragmentů a knihovny třetí strany *ViewPagerIndicator*. Tato komponenta umožňuje jednoduše vytvořit záložkový systém navigace s podporou gesta swiper pro přechod mezi stránkami.



Obrázek 12 Výsledná implementace pro platformu Android

ZÁVĚR

V teoretické části jsou nejdříve popsány nejpoužívanější mobilní platformy. Dále jsou vybrány nejčastější druhy multiplatformního vývoje a jsou porovnány jejich výhody a nevýhody. Práce se poté zaměřuje hlavně na popis frameworku Xamarin a usnadnění vývoje na frameworku Xamarin s využitím frameworku MVVMCross. Aby bylo možné dosáhnout dobrých výsledků při sdílení kódu mezi platformami, je důležité, aby byl kód implementován s využitím vhodných návrhových vzorů. Tyto vzory jsou rovněž popsány v teoretické části práce.

V praktické části práce je na případové studii, autorem vytvořené aplikaci, demonstrováno využití Xamarin frameworku společně s MVVMCross frameworkem pro vývoj mobilní aplikace „Studuj UTB“. Nejprve jsou popsány specifikace aplikace. Poté je popsána samotná implementace této aplikace z hlediska její architektury a také z pohledu využití návrhových vzorů.

V rámci případové studie provedené v praktické části práce se podařilo ověřit výhody použití Xamarin a MVVMCross frameworku při tvorbě multiplatformních aplikací. V první řadě se projevilo snížení času potřebného k vytvoření aplikace, dále byla usnadněna práce s aplikačním kódem, protože byl sepsán pomocí jednoho jazyka a jeho velká část byla sdílena napříč všemi platformami. V neposlední řadě se také projevila kvalita výsledné aplikaci, kterou potvrzuje příznivé přijetí uživateli aplikace, protože bylo možné definovat pro každou platformu nativní uživatelské rozhraní.

Hlavní přínos této práce spočívá v identifikaci a popisu nejvhodnějších návrhových vzorů pro vývoj multiplatformních nativních mobilních aplikací vytvořených pomocí frameworku Xamarin.

SEZNAM POUŽITÉ LITERATURY

[1] SHACKLES, Greg. Mobile development with C#. 1st ed. Sebastopol, CA: O'Reilly, c2012, xi, 155 p. ISBN 14-493-2023-6.

[2] GAROFALO, Raffaele. Building enterprise applications with Windows Presentation Foundation and the model view ViewModel Pattern. Sebastopol, Calif: O'Reilly Media, 2011. ISBN 978-073-5650-923.

[3] OLSON, Scott, John HUNTER, Ben HORGEN a Kenny GOERS. Cross-platform Mobile Development in c#. 1st ed. Indianapolis: Wiley Pub., Inc., 2012, p. cm. ISBN 978-1-118-15770-1.

[4] CHRISTIAN, Nagel, Bill EVJEN, Jay GLYNN, Karli WATSON a Morgan SKINNER. Professional C# 2012 and .NET 4.5. New York: Wiley, 2012. ISBN 978-111-8332-122.

[5] FOWLER, Martin. Patterns of enterprise application architecture. Boston: Addison-Wesley, c2003, xxiv, 533 s. The Addison-Wesley Signature Series. ISBN 978-0-321-12742-6.

[6] World Wide Smartphone Sales Share. [online]. [cit. 2014-05-16]. Dostupné z: http://en.wikipedia.org/wiki/File:World_Wide_Smartphone_Sales_Share.png

[7] The Android Source Code. [online]. [cit. 2014-05-16]. Dostupné z: http://source.android.com/source/index.html

[8] Android Dashboard. [online]. [cit. 2014-05-16]. Dostupné z: https://developer.android.com/about/dashboards/index.html

[9] XDA-Developers:Android. [online]. [cit. 2014-05-16]. Dostupné z: http://forum.xda-developers.com/wiki/Android

[10] CyanogenMod. [online]. [cit. 2014-05-16]. Dostupné z: http://wiki.cyanogenmod.org/w/About

[11] Apple support. [online]. [cit. 2014-05-16]. Dostupné z: https://developer.apple.com/support/appstore/

[12] IOS Resolution Quick Reference. [online]. [cit. 2014-05-16]. Dostupné z: http://www.iosres.com/

[13] Mobile Fragmentation. [online]. [cit. 2014-05-16]. Dostupné z: http://gettingstartedwithapps.com/2013/09/19/mobile-fragmentation/

[14] Flyrry statistic. [online]. [cit. 2014-05-16]. Dostupné z: http://www.flurry.com/bid/109749/Apps-Solidify-Leadership-Six-Years-into-the-Mobile-Revolution#.U3VDSSjepME

[15] OpenTable.com. [online]. [cit. 2014-05-16]. Dostupné z: http://media.opentable.com/marketing/founding-farmers-mobile-website.png

[16] Xmamarin Whitepaper. [online]. [cit. 2014-05-16]. Dostupné z: http://cdn1.xamarin.com/webimages/assets/Xamarin_Whitepaper-Key_Strategies_for_Mobile_Excellence.pdf

[17] Xamarin 2.0 vs Appcelerator Titanium vs PhoneGap. [online]. [cit. 2014-05-16]. Dostupné z: http://stackoverflow.com/questions/17249500/xamarin-2-0-vs-appcelerator-titanium-vs-phonegap

[18] Which Cross Platform Mobile Development Platform Should You Choose. [online]. [cit. 2014-05-16]. Dostupné z: http://simpleprogrammer.com/2013/07/01/cross-platform-mobile-development/

[19] Comparing Titanium and PhoneGap. [online]. [cit. 2014-05-16]. Dostupné z: http://kevinwhinnery.com/post/22764624253/comparing-titanium-and-phonegap

[20] Cross-Platform Frameworks. [online]. [cit. 2014-05-16]. Dostupné z: http://blog.mercdev.com/cross-platform-frameworks/

[21] Xamarin. [online]. [cit. 2014-05-16]. Dostupné z: http://xamarin.com/

[22] Mono. [online]. [cit. 2014-05-16]. Dostupné z: http://www.mono-project.com/Main_Page

[23] Mono compatibility. [online]. [cit. 2014-05-16]. Dostupné z: http://www.mono-project.com/Compatibility

[24] Xamarin sharing code options. [online]. [cit. 2014-05-16]. Dostupné z: http://docs.xamarin.com/guides/cross-platform/application_fundamentals/building_cross_platform_applications/sharing_code_options/

[25] Dependency Injection. [online]. [cit. 2014-05-16]. Dostupné z: http://msdn.microsoft.com/en-us/library/ff921152.aspx

[26] Inversion of Control. [online]. [cit. 2014-05-16]. Dostupné z: http://msdn.microsoft.com/en-us/library/ff921087.aspx

[27] IoC Comparsion. [online]. [cit. 2014-05-16]. Dostupné z: http://www.palmmedia.de/blog/2011/8/30/ioc-container-benchmark-performance-comparison

[28] Event aggregator. [online]. [cit. 2014-05-16]. Dostupné z: http://www.codeproject.com/Articles/52810/Event-Aggregator-with-Specialized-Listeners

[29] Asynchronous Programming Patterns. [online]. [cit. 2014-05-16]. Dostupné z: http://msdn.microsoft.com/en-us/library/jj152938%28v=vs.110%29.aspx

[30] MVVMCross scheme. [online]. [cit. 2014-05-16]. Dostupné z: http://qiita.com/amay077/items/c4227663b5a5e540dc13

[31] MVVMCross. [online]. [cit. 2014-05-16]. Dostupné z: https://github.com/MvvmCross/MvvmCross/wiki

[32] Martin Fowler. [online]. [cit. 2014-05-16]. Dostupné z: http://martinfowler.com

[33] Aspiring Craftsman. [online]. [cit. 2014-05-16]. Dostupné z: http://aspiringcraftsman.com/2007/08/25/interactive-application-architecture/

[34] NORTHRUP, Tony a Matthew A STOECKER. MCPD 70-518 exam ref: designing and developing Windows applications using Microsoft.NET Framework 4. Sebastopol, Calif.: O'Reilly Media, 2011, xviii, 316 p. ISBN 07-356-5723-8.

[35] Knockout. [online]. [cit. 2014-05-16]. Dostupné z: http://knockoutjs.com/

[36] Some MVVMCross apps. [online]. [cit. 2014-05-17]. Dostupné z: http://slodge.blogspot.cz/2014/03/some-recent-awesome-mvvmcross-apps.html

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

C#	Programovací jazyk platformy .NET
.NET	Název platformy pro vývoj software
IoC	Inversion of Control
DI	Dependency injection
iOS	Název mobilního operačního systému společnosti Apple
WP 8	Název mobilního operačního systému společnosti Microsoft
MVVM	Model View ViewModel
XA	Xamarin.Android
XI	Xamarin.iOS
PCL	Portable Class Library

SEZNAM OBRÁZKŮ

Obrázek 1 Rozdíl mezi standartní a přizpůsobenou webovou stránkou [15] 18
Obrázek 2 Xamarin Component Store [16] 28
Obrázek 3 Xamarin Studio [21] 31
Obrázek 4 Xamarin šablony pro Visual Studio 32
Obrázek 5 Android toolbar 33
Obrázek 6 iOS toolbar 33
Obrázek 7 Vlastnosti projektu u Xamarin.Android aplikace.....................................34
Obrázek 8 Vlastnosti projektu u Xamarin.iOS aplikace..35
Obrázek 9 Publish-subcribe návrhový vzor [28] 45
Obrázek 10 Výsledná implementace pro platformu Windows Phone 85
Obrázek 11 Výsledná implementace pro platformu iOS 85
Obrázek 12 Výsledná implementace pro platformu Android 86

SEZNAM GRAFŮ

Graf 1 Podíl mobilních operačních systémů [6] 10
Graf 2 Podíl verzí Androidu [8]...11
Graf 3 Fragmentace velikostí a rozlišení obrazovek [8]..12
Graf 4 Fragmentace verzí iOS [11]..13
Graf 5 Fragmentace Windows Phone OS [13] 14
Graf 6 Rozložení času stráveného používáním mobilních zařízení pro USA [14].....15
Graf 7 Vývoj trendu mobilní využívání mobilních webů a mobilních aplikací [14] .16
Graf 8 Počty stažení aplikace „Studuj UTB“ pro jednotlivé platformy 84

SEZNAM SCHÉMAT

Schéma 1 Jazyky a IDE jednotlivých platforem [16] 17
Schéma 2 „Write once, run anywhere“ nebo také „magic box“ [16] 20
Schéma 3 Shrnutí druhů vývoje 21
Schéma 4 Xamarin platforma [16] 22
Schéma 5 Princip přístupu Xamarin platformy 23
Schéma 6 Kompatibilita mono a .NET [23] 25
Schéma 7 Princip kompilace Xamarin.Android [16] 26
Schéma 8 Princip kompilace Xamarin.iOS [16] 27
Schéma 9 Linkování/klonování projektů v Xamarinu [24] 29
Schéma 10 Sdílení kódu pomocí PCL v Xamarinu [24] 30
Schéma 11 Princip Dependency injection [25] 39
Schéma 12 Princip IoC 42
Schéma 13 Synchronní a asynchronní volání 46
Schéma 14 Princip MVVMCross [30] 54
Schéma 15 Diagram případů užití 65
Schéma 16 Navigační diagram 68
Schéma 17 Aplikační diagram 70
Schéma 18 Diagram třídy pro *SimpleDetailViewModel* 72
Schéma 19 Diagram třídy pro *IconicItem* 72
Schéma 20 Diagram třídy pro *StagServices* 73
Schéma 21 Diagram třídy pro *BusyService* 74

SEZNAM KÓDŮ

Kód 1 Standartní API....37
Kód 2 Fluent API....38
Kód 3 Vzorová třída před DI....40
Kód 4 Vzorová implementace constructor DI....41
Kód 5 Vzorová implementace property DI....41
Kód 6 Ukázka IoC kontejneru (TinyIoC)....44
Kód 7 Implementace publish-subcribe vzoru v MVVLight....46
Kód 8 Implementace IAsyncResult....47
Kód 9 Synchronní volání v modelu APM....48
Kód 10 Cyklické dotazování v APM....48
Kód 11 Callback v APM....48
Kód 12 Asynchronní volání synchronní metody v APM....49
Kód 13 Volání v EAP....49
Kód 14 Vzorová implementace EAP....51
Kód 15 Synchronní volání v TAP....51
Kód 16 Cyklické dotazování v TAP....51
Kód 17 Řetězení tasků v TAP....52
Kód 18 Volání asynchronní metody v TAP pomocí await....52
Kód 19 Vzorová implementace TAP....53
Kód 20 Ukázka WPF bindingu....56
Kód 21 Ukázka MVVMCross bindingu pro Xamarin.Android....57
Kód 22 Ukázka MVVMCross bindingu pro Xamarin.iOS....57
Kód 23 Interface pluginu *Accelerometr*....59
Kód 24 Interface pluginu *Email*....59
Kód 25 Interface pluginu *File*....60
Kód 26 Interface pluginu *Location*....60
Kód 27 Interface pluginu *PhoneCall*....61
Kód 28 Interface pluginu *PictureChooser*....61
Kód 29 Interface pluginu *WebBrowser*....61
Kód 30 Rozhraní *ICommand*....75
Kód 31 Ukázka implementace commandů....76
Kód 32 Plošná registrace do IoC kontejneru....77

Kód 33 Registrace platformě závislých tříd v Android projektu 78
Kód 34 Registrace platformě závislých tříd v iOS projektu 78
Kód 35 Registrace platformě závislých tříd ve WP8 projektu 79
Kód 36 Definice rozhraní *IWebBrowserService* 79
Kód 37 Implementace rozhraní *IWebBrowserService* na platformě WP8 79
Kód 38 Implementace rozhraní *IWebBrowserService* na platformě iOS 79
Kód 39 Implementace rozhraní IWebBrowserService na platformě Android 80
Kód 40 Ukázka volání objektů z IoC kontejneru 80
Kód 41 Navigace v kontextu a bez parametrů 80
Kód 42 Přímé předání parametru v MVVMCrossu – cílový ViewModel..................81
Kód 43 Přímé předání parametru v MVVMCrossu – zdroj navigace 81
Kód 44 Sdílená service v MVVMCrossu 83
Kód 45 Zasílání zpráv v MVVMCrossu 83

SEZNAM PŘÍLOH

P I CD - ROM