VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

SPRÁVA VERZÍ APLIKACÍ

SOFTWARE MANAGEMENT

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **JIŘÍ KYZLINK**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. FILIP ORSÁG, Ph.D.**
SUPERVISOR

BRNO 2018

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2017/2018

Zadání bakalářské práce

Řešitel: **Kyzlink Jiří**
Obor: Informační technologie
Téma: **Správa verzí aplikací**
Software Management
Kategorie: Uživatelská rozhraní

Pokyny:

1. Seznamte se s dostupnými řešeními a možnostmi řešení problematiky aktualizace aplikací a služeb a prostudujte možnosti realizace kontroly stavu (*health check*) spuštěných aplikací.
2. Vyberte vhodné řešení systému aktualizací aplikací pro firmu Y Soft s ohledem na stávající řešení ve firmě.
3. Vybrané řešení navrhněte a implementujte (vytvořte uživatelské rozhraní umožnující instalovat zvolenou verzi aplikace, automatické vytvoření nového instalátoru, uložení na centrální server a sběr a zobrazení informací o stavu spuštěných aplikací do stávajícího webového rozhraní).
4. Aplikaci prakticky otestujte v prostředí firmy Y Soft.
5. Zhodnoťte dosažené výsledky.

Literatura:

- STACKPOLE, Bill. a Patrick. HANRION. *Software deployment, updating, and patching*. New York: Auerbach Publications, 2008. ISBN 978-0849358005.

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 a 2 zadání.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Orság Filip, Ing., Ph.D.**, UITS FIT VUT
Datum zadání: 1. listopadu 2017
Datum odevzdání: 16. května 2018

L.S.

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
612 66 Brno, Božetěchova 2

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

Abstrakt

Práce se zabývá řešením problému aktualizací aplikací běžících na platformě .NET Framework a následnou kontrolou jejich stavu při běhu. Byly vytvořeny dva frameworky, první, postavený na knihovně Squirrel.Windows, umožňující aktualizovat aplikaci na uživatelem zvolenou verzi pomocí grafického nebo konzolového uživatelského rozhraní, včetně podpory kanálů pro předběžné verze. Druhý, umožnující jednoduchou registraci aplikace a jejich služeb do nástroje *Consul*, který následně kontroluje jejich stav a umožňuje získání přehledu o spuštěných aplikacích a jejich stavu. Oba frameworky jsou zpracované modulárně a díky .NET Standardu cílené na co nejširší použití. Frameworky jsou nasazené a využívané ve firmě Y Soft.

Abstract

The thesis deals with updating applications written for the .NET Framework platform and monitoring their status. Thesis solves both requirements with two frameworks. The first framework is based on the Squirrel.Windows library allowing updating the application to version selected by a user via the graphical or terminal user interface and it also supports release channels. The second framework simplifies registration of an application and its services to the consul health checking tool and also obtaining information about available application and services from the tool. Consul provides the ability to monitor availability and health of the registered applications and its services. Both frameworks are modular, targeting the broad range of applications with use of the .NET Standard. Frameworks were successfully internally deployed in Y Soft Corporation.

Klíčová slova

verzování, správa verzí, kontrola stavu, Y Soft, .NET Standard, .NET Core, Squirrel.Windows, consul, DotVVM

Keywords

Versioning, Software Management, Health Check, Y Soft, .NET Standard, .NET Core, Squirrel.Windows, consul, DotVVM

Citace

KYZLINK, Jiří. *Správa verzí aplikací*. Brno, 2018. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Filip Orság, Ph.D.

Správa verzí aplikací

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci na téma „Správa verzí aplikací“ vypracoval samostatně pod vedením pana Ing. Filipa Orsága, Ph.D.. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Jiří Kyzlink
16. května 2018

Poděkování

Tímto bych rád poděkoval vedoucímu bakalářské práce Ing. Filipu Orságovi Ph.D. za cenné rady, podněty při návrhu řešení a za čas věnovaný konzultaci a vedení této bakalářské práce.

Obsah

Seznam obrázků

Kapitola 1

Úvod

Při vytváření zpočátku jednoduchých, interně využívaných, desktopových aplikací je často přehlížen proces distribuce, instalace a případné pozdější aktualizace. Aplikace je sestavena, otestována a uložena na interní server odkud si ji může každý zkopírovat a používat, pokud si všimne, že je vydaná nová verze je na uživateli si aplikaci aktualizovat.

Cílem této práce je zajistit jak pro uživatele, tak pro vývojáře pohodlné řešení pro prvotní instalaci, distribuci a aktualizaci aplikací napsaných pro platformu .NET za využití stávající infrastruktury dostupné ve firmě Y Soft. Řešení bude podporovat práci s více vývojovými větvemi, jejich změnu bez nutnosti stahovat si nový instalátor, bude mít přehled o všech aktuálně vydaných verzích aplikace a díky tomu umožní aktualizaci na novější verzi i na verzi dříve vydanou pomocí grafického i textového uživatelského rozhraní, nabídne upomenutí při zjištění nové verze a snadnou aktualizaci jedním kliknutím.

Pro vývojáře zajistí jednoduchou integraci do aplikace a procesu sestavení, automatizaci verzování. Automatické mazání starých verzí aplikací a zaniklých kanálů z úložiště. Možnosti monitoringu aktuálně běžících aplikací, včetně jednotlivých služeb, které aplikace můžou nabízet, a integraci přehledu stavu do stávajícího webového rozhraní.

1.1 Využití ve firmě Y Soft

Společnost Y Soft Corporation, a.s. se primárně zabývá vývojem tiskového řešení *YSoft SafeQ*, umožňující uživateli odeslat tiskovou úlohu na centrální server a následně ji po autentizaci vytisknout na kterémkoliv dostupném multifunkčním zařízení (MFD – z anglického *Multifunctional Device*). Systém sestává mj. z části přijímající a dočasně ukládající tiskové úlohy a z tzv. terminal serveru. Většina MFD disponuje vestavěným webovým prohlížečem a terminal server jim poskytuje webové stránky pro zobrazení, některé MFD ale takovou funkcionalitu nenabízejí a je nutné vyvinout nativní aplikaci, která bude pro komunikaci se SafeQ resp. terminal serverem využívat jeho HTTP API.

Vzhledem ke skutečnosti, že MFD zpravidla není možné emulovat a ověřit tak funkčnost a kompatibilitu vestavěného prohlížeče MFD s webovými stránkami poskytované terminal serverem nebo ověřit správnou funkčnost nativní aplikace automaticky na virtuálním zařízení, je nutné využít pro testování fyzické MFD dostupné ve vývojovém pracovišti firmy. Ověření funkčnosti dříve probíhalo výhradně manuálně, avšak postupně se přechází na automatickou validaci a verifikaci pomocí robotického systému vyvíjeného v rámci YSoft Labs[1].

[1] Podrobnosti možno nalézt na https://www.ysoft.com/en/company/ysoft-labs

Robotický systém umožňuje validaci a verifikaci funkčnosti všech druhů zařízení, od jednoduchých a jednoúčelových (např. kávovar) až po komplexní a složitá zařízení jakým může být třeba MFD. Pro dosažení schopnosti validovat a verifikovat funkčnost zařízení systém disponuje nejrůznějšími typy senzorů a aktorů. Systém se skládá z několika různých komponent a aplikací, z nichž některé budou využívat výsledky této práce pro správu verzí a monitorování jejich stavu.

Kapitola 2

Teorie vývoje, nasazení, verzování a sestavování softwaru

V této kapitole je popsán jak obecný postup při vývoji softwaru, tak specifické požadavky firmy Y Soft, jejichž splnění je nutné pro efektivní využití výsledků této práce v produkčním prostředí. Dále se zaobírá postupy a systémy pro verzování zdrojového kódu, sestavených aplikací a to včetně tzv. předběžných vydání[1].

2.1 Životní cyklus aplikací

Vývoj aplikace prochází zpravidla alespoň 5 etapami [18]:

1. Zachycení požadavků
2. Vývoj aplikace a její testování
3. Implementace a zavedení do provozu
4. Udržování a provoz
5. Stažení z provozu, ukončení používání

Do programového vybavení se zasahuje minimálně ve 3 etapách, ostatní nejsou pro tuto práci důležité protože při jejich provádění není potřeba zásahu do zdrojového kódu a tudíž ani vydání nových verzí aplikace.

Moderní vývojové metodiky kladou důraz na vývoj po částech, průběžné a automatizované testování a konzultaci dosažených výsledků se zákazníkem. Při vývoji aplikace se pracuje na více úkolech najednou, ale vývoj každého je oddělen od ostatních což umožňuje mj. efektivnější vývoj, jednodušší testování a rychlejší izolaci případných chyb. Podobný přístup se používá při každé změně zdrojových kódů (oprava chyby, následné přidání nové funkcionality, ...), tím vznikají tzv. vývojové větve. Systémy které se při vývoji používají musí být schopné s vývojovými větvemi správně pracovat a umožňovat jejich efektivní využití.

V týmu, který používá produkt této práce probíhá vývoj dle frameworku *scrum*[2] což znamená, že tým vyvíjí v časových intervalech tzv. sprintech (též označované jako iterace).

[1] Vydání, která nejsou určena pro produkční použití.

[2] Framework popisující role, procesy a události během vývoje softwarového produktu [20]

Na začátku každého sprintu je naplánováno jaké úkoly se budou zpracovávat, během sprintu se každý úkol provádí ve vlastní vývojové větvi a po dokončení se změny přenesou do větve hlavní. Během vývoje probíhá testovaní jak na jednotlivých vývojových větvích, tak na větvi hlavní, která by měla být za každých okolností stabilní a plně otestovaná. Na konci sprintu může být vydána nová, stabilní, verze produktu z hlavní vývojové větve obsahující změny z posledního sprintu.

2.2 Verzovací systém Git

Při vývoji softwaru se zpravidla používá systém, který zajišťuje verzování všech změn ve zdrojových souborech tzv. verzovací systém[3]. Mezi celosvětově nejrozšířenější verzovací systémy patří *Git* s přibližným tržním podílem 50 % a *Subversion*[4] s 42 %, další, méně rozšířené systémy jsou: *Mercurial Bazaar* a *CVS* které mají dohromady zbývajících 8 % trhu [1].

Distribuovaný verzovací systém Git [24] jehož původním autorem je Linus Torvalds[5] je aktuálně nejpoužívanější verzovací systém na světě a je jediným verzovacím systémem, který se ve firmě Y Soft aktivně používá. Za svoji velkou popularitu vděčí především rychlosti, distribuovanému návrhu a v neposlední řadě také popularitě služby *GitHub*[6].

Git je obsahově adresovatelný systém souborů [4] což znamená, že v jádru celého systému je jednoduché úložiště typu klíč-hodnota. Pro jakákoliv data, která se do repozitáře přidají vrátí unikátní klíč[7] pomocí kterého je možné data později získat. Každá revize je identifikovaná pomocí unikátního klíče (opět ve tvaru SHA-1 hash), který vznikne v okamžiku vytvoření nové revize na základě zprávy, data, jména a e-mailu autora, aktuálního stavu repozitáře, data zápisu a dalších známých dat (detailně popsáno v [4] v sekci *Commit Objects*).

Jak již bylo zmíněno, Git je distribuovaný systém a díky tomu může mít každý uživatel na svém disku celou kopii repozitáře a nepotřebuje centrální bod, ze kterého všichni uživatelé získávají změny. Ačkoliv Git sám o sobě centrální bod nepotřebuje často se využívá jeden repozitář, na který vývojáři své změny publikují a z kterého si stahují změny od ostatních. U projektů s otevřeným zdrojovým kódem je repozitář často hostovaný u služby *GitHub*, *GitLab* nebo *Atlassian Bitbucket* nabízející ve svém webové rozhraní mj. následující možnosti:

- procházení kódu bez nutnosti stažení celého repozitáře
- správu návrhů pull requestů[8] (zkráceně PR)
- komentování zdrojových souborů pro účely posouzení kódu
- wiki

Firmy a projekty s uzavřeným zdrojovým kódem hostují své repozitáře zpravidla lokálně, dostupné pouze z intranetové sítě. V případě Y Softu jsou repozitáře hostované pomocí řešení *Bitbucket Server*, které běží na interním cloudu firmy Y Soft.

[3] Někdy označován zkratkou *VCS* pocházející z anglického *Version Control Systems*
[4] Často zkráceně označován jako *SVN*
[5] Známý autorstvím kernelu *Linux*, pro který Git původně vznikal
[6] Webová služba umožnující hostování Git repozitáře, systému pro sledování problémů, Wiki, ...
[7] SHA-1 hash přidaných dat se specifickou hlavičkou přidanou Gitem
[8] Česky lze přeložit jako žádost o zanesení změn provedených v jedné větvi do jiné

Git nabízí možnosti, jak od sebe oddělit různé vývojové větve, následně v oddělené větvi zanést do zdrojových souborů požadované změny, otestovat je a sloučit zpět do kterékoliv jiné větve, díky tomuto mechanizmu může v jednom repozitáři pohodlně pracovat více vývojářů najednou a navzájem se neovlivňují. Standardem softwarového vývoje je vytvoření nové větve pro každý ucelený zásah do zdrojového kódu, nezávisle na tom, zda se jedná o opravu dříve hlášené chyby, přidání nové funkcionality či vyřešení kritického problému zaneseného předchozími změnami. Správné využívání větví umožní přehledné sledování vývoje změn a možnost vyvinout změnu odděleně a až po úplném odladění ji přenést do integrační a následně i do hlavní větve pomocí pull requestu.

Pro vytváření PR se ve Y Softu využívá dříve zmíněný *Bitbucket Server* v jehož webovém rozhraní se zvolí zdrojová, cílová větev, případný název PR, krátký popis a seznam vývojářů, kteří mají změny zkontrolovat. Po vytvoření PR je možné zhlédnout rozdíl mezi zdrojovou a cílovou větví a změny komentovat. V případě že jsou změny konfliktní tzn. že došlo ke změně stejné části kódu ve zdrojové i cílové větvi je nutné konflikty ručně opravit (označit, jak má zdrojový kód ve finále vypadat), v opačném případě není nutný další zásah vývojáře. Po schválení PR jsou změny aplikované na cílovou větev a PR je archivován pro případnou pozdější analýzu.

2.2.1 GitFlow

Při práci více vývojářů či týmů v jednom repozitáři je důležité zavést určitá pravidla pro práci s repozitářem, způsob pojmenování větví je oblast, kde není složité, ani časově náročné pravidla dodržovat, případně jejich dodržování kontrolovat a vyžadovat. Projekt, který výsledek této práce používá má v repozitáři větve pojmenované podle rozšířeného schématu nazvaného *GitFlow* (poprvé popsáno v [6]), které zavádí 2 typy větví, kde každý typ má více druhů, blíže popsáno níže:

1. Hlavní větve – existují po celou dobu života repozitáře
    - *master* – obsahuje nejnovější plně otestovaný produkční kód, který je možné kdykoliv vydat/nasadit
    - *develop* – (někdy nazývaný integrační větev) reflektuje poslední vývojové změny pro následující verzi produktu, na základě této větve vznikají pravidelná noční sestavení, měla by obsahovat otestovaný kód, avšak funkčnost není zaručena v každém okamžiku
2. Podpůrné – dočasné větve, ve kterých probíhá specifický typ vývoje
    - *feature* – vývoj nových vlastností
    - *release* – příprava pro vydání nové verze produktu, stabilizace
    - *hotfix* – kritické opravy již vydaných verzí, vychází z větve *master* a po opravě se změny reflektují zpět do větve *master* a *develop*

Pro zvýšení přehlednosti je v názvech podpůrných větví vhodné používat „`/`“, které je v grafických nadstavbách Gitu zpravidla interpretováno jako složka. Pro pojmenování konkrétních podpůrných větví je využíváno následující schéma: `{IssueId-ShortName}`, kde `IssueId` je jedinečný identifikátor požadavku v systému pro sledování požadavků a `ShortName` je krátký, pomlčkou oddělený název požadavku. Pro správnou funkci není nutné do názvu větve `ShortName` přidávat, ale takové chování vede ke snížení přehlednosti

repozitáře. V praxi je možné se setkat s větvemi pojmenovanými např. `master` pro hlavní větev, `feature/FF-1337-Add-something` pro podpůrnou větev implementující požadavek `FF-1337` s krátkým názvem `Add something` nebo `release/v4.2.0` pro stabilizační větev na jejímž základě bude vydána verze produktu `4.2.0`. Žádné jiné názvy větví nejsou povolené a vedou pouze ke zmatku v repozitáři.

V systému pro správu verzí se využívají Git větve pro automatické vytváření a pojmenování tzv. kanálů. Kanály jsou mezi sebou nezávislé stejně jako vývojové větve v Gitu a lze mezi nimi přecházet stejně jednoduše jako mezi Git větvemi. Název kanálů je odvozen z názvu větve, pokud je větev 1. typu (`master` nebo `develop`) použije se celý název větve, v případě, že je větev 2. typu použije se jako název pouze identifikátor požadavku z nástroje pro sledování požadavků např. `FF-1337`.

2.3 Verzování softwaru

Při vývoji softwaru je nutné mít informace o tom, kdy byla přidána např. specifická funkcionalita, narušena zpětná kompatibilita nebo opravena chyba. Pro snadnější orientaci a dokumentaci změn se využívají verze, které můžou mít různé podoby.

- Řetězcovou – př.: *Windows Vista*, využíváno hlavně pro marketingové účely, jednoduché oddělení majoritních verzí
- Číselnou – využíváno při kontinuálním vydávání, zpravidla se pouze inkrementuje jedno číslo
- Hybridní – př.: *v1.2.3+beta.5*, nejčastěji používaná, umožňuje největší variabilitu při odvozování nové verze

V případě řetězcové podoby je verze často vymyšlena a navržena tak, aby dobře zněla, případně na něco odkazovala, v ostatních případech se verze zpravidla odvozuje dle předem daného schématu a je známá ještě před vydáním. Schémat, na základě kterých se verze odvozuje bylo v průběhu vývoje vymyšleno mnoho a v následujícím výpisu je seznam několika běžně používaných:

- Časově závislé – př.: *Ubuntu 18.04* první číslo značí rok, druhé měsíc, kdy byla verze vydána
- Sekvenční – př.: *114.65.77.68*
  - Dle množství změn – k inkrementaci prvního číslo dochází při velkém množství změn v kódu, k inkrementaci druhého při menším počtu změn, atd.
  - Dle kompatibility – k inkrementaci prvního číslo dochází při narušení zpětné kompatibility, k inkrementaci druhého při přidání funkcionality a k inkrementaci třetího při opravě již existující funkcionality, blíže v kapitole 2.3.1.

Zvláštní, běžným postupům vymykající se, je verzovací schéma sázecího programu TEX, jehož třetí majoritní verze je dle přání autora poslední [12] a další (majoritní verze) vydána nebude. Při nutnosti vydání nového sestavení bude pouze přidána další číslice a to takovým způsobem, aby verze konvergovala k π, dalším přáním autora je, aby po jeho smrti byl program TEX vydán ve verzi π. Aktuální verze (k datu 8. Dubna 2018) programu TEX je 3.14159265. Podobné schéma verzování má i jazyk METAFONT, jen s tím rozdílem, že konverguje ke konstantě e.

2.3.1 Sémantické verzování

Zkráceně *SemVer* je specifikace popisující postupy při inkrementaci verzí vydání softwarového produktu (knihovny/aplikace) [17]. Zakládá na skutečnosti, že verze je definována řetězcem ve formátu `Major.Minor.Patch` za kterým můžou být upřesňující, pomlčkou oddělené, identifikátory předběžných verzí. V případě využití sémantického verzování ve verzi 2.0.0 je možné verzovací řetězec doplnit plusem oddělenými metadaty. Specifikace popisuje postup pro zvyšování verzí následujícím způsobem:

- V případě, že je major část verze nulová (`0.x.y`) softwarový produkt není vydaný a je možné modifikovat API bez zvyšování major verze, což je výhodné během prvotního vývoje, kdy se API často mění.
- V případě, že je major část verze nenulová mění se verze dle následujících pravidel
  - inkrementace patch – vydání oprav pro již existující API
  - inkrementace minor a nulování patch – přidání funkcionality bez narušení zpětné kompatibility API, označení části API jako zastaralé
  - inkrementace major a nulování minor a patch – při narušení zpětné kompatibility API

Identifikátor předběžného sestavení se používá pro oddělení již stabilních, produkčních sestavení od předběžných, testovacích vydání, často obsahuje *alpha*/*beta*/*rc* následované tečkou a číslem předběžného sestavení např. `alpha.42`. V práci bude identifikátor předběžných verzí generovaný na základě aktuálně sestavované větve, s pravidly uvedené v tabulce 2.1 následovaný číslem předběžného sestavení.

větev	**identifikátor předběžného sestavení**
master	*není předběžné sestavení*
develop	`alpha`
ostatní	{`zkrácený název větve`}

Tabulka 2.1: Vztah názvu větve a identifikátoru předběžného sestavení.

Při řazení se u textové části postupuje lexikálně, u číselné části číselně. Metadata mohou obsahovat libovolné alfanumerické znaky a pomlčku, často se využívají pro bližší určení, a proto obsahují např. identifikátor revize zdrojových souborů použitých k sestavení, datum sestavení nebo název počítače, na kterém sestavení proběhlo. V práci používaný tvar metadat bude: `+Built.180403.Branch.develop.Sha.badcafe`, díky tomu bude zpětně dohledatelné kdy, z jaké větve a z jaké revize sestavení vzniklo. Metadata neovlivňují řazení sémantických verzí.

Kapitola 3

Popis použitých frameworků, nástrojů a knihoven

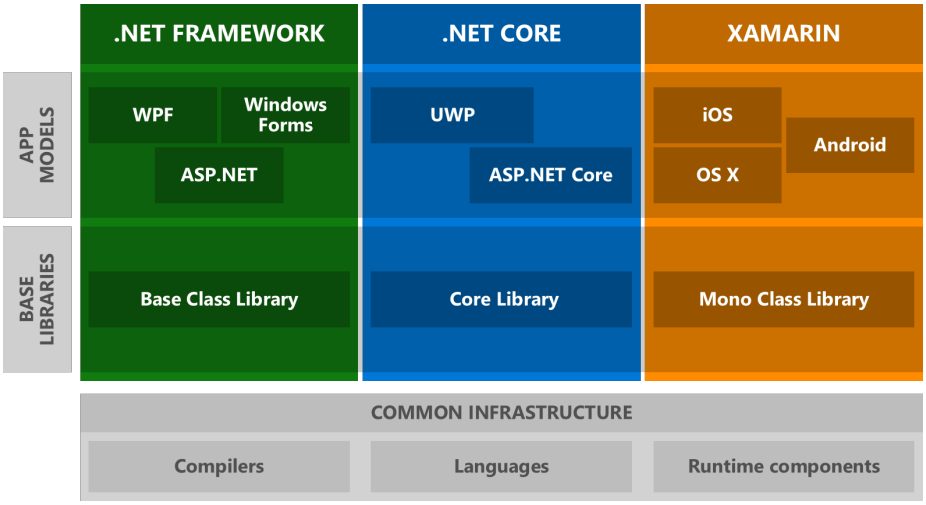
V této kapitole je popsána platforma .NET obecně, význam .NET Standardu pro multiplatformní aplikace, jednotlivé části ze kterých se skládá .NET Core a které jsou pro tuto práci významné. Dále zde jsou popsány nástroje používané v rámci frameworku pro aktualizaci a kontrolu stavu aplikací a knihovny umožnující realizaci funkcionality standardními cestami.

3.1 Platforma .NET

Počátky platformy .NET jsou spjaty s firmou *Microsoft*, která jej vyvíjela a s operačním systémem *Microsoft Windows*, který jako jediný podporoval běh aplikací vytvořených pro platformu .NET. Platforma nepředepisuje použití specifického programovacího jazyka, z důvodu, že všechny podporované přeloží do *Common Intermediate Language* (CIL, někdy též zkráceně IL) a v této formě jsou distribuovány. Při spuštění aplikace je IL znovu, již za běhu, přeložen pomocí *Common Language Runtime* (zkráceně CLR) do strojového kódu, kompatibilního s architekturou na které CLR běží, a následně vykonán.

Pro usnadnění a urychlení vývoje aplikací byl pro operační systém *Microsoft Windows* vytvořen .NET Framework nabízející pomocné funkce a metody, včetně pokrytí *Windows API*. Postupem času se stala platforma .NET i nejpoužívanější jazyk této platformy C# velmi populární a v roce 2013 *Miguel de Icaza* vydal první verzi projektu *Mono*. Jehož cílem bylo vytvoření multiplatformního ekosystému s otevřeným zdrojovým kódem, kompatibilního s .NET Frameworkem. Zpočátku nebyly cíle projektu Mono, z důvodu porušování softwarových patentů, firmou Microsoft podporovány, ale se změnou ve vedení se přístup změnil. Nyní je Mono vyvíjeno firmou Xamarin a podporováno prostřednictvím .NET Foundation i Microsoftem. Umožňuje běh aplikací napsaných pro .NET Framework na desktopových i mobilních operačních systémech postavených na jádře Linux.

S tím, jak se platformy (Mono, .NET Framework) více rozvíjely a používaly na různých zařízeních vznikala platformní omezení, která se obcházela pomocí preprocesoru a podmíněné kompilace určitých částí zdrojového kódu pro běh na určité platformě. Platformní omezení vznikala z důvodu reimplementace základní funkcionality ve frameworku, jak je vidět na obrázku 3.1 .NET Framework má *Base Class Library*, Mono má podobnou funkcionalitu implementovanou v *Mono Class Library*, která neumí emulovat celé *Windows API*, a právě tyto rozdíly činily multiplatformní vývoj složitý a obtížně testovatelný [23].



Obrázek 3.1: Nekompatibilita .NET platforem před zavedením .NET Standardu, [14].

Vznikl nápad, sjednotit základní funkcionalitu dostupných platforem jednotnou specifikací API, která dostala název .NET Standard. Specifikace je vydána v několika verzích, od 1.0 až po 2.0, avšak verzování udává množství dostupných funkcí a metod a čím vyšší verze, tím větší množství funkcionality je od platformy vyžadováno. Díky tomu je možné cílit projekty na jednotlivé verze .NET Standardu podle toho, kolik funkcionality od platformy vyžadují a poběží na všech platformách splňující danou nebo vyšší úroveň standardu, není nutné je sestavovat pro každou jednu platformu. .NET Standard umožní pohodlný vývoj multiplatformních aplikací za cenu limitovaného API.

Aby byl .NET Standard úspěšný a v budoucnu široce používaný, bylo nutné jeho podporu rozšířit mezi co největší množství běhových prostředí. Proto již .NET Framework 4.5 podporuje .NET Standard 1.0 i 1.1 a .NET Framework 4.6.1 podporuje nejvyšší verzi – .NET Standard 2.0. Ostatní běhové prostředí v čele s .NET Core velice rychle adoptovali nejvyšší možnou verzi .NET Standardu [25]. A z dnešního pohledu již je preferované cílení pouze na .NET Standard 2.0 před specifickou verzí .NET Frameworku/Coru. Vlastnost, která nezanedbatelně pomáhá rozšíření .NET Standardu je i skutečnost, že knihovny napsané pro .NET Framework můžou využívat jiné napsané pro .NET Standard dle specifikací uvedených v [25]. Toto neplatí naopak tzn. jakmile se ve stromu závislostí objeví knihovna pro .NET Framework navazující knihovna je již také pouze pro .NET Framework.

Další důležitou podmínkou pro úspěšnost .NET Standardu byl „přepis“[1] knihoven, kompilace pro ideálně co nejnižší verzi .NET Standardu a jejich publikování ve veřejné NuGet galerii. Díky tomu mohly na jejich základě vzniknout další a komplexnější knihovny opět cílící na .NET Standard a čím více knihoven bylo kompatibilních, tím více koncových produktů se začalo vytvářet pro .NET Standard/Core a kolo se roztáčelo.

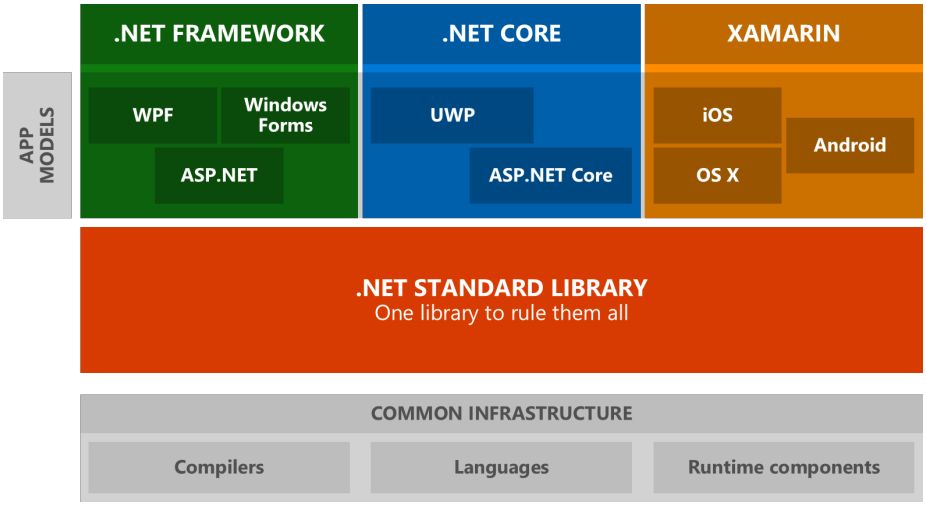
Jedna z aktuálních nevýhod .NET Standardu je, že neumožňuje vytvářet multiplatformní grafická uživatelská rozhraní (na obrázku 3.2 WPF *zatím*[2, 3] pouze pro .NET Fra-

[1] nejednalo se o přepis v pravém slova smyslu, avšak často se společně s .NET Standard kompatibilitou upravily i některé dříve špatně navržené části knihoven

[2] Jsou plány na implementaci XAML do .NET Standardu, ale to je jedna malá část mj. používaná i ve frameworku WPF.

[3] Na konferenci *Build 2018* byl oznámen .NET Standard 3.0 zahrnující GUI (grafické uživatelské rozhraní) avšak pouze pro Microsoft Windows.

mework). Existují cesty, jak takového řešení dosáhnout (Xamarin.Forms), ale zpravidla mají velice limitující omezení a hodí se jen pro jednoduché aplikace bez složitější grafiky, nikoliv pro větší komerční systémy.



Obrázek 3.2: Jednotná specifikace funkcionality .NET platforem po zavedením .NET Standardu, převzato z [14].

Microsoft také prostřednictvím .NET Foundation inicioval vývoj nového, multiplatformního frameworku nazvaného *.NET Core* s otevřeným zdrojovým kódem, dostupném na serveru GitHub[4]. .NET Core se skládá ze dvou základních částí – CoreCLR a CoreFX. CoreCLR je zkratka pro .NET Core runtime a jak již název napovídá jedná se o běhové prostředí zajišťující JIT kompilaci, základní sadu knihoven implementující mj. GC, typový systém a fundamentální typy, bez kterých by se běh neobešel (`Object`, `String`, `Thread`, ...). Nad CoreCLR je komponenta CoreFX, která implementuje zbytek API potřebného pro pokrytí .NET Standardu 2.0.

Díky vytvoření .NET Standardu, který specifikuje potřebné API, jeho referenční implementaci v rámci .NET Core je dnes možné provozovat platformu .NET na majoritní části dostupných operačních systémů bez nutnosti specifických úprav pro jednotlivé platformy. Ale v případě potřeby je možné knihovnu/aplikaci tzv. multitargetovat což znamená, že se sestaví z jednoho projektu vícekrát, pokaždé pro jiný framework, standard nebo RID.

3.2 Knihovna GitVersion

Zajištění správného sémantického verzování se z počátku může zdát jako nezanedbatelná práce navíc, se kterou by vývojáři nemuseli souhlasit avšak řešení této situace je prosté, stačí využít Git repozitář a sémantickou verzi dle jeho aktuálního stavu odvodit. Pro realizaci této funkcionality byla vybrána knihovna *GitVersion* nabízející tuto funkcionalitu, otevřený zdrojový kód se svobodnou licencí *MIT*[5] a aktivní komunitu [7].

Knihovna umožňuje generování sémantických verzí na základě konfiguračního souboru a stavu repozitáře. Při generování nové sémantické verze se postupně prochází repozitář od

[4]Zdrojové soubory jsou dostupné na adrese: https://github.com/dotnet/coreclr

[5]Licence *MIT* je svobodná, opravňující k používání, šíření i modifikaci zdrojových kódů za cenu přiložení kopie licence a jména autora.

nejnovější revize po nejstarší dokud se nenarazí na značku ve tvaru *v{Major.Minor.Patch}*, následně se prochází repozitář od značky zpět k nejnovější revizi a v komentářích revizí se hledají výskyty řetězce začínající `+semver:␣`, po kterém následuje `major`, `minor` nebo `patch`, a odpovídající část verze je inkrementována. GitVersion nabízí možnost vygenerování celé sémantické verze, avšak její schéma není pro použití v této práci vhodné, protože metadata obsahují pouze *sha* identifikaci revize. Z toho důvodu jsou metadata a identifikátor předběžné verze generovaná pomocí proměnných, které GitVersion navrací při spuštění viz výpis 3.1), během sestavování aplikace.

```
{
  "Major":13,
  "Minor":3,
  "Patch":7,
  "PreReleaseTag":"beta.1",
  "PreReleaseLabel":"beta",
  "PreReleaseNumber":1,
  "BuildMetaData":1,
  "MajorMinorPatch":"13.3.7",
  "SemVer":"13.3.7-beta.1",
  "AssemblySemVer":"13.3.0.0",
  "AssemblySemFileVer":"13.3.7.0",
  "FullSemVer":"13.3.7-beta.1+1",
  "BranchName":"release/13.3.7",
  "Sha":"28c853159a46b5a87e6cc9c4f6e940c59d6bc68a",
  "CommitsSinceVersionSource":1,
  "CommitsSinceVersionSourcePadded":"0001",
  "CommitDate":"2018-03-04"
}
```

Výpis 3.1: Výběr z proměnných získaných knihovnou GitVersion.

3.3 Systém NuGet

Důležitým nástrojem každé moderní vývojářské platformy je mechanizmus, pomocí kterého můžou vývojáři vytvářet, sdílet a využívat užitečné části kódu [2], tzv. balíčkovací systém. Pro platformu *.NET* je doporučené používat *NuGet*, což je jediný oficiálně podporovaný balíčkovací systém pro *.NET*. Ačkoliv je NuGet podporován firmou Microsoft a na vývoji se majoritní částí podílí její zaměstnanci, projekt jako takový je spravován komunitou a spadá pod nadaci *.NET Foundation*[6], podstatná část tohoto systému má svobodnou licenci a otevřený zdrojový kód dostupný na adrese https://github.com/NuGet.

Systém NuGet obsahuje mj. následující prvky nutné pro jednoduché a efektivní využití balíčků:

- Specifikaci balíčků, referenční manuály pro vytváření balíčků
- Nástroje pro vytváření, modifikaci, publikaci a stažení/instalaci balíčků

[6] Nezávislá organizace, založená Microsoftem v roce 2014, pro podporu vývoje a spolupráce na softwaru s otevřeným zdrojovým kódem v rámci ekosystému .NET [15]

  - *NuGet.Cli* – rozhraní příkazové řádky
  - *Nuget Package Explorer* – grafické rozhraní

- Systémy zajišťující distribuci a správu balíčků, tzv. repozitáře
  - `nuget.org` – nejpoužívanější, volně přístupný repozitář
  - Privátní nabízí mnoho systémů pro správu artefaktů (*JFrog Artifactory*, *Sonatype Nexus*, ...)

- Doplňky do integrovaných vývojových prostředí určené pro stažení, aktualizaci, správu závislostí, sjednocení verzí, ...
  - *NuGet Package Manager Console* – rozhraní příkazového řádku, vestavěné do prostředí Visual Studia
  - *Nuget Package Manager UI* – grafické rozhraní použitelné z prostředí Visual Studia

NuGet balíček musí mít koncovku `.nupkg` avšak soubor jako takový je běžný *ZIP* archiv s přesně specifikovanou strukturou: v kořenovém adresáři archivu se musí nacházet XML soubor pojmenovaný `{PackageId}.nuspec` obsahující základní informace o daném balíčku, mezi které patří:

- Identifikace balíčku tj. dvojice Id a verze - musí být jedinečné v celém repozitáři
- Autor
- Vlastník
- Odkaz na licenci a informaci, zda balíček vyžaduje explicitní souhlas s jejím zněním
- Popis balíčku – zobrazuje se v *Package Manager UI*, zřídkakdy obsahuje i změny provedené od posledního vydání
- Závislosti na ostatních balíčcích

Dále se v kořenovém adresáři archívu nachází zpravidla několik složek jejichž název určuje co obsahují, jak a kdy se mají použít. Balíček zpravidla obsahuje složku *lib*, která může obsahovat různé verze (zpravidla bez rozdílu funkce) sestavených knihoven pro použití s různými verzemi běhových prostředí a tudíž .NET Frameworků. Může obsahovat složku *runtimes*, ve které se můžou nacházet sestavené nativní knihovny pro různé procesorové architektury (x64, x86, ARM) a různé operační systémy (obecný Linux, Alpine Linux, RHEL, Debian, Windows, ...) identifikované pomocí tzv. RID[7], díky tomu nabízí NuGet balíčky možnost vytvářet knihovny se specifiky pro různé operační systémy a procesorové architektury. Kořenový adresář může obsahovat i jakékoliv jiné složky a soubory, které nástroje určené pro instalaci balíčků ignorují.

U balíčků, které zajišťují nějakou míru interakce s uživatelem je velice důležitou funkcionalitou systému NuGet možnost vytvářet balíčky, které můžou mít různé lokalizace. Toho bylo velice intenzivně využito například u knihovny *Humanizer*[8] zajišťující tzv. humanizaci („polidštění“) výstupu aplikací a má 40 různých lokalizací.

[7] Runtime ID - řetězec identifikující běhové prostředí. Více info viz https://github.com/dotnet/corefx/tree/master/pkg/Microsoft.NETCore.Platforms

[8] knihovna dostupná na adrese https://www.nuget.org/packages/Humanizer/

Vyvíjený systém pro správu verzí využívá balíčků NuGet pro distribuci sestavených objektů a to jak při plné tak i při rozdílové aktualizaci. Díky tomu je možné využít celou, již existující, infrastrukturu pro vytváření a distribuci balíčků, a to pouze za cenu vytvoření kompatibilního balíčku, což jak bylo zmíněno dříve, není nic náročného.

3.4 Framework Squirrel.Windows

Vytvoření systému, který by zajišťoval všechny úkony potřebné k balení, instalaci, aktualizaci aplikace je nad rámec této práce a proto byl zvolen framework, na jehož základu je výsledek práce postavený.

Byl vybrán framework *Squirrel.Windows*, který zajišťuje všechny úkony nutné k instalaci a pozdější aktualizaci aplikace. Celý framework má otevřený zdrojový kód pod svobodnou licencí *MIT*, autorem je *Paul Betts* a aktuálně (k 14. 4. 2018) 85 dalších přispěvatelů [21]. Framework aktivně používá např. desktopová aplikace populárního komunikátoru *Slack* nebo grafická nadstavba nad verzovacím nástrojem Git *Atlassian Sourcetree*. Lze jej tedy považovat za dostatečně stabilní a prověřený.

Framework původně vznikl jako náhrada s otevřeným zdrojovým kódem za *ClickOnce*, což je technologii vyvinutá firmou *Microsoft* umožňující instalaci aplikace kliknutím na odkaz ve webovém prohlížeči (primárně funguje v prohlížečích *Internet Explorer* a *Edge*). Následně umožňuje jednoduchou aktualizaci aplikace z vzdáleného serveru. Výhodou *ClickOnce* je jednoduchost instalace, izolace aplikací tzn. instalace nemůže způsobit problém s jinou aplikací, další, již méně podstatnou výhodou je jednoduchá integrace přímo z prostředí Visual Studia. Oproti tomu má technologie *ClickOnce* také několik nevýhod:

- uzavřený zdrojový kód – není možné změnit UI/UX
- nemožnost modifikace procesu instalace a aktualizace
- nemožnost spuštění instalace s vyššími právy – instalace pouze pro jednoho uživatele
- obskurní umístění nainstalované aplikace
- pevně daná struktura souborů na vzdáleném serveru
- není možné omezit přístup k souborům na vzdáleném serveru

Některé nevýhody framework řeší přímo např. uzavřený zdrojový kód a modifikace procesu instalace. Většinu ostatních (struktura a přístup k aktualizačním souborům na vzdáleném serveru) je možné jednoduše vyřešit vlastní implementací konkrétních rozhraní. Některé nevýhody přetrvávají (obskurní umístění), což je zpravidla daň za snadnou instalaci dostupnou i pro uživatele bez administrátorských oprávnění. Při vzniku frameworku *Squirrel.Windows* byly stanoveny základní cíle, které jsou nyní až na výjimky naplněny [22]:

- Jednoduchá integrace do stávající aplikace, jasné a srozumitelné API.
- Jednoduché vytvoření instalátoru a vydávání nových verzí, podpora rozdílových aktualizací.
- Snadná distribuce pomocí standardně používaných cest.
- Instalace bez průvodce (tzv. Wizard-Free^{TM}), nevyžadující administrátorská oprávnění, restarty operačního systému.

- Aktualizace aplikovatelné za běhu aplikace, nová, aktualizovaná, verze se spustí až při dalším spuštění aplikace, opět snaha vše provést bez restartu operačního systému.

Mezi důležité věci, které framework neřeší patří explicitní podpora tzv. kanálů, které jsou svoji funkcionalitou podobné větvím ve verzovacím nástroji Git. Framework se ve výchozím stavu chová tak, že jakmile je vydána nová verze aplikace, je aktualizace nabídnuta všem, kteří mají aplikaci nainstalovanou. Jediná možnost, jak omezit nabízení aktualizací je pomocí funkce cílení, kdy se v souboru `RELEASES` specifikuje procento uživatelů, kteří mají mít aktualizaci dostupnou a při prvotní instalaci se vygeneruje náhodné číslo, podle kterého se v těchto situacích rozhoduje. V práci byla podpora kanálů implementována do celého procesu aktualizace, od balení až po výběr dostupných kanálů. V práci jsou kanály použité právě pro vydávání specifických předběžných verzí aplikace, typicky sestavení pocházející z vývojových větví. Další podstatnou věcí, pro kterou nemá framework podporu a bude potřeba ji implementovat je sémantické verzování 2.0.0.

3.4.1 Architektura a princip funkce frameworku Squirrel.Windows

Pro dosažení stanovených cílů byly vytvořeny dvě nativní aplikace napsané v C++, používané pro prvotní instalaci a využívající *Windows API*. Dále vznikla aplikace `Update.exe` zajišťující zabalení, spouštění, aktualizaci a odinstalaci již nainstalované aplikace a knihovna `Squirrel.dll` v jazyku C# pro integraci do obsluhované aplikace.

Aplikace `Update.exe` je, jak již bylo zmíněno dříve, napsaná v jazyku C# a ještě před distribucí instalačních souborů má několik úkolů:

- Přebalení NuGet balíčku obsahujícího aplikaci do formátu kompatibilního s frameworkem Squirrel
- Vložení přebaleného balíčku do *vestavěných zdrojů* aplikace `Setup.exe`
- Vytvoření souboru *MSI* pro instalaci aplikace pomocí zásad skupiny v prostředí domény *Active Directory*

NuGet balíček vytvořený pomocí běžných nástrojů nemusí být kompatibilní s frameworkem *Squirrel.Windows* a proto je třeba tzv. *releasify* proces. Během tohoto procesu je balíček rozbalen do dočasné složky, zkontrolován, zda má správnou strukturu (obsahuje složku `lib`) a obsahuje potřebné soubory (`{packageId}.nuspec`), následně jsou přidány případné závislosti, protože framework *Squirrel.Windows* jako takový neřeší závislosti za běhu, je nutné distribuovat kompletní aplikaci, nakonec je dočasná složka zabalena do podoby NuGet balíčku, tentokrát s vyšší mírou komprese pro ušetření místa na vzdáleném serveru a urychlení stažení. *Releasify* proces dále zahrnuje vložení přebaleného balíčku a sebe sama (`Update.exe`) do vestavěných zdrojů (angl. embedded resources) aplikace *Setup.exe*, pomocí nativní aplikace `WriteZipToSetup.exe`, a vytvoření *MSI* instalačního souboru pomocí sady nástrojů *WiX*.

Aplikace `WriteZipToSetup.exe` slouží, jak název napovídá, k vestavění ZIP archivu do `Setup.exe`, toho je dosaženo použitím vestavěných zdrojů, které umožňují modifikovat některá data v již sestavených spustitelných souborech, bez nutnosti nového sestavení. Což umožňuje distribuovat jediný soubor, který obsahuje vše potřebné pro instalaci aplikace a není nutné jej při každém vydání znovu sestavovat.

Sada nástrojů *WiX* obsahuje nástroje užitečné při vytváření *MSI* instalačních souborů, použitelných v operačním systémech *Windows*. První používaný nástroj je `candle.exe`

určený ke kompilaci zdrojových souborů *WiX* do objektových souborů `.wixobj`. Pomocí nástroje `light.exe` je z objektových souborů vytvořen soubor *MSI* a ten je již možné použít pro hromadnou instalaci pomocí zásad skupiny [26].

První nativní aplikace s příhodným názvem `Setup.exe` zajišťuje dostupnost .NET Frameworku ve správné verzi a první část instalace aplikace po stažení. Nativní aplikace je použita z toho důvodu, že na cílovém počítači nemusí být nainstalován .NET Framework a nativní aplikace se, na rozdíl od aplikace napsané v C# bez problému spustí. Úkolem *Setup.exe* je: v případě, že není nainstalovaný .NET Framework jej nainstalovat, následně vytvořit složku s názvem instalované aplikace v cestě definované proměnnou prostředí `LocalAppData`[9] a extrahovat přebalený balíček (včetně `Update.exe`) z vestavěných zdrojů do dočasné složky. Posledním úkolem je spuštění `Update.exe` s přepínačem `/install`. Tím je úloha aplikace `Setup.exe` dokončená a pro další běh, aktualizaci nebo odinstalaci již není nutná.

Po distribuci a první části instalace aplikace pomocí `Setup.exe` má `Update.exe` dalších několik úkolů

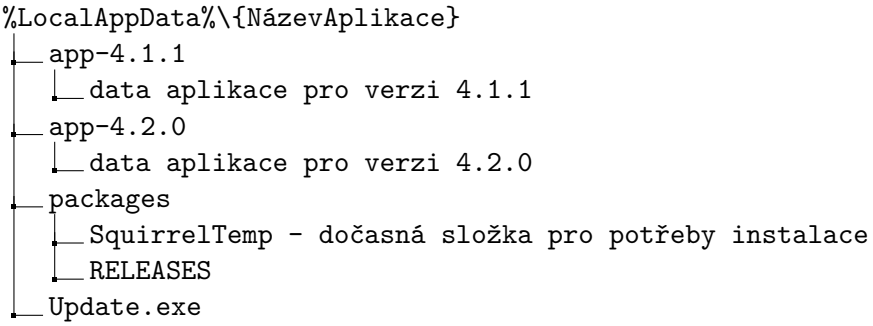
- Dokončení instalace
- Lokalizaci aplikace a její spouštění
- Aktualizaci a pozdější odinstalaci aplikace

Jak již bylo zmíněno dříve, práce `Setup.exe` končí spuštěním `Update.exe` s přepínačem `/install`, což zajistí dokončení instalace, tzn. načtení metadat (názvu a verze instalované aplikace) z dočasné složky, zkopírování dat do složky `app-{verzeAplikace}` a spuštění aplikace s parametry `--squirrel-install {verzeAplikace}`.

Jak bylo zmíněno v předchozím odstavci, aplikace se nachází ve složce v jejímž názvu je verze aplikace, kterou složka obsahuje tzn. v případě aktualizace vzniknou dvě složky, každá s jiným názvem, a tudíž se změní cesta ke spustitelnému souboru aktualizované aplikace, což by způsobilo nefunkčnost zástupců, odkazující se přímo na spustitelnou aplikaci. Tento problém je řešený tím, že se při instalaci vytvoří zástupce, který odkazuje na `Update.exe` s parametry `--processStart` a názvem aplikace. Při kliknutí na zástupce je nejdříve spuštěný `Update.exe`, který na základě údajů v souboru `RELEASES` lokalizuje složku, ve které se nachází nejnovější nainstalovaná verze aplikace a dle parametrů nalezne a spustí správný spustitelný soubor a svoji činnost ukončí. Souborová struktura kořenového adresáře aplikace po instalaci verze 4.1.1 a následné aktualizaci na verzi 4.2.0 je názorně ukázána na obrázku 3.3.

Při vydávaní nových sestavení je nutné někde udržovat seznam dostupných verzí spolu s kontrolními součty a případnými dalšími daty. Pro tyto účely framework využívá souboru `RELEASES`, jehož každý řádek specifikuje jedno vydání a má strukturu uvedenou v tabulce 3.1. Pomocí souboru `RELEASES` se kontroluje dostupnost nové verze a tudíž musí být dostupný na vzdáleném serveru. Další využití je čistě lokální, kdy soubor obsahuje informace o nainstalované verzi aplikace a na jeho základě se vybírá správná verze při spouštění, v tomto případě je soubor uložen v kořenovém adresáři nainstalované aplikace, hned vedle `Update.exe` a složek obsahující jednotlivé verze aplikace.

[9] Standardně obsahuje cestu složenou z proměnné prostředí `UserProfile` a `AppData\Local` tj. ve výchozím nastavení `C:\Users\{username}\Appdata\Local`



Obrázek 3.3: Souborová struktura kořenového adresáře aplikace.

SHA-1 hash balíčku	celý název balíčku	velikost v Bajtech

Tabulka 3.1: Struktura souboru `RELEASES`.

3.4.2 Vytváření instalačních balíčků

Pro vytvoření instalačních souborů je nutné sestavené data aplikace zabalit do NuGet balíčku, pro tento účel se v práci používá nástroj *Nuget.CLI*, ale lze použít i jiný, např. *NuGet Package Explorer*, pro využití *Nuget.CLI* je nutné použití souboru *nuspec* se specifikací obsahu balíčku. Cílem je vytvořit NuGet baliček obsahující složku `lib/net45`, která obsahuje vše potřebné pro běh aplikace, a soubor *nuspec* definující metadata balíčku, která budou použita při instalaci. Z vytvořeného NuGet balíčku se následně vytvoří instalátor, soubor pro plnou a případě i rozdílovou aktualizaci. To zajišťuje spuštění aplikace `Update.exe` s argumenty `--releasify {Cesta_k_NuGet_Balíčku}`, nástroj vytvoří (více informací o vytváření instalačních souborů v kapitole 3.4.1) složku *Releases* do které umístí následující soubory:

- `Setup.exe` při spuštění nainstaluje aplikaci.
- `Setup.msi` umožňuje jednoduchou instalaci aplikace na větším množství počítačů s pomocí zásad skupiny.
- `RELEASES` viz sekce 3.4.1.
- Soubor pro plnou aktualizaci `{PackageId}.{verze}-full.nupkg`, který obsahuje všechny soubory potřebné pro běh aplikace.
- Soubor pro rozdílovou aktualizaci `{PackageId}.{verze}-delta.nupkgd`

Při vytváření rozdílových aktualizací je nutné, aby byl dostupný instalační balíček (soubor `*-full.nupkg`) předchozí verze aplikace a soubor `RELEASES` vytvořený dříve. Poté `Update.exe` za pomocí binárního porovnání obsahu sestavených objektů (zpravidla spustitelných aplikací a dynamických knihoven) předchozí a nové verze vytvoří pro každý objekt soubor definující nutné změny předchozích objektů, pro dosažení nových (tzv. *patch* soubor), který je zpravidla (pokud se nezměnila celá knihovna) menší než sestavený objekt. Pro zajištění správnosti je také vypočítán kontrolní součet každého objektu nového sestavení. Tyto informace (patch soubory a kontrolní součty) jsou vloženy do rozdílového balíčku. Při instalaci rozdílové aktualizace se na objekty předchozí verze aplikují dříve zjištěné změny a verifikuje se zda jsou kontrolní součty objektů správné. V případě, že vše odpovídá je

nová verze standardně spuštěna, v opačném případě je provedena plná aktualizace, u které není možné, aby se kontrolní součty objektů lišily.

3.4.3 Distribuce instalačních souborů

Po vytvoření aktualizačních balíčků je potřeba mechanismus distribuce mezi uživatele aplikací, díky otevřenému zdrojovému kódu a promyšlenému architektonickému návrhu frameworku je možné distribuovat aktualizační balíčky téměř jakoukoliv cestou. Při využití obecného souborového serveru poskytující potřebné soubory dle názvu v *URL* není potřeba psát vlastní implementaci a je možné použít již implementovanou funkcionalitu, mezi další nativně podporovaná úložiště patří *Amazon S3* a *GitHub's releases*. V případě, že je požadavek na specifickou metodu distribuce aktualizačních souborů je nutné pro správnou funkci implementovat třídu implementující rozhraní `IFileDownloader`, které specifikuje dvě metody. První pro stažení souboru a vrácení jeho obsahu prostřednictví návratové hodnoty pole bajtů (`byte[]`), metoda je využívána pro stahování souboru `RELEASES`. Druhá metoda implementuje stažení souboru a jeho následné uložení, do adresáře specifikovaného parametrem využívaná pro stahování aktualizačních souborů obou typů (plné i rozdílové).

Pro jednoduché využití výsledků této práce je velice vhodné použít infrastrukturu, která je ve firmě Y Soft již zavedená a jsou s ní pozitivní zkušenosti. Pro ukládaní artefaktů[10] používá *JFrog Artifactory*. Toto řešení nabízí hostování více než 20 typů repozitářů [11] mezi něž patří i *NuGet repozitář*, který je pro hostování NuGet balíčků ideální řešení. *Artifactory* umožňuje do NuGet repozitáře uložit i jiná data, než jen NuGet balíčky čehož se využívá pro soubor `RELEASES` (jehož funkce byla zmíněna v kapitole 3.4.1) a také pro instalátory, které jsou vydávány pro poslední sestavení v každém kanále. NuGet repozitář hostovaný v *Artifactory* podporuje nahrávání artefaktů pomocí standardních nástrojů dostupných pro správu NuGet balíčků (viz 3.3), tyto nástroje ale podporují nahrávání pouze NuGet balíčků, nikoliv souboru `RELEASES` nebo instalátorů. Proto je pro nahrávání použité webové REST API, které umožňuje nahrávat jakýkoliv typ artefaktu včetně souboru `RELEASES`, mazat jednotlivé artefakty i celé složky, což se hodí zejména pro mazání již nepotřebných kanálů, a procházet dříve nahrané artefakty [10].

3.5 Možnosti grafických uživatelských rozhraní na platformě .NET

Během vývoje platformy .NET vzniklo mnoho frameworků pro grafické uživatelské rozhraní, které se liší svojí funkcionalitou, platformní závislostí mj. také použitým návrhovým vzorem. V případě, že není nutné vytvořit multiplatformní GUI jsou nejpoužívanějšími frameworky *Windows Forms* (zpravidla zkracované na *WinForms*) a *Windows Presentation Foundation* (zkráceně WPF).

Framework *WinForms* je dodáván jako součást .NET Frameworku již od jeho první verze, jde o sadu knihoven zpravidla obalující nativní *Windows API*, pomocí kterého je také většina grafických prvků vykreslována. Definice rozložení prvků v okně se vytváří pomocí WYSIWYG editoru, který na pozadí generuje kód sloužící k vykreslení nadefinovaného UI. *WinForms* neumožňuje jednoduché vytváření relativního rozložení prvků, ani složitější dynamickou změnu velikosti/pozice prvků při změně velikosti okna. Dalším problémem

[10] Za artefakt se považují sestavená, zpravidla binární, data např.: dll, jar, war soubory, NuGet/maven balíčky, docker kontejnery, instalátory, ...

jehož dopad se stále zvětšuje je chabá podpora displejů s vysokou hustotou pixelů (tzv. HiDPI displeje) na kterých vypadají aplikace rozmazaně nebo mají úplně rozbité rozložení prvků, velikost písma aj.. Základní podpora pro tento framework skončila již v r. 2011, rozšířená v r. 2016, ale Microsoft stále vydává opravy mj. i vylepšující podporu pro zmíněné HiDPI displeje. *WinForms* aktuálně využívají převážně dříve vyvinuté informační systémy a aplikace u kterých by se přepis do *WPF* nevyplatit, případně by naprosto nedával smysl. Pro nové aplikace už není doporučené *WinForms* používat.

Nejpoužívanějších GUI framework v .NET Platformě pro operační systém *Microsoft Windows* je aktuálně *WPF*. Jedná se o GUI framework s vektorovým jádrem, které není závislé na rozlišení a využívá grafické jádro pro akceleraci vykreslování UI. Definice rozložení a chování jednotlivých prvků UI se provádí pomocí značkovacího jazyka XAML a při spuštění aplikace se podle této definice jednotlivé prvky rozloží (není nutné generovat C# kód jako v případě *WinForms*). Díky skutečnosti, že XAML vychází z jazyka XML je možné vytvářet stromovou strukturu jednotlivých prvků. Aby bylo možné dát aplikacím s GUI ve WPF nějakou funkcionalitu, je XAML doplněný kódem v jazyce C# nebo Visual Basic specifikující akce vyvolané jednotlivými komponentami. WPF klade důraz na separaci kódu definujícího, jak má UI vypadat od kódu definující jeho chování, při implementaci GUI ve *WPF* se velice často používá návrhový vzor *MVVM*[11]. *WPF* je doporučený GUI framework pro všechny nové aplikace a v této době nemá žádné vážné limitace.

3.6 Sestavování projektů

Při sestavování komplexních projektů, kdy je nutné provést několik kroků (v terminologii sestavování *cílů*), které na sebe navzájem navazují a vytváří orientovaný graf cílů, produkují několik typů artefaktů a musí být opakovatelné je pravidlem využití nástroje pro automatizaci sestavení. Takový nástroj zajistí správné vykonání posloupnosti jednotlivých cílů a často nabídne i pomocné metody pro implementaci nejpoužívanějších cílů. Každý cíl má v rámci sestavení přesně daný úkol a také specifikován seznam cílů, které musí být vykonány před ním. Typická posloupnost cílů v projektech, které budou využívat výsledek této práce je přibližně následující:

1. *Checkout* – získání zdrojových kódů pro sestavení projektu z Git repozitáře
2. *Clean* – vyčištění pracovního adresáře od případných produktů předchozího sestavení a jiných nadbytečných souborů
3. *Restore* – stažení závislostí ze vzdáleného serveru, majoritní část závislostí tvoří NuGet balíčky
4. *Build* – vlastní sestavení projektu, zpravidla spuštění programu MSBuild se specifickými argumenty
5. *Test* – spuštění automatických testů nad nově sestaveným projektem, sběr výsledků a jejich uložení na předem dané místo
6. *Deploy* – uložení sestaveného projektu na vzdálený server a nahrání nových NuGet balíčků do repozitáře

[11] Návrhový vzor sestávající z **m**odelu, **v**iew, a **v**iew **m**odelu

V projektech, které budou výsledky práce využívat se používá nástroje *nuke*. *Nuke* je multiplatformní nástroj pro automatizaci sestavování primárně C# projektů s otevřeným zdrojovým kódem, který využívá aktuálních nástrojů pro vývoj v jazyce C# a umožňuje definovat cíle sestavení pomocí kódu v jazyce C# [13]. Jednotlivé cíle se často skládají z volání jiných programů se specifickými argumenty a zajímavostí nástroje *nuke* je, že kód pomocí kterého se ostatní programy volají je často generován na základě jejich dokumentace. Při sestavování se také používá knihovna vyvinutá ve firmě Y Soft *Nuke.Helpers*, která zjednodušuje definici běžných cílů a do které budou přidány pomocné metody, usnadňující využití vyvíjeného frameworku.

3.7 Nástroj Consul a knihovna consuldotnet

Pro zajištění přehledu o spuštěných aplikacích, jejich verzích a případných dalších metadatech bude v rámci práce využito nástroje pro kontrolu stavu aplikací, který má přehled o registrovaných aplikacích, metodách pro kontrolu jejich stavu a také kontrolu stavu provádí. Nástroje jsou zpravidla distribuované, vysoce dostupné, škálovatelné a odolné vůči chybám. Během registrace do nástroje se specifikuje název aplikace, identifikátor, síťové umístění a metody pomocí kterých se má kontrola stavu provádět. Nejčastěji používané metody pro kontrolu stavu jsou následující:

- Navázání TCP spojení – úspěšné, pokud je přijato ve specifikovaném časovém intervalu
- HTTP požadavek – úspěšné, pokud je stavový kód odpovědi v intervalu $\langle 200, 299 \rangle$

Již používaným nástrojem pro kontrolu stavu ve firmě Y Soft je *Consul*, který nabízí veškerou potřebnou funkcionalitu a otevřený zdrojový kód [9]. *Consul* disponuje jednoduchým webovým rozhraním, které umožňuje prohlížení aktuálně zaregistrovaných aplikací a služeb, avšak veškerá registrace a změna parametrů probíhá pomocí HTTP REST API. Pomocí REST API je také možné zjistit např. aktuální stav aplikací i jednotlivých metod pro kontrolu stavu, nad tímto rozhraním je postavená knihovna *consuldotnet* distribuovaná jako NuGet balíček a umožňující jednoduché využití všech možností *Consul*u z prostředí jazyka C# [16].

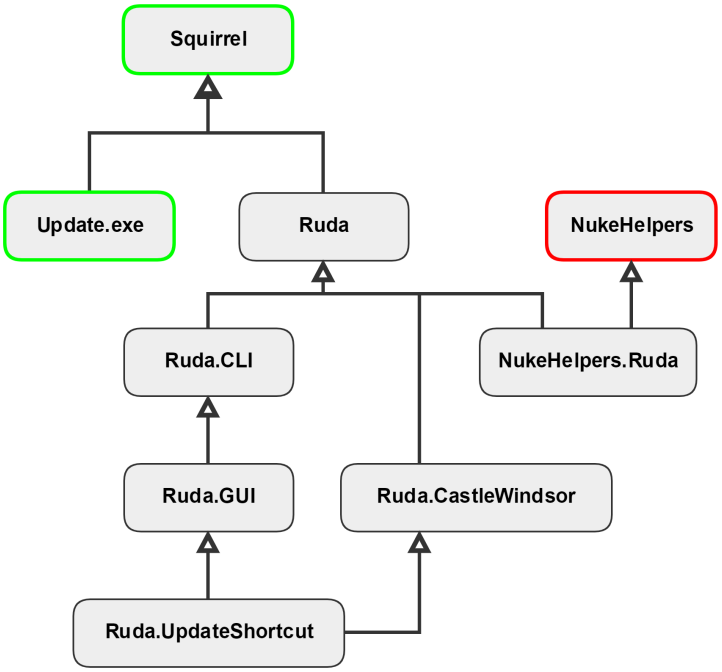
Kapitola 4

Návrh řešení

V této kapitole je popsáno plánované řešení a architektura vytvářeného frameworku pro aktualizaci aplikace, návaznosti, význam a zodpovědnost jednotlivých implementovaných knihoven. V druhé části kapitoly je popsán návrh řešení kontroly stavu aplikací.

4.1 Návrh řešení aktualizace aplikace

Pro větší přehlednost a oddělení zodpovědnosti za jednotlivé části funkcionality bude výsledný framework rozdělen do několika knihoven. Jejich závislosti je možné vidět na obrázku 4.1, kde každý obdélník znázorňuje jednu knihovnu nebo spustitelný soubor.



Obrázek 4.1: Závislosti knihoven a aplikací frameworku *Ruda*.

Zeleně ohraničené jsou objekty, které již byly implementovány (knihovna *Squirrel.Windows*), ale bude nutné je přizpůsobit pro plánované určení. Červeně je ohraničena knihovna, která

byla implementována ve firmě Y Soft již dříve a jejíž modifikace nebude nutná. Ostatní objekty budou implementovány v rámci této práce.

Volba programovacího jazyka je vzhledem k dosavadním zkušenostem a aktuálnímu programovacímu jazyku většiny komponent, kterých se bude výsledek této práce využívat velice jednoduchá, půjde o jazyk C# ve verzi 7.1.

4.1.1 Návrh modifikace frameworku Squirrel.Windows

Jak již bylo zmíněno dříve v kapitole 3.4 knihovna `Squirrel` nepodporuje sémantické verzování ve verzi 2.0.0. Místo toto používá předchozí verzi 1.0.0, jejíž implementace je pomocí Git submodulů převzata ze zdrojových kódů klienta NuGet v3 [1]. Vzhledem k následujícímu bouřlivému vývoji kolem balíčkovacího systému NuGet a vydání nové majoritní verze klienta i specifikace protokolů již není kód udržován. Knihovna NuGet bude nahrazena, novější a zatím nejlepší nalezenou alternativou je knihovna `NuGet.Common`, která zahrnuje novou implementaci sémantického verzování s podporou sémantického verzování 2.0.0. Stejná situace je i u programu `Update.exe`, který je závislý na `Squirrel` a tudíž tranzitivně i na submodulu NuGet. Bohužel `Squirrel` používá zastaralou knihovnu i pro opětovné balení NuGet balíčků, tato funkcionalita je nyní dostupná v knihovně `NuGet.Packaging`.

Na knihovně `Squirrel` bude záviset i knihovna `Ruda`, `NukeHelpers.Ruda` a další, viz obrázek 4.1 tyto knihovny budou psány tak, aby odpovídaly specifikaci *.NET Standard* a jejich běh byl možný na platformě *.NET Core* to znamená že můžou záviset pouze na knihovnách, které jsou zkompilovány pro *.NET Standard*, nikoliv na dříve běžně využívaných knihovnách pro *.NET Framework*. Pro vyřešení této situace bude nutné převést projekt `Squirrel` do nového tzv. *SDK Projektu*, který umožňuje kompilovat zdrojové soubory pro více různých cílových frameworků, v tomto případě .NET Framework 4.6.1, na kterém bude záviset `Update.exe`, a .NET Standard na kterém budou záviset všechny zbývající knihovny.

Nakonec bude potřeba vyřešit distribuci samostatné knihovny `Squirrel`, která je nyní staticky linkována s `Update.exe` a v NuGet balíčku *Squirrel.Windows* se samostatně nenachází. Tento problém bude vyřešen vytvořením nového NuGet balíčku *Squirre.Windows.Core*, který bude obsahovat pouze samostatnou knihovnu `Squirrel` a umožní ostatním projektům jednoduché přidání závislosti. Ačkoliv se na první pohled může zdát rozumné nepřidávat nově vytvořený balíček jako závislost pro *Squirrel.Windows* protože je `Update.exe` s knihovnou staticky linkován není tomu tak. `Update.exe` je samostatný spustitelný soubor, který umožňuje funkcionalitu popsanou v 3.4.1, ale není možné jej využít jako dynamickou knihovnu (využívat definovaných tříd a metod) a tudíž by v případě nepřidání závislosti nebylo možné aktualizaci programově ovládat. Původní NuGet balíček tuto situaci řešil tím způsobem, že obsahoval `Update.exe` (který je staticky linkován s knihovnou `Squirrel`) i samostatnou knihovnu `Squirrel` pro využití dalšími knihovnami nebo aplikacemi. Modifikace nativních aplikací nejspíš nebude nutná.

4.1.2 Návrh frameworku pro správu verzí aplikací

Jak je patrné z obrázku 4.1 na začátku této, kapitoly pro framework, který bude implementovaný v rámci této práce byl vymyšlen název *Ruda* (pocházející z **R**obot **upd**ater **a**pparatus). Při vývoji *Rudy* bude kladen důraz na rozdělení zodpovědnosti za jednotlivé funkcionality mezi několik navzájem spolupracujících knihoven. Takový návrh umožní poz-

[1] viz repozitář *Squirrel.Windows* https://github.com/Squirrel/Squirrel.Windows/tree/1.8.0/vendor

ději využít i jen část funkcionality a nepřidáním závislostí na všechny knihovny ze kterých se bude *Ruda* skládat zároveň ušetří místo.

Knihovna Ruda

Knihovna `Ruda` bude obsahovat jednotné jádro umožnující využití fundamentálních částí frameworku pro základní operace s verzemi aplikace, mezi které patří zejména zjišťování dostupnosti nové verze, aktualizace aplikace na poslední dostupnou verzi v aktuálním kanálu a aktualizaci aplikace na specifickou verzi a to i starší, než je aktuálně nainstalovaná. Řešení samotného procesu aktualizace aplikace není zodpovědnost knihovny `Ruda`, ale je již implementováno v aplikaci `Update.exe`. Důležitou součástí této knihovny bude vytvoření nového rozhraní `IChanneledDownloader` umožnující práci se vzdáleným serverem, podporující práci s kanály (jejich enumeraci a ověřování existence na vzdáleném serveru) a implementující rozhraní `IFileDownloader` používané frameworkem *Squirrel.Windows*. Bude vytvořena referenční implementace `IChanneledDownloader` použitá pro stahování metadat, informací o kanálech a aktualizačních balíčků ze serveru *JFrog Artifactory*.

Vzhledem ke skutečnosti, že bylo využito specifikace sémantického verzování 2.0.0, jenž oproti verzi 1.0.0 nabízí metadata bude knihovna `Ruda` obsahovat metody určené k parsování těchto metadat a k převodu získaných dat do podoby kolekce prvků typu klíč-hodnota.

Díky skutečnosti, že knihovna nemusí záviset na jiné, která byla vytvořena výlučně pro .NET Framework, bude možné ji sestavit pro .NET Framework i .NET Standard. Sestavení knihovny pro .NET Framework z důvodů zmíněných výše 3.1 není nutné, proto bude sestavena pouze pro .NET Standard.

Knihovna Ruda.CLI

Framework *Ruda* nabídne několik různých metod interakce s uživatelem a jednou z nich bude i prostředí příkazové řádky, které umožní knihovna `Ruda.CLI`. Mezi výhody řešení uživatelského rozhraní pomocí příkazové řádky oproti grafickému rozhraní patří jednoduchost implementace, možnost použití skriptů pro automatizaci a minimum závisejících knihoven. Pomocí této knihovny bude možné provádět minimálně následující úkony:

- aktualizace aplikace
  - na poslední verzi v aktuálním kanálu
  - na jakoukoliv verzi v kterémkoliv kanálu
- zobrazit informace o:
  - aktuálně nainstalované verzi včetně kanálu a metadat
  - dostupných kanálech
  - dostupných verzích

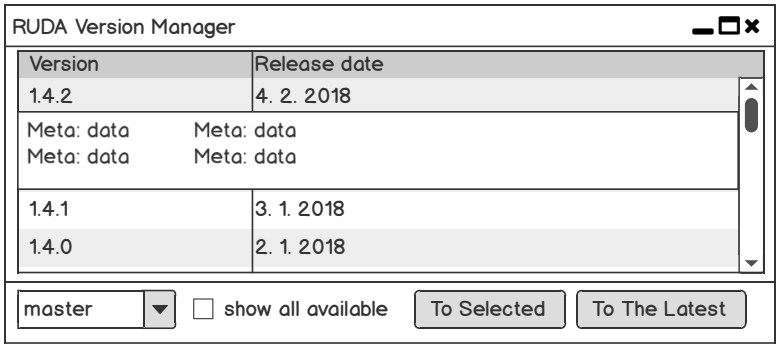
Jak je patrné z předchozího výpisu, bude `Ruda.CLI` dostatečný nástroj pro kompletní správu aplikace a může nalézt využití při automatizované instalaci/aktualizaci, kde by grafické rozhraní jen zvyšovalo složitost a náročnost údržby kódu provádějícího automatizované kroky. Rozhraní příkazové řádky bude mít oproti grafickému jednu funkci navíc a to možnost zvolit si, zda se má aplikace po aktualizaci restartovat nebo zda se má ukončit a vyčkat na případné další spuštění skriptem nebo uživatelem.

Důležitým prvkem knihovny `Ruda.CLI` bude možnost přidání dalších definic možných argumentů spuštění ostatními knihovnami. Této vlastni bude využito v knihovně frameworku zajišťující grafické rozhraní (`Ruda.GUI`), kdy bude možné pomocí definice argumentů předané knihovně `Ruda.CLI` a následného spuštění programu s těmito definovanými argumenty spustit grafické rozhraní z prostředí příkazové řádky.

Knihovna `Ruda.CLI` stejně jako `Ruda` nebude nezáviset na žádné knihovně vytvořené výlučně pro .NET Framework a bude sestavena pouze pro .NET Standard.

Knihovna Ruda.GUI

Interakce s uživatelem pomocí rozhraní příkazové řádky je sice jednoduchá, avšak pro většinu uživatelů není přívětivá, natož pohodlná. Proto bude prostřednictvím knihovny `Ruda.GUI` realizováno i grafické uživatelské rozhraní, které nabídne očekávanou přívětivost, pohodlnost a přehlednost. Funkcionalita grafického rozhraní bude, díky skutečnosti, že staví na stejném základu, přibližně stejná s dříve popsanou funkcionalitou CLI tzn. aktualizace aplikace na vybranou verzi, přepínaní a zobrazování dostupných kanálů a přehledné zobrazení podrobných informací o každé vydané verzi.



Obrázek 4.2: Návrh grafického rozhraní frameworku.

Návrh uživatelského rozhraní byl vytvořen v programu *Balsamiq* s využitím prvků dostupných grafickém subsystémem *Windows Presentation Foundation* zkráceně *WPF*. Při návrhu byl kladen důraz na co nejjednodušší provedení nejčastějších operací, těmi jsou: aktualizace na poslední dostupnou verzi v aktuálním kanálu a aktualizaci na poslední dostupnou verzi v jiném kanálu. Provedení první operace je na jediné kliknutí (tlačítko „*Update to the latest*“), druhá operace sestává z výběru kanálu v kombinovaném poli a kliknutí na tlačítko „*Update to the latest*“. Na obrázku 4.2 je patrné celé uživatelské rozhraní, majoritní část plochy zabírá tabulka s přehledem vydaných verzí v právě vybraném kanálu. Ve spodní části okna je umístěné zaškrtávací pole umožnující zobrazit všechny dostupné verze v aktuálním kanálu, ve výchozím nastavení je zobrazeno jen několik posledních. Po kliknutí na verzi v tabulce se v podobě klíč-hodnota zobrazí metadata vybrané verze. V levé spodní části se nachází kombinované pole umožňující zvolení aktualizačního kanálu a v pravé spodní části se nachází dvojice tlačítek, první pro aktualizaci na právě zvolenou verzi v tabulce a druhé pro aktualizaci na poslední dostupnou verzi ve zvoleném kanálu.

Grafický subsystém WPF je kvůli závislostem na knihovně `User32` z *Windows API* a *Direct3D* možné sestavit pouze pro .NET Framework a proto musí být i knihovna `Ruda.GUI` sestavena pro .NET Framework.

Aplikace Ruda.UpdateShortcut a knihovna Ruda.CastleWindsor

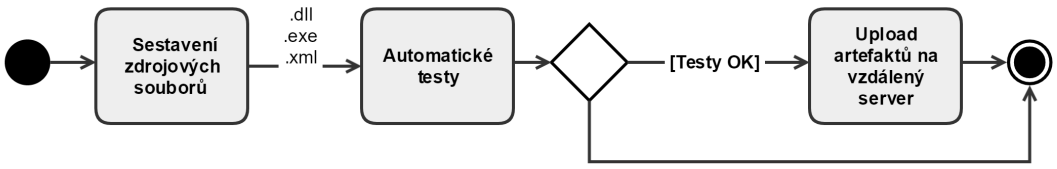
Pomocná aplikace `Ruda.UpdateShortcut` a knihovna `Ruda.CastleWindsor` budou vytvořeny pro jednodušší integraci frameworku do stávajících, případně i dalších aplikací. Úkolem aplikace `Ruda.UpdateShortcut` bude zobrazení grafického rozhraní frameworku bez závislosti na aplikaci, která jej integruje. Takové chování se může hodit v případě, že je framework využíván v aplikaci bez grafického uživatelského rozhraní a není žádoucí vytvářet závislosti mezi aplikací a grafickými knihovnami.

Většina aplikací, do kterých se bude framework integrovat obsahuje IoC (Inversion of Control[2]) kontejner *CastleWindsor*, knihovna `Ruda.CastleWindsor` bude určena přesně pro tyto aplikace kde usnadní registraci všech komponent a jejich následné použití. Registrace bude provedena pomocí metody rozšiřující rozhraní `IWindsorContainer` o metodu `UseRuda()` s případnými argumenty umožnující konfiguraci frameworku.

Aplikace `Ruda.UpdateShortcut` i knihovna `Ruda.CastleWindsor` budou vzhledem k závislosti na knihovnách vyžadujících .NET Framework sestaveny pouze pro něj. Knihovna `Ruda.CastleWindsor` by čistě teoreticky mohla být rozdělena ještě více: na část neregistrující grafické uživatelské rozhraní, která by mohla být sestavena i pro .NET Standard, a na část přidávající podporu registrace grafického uživatelského rozhraní, která by musela být sestavena pouze pro .NET Framework. Toto řešení aktuálně není potřebné, aplikace, kde se o využití knihovny `Ruda.CastleWindsor` uvažuje jsou sestavené pro .NET Framework, a rozdělení je možné realizovat i v budoucnu.

4.1.3 Návrh změn procesu sestavení a knihovny NukeHelpers.Ruda

Nyní se ve firmě Y Soft pro sestavení projektů, které budou využívat framework *Ruda*, používá nástroj pro automatizaci sestavení *Nuke* společně s již implementovanou knihovnou `NukeHelpers` ulehčující často prováděné kroky. Zjednodušený, nyní používaný, postup sestavení nové verze aplikace je na obrázku 4.3, obrázek neobsahuje úkony týkající se získání zdrojového kódu z repozitáře, ani vytváření a publikování knihoven v podobě NuGet balíčků, jelikož to není v kontextu implementace `NukeHelpers.Ruda` podstatné.



Obrázek 4.3: Zjednodušený proces sestavení, nyní.

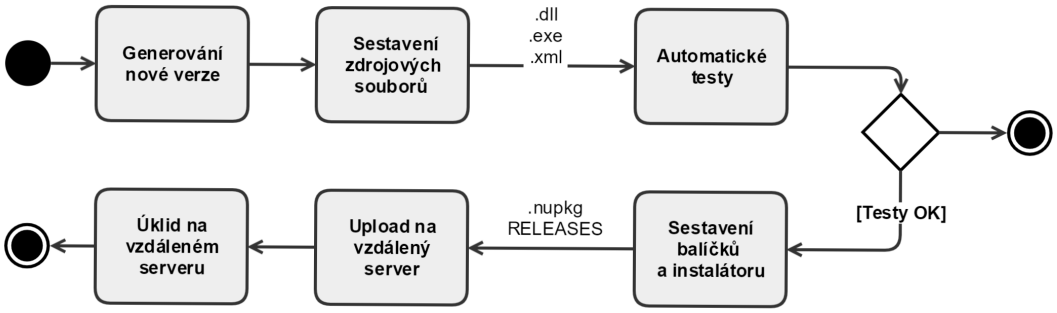
Pro potřeby využití frameworku *Ruda* by měl postup sestavení vypadat přibližně jako na obrázku 4.4, ze kterého je patrné, že přibylo několik *cílů*.

- Generování nové verze
- Sestavení balíčků a instalátoru
- Upload na vzdálený server

[2]Návrhový vzor, umožňující uvolnit vztahy mezi komponentami, kdy kontejner zajišťuje vytvoření a dodání komponenty potřebného typu/funkcionality na správné místo. [19]

- Úklid na vzdáleném serveru

Implementace těchto cílů bude provedena v rámci knihovny `NukeHelpers.Ruda`. Pro generování verze sestavení se použije knihovna zmíněna v kapitole 3.2 *GitVersion*, která na základě konfigurace a stavu repozitáře navrátí proměnné z nichž se vytvoří plné znění sémantické verze 2.0.0 tj. včetně identifikátorů předchozího sestavení a metadat.

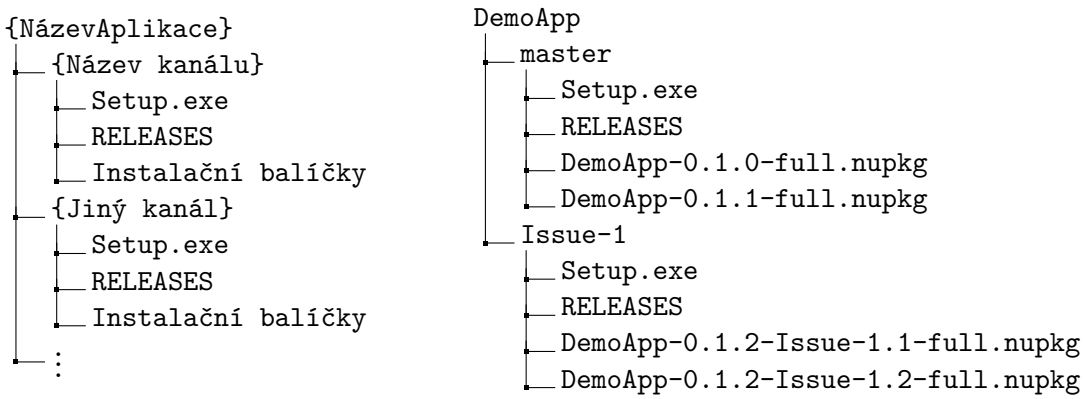


Obrázek 4.4: Zjednodušení proces sestavení s frameworkem *Ruda*.

Dalším implementovaným cílem bude vytváření instalačních balíčků a instalátorů, pro správné vytvoření instalačního balíčku bude nutné zjistit, zda se jedná o první sestavení daného kanálů. V případě že již kanál byl sestavován bude nutné před vytvořením balíčku stáhnout balíček dřívější společně se souborem `RELEASES`. Díky tomu bude možné vytvořit rozdílový balíček a přidat záznamy do souboru `RELEASES`. Po případném stažení předchozího sestavení bude na základě definice `nuspec` a sestavených objektů ve výstupní složce projektu vytvořen NuGet balíček. Ten bude následně předán aplikaci `Update.exe` (viz 3.4.1), jež vytvoří instalační balíček. V případě existence souboru `RELEASES` přidá záznamy o novém sestavení a vytvoří rozdílové balíčky, v opačném případě soubor `RELEASES` vytvoří jen s aktuálním sestavením. Po dokončení tohoto kroku budou ve specifikované složce připravené soubory pro upload na vzdálený server.

Pro správnou funkci frameworku *Ruda* bude nutné dodržení souborové struktury na vzdáleném serveru tzn. upload instalačního balíčku do správného kanálu a nahrazení správného souboru `RELEASES`. Předpokládaná souborová struktura je na 4.5. Využití běžných cest pro publikování NuGet balíčku v tomto případě není vhodné, protože soubor `RELEASES` není NuGet baliček a jeho publikování není touto cestou možné. Proto bude upload souborů na vzdálený server prováděn pomocí REST API rozhraní úložiště *Artifactory* a metody HTTP PUT.

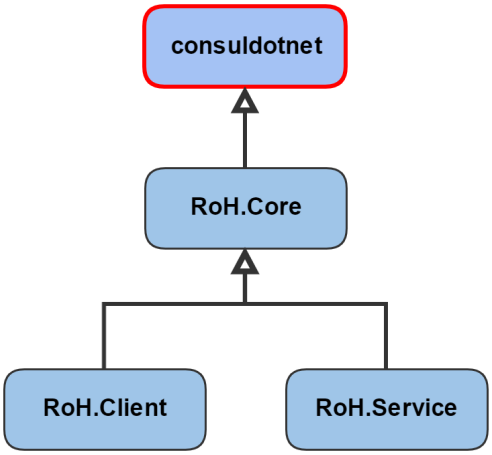
Při vývoji aplikace ve více lidech, nočních sestaveních a publikování instalačních souborů po každé změně zdrojových kódů bude vznikat velké množství instalačních souborů, které rychle zastarají a jejichž uchování není smysluplné. Proto bude důležitou součástí knihovny `NukeHelpers.Ruda` úklid na vzdáleném serveru alias mazání starých kanálů a sestavení. Při úklidu se budou mazat celé kanály, pro které už neexistují vývojové větve v Gitu a starší instalační balíčky. Při mazáno starších instalačních balíčků bude nutné stáhnout soubor `RELEASES`, vymazat z něj záznamy o smazaných instalačních balíčcích a následně jej nahrát zpět.



Obrázek 4.5: Souborová struktura na vzdáleném serveru. Na levé straně schéma, na pravé straně reálná struktura aplikace *DemoApp* se zanedbanými metadaty u verzí v názvu souboru.

4.2 Návrh frameworku pro kontrolu stavu aplikací

Framework pro kontrolu stavu aplikací bude z důvodu již probíhajícího využití nástroje *Consul* ve firmě Y Soft (viz kapitola 3.7) pro jiné účely, postaven právě na tomto nástroji. Framework bude stejně jako *Ruda* řešen modulárně, avšak knihoven bude méně a budou děleny jiným způsobem.



Obrázek 4.6: Závislosti frameworku *RoH*.

Na obrázku 4.1 jsou znázorněné plánované závislosti jednotlivých knihoven frameworku *RoH* (zkratka z **Ro**bot **H**ealthcheck). Blok s červeným okrajem je knihovna `consuldotnet`, jejíž implementace není předmětem této práce, černý okraj mají knihovny ze kterých se skládá framework *RoH* a jejichž implementace je předmětem této práce. Ústřední knihovnou na které bude celá funkcionalita frameworku záviset bude `consuldotnet`, která obaluje HTTP API nástroje *Consul*, jež se bude starat a samostatné zjišťování stavu jednotlivých komponent. Celý framework *RoH* bude, díky skutečnosti, že `consuldotnet` je sestaven pro .NET Standard a nebudou využívány žádné další knihovny, sestaven výhradně pro

.NET Standard což umožní jeho použíti v nejširší možné škále aplikací a služeb (viz kapitola 3.1) nezávisle na frameworku *Ruda*.

Jak je již zmíněno v kapitole 3.7 *Consul* používá pro rozlišení jednotlivých služeb jejich název a pro rozlišení instancí služeb unikátní řetězcový identifikátor. Každá instance služby může mít také přiřazené své *značky* (angl. *tagy*), což jsou jednoduché řetězce, a *metadata*, což jsou párové řetězcové hodnoty typu klíč-hodnota, tyto dodatečná data je možné následně použít při vyhledávání konkrétních instancí. Framework *RoH* bude umožňovat využití funkcionality tagů, funkcionalita metadat v rámci této práce implementována nebude. Je velice výhodné jak název služby, tak identifikátor instance sestavit pouze ze znaků, které můžou být použity pro doménové jméno[3] což umožní v budoucnu využít DNS server integrovaný v *Consulu* pro přímé propojení služeb prostřednictvím vhodně zvolených doménových jmen.

Navržené schéma pro pojmenování služeb a jejich instancí vychází ze skutečností, že jedna aplikace může zpřístupňovat žádnou nebo více služeb a služba může (ale nemusí) běžet v rámci aplikace a má následující podobu:

název aplikace	název služby	identifikátor hosta

Kde jednotlivé části budou mezi sebou odděleny znakem mínus („`-`“) takovým způsobem, že za názvem aplikace bude právě jedno mínus a to i v případě, že název služby a/nebo identifikátor hosta není vyplněn nebo v případě, že název aplikace není vyplněn, ale název služby je. Název aplikace a služby budou frameworku předané prostřednictvím konfigurace a identifikátor hosta se bude generovat na základě vlastností (virtuálního) stroje na kterém daná instance aplikace poběží. Příklady identifikátorů:

- validní identifikátor aplikace: `app-`
- nevalidní identifikátor aplikace: `app--`
- validní identifikátory služby
  - bez aplikace: `-svc`
  - s aplikací: `app-svc`
- validní identifikátory instancí služeb/aplikací
  - aplikace „`app`“ na hostu „`node`“: `app--node`
  - služba „`svc`“ na hostu „`node`“: `-svc-node`
  - aplikace „`app`“ poskytující službu „`svc`“ na hostu „`node`“: `app-svc-node`
  - aplikace „`app`“ na hostu „`-`“: `app---`
  - služba „`svc`“ na hostu „`-`“: `-svc--`

Jak je patrné z uvedených příkladů identifikátory můžou nabývat velmi různorodých hodnot a je nutné, aby všechny tyto hodnoty byly správně generované a následně i správně parsované[4].

[3] viz publikace RFC952 https://tools.ietf.org/html/rfc952 a upřesňující informace v RFC1123 https://tools.ietf.org/html/rfc1123#page-13

[4] Analyzované a rozdělené na trojici řetězců aplikace–služba–host

4.2.1 Knihovna RoH.Core

Ve chvíli, kdy je framework rozdělen na několik částí a různé část frameworku jsou používány na různých místech – `RoH.Client` v aplikaci, která bude vyžadovat zjištění stavu aplikací a `RoH.Service` v aplikaci jejíž stav, případně stav jejichž služeb, má být zjišťován je důležité aby se stavělo na stejném jádře, které zajistí bezproblémovou vzájemnou kompatibilitu. Mezi fundamentální prvky, které musí být stejné v rámci celého systému je kód pro vytváření a parsování identifikátorů, právě tento kód bude implementován v rámci knihovny `RoH.Core`. Další části, které budou v této knihovně implementované je jednotná konfigurace nástroje *Consul* a funkcionalita publikování značek do *Consulu*.

4.2.2 Knihovna RoH.Service

Pro jednoduchou registraci služby a/nebo aplikace do nástroje *Consul* a zajištění správného nastavení kontroly stavu bude vytvořena knihovna `RoH.Service`. Stav aplikací se bude zjišťovat prostřednictvím pokusu o TCP spojení na počítač, kde aplikace poběží. Tato knihovna bude zajišťovat registraci kontroly stavu a jelikož aplikace jako taková zpravila neposkytuje žádný TCP server, prostřednictvím kterého by bylo možné stav kontrolovat, bude obsahovat i implementaci jednoduchého TCP serveru. Úkolem implementovaného TCP serveru bude ustavit na volném portu TCP server, na kterém bude přijímat požadavky na připojení, ale vzápětí bude spojení uzavírat. Díky tomu, že *Consul* vyžaduje pro úspěšnou kontrolu stavu pomocí TCP pouze přijetí spojení ve specifikovaném časovém intervalu a neočekává žádnou další odpověď, je možné spojení uzavřít ihned po jeho přijetí.

Další součástí knihovny `RoH.Service` bude mechanismus pro kontrolu stavu služeb, které, jak již bylo zmíněno výše, můžou, ale nemusí běžet v rámci aplikace. Zpravidla se bude jednat o služby, které mají nějakou formu HTTP případně gRPC serveru a nebylo by optimální jejich kontrolu provádět pomocí pokusu o TCP spojení, z toho důvodu knihovna nabídne možnost definovat jeden či více typů kontrol stavu dané služby s využitím nástroje *Consul*. Mezi podporované typy kontrol stavu patří HTTP spojení na specifikovanou URL i gRPC Health Checking Protocol[5] [8].

Knihovna `RoH.Service` bude nabízet i možnost přidání značek k jednotlivým instancím aplikací/služeb. A to pomocí rozhraní využívající generický typ `ObservableCollection`, který umožňuje na modifikaci kolekce navázat akci, jež bude modifikovanou kolekci automaticky přenášet do nástroje *Consul* a ten tak bude reflektovat aktuální stav značek bez nutnosti jejich explicitní aktualizace.

4.2.3 Knihovna RoH.Client

Aplikace a služby se do nástroje registrují pomocí knihovny *consuldotnet* zmíněné dříve a pro získání seznamu dostupných aplikací/služeb z nástroje *Consul* bude vytvořena knihovna `RoH.Client`. Jejím úkolem bude obalit API knihovny `consuldotnet` takovým způsobem, aby bylo možné získat seznam právě běžících a aplikací/služeb pomocí jejich názvu. Jak již bylo zmíněno výše v *Consulu* se budou používat identifikátory navržené pro tento framework a při dotazování není pohodlné, vzhledem k jejich velké variabilitě, využívat řetězcovou hodnotu identifikátoru. Proto bude využita knihovna `RoH.Core`, která zajistí vytvoření identifikátoru podle daného schématu z dvojice *název aplikace* a *název služby*, název hosta, na kterém aplikace běží není pro vyhledávání žádoucí a proto se nebude používat.

[5] Podrobnosti o gRPC Health Checking Protocolu viz https://github.com/grpc/grpc/blob/master/doc/health-checking.md

V seznamu, který bude knihovna vracet bude možné zjistit stav jednotlivých kontrol pro každou instanci aplikace/služby, jejich IP adresu a port na kterém daná instance služby/aplikace běží. Také bude možné získat seznam značek pro každou instanci a plný identifikátor této instance.

4.2.4 Návrh integrace přehledu stavu aplikací/služeb do stávajícího webového rozhraní

Projekt, do kterého bude celý výsledek práce integrován nabízí webové rozhraní umožnující mj. konfiguraci různých komponent v rámci projektu. Mezi cíle této práce patří i přidání možnosti sledovat stav instancí služeb/aplikací včetně možností zobrazení značek. Webové rozhraní, do kterého se bude funkcionalita přidávat, je vytvořeno pomocí frameworku *DotVVM* a jak je již z názvu patrné využívá návrhový vzor MVVM. Do stávajícího webového rozhraní bude přidána nová stránka, která bude zobrazovat seznam dostupných služeb/aplikací a po kliknutí na službu/aplikaci se v přehledné formě zobrazí jednotlivé instance včetně plného názvu, IP adresy, portu, jejich aktuálního stavu a případných značek.

Vzhledem k použití návrhového vzoru MVVM bude implementován *view model*, který bude pomocí knihovny `RoH.Client` získávat informace o dostupných službách a aplikacích. Na základě těchto informací naplní *model*, který se bude prezentovat pomocí implementovaného *view*. Pro dosažení potřebné funkcionality ale bude potřeba implementovat dvě trojice *M-V-VM*, první pro základní přehled o dostupných službách a aplikacích, druhou pro detailní přehled o instancích.

4.3 Shrnutí

Kapitola 4 popisovala návrh rozdělení frameworků *Ruda* a *RoH* na jednotlivé komponenty, přiblížila závislosti jednotlivých komponent. U frameworku *Ruda* ukázala mj. návrh uživatelského rozhraní pro knihovnu `Ruda.GUI` i návrh modifikace procesu sestavení tak, aby bylo možné snadno a automaticky vydávat nové verze aplikací. U frameworku *RoH* bylo mj. popsáno schéma pro identifikátory instancí aplikací a služeb, navržen princip funkce frameworku a navržena implementace přehledu do webového rozhraní.

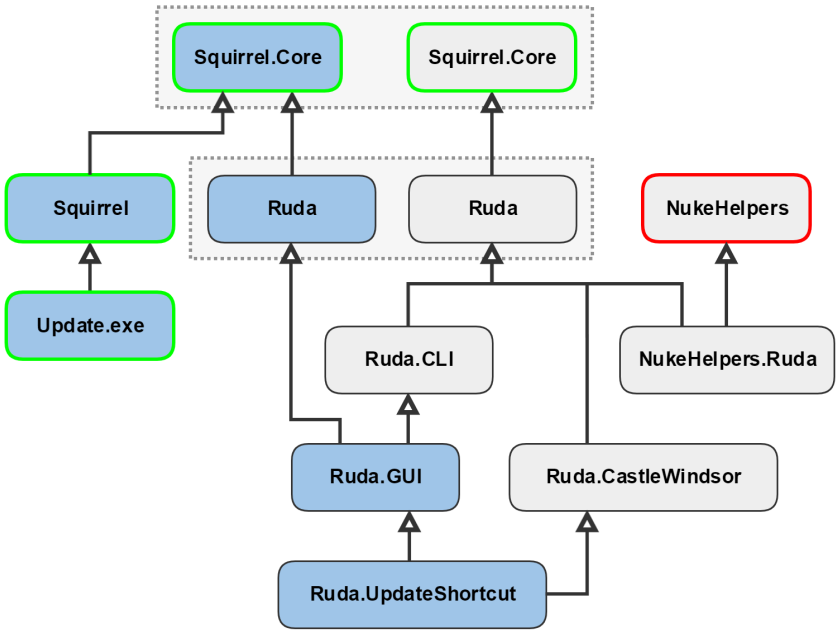
Kapitola 5

Implementace řešení

Kapitola popisuje nutné modifikace využívané knihovny `Squirrel.Windows` a vybrané implementační detaily knihoven tvořící frameworky *Ruda* a *RoH*. Zmiňuje, kdy a z jakého důvodu bylo nutné se odchýlit od dříve popsaného návrhu.

5.1 Implementace systému pro správu verzí aplikací

Při implementaci frameworku *Ruda* bylo postupováno podle návrhu avšak některé části bylo nutné přehodnotit. Na obrázku 5.1 je výsledný strom závislostí jednotlivých projektů, modrou barvou jsou projekty sestavené pro .NET Framework, šedou .NET Standard. U projektů `Squirrel.Core` a `Ruda` bylo nutné využít možnosti sestavení pro více cílových platforem, což je v obrázku naznačeno tečkovaným ohraničením.



Obrázek 5.1: Závislosti frameworku *Ruda*.

5.1.1 Modifikace Squirrel.Windows

Jak již bylo zmíněno v návrhu 4.1.1 celý framework byl před započetím prací kompatibilní se sémantickým verzování ve verzi 1.0.0 implementovaným pomocí submodulu se staršími zdrojovými kódy NuGet klienta. Submodul byl z projektu odstraněn a nahrazen knihovnou `NuGet.Versioning` obsahující implementaci sémantického verzování 2.0.0 knihovna má rozdílné API takže bylo nutné provést ještě několik desítek změn v kódu avšak bez dopadu na jeho sémantiku. Nejvíce změn se bylo nutné provést kvůli jinému pojmenování a rozložení tříd ve jmenných prostorech a také u parsování a přístupu k dílčím částem sémantické verze (číslo *Major*, *Minor*, *Patch* a identifikátor předběžné verze). Dále bylo nutné využívat plné znění sémantické verze tj. včetně metadat ve většině částí kódu. Modifikace nativních aplikací nebyla nutná. Těmito kroky byla vyřešena podpora sémantického verzování 2.0.0. Submodul zahrnoval i funkcionalitu pro rozbalení a zabalení NuGet balíčků, čehož framework využíval k získávání informací o NuGet balíčcích určených pro vytvoření instalačního souboru. Tato funkcionalita byla nahrazena knihovnou `NuGet.Packaging`, která měla stejně jako `NuGet.Versioning` nekompatibilní API a proto muselo být provedeno několik dalších změn ve zdrojových souborech frameworku. Konkrétně bylo třeba změnit metody pro čtení NuGet balíčku a modifikovat způsoby přístupu k metadatům (parsovaný soubor *nuspec*) NuGet balíčků.

Při příležitosti provedení popsaných změn byly jednotlivé projekty frameworku převedeny do nové tzv. SDK podoby, byl vytvořen nový NuGet balíček `Squirrel.Core` sestavený pro .NET Framework 4.7.1 i .NET Standard 2.0, proto je na obrázku 5.1 `Squirrel.Core` dvakrát. NuGet balíčku `Squirrel` byla přidaná závislost na vytvořeném `Squirrel.Core` celý projekt byl sestaven a ve formě NuGet balíčků publikován na interní NuGet repozitář firmy Y Soft.

5.1.2 Framework Ruda

Bylo implementováno jednotné jádro frameworku *Ruda*, které prostřednictvím třídy `Ruda` a knihovny *Squirrel.Core* umožní provádět všechny potřebné operace s nainstalovanou aplikací. Dále byla implementována dvojice uživatelských rozhraní: grafické a rozhraní příkazové řádky s ekvivalentní množinou možných operací. Pro urychlení prvotní integrace do aplikace byla vytvořena knihovna `Ruda.UpdateShortcut` umožňující využít vytvořené grafické uživatelské rozhraní bez nutnosti implementace knihovny `Ruda` do koncové aplikace, v případě že se bude *Ruda* využívat pomocí veřejného API byla vytvořena knihovna `Ruda.CastleWindsor` usnadňující registraci všech potřebných komponent pomocí rozhraní `IWindsorContainer`.

Framework *Ruda* obsahuje tzv. build projekt[1], který po spuštění sestaví ostatní projekty, následně vytvoří NuGet balíčky dle předem stanovených definic a nahraje je do privátního NuGet repozitáře firmy Y Soft. Proces sestavení je plně automatický a probíhá po každé změně zdrojových souborů.

Knihovna Ruda

Podle návrhu v sekci 4.1.2 byla vytvořena základní část frameworku implementující všechny potřebné metody pro správu verze aplikace, mezi něž patří:

- Zjištění dostupnosti nové verze v aktuálním kanálu,

[1] projekt, jehož jediným cílem je sestavení jiných projektů viz sekce 3.6 a nástroj *Nuke*

- zjištění všech dostupných verzí v rámci aktuálního (příp. specifikovaného) kanálu,
- aktualizace aplikace na zvolenou verzi,
- aktualizace aplikace na poslední dostupnou verzi v aktuálním kanálu,
- zjištění aktuálně nainstalované verze a odvození názvu kanálu,
- převedení metadat sémantické verze na kolekci dvojic klíč-hodnota,
- zjištění dostupných kanálů.

Všechny tyto metody jsou definované rozhraním `IUpdater`, které implementuje třída `Ruda` a `NullUpdater`. Třída `Ruda` obaluje třídu `UpdateManager` z frameworku *Squirrel.Windows*, jež má na starosti, společně s aplikací `Update.exe`, samotný proces aktualizace aplikace tj. extrahování staženého instalačního balíčku, přesun extrahovaného obsahu do odpovídající služky a případné ukončení a opětovné spuštění aplikace (detaily viz sekce 3.4.1). Druhá implementace zmíněného rozhraní třídou `NullUpdater` je prázdnou implementací a je určená pro lokální sestavení, kde by třída `Ruda` hlásila z důvodu neočekávané adresářové struktury množství chyb.

V rámci frameworku *Squirrel.Windows* je pro stahování instalačních souborů používané rozhraní `IFileDownloader`, které je kompletně izolováno od pojetí aktualizačních kanálů a proto bylo implementováno nové rozhraní `IChanneledDownloader`, mj. implementující `IFileDownloader`, jež kanály plně podporuje, nabízí metody pro získání kolekce dostupných kanálů na vzdáleném serveru, ověření existence specifikovaného kanálu a jeho změnu. Rozhraní `IChanneledDownloader` bylo implementováno třídou `ArtifactoryDownloader`, která, jak již název napovídá, slouží k získávání informací ze serveru *JFrog Artifactory*, třída pro vytváření a provádění HTTP požadavků využívá knihovnu `Flurl.Http` umožňujíc definici požadavku tzv. fluentně (řetězením rozšiřujících metod). Implementace je specifická pro *Artifactory* z důvodu rozdílného získávání kolekce aktuálně dostupných kanálů u každého typu vzdáleného serveru, v případě *JFrog Artifactory* bylo využito nativního prohlížeče struktury (*Artifactory* vrátí webovou stránku s výpisem souborů a složek v dotazovaném adresáři) a pomocí XPath[2] dotazu jsou zjištěny dostupné adresáře resp. kanály. Během testování byla implementována i třída `NexusDownloader` umožňující práci s úložištěm typu *Sonytype Nexus*, která z důvodu přechodu na *JFrog Artifactory* již není používaná, ale implementace je stále platná a plně funkční.

Knihovna Ruda.CLI

Další implementovanou částí frameworku byla knihovna `Ruda.CLI`, která umožňuje provádět všechny metody pro správu verze aplikace vyjmenované v předchozí sekci z prostředí příkazového řádku. Pro parsování argumentů předaných příkazovou řádkou byla použita knihovna `CommandLineParser` umožnující definici možných argumentů pomocí tříd s veřejnými vlastnostmi (angl. public property), které se mapují na předané argumenty tzn. jedna vlastnost uchová informace o jednom předaném argumentu. Je použité schéma ve kterém každá třída, specifikuje jeden příkaz tzv. *verb* (např. *update*, *info*), který může mít další parametry modifikující jeho průběh. Pro správnou definici takového chování musí mít celá třída (definující jeden příkaz) atribut `VerbAttribute` s volitelným argumentem `HelpText` a každá vlastnost v dané třídě, na kterou se mapují další parametry musí mít atribut `Option`

[2] Jazyk pro dotazování nad XML souborem

s volitelnými parametry specifikující krátký a dlouhý název, zda jde o povinný argument, atd.. Poté, co se předané argumenty pomocí knihovny `CommandLineParser` analyzují a naplní se jimi odpovídající třída popisující daný příkaz je instance třídy předaná metodě zajišťující, s pomocí knihovny `Ruda`, jeho provedení.

Implementované řešení si vystačí jen se dvěma příkazy

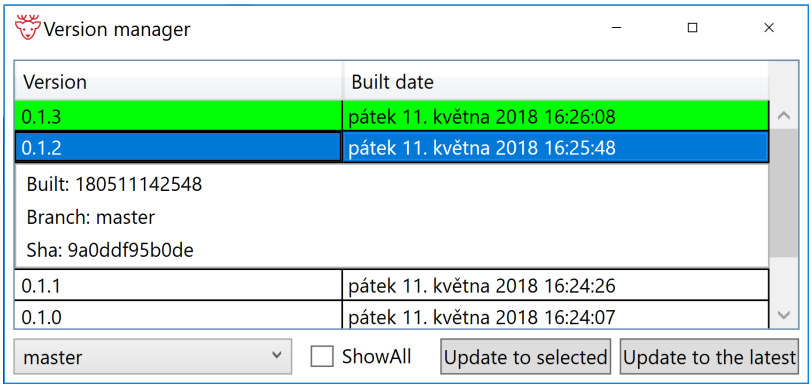
- `info` – poskytující informace o aktuálně nainstalované verzi a její aktuálnosti včetně kanálu, dále poskytuje informace o dostupných verzích a kanálech.
- `update` – aktualizace aplikace na poslední verzi v aktuální kanálu, případně na specifikovanou verzi v kterémkoliv kanálu

Do řešení byla také implementována možnost předat pomocí rozhraní `ICliExtension` k parsování další třídy specifikující jiné příkazy a jejich argumenty, čehož bylo využito u knihovny `Ruda.GUI` pro spuštění grafického uživatelského rozhraní z prostředí příkazového řádku.

Knihovna Ruda.GUI

Implementace grafického uživatelského rozhraní je realizovaná s pomocí grafického subsystému *WPF* v rámci knihovny `Ruda.GUI`. Základem pro vytvoření uživatelského rozhraní byl návrh v sekci 4.1.2.

Uživatelského rozhraní bylo nejdříve navržen v grafické knihovně *Windows Forms*, pro kterou existuje jednoduchý editor a vzhledem k předchozím zkušenostem byla implementace velmi rychlá. Daní za relativně rychlou implementaci jsou nevýhody knihovny *Windows Forms* zmíněné v sekci 3.1. Knihovna *Windows Forms* již je považována za přežitou a je nahrazena grafickým subsystémem *Windows Presentation Foundation* zkráceně *WPF*. Z těchto důvodu byla produkční implementace vytvořena ve *WPF*, který netrpí nedostatky *Windows Forms*. Při přechodu z *Windows Forms* na *WPF* bylo možné majoritní část kódu zajišťujícího funkčnost převzít z první implementace a znovu použít v druhé, produkční implementaci. Finální podoba grafického uživatelského rozhraní pro aplikaci `DemoApp` je na obrázku 5.2.



Obrázek 5.2: Finální podoba grafického uživatelského rozhraní frameworku *Ruda*. Zelený řádek je aktuální verze, modrý vybraná veze, pro kterou jsou zobrazené detaily.

Při implementaci se přišlo na skutečnost, že knihovna `Squirrel.Core` používá typ `Icon` z jmenného prostoru `System.Drawing`, jež není v platformě .NET Standard standardně implementován a proto byla jako náhrada použita knihovna `CoreCompat.System.Drawing`

která jej do .NET Standardu doplňuje. Typ `Icon` z této knihovny ale nemůže být použít zároveň s typem `Icon` z platformy .NET Framework. Proto bylo nutné knihovnu `Squirrel.Core` sestavit pro více platforem (tzv. multitarget) a to takovým způsobem aby se pro .NET Standard používal typ `Icon` z knihovny `CoreCompat.System.Drawing` a pro .NET Framework vestavěná implementace v platformě.

Aplikace Ruda.UpdateShortcut a knihovna Ruda.CastleWindsor

Knihovna `Ruda.CastleWindsor` se skládá jen z jedné statické třídy obsahující rozšiřující metodu pro typ `IWindsorContainer` s názvem `UseRuda`, která na základě konfigurace registruje vybranou implementaci rozhraní `IUpdater`. V případě, že mý být dle konfigurace framework *Ruda* neaktivní je registrovaná implementace `NullUpdater`, která, jak již bylo zmíněno dříve nedělá nic. Pokud je framework *Ruda* povolený a je předána implementace `IChanneledDownloader` nebo konfigurace výchozí implementace zmíněného rozhraní a aplikace je korektně nasazená (mj. odpovídá adresářová struktura) je pro implementaci rozhraní `IUpdater` registrovaná třída `Ruda`.

Implementace samostatné aplikace `Ruda.UpdateShortcut` zobrazující grafické uživatelské rozhraním pro správu verzí jiné aplikace, byla snadná. Pro její realizaci bylo využito knihovny `Ruda.GUI` a `Ruda.CastleWindsor`. Po startu se vytvoří DI (z anglického *Dependency Injection*) kontejner, ve kterém se registruje logování a pomocí `Ruda.CastleWindsor` i framework *Ruda*, po registraci se využije třída `GuiCliExtension` z knihovny `Ruda.GUI` implementující funkcionalitu zobrazení GUI pomocí příkazové řádky, předají se jí parametry indikující požadavek na zobrazení GUI a zavolá se vykonání tohoto požadavku. Takový postup vyústí v přesunutí zodpovědnosti za správné zobrazení GUI z knihovny `Ruda.UpdateShortcut` na `Ruda.GUI`. Při kompilaci se pomocí knihovny `Fody` a `Costura.Fody` sloučí všechny potřebné závislosti se spustitelným souborem a vznikne jediný spustitelný soubor obsahující všechny potřebné knihovny. Sloučení lze přirovnat ke statickému linkování při vývoji v C/C++ avšak to pouze na první pohled. Interně knihovna `Costura` potřebné dynamické knihovny ke spustitelné aplikaci přiloží jako vestavěné zdroje (*embedded resources*) a zařídí, aby při spuštění aplikace byly potřebné knihovny načtené do paměti. [5]

5.1.3 Implementace knihovny NukeHelpers.Ruda a výsledný proces sestavení aplikací

Jak již bylo zmíněno v kapitole 4.1.3 zabývající se návrhem nového procesu sestavení je použitý nástroj pro automatizaci sestavení *Nuke*, jež umožňuje definování cílů pomocí kódu v jazyce C#. Při implementaci knihovny `NukeHelpers.Ruda` bylo nutné rozhodnout, zda implementovat cíl, který by byl zodpovědný za provedení celého procesu nutného pro vytvoření a zveřejnění nové verze, nebo zda implementovat metody, které by byly použitelné v cílech definovaných v rámci každého projektu a společně by umožňovaly vytvoření a zveřejnění nové verze. Po zvážení výhod a nevýhod každého řešení bylo rozhodnuto implementovat jednotlivé metody, ze kterých může být v budoucnu případně sestaven i cíl umožnující vykonat celý proces samostatně. Byly implementované metody pro:

- Vytvoření NuGet balíčku ze sestavených objektů na základě *nuspec* konfigurace.
- Stažení staršího vydání specifického kanálu.
- Vytvoření aktualizačního balíčku a instalátoru, případně i *MSI* instalačního souboru.

- Upload vytvořených souborů na vzdálený server.
- Úklid na vzdáleném serveru.

Každá metoda přebírá parametr specifikující konfiguraci frameworku *Ruda* a prostřední, na kterém k sestavení dochází, taková konfigurace zahrnuje mj. umístění zdrojových souborů, výstupní složku pro balíčky, název kanálu.

Před vytvářením NuGet balíčku jsou ve složce se zdrojovými kódy hledány soubory s koncovkou `.appspec`, které se strukturou neliší od souborů *nuspec*, pro každý soubor je podle jeho názvu vyhledán odpovídající projekt, pokud je nalezen vytvoří se kopie souboru *appspec* s koncovkou `.nuspec` (protože NuGet soubory *appspec* odmítá), který je použitý pro následné balení. K vytvoření balíčku se využívá *NuGet* klient integrovaný v kompilátoru zdrojových souborů a použitelný díky třídě `NuGetTasks` z knihovny `Nuke`. Balíček se vytvoří pomocí metody `NuGetPack` s parametry specifikujícími cestu k *nuspec* (kopie *appspec*) souboru, verzi balíčku a výstupní složku. Metoda může vytvářet více NuGet balíčků během jednoho běhu, záleží na tom, kolik je v adresáři se zdrojovými kódy souborů *appspec* a odpovídajících projektů.

Stažení instalačního balíčku předchozího vydání není možné provést na jeden webový požadavek, protože není známá verze předchozího vydání a tudíž ani název souboru, který je potřeba stáhnout. Z toho důvodu je nejdříve snaha stáhnout soubor `RELEASES` (který se za každých okolností jmenuje stejně a větev jej musí obsahovat), pokud se stažení nepodaří větev na vzdáleném serveru ještě neexistuje, případně není platná, tím metoda končí. V případě, že se stažení podařilo je ze souboru `RELEASES` zjištěna předchozí verze a tudíž i název posledního instalačního souboru, díky tomu je možné jeho stažení do předem daného umístění.

Pro vytvoření instalačního balíčku a instalátoru je nutné mít vytvořený NuGet balíček a pokud se nejedná o první vydání v novém kanálu je vhodné mít stažený předchozí instalační balíček a soubor `RELEASES`. Díky tomu bude možné vytvořit rozdílový balíček, novou verzi přidat do souboru `RELEASES` a tím ji zpřístupnit pro uživatele. Vlastní vytvoření instalačních balíčku spočívá ve spuštění aplikace `Update.exe` s argumentem `--releasify` a cestou k NuGet balíčku. Aplikace na základě NuGet balíčku sestaví plný, případně i rozdílový instalační balíček a instalátor (`Setup.exe`) případně i `Setup.msi`, vše uloží do složky `releases`.

Poslední metodou, kterou je nutné provést pro umožnění rozšíření nové verze mezi uživatele je nahrání vytvořených instalačních balíčků a instalátorů na vzdálený server. Jak již bylo zmíněno v sekci 4.1.3 ačkoliv jsou instalační balíčky opět ve formě NuGet balíčků, tak instalátory a soubor `RELEASES` NuGet balíček není a není jednoduše možné jej nahrát do NuGet repozitáře pomocí nástrojů k tomu určených. Samotný proces nahrávání dat na vzdálený server začíná nalezením souboru `RELASES`, pokud není nalezený, proces končí. Pokud je nalezený, tak se společně s instalačním balíčkem, případným rozdílovým balíčkem a instalátorem nahrají na server *Artifactory*. Nahrávání probíhá prostřednictvím HTTP PUT kde se soubor odešle na server s hlavičkou *ContentType=application/octet-stream*. Celý HTTP požadavek je vytvořen a následně proveden s pomocí knihovny `Flurl.Http`. Rozložení souborů na vzdáleném serveru odpovídá obrázku 4.5 v sekci zabývající se návrhem.

Poslední avšak neméně důležitou funkcionalitou knihovny `NukeHelpers.Ruda` je uklízení na vzdáleném serveru tzn. smazání celých kanálů pro které již neexistuje odpovídající větev v Gitu a také starších instalačních a rozdílových balíčků. Mazání samotných instalátorů není nutné protože při nahrávání nového je původní přepsán. Před mazáním je nutné

zjistit, jaké kanály jsou dostupné na vzdáleném serveru a jaké větve jsou v Git repozitáři. Pro zjištění dostupných kanálů se používá metoda `GetAvailableChannelsAsync` z třídy `ArtifactoryDownloader` popsaná v sekci 5.1.2. Pro zjištění větví dostupných v Git repozitáři je použita knihovna `LibGit2Sharp`. Pro správnou funkci knihovny je nutné dodat konstruktoru třídy `Repository` cestu ke složce `.git`, následně je možné získat kolekci objektů popisující Git větev. Objekt má vlastnost `IsRemote`, jejíž návratová hodnota záleží na tom, zda plný název větve obsahuje prefix `refs/remotes/`. Tuto vlastnost je možné použít v případě větví v tzv. *tracking branch*[3] režimu, což je výchozí možnost při klonování větve avšak *Atlassian Bamboo*, jež se ve firmě Y Soft používá k sestavování softwaru větve neklonuje jako *tracking branch*, což činí vlastnost `IsRemote` nepoužitelnou. Proto v případě nenalezení žádné *tracking branch* (typický stav na *Bamboo*) je předpokládáno, že lokální a vzdálené větve se neliší. Při testování, ani při ostrém provozu nebylo zjištěno, že by byl předpoklad porušen. Následně je kolekce vzdálených větví pomocí knihovny `Ruda` převedena na kolekci názvů kanálů a kanály, které se nacházejí jen na vzdáleném serveru (*Artifactory*) jsou pomocí HTTP DELETE smazány. Poté se pro každý kanál stáhne soubor `RELEASES`, zjistí se počet vydaných sestavení a případě, že překračuje definované maximum jsou ze souboru `RELEASES` odstraněné záznamy o instalačních balíčcích (plných i rozdílových), balíčky jsou pomocí HTTP DELETE smazané ze vzdáleného serveru a soubor `RELEASES` je nahrán zpět na server.

5.2 Implementace frameworku RoH

Framework *RoH* je stejně jako *Ruda* implementován v programovacím jazyce C#. Při implementaci bylo vycházeno z návrhu popsaném v kapitole 4.2 tzn. framework byl rozdělen do tří knihoven dle jejich určení.

5.2.1 Knihovna RoH.Core

Nejdůležitější roli v celém frameworku zastává knihovna `RoH.Core`, jejíž hlavní zodpovědností je generování a parsování identifikátorů služeb a aplikací, další zodpovědností je instanciace třídy `ConsulClient`, která je používaná v celém frameworku pro komunikaci s nástrojem *Consul*, na základě konfigurace předané třídou `ConsulClientOptions`. Jednotlivé identifikátory je možné vytvořit přímo konstruktory tříd `RohAppId` pro identifikaci aplikace, `RohServiceId` pro identifikaci služby a `RohServiceInstanceId` pro identifikaci její instance. Třídy od sebe postupně dědí v pořadí od nejméně specifické po tu nejvíce specifickou. Třída `RohAppId` má řetězcovou vlastnost `AppId` obsahující název aplikace, virtuální vlastnost `ConsulId` obsahující řetězcový identifikátor aplikace používaný v nástroji *Consul*, v potomcích této třídy je vlastnost přepsána a obsahuje identifikátor služby resp. její instance, potomci také vždy přidají vlastnost s název přidané části identifikátoru (název služby, id hosta/instance). Další možností, jak vytvořit identifikátor je parsování již existujícího, tato funkcionalita je realizována metodou `Parse` v třídě `RohIdParser` zvládající parsovat všechny druhy identifikátorů a navrací vždy ten nejvíce specifický. Pro potřeby parsování specifického typu identifikátoru byla vytvořena generická metoda `Parse<T>` s návratovým typem `T` a omezením generického parametru na typ `RohAppId` a jeho potomky.

Poslední funkcionalitou této knihovny je generování názvů značek. V nástroji *Consul* mají značky tvar obecného řetězce, avšak pro použití ve frameworku *RoH*, a hlavně pro

[3] Větev „sleduje“ vývoj na vzdáleném serveru tzn. v případě změn na vzdáleném serveru je možné jejich stáhnutí a aplikace pomocí příkazu `git pull` [3]

zobrazení ve webovém rozhraní, by bylo vhodné, aby značky mohly být rozdělené do dvou skupin – viditelné ve webovém rozhraní tzv. veřejné značky a neviditelné ve webovém rozhraní tzv. privátní značky. Funkcionality bylo dosaženo třídou `RohTag`, jež má vlastnost `Value`, obsahující hodnotu značky, `Visibility` obsahující viditelnost (veřejná nebo privátní) a `RohValue` obsahující řetězec používaný v nástroji *Consul*. Majoritní část značek bude s privátní viditelností a proto je viditelnost implementovaná tak, že vlastnost `RohValue` je pro privátní značky stejná jako vlastnost `Value`, pro veřejné značky je před `Value` přidán prefix `-Pub`.

5.2.2 Knihovna RoH.Service

Ačkoliv název knihovny napovídá, že je určená pouze pro monitorování služeb, její určení je i pro monitoring samostatných aplikací. Knihovna zajišťuje registraci aplikací a služeb do nástroje *Consul*, který je následně monitoruje a stav zpřístupňuje dalším komponentám. Vlastní registrace se provádí pomocí knihovny `consuldotnet` a REST API nástroje *Consul*. Pro kontrolu stavu aplikace byla implementovaná třída `RohApplication` mj. s metodami `RegisterApp` a `DeregisterApp` umožňující provést registraci resp. odregistraci aplikace z nástroje *Consul*. Jak již bylo zmíněno v návrhu, v sekci 4.2.2, pro kontrolu aplikací je nutné implementovat TCP server, ke kterému budou probíhat pokusy o přijetí spojení z *Consulu*. TCP server byl implementován třídou `TcpServer`, která je postavená na třídě `TcpListener` a jediným úkolem je spojení přijmout a bez další interakce uzavřít, část kódu zajišťující tuto funkcionalitu je na výpisu 5.1, TCP server umožňuje při spuštění vybrat port, na kterém bude naslouchat, v případě předání čísla portu „0“ zahájí naslouchání na portu, který uzná za vhodný, jeho číslo je následně předáno do vlastnosti `Port` implementované třídy. TCP server je používán třídou `RohApplication` zajišťující jeho spuštění před počátkem kontroly stavu (registrací pomocí `RegisterApp`) a ukončení po odregistraci z kontroly stavu. V případě, že aplikace není odregistrovaná sama, ale je jen vypnuta (tzn. kontrola stavu hlásí nedostupnost) tak bude z *Consulu* po definované době záznam o službě odstraněn. Třída `RohApplication` v konstruktoru přebírá konfiguraci pomocí `RohApplicationOptions` konfigurace obsahuje mj. nastavení *Consul* klienta, název registrované aplikace, interval kontroly stavu, možnost určující, zda při kontrole stavu využít doménové jméno nebo IP adresu a statické značky, které jsou do registrace vždy přidané.

```
1 while (!_token.IsCancellationRequested) {
2         var tcpClient = await _listener
3                 .AcceptTcpClientAsync()
4                 .ConfigureAwait(false);
5         tcpClient.Close();
6 }
```

Výpis 5.1: Část kódu z TCP serveru příjmající a uzavírající TCP spojení.

Pro kontrolu stavu služeb byla implementována třída `RohAppService`, aby byla služba monitorovatelná pomocí této knihovny musí obsahovat TCP, HTTP nebo gRPC server, díky kterému bude možné její stav kontrolovat. Princip použití třídy je podobný jako v případě `RohApplication`, tzn. při konstrukci třídy se předá veškerá konfigurace, následně při spuštění služby se zavolá metoda `EnableCheck` a při nutnosti ukončení kontroly `DisableCheck`. Třída pro svoji funkci vyžaduje referenci na rozhraní `IRohApplication`, pomocí kterého probíhá komunikace s nástrojem *Consul*. Dále je nutné nakonfigurovat jakým způsobem,

případně více způsoby, se bude stav služby kontrolovat, v případě použití TCP kontroly stačí jen port, v případě HTTP je nutné specifikovat i URL.

V aplikacích, kde se bude kontrola stavu služeb využívat se používá DI což znamená, že v případě, kdy bude součástí aplikace více služeb bude v DI kontejneru registrovaných více instancí `RohAppService` a získání reference na tu správnou by mohlo být problémové[4]. Proto byla implementovaná i generická obdoba `RohAppService<T>`, umožnující využití generického typu pro rozlišení různých služeb. Např. služba `HttpSrvr` bude kontrolovaná pomocí `RohAppService<HttpSrvr>` a jiná služba `GrpcSrvr` bude kontrolovaná pomocí `RohAppService<GrpcSrvr>` což umožní v konstruktoru vyžádání správné instance `RohAppService<T>`. Generická obdoba třídy dědí od negenerické a nepřidává žádnou další vlastnost ani funkci.

Host, na kterém služba běží může mít více síťových rozhraní s různými adresami a je nutné vybrat tu správnou adresu, na kterou bude možné připojení z rozhraní nástroje *Consul*. Z toho důvodu vyžaduje `RohApplication` v konstruktoru implementaci rozhraní `IIpAddressProvider` jehož jediným úkolem je získání té správné IP adresy. V knihovně bylo rozhraní implementováno třídou `RohIpAddressProvider`, která z IP adres přiřazených hostovi vyberu tu, jež se spojí s *Consulem*, pro výběr se využívá třídy `Socket` u níž je možné po připojení na vzdáleného hosta zjistit zdrojovou adresu spojení (jedna z adres přiřazených hostovi). Při výběru IP adres se preferuje IPv6 před IPv4.

V předchozí části textu bylo zmíněno, že kontrola stavu podporuje předávání značek do nástroje *Consul*. Aby bylo předávání značek co nejjednodušší bylo implementováno rozhraní `ITagProvider` obsahující jedinou vlastnost `Tags` typu `ObservableCollection<RohTag>`. S typem `ObservableCollection` je možné zacházet jako s běžnou kolekcí (`ICollection`), avšak oproti kolekci nabízí výhodu v tom, že obsahuje událost `CollectionChanged` vyvolanou při jakékoliv modifikaci kolekce. Třída `RohAppService` této události využívá pro okamžitou aktualizaci značek v *Consulu* a díky tomu *Consul* vždy reflektuje aktuální stav značek v aplikaci.

5.2.3 Knihovna RoH.Client

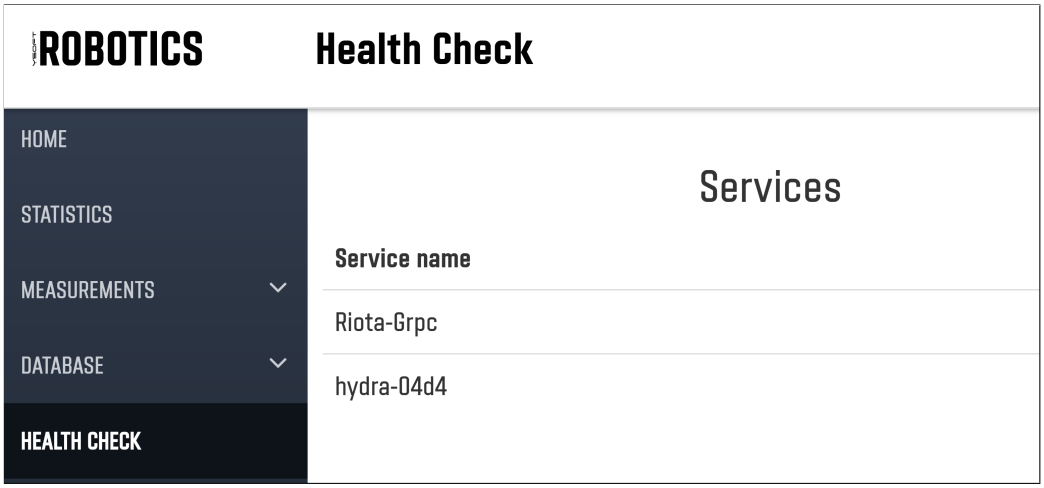
Poslední avšak neméně důležitou součástí frameworku je knihovna `RoH.Client` její hlavní účel je získávání informací z *Consulu* o dostupných instancích aplikací a služeb a jejich následně předání dalším komponentám. Pro komunikaci s *Consulem*, stejně jako ostatní části frameworku využívá knihovnu `consuldotnet`. Hlavní funkcionalita je realizovaná třídou `RohClient`. Třída je konfigurovatelná pomocí typu `ConsulClientOptions` obsahujícího údaje nutné pro spojení s *Consulem*. Pro zjišťování jednotlivých dostupných aplikací, služeb a jejich instancí obsahuje vlastnost `Services` typu `RohServiceEndpoint` nabízející možnosti dotazování a filtrování dat. Typ `RohServiceEndpoint` obsahuje metodu `All` určenou k získání kolekce dostupných služeb a aplikací (nikoliv instancí) jejíž princip je následující: získají se všechny dostupné aplikace a služby z nástroje *Consul*, u každé se vyhodnotí, zda má platný identifikátor (tzn. identifikátor je možné parsovat jako `RohServiceId`, viz 5.2.1). V případě, že identifikátor není platný je služba přeskočena, v opačném případě je na základě dat z *Consulu* vytvořena nová instance třídy `RohService` obsahující zpracovaný identifikátor a všechny dostupné značky. Po zpracování všech služeb je navrácena kolekce všech dostupných aplikací a služeb. Pro získání informací o jednotlivých instancích bylo implementováno několik přetížení metody `Service`, což v případě toho nejrozvinutějšího

[4] Hlavně v případě použití knihovny `Microsoft.Extensions.DependencyInjection`, která, na rozdíl např. od `Castle.Windsor`, nenabízí pojmenované komponenty.

znamená, že je možné specifikovat dle `RohServiceId`, stavu instance (celkový stav kontrol je binární – žije, nežije) a požadovaných tagů. Implementace těchto metod stačí k získání přehledu o dostupných službách i jejich jednotlivých instancích.

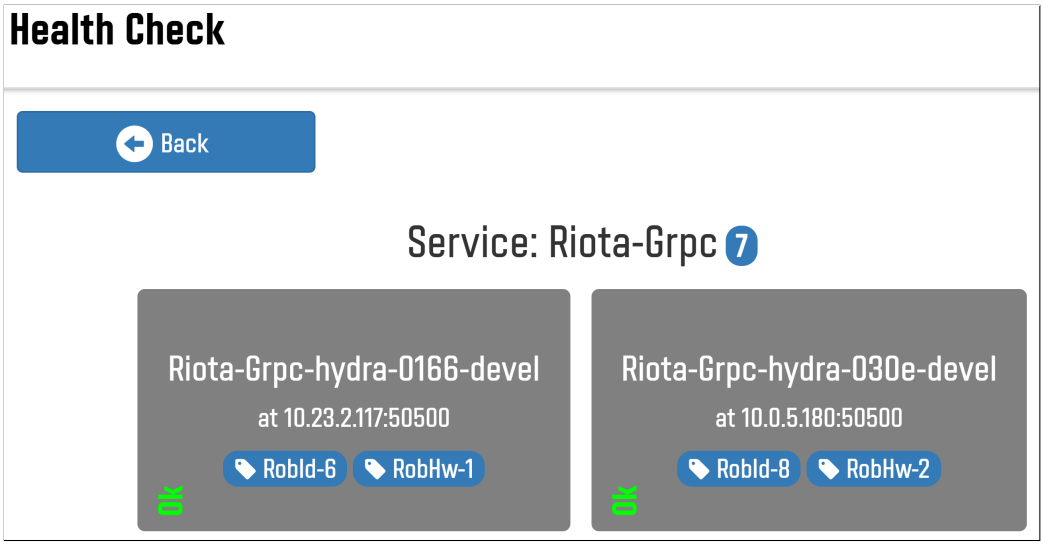
5.2.4 Implementace přehledu služeb do webového rozhraní

Jak již bylo zmíněno v sekci 4.2.4 o návrhu integrace je webové rozhraní vytvořeno pomocí frameworku `DotVVM` a pro implementaci *view* bylo využito již používané knihovny *Bootstrap*. Zobrazení přehledu dostupných aplikací a služeb zajišťuje *view* `HealthCheck.dothtml`, který v tabulce pomocí tagu `repeater` zobrazuje identifikátory dostupných aplikací a služeb. *View* získává data z implementovaného *view modelu* `HealthCheckServicesViewModel` obsahující kolekci dostupných služeb získaných z knihovny `RoH.Client` pomocí metody `RohServiceEndpoint.All`.



Obrázek 5.3: Přehled dostupných služeb ve webovém rozhraní (výřez).

Po kliknutí na identifikátor služby *view model* v metodě `LoadDetail` získá informace o všech instancích vybrané služby a pomocí instance třídy `RohServiceDetailModel` je předá *view* `RohServiceDetail.dotcontrol`. *View* tentokrát není dothtml (*view* celé stránky), ale dotcontrol (*view* jen části stránky), což umožňuje detail zobrazit na stejné stránce jako přehled, čímž se ušetří opětovné načítání celé stránky, jelikož jsou k uživateli do prohlížeče přenesena jen data nutná k zobrazení instancí, soubory HTML a JS zůstávají stejné (tento koncept byl již ve webovém rozhraní implementován a práce jej pouze využívá). Jak je patrné z obrázku 5.4 v detailu jsou jednotlivé instance zobrazeny jako tzv. badge, kde nejvíce prostoru zabírá identifikátor instance (např. *Riota-Grpc-hydra-0162-devel*), dále je zobrazená adresa a port služby, u levého okraje stav a při spodním okraji případné veřejné značky (*RobId-6*, *RobHw-1*).



Obrázek 5.4: Detail služby ve webovém rozhraní (výřez).

5.3 Příklad implementace frameworku Ruda v aplikaci

Framework může být využitý v GUI i v konzolových aplikacích napsaných pro .NET Framework. Vzhledem ke skutečnosti, že aplikace aktuálně využívající framework *Ruda* jsou používané pouze interně v rámci firmy Y Soft, nejsou veřejné a tudíž není možné prezentovat implementaci a funkčnost na nich byla vytvořena jednoduchá demonstrační aplikace `DemoApp`. Aplikace bude použita pro demonstraci implementace a předvedení funkcionality pouze v rámci této bakalářské práce.

`DemoApp` je aplikace s grafickým uživatelským rozhraním realizovaným pomocí *WPF* zobrazující jedinou, avšak nejdůležitější informaci, kterou je aktuální verze aplikace. Pro použití frameworku je nutné přidat potřebné NuGet balíčky, v případě `DemoApp` jsou to `YSoft.Ruda.GUI` a `YSoft.Rqa.Ruda.UpdateShortcut`, balíčky je možné přidat pomocí souboru `packages.config` nebo, pomocí novějšího přístupu, přímo do projektu s využitím XML značky `PackageReference` jak ukazuje výpis 5.2.

```
<ItemGroup>
        <PackageReference Include="YSoft.Rqa.Ruda.UpdateShortcut">
                <Version>2.1.2</Version>
        </PackageReference>
        <PackageReference Include="YSoft.Ruda.GUI">
                <Version>2.1.2</Version>
        </PackageReference>
</ItemGroup>
```

Výpis 5.2: Přidání frameworku *Ruda* do projektu.

Po přidání balíčků NuGet automaticky stáhne a přidá všechny potřebné závislosti (`Core`, `Cli` a `Squirrel.Windows`). Díky tomu je možné ve zdrojových souborech využít knihovnu `Ruda` a instanciovat ji například dle výpisu 5.3. Třída umožňuje kompletní správu verze aplikace mj. instalaci jiné verze, získávání informací o dostupných verzích a kanálech a získání informace o aktuálně nainstalované verzi pomocí metody `CurrentVersion`. Toho je v `DemoApp` využito pro zobrazení verze. Další součástí frameworku je třída `SelectPackage`

umožňující zobrazení GUI, ve kterém si uživatel může zvolit jinou verzi a následně na ni aktualizovat viz obrázek 5.2.

```
39 _ruda = new Ruda(
40         new ArtifactoryDownloader(ArtifactoryUrl, ArtifactoryPath),
41         new ConsoleLogger(nameof(Ruda))
42        );
```

Výpis 5.3: Příklad instanciace třídy `Ruda`.

Knihovna `YSoft.Rqa.Ruda.UpdateShortcut` umožňuje změnit verzi aplikace implementující framework bez jejího spuštění pomocí vlastního GUI. Aby aplikace umožnující tuto změnu měla rozumný název, je nutné přidat do akcí vyvolaných po úspěšném sestavení kód podobný výpisu 5.4. Kód zajistí přejmenování aplikace a zkopírování konfiguračního souboru, který obsahuje konfiguraci frameworku.

```
cd "$(TargetDir)"
move /Y Ruda.UpdateShortcut.exe "Update $(SolutionName).exe"
copy /Y "$(TargetFileName).config" "Update $(SolutionName).exe.config"
```

Výpis 5.4: Změna názvu pomocné aplikace pro aktualizaci a kopírování konfigurace.

Poslední důležitou součástí je modifikace sestavení. Jak již bylo dříve zmíněno, pro sestavení projektů se využívá nástroj `Nuke.Build` a vytvořená knihovna zjednodušující proces vydání, která obsahuje třídu `RudaTasks` s metodami umožňující sestavení vydat. Výpis 5.5 ukazuje správné pořadí volání metod, celý soubor definující proces sestavení je dostupný v projektu `.build` pod názvem `Build.cs`.

```
113 RudaTasks.CreateNugetPackage(() => conf);
114 await RudaTasks.DownloadOldPackages(() => conf);
115 RudaTasks.CreateRudaPackage(() => conf);
116 await RudaTasks.DeployRudaPackages(() => conf);
117 await RudaTasks.RudaCleanupRemoteServer(() => conf);
```

Výpis 5.5: Postup volání metod pro vydání nového sestavení.

Po vykonání všech metod jsou na vzdáleném serveru dostupné potřebné instalační balíčky, soubor `RELEASES` a instalátor `Setup.exe`. Spuštěním instalátoru dojde k instalaci obou aplikací – vlastní `DemoApp` a `Update␣DemoApp` z balíčku `UpdateShortcut`. Po instalace je automaticky spuštěna `DemoApp` a je zobrazena aktuální verze aplikace a tlačítko pro otevření správce verzí umožnující instalaci jiné verze i z jiného kanálu. Aplikace `Update␣DemoApp` umožňuje změnit verzi `DemoApp` i v případě, že aplikace není schopna spuštění.

5.4 Shrnutí

Kapitola 5 v první části popisuje postup implementace frameworku *Ruda* určeného pro správu verzí aplikací. Rozdělení frameworku na jednotlivé knihovny, jejich význam, funkci a způsob distribuce. Také přibližuje změny, které bylo nutné provést ve využívaném frameworku *Squirrel.Windows*. Ve druhé části kapitoly je popsán postup při vývoji frameworku *RoH* určeného k monitorování aplikací a jejich služeb. Rozdělení frameworku na část zajišťující registraci a část umožnující získání informací z nástroje *Consul*. Poslední část názorně ukázala, jak postupovat při implementaci frameworku *Ruda* do vlastní aplikace, včetně správného postupu pro vydání a zpřístupnění sestavení uživatelům.

Kapitola 6

Závěr

Práce je rozdělena na dvě části, první se zabývá návrhem a vytvořením frameworku pro správu verzí interních aplikací firmy Y Soft. Framework pro svoji funkci využívá softwarové vybavení, které je ve Y Softu již nasazené a prověřeně. Framework vývojářům umožňuje jednoduchou integraci do vyvíjené aplikace, snadnou modifikaci procesu sestavení pro automatické vydávání nových verzí a nabízí podporu více kanálů vytvářených na základě větví ve verzovacím nástroji Git. Uživatelům umožňuje prostřednictvím grafického uživatelského rozhraní nebo rozhraní příkazové řádky snadnou instalaci aplikací, jejich následnou aktualizaci na vybranou verzi i změnu kanálu. Druhá část práce se zabývá návrhem a implementací frameworku pro kontrolu a zjišťování stavu aplikací a jejich služeb pomocí nástroje *Consul*. A využitím frameworku pro implementaci přehledu monitorovaných aplikací a služeb do již existujícího webového rozhraní používaného v rámci firmy Y Soft.

V rámci první části práce byl proveden návrh frameworku pro správu verzí aplikací, rozdělení na jednotlivé komponenty a jejich následná implementace. Framework je rozdělen na jádro, uživatelské rozhraní příkazové řádky, grafické uživatelské rozhraní a pomocnou aplikaci. Pomocná aplikace umožňuje aktualizovat vlastní aplikaci s využitím grafického uživatelského rozhraní, bez nutnosti zásahu do zdrojového kódu vlastní aplikace. Zmíněné rozdělení umožňuje využít jen část frameworku v závislosti na potřebách vývojáře a možnostech cílového prostředí pro běh aplikace. Dále je implementována komponenta umožňující snadnou modifikaci procesu sestavení tak, aby zahrnoval vytvoření aktualizačního balíčku i instalátoru, nahrání vytvořených balíčků do úložiště *JFrog Artifactory* a zveřejnění nové verze. Každá komponenta je implementovaná jako samostatná knihovna (resp. aplikace) a je dostupná v podobě *NuGet* balíčku v interním repozitáři firmy Y Soft.

Druhá část práce popisuje návrh frameworku pro kontrolu stavu, který je také rozdělen na několik částí – jádro, část určenou pro monitorované aplikace a klientskou část. Část určená pro monitorované aplikace zajišťuje komunikaci s nástrojem *Consul*. Nástroj během registrace získá od frameworku specifikaci aplikace i jejích služeb a kontrol, pomocí kterých má být stav zjišťován. Po registraci se do nástroje automaticky přenáší značky umožnující snazší identifikaci jednotlivých instancí monitorovaných aplikací a služeb. Nástroj *Consul* provádí kontroly a udržuje informace o stavu aplikací a služeb. Klientská část frameworku umožňuje získat informace o službách a aplikacích z nástroje *Consul* v přehledné podobě a dále s nimi pracovat. Klientská část je využita pro implementaci přehledu stavu monitorovaných aplikací a služeb do již existujícího webového rozhraní.

Obě implementovaná řešení byla v praxi ověřena ve firmě Y Soft. Na základě podnětů uživatelů bylo následně upraveno grafické uživatelské rozhraní. Bylo dosaženo větší přehlednosti a nejčastější úkon, kterým je aktualizace aplikace na poslední verzi v aktuálním

kanálu byl zjednodušen a zrychlen. V současné době jsou obě řešení produkčně nasazena a používána v rámci několika týmů vývojového oddělení firmy Y Soft. Řešení jsou již několik měsíců používána bez dalších výtek nebo připomínek.

V průběhu používání frameworku pro správu verzí aplikací již nevznikaly požadavky na další funkční vylepšení malého a středního rozsahu a na základě této skutečnosti je možné framework považovat za dokončené řešení. Oproti tomu druhá část práce, věnující se kontrole stavu aplikací a služeb má potenciál k dalšímu rozšíření, zvláště s nástupem trendu mikroslužeb. Do frameworku mohou být přidány funkce přenášení metadat (řetězce klíč-hodnota) nebo zajišťování vzájemné kompatibility mikroslužeb.

Seznam zkratek

API	Application Programming Interface – rozhraní pro programování aplikací
ARM	Advanced RISC Machine – procesorová architektura
CIL	Common Intermediate Language – nízkoúrovňový jazyk používaný platformou .NET
CLI	Command Line Interface – rozhraní příkazové řádky
CLR	Common Language Runtime – virtuální stroj vykonávající CIL
DI	Dependency Injection – injekce závislostí
GC	Garbage collection – uvolnění nevyužívané paměti
gRPC	Google RPC – systém umožnující vzdálené volání procedur
GUI	Graphical User Interface – grafické uživatelské rozhraní
HTTP	Hypertext Transfer Protocol
HTML	značkovací jazyk pro vytváření webových stránek
IP	Internet Protocol
JIT	just-in-time – kompilace programu za jeho běhu
JS	jazyk JavaScript
MFD	multifunkční zařízení
MIT	svobodná licence
MSI	Windows Installer
MVVM	Návrhový vzor model-view-viewModel
REST	Representational State Transfer – architektura rozhraní
RHEL	operační systém Red Hat Enterprise Linux
RID	Runtime IDentifier – identifikátor cílových platforem
RoH	Robot Health-check – druhá část této práce umožňující kontrolu stavu aplikací
Ruda	Robot Updater Apparatus – první část výsledku této práce umožnující aktualizaci aplikací
R&D	Research and Development – oddělení výzkumu a vývoje
SDK	Software Development Kit – sada nástrojů umožňující/ulehčující vývoj
TCP	Transmission Control Protocol – transportní protokol
UI	User Interface – uživatelské rozhraní
UX	User Experience – uživatelský prožitek, jak se UI používá
WPF	Windows Presentation Foundation – subsystém umožnující vytváření GUI
WYSIWYG	what you see is what you get – editor umožnující vidět výslednou podobu během editace
XAML	Extensible Application Markup Language – deklarativní jazyk používaný pro návrh GUI ve WPF

Literatura

[1] Black Duck Open Hub: *The System Development Life Cycle (SDLC)*. Black Duck Software, 2018, [Online; navštíveno 19.01.2018].
URL https://www.openhub.net/repositories/compare

[2] Brockschmidt, K.; Myers, A.: *Nuget: An introduction to NuGet*. Říjen 2018, [Online; navštíveno 09.04.2018].
URL https://docs.microsoft.com/en-us/nuget/what-is-nuget

[3] Chacon, S.; Straub, B.: *Pro Git book: 3.5 Git Branching - Remote Branches*. Duben 2018, [Online; navštíveno 01.05.2018].
URL https://git-scm.com/book/en/v2/Git-Branching-Remote-Branches

[4] Chacon, S.; Straub, B.: *Pro Git book: Git Internals - Git Objects*. Duben 2018, [Online; navštíveno 08.04.2018].
URL https://git-scm.com/book/en/v2/Git-Internals-Git-Objects

[5] Costura.Fody contributors: *Costura.Fody: Readme*. [Online; navštíveno 19.04.2018].
URL https://github.com/Fody/Costura/blob/master/README.md

[6] Driessen, V.: *A successful Git branching model*. Leden 2010, [Online; navštíveno 20.01.2018].
URL http://nvie.com/posts/a-successful-git-branching-model/

[7] GitVersion contributors: *GitVersion: Easy Semantic Versioning for projects using Git*. 2018, [Online; navštíveno 21.01.2018].
URL https://github.com/GitTools/GitVersion

[8] HashiCorp: *Consul Documentation: Checks*. [Online; navštíveno 19.04.2018].
URL https://www.consul.io/docs/agent/checks.html

[9] HashiCorp: *HashiCorp Consul: Service Discovery and Configuration Made Easy*. [Online; navštíveno 19.04.2018].
URL https://consul.io

[10] JFrog: *JFrog Artifactory: Artifactory REST API*. [Online; navštíveno 16.04.2018].
URL https://www.jfrog.com/confluence/x/LI2-Ag

[11] JFrog: *JFrog Artifactory: Features*. [Online; navštíveno 15.04.2018].
URL https://jfrog.com/artifactory/features/

[12] Knuth, D. E.: *The future of TEX and METAFONT*. Říjen 1990, [Online; navštíveno 07.04.2018].
URL http://www.ntg.nl/maps/05/34.pdf

[13] Koch, M.: *nuke: Cross-platform build automation system.* [Online; navštíveno 17.04.2018].
URL https://nuke.build

[14] Landwerth, I.: *.NET Blog: Introducing .NET Standard.* [Online; navštíveno 19.04.2018].
URL https://blogs.msdn.microsoft.com/dotnet/2016/09/26/introducing-net-standard/

[15] .NET Foundation: *.NET Foundation: About.* [Online; navštíveno 10.04.2018].
URL https://www.dotnetfoundation.org/about

[16] PlayFab: *Consul.NET: .NET API for Consul.* [Online; navštíveno 18.04.2018].
URL https://github.com/PlayFab/consuldotnet

[17] Preston-Werner, T.: *Semantic Versioning: A meaningful method for incrementing version numbers.* 2013, [Online; navštíveno 20.01.2018].
URL https://semver.org/spec/v2.0.0.html

[18] Radack, S.: *The System Development Life Cycle (SDLC).* National Institute of Standards and Technology, Duben 2009, [Online; navštíveno 19.01.2018].
URL https://csrc.nist.gov/csrc/media/publications/shared/documents/itl-bulletin/itlbul2009-04.pdf

[19] Rossi, J.: *Castle Windsor: IoC.* [Online; navštíveno 19.04.2018].
URL https://github.com/castleproject/Windsor/blob/v4.1.0/docs/ioc.md

[20] Schwaber, K.; Sutherland, J.: *The Scrum Guide.* Listopad 2017, [Online; navštíveno 06.04.2018].
URL
http://www.scrumguides.org/docs/scrumguide/v2017/2017-Scrum-Guide-US.pdf

[21] Squirrel.Windows contributors: *Squirrel.Windows: An installation and update framework for Windows desktop apps.* 2018, [Online; navštíveno 24.01.2018].
URL https://github.com/Squirrel/Squirrel.Windows

[22] Squirrel.Windows contributors: *Squirrel.Windows: What Do We Want?* 2018, [Online; navštíveno 25.01.2018].
URL https://github.com/Squirrel/Squirrel.Windows

[23] de la Torre, C.: *.NET Core, .NET Framework, Xamarin – The "WHAT and WHEN to use it".* [Online; navštíveno 19.04.2018].
URL https://blogs.msdn.microsoft.com/cesardelatorre/2016/06/27/net-core-1-0-net-framework-xamarin-the-whatand-when-to-use-it/

[24] Torvalds, L.; Hamano, J.: *Git: Fast version control system.* 2015, [Online; navštíveno 19.01.2018].
URL http://git-scm.com

[25] Wenzel, M.; Lander, R.; contributors: *.NET Guide: .NET Standard.* [Online; navštíveno 19.04.2018].
URL https://docs.microsoft.com/en-us/dotnet/standard/net-standard

[26] WiX: *WiX documentation: List of Tools*. [Online; navštíveno 10.04.2018]. URL
http://wixtoolset.org/documentation/manual/v3/overview/alltools.html