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

Diplomová práce

Deklarativní jazyk a nástroj pro transformaci dat mezi XML dokumenty

Plzeň 2021 Bc. Jan Jirman

ZÁPADOČESKÁ UNIVERZITA V PLZNI

Fakulta aplikovaných věd

Akademický rok: 2020/2021

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Bc. Jan JIRMAN**
Osobní číslo:	**A18N0089P**
Studijní program:	**N3902 Inženýrská informatika**
Studijní obor:	**Softwarové inženýrství**
Téma práce:	**Deklarativní jazyk a nástroj pro transformaci dat mezi XML dokumenty**
Zadávající katedra:	**Katedra informatiky a výpočetní techniky**

Zásady pro vypracování

1. Seznamte se s požadavky na mapování a transformaci dat a formáty příslušných XML dokumentů používaných ve firmě Eurosoftware s.r.o.
2. Prostudujte přístupy, jazyky a nástroje vhodné pro mapování/transformaci dat mezi XML dokumenty z bodu 1.
3. Na základě bodu 2 navrhněte deklarativní jazyk vhodný pro mapování dat v XML dokumentech z bodu 1.
4. Dle předchozích bodů navrhněte a implementujte nástroj (v jazyce Java s podporou pro Maven) pro transformaci dat mezi XML dokumenty.
5. Navrhněte a implementujte nástroj, který vygeneruje dokumentaci k dané transformaci (HTML /PDF).
6. Vytvořte grafické uživatelské rozhraní pro vizualizaci mapování/transformací popsaných výše (tento bod je volitelný).
7. Výsledné řešení otestujte a zhodnoťte.

Rozsah diplomové práce: **doporuč. 50 s. původního textu**
Rozsah grafických prací: **dle potřeby**
Forma zpracování diplomové práce: **tištěná**

Seznam doporučené literatury:

dodá vedoucí diplomové práce

Vedoucí diplomové práce: **Ing. Roman Mouček, Ph.D.**
Katedra informatiky a výpočetní techniky

Datum zadání diplomové práce: **11. září 2020**
Termín odevzdání diplomové práce: **20. května 2021**

L.S.

Doc. Dr. Ing. Vlasta Radová
děkanka

Doc. Ing. Přemysl Brada, MSc., Ph.D.
vedoucí katedry

V Plzni dne 24. září 2020

Prohlášení

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

V Plzni dne 24. června 2021

Bc. Jan Jirman

Abstract

The main topic of the master's thesis is the design and implementation of a domain specific language suitable for mapping data in XML documents. The thesis first deals with the collection of information from which it is possible to create domain concepts. Based on the domain concepts, the syntax and semantics of the language is proposed. Along with the language, a library is developed that provides methods for transforming data between two XML files. This is followed by the implementation of a generator that can generate executable Java code with an already concrete data mapping. The result of the work is a usable domain-specific language that has been evaluated as user-friendly by selected users during testing and has the potential to replace third-party commercial mapping software at Eurosoftware.

Abstrakt

Hlavním tématem diplomové práce je návrh a implementace doménově specifického jazyka, který bude vhodný pro mapování dat v XML dokumentech. Práce se nejdříve zabývá shromážděním informací, ze kterých je možné vytvořit doménové koncepty. Na základě doménových konceptů je navržena syntaxe a sémantika jazyka. Společně s jazykem je vyvíjena knihovna, která poskytuje metody pro transformaci dat mezi dvěma XML soubory. Následuje implementace generátoru, který dokáže vygenerovat spustitelný Java kód s již konkrétním mapováním dat. Výsledkem práce je použitelný doménově specifický jazyk, který byl vybranými uživateli během testování zhodnocen jako uživatelsky přívětivý a má potenciál nahradit ve firmě Eurosoftware komerční mapovací software třetí strany.

Poděkování

Rád bych poděkoval *Ing. Radkovi Hoštičkovi* za čas, který mi věnoval při řešení dané problematiky. Dále chci poděkovat své rodině a svému nejbližšímu okolí za trpělivost a podporu při studiu. Děkuji všem, kteří si našli čas k otestování vytvořeného jazyka.

Obsah

1 Úvod

Pokud se rozhodneme pro vznik nového programovacího jazyka, pak očekáváme, že bude mít pro nás přínos [17]. Tvorba doménově specifických jazyků (zkráceně DSL) je oblíbená zejména proto, že zlepšují produktivitu u vývojářů a zároveň také zlepšují komunikaci s doménovými experty [11]. S dobře navrženou úrovní abstrakce lze kódu přirozeně porozumět [17] a může tak nahradit obecný nepřehledný kód za většinou kratší DSL kód. Jazyk často obsahuje názvy z konkrétní domény, a proto je i komunikace mezi zainteresovanými stranami jednodušší [11].

Ve firmě *Eurosoftware*, která je tvůrcem tohoto zadání, se momentálně pro mapování XML souborů používá grafický software Altova MapForce. S tímto softwarem pracuje více uživatelů z různých oddělení - od programátorů až po konzultanty - a každý má jiné zkušenosti. Právě proto vznikla primárně myšlenka, aby vznikl doménově specifický jazyk, který by byl pro všechny stejný, lehce naučitelný a zároveň, aby umožňoval některé funkce, které obsahuje grafický software.

Cílem této práce je vytvořit doménově specifický jazyk, který umožní uživatelům napsat vlastní mapování XML souborů bez použití Altova MapForce. Práce se zaměřuje na návrh jazyka i jeho implementaci. Souběžně s jazykem se práce zabývá implementací knihovny, která poskytne metody k transformování dat. Očekává se, že tento jazyk bude pro firmu přínosem, protože odpadnou problémy s grafickým softwarem a s mergováním vytvořených mapování.

2 Doménově specifické jazyky

Doménově specifické jazyky (zkráceně **DSL**) jsou programovací nebo specifikační jazyky [7], které jsou optimalizované pro danou třídu problémů, které se říká doména [31]. Na rozdíl od obecných jazyků (**GPL**), jako je Java nebo C, není účelem poskytovat obecná řešení mnoha problémů [7]. Jestliže máme doménu pokrytou konkrétním DSL [7], zvolením tohoto specifického přístupu můžeme dostat mnohem lepší řešení než použitím obecného jazyka [9]. DSL obsahují syntaxi a sémantiku, která modeluje koncepty na stejné úrovni abstrakce[1] jako problémová doména [12].

Mnoho vývojářů vytvořilo různé DSL a vznikly i známé jazyky jako Lex, PostScript, SQL [28], CSS, HTML nebo Mathematica [12].

2.1 Porovnání DSL s GPL

GPL umožňují obecná řešení mnoha problémů, ale realizace nemusí být vždy optimální. Ve srovnání s GPL poskytují DSL lepší řešení, ale pro výrazně menší množinu problémů [9].

2.1.1 Turingovská úplnost

GPL jsou obecné a turingovsky úplné. Znamená to, že je lze použít k implementaci čehokoli, co je Turingovým strojem vypočitatelné [31]. Existují i DSL, které mají tuto vlastnost. Jsou to například XSLT a PostScript [27].

Turingovská úplnost není zásadním kritériem vzniku DSL, a proto tyto jazyky často nejsou turingovsky úplné. Těmito jazyky lze vytvořit pouze některé programy - DSL jsou v tomto smyslu omezenější než GPL, a nejsou tak flexibilní. V některých případech lze vytvořit pouze správný program, což znamená, že uživateli není umožněno udělat chybu (*correct-by-construction*) [31].

Čím více je jazyk omezen, tím více může uživatel postrádat známé abstrakce obecného jazyka. K jazyku lze doplnit například aritmetiku, iteraci, struktury a další. Při rozšiřování DSL se jazyk stává méně doménově specifickým a více připomíná obecný [19]. V případě, že vznikne DSL tak

[1] **Abstrakce** - kognitivní proces lidského mozku, který nám umožňuje se soustředit na základní aspekty předmětu a ignorovat zbytečné detaily [12].

obecný jako Java, je potřeba zvážit, který jazyk se vyplatí použít. Existují i jazyky, které vykazují charakteristické znaky DSL i GPL. Například SQL nebo HTML jsou optimalizované (a limitované) pro konkrétní problémovou doménu, řadíme je tedy k DSL, zároveň však pozorujeme některé charakteristické rysy GPL [31].

2.1.2 Expresivita

DSL zvyšují míru expresivity notace, čímž snižují úroveň její obecnosti. Ve srovnání s GPL vykazují větší expresivitu [18], což znamená, že mohou vyjadřovat:

- Stejné věci jako jiné jazyky, ale stručněji.
- Věci, které ostatní jazyky nemohou vyjádřit [19].

DSL poskytují notace pro konkrétní doménu [18] a pokud se rozhodneme takový jazyk použít, řešíme pouze složitost domény. Pokud má navíc jazyk odpovídající úroveň abstrakce [12], mohou jej používat i lidé, kteří nejsou zkušenými programátory. Například doménoví experti dostanou vlastní produkční prostředí, jež jim umožní pracovat s jazykem, který je úzce spojen s jejich doménou [31]. Pro matematiky je uživatelsky vstřícný jazyk Mathematica, pro UI designery je vhodným jazykem HTML. Tyto jazyky tak přispívají ke zvýšení produktivity při vývoji softwaru a zároveň snižují požadavky na programátorské znalosti [5].

Martin Fowler ve své prezentaci *Introduction to Domain Specific Languages* na konferenci JAOO uvedl následující srovnání jazyků DSL a GPL (výpisy kódů 2.1 a 2.2):[2]

```
mapping('SVCL',ServiceCall) do
  extract 4..18, 'CustomerName'
  extract 19..23, 'CustomerID'
  extract 24..27, 'CallTypeCode'
  extract 28..35, 'DateOfCallString'
end
```

Výpis kódu 2.1: Ukázka řešení doménově specifickým jazykem (převzato z prezentace [2]).

Obě řešení budou fungovat - rozdíl je pouze v použitém jazyce. První kód (výpis kódu 2.1) je napsán vlastním DSL a na první pohled se zdá být

[2]Introduction to Domain Specific Languages. [online]. Copyright © 2006 C4Media. Dostupné z: `https://www.infoq.com/presentations/domain-specific-languages/`

```
public void Configure(Reader target){
  target.AddStrategy(ConfigureServiceCall());
}
private ReaderStrategy ConfigrureServiceCall(){
  ReaderStrategy result;
  result = new ReaderStrategy("SVCL", typeof(ServiceCall));

  result.AddFieldExtractor(4, 18, "CustomerName");
  result.AddFieldExtractor(19, 23, "CustomerID");
  result.AddFieldExtractor(24, 27, "CallTypeCode");
  result.AddFieldExtractor(28, 35, "DateOfCallString");
  return result;
}
```

Výpis kódu 2.2: Ukázka řešení obecným jazykem (převzato z prezentace [2]).

čitelnější a pochopitelnější než druhý kód (výpis kódu 2.2), který je napsán v Javě. Uživatel tedy vytvoří program v DSL i v případě, že nemá programátorské zkušenosti.

Deklarativní programování

V souvislosti s úrovní abstrakce můžeme slyšet pojem deklarativní programování [27]. Opakem je imperativní programování, se kterým se setkáváme častěji - máme sekvenci příkazů, nebo instrukcí, které mění stav programu [31]. Touto sekvencí instruujeme počítač, co má přesně vykonat [11, 29]. Výhodou tohoto přístupu je, že ladění imperativních programů je jednoduché, protože zahrnuje krokování instrukcí a sledování změn stavu [31].

Deklarativním programováním vyjadřujeme, čeho chceme dosáhnout [29], ale neurčujeme, jak je řešení nalezeno [31]. Necháme systém rozhodnout, jak dosáhne výsledku [29]. Tento přístup se dá vysvětlit na *Kowalského rovnici*: **algoritmus = logika + řízení**, kdy při deklarativním programování specifikujeme *logiku algoritmu*, ale už ne nutně *řízení* [1].

Doménově specifické jazyky jsou často deklarativní, protože využívají zvýšené míry expresivity [9].

2.2 Rizika a příležitosti

Vytvoření vlastního DSL zahrnuje rizika i příležitosti [9]. Dobře navržený jazyk dokáže těžit z mnoha výhod. Ty jsou:

- **Účast doménových odborníků**: Tím, že je notace a abstrakce úzce spjata s konkrétní doménou, umožňuje DSL dobrou integraci mezi vývojáři a doménovými experty. Doménoví experti mohou číst nebo psát programy, aniž by byli zatíženi implementačními detaily, které pro ně nejsou relevantní (viz kapitola 2.1.2).
- **Produktivita**: Dlouhý GPL kód nahrazujeme kratším DSL kódem [31].
- **Kvalita**, s níž souvisí **spolehlivost** a **udržovatelnost** [9]: Protože je výsledný produkt psaný v DSL přehlednější a jednodušší, měl by se lépe udržovat a zároveň by měl obsahovat méně chyb. Kvalita produktu se tedy zvyšuje. Pokud je DSL dobře navržený, programátoři nemají při psaní kódu přílišnou volnost, a můžeme tedy zabránit duplikaci kódu nebo umožnit uživateli psát pouze správné programy (viz kapitola 2.1.1).
- **Validace a optimalizace** : Implementace analýz je jednodušší, chybové hlášky jsou srozumitelnější, protože mohou používat koncepty domény [31].

Existují také rizika, na které je nutné si dát pozor:

- **Náklady**: Pokud teprve budeme vytvářet DSL, je potřeba počítat s náklady na návrh jazyka a jeho implementaci. Dále mohou být náklady spojené s jeho údržbou a zaškolením všech uživatelů [9]. Pokud už je DSL vytvořený, je výhodné ho použít, čímž se například finanční náklady eliminují [31].
- **Rozsah DSL**: Někdy může být problém najít správný rozsah jazyka. Jedná se hlavně o hranici mezi doménovou specifičností a konstrukcemi obecného jazyka (viz kapitola 2.1.1).
- **Omezená dostupnost**: Jazyk je dostupný pouze pro konkrétní doménu [9].
- **Mnoho DSL**: Jakmile se vývoj DSL stane technicky snadným, existuje nebezpečí, že vývojáři místo hledání již použitých DSL vytváří nové. Může se stát, že následně vznikne sada jazyků, které se vzájemně v dané doméně překrývají [31].

2.3 Interní a externí DSL

Doménově specifické jazyky jsou klasifikovány na základě způsobu jejich implementace [11]. Při návrhu jazyka se rozhodujeme, jestli vytvoříme interní, nebo externí DSL [20].

2.3.1 Interní DSL

Interní DSL již budujeme nad existujícím obecným jazykem (hostitelským jazykem). Hostitelský jazyk je často dynamicky typován a implementace DSL je založena na metaprogramování[3] [31]. Pokud si vybereme například hostitelský jazyk Ruby, pak kód interního DSL je stále Ruby, ale napsaný vlastním stylem - jazyk je často uzpůsobený pro účely konkrétní domény [11]. Často se takovým DSL říká Fluent API,[4] protože je takové DSL ve skutečnosti jen určitý druh API [11].

Fluent API zřetězuje volání metod. Každé volání metody vrací objekt, pro který je možné zavolat další metody. Ve výpisu kódu 2.3 lze vidět řetězení metod - takto napsaný kód je čitelnější [31].

```
Foo fooMock = EasyMock.createMock(Foo.class);
EasyMock.expect(foo.bar()).andReturn(new
                  BarWithParametersResults()).atLeastOnce();
```

Výpis kódu 2.3: Fluent API (převzato z [5]).

Syntaxe u interních DSL je založena na hostitelském jazyku. U některých hostitelských jazyků neexistuje žádný způsob, jak tuto syntaxi změnit. Některé jazyky s flexibilnější syntaxí mohou podporovat interní DSL, které více připomínají vlastní jazyky. Ve výpisu kódu 2.4 je příklad jazyka *Ruby on Rails* [31].

```
class Post < ActiveRecord::Base
  validates :name, :presence => true
  validates :title, :presence => true,
    :length => { :minimum => 5 }
end
```

Výpis kódu 2.4: Příklad jazyka Ruby on Rails (převzato z [31]).

2.3.2 Externí DSL

Rozdíl mezi externími a interními DSL je spojen s omezením syntaxe. Při návrhu interního DSL se řídíme syntaxí obecného jazyka. Nemůžeme použít klíčová slova, která jsou již vyhrazena v hostitelském jazyku nebo není

[3] **Metaprogramování** - psaní kódu, který generuje kód. Generování kódu může proběhnout při běhu programu nebo v době kompilace [12].

[4] DSL Q & A. martinfowler.com [online]. Copyright © Martin Fowler [cit. 08.05.2021]. Dostupné z: `https://martinfowler.com/bliki/DslQandA.html`

[5] JUnit and EasyMock [online]. Copyright © Michael T Minella. Dostupné z: https://lfm.iti.kit.edu/download/Junit_Easymock_rc028-010d-junit_0.pdf

možné použít určitou syntaxi, protože je v hostitelském jazyku neplatná. U externích jazyků tento problém nenastane, protože si definujeme vlastní jazyk - vymezíme si svá pravidla, která chceme v daném jazyce mít [23]. Externí DSL mají často vlastní syntaxi, ale je běžné použít i syntaxi jiného jazyka (např. XML) [11]. Script v externím DSL je obvykle parsován a převeden do obecného jazyka použitím kompilátoru nebo interpreta [11, 15].

Příkladem může být jazyk **BPEL (Business Process Execution Language)**: XML kód je parsován a transformován do datových struktur programovacího jazyka, jako je Java [15]. Typicky se k výrobě externích DSL používají různé nástroje, jako je například XText framework, OSLO framework [26]. **XText framework** je použit v této práci (více v kapitole 4).

2.4 Metodika vývoje DSL

Jak zjistit, co přesně bude DSL vyjadřovat? Jaké jsou relevantní notace a abstrakce? Takové otázky jsou jedny z klíčových při vývoji doménově specifického jazyka. Aby vzniklo kvalitní DSL, potřebujeme mnoho doménových znalostí a myšlení. Hlavním problémem je, že se nesnažíme pochopit pouze jeden problém, ale třídu problémů. K porozumění třídě problémů můžeme vynaložit velké úsilí [31].

Důležitým aspektem je vývoj DSL. U vývoje jazyka potřebujeme iterovat, protože tím snížíme riziko selhání. Místo toho, abychom vytvořili velký projekt, který až na konci poskytne funkční DSL, tak iterativním procesem vytvoříme za kratší dobu již použitelné DSL, které pokryje některé poddomény [30]. Začneme vývojem DSL pro vybrané problémy, kterým jsme hluboce porozuměli. Vytvoříme část jazyka, generátoru a poté ověříme na ukázkovém modelu, zda je vše správně implementováno [31]. Zpětná vazba od uživatelů (např. od doménových expertů) je nesmírně cenná, protože se každým hodnocením dá vylepšit návrh abstrakcí. Každou iterací zvyšujeme pokrytí dané problémové domény [30].

3 Životní cyklus DSL

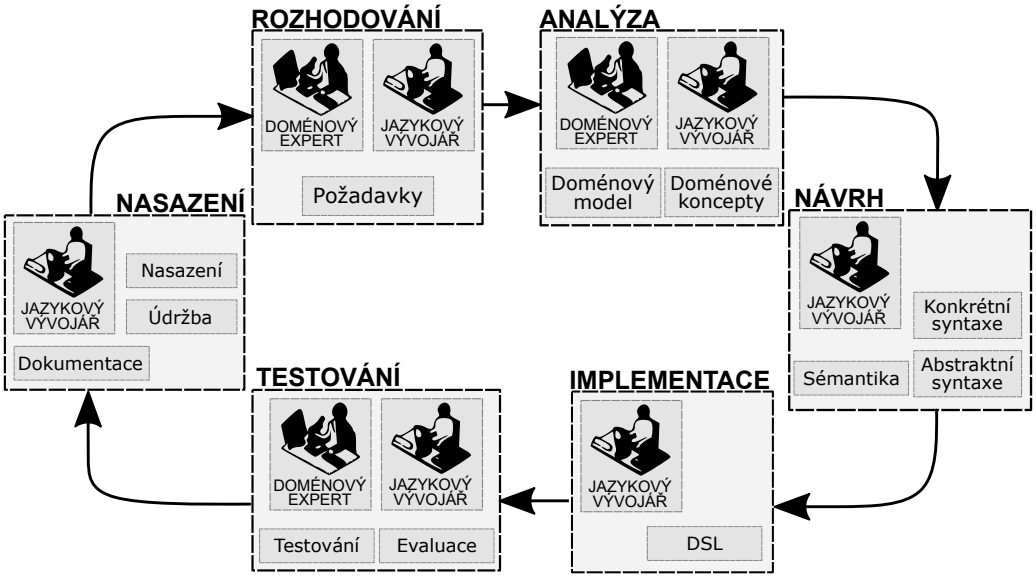
Životní cyklus, tak jak je popsán v článku od *Marjana Mernika* a kol. [18], sestává z pěti fází.

1. **Rozhodování**: rozhodujeme se, jestli vytvořit DSL či nikoli.
2. **Analýza**: analyzujeme aplikační doménu.
3. **Návrh**: navrhujeme architekturu a jazyk.
4. **Implementace**: vyvíjíme DSL a pomocné run-time systémy.
5. **Nasazení**: nasazení DSL a aplikací, které jazyk používají, do produkčního prostředí.

Eelco Visser ve svém článku [30] doplnil k fázi nasazení také údržbu DSL [30].

Dále v disertační práci, kterou napsala *Ankica Barisic* [6], a v článku od stejné autorky a kol. [4] je zmíněno, že v životním cyklu DSL chybí nejenom testování, ale i evaluace jazyka. Tato fáze by měla být před fází nasazení, protože zjišťujeme kvalitu implementovaného jazyka [4].

Všechny fáze životního cyklu DSL lze vidět na obrázku 3.1.



Obrázek 3.1: Životní cyklus DSL (převzato a modifikováno z [3, 6]).

3.1 Fáze rozhodování

Cílem této fáze je identifikovat potřebu vytvořit nový doménově specifický jazyk pro určitou doménu [6]. To zahrnuje i odůvodnění, že investice do vývoje DSL včetně nasazení se vrátí s následným ekonomičtějším vývojem nebo údržbou softwaru [18].

Rozhodnutí, že vytvoříme nové DSL, není obvykle dost snadné. V praxi se můžeme setkat s tím, že nepříliš promyšlené úvahy nebo nedostatek odborných znalostí domény způsobí, že se plány odloží na dobu neurčitou [18]. Aby bylo možné učinit rozhodnutí, musí zainteresované strany (*stakeholders*) - doménoví experti a jazykoví vývojáři - projednat požadavky domény [6]. Dále bereme v úvahu, jestli již existuje pro doménu nějaký jazyk a jestli je dobře zdokumentován. Pokud by takový jazyk existoval, je méně nákladnější ho použít než vytvářet nový, protože bychom potřebovali k tomu mnohem méně odborných znalostí [18].

3.1.1 Stakeholders

Typickými stakeholdery u DSL jsou:

- **Jazykoví vývojáři**, kteří jsou zodpovědní za výběr nebo implementaci odpovídajícího DSL [24, 31].
- **Doménoví experti**, kteří komunikují prostřednictvím doménového slovníku a svou doménu znají [12].
- **Uživatelé DSL**, kteří používají doménově specifický jazyk k tvorbě aplikací [6].

Někdy se stane, že uživatelé DSL jsou vývojáři jazyka. V těchto případech neexistuje mezera ve znalostech, protože rozumí problémům domény a zároveň vyvíjí konkrétní jazyk [31].

3.1.2 Požadavky

Požadavky mají vývojářům říci, co by měl přesně systém dělat. Nicméně požadavek už nepředepisuje, jakým způsobem danou funkcionalitu implementovat: architektura, návrh, volba technologie a jazyka je už na vývojáři [31].

Požadavky jsou prostředkem komunikace mezi lidmi, kteří ví, jak by měl systém pracovat, a lidmi, kteří systém vyvíjí. S tím jsou spojená rizika, že:

- Ti, kteří požadavky implementují, mohou mít jinou představu než ti, kteří požadavky píšou - zvyšuje se tak pravděpodobnost, že dojde k nedorozumění mezi skupinami.
- Ti, kteří požadavky píšou, nemusí ve skutečnosti vědět, co by měl přesně systém dělat (alespoň ze začátku). Požadavky se v průběhu vývoje mění [31].

Požadavky, které souvisí s DSL, jsou aplikovatelné i na obecné modelovací a programovací jazyky. Zde je výčet několika požadavků, které mohou vzniknout při této fázi životního cyklu DSL:

- **Shoda**: jazykové konstrukce musí odpovídat důležitým doménovým konceptům.
- **Integrovatelnost**: jazyk a jeho nástroje lze integrovat s jinými jazyky a nástroji.
- **Rozšiřitelnost**: jazyk je možné rozšířit o další konstrukce.
- **Jednoduchost**: jazyk by měl být co nejjednodušší, aby uživatelům usnadnil práci. Tento požadavek je obecně žádoucí pro jakýkoli jazyk.
- **Kvalita**: jazyk bude poskytovat obecné mechanismy k vytvoření kvalitního systému [24].

Požadavky a návrh

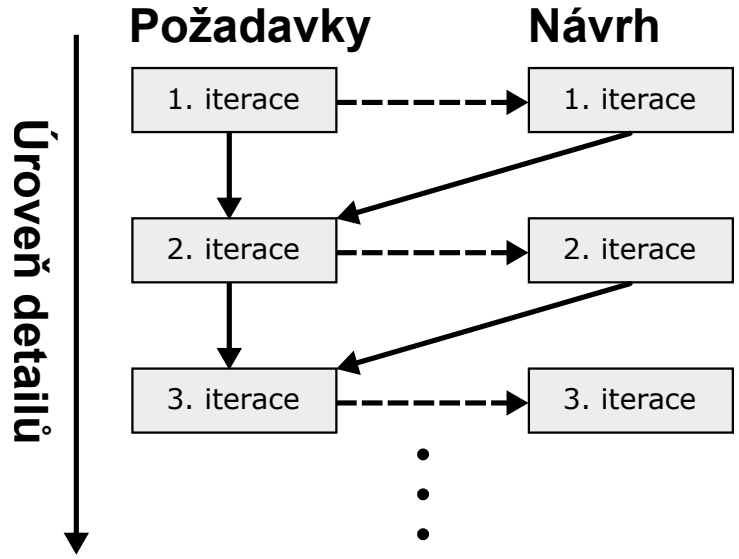
Tradičně se snažíme stanovit hranici mezi požadavky, návrhem a implementací [31].

Příklad: Požadavek může uvádět, že systém by měl být z 99.99 % spolehlivý. Pak by návrh mohl být, že k dodržení dané spolehlivosti použijeme horkou zálohu. V dalším kroku bychom implementovali konkrétní pohotovostní technologii [31].

Z praxe je známo, že musíme rozvíjet požadavky postupně - princip je následující:

1. **napíšeme požadavky**,
2. **vytvoříme prototyp**,
3. **ověříme požadavky, jestli dávají smysl**, poté se vracíme na 1. krok, ale požadavky více upřesníme [31].

Tento přístup je znázorněn na obrázku 3.2, kde je vidět, že úroveň detailů požadavků se zvyšuje s počtem iterací. Požadavky a návrh se navzájem ovlivňují, a proto je nejlepší je provádět iterativně a paralelně [31]. Obecně iterativní přístup byl rozebrán v kapitole 2.4.



Obrázek 3.2: Požadavky a návrh (převzato a modifikováno z [31]).

3.2 Fáze doménové analýzy

Doménové analýze se jinak říká *doménové modelování*. V této fázi shromažďujeme všechny informace a znalosti domény, abychom ji mohli dostatečně pochopit [4]. Analyzujeme základní vlastnosti a požadavky [30]. Doménoví experti pomáhají jazykovým vývojářům definovat hlavní koncepty domény, feature modely (v češtině by se dalo přeložit jako *modely vlastností*), funkční a technické požadavky [3, 6].

Příklad 1: První analýza domény by nás informovala o tom, že například vývoj webové aplikace zahrnuje datový model, objektově relační mapování, rozhraní, vstup a výstup, ověření dat a kontrola přístupu (*access control*). Důkladnější analýza podrobněji studuje konkrétní problém domény a stanoví terminologii a požadavky, které jsou vstupem pro návrh DSL [30].

3.2.1 Problémová doména

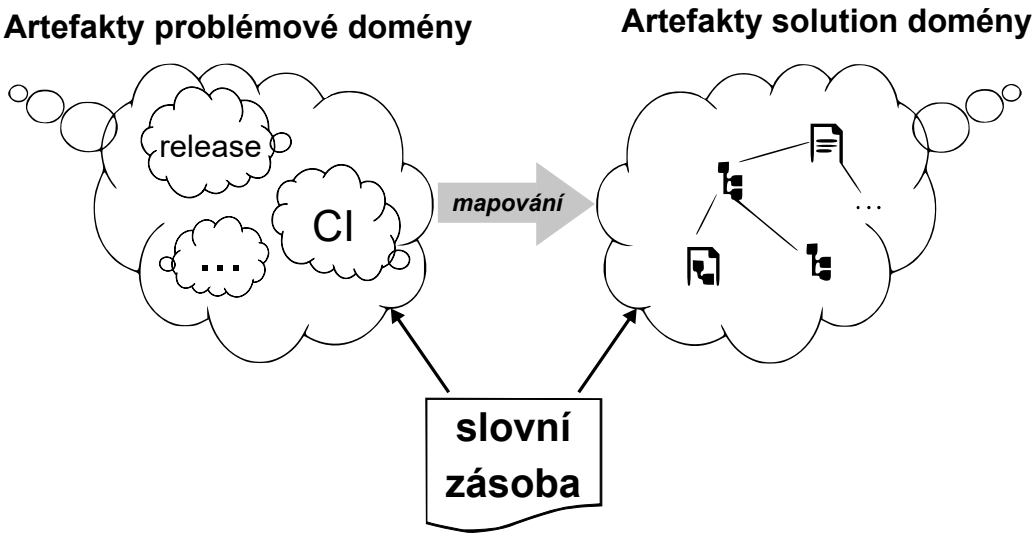
Při analýze zohledňujeme doménové termíny, výrazy a omezení, která jsou neodmyslitelnou součástí problému - jedná se o **problémovou doménu** [6].

Příklad 2: Doménový expert zanalyzoval problémovou doménu, identifikoval požadavky a přišel s nezbytnými jazykovými konstrukcemi, které by měly být v DSL. V tab. 3.1 jsou některé zobrazeny (příklad včetně tabulky je z knihy *DSLs in Action*) [12].

Doménový koncept	**Detaily**
Nová objednávka	• *Množství* by mělo být povinné. • Je třeba specifikovat, zda *koupit* nebo *prodat*.
Cena objednávky	• Nutné zmínit *jednotkovou cenu*.
....	....

Tabulka 3.1: Příklad doménových konceptů (převzato a modifikováno z [12]).

Pokud jsme identifikovali problémovou doménu, můžeme definovat prostor pro řešení (anglicky *solution domain* nebo *solution space*). Tento prostor poskytuje všechny nástroje, které můžeme použít k řešení problému [23]. Cílem je namapovat komponenty z problémové domény na příslušné techniky v *solution domain*. Pro usnadnění komunikace je vhodné zavést společnou slovní zásobu - doménovou terminologii. Mapování i sdílenou slovní zásobu lze vidět na obrázku 3.3.



Obrázek 3.3: Sdílená slovní zásoba a mapování (převzato a modifikováno z [12, 23])

.

Doménová terminologie

Rozdíly v terminologii používané různými stakeholdery brání ve smysluplné spolupráci [13]. Efektivnější přístup je standardizovat terminologii popisující doménu, vytvořit slovník [16] a sdílet jej mezi všemi stakeholdery [12] (viz obr. 3.3). Slovník je výsledkem shromažďování informací ve všech fázích analýzy [16].

Jednotná terminologie je pak použita napříč odvětvími: Tester pojmenuje testovací případy (*test case*) již domluvenou slovní zásobou. Programátor použije stejné názvy při pojmenování abstrakcí. Datový architekt využije slova ze slovníku při navrhování datových modelů [13]. S touto jednotnou terminologií lze vystopovat artefakt z problémové domény a najít k němu odpovídající artefakt ze *solution domain*.

3.2.2 Vstupy a výstupy analýzy

Před analýzou máme různé zdroje znalostí, např.: znalosti doménových expertů, GPL kód, průzkumy zákazníků [18], existující systémy a jejich artefakty (uživatelské příručky, dokumenty specifikace požadavků a další) a standardy [4].

Výstupy doménové analýzy se můžou lišit, ale v zásadě se skládá z terminologie a sémantiky, která je specifická pro danou doménu [18]. Konkrétním výstupem je doménový model, který představuje vlastnosti systému v doméně [6]. Doménový model se skládá z:

- definice domény (vymezení rozsahu),
- doménové terminologie (slovník),
- popisů doménových konceptů,
- feature modelů popisující společné rysy a variabilitu doménových konceptů (a vzájemné závislosti) [18].

3.3 Fáze návrhu

DSL se obvykle skládá ze tří základních prvků: abstraktní syntaxe, konkrétní syntaxe a sémantiky [25]. Po fázi doménové analýzy následuje fáze návrhu jazyka, kde tyto části definujeme [3].

Při návrhu DSL musí být učiněno několik rozhodnutí, která se týkají syntaxe nebo sémantiky. DSL může být například navrženo úplně od začátku, nebo

na základě již existujícího jazyka [6]. Rozhodnutí, zda jazyk bude interní nebo externí, bude mít dopad na ostatní aspekty jazyka. Výběr mezi těmito dvěma možnostmi vyžaduje pečlivé zvážení všech výhod a nevýhod [25].

3.3.1 Abstraktní syntaxe

Abstraktní syntaxe (tj. meta-model) neobsahuje notační detaily, jako jsou klíčová slova, symboly, prázdné znaky nebo pozice, velikosti a zbarvení v grafických notacích [31]. Vyjadřuje vztahy mezi jazykovými konstrukcemi [25]. Používá se pro analýzu a následné zpracování programů [31].

3.3.2 Konkrétní syntaxe

Konkrétní syntaxe definuje konkrétní DSL notaci [25]. S touto notací se uživatel setká při psaní kódu programu [31]. Obvykle je jedná o textovou nebo grafickou notaci [25], ale bývají i symbolické, tabulkové nebo jakékoli jejich kombinace [31].

Časté otázky při návrhu konkrétní syntaxe: Vyžaduje DSL specializovanou syntaxi? Je syntaxe hostitelského jazyka vhodná pro DSL? Kolik úsilí je potřeba vynaložit k tvorbě interního DSL ve srovnání s vytvořením DSL od úplného začátku [25]?

V této práci jsem se rozhodl vytvořit externí DSL, protože ve fázi analýzy jsme spolu se zadavatelem vybrali nástroj Xtext, který podporuje tvorbu externího DSL. Rozhodnutí zejména ulehčil fakt, že zaměstnanci ve firmě aktivně používají již několik let doménově specifický jazyk, který byl vytvořen ve zmíněném nástroji (více o Xtextu v kapitole 4).

3.3.3 Gramatika

Pokud pracujeme se syntaxí jazyka, musíme se zaměřit na gramatiku [11, 23]. Gramatika je sada přepisovacích pravidel, která popisuje, jak je proud (*stream*) textu převeden do syntaktického stromu [11]. Každé pravidlo popisuje, jak vypadá platný textový vstup (*věta* nebo *slovo*) [31]. Pravidly například určíme znak pro nový řádek, strukturu funkce nebo jak provádět různé operace. Syntaxe je přímo odvozená z gramatiky - gramatikou definujeme pravidla a syntaxe je to, co máme po implementaci těchto pravidel [23].

Bezkontextová gramatika

Bezkontextová gramatika je definována: **G = {T,N,P,S}**, kde:

- T: konečná množina terminálních symbolů (neobsahuje prvky z N),
- N: konečná množina neterminálních symbolů,
- P: konečná množina přepisovacích pravidel,
- S: počáteční symbol, který se používá k reprezentaci celé věty jazyka.

Pravidlo má následující formu: A $\rightarrow$ α. Neterminální symbol je A a na pravé straně je α, což je řetězec terminálních (A $\rightarrow$ a), neterminálních (A $\rightarrow$ B), nebo kombinace obou typů symbolů (A $\rightarrow$ aB) [22].[1] Přepisovacím symbolem je „$\rightarrow$“ [14].

Backus-Naurova forma

K definici gramatiky běžně používáme notaci zvanou **Backus-Naurova Forma** nebo zkráceně BFN. Jedná se o notační techniku, která definuje jazyk v bezkontextové gramatice (viz ukázka kódu 3.1) [23]. Přepisovací pravidla jsou psána touto formou: **S ::= $\mathbf{P_1}$...$\mathbf{P_n}$**. Symbol `S` je definován jako řada výrazů P_1 až P_n [31].

Z příkladu 3.1 si vezmeme počáteční přepisovací pravidlo.

```
<postal-address> ::= "letter" <name-part> <street-address>
```

Toto pravidlo obsahuje 3 výrazy (`S ::=` P_1 P_2 P_3). Tyto výrazy:

- **Odkazují na jiný symbol** [31]:
  Jedná se o výrazy P_2 a P_3. V příkladu se odkazuje na neterminální symboly: `<name-part>` a `<street-address>`.
- **Jsou konkrétní** [31]:
  Jedná se o výraz P_1, který je v příkladu 3.1 klíčovým slovem („`letter`“).

```
1 <postal-address> ::= "letter" <name-part> <street-address>
2      <name-part> ::= <personal-part> <last-name> <EOL>
3                    | <personal-part> <name-part>
4  <personal-part> ::= <initial> "." | <first-name>
5 <street-address> ::= <house-num> <street-name> <EOL>
```

Výpis kódu 3.1: Ukázka BNF (převzato a modifikováno z [23]).

[1] Formal Languages And Automata Models of Computation, Computability Basics of Recursive Function Theory. Copyright © 2010 Jean Gallier [online]. Dostupné z: https://www.cis.upenn.edu/ jean/gbooks/tcbookpdf2.pdf

Pokud pro symbol existuje více variant, lze tyto varianty zapsat jako samostatná pravidla, anebo varianty oddělit v jednom pravidlu znakem „|“ (v příkladu 3.1 u symbolů `<personal-part>` a `<name-part>`) [31].

Extended BNF (EBNF): Jedná se o několik jednoduchých rozšíření BNF, díky nimž je vyjádření gramatik pohodlnější [31]. Často jsou odvozená od syntaxe regulárních výrazů:

- **Znak „?“** (0 nebo 1 výskyt):
  příklad: `<expr> ::= "-"?`.
- **Znak „*“** (0 nebo více výskytů):
  příklad: `<expr> ::= <digit>*`.
- **Znak „+“** (1 nebo více výskytů):
  příklad: `<expr> ::= <digit>+`.
- **Použití závorek pro seskupování**:
  příklad: `<expr> ::= "-"? <digit>+ ("." <digit>*)?`.[2]

3.3.4 Parser

Jestliže máme syntaxi připravenou, musíme pro ni vyvinout **parser** [12]. Gramatika určuje posloupnost tokenů a slov, která tvoří strukturálně platný program. Parser následně rozpoznává validní programy v jejich textové podobě a vytváří abstraktní syntaktický strom [31] (převede text na strukturu v paměti [23]).

Lexikální analýza

Nejdříve musí být celý kód rozdělen na tokeny [7, 12]. Token může být klíčové slovo, operátor, identifikátor, literál nebo komentář [31].

```
1 James Smith (50)
2 John Anderson (40) employed
```

Výpis kódu 3.2: Ukázka DSL scriptu (převzato z [7]).

V příkladu 3.2 lze rozdělit text na následující tokeny:

- klíčové slovo: `employed`,
- operátory: „`)`“ a „`(`“
- textové literály: `James`, `Smith`, `John`, `Anderson`

[2] BNF and EBNF - prezentace [online]. Copyright © DePaul University. Dostupné z: `https://condor.depaul.edu/ichu/csc447/notes/wk3/BNF.pdf`

- celočíselné literály: `50`, `40` [7].

Převod sekvence znaků na tokeny se nazývá lexikální analýza. Program, který tento proces provádí, se jmenuje lexikální analyzátor [7]. Lexikální analýzu můžeme definovat ve stylu přepisovacích pravidel. Tato lexikální pravidla jsou dost podobná regulárním výrazům, protože implementace lexikálního analyzátoru používá obvykle API pro regulární výrazy [11].

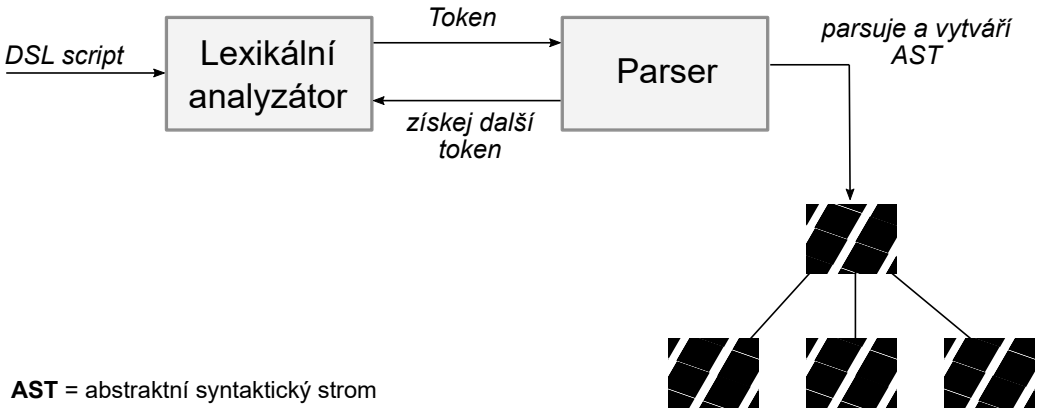
```
1 Identifier:
2   ('a'..'z' | 'A'..'Z')
3   ('a'..'z' | 'A'..'Z' | '0'..'9' | '_')*
4 ;
```

Výpis kódu 3.3: Lexikální pravidlo (převzato z [11]).

Důležitým operátorem je operátor rozsahu: „..“ - používá se k upřesnění, které znaky můžeme použít. V příkladu 3.3 je znázorněné lexikální pravidlo pro identifikátor. Rozsah písmen je od *a* až do *z* (velká i malá písmena) a u čísel můžeme použít čísla *0* až *9* [11].

Syntaktická analýza

Po lexikální analýze se musíme ujistit, že sekvence tokenů tvoří validní příkaz v jazyku. Znamená to, že respektuje syntaktickou strukturu očekávanou jazykem. Tato fáze se nazývá **syntaktická analýza** nebo parsování. Parser se spoléhá na lexikální analyzátor (viz obr. 3.4) [7].



Obrázek 3.4: Proces parsování (převzato a modifikováno z [12]).

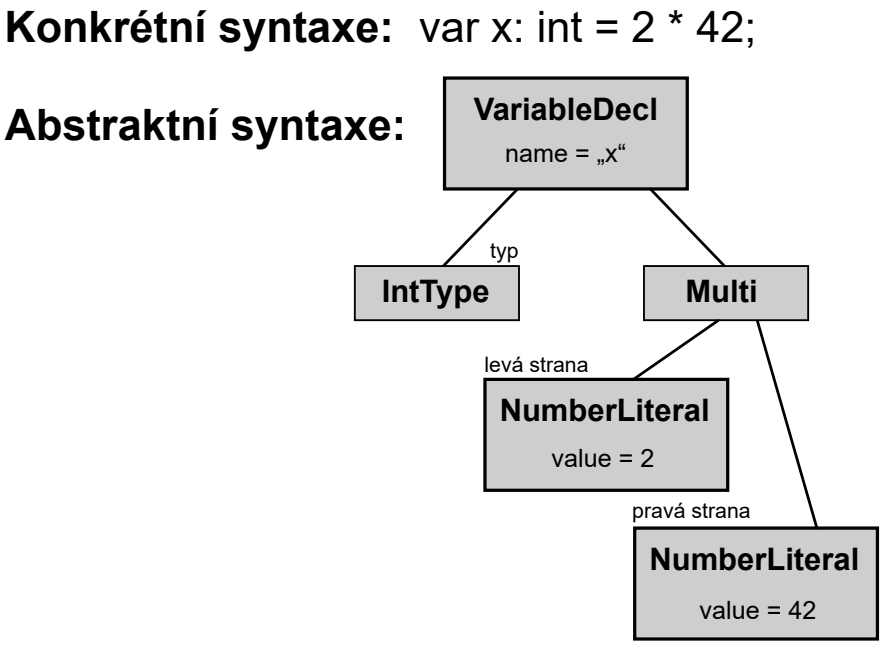
Existují nástroje, které se zabývají parsováním a není třeba, abychom psali parser sami. Pouze specifikujeme gramatiku jazyka a z této gramatiky je automaticky vygenerován kód pro lexikální analyzátor a parser. Takový nástroje nazýváme **generátory parserů** (*parser generators*) [7].

Nejznámějšími nástroji, které jsou spojené s jazykem C jsou **Bison** a **Flex** - Bison provádí syntaktickou analýzu a Flex lexikální analýzu. S Javou je spojen nástroj **ANTLR**, který umožňuje programátorovi napsat gramatiku do jednoho souboru a automaticky vygeneruje parser v Javě [7].

3.3.5 Abstraktní syntaktický strom

Jak již bylo v kapitole 3.3.4 řečeno, parsování není jediná činnost, kterou parser vykonává. Pokud je syntaxe programu validní, vytváříme v paměti strukturu naparsovaného programu. Jedná se o stromovou strukturu, která se nazývá **abstraktní syntaktický strom** (*Abstract syntax tree* - AST) [7]. Pokud je jednou strom načten v paměti, pak není třeba znova program parsovat [7].

AST představuje abstraktní syntaktickou strukturu programu [23]. Tento strom je vytvořen z konkrétní syntaxe uživatelem vytvořeného DSL scriptu: parser vytvoří a naplní daty instanci abstraktní syntaxe. Data jsou založena na informacích získaných v textu programu. V tomto případě se formálně konkrétní syntaxi říká gramatika (viz 3.3.3) [31].



Obrázek 3.5: Abstraktní a konrétní syntaxe. (převzato a modifikováno z [31]).

Na obrázku 3.5 lze vidět, že oproti konkrétní syntaxi, která obsahuje klíčová slova i znaky „:“ a „;“: (`var x: int = 2 * 42;`), tento strom taková slova a znaky ve svých uzlech neobsahuje [31].

3.3.6 Sémantika

Aby mohla být provedena **sémantická analýza**, potřebujeme mít již vytvořený AST (viz kapitola 3.3.5). Jestliže AST existuje, lze vytvořit sadu pravidel a omezení, která definují sémantiku jazyka [3]. Sémantika DSL přiřadí každému ze svých jazykových konstruktů přesný význam. Přesněji řečeno, *statická sémantika* omezuje sady platných programů, zatímco *dynamická sémantika* určuje, jak jsou vyhodnocovány za běhu [20].

Před spuštěním dynamické sémantiky programu je nutné ověřit jeho statickou sémantiku. Pro statickou sémantiku se používá omezení a typové systémy [31].

Omezení

Omezení jsou jednoduše booleovské výrazy, které kontrolují libovolnou vlastnost modelu. Lze například ověřit, že názvy různých entit jsou unikátní. Aby byl model staticky správný, musí být všechna omezení vyhodnocena jako *true*. Kontrola omezení by měla být prováděna pouze u modelu, který je syntakticky správný [31].

Strukturu jazyka definujeme v přepisovacích pravidlech. Někdy je lepší napsat volnější přepisovací pravidlo a vytvořit k tomu omezení. Například namísto použití multiplicity 1..N v přepisovacím pravidlu se často používá 0..N spolu s omezením, které kontroluje, zda existuje alespoň jeden prvek. Důvodem tohoto přístupu je, že pokud budeme mít původní pravidlo a bude chybět prvek, přijde z parseru možná chybová zpráva (např. `required (...)+ loop did not match anything at input '<EOF>'`). Pro uživatele DSL se jedná o zprávu, která není příliš užitečná. Pokud se použije tolerantnější (0..N) verze přepisovacího pravidla, lze získat chybovou zprávu až při kontrole omezení, kdy přesně víme, o co se jedná a co chyba vyznačuje - vzor zprávy může být následovný: `Elements not found, define at least one element` [31].

Typové systémy

V gramatice nemusí být typy psány explicitně programátorem. Typový systém závisí na sémantice. Mechanismus, kterým zjišťujeme typ, je obvykle nazýván **typová inference** [7].

Případy pro typovou inferenci jsou:

- **Fixní typy**:

Konstanty - celočíselná konstanta má typ *integer*, řetězcová konstanta je typu *string* a podobně [7].

- **Odvozené typy**:
  Výraz je typu, který je odvozen z jiných prvků jazyka (kontrolujeme typy dílčích výrazů) [7, 31]. Např.: odvozujeme typ u proměnné y z výrazu $x + 5$: `var x: int(5); var y = x + 5;`

- **Hierarchie typů**:
  Typové systémy můžou podporovat hierarchii typů (např. *integer* je podtypem *double*, a proto může být použit všude, kde se očekává *double*) [31].

Typová inference a kontrola se implementuje najednou. Obecně typ výrazu závisí na typech dílčích výrazů. Výraz není správně typovaný (*well-typed*), pokud není podtyp správně typovaný [7].

Příklad: výraz `j * true` není správně typovaný, protože násobení v našem jazyce lze provést pouze s celými čísly [7].

3.4 Fáze implementace

Poté, co je jazyk navrhnut a ověřen, přecházíme do implementační fáze [7]. Fáze jazykové implementace zahrnuje integraci DSL artefaktů, které vyplývají z fáze jazykového návrhu. V souvislosti s tím jsou implementovány také nezbytné transformace DSL kódu na objekty domény. V této fázi se zaměřujeme i na API k vlastnímu jazyku [3].

Artefakt AST (viz kapitola 3.3.5) lze použít k tvorbě vlastního **generátoru kódu** [12]. Obvykle chceme vygenerovat kód v jiném jazyce (například kód Java), konfigurační, XML nebo textový soubor. Ve všech těchto případech musíme napsat generátor [7].

Jakýkoli vygenerovaný kód je artefakt. Pokud jej ručně upravíme, pak ztratíme tyto změny při novém generování. Proto není doporučeno kód ručně měnit. Ideální je udržovat vygenerovaný a ručně psaný kód odděleně [11]. Generování lze implementovat různými přístupy (např. interpret, kompilátor), z nichž každý má své vlastní výhody [6].

Interpretace nebo kompilace: Jedná se o klasický přístup implementace nového jazyka. Hlavní výhodou sestavení kompilátoru nebo interpreta je, že implementace je zcela přizpůsobena DSL [9].

3.4.1 Interpretace

Tento přístup je vhodný pro jazyky s dynamickým charakterem, nebo pokud rychlost provádění (*excecution time*) není problém. Výhodami interpretace oproti kompilaci jsou: větší jednoduchost a jednodušší rozšíření [18].

Interpret prochází AST a přímo provádí akce v závislosti na obsahu AST. S interpretací se setkáme u omezení a typových systémů (viz kapitola 3.3.6): kontroly omezení a typů lze považovat za interpreta, protože procházíme strom a provádíme různé akce [31].

3.4.2 Kompilace

Kompilátory obvykle generují strojový kód [31]. U doménově specifického jazyka jsou konstrukce přeloženy do GPL. Součástí kompilátorů je i implementace optimalizací. Důvodem, proč je vygenerovaný kód většinou v obecném jazyku a není rovnou vytvořen strojový kód, je to, že chceme použít existující transformace a optimalizace poskytované GPL kompilátorem [31]. Kompilátorům DSL se často říká generátory aplikací [18].

3.5 Fáze testování

Fáze testování jazyka by měla zahrnovat *verifikaci* (testování, zda DSL poskytuje správnou funkčnost) a *validaci* (testování, zda je DSL vhodný pro jeho uživatele) [6].

3.5.1 Testování

Psaní automatizovaných testů je základní metodikou při vývoji softwaru. Pomáhá nám napsat kvalitní software, kde většina aspektů (možná všechny aspekty) je automaticky průběžně ověřována. Ačkoli úspěšné testy nezaručují, že software neobsahuje chyby, automatizované testy jsou nezbytnou podmínkou profesionálního programování [7].

Počet testů byl měl růst s rostoucí implementací. Testy by se měly provádět pokaždé, kdy vznikne nová funkce nebo je něco změněno. Výhodou je, že lze jakkoli vyvíjet jazyk a není třeba se obávat upravit implementaci. Po dokončení psaní kódu pouze spustíme celou testovací sadu a zkontrolujeme, zda všechno úspěšně proběhlo. Pokud některé testy selžou, stačí

zkontrolovat, zda se skutečně očekává selhání, nebo zda je třeba opravit námi napsaný kód [7].

3.5.2 Evaluace

Evaluace DSL se moc neliší od hodnocení běžných uživatelských rozhraní (*UI*). Většina požadavků souvisejících s evaluací UI je ve skutečnosti spojena s **použitelností**. Ve své práci *Ankica Barisic* [6] zmiňuje následující atributy použitelnosti a flexibility [3]:

- Efektivita - zaměřujeme se na to, kolik měl uživatel během psání kódu chyb.
- Účinnost - sledujeme, kolik uživatel strávil času u jednotlivých úkolů.
- Spokojenost - sledujeme, jak si uživatelé věřili v jednotlivých úkolech a jak jsou spokojení s jazykem.
- Přístupnost - zaměřujeme se na naučitelnost jazyka [3, 6].

3.6 Fáze nasazení

Po otestování a evaluaci následuje fáze nasazení. Vývojáři nebo odborníci na doménu v této fázi používají již funkční DSL ke specifikaci modelů. Výsledkem takové implementace je funkční software, který používají koncoví uživatelé.[3]

Dále do této fáze zahrnujeme údržbu jazyka - spravujeme a aktualizujeme doménově specifický jazyk tak, aby reflektoval nové požadavky [30].

3.6.1 Dokumentace

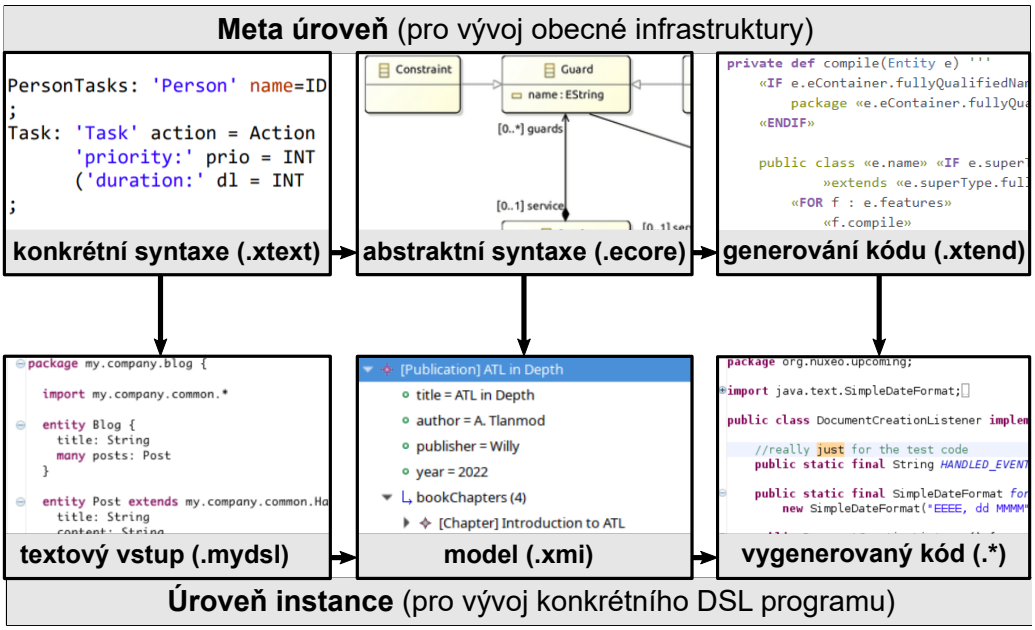
K tomuto životnímu cyklu DSL patří také dokumentace. V dokumentaci musíme popsat:

- strukturu a syntaxi,
- jak používat editory a generátory,
- jak a kde psát vlastní ruční kód (jak jej integrovat do vygenerovaného kódu),
- rozhodnutí o platformě [31].

Je důležité si uvědomit, že přečíst si uživatelskou dokumentaci může být pro uživatele někdy obtížné a může se stát, že si ji ani nepřečte. Pokud je to možné, je vhodné zvolit k dokumentaci i jiná média, jako je např. video nebo podcast. Dokumentace by neměla být pouze popis jazyka, ale z větší části by měla sloužit jako návod. Uživatel bude podle tohoto návodu vytvářet úvodní program v DSL [31].

4 Xtext framework

XText je Eclipse framework pro implementaci obecných i doménově specifických jazyků [7]. Na pozadí XText používá generátor syntaktického analyzátoru `LL(*)` ANTLR, který umožňuje pokrýt širokou škálu syntaxí [32]. Vývojář napíše definici gramatiky v ANTLR notaci a z této gramatiky XText vygeneruje lexer, parser, abstraktní strom (AST), AST konstrukci k parsovanému programu a plnohodnotný Eclipse editor, který dokáže například zvýraznit syntaxi [7, 8].



Obrázek 4.1: Přehled Xtext frameworku (převzato a modifikováno z [21]).

Přehled hlavních částí frameworku je znázorněn na obr. 4.1. Infrastruktura DSL je konstruována v meta-level pracovním prostoru. V `.xtext` souboru je definována gramatika (více o gramatice v kapitole 3.3.3 a 6.1), ze které následně vzniká automaticky `.ecore` model (nazýváme *meta-model*). Tento model reprezentuje abstraktní syntaxi jazyka. Na základě modelu lze sestavit generátor, který dokáže pracovat s třídami meta-modelu a dokáže tak vygenerovat například kód v jiném jazyku [21].

Uživatelé používají běhové prostředí, ve kterém píšou své DSL programy podle syntaxe jazyka. Při psaní kódu se vytváří automaticky instance ecore modelu (nazýváme *model*) - *.xmi* soubor, ve kterém jsou obsažené všechny

informace podle konkrétního DSL kódu. Na základě konkrétních dat a nadefinovaného algoritmu generátoru lze DSL kód transformovat např. na kód jiného jazyka [21].

4.1 Eclipse Modeling Framework

Při generovaní abstraktního syntaktického stromu se Xtext spoléhá na **Eclipse Modeling Framework** (EMF) [8]. Jedná se o sadu Eclipse pluginů, které lze použít k tvorbě datového modelu. Z tohoto modelu lze následně vygenerovat kód či jiný výstup. EMF je běžným standardem pro datové modely, na nichž je založeno mnoho technologií a frameworků.[1]

EMF rozlišuje mezi meta-modelem a modelem. Meta-model popisuje pouze strukturu, kdežto model je konkrétní instance meta-modelu.[2] Xtext používá EMF modely k reprezentaci všech analyzovaných textových souborů v paměti (např. AST) [10].

4.2 Proč Xtext?

- Umožňuje tvorbu plnohodnotných textových editorů pro obecné nebo doménově specifické jazyky.
- Je umožněno definovat cílový formát, do kterého je jazyk kompilován. Bez ohledu na to, zda se jedná o prostředí Java, C, XML, hodnoty oddělené čárkami nebo dokonce o binární formát.
- Výchozí chování Xtextu je optimalizováno tak, aby zahrnovalo širokou škálu jazyků a případů použití. Pro každý jazyk je možné standardní chování IDE nahradit vlastní implementací.
- Mnoho funkcí IDE poskytovaných Xtextem se automaticky přizpůsobuje vyvíjenému jazyku - stačí změnit definici gramatiky a chování textového editoru se aktualizuje bez dalších změn kódu [32].

[1] Eclipse Modeling Project | The Eclipse Foundation. Enabling Open Innovation & Collaboration | The Eclipse Foundation [online]. Copyright © Eclipse Foundation. All Rights Reserved. [cit. 19.05.2021]. Dostupné z: https://www.eclipse.org/modeling/emf/

[2] Eclipse Modeling Framework (EMF) - Tutorial. Eclipse, Android and Java training and support [online]. Dostupné z: `https://www.vogella.com/tutorials/EclipseEMF/article.html`

Podobným frameworkem je **JetBrains MPS**, který nabízí také vývoj vlastního doménově specifického jazyka.[3] Při vývoji jazyka není zahrnuta gramatika a ani syntaktický analyzátor. Místo toho se edituje přímo AST, který je v editoru promítnut jako text. Na základě tohoto přístupu, který se liší od Xtextu, lze použít textové, symbolické, tabulkové a grafické notace. MPS framework je volně dostupný a je vyvíjen společností JetBrains. Ačkoli umožňuje pokročilé funkce jako Xtext, není až tak známý [31].

4.3 Instalace Xtext frameworku

Vzhledem k tomu, že se jedná o *Eclipse* framework, lze Xtext nainstalovat do vývojového prostředí vložením webové adresy, na které je dostupná aktuální verze frameworku [7]:

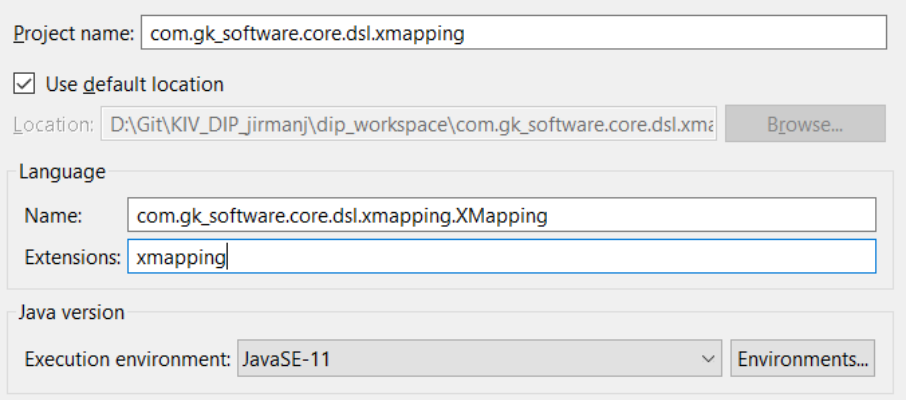
URL adresa: `http://download.eclipse.org/modeling/tmf/xtext/updates/composite/releases`

Tuto adresu stačí vložit do dialogového okna, které se objeví, pokud klikneme na **Help** → **Install new software...**. Jestliže tuto adresu potvrdíme, spustí se nám celá instalace Xtext frameworku [7] včetně knihovny Antlr generátoru [21].

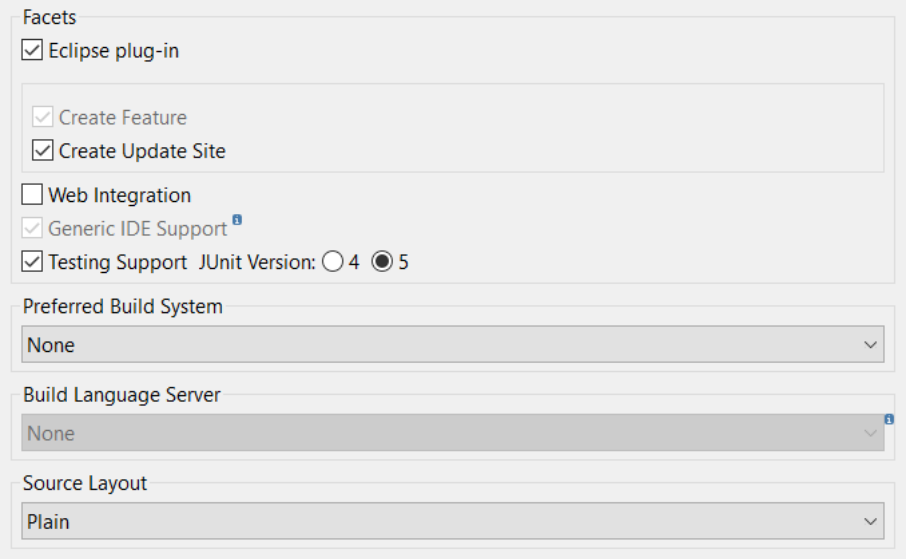
4.3.1 Vytvoření Xtext projektu

Vytvoření Xtext projektu není náročné. Standardně klikneme v nabídce na **File** → **New** → **Project**. Vybereme, že chceme vytvořit **Xtext Project** [21]. V dialogovým okně musíme nastavit *název projektu* a *jméno jazyka*. Dále *vlastní příponu*, kterou budeme používat k identifikování vlastního textového souboru. Na následujícím obrázku 4.2 jsou vidět již aktuální názvy nového mapovacího DSL.
Dále musíme kliknout na **Next**, protože chceme pokročilou konfiguraci, kde nastavíme, že se bude jednat o **Xtext plug-in**. Také zaškrtneme možnosti: **Create feature** a **Create Update Site** (viz obr. 4.3), abychom mohli plugin vygenerovat.

[3] JetBrains MPS | AlternativeTo - Crowdsourced software recommendations [online]. Dostupné z: `https://alternativeto.net/software/jetbrains-mps/about/`



Obrázek 4.2: Vytvoření nového Xtext projektu - základní nastavení.



Obrázek 4.3: Vytvoření nového Xtext projektu - pokročilé nastavení.

4.3.2 Jazyk Xtend

Po vytvoření projektu lze vidět několik modulů. V těchto modulech můžeme vidět `.java` soubory, `.xtext` soubor (vysvětleno v kapitole 6.1) a `.xtend` soubory, který využívají jazyk `Xtend`.

Xtend je staticky typovaný jazyk, který používá typový systém Javy. Většina jazykových konceptů jazyka Xtend je velmi podobná Javě - např.: třídy, rozhraní, metody. Jeden z cílů je mít „osekanou“, ale lepší verzi Javy, protože v Javě jsou některé jazykové vlastnosti nadbytečné a pouze činí programy

složitějšími [7]. Dokumentace je dostupná z [4]. Na následujícím výpisu kódu 4.1 je krátká ukázka jazyka Xtend.

```
class MyFirstXtendClass {
  val s = 'my field' // konstanta
  var myList = new LinkedList<Integer> // promenna

  def bar(String input) {
    var buffer = input
    buffer == s || myList.size > 0
    // metoda vraci posledni vyraz
  }
}
```

Výpis kódu 4.1: Ukázka jazyka Xtend (převzato a modifikováno z [7]).

[4]Xtend - Documentation. Enabling Open Innovation & Collaboration | The Eclipse Foundation [online]. Dostupné z: `https://www.eclipse.org/xtend/documentation/index.html`

5 Rozhodnutí a analýza

Dříve než se zaměříme na návrh a implementaci, je důležité objasnit, co vedlo k tvorbě nového doménově specifického jazyka. Popíšeme, jak probíhala analýza problémové domény.

5.1 Aktuální stav

Ve firmě *Eurosoftware*, která je zadavatelem diplomové práce, se k mapování XML souborů využívá licencovaný program **Altova MapForce**. Tento software používá více oddělení, takže program používají různí uživatelé - od konzultantů až po programátory. Každý uživatel má jiné znalosti a z tohoto důvodu vznikla první myšlenka, aby mapování XML souborů nebylo příliš obtížné pro kohokoli a aby nový uživatel po krátkém zaučení mohl mapovat soubory.

5.1.1 Altova MapForce

Altova MapForce je nástroj umožňující grafické mapování dat, konverzi a integraci. Mapuje jakékoliv kombinace např. XML, SQL databází, Excel nebo webových služeb. Poté může vygenerovat kód.[1] Výsledkem generování kódu je plnohodnotná aplikace (např. v jazyce Java, C++), která provádí operace mapování. Vygenerovaná aplikace je spustitelná a dá se využít k opakovanému použití.[2]

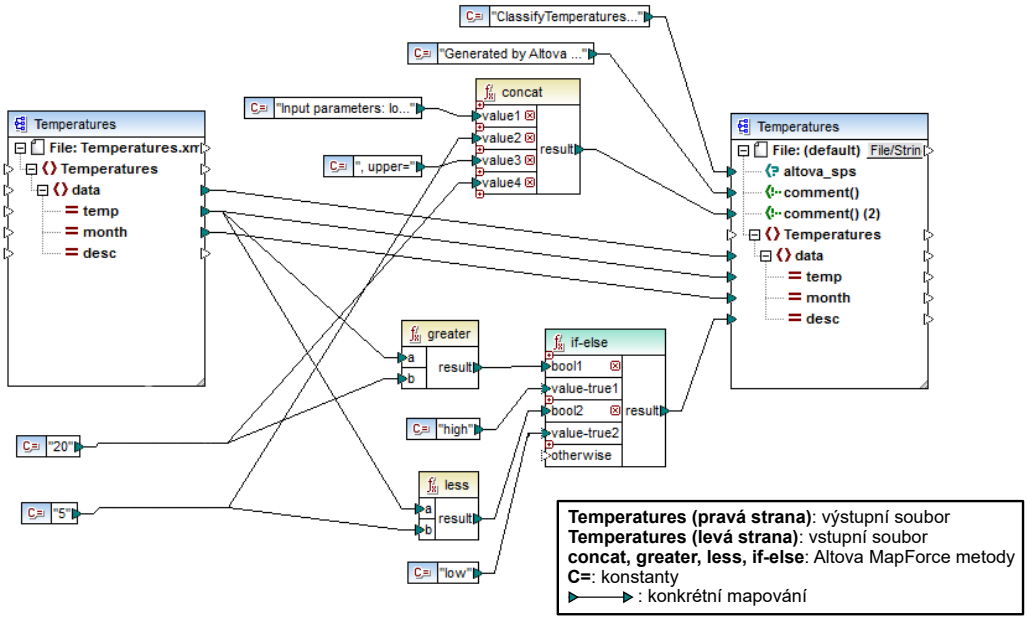
Výhody softwaru

1. Pokud se podíváme do dokumentace [2], pak objevíme spoustu zajímavých funkcí, které Altova MapForce nabízí. Například je možné od Altovy MapForce 2014 používat *wildcards*, které se týkají elementů: `<xs:any>` a `<xs:anyAttribute>`.

[1] Altova MapForce 2020 Enterprise Edition vč. 1 roku SMP | SW.cz . SW.cz - Specialista na Software [online]. Copyright © 2021 [cit. 24.05.2021]. Dostupné z: `https://www.sw.cz/podnikani-a-domacnost/databaze/mapforce-2020-enterprise-edition-vc-1-roku-smp/`

[2] Advanced Data Mapping Features | Altova. XML, Data Integration, and Mobile App Development Solutions by Altova | Altova [online]. Dostupné z: `https://www.altova.com/mapforce/advanced#DataMappingDocumentation`

2. Když se zaměříme na uživatelské rozhraní, tak i přesto, že v GUI se nemusíme ihned orientovat, tak jeho vizualizace mapování pomáhá k tomu, aby uživatel věděl, co je již namapované a co ne (viz obr. 5.1).



Obrázek 5.1: Grafická vizualizace mapování souborů (převzato a modifikováno z [3]).

Na obrázku 5.1 můžeme vidět, kromě jednoho vstupního a výstupního souboru (`Temperatures` vlevo a vpravo), použité funkce (`upper`, `greater` a `concat`) a konstanty jako např. `"20"` a `"low"`.

Nevýhody softwaru

1. Software má předdefinované funkce, které lze využít při transformaci dat. Omezující na tom je, že není možné přidat nové, které si uživatel napíše sám. Svůj vlastní kód lze přidat až do vygenerované spustitelné aplikace, což může být pro uživatele obtížné, protože by musel analyzovat, jaký algoritmus Altova MapForce používá.

2. Grafické uživatelské rozhraní je velice obsáhlé a nedá se v něm vždy dobře orientovat. Uživatel, který má se softwarem malé zkušenosti se musí často učit nebo si připomínat různé funkcionality z internetových návodů.

[3] Example: Returning a Value Conditionally. XML, Data Integration, and Mobile App Development Solutions by Altova | Altova [online]. Copyright © 2015 [cit. 27.05.2021]. Dostupné z: `https://www.altova.com/manual/Mapforce/mapforcebasic/mff_condition_example.html`

Další nedostatky

Tyto nedostatky se nedají zařadit k nedostatkům softwaru. Týkají se toho, jak je software ve firmě používán a jak je integrovatelný ve firemních procesech.

- Nevýhodou, která již byla zmíněna v úvodu kapitoly 5.1, je, že Altovu MapForce používá v *Eurosoftware* více oddělení, která mají odlišné zkušenosti s tímto softwarem.

  – Pokud by Altovu MapForce používali pouze konzultanti a zákazníci, pak by tento grafický software byl vyhovující.

  – Pokud by jej používali pouze programátoři, pak by bylo vyhovující mít mapování naprogramované v obecném jazyce - ideálně ve firmě často používané Javě.

  Momentálně je mapování XML souborů vytvářeno často vývojáři, ale připojují se i konzultanti a výjimečně zákazníci. Tudíž je grafický software nevyhovující, protože aktuálně pomáhá pouze části uživatelů. V opačném případě by existovalo mapování v obecném jazyku Java, které by vyhovovalo vývojářům, ale práci by to konzultantům neulehčilo.

- Dalším nedostatkem je verzování vytvořených transformací a přenášení změn mezi jednotlivými projekty. Pokud v produktovém oddělení uživatelé vytvoří a nahrají do centrálního repozitáře nový Altova MapForce soubor (zkratka *.mfd*), pak projektové oddělení si jej může bez problému převzít a upravit podle potřeb zákazníka. Komplikace nastává, když produktové oddělení udělá v nových verzích produktu změny, které se špatně mergují do konkrétního mapování zákazníka - i když jsou téměř shodné s původním mapováním starší verze produktu. V repozitáři nelze vidět provedené změny, protože se jedná o binární soubor *.mfd*, který není určen k verzování.

5.2 Cíl práce

Po důkladném zvážení všech kladů a záporů grafického softwaru vznikl ve firmě *Eurosoftware* požadavek vytvořit nástroj, který by usnadnil vytváření mapování mezi XML soubory.

Cílem této práce je vytvořit deklarativní doménově specifický jazyk, který sjednotí znalosti uživatelů ve všech firemních oddělení, kde se používá Altova Mapforce, tím, že nebude obsahovat nadbytečné funkce ze zmíněného

softwaru. Zároveň ponechá základní funkcionalitu na stejné úrovni jako Altova MapForce. Výhodou tohoto jazyka budou zdrojové textové soubory, které bude možné oproti binárním souborům mergovat.

5.2.1 Důvody

1. Vzhledem k aktuální situaci, že mapování tvoří často vývojáři, se hodí použít programovací jazyk. Java je příliš obecná, pokud chce konzultant na mapování pracovat. V této situaci je ideální vytvořit kompromis - doménově specifický jazyk. Pro vývojáře bude lepší psát DSL kód a zároveň tento jazyk umožní, aby jej psali i konzultanti, protože bude obsahovat přesně dané jazykové konstrukce a úplně odstíní mapovací logiku.
2. S DSL vznikne i vlastní textový soubor, který bude možné mergovat a vidět tak změny v repozitářích.
3. DSL lze připravit na různé uživatelské metody, které by v Altova MapForce šly obtížně integrovat.
4. Firma má zkušenost s doménově specifickými jazyky.

Tyto důvody jsou natolik důležité, že vedou k myšlence vzniku doménově specifického jazyka. Tento jazyk má vysoký potenciál v tom, že urychlí vývoj mapování a zároveň ponechá funkcionalitu na stejné úrovni jako Altova MapForce. Přenos mezi odděleními bude bezproblémový díky vlastnímu textovému souboru s DSL kódem.

5.3 Požadavky jazyka

S myšlenkou DSL přichází firma s požadavky, které se týkají samotného jazyka.

Potřebujeme doménově specifický jazyk, který bude:

- Pro všechna oddělení jednotný.
- Jednoduchý na použití, takže by dokázal nahradit někdy nepřehledné GUI Altova MapForce.
- Měl vlastní knihovnu, do které by šly přidávat uživatelské metody.
- Šel kdykoliv rozšířit.
- Klíčovými slovy přesně vyjadřoval, co konkrétního jazyk udělá.
- Vytvořil mapování stejné jako z licencovaného softwaru.

5.4 Doménová analýza

Po obhájení důvodů, proč mít ve firmě nový doménově specifický jazyk, přišlo závěrečné rozhodnutí, že tento jazyk vznikne. Po tomto rozhodnutí následovala doménová analýza. Při této analýze všichni stakeholdeři sjednotili informace spojené s danou doménou.

Pro tuto analýzu byl jeden z důležitých zdrojů informací samotný grafický software, který obsahuje mnoho klíčových slov, které by mohly být užitečné k definování doménově specifického jazyka. Také se analyzoval již vytvořený GPL kód pro konkrétní mapování XML souborů v projektu. Bylo důležité sesbírat poznatky a znalosti všech uživatelů daného softwaru, ze kterých je možné vytvořit základní doménové koncepty.

5.4.1 GPL kód

Jestliže v Altova MapForce existuje mapování, pak je umožněno vygenerovat kód v Javě, který dokáže provést dané mapování konkrétních souborů. Analýza tohoto kódu byla zejména důležitá pro vytvoření vlastní knihovny k mapování, které by DSL využívalo.

Analyzoval jsem konkrétní třídy a algoritmy, jak mapování probíhá. Ačkoli algoritmy byly výrazně složité a nepřehledné, snažil jsem se různých tříd a metod vybrat důležité názvy a konstrukce, které je vhodné použít a které uživatel Altova MapForce pozná při pohledu do Java vygenerovaného kódu doménově specifického jazyka. Názvy tříd algoritmů nemají pro uživatele žádný význam, protože je uživatel od logiky mapování odstíněn. Z tohoto důvodu jsem vybral pouze názvy pro vstupní a výstupní soubor, protože každý DSL kód vygeneruje Java třídy, které budou obsahovat instance vstupních a výstupních souborů. V tabulce 5.1 jsou tyto názvy vypsány.

GPL kód a DSL knihovna	**Popis**
třída `DocumentOutput`	třída reprezentující výstupní soubor
třída `FileInput`	třída reprezentující vstupní soubor
metoda `saveDocument()`	uložení výstupního souboru do file systému

Tabulka 5.1: Nalezení klíčových názvů v GLP kódu pro DSL knihovnu.

Dále bylo třeba zjistit, zda je vhodné, aby uživatel vytvářel své funkce s třídami z Java knihovny, nebo použitím tříd z DSL knihovny. Analýzu ovlivnila skutečnost, že Altova nabízí možnost vložit vlastní kód do vygenerovaného

Java kódu, ale je potřeba znát některé její třídy, které na první pohled nejsou vůbec jasné a podle názorů uživatelů, které s tím měli alespoň nějakou zkušenost, trvá dlouho, než pochopili způsob integrace vlastního kódu.

Dostupné třídy	Popis
`org.w3c.dom.Node`	Uživatel pracuje s uzelem dokumentu.
`java.lang.String`	Uživatel pracuje s textovým obsahem.

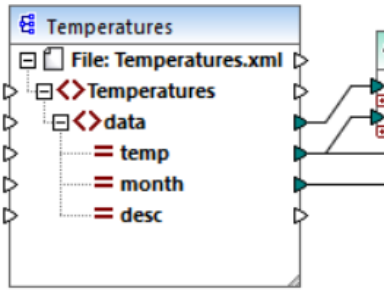
Tabulka 5.2: Seznam tříd pro vlastní uživatelské metody.

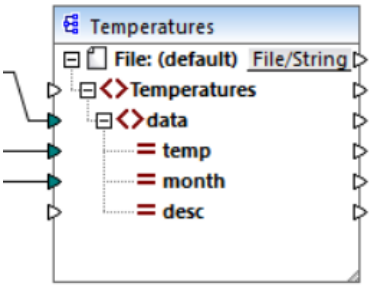
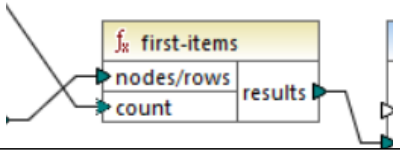
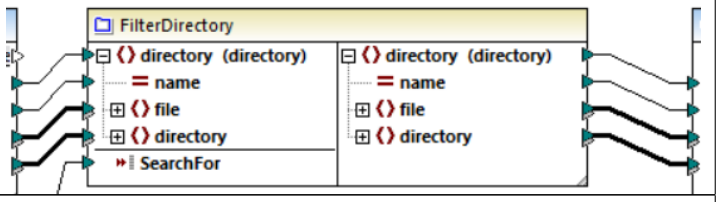
V tabulce 5.2 jsou dvě klíčové třídy, na kterých se stakeholdeři shodli, že by bylo vhodné je použít pro definování vlastních uživatelských funkcí, protože tak budou odstíněni od detailů tříd v DSL knihovně.

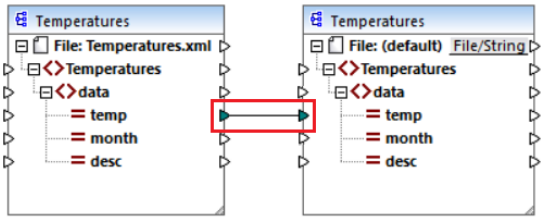
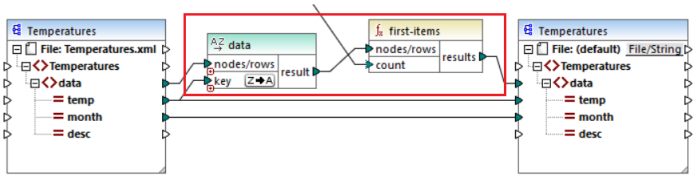
5.4.2 Software Altova MapForce

Analýza GPL kódu se více zabývala klíčovými názvy a konstrukcemi pro DSL knihovnu, kdežto analýza grafického softwaru byla hlavně důležitá pro definování společné terminologie, doménových konceptů a feature modelu. Bylo nutné najít množinu názvů a vlastností z Altova MapForce, která by byla použitelná pro DSL.

Po důkladném analýze tohoto softwaru vznikly doménové koncepty - tab. 5.3:

Doménový koncept	Detaily
Vstupní soubor (`.xsd`)	• Před použitím definovat cestu k `xsd` souboru. • Použít `xPath` pro výběr elementů při mapování. • Mít unikátní název, ale zachovat informaci, že se jedná o vstupní soubor. • Reprezentace *vstupního souboru*, který můžeme vidět v Altova MapForce: 
Výstupní soubor (`.xsd`)	• Před použitím definovat cestu k `xsd` souboru. • Použít `xPath` pro výběr elementů při mapování.

	• Mít unikátní název, ale zachovat informaci, že se jedná o výstupní soubor. • Reprezentace *výstupního souboru*, který můžeme vidět v Altova MapForce: 
Uživatelská metoda	• Typy formálních parametrů uživatelské metody: `org.w3c.dom.Node` - uzel dokumentu, `java.lang.String` - textový obsah. • Reprezentace *uživatelské metody*, kterou můžeme vidět v Altova MapForce: 
Uživatelská rutina	• Umožnit vytvářet v jazyku vlastní rutiny. • Vstupní parametry nebudou podmínkou. • Výstup nebude podmínkou. • Reprezentace *uživatelské rutiny*, kterou můžeme vidět v Altova MapForce: 
Proměnná	• Možné definovat proměnnou v jazyce. • Může být typu: `String`, `int`, `double`. • Mít unikátní název, ale zachovat informaci, že se jedná o proměnnou. • Reprezentace *proměnné*, kterou můžeme vidět v Altova MapForce: 
Přímé mapování	• Bude přímo mapovat vstupní element do výstupního.

	• Reprezentace *přímého mapování*, které lze vidět v Altova MapForce: 
Obecné mapování	• Mapování bude prováděno přes uživatelské metody nebo rutiny. • Reprezentace *obecného mapování*, které lze vidět v Altova MapForce: 

Tabulka 5.3: Konkrétní doménové koncepty

Z těchto doménových konceptů lze odvodit **společný slovník**, který pomůže v komunikaci mezi zainteresovanými stranami. V tabulce 5.4 lze vidět transformace mezi problémovou doménou a *solution doménou*. Klíčová slova jsou napsána v anglickém jazyce, aby mohl slovům porozumět kdokoli z firmy.

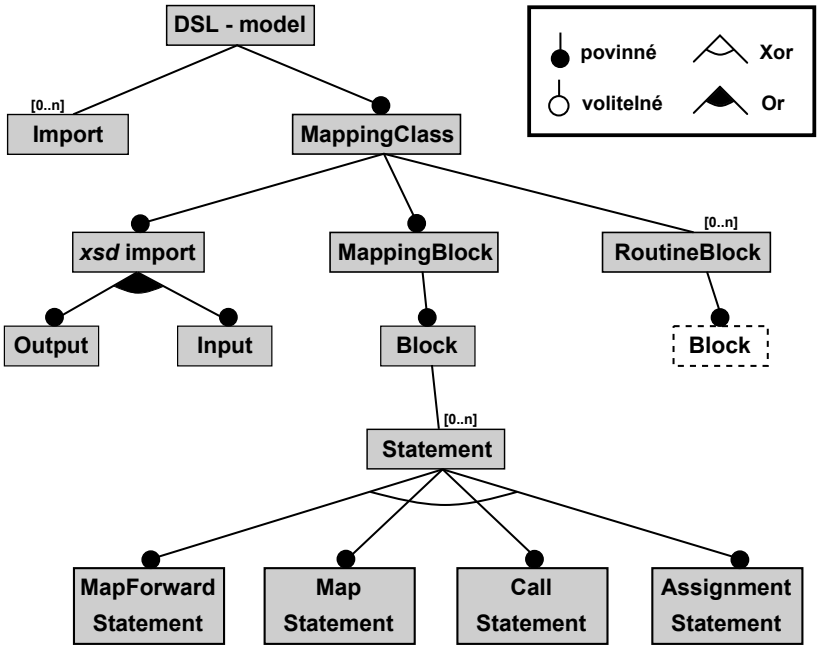
Problémová doména	**Solution doména**	**Popis**
`Input`	`Input`	Vstupní *xsd* soubor.
`Output`	`Output`	Výstupní *xsd* soubor.
`Variable`	`Variable`	Proměnná v DSL.
`Direct mapping`	`Map Forward`	Mapování vstupních elementů na výstupní bez uživatelských maker a metod.
`Mapping`	`Map`	Mapování vstupních elementů na výstupní s uživatelskými makry a metodami.
`Function/Method`	`Call`	Uživatelská metoda v Javě, která je v DSL kódu volána pro mapování.

`Routine`	`Routine`	Uživatelská rutina, která je vytvořená v DSL a obsahuje příkazy DSL jazyka.

Tabulka 5.4: DSL slovník

Na základě doménových konceptů a slovníku byl vytvořen **feature diagram**, který je také výstupem analýzy (viz obr. 5.2). Ve feature diagramu je znázorněn doménově specifický jazyk. V jednotlivých uzlech si lze všimnout, že jsou již použity názvy ze slovníku - např:

- Názvy `Input`, `Output` - z toho lze bez dalších znalostí odvodit, že se jedná o importování vstupních a výstupních souborů. Dále je vidět, že tento import souborů se provádí dříve než samotné psaní mapovacích příkazů, což bylo popsáno v doménovém konceptu pro vstupní i výstupní soubor.
- Uzel `Statement` reprezentuje jednotlivý příkaz v doménově specifickém jazyce. A zde si lze také všimnout, že jsou použity názvy ze slovníku pro solution doménu (např. `MapForward`, `Map` a `Call`) a je možné z toho opět poznat, že se bude jednat o přímé mapování, obecné mapování a volání uživatelských metod.



Obrázek 5.2: Feature model DSL.

5.5 Nástroje pro tvorbu jazyka

V této fázi jsem se zabýval otázkou, jaký software a frameworky budou vhodné pro tvorbu jazyka. Na výběr jsem měl ze dvou frameworků (Xtext a MPS), které jsou na internetu dostupné a které umožňují vytvořit vlastní jazyk. Jelikož se ve firmě používá několik let doménově specifický jazyk, který byl vytvořen ve frameworku `Xtext`, tak jsem se shodli ve firmě na tom, že by se měl jazyk vytvořit právě v tomto frameworku (více o Xtextu v kapitole 4). Navíc je tento nástroj podporován `Eclipse IDE`, které je ve firmě hodně využíváno a tudíž zde je jistota, že nebude problém nové DSL integrovat do zmíněného vývojového prostředí.

6 Návrh jazyka

V této fázi se dostáváme k vývoji jazyka. Vycházíme z informací z analýzy, protože máme definovaný slovník a také vybraný konkrétní framework pro tvorbu jazyka. Ze všeho nejdřív byl aktualizován *Eclipse IDE* a zprovozněn v tomto vývojovém prostředí `Xtext framework`. V kapitole 4.3 je vysvětlena instalace frameworku a založení projektu se všemi důležitými detaily.

Název DSL byl prodiskutován a tato informace byla vyplněna již při založení projektu:

- **Název DSL**: `XMapping`
- **přípona DSL souboru**: `.xmapping`

Znak `X` před `Mapping` byl vybrán z důvodu, že se pro nalezení elementů z dokumentu používá `xPath`. Navíc první znak musí být velké písmeno, protože musí dodržovat konvence pro pojmenování Java třídy.

Po vytvoření projektu se vytvoří několik modulů a z toho je pro návrh jazyka důležitý: `com.gk_software.core.dsl.xmapping`. V tomto modulu se nachází například `.xtext` soubor, ve kterém lze definovat gramatika jazyka. Dále v projektu existují vzorové soubory, které mají příponu `.xtend`.

6.1 Gramatika

Po založení projektu byl vygenerován Xtextem soubor `XMapping.xtext`, ve kterém se nacházela výchozí gramatika s dvěma přepisovacími pravidly (viz 6.1).

```
1  grammar com.gk_software.core.dsl.xmapping.XMapping
2  with org.eclipse.xtext.common.Terminals
3  generate xMapping "http://www.xtext.org/example/mydsl/
4                     XMapping"
5
6  Model:
7  greetings+=Greeting*;
8
9  Greeting:
10 'Hello' name = ID '!';
```

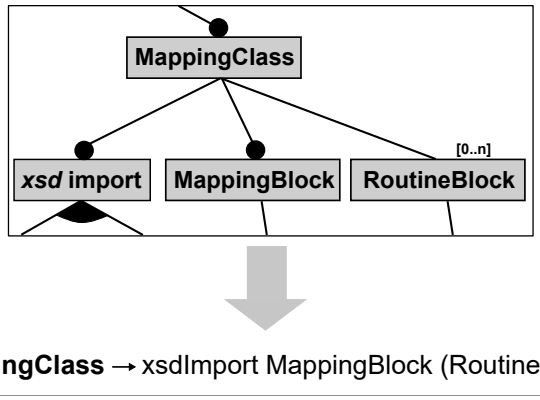
Výpis kódu 6.1: Ukázka nově vygenerované gramatiky.

Na začátku souboru se nachází název jazyka a gramatiky: `com.gk_software.core.dsl.xmapping.XMapping`. Tento název je shodný s plně kvalifikovaným názvem daného `.xtext` souboru. Dále je popsáno, že navíc využívá gramatiku `Terminals`, která definuje obecné věci jako např. identifikátory, komentáře a čísla. Nejdůležitější částí gramatiky jsou přepisovací pravidla.

Xtext se při generování abstraktního syntaktického stromu spoléhá na EMF (o EMF v kapitole 4.1). Z vytvořené gramatiky odvodí EMF meta-model, který reprezentuje abstraktní syntaxi. Pro každé pravidlo je vytvořeno v Javě rozhraní a také třída, která toto rozhraní implementuje [7]. Každý EMF element v meta-modelu dědí od třídy `EObject`.

6.1.1 Vývoj gramatiky

Základ gramatiky byl vytvořen podle získaných doménových konceptů a feature modelu. Feature model může být vyobrazen jako gramatika a díky tomu vznikla první přepisovací pravidla, ve kterých jsem dodržel požadavky z doménových konceptů - např. že import `.xsd` souborů bude v jazyce dříve definován než psaní mapovacích příkazů. Na obr. 6.1 jde vidět, jak lze feature model převést na bezkontextovou gramatiku.



Obrázek 6.1: Uzel feature modelu jako přepisovací pravidlo bezkontextové gramatiky.

.

Přepisovací pravidlo připomíná abstraktní syntaxi. Aby byl jazyk pro uživatele čitelný, bylo potřeba přidat klíčová slova. Ve výpisu kódu 6.2 je upravené přepisovací pravidlo z obr. 6.1.

```
MappingClass:
  'MappingClass' name=ID
   xsdImport=XsdImport
   block=MappingBlock
   (routineBlock+=RoutineBlock)*
;
```

Výpis kódu 6.2: Přepisovací pravidlo pro `MappingClass`

Gramatika se s přibývajícími iteracemi měnila a přizpůsobovala novým požadavkům zadavatele. Základ, který byl zmíněn na začátku kapitoly 6.1.1, se neměnil. Společně se zadavatelem jsem řešil, jak budou vypadat mapovací příkazy, aby uživatel pochopil jejich význam a zároveň neměl problém příkazy vytvořit. Na následujícím výpisu kódu 6.3 je ukázka jazyka s první navrženou gramatikou.

```
map Input.Element: "//xPath" -> Function.name => {
  map Function.name.Output.name.Element: "element_name" -> Function.name
    => Function.name => Function.name => {
        Function.name.Output.Element: //"xPath" ->
        Output.Element: "//xPath"
    }
}
```

Výpis kódu 6.3: Mapovací příkazy v jazyku - s rekurzí

Význam použitých jazykových konstrukcí:

- `map Input.Element:"//xPath" -> Function.name`
  - Volání uživatelské rutiny `Function.name` s jedním parametrem.
- `=> { ... }`
  - Blok pro konkrétní definování výstupů z uživ. rutiny.
- `Function.name.Output.name.Element:"element_name"`
  - Konkrétní výstup z uživ. rutiny
- `=> Function.name => Function.name`
  - Řetězení volání metody `Function.name` (pokud uživatelská rutina vrací pouze jeden prvek).
- `Function.name.Output.Element:"//xPath" -> Output.Element:"//xPath"`
  - Mapování výstupu uživ. rutiny do `Outputu` (výstupního souboru).

Po úvaze, že takový jazyk by byl pro uživatele nepřehledný, vznikl nový požadavek, že v jazyku je zakázáno používat rekurzi. Na dalším výpisu kódu 6.4 již není rekurze a jsou odebrány složené závorky - místo nich je přidán

příkaz `set-then`. Dále byl odebrán řetězec „=>“, aby byl používán k mapování pouze „->“ a nevznikly tak zbytečné chyby při psaní DSL programů.

```
Input.name.Element: "//xPath" ->
                          Output.name.Element: "//xPath";

map Input.name.Element: "//XPath" -> Macro.complex:
  set Macro.complex.Return.name -> Variable.funcReturn
  then Variable.funcReturn -> Output.name.Element: "//xPath";
```

Výpis kódu 6.4: Mapovací příkazy v jazyku - s příkazem `set-then`

Význam použitých jazykových kostrukcí:

- `Input.name.Element: "//xPath"-> Output.name.Element: "//xPath";`
  - Mapování z `Inputu` (vstupního souboru) do `Outputu` (výstupního souboru).
- `map Input.name.Element: "//XPath"-> Macro.complex`
  - Volání uživatelské rutiny `Macro.complex` s jedním parametrem.
- `: set ... then ...`
  - Blok pro konkrétní definování výstupů z uživ. rutiny.
- `set ...`
  - Příkaz pro nastavení všech výstupů z uživ. rutiny (např. uložení do proměnné).
- `then ...`
  - Lze provést mapování výstupů z uživ. rutiny.

Při zkoušení složitějších mapování jsem u příkazů `set-then` zjistil, že kód začne být nepřehledný a že by pro uživatele nebyl jazyk intuitivní. Bylo potřeba vymyslet jinou a jednodušší jazykovou konstrukci. Na následujícím výpisu kódu 6.5 je předposlední verze jazyka, která byla navržena.

```
map Input.Element:"//Xpath" -> Macro.name -> Variable.bool;
map Variable.bool -> Filter.name -> Output.Element:"//Xpath";
```

Výpis kódu 6.5: Mapovací příkazy v jazyku - uložení do proměnné

Význam použitých jazykových kostrukcí:

- `map Input.Element:"//Xpath"-> Macro.name -> Variable.bool;`
  - Volání uživatelské rutiny `Macro.name` s jedním parametrem.
  - `Macro.name` má jeden výstup, který ukládá do `Variable.bool`.

- `map Variable.bool -> Filter.name -> Output.Element:"//Xpath";`
  - Volání uživatelské rutiny `Filter.name` s jedním parametrem.
  - Parametrem je inicializovaná proměnná `Variable.bool`.
  - Výstup uživ. rutiny je namapován do `Outputu` (výstupního souboru).

Při posledních úpravách již zůstaly struktury příkazů stejné, pouze se měnila kritéria pro názvy proměnných, vstupních a výstupních souborů a rutin. Byl například odstraněn `.Element` za každým `Inputem` a `Outputem`, protože bylo jednoznačné, že uživatel chce hledat element z dokumentu. Defaultní `Output/Input` nemusí obsahovat jméno, další přidaný již musí - viz výpis kódu 6.6.

```
1 call Input:"//xPath" -> Class::method -> Variable.name;
2 Variable.name -> Output.name:"//xPath";
3 map Input:"//xPath" -> Macro.name -> Output:"//xPath";
```

Výpis kódu 6.6: Mapovací příkazy v jazyku - finální ukázka kódu

Význam použitých jazykových kostrukcí:

- `call Input:"//xPath"-> Class::method -> Variable.name;`
  - Volání uživatelské Java metody a uložení výstupu uživ. metody do proměnné `Variable.name`.
- `Variable.name -> Output.name:"//xPath";`
  - Mapování proměnné `Variable.name` do `Outputu` (výstupního souboru).
- `map Input:"//xPath"-> Macro.name -> Output:"//xPath";`
  - Volání uživatelské rutiny `Macro.name` s jedním parametrem.

6.1.2 Přepisovací pravidla

Pravidla jsou v Xtextu definována pomocí EBNF (o EBNF v kapitole 3.3.3). Celou gramatiku lze najít v odevzdaném archivu diplomové práce: `Aplikace_a_knihovny\sources\Xtext-project\XMapping_grammar.pdf`.

Počáteční pravidlo

Jazyk vždy začíná počátečním pravidlem (výpis kódu 6.7). To obsahuje klíčové slovo `package` a za ním následuje pravidlo pro kvalifikované jméno:

- `QualifiedName: Name ('.' Name)*;`
  - `Name` je pravidlo, které říká, jaký řetězec je pro jméno validní.

Lze si všimnout, že v počátečním pravidlu je možné napsat libovolný počet standardních Java importů, které jsou definovány v `Import` pravidle. Poslední je `MappingClass` pravidlo, kde je definována celá hlavní třída - viz výpis kódu 6.2.

```
Model
: 'package' name=QualifiedName ';'
  (namespaceImports+=Import)*
  class=MappingClass
;
```

Výpis kódu 6.7: Počáteční pravidlo XMapping jazyka

Reference v pravidlech

V pravidlech se mohou vyskytovat reference. Ve výpisu kódu 6.8 lze vidět, že `Attribute` obsahuje text označený hranatými závorkami. Před znakem „`|`“ se nachází pravidlo `Variable`, které v tomto případě není voláno, ale je pouze parsováno jméno proměnné (`VariableName`). To stejné lze vidět u ostatních pravidel v ukázce.

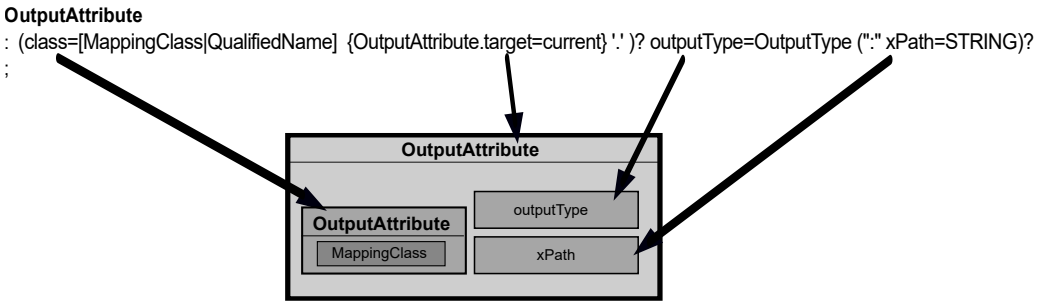
Skutečná proměnná je přiřazená k odkazu *type* až ve fázi linkování [10].

```
Attribute
: type=[Variable|VariableName] (":" xPath=STRING)?
;

OutputAttribute
: (class=[MappingClass|QualifiedName]
                  {OutputAttribute.target=current} '.' )?
   outputType=OutputType (":" xPath=STRING)?
;

OutputType
: outputRef=[Output|OutputName]
| returnRef=[Return|ReturnName]
;
```

Výpis kódu 6.8: Reference v pravidlu

Aby bylo umožněno používat reference napříč třídami, bylo potřeba v pravidlech vytvořit další Xtext konstrukci. Tato konstrukce lze najít v `Output-Attribute` a je definována složenými závorkami:

- `{OutputAttribute.target=current}`



Obrázek 6.2: Tvorba instancí v `OutputAttribute` pravidlu.

.

Tato akce se vyznačuje tím, že vytvoří novou instanci *OutputAttribute* a přiřadí k ní aktuální hodnoty proměnné *current* (viz obr. 6.2).
Zápis v Javě by vypadal následovně (výpis kódu 6.9):

```
1 OutputAttribute temp = new OutputAttribute();
2 temp.setTarget(current);
3 current = temp;
```

Výpis kódu 6.9: Java zápis `OutputAttribute.target` (převzato a modifikováno z [7]).

Díky tomu existuje stále jedna instance s hodnotami, které se následně dají použít při dalším vývoji jazyka. V praxi to znamená, že pokud je *`target`* `== null`, pak neexistuje ani reference na jinou třídu. Pokud *`target`* `!= null`, pak *target* obsahuje referenci na jinou třídu a je možné definovat rozsah platnosti objektu.

Lexikální pravidla

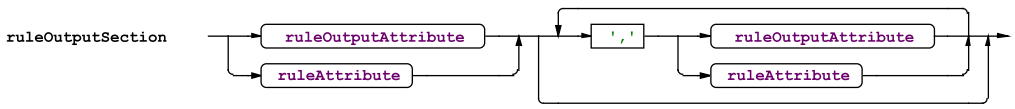
V kapitole 6.1 bylo zmíněno, že k vlastní gramatice se standardně používá Xtext gramatika `Terminals`. Tato gramatika byla nahrazena jinou - `Xtype`, která stejně jako `Terminals` definuje identifikátor, řetězec, komentáře a navíc dokáže rozpoznat typy jazyka Java. Ve vlastním souboru s gramatikou bylo přidáno pravidlo pro definování celého čísla. Všechna tato pravidla lze vidět na výpisu kódu 6.10. Tato pravidla jsou podobná regulárním výrazům.

```
//Xtype.xtext
terminal ID:
  '^'? ('a'..'z'|'A'..'Z'|'$'|'_')
  ('a'..'z'|'A'..'Z'|'$'|'_'|'0'..'9')*;

terminal STRING:
  '"' ( '\\' . /* ('b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\') */ |
  !('\\'|'"') )* '"'? |
  "'" ( '\\' . /* ('b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\') */ |
  !('\\'|"'") )* "'"?;

terminal ML_COMMENT: '/*' -> '*/';
terminal SL_COMMENT: '//' !('\n'|'\r')* ('\r'? '\n')?;

//XMapping.xtext file
terminal INT returns ecore::EInt: ('0'..'9')+;
```

Výpis kódu 6.10: Lexikální pravidla v XMapping gramatice

6.1.3 Debugování gramatiky

Ke tvorbě gramatiky neodmyslitelně patří ladění, protože se v některých případech stane, že gramatika je víceznačná.[1] Xtext někdy vypsal chybu: `The following alternatives can never be matched:3` a nešlo poznat, v jakém pravidle chyba nastala. V takovém případě bylo užitečné použít nástroj **ANTLRWorks** - dostupný z [2]. Nástroj například podporuje pro každé pravidlo zobrazení syntaktického diagramu (viz obr. 6.3).



Obrázek 6.3: Syntaktický diagram pro `OutputSection`.

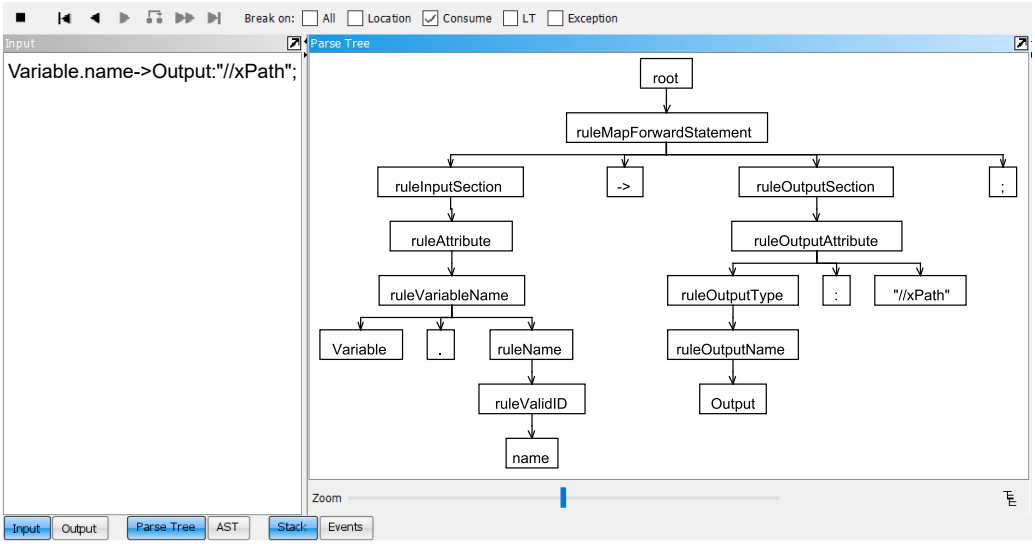
.

Aby šlo gramatiku debugovat, stačilo pouze nastavit v Xtextu generování ladící gramatiky. Ta se následně importuje do ANTLRWorks. Po zkompilování lze zadat vlastní kód a ten následně krokovat. Libovolně lze vybrat, z jakého pravidla bude jazyk začínat. Na obr. 6.4 začíná jazyk z `MapForwardStatement` pravidla. Tlačítkem ▶ lze dopředu krokovat příkazy

[1] Gramatika je víceznačná, pokud existují pro platné slovo v gramatice alespoň dva derivační stromy [29, 31].

[2] ANTLRWorks: The ANTLR GUI Development Environment. ANTLR Parser Generator [online]. Dostupné z: `https://www.antlr3.org/works/`

a tím i vidět, jak se rozšiřuje derivační strom. Tlačítkem ◀ je možné se dostat o libovolný počet kroků zpět.



Obrázek 6.4: Ukázka derivačního stromu při debugování.

6.2 Sémantika

Po gramatice následuje statická analýza. Ta se zabývala jazykovými omezeními a typovým systémem.

6.2.1 Omezení

Omezení lze najít v `com.gk_software.core.dsl.xmapping.validation`. V tomto balíčku jsou následující soubory:

- `XMappingValidator.java`
  Jedná se o předem vygenerovaný soubor - původní obecná omezení byla přepsána podle aktuální gramatiky.
- `XMappingDuplicateValidator.xtend`
  Tento soubor kontroluje duplicitu jmen objektů v DSL kódu.
- `XMappingStatementValidator.xtend`
  Soubor ověřuje, zda jsou správně napsány mapovací příkazy v DSL kódu.

V těchto třech souborech se nachází metody, které obsahují anotaci `@Check`. Xtext automaticky pozná, že může tuto metodu při validaci vyvolat. Metody mají vždy jeden formální parametr (lze vidět ve výpisu kódu 6.11).

```
//soubor: XMappingDuplicateValidator.xtend
@Check
def checkMappingClassAndPackage(MappingClass mappingClass){
  val packageName = (mappingClass.eContainer as Model).name
  if(mappingClass.name.equals(packageName)){
    val message = "Mapping class has same name as package name.";
    error(message,null, XMappingPackage.MODEL__NAME);
  }
}
//soubor: XMappingValidator.java
@Check
public void checkMappingClassName(MappingClass mappingClass) {
  int index = 0;
  if(!Character.isUpperCase(mappingClass.getName().charAt(index))) {
    String message = "Mapping class must start with capital letter.";
    error(message,null, XMappingPackage.MAPPING_CLASS);
  }
}
```

Výpis kódu 6.11: Ukázka metod s anotací `@Check`.

Jak je vidět z výpisu 6.11, tak pro daný typ lze definovat libovolný počet anotovaných metod, protože Xtext je vyvolá všechny. Uvnitř těchto metod jsou implementovány sémantické kontroly - ve výpisu kódu 6.11 je kontrolováno:

- Zda jméno hlavní mapovací třídy začíná velkým písmenem.
- Zda jméno hlavní mapovací třídy není stejné jako jméno balíčku (`package`).

Pokud sémantická kontrola selže, pak je volána metoda `error()`, která umožní ve vývojovém prostředí vypsat chybovou hlášku a vyznačit chybný kód. V projektu je použita také metoda `info()`, která podtrhne kód zeleně a vypíše informační hlášku. Existuje také metoda `warning()`, která v tomto projektu není použita.

6.2.2 Ověření mapovacích příkazů

Při ověřování mapovacích příkazů je použit typovací systém, který je popsán v kapitole 6.2.3. V souboru `XMappingStatementValidator.xtend` se kontrolují všechny mapovací příkazy, které může uživatel do svého kódu napsat.

AssignmentStatement

Tento příkaz slouží k deklaraci nebo inicializaci proměnných. Zde je důležité zkontrolovat, jestli uživatel přiřazuje hodnoty ke správným typům proměnných, protože přetypování v tomto jazyce není dovoleno. Pokud nastane například situace, kdy v kódu bude:

```
let Variable.first = 10;
Variable.first = "String";
```

Pak vývojové prostředí vyhodí chybovou hlášku: *„Variable.first is intType - can not be a new type.“*.

MapForwardStatement

Tento příkaz slouží pro přímé mapování bez volání Java metod a DSL rutin. Zde je kontrolováno:

- Obsahuje `Input` a `Output` xPath výraz?
- Obsahuje proměnná, která je typu struktury, xPath výraz?
- Má vstupní a výstupní prvek odlišný typ?
- Má přímé mapování stejný počet vstupních a výstupních prvků?

Pokud je alespoň na jednu otázku odpověď „ne“, pak vývojové prostředí vypíše chybovou hlášku a je potřeba chybu v DSL kódu opravit.

MapStatement

Tento příkaz slouží k obecnému mapování, ve kterém se volá DSL rutina. Validace využívá některé metody, které byly použity u předchozího příkazu:

- Neexistuje rekurze?
- Obsahuje `Input` a `Output` xPath výraz?
- Obsahuje proměnná, která je typu struktury, xPath výraz?
- Sedí počet vstupních prvků příkazu s formálními parametry DSL rutiny?
- Vrací rutina stejný počet objektů jako je ve výstupu příkazu počet prvků?
- Jsou všechny výstupní objekty rutiny v pořádku?
- Jsou všechny `Input` a `Output` objekty ze stejné třídy jako volaná DSL rutina?
- Mají vstupní a výstupní prvky správný typ?

CallStatement

Tento příkaz slouží k volání uživatelských Java metod. Ke kontrole jsou použity již vytvořené metody:

- Obsahuje `Input` a `Output` xPath výraz?
- Obsahuje proměnná, která je typu struktury, xPath výraz?
- Mají vstupní a výstupní prvky správný typ?
- Sedí počet vstupních prvků příkazu s formálními parametry uživatelské Java metody?
- Existuje pouze jeden výstupní prvek příkazu?

6.2.3 Typový systém

Celý typový systém jazyka lze najít v balíčku: `com.gk_software.core.dsl.xmapping.typing`. V tomto balíčku se nachází abstraktní třída `Type`, která má dva atributy (`String name` a `String codeType`) a k nim dané gettery.

Dále lze v balíčku vidět třídy reprezentující všechny typy, které jazyk podporuje. Třídy dědí od zmíněné abstraktní třídy `Type` a jsou návrhového vzoru singleton, aby pro každou třídu existovala pouze jedna instance (viz výpis kódu 6.12).

```
1  package com.gk_software.core.dsl.xmapping.typing;
2  /** Singleton Int type */
3  public final class IntType extends Type{
4    /** instance */
5    public static final IntType INSTANCE = new IntType();
6    /** name */
7    private static final String NAME = "intType";
8    /** code name */
9    private static final String CODE_TYPE = "int";
10   /** Constructor */
11   private IntType() {
12     super(NAME, CODE_TYPE);
13   }
14 }
```

Výpis kódu 6.12: Singleton pro celočíselný typ.

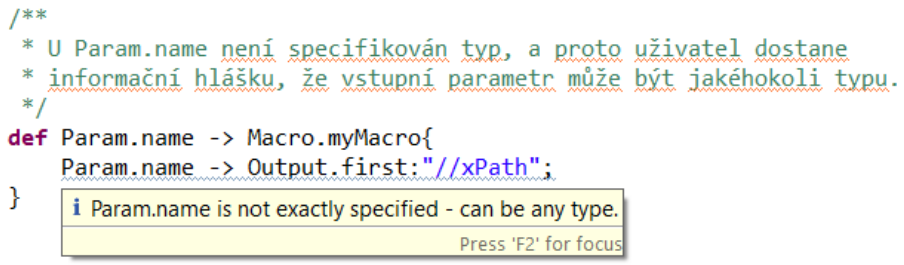
Dále je v balíčku třída `XMappingTypeProvider.xtend`, která obsahuje všechny metody pro zjištění typu objektu. V této třídě jsou statické atributy, které obsahují instance dostupných typů (příklad ve výpisu kódu 6.13).

```
class XMappingTypeProvider {
  /** singleton string type */
  public static val stringType = StringType.INSTANCE
  //.....
  /** singleton not type */
  public static val notType = NotType.INSTANCE
  //.....
  /** singleton not specified type */
  public static val notSpecified = NotSpecified.INSTANCE
```

Výpis kódu 6.13: Statické atributy v *type provideru*.

V ukázce 6.13 si lze všimnout typů `notType` a `notSpecified`. Při statické analýze se může stát, že typ proměnné není ještě definován, a proto *type provider* vrátí, že není žádného typu.

Speciální akcí je, že *type provider* ve statické analýze nedokáže rozpoznat konkrétní typ proměnné a pak vrací, že typ je nespecifikovaný. To se stává v případech, kdy uživatel volá rutinu, která má vstupní parametr a tento parametr použije v libovolném příkazu v těle rutiny - obr. 6.5.



Obrázek 6.5: Typ „notSpecified“

V tomto případě se dá typ zjistit až dynamicky a to těsně před vygenerováním Java kódu. Pokud typy nesedí, pak se objeví v konzoli chyba a generování kódu je neúspěšné. *Type provider* kvůli dynamickému zjištění typu obsahuje zásobník, do kterého se vkládají aktuální příkazy (`Statementy`), ze kterých lze dohledávat typy objektů. Zásobník je využit pouze tehdy, kdy má být vygenerován Java kód. Pokud se jedná o statickou analýzu, tak zásobník není využit a metoda vrátí typ „notSpecified“.

Jestliže *type provider* najde `AssignmentStatement`, kde je objekt inicializován, pak zjistí typ podle AST. `IntType`, `StringType` a další jsou třídy z gramatiky, které vytvořil generátor ANTLR.

```
/**
* Finds type for object in AssignmentStatement
* @param statement assignment statement
* @param object object
* @return object's type
*/
def dispatch Type typeFor(AssignmentStatement statement, EObject object) {
 //gets initialization
 val value = statement.initialization.assignment
 switch value {
   com.gk_software.core.dsl.xmapping.xMapping.IntType: return intType
   com.gk_software.core.dsl.xmapping.xMapping.StringType: return stringType
   com.gk_software.core.dsl.xmapping.xMapping.DoubleType: return doubleType
 }
 if(value.structure !== null){
   return structureType
 }
}
```

Výpis kódu 6.14: Zjištění typu objektu z `AssignmentStatement`.

6.3 Scoping

Scoping by se dal definovat v češtině jako rozsah platnosti objektů. Mapovací blok `StartMapping-EndMapping` a zároveň také všechny rutiny mají svůj vlastní kontext (*scope*). V DSL kódu nelze mít globální proměnné - každá proměnná je lokální pro daný blok. Kdekoliv DSL kódu lze odkazovat na `Input` a `Output` ze stejného souboru nebo volat metodu z jiného souboru (za určitých podmínek, které kontroluje validátor). Třídy lze najít v balíčku: `com.gk_software.core.dsl.xmapping.scoping`.

6.3.1 Index

Standardně je každý objekt, kterému lze dát jméno, automaticky v programu viditelný a je na něj možné odkazovat. Tento mechanismus je v Xtextu zprácován pomocí indexu, který ukládá informace o všech objektech v každém zdroji (*resource*). Do indexu není uložen samotný objekt, ale objekt typu `IEObjectDescription` [7].

Problém 1: Při implementaci globálního kontextu, který se spoléhá na index, nebyly některé objekty viditelné.

Možné řešení: Přepsat defaultní metodu `createEObjectDescriptions()`, která je vždy volána, když je index vytvořen, nebo jsou změněny zdroje. Proto vznikla třída `XMappingResourceDescriptionStrategy.java`, ve které

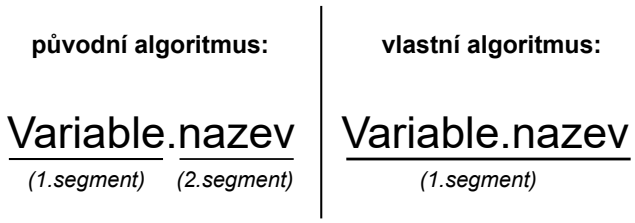
byla vlastní implementace zmíněné metody. Ukázalo se, že problém se nachází jinde, ale tato třída již v projektu zůstala, protože může být užitečná ve chvíli, kdy by byl jazyk dále rozšiřován.

Kvalifikovaná jména

Výchozí implementace indexu používá mechanismus založený na jménech [7]. Defaultně je k tomu využita třída `DefaultDeclarativeQualifiedNameProvider.java`, která slouží jako *name provider*. Z této třídy lze získat pro každý objekt jeho plně kvalifikované jméno - např.: `src.my.code.MyClass`, pro hlavní mapovací třídu se jménem „MyClass“.

V projektu byl vytvořený vlastní *name provider*, který dědí od zmíněné třídy. Ten je připraven k použití, pokud by byl jazyk dále rozšiřován.

Řešení problému 1: Důležitou vlastní třídou je `XMappingQualifiedNameConverter.java`, kde vznikají kvalifikované názvy objektů. Díky této třídě byl vyřešen problém s indexem, protože bylo zjištěno, že kvůli tečkové notaci jazyk nedokázal rozpoznat, co všechno je pouze název objektu. Příklad je na obrázku 6.6:



Obrázek 6.6: Kvalifikovaná jména pro proměnnou.

Na obrázku lze vidět, že původní algoritmus rozděluje název na dva segmenty, a pokud z tohoto názvu vznikne plně kvalifikované jméno, pak Xtext pracuje s tím, že objekt nese jméno: „nazev“. V jazyce XMapping musí být jméno proměnné „Variable.nazev“ a přesně tento princip je zajištěn ve vlastním *konvertoru*. I díky tomu začaly být všechny objekty viditelné.

Importy

Díky globálnímu kontextu bere Xtext v úvahu i jmenné prostory (*namespaces*). Pokud je v DSL kódu import `java.lang.*`, pak je možné odkazovat na objekty v tomto balíčku přímo a není potřeba používat plně kvalifikované jméno. V Xtextu je tento algoritmus standardně vytvořen.

K výchozímu algoritmu byla přidaná schopnost se automaticky odkazovat bez plně kvalifikovaného jména na objekty ze všech zdrojů, které se vyskytují ve stejném balíčku. V projektu byla vytvořena třída `XMappingImportedNamespaceAwareLocalScopeProvider.java`, která poskytuje vlastní implementaci metody `internalGetImportedNamespaceResolvers()`, která tuto vlastnost poskytuje.

6.3.2 Scope Provider

V projektu je vytvořen i vlastní *scope provider*, který definuje rozsah platnosti všech objektů. K definování rozsahu používá *scope provider* metodu: `IScope getScope (EObject context, EReference reference)`, kde formální parametry jsou:

- `context` - konkrétní objekt,
- `reference` - konkrétní reference na objekt v kódu.

V této metodě se nachází množina podmínek, ve kterých se testuje o jakou referenci se jedná. Pokud je reference nalezena, volají se metody pro definování rozsahu - buď již vygenerované Xtextem, nebo vlastní.

```
1  // Reference to Class:  -->[<MappingClass>].Marco.<macro_name>
2  // example:             MyMappingClass.Macro.name
3  if(reference == XMappingPackage.Literals.ROUTINE_SOURCE__CLASS) {
4    //default scope method
5    return namespaceScopeProvider.getScope(context, reference);
6  }
7  // Reference to Routines    -->[<RoutineName>] = <Some_Routine>.<name>
8  // example:                 Macro.myMacro | Filter.myFilter | etc..
9  if(reference == XMappingPackage.Literals.ROUTINE_SOURCE__SOURCE) {
10   //custom scope method
11   return routinesScope(context,reference);
12 }
```

Výpis kódu 6.15: Ukázka podmínky v metodě `getScope()`.

Jestliže objekt nemůže být viditelný v daném kontextu - například pokud bychom chtěli proměnnou z hlavní třídy použít v rutině - vývojové prostředí vypíše chybovou hlášku *Couldn't resolve reference to Variable 'Variable.name'*. Zvýraznění chyby ve vývojovém prostředí je stejné jako při validaci.

7 Implementace

Uživatel nyní může napsat vlastní DSL kód, ale nebude mít žádný výstup. Dalším krokem je implementace generování kódu a interpreta. Než bude generátor a interpret vysvětlen, zaměříme se na pomocné třídy.

7.1 Pomocné třídy

Všechny pomocné třídy lze najít v balíčku: `com.gk_software.core.dsl.x-mapping.helpers.util`. Tyto třídy pomáhájí v různých částech projektu k usnadnění práce a odstraňují duplicitu kódu.

Jak při validaci, tak i při generování kódu se ve formálních parametrech metod vyskytují rozhraní z meta-modelu (obecně `EObject` nebo konkrétní pravidlo - např. `CallStatement`). V těle metod se často ověřuje, zda třída obsahuje instanci jiné třídy. Každým ověřením se kód větví a začíná být nepřehledný (ukázka větvení ve výpisu kódu 7.1).

```
1  def void checkCallReturnOutput(CallStatement statement){
2    val outputSection = statement.outputSection
3    if(outputSection !== null){
4      val outputAttributes = outputSection.outputAttributes
5      if(outputAttributes !== null){
6        ....
7      }
8    }
9    ....
10 }
```

Výpis kódu 7.1: Ukázka větvení při práci s EMF třídou.

Jestliže pracujeme v metodě s `EObject`, pak je nutné často ověřovat, jestli je instance požadovaného typu - např.: `if(object instanceof InputType)`.

Často se kód opakuje, protože objekty nejsou použity pouze v jedné metodě, ale jsou používány napříč celým projektem. Proto vznikly pomocné třídy, ve kterých jsou časté sekvence kódu, které se v implementaci používají. Dále lze najít metody, které jsou použity zřídka, ale výrazně by znepřehlednily kód. V projektu jsou pomocné třídy:

- `XMappingEObjectUtil.xtend`
  Tato třída pomáhá pracovat s `EObject`. Důležitou metodou je získání instance „základního“ objektu (např. dokáže získat `Param` instanci z `InputAttribute` třídy).

- `XMappingGeneratorUtil.xtend`
  Tato třída je pomáhá generátoru kódu - například:

  – Transformuje název DSL rutiny na validní jméno Java metody.

  – Z `package` v DSL kódu vytvoří cestu pro souborový systém.

- `XMappingInterpretObjectUtil.xtend`
  Tato třída je využita u interpretace. Ve třídě se pracuje s `EObject` objektem, pro který například dokáže na základě typového systému vypsat odpovídající Java typ.

- `XMappingInterpretUtil.xtend`
  Tato třída je využita u interpretace. V metodách se nepracuje s třídou `EObject`, ale vždy s konkrétním pravidlem.

- `XMappingStatementUtil.xtend`
  Tato třída je využita u interpretace. Obsahuje složitější metody, které pracují s mapovacími příkazy.

7.2 Generování kódu

Poté, co má uživatel vlastní DSL script, je potřeba vygenerovat spustitelný Java kód. Xtext automaticky vytvořil generátor v balíčku: `com.gk_software.core.dsl.xmapping.generator`. Tento generátor není v projektu využit, protože neumožňoval generovat kód pro více souborů v projektu. Místo něj byla vytvořena třída generátoru `XMappingGenerator`, která implementuje rozhraní `IXMappingGenerator` a umožňuje generovat kód již pro více souborů.

Rozhraní `IXMappingGenerator` obsahuje dvě metody:

- `public void doGenerate(ResourceSet resources, IFileSystemAccess2 fsa, IGeneratorContext context);`
  - Hlavní metoda pro generování kódu.

- `public void setListener(IMessageListener listener)`
  - Metoda pro nastavení *message listeneru*.

Pokud dojde ve vývojovém prostředí ke změně DSL kódu a uživatel své změny uloží, je automaticky zahájeno generování. Xtext framework volá zmíněnou metodu `doGenerate()`.

Než dojde k procesu transformace kódu, je celý DSL script zkontrolován. Ověřuje se, že se v kódu nevyskytla chyba, která nemohla být objevena statickou analýzou (více v kapitole 6.2.3 - typ „notSpecified“). Pokud nastane při kontrole kódu chyba, je do konzole vypsána chybová hláška a procházení kódu končí. Generování do Java kódu vůbec neproběhne.

7.2.1 Maven

Se zadavatelem jsem v této fázi vývoje probíral, jak bude vypadat nástroj s podporou pro Maven. Tímto nástrojem je generátor, který dokáže vygenerovat spustitelný Java kód, ve kterém je zapsáno konkrétní mapování. Doménově specifický jazyk, který se několik let ve firmě používá, je založen na starší verzi Xtextu a k vygenerování Java souborů potřebuje Maven. Příkazem `mvn clean install` se Java kód vygeneruje.

Se zadavatelem jsme se domluvili na tom, že budu v implementaci pokračovat podle aktuálního Xtext návodu implementace generátoru, který již nemusí používat Maven k vygenerování Java kódu. S novějšími verzemi frameworku je možné vygenerovat kód automaticky po uložení DSL souboru a tuto funkcionalitu mám v XMapping jazyku.

Ačkoli se Maven nepoužívá k vygenerování Java kódu, může i přesto uživatel psát svůj DSL kód v Maven projektu. Do `pom.xml` může přidat `<dependency>` na svou knihovnu s metodami, které chce při mapování použít. Generátor si pro tyto metody dokáže najít jejich kvalifikované názvy.

7.2.2 FileCode

S generováním je spojena pomocná třída `FileCode`. Tato třída reprezentuje jeden `.java` soubor. Tato třída nese všechny informace, které mají být zapsány do souboru (viz výpis kódu 7.2). Instance vzniká ve chvíli, kdy je rozhodnuto, že vznikne nový soubor.

```
class FileCode {
  /** File package */
  String packageCode;
  /** Inputs which class contains */
  List<InputImport> inputs = new ArrayList<InputImport>();
  /** Outputs which class contains */
  List<OutputImport> outputs = new ArrayList<OutputImport>();
  /** Class name */
  String className;
  /** Imports to add (not defined in .xmapping file) */
  List<String> importsCode;
  /** charSequence of class main block  */
  CharSequence blockCode;
  ....
}
```

Výpis kódu 7.2: Třída `FileCode` a její atributy.

Tím, že je kód procházen řádek po řádku a sleduje se, zda v nějakém příkazu nenastala chyba, je možné rovnou zkontrolovaný kód převádět do Javy a zapsat do příslušné instance `FileCode` v paměti. K převodu DSL mapovacího příkazu na příkaz Javy byl vytvořen interpret (více v kapitole 7.3). Příkazy jsou postupně zapisovány do atributu *blockCode*.

7.2.3 Konzole

Veškeré úpravy vývojového prostředí pro uživatele lze provést v modulu: `com.gk_software.core.dsl.xmapping.ui`. Pro XMapping jazyk je upravena konzole, která dokáže vypsat informační a chybové hlášky z generování kódu.

Ve třídě `MessageProvider` jsou 2 atributy typu `IOConsoleOutputStream`. První atribut třídy je proud (*stream*) pro informační hlášení a druhý pro chybové hlášení. Ve třídě `PluginConsole` probíhá nastavení obou *streamů*, aby konzole mohla vypsat text.

Třída `MesaggeProvider` implementuje rozhraní `IMessageListener`, které obsahuje metody:

- `void writeOutput(String message, String where)`
  - metoda pro výpis informační hlášky.
- `void writeError(String message, String where)`
  - metoda pro výpis chybové hlášky.

Xtext před každým generováním kódu volá defaultní metodu `invokeGenerator()`. V této metodě je generátoru nastaven *listener* a také je volána metoda `doGenerate()`.

7.2.4 Generátor

Pokud neobsahuje DSL script žádné chyby, je převeden do Javy a zapsán do souboru. K vytvoření Java třídy nebo metody jsou používány šablony, do kterých jsou některé části kódu dynamicky přidávány.

Na obrázku 7.1 lze vidět šablonu pro generování DSL rutiny. Vše, co je označeno v šabloně šedě, je vždy staticky vygenerováno. Znaky „«“ a „»“ se používají k přepínání mezi staticky generovaným kódem a dynamicky generovaným kódem. Mezi těmito znaky může být:

- proměnná: `«name»`,
- volání metody: `«editPath(fileCode.outputs.get(i).path)»`,
- podmínka: `«IF block.statement !== null» «ENDIF»`,
- cyklus: `«FOR statement : block.statement» «ENDFOR»`.

```
/**
 * Template for routine
 * @param fileCode file which contains all information about routine
 * @return charSequence routine template
 */
def compileRoutine(FileCode fileCode)
'''
    package «fileCode.routinePackage»;
    «var name = fileCode.className»

    «FOR i : fileCode.importsCode»
        «i»;
    «ENDFOR»

    public class «name» {

        //variable for routines results
        private static «SourcesEnum.GENERAL_ARRAYLIST.value»<«SourcesEnum.CODE_RETURN_VALUE.va

        «routineBlocks.get(fileCode.getClassFullName.replace(".statement","")).blockCode»
    }

'''
```

Obrázek 7.1: Použití šablony při generování Java třídy.

7.3 Interpret

V gramatice je pravidlo `Statement`, které reprezentuje jakýkoli příkaz, který se dá použít v bloku hlavní třídy nebo rutiny:

- `AssignmentStatement`,
- `MapForwardStatement`,
- `CallStatement`,
- `MapStatement`.

Příkazy je potřeba převést do Java kódu, aby mohl následně uživatel mapovat XML soubory. Proto vznikla třída `XMappingInterpret`, která dědí od abstraktní třídy `AbstractXMappingInterpret`. Tato abstraktní třída obsahuje metody `interpret()`, které mají vždy jeden formální parametr. Na obrázku 7.2 lze vidět zmíněná abstraktní třída a její metody.

```
abstract class AbstractMappingInterpret {

    /**
     * Interprets an assignment statement
     * @param statement assignment statement
     */
    def dispatch boolean interpret(AssignmentStatement statement){
        return false;
    }
    /**
     * Interprets a map forward statement
     * @param statement map forward statement
     */
    def dispatch boolean interpret(MapForwardStatement statement){
        return false;
    }
    /**
     * Interprets a call statement
     * @param statement call statement
     */
    def dispatch boolean interpret(CallStatement statement){
        return false;
    }
    /**
     * Interprets a map statement
     * @param statement map statement
     */
    def dispatch boolean interpret(MapStatement statement){
```

Obrázek 7.2: Abstraktní třída `AbstractMappingInterpret`.

Třída `XMappingInterpret` přepisuje dané metody na konkrétní implementace. Aby generování Java kódu nebylo příliš složité, bylo vytvořeno další rozhraní: `IXMappingInterpretQueue`, ve kterém jsou metody, které lze při interpretaci použít. Toto rozhraní implementuje třída `XMappingInterpretQueue`.

V implementaci metod `interpret()` se často pracuje s AST, díky kterému lze odvodit, jak bude příkaz v Javě vypadat. Např. u `AssignmentStatement` lze zjistit, zda daný DSL příkaz deklaruje nebo inicializuje proměnnou: `if(statement.initialization !== null)`. Tvorbu Java kódu zajišťují pak implementované dílčí metody z `XMappingInterpretQueue`, kterými lze složit příkaz v Javě.

Příklad: Potřebujeme deklarovat proměnnou v Javě, pak by mohl být v metodě *interpret()* následující kód:

```
pushObjectType(object);
  pushCustomCode(" ");
pushObjectName(object);
  pushCustomCode(";");
```

Příkaz je rozdělen na jednotlivé části. Každá metoda je vyhodnocena podle vlastní implementace - např: `pushObjectType()` zjistí typ objektu a tento typ se v textové podobě vloží do fronty (viz výpis kódu 7.3). Podle příkladu by byla dále vložena prázdná mezera, jméno objektu a středník. Metoda `interpret()` je pro každý příkaz dynamicky volána v šabloně generátoru (o šablonách více v kapitole 7.2.4), ale nevrací žádný kód. Veškerý kód je ve zmíněné frontě, ze které jsou následně všechny prvky postupně odebírány a zapsány do šablony. Celý vygenerovaný kód (šablona + dynamicky přidaný) je pak uložen do konkrétní instance třídy `FileCode`.

```
override String pushObjectType(EObject object){
  val code = getObjectType(object)
  if(code !== null){
    //add import of object type if is needed
    checkImports(code);
    //adds into queue
    var request = new InterpretRequest(code,RequestType.REQUEST_COMMAND)
    request.pushRequest
  }
  //returns code because it can be used in interpret() method
  return code
}
```

Výpis kódu 7.3: Vlastní implementace metody `pushObjectType()`.

7.4 XMapping knihovna

Generátor kódu společně s interpretem vytvoří Java kód, který bude spustitelný až ve chvíli, kdy bude k dispozici **XMapping knihovna**. Samotné ma-

pování XML souborů je zajištěno právě v této knihovně, a proto DSL mapovací příkazy často generují různé metody, které jsou v této knihovně veřejné. Pro tuto knihovnu byl založen samostatný projekt.

7.4.1 Dokumenty

V projektu XMapping knihovny se nachází abtraktní třída **`IOFile`**. Ta obsahuje atributy: jméno, cestu k souboru, File a Document objekt. K těmto atributům jsou vytvořeny gettery a settery. Dále tato třída obsahuje abstraktní metodu `loadDocument()` a metodu, která dokáže poznat, zda daný soubor existuje.

Vstupní XML soubor je reprezentovaný třídou `FileInput`, která dědí od abstraktní třídy `IOFile`. Výstupní XML soubor je reprezentován třídou `DocumentOutput`, která také dědí od této abstraktní třídy. Obě dvě třídy mají vlastní implementaci metody `loadDocument()`, ve které je vytvořen atribut typu Document.

Třída pro výstupní soubor navíc obsahuje své vlastní metody a atributy, které jsou dále použity při mapování souborů.

Parsování souboru

U `FileInputu` není problém s parsování XML souboru. Parsování souboru a následný zápis do Document atributu vypadá následovně (výpis kódu 7.4).

```
1 if(super.documentExists()){
2   //parses the file into Document
3   FileInputStream fis = new FileInputStream(super.getFile());
4   super.setDocument(builder.parse(fis));
5 }else{
6   //empty document
7   super.setDocument(builder.newDocument());
8 }
```

Výpis kódu 7.4: Parsování vstupního XML souboru.

U *.xsd* souborů, které se v XMapping jazyku hojně používají, bylo potřeba udělat vlastní parsování. K tomu vznikly třídy: `XsdReader` a `SchemaSaxHandler`.

Třída `XsdReader`: Tato třída obsahuje jedinou metodu - `read()`, která vytvoří instanci námi definovaného *handleru* a nastaví jej pro `XMLReader`, který následně parsuje soubor.

Třída `SchemaSaxHandler`: Handler, ve kterém je řešeno parsování. Tato třída dědí od `DefaultHandler`, a proto přepisuje metody:

- `startElement()`
  Pokud *parser* objeví, že v souboru začíná nový prvek, je automaticky volána tato metoda. V této metodě se nachází podmínky, podle kterých lze zjistit, na jaký prvek *parser* narazil a na základě toho provést danou akci.
- `endElement()`
  Tato metoda se volá ve chvíli, kdy *parser* dojde na konec objeveného prvku. V metodě jsou také podmínky, které identifikují daný prvek.

Pro každý element `.xsd` souboru je vytvořena instance třídy `StructureNode`, ve které jsou definované:

- atributy,
- *complexType* a případně jeho název,
- potomci (objekty typu `StructureNode`).

Stromová struktura souboru

Tím, že se dá od *root* elementu (`StructureNode` objektu) dostat kamkoliv na jiný element, je po dokončení parsování volána rekurzivní metoda `makeTree()`, která vytvoří stromovou strukturu celého `.xsd` dokumentu. V konstruktoru `DocumentOutput` je následně volána metoda, která na základě vytvořeného stromu vytvoří konkrétní instanci Dokumentu, pro kterou je možné používat xPath. Tato stromová struktura se dále využívána při mapování souborů.

7.4.2 xPath

Hlavní vlastností jazyka je, že lze hledat pomocí xPath výrazů elementy v souboru. Pro uživatele jsou dostupné metody ve třídě `XPathResolver`. XMapping jazyk umí všechny tyto metody sám vygenerovat. Ve výpisu kódu 7.5 je vidět implementace metody, která mapuje data ze vstupního souboru do výstupního.

```
/**
* Main method for mapping input into output by xPath
* @param fileInput file input file
* @param inputXPath xPath for input file
* @param documentOutput document output object
* @param outputXPath xPath for output document
*/
public static void queryInputOutput(FileInput fileInput, String inputXPath,
                        DocumentOutput documentOutput, String outputXPath){

  try{
    basicQuery(fileInput,inputXPath,documentOutput,outputXPath);
  }catch(Exception e){
    e.getStackTrace();
  }
}
```

Výpis kódu 7.5: Mapování dat vstupního souboru do výstupního.

V ukázce kódu 7.5 je metoda `basicQuery()`, ve které se pracuje s xPath výrazy - nejdřív se kontroluje, zda existuje element výstupního souboru - pokud ano, pak je vyhodnocen i vstupní xPath výraz. Pokud nic neselže, jsou data ze vstupního souboru vložena do vlastních objektů `ExtendedNode`, které jsou následně uloženy do konkrétního uzlu stromové struktury (stromová struktura v kapitole 7.4.1). Konkrétní uzel stromu je zjištěn při vyhodnocení xPath výrazu pro výstupní soubor.

Metody, které může uživatel použít:

- `queryOutputOutput()`
  Metoda mapuje data z výstupního souboru (v kódu může být definována část Dokumentu jako proměnná) do výstupního souboru.

- `queryFunctionOutput()`
  Metoda přiřazuje vlastní uživatelskou metodu k výstupnímu souboru, která bude následně použita při mapování.

- `queryInputOutput()`
  Metoda mapuje data ze vstupního souboru do výstupního souboru.

- `queryValueOutput()`
  Metoda může vložit do výstupního souboru řetězec, celočíselné, nebo desetinné číslo.

Všechny tyto metody pracují se stromovou strukturou, ve které jsou všechna data zapsána. V této fázi zatím neexistuje žádný nový soubor, který by namapovaná data obsahoval.

7.4.3 Uživatelské metody

Uživatel si může nadefinovat vlastní metody, které pak lze při mapování použít. Tyto metody musí být veřejné a statické. Dále musí být pro tyto metody vytvořena vlastní Java knihovna, kterou lze následně importovat v XMapping projektu. Při psaní `CallStatementu` je tato knihovna dostupná a lze i přes nápovědu „*CTRL + mezerník*“ ve vývojovém prostředí vidět všechny její třídy a metody.

```
1  public class Nodes {
2
3    public static String exists(Node node){
4      if (node == null)
5        return "false";
6      return "true";
7    }
8    public static Node filterNodeOnTrue(Node node,
9                                        String isTrue){
10     try{
11       boolean value = Boolean.parseBoolean(isTrue);
12       if(value){
13         return node;
14       }
15     }catch(Exception e){}
16     return null;
17   }
18 }
```

Výpis kódu 7.6: Uživatelem napsané metody `filterNodeOnTrue()` a `exists()` pro mapování.

Uživatelská metoda může mít formální parametry typu `org.w3c.dom.Node` nebo `String`. Řetězec zastupuje obsah daného uzlu. `Node` může být užitečný například pro metodu, která vrátí „true“/„false“, zda daný uzel v dokumentu existuje (viz výpis kódu 7.6).

Výstupem musí být řetězec, nebo uzel. Ačkoliv může být boolean hodnota použita jako obsah uzlu, je nutné tuto hodnotu stále vrátit jako řetězec. Pokud je v metodě vrácen *null*, pak mapování přes funkci pro daný uzel končí a uzel je ve stavu jako před voláním dané funkce.

7.4.4 Vytvoření nového XML souboru

Posledním vygenerovaným příkazem v hlavní metodě je `output.generateDocument(false)`. Metoda je volána pro každý výstupní soubor. V ní na-

jdeme metodu `createXmlDocument()`, díky které je vytvářen požadovaný výstupní soubor (výpis kódu 7.7).

```
/**
* Main function for creating Document object with XML data
* @param tree tree with data
* @param node main (Document) node
*/
public static void createXmlDocument(Tree tree, Node node){
  TreeElement el = tree.getRootElement();
  //creates a XML Document object
  el.getUsedNodes().clear();
  LOGGER.info("Creating XML document.");
  createXml(el,new ArrayList<ElementInformation>(),node,null);
}
```

Výpis kódu 7.7: Mapování dat vstupního souboru do výstupního.

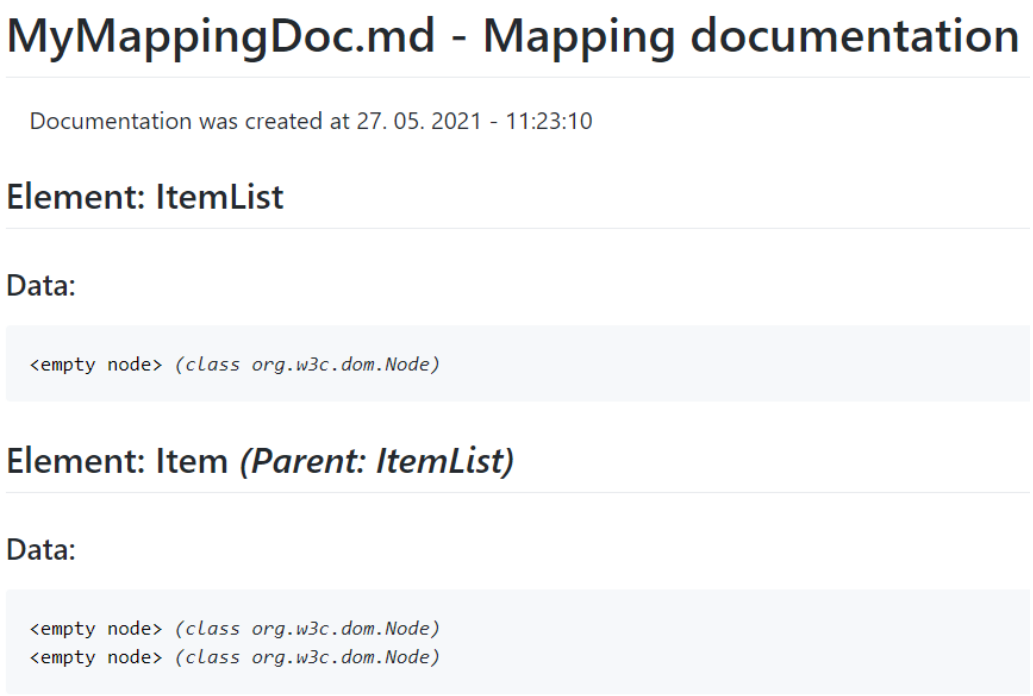
Ve výpisu 7.7 lze vidět metodu `createXml()`. Jedná se o rekurzivní metodu, ve které je celý algoritmus mapování - zde je zjednodušený postup:

- **Krok 1**: Zjisti, zda je v uzlu uživatelská metoda.
  - Pokud ANO: vykonej metodu a jdi na krok 2.
  - Pokud NE: jdi na krok 2.
- **Krok 2**: Obsahuje uzel nějaká data?
  - Pokud ANO: jdi na krok 3.
  - Pokud NE: proveď krok 6.
- **Krok 3**: Jsou již v rodičovských uzlech zapsána nějaká data?
  - Pokud ANO: jdi na krok 4.
  - Pokud NE: Jdi na krok 5.
- **Krok 4**: Souvisí data z rodičovských uzlů s aktuálními daty?
  - Pokud ANO: zjisti, která konkrétní data to jsou a proveď krok 6.
  - Pokud NE: Jdi na krok 5.
- **Krok 5**: Pro všechna data v uzlu proveď krok 6.
- **Krok 6**: Pokud je to možné, vytvoř element ve výstupním souboru, zapiš data a případně i atributy elementu. Pro každý potomek volej metodu `createXml()` (začni od 1. kroku).

7.4.5 XMapping dokumentace

S knihovou souvisí generování dokumentace k dané transformaci. Pokud je soubor úspěšně transformován a vytvořen, pak vzniká i dokumentace. Tato dokumentace má ukázat, jaký obsah byl převeden a v jakých uzlech nového XML dokumentu se nachází.

Vygenerovaná dokumentace obsahuje *.md* příponu. V souboru je použit značkovací jazyk Markdown, který dokáže jednoduše transformovat prostý text na formátovaný text. Výhodou je, že syntaxi Markdownu lze míchat s jazykem HTML. Markdown editory umožňují zobrazit již naformátovaný text, který lze vidět na obrázku 7.3.



Obrázek 7.3: Ukázka *.md* dokumentu.

Pokud je zobrazen jen zdrojový text, lze i z něj vyčíst důležité informace. V textu je kombinovaná syntaxe Markdown a HTML, aby byla výsledná dokumentace přehlednější, pokud je zobrazena v Markdown editoru.

```
# MyMappingDoc.md - Mapping documentation
Documentation was created at 27. 05. 2021 - 11:23:10

## Element: ItemList

### Data:
<pre><code>
  <span style="color:black">&lt;empty node&gt;</span>
  <i>(class org.w3c.dom.Node)</i>
</code></pre>

## Element: Item *(Parent: ItemList)*

### Data:
<pre><code>
  <span style="color:black">&lt;empty node&gt;</span>
  <i>(class org.w3c.dom.Node)</i>
  <span style="color:black">&lt;empty node&gt;</span>
  <i>(class org.w3c.dom.Node)</i>
</code></pre>
```

Výpis kódu 7.8: Ukázka zdrojového textu v.*md* souboru.

8 Testování a nasazení

S postupným vývojem bylo prováděno testování různých částí jazyka. Když byla vyvíjena gramatika, vznikaly testy na *parser*. Pokud byl vytvořen generátor, tester zkoušel vygenerovat validní DSL kód a sledoval chyby ve vygenerovaném kódu.

8.1 Jednotkové testy

Všechny jednotkové testy lze najít v modulu `com.gk_software.core.dsl.x-mapping.tests`. Na jednotkové testy se používá framework **Junit 5**. K definování, že se jedná o testovací metodu, je nutné přidat anotaci `@org.junit.Test`. Třídy, ve kterých jsou tyto metody, lze spustit přes předem připravenou konfiguraci „*Junit test*“.

V testech se používají `assert` metody, které jsou poskytovány JUnit frameworkem. Ve výpisu kódů 8.1 lze vidět, že se testuje *parser* a jsou použity metody:

- `assertNotNull()` - testuje se, zda výsledek není *null*.
- `assertEquals()` - testuje se, zda získané jméno odpovídá očekávánému jménu.

```
1 @Test
2 def void testPackage() {
3   val result = parseHelper.parse('''
4     package my.src.package;
5   ''')
6   Assertions.assertNotNull(result);
7   Assertions.assertEquals("my.src.package",result.name);
8 }
```

Výpis kódu 8.1: Jednotkové testování *parseru*.

Díky testům šly objevit rychle chyby, které při přidávání nových funkcionalit často vznikaly. Pak bylo potřeba se zamyslet, zda se opravdu jedná o chybu, nebo ne.

Pro *validator* jednotkové testy neexistují, protože bylo zjištěno, že aktuální verze Xtextu (2.24) obsahuje bug, který nedokáže v testech rozpoznat chybu. Ve výpisu kódu 8.2. lze vidět test, na kterém je tato chyba zjištěna.

Pokud by uživatel napsal kód z testu ve vývojovém prostředí, pak IDE vypíše hlášku *„Return must be in routine block.“*. U tohoto testu se tak nestane a test je *failed*. Test projde, pokud se testuje metoda `assertNoErrors()`, což není správně. Chyba je oficiálně oznámena zde: [1].

```
@Test
def void testReturnIsInRoutine() {
    val result = parseHelper.parse('''
    package jj.dsad.sad;
    MappingClass Task1
    Input:"ZSORTLST_WBBDLD05.xsd";
    Output.first:"masterData_Item.xsd";

    StartMapping
      // Return.0 can not be in main block
      let Return.0;
    EndMapping
  ''')

  result.assertError(XMappingPackage.Literals.RETURN, null, "
   Return must be in routine block.")
  //result.assertNoErrors()
}
```

Výpis kódu 8.2: Jednotkový test pro *validator*.

Momentálně již existuje vyšší verze Xtextu (2.25), kde je tento bug opraven. Jako vylepšení do budoucna může být migrace na vyšší verzi Xtextu.

8.2 Funkční testování

K testování DSL bylo dále použito funkční testování - testuje se DSL, jako by jej používal koncový uživatel. Toto testování bylo zadáno studentovi ze studentské laboratoře ve firmě *Eurosoftware*. Tento student neměl přesné znalosti vnitřního fungování jazyka, a proto volně zkoušel psát různé varianty DSL kódu a mohl tak vyhledat nedostatky, které se mohly objevit. Díky tomu také byla otestována požadována funkcionalita, která byla na začátku procesu definována - vyzkoušelo se generování a následné spuštění Java kódu, konkrétní mapování XML souborů, funkcionalita všech mapovacích příkazů a obecně jazykových konstrukcí.

[1] Junit5 InjectionExtension is broken in Xtext 2.24.0 · Issue #1648 · eclipse/xtext-core · GitHub. GitHub: Where the world builds software · GitHub [online]. Copyright © 2021 GitHub, Inc. Dostupné z: https://github.com/eclipse/xtext-core/issues/1648

8.3 Evaluace

Pro evaluaci byla oslovena skupina sedmi lidí z oboru IT. Tito lidé si vyzkoušeli napsat vlastní DSL program podle uživatelské dokumentace. V této dokumentaci se nachází úvodní informace o jazyku a dále následuje tutoriál rozdělený na 3 části. V každé části tutoriálu se uživatel naučí pracovat s jinými jazykovými konstrukcemi. Na konci dané části je úkol, který musí uživatel splnit. Po splnění všech úkolu je uživatel schopen vytvořit svůj vlastní DSL kód bez větších obtíží. Zde jsou vypsány části tutoriálu:

- Uživatel se v první části seznamuje s jazykem. Vytvoří si vlastní mapovací třídu, ve které si vyzkouší vytvořit proměnnou a přímé mapování. Na konci tutoriálu je úkol, ve kterém má uživatel přidat další přímá mapování.
- Uživatel se ve druhé části tutoriálu primárně setká s voláním uživatelských metod. Ve druhé části jsou také použity uživatelské rutiny, které ale uživatel při závěrečném úkolu nemodifikuje. Úkol je primárně zaměřen na uživ. metody a na přímé mapování.
- Uživatel ve třetí části pracuje primárně s uživatelskou rutinou a doplňuje do jejího těla mapovací příkazy. Tato část tutoriálu je krátká, ale obsahuje dlouhý samostatný úkol, ve kterém uživatel používá naučené jazykové konstrukce.

Pro zjištění, že DSL je uživatelsky přívětivý, byl vytvořen dotazník, který uživatelé vyplňují po zvládnutí všech úkolů. Dotazník se skládá z těchto částí a v příloze B lze najít souhrnný dotazník:

- obecné informace,
- efektivita (*effectiveness*),
- účinnost (*efficiency*),
- spokojenost (*satisfaction*),
- přístupnost (*accessibility*).

Důležité je pro tuto evaluaci zmínit, že výsledky XMappingu nejsou porovnávány s jiným softwarem. Altova MapForce je licencovaný software a bylo by obtížné shánět licenci pro každého uživatele. Proto jsem se rozhodl, že každé testování budu sledovat a závěrečný dotazník uživatelé vyplní na základě jejich subjektivních názorů. Z pozorování chování uživatelů při plnění

úkolů lze zjistit, jestli má uživatel problémy, nebo si naopak věří. I přesto, že v dotazníku jazyk není porovnáván s Altovou MapForce, lze vyčíst informace, které pomohou ohodnotit jazyk.

Každého uživatele jsem před začátkem testování seznámil s problematikou. Pokud se uživatel nikdy nesetkal s mapováním souborů, byl mu vysvětlen princip mapování na obrázcích ze softwaru Altova MapForce. Pokud měl uživatel nějaké otázky, mohl se na ně zeptat před tím než začal samostatně plnit úkol.

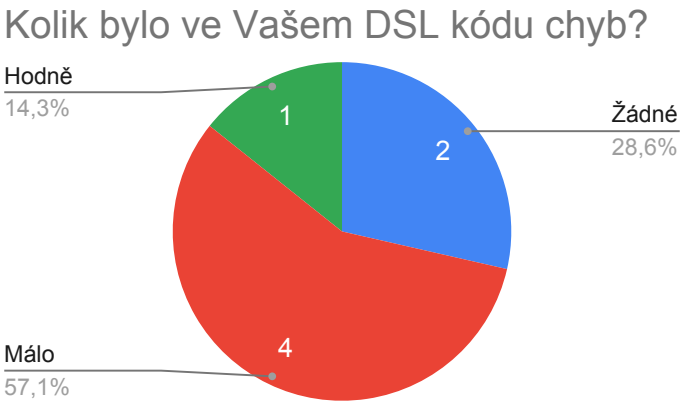
Z tabulky 8.1 vyplývá, že uživatelé mají odlišné zkušenosti s nástroji určenými k transformaci dat mezi soubory.

	Zkušenosti				
	Bez znalostí	Nováček	Průměrný	Zkušený	Expert
Altova MapForce	3	2	1	1	0
Jazyk XMapping	2	3	0	2	0

Tabulka 8.1: Tabulka počtu uživatelů se zkušenostmi s nástroji k transformování dat.

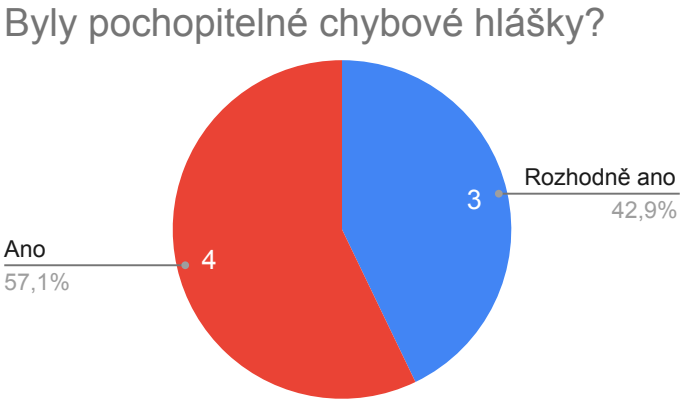
8.3.1 Efektivita

V této sekci se zabývám tím, zda měli uživatelé v kódu chyby. Z grafů lze vyčíst, že se 5 lidí ze 7 s chybami setkalo (obr. 8.1a), ale chybové hlášky jazyka byly srozumitelné (obr. 8.1c) a uživatelé dokázali chyby rychle opravit (obr. 8.1b).



(a) Rozložení odpovědí 7 respondentů na otázku: „Kolik bylo ve Vašem DSL kódu chyb?“.

(b) Rozložení odpovědí 7 respondentů na otázku: „Jak rychle jste chyby vyřešili?“.

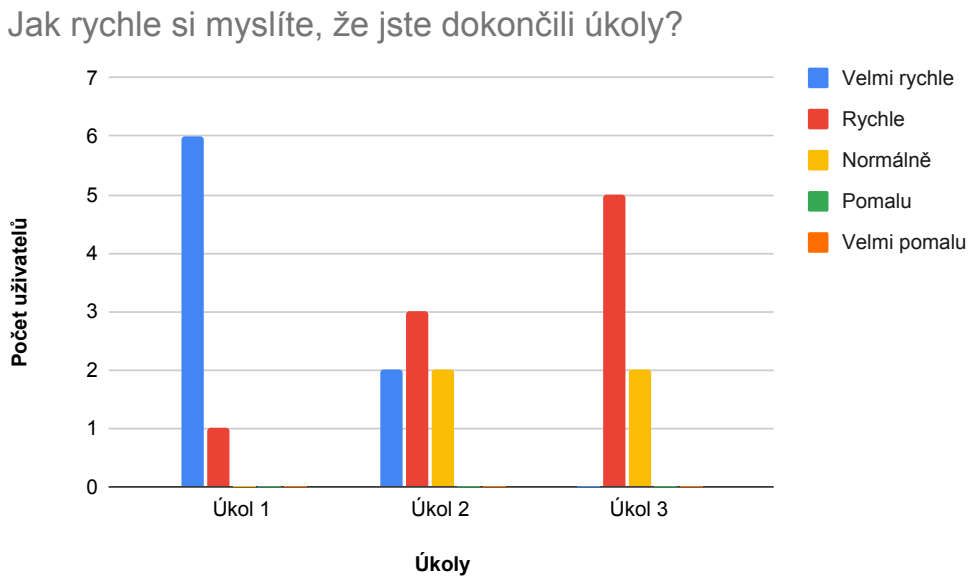


(c) Rozložení odpovědí 7 respondentů na otázku: „Byly pochopitelné chybové hlášky?“.

Obrázek 8.1: Grafy týkající se chyb v DSL jazyku.

8.3.2 Účinnost

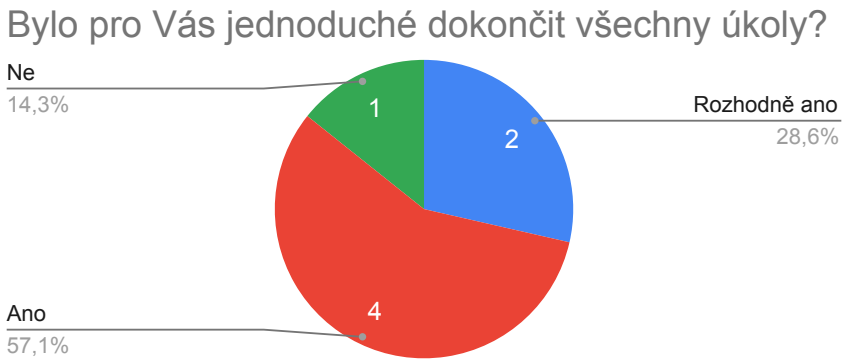
V této sekci jsem se zabýval, zda uživatelé mají pocit, že lze dosáhnout úspěšné transformace dat za krátkou dobu, nebo jestli jim přijde, že psaní DSL kódu je zdlouhavé. Odpovědi uživatelů jsou subjektivní, protože nemůžeme porovnat, jak dlouho by uživatelům trval vytvořit stejný úkol v Altova MapForce (obr. 8.2).



Obrázek 8.2: Odpovědi na otázku: „Jak rychle si myslíte, že jste dokončili úlohy?“.

Obtížnost každého úkolu se stupňuje a v grafu 8.2 lze vidět, že přibližně 85 % respondentů zvládne *velice rychle* vyřešit nejjednodušší úkol, ale u posledního úkolu už není žádná odpověď jako *velice rychle*. Nicméně 5 respondentů odpovědělo jako *rychle*.

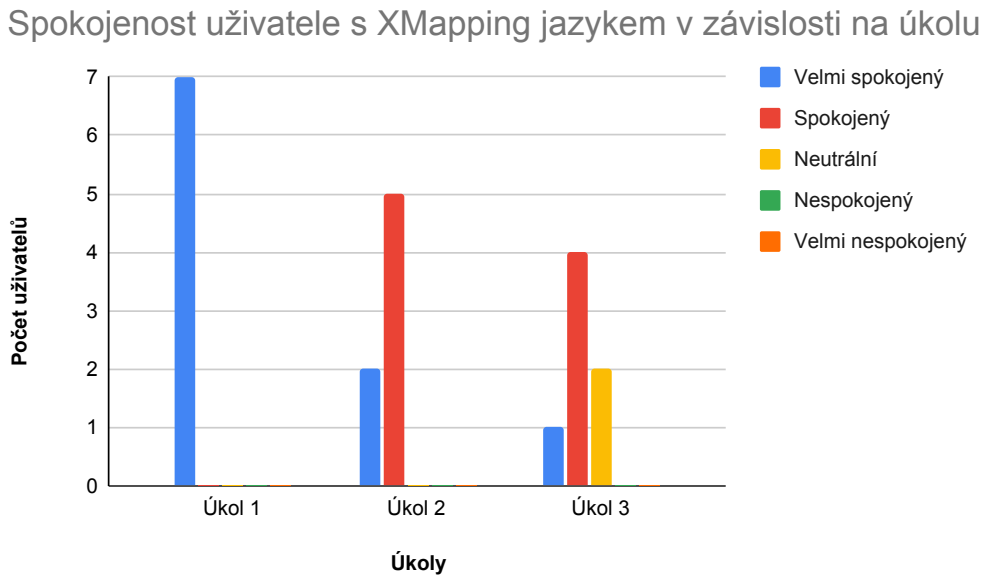
Respondenti dostali souhrnnou otázku, ve které zjišťuji, zda bylo nakonec jednoduché zvládnout všechny tři úkoly (obr. 8.3). 4 respondenti odpověděli *Ano*, 2 zaškrtli *Rozhodně ano*. Pouze jeden člověk odpověděl *Ne* s komentářem: „Nemám moc zkušeností s programováním, takže jsem měl zpočátku potíže s porozuměním jazyku“.



Obrázek 8.3: Rozložení odpovědí 7 respondentů na otázku: „Bylo pro Vás jednoduché dokončit všechny úkoly?“.

8.3.3 Spokojenost

V následující sekci se zabývám spokojeností uživatelů (obr. 8.4). Toto hodnocení je důležité v tom, že nejvíce vypovídá o tom, zda je jazyk uživatelsky přívětivý. Můžeme tím i zjistit, zda jsou jazykové konstrukce pro uživatele intuitivní.

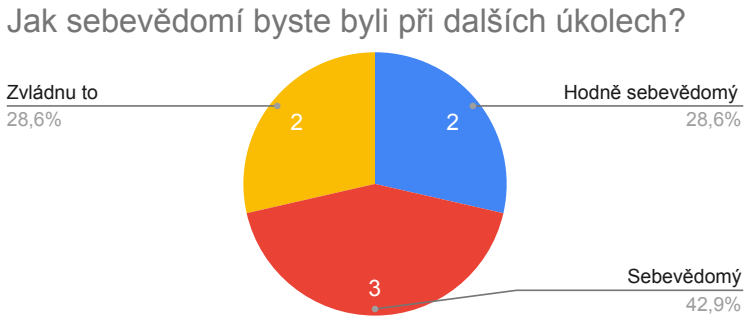


Obrázek 8.4: Graf spokojenosti respondentů s XMapping jazykem v závislosti na úkolu.

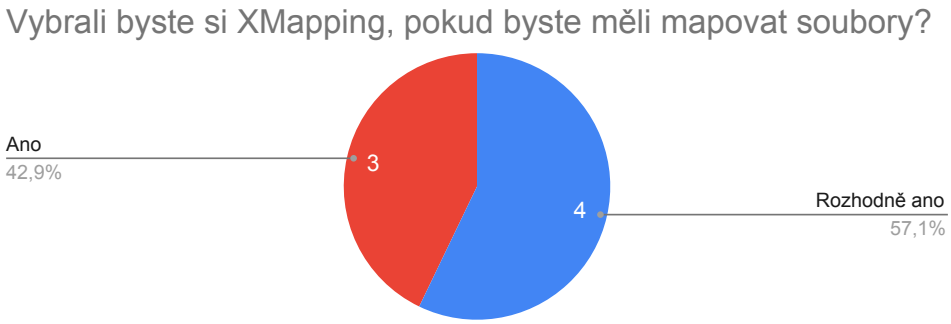
V grafu 8.4 lze vidět, že uživatelé nemají žádné problémy s prvním úkolem. Všech 7 respondentů odpovědělo možností *Velmi spokojený*. Ve druhém

úkolu se nejvíc objevuje *Spokojený* (5 krát). Ve třetím úkolu je 1 krát *Jde to*, ale opět je nejvíc odpovědí u možnosti *Spokojený*.

Dále jsem zkoumal, jak by si uživatelé věřili, kdyby dostali další úkol (obr. 8.5a). Na otázku „Doporučili byste XMapping svým přátelům, kteří by řešili mapování souborů?“ odpovědělo všech 7 lidí *Rozhodně ano*. Chtěl jsem zjistit také to, jestli by uživatel sám od sebe tento jazyk v budoucnu použil (obr. 8.5b).



(a) Rozložení odpovědí 7 respodentů na otázku: „Jak sebevědomí byste byli při dalších úkolech?“



(b) Rozložení odpovědí 7 respodentů na otázku: „Vybrali byste si XMapping, pokud byste měli mapovat soubory?“

Obrázek 8.5: Grafy týkající se spokojenosti uživatelů.

8.3.4 Přístupnost

U přístupnosti jsem se zajímal, jestli je jazyk pochopitelný. Všechny otázky ohledně přístupnosti jsou v tabulce 8.2.

	Odpovědi				
	Rozhodně ano	Ano	Nevím	Ne	Rozhodně ne
Lze jazyk lehce naučit?	3	4	0	0	0
Lze jednoduše pochopit jazykové konstrukce?	4	2	1	0	0
Je lehké vytvořit konkrétní mapování?	2	5	0	0	0

Tabulka 8.2: Tabulka počtu odpovědí uživatelů na otázky týkajících se přístupnosti.

8.4 Nasazení

Aby byla otestována funkcionalita jazyka, byl jazyk rovnou nainstalován na firemním počítači. Instalace XMapping jazyka probíhala standardně přes **Help → Install new software...**. S instalací na firemním počítači nebyl žádný problém, a tak šlo XMapping jazyk okamžitě použít.

K zaškolení uživatele byla vytvořena uživatelská příručka, která obsahuje úvodní informace o jazyku a dále návod na první DSL programy. Cílem je uživatele seznámit s jazykem tak, aby si vyzkoušel různé jazykové konstrukce a získal přehled o tom, co lze v jazyku použít. Po zvládnutí třetího úkolu je uživatel schopen napsat svůj vlastní DSL program.

Krok, který bude následovat, je přepis všech firemních mapování z grafického softwaru do DSL jazyka. Než přepisování začne, proběhne ohledně XMapping jazyka školení, ve kterém by měli uživatelé dostat obecný přehled o jazyku.

8.5 Možná vylepšení

Může se stát, že během přepisování mapování někteří uživatelé vymyslí zajímavé funkcionality, které by mohly být následně implementovány. Jazyk je připraven kdykoli k rozšíření. Je možné, že se budou potřeby uživatelů postupem času měnit a že se jim jazyk bude v nějaké míře přizpůsobovat.

9 Závěr

Cílem této práce bylo prozkoumat problematiku doménových specifických jazyků. Dále pak navrhnout a implementovat konkrétní doménově specifický jazyk, který by umožnil jednoduše napsat kód pro mapování XML souborů. Tento jazyk bude ve firmě *Eurosoftware* používán místo grafického softwaru Altova MapForce.

Na začátku bylo potřeba se seznámit s požadavky zadavatele a provést analýzu problémové domény. Vzniklo 7 doménových konceptů, na základě nich mohl následně vzniknout návrh jazyka. Tento návrh jazyka vznikl iterativně s postupně upřesňujícími požadavky na jazyk. Návrh je hlavně zaměřen na syntaxi a sémantiku. Bylo potřeba navrhnout vhodnou gramatiku. Vytvořená gramatika obsahuje 70 přepisovacích pravidel. Dále byl vytvořen typový systém, který aktuálně pozná 7 typů objektů, a omezení, které jsou použity při statické analýze.

Následovala implementace jazyka, do které spadalo vytvoření generátoru a interpreta. Průběžně s implementací byla vyvíjena knihovna, která obsahuje mapovací metody a dokáže transformovat XML soubory. Knihovna obsahuje 25 tříd a jeden *.properties* soubor, kterým lze konfigurovat zprávy v generované dokumentaci. Implementovaný generátor jazyka umožňuje vygenerovat Java kód s těmito metodami a tím je vytvořena spustitelná Java aplikace, která mapuje data mezi dvěma XML soubory stejně jako Altova MapForce.

V průběhu vývoje byla postupně testována gramatika. V tomto testování jsem zaměřil na mapovací příkazy a základní konstrukce jazyka, a proto vzniklo 16 Unit testů. V rámci testování byla také provedena evaluace, která je pro doménově specifický jazyk významná tím, že lze zjistit, zda je jazyk správně navržen a zda je pro uživatele intuitivní. Z výsledků je patrné, že jazyk je uživatelsky přívětivý.

Jazyk pomůže ve firmě vytvářet rychleji mapování XML souborů, protože nebude potřeba zaučovat uživatele na grafický software třetí strany. Dále bude možné změny v mapování dobře mergovat, protože v repozitáři bude textový soubor místo binárního souboru. Pokud bude zapotřebí, jazyk je kdykoliv připraven k rozšíření - mohou vzniknout uživatelské metody nebo i nové jazykové konstrukce.

Literatura

[1] Alpuente, M. – Barbuti, R. – Ramos, I. 1994 Joint Conference on Declarative Programming, GULP-PRODE'94 Peñiscola, Spain, September 19-22, 1994, Volume 2. 01 1994.

[2] *Altova MapForce 2021 Enterprise Edition User & Reference Manual* [online]. Altova GmbH, 2021. [cit. 2021/05/01]. Dostupné z: `https://www.altova.com/documents/MapForceEnt.pdf`.

[3] Alves, M. – Carreira, P. – Aguiar Costa, A. BIMSL: A generic approach to the integration of building information models with real-time sensor data. *Automation in Construction*. 09 2017, 84, s. 304–314. doi: 10.1016/j.autcon.2017.09.005.

[4] Bariic, A. – Amaral, V. – Goulão, M. Usability Evaluation of Domain-Specific Languages. 2012, s. 342–347. doi: 10.1109/QUATIC.2012.63.

[5] Bariic, A. – Amaral, V. – Goulão, M. Usability Evaluation of Domain-Specific Languages. 2012, s. 342–347. doi: 10.1109/QUATIC.2012.63.

[6] Barisic, A. *Usability Evaluation of Domain-Specific Languages*. PhD thesis, 12 2017.

[7] Bettini, L. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing, 2013. ISBN 1782160302.

[8] Bettini, L. Type errors for the IDE with Xtext and Xsemantics. *Open Computer Science*. 03 2019, 9, s. 52–79. doi: 10.1515/comp-2019-0003.

[9] Deursen, A. – Klint, P. – Visser, J. Domain-Specific Languages: An Annotated Bibliography. *SIGPLAN Notices*. 01 2000, 35, s. 26–36.

[10] Foundation, T. E. *Xtext Documentation* [online]. The Eclipse Foundation, 2014. [cit. 2021/05/01]. Dostupné z: `https://www.eclipse.org/Xtext/documentation/2.7.0/Xtext%20Documentation.pdf`.

[11] Fowler, M. *Domain Specific Languages*. Addison-Wesley Professional, 1st edition, 2010. ISBN 0321712943.

[12] Ghosh, D. *DSLs in Action*. Manning Publications Co., 1st edition, 2010. ISBN 9781935182450.

[13] Ghosh, D. DSL for the Uninitiated. *Commun. ACM*. July 2011, 54, 7, s. 44–50. ISSN 0001-0782. doi: 10.1145/1965724.1965740. Dostupné z: https://doi.org/10.1145/1965724.1965740.

[14] Hopcroft, J. E. – Motwani, R. – Ullman, J. D. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN 0321455363.

[15] Humm, B. – Engelschall, R. Language-Oriented Programming Via DSL Stacking. *ICSOFT 2010 - Proceedings of the 5th International Conference on Software and Data Technologies*. 01 2010, 2, s. 279–287.

[16] Kang, K. et al. Feature-Oriented Domain Analysis (FODA) feasibility study. 01 1990.

[17] Kosar, T. et al. Comparing General-Purpose and Domain-Specific Languages: An Empirical Study. *Computer Science and Information Systems*. 05 2010, 438. doi: 10.2298/CSIS1002247K.

[18] Mernik, M. – Heering, J. – Sloane, A. When and How to Develop Domain-Specific Languages. *ACM Comput. Surv.* 12 2005, 37, s. 316–. doi: 10.1145/1118890.1118892.

[19] Michaelson, G. Are there Domain Specific Languages? s. 1–3, 03 2016. doi: 10.1145/2889420.2892271.

[20] Méndez-Acuña, D. *Leveraging Software Product Lines Engineering in the Construction of Domain Specific Languages*. PhD thesis, 12 2016.

[21] Mooij, A. – Hooman, J. *Creating a Domain Specific Language (DSL) with Xtext* [online]. Radboud University, 2020. [cit. 2021/05/01]. Dostupné z: http://www.cs.ru.nl/J.Hooman/DSL/Creating_a_Domain_Specific_Language_(DSL)_with_Xtext.pdf.

[22] Nagy, B. Languages generated by context-free grammars extended by type AB -> BA rules. *Journal of Automata, Languages and Combinatorics*. 01 2009, 14, s. 175–186.

[23] Riti, P. *Practical Scala DSLs: Real-World Applications Using Domain Specific Languages*. Apress, 1st edition, 2017. ISBN 1484230353.

[24] S. Kolovos, D. et al. Requirements for Domain-Specific Languages. 05 2006.

[25] Sánchez Cuadrado, J. – Canovas Izquierdo, J. – Molina, J. Comparison between internal and external DSLs via RubyTL and Gra2MoL. *Formal and Practical Aspects of Domain-Specific Languages:*

Recent Developments. 09 2012, 2, s. 109–131. doi: 10.4018/978-1-4666-2092-6.ch005.

[26] Strembeck, M. – Zdun, U. An approach for the systematic development of domain-specific languages. *Softw., Pract. Exper.* 10 2009, 39, s. 1253–1292. doi: 10.1002/spe.936.

[27] Sun, Y. et al. Is My DSL a Modeling or Programming Language? 01 2008.

[28] Deursen, A. Domain-Specific Languages versus Object-Oriented Frameworks: A Financial Engineering Case Study. 1997.

[29] Van Roy, P. – Haridi, S. *Concepts, Techniques, and Models of Computer Programming*. The MIT Press, 1st edition, 2004. ISBN 0262220695.

[30] Visser, E. WebDSL: A Case Study in Domain-Specific Language Engineering. *Lecture Notes in Computer Science*. 01 2007, 5235/2008, s. 291–373. doi: 10.1007/978-3-540-88643-3_7.

[31] Voelter, M. *DSL Engineering: Designing, Implementing and Using Domain-specific Languages*. CreateSpace Independent Publishing Platform, 2013. Dostupné z: `https://books.google.it/books?id=J2i0lwEACAAJ`. ISBN 9781481218580.

[32] *Xtext - Language Engineering Made Easy!* [online]. The Eclipse Foundation, 2021. [cit. 2021/05/01]. Dostupné z: `https://www.eclipse.org/Xtext/`.

Seznam obrázků

Seznam tabulek

Seznam výpisů kódu

Seznam zkratek

DSL	Doménově specifický jazyk
GPL	Obecný jazyk
XML	Extensible Markup Language
BFN	Backus–Naurova forma
EBNF	Rozšířená Backus–Naurova forma
API	Application Programming Interface
AST	Abstraktní syntaktický strom
UI	Uživatelské rozhraní
ANTLR	ANother Tool for Language Recognition
EMF	Eclipse Modeling Framework

Přílohy

A Uživatelská dokumentace

V diplomové práci bude v příloze pouze začátek uživatelské dokumentace (6 stran). Celá dokumentace má dohromady 24 stran a obsahuje také tutoriál se třemi úkoly. K dispozici bude k v odevzdaném archivu diplomové práce:

`Aplikace_a_knihovny\XMapping_language_user_documentation.pdf`

XMapping language - user documentation

Introduction into XMapping language

1) What is XMapping language?

XMapping is a domain specific language which tries to replace software *Altova MapSource*.

What is Altova MapSource?

In general, *Altova MapSource* is used for mapping one XML file into another one. This program has a GUI and you must visually connect all needed transformations of XML elements. Sometimes it is hard to map because it is a very advanced program and the user has to be skilled.

2) When to use XMapping language?

If you do not have *Altova MapSource*, or do not want to use or learn this program, you can use another solution: XMapping. XMapping is a declarative programming language which provides the same base functionality as *Altova MapSource*.

Is there something to learn about the XMapping language?

If you want to use this language, you must learn language syntax. Syntax will be described in the tutorial.

Anything else?

This language generates Java code which will be executed to transform data from one XML into another. So, it is recommended to know something about the XMapping java library (when to use and how). But no worries, it will be explained in the tutorial.

Anything else?

xPath knowledge required (https://www.w3schools.com/xml/xpath_intro.asp).

3) Why should I use this XMapping language?

If you love programming, this language is yours!

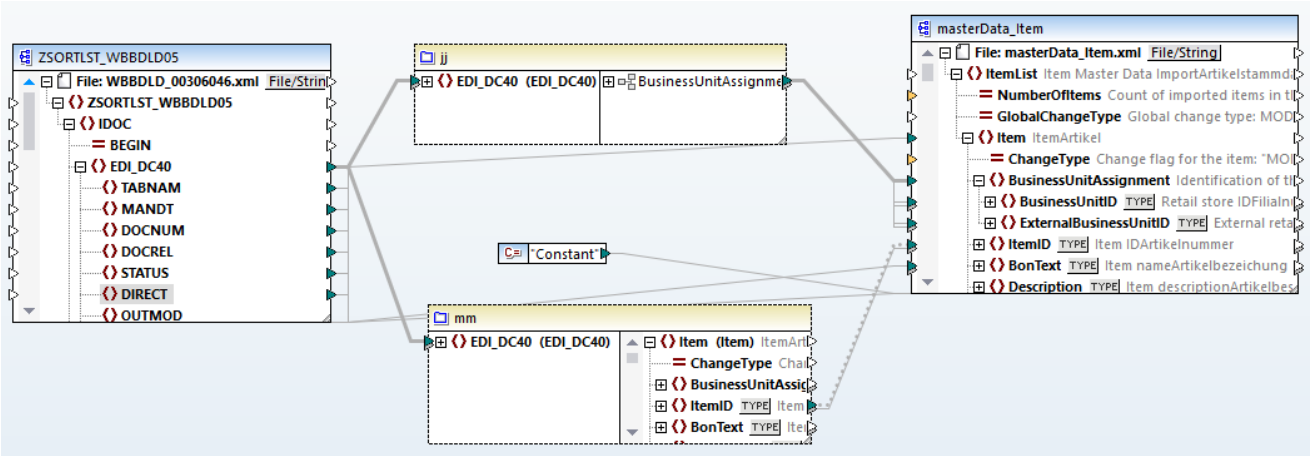
Before we start

Imagine that you have one **XML file** filled with the data and its **XSD file** *(XSD file describes structure of elements in XML).* And your job is actually to create a new XML file with data from the existing XML file. You got another the **XSD file.** This file defines how the structure of the output file will look. Simply, you have to map an XML file into another one.

You can choose *Altova MapSource* and create connections in advanced GUI. You will start with defining an input XML file, then defining connections (which transform data) and lastly you will choose an output file, where all data will be stored.

Description - you can see in the picture:

- *on the left side:* there is an **Input XSD file** where you can find the whole structure. It takes data from the Input XML file which is defined in properties of the XSD file cell.
- *on the right side:* there is an **Output XSD file** where input data are transformed.
- *in the middle:* there are **constants** + **macros (jj, mm)** which transform data too.



Result of this mapping (generated output XML file) should be :

GeneratedOutput.xml

```
<ItemList>
	<Item>
		<BusinessUnitAssignment>
			<BusinessUnitID>1</BusinessUnitID>
		</BusinessUnitAssignment>
		<ItemID>1</ItemID>
		<BonText>1</BonText>
		<BonText>1</BonText>
		<SupplierItemList>
			<SupplierItem>
				<SupplierID>INIT</SupplierID>
				<SupplierID>AEN</SupplierID>
				<SupplierItemID>Constant</SupplierItemID>
			</SupplierItem>
			<SupplierItem>
				<SupplierID>INIT</SupplierID>
				<SupplierItemID>Constant</SupplierItemID>
			</SupplierItem>
		</SupplierItemList>
	</Item>
	<Item>
		<BusinessUnitAssignment/>
		<BonText>1</BonText>
		<BonText>1</BonText>
		<SupplierItemList>
			<SupplierItem>
				<SupplierID>INIT</SupplierID>
				<SupplierID>AEN</SupplierID>
				<SupplierItemID>Constant</SupplierItemID>
			</SupplierItem>
			<SupplierItem>
				<SupplierID>INIT</SupplierID>
				<SupplierItemID>Constant</SupplierItemID>
			</SupplierItem>
		</SupplierItemList>
	</Item>
</ItemList>
```

XMapping project

XMapping project is primary Maven project - it uses maven 3.6.3.

New project: You can create a new one - everything you need is add specific *pom.xml* and create folder "*src-gen*" and set this folder as Source folder.

Existing project: it will contain:

- src (directory for all source *.xmapping* files),
- src-gen (directory for all generated *.Java* files),
- pom.xml (specific file which helps to add dependencies + to create executable jar).

Used pom.xml in project (copy this pom.xml if you created a new project)

- this pom.xml contains dependency on XMapping language library.
- **Make sure, that maven knows Xmapping library. If not then execute this query:**
  - replace <path_to_file\XmappingLib.jar> with real path and file.

```
mvn install:install-file -Dfile=<path_to_file/XMappingLib.jar> -DgroupId=com.gk_software.core.dsl.xmapping -
DartifactId=xmapping-lib -Dversion=1.0 -Dpackaging=jar
```

pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
        <modelVersion>4.0.0</modelVersion>
        <groupId>com.gk_software.core.dsl.xmapping</groupId>
        <artifactId>xmapping-tutorial</artifactId>
        <version>0.0.1-SNAPSHOT</version>
        <name>XMapping Tutorial</name>

        <build>
                <sourceDirectory>src-gen</sourceDirectory>
                <plugins>
                        <plugin>
                                <groupId>org.apache.maven.plugins</groupId>
                                <artifactId>maven-assembly-plugin</artifactId>
                                <configuration>
                                        <!-- <finalName>XMapping-executable</finalName> -->
                                        <descriptorRefs>
                                    <descriptorRef>jar-with-dependencies</descriptorRef>
                        </descriptorRefs>
                               <appendAssemblyId>false</appendAssemblyId>
                                        <archive>
                                                <manifest>
                                                        <mainClass>${class}</mainClass>
                                                </manifest>
                                        </archive>
                                </configuration>
                                <executions>
                                        <execution>
                                                <id>assemble-all</id>
                                                <phase>package</phase>
                                                <goals>
                                                        <goal>single</goal>
                                                </goals>
                                        </execution>
                                </executions>
                        </plugin>
                        <plugin>
                                <artifactId>maven-compiler-plugin</artifactId>
                                <version>3.8.0</version>
                                <configuration>
                                        <release>11</release>
                                        <includes>
                                                <include>**/*</include>
                                        </includes>
                                </configuration>
                        </plugin>
                </plugins>
        </build>

        <dependencies>
                <!-- xmapping-library -->
                <dependency>
                        <groupId>com.gk_software.core.dsl.xmapping</groupId>
                        <artifactId>xmapping-lib</artifactId>
                        <version>1.0</version>
                </dependency>

                <!-- user libraries --->

        </dependencies>


</project>
```

How to add user library

If you want too add own Java library, just add library into pom.xml

- **Make sure, that maven knows user library. If not then execute this query:**
  - replace <path_to_file\UserLib.jar> with real path and file.

```
mvn install:install-file -Dfile=<path_to_file/UserLib.jar> -DgroupId=com.gk_software.core.dsl.xmapping -
DartifactId=user-lib -Dversion=1.0 -Dpackaging=jar
```

pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
        ....
        <dependencies>
                <!-- xmapping-library -->
                ....

                <!-- user libraries -->
                <dependency>
                        <groupId>com.gk_software.core.dsl.xmapping</groupId>
                        <artifactId>user-lib</artifactId>
                        <version>1.0</version>
                </dependency>
        </dependencies>
</project>
```

How to run mapping

start command line on main project directory and write:

- replace <package.Class> with your main class qualified name.

```
mvn package -Dclass=<package.MainClass>
```

- go into directory "target" and there will be executable jar with name: *xmapping-tutorial-0.0.1-SNAPSHOT.jar.* To start mapping write *and e*xecute:
  - replace <path_to_file\input_xml_name> with real path and input file name.
  - replace <output_xml_name> with any output file name **(NAME without ".xml" !!)**
  - number of inputs/outputs must be same as number inputs/outputs defined in main xmapping file.
  - has to be executed with Java 11.

```
java -jar xmapping-tutorial-0.0.1-SNAPSHOT.jar "<path_to_file\input_xml_name>" "<output_xml_name>"
```

First steps in XMapping language

XMapping class template

1) If you have a project ready, we will start to write first commands into our ***mappingFile.xmapping*** file. You have to start with a package name. Package can have an arbitrary name.

mappingFile.xmapping

```
package src.altova;
```

2) *(Optional step)* You can define *imports* if you want to import another one *xmapping* file.

mappingFile.xmapping

```
package src.altova;

import src.altova.other.*;
import src.altova.next.MyOtherClass;
```

3) Define your XMapping class. You must write the keyword *MappingClass*.

mappingFile.xmapping

```
package src.altova;

import src.altova.other.*;
import src.altova.next.MyOtherClass;

MappingClass SAPToMasterData
```

4) Define your input files - we have input XML and XSD, but we use only XSD in your language. So input will be:

- Input:"<path_to_file>" (default version)
- Input.<name>:"<path_to_file>" (extended version)

<path_to_file> - file absolute path (recommended)
<name> - arbitrary name

5) Define your output files - we have only output XSD, so output will be:

- Output:"<path_to_file>" (default version)
- Output.<name>:"<path_to_file>" (extended version)

<path_to_file> - file absolute path (recommended) or relative path
<name> - arbitrary name

mappingFile.xmapping

```
package src.altova;

import src.altova.other.*;
import src.altova.next.MyOtherClass;

MappingClass SAPToMasterData
Input:"C:\Users\user\Desktop\ZSORTLST_WBBDLD05.xsd";                //or Input.myName:"..."
Output.first:"C:\Users\user\Desktop\masterData_Item.xsd";        //or Output:"..."
```

6) We will define the main block where you will write mapping commands. We start with: *StartMapping* and end with *EndMapping* - If you do that, you will have the main template of XMapping language:

mappingFile.xmapping

```
package src.altova;

import src.altova.other.*;
import src.altova.next.MyOtherClass;

MappingClass SAPToMasterData
Input:"C:\Users\user\Desktop\ZSORTLST_WBBDLD05.xsd";                //or Input.myName:"..."
Output.first:"C:\Users\user\Desktop\masterData_Item.xsd";        //or Output:"..."

StartMapping

EndMapping
```

B Souhrnný dotazník

1 Questionnaire - XMapping language (summary)

"++" = Yes, absolutelly / Totally (very positive meaning)
"+" = Yes / Very (positive meaning)
"−/+" = Neutral / Moderately (neutral)
"−" = No / Slightly (negative meaning)
"−−" = No, absolutelly / Not at all (very negative meaning)

General information	−−	−	−/+	+	++	Your comment
How would you rate your level of knowledge regarding to file mapping?	2	0	2	3	0	
Do you know Altova MapForce?	3	2	1	1	0	
Do you know XMapping language?	2	3	0	2	0	

1.1 Effectiveness

XMapping language	−−	−	−/+	+	++	Your comment
Did your code have small number of error/s?	0	1	0	4	2	
Did it take a short time to solve some problem/s?	0	0	0	3	4	
Is it easy to understand the meaning of error messages?	0	0	0	4	3	

1.2 Efficiency

XMapping language	−−	−	−/+	+	++	Your comment
Do you think that you quickly finished the Task 1?	0	0	0	1	6	
Do you think that you quickly finished the Task 2?	0	0	2	3	2	
Do you think that you quickly finished the Task 3?	0	0	2	5	0	
Was it easy to complete whole language tutorial?	0	1	0	4	2	

1.3 Satisfaction

XMapping language	−−	−	−/+	+	++	Your comment
What are your feelings about the Task 1?	0	0	0	0	7	
What are your feelings about the Task 2?	0	0	0	5	2	
What are your feelings about the Task 3?	0	0	2	4	1	
How confident would you be if you were given another task in XMapping language?	0	0	2	3	2	
Would you recommend XMapping language to other users?	0	0	0	0	7	
Would you choose this language if you had to map files?	0	0	0	3	4	

1.4 Accessibility

XMapping language	−−	−	−/+	+	++	Your comment
Is this language easy to learn?	0	0	0	4	3	
Is it easy to understand the meaning of language elements?	0	0	1	2	4	
Is easy to create the concrete mapping?	0	0	0	5	2	

Toto je souhrnný list odpovědí všech uživatelů. Jednotlivé dotazníky budou v odevzdaném archivu diplomové práce:

`Vysledky\dotazniky`.

C Struktura odevzdávaného archivu

- `Výsledky\dotazniky` (složka, ve které jsou všechny dotazníky)
- `Aplikace_a_knihovny`
  - `eclipse-java-2021` (spustitelný portable eclipse, kde je již nainstalován XMapping jazyk)
  - `knihovny` (obsahuje XMapping knihovnu a jednu uživatelskou knihovnu)
  - `plugin` (obsahuje plugin, který lze v Eclipse nainstalovat přes Help -> Install new software)
  - `projekty` (zkušební projekty)
    - `maven` (projekt, který využívá maven)
    - `without-maven` (pokud není maven na počítači, nebo není podporován, lze spustit verzi bez maven)
    - `README.txt`
  - `sources` (složka se zdrojovými kódy)
    - `XMapping-lib` (složka se zdrojovými kódy XMapping knihovny)
    - `User-lib` (složka se zdrojovými kódy uživ. knihovny)
    - `Xtext-project` (složka se zdrojovými kódy samotné implementace jazyka)
  - `start_script_maven.bat` (script vhodný při použití maven projektu)
  - `XMapping_language_user_documentation.pdf` (uživatelské dokumentace)
- `Poster`(složka s .pdf a .pub posterem)
- `Test_prace` (složka s diplomovou prací)
  - `jirmanj_DP` (složka se zdrojovými soubory)
  - `DP_Jan_Jirman_2021.pdf` (diplomová práce)
- `README.txt`