Aspektovo orientovaná syntaktická analýza

Marcel Tóth *
Marcel.Toth@tuke.sk

Abstrakt V tejto práci bolo hlavným cieľom preskúmať jeden z nových a perspektívnych prístupov k programovaniu, ktorým je aspektovo orientované programovanie (AOP), preskúmať funkčnosť tohoto prístupu v praxi a ilustrovať paradigmu AOP pomocou jednoduchého jazyka spolu s jeho prekladačom, čím mal byť vytvorený zrozumiteľný materiál pre základ štúdia aspektovo orientovaného programovania pre študentov, prípadne iných záujemcov. K tomuto účelu bol v tejto práci vytvorený jazyk (s názvom Kresl) pre zobrazovanie geometrických obrazcov spolu s jeho tkáčom, prekladačom do jazyka virtuálnych inštrukcií a interpréterom, teda virtuálnou tabuľou ako zobrazovačom výstupu. K jednoduchému jazyku Kresl bolo pridané jeho aspektové rozšírenie s príslušným tkáčom (špecializovaný prekladač spájajúci aspektový kód s pôvodným, komponentovým), ktorého použitím je možné overiť si efektivitu AOP prístupu k implementácii, prípadne návrhu softvérových aplikácií. Keďže vývoj AOP je ešte v počítačových "plienkach", existuje v AOP ešte viacero nedoriešených problémov, ktoré sú taktiež popísané v tejto práci a vyžadujú si ďalší výskum v danej oblasti. Jazyk Kresl so svojim tkáčom a prekladačom sa snaží prispieť k problematike AOP priblížením základného princípu AOP na úrovni syntaktickej analýzy, zotkávania a prekladu. Kvôli obmedzenému rozsahu tejto práce je väčšina informácií o výsledkoch podaná v skrátenej podobe.

Kľúčové slová: advice (rada, odporúčanie), aspect (aspekt), aspect oriented programming (aspektovo orientované programovanie), interpretation (interpretácia), joinpoint (bod spojenia), lexical analysis (lexikálna analýza), syntactic analysis (syntaktická analýza)

1 Úvod - potreba aspekovo orientovaného prístupu

Od časov vzniku prvých programov ulynulo veľa času, počas ktorého sa menili a vylepšovali aj techniky prístupu k návrhu a implementácii programov. Programovanie začalo na úrovni strojových inštrukcií, kde programátori strávili viac času uvažovaním o súbore inštrukcií daného stroja, ako nad riešením programátorského problému. Skúsenosťami a výskumom sa programovanie pomaly presúvalo k vyšším programovacím jazykom, ktoré umožňovali určitý stupeň abstrakcie stroja, na ktorom boli vykonávané. Avšak ako narastala komplexnosť navrhovaných programov a aplikácií, narastala potreba stále lepších techník abstrakcie a vyššieho stupňa znovupoužiteľnosti už naprogramovaného kódu. Vývoj postupoval cez procedurálne, štrukturované až k objektovo orientovanému programovaniu OOP (zámerne nie je vymenované funkcionálne programovanie odlišujúce sa od všetkých týchto, v praxi často používaných imperatívnych prístupov...). OOP nahliada na softvérový systém ako na súbor navzájom komunikujúcich objektov pripomínajúcich objekty reálneho sveta, teda tried, ktoré pomocou zapuzdrenia dovoľujú skryť implementačné detaily za rozhrania. OOP znamená aj polymorfizmus poskytujúci rovnaké rozhrania pre rôzne typy parametrov a návratových hodôt a nakoniec OOP samozrejme prináša

* Katedra elektrotechniky a informatiky FEI TU Košice, Letná 9, 040 22 Košice

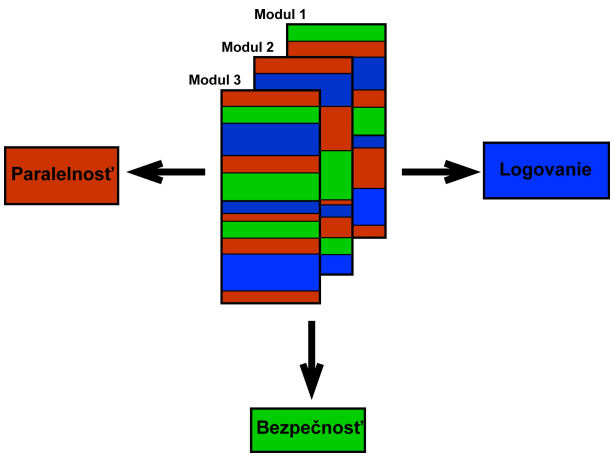
dedičnosť umožňujúcu rozširovanie a špecializáciu objektov (tried) bez potreby prístupu k implementácii základného konceptu. OOP, ktoré už preukázalo svoju silu v modelovaní objektov a štruktúr reálneho sveta, je prevládajúcim prístupom pri programovaní súčasných programových systémov. Napriek tomu sa programátori stretávajú s črtami, alebo vlastnosťami systémov, ktoré nekorešpondujú priamo so štruktúrami OOP, čím sa stávajú jeho organizačnými jednotkami ťažko popísateľné. Tieto vlastnosti pretínajú dizajn celého softvérového systému (sú poprepletané skrz celý systém), pretože kód ktorý ich implementuje sa objavuje vo viacerých organizačných jednotkách (metódach, triedach, ...). Takéto poprepletané vlastnosti nazývané tiež prekrížené súvislosti (crosscutting concerns) ovplyvňujú vývoj softvéru vo viacerých smeroch.

Príkladom takýchto prekrížených súvislostí (PS) je obrázok č. 1, ktorý zobrazuje tri moduly (povedzme, že ide o moduly v OOP, teda triedy), ktoré musia zabezpečiť určitú hlavnú úlohu (napríklad práca s distribuovanou databázou, ktorá zahŕňa modul pre čítanie, zápis a zálohovanie), pričom je potrebné vykonať aj nejaké bočné úlohy (to sú prekrížené súvislosti). V týchto moduloch je každá PS (logovanie, paralelnosť procesov, overovanie bezpečnosti) zobrazená jednou farbou. Z obrázku vidíme, že aj v rámci jedného modulu (triedy) je potrebné jednu PS zabezpečovať kódom na viacerých miestach, čím sa kód modulu stáva neprehľadným a nemodulárnym z určitého pohľadu dekompozície. Úloha pri aspektovom programovaní je nájsť tieto PS už pri návrhu softvérového systému, zlúčiť ich do súvisiacich celkov (aspektov), naprogramovať tieto celky samostatne (v moduloch oddelených od hlavného kódu, tým pádom viac modulárnych a lepšie udržiavateľných) a nechať špecializovaný aspektový prekladač (takzvaný tkáč) vygenerovať výsledný kód softvérovej aplikácie, alebo pri dynamickom tkaní dokonca zmeniť už bežiacu aplikáciu.

2 Princípy, prínosy a problémy AOP

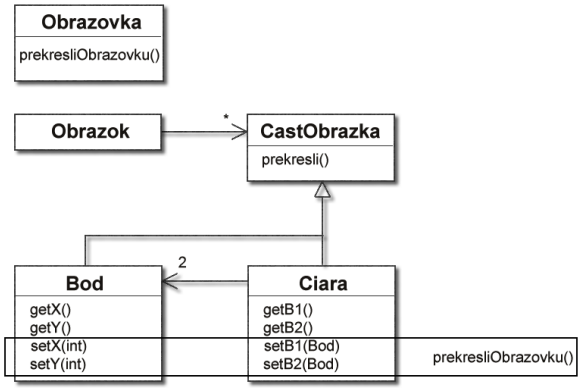
2.1 Základné pojmy AOP

Princípy AOP sú v tejto časti popísané všeobecne, avšak s väzbou na ich implementáciu v jazyku Kresl, čím je ozrejmené aj ich použitie vo vytvorenej aplikácii.



Obr. 1: Príklad zamotaného kódu

Základný rozdiel pri použití AOP spočíva v rozšírení jazyka komponentov o jazyk, alebo iba jazykové konštrukcie, pre spracovanie aspektov. Jazyk komponentov je klasický programovací jazyk (napríklad Java, C++, ...), ktorý bude rozširovaný o aspektové črty. Jazyk aspektov je jazyk používaný pro vyjadrenie aspektov a spôsobu ich spojenia s kódom komponentov (pôvodným programovacím jazykom). AOP tiež vyžaduje použitie niekoľkých nových princípov (v jazyku Kresl sú reprezentované pomocou kľúčových slov a logických výrazov), ktorých popis v skratke nasleduje. Popis pojmov AOP zároveň v skratke vysvetľuje aj základný princíp AOP.



Obr. 2: Príklad roztrúseného kódu modularizovateľného pomocou AOP

– **Aspekt**– Modulárna jednotka, ktorá zaobaľuje do jedného modulu jednu prekríženú súvislosť (crosscutting concern). Aspekt je väčšinou len syntaktické zaobalenie jednej súvislosti (concern) programu. V jazyku Kresl sa definuje kľúčovým slovom *aspect* rovnakým spôsobom ako trieda v OOP, ku ktorej je možné syntax definície aspektu prirovnať.

– **Bod spojenia** – Miesto v programe, kde môže dôjsť k spojeniu. Pod spojením sa rozumie prerušenie vykonávania programu, vykonanie kódu definovaného v príslušnom aspekte (v kóde rady) a pokračovanie v programe za bodom spojenia.

– **Prekrížená súvislosť** – Činnosť, úloha, ktorú program zabezpečuje v svojom kóde, ale nie je ju možné v danej štruktúre programu vyjadriť modulárne (teda ju zaobaliť do nejakej triedy, funkcie, ...). Obrázok č. 2 obsahuje zobrazenie takejto pretínajúcej aktivity (ide o zavolanie metódy na prekreslenie obrazovky po zmene obsahu obrazovky) v troch triedach (implementovaných V OOP) použitých v programe pre editáciu obrázkov, ktorú nie je možné, alebo je to prinajmenšom neprehľadné, vyjadriť ďalšou triedou, alebo inak modularizovať. Práve úlohou AOP je umožnenie uchovávania kódu vyjadrujúceho tieto pretínajúce aktivity na jednom mieste a tým odstránenie roztrúseného a zmotaného kódu.

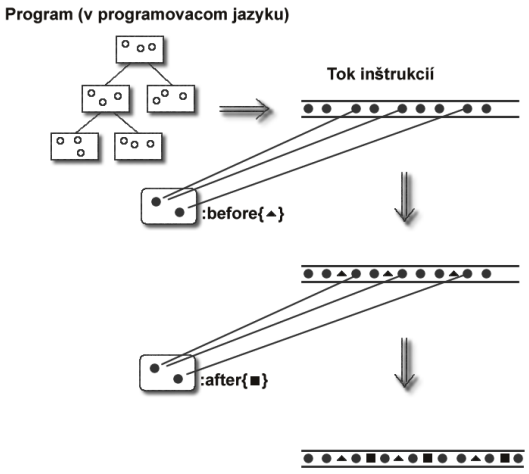
– **Rada**, alebo tiež **odporúčanie** – používa sa na definíciu kódu, ktorý je spustený po dosiahnutí určitých bodov spojenia určených značkovačom. V jazyku Kresl sa nachá-

dzajú tri typy rád. *Pred* (before), *po* (after) a *namiesto* (around). Tieto určujú, ako sa bude kód danej rady pripájať v určených bodoch spojenia. Napríklad pri type *pred* sa vykonáva program až po bod spojenia, v ktorom má nastať pripojenie rady, potom sa vykoná kód rady (pred tým, ako sa vykoná kód na ktorý sa rada viaže) a po kóde rady sa vykoná kód v bode spojenia, na ktorý sa rada viaže. Na obrázku č. 3 predstavujú kódy rád malý čierny trojuholník a malý čierny štvorec, pričom rada zobrazená ako trojuholník je typu pred a rada zobrazená ako štvorec je typu po. Obrázok zobrazuje aj tok inštrukcií po pripojení rád k bodom spojenia. Kód rady vo všeobecnosti môže meniť správanie sa programu (triedy, funkcie) na ktoré sa pripája, avšak nemôže meniť ich statickú štruktúru (definovanie nových typov, premenných, ...). Napríklad jazyk AspectJ (v súčasnosti vyvíjané a používané aspektové rozšírenie jazyka Java) toto rieši jazykovou konštrukciou *introduction* [10]

– **Tkáč aspektov** – Druh prekladača, ktorý vykonáva zotkávanie (weaving) zdrojového kódu aspektov so zdrojovým kódom komponentov. Komponentom je myslený klasický zdrojový kód vo funkcionálnom, štruktúrovanom, alebo objektovom programovacom jazyku. Zotkávanie prebieha ako pripájanie kódu rád (advice code) ku kódu komponentov, v definovaných bodoch spojenia. Body spojenia, v ktorých bude rada pritkaná, sú určené pomocou značkovačov. Tkáč môže vykonávať zotkávanie *statické* a *dynamické*. Pri statickom zotkávaní tkáč spája kód aspektov a komponentov už na úrovni zdrojových kódov, pred samotným spustením aplikácie a aj pred prekladom (kompiláciou) zdrojových textov, naproti tomu pri dynamickom zotkávaní je umožnené dokonca aj za behu programu votkať do programu nové aspekty (ide teda o modifikáciu programu za behu, čo je síce ešte predmetom výskumu, ale vytvára to priestor pre široké možnosti použitia [2], [17]). Tkáč jazyka Kresl vykonáva transformácie na úrovni zdrojového kódu, ide teda o statickú kompozíciu.

– **Značkovač(Pointcut)** – Značkovač je vlastne množinou bodov spojenia, ktorá je zo všetkých bodov spojenia v programe zložená pomocou výrazu značkovača (pointcut designator). Výraz značkovača je súčasťou definície rady (advice) a určuje tie body spojenia v kóde komponentu, na ktoré sa pripojí kód danej rady. Aspektové rozšírenia programovacích jazykov (napríklad AspectJ, AspectC), majú preddefinované funkcie značkovačov, ktoré vyberajú, filtrujú určitú množinu z bodov spojenia v programe. Príkladom z jazyka Kresl môžu byť funkcie *call(pointcut), execution(pointcut), set(pointcut)*, pričom *pointcut* môže byť napr. výraz *int ahoj()*. V takom prípade vráti funkcia *call(ahoj())* všetky body spojenia, ktoré sa viažu k volaniu funkcie ahoj (ktorá nemá parametre a vracia typ int). Povedzme funkcia *within(int ahoj())* by vrátila všetky body spojenia, ktoré sa nachádzajú v tele funkcie ahoj (ktorá nemá parametre a vracia typ int), táto funkcia značkovača však nie je v jazyku Kresl implementovaná (implementácia tkáča jazyka Kresl nemá za úlohu pokryť celú problematiku AOP, iba jej ilustratívnu časť). Silnou stránkou značkovačov všeobecne (aj v jazyku Kresl) je to, že môžu využívať tzv. výraz zhody (match expression), ktorý umožňuje definíciu značkovača na širšej množine s využitím žolíkov (wildcards). Pre ilustráciu opäť malý príklad : Výraz zhody v predchádzajúcich riadkoch bol *int ahoj()* (označujúci jednu funkciu). Nie je problém tento výraz zhody upraviť na tvar so žolíkom nahradzujúcim ľubovoľný znak, povedzme *% %oj()*, ktorý označuje všetky funkcie bez parametrov, ktorých meno sa končí na *oj* a vracajú ľubovoľný typ (znak %, žolík, nahrádza ľu-

bovoľný počet akýchkoľvek znakov). Značkovače a funkcie značkovačov je samozrejme možné spájať pomocou logických operátorov && (a súčasne) a ||| (alebo), čím sa dajú zadefinovať presné množiny bodov spojenia, ktoré spĺňajú výslednú podmienku (výsledný výraz značkovača)



Obr. 3: Princíp pripájania rady v bodoch spojenia

2.2 Výhody a problémy AOP

Potenciálnym najväčším prínosom AOP by malo byť vzájomné lepšie oddelenie súvisiacich častí zdrojového kódu programov, tým pádom aj prehľadnejšie a jednoduchšie programovanie. Koniec koncov, toto je trvalý cieľ programátorov od vzniku ich profesie. Nie celkom vyriešenými problémami pri AOP však stále sú otázky ako:

- Ako efektívne prevádzať dynamickú kompozíciu kódu aspektov s kódom bežiaceho programu ?
- Ako presne definovať body spojenia, na ktoré sa odkazujú výrazy značkovačov, a na ktoré sa majú kódy rád v aspektoch naviazať ?
- Je definovanie funkcií značkovačov dostatočným aparátom pre riešenie všetkých prípadov kompozície?
- Ako by bolo možné zmenšiť závislosť aspektov a výrazov značkovačov v nich od textu zdrojového kódu, a tým znížiť ich citlivosť na zmeny zdrojového kódu ?(Pri terajšom prístupe stačí zmena názvu premennej v zdrojovom kóde a zotkanie nevykoná určenú úlohu)

Tieto otázky sú neustále kladené a pracuje sa na ich riešení. V súčasnosti je síce stav AOP prirovnávaný ku stavu OOP pred 20timi rokmi, ale je predmetom výskumu viacerých skupín vo svete, čo by mohlo priniesť sľubné výsledky.

3 Jazyk Kresl s aspektovými črtami a jeho prekladač

Hlavným produktom tohto projektu sú prekladač, tkáč a interpreter jazyka Kresl. Úloha sa samozrejme týkala aj návrhu syntaxe a sémantiky spomenutého jazyka. Jazyk Kresl je jednoduchý štruktúrovaný jazyk obsahujúci príkazy pre pohyb pera po interpretačnej ploche, nastavenie jeho farby a na vykresľovanie geometrických obrazcov na určených pozíciách. Umožňuje vykresľovanie kruhov, obdĺžnikov, bodov, úsečiek a vypisovanie textov. Každú zmenu v zdrojovom kóde je takto možné ihneď po zotkaní, preklade a interpretácii zdrojového kódu vidieť názorne na interpretačnej ploche.

3.1 Jazyk Kresl

Syntax a sémantika jazyka Kresl vyplynuli z požiadavky čo najnázornejšieho zobrazenia výstupu jazyka s aspektovými črtami. Preto tento jazyk obsahuje základné jazykové konštrukcie štruktúrovaných programovacích jazykov, konkrétne kľúčové slová pre

- definíciu funkcií (**program**, **function**)
- definíciu premenných celočíselných a logických typov (**int**, **bool**, **void**)
- implementáciu logických, relačných a aritmetických operátorov a výrazov
- priradzovanie hodnôt premenným
- konštrukciu cyklov a podmienkových výrazov (**while**, **if**, **else**)
- primárne príkazy, teda presun kresliaceho pera, zobrazovanie obrazcov (bodov, čiar, oválov, obdĺžnikov a textov), zmeny farieb kresliaceho pera (**pen**, **line**, **oval**, **quad**, **text**, **color**)

Aspektové rozšírenie jazyka Kresl obsahuje nové jazykové konštrukcie a kľúčové slová, nepoužívané v objektovo orientovaných jazykoch. Tu je ich stručný výpis:

- definovanie aspektu (**aspect**)
- definícia rady (**advice**)
- definícia výrazu značkovača (**pointcut**)
- určenie typu rady (**before**, **after**, **around**)
- definície funkcií značkovačov (**execution**, **call**, **get**)

Kompletný formálny popis jazyka Kresl a jeho aspektového rozšírenia je mimo rozsah tohto dokumentu, preto sú v predchádzajúcom zozname uvedené aspoň niektoré kľúčové slová tohto jazyka s popisom ich použitia v sémantike jazyka.

3.2 Spôsob zotkávania v jazyku Kresl

Obrázok č. 3 obsahuje ilustratívne zobrazenie syntaktického stromu programovacieho jazyka Kresl, ktorý sa prekladá do toku inštrukcií (Zobrazené čiernymi krúžkami). Na body spojenia definované značkovačmi (zaoblené obdĺžníčky) medzi inštrukciami sú potom naviazané ďalšie inštrukcie určené kódmi rád (trojuholníček a štvorček v zložených zátvorkách). Takto je pomocou aspektov možné modifikovať pôvodný kód programu.

3.3 Implementácia modulov

Moduly tkáča a prekladača boli vzhľadom na zložitosť gramatiky jazyka Kresl (ide o gramatiku LR(k), pre ktorú je potrebná syntaktická analýza zdola nahor) generované pomocou sady generátorov kompilátorov Lex a Yacc, pre ktoré bola gramatika vytvorená v ich špecifikačných súboroch. Okenná aplikácia integrujúca všetky moduly do jedného celku bola implementovaná v jazyku C++, v prostredí C++ Builder.

4 Záver

Z analýzy súčasného stavu AOP je výsledkom práce predpoklad, že AOP je možné brať ako budúce ďalšie rozšírenie metodológií vývoja softvéru, teda napríklad ako rozšírenie najčastejšie používaného OOP, rovnako ako bolo OOP rozšírením štruktúrovaného programovania. Typickým príkladom podobného postupu z histórie programovacích jazykov bolo rozšírenie jazyka C o objektové črty, čím vznikol objektový jazyk C++. Samozrejme myšlienka AOP nie je viazaná na OOP, už vôbec nie na konkrétny jazyk, a dokonca ani na štruktúrované programovanie, pretože jeho postupy a princípy sú aplikovateľné na jazyk akéhokoľvek typu, dokonca aj na jazyk funkcionálny.

AOP je založené na pomerne zložitých princípoch(pre nováčika v tejto oblasti), aj vďaka tomu, že neexistuje veľa jeho jednoduchých aplikácií v praxi a tým pádom ani možností pre vytváranie príkladov a testovanie funkčnosti tejto paradigmy. Preto bolo cieľom aplikácií (prekladača, tkáča a interprétera) a jazyka(Kresl) navrhnutých počas projektu popisovaného touto prácou, vytvoriť jednoduchý jazyk demonštrujúci princípy AOP, funkčnosť statického zotkávania na tomto jazyku a nakoniec zobrazenie výstupu programu pomocou rôznych geometrických obrazcov. Všetky tieto úlohy boli splnené v miere potrebnej pre testovanie základov AOP a prekladu AOP. Pre moduly tkáča, prekladača a interprétera bolo tiež vytvorené user-friendly integrované prostredie (s názvom AsK) bežiace ako okenná aplikácia operačného systému Windows a úspešne spájajúce všetky moduly do jednej zrozumiteľnej aplikácie.

Programovacie jazyky vo všeobecnosti prichádzajú a odchádzajú, avšak problémy pri návrhu zložitých softvérových systémov ostávajú. A ambíciou AOP je prispieť aspoň malým kúskom do tejto evolúcie programovania. Či sa mu tu podarí, ukáže ďalší výskum v tejto oblasti (ku ktorému sa snaží svojím kúskom prispieť aj táto práca), prítomnosť dostupných jazykov a nástrojov pre AOP a samozrejme, čas.

Literatura

1. BARZILAY, O.: *AOP – Aspect oriented programming and the AspectJ tool*, University of Tel Aviv, 2003
2. BÖLLERT, K.: *On weaving aspects*, Kiel, Germany, 2000
3. BARZILAY, O.: *Advanced features using the AspectJ tool*, University of Tel Aviv, 2003
4. CONSTANTINIDES, C., SKOTINIOTIS, T.: *Reasoning About a Classification of Crosscutting Concerns in Object-Oriented Systems*, University of London & Northeastern university
5. CZARNECKI, K.: *Aspect–Oriented decomposition and composition*, Addison Wesley, 1999
6. ELRAD, T., AKSIT, M., KICZALES, G.: *Discussing aspects of AOP*, article in Communications of the ACM October 2001, vol. 40
7. ELRAD, T., FILMAN, E.R., BADER, A.: *Aspect–oriented programming*, article in Communications of the ACM October 2001, vol. 44
8. FRADET, P., SUDHOLT, M.: *AOP: Towards a generic framework using program transformation and analysis*
9. GRUNE, D., JACOBS, C.: *Parsing techniques (a practical guide)*, Dick Grune & Ceriel J. Jacobs, 1995, minor corrections 1998
10. KICZALES, G., HILSDALE, E., HUGUNIN, J., KERSTEN, M., PALM, J., GRISWOLD, G.W.: *An overview of AspectJ*, University of British Columbia & University of California, 2001
11. KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, CH., LOPES, C.V., LOINGTIER, J.–M., Irwin, J.: *Aspect–oriented programming*, Springer–Verlag LNCS 1241, June 1997
12. KIENZLE, J., GUERRAOUI, R.: *AOP: Does it make sense? The case of concurrency and failures*, Swiss Federal Institute of Technology in Lausanne (EPFL), Switzerland, 2001
13. MOLNÁR, Ľ., ČEŠKA, M., MELICHAR, B.:*Gramatiky a jazyky*, Alfa Bratislava 1987
14. PETRÁŠOVÁ, A.: *Prekladače*, Alfa, Bratislava, 1984
15. POPOVICI, A., GROSS, T., ALONSO, G.: *Dynamic weaving for aspect–oriented programming*, Swiss federal institut for technology Zürich , ACM, 2002
16. VÖLTER, M.: *Handling cross–cutting concerns: AOP and beyond*, 2003
17. WAND, M., KICZALES, G., DUTCHYN, CH.: *A Semantics for Advice and Dynamic Join Points in Aspect Oriented Programming*, Northeastern University & University of British Columbia, 2002
18. WIN, B.DE, VANHAUTE, B., DECKER, B.DE.: *Towards an open weaving process*, Department of computer science, K.U. Leuven, Heverlee, Belgium, August 2001
19. ZHAO, J.: *Towards a metrics suite for Aspect–Oriented Programming*, Information processing society of Japan, March 2002
20. Documentation of Aspect C++ sources, GNU general public license *The Lemon parser generator*, 2003

Annotation

Target of this diploma thesis is the resume of accessible informations about aspect oriented approach and explanation of basic principles of aspect oriented programming and aspect oriented syntactic analysis. Three co-operating modules implemented in this thesis are performing aspect oriented syntactic analysis, weaving and translation. Simple programming language for drawing geometric figures with aspect extension was developed as well as parser, interpretator and integrated development environment for this language. Part of this work describes advantages, drawbacks and possibilities of using AO programming in implementation of real projets.

Keywords advice, aspect, aspect orientation, interpretation, joinpoint, lexical analysis, syntactic analysis, parsing, AOP, programming