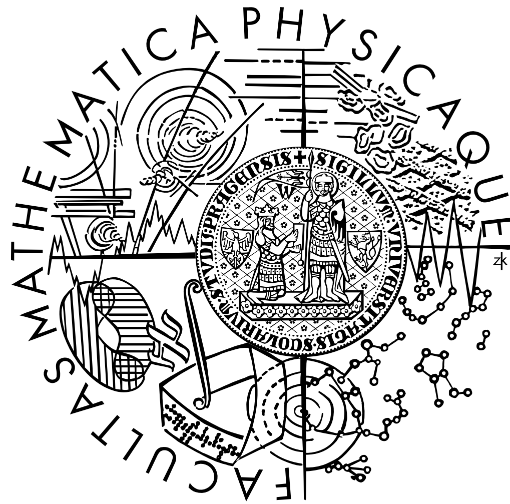
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Karel Jakubec

Ultimate Code Designer

Katedra softwarového inženýrství

Vedoucí bakalářské práce: RNDr. Jakub Yaghob, Ph.D.

Studijní program: Informatika, Programování

2009

Zde bych rád poděkoval panu RNDr. Jakubu Yaghobovi, Ph.D. za vedení této práce a za cenné rady a doporučení, které mi věnoval.

Prohlašuji, že jsem svou bakalářskou práci napsal samostatně a výhradně s použitím citovaných pramenů.

Souhlasím se zapůjčováním práce a jejím zveřejňováním.

V Praze dne: 6.8.2009 Karel Jakubec

Obsah

1 Úvod....6
1.1. Motivace....6
1.2 Osnova....7
2 Problematika generování kódu....8
2.1 Typy generátorů....8
2.1.1 Skriptované generátory....8
2.1.2 Konfigurovatelné generátory....9
2.1.3 Generátory v podobě IDE....9
2.2 Použití různých typů generátorů....10
3 Analýza Ultimate code designeru....11
3.1 Schopnosti generátoru....11
3.1.1 Jeden či více jazyků?....11
3.1.2 Nutná omezení....12
3.2 Možnosti konfigurace....13
3.3 Uživatelské rozhraní....14
3.4 Platformy....14
4 Podobné programy....15
4.1 Rational Rose RealTime....15
4.2 DialogBlocks....16
4.3 Altova UModel....16
4.4 ArgoUML....17
5 Implementace....19
5.1 Volba jazyka....19
5.2 Volba knihovny pro GUI....20
5.3 Formáty konfiguračních souborů....20
5.3.1 Formát XML....20
5.3.2 Parser....21
5.4 Rozdělení aplikace....21
5.4.1 Nezávislé části aplikace....21
5.4.2 Třída T_object a třídy z ní dědící....23
5.4.3 Třída Project a k ní se vztahující třídy....24
5.4.4 Třída Language a LanguageLoader....25

5.4.5 Třída Generator....28
5.4.6 Třída Logger....28
5.4.7 Knihovna GUI – WxWidgets....29
5.4.8 Třída Notifier....29
6 Závěr....31
6.1 Zhodnocení....31
6.2 Budoucnost programu....31
7 Reference....33
Příloha A – Uživatelská příručka....35
A 1 Obsah DVD a instalace....35
A 2 Projekt....35
A 3 Jazyky....35
A 4 Manipulace s objekty....35
Příloha B – Formát souborů jazyka....37
B 1 Stručný úvod do XML....37
B 2 Rozdělení souboru na části....37
B 2.1 Tag options a jeho podtagy....38
B 2.2 Tag types....39
B 2.3 Kořenový příkaz....40
B 2.3.1 Tag cmd....40
B 2.3.2 Tag elm....42

Název práce: Ultimate Code Designer

Autor: Karel Jakubec

Katedra: Katedra softwarového inženýrství

Vedoucí bakalářské práce: RNDr. Jakub Yaghob, Ph.D.

E-mail vedoucího: yaghob@ksi.ms.mff.cuni.cz

Abstrakt: Tato práce popisuje vývoj aplikace, která umožňuje navrhnout strukturu a kostru programu v grafickém rozhraní. Tato kostra je následně převedena do skutečného objektově orientovaného programovacího jazyka a vygenerována pro pozdější zkompilování.

V práci jsou probrány různé typy generátorů a rozebrán způsob vytváření a formát souborů popisujících objektový jazyk. Jsou zde popisována rozhodnutí, která bylo nutno učinit během analýzy problému a následného vývoje, vč. volby programovacího jazyka a použitých knihoven.

Klíčová slova: Generování kódu, Objektové jazyky, XML, GUI, wxWidgets

Title: Ultimate Code Designer

Author: Karel Jakubec

Department: Katedra softwarového inženýrství

Supervisor: RNDr. Jakub Yaghob, Ph.D.

Supervisor's e-mail address: yaghob@ksi.ms.mff.cuni.cz

Abstract: This thessis descibes design and implementation of the application, which can design structure of program and then generate this structure to the source code of some real object-oriented language.

Various types of code generators are discussed in this work as well as design and format of files, which describe these leanguages, are examined. Work goes through desicions, that had to be made during analysis of the problem and subsequent implementation, including choice of programming language and libraries used in the application.

Keywords: Code generation, Object languages, XML, GUI, wxWidgets

1 Úvod

1.1. Motivace

Tvorba každého jen trochu rozsáhlejšího projektu se skládá z řady kroků. Jako první, přeskočíme-li složité úvahy, zda se dílo vůbec vyplatí, by měl přijít na řadu obecný návrh programu, ještě v ideálním případě nezávislý na konkrétním jazyku, ale už poměrně jasně dělící projekt na nezávislé části a definující rozhraní mezi nimi. Poté následuje volba konkrétního jazyka a začíná samotná implementace neboli psaní kódu. Poslední v řadě, alespoň co se programátora týče, je provedeno ladění a testování programu.

Toto je samozřejmě ideální pořadí, v praxi se často první dva kroky prohodí. Důvod je poměrně zřejmý – návrh se dělá snáze, ví-li programátor jaký jazyk bude použit, protože může rovnou využít všech možných vlastností onoho jazyka (typickým příkladem je vícenásobná dědičnost). Výsledkem nemusí být nutně paskvil, ale může se stát, že budou přehlédnuty lepší alternativy v podobě ostatních programovacích jazyků – uvažování návrhářů je v tuto chvíli omezeno pouze na jazyk předem zvolený. Na druhou stranu, tento postup je poměrně pochopitelný v případě, že firma zaměstnává skupinu programátorů, kteří ovládají pouze a jen jeden jazyk a tudíž alternativy nepřicházejí v úvahu. Návrh projektu lze samozřejmě též úplně vynechat. Zde naopak je typickým výsledkem paskvil – nepřehledný, mnohokrát přepisovaný a špatně udržovatelný kód.

Třetí fáze (psaní kódu) naopak vynechána být nemůže a, ač to není správné, mnohokrát to je právě ona, která zabere během tvorby projektu nejvíce času. Mezi programátory je také dosti neoblíbená, protože se neřeší žádné zajímavé problémy (neplatí, vynechá-li se návrh, v tomto případě bude zajímavých problémů více než dost) a práce je redukována na mechanické ťukání do klávesnice a vzniká zde poměrně mnoho chyb (překlepy, nesprávná implementace algoritmů, nepochopení kódu jiného člena týmu apod.).

Zjednodušit psaní samotné dnes už příliš nejde. Většina lepších editorů obsahuje technologie umožňující doplňovat do textu identifikátory, kontrolovat typy, hlídat správné použití dereference apod. Editor je schopen analyzovat kód, pozná co je třída, jaké má metody a zda se je programátor nesnaží použít v rozporu s pravidly jazyka.

Z těchto poznatků je vidět, že všechny fáze by měly být co nejjednodušší. Programátor by měl v ideálním případě provést návrh, ve kterém vyřeší strojově jen

obtížně řešitelné úlohy. Není však velký problém nechat vygenerovat samotný kód, čímž odstraníme problémy s překlepy a ušetříme cenný programátorský čas.

Proto jsem se rozhodl napsat program, který by snad měl být schopen pomoci programátorům s návrhem jejich programů a měl by být schopen vygenerovat alespoň základní kostru programu. Tím by se ušetřila monotóní práce při psaní složitých struktur.

1.2 Osnova

V následující kapitole je diskutována obecně problematika generování dat z jiných dat. Je třeba si uvědomit, co vše je ve skutečnosti generátor kódu a jak můžeme generátory dělit a pohlížet na ně.

Kapitola 3 se zabývá analýzou a návrhem Ultimate Code Designeru. Jsou zde probírány otázky, které si bylo nutno položit ještě před napsáním první řádky zdrojového kódu - především jak by vůbec UCD měl vypadat, jaké jazyky podporovat, jakým způsobem provádět konfiguraci.

V kapitole 4 bych rád poukázal na již existující programy podobného typu. Všechny se zabývají generováním zdrojového kódu, z kterého následně vznikne aplikace, ale každá trochu po svém a rád bych zde zmínil jejich přednosti a zápory.

Nejobsáhlejší kapitolou je kapitola číslo 5. Zde je popsána implementace problému a částečně jsou zhodnocena rozhodnutí, která padla při analýze systému.

Závěr práce se nachází v kapitole 6, kde je provedeno především shrnutí ostatních předchozích kapitol a zhodnocení práce. Dále je nastíněn budoucí vývoj projektu.

Ve dvou přílohách je ve stručnosti popsáno nejprve ovládání aplikace a poté postup tvorby vlastního souboru jazyka.

2 Problematika generování kódu

Snaha ušetřit si práci je stará jako lidstvo samo, a proto se i programátoři snaží co nejvíce práce automatizovat. Ne vždy je to samozřejmě možné, ale v mnoha případech ano. Důkazem je dnes celá řada generátorů, které považujeme za tak samozřejmé, že o nich z tohoto hlediska vůbec neuvažujeme jako o generátorech.

Dobrým příkladem jsou konfigurační soubory pro testování programu. Často jsou velmi rozsáhlé a upravovat je řádek po řádku trvá zbytečně dlouho. Za generátor je zde možno považovat i jednoduchý regulární výraz aplikovaný pomocí sedu na soubor čísel, který z nich rázem vytvoří seznam IP adres čitelný třeba DNS serverem.

2.1 Typy generátorů

Generátory lze, pro lepší orientaci, rozdělit podle své složitosti a typického použití na několik základních typů:

- generátory v podobě malých skriptů
- generátory konfigurovatelné, které neumí upravovat již jednou vygenerovaný kód
- kompletní IDE, spravující obvykle celé projekty, která se umí vyrovnat s vnějšími zásahy do již vygenerovaného kódu

Netřeba asi dodávat, že ne každý generátor se podaří přesně zaškatulkovat. Tyto skupiny mají sloužit spíše pro orientaci v problematice a jako ukázkové příklady, co vše je a může být považováno za generátor kódu.

2.1.1 Skriptované generátory

Trochu jsem váhal, zda tyto programátorské počiny vůbec označovat za generátory, ale hlavní kritérium (tj. že ze zadaných dat vytvoří jiný soubor dat, typicky o mnoho rozsáhlejší, než byl původní soubor) splňují a jsou mocnými pomocníky leckterého programátora.

Jejich hlavními znaky jsou:

- jednoduchost – napsání je velice rychlé
- jednoúčelovost – jsou použity pro jeden konkrétní účel, formát a podoba

výsledného souboru dat je dána kódem generátoru

- malá konfigurovatelnost – konfigurace většinou argumenty předávanými na příkazové řádce, popřípadě vůbec žádná (je nutno zásah přímo do kódu generátoru)
- absence jakéhokoli grafického rozhraní
- minimální kontrola zdrojových dat – uživatel a programátor je často ta samá osoba, popřípadě kolegové z týmu a tím pádem se předpokládá schopnost uhlídat si správný formát dat

Skriptované je nazývám z toho důvodu, že pro jejich implementaci je většinou použit nějaký z pestré škály skriptovacích jazyků (v unixovém prostředí např. bash). Je možno samozřejmě použít některý z klasických programovacích jazyků, ale pak dochází k problémům s úpravami generátoru vzhledem k nutnosti kompilace.

2.1.2 Konfigurovatelné generátory

Od předešlých se liší především tím, že v drtivé většině případů jde už o plnohodnotné programy. Z toho jasně plyne nutnost konfigurovat program jinak než pouhým zásahem do jeho kódu, protože ten často není uživateli k dispozici.

Hlavní znaky:

- podpora konfigurace z externího souboru či přes GUI
- složitější formát vstupních dat (např. XML soubor, či nějaký jednoduchý jazyk)
- propracovanější uživatelské rozhraní – často interaktivní na příkazové řádce nebo přímo GUI

Generátory tohoto typu nejsou schopny upravit již jednou vygenerovaný kód jinak než kompletním přegenerováním – prostě výsledný soubor dat přepíší.

2.1.3 Generátory v podobě IDE

Tato skupina stojí na pomyslném žebříčku generátorů nejvýše. Zásadním

rozdílem oproti konfigurovatelným generátorům je schopnost upravovat již vygenerovaný kód a taktéž se vyrovnat se změnami v něm zvenčí. Z toho vyplývá, že program musí být schopen alespoň minimální analýzy již vygenerovaných dat, aby rozpoznal případné úpravy.

Typické znaky:

- GUI – ovládání z příkazové řádky by již bylo nepohodlné
- rozumná možnost konfigurace generátoru, většinou právě přes GUI
- schopnost alespoň částečné zpětné analýzy vygenerovaného kódu

2.2 Použití různých typů generátorů

Jak se od sebe liší jednotlivé generátory, tak se liší i jejich použití.

První skupina zcela jasně najde uplatnění především jako pomocné prográmky během vývoje většího projektu. Usnadňují testování, ale málokdy se (naštěstí) stávají produkčním kódem, který slouží běžným uživatelům. Jejich snadné vytváření je bohužel kompenzováno řádově nižší rychlostí provádění (danou obecnou rychlostí provádění skriptovaných jazyků oproti jazykům kompilovaným), takže pro větší projekty se nehodí.

Druhá skupina už netrpí neduhy, co se rychlosti týče (samozřejmě jen v případě, že generátor je alespoň trochu slušně napsán), ale je limitována neschopností upravit vygenerovaný kód jinak než přepsáním. Dobře se použijí pro vytvořené kostry programu, kam následně dopíšeme ještě nějaký kód.

Generátory v podobě IDE představují přesně to, co jejich název říká - integrované vývojové prostředí. Často nejenže vygenerují kód, ale v případě, že jde o kompilovatelný jazyk, ho i rovnou zkompilují, umí zobrazit výstup kompilátoru, popřípadě obsahují i debugger. Mají pod kontrolou celý projekt, tudíž jsou (v ideálním případě) jediným potřebným vývojovým nástrojem. To je ale i jejich výrazným omezením - vygenerovaný text může být pro člověka takřka nečitelný a velice nepřehledný a tudíž je víceméně nemožné do něj zasáhnout jinak než přes toto IDE.

3 Analýza Ultimate code designeru

Poté, co jsem se definitivně rozhodl napsat generátor zdrojového kódu jako bakalářskou práci, jsem si musel položit několik zásadních otázek. Odpovědi na ně budou definovat výsledný produkt a proto se musely pečlivě zvážit.

1. Co vše bude generátor schopen generovat?
2. Jak se bude provádět konfigurace?
3. Jak by asi mělo vypadat uživatelské rozhraní?
4. Na jaké platformy se zaměřit?

3.1 Schopnosti generátoru

Troufnu si tvrdit, že toto byla otázka zcela nejzásadnější. Jak vyplývá z kapitoly 2, generovat lze skutečně skoro cokoli, a proto je potřeba se omezit na nějakou rozumně velkou podmnožinu. Vycházel jsem z názoru, že by to měla být podmnožina taková, jaká ještě nebyla žádným jiným podobným programem pokryta. To se po krátkém průzkumu ukázalo jako takřka neřešitelné, takže jsem se začal soustředit na méně ambiciózní cíl a to napsat generátor, který by mohl někomu usnadnit práci.

3.1.1 Jeden či více jazyků?

Od začátku jsem chtěl vytvářet kód nějakého programovacího jazyka. Hlavním problémem bylo jakého, zda pouze jednoho a zda celý program, či jen jakousi kostru programu.

Co se jazyka týče, určitě by to měl být jazyk procedurální, nejsem si dodnes schopen představit, jak vyrobit něco takového třeba pro funkcionální jazyky, a asi i objektově orientovaný - všechny nejpoužívanější jazyky dneška nějakým způsobem podporují třídy a objekty.

S myšlenkou jednoho jednoho jazyka jsem si pohrával poměrně dlouho a nakonec jsem ji zavrhl. Po krátké úvaze jsem dospěl k názoru, že pouze jeden jazyk znám tak dobře, abych byl schopen napsat jeho skutečně dobrý generátor bez jakýchkoli omezení a to C/C++. Je pravdou, že generátor pro čistě jeden jazyk má celou řadu výhod (možnost soustředit se plně na jeden jazyk, tudíž jeho plná podpora, generování může být

"zadrátováno" přímo v kódu, potenciálně potřeba načítat o jeden konfigurační soubor méně, rychlejší generování), ale bohužel i omezení (jen ten jeden konkrétní jazyk, v případě "zadrátovaného řešení" obtížné úpravy a v případě načítání syntaxe jazyka ze souboru už není velký problém podpořit i další jazyky) a ta nakonec převážila.

Proto zvítězil návrh, dle kterého by měli být rozumně podporovány dnešní procedurální objektově orientované jazyky. Hlavní důraz je kladen na C/C++, ale i ostatní jazyky (Java, C#, Pascal/Object Pascal, PHP,...) by měly jít implementovat bez výraznějších omezení. Velikou výhodou tohoto řešení je i možnost použít UCD jako benchmark implementovaných jazyků. Program se navrhne pouze jednou a UCD podle zadaných parametrů vytvoří zdrojové soubory pro mnoho jazyků, které se následně mohou testovat.

3.1.2 Nutná omezení

Ač jsi jsou dnešní objektově orientované jazyky velice podobné, stále nalezneme řadu aspektů, ve kterých se liší. Z těchto důvodů je možné podpořit pouze takové vlastnosti, které jsou široce rozšířeny.

První a asi nejviditelnější takovou vlastností je vícenásobná dědičnosti. Původně jsem uvažoval ji do UCD vůbec nezahrnout. Jsou jazyky, které ji nepodporují vůbec a, pokud je mi známo, vždy ji lze nějak ekvivalentně nahradit. Na druhou stranu to však není ošklivá vlastnost, umožňuje pomocí abstraktních tříd velice hezky definovat potřebná rozhraní a skládat tak větší třídy postupně z menších, což může zpřehlednit a zkrátit program. Proto nakonec dostalo přednost řešení známe z Javy – třída (class) může dědit jednu třídu (class) a neomezeně rozhraní (interface). Pro velkou část jazyků to je řešení přirozené a u těch, co mezi třídou a rozhraním nerozlišují, to nebude vadit.

Dalším sporným bodem jsou lokální deklarace typů (struktur, tříd nebo i funkcí). V tomto případě je opět problém s rozdílnou podporou napříč jazyky a též bych narážel na potíže s vizualizací tohoto stavu v GUI. Lokální deklaraci by vždy mělo jít korektně nahradit deklarací globální a z těchto důvodů budu používat používat pouze typy globální.

S předchozím rozhodnutím padla i možnost generovat výsledný kód tak, aby se na něj už nemuselo sáhnout. Automaticky generovat jde jen to, co mají všechny jazyky podobné a to jsou základní deklarace a definice struktur, tříd a funkcí. Vnitřní kód metod se bohužel může velice lišit (některé jazyky například neznají ukazatele, příkazy break a continue mají odlišný význam apod.), a proto není reálné ho přímo generovat. Bude ale

možné ho pro každý jazyk a metodu ručně dopsat.

3.2 Možnosti konfigurace

Má-li UCD podporovat více jazyků, jediná rozumná možnost, jak reprezentovat jazyk, je pomocí konfiguračního souboru. Výsledný formát souboru a jeho syntaxe je popsána v dalších kapitolách, zde bude probrány pouze vysokoúrovňový návrh souboru.

Obecné předpoklady, pro úspěšné generování kódu:

- Není možné předpokládat, že existuje nějaká přesně daná struktura programu společná pro velkou většinu jazyků. Proto pořadí, jak po sobě půjdou jednotlivé části kódu (deklarace tříd, funkcí,...), nesmí být určeno napevno, nýbrž pořadím, jak budou po sobě uvedeny v souboru jazyka.

- Množina významů klíčových slov (klíčové slovo pro konstantní symbol, abstraktní metodu, deklaraci třídy,...) společných pro procedurální jazyky není příliš velká, proto nebude problém v UCD takovou množinu vytvořit. Je však nutné na tuto množinu namapovat skutečná klíčová slova jazyků, protože ta se u jednotlivých jazyků liší (C++/Java - const/final, C++/Object Pascal - class/object). Bude tudíž třeba mít v souboru jazyka část popisující toto mapování.

- Stejný problém nastává u názvů základních typů. V klasickém programování se neobejdeme bez 4 typů - typ pro celá a desetinná čísla, booleovský typ a řetězec. Na ně se namapují skutečné typy jazyka. Bude pouze na programátorovi, zda v případě C++ namapuje na řetězec char*, char[256] nebo rovnou std::string. Ale taktéž si programátor musí být schopen vytvořit vlastní typ, aby mohl zároveň použít char* a std::string. V jiném jazyku, který toto nerozlišuje, na oba namapuje jeden skutečný typ (pascal - string).

- Soubor jazyka by měl být psán tak volně, jak to jen půjde a zakazovat by se měly pouze zcela nesmyslné konstrukce. Tím bude umožněno rozšířit na maximum množinu generovatelných jazyků (např. o některé skriptovací jazyky, byť pro jejich implementaci se neobejdeme bez jistých "workaroundů").

3.3 Uživatelské rozhraní

Uživatelské rozhraní má být přehledné, jednoduché, intuitivní. Určitě grafické, jednotlivé objekty by se měly zobrazovat v okně jako malé krabičky a mělo by jít s nimi pohybovat. Dědičnost by měla být zřejmá na první pohled. Po straně by se mohl nacházet seznam všech objektů.

3.4 Platformy

Mou výchozí platformou jsou už po delší dobu Microsoft Windows, s WinAPI mám jisté zkušenosti z několika menších předchozích projektů. Proto jsem za cílovou platformu určil právě MS Windows. Multiplatformity se však úplně vzdát nechci, minimálně Linux bych rád také v budoucnu podpořil, čemuž odpovídá i volba knihoven a jazyka v následující kapitole. UCD však bude alespoň v první fázi testované na MS Windows.

4 Podobné programy

Během návrhu systému jsem samozřejmě čerpal inspiraci z již existujících programů. Sám jsem byl kolikrát překvapen, co vše je vlastně pouze generátor zdrojového kódu.

4.1 Rational Rose RealTime

Rational Rose Realtime[1] je program od firmy Rational, v dnešní době už patřící pod IBM[2]. Je zmíněn jako první, protože to byl on (nebo spíše jeho povedené grafické rozhraní), kdo mě inspiroval k této práci. Ze zde srovnávaných programů jde o zdaleka nejpokročilejší generátor.

Práce s RR vypadá následovně. Programátor si v poměrně přehledném grafickém prostředí vytvoří návrh automatu. Základním elementem automatu je tzv. kapsule, která v klasickém pojetí automatu přibližně reprezentuje stav. Kapsule může obsahovat další kapsule, přechody mezi nimi, rozhodovací body a porty, s kterými může komunikovat s dalšími kapsulemi na úplně jiných úrovních. Typicky se aplikace rozdělí na několik na sobě co nejméně závislých částí, každé se přiřadí jedna "top-level" kapsule a ty pak spolu už komunikují právě prostřednictvím zpráv zasílaných porty.

Kód vykonávaný v kapsulích nebo v přechodech mezi nimi píše uživatel sám - GUI zde slouží jako především vizualizátor toku programu. Je v něm krásně vidět, co se stane, když na ten či onen port přijde zpráva, zda se rovnou změní stav nebo se nejdříve program dostane do rozhodovacího bodu a pak teprve dojde k nějaké akci.

Malým problémem je praktická nemožnost ruční editace již vygenerovaného kódu, který je absolutně nepřehledný - RR převede návrh do "svého" systému šablonovaných tříd, ve kterém je přinejmenším velice obtížné se vyznat. Též je problematická správa souborů projektu v nějakém verzovacím systému - některé nejsou textové a jsou tam pochopitelně uloženy souřadnice objektů na obrazovce, což vyvolává časté konflikty a soubory se musí neustále ručně mergovat (což je poněkud složité v případě binárních souborů). Řešením je buď důsledné zamykání přístupu na úrovni verzovacího systému nebo zákaz pohybu s objekty (kapsulemi atd.). Obě řešení bohužel zpomalují práci. IBM vyvinulo vlastní verzovací systém ClearCase[3], který by měl tyto problémy řešit (zná formát projektových souborů a tím pádem je umí mergovat), ale ten má daleko k jednoduchosti a kompaktnosti nejpoužívanějších SCM, jako je CVS, SVN či

GIT.

4.2 DialogBlocks

Na DialogBlocks[4] jsem narazil víceméně náhodou, když jsem se poohlížel po nějakém grafickém editoru uživatelského rozhraní pro knihovnu wxWidgets. S jejich pomocí si je možno „naklikat“ aplikaci a ta bude následně vygenerována – buď jako C++ kód nebo XRC kód pro další použití ve vlastním kódu. Výhodou je možnost vytvořit si takto třeba jen jeden dialog, který lze následně zařadit do svého projektu.

DialogBlocks pracují podobně jako všechny asi všechny podobné editory GUI – vytvoří zkompilovatelný kód, pro korektní obsluhu GUI, avšak samotné akce, které mají nastat například po stisknutí tlačítka si musí programátor dopsat sám. Pro tento účel obsahují DialogBlocks jednoduchý a přehledný textový editor.

Pokud bych měl DialogBlocks zařadit do některé z kategorií, patřily by někam mezi 2. a 3. kategorii. Umožňují správu celého vašeho projektu, lze přímo v nich napsat smysluplnou aplikaci a je možno ji i zkompilovat (avšak kompilátor není součástí programu, musí se dodat zvenčí). Pro větší projekty je však lepší použít sofistikovanější editory, protože textový editor je zde velice jednoduchý a z funkcí usnadňujících psaní podporuje pouze zvýraznění syntaxe, což je na úrovni lepšího MS Notepadu. Velice příjemnou funkcí je generování komentářů čitelných Doxygenem pro pozdější snadnou tvorbu dokumentace.

Jako velice dobrá se mi osvědčila kombinace DialogBlocks a MS Visual Studia, kde v DialogBlocks si ve WYSIWYG editoru připravím a vygeneruji rozhraní a ve Visual Studiu ho následně upravím a doplním o vlastní metody. Vygenerovaný kód je velice přehledný a není problém se v něm orientovat.

4.3 Altova UModel

Aplikace UModel[5] od společnosti Altova je původně součástí mnohem většího balíčku aplikací pro vývoj software (MissionKit) od stejnojmenné firmy, jehož cílem je poskytnout podporu vývojářům takřka ve všech směrech - od snadné tvorby DTD pro XML dokumenty, vytváření aplikací pomocí UML rozhraní, automatické generování

dokumentace a složitých SQL dotazů až po aplikace pro mergovaní a diffování kódu, usnadňující správu zdrojových souborů. MissionKit se dodává ve dvou verzích, Enterprise a Profesional, Enterprise je na dobu 30 dní pro vyzkoušení zdarma.

UModel je ze zde srovnávaných generátorů asi nejbližším příbuzným UCD. Jeho cílem je návrh programu jako takového pomocí UML diagramů a následné vygenerování kódu do jednoho ze tří podporovaných jazyků (Java, C#, VB.NET).

UModel je, vzhledem k tomu, že se jedná o nástroj pro profesionální použití při vývoji software, o mnoho dokonalejší než UCD. Podporuje sice jen tři jazyky, ale zato plně, bez nutnosti psát si konfigurační soubory. Velikou výhodou je schopnost zařadit do projektu již napsané části zdrojového kódu, popřípadě namergovat projekt do již existujích zdrojových souborů. Je schopen spolupracovat s verzovacími systémy.

Bohužel jsem neměl možnost vyzkoušet UModel na větším projektu, kde by se asi nejvíce projevila výhoda v podobě grafického návrhu. Na menších projektech funguje dobře, ale nebyla zde vidět větší časová úspora, než kdyby byl kód psán klasickou formou v editoru.

4.4 ArgoUML

ArgoUML[6] je volně šiřitelná aplikace pro generování kódu z UML diagramů.

V tomto ohledu jde o pokročilejší generátor, než jakým si UCD klade za cíl být. Podporuje pomocí modulů široké spektrum jazyků (hned po stažení je k dipozici celkem 5 – C#, C++, Java, PHP4, PHP5) a je dokonce schopen zpětně parsovat zdrojové kódy Javy.

Rozhraní je poměrně přehledné. Využil jsem ArgoUML k několika menším projektům (max pár stovek řádků) jako generátor kostry programu. Těžko srovnávat, zda jsem dosáhl nějaké zásadní časové úspory, to by se opět projevilo spíše na větším díle. Hezkou funkcí je možnost exportovat grafiku do formátu .PNG. Díky tomu používám Argo pro kreslení schematických diagramů (mimo jiné i pro tuto prací), kde se ukázal být často o mnoho použitelnějším, než leckteré grafické editory.

Bohužel celkový dojem trochu kazí zdánlivě banální nedodělky a chyby, které však v celkovém součtu dokáží značně znepříjemnit práci. Chybí funkce undo – nebyl by to tak zásadní nedostatek, nebýt dalších chyb, díky kterým by se undo hodilo. Stejně tak chybí možnost udělat copy-paste. To je naopak, u jinak profesionálně vyhlížejícího nástroje, velikou vadou na kráse. Není možné objekty označit a změnit všem jednu

vlastnost (např. není možné označit soustavu linek a všem změnit barvu z modré na zelenou, musí se to dělat postupně). To mi přijde trochu nepochopitelné, protože naopak je možné označit více objektů a pohybovat s nimi naráz. Poslední chybou je zvláštní chování nástrojové lišty, která si občas pamatuje, který nástroj byl zvolen a naposledy používán, občas ne a občas i přes to, že byl zvolen již jiný nástroj, nechá nastaven starý (právě kvůli těmto nedostatkům bych velice přivítal možnost undo). Pokud je v tom nějaký systém, tak jsem na něj za více jak 4 měsíce používání nepřišel. Argo je též paměťově poměrně náročná aplikace. Ač po spuštění zabírá přijatelných 30MB, tak i přo krátkém kreslení digramů se dostane mnohdy nad 100MB.

I přes tyto chyby je ArgoUML použitelným nástrojem usnadňujícím vývojářům práci. Je na něm zatím bohužel příliš vidět, že jde o nekomerční produkt, který většinou potřebuje delší čas, aby „vyzrál".

5 Implementace

V této kapitole jsou popsány implementační detaily UCD. Zpočátku je zdůvodněna volba jazyka, knihoven a použitých technologií, následně rozdělení programů na nezávislé části a jejich vnitřní struktura.

5.1 Volba jazyka

Tato volba byla docela lehká, zvažoval jsem tři kandidáty:

- C++, jazyk ve kterém jsem začínal programovat, napsal jsem v něm několik menších projektů a který je hlavním programovacím jazykem v mém zaměstnání.
- C#, podle mnohých velice perspektivní jazyk, pro Windows dnes asi nejsnáze použitelný co se grafického rozhraní týče (.NET 3.0 a XAML), částečně ho ovládám.
- Java, multiplatformní, ale pro mě dosti neznámý jazyk.

Java vypadla jako první. Z těchto tří je asi "nejvíce multiplatformní", což je významný argument pro. Avšak ovládám ji pouze pasivně a to ještě často musím nahlížet do referenční příručky. Asi bych neměl zásadní problém během jednoho měsíce Javu nastudovat na úroveň nutnou pro napsání UCD, je to stejně jako C++ objektově orientovaný procedurální jazyk, ale v případě takto důležitého projektu jsem se bál riskovat.

U C# by studování bylo méně, napsal jsem v něm pár jednoduchých aplikací. Pod Windows to je dnes velice silný nástroj, GUI by se v něm psalo asi nejlépe ze všech zmiňovaných kandidátů. Jenže v této době stále pouze pod Windows a trochu mě též odradilo prostudování [7].

Volba padla tím pádem na C++. Hlavními argumenty pro byly mé několikaleté zkušenosti s programováním v něm. O multiplatformitě C++ bylo již mnoho napsáno[8] a řečeno, pokud se program píše rozumně, tak to lze, i když se asi nepůjde ubránit občasným ifdef konstrukcím.

5.2 Volba knihovny pro GUI

První myšlenka - napsat GUI přímo ve WinAPI - mi byla naštěstí rozmluvena mým vedoucím. Výsledkem by byl asi ne příliš úhledný kód, nemluvě o nutnosti psát si vlastní obsluhu dialogů. A také, když už jako jazyk bylo zvoleno C++, částečně pro svou relativně snadnou zkompilovatelnost pod Linuxem, byla by škoda o tuto možnost přijít použitím čistého WinAPI.

Nastalo tedy hledání vhodnějšího nástroje. Vzhledem k plánovanému vývoji v Microsoft Visual Studiu jsem zvažoval MFC. VS obsahuje hezký editor dialogů, jenž by práci hodně ulehčil. Jenže MFC je opět záležitostí pouze Windows a nechce se mi používat technologii, o níž se až příliš často vedou diskuze, zda již není mrtvá[9][10].

Nakonec jsem narazil na wxWidgets, knihovnu pro psaní aplikací s klasickým GUI jak pod Windows, tak pod *nixovými systémy[11]. Použití knihovny není nikterak složité a dokonce jsem nalezl velice hezký editor dialogů DialogBlocks[4] od společnosti Anthemion Software. Nabízí se v několika verzích, základní neregistrovaná je zdarma a pro vygenerování základní konstrukce programu a několika dialogů postačuje, složitější objekty je nutno si napsat svépomocí nebo zaplatit za lepší verzi.

Podrobnější popis implementace bude popsán v kapitole 5.

5.3 Formáty konfiguračních souborů

UCD bude potřebovat několik konfiguračních souborů - jednak pro vlastní nastavení programu, pak pro jednotlivé jazyky a pro ukládání rozdělaných projektů. Každý z těchto souborů musí mít nějaký formát, v ideálním případě všechny stejný, aby se zjednodušilo načítání.

Vytvářet si vlastní formát a psát pro něj parser mi přišlo jako vynalézat znovu kolo, a proto jsem se rozhodl pro dnes velice populární a široce rozšířený formát XML.

5.3.1 Formát XML

XML neboli Extensible Markup Language [12] je značkovací jazyk, který má velice široké využití. Obecně slouží k formátování a strukturalizaci dat.

Základním prvkem dokumentu je element, který může obsahovat další elementy nebo text. Elementu je možno přiřadit různé atributy, upřesňující jeho význam. Celý

dokument má stromový charakter [13] a právě jeden element je kořenem.

5.3.2 Parser

Obecně existují dva typy parserů :

- DOM (Document Object Model) - převede dokument do paměti počítače a uživatel/programátor si jej pak projde pomocí vlastních procedur.
- SAX (Simple API for XML) - během průchodu dokumentem volá uživatelský kód.

Není v popisu této práce rozebírat zde přednosti a zápory obou přístupů k dokumentu, ve stručnosti DOM je spíše pomalejší a paměťově náročnější, ale umožňuje přímý přístup k datům, SAX je rychlý a paměťově nenáročný, ale zato se hodí spíše pro sekvenční čtení celého dokumentu.

Pro účely UCD by bylo možno použít oba přístupy (jakékoli konfigurační soubory se vždy čtou celé), já jsem nakonec zvolil DOM parser od profesora Frank Vanden Berghena[14]. Ostatní parsery (XERXES[15], eXpat[16]) toho sice umí o mnoho více, ale v UCD je třeba pouze načítat a ukládat data v XML formátu, na což tento botahě postačuje. Jeho obrovskou výhodou je jednoduchost a malá velikost - skládá se ze dvou zdrojových souborů, které dohromady mají jen málo přes 100KB.

5.4 Rozdělení aplikace

5.4.1 Nezávislé části aplikace

Toto byl asi první problém, který jsem řešil při návrhu samotné struktury aplikace - jak moc dělit aplikaci na nezávislé moduly[17]. Čím více modulů, tím lepší zapouzdření vnitřní implementace, ale zase je nutno definovat více rozhraní (ač jsou moduly nezávislé, musí spolu nějak komunikovat).

Po delším rozmýšlení jsem dospěl k rozdělení na:

- objekty, které uchovávají data projektu – třída T_object a třídy z ní dědící
- správa projektu - třída Project

- jazyky – třída Language a LanguageLoader
- generátor - třída Generator
- načítání konfiguračních souborů - třídy ProjectLoader, LanguageLoader a ProjectSaver
- obsluha chybových hlášení - třída Logger
- grafické rozhraní, GUI – okna, dialogy, vykreslování
- rozhraní mezi GUI a správou projektu s generátorem - třída Notifier

Takto vypadal původní návrh na papíře. Hlavním cílem bylo oddělit co nejvíce GUI pro případ, že bych se v budoucnu někdy rozhodl použít jiné grafické rozhraní. To se však ukázalo jako poměrně složité, speciálně pro vykreslování grafické reprezentace objektů - musela by se předávat víceméně veškerá data objektu, aby bylo vůbec co vykreslit. Z toho důvodu má každý objekt medotu Draw, která kreslí přímo do DeviceContextu.

Správa projektu a generátor jsou logicky sice nezávislé časti, ale generátor musí používat projektem definované rozhraní, aby získal potřebná data, takže nezávislost je pouze jednostranná. Načítání konfigurace je též poměrně poměrně úzce svázané s projektem, takže opět jde spíše o jednostrannou závislost, ale v tomto případě z druhé strany.

Chybová hlášení reprezentována třídou Logger jsou skutečně samostatná, pomineme-li nutnost volat wxMessageBox při jistých událostech.

Asi největším problémem bylo, jak to celé svázat dohromady v jeden funkční celek. GUI bude muset volat metody ostatních tříd a oznamovat jim uživatelovy akce, aby na ně mohly adekvátně reagovat. Opačně též tyto třídy budou muset nějak sdělit GUI, že někde nastala změna (byl načten soubor apod.). První volba by byla psát volání těchto metod přímo do kódu, což by však nebylo příliš hezké vzhledem k výše uvedeným prohlášením. Proto jsem vytvořil třídu Notifier, jež slouží jako rozhraní mezi GUI a zbytkem programu - obě části povětšinou volají metody této třídy místo toho, aby se volaly navzájem. Tímto se efektivně předá informace a případná změna v kódu se projeví pouze v Notifieru a (většinou) ne dále.

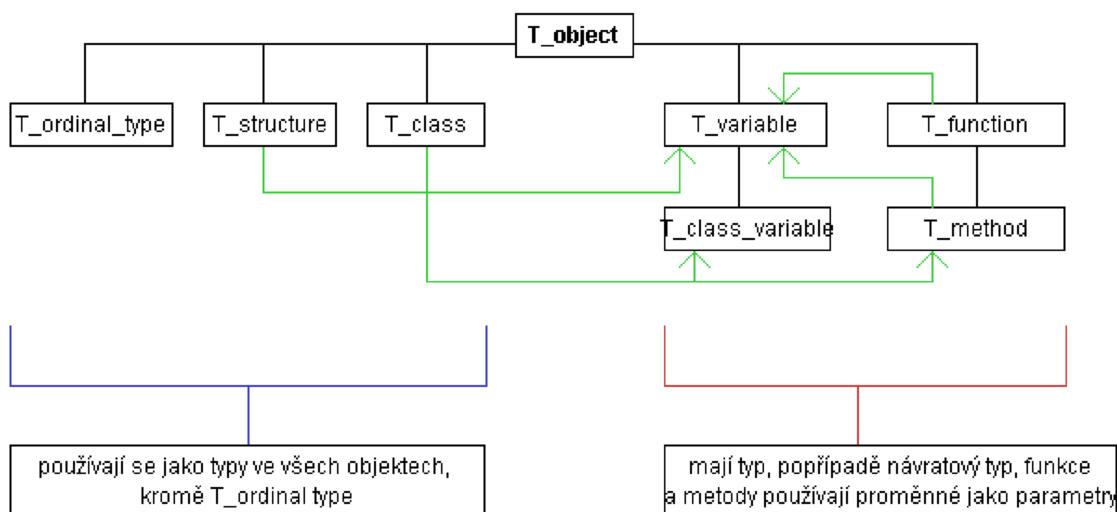
5.4.2 Třída T_object a třídy z ní dědící

Třída T_object je abstraktní třídou, reprezentující jeden objekt projektu. Za objekty projektu jsou zde považovány ordinální typy (T_ordinal_type), výčtové typy nebo-li struktury (T_structure), třídy (T_class), proměnné (T_variable) a funkce (T_function). Poslední dva jsou ještě dále rozvíjeny pro použití ve třídě a to do T_class_variable a T_method (obsahují navíc položku AccesLevel, která definuje možný přístup).

Ordinální a výčtové typy společně se třídami tvoří typy projektu. Každé proměnné a každému parametru funkce musí být přiřazen jeden z těchto typů a každá funkce může jeden z těchto typů vracet. Struktury a třídy dále obsahují své vnitřní proměnné a metody.

Je-li objekt nějakého typu, je možno vždy zvolit, zda je ukazatelem, odkazem či hodnotou. Stejně tak lze určit i návratový typ u funkce.

Následující obrázek stručně shrnuje tuto podkapitolu. Černé čáry zobrazují dědičnost a zelené šipky dávají na vědomí, že objekt využívá jako svou vnitřní proměnou jiný objekt.



5.4.3 Třída Project a k ní se vztahující třídy

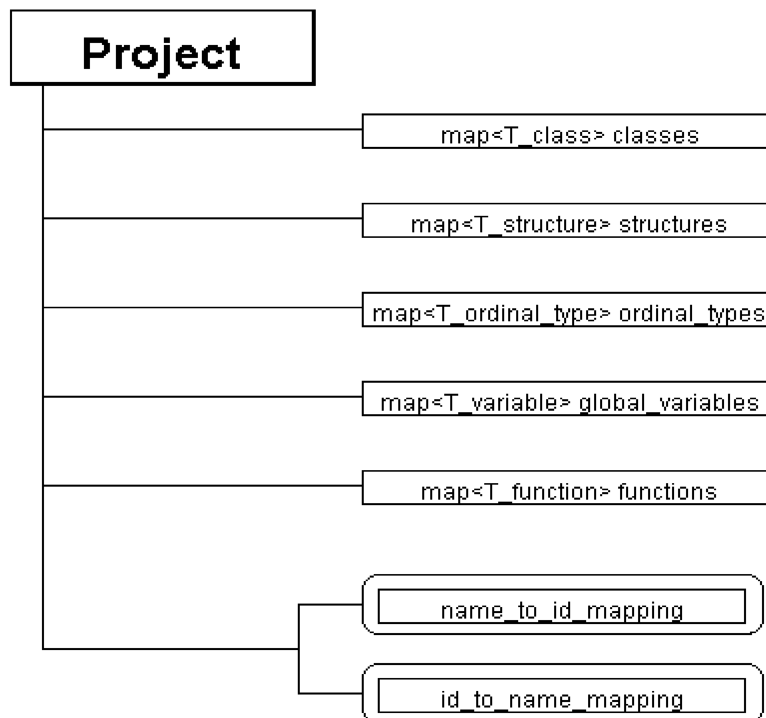
Třída Project reprezentuje, jak je již vidět z názvu, jeden projekt. Projekt je v tomto smyslu chápan jako soubor objektů, ze kterých vznikne výsledný kód. Hlavním úkolem třídy je tedy držet tato data v paměti a provádět v nich potřebné změny.

Pro uchovávání objektů je použito asociativních polí – umožňují rychlý přístup k objektu přes jeho jméno. Avšak bylo by zbytečně pomalé používat všude k získání odkazu na objekt pouze jeho jméno, proto si Project ještě uchovává ke každému objektu ID, které je jedinečné v celém projektu. Tato ID jsou držena ve dvou asociativních polích (ID -> jméno, jméno -> ID), aby byl možný rychlý převod na obě strany. Pro zkrácení a zpřehlednění kódu byly vytvořeny funkce get_name_from_id(...) a get_id_from_name(...), které zajišťují tyto konverze v celém kódu i v případě, že není možné přijatelným způsobem získat odkaz na aktivní projekt. Samotný objekt o ID neví, je to čistě informace uložená v oněch dvou polích, takže máme-li pouze odkaz na objekt, ID nezískáme. Toto řešení může vypadat zvláštně, ale při bližším pohledu je poměrně logické. ID má totiž smysl pouze v případě objektu globálního, tedy existujícího na úrovni projektu. Je hezké, aby funkce měla své ID, protože pak s ní jde zacházet podobně jako s kterýmkoli jiným globálním objektem. Na druhou stranu, v případě metody je ID nadbytečnou záležitostí, protože metoda je unikátní pouze a jen v působnosti třídy, a proto ID není třeba a bohatě postačí identifikace jménem. (Stejně tak globální proměnná a proměnná ve struktuře). Z těchto důvodů ID není přímou součástí žádného objektu, ale pouze postranní informací.

Projekt by nebyl příliš použitelným, pokud by nebylo možno ho uložit do souboru a následně opět načíst. K tomuto účelu slouží třídy ProjectLoader a ProjectSaver. Jsou to třídy typu worker, neuchovávají takřka žádná data, jejich typické použití vypadá tak, že se vytvoří na zásobníku jedna jejich instance, ta odvede svou práci a je zničena.

Soubory mají příponu .udp (ultimate designer project) a jsou, jako všechny konfigurační soubory UCD, ve formátu XML. V principu jde o seznam objektů a všech jejich vlastností. Formát XML je zde ideální, protože jeho stromovitá struktura[13] je takřka totožná se strukturou projektu (projekt má několik tříd, které mají několik proměnných a metod, které mají několik parametrů....).

Strukturu projektu ukazuje obrázek na konci podkapitoly.



5.4.4 Třída Language a LanguageLoader

Instance třídy Language definuje jeden z jazyků, do kterých je možno vygenerovat projekt. Jazyk je uložen v souboru s příponou .udl (ultimate designer language), je taktéž ve formátu XML. Struktura souboru je podrobněji popsána v příloze B.

Ač to může vypadat nezvykle, jazyky nejsou součástí projektu, ale samostatnou částí programu, která se načítá hned po startu. Bylo by zbytečné načítat jazyky znovu s každým novým projektem

Třída Language má podobnou strukturu jako třída Project, z čehož plyne, že má za úkol především uchovávat data o jazyce a způsobu, jak převést projekt do výsledného kódu. Zde by šlo rozdělit Language na dvě logické části.

První částí jsou specifikace klíčových slov jazyka a dalších podobných útvarů (jak vypadá komentář, zda jazyk rozlišuje mezi interface a class apod.). Zde jsou také vyjmenovány základní typy jazyka, na které je možno namapovat ordinální typy projektu.

Druhá část popisuje, jak generovat kód. Její struktura je podobná souboru

popsanému v příloze B, takže zde bude rozebrána spíše z hlediska návrhu a v příloze budou dopodrobna popsány všechny možné atributy.

Dnešní objektové jazyky patří mezi jazyky rozpoznávané Turingovým strojem a pokud bych je chtěl přesně popsat, byla by k tomu potřeba bezkontextová gramatika. Na druhou stranu, pro potřeby UCD není nutné popsat celý jazyk, stačí popsat, jak jsou zapsány jeho deklarační a definiční konstrukce. Proto bylo nutné vytvořit strukturu, která toto umožní. Taková struktura musí splňovat následující:

- umožnit jednoduché větvení – aby s některými objekty stejného typu mohlo být zacházeno trochu jinak (třeba v případě C++ nechceme generovat hlavičku funkce main())
- pořadí zápisu jednotlivých prvků je určující pro výsledný kód (viz kapitola 3.2.)
- stromovitá struktura – umožní snadnou definici zápisu vnořených objektů, jako např. metod u tříd a následně jejich parametrů
- rozumná velikost – soubor s jazykem by měl mít maximálně pár set řádků
- minimálně omezovat, ctít poučku GIGO[18], zakazovat jen zcela nesmyslné konstrukce – hack sice není hezké řešení, ale stále to je řešení a někdy se může hodit

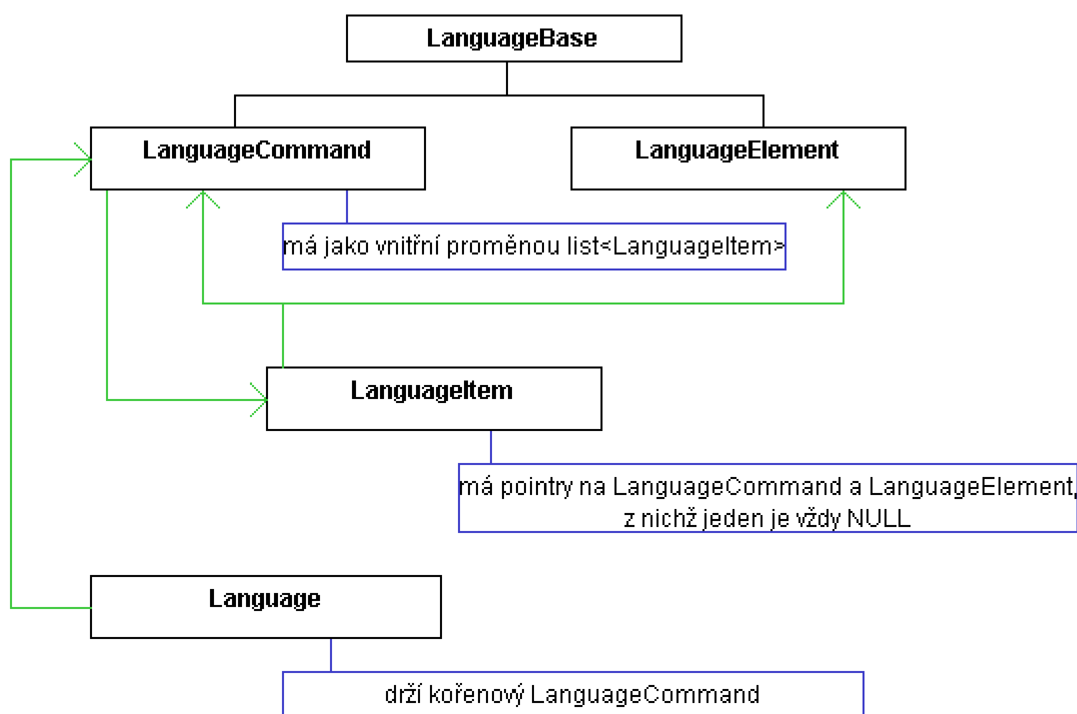
Proto jsem strukturu rozdělil do dvou větví – příkazy a elementy.

Příkaz negeneruje žádný kód, pouze určuje, co se má udělat s jeho vnitřkem. Rozhodující je jeho typ (viz příloha B). Například příkaz functions dává najevo, že jeho vnitřek se provede pro všechny funkce (s tím, že je možno z toho některé konkrétní funkce vyloučit). Každý příkaz má v sobě seznam podpříkazů a elementů. Existují dva speciální příkazy a to „root“ a „file“. Root je kořenový (top-level) příkaz a file určuje, jakou příponu bude mít soubor, do kterého se kód vygeneruje.

Element je ve stromě vždy listem. Element může být celkem třech typů – text (elm_text), proměnná (elm_var) a návěstí (elm_sign). Text je jednoduše řetězec, který se vypíše do souboru. Je to typicky buď komentář, který uvozuje nějakou část programu, nebo kus kódu, který se na daném místě vyskytuje vždy, bez ohledu na cokoli jiného. Proměnná je řetězec, vztahující se ke zrovna generovanému objektu – jeho jméno, typ apod. Její význam je tedy částečně určen příkazem, který je o úroveň výše než ona sama a

její obsah není předem znám (není specifikován v souboru jazyka ale přímo v projektu). Třetím typem jsou návěstí, která slouží především jako indikátory vlastností objektů (zda je objekt konstantní, zda to je ukazatel či reference apod.). Jsou v tomto podobná proměnným, ale liší se v tom, že jejich obsah je vypsán pouze v případě, splňuje-li objekt danou vlastnost a tento obsah je také vždy totožný (je to typicky nějaké klíčové slovo, popřípadě znak).

V programu je toto celé implementováno jako soustava celkem pěti tříd. LanguageBase je předek LanguageElement a LanguageCommand. LanguageItem zastřešuje tyto dvě třídy pro snadnější použití – drží v sobě ukazatel na obě dvě z čehož jeden je vždy NULL. Lze říci, že LangugeItem je buď příkaz nebo element. Každý příkaz má jako vnitřní proměnou list<LanguageItem>, což jsou jeho podpříkazy. Element nic takového nevlastní, protože se vždy nachází v listu. Třída Language pak má pouze jako proměnou kořenový příkaz, který obsahuje všechny ostatní jako své podpříkazy. Obrázek na další straně ukazuje celou tuto strukturu.



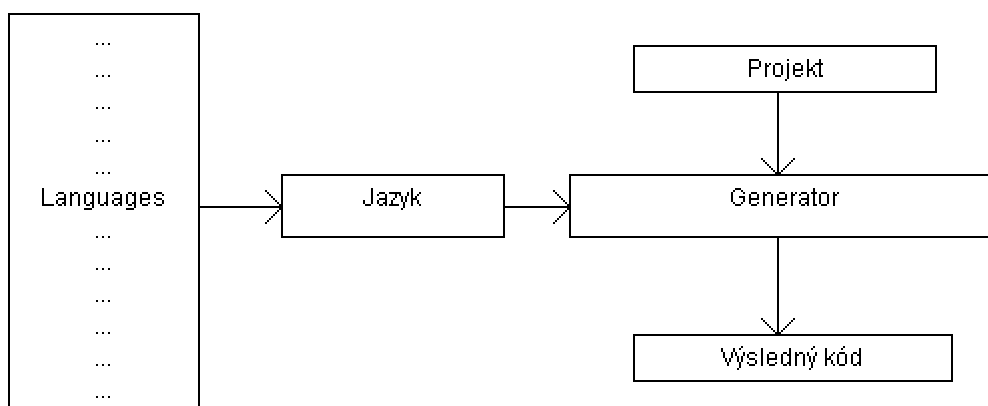
Pro nahrávání souborů do paměti byla vytvořena třída LanguageLoader. Její struktura je podobná jako v případě třídy ProjectLoader a používá i stejný XML Parser.

Může být překvapivé, že neexistuje žádný LanguageSaver – editor jazyků není součástí programu, takže za LanguageSaver jde považovat víceméně jakýkoli textový editor.

5.4.5 Třída Generator

Třída Generator obstarává generování výsledného kódu. Tato část, ač je pravděpodobně nejpodstatnější z celého projektu, není nejsložitější a třída nemá nikterak komplikovanou strukturu ani složité metody.

Má jen jednu podstatnou veřejnou metodu a tou je go(). Ta, jak je z názvu asi jasné, spustí celý proces generování kódu. Zde jde už v podstatě pouze o průchod stromem jazyka zleva doprava. Generátor zjistí, co má generovat (zda text, proměnou,...) a podle toho si od projektu vyžádá potřebná data. Průchod je kvůli tomuto částečně rekurzivní, protože je třeba si pamatovat část cesty nahoru, aby se vědělo, zda proměnná NAME je např. jméno třídy nebo metody.



5.4.6 Třída Logger

Logger je pomocnou třídou a zde bude zmíněn jen krátce, protože na funkčnost programu nemá zásadní vliv. Jeho úkolem je informovat uživatele o důležitých událostech a případných chybách.

Informace mohou být vypisovány do souboru (pokud je tato funkce zapnuta) a vypisovány na obrazovku. Existuje celkem 5 úrovní (trace, info, warning, error, fatal_error), to které se vypíší na obrazovku určuje konstanta LOG_POP_UP (její obsah je stanoven při kompilaci). Za šestou úroveň může být brána notifikace, která se vždy vypíše na obrazovku, ale nikoli do souboru.

5.4.7 Knihovna GUI – WxWidgets

WxWidgets je multiplatformní volně šiřitelná knihovna pro tvorbu aplikací s grafickým uživatelským rozhraním. Důvody, proč byla zvolena pro UCD, jsou rozebrány v kapitole 3.

Jako většina podobných knihoven, i WxWidgets má svou strukturu programu, která se trochu liší od klasického C/C++ a aplikace ji musí z větší části respektovat. Zde to spočívá v nutnosti mít třídu, která dědí z wxApp a ve které je pak program přes soustavu maker spuštěn. Podrobnější popis je k nalezení na webu[19].

Hlavní okno je reprezentováno třídou mainWindow. V něm se nacházejí 2 podokna – levé menu a pravý panel, na kterém se zobrazují jednotlivé objekty.

Důležitou částí rozhraní jsou dialogy, které zajišťují interakci s uživatelem. S jejich realizací mi částečně pomohl program Anthemion DialogBlocks, jehož přednosti a zápory jsou podrobněji rozebírány v kapitole 3. Dialog je ve své podstatě třída, která dědí ze třídy wxDialog. Pro snazší práci s již hotovými dialogy byla vytvořena třída Dialogs, která vykreslí dialog uživateli (buď prázdný nebo vyplněný údaji o objektu).

Rozhraní by nebylo úplné, pokud by objekty nebyly uživateli nějakým způsobem vizualizovány, a to tak, aby snadno mohl vidět vztahy mezi nimi. Zde bylo bohužel nutno částečně porušit zásadu o oddělení knihovního kódu wxWidgets a samotného jádra aplikace. Každý objekt má veřejnou metodu Draw, která ho vykreslí na obrazovku na v něm uložené souřadnice. Objekt je schopen určit, zda byl „zasažen“ kliknutím (metodo is_hit, které se předají souřadnice myši). Bylo by technicky realizovatelné zajišťovat tyto operace i externě, ale kopírovalo by se zbytečně velké množství dat (všechna zobrazovaná data).

5.4.8 Třída Notifier

Jak je již psáno v kapitole 5.4.1, bylo třeba najít způsob, jak co nejpružněji spojit kód aplikace a WxWidgets. Informace zde tečou celkem 2 směry – z GUI do projektu a opačně. První směr je reakce na uživatelské akce v GUI, druhý nastává především při načítání projektu ze souboru. Zvažoval jsem celkem 3 možnosti – volat metody přímo, vytvořit každé třídě, u které by to mělo smysl, speciální metody k tomuto účelu určené (něco na způsob zasílání zpráv) nebo vytvořit speciální třídu, která bude mít tuto komunikaci na starosti.

První a druhý způsob je poměrně podobný. Částečně je používán při vykreslování, kde se volají přímo metody Draw každého objektu. Nevýhodou tohoto způsobu je špatná udržovatelnost kódu, který je „rozstrkán“ po celé aplikaci a v případě změny je nutno nejdříve složitě vyhledat, kde všude je nutno provést přepis.

Z tohoto důvodu byla snaha soustředit většinu takovéhoto kódu do třídy Notifier. Ta obsluhuje oba směry toku informací a v případě potřeby informovat ať už projekt nebo hlavní okno se pouze zavolá jeho metoda notifyProject/MainWindow...(). Tento způsob se udržuje daleko snáze (všechen kód je na jednom místě) a v místě volání je hned jasné, co se má stát.

Notifier je v aplikaci používán jako globální proměnná. Byla zvažována i možnost použít pro návrh této třídy, stejně jako třídy Logger, návrhový vzor singleton. Po krátké analýze jsem dospěl k názoru, že by tato volba přinesla spíše komplikace – volání GetInstance() při každém přístupu a v případě, že bych v budoucnu chtěl mít více instancí (u Notifieru to není nepředstavitelná myšlenka), musel bych třídu přepsat.

6 Závěr

6.1 Zhodnocení

Cílem práce bylo vytvořit aplikaci, ve která bude programátor schopen snadno navrhnout menší softwarové projekty a následně je převést do jednoho ze skutečných jazyků. Tento cíl byl do značné míry splněn vytvořením aplikace Ultimate Code Designer. Primárně je psána pro Windows, ale s poměrně malým úsilím by díky použitým knihovnám šla zkompilovat pro Linux či OS X.

Aplikace umožňuje navrhnout v grafickém rozhraní strukturu aplikace. GUI vizualizuje vztahy mezi objekty a tak usnadňuje programátorovi práci.

Hlavní přednosti vidím především v jednoduchém, ale pro účely aplikace naprosto dostačujícím formátu souborů jazyků. Soubory jsou snadno čitelné a dobře umožňují popsat objektový jazyk. Díky použitým technologiím jsou i snadno rozšiřitelné o případné nové prvky. Přidat nový typ proměnné je otázkou maximálně deseti minut.

Pokud bych UCD srovnal s programy zmíněnými v kapitole 4, tak nezbývá říci, že některými zmiňovanými chybami a nedodělky trpí UCD též, jde speciálně o GUI, kterému k profesionálnímu vzhledu ještě leccos chybí. Udělat dobré a dobře vypadající GUI se ukázalo jako dosti náročný úkol.

6.2 Budoucnost programu

UCD by šlo v budoucnu rozšířit jak v části označované jako jádro programu (projekt, jazyky, generování kódu), tak především v oblasti uživatelského rozhraní. Možností se nabízí celá řada:

- Doplnit do GUI možnost kopírování přes schránku (copy-paste).
- Umožnit uživatelské nastavení GUI – barvy objektů, šipek, možnost přesunovat šipky, aby se navzájem nekřížily apod. Také povolit uživateli nastavit si množství zobrazovaných informací na ploše (jenom názvy tříd nebo i s metodami atd.)
- Doplnit možnost undo.
- Umožnit editaci objektů přímo na ploše, ne pouze v dialozích.
- Zobecnit formát jazyka, aby zahrnoval i neobjektové jazyky. Teď je možné sice zapsat i neobjektový jazyk, ale chceme-li projekt do něj převádět, musí se s tím při návrhu počítat

(např. všechny třídy lze převést do globálního prostoru, ale nastane problém, pokud mají stejně pojmenované proměnné).

- Doplnit možnost snazšího generování projektu do více souborů s různými jmény.

7 Reference

[1] Rational Rose;
http://www-01.ibm.com/software/awdtools/developer/rose/

[2] IBM Takeover (Totally) Rational; http://www.information-management.com/news/6161-1.html

[3] Rational ClearCase ;
http://www-01.ibm.com/software/awdtools/clearcase/

[4] DialogBlocks; http://www.dialogblocks.com/

[5] UML Tool; http://www.altova.com/umodel.html

[6] Argo UML; http://argouml.tigris.org/

[7] Sharp, John: Microsoft Visual C# 2005 krok za krokem; 2006;
80-251-1156-3

[8] Prata, Stephen: Mistrovství v C++; 2001; 80-7226-339-0

[9] Sivakumar, Nishant : Is MFC dead? Does MFC have a future?;
http://blog.voidnish.com/?p=86

[10] Cellfish: MFC is not dead ;
http://blogs.msdn.com/cellfish/archive/2008/04/12/mfc-is-not-dead.aspx

[11] Smart, Julijan; Hock, Kevin; Csomor Stefan: Cross-Platform GUI Programmingwith wxWidgets; 2005; 01-3147-381-6

[12] Extensible Markup Language (XML); http://www.w3.org/XML/

[13] XML Tree; http://www.w3schools.com/xml/xml_tree.asp

[14] Berghen, Frank Vanden : Small, simple, cross-platform, free and fast C++ XML Parser; http://www.applied-mathematics.net/myself/myself.html

[15] Xerces-C++ XML Parser; http://xerces.apache.org/xerces-c/

[16] The Expat XML Parser; http://expat.sourceforge.net/

[17] Wu, Lei; Feng Yi; Yan, Huan:Software reengineering with architecture decomposition; http://delivery.acm.org/10.1145/1250000/1244320/p1489-wu.pdf?key1=1244320&key2=8497649421&coll=GUIDE&dl=GUIDE&CFID=46442702&CFTOKEN=24240661

[18] Garbage In, Garbage Out; http://en.wikipedia.org/wiki/Garbage_In,_Garbage_Out

[19] Julian Smart, Robert Roebling, Vadim Zeitlin and other members of the wxWidgets teamPortions (c): wxWidgets 2.8.10: A portable C++ and Python GUI toolkit; http://docs.wxwidgets.org/stable/

Příloha A – Uživatelská příručka

A 1 Obsah DVD a instalace

Na přiloženém DVD se nacházejí 3 složky - bin, source a text.

Ve složce bin je zkompilovaná release verze UCD se všemi podadresáři, které k ní náleží. Instalace se provede jednoduchým zkopírováním obsahu této složky na pevný disk počítače.

V adresáři source se nacházejí zdrojové soubory programu. Nejpodstatnější jsou podsložky h a src. V první jsou hlavičkové soubory a ve druhé samotné zdrojové soubory. Projekt (VisualStudio sollution) je k nalezení ve složce win32. Pokud se rozhodnete si projekt postavit, binární soubory se vytvářejí ve složce build/debug, resp. build/release, pracovní složkou je pak test_data.

Ve složce text se nachází tato práce ve formátu PDF.

A 2 Projekt

Nový projekt vytvořte kliknutím na „File->New Project“. Zadejte jeho jméno a klikněte na OK.

Chcete-li načíst projekt ze souboru, kam jste ho před tím uložili, klikněte na „File->Load Project“. Otevře se Vám dialogové okno, kde si můžete zvolit, který projekt chcete načíst. Klikněte na OK.

Projekt uložíte do souboru tak, že kliknete na „File -> Save Project“ nebo „File -> Save Project as“. První volba uloží soubor do souboru, ze kterého byl načten. Pokud byl vytvořen jako nový, je postup stejný jako u druhé volby. Ta otevře dialogové okno, kde můžete zadat jméno nového souboru.

A 3 Jazyky

Všechny soubory jazyků (přípona .udl) se musí nacházet v adresáři /languages, odkud jsou automaticky načteny při startu programu. Formát těchto souborů je popsán v příloze B.

A 4 Manipulace s objekty

Objekt můžete vytvořit několika způsoby – buď v horním menu aplikace

kliknutím na „Object -> Add *“ nebo kliknutím pravým tlačítkem myši na plochu okna a vybráním z kontextového menu položku „Add *“, kde za * si doplňte jeden z 5 možných typů objektů. Otevře se dialog příslušného objektu. Vyplňte Vámi požadovaná data a klikněte na OK.

Již existující objekty se zobrazují v levém panelu se stromovým pořadačem. Tam můžete editovat již existující objekty. Pravým tlačítkem nad jménem objektu otevřete kontextové menu, kde můžete buď objekt smazat nebo vyvolat jeho dialog a upravit vlastnosti. Objekt také můžete přejmenovat editováním jeho názvu přímo v panelu.

Každý objekt, krom ordinálních typů, je vizualizován v podobě krabičky na pravé ploše. Kliknutím na něj je možno s ním pohybovat. Pravé tlačítko opět vyvolá kontextové menu k příslušnému objektu.

Vztahy mezi objekty jsou vyjádřeny pomocí šipek. Od objektu vede šipka k typu, využívá-li objekt tento typ jako svou vnitřní proměnou. Šipka též může vyjadřovat dědičnost mezi třídami. Jednotlivé druhy šipek jsou barevně odlišeny a jejich zobrazování lze vypnout v horním menu „View“.

A 5 Generování projektu

Projekt vygenerujete do jednoho z jazyků kliknutím na „Project -> Generate“. Otevře se dialogové okno, kde můžete zvolit název souboru a kliknutím na OK se provede samotné generování.

Příloha B – Formát souborů jazyka

B 1 Stručný úvod do XML

Vzhledem ke skutečnosti, že jsou soubory jazyků psány ve formátu XML, považuji za vhodné umístit do práce malý rychlokurz XML, který bude stačit pochopení zápisu těchto souborů.

XML je podobně jako HTML značkovací jazyk. Společně vycházejí z dnes už takřka nepoužívaného SGML.

XML dokument se skládá z tagů, což jsou textové řetězce uzavřené ve dvojici špičatých závorek (menšítko a většítko). Tagem jsou třeba *<tag>,<foo>*. Aby byl dokument validní, musí (mimo jiného) být každý tag uzavřen. Tag je uzavřen, existuje-li k němu jeho uzavírací tag nebo je-li samotný tag ukončen znakem „/“. Uzavírací tag je takový tag, který má stejný řetězec a je uvozen znakem „/“. *<tag></tag>* a *<foo />* jsou uzavřené tagy.

Tag může obsahovat další tagy nebo text. Každý tag uvnitř jiného tagu musí být též uvnitř tohoto tagu uzavřený – nejsou povoleny konstrukce jako *<tag><foo></tag></foo>*. Dokument obsahující podobný řetězec by nebyl prohlášen za validní. Naopak toto je v pořádku. *<tag><foo /></tag>*.

Dále může tag ještě obsahovat atributy a jejich hodnoty. Tyto atributy typicky doplňují a upřesňují význam tagu. Atribut se píše za řetězec tagu, následuje ho rovnítko a za ním v uvozovkách hodnota. Atributů může mít tag teoreticky libovolně mnoho. Následující tag ukazuje dva zápisy tagu s dvěma atributy. *<tag atr1="one" atr2="two"> </tag>*.

Následující kapitola popíše význam jednotlivých tagů, možných atributů a jejich hodnot. XML Schema napsáno není z prostého důvodu, že aplikací používaný parser validaci dle schematu nepodporuje.

B 2 Rozdělení souboru na části

Soubor je členěn do čtyřech logických částí, které jsou celé uzavřeny v tagu *language.* První částí jsou options (uzavřeno do tagu *options*), pak format (tag *format*), následuje types, kde jsou deklarovány typy jazyka (tag *types*) a nakonec, část kde je popsán způsob jak generovat výsledný kód (tag *cmd* s atributem *type* nastaveným na hodnotu

root).

Tyto části jsou na sobě nezávislé a nezáleží na pořadí, avšak pro dobrou čitelnost souboru je doporučeno dodržovat pořadí výše uvedené nebo alespoň kořenový *cmd* uvést až na konec.

B 2.1 Tag options a jeho podtagy

V této části jsou specifikována především klíčová slova jazyka. Žádný z tagů nemá specifikovány atributy a ani žádné podtagy a tím pádem budou všechny ignorovány.

- *<name>*
  - název jazyka, tento text se zobrazí v programu, musí být jedinečný přes množinu všechny jazyky používané v aplikaci
- *<comment_begin>*
  - řetězec uvozující (víceřádkový) komentář
- *<comment_end>*
  - řetězec ukončující (víceřádkový) komentář
- *<const>*
  - klíčové slovo pro konstantní objekt
- *<abstract>*
  - klíčové slovo pro abstraktní objekt (v C++ znám jako čistě virtuální)
- *<static>*
  - klíčové slovo pro statický objekt
- *<virtual>*
  - klíčové slovo pro virtuální objekt
- *<public>*
  - klíčové slovo pro veřejný přístup k položkám třídy a pro veřejnou dědičnost
- *<protected>*
  - klíčové slovo pro chráněný přístup k položkám třídy a totožně pro dědičnost
- *<private>*

- klíčové slovo pro privátní přístup k položkám třídy a totožně pro dědičnost
- *<returns_const>*
  - řetězec dávající najevo, že funkce/metoda vrací konstantní objekt
- *<returns_pointer>*
  - řetězec dávající najevo, že funkce/metoda vrací ukazatel
- *<returns_reference>*
  - řetězec dávající najevo, že funkce/metoda vrací odkaz
- *<pointer>*
  - řetězec dávající najevo, že dotyčná proměnná je ukazatelem
- *<reference>*
  - řetězec dávající najevo, že dotyčná proměnná je ukazatelem
- *<parameters_delim>*
  - řetězec oddělující parametry v zápisu funkce
- *<inheritance_delim>*
  - řetězec oddělující jednotlivé poděděné třídy
- *<inheritance_simple>*
  - řetězec uvozující skutečnost, že třída dědí nějaké další třídy a rozhraní, v tomto případě nerozlišuje mezi class a interface
- *<inheritance_class>*
  - řetězec uvozující skutečnost, že třída dědí nějaké další třídy, rozlišuje mezi class a interface
- *<inheritance_interface>*
  - řetězec uvozující skutečnost, že třída dědí nějaké další rozhraní, rozlišuje mezi class a interface

B 2.2 Tag types

Tento tag obsahuje pouze jeden druh podtagu a to podtag *<type>*. Ten uvozuje

nový (ne nutně, ale typicky) ordinální typ definovaného jazyka. Je zde možno specifikovat libovolný řetězec. Název typu je obsahem tagu.

Tag má povinný atribut *id*, což je číslo, které bude programem použito pro vnitřní identifikaci typu a jeho mapování na typy v projektu. Díky tomu je mapování zachováno i při změně pořadí zápisu typů v tomto souboru nebo při přidání nového typu. ID musí být jedinečné přes všechny typy, doporučuje se začínat od 1 a číslo by nemělo překročit hodnotu INT_MAX, což je typicky 2147483647. Toto číslo považuji za dostatečné, neznám dnes jazyk s více jak několika desítkami typů.

B 2.3 Kořenový příkaz

Tento tag může obsahovat pouze dva podtagy a to <*cmd*> a <*elm*>, které reprezentují příkaz a element. Jejich význam je upřesňován širokou sadou atributů.

Struktura je velice podobná struktuře popsané v kapitole 5.4.4.

B 2.3.1 Tag cmd

Tag cmd je zkratkou pro příkaz. Sám nikdy nezpůsobí vygenerování kódu, jen určuje, pro jaké objekty se provede jeho vnitřek. Je ho tedy možné považovat za jednoduchý cyklus, což však neplatí úplně vždy. Má povinný atribut *type*, který specifikuje, co vše provede a jaký *type* mohou mít jeho podpříkazy a elementy. Atribut type a jeho možné hodnoty popisuje následující tabulka.

Hodnota	***Význam***	***Doplňující atributy***	***Poznámka***
root	Kořenový příkaz, v celém dokumentu je pouze jeden.	Nemá.	Jediným možným podpříkazem je cmd s type nastaveným na hodnotu *file*.
file	Vše uzavřené v tomto tagu se vypíše do jednoho souboru. Název je specifikován v projektu, přípona atributem.	- suffix – přípona souboru	Možné podpříkazy jsou všechny, kromě těch majících vztah přímo ke třídě (methods,...) či jinému objektu (params). Z elementů je možný pouze typ text.
classes	Vše uzavřené v tomto tagu se provede pro všechny třídy projektu.	- without (nepovinný) – pro které třídy se toto nemá vykonat, třídy oddělit čárkou	
spec_class	Vše uzavřené v tomto tagu se provede pro všechny třídy specifikované v atributu with.	- with (povinný) – pro které třídy se toto má vykonat, třídy oddělit čárkou	
structures	Vše uzavřené v tomto tagu se provede pro všechny	- without (nepovinný) – pro které struktury se toto nemá vykonat,	

	struktury projektu.	struktury oddělit čárkou	
spec_structure	Vše uzavřené v tomto tagu se provede pro všechny struktury specifikované v atributu with.	- with (povinný) – pro které struktury se toto má vykonat, struktury oddělit čárkou	
functions	Vše uzavřené v tomto tagu se provede pro všechny funkce projektu.	- without (nepovinný) – pro které funkce se toto nemá vykonat, funkce oddělit čárkou	
spec_function	Vše uzavřené v tomto tagu se provede pro všechny struktury specifikované v atributu with.	- with (povinný) – pro které funkce se toto má vykonat, funkce oddělit čárkou	
parameters	Vše uzavřené v tomto tagu se provede pro všechny parametry funkce nebo metody.	- delimiter (nepovinný) – zda se má používat oddělovač pro parametry, 1 = ANO, 0 = NE, žádná hodnota = ANO	Má smysl pouze v příkazu typu functions/spec_function nebo methods/spec_method.
methods	Všechny příkazy a elementy uzavřené v tomto tagu se provedou pro všechny metody třídy.	- without (nepovinný) – pro které metody se toto nemá vykonat, metody oddělit čárkou - public (nepovinný) – zda se mají vykonat podpříkazy pro veřejné metody, 1 = ANO, 0 = NE, žádná hodnota = ANO - protected (nepovinný) - zda se mají vykonat podpříkazy pro chráněné metody, 1 = ANO, 0 = NE, žádná hodnota = ANO - private (nepovinný) - zda se mají vykonat podpříkazy pro privátní metody, 1 = ANO, 0 = NE, žádná hodnota = ANO - abstract (nepovinný) - zda se mají vykonat podpříkazy pro abstraktní metody, 1 = ANO, 0 = NE, žádná hodnota = ANO	Má smysl pouze uvnitř příkazu typu classes/spec_class.
spec_method	Všechny příkazy a elementy uzavřené v tomto tagu se provedou pro všechny metody třídy specifikované v atributu with.	- with (povinný) – pro které metody se toto má vykonat, funkce oddělit čárkou	Má smysl pouze uvnitř příkazu typu classes/spec_class.
variables	Všechny příkazy a elementy uzavřené v tomto tagu se provedou pro všechny proměnné třídy, struktury nebo globální proměnné projektu.	- without (nepovinný) – pro které metody se toto nemá vykonat, metody oddělit čárkou - public (nepovinný) – zda se mají vykonat podpříkazy pro veřejné metody, 1 = ANO, 0 = NE, žádná hodnota = ANO - protected (nepovinný) - zda se mají vykonat podpříkazy pro chráněné metody, 1 = ANO, 0 = NE, žádná hodnota = ANO - private (nepovinný) - zda se mají vykonat podpříkazy pro privátní metody, 1 = ANO, 0 = NE, žádná hodnota = ANO	Má smysl uvnitř příkazu typu classes/spec_class, structures/spec_structure a file. Atributy public, private, protected mají smysl pouze uvnitř příkazu typu classes/ spec_class.

spec_variable	Všechny příkazy a elementy uzavřené v tomto tagu se provedou pro všechny proměnné třídy, struktury nebo globální proměnné projektu specifikované v atributu with.	- with (povinný) – pro které proměnné se toto má vykonat, funkce oddělit čárkou	Stejné jako *variables*.
inheritance_class	Všechny příkazy a elementy uzavřené v tomto tagu se vykonají pro všechny třídy, které jsou děděny do této třídy.	Nemá.	Má smysl pouze uvnitř příkazu typu classes/spec_class.
inheritance_interface	Všechny příkazy a elementy uzavřené v tomto tagu se vykonají pro všechna rozhraní která jsou děděna do této třídy.	Nemá.	

Toto jsou všechny možné atributy tagu cmd. Tabulka je řazena dle atributu type, protože ten je zdaleka nejdůležitější a smysl ostatních se řídí především podle něj.

B 2.3.2 Tag elm

Tag *elm* (element) slouží pro přímý výpis kódu. Musí být vždy uzavřen v tagu *cmd*, protože jeho přímý nadřazený příkaz určuje do značné míry význam elementu.

Každý tag *elm* je ve stromě jazyka listem, takže už neobsahuje žádné další tagy . Atributy jsou do značné míry společné pro všechny elementy, výjimky budou uvedeny vždy u příslušného typu. Nejdůležitějším je opět *type*, který bude uveden až jako poslední a bude mu věnována speciální část kapitoly. Všechny jsou nepovinné, krom již zmiňovaného *type*. Atributy jsou:

- *newline*
  - význam – zda po vypsání tagu bude do proudu výstupních znaků zařazen znak nového řádku, určuje počet takovýchto to znaků
  - hodnota – celé číslo bez znaménka (kolik znaků '\n' bude vypsáno do výstupního proudu), při vynechání atributu standardně 0
- *spaces_before*
  - význam – počet znaků mezera před vypisovaným elementem
  - hodnota – celé číslo bez znaménka, (kolik znaků ' ' bude vypsáno do výstupního proudu), při vynechání atributu standardně 0

- *spaces_after*
  - význam – počet znaků mezera za vypisovaným elementem
  - hodnota – celé číslo bez znaménka, (kolik znaků ' ' bude vypsáno do výstupního proudu), při vynechání atributu standardně 1 (aby za každým elementem byla mezera)

Atribut *type* nabývá celkem třech hodnot, které budou opět seřazeny v tabulce. V dalších dvou tabulkách budou uvedeny rozpoznávané řetězce uvnitř tagů, závislé na hodnotě *type* elementu a na hodnotě *type* nadřazeného příkazu.

Hodnota	***Význam obsahu tagu***	
text	Text, který bude vypsán do souboru bez ohledu na jakékoli další okolnosti.	Musí být uzavřen (i nepřímo) alespoň v příkazu typu *file*.
var	Proměnná, která nabývá hodnoty dle typu přímo nadřazeného příkazu. Rozpoznávané hodnoty v následující tabulce.	
sign	Text, jehož vypsání je určeno nějakou vlastností objektu. Rozpoznáváme hodnoty opět v další tabulce.	

Rozpoznávané řetězce elementu, je-li jeho typ *var* a jejich významy v závislosti na nadřazeném elementu.

- NAME
  - význam – vždy jméno objekt
  - má smysl, je-li přímo nadřazený příkaz jakéhokoli typu, krom *file* a *root*
- TYPE
  - význam – typ objektu, jde-li o objekt, kterému logicky typ přísluší (proměnná, parametr), návratový typ (funkce, metoda), typ dědičnosti (tím myslíme public, protected, private) nebo typ třídy (class nebo interface)
  - nemá smysl pro strukturu a ordinální typ, pro třídu ano, ale význam je posunut
- DOC
  - význam – dokumentace k objektu, má smysl pro každý příkaz krom *file* a *root*
  - doplňující atributy:
    - in_comment – zda má být dokumentace uzavřena v komentářích, 1 = ANO, 0 = NE, vynechání atributu = ANO

        - force_output – zda vypsat prázdné komentáře i v případě, že dokumentací je prázdný řetězec (bude vynechán CELÝ element a případné newlines apod), 1 = ANO, 0 = NE, vynechání atributu = NE

- CODE
    - význam – zdrojový kód objektu, má-li to smysl (funce, metody)
- NAME_OF_ROOT_OBJECT
    - význam – jméno objektu o stupeň výše, než je přímo nadřazený objekt, pro metodu je to jméno třídy, ve které je obsažena
    - je rozpoznáváno pouze u metod

Rozpoznávané řetězce elementu, je-li jeho typ *sign*.

- CONST_IND
    - vypíše – klíčové slovo označující konstantu, je-li objekt konstantní
    - má smysl u proměnné, metody, parametru
- POINTER_IND
    - vypíše – řetězec označují objekt, který je ukazatelem, je-li objekt ukazatelem
    - má smysl u proměnné a parametru
- REFERENCE_IND
    - vypíše – řetězec označují objekt, který je odkazem, je-li objekt odkazem
    - má smysl u proměnné a parametru
- VIRTUAL_IND
    - vypíše – klíčové slovo označující virtuální objekt, je-li objekt virtuální
    - má smysl proměnné ve třídě a metody
- COMMENT_BEGIN
    - vypíše – začátek komentáře
    - má smysl vždy, krom příkazu typu *root*
- COMMENT_END

- vypíše – konec komentáře
- má smysl vždy, krom příkazu typu *root*

- RETURNS_CONST_IND
  - vypíše – řetězec označující skutečnost, že objekt vrací konstantní hodnotu, vrací-li objekt konstantní hodnotu
  - má smysl u funkcí a metod
- RETURNS_POINTER_IND
  - vypíše – řetězec označující skutečnost, že objekt vrací ukazatel, vrací-li objekt ukazatel
  - má smysl u funkcí a metod
- RETURNS_REFERENCE_IND
  - vypíše – řetězec označující skutečnost, že objekt vrací odkaz, vrací-li objekt odkaz
  - má smysl u funkcí a metod
- GETTER_IND
  - vypíše -
  - má smysl
- SETTER_IND
  - vypíše -
  - má smysl
- ABSTRACT_IND
  - vypíše – klíčové slovo označující abstraktní objekt, je-li objekt skutečně abstraktní
  - má smysl u metody
- INHERITANCE_SIMPLE_IND
  - vypíše – klíčové slovo uvozující skutečnost, že třída dědí jiné třídy a při tom jazyk nerozlišuje mezi třídami a rozhraními
  - má smysl u třídy
- INHERITANCE_CLASS_IND
  - vypíše – klíčové slovo uvozující skutečnost, že třída dědí jiné třídy a při tom jazyk

rozlišuje mezi třídami a rozhraními

  - má smysl u třídy

- INHERITANCE_INTERFACE_IND
  - vypíše – klíčové slovo uvozující skutečnost, že třída dědí jiné rozhraní a při tom jazyk rozlišuje mezi třídami a rozhraními
  - má smysl u třídy