SLOVENSKÁ TECHNICKÁ UNIVERZITA
V BRATISLAVE
Fakulta elektrotechniky a informatiky
Študijný odbor: Informatika

Ľubomír Nerád

Vizualizácia aspektov v programovacom jazyku AspectJ

Diplomová práca

Vedúci diplomovej práce: Ing. Valentino Vranić
Máj 2002

Slovenská technická univerzita v Bratislave
Fakulta elektrotechniky a informatiky
Vedecko-pedagogická rada odboru Informatika

ZADANIE DIPLOMOVEJ PRÁCE

Meno diplomanta: **Bc. Ľubomír NERÁD**
Odbor: INFORMATIKA
Názov diplomovej práce: ***Vizualizácia aspektov v programovacom jazyku AspectJ***

Zadanie diplomovej práce:
Analyzujte problematiku vizualizácie aspektov v aspektovo-orientovanom programovaní v zmysle programovacieho jazyka AspectJ. Navrhnite spôsob vizualizácie aspektov. Pri návrhu sa snažte o prehľadnosť vizuálnej reprezentácie, zvlášť pri veľkom počte aspektov a tried. Zvážte možnosť generovania aspektov z ich vizuálnej reprezentácie. Navrhnuté riešenie overte implementáciou prototypu prostredia, ktoré umožní vizualizáciu aspektov pre jazyk AspectJ.

Odporúčaná literatúra:

- Gregor Kiczales et al. An Overview of AspectJ. In Proc. of ECOOP 2001—15th European Conf. on Object-Oriented Programming, Budapest, Hungary, June 2001. http://aspectj.org
- Gregor Kiczales et al.: Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, Proc. of ECOOP' 97 — Object-Oriented Programming, 11th European Conference, Jyväskylä, Finland, June 1997. Springer-Verlag. http://www.parc.xerox.com/aop
- Proc. of the Aspect-Oriented Workshop at ECOOP' 97 — Object-Oriented Programming, 11th European Conference, Finland, June 1997. Springer-Verlag. http://wwwtrese.cs.utwente.nl/aopecoop99/aop97.html

Diplomová práca musí obsahovať:
1. Anotáciu v slovenskom a anglickom jazyku
2. Analýzu problému
3. Opis postupu riešenia
4. Opis súvisu s diplomovým projektom
5. Výsledky riešenia a ich zhodnotenie
6. Zoznam použitej literatúry
7. Technickú dokumentáciu
8. Elektronické médium obsahujúce vytvorený produkt spolu so všetkou dokumentáciou a anotáciou

Miesto vypracovania:
Katedra informatiky a výpočtovej techniky FEI STU v Bratislave

Termín odovzdania diplomovej práce: 17. mája 2002
Vedúci diplomovej práce: Ing. Valentino Vranić

prof. Ing. Milan Kolesár, CSc.
predseda VPRI

Bratislava 15. februára 2002

ANOTÁCIA

Slovenská technická univerzita v Bratislave
Fakulta elektrotechniky a informatiky

Študijný odbor: INFORMATIKA

Autor: Ľubomír Nerád
Diplomová práca:
Vizualizácia aspektov v programovacom jazyku AspectJ

Vedúci diplomovej práce: Ing. Valentino Vranić

máj 2002

Táto práca sa zaoberá vizualizáciou aspektov v programovacom jazyku AspectJ. Hlavným cieľom bolo navrhnutie vizualizácie, ktorá by sa dala použiť vo vývojových prostrediach pre jazyk AspectJ.

Navrhnutá metóda vizualizácie je založená na sekvenčných diagramoch definovaných v UML(1). Najskôr boli sekvenčné diagramy rozšírené o aspekty. Tieto rozšírené diagramy umožňujú zobraziť účinok aspektov na správanie systému a môžu byť použité v etape analýzy a návrhu systému. Potom bola notácia týchto rozšírených diagramov upravená tak, aby sa dali vytvárať automaticky na základe statickej analýzy zdrojových súborov AspectJ programu.

Automatické vytváranie rozšírených sekvenčných diagramov bolo implementované ako súčasť prototypu prostredia pre programovací jazyk AspectJ. Prototyp ďalej podporuje tvorbu programov v jazyku AspectJ, umožňuje filtrovanie zobrazených diagramov a podporuje navigáciu v programe prostredníctvom týchto diagramov.

---

(1) UML: Unified Modeling Language

ANNOTATION

Slovak University of Technology, Bratislava
Faculty of Electrical Engineering and Information Technology

Degree Course: INFORMATICS

Autor: Ľubomír Nerád
Thesis:
Visualization of aspects in AspectJ programming language

Supervisor: Ing. Valentino Vranić

2002, May

This thesis deals with the visualization of aspects in AspectJ programming language. The main goal was to design a visualization, which could be used in development environments for AspectJ language.

The visualization of aspects method that has been designed is based on the sequence diagrams defined in UML(1). First, sequence diagrams were extended with aspects. These extended diagrams enable to depict how aspects affect the system behavior and can be used in analysis and design phase of system development. Subsequently, the notation of these extended diagrams was adapted to enable their automatic generation according to the static analysis of an AspectJ program.

The automatic generation of the extended sequence diagrams was implemented as a part of the development environment for AspectJ programming language prototype. The prototype also supports the AspectJ programs creation, enables the filtering of displayed diagrams and supports the navigation in a program through these diagrams.

---

(1) UML: Unified Modeling Language

Čestne prehlasujem, že som túto diplomovú prácu vypracoval samostatne s použitím publikácií uvedených v zozname použitej literatúry.

Obsah

1. Úvod

Aspektovo-orientované programovanie predstavuje nový spôsob programovania podobne ako to bolo pri objektovo-orientovanom programovaní v dobe jeho vzniku. Pretože je to pomerne nová oblasť existuje len málo jazykov, ktoré využívajú princípy aspektovo-orientovaného programovania. Najznámejším z nich je jazyk AspectJ. Jeho výhodou je, že je to jazyk, ktorý nie je špecializovaný na jednu problémovú oblasť, ale sa dá použiť všeobecne na ľubovolné problémy.

Ako ku každému jazyku aj k AspectJ začínajú postupne vznikať podporné vývojové prostredia. Samozrejmá je snaha o poskytnutie čo najväčšej miery pohodlia vývojárom, ktoré vzniknuté prostredia používajú. S tým je spojená aj snaha o zobrazenie vzťahov a štruktúry vytvoreného aspektovo-orientovaného programu (vizualizácia prvkov jazyka). Vytváraná vizualizácia sa snaží postihnúť účinky aspektov na vytváraný systém.

Cieľom tejto práce je návrh a overenie nového spôsobu vizualizácie aspektov, ktorý by poskytol iný pohľad na systém ako vizualizácie existujúcich prostredí pre jazyk AspectJ. Pri návrhu sa bude klásť dôraz na to, aby sa vytvorený spôsob vizualizácie dal použiť na zobrazovanie aspektov na základe statickej analýzy zdrojových súborov AspectJ programov a teda, aby sa vytvorený spôsob vizualizácie dal v prípade potreby zakomponovať do vývojových prostredí. Naviac sa bude prihliadať na prehľadnosť vizuálnej reprezentácie, ktorá je dôležitá pri vývoji zložitejších systémov. Overenie navrhnutej metódy bude spočívať v implementácii prototypu prostredia, ktoré umožní vytvárať a vizualizovať programy napísané v jazyku AspectJ.

Táto práca je rozdelená do niekoľkých častí. Prvá časť práce je zameraná na analýzu problému. Súčasťou tejto analýzy je opis programovacieho jazyka AspectJ (kapitola 2) s ohľadom na nové prvky jazyka (aspekty). Tiež sú analyzované vybrané vývojové prostredia (kapitola 3) pre tento jazyk spolu s vizualizáciou, ktorá je v týchto prostrediach implementovaná. Na základe tejto analýzy je v druhej časti navrhnutý nový spôsob vizualizácie aspektov pre jazyk AspectJ (kaptitola 4). Ďalšia časť práce sa zaoberá overením navrhnutej vizualizácie vytvorením prototypu prostredia na vizualizáciu aspektov (kapitola 5). V poslednej časti práce sa venujem dosiahnutým výsledkom s ohľadom na projektovanie v diplomovom projekte (kapitola 6) a nakoniec celkovému prínosu a možností jeho využitia (kapitola 7). Posledná kapitola (kapitola 8) obsahuje zoznam použitej literatúry.

K prílohám práce patrí technická dokumentácia (príloha A), používateľská príručka k vytvorenému prototypu prostredia (príloha B), elektronický nosič (príloha D) s vytvoreným prototypom, testovacími údajmi a dokumentáciou v elektronickej forme. Podrobný popis obsahu elektronického nosiča je uvedený v prílohe C.

Na tomto mieste chcem poďakovať môjmu vedúcemu diplomovej práce Ing. Valentinovi Vranićovi za jeho odborné vedenie, rady a podnety, ktoré pomohli skvalitniť výsledok mojej práce.

2. Programovací jazyk AspectJ

AspectJ je programovací jazyk využívajúci princípy aspektovo-orientovaného programovania [1]. Je to aspektovo-orientovaný jazyk pre všeobecné použitie, čo znamená, že nie je viazaný len na jednu konkrétnu problémovú oblasť. Tento programovací jazyk nevznikal od základov, ale je nadstavbou jazyka Java. Ponecháva si všetky pôvodné prvky jazyka Java a naviac definuje nové jazykové konštrukcie – aspekty.

Nasledujúci opis jazyka AspectJ sa týka verzie 1.0 [2] a je možné, že niektoré popísané črty jazyka budú v ďalších verziách zmenené.

2.1. Aspekt

Inštancia aspektu sa dá v AspectJ prirovnať k objektu v jazyku Java. Má s ním niektoré spoločné vlastnosti, ktorými sú zapúzdrenie a dedičnosť. Podobne ako objekt aj inštancia aspektu definuje svoj vnútorný stav reprezentovaný hodnotami svojich atribútov a tiež definuje správanie (zmenu stavu) reprezentovanú svojimi metódami. Naviac od objektu môže dodefinovať správanie a atribúty iných objektov, čo nie je objektom v OOP (napríklad v Jave) dovolené.

Dôležitosť aspetov sa dá ukázať na príkladoch. Jedným z nich je príklad implementácie kontrolných výpisov do tried objektov. Predpokladajme, že chceme, aby každý vytvorený objekt vypísal na výstup správu o svojom vytvorení. Ďalej predpokladajme, že máme už jednotlivé triedy vytvorené. Implementácia kontrolných výpisov sa dá vytvoriť dvomi spôsobmi.

Prvý spôsob predstavuje priame dopísanie časti kódu s kontrolným výpisom do konštruktorov tried. To má za následok opakovanie sa kódu kontrolného výpisu na viacerých miestach programu (aj vo viacerých zdrojových súboroch). Okrem toho, že môžeme na nejakú triedu zabudnúť, takto implementované kontrolné výpisy vedú k horšej modifikovateľnosti programu, pretože zmena kontrolného výpisu vyžaduje zmenu vo všetkých triedach.

Druhým spôsobom je vytvorenie triedy, implementujúcej kontrolné výpisy (základná trieda) a jej zdedenie ostatnými triedami. Takto dosiahneme, že kontrolné výpisy budú na jednom mieste programu. Stále ale ostáva nutnosť dopísania dedenia tejto triedy do ostatných tried a prípadne aj volanie kontrolného výpisu implementovaného v základnej triede (toto volanie môžeme dať do konštruktora základnej triedy, ale nemusí to vyhovovať požiadavkám na kontrolný výpis). Hlavnou nevýhodou tohto prístupu je, že takto vytvorená hierarchia tried nebude zodpovedať hierarchii vytvorenej v návrhu. Trieda na kontrolné výpisy je pridaná do hierarchie nasilu kvôli implementácii kontrolných výpisov.

Riešenie poskytuje aspekt. Môžeme vytvoriť aspekt, ktorý bude sledovať vytváranie objektov ostatných tried a pri každom vytvorení môže na výstup vypísať správu o vytvorení inštancie triedy.

Riešenie za pomoci aspektu má viaceré výhody:

- ľahká modifikovateľnosť (kód kontrolného výpisu je na jednom mieste programu – v aspekte)
- zrozumiteľnosť (aspekt zachytáva, to čo chceme dosiahnuť)
- pridaním aspektu riešime problém kontrolných výpisov pre všetky triedy objektov (aj pre tie, ktoré budú do programu pridané neskôr)
- kontrolné výpisy zrušíme vymazaním aspektu, bez nutnosti zasahovania do definovaných tried

2.2. Zloženie aspektu

V Jave sa trieda skladá z atribútov a metód. Ako bolo už spomenuté aj aspekt má svoje atribúty a metódy. Okrem klasických atribútov a metód má ešte ďalšie prvky. Tieto slúžia na určenie bodov programu, v ktorých sa má beh programu prerušiť (*bodový prierez* - angl. *pointcut*) a určenie operácie, ktorá sa má v týchto miestach programu vykonať (*rada* - angl. *advice*). Aspekt má tiež prostriedky na pridanie kódu do existujúcich tried (*zavedenie* - angl. *introduction*).

Príklad aspektu:

```
aspect Trace
{
	static int count = 0;
	pointcut creating(): call(new(..));

	after() returning() : creating()
	{
		System.out.println(++count + ". Class "
			+ thisJoinPoint.getSignature().getDeclaringType()
			+ " created!");
	}
}
```

Tento aspekt obsahuje jeden atribút (`count`), jeden *bodový prierez* (`create`) a jednu *radu* (`after`).

Bodový prierez:

Bodový prierez je množina *spojovacích bodov* určujúcich, kde sa má beh programu prerušiť a má sa vykonať časť kódu aspektu. *Spojovací bod* (angl. *join point*) sa viaže sa na nejakú udalosť, ktorou môže byť volanie metódy (metód) alebo začatie vykonávania metódy (metód). Tiež to môže byť volanie konštruktora objektu alebo začatie vykonávania kódu konštruktora. Okrem volaní sa môžu definovať miesta vo vykonávanom programe, kde dochádza k nastaveniu alebo čítaniu určitého atribútu triedy, alebo nastane výnimka určitého typu.

Bodový prierez sa v programe definuje takto:
pointcut <názov>(<parametre>) : <popisovač>

pointcut	-	kľúčové slovo
<názov>	-	konkrétny identifikátor

<parametre> - konkrétne parametre zapisované rovnako ako parametre metód v Jave

<popisovač> - pozostáva z elementárnych popisovačov *spojovacích bodov* alebo negovaných elementárnych popisovačov *spojovacích bodov* (negácia - !) spojených logickými spojkami (a - &&, alebo - ||) a tiež môže pozostávať z názvov ďalších definovaných *bodových prierezov*

Elementárne popisovače:
Existuje viacero druhov elementárnych popisovačov, ktoré slúžia na určenie *spojovacích bodov* vo vykonávanom programe. Každý elementárny popisovač sa skladá z kľúčového slova a parametrov.

Obvykle každý *bodový prierez* obsahuje aspoň jeden z elementárnych popisovačov, ktorý určuje udalosť volania metód, konštruktora (`call`), alebo začatie vykonávania metód, konštruktora (`execution`), prípadne nastavenie (`set`), či čítanie atribútu (`get`) alebo zachytenie výnimky (`handler`). Ostatné elementárne popisovače slúžia na bližšie vymedzenie miesta vykonávaného programu, kde nás daná udalosť zaujíma (`args`, `this`, `target`, `within`, `withincode`, `cflow`, `cflowbelow`).

Elementárne popisovače používané pri definícii bodového prierezu sú nasledujúce:

- `call(S)`
  Určuje *spojovacie body*, v ktorých sa volajú metódy vymedzené v S, nezahŕňa volanie metód rodičovskej triedy, pri ktorom nedochádza k dynamickému linkovaniu.
  `S` môže byť názov metódy alebo konštruktora a môže obsahovať znaky *, .., +
  Príklad:
  `call(* get*(..))` - volania metód, ktorých mená začínajú s `get`, majú ľubovolné vstupné parametre a vracajúce hodnotu ľubovolného typu

- `execution(S)`
  Určuje *spojovacie body*, v ktorých sa začne vykonávať kód volaných metód vymedzených v S. Zahŕňa aj začiatok vykonávania metód rodičovskej triedy, ktorých adresa bola určená statickým linkovaním.
  `S` môže byť názov metódy alebo konštruktora a môže obsahovať znaky *, .., +
  Príklad:
  `execution(void Counter+.inc(int, ..))` -
  začiatok vykonávania metód s názvom `inc`, s prvým parametrom typu `int`, bez návratových hodnôt, definovaných v triede `Counter`, alebo v triedach z nej odvodených (znak +)

- `args(GTN, GTN, ...)`
  Určuje *spojovacie body*, v ktorých sa dajú získať argumenty rovankých typov a počtu ako ich vymedzuje postupnosť `GTN`.
  `GTN` môže byť výraz zložený z mien tried a môže obsahovať znaky *, .., +
  `GTN` môže byť nahradený parametrom
  Príklad:
  `args(short, .., short)` -
  prvý a posledný argument je typu short

- `get(S)`
  Určuje *spojovacie body*, kde sa číta hodnota atribútov vymedzených v `S`.
  `S` je názov atribútu a môže obsahovať znaky *, .., +
  Príklad:
  `get(int *..Line.x)`
  - čítanie atribútu `x` typu `int` triedy `Line` ľubovolného balíka
  `get(int *.x)` - čítanie atribútu `x` ľubovolnej triedy

- `set(S)`
  Určuje *spojovacie body*, kde sa nastavuje hodnota atribútov vymedzených v `S`.
  `S` je názov atribútu a môže obsahovať znaky *, .., +
  Príklad:
  `sets(int Line.x)` - nastavenie atribútu `x` triedy `Line`

- `handler(GTN)`
  Určuje *spojovacie body*, kde dôjde k začiatku vykonania obsluhy výnimiek typov špecifikovaných v `GTN`.
  `GTN` môže byť výraz zložený z mien typov výnimiek a môže obsahovať znaky *, .., +
  Príklad:
  `handler(RuntimeException)` -
  začiatok vykonania obsluhy výnimky `RuntimeException`

- `this(GTN)`
  Určuje všetky také miesta vykonávaného programu, kde aktuálny objekt je inštanciou triedy vymedzenej v `GTN`.
  `GTN` môže byť výraz zložený z mien tried a môže obsahovať znaky *, .., +
  `GTN` môže byť nahradený parametrom
  Príklad:
  `this(*)` – aktuálny objekt je inštanciou ľubovolnej triedy
  `this(Line || Circle)` - aktuálny objekt je inštanciou triedy `Line` alebo `Circle`

- `target(GTN)`
  Určuje všetky *spojovacie body*, ktorých cieľovým objektom je objekt triedy vymedzenej v `GTN`.
  `GTN` môže byť výraz zložený z mien tried a môže obsahovať parametre a znaky *, .., +
  `GTN` môže byť nahradený parametrom
  Príklad:
  `target(Line+)` -
  spojovacie body, ktorých cieľovým objektom je inštancia triedy `Line` alebo triedy od `Line` odvodenej

- `within(GTN)`
  Určuje všetky miesta vykonávaného programu, ktoré sú definované v triedach vymedzených v `GTN`.
  `GTN` môže byť výraz zložený z mien tried a môže obsahovať znaky *, .., +
  Príklad:
  `within(!Circle)` – miesta vykonávaného programu, ktoré nie sú definované v triede `Circle`

- `withincode(S)`
  Určuje všetky miesta vykonávaného programu, ktorý je definovaný v metódach alebo konštruktoroch vymedzených v `S`.
  `S` môže byť názov metódy alebo konštruktora a môže obsahovať znaky *, .., +
  Príklad:
  `withincode(int *(..))` - kód prislúchajúci ľubovolnej metóde, ktorá má ľubovolné vstupné parametre s ľubovolným počtom ale musí mať návratovú hodnotu typu `int`
  `withincode(void inc(*, *))` -
  kód prislúchajúci metóde s názvom `inc`, ktorá nemá žiadnu návratovú hodnotu, má práve dva parametre ľubovolného typu

- `cflow(P), cflowbelow(P)`
  Určuje všetky miesta vykonávaného programu, začínajúce *spojovacími bodmi* vymedzenými v `P` a končiace po skončení udalosti, na ktorú boli viazané.
  `P` je popisovač *bodového prierezu*
  Príklad:
```
pointcut top() : call(class02.method02());
pointcut flw() : cflow(top()), call(class01.method01());
```
  - `flw` definuje body, v ktorých sa volaná metóda `method01` triedy `class01`, ale iba počas volania metódy `method02` triedy `class02`

Rada

Rada sa v aspekte viaže na nejaký *bodový prierez*. Definuje operáciu, ktorá sa má vykonať vo *spojovacích bodoch* určených priradeným *bodovým prierezom*.

Zapisuje sa takto:
```
<rada> (<parametre>) : <popisovač> { <príkazy> }
```

<*rada*>	-	konkrétny typ (`before`, `after` alebo `around`)
<parametre>	-	konkrétne parametre zapisované rovnako ako parametre metód v Jave
<popisovač>	-	pozostáva z elementárnych popisovačov *spojovacích bodov* alebo negovaných elementárnych popisovačov *spojovacích bodov* (negácia - !) spojených logickými spojkami (a - &&, alebo - ||) a tiež môže pozostávať z názvov ďalších definovaných *bodových prierezov*
<príkazy>	-	postupnosť príkazov, ktorá sa má vykonať

Rada môže byť týchto typov:

- `before`
  Určuje, že postupnosť príkazov sa vykoná pred nastatím udalosti špecifikovanej v popisovači.
  Príklad:

```
pointcut moving(): call(void Line.move(int, int));
before(): moving()
{
      System.out.println("Before moving!");
}
```

  Pred každým volaním metódy `move` (2 vstupné parametre typu `int`, žiaden výstup) triedy `Line`, sa vypíše správa.

- `after`
  Určuje, že postupnosť príkazov sa vykoná po udalosti špecifikovanej v popisovači.
  Príklad:

```
pointcut creating(): call(new(..));
after() returning() : creating()
{
      System.out.println(
            thisJoinPoint.getSignature().getDeclaringType()
            + " created!");
}
```

  Po každom úspešnom vytvorení inštancie (zabezpečí to `returning()`) nejakej triedy sa vypíše správa o vytvorení spolu s menom vytvorenej triedy (`thisJoinPoint.getSignature().getDeclaringType()`).

- `around`
  Používa sa vtedy, keď je potrebné niečo vykonať pred aj po nastatí nejakej udalosti.
  Príklad:

```
pointcut moving(): call(void Line.move(int, int));
void around() : moving()
{
      System.out.println("Before moving!");
      proceed();
      System.out.println("After moving!");
}
```

  Pred každým a po každom volaním metódy `move` (2 vstupné parametre typu `int`, žiaden výstup) triedy `Line`, sa vypíše správa. Volanie metódy `proceed()` spôsobí vykonanie tela metódy `move`.

Parametre

Bodový prierez a *rada* môžu mať v deklaračnej časti parametre. Tieto parametre slúžia na získanie dodatočných informácií v danom *spojovacom bode*. Môžu sa pomocou nich získať napríklad hodnoty parametrov, s ktorými je volaná sledovaná metóda, alebo sa dá získať objekt, ku ktorému volaná metóda patrí.

Príklady:

```
pointcut moving(int x, int y) : call(void Line.move(int, int))
                                    && args(x, y);
after(int x, int y): moving(x, y)
{
      System.out.println("New position is " + x + ", " + y + "!");
}
```

Po každom volaní metódy `move` triedy `Line` sa vypíše správa o posune spolu s novými súradnicami. Pri zachytení volania metódy `move` sa naviažu parametre `x`, `y` uvedené v `args(x,y)` na konkrétne hodnoty. Budú sa sledovať volania len tých metód, ktoré majú vstupné parametre rovnakých typov ako majú parametre `x`, `y` uvedené v ľavej časti deklarácie *bodového prierezu* `moving` (`pointcut moving(int x, int y)`). Po volaní metódy `move` triedy `Line` sa naviažu parametre `x`, `y` uvedené v deklaračnej časti `after` na hodnoty, ktoré boli naviazané v *bodovom priereze*, pomocou `moving(x, y)` a môžu byť využité v kóde *rady*.

```
pointcut moving(Line l): call(void move(int, int)) && target(l);
after(Line l): moving(l)
{
      l.redraw();
}
```

Po každom volaní metódy `move` nejakého objektu triedy `Line`, sa zavolá metóda `redraw` posúvaného objektu. Znovu dochádza k naviazaniu parametrov uvedených v ľavých častiach deklarácie *bodového prierezu* a príkazu `after` na konkrétne hodnoty podľa pravej časti. Pričom najskôr dochádza k naviazaniu v *bodovom priereze* a až potom v príkaze `after`.

Špeciálne premenné

Pri vykonávaní kódu prislúchajúcemu k *rade* sú dostupné premenné s názvom `thisJoinPoint` a `thisJoinPointStaticPart`. Tieto bližšie charakterizujú *spojovací bod*, v ktorom sa začalo vykonávanie príkazov danej *rady*. Rozdiel medzi nimi je ten, že zatiaľ čo `thisJoinPoint` poskytuje všetky informácie o danom spojovacom bode, `thisJoinPointStaticPart` poskytuje len statické informácie, ktoré sa dajú zistiť zo samotného programu.

Premenná `thisJoinPoint` umožňuje prístup k objektu typu `JoinPoint`. Typ `JoinPoint` má definované nasledujúce metódy:

`getArgs()`	-	vracia hodnoty argumentov v danom *spojovacom bode*
`getKind()`	-	vracia druh *spojovacieho bodu*
`getSignature()`	-	vráti hlavičku *spojovacieho bodu*
`getSourceLocation()`	-	vracia miesto v programe prislúchajúce danému *spojovaciemu bodu*
`getStaticPart()`	-	vracia objekt, ktorý združuje statické informácie o danom *spojovacom bode*
`getTarget()`	-	vracia cieľový objekt v danom *spojovacom bode*
`getThis()`	-	vracia aktuálny objekt v danom *spojovacom bode*
`toLongString()`	-	vráti podrobný popis *spojovacieho bodu*
`toShortString()`	-	vráti skrátený popis *spojovacieho bodu*

Premenná `thisJoinPointStaticPart` umožňuje prístup k objektu typu `JoinPoint.StaticPart`. Typ `JoinPoint.StaticPart` má definované nasledujúce metódy:

`getKind()`	-	vracia druh *spojovacieho bodu*
`getSignature()`	-	vráti hlavičku *spojovacieho bodu*
`getSourceLocation()`	-	vracia miesto v programe prislúchajúce danému *spojovaciemu bodu*
`toLongString()`	-	vráti podrobný popis *spojovacieho bodu*
`toShortString()`	-	vráti skrátený popis *spojovacieho bodu*

Zavedenie

Zavedenie umožňuje aspektom dodefinovať triedam metódy (konštruktor), atribúty, rozhrania alebo dedenie. Deklaruje sa vo vnútri aspektov podobne ako *rady* a *bodové prierezy*.

Druhy zavedenia:

- zavedenie atribútu
  Príklad:
  ```
  private int Point.z = 0;
  ```
  dodefinuje atribút `z` do triedy `Point`, ktorý sa inicializuje na nulu

- zavedenie metódy
  Príklad:
  ```
  public int Point.getZ() { return z; };
  ```
  dodefinuje verejnú metódu `getZ` do triedy `Point`

- zavedenie konštruktora
  Príklad:
  ```
  public Point.new(int _x, int _y) { x = _x; y = _y; };
  ```
  dodefinuje konštruktor s dvoma parametrami typu `int` do triedy `Point`

- zavedenie rozhrania
  Príklad:
  ```
  declare parents: Point implements Comparable;
  ```
  pridá k rozhraniam triedy `Point` rozhranie `Comparable`

- zavedenie dedenia
  Príklad:
  ```
  declare parents: Point extends GeometricObject;
  ```
  špecifikuje, že trieda `Point` rozširuje triedu `GeometricObject`

Pri zavedení sa dajú naraz špecifikovať aj viaceré triedy. Napríklad dodefinovať metódu `getZ` môžeme naraz triede `Point`, `Line` a `Circle` takto:

```
public int (Point || Line || Circle).getZ() { return z; };
```

Tiež sa dajú pri dodefinovaní do viacerých tried používať znaky .., *.

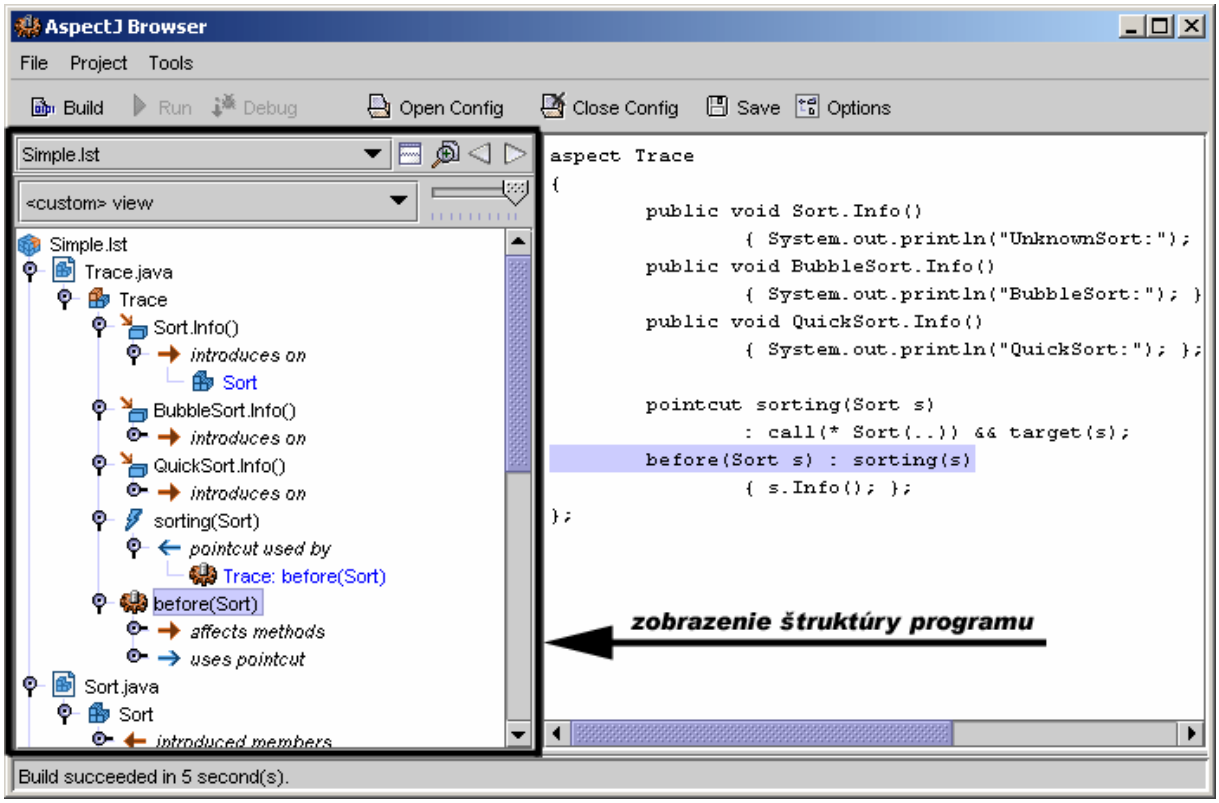
3. Analýza vývojových prostredí pre AspectJ

V tejto kapitole budú analyzované vybrané vývojové prostredia, ktoré umožňujú programovanie v jazyku AspectJ a tiež poskytujú vizualizáciu prvkov tohto jazyka. Pretože jazyk AspectJ je pomerne novým jazykom, väčšina existujúcich vývojových prostredí je vytvorená ako nadstavba vývojových prostredí pre jazyk Java. Takto boli vytvorené analyzované nadstavby AJDE (AspectJ Development Environment) for Java Builder a AJDE for Emacs. Výnimkou je AspectJ Browser, ktorý možno označiť za samostatne vytvorené prostredie pre jazyk AspectJ.

Okrem uvedených prostredí a nadstavieb, v súčasnosti existujú ešte dve AspectJ nadstavby vývojových prostredí (AJDE for Forte a AJVE for NetBeans). Tieto nadstavby sú však natoľko podobné nadstavbe AJDE for JavaBuilder, že ich uvádzanie nemalo zmysel.

3.1. AspectJ Browser

Analyzovaná bola verzia 1.0.4 tohto prostredia. Prostredie je vytvorené v jazyku Java. V uvedenej verzii neboli dopracované niektoré funkcie, ktoré podporujú bežné vývojové prostredia. V podstate je prostredie zamerané predovšetkým na zobrazenie štruktúry programu v jazyku AspectJ a navigáciu v zdrojových súborov prostredníctvom nej. Používateľské rozhranie tohto prostredia je zobrazené na obrázku 3.1.



Obrázok 3.1: Používateľské rozhranie prostredia AspectJ Browser

Práca s týmto prostredím je trochu ťažkopádna, kvôli spomínaným nedopracovaným funkciám. Prostredie napríklad neumožňuje vytvoriť projekt a špecifikovať zdrojové súbory, ktoré majú byť súčasťou projektu. Preto treba vytvoriť (mimo prostredia) konfiguračný súbor so zoznamom zdrojových súborov programu a až po otvorení (tlačítko `Open Config`) takéhoto súboru a skompilovaní projektu je možné s prostredím ďalej pracovať. Skompilovanie programu je nutné z toho dôvodu, že práve výstup z kompilátora je použitý na zobrazenie štruktúry programu a až prostredníctvom zobrazenej štruktúry sa dajú v prostredí otvárať a následne editovať súbory projektu. Na editáciu zdrojových súborov je v prostredí zabudovaný jednoduchý editor bez zvýrazňovania syntaxe (syntax highlighting) jazyka.

Nevýhodou prostredia je, že po zmene zdrojových súborov, je jedinou možnosťou ako zobraziť aktuálnu štruktúru programu prekompilovanie celého projektu. Naviac takáto kompilácia musí skončiť úspešne, lebo inak sa posledne zobrazená štruktúra programu nezmení. Jednou z chýbajúcich funkcií prostredia je aj možnosť spúšťania (`Run`) a odladovania (`Debug`) skompilovaného programu. Pre tieto funkcie existujú položky v menu a na nástrojovej lište prostredia, ale ich použitie je blokované. Dá sa predpokladať, že spomínané nedostatky prostredia budú odstránené v jeho ďalších verziách.

Zaujímavá z pohľadu vizualizácie je časť prostredia zobrazujúca štruktúru programu. Je podobná zobrazeniu štruktúry programu, ktorá je implementovaná v prostrediach pre objektovo-orientované jazyky (napr. JavaBuilder alebo Microsoft Visual C++). Podobne ako tieto prostredia aj AspectJ Browser zobrazuje jednotlivé prvky programu a ich vzájomné vzťahy. Okrem prvkov jazyka Java (balíky, triedy, metódy ...) a ich vzťahov, zobrazuje aj nové prvky jazyka AspectJ (aspekty, rady, zavedenia, bodové prierezy ...), ich vzájomné vzťahy a tiež ich účinky na ostatné prvky programu. Spôsob zobrazenia štruktúry AspectJ programu implementovaný v tomto prostredí bude prezentovaný na programe uvedenom na obrázku 3.2.

súbor QuickSort.java:

```
public class QuickSort extends Sort
{
      /* quick sort algorithm */
      public void Sort(int[] arr)
            { ... };
};
```

súbor BubbleSort.java:

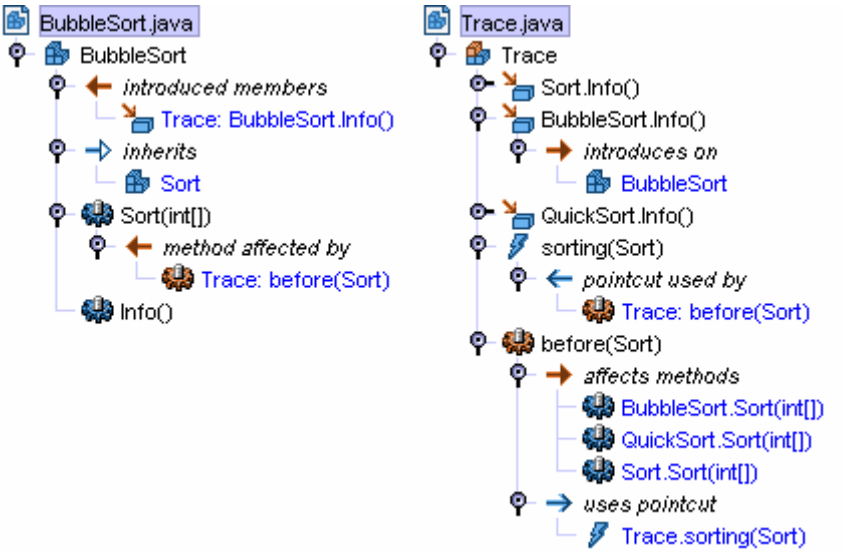
```
public class BubbleSort extends Sort
{
      /* bubble sort algorithm */
      public void Sort(int[] arr)
            { ... };
};
```

súbor Trace.java:

```
aspect Trace
{
      public void Sort.Info()
            { System.out.println("UnknownSort:"); };
      public void BubbleSort.Info()
            { System.out.println("BubbleSort:"); };
      public void QuickSort.Info()
            { System.out.println("QuickSort:"); };

      pointcut sorting(Sort s) : call(* Sort(..)) && target(s);
      before(Sort s) : sorting(s)
            { s.Info(); };
};
```

Obrázok 3.2: Program na triedenie polí

Program definuje triedy `QuickSort`, `BubbleSort`, abstraktnú triedu `Sort` (nie je znázornená) a aspekt `Trace`. Trieda `Sort` deklaruje abstraktnú metódu `Sort`. Táto metóda je definovaná v dcérskych triedach (trieda `QuickSort`, `BubbleSort`) tak, že implementuje konkrétny algoritmus triedenia vstupného poľa. Aspekt `Trace` zabezpečuje dodefinovanie metódy `Info` do tried `Sort`, `BubbleSort` a `QuickSort`, ktorá vypíše informáciu o druhu triedenia. Táto metóda je volaná z rady `before`, ktorá je vykonávaná pre miesta programu popísané bodovým prierezom s názvom `sorting`. Tento popisuje miesta programu, kde sa volá metóda `Sort` a cieľom volania je objekt triedy `Sort`, alebo triedy odvodenej od triedy `Sort`. Čiže rada `before` aspektu `Trace` zabezpečí vypísanie informácie o druhu triedenia, pred každým triedením poľa.



Obrázok 3.3: Zobrazenie štruktúry programu pre súbor BubbleSort.java a Trace.java

Na obrázku 3.3 je uvedená štruktúra programu pre súbory `BubbleSort.java` a `Trace.java`. Pre súbor `BubbleSort` je zobrazená trieda `BubbleSort` s metódami `Sort` a `Info`. Pri triede `BubbleSort` je uvedená jej nadtrieda `Sort` (časť `inherits`). Naviac je znázornená skutočnosť, že metóda `Info` bola dodefinovaná zavedením `BubbleSort.Info` v aspekte `Trace` (časť `introduced members`) a tiež, že operácia implementovaná metódou `Sort` je ovplyvnená radou `before` aspektu `Trace` (časť `method affected by`). Pre súbor `Trace.java` je na obrázku znázornený aspekt `Trace` spolu s definovanými zavedeniami `Sort.Info`, `BubbleSort.Info` a `QuickSort.Info`, bodovým prierezom `sorting` a radou `before` s parametrom typu `Sort`. Pri zavedení sú uvedené typy, ktoré zavedenie ovplyvňuje (časť `introduces on`). Pri bodovom priereze je uvedené, v ktorých radách je použitý (časť `pointcut used by`). Pri rade sú zobrazené metódy ovplyvnené danou radou (časť `affects methods`) a tiež pomenované bodové prierezy, ktoré sú použité pri definícii bodového prierezu danej rady (časť `uses pointcut`).

Prostredie AspectJ Browser okrem samotného zobrazenia štruktúry programu poskytuje možnosť filtrovania a usporiadania zobrazenej štruktúry.
Umožňuje špecifikovať:

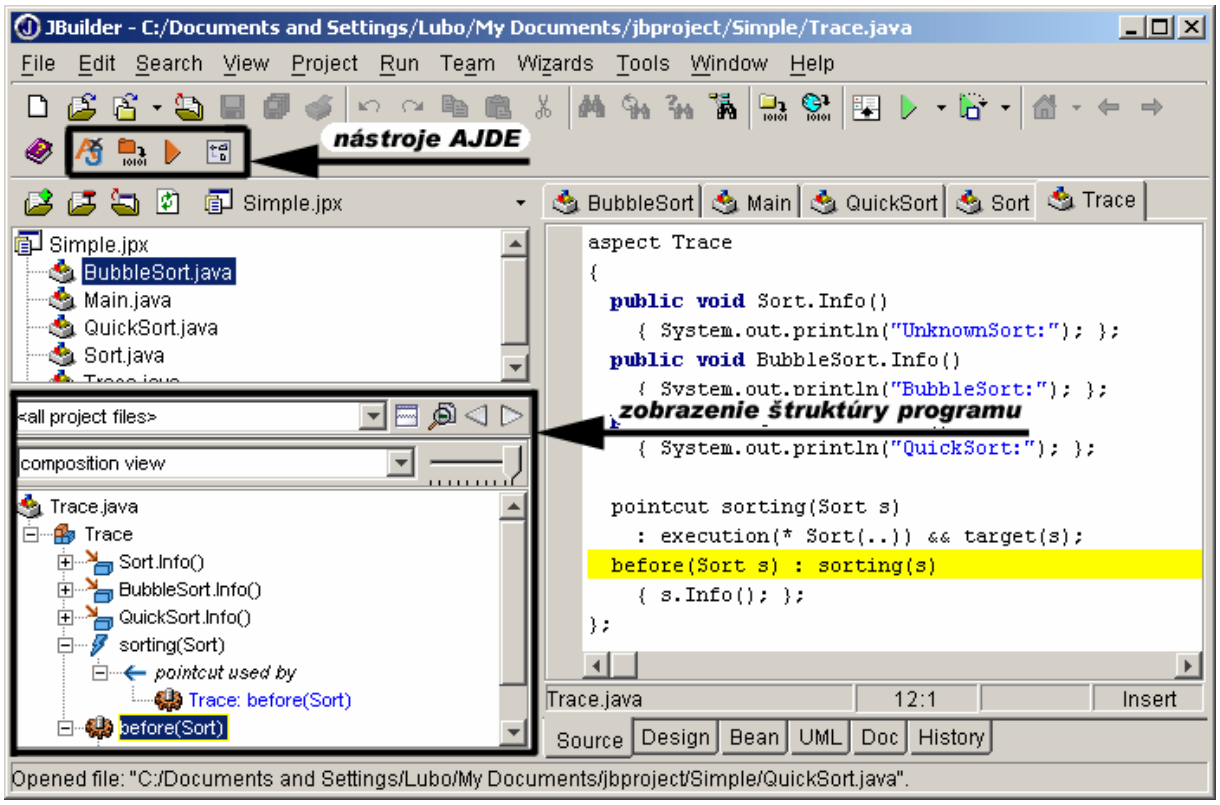
- zobrazené vzťahy
  uvádzanie nadtried, rád ovplyvňujúcich metódu, metód ovplyvnených radou ...
- usporiadanie prvkov
  podľa abecedy alebo podľa poradia v zdrojovom súbore

- zobrazené prvky
  - podľa prístupu - public, package, protected, private, privileged
  - podľa modifikátorov - static, final, abstract ...
  - podľa druhu - trieda, aspekt, metóda, rada ...
- granularitu zobrazenia
  úroveň súborov, balíkov, tried alebo členov

Zaujímavá je tiež navigácia v zdrojových súboroch projektu prostredníctvom zobrazenej štruktúry programu. Táto spočíva v možnosti zobrazenia a zvýraznenia prvku štruktúry programu v editore zdrojového súboru (viď obrázok 3.1, na ktorom je zvýraznená rada `before` aspektu `Trace` po kliknutí na túto radu v zobrazenej štruktúre programu).

3.2. AJDE for Java Builder

AJDE for Java Builder je nadstavbou vývojového prostredia Java Builder pre jazyk Java. Nadstavba ako aj prostredie boli vytvorené v jazyku Java. Analyzovaná bola verzia 1.0.4 tejto nadstavby s prostredím Java Builder verzie 6.0. Používateľské rozhranie rozšíreného prostredia je znázornené na obrázku 3.4.



Obrázok 3.4: Používateľské rozhranie rozšírenia AJDE for Java Builder

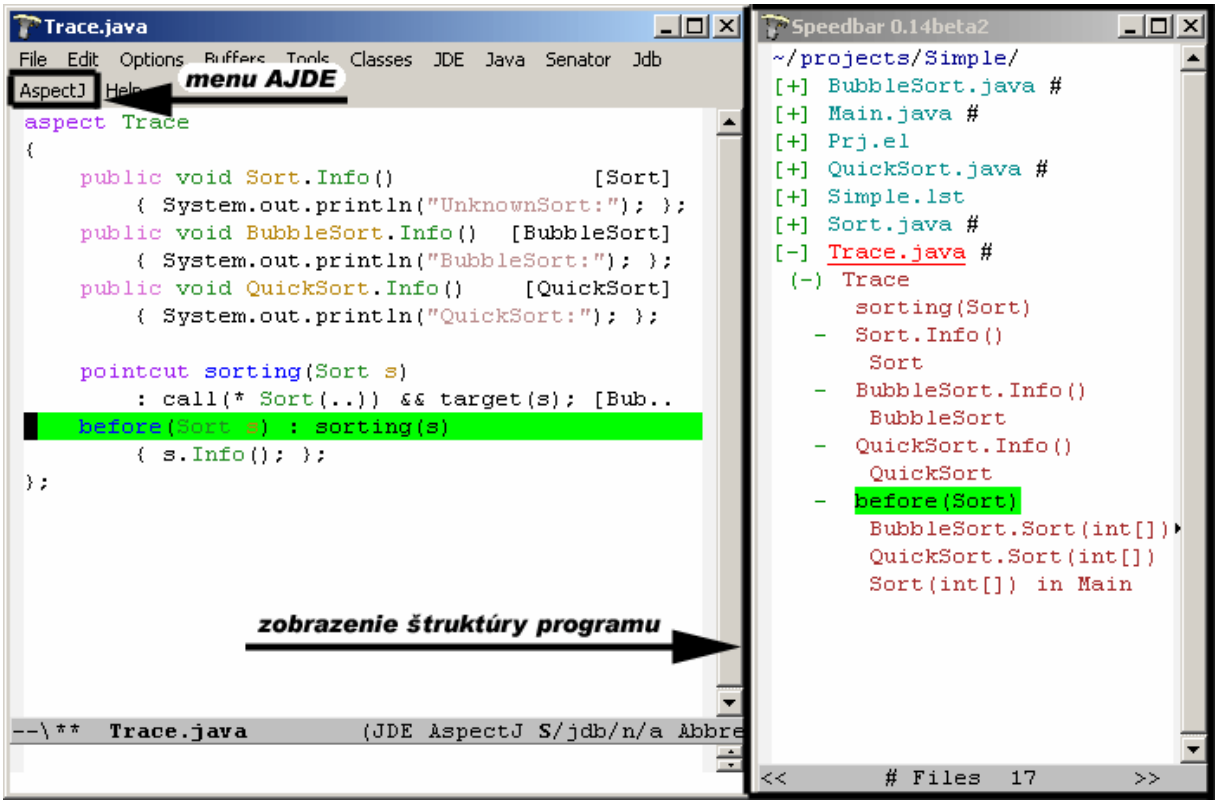
Rozšírenie pridalo do prostredia nové príkazy, prístupné cez menu programu a tiež prostredníctvom nástrojovej lišty. Pridané príkazy (nástroje AJDE) slúžia na zapnutie/vypnutie rozšírenia, kompilovanie projektu pomocou AspectJ kompilátora, spustenie takto skompilovaného projektu a nastavenie rozšírenia. Po zapnutí rozšírenia sa odblokujú ostatné príkazy nadstavby a tiež sa pridá okno, zobrazujúce štruktúru AspectJ programu. Toto zobrazenie je rovnaké ako zobrazenie štruktúry v prostredí AspectJ Browser a tiež závisí od

kompilácie projektu. Na kompiláciu a spustenie AspectJ programu je nutné použiť príkazy nadstavby, pretože použitie príkazov, ktoré na tento účel existovali v pôvodnom prostredí by skončilo s chybou (tieto príkazy používajú kompilátor jazyka Java a nie jazyka AspectJ). Pomocou príkazu nastavenia rozšírenia sa dá podobne ako v prostredí AspectJ Browser nastaviť filtrovanie a usporiadanie prvkov štruktúry programu. Zobrazená štruktúra programu tiež podporuje navigáciu v zdrojových súboroch projektu.

Na rozdiel od prostredia AspectJ Browser toto rozšírené prostredie umožňuje vytváranie projektu, špecifikovanie súborov projektu, ukladanie projektu na disk a jeho obnovu z disku. Na tento účel slúžia pôvodné príkazy prostredia. Na editáciu zdrojových súborov projektu je použitý editor prostredia Java Builder, ktorý podporuje zvýrazňovanie syntaxe, ale táto nebola nadstavbou rozšírená a teda zvýrazňuje len syntax jazyka Java.

3.3. AJDE for Emacs

AJDE for Emacs je podpora programovania v jazyku AspectJ pre textový editor Emacs. Podpora vyžaduje inštaláciu JDE (Java Development Environment) pre tento textový editor. Obidve nadstavby sú vytvorené v jazyku lisp (elisp), ktorý je súčasťou editora Emacs. Analyzovaná bola verzia 1.0.4 AJDE s textovým editorom NT Emacs verzie 21.2 a JDE verzie 2.2.9beta9. Používateľské rozhranie textového editoru spolu s uvedenými nadstavbami je zobrazené na obrázku 3.5.



Obrázok 3.5: Používateľské rozhranie rozšírenia AJDE for Emacs

Rozšírenie pre AspectJ pridalo do menu textového editoru nové príkazy (položka `AspectJ`). Tieto príkazy slúžia na kompilovanie editovaného programu a nastavenia rozšírenia. Spustenie skompilovaného programu zabezpečuje pôvodná nadstavba pre jazyk Java. Podobne ako to bolo v predchádzajúcich prostrediach, aj toto rozšírenie umožňuje zobrazenie štruktúry AspectJ programu, ktoré je znovu závislé od kompilácie projektu. Zobrazené elementy a ich vzťahy sú v podstate rovnaké ako v predchádzajúcich prostrediach a tiež umožňujú navigáciu v zdrojových súboroch projektu. Nevýhodou je, že zobrazenú štruktúru programu nie je možné filtrovať. Projekt je špecifikovaný buď súborom so zoznamom zdrojových súborov projektu (ako v prostredí AspectJ Browser) alebo adresárom, v ktorom sa nachádzajú všetky zdrojové súbory projektu. Zdrojové súbory sú editované v okne editora Emacs, ktorý umožňuje zvýrazňovanie syntaxe. Nadstavba AJDE rozširuje toto zvýrazňovanie tak, že sú zvýrazňované nielen lexikálne jednotky jazyka Java ale aj nové lexikálne jednotky jazyka AspectJ.

Zaujímavé je zobrazenie anotácií pri editovaní zdrojových súborov AspectJ, ktoré implementuje AJDE. Tieto anotácie sú vytvárané pre niektoré deklarácie tried, metód, aspektov, bodových prierezov, rád a zavedení a tiež pre príkazy volania. Tieto anotácie naznačujú väzby medzi triedami a aspektami.

Príklad zdrojových súborov spolu so zobrazenými anotáciami uvádza obrázok 3.6. Tento zachytáva už spomínaný program na triedenie neusporiadaných polí. Naviac je zobrazená hlavná trieda programu (Main Class), ktorou je trieda s názvom `Main`. Táto volá metódu `Sort` objektu triedy `QuickSort` so vstupnou hodnotou, ktorou je neutriedené pole.

```
public class Main
{
    public static void main(String[] cmdline)
    {
        int[] x = { 10, 3, 7, 1, 4, 5, 8, 2, 9, 6 };
        Sort sort = new QuickSort();
        sort.Sort(x);                                    [Trace]
    };
};
```

```
public class QuickSort extends Sort                      [Trace]
{
    /* quick sort algorithm */
    public void Sort(int[] arr) { ... };                 [Trace]
};
```

```
aspect Trace
{
    ...
    public void QuickSort.Info()                    [QuickSort]
        { System.out.println("QuickSort:"); };
    pointcut sorting(Sort s)
        : call(* Sort(..)) && target(s);
    before(Sort s) : sorting(s)  [BubbleSort, Main, QuickSort]
        { s.Info(); };
};
```

Obrázok 3.6: Zdrojové súbory AspectJ programu s anotáciami

Anotácie zobrazené na obrázku (v haranatých zátvorkách) uvádzajú názov triedy alebo aspektu, ktorý sa vzťahuje k danému miestu programu. AJDE naviac umožňuje vypísať ku každej anotácii zoznam konkrétnych častí aspektu (rady, bodové prierezy, zavedenia) alebo častí triedy (metódy, volania), ktoré sa viažu k danému miestu programu a tiež dovolí navigáciu v zdrojových súboroch prostredníctvom zobrazených anotácii (prechod na naviazaný prvok programu).

Ďalej budú uvedené väzby, ktoré anotácie zobrazujú spolu s konkrétnym výpisom pre príklad uvedený na obrázku 3.6.
Anotácie zachytávajú:

- **väzbu medzi volaním a radou**
  - pre volanie umožňujú vypísať zoznam rád viazaných na dané volanie
    pre volanie `sort.Sort(X)` v metóde `Main` je to:
    ```
    Trace: before(Sort)
    ```
  - pre radu umožnia vypísať volania metód, na ktoré je rada naviazaná
    pre radu `before` aspektu `Trace` je to:
    ```
    Sort(int[]) in Main
    ```
- **väzbu medzi metódou a radou**
  - pre metódu umožnia výpis rád viazaných na danú metódu
    pre metódu `Sort` triedy `QuickSort` je to:
    ```
    Trace: before(Sort)
    ```
  - pre radu umožňujú vypísať metódy, na ktoré je rada viazaná
    pre radu `before` aspektu `Trace` je to:
    ```
    BubbleSort.Sort(int[])
    QuickSort.Sort(int[])
    ```
- **väzbu medzi triedou a zavedením**
  - pre triedu umožňujú vypísať zoznam zavedení, ktoré ju dodefinujú
    pre triedu `QuickSort` je to:
    ```
    Trace: QuickSort.Info()
    ```
  - pre zavedenie umožňujú vypísať zoznam tried, ktorých sa dané zavedenie týka
    pre zavedenie `QuickSort.Info` aspektu `Trace` je to:
    ```
    QuickSort
    ```

3.4. Zhodnotenie

Z predchádzajúcej analýzy existujúcich prostredí pre AspectJ sa dajú vyvodiť niektoré závery, ktoré neskôr budú slúžiť pri návrhu vizualizácie aspektov v jazyku AspectJ a tiež pri špecifikácii požiadaviek na prototyp prostredia overujúci navrhnutú vizualizáciu.

Z pohľadu vizualizácie aspektov som identifikoval tri spôsoby vizualizácie, ktoré prostredia implementovali.
Sú to:

- zobrazenie štruktúry AspectJ programu
- zvýrazňovanie syntaxe AspectJ programu
- vytváranie anotácií v programe

K výhodám zobrazenia štruktúry programu patrí to, že umožňuje zobraziť nové prvky jazyka AspectJ a ich vzájomné vzťahy k ostatným prvkom jazyka. Prehľadnosť zobrazenia sa dá zvýšiť možnosťou filtrovania a usporiadania zobrazenej štruktúry. Veľmi výhodnou sa javila pri používaní týchto prostredí možnosť navigácie v zdrojových súboroch programu, ktorá sa stáva s príchodom aspektovo-orientovaného programu oveľa potrebnejšou.

Podobné výhody sa spájajú aj s vytváraním anotácií v programe. Tieto tiež umožňujú navigáciu, zobrazujú vzťahy medzi aspektami a triedami. Naviac zobrazujú väzby nielen na úrovni metód (rád) programu ale aj vo vnútri metód (rád), na úrovni jednotlivých príkazov.

Zvýrazňovanie syntaxe zase zvyšuje prehľadnosť a čitateľnosť zdrojových súborov a tiež umožňuje predísť niektorým triviálnym chybám pri písaní programu (napr. preklep pri písaní kľúčového slova).

K nevýhodám vizualizácie pomocou zobrazenia štruktúry programu a anotácií v opísaných prístupoch patrí predovšetkým potreba kompilácie programu na to, aby mohla byť vizualizácia zobrazená. Táto je naviac vykonaná kompilátorom jazyka AspectJ, ktorý je vytvorený v jazyku Java, čím sa stáva kompilácia rozsiahlejších programov zdĺhavá. Okrem toho sa požaduje, aby kompilácia prebehla úspešne, čo obmedzuje použitie tejto vizualizácie pri písaní zdrojových súborov programu, kedy chceme zobraziť čiastočne vytvorený program.

Ďalšou nevýhodou spomínaných vizualizácií je to, že zobrazujú vytváraný systém z rovnakého pohľadu. V podstate zachytávajú len statickú štruktúru vytváraného programu.

Z týchto predností a nedostatkov som vyvodil nasledujúce závery:

- vizualizácia by mala byť vykonaná na základe statickej analýzy programu, pričom túto statickú analýzu vykoná samotné prostredie pre jazyk AspectJ
- mala by umožňovať navigáciu v zdrojových súboroch programu
- pre zvýšenie prehľadnosti by mala umožniť filtrovanie zobrazenia
- mala by zobrazovať iný pohľad na vytváraný systém, ako existujúce vizualizácie

Rozhodol som sa, že navrhnem vizualizáciu, ktorá bude zachytávať správanie systému. Správanie systému poskytuje spomínaný „iný pohľad" na systém. O tom či je zobrazovanie správania vhodné pre vývojové prostredia bude rozhodnuté na vytvorenom prototype.

4. Návrh vizualizácie aspektov v AspectJ

V predchádzajúcej časti boli uvedené existujúce metódy použité na vizualizáciu aspektov vo vývojových prostrediach pre AspectJ. V tejto časti uvediem návrh novej metódy vizualizácie, ktorá je založená na rozšírení sekvenčných diagramov definovaných v univerzálnom modelovacom jazyku (angl. Unified Modeling Language, skratka UML) [3]. Výber metódy bol ovplyvnený predovšetkým tým, že sekvenčné diagramy umožňujú zobrazenie interakcie medzi jednotlivými objektmi systému (zachytávajú správanie a nie statickú štruktúru) a po rozšírení o aspekty umožňujú postihnúť účinok aspektov na túto interakciu. Naviac výber metódy (sekvenčných diagramov) bežne používanej v etape analýzy a návrhu a jej prispôsobenie na vizualizáciu v etape implementácie umožňuje spätné overenie niektorých aspektov analýzy a návrhu systému (porovnaním diagramov z jednotlivých etáp).

Návrh vizualizácie je rozdelený do niekoľkých častí. V prvej časti uvediem navrhované rozšírenie sekvenčných diagramov o účinky aspektov. Toto rozšírenie bude navrhnuté z pohľadu analýzy a návrhu systému. V druhej časti budú uvedené úpravy, ktoré boli nutné vykonať z dôvodu toho, že diagramy budú vytvárané prostredím pre jazyk AspectJ na základe statickej analýzy zdrojových súborov vstupného programu (v etape implementácie). Ďalšia časť návrhu vizualizácie bude zahŕňať spôsob zjednodušenia príliš zložitých diagramov, ktorý zlepšuje čitateľnosť diagramov vytvorených pre rozsiahlejšie programy. Nakoniec budú uvedené nevýhody navrhnutej vizualizácie spojené s vizualizáciou niektorých zavedení, deklarácií bodových prierezov a spätným vytváraním aspektov z vizuálnej reprezentácie.

4.1. Rozšírenie sekvenčných diagramov

Sekvenčné diagramy sú definované v UML na zobrazenie interakcie objektov (ľubovolných inštancií - vo všeobecnosti) identifikovaných v etape analýzy a návrhu systému. UML definuje notáciu týchto diagramov a prostredníctvom metamodelu vymedzuje možné vzťahy medzi jednotlivými prvkami sekvenčného diagramu.

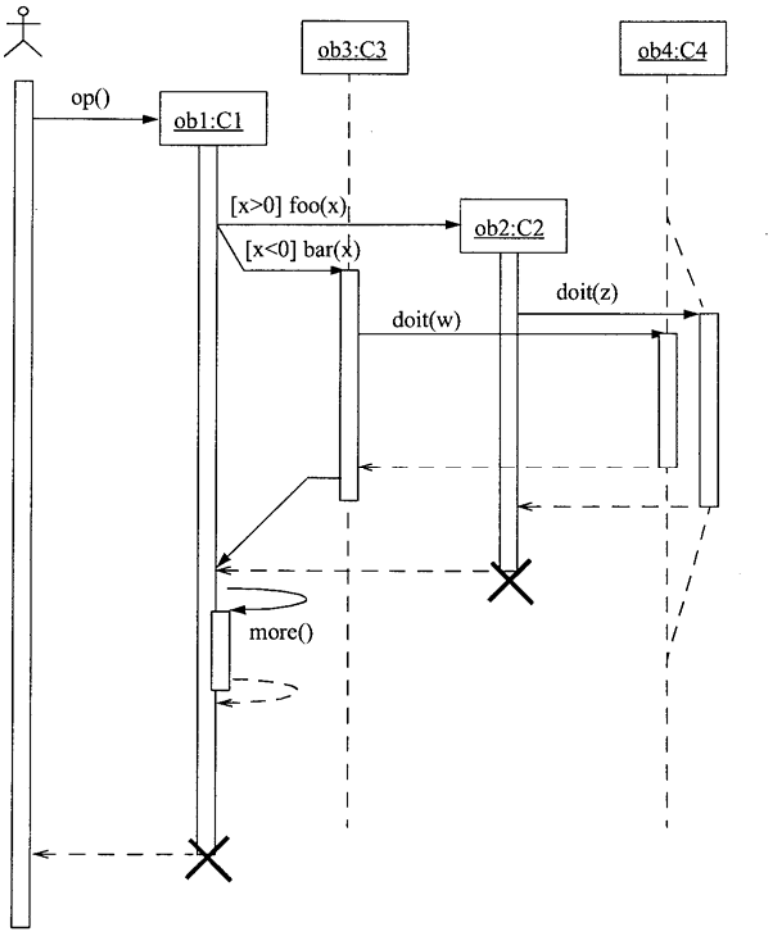
Sú používané dve formy týchto diagramov:

- generická forma
  Zobrazuje všetky možné sekvencie danej operácie. Zachytáva okrem interakcie medzi objektmi aj vetvenie a cykly.
- inštanciálna forma
  Zobrazuje jednu možnú sekvenciu konzistentnú s generickou formou. Znázorňuje interakciu bez vetvení a cyklov.

Sekvenčný diagram má dve dimenzie:

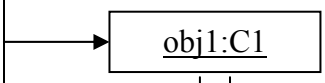
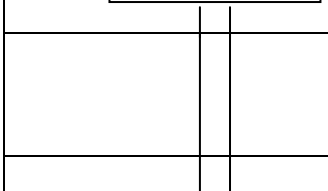
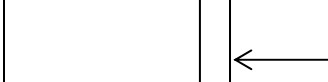
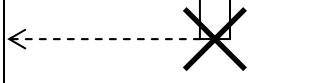
1. vertikálnu - reprezentujúcu čas
2. horizontálnu - reprezentujúcu rôzne objekty systému

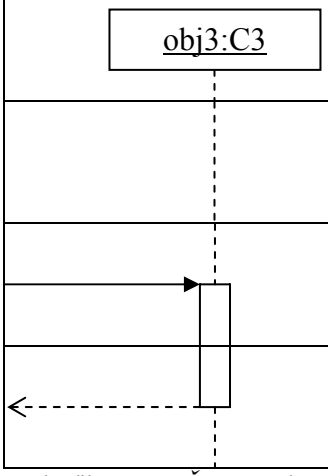
Príklad sekvenčného diagramu je uvedený na obrázku 4.1. Sú na ňom zobrazené jednotlivé objekty (inštancie), ku ktorým patria vertikálne časové línie a jednotlivé interakcie medzi nimi, reprezentované horizontálnymi stimulmi. Usporiadanie inštancií s prislúchajúcimi časovými líniami nie je podstatné (ovplyvnené je predovšetkým snahou o dobrú čitateľnosť diagramu). Stimuly sú usporiadané podľa časovej následnosti zhora nadol.



Obrázok 4.1: Príklad sekvenčného diagramu

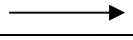
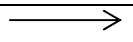
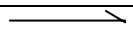
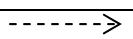
Časová línia umožňuje znázorniť existenciu prislúchajúcej inštancie, dobu, počas ktorej je inštancia aktívna a tiež určuje zdrojovú alebo cieľovú inštanciu stimulu. Spôsob znázornenia týchto skutočností uvádza tabuľka 4.1.

	Vytvorenie inštancie obj1 typu C1 v dôsledku stimulu.
	Naznačenie aktívnej inštancie (inštancie vykonávajúcej nejakú operáciu).
	Znázornená inštancia je zdrojovou inštanciou pre daný stimul.
	Znázornená inštancia je cieľovou inštanciou pre daný stimul.
	Zrušenie znázornenej inštancie a naznačenie vrátenia riadenia pomocou stimulu.

	Znázornenie inštancie obj3 typu C3, ktorá nebola vytvorená počas interakcie znázornenej na sekvenčnom diagrame.
	Naznačenie neaktívnej inštancie.
	Naznačenie aktivácie danej inštancie prostredníctvom stimulu.
	Naznačenie ukončenia aktivácie danej inštancie, pričom inštancia nezanikne (stane sa len neaktívnou).

Tabuľka 4.1: Časová línia sekvenčného diagramu

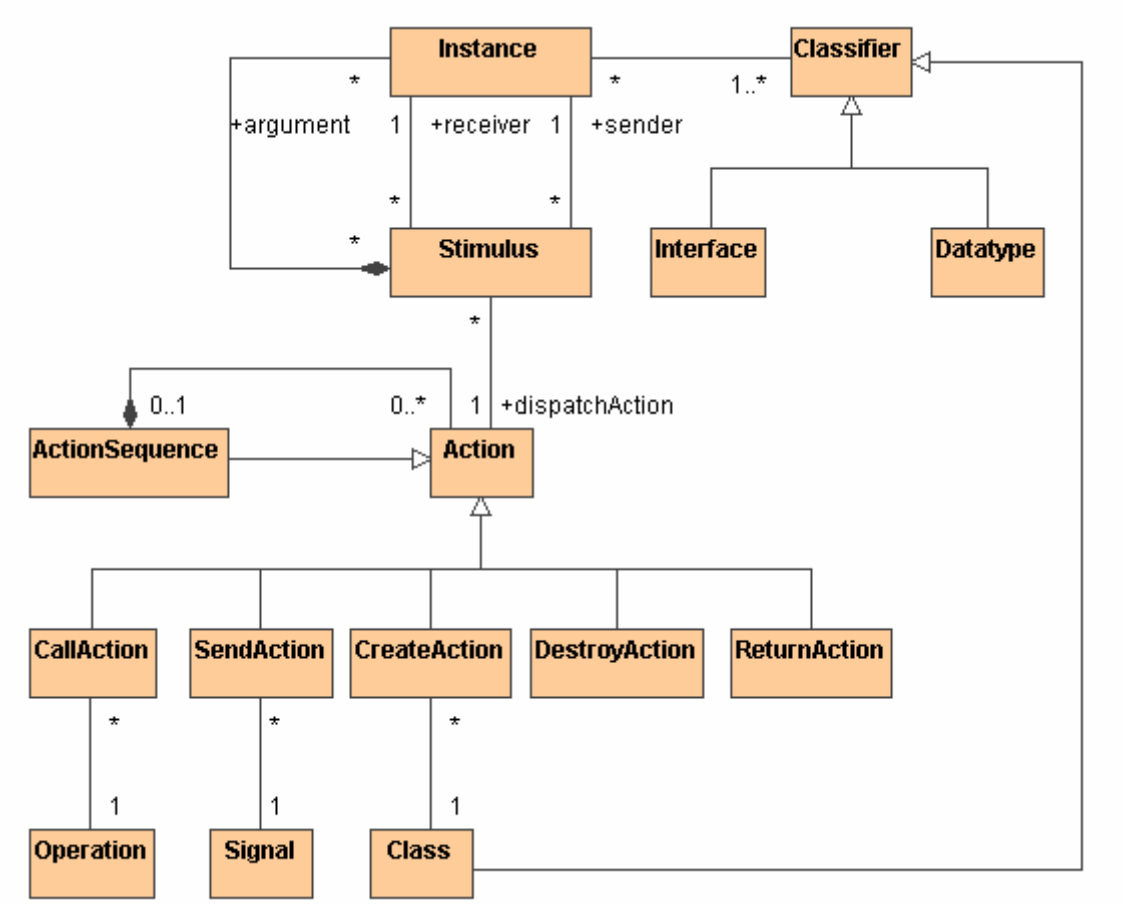
Stimuly znázorňujú interakciu medzi inštanciami. Stimul môže reprezentovať signál poslaný nejakej inštancii, alebo vyvolanie operácie. Tiež môže reprezentovať požiadavku na vytvorenie alebo zrušenie inštancie. Pri stimule sa na diagrame uvádza signatúra vyvolávanej operácie alebo signálu a hodnoty argumentov (konkrétne inštancie) potrebné pre danú operáciu alebo signál. Spôsob označenia rôznych stimulov, ktoré sekvenčný diagram rozlišuje je uvedený v tabuľke 4.2.

	Synchrónny stimul.
	Základný stimul.
	Asynchrónny stimul.
	Návrat z vykonávanej operácie.

Tabuľka 4.2: Rôzne stimuly zobrazené na sekvenčnom diagrame

Na rozšírenie sekvenčných diagramov potrebujeme poznať okrem uvedenej notácie sekvenčných diagramov aj časť metamodelu UML, ktorý sekvenčné diagramy opisuje. Táto časť metamodelu je zobrazená na obrázku 4.2.

Metamodel zachytáva vzťah medzi inštanciou (`Instance`) a stimulom (`Stimulus`). Znázorňuje, že každý stimul má inštanciu, ktorá daný stimul vysiela a inštanciu prijímajúcu stimul. Tiež sú naznačené argumenty (`argument`) pre operáciu alebo signál daného stimulu, ktoré sú tiež inštanciami. Okrem stimulu a inštancie uvedený metamodel znázorňuje aj akciu (`Action`) a klasifikátor (`Classifier`). Klasifikátor reprezentuje typ konkrétnej inštancie. Od klasifikátoru UML odvodzuje triedu (`Class`), rozhranie (`Interface`) a dátový typ (`DataType`). Akcia je zodpovedná za vytvorenie stimulu a bližšie špecifikuje to, čo sa má vykonať danou interakciu. Akcia je ďalej rozšírená konkrétnym druhom akcie. Význam jednotlivých akcií popisuje tabuľka 4.3.

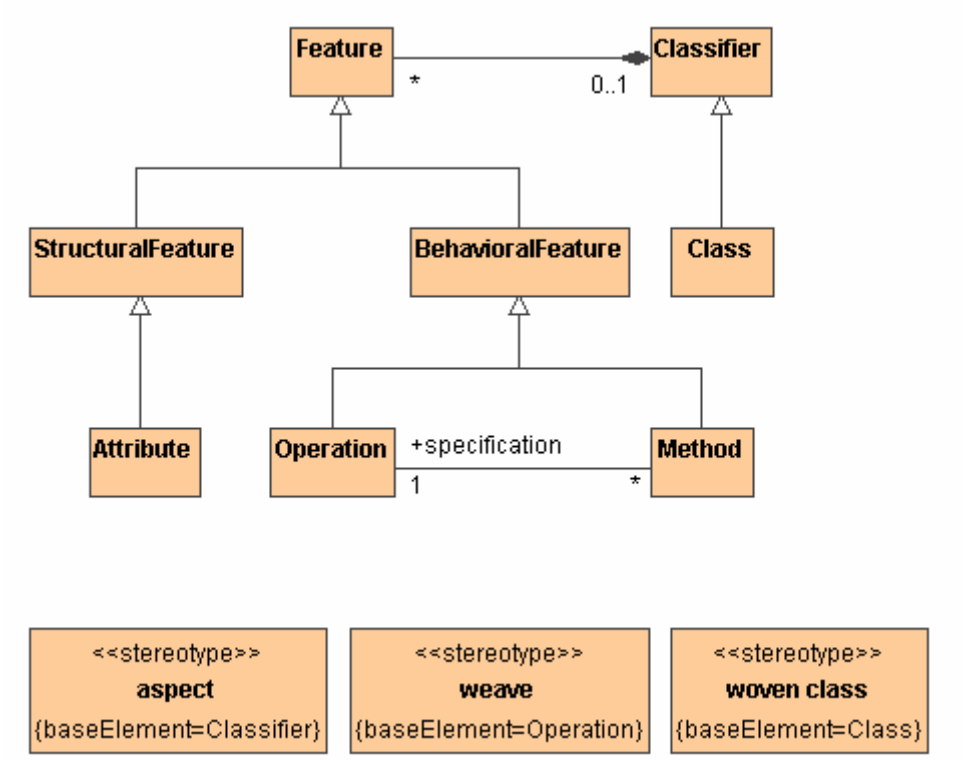


Obrázok 4.2: Časť metamodelu UML popisujúca elementy sekvenčných diagramov

Akcia	**Popis**
`CallAction`	Akcia spôsobí vytvorenie stimulu, ktorý vyvolá operáciu reprezentovanú priradenou metatriedou `Operation`.
`SendAction`	Akcia spôsobí vytvorenie stimulu, ktorý vyšle signál reprezentovaný priradenou metatriedou `Signal`.
`CreateAction`	Akcia vytvárajúca inštanciu vytvorenú na základe priradenej triedy.
`DestroyAction`	Akcia rušiaca inštanciu.
`ReturnAction`	Akcia vracajúca riadenie.
`ActionSequence`	Reprezentuje postupnosť akcií.

Tabuľka 4.3: Význam akcií vytvárajúcich stimul

Z uvedenej časti metamodelu vyplýva, že ako typ inštancie aspektu môžeme použiť len všeobecný klasifikátor, čo je dosť nepresné. Je zrejmé, že aspekt by mal byť podobne ako trieda odvodený od klasifikátora. Toto rozšírenie UML je navrhované aj v [4, 5]. Bolo vytvorené pre ľubovoľný aspektovo-orientovaný jazyk, aby bolo možné identifikovať aspekty už v etape analýzy a návrhu systému. Uvedená práca rozširuje UML pomocou stereotypov. Tieto stereotypy sú uvedené na obrázku 4.3. Tiež je na ňom pre úplnosť uvedená časť hierarchie metatried, ktorú stereotypy rozširujú.

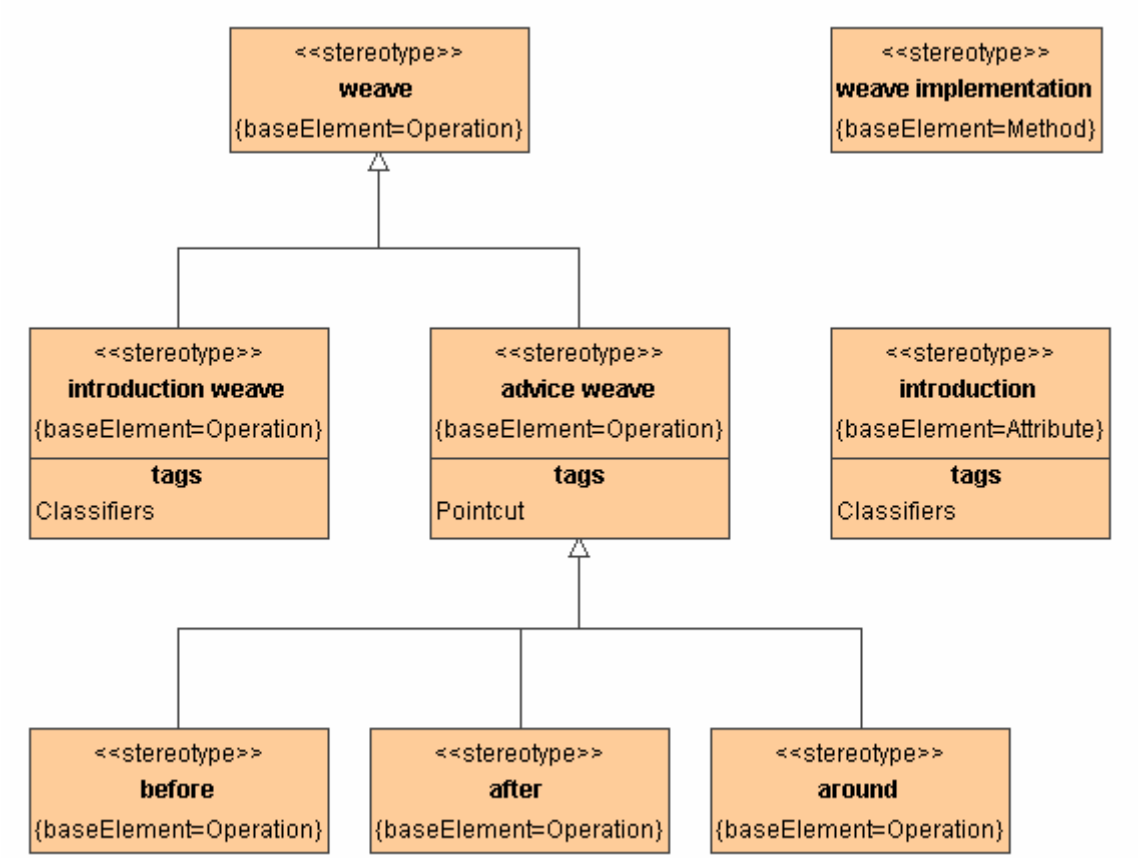


Obrázok 4.3: Existujúce rozšírenie UML o aspekty

Rozšírenie pozostáva z troch stereotypov. Stereotyp aspekt (`aspect`) predstavuje typ pre inštancie aspektov a je odvodený od základnej metatriedy klasifikátor. Takto môže mať aspekt vlastnosti - podobne ako ľubovolný klasifikátor. Vlastnosti (`Feature`) sa delia na štrukturálne vlastnosti (`StructuralFeature`) a vlastnosti definujúce správanie (`BehavioralFeature`). Štrukturálnou vlastnosťou je napríklad atribút (`Attribute`) a vlastnosťou definujúcou správanie je metóda (`Method`) a operácia (`Operation`), pričom operácia je špecifikáciou toho, čo má metóda vykonať. Čiže odvodením od klasifikátora je popísané to, že inštancie aspektu majú, podobne ako objekty (inštancie tried), stav definovaný atribútmi a správanie (zmena stavu) definované operáciami (implementované metódami).

Aspekty naviac môžu dodefinovať správanie a vlastnosti ostatným objektom (inštanciám) systému. Dodefinovanie správania reprezentuje stereotyp `weave`. Tento je odvodený od metatriedy operácia. Na reprezentáciu dodefinovania stavu (napr. zavedenie atribútu v AspectJ) nebol v uvedenej práci vytvorený žiaden stereotyp. Posledným navrhnutým stereotypom je stereotyp `woven class`. Tento predstavuje triedu vzniknutú zlúčením aspektu a triedy, ktorej správanie alebo stav aspekt dodefinuje. Spôsob zlúčenia závisí od konkrétneho programovacieho jazyka a konkrétneho zlučovacieho programu.

Pretože uvedené rozšírenie bolo vytvorené pre ľubovolný aspektovo-orientovaný jazyk, neopisuje úplne presne aspekty používané v AspectJ. Autori tohto rozšírenia navrhli, aby pre konkrétny aspektovo-orientovaný jazyk vznikli nové stereotypy odvodené od stereotypu `weave`. Takto navrhnuté rozšírenie pre jazyk AspectJ je naznačené na obrázku 4.4.



Obrázok 4.4: Navrhované rozšírenie UML pre AspectJ

Sú na ňom uvedené stereotypy `introduction weave` a `advice weave` odvodené od stereotypu `weave`. Stereotyp `introduction weave` reprezentuje zavedenie operácie (vo fáze implementácie je to zavedenie metódy alebo konštruktora). Stereotyp má naviac definovanú značku `Classifiers`, ktorá špecifikuje klasifikátory, ktorým je daná operácia dodefinovaná. Stereotyp `advice weave` predstavuje špecifikáciu rád. Má definovanú značku `Pointcut`, ktorá určuje, kde sa má daná rada vykonať. Od stereotypu `advice weave` sú ďalej odvodené stereotypy pre jednotlivé druhy rád, ktoré jazyk AspecJ definuje. Sú to stereotypy `before`, `after`, `around`.

Naviac k stereotypom odvodeným od stereotypu `weave`, bol vytvorený stereotyp `weave implementation` a `introduction`. Stereotyp `weave implementation` je odvodený od základnej metatriedy metóda. Podobne ako metóda je implementáciou operácie (obrázok 4.3), reprezentuje stereotyp `weave implementation` implementáciu stereotypu `weave`. Základnou metatriedou stereotypu `introduction` je atribút. Tento stereotyp predstavuje zavedenie atribútu do klasifikátorov špecifikovaných značkou `Classifiers`.

Na sekvenčných diagramoch môžeme naznačiť práve účinky stereotypov odvodených od stereotypu `weave` (zabezpečujúcich dodefinovanie správania).

Tieto účinky môžeme rozdeliť na:

1. **vyvolanie zavedenia** (stereotyp `introduction weave`)
   Presnejšie je to vyvolanie operácie objektu (inštancie), ktorá bola dodefinovaná aspektom.
2. **vyvolanie rady** (stereotyp `advice weave`)
   Toto vyvolanie môže spôsobiť predchádzajúca operácia (volanie metódy, vytvorenie objektu, modifikácia alebo čítanie hodnoty atribútu ...).

Vyvolanie zavedenia alebo rady spôsobia stimuly vytvorené nejakou akciou. V prípade volania je touto akciou `CallAction` (obrázok 4.2). Táto vytvára stimul, ktorý spôsobí vyvolanie operácie. Pretože stereotyp reprezentujúci zavedenie je odvodený od metatriedy operácia, nemusíme túto akciu meniť (bližšie špecifikovať). Stimul, ktorý spôsobí vyvolanie rady, je vytvorený akciou, ktorú dodá zlučovací program zabezpečujúci zlúčenie triedy a aspektu, ktorý dodefinuje správanie danej triedy. Táto akcia musí mať väzbu na vyvolávanú radu a pretože stereotyp rady je v navrhovanom rozšírení odvodený od základnej metatriedy operácia, môžeme aj v tomto prípade túto akciu špecifikovať metatriedou `CallAction`.

Ďalej treba určiť inštanciu vysielajúcu stimul a inštanciu prijímajúcu stimul. Vysielajúca inštancia je v prípade vyvolania zavedenia inštancia, ktorá vyvolanie spôsobila a v prípade vyvolania rady je to inštancia, ktorá bola aktívna ako posledná pred vyvolaním rady. Pri určovaní cieľovej inštancie máme dve možnosti. Buď bude cieľovou inštanciou inštancia, ktorej je dodefinované správanie aspektom, alebo inštancia aspektu, ktorý je zodpovedný za dodefinovanie správania. V podstate by to mala byť tá inštancia, ktorej stav priamo mení vyvolávaná operácia (rada alebo zavedenie). To je z pohľadu implementácie danej operácie inštancia, ktorej hodnoty atribútov môžeme meniť priamo.
Týmto spôsobom môžeme určiť cieľovú inštanciu stimulu z nasledujúceho príkladu:

```
public class Main
{
	public int x = 0;
	public void inc()
		{ ++x; };
	public static void main(String[] arg)
	{
		Main m1 = new Main();
		m1.inc();
		m1.inc();
		m1.print();
		Main m2 = new Main();
		m2.inc();
		m2.print();
	};
};
aspect Trace
{
	private int x = 0;
	public void Main.print()
		{ System.out.println("x = " + x); };
	pointcut increasing() : call(void Main.inc());
	after() : increasing()
	{
		++x;
		System.out.println("increased " + x + "x");
	};
};
```

V tomto príklade je definovaná trieda `Main` a aspekt `Trace`. V triede `Main` je definovaný celočíselný atribút `x` inicializovaný hodnotou `0` pri vytvorení inštancie triedy, metóda `inc`, ktorá zvýši hodnotu atribútu a o `1` a statická metóda `main`, ktorá vytvára inštancie triedy `Main` a volá jej metódy. Aspekt `Trace` tiež definuje celočíselný atribút `x` inicializovaný hodnotou `0`, naviac zavádza metódu `print` do triedy `Main`, ktorá vypíše hodnotu atribútu `x` a definuje radu `after`, vyvolávanú po každom zvýšení hodnoty atribútu `x` objektu triedy `Main`, ktorá zvýši hodnotu atribútu `x` inštancie aspektu o `1` a vypíše hodnotu tohto atribútu (počet volaní metódy `inc` pre všetky inštancie triedy `Main`).
Výsledok príkladu je:

```
increased 1x
increased 2x
x = 2
increased 3x
x = 1
```

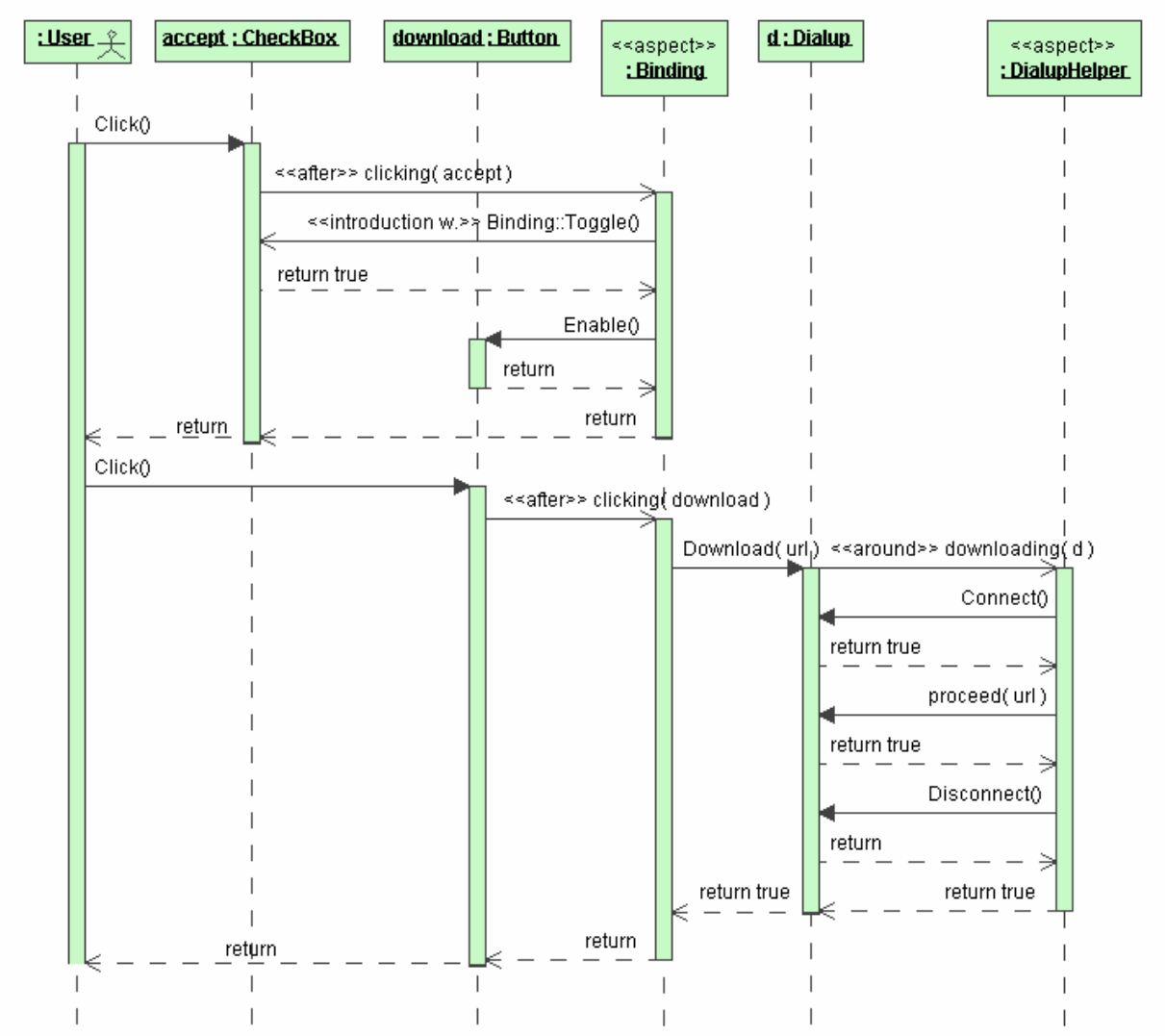
Z tohto príkladu je vidieť, že aj keď sú zavedenie a rada definované v tom istom aspekte, pri svojom vykonávaní pristupujú k rôznym inštanciám (tj. atribút `this` ukazuje na rôzne inštancie). Pri vykonaní zavedenia je vypísaná hodnota atribútu `x` objektu triedy `Main` a pri vykonaní rady je zvýšená a vypísaná hodnota atribútu `x` inštancie aspektu `Trace`. Čiže cieľovou inštanciou stimulu, ktorý spôsobí vyvolanie zavedenia je inštancia, ktorej správanie je dodefinované aspektom a cieľovou inštanciou stimulu, ktorý spôsobí vyvolanie rady je inštancia aspektu, ktorý danú radu definuje.

Nakoniec treba pre zobrazenie sekvenčného diagramu s navrhovaným rozšírením o aspekty určiť označenie inštancií aspektov a označenie stimulov vyvolávajúcich rady a zavedenia. Pri označení inštancií aspektov bude okrem názvu inštancie a klasifikátora (aspektu) uvedený aj vytvorený stereotyp reprezentujúci aspekt (stereotyp `aspect`). Stimul vyvolávajúci zavedenie bude označený pomenovaním operácie, ktoré dané zavedenie dodefinováva a hodnotami parametrov danej operácie. Pretože cieľová inštancia takéhoto stimulu je inštancia, ktorej bola daná operácia dodefinovaná a nie inštancia aspektu, ktorý zavedenie definuje treba na úplné identifikovanie zavedenia pridať k názvu operácie aj názov aspektu, v ktorom je zavedenie definované. Okrem toho bude k označeniu stimulu pridaný aj stereotyp reprezentujúci zavedenie (stereotyp `introduction weave`). Pretože rady nie sú v AspectJ pomenované a sú v podstate rozlišované bodovým prierezom, bude v označení stimulu vyvolávajúceho radu uvedený bodový prierez danej rady (buď jeho názov alebo popis). K označeniu budú pridané aj hodnoty vstupných parametrov rady a tiež stereotyp reprezentujúci konkrétny typ rady (stereotyp `before`, `after`, `around`).

Príklad sekvenčného diagramu so zobrazením účinku aspektov je na obrázku 4.5. Uvádza spôsob zobrazenia inštancie aspektu, stimulu vyvolávajúceho zavedenie a stimuly vyvolávajúce radu `after` (podobne by bola zobrazená aj rada `before`) a `around`. Príklad predstavuje sekvenčný diagram vytvorený pre fiktívny inštalačný program, ktorý umožňuje inštalovať aplikáciu z internetu prostredníctvom modemu. Pred nahratím aplikácie sa zobrazí používateľovi formulár s licenčnými podmienkami a tlačítkami `accept` a `download`. Tlačítko `accept` (trieda `CheckBox`) slúži na potvrdenie licenčných podmienok a tlačítko download (trieda `Button`) na spustenie sťahovania aplikácie z internetu. Požaduje sa, aby tlačítko `download` bolo prístupné až po potvrdení licenčných podmienok používateľom a aby bolo zabezpečené, že počas sťahovania aplikácie bude počítač, na ktorom je inštalačný program spustený pripojený na internet.

Prípad, keď stiahnutie inštalovanej aplikácie prebehne úspešne je prezentovaný uvedeným diagramom. Najprv je stlačené tlačítko `accept` (operácia `Click`). Po vykonaní tejto operácie je vyvolaná rada `after` inštancie aspektu `Binding`, pričom prislúchajúci stimul je označený stereotypom rady (`after`), názvom bodového prierezu (`clicking`) a hodnotou vstupného parametru (`accept`). Rada najskôr zmení stav tlačítka vyvolaním operácie `Toggle` dodefinovanej triede objektu `accept`. Príslušný stimul je označený stereotypom zavedenia (`introduction weave`), názvom dodefinovanej operácie (`Toggle`) a názvom aspektu (`Binding`), v ktorom je zavedenie definované. Operácia `Toggle` vracia hodnotu `true`, ak je tlačítko `accept` zaškrtnuté a v tom prípade rada odblokuje tlačítko `download`.

Po stlačení tlačítka `download` je znovu vykonaná rada aspektu `Binding`. Táto rada spôsobí volanie operácie `Download` slúžiacej na stiahnutie inštalovanej aplikácie z internetu. Na zabezpečenie pripojenia k internetu počas sťahovania slúži rada `around` aspektu `DialupHelper`. Tá je vykonaná pred operáciou `Download` a tiež po tejto operácii. Pred vykonaním operácie `Download` vykoná táto rada pripojenie k internetu (operácia `Connect`) a po vykonaní operácie `Download` odpojenie od internetu (operácia `Disconnect`). Vykonanie operácie `Download` z rady `around` aspektu `DialupHelper` je naznačené stimulom s označením `proceed` (v AspectJ na to slúži príkaz `proceed`).



Obrázok 4.5: Príklad sekvenčného diagramu so zobrazením aspektov

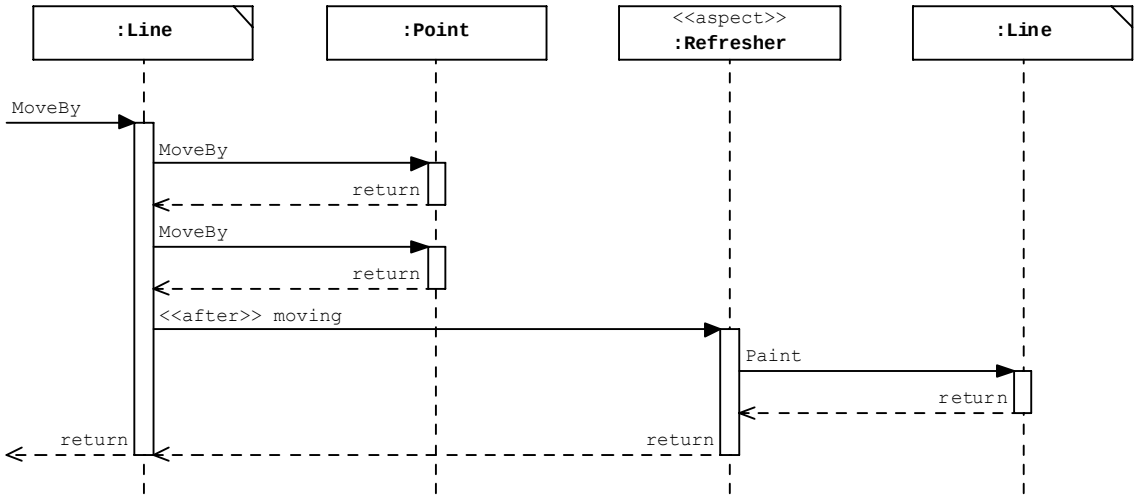
4.2. Úpravy pre automatické vytváranie

Z dôvodu, že sekvenčné diagramy majú byť zobrazované na základe statickej analýzy programu v AspectJ, treba vykonať niektoré modifikácie, ktoré nebudú v súlade s UML. Niektoré z nich sú vykonané kvôli tomu, že zdrojový program neposkytuje taký nadhľad na systém, aký má návrhár systému. Ďalšie sú vykonané na zvýšenie prehľadnosti zobrazených diagramov.

V prvom rade si treba uvedomiť, že keď vytvárame diagramy zo zdrojových súborov, tieto diagramy budú zachytávať systém z pohľadu implementácie. To znamená, že v zdrojových súboroch nájdeme len názvy implementačných tried (aspektov) a tie nemusia zodpovedať názvom tried (aspektov) vytvorených v návrhu. Taktiež sa v etape implementácie prechádza od operácii (rád, zavedení) k metódam (implementáciam rád, implementáciam zavedení), pričom jedna operácia (rada, zavedenie) môže byť implementovaná viacerými metódami (implementáciami rád, implementáciami zavedení). Čiže sekvenčný diagram zobrazený zo zdrojového súboru programu bude zachytávať inštancie implementačných tried a aspektov a zobrazené stimuly budú vyvolávať metódy (implementácie rád, implementácie zavedení).

V prípade špecifikácie alebo návrhu systému bývajú sekvenčné diagramy vytvorené pre nejaký prípad použitia systému. Takýto prípad použitia býva transformovaný do jednej alebo viacerých operácii, ktoré sú zase v etape implementácie transformované do metód. Preto by sme pri generovaní sekvenčných diagramov z metód zdrojových súborov programu mali dostať diagram, ktorý bude aspoň čiastočne zodpovedať pôvodnému diagramu vytvorenému v etape špecifikácie alebo návrhu systému. Takto generované diagramy v podstate znázorňujú tok riadenia metód programu.

Zmeny notácie sekvenčného diagramu budú vysvetlené na niekoľkých prípadoch. Prvý z nich je na obrázku 4.6. Znázorňuje tok riadenia metódy `MoveBy`. Táto metóda slúži na presunutie priamky (trieda `Line`). Sú z nej volané metódy `MoveBy` na presunutie začiatočného a koncového bodu priamky a po vykonaní metódy `MoveBy` triedy `Line` sa vyvolá rada aspektu `Refresher`, ktorá zabezpečí prekreslenie priamky (operácia `Paint` triedy `Line`).



Obrázok 4.6: Príklad sekvenčného diagramu generovaného pre metódu programu

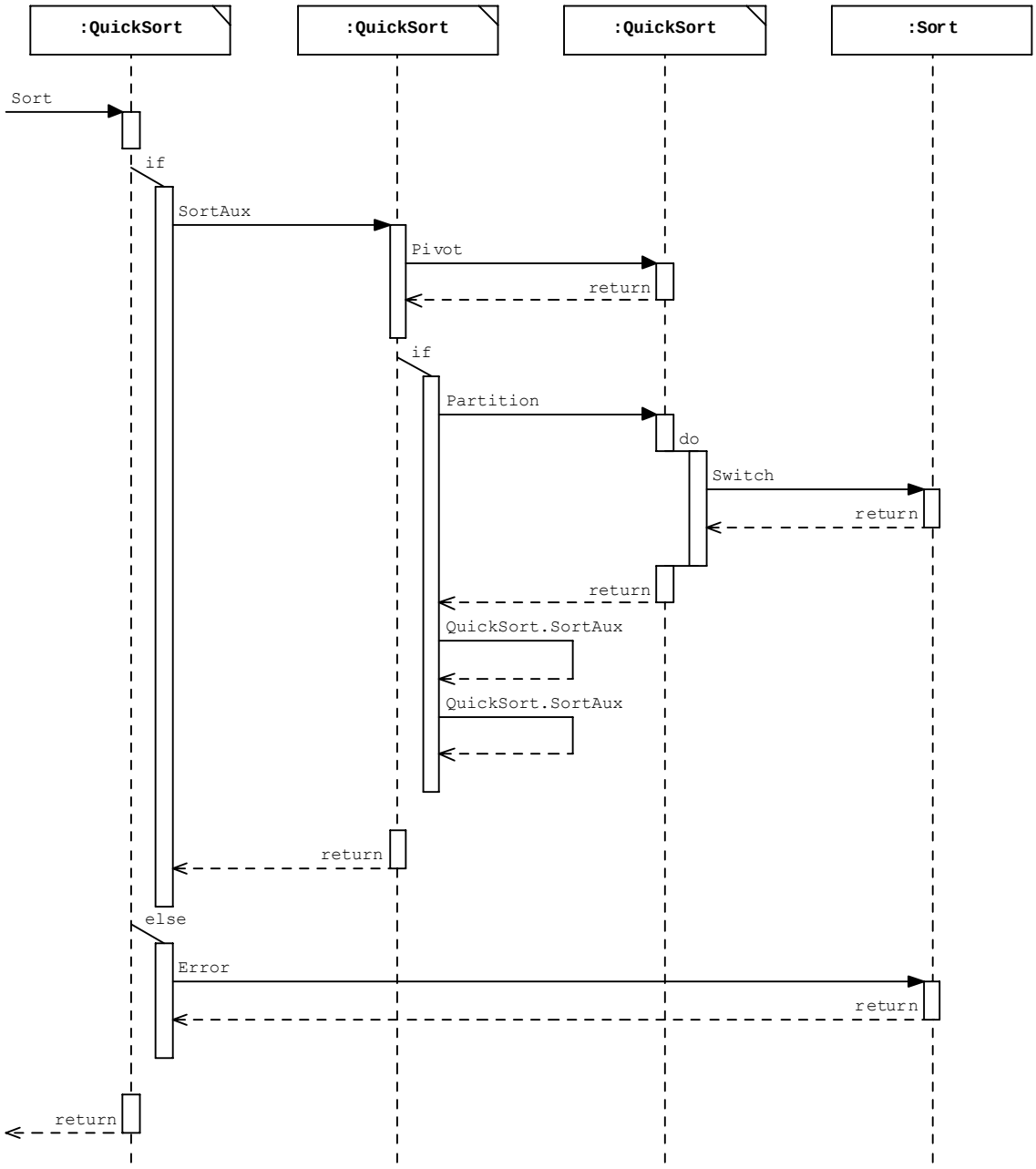
Prvá zmena, ktorú uvedený príklad znázorňuje sa týka názvu časovej línie. Tento má pozostávať z mena inštancie a názvu klasifikátora oddelených dvojbodkou. Celý názov má byť podčiarknutý, čo naznačuje, že ide o inštanciu. Pretože názov inštancie nie je možné získať zo zdrojových súborov programu, je názov časovej línie tvorený len názvom klasifikátora (triedy alebo aspektu). Naviac sa zo statickej analýzy nedajú vo všeobecnosti rozlíšiť jednotlivé inštancie rovnakého klasifikátora, čiže na takomto diagrame sú zobrazené skôr volania metód (implementácie rady, implementácie zavedenia) klasifikátorov ako konkrétnej inštancie, preto názov časovej línie nie je podčiarknutý. Z toho vyplýva aj to, že nie je označované vytvorenie a rušenie inštancií.

Ďalšia zmena sa týka označenie stimulov. Názov stimulu má pozostávať pri volaní operácie z názvu volanej operácie a hodnôt parametrov. V tomto prípade pozostáva len z názvu volanej metódy (implementácie rady, implementácie zavedenia) bez uvedenia hodnôt parametrov. Hodnoty parametrov sú vynechané preto, lebo sa nedajú zistiť zo statickej analýzy. Z rovnakého dôvodu je vynechaná aj návratová hodnota metódy. Všetky stimuly prislúchajúce volaniam sú naviac označené ako synchrónne (sú synchrónne vzhľadom na viazané metódy).

Kvôli väčšej prehľadnosti sú všetky volania diagramu zobrazené rovnakým smerom (zľava do prava) a návraty z volania opačným (zprava do ľava), čo v niektorých prípadoch vyžaduje duplikáciu časovej línie. Označenie duplikácie je znázornené na obrázku pri časových líniach triedy `Line`. Poslednou zmenou je, že prvý stimul diagramu nemá zdrojovú časovú líniu a posledný nemá cieľovú časovú líniu. To je dôsledkom toho, že každý diagram generovaný zo zdrojových súborov AspectJ programu zobrazuje tok riadenia nejakej počiatočnej metódy, pričom nevieme jednoznačne určiť inštanciu (názov ani jej klasifikátor), z ktorej je počiatočná metóda v programe volaná.

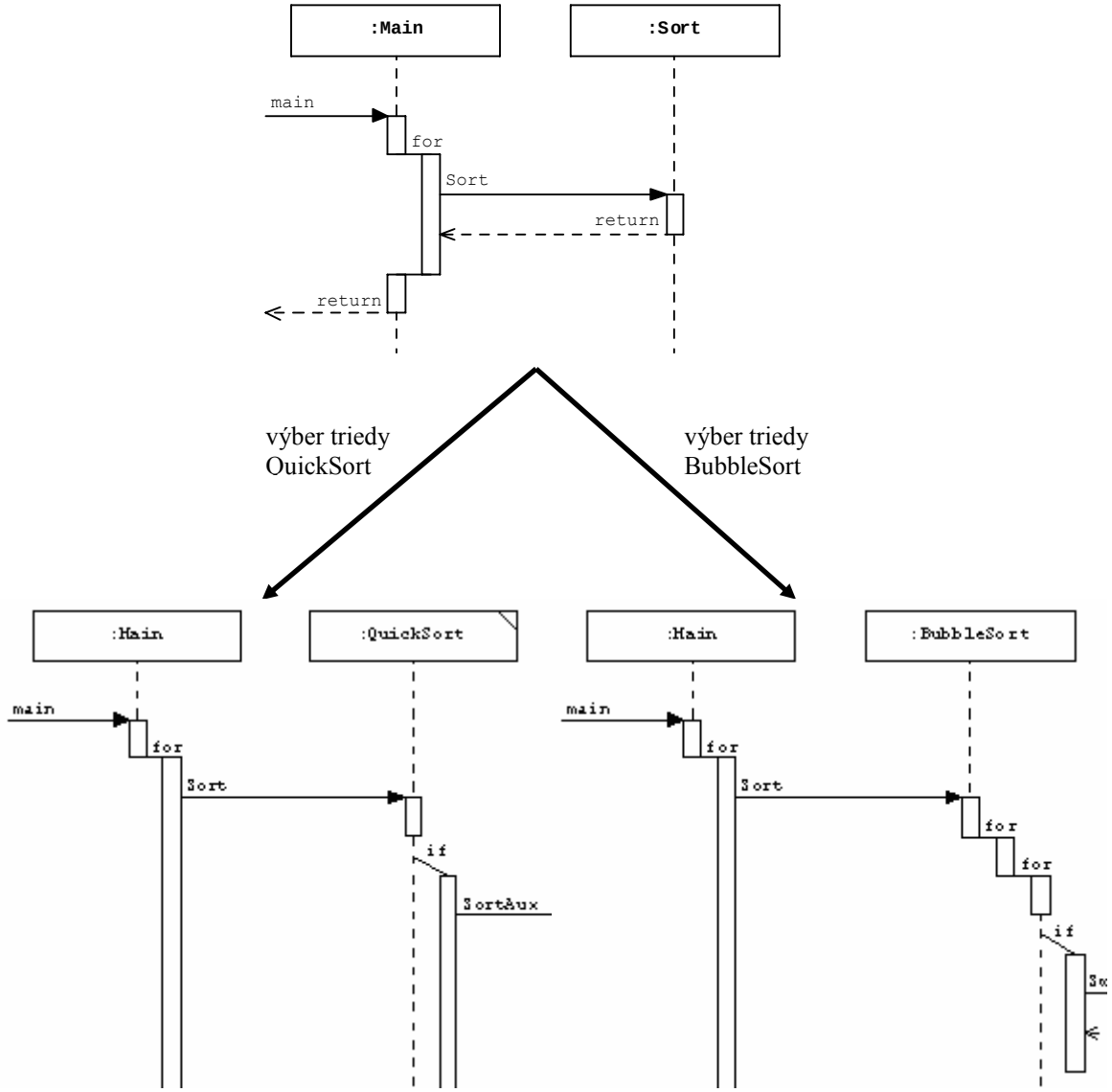
Druhý príklad generovaného sekvenčného diagramu je na obrázku 4.7. Tento príklad zobrazuje metódu `Sort` triedy `QuickSort` (odvodenej od triedy `Sort`). Táto na začiatku skontroluje, či vstupné pole obsahuje aspoň jeden prvok a ak áno zavolá metódu `SortAux`. V prípade, že je vstupné pole prázdne, dôjde k výpisu chybovej hlášky (operácia `Error` triedy `Sort`). Operácia `SortAux` získa pivot poľa (operácia `Pivot`). V prípade, že sa pivot poľa podarilo získať (pole nie je usporiadané), volá sa operácia `Partition`, ktorá pole rozdelí podľa pivota na dve časti. Potom sa rekurzívne volá operácia `SortAux` na ľavú časť poľa a na pravú časť poľa.

Z tohto príkladu je vidieť predovšetkým povahu diagramov generovaných na základe statickej analýzy vstupného programu. Tieto znázorňujú všetky možné varianty vykonávania zobrazenej metódy. Ide v podstate o generickú formu sekvenčných diagramov, čo je v protiklade so sekvenčnými diagramami vytváranými v etape špecifikácie a návrhu systému, ktoré sú najčastejšie v inštanciálnej forme. Na to, aby sme zobrazili všetky varianty vykonávania, potrebujeme jednotlivé varianty od seba oddeliť. Na to sekvenčné diagramy v UML používajú naznačenie alternatívneho vykonávania (rozdelenie časovej línie na dve paralelné) a opakovania stimulov. Notácia zvolená v UML však pretpokladá, že diagramy vytvára človek, ktorý vytvorený diagram upraví tak, aby bol dobre čitateľný. Pri dodržaní notácie UML, by sa automaticky generované diagramy stávali neprehľadnými. Preto bola zvolená prehľadnejšia notácia, naznačená na obrázku 4.7: posunutie časovej línie pre vnorený blok opakovania alebo podmieneného vykonávania a zobrazenie jednotlivých vetiev podmieneného vykonávania pod sebou. Obrázok tiež zachytáva spôsob naznačenia rekurzívneho volania metódy (volanie metódy `SortAux` z metódy `SortAux`). Rovnako ako je znázornená priama rekurzia na obrázku, by bola znázornená aj rekurzia nepriama.



Obrázok 4.7: Príklad sekvenčného diagramu generovaného pre metódu programu

Posledný príklad generovaných diagramov je znázornený na obrázku 4.8. Tento pozostáva z troch diagramov. Každý z nich je vytvorený pre rovnakú metódu, ktorou je metóda `main` aplikácie na triedenie polí. V tejto metóde je pole utriedené rôznymi algoritmami, ktoré sú implementované v triedach odvodených od triedy `Sort`. Objekty týchto tried sú uložené v poli a v metóde `main` je postupne volaná metóda `Sort` týchto objektov, ktorá utriedi vstupné neutriedené pole konkrétnym algoritmom. V jednotlivých iteráciach cyklu je volaná operácia `Sort` rôznych objektov, ktoré prislúchajú rôznym triedam. Zo statickej analýzy sa dá určiť len to, že tieto objekty sú objektmi triedy `Sort` akebo triedy odvodenej od triedy `Sort`. Preto prostredie vytvorené na vizualizáciu prostredníctvom sekvenčných diagramov musí umožniť používateľovi, aby mohol určiť konkrétnu triedu cieľového objektu volania a takto poskytuje používateľovi možnosť zobraziť všetky možné varianty sekvenčného diagramu danej počiatočnej metódy.



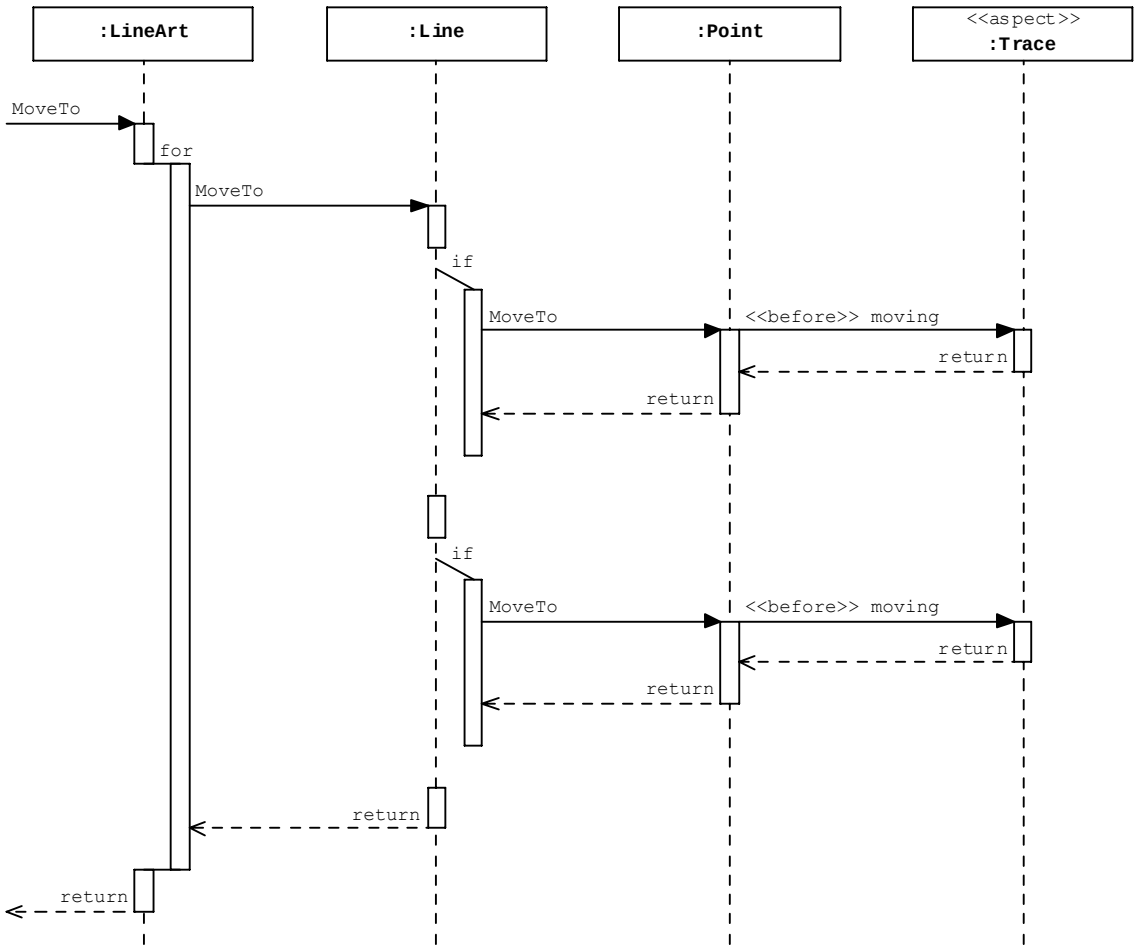
Obrázok 4.8: Zmena diagramu špecifikovaním triedy volaného objektu

4.3. Návrh filtrovania elementov diagramu

Medzi nevýhody navrhnutého zobrazenia aspektov patrí predovšetkým jeho neprehľadnosť pri znázornení metód programu so zložitým tokom riadenia. Je zrejmé, že zobrazenie hlavnej metódy (metóda `main`) môže byť veľmi rozsiahle aj pre relatívne jednoduché programy. Preto je potrebné vytvoriť spôsob, ktorým možno príliš komplikované diagramy zjednodušiť, pričom by sa zachovali väzby, ktoré sú v danom zobrazení podstatné. O tom, ktoré väzby diagramu sú dôležité a ktoré nie sa nedá rozhodnúť automaticky, preto bude toto rozhodnutie ponechané na používateľa. Úlohou je poskytnúť mu dostatočne jednoduchý a efektívny spôsob na určenie toho, čo má byť na diagrame znázornené.

Možnosti filtrovania

Prvým krokom návrhu filtrovania je identifikovanie elementov diagramu, ktoré môžu byť zo zobrazenia vynechané a určenie spôsobu naznačenia vynechaných elementov. Jednotlivé možnosti filtrovania budú vysvetlené na spoločnom diagrame, ktorý je na obrázku 4.9.



Obrázok 4.9: Pôvodný sekvenčný diagram

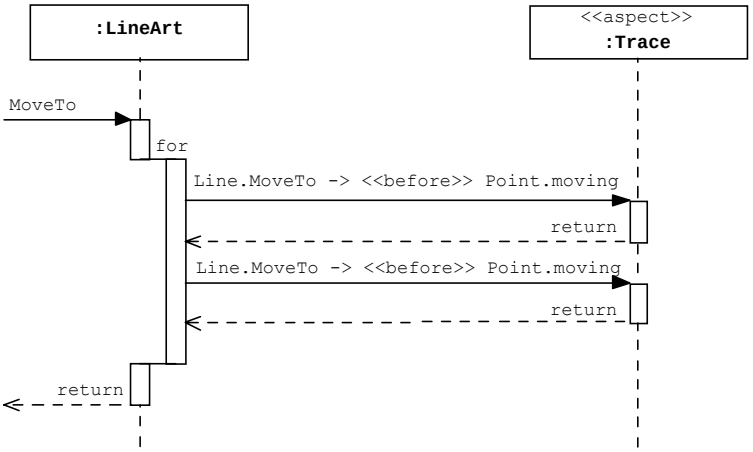
Je na ňom znázornené volanie metódy `MoveTo` triedy `LineArt`. Z tejto triedy je v cykle volaná metóda `MoveTo` triedy `Line`, ktorá následne volá dvakrát metódu `MoveTo` triedy `Point`, pričom každé takéto volanie sa nachádza vo vetve podmieneného príkazu `if`. Na volanie `MoveTo` triedy point je viazaná rada `before` aspektu `Trace`.

Vynechanie časovej línie

Prvým identifikovaným elementom, ktorý môže byť vynechaný pri zobrazení rozšíreného sekvenčného diagramu je vynechanie celej vertikálnej časovej línie. Zodpovedá to vynechaniu pre zobrazenie nedôležitej triedy. Otázkou je, čo robiť s volaniami metód vynechanej triedy.

Jednou možnosťou je vylúčiť celý tok riadenia týchto metód zo zobrazeného diagramu. Toto riešenie by ale v mnohých prípadoch príliš ochudobnilo diagram. Z tohto pohľadu je lepším riešením, keď sa volania vykonané v metódach vynechaných tried presunú o úroveň vyššie v toku riadenia (do metódy, z ktorej bola metóda vynechanej triedy volaná).

Mohlo by sa zdať, že v tomto prípade bude zjednodušenie pôvodného diagramu nepodstatné. Je to však pravda len v prípade, že sa vynechá len jedna trieda z pôvodného diagramu. Ak sa vynechá takýmto spôsobom viac tried so vzájomnými vzťahmi (vzájomnými volaniami metód), dôjde k žiadanému zjednodušeniu, pričom sa zachovajú dôležité vzťahy znázornené na pôvodnom diagrame. Príklad takto filtrovaného diagramu je na obrázku 4.10. Tento vznikol vynechaním časovej línie triedy `Line` a `Point` pôvodného diagramu (obrázok 4.9).

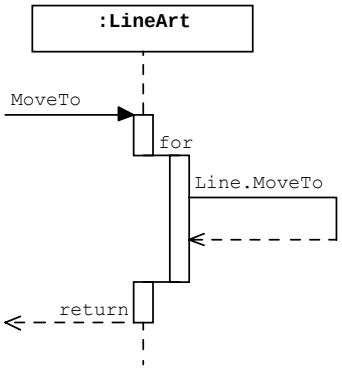


Obrázok 4.10: Vynechanie časovej línie

V tomto prípade bol tok riadenia metódy `MoveTo` triedy `Line` redukovaný na vyznačené volania rady `before` aspektu `Trace`. Z obrázku je tiež vidieť spôsob pomenovania takejto väzby, pri ktorom nevystačíme s jednoduchým názvom ako v pôvodnom diagrame, ale musíme označiť, že sa za touto väzbou ukrýva pôvodný tok riadenia, ktorý začína volaním metódy `MoveTo` triedy `Line` a končí radou `before` aspektu `Trace` viazanej na metódu triedy `Point`.

Vynechanie toku riadenia

Ďalším identifikovaným elementom, ktorý môže byť z diagramu vynechaný je celý tok riadenia začínajúci volaním určenej metódy a končiaci návratom z tohto volania. Zodpovedá to vynechaniu pre zobrazenie nedôležitej metódy. Vynechanie dátového toku môže spôsobiť aj vylúčenie časovej línie, ktorá bola zobrazená len kvôli tomuto toku riadenia. Príklad diagramu s vynechaným tokom riadenia je na obrázku 4.11. Vznikol vynechaním toku riadenia metódy `MoveTo` triedy `Line` z pôvodného diagramu (obrázok 4.9).



Obrázok 4.11: Vynechanie toku riadenia

Z obrázku je vidieť okrem spôsobu naznačenia filtrovaného toku riadenia aj jeho pomenovanie, ktoré označuje začiatok chýbajúceho toku (počiatočnú metódu). V tomto prípade vynechanie toku riadenia spôsobilo aj vynechanie časovej línie triedy `Line`, triedy `Point` a aspektu `Trace`, ktoré už nemalo zmysel zobrazovať.

Vynechanie bloku príkazov

Posledným identifikovaným prvkom diagramu, ktorý môže byť vynechaný je blok príkazov v danej metóde. Myslí sa tým vynechanie vetvy zobrazovaného podmieneného príkazu, celého podmieneného príkazu (všetkých vetví) alebo vynechanie zobrazenia príkazu cyklu. V prípade filtrovania nejakého bloku by sa zo zobrazenia diagramu vynechali aj všetky vnorené bloky a príslušné volania (celé toky riadenia týchto volaní), vykonané z nezobrazovaných blokov.

Použitie tohto spôsobu filtrovania rozšíreného sekvenčného diagramu by malo význam len v prípade, ak by vstupný program obsahoval príliš dlhé a komplikované metódy (metódy s veľkým počtom príkazov volania a veľkou hĺbkou vnorenia syntaktických blokov). Vytváranie takýchto metód (funkcií) sa v programovaní vo všeobecnosti neodporúča a považuje sa za zlý programátorský štýl, pretože bráni dobrej zrozumiteľnosti zdrojových súborov programu a prináša väčšiu mieru chýb. Preto sa predpokladá, že pri filtrovaní elementov diagramu vystačíme s prvými dvoma spomínanými možnosťami a túto možnosť nebudeme ďalej uvažovať.

Určenie elementov

Druhým krokom návrhu filtrovania je určenie spôsobu, ktorým používateľ špecifikuje prvky diagramu, ktoré majú byť zo zobrazenia vynechané. Ako bolo spomenuté, treba určiť buď triedy prislúchajúce vynechávaným časovým líniám diagramu alebo metódy, ktorými začínajú vynechávané toky riadenia.

Najprv treba určiť množinu atribútov (vlastností), ktorými sú tieto prvky určené v zdrojovom programe a pomocou ktorých by sa dali vyberať.

Pre triedu je to:

- identifikátor - pomenovanie triedy
- umiestnenie - trieda alebo balík, v ktorom je daná trieda definovaná
- nadtrieda - názov triedy, ktorú daná trieda rozširuje
- rozhrania - názvy rozhraní, ktoré daná trieda implementuje
- modifikátory - prípustné modifikátory triedy

Pre metódu je to:

- identifikátor - pomenovanie metódy
- umiestnenie - trieda, v ktorej je metóda definovaná
- modifikátory - prípustné modifikátory metódy
- návratová hodnota - typ návratovej hodnoty metódy
- parametre - typy vstupných parametrov

Vymedzením hodnôt týchto atribútov môže používateľ určiť množinu elementov, ktoré majú byť zo zobrazenia diagramu vynechané. V podstate môžu byť hodnoty atribútov určené rôznymi spôsobmi. Jednou možnosťou je výber týchto hodnôt zo zoznamu (podobne ako pri filtrovaní štruktúry programu v analyzovanom prostredí AspectJ Browser). Ďalšou možnosťou je spôsob, ktorý je použitý v samotnom jazyku AspectJ. Používa pri definícii bodových prierezov a umožňuje výber tried pomocou takzvaného typového vzoru (napr. `painter.elements.Line* || *..Point`) a výber metód pomocou vzoru metódy (napr. `!public void painter..LineArt.Move*(int, *)`). Výhodou použitia týchto vzorov je to, že umožňujú pomerne presne určiť množinu filtrovaných elementov a tento spôsob je naviac známy pre programátorov v AspectJ. Napriek tomu môže byť v niektorých prípadoch postačujúci výber zo zoznamu možných hodnôt, ktorý je na druhej strane rýchlejší. Preto sa predpokladá, že najvhodnejšie bude použitie obidvoch možností, prípadne ich kombinácie pri filtrovaní zobrazených diagramov.

Zaujímavou možnosťou určenia filtrovaných elementov, by mohlo byť vymedzenie elementov prostredníctvom značiek. Tieto značky môžu predstavovať dodatočnú informáciu o definovaných triedach a metódach programu, ktorú zadá programátor pri písaní programu a rozširujú množinu atribútov, ktorá určuje daný element programu. Vhodným výberom značiek je možné rozdeliť elementy programu do rôznych skupín, pričom jeden element programu môže patriť do viacerých skupín. Filtrované elementy programu môžu byť potom určené nielen atribútmi danými programovacím jazykom (AspectJ) ale aj špecifikovaním množiny filtrovaných značiek. Potom budú vynechané všetky elementy programu, ktoré sú v programe označené nejakou značkou z tejto množiny.

Možný spôsob definovania značiek uvádza nasledujúci príklad:

```
public class Line
{
	/*! @info !*/
	Point GetOrigin();
		{ ... };
	/*! @info !*/
	bool HitTest(int x, int y)
		{ ... };
	/*! @rotacia, @prekreslenie !*/
	public void Rotate(int angle)
		{ ... };
	/*! @skalovanie, @prekreslenie !*/
	public void Scale(int ratio)
		{ ... };
	/*! @presunutie, @prekreslenie !*/
	public void Move(int relx, int rely)
		{ ... };
};

public class Triangle
{
	/*! @info !*/
	Point GetOrigin();
		{ ... };
	/*! @info !*/
	bool HitTest(int x, int y)
		{ ... };
	/*! @rotacia, @prekreslenie !*/
	public void Rotate(int angle)
		{ ... };
	/*! @skalovanie, @prekreslenie !*/
	public void Scale(int ratio)
		{ ... };
	/*! @presunutie, @prekreslenie !*/
	public void Move(int relx, int rely)
		{ ... };
};
```

Ako je vidieť z tohto príkladu, značky sú uvedené v špeciálnom komentáre, ktorý je ignorovaný prekladačom jazyka AspectJ. Tento komentár spracuje prostredie jazyka AspectJ, ktoré zobrazuje navrhnuté sekvenčné diagramy a umožní ich filtrovanie prostredníctvom týchto značiek. V tomto prípade značky bližšie špecifikujú operáciu, ktorá je implementovaná uvedenými metódami tried a umožnia napríklad zobraziť na diagrame len tie metódy, ktoré spôsobia prekreslenie objektov.

4.4. Nevýhody navrhnutej vizualizácie

Navrhnutý spôsob vizualizácie má oproti vizualizácii implementovanej v analyzovaných prostrediach niektoré nevýhody. Prvou z nich je tá, že nezobrazuje všetky nové prvky jazyka, ktoré prináša jazyk AspectJ. Nezobrazovanými prvkami sú zavedenia štrukturálnych vlastností (atribútov, dedenia, rozhraní) a deklarované bodové prierezy. Vyplýva to z toho, že navrhnutá vizualizácia sa opiera o sekvenčné diagramy, ktoré zobrazujú predovšetkým správanie inštancií reprezentované operáciami (metódami, zavedeniami metód, radami), pričom uvedené zavedenia a deklarácie bodových prierezov súvisia skôr so statickými vlastnosťami programu (štruktúrou tried a aspektov). Vhodnou metódou na zobrazenie dodefinovaných štrukturálnych vlastností alebo deklarovaných bodových prierezov je práve zobrazenie štruktúry programu implementované v analyzovaných prostrediach, prípadne by sa dala znovu použiť notácia UML, pričom by sa tieto prvky mohli zobraziť prostredníctvom diagramu tried.

Ďalšou nevýhodou je to, že navrhnutá vizualizácia nie je vhodná na spätné vytváranie aspektov z vizuálnej reprezentácie. Pri vizualizácii prostredníctvom sekvenčných diagramov by prichádzalo do úvahy len vytváranie bodového prierezu rád špecifikovaním metód, ktorých operácie majú byť danou radou ovplyvnené. To však vyžaduje zobrazenie diagramu, na ktorom sú všetky volania metód, ktoré majú byť danou radou ovplyvnené, čo vyžaduje vyhľadanie zodpovedajúceho toku riadenia, pričom takýto diagram môže byť príliš zložitý (aj po filtrovaní) na to, aby malo spätné vytváranie aspektov nejaký zmysel. Možnosťou by bolo výber všetkých požadovaných metód bodového prierezu z viacerých sekvenčných diagramov, čo tiež neprináša žiadaný efekt. Navrhnutá metóda vizualizácie je skôr vhodná na overenie správnej implementácie existujúcich, prípadne vytváraných aspektov programu, ako ich spätné generovanie prostredníctvom vizualizácie.

5. Prototyp prostredia na vizualizáciu aspektov v AspectJ

V tejto časti dokumentácie sa zameriam na opis postupu, ktorý viedol k vytvoreniu aplikácie na overenie navrhnutej metódy vizualizácie aspektov. Ako pri ostatných projektoch, ktoré vedú k vytvoreniu softvérového produktu, bolo potrebné prejsť etapou špecifikácie, návrhu, implementácie a testovania.

5.1. Špecifikácia

Špecifikácia vytváraného prototypu prostredia je rozdelená do niekoľkých častí. Tieto časti zahŕňajú špecifikáciu používateľov, požiadaviek a údajov vytváraného systému.

Používatelia

Predpokladá sa, že aplikácia bude mať len jedného používateľa, ktorým bude programátor, vytvárajúci program v jazyku AspectJ. Nepredpokladá sa súčasný prístup viacerých používateľov k jednému vytváranému programu, ani súčasná práca viacerých používateľov s jednou inštanciou spustenej aplikácie.

Požiadavky

V prvom rade treba vymedziť požiadavky, ktoré budú na vytvorené vývojové prostredie kladené. Tieto požiadavky sa dajú rozdeliť do niekoľkých skupín. Prvú skupinu tvoria požiadavky, ktoré sú spojené s potrebou overenia navrhnutej vizualizácie aspektov. Druhú skupinu tvoria požiadavky, ktoré podmieňujú pohodlnú prácu s vytvoreným prostredím. Poslednú skupinu tvoria nefunkcionálne požiadavky, ktoré sú kladnené na výsledný produkt a sú dôležité pre jeho ďalšiu údržbu (modifikáciu a rozširovanie).

1. Požiadavky z hľadiska navrhnutej metódy vizualizácie
    - zobrazenie diagramov na základe statickej analýzy vstupného programu
      Prostredie by malo analyzovať vstupné súbory používateľom vytváraného programu a na základe tejto analýzy a používateľom určenej počiatočnej metódy, zobraziť tok riadenia vybranej metódy prostredníctvom sekvenčného diagramu spoločne so zmenami spôsobenými aspektami.

    - možnosť navigácie v zdrojových súboroch vstupného programu
      Po zobrazení sekvenčného diagramu nejakej metódy vstupného programu, by malo prostredie umožniť používateľovi zobraziť umiestnenie zobrazeného prvku diagramu v zdrojových súboroch vstupného programu.

    - možnosť zmeny zobrazenia diagramu používateľom systému
      Prostredie by malo umožniť zmeny zobrazeného toku riadenia, ktorý nemusí zodpovedať používateľom očakávanému zobrazeniu. Používateľ určí tie skutočnosti, ktoré sa nedajú rozhodnúť statickou analýzou (napr. konkrétnu triedu, do ktorej patrí cieľový objekt zobrazeného volania = dynamické viazanie). Naviac by malo prostredie umožniť zjednodušiť zobrazenie diagramu používateľom zadaným filtrom.

2. Požiadavky na prácu s prostredím
    - jednoduchá a intuitívna práca s prostredím
      Prostredie poskytne používateľovi grafické rozhranie, podobné s ostatnými vývojovými prostrediami. Takto sa zmenší čas potrebný na adaptáciu používateľa na nové vývojové prostredie.

    - editácia vstupného programu
      Pod touto požiadavkou sa myslí možnosť zmeny množiny zdrojových súborov, z ktorých pozostáva vstupný program, pridaním nového alebo existujúceho zdrojového súboru alebo vylúčením zdrojového súboru zo vstupného programu. Tiež sa predpokladá možnosť editovania textu jednotlivých zdrojových súborov vstupného programu priamo v prostredí s prípadným zvýrazňovaním syntaxe jazyka AspectJ.

    - kompilácia a spustenie vstupného programu
      Táto požiadavka patrí medzi základné požiadavky, ktoré sa kladú na akéhokoľvek vývojového prostredia pre ľubovolný programovací jazyk. Na splnenie tejto požiadavky bude potrebné použiť externý kompilátor jazyka AspectJ a externú implementáciu JVM (Java Virual Machine).

3. Nefunkcionálne požiadavky
    - rozšíriteľnosť
      Výsledná aplikácia by mala byť vytvorená modulárne, aby sa dala ľahko rozšíriť buď zmenou nejakého modulu alebo jeho úplným nahradením, pričom sa predpokladá minimálny dopad tejto zmeny na ostatné moduly alebo časti systému. Požaduje sa najmä ľahká modifikovateľnosť (zámena) časti aplikácie, ktorá vykonáva statickú analýzu vstupného programu.

    - prenositeľnosť
      Táto požiadavka predpokladá, že čo najmenší počet modulov systému bude závislých od konkrétnej platformy alebo od vývojového prostredia, v ktorom bola aplikácia vytvorená.

    - robustnosť
      Robustnosť predpokladá, že aplikácia bude odolná voči nesprávnym vstupom a na prípadné chyby upozorní používateľa.

Údaje

Všetky vstupno-výstupné údaje viditeľné pre používateľa, s ktorými bude prostredie narábať sa dajú označiť pod spoločným pojmom projekt. Projekt združuje všetky údaje, ktoré o používateľom vytváranom programe bude prostredie uchovávať v súboroch na disku. Patrí sem okrem zdrojových súborov vstupného programu (prípona `java`) aj súbor s nastaveniami projektu (prípona `ajp`). V tomto súbore bude uložená informácia o zdrojových súboroch vstupného programu (ich umiestnenie na disku, názov), informácia o zobrazenom diagrame (zobrazený tok riadenia, aplikovaný filter) a ostatné informácie slúžiace na obnovenie stavu, v ktorom sa nachádzala aplikácia pri poslednom uložení projektu.

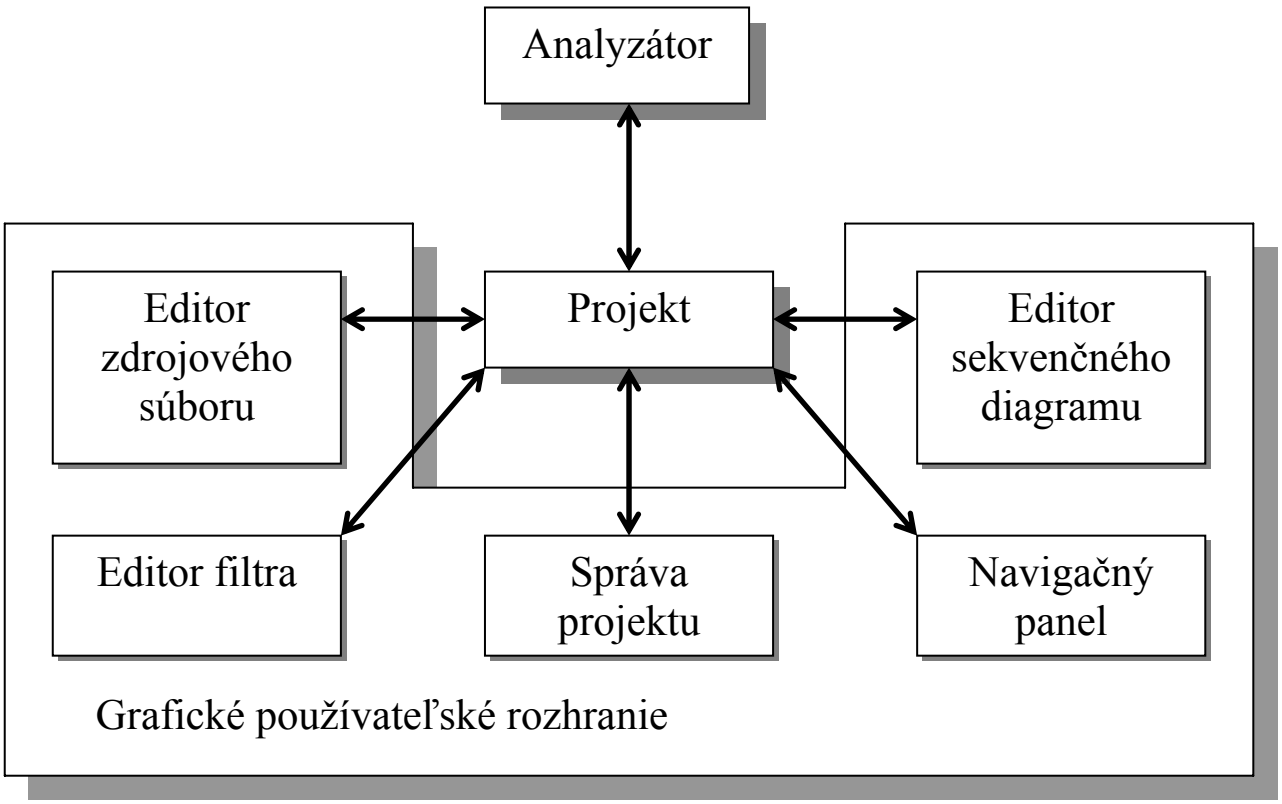
Poznámka: Špecifikácia súboru s nastaveniami projektu je uvedená v časti A.1 technickej dokumentácie.

5.2. Návrh

Táto časť vychádza zo špecifikácie požiadaviek na vytváraný systém. Opisuje niektoré aspekty návrhu systému. Najprv bude uvedená celková architektúra systému a následne bude uvedený návrh najdôležitejších častí systému. Pri návrhu bol použitý objektový prístup. Okrem toho bol pri návrhu použitý návrhový vzor Visitor, pričom boli vykonané niektoré jeho modifikácie, ktoré budú vysvetlené pri opise postupu návrhu časti systému, ktorá predstavuje výstup lexikálneho analyzátora.

Architektúra systému

Celková architektúra systému je uvedená na obrázku 5.1. Uvádza rozdelenie systému do jednotlivých modulov a prepojenie týchto modulov. Centrálnym modulom celej architektúry je modul `Projekt`. S ním sú prepojené všetky ostatné moduly systému, ktorými sú modul `Analyzátor`, `Editor zdrojového súboru`, `Editor sekvenčného diagramu`, `Editor filtra`, `Navigačný panel` a `Správa projektu`. Naznačené prepojenia predstavujú interakciu medzi prepojenými modulmi (napr. volania metód). Všetky moduly systému okrem modulu `Analyzátor` a `Projekt` implementujú časť používateľského rozhrania, pričom všetky tieto časti spoločne predstavujú používateľské rozhranie celej aplikácie.



Obrázok 5.1: Architektúra systému

Ďalej bude uvedená stručná charakteristika všetkých modulov systému. Podrobnejší opis niektorých dôležitejších modulov bude uvedený v ostatných častiach návrhu.

Modul Projekt
Združuje všetky dôležité údaje, s ktorými systém pracuje. Patria k nim údaje o zdrojových súboroch vstupného programu, vnútornej reprezentácii analyzovaného programu, vytvorených tokoch riadenia (zobrazených na sekvenčnom diagrame), nastaveniach filtra a pod. Ostatným modulom systému poskytuje rozhranie na prístup k týmto údajom, ich zápis alebo načítanie z disku a umožňuje tiež spustiť analýzu vstupného programu.

Modul Analyzátor
Úlohou tohto modulu je statická analýza zdrojových súborov projektu. Na základe tejto analýzy modul vytvára vnútornú reprezentáciu programu, ktorá ďalej slúži pri navigácii a pri vytváraní diagramov.

Modul Editor zdrojového súboru
Modul poskytuje používateľovi rozhranie na editovanie vybraného zdrojového súboru projektu. Umožní vkladanie textu, nahradenie textu, označenie bloku textu, kopírovanie, presúvanie alebo vymazanie označeného bloku a pod. Okrem toho môže zobraziť a zvýrazniť určitú časť zdrojového súboru, čo sa využíva pri navigácii prostredníctvom diagramu.

Modul Editor sekvenčného diagramu
Úlohou modulu je zobrazenie vybraného toku riadenia na sekvenčnom diagrame. Pri zobrazovaní berie do úvahy nastavenie filtra. Okrem zobrazenia umožní používateľovi zmenu zobrazeného toku riadenia špecifikovaním konkrétnej cieľovej triedy volania a tiež umožní navigáciu v programe prostredníctvom príslušného diagramu.

Modul Editor filtra
Poskytuje používateľovi rozhranie na špecifikáciu prvkov programu, ktoré nemajú byť na vytvorenom sekvenčnom diagrame zobrazené.

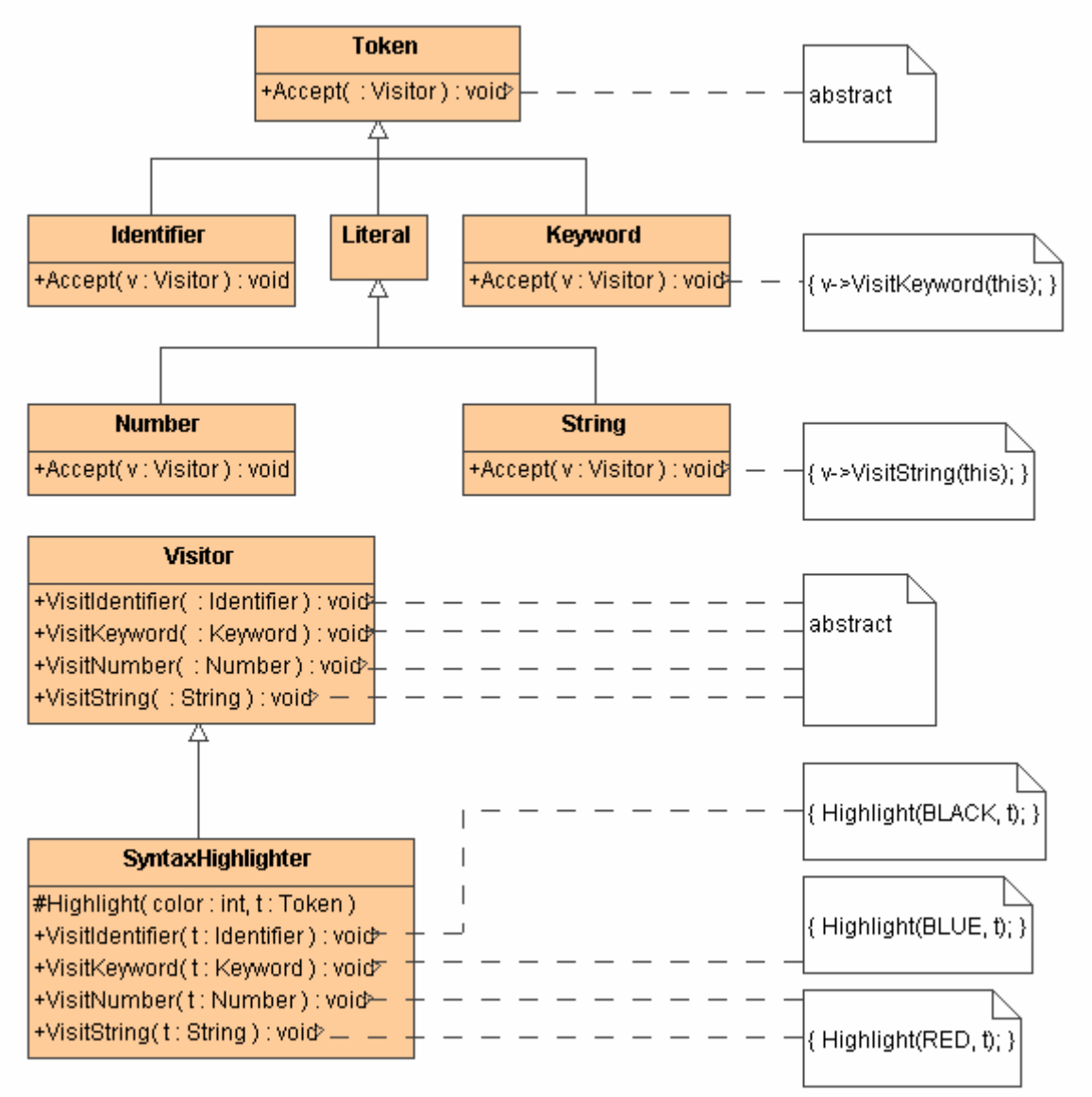
Modul Navigačný panel
Pretože editor zdrojového súboru zobrazuje naraz len jeden zdrojový súbor, tento modul umožní výber editovaného zdrojového súboru. Podobne umožní aj výber toku riadenia (daný počiatočnou metódou), ktorý má byť zobrazený editorom sekvenčného diagramu.

Modul Správa projektu
Modul Správa projektu umožní používateľovi vytvoriť nový projekt, uložiť existujúci projekt na disk alebo nahrať uložený projekt z disku. Okrem toho umožní pridávanie nových alebo existujúcich zdrojových súborov do projektu, iniciuje statickú analýzu zdrojových súborov projektu, kompiláciu a spustenie projektu.

Návrh s použitím vzoru Visitor

V ďalších častiach návrhu bude viackrát použitý návrhový vzor Visitor [6]. Tento návrhový vzor umožňuje oddeliť zložité objektové štruktúry od vzájomne nesúvisiacich operácií, ktoré s týmito štruktúrami narábajú. Pri jeho použití boli vykonané jednoduché modifikácie tohto návrhového vzoru, odstraňujúce niektoré jeho nedostatky, pričom sú dostatočne všeobecné na to, aby malo použitie modifikovaného vzoru zmysel aj v iných projektoch. Uvedený návrhový vzor a jeho modifikácie budú prezentované na príklade riešiacom konkrétny problém, ktorý sa vyskytol v návrhu.

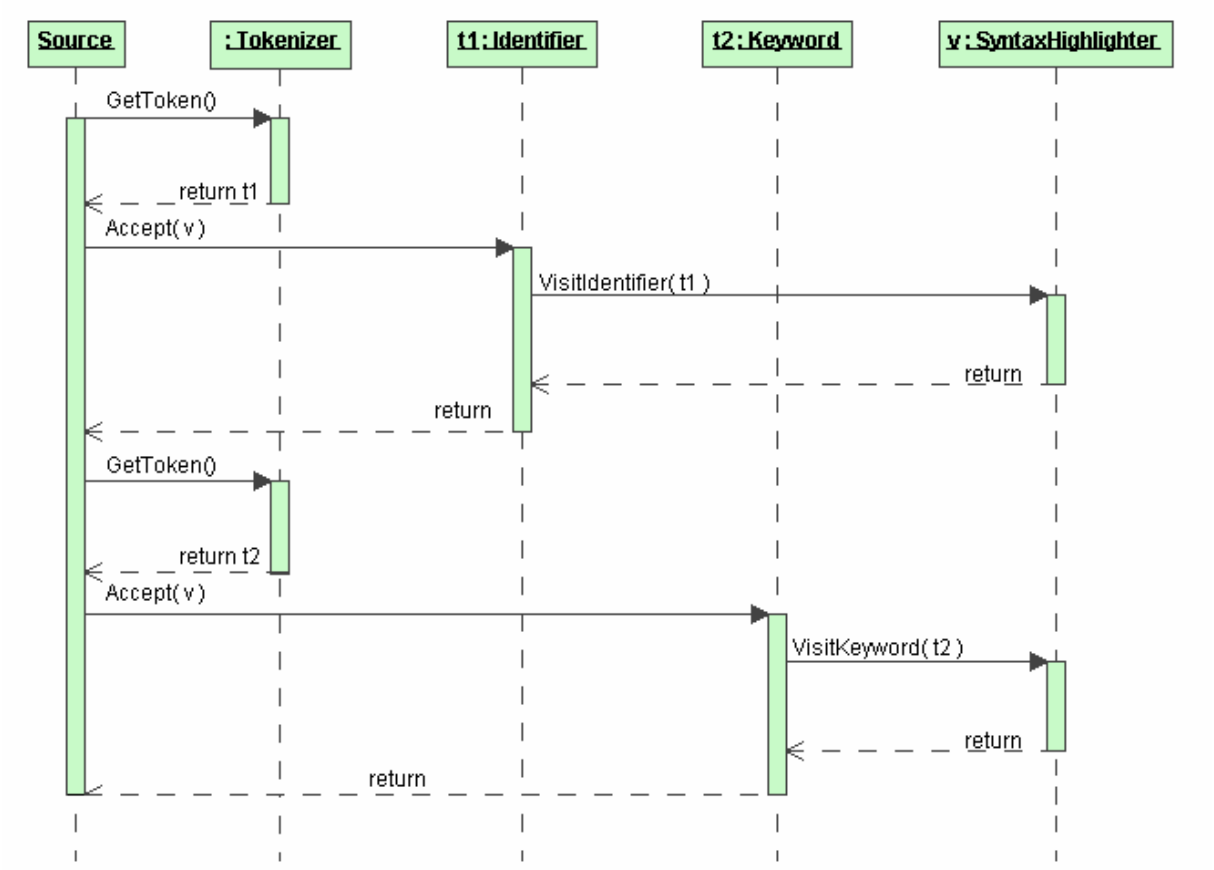
Úlohou je navrhnúť výstup lexikálneho analyzátora programu, ktorý by sa dal použiť na zvýrazňovanie syntaxe a na syntaktickú analýzu. Vo všeobecnosti sú výstupom lexikálneho analyzátora lexikálne jednotky. Pretože lexikálny analyzátor identifikuje lexikálne jednotky (identifikátory, kľúčové slová ...), nad ktorými majú byť definované rôzne, vzájomne nesúvisiace operácie, je vhodné použiť pre návrh lexikálnych jednotiek návrhový vzor Visitor.



Obrázok 5.2: Návrh lexikálnych jednotiek

Prvotný návrh lexikálnych jednotiek je na obrázku 5.2. Hierarchia uvedená na obrázku nie je úplná ale postačuje na naznačenie použitia návrhového vzoru. Od základnej triedy lexikálnej jednotky (trieda `Token`) sú odvodené triedy identifikátorov (trieda `Identifier`), kľúčových slov (trieda `Keyword`) a literálov (trieda `Literal`). Od triedy literálov sú ďalej odvodené triedy pre čísla (trieda `Number`) a reťazce (trieda `String`). Podľa návrhového vzoru Visitor, bola vytvorená trieda `Visitor` s abstraktnými operáciami `VisitIdentifier` s parametrom typu `Identifier`, `VisitKeyword` s parametrom typu `Keyword`, `VisitNumber` s parametrom typu `Number` a `VisitString` s parametrom typu `String`. Naviac do tried lexikálnych jednotiek bola pridaná operácia `Accept`, ktorej úlohou je zavolať príslušnú operáciu vstupného objektu typu `Visitor`. Čiže operácia `Accept` pre reťazec zavolá operáciu `VisitString` vstupného objektu typu `Visitor`.

Triedy odvodené od triedy `Visitor` definujú konkrétne operácie, ktoré majú byť vykonané pre objekty tried odvodených od základnej triedy lexikálnej jednotky. Pre názornosť je na obrázku uvedená trieda `SyntaxHighlighter` odvodená od triedy `Visitor`, ktorá by mohla byť použitá pri zvýrazňovaní syntaxe. Operácie tejto triedy rozhodnú o tom, aká farba má byť použitá pri znázornení konkrétnej lexikálnej jednotky. Identifikátoru priraďuje čiernu farbu, kľúčovému slovu modrú a číslu alebo reťazcu červenú. Operácia `Highlight` tejto triedy zabezpečí, že sa časti zdrojového súboru, ktorá zodpovedá vstupnej lexikálnej jednotke, priradí daná farba.

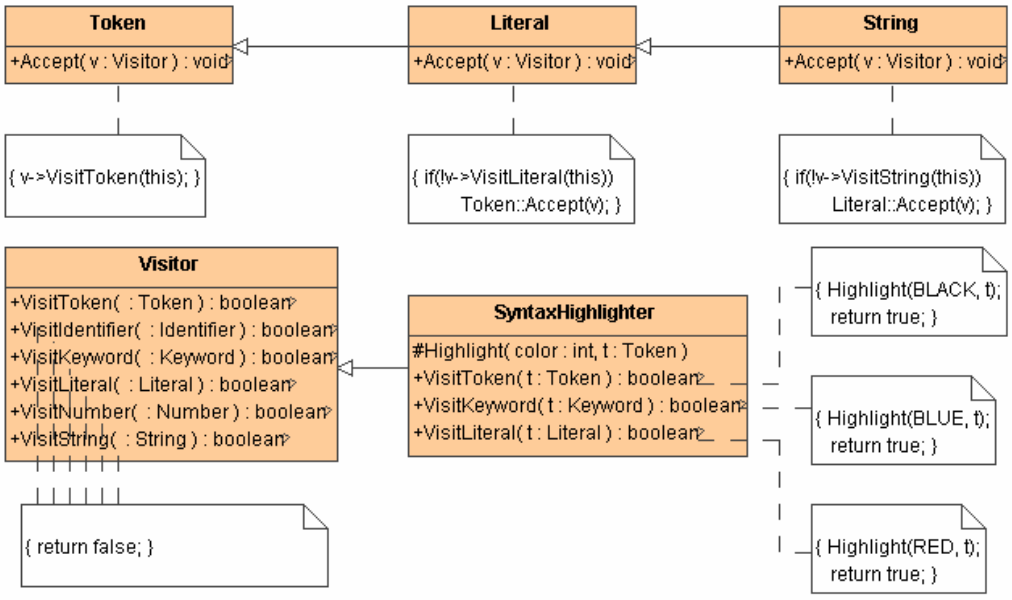


Obrázok 5.3: Príklad použitia triedy SyntaxHighlighter

Použitie triedy `SyntaxHighlighter` je znázornené na obrázku 5.3. Je na ňom objekt `Source`, ktorý najskôr získa od lexikálneho analyzátora (objekt triedy `Tokenizer`) ďalšiu lexikálnu jednotku (operácia `GetToken`). Keďže získaná lexikálna jednotka je identifikátor, volanie operácie `Accept` spôsobí vykonanie operácie `Accept` definovanej v triede `Identifier`. Táto operácia následne volá operáciu `VisitIdentifier` vstupného objektu, ktorým je v tomto prípade objekt v triedy `SyntaxHighlighter`. Tento postup je znázornený aj pre ďalšiu lexikálnu jednotku, ktorou je kľúčové slovo.

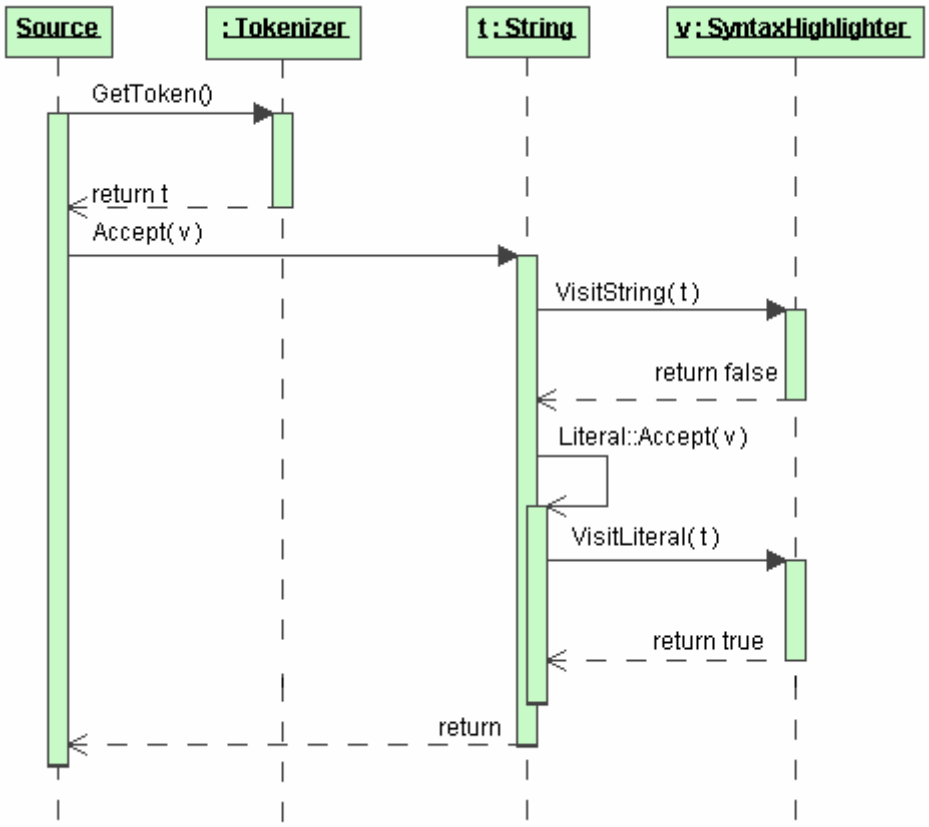
Toto riešenie má niektoré nevýhody. Jednou z nich je to, že ak pridáme do existujúcej hierarchie lexikálnych jednotiek novú lexikálnu jednotku (napríklad triedu pre znak odvodenú od literálu) a chceme aby pre ňu v triede `Visitor` existovala operácia, musíme ju pridať nielen do základnej triedy `Visitor` ale aj do všetkých tried od triedy `Visitor` odvodených. To je spôsobené tým, že návrhový vzor Visitor deklaruje všetky operácie v triede `Visitor` ako abstraktné, čiže objekty odvodených tried nemôžu byť vytvorené bez definovania všetkých operácií základnej triedy. Tomu sa dá predísť tým, že všetky deklarované operácie triedy `Visitor` definujeme už v tejto triede. Čiže pre všetky operácie vytvoríme prázdne metódy.

Ďalšou nevýhodou je to, že neberie plne do úvahy vytvorenú hierarchiu tried. Predpokladajme, že chceme zobraziť všetky literály rovnakou farbou. Pri danom návrhu nám neostáva nič iné, ako pre každú lexikálnu jednotku triedy `X` odvodenej od triedy `Literal` definovať operáciu `VisitX` v triede `SyntaxHighlighter`. Môžeme síce definovať operáciu `Accept` v triede `Literal` a operáciu `VisitLiteral` v triede `SyntaxHighlighter`, zabezpečujúcu priradenie danej farby, ale problém je, že to bude fungovať len pre objekty triedy `Literal`, pretože objekty tried odvodených od triedy `Literal` nevolajú operáciu `VisitLiteral` ale operácie prislúchajúce k svojej triede. Riešením je, ak zabezpečíme, aby pri nedefinovaní operácie `VisitX` v triede `SyntaxHighlighter` (resp. v triede odvodenej od triedy `Visitor`) bola zavolaná namiesto `VisitX` operácia `VisitY` prislúchajúca najbližšej rodičovskej triede `Y` triedy `X`, pre ktorú je operácia `VisitY` definovaná. Toto riešenie je znázornené na obrázku 5.4.



Obrázok 5.4: Upravený návrhový vzor Visitor

Prvou zmenou bolo deklarovanie operácií `VisitX` v triede `Visitor` s návratovou hodnotou typu `boolean`, ktoré sú definované tak, že vracajú hodnotu `false`. Operácie `VisitX` definované v triede odvodenej od triedy `Visitor` sú definované s návratovou hodnotou `true`. Táto hodnota sa využíva v operácii `Accept`, ktorá v prípade, že `VisitX` vráti hodnotu `false` volá naviac operáciu `Accept` svojej nadtriedy. Tým je zabezpečená požadovaná funkcionalita prezentovaná sekvenčným diagramom, ktorý je uvedený na obrázku 5.5.



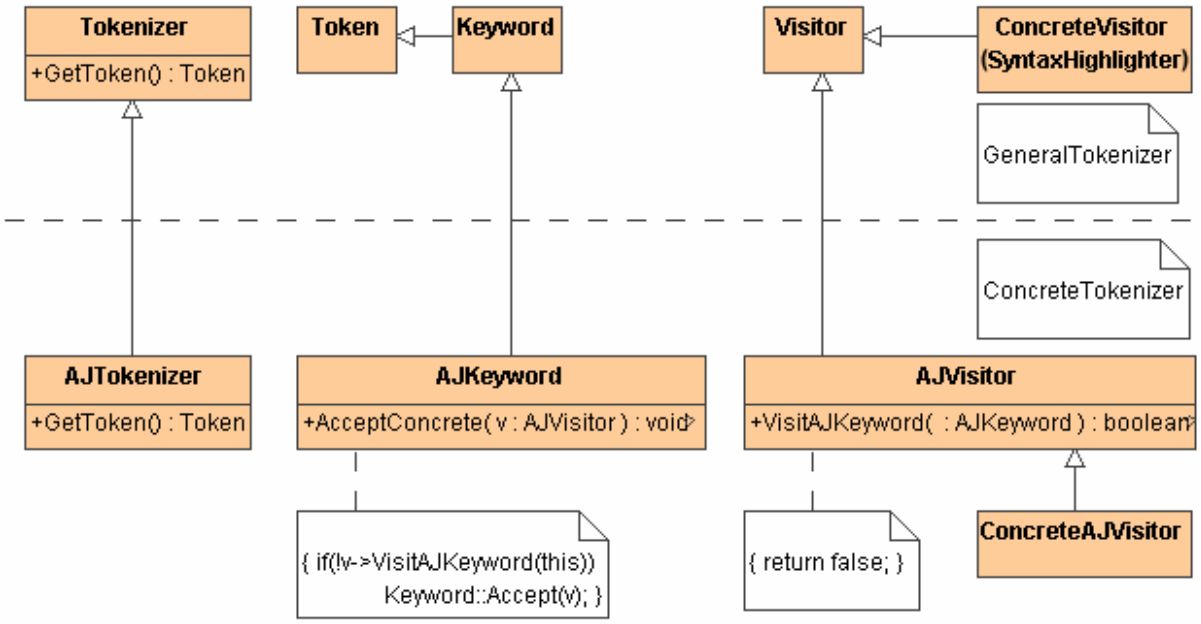
Obrázok 5.5: Sekvenčný diagram pre upravený návrhový vzor Visitor

Na obrázku je znovu znázornený objekt `Source`, ktorý získa lexikálnu jednotku od lexikálneho analyzátora (trieda `Tokenizer`). V tomto prípade je lexikálnou jednotkou reťazec a preto volanie operácie `Accept` spôsobí následné volanie operácie `VisitString` objektu triedy `SyntaxHighliter`. Pretože táto operácia nebola v triede `SyntaxHighliter` definovaná, vykoná sa operácia `VisitString` triedy `Visitor`. Tá je definovaná tak, že vracia hodnotu `false`. Preto sa po volaní operácie `VisitString` naviac volá operácia `Accept` definovaná v triede `Literal`. Tá spôsobí volanie `VisitLiteral` objektu triedy `SyntaxHighliter`, ktorá po vykonaní vráti hodnotu `true` (bola definovaná v triede `SyntaxHighlighter`), čo značí, že netreba ďalej volať ďalšie operácie `Accept` (operácie nadtried) a vykonávanie operácie `Accept` sa ukončí.

Uvedená úprava návrhového vzoru má aj ďalší dôsledok. Ak sa rozhodneme rozšíriť hierarchiu lexikálnych jednotiek pomocou dedenia (napríklad pridáme triedu pre reálne - trieda `Real`, a celé čísla - trieda `Integer`) a zmeníme len triedu `Visitor` (pridáme operácie `VisitReal`, `VisitInteger`), potom budú objekty triedy `SyntaxHighlighter` fungovať

naďalej tak, ako očakávame (budú priraďovať celým a reálnym číslam rovnakú farbu ako predtým lexikálnej jednotke číslo).

Doteraz uvedená hierarchia lexikálnych jednotiek by mohla byť vytváraná lexikálnym analyzátorom ľubovolného jazyka. Napriek tomu, že je vhodná pre zvýrazňovanie syntaxe, nebude postačujúca napríklad pre syntaktický analyzátor. Preto sa predpokladá, že pre lexikálny analyzátor konkrétneho jazyka bude ďalej rozšírená (napríklad konkrétnymi kľúčovými slovami daného jazyka). Pritom však nechceme, aby pri takomto rozšírení bolo nutné meniť triedu `Visitor`, pretože by sme touto zmenou obmedzili návrh súvisiaci s triedou `SyntaxHighlighter` len na jeden lexikálny analyzátor konkrétneho jazyka. Skúsme teda rozšíriť hierarchiu lexikálnych jednotiek a triedy `Visitor`, len s použitím dedenia. Takéto rozšírenie je na obrázku 5.6.



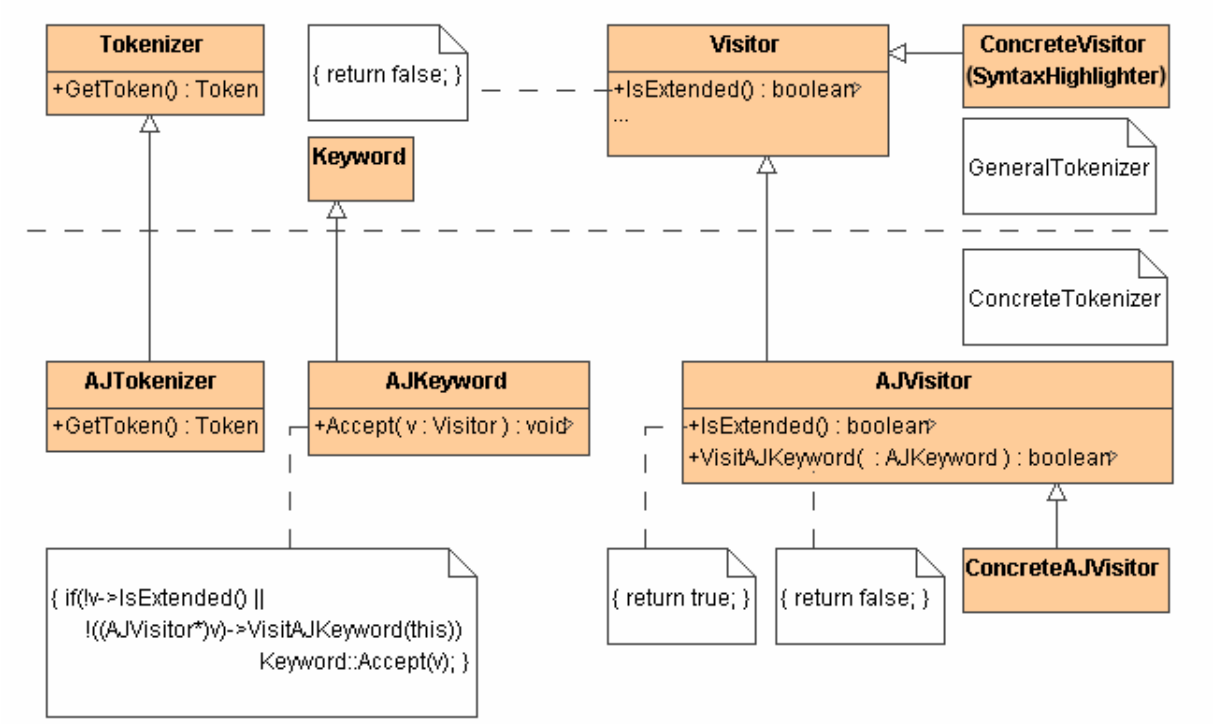
Obrázok 5.6: Prvotné rozšírenie všeobecného lexikálneho analyzátora

Uvedený obrázok je rozdelený na dve časti. Vrchná časť zodpovedá pôvodnej hierarchii tried navrhnutej pre všeobecný lexikálny analyzátor. Spodná časť zachytáva príklad rozšírenia pre jazyk AspectJ. V spodnej časti sa nachádza trieda `AJTokenizer`, ktorá reprezentuje lexikálny analyzátor jazyka AspectJ. Táto trieda je odvodená od triedy `Tokenizer` reprezentujúcej všeobecný lexikálny analyzátor a podobne ako trieda `Tokenizer` aj trieda `AJTokenizer` definuje operáciu `GetToken`, ktorá vracia lexikálnu jednotku typu odvodeného od triedy `Token`.

Naviac sa v spodnej časti nachádzajú triedy `AJKeyword`, `AJVisitor` a `ConcreteAJVisitor`. Trieda `AJKeyword` predstavuje príklad rozšírenej lexikálnej jednotky pre daný jazyk. V tomto prípade je to kľúčové slovo jazyka AspectJ. Trieda `AJVisitor` a trieda `ConcreteAJVisitor` sú v podobnom vzťahu ako trieda `Visitor` a `ConcreteVisitor` (napr. `SyntaxHighliter`) všeobecného lexikálneho analyzátora. To znamená, že trieda `ConcreteAJVisitor` definuje konkrétne operácie, ktoré majú byť vykonané pre rôzne typy lexikálnych jednotiek. V triede `AJVisitor` sú deklarované operácie `VisitX` pre všetky dodefinované triedy `X` reprezentujúce lexikálne jednotky

AspectJ. Tieto operácie sú definované s návratovou hodnotou `false` (podobne ako operácie triedy `Visitor`). V triede `AJKeyword` je deklarovaná operácia `AcceptConcrete`. Táto operácia je definovaná tak, že volá operáciu `VisitAJKeyword` vstupného objektu typu odvodeného od `AJVisitor`. V prípade, že táto operácia vráti hodnotu `false`, volá sa naviac operácia `Accept` triedy `Keyword`.

Problém tohto riešenia je, že operácia `AcceptConcrete` nemôže byť pre lexikálnu jednotku získanú z lexikálneho analyzátora volaná, pretože nie je deklarovaná v triede `Token`. Riešením by mohlo byť premenovať túto operáciu na `Accept` so vstupným parametrom typu `Visitor` a v tejto operácii pretypovať vstupný parameter na typ `AJVisitor` (upcasting). V takomto prípade ale dôjde k zrúteniu programu ak sa operácia `Accept` zavolá s objektom ktorého trieda nie je odvodená od triedy `AJVisitor` (napríklad s objektom triedy `SyntaxHighlighter`). Ak chceme, aby operácia `Accept` fungovala pre ľubovolný objekt triedy odvodenej od triedy `Visitor` (priamo alebo nepriamo), musíme vedieť v operácii `Accept` rozhodnúť, či je vstupný objekt objektom triedy odvodenej od `AJVisitor` alebo nie je. Na tento účel môžeme použiť RTTI (Run Time Type Information), čo je mechanizmus, ktorý umožňuje zistiť typ ľubovolného objektu počas vykonávania programu. Nevýhodou je, že nie všetky jazyky alebo kompilátory daného jazyka podporujú tento mechanizmus a v tomto prípade sa dá ľahko nahradiť tým, že si sami vytvoríme požadovanú informáciu o type objektu. Toto riešenie je na obrázku 5.7.

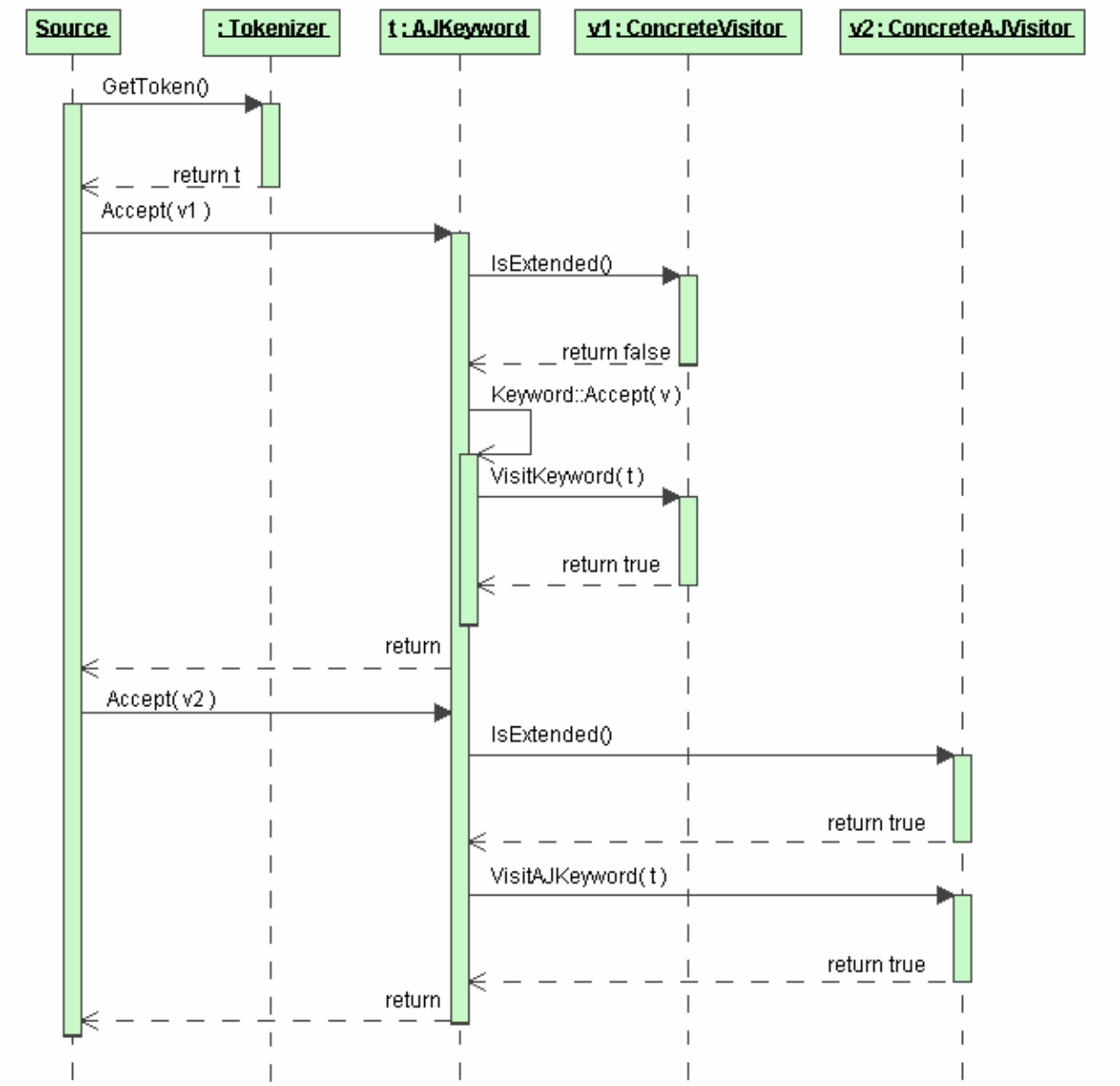


Obrázok 5.7: Výsledné rozšírenie všeobecného lexikálneho analyzátora

Riešenie spočíva v deklarácii operácie `IsExtended` s návratovou hodnotou typu `boolean` v triede `Visitor`. Táto je v triede `Visitor` definovaná tak, aby vracala návratovú hodnotu `false`. Operácia `IsExtended` slúži na rozlíšenie objektu typu odvodeného len od triedy `Visitor` od objektu typu odvodeného od triedy `AJVisitor` (vo všeobecnosti od triedy `Visitor` pre konkrétny lexikálny analyzátor). Preto je v triede `AJVisitor` definovaná táto operácia s návratovou hodnotou `true`. Operácia `IsExtended` je využívaná v operácii

`Accept` triedy `AJKeyword`. Táto operácia pretypuje vstupný objekt (na typ `AJVisitor`) a volá jeho operáciu `VisitAJKeyword`, len v prípade, ak operácia `IsExtended` vráti hodnotu `true` (vstupný objekt je typu odvodeného od `AJVisitor`). V prípade, že operácia `IsExtended` vráti hodnotu `false`, volá sa namiesto operácie `VisitAJKeyword` operácia `Accept` definovaná v rodičovskej triede (trieda `Keyword`) triedy `AJKeyword`.

Na obrázku 5.8 je uvedený sekvenčný diagram prezentujúci fungovanie rozšírenej hierarchie pre konkrétny lexikálny analyzátor.



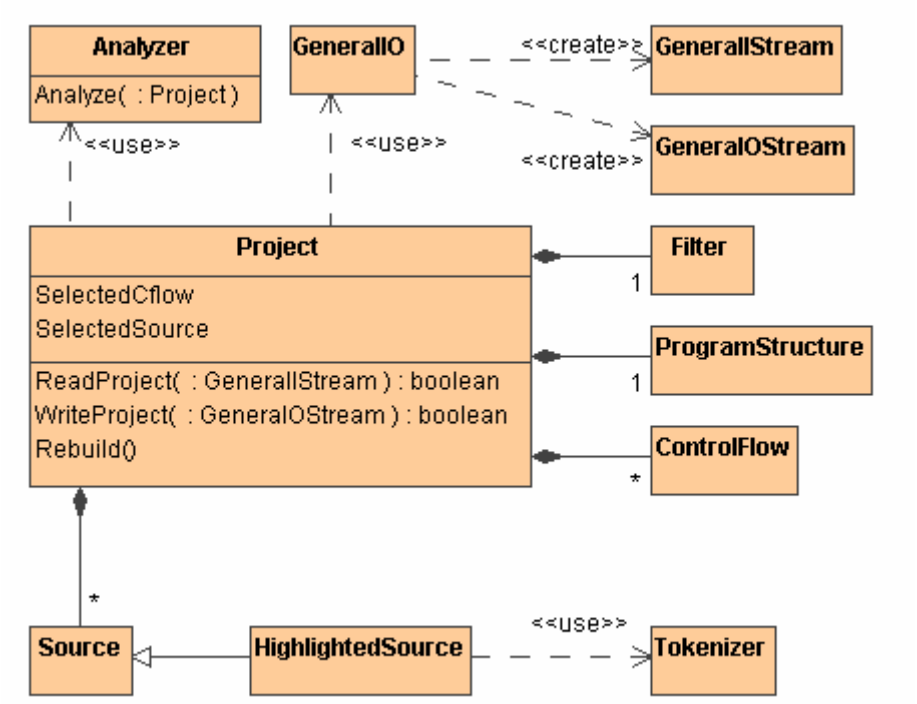
Obrázok 5.8: Sekvenčný diagram rozšírenia všeobecného lexikálneho analyzátora

Objekt `Source` zobrazený na diagrame najskôr získa lexikálnu jednotku od lexikálneho analyzátora (`Tokenizer`). Touto lexikálnou jednotkou je kľúčové slovo jazyka AspectJ (trieda `AJKeyword`). Ďalej sa volá dva razy operácia `Accept` tejto lexikálnej jednotky, pričom prvýkrát je parametrom objekt `v1` triedy `ConcreteVisitor` a druhýkrát objekt `v2` triedy `ConcreteAJVisitor`. Operácia `Accept` s objektom `v1` zavolá najskôr operáciu `IsExtended` tohto objektu. Pretože objekt `v1` nie je odvodený od triedy `AJVisitor`, toto

volanie skončí návratovou hodnotou `false`. To spôsobí v operácii `Accept` triedy `AJKeyword` volanie operácie `Accept` rodičovskej triedy (trieda `Keyword`) a táto volá operáciu `VisitKeyword` objektu `v1`, ktorá vracia hodnotu `true` a tým sa ukončí operácia `Accept`. Ak je parametrom operácie `Accept` objektu `t` objekt `v2`, tak sa volanie `IsExtednded` objektu `v2` skončí v tejto operácii návratovou hodnotou `true` (objekt `v2` je objektom triedy odvodenej od `ConcreteAJVisitor`). To má za následok volanie operácie `VisitAJKeyword` objektu `v2`. Keďže táto operácia vráti hodnotu `true`, operácia `Accept` objektu `t` sa skončí.

Modul Projekt

Úlohou modulu Projekt je združovať všetky dôležité údaje, s ktorými pracuje navrhovaný systém a poskytuje tiež rozhranie pre prístup a prácu s týmto údajmi ostatným častiam systému. Hrubý návrh tried tohto modulu je uvedený na obrázku 5.9.



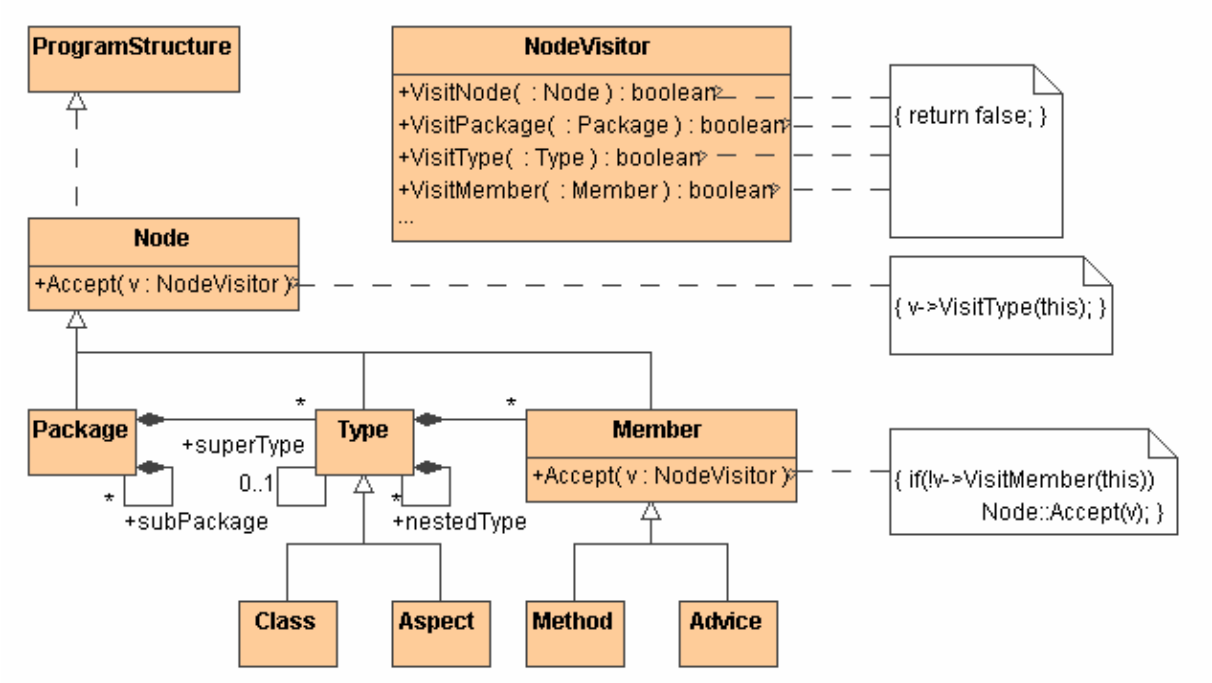
Obrázok 5.9: Diagram tried modulu Projekt

Najdôležitejšou triedou modulu je trieda `Project`. Táto trieda reprezentuje projekt, s ktorým pracuje používateľ systému. Projekt uchováva informácie o zdrojových súboroch vstupného programu (trieda `Source`), nastaveniach filtra (trieda `Filter`), vnútornej reprezentácii programu (trieda `ProgramStructure`). Tiež uchováva údaje o jednotlivých tokoch riadenia (trieda `ControlFlow`), ktoré zobrazuje editor sekvenčného diagramu. Okrem toho trieda Projekt deklaruje atribúty `SelectedCflow` a `SelectedSource`. Atribút `SelectedCflow` uchováva informáciu o editovanom toku riadenia a atribút `SelectedSource` informáciu o editovanom zdrojovom súbore.

Okrem atribútov sú na diagrame zobrazené aj niektoré dôležité operácie triedy `Project`. Sú to operácie `Rebuild`, `ReadProject` a `WriteProject`. Operácia `Rebuild` zbezpečuje spustenie statickej analýzy zdrojových súborov vstupného programu a na základe tejto analýzy je vytvorená vnútorná reprezentácia programu. Táto analýza nie je vykonaná modulom Projekt, ale zabezpečuje ju operácia `Analyze` definovaná v module Analyzátor, deklarovaná triedou `Analyzer` (trieda `Analyzer` predstavuje rozhranie modulu Analyzátor). Operácia `WriteProject` a `ReadProject` slúžia na uchovávanie stavu projektu na disk a obnovu stavu projektu z disku. Na prístup k disku je využívané rozhranie tvorené triedami `GeneralIO`, `GeneralIStream` a `GeneralOStream`. Trieda `GeneralIO` umožňuje otváranie súborov s daným názvom na čítanie alebo zápis. Na prístup k obsahu súboru slúžia operácie deklarované triedami `GeneralIStream` (čítanie) a `GeneralOStream` (zápis). Na diagrame je naviac zobrazená trieda `HighlightedSource`, odvodená od triedy `Source`, ktorá reprezentuje text zdrojového programu vstupného programu so zvýraznenou syntaxou. Na zvýraznenie syntaxe je používaný lexikálny analyzátor reprezentovaný triedou `Tokenizer`.

Vnútorná reprezentácia programu

Na obrázku 5.10 je uvedený návrh vnútornej reprezentácie programu. Táto je tvorená objektmi, ktoré sú v podobnom vzťahu ako jednotlivé prvky jazyka AspectJ. Tieto objekty tvoria hierarchiu, na ktorej vrchole je balík (trieda `Package`). Ten môže obsahovať ďalšie balíky (`subPackage`) a tiež definované typy (trieda `Type`). Typ môže obsahovať vnorené typy (`nestedType`), môže byť odvodený od nadtypu (`superType`) a obsahuje vlastnosti (trieda `Member`). Od triedy `Type` je odvodená trieda `Class` reprezentujúca triedu vstupného programu a trieda `Aspect` reprezentujúca aspekt programu. Od triedy `Member` je odvodená trieda pre metódy (trieda `Method`) a trieda pre rady (trieda `Advice`). Táto hierarchia pokračuje až k jednotlivým príkazom metód alebo rád, čo kvôli prehľadnosti nie je na diagrame obrázku 5.10 uvedené.



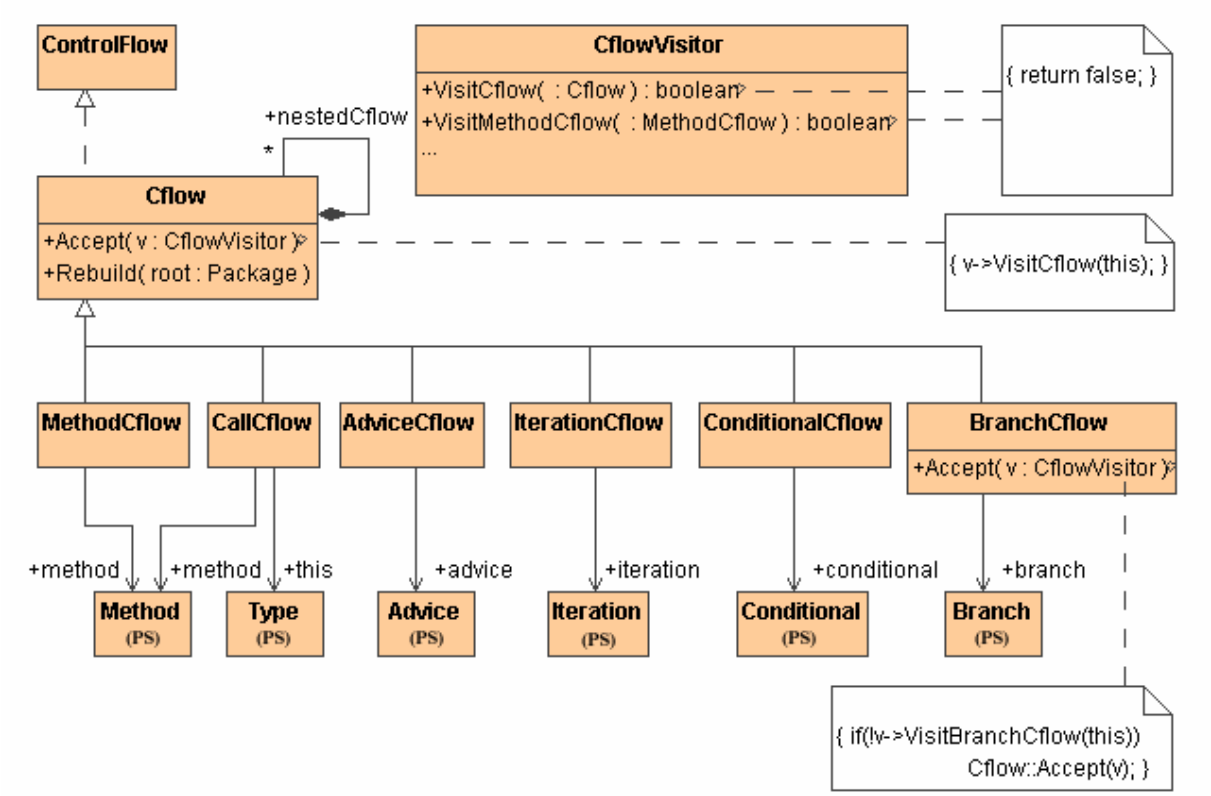
Obrázok 5.10: Podrobnejší návrh vnútornej reprezentácie programu

S vnútornou reprezentáciou programu pracujú viaceré časti systému a tieto nad ňou vykonávajú rôzne operácie. Z tohto dôvodu bol pri návrhu použitý návrhový vzor Visitor. Ten vytvára jednotný spôsob definovania rôznych operácií pracujúcich s vnútornou reprezentáciou. Preto, aby mohol byť návrhový vzor Visitor použitý, bolo potrebné vytvoriť nadtriedu (trieda `Node`) všetkých tried vnútornej reprezentácie a tiež triedu `NodeVisitor`, ktorá je základnou triedou tried definujúcich konkrétne operácie nad vnútornou reprezentáciou. Okrem vytvorenia týchto tried bola definovaná operácia `Accept` vo všetkých triedach (`X`) vnútornej reprezentácie a tiež príslušné operácie `Visit` (`VisitX`) v triede `NodeVisitor`. Obrázok 5.10 uvádza aj príklad definovania deklarovaných operácii (`Node::Accept`, `Member::Accept`, `NodeVisitor::VisitNode`, `NodeVisitor::VisitPackage` ...). Tieto definície naznačujú, že bol použitý modifikovaný návrhový vzor Visitor, ktorý zohľadňuje hierarchiu tried (z hľadiska dedenia).

Poznámka: Úplný diagram tried vnútornej reprezentácie programu je uvedený v časti A.2 technickej dokumentácie.

Tok riadenia

Na obrázku 5.11 je uvedený návrh toku riadenia. V tomto návrhu je tok riadenia (trieda `Cflow`) tvorený vnorenými tokmi riadenia (`nestedCflow`). Čiže tok riadenia tvorí stromová hierarchia objektov (nie sú umožnené cykly).



Obrázok 5.11: Podrobnejší návrh toku riadenia

Od triedy `Cflow` sú odvodené triedy predstavujúce toky riadenia konkrétnych elementov programu. Tieto elementy sú reprezentované triedami vnútornej reprezentácie programu (`PS`) a sú priradené k triedam odvodeným od triedy `Cflow`. Čiže tok riadenia metódy (trieda

`MethodCflow`) má priradenú metódu (trieda `Method`), tok riadenia volania (trieda `CallCflow`) má priradenú metódu a typ volaného objektu (trieda `Type`), tok riadenia rady (trieda `AdviceCflow`) má priradenú radu (trieda `Advice`), tok riadenia príkazu opakovania (trieda `IterationCflow`) má priradený príkaz opakovania (trieda `Iteration`), tok riadenia podmieneného príkazu (trieda `ConditionalCflow`) má priradený podmienený príkaz (trieda `Conditional`) a tok riadenia vetvy podmieneného príkazu (trieda `BranchCflow`) má priradenú danú vetvu podmieneného príkazu (trieda `Branch`). Priradené elementy daného toku riadenia slúžia na vytvorenie vnorených tokov riadenia, čo je zabezpečené operáciou `Rebuild`.

Ako naznačuje trieda `CflowVisitor`, aj v tomto prípade bol použitý návrhový vzor Visitor. Čiže všetky triedy predstavujúce toky riadenia definujú operáciu `Accept` a pre všetky takéto triedy je v triede `CflowVisitor` definovaná príslušná operácia `Visit`. Objekty tried odvodených od `CflowVisitor` sa využívajú napríklad pri vytváraní diagramu z daného toku riadenia.

Algoritmus vytvorenia toku riadenia

Tento algoritmus sa používa na vytvorenie celého toku riadenia metódy (vrátane vnorených tokov) z vnútornej reprezentácie programu. Jeho vstupom je tok riadenia metódy. Tomuto toku algoritmus vytvorí vnorené toky riadenia. Pre názornosť je uvedená len základná verzia algoritmu, ktorá nevytvára toky riadenia rád.

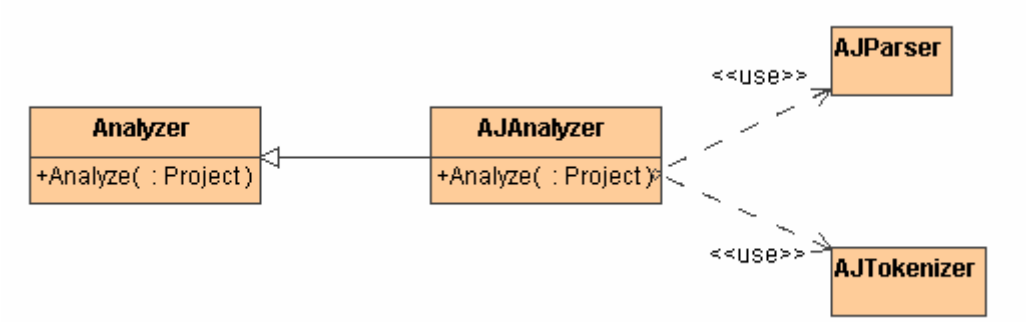
```
Algoritmus:

Vstup:      tok riadenia metódy
Výstup:     tok riadenia metódy s vytvorenými vnorenými tokmi riadenia

Kroky algoritmu:
1) vymaž vnorené toky riadenia vstupného toku
2) získaj príkazy vstupného toku riadenia
      2a) ak je vstup tok riadenia metódy
            1) získaj príkazy priradenej metódy
      2b) ak je vstup tok riadenia volania
            1) vyhodnoť cieľovú metódu priradeného volania
            2) získaj príkazy vyhodnotenej metódy
      2c) ak je vstup tok riadenia príkazu opakovania
            1) získaj príkazy priradeného príkazu opakovania
      2d) ak je vstup tok riadenia podmieneného príkazu
            1) získaj príkazy (vetvy) priradeného podmieneného príkazu
3) pre každý získaný príkaz
      1) vytvor prázdny tok riadenia daného príkazu
            1a) ak je to príkaz volania
                  1) vytvor tok riadenia volania
            1b) ak je to príkaz opakovania
                  1) vytvor tok riadenia príkazu opakovania
            1c) ak je to podmienený príkaz
                  1) vytvor tok riadenia podmieneného príkazu
            1d) ak je to vetva podmieneného príkazu
                  1) vytvor tok riadenia vetvy podmieneného príkazu
      2) pridaj vytvorený tok riadenia do zoznamu vnorených tokov riadenia
         vstupného toku
4) pre každý vnorený tok riadenia vstupného toku riadenia zavolaj
   rekurzívne tento algoritmus
```

Dôležitý v uvedenom algoritme je predovšetkým krok, ktorý pre tok riadenia volania vyhodnotí cieľovú metódu priradeného volania. Táto metóda je určená pomocou cieľovej triedy volania, ktorú môže používateľ zmeniť. Týmto spôsobom je používateľovi umožnené upravovať na diagrame zobrazený tok riadenia.

Poznámka: Implementácia algoritmu vytvorenia toku riadenia je uvedená v časti 4.3 technickej dokumentácie.

Modul Analyzátor

Modul analyzátor má za úlohu vykonať statickú analýzu zdrojových súborov vstupného programu a na základe tejto analýzy vytvoriť vnútornú reprezentáciu programu. Hrubý návrh tohto modulu je tvorený diagramom tried uvedeným na obrázku 5.12.



Obrázok 5.12: Hrubý návrh modulu Analyzátor

Rozhranie tohto modulu tvorí trieda `Analyzer`. Táto trieda deklaruje abstraktnú operáciu `Analyze` so vstupným parametrom typu `Project`. Takto je zabezpečená možnosť výmeny modulu Analyzátor buď inou implementáciou statického analyzátora jazyka AspectJ prípadne vytvorením analyzátora pre iný aspektovo-orientovaný jazyk, ktorý by mal podobné prvky ako AspectJ (aspekty, rady ...). V našom prípade je od abstraktnej triedy `Analyzer` odvodená konkrétna trieda `AJAnalyzer`, ktorá implementuje operáciu `Analyze` a predstavuje analyzátor pre jazyk AspectJ. Pri tejto analýze používa lexikálny analyzátor jazyka AspectJ (trieda `AJTokenizer`) a syntaktický analyzátor jazyka AspectJ (trieda `AJParser`).

Činnosť modulu Analyzátor sa dá rozdeliť do niekoľkých krokov:

1. krok: lexikálna analýza
2. krok: syntaktická analýza
3. krok: spracovanie sémantiky
4. krok: vytvorenie vnútornej reprezentácie programu

Lexikálna analýza

Lexikálna analýza zabezpečuje transformáciu textu vstupného programu do postupnosti lexikálnych jednotiek. Vo všeobecnosti potrebujeme na vytvorenie lexikálneho analyzátora poznať gramatiku lexikálnych jednotiek jazyka (identifikátorov, čísel, kľúčových slov ...). Táto gramatika býva zvyčajne regulárnou gramatikou. Pri vytváraní lexikálneho analyzátora

môžeme použiť existujúce programy, ktoré na základe gramatiky lexikálneho analyzátora a ďalších dodatočných informácii (napr. príkazov, ktoré majú byť pri nájdení konkrétnej lexikálnej jednotky vykonané) vytvoria zdrojový súbor lexikálneho analyzátora jazyka. Najrozšírenejším takýmto generátorom je program Lex [7]. Pôvodne bol vytvorený pre generovanie zdrojových súborov v jazyku C. V súčasnosti existujú rôzne jeho nadstavby a modifikácie generujúce zdrojové súbory lexikálneho analyzátora pre rôzne iné jazyky (napr. JLex pre jazyk Java).

Syntaktická analýza

Vstupom syntaktickej analýzy je postupnosť lexikálnych jednotiek. Výsledkom syntaktickej analýzy je syntaktický strom. Tento syntaktický strom má podobnú štruktúru ako vnútorná reprezentácia programu. Naviac obsahuje informácie potrebné pre spracovanie sémantiky a je viac závislý od gramatiky jazyka ako vnútorná reprezentácia programu. Pri vytváraní syntaktického analyzátora môžeme podobne ako pri lexikálnom analyzátore použiť program na generovanie syntaktického analyzátora z gramatiky jazyka a ďalších informácii špecifikujúcich výsledný analyzátor. V tomto prípade je najrozšírenejším generátorom program Yacc [8]. Často sa programy Lex a Yacc používajú spoločne (syntaktický analyzátor vygenerovaný programom Yacc používa lexikálne jednotky z lexikálneho analyzátora vygenerovaného programom Lex). Takto je napríklad vytvorený programovací jazyk PHP. Program Yacc bol tiež pôvodne vytvorený pre jazyk C, ale súčasné verzie sú vytvorené aj pre iné jazyky (napr. JYacc pre jazyk Java).

Spracovanie sémantiky

Tento krok vytvorí dodatočné väzby medzi objektmi syntaktického stromu. Tieto väzby vychádzajú zo sémantiky daného jazyka. Pre jazyk AspectJ je sémantika prvkov jazyka popísaná v [2, 9].

Príklad vytvorených väzieb je vysvetlený na nasledujúcom programe:

```
package base;
public class Element
{...};

package extended;
public class Line extends base.Element
{...};
```

Z hľadiska syntaxe je časť `extends` deklarácie triedy postupnosť identifikátorov oddelených bodkou. Z hľadiska sémantiky je to špecifikácia nadtriedy, ktorú deklarovaná trieda rozširuje. Vytvorenie väzby v tomto prípade je vyhľadanie objektu syntaktického stromu reprezentujúceho triedu, ktorá je v danom kontexte (kontext balíka `extended`) špecifikovaná názvom `base.Element` a vytvorenie väzby (napríklad formou smerníka) na tento objekt z objektu reprezentujúceho triedu `Line`.

Vytvorenie vnútornej reprezentácie programu

Tento krok transformuje syntaktický strom do objektov vnútornej reprezentácie programu. Pri tejto transformácii sa transformujú aj väzby vytvorené v kroku spracovania sémantiky. Tieto väzby nahradzujú niektoré objekty pôvodného syntaktického stromu (napríklad špecifikáciu nadtriedy v deklarácii triedy) a umožňujú, aby vnútorná reprezentácia programu mala jednoduchšiu štruktúru ako syntaktický strom, z ktorej vznikla.

Gramatika AspectJ

Pre vytvorenie lexikálneho a syntaktického analyzátora AspectJ potrebujeme poznať gramatiku tohto jazyka. Pretože sa jazyk AspectJ neustále vyvíja, neexistuje k nemu ustálená gramatika (nebola zatiaľ publikovaná). Preto súčasťou návrhu bolo aj vytvorenie tejto gramatiky.

Pretože jazyk AspectJ je nadstavbou jazyka Java, dá sa pri návrhu vychádzať z gramatiky jazyka Java. Pretože sa predpokladá použitie programu Lex a Yacc pre vytvorenie lexikálneho a syntaktického analyzátora, môžeme použiť verziu gramatiky jazyka Java pre tieto generátory vytvorenú Dmitrim Bronnikovom. Táto gramatika [10] pozostáva z gramatiky lexikálnych jednotiek jazyka Java a z gramatiky syntaxe jazyka Java.

Gramatika lexikálnych jednotiek jazyka Java bola rozšírená o lexikálne jednotky dodefinované jazykom AspectJ. V podstate boli pridané len nové kľúčové slová (`aspect`, `pointcut`, `before`, `after`, `around` ...) a lexikálna jednotka '`..`' používaná pri deklarácii bodového prierezu.

Pri rozširovaní gramatiky syntaxe jazyka Java sa vychádzalo z dostupných zdrojových súborov programovacieho jazyka AspectJ, ktoré sú asi jediným miestom, kde je gramatika jazyka AspectJ popísaná dostatočne formálne. Analýzou týchto zdrojových súborov sa dajú odvodiť nové pravidlá rozšírenej gramatiky.

Odvodenie časti gramatiky pre pravidlo `BeforeDeclaration` je uvedené na nasledujúcom príklade:

časť zdrojového súboru:

```
class BeforeDecParser extends DecParser {

      public Dec parse() {
            Modifiers modifiers = parseModifiers(Modifiers.STATIC | ...);

            eatKeyword("before");
            Formals formals = parseFormals();
            TypeDs _throws = parseThrows();
            eatTopToken(COLON);

            Pcd pointcut = parsePcd();

            return new BeforeAdviceDec(dummySource, modifiers, ...);
      }
}
```

odvodená gramatika:

Neterminálne symboly:

```
BeforeDeclaration, Modifiers, BeforeDeclarator, Throws, Pointcut, Block,
ParameterList
```

Terminálne symboly:

```
BEFORE, (, )
```

Pravidlá gramatiky:

```
BeforeDeclaration -> Modifiers BeforeDeclarator Throws : Pointcut Block
BeforeDeclaration -> Modifiers BeforeDeclarator        : Pointcut Block
BeforeDeclaration ->           BeforeDeclarator Throws : Pointcut Block
BeforeDeclaration ->           BeforeDeclarator        : Pointcut Block
BeforeDeclaration -> BEFORE ( ParameterList )
BeforeDeclaration -> BEFORE (               )
```

Poznámka: Navrhnuté rozšírenie gramatiky jazyka Java o nové prvky jazyka AspectJ je uvedené v časti A.2 technickej dokumentácie.

5.3. Implementácia

Výsledkom implementácie bol prototyp prostredia na vizualizáciu aspektov prostredníctvom navrhnutého spôsobu vizualizácie. Tento prototyp slúžil na overenie navrhnutej vizualizácie. V tejto časti opíšem vybraný implementačný jazyk a prostredie, uvediem existujúce knižnice a programy, ktoré boli použité pri implementácii prototypu. Okrem toho budú uvedené implementované časti prototypu a ukážka používateľského rozhrania.

Výber implementačného jazyka

Výsledný prototyp na overenie vizualizácie bol implementovaný v prostredí Microsoft Visual Studio 6.0 v jazyku C++. Toto prostredie umožňuje pohodlné vytváranie grafického používateľského rozhrania prostredníctvom knižnice MFC (Microsoft Foundation Classes). Naviac doporučená architektúra MFC aplikácii je podobná architektúre vytvorenej v návrhu. Touto architektúrou je architektúra dokument-pohľad [11]. Dokumentom je v tomto prípade projekt a pohľad naň predstavuje editor zdrojového súboru, editor sekvenčného diagramu, editor filtra, správa projektu a navigačný panel. Okrem toho výber tohto implementačného jazyka a prostredia boli podmienené mojimi predchádzajúcimi skúsenosťami s ním. Nevýhodou je, že knižnica MFC nie je prenositeľná a obmedzuje výslednú aplikáciu len na platformu Windows. V podstate sú od MFC závislé len moduly implementujúce používateľské rozhranie. Modul Projekt a Analyzátor boli implementované nezávisle od MFC a pri ich implementácii boli použité len prenositeľné knižnice, čiže môžu byť prenesené na inú platformu (napr. Linux). Mohlo by sa zdať, že vo vytvorenom systéme je malý počet od platformy nezávislých modulov (2 zo 7), treba si však uvedomiť, že implementácia týchto dvoch modulov predstavuje približne 75% celkovej implementácie.

Znovupoužitie

Pri vytváraní prototypu boli použité niektoré existujúce knižnice a tiež boli použité programy slúžiace na generovanie kódu. Zoznam použitých knižníc a generátorov kódu uvádza tabuľka 5.1.

	názov	**verzia**	**použitie**
knižnice	MFC	6.0	Triedy knižnice Microsoft Foundation Classes boli použité pri implementácii používateľského rozhrania prototypu.
	STL	6.0	Knižnica Standard Template Library implementuje rozšírené dátové typy použité v moduloch prototypu.
	Xerces-C	1.7	Knižnicu Xerces-C prototyp používa na čítanie a zápis súboru vo formáte XML [12]. V tomto formáte je ukladaný súbor s nastaveniami projektu (prípona `ajp`).
generátory	Flex	2.5	Program Flex (prepis programu Lex), bol použitý na generovanie zdrojového súboru lexikálneho analyzátora jazyka AspectJ.
	Bison	1.24	Program Bison (prepis programu Yacc), bol použitý na generovanie zdrojového súboru syntaktického analyzátora jazyka AspectJ.

Tabuľka 5.1: Použité knižnice a generátory kódu

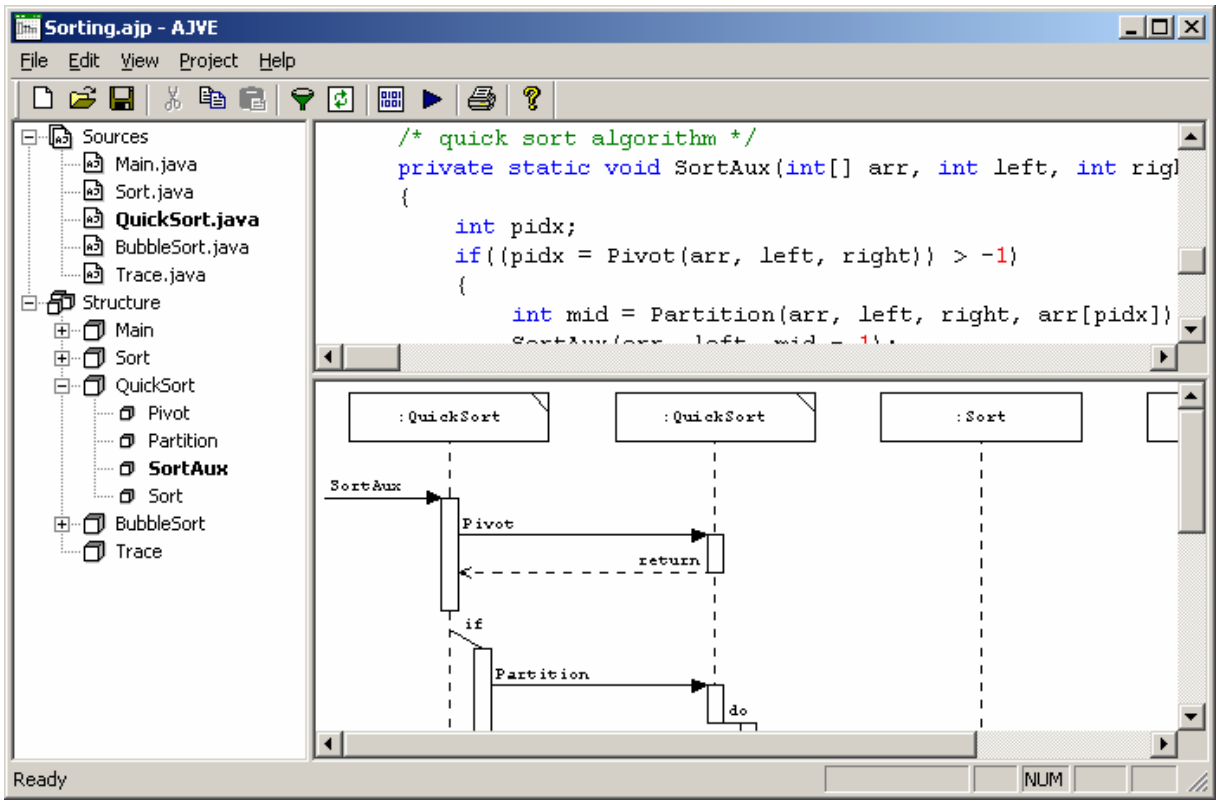
Implementované časti

Z pohľadu špecifikácie boli splnené všetky požiadavky kladené na prototyp. Vzhľadom na rozsiahlosť projektu boli však prijaté niektoré zjednodušenia, týkajúce sa statickej analýzy vstupného programu a filtrovania zobrazených sekvenčných diagramov. Výsledkom týchto zjednodušení je napríklad to, že prototyp nerozpozná zavedenia nachádzajúce sa v zdrojových súboroch projektu a vie rozpoznať len bodové prierezy s elementárnymi popisovačmi `call` a `execution`. Naviac bol implementovaný len jeden z navrhovaných spôsobov filtrovania, ktorým je filtrovanie metód zadaných prostredníctvom množiny filtrovaných značiek.

Používateľské rozhranie

Používateľské rozhranie prototypu je vytvorené pomocou MFC a je typu SDI (Single Document Interface). Použitie SDI má za následok, že vo vytvorenom prototype je možné súčasne otvoriť len jeden projekt. Ukážka používateľského rozhrania prototypu je uvedená na obrázku 5.13. Pozostáva z navigačného panelu, editora zdrojového súboru, editora sekvenčného diagramu, panelu nástrojov a menu.

Poznámka: Podrobnejší popis jednotlivých ovládacích prvkov rozhrania možno nájisť v používateľskej príručke prototypu.



Obrázok 5.13: Ukážka používateľského rozhrania

5.4. Testovanie

V tejto časti uvádzam spôsob testovania jednotlivých modulov systému počas jeho vytvárania a tiež testovanie vytvoreného prototypu. Úlohou finálneho testovania bolo nielen overenie správnej funkčnosti prototypu ale aj overenie navrhnutého spôsobu vizualizácie.

Testovanie modulov

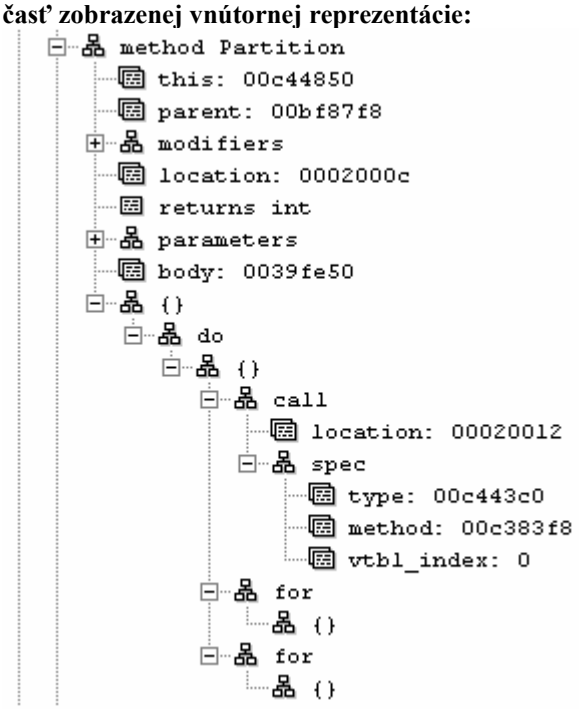
Moduly boli najskôr testované samostatne počas ich vytvárania. Toto testovanie spočívalo vo vytvorení testovacích vstupov pre modul a kontrolovaný bol výstup modulu pre tieto vstupy. Často boli do tried modulu za účelom testovania pridané odlaďovacie metódy na vytvorenie kontrolných výstupov, pričom deklarácie a definície týchto metód boli uvedené v podmienenom príkaze preprocesora riadenom prepínačom, čo umožňuje ich vylúčenie vo finálnej verzii systému. Pretože sa v mnohých moduloch prototypu vytvára hierarchická štruktúra objektov, bola pre testovacie účely vytvorená externá aplikácia umožňujúca zobrazenie vytvorenej hierarchie objektov, pričom odlaďovacie metódy používajú rozhranie definované touto aplikáciou na posielanie informácii o kontrolovaných objektoch.

Príkladom je vnútorná reprezentácia programu vytváraná v module analyzátor. Ukážku testovacieho vstupu a zobrazenia vnútornej reprezentácie programu pre tento vstup uvádza obrázok 5.14.

časť zdrojového súboru:

```
private int Partition(...)
{
	int i = left;
	int j = right;
	do
	{
		Switch(arr, i, j);
		for(;arr[i] < pivot;++i);
		for(;arr[j] >= pivot;--j);
	}
	while(i <= j);
	return i;
};
```

časť zobrazenej vnútornej reprezentácie:



Obrázok 5.14: Ukážka testovania modulu

Okrem funkčnosti modulu bola overená aj správna dealokácia alokovaných objektov modulu. Zistené nesprávne dealokácie (memory leaks) boli lokalizované a následne opravené v zdrojových súboroch programu. Na zistenie nesprávnej dealokácie objektov boli použité nástroje vývojového prostredia Microsoft Visual C++.

Testovanie prototypu

Výsledný prototyp prostredia bol testovaný tak, že boli v tomto prostredí vytvárané AspectJ programy. Pri vytváraní týchto programov bola testovaná možnosť vytvárania nových projektov, pridávania zdrojových súborov do projektu, kompilácia a spúšťanie projektu, uloženie projektu na disk a jeho obnova z disku. Okrem toho bola testovaná správnosť generovania sekvenčných diagramov (aj pre čiastočne vytvorené programy), navigácia v programe prostredníctvom diagramov a možnosť filtrovania týchto diagramov. Tiež bola sledovaná vhodnosť navrhnutej vizualizácie z pohľadu integrácie do prostredia a z pohľadu zobrazenia účinkov aspektov na vytváraný systém.

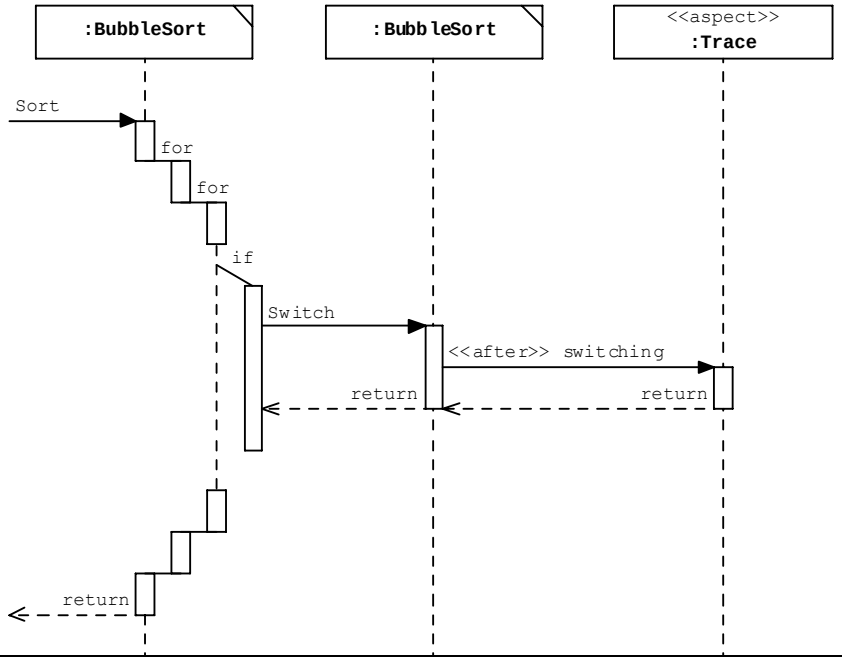
Ukážka prostredím vytvoreného diagramu pre metódu programu je uvedená na obrázku 5.15. Zobrazená je na ňom časť programu na triedenie polí a sekvenčný diagram metódy `Sort` triedy `BubbleSort` tohto programu.

Poznámka: Všetky diagramy nachádzajúce sa v častiach 4.2 a 4.3 boli tiež vytvorené v prototype prostredia. Niektoré z nich museli byť dodatočne editované kvôli neimplementovaným častiam prototypu.

časť programu:

```
public class BubbleSort extends Sort
{
	protected static void Switch(int[] arr, int i, int j)
	{
		int tmp = arr[i];
		arr[i] = arr[j];
		arr[j] = tmp;
	};
	public void Sort(int[] arr)
	{
		int l = arr.length;
		for(int i = l - 1; i > 0; --i)
			for(int j = 0; j < i; ++j)
				if(arr[j] > arr[j + 1])
					Switch(arr, j, j + 1);
	};
};

aspect Trace
{
	pointcut switching() : execution(* BubbleSort.Switch(..));
	after() : switching()
		{ ... };
};
```

vytvorený diagram:



Obrázok 5.15: Ukážka prostredím vytvoreného diagramu pre metódu programu

6. Súvis diplomovej práce s diplomovým projektom

Diplomová práca nadväzuje na diplomový projekt, ktorý som vytváral v prvom ročníku inžinierskeho štúdia. Témou tohto projektu bola Vizualizácia aspektov v aspektovo-orientovanom programovaní. V tomto diplomovom projekte bol navrhnutý spôsob vizualizácie modifikáciou sekvenčných diagramov, vyplývajúci z vykonanej analýzy jazyka AspectJ a existujúcich prostredí pre tento jazyk. Pretože som zistil, že aspektovo-orientované programovanie je veľmi všeobecne definované, bola táto vizualizácia obmedzená len pre tento konkrétny aspektovo-orientovaný jazyk. Vizualizácia bola overená prototypom, ktorý však nevykonával túto vizualizáciu na základe statickej analýzy vstupného programu, ale prostredníctvom pomocného súboru so špecifikáciou štruktúry programu. Naviac prototyp nebol implementovaný ako prostredie, ale len ako jednoduchý program generujúci html stránky s diagramami.

V diplomovej práci bola prehĺbená analýza existujúcich prostredí. Vizualizácia bola prepracovaná tak, aby mohla byť použitá aj pri návrhu systému (rozšírením UML pomocou stereotypov). Bola do nej pridaná možnosť vizualizácie zavedení operácií a tiež boli analyzované a navrhnuté možnosti filtrovania výslednej vizualizácie. Prototyp bol od základu prepracovaný, aby umožňoval vizualizáciu na základe statickej analýzy vstupného programu. Naviac bol prototyp vytvorený ako samostatné prostredie, ktoré v terajšom stave umožňuje vyvíjať jednoduché programy v jazyku AspectJ.

7. Zhodnotenie a záver

Ako bolo spomenuté v úvode, cieľom práce bolo navrhnutie nového spôsobu vizualizácie aspektov, ktorý by sa dal použiť v prostrediach pre programovací jazyk AspectJ. Tento cieľ bol splnený, pričom navrhnutý spôsob poskytuje iný pohľad na vytváraný systém (správanie systému) ako existujúce vizualizácie implementované v doterajších prostrediach pre jazyk AspectJ (statická štruktúra). Výhodou navrhnutého spôsobu je, že je založený na známej notácii sekvenčných diagramov definovaných v UML. Podobný prístup na vizualizáciu zvolili aj autori najnovšej verzie (6.0) prostredia JavaBuilder, ktorí používajú UML notáciu diagramov tried na zobrazenie statickej štruktúry programu. Výsledkom návrhu vizualizácie bolo okrem vizualizácie aj rozšírenie časti metamodelu UML zodpovedajúcej sekvenčným diagramom o aspekty. Toto rozšírenie bolo vytvorené pomocou stereotypov a umožňuje zobraziť účinky aspektov na interakciu objektov systému už v etape analýzy a návrhu systému.

K výsledkom práce patrí aj vytvorený prototyp prostredia na overenie navrhnutej vizualizácie. Pomocou tohto prototypu bola overená možnosť vytvárania sekvenčných diagramov na základe statickej analýzy vstupného programu, možnosť navigácie v programe prostredníctvom zobrazených diagramov a čiastočne aj možnosť filtrovania diagramov (nebola overená pre rozsiahlejšie AspectJ programy). Pri používaní prototypu som zistil, že zobrazené diagramy napomáhajú pochopeniu funkcie vizualizovaného AspectJ programu a zlepšujú navigáciu v programe. Domnievam sa, že navrhnutá vizualizácia je vhodná na použitie v prostrediach pre jazyk AspectJ. Nemyslím si však, že by úplne nahradila existujúce zobrazenie štruktúry programu, ale bola by doplnkovou metódou vizualizácie v týchto prostrediach. Okrem toho by mohla byť implementovaná (bez zobrazenia aspektov) aj v prostrediach pre objektovo-orientované jazyky, v ktorých by bola tiež komplementárnou k zobrazeniu štruktúry programu.

8. Zoznam použitej literatúry

[1] Kiczales, G. et al.: *Aspect-Oriented Programming*. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Springler-Verlang, 1997. http://www.parc.xerox.com/aop

[2] The AspectJ Team: *The AspectJ Programming Guide*. http://aspectj.org/doc/dist/progguide, 2001.

[3] OMG: *OMG Unified Modeling Language Specification Version 1.3*. OMG document ad/99-06-08. ftp://ftp.omg.org/pub/docs/ad/99-06-08.pdf , 1999.

[4] Suzuki, J., Yamamoto, Y.: *Exetending UML with Aspects: Aspect Support in Design Phase*. In: Proceedings of Aspect-Oriented Programming Workshop at ECOOP, 1999. http://citeseer.nj.nec.com/suzuki99extending.html

[5] Suzuki, J., Yamamoto, Y.: *Extending UML for Modeling Reflective Software Components*. http://www.yy.cs.keio.ac.jp/~suzuki/project/pub/uml99.pdf.zip, 1999.

[6] Gamma, E., Helm, R., Johnson, R. and Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[7] Lesk, M.E., Schmidt, E.: *Lex – A lexical analyzer generator*. Computing Science Technical Report No. 39, 1975, Bell Laboratories, Murray Hill, NJ.

[8] Johnson, S. C., Yacc: *Yet Another Compiler Compiler*. Computing Science Technical Report No. 32, 1975, Bell Laboratories, Murray Hill, NJ.

[9] Gosling, J., Joy, B., et Steele, G.: *The Java Language Specification 2nd edition*. Addison Wesley, 2000. http://java.sun.com/docs/books/jls

[10] Bronnikov D.: *Free Yacc-able Java grammar*. http://home.inreach.com/bronikov/grammars/java.html, 1998.

[11] Kruglinski, D.J.: *Mistrovství ve Visual C++*. Brno: Computer Press, 1999.

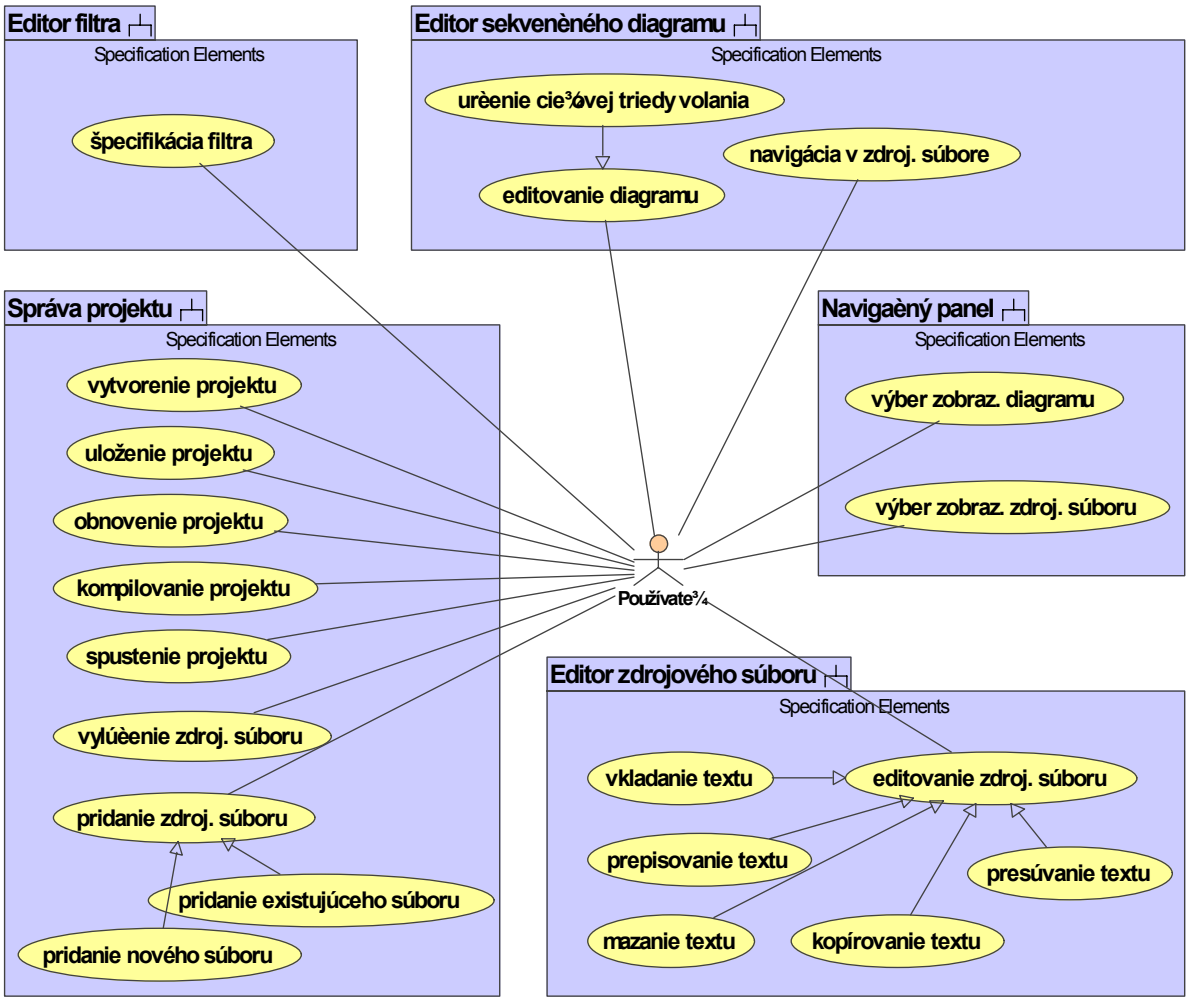
[12] W3C: *Extensible Markup Language (XML) 1.0 (Second Edition)*. http://www.w3.org/TR/2000/REC-xml-20001006, 2000.

Príloha A: Technická dokumentácia

A.1. Špecifikácia

Diagram prípadov použitia systému

Diagram prípadov použitia systému uvádza obrázok A.1. Za ním nasleduje popis jednotlivých prípadov použitia rozdelený podľa modulov, ktoré daný prípad použitia zabezpečujú.



Obrázok A.1: Diagram prípadov použitia systému

modul Správa projektu

- vytvorenie projektu - Vytvorenie prázdneho projektu.
- uloženie projektu - Uloženie posledne editovaného projektu na disk.
- obnovenie projektu - Obnovenie uloženého projektu z disku.
- kompilovanie projektu - Skompilovanie projektu kompilátorom jazyka AspectJ.

- spustenie projektu - Spustenie skompilovaného projektu prostredníctvom virtuálneho stroja jazyka Java.
- pridanie zdroj. súboru - Pridanie zdrojového súboru AspectJ programu do projektu. Systém umožňuje buď pridanie nového zdrojového súboru (takýto súbor bude vytvorený), alebo pridanie existujúceho zdrojového súboru do projektu.
- vylúčenie zdroj. súboru - Vylúčenie zdrojového súboru nespôsobí jeho vymazanie z disku. Má len za následok, že takýto súbor nebude možné ďalej editovať v prostredí a nebude zahrnutý do kompilácie projektu.

modul Editor zdrojového súboru

- editovanie zdroj. súboru - Editácia textu zdrojového súboru. Zahŕňa vkladanie, prepisovanie a mazanie textu zdrojového súboru. Okrem toho umožňuje kopírovanie a presúvanie bloku textu.

modul Editor sekvenčného diagramu

- navigácia v zdroj. súbore - Navigácia v zdrojových súboroch projektu prostredníctvom zobrazeného sekvenčného diagramu. Spočíva v možnosti zobrazenia zdrojového súboru a miesta v ňom, kde sa nachádza určený element diagramu (volanie, trieda ...).
- editovanie diagramu - Špecifikácia informácií, ktoré sa nedajú zistiť statickou analýzou zdrojových súborov projektu. Predovšetkým je to určenie cieľovej triedy cieľového objektu zobrazeného volania.

modul Editor filtra

- špecifikácia filtra - Určenie elementov diagramu, ktoré majú byť vynechané zo zobrazenia.

modul Navigačný panel

- výber zobr. diagramu - Výber počiatočnej metódy, ktorej tok riadenia má byť zobrazený na sekvenčnom diagrame v editore sekvenčného diagramu.
- výber zobr. zdroj. súboru - Výber zdrojového súboru patriaceho do projektu, ktorý má byť zobrazený v editore zdrojového súboru.

Formát vstupného súboru projektu

Vstupný súbor projektu (prípona `ajp`) obsahuje rôzne nastavenia projektu. Tento súbor je XML dokument. Presný formát tohto súboru je určený nasledujúcou špecifikáciou vo formáte DTD, ktorá je zároveň použitá pri validácii súboru projektu:

```
<?xml version="1.0"?>

<!ELEMENT project (sources | cflows | filter)*>
<!ELEMENT sources (source)*>
<!ELEMENT cflows (cflow)*>
<!ELEMENT filter (tag)*>
<!ELEMENT source (#PCDATA)>
<!ELEMENT cflow (#PCDATA)>
<!ELEMENT tag (#PCDATA)>

<!ATTLIST   project
            main        CDATA           #IMPLIED>

<!ATTLIST   source
            name        CDATA           #REQUIRED
            path        CDATA           #REQUIRED>

<!ATTLIST   cflow
            type        CDATA           #REQUIRED
            method      CDATA           #REQUIRED>

<!ATTLIST   tag
            name        CDATA           #REQUIRED>

<!ATTLIST   sources
            selected    CDATA           #IMPLIED>

<!ATTLIST   cflows
            selected    CDATA           #IMPLIED>
```

A.2. Návrh

Rozšírenie gramatiky jazyka Java

Navrhnuté rozšírenie gramatiky syntaktického analyzátora Javy o nové prvky jazyka AspectJ akceptuje v súčasnom stave len bodové prierezy a rady aspektu. Zápis pravidiel rozširujúcich gramatiku jazyka Java je totožný so zápisom gramatiky vo vstupných súboroch programu Yacc. Slová s veľkými písmenami (všetky veľké) predstavujú terminálne symboly gramatiky a ostatné slová sú neterminálne symboly. Pravidlá sa zapisujú tak, že najprv sa napíše ľavá strana pravidla, za ktorou nasleduje znak ':' a potom nasledujú alternatívne pravé strany pravidiel oddelené znakom '|'. Pravidlo sa končí znakom ';'. Rozširujúce pravidlá sú vytvorené na rozšírenie gramatiky jazyka Java pre program Yacc navrhnutej Dmitrim Bronnikovom. Sú to nasledujúce pravidlá:

```
TypeDeclaration
   | AspectHeader '{' AspectFieldDecs '}'
   | AspectHeader '{' '}'
   ;

AspectHeader
   : Modifiers ASPECT IDENTIFIER Extends Interfaces AspectExtension
   | Modifiers ASPECT IDENTIFIER Extends Interfaces
   | Modifiers ASPECT IDENTIFIER Extends            AspectExtension
   | Modifiers ASPECT IDENTIFIER Extends
   | Modifiers ASPECT IDENTIFIER         Interfaces AspectExtension
   | Modifiers ASPECT IDENTIFIER         Interfaces
   | Modifiers ASPECT IDENTIFIER                    AspectExtension
   | Modifiers ASPECT IDENTIFIER
   |           ASPECT IDENTIFIER Extends Interfaces AspectExtension
   |           ASPECT IDENTIFIER Extends Interfaces
   |           ASPECT IDENTIFIER Extends            AspectExtension
   |           ASPECT IDENTIFIER Extends
   |           ASPECT IDENTIFIER         Interfaces AspectExtension
   |           ASPECT IDENTIFIER         Interfaces
   |           ASPECT IDENTIFIER                    AspectExtension
   |           ASPECT IDENTIFIER
   ;

AspectExtension
   : Dominates Of
   | Dominates
   |           Of
   ;

Dominates
   : DOMINATES Gtn
   ;

Of
   : PERTHIS '(' Pointcut ')'
   | PERTARGET '(' Pointcut ')'
   | PERCFLOW '(' Pointcut ')'
   | PERCFLOWBELOW '(' Pointcut ')'
   | ISSINGLETON '(' ')'
   ;
AspectFieldDecs
   : AspectFieldDecOptSemi
   | AspectFieldDecs AspectFieldDecOptSemi
   ;

AspectFieldDecOptSemi
   : AspectFieldDec
   | AspectFieldDec SemiColons
   ;
```

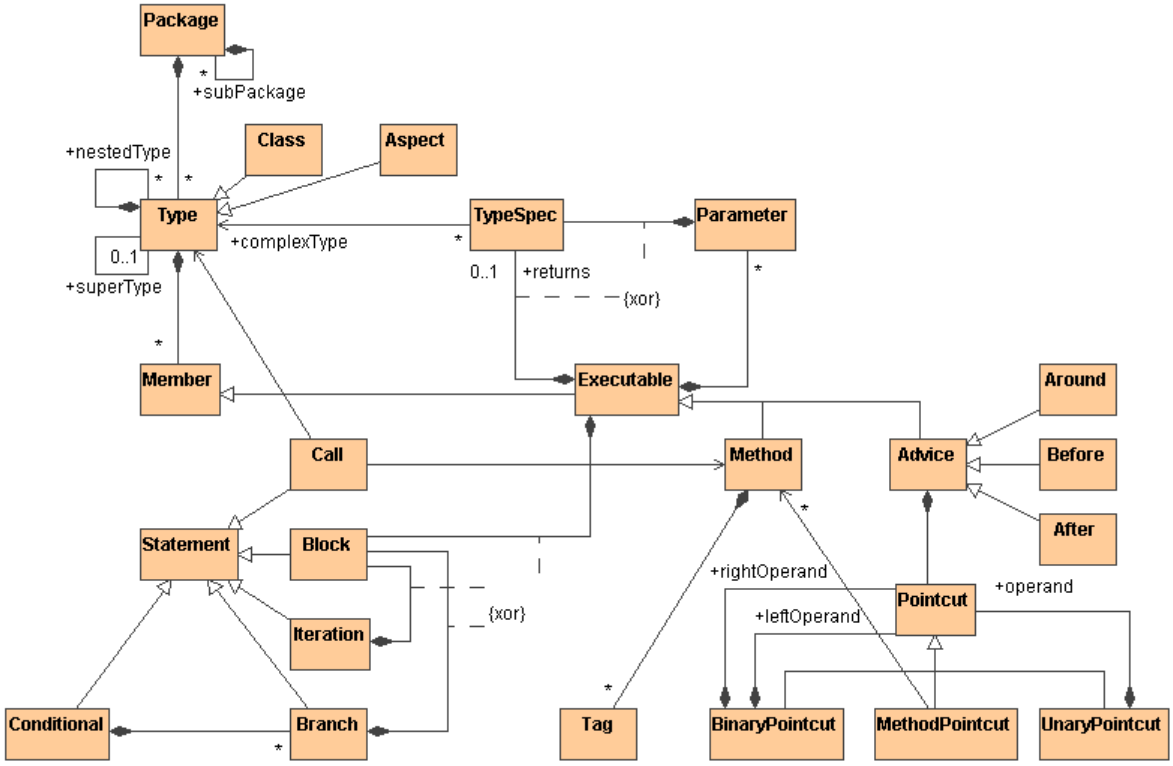
```
AspectFieldDec
   : FieldVariableDeclaration ';'
   | MethodDeclaration
   | ConstructorDeclaration
   | StaticInitializer
   | NonStaticInitializer
   | PointcutDeclaration ';'
   | BeforeDeclaration
   | AfterDeclaration
   | AroundDeclaration
   | TypeDeclaration
   ;

PointcutDeclaration
   : PointcutDeclarationHeader ':' Pointcut
   | PointcutDeclarationHeader
   ;

PointcutDeclarationHeader
   : Modifiers POINTCUT IDENTIFIER '(' ParameterList ')' Returns
   | Modifiers POINTCUT IDENTIFIER '(' ParameterList ')'
   | Modifiers POINTCUT IDENTIFIER '('               ')' Returns
   | Modifiers POINTCUT IDENTIFIER '('               ')'
   |           POINTCUT IDENTIFIER '(' ParameterList ')' Returns
   |           POINTCUT IDENTIFIER '(' ParameterList ')'
   |           POINTCUT IDENTIFIER '('               ')' Returns
   |           POINTCUT IDENTIFIER '('               ')'
   ;

Returns
   : RETURNS TypeSpecifier
   ;

Pointcut
   : OrPointcutExpression
   ;

OrPointcutExpression
   : AndPointcutExpression
   | OrPointcutExpression OP_LOR AndPointcutExpression
   ;

AndPointcutExpression
   : UnaryPointcutExpression
   | AndPointcutExpression OP_LAND UnaryPointcutExpression
   ;

UnaryPointcutExpression
   : PointcutPrimary
   | '!' PointcutPrimary
   ;
PointcutPrimary
   : '(' Pointcut ')'
   | QualifiedName        '(' FormalsPattern ')'
   | QualifiedName        '('                ')'
   | CALL                 '(' ConstructorOrMethodPattern ')'
   | EXECUTION            '(' ConstructorOrMethodPattern ')'
   | INITIALIZATION       '(' ConstructorOrMethodPattern ')'
   | GET                  '(' FieldPattern ')'
   | SET                  '(' FieldPattern ')'
   | THIS                 '(' Gtn ')'
   | TARGET               '(' Gtn ')'
   | STATICINITIALIZATION '(' Gtn ')'
   | WITHIN               '(' Gtn ')'
   | WITHINCODE           '(' Gtn ')'
   | HANDLER              '(' Gtn ')'
   | CFLOW                '(' Pointcut ')'
   | CFLOWBELOW           '(' Pointcut ')'
   | ARGS                 '(' FormalsPattern ')'
   | ARGS                 '('                ')'
   | IF                   '(' Expression ')'
   ;

FormalsPattern
   : FormalPattern
   | FormalsPattern ',' FormalPattern
   ;
```

```
FormalPattern
   : Gtn
   | DOUBLEDOT
   ;

FieldPattern
   : ModifiersPattern Gtn TypeNameAndId
   |                  Gtn TypeNameAndId
   ;

ModifiersPattern
   : ModifierPattern
   | ModifiersPattern ModifierPattern
   ;

ModifierPattern
   : Modifier
   | '!' Modifier
   ;


TypeNameAndId
   : GtnPrimary '.' PATTERNNAME
   | PATTERNNAME
   ;

ConstructorOrMethodPattern
   : MethodPattern
   | ConstructorPattern
   ;

ConstructorPattern
   : ModifiersPattern TypeNameAndId '(' FormalsPattern ')' Throws
   | ModifiersPattern TypeNameAndId '(' FormalsPattern ')'
   | ModifiersPattern TypeNameAndId '('                ')' Throws
   | ModifiersPattern TypeNameAndId '('                ')'
   |                  TypeNameAndId '(' FormalsPattern ')' Throws
   |                  TypeNameAndId '(' FormalsPattern ')'
   |                  TypeNameAndId '('                ')' Throws
   |                  TypeNameAndId '('                ')'
   ;

MethodPattern
   : ModifiersPattern Gtn TypeNameAndId '(' FormalsPattern ')' Throws
   | ModifiersPattern Gtn TypeNameAndId '(' FormalsPattern ')'
   | ModifiersPattern Gtn TypeNameAndId '('                ')' Throws
   | ModifiersPattern Gtn TypeNameAndId '('                ')'
   |                  Gtn TypeNameAndId '(' FormalsPattern ')' Throws
   |                  Gtn TypeNameAndId '(' FormalsPattern ')'
   |                  Gtn TypeNameAndId '('                ')' Throws
   |                  Gtn TypeNameAndId '('                ')'
   ;

Gtn
   : OrGtnExpression
   ;

OrGtnExpression
   : AndGtnExpression
   | OrGtnExpression OP_LOR AndGtnExpression
   ;

AndGtnExpression
   : UnaryGtnExpression
   | AndGtnExpression OP_LAND UnaryGtnExpression
   ;

UnaryGtnExpression
   : GtnPrimary
   | '!' GtnPrimary
   ;
```

```
GtnPrimary
   : '(' Gtn ')'
   | PATTERNNAME
   | ExtendetPatternName
   ;

ExtendetPatternName
   : PATTERNNAME '+' Dims
   | PATTERNNAME '+'
   | PATTERNNAME     Dims
   ;

BeforeDeclaration
   : Modifiers BeforeDeclarator Throws ':' Pointcut Block
   | Modifiers BeforeDeclarator        ':' Pointcut Block
   |           BeforeDeclarator Throws ':' Pointcut Block
   |           BeforeDeclarator        ':' Pointcut Block
   ;

BeforeDeclarator
   : BEFORE '(' ParameterList ')'
   | BEFORE '('               ')'
   ;

AfterDeclaration
   : Modifiers AfterDeclarator Exiting Throws ':' Pointcut Block
   | Modifiers AfterDeclarator Exiting        ':' Pointcut Block
   |           AfterDeclarator Exiting Throws ':' Pointcut Block
   |           AfterDeclarator Exiting        ':' Pointcut Block
   | Modifiers AfterDeclarator         Throws ':' Pointcut Block
   | Modifiers AfterDeclarator                ':' Pointcut Block
   |           AfterDeclarator         Throws ':' Pointcut Block
   |           AfterDeclarator                ':' Pointcut Block
   ;

AfterDeclarator
   : AFTER '(' ParameterList ')'
   | AFTER '('               ')'
   ;

Exiting
   : RETURNING '(' Parameter ')'
   | RETURNING '('           ')'
   | RETURNING
   | THROWING  '(' Parameter ')'
   | THROWING  '('           ')'
   | THROWING
   ;

AroundDeclaration
   : Modifiers TypeSpecifier AroundDeclarator Throws ':' Pointcut Block
   | Modifiers TypeSpecifier AroundDeclarator        ':' Pointcut Block
   |           TypeSpecifier AroundDeclarator Throws ':' Pointcut Block
   |           TypeSpecifier AroundDeclarator        ':' Pointcut Block
   ;

AroundDeclarator
   : AROUND '(' ParameterList ')'
   | AROUND '('               ')'
   ;
```

Diagram tried vnútornej reprezentácie programu

Vnútorná reprezentácia programu predstavuje hierarchickú štruktúru objektov, ktoré predstavujú jednotlivé časti (balíky, triedy ...) vstupného programu v jazyku AspectJ. Diagram tried vnútornej reprezentácie programu je uvedený na obrázku A.2. Za ním nasleduje popis jednotlivých tried uvedených na diagrame.



Obrázok A.2: Diagram tried vnútornej reprezentácie programu

Triedy

- `Package` - Reprezentuje balík programu. Balík môže obsahovať ďalšie balíky (`subPackage`) a tiež v danom balíku definované typy.
- `Type` - Reprezentuje typy definované v programe. Daný typ môže obsahovať definície vnorených typov (`nestedType`) a môže byť dcérskym typom svojho rodičovského typu (`superType`). Okrem toho obsahuje definície členských prvkov.
- `Class` - Predstavuje triedu definovanú v programe. Trieda je odvodená od typu.
- `Aspect` - Predstavuje aspekt, ktorý je definovaný v programe. Aspekt je odvodený od typu.
- `Member` - Reprezentuje členský prvok nejakého typu programu.

- `Executable` - Reprezentuje vykonateľný prvok definovaný pre typ programu. Môže obsahovať špecifikáciu typu návratovej hodnoty (`returns`) a typov vstupných parametrov.
- `TypeSpec` - Predstavuje špecifikáciu typu. Môže byť buď primitívnym typom (`int`, `bool` ...) alebo typom definovaným v programe (`complexType`). Špecifikácia typu musí byť časťou návratovej hodnoty alebo parametra vykonateľného prvku triedy.
- `Parameter` - Predstavuje parameter vykonateľného prvku (metódy, rady) v programe definovaného typu. Obsahuje špecifikáciu typu daného parametra.
- `Statement` - Predstavuje vykonateľný príkaz.
- `Call` - Predstavuje príkaz volania. Volanie je špecifikované metódou a typom.
- `Block` - Predstavuje blok príkazov programu. Musí byť súčasťou vykonateľného prvku triedy, príkazu opakovania alebo vetvy podmieneného príkazu. Blok je odvodený od vykonateľného príkazu.
- `Iteration` - Reprezentuje príkaz opakovania. Príkaz opakovania je odvodený od vykonateľného príkazu.
- `Conditional` - Reprezentuje podmienený príkaz. Skladá sa z vetví podmieneného príkazu. Podmienený príkaz je odvodený od vykonateľného príkazu.
- `Branch` - Reprezentuje vetvu podmieneného príkazu. Vetva podmieneného príkazu je odvodená od vykonateľného príkazu.
- `Method` - Predstavuje metódu definovaná v type programu. Metóda je odvodená od vykonateľného prvku typu. Môže obsahovať pre ňu definované značky.
- `Tag` - Reprezentuje značku definovanú v programe.
- `Advice` - Reprezentuje radu programu. Rada je odvodená od vykonateľného prvku typu. Obsahuje bodový prierez.
- `Before` - Reprezentuje radu `before`. Odvodená je od rady.
- `After` - Reprezentuje radu `after`. Odvodená je od rady.
- `Around` - Reprezentuje radu `around`. Odvodená je od rady.
- `Pointcut` - Reprezentuje bodový prierez priradený rade.
- `UnaryPointcut` - Reprezentuje výraz skladajúci sa z bodového prierezu a unárneho operátora (negácia - `!`).
- `BinaryPointcut` - Reprezentuje výraz skladajúci sa z dvoch bodových prierezov a binárneho operátora (a zároveň - `&&`, alebo - `||`).
- `MethodPointcut` - Reprezentuje bodový prierez, ktorý sa viaže na metódy programu.

A.3. Implementácia

Rozdelenie tried do modulov

Tabuľka A.1 uvádza rozdelenie implementačných tried do jednotlivých modulov systému a účel týchto tried.

Modul	Trieda	Účel
Analyzátor	CAJAnalyzer, CAnalyzer, CSourceParserInput	statický analyzátor jazyka AspectJ
	CAJParser, CParser, CParserInput, CAJTagger	syntaktický analyzátor jazyka AspectJ
	CAJTokenizer, CTokenizer, CTokenizerInput	lexikálny analyzátor jazyka AspectJ
	CAJTokenKeyword, CAJTokenDelimiter, CAJTokenSeparator, CAJTokenBoolean, CAJTokenNumber, CToken, CTokenSpecial, CTokenComment, CTokenIdentifier, CTokenKeyword, CTokenDelimiter, CTokenSeparator, CTokenLiteral, CTokenBoolean, CTokenNumber, CTokenString, CTokenCharacter, CAJTokenModifVisitor, CAJTokenConstVisitor, CTokenModifVisitor, CTokenConstVisitor	lexikálne jednotky jazyka AspectJ
	CAJT, CAJTIdentifier, CAJTQualifiedName, CAJTImportBase, CAJTImportStatement, CAJTImportStatements, CAJTPackageStatement, CAJTCompilationUnit, CAJTTypeBase, CAJTTypeSimple, CAJTTypeName, CAJTModifiers, CAJTExtends, CAJTTypeNameList, CAJTInterfaces, CAJTTypeHeaderBase, CAJTTypeHeaderClass, CAJTTypeHeaderInterface, CAJTType, CAJTNamedType, CAJTClass, CAJTInterface, CAJTTypeDeclarations, CAJTDeclaratorName, CAJTParameter, CAJTParameterList, CAJTGeneralMethodDeclarator, CAJTFieldDeclaration, CAJTGeneralMethodDeclaration, CAJTMethod, CAJTConstructor, CAJTStaticInitializer, CAJTNonstaticInitializer, CAJTFieldDeclarations, CAJTPackage, CAJTExpressionBase, CAJTPrimarySimple, CAJTFieldAccess, CAJTPrimaryQualified, CAJTExpressionList, CAJTMethodCall, CAJTArrayAccess, CAJTExpressionUnary, CAJTExpressionBinary, CAJTExpressionConditional, CAJTExpressionAssignment, CAJTVariableInitializer, CAJTExpressionInitializer, CAJTArrayInitializer, CAJTVariableDeclarator, CAJTVariableDeclarators, CAJTFieldVariableDeclaration, CAJTStatementBase, CAJTStatementLabel, CAJTStatementJump, CAJTStatementSelection, CAJTStatementIteration, CAJTLocalVariableDeclaration, CAJTStatementList, CAJTStatementBlock, CAJTStatementExpression, CAJTNestedType, CAJTPatternName, CAJTGtnBase, CAJTGtnPattern, CAJTGtnUnary, CAJTGtnBinary, CAJTTypeNameAndId, CAJTFieldPattern, CAJTFormalPattern, CAJTFormalsPattern, CAJTGeneralMethodPattern, CAJTConstructorPattern, CAJTMethodPattern, CAJTPointcutBase, CAJTPointcutQualified, CAJTPointcutMethod, CAJTPointcutField, CAJTPointcutGtn, CAJTPointcutFlow, CAJTPointcutArgs, CAJTPointcutIf, CAJTPointcutUnary, CAJTPointcutBinary, CAJTOfClause, CAJTAspectExtension, CAJTTypeHeaderAspect, CAJTAspect, CAJTPointcutHeader, CAJTPointcutDeclaration, CAJTGeneralAdviceDeclaration, CAJTExitingClause, CAJTBefore, CAJTAfter, CAJTAround, CAJTTags, CAJTVisitor	syntaktický strom jazyka AspectJ
	CAJTVisitorContextFinder, CAccess, CAJTVisitorAcceptor, CAJTVisitorPartFinder, CAJTResolver, CAJTVisitorNameFinder, CAJTVisitorImportsLinker, CAJTVisitorTypesLinker, CAJTVisitorExpressionsLinker, CAJTVisitorPatternFinder, CAJTVisitorTypesFinder, CAJTVisitorPointcutsLinker	spracovanie sémantiky
	CAJTVisitorPSTSpecBuilder, CAJTVisitorPSTBuilder, CAJTVisitorPSTTranslator	vytvorenie vnútornej reprezentácie programu

Projekt	CProjectEx, CProject, CGenIStream, CGenOStream, CGenIO, CGenIStreamAdaptor, CGenInputAdaptor, CGenOutputAdaptor	projekt
	CPJSource, CHighlightedSource, CSource, CRange, CPos	zdrojový súbor so zvýraznenou syntaxou jazyka AspectJ
	CSourceHighlighter, CSourceTokenizerInput	zvýraznenie syntaxe jazyka AspectJ
	CModifiers, CPST, CPSTPackage, CPSTType, CPSTClass, CPSTAspect, CPSTMember, CPSTTypeSpec, CPSTParameter, CPSTMemberExecutable, CPSTStatement, CPSTStatementBlock, CPSTStatementIter, CPSTStatementBranch, CPSTStatementCond, CPSTCallSpec, CPSTStatementCall, CPSTMethod, CPSTVTBLEntry, CPSTPointcut, CPSTPointcutUnary, CPSTPointcutBinary, CPSTPointcutMethod, CPSTAdvice, CPSTBefore, CPSTAfter, CPSTAround, CPSTTag, CPSTModifVisitor, CPSTConstVisitor	vnútorná reprezentácia programu
	CPSTVisitorTypesFinder, CPSTVisitorMethodsFinder, CPSTVisitorPackageLinker, CPSTVisitorVTBLIdxLinker	dokončenie vnútornej reprezentácie programu (vytvorenie pomocných väzieb)
	CPJCflow, CPSCflow, CPSCflowIter, CPSCflowCond, CPSCflowBranch, CPSCflowCall, CPSCflowMethod, CPSCflowAdvice, CPSCflowModifVisitor, CPSCflowConstVisitor	tok riadenia
	CPSTVisitorCflowBuilder, CPSTVisitorAdviceFinder, CPSCflowRecursionDetector, CPSTVisitorValidityTester	vytvorenie toku riadenia
	CPJFilter, CPSFilter	filter
	CPSFilterBuilder	dokončenie filtra (vyhľadanie filtrovaných elementov vnútornej reprezentácie programu)
Editor sekvenčného diagramu	CDEView	editor sekvenčného diagramu
	CDEExDiagram , CDEExDrawHelper, CDEExDrawInfo, CDEExExtension, CDEDiagram, CDELayoutInfo, CDEExtension	sekvenčný diagram
	CDEExCflowCallBegin, CDEExCflowAdviceBegin, CDEExCflowCallEnd, CDEExCflowAdviceEnd, CDEExCflowCondBegin, CDEExCflowCondEnd, CDEExCflowIterBegin, CDEExCflowIterEnd, CDEExCflowBranchBegin, CDEExCflowBranchEnd, CDEExCflowCallBack, CDEExPartSolid, CDEExPartDashed, CDEExTimeline, CDECflow, CDECflowCallBegin, CDECflowAdviceBegin, CDECflowCallEnd, CDECflowAdviceEnd, CDECflowIterBegin, CDECflowIterEnd, CDECflowCondBegin, CDECflowCondEnd, CDECflowBranchBegin, CDECflowBranchEnd, CDECflowCallBack, CDETimelinePart, CDETimeline	elementy sekvenčného diagramu
	CDEExDiagramBuilder, CDEExTimelineCreator, CDEExAdviceTester	vytvorenie sekvenčného diagramu z toku riadenia
Editor filtra	CFilterEditorDlg	editor filtra
	CPSTTagsFinder	vyhľadanie všetkých značiek vo vnútornej reprezentácii programu
Editor zdrojového súboru	CCEView	editor zdrojového súboru
Navigačný panel	CAJVEView	navigačný panel
	CAJVEStructBuilder	vytvorenie zobrazenej štruktúry programu (vyhľadanie všetkých typov a ich metód vo vnútornej reprezentácii programu)
Správa projektu	CAJVEDoc	práca s projektom
	CFileGenIStream, CFileGenOStream, CFileGenIO	prístup k súborom disku
	CPrjOpenDlg	dialóg na pridanie súboru do projektu
	CPrjSettingsDlg	dialóg na nastavenie projektu
	CAboutDlg, CAJVEApp, CMainFrame, CMySplit	triedy systému implementujúce kostru aplikácie
Odladťovacia aplikácia	CAboutDlg, CConsoleApp, CConsoleDoc, CDTViewBuilder, CConsoleView, CMainFrame, CServer, CDTVisitor, CDTNode, CDTNodeSimple, CDTNodeString, CDTNodeNterm, CDTNodeSerializer, CDTConsoleComm	triedy implementujúce odladťovaciu aplikáciu

Tabuľka A.1: Rozdelenie implementačných tried do modulov

Ukážka implementovaného algoritmu

Nasledujúca časť programu implementuje algoritmus na vytvorenie toku riadenia z vnútornej reprezentácie programu. Vytváranie začína volaním metódy `Rebuild` nejakej triedy reprezentujúcej konkrétny tok riadenia. Táto metóda vytvára prostredníctvom objektu implementačnej triedy `CPSTVisitorCflowBuilder` vnorené toky riadenia a následne pre každý takýto vnorený tok volá znovu metódu `Rebuild`. Trieda `CPSTVisitorCflowBuilder` je odvodená od základnej triedy `CPSTModifVisitor` reprezentujúcej vizitora elementov vnútornej reprezentácie programu. Takto môže vytvoriť pre konkrétny element vnútornej reprezentácie konkrétny tok riadenia tohto elementu (napr. pre metódu vnútornej reprezentácie tok riadenia metódy).

```
CPSTVisitorCflowBuilder::CPSTVisitorCflowBuilder(CPSCflow* iCflow)
{
	mCflow = iCflow;
	mBefores = NULL;
	mAfters = NULL;
	mArounds = NULL;
}

CPSTVisitorCflowBuilder::CPSTVisitorCflowBuilder(CPSCflow* iCflow, CBefores* iBefores,
CAfters* iAfters, CArounds* iArounds)
{
	mCflow = iCflow;
	mBefores = iBefores;
	mAfters = iAfters;
	mArounds = iArounds;
}

bool CPSTVisitorCflowBuilder::Visit(CPSTStatementBlock* iBlock)
{
	assert(mCflow != NULL);
	int i;

	if(mBefores != NULL)
	{
		CBefores::iterator tB = mBefores->begin();
		CBefores::iterator tE = mBefores->end();
		for(; tB != tE; ++tB)
			mCflow->AddCflow(new CPSCflowAdvice(*tB));
	}

	int tNumStatements = iBlock->NumStatements();
	for(i = 0; i < tNumStatements; ++i)
		iBlock->GetStatement(i)->Accept(this);

	if(mAfters != NULL)
	{
		CAfters::iterator tB = mAfters->begin();
		CAfters::iterator tE = mAfters->end();
		for(; tB != tE; ++tB)
			mCflow->AddCflow(new CPSCflowAdvice(*tB));
	}
	return true;
}

bool CPSTVisitorCflowBuilder::Visit(CPSTStatementIter* iIter)
{
	assert(mCflow != NULL);
	CPSTVisitorValidityTester tValidityTester;
	iIter->Accept(&tValidityTester);
	if(tValidityTester.IsValid())
		mCflow->AddCflow(new CPSCflowIter(iIter));
	return true;
}
```

```
bool CPSTVisitorCflowBuilder::Visit(CPSTStatementCond* iCond)
{
	assert(mCflow != NULL);
	CPSTVisitorValidityTester tValidityTester;
	iCond->Accept(&tValidityTester);
	if(tValidityTester.IsValid())
		mCflow->AddCflow(new CPSCflowCond(iCond));
	return true;
}

bool CPSTVisitorCflowBuilder::Visit(CPSTStatementBranch* iBranch)
{
	assert(mCflow != NULL);
	CPSTVisitorValidityTester tValidityTester;
	iBranch->Accept(&tValidityTester);
	if(tValidityTester.IsValid())
		mCflow->AddCflow(new CPSCflowBranch(iBranch));
	return true;
}

bool CPSTVisitorCflowBuilder::Visit(CPSTStatementCall* iCall)
{
	assert(mCflow != NULL);
	if(iCall->HasSpec())
	{
		CPSTCallSpec* tCallSpec = iCall->GetSpec();
		if(tCallSpec->HasMethod() &&
			tCallSpec->HasType())
			mCflow->AddCflow(new CPSCflowCall(iCall));
	}
	return true;
}

void CPSCflowIter::Rebuild(CPSTPackage* iRoot)
{
	CPSTVisitorCflowBuilder tCflowBuilder(this);
	EmptyCflows();
	if((mIter != NULL) && (mIter->HasBlock()))
	{
		int i;
		mIter->GetBlock()->Accept(&tCflowBuilder);

		int tNumCflows = NumCflows();
		for(i = 0; i < tNumCflows; ++i)
			GetCflow(i)->Rebuild(iRoot);
	}
}

void CPSCflowCond::Rebuild(CPSTPackage* iRoot)
{
	CPSTVisitorCflowBuilder tCflowBuilder(this);
	EmptyCflows();
	if(mCond != NULL)
	{
		int i;

		int tNumBranches = mCond->NumBranches();
		for(i = 0; i < tNumBranches; ++i)
			mCond->GetBranch(i)->Accept(&tCflowBuilder);

		int tNumCflows = NumCflows();
		for(i = 0; i < tNumCflows; ++i)
			GetCflow(i)->Rebuild(iRoot);
	}
}
```

```
void CPSCflowBranch::Rebuild(CPSTPackage* iRoot)
{
	CPSTVisitorCflowBuilder tCflowBuilder(this);
	EmptyCflows();
	if((mBranch != NULL) && (mBranch->HasBlock()))
	{
		int i;
		mBranch->GetBlock()->Accept(&tCflowBuilder);

		int tNumCflows = NumCflows();
		for(i = 0; i < tNumCflows; ++i)
			GetCflow(i)->Rebuild(iRoot);
	}
}

void CPSCflowAdvice::Rebuild(CPSTPackage* iRoot)
{
	CPSTVisitorCflowBuilder tCflowBuilder(this);
	EmptyCflows();
	if((mAdvice != NULL) && (mAdvice->HasBody()))
	{
		int i;
		mAdvice->GetBody()->Accept(&tCflowBuilder);

		int tNumCflows = NumCflows();
		for(i = 0; i < tNumCflows; ++i)
			GetCflow(i)->Rebuild(iRoot);
	}
}

void CPSCflowCall::Rebuild(CPSTPackage* iRoot)
{
	assert(iRoot != NULL);

	mRecursive = false;

	CPSTMethod* tMethod = GetMethod();
	CPSCflowRecursionDetector tRecursionDetector(tMethod);
	EmptyCflows();
	if((tMethod != NULL) && (tMethod->HasBody()))
	{
		/*** skontroluj rekurziu ***/
		if(HasParent())
		{
			GetParent()->Accept(&tRecursionDetector);
			if(tRecursionDetector.IsRecursive())
			{
				mRecursive = true;
				return;
			}
		}

		int i;
		CBefores tBefores;
		CAfters tAfters;
		CArounds tArounds;
		CPSTVisitorAdviceFinder tAdviceFinder(this, &tBefores, &tAfters, &tArounds);
		CPSTVisitorCflowBuilder tCflowBuilder(this, &tBefores, &tAfters, &tArounds);

		iRoot->Accept(&tAdviceFinder);
		tMethod->GetBody()->Accept(&tCflowBuilder);

		int tNumCflows = NumCflows();
		for(i = 0; i < tNumCflows; ++i)
			GetCflow(i)->Rebuild(iRoot);
	}
}
```

```
void CPSCflowMethod::Rebuild(CPSTPackage* iRoot)
{
	assert(iRoot != NULL);

	CPSTVisitorCflowBuilder tCflowBuilder(this);
	EmptyCflows();
	if((mMethod != NULL) && (mMethod->HasBody()))
	{
		int i;
		CBefores tBefores;
		CAfters tAfters;
		CArounds tArounds;
		CPSTVisitorAdviceFinder tAdviceFinder(this, &tBefores, &tAfters, &tArounds);
		CPSTVisitorCflowBuilder tCflowBuilder(this, &tBefores, &tAfters, &tArounds);

		iRoot->Accept(&tAdviceFinder);
		mMethod->GetBody()->Accept(&tCflowBuilder);

		int tNumCflows = NumCflows();
		for(i = 0; i < tNumCflows; ++i)
			GetCflow(i)->Rebuild(iRoot);
	}
}
```

Príloha B: Používateľská príručka

Úvod

Prostredie AJVE (AspectJ Visualisation Environment) slúži na programovanie v jazyku AspectJ. Umožňuje vytváranie AspectJ programov, ich kompilovanie a spúšťanie. Okrem toho zobrazuje sekvenčné diagramy pre vybrané metódy vytváraného programu, ktoré napomáhajú pochopeniu funkcie programu, zobrazujú účinky aspektov a tiež umožňujú pohodlnú navigáciu v zdrojových súboroch programu.

Požiadavky

Aplikácia AJVE je vytvorená pre operačný systém Windows (Windows NT). Vyžaduje minimálne verziu Windows 95 alebo Windows NT 4.0 tohto operačného systému. Pre kompilovanie a spúšťanie vytváraných AspectJ programov požaduje nainštalovanie programu Java 2 SDK (Software Development Kit) verzie 1.3 alebo vyššej a AspectJ Compiler and Core Tools minimálne verzie 1.0. Nainštalovaná aplikácia AJVE zaberá okolo 5MB diskového priestoru.

Minimálna konfigurácia, pre ktorú bola aplikácia testovaná je nasledovná:

- notebook 486
- 8MB RAM
- Windows 95
- Java 2 SDK 1.3
- AspectJ Compiler and Core Tools 1.0

Kvôli dostatočne rýchlej kompilácii a spúšťaniu vytvorených projektov sa treba riadiť požiadavkami danými programom Java 2 SDK 1.3, ktoré doporučujú procesor aspoň Pentium 166MHz s minimálne 32MB RAM pamäte.

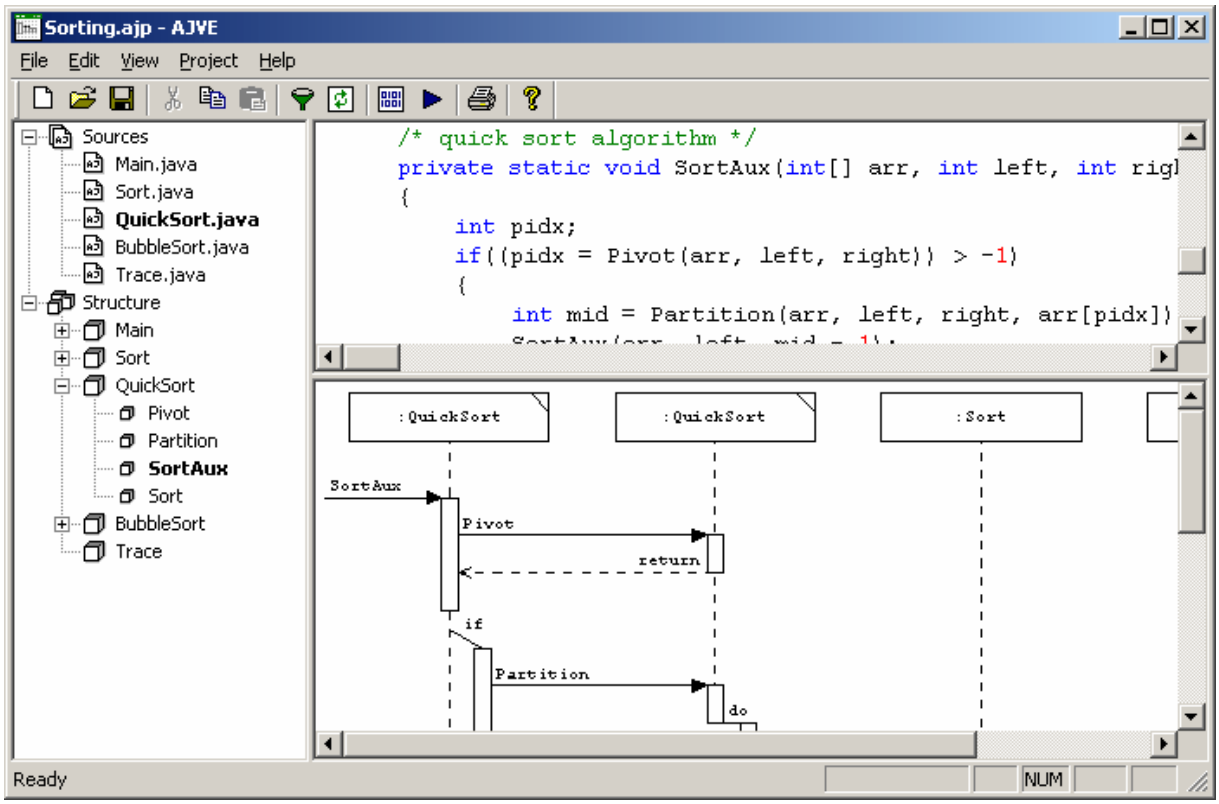
Inštalácia

Aplikácia AJVE sa inštaluje prostredníctvom programu `Setup.exe`, ktorý sa nachádza v hlavnom adresáre priloženého CD. Pre nainštalovanie aplikácie treba v úvodnej obrazovke inštalácie akceptovať licenčné podmienky a potom vybrať spôsob inštalácie. Pre väčšinu používateľov bude vyhovovať typická inštalácia (`Typical`), ktorá je zároveň doporučeným typom inštalácie.

Po úspešnej inštalácii sa vytvorí v ponuke tlačítka `Štart` operačného systému Windows záložka `AspectJ Visualisation Environment`. V tejto záložke sa nachádza položka umožňujúca spustenie aplikácie (`AspectJ Visualisation Environment`). Okrem toho záložka aplikácie obsahuje podzáložku `Examples`, ktorá umožňuje otvorenie vzorových projektov vytvorených v prostredí AJVE.

Používateľské rozhranie

Používateľské rozhranie prostredia AJVE je znázornené na obrázku 1. Prostredie umožňuje naraz otvoriť len jeden projekt, reprezentujúci aplikáciu vytváranú v jazyku AspectJ. Tomu zodpovedá aj zobrazené používateľské rozhranie, ktoré je tvorené jedným hlavným oknom rozdeleným na niekoľko častí. V hornej časti tohto okna sa nachádza editor zdrojového súboru projektu, v spodnej časti je zobrazený sekvenčný diagram vybranej metódy a v ľavej časti je zobrazený navigačný panel, umožňujúci prepnutie editovaného zdrojového súboru a výber metódy, pre ktorú má byť zobrazený sekvenčný diagram. Príkazy slúžiace na prácu s prostredím sa nachádzajú v hlavnom menu prostredia (vrchná lišta okna), pričom niektoré z nich sú prístupné aj prostredníctvom panelu nástrojov, ktorý je tvorený ikonkami týchto príkazov. Dodatočné informácie pre prácu s prostredím sa zobrazujú v stavovom panely (dolná lišta okna).

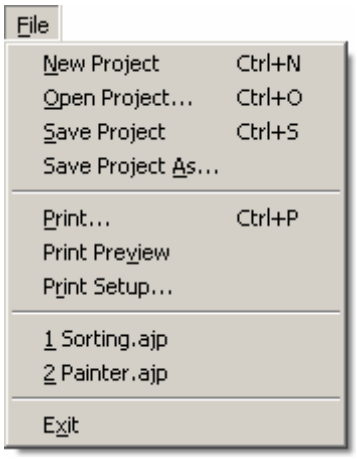


Obrázok 1: Používateľské rozhranie prostredia AJVE

Príkazy prostredia

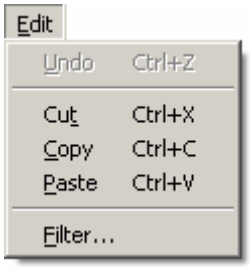
Hlavné menu aplikácie je rozdelené do niekoľkých častí. Každú časť tvorí jedno vnorené menu, ktoré zoskupuje podobné príkazy slúžiace na prácu s prostredím. Toto zoskupenie bude zachované aj pri nasledujúcom popise príkazov prostredia. Popis pre každú skupinu príkazov bude pozostávať z obrázka daného vnoreného menu a tabuľky popisujúcej samotné príkazy. Pri príkazoch prístupných aj prostredníctvom navigačného panelu bude okrem názvu uvedená aj ikonka, ktorou je daný príkaz zastúpený v tomto panely.

Menu File



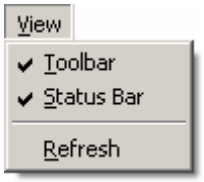
Príkaz	**Popis**
New Project	Vytvorí nový projekt.
Open Project...	Zobrazí dialóg `Open`, ktorý umožní otvorenie uloženého projektu.
Save Project	Uloží projekt na disk.
Save Project As...	Zobrazí dialóg `SaveAs`, ktorý slúži na uloženie otvoreného projektu pod novým názvom.
Print...	Zobrazí dialóg `Print` slúžiaci na vytlačenie aktuálneho zdrojového súboru alebo aktuálneho diagramu.
Print Preview	Vytvorí a zobrazí náhľad tlače.
Print Setup...	Zobrazí dialóg `Print Setup`, prostredníctvom ktorého sa dajú nastaviť parametre tlače.
Recent File List	Zobrazuje posledne otvárané projekty a umožňuje rýchly prístup k týmto projektom.
Exit	Ukončí aplikáciu.

Menu Edit



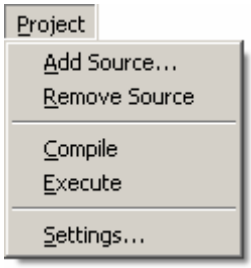
Príkaz	Popis
Undo	Vráti späť poslednú akciu. V tejto verzii prostredia nie je tento príkaz implementovaný.
Cut	Vystrihne v editore zdrojového súboru označený text.
Copy	Skopíruje v editore zdrojového súboru označený text, alebo skopíruje aktuálny sekvenčný diagram.
Paste	Prilepí skopírovaný alebo vystrihnutý text v editore zdrojového súboru na miesto, na ktorom sa nachádza kurzor.
Filter...	Zobrazí dialóg `Filter Editor`, ktorý slúži na editovanie filtra pre zobrazenie sekvenčného diagramu.

Menu View



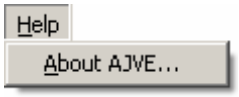
Príkaz	Popis
Toolbar	Zobrazí/schová panel nástrojov.
Status Bar	Zobrazí/schová stavový panel.
Refresh	Obnoví zobrazený diagram, aby zachytával posledné zmeny vykonané v zdrojových súboroch projektu.

Menu Project



Príkaz	Popis
Add Source...	Zobrazí dialóg `Add Source` umožňujúci pridanie nového alebo existujúceho zdrojového súboru do projektu. Vyžaduje, aby bol projekt od svojho vytvorenia aspoň raz uložený na disk.
Remove Source	Vylúči označený zdrojový súbor z projektu.
Compile	Skompiluje projekt. Dá sa použiť len v prípade, že bol prostrediu nastavený kompilátor jazyka AspectJ (dialóg `Settings`) a ak projekt obsahuje aspoň jeden zdrojový súbor.
Execute	Spustí projekt. Dá sa použiť len v prípade, že bol prostrediu nastavený virtuálny stroj jazyka Java a ak bol projektu nastavený názov hlavnej triedy (dialóg `Settings`).
Settings...	Zobrazí dialóg `Settings` umožňujúci nastavenie parametrov.

Menu Help



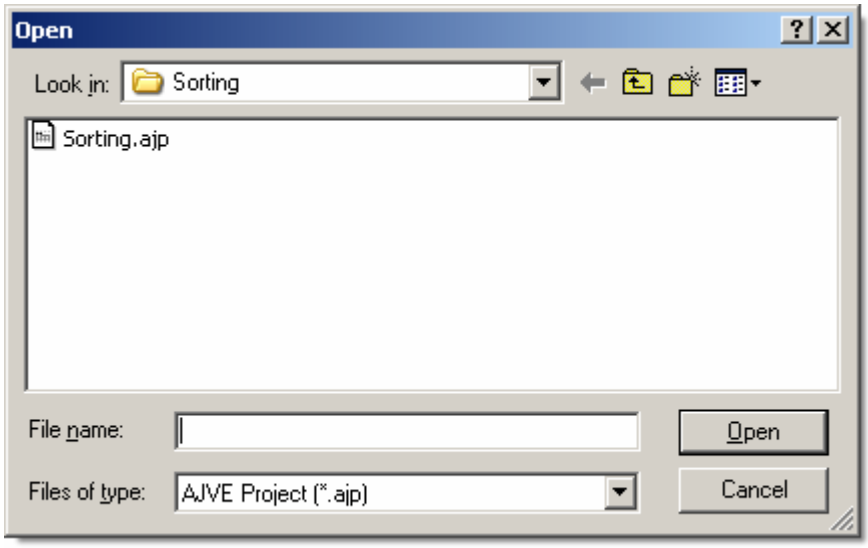
Príkaz	Popis
About AJVE...	Zobrazí dialóg `About AJVE` zobrazujúci informácie o prostredí.

Dialógy prostredia

Spustenie niektorých príkazov prostredníctvom menu alebo panelu nástrojov spôsobí zobrazenie dialógu, ktorý umožňuje bližšie špecifikovať požadovanú akciu. V tejto časti bude uvedený popis zobrazovaných dialógov prostredia spolu s popisom toho, čo umožňujú vykonať po potvrdení. Na stornovanie akcie slúži v zobrazovaných dialógoch tlačítko `Cancel`.

Dialóg Open

Dialóg `Open` (obrázok 2) slúži na výber projektu, ktorý má byť otvorený v prostredí. Otvára súbory s príponou `ajp`, ktoré obsahujú informácie o projekte. Dialóg umožňuje buď napísanie názvu otváraného súboru (`File name`), alebo jeho výber zo zobrazenej ponuky, ktorá sa dá meniť určením adresára (`Look in`).

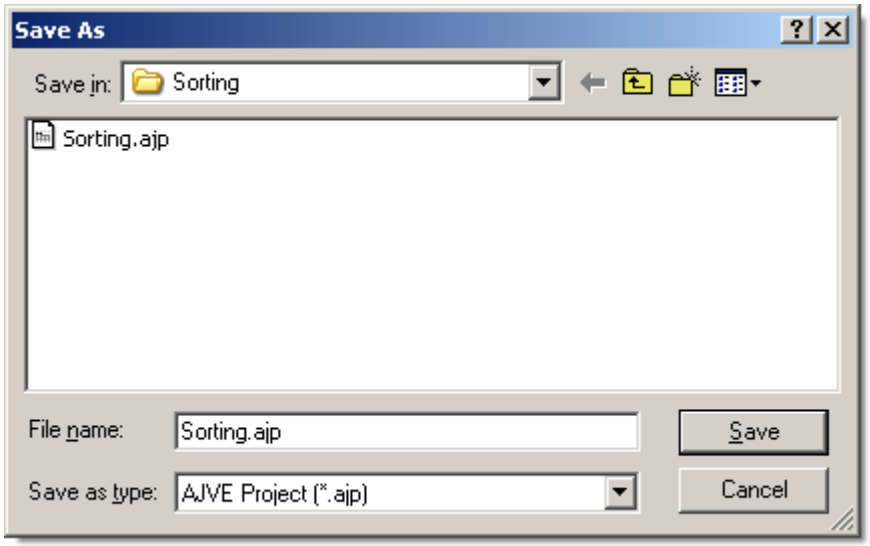


Obrázok 2: Dialóg Open

Dialóg Save As

Dialóg `Save As` (obrázok 3) slúži na uloženie projektu pod novým názvom. Okrem súboru s informáciami o projekte (prípona `ajp`) sa ukladajú aj jednotlivé zdrojové súbory patriace do projektu (prípona `java`). Dialóg umožňuje buď uvedenie názvu (`File name`), pod ktorým

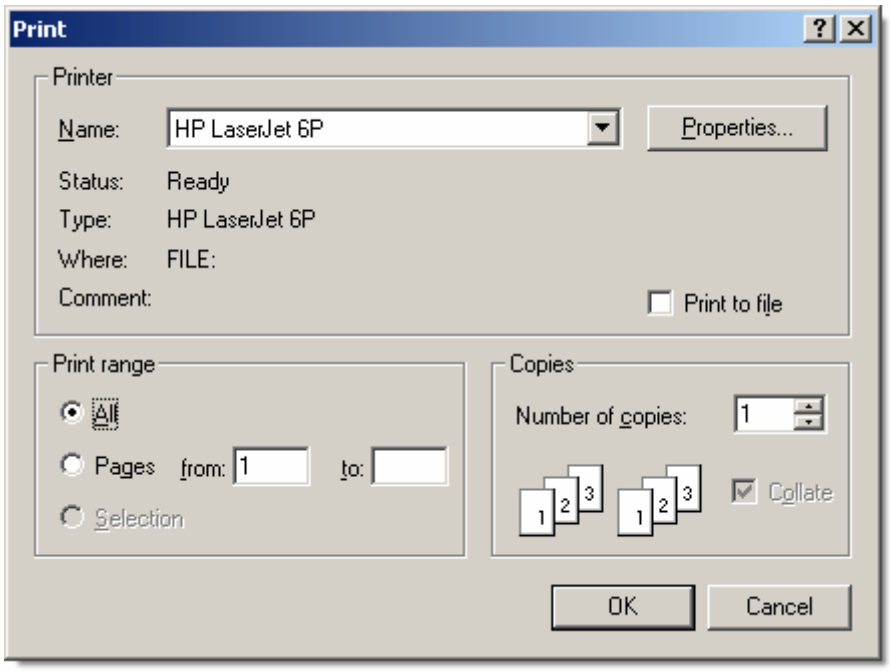
má byť projekt uložený, alebo výber existujúceho projektu z ponuky (dá sa meniť určením adresára - `Look In`), pričom vybraný projekt bude prepísaný ukladaným projektom. Doporučuje sa uchovávať projekty v samostatných adresároch disku (aby nedošlo ku konfliktu mien zdrojových súborov viacerých projektov).



Obrázok 3: Dialóg Save As

Dialóg Print

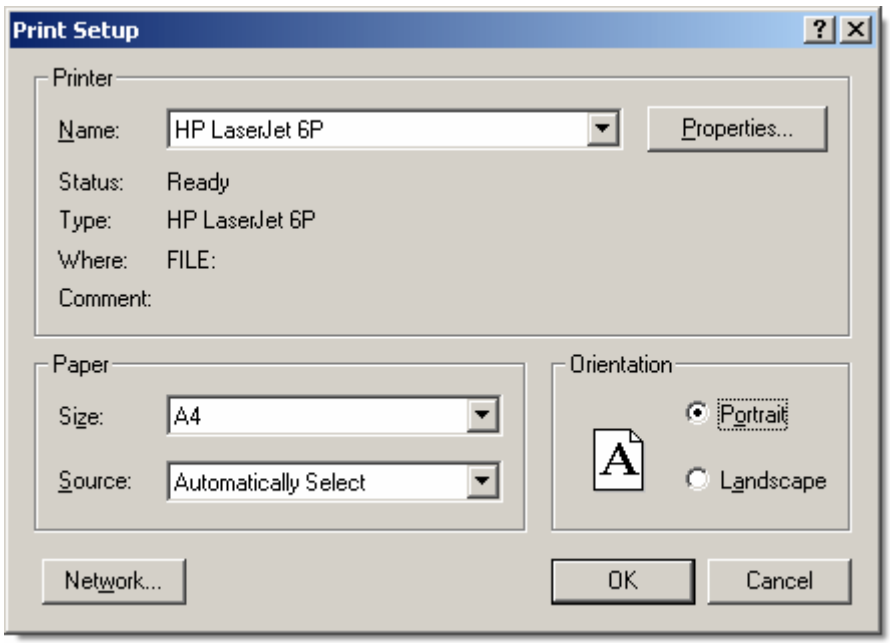
Dialóg `Print` (obrázok 4) umožňuje spustenie tlače. Podľa toho, či je v hlavnom okne prostredia aktívny editor zdrojového súboru alebo editor sekvenčného diagramu sa spustí buď tlač zobrazeného zdrojového súboru alebo zobrazeného sekvenčného diagramu. Dialóg umožňuje výber tlačiarne (`Name`), rozsahu tlače (`Print range`), počtu (`Number of copies`) a usporiadania kópií (`Collate`). Okrem toho sa dajú nastaviť aj dodatočné parametre tlačiarne (`Properties`).



Obrázok 4: Dialóg Print

Dialóg Print Setup

Dialóg `Print Setup` (obrázok 5) slúži na nastavenie tlače. Súčasťou týchto nastavení je meno tlačiarne (`Name`), veľkosť papiera (`Size`), zdroj papiera (`Source`) a jeho orientácia (`Orientation`). Okrem toho sa dajú nastaviť aj ďalšie parametre tlačiarne (`Properties`).

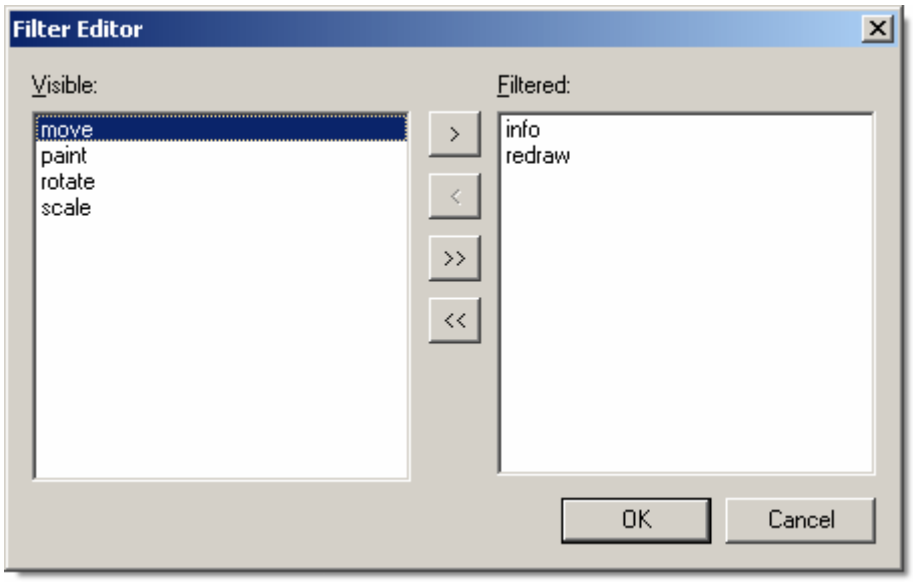


Obrázok 5: Dialóg Print Setup

Dialóg Filter Editor

Dialóg `Filter Editor` (obrázok 6) umožňuje nastavenie filtrovania zobrazeného sekvenčného diagramu. Filtrovanie je nastavené určením množiny zobrazených (`Visible`) a filtrovaných (`Filtered`) značiek. Značky určuje programátor pri písaní metód v špeciálnom komentáre. Tento komentár je umiestnený pred deklaráciu vzťažnej metódy. Po špecifikovaní množiny filtrovaných značiek nebudú na sekvenčnom diagrame zobrazené toky riadenia tých metód, ktoré boli v programe označené niektorou značkou z tejto množiny.
Príklad triedy so špecifikovanými značkami:

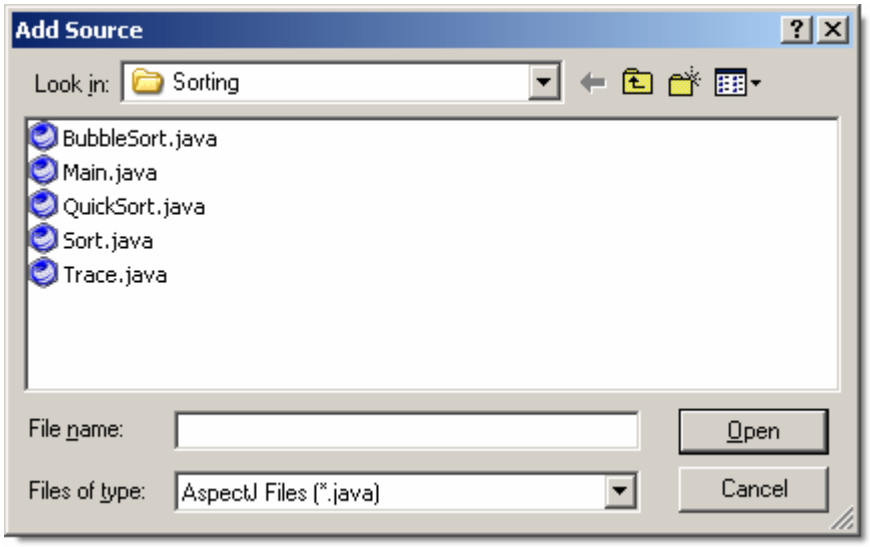
```
public class Line
{
      /*! @info !*/
      Point GetOrigin();              { ... };
      /*! @rotate, @redraw !*/
      public void Rotate(int angle) { ... };
      /*! @scale, @redraw !*/
      public void Scale(int ratio)  { ... };
      /*! @move, @redraw !*/
      public void Move(int relx, int rely)
                                      { ... };
};
```



Obrázok 6: Dialóg Filter Editor

Dialóg Add Source

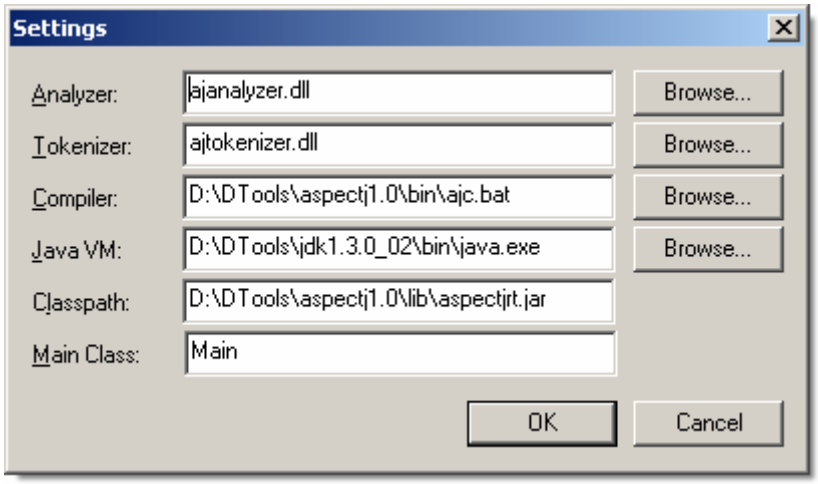
Dialóg `Add Source` (obrázok 7) slúži na pridanie zdrojového súboru do projektu. Umožňuje jednak vytvoriť nový súbor uvedením jeho mena (`File name`), alebo pridanie existujúceho zdrojového súboru do projektu. Existujúci zdrojový súbor môžeme tiež určiť pomocou mena alebo výberom z ponuky, ktorá sa dá meniť špecifikovaním adresára (`Look in`). Súbor pridávaný do prostredia (vytváraný alebo existujúci) musí byť uložený v adresáre, v ktorom je uložený samotný projekt alebo v podadresároch tohto adresára.



Obrázok 7: Dialóg Add Source

Dialóg Settings

Dialóg `Settings` (obrázok 8) slúži na zmenu nastavenia projektu a prostredia.



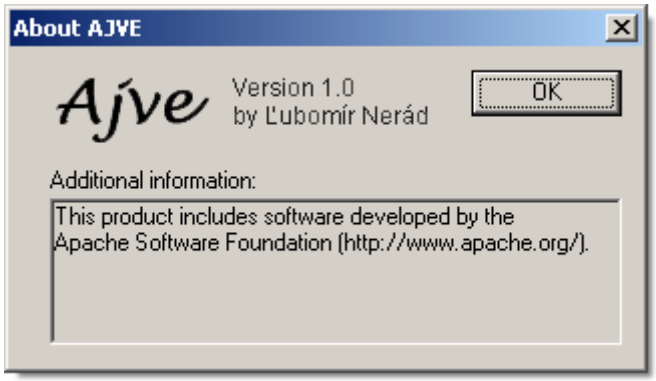
Obrázok 8: Dialóg Settings

Umožňuje zmeniť hodnoty nasledujúcich atribútov:

Atribút	**Popis**
Analyzer	Statický analyzátor jazyka použitý pri vytvorení sekvenčného diagramu. Doporučuje sa ponechať nastavenú hodnotu (`ajanalyzer.dll`).
Tokenizer	Lexikálny analyzátor jazyka použitý na zvýrazňovanie syntaxe jazyka v editore zdrojového súboru. Doporučuje sa ponechať nastavenú hodnotu (`ajtokenizer.dll`).
Compiler	Kompilátor jazyka použitý na kompilovanie projektov. Treba nastaviť na súbor `ajc.bat` (súčasť aplikácie AspectJ Compiler and Core Tools), ktorý slúži na kompilovanie programov v jazyku AspectJ.
Java VM	Virtuálny stroj jazyka Java použitý na spúšťanie projektu. Treba nastaviť na súbor `java.exe` (súčasť aplikácie Java 2 SDK), ktorý spúšťa skompilované triedy jazyka Java.
Classpath	Cesta ku knižniciam vytvoreným v jazyku Java alebo AspectJ. Používa sa pri kompilácii a spúšťaní projektu. Minimálne musí obsahovať knižnicu `aspectjrt.jar` (súčasť aplikácie AspectJ Compiler and Core Tools).
Main Class	Názov hlavnej triedy projektu. Táto trieda musí definovať statickú metódu s názvom `main`, ktorá má vstupný parameter typu `String[]` a nevracia žiadnu návratovú hodnotu.

Dialóg About AJVE

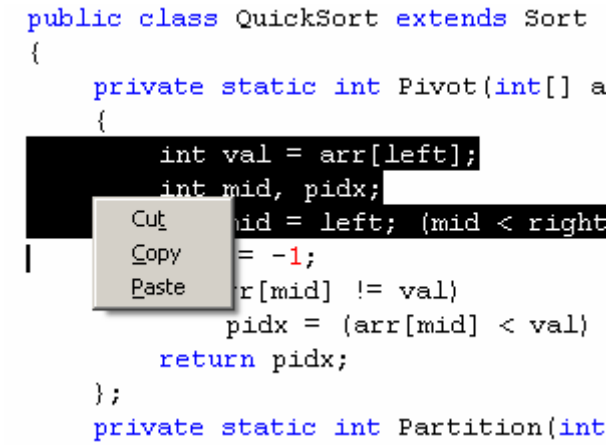
Dialóg `About AJVE` (obrázok 9) poskytuje stručné informácie o prostredí AJVE.



Obrázok 9: Dialóg About AJVE

Editor zdrojového súboru

Editor zdrojového súboru slúži na editovanie aktuálneho zdrojového súboru projektu. Text zdrojového súboru je zobrazený so zvýraznenou syntaxou jazyka AspectJ. Okrem základnej editácie (vkladanie textu, prepisovanie textu, vymazanie textu) presúvanie kurzora) podporuje editor aj označovanie blokov textu, ich kopírovanie, presúvanie a mazanie.



Obrázok 10: Editor zdrojového súboru so znázorneným kontextovým menu

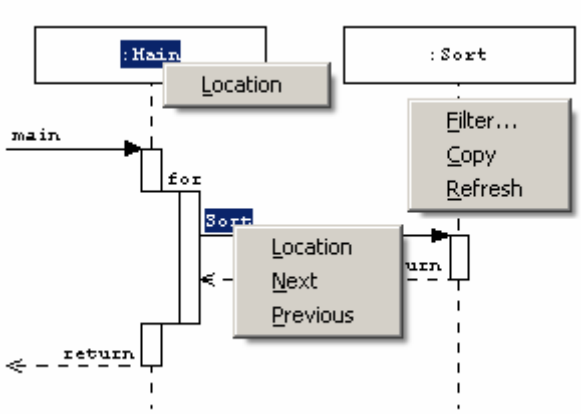
Na prácu s blokmi slúžia príkazy kontextového menu (obrázok 10), ktoré sa zobrazí pri kliknutí pravého tlačítka myši v editore. Význam týchto príkazov uvádza nasledujúca tabuľka:

Príkaz	Popis
Cut	Vystrihne označený text.
Copy	Skopíruje označený text.
Paste	Prilepí skopírovaný alebo vystrihnutý text.

Tieto príkazy sú prístupné aj cez hlavné menu a navigačný panel prostredia.

Editor sekvenčného diagramu

Editor sekvenčného diagramu zobrazuje sekvenčný diagram vytvorený pre vybranú metódu programu. Okrem navigácie v zdrojových súboroch prostredníctvom zobrazeného diagramu umožňuje aj špecifikáciu cieľovej triedy zobrazeného volania.



Obrázok 11: Editor sekvenčného diagramu so zobrazenými rôznymi druhmi kontextového menu

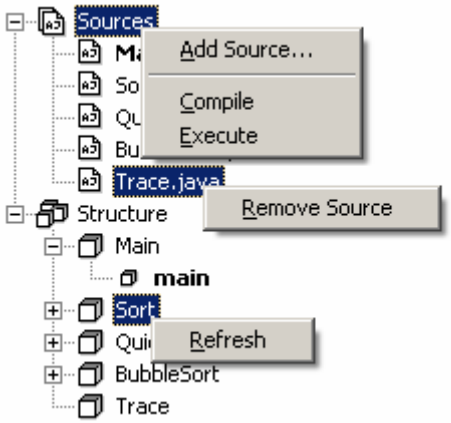
Spomínané funkcie editora zabezpečujú príkazy prístupné cez rôzne druhy kontextového menu (obrázok 11) vytvárané pre jednotlivé elementy diagramu. Kontextové menu elementu diagramu sa vyvoláva pravým tlačítkom myši. Okrem toho diagram zobrazuje kontextové menu celého diagramu pri kliknutí pravého tlačítka myši mimo elementov diagramu. Príkazy rôznych druhov kontextového menu editora sekvenčného diagramu uvádza nasledujúca tabuľka:

Príkaz	**Popis**
Location	Zobrazí umiestnenie elementu diagramu v zdrojovom súbore. Rovnaký účinok má aj kliknutie ľavého tlačítka myši na danom elemente.
Next	Príkaz je zobrazený len pre volanie a umožňuje zobraziť ďalšiu možnú cieľovú triedu daného volania.
Previous	Príkaz je zobrazený len pre volanie a umožňuje zobraziť predchádzajúcu možnú cieľovú triedu daného volania.
Filter...	Zobrazí dialóg Filter Editor, ktorý slúži na editovanie filtra pre zobrazenie sekvenčného diagramu.
Copy	Skopíruje diagram.
Refresh	Obnoví zobrazený diagram, aby zachytával posledné zmeny vykonané v zdrojových súboroch projektu.

Niektoré z uvedených príkazov sú prístupné aj cez hlavné menu a navigačný panel prostredia.

Navigačný panel

Navigačný panel slúži na výber zobrazeného zdrojového súboru v editore zdrojového súboru a na výber metódy, pre ktorú má byť vytvorený sekvenčný diagram v editore sekvenčného diagramu. Na tento účel panel zobrazuje (obrázok 12) zoznam zdrojových súborov projektu a tiež zoznam všetkých metód projektu (zoskupených podľa tried). Výber je uskutočnený dvojitým kliknutím ľavého tlačítka myši na požadovaný zdrojový súbor alebo metódu.



Obrázok 12: Navigačný panel so zobrazenými rôznymi druhmi kontextového menu

Okrem toho navigačný panel podporuje zobrazenie kontextového menu pre jednotlivé zobrazené elementy (obrázok 12). Kontextové menu elementu sa zobrazí kliknutím pravého tlačítka myši na daný element. Jednotlivé príkazy rôznych druhov kontextového menu panelu uvádza nasledujúca tabuľka:

Príkaz	Popis
Add Source...	Zobrazí dialóg Add Source umožňujúci pridanie nového alebo existujúceho zdrojového súboru do projektu. Vyžaduje, aby bol projekt od svojho vytvorenia aspoň raz uložený na disk.
Remove Source	Vylúči daný zdrojový súbor z projektu.
Compile	Skompiluje projekt. Dá sa použiť len v prípade, že bol prostrediu nastavený kompilátor jazyka AspectJ (dialóg Settings) a ak projekt obsahuje aspoň jeden zdrojový súbor.
Execute	Spustí projekt. Dá sa použiť len v prípade, že bol prostrediu nastavený virtuálny stroj jazyka Java (dialóg Settings) a ak bol projektu nastavený názov hlavnej triedy (dialóg Settings).
Refresh	Obnoví zobrazený diagram, aby zachytával posledné zmeny vykonané v zdrojových súboroch projektu.

Uvedené príkazy sú prístupné aj cezhlavné menu prostredia a niektoré z nich aj prostredníctvom navigačného panelu.

Príloha C: Obsah elektronického nosiča

Popis najdôležitejších adresárov a súborov, ktoré sa nachádzajú na priloženom elektronickom nosiči udáva tabuľka C.1.

Adresár/Súbor	**Popis**
`citajma.txt`	popis obsahu elektronického nosiča
`annotation.txt`	anotácia v anglickom jazyku
`anotacia.txt`	anotácia v slovenskom jazyku
`Dokumentacia\`	adresár s vytvorenou dokumentáciou
`Dokumentacia\Dokumentacia.doc`	dokumentácia vo formáte MS Word 97
`Implementacia\`	adresár so zdrojovými súbormi prototypu
`Testovanie\`	testovacie projekty pre prototyp prostredia
`Testovanie\Sorting`	adresár testovacieho projektu `Sorting`
`Testovanie\Sorting\Sorting.ajp`	testovací projekt `Sorting`
`Testovanie\Painter`	adresár testovacieho projektu `Painter`
`Testovanie\Painter\Painter.ajp`	testovací projekt `Painter`
`Testovanie\Simple`	adresár testovacieho projektu `Simple`
`Testovanie\Simple\Simple.ajp`	testovací projekt `Simple`
`Podpora\`	adresár s inštalačnými súbormi aplikácií, ktoré používa prototyp pri kompilovaní a spúšťaní projektu
`MSI\`	pomocný adresár inštalačného programu
`Bronnikov\`	adresár s pôvodnou gramatikou jazyka Java pre programy Lex a Yacc
`Bronnikov\java.y`	gramatika lexikálneho analyzátora jazyka Java pre program Lex
`Bronnikov\java.l`	gramatika syntaktického analyzátora jazyka Java pre program Yacc
`AJVE&Console\`	adresár verzie prototypu, ktorá podporuje odlaďovanie prostredníctvom odlaďovacej aplikácie
`AJVE&Console\AJVE.exe`	prototyp s podporou odlaďovania
`AJVE&Console\Console.exe`	odlaďovacia aplikácia
`AJVE\`	adresár verzie prototypu nepodporujúcej odlaďovanie
`AJVE\AJVE.exe`	prototyp bez podpory odlaďovania
`Setup.exe`	inštalačný program prototypu (nainštaluje verziu prototypu podporujúcu odlaďovanie, odlaďovaciu aplikáciu a testovacie projekty)
`AJVE.msi`	údaje inštalačného programu

Tabuľka C.1: Výpis obsahu elektronického nosiča

Príloha D: Elektronický nosič (CD-ROM)