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

Bakalářská práce

Generování konfigurací softwarových komponent z modelů vlastností

Plzeň 2019 Vaněk Jakub

Místo této strany bude zadání práce.

Prohlášení

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů. V knize jsou použity názvy programových produktů firem apod, které mohou být ochrannými známkami nebo registrovanými ochrannými známkami příslušných vlastníků.

V Plzni dne 21. června 2019

Vaněk Jakub

Abstract

Configuration of Software Components Generator from Feature Models. Goal of this thesis is to generate feature model from grammar of tesa language written in Xtext and to generate final source code from chosen features of this model. This thesis describes and uses knowledge about feature modeling and generative programming. In the opening part of the thesis reader is introduced with feature modeling, tools used for feature modeling, generative programming and Xtext framework. In the later part implementation design and final solution is described.

Abstrakt

Cílem této práce bude vygenerovat šablonu jako model vlastností ze zápisu gramatiky jazyka tesa pomocí frameworku Xtext a na základě vybraných vlastností z tohoto modelu vygenerovat finální zdrojový kód. V práci jsou popsány a využity znalosti o modelování vlastností a generativním programování. Čtenář je v úvodní části seznámen s problematikou modelování vlastností, nástroji které jsou pro modelování, nebo tvorbu modelů využívány. Dále bude seznámen s problematikou generativního programování a frameworkem Xtext. V pozdějších částech bude vysvětlen návrh implementace a finální řešení problému.

Obsah

1 Úvod

Bakalářská práce byla zadána společností ZF. Společnost ZF je německou automotive společností. Účelem bakalářské práce je ušetřit čas vývojářům při tvorbě konfigurátoru jeho generováním.

Generování softwarových komponent bude využívat generativního programování. Generativní programování je způsob programování, kdy je zdrojový kód programu generován na základě šablony. Tuto šablonu tvoří různé vzory.

K vytvoření šablony, pro vygenerování finálního zdrojového kódu bude v práci využíván model vlastností. Model vlastností zachycuje všechny různorodosti a podobnosti všech možných variant finálního produktu. Různé charakteristiky, které finální produkty rozlišují se nazývají právě vlastnosti. Vlastnosti jsou základními stavebními kameny modelu vlastností, který zobrazuje závislosti mezi nimi.

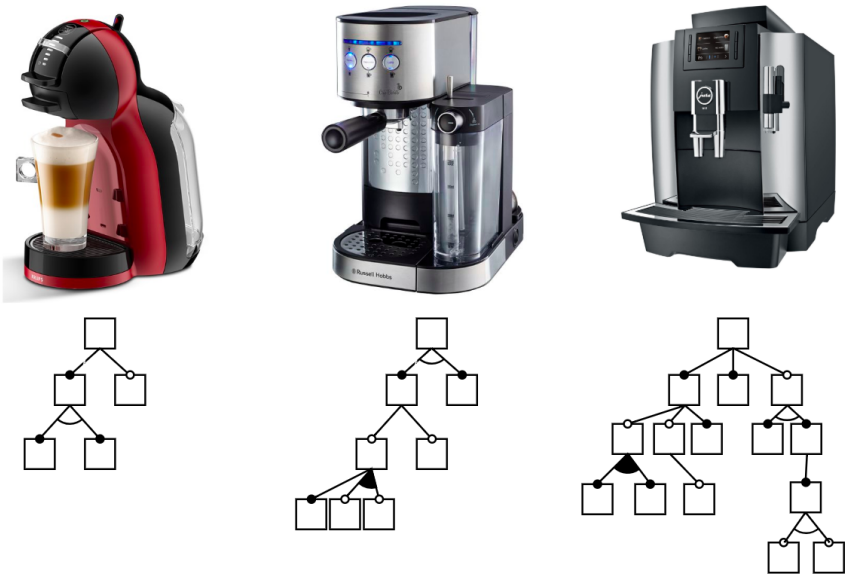
V úvodní části se tato práce zabývá analýzou problematiky modelů vlastností, nástroji které se pro modelování vlastností využívají, generativního programování a Xtextu. Následně bude popsán návrh implementace, popis samotné implementace a na závěr zhodnocení dosažených výsledků.

2 Modelování vlastností

Účelem této kapitoly je seznámit čtenáře s problematikou *modelování vlastností* (z angl. *feature modeling*), které je v práci použito jako šablona pro generování zdrojového kódu konfigurátoru.

2.1 Motivace

Motivaci pro modelování vlastností si ukážeme na příkladě. Představme si, že společnost vyvíjí kávovar. Kávovar bude vždy připravovat kávu, avšak jeho vlastnosti se mohou lišit na základě předem určených specifikací. Součástí specifikace může být požadavek, že kávovar bude vyvíjen pro různé trhy, například pro Evropský trh a trh v USA. Dále může být vyžadováno vytvoření dvou různých edic kávovaru, pro příklad standartní edici a deluxe edici, kde deluxe edice bude oproti standartní edici obsahovat trysku na čistou horkou vodu a displej. Za těchto předpokladů je třeba si uvědomit, že kávovar pro Americký a Evropský trh bude využívat jiný adaptér. Pro Evropský trh je potřeba klasických 220V a pro USA 120V. Zároveň je u kávovaru možno si zvolit, zda bude nebo nebude mít nastavitelné množšství kávových zrn a množství vody, ze které bude káva připravena.



Obrázek 2.1: příklad různých složitostí diagramů

Kávovar se v tomto případě nazývá *produktovou řadou* (z angl. *product*

line), *produktovou rodinou* (z angl. *product family*), nebo také *konceptem* (z angl. *concept*). Produktová rodina, či koncept, jsou pojmy, které označují skupinu produktů, které fungují na stejném principu, ale liší se od sebe různými vlastnostmi, tudíž je vytvořeno několik variant. Varianty je třeba spravovat. Je potřeba znázornit, které vlastnosti bude jaká varianta obsahovat, jak na sobě vlastnosti závisí, které a zda jsou potřeba. K tomu nám slouží právě *model vlastností* (z angl. *feature model*).

Model vlastností si také můžeme ukázat na softwarových produktech. Příkladem rodiny produktů může být produkt společnosti Microsoft. Produktovou rodinou je zde například operační systém Windows 10. Windows 10 je vydáván v několika edicích. Těmito edicemi jsou *Home*, *Pro*, a *Enterprise*. Všechny tyto edice jsou operačním systémem Windows 10, avšak liší se v několika vlastnostech. V tabulce 2.1 je vidět výčet několika vlastností, ve kterých se operační systémy liší.

Windows	**Home**	**Pro**	**Enterprise**
Max. RAM	4GB 32bit 128GB x86-64	4GB 32bit 2T x86-64	4GB 32bit 2T x86-64
Microsoft Edge	Ano	Ano	Ano
Windows To Go	Ne	Ano	Ano
DirectAccess	Ne	Ne	Ano
AppLocker	Ne	Ne	Ano

Tabulka 2.1: Srovnání edicí operačního systému Windows 10

Tento model je tvořen procesem, který nazýváme modelování vlastností. Vytvořit takovýto model má hned několik výhod. Základní výhodou je přehlednost. Pokud zkonstruujeme správný model vlastností, je v něm na první pohled vidět, jaké produkty můžeme z vlastností sestavit. Model se poté dá použít pro nové verze stejného produktu tím, že se některé vlastnosti změní, nebo také jednoduše přidají.

Hlavní motivací je tedy identifikace a zachycení variability, znovupoužitelnost a rozšířitelnost systému či produktu.

2.2 Řada softwarových produktů

Řada softwarových produktů (z angl. *Software product line*) je v softwarovém inženýrství pojem, který označuje kolekci podobných softwarových systémů s podobným zaměřením. Klade důraz na podobnosti mezi softwarovými produkty. Během tohoto procesu je vytvořen koncept, kde lehké odlišnosti

vytvoří sadu konkrétních produktů. Nejedná se ale pouze o recyklaci využitelných částí jednoho z hotových produktů, ale o strategické vytvoření základních stavebních kamenů tak, aby byli jednodušše rozšiřitelné a neměnné.

2.3 Model rodiny produktů

Model rodiny produktů (z angl. *Family model*) je podle [10] model, který popisuje jak budou výsledné produkty skupiny produktů sestaveny nebo generovány ze specifikovaných částí. Každá komponenta modelu rodiny produktů reprezentuje jeden nebo více funkčních prvků produktu z řady produktů.

2.4 Vlastnosti

Každý koncept má určité *vlastnosti* (z angl. *features*). Pojem vlastnosti můžeme aplikovat jak na vyráběné produkty, tak na různé vlastnosti systému. Příkladem vlastností u vyráběných produktů mohou být vlastnosti znázorněné v příkladu s kávovarem. Vlastnosti jsou také užitečné při vyvíjení softwarových produktů. Požadavkem může být například kompatibilita s různými operačními systémy. Ve výsledném systému bude mít pak každá vlastnost odlišnou implementaci. Na základě těchto vlastností je možné navrhnout odlišnou implementaci pro jednotlivé varianty. Vlastnosti nám tedy zajišťují variabilitu mezi odlišnými produkty se stejným základem.

2.4.1 Model vlastností

Model vlastností znázorňuje závislosti mezi vlastnostmi. Vlastnosti v modelu tvoří strom, kde kořenovou vlastností je koncept. Koncept obsahuje další vlastnosti jako své potomky.

2.4.2 Model variant

Model variant (z angl. *Variant model*) je podle [10] model, který z vybraných vlastností modelu vlastností tvoří finální produkt. Jsou v něm zachycené pouze výsledně použité vlastnosti.

2.4.3 Grafické znázornění

Modely vlastností jsou zobrazovány v *diagramu vlastností* (z angl. *feature diagram*). Diagram vlastností je zakreslován jako stromový graf, kde koncept

je kořenovým uzlem a všechny další uzly jsou jeho vlastnostmi. Každý uzel může mít N potomků.

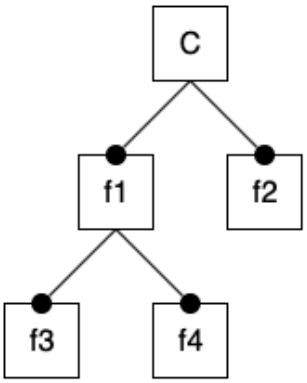
2.4.4 Typy vlastností

Jak už bylo řečeno, každý produkt u kterého je požadavek na určitou variabilitu, znovupoužitelnost či rozšiřitelnost obsahuje vlastnosti. Tyto vlastnosti mohou být několika typů. V této části se budeme věnovat základním typům těchto vlastností, tak jak jsou popsány v [3].

Povinné vlastnosti

Povinné vlastnosti (z angl. *Mandatory features*) jsou vlastnosti, které výsledný produkt musí obsahovat. Jsou to vlastnosti, na kterých je koncept založen a které jsou zahrnuty v jeho popisu. Povinná vlastnost musí být ve výsledném modelu zahrnuta, pokud je zahrnutý i její rodič. Například náš kávovar bude mít vždy mlýnek na kávu, odkapávač, zásobník zbytků, zásobník vody a další. Bez těchto vlastností se neobejde žádná varianta výsledného produktu. Tyto vlastnosti mají význam hlavně v případě, že jejich rodičovská vlastnost povinná není a nemusí být v modelu zahrnuta. Pokud je tedy zahrnut rodič, je nutné zahrnout i tuto vlastnost.

Povinnou vlastnost v diagramu značíme jednoduchou hranou zakončenou vybarveným kruhem.

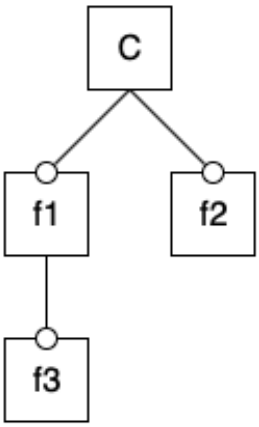


Obrázek 2.2: značení povinných vlastností

Každá instance konceptu C má vlastnost *f1* a *f2* a každá co má *f1* má *f3* a *f4*. Z toho vyplývá, že každá instance konceptu C má vlastnosti *f3* a *f4*. Můžeme tedy říct, že koncept C je popsán sadou vlastností:
{C, f1, f2, f3, f4}.

Volitelné vlastnosti

Volitelné vlastnosti (z angl. *Optional features*) mohou být zahrnuty v popisu konceptu. Jinými slovy, pokud je zahrnut rodič, volitelná vlastnost může a nemusí být zahrnuta. Pokud rodič volitelné vlastnosti zahrnutý není, nemůže být zahrnuta ani volitelná vlastnost na něm závislá. V příkladě s kávovarem může být touto vlastností například zmíněné nastavitelné množství kávových zrn a množství vody. Tuto vlastnost mít kávovar může, ale zároveň nemusí a tvoří tedy další varianty produktu. Volitelné vlastnosti v diagramu značíme jednoduchou hranou zakončenou prázdným kruhem.



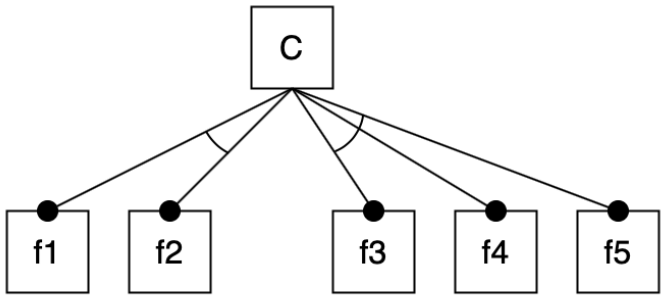
Obrázek 2.3: značení volitelných vlastností

Každá instance konceptu může mít vlastnost *f1*, vlastnost *f2*, obě, nebo žádnou. Pokud má vlastnost *f1*, může mít také vlastnosti *f3* a *f4*. Koncept můžeme popsat sadou vlastností:

- *{C}*
- *{C, f1}*
- *{C, f2}*
- *{C, f1, f2}*
- *{C, f1, f3}*
- *{C, f1, f2, f3}*

Alternativní vlastnosti

Alternativní vlastnosti (z angl. *Alternative features*) jsou vlastnosti, kde existuje možnost výběru mezi více vlastnostmi. V kávovaru je takovou vlastností edice, kdy je potřeba si vybrat mezi standart nebo deluxe edicí, ale výsledný produkt nemůže obsahovat obě. Alternativní vlastnosti mohou být volitelné, nebo povinné. Pokud je soubor alternativních vlastností povinný, je třeba vybrat právě jednu z nich. Pokud jsou alternativní vlastnosti volitelné, je třeba vybrat nejvýše jednu z nich. Alternativní vlastnosti v diagramu značíme prázdným obloukem mezi hranami.



Obrázek 2.4: značení alternativních vlastností

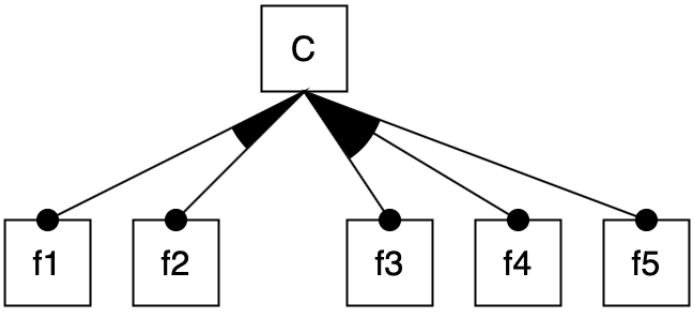
V instanci jsou znázorněný povinné alternativní vlastnosti. Musíme si tedy v levé větvi vybrat mezi vlastnostmi *f1* a *f2* a na pravé větvi mezi vlastnostmi *f3*, *f4* a *f5*. Koncept můžeme popsat sadou vlastností:

- *{C, f1, f3}*
- *{C, f1, f4}*
- *{C, f1, f5}*
- *{C, f2, f3}*
- *{C, f2, f4}*
- *{C, f2, f5}*

Slučitelné vlastnosti

Slučitelné vlastnosti (z angl. *Or-features*) (návrh překladu dle [9]) jsou vlastnosti, kde stejně jako u alternativních vlasntostí existuje možnost výběru. Oproti alternativním vlastnostnem ale znázorňují situaci, kdy je možnost výběru více než jedné vlastnosti. Mohou být opět volitené nebo povinné.

Pokud je slučitelná možnost volitelná, není třeba vybrat žádnout z nich. Pokud je povinná, je třeba vybrat alespoň jednu z nich. Slučitelné vlastnosti v diagramu značíme plným obloukem mezi hranami.



Obrázek 2.5: značení slučitelných vlastností

Dalši typy vlastností

Model vlastností může nabývat velké složitosti. Proto je nutné zavést další typy vlastností a rozšíření původního modelu vlastností. Jedním z těchto rozšíření je zavedení pojmu *kardinalita*. Kardinality označují počet kopií jednotlivé vlasnosti. Počet těchto vlastností v diagramu označujeme jako interval ve tvaru $[m...n]$, kde m je dolní a n je horní hranice počtu výskytů těchto vlastností. Tato notace se poprvé vyskytla v publikaci [11], kde se autoři inspirovali UML diagramy.

Autoři [11] také nahrazují alternativní a slučitelné vlastnosti pojmem *skupina vlastností* (z angl. *feature group*). Kardinalita zde označuje kolik vlastností z dané skupiny je možné použít. Alternativní vlastnosti zde nazýváme *XOR skupinou* (z angl. *XOR group*) a slučitelné vlastnosti *OR skupinou* (z angl. *OR group*).

Diagram vlastností dokáže efektivně zachytit veškeré vlastnosti a závislosti mezi nimi. Problém však může nastat v případě, kdy nějaká vlastnost vyžaduje jinou vlastnost, která není jejím přímým rodičem. Pro takové případy je zde zaveden pojem *omezení* (z angl. *constraints*). Tato omezení si můžeme představit u požadavku, že kávovar bude mít displej pouze v případě, že velikost nádrže na vodu bude více než půl litru. Tyto dvě vlastnosti spolu v diagramu nijak nesouvisí a využívají tedy omezení. Omezení mohou být dvojího typu. Podle autorů [8] to jsou *lokální omezení* (z angl. *local constraints*) a *globální omezení* (z angl. *global constraints*). Pojem lokální omezení je používán, pokud se toto omezení vztahuje pouze ke společnému rodiči těchto vlastností. Pojem globální omezení je používán pokud se tato

omezení vztahují na vlastnosti napříč diagramem. Tato omezení se budou značit značkou *requires*, pokud vlastnost vyžaduje jinou vlastnost napříč diagramem a značkou *excludes*, pokud tato vlastnost nějakou jinou vlastnost vylučuje.

Shrnutí

Pomocí těchto typů vlastností dokážeme z libovolného konceptu sestavit kompletní model vlastností. Díky němu jsme schopni zachytit veškeré varianty finálního produktu. Jsme schopni zjistit co jaká varianta vyžaduje a co musí obsahovat.

Diagram by krom názvů vlastností a typu závislostí měl obsahovat i informace o vlastnostech. Tyto informace by měly obsahovat důvod, proč je tato vlastnost vyžadována, jak souvisí se zbytkem modelu a veškeré další informace, které mohou být při vývoji užitečné.

3 Modelovací nástroje

V této kapitole se budeme zabývat nástroji, které umožňují modelování vlastností a které by bylo vhodné použít pro účely práce. Cílem je ukázat, jaké modelovací nástroje existují, k čemu se používají a zachytit rozsah jejich funkcí. Bude vysvětleno, jaké mají nástroje výhody a nevýhody a odůvodnění výběru nástroje.

3.1 pure::variants

Pure::variants je jeden z mála komerčně využívaných modelovacích nástrojů od společnosti pure-systems. Není zaměřen pouze na modelování vlastností, ale svou funkcionalitou se snaží pokrýt všechny fáze vývoje software. Samotný software je plug-inem do vývojového prostředí Eclipse. Umožňuje práci s několika modely a pro každý z těchto modelů má vlastní editor.

Hlavním modelem je *Feature Model* nebo-li model vlastností. Software zobrazuje model vastností ve stromové architektuře. Umožňuje vytvoření čtyř různých závislostí: povinné, volitelné, alternativní a slučitelné. Umožňuje také do modelu zanést pravidla o omezeních, která jsou nezbytná při výsledné konfiguraci.

Dalšími modely jsou *Family model* nebo-li model rodiny produktů. Tento model zobrazuje elementy této rodiny a dokáže na ně namapovat vlastnosti. Dále *Variant description model*, který vlastnosti dokáže konfigurovat. Posledním modelem je *Variant result model*, který narozdíl od předchozích jako jediný nemá vlastní editor. Tento model popisuje konkrétní výstupní variantu produktu, její popis a informace pro její sestavení.

V našem případě využijeme feature model jako zobrazení závislostí mezi částmi konfigurace. Na základě tohoto feature modelu budeme schopni vytvořit libovolné množství variant description modelů, ze kterých bude generován výsledný kód.

3.2 XFeature

Nástroj *XFeature* je dalším plug-inem do vývojového prostředí Eclipse, který poskytuje grafické uživatelské rozhraní pro práci s modely vlastností. Modely vlastností zde vyjadřují model rodiny produktů a modely aplikací. K modelu vlastností přistupují jako k meta-modelu vlastností. Model vlastností i jeho

konfigurace jsou zde popsány pomocí XML dokumentu, který odpovídá určitému XML schématu (meta-modelu). Uživatel je schopen vytvořit vlastní XML schéma, které však musí odpovídat jeho meta-modelu. Nové vlastnosti jdou tedy tvořit pouze v souladu s meta-modelem.

Tvorba modelů vlastností je zde prováděna pomocí kontextového menu. Umožňuje tvorbu povinných, volitelných i alternativních vlastností. Dle autorů [8] nástroj také umožňuje zavedení globálních omezení. Autoři také zavádí tzv. *podmíněná omezení* (z angl. *Conditional Constraints*), skrz které je možné zavést různé podmínky a na jejich základě aplikovat tato omezení.

Během tvorby nebo editace přes uživatelské rozhraní je model současně kontrolován a uživateli jsou nabízeny všechny validní rozšíření stávajícího modelu, což vede k přehlednému a rychlému rozšiřování modelu.

Autoři [8] sami uvádějí několik problému s tímto nástrojem. Jedním z nich je manipulace s velkými komplexními modely, která se může stát nepřehlednou. Dalším problémem, který uvádějí, je absence uložení stávající relace. Pokud chce uživatel relaci přerušit a pokračovat jindy, je nutné využít tzv. *model pro uložení* (z angl. *save model*).

3.3 Feature Modeling Plug-in

Nástroj *Feature Modeling Plug-in* (FMP) je také zásuvným modulem do vývojového prostředí Eclipse. Umožňuje vytváření, editaci i výslednou konfiguraci vlastností s kardinalitou a atributy podle [5]. Tento nástroj se již dále nevyvíjí.

Dle [6] nástroj umožňuje pomocí kontextového menu klonovat vlastnosti, jejichž horní hranice kardinality je větší než jedna. Autoři [1] také popisují schopnosti nástroje vyplnit omezení mezi vlastnostmi.

Jako ve většině nástrojů je model vlastností zobrazován ve stromové struktuře. Velkou výhodou je možnost rozdělit celý model do menších celků, na které se dá odkazovat a celý model tím zpřehlednit. Dle [1] je ale možnost sbalení a rozbalování jednotlivých skupin vlastností natolik užitečná a přehledná, že referencování jiných modelů není potřeba. Názvy featur je také možné měnit přímo ve stromě a není nutné otevírat kontextové menu. Po grafické stránce se velmi podobá nástroji pure::variants. Ovládání probíhá skrz kontextové menu a je intuitivní a přehledné. Nástroj se dá také ovládat pouze klávesnicí.

Nástroj neumožňuje vytváření samostatných modelů pro výslednou konfiguraci. Tato konfigurace probíhá na stejném stromě jako editace modelu vlastností pomocí úprav jeho částí, což může celý model znepřehlednit.

3.4 Software Product Lines Online Tools

Nástroj *Software Product Lines Online Tools* (SPLOT) je nástroj implementovaný jako webová aplikace. Aplikace je zdarma a volně k použití. Umožňuje vytvoření a editaci modelu vlastností pomocí grafického rozhraní. Lze zde přidávat povinné, volitelné vlastnosti i OR a XOR skupiny. Editor také umožňuje vytvoření globálních omezení. Model vlastností je ukládán do databáze a je možné ho sdílet s ostatními uživateli. Konfiguraci následně umí exportovat do souboru formátu CSV nebo XML.

Tvorba konfigurace z modelu vlastností je tvořena v samostatném okně. V tomto okně je vidět strom modelů vlastností a konfigurace jde tvořit dvěma způsoby. Jedním způsobem je ze stromu vybírat vlastnosti, které chceme ve výsledné konfiguraci, pomocí klikání. Druhým způsobem je nechat si automaticky vyplnit celou konfiguraci ze všech možných vlastností a poté vlastnosti odebírat. Nástroj umí během tvorby konfigurace hlídat dodržení pravidel, včetně omezení, a dokáže konfiguraci opravovat na základě vytvořeného modelu vlastností.

Zdrojové kódy nástroje jsou již od jeho vytvoření volně dostupné na GitHubu, kde jej můžou vývojáři volně zkoumat a vylepšovat. Nástroj také umožňuje stahovat a nahlížet do jiných modelů vlastností, které vytvořili ostatní uživatelé. Tyto modely se nacházejí v repozitáři, který obsahuje stovky různých modelů, které uživatelé vytvořili.

3.5 Vlastní nástroj

Další možností využití nástrojů pro modelování vlastností je vytvořit nástroj vlastní. Jelikož je zadání velmi specifické, bylo by možné vytvořit vlastní nástroj, který bude umět pracovat se specifickými daty. Požadavky na takový nástroj by byly:

- graficky zobrazit model vlastností z gramatiky psané v Xtextu
- umožnit vytvoření modelu variant z vytvořeného modelu vlastností
- validace vytvořené varianty na základě typů vlastností včetně globálních omezení
- export a import modelů variant, kvůli sdílení mezi uživateli
- vygenerování šablony v jazyce tesa

Nástroj by nemusel umět editaci modelu vlastností, jelikož by měl pouze zobrazovat závislosti zavedené v gramatice.

Odhad času vývoje

Vývoj takového nástroje musí projít všemi fázemi vývoje software. Těmito fázemi jsou analýza, návrh implementace, implementace, testování nástroje a validace všech předešlých fází. Během vývoje by docházelo k několika iteracím, kde by se na základě problémů v pozdějších fázích muselo vracet k dřívějším fázím a modifikovat je tak, aby výsledný nástroj souhlasil se všemi požadavky.

Fáze	**Část**	**Odhadovaný čas [hod]**
Analýza	Specifikace	16
	Model vlastností	40
	TesaTK	24
	Xtext	40
Návrh Implemetnace	Jádro	40
	GUI	16
	Parser	16
Implementace	Struktury modelu vlastností	40
	Závistlosti vlastností	120
	GUI	100
	navázání GUI na struktury	60
	Parser Xtext – nástroj	120
	Generátor nástroj – TesaTK	120
	Import/Export	100
Testování	Jednotkové testy	120
	Závistlosti vlastností	24
	GUI	24
	Parser Xtext – nástroj	40
	Generátor nástroj – TesaTK	40
	Celkem:	**1100**

Tabulka 3.1: Odhad času vývoje vlastního nástroje s jeho fázemi

Analýza by zahrnovala sběr požadavků od koncových uživatelů, jejich vyhodnocení a seznámení s technologiemi potřebnými pro vývoj, jemiž jsou gramatika psaná v Xtextu, seznámení s konfigurátorem TesaTK, seznámení se s technologií modelování vlastností a její konfigurace. Dále by bylo třeba zvážit, jaký programovací jazyk by byl pro vývoj nejvhodnější a výběr odůvodnit. Odhadovaný čas analýzy by v takovém případě byl 120 hodin, tedy 15 pracovních dní.

Během návrhu implementace by bylo potřeba navrhnout parser, který dokáže z gramatiky v Xtextu dynamicky tvořit model vlastností, tedy namapovat typy vlastností na syntaxi jazyka Xtext. Dále by bylo potřeba vybrat vhodné prostředky pro jejich zobrazení, což úzce souvisí s výběrem programovacího jazyka a frameworku, který by se využil. Na základě objektové analýzy by bylo potřeba navrhnout strukturu aplikace. Odhadovaný čas návrhu implementace by byl 72 hodin, tedy 9 pracovních dní.

Implementace by zahrnovala vytvořit parser z gramatiky, který by importoval model vlastností do nástroje, celé uživatelské prostředí včetně zobrazení modelu vlastností, generátor konfigurace variant jako šablon v jazyce Tesa a export modelů variant. Dále by bylo třeba vypořádat se se všemi problémy, které by během implementace mohli nastat. Odhadovaný čas implemetace je 660 hodin, tedy 83 pracovních dní.

Během testování by bylo třeba napsat jednotkové testy a otestovat tak celou aplikaci a opravit veškeré chyby, které by se při implementaci mohli objevit. Odhadovaný čas testování je 248 hodin, tedy 31 pracovních dní.

Pokud by všechny předešlé fáze dopadly úspěšně, proběhla by validace všech fází a nástroj by se mohl začít používat. Celkový odhadovaný čas strávený vývojem této aplikace by tak byl 1100 hodin, tedy 138 pracovních dní. Pokud odhadneme cenu jedné programátorské hodiny na 1000kč, dostaneme odhadovanou cenu nástroje, která by činila 1,1 milion korun. Důležité je vzít v potaz, že odhadovaný čas se může lišit o více jak 100% času stráveného na vývoji. Při vývoji by bylo vytvořeno několik prototypů, na které by se nabalovaly další funkce. Kvůli možným problémům by se vývoj mohl nátáhnout i na několikanásobek odhadovaného času.

Takový nástroj je proprietární, což je jeho největší výhoda. Při jeho používání nemůže nastat situace, že by přišla nová verze, která již nebude splňovat specifické požadavky na interní nástroj. Dalšími výhodami může být jeho specifická funkčnost, která bude splňovat přesné požadavky.

3.6 Výběr

V této části bude zhodnocen výběr nástroje, který bude v práci využit, na základě ceny a funkcí, které jednotlivé nástroje nabízejí. Jednotlivým nástrojům bude přiřazeno bodové ohodnocení na základě splnění kritérií. Každé kritérium bude hodnoceno 1-5 body. Význam bodového ohodnocení je znázorněn v tabulce 3.2. Součástí bodového ohodnocení bude i vlastní nástroj, kde jsou body přiřazeny na základě subjektivního hodnocení jednotlivých částí. Posledním kritériem výběru je cena, kde bude 1 bod znázorňovat nejdražší nástroj, 5 bodů nástroje, které jsou zdarma dostupné a zbytek bodů bude vypočten poměrem.

Body	Význam
1	Nepoužitelné
2	Téměř použitelné
3	Sotva použitelné
4	Použitelné
5	Použitelné s výhodami

Tabulka 3.2: Význam bodového hodnocení

3.6.1 Import/Export

Prvním kritériem je schopnost nástroje importovat a exportovat modely vlastností a konfigurací ve formě souborů v různých formátech.

Nástroj pure::variants umožňuje import i export v několika formátech. Model vlastností je možné importovat jako soubor *.csv*. Nedostatkem importu je absence možnosti importu omezení. Pokud budeme importovat .csv soubor, není tato omezení možné naimportovat a je možné importovat pouze základní typy závislostí. Tato omezení je třeba přidat přímo v nástroji. Model vlastností včetně omezení se dá následně sdílet pomocí projektových souborů. Kvůli této skutečnosti je nástroj hodnocen jako použitelný. Export modelu vlastností je možný ve formátech *.xml*, *.csv*, *.html* a jako obrázek. Výsledná konfigurace lze exportovat ve formátech *.csv* a *.xml*.

Nástroj XFeature neumožňuje import ani export v žádném formátu. Za toto kritérium by tedy dostal hodnocení jako nepoužitelný. Avšak díky jeho reprezentaci pomocí xml souborů by bylo možné naimportovat nebo vyexportovat tyto soubory za účelem dalšího sdílení nebo generování z projektových souborů. Díky této možnosti je hodnocen jako téměř použitelný. Toto hodnocení jej vylučuje z hodnocení, zbytek hodnocení pro tento nástroj bude tedy spíše demonstrativní.

Nástroj FMP umožňuje import a export modelu vlastností i výsledné konfigurace ve formátu *.xml*, ale bohužel pouze ve starší verzi. Tato možnost byla z verze 0.6.6 odebrána. K importu a exportu by bylo tedy třeba využít starší verzi, což z aktuální verze činí verzi nepoužitelnou. Budeme se tedy zabývat verzí starší. Bohužel ani ve starší verzi neumožňuje žádné jiné formáty. Oproti pure::variants však umožňuje starší verze import i export, včetně globálních omezení, což je velkou výhodou. Nástroj díky popsaným skutečnostem bude hodnocen jako použitelný.

SPLOT umožňuje export modelu vlastností a konfigurací ve formátech *.xml* a *.csv*. Bohužel ale neumožňuje žádný import. Import modelu vlastností je možný pouze z online repozitáře, který obsahuje jiné modely a nový model je tedy potřeba vytvořit přímo v nástroji. Kvůli této skutečnosti se nedá k generování modelu vlastností použít a je tedy hodnocen jako nepoužitelný. Další hodnocení nástroje bude tedy stejně jako XFeature spíše demonstrativní.

Vlastní nástroj by měl umožnit import a export modelu vlastností i jeho konfigurace. Implementace by pravděpodobně obstarávala možný import a export souborů pouze v jednom formátu, což činí nástroj použitelným.

3.6.2 Přehlednost

Druhým kritériem je přehlednost reprezentace modelu vlastností. Požadavkem je tvořit a editovat modely vlastností a jejich konfigurace ve stromovém grafu, nebo jiném uživatelsky přehledném zobrazení a možnost získat z nástroje graf dle [3].

Nástroj pure::variants zobrazuje model vlastností i jeho konfigurace v přehledné stromové architektuře. Konfigurace probíhá v jiném stromě než původní model vlastností. Model vlastností je také možno zobrazit v grafu, který však neodpovídá značení podle [3]. Globální omezení jsou tvořeny v dialogovém okně a následně jsou ve stromě znázorněny přímo u vlastností. Pokud je na vlastnost ukázáno myší, barevně se rozsvítí ostatní vlastnosti, na které se tato globální omezení vztahují. Tento nástroj splňuje daná kritéria, až na zobrazení grafu dle [3]. Je tedy hodnocen jako použitelný.

V nástroji XFeature je model vlastností zobrazován grafem. V tomto grafu probíhá tvoření i editace modelu vlastností i konfigurací. Tento graf využívá jiné značení, než které je uvedeno v [3]. Toto zobrazení se však stává velmi nepřehledným ve větších a komplexnějších modelech což z něj činí nástroj téměř použitelný pro toto kritérium.

Nástroj FMP zobrazuje model vlastností v přehledné stromové architektuře podobné jako pure::variants. Avšak konfigurace je tvořena ve stejném

stromě, což může konfiguraci znepřehlednit. FMP neumožňuje model vlastností zobrazit v grafu. Další nevýhodou jsou globální omezení, která se zobrazují v jiném okně a ne přímo u vlastností. Nástroj je tedy hodnocen jako sotva použitelný.

SPLOT zobrazuje model vlastností opět ve stromové architektuře, která je velmi přehledná. Přehledná je také konfigurace, která probíhá v jiném okně pomocí vybírání jednotlivých vlastností. Globální omezení jsou popsány pomocí značek podobných matematickým symbolům průniku a sjednocení a jsou zapsány pod stromem. Přímo ve stromě však nikde nejsou vidět. Nástroj také neumožňuje zobrazení modelu vlastností v grafu. Nástroj je tedy sotva použitelný.

Přehlednost ve vlastním nástroji by musela být zařízena stromovým zobrazením a schopností vygenerovat graf podle [3]. Zobrazení by bylo pravděpodobně inspirováno nástrojem pure::variants. V dostupném čase na vývoj nástroje by pravděpodobně nástroj neměl možnost zobrazení v grafu. Zajistit celkovou přehlednost by bylo složité. Nástroj je tedy hodnocen jako sotva použitelný.

3.6.3 Konfigurace

Třetím kritériem je tvorba konfigurace z modelu vlastností. Důraz je kladen na hlídání závislostí.

Nástroj pure::variants umožňuje tvorbu konfigurace v jiném okně, než je původní model vlastností. Pro každou konfiguraci je tvořen samostatný soubor. Konfigurace je tvořena pomocí zaškrtávání jednotlivých vlastností a nástroj sám hlídá dodržení všech závislostí. Pokud jsou použita omezení, nástroj při výběru vlastnosti sám zaškrtá jiné vlastnosti, které vlastnost vyžaduje, nebo odškrtá některé, které vylučuje. Všechny konfigurace se také dají zobrazit vedle sebe v jednom z pohledů. V tomto pohledu jsou také vidět upozornění, které značí uživateli, že někde mohl vybrat variantu, která s modelem vlastností nekoresponduje a jsou mu nabízeny opravy. Nástroj tvoří konfigurace bez žádných problémů a poskytuje mnoho výhod uživateli. Je tedy hodnocen jako použitelný s výhodami.

V nástroji XFeature uživatel při tvorbě konfigurace musí opět vytvářet nový stromový diagram, u kterého jsou mu nabízeny pouze volby podle daného modelu vlastností. Tato tvorba tedy zahrnuje vytvořit strom znovu, pouze s vlastnostmi, které vyžaduje a ne formou vybírání již existujících vlastností. Tato konfigurace je validována na základě původního modelu vlastností, což uživateli nedovoluje vytvořit konfiguraci, která nekoresponduje s původním modelem. Nástroj je kvůli této formě konfigurace sotva

použitelný.

Nástroj FMP stejně jako pure::variants umožňuje tvorbu konfigurací, které jsou tvořeny pomocí zaškrtávání jednotlivých vlastností v původním stromě. Konfigurace však probíhá přímo v modelu vlastností, což může být nepřehledné. Při tvorbě jsou uživateli nabízeny všechny validní možnosti, ze kterých může při konfiguraci vybírat a jejich výběr je hlídán tak, aby korespondoval s původním modelem vlastností. Po vytvoření konfigurace se tato konfigurace uloží a je možné ji zobrazit přímo ve stromě. Nástroj je tedy použitelný.

Konfigurace v nástroji SPLOT je tvořena v jiném okně než je model vlastností. Možnosti konfigurace jsou opět hlídány na základě původního modelu vlastností. Velkou výhodou jsou dva přístupy popsané v sekci o nástroji. Po vytvoření konfigurace je možné je exportovat ve formátu *.xml* a *.csv*. Nevýhodou je však, že tuto konfiguraci nelze nijak uložit pro případné úpravy. Pokud je třeba vytvořit konfiguraci, která se bude oproti jiné lišit například ve výběru jediné vlastnosti, je třeba celou konfiguraci vytvořit znovu, což činí nástroj sotva použitelným.

Vlastní nástroj by měl umět konfigurace zobrazovat v jiném okně a hlídat ji tak, aby korespondovala s původním modelem vlastností. Tuto konfiguraci by mělo být možné uložit, načíst a exportovat v různých formátech. Náročnost tvorby této konfigurace souvisí se závislostmi mezi vlastnostmi, GUI a exportem importem. V dostupném čase by se dosáhlo sotva použitelnému tvoření konfigurace.

3.6.4 Cena

Posledním kritériem je cena. Cena je důležitým faktorem při výběru nástroje. Nejdražší nástroj bude hodnocen 1 bodem a nejlevnější nástroj 5 body. Ostatní budou hodnoceny poměrem.

Pure::variants je komerčně využívaný nástroj s velkým množstvím funkcí a podporou od svého vydavatele. To z něj činí nástroj s vysokou cenou. Cena jedné licence je 10 000 eur. Tuto licenci je třeba každý rok prodloužit. Cena tohoto prodloužení je 20% z pořizovací ceny, tudíž 2000 eur. Cena takového nástroje by tedy rostla s délkou jeho používání. Hodnotíme jej tedy jako nejdražší nástroj a přiřadíme mu 1 bod.

Nástroje XFeature, FMP a SPLOT jsou zdarma a jsou tedy hodnoceny 5 body.

Cenu vlastního nástroje jsme odhadli na 1,1 milionu korun, která je srovnatelná se čtyřmi licencemi na nástroj pure::variants. Za předpokladu, že tedy budou využity čtyři licence pro práci s nástrojem, vlastní nástroj bude

mít návratovost jeden rok. Po jednom roce se nástroj pure::variants zdraží o 20%. Kvůli tomu je vlastní nástroj hodnocem 2 body.

3.7 Shrnutí

	Imp/Exp	**Přehlednost**	**Konfigurace**	**Cena**	Celkem
pure::variants	4	4	5	1	**14**
XFeature	1	(2)	(3)	(5)	**(11)**
FMP	4	3	4	5	**16**
SPLOT	1	(3)	(4)	(5)	**(13)**
Vlastní nástroj	4	3	3	2	**12**

Tabulka 3.3: Bodové ohodnocení jednotlivých nástrojů

Pokud srovnáme bodové ohodnocení všech nástrojů, zjsistíme, že se příliš neliší. Každý nástroj má své výhody a nevýhody, které jsou popsány ve srovnání. Nejvyšší počet dostal nástroj FMP. Tento nástroj však získal vysoké bodové ohodnocení oproti nástroji pure::variants, které dosáhlo nižšího hodnocení pouze o dva body, hlavně kvůli své ceně. Zbytek hodnocení za nástrojem pure::variants zaostává. Pokud by byl nástroj FMP dolazen o přehlednější tvorbu konfigurace a oproti starší verzi by nebyla odebrána možnost importu a exportu, byl by zcela jistě použit pro účely bakalářské práce. Na druhém místě skončil nástroj pure::variants. Z tabulky je jasné, že cena je jeho největší nevýhodou a že zbytek kritérií s lehkými nedostatky splňuje. Nejnižší hodnocení dostal nástroj XFeature, který se zdá být nevhodným nástrojem pro potřeby bakalářské práce již kvůli prvnímu kritériu. Největší nevýhodou nástroje je absence přímého exportu a importu modelu vlastností a jeho konfigurací a uživatelsky nepřívětivá tvorba konfigurací. Třetím nejlépe hodnoceným nástrojem je překvapivě nástroj SPLOT, který by pravděpodobně byl nejvhodnějším nástrojem, kdyby umožňoval import modelu vlastností. Na posledním místě také skončil vlastní nástroj, jehož hodnocení bylo odvozeno na základě představy implementace v dostupném čase, tudíž žádný z jeho bodů nedostal plné hodnocení.

Důležitým faktorem jsou také preference společnosti ZF. Tato společnost již zakoupila několik licencí na nástroj pure::variants i přes možnou implementaci vlastního nástroje, který by mohl být ve finále mnohem vhodnějším nástrojem a na kterém by v budoucnu ušetřila. Avšak preferencí ZF je čas. Společnost momentálně dává přednost zakoupit licence na nástroj, který bude používat, než strávit půl roku vývojem funkčního prototypu vlastního nástroje.

Z těchto důvodů je pure::variants nejvhodnějším nástrojem a bude použit pro potřeby bakalářské práce.

4 Generativní programování

Generativní programování je druh programování, který odděluje model od implementace. Realizuje se pomocí generátorů. Generátor využívá modelu jako šablony, ze které je generován výsledný kód. Generátory jsou založeny na modelech, které definují sémantiku.

4.1 Motivace

Motivace pro generativní programování vznikla současně se vznikem počítačů. Stroje nerozumí naší řeči a jejich funkce jsou pouhými elektrickými signály. Vznikla potřeba vymyslet způsob, kterým budeme procesoru předávat informace o tom, co se po něm vyžaduje. Psaní těchto instrukcí pomocí nul a jedniček je ale nepřehledné a pro člověka nesrozumitelné. Vznikly tedy první sady instrukcí procesorů, které byli pro člověka čitelné. Tyto instrukce byly překládány přímo do strojového kódu, podle kterého procesor pracoval. Tento překlad je právě generováním kódu. Instrukce jsou využity jako šablona a překladač generuje strojový kód na základě těchto instrukcí.

S rozmachem programování vznikl požadavek na vyšší abstrakci tak, aby se instrukce co nejvíc podobali lidské řeči. To vedlo ke vzniku velkého množství programovacích jazyků, překladačů a interpreterů, které tyto komplexnější instrukce překládají na strojový kód. Překladače jsou tedy generátory, které na základě šablony generují strojový kód.

Generativní programování je o návrhu a implementaci softwarových modulů, které je možné zkombinovat a generovat tak specializované a rozsáhlé systémy splňující specifické požadavky [3]. Cílem je zmenšit mezeru zdrojovým kódem a konceptem, dosáhnout vysoké znovupoužitelnosti a adaptability software a zjednodušit správu velkého množství variant komponenty a zvýšit efektivitu [4].

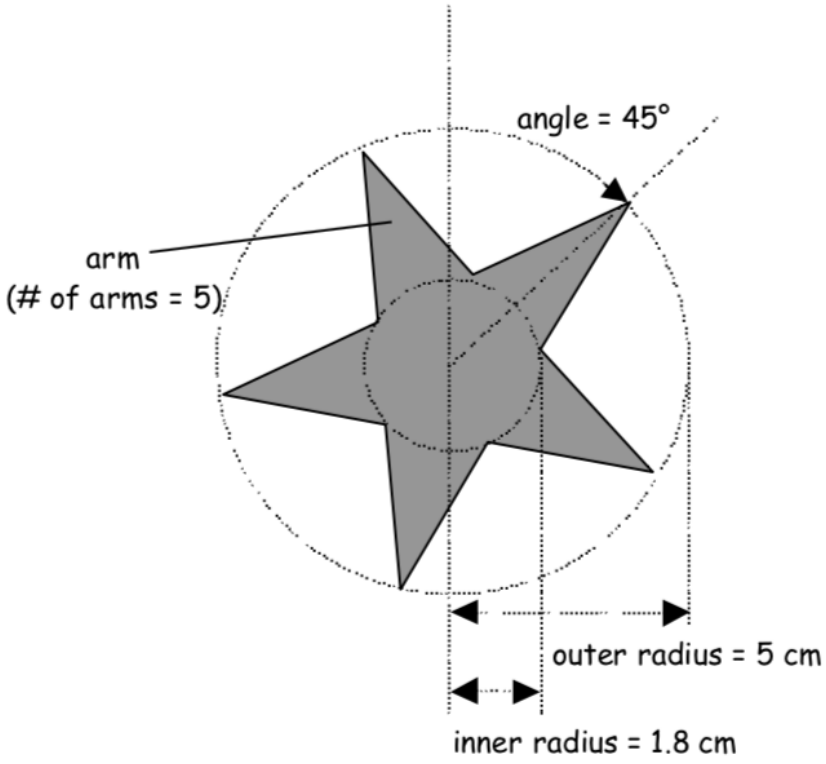
4.2 Generátory

Podle [3] jsou pro generování použity dvě základní metody: *kompoziční* a *transformační*. Generátory založené na kompozici se nazývají *kompoziční generátory* a generátory založené na transformaci se nazývají *transformační generátory*. Oba dva typy generátorů si ukážeme na příkladě, kde budeme vytvářet instanci hvězdy z různých komponent. Hvězda je sestavena z modelu

vlastností, který obsahuje:

- počet cípů
- vnitřní poloměr
- vnější poloměr
- úhel popisující naklonění prvního cípu

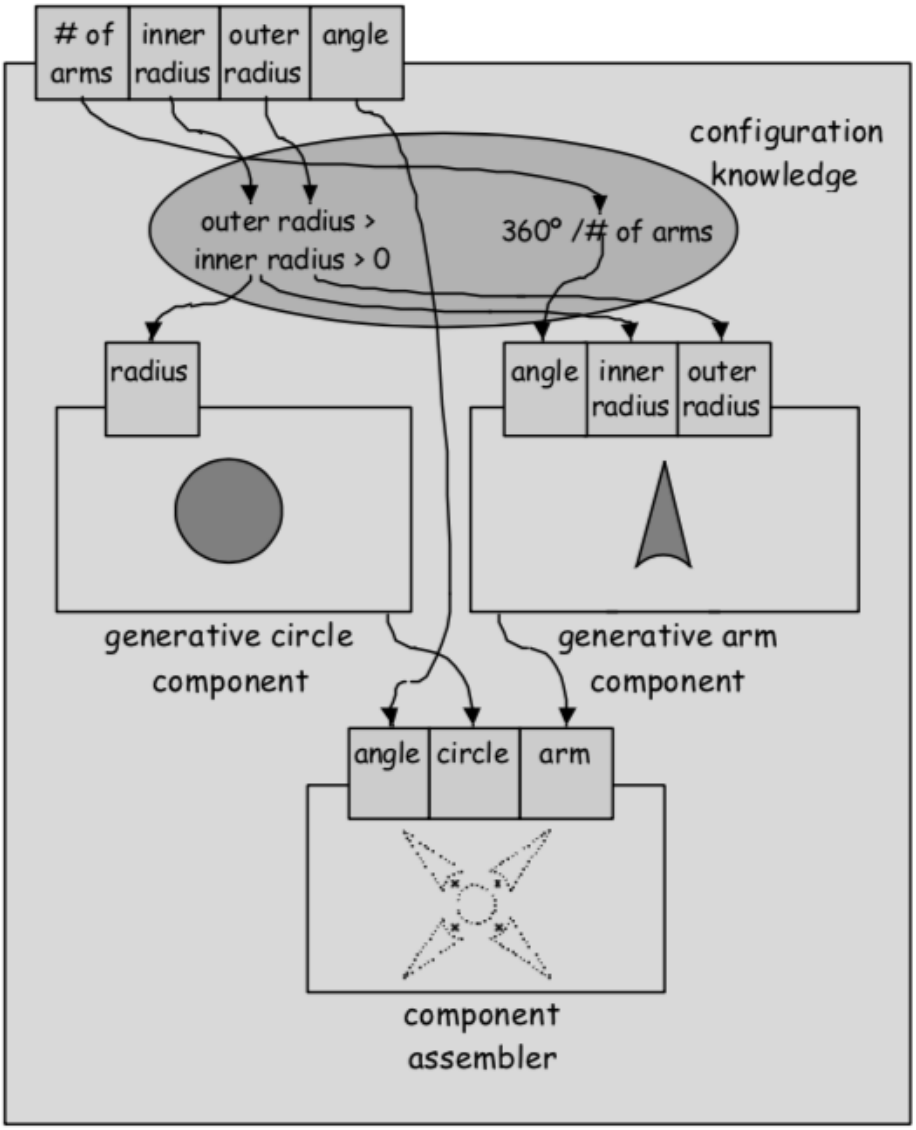
Nyní si ukážeme, jak k vytvoření hvězdy přistupují obě dvě metody.



Obrázek 4.1: Koncept hvězdy podle [3]

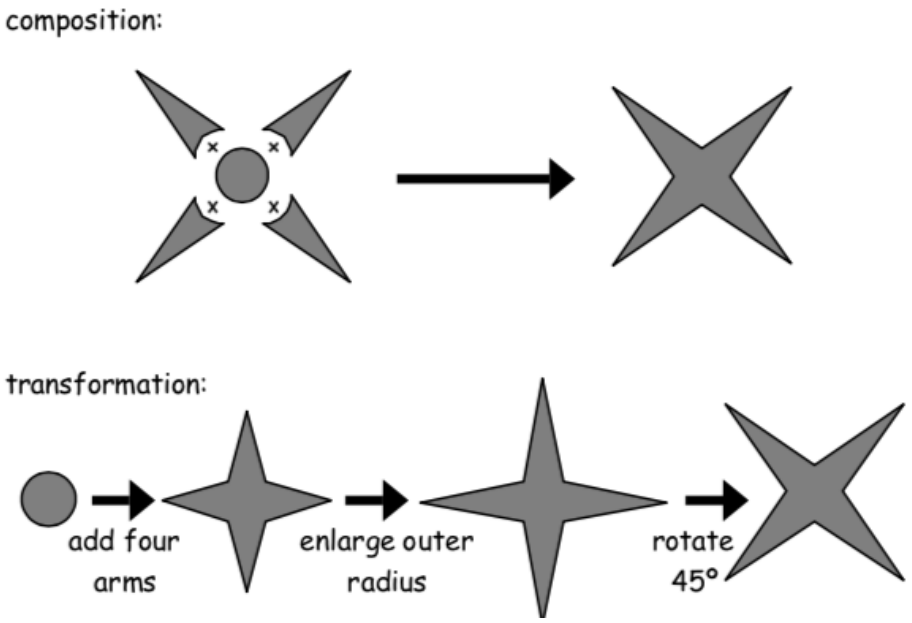
4.2.1 Kompozice

V kompozičním modelu spojíme několik komponent dohromady, které tak vytvoří požadovaný celek. K tomu, abychom byli schopni generovat různé hvězdy, je potřeba mít sadu konkrétních komponent různých velikostí a tvarů.



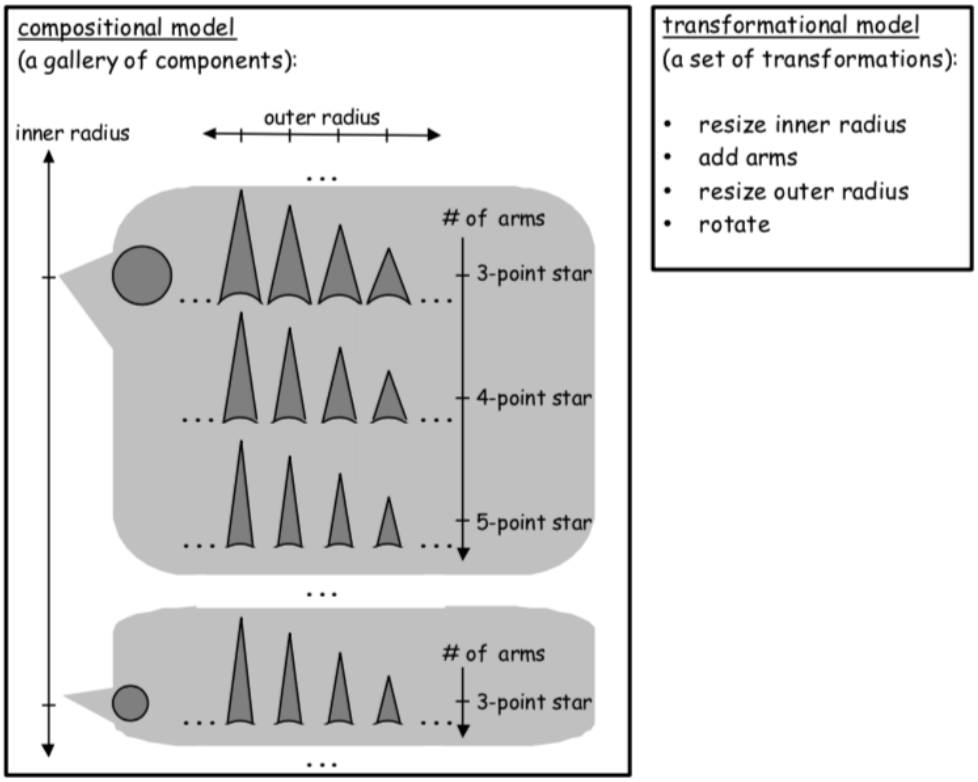
Obrázek 4.2: Proces generování hvězdy podle [3]

Kruh je popsán pouze vnitřním poloměrem, zatímco cípy jsou popsány vnitřním poloměrem, jejich počtem a vnějším poloměrem. K tomu abychom sestavili čtyřcípou hvězdu, která je vidět na obrázku, je potřeba jeden kruh a čtyři cípy vybrané ze sady konkrétních komponent, na základě jejich vlastností.



Obrázek 4.3: Sestavení hvězdy pomocí kompozice nebo transformace podle [3]

Efektivnějším způsobem sestavení instance je použití *generativních komponent* místo konkrétních komponent. Generativní komponenta využívá abstraktního popisu komponent a generuje komponentu na základě popisu. Například místo celé sady všech možných kruhů a cípů potřebujeme dvě generativní komponenty, a to generativní kruh a generativní cíp. Generativní kruh má jako parametr vnitřní poloměr a generativní cíp má jako své parametry vnitřní poloměr, vnější poloměr a úhel.



Obrázek 4.4: Doménový model hvězdy pomocí kompozice (vlevo) nebo transformace (vpravo) podle [3]

Konkrétní komponenty jsou následně vygenerovány a poskládány do požadovaného obrazce.

4.2.2 Transformace

Narozdíl od kompozice nespojujeme jednotlivé komponenty, ale provádíme určitý počet transformací, které vyústí v požadovaný výsledek. Není potřeba mít nadefinovanou sadu komponent, nebo jejich generování, ale je třeba mít nadefinované transformace, které můžeme s instancí provádět. V tomto příkladě jsou to transformace:

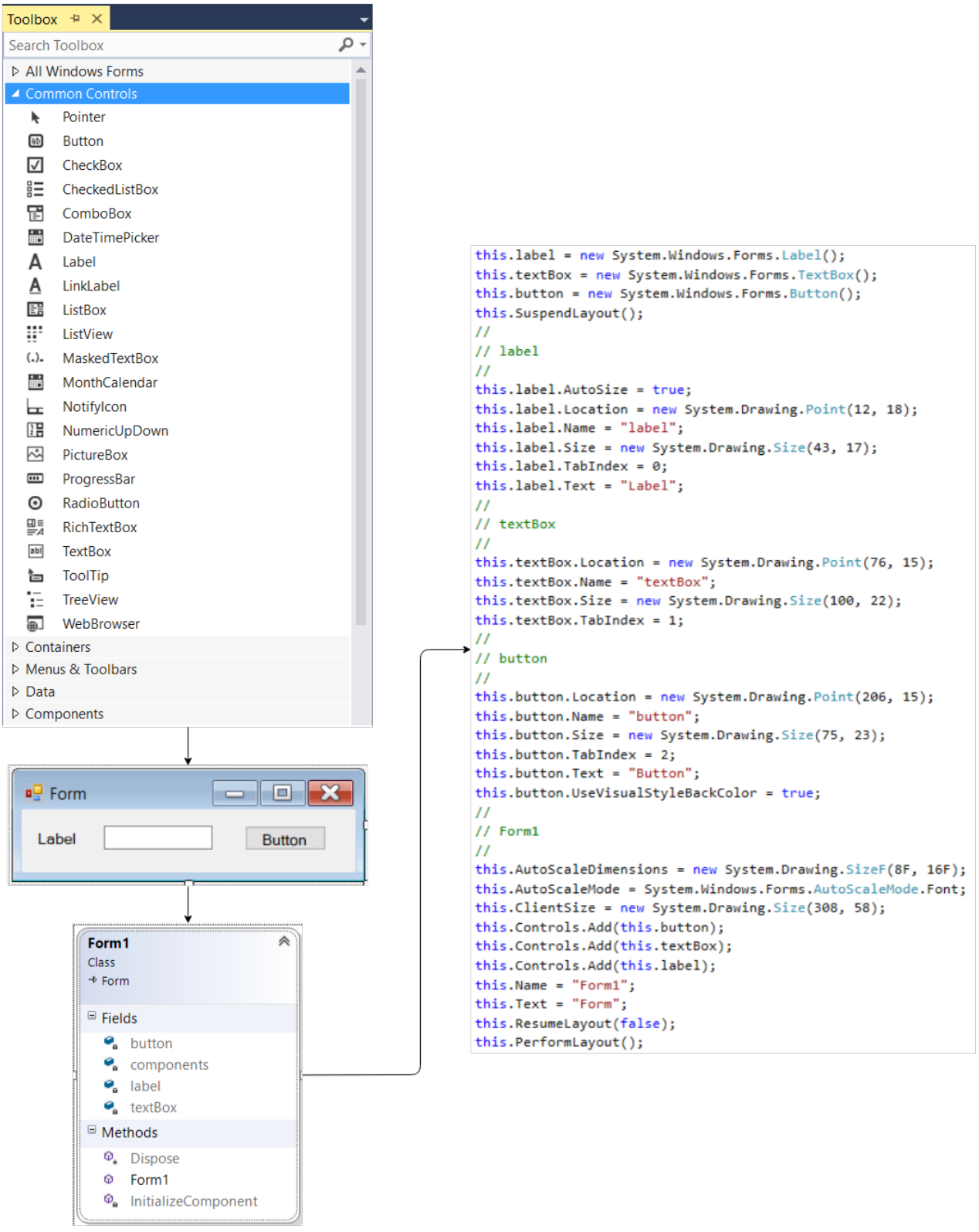
- přidej 4 cípy
- zvětš vnější poloměr
- otoč o 45 stupňů

4.2.3 Aplikace na software

V minulých částech bylo popsáno, jakými způsoby jsou tvořeny generátory. Generátory se však používají především v softwarovém odvětví. Příklad, na kterém bude generátor demonstrován, bude editor GUI.

Programátorské prostředí Visual Studio od společnosti Microsoft je umožňuje tvorbu aplikací pro windows, nebo-li *Windows Form Application*. Součástí těchto aplikací je grafické uživatelské prostředí (*GUI*). GUI je stejně jako logika aplikace popsáno zdrojovým kódem. Tento kód je ale možné generovat přes editor GUI. Po otevření editoru je programátorovi nabídnuto okno výchozí velikosti. Dále je mu nabídnut kontejner, který obsahuje veškeré komponenty, které se v tomto druhu aplikací objevují. Obsahuje například tlačítka (z angl. *button*), textová pole (z angl. *textBox*) a mnoho dalších. Programátor vybírá komponenty z kontejneru a umišťuje je do okna, podle potřeby. Každá komponenta má vícero vlastností. Těmito vlastnostmi jsou například umístění, text a další.

Během umišťování komponent do okna je Visual Studiem generován kód, který tyto komponenty popisuje. Programátor typicky do tohoto kódu následně dopisuje požadované reakce na různá tlačítka, obsahy různých listů a mnoho dalšího. Tento výsledný kód je následně součástí aplikace.



Obrázek 4.5: Proces generování

Tento přístup k tvorbě GUI šetří programátorovi spoustu času, který by bez editoru musel vynaložit na napsání kódu celého uživatelského rozhraní.

4.3 Shrnutí

I když se transformační generátor zdá být jednodušší, většina generátorů, které známe, jsou kompoziční. Příkladem takového generátoru může být editor GUI, který na základě grafického editoru, ve kterém poskládáme grafické komponenty dohromady, vygeneruje kód, který je popisuje, jak je vidět na obrázku 4.5.

Kompoziční generátor bude použit v této práci. Na základě vytvořené varianty z modelu vlastností bude generován kód jazyka tesa. Komponenty zde budou zaznamenány jako záznamy v souboru formátu XML. Z těchto komponent bude následně vygenerován výsledný kód.

5 Nástroj pro tvorbu jazyků

V této kapitole budou popsány typy jazyků a nástroj pro jejich tvorbu. Budou zde popsány *doménově specifické jazyky*, *jazyky pro obecné použití* a nástroj Xtext.

5.1 GPL a DSL

Programovací jazyky jsou nástrojem člověka, pomocí kterého dokáže sdělit počítači požadavky, které vykonávat. Programovací jazyky mají různá zaměření a funkce. Hlavními dvěma typy jsou *doménově specifické jazyky* a *jazyky pro obecné použití*. Tyto dva typy budou popsány v následujících částech.

5.1.1 GPL

Jazyk pro obecné použití (z angl. *General-Purpose Language*) (GPL) je programovací jazyk používaný pro řešení veškerých libovolně složitých problémů. Těmito jazyky je většina známých a používaných jazyků. Příkladem mohou být jazyky *C++*, *C#*, *Java*, *PHP*, *BASIC*, *Assembler*, *Python* a mnoho dalších. Platí, že každý jazyk je vhodný pro jiné platformy, nebo typy problémů.

Jednotlivé jazyky také rozlišujeme podle jejich úrovně. Nízkoúrovňové jazyky vyžadují přímou práci s pamětí počítače, kde proměnné a jiné struktury alokujeme na přesné místo v paměti. Takovými jazyky jsou například jazyky *Assembler*, *BASIC* a další.

Krom nízkoúrovňových jazyků existují i vysokoúrovňové jazyky. Takové jazyky mají práci s pamětí již implementovanou u svého překladu a není vyžadována po programátorovi, který jazyk využívá. Také obsahují velké množství datových struktur, které by si programátor musel vytvořit v nízkoúrovňovém jazyce sám. Takovými strukturami mohou být různé typy listů, hashmapy, nebo objekty. Takovými jazyky jsou například jazyky *C#* nebo *Java*.

5.1.2 DSL

Doménově specifické jazyky (z angl. *Domain-Specific Language*) (DSL) jsou jazyky, které jsou určené k přímému popisu nějaké domény. Doménou rozumíme konkrétní oblast kolem nějakého problému, či technologie. Dle [2] je

hlavní myšlenkou mít koncept a zápis co nejblíže reálnému problému v dané doméně.

Autoři uvádějí problém ze života, kde chceme z jablka odstranit jádro. GPL přirovnávají k noži, kterým jsme schopni tento proces uskutečnit, avšak je vhodný pouze pokud ho neděláme často. Ale pokud tento problém řešíme frekventovaně, je lepší použít vykrajovač jablek, v tomto případě DSL.

Dle autorů [2] existuje několik velmi známých doménově specifických jazyků. Jedním z nich je například jazyk *SQL*, který se zaměřuje na dotazování relačních databází. Dalším příkladem mohou být regulární výrazy nebo například jazyky poskytované nástroji jako je *MatLab*.

Dle [7] dělíme DSL na interní a externí. Interní DSL je jazyk, který je psán ve svém hostujícím jazyce. Tento způsob využívá svého hostujícího jazyka a vytváří v něm určité funkce, které jsou doménově specifické. Tento typ DSL může být popsán formou knihoven, které jsou do projektu psaného v obecném jazyce přidány a přidávají mu tak funkce, struktury, objekty a podobně. Jako příklad je možné uvést binární strom. Tento typ může využívat nějaký projekt, který potřebuje pro své fungování binární strom. Může tedy existovat knihovna pro tento obecný jazyk, která binární strom implementuje, implementuje nad ním různé funkce a programátor již jen využívá funkcí, které jsou pro strom standartní a nemusí znát jejich implementaci. Tato knihovna je tedy interním doménově specifickým jazykem.

Externí doménově specifický jazyk narozdíl od interního nevyužívá syntaxi hostujícího jazyka, ale je samostatně stojícím jazykem. Takový jazyk se zaměřuje na doménu, pro kterou je vytvořen. V softwarovém inženýrství je tento typ jazyka něvedomky hojně využíván. Tímto jazykem je například již zmíněný jazyk SQL, který je využíván k dotazování relačních databází. Dalším jazykem může být například jazyk *HTML*, který se využívá k tvorbě internetových stránek, nebo jazyk *CSS*, který jazyk *HTML* rozšiřuje o jednotné stylování.

5.2 Xtext

Dle [2] je Xtext profesionální open-source framework pro tvorbu programovacích jazyků vytvořen vývojáři společnosti *itemis*. Xtext poskytuje vývojáři sadu doménově specifických jazyků a moderních API k popisu různých aspektů tvořeného programovacího jazyka. Poskytuje úplnou implementaci tvořeného jazyka pro Java Virtual Machine. Komponenty kompilátoru jsou nezávislé na vývojovém prostředí Eclipse a může být využit v jakémkoliv prostředí pro vývoj Javy. Xtext využívá *EMF* (*Eclipse Modeling Framework*)

jako metamodel tvořeného jazyka. Tento model tvoří jádro našeho jazyka a jsou pomocí něho generovány soubory potřebné k překladu.

5.2.1 Instalace

Nástroj je volně stažitelný pomocí vývojového prostředí Eclipse. Postup instalace je k dispozici na stránce `https://www.eclipse.org/Xtext/download.html`. Na stránkách je k nalezení i stručný návod popisující, jak s nástrojem pracovat.

5.2.2 Tvorba jazyka

Tvorba nového doménově specifického jazyka vyžaduje popis gramatiky. *Gramatika* jazyka má dvě funkce. Zaprvé popisuje konkrétní syntaxi našeho tvořeného jazyka. Zadruhé obsahuje informace, jakým způsobem má parser tvořit model během parsování. V Xtextu má každá gramatika svůj unikátní název, který stejně jako veřejná třída jazyka Java označuje umístění souboru. Při vytváření je používána knihovna *Terminals*. Tato knihovna je součástí Xtextu a má předdefinovaná nejběžnější pravidla, jako jsou *ID*, *STRING* a *INT*. Tuto gramatiku je možné otevřít a na pravidla se podívat. Vyšlo najevo, že sada těchto pravidel je často stejná a často používaná, takže většina jazyků psaných v Xtextu tuto gramatiku rozšiřují. Není to ale podmínkou.

```
grammar org.xtext.example.mydsl.HelloWorldDSL with org.eclipse.xtext.common.Terminals

generate helloWorldDSL "http://www.xtext.org/example/mydsl/HelloWorldDSL"

Model:
    greetings+=Greeting*;

Greeting:
    'Hello' name=ID '!';
```

Obrázek 5.1: Příklad zápisu gramatiky úrovně HelloWorld

Po vytvoření gramatiky je z ní nástroj Xtext schopný vygenerovat parser, textový editor a další infrastrukturu. Po vygenerování jsme schopni spustit instanci prostředí eclipse, která obsahuje plug-in s námi vytvořeným jazykem. Zde je již možné vytvořit nový projekt a v něm vytvořit soubory s příponou vytvořeného doménově specifického jazyka a je možné psát v jeho syntaxi. Prostředí samo hlídá dodržení syntaxe a vypisuje chyby, pokud syntaxe dodržena nebyla. Klávesovými zkratkami pro napovídání je možné vy-

pisovat všechny možnosti, které je na základě syntaxe možné na dané místo psát.

```
Hello Jakub!
Hello World!
```

Obrázek 5.2: Syntaxe doménově specifického jazyka z gramatiky HelloWorld

Syntaxe psaní gramatiky v Xtextu není složitá a je možné se ji naučit v řádu desítek minut. Ukázka zápisu gramatiky bude přejata z návodu na příslušných internetových stránkách popsaných v sekci instalace. Na základě této gramatiky bude vytvořen soubor psaný v tomto doménově specifickém jazyce. Popis gramatiky musí začínat tzv. počátečním pravidlem. Obsahuje základní pravidla, která musí náš jazyk dodržovat. Toto pravidlo začíná návěštím, které si sami určíme.

```
Model:
    (elements+=Type)*;
```

Obrázek 5.3: Počáteční pravidlo

V tomto příkladě toto pravidlo vyjadřuje, že náš doménový model bude obsahovat libovolný počet (*) elementů typu *Type*, které jsou přidány do proměnné s názvem *elements*.

```
Type:
    DataType | Entity ;
```

Obrázek 5.4: pravidlo Type

Pravidlo typu *Type* deleguje pravidlům *DataType* nebo (|) pravidlu *Entity*.

```
DataType:
    "Datatype" name=ID;
```

Obrázek 5.5: pravidlo DataType

Pravidlo pro *DataType* začíná klíčovým slovem *Datatype*. Za ním následuje identifikátor, který je vyparsován z pravidla pro *ID*, které je k nalezení v knihovně *terminals*.

```
Entity:
    "Entity" name=ID ("extends" superType=[Entity])? "{"
        (features+=Feature)*
        "}";
```

Obrázek 5.6: pravidlo Entity

Pravidlo pro *Entity* opět začíná klíčovým slovem *Entity* následováno názvem. Za ním následuje volitelné (*?*) klíčové slovo *extends*, které využívá typu *superType* a přiřazuje mu proměnnou typu *Entity*. Tímto způsobem je možné dědit různé typy stejně jako v objektově orientovaném programování.

```
Feature:
    (many?="List")? name=ID "=" type=[Type];
```

Obrázek 5.7: pravidlo Feature

Poslední pravidlo v našem příkladě je klíčové slovo *many*, které je doporučené pro používání k popisu souboru více hodnot, s pojmenováním *List*. Přiřazovací operátor (*?=*) značí, že vlastnost *many* je typu *boolean*.

```
grammar org.xtext.example.mydsl.TutorialDSL with org.eclipse.xtext.common.Terminals

generate tutorialDSL "http://www.xtext.org/example/mydsl/TutorialDSL"


Model:
    (elements+=Type)*;

Type:
    DataType | Entity ;


DataType:
    "Datatype" name=ID;

Entity:
    "Entity" name=ID ("extends" superType=[Entity])? "{"
        (features+=Feature)*
        "}";

Feature:
    (many?="List")? name=ID "=" type=[Type];
```

Obrázek 5.8: kompletní zápis tvořené gramatiky

Z těchto příkladů je vidět několik nejzákladnějších syntaktických pravidel Xtextu. Klíčová slova jsou zapsána v uvozovkách. Jednoduché přiřazení je zapsáno pomocí (=), zatímco přiřazení vícero hodnot je zapsáno jako (+=)

a přiřazení typu boolean zapsáno jako (*?=*). Také je na příkladě vidět přiřazování kardinality elementům a to *?* pro volitelné elementy, *** pro jakýkoliv počet elementů a *+* pro jeden nebo více elementů.

Na základě této syntaxe jsme schopni vytvořit jednoduchý doménově specifický jazyk. Tato syntaxe byla použita pro zápis jednoduché databáze studentů. Data budou typu Int a String. První Entitou, která je v příkladě uvedena je entita *StudentDatabase*, která má svůj název a více studentů. Každý student zapsán pomocí entity *Student* má své jméno, ročník, ID, záznam o nějaké bakalářské nebo diplomové práci a list studovaných předmětů. Entita práce (*Thesis*) má svůj název, počet stran a vícero zdrojů. Entita zdroj má svůj název, rok vydání a jméno autora. A jako poslední je popsána Entita předmět (*Subject*), která má svůj název, jméno lektora a ročník, ve kterém je studován.

```
Datatype Int
Datatype String

Entity studentDatabase{
    Name = String
    List students = Student
}
Entity Student{
    name = String
    year = Int
    ID = String
    thesis = Thesis
    List subjects = Subject
}
Entity Thesis{
    title = String
    pages = Int
    List sources = Source
}
Entity Source{
    title = String
    year = Int
    author = String
}
Entity Subject{
    name = String
    lectorName = String
    grade = Int
}
```

Obrázek 5.9: Doménově specifický jazyk psaný podle gramatiky

Nástroj Xtext obsahuje krom těchto syntaktických pravidel velké množství dalších funkcí a pravidel, pomocí kterých je možné vytvořit libovolně

složitý jazyk. Tato pravidla jsou k nalezení v [2].

5.2.3 TesaTK

TesaTK (*TE Software Assets Tool Kit*) je softwarová komponenta, která byla vytvořena ve společnosti ZF. Tato komponenta slouží ke konfiguraci řídícího programu, který je nahráván do procesorů jednotek, které společnost ZF produkuje. Program ve všech jednotkách obstarává stejnou funkci, avšak je zapotřebí ho modifikovat pro jednotlivé rozdílné jednotky. Modifikace programu by znamenala měnit zdrojové kódy. Zdrojový kód je ve všech jednotkách téměr stejný, ale obsahuje velké množství různých polí a proměnných, které by bylo potřeba měnit. O tyto změny se stará právě komponenta TesaTK. Ta obsahuje dvě části. První částí je doménově specifický jazyk *tesa* a editor, vytvořené pomocí nástroje Xtext. Druhou částí je generátor, který na základě konfigurace generuje validní zdrojové kódy programu do jednotek. Generátor není pro účely bakalářské práce relevantní a nebude její součástí.

Jazyk tesa slouží k popisu a konfiguraci signálů. Gramatika jazyka obsahuje desítky vlastností, které může výsledná konfigurace obsahovat. Zápis gramatiky obsahuje krom základních syntaktických pravidel i různé výčtové typy, konstanty a další. Pomocí tohoto jazyka je možné zapsat všechny možné varianty konfigurací na základě požadovaných vstupních a výstupních signálů, variant, požadovaných hardwarových komponent, systémů a subsystémů, pro které je konfigurace tvořena.

Výhodou tohoto jazyka je hlavně přehlednost konfigurace oproti původním zdrojovým kódům programu nahrávaného do jednotek. Díky nástroji Xtext jsou při tvorbě konfigurace nabízeny možnosti, které je možné v konfiguraci zapsat a jsou hlídány syntaktické chyby, které by mohly mít velký dopad při aplikaci těchto konfigurací. Díky popisu konfigurace jako doménově specifického jazyka se specifickou syntaxí zůstávají všechny konfigurace v průběhu času jednotné a jsou tak uživatelsky přehlednější a jejich tvorba mnohem rychlejší. Při změně požadavků, nebo přidání vlastností, které jsou ke konfiguraci potřeba, stačí tyto změny zanést do gramatiky popsané v Xtextu a všechny konfigurace modifikovat na základě těchto změn. Generátor následně na základě této konfigurace vygeneruje zdrojové kódy požadovaného programu.

5.3 Další nástroje

V této části budou stručně popsány další nástroje, které slouží k tvorbě jazyků.

5.3.1 textX

Nástroj *textX* je pythonový framework inspirovaný nástrojem Xtext. Nástroj nevyužívá *EMF* jako Xtext, ale využívá metaprogramování Pythonu k reprezentaci tříd v paměti. Oproti Xtextu také nástroj textX negeneruje vývojové prostředí, ve kterém lze doménově specifický jazyk psát. Meta model je také možné vizualizovat pomocí nástroje *GraphViz*.

5.3.2 Spoofax

Spoofax je další textový nástroj pro tvorbu jazyků. Stejně jako Xtext je distribuován jako zásuvný model vývojového prostředí Eclipse, ale je možné jej používat i ve vývojovém prostředí IntelliJ. Nástroj není komerční a je určen pro výzkum. Svou funkcionalitou připomíná nástroj Xtext. Stejně jako Xtext generuje vývojové prostředí, které je schopno s vytvořeným jazykem pracovat.

5.3.3 JetBrains MPS

JetBrains MPS je nástroj, který oproti ostatním zmíněným nástrojům není textový. Nástroj je především určený ke generování zdrojových kódů jazyků pro obecné použití. Využívá reprezentace zdrojových kódů v *Abstraktním syntaktickém stomě* (z angl. *Abstract syntax tree*) (AST). Každý uzel tohoto stromu reprezentuje určitou konstrukci v programovacím jazyce a může mít skupinu potomků a rodiče. Tento nástroj na základě tvorby a editace tohoto stromu generuje zdrojový kód a umožňuje tak vytvořit program netextovým přístupem, ale pomocí editace diagramů a tabulek. Tento nástroj také umožňuje jazyky rozšiřovat o nové funkce nebo vytvořit zcela nový jazyk.

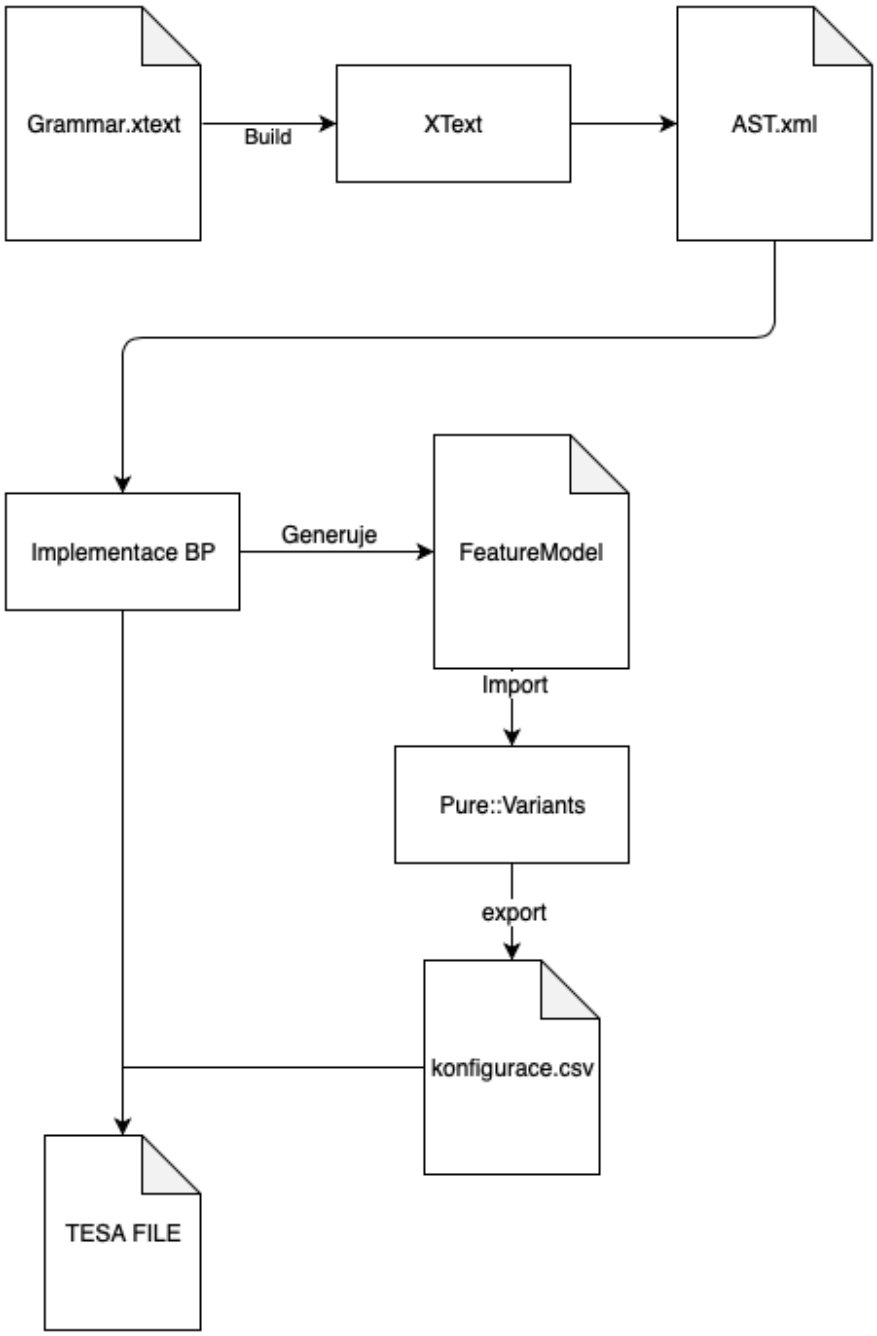
5.4 Shrnutí

Pomocí nástroje Xtext jsme schopni jednoduše a rychle vytvořit vlastní doménově specifický jazyk a současně dostat vývojové prostředí, které bude vytvořený jazyk podporovat. Jazyk TesaTK je jazyk napsaný právě pomocí nástroje Xtext. Gramatika popsaná v nástroji obsahuje výčet vlastností a

díky syntaxi gramatiky bude možné z gramatiky vytvořit model vlastností. Z tohoto modelu bude možné vytvořit varianty konfigurace a na základě těchto konfigurací bude možné generovat zdrojové kódy v jazyce tesa. Díky nástroji Xtext bude také možné tyto generované soubory kontrolovat a následně upravovat.

6 Návrh implementace

V této kapitole bude popsán návrh implementace. Vzhledem k potřebě využití nástroje pure::variants, je třeba aplikaci použít dvoufázově. V první fázi musí být program schopný vytvořit soubor modelu vlastností, importovatelný do pure::variants, ze zápisu gramatiky, kterou je třeba číst. V druhé fázi je potřeba soubor modelu variant, který je exportován z nástroje pure::variants. Tento soubor bude sloužit programu jako seznam vybraných komponent. Z těchto komponent poté program vygeneruje požadovaný soubor.



Obrázek 6.1: Proces aplikace

6.1 Načtení AST

Pro vytvoření souboru modelu vlastností je potřeba zkoumat zápis gramatiky popsaný v nástroji Xtext. Z této gramatiky je vhodné vytvořit *abstraktní syntaktický strom*. Tato struktura bude následně obsahovat veškéré komponenty zapsané gramatiky, včetně klíčových slov a přiřazení proměnných. Model vlastností bude vytvořen z tohoto stromu jako jeho část. V této části nebudou zanesena klíčová slova, která se v gramatice vyskytují.

Ke čtení se nabízejí 3 různé přístupy, které budou se svými výhodami a nevýhodami popsány dále.

6.1.1 Vlastní parser

Prvním přístupem, který se nabízí ke čtení gramatiky, je vytvořit vlastní parser, který bude číst textový zápis gramatiky a vytvoří z ní abstraktní syntaktický strom. Čtení gramatiky by probíhalo znak po znaku s jejich vyhodnocováním a postupným sestavováním.

Problémem, který v takovém případě nastane je rozmanitost způsobu zápisu. Uživatel si může svůj jazyk popsat jakýmkoliv způsobem. Různé způsoby mohou zahrnovat různé úrovně zanoření, tvořený jazyk může obsahovat speciální znaky, které se využívají i při vlastním zápisu nebo může být popsaný na různém počtu řádků. Všechny tyto problémy je možné vyřešit a napsat tak parser, který dokáže vytvořit abstraktní syntaktický strom z gramatiky jakéhokoliv jazyka. Takový parser se zdá být vhodný pro účely bakalářské práce, avšak jeho tvorba je zbytečná. Nástroj Xtext dokáže z libovolné gramatiky vygenerovat prostředí, ve kterém je možné náš jazyk psát s veškerými výhodami vývojového prostředí Eclipse. Znamená to tedy, že nástroj samotný gramatiku čte a abstraktní syntaktický strom z ní tvoří. Z tohoto důvodu není vhodné takový parser psát.

6.1.2 Ecore

Druhým přístupem, který se k sestavení stromu nabízí, je soubor *Ecore*. Tento soubor je vytvořen nástrojem Xtext při tvorbě všech komponent potřebných pro vygenerování prostředí. Soubor je popsán ve formátu XML. V tomto souboru jsou komponenty gramatiky zaneseny jako objekty typu *EObject*. Struktura xml souboru je narozdíl od původní gramatiky značně čitelnější a konzistentní. Z toho souboru je jasně vidět, jaké "třídy"gramatika obsahuje a seznam proměnných, které jsou ve třídě zaneseny.

Ačkoliv se tento přístup zdá být vhodný, obsahuje také několik problémů. Nejzásadnějším problémem, který činí tento soubor nepoužitelným pro účely

aplikace, je absence všech klíčových slov, které jsou pro generování jazyka potřeba. Soubor obsahuje pouze názvy proměnné a třídy, na které tyto proměnné ukazují. Dalším problémem je absence kardinalit jednotlivých proměnných. Tyto kardinality jsou popsány v části 5.2.2.

Tyto problémy je možné vyřešit vytvořením stromu ze souboru ecore a strom na základě načtených proměnných doplnit přímo ze souboru se zápisem gramatiky. S tím však souvisí problémy z předchozí části. Zároveň vytvořený ecore soubor v některých místech nemusí odpovídat přímému zápisu gramatiky. Příkladem této nesrovnalosti může být třída, která v zápisu obsahuje výběr ze dvou dalších tříd, které mají několik společných proměnných. Během tvorby ecore souboru jsou společné proměnné těchto tříd přiřazeny třídě nadřazené. Kvůli těmto rozdílům jsou některé proměnné obtížné dohledat.

6.1.3 GrammarAccess

Třetí přístup doporučil jeden z předních vývojářů nástroje Xtext Sebastian Zarnekow. Xtext obsahuje parser, který je schopný gramatiku přečíst a abstraktní syntaktický strom vytvořit. K tomu využívá nástroj *Antlr*. Parser při spuštění načte gramatiku právě do abstraktního syntaktického stromu. Tento strom obsahuje veškéré komponenty popsané gramatiky, včetně kardinalit a klíčových slov. Tento model je uložen v proměnné typu *GrammarAccess*. Tuto proměnnou je možné získat pomocí techniky *vkládání závislostí* (z angl. *Dependency Injection*). Do některého souboru je možné vložit závislost na *GrammarAccess* a následně je možné programově gramatiku prozkoumat. Z tohoto důvodu bude tato metoda v implementaci použita.

Tento strom je nutné načíst do vlastních struktur ve tvořené aplikaci. Je tedy potřeba tento model extrahovat z nástroje Xtext. Toho je možné docílit vytvoření souboru ve formátu xml, který tento strom bude reprezentovat se všemi komponentami, které budou použity při generování. Tento soubor bude tvořen postupným procházením stromu vytvořeného nástrojem Xtext a postupným zapisováním do tohoto souboru.

6.2 Tvorba modelu vlastností

Model vlastností bude zobrazen v nástroji pure::variants. Nástroj umožňuje import modelu vlastností ve formátu CSV. Soubor musí být strukturován způsobem popsaným v tabulce 6.1. Každý řádek tabulky odpovídá jednomu sloupci požadovaného souboru ve formátu csv.

Unique Name	Unikátní název elementu
Unique ID	Unikátní identifikátor elementu
Visible Name	Viditelné jméno elementu
Variation Type	Typy vlastností. Možné hodnoty jsou: ps:mandatory pro povinné vlastnosti, ps:optional pro volitelné vlastnosti, ps:or pro slučitelné vlastnosti a ps:alternative pro alternativní vlastnosti. Pokud není vyplněno, je použit typ ps:mandatory
Parent Unique ID	Unikátní identifikátor nadřazeného elementu
Parent Unique Name	Unikátní název nadřazeného elementu
Parent Visible Name	Viditelné jméno nadřazeného elementu
Parent Type	Typ vlastnosti nadřazeného elementu
Class	Třída elementu, ps:feature pro model vlastností
Type	Typ elementu, ps:feature pro model vlastností

Tabulka 6.1: Formát souboru pro import dle [10]

Program bude procházet abstraktní syntaktický strom načtený do paměti a tvořit z něj soubor ve formátu CSV tak, aby jej bylo možné naimportovat do pure::variants. Strom již musí obsahovat veškeré typy vlastností načtené z gramatiky.

Po načtení modelu vlastností do nástroje pure::variants je uživatel schopen vytvořit jednotlivé konfigurace nad modelem vlastností, které může z pure::variants vyexportovat.

6.3 Generování

Ke generování výsledného kódu je nutné mít v paměti načtené všechny komponenty, ze kterých se bude následný kód skládat. Hlavními komponentami v této fázi jsou klíčová slova a proměnné. Tyto komponenty budou již načteny v AST, avšak mezi nimi nejsou zaneseny žádné vazby. Například, pokud budeme mít proměnnou s názvem *configuration*, která před sebou obsahuje klíčové slovo *"Configuration"*, neexistuje mezi nimi žádná vazba, která by jednoznačně určovala, že toto klíčové slovo patří právě k této proměnné.

Generátor tedy musí tato klíčová slova hledat na základě pozic v syntaktickém stromě a hodnot klíčových slov. Tyto pozice se v gramatice opakují a je tedy možné najít vzory, podle kterých lze rozhodnout, která klíčová slova mají být generována ke konkrétním proměnným. Bohužel jsou tyto vzory specifické pro jednotlivé doménově specifické jazyky, které lze v nástroji Xtext vytvořit a generátor tedy nemůže fungovat pro jakýkoliv jazyk

popsaný v Xtextu, narozdíl od tvorby AST a modelu vlastností, které je možné vytvořit z jakéhokoliv jazyka.

Generátor bude procházet původní AST a kontrolovat, zda se ve vyexportovaném souboru z pure::variants nachází. Pokud ano, vygeneruje příslušné komponenty.

Součástí generování je také nahrazování názvů jednotlivých proměnných unikátní reprezentací tak, aby výsledná konfigurace prošla validátorem bez chyby. Je tedy potřeba zanést několik specifických podmínek pro názvy jednotlivých proměnných tak, aby byla výsledná konfigurace bezchybná.

7 Implementace

V této kapitole bude popsána implementace aplikace. Budou zde popsány jednotlivé třídy a metody, které se starají o celkovou funkci aplikace. První částí, která zde bude popsána bude gramatika a její extrakce z nástroje Xtext. Následně bude popsána implementace vlastní aplikace.

7.1 Gramatika

Tato část se zabývá extrahováním gramatiky z nástroje Xtext. Gramatika je popsána v nástroji v samostatném souboru s příponou *xtext*. Tento soubor byl modifikován pro účely bakalářské práce společností ZF Engineering kvůli zachování výrobního tajemství.

Podle návrhu popsaném v části 6.1.3 bude k přečtení gramatiky využito rozhraní *IGrammarAccess*. Toto rozhraní je možné vložit do jedné ze tříd, která je používána při běhu nástroje, konkrétně do třídy *ConfiguratorValueConverter*. K tomuto vložení se využívá techniky *vkládání závislostí* (z angl. *Dependency Injection*). Přístup k rozhraní je realizován pomocí proměnné *grammarAccess*, která je typována právě na vložené rozhraní. Model gramatiky je poté možné získat metodou *getGrammar()* přímo nad proměnnou s rozhraním. Tento model je uložen do proměnné *grammar* a obsahuje veškeré komponenty popsané gramatiky.

O sestavení modelu, který bude transformován do xml souboru je využito knihovny *org.w3c.dom*. Tato knihovna umožňuje tvořit objekty typu *Element*, přidávat jim atributy a napojovat další elementy jako potomky.

Model gramatiky obsahuje základní pravidla, která jsou v gramatice popsána. K těmto pravidlům je nad proměnnou s gramatikou možné přistoupit metodou *getRules()*, jejíž návratovou hodnotou je list objektů typu *AbstractRule*. Tento list je v cyklu procházen a každé pravidlo zpracováno. Pravidla jsou do xml souboru vypsána s klíčovým slovem *Rule*, obsahující atribut *name*.

Po vytvoření každého pravidla je volána metoda *processType*, která se stará o zpracování elementů, které obsahuje. Tyto elementy jsou několika typů. Tyto typy jsou:

- *KeywordImpl* - označuje klíčové slovo
- *AssignmentImpl* - označuje přiřazení

- *GroupImpl* - obsahuje skupinu dalších elementů, u nichž záleží na pořadí
- *UnorderedGroupImpl* - obsahuje skupinu dalších elementů, u nichž nezáleží na pořadí
- *ActionImpl* - označuje akci, nejsou relevantní pro účely bakalářské práce
- *RuleCallImpl* - označuje další pravidlo, na které tento elemenent ukazuje
- *AlternativesImpl* - označuje alternativy, mezi kterými lze vybírat
- *CrossReferenceImpl* - slouží k přiřazení již existujícího elementu
- *EnumLiteralDeclarationImpl* - označuje výčtové typy

Tyto typy jsou získány metodou *getSimpleName()* třídy *Class*, která vrací hodnotu typu *String*. Tyto hodnoty rozlišuje switch, který na základě těchto typů volá jednotlivé metody. Tyto metody se starají o zpracování různých typů a přidáním těchto elementů do modelu. Všechny metody jsou popsány dále.

Po vytvoření modelu je vytvořen finální xml soubor, do kterého je tento model transformován třídou Transformer z knihovny *javax.xml.transform*.

processKeyword

Metoda se stará o zpracování klíčových slov. Do souboru xml je element vypsán s klíčovým slovem *Keyword* a obsahuje atribut *value*, kterému je přiřazena hodnota klíčového slova.

processAssignment

Metoda se stará o zpracování přiřazení. Do souboru xml je element vypsán s klíčovým slovem *Assignment* a obsahuje atribut *name*, kterému je přiřazena hodnota podle názvu elementu. Dále obsahuje atribut *cardinality*, který značí typ vlastnosti. Následně je v metodě znovu volána metoda *processType*, která tomuto elementu přiřadí další element, který je mu přiřazen.

processGroup

Metoda zpracovává skupinu dalších elementů. Do souboru xml je vypsána klíčovým slovem *Group* a obsahuje atribut *cardinality*, který značí typ vlastnosti. Následně je volána metoda *processType* pro všechny elementy, které tato skupina obsahuje.

processUnorderedGroup

Metoda zpracovává skupinu totožně, jako metoda *processGroup*. Pro účely bakalářské práce není potřeba mezi těmito dvěma typy dále rozlišovat.

processAction

Metoda akce nijak nezpracovává, protože nejsou relevantní pro účely bakalářské práce. Metoda je zde pouze z důvodu možnosti pozdějšího rozšíření.

processRule

Metoda zpracovává pravidla, která jsou přiřazena nějakému dalšímu elementu. Tomuto elementu pak vytvoří do souboru xml prvek s klíčovým slovem *Parent* a atributem *name*, kterému je přiřazena hodnota názvu tohoto elementu.

processAlternatives

Metoda zpracovává alternativy, mezi kterými lze vybírat. Činí tak přiřazením atributu *cardinality* s hodnotou / předchozímu elementu. Následně volá metodu *processType* pro všechny své elementy.

processCrossReference

Metoda zpracovává reference napříč gramatikou. Do souboru xml je vypíše element s klíčovým slovem *Reference* s atributem *name*, který nabývá hodnoty názvu referencovaného elementu.

processEnumLiteral

Metoda zpracovává výčtové typy. Do souboru xml je element vypsán s klíčovým slovem *Enum* s atributy *name*, který nabývá hodnoty názvu výčtového typu a *value*, který nabývá hodnoty tohoto elementu.

7.2 Vstup aplikace

Aplikace je pouze konzolová a je spouštěna z příkazové řádky se vstupními parametry. Všechny parametry jsou aplikací kontrolovány. Prvním parametrem je soubor reprezentující gramatiku, jehož generování je popsáno v předchozí části. Druhým parametrem po spuštění aplikace je parametr *Režim*, kterým uživatel určuje užití aplikace.

Parametr může nabývat hodnot *{MODEL; TESA}*. Hodnotu *MODEL* uživatel použije v případě, že po aplikaci požaduje vytvořit soubor importovatelný do nástroje Pure::Variants. V takovém případě je třetím vstupním parametrem název souboru, který aplikace vytvoří. Parametr s hodnotou *TESA* použije uživatel v případě, že již má vytvořenou konfiguraci z nástroje Pure::Variants a chce vygenerovat validní tesa soubor. V takovém případě je třetím parametrem název souboru s vytvořenou konfigurací z nástroje Pure::Variants a čtvrtým parametrem název souboru, který aplikace vygeneruje.

7.3 Vytvoření AST

V obou případech nastavení režimu aplikace je nejprve spuštěna metoda *loadTree()*. Tato metoda se stará o vytvoření abstraktního syntaktického stromu v paměti aplikace. Metoda otevře soubor a použije metodu *parse()* k transformaci xml souboru do proměnné *doc*. V této proměnné jsou k nalezení elementy z xml souboru. Metoda následně v cyklu načítá všechna pravidla pomocí metody *getElementsByTagName("Rule")*, kde *"Rule"* je značka použitá pro pravidla v xml souboru. Jednotlivé elementy, které cyklus načítá jsou přetypovány na typ *Element*, který umožňuje zkoumání jednotlivých atributů xml záznamu. Pravidla, které cyklus načítá jsou uloženy do hashovací tabulky, kde klíčem je název pravidla a hodnotou objekt typu *Rule*.

Po načtení pravidel jsou do nich přidány další elementy. O to se stará metoda *addGrammarElements()*, která prochází všechny potomky pravidel, které xml soubor obsahuje. Elementy jsou stejně jako při extrakci gramatiky několika typů. O rozdělení těchto typů se stará switch, který na základě názvu záznamu vytvoří objekt správného typu. Tyto elementy jsou přidány do proměnné *elementList*, kterou objekt *Rule* obsahuje. Proměnná je implementována jako kolekce *ArrayList*. Kvůli potřebě přidání všech typů do listu elementů byla vytvořena třída *GrammarElement*, od které jsou odděděny všechny další třídy reprezentující elementy gramatiky. Zpracováním všech elementů, které xml soubor obsahuje, je v aplikaci vytvořen abstraktní syntaktický strom. Všechny prvky tohoto stromu obsahují odkaz na svůj nad-

řazený element a současně list elementů, které dále obsahují.

7.4 Vytvoření modelu vlastností

Z AST je následně potřeba vytvořit model vlastností. Do tohoto modelu nebudou zaneseny všechny typy elementů z AST. Vlastnosti zde reprezentují pouze některé typy. Těmito typy jsou:

- *Assignment*
- *Rule*
- *Parent*
- *Enum*

Tyto typy reprezentují vlastnosti, které budou zobrazovány v nástroji Pure::Variants.

Model vlastností je opět implementován stromovou architekturou. Listy tohoto stromu jsou implementovány třídou *FeatureModelNode* podobně jako listy AST, s tím rozdílem, že obsahuje atributy nutné k vytvoření souboru importovatelného do nástroje Pure::Variants. Těmito atributy jsou *id*, který slouží jako unikátní identifikátor, *puid*, neboli identifikátor nadřazeného prvku a atribut *element*, který odkazuje na element AST, který reprezentuje. Obsahuje také metodu *getPVString()*, která vrací řetězec, jehož struktura odpovídá jednomu záznamu v požadovaném souboru.

Vytvoření modelu je realizováno v metodě *createFeatureModel()*, ve které je vytvořen kořen stromu. Na tento kořen jsou napojovány další elementy pomocí metody *addNode()*. Tato metoda rekurzivně prochází původní AST a z jednotlivých elementů tvoří vlastnosti modelu. K přiřazení identifikátorů je používáno počitadlo, které je inkrementováno po vytvoření každého nového listu.

Aby bylo možné model vlastností v nástroji zobrazit, je třeba vytvořit soubor formátu *CSV*. Struktura tohoto souboru je vidět v tabulce 6.1. Soubor je vytvářen v případě, kdy uživatel použil parametr 1 při spuštění, pro vytvoření souboru s modelem vlastností. O vytvoření tohoto souboru se stará metoda *WriteToCSV()*. Metoda otevře nebo vytvoří soubor pro zapsání modelu vlastností, zapíše do něj názvy jednotlivých sloupců, záznam s kořenem stromu a následně spustí metodu *printNodeToCSV*, která rekurzivně prochází model vlastností do hloubky a pomocí *StringBuilderu* postupně zapisuje jednotlivé vlastnosti do souboru. Tento soubor je následně možné importovat do nástroje Pure::Variants.

7.5 Generování

Generování finálního souboru je spuštěno v případě, že uživatel použil parametr 2 při spuštění aplikace. Aplikace vytvoří abstraktní syntaktický strom a model vlastností ve své paměti stejným způsobem, jako při spuštění s parametrem 1. Po vytvoření těchto datových struktur však není spuštěna metoda *WriteToCSV()*, ale metoda *exportTesa()*. Metoda otevře soubor s konfigurací, která byla exportována z nástroje Pure::Variants. Následně celý soubor řádek po řádku přečte a z každého řádku uloží název vlastnosti s jejím identifikátorem do hashovací tabulky *exportMap*.

Následně metoda vytvoří *StringBuilder*, do kterého jsou všechny elementy generovány. Poté je spuštěna metoda *parsetTree()*, která rekurzivně prochází původní model vlastností opět do hloubky. Při procházení testuje, zda hashovací tabulka *exportMap* obsahuje aktuální zkoumanou vlastnost. Pokud ne, je vlastnost přeskočena. Pokud ano, znamená to, že uživatel aktuální vlastnost požaduje ve své konfiguraci a pro element z AST, který reprezentuje, je spuštěna metoda *exportElement()*. Tato metoda se stará o veškeré generování komponent do *StringBuilderu*. Návratovou hodnotou metody je booleanovská hodnota, která značí, zda byla vygenerována složená závorka *{*. Tato návratová hodnota slouží k doplnění uzavíracích složených závorek a k odsazení ve výsledném vygenerovaném souboru. Metoda na základě pozice elementu a jeho okolních elementů vyhodnocuje, jakým způsobem a jaké elementy budou do finálního souboru vygenerovány. O generování elementů se starají jednotlivé podmínky, které budou popsány dále. Po vygenerování veškerých komponent do *StringBuilderu* je jeho obsah pomocí metody *write* třídy *PrintWriter* zapsán do souboru specifikovaném při spuštění aplikace.

7.5.1 Specifické podmínky

Metoda obsahuje specifické podmínky, které jsou nutné k pojmenování některých proměnných tak, aby byl finální soubor validní. Tyto metody zkoumají pouze název aktuálního nebo naposledy generovaného elementu. Podle těchto názvů následně generují validní hodnoty, které jsou dodatečně specifikované zadavatelem. Podmínky kontrolují:

- *OutDigitalDriverTableRef* - vygenerováno *Null*
- *InDifitalDriverTableRef* - vygenerováno *Null*
- *tempSensor* - vygenerováno *TemperatureSensor Signals_name.InputSignal_name*

- *importedNamespace* - vygenerováno*use importedNamespace.**
- *maximalNumberInputSubsystems* - vygenerováno *7*
- *maximalNumberOutputSubsystems* - vygenerováno *5*
- *sortingIndex* - vygenerována hodnota počitadla *sortingIndexCounter*
- *ConfigSubsystemItem* - vygenerováno *System_name.NameIUser_name*
- *InputDriverType* - vygenerováno *System_name.NameINull_name*
- *OutputDriverType* - vygenerováno *System_name.NameONull_name*

Pokud generátor nenarazí na jednu z těchto specifických podmínkek začne procházet další podmínky tak, aby bezchybně generoval zbytek konfigurace.

7.5.2 Obecné podmínky

Metoda zjistí, jakého je generovaný element typu a na jaké pozici se nachází v listu elementů svého nadřazeného elementu. Tuto pozici uloží do lokální proměnné *index*. Switch rozpozná, zda se jedná o typ *Assignment*, *Rule*, nebo *Enum*. Pokud je element typu *Assignment*, podmínky nejprve zjístí, zda aktuální element obsahuje další elementy ve svém listu elementů. Pokud obsahuje pouze 1 další element, který je typu *Keyword*, vypíše tento *Keyword* do generovaného souboru.

Pokud element žádné další elementy neobsahuje je testováno, zda je na počáteční pozici v nadřazeném listu elementů, nebo na vyšší pozici. Pokud je element na vyšší než počáteční pozici, je zkoumáno, zda před ním stojí jiný element typu *Keyword*. Ve chvíli, kdy se na pozici před ním nachází *Keyword*, je dále testováno, zda element obsahuje referenci na nějaký další element. Pokud tuto referenci neobsahuje, je vypsán *Keyword* na pozici před elementem a současně název elementu. Pokud element obsahuje referenci na další element, je opět vypsán *Keyword* před ním a switch následně rozhoduje, jakým způsobem bude tento element generován, podle své reference. Pokud element referuje na datový typ *STRING*, je vypsán název elementu v uvozovkách. Pokud referuje na datový typ *ID*, je opět vypsán pouze jeho název. V případě, že element referuje na něco jiného, je vygenerován název reference, ke kterému je přidán suffix *"_name"*. V poslední řadě je zkoumáno, zda generovaný element není poslední ve svém nadřazeném listu elementů. Pokud poslední není a následuje za ním složená závorka (*{*), je tato závorka vygenerována a je nastaven příznak návratové hodnoty na *true*.

Pokud switch rozhodne, že element je typu *Rule*, je nejprve testován, zda se jedná o jeden z výchozích datových typů pomocí metody *exportDataType()*. Tyto datové typy jsou:

- *UINT* - vygenerována hodnota *0*
- *INT* - vygenerována hodnota *0*
- *NATIVE_FLOAT* - vygenerována hodnota *0.0*
- *HEX* vygenerována hodnota *0x00*

Pokud element není jednoho z těchto datových typů, je kontrolováno, zda na první pozici jeho listu elementů stojí *Keyword*. Pokud je současně element na druhé pozici také *Keyword* a současně složená závorka *{*, jsou tyto dva elementy vygenerovány do finálního souboru a současně je nastaven přiznak návratové hodnoty na true.

Posledním typem elementu, který switch rozpoznává je typ *Enum*. V takovém případě pouze vygeneruje jeho hodnotu.

8 Závěr

V bakalářské práci byla popsána problematika modelování vlastností software a generativního programování. Byl prozkoumán dostupný a zhodnocen software pro modelování vlastností a dostupný software pro tvorbu vlastních jazyků. Na základě nabytých znalostí byl navržen a implementován program schopný vytvořit model vlastností ze zápisu libovolného jazyka v nástroji Xtext, importovatelný do nástroje pure::variants. Program je dále schopný z konfigurace, exportované z nástroje pure::variants, generovat validní kód jazyka tesa. Funkčnost řešení byla potvrzena předem vytvořeným validátorem.

Navzdory původním předpokladům však program neušetří vývojářům žádný čas. Tato skutečnost vznikla ze dvou důvodů. Prvním důvodem je předem neočekávaná komplexnost původní gramatiky, jejíž model vlastností obsahuje přes 2000 vlastností. Pro účely bakalářské práce byla původní gramatika omezena pouze na základní moduly. I v této omezené podobě však model obsahuje 480 vlastností. Takový počet je i v přehledném prostředí pro tvorbu konfigurace příliš komplexní a pro uživatele nepřehledný. Tento problém by se dal vyřešit optimalizací původní gramatiky pro toto specifické použití.

Druhým důvodem je absence možnosti importu globálních závislostí do nástroje pure::variants. Uživatel je nucen tato omezení vytvořit a udržovat manuálně přímo v nástroji. Absence importu byla zjištěna již v analytické části při popisu nástroje, avšak zadavatelem projektu nebyla přiřazena příslušná závažnost. Vzhledem k této skutečnosti je potřeba tato omezení znovu tvořit v celém modelu při jakékoliv změně původní gramatiky jazyka a importu do nástroje. Řešením tohoto problému by mohlo být vytvoření vhodnějšího nástroje pro práci s modelem vlastností.

Čas, který by bylo třeba strávit řešením zmíněných problémů, mnohonásobně převyšuje čas, který může výsledný produkt ušetřit. Tato práce tedy dokázala, že původní představa o efektivitě byla mylná. Toto řešení tedy není vhodné pro zadaný problém a vývoj nebude dále pokračovat.

I přes zmíněné problémy se modelování vlastností ukázalo jako efektivní technika při tvorbě fyzických nebo softwarových produktů.

Literatura

[1] Antkiewicz, M. – Czarnecki, K. *FeaturePlugin: Feature Modeling Plug-In for Eclipse.* Publikováno na konferenci The 2004 OOPSLA Workshop on Eclipse Technology eXchange, 2004. Canada, WaterLoo: University of Waterloo, 2004.

[2] Behrens, H. et al. *Xtext User Guide*, 2008. Dostupné z: `https://www.eclipse.org/Xtext/documentation/1_0_1/xtext.pdf`.

[3] Czarnecki, K. *Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models.* PhD thesis, Technische Universit at Ilmenau, Germany, Ilmenau, 1998. Department of Computer Science and Automation, Vedoucí práce Prof. Dr. U. W. Eisenecker.

[4] Czarnecki, K. – Eisenecker, U. – Glück, R. *Generative Programming and Active Libraries.* Draft submitted for publication, 1998.

[5] Czarnecki, K. – Helsen, S. – Eisenecker, U. *Formalizing Cardinality-based Feature Models and their Specialization*, 2005. Canada, Waterloo: University of Waterloo.

[6] Czarnecki, K. – Helsen, S. – Eisenecker, U. Staged Configuration Through Specialization and Multilevel Configuration of Feature Models. *Software Process: Improvement and Practice.* 2005, 10. doi: 10.1002/spip.225.

[7] Fowler, M. *Domain Specific Languages.* Addison-Wesley Professional, 2010. ISBN 032171294978032171294 3.

[8] Pasetti, A. – Rohlík, O. *Technical Note on a Concept for the XFeature Tool.* Technical report, P&P Software, [s.l.], 2005. Dostupné z: `http://www.pnp-software.com/XFeature/pdf/XFeatureToolConcept.pdf`.

[9] Pikl, J. *Nástroj pro modelování vlastností software.* Master's thesis, University of West Bohemia, Czech Republic, Pilsen, 2013.

[10] GmbH. *pure::variants User's Guide.* pure-systems, [s.l.], 2013. Dostupné z: `http://www.pure-systems.com/fileadmin/downloads/pure-variants/doc/pv-user-manual.pdf`.

[11] Riebisch, M. et al. *Extending Feature Diagrams with UML Multiplicities*, 2002. Germany, Ilmenau: Ilmenau Technical University.