FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Analýza a redesign architektury frameworku x-definice
Student:	Bc. Filip Šmíd
Vedoucí:	Ing. Jindřich Kocman
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce zimního semestru 2020/21

Pokyny pro vypracování

X-definice je rozsáhlý framework určený k validaci a transformaci strukturovaných dat, jehož součástí může být dynamický přístup k externím zdrojům. Cílem této práce je analyzovat a zdokumentovat stávající architekturu tohoto frameworku, diskutovat a navrhnout pro něj novou koncepčně čistou a dobře rozšiřitelnou architekturu a navrhnout postup přechodu na tuto architekturu.

Postupujte v těchto krocích:

1. seznamte se se stávající implementací frameworku x-definice

2. diskutujte a navrhněte novou - koncepčně čistou a dobře rozšiřitelnou architekturu frameworku se zřetelem na zpracování formátu XML a JSON. Současná specifikace jazyka X-definic nemusí být pro navrženou architekturu 100% závazná.

3. návrh architektury podrobně rozpracujte a dokumentujte

4. navrhněte postup přechodu stávajícího systému na novou architekturu

5. v rámci ověření návrhu zvolte vhodnou část systému, tu transformujte do nové architektury

6. řešení řádně otestujte a zdokumentujte

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 21. února 2019

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Analýza a redesign architektury frameworku X-definice

Bc. Filip Šmíd

Katedra softwarového inženýrství
Vedoucí práce: Ing. Jindřich Kocman

9. května 2019

Poděkování

Rád bych poděkoval vedoucímu práce Ing. Jindřichu Kocmanovi za zasvěcení do tématu, odborné rady a čas strávený při vedení této práce. Také bych rád poděkoval Ing. Michalovi Valentovi, Ph.D. za zprostředkování velmi zajímavého tématu práce. Na závěr bych rád poděkoval mým milým kamarádkám Kristýně a Lucce za jejich ochotu a čas strávený při korektuře práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické 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. § 46 odst. 6 tohoto zákona 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.

V Praze dne 9. května 2019 .....

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2019 Filip Šmíd. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení na předchozí straně, je nezbytný souhlas autora.

Odkaz na tuto práci

Šmíd, Filip. *Analýza a redesign architektury frameworku X-definice*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2019.

Abstrakt

Diplomová práce se zabývá návrhem architektury frameworku X-definice, který slouží ke zpracování strukturovaných dat. Cílem práce je redesign architektury frameworku X-definice a implementace částí nové architektury pro ověření její správnosti. Výsledkem práce je návrh architektury. Navržená architektura popisuje hlavní komponenty systému, jejich rozhraní a základní odpovědnosti. V návrhu architektury je zohledněn požadavek na její rozšiřitelnost. Dílčím výsledkem je i základní implementace systému se všemi navrženými komponentami.

Klíčová slova architektura, JSON, redesign, strukturovaná data, XML, X-definice

Abstract

The diploma thesis deals with the design of X-definice framework, which is used to process structured data. The goal of the thesis is to redesign the X-definice framework architecture and the implementation part of the new architecture to verify its accuracy. The result of the thesis is the design of architecture. The designed architecture describes the main components of the system, their interfaces and basic responsibilities. In the architecture design, the requirement for extensibility is taken into account. The partial result is a basic implementation of the system with all the designed components.

Keywords architecture, JSON, redesign, structured data, XML, X-definice

Obsah

Seznam obrázků

Seznam tabulek

Seznam zdrojových kódů

Úvod

V dnešní době existuje mnoho softwarových systémů, které bychom mohli označit jako modulární. Modulární systém není jeden velký celek, ale skládá se z menších částí (modulů), které mezi sebou komunikují. Během komunikace si mezi sebou vyměňují data, ať už prostřednictvím vlastních formátů, či formátů obecných jako např. XML (Extensible Markup Language) nebo JSON (JavaScript Object Notation). Dává tedy smysl mít technologie, které dokáží tato data validovat nebo je upravovat. Díky tomu můžeme například předejít problémům s nevalidními daty, která můžeme eliminovat, či upravit ihned při jejich přijetí.

Při tvorbě softwarového systému je vhodné postavit jeho návrh na konkrétní architektuře. Výhodou projektu s jasně definovanou architekturu je, že v případě fluktuace lidí na projektech je jejich adaptace na novém projektu snazší, rychlejší a v konečném důsledku tedy i finančně méně náročná.

Diplomová práce je primárně určena pro společnost Syntea Software Group a.s., která je tvůrcem celé technologie X-definice, a měla by ji využít jako základní kámen reimplementace technologie. Nicméně práce může sloužit i jako inspirace při řešení podobného problému.

Práce se zabývá návrhem architektury systému s důrazem na zpracování formátů XML a JSON. Ale už se nezabývá definicí struktury X-definice pro formát JSON.

Cílem teoretické části práce je seznámit se s technologií X-definice, představit a porovnat datové formáty XML a JSON a seznámit se s přístupy k jejich zpracování. Poslední částí cíle je seznámení se s obecnými principy pro návrh softwarové architektury. Cílem analýzy je analyzovat současnou verzi frameworku X-definice, definovat požadavky na nový systém a provést analýzu dostupných technologií, které lze využít ke splnění požadavků. Cílem návrhu je samotný návrh architektury systému splňující požadavky, které jsou na ni kladeny. Cílem implementace je ověření navržené architektury. V rámci splnění cíle by měla být implementována konkrétní část nového systému. Cílem testování je vytvořit sadu testů pro ověření kvality implementace.

Diplomová práce se skládá z osmi kapitol. Kapitoly 1–4 se věnují teoretické části práce a vysvětlení všech pojmů, které jsou v práci použity. Kapitola 5 se věnuje analýze implementace současného řešení. Dále se kapitola věnuje analýze požadavků a dostupných technologií. Tato kapitola spolu s kapitolou 1 pokrývají první bod zadání. Kapitola 6 se věnuje samotnému návrhu architektury a důvodům, které za tímto návrhem stojí. Kapitola pokrývá druhý, třetí a čtvrtý bod zadání. Kapitola 7 se zabývá implementací zvolené části nové architektury za účelem ověření jejího návrhu. Kapitola pokrývá pátý bod zadání. Kapitola 8 se věnuje druhům testů a testování implementované části. Společně s kapitolou 7 pokrývá šestý bod zadání.

Kapitola 1

X-definice

Technologie X-definice je rozsáhlý open-source framework vyvinutý firmou Syntea software group a.s. umožňující validaci a transformaci strukturovaných dat ve formátu XML. Dle [13] bylo hlavní myšlenkou vzniku frameworku sjednocení struktury dat s jejich předpisem pro validaci a dále pak potřeba zpracovávání velkých dokumentů. Na rozdíl od běžně používaných technologií pro popis XML dokumentů, jako je například XML Schéma či Relax NG, je popis dokumentu pomocí X-definice pevně spjatý s popisovanou strukturou a díky tomu se zvyšuje celková přehlednost a srozumitelnost popisu dokumentu. Technologie X-definice umožňuje zpracování dokumentu ve dvou režimech: režim validace a režim konstrukce. Oba režimy lze navzájem kombinovat, což představuje další výhodu oproti dostupným řešením. Jednotlivé režimy se od sebe liší podle toho, kým je řízeno zpracování dat, jestli X-definicí nebo vstupními daty [13]. V následujících částech kapitoly představím základní prvky a koncepty technologie X-definice, které jsou důležité pro pochopení dalších částí práce. V této části čerpám z oficiální dokumentace technologie X-definice verze 3.1 dostupné z [30] a dále pak z přednášky Ing. Kamenického dostupné z [13].

Termín X-definice lze chápat několika možnými způsoby:

1. jako jazyk umožňující popsat strukturu, obsah a zpracování nebo konstrukci XML dokumentů,

2. jako XML dokument strukturou podobný struktuře a datům popisovaného dokumentu,

3. nebo souhrnně jako technologii pro popis a zpracování XML dokumentů.

Z pohledu ad 2 je X-definice dobře strukturovaný (well-formed) dokument, jehož kořenový element je určený značkou (tagem) `<xd:xdef>`[1]. Každá

[1] Pro jména speciálních elementů a atributů X-definice je použit prefix `xd`, který slouží k odlišení prvků náležících do jmenného prostoru `http://www.syntea.cz/xdef/3.1`.

X-definice je určena svým jménem, které je definováno jako hodnota atributu `xd:name` a v případě používání více X-definicí současně, musí být jméno v tomto kontextu jedinečné. Kořenový element X-definice obsahuje modely elementů, definice skupin a deklaraci globálních proměnných a uživatelských metod. V atributu `xd:root` kořenového elementu je definován seznam jmen modelů elementů, oddělených „svislítkem“ (symbol |), které mohou být kořenovým elementem popisovaného XML dokumentu.

```
<?xml version="1.0" encoding="UTF-8" ?>
<xd:def xmlns:xd="http://www.syntea.cz/xdef/3.1"
        xd:name="def-zamestnanec"
        xd:root="Zamestnanec">
    <!-- Model elementu Zamestnanec -->
    <Zamestnanec>
        <Osoba Jmeno="required string(1, 30)"
               Prijmeni="required string(1, 30)"
               DatumNarozeni="required datetime('d.M.yyyy')"
               Plat="optional decimal()"/>
        <Adresa Ulice="required string(1, 250)"
                CisloDomu="required int()"
                Obec="required string(1, 250)"
                Okres="required string(1, 250)"
                PSC="required int(10000, 99999)"/>
        <Poznamka>optional string(1, 1000)</Poznamka>
    </Zamestnanec>
</xd:def>
```

Zdrojový kód 1.1: Ukázka X-definice, upraveno z [30]

1.1 Model elementu

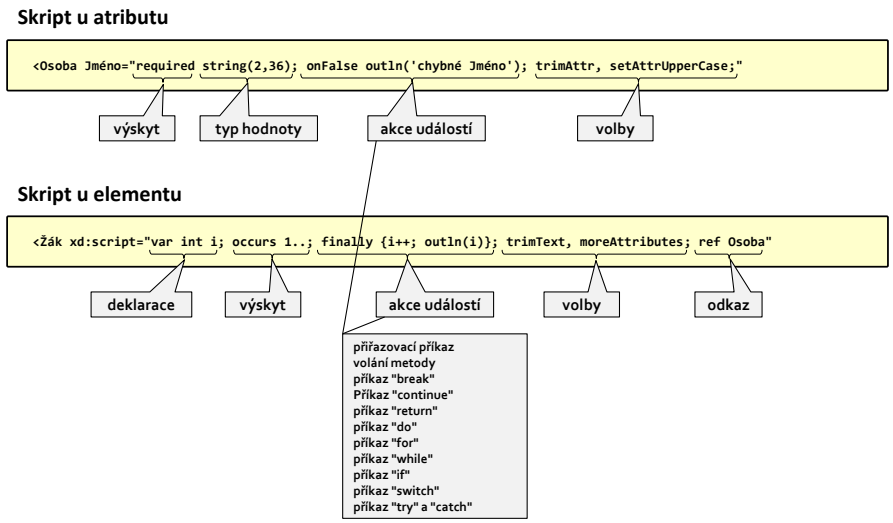
Základním prvkem X-definice je model elementu, který popisuje strukturu XML elementu, jeho atributů a také jeho obsahu, tedy textovou hodnotu a potomky. Model je jasně určen svým jménem, které musí být v rámci jedné X-definice jedinečné. V X-definici se mohou vyskytovat i modely elementů, které nepopisují žádný element ve zpracovávaných datech, ale slouží jako vzor pro ostatní elementy, které na něj mohou odkazovat – nejčastěji se uplatňuje na společné vlastnosti podobných elementů podobně jako koncept dědičnosti v OOP (Objektově orientované paradigma). Pro popis vlastností jednotlivých elementů slouží skript X-definic.

1.2 Skript

Skript X-definic je jazyk syntaxí velmi podobný jazyku Java, který se zapisuje jako textová hodnota atributu, nebo elementu. Z důvodu odlišení pomocných konstruktů jazyka X-definic od samotného popisu struktury elementu je použit pro zápis skriptu elementu pomocný atribut `xd:script`. Skript popisuje vlastnosti dat jako typ, či validační pravidla a dále definuje hodnoty akcí, které jsou prováděny jako reakce na události vyvolané během zpracování dokumentu. Jednotlivé skripty (elementu, atributu a textové hodnoty) nabízejí prakticky stejné možnosti, liší se jen v několika málo odlišnostech. Skript může obsahovat následující části:

- validační sekce,
- akce,
- volby,
- odkaz,
- deklarace proměnných a funkcí.

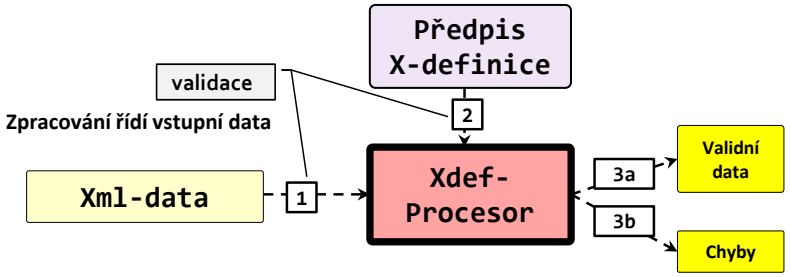
Validační sekce obsahuje pravidla pro určení kvantity, kterou lze zapsat mnoha možnými způsoby viz. tab. C.1 a C.2. Dále pak může obsahovat výraz pro kontrolu typu např. `string(2, 10)`. Dostupné události se liší podle toho, zda se jedná o skript elementu, atributu, či textové hodnoty elementu viz. tab. C.3 a tab. C.4. Schéma skriptu je zobrazeno na obr. 1.1.



Obrázek 1.1: Schéma skriptu X-definic, upraveno z [13]

1.3 Validační mód

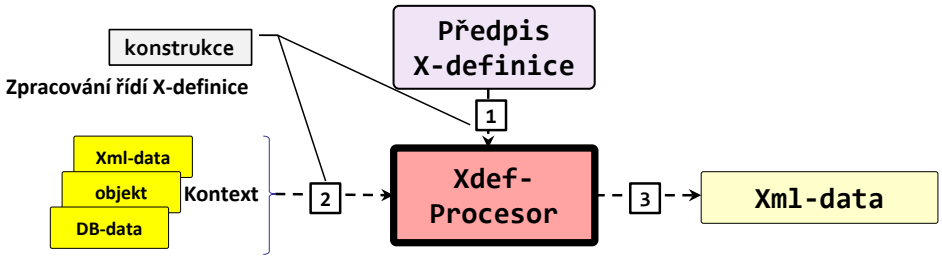
Validační mód slouží k validaci vstupního XML dokumentu podle předpisu definovaného příslušnou X-definicí. Ve validačním módu je zpracování dokumentu, tedy jeho validace, řízena vstupními daty, pro která se v X-definici vyhledávají odpovídající modely. Výstupem validace je, podle předpisu X-definice, validní XML dokument a případně informace o obsažených chybách. Ve validačním módu se nejedná čistě jen o validaci, ale může docházet také ke změně textových hodnot elementů a atributů. Výstupem pak může být částečně pozměněný vstupní XML dokument, např. hodnoty odkazující na nějaký externí číselník, mohou být nahrazeny odpovídající hodnotou z číselníku apod.



Obrázek 1.2: Schéma běhu ve validačním módu, upraveno z [13]

1.4 Konstrukční mód

Konstrukční mód slouží, jak už název napovídá, k vytváření nového XML dokumentu, nebo k transformaci vstupního dokumentu. Na rozdíl od validačního módu je proces zpracování v módu konstrukce řízen příslušnou X-definicí. V tomto případě tedy X-definice definuje strukturu výstupního XML dokumentu. Vzhledem k tomu, že X-definice je dobře strukturovaný (well-formed) dokument, je také výstupní XML dokument dobře strukturovaný (well-formed) dokument. To je jedna z hlavních výhod oproti XSLT (Extensible Stylesheet Language Transformations), které po aplikování transformačních pravidel nezaručuje formát výstupu. Zpracování je řízeno pomocí speciální akce `create`, která na základě vyhodnocení vstupního výrazu vytvoří příslušný výstup, který je následně zpracován procesorem X-definic. Výstupem je tedy dobře strukturovaný (well-formed) dokument, případně i validní dokument, protože oba módy lze kombinovat.



Obrázek 1.3: Schéma běhu v konstrukčním módu, upraveno z [13]

Kapitola 2

Strukturovaná data

V této kapitole se zaměřím na vymezení pojmu *strukturovaná data*, zařazení formátů XML a JSON do kontextu strukturovaných dat a dále pak na popis formátů XML a JSON.

Z pohledu struktury dat můžeme data klasifikovat do třech skupin:

- **Strukturovaná data** mají pevně stanovený datový model, který popisuje jejich strukturu. Taková data se snadno zapisují, ukládají, dotazují a analyzují. Typickým příkladem jsou data v relačních databázích.
- **Polo-strukturovaná data** nemají striktně definovanou strukturu. Nejčastěji jsou organizovaná prostřednictvím značek. Typickým příkladem jsou data popsaná XML či JSON.
- **Nestrukturovaná data** jsou všechna ostatní data, tedy data, která nelze dobře klasifikovat. [1]

Pokud bychom data klasifikovali jen na strukturovaná a nestrukturovaná, pak můžeme na strukturovaná data nahlížet jako na data, která mají nějakou strukturu (ne nutně striktní), a dává smysl s takovými daty pracovat, dotazovat se na ně apod.

2.1 XML

XML je značkovací jazyk, který byl vyvinut a standardizován konzorciem W3C (World Wide Web Consortium). Jedná se o datový formát, který je využíván napříč všemi oblastmi světa IT od komunikačních protokolů jako je například SOAP (Simple Object Access Protocol) či XML-RPC (XML Remote Procedure Call), přes formát na ukládání dat např. XML databáze až po formát na výměnu dat.

XML dokument se skládá z následujících prvků:

- značky (tagy),

- atributy,
- textové hodnoty,
- jmenné prostory a
- pomocné konstrukce jako komentáře, procesní instrukce, entity, CDATA sekce a odkazy na DTD definice [13].

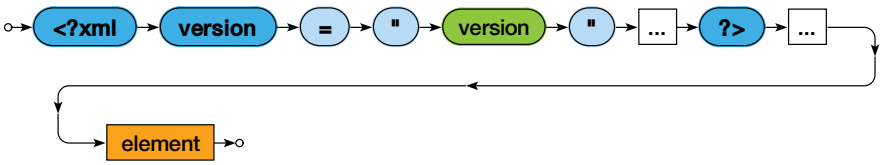
Každý dokument musí obsahovat jeden kořenový element, který může obsahovat libovolný počet dalších elementů, textových hodnot, nebo může být prázdný. Na XML dokument lze nahlížet jako na stromovou strukturu, což pro nás bude důležité v dalších částech práce.

Deklarace XML

Jedná se o speciální procesní instrukci, která se může nacházet na začátku dokumentu a která obsahuje důležité informace o dokumentu. Deklarace může obsahovat tři informace:

- verzi[2] XML,
- použitou znakovou sadu a
- informaci, zda je nutné ke správné interpretaci obsahu použít externích deklarací značkování [2].

Pokud není deklarace v dokumentu uvedena, předpokládá se verze 1.0 a kódování UTF-8 [2, 4]. Schéma deklarace XML je na obr. 2.1. Díky definování použitého kódování dokumentu, můžeme snadno zpracovávat i dokumenty z neznámých zdrojů bez nutnosti zjišťování použitého kódování pro správné zpracování dokumentu.

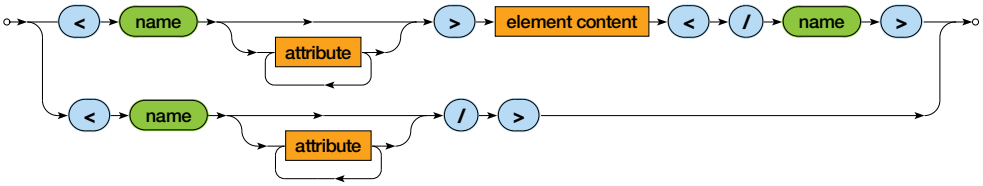


Obrázek 2.1: Schéma deklarace dokumentu XML [28]

[2] Aktuálně existují dvě verze XML a to verze 1.0 a verze 1.1.

Element

Element je základní prvek XML dokumentu. Každý dobře strukturovaný (well-formed) dokument musí obsahovat jeden element (kořenový element). Za součást elementu se považuje vše, co je mezi počátečním a koncovým tagem. Počáteční tag je definován jménem elementu, které je ohraničeno symboly < a >, a koncový tag je definován stejným jménem elementu, které je ale ohraničeno symboly </ a >. Součástí počátečního tagu může být seznam atributů elementu viz. schéma elementu na obr. 2.2.



Obrázek 2.2: Schéma elementu XML [28]

Každý element musí být úplně uzavřen v jiném elementu (s výjimkou kořenového elementu), nelze tedy vytvářet překrývající se elementy viz. ukázka kódu 2.1.

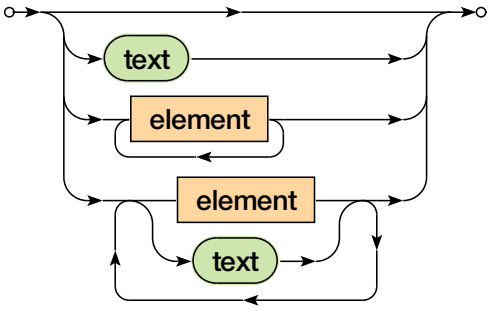
```
<?xml version="1.0" encoding="UTF-8"?>
<root>
<parent>
<child>
</parent>
</child>
</root>
```

Zdrojový kód 2.1: Překrytí elementů

Podle typu obsahu elementu dělíme obsah do tří skupin:

- elementový obsah – element obsahuje pouze další elementy nikoliv text,
- datový obsah – element obsahuje pouze text a
- smíšený obsah – element obsahuje jak další elementy, tak i textové hodnoty.

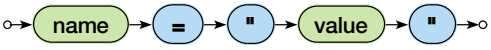
Textový obsah elementu je jasně vymezen tagy elementů. Na obr. 2.3 je schéma smíšeného obsahu elementu, kde textový obsah je rozdělen na dvě části. Nicméně části obsahu nemusí mít mezi sebou jakoukoliv souvislost.



Obrázek 2.3: Schéma smíšeného obsahu XML elementu [28]

Atribut

Atributy poskytují dodatečnou informaci o elementu. Jsou definovány jako součást počátečního tagu elementu. Atribut se skládá z jména atributu a hodnoty atributu, která je uzavřena do uvozovek. Jméno a hodnota atributu jsou od sebe odděleny symbolem rovnítka. Schéma atributu je zobrazeno na obr. 2.4.



Obrázek 2.4: Schéma atributu XML [28]

Jmenné prostory

Jedním ze specifik XML jsou jmenné prostory. Jmenné prostory umožňují rozlišení stejně pojmenovaných elementů a atributů. K identifikaci jmenného prostoru se používá URL (Uniform Resource Locator) adresa. Využívá se faktu, že URL adresa je jedinečná a to zaručuje i jednoznačnou identifikaci jmenného prostoru. Elementy a atributy z různých jmenných prostorů jsou od sebe navzájem odlišeny připojením prefixu k jejich jménu. Prefix slouží jako zástupný řetězec pro identifikátor jmenného prostoru a je od jména oddělen dvojtečkou.

2.2 JSON

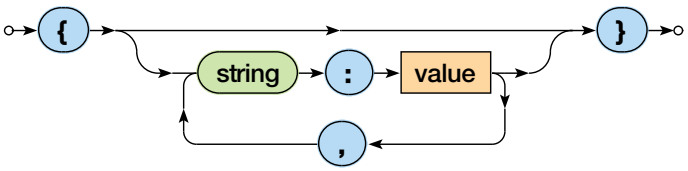
JSON je datový formát pro výměnu dat. Stejně jako u XML se jedná o datový formát, který je dobře strojově zpracovatelný a zároveň i dobře čitelný pro člověka. Na rozdíl od XML nabízí JSON mnohem volnější strukturu a podporuje datové typy. Většina RESTful rozhraní používá JSON jako datový formát pro výměnu dat [15]. Celý formát je založen na dvou základních strukturách:

- neuspořádaná kolekce párů jméno-hodnota a
- uspořádaným polem hodnot [6].

S příchodem nové verze standardu uvedeném v [3], může být kořenem dokumentu libovolná JSON hodnota, zatímco starší standard RFC (Request For Comments) 4627 připouštěl jako kořen dokumentu pouze objekt nebo pole [34]. Stejně jako u XML, lze na formát JSON nahlížet jako na stromovou strukturu. Pokud není explicitně uvedeno, tak v této části o formátu JSON čerpám informace ze standardu formátu dostupného z [3].

Objekt

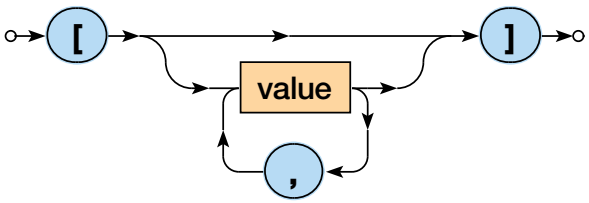
Objekt je neuspořádaná kolekce reprezentovaná symboly složených závorek ({ a }) obsahující libovolný počet párů jméno-hodnota (může být i prázdná), které jsou od sebe odděleny čárkou. Jméno je řetězec a mělo by být v daném objektu unikátní (specifikace formátu striktní unikátnost nevyžaduje, nicméně mnoho aplikací považuje neunikátnost jako chybu [3]). Jméno je od hodnoty odděleno dvojtečkou. Schéma objektu je zobrazeno na obr. 2.5.



Obrázek 2.5: Schéma JSON objektu [29]

Pole

Pole je uspořádaná kolekce hodnot oddělených čárkou a je reprezentované symboly hranatých závorek ([a]). Může obsahovat libovolný počet hodnot, nebo být prázdné. Jednotlivé hodnoty mohou být různých typů. Schéma pole je zobrazeno na obr. 2.6.



Obrázek 2.6: Schéma JSON pole [29]

Řetězec (string)

Řetězec je sekvence Unicode znaků, která je ohraničena dvojitými uvozovkami. Může obsahovat libovolné znaky, ale speciální znaky jako například symboly uvozovek musí být escapovány. Řetězcem je také každé jméno, takže má stejnou strukturu jako řetězec v roli hodnoty.

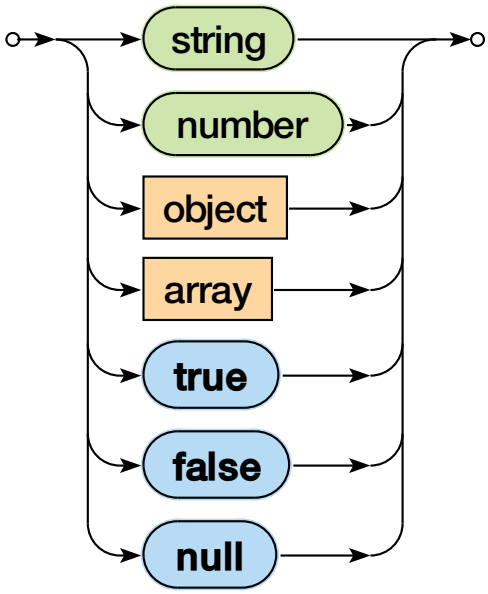
Číslo (number)

Číslem lze reprezentovat jak celou, tak desetinnou hodnotu. Pro zápis číselné hodnoty je používán podobný zápis jako v mnoha programovacích jazycích (např. zápis `10e-5`, který reprezentuje hodnotu 10^{-5} je shodný se zápisem v jazyce C).

Hodnota

Hodnotou mohou být výše zmíněné struktury jako object, pole, číslo a řetězec, a nebo následující tři literály:

- `true`,
- `false` a
- pro neurčenou informaci `null`.



Obrázek 2.7: Schéma JSON hodnoty [28]

2.3 XML vs. JSON

Oba datové formáty mají mnoho společného, ať už z hlediska použití, tak z hlediska jejich vlastností. Jednou z hlavních výhod formátu XML může být jeho vysoká obecnost. I z tohoto důvodu je mnohem snazší převod JSON na XML než obráceně. Za výhody formátu JSON lze považovat jeho volnější strukturu, úspornější zápis, ale hlavně podporu základních datových typů. I z těchto důvodů začíná být JSON jako datový formát na výměnu dat preferovanější [15]. Základní vlastnosti obou formátů jsou shrnuté v tab. 2.1.

Tabulka 2.1: Porovnání formátu XML a JSON

	XML	**JSON**
Čitelný pro člověka	✓	✓
Čitelný pro stroj	✓	✓
Datové typy		✓
Stromová reprezentace	✓	✓
Jmenné prostory	✓	

Kapitola 3

Zpracování dat

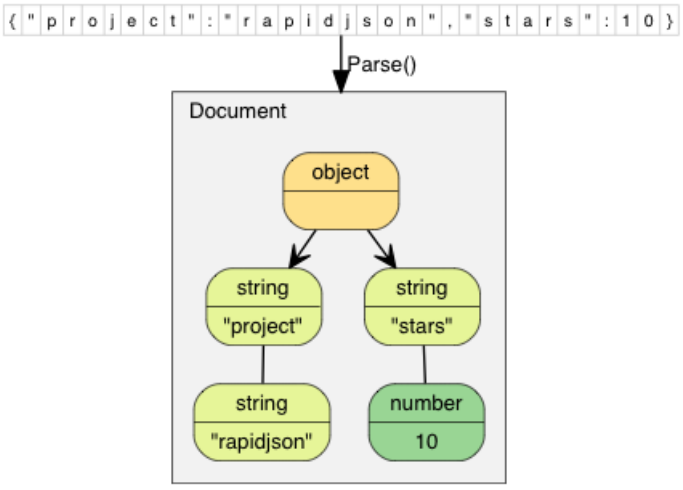
V této kapitole představím možnosti přístupu ke zpracování strukturovaných dat. Pokusím se ukázat jaké výhody a nevýhody mají jednotlivé přístupy a ve kterých situacích se hodí použít který.

První věcí, kterou musíme udělat, než začneme zpracovávat nějaká data, je zvolit si způsob, jak s daty budeme pracovat. Nezáleží na tom, zda budeme zpracovávat data uložená v souboru, či budeme přistupovat k datům skrz webovou službu a podobně. Můžeme si vybrat ze dvou přístupů, kde každý přístup má jak svoje výhody, tak nevýhody a každý z nich se hodí na zpracování jiných dat, či nabízí různé možnosti při jejich následném zpracování. V prvním přístupu provedeme načtení všech dat do operační paměti počítače, kde s nimi následně pracujeme. Pokud má vstupní datový formát jasně definovanou strukturu, můžeme pro něj definovat objektový model, který je jeho reprezentací. Tento první přístup se označuje jako objektový model. V druhém případě se nesnažíme načíst celá vstupní data do paměti, ale čteme je postupně a ihned je zpracováváme. Tento přístup se označuje jako proudový přístup. Oba přístupy lze samozřejmě kombinovat, takže můžeme určitou část dokumentu zpracovat proudově a další část dokumentu načíst do objektového modelu.

3.1 Objektový model

Objektový model je objektová reprezentace vstupních dat. Pro vytvoření objektového modelu načítáme vstupní data a na základě předem známých identifikátorů, které jsou typické pro daný vstupní formát, z nich sestavujeme objektový model. Tento proces nazýváme parsování dat a objekt, který provádí parsování, označujeme jako parser. Například pro formáty založené na stromové struktuře bude jejich objektová reprezentace konkrétní strom. Hlavní výhodou tohoto přístupu je, že data v paměti můžeme libovolně měnit, opakovaně je procházet apod. Hlavní nevýhodou je paměťová náročnost objektového modelu. Nejenže nejsme schopni načíst do paměti libovolně velká

data, ale při vytvoření objektové reprezentace může vzrůst použití paměti až několikanásobně. Příklad objektového modelu pro jednoduchý dokument ve formátu JSON je zobrazen na obr. 3.1.



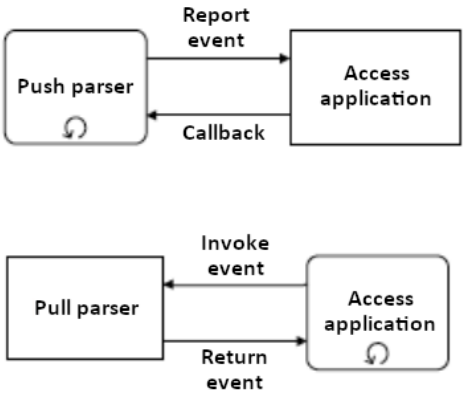
Obrázek 3.1: Příklad objektového modelu JSON dokumentu, upraveno z [33]

3.2 Proudové zpracování

Zpracování vstupních dat probíhá prakticky totožně jako v předchozím případě. Načítáme data a podle známých identifikátorů provádíme jejich parsování. Při proudovém zpracování na rozdíl od objektového přístupu není výsledkem parsování objektový model celého dokumentu, ale jen konkrétní část dokumentu. Na uživateli parseru pak je, jak s touto informací naloží. Například na získané části provedeme určité operace jako třeba změnu dat a na závěr provedeme její zápis na výstup. Po dokončení zpracování dané části si buď zažádáme o další část, nebo ji automaticky dostaneme. Z toho vyplývá, že existují dva přístupy k proudovému zpracování, které se liší v tom, kdo zpracování vstupních dat řídí. Schéma obou přístupů je zobrazeno na obr. 3.2.

3.2.1 Push model

Push model neboli model založený na událostech (event-based model) je přístup k proudovému zpracování, kde zpracování řídí volaný. Volaný načítá vstupní data a po přečtení významné části dat informuje volajícího. Volající na tuto událost zareaguje předem určenou akcí. Tento model lze použít pouze pro čtení, nikoliv však pro zápis. Další nevýhodou je, že tento model nepodporuje kombinování objektového a proudového zpracování.



Obrázek 3.2: Push vs. pull model, upraveno z [20]

3.2.2 Pull model

V pull modelu řídí zpracování volající. Pull model se dělí na dva přístupy, které se liší v tom, jaká informace je předávána volajícímu. Pro první přístup se můžeme setkat s označením jako model založený na tokenech (token-based model). Tokenem je zde označen významný identifikátor ve vstupním formátu (např. pro JSON to jsou symboly složených závorek apod.) V tomto přístupu je volajícímu předán pouze tento token. Podle druhu tokenu se pak volající rozhoduje, jaká data si na parseru vyžádá. Pro druhý přístup se můžeme setkat, stejně jako u předchozího modelu, s označením model založený na událostech (event-based model). Volajícímu je předána událost (event). Událost v sobě nese nejen informaci o svém typu, ale také data, která dané události přísluší. Například událost reprezentující začátek XML elementu v sobě obsahuje informaci o jeho jménu.

Kapitola 4

Softwarová architektura

V této kapitole zadefinuji pojem *softwarová architektura* a představím principy a doporučení, kterými je vhodné se řídit při jejím návrhu.

Existuje celá řada definic pojmu softwarová architektura. Jednou z definic může například být: *„Architektura je základní organizace systému, daná jeho komponentami, vzájemnými vztahy těchto komponent a jejich vztahy k okolnímu prostředí a principy řídícími její návrh a evoluci“* [7].

Při hledání nějaké architektury, či architektonického stylu, který by popisoval, jak navrhovat knihovny a frameworky, zjistíme, že žádný takový zatím není. Na rozdíl od návrhu architektury softwarových aplikací, kde můžeme narazit na pojmy jako vícevrstvá architektura, architektura klient-server a další. Proto zde uvedu obecně platné principy, které lze použít jak při návrhu architektury, tak i při samotném návrhu funkcí a tříd systému.

4.1 Obecné principy

V této části představím několik základních principů, na které je potřeba myslet při návrhu jednotlivých částí systému od návrhu jednotlivých tříd přes komponenty systému až po celý systém jako celek. V této části vycházím z přednášky Ing. Špačka, Ph.D. dostupné z [31].

4.1.1 Don't Repeat Yourself

„Každá dílčí znalost musí mít v systému jedinou, jednoznačnou a směrodatnou reprezentaci.“

DRY (Don't Repeat Yourself) princip říká, že pokud se informace vyskytuje na více místech současně, mělo by být jedno místo označeno jako směrodatné a ostatní by se od něj měly odvozovat. Princip se nevztahuje jenom na zdrojové kódy, ale také na všechny další materiály softwarového produktu jako například specifikaci, dokumentaci a další. Porušení nejčastěji vede na

rozchod jednotlivých informací (např. v případě chyby v kódu, kdy její oprava na jednom místě nezaručuje, že se chyba v systému již nenachází).

4.1.2 Keep It Simple Stupid

„Většina systémů funguje nejlépe, pokud jsou spíše jednoduché než složité. Cílem návrhu by mělo být vyhnout se zbytečné složitosti.“

KISS (Keep It Simple Stupid) říká, že v návrhu by se mělo myslet na to, aby se podařilo splnit požadovanou funkcionalitu co možná nejjednodušším způsobem. Ovšem tento princip neznamená, že by výsledná kvalita řešení měla být špatná.

4.1.3 Separation of Concerns

SoC (Separation of Concerns) říká, že bychom měli systém navrhovat tak, abychom oddělili části systému se společnou funkcionalitou od ostatních. Vytváříme modulární systém, kde každý modul komunikuje s ostatními prostřednictvím jasně definovaného rozhraní. Typickým příkladem, který se nejčastěji v souvislosti s tímto principem uvádí, jsou webové technologie: HTML (Hypertext Markup Language) popisuje strukturu, CSS (Cascading Style Sheets) se stará o formát zobrazení a JavaScript o výkonný kód.

4.1.4 You Are not Going to Need It

„Programátor by neměl přidávat funkcionalitu nad rámec řešení problému.“

YAGNI (You Are not Going to Need It) tedy říká, že bychom neměli přidávat nic, co není potřeba pro splnění požadovaného zadání.

4.1.5 Law of Demeter

„Každá jednotka by měla mít omezené znalosti o ostatních jednotkách s výjimkou jednotek, se kterými je úzce spjata.“

V OOP ke splnění principu napomáhá dodržování správného zapouzdření tříd.

4.1.6 Zero One Infinity Rule

„Nemělo by být povoleno definovat konkrétní počet instancí konkrétní entity. Jednotka by měla být buď zakázána, nebo mít jednu instanci, nebo libovolný počet instancí.“

ZOIR (Zero One Infinity Rule) vlastně říká, že na úrovni návrhu by se neměly uvažovat případy omezení konkrétního počtu instancí konkrétní entity.

4.2 SOLID

Asi za nejznámější principy spojené s návrhem softwaru lze označit pět principů pod souhrnným označením SOLID. V následující části vycházím z [16, 32].

4.2.1 The Single Responsibility Principle

Obecné znění principu SRP (Single Responsibility Principle) je následující: *„Modul by měl mít jen jednu reakci na změnu.“* V [16] je uvedena konkrétnější verze a to: *„Modul by měl být odpovědný jednomu jedinému účastníkovi.“* Modulem je zde myšlen konkrétní soubor se zdrojovými kódy (například kolekce funkcí nebo kolekce datových struktur). Každý modul by měl obsahovat spolu související funkce a díky tomu by měl mít jeden jediný důvod ke změně.

4.2.2 The Open-Closed Principle

„Softwarový artefakt by měl být otevřený pro rozšíření, ale uzavřený pro změnu.“

OCP (Open-Closed Principle) říká, že chování softwarového artefaktu, jako je např. modul, třída, funkce a jiné, by mělo být rozšiřitelné bez nutnosti změny původního artefaktu. Typickým příkladem dodržení principu může být preferování použití rozhraní před konkrétní implementací (využití abstrakce).

4.2.3 The Liskov Substitution Principle

„Podtypy musí být schopny zastoupit své nadtypy z hlediska struktury, ale i z hlediska chování.“

LSP (Liskov Substitution Principle) určuje, co by měla splňovat hierarchie navržených tříd. Typickým příkladem porušení principu je návrh tříd tvarů obdélníku a čtverce, kde čtverec je navržen jako potomek obdélníku.

4.2.4 The Interface Segregation Principle

„Klienti by neměli být nuceni záviset na metodách, které nepoužívají.“

ISP (Interface Segregation Principle) vlastně mluví o návrhu rozhraní mezi jednotlivými částmi. Rozhraní, prostřednictvím kterého spolu jednotlivé části komunikují, by mělo obsahovat jen ty metody, které skutečně navzájem využívají.

4.2.5 The Dependency Inversion Principle

„Vysokoúrovňové moduly by neměly záviset na nízkoúrovňových. Měly by záviset jen na jejich abstrakci (na rozhraní).“

„Abstrakce by neměla záviset na detailu (na konkrétní implementaci). Detail by měl záviset na abstrakci.“

Pro DIP (Dependency Inversion Principle) se můžeme také setkat s méně formální definicí: *„Nevolejte nás, my zavoláme vás.“*

4.3 Principy komponent systému

V této části se zaměřím na principy spojené s návrhem dělení systému do jednotlivých komponent. Tyto principy jsou velmi důležité vzhledem k povaze této práce.

Komponenta je jednotka systému, která může být samostatně nasaditelná. Jedná se o nejmenší část systému poskytující konkrétní uzavřenou funkcionalitu. Například .jar soubory v Javě, .dll v .Net, nebo gem v Ruby [17]. Komponenty mohou být spolu dynamicky linkovány, nebo nasazovány společně v rámci celých balíčků (.war apod.).

4.3.1 Soudržnost komponent

V této části se zabývám soudržností komponent systému. Uvádím zde tři principy podle [18] a dále čerpám z [5].

4.3.1.1 The Reuse/Release Equivalence Principle

„Jednotka přepoužitelnosti je zároveň i jednotkou vydání.“

REP (Reuse/Release Equivalence Principle) říká, že pokud je jednotka systému použitá vícekrát, měla by být i samostatně vydávaná. Například v kontextu Javy, pokud nějakou část funkcionalit využívám na více místech, měl bych ji vydat jako samostatný .jar soubor.

4.3.1.2 The Common Closure Principle

„Seskupte jednotlivé třídy do komponent tak, aby platilo, že třídy v jedné komponentě se mění ve stejný čas a ze stejného důvodu. Naopak oddělte od sebe třídy, které se mění v různý čas a z různého důvodu.“

Zjednodušeně řečeno, komponenta systému by neměla mít více důvodů ke změně. Princip tedy určuje, podle čeho bychom měli sdružovat jednotlivé třídy systému do komponent. Jedná se o SRP zobecněný na komponenty systému.

4.3.1.3 The Common Reuse Principle

„Nenuťte uživatele komponent záviset na věcech, které nepotřebují.“

CRP (Common Reuse Principle) je zobecnění ISP na úroveň komponent systému. Rozhraní komponent, pomocí kterých spolu komunikují by neměla obsahovat nic navíc, co komunikující komponenty nepotřebují.

4.3.2 Provázanost komponent

V této části se zabývám vazbami mezi komponentami systému. Uvádím zde tři principy podle [19] a dále čerpám z [5].

4.3.2.1 The Acyclic Dependencies Principle

„Graf závislostí mezi komponentami nesmí obsahovat žádný cyklus.“

Již z definice je patrné, co ADP (Acyclic Dependencies Principle) říká. Cyklus v grafu závislostí je špatně nejen podle tohoto principu, ale může také způsobovat problémy při sestavování výsledného artefaktu. Pokud bychom v grafu závislostí nalezli cyklus, můžeme použít DIP k jeho odstranění.

4.3.2.2 The Stable Dependencies Principle

„Buď závislý ve směru stability.“

SDP (Stable Dependencies Principle) říká, že komponenta by neměla být závislá na komponentě, která se často mění. Porušením tohoto principu vzniká tzv. dominový efekt (změna komponenty vyvolá změnu v komponentách, které jsou ní závislé). Tento princip zavádí pojem nestabilita (I), který je definován jako:

$$I = \frac{\#\text{příchozích závislostí}}{\#\text{příchozích závislostí} + \#\text{odchozích závislostí}}$$

Podle SDP by měla být nestabilita komponenty větší než nestabilita komponent, na kterých je závislá. Pro dodržení principu by komponenta měla být závislá pouze na stabilnějších komponentách, než je ona sama.

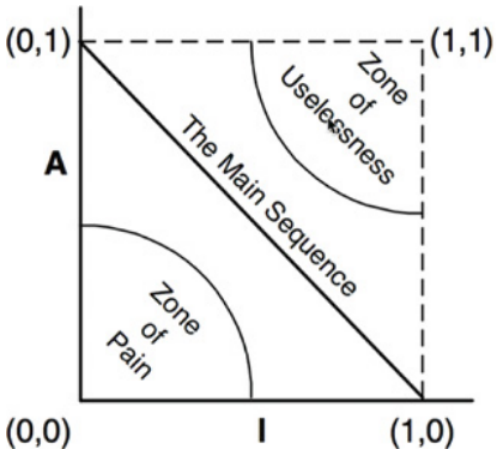
4.3.2.3 The Stable Abstractions Principle

„Komponenta by měla být tak abstraktní jako je stabilní.“

SAP (Stable Abstractions Principle) definuje pojem abstraktnost (A):

$$A = \frac{\#\text{tříd v komponentě}}{\#\text{abstraktních tříd} + \#\text{rozhraní v komponentě}}$$

a dává ho do souvislosti s pojmem nestabilita. Jedná se o metriku, která určuje vztah mezi těmito pojmy. Cílem návrhu by mělo být najít rovnováhu mezi oběma veličinami. Vztah obou pojmů je zobrazen na obr. 4.1.



Obrázek 4.1: Zóny vyloučení [5]

Kapitola 5

Analýza

V této kapitole se zaměřím na analýzu současného řešení frameworku X-definice a představím jeho základní prvky, jeho chování a použití. Následně zadefinuji požadavky, které jsou kladené na novou verzi frameworku. Na základě požadavků pak analyzuji potřebné technologie, které budou v nové verzi použity.

5.1 Současné řešení

Jak už bylo popsáno v kapitole 1, technologie X-definice se skládá z vlastního jazyka pro provádění výkonného kódu, předpisu pro XML dokument a frameworku v jazyce Java. Celý framework je prakticky jen překladač a procesor. Překladač překládá jazyk X-definic do instrukcí a ty jsou následně prováděny procesorem X-definic.

Celé řešení je navrženo tak, aby mělo co nejvíce stupňů volnosti. To dokládá i fakt, že v rámci X-definice, lze popsat i strukturu samotné X-definice. Samotný framework postrádá nějaký výrazný architektonický vzor. Ze softwarového hlediska je řešení značně ovlivněné způsobem jeho vzniku, kdy původní verze byla rozšiřovaná o podporu nových funkcionalit. Nachází se zde mnoho tříd, jejichž celkový počet řádků kódu překračuje řád tisíců. Celkové řešení je také poznamenané přechodem na novou verzi Javy, kdy v kódu frameworku zůstaly konstrukty jazyka, které mohly být při přechodu na vyšší verzi odstraněny, a to zvyšuje jeho nepřehlednost.

5.1.1 Základní komponenty

Pokud se na celý framework podíváme z hlediska logických celků, můžeme ho rozdělit na tři základní komponenty:

- kompilátor,
- komponentu starající se o zpracování (runtime) a

- výkonný kód (skript).

5.1.1.1 Kompilátor

Kompilátor pracuje pouze s X-definicí resp. s množinou X-definic a provádí její překlad (kompilaci) do vnitřní reprezentace. Během kompilace se provádí validace všech součástí X-definice, jako jsou modely elementů, deklarace proměnných a metod atd., a také validaci všech částí skriptu X-definice. Během procesu kompilace dochází k nahrazení referencovaných modelů a jejich částí. Dále je také provedeno rozparsování skriptu a nahrazení všech variabilních výrazů jejich jednotnou formou. Výkonné části skriptu jsou přeloženy na instrukce procesoru X-definic. Výstupem kompilace je reentrantní objekt `XDPool`, který v sobě obsahuje kolekci zkompilovaných X-definic.

5.1.1.2 Runtime

Runtime se stará o průchod zkompilovanou X-definicí a vstupními daty. Provádí načítání a parsování vstupního dokumentu. Podle zvoleného režimu je jeho chod řízen buď vstupními daty, nebo X-definicí.

V režimu validace Runtime načítá prostřednictvím SAX (Simple API for XML) parseru vstupní dokument a podle předpisu určeného X-definicí provádí jeho validaci. V případě, že předpis obsahuje speciální konstrukce jazyka (např. atribut `xd:text`), může provádět i částečnou změnu dat. Pokud je během zpracování zaznamenána chyba, tak se v závislosti na konfiguraci provede příslušná akce (např. chyba je zaznamenána a pokračuje se dál, nebo proces skončí apod.). Výstupem validace je DOM (Document Object Model) struktura (metody spouštějící process zpracování vrací `org.w3c.dom.Element`). Výsledný DOM může reprezentovat buď celý vstupní dokument, nebo jen jeho fragment. Fragment dokumentu je vrácen v případě, že je proces validace nakonfigurován tak, že se během procesu provádí průběžný zápis do výstupního proudu (specifikováno v rámci X-definice). Přístup s průběžným zápisem je využíván při zpracovávání velkých vstupních dokumentů.

V režimu konstrukce je situace běhu Runtime komponenty velmi podobná jako v režimu validace s tím rozdílem, že se zpracování řídí podle X-definice. Prochází se vstupní X-definice a v případě, kdy procesor narazí na akci `create`, provede vytvoření příslušných částí výstupního dokumentu[3]. V průběhu zpracování může X-definice získávat data z kontextu, který ji byl na začátku běhu předán. Kontext dat může být následujícího typu:

- `XDContainer` – speciální interní struktura pro reprezentaci množiny dat,

[3]Logika zpracování akce `create` je velmi komplikovaná a pro tento text není úplně podstatná. Proto se ji v textu blíže nevěnuji. V případě zájmu je logika zpracování popsána v oficiální dokumentaci pro konstrukční režim dostupné z `http://xdef.syntea.cz/tutorial/cs/userdoc/xdef3.1_construction_mode_ces.pdf`.

- `XDResultSet` – interní struktura reprezentující data z databáze,
- `org.w3c.dom.Node` – standardní XML struktura reprezentující libovolný prvek XML a
- `String` – může reprezentovat buď cestu k souboru s daty, nebo URL adresu, či samotná data.

Pokud během procesu konstrukce dojde k chybě záleží na konfiguraci, jak se procesor zachová (viz. proces validace). Výstupem je stejně jako v režimu validace DOM, který obsahuje buď celý výstupní dokument, nebo jen jeho fragment.

5.1.1.3 Skript

Komponenta Skript provádí vykonávání samotných instrukcí, které byly na základě X-definice vytvořeny při překladu. Obsahuje všechno, co je spojené s výkonným kódem X-definic.

5.1.2 Veřejné rozhraní

Veřejné rozhraní frameworku X-definice částečně kopíruje jeho dekompozici výše. Použití frameworku se skládá ze čtyř fází, které lze vidět na ukázce zdrojového kódu 5.1.

Fáze kompilace

V této fázi probíhá kompilace všech vstupních X-definic. Výstupem kompilace je množina zkompilovaných X-definic označována jako `XDPool`. Proces kompilace lze zahájit několika způsoby. Nejzákladnějším způsobem je zavolání statické metody `compile` na třídě `XDPool`, která očekává proměnný počet parametrů jako zdrojů X-definic. X-definice mohou pocházet z různých zdrojů např. ze souboru, ze vzdálené URL, či jako textový řetězec apod. Při zahájení kompilace lze specifikovat tzv. reportér, který slouží k záznamu chyb či varování vzniklých během procesu kompilace. V případě, že je reportér nastaven na hodnotu `null`, jsou tyto informace součástí vyhozené výjimky typu `SRuntimeException`.

Fáze inicializace

V této fázi je z `XDPoolu` na základě jména „vyzvednuta“ X-definice, což je prováděno zavoláním metody `createXDDocument` na `XDPoolu`. Metoda vytvoří a vrátí jednorázový objekt typu `XDDocument`. Tento objekt je reprezentací X-definice, ale na rozdíl od X-definic v `XDPoolu` už není reentrantní. `XDDocument` slouží pro nastavení kontextů, které jsou využity při samotném

```
try {
  // Fáze 1
  // Kompilace X-definice
  XDPool xdPool = XDFactory.compileXD(null, "example.xdef");

  // Fáze 2
  // Získání X-definice 'exampleDef'
  // z kontejneru zkompilovaných X-definic
  XDDocument xDoc = xdPool.createXDDocument("exampleDef");

  // Fáze 3
  // Spuštění v režimu validace
  Element vElement = xDoc.xparse("example.xml", null);

  // Fáze 2
  // Spuštění v režimu konstrukce pro model 'ExampleElem'
  // Nastavení vstupních dat do režimu konstrukce
  xDoc = xdPool.createXDDocument("exampleDef");
  xDoc.setXDContext(vElement);

  // Fáze 3
  Element cElement = xDoc.xcreate("ExampleElem", null);
} catch (Exception ex) {
  // Fáze 4
  // výpis chyb
  System.out.println(ex.getMessage());
}
```

Zdrojový kód 5.1: Ukázka základního použití X-definic

procesu zpracování (např. nastavení dat pro konstrukční režim, nebo výstupního proudu pro průběžný zápis během procesu zpracování). Pokud byl objekt `XDDocument` již jednou použit, měl by být zahozen a v případě potřeby vytvořen a nainicializován znova.

Fáze zpracování

V této fází probíhá zahájení samotného procesu zpracování. Pro zpracování v režimu validace je na objektu `XDDocument` zavolána metoda `xparse`, která na vstupu očekává zdrojový XML dokument. Metoda opět podporuje mnoho různých zdrojů dokumentu. Stejně jako ve fázi kompilace může být metodě předán reportér pro zápis chyb. Výstupem metody je DOM, nebo v případě neúspěchu zaznamenané chyby.

Pro spuštění režimu konstrukce slouží na objektu `XDDocument` metoda `xcreate`. Metoda na vstupu očekává jméno modelu, který je součástí X-definice a od kterého má začít zpracování. Druhým parametrem je opět reportér a stejně jako u předchozího režimu je vrácen DOM, nebo chyby.

Fáze reportu

V poslední fázi se provádí vyhodnocení výsledku zpracování. V případě použití reporteru lze zjistit, jak zpracování dopadlo a následně vypsat výsledek. V případě běhu bez reportéru lze vypsat chyby jako zprávu, která je součástí výjimky.

5.1.3 Podpora JSONu

Současná verze technologie X-definic nabízí i částečnou podporu pro práci s daty ve formátu JSON. JSON dokument je popsán standardní X-definicí tedy XML dokumentem. Vstupní data ve formátu JSON jsou při zpracování mapovány na odpovídající XML strukturu a dále jsou zpracována klasicky jako by se jednalo o XML dokument. Pro zachování informace o typu dat, jsou pro hodnoty definovány speciální funkce jako např. `jnumber()` apod.

5.2 Analýza požadavků

Jednou z činností v softwarovém procesu je analýza a sběr požadavků. Díky definování všech požadavků, které na systém máme, si lze snadno udělat přehled o velikosti a složitosti softwarového systému. Na základě definovaných požadavků jsme schopni určit a následně ověřit, co má daný systém splňovat. Dále jsme také schopni určit, jaké všechny technologie budeme potřebovat. Níže zmíněné požadavky vycházejí z aktuální verze technologie X-definice, nebo ze samotného zadání diplomové práce, či byly dodefinovány během konzultací s vedoucím práce.

5.2.1 Funkční požadavky

Funkční požadavky definují požadavky na systém z pohledu funkcionality, tedy popisují to, co by měl daný systém dělat. Seznam funkčních požadavků pro framework obsahuje následujících šest požadavků.

F1 Validace dokumentů ve formátu XML

Knihovna umožňuje validování dokumentů ve formátu XML podle předpisu definovaného v X-definici. Výstupem je validní XML dokument, informace o obsažených chybách, nebo obojí. Požadavek kopíruje funkcionalitu současné verze.

F2 Konstrukce (transformace) dokumentů ve formátu XML

Knihovna umožňuje konstrukci dokumentů ve formátu XML podle předpisu definovaného v X-definici. Výstupem je XML dokument, nebo informace o chybách během zpracování. Požadavek kopíruje funkcionalitu současné verze.

F3 Validace dokumentů ve formátu JSON

Knihovna umožňuje validování dokumentů ve formátu JSON podle předpisu definovaného v X-definici. Výstupem je validní JSON dokument, informace o obsažených chybách, nebo obojí. Požadavek kopíruje funkcionalitu současné verze pro dokumenty ve formátu XML.

F4 Konstrukce (transformace) dokumentů ve formátu JSON

Knihovna umožňuje konstrukci dokumentů ve formátu JSON podle předpisu definovaného v X-definici. Výstupem je JSON dokument, nebo informace o chybách během zpracování. Požadavek kopíruje funkcionalitu současné verze pro dokumenty ve formátu XML.

F5 Pracování velkých vstupních souborů

Knihovna umožňuje zpracovávat velké vstupní dokumenty (řádově až gigabajty dat).

F6 Identifikace v dokumentu

Všechna načtená data je možné jednoznačně identifikovat ve zdrojovém dokumentu. Pro každou významnou část lze určit na jaké řádce a v jakém sloupci začíná (např. element X začíná na řádce 1 ve sloupci 5).

5.2.2 Nefunkční požadavky

Nefunkční požadavky, na rozdíl od těch funkčních, nedefinují požadavky kladené na funkcionalitu systému, ale na chod systému, jeho fungování, či jeho samotnou tvorbu. Nefunkční požadavky přímo neovlivňují používání daného systému.

N1 Koncepčně čistá architektura

Architektura knihovny je navržena tak, aby neporušovala běžné koncepty a doporučení pro návrh softwarových architektur.

N2 Rozšiřitelná architektura

Architektura knihovny musí umožňovat snadné rozšíření o další funkcionality, např. přidání dalšího formátu vstupních dokumentů.

N3 Logování

Knihovna používá rozhraní pro zaznamenávání jejího běhu (tzn. logování). Logování lze snadno konfigurovat a měnit.

N4 JVM

Knihovna je realizována v jazyku umožňující běh pod JVM (Java Virtual Machine) a který je plně kompatibilní s jazykem Java. Výslednou knihovnu je tedy možné použít v projektu v Javě standardní způsobem, jakoby se použila knihovna implementovaná přímo v jazyce Java.

N5 Build systém

Výsledná knihovna bude sestavována prostřednictvím softwaru na projektový management Maven.

5.3 Programovací jazyk

Na základě požadavku N4 jsem analyzoval dostupné jazyky pro JVM. Provedl jsem analýzu čtveřice dnes nejrozšířenějších technologií pro vývoj aplikací pro JVM. U každé z nich jsem se primárně zaměřil na její kompatibilitu s jazykem Java.

5.3.1 Java

Pokud hledáme programovací jazyk, která je kompatibilní s jazykem Java, tak první volba je samotný jazyk Java. Java je objektově orientovaný programovací jazyk. Jedná se o primární jazyk pro psaní aplikací pro JVM. Jazyk Java je striktně typovaný, tzn. každá deklarovaná proměnná má svůj typ a během existence nemůže typ měnit (nelze do ni přiřadit objekt jiného typu). Všechno v Javě je objekt s výjimkou primitivních typů jako `int`, `double` apod. Java je jedním z nejpoužívanějších programovacích jazyků na světě [38]. Od verze 1.8 se v jazyku objevují prvky funkcionálního programování jako jsou např. lambda funkce.

5.3.2 Kotlin

Kotlin je jazyk navržený a vyvinutý společností JetBrains opírající se o rozsáhlou komunitu jeho příznivců. Od roku 2017 je vedle Javy jedním z oficiálních jazyků pro vývoj aplikací pro mobilní platformu Android. Jedná se o kombinaci objektově orientovaného programovacího jazyka s funkcionálním jazykem. Na rozdíl od Javy v Kotlinu platí, že všechno je objekt. Výhodou Kotlinu je jeho plná kompatibilita s jazykem Java, tzn. je možné z kódu v Kotlinu volat kód v Javě a naopak. Kotlin je označován jako null safety jazyk, tzn.

nabízí konstrukty jazyka, jak elegantně pracovat s `null` hodnotou a předcházet tím chybám typu `NullPointerException`. Kotlin je stejně jako Java striktně typovaný jazyk. V porovnání s kódem v jazyce Java, je kód Kotlinu mnohem úspornější, což může vést, ale ne nutně, na přehlednější kód. Na druhou stranu může být vysoká variabilita možností v Kotlinu někdy na škodu. Jazyk Kotlin lze použít i pro psaní JavaScript aplikací nebo přímo pro vývoj nativních aplikací.

5.3.3 Scala

Scala je moderní programovací jazyk kombinující jak objektově orientovaný přístup, tak funkcionální přístup. Jedná se o jazyk vyvinutý na půdě švýcarské univerzity École Polytechnique Fédérale Lausanne. Jedná se o staticky typovaný jazyk umožňující vytváření programů jak pro JVM, tak i pro JS (JavaScript). Scala jde plně kombinovat s Javou a naopak. [8]

5.3.4 Groovy

Groovy je volitelně typovaný dynamický jazyk. Pro platformu Java umožňuje statickou typovost a statickou kompilaci. Groovy nabízí jak objektový, tak funkcionální přístup. Jazyk je plně kompatibilní s jazykem Java. Oproti Javě nabízí Groovy volnější syntaxi jazyka. [35]

Vedle statické kompilace nabízí Groovy i podporu skriptování. V případě použití skriptovacího přístupu se ztrácí kompatibilita s Javou.

5.3.5 Shrnutí

Všechny analyzované jazyky mají mnoho společných vlastností a zároveň splňují požadavek na kompatibilitu s jazykem Java. Hlavní rysy jazyků jsou shrnuté v tab. 5.1.

Tabulka 5.1: Porovnání jazyků pro JVM

	Java	**Kotlin**	**Scala**	**Groovy**
Paradigma	Objektové	Objektové, funkcionální	Objektové, funkcionální	Objektové, funkcionální
Kompatibilita s Javou	Ano	Ano	Ano	Při kompilaci
Typovost	Statická	Statická	Statická	Statická i dynamická

5.4 Logování

Jedním z požadavků je umožnit v rámci běhu frameworku důkladné logování, které bude umožňovat vhodnou konfigurovatelnost. Pro logování v Javě existuje mnoho knihoven. V této části se zaměřím na ty nejčastěji používané knihovny poskytující podporu pro logování v jazyku Java.

5.4.1 java.util.logging

Knihovna java.util.logging je přímo součástí JDK (Java Development Kit). Jedná se o nativní knihovnu pro logování. Hlavní výhodou je, že není potřeba přidávat žádné externí závislosti. Jako nevýhoda oproti ostatním řešením může být menší konfigurovatelnost.

5.4.2 Apache Log4j 2

Apache Log4j 2 je druhou generací knihovny pro logování. Nabízí vysokou konfigurovatelnost a podporuje změnu konfigurace za běhu. Knihovna je rozdělena na core a API modul z důvodu minimalizace závislosti projektu na konkrétní implementaci. Apache Log4j 2 nabízí velké množství tzv. appenderů, které slouží pro zaznamenávání logů do různých služeb např. do databází jako třeba MongoDB, CoucheDB a další. Apache Log4j 2 také obsahuje API (Application Programming Interface) rozšíření pro Kotlin. [36]

5.4.3 Logback

Logback je nástupce knihovny pro logování log4j. Logback nabízí nativní implementaci SLF4J (Simple Logging Facade for Java). [25]

5.4.4 SLF4J

SLF4J není přímo logovací knihovna. Je to jednoduchá fasáda nad nejběžněji používanými logovacími knihovnami. SLF4J podporuje logování prostřednictvím standardního logu (java.util.logging), logback, nebo log4j. [26]

5.4.5 Shrnutí

Každá z výše zmíněných knihoven pro logování (vyjma SLF4J) nabízí podobnou funkcionalitu snad jen s výjimkou java.util.logging. Z analyzovaných knihoven lze tedy použít buď SLF4J v kombinaci s nějakou podporovanou knihovnou, nebo pak Apache Log4j 2.

5.5 Frameworky pro zpracování dat

Při analýze dostupných frameworků pro zpracování dat ve formátech XML a JSON jsem se zaměřil na několik funkcionalit, které plynou z definovaných požadavků:

- zpracování do objektové reprezentace,
- proudové zpracování,
- podpora lokalizace zpracovávaných objektů ve vstupních datech.

5.5.1 XML

Pro zpracování dokumentů ve formátu XML existuje řada přístupů, z nichž některé jsou standardizované nebo je jejich implementace součástí přímo JDK. V této části analýzy se zabývám buď obecným přístupem ke zpracování tj. definovaným standardem jako DOM či SAX, nebo přímo konkrétní knihovnou nabízející vlastní přístup, než který je součástí standardu.

5.5.1.1 DOM parser

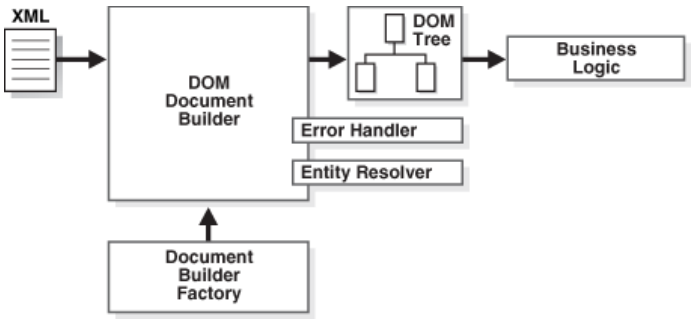
DOM je platformně a jazykově nezávislý objektový model pro XML dokumenty definovaný a spravovaný W3C. DOM definuje aplikační rozhraní pro přístup a modifikaci stylu, struktury a obsahu XML dokumentů. XML parsery podporující DOM implementují toto rozhraní [37, 39].

DOM je stromová struktura reprezentující parsovaný XML dokument, kde každý uzel reprezentuje nějaký prvek XML dokumentu např. element, textovou hodnotu, komentář apod. Standardní implementace DOM a příslušného parseru je přímo součástí JDK. Na obr. 5.1 je schéma zpracování dokumentu DOM parserem. Načtení XML dokumentu a následná práce s daty je na ukázce zdrojového kódu E.2. Jedná se o parser, který převádí vstupní dokument na jeho objektovou reprezentaci (DOM), takže ho nelze bezpečně použít na neomezeně velkých datech.

5.5.1.2 Java Document Object Model

Jak jsme si mohli všimnout na ukázce použití předchozího parseru, či spíše na práci s výsledným DOM, práce s tímto API není zrovna nejkomfortnější. To je způsobeno platformní a jazykovou nezávislostí celého přístupu. Přesně z tohoto důvodu existuje uživatelsky přívětivější rozhraní pro práci s XML dokumenty v jazyce Java.

JDOM (Java Document Object Model) je open-source knihovna určená pro práci s XML dokumenty v jazyce Java, definující vlastní objektovou reprezentaci dokumentu a také vlastní rozhraní pro práci s ním. JDOM neobsahuje vlastní XML parser, ale poskytuje rozhraní, pro načtení dokumentu



Obrázek 5.1: Schéma použití DOM parseru [24]

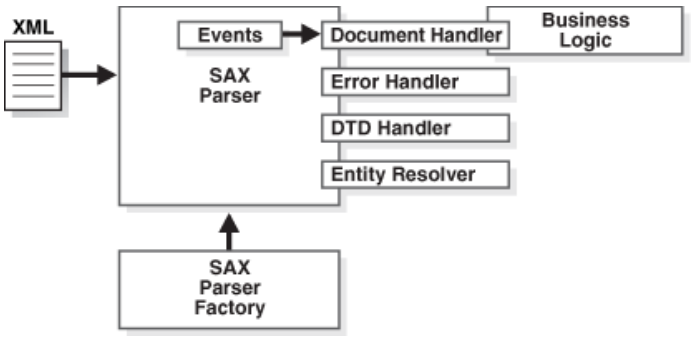
prostřednictvím SAX, StAX parseru, nebo již z existujícího DOM. Načtení XML dokumentu a následná práce s daty je na ukázce zdrojového kódu E.1.

5.5.1.3 Document Object Model for Java

Ze stejného důvodu, jako vznikla knihovna JDOM, vznikla i knihovna DOM4J (Document Object Model for Java). DOM4J na rozdíl od JDOM obsahuje vlastní implementaci XML parserů.

5.5.1.4 SAX

SAX je proudový push parser. Jedná se o standard definovaný konzorciem W3C pro proudové zpracování dokumentů ve formátu XML. Standard má mnoho implementací a jedna z nich je přímo součástí JDK. SAX parser může být použit jako parser pro další frameworky spojené s XML, např. pro již zmíněný JDOM. Standard definuje několik handlerů, které slouží k obsluze příslušných událostí. Základní schéma parsování s použitím SAX parseru je na obr. 5.2.



Obrázek 5.2: Schéma použití SAX parseru [24]

5.5.1.5 StAX

StAX (Streaming API for XML) je pull parser na parsování XML dokumentů. Dle [23] bylo hlavním cílem vzniku dalšího přístupu k parsování XML dokumentů, umožnit programátorovi vlastní řízení zpracování dat a dále pak řešení omezení spojených s použitím DOM, či SAX parseru. StAX API se skládá ze dvou částí:

- Cursor API – nízkoúrovňové API pro zpracování XML dokumentu a
- Iterator API – objektový přístup ke zpracování [22].

5.5.1.6 Shrnutí

Pro zpracování dokumentů ve formátu XML můžeme vybírat z mnoha dostupných přístupů. Obecně ale můžeme říct, že za hlavní dělící kritérium lze považovat to, zda chceme použít standardizovaný přístup (DOM a SAX), či alternativní způsob. Standardizovaný přístup je obvykle spojen s velmi nízkoúrovňovým řešením, z čehož plyne jeho vysoká obecnost. Alternativní přístup je naopak spojen spíše s vysokoúrovňovým řešením, které sice není tak obecné, ale nabízí nám mnohem komfortnější použití. Porovnání jednotlivých frameworků z hlediska požadavků vytyčených na začátku podkapitoly 5.5 je v tab. 5.2.

Tabulka 5.2: Porovnání přístupů pro zpracování XML

	DOM	**JDOM**	**DOM4J**	**SAX**	**StAX**
Objektový model	Ano	Ano	Ano	Ne	Ne
Proudový model	Ne	Ne	Ne	Push	Pull
Lokalizace	Ne	Ano	Ano	Ano	Ano

5.5.2 JSON

Na rozdíl od zpracování XML, pro které existuje několik standardů a jejichž základní implementace je součástí přímo JDK, je pro JSON situace úplně opačná. Aktuálně neexistuje žádný standard, který by definoval objektovou strukturu pro JSON dokument. Proto každý z vývojářů nástrojů pro zpracování JSON si objektovou reprezentaci pro JSON navrhuje podle sebe, i když jednotlivé návrhy jsou si v mnohém podobné. Stejně tak jako není standard pro objektovou reprezentaci, není standard ani pro proudové zpracování. To může být jedním z důvodů, proč není podpora pro práci s JSON formátem přímo součástí Java SE (Java Standard Edition).

Existuje tedy celá řada implementací zpracování formátu JSON v jazyce Java. Zaměřil jsem se, alespoň podle mě, na ty nejznámější a nejrozšířenější frameworky. Za některými z nich stojí společnosti jako Google, či Oracle.

5.5.2.1 Jackson

Jackson je rozsáhlý framework nejen na zpracování dokumentů ve formátu JSON. Framework umožňuje zpracovávat dokument jak v jeho objektové reprezentaci, tak jako proud dat [9]. V proudovém režimu framework umožňuje lokalizování objektů ve zdroji dat. V objektovém režimu tuto možnost nenabízí.

5.5.2.2 Gson

Gson je knihovna vyvinutá společností Google na serializaci a deserializaci Java objektů do jejich reprezentace v JSON formátu a naopak [10]. Jedná se o velmi populární knihovnu pro práci s formátem JSON v mobilních aplikací pro Android. Hlavní funkcionalitou knihovny je zpracování dokumentu do objektové reprezentace. Knihovna umožňuje i zpracování v proudovém režimu a podporuje kombinaci obou režimů. Dále nabízí implementaci zpracování částí proudu dat do objektové reprezentace. To lze ale využít jen v případně málo vnořených JSON struktur. Bohužel knihovna nepodporuje lokalizaci objektů ve zdroji dat.

5.5.2.3 JSON-P

JSON-P (JSON Processing) je rozhraní pro zpracování, jako je načítání, vytváření, transformace a dotazování, dokumentů ve formátu JSON. Rozhraní umožňuje zpracování dokumentu formou objektového modelu, nebo proudové zpracování [21]. Rozhraní je navržené firmou Oracle a aspiruje na to, stát se přímo součástí JDK. Nyní je pouze součástí implementace Java EE (Java Enterprise Edition) od GlassFish[4]. Rozhraní podporuje lokalizaci ve zdrojových datech, ale jen při zpracování v proudovém režimu.

5.5.2.4 Shrnutí

Jednotlivé analyzované frameworky nabízejí velmi podobnou funkcionalitu a je těžké mezi nimi hledat rozdíly. Za zmínku například stojí, že framework Jackson je velmi robustní, zatímco JSON-P absolutně minimalistický, protože se v podstatě jedná jen o definované rozhraní. Jako zlatá střední cesta se může jevit framework Gson, který ale nevyhovuje stanoveným požadavkům. Porovnání jednotlivých frameworků z hlediska požadavků vytyčených na začátku podkapitoly 5.5 je v tab. 5.3.

[4] `https://javaee.github.io/glassfish/`

Tabulka 5.3: Porovnání frameworků pro zpracování JSON

	Jackson	**Gson**	**JSON-P**
Objektový model	Ano	Ano	Ano
Proudový model	Ano	Ano	Ano
Lokalizace	Částečně	Ne	Částečně

Kapitola 6

Návrh

V této kapitole představím návrh architektury nového frameworku X-definice. V rámci podkapitoly 6.2 se mimo jiné pokusím nastínit myšlenky a důvody, které stály za návrhem architektury. V návrhu se zaměřuji na jednotlivé komponenty vznikajícího systému, jejich vzájemnou komunikaci a interakci. V dalších částech kapitoly se věnuji popisu navržených komponent, jejich hlavních částí a funkcionalitou. Dále se pak zabývám popisem rozhraní komponent a modelům pro výměnu dat. Na závěr kapitoly představím metodiku přechodu ze stávající architektury frameworku na architekturu novou. Všechny diagramy, které jsou v této kapitole zmíněny, jsou ve své skutečné velikosti součástí přílohy na CD.

Samotný návrh architektury se opírá o definované funkční a nefunkční požadavky. Dále je v něm zohledněna funkcionalita a použití stávajícího řešení.

6.1 Vymezení navrhované části

Jak jsem nastínil v kapitole 1, technologie X-definice je velmi rozsáhlý framework nabízející výkonné prostředky, které jsou popsány vlastním jazykem. Bylo by krajně nemožné v rámci tohoto prvotního návrhu postihnout celou jeho podporovanou funkcionalitu. Proto jsem se v návrhu omezil na jeho hlavní části. Na ty ostatní neméně významné jsem se úplně nesoustředil, i když jsem k nim samozřejmě přihlížel. V návrhu jsem se zabýval následujícími částmi:

- zpracování dokumentů ve validačním a konstrukčním režimu,
- X-definice jako předpis pro modely dokumentu,
- definování možného výskytu,
- definování událostí a reakce na ně a
- definování vlastních proměnných a funkcí.

Na základě požadavků jsem se v návrhu dále soustředil na možnosti rozšíření o podporu formátu JSON a případně i dalších formátů.

6.2 Architektura

Při návrhu architektury jsem vycházel z dekompozice současného řešení popsané v části 5.1.1. Ke každé z logických komponent jsem navrhl odpovídající fyzickou komponentu. Každá z těchto komponent zaštiťuje určitou podmnožinu funkcionality původního systému a má svoje jasně definované rozhraní, prostřednictvím kterého může komunikovat s ostatními komponentami. Pro výměnu dat slouží modelové třídy, které jsou předávány mezi komponentami. Z této první fáze návrhu vznikly tři následující komponenty:

- Compiler,
- Runtime a
- XScript.

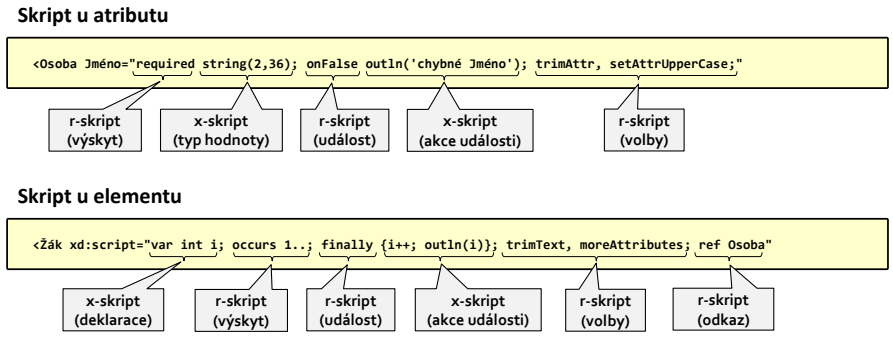
Compiler obsahuje funkcionality spojené s kompilací X-definice, Runtime obsahuje funkcionality spojené se zpracováním dat a XScript funkcionalitu spojenou se skriptem X-definic.

Tím, že jsem jednotlivé části systému takto striktně oddělil, otevřel se další prostor i pro rozdělení samotného jazyka X-definic. Jak bylo popsáno v podkapitole 1.2 skript X-definice je složen z několika částí. Jednotlivé části ale můžeme uspořádat do dvou skupin podle jejich významu. Do první skupiny patří části, které souvisí se zpracováním, či ke zjednodušení popisu a které mají jasně určenou svoji syntaxi. Patří sem:

- výskyt,
- definice událostí,
- volby a
- odkaz.

Do druhé skupiny patří zbylé části. Jsou to části skriptu, které by se daly označit za dynamické. Jedná se o tyto části:

- určení typu hodnoty,
- akce,
- definice proměnných a funkcí.



Obrázek 6.1: Schéma rozdělení skriptu X-definic

Na základě tohoto dělení zavedu pro první skupinu označení r-skript a pro druhou pak x-skript. Schéma skriptu rozděleného na jednotlivé části je zobrazeno na obr. 6.1.

Na základě předchozí části návrhu by zpracování skriptu X-definice náleželo komponentě XScript. Díky rozdělení jazyka X-definic můžeme přiřadit zpracování jazyka r-skript komponentě Runtime a zpracování jazyka x-skript ponecháme komponentě XScript. Díky tomu se architektura stává nezávislá na aktuálně naimplementované funkcionalitě jazyka X-definic a otevírá se prostor pro definici jazyka vlastního, či pro použití existujícího jazyka např. Pythonu apod.

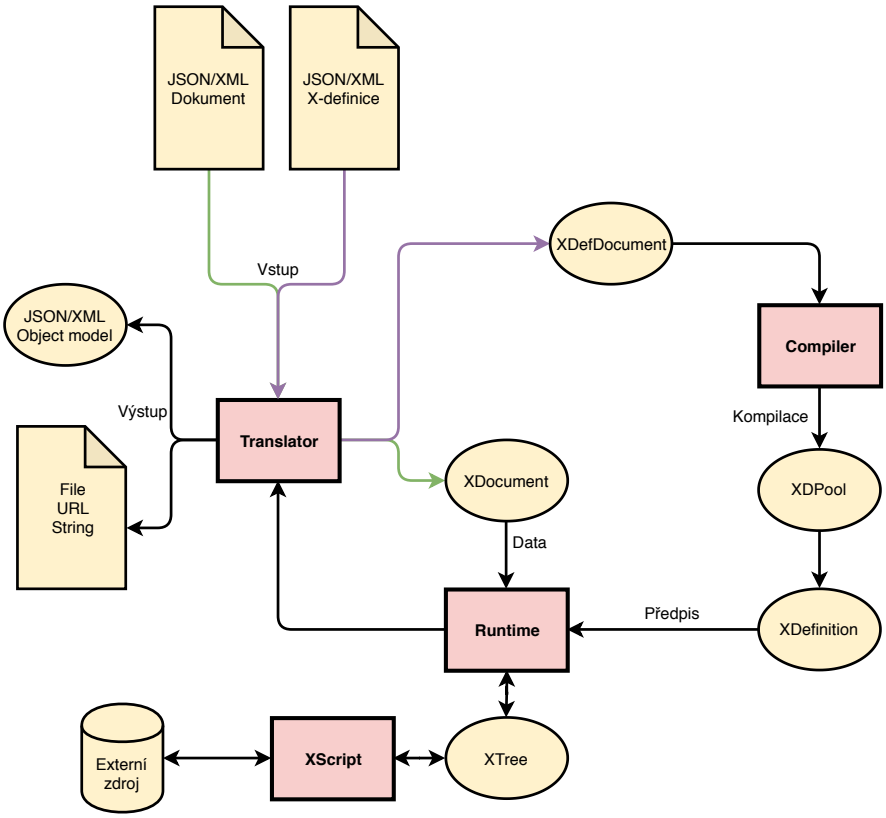
Poslední částí návrhu, kterou bylo potřeba udělat, bylo navrhnout komponenty systému tak, aby bylo možné zpracovávat jak formát XML, tak formát JSON. V kapitole 2, která se zabývá popisem obou formátů, jsme si mohli všimnout, že oba datové formáty jsou založené na stromové struktuře. Na základě této podobnosti jsem v návrhu začal uvažovat nad jednotnou strukturou pro oba formáty, čímž by se zamezilo roztříštěnosti systému dle podporovaných formátů dat.

Pro zajištění podpory různých formátů jsem navrhl jednu interní stromovou strukturu, do které jsou datové formáty překládány. Za účelem překladu datových formátů do interní struktury jsem navrhl další komponentu Translator. Díky jednotné vnitřní struktuře, není pak systém roztříštěný podle příslušného formátu, ale ke každému formátu se přistupuje stejně (v ostatních komponentách systému, se nevyskytuje informace o formátu dat).

6.3 Komponenty systému

Nyní máme systém rozdělený na čtyři komponenty (Compiler, Runtime, XScript a Translator). Potřebujeme ještě komponentu, která je všechny propojí a vystaví hlavní rozhraní systému. Za tímto účelem jsem navrhl pátou komponentu X-def. Komponenta vytváří tzv. fasádu nad celým systémem.

Celý návrh je zobrazen na diagramu komponent obr. D.1. Pro lepší pochopení zpracování dokumentů je na obr. 6.2 zobrazen diagram toku dat.



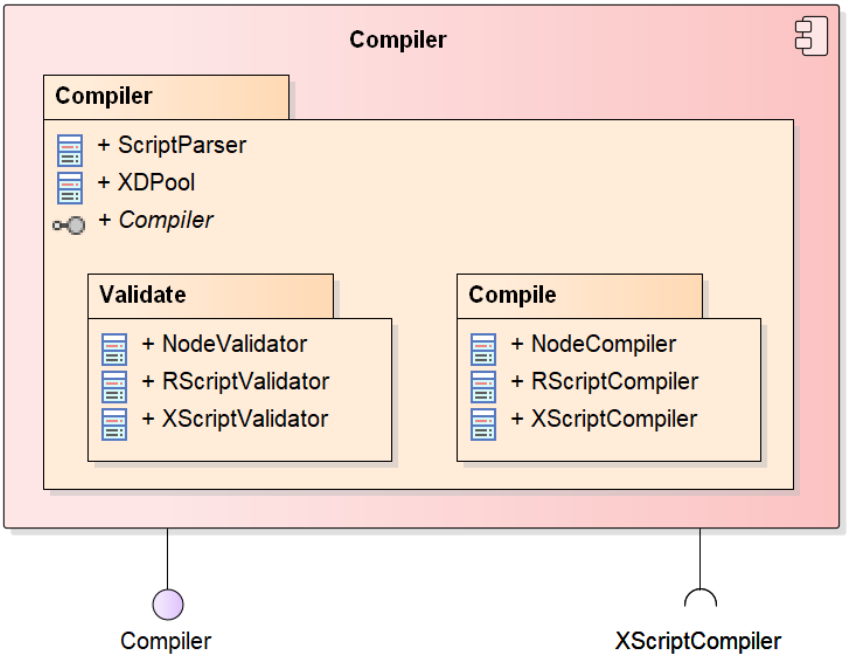
Obrázek 6.2: Digram toku dat

6.3.1 Compiler

Komponenta Compiler zajišťuje logiku kompilace X-definice, resp. kolekce X-definic. Komponenta pracuje s interní strukturou X-definice tzn. se strukturou `XDefDocument`. Výstupem kompilace je struktura `XDPool`, která v sobě obsahuje zkompilované X-definice (struktura `XDefinition`) a dále pak informaci o použitém XScriptu (struktura `XScriptInfo`). Proces kompilace se stává ze dvou kroků. Prvním krokem je zvalidování všech částí X-definice, druhým krokem je pak samotná kompilace X-definice. Základní části komponenty jsou:

- `Compiler` – rozhraní komponenty (více v části 6.4.1);
- `XDPool` – výstupní struktura obsahující kolekci zkompilovaných X-definic;

- `ScriptParser` – třída zajišťující rozparsování skriptu X-definice na jednotlivé části jako např. validační pravidlo, události a jejich akce apod., převede všechna variabilní vyjádření na jednotnou variantu;
- `NodeValidator` – provádí validaci struktury všech částí uzlu X-definice;
- `RScriptValidator` – provádí validaci částí jazyka r-skript;
- `XScriptValidator` – provádí validaci částí jazyka x-skript, za tímto účelem komunikuje s komponentou XScript;
- `NodeCompiler` – provádí kompilaci uzlu X-definice, během kompilace dochází k nahrazení všech odkazovaných modelů;
- `RScriptCompiler` – provádí kompilaci částí jazyka r-skript;
- `XScriptCompiler` – provádí kompilaci částí jazyka x-skript, za tímto účelem komunikuje s komponentou XScript.



Obrázek 6.3: Základní struktura komponenty Compiler

Validace a kompilace částí x-skriptu je delegována na komponentu XScript. Proces kompilace X-definice se skládá z několika kroků:

1. parsovaní skriptu X-definice,

2. vyhodnocení odkazů a validace struktury,
3. validace r-skriptu,
4. kompilace r-skriptu,
5. validace inicializační části x-skriptu,
6. kompilace inicializační části x-skriptu,
7. nastavení prostředí – XScriptu je předána zkompilovaná inicializační část (důležité pro zpracování dalších částí),
8. validace části pro kontrolu typu hodnoty,
9. kompilace části pro kontrolu typu hodnoty,
10. validace akcí,
11. kompilace akcí.

6.3.2 Runtime

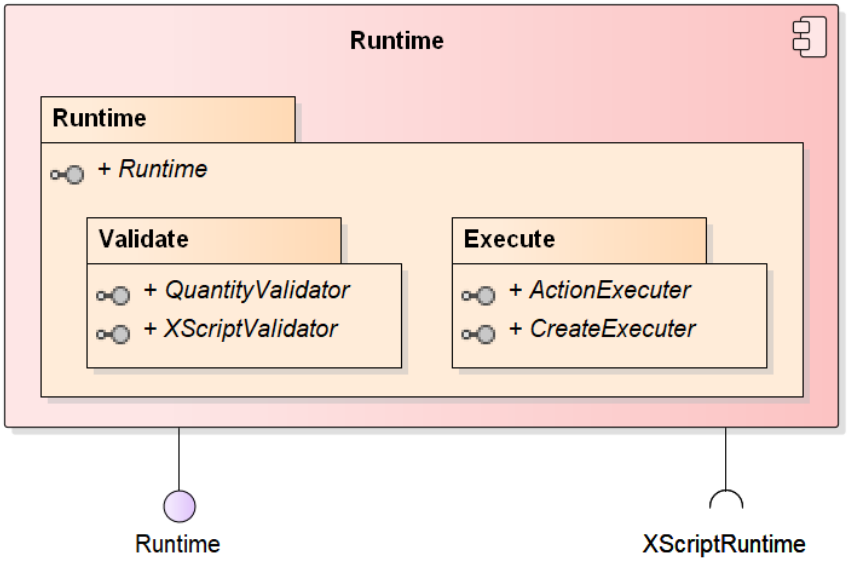
Komponenta zajišťuje proces zpracování. Podle zvoleného režimu provádí buď validaci, nebo konstrukci. Komponenta si udržuje kontexty zpracovávaného dokumentu, kontext X-definice, reportér pro zápis informací během zpracování a dále pak kontejner na ukládání dalších doplňujících informací, které lze využívat k dalšímu zpracování (struktura `InfoCollector`). Komponenta zajišťuje zpracování částí jazyka r-skript a dále provádí delegování zpracování částí jazyka x-skript na komponentu XScript.

Vzhledem k tomu, že části kompilace a samotného běhu od sebe mohou být oddělené, je nutné před zahájením zpracování provést kontrolu použitých XScriptů. Komponenta předá strukturu `XScriptInfo`, která je součástí vstupní X-definice, komponentě XScript, která rozhodne, zda je kompatibilní s XScriptem, který byl použit během kompilace. Komponenta se stará o vytvoření veškerého výstupu tzn. pokud je to vyžadováno provádí záznam informací o průběhu do reportéru a na konci zpracování vrátí interní strukturu dat, kterou během procesu zpracování vytvořila.

Hlavní části komponenty Runtime:

- `Runtime` – rozhraní komponenty (více v části 6.4.2);
- `QuantityValidator` – provádí validaci výskytu, pracuje tedy s částí r-skriptu, která slouží k definovaní výskytu, udržuje si počty jednotlivých objektů a případně oznamuje chybu např. překročení hranice možného výskytu;
- `XScriptValidator` – deleguje validaci typu hodnoty na komponentu XScript a určuje vykonání příslušných akcí;

- `ActionExecuter` – zajišťuje provedení definovaných akcí;
- `CreateExecuter` – speciální třída na vykonání akce create a zpracování jejího výsledku.



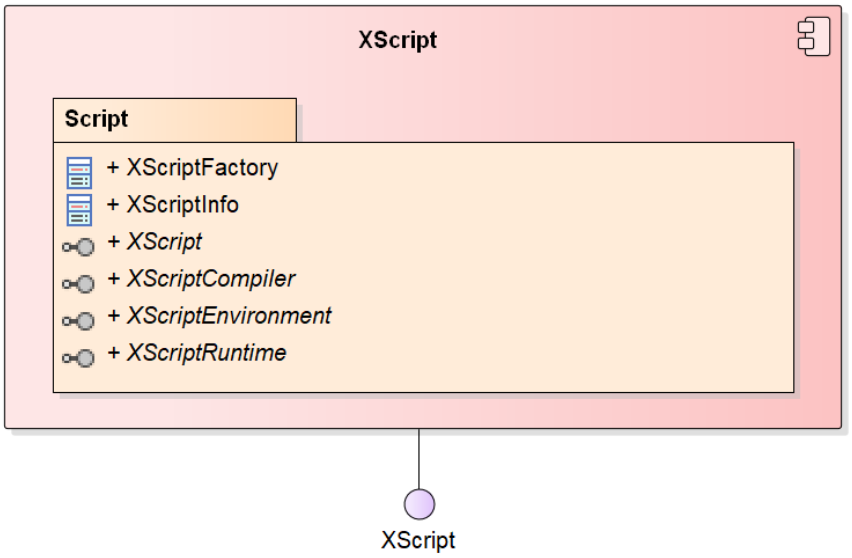
Obrázek 6.4: Základní struktura komponenty Runtime

6.3.3 XScript

Komponenta zajišťuje všechno, co je spojené s jazykem x-skript. Provádí jeho validaci, kompilaci a zajišťuje jeho provedení. Komponenta si udržuje kontext vlastního běhového prostředí, je proto nutné, aby nebyla používána opakovaně. Proto je její vytváření prováděno prostřednictvím třídy `XScriptFactory`. Třída `XScriptFactory` je také důležitým prvkem pro zajištění podpory různých verzí této komponenty. V rámci tvorby verze komponenty musí být provedena implementace `XScriptFactory`, která prostřednictvím faktory metody vrací novou instanci komponenty.

Struktura komponenty je plně v rukou jejího tvůrce. Měla by ale obsahovat části, které souvisí s jejím použitím jako např. část pro validaci x-skriptu, část pro kompilaci x-skriptu, část zajišťující běhové prostředí, ve kterém je x-skript vykonáván apod. Komponenta musí obsahovat implementaci následujících rozhraní:

- `XScript` – hlavní rozhraní komponenty (více v části 6.4.3) a
- `XScriptFactory` – zajišťuje vytváření nových instancí XScriptu.



Obrázek 6.5: Základní struktura komponenty XScript

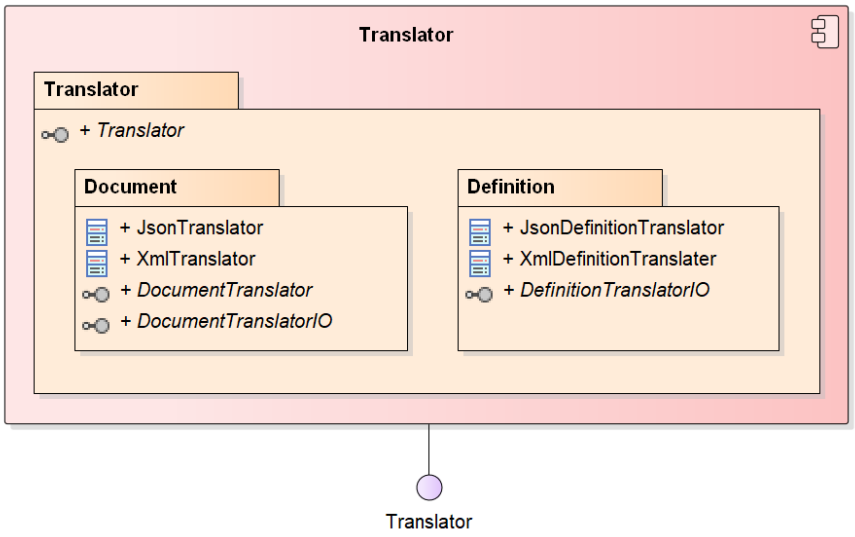
6.3.4 Translator

Komponenta Translator provádí překlad datových formátů do interních struktur a naopak. Zajišťuje načtení i zápis dat. Dále zprostředkovává překlad interní struktury do objektové reprezentace příslušného datového formátu. Dokumenty jsou překládány do struktury `XDocument` a dokumenty s X-definicemi do struktury `XDefDocument`. Komponenta podporuje překlad jak do objektového modelu, tak při proudovém zpracování. Jedná se o jedinou komponentu, která je závislá na datovém formátu.

Komponenta se skládá z následujících částí:

- `Translator` – hlavní rozhraní komponenty (více v 6.4.4),
- části pro překlad dokumentů a
- části pro překlad dokumentů s X-definicemi.

Výstupem samotného zpracování nezávisle na spuštěném režimu je interní struktura s příslušnými daty, které mají být vráceni uživateli. Z tohoto důvodu je potřeba mít pro každý formát zvolenou objektovou reprezentaci, do které je tato interní struktura přeložena a následně vrácena uživateli. Na základě důvodů zmíněných v části 5.5.1.2 jsem pro XML zvolil objektový model, který definuje knihovna JDOM. Pro formát JSON jsem zvolil objektový model definovaný JSON-P, protože se jedná o model, který by se mohl stát standardním objektovým modelem pro reprezentaci JSON dokumentů v Javě. Díky tomu by pak odpadla závislost na knihovně třetí strany.



Obrázek 6.6: Základní struktura komponenty Translator

6.3.5 X-def

X-def zajišťuje provázání všech předchozích komponent a zprostředkovává komunikaci mezi nimi. X-def vytváří tzv. fasádu nad rozhraními ostatních komponent. V případě potřeby tato rozhraní rozšiřuje o další metody a provádí mapování nových metod na metody původního rozhraní.

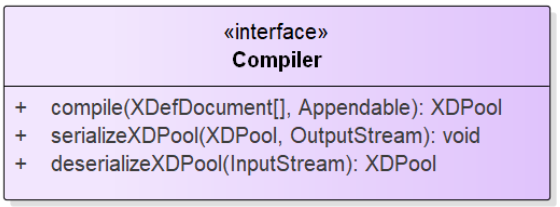
6.4 Rozhraní komponent

Jak už bylo popsáno výše, každá komponenta definuje vlastní rozhraní, prostřednictvím kterého může komunikovat s ostatními. Jméno rozhraní vždy odpovídá jménu komponenty, která ho implementuje. Byly navrženy hlavní čtyři rozhraní plus páté, které je rozhraním pro celý framework.

6.4.1 Rozhraní komponenty Compiler

Rozhraní deklaruje metody pro validaci a kompilaci X-definic. Schéma rozhraní je na obr. 6.7. Rozhraní deklaruje následující metody:

- `compile` – metoda provede kompilaci X-definic a vytvoří z nich object XDPool,
- `serializeXDPool` – metoda provede serializaci XDPoolu,
- `deserializeXDPool` – metoda provede deserializaci XDPoolu.



Obrázek 6.7: Rozhraní komponenty Compiler

V současném řešení, je možné při kompilaci předat kontejner tzv. reportér, do kterého jsou zaznamenávány chyby vzniklé během kompilace. Pro zajištění této funkcionality, obsahuje metoda `compile` parametr `reporter`, který je typu `Appendable`. Pokud nebude parametr specifikován, použije se pro zápis standardní výstup. Hlavní myšlenka tohoto přístupu, je založena na tom, že standardní výstup implementuje rozhraní `Appendable`. Díky tomu je zajištěn jednotný přístup pro předávání informací o průběhu, bez ohledu na to, zda byl reportér specifikován, či nikoliv. Odpadá tedy logika, která je v současném systému, že když není reportér specifikován, tak se chyba propaguje jako výjimka, jinak je zapsána do reportéru.

6.4.2 Rozhraní komponenty Runtime

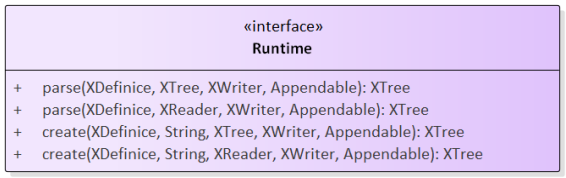
Rozhraní komponenty Runtime deklaruje metody pro proces zpracování. Jedná se o metody spojené s režimem validace a o metody spojené s režimem konstrukce. Pro režim validace je metoda `parse`, pro režim konstrukce metoda `create`. Každá metoda existuje ve dvou variantách:

- varianta pro zpracování z objektového modelu a
- varianta pro zpracování z proudu.

Společné parametry metod:

- definice – X-definice podle které probíhá zpracování dokumentu;
- vstup – buď může být typu `XTree`, nebo `XReader` (v případě konstrukčního režimu je nepovinný);
- výstup – objekt `XWriter` pro průběžný zápis (nepovinný);
- reportér – objekt pro zapisování stavů během zpracování, pro který platí stejná logika, jako pro reportér u procesu kompilace.

Metoda `create` obsahuje navíc parametr `modelName`, který definuje název modelu, od kterého má začít zpracování. Schéma rozhraní je zobrazeno na obr. 6.8.



Obrázek 6.8: Rozhraní komponenty Runtime

6.4.3 Rozhraní komponenty XScript

Schéma rozhraní komponenty je zobrazeno na obr. 6.9. Rozhraní komponenty se skládá ze třech částí:

- `XScriptEnvironment`,
- `XScriptCompiler` a
- `XScriptRuntime`.

V rámci návrhu tohoto rozhraní byl zohledněn princip CRP.

`XScriptEnvironment` deklaruje metody spojené s obsluhou runtime prostředí komponenty XScript. Metoda `setupScope` slouží k definici proměnných a funkcí, které mají být dostupné v dané části. Metoda `cancelScope` tuto definici naopak zruší.

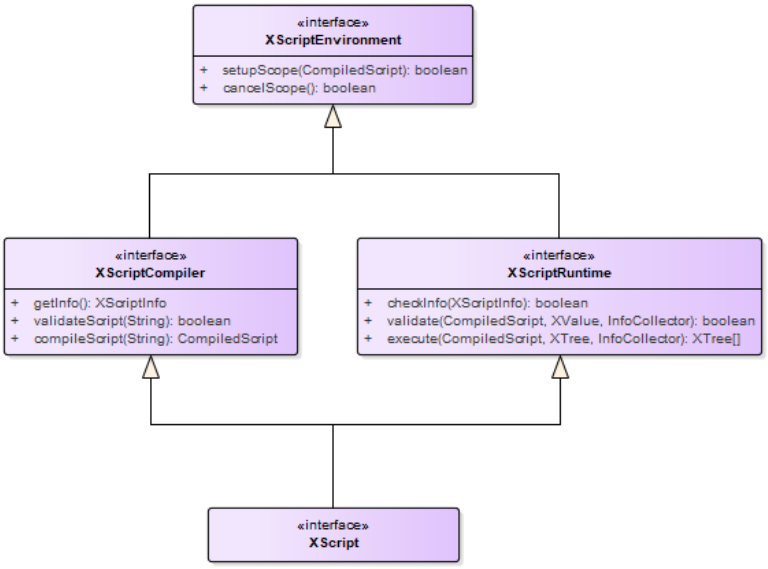
`XScriptCompiler` je část rozhraní XScriptu, na kterém je závislá komponenta Compiler. Rozhraní je potomkem `XScriptEnvironment`, protože i při kompilaci je potřeba upravovat prostředí XScriptu. Rozhraní deklaruje metody spojené s kompilací x-skriptu. Metoda `validateScript` provádí validaci jazyka x-skript a metoda `compileScript` pak jeho kompilaci. Metoda `getInfo` slouží k získání informací o použitém XScriptu během kompilace.

`XScriptRuntime` je část rozhraní XScriptu, na kterém je závislá komponenta Runtime. Rozhraní je také potomkem `XScriptEnvironment` a deklaruje metody pro provádění x-skriptu během zpracování. Metoda `checkInfo` slouží k ověření, že aktuální komponenta XScript je kompatibilní s komponentou, která byla použita ke kompilaci X-definice. Metoda `validate` slouží pro provedení kontroly typu. Metoda `execute` zajišťuje provedení příslušné akce.

6.4.4 Rozhraní komponenty Translator

Rozhraní komponenty Translator je závislé na podporovaných datových formátech. Pro každý podporovaný formát obsahuje metodu pro vytvoření překladače formátu. Aktuálně obsahuje dvě metody:

- `createXmlTranslator` a
- `createJsonTranslator`.



Obrázek 6.9: Rozhraní komponenty XScript

Rozhraní konkrétních překladačů je ale jednotné, liší se jen typem příslušející objektové reprezentaci formátu, který slouží jako návratová hodnota pro metody zajišťující překlad do objektové reprezentace. Schéma rozhraní je i s konkrétní implementací pro podporované formáty zobrazeno na obr. 6.10.

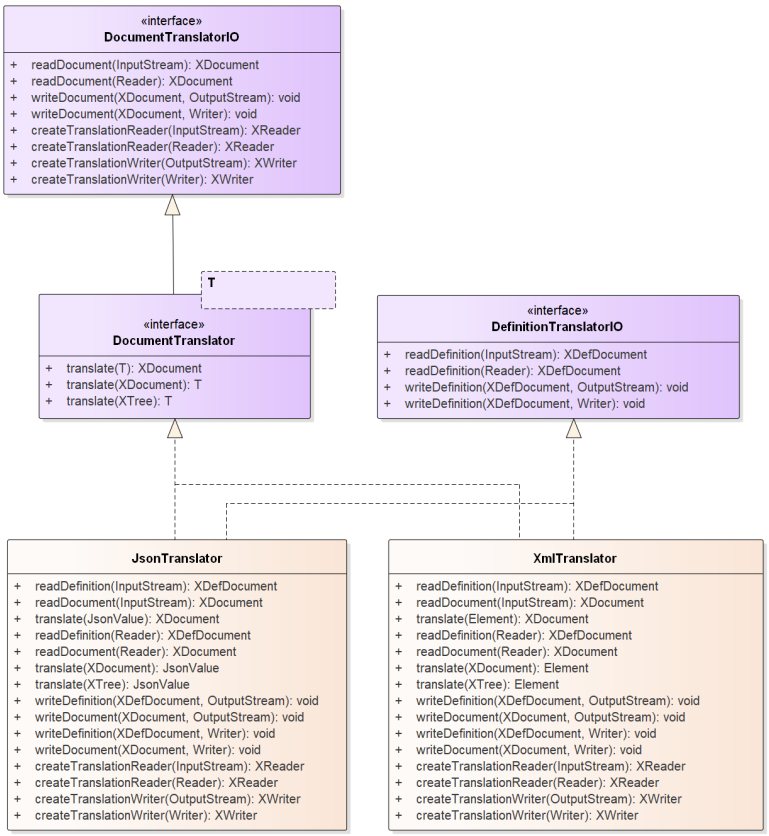
Rozhraní `DocumentTranslatorIO` resp. `DefinitionTranslatorIO` deklarují metody pro načtení a překlad vstupního dokumentu resp. dokumentu s X-definicí. Každá metoda umožňuje dvojí konfiguraci. Pokud je parametr metody datový proud (`Input/OutputStream`) určí se kódování dokumentu na základě dat[5]. Pokud je parametr metody objekt `Reader/Writer` budou data dekódovány/kódovány podle kódování v `Reader/Writer`. Pokud ale použité kódování neodpovídá kódování použitém v datech, skončí operace chybou. X-definice na rozdíl od dokumentů reprezentující data je potřeba mít v operační paměti načtené celé, proto rozhraní pro překlad dokumentů s X-definicemi nedeklaruje metody pro proudový překlad.

Rozhraní `DocumentTranslator` deklaruje metody pro zajištění překladu z objektové reprezentace formátu do interní struktury dokumentu a naopak. Nabízí i metodu pro překlad fragmentu dokumentu.

6.4.5 Rozhraní komponenty X-def

Hlavním úkolem rozhraní komponenty X-def je deklarovat metody pro spuštění procesu zpracování, které specifikují typ vstupního a výstupního datového formátu. Taková metoda má tvar `parseXtoY` resp. `createXtoY`, kde X

[5]Formát JSON podporuje také dynamické určení kódování stejně jako XML.



Obrázek 6.10: Rozhraní konkrétních překladačů

reprezentuje název vstupního formátu a Y název výstupního formátu. Rozhraní tedy deklaruje minimálně $2 * n^2$ metod, kde n je počet podporovaných datových formátů. Alternativou k tomuto přístupu bylo do názvu metody zakomponovat jen výstupní formát (je potřeba pro specifikování typu návratové hodnoty) a typ vstupního formátu předat jako parametr metody. V tomto případě by ale každá metoda obsahovala větvení na základě tohoto parametru, které by vybíralo příslušný překladač. Rozhodl jsem se pro první přístup, protože se mi zdál více názorný.

6.5 Modelové třídy

Modelové třídy slouží k reprezentaci interních struktur a definují jejich rozhraní. Jsou také použity pro komunikaci mezi komponentami.

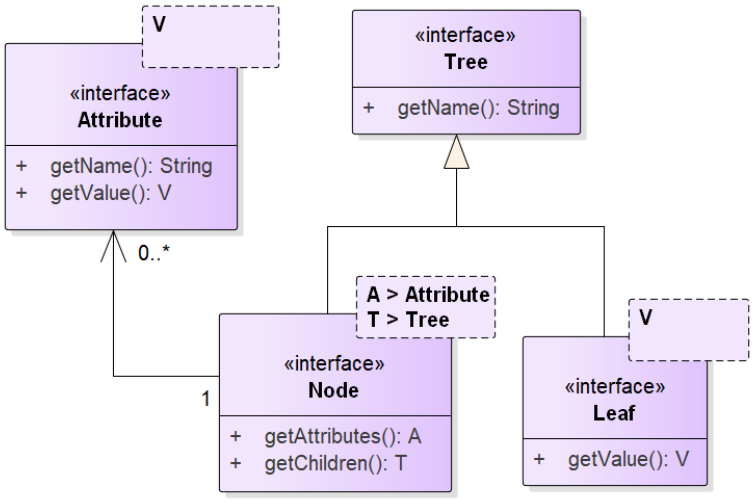
6.5.1 Localizable

Jedním z požadavků je podpora lokalizace prvku interní struktury ve zdrojových datech. Pro určení pozice ve zdrojových datech, jak pro vstupní dokumenty, tak pro X-definice, slouží rozhraní `Localizable`. Rozhraní deklaruje dvě metody:

- `getLineNumber` – metoda pro vrácení čísla řádku a
- `getColumnNumber` – metoda pro vrácení čísla sloupce.

6.5.2 Tree a Attribute

Pro definování základní stromové struktury, ze které vycházejí všechny ostatní modely založené na stromové struktuře, slouží čtveřice rozhraní. `Tree` reprezentuje stromovou strukturu a definuje její minimální rozhraní. Má právě dva potomky: `Node` a `Leaf`. Rozhraní `Node` definuje minimální strukturu pro uzly stromu (nejen pro uzly vnitřní). Rozhraní `Leaf` definuje minimální strukturu pro uzly, které mají hodnotu. Posledním rozhraním je `Attribute`, které definuje minimální strukturu pro atributy uzlů. Schéma struktury je zobrazeno na obr. 6.11.



Obrázek 6.11: Základní stromová struktura

6.5.3 XDocument a XTree

Následující modelové třídy reprezentují interní model dokumentu. Jedná se o jediný model, který není neměnný (angl. *immutable*). Možnost měnit data a strukturu modelu potřebujeme jak pro režim konstrukce, tak pro režim validace. Interní model se skládá z tříd:

- `XDocument` – reprezentuje dokument, obsahuje dva atributy: atribut `type` definuje typ vstupního dokumentu jako XML, JSON apod. a atribut `root`, který odkazuje na kořen stromu popisující vstupní dokument;
- `XTree` – reprezentuje stromovou strukturu dokumentu;
- `XNode` – reprezentuje uzel, který může mít potomky a atributy;
- `XLeaf` – reprezentuje uzel, který obsahuje hodnotu;
- `XAttribute` – třída reprezentující atribut uzlu (klíč – hodnota);
- `XValue` – třída reprezentující hodnotu.

Diagram tříd zobrazující vztahy mezi jednotlivými třídami je na obr. D.2.

Každý z podporovaných datových formátů si tento základní model rozšiřuje pro své typy. Důvod vytvoření vlastních tříd pro jednotlivé formáty je kvůli zajištění zachování struktury dokumentu. Například formát JSON může mít objekty nebo pole a pokud by se oba typy transformovaly na stejný typ, tato informace by se ztratila[6]. Pro formát XML vznikly následující třídy:

- `XmlXDocument` – reprezentuje dokument ve formátu XML,
- `XmlXNode` – reprezentuje XML element,
- `XmlXLeaf` – reprezentuje uzel s textovou hodnotu uzlu,
- `XmlXAttribute` – reprezentuje XML atribut,
- `XmlTextXValue` – reprezentuje hodnotu jak atributu, tak uzlu.

Pro formát JSON jsou to pak následující třídy:

- `JsonXDocument` – reprezentuje dokument ve formátu JSON,
- `JsonXNode` – třída reprezentující JSON element. Podle typu elementu jsou dále:
  - `JsonObjectXNode` – pro uzel reprezentující JSON objekt,
  - `JsonArrayXNode` – pro uzel reprezentující JSON pole.
- `JsonXLeaf` – reprezentuje uzel s hodnotou,
- `JsonXAttribute` – reprezentuje atribut uzlu s primitivní JSON hodnotou,
- `JsonXValue` – reprezentuje hodnotu jak atributu, tak uzlu. Hodnota může být následujících typů:

[6]Další možností bylo definovat typ prostřednictvím speciální příznaku. Já jsem se ale přiklonil k objektovému přístupu.

- `JsonStringXValue` – hodnota typu řetězec,
- `JsonNumberXValue` – hodnota typu číslo,
- `JsonBooleanXValue` – hodnota nabývající buď `true`, nebo `false`.

6.5.4 XDefDocument a XDefTree

I když je X-definice také dokument v požadovaném formátu, bylo nutné pro reprezentaci vytvořit strukturu vlastní. Při návrhu jsem narazil hned na dva důvody k oddělení od předchozí struktury.

Prvním z důvodů byla potřeba podpory speciálního atributu `xd:script`. V XML dokumentech je toho docíleno pomocí jmenných prostorů, ale v JSON žádný podobný koncept k jmenným prostorům neexistuje. U JSON toto odlišení závisí na autorovi formátu X-definice. Aby vnitřní struktura nebyla pevně svázána s formátem X-definice pro JSON dokumenty, který v tuto chvíli nebyl ještě definovaný, byla struktura pro X-definici oddělena. Při realizaci mapování je pak na autorovi mapování, jakou část dokumentu považuje za skript.

Druhý důvod byl ten, že různé formáty mají různá omezení pro svůj formát. Například jak bylo popsáno v podkapitole 2.2 o formátu JSON, specifikace připouští opakování jmen v rámci jednoho objektu. Z tohoto důvodu tedy model pro X-definici obsahuje další dva atributy, které určují povolené akce a povolené možnosti výskytů a lze je tedy nastavit na základě formátu během překladu.

Model pro X-definici je rozšířen o prvky popsané rozhraními `Scriptable` a `XDefinitionSpecifier`, kde rozhraní `Scriptable` určuje, že daný objekt obsahuje část se skriptem X-definice a rozhraní `XDefinitionSpecifier` specifikuje povolené akce a výskyty. Stromová struktura pak prakticky jen kopíruje předchozí strukturu, proto ji nebudu dále vysvětlovat. Celé schéma vnitřní struktury pro X-definici je zobrazené na obr. D.3.

6.5.5 XDefinition a XDefinitionTree

Následující třídy reprezentují strukturu zkompilované X-definice. Ta je velmi podobná struktuře, která je použita pro reprezentaci X-definice v současném řešení. Všechny třídy implementují rozhraní `Serializable` pro podporu Java objektové serializace. `XDefinition` reprezentuje zkompilovanou X-definici. Obsahuje informaci o použitém XScriptu během kompilace a referenci na kořen stromové struktury X-definice (strukturu `XDefinitionTree`). Stromová struktura se skládá z `XDefinitionNode` reprezentující uzel stromu a `XDefinitionLeaf` reprezentující list stromu obsahující hodnotu. Atribut uzlu je reprezentován `XDefinitionAttribute`. Stromová struktura i atribut implementují rozhraní `Scripted`, které deklaruje metody vycházejí z formátu skriptu X-definice. Metody rozhraní `Scripted` a jeho předků:

- `getActions` – vrátí mapu typu `Event-CompiledScript`, mapa obsahuje události, které byly definovány ve skriptu a ke každé události zkompilovaný x-skript;
- `getOption` – vrátí množinu možností (Options), které byly definovány ve skriptu;
- `getSetupScopeCommand` – vrátí zkompilovaný x-skript, který reprezentuje deklaraci ve skriptu X-definice;
- `getOccurrence` – vrátí typ výskytu;
- `getMinBound` – vrátí dolní hranici počtu povolených výskytů prvku;
- `getMaxBound` – vrátí horní hranici počtu povolených výskytů prvku.

`XDefinitionLeaf` a `XDefinitionAttribute` obsahují navíc zkompilovanou část x-skriptu, která slouží pro validaci typu hodnoty. Tato část je zapsána jako atribut `value` příslušné třídy. Zkompilovaný x-skript je reprezentován rozhraním `CompiledScript`, které je pouze tzv. marked interface. Implementace tohoto rozhraní záleží na tvůrci XScriptu. Schéma celé struktury pro zkompilovanou X-definici je zobrazeno na obr. D.2.

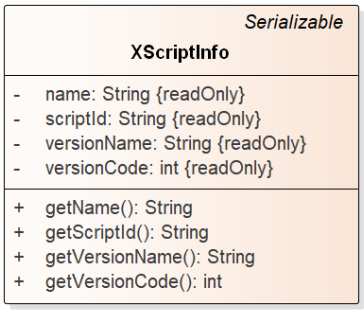
6.5.6 XEvent

Proudový režim je realizován jako event-based pull model. Pro komunikaci slouží několik událostí, které odpovídají částem interní struktury použité pro reprezentaci dokumentu. Každá událost obsahuje informaci o pozici ve zdrojovém souboru. Jedná se o následující události:

- `StartDocumentEvent` – signalizuje začátek dokumentu,
- `EndDocumentEvent` – signalizuje konec dokumentu,
- `StartNodeEvent` – signalizuje začátek uzlu a obsahuje jeho jméno,
- `EndNodeEvent` – signalizuje konec uzlu a obsahuje jeho jméno (je odeslán po zpracování všech potomků uzlu),
- `AttributeEvent` – signalizuje atribut uzlu a obsahuje jeho jméno a hodnotu atributu,
- `ValueEvent` – signalizuje list, resp. jeho hodnotu a obsahuje ji,
- `NotProcessedEvent` – událost je odeslána v případě, kdy se ve zdrojových datech nacházejí části data, která nejsou důležitá pro zpracování, ale chceme zajistit jejich zachování.

6.5.7 XScriptInfo

`XScriptInfo` obsahuje informace o použitém XScriptu. Kontejner se zkompilovanými X-definicemi může být serializován a použit jindy. Takto vytvořený kontejner, ale obsahuje X-definice, které byly zkompilovány s použitím určité verze a druhu XScriptu. Zatím co v aplikaci, která tento kontejner využívá, může být úplně jiná verze, či jiný typ. Je tedy nutné mít přehled o použitém XScriptu při kompilaci. Pro výměnu těchto informací slouží právě tento model.



Obrázek 6.12: Model XScriptInfo

Atribut `name` obsahuje jméno XScript. Atribut slouží pro definování smysluplného jména, které by mohlo být zobrazeno uživateli, např. `X-python`. Atribut `scriptId` obsahuje jasný a unikátní identifikátor XScriptu např. package name jako `org.xdef.python`. Atribut `versionName` obsahuje označení verze XScriptu např. `1.0-SNAPSHOT`. A atribut `versionCode` obsahuje jasný identifikátor verze např. počet verzí (commitů) v GIT repozitáři obsahující daný XScript jako např. `2048`.

6.6 Rozšiřitelnost

Takto navržená architektura systému obsahuje minimálně dvě roviny rozšiřitelnosti. Díky jednotnému vnitřnímu modelu můžeme snadno přidat podporu dalšího formátu např. YAML (YAML Ain't Markup Language) apod. Pro přidání nového formátu je potřeba navrhnout vhodné mapování na vnitřní strukturu a přidat do komponenty Translator příslušnou implementaci. Přidání formátu ovlivní jen dvě komponenty systému a to již zmíněnou komponentu Translator a dále pak komponentu X-def, ale jen na místě definice rozhraní systému a to jen ve smyslu jeho rozšíření, což zaručuje zpětnou kampatibilitu. Pro přidání nového formátu jsou nutné pouze tyto kroky:

1. volba vhodné objektové reprezentace (slouží pro návratovou hodnotu z metod parse a validate),

2. implementace mapování vstupního formátu do vnitřní struktury a naopak,

3. rozšíření rozhraní komponenty Translator o metodu umožňující vytvořit překladač pro daný typ a

4. rozšíření vstupního rozhraní systému o metody spouštějící zpracování nového formátu.

Za druhou rovinu rozšiřitelnosti lze považovat rozšiřitelnost jazyka x-skript, či přidání podpory dalšího jazyka jako třeba Ruby, či vlastního jazyka. Díky nezávislosti systému na XScriptu rozšíření v tomto směru vůbec neovlivňuje systém.

Za zmínku ještě stojí možné rozšíření jazyka r-skript, který je ale nejvíce provázaný v celém systému. S jazykem r-skript se pracuje ve všech komponentách kromě komponenty XScriptu. Rozšíření by tedy znamenalo mnohem větší zásah, nicméně pouze na úrovni daných komponent, protože rozhraní komponent je přesně pro tento případ navrženo obecně.

6.7 Metodika přechodu na novou architekturu

Na základě analýzy stávajícího systému jsem po konzultaci s vedoucím práce dospěl k závěru, že bude mnohem výhodnější navrhnout celý systém od znovu a pro jeho realizaci využít znalosti z již existujícího systému. Metodiku přechodu bych shrnul do těchto bodů:

1. seznámení se s navrženou architekturou,

2. nalezení bodů dekompozice stávajícího systému,

3. nalezení přepoužitelných částí systému,

4. použití/reimplementace existujících částí a

5. otestování nového systému vůči původnímu systému.

Díky nezávislosti jednotlivých komponent, může přechod na novou architekturu probíhat paralelně. Jediné limity nastanou při nutnosti rozšíření navrženého rozhraní mezi komponentami.

Kapitola 7

Implementace

V této kapitole se věnuji implementaci nového frameworku X-definice. Na začátku představím strukturu projektu a popíšu její jednotlivé části. Dále se věnuji představení konceptů a technologií, které jsem v implementaci použil. Na závěr představím nejdůležitější části implementace. A jako poslední popíši, jak vypadá výstup implementace a jeho použití.

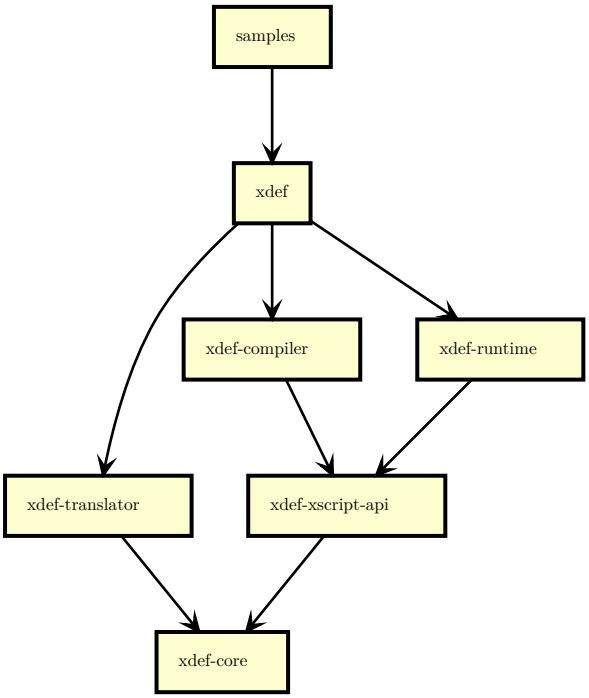
7.1 Vymezení implementované části

První částí implementace je vytvoření základní struktury celého projektu. Součástí této části je také vytvoření navržených rozhraní jednotlivých komponent a implementace modelů pro komunikaci mezi komponentami systému.

Druhá část implementace se týká implementace komponenty Translator. Důvodů výběru této komponenty bylo hned několik. Jedná se o jedinou novou komponentu systému, takže dává smysl se zabývat přímo jít. Ostatní komponenty již v systému existují, byť jen na logické úrovni. Další z důvodu je, že celá architektura je postavena nad jednotným vnitřním modelem a tvorba tohoto modelu probíhá právě v komponentě Translator. Implementace této komponenty má tedy ukázat, že je možné, byť s určitými výjimkami definovat mapování a následný převod datových formátů do vnitřní struktury a naopak.

7.2 Struktura projektu

Dle jednoho z požadavků má být celý systém postaven nad sestavovacím nástrojem Maven. Systém je tedy navržen jako více modulární Maven projekt. Pro každou komponentu systému existuje jeden Maven modul. Komponenty pak mají jasně definované závislosti mezi sebou. Důvod rozdělení komponent do modulů a ne jen do různých balíčků v jednom modulu je ten, že první přístup je mnohem striktnější a neumožňuje tedy přístup ke všem třídám odkudkoliv. Mimo modulů příslušejících jednotlivým komponentám existuje



Obrázek 7.1: Schéma závislostí modulů v projektu

v projektu ještě několik dalších modulů. O jejich významu se zmíním níže. Graf závislostí mezi moduly je zobrazen na obr. 7.1. V návrhu struktury projektu jsem se inspiroval v projektu OkHttp[7].

7.2.1 xdef

Xdef je hlavní modul celého frameworku. Definuje jeho rozhraní a díky vazbám na ostatní moduly umožňuje sestavení výsledného .jar souboru. Package name modulu je `org.xdef`. Modul obsahuje třídy a rozhraní, které zpřístupňují rozhraní dalších modulů, nebo které rozhraní modulů rozšiřují, takže obsahují logiku pro mapování komplexnějšího rozhraní na jednodušší viz. příklad 7.1.

7.2.2 xdef-core

Modul xdef-code v sobě obsahuje veškeré modelové třídy, které slouží na komunikaci mezi komponentami, dále může také obsahovat různé pomocné funkce, které se mohou využívat napříč moduly projektu. Package name modulu je `org.xdef.core`. Hlavní důvod zavedení tohoto modulu je, aby nevznikal cyklus na grafu závislostí a byla umožněna komunikace mezi komponentami.

[7] `https://github.com/square/okhttp`

```
// metoda rozhraní deklarovaná komponentou Compiler
fun deserializeXDPool(poolStream: InputStream) {...}

// rozšíření rozhraní
fun deserializeXDPool(poolFile: File)
  = poolFile.inputStream().use { deserializeXDPool(it) }
```

Zdrojový kód 7.1: Ukázka rozšíření rozhraní

7.2.3 xdef-translator

xdef-translator je modul pro komponentu Translator. Obsahuje veškerou implementaci spojenou s překladem dokumentu, či dokumentu X-definice do interní reprezentace a naopak. Dále obsahuje implementaci spojenou s načítáním a zápisem dokumentů. Jedná se o jediný modul, s výjimkou modulu xdef, který je závislý na formátu dokumentů. Obsahuje příslušné závislosti na externí knihovny pro práci s formáty dokumentů. Obsahuje potomky třídních modelů interních struktur, pro zajištění speciálního zpracování, např. `JsonObjectXNode`, který rozšiřuje klasický `XNode` o informaci, která je závislá na formátu vstupního dokumentu. Další z důležitých součástí je výčet s podporovanými formáty dokumentů. Package name modulu je `org.xdef.translator`. Modul je závislý pouze na xdef-core modulu.

7.2.4 xdef-compiler

xdef-compiler je modul pro komponentu Compiler. Obsahuje deklaraci rozhraní této komponenty a dále pak veškerou implementaci spojenou s kompilací X-definic. Package name modulu je `org.xdef.compiler`. Mimo závislosti na xdef-core modulu, závisí na modulu xdef-xscript-api.

7.2.5 xdef-runtime

xdef-runtime je modul pro komponentu Runtime. Obsahuje deklaraci rozhraní této komponenty a pak veškerou implementaci spojenou s procesem zpracování v obou režimech. Package name modulu je `org.xdef.runtime`. Mimo závislosti na xdef-core modulu, závisí ještě na modulu xdef-xscript-api.

7.2.6 xdef-xscript-api

xdef-xscript-api obsahuje deklaraci rozhraní pro komponentu XScript. Modul je spojovacím prvkem, který umožňuje propojení frameworku X-definic s konkrétní implementací XScriptu. Pro tento důvod je jednou z hlavních součástí modulu, mimo samotného rozhraní, rozhraní `XScriptFactory`, které deklaruje metody pro vytvoření nové instance XScriptu. Package name modulu je `org.xdef.script`. Modul je závislý pouze na xdef-core modulu.

7.2.7 samples

Modul samples obsahuje příklady použití frameworku. U open-source projektů bývá standardem, že obsahují modul s příklady, který je součástí přímo projektu s implementací. Obsahuje tři podmoduly pro ukázku použití frameworku X-definic ve validačním módu, v konstrukčním módu a ukázku použití v projektu v Javě.

7.3 Programovací jazyk

Na základě analýzy v podkapitole 5.3 o možnostech tvorby aplikací pro JVM s kompatibilitou s jazykem Java, jsem se rozhodl zvolit programovací jazyk Kotlin. Všechny analyzované jazyky nabízejí kompatibilitu s jazykem Java, nelze se tedy při výběru jazyka rozhodnout, podle toho, co od něho požadujeme. Proto jsem se při výběru jazyka rozhodoval na základě zkušeností s daným jazykem. Největší zkušenost mám s jazykem Java a Kotlin. Kotlin jsem vybral také z důvodu, že ho považuji za modernější jazyk a jelikož je to oblíbený jazyk v komunitě kolem JVM, může to napomoct k budoucímu rozvoji frameworku.

7.3.1 Sealed class

Jedním ze specifik jazyka Kotlin jsou tzv. Sealed class. Sealed třídy rozšiřují princip výčtového typu na celou hierarchii tříd. Položky výčtu obsahují v projektu jednu jedinou instanci. Tento nedostatek Sealed třídy eliminují. Nabízejí možnost definování přesně omezeného počtu typů, kterých můžeme vytvořit libovolný počet instancí. [12]

```
sealed class Expr
data class Const(val number: Double) : Expr()
data class Sum(val e1: Expr, val e2: Expr) : Expr()
object NotANumber : Expr()

...
fun eval(expr: Expr): Double = when(expr) {
  is Const -> expr.number
  is Sum -> eval(expr.e1) + eval(expr.e2)
  NotANumber -> Double.NaN
  // `else` větev není potřeba, protože všechny
  // přímí potomci jsou vyjmenováni
}
```

Zdrojový kód 7.2: Ukázka Sealed tříd [12]

Tento koncept jsem použil v implementaci modelových tříd a událostí. Využívám objektové reprezentace před reprezentací, kde se od sebe jednotlivé typy rozlišují mocí příznaku, který je definován jako jeden z atributů objektu.

7.4 Použité knihovny a použité rozhraní

V této části představím knihovny, které jsem v implementaci použil, a popíši jejich základní rozhraní.

7.4.1 Kotlin Standard library

Součástí jazyka Kotlin je jeho rozsáhlá knihovna standardních funkcí a tříd. Knihovna například obsahuje vlastní implementaci některých základních kolekcí, které jsou standardně součástí JDK. V projektu je použita standardní knihovna Kotlinu pro JDK verze 8.

7.4.2 JDOM

Pro práci s objektovým modelem XML dokumentů jsem v implementaci použil knihovnu JDOM. Jak bylo zmíněno v části 5.5.1.2 knihovna neobsahuje vlastní parser, takže pro parsování XML dokumentu je použit standardní SAX parser z JDK. Pro podporu lokalizace XML objektu ve zdroji dat, je při parsování použita implementace `LocalizedJDOMFactory`.

```
fun readDocument(input: Reader) {
  val document = SAXBuilder()
      .apply { jdomFactory = LocatedJDOMFactory() }
      .build(input)
}

fun writeDocument(document: Document, output: Writer) {
  XMLOutputter().output(document,output)
}
```

Zdrojový kód 7.3: Použití JDOM rozhraní pro parsování dokumentu

7.4.3 StAX

Pro čtení XML dokumentů v proudovém režimu jsou použity komponenty ze StAX API. Pro čtení je použita komponenta `XMLEventReader` a pro zápis odpovídající writer a to `XMLEventWriter`. Obě komponenty jsou vytvářeny pomocí odpovídající tovární třídy. Příklad vytvoření komponent je na ukázce zdrojového kódu 7.4.

`XMLEventReader` parsuje vstupní dokument, který může být komponentě předán v mnoha různých formách např. jako `Reader`, `InputStream` nebo soubor. Na základě dat vytváří příslušné události. Každá událost má svůj typ, pozici ve zdrojovém souboru a podle typu obsahuje příslušná data. Události jsou následujících typů:

- `START_DOCUMENT` – událost identifikuje začátek zpracování dokumentu a obsahuje informace o dokumentu jako například kódování, verzi XML apod.;
- `END_DOCUMENT` – událost identifikuje konec zpracování dokumentu;
- `START_ELEMENT` – událost identifikuje začátek zpracování elementu a obsahuje jeho jméno, atributy a deklarace jmenných prostorů;
- `END_ELEMENT` – událost identifikuje konec elementu a obsahuje jeho jméno;
- `CHARACTERS` – událost identifikuje textová data a obsahuje tyto data;
- `PROCESSING_INSTRUCTION` – událost identifikuje procesní instrukci;
- `COMMENT` – událost identifikuje komentář a obsahuje ho;
- `ENTITY_REFERENCE` – událost identifikuje odkaz na XML entity;
- `DTD` – událost identifikuje textová data a obsahuje tyto data;
- `NOTATION_DECLARATION` – událost identifikuje deklaraci XML notace;
- `ENTITY_DECLARATION` – událost identifikuje deklaraci XML entity.

Na základě předané události `XMLEventWriter` extrahuje data z události a provede zápis příslušné informace v odpovídajícím formátu na výstup, který může být opět předán mnoha způsoby.

7.4.4 JSON-P

Pro objektový model JSON dokumentu jsem zvolil knihovnu JSON-P. Knihovna obsahuje jen rozhraní, která deklarují metody pro práci s formátem JSON v jazyce Java. Obsahuje i rozhraní, která určují podobu objektového modelu JSON dokumentu. Názvy jednotlivých rozhraní objektového modelu pro JSON odpovídají názvům prvků dle specifikace formátu viz. podkapitola 2.2:

- `JsonValue` – rozhraní reprezentující JSON hodnotu,
- `JsonObject` – rozhraní reprezentující JSON objekt,
- `JsonArray` – rozhraní reprezentující JSON pole,
- `JsonString` – rozhraní reprezentující JSON řetězec,
- `JsonNumber` – rozhraní reprezentující JSON číslo.

```
// vytvoření StAX parseru
val inputFactory: XMLInputFactory
        = XMLInputFactory.newFactory()
val reader: XMLEventReader
        = inputFactory.createXMLEventReader(/*vstup*/)

// vytvoření StAX generatoru
val outputFactory: XMLOutputFactory
        = XMLOutputFactory.newFactory()
val writer: XMLEventWriter
        = outputFactory.createXMLEventWriter(/*výstup*/)

// průchod vstupního dokumentu
while (reader.hasNext()) {
  // vyžádání další události
  val event: XMLEvent = reader.nextEvent()

  // zjištění typu události
  event.eventType

  // zápis události
  writer.add(event)
}
```

Zdrojový kód 7.4: Ukázka StAX rozhraní pro zpracování dokumentu

7.4.5 Jackson

Pro zpracování dokumentů ve formátu JSON jsem použil knihovnu Jackson, respektive jen její rozhraní pro proudové zpracování. Pro vytvoření prvků pro práci s dokumentem slouží komponenta `JsonFactory`. Ta nabízí rozhraní pro vytvoření dalších komponent. Pro parsování dokumentu slouží komponenta `JsonParser`. Ta umožňuje získání potřebných informací jako aktuální token, pozici tokenu ve vstupním souboru apod.

Jackson Streaming API je token-based pull parser, takže parser předává informaci jen o aktuálním tokenu. Na uživateli pak je, aby si vyžádal další informace. Tokeny jsou následujících typů:

- `START_OBJECT` – identifikuje začátek JSON objektu,
- `END_OBJECT` – identifikuje konec JSON objektu,
- `START_ARRAY` – identifikuje začátek JSON pole,
- `END_ARRAY` – identifikuje konec JSON pole,

- `FIELD_NAME` – identifikuje jméno hodnoty,
- `VALUE_STRING` – identifikuje řetězec,
- `VALUE_NUMBER_INT` – identifikuje celé číslo,
- `VALUE_NUMBER_FLOAT` – identifikuje desetinné číslo,
- `VALUE_TRUE` – identifikuje `true` hodnotu,
- `VALUE_FALSE` – identifikuje `false` hodnotu,
- `VALUE_NULL` – identifikuje `null` hodnotu,
- `VALUE_EMBEDDED_OBJECT` – identifikuje vnořený objekt (při normálním zpracování by tento token neměl být vrácen),
- `NOT_AVAILABLE` – speciální chybový token.

Pro získání pozice aktuálního tokenu ve vstupních datech slouží metoda parseru `getCurrentLocation`, která vrací objekt `JsonLocation` obsahující příslušné informace.

Pro zápis dokumentu slouží komponenta `JsonGenerator`, která definuje metody jak pro zápis tokenů, tak pro zápis dat. Ukázka rozhraní je ve zdrojovém kódu E.3.

7.4.6 log4j Kotlin API

Za logovací framework jsem zvolil Apache Log4j verze 2, respektive jeho rozšíření pro Kotlin. Pro přidání reference na `logger` objekt stačí přidat mezi předky třídy rozhraní `Logging` jako např. na ukázce 7.5. Rozhraní loggeru je standardní. Nabízí šest úrovní závažnosti logu. Pro vytvoření zprávy logu, lze použít lambda funkce, takže v případě, že daný level, nemá být logován, odpadá režie spojená s vytvářením zprávy logu.

```
class JsonXWriter(writer: Writer) : XWriter, Logging
```

Zdrojový kód 7.5: Použití Apache Log4j Kotlin API

7.5 Implementace rozhraní JSON-P

Při tvorbě vlastní implementace JSON-P rozhraní, jsem vycházel z implementace, která je součástí Java EE od GlassFish. Pro zajištění podpory lokalizace ve zdrojových datech implementuje každá třída rozhraní `Localizable`. Implementace obsahuje následující třídy:

- `LocalizedJsonObject` – reprezentuje JSON objekt,

- `LocalizedJsonArray` – reprezentuje JSON pole,
- `LocalizedJsonString` – reprezentuje JSON řetězec,
- `LocalizedJsonNumber` – reprezentuje JSON číslo,
- `LocalizedJsonBoolean` – reprezentuje logickou hodnotu,
- `LocalizedJsonNull` – reprezentuje JSON `null` hodnotu.

JSON-P pro logické hodnoty a pro hodnotu `null` obsahuje implementaci, kde každá z hodnot je singleton. Z důvodu přidání informací o pozici ve zdroji dat, bylo potřeba vytvořit pro každou z hodnot speciální implementaci.

7.6 Implementace komponenty Translator

Komponenta Translator obsahuje dvě hlavní části. Jedná se o části související s typem zpracování:

- překlad vstupního dokumentu z objektové reprezentace do interní struktury a naopak,
- překlad v proudovém režimu.

Překlad do interní struktury se provádí jak pro klasické dokumenty, tak pro dokumenty s X-definicemi. Překlad obou druhů dokumentů si jsou velmi podobné, jen překlad probíhá do jiných struktur. Jelikož JSON dokument neobsahuje žádné speciální informace, které bychom chtěli v datech zachovat, i když je při zpracování nevyužíváme, je tento překlad velmi přímočarý. Pro každý typ JSON hodnoty existuje příslušný objekt, který ji reprezentuje, a při překladu je jen naplněn odpovídajícími daty. U překladu XML dokumentů je situace mírně odlišná. XML obsahuje mnoho informací, které nebudeme zpracovávat (např. komentáře, procesní instrukce apod.). Pokud ale tyto informace chceme zachovat, je potřeba je při překladu přenést i do interní struktury. Pro jednoduchost jsem proto třídy, které nesou informace z XML dokumentů, navrhl tak, že jejich hlavní atribut je odpovídající objekt z jejich objektové reprezentace a části, které nás zajímají, jsou namapovány na metody rozhraní interní struktury.

Překlad v proudovém režimu se týká jen dokumentů, nikoliv X-definic, protože při následném zpracování potřebujeme mít v paměti načtenou X-definici celou. V případě potřeby ho lze ale snadno doimplementovat. Pro překlad v proudovém režimu jsou stěžejní dvě rozhraní. Rozhraní `XReader` deklaruje metody pro čtení vstupního dokumentu. `XReader` je navržený jako iterátor, takže lze jednoduše procházet celým dokumentem.

Druhým rozhraním je `XWriter`, který deklaruje metody pro zápis události (`XEvent`) do výstupního souboru v příslušném datovém formátu.

```
interface XReader : Iterator<XEvent>, AutoCloseable, Closeable {

  /**
   * Vrátí false pokud jsme nakonci dokumnetu
   * jinak true
   */
  override fun hasNext(): Boolean

  /**
   * @return Vrátí následující událost
   */
  override fun next(): XEvent

  /**
   * @return Vrátí aktuální událost
   */
  fun peek(): XEvent

  /**
   * Uzavře vstupní stream
   */
  override fun close()
}
```

Zdrojový kód 7.6: Rozhraní XReader

7.6.1 Vybraná mapování

Pro překlad datového formátu do interní struktury bylo potřeba navrhnout vhodné mapování mezi prvky datového formátu a prvky interní struktury. V rámci implementace mapování X-definice ve formátu JSON, jsem hodnotu jménem `xd:script` považoval za část, která slouží pro zapsání skriptu X-definice.

7.6.1.1 XML

Pro XML není mapování nijak zvláštní, neboť interní struktura se velmi podobá struktuře objektového modelu pro XML. Element je mapován na `XmlXNode`. Jeho atributy jsou mapovány na `XmlXAttribute`, kde hodnota je typu `XmlTextXValue`. Textové hodnoty jsou mapovány na strukturu `XmlXLeaf`, kde hodnota je opět `XmlTextXValue`.

```
interface XWriter : AutoCloseable, Closeable, Flushable {

  /**
   * Přeloží událost a zapíše ji na výstup
   * @param event Událost
   */
  fun writeEvent(event: XEvent)

  /**
   * Vynutí si zapsání dat z bufferu na výstup
   */
  override fun flush()

  /**
   * Uzavře používané zdroje
   */
  override fun close()
}
```

Zdrojový kód 7.7: Rozhraní XWriter

7.6.1.2 JSON

U JSON je situace mírně rozdílná. První problém je, že se v JSON vyskytují prvky, které nemají žádné jméno, proto bylo potřeba jméno uměle vytvořit. Pro kořenový prvek je použito označení `org.xdef.root`. Pro prvky pole se jméno skládá z kombinace jména pole a pozice prvku v poli, která je od jména oddělena tečkou např. `myArray.5`. Druhým problémem bylo určit, jaký prvek bude mapován na uzel, jaký na atribut a jaký na hodnotu. JSON objekt je mapován na `JsonObjectXNode`. Všechny prvky objektu, které mají primitivní hodnotu tzn. řetězec, číslo, `true`, `false` a `null`, jsou namapovány na atributy. Jméno prvku je jméno atributu a hodnota prvku odpovídá hodnotě atributu. Pole je mapováno na `JsonArrayXNode` a prvky pole pak na `JsonXLeaf` s příslušnou hodnotou.

Při pracování v proudovém režimu je mapování obdobné, jen se provádí na mapování příslušné události, která reprezentuje danou část.

7.6.2 Limity implementace

Limity implementace lze rozdělit do dvou typů:

1. limity způsobené datovým formátem a
2. limity vyplývající ze zvoleného mapování.

Ad 1 je jedním z limitů převod JSON na XML. Jméno u JSON prvků dovoluje použití prakticky libovolných znaků, zatím co u XML je mnoho znaků, které se ve jménu nemohou vyskytnout. Dalším problémem jsou pak konverze bílých znaků. Je potřeba určit, kdy jsou pro nás bílé znaky relevantní a kdy ne.

Ad 2 je první problém spojen s konverzí JSON na XML v proudovém režimu. Mapování nezaručuje, že všechny události, které přísluší události o atributu, budou poslány ihned po sobě. U XML se ale atributy zapisují jako součást otevíracího tagu elementu. Proto je nutné si v paměti udržovat celý element a to nemusí být v každé situaci možné. Druhý problém je spojen s mapováním pole, které je součástí X-definice. V tomto případě se v poli může vyskytnout každý typ hodnoty pouze jednou, protože mapování nedává žádný mechanismus, jak například určit, který ze dvou modelů pro objekt má popisovat zrovna tento objekt ve vstupních datech (u XML se to určuje podle jména, které ale v tomto případě nemáme).

7.7 Hlavní rozhraní

Celý framework je reprezentován třídou `XDef`. Třída `XDef` obsahuje jednak všechny metody, které poskytují jednotlivé komponenty a mají být zpřístupněné i vně frameworku, a také metody, které definuje třída sama a slouží k rozšíření metod příslušných komponent. To nabízí velké množství možných kombinací, proto je vhodné celou tuto třídu generovat.

Pro jednodušší vytvoření instance třídy `XDef` jsem naimplementoval třídu `XDef.Builder`, která slouží k prvotní inicializaci a vytvoření instance třídy `XDef`, tedy celého frameworku. Builder zajišťuje vytvoření instance, která již obsahuje kolekci X-definic. Základní rozhraní třídy `XDef.Builder` je zobrazeno v ukázce kódu E.4.

7.8 Publikace a použití

Celý projekt je 6 Maven modulů. Po kompilaci aplikace vznikne pro každý modul samostatný .jar soubor a .pom soubor, který obsahuje seznam závislostí modulu. Použití takto nakonfigurované knihovny nenese na projektech, které používají nějaký build systém, jako je třeba Maven, Gradle apod., žádná omezení, protože tento systém dokáže chybějící závislosti doinstalovat. Určitá omezení nastávají ještě před samotným použitím a to v momentě, kdy chceme celý framework distribuovat. V tomto případě musíme provést distribuci každého artefaktu zvlášť.

Jelikož tato konfigurace nepřináší žádné výhody, rozhodl jsem se build knihovny nakonfigurovat tak, aby vytvářela tzv. fatjar. Výstupem kompilace jsou jen dva artefakty. První artefakt v sobě obsahuje všechny zdrojové soubory z modulů xdef, xdef-compiler, xdef-runtime, xdef-translator a xdef-core.

Druhý artefakt je pak samotný modul xdef-xscript-api a xdef-core modul. Tento artefakt je potřeba, aby byla zajištěna podpora tvorby vlastních implementací komponenty XScript.

Použití knihovny v projektu znamená přidání závislosti na samotnou knihovnu a pak na implementaci XScriptu, kterou chceme v projektu používat. Přidání frameworku mezi závislosti Maven projektu může vypadat jako na ukázce 7.8.

```
<dependencies>
    <dependency>
        <groupId>org.xdef</groupId>
        <artifactId>xdef</artifactId>
        <version>1.0.0</version>
    </dependency>
    <dependency>
        <groupId>org.xdef</groupId>
        <artifactId>xscript-java</artifactId>
        <version>1.0.0</version>
    </dependency>
    ...
</dependencies>
```

Zdrojový kód 7.8: Ukázka přidání závislostí v Maven projektu

Kapitola 8

Testování

V této kapitole se věnuji testování vytvořené implementace. Představím druhy testů a jejich pokrytí jednotlivých částí implementace. Stručně představím frameworky a knihovny, které jsem pro vytvoření testů použil.

8.1 Použité frameworky a knihovny

8.1.1 Kotlintest

Kotlintest je framework pro tvorbu testů v jazyce Kotlin. Framework je založen na JUNIT5. Framework nabízí mnoho stylů pro psaní testů. Jednou z jeho výhod je, že obsahuje řadu implementovaných funkcí, které lze použít k testování kolekcí, typů objektů apod. Jednou z hlavních výhod je pak definování testů, které při správném pojmenování obsahují všechny informace o provedeném testu. [14]

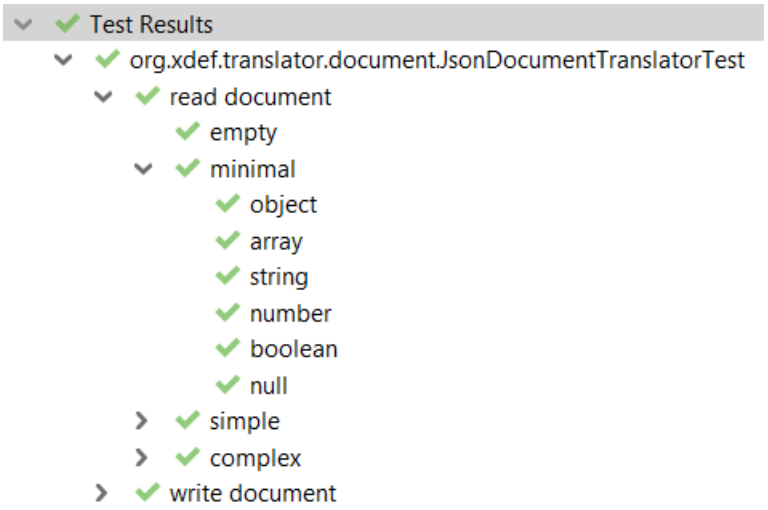
Pro tvorbu testů jsem použil styl `FreeSpec`, který umožňuje psaní vnořených testů. Výstup z testů spuštěných v IntelliJ IDEA je zobrazen na obr. 8.1.

8.1.2 XMLUnit

XMLUnit je knihovna pro Javu a .NET na testování XML dokumentů. Knihovna nabízí podporu pro testování podobnosti XML dokumentů. [40]

8.1.3 JSONAssert

Knohovna JSONAssert nabízí podporu pro testování podobnosti JSON dokumentů. [27]



Obrázek 8.1: Ukázka výstupu KotlinTest frameworku

8.2 Jednotkové testy

Jednotkové testy jsou základní technikou na zajištění kvality a správnosti softwarového produktu. Jednotkové testy typicky pokrývají testování funkcionality na úrovni jednotlivých funkcí resp. metod.

8.2.1 Pokrytí testy

Jednotkové testy se nacházejí v adresáři test. Testy pokrývají překlad dokumentu do vnitřní struktury a překlad vnitřní struktury do výstupního dokumentu a to pro oba druhy zpracování objektové a proudové. V rámci těchto testů je pokryt i překlad objektového modelu příslušného datového formátu do vnitřní struktury a naopak. Podobnými testy až na testy proudového zpracování je pokryt i překlad dokumentů s X-definicí. Testovací dokumenty resp. dokumenty s X-definicemi jsou rozděleny do tří skupin:

- minimální dokumenty,
- jednoduché dokumenty a
- komplexnější dokumenty.

Projekt obsahuje také testy, které testují, zda objektový model pro formát JSON obsahuje správnou informaci o pozici prvku ve zdroji dat.

8.3 Integrační testy

Integrační testy slouží k testování větších částí aplikace typicky na úrovních jednotlivých komponent. Jak už název napovídá, testy testují vzájemnou integraci jednotlivých komponent. Lze mezi ně zařadit i testy týkající se testů komponenty jako celku.

8.3.1 Pokrytí testy

Integrační testy jsou odděleny do samostatného adresáře pojmenovaného integTest. V rámci integračních testů jsem se zaměřil na kombinace podporovaných formátů tzn. kombinace různých formátů na vstupu a na výstupu. U každé kombinace jsem testoval překlad jak do objektové reprezentace, tak i proudového zpracování. Obdobné testy pokrývají i překlad dokumentů s X-definicemi. Testovací dokumenty jsou, stejně jako u jednotkových testů, rozděleny do tří skupin:

- minimální dokumenty,
- jednoduché dokumenty a
- komplexnější dokumenty.

Závěr

Hlavním cílem diplomové práce bylo vytvoření návrhu nové architektury frameworku X-definice. Pro splnění tohoto cíle bylo potřeba udělat několik dílčích kroků. V první řadě bylo nutné seznámit se s celou technologií X-definice. Dále pak již bylo možné provést analýzu současného řešení potažmo přímo implementace frameworku X-definice. Na základě těchto poznatků, znalostí základních principů návrhu softwaru, struktury podporovaných datových formátů a přístupů ke zpracování dat, bylo možné již přistoupit k samotnému návrhu. Dalším cílem bylo provést ověření tohoto návrhu. V rámci splnění tohoto cíle byla vytvořena základní struktura projektu nového frameworku a byla implementována nová komponenta frameworku zajišťující překlad datových formátů do jednotné interní struktury. Výsledná komponenta byla pokryta sadou testů pro ověření správnosti překladu formátů.

Budoucí rozvoj práce je zřejmý už z jejího tématu. Práce se zabývala převážně jen samotným návrhem architektury, tudíž jako další rozvoj se nabízí dokončení přechodu ze současného řešení na navrženou architekturu. Během řešení práce se otevřelo i mnoho dalších otázek, které nebyly v práci zodpovězeny a které nabízejí možný rozvoj. Jednou z otázek je definice samotné struktury X-definice ve formátu JSON. Další z otázek a tedy i z témat budoucího rozvoje je definování transformace formátu XML na formát JSON a naopak. Oba formáty mají mnoho společných prvků, ale každý má svá specifika a bylo by vhodné najít mapování mezi formáty, které by tento problém vyřešilo. Jako poslední možnost rozvoje, kterou zde uvedu, může být přidání podpory pro zpracování datového formátu YAML.

Literatura

[1] BEAL, Vangie: Structured data. In: *webopedia.com*, [online], webopedia, [cit. 2019-04-14]. Dostupné z: `https://www.webopedia.com/TERM/S/structured_data.html`

[2] BRADLEY, Neil: *XML: kompletní průvodce*. Praha: Grada, první vydání, 2000, ISBN 80-7169-949-7, 43 s.

[3] BRAY, Tim: *The JavaScript Object Notation (JSON) Data Interchange Format*. [online], 2014, [cit. 2019-03-31]. ISSN 2070-1721. Dostupné z: `https://www.ietf.org/rfc/rfc7159.txt`

[4] BRAY, Tim et al.: *Extensible Markup Language (XML) 1.1 (Second Edition)*. [online], W3C, 2006, ©W3C, [cit. 2019-04-13]. Dostupné z: `https://www.w3.org/TR/2006/REC-xml11-20060816/#sec-prolog-dtd`

[5] CITU, Adrian: Book Review: Clean Architecture. In: *adriancitu.com*, [online], Adrian Citu, 2018, [cit. 2019-04-27]. Dostupné z: `https://adriancitu.com/tag/the-common-reuse-principle/`

[6] CROCKFORD, Douglas: *Introducing JSON*. [online], Douglas Crockford, [cit. 2019-03-31]. Dostupné z: `https://www.json.org/`

[7] EELES, Peter a Peter CRIPPS: *Architektura softwaru*. Brno: Computer Press, vyd. 1 vydání, 2011, ISBN 978-80-251-3036-0, 24 s.

[8] ÉCOLE POLYTECHNIQUE FÉDÉRALE LAUSANNE: *The Scala programming language*. [online], École Polytechnique Fédérale Lausanne, 2019, ©École Polytechnique Fédérale Lausanne (EPFL) Lausanne, Switzerland, [cit. 2019-04-11]. Dostupné z: `https://www.scala-lang.org/`

[9] FASTERXML: *Jackson*. [online], GitHub, Inc., 2019, ©GitHub, Inc., [cit. 2019-04-20]. Dostupné z: `https://github.com/FasterXML/jackson`

[10] GOOGLE: *Gson*. [online], GitHub, Inc., 2019, ©GitHub, Inc., [cit. 2019-04-20]. Dostupné z: https://github.com/google/gson

[11] GUPTA, Lokesh: Java Read XML – Java DOM Parser Example. In: *howtodoinjava.com*, [online], [cit. 2019-04-12]. Dostupné z: https://howtodoinjava.com/xml/read-xml-dom-parser-example/

[12] JETBRAINS s.r.o.: *Sealed Classes*. [online], JetBrains s.r.o., 2019, ©JetBrains s.r.o., [cit. 2019-04-26]. Dostupné z: https://kotlinlang.org/docs/reference/sealed-classes.html

[13] KAMENICKÝ, Jiří: X-definice – zpracování a transformace dat [přednáška]. In: *YouTube* [online]. Praha: FIT ČVUT v Praze, 2018-11-19, [cit. 2019-03-12]. Dostupné z: https://www.youtube.com/watch?v=Miyvdqqgo54

[14] KOTLINTEST: *KotlinTest*. [online], GitHub, Inc., 2019, ©GitHub, Inc., [cit. 2019-04-27]. Dostupné z: https://github.com/kotlintest/kotlintest/blob/master/doc/reference.md

[15] MANCA, Florimond: *RESTful API Design: 13 Best Practices to Make Your Users Happy*. In: *blog.florimondmanca.com*, [online], Florimond Manca, 2018, [cit. 2019-04-14]. Dostupné z: https://blog.florimondmanca.com/restful-api-design-13-best-practices-to-make-your-users-happy

[16] MARTIN, Robert C.: *Clean architecture: a craftsman's guide to software structure and design*. Boston: Prentice Hall, 2018, ISBN 978-0-13-449416-6, 57–91 s.

[17] MARTIN, Robert C.: *Clean architecture: a craftsman's guide to software structure and design*. Boston: Prentice Hall, 2018, ISBN 978-0-13-449416-6, 96 s.

[18] MARTIN, Robert C.: *Clean architecture: a craftsman's guide to software structure and design*. Boston: Prentice Hall, 2018, ISBN 978-0-13-449416-6, 103–110 s.

[19] MARTIN, Robert C.: *Clean architecture: a craftsman's guide to software structure and design*. Boston: Prentice Hall, 2018, ISBN 978-0-13-449416-6, 111–132 s.

[20] OLIVEIRA, Bruno, Vasco SANTOS a Orlando BELO: Processing XML with Java – A Performance Benchmark. In: *www.researchgate.net*, [online], ResearchGate, 2019, ©ResearchGate, [cit. 06.04.2019]. Dostupné z: https://www.researchgate.net/publication/260247328_Processing_XML_with_Java_-_A_Performance_Benchmark

[21] ORACLE: *Java API for JSON Processing*. [online], Oracle, 2017, ©Oracle, [cit. 2019-04-20]. Dostupné z: https://javaee.github.io/jsonp/

[22] ORACLE: *StAX API*. [online], Oracle, 2017, ©Oracle, [cit. 2019-04-12]. Dostupné z: https://docs.oracle.com/javase/tutorial/jaxp/stax/api.html

[23] ORACLE: *Why StAX?* [online], Oracle, 2017, ©Oracle, [cit. 2019-04-12]. Dostupné z: https://docs.oracle.com/javase/tutorial/jaxp/stax/why.html

[24] ORACLE: XML Parsing for Java. In: *docs.oracle.com*, [online], 2019, ©Oracle, [cit. 2019-04-12]. Dostupné z: https://docs.oracle.com/cd/B28359_01/appdev.111/b28394/adx_j_parser.htm#ADXDK3000

[25] QOS.ch: *Logback Project*. [online], QOS.ch, 2018, ©QOS.ch, [cit. 2019-04-20]. Dostupné z: https://logback.qos.ch/

[26] QOS.ch: *Simple Logging Facade for Java (SLF4J)*. [online], QOS.ch, 2019, ©QOS.ch, [cit. 2019-04-20]. Dostupné z: https://www.slf4j.org/

[27] SKYSCREAMER: *JSONassert*. [online], GitHub, Inc., 2019, ©GitHub, Inc., [cit. 2019-04-27]. Dostupné z: https://github.com/skyscreamer/JSONassert

[28] SVOBODA, Martin: *Data Formats*. [online prezentace]. 2018-10-30, 6–9 s. [cit. 2019-03-20]. Dostupné z: https://www.ksi.mff.cuni.cz/~svoboda/courses/181-MIE-PDB/lectures/MIEPDB16-Lecture-05-Formats.pdf

[29] SVOBODA, Martin: *Data Formats*. [online prezentace]. 2018-10-30, 14–16 s. [cit. 2019-03-20]. Dostupné z: https://www.ksi.mff.cuni.cz/~svoboda/courses/181-MIE-PDB/lectures/MIEPDB16-Lecture-05-Formats.pdf

[30] SYNTEA SOFTWARE GROUP a.s. [online]: *X-definice 3.1*. 2018, [cit. 2019-03-12]. Dostupné z: http://xdef.syntea.cz/tutorial/cs/userdoc/xdef3.1_ces.pdf

[31] ŠPAČEK, Petr: *Architecture and Design Patterns*. [online prezentace]. 14–27 s. [cit. 2019-04-20]. Dostupné z: https://moodle.fit.cvut.cz/pluginfile.php/197/course/section/10597/miadp-2018-lecture02-principles.pdf

[32] ŠPAČEK, Petr: *Architecture and Design Patterns*. [online prezentace]. 28–57 s. [cit. 2019-04-20]. Dostupné z: https://moodle.fit.cvut.cz/pluginfile.php/197/course/section/10597/miadp-2018-lecture02-principles.pdf

[33] TENCENT: simpledom. In: *rapidjson.org*, [online], Tencent, 2015, ©THL A29 Limited, a Tencent company, and Milo Yip, [cit. 2019-04-21]. Dostupné z: http://rapidjson.org/simpledom.png

[34] TENCENT: *RapidJSON: Tutorial*. [online], Tencent, 2015, ©THL A29 Limited, a Tencent company, and Milo Yip, [cit. 2019-03-31]. Dostupné z: http://rapidjson.org/md_doc_tutorial.html

[35] THE APACHE GROOVY PROJECT: *The Apache Groovy programming language*. [online], The Apache Software Foundation, 2019, ©The Apache Groovy project, [cit. 2019-04-11]. Dostupné z: http://groovy-lang.org/

[36] THE APACHE LOGGING PROJECT: *Apache Log4j 2*. [online], The Apache Software Foundation, 2019, ©Apache Logging project, [cit. 2019-04-20]. Dostupné z: https://logging.apache.org/log4j/2.x/

[37] TUTORIALS POINT INDIA PRIVATE LIMITED: *Java DOM Parser – Overview*. [online], Tutorials Point India Private Limited, 2019, [cit. 2019-04-12]. Dostupné z: https://www.tutorialspoint.com/java_xml/java_dom_parser.htm

[38] VOGELS, Rebecca: Which Programming Languages Should You Learn in 2018? In: *usersnap.com*, [online], Usersnap, 2019, [cit. 2019-05-02]. Dostupné z: https://usersnap.com/blog/programming-languages-2018/

[39] W3C DOM IG: *W3C Document Object Model*. [online], 2005, ©W3C, [cit. 2019-04-12]. Dostupné z: https://www.w3.org/DOM/

[40] XMLUNIT: *Unit Testing XML for Java and .NET*. [online], GitHub, Inc., 2019, ©GitHub, Inc., [cit. 2019-04-27]. Dostupné z: https://www.xmlunit.org/

Příloha **A**

Seznam použitých zkratek

API Application Programming Interface

CRP Common Reuse Principle

CSS Cascading Style Sheets

DOM Document Object Model

DOM4J Document Object Model for Java

DRY Don't Repeat Yourself

HTML Hypertext Markup Language

ISP Interface Segregation Principle

Java EE Java Enterprise Edition

Java SE Java Standard Edition

JDK Java Development Kit

JDOM Java Document Object Model

JS JavaScript

JSON JavaScript Object Notation

JSON-P JSON Processing

JVM Java Virtual Machine

KISS Keep It Simple Stupid

LSP Liskov Substitution Principle

OCP Open-Closed Principle

OOP Objektově orientované paradigma

REP Reuse/Release Equivalence Principle

RFC Request For Comments

SAP Stable Abstractions Principle

SAX Simple API for XML

SDP Stable Dependencies Principle

SLF4J Simple Logging Facade for Java

SOAP Simple Object Access Protocol

SoC Separation of Concerns

SRP Single Responsibility Principle

StAX Streaming API for XML

URL Uniform Resource Locator

W3C World Wide Web Consortium

XML Extensible Markup Language

XML-RPC XML Remote Procedure Call

XSLT Extensible Stylesheet Language Transformations

YAGNI You Are not Going to Need It

YAML YAML Ain't Markup Language

ZOIR Zero One Infinity Rule

Příloha B

Slovník

dobře strukturovaný (well-formed) dokument je dokument, který splňuje všechny formální požadavky, které definuje specifikace XML

procesor X-definic je komponenta frameworku X-definice, která na základě instrukcí vzniklých kompilací X-definice, provádí jimi definovaný kód

r-skript je část skriptu X-definice, jejíž zpracování provádí komponenta Runtime

reentrantní objekt je objekt, který neobsahuje proměnná data

x-skript je část skriptu X-definice, jejíž zpracování provádí komponenta XScript

Příloha C

Přehled tabulek

Tabulka C.1: Specifikace výskytu elementů [30]

Klíčové slovo	**Popis**	**Alternativní zápis**
required	element se musí vyskytnout maximálně jednou	1, 1..1
optional	element se může vyskytnout maximálně jednou	?, 0..1
ignore	element je při zpracování ignorován	
illegal	element se nesmí vyskytnout	
*	element se nemusí vyskytnout nebo počet výskytů není omezen	0..*
+	element se musí vyskytnout minimálně jednou	1..*
m	element se musí vyskytnout právě m-krát	
m..n	element se musí vyskytnout minimálně m-krát a maximálně n-krát	
n..*	element se musí vyskytnout minimálně n-krát	

Tabulka C.2: Specifikace výskytu textových hodnot a atributů [30]

Klíčové slovo	**Popis**	**Alternativní zápis**
required	atribut nebo textová hodnota je povinná	1, 1..1
optional	atribut nebo textová hodnota je nepovinná	?, 0..1
ignore	atribut nebo textová hodnota je při zpracování ignorována	
illegal	atribut nebo textová hodnota je zakázaná	

Tabulka C.3: Specifikace událostí elementů [30]

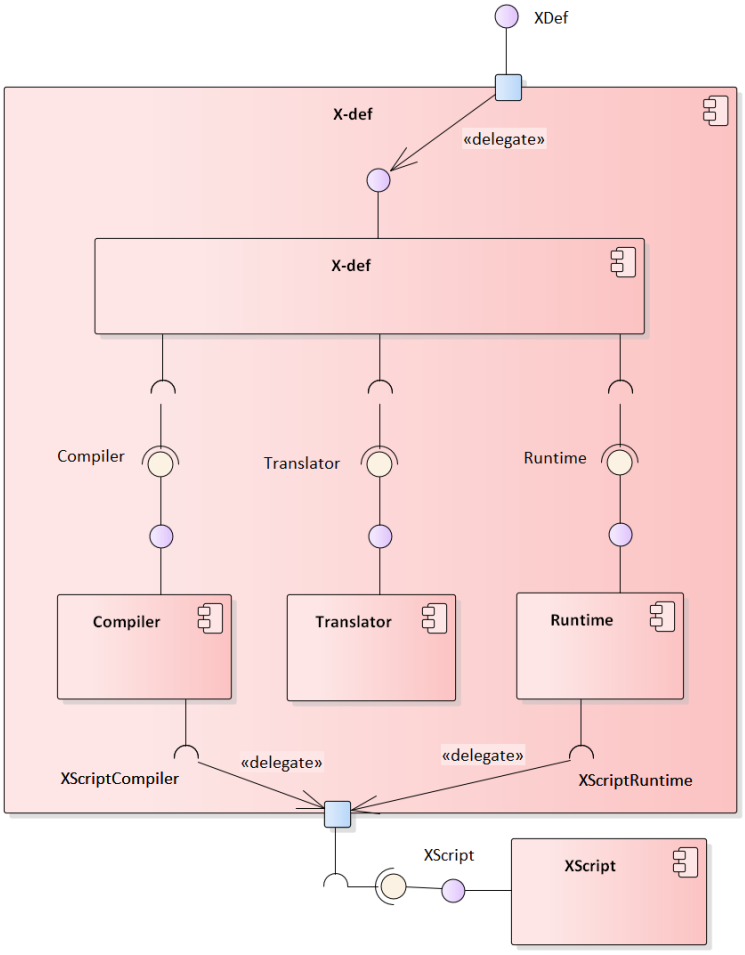
Klíčové slovo	**Popis**
create	Akce nastává jen v režimu konstrukce.
match	Akce nastává před dalším zpracováním elementu. Akce vrací hodnotu `true`, nebo `false`. Pokud je vrácena hodnota `true`, pokračuje zpracování normálně, jinak není aktuální element zpracován podle modelu elementu, v němž je akce zapsána.
finally	Akce nastane po zpracování elementu.
forget	Akce nastane po akci finally. Akce způsobí, že se příslušný element odstraní z paměti.
init	Akce slouží pro inicializaci zpracování. Provede se před zpracováním objektu.
onAbsence	Akce nastane, pokud není splněna podmínka minimálního výskytu elementu.
onExcess	Akce nastane, pokud počet výskytů elementu překračuje horní povolenou hranici.
onIllegalAttr	Akce se provede v případě, že se v elementu vyskytuje nedefinovaný, nebo nepovolený atribut.
onIllegalElement	Akce se provede v případě, že se v elementu vyskytuje nedefinovaný, nebo nepovolený element.
onIllegalText	Akce se provede v případě, že se v elementu vyskytuje nedefinovaná, nebo nepovolená textová hodnota.
onIllegalRoot	Akce je povolena pouze ve skriptu X-definice nikoliv v modelech. Akce se provede, když není v datech nalezen odpovídající kořenový element.
onStartElement	Akce nastane po zpracování seznamu atributů, ale ještě před zpracováním obsahu elementu.

Tabulka C.4: Specifikace událostí textových hodnot a atributů [30]

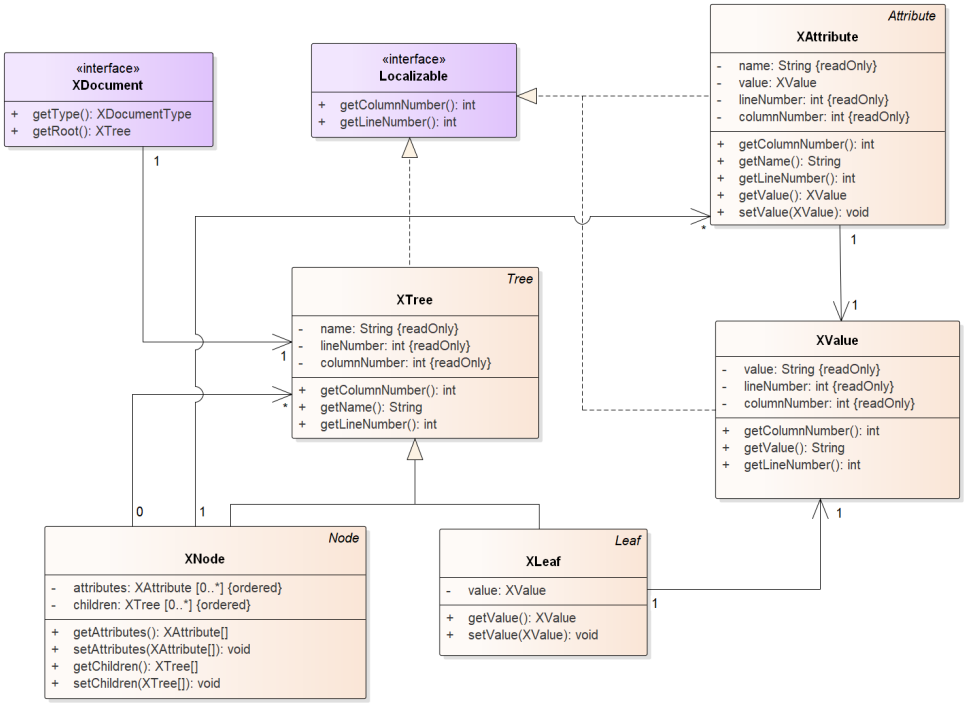
Klíčové slovo	**Popis**
create	Akce nastává jen v režimu konstrukce.
default	Akce se provede v případě, pokud atribut, nebo textová hodnota neexistuje.
match	Akce nastává před dalším zpracováním atributu. Akce vrací hodnotu `true`, nebo `false`. Pokud je vrácena hodnota `true`, pokračuje zpracování normálně, jinak není aktuální element zpracován podle modelu elementu, v němž je akce zapsána.
init	Akce slouží pro inicializaci zpracování. Provede se před zpracováním objektu.
onAbsence	Akce nastane, pokud není splněna podmínka minimálního výskytu elementu.
onExcess	Akce nastane, pokud počet výskytů elementu překračuje horní povolenou hranici.
onTrue	Akce nastane v případě, je-li výsledek vyhodnocení kontroly typu `true`.
onFalse	Akce nastane v případě, je-li výsledek vyhodnocení kontroly typu `false`.

Příloha D

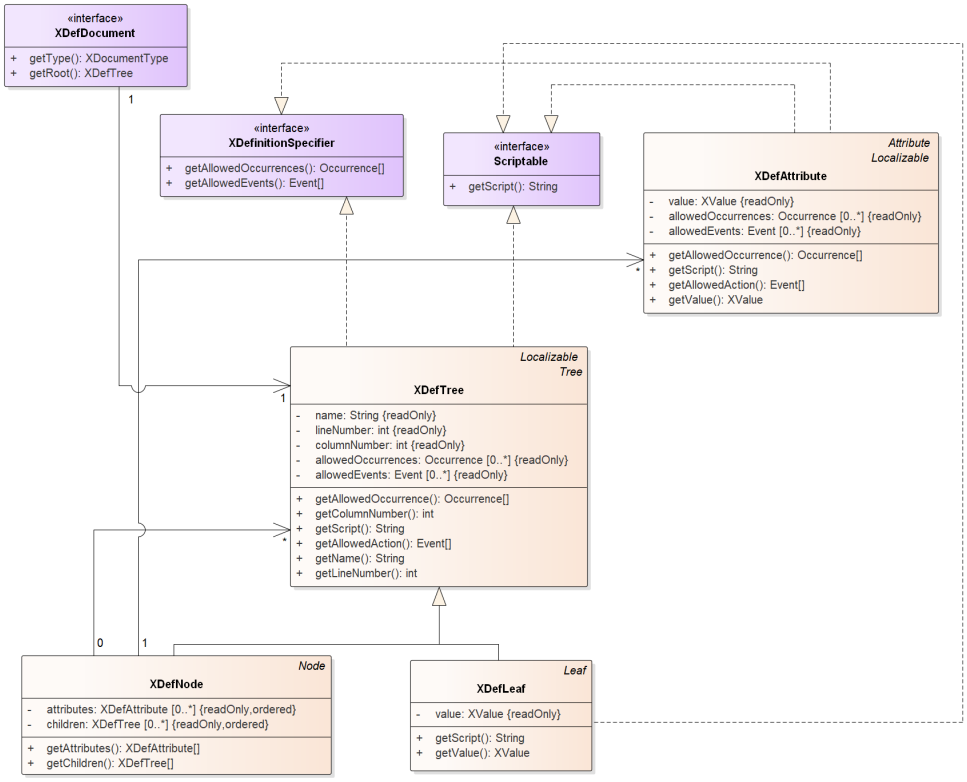
Obrázky a schémata



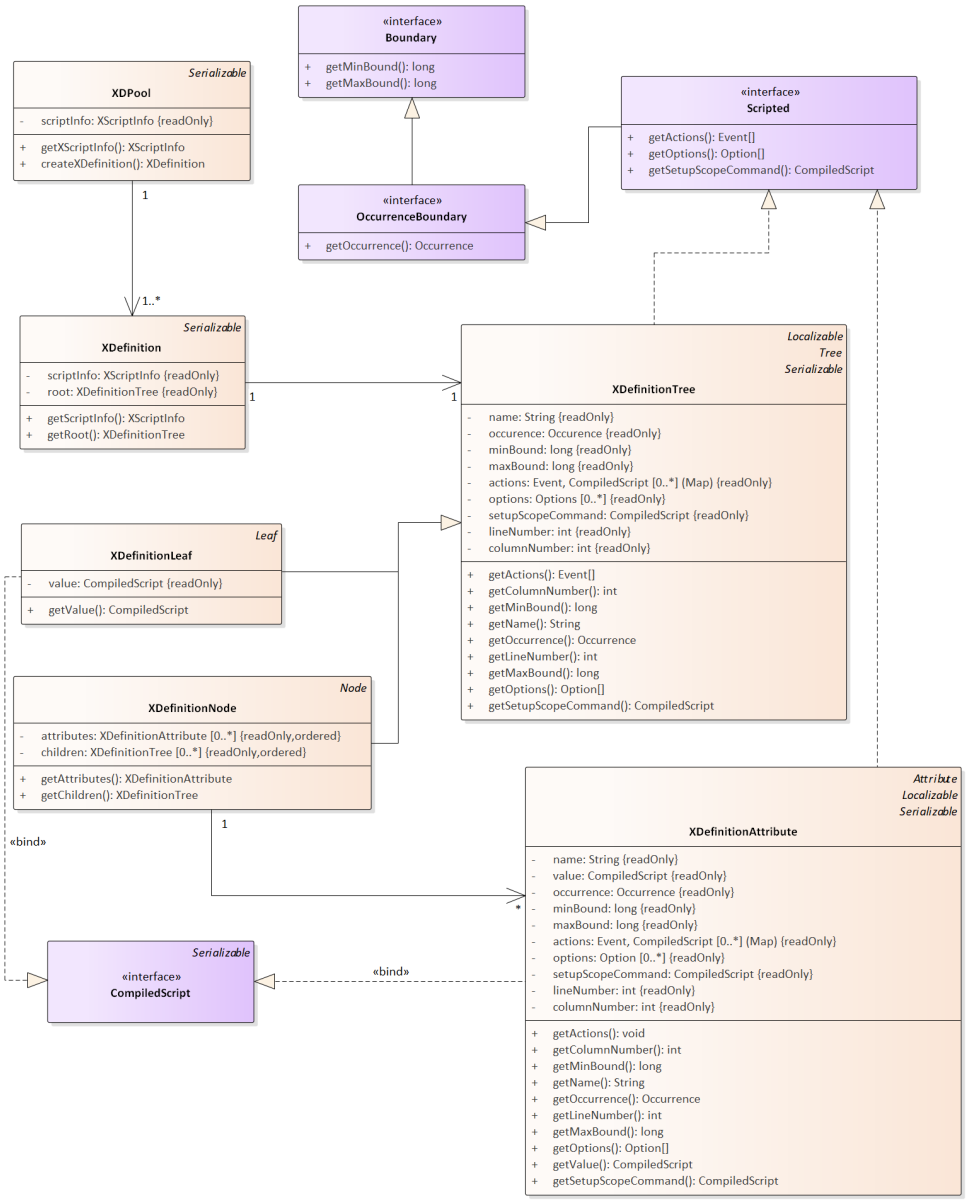
Obrázek D.1: Diagram komponent



Obrázek D.2: Digram tříd pro dokument



Obrázek D.3: Digram tříd pro dokument s X-definicí



Obrázek D.4: Digram tříd pro zkompilovanou X-definici

Příloha E

Ukázky zdrojových kódů

```
//Vytvoření factory pro vytváření dokumentů
SAXBuilder builder = new SAXBuilder();
//Vytvoř dokument
Document document = builder.build(new File("employees.xml"));

//Získej kořen dokumentu a vypiš jeho jméno
Element root = document.getRootElement();
System.out.println(root.getName());

//Získej všechny elementy 'employee'
for (Element node : root.getChildren("employee")) {
  //Pro každého 'employee' vypiš jeho detail
  System.out.println("Employee id : " +
  ↪  node.getAttribute("id"));
  System.out.println("First Name : " +
  ↪  node.getChildText("firstName"));
  System.out.println("Last Name : " +
  ↪  node.getChildText("lastName"));
  System.out.println("Location : " +
  ↪  node.getChildText("location"));
}
```

Zdrojový kód E.1: Ukázka práce s JDOM knihovnou

```
//Vytvoření factory pro vytváření dokumentů
DocumentBuilderFactory factory =
↪  DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
//Vytvoř dokument
Document document = builder.parse(new File("employees.xml"));

//Znormalizuj XML strukturu
document.getDocumentElement().normalize();

//Získej kořen dokumentu a vypiš jeho jméno
Element root = document.getDocumentElement();
System.out.println(root.getNodeName());

//Získej všechny elementy 'employee'
NodeList nodes = document.getElementsByTagName("employee");

for (int i = 0; i < nodes.getLength(); i++) {
  Node node = nodes.item(i);
  if (Node.ELEMENT_NODE == node.getNodeType()) {
    //Pro každého 'employee' vypiš jeho detail
    Element eElement = (Element) node;
    System.out.println("Employee id : " +
    ↪  eElement.getAttribute("id"));
    System.out.println("First Name : " +
    ↪  eElement.getElementsByTagName("firstName").item(0)
    ↪  .getTextContent());
    System.out.println("Last Name : " +
    ↪  eElement.getElementsByTagName("lastName").item(0)
    ↪  .getTextContent());
    System.out.println("Location : " +
    ↪  eElement.getElementsByTagName("location").item(0)
    ↪  .getTextContent());
  }
}
```

Zdrojový kód E.2: Ukázka práce s DOM parserem [11]

```
// vytvoření továrny pro vytváření pruků
// pro práci s JSON dokumenty
val factory: JsonFactory = JsonFactory()

// vytvoření parseru
val reader: JsonParser = factory.createParser(/*vstup*/)
// vytvoření generátoru výstupu
val writer: JsonGenerator = factory.createGenerator(/*výstup*/)

// průchod vstupního dokumentu
while (reader.nextToken() != null) {
  // zjištění pozice tokenu
  reader.currentLocation.lineNr // číslo řádky
  reader.currentLocation.columnNr // číslo sloupce

  when (reader.currentToken()) {
    JsonToken.START_OBJECT -> writer.writeStartObject()
    JsonToken.END_OBJECT -> writer.writeEndObject()
    JsonToken.START_ARRAY -> writer.writeStartArray()
    JsonToken.END_ARRAY -> writer.writeEndArray()
    JsonToken.FIELD_NAME ->
        writer.writeFieldName(reader.currentName)
    JsonToken.VALUE_EMBEDDED_OBJECT ->
        writer.writeEmbeddedObject(reader.embeddedObject)
    JsonToken.VALUE_STRING ->
        writer.writeString(reader.valueAsString)
    JsonToken.VALUE_NUMBER_INT ->
        writer.writeNumber(reader.intValue)
    JsonToken.VALUE_NUMBER_FLOAT ->
        writer.writeNumber(reader.floatValue)
    JsonToken.VALUE_TRUE,
    JsonToken.VALUE_FALSE ->
        writer.writeBoolean(reader.valueAsBoolean)
    JsonToken.VALUE_NULL -> writer.writeNull()
    JsonToken.NOT_AVAILABLE -> AssertionError()
    else -> AssertionError()
  }
}
```

Zdrojový kód E.3: Ukázka Jackson rozhraní pro zpracování dokumentu

```
class Builder {

  /**
   * @param type   Typ dokumentu s X-definicí např. XML
   * @param source X-definice
   * @param others Další X-definice stejného typu
   */
  fun addRuleSource(type: SupportedDataType, source: File,
                    vararg others: File): Builder

  /**
   * @param type          Typ dokumentu s X-definicí např. XML
   * @param sourceCharset Kódování dokumentu s X-definicí
   * @param source        X-definice
   * @param others        Další X-definice stejného typu
   */
  fun addRuleSource(type: SupportedDataType, sourceCharset:
  ↪  Charset, source: File, vararg others: File): Builder

  /**
   * @param factory Tovární třída pro vytváření XScriptu
   */
  fun setScriptFactory(factory: XScriptFactory): Builder

  /**
   * Při vytvoření XDef provede i kompilaci všech X-definic
   * @return Instance frameworku
   */
  @Throws(IllegalStateException::class, IOException::class)
  fun buildWithCompile(): XDef

  /**
   * Vytvoří XDef, kompilace se provádí až v okamžiku
   * prvního použití
   * @return Instance frameworku
   */
  @Throws(IllegalStateException::class)
  fun build(): XDef
}
```

Zdrojový kód E.4: Fragment rozhraní třídy XDef.Builder

Příloha F

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `doc`
  - `design` ..... adresář s diagramy
  - `kdoc` ..... adresář s dokumentací kódu ve formátu KDoc
- `src`
  - `xdef` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `DP_Smid_Filip_2019.pdf` ..... text práce ve formátu PDF