VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA PODNIKATELSKÁ
FACULTY OF BUSINESS AND MANAGEMENT

ÚSTAV INFORMATIKY
INSTITUTE OF INFORMATICS

MOBILNÍ APLIKACE PRO ADMINISTRACI CMS
MOBILE APPLICATION FOR CMS ADMINISTRATION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. Michal Ingr**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. Petr Dydowicz, Ph.D.**
SUPERVISOR

BRNO 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA PODNIKATELSKÁ

Zadání diplomové práce

Ústav:	Ústav informatiky
Student:	**Bc. Michal Ingr**
Studijní program:	Systémové inženýrství a informatika
Studijní obor:	Informační management
Vedoucí práce:	**Ing. Petr Dydowicz, Ph.D.**
Akademický rok:	2016/17

Ředitel ústavu Vám v souladu se zákonem č. 111/1998 Sb., o vysokých školách ve znění pozdějších předpisů a se Studijním a zkušebním řádem VUT v Brně zadává diplomovou práci s názvem:

Mobilní aplikace pro administraci CMS

Charakteristika problematiky úkolu:

Úvod
Vymezení problému a cíle práce
Teoretická východiska práce
Analýza problému a současné situace
Vlastní návrh řešení, přínos práce
Závěr
Seznam použité literatury

Cíle, kterých má být dosaženo:

Cílem práce je návrh aplikace pro mobilní telefony s platformou Windows 10, která bude umožňovat administraci Kentico CMS prostřednictvím REST rozhraní. Vývoj aplikace bude probíhat technikou „Test–driven development“. V práci bude rovněž diskutován multiplatformní vývoj a ekonomické zhodnocení vývoje aplikace.

Základní literární prameny:

GARGENTA, M. Learning Android. 1. vyd. Sebastopol, Calif.: O'Reilly, c2011. 245 p. ISBN 14-49--9050-1.

LEE, W.,M. Beginning Android application development. Indianapolis, 1. vyd. IN: Wiley Pub., 2011. 428 s. ISBN 978-111-8087-800.

MARTIŠEK, D. Algoritmizace a programování v Delphi. 1. vyd. Brno: Littera, 2007. 230 s. ISBN 978-80-85763-37-9.

UJBÁNYAI, M. Programujeme pro Android. 1. vyd. Praha: Grada, 2012. 187 s. ISBN 978-80-247-3995-3.

VELTE, A., T. VELTE a R. ELSENPETER. Cloud Computing: praktický průvodce. 1. vyd. Brno: Computer Press, 2011. 344 s. ISBN 978-80-251-3333-0.

Termín odevzdání diplomové práce je stanoven časovým plánem akademického roku 2016/17.

V Brně, dne 28. 2. 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA PODNIKATELSKÁ
L. S.

doc. RNDr. Bedřich Půža, CSc.
ředitel

doc. Ing. et Ing. Stanislav Škapa, Ph.D.
děkan

ABSTRAKT

Diplomová práce popisuje návrh designu a vývoj mobilní aplikace pro vzdálenou správu systému Kentico CMS/EMS přes rozhraní REST. Práce klade důraz na agilní přístupy k vývoji, především na metodiku Test-Driven Development a automatické testování.

ABSTRACT

The master's thesis describes the designing and developing mobile application for remote management of Kentico CMS/EMS system via REST interface. The thesis emphasized agile approaches to development, especially Test-Driven Development and automated testing.

KLÍČOVÁ SLOVA

Windows 10 Mobile, Visual Studio, vývoj aplikace, C#, XAML, REST, agilní vývoj, Test-Driven Development, automatické testování, nUnit, MSTest framework, MVVM

KEYWORDS

Windows 10 Mobile, Visual Studio, application development, C#, XAML, REST, agile development, Test-Driven Development, automated testing, nUnit, MSTest framework, MVVM

BIBLIOGRAFICKÁ CITACE

INGR, M. Mobilní aplikace pro administraci CMS. Brno: Vysoké učení technické v Brně, Fakulta podnikatelská, 2017. 75 s. Vedoucí diplomové práce Ing. Petr Dydowicz, Ph.D..

ČESTNÉ PROHLÁŠENÍ

Prohlašuji, že předložená diplomová práce je původní a zpracoval jsem ji samostatně. Prohlašuji, že citace použitých pramenů je úplná, že jsem ve své práci neporušil autorská práva (ve smyslu Zákona č. 121/2000 Sb., o právu autorském a o právech souvisejících s právem autorským).

V Brně dne 22. května 2017 .....

Podpis

PODĚKOVÁNÍ

Chtěl bych poděkovat především svému vedoucímu diplomové práce Ing. Petru Dydowiczovi, Ph.D. za odborné vedení, rady a připomínky při vypracování této práce. Dále bych chtěl poděkovat společnosti Kentico Software s.r.o. a jejím zaměstnancům za poskytnutí odborných konzultací při návrhu aplikace.

OBSAH

ÚVOD

Současné technologické trendy nahrávají mobilním zařízením. Jejich výkon neustále roste, proto postupně začínají zastávat funkci počítačů při komunikaci či běžné kancelářské práci a jejich uživatelé tak mohou pracovat z domu či na cestách.

Na druhé straně jsou však softwarové společnosti, kterým rychle se vyvíjející technologie způsobují nemalé problémy. Z historie známe i firmy (např. Nokia), které vývoji trhu a jeho trendům nepřikládaly vysokou důležitost nebo je zcela ignorovaly, což vyústilo ve ztrátu dominantní pozice na trhu či dokonce ukončení činnosti v dané oblasti.

V této diplomové práci bude představena brněnská, avšak celosvětově známá společnost Kentico Software, která vyvíjí All-in-One řešení pro správu webového obsahu s pokročilými nástroji pro e-commerce, online marketing, personalizaci obsahu a dalšími. Díky tomuto systému běží na celém světě více než 25 tisíc webových prezentací a portálů. I tato společnost však musí udržet krok s dobou a zaměřit své cíle i na stávající zákazníky a podporu jejich loajality.

Práce se proto bude zabývat vývojem aplikace pro mobilní platformu Windows 10, která klientům Kentico umožní spravovat jejich systém prostřednictvím chytrého telefonu. Aplikace bude vyvíjena v rámci programu Inovation Time, který společnost v podobě benefitu nabízí svým zaměstnancům k prohlubování vědomostí a realizaci vlastních projektů v rámci vymezené části pracovní doby.

Při vývoji softwaru je třeba také dbát na aktuální trendy v softwarovém inženýrství. V současnosti je důraz kladen na kvalitu softwaru, s čímž úzce souvisí důkladné testování kódu během vývoje, a to především automatickými testy. V teoretických východiscích práce proto budou popsány tradiční přístupy k vývoji softwaru vč. důvodů, proč je od nich v současné době upouštěno. Představeny budou také agilní přístupy, především metodika Test-Driven Development, při jejíž aplikaci lze dosáhnout téměř stoprocentního pokrytí kódu automatickými testy. Tato metodika bude použita i při samotném návrhu řešení, při níž budou popsány problémy vzniklé jejím nasazením do vývoje a postupy, jak je řešit.

1 VYMEZENÍ PROBLÉMU A CÍLE PRÁCE

V době, kdy téměř každý vlastní a používá chytré mobilní zařízení, se společnosti nabízející své služby pro desktopové počítače snaží do tohoto rychle se rozvíjejícího odvětví vstoupit a investují své prostředky do vývoje mobilních aplikací. Jednou z takových společností je brněnské Kentico Software s.r.o., které svým zákazníkům chce poskytnout mimo svého All-in-One řešení pro správu webového obsahu i mobilní aplikaci, pomocí které bude možné vytvářet a upravovat základní data přímo z chytrého telefonu.

Cílem této diplomové práce je návrh výše zmíněné aplikace pro platformu Windows 10 Mobile, která umožní pohodlnou administraci webového obsahu spravovaného pomocí software Kentico CMS/EMS.

V úvodu práce bude představen mobilní operační systém Windows 10 a nástroje, které jsou běžně používány pro vývoj aplikací pro tuto platformu, včetně programovacího jazyka C# a značkovacího jazyka XAML. Budou také rozebrány metodiky programování, především rozšiřující se agilní přístup, včetně některých moderních technik programování.

V práci bude také analyzován trh s mobilními operačními systémy a jeho vývoj, trh s mobilními aplikacemi a také samotná společnost Kentico Software.

Dále práce bude v praktické části popisovat vývoj aplikace. Ten bude probíhat ve vývojovém prostředí Microsoft Visual Studio a bude při něm kladen důraz na kvalitu aplikace s ohledem na pozdější rozšiřitelnost. Proto bude aplikace vyvíjena pomocí agilní metodiky Test-Driven Development, která svými postupy vede vývojáře k vytváření kvalitního kódu, který je testovatelný a snadno škálovatelný.

V práci bude rovněž diskutován přínos aplikace a její ekonomické zhodnocení.

2 TEORETICKÁ VÝCHODISKA

Předmětem této diplomové práce je vývoj aplikace pro mobilní platformu Windows 10 Mobile, proto nejprve tento operační systém představím včetně jeho důležitých milníků v historii. Také budete seznámeni s programovacím jazykem a vývojovým prostředím z dílny Microsoftu, ve kterých bude aplikace vyvinuta. Dále nastíním obvyklé problémy spojené s vývojem tradičními přístupy a jak tyto problémy může eliminovat agilní vývoj. Podrobněji pak rozeberu metodiku zvanou Test-Driven Development, kterou se budu snažit při vývoji aplikace práce používat.

2.1 WINDOWS 10 MOBILE

Operační systém Windows pro mobilní zařízení byl představen již v roce 1996, tehdy se však jednalo o takzvané kapesní počítače, které se podstatně lišily od současných chytrých telefonů. První verze mobilního operačního systému byla nazvána Windows Pocket PC 2000, který, jak už název napovídá, byl na trh uveden na přelomu tisíciletí. Vizuálně se podobal tehdejším Windows 98 a obsahoval upravené verze známých aplikací Microsoft Office, Internet Explorer a Windows Media Player. Prvním operačním systémem Windows pro mobilní telefony byl však až o dva roky později vydaný Pocket PC 2002 Phone Edition, který již podporoval funkce jako volání a posílání SMS [1].

V roce 2003 došlo k přejmenování platformy na Windows Mobile, nejprve verzi 2003, později 2003 Second Edition. Systém byl nabízen ve třech verzích – pro mobilní zařízení s telefonním modulem i bez něj a pro telefony bez dotykového displeje. V roce 2005 došlo ke změně značení verzí, nově byl totiž představen Windows Mobile 5, který přinesl podporu flash pamětí. Do té doby používaly zařízení pouze paměť RAM a při úplném vybití baterie se ztratila všechna data, která nebyla uložena na externích paměťových kartách. Poslední, značně vzhledově upravená verze 6 pak vyšla v roce 2007 [1].

Do té doby byly mobilní operační systémy od Microsoftu otevřené a výrobci proto systém hojně modifikovali pro svá zařízení, čímž se však značně komplikovala práce vývojářům. S příchodem iOS od Apple začal Microsoft ztrácet podíl na trhu, a proto se rozhodl vyvinout zcela nový systém Windows Phone. Jeho vývoj však trval přiliž dlouho a první verze Windows Phone 7 byla představena až v roce 2010, kdy na trhu již dominovaly iOS a Google se svým systémem Android [2].

V novém Windows Phone bylo kompletně přepracováno uživatelské prostředí a poprvé se uživatelé setkávají s dynamickými dlaždicemi, které nabízí rychlý přehled stavu aplikací, aniž by musely být aplikace otevřeny. Počet aplikací byl však velký problém, protože na novém systému nebylo možné instalovat aplikace pro předchozí Windows Mobile. Vývojáři tedy museli své aplikace vyvíjet pro tento systém znovu [2].

V roce 2011 přichází Windows Phone 7.5, který přidává několik stovek nových funkcí, lokalizaci do češtiny a opravuje mezi uživateli nejčastěji diskutované nedostatky. O rok později je na trh uveden Windows Phone 8, který však způsobil další komplikace vývojářům, kteří k vývoji svých aplikací použili XNA Framework, jehož podpora byla v této verzi systému ukončena. Windows Phone se začíná velmi podobat desktopovým Windows 8, tedy jeho Modern UI. Aplikace pro mobilní telefony se daly jednoduše portovat do desktopových (nebo tabletových) verzí, a proto se v Microsoftu zrodila myšlenka jednotného systému pro všechny zařízení. V počítačích a noteboocích by byl kompletní systém, mobilní telefony, tablety anebo herní konzole Xbox by však obsahovaly stejný systém, jen ořezaný o některé části [2] [3].

Windows Phone se další větší aktualizace na verzi 8.1 dočkává až v roce 2014, kdy se Microsoft zaměřil na vylepšení funkcí na základě zpětné vazby od uživatelů. V té době je však vize jednotného systému zahrnuta do strategie Microsoftu, a proto v roce 2016 uvádí systém Windows 10 Mobile. Ten se v současné době již téměř neliší od moderního uživatelského rozhraní desktopové verze a je zřejmé, že rozdíly se budou dále vytrácet, až půjde opravdu o jeden systém [3] [4].

Stejnému sloučení se dostává i u aplikací – pro systém Windows 10 lze vyvíjet takzvané univerzální aplikace (Universal Apps), které lze spouštět jak na počítačích, tak na tabletech, mobilních telefonech, a dokonce i dalších zařízeních jako jsou Xbox, Surface Hub nebo HoloLens. Jednotná platforma (Universal Windows Platform, zkráceně UWP) pak nabízí jednotné API a vývojáři se tak nemusí starat o to, na jakém zařízení aplikace poběží, pouze přizpůsobí vzhled a rozložení aplikace pro různé velikosti displejů, tedy takzvané adaptivní uživatelské rozhraní. Odměnou pro vývojáře je mnohem větší základna potenciálních uživatelů jejich aplikací. Výhodou je také mnohem efektivnější udržovatelnost kódu, protože pro jednotný systém je potřeba pouze jedna univerzální

aplikace a není tedy třeba upravovat zvlášť aplikaci pro počítače a zvlášť pro mobilní zařízení [5] [6].

Obrázek 1: Universální Windows platforma

Zdroj: Převzato z [6]

2.2 PROGRAMOVACÍ JAZYK C#

Kvalita, stabilita a rychlost aplikace mnohdy závisí na zvoleném programovacím jazyku. Oblíbeným a kvalitním jazykem pro tvorbu malých, ale i obsáhlých a robustních aplikací pro Windows je C#, který vyvinul, stejně jako operační systém, Microsoft [7].

Vzniku jazyka C# ovšem předcházelo představení platformy .NET. Tato platforma byla poprvé zmíněna Microsoftem roku 1999 a měla umožňovat částem programu či komponentám, které byly psané v různých programovacích jazycích, aby spolu vzájemně komunikovaly. Platforma .NET se skládá z CLR (Common Language Runtime), který překládá spravovaný kód IL (Intermediate Language) do strojového kódu typického pro daný hardware. IL, někdy nazývaný jako mezijazyk, je prostředník mezi programovacími jazyky a strojový kódem. To umožňuje, aby program, jehož komponenty jsou psané v různých jazycích, spolu komunikovaly. Kód v programovacím jazyku se totiž přeloží právě do IL a následně CLR až za běhu překládá potřebný spravovaný kód v IL do strojového kódu. Vize této platformy byla oficiálně představena v roce 26. června 2000 a o čtyři dny později Microsoft ohlásil uvedení jazyka C#, který byl na platformu .NET šitý na míru a dokázal ji efektivně využít [5] [7].

Syntaxe C# vychází z jazyku Java a C++, největším rozdílem však je, že u C# není potřeba spravovat paměťové prostředky. O alokaci a uvolňování paměti se velmi efektivně stará Garbage Collector (GC), který je součástí CLR platformy .NET. Na rozdíl od C++ také

není potřeba při vývoji optimalizovat kód pro různá prostředí (např. 32bitová vs. 64bitová platforma), ve kterých se předpokládá běh vyvíjené aplikace. Kompilovaný kód v C# je překládán do strojového až na daném stroji (Just In Time) a o optimalizaci se tak stará přímo CLR. Kvůli těmto výhodám a podobné syntaxi starších jazyků se C# stal velmi rychle oblíbený a hojně využívaný jazyk [7].

Od svého vzniku dostal jazyk již několik aktualizací, kdy byly postupně doplněny funkce a nástroje, které programování v tomto jazyku činí ještě snazším. Příkladem mohou být anonymní metody, genericita nebo částečné typy ve verzi 2.0., implicitně typované lokální proměnné, anonymní typy, automatická implementace vlastností, částečné metody, a především podpora integrovaného dotazovacího jazyka LINQ ve verzi 3.0. Ve verzi 4.0 přibyly volitelné a pojmenované parametry metod a dynamicky typované objekty. Současná verze C# 5.0 vydaná v roce 2012 se zaměřuje především na práci s externími datovými zdroji a zavádí proto podporu asynchronního programování [7].

Programovací jazyk C# je standardizovaný, silně typovaný, objektově orientovaný jazyk, ECMA (standardizační komise) jej definuje jako jednoduchý, moderní, mnohoúčelový jazyk, který vyniká svojí robustností, trvanlivostí a programátorskou produktivitou. Naproti tomu se může zdát, že jazyk bude méně výkonný z důvodu překládání řízeného kódu do strojového až v momentě, kdy je daný kus kódu potřeba, tzv. Just In Time (JIT). Avšak opak je být pravdou, překládání JIT částí kódu, které jsou potřeba, zajišťují menší nároky na operační paměť, a navíc se kód překládá pouze jednou. Pokud je tedy potřeba část kódu, která již byla přeložena, načítá se z paměti. Některé části kódu se dokonce nemusí kompilovat vůbec, pokud např. danou komponentu uživatel aplikace při jejím běhu nevyužívá. Za zmínku stojí také, že jazyk C# je case sensitive, tedy rozlišuje velikost písmen [7].

2.3 ZNAČKOVACÍ JAZYK XAML

Programovací jazyk C# slouží k programování logické části aplikace, k vytváření prezenční vrstvy se však často používá značkovací jazyk XAML (Extensible Application Markup Language). Tento jazyk, který byl vyvinut také společností Microsoft, se hojně využívá při tvorbě aplikací založených na Windows Presentantion Fundation, Silverlight či UWP. XAML je založený na standardizovaném jazyku XML (Extensible Markup

Language), který je velmi rozšířený především při přenášení dat mezi aplikacemi. Z tohoto důvodu je i práce s XAML pro značnou část vývojářů jednoduchá a intuitivní. Veškeré části XAML kódu lze vytvořit i pomocí jazyků C# nebo VisualBasic, ale návrh uživatelského prostředí aplikace v XAML je mnohem snazší a přehlednější [8] [9].

2.4 ROZHRANÍ REST A OBJEKTOVÝ ZÁPIS JSON

V současné době téměř každá aplikace komunikuje s nějakým serverem, jinak tomu nebude ani u navrhované aplikace pro správu systému Kentico. K vzájemné komunikaci dochází posíláním požadavků směrem k serveru a odpovědí směrem ke klientovi. Data v nich musí mít určitou strukturu, aby klient i server těmto datům rozuměl. K tomuto účelu byl vytvořen jazyk XML, který byl zmíněn v předchozí kapitole, v současné době se ovšem stále více používá zápis pomocí JSON. Data přenášená pomocí tohoto zápisu jsou nezávislá na počítačové platformě, oproti XML odlehčená a lépe čitelná pro programátora a umí pojmout pole hodnot, celé objekty i jednotlivé hodnoty. Oba tyto zápisy objektů, i další méně používané, lze použít při komunikaci klient-server pomocí rozhraní REST (Representational State Transfer) [8] [10] [11] [12].

REST je datově orientovaná architektura, přenášená data tak reprezentují aktuální stav objektu, tedy data, který tento objekt popisují. Komunikace probíhá pomocí jednoduchých HTTP volání GET, POST, PUT a DELETE. Jak názvy napovídají, prostřednictvím těchto metod lze data přijímat, vkládat, aktualizovat nebo mazat. U všech metod je potřeba znát k přístupu k objektu jeho identifikátor, jedinou výjimkou je metoda PUT, kdy je vytvořen nový objekt na straně klienta a jeho identifikátor je mu přidělen až serverem [12] [13].

Zjednodušeně lze průběh komunikace např. u metody GET popsat následovně:

1. Klient vyšle požadavek GET s URI požadovaného objektu či kolekci objektů
2. Server odešle objekt převedený (serializovaný) do XML, JSON
3. Klient obdrží data objektu a pokusí se je převést (deserializovat) na zpět na objekt

Toto zjednodušené schéma používají i metody POST a PUT s rozdílem, že serializované objekty nyní odesílá klient na server. U metody DELETE se objekty přenášet nemusejí, stačí vyslat HTTP požadavek na URI konkrétního objektu, který má být smazán [12] [13].

Navrhovaná aplikace bude se serverem, na kterém běží systém Kentico CMS, komunikovat právě přes rozhraní REST všemi zmíněnými metodami.

2.5 VÝVOJOVÉ PROSTŘEDÍ VISUAL STUDIO

K vývoji aplikací pro Windows je bezesporu nejvíce vhodné použít vývojové prostředí společnosti, která vyvinula i jazyky k tvorbě aplikací pro tuto platformu, tedy Microsoft Visual Studio. Mezi hlavní výhody patří jistě pokročilá funkce IntelliSense, která pomáhá při psaní kódu automatickým doplňováním částí kódů, navrhováním, opravováním či kontrolou syntaxe. Snadné ladění a diagnostika aplikací s možností změnou některých částí kódu za běhu aplikace v ladícím režimu lze považovat také za nespornou výhodu, zejména u návrhu uživatelského prostředí. Součástí Visual Studia jsou také testovací nástroje, které umožňují spouštění automatických testů. Nechybí ani emulátory mobilních zařízení či návrhové prostřední Blend for Visual Studio určené pro návrh prezenční vrstvy aplikace v jazyku XAML tak, aby vývojář musel psát co nejméně kódu. Vývojové prostředí lze dále rozšířit o mnoho doplňků, nástrojů a frameworků. Toto prostředí nabité pokročilými vývojářskými funkcemi je navíc nabízeno zdarma pro vývojáře jednotlivce, postačí k tomu pouze registrovaný vývojářský účet u Microsoftu [14].

Visual Studio podporuje také správu kódu pomocí lokálních Git repositářů s možností synchronizace např. s oblíbeným Github či Visual Studio Team Services. Oba tyto repositáře nabízejí prostor pro uchovávání různých verzí kódu včetně informací o provedených změnách s možností spolupráce s více vývojáři, komentování či vrácení změn apod. Visual Studio Team Services na rozdíl od Github dovoluje vytvářet privátní repositáře i v neplacené verzi a přidává navíc i podporu pro agilní metodu vývoje SCRUM, která bude popsána později [14] [15].

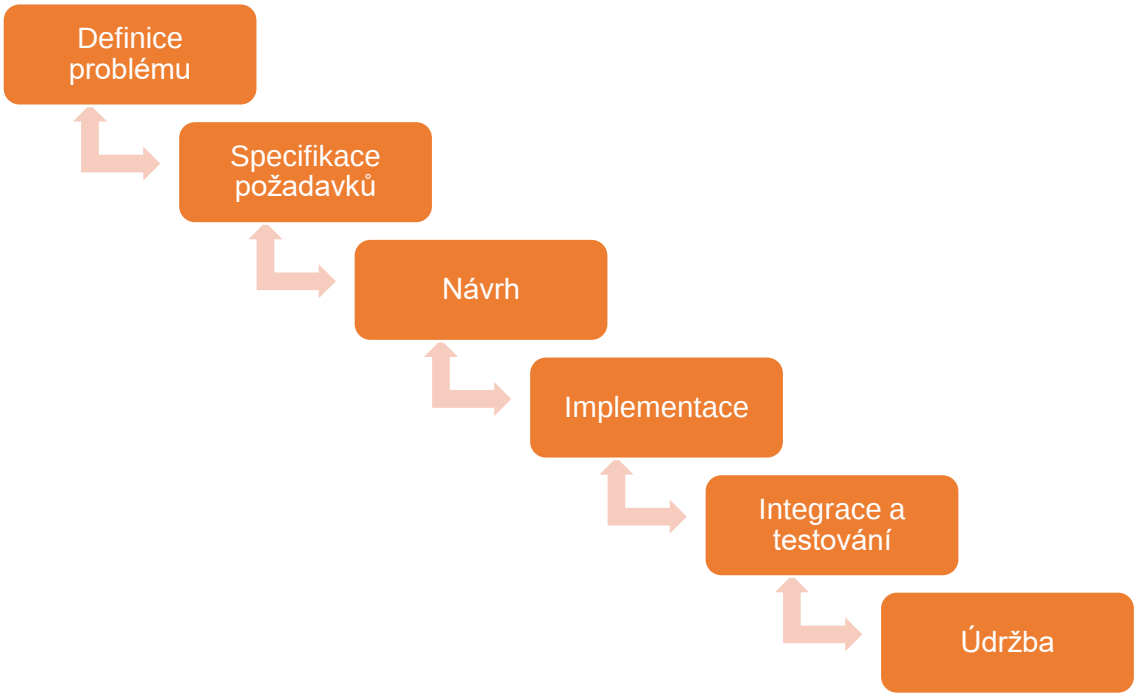
2.6 TRADIČNÍ PŘÍSTUP VÝVOJE

Při vývoji softwaru se často vývojáři nebo i celé společnosti setkávají s řadou různých problémů, jako je dlouhý vývoj, změna požadavků v průběhu vývoje, nemožnost či nesnadnost údržby a rozšiřování softwaru, nízká kvalita, vysoké náklady na vývoj apod. V důsledku těchto problémů vznikla na konci šedesátých let softwarová krize, kdy neefektivita při vývoji často způsobovala velké prodražení projektů, které mělo za

následek malé zisky či dokonce ztrátu. Do posledního okamžiku dokonce nebylo jisté, zda se vývoj vůbec podaří dokončit [16].

Hlavními problémy, které se značnou měrou podílely na softwarové krizi, byly především špatná komunikace (se zákazníky, ale i napříč společností), nesprávný přístup (obecně platilo, že programátor má vždy pravdu a zájmy a požadavky zákazníků byly spíše v pozadí), špatné odhady (času, ceny, rozsahu apod.), špatné plánování, podcenění hrozeb a rizik, a v neposlední řadě, již zmíněná, nízká produktivita práce. V sedmdesátých letech se proto začínají používat techniky nově vznikajícího softwarového inženýrství, např. návrhové vzory, testování, zajištění kvality, modely životního cyklu vývoje apod. [16].

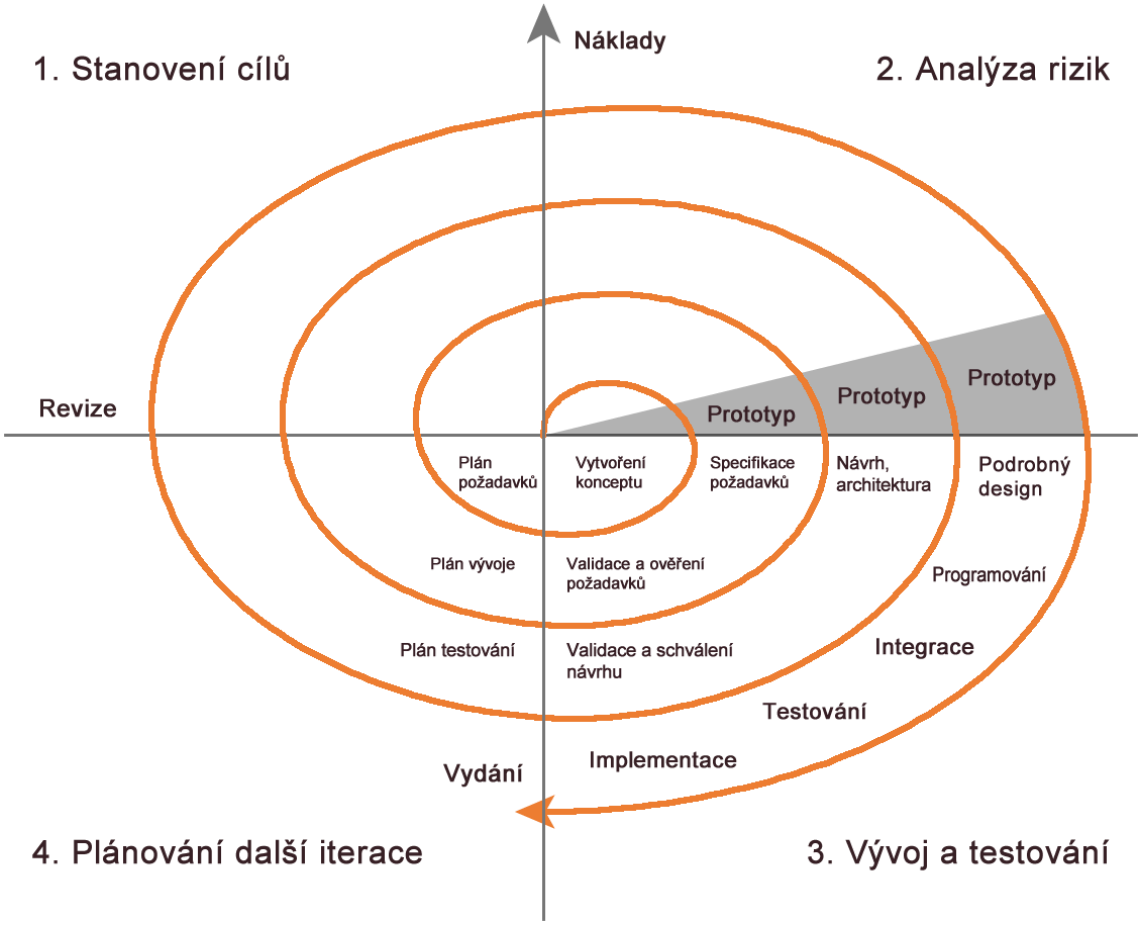
Asi nejznámějším modelem životního cyklu vývoje je vodopádový model z roku 1970, který se však v současné době považuje již za zastaralý a většina softwarových firem od něj upouští. Hlavními nevýhodami jsou nepružnost vývoje a komunikace se zákazníkem probíhá většinou před vývojem a poté až po ukončení vývoje. Problém pak vzniká tehdy, kdy se změní zákazníkovy požadavky a na konci dlouhého vývoje je zákazníkovi předán produkt, který neodpovídá jeho aktuálním požadavkům [16].



Obrázek 2: Příklad definice vodopádového modelu vývoje

Zdroj: Vlastní zpracování dle [16]

Některé z těchto problémů se snaží eliminovat spirálový model životního cyklu z roku 1985, ovšem i tento riziky řízený přístup má několik nevýhod. Jedněmi z hlavních jsou, že produkt se zákazníkovi předává až po skončení posledního cyklu a během jednotlivých cyklů nelze provádět změny v požadavcích (pouze mezi jednotlivými iteracemi cyklu). Navíc je nutná přítomnost expertních analytiků, kteří dokáží analyzovat rizika a roztřídit je na rizika klíčová a nepodstatná. Komplikovanost a zvýšená byrokracie modelu mohou být považovány také za nevýhody [16].



Obrázek 3: Spirálový model vývoje

Zdroj: Vlastní zpracování dle [16]

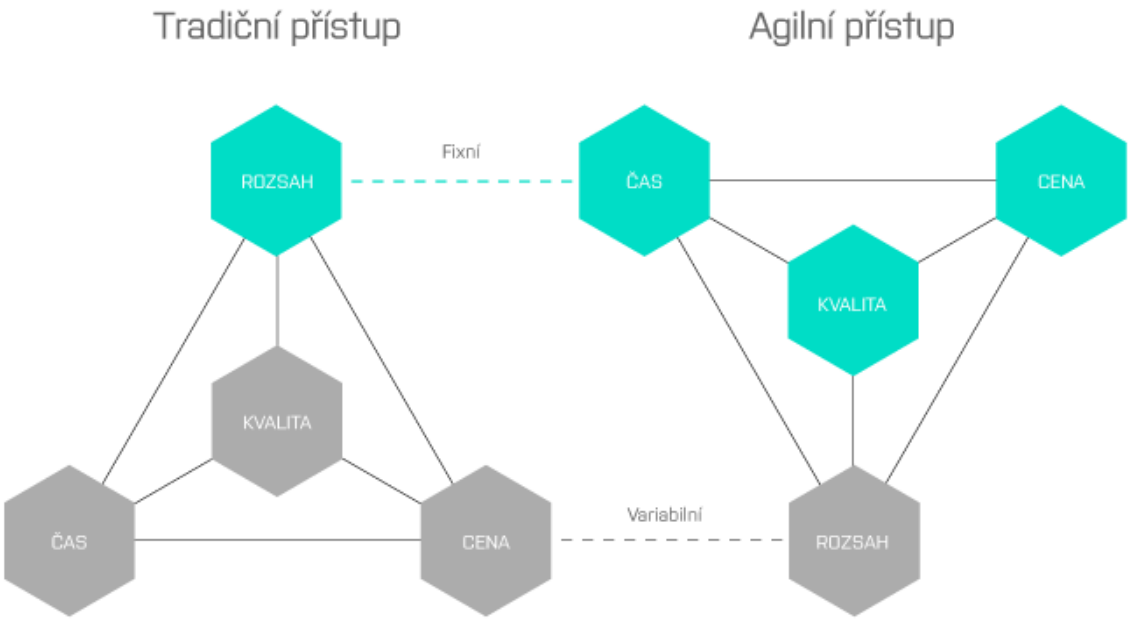
Ze spirálového modelu vývoje vzešlo i několik dalších odvozenin, např. Rational Unified Process či rozšířený spirálový model Win-Win, které představují velký pokrok v softwarovém inženýrství, v současnosti jsou však spirálové modely spíše na ústupu z důvodu nedostatečné pružnosti, která je v současné době rychle se rozvíjejících trzích se softwarem nedostačující [16].

2.7 AGILNÍ METODIKY VÝVOJE

Tradičních přístupů k vývoji existuje celá řada, avšak popisování jich je nad rámec této práce, tradiční metody byly uvedeny pouze z toho důvodu, aby bylo možné s nimi porovnat agilní metodiky [16].

Agilní vývoj jsou metodiky jsou inovativní přístupy k vývoji softwaru, které zvyšují efektivitu a rychlost vývoje, kvalitu softwaru a schopnost uspokojit rychle se měnící požadavky uživatelů či zákazníků. Vychází z tzv. Agilního manifestu, který byl sepsán jako reakce na měnící se trh se softwarem. Dříve bylo vývojářů velmi málo, zákazníci tak byli ochotni na software čekat dlouhou dobu a zaplatit nemalé peníze. V dnešní době je však situace jiná, poptávka po softwaru roste, a ještě rychleji roste počet zdatných vývojářů. Ti se snaží odlišit se od konkurence, a proto rychlost a cena začíná hrát při budování důležitou roli. A právě ke snížení nákladů a zvýšení efektivity (a tím rychlosti) a kvality (tj. snížení počtu později opravovaných chyb) přispívají velkou měrou agilní metodiky [16].

Jednoduché srovnání tradičních a agilních přístupů lze nalézt na následujícím obrázku.



Obrázek 4: Srovnání tradičního a agilního přístupu k vývoji

Zdroj: Převzato z [17]

Pokud se zaměříme na tradiční přístup, je vidět, že rozsah projektu je považován za fixní, cena, čas a kvalita za variabilní, tzn. že v tradičním přístupu se za každou cenu a za jakýkoli čas musí celý rozsah (čili požadavky stanovené na začátku vývoje) splnit. S tím souvisí i kvalita, která se při zrychlujícím se vývoji a snižující se cenou klesá. Naproti tomu agilní přístup k problematice přistupuje přesně naopak. Rozsah je variabilní (požadavky zákazníků se v současné době mohou rychle měnit i v průběhu vývoje), pevná je však cena, čas a tím pádem i kvalita [16] [17].

Jaké jsou hlavní rozdíly, které dělají agilní přístup tak efektivní? Jsou to především neustálá komunikace a validizace se zákazníkem či uživatelem, iterativní a inkrementální vývoj ve velmi krátkých iteracích, osobní komunikace v týmu a v neposlední řadě opakované a průběžné automatické testování [16].

Prvním jmenovaným rozdílem oproti tradičním přístupům je častá komunikace se zadavatelem, která spočívá v neustálém získávání zpětné vazby na již vyvinuté části programu a v průběžném zjišťování požadavků k funkcím či komponentám, které budou vyvíjeny v nejbližších iteracích. S tímto souvisí iterativní a inkrementální vývoj, jehož cílem je přinášet zákazníkovi hodnotu (i když jen malou) v co nejkratších intervalech. Zákazník tak získává produkt po částech přesně podle jeho aktuálních požadavků a při jeho vývoji se může sám podílet na návrhu softwaru. V průběhu vývoje je pak důležité, aby se informace od zákazníka co nejrychleji dostávaly ke všem členům vývojového týmu, proto je preferovaná osobní komunikace zpravidla formou pravidelných (obvykle denních) schůzek, které mimo jiné přispívají k dřívějšímu odhalování problémů a ke kontrole postupu ve vývoji [16].

Samostatnou kapitolou v agilních metodikách jsou automatické testy, které průběžně a opakovaně pomáhají odhalovat chyby v produktu a tím pomáhají k doručování bezchybného softwaru zákazníkům. Protože vývoj v krátkých iteracích umožňuje reagovat na změnu požadavků zadavatele, je pravděpodobné, že bude častěji zasahováno do kódu, který byl vytvořen dříve, přidávána nová funkcionalita, a naopak odebírány části kódu, které jsou již neaktuální či nevyužívané. Z těchto důvodů je pokrytí co největšího podílu zdrojového kódu automatickými testy více než žádoucí. Během průběžného testování lze pak odhalit, že přidáním, úpravou či odebráním některé části kódu se vytvořila chyba v jiné části. V současných trendech je snaha o 100% pokrytí zdrojového

kódu, avšak existuje metodika, která úplné pokrytí umožňuje (možná spíše vyžaduje). Tuto metodiku zvanou Test-Driven Development blíže popíši v následující podkapitole, nyní však ještě uvedu několik jiných zástupců agilního přístupu [16] [18].

Pravděpodobně nejznámější metodikou v agilním vývoji je Extrémní programování (Extreme Programming), ta si klade za cíl použití běžných postupů a myšlenek, avšak dotažených do extrémů. Příkladem může být úvaha o jednoduchosti – při extrémním programování je vyžadována jednoduchost, extrémní jednoduchost. Té je docíleno tak, že se vývojáři snaží vytvořit nejjednodušší možnou část softwaru, která však ještě bude splňovat požadavky a bude funkční. Tj. nic navíc [16].

Další, v poslední době velmi oblíbenou, metodikou je SCRUM Development Process (zkráceně jen SCRUM). Tento přístup byl nastíněn již v roce 1995 a jeho hlavní charakteristikou jsou tzv. sprinty – 14denní až měsíční intervaly, během nichž tým vybere a přijme předem ohodnocené úkoly, poté vyvíjí a na konci zákazníkovi předvádí dodanou funkcionalitu. V rámci sprintů se pak pořádají každodenní schůzky (Scrum Meetings), během kterých se členové týmu informují o dosavadním postupu, odhalují problémy, identifikují nové úkoly a nepřímo také zvyšují soudržnost týmu. Tyto schůzky de facto nahrazují podrobné plánování na celý sprint dopředu a lze se tak mnohem flexibilněji přizpůsobovat nečekaným změnám či problémům. Mimo to je potřeba shromažďovat a ohodnocovat úkoly pro další sprinty. Úkoly se průběžně zadávají do tzv. Backlogu, jejich ohodnocení se pak provádí na schůzkách nazývaných Grooming, kdy členové týmu diskutují nad řešením úkolů a poté společně úkoly ohodnotí podle obtížnosti [16].

Lean Development je jednou z dalších metodik, jejímž základem je absolutní odstranění všeho zbytečného, co vzniká při vývoji, snižuje efektivitu vývojářů a zvyšuje náklady. Původně byla metodika vytvořena při restrukturalizaci japonské automobilky, později byla převedena i do softwarového vývoje. K naplnění této metodiky bylo definováno desatero pravidel, které pomáhají zamezit plýtvání zdroji, ať už lidskými, výrobními nebo finančními [16].

Dalšími nejznámějšími zástupci jsou Adaptive Software Development, Feature-Driven Development, Crystal Methodologies či Dynamic System Development Method. Nové

metody založené na Agilním manifestu vznikají neustále, avšak jejich výčet by byl spíše zbytečný, pro představu, jak funguje agilní vývoj jsou dostačující předchozí uvedené [16].

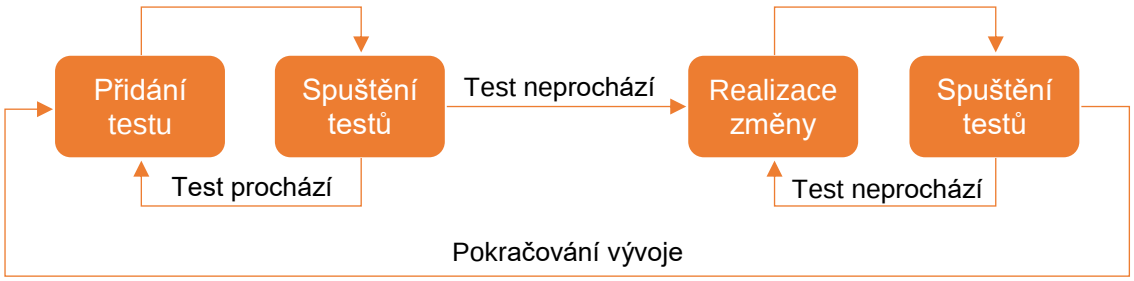
2.7.1 TEST-DRIVEN DEVELOPMENT

Popis jedné metodiky agilního vývoje jsem však nechal až na konec, protože je jako jediná metoda postavená trochu odlišně. V kapitole o agilních metodikách jsem uvedl, že součástí každé metodiky je automatické testování. V Test-Driven Development (TDD, v překladu Programování řízené testy) to však platí dvojnásob. Automatické testy jsou totiž jeho základní stavení jednotkou [16] [18].

Během vývoje rozsáhlejšího softwaru se bude zvyšovat i potřeba udržování staršího kódu. Růst podílu údržby na vývoji však ve většině případů roste rychleji než velikost produktu, proto se dříve či později vývojáři dostanou do situace, kdy již nebudou mít čas na vývoj nových funkcí produktu, ale budou jen opravovat chyby a udržovat kód. Tomuto však mohou zabránit automatické testy [16] [18].

TDD je metodika, při jejímž dodržení bude zdrojový kód pokrytý testy v celém rozsahu, tzn. že úplně každá funkcionalita bude automaticky testována. To umožňuje odlišný přístup k vývoji softwaru, kdy základní myšlenkou je, že před samotným vývojem funkcionality se nejdříve napíše test této funkcionality a teprve poté se vyvíjí kód tak, aby test procházel [16] [18].

Programování řízené testy má čtyři fáze. V první se co nejrychleji přidá nový test, který bude testovat později přidávanou funkci, vlastnost či komponentu apod. Předpokladem však je, že tento test nesmí procházet, tzn. musí skončit chybou. Ve druhé fázi se spustí veškeré dosavadní testy, aby se vývojář ujistil, že software je před jeho zásahem v pořádku a automatické testy proházejí. Následně vývojář píše či upravuje samotný zdrojový kód tak, aby ve čtvrté fázi prošly veškeré testy. V poslední fázi se provádí tzv. refactoring, kdy se odstraňují duplicitní části kódu a daná iterace se zakončí ověřením integrity testů [18].



Obrázek 5: Grafické znázornění metodiky Test-Driven Development

Zdroj: Vlastní zpracování dle [18]

V průběhu celého vývoje se pak těchto pět fází neustále opakuje, a pokud vývojář (nebo vývojáři) tuto metodiku dodrží, bude výsledkem zdrojový kód zcela pokrytý automatickými testy a navíc, protože vývoj bude probíhat po malých iteracích s postupným přidáváním jednotlivých funkcionalit, bude kód přehledný a snadno udržovatelný [18].

Nevýhodou této metodiky je však nutnost jejího úplného dodržení, což může být pro některé programátory velmi nepohodlné. Navíc je k aplikování TDD potřeba využít nějakého nástroje, který podporuje testování. Jedním z volně dostupných (open-source) je rodina nástrojů xUnit, které umožňuje testovat kód psaný v jazyku Java, Visual Basic, C#, PHP a mnoho dalších, avšak ten v současné verzi nepodporuje testování UWP aplikací. Podobné funkce nabízí i framework přímo od Microsoft, který byl nakonec zvolen pro vývoj navrhované aplikace v rámci této práce [18].

2.7.2 FRAMEWORK MSTEST

Framework MSTest lze jako doplněk nainstalovat přímo do vývojového prostředí Visual Studio, čímž lze pohodlně spravovat zdrojový kód a testy v rámci jednoho řešení. Níže je uveden příklad jednoduchého testu, který využívá tohoto nástroje [5] [6].

```
[TestMethod]
public void HelloWorldTest()
{
    string expected = "Hello world!";
    Assert.AreEqual(expected, HelloWorld(), "Strings are not equal.");
}
```

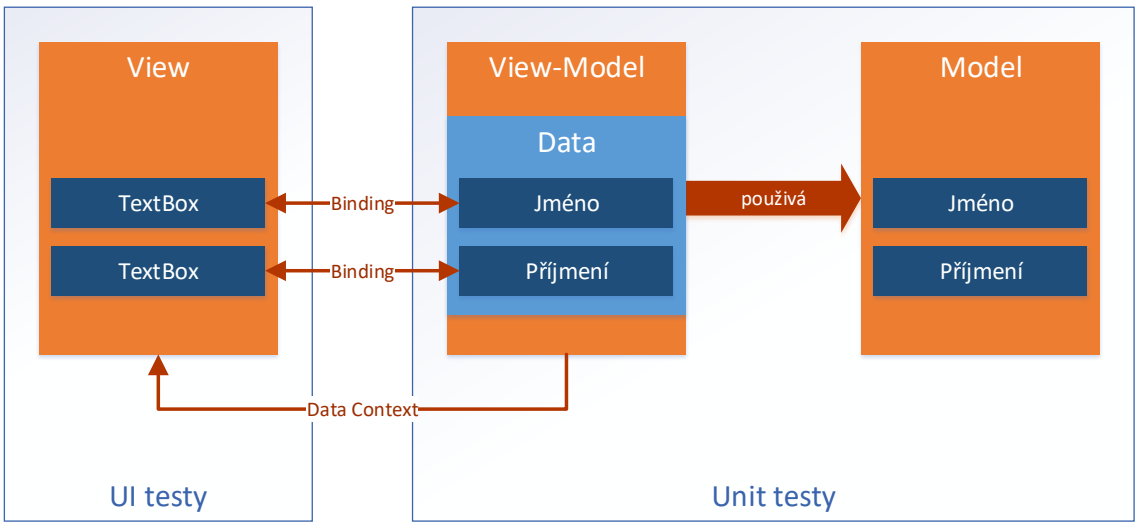
Test, tedy metoda `HelloWorldTest()`, je velmi primitivní a nedělá nic jiného, než že porovná výstup metody `HelloWorld()` s očekávaným výsledkem (proměnná `expected`). K porovnávání slouží třída `Assert`, která obsahuje několik metod. Jednou z nich je právě metoda `AreEqual()`, která jako parametry přijímá prvky, které má porovnat. Testovací framework také umožňuje ke každému assertu přidat text, který se bude zobrazovat ve vývojovém prostředí v případě, že test neprojde. Je tedy vhodné přidávat co nejpopisnější text, který dopomůže k rychlému objasnění důvodu, proč některý z testů neprochází [5].

Třída `Assert` obsahuje ještě několik dalších metod pro testování, mezi nejčastěji používané patří `AreNotEqual()` (test prochází v případě, že objekty nejsou stejné), `IsTrue()`, `IsFalse()` (test prochází v případě, že je parametr vyhodnocen jako pravda – true, respektive nepravda – false), `IsNull()` a `IsNotNull()` (test prochází, jestliže zadaný parametr je respektive není typu Null). Samozřejmě metod pro porovnávání existuje celá řada, každá je vhodná pro odlišné situace [5].

2.8 NÁVRHOVÝ VZOR MVVM

Programování řízené testy s sebou přináší požadavky na rozdělení kódu na malé kousky (jednotky), kdy každý z nich by měl vykonávat pouze jednu funkci tak, aby byly snadno testovatelné. Při vývoji softwaru však můžeme narazit na to, že velká část kódu je netestovatelná nebo testovatelná pouze rozsáhlými integrovanými testy z důvodu velké provázanosti s uživatelským rozhraním či externími zdroji dat. Takové testy pak mohou u většího projektu běžet i několik hodin a vývojář tím ztrácí rychlou zpětnou vazbu, zda neprovedl v kódu nějakou chybnou úpravu. Ideálně by však podle TDD měl být kód zcela pokrytý jednotkovými testy, které poskytují rychlou či téměř okamžitou odezvu. K tomu nám může pomoci právě návrhový vzor MVVM, který odděluje prezenční vrstvu (View) od aplikační logiky (Model), a proto může být funkcionalita kompletně testována bez jakýchkoli vazeb na prezenční vrstvu. Komunikaci uživatelského prostředí a aplikační logiky zprostředkovává takzvaný View-Model. Ten ve svých vlastnostech uchovává data instancí, která získává z Modelu obsahujícího logiku objektů. Prezenční vrstva pak přistupuje k instanci View-Modelu, který má nastavený jako datový kontext, a mapuje (binduje) data z jeho vlastností do ovládacích prvků uživatelského rozhraní. Aplikační logika zahrnující View-Model a Model je tak zcela nezávislá na prezenční vrstvě a může

být testována jednotkovými testy. View je pak možno testovat pomocí testů uživatelského rozhraní (UI) [19].



Obrázek 6: Návrhový vzor MVVM s vyznačením testovatelnosti

Zdroj: Vlastní zpracování dle [19]

3 ANALÝZA PROBLÉMU A SOUČASNÉ SITUACE

Chytré mobilní telefony zažívají v posledních letech obrovský boom a staly se neodmyslitelnou součástí života každého z nás. Lidé je mimo uskutečňování hovorů, pro které byly původně stvořeny, využívají stále častěji ke každodenním úkonům a činnostem, ke kterým bylo dříve potřeba využít stolní počítač nebo notebook [20].

V loňském roce bylo na celém světě více než 2,6 miliardy smartphonů a odhaduje se, že v roce letošním bude chytrý telefon používat polovina světové populace. Mezi nejběžnější činnosti patří procházení webových stránek, vyhledávání informací na internetu, čtení a posílání e-mailů, komunikace a sdílení informací na sociálních sítí či hraní her. Protože k většině aktivitám je potřeba datové připojení, došlo v loňském roce poprvé v historii k většímu internetovému provozu z mobilních zařízení než z klasických počítačů. Rostoucí trend smartphonů zaznamenávají i elektronické obchody, kdy téměř polovina objednávek v České republice byla uskutečněna prostřednictvím chytrého telefonu nebo tabletu [20] [21].

Vzrůstající preference mobilních zařízení před počítači jde ruku v ruce s neustále zvyšujícími se nároky na funkce a aplikace, kterými je smartphone vybaven. Tím se mění i priority softwarových společností, které se snaží pokrýt rostoucí poptávku po mobilních aplikacích a dodat svým zákazníkům či uživatelům služeb přidanou hodnotu, kterou se odliší od konkurence.

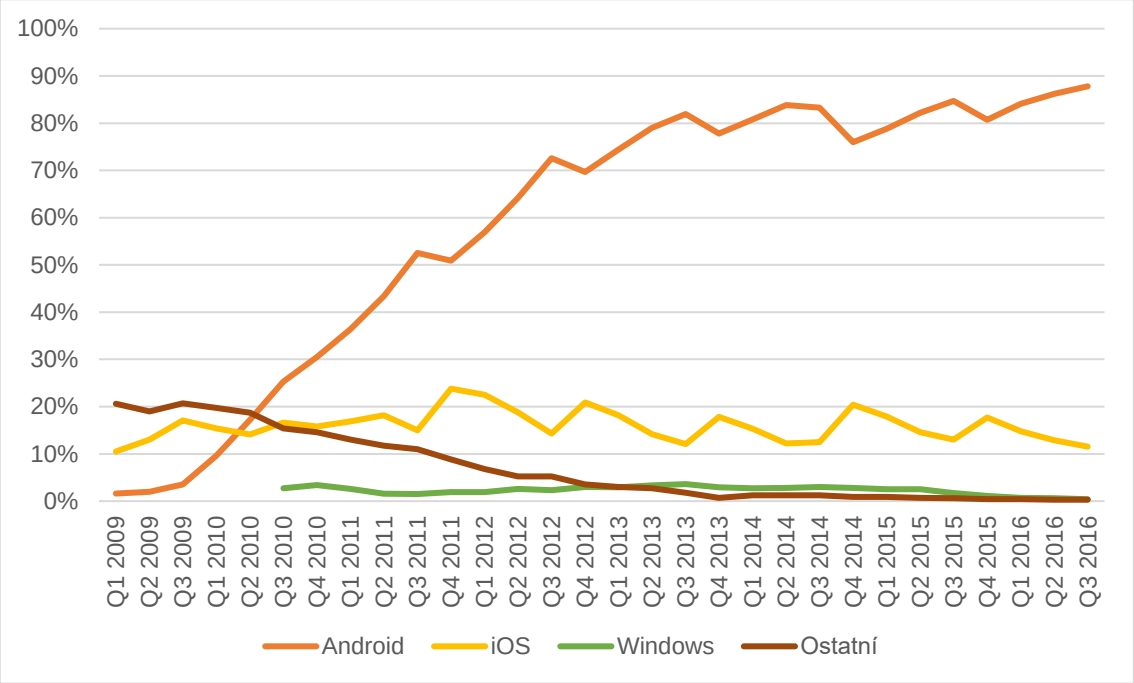
3.1 ANALÝZA TRHU MOBILNÍCH OPERAČNÍCH SYSTÉMŮ

Na trhu mobilních operačních systémů již více než 5 let vládne Android. Ve třetím kvartálu roku 2016 bylo prodáno přes 300 miliónů smartphonů právě s tímto operačním systémem, což činilo 87,8 % ze všech prodaných mobilních zařízení. Druhou příčku obsadily produkty Apple s operačním systémem iOS, kterých se prodalo 11,5 %. Smartphony s operačním systémem Windows se podílely 0,4 % na celkovém prodeji a ostatní operační systémy si rozdělily zbývající 3 desetiny procenta [22].

Společnosti Google, která od roku 2008 Android vyvíjí, se podařilo vybudovat kvalitní a stabilní systém, který velmi rychle získal přízeň mnoha uživatelů. K jeho rozšíření také pomohla distribuční politika, kdy se Android nabízí jako platforma, která obsahuje

otevřený operační systém i kompletní řešení pro výrobce, kteří jej mohou jednoduše nasadit ve svých zařízeních. S Androidem se v současné době můžeme setkat na zařízeních Samsung, Sony, LG, HTC, Asus, Huawei, Xiaomi, Google a dalších [23].

Tento přístup k distribuci operačního systému se zcela liší od přístupu společnosti Apple, která naopak nabízí uzavřený systém iOS pouze ve svých zařízeních iPhone, iPod a iPad. Pro uživatele má uzavřenost systému jednu zásadní výhodu, kterou je stabilita a výkonnost systému a provázanost s ostatními produkty této společnosti. Nevýhodou pak je odkázanost uživatelů na produkty jednoho výrobce. I přes to, že Apple byl inovátorem, který přišel jako první s cela jiným pohledem na mobilní telefony, jeho podíl na trhu stagnuje, v posledních letech dokonce mírně klesá [22] [24].



Graf 1: Podíl prodejů mobilních operačních systémů ve světě

Zdroj: Vlastní zpracování dle [22]

Třetím operačním systémem, který sice v současné době dosahuje v porovnání s předešlými konkurenty velmi nízkých prodejů je mobilní systém Windows (nebo Windows Phone) vyvíjený společností Microsoft. Windows je známý především z desktopových počítačů a notebooků. Počátek vývoje jeho mobilní verze však sahá do roku 2000, kdy byl na trh uveden operační systém Počet PC 2000 pro tehdejší kapesní počítače. Ty však představují mírně odlišnou kategorii přenosných zařízení, a proto lze

za první operační systém pro chytré telefony považovat až systém Windows Pocket PC 2002 Phone edition představený roku 2002, i když v tomto případě šlo jen o přidání telefonních funkcí. Proto v grafu výše uvádím mobilní operační systém Windows až od roku 2010, kdy byla představena nová platforma Windows Phone. Společnost Microsoft k distribuci systému přistupuje podobně jako Google, nabízí jej tedy různým výrobcům. Se systémem Windows se proto můžeme setkat v zařízeních jak od Microsoftu, tak i jiných výrobců jako Acer, Alcatel, HTC, HP, LG, VAIO, Xiaomi a dříve v zařízeních Nokia [5] [25].

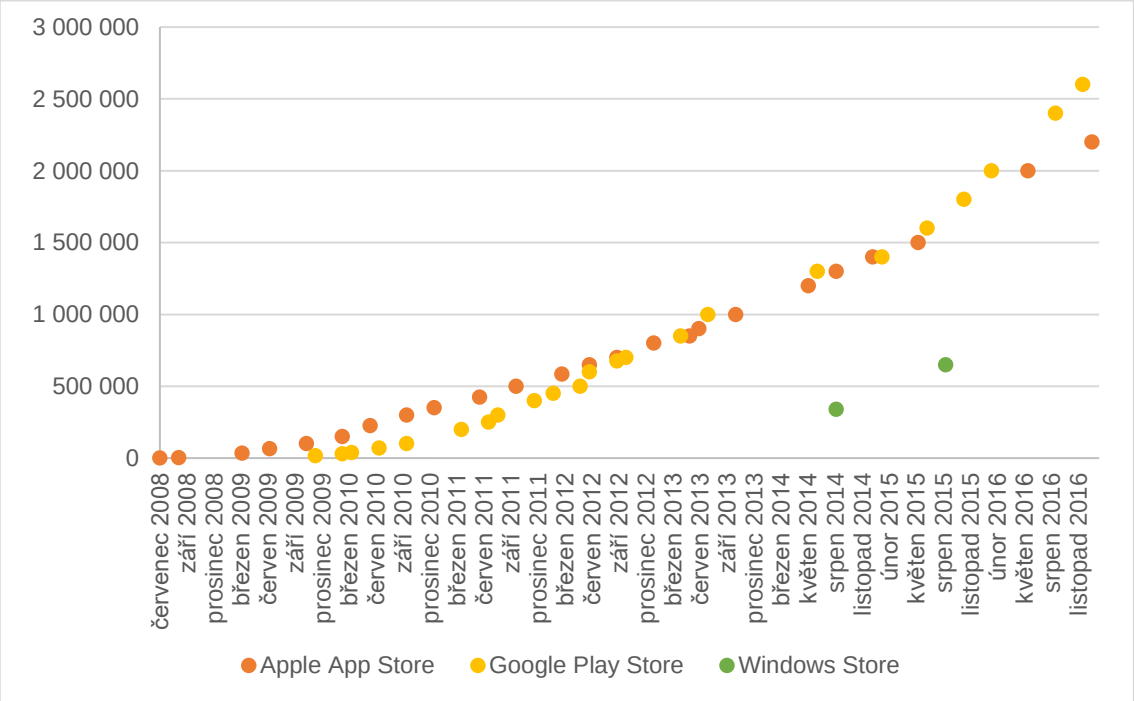
I když by se na první pohled podle klesajících prodejů mohlo zdát, že mobilní platforma Windows je neperspektivní a nevyplatí se do ní investovat, opak může být pravdou. Vyplývá to především z toho, že Microsoft si je vědom změn trhu s výpočetní technikou a pokud chce udržet dominantní místo na trhu, musí se mu přizpůsobit a zaměřit se i na mobilní zařízení, která tento trh začínají ovládat. Klesající prodeje lze přisuzovat převážně tomu, že v loňském roce bylo představeno minimum modelů Microsoft Lumia, protože se společnost soustředí na svoji chystanou „vlajkovou loď" Surface Phone, který by měl být představen v průběhu letošního roku. Lze tedy předpokládat, že vývoj mobilní verze systému Windows nebude ukončen, ale naopak na jej bude kladen větší důraz. Toto tvrzení podporuje i několik kritiků smartphonů, kteří shledávají systém Windows plnohodnotným a velmi povedeným. Důvodem nízkých prodejů může být také právě nedostatečná podpora ze strany vývojářů mobilních aplikací [26].

3.2 ANALÝZA TRHU S APLIKACEMI

Aplikace jsou nepostradatelnou součástí chytrých telefonů, které zpřístupňují nové a pokročilé funkce. Podle průzkumu portálu Seznam.cz může dokonce konkrétní aplikace hrát zásadní roli při výběru nového smartphonu, kdy její absence na některé platformě může způsobit vyřazení všech mobilních zařízení s touto platformou [27].

Počet nabízených aplikací neustále roste, v současné době nabízí Google Play Store přes 2,6 milionu aplikací. Podobně je na tom i Apple App Store, který nabízí 2,2 milionu aplikací a každým měsícem se počet zvyšuje. Microsoft počet aplikací ve svém Windows Storu příliš často nezveřejňuje, ale tvrdí, že každý den přibývají stovky nových aplikací. S příchodem Windows 10 se navíc aplikace pro desktopové počítače, tablety a mobilní

telefony vyvíjejí stejně a s minimem práce navíc mohou být aplikace takzvaně universální. Tím se skokově zvětšil počet uživatelů, kteří si mohou aplikaci z Windows Storu nainstalovat, a zdálo se, že často zmiňovaný problém, kdy vývojáři nechtějí vyvíjet aplikace pro Windows právě z důvodu nedostatečně velké cílové skupiny uživatelů, se vyřeší. Očekávání se však nenaplnila a zájem vývojářů o aplikace pro mobilní Windows se příliš nezvýšil [28] [29] [30].



Graf 2: Počet aplikací v Apple App Store, Google Play Store a Windows Store

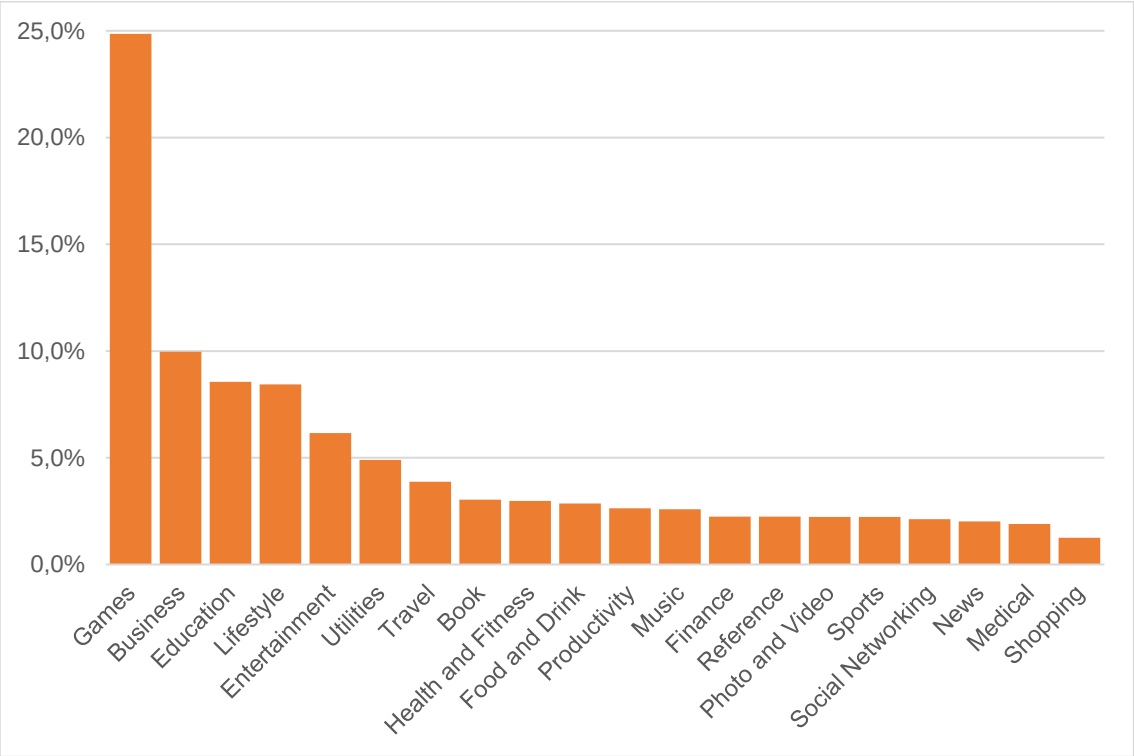
Zdroj: Vlastní zpracování dle [28] [29] [30]

Vyvstává však otázka, zda čísla množství aplikací nejsou v současné době už jen pouhým lákadlem na uživatele. V obchodech se totiž často objevují aplikace, které jsou si podobné, někdy dokonce téměř identické, což potvrzuje i fakt, že aplikací s průměrným hodnocením více než 3 hvězdičky (při minimu 1 a maximu 5) bylo na konci roku 2014 v Apple Store pouze 37 % ze všech ohodnocených, což odpovídá pouhým 17 % v případě započtení i neohodnocených aplikací [28].

Kvalitních aplikací, ať už pomůcek pro každodenní činnosti, plánovačů, multimediálních aplikací, přehrávačů, kancelářských aplikací či her, je napříč všemi obchody podstatně méně. Nicméně i počet těchto aplikací stále roste a s tím roste i množství uživatelů, kteří

jsou ochotni za aplikace platit, což podporuje opět růst zájmu vývojářů, protože se zvětšuje jejich cílová skupina. Tento nekonečný cyklus se pravděpodobně nezastaví, dokud nebude trh chytrými telefony a jejich aplikacemi přesycen.

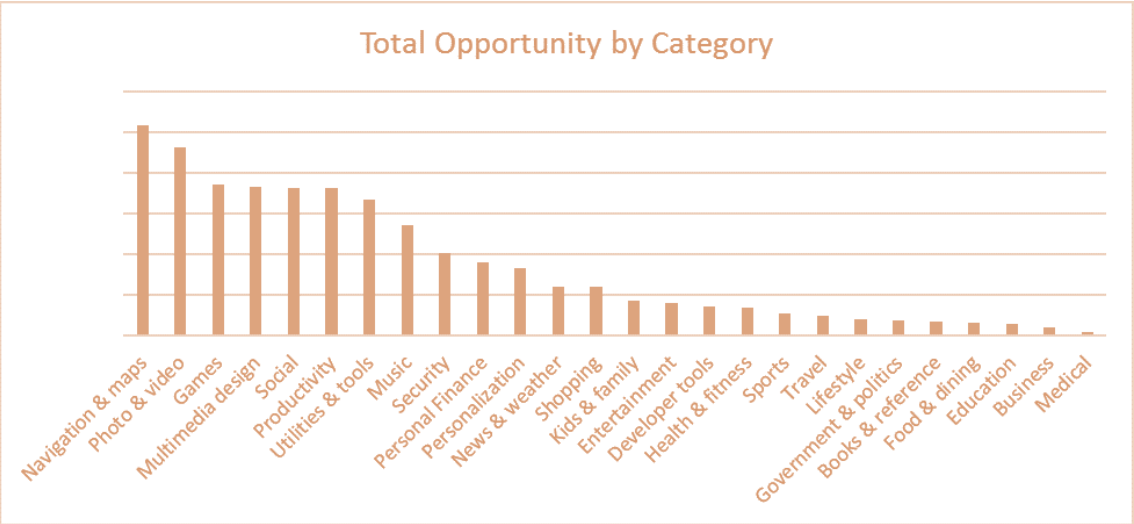
V současné době zájem vývojářů stále roste a tím roste i jejich potřeba nacházet mezery na trhu. K rozhodování jim mohou pomoci nejrůznější statistiky, např. na grafu níže je zobrazen podíl aplikací v jednotlivých kategoriích na Apple App Store [31].



Graf 3: Podíl aplikací v jednotlivých kategoriích na Apple App Store

Zdroj: Vlastní zpracování dle [31]

Daleko přínosnější však pro vývojáře mohou být statistiky o posledním vývoji aplikací v kategorii, jako např. nejrychleji rostoucí kategorie, průměrný počet stažení každé aplikace apod. [31].



Graf 4: Celková příležitost dle kategorií

Zdroj: Převzato z [32]

Microsoft vydává na svém blogu články „Windows Store Trends“, kde několik klíčových statistik zveřejňuje, a dokonce uvádí i graf „Celková příležitost podle kategorií“, který vývojáři poskytne rychlý přehled o tom, ve které kategorii mají aplikace největší poměr stažení ku celkovému počtu stažení všech aplikací [32].

3.3 ANALÝZA SPOLEČNOSTI KENTICO SOFTWARE

Brněnská společnost Kentico Software s.r.o. byla založena v roce 2004, v témž roce byla vydána první verze softwaru Kentico CMS for ASP.NET. Od té doby se společnost postupně rozrůstá a v současné době její hlavní produkt Kentico CMS používá téměř 15 % společností s největšími tržbami ve světě, jako jsou Heineken, Hyundai, Honeywell nebo John Deere, z českých společnostní například Sazka. V současnosti na tomto systému běží více než 25 tisíc webových prezentací a kampaní ve více než 100 zemích po celém světě a společnost spolupracuje s přibližně tisícem partnerů [33].

Firma před nedávnem otevřela novou pobočku v Nizozemí, mimo to má další pobočku v USA, Austrálii a Velké Británii. Hlavní sídlo společnosti je však stále v Brně, kde je soustředěna veškerá vývojová síla čítající téměř 200 zaměstnanců [33].

Kentico od svého začátku cílí na koncové zákazníky a snaží se jim dodat produkt, který chtějí, který budou rádi používat a který jim pomůže dosáhnout jejich cílů. Důraz také klade na nepřetržitou zákaznickou podporu, bezpečnost produktu a jeho kvalitu. Pokud

zákazník například najde v produktu nějakou chybu, tzv. bug, je vývojáři opraven v rámci sedmidenní hotfix politiky, což nemá v softwarových firmách podobných velikostí obdoby. Tento přístup je zákazníky velice kladně hodnocen a produkt je jimi doporučován dále. Firma tak dokáže snadno konkurovat i větším společnostem, a to i při mnohem nižších nákladech na marketingové kampaně [33].

V roce 2016 bylo Kentico jako první česká firma zařazena do Gartnerova magického kvadrantu, a to hned ve dvou kategoriích. Gartner je jedna z neuznávanějších a největších světových analytických organizací, která každý rok doporučuje vhodné softwarové řešení pro různé kategorie a tisíce firem po celém světě tato doporučení zohledňují při výběru produktu, který budou používat pro své podnikání [33] [34].

Pro Kentico je důležitou kategorií „Web Content Management“, volně přeloženo „Systém pro správu webového obsahu“, protože právě do ní nejvíce spadá software Kentico CMS. Firmy jsou v magických kvadrantech posuzovány podle dvou základních ukazatelů, podle úplnosti firemní vize a podle schopnosti realizace produktu a doplňujících služeb. Kentico bylo zařazeno mezi „Challengers“ – vyzyvatele [34].

Do tohoto kvadrantu jsou zařazeny společnosti, které mají dobré postavení z pohledu realizace produktu, ale nemusejí mít dostatečně silnou a propracovanou strategii či vizi, aby dokázali doručovat zákazníkům dlouhodobě stabilní přidanou hodnotu. To může být způsobeno tím, že společnosti dostatečně rychle nereagují na změnu trhu či jemu směřování správně neporozumí. Vyvíjený produkt pak může snižovat míru uspokojování požadavků zákazníků a poptávka po něm může v čase klesat, i když by se jednalo o zcela bezchybné a kvalitní řešení [33] [34].

Obrázek 7: Gartner Magic Quadrant for WCM, 2016

Zdroj: Převzato z [34]

Kentico si tuto situaci uvědomuje, a proto před dvěma lety pomocí interního start-up projektu započalo vývoj nového softwaru u kterého věří, že jednou zcela nahradí dosavadní Kentico CMS. Jedná se o podobný software, který je však nabízen v Cloudu jako SaaS (Software as a Service). Je vyvíjen ve 14denních iteracích, kdy se všechny nové funkce dostávají k zákazníkům téměř okamžitě. Po dvou letech vývoje má společnost k dispozici produkt v segmentu, který se teprve rozvíjí a podle nejrůznějších odhadů jeho masivní růst a popularita se dá teprve očekávat. Kentico se tak značnou mírou podílí na tom, jakým směrem se Cloudové služby z pohledu správy webového

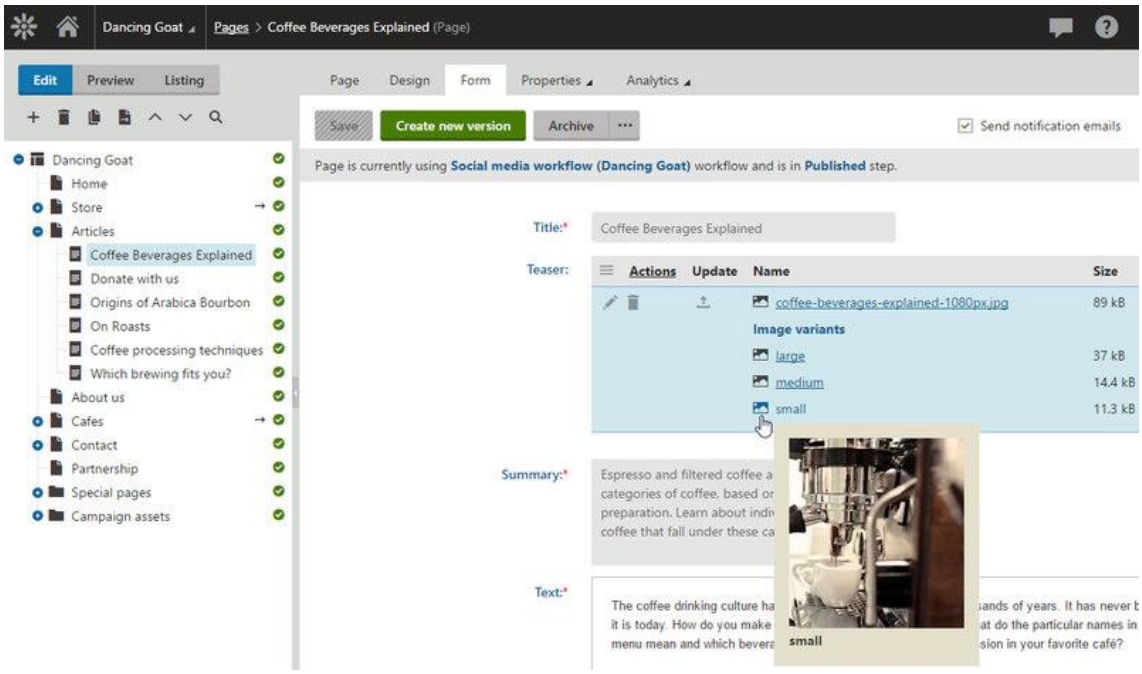
obsahu budou nadále ubírat. A jako u Kentico CMS je i u tohoto produktu kladen největší důraz na potřeby zákazníků [33].

3.3.1 KENTICO CMS

Software Kentico CMS není jen systém pro správu obsahu (content management system), jak by se z názvu mohlo zdát. Zkratka CMS je v názvu v současné době převážně jen z historických důvodů, a proto se někdy lze setkat i s označením Kentico CMS/EMS nebo jen Kentico [33].

Systém je nabízen jako All-in-One řešení, které sdružuje nástroje pro správu webového obsahu, tvorbu internetových obchodů a řízení rozsáhlých online marketingových kampaní. V jednom systému tak lze vytvořit např. webovou prezentaci s e-shopem a personalizací obsahu podle určitého chování návštěvníků na stránce. V rámci téhož systému lze také provozovat více projektů na různých webových doménách, což ocení společnosti, které se zabývají tvorbou webových stránek, kdy jednotlivé projekty jsou v rámci aplikace odděleny tak, aby se vzájemně neovlivňovaly a zabezpečovaly plynulý provoz všech služeb [33].

Důraz je mimo bezpečnost kladen také na optimalizaci, aby výkon aplikace dosahoval co nejlepších výsledků. Poslední verze Kentico 10 je velice robustní, že dokáže zpracovávat až 100 miliónů kontaktů a až miliardu marketingových aktivit. Pokročilé funkce ocení také developeři, kteří budou se systémem pracovat ať už z důvodu úpravy, tvorby front-endové části webu či vývoje vlastních modulů [33].



Obrázek 8: Ukázka back-endové části softwaru

Zdroj: Převzato z [33]

Software je nabízen v různých variantách, kdy v každé z nich jsou zpřístupněny na základě dané licence pouze některé části modulu. Kompletní verze je pak nabízena se všemi moduly a za příplatek lze ještě poskytnout celý zdrojový kód, což ocení společnosti, které chtějí systém více modifikovat pro vlastní potřeby. Nové verze Kentica jsou zveřejňovány vždy jednou za rok. Během této doby se development zaměřuje na vývoj nových funkcionalit a úpravu těch stávajících podle požadavků zákazníků [33].

Mimo to se společnost zabývá i vývojem podpůrných nástrojů a aplikací. Na základě zákaznických požadavků a v souvislosti s vývojem trhu mobilních aplikací, který byl popsán v předchozí části této diplomové práce, se společnost rozhodla vyvinout aplikace pro správu obsahu prostřednictvím mobilních telefonů, a to pro všechny 3 operační systémy s největším podílem na trhu. Dále se tato práce bude zabývat právě vývojem požadované mobilní aplikace pro platformu Microsoft Windows 10 [33].

3.3.2 SOCIÁLNÍ FAKTORY

Typickým klientem společnosti Kentico je firma zabývající se tvorbou velkých webových řešení pro své koncové zákazníky, jedná se tedy o B2B vztah, a proto tyto klienty nazýváme partnery. Klientem však může být i přímo společnost, která potřebuje nějakou

webovou prezentaci, marketingovou kampaň či e-shop a má vlastní developery. Příkladem v České republice může být společnost Sazka a její webový portál [35].

Přestože se množství klientů zvyšuje, je neustále potřeba hledat nové zákazníky, kteří budou produkt využívat. Kentico má proto několik zaměstnanců, kteří se zaměřují pouze na propagaci produktu, která je směřována především na získávání nových partnerů. Ti mají zpravidla vícero svých vlastních zákazníků, jimž mohou produkty Kentico nabídnout [35].

3.3.3 LEGISLATIVNÍ FAKTORY

I v informatice je potřeba dodržovat legislativní nařízení. Pro společnost, která působí na celém světě je pak tento faktor obzvláště důležitý např. při sestavování licenčních podmínek či při uzavírání smluv se zahraničními partnery. Z tohoto důvodu Kentico najímá externí právní společnost s experty na legislativu v jednotlivých zemích, ve kterých se produkty prodávají [35].

3.3.4 EKONOMICKÉ FAKTORY

Kentico bylo založeno v Brně, kde stále sídlí jeho vedení a celé vývojové oddělení, přitom hlavní klientela se nachází v Americe. Toto však má svůj opodstatněný důvod, kterým je nízká cena kancelářských prostor oproti městům v USA či Británii a také větší množství IT škol, ze kterých vychází více absolventů v oblasti IT než v ostatních městech a náklady na platy vývojářů pak nejsou tak vysoké, jako v ostatních městech ČR. Avšak poptávka po zkušených a kvalitních programátorech stále stoupá, a proto lze do budoucna očekávat růst mezd z důvodu zvyšující se poptávky po těchto kvalifikovaných lidech [35].

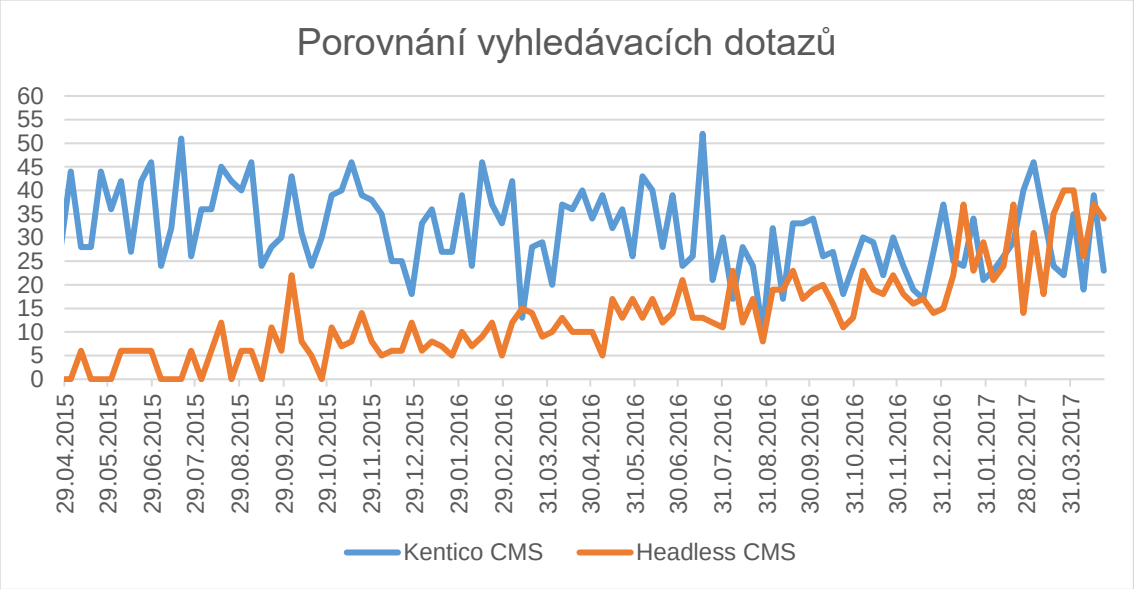
3.3.5 POLITICKÉ FAKTORY

Jelikož je Kentico nadnárodní společnost, ovlivňují ji politické faktury z několika zemí současně. Nejvíce je společnost závislá na příjmech z Amerického trhu, a proto zásadní změny v tamní politice mohou mít vliv na ekonomiku podniku a vývoj tržeb, nikoli však na samotnou existenci z důvodu dostatečně stabilních příjmů i z ostatních zemí [35].

3.3.6 TECHNOLOGICKÉ FAKTORY

Technologie se v IT oblasti mění rychleji než v jakémkoli jiném odvětví. S nimi se mění i požadavky zákazníků, kteří používají nová zařízení a služby. Příkladem může být již

zmiňovaný nový produkt Kentico Cloud, který odráží nové požadavky na poskytování produktů jako služby, přitom ještě před dvěma lety byla tato myšlenka v oblasti správy webového obsahu spíše úsměvná. Dokazuje to i následující graf, ve kterém jsou proti sobě postaveny vyhledávací dotazy na Kentico CMS a Headless CMS, jak jsou systémy pro správu obsahu poskytované jako služby nazývány. V průběhu dvou let se týdenní množství obou dotazů srovnalo nebo dokonce CMS jako SaaS začíná být v posledních týdnech vyhledáváno častěji [35].



Graf 5: Porovnání vyhledávacích dotazů

Zdroj: [35]

Podobně je na tom i rozšiřování chytrých telefonů a jejich funkcí. V současné době je možné přes smartphone udělat téměř jakoukoli kancelářskou práci, ke které byl ještě před nedávnem potřeba počítač. Kentico se v poslední verzi proto zaměřilo na optimalizace systému CMS pro zobrazení na mobilních zařízeních, avšak stále chybí nativní podpora v podobě aplikací [35].

3.3.7 SILNÉ A SLABÉ STRÁNKY SPOLEČNOSTI

Společnost působí na trhu téměř 15 let a za tu dobu se jí podařilo vybudovat prestiž a získat mnoho spokojených zákazníků převážně na americkém trhu, ale i na dalších trzích po celém světě. Po celou dobu je firma financována pouze z vlastních příjmů a je

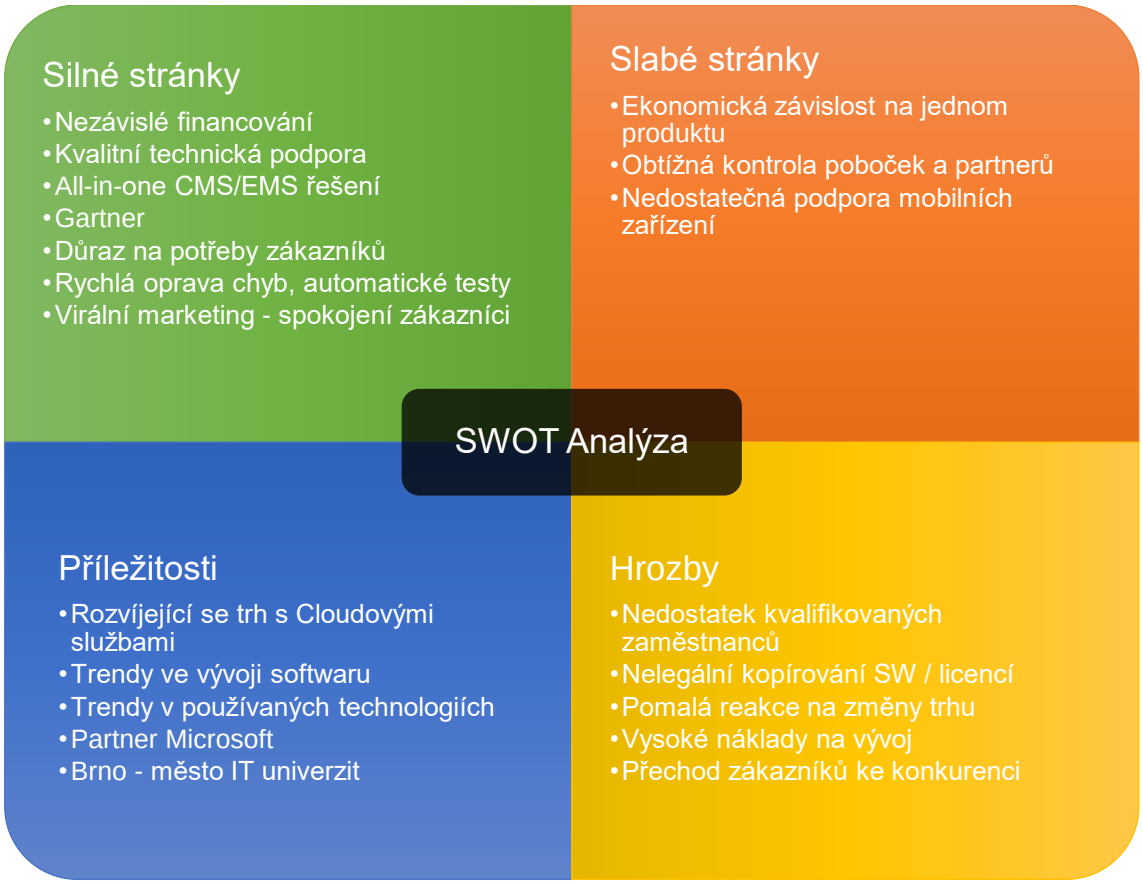
tedy zcela nezávislá na externím financování. Mezi silné stránky společnosti patří také orientace na zákaznické potřeby a na rychlou opravu případných chyb. S tím také souvisí rozvoj automatického testování softwaru pro eliminaci lidských chyb při vývoji.

Největší slabinou společnosti je závislost na jednom produktu. I když se v posledních letech investuje mnoho finančních i lidských zdrojů do vývoje Kentico Cloud, je tento produkt na začátku svého životního cyklu a příjmy z jeho prodeje jsou zatím marginální. Většinu příjmů tedy do společnosti přivádí jediný produkt – Kentico CMS. Další slabou stránkou jsou vysoké vzdálenosti centrály od ostatních poboček, což ztěžuje případné organizační změny nebo kontroly. Tuto slabou stránku však nelze eliminovat, protože z hlediska nepřetržité zákaznické podpory je rozmístění poboček po světě ideálním řešením.

Naopak umístění celého developmentu v Brně je pro společnosti velkou příležitostí. V Brně je hned několik vysokých škol nabízejících řadu IT studijních oborů, ze kterých pak vycházejí ambiciózní programátoři. Na HR oddělení společnosti je tuto příležitost využít a tyto studenty oslovovat už během studia, protože kvalitních vývojářů je dlouhodobě nedostatek. Pokud by společnost nestíhala nabírat nové zaměstnance tak rychle, jak by bylo potřeba, začal by se brzdit vývoj softwaru, což by mohlo mít negativní vliv na konkurenceschopnost. Podobně by společnost měla využívat i partnerství se softwarovým gigantem Microsoft, který může být zdrojem kvalitních konzultantů a poradců.

I přes minimální příjmy Kentico Cloud je tento produkt obrovskou příležitostí, jak být u začátku formování nového trhu cloudových služeb, stát se inovátorem v odvětví a získat spoustu nových zákazníků. S tímto úzce souvisí i nové trendy ve vývoji softwaru. V posledních letech se velmi rozmáhají webové aplikace, které jsou optimalizovány tak, aby po internetu přenášely co nejméně dat a zároveň nabízely rozsáhlé funkce běžných aplikací přímo ve webovém prohlížeči. S tímto však souvisí nelegální kopírování softwaru či částí zdrojového kódu webové aplikace. Ten se většinou v podobě JavaScriptových souborů přenáší přímo do prohlížeče uživatele a v něm se teprve vykonává. I když je kód většinou minifikovaný, velice špatně čitelný a rozdělený do několika souborů, které samy o sobě příliš funkcionality neobsahují, existuje zde riziko, že konkurence tento kód zkopíruje a použije jej u svého řešení.

Mezi další hrozby patří také zvyšující se náklady na vývoj Kentico Cloud, který ovšem prozatím generuje téměř nulové příjmy. Aby jej však uživatelé začali masivně využívat, je potřeba investovat ještě do jeho vývoje mnohem více, což může být pro společnost rizikové v případě, kdy by příjmy z primárního produktu začaly klesat.



Obrázek 9: SWOT analýza společnosti Kentico Software

Zdroj: Vlastní zpracování

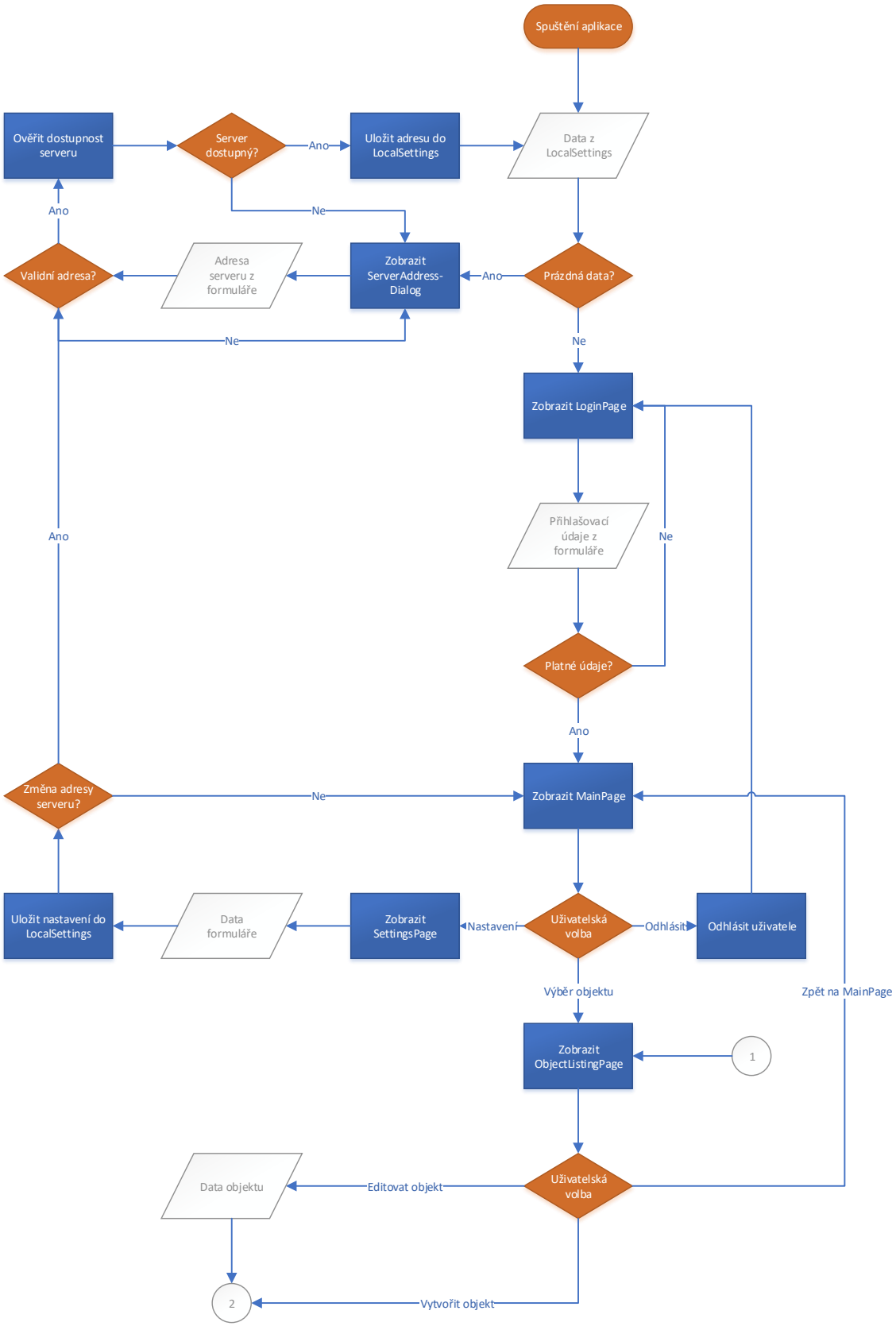
4 VLASTNÍ NÁVRH ŘEŠENÍ

Hlavním cílem této diplomové práce je návrh mobilní aplikace pro správu systému Kentico CMS/EMS. V této části proto bude popsán postup vývoje některých částí aplikace. Nejdříve však budou analyzovány procesy v rámci aplikace a jejich závislosti a posloupnosti. Poté bude navržen design tak, aby odpovídal webovému rozhraní softwaru Kentico CMS a podporoval různé režimy zobrazení na různých velikostech displeje. Dále bude nastíněno programování při využití metodiky Test-Driven Development a popsány některé problémy, které při Unit testování vznikají. Také budou diskutovány možnosti multiplatformního vývoje, jejich výhody či nevýhody a důvody, proč tato aplikace nebyla od základu vyvíjena jako multiplatformní. Na závěr provedu zhodnocení přínosu realizace a ekonomické zhodnocení vývoje.

4.1 NÁVRH PROCESŮ APLIKACE

Aplikace, které komunikují s nějakou webovou službou, mají vesměs stejnou strukturu. První stránkou zobrazenou uživateli je obvykle přihlašovací formulář. U této aplikace tomu nebude jinak s výjimkou případu, kdy se aplikace bude spouštět zcela poprvé. Protože software Kentico provozují zákazníci na svých doménách, nelze adresu serveru zakomponovat přímo do aplikace, ale bude jej muset uživatel zadat ručně. Proto vstupní obrazovkou bude formulář pro nastavení adresy serveru. Pokud bude zadána validní adresa, zobrazí aplikace uživateli již běžný přihlašovací formulář (LoginPage).

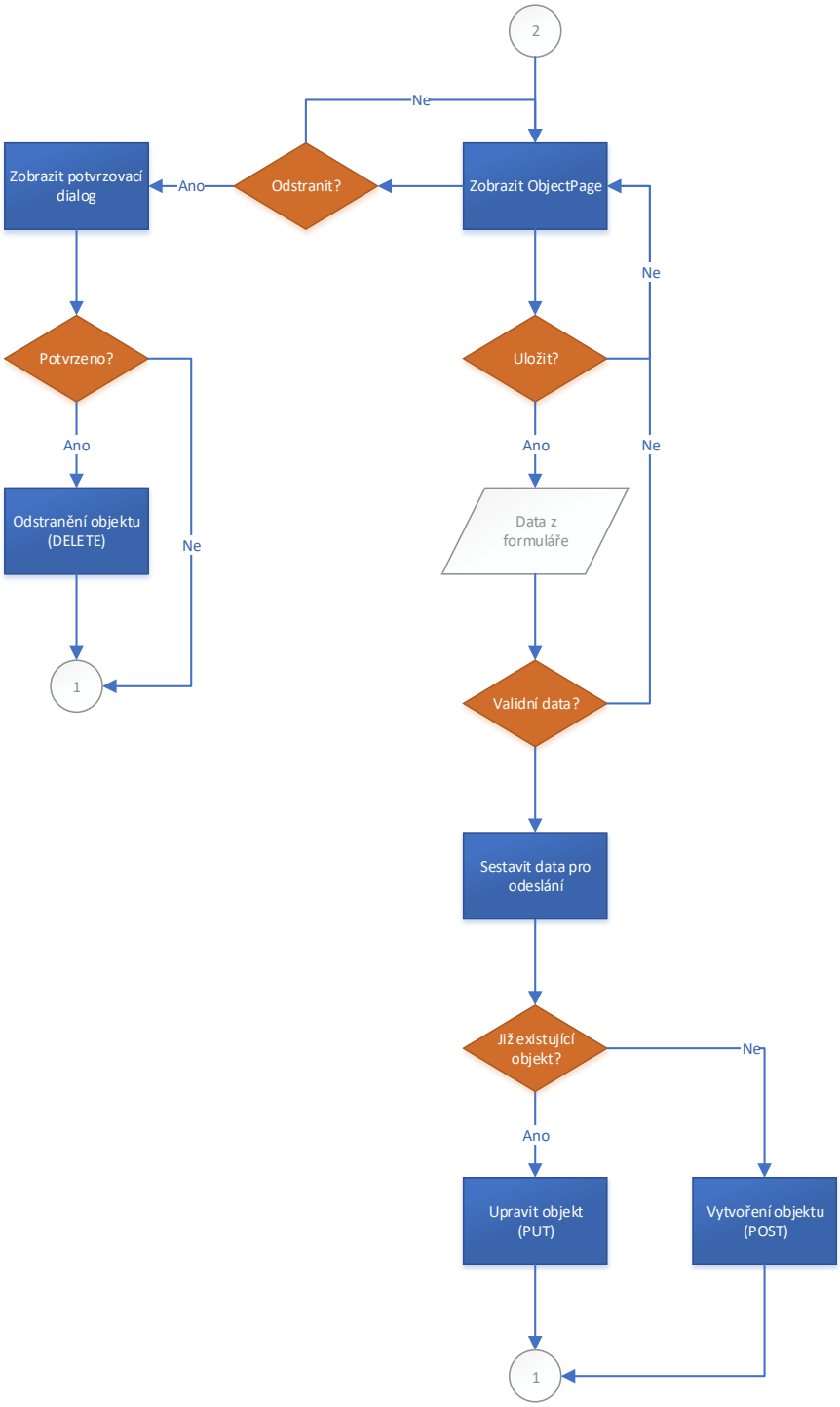
Po přihlášení bude zobrazen tzv. dashboard, který slouží jako rozcestník. Tato „MainPage“ bude obsahovat dlaždice objektů, které lze modifikovat pomocí rozhraní REST. Po kliknutí na dlaždici se uživateli zobrazí seznam objektů (ObjectListingPage) a po kliknutí na konkrétní objekt se zobrazí editační formulář (ObjectPage) se všemi daty objektu. Protože seznamy objektů i objekty samotné jsou si velice podobné, budou tyto stránky generovány automaticky na základě typu objektu. Např. seznam uživatelů může vypadat podobně, jako seznam webových stránek, oba seznamy budou zobrazovat názvy objektů, u uživatelů bude navíc zobrazen ještě např. email. Pro každý objekt proto bude vytvořena šablona položky, ale samotné plnění daty bude probíhat pro všechny objekty stejně.



Obrázek 10: Diagram procesů aplikace - 1. část

Zdroj: Vlastní zpracování

Vraťme se ještě zpět k editačnímu formuláři, ten bude totiž téměř nebo zcela stejný jak pro editaci, tak i pro vytváření nového objektu. Lišit se bude pouze v načítání dat případného editovaného objektu, viditelnosti některých editačních polí u některých objektů a také viditelnosti tlačítka pro odstranění objektu.



Obrázek 11: Diagram procesů aplikace - 2. část

Zdroj: Vlastní zpracování

Po vytvoření či ukončení editace objektu bude uživatel přesměrován zpět na seznam objektů, kde může otevřít další objekt nebo se vrátit zpět k MainPage. Z té lze pak vyvolat odhlášení, které uživatele přesměruje na přihlašovací stránku, nebo zobrazení nastavení aplikace. Podle podmínek Windows Store nesmí aplikace obsahovat ovládací prvky pro zavření aplikace, její ukončení proběhne automaticky v případě, že bude na pozadí (stav suspended) a systému bude docházet volná operační paměť, nebo manuálním zavřením prostřednictvím multitasking manageru.

4.2 VYTVOŘENÍ PROJEKTU

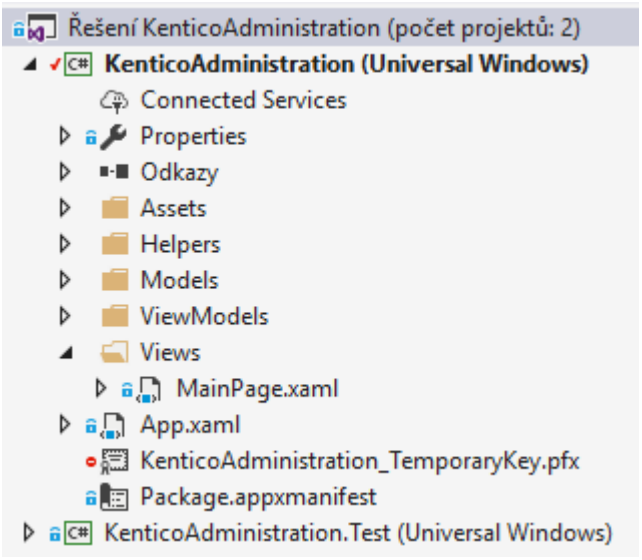
Založení projektu ve Visual Studiu není nic složitého, stačí intuitivně kliknout na Nový projekt a následně vybrat šablonu prázdné UWP aplikace. Aplikace pro Kentico CMS má však být vyvíjena pomocí metodiky Test-Driven Development, bude proto potřeba do nově vytvořeného řešení (Solution) přidat ještě další projekt pro jednotkové testy.

Původně jsem v této práci chtěl používat oblíbený testovací framework nUnit, ten však zatím nepodporuje testování univerzálních aplikací pro Windows 10, proto jsem si musel vystačit s MSTest frameworkem. Ten bývá standardně součástí instalace, ovšem pokud by chyběl, je možné jej doinstalovat přes rozšíření a pluginy. Podobně by se instaloval i zmíněný nUnit framework. Přidání druhého projektu se provede volbou šablony „Aplikace testů jednotek (univerzální pro Windows)“. Je dobré se řídit doporučenými konvencemi a projekt pojmenovat shodně, pouze s příponou „.Test“. Projekt s testy se bude proto jmenovat „KenticoAdministration.Test“.

Po vytvoření obou projektů je potřeba zahrnout projekt aplikace do referencí testovacího projektu, aby testy mohly přistupovat k metodám, které se mají testovat. Z tohoto také vyplývá, že veškeré metody, které budou automaticky testovány, musí být veřejné, aby byly pro testy přístupné. Struktura obou projektů by měla být co nejvíce podobná, pokud tedy vytvořím v projektu aplikace složku Helpers, měla by tato složka být i v testovém projektu. Podobně je to i s názvy jednotlivých souborů, které by se měly jmenovat stejně, respektive se sufixem Tests v projektu s testy.

Nyní si připravíme strukturu projektu pro lepší pozdější orientaci. Ke složce Assets, která obsahuje několik vzorových obrázků použitých v aplikaci, přidáme další složky Views pro soubory stránek, Models pro třídy objektů, ViewModels pro třídy k propojování

aplikační logiky a prezenční vrstvy. Také můžeme vytvořit složku Helpers pro pomocné třídy. Šablona prázdného projektu ve skutečnosti není zcela prázdná, obsahuje stránku App s kódem na pozadí, který se stará o spouštění aplikace a navigaci a prázdnou stránku MainPage. Tato stránka bude zobrazovat data aplikace, měla by být tedy umístěna ve složce Views. Pouhé přesunutí by však vyžadovalo mnohem více práce, než se na první pohled zdá, proto tuto stránku smažeme a vytvoříme novou se stejným názvem ve zmíněné složce. Při pokusu o kompilaci budeme ale upozorněni na chybu v navigaci, protože cílová stránka MainPage neexistuje. Otevřeme proto kód stránky App (App.xaml.cs) a přidáme direktivu `using`, která bude odkazovat do složky Views.



Obrázek 12: Výsledná struktura připraveného projektu

Zdroj: Vlastní zpracování

Tímto jsme dokončili přípravu projektu pro aplikaci a testového projektu.

4.3 NÁVRH DESIGNU

Design aplikace bude odrážet design webového prostředí softwaru Kentico, aby se v ní uživatelé snadno orientovali. Proto nejdříve vytvoříme slovník prostředků (Resouce Dictionary), ve kterém nadefinujeme barvy používané v Kentico CMS/EMS. K těmto definicím pak budeme přistupovat při navrhování jednotlivých ovládacích prvků. Tím zajistíme, že barvy budou vždy odpovídat a navíc, v případě potřeby barvu změnit, ji změníme pouze ve slovníku a změna se projeví ve všech částech aplikace. Podobně lze připravit i číselné hodnoty, textové řetězce, font, viditelnost, velikost okrajů a další.

Vytvoříme tedy slovník prostředků Styles.xaml a umístíme jej do nové složky Common. Do slovníku přidáme barvy v tomto tvaru:

```
<SolidColorBrush x:Key="Orange" Color="#FFF05A22"/>
```

Abychom mohli k barvám přistupovat v rámci celé aplikace, je potřeba zdroje prostředků sloučit s prostředky aplikace. Otevřeme proto soubor App.xaml, do kterého umístíme odkaz na slovník:

```
<Application.Resources>
    <ResourceDictionary>
        <ResourceDictionary.MergedDictionaries>
            <ResourceDictionary Source="Common/Styles.xaml"/>
        </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
</Application.Resources>
```

Podobné slovníky lze později vytvářet i pro šablony ovládacích prvků a dat objektů. Tím je získána možnost používat stejné prvky napříč celou aplikací přitom definice jejich stylů a šablon budeme mít pohromadě.

Nyní se již pustíme do návrhu designu. Začneme pravděpodobně nejjednodušší stránkou aplikace, kterou bude přihlašovací obrazovka. Vytvoříme prázdnou stránku LoginPage a otevřeme ji v prostředí Blend for Visual Studio.

Do prázdné stránky vložíme prvky TextBox, PasswordBox a Button. Význam jednotlivých ovládacích prvků není třeba popisovat, pokud se ale podíváme na náhled stránky, zjistíme, že se všechny prvky překrývají a jsou umístěny v levém horním rohu. Prvky sice můžeme libovolně přesouvat, ale v XAML kódu zjistíme, že se mění vlastnost `Margin`, která představuje odsazení od okrajů stránky. Tímto bychom prvky umístili absolutně.

Absolutní pozicování se však v současných aplikacích využívá výjimečně, protože ne všechna zařízení mají stejně velký displej a prvky umístěné uprostřed obrazovky jednoho zařízení by se např. zobrazovaly v levé horní části většího displeje. Nemluvě o otočení mobilního telefonu na šířku, to by se prvky nemusely zobrazit vůbec, protože by

„přetekly“ mimo viditelnou oblast stránky. Z tohoto důvodu je mnohem vhodnější relativní pozicování. V jazyku XAML existuje několik možností, jak rozmístit prvky po stránce, všemi lze docílit podobného či stejného výsledku. Asi nejpoužívanějšími je Grid (mřížka). Ta rozdělí stránku na více částí (buněk) pomocí řádků a sloupců, které mohou mít pevnou velikost, procentuální nebo automatickou, kdy se rozměry přizpůsobují obsahu.

V našem případě budeme chtít mít v horní části obrazovky logo a uprostřed přihlašovací formulář. Rozdělíme proto mřížku na tři řádky, kdy prostřední řádek se bude přizpůsobovat velikosti formuláře (hodnota `Auto`) a horní a spodní řádek budou mít výšku určenou takzvanou hvězdičkovou notací. Ta představuje, v jakém poměru bude velikost řádků (respektive sloupců) zadaných hvězdičkovou notací. V našem případě budou mít oba dva řádky výšku zadanou hodnotou `1*`. V případě, že by jeden z nich měl velikost `2*`, znamenalo by to, že bude jeho výška dvakrát větší než výška druhého. Vyplněním atributu `Grid.Row` a `Grid.Column` určíme, do kterého řádku a sloupce (číslovaného od nuly) se má prvek umístit. Naše formulářové prvky umístíme do druhého řádku, nastavíme jim tedy parametr `Grid.Row="1"`.

Prvky jsou již na správném místě, ale stále se vzájemně překrývají, místo toho, aby se řadily pod sebe. K tomuto účelu slouží StackPanel, který zobrazuje jeho potomky řazené vertikálně nebo horizontálně. Postačí obalit jím ovládací prvky, které se mají zobrazovat pod sebou. Mimo absolutní pozicování se také nedoporučuje zadávání rozměrů v pevných hodnotách. Vhodnější je použití kombinací hodnoty `Auto`, parametrů `MinWidth`, `MaxWidth`, `MinHeight` a `MaxHeight` a zarovnání. K běžným hodnotám zarovnání patří v XAML i hodnota `Stretch`, která daný prvek roztáhne přes celou šířku, respektive výšku nadřazeného objektu. Vhodně použitými parametry lze docílit požadovaného designu, který se umí přizpůsobit změnám velikosti nadřazených objektů či celého displeje. Díky tomu lze navrhovat jednotné uživatelské prostředí pro mobilní telefony, tablety či desktopové počítače.

Vzhled ovládacích prvků lze měnit pomocí jejich vlastností, ovšem později bychom zjistili, že některé vlastnosti designu měnit nelze. V případě tlačítek se jedná například o vzhled při jejich stisknutí či deaktivování, u textových polí pak orámování v případě

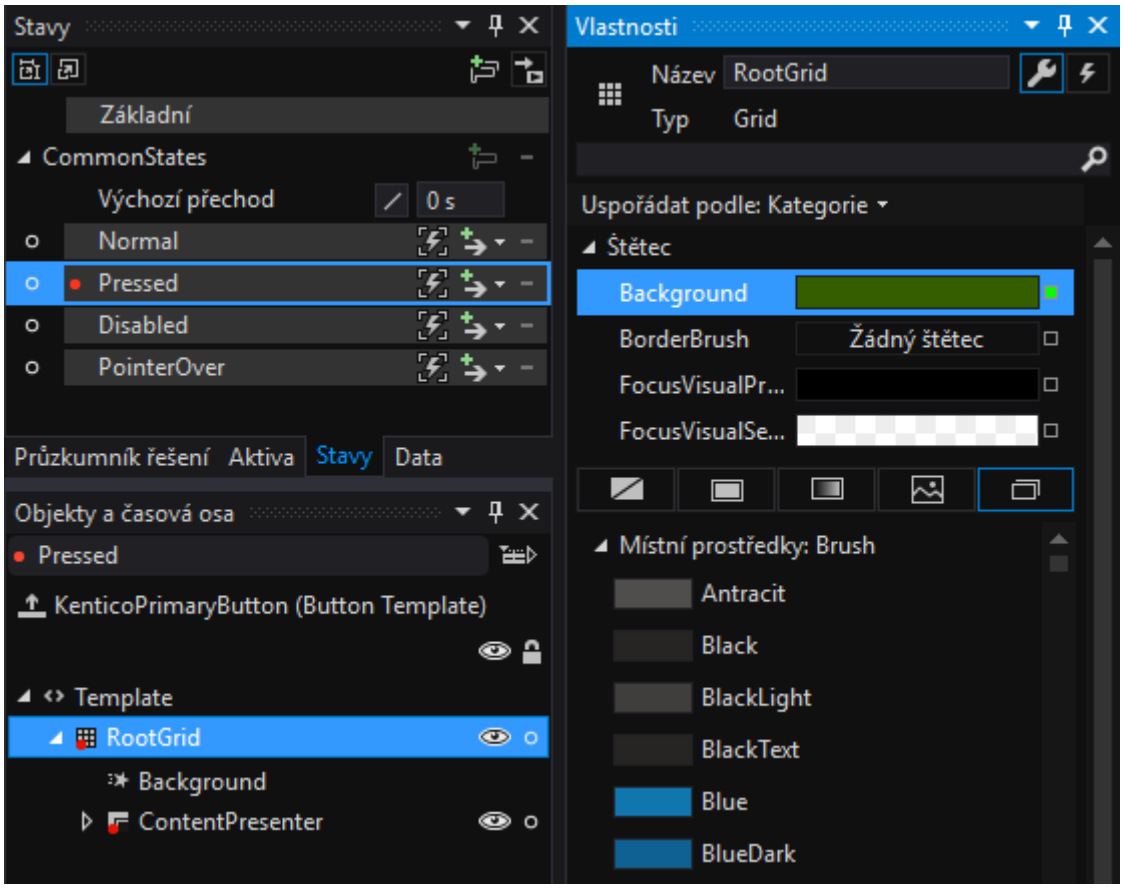
aktivního pole apod. Na následujícím obrázku lze tyto rozdíly vidět – nahoře je neaktivní prvek a dole aktivovaný.

Obrázek 13: Porovnání neaktivních a aktivních prvků po nastavení vlastností

Zdroj: Vlastní zpracování

Návrhové prostředí Blend nám však velmi jednoduše umožňuje vytvořit si vlastní šablonu. Stačí na prvek kliknout pravým tlačítkem, zvolit Upravit šablonu a poté Vytvořit kopii. Tímto se vygeneruje XAML kód popisující veškeré objekty, stavy a styly daného prvku. Kódu bývá většinou hodně a je těžké se v něm vyznat, pokud však zobrazíme v okna „Stavy“ a „Objekty a časová osa“, můžeme styly stavů těchto objektů editovat bez zásahu do kódu.

V prvním zmíněném okně nalezneme všechny nadefinované stavy daného prvku. U tlačítka to jsou Normal, Pressed, Disabled a PointOver, které definují pouze rozdílné vlastnosti oproti Základnímu stavu. V okně Objekty a časová osa se zobrazují objekty, ze kterých je daný ovládací prvek sestaven a v případě zvolení jiného než základního stavu, bude u jednotlivých upravených objektů zobrazena i změněná vlastnost. U námi vytvořeného tlačítka pro přihlášení vybereme RootGrid a přepneme se do stavu Pressed. U vybrané mřížky se nyní zobrazí červená tečka signalizující nějaké změněné vlastnosti, po rozbalení zjistíme, že v tomto stavu je upraveno pozadí. Pokud nyní zvolíme novou barvu pozadí RootGrid, změna se provede pouze ve vybraném stavu Pressed, ostatní stavy zůstanou beze změny. Úpravou dalších vlastností můžeme upravit tlačítko přesně podle našeho přání, a dokonce můžeme měnit i samotnou strukturu prvku.



Obrázek 14: Úprava šablony tlačítka

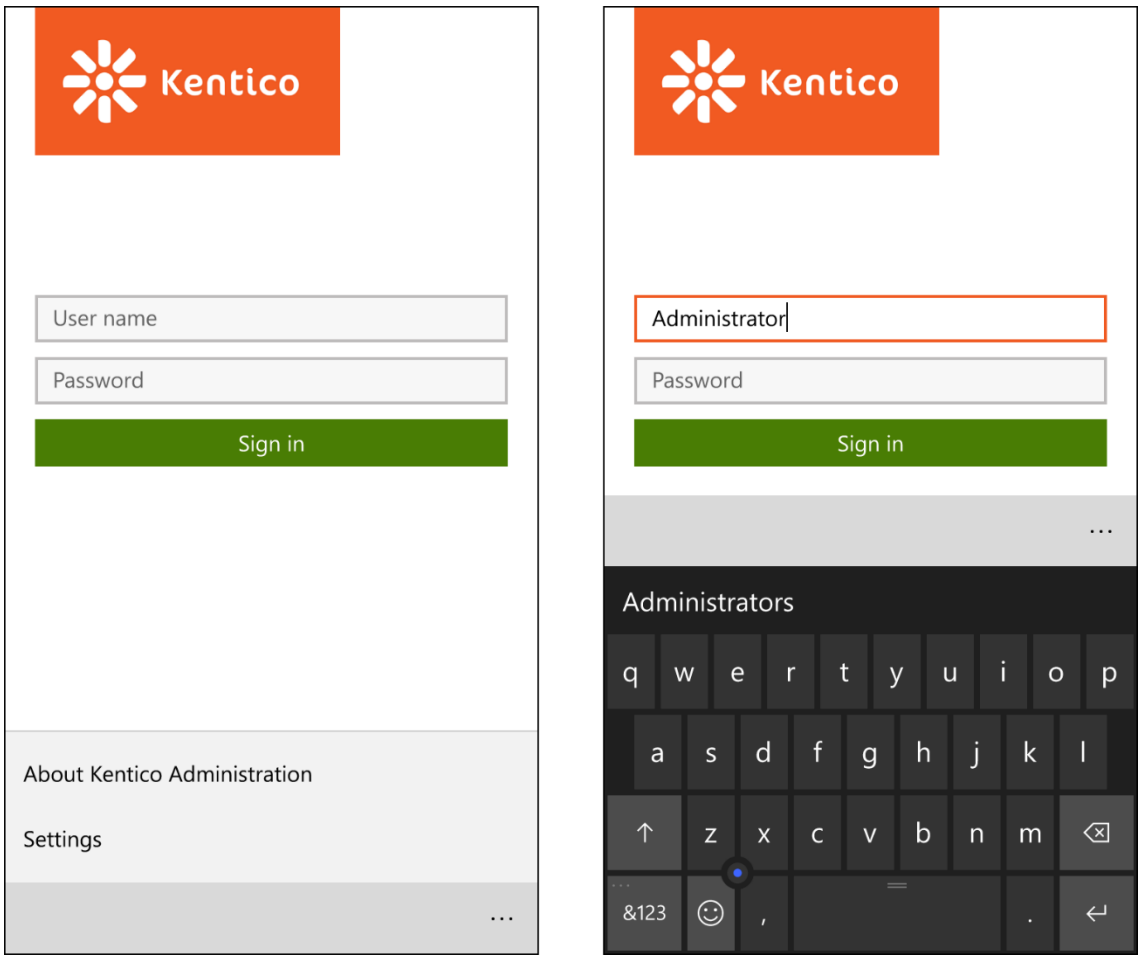
Zdroj: Vlastní zpracování

Takto vytvořený styl lze použít na kterékoli tlačítko na stránce, a samozřejmě, pokud šablonu přesuneme do zdrojů aplikace, na jakékoli tlačítko v celé aplikaci. Tímto se otevírají široké možnosti pro originální stylizování aplikací.

Dalším krokem bude vložení loga. Obrázek musí být nejdříve vložen do projektu, obvykle do složky Assets. Poté vložíme prvek Image, jemuž nastavíme vlastnost `Source`, která udává relativní cestu k souboru v rámci projektu.

Posledním krokem bude přidání tlačítek pro zobrazení nastavení aplikace a stránky o aplikaci. Tato tlačítka nebudeme vkládat přímo do stránky, jako tomu bylo u předešlého, ale vložíme je do spodní lišty aplikace, která je pro to určena. Prvek BottomAppBar obsahuje dvě skupiny pro vkládání tlačítek, první z nich je PrimaryCommands a slouží k přidání tlačítek v podobě ikon na spodní lištu. Do SecondaryCommands se vkládají tlačítka, která se zobrazí až po kliknutí na symbol tří

teček. V našem případě vložíme obě tlačítka až do druhé skupiny, protože nejsou příliš důležitá a uživatel je nebude často používat.

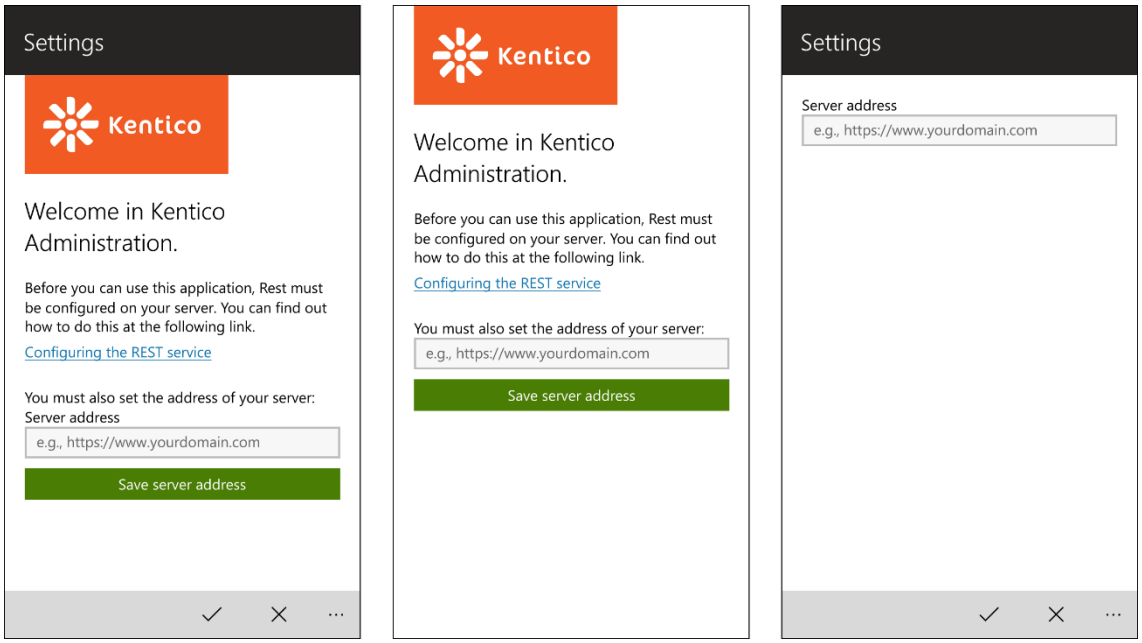


Obrázek 15: Výsledný design přihlašovací stránky

Zdroj: Vlastní zpracování

Nyní můžeme přejít na návrh stránky pro nastavení. Tato stránka bude zajímavá tím, že bude sloužit jako vstupní obrazovka při úplně prvním spuštění. Jak bylo dříve zmíněno, aplikace komunikuje se serverem, jehož adresa není předem známa, proto je potřeba, aby ji uživatel nejdříve zadal. Kdybychom však zobrazili strohou stránku s nastavením, mohlo by to uživatele odradit či překvapit. Bylo by tedy vhodné uživatele nějak uvítat a sdělit mu co má pro nastavení aplikace udělat. Při pozdějším zobrazení této stránky by však mohly být některé tyto informace matoucí. Nabízí se možnost vytvoření dvou stránek s nastavením, ale to je zcela zbytečné. Využijeme totiž vlastní stavy stránky a podle toho, zda se jedná o prvotní spuštění, budeme uživateli zobrazovat jeden z nich.

Nejdříve navrhneme stránku se všemi prvky, které se budou zobrazovat v obou variantách. Poté vytvoříme dva stavy a provedeme skrytí prvků, které nemají být v daném stavu vidět. Volbu požadovaného stavu bude mít kód na pozadí této stránky.



Obrázek 16: Stránka s nastavením v různých stavech

Zdroj: Vlastní zpracování

Poslední stránkou, jejíž návrh budu rozebírat je MainPage. Ta bude zobrazovat dlaždice představující jednotlivé objekty, které jdou touto aplikací číst či upravovat. Začneme však s hlavičkou. Na předchozím obrázku je vidět tmavé záhlaví, které koresponduje se záhlavím webové administrace Kentico. To bude vidět již na každé další stránce a bude podávat informaci o tom, jaký objekt či kolekce objektů je právě upravována. Vytvoříme si proto styl pro StackPanel, který bude tuto hlavičku představovat a který bude použit i na ostatních stránkách. Na MainPage se však ještě žádný objekt neupravuje, proto můžeme do hlavičky umístit selektor webových stránek (sites), jejíž objekty se mají zobrazovat, podobně, jako je tomu v záhlaví webové verze. K tomu je nejvíce vhodný prvek ComboBox. Po jeho přidání do stránky však zjistíme, že neobsahuje žádné položky, a proto by bylo obtížné jej stylovat. Sice je možné položky přidat ručně, ale později, až bychom chtěli načítat skutečná data ze serveru, by musely být odstraněny a pro případné úpravy zase přidány atd. Lepší bude použít vkládání dat z kódu, vytvoříme proto ve složce ViewModels třídu `SitesViewModel` s následujícím kódem:

```
public class SitesViewModel
    // Veřejně přístupná vlastnost (property) s kolekcí všech stránek
    public ObservableCollection<Site> AllSites { get; set; }
    // Veřejně přístupná vlastnost s aktuálně vybranou stránkou
    public Site SelectedSite { get; set; }

    // Konstruktor pro vytvoření instance ViewModelu
    public SitesViewModel()
    {
        AllSites = new ObservableCollection<Site>();
        LoadSites()
    }

    public void LoadSites()
        // Pokud je aktivní návrhářský mód
        if (Windows.ApplicationModel.DesignMode.DesignModeEnabled)
        {
            // Vytvoří seznam fiktivních stránek
            List<Site> allSites = new List<Site>()
            {
                new Site("Site 1"),
                new Site("Site 2")
            };

            // Vloží do vlastnosti AllSites každou stránku ze seznamu
            foreach (Site site in allSites)
            {
                AllSites.Add(site);
            }

            // Vloží do vlastnosti SelectedSite první položku z kolekce
            SelectedSite = AllSites.First();
        }
        else
        {
            // TODO: Kód, který načte skutečná data ze serveru
        }
    }
}
```

Tato třída bude přiřazena do datového kontextu ComboBoxu, což způsobí, že při načtení stránky MainPage se zavolá konstruktor této třídy, čímž se naplní uvedené vlastnosti na prvních řádcích. Data z nich se pak budou tzv. bindovat do prezenční vrstvy. Všimněme si však ještě řádku s `DesignModeEnabled` – tato podmínka umožňuje načítat vkládat do prezenční vrstvy jiná data při návrhu a jiná při běhu aplikace. V návrháři proto vidíme vzorová data, která nám pomohou při stylování prvků, a přitom se v aplikaci zobrazují reálná data bez jakéhokoli zásahu do kódu. Pro úplnost dodám, že ve složce Models byla vytvořena třída `Site`, která obsahuje jednoduchý konstruktor pro vytvoření instance třídy, který byl využit při generování vzorových dat. Její kód je uveden níže. Takováto struktura odpovídá MVVM architektuře.

```
public class Site
{
    // Veřejná vlastnost se třídy
    public string SiteName { get; set; }
    // Konstruktor
    public Site(string siteName) { SiteName = siteName; }
}
```

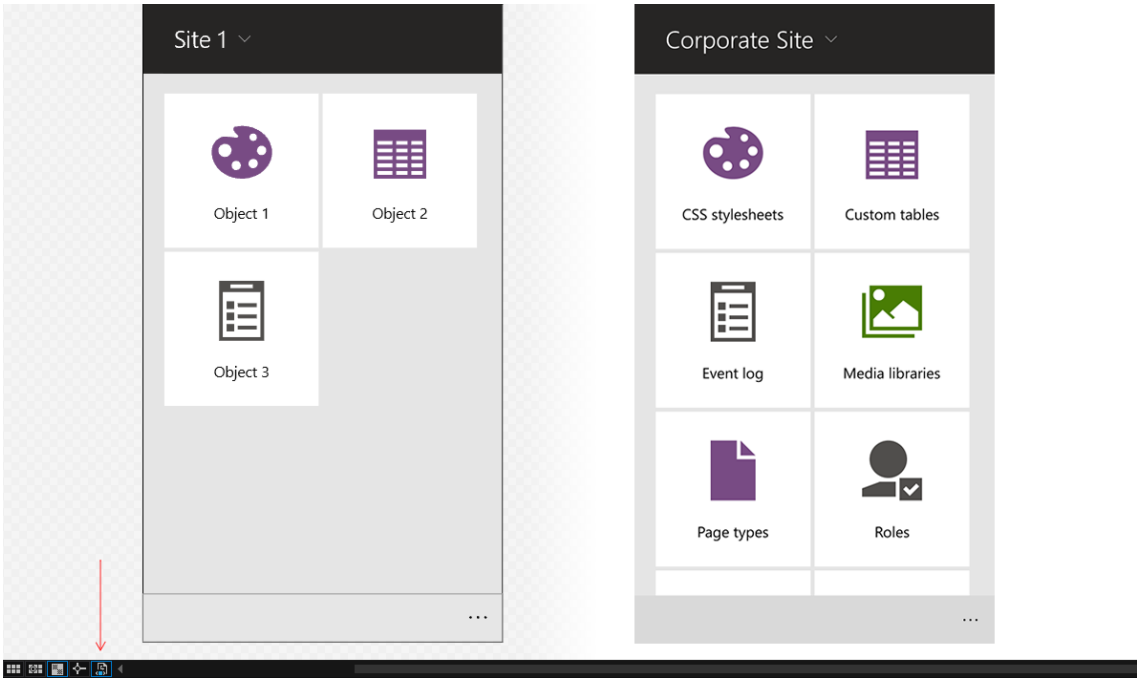
Bindování dat poté probíhá velmi snadno a to tak, že v ComboBoxu nastavíme DataContext na třídu `SitesViewModel`, vytvoříme vazbu vlastnosti ItemsSource na vlastnost `AllSites` a vazbu mezi vlastnostmi `SelectedItem` v ComboBoxu a třídy `SitesViewModel`. Obdobně je potřeba vytvořit vazbu mezi `SiteName` a prvkem, který tuto vlastnost bude vypisovat. Takový prvek však zatím v našem selektoru neexistuje, můžeme jej však snadno vytvořit kliknutím na něj pravým tlačítkem a volbou „Upravit další šablony“ a poté „Upravit generované položky (ItemTemplate)“. Do nové šablony lze vkládat libovolné ovládací prvky, nám však postačí obyčejný TextBlock, který bude onu hodnotu vlastnosti `SiteName` jednotlivých položek zobrazovat. Vše lze nastavit pomocí návrháře Blend nebo zápisem podobného XAML kódu:

```
<StackPanel.Resources>
    <DataTemplate x:Key="ComboBoxItems">
        <TextBlock Text="{Binding SiteName}" />
    </DataTemplate>
</StackPanel.Resources>
```

```
<ComboBox ItemsSource="{Binding AllSites}" SelectedItem="{Binding
    SelectedSite}" ItemTemplate="{StaticResource ComboBoxItems}">
    <ComboBox.DataContext>
        <viewModels:SitesViewModel/>
    </ComboBox.DataContext>
</ComboBox>
```

Po sestavení řešení a povolení kódu projektu v návrháři (označeno šipkou na následujícím obrázku) bude ComboBox zobrazovat vzorová data a my můžeme pokračovat v jeho stylování. Díky těmto úpravám máme také přípravu pro zobrazování reálných dat ze serveru. Tím se však budeme zabývat až v další části práce.

Po dokončení designování selektoru pro výběr stránek je potřeba navrhnout zobrazování dlaždic. Pro tento účel existuje GridView. Ten se podobá StackPanelu s tím rozdílem, že se v něm jednotlivé prvky za sebe řadí v obou směrech, tedy z leva doprava a poté shora dolů. Dlaždice by měla vypadat podobně, jako ve webové verzi administrace, proto bude do projektu potřeba umístit obrázkové písmo, pomocí kterého budou zobrazovány ikony dlaždic. Mimo to se na dlaždici bude zobrazovat i název funkcionality a dlaždice mohou být různě barevně laděny.

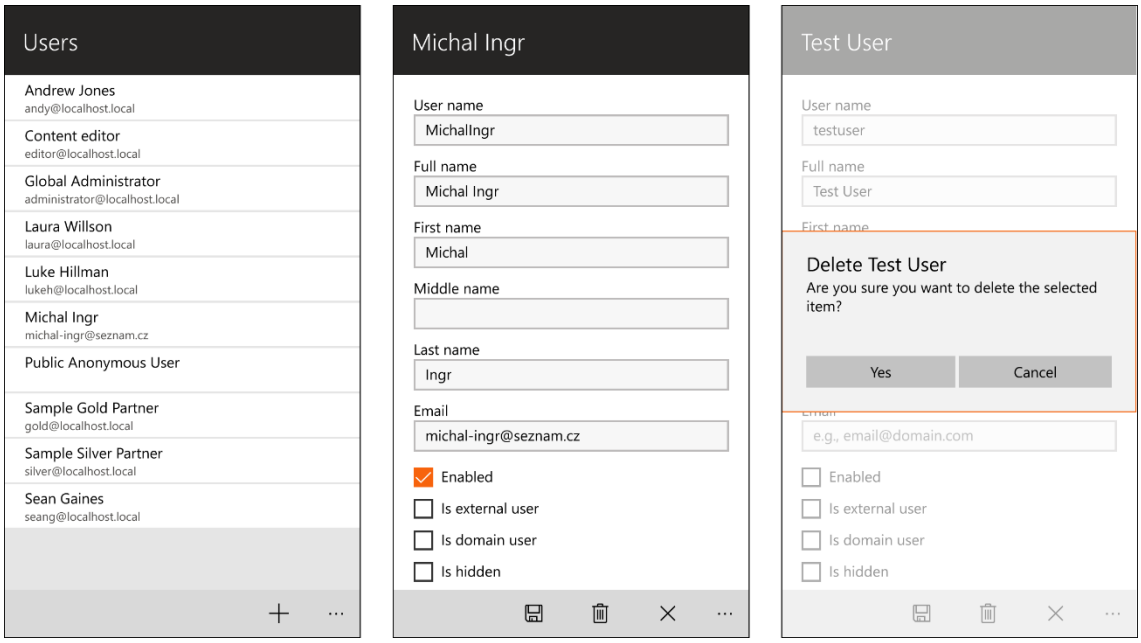


Obrázek 17: Porovnání vzorových dat v návrháři a reálných dat v aplikaci

Zdroj: Vlastní zpracování

Pro účely stylování vytvoříme podobně jako u ComboBoxu třídu, která bude popisovat jednotlivé dlaždice, bude tedy obsahovat vlastnosti pro barvu, ikonu a název. Samozřejmě bude obsahovat i konstruktor, pomocí něhož ve ViewModelu budeme generovat vzorová data.

Ostatní stránky se příliš neliší od již popsaných, proto je zmíním pouze okrajově. Stránka se seznamem objektů ObjectListPage bude obsahovat StackPanel a stránka konkrétního objektu ObjectPage bude zobrazovat editační formulář s několika textovými bloky (TextBlock) a formulářovými poli (TextBox), případně i zaškrtávacími tlačítky (CheckBox) a rozevíracími seznamy (ComboBox).



Obrázek 18: Design ObjectListPage a ObjectPage s dialogem

Zdroj: Vlastní zpracování

4.4 PROGRAMOVÁNÍ FUNKCIONALITY

V rámci této práce má být aplikace vyvíjena pomocí metodiky Test-Driven Development. V předešlé části byla nastíněna funckionalita selektoru stránek, při němž byly vytvořeny základy tříd Sites a SitesViewModel. V druhé jmenované jsme vytvořili metodu `LoadSites()`, kterou je již možné testovat, vytvoříme proto třídu SitesViewModelTests v projektu pro testy. Dodržíme při tom stejnou strukturu, jako u projektu aplikace, bude proto shodně umístěn do složky ViewModels. První test bude kontrolovat, že po zavolání

konstruktoru se načtou stránky z externího zdroje. Kód testovací třídy může vypadat následovně:

```
[TestClass]
public class SitesViewModelTests
{
    [TestMethod]
    public void LoadAllSites_CountOfLoadedSitesIsCorrect()
    {
        var viewModel = new SitesViewModel();
        Assert.AreEqual(???, viewModel.AllSites.Count);
    }
}
```

Zde se ale dostáváme k prvnímu problému. Protože data načítáme z externího zdroje, může být jejich počet pokaždé jiný. Navíc server může být nedostupný či se může změnit jeho adresa. Je tedy potřeba nějak zajistit, že data budou pokaždé stejná. Řešením je tzv. mockování (imitování, podstrkávání) dat namísto jejich načítání z externího zdroje.

Vytvoříme proto rozhraní, které bude definovat metodu pro načítání stránek ze serveru a umístíme jej do nové složky DataProviders. Při operacích, které by mohly trvat déle než 50 ms je doporučeno používat asynchronní metody, aby se nezasekávalo uživatelské rozhraní a uživatel nenabyl dojmu, že aplikace přestala odpovídat. Proto místo návratového typu `List<Site>` budeme vracet úlohu, tedy `Task<List<Site>>`.

```
public interface ISitesDataProvider
{
    Task<List<Site>> GetAllSitesAsync();
}
```

Nyní je potřeba ve view-modelu nějak určit, kdy se mají načítat testovací data a kdy reálná. To uděláme pomocí konstruktoru `SitesViewModel()`, kterému přidáme parametr, jehož typ bude vytvořené rozhraní a ve kterém budeme předávat provider. Konstruktor tohoto providera uloží do privátní proměnné a metoda pro načtení dat pak bude volána na něm. Na následující ukázce kódu jsou změny vyznačeny tučně a některé části dříve uvedeného kódu jsou vynechány.

```
private ISitesDataProvider dataProvider;
public SitesViewModel(ISitesDataProvider dataProvider)
{
    this.dataProvider = dataProvider;
    AllSites = new ObservableCollection<Site>();
    LoadSites ();
}

public async void LoadSites ()
{
    if (Windows.ApplicationModel.DesignMode.DesignModeEnabled) { ... }
    else
    {
        List<Site> allSites = await dataProvider.GetAllSitesAsync();
    }
}
```

Dále vytvoříme v testovacím projektu třídu `SiteDataProviderMock`, která bude implementovat rozhraní `ISitesDataProvider` a její metoda `GetAllSiteAsync()` bude později vracet testovací data. „Později“ uvádím záměrně, protože podle správných technik TDD musíme nejdříve napsat (a také spustit) test, který neprochází a teprve poté psát samotnou funkcionalitu. Spuštění testů se provádí kombinací klávesových zkratek CTRL+R, A, případně kliknutím na Spustit testy v nabídce Test.

I když se spouštění testů v tomto případě zdá zbytečné, jedná se o dobrý návyk, který by měl každý programátor dodržovat. Není totiž příliš těžké napsat omylem takový test, který bude procházet vždy, i když by neměl. Jde tedy o jakousi primitivní kontrolu, že test není napsán špatně.

Po spuštění testu a zjištění, že neprochází, můžeme dopsat funkcionalitu metody.

```
public class SitesDataProviderMock : ISitesDataProvider
{
    public Task<List<Site>> GetAllSitesAsync()
    {
        TaskCompletionSource<List<Site>> result = new
            TaskCompletionSource<List<Site>>();
        var sites = new List<Site>()
```

```
        {
            new Site("testsite1", "Test site 1"),
            new Site("testsite2", "Test site 2")
        };
        result.SetResult(sites);
        return result.Task;
    }
}
```

Nyní přichází na řadu opětovné spuštění testu, který by měl již procházet. Tímto testem jsme otestovali, že data, která budou přijata z externího zdroje budou správně vložena do vlastnosti `AllSites` ve view-modelu. Data jsme však mockovali a je tedy potřeba dopsat metody pro načítání dat ze serveru a jejich serializaci do objektů.

Připravme si proto správnou strukturu modelu Sites, která bude odpovídat datům načítaných ze serveru ve formátu JSON, jejichž příklad je uveden níže. Všimněme si, že jde o několik vnořených polí objektů. Kořenový objekt `cms_sites` obsahuje pole objektů, z nichž první `CMS_Site` obsahuje pole objektů představujících námi požadované objekty (stránky), druhý pak obsahuje pole, jehož jediným objektem je údaj o počtu záznamů.

```
{
      "cms_sites":
      [{
            "CMS_Site":
            [{
                  "SiteDisplayName": "Corporate Site",
                  "SiteName": "CorporateSite",
                  "SiteID": 2,
                  ...
            }]
      }, {
            "TotalRecords":
            [{
                  "TotalRecords": "2"
            }]
      }]
}
```

Počet záznamů se vyskytuje v každé odpovědi serveru, proto vytvoříme třídu `TotalRecord`, která bude obsahovat jednu vlastnost `TotalRecords`. Také bude obsahovat prázdný konstruktor, protože serializace objektů probíhá tak, že serializátor nejdříve vytvoří instanci objektu a pak jeho vlastnostem přiřazuje hodnoty. Pokud by konstruktor chyběl, nemohla by být instance objektu vytvořena a serializace by skončila výjimkou.

Vedle třídy `Site`, do které doplníme veškeré vlastnosti získávané ze serveru, vytvoříme ještě další třídy tak, aby odpovídaly JSON datům. Budeme muset vytvořit obalující třídu `Sites`, která bude obsahovat seznam (vylepšené pole) objektů. Tento objekt bude definovaný další s vlastnostmi, z nichž jedna bude obsahovat seznam objektů typu `Site` a druhá seznam objektů typu `TotalRecord`. U každé třídy je třeba nezapomenout na prázdný konstruktor. Tímto docílíme stejné struktury, jako je obsažena ve zprávách ze serveru. Celou strukturu objektu lze nalézt v příloze.

K implementaci načítání dat ze serveru a jejich serializaci vytvoříme soubor se třídou Repository. Tato třída bude statická, protože repositář bude vždy pouze jeden v rámci celé aplikace. Protože již máme připravena objekt, na který mají být přijatá data převedena a také známe přesnou strukturu dat, nejdříve vytvoříme metodu, která bude data serializovat.

Vytvořme tedy prázdnou metodu `SerializeData()`, která bude přijímat data načtená ze serveru ve formě Stream a vracet seznam objektů typu Site.

```
public static List<Site> SerializeData(Stream stream) { return null; }
```

K této metodě nyní napíšeme test. Vytvořme proto soubor RepositoryTests a také ve složce DataMock soubor cms_sites.txt, jehož obsahem budou vzorová serverová data. V testovací metodě se tato data načtou a převedou na typ Stream, což je typ dat v odpovědi serveru. Dále se připraví objekty, na které mají být data převedena, a nakonec se zavolá metoda `SerializeData()`. Očekávaná kolekce objektů se porovná s výstupem metody. Kolekce se porovnávají také metodou `AreEqual()`, avšak umístěné ve třídy `CollectionAssert`.

```
[TestMethod]
public void SerializeAllSites_AllSitesAreSerialized()
{
    Stream stream = new FileStream("DataMock/cms_sites.txt",
                                   FileMode.Open, FileAccess.Read);
    List<Site> allSitesExpected = new List<Site>
    {
        new Site() { SiteID = 1, SiteName = "DancingGoat", ... },
        new Site() { SiteID = 2, SiteName = "CorporateSite", ... }
    };
    List<Site> allSites = Repository.SerializeData(stream);
    CollectionAssert.AreEqual(allSitesExpected, allSites);
}
```

Bohužel metoda `AreEqual()` u kolekcí nefunguje tak, jak bychom předpokládali. Namísto toho, aby porovnala zda dvě různé kolekce obsahují objekty se stejnými hodnotami vlastností, vrací metoda `AreEqual()` hodnotu `true` pouze v případě, že se jedná o stejné kolekce se stejnými instancemi objektů. Existuje však řešení, jak kolekce porovnávat a tím je přepsání metody `Equal()`. Než se tedy pustíme do psaní testu na serializaci, vytvoříme ještě testy na porovnávání objektů i odpovídající implementaci.

Vytvoříme třídu `ObjectsComparer` s prázdnou metodou `AreObjectsEqual()` ve složce Helpers a obdobně testovací třídu `ObjectsComparerTests`. První testovací metoda vytvoří dva různé objekty se stejnými hodnotami vlastností. Poté porovná oba objekty metodou, kterou budeme vzápětí implementovat.

```
[TestMethod]
public void TwoObjectsWithSameProperties_ObjectsAreEqual()
{
    Site site1 = new Site() { SiteName = "Test", SiteID = 5 };
    Site site2 = new Site() { SiteName = "Test", SiteID = 5 };
    Assert.IsTrue(ObjectsComparer.AreObjectsEqual(site1, site2));
}
```

Můžeme rovnou přidat ještě obdobný test, který však bude porovnávat dva objekty s různými hodnotami vlastností a bude předpokládat, že porovnání vrátí hodnotu `false`.

Po ověření, že testy neprocházejí, můžeme přejít k implementaci. U této statické metody použijeme generické typy, abychom mohli metodu později využívat na různé objekty, ale přitom se nepřipravili o možnost kontroly typů již přímo v editoru či při kompilaci kódu. Metoda tedy bude definována takto:

```
AreObjectsEqual<T>(T object1, T object2)
```

kde `T` značí generický typ a platí, že typ obou objektů musí být stejný. Metoda nejdříve zjistí, zda jsou porovnávané objekty instancemi a pokud ano, pokračuje zjištěním typu a načtením všech jeho vlastností. Poté už stačí porovnat jednotlivé hodnoty vlastností a v případě výskytu první rozdílné hodnoty ukončit cyklus a rovnou vrátit hodnotu `false`.

```
public static bool AreObjectsEqual<T>(T object1, T object2)
{
    if (object1 == null || object2 == null)
        return false;

    foreach (var property in object1.GetType().GetProperties())
    {
        var property1 = property.GetValue(object1);
        var property2 = property.GetValue(object2);

        if (property1 == null && property2 == null)
            continue;

        if (property1.ToString() != property2.ToString())
            return false;
    }
    return true;
}
```

Tuto metodu budeme volat ve třídě `Site`, kde přepíšeme implementaci metody `Equal()`.

```
public override bool Equals(object obj)
{
    return ObjectsComparer.AreObjectsEqual(this, (Site)obj);
}
```

Nyní se vraťme k původnímu testu porovnávání kolekcí, který je nyní kompletní. Po spuštění testů dostáváme očekávaný výsledek, tedy neprocházející právě tento test. Provedeme tedy implementaci metody, která na rozdíl od testu, bude v tomto případě velice krátká.

```
public static List<Site> SerializeData(Stream stream)
{
    var serializer = new DataContractJsonSerializer(typeof(Sites));
    Sites rootObject = (Sites)serializer.ReadObject(stream);
    return rootObject.Cms_Objects.First().Cms_Object;
}
```

Nyní zbývá dopsat metodu pro získání reálných dat ze serveru. Tuto metodu již jednotkově testovat nebudeme z důvodů zmíněných dříve. Ve skutečnosti se bude jednat o dvě metody, jedna z nich však bude privátní a bude obsahovat kód, který by byl při načítání ostatních objektů ze serveru totožný. Stahování dat bude probíhat asynchronně, pomocí třídy `HttpClient`, která již obsahuje asynchronní metody pro zasílání požadavků na server.

```
public static async Task<List<Site>> GetAllSitesAsync()
{
    if (allSites == null)
    {
        Sites sites = await GetRootObject<Sites>("cms.site") as Sites;
        allSites = sites.Cms_Objects.First().Cms_Object;
    }
    return allSites;
}
```

Tato metoda nejdříve zkontroluje, zda data již nebyla ze serveru dříve načtena a uložena do proměnné `allSites`. Pokud ano, rovnou obsah této proměnné vrátí. V opačném případě zavolá metodu `GetRootObject()`, která opět používá generické typy a předáme adresu požadovaného seznamu objektů. Ta poté ověří, zda již nebyla vytvořena instance http klienta, pokud ne, bude klient vytvořen s patřičnými vlastnostmi, jakými jsou `BaseAddress` představující adresu serveru a `Authorization`, což jsou kódované přihlašovací údaje. Obě tyto hodnoty získáme od uživatele při spuštění aplikace.

Kódování přihlašovacích údajů je hned dalším adeptem na unit test a následnou implementaci. Prozatím však použijeme proměnnou, ve které bude uložen již zakódovaný textový řetězec. Na závěr metoda zavolá dříve implementovanou metodu pro serializaci objektu, jejíž návratovou hodnotu vrací i tato metoda.

```
private static async Task<object> GetRootObject<T>(string requestPath)
{
    if (client == null)
    {
        client = new HttpClient() { BaseAddress = baseAdress };
        client.DefaultRequestHeaders.Authorization = authorization;
    }
    Stream stream = await
        client.GetStreamAsync($"rest/{requestPath}?format=json");
    return SerializeObject<T>(stream);
}
```

Další úpravou, kterou je nutné udělat, je vytvoření providera pro načítání reálných dat a jeho zavolání při navigaci na stránku MainPage, protože zde se vyskytuje ComboBox pro výběr stránek.

Vytvoříme třídu `SitesDataProvider` ve složce s rozhraním tohoto providera a rozhraní ve třídě implementujeme.

```
public class SitesDataProvider : ISitesDataProvider
{
    public Task<List<Site>> GetAllSitesAsync()
    {
        return Repository.GetAllSitesAsync();
    }
}
```

V kódu stránky MainPage přepíšeme metodu `OnNavigatedTo()`, která se spouští po zobrazení dané stránky v aplikaci, tedy kdy je na ni navigováno. V této metodě přiřadíme ComboBoxu datový kontext vázaný na instanci `SiteViewModel`. Ten bude vytvořen s konstruktorem, v jehož parametru bude předána instance providera. Tím docílíme toho, že v aplikaci budou načítána reálná data.

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    base.OnNavigatedTo(e);
    var dataProvider = new SitesDataProvider();
    siteComboBox.DataContext = new SitesViewModel(dataProvider);
}
```

Po spuštění aplikace nás však čeká malé překvapení. Data se sice načetla, ale není vybrán žádná stránka jako výchozí, přitom v kódu jeho výběr a přiřazení do `SelectedItem` provádíme.

Obrázek 19: Do UI nejsou propagovány změny některých vlastností

Zdroj: Vlastní zpracování

Je to dáno tím, že se uživatelské rozhraní zobrazí dříve, než se načtou data vlastností ze serveru, na které jsou prvky rozhraní bindovány. Prezenční vrstva se nedozví o tom, že se hodnoty vlastností změnily a prvky neaktualizuje. Výjimku však tvoří `ObservableCollection`, která je pro tuto situaci určena při změně jejího obsahu sama vyvolá událost změny. U ostatních vlastností tuto událost musíme vyvolat sami, a to pomocí implementace rozhraní `INotifyPropertyChanged`. Tím ve třídě přibyde událost typu `PropertyChangedEventHandler`, pomocí které zaregistrujeme změnu vlastnosti `SelectedSite` v případě, že bude tato vlastnost změněna. K tomu využijeme úplný zápis vlastnosti, kde definujeme procedury v případě čtení (`get`) a zápisu (`set`) její hodnoty. Při zápisu hodnoty nejdříve zkontrolujeme, zda je hodnota jiná než předchozí. Pokud ano, hodnotu aktualizujeme a zaregistrujeme změnu vlastnosti.

```
public class SitesViewModel : INotifyPropertyChanged
{
    private Site selectedSite;
    public Site SelectedSite
    {
```

```
        get { return selectedSite; }
        set
        {
            if (selectedSite != value)
            {
                selectedSite = value;
                PropertyChanged?.Invoke(this,
                    new PropertyChangedEventArgs(nameof(SelectedSite)));
            }
    }
    public event PropertyChangedEventHandler PropertyChanged;
    ...
```

Tímto je načítání dat ze serveru do ComboBoxu dokončeno včetně testů. Třídy, které byly v této části práce popsány jsou v celém znění uvedeny v přílohách. Jejich metody lze ve většině případů použít pro načítání dalších objektů. Vždy je zapotřebí vytvořit model objektu, který odpovídá struktuře dat přijatých ze serveru, tato data serializovat a poté načíst v daném view-modelu, vše je však natolik podobné, že by podrobnější popisování těchto postupů bylo zbytečné.

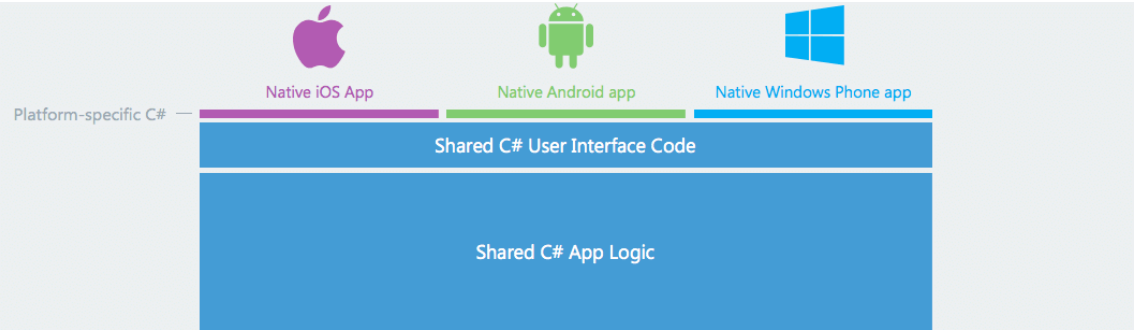
4.5 MOŽNOSTI MULTIPLATFORMNÍHO VÝVOJE

Na začátku práce bylo zmíněno, že bude vyvíjena tzv. univerzální aplikace. Lze ji totiž spustit jak v mobilním zařízení, tak i v běžném počítači či tabletu. Jak je to ale s jinými platformami? Na zařízení s Android nebo iOS tato aplikace spustit nepůjde. Existují však frameworky, pomocí nichž lze vyvíjet opravdu univerzální aplikace.

Pokud by byla aplikace vyvíjena v HTML/CSS/JS, nabízí se velmi oblíbený framework Apache Cordova. Aplikace vytvořená pomocí něj lze pak spustit na všech třech zmíněných platformách, ale dokonce i na BlackBerry, Ubuntu, LG webOS či FireOS. Nevýhodou je však poměrně složitější nastavování (k vývoji je potřeba Node.js) a výstupem jsou pouze webové aplikace spustitelné na těchto zařízeních, které nevyužívají veškerých potenciálů svých zařízení.

Pro vývoj nativních aplikací lze využít framework Xamarin. Vývoj probíhá v buď v Xamarin Studiu nebo Visual Studiu. Aplikační logika využívá univerzálnosti jazyka C#.NET, rozdíl je tedy pouze v prezenční vrstvě. Ta využívá Xamarin.Forms, ve kterém

jsou pomocí jeho vlastní syntaxe vytvářena univerzální uživatelská rozhraní. Ta se automaticky přizpůsobují stylům a zvyklostem jednotlivých platforem. Xamarin však podporuje pouze Android, iOS a Windows Phone. Univerzální aplikace pro Windows 10, a tedy i pro Windows Mobile, nejsou zatím zcela podporované. Chybí dokončení některých UI prvků a je známo několik chyb, které způsobují pád aplikace.



Obrázek 20: Architektura aplikace při použití frameworku Xamarin.Forms
Zdroj: Převzato z [36]

Xamarin navíc nabízí velmi zajímavou službu – Xamarin Test Cloud, která umožňuje testovat aplikace na více než 2 tisících mobilních zařízeních umístěných v cloudu. Vývojáři tak získají přehled výkonosti a stabilitě aplikací v přehledných statistikách a také screenshoty aplikace ze všech těchto zařízení.

Aplikace vyvíjená v rámci diplomové práce však byla cílena pouze na zařízení s Windows 10 Mobile, protože na ostatních platformách již aplikace existuje či je právě ve vývoji. Také aplikace nebyla vyvíjena jako multiplatformní z důvodu neúplné podpory Windows 10 ze strany Xamarin a odlišné architektury aplikace v případě Apache Cordova.

4.6 EKONOMICKÉ ZHODNOCENÍ A PŘÍNOS REALIZACE

Aplikace byla vyvíjena pro univerzální Windows platformu tak, aby ji bylo možné při provedené drobných designových úprav spouštět jak na mobilních zařízeních, tak i těch desktopových.

Hodnotu pro zákazníky přináší v možnosti ovládat pomocí mobilního telefonu systém Kentico. Zákazníci tak při naléhavých případech mohou urychleně zkontrolovat stav

systému, jednotlivých stránek či objektů a některé z nich v případě potřeby upravovat, vytvářet nebo mazat.

Během vývoje však bylo zjištěno několik omezení. Jedním z nich je orientace REST pouze na stavy objektů. Pomocí tohoto rozhraní nelze vzdáleně volat procedury, proto některé důležité funkce nemohou být aplikací podporovány. Příkladem je restartování systému či vymazání serverové cache. K těmto a dalším funkcím by však bylo možné vytvořit webové API, kdy by se vyžádáním konkrétních adres serveru volaly metody pomocí Kentico API.

Přínosem pro Kentico Software je pak přidaná hodnota pro zákazníka oproti konkurenci, která může společnosti přispět ke zvýšení loajality svých zákazníků, kteří nebudou mít zájem přecházet ke konkurenčním produktům. Také to dokazuje, že firma je schopna reagovat na požadavky zákazníku a na trendy ve vývoji a své portfolio a strategické plány operativně přizpůsobovat. Tímto je společnosti nepřímo přinášen i potenciál pro zvýšení příjmů i když samotná aplikace bude pravděpodobně nabízena za symbolickou částku 1$.

Mně, začínajícímu vývojáři, tato práce přinesla mnoho nových informací a znalostí, které jsem díky benefitům společnosti Kentico mohl nabývat ve vymezené části pracovní doby z kvalitních informačních zdrojů včetně rozsáhlé firemní knihovny. Aplikaci jsem vyvíjel také v rámci pracovní doby, za její vývoj mi tedy náleží běžná hodinová mzda. I když vývoj aplikace byl návrhem společnosti a její vývoj včetně budoucí údržby bude konzultován s odpovědnými zaměstnanci společnosti, bude publikována pod mým vývojářským účtem Microsoftu. Z každého stažení aplikace tedy budu přijímat i symbolickou prodejní cenu.

ZÁVĚR

Technologické pokroky a vývoj trhu nemůže žádná společnost velkou měrou ovlivnit. Nezbývá jim proto nic jiného než sledovat aktuální trendy, trhu se přizpůsobit a pokusit se svým zákazníkům oproti konkurenci nabídnout navíc nějakou přidanou hodnotu. Společnost Kentico Software si je tohoto vědoma, a proto se v současné době snaží zaměřit i na podporu mobilních zařízení pro správu systému Kentico CMS/EMS.

Softwarové společnosti musejí taktéž sledovat aktuální trendy v softwarovém inženýrství. Standardní metodiky vývoje v současné době již nedostačují, proto je třeba hledat řešení v alternativních agilních přístupech. Důraz je aktuálně kladen také na kvalitu kódu. Nekvalitní kód může způsobovat nestabilitu aplikace či zvyšuje nákladnost případných úprav a údržby.

Standardní i agilní přístupy vývoje softwaru byly popsány v teoretické části práce včetně jejich porovnání. Podrobněji byla rozebrána metodika Test-Driven Development, jejímž použitím ve vývojovém procesu lze dosáhnout téměř úplného pokrytí kódu automatickými jednotkovými testy. Ty poskytují vývojářům rychlou zpětnou vazbu o tom, zda při úpravě nějaké části kódu nezanesli do aplikace chyby. V práci byl rovněž analyzován trh mobilních operačních systémů včetně trhu s mobilními aplikacemi a samotná společnost Kentico Software.

Hlavním cílem diplomové práce byl návrh aplikace pro správu systému Kentico CMS/EMS pro mobilní platformu Windows 10. V teoretické části proto byl tento systém představen včetně jeho rozdílů oproti předchozím verzím. Dále bylo popsáno vývojové prostředí včetně programovacích či značkovacích jazyků, které byly při vývoji použity včetně návrhového vzoru MVVM.

V praktické části byly analyzovány procesy navrhované aplikace a sestrojen vývojový diagram. Dále byl popsán návrh designu aplikace v prostředí Blend a pomocí značkovacího jazyka XAML. Zdůrazněno bylo vytváření vlastních šablon a návrh stavů stránek či ovládacích prvků.

K vývoji aplikační logiky byla použita výše zmíněná metodika Test-Driven Development, praktická část proto také obsahuje úkony nutné k vytvoření sestavy dvou

projektů, z nichž jeden představoval samotnou aplikaci a druhý automatické testy. Během návrhu řešení však bylo zjištěno několik problémů, které zavedení automatického testování doprovází. Ty lze eliminovat úpravami kódu, které byly předvedeny na konkrétních příkladech. Na závěr byly diskutovány možnosti multiplatformního vývoje aplikace.

K zavedení automatického testování bylo původně zamýšleno použití nUnit frameworku, při sestavování projektu však bylo zjištěno, že toto populární rozšíření zatím nepodporuje testování univerzálních aplikací Windows (UWP), proto musel být nahrazen MSTest frameworkem, který standardně bývá již součástí vývojového prostředí Visual Studio. Bylo však zjištěno, že oba tyto frameworky pracují téměř shodně.

Výstupem práce je fungující aplikace podporující správu několika základních objektů systému Kentico, v budoucnu by aplikace měla být rozšířena o další funkcionalitu, případně i pokročilejší funkce, které však v současné době nelze implementovat z důvodu omezení rozhraní REST, prostřednictvím kterého je systém aplikací vzdáleně ovládán. Nabízí se tedy možnost doprogramovat webové API na straně serveru, prostřednictvím kterého by bylo možné využít i rozšířenou funkcionalitu.

SEZNAM POUŽITÝCH ZDROJŮ

[1] HRMA, Jiří. Platforma Pocket PC/Windows Mobile dnes slaví 10 let od svého vzniku. *SMARTmania.cz* [online]. SMARTmania, 2010 [cit. 2017-04-10]. Dostupné z: http://smartmania.cz/clanky/platforma-pocket-pc-windows-mobile-dnes-slavi-10-let-od-sveho-vzniku-244

[2] Historie aktualizací Windows Phone 7. *Windows Phone* [online]. Microsoft, 2017 [cit. 2017-04-15]. Dostupné z: http://www.windowsphone.com/cs-cz/how-to/wp7/basics/update-history

[3] Historie aktualizací Windows Phone 8. *Windows Phone* [online]. Microsoft, 2017 [cit. 2017-04-15]. Dostupné z: http://www.windowsphone.com/cs-CZ/how-to/wp8/basics/windows-phone-8-update-history

[4] MICROSOFT. *News Center* [online]. © 2015 [cit. 2017-04-15]. Dostupné z: http://news.microsoft.com/

[5] *MSDN-the microsoft developer network* [online]. Microsoft, 2015 [cit. 2017-04-20]. Dostupné z: http://msdn.microsoft.com/

[6] *Windows Dev Center* [online]. Redmond: Microsoft, 2017 [cit. 2017-04-20]. Dostupné z: http://developer.microsoft.com/

[7] NASH, Trey. *C# 2010: rychlý průvodce novinkami a nejlepšími postupy*. 1. vydání. Brno: Computer Press, 2010. ISBN 978-802-5130-346.

[8] XML Tutorial. *W3Schools: Online Web Tutorials* [online]. ©1999-2017 [cit. 2017-04-21]. Dostupné z: http://www.w3schools.com/xml/

[9] NATHAN, Adam. *Universal Windows apps with XAML and C# unleashed*. 1st edition. Indianapolis: Pearson Education, 2015, 768 s. Unleashed. ISBN 978-0-672-33726-0.

[10] *JSON* [online]. b.r. [cit. 2017-04-21]. Dostupné z: http://www.json.org/

[11] ISO/IEC DIS 21778. *ECMA-404: The JSON Data Interchange Format*. 1st edition. Geneva: Ecma International, 2013.

[12] *Kentico 10 Documentation* [online]. Brno: Kentico Software, 2016 [cit. 2017-04-23]. Dostupné z: https://docs.kentico.com/k10

[13] FREDRICH, Todd. *RESTful Service Best Practices: Recommendations for Creating Web Services* [online]. b.r. [cit. 2017-04-23]. Dostupné z: http://www.restapitutorial.com/

[14] *Visual Studio IDE* [online]. Redmond: Microsoft, 2017 [cit. 2017-04-26]. Dostupné z: https://www.visualstudio.com/

[15] CHACON, Scott. *Pro Git*. 1. vydání. Praha: CZ.NIC, 2009. CZ.NIC. ISBN 978-80-904248-1-4. Dostupné také z: https://knihy.nic.cz/files/edice/pro_git.pdf

[16] KADLEC, Václav. *Agilní programování: metodiky efektivního vývoje softwaru*. 1. vydání. Brno: Computer Press, 2004. ISBN 978-802-5103-425.

[17] PROCHÁZKA, Jan. Agilní projekty z pohledu zákazníka. *Webová integrace* [online]. Praha: Lundegaard, 2014 [cit. 2017-04-30]. Dostupné z: http://www.web-integration.info/cs/blog/agilni-projekty-zpohledu-zakaznika/

[18] BECK, Kent. *Programování řízené testy*. 1. vydání. Praha: Grada, 2004. Moderní programování. ISBN 80-247-0901-5.

[19] WEIL, Arnaud. *Learn WPF MVVM: XAML, C# and the MVVM pattern* [online]. 1st edition. Victoria: Leanpub, 2017, 168 s. [cit. 2017-04-18]. ISBN 978-1-326-84799-9. Dostupné z: https://leanpub.com/learnwpf

[20] LÁSKA, Jan. V příštím roce bude chytrý telefon používat skoro polovina světové populace. *MobilMania.cz* [online]. Praha: CN Invest, 2016 [cit. 2017-04-30]. ISSN 1213-8991. Dostupné z: http://www.mobilmania.cz/bleskovky/v-pristim-roce-bude-chytry-telefon-pouzivat-skoro-polovina-svetove-populace/sc-4-a-1336962

[21] SKOŘEPA, Martin. Trend je jasný: stále častěji nakupujeme přes mobilní zařízení. *Mobilizujeme.cz* [online]. Praha, 2016 [cit. 2017-04-30]. Dostupné z: https://mobilizujeme.cz/clanky/trend-je-jasny-stale-casteji-nakupujeme-pres-mobilni-zarizeni

[22] Smartphone OS global market share 2009-2016: Statistic. *Statista* [online]. Hamburg: Statista, 2017 [cit. 2017-05-03]. Dostupné z: https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/

[23] *SvetAndroida.cz* [online]. Praha: Petr Mišák, 2014 [cit. 2017-05-03]. Dostupné z: https://www.svetandroida.cz

[24] VODÁK, Jakub. *Návrh a realizace mobilní aplikace pro zařízení iPhone*. Brno, 2013. Diplomová práce. Vysoké učení technické v Brně, Fakulta podnikatelská. Vedoucí práce Ing. Petr Dydowicz, Ph.D.

[25] *Upgrade na Windows 10 a aktualizace pro mobilní zařízení Windows 8.1* [online]. Redmond: Microsoft, 2017 [cit. 2017-05-08]. Dostupné z: http://www.windowsmobile.com

[26] LUKEŠ, Jindřich. Microsoft ukončil výrobu Lumie 950 (XL). Má vůbec smysl dnes kupovat zařízení s Windows 10 Mobile?: Obsáhlý pohled a shrnutí. *Windows Mobile Mania* [online]. Cheb: Jindřich Lukeš, 2016 [cit. 2017-05-12]. Dostupné z: https://wmmania.cz/clanek/byla-ukoncena-vyroba-lumie-950-a-ma-vubec-smysl-dnes-kupovat-zarizeni-s-windows-10-mobile/

[27] Jak si vybíráme chytrý telefon. *Seznam.cz: Výzkumník* [online]. Seznam.cz, 2014 [cit. 2017-05-10]. Dostupné z: https://vyzkumnik.seznam.cz/zpravodaj/jak-si-vybirame-chytry-telefon

[28] Apple App Store: number of available apps 2017: Statistic. *Statista* [online]. Hamburg: Statista, 2017 [cit. 2017-05-12]. Dostupné z:

https://www.statista.com/statistics/263795/number-of-available-apps-in-the-apple-app-store/

[29] Google Play Store: number of apps 2009-2017: Statistic. *Statista* [online]. Hamburg: Statista, 2017 [cit. 2017-05-12]. Dostupné z: https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/

[30] TUNG, Laim. 'Microsoft by the numbers' 2015: 700k Windows Store Apps, 1.2bn Office users. *ZDNet* [online]. Cameron Park: CBS Interactive, 2015 [cit. 2017-05-12]. Dostupné z: http://www.zdnet.com/article/microsoft-by-the-numbers-2015-700k-windows-store-apps-1-2bn-office-users/

[31] Most popular Apple App Store categories 2017. *Statista* [online]. Hamburg: Statista, 2017 [cit. 2017-05-12]. Dostupné z: https://www.statista.com/statistics/270291/popular-categories-in-the-app-store/

[32] ZAMORA, Bernardo. Windows Store Trends – February 2016. *Windows Blog* [online]. Redmond: Microsoft, 2016 [cit. 2017-05-12]. Dostupné z: https://blogs.windows.com/buildingapps/2016/02/04/windows-store-trends-february-2016

[33] *Kentico* [online]. Brno: Kentico Software, © 2004-2017 [cit. 2017-04-28]. Dostupné z: http://www.kentico.com

[34] Magic Quadrant for Web Content Management. *Gartner* [online]. Stamford: Gartner, 2016 [cit. 2017-04-26]. Dostupné z: http://www.gartner.com/doc/3458418

[35] *Interní dokumenty Kentico*. Brno, 2004-2017.

[36] *Mobile App Development & App Creation Software - Xamarin* [online]. San Francisco: Xamarin, 2017 [cit. 2017-05-18]. Dostupné z: https://www.xamarin.com/

SEZNAM OBRÁZKŮ

SEZNAM GRAFŮ

SEZNAM PŘÍLOH

Příloha 1: Zdrojový kód MainPage.xaml.cs

```
using KenticoAdministration.DataProviders;
using KenticoAdministration.ViewModels;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Navigation;

namespace KenticoAdministration.Views
{
    public sealed partial class MainPage : Page
    {
        public MainPage()
        {
            this.InitializeComponent();
        }

        protected override void OnNavigatedTo(NavigationEventArgs e)
        {
            base.OnNavigatedTo(e);
            var dataProvider = new SitesDataProvider();
            titleStackPanel.DataContext = new SitesViewModel(dataProvider);
        }
    }
}
```

Příloha 2: Zdrojový kód ViewModels.SitesViewModel.cs

```
using KenticoAdministration.DataProviders;
using KenticoAdministration.Models;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.ComponentModel;
using System.Linq;

namespace KenticoAdministration.ViewModels
{
    public class SitesViewModel : INotifyPropertyChanged
    {
        public ObservableCollection<Site> AllSites { get; set; }
        private Site selectedSite;
        public Site SelectedSite
        {
            get { return selectedSite; }
            set
            {
                if (selectedSite != value)
                {
                    selectedSite = value;
                    PropertyChanged?.Invoke(this, new
PropertyChangedEventArgs(nameof(SelectedSite)));
                }
            }
        }
        public event PropertyChangedEventHandler PropertyChanged;

        private ISitesDataProvider dataProvider;


        public SitesViewModel(ISitesDataProvider dataProvider)
        {
            this.dataProvider = dataProvider;
            AllSites = new ObservableCollection<Site>();
            LoadSites();
        }


        public async void LoadSites()
        {
            List<Site> allSites = await dataProvider.GetAllSitesAsync();

            foreach (Site site in allSites)
            {
                AllSites.Add(site);
            }
            SelectedSite = AllSites.First();
        }
    }
}
```

Příloha 3: Zdrojový kód Helpers.Repository.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using KenticoAdministration.Models;
using System.IO;
using System.Runtime.Serialization.Json;
using System.Net.Http;
using System.Net.Http.Headers;
using System.Net;

namespace KenticoAdministration.Helpers
{
    public class Repository
    {
        private static Uri baseAdress = new Uri("http://www.example.com");
        private static AuthenticationHeaderValue authorization = new
AuthenticationHeaderValue(AuthenticationSchemes.Basic.ToString(), "CODEDSTRING");
        private static List<Site> allSites;
        private static HttpClient client;

        public static async Task<List<Site>> GetAllSitesAsync()
        {
            if (allSites == null)
            {
                Sites sites = await GetRootObject<Sites>("cms.site") as Sites;
                allSites = sites.Cms_Objects.First().Cms_Object;
            }
            return allSites;
        }

        private static async Task<object> GetRootObject<T>(string requestPath)
        {
            if (client == null)
            {
                client = new HttpClient() { BaseAddress = baseAdress };
                client.DefaultRequestHeaders.Authorization = authorization;
            }

            Stream stream = await
client.GetStreamAsync($"rest/{requestPath}?format=json");
            return SerializeObject<T>(stream);
        }

        public static object SerializeObject<T>(Stream stream)
        {
            var serializer = new DataContractJsonSerializer(typeof(T));
            return (T)serializer.ReadObject(stream);
        }

        public static List<Site> SerializeData(Stream stream)
        {
            var serializer = new DataContractJsonSerializer(typeof(Sites));
            Sites rootObject = (Sites)serializer.ReadObject(stream);
            return rootObject.Cms_Objects.First().Cms_Object;
        }
    }
}
```

Příloha 4: Zdrojový kód Helpers.ObjectComparer.cs

```
using System.Reflection;

namespace KenticoAdministration.Helpers
{
    public class ObjectsComparer
    {
        public static bool AreObjectsEqual<T>(T object1, T object2)
        {
            if (object1 == null || object2 == null)
                return false;

            foreach (var property in object1.GetType().GetProperties())
            {
                var property1 = property.GetValue(object1);
                var property2 = property.GetValue(object2);

                if (property1 == null && property2 == null)
                    continue;

                if (property1.ToString() != property2.ToString())
                    return false;
            }

            return true;
        }
    }
}
```

Příloha 5: Zdrojový kód DataProviders.ISitesDataProvider

```
using KenticoAdministration.Models;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace KenticoAdministration.DataProviders
{
    public interface ISitesDataProvider
    {
        Task<List<Site>> GetAllSitesAsync();
    }
}
```

Příloha 6: Zdrojový kód DataProviders.SitesDataProvider.cs

```
using System.Collections.Generic;
using System.Threading.Tasks;
using KenticoAdministration.Models;
using KenticoAdministration.Helpers;

namespace KenticoAdministration.DataProviders
{
    public class SitesDataProvider : ISitesDataProvider
    {
        public Task<List<Site>> GetAllSitesAsync()
        {
            return Repository.GetAllSitesAsync();
        }
    }
}
```

Příloha 7: Zdrojový kód Models.Sites.cs

```
using KenticoAdministration.Helpers;
using System.Collections.Generic;
using System.Runtime.Serialization;

namespace KenticoAdministration.Models
{
    /// <summary>
    /// Obalovací třída
    /// </summary>
    [DataContract]
    public class Sites
    {
        [DataMember(Name = "cms_sites")]
        public List<ObjectList> Cms_Objects { get; set; }

        public Sites()
        {

        }
    }

    /// <summary>
    /// Třída, která obsahuje seznam objektů Site a seznam objektů TotalRecord
    /// </summary>
    [DataContract]
    public class ObjectList
    {
        [DataMember(Name = "CMS_Site")]
        public List<Site> Cms_Object { get; set; }
        [DataMember]
        public List<TotalRecord> TotalRecords { get; set; }

        public ObjectList()
        {

        }
    }

    /// <summary>
    /// Třída popisující objekt Site
    /// </summary>
    public class Site
    {
        [DataMember(EmitDefaultValue = false)]
        public int? SiteID { get; set; }

        [DataMember(EmitDefaultValue = false)]
        public string SiteName { get; set; }

        [DataMember(EmitDefaultValue = false)]
        public string SiteDisplayName { get; set; }

        [DataMember(EmitDefaultValue = false)]
        public string SiteDescription { get; set; }

        [DataMember(EmitDefaultValue = false)]
        public string SiteStatus { get; set; }
```

```
        [DataMember(EmitDefaultValue = false)]
        public string SiteDomainName { get; set; }

        [DataMember(EmitDefaultValue = false)]
        public int? SiteDefaultStylesheetID { get; set; }

        [DataMember(EmitDefaultValue = false)]
        public string SiteGUID { get; set; }

        [DataMember(EmitDefaultValue = false)]
        public string SiteLastModified { get; set; }

        [DataMember(EmitDefaultValue = false)]
        public string SiteIsOffline { get; set; }

        [DataMember(EmitDefaultValue = false)]
        public string SiteOfflineMessage { get; set; }

        [DataMember(EmitDefaultValue = false)]
        public string SitePresentationURL { get; set; }

        [DataMember(EmitDefaultValue = false)]
        public string SiteIsContentOnly { get; set; }

        [DataMember(EmitDefaultValue = false)]
        public string SiteDefaultEditorStylesheet { get; set; }

        [DataMember(EmitDefaultValue = false)]
        public string SiteDefaultVisitorCulture { get; set; }

        [DataMember(EmitDefaultValue = false)]
        public string SiteOfflineRedirectURL { get; set; }


        public Site()
        {

        }

        public Site(string siteDisplayName)
        {
            SiteDisplayName = siteDisplayName;
        }


        public override bool Equals(object obj)
        {
            return ObjectsComparer.AreObjectsEqual(this, (Site)obj);
        }

        public override int GetHashCode()
        {
            return base.GetHashCode();
        }
    }
}
```

Příloha 8: Zdrojový kód Models.TotalRecord.cs

```
namespace KenticoAdministration.Models
{
    /// <summary>
    /// Třída popisující objekt TotalRecord
    /// </summary>
    public class TotalRecord
    {
        public string TotalRecords { get; set; }

        public TotalRecord()
        {

        }
    }
}
```

Příloha 9: Zdrojový kód Test.ViewModels.SiteViewModelTests.cs

```
using System.Collections.Generic;
using System.Threading.Tasks;
using KenticoAdministration.Models;
using KenticoAdministration.Helpers;

namespace KenticoAdministration.DataProviders
{
    public class SitesDataProvider : ISitesDataProvider
    {
        public Task<List<Site>> GetAllSitesAsync()
        {
            return Repository.GetAllSitesAsync();
        }
    }
}
```

Příloha 10: Zdrojový kód Test.Helpers.RepositoryTests.cs

```
using System.Collections.Generic;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using System.IO;
using KenticoAdministration.Models;

namespace KenticoAdministration.Helpers.Test
{
    [TestClass]
    public class RepositoryTests
    {
        [TestMethod]
        public void SerializeAllSites_AllSitesAreSerialized()
        {
            Stream stream = new FileStream("DataMock/cms_sites.txt",
FileMode.Open, FileAccess.Read);
            List<Site> allSitesExpected = new List<Site>
            {
                new Site()
                {
                    SiteID = 2,
                    SiteName = "CorporateSite",
                    SiteDisplayName = "Corporate Site",
                    SiteDescription = "Sample Corporate web site",
                    SiteStatus = "STOPPED",
                    SiteDomainName = "localhost",
                    SiteDefaultStylesheetID = 16,
                    SiteGUID = "733ee268-eae4-45c0-9cbb-1d397b2532a0",
                    SiteLastModified = "2017-03-31T18:51:15.4021253+02:00",
                    SiteIsOffline = "false",
                    SiteOfflineMessage = ""
                },
                new Site()
                {
                    SiteID = 1,
                    SiteName = "DancingGoat",
                    SiteDisplayName = "Dancing Goat",
                    SiteDescription = "",
                    SiteStatus = "RUNNING",
                    SiteDomainName = "localhost",
                    SiteGUID = "1bafe5ff-317b-4d1c-9c8f-e8238ef05d6f",
                    SiteLastModified = "2017-03-31T19:13:08.6251253+02:00",
                    SiteIsOffline = "false",
                    SiteOfflineMessage = ""
                }
            };


            List<Site> allSites = Repository.SerializeData(stream);

            CollectionAssert.AreEqual(allSitesExpected, allSites);
        }
    }
}
```

Příloha 11: Zdrojový kód Test.Helpers.ObjectsComparerTests.cs

```
using KenticoAdministration.Models;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace KenticoAdministration.Helpers.Test
{
    [TestClass]
    public class ObjectsComparerTests
    {
        [TestMethod]
        public void TwoObjectsWithSameProperties_ObjectsAreEqual()
        {
            Site site1 = new Site()
            {
                SiteName = "Test",
                SiteID = 5
            };
            Site site2 = new Site()
            {
                SiteName = "Test",
                SiteID = 5
            };

            Assert.IsTrue(ObjectsComparer.AreObjectsEqual(site1, site2));
        }

        [TestMethod]
        public void TwoObjectsWithDifferentProperties_ObjectsAreNotEqual()
        {
            Site site1 = new Site()
            {
                SiteName = "Test",
                SiteID = 5
            };
            Site site2 = new Site()
            {
                SiteName = "Test",
                SiteID = 1
            };

            Assert.IsFalse(ObjectsComparer.AreObjectsEqual(site1, site2));
        }

        [TestMethod]
        public void TwoObjects_OneIsNull_ObjectsAreNotEqual()
        {
            Site site1 = new Site()
            {
                SiteName = "Test",
                SiteID = 5
            };
            Site site2 = null;

            Assert.IsFalse(ObjectsComparer.AreObjectsEqual(site1, site2));
        }
    }
}
```

Příloha 12: Zdrojový kód Test.DataProviders.SitesDataProviderMock.cs

```
using KenticoAdministration.Models;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace KenticoAdministration.DataProviders.Test
{
    public class SitesDataProviderMock : ISitesDataProvider
    {
        public Task<List<Site>> GetAllSitesAsync()
        {
            TaskCompletionSource<List<Site>> result = new
TaskCompletionSource<List<Site>>();

            var sites = new List<Site>()
            {
                new Site("testsite1", "Test site 1"),
                new Site("testsite2", "Test site 2")
            };
            result.SetResult(sites);
            return result.Task;
        }
    }
}
```