MATEMATICKO-FYZIKÁLNÍ
FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Jakub Kolšovský

Metajazyk generující zdrojové kódy

Katedra softwaru a výuky informatiky

Vedoucí bakalářské práce: RNDr. Martin Pergel, Ph.D.

Studijní program: Informatika

Studijní obor: Programování a softwarové systémy

Praha 2020

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V …..... dne..... podpis

Týmto by som chcel poďakovať vedúcemu mojej práce, pánovi RNDr. Martinovi Pergelovi, Ph.D., za užitočné rady pri vývoji aplikácie a následnému spracovaniu bakalárskej práce.

Název práce: Metajazyk generující zdrojové kódy

Autor: Jakub Kolšovský

Katedra / Ústav: Katedra softwaru a výuky informatiky

Vedoucí bakalářské práce: RNDr. Martin Pergel, Ph.D., Katedra softwaru a výuky informatiky

Abstrakt: Novodobé programování jde ruku v ruce s objektovo-orienovanými programovacími jazyky. Tyto jazyky nám nabízeji mnoho, ať už se jedná o built-in prvky jazyka, nebo standardní knihovny daného programovacího jazyka. Šikovný programátor umí vlastnosti daného programovacího jazyka využít na maximum. Mezi znalosti takového programátora patří i návrhové vzory, dnes už neodmyslytelná část programování. V praxi, tato znalost pomáhá při tvorbě přehledného jednoduchého a rozšiřitelného zdrojového kódu. Tyto vlastnosti jsou klíčové při tvorbě kvalitního softwaru. Cílem této práce je čtenáři přiblížit tyto vzory a ukázat mu, jak by měly být implementovány. Mimo jiné obsahuje i několik šablon, které je dobře poznat.

Klíčová slova: Návrh programovacího jazyka, implementace překladače, návrhové vzory, šablony

Title: Metalanguage generating source codes

Author: Jakub Kolšovský

Department: Katedra softwaru a výuky informatiky

Supervisor: RNDr. Martin Pergel, Ph.D., Katedra softwaru a výuky informatiky

Abstract: Modern programming goes hand-in-hand with object-oriented programming languages. These languages offer a lot of features, either built-in elements or standard libraries. Skillful programmer is able to maximize the effectivity of these features. One of the skills of such programmer is considered to be knowledge of the design patterns. In practice, they help with producing simple, clean and extendible source code. Such code leads to development of top-notch software. Goal of this thesis is to show some of the design patterns and, more importantly, show how they are supposed to be implemented. Among other things, I will also mention few templates that I personally consider useful to know.

Keywords: programming language design, implementation of a compiler, design patterns, templates

Názov práce: Metajazyk generujúci zdrojové kódy

Autor: Jakub Kolšovský

Katedra / Ústav: Katedra softwaru a výuky informatiky

Vedúcí bakalárskej práce: RNDr. Martin Pergel, Ph.D., Katedra softwaru a výuky informatiky

Abstrakt: Novodobé programovanie ide ruka v ruke s objektovo-orientovanými programovacími jazykmi. Tieto jazyky nám toho ponúkajú veľa, či už sa jedná o zabudované prvky jazyka, alebo štandardizované knižnice pre daný programovací jazyk. Šikovný programátor vie vlastnosti daného programovacieho jazyka využiť na maximum. Medzi znalosti takého programátora patria aj návrhové vzory, dnes už neodmysliteľná časť programovania. V praxi, táto znalosť pomáha pri tvorbe prehľadného, jednoduchého a rozšíriteľného zdrojového kódu. Tieto vlastnosti sú kľúčové pri tvorbe kvalitného softwaru. Cieľom tejto práce je čitateľovi priblížiť tieto vzory a ukázať mu, ako by mali byť implementované. Okrem iného, obsahuje aj zopár šablón, ktoré je dobré poznať.

Kľúčové slová: Návrh programovacieho jazyka, implementácia prekladača, návrhové vzory, šablóny

Obsah

Úvod

Ako sa pomaly vyvíjali programovacie jazyky, sila a možnosti týchto jazykov sa zväčšovali. Prešli sme si časmi, kedy bolo bežné, že najpoužívanejšie a najmodernejšie programovacie jazyky fungovali na báze ukazovateľov a manuálnej práce s nimi. Toto bola situácia v 60. a 70. rokoch minulého storočia. Už vtedy sa začalo rozmýšľať, ako túto myšlienku zmodernizovať. Veľa vizionárov prichádzalo so zaujímavými nápadmi, ktoré naberali na popularite od začiatku 80. rokov. Najviac medzi programátormi a dizajnérmi programovacích jazykov zarezonovala myšlienka objektovo-orientovaného programovania.

Čo je podstata objektovo-orientovaného programovania? V krátkosti, všetko je objekt. Objekt je forma dátovej štruktúry obsahujúca vlastnosti a metódy. Ako dobrý príklad si môžeme predstaviť auto, ktoré má isté vlastnosti (farba, motor, atď.) a isté metódy, resp. funkcie (otvor dvere, naštartuj, atď.). Objekt definujeme triedou. Každá aplikácia v takomto jazyku sa dá klasifikovať ako súhrn jedného či viacerých objektov, ktoré spolu kominukujú.

Prístup jednotlivých spoločností k vývoju ich vlastného objektovo-orientovaného jazyka sa líši. Tieto jazyky môžeme rozlišovať podľa rôznych kritérií. Medzi tieto kritériá patrí napríklad objektová „čistota“ jazyka, t.j. nerozlišujeme medzi primitívnymi typmi a objektami (Smalltalk), alebo či daný jazyk podporuje prvky imperatívneho alebo funkcionálneho programovania (Kotlin), alebo či je daný jazyk staticky alebo dynamicky typovaný (PHP).

Moderný programátor potrebuje efektívne využiť objektovo-orientované prvky jazyka, ale aj tie ostatné. Prísť na optimálne riešenie nie je jednoduché. Treba vziať do úvahy rôzne faktory. Medzi najdôležitejšie faktory patria napríklad časová a priestorová zložitosť, jednoduchosť zdrojového kódu alebo tzv. „open-closed“ princíp. Tento princíp hovorí, že zdrojové kódy by mali byť uzavreté na zmeny („closed“) a otvorené („open“) na rozšírenia. Ako maximalizovať efektivitu naprieč všetkými týmito faktormi?

Každý problém je iný, vždy musí programátor vymyslieť nové riešenie. Generické riešenie neexistuje, no v roku 1994 vyšla kniha Design Patterns: Elements of Reusable Object-Oriented Software od tzv. „The Gang of Four“: Erich Gamma, Richard Helm, Ralph Johnson a John Vlissides [1]. V knihe píšu o 23 návrhových

vzoroch, t.j. doporučených riešení daných problémov v objektovo-orientovaných jazykoch. Každý z týchto vzorov načŕta jednu situáciu a prináša ideálnu implementáciu rozhrania. Treba spomenúť, že táto kniha je stará vyše 25 rokov a za ten čas si jazyky prešli podstatnými zmenami, teda príklady obsiahnuté v tejto knihe môžeme považovať za zastaralé.

Jeden z hlavných cieľov tejto práce je vziať myšlienky tejto knihy, aktualizovať jednotlivé implementácie a zahrnúť ich do nového programovacieho jazyka. Podstatou tohto nového jazyka je, že obsahuje prvky moderných jazykov ako je Java a C# a nad týmito jazykmi sú pridané ďalšie kľúčové slová, ktoré podporujú generovanie návrhových vzorov a šablón. Je treba podotknúť, že nie každý návrhový vzor sa dá prekladačom vygenerovať, teda práca obsahuje aj analýzu toho, či daný návrhový vzor je implementovateľný, alebo nie.

Ak by sme chceli zhrnúť v pár vetách tento prekladač, tak za prvé by išlo o vývin nového programovacieho jazyka založeného na prieniku jazykov Java a C#, ktorý je obohatený o nadstavbu podporujúcu návrhové vzory a šablóny. Tento jazyk dostal názov MetaDPT, keďže sa jedná o metajazyk, ktorý podporuje generovanie návrhových vzorov (Design Pattern) a šablón (Template). Po druhé, keďže gramatika je prinekom týchto dvoch jazykov, tak používateľ má na výber, či výstupom tejto aplikácie má byť zdrojový kód v jazyku C# alebo Java.

1. Základné princípy prekladača

Na slovo prekladač sa môžme pozrieť z pohľadu rôznych vedeckých odvetví. Nezáležiac na tom, či sa na to pozrieme z pohľadu prekladu prirodzeného alebo programovaciecho jazyka, pointa ostáva. Ide o **preklad** z jedného **jazyka** do druhého.

Pri **myšlienke prekladu** jazyka v programovaní každému napadne ako prvé, že ide o preklad programovacieho jazyka do strojového jazyka procesora. Nejedná sa vždy len o túto možnosť. Jazyky môžme prekladať do rôznych foriem. Môže ísť o preklad do iného programovacieho jazyka, ktorý nemusí byť striktne strojový jazyk procesora. Dnes sa vyvýjajú aj programy, ktoré navzájom prekladajú prirodzené jazyky. V inom prípade si môžme predstaviť preklad jazyka, ktorého výstup tvorí výsledok (preklad jazyka infixových výrazov).

Čo vlastne ten prekladač je? Z pohľadu používateľa prekladača ide o tzv. **čiernu skrinku**. Príde jej vstupný súbor, vo vnútri sa udeje nejaká „mágia“ a na výstupe sa zobrazí výsledok prekladu, ktorý je popísaný v užívateľskej dokumentácii daného prekladača. Táto informácia používateľovi stačí, aby ho vedel používať.

Z pohľadu vývojára prekladača je táto situácia komplexnejšia. Odborne povedané: „Majme vstupný jazyk L_{in} generovaný gramatikou G_{in}, výstupný jazyk L_{out}, generovaný gramatikou G_{out} alebo akceptovateľný automatom A_{out}. Prekladač je zobrazenie $L_{in} \rightarrow L_{out}$, kde $\forall w_{in} \in L_{in}\ \exists\ w_{out} \in L_{out}$. Zobrazenie neexistuje pre $w_{in} \notin L_{in}$“ [2]. Voľnejšie povedané, ide o program, ktorého vstup a výstup spĺňa dané normy (gramatiku) a pre každý vstup spĺňajúci normy vstupu existuje výstup spĺňajúci normy výstupu. Tento proces prekladu sa delí na 3 fázy. Ide o **lexikálnu**, **syntaktickú** a **sémantickú** analýzu.

Prekladače fungujú na princípoch už spomínaných gramatík, resp. automatov. Vo všeobecnosti, **automat** je pätica

$$A = (Q, \sum, \delta, S, F),$$

kde Q je množina stavov, ∑ je množina vstupných symbolov, δ je prechodová funkcia, S je množina počiatočných stavov a F množina koncových stavov [3]. Niektoré komplexnejšie automaty toho obsahujú viac, ale vo všeobecnosti je toto základom. **Gramatika** je štvorica

$$G = (V, T, P, S),$$

kde V je množina neterminálov, T je množine terminálov, P je množina produkčných pravidiel reprezentujúca rekurzívnu definíciu jazyk a S je počiatočný symbol gramatiky [3].

1.1. Lexikálna analýza

Lexikálna analýza je prvá fáza pri preklade. Ako vstup dostane vstupný súbor, ktorý spracuje a jej výstupom je **postupnosť** tzv. **tokenov**. V oblasti gramatík sa token považuje za terminál.

Princípom lexikálnej analýzy je vytvorenie **konečného automatu**, tzv. skeneru, ktorý rozpoznáva tokeny popísané v „slovníku“ lexikálneho analyzátora. Po každom rozpoznanom tokene sa automat reštartuje a ide odznova. Pri každom reštarte uloží rozpoznaný token do výstupnej postupnosti, ktorá bude následne posuntá ďalej syntaktickej analýze [2].

Vo väčšine prípadov, slovník lexikálneho analyzátora neobsahuje len tokeny, resp. **terminály**. Kleeneho veta hovorí, že každý **regulárny výraz** vieme popísať ako λ-NFA automat, nedeterministický konečný automat s λ prechodmi. Tento automat vieme následne previesť na DFA, deterministický konečný automat. Z tohto vyplýva, že ak lexikálny analyzátor využíva konečný automat, tak lexikálny analyzátor dokáže spracovať okrem terminálov aj regulárne výrazy. Toto nie je žiaden prelomový objav, dnes je bežné, že regulárne výrazy sú súčasťou lexikálnych analyzátorov.

Keďže lexikálna analýza funguje na princípe DFA, tak charakterizujeme jazyky prijímané v tejto analýze ako regulárne, teda môžme v tejto analýze okrem terminálov a regulárnych výrazov využívať aj **produkčné pravidlá**. Aby konečný automat dokázal spracovať produkčné pravidlá, tak musia spĺňať formu:

$$A \rightarrow \omega B,\ A \rightarrow \omega,\ A,B \in V,\ \omega \in T^*.$$

Takúto gramatiku nazývame regulárnu a tú vieme previesť na konečný automat [3] a môžme v slovníku použiť produkčné pravidlá s maximálne jedným neterminálom.

Na záver lexikálnej analýzy sa ponúka otázka, prečo obmedzovať lexikálny analyzátor len na regulárne jazyky a im prislúchajúce automaty? Proces lexikálnej analýzy môže zabrať 60 – 80 % času prekladu [2], preto je potrebné, aby bola použitá **najefektívnejšia** technológia. Ak by sme dali lexikálnemu analyzátoru väčšiu výpočtovú silu vo forme bezkontextovej gramatiky a zásobníkového automatu, dramaticky by sa zvýšila doba trvania tejto analýzy. Toto pozorovanie vychádza z faktu, že časová zložitosť zásobníkového automatu je $O(n^2 \log_n)$ [8].

1.2. Syntaktická analýza

Syntaktická analýza nasleduje lexikálnu, t.j. druhá v poradí fáza pri preklade jazyka. Ako vstup dostane postupnosť tokenov, ktorá je výstupom lexikálnej analýzy. Túto postupnosť spracuje do **derivačného stromu**.

Jej hlavnou úlohou je rozhodnúť, či daný token, resp. postupnosť tokenov, patrí do vstupného jazyka L_{in}. Rozhoduje sa na základe syntaxe daného jazyka, ktorá je definovaná **súborom gramatických pravidiel**. Ponúka sa analógia k prirodzenému jazyku, kde slovník je súbor slov (tokenov) a syntax (vstupná gramatika G_{in}) je návod, ako tieto slová skladať do viet (statementy jazyka).

Ďalšou dôležitou úlohou tejto analýzy je staviať už spomenutý derivačný strom. Ide o **zakorenený strom**, kde vrcholy sú terminály a neterminály. Hrany vedú z vnútorných vrcholov reprezentujúcich neterminály na ľavých stranách pravidiel do všetkých (ne)terminálov z pravých strán pravidiel [2].

V tejto analýze chceme definovať zložitejšiu syntax, nevystačíme si len s jedným neterminálom na pravej strane. Kým pri lexikálnej analýze bol maximálne jeden neterminál na pravej strane, priebeh automatu bol lineárny. Derivovalo sa podľa jedného produkčného pravidla a potom, ak existovalo, sa pokračovalo v derivácii podľa neterminálu na pravej strane. Takéto produkčné pravidlá

neprinášajú veľkú výpočtovú silu. K definovaní slovníka to ale bolo adekvátne a časovo optimálne. Pri syntaktickej analýze sa rozširuje výpočtová sila v zápisoch produkčných pravidiel. Sú povolené **bezkontextové** jazyky, t.j. na pravej strane produkčného pravidla je môžný zápis viac ako jedného neterminálu podľa nasledujúcej normy:

$$A \rightarrow \omega,\ A \in V,\ \omega \in (V \cup T)^*.$$

Priebeh syntaktickej analýzy už nebude lineárny a nevystačí si len s (ne)deterministickým konečným automatom. Vyžaduje sa výkonnejší automat. Pri prechode jednotlivými pravidlami sa automat vnára do rekurzie a potom z nej vynára. Na pomoc pri rekurzii je ideálny zásobník, preto najmenej komplexnejší automat, ktorý dokáže prijať bezkontextové jazyky bude **zásobníkový automat**, PDA. V skratke, hrany tohoto automatu okrem požiadavku prechodovej funkcie obsahujú aj znak, resp. množinu znakov, ktorá sa uloží na vrchol zásobníku, ak sa po tejto hrane pohne automat ďalej. Špeciálnym znakom sa vyjadruje, že zásobník nemá ukladať nič, ale má z vrchola zásobníku odoberať.

1.3. Sémantická analýza

Treťou fázou prekladu je sémantická analýza. Každému symbolu gramatiky priradí **množinu atribútov**. Hodnoty týchto atribútov im nastaví prekladač na základe sémantických pravidiel. Syntaktický strom s atribútmi nazývame označkovaný syntaktický strom. Napríklad máme gramatiku:

$$S \rightarrow D\ \text{‘+’}\ D;$$

$$D \rightarrow [0\text{-}9]^*;$$

V takomto prípade by sa pravidlu priradili atribúty:

$$S.val \rightarrow D.val + D.val.$$

V skratke, sémantická analýza dodáva hodnoty premenným, **spracováva chyby**, atď.

2. Návrhové vzory a šablóny

Na každý problém existuje veľa rôznych riešení, no nie každé je **optimálne**. Optimálne riešenie musí byť čo najefektívnejšie po väčšine stránkach. Medzi najdôležitejšie faktory patrí časová zložitosť, priestorová zložitosť a tzv. „open-closed“ princíp, ktorý hovorí, aby zdrojový kód bol uzavretý na zmeny a otvorený na rozšírenia. Medzi menej podstatné faktory môžeme zaradiť napríklad jednoduchosť a prehľadnosť zdrojového kódu. Existujú problémy, na ktoré existuje jedno ideálne riešenie, no to nemusí platiť genericky. Programátor sa stretne aj s problémami, ktorých riešenie nie je jednoznačné. Napríklad, jedno riešenie môže mať lepšiu časovú zložitosť, druhé tú priestorovú. Vtedy sa musí programátor zamyslieť a obetovať jeden z faktorov na úkor toho druhého, ktorý je v danej situácii dôležitejší.

Toto zamyslenie nás privádza k pojmu **návrhový vzor**. S týmto pojmom sa programátor stretne už v jeho začiatkoch a myslí sa ním návod, ako vyriešiť daný implementačný problém. S prvou príručkou prišli v roku 1994 štyria páni (E.Gamma, R. Helm, R. Johnson, J. Vlissides), ktorí si vytypovali 23 často sa opakujúcich problémov a vymysleli na ne riešenie. Tieto riešenia efketívne využívajú objektovo-orientované prvky jazyka C++ a Smalltalk a maximalizujú efektivitu aj vo vyššie spomínaných ohľadoch. Táto príručka nám nepriniesla len efektívne riešenia na 23 problémov, ale aj rozšírila slovnú zásobu. V praxi, programátori pri riešení problému nemusia dopodrobna strácať čas vysvetľovaním nápadu, stačí im pomenovať návrhový vzor a ostatní už vedia, o čo sa jedná.

Každý z návrhových vzorov rieši iný problém, no niektoré problémy sú podobné. Na základe týchto podobností vieme 23 základných návrhových vzorov rozdeliť na tri skupiny: **tvorivé**, **štrukturálne** a **behaviorálne** návrhové vzory.

Tvorivé návrhové vzory sú také, ktoré ovplyvňujú tvorbu objektu. Pointou týchto návrhových vzorov je tvorba objektu formou zapúzdrenia, t.j. používa sa metóda, ktorá vracia daný objekt, navonok sa nevolá konštruktor pomocou kľúčového slova **`new`** [4].

Čo sa týka štrukturálnych návrhových vzorov, ide o vzory, kde sú triedy a objekty zložené do väčších štruktúr. Buď ide o využitie dedičnosti za účelom skladania rozhraní alebo implementácie, alebo ide o skladanie objektov za účelom dosiahnutia novej funkcionality, ideálne skladanie až za chodu programu [4].

No a nakoniec pri behaviorálnych návrhových vzoroch hovoríme o tom, ako sa dané triedy a objekty majú správať. Funkcionalita a zodpovednosť sa rozdeľuje medzi triedy a dôraz sa dáva najmä na komunikáciu medzi objektami [4].

Rovnaká myšlienka ako pri návrhových vzoroch môže byť aplikovaná aj na iné problémy, ktoré sa nejakým spôsobom od týchto vzorov líšia. Ide hlavne o problémy, kde dominantná súčasť riešenia nie je postavená na princípe objektovo-orientovaných prvkov daného programovacieho jazyka. Riešenia na tieto problémy môžeme nazvať **šablóny**. Príkladom šablóny môže byť základná implementácia komunikácie medzi klientom a serverom alebo inicializácia čítačky XML dokumentov.

3. Myšlienkové jadro prekladača

Ako to už platí pri každom softwari, výber **správneho** programovacieho jazyka, nástroja a implementačného prostredia je podstatný pri vývoji kvalitnej aplikácie. V nadväznosti na minulú kapitolu, je potrebné vybrať nástroj na jednotlivé analýzy a implementačný jazyk, ktorý daný nástroj podporuje.

3.1. Výber prekladového nástroja

Najznámejší, a najstarší, nástroj na lexikálnu analýzu je Lex, v novšej verzii **Flex**. Jemu prislúchajúci syntaktický, resp. sémantický, analyzátor je YACC, v novšej verzii **Bison**. Ako protipríklad sa ponúka nástroj **ANTLR**, ANother Tool for Language Recognition. Tento nástroj pod sebou zahŕňa funkcie všetkých troch analýz. Treba zvážiť plusy a mínusy oboch týchto alternatív.

Z pohľadu rýchlosti je Flex/Bison na tom lepšie ako ANTLR, no síce mať rýchly software je pridanou hodnotou, ale nemôžeme vziať do úvahy len toto kritérium.

Ďalej je potrebné zvážiť praktickú využiteľnosť týchto nástrojov. Kým Flex/Bison podporuje len C, C++ a Javu ako implementačné jazyky, množina takýchto jazykov pri nástroji ANTLR je väčšia. ANTLR podporuje rovnaké jazyky ako Flex/Bison a naviac ešte Python, PHP, JavaScript, Go, Swift a C#. Z tohto pohľadu, ak by niekto chcel využiť gramatiku a nad ňou postaviť novú aplikáciu, môže tak spraviť, aj keď nepozná C, C++ a Javu. Kým C/C++ nie sú jazyky, ktoré sa líšia ostatným objektovo-orientovaným jazykom až na syntax, využiteľnosť tohto prekladača z programátorského hľadiska bude vyššia, ak sa pri výbere implementačného jazyka odtienime od jazykov, v ktorých si programátor menežuje pamäť sám. Na základe tejto myšlienky sa hodia najviac jazyky Java, C#, Python a Smalltalk. Front-end jazyky ako PHP a JavaScript sa nehodia, pretože neriešime žiadnu vizualizáciu. Smalltalk síce bol jedným zo základných jazykov použitých v knihe od *The Gang of Four* [1], no tento jazyk sa už dlhé roky nevyužíva a drvivá väčšina súčasných programátorov ho nepozná. Keďže výstup prekladača bude

v jazykoch Java alebo C#, tak by sa hodilo, aby aj implementačný jazyk bol v jeden z týchto dvoch jazykov. Pri rozhodnutí medzi jazykmi Java a C# rozhodol najmä argument o prístupe jednotlivých jazykov ku dedičnosti, na ktorej bude postavené jadro aplikácie. Vďaka možnosti rozlíšenia medzi **virtuálnymi** a **nevirtuálnymi** metódami, ktoré jazyk C# ponúka, tak tento jazyk vyhovuje viac ako Java.

Ako posledné kritérium sa patrí zvážiť, aké **jazyky** jednotlivé nástroje podporujú z pohľadu teórie gramatík. Bison predvolene generuje LALR(1) analyzátor, ktorý výpočtovou silou je efektívnejší ako LL(*) jazyky, ktoré podporuje ANTLR. Z tohto pohľadu by sa pýtalo vybrať si Flex/Bison, no na princípe, na akom bude postavená gramatika, si prekladač vystačí s jazykmi, ktoré podporuje ANTLR.

Po zvážení týchto kritérií by síce bolo pekné mať rýchlejšiu aplikáciu, no na druhej strane myšlienka implementácie prekladača v jazyku C# je opodstatnená z viacerých hľadísk. Záver z posledného kritéria bol neutrálny, čiže padlo rozhodnutie pre **ANTLR** a implementačný jazyk **C#**.

3.2. Základ gramatiky

Ďalšia z otázok, ktorú bolo potrebné si zodpovedať, bola, ako by mala vyzerať gramatika jazyka MetaDPT. Táto otázka znela nasledovne: Inšpirovať sa existujúcimi jazykmi a ak áno, tak ktorými?

Prvá časť položenenej otázky má priamočiaru odpoveď. Používateľ siahne po tejto aplikácií s cieľom naučiť sa nové návrhové vzory, resp. šablóny. Chceme, aby používateľ bol schopný za čo **najmenší** čas sa zorientovať v pravidlách jazyka. Nemá zmysel využívať tento jazyk, ak sa človek musí nanovo učiť syntax. V takomto prípade by praktické využitie tohto nového jazyka bolo nízke alebo dokonca aj nulové.

Pri odpovedi na druhú časť otázky, tj. ktorými jazykmi sa inšpirovať, je potrebné sa zamyslieť hlbšie. Rovnako ako predtým, výber málo známych, resp. „mŕtvych“ jazykov, by znamenalo, že nový jazyk je nepoužiteľný pre drvivú väčšinu používateľov. Pri tomto rozhodnutí bolo dôležité, aby vybrané jazyky boli **objektové**, inak by sme návrhové vzory len zťažka implementovali. Po dôkladnom zvážení týchto kritérií padlo rozhodnutie pre dva podobné objektovo-orientované jazyky, a to jazyky Java a C#. Navzájom si sú veľkými **konkurentami**, takže

vývojári týchto jazykov sa snažili dobiehať jeden druhého, aby oba mali vydobytky moderného jazyka. Napríklad, nikdy nebolo v pláne, aby v Jave bolo generika, no z konkurenčných dôvodov ju tam museli pridať.

Táto podobnosť bola inšpiráciou, aby základná syntax jazyka MetaDPT bol **prienik** týchto dvoch jazykov. Nielenže prienik týchto jazykov umožňuje túto aplikáciu jednoducho používať Java a C# programátorom, tak tento prienik bol aj základom toho, ako môže vyzerať výstup tejto aplikácie. Používateľ si v zdrojovom kóde tohoto nového jazyka môže vybrať, či výstup má byť v jazyku Java alebo C#.

Na zváženie bola aj inklúzia ďalších jazykov, najmä jazyk C++, ktorý zdieľa myšlienky objektovo-orientovaného programovania. Na jednej strane by to zväčšilo praktickú využiteľnosť tohto programovacieho jazyka a elegantne by to doplnilo trojicu programovacích jazykov, ktorej sa venuje najviac pozornosti na odbore Informatika na Matematicko-fyzikálnej fakulte Univerzity Karlovej v Prahe. Na druhej strane **malá podobnosť** jazyka C++ s ostatnými dvoma je veľkým problémom. Stačí sa len zamyslieť nad podobnosťou narábania s objektami medzi týmito jazykmi a spätnou kompatibilitou C++ voči jazyku C. Implementovanie funkcionality pre tento jazyk by prekladač zbytočne skomplikovalo a pri tejto nadbytočnej komplexnosti by sa stratili podstatné prvky jazyka MetaDPT.

3.3. Nadstavba gramatiky

Rovnaké kritéria, použité pri výbere jazyka, bolo potrebné zvážiť aj pri rozmyslení, ako bude vyzerať **nadstavba** gramatiky, ktorá bude podporovať generovanie návrhových vzorov a šablón. Ako čo najjednoduchšie vymyslieť syntax, aby bola pre všetky návrhové vzory a šablóny jednotná a čo **najintuitívnejšia**?

Prvá možnosť, ktorá sa ponúkala, bola, aby sa návrhové vzory generovali z tých tried, ktoré by boli upresnené vo forme **prepínačov**.

```
metadpt -f Program.cs -dp Singleton
```

Prepínače sú bežná forma upresnenia chovania danej aplikácie, takže bolo potrebné zvážiť túto možnosť. Argumenty proti boli jednoznačné. Neintuitivita používania,

náročnosť rozhodovacieho stromu pri načítavaní prepínačov a zložitejšie ošetrenie chýb rozhodlo, že toto riešenie nie je ideálne.

Ďalšia možnosť bola, aby každý zdrojový kód jazyka MetaDPT obsahoval **blok**, kde by sa nejakou formou statementov definovalo generovanie návrhových vzorov a šablón. Na rozdiel od predchádzajúcej možnosti, ošetrenie chýb by za programátora vyriešil prekladač a teda táto možnosť by nebola zložitá na implementáciu, no výsledný zdrojový kód v tomto jazyku by bol na prvý pohľad rozoznateľný od jazykov prieniku.

```
pattern { Singleton(X), Builder(Y) }
```

Nápad, ktorý ošetrovanie chýb necháva na prekladač, bol dobrý. Ostávalo vymyslieť, ako rozšíriť gramatiku tak, aby dané kľúčové slová, resp. statementy, vyzerali prirodzene k jazykom Java a C#. Keďže návrhové vzory sa viažu na jednotlivé triedy, vynorila sa myšlienka, aby sa danej triede pridalo v jej hlavičke ďalšie **kľúčové slovo**. Toto kľúčové slovo vyjadruje by vyjadrovalo, že táto trieda má byť pretransformovaná na základe výberu návrhového vzoru. Napríklad pre návrhový vzor **`Singleton`** by zápis hlavičky mohol vyzerať nasledovne:

```
public singleton class A { ... }.
```

Toto riešenie je jednoduché a elegenatné, no niektorým návrhovým vzorom len táto jedna informácia nestačí, preto vyžadujú kľúčové slová aj pri niektorých položkách alebo metódach.

3.4 Generovanie výstupu

Ako ďalšiu vec bolo potrebné rozmyslieť, ako sa bude generovať výstup. Niektoré návrhové vzory pozostávajú z viac než jeden vygenerovanej triedy. Vygenerovať všetky triedy do jedného súboru by bolo možné riešenie, ale to by bolo v rozpore so **zaužívaným pravidlom**: „Jedna trieda, jeden súbor.“ Vo všeobecnosti, toto pravidlo pomáha, aby jednotlivé súbory boli prehľadnejšie a názov týchto súborov jednoducho vysvetľoval, čo daný súbor obsahuje. Na základe tohto pravidla bude mať každá trieda vlastný súbor s výstižným menom podľa toho, čo za návrhový vzor, resp. šablónu, obsahuje.

Ostáva už len vymyslieť, čo so vstupným súborom. Riešení je viacero. Mohli by sme v ňom uložiť implementáciu základnej triedy v danom jazyku, t.j. odobrali by sme nadstavbu gramatiky a pretransformovali konštrukty jedného jazyka na ten cieľový. To by ale nebolo najmúdrejšie, pretože by zakaždým užívateľ prišiel o vstupný súbor. Z tohto vyplýva, že najlepšie je vstupný súbor len čítať a pre pretransformovanú verziu tohto súboru vytvoriť **vlastný súbor**.

3.5 Výber návrhových vzorov a šablón

V momente, čo bol vymyslený základ gramatiky, bolo potrebné **vybrať** návrhové vzory a šablóny, ktoré by jazyk MetaDPT bol schopný generovať. Pri tomto rozhodnutí neišlo iba o to, ktoré návrhové vzory a šablóny sa dajú implementovať jednoducho, ale aj o to, aby daná nadstavba gramatika bola stále čo najintuitívnejšia. Na základe toho nie každý návrhový vzor zo základnej dvadssaťtrojky bol zahrnutý do funkcionality jazyka MetaDPT. V nasledujúcich podkapitolách je načrtnutý každý zo základných 23 návrhových vzorov a naviac je buď popísané, ako bol tento vzor pridaný, alebo prečo nebol.

3.5.1 Výber tvorivých návrhových vzorov

Pri tejto triede návrhových vzorov je bežné, že máme napríklad takto štrukturovanú dedičnosť: abstraktná trieda Chair, potomkovia tejto triedy WoodenChair, StoneChair, atď. Abstraktné triedy budeme nazývať **abstraktné produkty**, potomkov budeme nazývať **produkty**.

Najznámejším tvorivým návrhovým vzorom je **Singleton**. Ide o návrhový vzor, ktorý obmedzuje triedu, aby mohla existovať len jedna inštancia tejto triedy. Toto je docielené tak, že trieda obsahuje privátny field, kde je táto jedna inštancia uložená. Zvonku sa k nej pristupuje verejným getterom, ktorý pri zavolaní sa opýta, či už bola inštancia vytvorená. Ak nebola, tak ju vytvorí, uloží do privátneho fieldu a vráti. Ak bola, tak vráti hodnotu privátneho fieldu. Aby sa tento getter nedal obísť, dôležitý je privátny konštruktor tejto triedy. Týmto spôsobom sa docielila aj tzv. lazy inicializácia, pri ktorej sa daný objekt nainicializuje až keď je prvý krát potrebný v zdrojovom kóde, t.j. pri zavolaní jemu prislúchajúcemu getteru [4].

Problém s touto implementáciou nastáva, ak by k tejto triede chceli pristupovať **viaceré vlákna** naraz. Môže nastať nešťastná situácia, kedy obe vlákna prejdú podmienkou

```
if (instance == null)
```

a tým pádom by obe vlákna pracovali s rôznymi inštanciami. Z tohto dôvodu existuje upravená verzia Singletonu, ktorá tomuto problému pri viacvláknových programoch zabraňuje. Pomocou vnorenej triedy, ktorá obsahuje statickú inštanciu, sa elegantne tomuto problému vyhneme, keďže **statický konštruktor**, kde sa táto inštancie bude inicalizovať, je z nátury oboch jazykov prevedený atomicky. Dodatočne, pri tejto verzii sú všetky fieldy danej triedy pretransformované tak, aby ich gettery a settery boli tiež atomické. Túto atomickosť dosahujeme využitím **zámkov**.

Obe tieto verzie návrhového vzoru Singleton pridáme do nadstavby gramatiky jednoducho. Stačí, aby hlavička danej triedy obsahovala kľúčové slovo **`singleton`**, resp. **`ts-singleton`**.

Nasledujú dva továrenske tvorivé návrhové vzory. Prvým z nich je **FactoryMethod**. Pointou tohto návrhového vzoru je oddeliť tvorbu objektu od jeho použitia. Využíva sa v prípadoch, kedy vytvárame inštanciu triedy, no chceme, aby typ **produktu**, ktorý vytvárame, bol vybraný za behu programu formou rozhodovacieho stromu. Zároveň chceme, aby pri rozšírení programu o nový produkt sme vedeli presne, kde treba rozšíriť rozhodovací strom a aby daný strom bol oddelený od ostatnej biznis logiky. Tento problém riešime vytvorením samostatnej triedy (tzv. **továreň**), ktorá bude obsahovať jednu továrenskú metódu, do ktorej je prekopírovaný rozhodovací strom. Nakoniec na mieste, kde bol rozhodovací strom, sa doplní volanie továrenskej metódy. Tú voláme cez privátny field, v ktorom je uložená inštancia továrne. Tá sa inicializuje v konštruktore [4].

Pri tomto návrhovom vzore je nadstavba gramatiky zložitejšia, skladá sa z viacerých častí. Prvá časť je označenie produktu, ktorý chceme v továrni vytvárať, kľúčovým slovom **`product`**:

```
product Pizza pizza;
```

Druhá časť tvorí blok tvorby. V tomto bloku používateľ naimplementuje rozhodovací strom, ktorý sa bude nachádzať v továrenskej metóde. Ako parametre tohto bloku je potrebné špecifikovať všetky premenné (aj ich typy), ktoré sú dôležité pre tvorbu objektu a novej triede nebudú prístupné:

```
creation(string type) { //rozhodovací strom }
```

Následne do bezparametrového **`usage`** bloku sa vyznačí biznis logika, ktorá ostane nezmenená v danej metóde, iba sa odstráni daný blok:

```
usage { //biznis logika }
```

Druhým továrenským tvorivým návrhovým vzorom je **AbstractFactory**. Podobne ako pri predchádzajúcom vzore, oddeľuje sa tvorba objektov od ich využívania. Rozdiel je v tom, že kým pri vzore FactoryMethod bola abstrakcia len na strane produktu, v tomto prípade pridávame **abstraktnú triedu pre továreň**. Na tejto triede sa zadefinujú abstraktné metódy, ktoré budú mať za úlohy tvorbu jednotlivých produktov, s návratovou hodnotou, ktorej typ je typ daného abstraktného produktu. Následne jednotliví potomkovia tejto továrne budú overridovať abstraktné metódy a budú v nich vraciať jednotlivé produkty daného abstraktného produktu. Rozdiel medzi jednotlivými konkrétnymi továrňami je, že každá takáto továreň produkuje len jeden druh každého abstraktného produktu, tj. ak sú abstraktné produkty stôl a stolička, tak jedna továreň bude produkovať len drevené, druhá napríklad kamenné [4].

Generovanie tohto návrhového vzoru je podstatne jednoduchšie z pohľadu používateľa. Vyžaduje sa použitie dvoch kľúčových slov, **`aproduct`** a **`product`**. Prvé kľúčové slovo patrí triede abstraktnému produktu, druhé triede produktu. Aby bola zachovaná jednoduchosť a boli len dve kľúčové slová, tak je veľmi podstatné, aby triedy označené kľúčovým slovom **`product`** dedili od nejakej triedy. Ak takáto trieda nededí od žiadnej inej triedy, tak sa ako abstraktný produkt vezme prvý interface, ktorý táto trieda implementuje. Ak trieda nededí od ničoho, vyskočí výnimka. Ak príde trieda, ktorej hlavičke obsahuje kľúčové slovo **`product`**, ale jej predok ešte nebol spracovaný, tak sa táto dvojica uloží do cache, tj. poradie definovania tried nie je rozhodujúce.

Ako poslednú vec, ktorú treba zmieniť, je, ako **rozdeliť produkty medzi** jednotlivé **továrne**. Vymýšľať nejaký zložitý algoritmus, ktorý by vydedukoval z názvu produktu to, do ktorej továrne patrí, môže pripadať ako prehnané riešenie. Ak sa zamyslíme, tak už ku spomínanému príkladu so stolmi a stoličkami by existovali tieto produkty: **`StoneTable`**, **`StoneChair`**, **`WoodenTable`** a **`WoodenChair`**. Logicky do jednej továrne patria tie, ktorých prefix je rovnaký, tj. Stone, resp. Wooden. Na základe tohto sú vždy produkty usporiadané podľa abecedy a tak zaradom priradené do jednotlivých továrni. Toto riešenie je podstatne jednoduchšie, núti používateľa vymýšľať rozumné názvy tried, no nemusí to vždy vyprodukovať výstup, ktorý používateľ očakáva. Môže sa stať, že ak jednotlivé produkty nie sú pomenované správne, neskončia v správnych továrňach.

Predposledným tvorivým návrhovývm vzorom je **Protype**. Tento návrhový vzor sa opiera o vytváranie **kópií** daného objektu, tzv. klonov. Trieda, ktorá tieto klony vytvára, funguje na štýl továrne. Obsahuje metódu vracajúcu abstraktný produkt, ktorá na základe rozhodovacieho stromu tvorí jednotlivé produkty. Ďalej pre každý produkt táto továreň obsahuje field, kde má uložený tzv. **`prototyp`** (inštanciu) tohto produktu. Tvorba nového produktu sa na rozdiel od predošlých tovární nevykonáva pomocou kľúčového slova **`new`**, no pomocou abstraktnej metódy **`clone()`**, ktorá je zadefinovaná na abstraktnom produkte. Táto metóda sa volá na jednotlivých prototypoch, tj. vždy sa vytvára identický objekt k prototypu [4].

Obdobná syntax ako pri abstraktnej továrni, máme dvojicu kľúčových slov **`aprototype`** a **`prototype`**. Abstraktný produkt dostáva prvé kľúčové slovo, produkt dostáva to druhé. Pravidlá platia rovnako ako pri návrhovom vzore AbstractFactory.

Posledným tvorivým návrhovým vzorom je **Builder**. Pointou tohto návrhového vzoru je znova oddelenie tvorby objektu od jeho implementácie. Naviac, dôraz je daný na riadenie procesu výroby, ktorý sa deje **po častiach**. Tento návrhový vzor je ideálny pre triedy, ktorých fieldy vieme rozdeliť na **povinné** a **voliteľné**. Ak by rovnakú vlastnosť voliteľných fieldov chcel programátor bez tohto návrhového vzoru, tak by musel napísať konštruktor pre každú možnú podmnožinu fieldov. Ako príklad si vieme predstaviť tvorbu objektu typu Pizza. Povinné fieldy,

tj. to čo sa nachádza na každej pizzy, sú cesto a základ. Následne môže existovať veľké množstvo ingrediencií, ktoré chceme na ňu pridať. Príklad na tvorbu objektu po častiach môže byť taký, že kuchár pridá prvú ingredienciu, objekt nadobudne novú podobu a tak ďalej pri pridávaní ďalších ingrediencií [4].

Pri tomto návrhovom vzore okrem kľúčového slova **`builder`** pre triedu je potrebné aj označiť fieldy tejto triedy pomocou kľúčových slov **`builder-req`** a **`builder-opt`**. Prvým kľúčovým slovom sa označí field, ktorý je povinný v každej inštancii, tým druhým sa označí ten, ktorý je nepovinný. Následne je doplnená hlavná trieda o konštruktor s parametrom typu Builder a je vygenerovaná táto štvorica súborov tvoriaca jednotlivé časti tohto návrhového vzoru: AbstractBuilder, AbstractDirector, Builder a Director. Triedy Builder budujú daný produkt, triedy Director riadia a zapúzdrujú budovanie daného produktu.

3.5.2 Výber štrukturálnych návrhových vzorov

Ako prvý štrukturálny vzor spomenieme návrhový vzor **Facade**. Jeho pointou je zložiť viacero metód do jednej. Na príklade prekladača, potrebujeme zavolať metódy vykonávacie jednotlivé analýzy. Namiesto viacerých volaní si vytvoríme jednu metódu, potencionálne na novej triede, ktorá vykoná všetky tieto volania vrámci svojho tela a zvonku to vykonáme vrámci jedného volania (napr. metóda Compile bude v sebe obsahovať všetky jednotlivé volania pri inicalizácii prekladača). Na druhej strane je dôležité, aby všetky jednotlivé metódy boli dostupné aj samostatne, nevyžaduje sa úplné zapúzdrenie [4].

Nadstavba gramatiky je podobná ako pri vzore FactoryMethod. V danej metóde sa vytvorí takýto blok:

```
facade X(params) { .... }.
```

X bude názov fasádovej metódy, params sú čiarkami oddelené otypované premenné, ktoré sú potrebné pre statementy vo vnútri bloku, resp. pri presunutí daných statementov tieto premenné budú neprístupné. Vnútro bloku sa následne len nakopíruje do novej metódy, ktorá sa vygeneruje na konci triedy. Tu treba ešte

zdôrazniť, aby sa zbytočne nekomplikoval návrh aplikácie, v jednej metóde môže byť len jeden FactoryMethod alebo Facade.

Ďalším štrukturálnym návrhovým vzorom je návrhový vzor **Proxy**. Pri tomto návrhovom vzore máme týchto štyroch účastníkov: používateľ, objekt, rozhranie objektu a proxy. Používateľ chce pomocou rozhrania narábať s objektom. To sa ale deje tranzitívne, pretože všetky requesty prechádzajú cez proxy triedu, ktorá zdieľa rozhranie objektu. Inými slovami, proxy vytvára **bránu** medzi používateľom a reálnym objektom. Využitie tohto vzoru sú rôzne, či už ide o umožnenie používateľovi pristupovať k objektu, ktorý sa môže nachádzať v inom mennom priestore, alebo proxy brána sa dá využiť na ošetrenie prístupu. Príkladom môžu byť tzv. smart pointre, ktoré tvoria bránu medzi používateľom a blokom pamäte [4].

Triede označenej kľúčovým slovom **`proxy`** bude vytovrená nová proxy trieda, ktorá bude pozostávať z **privátneho fieldu obsahujúceho inštanciu** danej triedy. Zbytok triedy budú tvoriť rovnako sa volajúce metódy ako v zdrojovej triede, ktoré budú volať metódy tejto zdrojovej triedy cez inštanciu uloženú v privátnom fielde. Táto inštancia je v konštruktore bez parametrov, preto je potrebné, aby trieda, ktorej sa vytvára proxy, mala bezparametrový prístupný konštruktor alebo nemala žiaden.

Podobný ako proxy je návrhový vzor **Adapter**. Tiež v istom slova zmysle prepája dve triedy, no za iným účelom. Účel tohto návrhového vzoru je spojiť existujúce triedy s **nekompatibilným rozhraním**. Návrh je rovnaký, používateľ chce využívať funkcionalitu objektu, no nevie to vykonať priamo, musí ísť tranzitívne cez adaptér. Vytvorí sa rozhranie, ktoré adaptér implementuje a používateľ využíva. V tomto rozhraní sa zadefinuje forma prístupu k objektu, ktorý by bol inak používateľovi neprístupný. Medzi adaptérom a objektom platí vzťah „has-a“, tj. adaptér má ako privátny field uloženú inštanciu objektu a používateľ pomocou API adaptéru tranizívne volá API objektu. Podstatou celého návrhového vzoru je princíp dvojitej dedičnosti, čo nie je bežný jav v objektovo-orientovaných programovacích jazykoch (napr. v C++). Napriek tomu, že cieľové jazyky nepodporujú dvojitú dedičnosť tried, je v nich možná adaptácia dvoch rozhraní, no len z informácií, ktoré máme v definícií dvoch rozhraní ťažko rozmyslieť, ako vytvoriť prvému adaptér na druhý. Trieda, ktorá by len tieto

rozhrania spájala by veľa nenapovedala k funkcionalite tohto návrhového vzoru a nelíšila by sa od triedy vygenerovanej ako návrhový vzor proxy.

Pokračujeme návrhovým vzorom **Bridge**. Využíva sa pri refactoringu objektového návrhu, ktorý pozostáva z viacnásobného „is-a“ vzťahu, tj. strom dedičnosti má **hĺbku aspoň 2**. Každý takýto vzťah sa premení na „has-a“ vzťah tak, aby každý strom dedičnosti mal hĺbku maximálne 1. V niektorých prípadoch ale ide o kontext. Niekde dáva zmysel, aby strom dedičnosti hĺbky 4 bol rozdelený na štyri stromy dedičnosti o hĺbke 1, niekde na dva stromy o hĺbke 2, atď. Práve kvôli tomu nie je jednoduché zistiť, ako si to používateľ predstavuje len s použitím kľúčového slova **bridge**. Potencionálne riešenie by bolo prísť s kľúčovými slovami bridgei, kde i by reprezentovalo, že táto trieda patrí do i-teho podstromu. V tomto prípade by bola ale pádna otázka, ktoré kľúčové slovo by bolo posledné v slovníku prekladača? Bridge5? Bridge10? Hociaká odpoveď na túto otázku by bola neideálna, pretože ak sa podporuje hĺbka 10, prečo by nemala aj 11? Keďže veľkosť slovníka musí byť konečná, neexistuje rozumná implementácia tohto návrhového vzoru a preto je z nadstavby gramatiky **vynechaný** [4].

Návrhový vzor **Composite** je ďalším štrukturálnym návrhovým vzorom. Vytvára pre danú abstraktnú triedu, od ktorej dedí, alebo rozhranie, ktorý implementuje, také API, ktoré si v privátnom fielde schováva list inštancii potomkov danej abstraktnej triedy. S týmto listom sa narába verejnými metódami **`add()`** a **`remove()`**. Pointou je skryť pred používateľom fakt, že **interne** sa chová trieda ako **kolekcia** [4].

Jednoduchým pridaním kľúčového slova **`composite`** sa docieli pretransformovanie danej abstraktnej triedy alebo rozhrania. Vygeneruje sa nová trieda, ktorá bude mať daný privátny list a naviac bude overridovať, resp. implementovať, všetky potrebné metódy definované na predkovi.

Dostávame sa k ďalšiemu štrukturálnemu návrhovému vzoru s názvom **Decorator.** Na prvý pohľad podobný tvorivému návrhovému vzoru Builder, no líši sa tým, že pri vzore Builder sme tvorili po objekt po častiach, pri tomto návrhovom vzore je základny objekt tiež považovaný za nepovinnú časť a po častiach mu pridávame **dekorácie**. Napríklad, objednáme si vo fast-foode Hambuger, ktorý dedí

od triedy Meal. To je základ a dostávame na výber, či nechceme pridať nejaké ingrediencie najviac, tj. dekorácie, ktoré sú tiež potomkami abstraktnej triedy. V každom dekorovanom potomkovi nájdeme privátny field obsahujúci inštanciu nejakého potomka, ktorá sa odkazuje na predchádzajúci tvar bez momentálnej dekorácie, ktorú tvorí trieda, v ktorej sa nachádzame. Z tohto vychádza, že každá takáto trieda musí obsahovať konštruktor, ktorý ako parameter berie premennú typu abstraktnej triedy. Následne obsahuje metódu, ktorá jednotlivými volaniami tej istej metódy, ale pomocou inštancie uloženej v privátnom fielde, vráti nejaký výsledok (napr. úplná cena so všetkými pridanými ingredienciami). Týmto pseudorekurzívnym spôsobom sa pri výpočte program dostane do každej jednej dekorácie, ktorá bola pridaná [4].

Pridanie do gramatiky funguje rovnako ako pri AbstractFactory a Prototype. Zbierajú sa informácie o abstraktných triedach, ktoré majú v hlavičke kľúčové slovo **`decorator`**, ich potomkoch s kľúčovým slovom **`decoration`**. Naviac, na abstraktnej triede je dôležité vyznačiť presne jednu metódu, ktorá sa bude starať o prechod všetkými dekoráciami. Táto metóda sa vyznačí kľúčovým slovom **`decorate`**. Na základe tohto označkovania sa vytvorí stromová štruktúra, kde v dekorovacej metóde je napísané len volanie na metódu, logiku je potrebné doplniť.

Pokračujeme návrhovým vzorom **Flyweight**. Tento návrhový vzor funguje na štýl **továrne**. Znova máme abstraktný produkt a produkty, tj. potomkov abstraktného produktu. Továrenská trieda bude obsahovať **privátny slovník** (Dictionary, resp. HashMap), kde si bude priebežne ukladať produkty, ktoré už vyrobil. Vždy, keď používateľ popýta továreň o tvorbu nového objektu, tak sa najprv pozrie, či už objekt, aký vyžaduje, nemá v slovníku. Ak áno, vráti inštanciu v slovníku, ak nie, vytvorí novú a uloží ju do slovníka. Kľúče v tomto slovníku reprezentujú premennú, podľa ktorej sú vytvárané jednotlivé objekty, tj. triedy, pre ktoré je vytvorená flyweight továreň, musia obshahovať prístupný konštruktor, ktorý má ako parameter tento kľúč. Na základe tohto kľúča sa potom rozlišujú jednotlivé objekty [4].

S týmto návrhovým vzorom do gramatiky prichádza táto trojica kľúčových slov: **`flyweight`**, **`flyweightee`** a **`flykey`**. Prvým kľúčovým slovom sa označí trieda abstraktného produktu, druhým triedy produktov. Tretie kľúčové slovo slúži na označenie fieldu v abstraktnom produkte, ktorý bude tvoriť kľúč v slovníku a podľa neho sa budú rozlišovať jednotlivé objekty. Pre každý produkt označený ako **flywieghtee** bude v továrni vygenerovaná metóda **`CreateX(flykey)`**.

3.5.3 Výber behaviorálnych návrhových vzorov

Začneme návhrovým vzorom **State**. Podstatou tohto vzoru je, že trieda pracuje na základe stavov, vo väčšine prípadov ide o field obsahujúci hodonotu nejakého enumu. Tento návrh vynucuje vetviť funkcionalitu tejto triedy pomocou if, resp. switch, statementov odvíjajúcich sa od fieldu, ktorý vyjadruje momentálny **stav**. Je veľmi bežné, že vo viacerých vetvách sa opakuje kód. Druhý problém je to, že pri pridaní nového stavu je potrebné meniť hlavnú triedu, čo ideálne nechceme. Toto sa rieši dedičnosťou, v prípade enumu sa vytvorí abstraktná trieda, resp. interface, a každej hodnote tohto enumu sa vytvorí trieda, ktorá bude dediť od tej abstraktnej, resp. implementovať interface [4].

Tieto nové triedy zvyknú využívať aj privátne fieldy a metódy hlavnej triedy. V jazyku C++ sa tento problém rieši najjednoduchšie, pomocou kľúčového slova **`friend`**. V jazykoch Java a C# sa to musí riešiť pomocou **vnorenej triedy**.

Pridanie tohto vzoru do jazyka MetaDPT je **náročné**. Pri stavovej premennej, ktorá nie je enum (napr. int), je veľmi ťažké povedať, koľko nových tried vytvoriť, ako ich rozdeliť, ako ich nazvať. Pri premennej typu enum by to možné bolo, ak by sme mali informáciu o tom, aké všetky hodnoty tento enum môže nadobudnúť. Ak sa jedná o enum, ktorý je definovaný v zdrojovom súbore, tak by vygenerovanie tohto vzoru možné bolo. Problém ale je, ak sa daný enum nenachádza v zdrojovom súbore, tak prekladač nedokáže zistiť všetky možné hodnoty. Jedno možné riešenie je, aby kľúčové slovo state dostalo parametre, kde by sa vymenovali všetky hodnoty:

```
public state(A,B,C,D) MyEnum value;
```

Toto by šlo, no vo väčšine prípadov hodnoty enumov obsahujú viac písmen a aj viac hodnôt:

```
public state(Anonymous, Basic, Digest, Dpa, External,
Kerberos, Msn, Negotiate, Ntlm, Sicily) AuthType
authType;
```

Takýmto spôsobom by zápis stavovej premennej bol príliš dlhý, neprehľadný a vizuálne nevyhovujúci. Tiež by to bolo v rozpore s myšlienkou práce vytvoriť nadstavbu gramatiky čo najelegantnejšie a vizuálne najpodobnejšie cieľovým jazykom, preto je tento návrhový vzor z jazyka **vynechaný**.

Ďalším behaviorálnym návrhovým vzorom je **Memento**. Motiváciou tohto návrhového vzoru je možnosť **uložiť stav** nejakého objektu a možnosť **vrátiť** objekt do niektorého z uložených stavov [4].

Pre každú premennú, ktorú používateľ označí kľúčovým slovom **`memento`**, sa vygenerujú tieto dve metódy:

```
CreateXMemento()
```

```
SetXMemento(XMemento memento),
```

kde X je názov memento premennej. Prvá metóda vytvorí memento z momentálneho stavu danej premennej, druhá navráti stav uložený v parametri memento. Následne sa vygeneruje trieda **`XMemento`**. Táto trieda pozostáva z privátneho fieldu, kde je uložená premenná definujúca memento, a dvoch metód. Prvou metódou je **`SetState`**, ktorá ako parameter berie memento premennú, naklonuje ju a uloží si klon do svojho privátneho fieldu. Druhou metódou je bezparametreový **`GetState`**, ktorý len vráti momentálnu hodnotu uloženú v privátnom fielde.

Pokračujeme návrhovým vzorom **Iterator**. Tento návrhový vzor nám ponúka možnosť **sekvenčne prejsť** nejakým **objektom** bez toho, aby sme museli prezrádzať jej vnútornú implementáciu. V niektorých programovacích jazykoch na objekt, ktorý má naimplementovaný iterátor, je aplikovateľný foreach cyklus. Znova nie je jednoduché zistiť, ako si používateľ ten iterátor predstavuje. Čo ale prekladač vie

zvládnuť, je iterovať poľom alebo kolekciou, ktorá je uložená vo fielde danej triedy [4].

Pri generovaní tohto návrhového vzoru používateľovi stačí využiť kľúčové slová **`iterator`** a **`iterate`**. Prvým kľúčovým slovom sa označí hlavička triedy, ktorej bude vytvorený iterátor. Druhým kľúčovým slovom sa označí kolekcia alebo pole, ktoré bude iterované. V jazyku C# sa aj pole, aj kolekcia riešia jednoducho cez **`yield return`** statement. V Jave sa na kolekcii zavolá metóda **`iterator()`** a jej výsledkom sa vráti. Keďže polia túto metódu nemajú, v ich prípade sa najprv prevedú na list pomocou metódy **`Arrays.asList(...)`**.

Ďalším behaviorálnym návrhovým vzorom je **Command**. Tento návrhový vzor zapúzdruje chovanie nejakej triedy do jednej metódy **`execute()`**. Používateľ ma k dispozícii sériu jednotlivých **príkazov**, ktoré v konštruktore pošle objektu typu **Invoker**, cez ktorý bude používateľ tranzitívne volať jednotlivé zapúzdrené metódy [4].

Pre triedu, ktorá bude mať v hlavičke kľúčové slovo **`command`**, sa vygeneruje nasledovné API. Interface **`ICommand`** s jednou void metódou **`execute()`**. Ďalej, pre každú nestatickú void metódu sa vygeneruje nová trieda, ktorá bude pozostávať z 3 častí a implementovať rozhranie **`ICommand`**. Prvá časť je privátny field, ktorý obsahuje referenciu na inštanciu hlavnej triedy. Druhá časť je konštruktor, ktorý ako parameter berie inštanciu hlavnej triedy, ktorú následne uloží do už spomínaného privátneho fieldu. Tretia časť je implementácia metódy **`execute()`** z rozhrania. Jej telo je len volanie tej metódy na hlavnej triede, z ktorej je táto trieda vygenerovaná.

Dostávame sa k návrhovému vzoru **Interpreter**. Využíva sa na vyjadrenie, ako sa na základe nejakej **gramatiky** má niečo vykonať. Dobrým príkladom je jednoduchý program počítajúci výrazy v postfixovej forme. Tento program pri čítaní vstupu vytvára strom. Uzly sú operácie, listy sú čísla. Nad triedami jednotlivých častí stromu sa vytvorí rozhranie, ktoré bude obsahovať presne jednu metódu **`interpret()`**. Táto metóda bude programu napovedať, ako má daný objekt interpretovať, tj. pre uzol daného stromu reprezentujúci sčítavanie bude interpretácia

„sčítaj hodnotu ľavého a pravého podstromu“ a pre list bude interpretácia „vráť hodnotu v liste“. Celý tento program bude po načítaní vstupu spustený zavolaním metódy **`interpret()`** na koreni stromu a pred používateľom sú skryté všetky implementačné detaily [4].

Pridanie do nadstavby gramatiky zložité nie je, no veľa používateľovi o tomto vzore nenapovie. Každá trieda, ktorá má mať funkcionalitu interpretovacieho rozhrania, by sa označila kľúčovým slovom **interpreter**, transformácia by zahŕňala pridanie interpretovacieho rozhrania do hlavičky triedy a pridanie interpret metódy do tela. Čo prekladač nedokáže zistiť, je, ako sa má daná trieda interpretovať. Kvôli tomu nie je možné zistiť, akú návratovú hodnotu bude mať metóda **`interpret()`** a ani, čo bude tvoriť jej telo, preto je tento vzor z jazyka MetaDPT **vynechaný**.

Pokračujeme ďalším behaviorálnym návrhovým vzorom a ten je **Chain of Resposibility**. Majme štruktúru, v ktorej sa nachádza abstraktná trieda a jej potomkovia. Tento návrhový vzor **deleguje** zodpovednosť **medzi** jednotlivé **triedy**. Vytvorí sa abstraktný handler, v ktorom sa v protected fielde zadefinuje nasledovník, setter na tohto nasledovníka a abstraktná metóda **`Handle(X x)`**. Následne sa pre každého potomka hlavnej abstraktnej triedy vytvorí vlastný handler. V každom z týchto handlerov sa v metóde **`Handle(X x)`** spýta, či parameter x je v takej podobe, že ho má daný handler na starosti. Ak áno spracuje ho, ak nie pošle sa nasledovníkovi. Parameter sa medzi nasledovníkmi dovtedy, dokiaľ nenájde ten handler, ktorý má tento parameter na starosti [4].

Pre tento návrhový vzor sú vyčlenené kľúčové slová **`acor`** a **`cor`**. Prvé slúži pre abstraktnú triedu, druhé pre potomkov. Pre každú triedu označenú kľúčovým slovom **`cor`** sa vygeneruje vlastný handler, ktorý overriduje metódu **`handle()`**.

Prejdime na ďalší návrhový vzor s názvom **Observer**. Pridáva triede neurčitý počet **pozorovateľov**, tzv. observerov, vo vzťahu one-to-many (jedna trieda, n observerov). Títo observeri **reagujú na zmenu stavu** daného objektu. Ak táto zmena nastane, observeri sú upozornení a vykonajú svoju **`update(...)`** metódu [4].

Pri tomto návrhovom vzore sa využíva dvojica kľúčových slov, **`observer`** a **`observable`**. Prvé z týchto kľúčových slov patrí do hlavičky triedy, ktorej má byť observer API vytvorené. Druhé kľúčové slovo patrí do definície fieldov, na ktorých zmenu budú observeri reagovať.

Observer API na hlavnej triede pozostáva z privátneho fieldu a implementovania rozhrania, ktoré obsahuje tri metódy. Privátny field je kolekcia, ktorá slúži na ukladanie aktívnych observerov. Metódy **`RegisterObserver`** a **`UnregisterObserver`** slúžia na pridanie, resp. odobranie, aktívneho observeru. Poslednou metódou je **`NotifyObservers`**, ktorá informuje všetkých aktívnych observerov, že nastala zmena.

Samotná trieda jednotlivých observerov pozostáva z implementácie rozhrania, ktoré vynucuje metódu **`Update`**. Táto metóda je volaná vo vyššie spomínanej metóde **`NotifyObservers`**. Zbytok triedy tvoria privátne fieldy, ktoré reflektujú tie fieldy, ktoré boli v hlavnej triede označené ako **`observable`**.

Ďalším behaviorálnym návrhovým vzorom je **Mediator**. Pointou tohto návrhového vzoru je prepojiť triedy, aby mohli spolu komunikovať. Neideálny prístup k tomuto problému je, aby jednotlivé triedy spolu komunikovali priamo, pretože to nie je jednoducho rozšíriteľný návrh. Mediátor **prepája** všetkú **komunikáciu** medzi jednotlivými triedami a ak je potrebné rozšírenie, meníme len tento mediátor [4].

Všetkým triedam, ktoré sa majú nachádzať v mediátore, sa pridá do hlavičky kľúčové slovo **`mediator`**. Každej tejto triede sa vygeneruje privátny field, ktorý bude obsahovať referenciu na mediátor a verejný konštruktor, ktorého parametrom bude premenná typu rozhrania **`IMediator`**. Ak trieda už obsahuje iný konštruktor, je potrebné tento konštruktor z nového konštruktora zavolať. Ďalej sa vygeneruje rozhranie **`IMediator`**, ktoré bude obsahovať metódy **`GenerateX(X x)`** pre každé X, kde X je trieda označená kľúčovým slovom **`mediator`**. Nakoniec sa vygeneruje trieda **`Mediator`** implementujúca rozhranie **`IMediator`**. Ku každej metóde je vytvorený privátny field. Do tohto privátneho fieldu sa uloží parameter jednotlivých **`GenerateX`** metód. Aké metódy majú mať ostatné triedy a mediátor,

to už je na implementácii používateľa, prekladač mu vygeneruje len základnú štruktúru, viac informácií nemá.

Pokračujeme návrhovým vzorom **Strategy**. Tento rieši problém **výberu správneho algoritmu**, resp. metódy. Neideálnym riešením tohto problému je if/switch statement, kde na základe tohto rozhodovacieho stromu v jednotlivých vetvách volajú dané algoritmy. Najväčší problém je znova fakt, že ak by sme chceli pridať nový algoritmus, musíme modifikovať daný if/switch statement. To nechceme, chceme dodržiavať open-closed princíp. Tiež chceme pred používateľom skryť implementačné detaily. Strategy návrhový vzor tieto algoritmy zapúzdruje tak, aby výber algoritmu nebol závislý na používateľovi, ale vybral za behu programu na základe parametrov [4].

Tomuto návrhovému vzoru je pridelené kľúčové slovo **`strategy`**. Toto kľúčové slovo sa pridáva do hlavičky metód a tried, kde sa tieto metódy nachádzajú. Každá takáto metóda nesmie byť static. Na konci valídnej strategy metódy je vygenerované volanie metódy rozhrania **`IStrategy Execute()`**. Za telo tejto metódy sa vytvorí privátny field, kde sa uloží inštancia nejakého potomka rozhrania **`IStrategy`**, cez ktorý sa v danej metóde volá **`Execute()`**. Na konci strategy triedy je vygenerovaný verejný konštruktor, ktorý berie toľko **`IStrategy`** parametrov, koľko je strategy metód v triede, resp. koľko je privátnych strategy fieldov, ktoré v tomto konštruktore sa nainicializujú. Mimo hlavnej triedy, ak ešte sa tak nestalo, vytvorí sa rozhranie **`IStrategy`** s jednou metódou **`Execute()`**. Ako posledná vec sa vygeneruje príklad na triedu, ktorá implementuje toto rozhranie s názvom **`ExampleStrategy`**. Vo vnútri tejto triedy si môže používateľ naimplementovať logiku vlastnej stratégie.

Predposledným behaviorálnym návrhovým vzorom je **TemplateMethod**. **Definuje kostru algoritmu** ako sekvenciu volaní inštančných metód na danej abstraktnej triede. Tieto inštančné metódy sú všetky **`protected`** a delia sa na abstraktné a virtuálne. Tento návrh umožňuje potomkom spomínanej abstraktnej triedy prepisovať implementáciu jednotlivých metód podľa potreby, a pri rozšírení aplikácie o novú triedu netreba v rodičovskej triede meniť nič. Definícia kostry je naimplementovaná vo verejnej metóde **`Template()`** [4].

Pri tomto návrhovom vzore do nadstavby gramatiky prichádzajú tieto tri kľúčové slová: **`template`**, **`hook`** a **`primitive`**. Prvé kľúčové slovo patrí abstraktnej triede, ktorej bude template metóda vytvorená. Ďalšie dve kľúčové slová patria na označenie metód, ktorých volania budú obshiahnuté v template metóde. Kľúčové slovo **`hook`** slúži pre virtuálne metódy, kľúčové slovo **`primitive`** pre abstraktné. Ak je jedným z týchto dvoch kľúčových slov označená metóda, ktorá nie je **`protected virtual`**, resp. **`protected abstract`**, tak prekladač jej tieto modifikátory upraví. Dôležité je, že v akom poradí budú na template triede jednotlivé hook a primitive metódy definované, v takom poradí budú volané v template metóde. Generovanie podporuje len hook a primitive bezparametrové void metódy, no ak používateľ označí týmito kľúčovými slovami aj iné metódy, tak všetko sa prevedie, len výstupný zdrojový kód nebude možné skompilovať, používateľ bude musieť doplniť jednotlivé parametre a zachytávať return hodnoty. Čisto prekladač nedokáže zistiť presný záujem používateľa, tj. aké parametre sa majú akej metóde predať, resp. aké premenné sa majú vytvoriť a predať ako parametre.

Posledným behaviorálnym návrhovým vzorom je **Visitor**. Využíva na to, aby implementácie jednotlivých tried boli **čítateľnejšie**, znížila sa **repetitivita** a zľahčila **rozšíriteľnosť**. Toto sa docieli tak, že sa vezme trieda, jej fieldy a konštruktory sa zanechajú a všetky inštančné metódy sa nahradia jednou, a tou je:

```
accept(IVisitor visitor).
```

Táto metóda vychádza z rozhrania **`IAcceptable`**, ktoré po novom bude implementovať každá trieda, ktorá bude súčasťou visitora. **`IVisitor`** je rozhranie metódou **`visit(X x)`** preťaženou pre všetky X, kde X je trieda, ktorá je súčasťou visitora. Z tohto vyplýva, že telo metódy **`accept(..)`** bude len volanie metódy **`visitor.visit(this)`**. Všetká funkcionalita odstránená z pôvodných tried bude presunutá do triedy, ktorá bude implementovať rozhranie **`IVisitor`**. Tým pádom pri pridaní novej funkcionality meníme len túto triedu, pôvodných tried sa dotknúť nemusíme [4].

Pri pridaní tohto vzoru do jazyka MetaDPT sa vynorilo zopár ťažko zodpovedateľných otázok. Ako má vyzerať finálne verzia presunutých metód

v **`IVisitor`** triede? Majú byť spojené do seba alebo jednotlivo? Aké parametre sa majú predávať? Odpovede na tieto otázky sú príliš závislé na tom, čo chce používateľ dosiahnuť, preto bol aj tento návrhový vzor z jazyka MetaDPT **vynechaný**.

3.5.4 Výber šablón

Kvôli náročnosti implementácie niektorých návrhových vzorov sme nedosiahli číslo 23, čo je počet základných návrhových vzorov. Z toho dôvodu je jazyk doplnený o podobný koncept ako sú návrhové vzory, t.j. šablóny. Po prvé, aj návrhový vzor si vieme predstaviť ako formu nejakej šablóny, preto sa to do jazyka hodí. Po druhé, generovanie šablón vieme pridať rovnakým spôsobom do gramatiky ako sme pridali generovanie návrhových vzorov. Z toho teda vyplýva, že tri návrhové vzory, ktoré boli z jazyka MetaDPT vynechané, budú nahradené troma šablónami.

Prvou šablónou je základná implementácia **klienta** a **servera** pri komunikácii po sieti. Triede, ktorá bude označená kľúčovým slovom **`client`**, bude vygenerovaná metóda main, ktorá bude spúšťať klienta, a pomocné metódy, aby toto spustenie bolo možné. Na druhej strane, triede, ktorá bude označená kľúčovým slovom **`server`**, bude taktiež vygenerovaná metóda main, ktorá ale bude spúšťať server.

Druhou šablónou, ktorá je pridaná do jazyka MetaDPT je inicializátor pre rozhrania SAX a DOM, ktoré sú súčasťou jazyka **Java**. Rozhranie SAX rekurzívne číta vstupný **XML** súbor. Ak používateľ vytvorí triedu, ktorá dedí od triedy DefaultHandler, tak jej inštanciu bude môcť pridať do parseru XML a v nej si môže definovať vlastné chovanie pomocou overridov, tj. čo sa má diať pri prechode jednotlivými časťami XML súboru.

Pridanie do hlavičky triedy kľúčové slovo **`xmlsax`** vygeneruje inicializátor xml súboru v metóde **`parseXML(String filePath)`**. Parameter tejto metódy je cesta k zdrojovému xml súboru. Ďalej sa vygeneruje príklad na triedu, ktorá je pridaná do XML parseru.

Rozhranie DOM slúži na vykonanie zmien v XML súbore. Je mu pridelené kľúčové slovo **`xmldom`**, ktoré sa pridáva do hlavičky triedy. Následne sa danej triede vygeneruje metóda **`transformXML(String filePath, String outputFile)`**. Prvý parameter je vstupný súbor, druhý parameter je súbor, kde sa má uložiť pretransformovaný vstupný súbor. Rovnako ako pri SAX, naviac sa vygeneruje trieda **`ExampleDomTransformer`** s metódou **`transform()`**. Táto metóda sa volá v inicalizátore, čiže ju a celú triedu môže používateľ rovno použiť na svoje implementácie.

Treťou a poslednou šablónou bude čítačka **XML** pre jazyk **C#**. S kľúčovým slovom **`xmlcs`** v hlavičke triedy sa vygeneruje metóda **`ReadXML(string xmlFile)`**. V tejto metóde je nainicializovaná čítačka xml súboru.

4. Užívateľská dokumentácia

Táto kapitola slúži pre používateľa. Dozvie sa, ako sa aplikácia inštaluje a ako sa má používať.

4.1. Inštalačný manuál

Najprv je potrebné si extrahovať balík setup.zip na zariadení s operačným systémom Windows. Tento balík obshauje inštalačný program setup.exe a všetko ostatné, čo k tomu potrebuje. Po extrahovaní balíka sa aplikácia nainštaluje kliknutím na setup.exe. Postupuje sa podľa inštrukcií inštalátora. Po ukončení inštalácie sa aplikácia ihneď zapne a môže sa používať.

4.2. Návod na použitie

Po zapnutí aplikácie sa zobrazí príkazový riadok, ktorý uvíta používateľa a následne sa vypíše veta: „Type in the file path or type ‘end’ to leave“. Aplikácia čaká, aby na príkazový riadok používateľ napísal cestu k prvému vstupnému súboru. Používateľ môže cestu buď napísať ručne, alebo preniesť súbor do príkazového riadku, čo automaticky doň napíše cestu daného súboru. Po zadaní cesty používateľ potvrdí vstup tlačidlom Enter.

Vstup sa spracuje a nasleduje jeden z **možných scenárov**. Ak zadaná cesta k súboru nie je valídna, alebo nie je možné daný súbor čítať, vypíše sa veta: „Problem with opening/reading input file“ a aplikácia pýta vstupný súbor. Ak zadaný súbor je valídny, dá sa čítať, tak aplikácia pokračuje ďalej. Začne súbor spracovávať. Ak sa zistí problém s obsahom vstupného súboru, ktorý nie je fatálny, tak aplikácia len informuje o probléme a pokračuje sa ďalej, no výstup nemusí byť v tomto prípade korektný. Ak sa jedná o fatálny problém, aplikácia skončí, ak sa stihol vytvoriť nejaký výstup, tak nemusí byť korektný a používateľ je na príkazovom riadku informovaný správou: „Operation failed, message: ...“, alebo správou „Not following language guidelines, user input resulted in an error.“ Ako pri predošlom probléme, aplikácia si znova pýta nový vstupný súbor. Ak nenastal žiaden problém, výstup je korektný a používateľ je oboznámený na príkazovom riadku správou: „Operation was successful.“ Rovnako ako pri všetkých predošlých

scenároch, aplikácia pýta ďalší vstupný súbor. Ak chce používateľ ukončiť aplikáciu, napíše miesto cesty k vstupnému súboru slovo ‘end’ a aplikácia končí. Všetky úspešne vytvorené súbory sa nachádzajú v rovnakom priečinku ako vstupný súbor.

Ak sa chce používateľ vrátiť k tejto aplikácií, nájde ju pod menom MetaDPTCompiler medzi svojimi nainštalovanými aplikáciami.

4.3. Syntax jazyka MetaDPT

Základom syntaxe je prienik jazykov Java a C#. To, čo je v oboch jazykoch, a má rovnkú syntax, tak to je aj v jazyku MetaDPT. Dôležitejšie sú tie konštrukcie jazyka, ktoré sa nachádzajú v oboch z cieľových jazykov, no majú rôznu syntax.

Začnime kľúčovými slovami. Ak nastáva rozdielnosť v kľúčovom slove (C# **`namespace`**, Java **`package`**), tak je možné použiť ktorékoľvek z nich. Platí to aj pre typy ako **`String`** vs. **`string`** alebo **`int?`** vs. **`Integer`**. Rozdiel prichádza pri kľúčových slovách **`virtual`** a **`override`**. Keďže **`virtual`** je v jazyku Java implicitný a **`override`** sa píše ako anotácia, tak v syntaxe jazyka je využitá iba verzia jazyka C#, kde sa oba používajú ako kľúčové slovo v hlavičke funkcie.

Čo sa týka zápisu cyklov, jediný rozdiel je pri **`foreach`** cykle. Ako kľúčové slovo je možné použiť aj **`for`**, aj **`foreach`**, ale vo vnútri je vynútené kľúčové slovo **`in`**. Ostatné cykly, podmienky a im podobné blokové statementy sú rovnaké a ak sa v cieľovom jazyku líšia v kľúčovom slove, je možné použiť oba.

Zápis menného priestoru je prebratý z jazyka C#. Možnosť kombinácie oboch zápisov bola implementovateľná, ale gramatika by bola podstatne menej prehľadná.

Ako poslednú vec z prieniku gramatiky je potrebné spomenúť generiku a dedičnosť. V tomto prípade princíp jazyka C# nie je zahrnutý do jazyka, pretože C# pri zápise dedičnosti a generického vymedzenia nerozlišuje medzi triedami a rozhraniami a túto informáciu prekladač vyžaduje. Preto je v oboch prípadoch využitý len princíp, ktorý patrí jazyku Java.

Nadstavba syntaxe tohto prieniku tvorí target statement a nadstavba pre návrhové vzory a šablóny. Target statement je vždy prvý riadok každého

vstupného súboru, kde si používateľ vyberie cieľový jazyk výstupu. Tento statement môže vyzerať nasledovne:

```
target csharp;

target java;
```

Nadstavba syntaxe, ktorá podporuje generovanie návrhových vzorov a šablón je detailne popísaná v kapitole 3.5.

V balíku setup.zip je pribalená aj séria testovacích vstupov, ktoré môže používateľ vyskúšať.

5. Programátorská dokumentácia

Aplikácia pozostáva z viacero častí, ktoré spolu komunikujú. Tieto časti vieme rozdeliť na dva typy podľa toho, aká je ich funkcia. Buď sú tieto časti naviazané na **gramatiku** alebo na **lexikálny** (tzv. lexer), resp. **syntaktický analyzátor** (tzv. parser).

5.1. Gramatika

Gramatika je písaná v súboroch s koncovkou .g4 v **rozvinutej Backus-Naurovej forme** (EBNF). Táto forma je neusporiadaný súhrn pravidiel jazyka, ktorý ponúka vyjadrovaciu silu, ktorá pozostáva zo schopnosti vyjadrenia postupnosti derivácie (a: b c;), možnosti na výber (a: b | c;), možnosti nepovinného (ne)terminálu (a: b [c];) a možnosti opakovania daného (ne)terminálu (a: { b }) [5].

Túto formu gramatiky používa program **ANTLR** [6] ako vstup. Vývojári programu ANTLR si prispôsobili EBNF formu [5] tak, aby bola intuitívnejšia pre používateľa. Napríklad, opakovanie 0 a viac krát sa vyjadruje pomocou hviezdičky, resp. 1 a viac sa vyjadruje pomocou znaku plus, tak ako to poznáme z regulárnych výrazov. Túto myšlienku zvolili aj pri vyjadrení nepovinného (ne)terminálu a využíva sa znak otáznika.

ANTLR podporuje myšlienku dvoch súborov, ktorá je zaužívaná v nástroji lex/bison. Ísť touto cestou v ANTLR nie je povinné, ale minimálne je to prehľadnejšie a obsahovo rozčlenené. Prvý súbor, lexikálny analyzátor, je analógiou k .lex súboru (flex) a druhý súbor, syntaktický analyzátor, je analógia k .y súboru (bison). Tieto súbory sa rozlišujú zápisom na ich začiatku.

Jednotlivé terminály a neterminály v gramatike tvoria **prienik jazykov** Java a C#. Táto gramatika bola vytvorená rovnako, na základe už existujúcich ANTLR gramatík [7] a z nich vytvoreného prieniku.

5.1.1 Lexikálny analyzátor

Vrámci lexikálneho analyzátora môžeme nájsť dva typy zápisov. Ako prvý, ktorý tvorí väčšinu zápisov v lexeri, je zápis **terminálu**. Napríklad `TARGET: 'target';`. Teoreticky sa nejedná o terminál, ale ANTLR takýto zápis berie ako terminál z dôvodu, aby sa pri každom použití v pravidle nemusel písať presný text v apostrofoch.

Terminály obsiahnuté v lexikálnom analyzátore sú rozdelené do viacerých skupín. Ide o terminály kľúčových slov (pre výber jazyka, z prieniku, pre návrhové vzory a šablóny) a špeciálnych znakov.

Na **výber jazyka** sú potrebné len tri kľúčové slová. Kľúčové slovo popisujúce začiatok výberu jazyka a potom dve kľúčové slová popisujúce, ktorý jazyk si používateľ vybral, t.j.:

```
TARGET: 'target';,

CSHARP: 'csharp'; a

JAVA: 'java';.
```

Čo sa týka niektorých kľúčových slov z prienika, je potrebné dodať, že zopár kľúčových slov je v jazykoch Java a C# iných, no vyjadrujú to isté. Preto takéto neterminály majú na výber z dvoch kľúčových slov, napríklad:

```
BASE: 'base' | 'super';.
```

Inak sa jedná o klasický zápis terminálu, napríklad:

```
CLASS: 'class';.
```

Ďalšiu skupinu tvoria také kľúčové slová, ktoré sa síce nachádzajú len v jednom z jazykov, no napriek tomu sú obsiahnuté v gramatike. Napríklad kľúčové slovo `virtual` sa v Jave nevyskytuje, no jeho význam je pri metódach implicitný. Z dôvodu, že v jazyku C# je to inak a je rozdiel medzi virtuálnou a nevirtuálnou

metódou, je toto kľúčové slovo obsiahnuté v gramatike. Medzi takéto kľúčové slová patrí napríklad aj

```
EXTENDS: 'extends';.
```

Ako posledná séria kľúčových slov sú také, ktoré dodávajú jazyku funkcionalitu pre generovanie **návrhových vzorov** a **šablón**. Napríklad, pre návrhový vzor **`Singleton`** sa jedná o dvojicu kľúčových slov:

```
SINGLETON: 'singleton';

TS_SINGLETON: 'ts-singleton';,
```

kde rozdeľujeme medzi základným a thread-safe singletonom.

Medzi špeciálne znaky sú zaradené napríklad **operátory**, no všeobecne sa to dá pojať ako každý terminál, ktorý neobsahuje písmená. Aj v tomto prípade nachádzame príklad, kedy terminál **`ARROW`** sa vie zderivovať na **`->`**, resp. **`=>`** (podľa príslušného jazyka). Sem patria terminály zátvoriek, bodky, čiarky a bodkočiarky, operátory (+,-,*, atď) a skladané operátory (+=, -=, atď).

Okrem zápisu terminálov v tomto súbore nájdeme aj **fragmenty** a špeciálne pravidlá lexikálneho analyzátora. Jedná sa o produkčné pravidlá, ktorým ale ANTLR nevygeneruje Enter a Exit metódy. Pointou fragmentov je možnosť si vytvoriť pomocné pravidlá, ktoré sa potom môžu použiť v iných pravidlách lexikálneho analyzátora, v syntaktickom analyzátore ich už ANTLR nerozpozná. Napríklad:

```
fragment Digit: '0' | NonZeroDigit;,
```

kde sa toto pravidlo derivuje na nulu, resp. fragment derivujúci sa na číslice 1 až 9.

Dôvody na využitie možnosti fragmentov môžu byť rôzne, či už sa jedná o **zvýšenie prehľadnosti** parseru, alebo ak sa jedná o veľmi dlhý fragment. Dobrou analógiou je dôvod, prečo vytvárať pomocné funkcie a nepísať celý program do jednej metódy, resp. funkcie. Na základe tohto argumentu boli pravidlá a im prislúchajúce pomocné fragmenty využité na definovanie čísel a identifikátorov

(spolu tzv. literálov), ktoré gramatika akceptuje. Napríklad pravidlo lexikálneho analyzátora **`NumberLiteral`**, ktoré sa skladá zo 4 fragmentov popisujúcich zápis čísla v binárnej, osmičkovej, desiatkovej alebo šestnástkovej forme.

Tieto pravidlá sú doplnené už len pravidlami, ktoré zachytávajú komentáre, jednoriadkové aj blokové, a biele znaky. Pri blokovom komentári sú medzi blokmi akceptované všetky znaky. Čo sa týka jednoriadkového komentára, je potrebné podotknúť, ako je vidno zo zápisu, že vo vnútri takéhoto komentára sa nesmú vyskytovať znaky konca riadkov, či už pre operačný systém UNIX alebo Windows. Týmto pravilám je ale pridelená funkcia ANTLR **`-> skip`**, čo znamená, že gramatika tieto komentáre akceptuje, no v **lexikálnom** analyzátore **končia**, do parseru sa nedostanú. Na tomto príklade je elegenatne vidno ďalší dôvod, prečo definovať v lexikálnom analyzátore pravidlá, resp. fragmenty. Funkcia „skip" uľahčuje pravidlá definované v syntaktickom analyzátore, keďže nie je potrebné narábať s bielymi znakmi. Ak by sa napríklad niekto rozhodol definovať literály až v parseri, dôsledkom toho by buď v gramatike boli akceptovateľné literály obshajúce medzery, resp. konce riadkov, alebo by musel v každom pravidle explicitne zdôrazniť, kde sa biele znaky môžu a kde nemôžu vyskytovať. Koniec koncov by to vývojárovi tejto gramatiky pridalo zbytočnú prácu naviac.

Ako posledné pravidlo lexikálneho analyzátora, ktoré je dôležité explicitne spomenúť, je **`DOUBLEHASH_STATEMENT: '#' (~'#')+ '#';`**. Jedná sa o bonusovú funkcionalitu pridanú do jazyka. Hociaký statement medzi dvoma mriežkami bude 1:1 nakopírovaný na výstup (viac v 5.2.2).

5.1.2 Syntaktický analyzátor

Syntaktický analyzátor môžme rozdeliť na hlavičku, triedy, metódy, a súhrn pravidiel gramatiky.

Hlavička obsahuje viacero častí. Tam sa nachádza statement označujúci tento súbor ako syntaktický analyzátor. Ďalej v **`options`** nájdeme referenciu na lexikálny analyzátor.

Sekcia **`@parser::header`** obsahuje **importy** na knižnice, ktoré sú potrebné pre triedy a metódy definované v syntaktickom analyzátore. Táto sekcia je prázdna, v prípade tohto prekladača si vystačíme len s importami, ktoré predvolene využíva a zapíše ANTLR.

Nakoniec v sekcii **`@parser::members`** je definovaný **`enum Lang`**, ktorý v syntaktickom analyzátore určuje, ktorý je cieľový jazyk. Tento enum je dôležitý pre efektívne využitie rozhodovacieho stromu vygenerovaného automatu. Do premennej **`private Lang lang`** si pri analyzovaní pravidla **`entry`** nastavíme hodnotu reprezentujúcu typ cieľového jazyka. Túto premennú následne budeme využívať vo **`switch`** statemente, ktorý je potrebný pri ostatných metódach definovaných v syntaktickom analyzátore.

Jednou z takýchto metód je **`HandleNullableTypes`**, ktorá berie ako parametre token reprezentujúci typ premennej, token reprezentujúci otáznik, ktorý definuje **nulitu** daného primitívneho typu, a textovú hodnotu, ktorá sa má nastaviť, ak je splnená podmienka. Pointou tejto metódy je už v gramatike nastaviť primitívnym typom správny text. V gramatike jazyka MetaDPT sa na vyjadrenie nulity daného primitívneho typu využíva princíp z jazyka C#, kde sa za názov primitívneho typu napíše otáznik. Ak je cieľovým jazykom C#, tak sa zanechajú hodnoty tokenov nezmenené. Ak je ale cieľovým jazykom Java a token otáznika je nenulový, tak sa nastaví nová textová hodnota, ktorá reflektuje prístup jazyka Java k nulovateľným primitívnym typom.

Dôvod, prečo je táto metóda, a jej podobné, využitá už v súbore syntaktického analyzátora je jednoduchý. ANTLR automat už využíva **rozhodovací strom** pri výbere, ako sa pravidlo ďalej derivuje. Ak by sme tieto náležitosti riešili až v listeneroch, tak by sme sa museli znova pýtať, na aké pravidlo, resp. terminál, sa momentálne sekvencia derivuje pomocou if/switch statementov, ktoré už raz boli vykonané v kóde automatu.

Rovnako ako pri predchádzajúcej metóde, pointa metódy **`HandleArrowToken`** je rozdielnosť syntaxe tokenov. Táto metóda pomáha riešiť rozdiel v prípade terminálu, ktorý popisuje znak šípky, vyskytujúcej sa v tzv. lambda výrazoch. Java používa symbol **`->`**, C# využíva symbol **`=>`**. Logika metódy ostáva

rovnaká ako pri predchádzajúcej metóde. Nastaví textovú hodnotu parametru tokenu šípky podľa hodnoty uloženej v premennej **`lang`**. Obdobne nájdeme v telách pravidiel viacero podobných **`switch`** statementov, ktoré riešia rozdiel v syntaxi. Rozdiel medzi tým, že token šípky má vlastnú metódu a ostatné obodbné prípady nemajú, je, že tieto obdobné prípady sa vyskytujú len raz, kdežto token šípky sa vyskytuje na viacerých miestach, t.j. zaslúži si vlastnú metódu, aby sa nevyskytoval rovnaký kód na viacerých miestach.

Vstupným pravidlom gramatiky je pravidlo **`entry`**, kde sa vyberá **cieľový jazyk**. Toto pravidlo sa ďalej derivuje na typický začiatok každého súboru jazykov Java a C#, a to sú importy a názov menného priestoru. Po tejto časti prichádzajú triedy, rozhrania (interface) alebo enumy, ktoré sa ďalej derivujú klasicky podľa pravidiel prieniku jazykov Java a C#. Niektoré pravidlá je ale potrebné ešte explicitne zmieniť.

Na začiatok je potrebné zmieniť typy. Pravidlo **`type`** sa derivuje na **`simple_type`** alebo **`class_type`**. Primitívny typ sa ďalej derivuje na jednotlivé terminály s názvami primitívnych typov, ktoré potom spracováva už spomínaná metóda **`HandleNullableTypes`**. Typ reprezentujúci triedu sa nakopíruje 1:1, prekladač sa nepozerá, či takýto typ existuje v dannom súbore, resp. v naimportovaných knižniciach. Jedinú výnimku tvoria typy **`String`** a **`Object`**. Tieto dva typy musia prejsť už spomínaným switch statementom, pretože medzi jazykmi Java a C# sú rozdielnosti v zápise terminálu týchto typov.

Ďalšie pravidlá, ktoré stoja za explicitnú zmienku, sú **výrazy**. Tie sa delia na priradzovacie a nepriradzovacie. Priradzovacie výrazy sú buď unárne výrazy, t.j také výrazy vyjadrujúce aritmetické operácie, in/dekrementácie, atď., alebo unárne výrazy, ktorým predchádzajú priradzovacie operátory (=, +=, -=, ...). Nepriradzovacie výrazy zahrňujú lambda výrazy alebo sekvencie výrazov poprepájaných operátormi, tzv. cast výrazy, prístupy k položkám objektu, atď. Všetky tieto typy môžu byť ľubovoľne nakombinované do zložitejších výrazov.

Iné dôležité pravidlá zahrňujú všetky typy statementov ako sú cykly, podmienky, switch statementy, zámky, atď. Tieto typy statementov sa delia na dva typy, buď ide o **jednoriadkové** statementy, alebo o **blokové** statementy. Toto

rozdelenie je dôležité pri jednotlivých listeneroch, lebo oba tieto typy majú rozdielne prístupy k ich spracovaniu. Špeciálne statementy sú deklarácie lokálnych premenných a konštant a kopírovací statement. Tieto sa na výstup kopírujú 1:1, pri kopírovacom statemente sa naviac len odstráni mriežka na začiatku a na konci daného statementu. Ak sa napríklad v deklarácii vyskytujú terminály, resp. kľúčové slová, ktoré su rozdielne v jazykoch Java a C#, tie sú, ako bolo spomenuté, vyriešené pomocou if/switch statementov, resp. niektorou handle metódou, v telách pravidiel syntaktického analyzátora, takže sa môžu jednoducho nakopírovať 1:1.

Ako posledné by sa patrilo spomenúť pravidlá, ktoré sú používané vo viacerých častiach gramatiky. Príkladom takéhoto pravidlo môže byť **`variant_type_param_list`**, ktoré vyjadruje definovanie generiky pre triedu, rozhranie, metódu a definíciu metódy v rozhraní. Toto opätovné použitie jedného pravidla spriehľadňuje gramatiku, ale môže byť problémové pri spracovávaní tohto pravidla (viac v 4.4 Listenery). Medzi takto recyklovateľné pravidlá patria aj **`method_name_params`**, **`block`**, **`implement`** a pravidlá derivujúce sa na jednotlivé modifikátory viditeľnosti, dedičnosti, návrhových vzorov a šablón.

5.2. Main metóda a rozdelenie aplikácie

Main metóda, vstupný bod programu, obsahuje inicializáciu vstupného súboru, analyzátorov a spustenie prekladača. Pri každom novom vstupe sa tento proces opakuje. Prekladač sa spustí posledným statementom, ktorý na objekte syntaktického analyzátora zavolá jedno z jeho pravidiel, t.j. v tomto prípade sa volá pravidlo **`entry`**. Na základe tohto princípu je možné mať viacero vstupných pravidiel a len si jedno v inicializácii vybrať.

Okrem main metódy je aplikácia rozdelená na dve časti. Prvá časť sú **listenery**. Ide o triedy, ktoré dedia od **`MetaDPTParserBaseListener`** a implementuje override metódy (EnterX a ExitX metódy) na jednotlivé virtuálne metódy tejto rodičovskej triedy. Tieto metódy sa volajú pri začiatku, resp. konci, derivácie podľa pravidla X.

V aplikácii sa nachádzajú tri takéto triedy. Overridy metód sú rozdelené medzi tieto tri triedy podľa obsahu. Ak sa jedná o override, ktorý spracováva

hlavičku súboru, je zaradený do triedy **`EntryTargetListener`**. O overridy, ktoré spracovávajú konštrukty patriace pod abstraktnú triedu **`Modifiable`**(viac v 5.3.1), sa stará trieda **`ClassListener`**. Vnútro tried a metód, jednotlivé statementy, atď., sú zahrnuté v triede **`StatementListener`**. Druhú časť tvoria zapisovače, je ich dokopy 5. Navrchu je abstraktná trieda **`Writer`**, od nej dedí abstraktný **`CsharpWriter`** a **`CsharpDesignPatternWriter`**, ktorý je už sealed trieda ,analogicky pre cieľový jazyk Java.

Od spustenia prekladača medzi sebou tieto časti komunikujú, všetkú komunikáciu inicializuje vygenerovaný ANTLR automat. Ten volá **overridy** v listeneroch, ktoré následne volajú metódy zapisovačov, ak je potrebné niečo zapísať.

5.3. Konštrukty

Pri prechádzaní jednotlivých pravidiel je občas potrebné si **ukladať informácie** do pamäte. Nie každá časť zdrojového kódu môže byť ihneď vypísaná na výstup, niekde je potrebné zbierať informácie a až tak vypísať konštrukt ako celok (viac v 5.5 Zapisovače). Z tohto dôvodu bolo potrebné naimplementovať jednotlivé triedy, ktoré reprezentujú triedy, rozhrania, enumy, metódy, návrhové vzory a šablóny.

5.3.1 Abstraktná trieda Modifiable a jej potomkovia

Abstraktná trieda Modifiable je rodič všetkých tried, rozhraní, enumov, fieldov a metód. Obsahuje kolekcie, do ktorých sa ukladajú jednotlivé informácie o danom konštrukte.

Všetky tieto kolekcie sú uzavreté do generickej triedy **`Lazy`**. Ide o triedu, ktorá bráni inicializácii objektu v jej vnútri, dokiaľ nebol tento objekt explicitne referencovaný v kóde. Týmto spôsobom **šetríme** čas, ktorý je potrebný na inicializáciu jednotlivých kolekcií a priestor, ktorý jednotlivé kolekcie zaberajú po svojej inicializácii. Vďaka tejto optimalizácii môžme túto triedu využiť ako rodiča aj pre taký typ konštruktu, ktorý nevyžaduje všetky tieto kolekcie k svojmu fungovaniu. Tento prístup nám tým pádom v listeneroch dovolí uložiť všteky takéto

konštrukty do jednej kolekcie (viac 5.4 Listenery). Tie kolekcie, ktoré daný konštrukt nepotrebuje, sa nenainicializujú, bude pracovať len s tými, ktoré potrebuje. Na referenciu jednotlivých kolekcií je každej vytvorený interný getter, ktorý vracia vnútro jednotlivého **`Lazy`** objektu.

Niektoré konštrukty obsahujú naviac ďalšie kolekcie. Tieto bonusové kolekcie nie sú pridané do abstraktnej triedy ako ostatné, pretože sú využité len v jednom potomkovi. Ak by prišla zmena, ktorá by rovnakú kolekciu naimplementovala do ďalšieho potomka, potom by bolo rozumné presunúť túto kolekciu do rodičovskej triedy. Ostatní potomkovia len dedia od rodičovskej triedy a ich telá sú prázdne, navzájom ich rozlišujeme pomocou názvov týchto tried.

Ako posledný field, ktorý na tejto triede nájdeme, je:

```
internal Constraint ClassConstraint { get; set; }.
```

V tomto objekte je obsiahnutá informácia o triede

```
internal string Class { get; set; },
```

od ktorej daný **`Modifiable`** objekt dedí , resp. rozhraniach

```
private Lazy<HashSet<string>> ifaceConstraints,
```

ktoré daný **`Modifiable`** objekt implementuje. Dodatočne na tejto triede existuje metóda vracajúca bool, ktorý reprezentuje, či tento objekt je prázdny, t.j. **`Class`** je **`null`** a **`ifaceConstraints.IsValueCreated`** (property na generickej triede **`Lazy`**, ktorá hovorí, či už bol objekt vo vnútri tohto zapúzdrenia inicializovaný) je **`false`**.

5.3.2 Štruktúra dedičnosti nad abstraktnou triedou DesignPattern

Telo tejto triedy je stručné. Obsahuje int **`IndentCount`**, ktorý definuje hĺbku indentácie a dva gettery, ktoré vracajú indentáciu (obdobne ako v 5.5.1). Od tejto triedy dedia tie návrhové vzory, ktoré si s týmto pri svojom generovaní vystačia.

Niektoré návrhové vzory naviac vyžadujú **referenciu na triedu**, na ktorú je aplikovaná transformácia. Kvôli tomu, že je takýchto návrhových vzorov viac, bola vytvorená abstraktná trieda **`ClassBasedDesignPattern`**, ktorá dedí od triedy **`DesignPattern`** a pridáva konštruktor, ktorého parameter typu **`Class`** uloží do svojho fieldu **`ClassTarget`**.

Ďalšia trieda návrhových vzorov je taká, ktorá sa neviaže na triedu, ale na **statement**. Tieto návrhové vzory vyžadujú parametre. Pre takéto vzory bola vytvorená abstraktná trieda **`BlockStatementDP`**, ktorá dedí od triedy **`DesignPattern`** a pridáva list triedy **`LocalVariable`**, kde sú uložené parametre, a int, kde sa pamätá indentácia pred týmto špeciálnym statementom. Niektoré takéto návrhové vzory ale tiež vyžadujú referenciu na triedu, teda pre ne bola pridaná abstraktná trieda **`ClassBasedBlockStatementDP`**, ktorá dedí od **`BlockStatementDP`** a dodáva to isté, čo **`ClassBasedDesignPattern`**.

Poslednou dodatočnou abstraktnou triedou je trieda **`Factory`**, ktorá dedí priamo od triedy **`DesignPattern`**. Od tejto triedy dedia návrhové vzory, ktoré potrebujú zbierať informácie o dedičnosti jednotlivých tried vo vstupnom súbore. Táto trieda obsahuje dva slovníky. Do jedného si ukladá abstraktné triedy a triedy, ktoré od nich dedia. Do druhého si ukladá také triedy, ktorých abstraktné triedy ešte neboli definované, resp. budú definované až v neskoršej časti súboru. Jednotlivé triedy sa do týchto slovníkov pridávajú pomocou metódy **`AddAbstractProduct`** pre abstraktné a metódy **`AddProduct`** pre neabstraktné.

5.3.3 Abstraktná trieda Template

Obsahuje textovú hodnotu triedy, ktorej má byť vygenerovaná nejaká šablóna. Táto hodnota je inicializovaná cez konštruktor. Jednotlivé inštancie šablón sa vytvárajú v triede **`TemplateFactory`**, ktorá má na štýl návrhového vzoru **FactoryMethod** vytvorenú továreň na inštancie potomkov triedy **`Template`**.

5.4 Listenery

Na pripomenutie, jedná sa o triedy, ktoré dedia od **`MetaDPTParserBaseListener`** a obshaujú overridy na virutálne metódy

tejto rodičovskej triedy, t.j. **`EnterX`** a **`ExitX`** pre všetky pravidlá syntaktického analyzátora. Všetky potrebné overridy sú rozdelené do troch tried, **`EntryTargetListener`**, **`ClassListener`** a **`StatementListener`**. Jednotlivé listenery sú do prekladača pridávané v jeho inicializátore, t.j. v metóde **`main`**, pomocou metódy **`AddParserListener`** na triede **`MetaDPTParser`**.

5.4.1 Trieda EntryTargetListener

Táto trieda obsahuje overridy na pravidlá riešiace **cieľový jazyk**, **importy** a začiatok **menného priestoru**. Nastavenie cieľového jazyka, metóda **`ExitTarget(...)`**, vytvára inštanciu typu **`Writer (internal static Writer Writer { get; set; })`** podľa toho, aký cieľový jazyk bol vybraný. Vyberie sa správny potomok triedy Writer a na základe tohto výberu fungujú všetky zápisy, t.j. jediné miesto, kde program rozlišuje medzi jazykmi Java a C#, je táto exit metóda. O ostatné sa postará dedičnosť a overridy.

Konštruktor triedy **`Writer`** obsahuje aj textovú hodnotu názvu súboru, ktorý sa má vytvoriť a doň následne zapisovať. Do tejto hodnoty je dosadený názov vstupného súboru. Tento objekt je statický, pretože je potrebná len jedna inštancia tohto objektu. Viditeľnosť je nastavená na internú, aby mohli aj ostatné listenery referencovať tento objekt.

Čo sa týka menného priestoru a importov, volajú sa metódy na objekte typu **`Writer`**. Keďže v gramatike je výber jazyka, ak sa odmyslia biele znaky, prvý riadok zdrojového kódu jazyka MetaDPT, tak máme zaistné, že tento objekt je správne nainicializovaný.

Na konci prekladača, t.j. pri zavolaní metódy **`ExitEntry`**, kde **`entry`** je vstupné pravidlo, sa uzavrie súbor, ktorý je otvorený na objekte **`Writer`**.

5.4.2 Trieda ClassListener

Najväčší listener súbor. Rieši **Enter/Exit metódy** pre všetky pravidlá gramatiky, ktoré riešia triedy, rozhrania a enumy. V nich sa stará o jednotlivé

hlavičky a o to, čo tieto konštrukty obshaujú, okrem statementov v telách týchto konštruktov.

Zber informácií funguje nasledovne. Rovnaký princíp platí jednotne)pre rozhrania, metódy, triedy aj enumy, t.j. všetky takéto konštrukty, ktoré obsahujú hlavičku a telo je uzatvorené do množinových zátvoriek. Nastaví sa základná hodnota odsadenia do fieldu:

`internal static int IndentCount`.

Tá sa pri vnáraní zvyšuje, pri vynáraní znižuje. Dôležité je si uvedomiť, ak vstupný súbor obsahuje rozhádzanú indentáciu, tak na výstupe bude indentácia upravená. Na začiatku každého konštruktu sa prečíta pravidlo, ktoré v sebe obsahuje informácie o hlavičke. Tá sa spracuje, zapíše do súboru spolu s ľavou množinovou zátvorkou, zvýši sa indentácia, vytvorí sa objekt typu Modifiable, v ktorom sa uloží potomok podľa konštruktu, ktorý sa momentálne spracováva, a tento objekt sa uloží na zásobník:

`internal static readonly Stack<Modifiable> Constructs`.

Veľmi dôležité pri riešení vnorených konštruktov. Operácie, ktoré vyžadujeme od tejto kolekcie, sú pridaj na vrch pri vnorení do nového konštruktu a odober z vrchu, pri vynorení z konštruktu. Tieto operácie zvládne zásobník v O(1), t.j. je to ideálna dátová štruktúra. Ako bolo spomenuté v 4.3.1, máme všetky takéto konštrukty ako potomkov triedy **`Modifiable`**, teda môže ich vložiť do jednej kolekcie.

Máme na vrchu zásobníku najnovší konštrukt v zanorení, hlavička je zapísaná. Pokračuje sa pravidlom, ktoré popisuje telo daného konštruktu. Na príklade konštruktu triedy, ak príde field, ten si spracujeme a uložíme do konštruktu na vrch zásobníka. Ak príde metóda, resp. funckia, nastaví sa **`private bool MethodCtx`** na true, všetky spracované informácie sa v tomto kontexte neukladajú do objektu na vrchu zásobníka, ale do

`private Lazy<Stack<Method>> methods`,

čo je zásobník na triede **`Modifable`**, t.j. informácie sa ukladajú do metódy na vrchu tohto zásobníka, ktorý je obsiahnutý v objekte na vrchu zásobníka **`Constructs`**. Tento prístup, využívajúci zásobníky, nám dovoľuje recyklovať jednotlivé pravidlá a listenery k ním prislúchajúce, pretože spracované informácie ukladáme na vrch zásobníka, nemusíme presne vedieť, o aký konštrukt ide, pričom tie informácie, ktoré nám netreba, tak tie sa nikdy nenainicializujú zo svojej lazy formy. Následne sa vypíše hlavička s momentálnou indentáciou, zvýši sa indentácia, a telo metódy je ponechané triede **`StatementListener`**. Pri vynorení z metódy sa MethodCtx vracia na false a odstráni zo zásobníka. Rovnako pri jednotlivých konštruktoch, ak jeho telo končí, najprv sa program pozrie, či daný konštrukt treba pretransofrmovávať podľa návrhového vzoru, resp. šablóny. Potom sa zavolá zapisovacia metóda prislúchajúca konštruktu na vrchu zásobníka a celý konštrukt sa ukončí metódou

`CheckedPopAndCloseBlock(Type type)`.

Z vrchu zásobníka sa odoberie konštrukt, vykoná sa podmienka, či typ toho konštruktu zodpovedá typu predávanému ako parameter. Ten sa predáva podľa toho, z akého pravidla sa táto metóda volá. Ak je to úspešné, zníži sa indentácia a uzavrie sa blok pomocou metódy **`CloseBlock`** na objekte **`Writer`**. Tento prístup je jednotný k rozhraniam a jej metódam a aj enumom a jeho konštantám.

5.4.3 Trieda StatementListener

Táto trieda obshauje listenery, ktoré spracovávajú telá metód, t.j. jednotlivé statementy. Rozdeľujeme ich na dva typy, tzv. „one-liner“ statementy a blokové statementy.

Pri **jednoriadkových statementoch** hovoríme o mriežkovom statemente, lokálnych deklaráciách a ostatných. V týchto statementoch sa po ich spracovaní hneď vypíšu.

Pri **mriežkovom statemente** sa odoberie mriežka zo začiatku a konca tohto statementu, a jeho vnútro sa 1:1 nakopíruje na výstup pomocou metódy **`WriteCopied`**.

Lokálne deklarácie a ostatné jednoriadkové statementy sa spracovávajú rovnako, pomocou extension metódy **`GetSpacedText`** na triede **`ParserRuleContext`**. Keďže v lexikálnom analyzátore vypúšťame biele znaky, tak preddefinovaná metóda **`GetText`** na **`ParserRuleContext`**, ktorá nám vracia pospájaný text všetkých listov v momentálnom podstrome, všetko pospája a nevieme povedať, kde presne v danom statemente, resp. v danom výraze, majú byť medzery. Preto bolo potrebné definovať metódu **`GetSpacedText`**, ktorá medzerami rekurzívne pospája textové hodnoty všetkých listov, t.j. dostaneme správne oddelené tokeny v podstrome. Jednotlivé listy sa hľadajú algoritmom DFS, teda metóda správne vráti pospájané tokeny zľava doprava. Priebežne sa textové hodnoty tokenov ukladajú do listu. Tento list po jeho naplnení prejdeme a na základe hodnôt obsiahnutých v SpacedTerminals liste prerozdelíme medzery. Ak terminál vyžaduje medzery, alebo je to vizuálne lepšie, aby ich okolo seba mal, tak potom taký terminál je v tomto liste.

Blokové statementy, t.j. cykly, podmienky, zámky, atď., fungujú na podobnom princípe ako konštrukty v minulej podkapitole. Prečíta sa hlavička, tá sa vypíše s ľavou množinovou zátvorkou, zvýši sa indentácia, ďalej sa pokračuje telom. Keď skončí telo, zavolá sa metóda **`ExitBlockStatement`**, ktorá zníži indentáciu a zapíše pravú množinovú zátvorku. Na tomto princípe fungujú všetky blokové statementy. Ak napríklad nejaký cyklus neobsahuje množinové zátvorky, ale jeho telo tvorí len jeden statement, na výstupe sa už tie zátvorky objavia.

5.5. Zapisovače

Zapisovače sú rozčlenené do troch abstraktných tried a dvoch konkrétnych. Navrchu je abstraktný **`Writer`**, pod ním sú **`CsharpWriter`** a **`JavaWriter`** a pod nimi **`CsharpDesignPatternWriter`** a **`JavaDesignPatternWriter`**. Funkcionalita jednotlivých tried je rozdelená podľa obsahu. Výpisy patriace prieniku jazykov sú definované na triede **`Writer`** a overridenuté na **`CsharpWriter`**a **`JavaWriter`**. Metódy, ktoré vypisujú návrhové vzory, sú definované na triede **`Writer`** a overridenuté na triedach **`CsharpDesignPatternWriter`** a **`JavaDesignPatternWriter`**.

5.5.1 Indentácia a metóda GetNewStringBuilder

Indentácia sa rieši nezávisle na tom, ako je indentovaný vstupný súbor. Zvyšuje sa pri vnáraní do bloku, znižuje pri vynáraní z bloku. Ako už bolo spomenuté, dôsledkom toho je vo výstupnom súbore opravená indentácia oproti tomu vstupnému.

Na vyjadrenie momentálnej indentácie v hlavnom výstupnom súbore sa využíva už spomínaná property **`IndentCount`** naimplementovaná na triede **`ClassListener`**. Táto property je využívaná naprieč všetkými zapisovačmi, no nie priamo. Na textové vyjadrenie momentálnej indentácie slúžia štyri properties a jedna metóda.

Prvá property je **`Indent`**, ktorej getter vracia text pozostávajúci z toľkých znakov tabulátoru, koľko je hodnota v property **`IndentCount`**. Druhá property je **`NlIndent`**, ktorej getter vracia to isté ako **`Indent`** getter, ale ešte pred to doplní znak nového riadku (**`System.Environment.Newline`**). **`GetNewStringBuilder`** je metóda, ktorá vracia novú inštanciu triedy StringBuilder, do ktorej je dopredu pridaná momentálna textová hodnota indentácie:

```
protected StringBuilder GetNewStringBuilder() => new
        StringBuilder(Indent);.
```

Na vyjadrenie indentácie v dodatočne vygenerovaných výstupných súboroch, ktoré obsahujú návrhové vzory a šablóny, sa využívajú properties naimplementované na triede **`DesignPattern`**. Nemôžme využiť vyššie spomenuté properties, pretože v týchto súboroch začíname odznova na indentácií s hodnotou 0. Z tohto dôvodu rozlišujeme aj medzi poslednými dvoma abstraktnými properties, **`OpenBrace`** a **`DPOpenBrace`**. Prvá property vyjadruje štýl zátvorkovania v hlavnom výstupnom súbore, druhá štýl zátvorkovania v súboroch s návrhovými vzormi, resp. šablónami. Sú spomenuté v tejto časti, pretože môžu využívať **`Indent`**, resp. **`NlIndent`** vo svojich overridoch.

5.5.2 Abstraktná trieda Writer

Táto abstraktná trieda, okrem vyššie spomínaných indentačných nástrojov, obsahuje jeden field: **`private StreamWriter FileWriter;`**. Tu je v konštruktore nainicializovaný výstupný súbor, kde je pretransformovaný vstupný súbor z jazyka MetaDPT do cieľového jazyka vybraného používateľom. Metódy **`Write`** a **`WriteLine`** nad týmto fieldom sú zapúzdrené do metód **`Write`** a **`WriteLine`** na triede **`Writer`**, obohatené o **`Write`**, resp. **`WriteLine`**, na konzolu za účelom logovania.

Zbytok tejto triedy tvoria interné zapisovacie metódy a protected pomocné metódy. Názov každej zapisovacej metódy vyjadruje, aký konštrukt, resp. statement, táto metóda zapisuje. Ak daná metóda je abstraktná, to znamená, že daný konštrukt, resp. statement, má rozdielnú syntax v jazykoch Java a C# (napr. **`WriteForEach()`**). Ostatné zapisovacie metódy sú len **`void`**, pretože majú v oboch cieľových jazykoch rovnakú syntax (napr. **`WriteIf()`**).

Pomocné protected metódy slúžia na vyjadrenie konštruktov, resp. statementov, ktoré sa v generovaní výstupu opakujú na viacerých miestach. Sem patria metódy, ktoré vracajú textovú hodnotu getterov, setterov, fieldov, jednoriadkových funkcií a podobných často opakovateľných syntaktických konštruktov. Tiež sú abstraktné, keďže sa líšia svojou syntaxou v cieľových jazykoch. Vďaka tomu, že tieto časti sú naimplementované na jednotlivých konkrétnych zapisovačoch, tak môžme vytvoriť jednotné **`void`** metódy na triede Writer, ktoré budú fungovať pre oba cieľové jazyky (napr. **`WriteDP(Builder builder)`**, **`WriteDP(Singleton singleton)`**, atď.). V prípade týchto metód sa ich jednotlivé fragmenty riešia cez dedičnosť, metóda je len jedna a tá je nevirtuálna a neabstraktná.

Keďže samotná trieda bola príliš veľká, bola označená ako **`partial`** a rozdelená do dvoch súborov, **`Writer.cs`** a **`DPTWriter.cs`**. Metódy, ktoré obsahovo patria k návrhovým vzorom sú v druhom z menovaných súborov, ostatné v tom prvom.

5.5.3 Abstrakté triedy JavaWriter a CsharpWriter

Tieto triedy dedia od abstraktnej triedy **`Writer`**. Obsahujú jednotlivé overridy a naviac zopár pomocných metód. Aby bolo dodržané obsahové rozdelenie a neboli jednotlivé zapisovacie triedy príliš veľké, tieto triedy obsahujú iba overridy na metódy, ktoré riešia výpis z prieniku do cieľového jazyka (statementy typu **`WriteForeach`**, konštrukty typu **`WriteModifiable`** atď.). Ostatné metódy majú override až v triedach o úroveň nižšie.

Čo sa týka pomocných metód, ich princíp je rovnaký, nechceme písať na viac miest ten istý kód. V parametroch sa im okrem iného predáva aj objekt typu **`StringBuilder`**, do ktorého sa pridávajú jednotlivé textové hodnoty. Príkladom takejto metódy je:

`FillBuilderWithBases(StringBuilder builder, Constraint constraint)`,

ktorá do objektu typu **`StringBuilder`** pridá textovú hodnotu reprezentácie zápisu dedičnosti v danom cieľovom jazyku. Obdobne existujú metódy, ktoré naplnia objekt typu **`StringBuilder`** zápisom dedičnosti, generiky alebo parametrov.

5.5.4 Triedy JavaDesignPatternWriter a CsharpDesignPatternWriter

Tieto triedy sú potomkovia tried **`JavaWriter`**, resp. **`CsharpWriter`**, a tranzitívni potomkov triedy **`Writer`**. Obsahujú overridy na metódy a properties, ktoré obsahovo nepatria do ich priamych rodičov, t.j. generátory na syntaktické konštrukty, ktoré sa opakujú v generovaní návrhových vzorov a šablón (gettery, settery, fieldy, atď.).

Tiež obsahujú override na properties vyjadrujúce kľúčové slová. Tieto properties sú dôležité, aby mohla existovať jednotná metóda na generovanie jednotlivých návrhových vzorov a v nemuseli sa na niekoľkých miestach pýtať na to, aký máme cieľový jazyk. Toto celé rozhodovanie vďaka overridom na properties nechávame na dedičnosť.

Ostatné metódy na generovanie syntaktických konštruktov majú veľa parametrov, no rovnako nazvané parametre v rôznych funkciách spĺňajú ten istý účel. **`String indent`** je textová hodnota momentálnej indentácie, **`params string[] access`** je pole obsahujúce modifikátory viditeľnosti, `bool last` reprezentuje, či daný konštrukt je posledný v triede, t.j. nemá generovať odsadenie v novom riadku, atď. Niektoré zložitejšie metódy obsahujú aj **`int offset`**, ktorý obsahovo rozdeľuje **`params string[] access`** na dve časti. To sa používa, ak potrebujeme mať viac ako jeden modifikátor viditeľnosti pre daný konštrukt.

5.5.5 Princíp zápisu návrhového vzoru

Nový súbor obsahujúci náávrhový vzor sa vytvorí protected metódou:

`GetNewStreamWriter(string fileName)`.

Táto metóda vezme parameter a pred neho nalepí cestu k zdrojovému súboru, aby sa nové súbory vytvorili v rovnakom adresári, ako sa nachádza vstupný súbor.

Každý using blok so StreamWriter-om začína metódou:

`WriteDPNamespace(StreamWriter writer, DesignPattern pattern),`

ktorá zapíše menný priestor podľa štýlu cieľového jazyka. Ďalej v tomto bloku je vytvorená premenná typu StringBuilder, do ktorej sa priebežne ukladá implementácia návrhového vzoru. Na konci bloku sa vždy volá metóda:

`WriteDPNamespaceEnd(StreamWriter writer),`

ktorá uzavrie menný priestor podľa cieľového jazyka.

Doslov

V práci sme sa venovali problematike gramatík, prekladačov a návrhových vzorov. Vytvorili sme nový programovací jazyk MetaDPT. Gramatika pre tento jazyk spočívala v prieniku objektovo-orientovaných jazykov Java a C#. Následne túto gramatiku využíval prekladač, ktorého API sme dodali z nástroja ANTLR a implementáciu tohto API sme vykonali v jazyku C#. Tento prekladač dokázal z danej gramatiky prieniku premeniť vstupný súbor na zdrojový súbor jazyka Java alebo jazyka C#. Naviac, do nadstavby tohto prieniku sme pridali súbor terminálov a pravidiel, vďaka ktorým bonusová funkcionalita jazyka MetaDPT je generovanie 20 z 23 zákaldných návrhových vzorov pre triedy, rozhrania a metódy obsiahnuté vo vstupných súboroch. Namiesto troch, ktoré sme sa z relevantných dôvodov rozhodli do jazyka nepridať, sme pridali tri šablóny, aby sme uzavreli číslo 23.

Prínos práce nespočíva v základnej funkcionalite prekladača, ale v tej bonusovej. Programátor môže využiť tento jazyk a prekladač na rýchlu implementáciu návrhových vzorov pre svoj projekt. Na druhej strane, študent môže tento jazyk a prekladač využiť na edukačné účely pri študovaní problematiky návrhových vzorov.

Ideálnym rozšírením do budúcnosti by bola tvorba pluginu pre nejaké IDE, aby používateľ nemusel narábať s príkazovým riadkom. Namiesto toho by mohol narábať so súborom tlačidiel v nejakom IDE a ušetriť si kúsok času a roboty.

Seznam použité literatury

[1] GAMMA, Erich a kolektív. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional. 416 s. ISBN 978-0201633610.

[2] YAGHOB, Jakub. *Študijný materiál predmetu Princípy prekladačov (NSWI098) pre MFF.* Dostupné na: https://www.ksi.mff.cuni.cz/teaching/nswi098-web/#@tab_lectures.

[3] M. Vomlelová. *Automaty a gramatiky*. Dostupné na: http://ktiml.mff.cuni.cz/~marta/automatyVse.pdf.

[4] ZAVORAL, Filip a študenti. *Študijný materiál predmetu Návrhové vzory (NPRG024) pre MFF.* Dostupné na: https://www.ksi.mff.cuni.cz/teaching/nprg024-web/.

[5] PATTIS, Richard E. *EBNF: A Notation to Describe Syntax.* Dostupné na: https://www.ics.uci.edu/~pattis/misc/ebnf2.pdf.

[6] PARR, Terence a kolektív. *ANTLR.* Dostupné na: www.antlr.org.

[7] PARR, Terence a kolektív. *Grammars-v4.* Dostupné na: https://github.com/antlr/grammars-v4.

[8] AHO, A. V. a kolektív. *Time and Tape Complexity of Pushdown Automaton Languages*. Dostupné na: https://core.ac.uk/download/pdf/82622069.pdf.