ZÁPADOČESKÁ UNIVERZITA V PLZNI

FAKULTA EKONOMICKÁ

Bakalářská práce

Návrh a implementace modulárního generátoru úloh

Design and implementation of a modular task generator

Milan Balon

Plzeň 2012

Prohlašuji, že jsem bakalářskou práci na téma

„Návrh a implementace modulárního generátoru úloh“

vypracoval samostatně pod odborným dohledem vedoucího bakalářské práce za použití pramenů uvedených v přiložené bibliografii.

V Plzni, dne 3. května 2012 ……..

Milan Balon

Poděkování

Děkuji vedoucímu bakalářské práce RNDr. Mikuláši Gangurovi, Ph.D., za jeho cenné připomínky a čas, který mi při řešení dané problematiky věnoval.

Obsah

Úvod

Cílem této práce je navrhnout a implementovat elementární části otevřeného počítačového systému, umožňujícího na základě vstupních dat vygenerovat jedinečné zadání úlohy, spolu s jejími výsledky. Systém by přitom měl být modulární, tedy jednoduše rozšiřitelný o další typy úloh a proměnných. Dílčími cíli práce jsou:

- návrh vhodné struktury vstupního XML dokumentu zadání úlohy
- návrh struktury pro uchování generovaných vstupních a výstupních dat
- harmonizace výstupního souboru se standardem QTI
- použití nejvhodnějších dostupných technologií
- otestování funkčnosti implementovaného systému
- návrh možností dalšího vývoje systému.

Jako nejvhodnější kandidát pro reprezentaci vstupních dat, byl zvolen platformě nezávislý formát XML, jehož výhody a filosofie je popsána v první kapitole teoretické části.

Výstupem tohoto systému je potom dokument v univerzálním XML formátu dle standardu QTI o kterém je v teoretické části také pojednáno. Z tohoto univerzálního formátu bude v budoucnu možné pomocí transformací vytvořit dokument v libovolném požadovaném formátu jako např. Adobe PDF (Portable Document Format), HTML atd.

Teoretická východiska práce jsou popsána v prvních pěti kapitolách, jež popisují základní technologie, které byly v práci použity. První kapitola pojednává o rozšiřitelném značkovacím jazyce XML, jeho historii a způsobu použití. Ve druhé kapitole jsou popsána XML schémata, pomocí kterých definujeme obsah XML dokumentů. Detailněji je zde popsáno W3C XML Schema, které je v práci použito. Třetí kapitola pojednává o normě QTI, jakým způsobem jsou dokumenty tvořeny, a které části specifikace v práci použijeme. Následující, čtvrtá kapitola popisuje problematiku XML parserů, což jsou nástroje pro zpracování XML dokumentů. Jednotlivé dostupné technologie jsou popsány spolu s výhodami a nevýhodami. V poslední kapitole teoretické části se zabýváme problematikou programovacích jazyků a výběrem nejvhodnějšího pro tuto práci. Jako nejvhodnější vývojové prostředí byl nakonec vybrán objektově orientovaný programovací jazyk Java.

Praktická východiska práce, obsahující konkrétní řešení problému, jsou popsána v kapitolách šest až deset. Kapitola šest je věnována návrhu vhodné struktury vstupního

XML dokumentu zadání úlohy spolu s definicí XML schématu. Sedmá kapitola popisuje proces kontroly správnosti vstupního dokumentu na základě definovaného schématu a načtení tohoto dokumentu do systému. Osmá kapitola se zabývá návrhem struktury pro uchování generovaných vstupních a výstupních dat spolu s návrhem a implementací podpůrných systémů pro zpracování. V deváté kapitole je popsáno, jakým způsobem generovaná data tiskneme do výstupního XML dokumentu se zachováním normy QTI.

Poslední kapitola práce se zabývá implementací jednotlivých částí systému v kompletním jádře generátoru úloh a použitím tohoto jádra v testovací aplikaci pro demonstraci funkčnosti navrženého řešení. Dále je zde popsána struktura distribuce hotového řešení na přiloženém CD.

1 XML

1.1 Úvod

Značkovací jazyk XML (eXtensible Markup Language) (W3C, 2008), slouží k uchování a zpracování strukturovaných textových dokumentů. Jedná se o standard konsorcia W3C (World Wide Web Consortium) (W3C, 2012, a), které zajišťuje standardizaci a vývoj technologií využívaných převážně v internetových aplikacích, a tento standard je velice hojně využíván díky jeho velmi dobrým vlastnostem. (Herout, 2007)

Jak již název jazyka napovídá, jedná se o rozšiřitelný značkovací jazyk. Základním stavebním kamenem jazyka je element, reprezentován otevírací značkou (tagem), obsahem a uzavírací značkou. Z elementů je potom sestaven celý dokument v požadované struktuře, které dosáhneme pomocí zanořování elementů do sebe.

Hlavní výhodou dokumentu v XML je nezávislost na platformě. Nezáleží proto na jakém počítači, operačním systému, či v jakém konkrétním softwaru dokument zpracováváme, vždy budeme schopni s ním pracovat. Toho je dosaženo díky tomu, že takový dokument není uložen v binární podobě, nýbrž v podobě prostého textu. V takovém dokumentu nemusíme řešit problémy s implementací ukládání čísel, například v pořadí bajtů big-endian či little-endian, a podobně. Jediné, co je pro korektní funkčnost nutné definovat, je použité kódování textových znaků. (Herout, 2007)

Díky této vlastnosti je známé následující tvrzení: „Java poskytuje přenositelný kód, XML přenositelná data“. (Herout, 2007, s. 17)

Z této věty můžeme vydedukovat, který programovací jazyk je vhodný pro práci s XML dokumenty využít. Řada konkurenčních programovacích jazyků sice obsahuje výborné nástroje pro práci s tímto formátem, ale žádný z nich plně nevyužije výhod spolupráce Javy s XML, spočívajícího právě v nezávislosti na použité platformě.

1.2 Historie

V historii vznikla řada značkovacích jazyků, tyto jazyky však byly zamýšleny jako prezentační, takže definovali pouze vzhled jednotlivých fragmentů dokumentu. Příkladem může být jazyk používaný programem TEX, sloužící k sazbě dokumentů pro tisk. V dnešní době informačních technologií však nechceme psát dokument pouze pro

tisk, chceme ho psát tak, abychom ho moli využít také k jiným druhům distribuce, například v podobě webové prezentace. To vedlo ke vzniku značkovacích jazyků, umožňujících vyznačit v textu význam, nikoli vzhled. (Kosek, 2000)

Jedním z takových jazyků byl SGML (Standard Generalized Markup Language), který původně využívala vládní infrastruktura Spojených států amerických (USA) pro výměnu dokumentů. Tento jazyk byl však příliš obecný a složitý, proto se příliš neujal. (Kosek, 2000)

Existuje však aplikace tohoto jazyka používaná v současnosti, a to HTML (Hypertext Markup Language), která se stala oblíbenou díky své jednoduchosti. Z důvodu existence potřeby definovat vlastní značky však nebylo HTML vhodné pro uchování strukturovaných informací a na druhé straně SGML bylo velmi složité na implementaci. Proto byl odstraněním některých nevyužívaných funkcionalit SGML vytvořen nový jazyk XML. XML je podmnožina SGML s téměř všemi výhodami tohoto jazyka. (Kosek, 2000)

1.3 Filosofie

Posláním XML je poskytnout nástroj pro výměnu informací ať už mezi lidmi, tak mezi aplikacemi. Každý subjekt má však jiné požadavky na obsah takového dokumentu a to právě XML řeší tím, že je možné definovat pravidla, co může dokument obsahovat (pomocí definování schématu, viz kapitolu XML schémata). Tato pravidla však nejsou nutná specifikovat. Existují ale pravidla, která musí splňovat každý XML dokument, a splňuje-li je, říkáme, že je tento dokument správně strukturovaný (well-formed). Tento mechanismus umožňuje snadné strojní zpracování dokumentu, jelikož víme, jakou musí mít strukturu. Pokud dokument požadovanou strukturu nemá, nezpracujeme jej. Připojíme-li k dokumentu schéma, můžeme kontrolovat, zda je XML dokument v souladu se schématem. Tomuto procesu se říká validace.

V zásadě se používají dva typy dokumentů, datově orientované dokumenty a dokumenty orientované na sdělení. Datově orientované dokumenty jsou dokumenty, obsahující strukturovanou informaci, kterou chceme nějakým způsobem předat či uchovat. To je tedy typ dokumentu, který potřebujeme pro návrh a implementaci vstupního souboru generátoru úloh. Dokumenty orientované na sdělení potom nesou informace určené primárně pro transformaci dokumentu do pro běžného uživatele čitelné podoby, například HTML. (Herout, 2007)

1.4 Základní syntaxe

Základním prvkem každého dokumentu je XML deklarace. Jedná se o jasnou identifikaci, že předložený textový dokument je XML dokumentem. Říká nám, jaká verze jazyka je použita a v neposlední řadě řeší problém s kódováním textových znaků na různých platformách tím, že je toto v deklaraci explicitně uvedeno. Deklaraci je nutné uvést hned na začátku dokumentu.

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

Jak již bylo řečeno v úvodu, základem dokumentu jsou elementy, které jsou tvořeny množinou: otevírací značka, obsah elementu, uzavírací značka. Otevírací značka dále obsahuje libovolně velkou množinu atributů a jejich hodnot.

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

V obsahu tohoto kořenového elementu je poté text, ve většině případů však další zanořené elementy, pro která platí stejná pravidla, je tak možné vytvořit složitou datovou strukturu.

Výše popsaný postup je potom základním mechanismem konstrukce XML dokumentu.

```
<?xml version="1.0" encoding="UTF-8" ?>
<clovek pohlavi="muž">
 <jmeno>Milan</jmeno>
 <prijmeni>Balon</prijmeni>
</clovek>
```

(Kosek, 2000)

2 XML schémata

Tato krátká kapitola pojednává o XML schématech, takže jak již bylo řečeno v kapitole předchozí, nástroji, který umožní říci, jak přesně by měl být dokument strukturován. Další klíčovou předností, proč vůbec vytvářet a připojovat schéma k našemu XML dokumentu, je možnost validace.

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).

2.1 DTD (Document Type Definition)

Jedná se o vůbec první jazyk, sloužící primárně pro popis dokumentů orientovaných na sdělení, kde se používá dodnes. Pro datově orientované dokumenty však vhodný není. (Herout, 2007)

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

Největší problém spočívá v tom, že schéma není psáno v XML, nýbrž speciální syntaxí. Dalším problémem je nemožnost definovat typy a rozsahy dat, které elementy obsahují, což je primární důvod, proč není DTD pro datově orientované dokumenty vhodný. Poslední nevýhodou je potom absence podpory práce se jmennými prostory, což je nástroj, kterým je možné zrealizovat použití více sad elementů v jednom dokumentu. (Herout, 2007)

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.

2.2 W3C XML Schema

Jak již název napovídá, je W3C XML Schema (W3C, 2012, b), stejně jako XML, dílem W3C konsorcia. Lze jej najít pod zkratkou WXS, častěji však jako XSD (XML Schema Definition).

Tento jazyk je v posledních letech preferován před jeho konkurenty Relax NG (OASIS, 2001) a Schematron (Schematron, 2012). I když jsou tyto 2 jazyky v několika ohledech lepší, jejich problémem je, že nejsou všeobecně rozšířené.

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

Podstatou tohoto jazyka je možnost definovat, jaká data může element obsahovat. Například atribut *pohlavi* definujeme tak, že nastavíme jeho obsah na normalizovaný řetězec, tedy řetězec bez bílých znaků a pomocí restrikcí můžeme dále definovat konkrétní hodnoty, rozsahy a podobně. V tomto případě nastavíme restrikci na hodnoty „muž“ a „žena“. Základní myšlenka potom je, že se veškerý obsah snažíme pomocí restrikcí co nejvíce omezit, díky čemuž může být aplikace, která s takovým dokumentem pracuje, mnohem jednodušší, protože nemusí obsah kontrolovat. Aplikace jen zkontroluje soulad se schématem, a pokud je dokument v pořádku, je možné data dále zpracovávat.

Další podstatnou funkcí je definice toho, v jakém pořadí mají elementy být, zdali tam vůbec musí být, a kolikrát.

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.

3 QTI

QTI (IMS GLC, 2012, a) je anglická zkratka pro specifikaci „Question & Test Interoperability“ konsorcia IMS Global Learning Consortium (IMS GLC, 2012, b), se sídlem v USA.

Tato specifikace popisuje datový model pro reprezentaci testových otázek, jejich výsledků a také mechanismů pro posouzení jejich správnosti. Systém je v základu navržen abstraktně pomocí UML (Unified Modeling Language)(UML, 2012) diagramů, a pro jeho přenositelnost je implementován pomocí jazyka XML. Použití této implementace je tvůrci doporučováno. (IMS GLC, 2005)

Poslední verze specifikace nese označení QTI v 2.1 - Public Draft Specification Version 2, avšak jedná se pouze o veřejný návrh a tvůrci upozorňují na to, že se jedná o nekompletní specifikaci, která ještě nebyla schválena IMS GLC. (IMS GLC, 2006) Z toho důvodu budeme nadále vycházet z poslední schválené verze, a to verze QTI v2.0 Final Specification, která byla zveřejněna v roce 2005.

Specifikace je navržena pro snadnou přenositelnost mezi vícero systémy, s nimiž pracují různí uživatelé s různými rolemi. Hlavní úloha systému je poskytnout dokumentovaný formát pro uchování a tvorbu testových otázek, jejich sběr v bankách úloh a výstup z těchto bank do jednoho výukového systému. V neposlední řadě poskytuje systémy pro reportování výsledků testu. (IMS GLC, 2005)

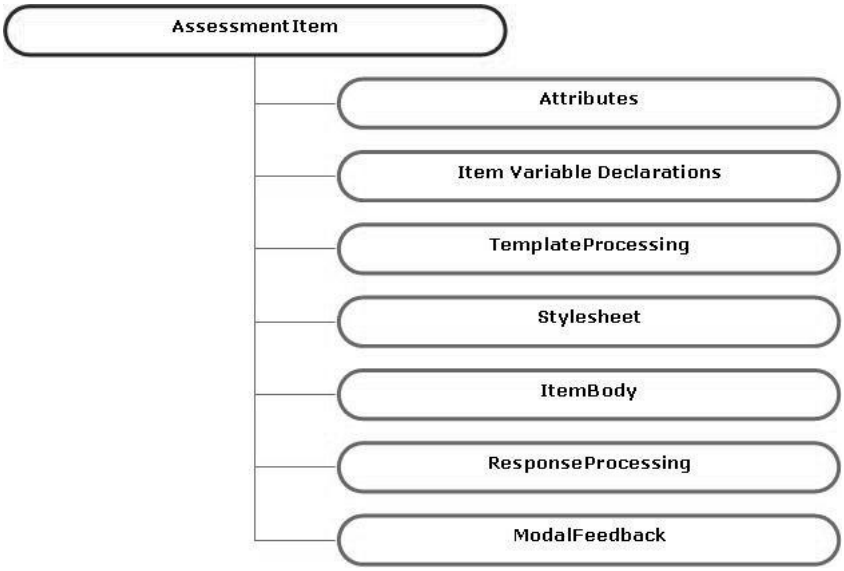
3.1 Validace

IMS GLC poskytuje dva nástroje pro validaci v podobě volně dostupných validačních souborů (schémat). První způsob, který poskytuje, je validace oproti DTD (*imsqti-v2p0.dtd*), druhý potom validace pomocí WXS, neboli XSD (*imsqti-v2p0.xsd*). Druhý způsob je IMS GLC doporučován. Výhody XSD oproti DTD u datově orientovaných dokumentů, což XML dokument dle specifikace QTI nesporně je, byly již detailně popsány v předchozí kapitole XML schémata.

Provedeme-li validaci oproti schématu, máme zaručenou kompatibilitu se standardem, což je jedna ze zásad, kterou chceme v rámci práce dodržet.

3.2 Definice úlohy

Úloha je v QTI definována elementem *assesmentItem*, což je kořenový element dokumentu, popisující právě jednu testovou úlohu. Toto je proto právě ta část QTI specifikace, která nás pro implementaci generátoru úloh zajímá. Veškeré informace o úloze jsou potom obsahem tohoto elementu. Element má dále povinné a volitelné atributy, které blíže specifikují konkrétní úlohu.



Obr. č. 1: Struktura elementu assesmentItem (Zdroj: (JISC CETIS, 2006))

3.2.1 Atributy

Atributy elementu *assesmentItem* nesou základní informace o úloze. V následující tabulce je přehled základních atributů předdefinovaných v QTI specifikaci. Jak již bylo řečeno, specifikace obsahuje povinné a volitelné atributy. Krom těchto atributů obsahuje element samozřejmě také atributy, které nesou informaci o jmenných prostorech a použitém schématu.

Tab. č. 1: Přehled atributů elementu assesmentItem

Atribut	Popis	Povinnost výskytu
identifier	unikátní identifikátor úlohy	povinný
title	krátký titulek	povinný
adaptive	říká, zda se úloha má přizpůsobovat na základě počtu pokusů o její vyřešení	povinný
timedependent	říká, zda je hodnocení úlohy ovlivněno rychlostí zodpovězení	povinný
label	krátké označení úlohy	volitelný
language	jazyk, ve kterém je úloha napsána	volitelný
toolname	jméno nástroje, ve kterém byla úloha generována	volitelný
toolversion	verze nástroje, ve kterém byla úloha generována	volitelný

Zdroj: (JISC CETIS, 2006)

3.2.2 Definice proměnných

Proměnné slouží k uchování informací uvnitř testové otázky. Existují tři druhy proměnných:

1. Response variables, neboli proměnné deklarované pro odpovědi, slouží pro uchování odpovědí.
2. Outcome variables, neboli výstupní proměnné, slouží pro uchování bodového ohodnocení otázky.
3. Template variables, neboli proměnné deklarované pro šablony, slouží pro potřeby mechanismu klonování otázek pomocí šablon.

Jak vidíme, pro potřeby implementace výstupního souboru generátoru úloh budeme potřebovat proměnné deklarované pro odpovědi, sloužící pro uchování informace o správném výsledku. Tato proměnná se definuje v sekci *responseDeclaration*.

Každý druh proměnné je tak definován ve zvláštní sekci pro ni určené. První typ je, jak již bylo řečeno, deklarován v elementu *responseDeclaration*, druhý typ je deklarován v sekci *outcomeDeclaration* a poslední typ v sekci *templateDeclaration*.

Nyní již víme, jak ve výstupním souboru definovat výslednou hodnotu. Proměnná s výsledkem má dva povinné a jeden nepovinný atribut. *identifier* obsahuje jméno proměnné a je povinný. *cardinality* říká, kolik hodnot může proměnná obsahovat a je také povinný. Jeho nejčastější hodnota je „single“, což znamená, že obsahuje jednu hodnotu. Pokud by proměnná obsahovala více hodnot (multiple, ordered, record), říká

se proměnné kontejner. Poslední atribut je *baseType* a nese informaci o typu proměnné. V následujícím příkladu je to integer, tedy celé číslo, a tento atribut je nepovinný.

```
<responseDeclaration identifier="vysledek" cardinality="single"
baseType="integer" />
```

Výše zmíněnou deklarací definujeme proměnnou, kam se uloží odpověď od uživatele. Můžeme ale ještě definovat defaultní hodnotu proměnné pro případ, že nebude zadána, správnou hodnotu proměnné pro pozdější vyhodnocení a nakonec můžeme nastavit mapování odpovědi na číslo, což může také sloužit potřebám strojního vyhodnocování (například přiřadíme slovním odpovědím určitá bodová ohodnocení). V následujícím přikladu bude popsáno, jak specifikovat správnou hodnotu proměnné, což je ve většině případů užití nutné provést.

```
<correctResponse>
  <value fieldIdentifier="vysledek_correct">50</value>
<correctResponse>
```

3.2.3 Definice těla testové otázky

Tělo testové otázky je hlavní část otázky, ve které je vlastní znění otázky spolu s bloky určenými pro interakci s uživatelem neboli testovanou osobou. Tělo otázky je reprezentováno elementem *itemBody*. Tento element potom obsahuje bloky pro interakci (multiple choice, posuvníky, přiřazování,…) a statické bloky, například odstavce (element „*p*“). Existuje také možnost definovat vlastní bloky pro interakci.

3.2.4 Zpracování odpovědi

Část pro zpracování odpovědí je reprezentována elementem *responseProcessing*, a je v ní popsáno, jakým způsobem vyhodnotit uživatelský vstup. Vyhodnocení probíhá pomocí základních podmínek používaných i v programovacích jazycích. Hodnotíme-li odpověď, uzavřeme tento proces do elementu *responseCondition*, který musí povinně obsahovat element *responseIf,* obsahující podmínku. QTI obsahuje řadu elementů, které můžeme v podmínkách použít. V následujícím příkladu je použit element *equal*, který testuje na rovnost, což je asi nejčastější činnost, kterou chceme provádět.

Povinným atributem je *toleranceMode*, kde nastavíme, zda má být shoda přesná (exact) a poté nemusíme toleranci nastavovat nebo toleranci povolujeme. Tolerance může být udána v procentech (relative) nebo absolutně (absolute) a uvádíme ji v atributu *tolerance*. Zbývá definovat proměnnou, kterou kontrolujeme a její správnou hodnotu

pro porovnání, to děláme pomocí elementů *variable* a *correct* a jako atributy *identifier* použijeme hodnoty, které jsme zvolili v *responseDeclaration*.

```
<responseProcessing>
  <responseCondition>
    <responseIf>
      <equal toleranceMode="absolute" tolerance="0.1">
        <variable identifier="vysledek"/>
        <correct identifier="vysledek_correct"/>
      </equal>
    </responseIf>
  </responseCondition>
</responseProcessing>
```

Chceme-li přiřazovat bodové ohodnocení, pro které jsme si připravili proměnné v sekci *outcomeDeclaration*, použijeme uvnitř bloků *responseIf* a *responseElse* element *setOutcomeValue* s atributem *identifier*, vedoucím k příslušné proměnné a bodové ohodnocení bude potom obsahem tohoto elementu.

4 XML parsery

Základní činností, kterou chceme s dokumentem ve formátu XML provádět, je čtení informací z něj. První věc, co bychom s dokumentem mohli udělat, je otevřít ho v programovacím jazyce 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, které to umožňují a nazývají se parsery. Jedná se o nástroj, který nám umožňuje pracovat s XML dokumentem. Termínem pracovat se potom rozumí číst jej, měnit stávající elementy, přidávat nové elementy a transformovat do jiných formátů. (Herout, 2007)

4.1 SAX

SAX (Simple API for XML) (SAX, 2012) je jeden ze základních parserů, 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.

Parser pracuje s proudem dat a tudíž používá proudové čtení dat. To 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.

SAX je push parser a to znamená, že čtení probíhá automaticky od začátku do konce. Parser přitom v průběhu čtení vyvolává obslužné funkce, které s načteným kusem dokumentu pracují (například narazí-li na otevírací značku, zavolá parser funkci `startElement()`, jejíž obsah musíme sami vytvořit, a tím nám umožní na tuto skutečnost reagovat). (Herout, 2007)

Tento parser 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 umožňuje automaticky, bez nutnosti reagovat na události.

4.2 DOM

DOM (Document Object Model) (W3C, 2004) je druhým základním parserem a jedná se o standard společnosti W3C pro práci s XML. Tento parser je široce používán a podporován a to právě z důvodu, že se jedná o standard W3C.

Parser pracuje se stromovou reprezentací dokumentu. 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 „překlopit“ zpět do XML.

Tento způsob s sebou nese řadu výhod a nevýhod. Výhodou je, že máme celý infoset v paměti a můžeme s ním pohodlně pracovat, a jelikož je v paměti vnitřní, tak je tato práce i velice rychlá. 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)

Vidíme, že tento parser použijeme, chceme-li v dokumentu něco měnit nebo vkládat nové elementy. Tento dokument však nesmí být příliš objemný, aby se vešel do vnější paměti. Chceme-li dokument pouze číst, není DOM vůbec vhodné řešení.

4.3 StAX

StAX (Streaming APIs for XML) (Oracle, 2012) je speciální parser vyvinutý speciálně pro programovací jazyk Java a byl zveřejněn v balíku JWSDP (Java Web Services Developer Pack) v roce 2007. Budeme-li však hledat nějaké informace o tomto parseru, zjistíme, že StAX není jediné jméno, které ho reprezentuje, najdeme ho totiž ještě pod

kódovým jménem Zephyr nebo zkratkou SJSXP (Sun Java Streaming XML Parser). (Herout, 2007)

StAX je stejně jako SAX proudový parser, nikoli však typu push, ale pull. Pull parser znamená, že čtení neprobíhá automaticky ale na naši žádost. Můžeme tak zpracovat část dokumentu, potom dělat něco jiného a časem se vrátit a zpracování dokončit.

Tento parser umí dokument i vytvářet a měnit, výhoda oproti DOMu potom vyplývá z proudového zpracování, kdy není nutné nic ukládat do paměti, a proto je vytváření a měnění dokumentů paměťově nenáročné. Způsob zápisu programu pracujícího se StAXem je také ve většině případů mnohem jednodušší, než s technologií SAX nebo DOM. Posledním specifikem oproti předchozím parserům je fakt, že pomocí StAX nelze dokument kontrolovat proti schématu. To může být jak výhoda, tak nevýhoda, záleží na způsobu použití.

Chceme-li pracovat s XML dokumentem v programovacím jazyce Java, je ve většině případů vhodnější sáhnout po tomto parseru, i když například pro validaci budeme muset použít SAX. Musíme však vždy zvážit, zda není lepší použít místo proudového parseru některou technologii se stromovou reprezentací dokumentu, jako DOM nebo JAXB. To se hodí zejména v případě, provádíme-li mnoho modifikací na více místech a je nutné se vracet.

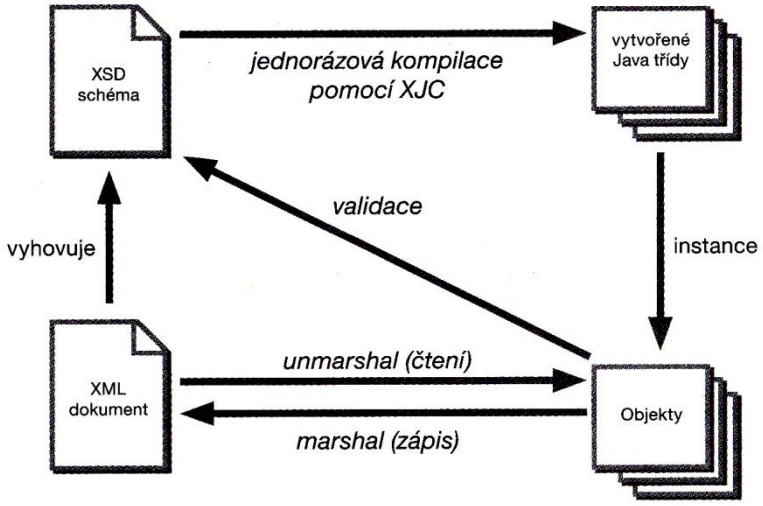
4.4 JAXB

JAXB (Java Architecture for XML Binding) (Oracle, 2003) je technologie pro zpracování XML, která je stejně jako StAX součástí JWSDP a jedná se o specialitu určenou pouze pro programovací jazyk Java.

Fungování této technologie je zcela odlišné od fungování technologií předešlých a staví na spolupráci s XSD schématy. XSD schéma je totiž nejprve pomocí speciálního programu *xjc* přetvořeno na sadu tříd v Javě, které po zkompilování využíváme v našem programu. Vytváříme-li zcela nový dokument, jednoduše ho sestavíme v Javě pomocí objektů a tyto objekty následně vytiskneme pomocí nástroje marshaller do XML. Chceme-li dokument číst nebo měnit, použijeme nástroj unmarshaller, který dokument rozparsuje do objektového modelu. Tento model potom libovolně měníme a nakonec zase pomocí marshalleru uložíme. Celý princip je vyobrazen na ilustračním obrázku.

Výhodou tohoto řešení je fakt, že jsme kompletně odstíněni od problematiky XML a s dokumenty pracujeme čistě objektově v programovacím jazyku. Nevýhoda potom je, že musíme mít k dispozici XSD schéma, to by ale mělo existovat pro všechny XML dokumenty, které jsou určeny pro strojové zpracování. Hlavní nevýhoda je stejná jako u technologie DOM, a to paměťová náročnost, jelikož je objektový model, stejně jako infoset, uložen ve vnitřní paměti.

Technologii JAXB se vyplatí použít především při dynamickém vytváření zcela nových dokumentů, což je oproti technologiím StAX nebo DOM značně jednodušší. Lze ji však, po zvážení všech výhod a nevýhod, využít pro veškerou práci s XML dokumenty.



Obr č. 2: Princip fungování JAXB (Zdroj: (Herout, 2007))

5 Programovací jazyky

5.1 Úvod

Definice programovacího jazyka dle Wikipedie říká: „Programovací jazyk je prostředek pro zápis algoritmů, jež mohou být provedeny na počítači. Zápis algoritmu ve zvoleném programovacím jazyce se nazývá program.“ (Wikipedie, 2012, a) Jedná se tedy o nástroj, jak říci počítači, co má udělat. Program je potom předpisem těchto akcí.

Základní dělení programovacích jazyků je na strojově orientované a vyšší jazyky. Strojově orientované jazyky slouží pro zápis programu pomocí instrukcí určených pro konkrétní procesor a napíšeme-li v tomto jazyce program, lze ho použít jen na úzké skupině počítačů. Příkladem je jazyk symbolických adres, neboli jazyk assembleru, kdy assembler je překladač tohoto jazyka do strojového kódu.

Druhá skupina programovacích jazyků, takzvané vyšší jazyky, jsou charakteristické tím, že jejich syntaxí není zápis instrukcí procesoru, ale snaží se přiblížit takovému zápisu, jakým způsobem přemýšlí lidský mozek. Tím se stává programování jednodušším a programy jsou použitelné na širší škále počítačů.

Vyšší jazyky se potom v základu dělí na procedurální, neprocedurální a speciální. Speciální jazyky slouží pro vykonávání specifických činností, příkladem může být jazyk SQL, který slouží speciálně pro komunikaci se systémem řízení báze dat. Neprocedurální jazyky se liší od procedurálních tím, že popisují co je cílem programu a nikoli algoritmy, jakým způsobem k cíli dojít, což je doména procedurálních jazyků. Mezi tyto jazyky se řadí například Lisp nebo Prolog. Poslední skupina programovacích jazyků, jsou jazyky procedurální, které pomocí algoritmů popisují přesný postup řešení problému. Tento typ programovacích jazyků je v současné době nejrozšířenějším jazykem pro tvorbu softwaru.

Procedurální programovací jazyky se dále dělí na strukturované a objektově orientované. Mezi nejpoužívanější strukturované programovací jazyky patří jazyky C, (Visual) Basic, Perl a Pascal. Majoritu v programovacích jazycích však v současné době tvoří objektově orientované jazyky, jejichž zástupci jsou jazyky Java, C#, C++, Objective-C, PHP. (Herout, 2009)

Základní rysy procedurálních jazyků jsou:

- zpracovávané údaje mají formu datových objektů různých typů, které jsou v programu reprezentovány pomocí proměnných resp. konstant
- program obsahuje deklarace a příkazy
- deklarace definují význam jmen (identifikátorů)
- příkazy předepisují akce s datovými objekty nebo způsob řízení výpočtu

(Zdroj: Herout, 2009)

5.1.1 Strukturované programování

„Strukturované programování (též strukturovaný programovací jazyk) označuje v informatice programovací techniku, kdy se implementovaný algoritmus rozděluje na dílčí úlohy (t.j. procedury či funkce), které se spojují v jeden celek.“ (Wikipedia, 2012, b)

Celý program potom běží jako sekvence těchto bloků za sebou a základní myšlenka spočívá v zákazu příkazu skoku. Běh programu je tak řízen jen pomocí řídících struktur (podmínky,…), kdy je na základě současného stavu vybrán určitý blok kódu. Poslední povolenou akcí jsou cykly, kdy je sekvence opakována, dokud není splněna určitá podmínka.

5.1.2 Objektově orientované programování (OOP)

Nejpokročilejší myšlenka psaní programu spočívá v objektově orientovaném programování, kdy je celý program poskládán z objektů, které mají za úkol modelovat objekty reálného světa. Tyto objekty jsou potom pouze modely pracující s určitou mírou abstrakce, kterou v konkrétním programu od objektu požadujeme. Cílem je potom namodelovat reálný problém pomocí jednotlivých objektů.

Výhoda tohoto přístupu je, že napíšeme-li objekt kvalitně jednou, můžeme ho v nezměněné podobě použít v jiných projektech a program vystavět jako stavebnici z hotových dílů. (Wikipedia, 2012, c)

Abstraktní popis objektu je popsán ve třídě, která obsahuje atributy nesoucí informaci o stavu objektu a metody, které popisují schopnosti třídy. Takto popsaná třída poté slouží jako šablona pro tvorbu instancí tříd, jinak také objektů. (Herout, 2008)

Základní principy OOP jsou zapouzdření, dědičnost a polymorfismus. Zapouzdření spočívá v tom, že zakážeme zvenčí přistupovat k atributům objektu, což by při jejich neautorizované změně mohlo způsobit nekonzistenci dat. Atributy povolíme měnit jen pomocí metod uvnitř objektu a tyto metody se nazývají getry (pro zjištění stavu) a setry (pro nastavení stavu).

Dědičnost je nástroj, pomocí kterého vytvoříme třídu, která má všechny atributy a metody jako rodičovská třída, a v této třídě poté můžeme přidávat nové atributy a metody, nebo původní metody přepsat. Nejčastěji se tímto způsobem vytvářejí ze základního objektu, objekty konkrétnější. Například třída *Dveře* má metody `otevřít()` a `zavřít()`, což jsou dovednosti, které by měli splňovat všechny dveře. Od ní zděděná třída *DveřeAutomobilu* má navíc metody `stáhniOkénko()` a `vytáhniOkénko()`. Použijeme-li v programu instanci třídy *DveřeAutomobilu* a zavoláme metodu `otevři()`, provede se metoda, která je zapsána ve třídě *Dveře* a ve třídě *DveřeAutomobilu* vůbec není a být nemusí. Tato technika velice zjednodušuje

psaní nových tříd, které jsou většinou podobné třídám, které již někdo v minulosti napsal.

Posledním základním principem OOP je polymorfismus neboli mnohotvarost. Tento princip vychází z předchozího principu dědičnosti, a spočívá v tom, že můžeme manipulovat se sadou různých objektů jako s objektem jednoho typu. Podmínkou však je, že tento jednotný typ je společným předkem všech objektů, s kterými chceme nějakým způsobem manipulovat. Příkladem může být program na vykreslení čtverce, obdélníku a kruhu. Jejich společným předkem je třída *GeometrickýObjekt*, která obsahuje metodu `vykresli()`. Polymorfismus nám potom dává možnost postupně náhodně vytvářet výše zmíněné geometrické objekty a ukládat je například do seznamu. Každý objekt může mít rozličné atributy a speciální metody, metodu `vykresli()` však obsahují všechny z nich. Ve chvíli, kdy je chceme vykreslit, jenom jednoduše projdeme celý seznam a nad každým jednotlivým objektem zavoláme metodu `vykresli()`.

Na tomto příkladu jsou vidět výhody tohoto přístupu, kdy můžeme s libovolně velkou sadou objektů manipulovat jako s jedním, což přináší značné zjednodušení.

5.2 Java

Vývoj programovacího jazyka Java, jež byl původně určen pro vestavěné systémy (mikroprocesorem řízená zařízení), započal v roce 1991 (tehdy ještě s pracovním názvem „Oak“) ve firme Sun Microsystems. Později však, z důvodu nepříliš veliké popularity a vzrůstajícímu zájmu o webové aplikace, nabral projekt nový směr. Programovací jazyk byl oficiálně představen v roce 1995 a okamžitě bylo jasné, jaký v sobě Java skýtá potenciál. (Herout, 2008)

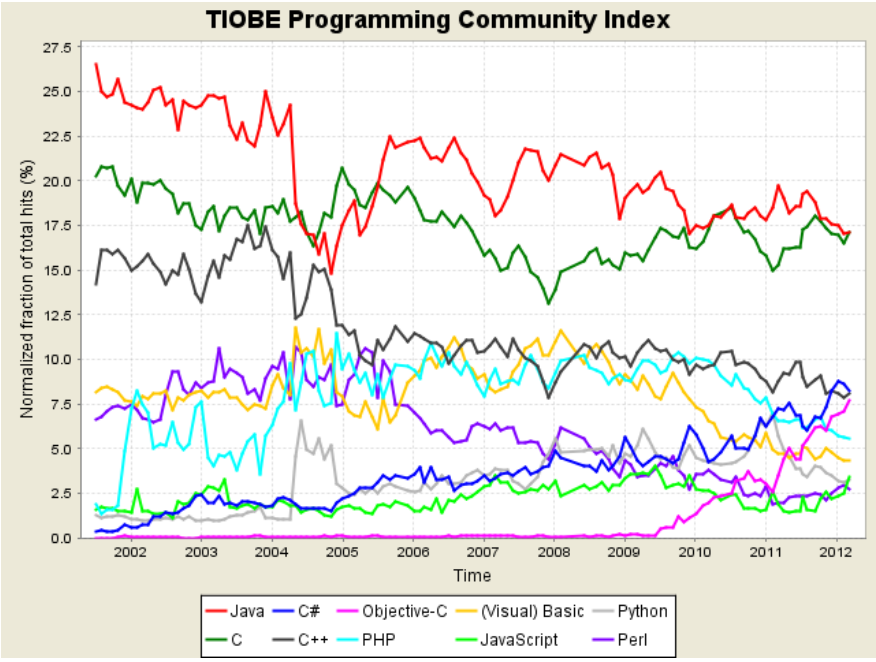
V roce 2009 byla společnost Sun Microsystems odkoupena společností Oracle, a Javu tak nyní vlastní společnost Oracle. Nic se tím však nemění a její vývoj bude nadále pokračovat. (*Redwood Shores, CA.*, 2009)

Základní vývojový nástroj Javy se nazývá JDK (Java Development Kit), který je firmou Oracle poskytován kompletně zdarma. Je rozdělen do tří balíčků, JDK SE (Standard Edition), který slouží pro vývoj základních aplikací, JDK ME (Mobile Edition), určený pro vývoj aplikací pro mobilní zařízení a JDK EE (Enterprise Edition), určený pro podnikové a webové aplikace.

Značení verzí JDK je dosti problematické, hlavní verze je stále 1, subverze se měnila v průběhu času od 0 až po 7, poslední subverze je 1.7. Nakonec je označeno číslo update verze. Z toho důvodu, že se hlavní verze nemění, označuje se JDK pouze čísly subverze a update, například JDK 7u1 a takto je to i prezentováno na webových stránkách spolčenosti Oracle, kde je JDK k dispozici volně ke stažení.

Hlavní výhoda aplikací napsaných v programovacím jazyce Java, je jejich multiplatformní přenositelnost. Aplikace lze spustit na libovolném systému, pro který je k dispozici platforma Java. Platforma Java se skládá z virtuálního stroje JVM (Java Virtual Machine) a Java Core API (Application Programming Intefrace). Java Core API je balík základních knihoven, které musí být k dispozici pro běh aplikace napsané v Javě, a vývojář má tak zaručeno, že při použití těchto knihoven nenastane po přenesení aplikace žádný problém. JVM potom zprostředkuje běh zkompilované aplikace na daném hardwaru.

Dle výzkumu společnosti TIOBE Software, založeném na ratingu podle frekvence vyhledávání na osmi největších internetových vyhledávačích (Tiobe, 2008), je programovací jazyk Java během posledních deseti let dlouhodobě nejpoužívanější objektově orientovaný jazyk (17.110%). Absolutní prvenství však nemá, neboť stále soupeří s velice populárním strukturovaným procedurálním jazykem „C“ (17.087%). Tyto dva programovací jazyky jsou tak v současné době na vrcholu. (Tiobe, 2012)



Obr č. 3: Vývoj oblíbenosti programovacích jazyků (Zdroj: (TIOBE, 2012))

6 Návrh struktury vstupního XML dokumentu zadání úlohy

První problém, který byl při návrhu systému nutný vyřešit, spočíval v návrhu vstupní šablony zadání úlohy. Nejvhodnější formát pro uchování strukturovaných textový dat, je v současné době formát XML. Ten nám, jak již bylo řečeno v samostatné kapitole XML, poskytne multiplatformní přenositelnost, přehlednost a v neposlední řadě umožní snadné strojové zpracování.

Při řešení tohoto problému byla snaha o to, aby byl tento dokument co nejjednodušší. Tento požadavek vychází z toho, že vstupní zadání sestavuje člověk a nikoli stroj. Proto je nutné, aby tato práce byla co nejefektivnější a to jak z hlediska rychlosti, tak nízké chybovosti.

Při návrhu XML dokumentu musí nejprve padnout rozhodnutí, jaké kódování pro tento dokument použijeme. V celé práci použijeme celosvětově nejrozšířenější a všeobecně podporované kódování s proměnnou délkou znaku (1-4 bajty) UTF-8. Toto kódování je navíc implicitně použito všemi XML dokumenty v případě, že jej neuvedeme v XML deklaraci. Ta tak bude pro všechny dokumenty vypadat takto:

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

Pro popis XML dokumentu použijeme jazyk XSD. Důvodem je fakt, že se jedná o datově orientovaný dokument, a jak již bylo řečeno v kapitole o XML schématech, pro popis datově orientovaných dokumentů je XSD velice vhodná volba. Tento jazyk má předdefinovaný jmenný prostor *xs*, tak v následujícím textu jednoduše identifikujeme příklady vycházející ze schématu (*<xs:značka>*).

Základem vstupního dokumentu jsou prázdné elementy *vi* (vstupní proměnná) a *vo* (výstupní proměnná) se dvěma povinnými atributy, *name* a *type*.

```
<vi name="operator" type="Text"/>
```

Tento element nám říká, že na jeho místě má být dynamicky generovaná informace. Atribut *name*, který si libovolně zvolíme sami, nese informaci o jménu proměnné. Toto jméno musí být typu *normalizedString*, což znamená řetězec bez bílých znaků na začátku či konci. Tento typ je vhodné použít u všech atributů, proto v dalším textu nebude uvedeno, jakého typu atributy jsou, a budeme předpokládat, že jsou právě typu *normalizedString*.

```
<xs:simpleType name="varNameType">
       <xs:restriction base="xs:normalizedString"/>
</xs:simpleType>
```

Druhým zásadním požadavkem na atribut *name*, je požadavek unikátnosti. Ten je definovaný pomocí konstrukce založené na XPath (jazyk určený pro odkazování v rámci XML dokumentu (W3C, 2010)) výrazu v rámci definice kořenového elementu ve schématu. Definujeme, že všechny atributy *name* obsažené kdekoli v elementu *content*, musí být jedinečné. To nám umožní jednoznačné identifikování jednotlivých proměnných.

```
<xs:element name="task" type="taskType">
       <xs:unique name="uniqueAttributeNameForVariables">
            <xs:selector xpath="./content/*"/>
            <xs:field xpath="@name"/>
       </xs:unique>
</xs:element>
```

Atribut *type* potom říká, jakého typu generovaná proměnná musí být. Název přitom vychází z názvu třídy, která daný datový typ implementuje. Povolený obsah lze proto ve schématu explicitně definovat restrikcí jako výčtový typ. Z toho vyplývá, že při rozšiřování systému o nový datový typ je nutné přidat položku s jeho názvem do schématu.

```
<xs:simpleType name="varTypeType">
        <xs:restriction base="xs:normalizedString">
            <xs:enumeration value="Number" />
            <xs:enumeration value="Text" />
             <!-- zde budeme přidávat nové položky -->
        </xs:restriction>
</xs:simpleType>
```

Protože název třídy může být v některých případech příliš dlouhý a psaní takových názvů ručně zvláště nepohodlné, rozhodli jsme se zavést systém zástupných jmen, který by tento problém eliminoval. Detailní popis systému zástupných jmen bude popsán v samostatné podkapitole kapitoly 8.

Kořenový element vstupní šablony se jmenuje *task* a jeho jediným povoleným obsahem jsou elementy *title*, *content* a *comment* v takto striktně daném pořadí. Element má jeden povinný atribut *identifier*, který slouží pro jedinečnou identifikaci úlohy. Tento identifikátor je shodný s atributem *identifier* ve výstupním QTI dokumentu.

Prvním elementem, jenž je obsahem kořenového elementu, je element *title*, který obsahuje krátký popisný titulek úlohy. Tento titulek je 1:1 přenášen do výstupního QTI dokumentu jako stejnojmenný atribut kořenového elementu.

```
<xs:simpleType name="titleType">
       <xs:restriction base="xs:string" />
</xs:simpleType>
```

Element *content* slouží pro uchování kompletního znění zadání úlohy, jedná se tudíž o stěžejní část dokumentu. Obsahuje veškerý obsah, který chceme nakonec zobrazit, smíchaný s elementy *vi* a *vo*, na jejichž místech buďto vygenerujeme určité hodnoty nebo na těchto místech očekáváme interakci uživatele.

```
<xs:complexType name="contentType" mixed="true">
        <xs:choice minOccurs="0" maxOccurs="unbounded">
            <xs:element name="vi" type="viType"/>
            <xs:element name="vo" type="voType"/>
        </xs:choice>
</xs:complexType>
```

Tento způsob zadávání úlohy nám zaručí maximální univerzálnost, nejsme totiž ničím omezeni. Potřebujeme-li zavést novou funkcionalitu, jednoduše definujeme nový datový typ, který poté v zadání použijeme.

Poslední prvek struktury je *comment*. Sem může autor volitelně vepsat další informace k úloze, například často opakované chyby. Povolený obsah je libovolný řetězec bez žádných dalších restrikcí. Definice ve schématu je shodná s definicí elementu *title*.

Příklad zadání úlohy pro výpočet součtu nebo rozdílu dvou čísel:

```
<?xml version="1.0" encoding="UTF-8"?>
<task identifier="scitani_odcitani">
    <title>Sčítání a odčítání dvou čísel</title>
    <content>
       Spočítejte výsledek
       <vi name="a" type="Number"/>
       <vi name="operator" type="Text"/>
       <vi name="b" type="Number"/> =
       <vo name="vysledek" type="Number" />
    </content>
    <comment>
        Úloha testuje základní početní úkony.
    </comment>
</task>
```

Výstupem této kapitoly je formát XML dokumentu sloužícího jako vstup systému a XSD schéma (inputDocumentSchema.xsd), na základě kterého vstupní dokumenty vytváříme. Kompletní znění XSD schématu je uvedeno v příloze A.

7 Zpracování vstupních dat

Před tím, než začneme vytvářet systém, který přetvoří vstupní dokument ve výstupní, je nutné se rozhodnout, v jakém programovacím jazyce systém implementujeme. V kapitole programovací jazyky byla tato problematika vysvětlena a došli jsme k závěru, že pro náš systém nejlépe poslouží procedurální, objektově orientovaný programovací jazyk Java. Důvodem je vynikající spolupráce s XML, nezávislost na platformě a v neposlední řadě také skutečnost, že Java je v současné době jedním z nejpoužívanějších programovacích jazyků.

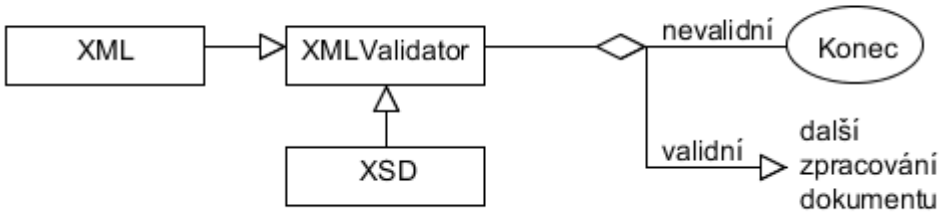
V následujících podkapitolách bude popsán postup zpracování vstupního dokumentu v jednotlivých logických krocích.

7.1 Validace

Při zpracování vstupního XML dokumentu je nutné tento dokument nejprve validovat oproti schématu. Tato nutnost vyplývá z faktu, že dokument vytváří člověk, a je proto nutné eliminovat chybovost lidského faktoru. Validace by měla být rychlá, proto je vhodné zvolit proudový parser. K tomuto účelu se proto nejvíce hodí parser SAX, kterému pomocí metody `setValidating(true)` řekneme, že má validovat. Dále musíme poskytnout cestu k souboru se schématem a také cestu k vlastnímu vstupnímu dokumentu.

Pro potřeby validace nemusíme pro SAX psát reakce na události, takzvaný *ContentHandler*, jelikož chceme pouze validovat, nikoli reagovat na události. Co však musíme parseru dodat, je způsob reakce na chyby. To zajistíme pomocí metody `setErrorHandler()`, kde je jako argument očekáván objekt implementující rozhraní *ErrorHandler*. Tento objekt se v naší implementaci nazývá *ReportValidationError* a zajistí, že v případě nesouladu se schématem bude vyhozena výjimka s přesnou informací o tom, kde v dokumentu tento problém vznikl. Tuto výjimku však neošetřujeme přímo v tomto objektu, ale delegujeme jí o úroveň výše. To umožní nadřazené třídě pracovat jednotně se všemi výjimkami, které při zpracování vzniknou, a zastavit běh zpracování. Kompletní znění třídy *ReportValidationError* je obsaženo v příloze B.

Celá implementace validátoru v systému je realizována třídou *XMLValidator*, které při vytváření instance předáme cesty k dokumentu a schématu. Kód validátoru je uveden v příloze C.



Obr č. 4: Proces validace (Zdroj: vlastní zpracování, 2012)

7.2 Načtení dokumentu

Načtení vstupního dokumentu do systému je prováděno třídou *InputParser*, které při vytváření instance předáme cestu k požadovanému dokumentu. Uvnitř třídy je napevno uvedena cesta ke schématu, což je možné z toho důvodu, že schéma je pouze jedno a cesta k němu se nemění.

Nejprve je vytvořen objekt výše popsaného validátoru, což nám zajistí soulad se schématem. Když víme, že je dokument validní, můžeme přistoupit k jeho zpracování.

Naším cílem je nahrát celý dokument do paměti ve formě objektu, se kterým můžeme později jednoduše pracovat. Tento objekt se jmenuje *TaskInput* a obsahuje atributy *identifier, title, content,* a *comment.* Všechny tyto atributy jsou typu řetězec. Jelikož v celém systému využíváme vlastnost OOP zapouzdření, všechny atributy jsou chráněné (private). Objekt proto také obsahuje metody pro získání hodnot atributů, takzvané gettry. Inicializace proměnných probíhá v konstruktoru, a jelikož data nechceme později měnit, neobsahuje třída z bezpečnostních důvodů žádnou metodu pro přenastavení hodnot atributů. Takové metody se nazývají settry.

Data chceme z dokumentu pouze přečíst a vytvořit instanci třídy *TaskInput,* reprezentující vstupní dokument. Zde se opět vyplatí použít proudový parser, v tomto případě konkrétně StAX. Výhody použití této technologie jsou popsány v kapitole XML parsery. Hlavní výhoda oproti technologii SAX spočívá v mnohem efektivnějším způsobu zápisu programu.

Načtení všech atributů, kromě atributu *content*, nám nečiní žádný problém, pouze převezmeme řetězce, které nám poskytne StAX. Element *content* je ale problematický, jelikož obsahuje řetězec kombinovaný s elementy *vi* a *vo.* Musíme proto definovat

způsob, jakým z elementu *content,* vytvořit řetězec, který uložíme do atributu *content* třídy *TaskInput.*

Načtení elementu *content* provádíme v metodě `contentProcessing()`, jejímž výstupem je požadovaný řetězec. V metodě postupně procházíme obsah elementu a narazíme-li na sekvenci znaků, uložíme ji do výstupního řetězce. Narazíme-li na element *vi* nebo *vo*, vezmeme jeho jméno z atributu *name* a přidáme prefix ### a sufix ##. Takto upravený řetězec přidáme do výstupního řetězce.

Při zpracování elementů *vi* a *vo* pouze výše uvedeným způsobem bychom přišli nejen o informaci o typu proměnné z atributu *type,* ale také informaci o stavu proměnné (vstupní / výstupní). To si samozřejmě nemůžeme dovolit, a proto při zpracování těchto elementů vytvoříme ještě tříprvkové pole, obsahující jméno, typ a stav proměnné. Výsledné pole uložíme do seznamu (kolekce programovacího jazyka Java - List) pojmenovaného *checkingList.*

Třída *InputParser* tak poskytuje dva objekty, *TaskInput* pojmenovaný *uloha* a `List<String[]>` (seznam polí řetězců), pojmenovaný *checkingList.* Tyto objekty se automaticky vytvoří při vytvoření instance *InputParser* a získáme je pomocí metod `getUloha()` a `getCheckingList()`.

Kompletní znění tříd *InputParser* a *TaskInput* je uvedeno v přílohách D a E.

8 Návrh struktury generovaných vstupních a výstupních dat

V objektu typu *TaskInput* máme nyní definováno, jaké informace od systému očekáváme. Dále je proto třeba vyřešit problém, jakým způsobem data generovat a v jaké struktuře je následně ukládat.

Již ve vstupním dokumentu máme definován datový typ a název proměnné, kdy název je vždy unikání, což je zaručeno pomocí XSD schématu. Této vlastnosti využijeme při výběru datové struktury pro uchování generovaných dat. Nejprve je však třeba navrhnout strukturu pro uchování jednotlivých proměnných. K tomuto účelu jsme se rozhodli plně využít výhod objektově orientovaného programování a jednotlivé proměnné reprezentovat pomocí zvláštních objektů.

8.1 Reprezentace datových typů proměnných

Pro všechny proměnné, s kterými bude generátor úloh pracovat, je nutné vytvořit v balíku *modules* třídu. Balík *modules* slouží jako jednotné úložiště pro ukládání všech typů proměnných, se kterými chceme pracovat a základní třídou v tomto balíku je abstraktní třída *TaskVarContainer*. Tato třída je rodičem všech tříd v tomto balíku obsažených a její kompletní znění je uvedeno v příloze F.

TaskVarContainer je abstraktní třída, protože obsahuje abstraktní metody. Tyto metody nemají tělo a jejich implementace je vynucena na potomcích. Další vlastností abstraktní třídy je nemožnost vytvoření instance. Třída však může obsahovat i normální metody a atributy a v tom se liší od rozhraní, které by bylo vhodnější použít v případě, že bychom nechtěli v rodiči uchovávat žádné atributy.

V našem případě ale chceme v rodiči uchovávat ty údaje, které jsou pro všechny námi vytvořené datové typy stejné, a to řetězce jméno a stav. Stavem se rozumí skutečnost, zda je proměnná vstupní nebo výstupní. Ve třídě *TaskVarContainer* jsou dále příslušné metody pro nastavení a získání těchto atributů z vnějšku balíku *modules*. V potomcích třídy ale chceme umožnit přímý přístup k těmto atributům, čehož docílíme záměnou specifikátoru přístupu private za protected. Princip zapouzdření zůstane zachován a potomci mají volný přístup k vlastním datům.

```
protected String name;
```

Poslední a nejdůležitější částí třídy *TaskVarContainer,* jsou tři abstraktní metody `printResponseDeclaration()`, `printItemBody()` a `printResponse`

`Condition()`, které nutí potomky implementovat způsob, jakým je daný datový prvek možné zobrazit pomocí standardu QTI uvnitř příslušných elementů.

```
public abstract Object printItemBody(ObjectFactory of);
```

Při vytváření nových datových typů postupujeme takto. Nejprve navrhneme jméno a funkcionalitu třídy, která bude reprezentovat nový datový typ. Podmínka pro její použití v systému je, jak již bylo řečeno, nutnost být potomkem abstraktní třídy *TaskVarContainer* a musí obsahovat explicitní konstruktor pro vytvoření instance. Dalším nutným krokem je proto tvorba explicitního konstruktoru a implementace výše zmíněných abstraktních metod.

Jako příklad definování nových typů proměnných uvedeme postup, jaký byl použit při vytváření v systému již implementovaného typu Text. Nejprve definujeme, co by měla proměnná obsahovat. Zcela jistě musí obsahovat atribut typu String, nesoucí textovou informaci, kterou chceme uložit. V našem případě se tato proměnná jmenuje *value*.

```
private String value;
```

Dále musíme implementovat všechny abstraktní metody abstraktní třídy *TaskVarContainer*, toho docílíme klíčovým slovem *extends* v hlavičce třídy. Používáme-li pokročilejší IDE (Integrated Development Environment), doplní se skelety potřebných metod automaticky. V opačném případě je musíme okopírovat nebo opsat ze souboru *modules.TaskVarContainer.java*. Obsah těchto metod bude objasněn v kapitole 9, pojednávající o tisku výstupního XML dokumentu ve standardu QTI.

```
public class Text extends TaskVarContainer {}
```

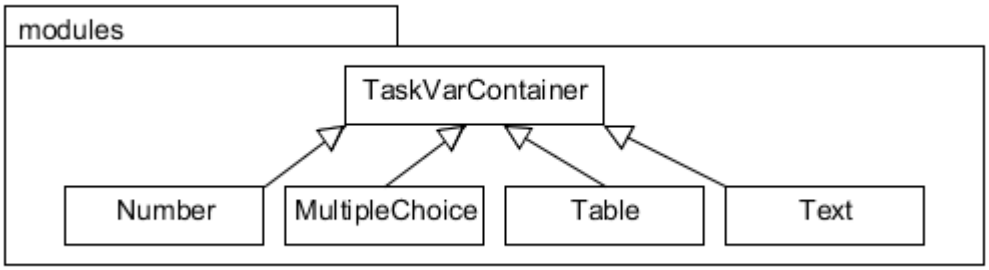
Poté definujeme konstruktor, který inicializuje všechny atributy třídy, v tomto případě jen atribut *value*.

```
public Text(String value) {this.value = value;}
```

Nakonec ještě nesmíme zapomenout zavést název nové proměnné do XSD schématu způsobem popsaným v kapitole 6. Tím máme novou proměnnou připravenou k použití. Plné znění třídy *Text* je uvedeno v příloze G.

Tento způsob tvorby nových datových typů přináší téměř neomezené možnosti v rozšiřování funkcionality systému. Je možné například vytvářet datové typy složené ze stávajících datových typů. Příkladem tohoto skládání je datový typ *Table* uvedený v příloze J, kde může být v jednotlivých buňkách tabulky libovolný objekt, který je potomkem *TaskVarContainer*. V systému jsou navíc již implementovány datové typy

Number (příloha H), reprezentující číselnou hodnotu a MultipleChoice (příloha I), reprezentující testovou otázku s výběrem odpovědí, tzv. multiple choice.



Obr. č. 5: Balík modules (Zdroj: vlastní zpracování, 2012)

8.2 Kontejner pro uchování generovaných dat

Jednotlivé proměnné, které budou v systému vytvářeny na základě vstupního XML dokumentu, je nutné uložit do nějakého centrálního úložiště. K tomuto účelu se nejlépe hodí datová struktura asociativní pole, která umožní uživatelsky přívětivé vkládání a vybírání proměnných na základě klíče.

V programovacím jazyce Java je asociativní pole implementováno kolekcí, která se nazývá mapa (java.util.HashMap), protože mapuje klíče na hodnoty. Předpokladem pro uložení hodnoty do mapy je unikátnost klíče, která je zajištěna v XSD schématu vstupního dokumentu. Do mapy vkládáme uspořádanou dvojici klíč-hodnota, kde klíčem je jméno proměnné (řetězec) a hodnotou je objekt typu *TaskVarContainer*.

```
protected Map<String, TaskVarContainer> kontejner =
new HashMap<String, TaskVarContainer>();
```

Tento kontejner je obsažen uvnitř abstraktní třídy *AbstractSolver* (Příloha K). Jedná se o rodiče všech objektů, které budou generovat data pro jednotlivé úlohy. Metoda `getKontejner()` slouží k získání již naplněného kontejneru pro další zpracování. Abstraktní metody `generator()` a `solver()` vynutí implementaci generátoru vstupních dat a řešitele výstupních dat v konkrétním objektu generátoru.

```
protected abstract void generator();
```

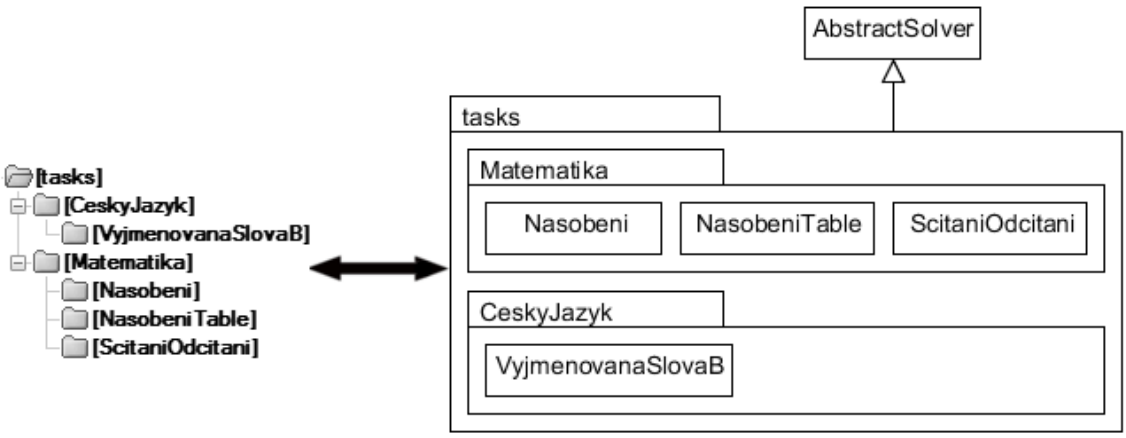
Volání těchto metod již v konstruktoru třídy *AbstractSolver* nám zajistí jejich provedení ve správném pořadí a odstraní nutnost vytvářet konstruktor u potomků, zvláště když tento konstruktor pouze volá dvě metody ve správném pořadí a pro všechny potomky by byl stejný.

8.3 Banka příkladů

Pro implementaci jádra systému bylo nutné definovat způsob, jakým provázat vstupní XML dokumenty s příslušnými generátory v podobě tříd programovacího jazyka Java. Proto byl navržen systém provázání, který na základě zvoleného vstupního dokumentu automaticky zvolí příslušný generátor.

K tomuto provázání byla použita vlastnosti všech operačních systémů - souborový systém a vlastnost programovacího jazyka Java - možnost hierarchického umísťování tříd do balíků a podbalíků. Provázání poté spočívá v zrcadlení systému složek a souborů obsahujících vstupní XML dokumenty se systémem balíků uvnitř programu. Kořenová složka pro ukládání vstupních souborů se nazývá *tasks* a může obsahovat libovolné množství dalších složek a podsložek. Tuto strukturu poté kopíruje systém balíků s kořenovým balíkem *tasks.* Složky jsou zde reprezentovány jako balíky, vstupní XML dokumenty jako třídy programovacího jazyka. Vstupní soubor, který operuje s příslušným generátorem, se musí nacházet ve složce se stejným jménem jako generátor.

V případě, že zachováme výše uvedené podmínky, systém sám najde a vytvoří příslušný generátor, v opačném případě skončí proces generování výjimkou. Pro předvedení funkčnosti generátoru jsou nyní v systému implementovány čtyři ukázkové příklady se čtyřmi generátory ve dvouúrovňové hierarchii. První úroveň tvoří název předmětu, ke kterému se příklad váže, druhá úroveň je konkrétní název úlohy svázané se stejnojmenným generátorem. Vstupní dokument má stejný název jako nadřazená složka, například *Nasobeni.xml,* to však není nutné a slouží to pouze k zachování přehlednosti. Celý systém je znázorněn na následujícím obrázku.



Obr. č. 6: Provázání banky příkladů s generátory (Zdroj: vlastní zpracování, 2012)

8.4 Generátory dat

Jak bylo řečeno v podkapitole pojednávající o kontejneru pro uchování generovaných dat, všechny generátory dat musí být potomkem abstraktní třídy *AbstractSolver*. To nutí tyto generátory implementovat dvě metody, `solver()` a `generator()`. Děděním však také získáme přímý přístup k definovanému kontejneru, kam můžeme přímo ukládat generovaná data.

Naším úkolem je v metodách `solver()` a `generator()` naplnit tento kontejner všemi potřebnými daty, jež jsou očekávány dle specifikace ve vstupním dokumentu. K tomu můžeme použít obě nebo jenom jednu nabízenou metodu. Dvě metody použijeme v případě, že lze striktně oddělit automaticky generované proměnné (`generator()` - generátor) a proměnné na základě generovaných dat vypočtené (`solver()` - řešitel), toto oddělení je však zavedeno pouze z estetických důvodů, pro zpřehlednění zdrojového kódu. Pro naplnění kontejneru je možné použít pouze jednu libovolnou metodu, musíme však mít na paměti, že nejprve bude vždy spuštěn generátor a až poté řešitel.

V systému jsou nyní, jak je vidět na obrázku z předchozí kapitoly, implementovány čtyři generátory. Jako příklad vytváření a použití generátoru použijeme generátor úlohy *ScitaniOdcitani*, který je uveden v příloze N a jejíž vstupní XML dokument byl uveden v kapitole o návrhu vstupního XML dokumentu.

Vstupní soubor zjednodušeně definuje znění úlohy takto

```
a operator b = vysledek
```

kde *a*, *b* a *vysledek* jsou typu *Number*, *operator* typu *Text.*

Vidíme, že po vytvoření instance objektu generátoru *ScitaniOdcitani* musí náš kontejner obsahovat čtyři proměnné, z toho tři náhodně generované a jednu vypočtenou. Nejprve vytvoříme třídu s názvem *ScitaniOdcitani*, která dědí od *AbstractSolver.*

```
class ScitaniOdcitani extends AbstractSolver {}
```

Poté implementujeme metodu `generator()`, která náhodně vygeneruje dvě čísla pro potřeby proměnných *a* a *b*. Na základě třetího náhodného čísla rozhodneme, zda operátor bude + nebo -. Jakmile máme připraveny tyto informace, můžeme je vložit do našeho kontejneru. To provedeme pomocí metody `put(klíč,hodnota)`, která je již implementována v JDK. Vkládání proměnné *operator*, která je dočasně uložena ve stejnojmenné proměnné typu *String* uvnitř metody, vypadá v našem příkladě takto

```
kontejner.put("operator", new Text(operator));
```

Na tomto příkladu vkládání je vidět význam nutnosti existence explicitního konstruktoru v námi definovaných datových typech, což bylo definováno v kapitole o reprezentaci datových typů proměnných. Kdyby datový typ konstruktor neměl, museli bychom složitě přiřazovat všechny parametry ručně, v tomto případě však přehledně zavoláme příslušný konstruktor přímo v metodě `put()` a všechna data vložíme do proměnné na jednom místě. Takto unifikovaný systém vkládání proměnných má význam zejména při ladění nových generátorů a hledání chyb.

Nakonec ještě implementujeme metodu `solver()`, ve které na základě vygenerovaných dat vypočítáme výsledek a ten opět vložíme do kontejneru.

A to je vše, co od objektu generátoru očekáváme, máme tak obrovskou volnost, jakým způsobem zajistíme, aby kontejner potřebná data obsahoval. Další implementované generátory jsou uvedeny v přílohách L, M a O.

Tento způsob tvorby generátorů má své výhody i nevýhody. Hlavní nevýhoda je, že může nastat situace, kdy zapomeneme některou proměnnou do kontejneru vložit, nebo ji vložíme, ale jako špatný datový typ. Tento problém je v systému podchycen, takže je možné problém opravit. Výhoda naopak spočívá v obrovské volnosti, jakým způsobem data generovat. Můžeme si například pro vkládání některých specifických dat vytvořit externí preprocesor, který budeme uvnitř příslušného generátoru používat a výsledná data vkládat do kontejneru. Systém tak splňuje vlastnost jednoduché rozšiřitelnosti o zcela nové typy úloh.

8.5 Systém zástupných jmen

V kapitole 6 bylo řečeno, že název třídy, která reprezentuje datový typ proměnné, může být v některých případech příliš dlouhý, což znesnadňuje ruční vytváření vstupních dokumentů. Pro zachování pořádku a přehlednosti v datových typech je ale žádoucí, aby název plně vystihoval povahu proměnné. Pro zjednodušení tohoto stavu jsme se proto rozhodli zavést systém zástupných jmen, neboli aliasů, který tento problém eliminuje.

Řešení spočívá v zavedení uživatelem definovaných aliasů ke každému nově vytvořenému datovému typu. Tyto aliasy jsou poté použity ve vstupním dokumentu namísto plných jmen příslušných tříd a systém je následně schopen spárovat očekávaný typ proměnné s definovaným aliasem.

Zástupná jména definujeme v konfiguračním XML dokumentu, který je pro současnou implementaci zobrazen v příloze R, spolu s XSD schématem, který tento konfigurační soubor upravuje.

Soubor obsahuje čtyři druhy elementů. Kořenový element *typeAliases* může obsahovat nekonečně mnoho elementů *def*, jež reprezentují právě jednu definici zástupného jména. Definice se skládá z elementů *module* a *alias* v uvedeném pořadí a obsahem těchto elementů je *normalizedString*, takže před a za textem nesmí být bíle znaky. Element *module* obsahuje název datového typu z balíku *modules*, pro který alias definujeme, *alias* potom představuje jméno, pod kterým bude příslušný modul ve vstupním dokumentu uveden.

```
<typeAliases>
    <def>
        <module>Number</module>
        <alias>num</alias>
    </def>
    ...
</typeAliases>
```

Po zavedení tohoto zástupného jména je nutné jej zařadit do XSD schématu vstupního dokumentu do seznamu povolených jmen, případně originální jméno smazat. Ve schématu bude poté namísto

```
<xs:enumeration value="Number" />
```

tento záznam

```
<xs:enumeration value="num" />.
```

Při vytváření vstupních dokumentů budeme do atributu *type* psát namísto *Number* pouze *num*, takže se vstupní dokument zjednoduší a místo

```
<vi name="a" type="Number"/>
```

bude obsahovat

```
<vi name="a" type="num"/>.
```

Při vytváření nových datových typů proto ideálně postupujeme takto:

1. Vytvoříme nový datový typ
2. Zaneseme zástupné jméno do konfiguračního souboru
3. Zaneseme zástupné jméno do XSD schématu vstupního dokumentu

Tento systém přináší značné zjednodušení u datových typů s velice dlouhým názvem, v systému bychom například chtěli mít více druhů multiple choice proměnných a jeden z nich by se mohl jmenovat *MultipleChoiceNumber*, tento název plně vystihuje, o jakou

proměnnou se jedná, při ručním psaní ale raději použijeme alias, který může být například *mcnum*. Zjednodušení ale přináší i u kratších názvů typů proměnných, jak je vidět na výše uvedeném příkladě u typu *Number*.

8.5.1 Načtení konfigurace zástupných jmen

Konfiguraci zástupných jmen je potřeba nějakým způsobem načíst do systému. Postupujeme podobným způsobem jako při načítání vstupního XML dokumentu. Z toho důvodu použijeme pro parsování konfiguračního souboru opět technologii StAX. Umístění konfiguračního souboru a příslušného schématu je pevné, proto ho zde nemusíme získávat zvenčí. Pevný však není obsah tohoto souboru a proto jej před parsováním validujeme stejným validačním postupem, jako je uvedeno v kapitole 7. Uspořádané dvojice alias - modul (datový typ) ukládáme do kolekce mapa, kde alias je klíčem a modul je hodnotou. Produktem tohoto systému je mapa, obsahující všechna zástupná jména, která jsou v systému použita.

Implementace tohoto systému se nachází ve třídě *AliasParser* popsané v příloze Q. Tato třída obsahuje kromě konstruktoru, který provede načtení mapy, pouze metodu `getAlias()`, jež vrátí hotovou mapu pro další zpracování.

8.6 Kontrola generovaných dat

Generovaná data, která jsou uložena v námi definovaném kontejneru, mohou být z nějakého důvodu chybná, nebo mohou zcela chybět. To může být způsobeno mnoha faktory a největší nebezpečí spočívá v chybě způsobené faktorem lidským, jelikož kontejner generovaných dat plní sám uživatel při implementaci příslušného generátoru. Lehce se proto může stát, že některou proměnnou vygenerujeme, ale zapomeneme uložit do kontejneru. Systém ale očekává všechna data, která jsou uvedena ve vstupním dokumentu, a při jejich neexistenci by selhal. Abychom této situaci zabránily, vytvořili jsme systém pro kontrolu generovaných dat, který zkontroluje, zda generátor naplnil kontejner minimálně těmi daty, která jsou uvedena ve vstupním dokumentu.

Tento systém pracuje se třemi kolekcemi, které jsme si v průběhu zpracování vytvořili. První kolekce se nazývá *checkingList* a jedná se o seznam polí řetězců, který nám poskytne *InputParser*. Pole v tomto seznamu nám podávají informace o očekávaných datech ze vstupního dokumentu a způsob jejich vytváření je popsán v kapitole 7. Další kolekce je mapa zástupných jmen *aliasesMap*, kterou nám poskytne *AliasParser* a

slouží pro párování skutečných datových typů s jejich zástupnými jmény. Poslední kolekcí je naplněný kontejner generovaných dat, nazvaný *kontejner*.

Implementace kontroly dat je provedena ve třídě *Checker*, jejíž plné znění je uvedeno v příloze P. Této třídě musíme při vytváření instance předat všechny tři kontejnery a poté se okamžitě provede automatická kontrola. V případě úspěchu se nic nestane a program může pokračovat, v opačném případě *Checker* vyhodí výjimku s popisem chyby a její ošetření není opět realizováno uvnitř třídy, ale je delegována o úroveň výše pro zpracování v nadřazené aplikaci.

Princip kontroly je následující, systém postupně prochází první kolekci *checkingList*, která obsahuje jméno, typ a stav proměnných definovaných ve vstupním dokumentu. Nejdříve je provedena kontrola existence této proměnné v generovaném kontejneru. To zjistíme jednoduše tak, že se pokusíme z mapy *kontejner* získat proměnnou se stejným jménem a pokud se nám to nepodaří, takto pojmenovaná proměnná v kontejneru není a okamžitě ukončujeme kontrolu s vyhozením výjimky.

Druhým krokem je kontrola typu proměnné, kde porovnáváme očekávaný typ se skutečným typem. Typ uložený v seznamu *checkingList* je však ve skutečnosti pouze alias, a proto musíme nejdříve zjistit skutečné jméno datového typu výběrem z mapy *aliasesMap*. Toto jméno musí souhlasit se jménem typu proměnné, kterou jsme v předchozím kroku vybrali z mapy *kontejner* a pokud tomu tak není, opět vyhazujeme výjimku.

Posledním krokem je automatické nastavení jména a stavu proměnné uvnitř mapy *kontejner*. Toto nastavení provádíme až zde, protože máme jedinečnou příležitost projít všechny proměnné uvnitř kontejneru a nastavení provést. Jinak by nastavení musel provádět uživatel při vytváření generátoru, což je zbytečné, když tyto informace již zadával do vstupního dokumentu.

Po provedení kontroly máme jistotu, že systém obsahuje všechna data definovaná ve vstupním dokumentu a můžeme přistoupit k poslednímu kroku zpracování, jímž je tisk do výstupního XML dokumentu.

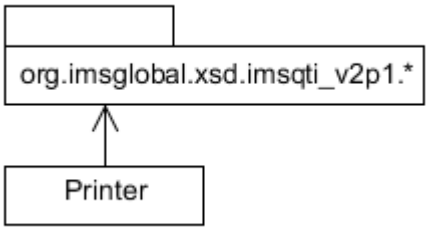
9 Tisk úlohy do výstupního QTI dokumentu

Posledním krokem zpracování je tisk dat ze vstupního XML dokumentu společně s generovanými daty do výstupního QTI dokumentu. Tisk je implementován ve třídě *Printer* uvedené v příloze S. Veškeré informace, které pro tisk potřebujeme, máme nyní uloženy v paměti počítače. Údaje ze vstupního XML dokumentu jsou uloženy v objektu třídy *TaskInput* a generovaná data v kolekci mapa. Tyto informace musíme tiskárně předat při vytváření instance, a předáme jí také požadované jméno výstupního souboru. Po vytvoření instance začne okamžitě proces tisku zavoláním metody `printDocument()`.

Než začneme data nějakým způsobem tisknout, to znamená vytvářet XML dokument, musíme se rozhodnout, jakou technologii pro vytváření použijeme. Víme, že budeme skládat velice složitý XML dokument, který navíc musí odpovídat normě QTI. Z těchto důvodů je nejvhodnější použít technologii JAXB, která dovolí intuitivně sestavovat dokument v podobě objektů a navíc neumožní sestavit nevalidní dokument.

Technologii JAXB je věnována samostatná podkapitola v kapitole o XML parserech, kde je uveden i způsob použití této technologie.

Podstata technologie spočívá v přetvoření XSD schématu pomocí nástroje *xjc* na balík tříd pomocí kterých pohodlně sestavíme a vytiskneme dokument, který bezpečně odpovídá zadanému schématu. Pro vytvoření sady tříd bylo použito XSD schéma poskytované IMS GLC, a proto je zajištěna kompatibilita se standardem QTI. Balík vygenerovaných tříd *org.imsglobal.xsd.imsqti_v2p1* pouze vložíme do našeho systému a můžeme s nimi libovolně pracovat.



Obr. č. 7: Použití balíku tříd standardu QTI v systému (Zdroj: vlastní zpracování, 2012)

Struktura dokumentu QTI je uvedena v samostatné kapitole o tomto standardu. V této kapitole jsou definovány tři klíčové části, které v dokumentu potřebujeme a to sadu

elementů *responseDeclaration* a elementy *itemBody* a *responseProcessing.* Pro sestavení těchto elementů obsahuje třída *Printer* speciální metody

```
private List<ResponseDeclarationType>
  createResponseDeclarations(){}
private ItemBodyType createItemBody(){}
private ResponseProcessingType createResponseProcessing(){}.
```

V těchto metodách se předpřipraví obsah, který se následně vytiskne. Vidíme, že technologie JAXB připravila třídy, které se jmenují stejně jako příslušné elementy, ale na konci mají slovo type. Pro vytvoření elementu *itemBody* tak potřebujeme objekt třídy *ItemBodyType*, který pomocí automaticky vygenerovaných metod nastavíme. Tvorba těchto objektů však neprobíhá klasicky pomocí klíčového slova `new`, ale na základě návrhového vzoru Factory (Továrna). Tento systém funguje tak, že instance objektů vytváříme pomocí speciální třídy - továrny, která se zde jmenuje *ObjectFactory*. Instanci této třídy jako jedinou vytvoříme pomocí klíčového slova `new`.

```
ObjectFactory of = new ObjectFactory();
```

Pro vytváření celé struktury dokumentu potom musíme využívat tento objekt.

Princip tvorby výstupního dokumentu je následující. Nejprve je z konstruktoru zavolána metoda `printDocument()`, kde vytvoříme a nastavíme marshaller, který slouží pro zápis vytvořených objektů do XML dokumentu. Marshalleru musíme předat objekt kořenového elementu a výstupní proud dat do souboru, jehož název byl předán při tvorbě instance třídy *Printer*.

Objekt kořenového elementu pro marshaller vytváříme pomocí metody `createDocument()`. Zde nejprve vytvoříme výše zmíněnou továrnu *ObjectFactory*, na základě které vytvoříme objekt *AssesmentItemType*. Tomu nastavíme povinné atributy získané z *TaskInput* a vložíme do nich obsah tří základních elementů pomocí volání příslušných metod. Tím máme kořenový element připraven k tisku, musíme ho ale ještě převést na objekt typu *JAXBElement*, který marshaller očekává. Takto připravený kořenový element vracíme zpět do metody `printElement()`. Marshaller má v tomto okamžiku všechny potřebné informace a provádí se tisk.

Tvorba sady elementů *responseDeclaration*, která je implementována v metodě `createResponseDeclarations()`, probíhá následujícím způsobem. Jelikož víme, že v dokumentu může být těchto deklarací více, je nutné nejdříve vytvořit kolekci, do které budeme jednotlivé deklarace ukládat. Jako nejvhodnější se pro to jeví kolekce seznam, kterou je možné přímo vložit do kořenového elementu. Plnění této kolekce

probíhá v několika krocích. Nejdříve vybereme z naplněného kontejneru generovaných dat jednu proměnnou. Potom zjistíme, zda její stav je výstupní, protože *responseDeclaration* chceme tisknout pouze pro proměnné, které bude uživatel zadávat. Jestliže tomu tak je, zavoláme metodu `printResponseDeclaration()`, jež je nutně implementována v každém datovém typu proměnné. Této metodě musíme předat továrnu *ObjectFactory* a metoda nám vrátí seznam elementů *responseDeclaration*, a to z toho důvodu, protože mohou existovat proměnné, například tabulka, které obsahují více proměnných. Konkrétní obsah deklarací je plně v režii tříd v balíku modules a závisí na povaze obsahu proměnných. Tento postup opakujeme tak dlouho, dokud neprojdeme celou mapu *kontejner*. Seznam všech deklarací nakonec vrátíme metodě `createDocument()`, která jej vloží do kořenového elementu.

Naprosto stejným způsobem probíhá tvorba elementu *responseProcessing* v metodě `createResponseProcessing()`. Jediná odlišnost spočívá v tom, že tento element je v dokumentu pouze jeden, obsahuje však sadu elementů *responseCondition* a tyto elementy jsou již tvořeny stejně jako elementy *responseDeclaration.* Elementy *responseCondition* tak tvoříme v metodě `printResponseCondition()`, uvnitř proměnných, opět na základě povahy uchovávaných dat.

Tvorba posledního elementu *itemBody* je nejsložitější a probíhá v metodě `createItemBody()`. Cílem je do textu, uvedeného v elementu *content* vstupního XML dokumentu, vložit generovaná data na místa deklarovaná elementy *vi* a *vo*. V objektu *InputParser* jsme již předpřipravili strukturu, kterou jsme vložili do objektu *TaskInput* jako atribut *content*. Tento předpřipravený řetězec načteme do proměnné *rawContent*, a přetvoříme ho na obsah elementu *itemBody*, a to pomocí nahrazování bloků *###název##* za příslušné hodnoty. Proces nahrazování probíhá dvoufázově. V první fázi procházíme kontejner generovaných dat a nad jednotlivými objekty voláme metodu `printItemBody()`. Od této metody očekáváme, že v případě generovaných vstupních dat, které se mají otisknout do zadání, vrátí metoda řetězec a v opačném případě vrátí element, pomocí kterého probíhá dle standardu QTI zadávání odpovědi. Podle vrácených dat tak rozhodujeme, co uděláme. V případě vrácení řetězce jím přímo nahradíme blok pro vložení proměnné. Objekty pro zadávání dat uživatelem si ukládáme do mapy název - objekt pro druhou fázi zpracování a blok pro nahrazování v řetězci *rawContent* ponecháme.

Nyní máme k dispozici kompletní textovou informaci zadání úlohy a v druhé fázi zpracování pouze rozdělíme tento text za pomoci znaků ## na jednotlivé části do pole. Poté toto pole procházíme a začíná-li hodnota jedním znakem # který nám zbyl u nahrazovacích bloků, vložíme příslušný objekt z mapy vytvořené v první fázi do elementu *itemBody*. Nezačíná-li záznam v poli znakem #, jedná se o již hotový text, který již přímo vložíme do *itemBody*.

Po provedení těchto činností máme element *itemBody* kompletně vytvořen a vracíme ho metodě `createDocument()`.

Tím je proces tisku dokončen a na zadaném místě na disku počítače je vytvořen hotový generovaný XML dokument, plně vyhovující standardu QTI. Příklad výsledku generování pro úlohu *ScitaniOdcitani* je uveden v příloze V.

10 Implementace kompletního systému

10.1 Jádro systému

Jednotlivé výše popsané komponenty systému jsou sestaveny ve třídě *TaskGenerator*, jež je uvedena v příloze T. Tato třída slouží pro generování právě jednoho výstupního souboru a je připravena k použití jako zásuvný modul do většího systému, který může umožňovat generování více úloh najednou.

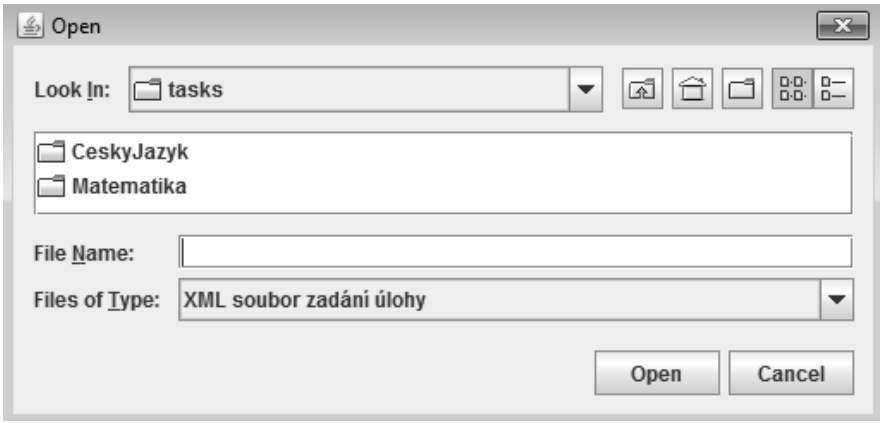
Jako parametry při vytváření objektu tohoto jádra generátoru úloh jsou očekávány vstupní XML dokument, název výstupního dokumentu a předpřipravený objekt *AliasParser*. Důvodem, proč je alias parser předáván zvenčí je skutečnost, že zástupná jména jsou pro všechny úlohy stejná a bylo by zbytečné pro každou úlohu znovu zdlouhavě načítat konfigurační XML soubor. Proto instanci třídy *AliasParser* vytvoříme v nadřazené aplikaci a objekt obsahující hotovou mapu aliasů předáme konstruktoru třídy *TaskGenerator*.

Jádro systému pak v konstruktoru postupně vytvoří *InputParser*, jež načte vstupní dokument, poté pomocí metody `getSolver()` vytvoří instanci příslušného generátoru úloh, následně se vytvoří instance třídy *Checker*, který zkontroluje obsah kontejneru s vygenerovanými daty a nakonec se vytvoří *Printer*, který vše zapíše na disk. Případné výjimky vzniklé v jednotlivých částech systému jsou opět vyhozeny o úroveň výše, aby na ně mohla reagovat nadřazená aplikace.

Metoda `getSolver()` slouží k získání správného generátoru úloh na základě cesty ke vstupnímu dokumentu. Zde je právě využit systém provázání generátorů a vstupních souborů definovaný v kapitole 8. Systém nejprve z uvedené cesty rekonstruuje umístění úlohy ve složce *tasks* a z těchto informací sestaví předpokládané umístění v příslušném balíku s generátory. Poté se pokusí vytvořit instanci objektu nacházejícího se na tomto umístění a v případě úspěchu je tento objekt vrácen konstruktoru pro další použití. Pokud se generátor na daném umístění nenachází, není zachován systém provázání a metoda vyhazuje výjimku.

10.2 Testovací aplikace

Pro ověření funkčnosti celého systému byla vytvořena testovací aplikace s jednoduchým grafickým uživatelským rozhraním, která umožňuje pohodlně vybrat úlohu z banky a provést generování. Implementace této aplikace se nachází ve třídě *Application* (Příloha U). Uvnitř této třídy je nejprve vytvořena instance objektu *AliasParser* pro načtení konfiguračního souboru systému aliasů do mapy aliasů. Následně je pomocí metody `createJFileChooser()` vytvořen nástroj pro výběr vstupního dokumentu z banky úloh, který je poté zobrazen uživateli.



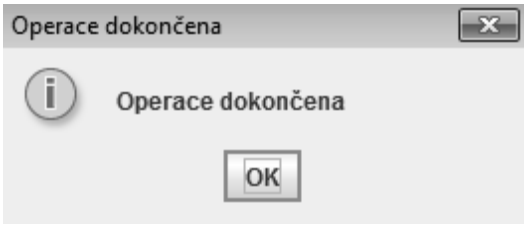
Obr. č. 8: Dialog pro výběr vstupního XML dokumentu (Zdroj: vlastní zpracování, 2012)

Ihned po zvolení příslušného vstupního dokumentu a stisknutí tlačítka Open se vytvoří dialogové okno informující o průběhu generování úlohy zavoláním metody `createJDialog()`. Vytvořené okno se zobrazí uživateli.

Obr. č. 9: Dialogové okno informující o průběhu zpracování (Zdroj: vlastní zpracování, 2012)

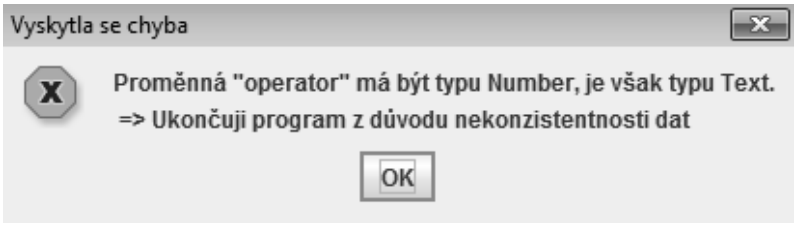
Zatímco uživatel vidí tento dialog, vytváří aplikace instanci třídy *TaskGenerator*, které předává vstupní XML soubor vybraný uživatelem při spuštění programu, objekt *AliasParser* a cestu k výstupnímu souboru, která je zde napevno nastavena na „output/vygenerovaneQTI.xml“.

Jestliže bylo generování úspěšné, zobrazí se nakonec dialog s informací o úspěšném vykonání operace a na disku počítače se ve zvoleném umístění objeví vygenerovaný QTI dokument.



Obr. č. 10: Dialog úspěšného vykonání operace generování (Zdroj: vlastní zpracování, 2012)

V případě, že se při zpracování objeví problém a bude vyhozena výjimka, zobrazí se v novém dialogovém okně, a program je ukončen.

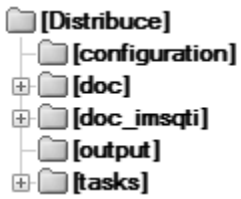


Obr. č. 11: Dialog s textem vyhozené výjimky (Zdroj: vlastní zpracování, 2012)

Celý model implementovaného systému je vyobrazen v příloze W.

10.3 Organizace distribuce

Distribuce systému je realizována pomocí nástroje Apache Ant (The Apache Ant Project, 2012), který na základě námi definovaného konfiguračního XML souboru *build.xml*, sestaví softwarový produkt do požadované funkční podoby. Stromová struktura složek distribuce je uvedena na následujícím obrázku.



Obr. č. 12: Stromová struktura složek distribuce systému (Zdroj: vlastní zpracování, 2012)

Ve složce *configuration* se nachází konfigurační soubory systému aliasů *typeAlias.xml* a *typeAliasSchema.xsd.* Složka obsahuje také XSD schéma vstupních dokumentů *inputDocumentSchema.xsd*.

Složka *doc* obsahuje dokumentaci systému ve formátu HTML, vygenerovanou nástrojem *javadoc.exe* z „javadoc“ komentářů uvedených ve zdrojových souborech. Stejný typ dokumentace je také ve složce *doc_imsqti*, zde se však jedná o dokumentaci ke třídám, které byly generovány nástrojem *xjc* z XSD schématu standardu QTI.

Do složky *output* systém generuje výstupní QTI dokument *vygenerovaneQTI.xml*. Poslední složka *tasks* je již mnohokrát zmiňovaná banka testových úloh.

Spustitelná aplikace se jmenuje *ModularTaskGenerator.jar* a je umístěna přímo ve složce *Distribuce*. Pro spuštění aplikace je, jak bylo řečeno v kapitole pojednávající o programovacím jazyce Java, nutné mít v počítači nainstalovánu platformu Java.

Složka *Distribuce* je uložena na přiloženém CD (Compact Disc). Na tomto CD dále najdeme soubor *popis.txt* informující o obsahu disku a složku *EclipseWorkspace*. V této složce je pracovní složka integrovaného vývojového nástroje Eclipse (The Eclipse Foundation, 2012), sloužící pro další vývoj systému.

11 Závěr

Cílem této práce bylo navrhnout a implementovat elementární části otevřeného počítačového systému, umožňujícího na základě vstupních dat vygenerovat jedinečné zadání úlohy, spolu s jejími výsledky. Softwarový produkt, který byl v rámci této práce vytvořen, je v souladu se všemi požadavky, které na něj byly od začátku kladeny.

Byla navržena struktura vstupního XML dokumentu zadání úlohy, která umožňuje vhodným způsobem zapsat šablonu úlohy pro zpracování vytvořeným systémem. Pro popis této struktury byl použit jazyk W3C XML Schema.

Provedli jsme návrh a implementaci struktur pro ukládání generovaných vstupních a výstupních dat, které umožňují téměř neomezené možnosti v rozšiřování funkcionality systému. Spolu s tím byly navrženy systémy pro zpracování úlohy, umožňující vlastní generování úloh do výstupního dokumentu.

Zajištění harmonizace výstupního XML dokumentu se standardem QTI bylo provedeno pomocí technologie JAXB, která nám nedovolí sestavit nevalidní dokument.

Při návrhu jednotlivých částí systému byly použity nejvhodnější dostupné technologie a celý systém je založen na multiplatformním objektově orientovaném programovacím jazyce Java.

Funkčnost systému byla otestována za pomoci testovací aplikace, která byla během práce vytvořena a jejíž popis je uveden v kapitole deset. Tato aplikace operuje s jádrem systému, které je sestaveno z dílčích systémů zpracování úlohy. Jádro systému bylo navrženo takovým způsobem, aby mohlo být použito v daleko sofistikovanější aplikaci, než v té současně implementované.

Dalším krokem vývoje by měl být návrh systému, který by umožňoval transformaci výstupního QTI dokumentu do reálně použitelných formátů jako Adobe PDF nebo HTML. Vhodné by bylo také umístění jádra generátoru do aplikace s grafickým uživatelským rozhraním, která by umožňovala jednodušší tvorbu vstupních XML dokumentů a pokročilejší možnosti generování. V této aplikaci by bylo možné vybrat počet vyhotovení jednotlivých úloh nebo sestavit celý test z jednotlivých úloh.

Posledním krokem vývoje základního systému by měla být integrace se systémem transformací, takže bychom mohli vytvořit celý test a ten rovnou transformovat do požadovaných výstupních formátů.

Když se blíže seznámíme se systémem, který jsme implementovali, zjistíme, že jej není nutné využívat pouze pro účely generování testových úloh za účelem vzdělávání. Konstrukce systému umožňuje jeho využití i v jiných aplikacích, například jako generátor testovacích dat při vývoji nových algoritmů nebo generátor událostí pro diskrétní simulace. Možností systému je mnoho a existuje tak mnoho směrů, kterými se můžeme v budoucím vývoji vydat.

12 Seznam tabulek a obrázků

13 Seznam zkratek

API - Application Programming Interface

CD - Compact Disc

DOM - Document Object Model

DTD - Document Type Definition

HTML - Hypertext Markup Language

IDE - Integrated Development Environment

IMS GLC - IMS Global Learning Consortium

JAXB - Java Architecture for XML Binding

JDK - Java Development Kit

JVM - Java Virtual Machine

OOP - Object-oriented Programming

PDF - Portable Document Format

QTI - Question & Test Interoperability

SAX - Simple API for XML

SJSXP - Sun Java Streaming XML Parser

StAX - Streaming APIs for XML

SGML - Standard Generalized Markup Language

JWSDP - Java Web Services Developer Pack

USA - United States of America

UML - Unified Modeling Language

W3C - World Wide Web Consortium

WXS - W3C XML Schema

XSD - XML Schema Definition

XML - eXtensible Markup Language

14 Seznam použité literatury

HEROUT, P. *Java a XML*. České Budějovice: Kopp, 2007. ISBN 978-80-7232-307-4

HEROUT, P. *Učebnice jazyka Java.* České Budějovice: Kopp, 2008. ISBN 978-80-7232-355-5

HEROUT, P. Počítače a programování 1. (přednáška) Plzeň: FAV ZČU v Plzni, 23.09.2009. [cit. 2012-03-01] Dostupné na www: <http://www.kiv.zcu.cz/~herout /vyuka/ppa1/portal/prednasky/ppa1-1a5-na-a4.pdf>

IMS GLC. [online] IMS Question and Test Interoperability Overview, 2005, Aktualizace 24.01.2005, [cit. 2012-03-01] Dostupné na www: <http://www.imsglobal.org/question/qti_v2p0/imsqti_oviewv2p0.html>

IMS GLC. [online] IMS Question and Test Interoperability Implementation Guide, 2006, Aktualizace 08.06.2006, [cit. 2012-03-01] Dostupné na www: <http://www.imsglobal.org/question/qtiv2p1pd2/imsqti_implv2p1pd2.html>

IMS GLC. (a) [online] IMS Question & Test Interoperability Specification, 2012, [cit. 2012-04-29] Dostupné na www: <http://www.imsglobal.org/question/index.html>

IMS GLC. (b) [online] Domovská stránka, 2012, [cit. 2012-04-29] Dostupné na www: <http://www.imsglobal.org >

JISC CETIS. [online] QTI Training Guide, 2006, [cit. 2012-03-01] Dostupné na www: < http://wiki.cetis.ac.uk/QTI_Training_Guide>

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

OASIS. [online] RELAX NG Specification, 2001, [cit. 2012-04-29] Dostupné na www: <http://www.oasis-open.org/committees/relax-ng/spec-20011203.html>

Oracle. [online] Streaming API for XML, 2012, [cit. 2012-04-29] Dostupné na www: <http://docs.oracle.com/javase/tutorial/jaxp/stax/index.html>

Oracle. [online] Java Architecture for XML Binding (JAXB), 2003, [cit. 2012-04-29] Dostupné na www: <http://www.oracle.com/technetwork/articles/javase/index-140168.html>

Redwood Shores, CA. [online] Oracle Buys Sun, 2009, [cit. 2012-03-24] Dostupné na www: <http://www.oracle.com/us/corporate/press/018363>

SAX. [online] Domovská stránka, 2012, [cit. 2012-04-29] Dostupné na www: <http://www.saxproject.org>

Schematron. [online] Domovská stránka, 2012, [cit. 2012-04-29] Dostupné na www: <http://www.schematron.com>

The Apache Ant Project [online] Domovská stránka, 2012, [cit 28. 04. 2012]. Dostupné na www: <http://ant.apache.org/>

The Eclipse Foundation. [online] Domovská stránka, 2012, [cit. 2012-04-29] Dostupné na www: <http://www.eclipse.org>

TIOBE Software BV. [online] TIOBE Programming Community Index Definition, 2008, [cit. 2012-03-25] Dostupné na www: <http://www.tiobe.com/index.php/content/paperinfo/tpci/tpci_definition.htm>

TIOBE Software BV. [online] TIOBE Programming Community Index for March 2012, 2012, [cit. 2012-03-25] Dostupné na www: <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>

UML. [online] Domovská stránka, 2012, [cit 25. 03. 2012]. Dostupné na www: <http://www.uml.org>

Wikipedie: Otevřená encyklopedie. (a) [online] Programovací jazyk, 2012, [cit 25. 03. 2012]. Dostupné na www: <http://cs.wikipedia.org/w/index.php?title=Programovac%C3%AD_jazyk&oldid=8282605>

Wikipedie: Otevřená encyklopedie. (b) [online] Strukturované programování, 2012, [cit 25. 03. 2012]. Dostupné na www: <http://cs.wikipedia.org/wiki/Strukturovan%C3%A9_programov%C3%A1n%C3%A>

Wikipedie: Otevřená encyklopedie. (c) [online] Objektově orientované programování, 2012, [cit 25. 03. 2012]. Dostupné na www: <http://cs.wikipedia.org/wiki/Objektov%C4%9B_orientovan%C3%A9_programov%C3%A1n%C3%AD >

World Wide Web Consortium (W3C). (a) [online] Domovská stránka, 2012, [cit. 2012-04-28] Dostupné na www: < http://www.w3.org/>

World Wide Web Consortium (W3C). (b) [online] W3C XML Schema Definition Language (XSD) 1.1, 2012, [cit. 2012-04-29] Dostupné na www: <http://www.w3.org/TR/xmlschema11-1/>

World Wide Web Consortium (W3C). [online] XML Path Language (XPath) 2.0, 2010, [cit. 2012-04-28] Dostupné na www: < http://www.w3.org/TR/xpath20>

World Wide Web Consortium (W3C). [online] Extensible Markup Language (XML) 1.0, 2008, [cit. 2012-04-29] Dostupné na www: < http://www.w3.org/TR/xml>

World Wide Web Consortium (W3C). [online] Document Object Model (DOM) Level 3 Core Specification, 2004, [cit. 2012-04-29] Dostupné na www: <http://www.w3.org/TR/DOM-Level-3-Core>

15 Seznam příloh

Příloha A: XSD schéma vstupního XML dokumentu zadání úlohy (configuration/inputDocumentSchema.xsd)

Příloha B: ErrorHandler parseru SAX (data.ReportValidationError.java)

Příloha C: Validátor XML dokumentu, realizovaný technologií SAX (data.XMLValidator.java)

Příloha D: Třída pro zpracování vstupního XML dokumentu (data.InputParser.java)

Příloha E: Třída pro uchování obsahu XML dokumentu (data.TaskInput.java)

Příloha F: Rodičovská abstraktní třída datových typů (modules.TaskVarContainer.java)

Příloha G: Implementace datového typu Text (modules.Text.java)

Příloha H: Implementace datového typu Number (modules.Number.java)

Příloha I: Implementace datového typu MultipleChoice (modules.MultipleChoice.java)

Příloha J: Implementace datového typu Table (modules.Table.java)

Příloha K: Rodičovská abstraktní třída řešitelů úloh (logic.AbstractSolver.java)

Příloha L: Řešitel úlohy násobení dvou čísel (tasks.Matematika.Nasobeni.java)

Příloha M: Řešitel úlohy násobení dvou čísel v tabulce (tasks.Matematika.NasobeniTable.java)

Příloha N: Řešitel úlohy sčítání a odčítání dvou čísel (tasks.Matematika.ScitaniOdcitani.java)

Příloha O: Řešitel testové úlohy z českého jazyka (tasks.CeskyJazyk.VyjmenovanaSlovaB.java)

Příloha P: Systém testování očekávaných a generovaných dat (logic.Checker.java)

Příloha Q: Parser pro načtení konfigurace systému aliasů (data.AliasParser.java)

Příloha R: XSD schéma a konfigurační XML pro systém aliasů (configuration/typeAliasSchema.xsd; configuration/typeAlias.xml)

Příloha S: Tiskárna výstupního XML dokumentu ve standardu QTI (data.Printer.java)

Příloha T: Jádro generátoru testových úloh (logic.TaskGenerator.java)

Příloha U: Aplikace pro testování jádra generátoru úloh (logic.Application.java)

Příloha V: Příklad vygenerované úlohy ScitaniOdcitani (output/vygenerovaneQTI.xml)

Příloha W: Model implementovaného generátoru testových úloh

Příloha A: XSD schéma vstupního XML dokumentu zadání úlohy (configuration/inputDocumentSchema.xsd)

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:simpleType name="identifierType">
        <xs:restriction base="xs:normalizedString" />
    </xs:simpleType>
    <xs:simpleType name="titleType">
        <xs:restriction base="xs:string" />
    </xs:simpleType>
    <xs:simpleType name="varNameType">
        <xs:restriction base="xs:normalizedString"/>
    </xs:simpleType>
    <xs:simpleType name="varTypeType">
        <xs:restriction base="xs:normalizedString">
            <xs:enumeration value="num" />
            <xs:enumeration value="text" />
            <xs:enumeration value="multichoice" />
            <xs:enumeration value="tab" />
         </xs:restriction>
    </xs:simpleType>
    <xs:complexType name="viType">
        <xs:attribute  name="name" type="varNameType" use="required"/>
        <xs:attribute name="type" type="varTypeType" use="required"/>
    </xs:complexType>
    <xs:complexType name="voType">
        <xs:attribute  name="name" type="varNameType" use="required"/>
        <xs:attribute name="type" type="varTypeType" use="required"/>
    </xs:complexType>
    <xs:complexType name="contentType" mixed="true">
        <xs:choice minOccurs="0" maxOccurs="unbounded">
            <xs:element name="vi" type="viType"/>
            <xs:element name="vo" type="voType"/>
        </xs:choice>
    </xs:complexType>
    <xs:simpleType name="commentType">
        <xs:restriction base="xs:string" />
    </xs:simpleType>
    <xs:complexType name="taskType">
        <xs:sequence>
            <xs:element name="title" type="titleType" />
            <xs:element name="content" type="contentType" />
            <xs:element name="comment" type="commentType" />
        </xs:sequence>
        <xs:attribute name="identifier" type="identifierType" />
    </xs:complexType>

    <xs:element name="task" type="taskType">
      <xs:unique name="uniqueAttributeNameForVariables">
            <xs:selector xpath="./content/*"/>
            <xs:field xpath="@name"/>
       </xs:unique>
    </xs:element>
</xs:schema>
```

(Zdroj: vlastní zpracování, 2012)

Příloha B: ErrorHandler parseru SAX (data.ReportValidationError.java)

```
package data;

import org.xml.sax.ErrorHandler;
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;

public class ReportValidationError implements ErrorHandler {

 @Override
 public void error(SAXParseException e) throws SAXException {
       throw new SAXException("Chyba při validaci XML dokumentu: \n"
                           + textHlaseni(e));
 }

 @Override
 public void fatalError(SAXParseException e) throws SAXException {
       throw new SAXException("Fatální chyba při validaci XML dokumentu: \n"
                           + textHlaseni(e));
       }

 @Override
 public void warning(SAXParseException e) throws SAXException {
       throw new SAXException("Varování při validaci XML dokumentu: \n"
                           + textHlaseni(e));
       }

 private String textHlaseni(SAXParseException e) {
       return "soubor: " + e.getSystemId() + "\n" + "řádka: "
             + e.getLineNumber() + "\n" + "sloupec: " + e.getColumnNumber()
             + "\n" + "chyba: " + e.getLocalizedMessage() + "\n"
             + "=> naformátujte prosím XML dokument dle zadaného schématu";
 }
}
```

(Zdroj: vlastní zpracování, 2012)

Příloha C: Validátor XML dokumentu, realizovaný technologií SAX (data.XMLValidator.java)

```
package data;

import java.io.File;
import java.io.FileInputStream;

import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;

import org.xml.sax.InputSource;
import org.xml.sax.XMLReader;

public class XMLValidator {

 public XMLValidator (File xmlDocument, File schema) throws Exception {
       SAXParserFactory spf = SAXParserFactory.newInstance();
       spf.setValidating(true);

       spf.setFeature("http://xml.org/sax/features/namespaces", true);
       spf.setFeature("http://apache.org/xml/features/validation/
        schema", true);

       SAXParser sp = spf.newSAXParser();

       sp.setProperty("http://java.sun.com/xml/jaxp/properties/
        schemaLanguage", "http://www.w3.org/2001/XMLSchema");
       sp.setProperty("http://java.sun.com/xml/jaxp/properties/
        schemaSource", schema);

       XMLReader parser = sp.getXMLReader();
       parser.setErrorHandler(new ReportValidationError());

       FileInputStream stream = new FileInputStream(xmlDocument);
       InputSource inputSource = new InputSource(stream);
       parser.parse(inputSource);
 }
}
```

(Zdroj: vlastní zpracování, 2012)

Příloha D: Třída pro zpracování vstupního XML dokumentu (data.InputParser.java)

```
package data;

import java.io.File;
import java.io.FileInputStream;
import java.util.ArrayList;
import java.util.List;

import javax.xml.stream.XMLInputFactory;
import javax.xml.stream.XMLStreamReader;

public class InputParser {
 private final File SCHEMA =
       new File("configuration/inputDocumentSchema.xsd");
 private List<String[]> checkingList = new ArrayList<String[]>();
 private TaskInput uloha;

 public InputParser(File soubor) throws Exception {
  try {
   new XMLValidator(soubor, SCHEMA);
   XMLInputFactory factory = XMLInputFactory.newInstance();
   XMLStreamReader reader =
       factory.createXMLStreamReader(new  FileInputStream(soubor));

   String identifier = "";
   String title = "";
   String comment = "";
   String content = "";

   while (reader.hasNext()) {
       reader.next();
       if (reader.isStartElement()) {
         if (reader.getLocalName().equals("task")) {
           identifier =
             reader.getAttributeValue(null,"identifier").trim();
         } else if (reader.getLocalName().equals("title")) {
             title = reader.getElementText().trim();
         } else if (reader.getLocalName().equals("content")) {
             content = contentProcessing(reader).trim();
         } else if (reader.getLocalName().equals("comment")) {
             comment = reader.getElementText().trim();
         }
       }
   }

  uloha = new TaskInput(identifier, title, comment, content);

  } catch (Exception e) {
       throw e;
  }
 }
```

```
 private String contentProcessing(XMLStreamReader reader)
 throws  Exception {
  StringBuffer outputBuffer = new StringBuffer();

  while (!(reader.isEndElement() && reader.getLocalName().equals("content")))
  {
       reader.next();

       if (reader.isCharacters()) {
             outputBuffer.append(reader.getText());
       } else if (reader.isStartElement()&&
                   (reader.getLocalName().equals("vi") ||
                   (reader.getLocalName().equals("vo")))) {
             String state = "INPUT";
             if (reader.getLocalName().equals("vo")) {
                   state = "OUTPUT";
             }

             String name = reader.getAttributeValue(null, "name");
             String type = reader.getAttributeValue(null, "type");

             String[] parsedData = new String[3];
             parsedData[0] = name;
             parsedData[1] = type;
             parsedData[2] = state;

             checkingList.add(parsedData);
             outputBuffer.append("###" + name + "##");
       }
  }
  return outputBuffer.toString();
 }

 public List<String[]> getCheckingList() {
       return checkingList;
 }

 public TaskInput getUloha() {
       return uloha;
 }
}
```

(Zdroj: vlastní zpracování, 2012)

Příloha E: Třída pro uchování obsahu XML dokumentu (data.TaskInput.java)

```
package data;

public class TaskInput {
  private String identifier;
  private String title;
  private String content;
  private String comment;


  public TaskInput(String identifier, String title, String comment, String
  content) {
      this.identifier = identifier;
      this.title = title;
      this.comment = comment;
      this.content = content;
  }

  public String getIdentifier() {
      return identifier;
  }

  public String getTitle() {
      return title;
  }

  public String getComment() {
      return comment;
  }

  public String getContent() {
      return content;
  }

  @Override
  public String toString() {
      return "TaskInput [identifier=" + identifier +
             ", title=" + title + ", comment="
             + comment + ", content=" + content + "]";
  }
}
```

(Zdroj: vlastní zpracování, 2012)

Příloha F: Rodičovská abstraktní třída datových typů (modules.TaskVarContainer.java)

```
package modules;

import java.util.List;

import org.imsglobal.xsd.imsqti_v2p1.*;

public abstract class TaskVarContainer {
  protected String state;
  protected String name;

  public abstract List<ResponseDeclarationType>
    printResponseDeclaration(ObjectFactory of);

  public abstract Object printItemBody(ObjectFactory of);


  public abstract List<ResponseConditionType>
    printResponseCondition(ObjectFactory of);

      public String getState() {
            return state;
      }

      public void setState(String state) {
            this.state = state;
      }

      public String getName() {
            return name;
      }

      public void setName(String name) {
            this.name = name;
      }
}
```

(Zdroj: vlastní zpracování, 2012)

Příloha G: Implementace datového typu Text (modules.Text.java)

```
package modules;

import java.util.ArrayList;
import java.util.List;

import org.imsglobal.xsd.imsqti_v2p1.BaseTypeType;
import org.imsglobal.xsd.imsqti_v2p1.CardinalityType;
import org.imsglobal.xsd.imsqti_v2p1.CorrectResponseType;
import org.imsglobal.xsd.imsqti_v2p1.ExtendedTextInteractionType;
import org.imsglobal.xsd.imsqti_v2p1.ObjectFactory;
import org.imsglobal.xsd.imsqti_v2p1.ResponseConditionType;
import org.imsglobal.xsd.imsqti_v2p1.ResponseDeclarationType;
import org.imsglobal.xsd.imsqti_v2p1.ValueType;

public class Text extends TaskVarContainer {

  private String value;


  public Text(String value) {
      this.value = value;
  }


  public String getValue(){
      return value;
  }


  @Override
  public List<ResponseDeclarationType>
    printResponseDeclaration(ObjectFactory of) {
      ResponseDeclarationType responseDeclaration =
       of.createResponseDeclarationType();
        responseDeclaration.setIdentifier(this.name);
        responseDeclaration.setCardinality(CardinalityType.SINGLE);
        responseDeclaration.setBaseType(BaseTypeType.STRING);

        CorrectResponseType spravnaOdpoved = of.createCorrectResponseType();
        ValueType hodnota = of.createValueType();
        hodnota.setValue(this.value);
        spravnaOdpoved.getValue().add(hodnota);

        responseDeclaration.setCorrectResponse(spravnaOdpoved);

        List<ResponseDeclarationType> list =
             new ArrayList<ResponseDeclarationType>();
        list.add(responseDeclaration);
        return list;
      }
```

```
  @Override
  public Object printItemBody(ObjectFactory of) {
       if (this.state.equals("INPUT")) {
              return this.value;
       } else {
              ExtendedTextInteractionType etit =
                     of.createExtendedTextInteractionType();
              etit.setResponseIdentifier(this.name);
              return etit;
              }
       }

  @Override
  public List<ResponseConditionType>
    printResponseCondition(ObjectFactory of) {
       return null;
  }
}
```

(Zdroj: vlastní zpracování, 2012)

Příloha H: Implementace datového typu Number (modules.Number.java)

```
package modules;

import java.util.ArrayList;
import java.util.List;

import org.imsglobal.xsd.imsqti_v2p1.BaseTypeType;
import org.imsglobal.xsd.imsqti_v2p1.CardinalityType;
import org.imsglobal.xsd.imsqti_v2p1.CorrectResponseType;
import org.imsglobal.xsd.imsqti_v2p1.CorrectType;
import org.imsglobal.xsd.imsqti_v2p1.EqualType;
import org.imsglobal.xsd.imsqti_v2p1.ExtendedTextInteractionType;
import org.imsglobal.xsd.imsqti_v2p1.ObjectFactory;
import org.imsglobal.xsd.imsqti_v2p1.ResponseConditionType;
import org.imsglobal.xsd.imsqti_v2p1.ResponseDeclarationType;
import org.imsglobal.xsd.imsqti_v2p1.ResponseIfType;
import org.imsglobal.xsd.imsqti_v2p1.ToleranceModeType;
import org.imsglobal.xsd.imsqti_v2p1.ValueType;
import org.imsglobal.xsd.imsqti_v2p1.VariableType;

public class Number extends TaskVarContainer {
  private String value;
  private String tolerance;
  private String pocetDesetinnychMist;
  private String jednotka;

  public Number(String value, String tolerance, String pocetDesetinnychMist,
    String jednotka) {

      this.value = value;
      this.tolerance = tolerance;
      this.pocetDesetinnychMist = pocetDesetinnychMist;
      this.jednotka = jednotka;
  }


  public String getValue(){
      return value;
  }


  @Override
  public List<ResponseDeclarationType>
    printResponseDeclaration(ObjectFactory of) {
      ResponseDeclarationType responseDeclaration =
        of.createResponseDeclarationType();
      responseDeclaration.setIdentifier(this.name);
      responseDeclaration.setCardinality(CardinalityType.SINGLE);
      responseDeclaration.setBaseType(BaseTypeType.STRING);

      CorrectResponseType spravnaOdpoved = of.createCorrectResponseType();
      ValueType hodnota = of.createValueType();
      hodnota.setValue(this.value);
      hodnota.setFieldIdentifier(this.name + "_correct");
      spravnaOdpoved.getValue().add(hodnota);

      responseDeclaration.setCorrectResponse(spravnaOdpoved);
```

```
        List<ResponseDeclarationType> list =
          new ArrayList<ResponseDeclarationType>();
               list.add(responseDeclaration);
               return list;
        }

    @Override
    public Object printItemBody(ObjectFactory of) {
        if (this.state.equals("INPUT")) {
               if (this.jednotka == null) {
                      return this.value;
               } else {
                      return this.value + " " + this.jednotka;
               }

        } else {
               ExtendedTextInteractionType etit =
                      of.createExtendedTextInteractionType();
               etit.setResponseIdentifier(this.name);
               return etit;
        }
    }

    @Override
    public List<ResponseConditionType>
      printResponseCondition(ObjectFactory of) {
        VariableType variable = of.createVariableType();
        variable.setIdentifier(this.name);

        CorrectType correct = of.createCorrectType();
        correct.setIdentifier(this.name + "_correct");

        EqualType equal = of.createEqualType();
        equal.setToleranceMode(ToleranceModeType.ABSOLUTE);
        equal.getTolerance().add(this.tolerance);
        equal.getExpressionElementGroup().add(variable);
        equal.getExpressionElementGroup().add(correct);


        ResponseIfType responseIf = of.createResponseIfType();
        responseIf.setEqual(equal);

        ResponseConditionType condition = of.createResponseConditionType();
        condition.setResponseIf(responseIf);

        List<ResponseConditionType> list =
          new ArrayList<ResponseConditionType>();
        list.add(condition);
        return list;
    }
}
```

(Zdroj: vlastní zpracování, 2012)

Příloha I: Implementace datového typu MultipleChoice (modules.MultipleChoice.java)

```
package modules;

import java.util.ArrayList;
import java.util.List;

import org.imsglobal.xsd.imsqti_v2p1.BaseTypeType;
import org.imsglobal.xsd.imsqti_v2p1.CardinalityType;
import org.imsglobal.xsd.imsqti_v2p1.ChoiceInteractionType;
import org.imsglobal.xsd.imsqti_v2p1.CorrectResponseType;
import org.imsglobal.xsd.imsqti_v2p1.ObjectFactory;
import org.imsglobal.xsd.imsqti_v2p1.PromptType;
import org.imsglobal.xsd.imsqti_v2p1.ResponseConditionType;
import org.imsglobal.xsd.imsqti_v2p1.ResponseDeclarationType;
import org.imsglobal.xsd.imsqti_v2p1.SimpleChoiceType;
import org.imsglobal.xsd.imsqti_v2p1.ValueType;

public class MultipleChoice extends TaskVarContainer {
  private String question;
  private String[] answers;
  private int[] rightAnswers;

  public MultipleChoice(String question, String[] answers,
    int[] rightAnswers) {
      super();
      this.question = question;
      this.answers = answers;
      this.rightAnswers = rightAnswers;
  }

  @Override
  public List<ResponseDeclarationType>
    printResponseDeclaration(ObjectFactory of) {
      ResponseDeclarationType responseDeclaration =
        of.createResponseDeclarationType();
      responseDeclaration.setIdentifier(this.name);
      responseDeclaration.setCardinality(CardinalityType.MULTIPLE);
      responseDeclaration.setBaseType(BaseTypeType.IDENTIFIER);

      CorrectResponseType spravnaOdpoved = of.createCorrectResponseType();
      for (int i = 0; i < rightAnswers.length; i++) {
           ValueType hodnota = of.createValueType();

           hodnota.setValue(Character.toString((char)(65+rightAnswers[i])));
           spravnaOdpoved.getValue().add(hodnota);
      }
      responseDeclaration.setCorrectResponse(spravnaOdpoved);

      List<ResponseDeclarationType> list =
        new ArrayList<ResponseDeclarationType>();
      list.add(responseDeclaration);
      return list;
  }
```

```
@Override
  public Object printItemBody(ObjectFactory of) {
       ChoiceInteractionType multichoice = of.createChoiceInteractionType();
       multichoice.setResponseIdentifier(this.name);

       PromptType prompt = of.createPromptType();
       prompt.getContent().add(this.question);
       multichoice.setPrompt(prompt);

       for (int i = 0; i < answers.length; i++) {
              SimpleChoiceType choice = of.createSimpleChoiceType();

              int ascii = 65 + i;
              char letter = (char)ascii;

              choice.setIdentifier(Character.toString(letter));
              choice.setFixed(true);
              choice.getContent().add(answers[i]);

              multichoice.getSimpleChoice().add(choice);
       }
       return multichoice;
  }

  @Override
  public List<ResponseConditionType>
    printResponseCondition(ObjectFactory of) {
       return null;
  }
}
```

(Zdroj: vlastní zpracování, 2012)

Příloha J: Implementace datového typu Table (modules.Table.java)

```
package modules;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

import org.imsglobal.xsd.imsqti_v2p1.ObjectFactory;
import org.imsglobal.xsd.imsqti_v2p1.ResponseConditionType;
import org.imsglobal.xsd.imsqti_v2p1.ResponseDeclarationType;
import org.imsglobal.xsd.imsqti_v2p1.TableType;
import org.imsglobal.xsd.imsqti_v2p1.TbodyType;
import org.imsglobal.xsd.imsqti_v2p1.TdType;
import org.imsglobal.xsd.imsqti_v2p1.TrType;

public class Table extends TaskVarContainer {

  private TaskVarContainer[][] content;



  public Table(TaskVarContainer[][] content){
      this.content = content;
  }

  @Override
  public List<ResponseDeclarationType>
    printResponseDeclaration(ObjectFactory of) {
      List<ResponseDeclarationType> list =
        new ArrayList<ResponseDeclarationType>();
      for (int i = 0; i < content.length; i++) {
            for (int j = 0; j < content[0].length; j++) {
                  list.addAll(content[i][j].printResponseDeclaration(of));
            }
      }
      return list;
  }

  @Override
  public Object printItemBody(ObjectFactory of) {
      TbodyType body = of.createTbodyType();
      for (int i = 0; i < content.length; i++) {
            TrType radek = of.createTrType();
            for (int j = 0; j < content[0].length; j++) {
                  TaskVarContainer c = content[i][j];
                  TdType bunka = of.createTdType();

                  Object cellBody = c.printItemBody(of);
                  if (cellBody instanceof String) {
                        bunka.getContent().add(cellBody.toString());
                        radek.getTableCellElementGroup().add(bunka);
                  } else {
                        bunka.getContent().add((Serializable)cellBody);
                        radek.getTableCellElementGroup().add(bunka);
                  }
            }
            body.getTr().add(radek);
      }
```

```
        TableType table = of.createTableType();
        table.getTbody().add(body);
        return table;
    }

    @Override
    public List<ResponseConditionType>
      printResponseCondition(ObjectFactory of) {
        List<ResponseConditionType> list =
          new ArrayList<ResponseConditionType>();
        for (int i = 0; i < content.length; i++) {
              for (int j = 0; j < content[0].length; j++) {
                    List<ResponseConditionType> polozka =
                      content[i][j].printResponseCondition(of);
                    if (polozka != null) {
                          list.addAll(polozka);
                    }
              }
        }
        return list;
    }
}
```

(Zdroj: vlastní zpracování, 2012)

Příloha K: Rodičovská abstraktní třída řešitelů úloh (logic.AbstractSolver.java)

```
package logic;

import java.util.HashMap;
import java.util.Map;

import modules.TaskVarContainer;

public abstract class AbstractSolver {
  protected Map<String, TaskVarContainer> kontejner =
    new HashMap<String, TaskVarContainer>();

  public AbstractSolver() {
      generator();
      solver();
  }

  protected abstract void generator();
  protected abstract void solver();

  public Map<String, TaskVarContainer> getKontejner() {
      return kontejner;
  }
}
```

(Zdroj: vlastní zpracování, 2012)

Příloha L: Řešitel úlohy násobení dvou čísel (tasks.Matematika.Nasobeni.java)

```
package tasks.Matematika;
import java.util.Random;

import logic.AbstractSolver;
import modules.Number;
import modules.Text;

public class Nasobeni extends AbstractSolver {
  private int a;
  private int b;
  protected void solver() {
       kontejner.put("vysledek", new Number(Integer.toString(a*b),null, null,
         null));
  }

  protected void generator() {
       Random random = new Random();
       a = random.nextInt(10);
       b = random.nextInt(10);

       kontejner.put("a", new Number(Integer.toString(a), null, null, null));
       kontejner.put("b", new Number(Integer.toString(b), null, null, null));
       kontejner.put("operator", new Text("*"));
  }
}
```

(Zdroj: vlastní zpracování, 2012)

Příloha M: Řešitel úlohy násobení dvou čísel v tabulce

(tasks.Matematika.NasobeniTable.java)

```
package tasks.Matematika;

import java.util.Random;
import logic.AbstractSolver;
import modules.Number;
import modules.Table;
import modules.TaskVarContainer;
import modules.Text;

public class NasobeniTable extends AbstractSolver {
  TaskVarContainer[][] content;
  protected void solver() {}

  protected void generator() {
       final int SLOUPCE = 6;
       final int RADKY = 5;
       content = new TaskVarContainer[RADKY][SLOUPCE];
       Random random = new Random();
       content[0][0] = new Text("*");
       content[0][0].setState("INPUT");
       content[0][0].setName("cell00");

       for (int i = 0; i < RADKY; i++) {
         for (int j = 0; j < SLOUPCE; j++) {
           if (i == 0 && j == 0) {
             continue;
           } else if (i == 0 || j == 0) {
             int randomInt;
             do {
               randomInt = random.nextInt(20);
             } while (randomInt == 0);
               content[i][j] =
                 new Number(Integer.toString(randomInt), null, null, null);
               content[i][j].setState("INPUT");
               content[i][j].setName("cell" + Integer.toString(i)
                 + Integer.toString(j));

           } else {
             int a = Integer.parseInt(((Number)
             content[0][j]).getValue());
             int b = Integer.parseInt(((Number)content[i][0]).getValue());
             int vysledek = a * b;

             content[i][j] =
               new Number(Integer.toString(vysledek), null, null, null);
             content[i][j].setState("OUTPUT");
             content[i][j].setName("cell" + Integer.toString(i)
               + Integer.toString(j));
           }
         }
       }
       kontejner.put("tabulka", new Table(content));
  }
}
```

(Zdroj: vlastní zpracování, 2012)

Příloha N: Řešitel úlohy sčítání a odčítání dvou čísel

(tasks.Matematika.ScitaniOdcitani.java)

```
package tasks.Matematika;
import java.util.Random;

import logic.AbstractSolver;
import modules.Number;
import modules.TaskVarContainer;
import modules.Text;

public class ScitaniOdcitani extends AbstractSolver {
  private int a;
  private int b;
  private String operator;

  protected void solver() {
       TaskVarContainer operator = kontejner.get("operator");
       String op = ((Text) operator).getValue();
       int vysledek;
       if (op.equals("+")) {
             vysledek = a + b;
       } else {
             vysledek = a - b;
       }
       kontejner.put("vysledek", new Number(Integer.toString(vysledek),null,
         null, null));
  }

  protected void generator() {
       Random random = new Random();
       a = random.nextInt(10);
       b = random.nextInt(10);

       int c = random.nextInt(10);
       operator = (c < 5) ? "-" : "+";

       kontejner.put("a", new Number(Integer.toString(a), null, null, null));
       kontejner.put("b", new Number(Integer.toString(b), null, null, null));
       kontejner.put("operator", new Text(operator));
  }
}
```

(Zdroj: vlastní zpracování, 2012)

Příloha O: Řešitel testové úlohy z českého jazyka

(tasks.CeskyJazyk.VyjmenovanaSlovaB.java)

```
package tasks.CeskyJazyk;
import logic.AbstractSolver;
import modules.MultipleChoice;

public class VyjmenovanaSlovaB extends AbstractSolver {
  protected void solver() {}
  protected void generator() {
       String question = "Vyberte správnou variantu";
       String[] answers = new String[] {"Babička bydlí v Kolíně",
         "V Kolíně bívalo krásně","Babička už není obivatel Kolína",
         "Je to krásný nábytek"};
       int[] rightAnswers = new int[]{0,3};
       kontejner.put("zadani", new MultipleChoice(question, answers,
         rightAnswers));
  }
}
```

(Zdroj: vlastní zpracování, 2012)

Příloha P: Systém testování očekávaných a generovaných dat (logic.Checker.java)

```
package logic;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.List;
import java.util.Map;

import modules.TaskVarContainer;

public class Checker {
  private List<String[]> checkingList = new ArrayList<String[]>();
  private Map<String, TaskVarContainer> kontejner =
    new HashMap<String, TaskVarContainer>();
  private Map<String, String> aliasesMap = new HashMap<String, String>();

  public Checker(List<String[]> checkingList,
    Map<String, TaskVarContainer> kontejner, Map<String, String> aliasesMap)
     throws Exception {
      this.checkingList = checkingList;
      this.kontejner = kontejner;
      this.aliasesMap = aliasesMap;
      check();
  }

  private void check() throws Exception {
      final int NAME = 0;
      final int TYPE = 1;
      final int STATE = 2;

      for (Iterator<String[]> iterator = checkingList.iterator();
        iterator.hasNext();) {
            String[] checkItem = (String[]) iterator.next();
            TaskVarContainer c = kontejner.get(checkItem[NAME]);
            if (c == null) {
                  String msg = "Proměnná \"" + checkItem[NAME] + "\" " +
                    "nebyla inicializována.";
                  throw new Exception(msg);
            }

            String awaitedType = aliasesMap.get(checkItem[TYPE]);
            String actualType = c.getClass().getSimpleName();

            if (!actualType.equals(awaitedType)) {
                  String msg = "Proměnná \"" + checkItem[TYPE] + "\" " +
                    "má být typu " + awaitedType + ", je však typu " +
                      actualType + ".";
                  throw new Exception(msg);
            }

            c.setName(checkItem[NAME]);
            c.setState(checkItem[STATE]);
      }
  }
}
```

(Zdroj: vlastní zpracování, 2012)

Příloha Q: Parser pro načtení konfigurace systému aliasů (data.AliasParser.java)

```
package data;
import java.io.File;
import java.io.FileInputStream;
import java.util.HashMap;
import java.util.Map;

import javax.xml.stream.XMLInputFactory;
import javax.xml.stream.XMLStreamReader;

public class AliasParser {
  private final File ALIAS_FILE = new File("configuration/typeAlias.xml");
  private final File ALIAS_FILE_SCHEMA =
    new File("configuration/typeAliasSchema.xsd");
  private Map<String, String> aliasesMap = new HashMap<String, String>();

  public AliasParser() throws Exception {
    try {
      new XMLValidator(ALIAS_FILE,ALIAS_FILE_SCHEMA);
      XMLInputFactory factory = XMLInputFactory.newInstance();
      XMLStreamReader reader =
        factory.createXMLStreamReader(new FileInputStream(ALIAS_FILE));
      String module = null;
      while (reader.hasNext()) {
            reader.next();
            if (reader.isStartElement()) {
                  if (reader.getLocalName().equals("module")) {
                        module = reader.getElementText();
                  }
                  if (reader.getLocalName().equals("alias")) {
                        aliasesMap.put(reader.getElementText(), module);
                  }
            }
      }
    } catch (Exception e) {
      throw e;
    }
  }

  public Map<String, String> getAlias() {
      return aliasesMap;
  }
}
```

(Zdroj: vlastní zpracování, 2012)

Příloha R: XSD schéma a konfigurační XML pro systém aliasů

(configuration/typeAliasSchema.xsd; configuration/typeAlias.xml)

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:simpleType name="moduleType">
        <xs:restriction base="xs:normalizedString" />
    </xs:simpleType>

    <xs:simpleType name="aliasType">
        <xs:restriction base="xs:normalizedString" />
    </xs:simpleType>

    <xs:complexType name="defType">
        <xs:sequence>
            <xs:element name="module" type="moduleType" />
            <xs:element name="alias" type="aliasType" />
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="typeAliasesType">
        <xs:sequence>
            <xs:element name="def" type="defType" maxOccurs="unbounded" />
        </xs:sequence>
    </xs:complexType>

    <xs:element name="typeAliases" type="typeAliasesType" />
</xs:schema>
```

(Zdroj: vlastní zpracování, 2012)

```
<?xml version="1.0" encoding="UTF-8"?>
<typeAliases>
    <def>
        <module>Number</module>
        <alias>num</alias>
    </def>
    <def>
        <module>Text</module>
        <alias>text</alias>
    </def>
    <def>
        <module>MultipleChoice</module>
        <alias>multichoice</alias>
    </def>
    <def>
        <module>Table</module>
        <alias>tab</alias>
    </def>
</typeAliases>
```

(Zdroj: vlastní zpracování, 2012)

Příloha S: Tiskárna výstupního XML dokumentu ve standardu QTI (data.Printer.java)

```
package data;

import java.io.FileOutputStream;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.List;
import java.util.Map;

import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBElement;
import javax.xml.bind.Marshaller;

import modules.TaskVarContainer;

import org.imsglobal.xsd.imsqti_v2p1.AssessmentItemType;
import org.imsglobal.xsd.imsqti_v2p1.ItemBodyType;
import org.imsglobal.xsd.imsqti_v2p1.ObjectFactory;
import org.imsglobal.xsd.imsqti_v2p1.PType;
import org.imsglobal.xsd.imsqti_v2p1.ResponseConditionType;
import org.imsglobal.xsd.imsqti_v2p1.ResponseDeclarationType;
import org.imsglobal.xsd.imsqti_v2p1.ResponseProcessingType;

public class Printer {
 private final String ENCODING = "utf-8";
 private String soubor;
 private ObjectFactory of;
 private TaskInput sablona;
 private Map<String, TaskVarContainer> kontejner;


  public Printer(TaskInput sablona, Map<String, TaskVarContainer> kontejner,
    String soubor) throws Exception {
    this.sablona = sablona;
    this.kontejner = kontejner;
    this.soubor = soubor;
    printDocument();
  }

  private void printDocument() throws Exception {
    JAXBContext jc=JAXBContext.newInstance("org.imsglobal.xsd.imsqti_v2p1");

    Marshaller m = jc.createMarshaller();
    m.setProperty(Marshaller.JAXB_ENCODING, ENCODING);
    m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);

    JAXBElement<AssessmentItemType> element = createDocument();
    m.marshal(element, new FileOutputStream(soubor));
  }

  private JAXBElement<AssessmentItemType> createDocument() {
    of = new ObjectFactory();
    AssessmentItemType assesmentItem = of.createAssessmentItemType();
    assesmentItem.setAdaptive(false);
    assesmentItem.setTimeDependent(false);
    assesmentItem.setTitle(sablona.getTitle());
    assesmentItem.setIdentifier(sablona.getIdentifier());
```

```
    assesmentItem.getResponseDeclaration()
.     addAll(createResponseDeclarations());
    assesmentItem.setItemBody(createItemBody());
    assesmentItem.setResponseProcessing(createResponseProcessing());

    JAXBElement<AssessmentItemType> element =
      of.createAssessmentItem(assesmentItem);
    return element;
  }

  private ItemBodyType createItemBody() {
    ItemBodyType itemBody = of.createItemBodyType();
    String rawContent = sablona.getContent();

    Map<String, Object> interactions = new HashMap<String,Object>();
      for (Iterator<Map.Entry<String, TaskVarContainer>> it = kontejner
        .entrySet().iterator(); it.hasNext();) {
      Map.Entry<String, TaskVarContainer> e = it.next();

      Object value = e.getValue().printItemBody(of);
      if (value instanceof String) {
        rawContent = rawContent.replace("###" + e.getKey() + "##",
          value.toString());
      } else {
            interactions.put(e.getKey(), value);
      }
    }

    String[] preprocessedContent = rawContent.split("##");
      for (String partialContent : preprocessedContent) {
        if (partialContent.contains("#")) {
          partialContent = partialContent.trim();
          itemBody.getBlockElementGroup().add(interactions
            .get(partialContent.substring(1)));
        } else {
            PType odstavec = of.createPType();
            odstavec.getContent().add(partialContent);
            itemBody.getBlockElementGroup().add(odstavec);
        }
      }
    return itemBody;
  }

  private List<ResponseDeclarationType> createResponseDeclarations() {
    List<ResponseDeclarationType> responseDeclarationTypeList =
      new ArrayList<ResponseDeclarationType>();
    for (Iterator<Map.Entry<String, TaskVarContainer>> it = kontejner
      .entrySet().iterator(); it.hasNext();) {
      Map.Entry<String, TaskVarContainer> e = it.next();
      TaskVarContainer c = e.getValue();
      if (c.getState().equals("OUTPUT")) {
        List<ResponseDeclarationType> list = c.printResponseDeclaration(of);
        if (list != null) {
          responseDeclarationTypeList.addAll(list);
        }
      }
    }
    return responseDeclarationTypeList;
  }
```

```
  private ResponseProcessingType createResponseProcessing() {
    List<ResponseConditionType> responseConditionTypeList =
      new ArrayList<ResponseConditionType>();
      for (Iterator<Map.Entry<String, TaskVarContainer>> it = kontejner
       .entrySet().iterator(); it.hasNext();) {
        Map.Entry<String, TaskVarContainer> e = it.next();

        TaskVarContainer c = e.getValue();
        if (c.getState().equals("OUTPUT")) {
          List<ResponseConditionType> list = c.printResponseCondition(of);
          if (list != null) {
            responseConditionTypeList.addAll(list);
          }
        }
      }

      ResponseProcessingType responseProcessing =
        of.createResponseProcessingType();
      responseProcessing.getResponseRuleElementGroup()
        .addAll(responseConditionTypeList);
      return responseProcessing;
  }
}
```

(Zdroj: vlastní zpracování, 2012)

Příloha T: Jádro generátoru testových úloh (logic.TaskGenerator.java)

```
package logic;

import java.io.File;
import java.util.regex.Pattern;

import data.AliasParser;
import data.InputParser;
import data.Printer;

public class TaskGenerator {

  public TaskGenerator(File inputFile, AliasParser ap, String outputFile)
   throws Exception {
    InputParser uloha = new InputParser(inputFile);
    AbstractSolver sol = getSolver(inputFile);
    new Checker(uloha.getCheckingList(), sol.getKontejner(), ap.getAlias());
    new Printer(uloha.getUloha(), sol.getKontejner(), outputFile);
  }


  private AbstractSolver getSolver(File inputFile) throws Exception{
    String path = inputFile.getPath();
    String delimiter = "tasks";
    path = path.substring(path.indexOf(delimiter) + delimiter.length() + 1);

    String[] partialPath = path.split(Pattern.quote(File.separator));

    StringBuffer solverPath = new StringBuffer();
    solverPath.append(delimiter);
    for (int i = 0; i < partialPath.length-1; i++) {
      solverPath.append("." + partialPath[i]);
    }

    AbstractSolver solver = null;
    try {
      solver =
       (AbstractSolver) Class.forName(solverPath.toString()).newInstance();
    } catch (ClassNotFoundException e) {
      throw new Exception("Problém s načtením solveru úlohy z balíku: "
        + solverPath.toString());
    }
    return solver;
  }
}
```

(Zdroj: vlastní zpracování, 2012)

Příloha U: Aplikace pro testování jádra generátoru úloh (logic.Application.java)

```
package logic;

import java.awt.BorderLayout;
import java.awt.Dimension;
import java.io.File;

import javax.swing.JDialog;
import javax.swing.JFileChooser;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import javax.swing.JProgressBar;
import javax.swing.filechooser.FileFilter;

import data.AliasParser;

public class Application {
  public static void main(String[] args) {
    try {
      AliasParser ap = new AliasParser();
      JFileChooser fileChooser = createJFileChooser();
      if (fileChooser.showOpenDialog(new JFrame("Vyberte soubor")) ==
        JFileChooser.APPROVE_OPTION) {
            File soubor = fileChooser.getSelectedFile();

            JDialog dialog = createJDialog();
            dialog.setVisible(true);
            new TaskGenerator(soubor, ap, "output/vygenerovaneQTI.xml");
            dialog.setVisible(false);

            JOptionPane.showMessageDialog(null, "Operace dokončena",
              "Operace dokončena", JOptionPane.INFORMATION_MESSAGE);
            System.exit(0);
      }
      System.exit(0);

    } catch (Exception e) {
      e.printStackTrace();
      JOptionPane.showMessageDialog(null, e.getMessage()
        + "\n => Ukončuji program z důvodu nekonzistentnosti dat",
          "Vyskytla se chyba", JOptionPane.ERROR_MESSAGE);
      System.exit(0);
    }
  }

  private static JDialog createJDialog() {
      JDialog dialog = new JDialog();
      dialog.setSize(new Dimension(300, 100));
      dialog.setLocationRelativeTo(null);
      dialog.setLayout(new BorderLayout());
      dialog.getContentPane().add(new JLabel("Generuji úlohu..."),
        BorderLayout.NORTH);

      JProgressBar pb = new JProgressBar();
      pb.setIndeterminate(true);
      dialog.getContentPane().add(pb, BorderLayout.SOUTH);
      return dialog;
  }
```

```
  private static JFileChooser createJFileChooser() {
      JFileChooser fileChooser = new JFileChooser("./tasks");
      fileChooser.setFileFilter(new FileFilter() {

            @Override
            public String getDescription() {
                  return "XML soubor zadání úlohy";
            }

            @Override
                  public boolean accept(File file) {
                  if (file.isDirectory()) {
                        return true;
                  }

                  if (file.getName().toLowerCase().endsWith(".xml")) {
                        return true;
                  }
                        return false;
                  }
            });
      return fileChooser;
  }
}
```

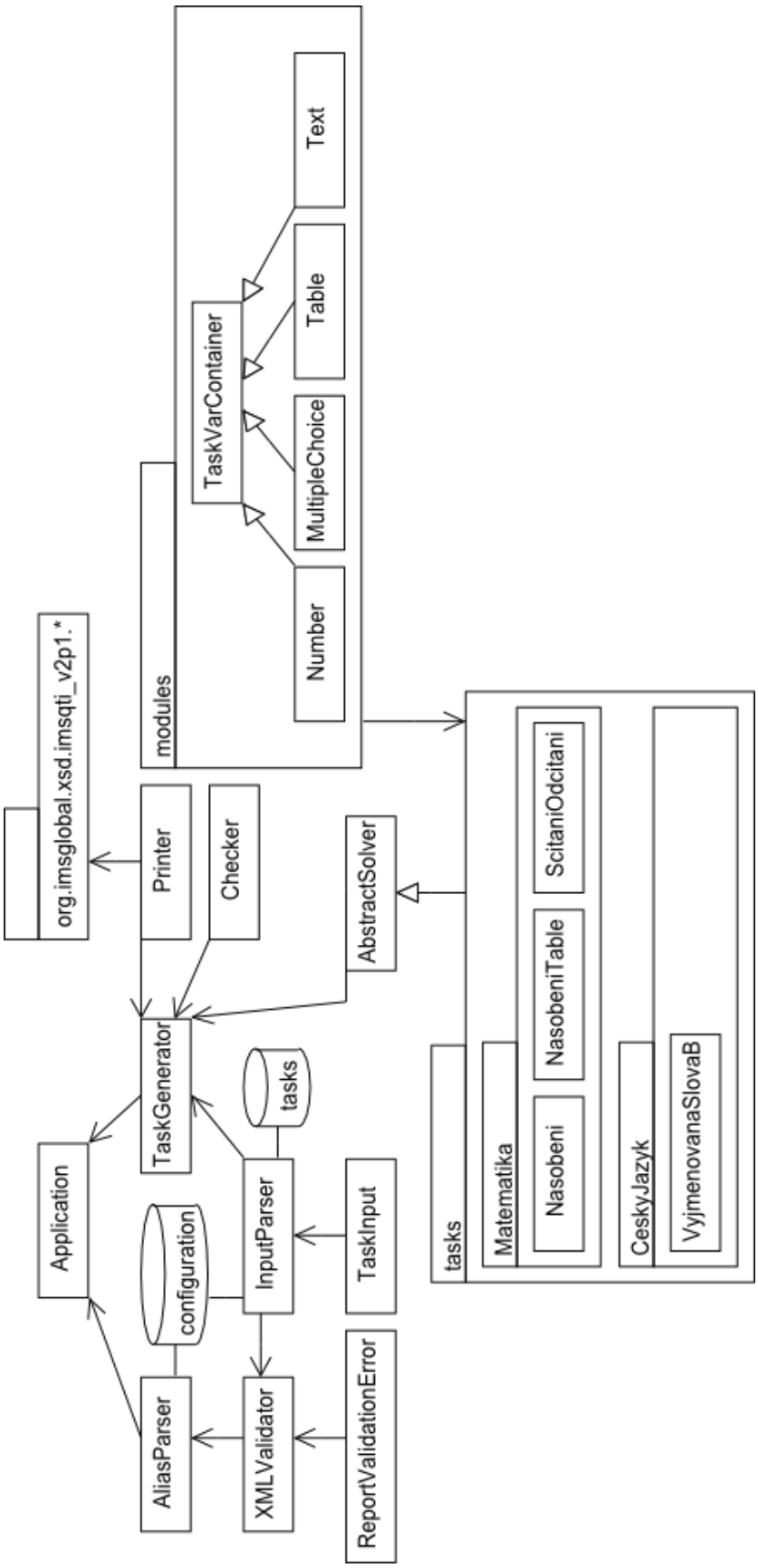
(Zdroj: vlastní zpracování, 2012)

Příloha V: Příklad vygenerované úlohy ScitaniOdcitani (output/vygenerovaneQTI.xml)

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<assessmentItem timeDependent="false" adaptive="false"
  title="Sčítání a odčítání dvou čísel" identifier="scitani_odcitani"
  xmlns="http://www.imsglobal.org/xsd/imsqti_v2p1">
    <responseDeclaration baseType="string" cardinality="single"
      identifier="vysledek">
        <correctResponse>
            <value fieldIdentifier="vysledek_correct">15</value>
        </correctResponse>
    </responseDeclaration>
    <itemBody>
        <p>Spočítejte výsledek 9 + 6 = </p>
        <extendedTextInteraction responseIdentifier="vysledek"/>
    </itemBody>
    <responseProcessing>
        <responseCondition>
            <responseIf>
                <equal tolerance="" toleranceMode="absolute">
                    <variable identifier="vysledek"/>
                    <correct identifier="vysledek_correct"/>
                </equal>
            </responseIf>
        </responseCondition>
    </responseProcessing>
</assessmentItem>
```

(Zdroj: vlastní zpracování, 2012)

Příloha W: Model implementovaného generátoru testových úloh



(Zdroj: vlastní zpracování, 2012)

Abstrakt

BALON, M. *Návrh a implementace modulárního generátoru úloh.* Bakalářská práce. Plzeň: Fakulta ekonomická ZČU v Plzni, 53 s., 2012

Klíčová slova: XML, QTI, parser, programovací jazyk, Java, generátor úloh

Předložená práce je zaměřena na návrh a implementaci softwarového nástroje pro využití ve vzdělávání. Jedná se o univerzální a modulární generátor testových úloh, jež na základě vstupních dat generuje data výstupní. K tomu využívá nejmodernější technologie jako XML, XSD schémata, objektově orientovaný jazyk Java a technologii JAXB. Výstupem tohoto systému je XML dokument vyhovující normě QTI. Modularita systému je zajištěna sofistikovaným návrhem provázání vstupních dat, generátorů a uživatelem definovaných datových typů. Jádro modulárního generátoru úloh je opatřeno jednoduchou grafickou nadstavbou pro demonstraci funkčnosti systému, systém je však připraven pro použití jako zásuvný modul do aplikace, která bude umožňovat generování libovolného počtu úloh a jejich transformaci z QTI do jiných v současné době používaných formátů, například Adobe PDF pro tisk, nebo HTML pro užití v internetových aplikacích.

Abstract

BALON, M. *Design and implementation of a modular task generator*. Bachelor's thesis. Pilsen: Faculty of Economics UWB Pilsen, 53 s., 2012

Key words: XML, QTI, parser, programming language, Java, task generator

Presented thesis focuses on design and implementation of a software tool for use in education. It is universal and modular task generator that generates output data on the basis of input data. System uses the latest technologies as XML, XSD, object-oriented programming language Java and JAXB technology. Output of this system is XML document that corresponds to QTI specification. Modularity of system is achieved by a sophisticated design of interconnections between input data, their generators and data types defined by the user. Kernel of the modular task generator is equipped with simple graphical user interface for demonstration of system functionality, but is prepared for use as a plug-in for other applications. This application will allow generate any number of tasks and their transformation from QTI to other formats that are used these days, for example Adobe PDF to print or HTML to use in web applications.