ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Adámek** Jméno: **Štěpán** Osobní číslo: **435511**

Fakulta/ústav: **Fakulta informačních technologií**

Zadávající katedra/ústav: **Katedra softwarového inženýrství**

Studijní program: **Informatika**

Studijní obor: **Webové a softwarové inženýrství**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Návrh a implementace C# knihovny pro práci s frameworkem X-definice

Název diplomové práce anglicky:

Design and implementation of C# library for working with X-definice framework

Pokyny pro vypracování:

X-definice je framework určený k validaci a transformaci strukturovaných dat podporující též stavové zpracování dokumentů.Cílem této práce je zprostředkovat služby frameworku tak, aby bylo možné jej používat z programovacího jazyka C#/.NET(.NET Core).
1. Seznamte se s frameworkem X-definice.2. Analyzujte možnosti zpřístupnění frameworku X-definice v C# (přepis celého kódu, komunikační rozhraní, interakce s JVM, ...).3. Zvolte variantu řešení a volbu zdůvodněte.4. Zvolenou variantu implementujte, řádně otestujte a zdokumentujte.

Seznam doporučené literatury:

Dodá vedoucí práce.

Jméno a pracoviště vedoucí(ho) diplomové práce:

Mgr. Václav Trojan, katedra softwarového inženýrství FIT

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **30.06.2019** Termín odevzdání diplomové práce: _____

Platnost zadání diplomové práce: _____

Mgr. Václav Trojan podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	doc. RNDr. Ing. Marcel Jiřina, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

.	
Datum převzetí zadání	Podpis studenta

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Návrh a implementace C# knihovny pro práci s frameworkem X-definice

Bc. Štěpán Adámek

Katedra softwarového inženýrství
Vedoucí práce: Mgr. Václav Trojan

27. května 2020

Poděkování

Rád bych poděkoval všem, kteří mě podporovali při psaní této diplomové práce. Děkuji vedoucímu práce Mgr. Václavu Trojanovi za cenné rady a hlavně možnost podílet se na vývoji takto zajímavého projektu. Dále bych chtěl poděkovat všem, kteří mi byli oporou po celou dobu studia, zejména mým přátelům, rodině, přitelkyni a členům kapely The Latecomers.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu) licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 27. května 2020 . . . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií

Odkaz na tuto práci

Adámek, Štěpán. *Návrh a implementace C# knihovny pro práci s frameworkem X-definice*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2020. Dostupný také z WWW: ⟨`https://github.com/adameste/xdef.net`⟩.

Abstrakt

Tato diplomová práce se zabývá převodem frameworku X-definic z jazyka Java do jazyka C#. V rámci tohoto úkolu proběhne analýza existujících možností převodu Java knihovny a dalších možností realizace tohoto úkolu, na základě které proběhne návrh výsledné knihovny. Výsledkem je funkční .NET knihovna ve formě balíčku NuGet.

Klíčová slova X-definice, XML, Java, .NET, framework.

Abstract

This master's thesis is focused on implementation of existing Java library X-definitions to C# programming language. To achieve this task, existing options of transforming Java library to C# as well as other options will be analyzed and library will be designed using the gained knowledge. Result of this thesis is fully functional .NET library in the form of NuGet package.

Keywords X-definition, XML, Java, .NET, framework.

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Framework X-definice slouží pro popis struktury XML dokumentů, zároveň však umožňuje i popsat jejich zpracování nebo konstrukci. Tedy s jeho pomocí je možné provádět validaci XML dokumentů, vytvářet je či jinak zpracovávat. Nástroj byl vytvořen firmou Syntea software group a.s. a 19.6.2018 byl uvolněn zdrojový kód pod licencí Apache 2.0.[5] Framework funguje jako knihovna v jazyce Java a stále na něm probíhá vývoj. Jako příklad lze uvést přidání podpory pro zpracování dokumentů ve formátu JSON, kterého se framework dočkal v roce 2019. [6]

Framework je primárně určen pro aplikace, které zpracovávají obrovské množství XML dokumentů, případně velké dokumenty – framework dokáže zpracovat dokumenty neomezené velikosti v režimu „stream“. [5]

Hlavním úskalím X-definic je, že v současné době je k dispozici pouze verze pro programovací jazyk Java. Cílem této práce je potom vytvořit implementaci knihovny v jazyce C#, který se také těší velké oblibě programátorů.

Toto téma jsem si vybral protože od druhého ročníku bakalářského studia se živím programováním .NET aplikací v jazyce C#. Tedy v mém případě necelých pět let. Také jsem chtěl prakticky zaměřenou diplomovou práci, jejíž předmětem bude implementace nějaké aplikace nebo knihovny a bude užitečná pro další lidi.

Diplomová práce je vypracovaná na základě spolupráce s firmou Syntea, zejména pak vedoucím této práce Mgr. Václavem Trojanem, která stojí za celou technologii X-definice.

Kapitola 1

Specifikace cíle

1.1 Cíle práce

Cílem této práce je přidat podporu pro X-definice do jazyka C#. Je potřeba zvážit možnosti implementace s přihlédnutím k rozsahu původního zdrojového kódu a jeho budoucí udržovatelnosti. Původní Java knihovna je multiplatformní a bylo by dobré tuto vlastnost zachovat i u implementace v jazyce C#. Výsledná implementace této knihovny bude řádně otestovaná jak z hlediska funkčnosti, tak z hlediska výkonu. Velký důraz je přitom kladen na řádnou dokumentaci kódu. Výstupem této práce bude knihovna nesoucí název xdef.net publikovaná jako hotový balíček ve správci balíčků NuGet [7], což je správce balíčků pro .NET (.NET Framework i .NET Core).

Vzhledem k tomu, že jazyk C# a jeho hlavní běhové prostředí .NET prošly během posledních let a vzniklo i nové běhové prostředí .NET Core, které je multiplatformní a s nízkým overheadem. Navíc stále probíhá jeho vývoj a čím dál více se upřednostňuje před klasickým .NET Frameworkem. Například již pro psaní své bakalářské práce v roce 2017 jsem jej zvolil jako framework (tehdy ještě ve verzi 1) pro její implementaci, kvůli nutnosti provozu na OS Linux. [8] Pro tvorbu knihovny je tedy zapotřebí využít nový framework pro knihovny .NET Standard, který umožní běh jak na jak na .NET Core, tak i na .NET Framework – tedy knihovnu bude možné použít jak ve starších tak i novějších aplikacích. [9]

Cílem práce tedy bude vytvořit knihovnu v .NET Standard 2.0, která by měla mít dostatečnou funkcionalitu a zároveň podporovat všechny cílové platformy. Přehled podporovaných platforem je uveden v tabulce 1.1. Kompatibilita je uvedena pouze pro verze 1.3 a novější.

.NET Standard	**1.3**	**1.4**	**1.5**	**1.6**	**2.0**	**2.1**
.NET Core	1.0	1.0	1.0	1.0	2.0	3.0
.NET Framework	4.6	4.6.1	4.6.1	4.6.1	4.6.1	N/A
Mono	4.6	4.6	4.6	4.6	5.4	6.4
Xamarin.iOS	10.0	10.0	10.0	10.0	10.14	12.16
Xamarin.Mac	3.0	3.0	3.0	3.0	3.8	5.16
Xamarin.Android	7.0	7.0	7.0	7.0	8.0	10.0
UWP	10.0	10.0	10.0.16299	10.0.16299	10.0.16299	TBD
Unity	2018.1	2018.1	2018.1	2018.1	2018.1	TBD

Tabulka 1.1: Kompatibilita jednotlivých verzí .NET Standard. [3]

1.2 Současný stav

V době psaní této práce je framework X-definice ve verzi 4.0. Kód je celý napsaný v jazyce Java a samotné X-definice podporují volání externích Java funkcí, případně sdílení proměnných, což bude představovat největší výzvu při převodu do jazyka C#. Základními vlastnostmi podle oficiálních stránek X-definic jsou: [5]

- Intuitivně čitelný popis struktury XML dokumentů, založený na XML.
- Kontrola (validace) a zpracování XML dat.
- Konstrukce a transformace XML dat.
- Snadná údržba velkého množství datových struktur v rozsáhlých projektech
- Zpracování a transformace dat v různých jazykových verzích („lexicon“)
- Podrobné hlášení chybových situací a možnost programového zpracování chyb
- Možnost volání externích metod v Javě v různých fázích zpracování
- Generace Java tříd reprezentujících zpracovaná XML data
- Spojení s databázemi a jinými externími zdroji dat
- Zpracování neomezeně rozsáhlých dat (mnoho GB, režim „stream“)

V roce 2018 byl uvolněn zdrojový kód pod licencí Apache license 2.0, takže bude vhodné implementaci v C# vydat také pod touto licencí. Podrobnější popis funkcionality frameworku X-definice jsou v kapitole 2.

Kapitola 2

X-definice

V této kapitole jsou popsány funkce frameworku X-definic, aby bylo jasné, co všechno je potřeba implementovat pro dosažení stanovených cílů této práce. V průběhu psaní došlo k vydání nové verze 4.0, kterou se budeme v následujících sekcích zabývat. Framework X-definice je projektem firmy Syntea, přičemž o jeho rozšíření se postarali i studenti ČVUT formou závěrečných prací – tedy jediným zdrojem informací jsou závěrečné práce mých předchůdců a oficiální dokumentace od firmy Syntea.

Text se drží anglických výrazů pro pojmenování jednotlivých součástí systému, oficiální dokumentace je psána také v anglickém jazyce a předpokládá se, že programátoři jsou více důvěrně seznámeni s anglickými výrazy jako element – prvek, node – uzel apod.

2.1 XML

XML (eXtensible Markup Language) je markup jazyk (česky rozšiřitelný značkovací jazyk) – tedy slouží ke strukturování textu pomocí značek. „Značkovací jazyky užívají jako značek k oddělení jednotlivých prvků zpracovávaného textu definovaných posloupností znaků. Posloupnost znaků, která je značkou, se nesmí vyskytovat ve zpracovávaném textu jinak, než právě v roli značky definující jednotlivé prvky.“ [10] Značky lze rozdělit na 2 typy:

1. **Procedurální** – Definuje konkrétní akci, která se má s prvkem provést.

2. **Deklarativní** – Definuje prvek jako logickou část v rámci značkovaného textu.

[10]

Historie jazyka XML začala vývojem SGML (Standardized Generalized Markup Language), který vyvinuli Charles Goldfarb, Ed Mosher a Ray Lorie v laboratořích IBM v 70. letech minulého století. SGML, v rozporu s jeho názvem, není markup (značkovací) jazyk, ale spíše jazyk určený ke specifikaci značkovacích jazyků. Vedle XML je jednou z aplikací SGML i jazyk HTML, který na rozdíl od XML slouží k prezentaci dat a nepovažuje se za vhodný jazyk k jejich ukládání.[11]

Jazyk XML je čitelný jak strojově, tak i člověkem. Zároveň je dostatečně flexibilní, aby umožnil výměnu dat mezi různými platformami a architekturami. [11] Příkladem XML je následující XML dokument obsahující údaje o osobě:

```
1 <Osoba>
2   <Jméno>Karel</Jméno>
3   <Příjmení>Novák</Příjmení>
4   <Město>Praha</Město>
5   <Plat>44000</Plat>
6 </Osoba>
```

Listing 2.1: Příklad XML dokumentu

2.2 X-definice

X-definice je open-source nástroj pro popis, zpracování a konstrukci dat ve formátu XML vyvinutý firmou Syntea. Poskytuje nástroje pro popis struktury i vlastností hodnot dat XML (JSON) dokumentu. Navíc umožňuje popsat zpracování či konstrukci XML objektů – tedy jejím cílem je nahrazení dnes běžně používaných technologii jako jsou XML Schema a DTD (Data Type Definition). [4, s. 7]

Hlavní vlastností X-definice je zachování co největšího ohledu na popisovaná data – X-definice je XML dokument strukturou podobný popisovaným datům. „This makes possible quickly and intuitively describe given XML data and its processing. “[4, s. 7] – Data a jejich zpracování je možné rychle a intuitivně popsat.

Ukázkovou X-definicí popíšu příklad XML dokumentu 2.1. Všimneme si, že v „xd:def“ je definován kořenový prvek . X-definice pro tento dokument vypadá následovně:

```
1 <xd:def xmlns:xd="http://www.xdef.org/xdef/4.0" root="Osoba">
2   <Osoba>
3     <Jméno>string()</Jméno>
4     <Příjmení>string()</Příjmení>
5     <Plat>int(10,100000)</Plat>
```

```
6     </Osoba>
7   </xd:def>
```

Listing 2.2: Příklad X-definice pro ukázku 2.1

2.2.1 Módy operace

X-definice podporují dva operační módy – konstrukční a validační. [4, s. 72]

2.2.1.1 Validační mód

Tento mód je řízen vstupními daty, která jsou validována/zpracovávána oproti modelům v poolu X-definic. Dochází k sekvenčnímu zpracování dat a dochází k vyvolání akcí během různých událostí a úrovní zpracování. Vstupní data mohou být také předána procesoru ve formě tříd z knihovny Java *org.w3c.dom* Element a Document. [4, p. 72] Algoritmus zpracovává data následovně:

1. **Začne se zpracovávat root element** – Procesor X-definic najde root element a provede akce definované v XML atributu *xd:script*.
2. **Zpracování elementu** – Procesor najde další element v datech, a pokusí se daný element najít i v X-definici. Pokud je element definován v X-definici, provede se kontrola maximálního počtu výskytů. V případě, že se nevyskytnou žádné chyby, pokračuje se v dalším zpracování.
3. **Zpracování atributů elementu** – Seznam atributů elementu v datech je porovnán se seznamem atributů v X-definici, a pokud je specifikováno, je provedena i kontrola hodnoty atributu.
4. **Začátek zpracování obsahu elementu (child nodes)** – Proběhne vyvolání události *onStartElement* a v závislosti na obsahu se pokračuje krokem 5 nebo 6.
5. **Zpracování child elementů** – Pokud prvek obsahuje child elementy, jsou zpracovány rekurzivně podle kroku 2.
6. **Zpracování textu child nodu** – U každého textového child nodu (česky uzel), podobně jako u atributů dojde k validaci oproti definici. Tyto nody již neobsahují další elementy, vyhodnocování se tedy v této rekurzi vrací o úroveň výše.
7. **Konec zpracování obsahu elementu** – Provede se kontrola minimálního počtu výskytů daného elementu, a pokud je úspěšná vyvolá se událost *finally*.

[4, s. 72]

2.2.1.2 Konstrukční mód

Konstrukční mód, na rozdíl od toho validačního, není řízen vstupními daty, ale samotnou X-definicí. Na začátku musí být definován model, podle kterého bude probíhat konstrukce. Následně jsou rekurzivně zpracovávány podřazené prvky. Data potřebná pro vytvoření výsledného dokumentu jsou specifikována v sekci modelu *create*. Tato sekce je ve validačním režimu ignorována. V takovém modelu X-definice lze definovat konstrukci elementů, atributů i textových nodů.[12, 4]

Procesor X-definic při práci postupuje následovně:

1. **Výběr modelu root elementu** – Model root elementu je specifikovaný v metodě *xcreate*, pokud nedojde k chybě, pokračuje se krokem 2.
2. **Vytváření elementu** – Pro všechny objekty specifikované v X-definici je vykonána akce *create*. Začíná se kořenovým prvkem a postupuje se rekurzivně. Mohou být vyvolány události stejné jako v případě validačního módu. Pokud není akce *create* v skriptu specifikována, provede se výchozí akce. Ta použije aktuální hodnotu tzv. kontextu – což je struktura, která je přístupná procesoru X-definic na daném místě.

[4, s. 73]

2.2.2 Modely v X-definici

V X-definicích se používají dva druhy modelů – modely prvků a modely datových hodnot (v atributech a textových nodech).

- **Modely elementů** – Model elementu je vždy přímý podřazený prvek X-definice. Lze se z něj odkazovat na jiné modely elementů. Může tedy být popisem root elementu, nebo jeho child elementů. Je to tedy taková základní jednotka, ze které jsou X-definice složené.
- **Modely datových hodnot** – Slouží k validaci hodnot uvnitř atributů a textových nodů. Pro kontrolu platnosti hodnoty se volá „validační funkce“. Příkladem mohou být například funkce *int()*, *string()*, *int(min, max)*.

[4, s. 8]

2.2.3 X-script

X-script je jazykem používaným pro popis vlastností datových objektů v X-definici. Může být deklarován v hodnotě pomocného atributu *xd:script* v modelu elementu nebo jako popis textového nodu. Dále může být definován v hlavičce X-definice, kde může popisovat více vlastností X-definice. Vedle popisu

vlastností dat také může specifikovat akce, které se mají provést při vyvolání různých událostí. [4, s. 23]

Jednotlivé tokeny mohou být odděleny libovolným množstvím whitespace znaků. Může mít více sekcí deklarovaných v libovolném pořadí. Tyto sekce jsou odděleny středníkem (;). Středník může být vynechán na konci skriptu nebo za složeným výrazem ve složených závorkách (). [4, s. 23]

V X-Skriptu můžeme deklarovat následující sekce:

- **Validační sekce** je je odlišná pro textové hodnoty a prvky.
  - **Textové hodnoty** jsou popsány specifikací požadovaného počtu výskytů (*required*, *optional*), volitelně specifikací pro validaci hodnoty.
  - **Elementy** mají určený minimální a maximální počet výskytů.
- **Sekce popisující akce v návaznosti na události** – Specifikují, co se má udělat pokud nastanou různé události. Každá takováto specifikace začíná názvem události, na ten navazuje akce, která se má při vyvolání dané události provést. V jednom X-scriptu lze specifikovat více akcí. Například mohou navazovat na samotnou validaci, třeba událostmi *onTrue* (validace v pořádku) nebo *onFalse* (chyba validace).
- **Možnosti** – Specifikace začíná klíčovým slovem *options*, následuje čárkou oddělený seznam možností z tabulky v dokumentaci, např. „trim“. Je možné uvést pouze jeden seznam možností.
- **Reference** – Odkaz na model prvku z X-definice, je možné specifikovat pouze jeden. Začíná klíčovým slovem *ref* následovaný adresou.
- **Deklarace proměnných** – Deklarace proměnných spojených s instancí daného prvku. Je uvozena klíčovým slovem *var* a musí být deklarována na začátku skriptu.
- **Komentáře** – Komentáře uzavřené v sekvenci znaků */* */*.

[4, s. 23-24]

2.2.4 Externí metody

Často využívanou možností X-definic je volání externích funkcí. Externí metody musí být implementovány v programu jako statické a návratová hodnota musí korespondovat s deklarací této metody v X-definici. Externí metody jsou deklarovány v bloku *xd:declaration* pomocí klíčového slova *external* následovně: [4]

```
1 <xd:declaration>
2   external method boolean
       com.myproject.ExampleClass.exampleMethodB(XXElement);
3   ...
4 </xd:declaration>
5 <a attr1="string;exampleMethodB();onFalse()error();">
6 </a>
7 ...
```

Listing 2.3: Deklarace externích volání.

Tedy po deklaraci metod je můžeme volat podobně jako metody zabudované v X-definicích. Argumenty typu *XXNode*, *XXNode* a *XXData* jsou metodě předány automaticky (pokud jsou definované jako první argument) v závislosti na kontextu. Je potřeba si dát pozor na kontext X-definice, metody deklarované v jedné X-definici nelze v případě ukázky volat z jiných X-definic, do deklarace by se musel přidat atribut *scope="global"*. [4]

Jednotlivé datové typy v X-definici jsou mapovány na datové typy v Javě podle následující tabulky 2.1.

Type v prostředí X-script	**Typ předaný externí metodě**
boolean	java.lang.boolean
Datetime	java.util.Calendar
Float	java.lang.double
Int	java.lang.long
Regex	java.util.regex.Pattern
RegexResult	java.util.regex.Matcher
String	java.lang.String
Element	org.w3c.dom.Element
Bytes	java.lang.byte[]
Container	org.xdef.XDContainer

Tabulka 2.1: Mapování datových typů při volání externích metod. [4]

2.2.5 X-pozice

V X-definicích je možné se z modelu odkazovat na jiné modely pomocí X-pozice. Jedná se o velmi důležitou součást systému. Při vyhodnocování X-definic se často používá více X-definic najednou, které jsou zkompilované v objektu *XDPool* a je potřeba umožnit vzájemné reference (pomocí klíčového slova *ref*).

K tomuto účelu slouží *XDPosition*, což je zápis který umožní přesně specifikovat element, případně atribut, na který je odkazováno. X-pozice začíná názvem odkazované X-definice následovaný znakem #. Za tím následuje cesta k cílovému elementu nebo atributu. Zde je ukázka základního použití X-pozice: [4]

```
<xd:def name="A">
  <a>
  <x/>
    <y z="string()"/>
  <x/>
  </a>
</xd:def>
<xd:def name="B">
  <b xd:script="ref A#a"/>
</xd:def>
```

Listing 2.4: Ukázka X-pozice

Pomocí X-pozice se lze odkazovat také na child elementy, např. na element *y* odkazuje cesta *A#a/x/y*, případně lze deklarovat i cestu k atributu z pomocí *A#a/x/y@z* nebo obdobně cestu k textové hodnotě nahrazením znaku @ za $. Pokud se cílový element nachází ve stejné X-definici, potom lze vynechat název X-definice a znak #. [4, 13]

2.2.6 Reportery

Reportery v X-definicích slouží k zaznamenávání chyb a varování, například při jejich kompilaci nebo validaci. K dispozici jsou interface *ReportReader* a *ReportWriter*, které se předávají volání metod nebo se dají nastavit jako výchozí hodnoty přímo v objektu. Nejběžnější používanou třídou je *ArrayReporter*, který implementuje obě dvě výše zmíněná rozhraní (interface) a X-definice jej využívají jako výchozí hodnotu, protože obsahuje funkci, která v případě výskytu chyb vyvolá výjimku, což je výchozí chování.

2.2.7 X-komponenty

Tato podsekce se věnuje X-komponentám, což je počeštěný název pro součást frameworku X-definice – X-Component. Vzhledem k tomu, že píši práci v češtině a slovo X-Component se poměrně špatně skloňuje v českých větách, tak ve většině textu bude použita počeštěná varianta.

X-komponenta je zdrojový kód Java třídy vygenerovaný ze struktury modelu v X-definici. Jeden model elementu v X-definici odpovídá jedné vygenerované X-komponentě. Hodnoty atributů, elementů a textových hodnot jsou

přístupné přes Java gettery a settery, jejichž jména odpovídají názvům původních prvků. Podobně jako u volání externích metod dochází k převodu jednotlivých typů X-scriptu na Java typy. [13]

Hodnoty child elementů jsou reprezentovány X-Component objekty. V případě, že daný objekt má definovaný výskyt větší než 1, je reprezentován jako typ *java.util.List<>* obsahující příslušné objekty. Totéž platí pro textové hodnoty. Všechny datové typy jsou objekty, nikoliv primitivní typy z důvodu zajištění *null* hodnot. Pro některé datové typy je vygenerováno více getterů, například u *datetime* jsou to navíc metody *timestampOf*, *calendarOf* a *dateOf*. Tyto metody vracejí příslušná data v různém formátu.

X-komponenty byly přidány až v pozdějších verzích frameworku a nepočítalo se s jejich velkým rozšířením, nicméně se začaly hojně používat díky značnému ulehčení práce programátorů – s Java objety se pracuje daleko lépe než s XML daty. Dají se využít i při transformaci dat, nicméně framework X-definice může fungovat i bez nich, pouze na základě samotných XML souborů s X-definicemi.

Jednoduchý příklad X-komponenty je z dokumentace: [13]

```
1 <xd:def xmlns:xd="http://www.xdef.org/xdef/4.0"
     xd:name="Vehicle" xd:root="Vehicle">
2     <Vehicle VIN ="required string()"/>
3     <xd:component>
4     %class com.myproject.xcomponents.Vehicle
5         %link Vehicle#Vehicle;
6     </xd:component>
7 </xd:def>
```

Listing 2.5: Ukázka XComponent

Vygenerovaná Java třída potom vypadá následovně:

```
1 package com.myproject.xcomponents;
2 public class Vehicle implements
     org.xdef.component.XComponent{
3    public String getVIN() {return _VIN;}
4    public void setVIN(String x){_VIN=x;}
5    public String xposOfVIN(){return XD_XPos+"/@VIN";}
6    ...
7 }
```

Listing 2.6: Ukázka vygenerované Java třídy s X-komponenty

Na základě této X-definice lze pomocí třídy *GenXComponent* vygenerovat příslušné X-komponenty do zvolené složky. V tomto případě se jedná pouze o jednu třídu – *Vehicle*. Jednoduchá třída, která obsahuje pouze jeden atribut jako ta z ukázky má ve výsledku asi 110 řádků vygenerovaného Java kódu.

2.2.8 Instance X-Component

Každá X-komponenta implementuje společný interface *org.xdef.component.XComponent*, který umožní převod z/do XML (případně JSON) formátu. X-komponenta může být vytvořena pří procházení XML podle modelu v X-definici, kdy jsou její data načtena z XML (unmarshalling). Také může být vytvořena voláním jejího konstruktoru z kódu Java a převedena na XML nebo JSON voláním příslušné metody z interface XComponent (marshalling). [13]

2.2.9 Jednotlivé příkazy generátoru X-Component

Tato podsekce se věnuje jednotlivým příkazům použitelným při generování X-komponent, které se specifikují v sekci X-definice *xd:component*. Jednotlivé příkazy jsou oddělené středníkem (;). [13]

2.2.9.1 Příkaz %class

Tento příkaz definuje Java třídu, která je vygenerovaná na základě modelu elementu v X-definici. Za slovem *%class* následuje plně kvalifikovaný název Java třídy, tedy včetně package. Dále obsahuje klíčové slovo *%link*, za kterým následuje X-pozice modelu v rámci X-definice. Dále je možné specifikovat, které ze které třídy bude výsledná vygenerovaná třída dědit, případně specifikovat implementované interface. To se dělá pomocí klíčových slov *extends ClassName* a *implements InterfaceName* – opět se musí jednat o plně kvalifikované názvy. Syntaxe odpovídá syntaxi z jazyka Java. [13]

2.2.9.2 Příkaz %bind

Příkaz %bind se používá pro přiřazení elementu, atributu nebo textové hodnoty k hodnotě ve vygenerované Java třídě, včetně příslušných getterů a setterů. Za klíčovým slovem %bind následuje jméno hodnoty, které je použito místo automaticky vygenerovaného. Dále se pomocí klíčového slova %with určí cílová Java třída (opět plně kvalifikovaným názvem) a konečně následuje klíčové slovo %link, za kterým následuje X-pozice cílového modelu/atributu/textové hodnoty. [13]

Příkaz %bind se dá také použít k vytvoření getterů a setterů, které by nebyly jinak automaticky generované, například pokud model třídy odkazuje klíčovým slovem *ref* na nějaký jiný – v takovém případě jsou tyto gettery a settery vygenerovány pouze v předkovi dané třídy a jsou pouze zděděny. [13]

2.2.9.3 Příkaz %interface

Příkaz %interface je použit v případě, že finální model přebírá strukturu z jiného modelu (pomocí klíčového slova *ref*), kterou případně vhodně rozšiřuje o další atributy, elementy nebo textové hodnoty. Z těchto modelů elementů, které se chovají jako interface a odkazují na ně další modely, lze nechat vygenerovat místo třídy (pomocí příkazu *%class*) i interface. Příkaz %interface následuje opět plně kvalifikované jméno výsledného interface v Javě a klíčové slovo *%link* následované XDPozicí modelu elementu v X-definici. [13]

2.2.10 Příkaz %ref

Mnohdy se stává, že *XDPool* je generovaný z více X-definic. Pokud je X-komponenta vygenerována z jiného *XDPoolu*, než který se aktuálně používá, může nastat situace, že X-komponenta je již vygenerována, klidně v jiném souboru *.jar*. V takovém případě je potřeba zabránit jejímu opětovnému vygenerování a to pomocí příkazu *%ref*, pomocí kterého lze definovat již vygenerovanou X-komponentu. Příkaz *%ref* je následovaný plně kvalifikovaným názvem již vygenerované Java třídy, pokračuje klíčovým slovem *%link*, za kterým následuje X-pozice modelu v X-definici. [13]

2.2.11 Příkaz %enum

Pokud je v X-definici specifikován datový typ enum, jsou tato data ve výchozím nastavení v X-komponentách dostupná jako textový řetězec (angl. string). Pokud chceme omezit hodnoty pouze na definované hodnoty, můžeme použít příkaz *%enum*. Datový typ je definovaný v sekci X-scriptu *xd:declaration*, z něj lze v sekci *xd:component* nechat vygenerovat Java enum příkazem *%enum <plně kvalifikovaný název Java třídy> <název enumu>*. [13]

2.2.12 Práce s X-komponentami

Jsou celkem 4 věci, které podle dokumentace s X-komponentami provádět. První je jejich samotné generování pomocí třídy *GenXComponent*, které vygeneruje potřebný Java kód. X-komponenty se kompilují z objektu *XDPool*, kde jsou zkompilované zdrojové kódy jedné nebo více X-definic. Zavoláním funkce *GenXComponent.genXComponent* potom dojde k vygenerování všech X-komponent deklarovaných v definici do příslušné složky. [13]

Dalé je možnost vytvoření instance X-komponenty z XML dat (unmarshalling). X-komponenta se vytváří voláním funkce *createXComponent(<zdroj>, <Java třída>, <reporter>)* nad třídou *XDocument*, kterou lze vytvořit z instance třídy *XDPool*. Toto volání vrátí objekt typu *XComponent*, který lze následně přetypovat na požadovanou třídu.

Třetí funkcionalitou je opětovný převod X-komponenty na XML data (marshalling). Interface *XComponent* definuje metodu *toXml()*, která vrátí objekt typu *org.w3c.Element*.

Poslední funkcionalitou je transformace dat s použitím X-komponent, které je docíleno použitím příkazu *%bind*. Tedy různé elementy, atributy či textové hodnoty se dají namapovat na jiné proměnné (s gettery a settery) v X-komponentě.

Kapitola 3

Analýza

V této kapitole zanalyzujeme existující a možná nová řešení pro provozování Java knihovny v prostředí .NET. Dále se podíváme na kritické sekce X-definic a případné další problémy, se kterými by mohl být největší problém během převodu a na základě tohoto srovnání zvolíme vhodný postup pro zpřístupnění X-definic z .NET aplikací.

Před zahájením analýzy je potřeba si definovat několik klíčových pojmů, které se budou vyskytovat v následujících kapitolách. Tyto pojmy se týkají samotné kompilace a spuštění daného jazyka.

V Javě máme na začátku zdrojový kód, který je následně přeložen kompilátorem do binárního kódu, který je možné spustit na JVM (Java Virtual Machine). Tedy nevzniká binární kód nativní pro dané zařízení v podobě instrukcí pro daný procesor. Instrukce jsou místo toho určené pro virtuální stroj, který zajistí jejich vykonání na běžící platformě – jedná se tedy o interpretaci binárního kódu Java.

V případě .NET je situace o něco složitější. Na začátku procesu je zdrojový kód v některém z jazyků .NET – C#, Visual Basic, IronPython, IronRuby, F#, ...Tento zdrojový kód je následně kompilátorem přeložen do CIL (Common Intermediate Language), jehož instrukční sada je definovaná specifikací CLI (Common Language Infrastructure). Máme tedy přeložený program v CIL. Ten je spuštěn na virtuálním stroji CLR (Common Language Runtime), v této práci však často zmiňován pouze jako .NET runtime. CLR provádí JIT kompilaci CIL do nativního kódu dané platformy, který je následně spuštěn.

3.1 Možnosti převodu/spuštění Java knihovny v .NET

V této sekci se podíváme jaké jsou možnosti při převodu Java knihovny do .NET, ještě lépe její spuštění a budeme se věnovat i jiným možnostem převodu.

3.1.1 Manuální přepis celého kódu

Metoda s pravděpodobně nejlepším předpokládaným výsledkem co se rychlosti a uživatelské (pro koncové) přívětivosti týče. Naopak hlavní nevýhodou je nepřívětivost pro programátora, protože je potřeba přepsat celý kód od nuly, protože X-definice využívají Java třídy a metody pro práci s XML a další, nemluvě o volání externích metod a dalších funkcionalitách, kterými by bylo potřeba se zabývat. Celý kód současné knihovny má zhruba 100 tisíc řádků kódu (a zhruba 31 tisíc řádku komentáře).

Další nevýhodou by byla nutnost náročné údržby dvou knihoven místo jedné a přidávání nových funkcionalit do obou těchto knihoven, nehledě na nutnost testovat obě verze a udržovat stejnou základní funkcionalitu, což je správné validování/konstrukce XML dokumentů podle X-definic.

3.1.2 Automatická konverze kódu do C#

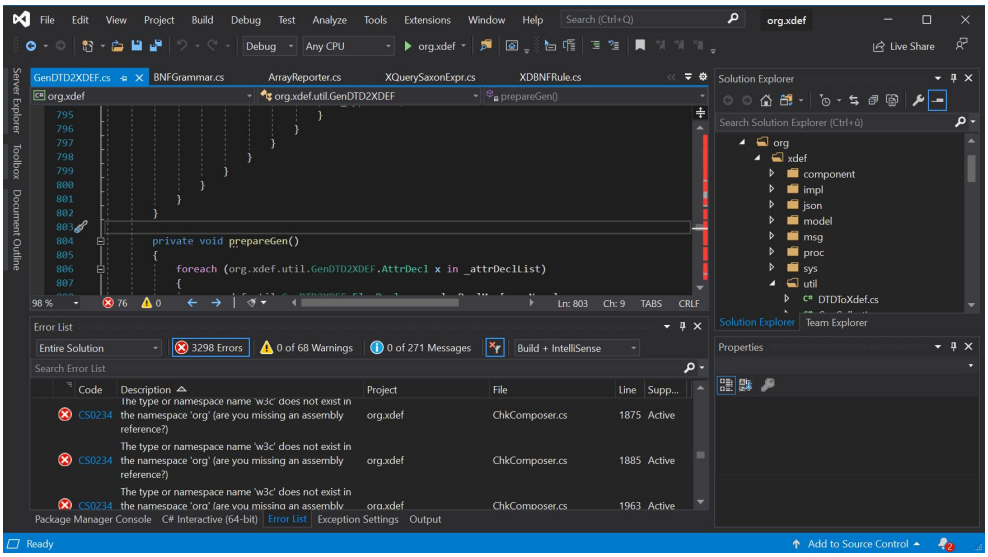
Nechat celý kód Java knihovny převést do C# je možné, předpokládá se však obrovské množství nutných ručních úprav, nehledě na to, že v C# by bylo potřeba mít implementaci knihovních funkcí z Javy a dalších knihoven, které by výsledný kód volal. Mezi nejznámější nekomerční nástroje pro převod Java kódu do jazyka C# patří Sharpen a Janett. Oba tyto nástroje jsou open-source.

Co se Janett týče, jediný funkční web se zmínkami je stránka na serveru Github.com, kde je poslední commit provedený před 6 lety [14], tedy projekt vypadá celkem mrtvě.

Sharpen vznikl v rámci projektu mono, jak napovídá repositář na githubu. Mono implementuje virtuální stroj pro CIL (Common Interface Language) – do toho je jako mezistupeň kompilován jazyk C# (včetně ostatních jazyků .NET, jako je např. Visual Basic). Mono je starší než .NET Core a přinesl podporu pro mnoho architektur a operačních systémů, na rozdíl od tehdejšího .NET Frameworku, který běžel pouze na OS Windows.

Provedl jsem zkušební převod kódu X-definic z javy do C# pomocí nástroje Sharpen. Poslední úprava podle repositáře proběhla před 5 lety, tedy

o rok později než u Janett. Vzhledem k tomu, že tento projekt je také pravděpodobně mrtvý jsem neočekával žádné dobré výsledky. Výsledkem převodu byla hromada zdrojových souborů bez projektu, soubory jsem tedy vložil do projektu, přidal potřebný NuGet balíček, který měl obsahovat některé Java knihovny.



Obrázek 3.1: Výsledný projekt po převodu pomocí nástroje Sharpen.

Kompilace hlásila zhruba 9000 chyb, po promazání projektu od všech testů a GUI komponent se mi povedlo celkový počet chyb zredukovat na 3298, většina z nich se týkala chybějících Java knihoven, tedy oprava by mi zřejmě zabrala déle než ruční přepsání. Navíc výsledkem by byla hromada automaticky generovaného, špatně udržovatelného kódu (zejména při dalších aktualizacích původní knihovny) – viz. obrázek 3.1. [15, 16]

3.1.3 JNI – Java Native Interface

JNI je vrstva kompatibility, která umožňuje Java kódu, který běží v JVM (Java Virtual Machine) volat nativní funkce a opačně. To umožňuje vzájemnou komunikaci aplikací napsaných v javě a jazycích jako jsou C, C++ a Assembler. Používá se zejména v případech, kdy vývojář potřebuje přístup k systémovým funkcím, které nejsou z javy dostupné nebo přístupu k nativním knihovnám. [17]

Mnoho funkcí ze standardní knihovny Java se spoléhá právě na JNI, např. čtení a zápis souborů, případně ho využívají některé další nástroje zmíněné

v této práci. Jednotlivé nativní funkce a datové typy jsou mapovány na funkce a typy v Javě. [17]

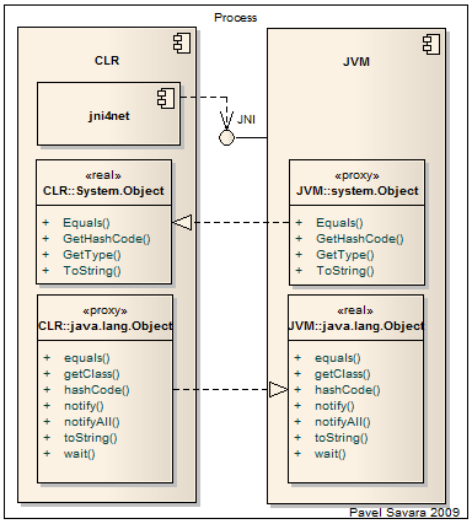
Velkou výhodou je rychlost takovéto vzájemné komunikace. Problém je, že cílem práce je vytvořit knihovnu pro jazyk C#, tedy knihovna napsaná v C/C++ by musela komunikovat jak s Javou, tak se samotným .NET (např. pomocí dalšího mapování z knihovny *System.Runtime.InteropServices*). Takže by musela vzniknout knihovna, která by sloužila jako jakýsi „prostředník“ mezi .NET a Javou – navíc pro každý operační systém a architekturu jiná, což není moc jednoduché na napsání a ani příliš flexibilní. [17, 18]

3.1.4 jni4net

Jni4net je open-source nástroj pro přemostění Javy a .NET založený mimo jiné na JNI. Pomocí reflection jsou získány signatury *public* metod, na základě kterých jsou vygenerovány třídy proxy pro druhou stranu. Proxy třídy jsou pak využity k volání reálných objektů na druhé straně – viz. obrázek 3.2. [1]

Výhodou je, že oba virtuální stroje běží v jednom procesu (jni4net má vestavěný JVM), takže všechna data jsou ve stejném procesu a všechna volání využívají stejné vlákno se stejným zásobníkem. Pokud nedojde k cyklickým referencím mezi oběma stroji, tak normálně funguje i *garbage collector* na obou virtuálních strojích. Výhodou je i dostupnost nástrojů pro vytváření proxy tříd a základní třídy JDK a .NET jsou implementovány na obou platformách.

A teď se dostáváme k nevýhodám tohoto řešení. Opakuje se situace s nástroji pro automatický převod kódu z Javy do C# – tedy vývoj nástroje se zdá být poněkud mrtvý. Z toho plyne i hlavní nevýhoda, což je omezenost platforem. Jni4net funguje pouze pod OS Windows a funguje pouze pod .NET Framework, nikoliv pod .NET Core, jemuž dá předpokládám většina vývojářů přednost. V odnožích repositáře na Githubu jsem v diskuzích našel požadavky a dotazy na implementaci pro .NET Core, ale ta zůstává zatím v nedohlednu.



Obrázek 3.2: Architektura jni4net. [1]

3.1.5 IKVM

IKVM je implementace JVM v prostředí .NET Framework a Mono. Na rozdíl od jni4net se nejedná o jakési přemostění, ale dochází k překladu binárního kódu Javy do CIL. Umožňuje provádět volání z Java programů do prostředí .NET a Mono a z nich lze naopak volat knihovny napsané v jazyce Java. Tedy IKVM spoléhá na cross-kompilaci místo přemosťování. [19, s. 245]

IKVM má za sebou 15 let vývoje, který byl však ukončen v roce 2017, poté co se hlavní autor postupně ztratil důvěru v .NET a částečně i javu. [20] V době psaní této práce se zdá, že se nenašel žádný nástupce, který by IKVM přepsal i pro .NET Core a pokračoval ve vývoji. Tedy hlavním problémem je nedostupnost projektu na platformách, které byly stanoveny jako cíle práce.

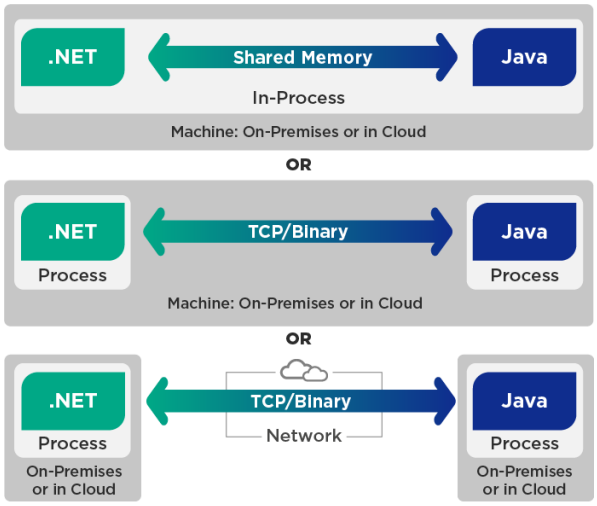
3.1.6 Komerční software

Vedle již zmíněných open-source nástrojů existuje i komerční alternativa JNBridgePro vyvíjená firmou JNBridge. Toto řešení funguje na podobném principu jako jni4net, tedy vzájemně komunikující virtuální stroje pomocí JNI. Na rozdíl od jni4net neustále probíhá vývoj a nejnovější verze podporuje i .NET Core – nicméně podle oficiálních stránek funguje pouze na OS Windows a Linux.

Vedle podpory běhu na stejném zařízení a vzájemné komunikaci obou strojů pomocí JNI nástroj podporuje i vzájemnou komunikaci pomocí TCP využívající proprietární binární komunikaci, tedy virtuální stroje vůbec nemusí běžet

na stejném zařízení, jak je vidět z reklamních materiálů na stránkách – viz. obrázek 3.3.

Nevýhodou je, že toto closed-source řešení je naprosto nevhodné pro open-source projekt, jako jsou X-definice a nelze jej tedy použít. Navíc chybí podpora pro jiné operační systémy, než Windows a Linux, které jsou podporovány knihovnou napsanou v .NET Standard 2.0.



Obrázek 3.3: Podporované režimy nástroje JNBridgePro. [2]

3.1.7 Java wrapper knihovny komunikující s .NET

Poslední možností, kterou se budu zaobírat v této práci je napsání vlastní Java aplikace, která bude schopná využívat stávající knihovnu X-definice a bude komunikovat s protějškem v .NET. Pro vzájemnou komunikaci lze využít více protokolů, v úvahu připadají TCP/IP případně COM. Nevýhodou COM je jeho omezenost platformou (pouze OS Windows), tedy vzájemná komunikace bude muset probíhat přes TCP/IP.

Výhodou takto komunikujících knihoven přes TCP/IP je, že virtuální stroje nemusí běžet na stejném fyzickém stroji, a wrapper knihovny X-definic je teoreticky možné provozovat jako službu pro více klientských knihoven.

Hlavní nevýhodou tohoto řešení je zatím neznámý dopad na rychlost a nutnost napsat dvě aplikace/knihovny – jednu v Javě a jednu jako .NET Standard 2.0 knihovnu v jednom z podporovaných jazyků. Výhodou tohoto řešení je, že výsledná .NET knihovna bude funkční nejen v jazyce C#, což je zadání práce, ale i v dalších jazycích z rodiny .NET (Visual Basic, F#, IronPython, IronRuby, ...).

Co se pracnosti převodu týká, je dle mého odhadu zhruba střední. Není to tak jednoduché jako využití jni4net nebo komerční alternativy, ale zároveň ne tak pracné jako přepsání celé knihovny nebo snaha o zprovoznění automaticky vygenerovaného kódu. Další výhodou jsou celkem minimální úpravy kódu v případě nové verze X-definic – ty musí proběhnout pouze v případě, pokud se změní veřejné rozhraní knihovny X-definic.

3.2 Výběr řešení pro převod

Proběhla analýza několika různých přístupů k převodu Java knihovny do jazyka C# a jsou tři, které splňují zadání – tedy funkčnost i v .NET Core a zachování open-source. První je napsání vlastního mostu mezi .NET a Javou pomocí JNI, případně rozšíření stávající technologie jni4net, což by vyžadovalo detailní znalost obou virtuálních strojů a nutnost psaní/generování C/C++ mostu, takže by se ve finále musela řešit vzájemná komunikace tří jazyků a bylo by asi jednodušší přepsat celý kód X-definic.

Druhá varianta je celé X-definice přepsat manuálně, což je velice pracná varianta i co se následné údržby týče. Navíc by přibyla nutnost celé X-definice znovu důkladně testovat. To by byla škoda, protože máme důkladně otestovaný a fungující kód v Javě.

Třetí varianta je využití stávajícího Java kódu a jeho zprostředkování do prostředí .NET pomocí TCP/IP. Jedná se o rozumně pracnou úlohu, jediné čeho se obávám je výsledná rychlost komunikace a volání externích funkcí. Výhodou je, že server nemusí běžet na stejném zařízení, ale v tomto případě se výsledná rychlost ještě sníží, navíc umožní velkou flexibilitu při vzájemné komunikaci a tedy může dojít k převádění datových typů a případné komunikaci mezi nově vytvořenými objekty, aniž by se muselo jakkoliv zasahovat do původního kódu.

Nicméně ani takto se pravděpodobně nutnosti úpravy a rozšíření samotných X-definic nevyhneme, bude potřeba vyřešit některé problémy, na které jsem narazil během analýzy a během toho, co jsem se seznamoval se samotným frameworkem. Tyto problémy jsou blíže popsané v sekci 3.3.

3.2.1 Užitečné poznatky z ostatních řešení

Během analýzy jsem narazil na několik užitečných poznatků, které by mohly být užitečné při návrhu naší vlastní implementace. V první řadě se jedná o fakt, že všechny nástroje – IKVM, jni4net i komerční alternativy generují kód proxy objektů za pomoci generátorů kódu. Tyto proxy poté provádí serializaci příkazů a odesílají je ke zpracování druhému runtime (běhovému prostředí).

Toto předávání požadavků probíhá různými kanály, nejčastěji však pomocí kombinace P/Invoke, JNI a nativní knihovny mezi nimi, což je varianta asi nejrychlejší. Například komerční JNBridgePro umožňuje realizaci této interakce pomocí protokolu TCP/IP.

Za zmínku stojí také způsob uvolňování paměti v jni4net. Vygenerované proxy třídy implementují finalizér. Finalizér je metoda, kterou volá garbage collector před uvolněním paměti dané třídy. V tomto finalizéru je potom předána informace JVM o uvolnění dané třídy, na základě kterého je smazána příslušná reference na skutečný objekt. Po smazání této reference je reální objekt připravený k odstranění garbage collectorem – tedy pokud na něj neexistují další reference. [21]

Další vlastností jni4net je nutnost vzájemné konverze datových typů, tedy jejich serializace a deserializace. Například textové řetězce se převádějí na bajty v kódování UTF-8 a zpět na textový řetězec v daném jazyce. [21]

Dále máme k dispozici zdrojový kód vygenerovaný pomocí nástroje Sharpen, jehož některé části jsou převedené celkem obstojně. Můžeme z něj tedy v případě potřeby čerpat při implementaci. To je vhodné například při přepisování konstant nebo dokumentace, kterou nástroj převedl do požadovaného formátu.

3.3 Úskalí převodu X-definic

V této sekci se budeme věnovat funkcím a vlastnostem knihovny X-definice, které budou působit největší problémy pří jejich převodu do .NET a jaké jsou možnosti jejich řešení. Vzhledem k tomu, že celá knihovna je úzce navázána na Java knihovny a třídy, například rozhraní pro práci s XML jsou rozdílná a některé funkce X-definic generují vlastní kód v Javě, se kterým pak pracují se nedá předpokládat, že by jakékoliv řešení (včetně těch komerčních) bylo schopné jen tak umožnit plné fungování X-definic pod .NET.

3.3.1 Rozdílné knihovny pro práci s XML

X-definice závisí na Java Java třídách pro práci s XML jako jsou *org.w3c.Element* nebo *org.w3c.Document*, které jsou častým argumentem nebo návratovou hodnotou funkcí X-definic. Oproti tomu v C# jsou pro práci s XML dostupné knihovny dvě.

První a starší knihovnou je XML DOM API, která obsahuje třídy jako *XMLDocument* a *XMLElement*. Tato knihovna se používala ještě v dobách, kdy neexistovala novější knihovna, která lepé zapadá do moderního jazyka

C# – LINQ to XML. LINQ (Language Integrated Query) je součást jazyka C# pro práci především s daty. Ať už se jedná o řetězce, kolekce nebo SQL dotazy, se všemi je možnost pracovat za pomoci LINQ. Tato nová knihovna obsahuje mimo jiné třídy *XDocument* a *XElement*. S těmi se i přes podobnost názvů se starší knihovnou pracuje rozdílně a nelze je vzájemně kombinovat (např. do XElementu nelze vložit XMLElement a obráceně). [22]

Tedy v době psaní této diplomky již nemá smysl se zaobírat starší knihovnou a je potřeba zajistit převod Java tříd na novější variantu na C# straně a dát si pozor, aby všechny převedené typy byly z knihovny LINQ to XML aby nedocházelo k potížím s jejich používáním na straně .NET.

Naštěstí jejich převod z Javy do C# a vlastně skoro jakéhokoliv jazyka je triviální záležitostí, protože XML je jazyk určený mimo jiné k výměně dat, tedy před odesláním stačí nechat daný typ převést na XML data a na druhé straně je převést na příslušnou třídu – což jak Java i C# zvládají bez problémů.

Tento problém tedy není nijak zásadní, vzájemné posílání a zpracování XML dat není takový problém, nicméně zatím není jasné, jaké bude mít tento převod dopady na výkon, to se ukáže až během testování.

3.3.2 Práce s databází

O něco větší problém je možnost X-definic pracovat s databází pomocí jdbc. K tomuto účelu slouží třída *XDService*, která se vytváří z objektu XDFactory, buď poskytnutím uspořádané trojice adresa, uživatel, heslo – a předání tří textových argumentů pro vytvoření není takový problém. Navíc v tomto případě zůstane připojení na JVM a není tedy v tomto ohledu potřebná žádná další jeho interakce s .NET.

X-definice umožňují i použití vlastního připojení pomocí třídy *java.sql.Connection*, které je následně využito samotnými X-definicemi. Bude potřeba dopsat wrapper pro spojení, který bude na straně JVM implementovat interface *org.xdef.XDService* a příkazy bude odesílat do prostředí .NET, které je vyhodnotí a vrátí výsledek, který bude opět převeden na příslušný datový typ.

Funkce definované v tomto interface dále vracejí datové typy frameworku X-definic, jako jsou *XDValue*, *XDResultSet* a *XDStatement*. Bude tedy potřeba zajistit fungování všech těchto typů při napojení na .NET aplikaci. Interface *XDService* je naštěstí celkem osekaný oproti *java.sql.Connection* a obsahuje pouze podmnožinu funkcí:

- **prepareStatement** – Připraví sql příkaz ke spuštění, pokud by měl být prováděn s různými parametry vícekrát z důvodu zvýšení výkonu.

- **execute** – Spustí SQL příkaz s parametry a vrátí výslednou hodnotu, zpravidla číselnou hodnotu s počtem změněných záznamů, ale může to být třeba i počet nalezených hodnot odpovídajících daným kritériím nebo jiné.
- **query** – Spustí query a vrátí výslednou tabulku výsledků v podobě *XDResultSet*.
- **queryItems** – Jako příkaz query, navíc přidává argument *itemName*, který určuje jak se mají jmenovat vrácené položky.
- **close** – Uzavře spojení.
- **commit** – Uloží probíhající transakci.
- **rollback** – Provede rollback.
- **isClosed** – Vrátí true, pokud je spojení již uzavřeno.
- **další** – Obsahuje ještě několik dalších funkcí, které nesouvisí s SQL

Tedy implementace takovéto služby v C# by neměla být zásadně problematická, možná potíž je v dopadu na výkon, pokud se přidá další mezistupeň mezi SQL server a klienta. Za zmínku stojí, že příkazy v C# spojení s databází se v základu neprovádějí jako transakce a je tedy potřeba transakci vytvořit manuálně a spravovat ji.

3.3.3 Volání externích metod

Ještě více problematické je možnost volání externích metod z frameworku X-definice. Externí funkce jsou definované v aplikaci využívající framework, popřípadě v jiné knihovně a aby bylo možné tuto funkcionalitu používat, je potřeba upravit stávající X-definice tak, aby odesílaly požadavky na volání externích funkcí do .NET, kde budou vyhodnoceny a vrácen výsledek.

Samotné nalezení a spuštění statické funkce v daném namespace (jmenném prostoru) není v C# žádný problém díky třídě Reflection. Problém nastává při nutném převodu datových typů používaných v X-definicích a datových typech v C#, jejich serializace a deserializace pro TCP přenos.

Vzhledem k tomu, že celý projekt je opensource, včetně nově vznikajících wrapperů a knihoven, tak je možné umístit externí metody do wrapperu běžícího na JVM a ty následně volat. Tedy vzniká nutnost rozlišení Java a .NET funkcí. Toho je možné dosáhnout například přidáním klíčového slova do deklarace externí metody v X-definici, když se vrátíme k příkladu 2.3, mohl by výsledek se stejnými funkcemi definovanými v .NET klientovi vypadat následovně:

```
1  <xd:declaration>
2    external method (dotnet) bool
        sharpExample.ExampleClass.exampleMethodB(XXElement);
3    ...
4  </xd:declaration>
5  <a attr1="string;exampleMethodB();onFalse()error();">
6  </a>
7  ...
```

Listing 3.1: Rozšíření o volání externích funkcí z .NET

Díky tomuto rozšíření bude možné jednoduše určit, jak se má daná externí metoda vyhodnocovat a může být odeslána do prostředí .NET k vyhodnocení.

Veškerá správa volání externích metod probíhá ve třídě *XCodeProcessorExt*, kterou bude potřeba upravit na základě spolupráce s autorem. Tato třída navazuje na interní fungování X-definic, s různým předáváním informací pomocí zásobníků. Nicméně třída obsahuje pouze pět public funkcí, jejichž funkcionalitu bude potřeba upravit tak, aby volání externích metod v .NET delegovaly na nějaký další objekt. Jedná se o funkce *perform1v*, *perform2*, *perform2v*, *perform* a *performX*. Podle názvů nejsem schopný určit, co přesně dané metody dělají a jaký je v nich rozdíl, tato interní funkcionalita frameworku není příliš zdokumentovaná.

3.3.4 X-komponenty

Největší výzvou pří převodu knihovny pro .NET jsou X-komponenty, které jsou úzce spojeny s Javou a pro fungování v .NET bude nezbytné celý systém přepsat a navíc zajistit komunikaci se stávající knihovnou. Bude potřeba vytvořit kompletní systém knihoven na straně .NET a generátor kódu alespoň pro C# – v .NET běží více jazyků, nicméně není problém tento vygenerovaný kód obsáhnout v separátní knihovně a odkazovat se na něj.

Vytváření mnoha generátorů kódu pro mnoho jazyků z rodiny .NET by bylo přinejmenším velmi pracné a pravděpodobně zbytečné, protože jazyk C# je z nich zdaleka nejpoužívanější a snad už se pomalu upouští od používání Visual Basicu, minimálně podle statistik používanosti programovacích jazyků. [23]

Implementaci X-komponent v původní knihovně najdeme v balíčku *org.xdef.component*, vedle něj bude potřeba implementovat i další části knihovny, zejména pak třídy pro správu hodnot *XDValue* a z něj odvozený inteface *XXData.*

Implementačně se jedná o daleko složitější úlohu než volání externích metod. U těch stačí pouze zavolat danou funkci a vrátit výsledek v požadovaném formátu, zjednodušeně řečeno. U X-komponent bude potřeba napsat kompletně nový systém pro „vzdálené“ X-komponenty a ten napojit na kompletně nový systém na straně .NET knihovny.

3.3.5 Úprava cílů práce na základě zjištěných poznatků

Vzhledem k zjištěným problémům při převodu, které byly nalezeny během analýzy – zejména pak absence funkčního nástroje pro jednoduché využívání Java knihoven pod .NET a velkou napojenost některých části X-definic na Javu je potřeba na základě konzultace s vedoucím práce upravit cíle práce na na realizovatelnou úroveň.

Práce na projektu budou muset pokračovat i nad rámec této práce za účelem zprovoznění volání externích metod a hlavně X-komponent. Je potřeba přiřadit jednotlivým cílům prioritu podle náročnosti jejich implementace a toho, jak moc jsou využívány. Cílem této práce bude zprovoznění validačního a konstrukčního režimu, prozatím bez volání externích metod a X-komponent – což je dosažitelný cíl.

3.4 Specifikace požadavků

Specifikaci požadavků jsem poněkud nekonvenčně zařadil na konec kapitoly analýza z důvodu nutnosti vyhodnocení možností při konverzi knihovny a její pracnosti. Po úpravě cílů práce a analýze problémů, které mohou během převodu nastat už je možné formulovat funkční i nefunkční požadavky na systém.

3.4.1 Funkční požadavky

Funkční požadavky jsou zprovoznění validačního a konstrukčního režimu X-definic. Tyto požadavky se dají dále rozepsat do těchto bodů:

- **Implementace veřejného rozhraní třídy XDFactory** – Třída *XDFactory* je základem celého systému a umožňuje kompilaci zdrojových XML souborů X-definic do binární podoby objektu *XDPool*.
- **Implementace veřejného rozhraní třídy XDPool** – Tato třída obsahuje zkompilovanou sadu X-definic a je možné z ní dále vytvářet objekty *XDDocument*, případně ji zapsat v binární podobě pro další použití.
- **Implementace veřejného rozhraní třídy XDDocument** – *XDDocument* reprezentuje model v X-definicích na jehož základě je možné

volat konstrukční a validační režim X-definic. Veřejné funkce této třídy obsluhují jak validační, tak konstrukční režim X-definic.

  - **Implementace funkcí validačního režimu** – Funkce pro spuštění validačního režimu se jmenuje *xparse* a v současné implementaci má 7 overloadů (přetížení) pro různé typy vstupů a výstupů.
  - **Implementace funkcí konstrukčního režimu** – Konstrukční režim X-definic je spuštěn Zavoláním metody *xcreate*, tato má 3 overloady.

3.4.2 Nefunkční požadavky

Nefunkční požadavky na výslednou knihovnu zahrnují následující body:

- **Řádná dokumentace veřejných metod** – Je potřeba mít řádně zdokumentovanou každou veřejnou metodu nově vzniklé knihovny, podobně jako je tomu v případě Java X-definic.
- **Knihovna poběží i pod .NET core** – Je potřeba aby knihovna fungovala i v prostředí .NET Core, z důvodu zajištění budoucí kompatibility.
- **Zachování rozhraní původní knihovny** – Kde to lze, je vhodné dodržet rozhraní původní Java knihovny, takže stávající uživatelé nebudou mít problém s přechodem a minimalizuje se nutnost dokumentovat nové rozhraní, když už existuje dokumentace k tomu stávajícímu.

Kapitola 4

Návrh

Tato kapitola se věnuje návrhu výsledného řešení na základě poznatků z analýzy. Z té nám jako nejvhodnější způsob vyšlo napsání Java aplikace a přidružené .NET knihovny, které budou komunikovat pomocí TCP/IP. Komunikace bude probíhat obousměrně pomocí vlastního binárního protokolu kvůli rychlosti a flexibilitě.

Z kapitoly analýza (3), nám vyplynulo, že neexistuje open-source univerzální a jednoduché řešení pro převod Java knihovny nebo její spuštění pod .NET. Pro jazyk C# tedy bude nutné provést převod všech funkcí manuálně. Přestože většinou se obejdeme bez jakýchkoliv složitostí a pouze voláme funkce objektů na JVM, framework X-definic má pokročilé funkce jako je například volání externích metod, možnost ladění a X-komponenty.

Tyto funkce bude potřeba napsat od základu ve verzi pro C# a v čase, který jsem si vyhradil na vypracování této práce určitě nestihnu implementovat všechno. Cílem návrhu bude tedy bude vytvoření funkční verze X-definic pro .NET, která bude obsahovat co nejvíce nejpoužívanějších funkcí.

Implementaci .NET knihovny lze použít kterýkoliv jazyk z rodiny .NET, nicméně pro mě je jasná volba C#, protože v něm programuji posledních šest let v rámci práce i vlastních projektů. Samotné X-definice běží na Javě 6 (1.6.x) a novějších, nicméně jedná se o poměrně starou verzi a z důvodu kompatibility jsem zvolil Javu 8 z roku 2014, která je stále produkčním standardem pro nové aplikace. [24]

Cílem je navrhnout knihovnu tak, aby co bylo rozhraní co nejvíce podobné stávající Java verzi, tedy pokud možno zachovat rozhraní a hierarchii tříd. Dále bude vhodné zajistit převod použitých knihovních Java tříd na knihovná třídy v C# a vice versa. Například v javě se rozlišují 2 typy proudů (angl.

streams) – *InputStream* a *OutputStream*. V C# je pouze *Stream* s vlastnostmi *CanRead* a *CanWrite*.

V konečném důsledku tak vzniknou dvě aplikace. První je wrapper knihovny X-definice v Javě pojmenovaný **xdef.bridge**. Druhou je samotná knihovna s ní komunikující – **xdef.net**.

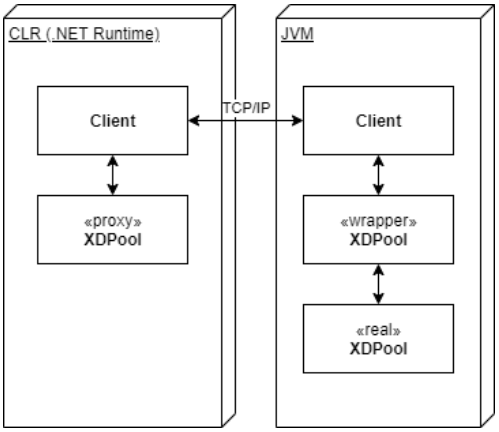
4.1 Klient a server

Pro navázání TCP spojení jsou potřeba dvě strany – klient a server. Serverová strana poběží na JVM, na ni se může připojit několik klientů z více několika knihoven xdef.net – jedině takto zužitkujeme výhodu plynoucí z vybraného způsobu vzájemné interakce virtuálních strojů, tedy možnost provozovat vyhodnocovat X-definice na jiném fyzickém stroji, než na kterém běží aplikace využívající knihovnu.

Knihovna xdef.net se připojí k xdef.bridge pomocí běžného TCP spojení, které je udržováno po celou dobu běhu aplikace. V budoucnu je možné přidat možnost šifrování pomocí TLS, které prozatím není cílem, neboť se předpokládá, že ve většině případů obě strany poběží na stejném stroji, případně v rámci firemní/domácí sítě a tedy není nezbytně nutné zvyšovat nároky na výpočetní výkon nutností šifrovat a dešifrovat veškerou probíhající komunikaci.

4.2 Vzájemná interakce

V této sekci je popsán návrh vzájemné interakce virtuálních strojů pomocí TCP/IP rozhraní. Na straně .NET jsou proxy objekty, reprezentující skutečné objekty, které jsou však umístěné v paměti JVM. Volání funkce takového objektu nejprve vytvoří tcp požadavek který je odeslán na JVM. Tam se požadavek zpracuje a zavolá se příslušná funkce na reálném objektu. Příklad je na obrázku 4.1, kde je vyvolán požadavek na proxy objektu *XDPool*, který je zpracován reálným objektem *XDPool* na straně JVM a přes wrapper a TCP/IP spojení je odeslán výsledek zpět.



Obrázek 4.1: Ukázka vzájemné interakce objektů při volání třídy XDPool.

Aby toto bylo co nejjednodušší pro koncového uživatele knihovny, je potřeba blokovat vlákno do té doby, než je funkce vykonána a v případě potřeby je k dispozici výsledek. Čtení ze síťového proudu je možné pouze v jednom vlákně, otevření více spojení by program zbytečně zkomplikovalo. Tedy hlavní vlákno na kterém probíhá čtení se musí postarat i o odblokování vláken, která čekají na výsledek volání.

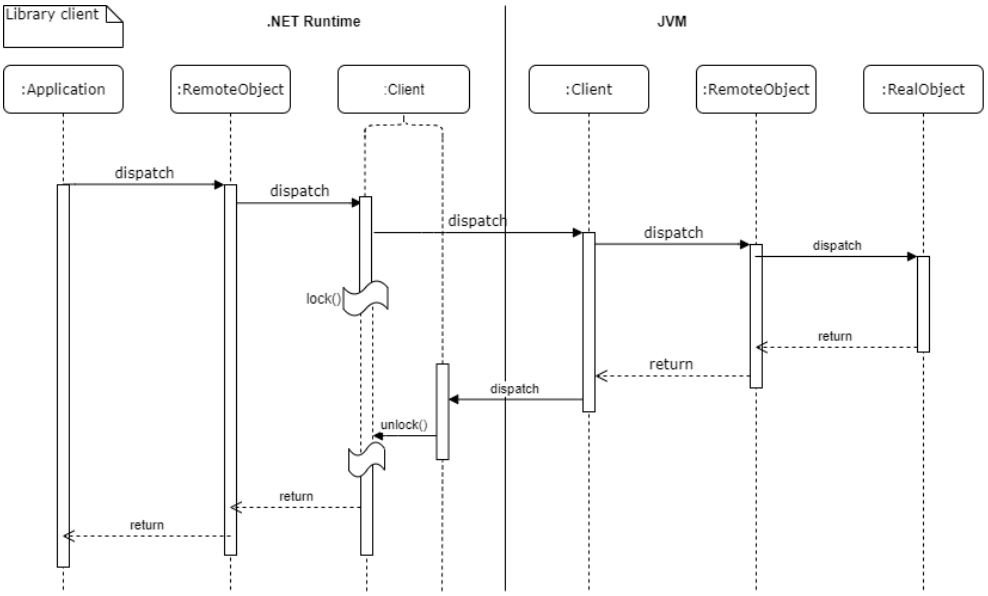
Tato interakce je znázorněna v sekvenčním diagramu 4.2. Z něj je patrné, že volající vlákno z aplikace využívající knihovnu je blokováno do chvíle, než je odemčeno z vlákna, které zpracovává příchozí zprávy. Po odemčení je získán výsledek, který je dále zpracován v objektu *RemoteObject* a výsledek je vrácen původci volání. Diagram znázorňuje volání ze strany .NET, ale v opačném případě vše probíhá stejně.

O přiřazování jednotlivých výsledků a odblokování příslušného vlákna se stará třída *Client*. Ke zjednodušení tohoto párování požadavek-odpověď je potřeba uzpůsobit komunikační protokol.

Další možností je volání na bez určeného objektu protistrany, například pokud je potřeba vytvořit nový objekt, nebo smazat již nepotřebný. Tato volání jsou obsloužena ve třídě *ObjectlessRequestHandler*, který provede příslušnou operaci a případně vrátí výsledek.

Výsledek není potřeba vracet v případě požadavku o smazání. Ten vzniká ve chvíli, kdy je wrapper vzdáleného objektu mazán garbage collectorem (GC). V případě C# je GC vždy volá finalizer označený jako *ClassName()*, na straně Javy je to metoda *finalize()*. Po zavolání této funkce je odeslán protistraně požadavek o odstranění příslušného objektu. Přístup ke všem objektům má

třída *Client*, která mezi ně rozděluje příchozí požadavky. Odkaz na objekt je odstraněn a tedy na něj neexistuje žádný odkaz ze zásobníku – to znamená, že objekt bude odstraněn GC.



Obrázek 4.2: Sekvenční diagram volání vzdáleného objektu.

4.3 Komunikační protokol

Komunikační protokol je binární kvůli rychlosti, předpokládá se přenos většího množství binárních dat a použití nějakého jiného formátu (např. json, XML, ...) by přidalo zbytečný „overhead“ při přenosu dat. Za zvážení stálo i HTTP, nicméně to je nevhodné z důvodu nutnosti obousměrné komunikace (požadavky zasílá jak server, tak klient) – např. objekty *Stream* jsou umístěné v .NET a JVM jen odesílá požadavky.

Požadavky na protokol jsou nutnost snadného párování požadavek-odpověď v případě asynchronního zpracování požadavků, jednoduchost při ladění a snadná identifikace vzdálených objektů, na které požadavek směřuje. Dále musíme zohlednit, že v Javě neexistují „unsigned“ primitivní typy. Návrh výsledného protokolu je v tabulce 4.3.

Název	Typ	Velikost (B)	Offset (B)
ObjectId	int32	4	0
Function	int32	4	4
ClientRequestId	int32	4	8
ServerRequestId	int32	4	12
Payload	int32	4	16
Data	byte[]	Payload	20

Tabulka 4.1: Návrh komunikačního protokolu.

Jednotlivé parametry protokolu jsou stejné při komunikaci oběma směry. Níže jsou podrobněji popsány jednotlivé parametry:

- **ObjectId** – Id vzdáleného objektu, na který směruje dotaz. Nula reprezentuje příkaz bez konkrétního objektu.
- **Function** – Funkce, která se má vykonat na vzdáleném objektu/bez objektu. Každý pár objektů (vzdálený a jeho lokální reprezentace) má vlastní definované funkce, na základě kterých je určena funkce, která se má zavolat.
- **ClientRequestId** – Id požadavku z .NET klientské knihovny, každý požadavek má unikátní id tak, aby se k němu dala snadno přiřadit odpověď. Příkazy, které nevyžadují odpověď a samotné odpovědi toto nastavené mít nemusí (hodnota je 0). hodnota tohoto parametru v odpovědi je rovna požadavku, na který odpovídá.
- **ServerRequestId** – Id požadavku z Java aplikace. Stejné jako v případě *ClientRequestId*. Položky jsou rozdělené z důvodu usnadnění případného ladění aplikace.
- **Payload** – Délka dat v tomto požadavku, pokud je 0, potom požadavek neobsahuje žádná data.
- **Data** – Samotná data, která jsou definovaná na úrovni jednotlivých objektů. Obsahují například parametry volané funkce, nebo její návratovou hodnotu.

Data jsou specifická pro jednotlivé objekty, jednotlivé typy se postupně zapisují za sebe. Řetězci předchází čtyřbajtové *signed* číslo, následují jednotlivé bajty řetězce kódovaného pomocí UTF-8. Odkazy na objekty se předávají pomocí Id objektu (odpovídá *ObjectId*).

Při odpovídání na požadavek již není potřeba specifikovat funkci, k párování požadavek-odpověď dochází za pomoci id požadavku. Pole funkce v tomto případě reprezentuje stavový kód, kde 0 znamená OK, cokoliv jiného je vyhodnoceno jako Chyba a doraz je zpracován jako výjimka, která bude vyvolána.

4.4 Spouštění xdef.bridge

V této sekci si rozebereme možnosti spuštění procesu Java, ve kterém poběží aplikace xdef.bridge – pokud tedy nepoběží jako server na jiném fyzickém stroji. Cílem je navrhnout systém takový, aby zahrnoval co nejméně práce na straně koncového uživatele knihovny. Určitě se nevyhneme požadavku na uživatele mít nainstalovanou javu. Nicméně co se distribuce a spouštění samotné knihovny existuje několik variant.

První z nich je distribuce xdef.bridge separátně s tím, že bude na uživateli aby spustil xdef.bridge a nastavil příslušné údaje nutné pro připojení před voláním knihovny.

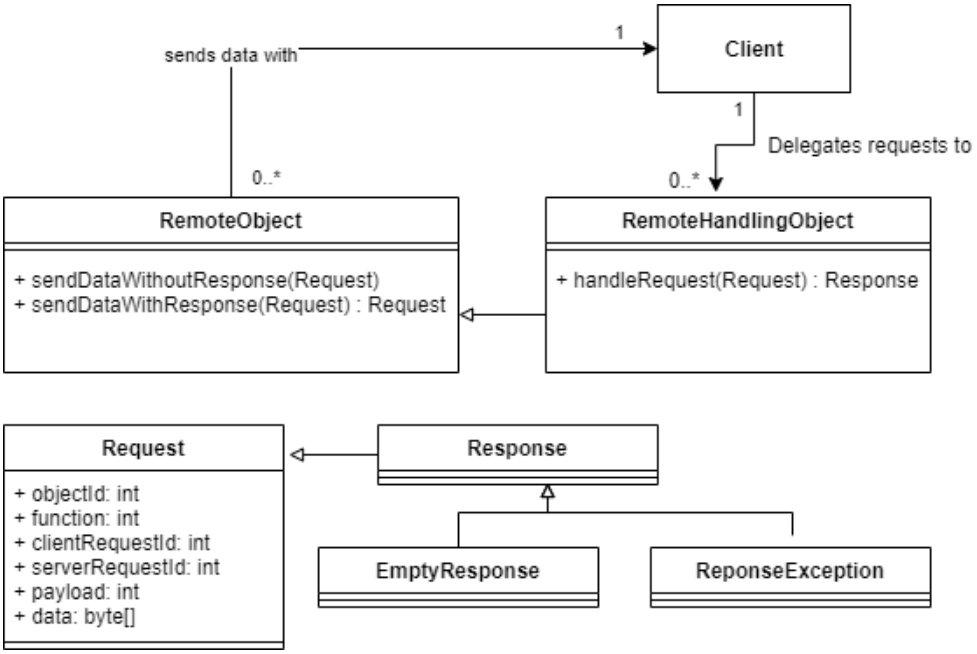
Další variantou je přibalení *.jar* souboru do výsledného balíčku knihovny a jeho spuštění samotnou knihovnou v případě potřeby. Toto je celkem dobrá varianta, ale vzniká nutnost mít v rámci knihovny více souborů a ještě k tomu jiné, než standardní *.dll* jako je běžné u jiných .NET knihoven.

Třetí variantou, které přichází v úvahu je zakompilování *.jar* souboru do výsledného souboru *.dll*. Výhodou je, že uživatel ani nepozná, že součástí knihovny je nějaký jiný soubor než jeden trochu větší *.dll*, nevýhodou je nutnost xdef.bridge před samotným spuštěním nejdříve rozbalit do nějakého dočasného umístění, spustit a po ukončení opět smazat.

Výsledná aplikace xdef.net bude podporovat první a poslední variantu, s tím že výchozí bude automatické rozbalení a spuštění. V případě potřeby však bude možné v konfiguraci nastavit server, na kterém již běží xdef.bridge a nedojde k automatickému rozbalění a spuštění.

4.5 Struktura tříd

V této sekci je rozebraná struktura základních tříd pro komunikaci a vyřizování požadavků. Struktura je stejná na straně Javy i C#, je k vidění na obrázku 4.3.



Obrázek 4.3: Struktura tříd sloužící pro vzájemnou komunikaci.

- **Client** – Abstraktní třída, která se stará o odesílání a přijímání požadavků. Udržuje informaci o všech třídách, které jsou umístěné v běhovém prostředí – tedy wrappery, které čekají na požadavky z druhé strany klienta a zaobalují skutečné objekty.
- **RemoteObject** – Abstraktní třída, která reprezentuje vzdálený objekt a zajišťuje komunikaci s jeho wrapperem. Nepřijímá tedy žádné požadavky od druhé strany, pouze požaduje vyhodnocení funkcí.
- **RemoteHandlingObject** – Abstraktní třída, ze které dědí všechny wrappery lokálních objektů. Vyhodnocuje a vytváří odpovědi na vzdálené požadavky.
- **Request** – Datová třída, která obsahuje základní strukturu datové komunikace a je možné ji zapsat do binárního proudu.
- **Response** – Reprezentuje odpověď na požadavek, má rozdílné konstruktory pro potřeby odpovídání na požadavky. Například není potřeba udávat číslo funkce, ani ostatní parametry, které se vypní na vyšší úrovni.
- **EmptyResponse** – Reprezentuje odpověď bez žádného datového obsahu, typicky pro metody bez návratové hodnoty.

- **ReponseException** – Reprezentuje odpověď na požadavek, jehož zpracování vyvolalo výjimku. Tedy konstruktor nastaví příslušný návratový kód a zapíše obsah zachycené výjimky. Návratový kód je uložen do pole funkce, které by bylo jinak nevyužité.

4.6 Skutečné umístění objektů

Z výše specifikovaného návrhu je patrné, že obě strany základního rozhraní *xdef.net* i *xdef.bridge* jsou mimo implementační detaily stejné. Podle návrhu je možné vytvářet jakési „proxy“ objekty komunikující pomocí wrapperu na druhém virtuálním stroji s objektem skutečným. Tedy je možné za pomoci tohoto systému zpřístupňovat skutečné objekty oběma směry komunikace.

Díky této možnosti se musíme rozhodnout, kde bude reálný objekt umístěný. Preferované umístění objektu je na straně JVM, protože celé vyhodnocování běží na straně JVM a snažíme se co nejvíc minimalizovat dopady na výkon. To znamená, že v ideálním případě se při žádosti o vyhodnocení X-definice (ať už ve validačním, či konstrukčním režimu) přenese pouze požadavek na spuštění konkrétní funkce s danými argumenty a zpět se přenese pouze výsledek dané operace.

Některé objekty přesto nelze jen tak vytvořit na straně JVM, jedná se například o Streamy, které knihovna dostane jako argument a nemá tušení, o jaký stream se jedná (soubor, standardní vstup, síť, ...). Tedy faktem je, že skutečný objekt je umístěný na straně .NET a my jej musíme javě zprostředkovat pomocí interface *InputStream*, případně *OutputStream*.

Další skupinou tříd a interface je systém reporterů. V X-definicích existuje několik typů reporterů, základním z nich je *ArrayReporter*. Dále jsou tu další reportery například pro logování do souborů, nebo *NullReporter*. Ačkoliv by dávalo smysl reportery implementovat na straně .NET, kam by Java knihovna akorát odesílala jednotlivé reporty, rozhodl jsem se pro jejich ponechání na straně JVM, tedy .NET si pouze řekne o report po provedení požadovaných operací.

Výhodou tohoto přístupu je rychlejší funkčnost samotných X-definic i se specifikovaným reportováním, neboť během vyhodnocování prostřednictvím reporteru nedochází k žádné komunikaci přes TCP. Další výhodou je, že reportery obsahují poměrně rozsáhlou funkcionalitu co se lokalizace týče, a její opětovná implementace na straně .NET by byla velmi pracná.

4.6.1 Statické metody

Další výzvou jsou statické metody, které se musí dát spustit i bez instance objektu. V tomto případě mi přišlo celkem rozumné mít obsluhu statických i nestatických metod v jednom wrapperu, aby nemusely vznikat pro každou třídu s oběma typy metod wrappery dva. Ve finále jsem se rozhodl pro vytvoření jedné instance wrapperu bez přiřazeného skutečného objektu (ukazatel na skutečný objekt je *null*). Tato instance se pak bude starat pouze o obsluhu statických metod.

Na Java straně je to celkem jasné, vznikne další instance implementující *remoteHandlingObject*, která vyřizuje požadavky ze serveru. U C# jsem se rozhodl implementovat do tříd se statickými metodami ještě statické pole typu *Lazy<int>*, ve kterém je uchováno id objektu obsluhující statické funkce této třídy. Jak už zaznělo v minulých sekcích, třída *Lazy* je thread safe a máme tedy zaručeno, že vznikne pouze jeden objekt pro obsluhu statických funkcí na jednu třídu.

Díky tomuto mohou být všechny metody implementovány v jedné třídě jak v .NET knihovně, tak v Java wrapperu dané třídy.

4.7 Párování funkcí

Největším problémem při zvoleném způsobu komunikace je vzájemné párování volaných funkcí. Funkce na klientské straně reprezentuje funkci na skutečném objektu, ale je potřeba jí přiřadit id, aby mohl wrapper správně rozpoznat, kterou funkci má vykonat. Nehledě na to, že situaci ještě komplikují statické metody, které musí volat jiný objekt.

K této části analýzy se vracím až po zahájení implementace, protože neustálé vytváření id funkcí, příslušných metod a kódu, který tyto metody spouští se ukázalo jako neefektivní a zdlouhavé. Víme, které funkce je potřeba vytvořit – všechny veřejné funkce, které se týkají konkrétního zrovna implementované třídy/interface. Pro usnadnění práce je potřeba vytvoření generátoru kódu, který co nejvíce usnadní práci, ale zároveň nebude složitý na napsání.

4.7.1 Generátor kódu

Pro ulehčení práce bude vhodné využít nějaký automatický generátor kódu, který vygeneruje základní rozhraní třídy na obou stranách. Základním rozhraním je v tomto kontextu myšleno vytvoření konstant pro každou funkci, vytvoření základní kostry každé funkce a její zavedení do rozhraní dané třídy – zejména přidání do funkce *RemoteHandlingObject.handleRequest()*.

Funkce se budou generovat na základě jejich veřejného rozhraní, které je možno získat z *.jar* souboru X-definic pomocí příkazu *javap -cp <jar> <classmate>*. Výstupem tohoto příkazu je veřejné rozhraní zadané třídy, včetně typů proměnných a návratových hodnot. Na základě těchto informací se dá vygenerovat jednak kód na straně C#, jehož rozhraní odpovídá v maximální možné míře rozhraní původní knihovny, druhak obslužná funkce v xdef.bridge.

Generátor kódu by neměl být příliš komplikovaný, aby jeho napsání nezabralo více času, než manuální přepsání všech funkcí, tedy výsledný kód nemusí být kompilovatelný bez úprav. To platí zejména pro funkce, kde je argumentem nebo návratovou hodnotou nějaký z datových typů specifických pro knihovnu X-definice.

Vzhledem k tomu, že známe pořadí argumentů ve funkci a jejich typy, lze na základě tohoto pořadí provádět jejich serializaci či deserializaci. To znamená, že pro každý argument je potřeba vygenerovat kód pro jeho serializaci či deserializaci, přičemž pořadí argumentů je stejné na obou stranách a odpovídá pořadí argumentů funkce, na základě které je kód generován.

Takto může generátor kódu správně zpracovat argumenty, které jsou základního typu. V případě neznámých argumentů nebo návratových typů by na daném řádku měla vzniknout alespoň kompilační chyba, aby programátor věděl, kde je potřeba provádět úpravy. Stejně tak lze postupovat v případě návratových hodnot.

V neposlední řadě by měl generátor kódu generovat i obsluhu případných výjimek vyhozených při vykonávání původní funkce, pokud je to možné zavolání původní funkce a komentář obsahující strukturu funkce, na základě které byl kód vygenerován.

Generátor kódu bude vytvořen jako C#/.NET Core WPF aplikace, protože s vývojem klasických okenních aplikací mám nejvíce zkušeností. Nevýhodou je její použitelnost pouze na OS Windows, protože WPF funguje pouze na tomto systému. Tato skutečnost by však neměla představovat problém, cílem aplikace je pouze vygenerovat kód.

Generátoru stačí zatím implementovat pouze případ, kdy je skutečný objekt umístěný na straně JVM. Vyplývá to z předpokladu, že naprostá většina skutečných objektů bude umístěna na straně JVM, tedy generování tímto směrem bude podstatně častější, jak plyne ze sekce 4.6. Opačným směrem bude tedy potřeba psát kód ručně, ale toho není takové množství.

Generátor by si měl také správně poradit se statickými metodami. To v .NET znamená vygenerovat metodu jako statickou a odeslat požadavek na správný objekt – v případě statických metod speciální instanci wrapperu pro práci se statickými metodami. Na straně Javy to znamená u statických metod volání nedelegovat na instanci skutečného objektu.

4.8 Dokumentování kódu

Jedním z nefunkčních požadavků je řádná dokumentace nově vzniklého kódu. X-definice jako také zůstávají zatím nedotčené, není tedy potřeba vytvářet dokumentaci. Rozhraní knihovny zůstává také podobné, jediným zásadním rozdílem je pojmenování funkcí s velkým písmenem na začátku – podle standardních konvencí v jazyce C#. Stačí tedy převést Javadoc stávající knihovny do formátu v C# a upravit podle případných změn.

V sekci 3.1.2 3 jsem se díval na možnosti automatického převodu kódu z Javy do C#. V rámci této analýzy jsem provedl automatický převod kódu pomocí nástroje Sharpen. Takto vzniklý kód byl plný chyb a opravy by byly asi podobně náročné jako ruční přepsání celého kódu. Nicméně základní konstrukce jazyka Java byly převedeny do jazyka C# naprosto perfektně, včetně dokumentace. Tuto dokumentaci lze tedy s drobným množstvím úprav pouze překopírovat do projektu xdef.net, aniž by se musela přepisovat ručně.

Kapitola 5

Implementace

V této kapitole se budeme zabývat implementací jednotlivých částí obou knihoven, problémům a jejich řešením, na které jsem během implementace narazil. Prvním problémem byla nutnost osvěžit si znalost Javy, protože jsem v ní napsal pouze jednu aplikaci v předmětu BI-PJV 5 let zpátky. Mezitím jsem napsal akorát jednu aplikaci v Groovy v rámci týmového projektu a jednu aplikaci v Kotlinu v rámci BI-KOT.

Oba výše zmíněné jazyky sice běží na JVM, ale to je tak jediné co mají s Javou společného. Samotný C# je javě naštěstí více podobný, než oba jazyky, takže naučení osvěžení si základů nebyl zas takový problém.

Během analýzy došlo na základě poznatků o náročnosti celkového převodu k bližší specifikaci cíle. Cílem návrhu je tedy návrh systému, který zajistí fungování validačního a konstrukčního režimu X-definic, zatím bez X-komponent a volání externích funkcí.

Vzhledem k tomu, že struktura tříd a specifikace veřejných metod už je daná dokumentací javy

5.1 Vzájemná interakce

Základem celé knihovny je vzájemná komunikace mezi virtuálními stroji realizovaná pomocí TCP spojení. Předpokládá se provozování knihovny ve více paralelních vláknech, tedy komunikace a vyhodnocování požadavků by nemělo probíhat pouze na jednom vlákně na obou stranách.

5.1.1 Java server

Hlavní částí serveru je vlákno, které vyřizuje příchozí požadavky na navázání spojení a vytváří jednotlivá vlákna pro jednotlivé klienty. Toto vlákno běží v abstraktní třídě *Listener*, které se spouští z hlavního vlákna s parametry, které se získají z argumentů při spuštění. Hlavní vlákno tak může běžet dál a číst ze standardního vstupu pro další příkazy, zatím je implementován pouze příkaz exit, který korektně ukončí aplikaci.

Ze třídy *Listener* zatím dědí jediná třída *TcpListener*, ale to umožní případné snadné rozšíření systému o další metody komunikace, např. *TlsTcpListener*. Při příchozím spojení třída *Listener* vytvoří příslušnou instancí abstraktní třídy *Client*, a spustí komunikaci v dalším vlákně. Vlákna jsou vytvářena pomocí „threadpoolu“, který opakovaně využívá stejná vlákna, pokud jsou již opět k dispozici (klient ukončil spojení).

Třída *Client* je srdcem celého systému. Uchovává odkazy na všechny spravované objekty, rozděluje mezi ně práci a odesílá data. Pro odesílání dat jsou k dispozici 2 metody – *sendRequestWithoutResponse* a *sendRequestWithResponse*. Varianta bez odpovědi pouze odešle příslušná data, zpravidla odpověď na požadavek klienta. V případě varianty čekající na odpověď je registrován objekt *ResponseWaiter*, ve kterém je i *Semaphore*, na jehož uvolnění čeká odesílající vlákno. Jakmile dorazí odpověď druhé strany (pozná se podle id požadavku), je uložena odpověď, a uvolněn příslušný *Semaphore*. Příslušné čekající vlákno je uvolněno a je vrácena uložená odpověď. Tato funkce je v ukázce 5.1. Mimo jiné kontroluje, zda nedošlo k vyvolání výjimky na druhé straně – v takovém případě dojde k vyvolání *RemoteCallException*, zpráva je převzatá ze vzdálené strany.

```
1  public Request sendRequestWithResponse(Request request) {
2      int id = serverRequestId.incrementAndGet();
3      request.setServerRequestId(id);
4      RequestWaiter waiter = new RequestWaiter(id);
5      waitingRequests.put(id, waiter);
6      sendRequestData(request);
7      try {
8          waiter.getSemaphore().acquire();
9      } catch (InterruptedException e) {
10         return null;
11     }
12     if (ResponseException.isException(request)) {
13         RemoteCallException ex =
               ResponseException.getException(request);
14         System.err.println("Remote call exception thrown
               code: " + ex.getErrorCode() + " message: " +
               ex.getMessage());
```

```
15      }
16      return waiter.getResponse();
17  }
```

Listing 5.1: Odeslání požadavku s odpovědí z Javy.

Vzhledem k tomu, že všechno může probíhat paralelně na více vláknech najednou, je potřeba mít nastavené synchronizační mechanismy. Naštěstí v javě jsou pro tyto účely implementované prostředky, které zajistí, že je potřeba zamykat minimální části kódu. Čítače, podle kterých se přiřazuje id objektu/požadavku jsou implementovány pomocí třídy *AtomicInteger*, které zajistí atomicitu prováděných operací a tedy každý požadavek/objekt bude mít unikátní id i v případě zpracovávání více požadavků najednou.

Třída dále obsahuje dvě mapy – jedna mapuje id objektu na objekt samotný, druhá obsahuje požadavky čekající na odpověď podle id požadavku. Obě tyto kolekce jsou implementovány jako *ConcurrentHashMap*, která je thread-safe, tedy je zajištěna konzistence dat i při současné manipulaci více vlákny.

5.1.2 Klient v .NET

Zde je situace velmi podobná javě, neboť komunikace využívá v obou směrech stejný datový formát. Jedná se o klientskou aplikaci a není tedy potřeba třída *Listener*, knihovna si vystačí se třídou *Client*, která naváže komunikaci s již běžícím serverem. O správu klienta se stará singleton třída *XD*, jejíž instance je v C# označená jako *internal* – přístupná pouze ze stejného „assembly“. Tedy je zajištěn přístup ke klientovi odkudkoliv z wrapperu knihovny (např. z konstruktorů objektů) a zároveň k němu nemá přístup uživatel knihovny.

5.1.2.1 Třída *XD*

Třída *XD* se tedy stará o navázání připojení a jeho distribuci mezi jednotlivé třídy. Další její funkcí je poskytnutí přístupu ke statickým třídám na JVM, které jsou implementovány stejně jako nestatické třídy pomocí wrapperů na obou stranách komunikace. Přístup ke statické třídě *XDFactory* je na straně C# proveden jako volání *XD.Factory*.

Výhodou singletonu je, že se inicializuje až po prvním požadavku na vlastnost *Instance*, tedy v tuto chvíli je vhodné vytvořit spojení s JVM podle konfigurace (která je v rámci třídy statická a tedy nespustí inicializaci). Po prvním zavolání vlastnosti (tzv. Property v C#) *Instance* se v závislosti na konfiguraci spustí Java proces (viz. sekce 5.1.2.2) a provede se navázání spojení.

Singleton je realizován pomocí třídy *Lazy*, která mimo jiné zajistí thread-safe vytvoření instance při jejím volání, zkrácená implementace třídy je v ukázce kódu 5.2.

```
1  public sealed class XD
2  {
3    private static Lazy<XD> _instance = new Lazy<XD>(() => new
         XD());
4    private Lazy<XDFactory> _xdFactory;
5    internal Client Client { get; private set; }
6    private XD() {
7      StartJavaBridge(); // Starts Java process if needed
8      StartClient(); // Creates and connects client
9      CreateXdFactoryInitializer(); // Init using newly
           created client
10   }
11   ~XD() {
12     DisconnectClient();
13     ExitProcess();
14   }
15   internal static XD Instance => _instance.Value;
16   public static XDFactory Factory =>
         Instance._xdFactory.Value;
17 }
```

Listing 5.2: Ukázka implementace třídy *XD*.

5.1.2.2 Spuštění procesu xdef.bridge

Spouštění procesu xdef.bridge probíhá automaticky při inicializaci třídy *XD*. V tu chvíli je jasné, že knihovna bude použita a je tedy potřeba aby proces běžel. Před prvním zavoláním nějaké funkce X-definic se dají pomocí statických vlastností (properties) třídy *XD* nastavit možnosti spuštění procesu. Při spouštění je v současné době možné nastavit tyto parametry:

- **StartProcess (bool)** – Zda se má proces spouštět, nebo se o jeho puštění již postaral uživatel knihovny. Zejména užitečné při ladění, kdy je možné mít debugger takto připojený k oběma běžícím aplikacím.
- **Port (int)** – Port na kterém běží TCP komunikace. Výchozí hodnota je 42268. Pokud se spouští proces, číslo portu je předáno jako argument při spuštění.
- **Hostname (string)** – Hostname, na kterém běží vzdálená strana xdef.bridge. Výchozí hodnotou je „localhost“.
- **JavaExePath (string)** – Cesta k binárnímu souboru Java. Výchozí hodnota je *null* – v takovém případě je zavolání pouze příkaz *java*, a musí být v proměnné *PATH*.

Parametry nutné pro spuštění xdef.bridge jsou tedy předány pomocí argumentů při startu. Naopak ve chvíli, kdy je server připraven přijímat spojení, vypíše na standardní výstup řádek „Listening for connections.“. C# knihovna tento první řádek načte a tak pozná, že server je připraven a může tedy proběhnout navázání spojení. X-definice dále obsahují několik funkcí, které vypisují data na standardní výstup, například pro pomoc s laděním. C# aplikace tedy začne zpracovávat standardní výstup xdef.bridge pomocí metody *BeginOutputReadLine*, která zajistí, že při vypsání řádku je v .NET vyvolána událost, na kterou je navázána funkce, která tento výstup překopíruje do výstupu samotné .NET aplikace. Tedy standardní výstup je přístupný i z xdef.net.

5.1.2.3 Třída Client

Poté, co je navázáno spojení už se o hlavní komunikaci stará abstraktní třída *Client*. Tedy funguje skoro stejně, jako stejnojmenná třída v Javě. Uchovává si odkazy na lokální objekty zpřístupněné pomocí wrapperu Java protistraně. Vedle toho odesílá a přijímá data, přičemž musí zachovat bezpečnost dat při souběžném přístupu více vláken – thread safety.

Klient naslouchá příchozím požadavkům ve vlastním vlákně. Toto vlákno běží po celou dobu běhu programu a zpracovává příchozí dotazy. Pokud se jedná o odpověď, tak jen odpověď uloží a odblokuje příslušné čekající vlákno. Odblokování vlákna je stejné jako v případě Javy, tedy pomocí semafóru. Pro implementaci semafóru jsem v tomto případě použil třídu *SemaphoreSlim*, což je osekaná verze normálního semafóru, nicméně pro potřeby knihovny je plně dostačující. Čekání na odpověď a odblokování vlákna poté, co odpověď dorazila se provádí pomocí jeho metod *Wait* a *Release()*.

Pokud dorazil požadavek ke zpracování, je zpracován jako paralelní úloha pomocí task systému v C#. Tento sytém zadané úlohy rozděluje mezi thready v interním threadpoolu a programátor se tedy nemusí starat. Vyhodnocování odpovědí na požadavky probíhá mimo tyto úlohy, aby nedošlo k deadlocku v případě, že nám v threadpoolu nezůstanou volná vlákna pro uložení odpovědi a odblokování jednoho z čekajících vláken.

Dalším problémem při implementaci vlákna, které přijímá požadavky ke zpracování je, že program končí ve chvíli, kdy jsou ukončena všechna vlákna. Toto vlákno běží pořád a není jisté, zda bude korektně ukončeno, k ukončení hlavního vlákna může dojít kdykoliv a toto naše vlákno by tak blokovalo ukončení celé aplikace, což je chování, kterému se chceme vyvarovat. Tento problém je naštěstí snadno řešitelný, jelikož vlákno stačí označit jako běžící na pozadí (background thread), které se poslušně ukončí s hlavním vláknem.

Řešení samotné komunikace a thread safety je velmi podobné jako v Javě, jelikož v C# také existují způsoby atomické inkrementace hodnot a thread safe kolekce. Pro uchování požadavků čekajících na odpověď a lokálních objektů, na které se může dotazovat vzdálená strana je použita kolekce *ConcurrentDictionary*. Atomická inkrementace proměnných na rozdíl od Javy nevyžaduje vytvoření speciálního typu *AtomicInteger*, ale stačí použít volání *Interlocked.Increment*, kterému je referencí předána požadovaná hodnota k inkrementaci. Výsledkem tohoto volání je potom hodnota navýšená o 1.

Samotné vytváření úloh ke zpracování – tedy příchozích požadavků, pokud se nejedná o odpovědi je řešeno vestavěným task systémem. Vytvoření nové úlohy tak probíhá pomocí volání *Task.Factory.StartNew*. Není tedy potřeba udržovat vlastní threadpool.

5.2 Endianita

Po napsání obou stran komunikace jsem se pustil do prvního testování funkčnosti komunikace, nicméně se ukázalo, že přijaté požadavky na obou stranách obsahují nesmyslné hodnoty v numerických polích. Tedy systém vůbec nefungoval a po chvíli zkoumání se ukázalo, že Java, nezávisle na platformě, pracuje s big endianem. Oproti tomu .NET se drží endianity dané platformy – což na mém x86 systému představuje little endian.

Problém jsem se rozhodl řešit na straně .NET přehazováním pořadí bajtů tak, aby komunikace probíhala v big endianu. Pro tento způsob jsem se rozhodl s přihlédnutím k tomu, že big endianu se také říká network byte order. [25] Problémem je, že podle dokumentace knihovna .NET vždy zapisuje data v little endianu, musel jsem tedy dopsat patřičné pomocné třídy, které se starají o korektní zápis v big endianu. Vznikly dvě nové třídy, které přetěžují operátory knihovních tříd *BinaryReader*, *BinaryWriter* a *BitConverter*.

Nově vzniklé třídy se jmenují *BigEndianBinaryReader*, *BigEndianBinaryWriter* a *BigEndianBitConverter*. Dále vznikla třída *BigEndianDataBuilder*, která slouží k ulehčení serializace seznamu argumentů jednotlivých metod. Tuto třídu využívá i kód generovaný pomocí generátoru kódu.

5.3 Generátor kódu

V této sekci se budeme věnovat vzniklému generátoru kódu, který generuje C# kód na základě popisu veřejného rozhraní Java třídy. K tomuto kódu generuje odpovídající Java zdrojový kód. Oba tyto kódy pak spolu komunikují a na programátorovi už je jenom vyřešit chyby, případně vygenerované funkce

dočasně zakomentovat – to pokud nejsou potřeba k dosažení stanovených cílů. Takto jsou třeba zakomentované všechny metody týkající se X-komponent.

Generátor kódu umí bez problému vygenerovat metody, které využívají pouze primitivní typy, případně typy které jsou zanesené ve slovníku generátoru. Tedy pokud se jedná o gettery a settery základních typů, stačí k nim pouze dopsat dokumentaci.

Funkce, které mají více overloadů (přetížení) je nutné detekovat a očíslovat jejich názvy. Všechny funkce na straně Javy mají totiž stejné rozhraní, kdy na vstupu je jediný argument typu *BinaryDataReader* a výstupem funkce je návratová hodnota *Response*. Je tedy potřeba vygenerovat unikátní názvy.

Toho je v případě implementovaného generátoru kódu docíleno pomocí přidaného čísla za název původní metody, pokud je to nutné. Stejný úkon je proveden i u vygenerovaných funkčních konstant. Tedy pro první overload funkce *xcreate* je výsledkem generování Java funkce *xreate1* a příslušná konstanta *FUNCTION_XCREATE_1*.

Výstupem generátoru při generování funkce *exists* objektu *XDPool* je následující kód:

```
1  // autogenerated method
2  // public abstract boolean exists(java.lang.String);
3  public bool Exists(string arg0)
4  {
5      using (var builder = new BigEndianDataBuilder())
6      {
7          // Serialize args here
8          builder.Add(arg0);
9          var res = SendRequestWithResponse(new
               Request(FUNCTION_EXISTS, builder.Build(),
               ObjectId));
10         using (var reader = res.Reader)
11         {
12             return reader.ReadBoolean();
13         }
14     }
15 }
```

Listing 5.3: Ukázka vygenerovaného kódu - C#.

Jedná se o funkci, která ma na vstupu i výstupu pouze primitivní typy a řetězce, generátor tedy dokáže vygenerovat kód, který se obejde bez dalších úprav – tedy je jen potřeba refaktorovat názvy argumentů a napsat dokumentaci.

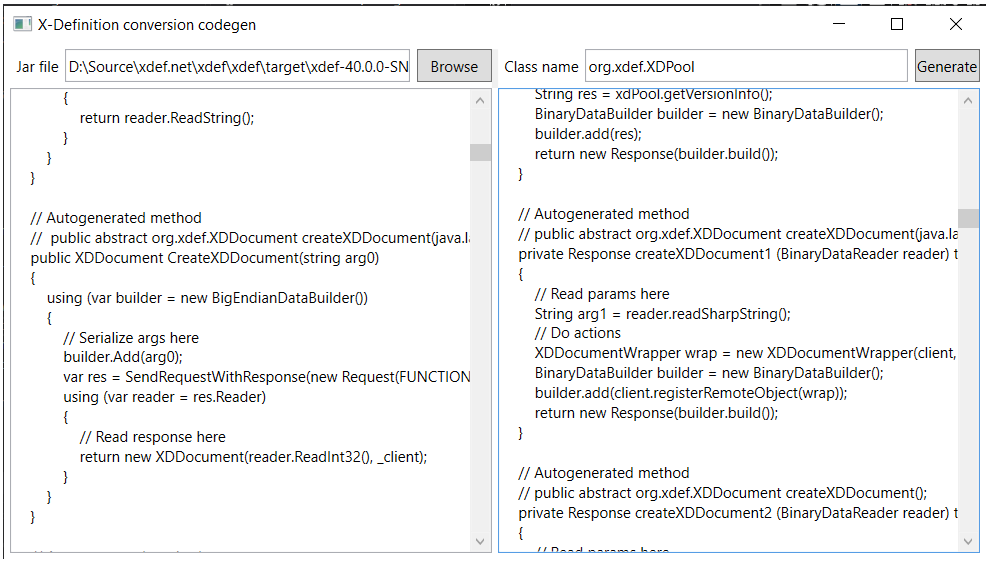
Kód stejné funkce, vygenerované v javě vypadá následovně:

```
1  // Autogenerated method
2  // public abstract boolean exists(java.lang.String);
3  private Response exists (BinaryDataReader reader) throws
       IOException
4  {
5      // Read params here
6      String arg1 = reader.readSharpString();
7      // Do actions
8      boolean res = xdPool.exists(arg1);
9      BinaryDataBuilder builder = new BinaryDataBuilder();
10     builder.add(res);
11     return new Response(builder.build());
12 }
```

Listing 5.4: Ukázka vygenerovaného kódu - Java.

Jak je vidět, kód na obou stranách využívá pomocné třídy, které značně usnadňují práci při serializaci a deserializaci dat. V případě nového datového typu tak mnohdy stačí pouze přidat příslušný overload funkce add.

Jako zdroj dat slouží volání příkazu *javap*, který je následně zpracován kombinací regexu (regulárního výrazu) a C# kódu. Uživatelské rozhraní generátoru je velmi minimalistické a bez validace vstupů. Ukázka vzhledu tohoto rozhraní je na obrázku 5.1.



Obrázek 5.1: Uživatelské rozhraní generátoru kódu.

5.4 Streamy

Tato sekce se zabývá implementací streamů (proudů), které lze použít jako jeden z typů zdrojů při volání funkcí frameworku X-definic. S jejich pomocí je možné předat do JVM data z mnoha různých zdrojů, a jejich implementace tedy představuje celkem zásadní součást systému – tedy alespoň podle mého úsudku.

Problémem, který jsem řešil je nevyhnutelný dopad na výkon procesoru X-definic ve srovnání třeba s voláním funkce, která pouze předá název souboru, který je poté načten s JVM a vyhneme se tak zbytečné komunikaci mezi oběma runtimy (běhovými prostředími). Tento fakt bude vhodné zanést do dokumentace a doporučit třeba uložení požadovaných dat do dočasného souboru, pokud je to možné. Skutečný dopad na výkon zjistíme během testování.

Reálný objekt je na straně .NET, tedy zde je implementovaný jeho wrapper, který odpovídá na jednotlivé požadavky z JVM. Na straně JVM je potom objekt, který je napojený na tento wrapper a objekty implementující rozhraní *InputStream* a *OutputStream*. Výhodou je, že nám stačí pouze jeden wrapper, jelikož C# nemá rozdílná rozhraní pro vstupní a výstupní stream.

Abychom zabránili neustálé TCP komunikaci, je potřeba vytvořit si lokální buffer a data načítat po větších blocích (výchozí velikost mé implementace je 16KB). To je vhodné zejména v případech, kdy knihovna načítá vstup nebo zapisuje výstup po jednom znaku.

Mnou implementované třídy *RemoteInputStream* a *RemoteOutputStream* obě využívají stejný wrapper vzdáleného stream objektu. Pro jejich implementaci jsem použil zdrojový kód tříd *BufferedInputStream* a *BufferedOutputStream* z OpenJDK8 a upravil jej pro vlastní potřeby, tedy aby čtení/zápis prováděly pomocí našeho wrapper objektu. Pro správné fungování těchto tříd bylo potřeba implementovat pouze čtyři funkce ve wrapper objektu – *available*, *close*, *read*, *write*.

5.5 Dokumentace

Jedním z nefunkčních požadavků je vytvoření řádné dokumentace pro novou knihovnu. To je zajištěno pomocí dokumentace jednotlivých veřejných funkcí v C# kódu. Při psaní dokumentace byl využit kód vygenerovaný nástrojem Sharpen, neboť veřejné rozhraní proxy objektu z velké části odpovídá rozhraní objektu reálného.

Takto napsaná dokumentace se poté programátorovi zobrazuje přímo v našeptávači, případně je možné z ní nechat vygenerovat dokumentaci v podobě HTML stránky za pomoci nástroje Doxygen.

5.6 NuGet

Projekt knihovny xdef.net je připraven pro zabalení do balíčku NuGet. Tento balíček pak může být nahrán do centrálního repositáře balíčků. Odsud může být přidán do libovolného projektu .NET kompatibilního s .NET Standard 2.0 a používán pro různé projekty. Aby to bylo možné, je potřeba nejprve registrovat účet na tomto centrálním repositáři. Veškerý zdrojový kód je také v dispozici na `https://github.com/adameste/xdef.net` pod licencí Apache 2.0.

5.7 Srovnání kódu

Na závěr kapitoly přikládám srovnání použití knihovny v Javě a ekvivalentní kód v C#. Ukázka zahrnuje kompilaci zdrojového kódu X-definice a její validaci za použití reporteru. Zdrojový kód v javě a C# můžeme srovnat v následujících ukázkách kódu:

```
1 XDPool pool = XDFactory.compileXD(null,
2   new File("X-definition file path"));
3 XDDocument doc = pool.createXDDocument();
4 ArrayReporter arrayReporter = new ArrayReporter();
5 doc.xparse(new File("xml file path"), arrayReporter);
6 if (arrayReporter.errors()) {
7     // Chyba
8 }
```

Listing 5.5: Ukázka použití knihovny v Javě.

```
1 var pool = XD.Factory.CompileXD(null,
2   new FilePath("X-definition file path"));
3 var doc = pool.CreateXDDocument();
4 var arrayReporter = new ArrayReporter();
5 doc.Xparse(new FilePath("xml file path"), arrayReporter);
6 if (arrayReporter.Errors)
7 {
8     // Chyba
9 }
```

Listing 5.6: Ukázka použití knihovny v C#.

Kapitola 6

Testování

Tato kapitola je věnována testování výsledné knihovny z hlediska funkcionality ale především z hlediska výkonu.

6.1 Testování funkcionality

Testování functionality je řešeno pomocí VS (Visual Studio) projektu s unit testy pomocí frameworku MSTest. Testovací projekt je napsaný v .NET Core 3.1 – tedy nejnovější aktuálně dostupná verze. Testy jsou napsané jako black box (černá skříňka) testy, protože jak samotné funkce fungují nás moc nezajímá, zajímá nás pouze, že vrací správné výsledky a neházejí chyby. Testování samotného připojení není podstatné, to bude důkladně prověřeno všemi ostatními testy.

V unit testech jsem zvolil strukturu stejnou jako v projektu, tedy jedna testovací třída odpovídá jedné třídě testované. Jedna testovací metoda může testovat více funkcí v knihovně – třeba getter a setter najednou, ale může být i více testovacích metod pro jednu knihovní funkci.

Unit testy využívají možnosti nespouštět zabudovaný xdef.bridge, ale připojují se k již běžící instanci na lokálním stroji. Je to kvůli možnosti rychlých úprav na obou stranách knihovny a možnosti jejich ladění (debugger je připojen jak k JVM, tak k .NET aplikaci).

Během testování jsem narazil na chybu ve starší knihovně, kdy se varianta metody *XDFactory.xcreate* pokouší přetypovat v parametru předaný interface *ReporterWriter* na *ArrayReporter*, což skončí chybou, protože komunikační knihovna používá jinou implementaci toho interface, tedy ji nelze takto přetypovat.

Také bylo otestováno, že všechny testy prochází i na platformě s OS Linux, v tomto případě pomocí WSL, protože nemám Linux nainstalovaný na žádném počítači. Testování proběhlo na distribuci Ubuntu 18.4 LTS a všechny testy prošly úspěšně, stejně jako na OS Windows.

6.2 Testování výkonu

V této sekci budeme testovat rychlost .NET knihovny při sekvenčním zpracování X-definic. Nejprve bude rychlost testována na malých vstupních datech (jednotky KB), což by měl být nejhorší možný případ, protože část času strávená komunikací CLR a JVM bude vetší, než případě větších vstupních dat. Následně proběhne testování, kdy na vstupu budou větší XML soubory (jednotky MB), aby se otestovala rychlost v lepším případě. Předpokladem je, že u větších dat bude více času věnováno samotnému výpočtu místo komunikace.

Dále budeme testovat výkon při paralelním volání funkcí knihovny a budeme měřit paralelní zrychlení pro různý počet vláken. Tím také zjistíme, jaký dopad na výkon mají implementované synchronizační mechanismy, protože všechna vlákna používají pro komunikaci stejný socket.

Všechny testy jsou provedeny na počítači s procesorem AMD Ryzen 3900X s 32GB paměti. Procesor má 12 jader schopných najedou zpracovat až 24 vláken, nicméně čím méně jader je využito, tím vyšší frekvence dosahují. Při použití jednoho jádra je dosažená frekvence asi 4,5GHz, při vytížení všech jsou to necelé 4GHz, což se odrazí na výsledcích testování. Všechny testy probíhají na knihovně zkompilované v režimu release bez připojeného debuggeru.

Všechny výsledky jsou zprůměrovaný čas běhu pěti měření. Jako srovnání bude sloužit čas potřebný k výpočtu stejné úlohy při použití knihovny X-definic pouze v Javě. K měření času budou použity vestavěné nástroje daného programovacího jazyku. Jednou z věcí, které je potřeba vzít v úvahu je nutnost knihovny .NET knihovny navázat TCP spojení a JIT kompilace kódu při jeho prvním spuštění. Oba tyto faktory eliminuji před každým testem spuštěním dané úlohy jednou před zahájením samotného testování.

6.2.1 Testování rychlosti na malých datech

V tomto testu je provedeno je simulováno a měřeno vytvoření dokumentu z objektu *XDPool*. Dále je vytvořen objekt typu *ArrayReporter*, který je použit pro validaci XML souboru, a je zkontrolováno, zda nejsou v reportu žádné chyby. To vše je provedeno několikrát (parametr **n**), výsledky jsou uvedeny

v tabulce 6.1. V tabulce je také k vidění údaj o rychlosti .NET implementace oproti použití původní Java knihovny v procentech. Názvy jednotlivých sloupců byly zkráceny, aby se vešly na stránku následovně N_{typ} znamená .NET, J_{typ} je výsledek pro Javu. Všechny rychlosti jsou uvedeny v sekundách zaokrouhlených na dvě desetinná místa.

Jako vstup je použita X-definice obsahující model *World* obsahující více modelů *Family* – jejich počet se odvíjí od požadované velikosti souboru. Na vstupu tohoto testu je soubor s velikostí zhruba 1KB a pouze jedním objektem *Family*.

n	$\mathbf{N_{avg}}$	$\mathbf{N_{min}}$	$\mathbf{N_{max}}$	$\mathbf{J_{avg}}$	$\mathbf{J_{min}}$	$\mathbf{J_{max}}$	**Rychlost**
100	0,14	0,13	0,14	0,07	0,05	0,12	200,00%
1000	1,48	1,33	2,04	0,33	0,25	0,54	448,48%
10000	16,04	15,02	17,23	2,45	2,33	2,91	654,69%

Tabulka 6.1: Tabulka naměřených rychlostí pro malá data.

Z výsledků je patrné, že pro takto malé definice dochází k velkému zpomalení oproti původní knihovně. K většímu zpomalení u vyššího počtu výpočtů dochází z důvodu nutnosti správy více objektů, jejich uvolňování garbage collectorem a nutnost neustálých úprav tabulky objektů.

6.2.2 Testování rychlosti na velkých datech

Testování probíhalo stejně jako v případě malých dat, jen bylo potřeba upravit počty provedených validací kvůli nižší rychlosti. Všechny slupce tabulky jsou stejné jako v případě malých dat. Výsledky jsou v tabulce 6.2. Velikost vstupního XML souboru je zhruba 5MB.

n	$\mathbf{N_{avg}}$	$\mathbf{N_{min}}$	$\mathbf{N_{max}}$	$\mathbf{J_{avg}}$	$\mathbf{J_{min}}$	$\mathbf{J_{max}}$	**Rychlost**
10	4,65	4,17	5,17	2,43	2,29	2,53	191,36%
100	48,1	47,94	48,25	22,96	22,68	23,58	209,49%
300	144,3	142,07	15,85	69,61	69,32	39,83	207,30%

Tabulka 6.2: Tabulka naměřených rychlostí pro velká data.

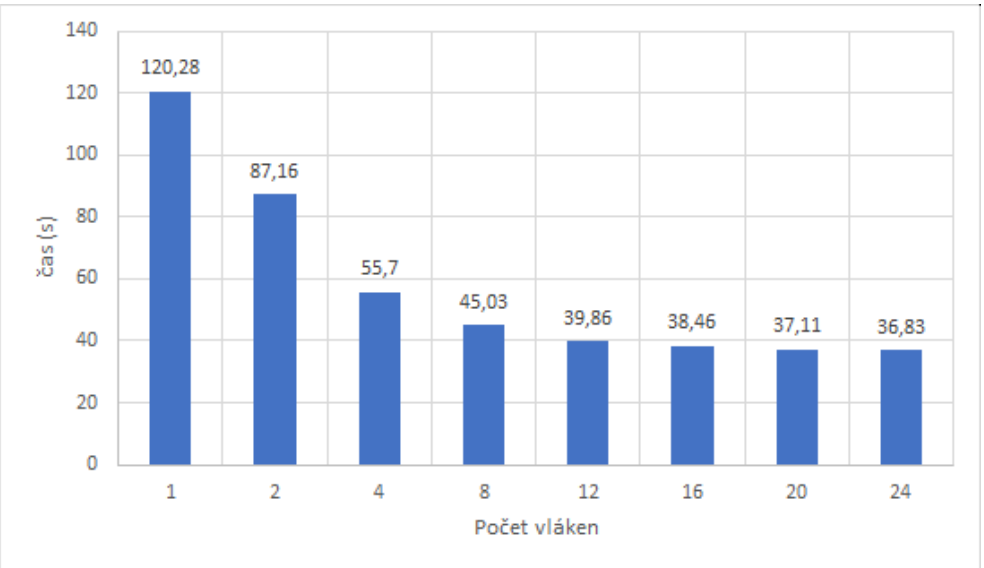
Tyto výsledky již vypadají pro .NET knihovnu příznivěji, nedochází k tolika alokacím a uvolňování, nicméně funkce pro validaci stále vrací třídu *Element* reprezentující zpracovaná data, u které probíhá její převod na třídu *XElement*, aby se s ní dalo pracovat v .NET.

6.2.3 Testování rychlosti paralelního zpracování

V této sekci jsou výsledky testování knihovny při běhu na více vláknech. Na vstupu je XML soubor střední velikosti – 117KB a jeho vyhodnocení se provádí 10000×. Tedy $n = 10000$ a počet souběžně pracujících vláken je v tabulce. Výsledek je uveden v sekundách a jedná se o průměr pěti měření. Vedle času jsou v tabulce dopočítané údaje $S(n, p)$ – paralelní zrychlení a $E(n, p)$ – paralelní efektivnost. Naměřené údaje jsou v tabulce 6.3 a průměrné časy ve formě grafu jsou na obrázku 6.1.

p	$\mathbf{T(n,p)_{avg}}$	$\mathbf{T(n,p)_{min}}$	$\mathbf{T(n,p)_{max}}$	**S(n, p)**	**E(n, p)**
1	120,28	118,92	121,3	1	1
2	87,16	84,84	88,72	1,38	0,69
4	55,7	55,16	56,41	2,16	0,54
8	45,03	44,46	45,48	2,67	0,33
12	39,86	39,38	40,25	3,02	0,25
16	38,46	37,58	39,23	3,13	0,20
20	37,11	36,51	37,7	3,24	0,16
24	36,83	36,2	37,11	3,27	0,14

Tabulka 6.3: Tabulka výsledků paralelního běhu knihovny.



Obrázek 6.1: Graf času výpočtu v závislosti na počtu vláken.

Jak je vidět z tabulky, tak výkon knihovny v C# se příliš dobře neškáluje s rostoucím počtem vláken, pravděpodobně díky přetížení synchronizačních

mechanismů nebo socketu. Další variantou je, že nám prostě došla vlákna přiřazená pro zpracovávání požadavků a tedy některé požadavky musely čekat ve frontě na uvolnění některého z nich. Pozitivní je, že nedošlo k žádnému deadlocku nebo pádu knihovny. Všechny paralelní testy běžely postupně na stejné instanci xdef.bridge – tedy knihovna .NET se vždy připojila jako nový klient.

Spotřeba paměti nebyla měřena, ale ve správci úloh si Java proces zabral necelých 5GB paměti, které nikdy nepřekročil ani při opakování testů. Nicméně spotřeba paměti ve správci úloh se nedá považovat za průkaznou, jelikož se jedná o paměť alokovanou v systému procesem JVM. To znamená, že skutečnou velikost alokované paměti zná pouze JVM.

6.2.4 Testování rychlosti streamovaných dat

V sekci 5.4 věnované streamům bylo zmíněno, že přenos dat mezi knihovnami tímto způsobem bude nejspíše pomalejší, než například předání názvu souboru. Abych tuto hypotézu potvrdil, rozhodl jsem se rozdíl ve výkonu funkce používající stream srovnat s výkonem funkce předávajícím název souboru.

Parametry testování jsou stejné jako u předchozího testování. Výsledek je tedy průměrnou hodnotou pěti měření a jako data je použit 5MB velký soubor stejný jako během testování velkými daty. Měřím rychlost, za jakou se tento soubor zpracuje 50×. Při předání dat názvem souboru operace zabere v průměru **9,91** sekundy. Stejná operace za použití streamů zabere v průměru **16,37** sekundy. Tedy můžeme definitivně konstatovat, že použití streamů zpomaluje běh knihovny.

Závěr

Cílem této práce bylo vytvoření verze Java knihovny X-definice pro jazyk C#. Na základě analýzy problému a návrhu sytému vznikly dvě aplikace – xdef.net a xdef.bridge, které vzájemnou komunikací zprostředkovávají rozhraní X-definic v prostředí .NET.

Během analýzy se narazilo na některé zásadní problémy, jejich řešení by bylo mimo přiměřený rozsah této práce. Z tohoto důvodu byly po konzultaci s vedoucím práce cíle blíže specifikovány cíle práce a výsledkem je funkční rozhraní pro validační a konstrukční mód frameworku X-definic v prostředí .NET.

V budoucnu je možné výsledek rozšířit o chybějící funkcionalitu. Vzhledem k tomu, že výsledná komunikace je celkem nezávislá na protokolu TCP/IP, tak se dá uvažovat o přepsání komunikační vrstvy tak, aby využívala JNI, čímž by došlo ke zrychlení výsledku. Další možností budoucího rozvoje je odtržení celého komunikačního systému a generátoru kódu od X-definic a jeho využití jako samostatného nástroje pro zpřístupnění rozhraní Java knihoven v .NET.

Hlavním přínosem této práce je ve výsledku funkční knihovna X-definic v .NET. V neposlední řadě jsou to zkušenosti a znalosti, které jsem získal během analýzy a implementace tohoto řešení, zejména pak bližší seznámení s programovacím jazykem Java, se kterým jsem na začátku neměl příliš zkušeností.

Literatura

[1] jni4net - bridge between Java and .NET. [online], [cit. 11.5.2020]. Dostupné z: http://jni4net.com/index.html

[2] JNBridge: Connect anything Java with anything. NET. [online], [cit. 11.5.2020]. Dostupné z: https://jnbridge.com/

[3] .NET Standard. [online], únor 2020, [cit. 4.5.2020]. Dostupné z: https://docs.microsoft.com/cs-cz/dotnet/standard/net-standard

[4] X-definition 4.0 Language description. [online], [cit. 4.5.2020]. Dostupné z: http://xdef.syntea.cz/tutorial/en/userdoc/xdef-4.0.pdf

[5] Trojan, V.: X-definice by Syntea software group s.r.o. [online], [cit. 4.5.2020]. Dostupné z: https://www.xdefinice.cz/

[6] Melingerová, G.: NÁVRH A IMPLEMENTACE ZPRACOVÁNÍ FORMÁTU JSON VE FRAMEWORKU X-DEFINICE. 2019.

[7] NuGet Gallery | Home. [online], [cit. 4.5.2020]. Dostupné z: https://www.nuget.org/

[8] Štěpán Adámek: *SYSTÉM EVIDENCE ÚČASTNÍKŮ AKCÍ PRO KLUB ČESKÝCH TURISTŮ*. Bakalářská práce, ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE, Praha, 2017.

[9] Haramule, V.: .NET Core – Úvod do platformy. [online], březen 2017, [cit. 4.5.2020]. Dostupné z: https://www.skeleton.cz/net-core-uvod-do-platformy

[10] Vochozka, J.: Značkovací jazyky a XML. [online], prosinec 2000, [cit. 5.5.2020]. Dostupné z: http://webserver.ics.muni.cz/bulletin/articles/201.html

[11] Collins, C.: X-definition 4.0 Language description. [online], březen 2008, [cit. 5.5.2020]. Dostupné z: https://ccollins.wordpress.com/2008/03/03/a-brief-history-of-xml/

[12] X-definition 4.0 Introduction to the Construction Mode. [online], [cit. 5.5.2020]. Dostupné z: http://xdef.syntea.cz/tutorial/en/userdoc/xdef-4.0_construction_mode.pdf

[13] X-component 4.0 User manual. [online], [cit. 5.5.2020]. Dostupné z: http://xdef.syntea.cz/tutorial/en/userdoc/xdef-4.0_X-component.pdf

[14] mehdimo/janett: Java to C#.Net Translator. [online], [cit. 9.5.2020]. Dostupné z: https://github.com/mehdimo/janett

[15] mono/sharpen: Sharpen is an Eclipse plugin created by db4o that allows you to convert your Java project into c#. [online], [cit. 9.5.2020]. Dostupné z: https://github.com/mono/sharpen

[16] The Mono Runtime | Mono. [online], [cit. 1.5.2020]. Dostupné z: https://www.mono-project.com/docs/advanced/runtime/

[17] Mikhalenko, P.: Discover how the Java Native Interface works - TechRepublic. [online], červen 2006, [cit. 11.5.2020]. Dostupné z: https://www.techrepublic.com/article/discover-how-the-java-native-interface-works/

[18] Geerinck, X.: How-To Bind C++ Code with Dotnet Core - Xavier Geerinck - Medium. [online], srpen 2017, [cit. 11.5.2020]. Dostupné z: https://medium.com/@xaviergeerinck/how-to-bind-c-code-with-dotnet-core-157a121c0aa6

[19] Dumbill, E.; Bornstein, N. M.: *Mono a developers notebook*. OReilly, 2004, ISBN 978-0596007928.

[20] IKVM.NET Weblog. [online], duben 2017, [cit. 11.5.2020]. Dostupné z: http://weblog.ikvm.net/

[21] Šavara, P.: Zamboch: How calling from .NET to Java works in jni4net. [online], říjen 2009, [cit. 25.5.2020]. Dostupné z: http://zamboch.blogspot.com/2009/10/how-calling-from-net-to-java-works.html

[22] Adam FREEMAN, J. R.: *Pro LINQ*. Apress, 2010, ISBN 978-1-4302-2654-3.

[23] PYPL PopularitY of Programming Language index. [online], květen 2020, [cit. 22.5.2020]. Dostupné z: http://pypl.github.io/PYPL.html

[24] Which Version of Java Should You Use? | StackChief. [online], březen 2020, [cit. 14.5.2020]. Dostupné z: https://www.stackchief.com/blog/Which%20Version%20of%20Java%20Should%20You%20Use%3F

[25] Steed, A.: Network Byte Order - an overview | ScienceDirect Topics. [online], 2010, [cit. 23.5.2020]. Dostupné z: https://www.sciencedirect.com/topics/computer-science/network-byte-order

Příloha **A**

Seznam použitých zkratek

XML Extensible markup language

JSON JavaScript Object Notation

JAXB Java Architecture for XML Binding

SGML Standardized Generalized Markup Language

HTML HyperText Markup Language

DTD Data Type Definition

CIL Common Intermediate Language

CLI Common Language Infrastructure

CLR - Common Language Runtime

OS Operační Systém

JNI Java Native Interface

JVM Java Virtual Machine

TCP Transmission Control Protocol

COM Component Object Model

GC Garbage collector

TLS Transport Layer Security

LINQ Language Integrated Query

Regex - Regular Expression

JDK - Java Development Kit

VS - Visual Studio

JIT - Just In Time

WSL - Windows Subsystem for Linux

LTS - Long Term Support

Příloha **B**

Obsah přiloženého DVD

- `readme.txt` ..... stručný popis obsahu DVD
- `lib` ..... adresář se spustitelnou formou implementace
  - `xdef.net.dll` ..... knihovna ve formátu .dll
  - `xdef.net.nupkg` ..... knihovna v podobě NuGet balíčku
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF