Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Generátor XML souborů řízený XSD schématem

Plzeň 2014 Milan Balon

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 14. května 2014

Milan Balon

Poděkování

Rád bych poděkoval vedoucímu práce doc. Ing. Pavlu Heroutovi, Ph.D. za jeho cenné rady, věcné připomínky a vstřícnost při konzultacích k vypracování diplomové práce.

Abstract

XSD schema driven XML generator

The goal of this thesis is to develop an application that can under certain constraints create a graphical user interface from XSD schema and then produce valid XML documents according to the schema. We emphasize especially on robustness and reusability of created solution. Solved problem has been designed and implemented under the .NET platform using XML technologies and technology WPF. Created solution provides a modular kernel that is capable of converting an XSD to XAML, its displaying, saving entered values into XML or loading previously created XML into GUI. Part of the solution is an implementation of a configuration tool intended for contracting authority. The result of this thesis allows the creation of applications driven by schema for maximally simplified creation of arbitrarily complex XML documents.

Generátor XML souborů řízený XSD schématem

Cílem této práce je vytvoření aplikace, která za určitých omezujících podmínek dokáže z XSD schématu vytvořit grafické uživatelské rozhraní, produkující XML dokument validní podle tohoto schématu. Důraz je kladen zejména na robustnost a vícenásobnou použitelnost vytvořeného řešení. Řešený problém byl navržen a implementován pod platformou .NET pomocí XML technologií a technologie WPF. Vytvořené řešení poskytuje modulární jádro schopné převedení XSD na XAML, jeho zobrazení, uložení zadaných dat do XML nebo načtení dříve vytvořeného XML do GUI. Součástí řešení je implementace konfiguračního nástroje určená pro zadavatele práce. Výsledek této práce umožňuje vytváření aplikací řízených schématem pro maximální zjednodušení tvorby libovolně složitých XML dokumentů.

Obsah

1 Úvod

Významný průmyslový partner Katedry informatiky a výpočetní techniky Západočeské univerzity v Plzni, který je mimo jiné výrobcem zabezpečovací techniky pro oblast kolejové dopravy, používá pro konfiguraci této techniky konfigurační soubory ve formátu XML.

Pro přípravu souborů využívá aplikaci s pevně daným uživatelským rozhraním. Problémem současného řešení je, že se struktura jednotlivých konfiguračních souborů časem mírně mění, takže je nutné výstup aplikace ručně upravovat. Aplikace navíc neumožňuje tvorbu konfiguračních souborů pro složitější zařízení, které jsou tvořeny syntézou dílčích konfigurací.

Cílem této práce je návrh a implementace jádra aplikace, která umožní na základě XSD schématu popisujícího cílový dokument, vytvořit grafické uživatelské rozhraní, jež bude produkovat validní XML dokumenty. To znamená, že pro odlišná XSD bude aplikace vytvářet odlišná uživatelská rozhraní. Jádro navíc musí být schopné do vytvořeného uživatelského rozhraní načíst dříve vytvořený XML dokument, jeho editaci a uložení. Samozřejmostí je automatická validace vytvořeného/upraveného XML podle schématu.

Řešením práce by měl být robustní, znovupoužitelný modul jádra, jehož funkčnost otestujeme ve dvou implementacích. První by měla být jednoduchá aplikace, demonstrující správnou funkčnost jádra. Druhou poté reálná aplikace pro projektování konfigurací řídícího jádra železničních přejezdů.

2 XML technologie

2.1 XML

Rozšiřitelný značkovací jazyk XML (eXtensible Markup Language) (W3C, 2008b), slouží k uchování, zpracování a přenosu strukturovaných dokumentů ve standardizovaném textovém formátu. XML je jedním z mnoha standardů konsorcia W3C (World Wide Web Consortium) (W3C, 2014), které zajišťuje standardizaci a vývoj technologií využívaných převážně v oblasti internetových aplikacích. Standard je velice hojně využíván díky jeho velmi dobrým vlastnostem, zejména přenositelnosti. (Herout, 2007)

Prvním značkovacím jazykem, určeným pro označování významu dat, byl značkovací jazyk SGML (Standard Generalized Markup Language). Jazyk původně využívala vládní infrastruktura Spojených států amerických pro výměnu dokumentů, tento jazyk byl však příliš obecný a složitý, proto se příliš neujal. Důsledkem toho byl odstraněním některých nevyužívaných funkcionalit SGML vytvořen nový jazyk XML. XML je podmnožina SGML s téměř všemi výhodami tohoto jazyka. (Kosek, 2000)

2.1.1 Základní syntaxe

Základní stavební kameny

Základními stavebními kameny jazyka jsou element, atribut a text. Element je reprezentován otevírací značkou, obsahem a uzavírací značkou. Z elementů je potom sestaven celý dokument v požadované struktuře, které dosáhneme pomocí zanořování elementů do sebe. Element má vždy otevírací a uzavírací značku.

```
<element>obsah</element>
```

Obsah elementu se uvádí mezi otevírací a uzavírací značku elementu a obvykle reprezentuje data související s elementem a jeho atributy. Textem může být libovolná textová informace, libovolné množství vnořených elementů, nebo kombinace elementů a textu.

Atributy obsahují hodnoty, které jsou nějakým způsobem svázané s elementem, a jsou vždy součástí otevírací značky elementu.

```
<element atribut="hodnota">obsah</element>
```

Atributů je možné k elementu přiřadit neomezené množství. Název atributu je následován znakem = a hodnota atributu je uzavřena v jednoduchých (') nebo dvojitých (") uvozovkách. Opačný druh uvozovek, než který byl použit k uvození atributu, může být použit v textu hodnoty atributu.

Zvláštním případem elementu je prázdný element, který nemá obsah. Takovýto element je možné ukončit přímo v otevírací značce pomocí znaku / na konci značky. Dobrým zvykem je dávat před znak lomítka mezeru. Prázdný element může mít v otevírací značce libovolné množství atributů. (Benz – Durant, 2003)

```
<element />
```

Struktura dokumentu

Základním prvkem každého XML dokumentu je XML deklarace (prolog). Jedná se o jasnou identifikaci, že předložený textový dokument je XML dokumentem. Říká nám, jaká verze jazyka je použita a řeší problém s kódováním textu na různých platformách. Deklaraci je nutné uvést hned na začátku dokumentu. Tyto informace využívají zejména XML parsery (viz kap. 2.2) ke korektnímu zpracování dané verze a daného kódování. (Benz – Durant, 2003)

```
<?xml version="1.0" encoding="UTF-8" ?>
```

Pod XML deklarací se musí nacházet právě jeden kořenový element dokumentu. V obsahu tohoto kořenového elementu je poté text, ve většině případů však další zanořené elementy, je tak možné vytvořit složitou stromovou strukturu. (Kosek, 2000)

```
<?xml version="1.0" encoding="UTF-8" ?>
<člověk pohlaví="muž">
  <jméno>Milan</jméno>
  <příjmení>Balon</příjmení>
</člověk>
```

Zpracovávací instrukce

Zpracovávací instrukce se používají pro speciální komunikaci XML s aplikací, která XML zpracovává. Takovou aplikací může být například webový prohlížeč, kterému v dokumentu předáme informaci o tom, jaký má použít styl pro zobrazení dokumentu. Každá zpracovávací instrukce má cíl (target) a pseudoatributy.

```
<?xml-stylesheet type="text/xsl" href="transformace.xslt" ?>
```

V tomto příkladě je cílem `xml-stylesheet` a pseudoatributy `type` a `href`. Každý cíl má svou sadu pseudoatributů, zde nesou informaci o typu stylopisu a místě, kde se nachází. Konktrétně se jedná o XSL transformaci, které je věnována kapitola 2.5. (Fawcett et al., 2012)

2.1.2 Jmenné prostory

Jmenné prostory (namespaces) slouží jako nástroj pro seskupování značek XML. Pokud chceme sloučit dva XML dokumenty do jednoho, je nemalá pravděpodobnost vzniku kolizí jmen elementů. Jména elementů každého dokumentu se proto seskupí do jmenného prostoru. Tomu je přiřazen unikátní identifikátor (URI) a v rámci dokumentu unikátní prefix, kterým se označí elementy, které do daného prostoru patří. (Fawcett et al., 2012)

```
<kořen xmlns="http://balon.cz/implicitni"
       xmlns:xy="http://balon.cz/xy" >
  <xy:jméno>Milan</xy:jméno>
  <jméno>Anna<jméno>
</kořen>
```

Deklarace jmenného prostoru je atribut, který je na rozdíl od ostatních atributů v dokumentu označena prefixem `xmlns:` po němž bezprostředně následuje prefix jmenného prostoru. Hodnotou deklarace je unikátní identifikátor. Je možné deklarovat i implicitní jmenný prostor tím, že neuvedeme jeho prefix. Všechny elementy v dokumentu bez prefixu poté budou patřit do implicitního jmenného prostoru. (Benz – Durant, 2003)

2.1.3 Správně strukturovaný XML dokument

Správně strukturovaný (well-formed) XML dokument je takový dokument, který je ve všech ohledech vytvořen v souladu s W3C XML standardem. Puristé proto tvrdí, že něco jako správně strukturované XML neexistuje, protože dokument buďto je XML, a poté musí být z principu věci správně strukturovaný, nebo je to jen prostý textový dokument. Tento pojem se však všeobecně zažil a je používán ve všech návazných technologiích a aplikacích jako kontrola souladu dokumentu se standardem. (Fawcett et al., 2012)

Pokud dokument není správně strukturován, neměl by být cílovou aplikací zpracován. To umožňuje snadné strojové zpracování dokumentu, jelikož přesně víme, jakou může mít strukturu. Tato skutečnost značně zjednodušuje implementaci nástrojů pro zpracování XML (kap. 2.2). (Fawcett et al., 2012)

2.1.4 Validní XML dokument

Jak je vidět v kapitole 2.1.3, jsou na XML dokument kladena přísná pravidla pro to, aby byl vůbec za XML považován. Nicméně tento základní mechanismus nijak neupravuje to, jakou sadu značek je možné použít, jak je možné strukturovat elementy, jaké atributy můžou jednotlivé elementy obsahovat a jaké jsou kladeny požadavky na data v těchto elementech a atributech uváděná.

Všechny tyto informace je možné přesně definovat pomocí XML schémat, o kterých pojednává kapitola 2.3. XML dokument se poté porovná s pravidly, která jsou ve schématech uvedena. Tomuto procesu se říká validace, a dokument, který tímto procesem úspěšně projde se označuje jako validní XML dokument. (Benz – Durant, 2003)

2.2 XML parsery

Základní činností, kterou chceme s dokumenty ve formátu XML provádět, je jejich čtení. První věc, co bychom s dokumentem mohli udělat, je otevřít ho v programu jako textový soubor a pomocí nějakého vlastního mechanismu z něj data číst (například budeme proudově číst dokument, a když narazíme na speciální znaky XML, provedeme určitou činnost). Tímto způsobem to samozřejmě provést lze, práce je to ale značně kontraproduktivní, jelikož minimálně číst z XML dokumentu chce v programech každý.

Proto již existují speciální nástroje pro zpracování XML, které toto umožňují. Tyto nástroje se nazývají parsery. Termínem zpracování se potom rozumí čtení, změna stávajících prvků, přidávání nových prvků a transformace dokumentu do jiných formátů. (Herout, 2007)

V následujících podkapitolách jsou popsány základní typy parserů, které se v současné době používají. U každého typu je uvedena i konkrétní implementace v programovacím jazyce Java a prostředí Microsoft .NET Framework.

2.2.1 Proudové parsery

Proudový parser využívá proudové čtení dat, což mimo jiné znamená, že čte dokument postupně od začátku do konce. Výhoda proudového čtení spočívá ve veliké rychlosti a malé paměťové náročnosti. Toto řešení s sebou však nese i určité nevýhody a hlavní nevýhodou je pak skutečnost, že se při čtení nelze vracet zpět. Existují dva základní druhy proudových paserů, typ „push“ a „pull“.

Proudový „push“ parser

Proudový parser typu „push“ čte proud dat automaticky od začátku do konce (proud dat sám „tlačí“). Parser přitom v průběhu čtení vyvolává obslužné funkce, které s načteným kusem dokumentu pracují. Narazí-li například na otevírací značku elementu, zavolá se automaticky funkce `startElement()`, jejíž obsluhu sami implementujeme, a tím je nám umožněno na tuto skutečnost nějakým způsobem reagovat. (Herout, 2007)

SAX (Simple API for XML) (SAX, 2004) je základní „push“ parser, který původně vznikl pouze pro programovací jazyk Java, v současné době ale existuje jeho mutace pro velké množství programovacích jazyků (PHP, .NET, Pascal, Python, ...) a stal se tak základním nástrojem pro zpracování XML dokumentu.

Proudový parser SAX je v programovacím jazyce Java k dispozici v JAXP (Java API for XML Processing) (GlassFish, 2014). Prostředí .NET již SAX přímo nepodporuje a místo něj se doporučuje použít „pull“ parser.

Tento druh parseru je výhodné použít v případech, kdy požadujeme rychlé čtení s ne příliš velkým množstvím událostí. Vhodný je tak například pro validaci dokumentu oproti schématu, což například SAX umožňuje automaticky, bez nutnosti reakcí na události.

Proudový „pull“ parser

Proudový parser typu „pull“ nečte, na rozdíl od „push“ parseru, proud dat od začátku do konce automaticky, ale na naši žádost (proud dat sami „taháme“). Můžeme tak zpracovat část dokumentu, potom dělat něco jiného a časem se vrátit a zpracování dokončit. Data tak čteme pouze v tu chvíli, kdy je právě potřebujeme. Díky tomu může být kód aplikace mnohem přehlednější a přímočařejší.

Pull parsery zpravidla poskytují rozhraní jak pro čtení, tak pro zápis. To nám umožňuje provádět změny v právě přečteném dokumentu s jejich okamžitým zápisem. Vytváření a změna dokumentů je paměťově nenáročná, proto je tato technologie vhodná zejména pro zpracování objemných dokumentů, ve kterých provádíme minimální modifikace.

Technologie push parserů je dostupná na různých platformách pod různými názvy. V programovacím jazyce Java je tento parser reprezentován technologií StAX (Streaming API for XML) (Oracle, 2014b), přes rozhraní `XmlStreamReader` a `XmlStreamWriter`. V prostředí .NET nemá technologie speciální název, ale je dostupná jako třídy `XmlReader` a `XmlWriter` pod jmenným prostorem `System.Xml` (MSDN, 2014b). Rozhraní se mezi platformami mírně liší, filosofie jejich použití je ale vždy stejná.

2.2.2 DOM

DOM (Document Object Model) (W3C, 2004a) je standard společnosti W3C pro práci s XML. Parser pracuje se stromovou reprezentací dokumentu. Ten nejprve přečte celý dokument a v paměti počítače vytvoří jeho kopii ve stromové struktuře. Tomuto stromu se říká infoset. S infosetem potom můžeme libovolně pracovat, číst jej, měnit a zapisovat nové informace. Provádíme-li však změny, je po ukončení veškerých prací nutné celý infoset uložit zpět do XML.

Tento způsob zpracování s sebou nese řadu výhod a nevýhod. Výhodou je, že máme celý infoset v paměti a můžeme s ním pohodlně a velice rychle pracovat. Hlavní nevýhodou tohoto zpracování je nízká rychlost načítání do paměti (sestavuje se složitý strom) a velká paměťová náročnost (celý dokument se přenáší do paměti). (Herout, 2007)

DOM je vhodné použít, chceme-li v dokumentu něco měnit nebo vkládat nové elementy. Tento dokument však nesmí být příliš objemný, aby se vešel do vnitřní paměti. Chceme-li dokument pouze číst, není DOM vůbec vhodné řešení.

Díky standardizaci W3C je DOM široce rozšířen a je k dispozici ve velkém množství programovacích jazyků. V programovacím jazyce Java je DOM, stejně jako SAX, k dispozici přes JAXP. Prostředí .NET jej podporuje ve třídě `XmlDocument` pod jmenným prostorem `System.Xml`.

2.2.3 Mapování XML na objekty

Poslední technologií sloužící pro zpracování XML dokumentů je mapování XML dokumentu na objekty programovacího jazyka. Technologie je využitelná v objektově orientovaných programovacích jazycích.

Před použitím technologie je nutné si nejprve připravit objektový model XML dokumentu, do kterého se bude následně mapovat. To je možné provést buďto ručně, nebo automatizovaně s využitím XSD schématu. Technologie je tak použitelná pouze v případě, kdy předem přesně známe podobu zpracovávaného dokumentu.

Do modelu je poté možné XML dokument načíst, nebo základními prostředky programovacího jazyka vytvořit dokument nový. Stejným způsobem

poté můžeme model měnit. Po ukončení práce s dokumentem jej namapujeme zpět do XML.

V programovacím jazyce Java se technologie mapování nazývá JAXB (Java Architecture for XML Binding) (Oracle, 2014a). Pro tvorbu objektového modelu z XSD schématu se používá program `xjc`, který je součástí distribuce JDK (Java Development Kit). Pro mapování XML na objekty se používá pojem „marshalling“, mapování objektů zpět do XML potom „unmarshalling“. (Herout, 2007)

Prostředí .NET opět nemá pro technologii speciální název a je reprezentována jmenným prostorem `System.Xml.Serialization` (MSDN, 2014e). Generování objektového modelu zajišťuje nástroj XML Schema Definition Tool (`xsd.exe`), který je také součástí .NET SDK (Software Development Kit). Jak je vidět z názvu jmenného prostoru, pro mapování se zde používají termíny „serializace“ a „deserializace“.

Výhodou tohoto řešení je kompletní odstínění od problematiky XML, a s dokumenty tak pracujeme čistě objektově v programovacím jazyce. Hlavní nevýhoda je stejná jako u technologie DOM, a to paměťová náročnost, jelikož je objektový model, stejně jako infoset, uložen ve vnitřní paměti.

Technologii mapování se vyplatí použít především při dynamickém vytváření zcela nových dokumentů, což je oproti proudovým parserům nebo DOM značně jednodušší.

2.3 XML schémata

XML schémata nám dávají mechanismus pro definici množiny značek, které je možné v dokumentu použít, jak je možné strukturovat elementy, jaké atributy můžou jednotlivé elementy obsahovat a jaké jsou kladeny požadavky na data v těchto elementech a atributech uváděná. Jedná se tak o jednoznačnou a formální definici nově vytvořeného značkovacího jazyka. (Kosek, 2013)

Jako příklad pro osvětlení problematiky XML schémat použijeme modelovou situaci, kdy chceme předávat určitá data našemu obchodnímu partnerovi. Dohodneme se, že data budou strukturovaně uložena v XML, ale protože neexistuje schéma, může si tam každý z nás psát vlastní elementy, přestože dokument bude všechna klíčová data obsahovat. To by potom znamenalo, že bychom při získání dokumentu museli nejprve zkoumat, jaká je jeho struktura a poté změnit software, který dokument načítá, případně do něj zapisuje.

Z tohoto příkladu je asi patrné, že tento systém v reálném světě není možné bez větších problémů provozovat. Zde přichází do hry právě XML schémata. Dohodnou-li se obchodní partneři na nějaké struktuře, musí ji oba dodržovat a tím je zajištěna kompatibilita těchto dokumentů se zpracujícím softwarem, nebo jen zajistí přehlednost při čtení v textovém editoru (již není nutné zkoumat, co který element nese za informaci).

V současné době existují tři základní druhy schémat, schémata založená na gramatice, na pravidlech a na jmenných prostorech. Každý druh slouží k popisu jiného druhu omezení, a pokud chceme dokument XML popsat dokonale, je doporučeno k dokumentu definovat všechna tři.

Typy schémat, včetně konkrétních implementací, jsou definovány ve standardu organizace ISO (International Organization for Standardization) pod označením DSDL (Document Schema Definition Languages) (DSDL, 2010). Standard definuje pro každý typ schématu právě jednu implementaci.

Demonstraci jednotlivých schémových jazyků provádíme na následujícím XML dokumentu.

```
<?xml version="1.0" encoding="UTF-8"?>
<produkt xmlns="http://www.balon.org/2014"
    cena="25" dph="5.25" sazbaDph="21">
    <název>Čokoláda</název>
    <datumVýroby>2014-01-10</datumVýroby>
    <datumExpirace>2014-02-15</datumExpirace>
</produkt>
```

2.3.1 Schémata založená na gramatice

Schémata založená na gramatice slouží k definici slovníku elementů (množiny značek) a atributů. Z tohoto slovníku je vytvořen strukturní model cílového dokumentu. Pokročilé jazyky umožňují i definici datových typů obsahu a jejich omezení. (Fawcett et al., 2012)

DTD

DTD (Document Type Definition) (W3C, 2008a) je vůbec první jazyk, sloužící primárně pro popis dokumentů orientovaných na sdělení, který se používá dodnes. Pro datově orientované dokumenty však vhodný není. Specifikace jazyka je přímo součástí specifikace XML a jazyk pochází ještě z dob SGML. (Kosek, 2013)

„Současný názor na DTD je dosti „odsuzující“, protože je považován za: „největší chybu ve vývoji XML“.“ (Herout, 2007, s. 52)

Výhodou jazyka je jeho všeobecná podpora v aplikacích. Nevýhod je potom celá řada. Špatná podpora jmenných prostorů XML, špatná podpora datových typů a podstatné limity v popisu modelu obsahu. (Fawcett et al., 2012) Právě nemožnost definovat typy a rozsahy dat, které elementy obsahují, je primární důvod, proč není DTD pro datově orientované dokumenty vhodný. Další problémem může být to, že schéma není psáno v XML, nýbrž speciální syntaxí. (Herout, 2007)

```
<!ELEMENT produkt (název, datumVýroby, datumExpirace)>
<!ATTLIST produkt
          xmlns CDATA #FIXED "http://www.balon.org/2014"
          cena CDATA #REQUIRED
          dph CDATA #REQUIRED
          sazbaDph (15 | 21) #REQUIRED>
<!ELEMENT název (#PCDATA)>
<!ELEMENT datumVýroby (#PCDATA)>
<!ELEMENT datumExpirace (#PCDATA)>
```

Schéma k modelovému dokumentu jasně definuje elementy, atributy i jejich strukturu. Na příkladu je vidět slabá podpora datových typů (prakticky pouze znakový typ `CDATA` nebo parsovatelný znakový typ `PCDATA`), nutnost definice implicitního jmenného prostoru fixním atributem a jediná možnost provedení skutečného omezení obsahu - seznamem hodnot u definice atributu `sazbaDph`.

DTD se pro datově orientované dokumenty stále používá spíše z historických důvodů, kdy nebylo možné použít jinou alternativu a přepsání do jiného jazyka v současnosti již není nutné. Aplikace s takovými dokumenty pracující již obsahují mechanismy, které nevýhody DTD odstraňují, například kontrolu typu vstupních dat. Pro nové dokumenty však není vhodné tento jazyk z výše uvedených důvodů používat vůbec.

W3C XML Schema

W3C XML Schema (W3C, 2004b) je, stejně jako XML, dílem W3C konsorcia. Lze jej najít pod zkratkou WXS, častěji však jako XSD (XML Schema Definition).

Jazyk XSD je velice komplexní, schéma se zapisuje do XML, takže je možné kontrolovat správnost zápisu, nevýhodou je však skutečnost, že je v mnoha případech XML soubor se schématem větší, než soubor XML na základě tohoto schématu vytvořený. (Herout, 2007)

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.balon.org/2014"
    xmlns="http://www.balon.org/2014">
    <xs:simpleType name="sazbaDphType">
        <xs:restriction base="xs:nonNegativeInteger">
            <xs:enumeration value="15" />
            <xs:enumeration value="21" />
        </xs:restriction>
    </xs:simpleType>

    <xs:complexType name="produktType">
        <xs:sequence>
            <xs:element name="název" type="xs:string" />
            <xs:element name="datumVýroby" type="xs:date" />
            <xs:element name="datumExpirace" type="xs:date" />
        </xs:sequence>
        <xs:attribute name="cena" type="xs:decimal" />
        <xs:attribute name="dph" type="xs:decimal" />
        <xs:attribute name="sazbaDph" type="sazbaDphType" />
    </xs:complexType>
    <xs:element name="produkt" type="produktType" />
</xs:schema>
```

Příklad XSD schématu k modelovému dokumentu jasně ukazuje jeho největší nevýhodu popsanou výše. Schéma několikanásobně převyšuje velikost dokumentu. Ve všech ohledech však splňuje definici schématu založeného na gramatice. Jasně definuje elementy, atributy, strukturu i datová omezení. Jasně je zde definován i cílový jmenný prostor dokumentu, který je rovnou nastaven na implicitní.

Tento jazyk je velice robustní a pro datově orientované XML dokumenty velice vhodný, další argument pro jeho výběr je všeobecné doporučení jak konsorciem W3C, tak z důvodu podpory velkých korporací. (Kosek, 2013)

Protože je XSD hlavním předmětem této práce, jsou detailní informace uvedeny v samostatné kapitole 3.

RELAX NG

Technologie RELAX NG (RELAX NG, 2014) je referenční implementace schématu založeného na gramatice standardu DSDL (druhá část). RELAX NG vznikl krátce po vzniku XSD, a jeho cílem je poskytnout funkcionalitu XSD v jednodušší podobě.

Jazyk je jednoduchý na naučení. Má dvě rovnocenné reprezentace, XML a kompaktní. Reprezentace formou XML je určena pro strojové zpracování, kompaktní reprezentace pro návrh schématu lidmi. Před použitím je kompaktní schéma vždy převedeno do XML reprezentace. Elementy a atributy jsou považovány za rovnocenné. (Fawcett et al., 2012)

RELAX NG jako takové nepodporuje datové typy, obsahuje však sofistikovaný mechanismus, který nám dovoluje „dodat“ datové typy z jiného jazyka, například XSD. (Kosek, 2013)

Oproti XSD, které je založeno na datových typech, je RELAX NG založen na porovnávání vzorů (pattern matching). Schéma abstraktně reprezentuje stromovou strukturu cílového dokumentu, se kterou se musí shodovat. (Fawcett et al., 2012)

```
default namespace = "http://www.balon.org/2014"
datatypes xsd = "http://www.w3.org/2001/XMLSchema-datatypes"

element produkt {
    attribute cena {xsd:decimal},
    attribute dph {xsd:decimal},
    attribute sazbaDph {"15" | "21"},
    element název {xsd:string},
    element datumVýroby {xsd:date},
    element datumExpirace {xsd:date}
}
```

V ukázce schématu pro modelový příklad je uveden zápis kompaktní reprezentace. Příklad ukazuje jednoduchost definice implicitního jmenného prostoru a importu datových typů z XSD. Dále je zde demonstrován rovnocenný přístup k atributům a elementům (liší se jen klíčovými slovy).

Jakkoli se RELAX NG zdá být lepší volbou, než XSD, jeho použití je vykoupeno všeobecnou absencí podpory ze strany velkých výrobců softwaru.

Zatímco XSD je integritní součástí práce s XML v moderních vývojových platformách (např. Java a .NET), je pro validaci oproti RELAX NG schématu nutno sehnat odpovídající knihovnu třetí strany. Tato skutečnost bohužel staví RELAX NG ve své kategorii na pomyslnou druhou pozici.

2.3.2 Schémata založená na pravidlech

Zatímco schématem založeným na gramatice dokonale popíšeme model dokumentu, nemáme přesně zajištěno co, kde a za jakých podmínek se v dokumentu objeví. V modelovém příkladu této kapitoly jsme pomocí XSD jasně definovali atribut `sazbaDph` výčtem hodnot, které může nabývat. Nemáme ale nijak zajištěno, že atribut `dph` bude skutečně odpovídat hodnotě DPH vypočtené z ceny uvedené v atributu `cena` s použitím vybrané sazby. Stejně tak nemáme zajištěno, že `datumExpirace` bude nastaven na datum, který následuje po `datumVýroby`. Schémata založená na pravidlech slouží právě k definici těchto konkrétních pravidel.

Ve třetí části ISO standardu DSDL je definována technologie Schematron (Schematron, 2010). Tato technologie je postavená nad technologiemi XPath a XSLT, o kterých pojednávají kapitoly 2.4 a 2.5. Schéma napsané ve Schematronu se sadou transformací převede na transformační scénář, který se aplikuje na validovaný dokument. Výstupem validace je XML dokument nazývaný SVRL (Schematron Validation Report Language). Z něj je možné pomocí některého XML parseru (kap. 2.2) jednoduše získat informace o průběhu validace.

Jazykem XPath se zapisují podmínky, které v dokumentu testujeme. Můžeme buďto provádět aserci přítomnosti vzoru v dokumentu (`assert`) nebo reporting výskytu vzoru, který v dokumentu nesmí figurovat (`report`). Chybové hlášky, které se objeví v SVRL, přitom definuje sám návrhář schématu. (Fawcett et al., 2012)

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron"
        queryBinding="xslt2">
 <pattern>
  <rule context="/produkt">
   <assert test="@dph = (@cena * (@sazbaDph div 100))">
    DPH musí mít hodnotu
    <value-of select="@cena * (@sazbaDph div 100)"/>
   </assert>
   <report test="datumVýroby &gt; datumExpirace">
    Datum výroby nemůže nastat dříve, než datum expirace.
   </report>
  </rule>
 </pattern>
</schema>
```

Ukázkové schéma definuje pravila, která byla uvedena v úvodu této podkapitoly. Příklad demonstruje použití aserce a reportingu s definicí vlastních chybových hlášek.

Technologie Schematron umožňuje realizovat omezení dokumentu, která schémata založená na gramatice neumožňují. Ve většině dokumentů se nějaké takové závislosti objevují, proto je vhodné pro tyto dokumenty definovat minimálně jedno schéma založené na gramatice (například XSD) a schéma založené na pravidlech – Schematron. Přitom nemusíme brát zřetel na podporu Schematronu v aplikacích, protože je založen na technologii XSLT, která všeobecně podporována je.

2.3.3 Schémata založená na jmenných prostorech

NVDL (Namespace-based Validation Dispatching Language) (NVDL, 2009) je třetím typem schémového jazyku specifikovaného ve čtvrté části ISO standardu DSDL. Tento jazyk slouží primárně pro specifikaci validačního scénáře dokumentu složeného z více jmenných prostorů. Každý jmenný prostor je přitom reprezentován vlastní množinou schémat. Jazyk sám o sobě nedefinuje žádná dodatečná omezení. Proto se schématu NVDL říká meta-schéma, které validuje XML dokument pomocí sady subschémat na základě definovaných pravidel.

Pro použití NVDL nemusíme mít nutně komponovaný dokument z několika jmennými prostory, ale lze jej použít i pro definici množiny schémat, která se mají aplikovat na jednoduchý dokument, například náš modelový příklad. (Kosek, 2013)

```
<?xml version="1.0" encoding="UTF-8"?>
<rules xmlns="http://purl.oclc.org/dsdl/nvdl/ns/structure/1.0">
    <namespace ns="http://www.balon.org/2014">
        <validate schema="document.xsd" />
        <validate schema="document.sch" />
    </namespace>
</rules>
```

Schéma zcela jasně definuje, že se má dokument obsahující jmenný prostor `http://www.balon.org/2014"` validovat jak pomocí schématu v jazyce XSD (`document.xsd`), tak pomocí Schematronu (`document.sch`). Na jednom místě jsme tak definovali pravidlo, které říká za jakých podmínek je dokument validní, aniž bychom jakkoli zasahovali do zdrojového dokumentu.

2.4 XPath

Jazyk XPath (XML Path Language) (W3C, 2011) je základem technologie XSLT, popisované v následující kapitole, a dalších návazných technologií (XPointer, Schematron, ...). Smyslem jazyka je adresace uzlů v XML dokumentu pomocí výrazů. Operuje při tom nad abstraktní, logickou strukturou dokumentu, která se nazývá „data model“. Tato struktura dokumentu je podobná infosetu, využívaném technologií DOM (kap. 2.2.2).

Výsledkem XPath výrazu může být vyhovující množina uzlů nebo atomická hodnota. Jazyk tak lze použít pro hromadné zpracování více uzlů nebo k jednoznačné identifikaci jednoho uzlu.

Syntaxe jazyka je podobná syntaxi adresace adresářové struktury v UNIX systémech. Jednotlivé úrovně stromové struktury XML dokumentu jsou oddělovány dopředným lomítkem (`/`), to zároveň reprezentuje kořen dokumentu (nadřazen kořenovému elementu). Pro jednoznačnou identifikaci elementu v dané úrovni stromu, lze do hranatých závorek přidat jeho index, začínající od 1 (`/element[1]`).

Pro adresaci atributů se používá znak `@`. Atribut je považován za podřízený uzel elementu, ve kterém je uveden. Jednoznačná adresace atributu v kořenovém elementu má například následující podobu `/element[1]/@atribut`. Atribut není nutné identifikovat indexem, protože element nemůže obsahovat dva atributy se stejným jménem.

Tento základ syntaxe je pouze velice úzká podmnožina možností, které jazyk XPath poskytuje. Zároveň se jedná o část jazyka použitou přímo v této práci pro jednoznačnou identifikaci uzlů v XML dokumentu.

2.5 XSLT

Technologie XSLT (eXtensible Stylesheet Language Transformations) (W3C, 2007) je standardem konsorcia W3C sloužící k transformaci XML dokumentu do jiného formátu (HTML, PDF, SVG, jiné XML, ...). K provedení transformace je nutné v XSLT stylopisu definovat pravidla převodu XML dokumentu. Na základě tohoto popisuje poté můžeme pomocí XSLT procesoru transformovat XML dokument.

XSLT stylopis je XML dokument, proto je na něj možné aplikovat jinou transfromaci, jejíž výsledkem bude třetí transformace. (Tidwell, 2008) Této skutečnosti využívá například validace pomocí jazyka Schematron popisovaná v kapitole 2.3.2.

Tato technologie je alespoň ve verzi 1.0 dostupná v moderních programovacích jazycích a zabudovaný procesor obsahují i majoritní webové prohlížeče. Pokud bychom chtěli využít pokročilé techniky XSLT 2.0 je nutné použít knihovny třetích stran. Nejrozšířenějším XSLT procesorem je Saxon (Kay, 2013), který je možné použít pod platformou Java a .NET. Verze CE je implementace procesoru Saxon napsaná v Javascriptu a určená pro webové prohlížeče.

3 W3C XML Schema

Tato kapitola detailněji popisuje technologii W3C XML Schema (XSD) a volně navazuje na přehledovou kapitolu 2.3 o XML schématech. Jejím cílem není vyčerpávajícím způsobem technologii popsat, ale uvést klíčové oblasti použité v této práci. XSD schéma je XML dokumentem vytvořeným dle specifikace W3C ve jmenném prostoru `http://www.w3.org/2001/XMLSchema`, obvykle označeným prefixem `xs`.

Kořenovým elementem schématu je `xs:schema`. Jaké elementy mají být potomkem kořenového elementu se řídí zvoleným návrhovým vzorem schématu (viz kap. 3.3). Elementy uvedené v úrovni bezprostředně pod kořenovým elementem se nazývají globální deklarace, více zanořené poté lokální deklarace.

Elementy cílového dokumentu jsou ve schématu reprezentovány elementem `xs:element`, atributy `xs:attribute`. Název prvku, pod kterým bude v cílovém dokumentu vystupovat, je povinně uveden v atributu `name`.

Protože je XSD založeno na datových typech (kap. 3.1), je nutné specifikovat typ každého elementu a atributu dokumentu. První možností je použití atributu `type` příslušného prvku. Tato možnost lze použít pro vestavěné datové typy nebo globálně deklarované uživatelské typy. Druhou možností je lokální deklarace uživatelského typu jakožto potomka příslušného elementu. Následující příklad ukazuje základní deklaraci elementu `název` vestavěného typu řetězec.

```
<xs:element name="název" type="xs:string" />
```

Deklarace elementu a atributu může obsahovat mnohé další atributy. Důležitý je atribut `ref`, pomocí kterého je možné navázat lokální deklaraci prvku na globální deklaraci a `fixed` pro specifikaci fixní (neměnné) hodnoty.

Deklarace atributu navíc obsahuje velice důležitý atribut `use`, kterým specifikujeme povinnost výskytu deklarovaného atributu. Výchozí hodnota `use` je bohužel „`optional`“ – volitelný, proto je vždy nutné tento atribut uvést s hodnotou „`required`“ – povinný.

3.1 Datové typy

Schémata XSD jsou založena na datových typech, které jsou vyhrazeny v samostatné části specifikace XSD (W3C, 2004c). Každý prvek XML dokumentu (element, atribut) jej musí mít explicitně specifikován.

Datovým typem může obecně být vestavěný typ nebo globálně či lokálně deklarovaný uživatelsky typ. Vestavěné typy poskytují primitivní datové typy pro specifikaci řetězců (`xs:string`), čísel (`xs:decimal`), logických hodnot (`xs:boolean`), hodnot dat a časů (`xs:dateTime`) a další. Od primitivních typů jsou odvozeny další vestavěné typy. Jejich kompletní přehled je uveden v příloze A na obrázku A.1, převzatém ze specifikace XSD.

Uživatelsky deklarované datové typy se dělí na jednoduché (simple) a komplexní (complex). (Kosek, 2013)

3.1.1 Jednoduchý datový typ

Jednoduchý datový typ se používá ke specifikaci obsahu koncového elementu nebo atributu. Ve většině případů nám nestačí pro tyto prvky definovat vestavěný datový typ, ale chceme jej nějakým způsobem co nejvíce omezit (provést restrikci). To nám právě jednoduché typy umožňují. Ve spolupráci s vestavěnými datovými typy vytváříme restrikcí nový datový typ, jehož popisem se musí obsah daného prvku řídit. Dobrým zvykem je deklarovat jednoduchý typ pro všechny koncové prvky dokumentu a to i v případě, že chceme použít pouze vestavěný datový typ bez dalších integritních omezení.

Deklaraci jednoduchého typu reprezentujeme elementem `xs:simpleType`. V případě globální deklarace je nutné v atributu `name` uvést jméno nového typu, kterým se na něj budeme moci odkazovat. Lokální deklaraci není nutné pojmenovat, protože je přímo svázána s prvkem, který ji používá. Konvence pojmenování typů je kombinace jména prvku, pro který typ definujeme a postfixu „`Type`“. Uveďme příklad deklarace jednoduchého typu pro atribut `sazbaDph`, který se dle konvence bude jmenovat `sazbaDphType`.

```
<xs:simpleType name="sazbaDphType">
  zde popis typu
</xs:simpleType>
```

Uvnitř elementu `xs:simpleType` následně definujeme jeden element pro popis cílového datového typu. Popis můžeme provést seznamem (`xs:list`), sjednocením typů (`xs:union`) nebo pomocí restrikce (`xs:restriction`).

Popis seznamem slouží k definici obsahu, který je tvořen výčtem hodnot určitého typu, oddělených mezerou. Takový obsah je velice neobvyklý, neboť zcela odporuje filosofii XML. Definici typu jednotlivých hodnot, tvořících seznam provedeme v atributu `itemType` za použití vestavěných datových typů, nebo dříve deklarovaných globálních jednoduchých typů. Druhou variantou je provedení lokální deklarace jednoduchého datového typu uvnitř elementu `xs:list`.

Sjednocení jednoduchých typů použijeme v případě že chceme, aby prvek mohl obsahovat heterogenní obsah (různých typů). To je pro XML také poměrně neobvyklé. Sjednocované globální jednoduché typy a vestavěné typy můžeme výčtem s mezerami zapsat do atributu `memberTypes`, lokální jednoduché typy deklarujeme uvnitř elementu `xs:union`.

Nejpoužívanějším typem popisu jednoduchého typu je restrikce. Ta omezí základní (base) datový typ pomocí integritních omezení. Základní datový typ je některý vestavěný typ nebo jiný globálně deklarovaný jednoduchý typ. Ten uvedeme v povinném atributu `base`. Obsahem restrikce jsou poté nepovinná integritní omezení, která se vztahují k základnímu datovému typu. Pro popis omezení atributu `sazbaDph` ze dříve uvedeného příkladu, použijeme vestavěný datový typ `xs:nonNegativeInteger`, protože požadujeme kladnou celočíslenou hodnotu.

```
<xs:restriction base="xs:nonNegativeInteger">
  zde integritní omezení typu
<xs:restriction>
```

Integritní omezení

Použitelná integritní omezení se řídí zvoleným základním datovým typem. Některá omezení tak lze aplikovat na textové řetězce, jiná třeba na číselné hodnoty. Vybraná omezení lze k dosažení požadovaného výsledku kombinovat. Všechna integritní omezení se definují pomocí speciálně pojmenovaného elementu s atributem `value`, kterým předáme parametr daného omezení.

Omezení použitelná pouze pro textové řetězce jsou omezení délky. Pro určení přesné délky řetězce použijeme element `xs:length`. Definovat lze i minimální (`xs:minLength`) nebo maximální (`xs:maxLenght`) délku řetězce. Parametrem všech těchto omezení je kladná celočíselná hodnota.

Pro všechny číselné hodnoty lze definovat minimální a maximální přípustnou hodnotu ve dvou variantách. Buďto včetně hodnoty uvedené v parametru pomocí elementů `xs:minInclusive` a `xs:maxInclusive`, nebo bez této hodnoty elementy `xs:minExclusive` a `xs:maxExclusive`. Dále je možné určit maximální celkový počet číslic `xs:totalDigits` a pro desetinná čísla i maximální počet číslic desetinné části (`xs:fractionDigits`). Pro příklad s atributem `sazbaDph` bychom mohli použít omezení maximální hodnoty, protože daňová sazba je udávána v procentech s maximální hodnotou 100.

```
<xs:maxInclusive value="100" />
```

Omezeními použitelnými pro všechny vestavěné datové typy jsou výčet hodnot (`xs:enumeration`) a vzor pomocí regulárního výrazu (`xs:pattern`). Výjimkou je typ `xs:boolean`. Výčtem hodnot se rozumí množina elementů `xs:enumeration` uvnitř elementu `xs:restriction`, přičemž každý obsahuje právě jednu hodnotu. Vhodnější integritní omezení atributu `sazbaDph` z našeho příkladu tak logicky představuje výčet hodnot, neboť aktuální právní úprava povoluje právě dvě sazby, 15 a 21 %.

```
<xs:enumeration value="15" />
<xs:enumeration value="21" />
```

3.2 Komplexní datový typ

Komplexní datové typy se používají k definici obsahu elementů, které obsahují atributy nebo další elementy jako potomky. Deklarace komplexního typu je nástrojem k modelování struktury dokumentu.

Komplexní typ deklarujeme elementem `xs:complexType`, který má v případě globální deklarace povinný atribut `name` s jednoznačným pojmenováním typu. Jmenná konvence je stejná jako u jednoduchých typů. Uvnitř elementu se následně nejprve definuje obsah cílového elementu a poté výčet jeho atributů. Jako příklad uvádíme deklaraci komplexního typu elementu `produkt`.

```
<xs:complexType name="produktType">
   zde definice obsahu
   zde výčet atributů
<xs:complexType>
```

3.2.1 Definice obsahu

Definice obsahu cílového elementu má čtyři varianty. První variantou je neuvádět nic pro případ prázdného elementu nebo prázdného elementu s atributy.

Nejčastější variantou definice obsahu je použití některého z kompozitorů `xs:sequence`, `xs:all` a `xs:choice`. Uvnitř něj definujeme seznam elementů tvořících obsah a podle zvoleného kompozitoru vynucujeme určité chování. Tuto variantu zvolíme v případě, že cílový element má podřízené elementy a nechceme rozšiřovat dříve deklarovaný komplexní typ. (Kosek, 2013)

Kompozitor `xs:sequence` vynucuje přesné pořadí elementů v něm uvedených, `xs:all` požaduje použití všech uvedených elementů, ale v libovolném pořadí a `xs:choice` slouží k výběru jednoho z definovaných elementů.

Uvnitř kompozitorů lze navíc v deklaracích elementů `xs:element` používat atributy pro definici minimálního a maximálního počtu povolených výskytů `minOccurs` a `maxOccurs`. Pro neomezený počet výskytů použijeme hodnotu `unbounded`. Tyto atributy lze využít i nad samotnými kompozitory pro jejich řetězení.

Třetí variantou definice je jednoduchý obsah (`xs:simpleContent`). Ten použijeme v případě, že cílový element neobsahuje žádné podřízené elementy, ale je to koncový element s atributy. Rozšířením (`xs:extension`) jednoduchého typu, který reprezentuje obsah tohoto elementu, můžeme dodefinovat potřebné atributy. Rozšiřovaný typ uvedeme v atributu `base`.

Poslední varianta je komplexní obsah (`xs:complexContent`). Slouží k rozšíření dříve deklarovaných komplexních typů o další elementy opět pomocí `xs:extension`. Simuluje se tak princip dědičnosti. Množina přidávaných elementů se uvnitř rozšíření definuje pomocí kompozitoru.

3.3 Návrhové vzory

Návrhové vzory schématu řeší problém s rozhodnutím, kdy má být element nebo typ deklarován globálně, a kdy lokálně. Postupem času se vyvinuly tři návrhové vzory:

- ruská panenka – matrjoška (Russian Doll Design),
- salámová kolečka (Salami Slice Design),
- slepý Benátčan (Venetian Blind Design).

(Kosek, 2013; xFront, 2006)

3.3.1 Russian Doll Design

Schéma dle vzoru matrjoška svou strukturou kopíruje strukturu cílového dokumentu, kdy jsou všechny deklarace elementů a atributů zanořeny do jediné globální deklarace kořenového elementu. Jednotlivé úrovně jsou do sebe zanořeny, jako je tomu u ruské panenky, odtud vzešlo pojmenování tohoto vzoru. Všechny vnořené deklarace včetně deklarace typů jsou zde lokální.

Nevýhodou tohoto přístupu je úplná neprůhlednost schématu bez možnosti jakékoli znovupoužitelnosti, vše je lokální. Výhodou je naopak kompaktnost schématu a skutečnost, že jeho změny neovlivňují návazné elementy (žádné nejsou, nejde to). (Kosek, 2013; xFront, 2006)

3.3.2 Salami Slice Design

Tento vzor je naprostým opakem vzoru matrjoška. Všechny deklarace elementů a atributů jsou globální a schéma je sestaveno pomocí referencí (atribut `ref`). Problémem je, že není jasně definován kořenový element cílového dokumentu, může jím být kterýkoli z globálně deklarovaných (zde všechny). Název je odvozen od salámu, který je teoreticky tvořen jednotlivými kolečky slepenými dohromady. Kolečka salámu jsou jednotlivé globální deklarace, které jsou pomocí referencí spojeny do celého schématu – salámu. Definice datových typů jsou v tomto případě opět lokální.

Výhodou tohoto přístupu je znovupoužitelnost elementů a atributů v jiných schématech. Schéma je také velice přehledné. Nevýhodou je provázání elementů a atributů s jejich typy napříč všemi dokumenty, kde byly použity. Nelze tak například mít v jednom dokumentu dvakrát elementy stejného jména a různého obsahu. Provázání také přináší problém, kdy změna jedné deklarace ovlivní neznámý počet jiných. (Kosek, 2013; xFront, 2006)

3.3.3 Venetian Blind Design

Přístup slepý Benátčan je od předchozích dvou odlišný a z každého z nich si přináší část vlastností. Namísto globálních nebo lokálních prvků používá globální typy. Každý element a atribut má globálně deklarovaný datový typ. Na konci schématu je uveden jeden globálně deklarovaný element, který je tak jednoznačně daný. Ostatní elementy a atributy jsou uvnitř globálních deklarací typů automaticky lokální.

Výhodou tohoto přístupu je maximální znovupoužitelnost typů. To naprosto odstraňuje nevýhody matrjošky (žádná znovupoužitelnost) a salámových koleček (typ svázán s deklarací). Problémem je opět navázání neznámého počtu prvků a trochu větší složitost. (Kosek, 2013; xFront, 2006)

Vzor slepý Benátčan je ve většině případů nejlepší volbou. Nevhodný je snad pouze v případech, kdy vyžadujeme co nejmenší velikost schématu, což není častým požadavkem. (Kosek, 2013)

4 WPF a XAML

Tato kapitola popisuje použitou technologii pro tvorbu grafických uživatelských rozhraní – GUI (Graphical User Interface) – pro desktopové aplikace operačního systému Microsoft Windows. Technologie se nazývá WPF (Windows Presentation Foundation) (MSDN, 2014d) a je součástí Microsoft .NET Framework verze 3.0 (MSDN, 2014a) a vyšší.

4.1 Základní přístupy k tvorbě GUI

Současné technologie poskytují dva základní způsoby tvorby grafických uživatelských rozhraní pro desktopové aplikace MS Windows.

První způsob spočívá v návrhu rozhraní i jeho aplikační logiky pomocí nástrojů poskytovaných programovacími jazyky. Tento přístup k návrhu je obecně k dispozici ve všech současných technologiích pro tvorbu GUI. Jeho problémem je míchání návrhu grafického rozhraní s aplikační logikou, což je v rozporu s moderními přístupy vývoje softwaru.

V prostředí programovacího jazyka Java se k návrhu uživatelského rozhraní tímto přístupem používá technologie Swing (Oracle, 2014c). Prostředí .NET poskytuje technologii WinForms (MSDN, 2014c).

Druhým způsobem je v současné době velice populární návrh s využitím technologie XML. Návrh GUI je logicky a přehledně zapsán do speciálního XML dokumentu, který částečně kopíruje strukturu navrhovaného grafického rozhraní. Aplikační logika rozhraní se následně definuje v odděleném programu. Právě v oddělení návrhu od chování spočívá popularita tohoto přístupu. Programátor aplikace se stará pouze o její funkcionalitu, nikoli vzhled. Grafický návrhář pak řeší pouze vhled aplikace a to rovnou pomocí dostupných komponent bez toho, aby se musel učit programovat.

Programovací jazyk Java tuto technologii nativně neobsahuje, existují však více či méně kvalitní knihovny třetích stran. Prostředí .NET obsahuje, jak již bylo řečeno, popisovanou technologii WPF.

4.2 WPF

WPF je moderní zobrazovací technologie pro operační systém Windows. Technologie pracuje s hardwarovou akcelerací DirectX a podporuje nezávislost návrhu na cílovém rozlišení. Je to jediná technologie, která je na míru navržená pro spolupráci se systémem Windows, a to již od verze XP.

Další výhodou WPF je plná kontrola nad vzhledem všech komponent. Starší technologie měly pevně danou množinu komponent s fixním vzhledem. WPF má také definovanou funkcionalitu komponent spolu s jejich vzhledem, ale tento vzhled lze libovolně měnit.

Poslední výhodou je již zmíněná možnost deklarativního návrhu rozhraní do XML. K dispozici je však i možnost návrhu programově, což není doporučováno. XML pro návrh grafického rozhraní se nazývá XAML (eXtensible Application Markup Language). (MacDonald, 2012)

4.3 XAML

XAML (výslovnost „zaml“) je značkovací jazyk používaný pro instanciaci .NET objektů. Primární roli však hraje v technologii WPF pro konstrukci uživatelského rozhraní, kde je použita podmnožina WPF XAML. (MacDonald, 2012)

Jakmile si uvědomíme základní pravidla, je standard XAML poměrně jednoduchý na pochopení. Každý element se mapuje na instanci třídy, jejíž název se shoduje s názvem elementu. Jako v každém dokumentu XML je možné vnořovat elementy do sebe, čímž vyjádříme vztah podřízenosti objektů. Posledním pravidlem je, že členské proměnné objektu lze nastavit pomocí stejně pojmenovaného atributu nebo speciálně pojmenovaným podřízeným elementem. Pojmenování speciálního elementu je složeno ze jména nadřízeného elementu a členské proměnné, oddělené tečkou. (MacDonald, 2012)

4.3.1 Základní syntaxe

Základní syntax návrhu okna v XAML vysvětlíme následujícím příkladem.

```
<Window x:Class="TridaAplikacniLogiky"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Titulek okna" Height="500" Width="300">

   zde komponenty obsahu okna

</Window>
```

Všechny elementy v XAML musí začínat velkým písmenem, jako je tomu u názvů tříd. Pro vytvoření okna použijeme kořenový element `Window`, kterému musíme definovat dva v příkladu uvedené jmenné prostory, čímž se z obyčejného XML stane XAML. Implicitní jmenný prostor definuje všechny komponenty WPF. Prostor s prefixem `x` přidává další užitečnou funkcionalitu pro ovlivnění interpretace dokumentu.

Pomocí atributu `x:Class` rovnou svážeme XAML s jeho aplikační logikou, uvedením názvu třídy, která ji obsahuje. Příklad také ukazuje definici některých členských proměnných jakožto atributů.

Pro komunikaci s komponentami uživatelského rozhraní z aplikační logiky je vhodné definovat jejich unikání identifikátor atributem `x:Name` nebo jen `Name`. Pomocí tohoto pojmenování můžeme unikátně adresovat jednotlivé komponenty, číst jejich hodnoty nebo s nimi jinak programově manipulovat. (Moser, 2011)

4.3.2 Načítání a kompilace

Protože jsou XAML a WPF oddělené technologie, je nutné je nějakým způsobem propojit. Díky jejich oddělení je také možné vytvořit WPF aplikaci bez použití XAML. Propojení je možné provést dvěma způsoby, přičemž preferovaným je kompilace XAML do optimalizované binární podoby (BAML). Propojení se provede automaticky, čtení BAML je rychlejší a zároveň se přibalí do spustitelného souboru, takže jej není nutné distribuovat zvlášť.

Druhou možností je využití nekompilovaného XAML. To nám umožní propojení aplikace s předem připraveným XAML přímo za běhu programu pomocí `XAMLReader`. Načtený dokument může být celé okno, které rovnou zobrazíme, nebo jen podstrom, který připojíme do jiného, například kompilovaného okna. Protože připojovaný podstrom není kompilovaný, nelze k jeho pojmenovaným komponentám v aplikační logice přistupovat přímo, ale pomocí `LogicalTreeHelper`. Nevýhodou tohoto přístupu je nutnost XAML ručně načítat a připojovat, složitější přístup k pojmenovaným komponentám a nižší rychlost vykreslování.

Pro většinu aplikací, které mají staticky navržené uživatelské rozhraní, použijeme kompilovaný XAML. Pro aplikace, ve kterých potřebujeme za běhu vyměňovat části rozhraní, použijeme nekompilovaný XAML. Ten použijeme zejména v případě, kdy celé nebo část XAML generujeme až za běhu programu. (MacDonald, 2012)

4.4 Kontejnery rozložení

Kontejnery rozložení (layouty) slouží pro automatické uspořádání jejich obsahu. Každý druh layoutu má jinou vnitřní logiku, jak komponenty rozmisťuje. Některé skládají komponenty za sebe, jiné do mřížky, jejich cílem je však vždy korektní zobrazení komponent v prostředích s různým rozlišením obrazovky. Proto se nedoporučuje používat absolutní zarovnávání, ale vhodný layout, který to provede automaticky a korektně.

V následujícím textu detailněji popíšeme layouty `Grid` a `DockPanel`, které byly přímo použity v této práci.

4.4.1 Grid

Mřížka, neboli `Grid`, je nejmocnější, ale také nejsložitější WPF layout. Slouží k rozdělení vymezené oblasti na definovaný počet řádků a sloupců. Do vzniklých buněk můžeme vkládat buďto jednu komponentu přímo, nebo více komponent zapouzdřených v jiném layoutu (jinak by se překrývaly).

Definice řádků a sloupců je na první pohled složitá, má ale svůj význam. Řádky se definují jako členská proměnná mřížky v podřízeném ele-

mentu `Grid.RowDefinitions`. Definicí řádky je element `RowDefinition` a počet těchto elementů určuje počet řádků v mřížce. Tím je umožněno nastavit každé řádce odlišné vlastnosti. Sloupce se definují obdobně elementy `ColumnDefinition` uvnitř členské proměnné `Grid.ColumnDefinitions`.

Komponenty, které chceme do mřížky vsadit, je nutné umístit do elementu `Grid`. Navíc jim musíme atributy `Grid.Row` (řádek) a `Grid.Column` (sloupec) explicitně specifikovat číslo řádku a sloupce, kam mají být umístěny.

Následující příklad reprezentuje mřížku se dvěma řádky a dvěma sloupci, která má v pravé dolní buňce umístěné tlačítko.

```
<Grid>
   <Grid.RowDefinitions>
      <RowDefinition />
      <RowDefinition />
   </Grid.RowDefinitions>
   <Grid.ColumnDefinitions>
      <ColumnDefinition />
      <ColumnDefinition />
   </Grid.ColumnDefinitions>
   <Button Grid.Row="1" Grid.Column="1">Tlačítko</Button>
</Grid>
```

4.4.2 DockPanel

`DockPanel` slouží k zarovnávání (dokování) vnitřních komponent k vnějším hranám kontejneru. Ke které hraně se má zanořená komponenta zarovnat specifikujeme pomocí atributu `DockPanel.Dock` jednou z hodnot Top (horní), Bottom (dolní), Left (levá), Right (pravá). Tento layout je vhodné použít, chceme-li řízeným způsobem zaplnit celou vyhrazenou plochu.

Příklad layoutu zobrazuje jeho použití pro dvě tlačítka, která vyplňují celý prostor, přičemž jedno je vlevo a druhé vpravo.

```
<DockPanel>
   <Button DockPanel.Dock="Left">Levé</Button>
   <Button DockPanel.Dock="Right">Pravé</Button>
</DockPanel>
```

4.5 Základní komponenty

Vizuálním komponentám se ve WPF říká kontrolky (controls), protože jsou potomky třídy `Control`. Podle jejich vlastností se dělí do šesti skupin. Základní komponenty obsahují všechny běžně používané vizuální prvky, jako štítky (`Label`), textová pole (`TextBlock`), tlačítka (`Button`), zaškrtávací boxy (`CheckBox`) a další. Pro vstup textu je k dispozici základní komponenta `TextBox`.

Komponentu `ComboBox` použijeme v případě, že chceme z větší množiny položek vybrat právě jednu. `ComboBox` je klasická komponenta, která zobrazuje vybranou hodnotu a při kliknutí zobrazí roletovou nabídku pro výběr jiné alternativy. Ty jsou definované kolekcí elementů `ComboBoxItem`.

4.5.1 Speciální kontejnery

Expander

Komponenta `Expander` se řadí do skupiny kontrolek s hlavičkou a obsahem. Umožňuje rozbalit (expand) nebo skrýt svůj obsah. Hlavička komponenty je vidět vždy spolu s indikátorem rozbalení a definuje se atributem `Header`. Obsah se zpravidla definuje zanořeným elementem, nebo atributem `Content`.

Obsahem i hlavičkou může být pouze jedna komponenta. Pokud jich chceme vložit více, je nutné je umístit do layoutu. Obsahem potažmo můžou být další komponenty `Expander` a tím nám vzniká jedinečný nástroj pro modelování vizualizace stromové struktury.

Pro lepší představu o funkcionalitě komponenty uvádíme příklad jejího použití, spolu s obrázkem 4.1, který zobrazuje možné stavy komponenty.

```
<Expander Header="Hlavička">
  <Label>Skrytý text</Label>
</Expander>
```

Obrázek 4.1: Stavy komponenty `Expander`

ScrollViewer

Pokud chceme umístit větší množství obsahu do menší plochy, je nutné ji opatřit posuvníky. Ty nám následně umožní posunem zpřístupnit veškerý obsah. Tuto funkcionalitu plní kontejner `ScrollViewer`, do kterého stačí požadovaný obsah umístit a v případě, že se tam nevejde, automaticky se potřebné posuvníky vykreslí.

Menu

Hlavní nabídka aplikace se tvoří pomocí elementu `Menu`. Hlavní menu musíme explicitně, například pomocí layoutu `DockPanel`, umístit k horní hraně okna. Položky hlavní nabídky tvoříme pomocí `MenuItem` s popiskem uvedeným v atributu `Header`. Podřízené položky se vytvářejí stejným způsobem zanořením do rodičovské položky.

4.6 Uživatelsky definované komponenty

WPF umožňuje z dostupných komponent sestavit kompozitní znovupoužitelnou komponentu. Tu definujeme jako element `UserControl` ve zvláštním souboru. Takováto komponenta je zcela autonomní s vlastní aplikační logikou. (Moser, 2011)

Uživatelsky definovanou komponentu použijeme v případě, že chceme vyčlenit část uživatelského rozhraní do vlastního souboru, nebo danou komponentu budeme využívat na více místech programu. Jako uživatelsky definované komponenty je vhodné použít i nekompilované části XAML, které připojujeme nebo generujeme za běhu programu (viz kap. 4.3.2).

4.7 Rozšiřující knihovny

Během realizace této práce jsme narazili na problémy, které nešly jednoduše řešit standardními komponentami jazyka. Proto bylo potřeba použít některé rozšiřující knihovny, které potřebné komponenty obsahovaly.

4.7.1 Extended WPF Toolkit

Pro rozšíření nabídky potřebných kontrolek byla použita knihovna Extended WPF Toolkit™ Community Edition (Xceed, 2014). Knihovna je postavena nad základními komponentami WPF a výrazně rozšiřuje jejich funkcionalitu.

K použití je nutné knihovnu importovat do projektu a před použitím definovat v některém nadřazeném elementu (nejlépe `Window` nebo `UserControl`) jmenný prostor

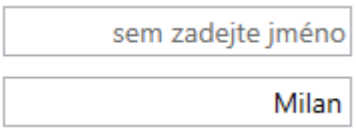
```
clr-namespace:Xceed.Wpf.Toolkit;assembly=Xceed.Wpf.Toolkit
```

s doporučovaným prefixem `xctk`. Komponenty knihovny se poté používají stejně jako ty standardní, jenom se musí opatřit tímto prefixem. V dalším textu jsou popsány komponenty použité v této práci.

WatermarkTextBox

Tato komponenta vylepšuje základní `TextBox` o možnost zobrazení vodoznaku. Díky němu můžeme uživateli napovědět, co má do pole zadat. Jakmile uživatel do pole něco napíše, vodoznak zmizí. Definice textu vodoznaku se provádí atributem `Watermark`. Dále uvádíme příklad definice komponenty `WatermarkTextBox` spolu se způsobem vykreslení na obrázku 4.2.

```
<xctk:WatermarkTextBox
 Watermark="sem zadejte jméno"
/>
```



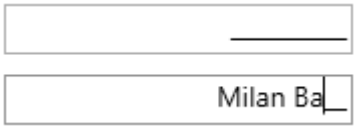
Obrázek 4.2: Stavy komponenty `WatermarkTextBox`

MaskedTextBox

Komponenta `MaskedTextBox` je opět rozšířením `TextBox` a slouží k limitaci uživatelského vstupu dle specifikované masky (atribut `Mask`). Maska se tvoří pomocí speciálních znaků, které jsou uvedeny v dokumentaci.

V následujícím příkladu je ukázka komponenty `MaskedTextBox` s maskou složenou z deseti znaků „C“. To znamená, že do pole musíme zadat deset libovolných znaků. Atribut `ResetOnSpace` nám umožňuje zadat i znak mezery. Chování komponenty je vyobrazeno na obrázku 4.3.

```
<xctk:MaskedTextBox
   Mask="CCCCCCCCCC"
   ResetOnSpace="False" />
```



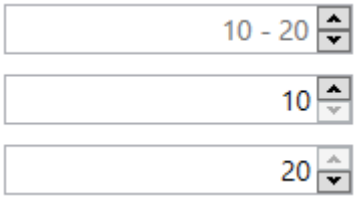
Obrázek 4.3: Stavy komponenty `MaskedTextBox`

DecimalUpDown

`DecimalUpDown` je komponenta pro kontrolované zadávání celých i desetinných čísel. Lze jí definovat minimální (`Minimum`) a maximální (`Maximum`) přijímanou hodnotu. Formátovacím řetězcem (`FormatString`) s prefixem „F“ navíc můžeme určit požadovaný počet desetinných míst čísla. Je možné také zadat vodoznak s nápovědou, například možným rozsahem čísel.

Příklad zobrazuje komponentu `DecimalUpDown` pro zadávání celého čísla (řetězec „F0“ definuje 0 desetinných míst) v rozsahu 10 až 20. Vzhled komponenty uvádíme na obrázku 4.4. Na obrázku je vidět, že hodnotu lze vybrat i pomocí spinneru (šipky nahoru/dolů).

```
<xctk:DecimalUpDown
   Watermark="10 - 20"
   Minimum="10"  Maximum="20"
   FormatString="F0" />
```



Obrázek 4.4: Stavy komponenty `DecimalUpDown`

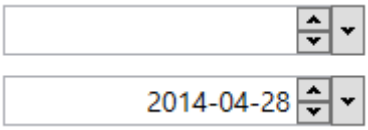
DateTimePicker

Poslední použitou komponentou z této knihovny je `DateTimePicker`. Jedná se o velice sofistikovanou komponentu pro zadávání dat a časů. Nejdůležitější

vlastností je možnost definice vlastního formátu, ve kterém hodnotu potřebujeme. Na vlastní formát komponentu přepneme pomocí `Format="Custom"`, ten pak zadáme jako formátovací řetězec atributem `FormatString`.

Následující příklad zobrazuje použití komponenty pro zadání data v mezinárodním formátu. První možností výběru hodnoty je použití spinneru, což je naznačeno na obrázku 4.5. Druhou je výběr z nabídky, která se zobrazí po kliknutí na tlačítko zcela vpravo (obr. 4.6).

```
<xctk:DateTimePicker
   Format="Custom" FormatString="yyyy-MM-dd" />
```



Obrázek 4.5: Stavy komponenty `DateTimePicker`

Obrázek 4.6: Výběr data pomocí `DateTimePicker`

4.7.2 WPF About Box

Starší technologie WinForms poskytuje vykreslení klasického „about“ boxu s využitím údajů, které jsou součástí konfigurace projektu (assembly information). Takovou funkcionalitu bohužel WPF již neposkytuje. Proto byla použita knihovna WPF About Box (Gattnar, 2012), která to umožňuje.

5 Analýza potřeb zadavatele

Zadavatelem práce je významný průmyslový partner Katedry informatiky a výpočetní techniky Západočeské univerzity v Plzni, který je dodavatelem a výrobcem zabezpečovací, telekomunikační, informační a automatizační techniky, zejména se zaměřením na oblast kolejové a silniční dopravy včetně telematiky a dalších technologií.

Pro konfiguraci zabezpečovací techniky využívá konfigurační soubory ve formátu XML, které mohou být validovány podle existujících XSD schémat. Příprava konfiguračních souborů je v současnosti realizována aplikací s pevně daným grafickým uživatelským rozhraním s názvem `JazzConf`.

Problémem současného řešení je, že se struktura jednotlivých konfiguračních souborů časem mění (a potažmo i XSD souborů), takže výstup z GUI aplikace je nutné dodatečně ručně upravovat. Současná aplikace zároveň neposkytuje možnosti tvorby konfiguračních souborů složitějších zařízení, které vznikají specifickou syntézou dílčích konfigurací, což je také nutné dělat ručně.

Posledním problémem je, že z povahy bezpečnostně kritických aplikací musejí být všechny XML konfigurační soubory (a jim příslušné XSD soubory) archivovány. Možnosti budoucí změny těchto souborů jsou velmi omezené, prakticky je možná pouze ruční úprava.

5.1 Požadavky zadavatele

Zadavatel požaduje vytvořit jádro desktopové aplikace s grafickým uživatelským rozhraním pro operační systém Windows XP a vyšší, jejíž ovládací prvky budou tvořeny a ovládány podle XSD schématu. To například znamená, že bude-li v XSD definován prvek s výběrem z možností, bude v GUI existovat výběrový seznam (`ComboBox`) se stejnou nabídkou možností. Nebo obsahuje-li XSD prvek s obecným textovým obsahem, v GUI bude `TextBox`. Jinak řečeno, pro odlišné prvky XSD bude mít GUI odlišné ovládací prvky.

Další funkcí jádra aplikace musí být načítání existujícího XML (dle příslušného XSD) do GUI, jeho editace a následné uložení. Samozřejmostí je automatická validace vytvořeného/upraveného XML podle schématu.

Jádro aplikace bude v praxi použito ve vícero klientských aplikacích pro tvorbu konfiguračních souborů. Naším úkolem je vytvořit dvě implementace klienta. První implementací by měla být jednoduchá aplikace, demonstrující správnou funkčnost jádra. Druhou poté reálná aplikace pro projektování konfigurací řídícího jádra železničních přejezdů.

Je zřejmé, že pro tuto úlohu neexistuje obecné řešení, ale je řešitelná za určitých omezujících podmínek kladených na parametrizující XSD soubor. Cílem je stanovit tyto podmínky a připravit funkční aplikace.

5.2 Současný stav

V následujícím textu stručně popíšeme současnou podobu systému, jež je předmětem této práce. Vycházíme při tom z informací uvedených v interní dokumentaci zadavatele.

5.2.1 Nástroj JazzConf

Nástroj `JazzConf` je univerzální nástroj pro tvorbu konfiguračních souborů. Aplikace má dvě uživatelská rozhraní. Variantu s grafickým uživatelským rozhraním WinForms `JazzConf.WindowsFormsGUI` a variantu s rozhraním příkazové řádky `JazzConf.ConsoleUI`.

Nás zajímá hlavně varianta s příkazovou řádkou, protože GUI budeme vytvářet vlastní, ale budeme potřebovat masivní aplikační logiku aplikace `JazzConf` pro dávkový převod konfiguračního XML do finální podoby, kterou je možné nahrát do cílového zařízení. Dávkový převod spustíme následujícím příkazem, kde `derivation_script.txt` je soubor s jednotlivými příkazy.

```
JazzConf.ConsoleUI.exe --batch "include derivation_script.txt"
```

5.2.2 Konfigurační soubory

Aplikace `JazzConf` operuje se třemi druhy souborů. Postupnými převody uvnitř aplikace se zdrojový konfigurační soubor dostává blíže reálnému zařízení, na kterém bude nasazen.

JAZZ

Při založení nového projektu konfigurace je vytvořen zdrojový XML soubor `jazz.xml` typu „JAZZ“. Ten obsahuje veškeré informace ve strukturované podobě a tento typ souboru by mělo vytvářet jádro naší aplikace. Schéma k tomuto druhu souboru však neexistuje, budeme jej proto muset vytvořit sami. Pro lepší představu uvádíme zkrácený příklad.

```
<JAZZ version="0.2">
 <System>
  <system type="Enum" label="2oo2">1</system>
  <platform type="Enum" label="Linux">1</platform>
 <System>
</JAZZ>
```

JAZZ/Cfg

Soubor typu „JAZZ“ se následně příkazem `load jazz.xml` načte a příkazem `derive JAZZ/Cfg` převede na interní reprezentaci XML souboru typu „JAZZ/Cfg“. Ten je svou strukturou naprosto odlišný od zdrojového souboru, nese však všechny důležité informace pro cílové zařízení.

Pouze k tomuto druhu souboru měl zadavatel zpracované nepříliš kvalitní XSD schéma. Soubor XML je však navržen s ohledem na cílové zařízení a napsat k němu kvalitní XSD je nemožné, protože skoro všechny elementy mají název `CFG` a liší se pouze hodnotou atributu `name`. Tento nedostatek v návrhu nám však nevadí, neboť veškerou manipulaci s tímto souborem budeme provádět přes aplikaci `JazzConf` příkazem `set` s parametrem podobným XPath výrazu. Schéma k tomuto souboru tak vlastně vůbec není potřeba. Následně uvádíme ukázku tohoto druhu souboru.

```
<CFG_ROOT>
 <CFG name="application">
  <CFG name="idleTime"><VALUE type="UI_32" data="2000" /></CFG>
 </CFG>
</CFG_ROOT>
```

JAZZ/CfgBin

Posledním krokem je převod „JAZZ/Cfg“ na binární „JAZZ/CfgBin“ příkazem `export cfg.bin JAZZ/CfgBin`. Výsledek je již možné nasadit na cílovém zařízení.

5.3 Výběr základních technologií

Před započetím návrhu a implementace bylo nutné vybrat některé základní technologie, na kterých práci postavíme. Zvolení použitých technologií vycházelo hlavně z požadavků zadavatele. Konfigurační soubory jsou ve formátu XML (viz kap. 2.1), k jejichž popisu využijeme XML schémata (viz kap. 2.3).

Jako schéma založené na gramatice bylo zadavatelem vybráno XSD, protože ho již v současnosti používá. To bude zároveň podkladem pro generování uživatelského rozhraní. S výběrem této varianty nemáme žádný problém a plně s ní souhlasíme. Pro návrh schémat bude použit návrhový vzor slepý Benátčan (Venetian Blind Design) z důvodů uvedených v kapitole 3.3.

Pro rozšíření možností popisu XML dokumentu použijeme schéma založené na pravidlech v jazyce Schematron. V konfiguračních souborech je řada pravidel, která je nutné dodržet. Využití jazyka Schematron je ideální volbou k popisu těchto pravidel, což přispěje k robustnosti aplikace. Navíc nám umožní definovat uživatelsky přívětivé chybové výpisy v případě selhání validace.

Výběr cílové technologie pro tvorbu GUI je popsán v kapitole 4. Zvolena byla technologie WPF, která umožňuje definovat uživatelské rozhraní pomocí XML (XAML). Technologie je součástí Microsoft .NET Framework a poslední verze, která podporuje Windows XP, je 4.0. Ta byla také použita.

Použitým programovacím jazykem je C#, protože je základním jazykem platformy .NET a neměli jsme žádný důvod k použití jiného jazyka, který tato platforma podporuje.

5.4 Existující řešení

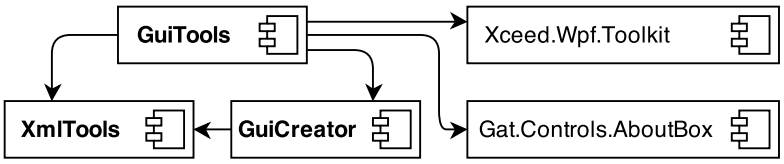
Jak bylo očekáváno, obecné řešení aplikace pro tvorbu GUI řízeného XSD schématem neexistuje, a pravděpodobně nikdy nebude. Je těžké si například jen představit uživatelské rozhraní formulářového typu vytvořené na základě XSD schématu pro XHTML. A to je konkrétní příklad, obecné řešení by muselo být mnohem složitější.

Při hledání existujících řešení je možné nalézt programy, které danou problematiku nějakým způsobem řeší, žádný však nesplňuje naše požadavky. Jako příklad uvedeme nástroje `XSD2GUI` (Batra, 2013) a `XSD-FORMS` (Moten, 2014). Jejich hlavním problémem je, že generují GUI webové, nikoli desktopové aplikace.

Nástroj `XSD2GUI` slouží ke generování webových formulářů dle XSD, pomocí technologií AJAX a DOM. Obsah schématu je nejprve nutné do programu nakopírovat, čímž dojde k vygenerování příslušného rozhraní, které můžeme následně používat. Použití nástroje je krajně nepraktické, pravděpodobně již není vyvíjen a obsahuje řadu chyb.

`XML-FORMS` generuje webový formulář na základě XSD schématu, které je však nutné doplnit o anotace řídící vzhled cílového rozhraní. Aplikace je aktivně vyvíjena a její použití je bezproblémové. Uživatelské rozhraní je zde také nutné předpřipravit nakopírováním XSD schématu do aplikace a výsledný XML soubor se zobrazuje přímo pod úspěšně odeslaným formulářem. Je proto nutné jej ručně zkopírovat a uložit. Řešení tak není bezproblémové, ale jeho vývoj se ubírá správným směrem.

6 Návrh a implementace jádra aplikace



Obrázek 6.1: Diagram modulů jádra aplikace

Jádro aplikace musí umožňovat tvorbu grafického uživatelského rozhraní ve formátu XAML na základě XSD schématu, jeho zobrazení, vkládání dat a uložení do XML dokumentu. Dále musí umožňovat do zobrazeného GUI načíst data z již hotového XML dokumentu, který je na základě příslušného XSD vytvořen. Pro zajištění robustnosti řešení je nutné při každém načítání a ukládání XML dokumentu provádět validaci.

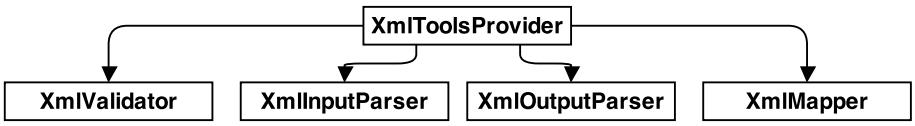
Při návrhu byl kladen důraz na maximální znovupoužitelnost jádra. Proto jsme jej navrhli a implementovali pomocí knihoven (class library), kde každá knihovna představuje modul s určitou funkcionalitou. Na diagramu 6.1 jsou vyobrazen vztahy mezi jednotlivými moduly, přičemž tučně jsou zobrazeny moduly námi vytvořené. Ostatní jsou rozšiřující knihovny třetích stran, kterými jsou:

Xceed.Wpf.Toolkit Knihovna Extended WPF Toolkit (viz kap. 4.7.1) rozšiřuje nabídku kontrolek, potřebných k realizaci.

Gat.Controls.AboutBox WPF About Box (viz kap. 4.7.2) je knihovna umožňující zobrazení „about“ boxu s využitím údajů v konfiguraci projektu (assembly infromation).

Aplikace, která bude jádro využívat, k němu bude přistupovat pouze pomocí modulu `GuiTools`, který prostřednictvím ostatních zajišťuje veškerou potřebnou funkcionalitu. V následujícím textu detailněji popíšeme funkcionalitu, návrh a implementaci jednotlivých modulů.

6.1 Modul XmlTools



Obrázek 6.2: Diagram tříd modulu `XmlTools`

Modul `XmlTools` slouží k načítání, ukládání a validaci XML souborů. Jedná se o rozhraní mezi reprezentací dat v GUI a reálnými XML dokumenty, uloženými na pevném disku počítače. Architektura modulu je popsána diagramem tříd 6.2.

Vnitřní reprezentací XML v tomto modulu je kolekce `Dictionary` (slovník, mapa), se záznamy XPath – Hodnota. Pro svázání hodnot uvedených v XML dokumentu s jejich ekvivalentem v GUI používáme podmnožinu jazyka XPath (viz kap. 2.4) pro jednoznačnou identifikaci jednoho uzlu. XPath v sobě nese veškeré informace o struktuře dokumentu, které nám později umožní zpětnou rekonstrukci záznamů do XML dokumentu. Reprezentace XML dokumentu slovníkem XPath – Hodnota je tak zcela bezztrátová.

Jednoznačný identifikátor komponenty XAML nastavujeme atributem `Name`. Jeho hodnotu bychom proto potřebovali nastavit na XPath výraz, který jednoznačně identifikuje prvek XML dokumentu, čímž by došlo ke svázání komponenty a prvku. Na hodnotu atributu jsou však aplikována stejná pravidla jako na identifikátor jazyka C# a XPath i XML obsahují znaky, které v identifikátoru nelze použít. Tyto znaky je nutné nějakým způsobem nahradit. Tím vznikne reprezentace dokumentu v podobě slovníku Name – Hodnota, pomocí kterého tento modul komunikuje s ostatními moduly. Pro identifikaci komponent, které obsahují námi sledované hodnoty, ještě přidáme hodnotě `Name` jako prefix kontrolní sekvenci, která to umožní.

Rozhraní modulu tvoří veřejná třída `XmlToolsProvider`, která obsahuje následující metody:

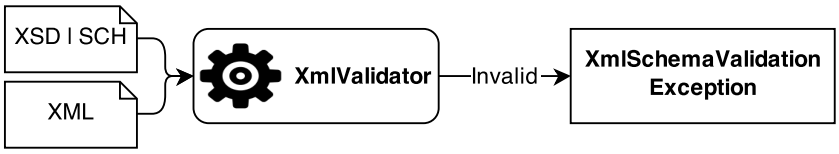
`LoadXml()` Tato metoda převede XML dokument z pevného disku do slovníku Name – Hodnota. Nejprve se pomocí třídy `XmlValidator` ověří, zda je dokument validní a má ho smysl dále zpracovávat. Poté jej s využitím třídy `XmlInputParser` převedeme na slovník XPath – Hodnota, který nakonec `XmlMapper` převede na slovník Name – Hodnota.

`SaveXml()` Metodu použijeme k uložení slovníku Name – Hodnota do XML dokumentu. Pomocí třídy `XmlMapper` převedeme slovník Name – Hodnota na slovník XPath – Hodnota. Ten `XmlOutputParser` převede na `XmlDocument` (DOM), který před uložením ověří `XmlValidator`.

Metody `MapSingleNameToXpath()` a `MapSingleXPathToName()` pouze volají stejnojmenné metody třídy `XmlMapper` pro převod jednoho XPath výrazu na Name a zpět. Stejně tak je tomu u metody `Validate()`, která volá validační metodu třídy `XmlValidator`.

V následujícím textu popíšeme třídy `XmlValidator`, `XmlInputParser`, `XmlOutputParser` a `XmlMapper` zajišťující funkcionalitu modulu.

6.1.1 XmlValidator



Obrázek 6.3: Princip funkcionality třídy `XmlValidator`

Třída `XmlValidator` umožňuje validaci XML dokumentu proti schématu jazyka XSD nebo Schematron. Validovat lze buďto XML soubor na disku, což potřebujeme při načítání dokumentu, nebo DOM dokument (`XmlDocument`), který chceme zpravidla na disk uložit. To je realizováno přetížením příslušných metod.

Jak je naznačeno na obrázku 6.3, validátor v případě neúspěchu vyhodí výjimku `XmlSchemaValidationException`, která obsahuje zprávu s popisem problému, kterou je možné zobrazit uživateli. Pokud je dokument validní, validátor neudělá nic a volající program může pokračovat.

Validace pomocí schématu XSD

Validaci proti schématu XSD zajišťuje metoda `Validate()`. Implementace využívá možností platformy .NET, která validaci pomocí XSD interně umožňuje, a to pomocí volání stejnojmenné metody nad některým z objektů

reprezentujících XML dokument. Pro reprezentaci dokumentu při načítání je použita třída `XDocument`, při ukládání pracujeme s dodaným objektem `XmlDocument`. Objektům je nutné před validací explicitně specifikovat XSD schéma.

Abychom mohli reagovat na výsledek validace, je nutné předat metodě `Validate()` obslužnou metodu, která se v případě neúspěchu zavolá. Té je mimo jiné předána výjimka `XmlSchemaValidationException` s výchozí zprávou, kterou .NET vygeneruje.

Obslužnou metodou pro načítání je `OpenValidationCallBack()`. Protože je charakter výjimky předem neznámý, použijeme přijatou výchozí výjimku, a v nezměněné podobě ji vyhodíme. `SaveValidationCallBack()` obsluhuje neúspěšnou validaci při ukládání dokumentu. Výjimky validátoru při ukládání jsou spojeny s chybným zadáním v GUI a budou zobrazovány uživateli častěji. Bylo proto potřeba jejich výchozí text upravit do uživatelsky přívětivější podoby, aby uživatel okamžitě pochopil, co je zadáno špatně.

Validace pomocí schématu Schematron

Platforma .NET jazyk Schematron nijak nepodporuje, ten je však navržen takovým způsobem, aby šel použít nezávisle na platformě pomocí XSLT transformací (viz kap. 2.5), které jsou všeobecně podporovány. Validace je implementována metodou `ValidateSchematron()`.

Výsledkem validace pomocí schématu Schematron je XML dokument SVRL. Ten získáme voláním metody `GetSchematronResult()`. Zde použijeme třídu `XslCompiledTransform` k vykonání potřebných transformací. Nejprve se sadou transformací schématu připraví validační transformační scénář. Ten aplikujeme na kontrolovaný XML dokument a výsledek, jímž je SVRL, vrátíme.

Z dokumentu SVRL následně zjišťujeme, zda byla validace úspěšná. Jazyk Schematron podporuje dva druhy zápisu podmínek, aserci (`assert`) a reporting (`report`). V případě neúspěchu aserce se v dokumentu objeví element `failed-assert` s chybovým textem specifikovaným ve schématu. Obdobně se při úspěšném reportingu v dokumentu objeví element `successful-report` s textem chyby. To znamená, že pokud SVRL obsahuje alespoň jeden z těchto elementů, je kontrolovaný dokument nevalidní.

V případě neúspěšné validace nakonec všechny chybové zprávy agregujeme do přehledného textového reportu, který použijeme jako zprávu výjimky `XmlSchemaValidationException`.

6.1.2 XmlInputParser

Obrázek 6.4: Princip funkcionality třídy `XmlInputParser`

Jedním z požadavků jádra aplikace je načítání již hotových XML dokumentů do GUI. K tomu napomáhá třída `XmlInputParser` vhodnou přípravou dat. Jak je naznačeno na obrázku 6.4, je jejím cílem transformovat specifikovaný XML dokument do kolekce `Dictionary` (slovník, mapa), jako záznamy XPath – Hodnota.

Pro převod dokumentu do slovníku jsme zvolili proudový „pull“ parser (viz kap. 2.2.1) `XmlReader`. Dokument nepotřebujeme nijak upravovat, pouze sekvenčně číst, k čemuž se proudové parsery hodí nejvíce. Konkrétní implementaci „pull“ parseru jsme zvolili z důvodu implicitní přítomnosti v .NET.

Algoritmus převodu

Převod zajišťuje metoda `Parse()`, ta provede inicializaci potřebných objektů a metodou `Run()` celý proces spustí. Parsování reaguje na tři druhy uzlů, otevírací značku elementu, textový uzel a uzavírací značku elementu. Ostatní uzly ignorujeme.

V objektu `StringBuilder` postupně sestavujeme výraz XPath aktuálně zpracovávaného prvku. Přidání nebo ubrání poslední části výrazu (za lomítkem) v závislosti na úrovni zanoření v XML dokumentu ovládáme metodami `NestDown()` a `NestUp()`.

Zpracování textového uzlu

Zpracování textového uzlu provádí metoda `ProcessCharacters()`. Jejím úkolem je uložení aktuálního XPath výrazu a hodnoty textového uzlu jako záznam do slovníku. Kontroluje se přitom, zda výraz již ve slovníku není, což se může stát v případě smíšeného obsahu elementu. Pokud tato situace nastane, přidáme aktuální hodnotu textového uzlu k hodnotě, která již je ve slovníku uložená.

Zpracování koncové značky elementu

Koncová značka elementu je signálem k redukci XPath výrazu o aktuální úroveň. Metoda `ProcessEndElement()`, která jej zpracovává, připraví část výrazu k odstranění a předá ji metodě `NestUp()`. Ta se postará o vlastní redukci.

Zpracování otevírací značky elementu

Nejsložitější je zpracování otevírací značky elementu, které zajišťuje metoda `ProcessStartElement()`. Pro stanovení korektního indexu elementu v XPath výrazu (uveden v hranatých závorkách) musí udržovat seznam elementů, které se vyskytují v dané úrovni, spolu s posledním použitým indexem. Pomocí zjištěného indexu a názvu elementu zkonstruuje výraz, který metodou `NestDown()` přidá k aktuálnímu výrazu.

Pokud element nemá hodnotu (textový uzel), musíme sami zavolat metodu `ProcessCharacters()`, aby se záznam o jeho existenci, i když bez hodnoty, zanesl do generovaného slovníku.

Dále zjišťujeme, zda se nejedná o otevírací značku prázdného elementu, který nemá značku uzavírací. V tom případě by se neprovedla redukce XPath výrazu. Jestliže tato situace nastane, musíme zavolat metodu `NestUp()`, která redukci provede.

Otevírací značka může navíc obsahovat atributy. Pokud je tomu tak, zavoláme metodu `ProcessAttributes()`. Tato metoda ve spolupráci s metodou `ProcessCharacters()` přidá do generovaného slovníku korektní záznamy pro všechny atributy, které otevírací značka obsahuje.

6.1.3 XmlOutputParser

Obrázek 6.5: Princip funkcionality třídy `XmlOutputParser`

Klíčovou funkcionalitou jádra aplikace je ukládání dat, která uživatel zadá v GUI, do dokumentu XML. Jak bylo řečeno v úvodu této podkapitoly, jsou pro svázání hodnot uvedených v GUI s jejich ekvivalentem v XML dokumentu použity výrazy jazyka XPath. Na obrázku 6.5 je znázorněn princip funkcionality třídy `XmlOutputParser`. Vstupem je slovník XPath – Hodnota, který je obrazem cílového dokumentu. Úkolem této třídy je převedení slovníku do podoby XML dokumentu.

Pro tvorbu dokumentu jsme zvolili technologii DOM (viz kap. 2.2.2), protože tvoříme zcela nový dokument, o kterém předem nic nevíme. Nelze tak použít proudový parser ani serializaci.

Algoritmus převodu

Převod probíhá v metodě `Parse()`, ta vytvoří nový `XmlDocument` a následně iteruje přes slovník XPath – Hodnota. Nad každým záznamem se zavolá rekurzivní metoda `ProcessXPath()`, která vrátí platný uzel `XmlNode`, korektně zařazený v dokumentu. V případě, že má uzel v záznamu specifikovanou hodnotu, nastavíme ji. Tím vznikne kompletní XML dokument, který je výstupem algoritmu.

Algoritmus plně podporuje práci se jmennými prostory, ačkoli je v této práci nevyužíváme. Byl však navržen pro všeobecné použití i pro případ budoucího rozšíření této práce. Ke správě jmenných prostorů dokumentu v algoritmu používáme .NET třídu `XmlNamespaceManager`.

Rekurzivní metoda ProcessXPath()

Rekurzivní metoda `ProcessXPath()` je naše vlastní implementace XPath processoru, jež dokáže zpracovat podmnožinu XPath, kterou používáme. Výraz se rozdělí do pole na jednotlivé části oddělené lomítkem, tedy jednotlivé úrovně v dokumentu. Každá úroveň rekurze poté zpracuje první prvek v poli, které se při zanoření o tento prvek zmenšuje.

Podmnožinu výrazu následně předzpracujeme v metodě `Init()`. Metoda nejprve podle znaku `@` na začátku zjistí, zda se jedná o atribut a pokud ano, znak odstraní. Jestliže jde o element, zjistí se jeho pozice uvedená v hranatých závorkách a ty jsou následně také odstraněny. Tím získáme kvalifikované jméno prvku. Jestliže kvalifikované jméno obsahuje prefix jmenného prostoru a ten ještě není ve správci jmenných prostorů dokumentu, vloží se do něj s dočasným URI, protože skutečné zatím neznáme.

V případě, že ještě není vytvořen kořenový element dokumentu, voláme metodu `CreateDocumentElement()`. Ta vytvoří nový dokument se specifikovaným kořenovým elementem, včetně případné deklarace jmenného prostoru, a vloží do dokumentu XML deklaraci. Pokud již dokument kořenový element má, voláme pro elementy metodu `ProcessElement()` a pro atributy `ProcessAttribute()`.

Metoda `ProcessElement()` postupně zjišťuje, zda v dokumentu existují všechny elementy daného jména až do dané pozice. Pokud neexistují, jsou vytvořeny a přidány do dokumentu. Tento postup byl zvolen z toho důvodu, že do dokumentu z principu nelze zařadit element na určitou pozici. Chceme-li například vložit element s indexem 2, musí již v dokumentu existovat element s indexem 1. Pokud neexistuje, musíme ho vytvořit dříve, než aktuálně vytvářený. URI jmenného prostoru elementu získáme metodou `GetNamespaceURI()` ze správce jmenných prostorů dokumentu.

Atributy zpracovává metoda `ProcessAttribute()`. Nejprve zjišťujeme, zdali rodičovský element stejnojmenný atribut již neobsahuje, protože dva atributy stejného jména element mít nemůže. Jestliže takový atribut neobsahuje, ověříme, zda atribut není deklarace jmenného prostoru (prefix `xmlns`). Pokud ano, zavoláme metodu `DefineNamespace()`, která se postará o založení jmenného prostoru do správce jmenných prostorů dokumentu a záměnu falešných URI za korektní. Nakonec atribut vytvoříme a vložíme jej do rodičovského elementu, opět s využitím metody `GetNamespaceURI()` pro zjištění URI jmenného prostoru atributu.

6.1.4 XmlMapper

Obrázek 6.6: Princip funkcionality třídy `XmlMapper`

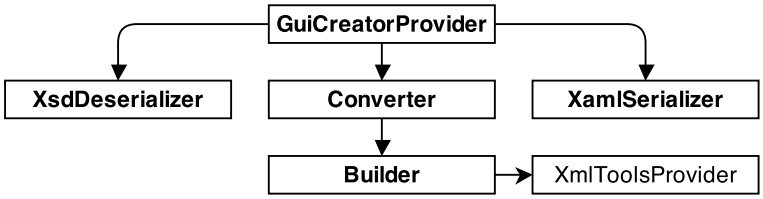
`XmlMapper` slouží k převodu unikátních XPath výrazů do formátu, který lze použít jako hodnota atributu `Name` v XAML a zpět. Atribut `Name` je totiž identifikátor, který nesmí obsahovat znaky `/, [, ], @, -, ., :`. Ty je proto nutné nějakým způsobem nahradit.

Nejvhodnější volbou zástupného znaku je takový, který lze použít jako identifikátor, ale nelze jej použít jako název elementu nebo atributu v XML. Takovým znakem může být například jakýkoli znak z Unicode kategorie Lm (Letter, Modifier). Vybrali jsme znak `U+02C9`, který vypadá jako pomlčka v horním indexu. Abychom nemuseli pro každý zakázaný znak hledat jiný zástupný znak, použijeme námi vybraný znak jako prefix čísla, které konkrétní zakázaný znak identifikuje. Vznikne tak množina mapovacích pravidel znak – zástupná dvojice, kterou jsme implementovali dvojrozměrným polem `REPLACE_RULES`.

Metoda `MapSingleXPathToName()` zajišťuje převod XPath na identifikátor. Ta postupně aplikuje nahrazení dle mapovacích pravidel a nakonec celému identifikátoru přidá jako prefix kontrolní sekvenci, pomocí které budeme moci v XAML identifikovat komponenty, které obsahují hodnoty. Opačný převod provádí metoda `MapSingleNameToXPath()`, která nejprve z identifikátoru kontrolní sekvenci odstraní a následně aplikuje mapovací pravidla.

Klíčovou funkcionalitou třídy `XmlMapper` je mapování celých slovníků, což naznačuje obrázek 6.6. `XmlInputParser` produkuje a `XmlOutputParser` konzumuje pouze slovníky ve formátu XPath – Hodnota. Naopak klient modulu `XmlTools` požaduje nebo poskytuje slovníky ve formátu Name – Hodnota. Proto je důležité mít mechanismus pro jejich převod jako celku. Převod zajišťují metody `MapXPathToNames()` a `MapNamesToXPath()`. Ty iterují přes celý slovník a nad jednotlivými záznamy volají dříve zmíněné mapovací metody.

6.2 Modul GuiCreator



Obrázek 6.7: Diagram tříd modulu `GuiCreator`

Modul `GuiCreator` je stěžejním prvkem jádra aplikace. Jeho úkolem je ze schématu XSD vytvořit uživatelské rozhraní XAML, které budeme moci později zobrazit. Architektura modulu je zobrazena na diagramu 6.7, kde je naznačena i spolupráce s modulem `XmlTools`.

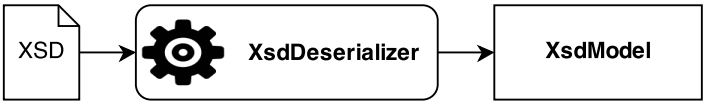
Pro parsování XSD schématu byla zvolena technologie mapování XML na objekty (viz kap. 2.2.3), neboli serializace. Struktura XSD je předem známá, proto je možné vytvořit obecný objektový model pro serializaci, který umožní jednodušší práci s dokumentem. My jsme však nepoužili automatický generátor objektového modelu `xsd.exe`, ale v souboru `XsdModel` jsme si vytvořili vlastní model. Tím jasně a přesně definujeme požadovanou podmnožinu a strukturu XSD (omezující podmínky), se kterou naše jádro aplikace dokáže pracovat.

Tvorbu XAML také realizujeme pomocí serializace. Předem jsme navrhli vzhled a strukturu všech komponent, proto bylo možné vytvořit objektový model XAML – `XamlModel`. Ten nám umožní sestavit dokument pomocí objektů, který následně serializujeme do souboru XAML.

Rozhraní modulu tvoří veřejná třída `GuiCreatorProvider`. Obsahuje jedinou metodu `CreateGui()`, která spustí proces tvorby XAML. Nejdříve je pomocí třídy `XsdDeserializer` z XSD vytvořen `XsdModel`, který předáme třídě `Converter`. Výsledkem je `XamlModel`, který `XamlSerializer` serializuje na pevný disk do souboru XAML.

V následujícím textu popisujeme výše zmíněné třídy `XsdDeserializer`, `XamlSerializer`, `Converter` a `Builder`.

6.2.1 XsdDeserializer



Obrázek 6.8: Princip funkcionality třídy `XsdDeserializer`

`XsdDeserializer` zajišťuje, jak naznačuje obrázek 6.8, převod (deserializaci) XSD schématu do objektového modelu `XsdModel`. Deserializaci automaticky zajišťuje .NET třída `XmlSerializer`. Před vlastní deserializací nejprve zkontrolujeme, zda je načítaný dokument validním XSD schématem. Tato kontrola je prováděna vnitřními nástroji .NET.

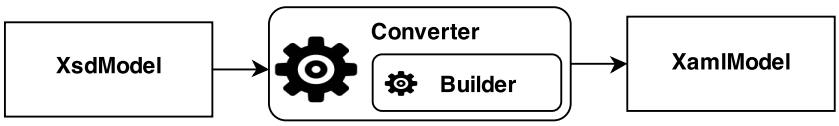
Pokud je dokument v pořádku, přistoupíme k vlastnímu převodu. Objektu `XmlSerializer` nastavíme metody pro reakci na události výskytu neznámého atributu (`UnknownAttribute`) a elementu (`UnknownElement`). Neznámý element nebo atribut je ten, který se v dokumentu objeví v rozporu s objektovým modelem `XsdModel`. Tento mechanismus je výborným nástrojem ke kontrole omezujících podmínek XSD dokumentu. Dokument musí vypadat přesně tak, jak jsme ho navrhli v objektovém modelu, a pokud tomu tak není, zavolá se obsluha neznámého prvku. V metodě pro obsluhu připravíme uživatelsky srozumitelnou chybovou hlášku co je v dokumentu špatně, a vyhodíme ji pomocí výjimky `SchemaDeserializeException`.

6.2.2 XamlSerializer

Obrázek 6.9: Princip funkcionality třídy `XamlSerializer`

Třída `XamlSerializer` provádí serializaci objektového modelu `XamlModel` do souboru XAML. Její funkcionalita je znázorněna na obrázku 6.9. K serializaci je opět použita třída `XmlSerializer` bez žádných speciálních opatření, očekáváme totiž, že je objektový model sestavený třídou `Converter` správný.

6.2.3 Converter



Obrázek 6.10: Princip funkcionality tříd `Converter` a `Builder`

Jak naznačuje obrázek 6.10, slouží třída `Converter` k převodu objektového modelu XSD schématu `XsdModel` na objektový model XAML uživatelského rozhraní – `XamlModel`. Během převodu dává třída `Converter` příkazy třídě `Builder`, která je zodpovědná za sestavení objektového modelu `XamlModel`.

Zpracování deklarací elementů (`xs:element`) zajišťujeme pomocí metody `ProcessElement()`. Deklarace atributů (`xs:attribute`) zpracovává metoda `ProcessAttribute()`. Metody jsou vždy volány se dvěma parametry. První je objekt deklarace prvku, který zpracováváme, druhým je XPath výraz identifikující rodiče. Tento výraz musíme konstruovat proto, abychom později mohli korektně nastavit atribut `Name` cílové komponenty GUI. Při rekurzivním volání tak vždy přidáme výraz pro aktuální prvek.

Převod spustíme zavoláním metody `Convert()`. Ta z modelu `XsdModel` získá deklaraci kořenového elementu a předá ji metodě `ProcessElement()`. Metoda následně zpracuje veškerý obsah kořenového elementu a po jejím doběhnutí si `Converter` vyžádá od třídy `Builder` hotový `XamlModel`, který metoda `Convert()` vrátí.

ProcessElement()

Před vlastním zpracováním deklarace elementu provedeme sérii dodatečných kontrol omezujících podmínek XSD. Kontrolujeme, zda má element korektně nastaven typ (`type`). V případě, že je deklarován minimální počet výskytů (`minOccurs`), není nastaven na hodnotu „0“. A v případě, že má deklarován maximální počet výskytů (`maxOccurs`), nemá hodnotu „unbounded“. Na chybné stavy reagujeme vyhozením výjimky se srozumitelným hlášením.

Následně zjistíme, kolik elementů máme podle deklarace vygenerovat. Pokud se minimální počet výskytů rovná maximálnímu, budeme generovat minimální počet. Pokud je však maximální počet výskytů větší, musíme se

uživatele voláním metody `GetElementOccurs()` zeptat, kolik jich má být. Metoda zobrazí dialog, do kterého uživatel potřebný počet zadá.

Pro každý element poté voláme následující sérii metod. Protože předem nevíme, co bude obsahovat, oznámíme třídě `Builder`, že má v GUI vytvořit nový podstrom, který bude tento element reprezentovat, voláním metody `CreateSubtree()`. Následně zavoláme metodu `ParseType()`, které předáme typ, jméno a aktuální výraz XPath. Tato metoda na základě typu vytvoří obsah elementu. Nakonec oznámíme třídě `Builder`, že má podstrom aktuálního elementu uzavřít voláním metody `CloseSubtree()`.

ProcessAttribute()

Zpracování deklarací atributů je podobné zpracování deklarací elementů. Nejprve zkontrolujeme, zda má deklarace nastaven typ, a zda má nastaven atribut `use` na hodnotu „required“. Na chybné stavy reagujeme vyhozením výjimky. Poté přepneme globální přepínač `processingAttribute` na „true“, abychom věděli, že zpracováváme atribut, a zavoláme metodu `ParseType()`. Po doběhnutí metody přepneme globální přepínač zpět na „false“.

ParseType()

Metoda `ParseType()` na základě datového typu nepřímo vytvoří obsah elementu nebo atributu. Datovým typem může být některý z povolených vestavěných datových typů nebo uživatelsky definovaný jednoduchý či komplexní datový typ.

Pro daný datový typ se nejprve snažíme nalézt příslušnou deklaraci mezi komplexními (`xs:complexType`) a jednoduchými (`xs:simpleType`) typy. Pokud ji nenalezneme, musí se jednat o vestavěný datový typ. Každý druh deklarace zpracovává jiná metoda. Komplexní typy řeší `ParseComplexType()`, jednoduché `ParseSimpleType()` a vestavěné `ParsePrimitiveType()`.

ParseComplexType()

Komplexním typem lze definovat tři druhy elementů. Prázdný element, element s atributy a hodnotou nebo element s atributy a podřízenými elementy.

Pokud se jedná o deklaraci prázdného elementu, žádné další informace nepotřebujeme a můžeme třídě `Builder` metodou `GenerateEmptyElement()` přikázat vytvoření prázdného elementu reprezentovaného skrytou komponentou `WatermarkTextBox`.

Deklaraci komplexního typu elementu s atributy a hodnotou řeší metoda `ParseSimpleContentExtension()`. Ta pro všechny deklarace atributů zavolá metodu `ProcessAttribute()` a pro vygenerování obsahu metodu `ParseType()`, které předáme specifikovaný jednoduchý typ.

Pro deklaraci elementu s atributy a podřízenými elementy nejprve pro všechny deklarace atributů voláme metodu `ProcessAttribute()`, protože chceme aby v uživatelském rozhraní byly atributy uvedeny před zanořenými elementy. Poté pro všechny deklarace elementů v kompozitoru `xs:sequence` voláme metodu `ProcessElement()`. Kontrolujeme přitom, zda není element stejného jména deklarován dvakrát, což nepovolujeme a vyhodíme výjimku. Deklarace podřízených elementů je možná jen kompozitorem `xs:sequence`, protože pro zobrazení v GUI potřebujeme přesně znát pořadí elementů.

ParseSimpleType()

Při zpracování jednoduchého typu nás zajímá základní vestavěný datový typ a jeho restrikce. Tyto informace z deklarace typu extrahujeme a předáme je metodě `ParseType()`, které v tomto případě navíc předáme i seznam restrikcí.

ParsePrimitiveType()

Při zpracování vestavěných datových typů již máme dost informací k tomu, abychom mohli třídě `Builder` předat instrukce ke generování komponent uživatelského rozhraní.

Vestavěné datové typy jsme rozdělili do pěti kategorií a jejich názvy fyzicky umístili do pěti řetězcových polí. Zakázané typy (`forbiddenTypes`), nepodporované typy (`unsupportedTypes`), číselné typy (`numericTypes`), řetězcové typy (`stringTypes`) a typy dat a časů (`dateTypes`). Specifický typ `boolean` není zařazen do žádné kategorie, protože pro něj nelze definovat integritní omezení. Rozdělení ostatních kategorií podporovaných typů je založeno právě na možnostech definování integritních omezení.

Zakázanými typy jsou `anyType` a `anySimpleType`, které umožňují, aby prvek mohl mít obsah jakéhokoli typu nebo jakéhokoli jednoduchého typu deklarovaného ve schématu. Tuto možnost nepovolujeme, protože potřebujeme předem přesně znát strukturu GUI. V případě výskytu těchto typů proto vyhazujeme výjimku.

Množina nepodporovaných typů obsahuje exotické typy, které se příliš nedoporučují používat. Jejich použití však nebráníme, pouze uživateli zobrazíme varování, že používá nepodporovaný datový typ. Třídě `Builder` poté metodou `GenerateGenericBox()` přikážeme vygenerovat komponentu `WatermarkTextBox`, která bude obsahovat nápovědu s názvem požadovaného datového typu.

Komponentu `CheckBox` pro typ `boolean` přikážeme třídě `Builder` vytvořit voláním metody `GenerateBooleanBox()`.

Ze všeho nejdříve se podíváme, jestli nemá datový typ definovánu restrikci seznamem přípustných hodnot (`xs:enumeration`). V takovém případě nás nemusí zajímat datový typ a voláme metodu `ProcessEnumerationValue()`. Ta z restrikcí získá seznam hodnot a voláním `GenerateEnumerationBox()` třídy `Builder` přikážeme vytvořit komponentu `ComboBox`, která bude hodnoty obsahovat. Pokud není definována restrikce seznamem, voláme jednu z následujících metod podle toho, do které skupiny datový typ patří.

`ProcessNumericValue()` Zpracování číselných datových typů. Metoda dle datového typu nastaví minimální a maximální přípustnou hodnotu a formátovací řetězec čísla. Tyto hodnoty následně upravíme o případná integritní omezení a metodou `GenerateNumericBox()` třídy `Builder` jí přikážeme vytvořit komponentu `DecimalUpDown` pro zadávání čísel.

`ProcessStringValue()` Zpracování řetězcových typů. Pomocí integritních omezení nastavíme minimální a maximální délku řetězce, případně regulární výraz. Tyto informace zašleme metodou `GenerateStringBox()` třídě `Builder`, která vytvoří komponentu `MaskedTextBox` (při omezení `xs:length`) nebo `WatermarkTextBox` (se zobrazenými požadavky na délku nebo regulárním výrazem) pro zadávání řetězců.

`ProcessDateValue()` Zpracování typů dat a časů. Na základě konkrétního typu vybereme z předpřipraveného pole korektní formátovací řetězec a metodou `GenerateDateBox()` přikážeme třídě `Builder` vytvořit komponentu `DateTimePicker` pro zadávání dat a časů zvoleného formátu.

6.2.4 Builder

Třída `Builder` slouží ke tvorbě objektového modelu XAML dokumentu. Tvorba je přitom řízena nadřízenou třídou `Converter`, která voláním metod této třídy zajišťuje tvorbu potřebných komponent.

V konstruktoru vytváříme novou instanci třídy `XamlModel`, ta reprezentuje kořenový element cílového dokumentu XAML – `UserControl`. Dokument bude obsahovat uživatelsky definovanou komponentu, kterou bude možné načíst do cílového uživatelského rozhraní.

Základní komponentou rozhraní je `Expander` reprezentující element, který obsahuje podřízené prvky. Uvnitř komponenty `Expander` bude vždy umístěn třísloupcový kontejner rozložení `Grid`. Do prvního sloupce budeme umísťovat ikonu (`Image`). Modrou s písmenem E, pokud je zadávaná hodnota obsahem elementu, nebo červenou s písmenem A, pokud je hodnota obsahem atributu. Druhý sloupec bude obsahovat štítek (`Label`) se jménem prvku a třetí komponentu pro vlastní zadání hodnoty.

V případě, že má podřízený element další potomky, bude jeho komponenta `Expander` zabírat všechny tři sloupce layoutu.

Následně uvádíme popis metod, které volá třída `Converter` pro generování komponent uživatelského rozhraní.

`CreateSubtree()` Pokyn k vytvoření nové úrovně zanoření – komponenty `Expander`. Vytvoření komponenty provádí metoda `CreateExpander()`, která objekt vytvoří, nastaví potřebné parametry a vloží do něj kontejner rozložení `Grid`, jenž vytvoří metodou `CreateGrid()`.

Vytvořená komponenta `Expander` je vložen do dokumentu a do seznamu zanoření. Ten udržujeme, abychom mohli získat referenci na předka při vynoření.

`CloseSubtree()` Uzavření podstromu – vynoření. Obsah aktuální komponenty `Expander` je kompletní, proto ji vyjmeme ze seznamu zanoření. Pokud navíc obsahuje pouze jeden řádek pro podřízený element, vyměníme v rodičovské komponentě tento řádek za aktuální `Expander`. Tím zabráníme tomu, aby měl každý element vlastní `Expander`, což by vedlo ke značnému znepřehlednění a snížení ergonomie GUI.

`Generate...()` Všechny metody s prefixem `Generate` vytvoří a nastaví specifickou komponentu, která je vhodná pro zadávání určitého typu dat. O jaké komponenty se jedná, je uvedeno v popisu třídy `Converter` u volání metod s tímto prefixem.

Po vytvoření komponenty požádáme třídu `XmlToolsProvider` o převedení XPath výrazu na identifikátor a komponentě jej jako atribut `Name` nastavíme. Nakonec zavoláme metodu `PrepareGridRow()`, které předáme komponentu, jméno a informaci zda se jedná o atribut.

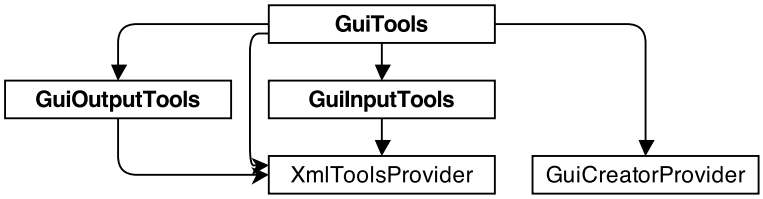
Tato metoda postupně s pomocí `CreateIcon()` a `CreateLabel()` vytvoří řádek layoutu `Grid`, do kterého jej posléze prostřednictvím metody `AppendGridRow()` vloží.

6.2.5 Omezující podmínky XSD souboru

Z návrhu a implementace tohoto modulu vyplývají omezující podmínky, které klademe na parametrizující XSD schéma. Omezení struktury dokumentu je definováno a zdokumentováno objektovým modelem `XsdModel`, další podmínky stanovuje zpracování třídou `Converter`. Následně uvádíme nejzásadnější z nich.

- XSD musí být navrženo dle návrhového vzoru slepý Benátčan (Venetian Blind Design),
- deklarace obsahu jednoduchých datových typů používají jen restrikce (`xs:restriction`), přičemž základem (`base`) je vestavěný datový typ,
- deklarace obsahu komplexních datových typů mohou obsahovat pouze kompozitor `xs:sequence`,
- hodnota atributů `minOccurs` nesmí být „0“,
- hodnota atributů `maxOccurs` nesmí být „unbounded“,
- deklarace atributů musí mít nastaven atribut `use` na „required“,
- nikde nelze použít vestavěné datové typy `anyType` a `anySimpleType`,
- nelze deklarovat jmenné prostory.

6.3 Modul GuiHostTools



Obrázek 6.11: Diagram tříd modulu `GuiHostTools`

`GuiHostTools` je hlavním modulem jádra aplikace. Jeho prostřednictvím klientské aplikace využívají veškeré funkcionality jádra. Architektura modulu je naznačena na diagramu 6.11, kde je vidět, že modul při své práci spolupracuje se všemi ostatními moduly. Modul samotný se stará o zobrazení a ovládání předem vytvořeného uživatelského rozhraní.

Rozhraním modulu a zároveň jádra je veřejná třída `GuiTools`. Ta při instanciaci provede inicializaci jádra a poté umožní pomocí ovládacích metod jádro ovládat.

Inicializace jádra

Konstruktoru třídy `GuiTools` musíme předat referenci na okno (`Window`), do kterého budeme načítat vytvořený XAML. Rodičovské okno musí obsahovat komponentu `ScrollViewer` se jménem `contentDock`. Do té se umístí modulem `GuiCreator` vytvořený `UserControl`, uložený v XAML.

Volitelně je možné v konstruktoru specifikovat cestu ke schématu XSD, schématu v jazyce Schematron a cílové umístění výsledného XML souboru. Pokud konstruktoru nepředáme cestu ke schématu XSD, zavoláme metodu `FileChooser()`, která zajistí, aby uživatel korektní XSD vybral, bez něj totiž nemůže program pokračovat. Pokud uživatel žádné schéma nevybere, program končí.

Následně automaticky kontrolujeme, jestli ve stejném umístění jako XSD schéma neexistuje stejnojmenné schéma v jazyce Schematron. Pokud ano, načteme jej, v opačném případě se nic neděje, validace oproti Schematronu neproběhnou.

Poté schéma XSD předáme modulu `GuiCreator`, který na jeho základě vytvoří GUI, uloží jej na disk a předá nám jeho adresu. Nakonec konstruktor volá metodu `Init()`.

Metoda `Init()` pomocí třídy `XamlReader` načte z XAML komponentu `UserControl`. Poté se připravená komponenta nastaví jako obsah komponenty `contentDock` rodičovského okna.

Nakonec se vytvoří instance tříd `GuiInputTools` a `GuiOutputTools`, zajišťujících načítání a ukládání dat. Těm předáme komponentu `UserControl` jako kořenový element podstromu, nad kterým budou operovat a instanci `XmlToolsProvider`, aby mohli provádět operace načítání a ukládání XML.

Ovládací metody

Třída `GuiTools` dále obsahuje ovládací metody, které může rodičovská aplikace používat k ovládání jádra. Názvy metod obvykle odpovídají položkám menu aplikace.

`New()` Opětovně inicializuje uživatelské rozhraní. Nejdříve požádá modul `GuiCreator` o vytvoření nového XAML pro případ, že by se změnilo XSD schéma, a následně znovu zavolá metodu `Init()`.

`Open()` Načtení XML dokumentu do uživatelského rozhraní. Uživatel prostřednictvím volání metody `FileChooser()` zvolí dokument, který se pomocí `GuiInputTools` načte do GUI.

`Save()`, `SaveAs()` a `SaveAndClose()` Uložení dat z GUI do XML dokumentu. Pokud ještě ukládání neproběhlo, je uživatel požádán o výběr cílového souboru voláním metody `FileChooser()`. `SaveAs()` výběr cílového souboru vynucuje. `SaveAndClose()` navíc po uložení dokumentu provede zavření rodičovského okna. Uložení dokumentu zprostředkovává třída `GuiInputTools`.

`Close()` a `CloseGui()` Zavře rodičovské okno a ukončí program. Varianta `CloseGui()` zavře pouze část GUI spravovanou jádrem, odstraněním obsahu komponenty `contentDock`.

`About()` Zobrazí „about“ box s využitím údajů, které jsou součástí konfigurace projektu (assembly information). K tomu využívá knihovnu WPF About Box.

Metoda Execute()

Pokud chceme využívat jednotné zpracování a zobrazení výjimek s ukládáním záznamů do žurnálu, nesmíme v klientské aplikaci volat výše zmíněné ovládací metody přímo, ale nepřímo prostřednictvím statické metody `Execute()`. Té předáme delegáta volané metody. Pokud bychom měli v klientské aplikaci instanci třídy `GuiTools` pojmenovanou `guiTools`, vypadal by požadavek na uložení dat následovně:

```
GuiTools.Execute(guiTools.Save);
```

Metoda `Execute()` interně zavolá specifikovanou metodu (zde `Save()`) a v případě, že metoda vyhodí výjimku, zavolá `ProcessException()`. Zde zjistíme o jaký typ výjimky se jedná. Pokud jde o validační výjimku, zapíšeme ji do žurnálu `validation.log`. Ostatní výjimky zapíšeme do chybového žurnálu `error.log`. Uživatele o vzniklém problému informujeme adekvátním hlášením v podobě okna `MessageBox`. Validační výjimky jako varování, ostatní jako chyby.

6.3.1 GuiInputTools

Obrázek 6.12: Princip funkcionality třídy `GuiInputTools`

Pro naplnění zobrazeného uživatelského rozhraní daty z načteného XML souboru slouží třída `GuiInputTools`. Princip je naznačen na obrázku 6.12. Vlastní zpracování zajišťuje metoda `LoadIntoGui()`.

Nejprve s využitím modulu `XmlTools` získáme z požadovaného vstupního XML dokumentu slovník Name – Hodnota. Poté postupně procházíme všechny záznamy ve slovníku a pro každý záznam nalezneme v uživatelském rozhraní komponentu s příslušným identifikátorem. Podle druhu komponenty vybereme metodu, která jí nastaví hodnotu.

Každá komponenta totiž vyžaduje odlišné nastavení hodnoty. Komponentám typu `TextBox` nastavíme hodnotu atributem `Text`. `DecimalUpDown` a `DateTimePicker` se nastavují pomocí atributu `Value`. Vybraná položka v komponentě `ComboBox` je nastavena atributem `SelectedIndex`. U ní je proto nutné mezi nabízenými položkami nalézt index námi požadované položky, a ten jako tento atribut nastavit. Zatržení komponenty `CheckBox` reprezentuje atribut `IsChecked`. Pokud je hodnota v XML dokumentu „true“ nebo „1“, nastavíme `IsChecked` na „true“, jinak na „false“.

6.3.2 GuiOutputTools

Obrázek 6.13: Princip funkcionality třídy `GuiOutputTools`

Třída `GuiOutputTools` slouží k extrakci dat z námi sledovaných komponent uživatelského rozhraní do slovníku Name – Hodnota. Tento slovník je následně předán modulu `XmlTools` k převodu do XML. Funkcionalita je naznačena na obrázku 6.13. Zpracování zajišťuje metoda `SaveIntoXml()` nebo `SaveIntoDom()`. Ty založí nový slovník a zavolají rekurzivní metodu `EnumerateControls()`, která jej naplní daty. Nakonec je slovník předán modulu `XmlTools`.

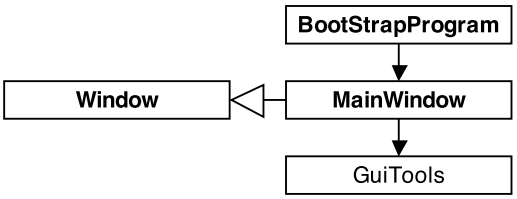
Rekurzivní metoda `EnumerateControls()` prochází metodou pre-order strom komponent uživatelského rozhraní pomocí třídy `VisualTreeHelper`. Jestliže je aktuálně nalezená komponenta kontrolkou, ověříme podle kontrolní sekvence, která je prefixem identifikátoru, zda se jedná o kontrolku obsahující hodnotu. Pokud ano, získáme ji pomocí metody `GetValue()` a identifikátor s hodnotou uložíme do slovníku. Nad aktuální komponentou následně rekurzivně zavoláme metodu `EnumerateControls()`.

Metoda `GetValue()` získá na základě typu komponenty její hodnotu. U komponent typu `TextBox`, `DecimalUpDown`, `DateTimePicker` a `ComboBox` je hodnota obsažena v atributu `Text`. Hodnota komponenty `CheckBox` je uložena v atributu `IsChecked`.

7 Implementace klientských aplikací

Tato kapitola pojednává o implementaci klientských aplikací jádra. První implementací je jednoduchá aplikace `HostApplication`, pomocí které otestujeme jeho správnou funkčnost. Druhou je aplikace pro projektování řídícího jádra železničních přejezdů `Configurator`, kde produkt otestujeme na reálných datech. V závěru kapitoly jej srovnáme se současným řešením.

7.1 HostApplication



Obrázek 7.1: Diagram tříd aplikace `HostApplication`

Aplikace `HostApplication` slouží k předvedení a otestování vytvořeného aplikačního jádra. Sama nepřidává žádnou funkcionalitu, všechny operace jsou prováděny voláním metod jádra. Součástí této kapitoly je příloha B, která obsahuje obrázky aplikace pořízené v rámci testování.

Diagram 7.1 zobrazuje architekturu aplikace. Metoda `Main()`, umístěná ve třídě `BootStrapProgram`, vytvoří při spuštění instanci třídy `MainWindow`, která je potomkem třídy `Window`, a předá jí řízení.

`MainWindow` je ovládací třída kompilovaného XAML s následující strukturou (zjednodušeně):

```
<Window x:Class="HostApplication.MainWindow">
  <DockPanel>
    <Menu DockPanel.Dock="Top"> zde položky menu </Menu>
    <ScrollViewer Name="contentDock" DockPanel.Dock="Bottom" />
  </DockPanel>
</Window>
```

Ovládací třída inicializuje komponenty, vytvoří instanci třídy `GuiTools` a nakonec okno vykreslí. Obsluha položek menu je přímo navázána na stejnojmenné metody třídy `GuiTools` prostřednictvím metody `Execute()`. Položkami menu jsou `New`, `Open`, `Save`, `SaveAs`, `SaveAndClose`, `Close` a `About`.

Aplikace má následující adresářovou strukturu:

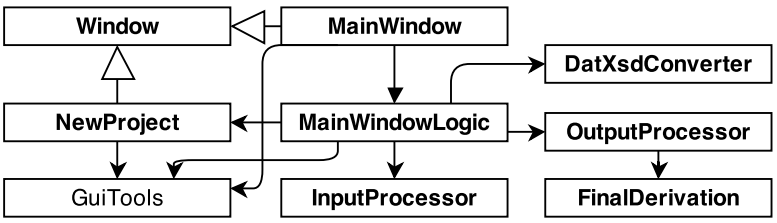
```
HostApplication/
├── schematron/ transformační scénáře pro validaci
├── xaml/ vygenerovaná uživatelská rozhraní schémat
│   ├── restriction_test.xaml
│   └── structure_test.xaml
├── xml/ zde budou uloženy výstupní XML soubory
├── xsd/ připravená schémata, mezi kterými si budeme vybírat
│   ├── restriction_test.xsd
│   ├── structure_test.sch
│   └── structure_test.xsd
├── Gat.Controls.AboutBox.dll
├── GuiCreator.dll
├── GuiHostTools.dll
├── HostApplication.exe spouštěč aplikace
├── Xceed.Wpf.Toolkit.dll
└── XmlTools.dll
```

XSD schémata, která chceme použít pro generování GUI, musíme předem umístit do složky `xsd`. Při spuštění aplikace vybereme požadované schéma, do složky `xaml` se pro něj vygeneruje uživatelské rozhraní, a zobrazí se. Složka `xml` slouží k ukládání a načítání vytvořených XML dokumentů.

K otestování korektní výstavby uživatelského rozhraní dle struktury dokumentu a druhů elementů jsme vytvořili schéma `structure_test.xsd`. To obsahuje kořenový element s potomky, koncový element s atributy, prázdný element s atributy a prázdný element. Vytvořené uživatelské rozhraní je zobrazeno na obrázku B.1. Pro ověření korektní funkcionality validace pomocí jazyka Schematron jsme vytvořili schéma `structure_test.sch`. Hlášení neúspěšné validace zobrazuje obrázek B.2.

Test, zda jádro generuje korektní komponenty pro různé vestavěné datové typy a správně interpretuje jejich integritní omezení, jsme provedli pomocí schématu `restriction_test.xsd`. Výsledné GUI je na obrázku B.3. Ukázka dialogového okna informujícího o neúspěšné validaci oproti schématu XSD je na obrázku B.4.

7.2 Configurator



Obrázek 7.2: Diagram tříd aplikace `Configurator`

Druhou implementací jádra je aplikace pro projektování řídícího jádra železničních přejezdů `Configurator`. Detailní funkcionalita aplikace přesahuje rozsah této práce, proto ji zde popisujeme jen stručně. Architektura aplikace je uvedena na diagramu 7.2. Součástí kapitoly je příloha C s obrázky.

Jedním z elementů konfiguračního souboru typu JAZZ je `VitalStack`. Ten obsahuje elementy `ANET`, z nichž každý reprezentuje nastavení právě jednoho zařízení, které řídící jádro železničních přejezdů ovládá. Úkolem této aplikace je na základě informací o počtu jednotlivých typů zařízení ovládaných řídícím jádrem, vytvořit pro každé z nich konfigurační soubor s kořenovým elementem `ANET`. Ty následně agregovat do elementu `VitalStack`, který vložíme do předem připraveného konfiguračního souboru typu JAZZ.

7.2.1 Návrh potřebných schémat

Pro zajištění maximální robustnosti řešení je nutné vznikající konfigurační soubory na všech úrovních validovat. Proto jsme pro každou úroveň konfigurace vytvořili XSD schéma. Pro konfiguraci jednoho zařízení s kořenovým elementem `ANET` schéma `single_anet.xsd`, pro element `VitalStack` schéma `vital_stack.xsd` a pro celý soubor typu JAZZ schéma `input_jazz.xsd`.

Některé prvky uvnitř elementu `ANET` je nutné generovat automaticky, například na základě konfigurace projektu, a nechceme, aby je uživatel mohl měnit. Pro generování GUI jsme proto vytvořili schéma `single_anet_gui.xsd`, definující podmnožinu prvků elementu `ANET`, jež je nutné zadat. Dodatečné požadavky na zadávaná data jsme definovali ve schématu jazyka Schematron `single_anet_gui.sch`.

Pro konfiguraci projektu používáme XML dokument, který je popsaný schématem `project_init.xsd`. Při zakládání projektu zadáváme název projektu, pod kterým jej uložíme, železniční stanici, pod kterou ovládané přejezdy spadají, a počty jednotlivých zařízení, která bude řídící jádro ovládat.

7.2.2 Adresářová struktura projektu

```
projects/
├── Kryry/ ukázkový projekt
│   ├── devices/ složka pro ukládání konfigurací zařízení
│   ├── cfg.bin
│   ├── compiled_jazz.xml
│   ├── derivation_script.txt
│   └── input_jazz.xml
└── Kryry.prj konfigurace projektu
```

7.2.3 Adresářová struktura aplikace

```
Configurator/
├── conf/
│   ├── input_jazz.xsd
│   ├── project_init.xsd
│   ├── single_anet.xsd
│   ├── single_anet_gui.sch
│   ├── single_anet_gui.xsd
│   └── vital_stack.xsd
├── jazzConfBin/ nástroj JazzConf
├── projects/ složka pro ukládání projektů
├── schematron/ transformační scénáře pro validaci
├── xaml/ vygenerovaná uživatelská rozhraní schémat
│   ├── project_init.xaml
│   └── single_anet_gui.xaml
├── Configurator.exe spouštěč aplikace
├── Gat.Controls.AboutBox.dll
├── GuiCreator.dll
├── GuiHostTools.dll
├── Xceed.Wpf.Toolkit.dll
└── XmlTools.dll
```

7.2.4 Implementace aplikace

Při spuštění aplikace je automaticky vytvořena instance třídy `MainWindow`, ovládací třída kompilovaného XAML se strukturou téměř totožnou s aplikací `HostApplication`. Mezi komponentami `Menu` a `contentDock` byl přidán `DockPanel` pro dodatečné informace a ovládací tlačítka.

`MainWindow` inicializuje komponenty a vytvoří instanci třídy aplikační logiky `MainWindowLogic`. Při výběru položky menu nebo stisku tlačítka, se zavolá příslušná metoda prostřednictvím metody `Execute()` třídy `GuiTools`. Tímto způsobem využíváme předpřipravený systém zpracování výjimek modulu `GuiTools`. Pouze položka „Help – About“ je obsluhována přímo metodou `About()` třídy `GuiTools`.

Položka „Import – Station DB“ zavolá metodu `ImportStationDB()`, která slouží k importu databáze železničních stanic ze souboru `.dat` do schématu `project_init.xsd`, prostřednictvím třídy `DatXsdConverter`. Stanice jsou přidány jako sada restrikcí `xs:enumeration` typu `staniceType`. Ke změně schématu je použita serializace pomocí třídy `XsdDeserializer` modulu `GuiCreator`.

Nabídka „Project“ slouží k zakládání nových projektů („New“), otevírání existujících („Open“) a zavírání právě otevřených („Close“).

Založení nového projektu

Pro založení nového projektu voláme metodu `NewProject()`. Ta vytvoří instanci třídy `NewProject`, která je ovládací třídou kompilovaného XAML, jehož struktura je totožná se strukturou XAML okna `MainWindow`. Po inicializaci komponent vytvoříme instanci třídy `GuiTools`, které předáme schéma `project_init.xsd`. Ta se postará o vytvoření uživatelského rozhraní pro tvorbu konfigurace projektu, vyobrazeného na obrázku C.1.

V GUI nastavíme počet železničních přejezdů (elementů `přejezd_PZZ`), zadáme název projektu, pod nímž se uloží, vybereme železniční stanici, pod kterou ovládané přejezdy spadají a nastavíme počty jednotlivých zařízení, která bude řídící jádro ovládat. Tlačítkem „Create“ konfiguraci s pomocí metody `Save()` třídy `GuiTools` uložíme do složky `projects`, jako soubor s názvem projektu a koncovkou `.prj`. Nakonec voláme metodu `ProjectInit()`.

Otevření existujícího projektu

Při otevření projektu voláme metodu `OpenProject()`. Zde pomocí metody `FileChooser()` třídy `GuiTools` vybereme dříve vytvořenou konfiguraci projektu (`.prj`) a zavoláme metodu `ProjectInit()`.

Zavření otevřeného projektu

Zavření projektu je realizováno metodou `CloseProject()`. Ta zavolá metodu `CloseGui()` třídy `GuiTools` a zneviditelní informační lištu s tlačítky. Jinými slovy uvede aplikaci do stavu, ve kterém byla při jejím spuštění.

ProjectInit()

Metoda `ProjectInit()` nejprve ve složce `projects` vytvoří složku s názvem projektu pro data projektu a její podsložku `devices` pro ukládání jednotlivých konfigurací zařízení. Pokud tyto složky již existují, nejsou vytvořeny. Následně je zavolána metoda `OpenValidInputJazz()`. Ta zkontroluje, zda je ve složce projektu přítomna vstupní konfigurace typu JAZZ `input_jazz.xml` a pokud ne, nechá ji uživatele vybrat pomocí metody `FileChooser()` třídy `GuiTools`. Soubor je validován oproti schématu `input_jazz.xsd` metodou `Validate()` třídy `XmlToolsProvider`, a pokud je v pořádku, nakopíruje se do složky projektu.

Poté je vytvořen `InputProcessor`. Ten zpracuje informace uvedené v konfiguraci projektu a vstupní konfiguraci typu JAZZ. Předpřipraví se data jednotlivých zařízení a upraví se schéma `single_anet_gui.xsd`, aby umožňovalo výběr zařízení dostupných ve vstupní konfiguraci.

Informace předzpracované třídou `InputProcessor` jsou předány instanci třídy `OutputProcessor`, která s nimi bude pracovat. Následně vytvoříme instanci třídy `GuiTools`, které předáme schémata `single_anet_gui.xsd` a `single_anet_gui.sch` pro vytváření konfigurace prvního zařízení. Výsledné uživatelské rozhraní je zobrazeno na obrázku C.2.

Nakonec voláme metodu `OpenDevice()`, která zkontroluje, zda ve složce `devices` není již konfigurace daného zařízení vytvořena, a pokud ano, je metodou `Open()` třídy `GuiTools` do uživatelského rozhraní načtena. Metoda se

stará také o povolování tlačítek „Previous“, „Next“ a „Compile Project“. Při konfiguraci prvního zařízení je zakázáno tlačítko „Previous“, při posledním je tlačítko „Next“ nahrazeno tlačítkem „Compile Project“.

Ovládací tlačítka

Tlačítko „Next“ volá metodu `NextDevice()`. Zde nejprve zavoláme metodu `SaveCurrentDevice()`, která pomocí metody `Save()` třídy `GuiTools` konfiguraci zařízení uloží. Zároveň zajistí, že nepřejdeme ke konfiguraci dalšího zařízení, aniž by byla aktuální konfigurace validní. Jestliže je vše v pořádku, zavoláme metodu `OpenDevice()` pro další zařízení. Stejným způsobem pracuje metoda `PreviousDevice()` tlačítka „Previous“, metodu `OpenDevice()` zde ale voláme pro předchozí zařízení.

Stisk tlačítka „Compile Project“ zpracovává metoda `CompileProject()`. Ta také nejprve zavolá metodu `SaveCurrentDevice()` pro uložení aktuálního záznamu a následně metodu `CompileOutput()` třídy `OutputProcessor`.

Sestavení výsledné konfigurace

Metoda `CompileOutput()` vytvoří ve složce projektu kopii vstupní konfigurace `input_jazz.xml` s názvem `compiled_jazz.xml`. Dokument pomocí technologie DOM načte, a syntézou dílčích konfigurací jednotlivých zařízení ve složce `devices` vytvoří obsah elementu `VitalStack`. Pro zpracování jednotlivých elementů `ANET` je volána metoda `CompileXmlOutput()`. Ta doplní informace zadané uživatelem o automaticky generované informace, aby byla konfigurace každého zařízení validní vůči schématu `single_anet.xsd`. K ověření validity je volána metoda `Validate()` třídy `XmlToolsProvider`.

Výsledný dokument je pro zajištění maximální robustnosti validován schématem `input_jazz.xsd`. Pokud je vše v pořádku, vytvoříme instanci třídy `FinalDerivation`, které projekt předáme. Ta vytvoří do složky projektu převodní skript `derivation_script.txt` nástroje `JazzConf`, voláním metody `CreateDerivationScript()`. Poté spustí metodou `RunJazzConfProcess()` proces `JazzConf.ConsoleUI.exe`, kterému parametrem předáme převodní skript. Výsledkem je konfigurační soubor `cfg.bin` typu JAZZ/CfgBin, který můžeme nahrát do řídícího jádra železničních přejezdů.

7.3 Srovnání výsledků se současným řešením

Naše řešení odstraňuje mnohé problémy, se kterými se současné řešení potýká. Největším přínosem je odstranění nutnosti ručních oprav konfiguračních souborů, které jsou nebezpečné, protože mohou do systému zanést obtížně odhalitelné chyby.

Pro konfigurační soubory typu JAZZ jsme vytvořili XSD schéma. Změna na úrovni tohoto souboru poté znamená pouze změnu schématu. Aplikace postavené na našem jádru se na tuto změnu automaticky adaptují, odpadá tak nutnost jakéhokoli ručního opravování.

Problémy, které nejsou obecně řešitelné nástrojem `JazzConf`, řeší zadavatel ruční editací konfiguračních souborů. Díky této práci je nyní možné pro všechny tyto problémy jednoduše vytvořit specializovanou aplikaci typu `Configurator`, která bude potřebné operace provádět automaticky, a striktní validací zajistí maximální robustnost.

Při archivaci konfiguračních souborů a k nim příslušných schémat, je jejich budoucí změna jednoduchá. Aplikace rekonstruují uživatelské rozhraní platné v době vytvoření těchto souborů a umožní je jednoduše měnit a ukládat.

8 Závěr

Cílem práce bylo navrhnout a implementovat jádro aplikace, která na základě XSD schématu popisujícího cílový dokument umožní vytvořit grafické uživatelské rozhraní, jež bude produkovat validní XML dokument. Výsledné jádro mělo být otestováno na konkretních implementacích klientských aplikací.

Teoretická východiska práce byla popsána v kapitolách 2, 3 a 4. Popisujeme zde použité XML technologie včetně alternativ, provádíme rozbor XSD schémat a seznamujeme čtenáře s možnými technologiemi vytváření grafických uživatelských rozhraní aplikací.

V praktické části práce byly vyřešeny všechny požadavky zadavatele a všechny řešitelné zásady pro vypracování vyplývající ze zadání. Zásady nejsou řešeny přímočaře v jednotlivých kapitolách, ale po částech v rámci různých fází práce. To je způsobeno tím, že důkladná analýza potřeb zadavatele proběhla v rámci práce až po jejím zadání.

Důkladnou analýzu potřeb zadavatele, včetně analýzy dosavadního nastavovacího programu jsme provedli v kapitole 5. Analýzou jsme zjistili, že třetí bod zadání je nutné splnit jiným způsobem – místo převodu jsme vytvořili nové schéma. Jak popisujeme v kapitole 5.2.2, neměl zadavatel pro potřebný druh konfiguračního souboru žádné XSD schéma, proto jsme jej nemohli převést do zvoleného návrhového vzoru (způsob zápisu). Během analýzy jsme provedli i průzkum existujících řešení, z nichž dvě jsme v kapitole 5.4 popsali.

V kapitole 6 jsme navrhli řešení generátoru XML souborů řízeného schématem a ve zvolené technologii jej implementovali. Důraz jsme kladli i na maximální robustnost řešení. V rámci návrhu jsme v kapitole 6.2.5 definovali omezující množinu použitých prvků XSD. Řešení jsme v kapitole 7 důkladně otestovali na reálných datech, srovnali výsledky se současným řešením a kriticky je zhodnotili.

Výsledkem práce je nástroj, který umožní vytváření dynamických aplikací pro maximální zjednodušení tvorby libovolně složitých XML dokumentů. Jeho použití přitom není vázáno na zadavatele práce, ale díky jednoduché integraci do klientské aplikace jej může použít kdokoli, kdo má o jeho funkcionalitu zájem.

Seznam použitých zkratek

API Application Programming Interface

DOM Document Object Model

DSDL Document Schema Definition Languages

DTD Document Type Definition

GUI Graphical User Interface

IDE Integrated Development Environment

ISO International Organization for Standardization

JAXB Java Architecture for XML Binding

JAXP Java API for XML Processing

JDK Java Development Kit

NVDL Namespace-based Validation Dispatching Language

SAX Simple API for XML

SDK Software Development Kit

StAX Streaming API for XML

SGML Standard Generalized Markup Language

SVRL Schematron Validation Report Language

URI Uniform Resource Identifier

W3C World Wide Web Consortium

WPF Windows Presentation Foundation

WXS W3C XML Schema

XAML eXtensible Application Markup Language

XSD XML Schema Definition

XML eXtensible Markup Language

XPath XML Path Language

XSL eXtensible Stylesheet Language

XSLT XSL Transformations

Literatura

Batra, U. *XSD2GUI and GUI2XML* [online]. 2013. [cit. 2014-05-05]. Dostupné z: <http://xsd2gui.sourceforge.net>.

Benz, B. – Durant, J. R. *XML Programming Bible*. Wiley Publishing, Inc., 2003. ISBN 0-7645-3829-2.

DSDL. *Domovská stránka* [online]. 2010. [cit. 2014-04-14]. Dostupné z: <http://www.dsdl.org>.

Fawcett, J. – Quin, L. R. E. – Ayers, D. *Beginning XML*. John Wiley & Sons, Inc., 2012. ISBN 978-1-118-16213-2.

Gattnar, C. *WPF About Box* [online]. 2012. [cit. 2014-04-28]. Dostupné z: <http://aboutbox.codeplex.com>.

GlassFish. *Project JAXP* [online]. 2014. [cit. 2014-04-12]. Dostupné z: <https://jaxp.java.net>.

Herout, P. *Java a XML*. Kopp, 2007. ISBN 978-80-7232-307-4.

Kay, M. H. *SAXON - The XSLT and XQuery Processor* [online]. 2013. [cit. 2014-04-15]. Dostupné z: <http://saxon.sourceforge.net>.

Kosek, J. *XML schémata* [online]. 2013. [cit. 2013-10-17]. Dostupné z: <http://www.kosek.cz/xml/schema>.

Kosek, J. *XML pro každého*. Grada Publishing, 2000. ISBN 80-7169-860-1.

MacDonald, M. *Pro WPF 4.5 in C♯*. Apress, 2012. ISBN 978-1-4302-4365-6.

Moser, C. *WPF Tutorial* [online]. 2011. [cit. 2014-04-29]. Dostupné z: <http://wpftutorial.net>.

MOTEN, D. *XSD-FORMS* [online]. 2014. [cit. 2014-05-05]. Dostupné z: <https://github.com/davidmoten/xsd-forms>.

MSDN. *.NET Development* [online]. 2014a. [cit. 2014-04-27]. Dostupné z: <http://msdn.microsoft.com/en-us/library/ff361664>.

MSDN. *System.Xml Namespace* [online]. 2014b. [cit. 2014-04-12]. Dostupné z: <http://msdn.microsoft.com/library/system.xml>.

MSDN. *Windows Forms* [online]. 2014c. [cit. 2014-04-29]. Dostupné z: <http://msdn.microsoft.com/en-us/library/dd30h2yb>.

MSDN. *Windows Presentation Foundation* [online]. 2014d. [cit. 2014-04-27]. Dostupné z: <http://msdn.microsoft.com/en-us/library/ms754130>.

MSDN. *System.Xml.Serialization Namespace* [online]. 2014e. [cit. 2014-04-12]. Dostupné z: <http://msdn.microsoft.com/library/system.xml.serialization>.

NVDL. *Domovská stránka* [online]. 2009. [cit. 2014-04-09]. Dostupné z: <http://www.nvdl.org>.

ORACLE. *Java Architecture for XML Binding (JAXB)* [online]. 2014a. [cit. 2014-04-12]. Dostupné z: <http://docs.oracle.com/javase/7/docs/technotes/guides/xml/jaxb>.

ORACLE. *Streaming API for XML* [online]. 2014b. [cit. 2014-04-12]. Dostupné z: <http://docs.oracle.com/javase/tutorial/jaxp/stax>.

ORACLE. *Swing (Java Foundation Classes)* [online]. 2014c. [cit. 2014-04-29]. Dostupné z: <http://docs.oracle.com/javase/7/docs/technotes/guides/swing>.

RELAX NG. *Domovská stránka* [online]. 2014. [cit. 2014-04-09]. Dostupné z: <http://relaxng.org>.

SAX. *Domovská stránka* [online]. 2004. [cit. 2014-04-12]. Dostupné z: <http://www.saxproject.org>.

SCHEMATRON. *Domovská stránka* [online]. 2010. [cit. 2014-04-09]. Dostupné z: <http://www.schematron.com>.

TIDWELL, D. *XSLT*. O'Reilly, 2008. ISBN 978-0-596-52721-1.

W3C. *Document Type Definition Declaration* [online]. 2008a. [cit. 2014-04-14]. Dostupné z: <http://www.w3.org/TR/xml/#dt-doctype>.

W3C. *Document Object Model (DOM) Level 3 Core Specification* [online]. 2004a. [cit. 2014-04-09]. Dostupné z: <http://www.w3.org/TR/DOM-Level-3-Core>.

W3C. *Extensible Markup Language (XML) 1.0 (Fifth Edition)* [online]. 2008b. [cit. 2014-04-09]. Dostupné z: <http://www.w3.org/TR/xml>.

W3C. *XML Path Language (XPath) 2.0 (Second Edition)* [online]. 2011. [cit. 2014-04-15]. Dostupné z: <http://www.w3.org/TR/xpath20>.

W3C. *XML Schema Part 0: Primer Second Edition* [online]. 2004b. [cit. 2014-04-09]. Dostupné z: <http://www.w3.org/TR/xmlschema-0>.

W3C. *XML Schema Part 2: Datatypes Second Edition* [online]. 2004c. [cit. 2014-04-25]. Dostupné z: <http://www.w3.org/TR/xmlschema-2>.

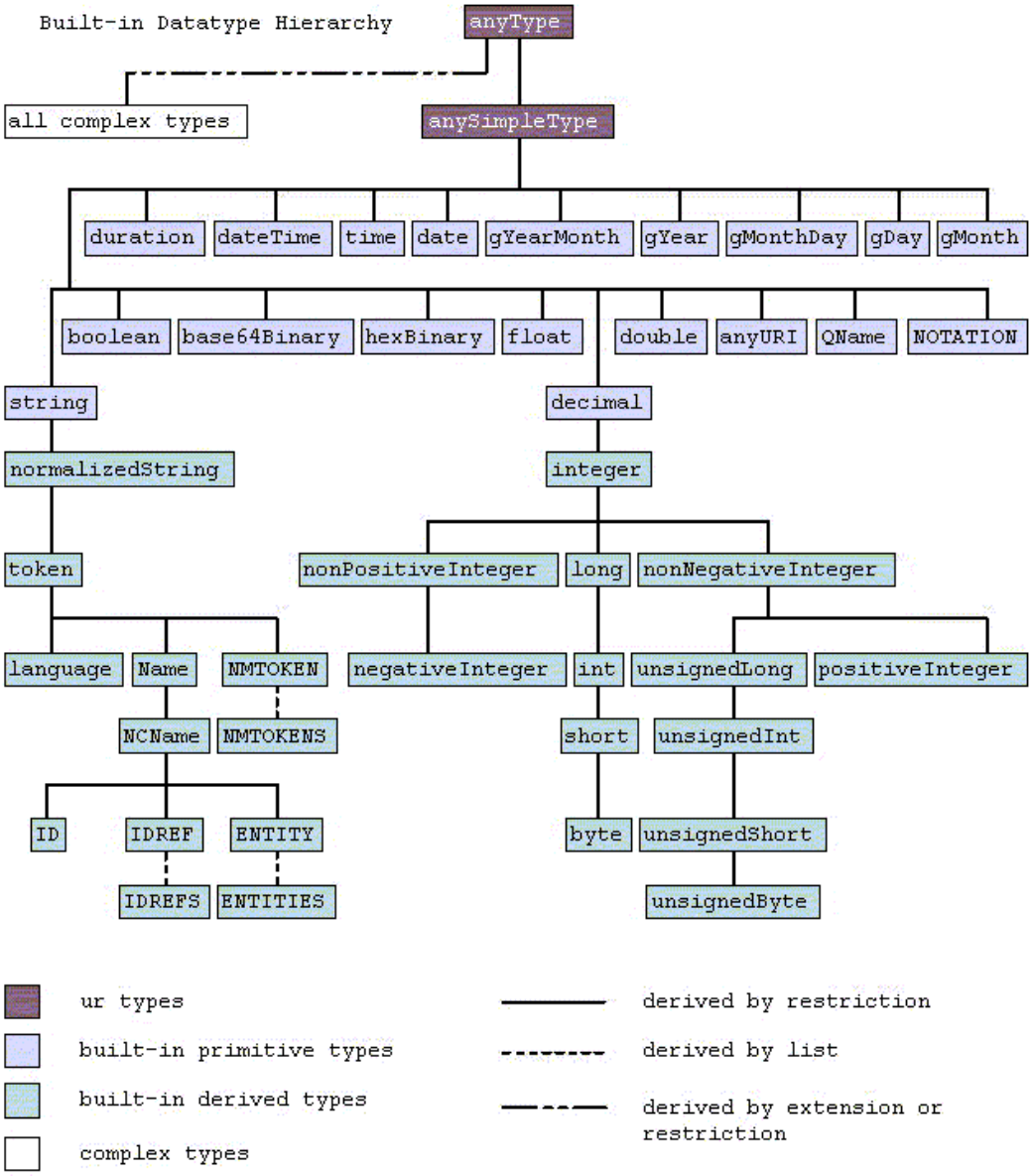
W3C. *XSL Transformations (XSLT) Version 2.0* [online]. 2007. [cit. 2014-04-15]. Dostupné z: <http://www.w3.org/TR/xslt20>.

W3C. *Domovská stránka* [online]. 2014. [cit. 2014-04-09]. Dostupné z: <http://www.w3.org>.

Xceed. *WPF About Box* [online]. 2014. [cit. 2014-04-28]. Dostupné z: <https://wpftoolkit.codeplex.com>.

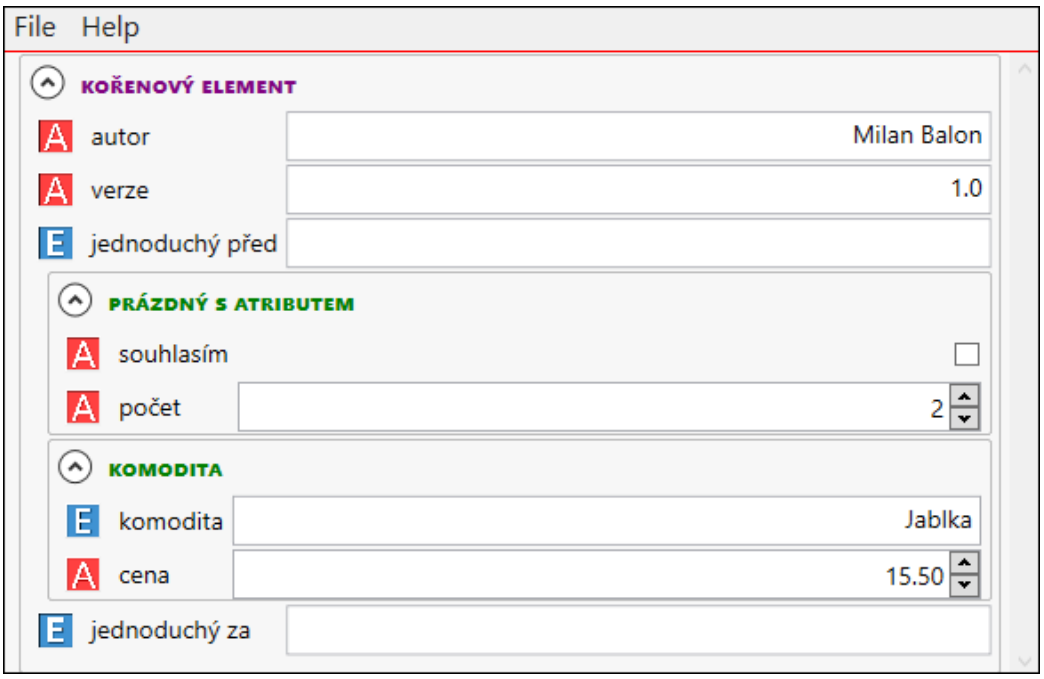
xFront. *Global versus Local* [online]. 2006. [cit. 2014-04-26]. Dostupné z: <http://www.xfront.com/GlobalVersusLocal.pdf>.

A Vestavěné datové typy XSD

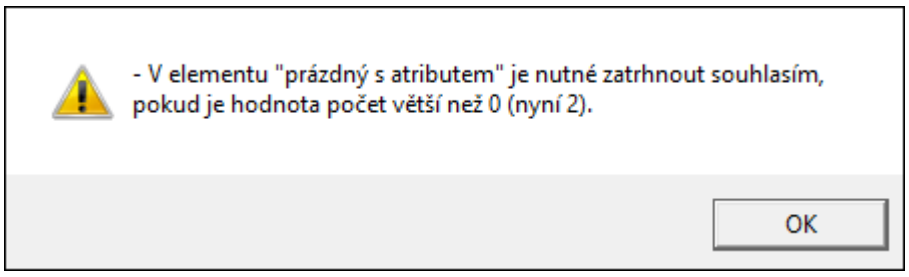


Obrázek A.1: Vestavěné datové typy XSD, Zdroj: (W3C, 2004c)

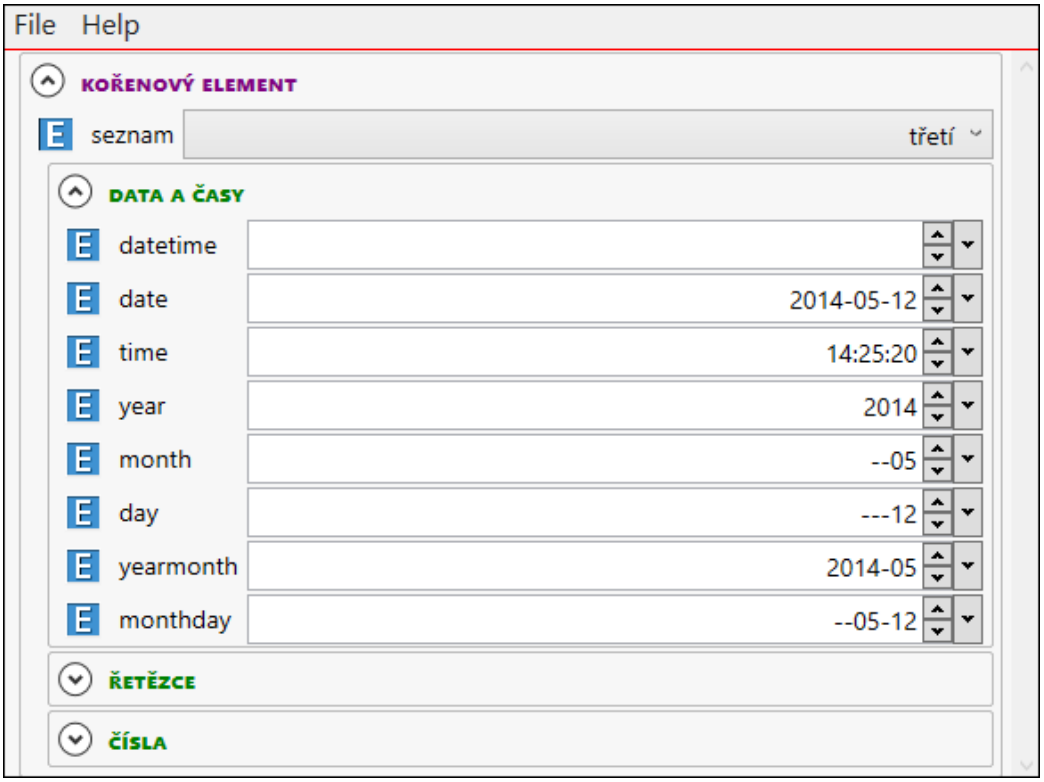
B Uživatelské rozhraní aplikace HostApplication



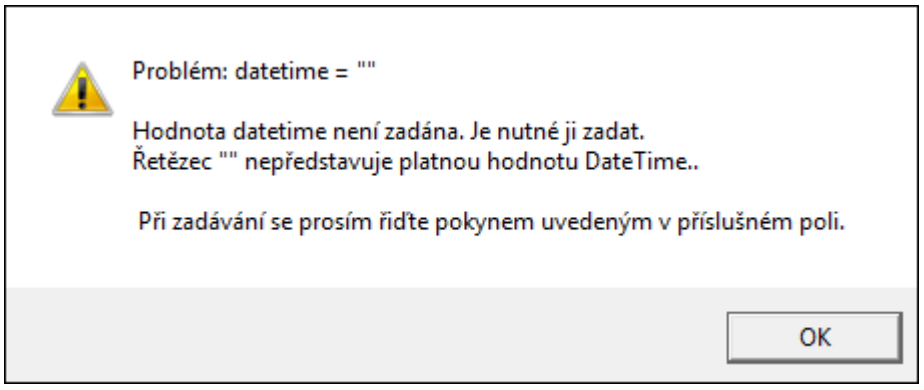
Obrázek B.1: Ukázka uživatelského rozhraní `structure_test.xsd`



Obrázek B.2: Chyba validace Schematron schématem

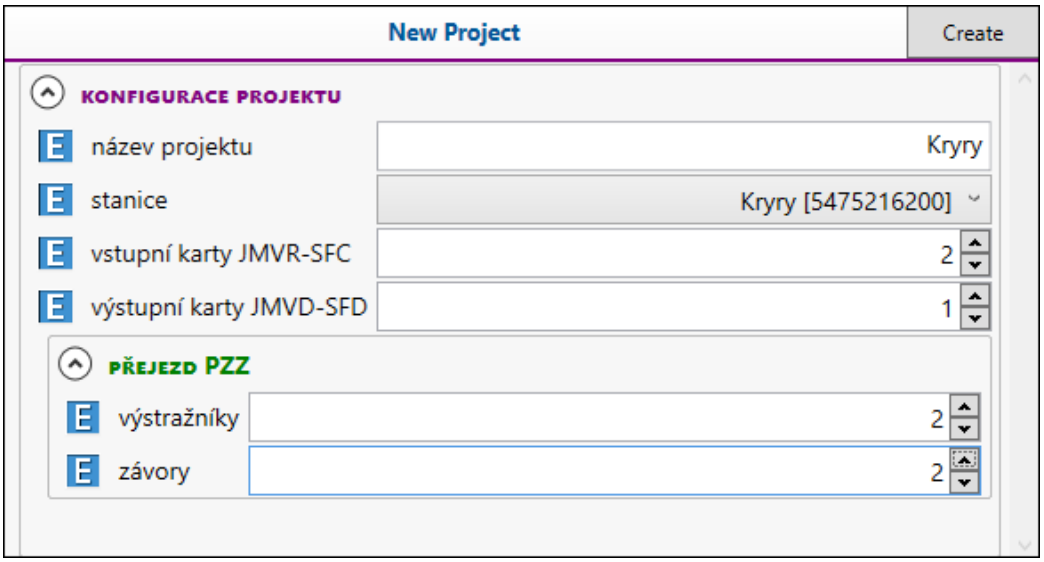


Obrázek B.3: Ukázka uživatelského rozhraní `restriction_test.xsd`

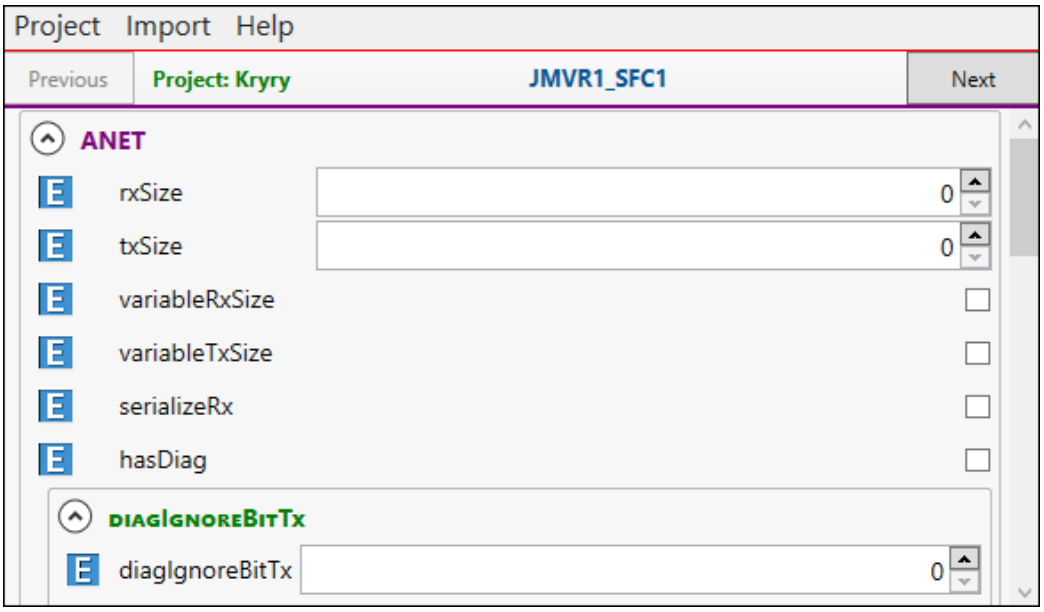


Obrázek B.4: Chyba validace XSD schématem

C Uživatelské rozhraní aplikace Configurator



Obrázek C.1: Konfigurace projektu



Obrázek C.2: Konfigurace zařízení