Slovenská technická univerzita v Bratislave

FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ

Študijný program: SOFTVÉROVÉ INŽINIERSTVO

Bc. Tomáš Vanderka

Prekonanie rozdielov medzi aspektovo orientovanými jazykmi pomocou prístupu MDA

Diplomová práca

Vedúci diplomovej práce: Ing. Valentino Vranić PhD.

Máj, 2007

Čestné prehlásenie:

Čestne prehlasujem, že som túto diplomovú prácu vypracoval samostatne s použitím uvedenej literatúry a zdrojov.

Máj, 2007

ANOTÁCIA

Slovenská technická univerzita v Bratislave
FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ
Študijný program: SOFTVÉROVÉ INŽINIERSTVO

Autor: Tomáš Vanderka
Diplomový projekt: Prekonanie rozdielov medzi aspektovo orientovanými jazykmi pomocou prístupu MDA
Vedenie diplomového projektu: Ing. Valentino Vranić PhD.
Máj, 2007

Táto práca sa zaoberá využitím Model Driven Architecture pri návrhu aspektovo orientovaných programov. Porovnáva vlastnosti a možnosti niekoľkých aspektovo orientovaných jazykov. Analyzuje súčasné možnosti ich reprezentácie modelmi v jazyku UML, alebo inými prostriedkami. Následne sa zaoberá možnými transformáciami týchto modelov. Dotýka sa niektorých problémov spojených s reprezentáciou aspektovo orientovaných programov v modeloch použiteľných v MDA. Výsledkom práce je návrh modelov niekoľkých konkrétnych jazykov, a tiež jedného abstraktnejšieho všeobecného modelu. Modely sú definované ako štandardné rozšírenia jazyka UML a teda je možné ich použitie v akomkoľvek momentálne dostupnom nástroji na modelovanie v jazyku UML. Práca obsahuje neformálne návrhy transformácií medzi týmto všeobecným modelom a modelmi konkrétnych jazykov. Táto práca môže slúžiť ako prehľad možností a problémov pri implementácii komplexnejších modelov, transformácií alebo transformačných nástrojov pre aspektovo orientovaný vývoj.

ANNOTATION

Slovak University of Technology Bratislava
FACULTY OF INFORMATICS AND INFORMATION TECHNOLOGIES
Degree Course: SOFTWARE ENGINEERING

Author: Tomáš Vanderka
Thesis: Overcomming the differences between aspect oriented languages with MDA
Supervisor: Ing. Valentino Vranić PhD.
2007, May

This thesis deals with incorporating Model Driven Architecture into design of aspect oriented programs. It analyzes properties and features of a few aspect oriented languages. It presents current methods of representing these languages in UML or other models. Further it deals with transformations between those models. It touches some issues with representing aspect oriented programs in models usable in MDA. Result of this thesis is a proposal of models for few aspect oriented languages, and also one more abstract general model. Models are defined as standard UML extensions so there is great support in any current modeling tools. It contains informal definition of transformations from the general model to models of concrete languages. This thesis can serve as an overview of possibilities and problems encountered while developing more complex specification of models and transformations or transformation tools for aspect oriented development.

Zadanie:

Aspektovo orientované programovacie jazyky sú založené na spoločnom princípe modularizácie pretínajúcich záležitostí, ale medzi nimi sú značné rozdiely. Aspektovo orientovaný návrh softvéru by mal byť v čo najväčšej miere nezávislý od špecifického aspektovo orientovaného jazyka. Transformáciu návrhu do implementácie v špecifickom jazyku by malo byť možné automatizovať a preto je vhodné použiť prístup Model Driven Architecture (MDA).

Identifikujte spoločné a rozdielne vlastnosti vybraných aspektovo orientovaných jazykov. Analyzujte možnosti na reprezentáciu modelov týchto jazykov ako platforiem v zmysle prístupu MDA. Navrhnite zobrazenia potrebné na postupnú transformáciu modelu nezávislého od platformy do modelu špecifického pre vybraný aspektovo orientovaný jazyk.

Literatúra: J. Brichau and M. Haupt (editors). Survey of Aspect-oriented Languages and Execution Models. AOSD-Europe-VUB-01. http://www.aosd-europe.net/ S. J. Mellor, K. Scott, A. Uhl, and D. Weise. MDA Distilled: Principles of Model-Driven Architecture. Addison Wesley, 2004. OMG. OMG Model Driven Architecture. http://www.omg.org/mda/

OBSAH:

Úvod 1

1 Pojmy 2

2 Analýza jazykov 4

2.1 Vznik aspektovo orientovaného programovania 4

2.2 Princípy AOP 4

2.3 Vetvy AOP 7

2.3.1 Asymetrický prístup – AspectJ (PARC AOP) 8

2.3.2 Symetrický prístup – SOP / MDSoC (Hyper/J) 10

2.3.3 Adaptívne programovanie (DemeterJ) – Prechádzanie stromom 12

2.3.4 Kompozičné filtre – ComposeJ, Compose*, ConcernJ 13

2.4 Popis vybraných jazykov 14

2.4.1 AspectJ 14

2.4.2 CaesarJ 17

2.4.3 AspectC++ 21

2.4.4 JBOSS AOP 23

2.4.5 Hyper/J 24

3 Model Driven Architecture (MDA) 25

4 Prvky AOP jazykov 27

5 Možnosti modelovania v AOSD 29

5.1 Aktuálny stav v oblasti aspektovo orientovaného modelovania 31

5.2 UML profily 33

5.3 MOF metamodely 34

5.4 Hybridný prístup, Theme/UML 35

5.5 AODM 36

5.6 JPDD 37

5.7 Iné prístupy 37

6 Návrh AOM jazyka pre MDA 38

6.1 Špecifikácia požiadaviek na AO model 39

6.2 PSM model pre platformu AspectJ 41

6.3 PSM model pre platformu CaesarJ 43

6.4 PSM model pre Kompozičné filtre 45

6.5 Návrh PIM AO modelu 47

6.6 Návrh transformácií z PIM do PSM 52

6.6.1 Transformované prvky PIM 53

6.6.2 Transformácia do AspectJ PSM 53

6.6.3 Transformácia do CaesarJ PSM 55

6.6.4 Transformácia do CF PSM 56

7 Zhodnotenie 57

8 Literatúra 59

9 Prílohy 63

ZOZNAM OBRÁZKOV:

Obrázok 1 - Pretínajúce záležitosti 5

Obrázok 2 - Pripojenie pretínajúcej záležitosti k operáciám 7

Obrázok 3 - MDSoC a implementácia v jave HyperJ 10

Obrázok 4 - Graf tried 12

Obrázok 5 - Kompozičné filtre [6] 13

Obrázok 6 - MDA transformácia s použitím AOP [15] 26

Obrázok 7 - Konštrukcie v jednotlivých AOP jazykoch (AspectJ, Hyper/J, DemeterJ) 28

Obrázok 8 - Vertikálna a horizontálna dekompozícia 30

Obrázok 9 - Kritériá hodnotenia modelovacích jazykov [17] 32

Obrázok 10 - Príklad UML profilu na modelovanie AspectJ 33

Obrázok 11 - AspectJ metamodel [22] 34

Obrázok 12 - Theme UML aspect [18] 35

Obrázok 13 - AODM Diagramy [21] 36

Obrázok 14 - JPDD diagramy [29] 37

Obrázok 15 - Základný model [26][30] 40

Obrázok 16 - Priama jednoúrovňová transformácia z PIM do PSM 40

Obrázok 17 - Viacúrovňová transformácia 41

Obrázok 18 - Observer v AspectJ 42

Obrázok 19 - Observer v CeaserJ 44

Obrázok 20 - Observer cez Kompozičné filtre 46

Obrázok 21 - Všeobecný AO model 48

Obrázok 22 - Dvojitá medzitypová deklarácia 49

Obrázok 23 - Bodové prierezy 50

Obrázok 24 - Kompletný PIM model na príklade 52

Úvod

Objektovo orientované programovanie je v súčasnosti najrozšírenejšia paradigma programovania. Vyvíjala sa od sedemdesiatych rokov minulého storočia a podporuje zapúzdrenosť, polymorfizmus, modularitu, dedičnosť a znovupoužiteľnosť. Existujú ale problémy, ktoré sa nedajú efektívne vyjadriť ani v tejto paradigme. Spôsobené sú hlavne tým, že objektovo orientované problémy, preberajú svoju hierarchiu objektov, podľa špecifickej domény problému. V prípade že problém zasahuje viaceré oblasti, ktoré sa vzájomne prekrývajú, vzniká viac modulov a zvyšuje sa roztrúsenosť kódu a tým sa znižuje udržiavateľnosť programu. V prípade zmien v jednom module je potom nutné vykonať mnoho zmien v moduloch, ktoré zdanlivo nemajú so zmenou logicky takmer nič spoločné. Na tieto účely sa začali používať nástroje na podporu refaktorizácie programov, ktoré toto uľahčujú a automatizujú veľa potrebných úkonov.

Na riešenie týchto problémov bolo vytvorené aspektovo orientované programovanie. Toto je postavené na objektovo orientovanom prístupe a rozširuje ho o pojem pretínajúcich sa záležitostí. Ide práve o logicky nesúvisiace oblasti ako aplikačná logika, bezpečnosť, logovanie, ladenie ktoré logicky spolu nesúvisia, ale v rámci programu sa vzájomne prelínajú. Aspektovo orientované programovanie poskytuje prostriedky na modularizáciu takýchto pretínajúcich sa záležitostí, čím sa zníži nežiaduce prepletenie kódu.

V poslednom čase sa do popredia dostáva generatívny vývoj programov, keď sa programy špecifikujú pomocou modelov a výsledná implementácia je aspoň čiastočne automaticky vygenerovaná. Môže ísť o generovanie kostry programu, prípadne kompletnej aplikácie podľa databázovej schémy.

Táto práca sa zaoberá možnosťou aplikovať generatívny prístup, konkrétne Model Driven Architecture na aspektovo orientované jazyky, s cieľom preklenúť rozdiely v týchto jazykoch a umožniť tak jednotný vývoj aplikácií nezávisle od použitého konkrétneho aspektovo orientovaného jazyka. Na tento účel sa v práci analyzujú súčasné aspektovo orientované jazyky, ich možnosti, spôsoby ich modelovania a vzájomnej transformácie.

1 Pojmy

- Pretínajúce záležitosti (crosscutting concerns) – sú to funkcie programu zasahujúce do viacerých modulov, ktoré nie je možné modulárne vyjadriť vo vybranej dekompozícii. Následne sú roztrúsené a previazané s inými záležitosťami a modulmi. Ide napríklad o funkcie logovania, ktoré sa zvyčajne používajú na mnohých miestach v aplikácii.
- Zviazanosť kódu (code tangling) – vyjadruje zmiešanie elementov viacerých záležitostí v jednom module.
- Roztrúsenosť kódu (code scattering) – prítomnosť elementov jednej záležitosti v moduloch zaobaľujúcich iné záležitosti.
- Aspekt (aspect) – je jednotkou modularizácie nejakej pretínajúcej záležitosti.
- Bod spájania (joinpoint) – bod vo vykonávaní programu, v ktorom môže nastať prepojenie pretínajúcich záležitostí, môže byť statický alebo dynamický. Statické sú definovateľné pred vykonaním programu, dynamické sú závislé od podmienok a stavu, ktorý nastane až počas vykonávania programu.
- Bodový prierez (pointcut) – definícia viacerých bodov spájania. Množina obsahujúca body spájania, s ktorou je možné pracovať akoby sa jednalo o jeden bod spájania.

Skratky

- OOP (Object-Oriented Programming) – Objektovo orientované programovanie.
- AOP (Aspect-Oriented Programming) – Aspektovo orientovane programovanie.
- AOM (Aspect Oriented Modelling) – Aspektovo orientované modelovanie.
- AOSD (Aspect-Oriented Software Development) – aspektovo orientovaný vývoj softvéru.
- UML (Unified Modelling Language) – štandardizovaný špecifikačný jazyk pre objektovo orientované programy.
- OCL (Object Constraint Language) – špecifikačný jazyk určený na definovanie obmedzení v jazyku UML.
- MDA (Model Driven Architecture) – spôsob špecifikácie programu pomocou abstraktných modelov popisujúcich jeho štruktúru a správanie. Konkrétna

implementácia programu je realizovaná pomocou transformácie tohto abstraktného modelu do konkrétneho jazyka cieľovej platformy.

- PIM (Platform Independent Model) – je to model systému, ktorý je platformovo nezávislý. Ako platformu môžeme chápať napríklad konkrétny programovací jazyk alebo operačný systém.
- PSM (Platform Specific Model) – platformovo závislý model systému, jednotlivé prvky modelu sú špecifické pre konkrétnu platformu, napríklad jazyk Java.
- JPDD (Join Point Designation Diagram) – diagram určujúci body spájania pomocou sekvenčných UML diagramov.
- ASoC (Advanced Separation of Concerns) – pokročilé separovanie záležitostí, zastrešuje všetky prístupy zaoberajúce sa oddelením pretínajúcich zámerov v celom procese vývoja softvéru.

2 Analýza jazykov

Takmer každý aspektovo orientovaný jazyk je tvorený ako rozšírenie existujúceho objektovo orientovaného jazyka. Väčšina implementácií je rozšírením Javy. Existujú ale aj prístupy, ktoré nepoužívajú ako základ objektový jazyk (napr. MAKAO).

2.1 Vznik aspektovo orientovaného programovania

Už dávno je známe že rozdelením programu na menšie časti získame prehľadnejší a ľahšie udržiavateľný zdrojový kód. E.W. Dijkstra v jeho práci zaoberajúcej sa modularizáciou použil spojenie „separation of concerns", čo sa dá preložiť ako oddelenie záležitostí. Aplikácia má bežne niekoľko rôznych záujmov a ich oddelenie do samostatných modulov nie je pomocou OOP vždy možné. Rôzne záujmy sa v moduloch prelínajú a miešajú. Oddelenie práve takýchto pretínajúcich sa záležitostí je základom aspektovo orientovaného programovania.

Začiatky AOP položili vo výskumnom centre Xerox PARC, kde vytvorili momentálne stále najpoužívanejší aspektovo orientovaný jazyk AspectJ. Ešte predtým ale na Northeastern University pracovali na paradigme adaptívneho programovania (AP), ktorá sa dá prirovnať k určitej podmnožine AOP, implementáciou AP je napríklad jazyk DemeterJ. Najstarším príbuzným AOP sú ale kompozičné filtre, ktoré sa vyvíjali už od deväťdesiatych rokov minulého storočia.

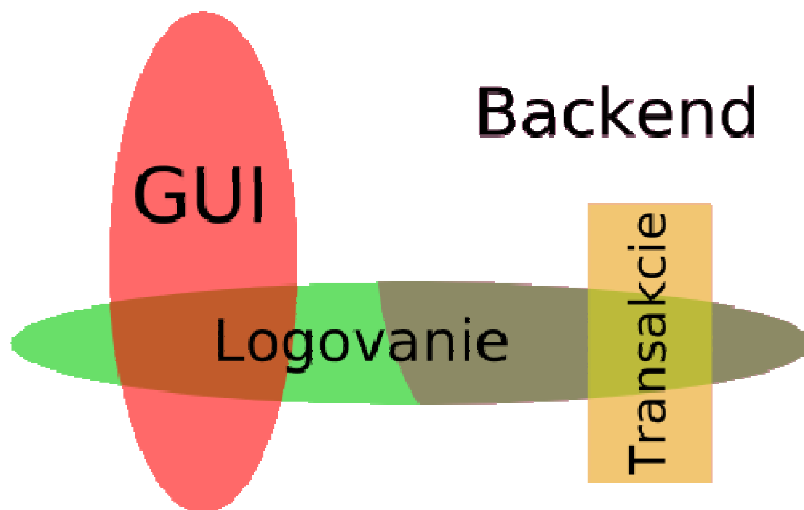
V súčasnosti existuje veľké množstvo jazykov, hlavne tých odvodených od AspectJ. Existujú ale aj iné, ktoré používajú iný spôsob zápisu aspektov alebo aj iný celkový pohľad.

2.2 Princípy AOP

Aspektovo orientované programovanie je paradigma, ktorá sa snaží o oddelenie pretínajúcich záležitostí, alebo rozloženie programov na oddelené časti, ktorých funkcionalita sa prekrýva iba minimálne. AOP sa zameriava na modularizáciu takýchto rozdielnych a oddelených pretínajúcich záležitostí.

AOP možno rozdeliť na dve skupiny a to symetrické a asymetrické prístupy. Pri symetrických neexistuje hlavný modul, ale všetky moduly aplikácie sú si rovné, a vzájomnou kombináciou tvoria výsledný celok. Pri asymetrickom prístupe je jeden modul vždy hlavný bez akýchkoľvek aspektových prvkov a ďalšie moduly špecifikujú ako a čo sa má upraviť v tomto hlavnom module.

Existujú aj iné paradigmy zameriavajúce sa na modularizáciu funkcionality do samostatných entít, ako napríklad procedúr, tried, knižníc. Niektoré záležitosti nie je možné takýmto spôsobom zapuzdriť, pretože sa nachádzajú na mnohých miestach programu, sú príliš rozptýlené. Tieto nazývame „pretínajúce záležitosti“. Najlepším príkladom je systém logovania. Pretože logovanie z princípu zasahuje takmer do každej časti systému, tried a procedúr.



Obrázok 1 - Pretínajúce záležitosti

AOP sa pokúša zapuzdriť a izolovať práve takéto roztrúsené záležitosti pomocou „aspektu“. Aspekt sa dá popísať ako funkcionalita, ktorá sa naviaže na určité miesta zvyšných častí programu. Aspekt určuje čo, kam a ako pripojiť. Aspektovo orientované jazyky sa líšia hlavne spôsobom zápisu týchto troch vecí, a implementáciou spájania aspektov do výsledného programu.

Jednoduchý fragment programu na prevod prostriedkov z jedného účtu na druhý napríklad v Jave demonštruje použitie aspektov:

```
void transfer(Account from, Account to, int amount) {
  if (from.getBalance() < amount) {
    throw new InsufficientFundsException();
  }

  from.withdraw(amount);
  to.deposit(amount);
}
```

V skutočnosti ale musíme v takej aplikácii myslieť aj na ďalšie záležitosti ako napríklad logovanie a bezpečnosť. Aby boli podrobne zaznamenané všetky operácie nad kontom, musí byť zabezpečená transakčná bezpečnosť aby v prípade zlyhania nedošlo k strate prostriedkov zo zdrojového konta bez toho aby sa zapísali na cieľové. Po pridaní potrebných častí to môže vyzerať napríklad takto:

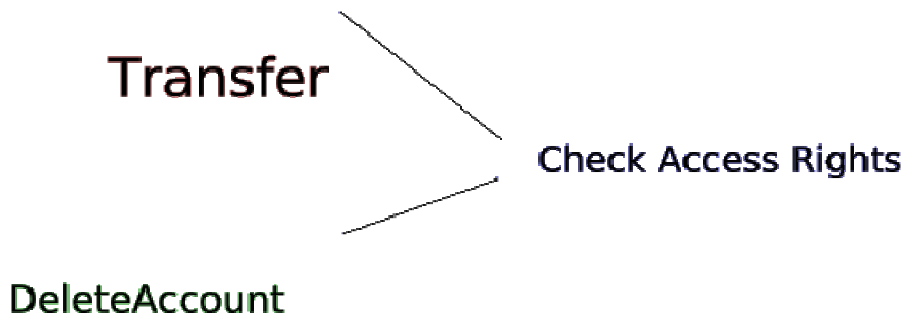
```
void transfer(Account from, Account to, int amount) {
  if (!getCurrentUser().ownsAccount(from)) {
    throw new SecurityException();
  }

  if (from.getBalance() < amount) {
    throw new InsufficientFundsException();
  }

  Transaction tx = database.newTransaction();

  try {
     from.withdraw(amount);
     to.deposit(amount);
     tx.commit();
     systemLog.logTransfer(from, to, amount);
  }
  catch(Exception e) {
     tx.rollback();
  }
}
```

V tomto fragmente sú všetky pretínajúce záležitosti zviazané. Podobne by vyzeralo veľké množstvo metód vo výslednej aplikácii. Ak by sme neskôr chceli zmeniť napríklad zabezpečenie aplikácie museli by sme prepísať podstatnú časť celej aplikácie, čo by si vyžadovalo veľa práce a zároveň by to zvyšovalo pravdepodobnosť vytvorenia chyby. To by bolo v prípade zabezpečenia obzvlášť nepríjemné. V súčasnosti tieto problémy čiastočne pomáhajú riešiť nástroje na refaktorizáciu kódu, ale neodstraňujú príčinu týchto problémov.

Vidno že pretínajúce záležitosti nie sú v programe dostatočne štruktúrované a oddelené. Pomocou AOP tieto pretínajúce záležitosti môžeme oddeliť od hlavnej logiky a vytvoriť tak

nezávislé moduly - aspekty, ktoré budú implementovať iba jeden konkrétnu záležitosť. Napríklad bezpečnostný modul môže kontrolovať prístupové práva pred vykonaním privilegovaných funkcií určitých tried.

Aspekt môže napríklad špecifikovať, že pred operáciou transfer a niekoľkých iných sa má vykonať overenie práv používateľa. Tým oddelíme implementáciu bezpečnosti od zvyšku logiky, a zároveň znížime duplicitu kódu a umožníme jej viacnásobné použitie aj pre iné časti programu.



Obrázok 2 - Pripojenie pretínajúcej záležitosti k operáciám

2.3 Vetvy AOP

Aspektovo orientované programovanie sa vyvíja viacerými smermi, v závislosti od zvoleného spôsobu oddelenia pretínajúcich záležitostí. Každý z týchto smerov má svoje výhody a nevýhody v závislosti od riešeného problému. Štyri hlavné smery sú:

1. PARC AOP (AspectJ) – Ide o asymetrické oddelenie hlavného programu, ktorý obsahuje takmer všetku logiku aplikácie, a rozptýlených pretínajúcich záležitostí zasahujúcich do viacerých modulov programu. Jednotlivé pretínajúce záležitosti sú izolované do aspektov, ktoré sa potom aplikujú na potrebné body spájania – miesta v hlavnom programe.
2. Viacrozmerná separácia záležitostí (MDSoC) – Vychádza zo subjektívneho programovania, ktoré výsledný program spája z viacerých rovnocenných pohľadov na jednu aplikáciu. Jednotlivé pohľady sa môžu prekrývať, nahrádzať svoje súčasti, modifikovať správanie. Výsledkom je program spojený podľa daných pravidiel zo zadaných súčastí.

3. Adaptívne programovanie – Vychádza z objektovej štruktúry a jej hlavným cieľom je oddelenie štruktúry od správania. Týmto sa stávajú menej náchylné na zmeny objektovej štruktúry. Snaží sa to dosiahnuť pomocou prechodov po hranách objektových grafov. Pri prechode nejakou triedou sa vyvoláva definované správanie.
4. Kompozičné filtre – Modifikujú správanie objektov pomocou aplikácie vstupno-výstupných filtrov. Dokážu pridať, zmeniť funkcionalitu na vstupoch aj výstupoch objektov. Výsledný program je zložením viacerých filtrov. Kompozičné filtre dovoľujú modifikovať správanie objektov iba pomocou odchytávania správ, čiže volania metód jednotlivých tried.

2.3.1 Asymetrický prístup – AspectJ (PARC AOP)

Z jazyka AspectJ sa vyvinulo mnoho ďalších, ktoré používajú podobný model. Tieto jazyky stavajú na asymetrickom prístupe, keď sa hlavný program modifikuje použitím aspektov. Toto je hlavná v súčasnosti používaná vetva AOP, pretože je jednoduché pochopiť jej základy, a zároveň sú nutné minimálne zásahy do originálnej aplikácie.

Základným prvkom takéhoto asymetrického jazyka je aspekt, ktorý definuje a zapúzdruje samotnú pretínajúcu záležitosť.

Každý z týchto jazykov používa na špecifikáciu aspektov podobný model, ktorý definuje:

- Body spájania – určujú miesta vo vykonávaní programu, kam sa môže naviazať funkcionalita z aspektu.
- Špecifikáciu bodových prierezov – spôsob ako popísať množinu bodov spájania, v ktorých sa aplikuje zmena definovaná aspektom.
- Správanie, videnie (advice) – samotná špecifikácia čo sa má vykonať v bodových prierezoch.

Aspekty

Aspekt je modul implementujúci pretínajúce záležitosti. Obsahuje špecifikáciu videní a ich priradenie k bodovým prierezom. Niektoré jazyky umožňujú aspektom dedičnosť iné nie. Aspekty bývajú najčastejšie definované ako špeciálne triedy.

Body spájania

Určujú miesta vo vykonávaní programu kam sa môže aplikovať funkčnosť definovaná v nejakom aspekte. Môžu byť statické, alebo dynamické podľa toho či sa určujú z informácií o statickej štruktúre programu, alebo na základe pravidiel vyhodnocovaných za behu programu. Bodmi spájania môžu byť napríklad volania metód, zápis a čítanie atribútov, vytvorenie objektu a podobne. Každý konkrétny jazyk definuje svoju vlastnú množinu základných bodov spájania.

Statické

Statické body spájania sú definovateľné už pred spustením programu. Ide napríklad o priraďovanie alebo čítanie premenných, volanie funkcií s určeným typom návratovej hodnoty a podobne. Tieto body ale musia byť jednoznačne známe zo statickej štruktúry programu.

Dynamické

Dynamické body spájania umožňujú naviazanie funkčnosti aspektu na miesta v programe, určené za behu. Napríklad volanie troch konkrétnych metód rekurzívne za sebou (podľa zoznamu v zásobníku), alebo vyvolanie výnimky. Toto nie je možné určiť pred spustením programu.

Bodové prierezy

Bodové prierezy špecifikujú množinu bodov spájania na ktoré sa má aplikovať videnie (advice). Môžu byť definované špeciálnym jazykom, alebo pomocou funkcií a logických operátorov nad základnými bodmi spájania, poznámkami v zdrojovom texte programu a podobne.

Príklad:

„* *.hello (*)“

This(Account) && execution(“* *.transfer (..)”)

@PointcutDef("MyAspect.pojoFieldReads OR MyAspect.pojoFieldWrites")

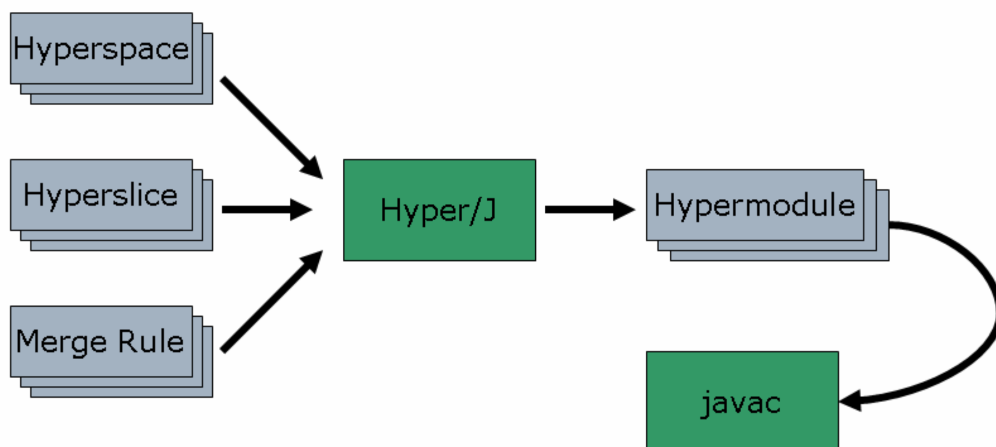
Toto je pravdepodobne časť, v ktorej sa AOP jazyky najviac odlišujú (ak neberieme do úvahy samotný proces spájania a vykonávania programov, ktorý už ale nie je priamo viditeľný pri vývoji aplikácie a vykonávajú ho automaticky prekladače alebo virtuálne stroje).

Videnia (Advice)

Videnia obsahujú kód, ktorý sa má pripojiť na špecifikované miesta dané bodovým prierezom. Môžu sa vykonať pred, po, okolo alebo namiesto vybraného bodového prierezu.

2.3.2 Symetrický prístup – SOP / MDSoC (Hyper/J)

HyperJ je implementáciou viacrozmernej separácie a integrácie záležitostí, vychádza zo subjektovo orientovaného programovania. Základom sú viaceré pohľady cez funkcionalitu na ten istý program. Jednotlivé pohľady sa potom zlúčia, poprekrývajú a vytvoria finálny celok so všetkými pretínajúcimi záležitosťami. Každý z týchto pohľadov sa nazýva hypervýrez (hyperslice). Každá pretínajúca záležitosť by mala byť implementovaná samostatným hypervýrezom pomocou bežného jazyka. Tieto hypervýrezy sa potom kombinujú pomocou pravidiel spájania a vytvoria finálny program, ktorý obsahuje všetky pretínajúce záležitosti. Jedným veľmi dôležitým obmedzením tohto prístupu je že každý kypervýrez musí byť deklaratívne kompletný, to znamená že musí obsahovať deklarácie všetkých premenných, tried, funkcií, ktoré využíva. Jednoducho musí byť sám o sebe syntakticky správny a kompilovateľný aj bez ostatných hypervýrezov. Ďalším obmedzením je že výsledný program je kompletne zostavený ešte pred spustením, takže nie je možné dynamicky meniť prekrývanie a spájanie jednotlivých hypervýrezov, tak ako je to možné v iných asymetricky orientovaných prístupoch, kde je možné aspekty aj za behu aktivovať a deaktivovať.



Obrázok 3 - MDSoC a implementácia v jave HyperJ

Jednotlivé triedy implementujúce dve oddelené záležitosti:

```
public class Foo {
  public void sayHello() {
    System.out.println("Hello");
  }
}

public class Bar {
  public void sayWorld() {
    System.out.println("World");
  }
}
```

Definícia spojiteľných súčastí, ide v podstate o zoznam tried:

```
  hyperspace helloworld
  composable class Foo;
  composable class Bar;
```

Mapovanie balíkov, tried, a metód do dimenzií a záležitostí

```
  class Foo : Feature.hello
  class Bar : Feature.world
```

Pravidlá spájania do výsledného hypermodulu:

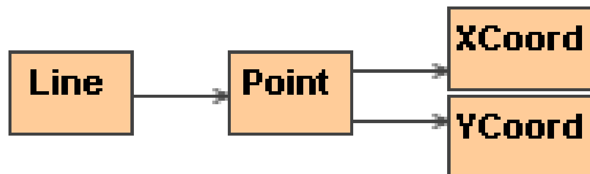
```
  hypermodule Demo
    hyperslices: Feature.hello, Feature.world;
    relationships:
      mergeByName;
      equate operation Feature.hello.sayHello,
                       Feature.world.sayWorld into greet;
    merge class Feature.hello.Foo, Feature.world.Bar;
  end hypermodule;
```

Spojí triedy Foo a Bar do jednej triedy a spojí operácie hello a world do novej nazvanej greet.

2.3.3 Adaptívne programovanie (DemeterJ) – Prechádzanie stromom

Je implementáciou adaptívneho programovania nad jazykom Java. Základom sú grafy tried a prechod po hranách v týchto grafoch pomocou objektov - návštevníkov.

DemeterJ potrebuje definovať súvislosti medzi triedami vo forme grafu. Môže byť definovaný pomocou grafického nástroja alebo jednoduchou gramatikou napríklad:



Obrázok 4 - Graf tried

Na tomto grafe potom definuje prechodovú stratégiu, čo je množina ciest medzi jednotlivými triedami. Napríklad najjednoduchší je prechod jednou hranou { Point -> XCoord }

Umožňuje kombinovať jednoduché stratégie do zložitejších pomocou konštrukcií ako:

- Vynechanie triedy
- Určenie viacero cieľov
- Viacero ciest k cieľu cez inú triedu
- * výber, všetkých dostupných cieľov

Prechod grafom je implementovaný prechodovými funkciami, ktoré pre danú prechodovú stratégiu aplikujú metódy vybraného návštevníka.

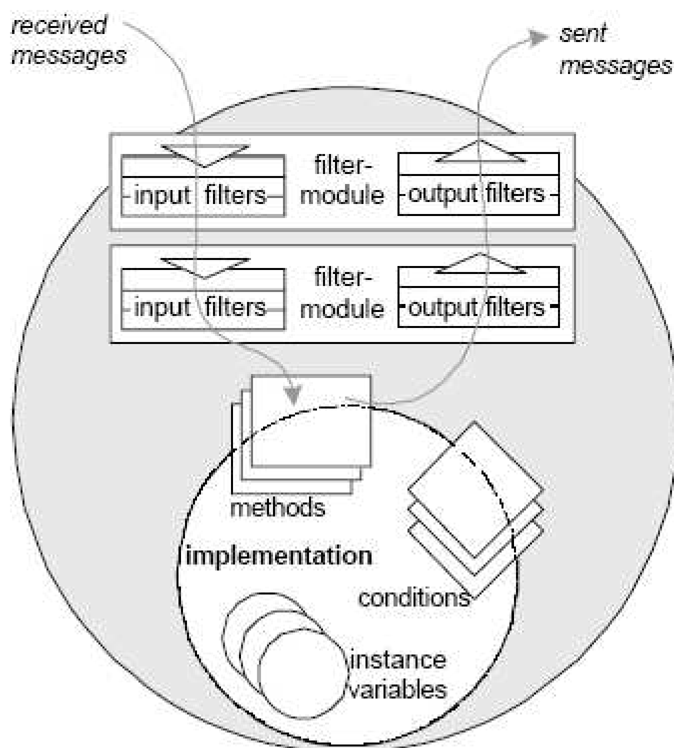
Samotný kód, ktorý sa má vykonať pri prechode konkrétnou triedou je definovaný v triede návštevníka ako metóda before, after alebo around.

Samotná adaptívna metóda sa vykonáva tak, že spustí nejakú funkciu pre každý objekt, na ktorý narazí pri prechode grafom, a ako parameter jej predá aktuálny objekt. Napríklad adaptívna metóda pre nakreslenie čiary môže byť implementovaná ako volanie prechodovej funkcie pre každý bod čiary, s návštevníkom, ktorý každý bod vykreslí.

2.3.4 Kompozičné filtre – ComposeJ, Compose*, ConcernJ

Kompozičné filtre modifikujú správanie objektov na úrovni ich rozhraní, modifikáciou vstupných a výstupných správ. Takže nemodifikujú priamo implementáciu konkrétneho objektu, ale implementujú niečo ako „wrapper“. Na rozdiel od bežných OOP wrapperov je ich nasadenie transparentné vo zvyšku aplikácie, čiže netreba meniť typ objektov, na ktoré sa aplikujú. Ďalší rozdiel je že ak bežný wrapper (ktorý nie je odvodenou triedou) chce použiť metódy definované v originálnej triede musí tieto metódy nanovo definovať, aj keď budú obsahovať iba volanie metódy zaobaleného objektu. Kompozičné filtre toto riešia automaticky a pokiaľ filter nedefinuje nejakú metódu – správu, deleguje ju automaticky do interného objektu.

Filtre sa pridávajú k základným objektom dynamicky za behu modulárnym spôsobom. Samotná štruktúra rozhrania môže byť popísaná platformovo nezávisle, až konkrétna implementácia pracujúca nad týmto rozhraním je závislá na konkrétnom jazyku. Tým dosiahneme oddelenie jazykovo závislej a nezávislej časti.



Obrázok 5 - Kompozičné filtre [6]

Kompozičné filtre využívajú deklaratívnu syntax na opis filtrov. Táto časť je nezávislá od implementácie vnútorných objektov a nových správ, jazykovo závislé definície sú znázornené iba vo vnútornom kruhu na obrázku 5.

2.4 Popis vybraných jazykov

V tejto kapitole sú popísané špecifické vlastnosti vybraných aspektovo orientovaných jazykov, ktorými sa odlišujú od zvyšných jazykov. Niektoré jazyky aj keď sa podobajú obsahujú síce malé ale podstatné rozdiely, ktoré určujú spôsob implementácie programov v týchto jazykoch.

2.4.1 AspectJ

Je jeden z prvých AOP jazykov a prvou implementáciou prístupu vyvinutého v Xerox PARC. Používa asymetrický prístup a dynamický model bodov spájania. Pre aspekty dovoľuje sprístupniť kontext vykonávania ako lokálne premenné videnia pomocou špeciálnych kľúčových slov v definícii bodového prierezu. Okrem klasického modelu s videniami poskytuje aj medzitypové deklarácie na vkladanie nových atribútov alebo metód do existujúcich tried. Do jazyka zavádza nové konštrukcie a kľúčové slová na definíciu aspektov, bodových prierezov a medzitypových deklarácií.

Tak ako je uvedené v popise asymetrických prístupov, AspectJ zavádza do jazyka:

- Aspekty – popisujú a zapúzdrujú jednu pretínajúcu záležitosť
- Body spájania – možné miesta v programe, kde je možné zapojiť videnia
- Bodové prierezy – určujú prepojenie videní na konkrétne body spájania
- Videnia – implementujú funkcionalitu pretínajúcej záležitosti
- Medzitypové deklarácie – upravujú definíciu existujúcich tried

Body spájania (join points)

AspectJ umožňuje použiť okrem statických aj dynamické body spájania, ktoré nepodporujú všetky aspektovo orientované jazyky. V starých verziách bolo možné použiť iba statické body spájania.

- call – volanie metódy alebo konštruktora
- get, set – prístup a zapisovanie do atribútov triedy
- execution – vykonanie metódy alebo konštruktora
- this, target, args – sprístupnenie kontextu videniu
- cflow, cflowbelow – každý bod prekrytý daným bodom spájania.

- staticinitialization, initialization, preinitialization – inicializácia objektov určeného typu
- handler – vykonanie obsluhy výnimky
- adviceexecution – vykonanie videnia

Bodové prierezy (pointcut)

Sú definované v rámci aspektov alebo tried, pomocou logických operácií and, or a negácie nad primitívnymi bodovými prierezmi, ktoré vychádzajú z bodov spájania a pridávajú ďalšie možnosti:

```
pointcut Nazov(): call(* func(..)) && within(Trieda);
```

Ako všeobecný kvantifikátor sa môže použiť „*“ a „..“. Môžeme špecifikovať podľa názvu balíka, triedy, metódy, parametrov, typu návratovej hodnoty a typu vyvolanej výnimky. Môžu byť pomenované alebo anonymné. Umožňujú použiť dedičnosť, takže v zdedenom aspekte môžu byť predefinované.

Videnia (advice)

AspectJ podporuje nasledujúce typy videní

- before – vykoná sa pred bodom spájania
- after returning – vykoná sa po bode spájania, a umožnuje prístup k návratovej hodnote
- after throwing – vykoná sa po bode spájania, ktorý vyvolá výnimku
- after – vykoná sa po bode spájania
- around – vykoná sa namiesto bodu spájania, ktorý môže ale nemusí vykonať pomocou kľúčového slova „proceed“.

Sú definované ako metódy v aspektoch, a priamo v definícii sa špecifikujú bodové prierezy, na ktoré sú naviazané. Videnia sú v AspectJ nepomenované.

```
before ( ) : call(void xy(..)) {
      System.out.println("Volanie funkcie xy." ) ;
}
```

Aspekty (aspect)

Aspekty sú definované podobne ako triedy, s použitím kľúčového slova „aspect“:

```
aspect Tracing {
      before ( ) : call(void xy(..)) {
            System.out.println("Volanie funkcie xy." ) ;
      }
}
```

Umožňujú použiť dedičnosť a implementáciu rozhraní. Dedičnosť môžeme využiť pri definícii bodových prierezov aj implementácii videní. Klasické Java triedy ale nemôžu byť odvodené od aspektov, môžu iba obsahovať videnia.

Inštancie aspektov nie je možné dynamicky vytvárať, sú vytvorené automaticky. Je ale možné špecifikovať rozsah ich vytvárania pomocou modifikátorov:

- perthis – vytvorí sa pre každý objekt špecifikovaný prierezom this()
- pertarget – vytvorí sa pre každý objekt špecifikovaný prierezom target()
- percflow – vytvorí sa pre každý blok kódu určený prierezom cflow()
- percflowbelow – vytvorí sa každý bok kódu určený prierezom cflowbelow()
- pertypewithin – vytvorí sa pre každý typ daný prierezom within()

Ak nie je uvedené nič tak je inštancia aspektu vytvorená ako singleton pre celú aplikáciu.

Medzitypové deklarácie (introduction)

Medzitypové deklarácie pretínajú triedy a umožňujú pridať nové atribúty, metódy alebo implementovať rozhrania pre vybranú množinu tried. Tieto deklarácie sa aplikujú staticky, čiže už počas prekladu programu. Používa sa na to kľúčové slovo „declare“, a zápis uvedený v príklade.

Príklad, pridanie farby do triedy Point:

```
aspect ColorAspect {
    Color Point.color = new Color();
    public void Point.setColor(Color c) {this.color = c};
    ...
}
```

2.4.2 CaesarJ

CaesarJ je nový aspektovo orientovaný programovací jazyk, zameraný na modularitu, viacnásobné použitie, flexibilitu a korektnosť programov. Je založený na objektovom prístupe a jazyku Java. Je podobný jazyku AspectJ, a podporuje väčšinu konštrukcií tohto jazyka. Umožňuje napríklad abstrakciu aspektov a viac oddeľuje implementáciu aspektu od samotného hlavného programu, s ktorým sa aspekt spája. Aspekty sú implementované ako samostatné komponenty a je možné ich jednoducho znovu použiť. Preberá aj isté črty subjektovo orientovaného programovania, pomocou konceptu virtuálnych tried, ktoré je možné jednoducho vzájomne kombinovať a tak dosiahnuť výsledok, ktorý obsahuje vlastnosti jednotlivých častí.

Na oddelenie pretínajúcich záležitostí používa CaesarJ rovnakým spôsobom ako AspectJ bodové prierezy a videnia. Ďalej ich modularizuje do komponentov, ktoré pozostávajú z viacerých spolupracujúcich tried. Potom definuje a implementuje rozhranie pre spoluprácu tejto triedy s okolím, a nakoniec definuje spojenie medzi komponentom a aplikáciou. Týmto sa zabezpečí vyššia miera nezávislosti pretože komponent nemusí vedieť implementačné detaily aplikácie, a na druhej strane konkrétna implementácia komponentu je skrytá za rozhraním pre spoluprácu.

Lepšou modularizáciou pretínajúcich záležitostí podporuje znovupoužitie komponent. Vďaka vyššej nezávislosti komponentov od konkrétneho naviazania na aplikáciu je možné tieto komponenty použiť aj v iných kontextoch. Zároveň umožňuje použitie viacerých komponent pri použití jedného naviazania na aplikáciu.

Na tkanie používa implementáciu z jazyka AspectJ čoho výsledok je efektívny bytekód. Skladané triedy transformuje na štandardné triedy v Jave. A implementácia naviazaní používa hašovacie tabuľky na rýchle mapovanie komponentov.

Hlavnou črtou a zmenou oproti AspectJ je implementácia takzvaných „virtuálnych tried" a ich spájania. Toto riešenie ponúka podobné výhody ako klasické virtuálne metódy, ale na úrovni tried. Spájanie zas umožňuje skĺbiť vlastnosti dvoch tried podobne ako je to u subjektovo orientovaného programovania.

CaesarJ Triedy

CaesarJ používa špeciálne kľúčove slovo „cclass" na definíciu tried, aby boli oddelené od jednoduchých tried Jave. Nie je možné dedenie medzi Caesar triedami a čistými Javovskými triedami. Dovolená je iba objektová kompozícia a implementácia rozhraní.

```
public interface AJavaInterface {
}

public class APlainJavaClass {
    ACaesarClass aRefToACaesarClass;
}

public cclass ACaesarClass implements AJavaInterface {
    APlainJavaClass aRefToAPlainJavaClass;
    java.util.Vector aRefToAnotherPlainJavaClass;
}
```

Spolupracujúce triedy

Triedy, ktoré spolupracujú sa dajú spojiť do jednej skupiny. Táto sa v CaesarJ nazýva „Collaboration". Syntakticky ide o triedu, ktorá obsahuje aspoň jednu ďalšiu caesarj podtriedu.

```
// collaboration
public cclass Graph {
    public cclass Edge {...}
    public cclass UEdge extends Edge {...}
    public cclass Node {...}
}
```

Virtuálne triedy

Všetky vnútorné triedy v skupine spolupráce sú virtuálne, podobne ako metódy v bežných Java triedach. Vnútorné triedy môžu byť v CaesarJ predefinované v každom potomkovi zaobaľujúcej triedy.

```
public cclass Graph {
    public cclass Edge {
        Node start, end;
    }
    public cclass UEdge extends Edge {...}
    public cclass Node {...}
}
```

```
public cclass WeightedGraph extends Graph {
    public cclass Edge {
        float cost;
    }
    public cclass Node {
        float cost;
    }
}
```

V tomto príklade bol do triedy spolupráce WeightedClass so podtried Edge a Node pridaný atribút cost. Virtuálne triedy umožňujú dedičnosť tried podobne ako virtuálne metódy v klasickej Jave.

Podstatné je že každý odkaz na virtuálnu triedu je vždy naviazaný na najšpecifickejšiu definíciu v kontexte objektu, čiže v kontexte skupiny WeightedGraph napríklad trieda UEdge zdedí novú implementáciu triedy Edge aj s novými atribútmi (a nie Graph.Edge). To isté platí aj pre premenné start a end pôvodne definované v Graph.Edge, tie budú tiež obsahovať atribút cost z novej triedy WeightedGraph.Node.

Mechanizmus spájania

CaesarJ umožňuje spájanie tried pomocou operátora &.

```
public cclass ColoredGraph extends Graph {
    public cclass Node {
        Color color;
    }
}

public cclass ColoredWeightedGraph extends ColorGraph & WeightedGraph {
    ...
}
```

V príklade sú spojené dve triedy do novej, ktorá má vlastnosti oboch pôvodných tried. Navyše toto spojenie sa prenesie aj do všetkých vnútorných virtuálnych tried a premenných v nich definovaných. Čiže napríklad ColoredWeightedGraph.Node bude obsahovať atribúty cost aj color.

Body spájania a videnia

CaesarJ je implementovaný nad AspectJ kompilátorom a až na malé obmedzenia niektorých bodov spájania je možné použiť takmer rovnaký model bodov spájania a bodových prierezov. Každá trieda v CaesarJ môže obsahovať bodový prierez a videnie, rovnako ako v AspectJ.

```
public cclass AnAspect {
    public pointcut APointcut() : .... ;
    public before() : APointcut() { ... }
}
```

Aktivácia aspektov

Oproti AspectJ umožňuje CaesarJ dynamické vytváranie a aktivovanie aspektov. Používa na to kľúčové slovo „deploy", ktoré určuje rozsah platnosti aspektu.

```
AnAspect a = new AnAspect();
// a neaktívny
deploy( a ) {
    // a je aktívny
    ...
}
// a je opäť neaktívny
```

Ak chceme aby aspekt bol aktivovaný automaticky musíme to špecifikovať kľúčovým slovom „deployed" v definícii aspektu (triedy).

```
public deployed cclass AnAtCompileTimeActivatedAspect {
   // ...pointcuts...
   // ...advices...
}
```

Baliace triedy

Namiesto priamej dedičnosti z čistých Java tried umožňuje CaesarJ vytvárať takzvané baliace triedy (wrapper classes). Objekty týchto tried sú dynamicky vytvárané a sú naviazané na balený objekt. Ak je zabalený objekt zrušený manažérom pamäte tak je automaticky zrušený

aj baliaci objekt. V prípade že už boj nejaký objekt raz zabalený, použije sa prvýkrát vytvorený baliaci objekt.

```
public class X {
    public int x1;
    public int m1() {...}
}

public cclass ACollaboration {
    public cclass XWrapper wraps X {
        int x2;
        public int m2() {
            return wrappee.m1() + x2;
        }
    }

    public void doSomethingWith(ApplicationModel.X x) {
        XWrapper(x).m2(); // wrapping x
    }
}
```

2.4.3 AspectC++

Ide o snahu o implementáciu aspektovo orientovaného rozšírenia pre C++. Vychádza z AspectJ a v maximálnej možnej miere kopíruje jeho syntax a sémantiku.

Model bodov spájania a jazyk bodových prierezov

AspectC++ poskytuje statické aj dynamické body spájania.

Statické – sú definované pomocou regulárnych výrazov. Nie všetky sú podporované ako cieľ videnia.

- Triedy, štruktúry, uniony
- Menné priestory - namespaces
- Funkcie (všetky typy funkcií ako metódy, operátory, atd.)

Dynamické – sú definované funkciami (kľúčovými slovami) nad regulárnymi výrazmi.

- Volanie funkcie – call(...) a execution(...)
- Vytvorenie objektu – construction(...)
- Deštrukcia objektu – destruction(...)

Bodové prierezy sú špecifikované a filtrované pomocou funkcií a operátorov

- &&, ||, ! – logické and, or a negácia
- cflow(pointcut) - všetky body spájania v dynamickom kontexte parametra
- base a derived – podľa dedičnosti triedy
- within – iba body v rámci statického kontextu parametra (v triede, funkcii)
- that, target, result, args – podľa typu objektu, typu objektu volanej metódy, typu výsledku a typu parametrov.

Podporované regulárne výrazy ponúkajú „%“ a „...“ ako ?hviezdičku?, takže umožňuje použiť napríklad takéto zápisy:

- „Foo::...::Bar(...,int)“ – funkcia Bar v kontexte Foo a s posledným parametrom typu int
- „const % *“ – všetky pointre na konštantné typy

Bodové prierezy môžu byť pomenované. V kontexte aspektu môžu byť virtuálne, čo umožní ich predefinovanie v zdedených aspektoch.

Videnia

Pre statické body spájania umožňuje použiť iba typ „introduction“ na vloženie nových prvkov do existujúcich štruktúr. Použiteľné sú všetky syntakticky správne konštrukcie (v závislosti od cieľa vkladania).

```
advice "AClass" : int novy_atribut;
```

Pre dynamické body spájania umožňuje použiť štandardné typy rád a to:

- before
- after
- around

Každé takéto videnie má k dispozícii kontextový objekt „tjp“. Tento objekt poskytuje informácie o type objektu, cieľovom objekte volanej metódy, návratovej hodnote a parametroch.

2.4.4 JBOSS AOP

Aspekty sú implementované ako čisté Java triedy, nezavádza žiadne nové prvky do jazyka. Externé definície bodových prierezov sú v XML. Naviazanie videní na bodové prierezy je definované tiež cez XML, alebo anotáciami v zdrojovom kóde (Java 5). Využíva sa v J2EE aplikačnom serveri JBoss 4.

Body spájania

- execution – Vykonanie metódy alebo konštruktora
- get, set – getter/setter metódy na atribúty
- field – čítanie alebo zápis atribútov
- all – čokoľvek spojené s konkrétnou triedou
- within / withincode – v rámci typu / kódu
- has – ak má metódu alebo konštruktor
- hasfield – ak má atribút

Jednotlivé body spájania sa môžu kombinovať logickými operátormi ako AND, OR ...

Dynamické bodové prierezy sú definované ako triedy s implementovaným rozhraním DynamicCFlow.

```
public interface DynamicCFlow {
      boolean shouldExecute(Invocation invocation);
}
```

Ako videnie sa môže použiť akákoľvek metóda s nasledujúcou signatúrou

```
Object methodName(Invocation object) throws Throwable
```

2.4.5 Hyper/J

Implementácia viacrozmernej separácie záležitostí (MDSoC) v jazyku Java. Tento prístup je nezávislý na jazyku a vychádza zo subjektovo orientovaného programovania, aj preto využíva symetrický prístup.

Umožňuje iba statické definovanie aspektov, pretože pracuje na úrovni zdrojových textov, ktoré spája a modifikuje. Výsledkom spájania sú opäť zdrojové texty.

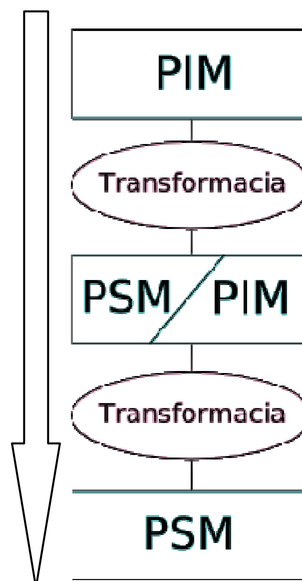
Aplikácia sa delí na hypermoduly a hypervýrezy, ktoré sú do výsledného programu kombinované na základe spájacích pravidiel (merge rules). Viac v kapitole 2.3.2

3 Model Driven Architecture (MDA)

Ide o metodológiu softvérového návrhu [24] vypracovanú členmi OMG (Object Management Group). Funkčnosť softvérového systému je definovaná pomocou platformovo nezávislého modelu, ktorý sa pomocou transformácie prevedie na platformovo závislý model, čo nakoniec môže viesť až ku konkrétnemu kódu. Tieto transformácie sú väčšinou vykonávané pomocou automatizovaných nástrojov. MDA definuje pravidlá na špecifikáciu softvéru pomocou modelu. MDA spája viacero štandardov ako UML (Unified Modelling Language), MOF (Meta Object Facility), XMI (Extensible Model Interchange).

MDA definuje štyri modely: Computation Independent Model (CIM), Platform Independent Model (PIM), Platform Specific Model (PSM) a Implementation Specific Model (ISM).

Platforma v ponímaní MDA je dosť vysoko abstraktný pojem a je závislý od úrovne pohľadu. Platforma môže reprezentovať napríklad konkrétny programovací jazyk, aplikačný server, alebo operačný systém.



CIM – model nezávislý od výpočtov a od výkonnej logiky, obsahuje iba popis domény.

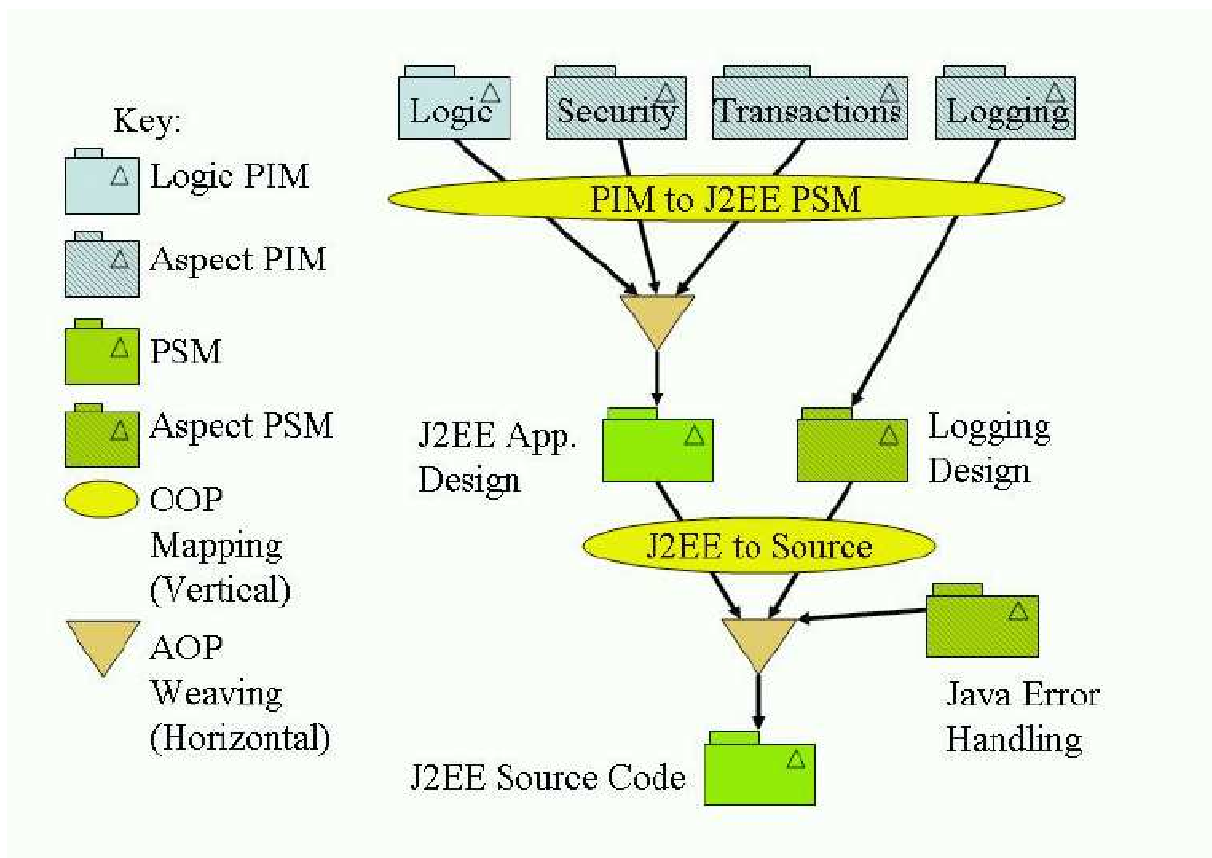
PIM – platformovo nezávislý model popisuje systém nezávisle od platformy, čiže nepoužíva žiadne prvky, ktoré by bránili jeho transformácii aj na iné cieľové platformy. Po transformácii sa PIM stáva PSM vzhľadom na prvú platformu, ale môže sa stať opäť PIM pre inú špecifickejšiu platformu. Takýmto zreťazením je možné z modelu dostať konkrétnu implementáciu softvérového systému.

PSM – model špecifický a obmedzený ohraničeniami platformy, napríklad programovacím jazykom alebo operačným systémom.

Modely poskytujú abstrakciu od fyzického systému, čo umožňuje vývojárom sústrediť sa na dôležité funkčné prvky systému a nie na implementačné detaily špecifické pre danú cieľovú platformu. Modely môžu byť vytvorené pred systémom, alebo odvodené od už existujúceho systému ako pomôcka k pochopeniu jeho štruktúry a správania. Pretože na jeden systém môže existovať viacero pohľadov a požiadaviek s rôznym zameraním, je vhodné modely

koncipovať tak aby ich bolo možné prevádzať na iné zobrazenia, vystihujúce konkrétne črty systému, ktoré nás práve zaujímajú.

Modely sú jadrom prístupu MDA. Preto sa najprv musíme zamerať na modelovanie jazykov AOP, a to tak aby boli skryté črty špecifických implementácií. Vo svete OOP je už štandardom používanie UML modelov počas návrhu aplikácií, bohužiaľ UML nie je priamo schopné modelovať pretínajúce záležitosti z AOP. Dá sa to ale riešiť rozšírením UML o meta-modely, ktoré umožnia mapovať aj roztrúsené záležitosti (snaží sa o to napríklad Theme/UML).



Obrázok 6 - MDA transformácia s použitím AOP [15]

Synchronizácia modelov a zdrojových kódov sa takmer výlučne vykonáva plne manuálne, prípadne s malou pomocou nástrojov, ktoré dokážu vytvárať jednoduché implementácie tried v objektovo orientovaných jazykoch. MDA sa snaží o automatizáciu tohto procesu v čo najväčšej miere.

V súčasnosti existujú iba obmedzené nástroje implementujúce MDA. Väčšinou ide iba o transformácie z jednoduchého doménového modelu, prípadne objektovo relačné mapovanie do relačnej databázy alebo do J2EE kontajnerov. Nástroje na transformáciu modelov sú

napríklad balíky zo série IBM Rational Software, XCode, AndroMDA, OpenMDX, openArchitectureWare [25].

Postupom času sa čoraz viac usiluje o spojenie MDA a AOP, zatiaľ ale iba na úrovni transformácie modelov pre konkrétne jazyky do kódu [27][25].

Cieľom tejto práce je pokúsiť sa priblížiť MDA k aspektovo orientovanému programovaniu, s cieľom definovať niektoré konkrétne AO jazyky ako platformy v ponímaní MDA. Pomocou transformácií modelov by malo byť možné automaticky transformovať jednoduché AO programy z jednej platformy-jazyka na inú.

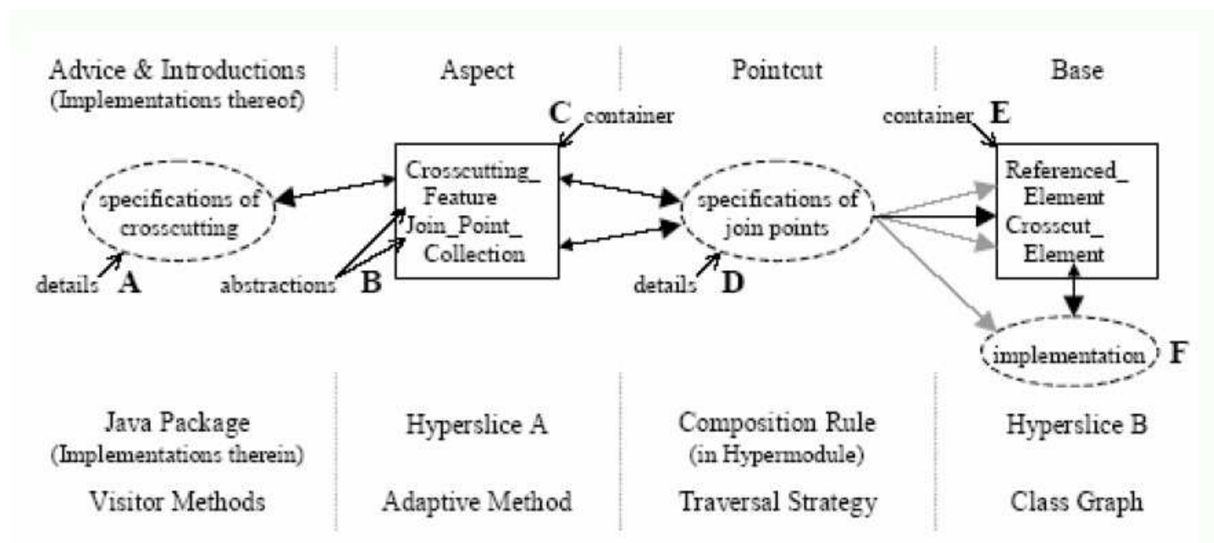
4 Prvky AOP jazykov

Na vytvorenie jazykovo nezávislého modelu musíme vytvoriť určitú abstrakciu od všetkých aspektovo orientovaných jazykov. Táto by mala byť nadmnožinou funkčnosti jednotlivých vybraných jazykov, ktorými sa budeme zaoberať. Tento model by nám mal umožňovať použitie symetrického aj nesymetrického prístupu, medzitypové deklarácie, statické aj dynamické body spájania. Základné prvky tohto modelu môžeme definovať pomocou týchto kritérií, ktoré pokrývajú artefakty takmer každého aspektovo orientovaného jazyka:

1. Čo sa pretína – samotná implementácia funkčnosti pretínajúcej záležitosti. Môže ísť o implementáciu štruktúry alebo správania. V prípade AspectJ je to kód videnia, alebo nové atribúty a metódy tried. V prípade adaptívneho programovania sú to visitor metódy, a v prípade kompozičných filtrov je to implementácia metód filtra. Táto časť by mala byť v maximálnej miere nezávislá od ostatných tak aby bolo možné flexibilné použitie na rôzne problémy.
2. Kde sa to môže pretínať – body spájania a bodové prierezy, miesta v programe kde môže nastať pretínanie rôznych záležitostí. Napríklad volanie metód, prístup k atribútom v triedach, vytváranie nových objektov, vyvolanie výnimky, a podobne. V prípade kompozičných filtroch je to iba volanie metód. Tieto miesta môžu byť statické, teda určiteľné pomocou štruktúry programu už počas návrhu alebo dynamické, čiže je ich možné určiť až z dynamických parametrov získaných za behu programu.
3. Ako sa to pretína – určuje vzťah medzi implementáciou pretínajúcej záležitosti a miestom kam sa má aplikovať spolu so spôsobom ako. AspectJ toto obsahuje v definícii videnia, kde je priamo uvedený bodový prierez, ComposeJ zas využíva definíciu filtra. Aspektovo

orientované jazyky väčšinou definujú tri typy a to pred, za alebo namiesto pôvodnej implementácie (before, after, around). Môžu ešte existovať iné ako napríklad presmerovanie metódy na inú metódu, alebo úprava parametrov volania metódy. V podstate sú ale všetky tieto možnosti obsiahnuté v „around“ pretože samotná implementácia môže zavolať pôvodnú pred alebo za vykonaním svojho kódu. Jediná podmienka aby toto bolo umožnené je dostatočný prístup ku kontextu bodu spájania. A zároveň je prehľadnejšie priame použitie napríklad videnia „after“ namiesto volania pôvodného bodu spájania na začiatku „around“ videnia.

4. Výsledok spájania – výsledný kód aplikácie či už vo forme zdrojových textov alebo binárne skompilovaná podoba, ktorá obsahuje všetky pretínajúce záležitosti. Môže ale ísť aj o konštrukcie, ktoré je možné dynamicky aplikovať za behu programu.



Obrázok 7 - Konštrukcie v jednotlivých AOP jazykoch (AspectJ, Hyper/J, DemeterJ)

Aby bolo možné využiť MDA musí vytvorený základný PIM model jasne a čo najjednoduchšie definovať všetky štyri časti. Zároveň musí byť transformovateľný do nižších PSM modelov pre konkrétne jazyky, takže musí byť dostatočne abstraktný. Navyše niektoré vlastnosti jazykov nie sú dostupné v iných jazykoch, takže je potrebné aby bolo možné tieto transformovať do iných konštrukcií v cieľovom jazyku, ktoré nahradzujú chýbajúcu funkcionalitu. Napríklad prístup k atribútom objektov nie je zachytiteľný pomocou kompozičných filtrov, toto je ale možné obísť tak že sa k nim bude pristupovať iba pomocou metód, ktoré už je možné odchytiť.

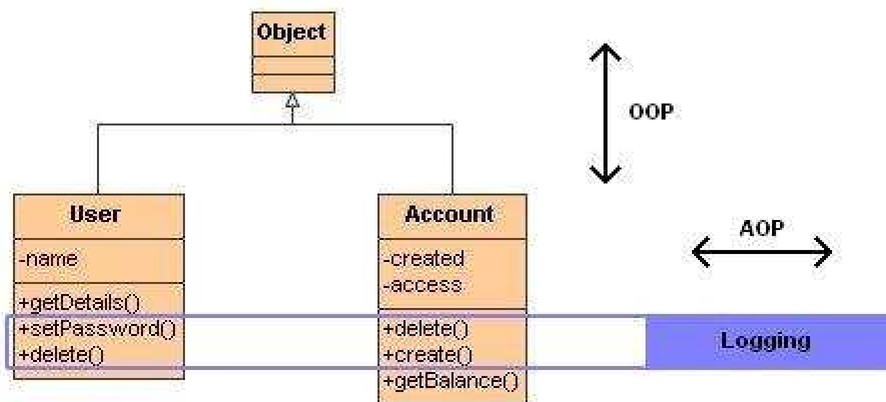
5 Možnosti modelovania v AOSD

Pre prístup MDA sú základom modely nad MOF (Meta Object Facility [24]), preto sa musíme vo veľkej miere zamerať na modelovanie AOP jazykov. Aby sme mohli využiť prínosy MDA k návrhu systémov, musíme mať najprv definované konkrétne platformovo nezávislé ale aj závislé (na konkrétnom jazyku) modely.

Pre štandardný OOP prístup bol vyvinutý jazyk UML, ktorý poskytuje prostriedky na vertikálnu dekompozíciu programov. Pre AOP zatiaľ ale neexistuje žiadny štandardný kompletný platformovo nezávislý spôsob modelovania štruktúry systému. Väčšina navrhnutých riešení sa zakladá na jazyku AspectJ a cieľom je transformácia priamo do kódu tohto jazyka. Spôsobené je to pravdepodobne tým že AOSD je pomerne málo rozšírené a AspectJ je najpopulárnejší AOP jazyk.

AOP predstavuje posun v paradigme programovania a tým pádom aj modelovania. Preto je potrebné také rozšírenie, ktoré bude schopné zachytiť aspekty a pretínajúce záležitosti už na úrovni špecifikácie a modelu. Pretože AOP predstavuje nadmnožinu k OOP musí ísť pri modelovaní o rozšírenie, ktoré je nadmnožinou jazyka UML s využitím mechanizmov, ktoré nám UML ponúka. V aspektovo orientovanom modelovaní by malo ísť čisto o prechod od OOP k AOP modelom so zachovaním jazykovej a platformovej nezávislosti.

Zatiaľ čo UML poskytuje možnosti na modelovanie vertikálnej štruktúry OOP systémov kde je dominantná objektová dekompozícia na nadradené a odvodené triedy, AOP vyžaduje modelovanie horizontálne rozložených pretínajúcich záležitostí. Ak by sme sa snažili modelovať AOP aplikáciu v štandardnom UML dospeli by sme k riešeniu, ktoré by malo rovnakú zviazanosť ako implementácia v klasickom OOP jazyku bez AOP rozšírení. Je to preto lebo UML neobsahuje možnosť ako určiť vzťah medzi entitami v modeli bez vytvorenia väzby medzi nimi. Pretože hlavný zámer AOP je oddelenie pretínajúcich záležitostí potrebujeme pri modelovaní možnosť špecifikovať na jednom mieste všetky bodové prierezy na ktoré sa majú aplikovať dané videnia.



Obrázok 8 - Vertikálna a horizontálna dekompozícia

V tejto práci sa budem zaoberať hlavne využitím rôznych rozšírení UML. Je prirodzené zaoberať sa použiteľnosťou jazyka UML pri aspektovo orientovanom modelovaní, pretože je to rozšírený štandard v oblasti objektovo orientovaného modelovania, ponúka možnosti na špecifikáciu, vizualizáciu a dokumentáciu prvkov systému. Ďalej UML je všeobecný jazyk použiteľný pre veľké množstvo aplikačných domén. Navyše je rozšíriteľný aby umožňoval modelovanie špeciálnych prípadov.

Pretože štandardný jazyk UML neposkytuje dostačujúce možnosti musia sa vytvoriť rozšírenia, ktoré budú použiteľné na modelovanie aspektov. Pre cieľ tejto práce podstatné vlastnosti takýchto rozšírení sú:

1. Platformová nezávislosť – pretože cieľom je zjednotiť využitie rôznych AOP jazykov pomocou MDA, potrebujeme vytvoriť minimálne jeden platformovo nezávislý model systému, ktorý budeme môcť transformovať na špecifickejšie modely a nakoniec až na samotný kód.
2. Všestrannosť – modelovací jazyk musí byť dostatočne flexibilný aby sme boli schopní modelovať všetky typy problémov, ktoré sú riešiteľné pomocou AOP jazykov. Niektoré návrhy modelovacích jazykov sa obmedzujú na konkrétne špeciálne typy aplikácií alebo iba na jednu oblasť modelu ako napríklad na aspektovo orientované modely prípadov použitia.
3. Dodržiavať oddelenie pretínajúcich záležitostí – model musí oddeľovať samotné pretínajúce záležitosti tak aby mal čo najmenšiu zviazanosť jednotlivých prvkov v modeli. Špecifikácia aspektov, bodov spájania a bodových prierezov by mala byť oddelená.

Pretože definovanie kompletného nezávislého modelu pokrývajúceho možnosti platformovo závislých modelov pre všetky vybrané jazyky by bolo príliš zložité, zameriam sa v tejto práci na špecifikáciu jedného jednoduchého platformovo nezávislého jazyka, ktorý bude pokrývať dostatočne veľkú časť vybraných AOP jazykov.

Aby sme sa mohli zaoberať modelovaním aspektov musíme si definovať základné prvky modelu na základe spoločných vlastností vybraných AOP jazykov.

5.1 Aktuálny stav v oblasti aspektovo orientovaného modelovania

Momentálne prebieha intenzívny výskum s cieľom definovať štandardný prístup k modelovaniu v AOSD [17], ktorý by bol dostatočne platformovo nezávislý a kompletný. Podľa výsledkov tohto výskumu sa ako najpoužiteľnejšie javí použitie Theme/UML v spojení s expresívnejšími a komplexnejšími prostriedkami na modelovanie bodov spájania ako napríklad JPDD [37]. Doteraz ale nebolo predstavené žiadne riešenie, ktoré by spĺňalo všetky požiadavky na použitie v MDA. Tak ako Theme/UML aj JPDD obsahujú rozšírenia sémantiky nad rámec UML a preto by ich použitie vyžadovalo špeciálnu podporu nástrojov aby bolo možné ich využitie v MDA.

V [17] sú ako predbežné výsledky analýzy súčasných prístupov modelovania AO programov spomenuté tieto zistenia

Prevaha UML – drvivá väčšina AOM prístupov je založená na jazyku UML. Prístupy založené na iných, väčšinou doménovo špecifických jazykoch sa ešte iba začínajú objavovať. Takisto veľmi málo jazykov vychádza z novej špecifikácie UML 2.

Všeobecné jazyky – AOM prístupy sú väčšinou nezávislé od konkrétnej domény a aspektov, ktoré sa snažia modelovať. Ale objavujú sa aj prístupy, ktoré sa snažia byť doménovo závislé kde navrhujú aby AOM jazyk obsahoval konštrukcie pre konkrétne aspekty v rámci domény ako logovanie, bezpečnosť a podobne.

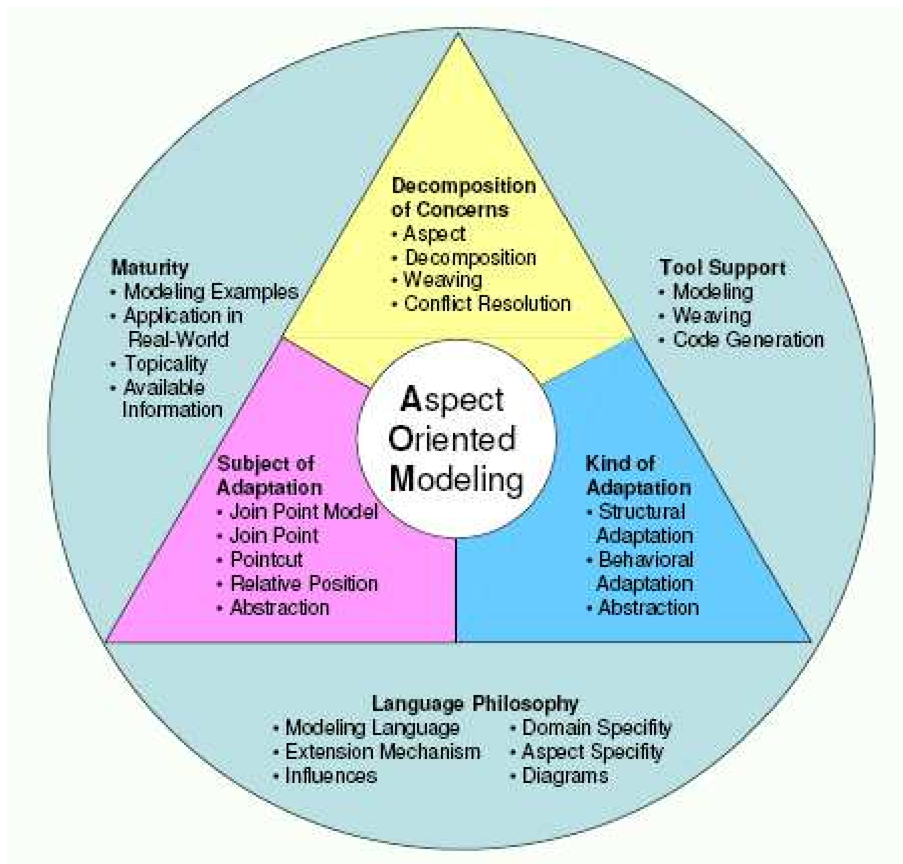
Dynamické diagramy – každý prístup vo veľkej miere používa statické štrukturálne diagramy, naopak iba málo využíva aj dynamické diagramy správania, na popis dynamických vlastností aspektov alebo špecifikáciu kedy sa má použiť pretínajúce správanie.

Podpora nástrojov – Existuje mnoho nástrojov na modelovanie v UML, vďaka čomu je možné modelovanie pre prístupy využívajúce UML profily. Ale čo sa týka tkania aspektov alebo generácie kódu tak takáto podpora prakticky neexistuje. Vyplýva to hlavne z toho že neexistuje žiadna ustálená špecifikácia AOM jazyka, na ktorej by sa dalo stavať.

Použitie v reálnych aplikáciách – Takmer žiadny doteraz predstavený AOM jazyk nebol použitý v skutočnom projekte.

Model bodov spájania – Takmer všetky AOM prístupy preberajú špecifikáciu bodov spájania a bodových prierezov z konkrétneho jazyka, čím sa stávajú platformovo závislými vzhľadom na ich použitie v MDA.

Symetrickosť prebratá z jazykov – symetrickosť AOM prístupu je vo veľkej miere prebratá z konkrétneho AOP jazyka, podobne ako tomu je v prípade modelu bodov spájania. Pretože AOSD je tvarovaný jazykmi ako AspectJ je veľmi veľká časť AOM prístupov zameraná asymetricky. Prístupy, ktoré vychádzajú z modelu subjektívneho programovania zas naopak využívajú prevažne symetrický prístup.



Obrázok 9 - Kritériá hodnotenia modelovacích jazykov [17]

V nasledujúcich kapitolách nasleduje popis niektorých najčastejšie spomínaných spôsobov modelovania apektovo orientovaných programov.

5.2 UML profily

UML profily rozširujú štandardný UML o nové entity ako stereotypy, ich meta-atribúty a obmedzenia ktoré môžu byť použité na identifikáciu AOP entít v systéme. Takéto riešenia navrhol napríklad Aldawud [15] a Stein [21]. Použitie profilov je analyzované hlbšie v [28].

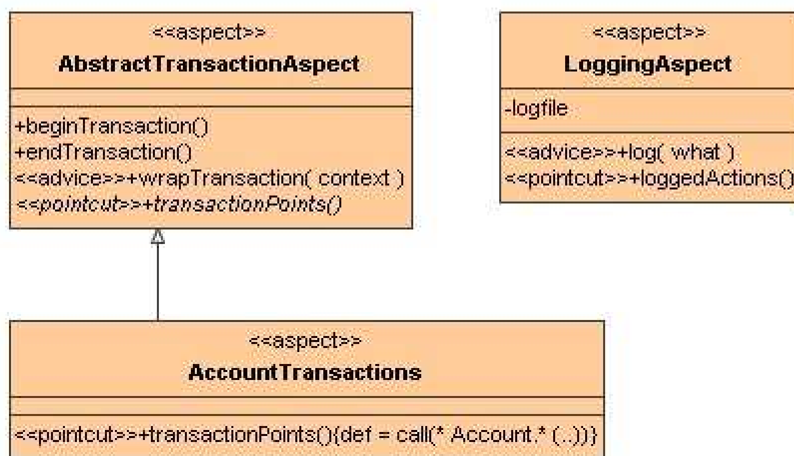
Ponúkajú grafickú reprezentáciu aspektov a vzťahov medzi videniami a zviazanými bodmi spájania. Ich výhoda je že sú jednoduché a majú dobrú podporu v nástrojoch.

Hlavným problémom je nemožnosť jednoduchého zápisu bodových prierezov a platformová závislosť na konkrétnom aspektovo orientovanom jazyku. Väčšinou je tento spôsob spojený s jazykom AspectJ a jemu podobnými.

Bodové prierezy v prípade UML profilu buď označíme pomocou stereotypov, v tom prípade musíme ku každému bodu spájania priradiť daný stereotyp. Toto ale spôsobí roztrúsenú definíciu čo je proti pointe AOP. Pri väčších projektoch sa takáto reprezentácia stáva neprehľadnou a nie je dostatočne škálovateľná.

Druhý spôsob je bodové prierezy špecifikovať pomocou hodnôt meta-atribútov textovou formou v UML modeli. V tomto prípade ale musíme použiť platformovo závislú špecifikáciu na opis bodových prierezov a bodov spájania napríklad použitím jazyka AspectJ.

UML profily musia zároveň spĺňať základnú špecifikáciu jazyka UML a dodržiavať všetky jeho obmedzenia, takže nie je možné efektívne modelovať niektoré vlastnosti.

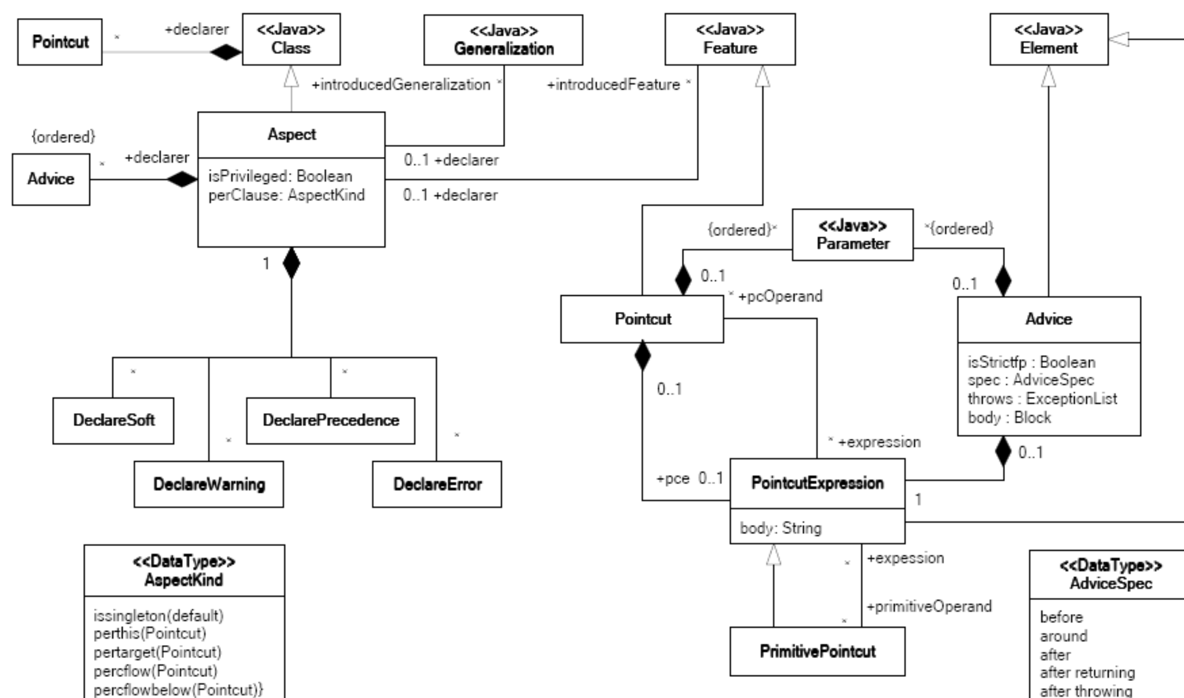


Obrázok 10 - Príklad UML profilu na modelovanie AspectJ

5.3 MOF metamodely

Ide takmer o rovnaký prístup ako v prípade UML profilov ale na nižšej úrovni špecifikácie UML jazyka. Základom takýchto modelov je špecifikácia MOF, na ktorej je postavený aj jazyk UML. Metamodely definujú nové modelovacie jazyky nad MOF rovnocenné napríklad triedam, závislostiam a podobne. Takýmto modelom je možné presnejšie popísať konkrétny AO jazyk, ale za cenu zhoršenia alebo až straty podpory modelovacích nástrojov, ktoré nebudú schopné pracovať s takýmto metamodelom. Úpravou metamodelu získame väčšie možnosti ako rozšírením jazyka UML pre prácu s AOP, pretože nebude obsahovať obmedzenia definované v špecifikácii UML, ktorá je stavaná vyslovene na OOP jazyky.

Metamodely vyjadrujú väčšinou priamo model konkrétneho AOP jazyka so všetkými jeho prvkami a vzťahmi medzi nimi.



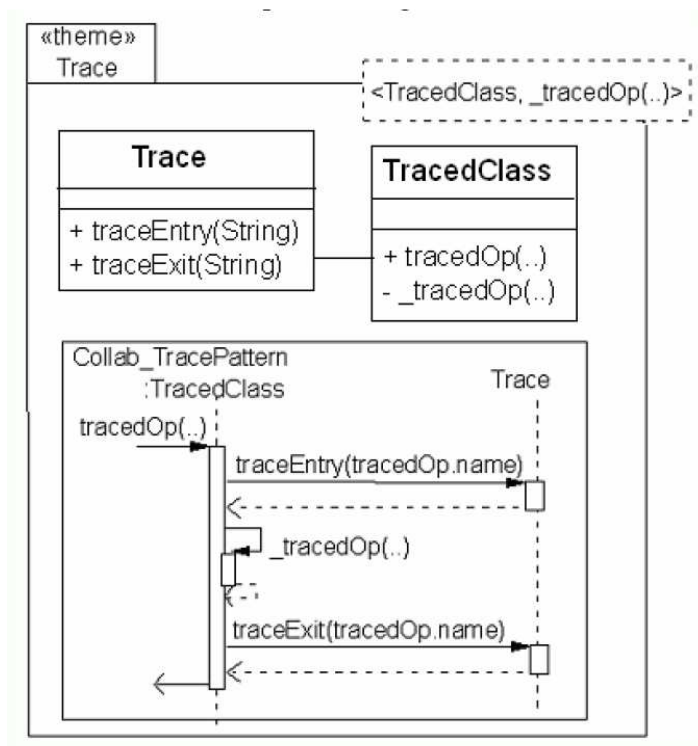
Obrázok 11 - AspectJ metamodel [22]

5.4 Hybridný prístup, Theme/UML

Hybridné prístupy kombinujú použitie metamodelov a rozšírení štandardného UML. Metamodel pre modelovanie základných prvkov AOP jazyka je rozšírený aby umožňoval vyjadrovanie bodov spájania, a ich prepojení ku konkrétnym aspektom a vkladaným parametrom tried. Na základe týchto prepojení sa potom spájajú viaceré pretínajúce záležitosti.

Theme/UML je prístup využívajúci „témy" čo sú v podstate parametrické triedy alebo balíky, ktoré sa aplikujú na konkrétny základný komponent, a umožňujú pridávať do tried nové atribúty, operácie a tiež špecifikovať videnia pre konkrétne body spájania.

Téma je reprezentovaná balíkom v UML, ďalej využíva diagram tried a sekvenčné diagramy. Viac je o tom v [19].



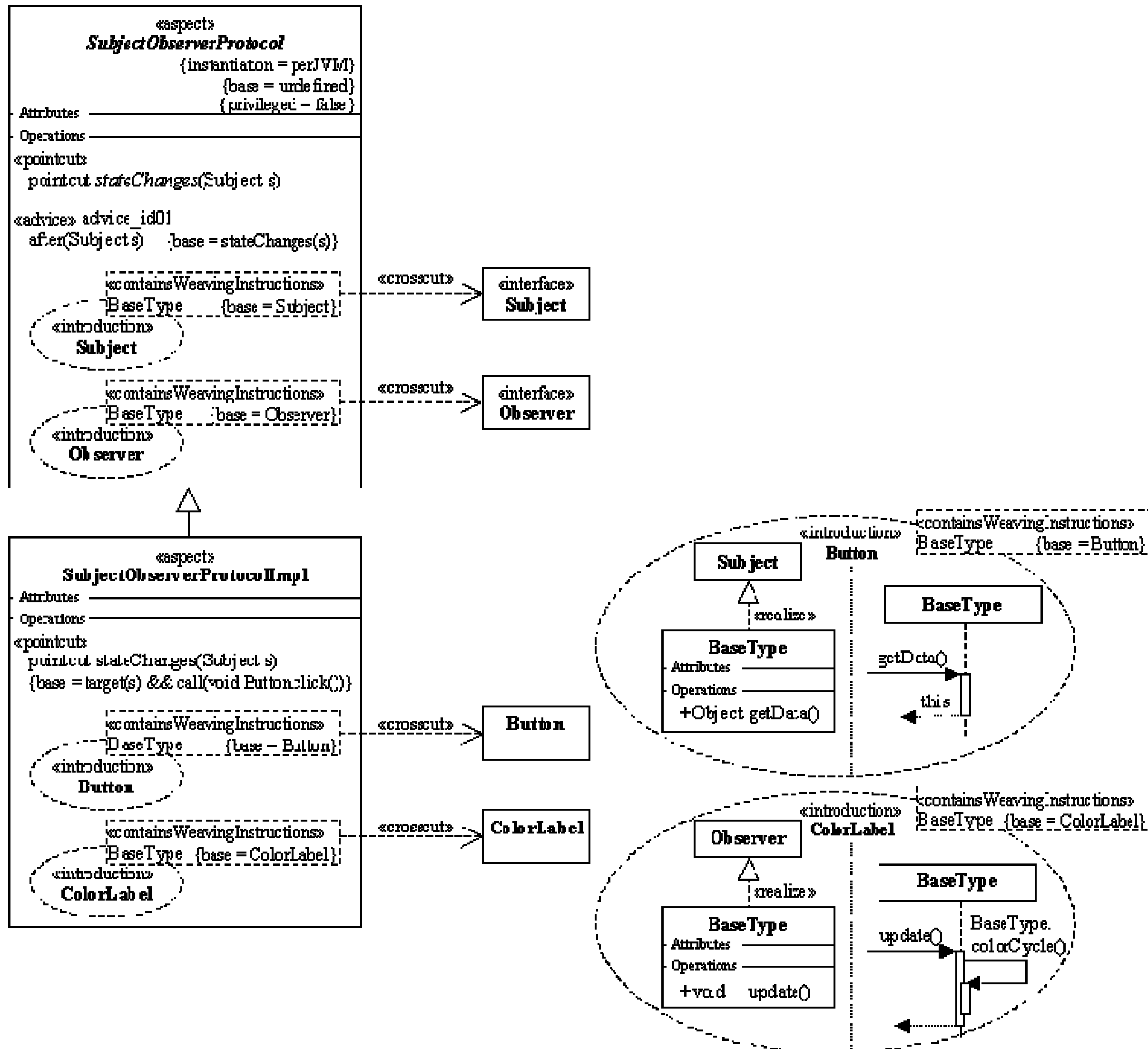
Obrázok 12 - Theme UML aspect [18]

5.5 AODM

Aspektovo orientovaný návrhový model (AODM [21]) je založený na UML profile (verzie 1.3) a modeluje jazyky založené na AspectJ. Poskytuje všetky konštrukcie jazyka ako aspekty, bodové prierezy, videnia a medzitypové deklarácie.

Aspekty sú reprezentované triedami so stereotypom, bodové prierezy a videnia sú operácie. Navyše tieto stereotypy obsahujú meta-atribúty, ktoré popisujú špecifické vlastnosti konštrukcií jazyka AspectJ. Popis pretínania je modelovaný parametrickými diagramami spolupráce.

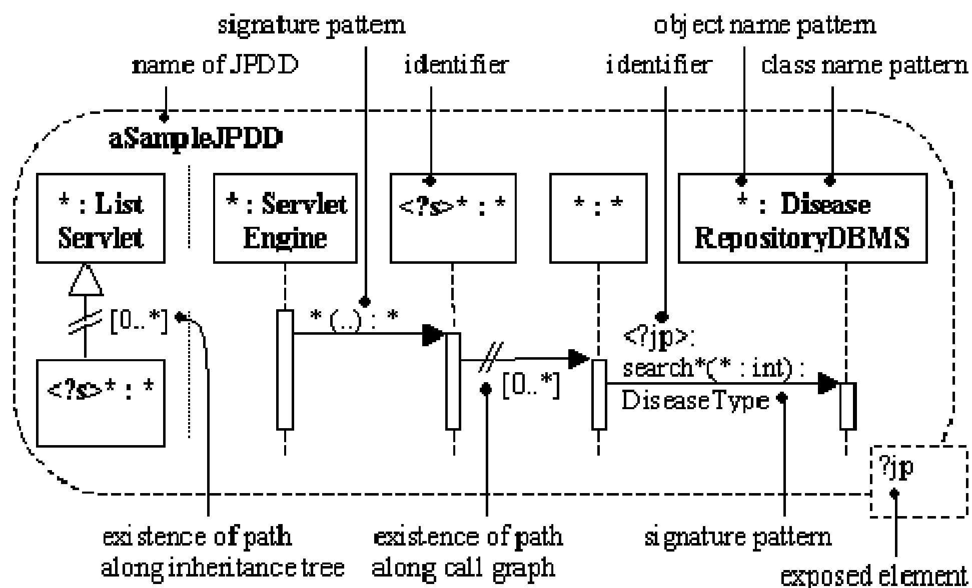
Hlavným nedostatkom čistého UML je nemožnosť dostatočného modelovania bodových prierezov. Toto viedlo k vývoju Join Point Designation Diagrams (JPDD) – diagramov na popis bodových prierezov.



Obrázok 13 - AODM Diagramy [21]

5.6 JPDD

Join Point Designation Diagrams (JPDD) [29] – sú diagramy na vizuálny popis bodových prierezov v AOM. Umožňujú modelovanie statických aj dynamických bodov spájania, na úrovni sekvenčného diagramu zobrazujú aj zložité dynamické výbery bodových prierezov. S bodovými prierezmi je v AOM všeobecne značný problém pretože na vyjadrenie všetkých vzťahov v modeli UML sa používajú asociácie, ak by sme chceli použiť rovnaký spôsob na definovanie bodových prierezov, výsledok by bol veľmi neprehľadný a obsahoval by množstvo neprehľadných relácií a celkovo by model nebolo možné udržiavať. Ďalší problém je vizualizácia výberu bodov spájania, tento problém riešia JPDD aj keď za cenu značne komplikovaných diagramov.



Obrázok 14 - JPDD diagramy [29]

5.7 Iné prístupy

Existujú aj iné prístupy založené na grafoch, UML 2 závislosti „package merge“, doménovo špecifických prístupoch. Prehľad viacerých takýchto spôsobov je napríklad v [17] a [18]. Väčšina je ale založená buď na UML profile alebo MOF metamodeli, pričom zatiaľ nie je jednoznačné, ktorý prístup je vhodnejší, pretože existuje viacero argumentov pre profily [21][33][28] ako aj pre metamodely [22][23].

6 Návrh AOM jazyka pre MDA

Na overenie použiteľnosti MDA pri aspektovo orientovanom návrhu systémov je potrebné vytvoriť použiteľné PIM a PSM modely pre vybrané jazyky. Pretože MDA vychádza z MOF musia byť tieto modely vytvorené nad MOF (ako je napríklad UML).

Zatiaľ neexistuje žiadny štandardný všeobecný AOM jazyk na modelovanie aspektovo orientovaných systémov, najpoužiteľnejšie sa javí použitie Theme/UML spolu s JPDD na špecifikáciu bodov spájania. Toto riešenie je ale značne komplikované a jeho využitie v transformáciách veľmi náročné, preto sa mu budem snažiť vyhnúť a použiť iný spôsob.

Doteraz boli vytvorené modely väčšinou pre jazyk AspectJ. Pre jazyk DemeterJ je možné využiť štandardné diagramy tried. Hyper/J metamodel je definovaný napr. v [23]. Tieto modely sú ale veľmi jazykovo špecifické a majú iba málo spoločného takže je ťažké vytvoriť nejaký všeobecný model, ktorý by ich všetky prekrýval.

Vzhľadom na zložitosť doteraz predstavených všeobecných modelov a ich vzájomnej odlišnosti som sa rozhodol vytvoriť skupinu jednoduchých modelov založených na UML profile, ktoré budú umožňovať modelovať základné prvky AOP a tiež hlavné prvky vybraných jazykov.

V tejto práci som sa zameral na modely a transformáciu pre jazyky AspectJ, CaesarJ a Kompozičné filtre (ComposeJ alebo Compose*). AspectJ som zvolil pretože je momentálne najpoužívanejším aspektovo orientovaným jazykom používaným v praxi, a navyše je to typický predstaviteľ asymetrického prístupu. CaesarJ zas ponúka zaujímavé možnosti vďaka virtuálnym triedam, a aj keď niektoré konštrukcie majú veľmi blízko k AspectJ a používa sa aj rovnaký kompilátor, tento jazyk nie je úplne asymetrický. Skôr je niekde medzi oboma prístupmi, pretože virtuálne triedy je možné vzájomne kombinovať do výsledného systému, pričom sa vzájomne správajú symetricky. Pritom je stále možné využiť asymetrické wrappery a naviazania spolu s AspectJ bodovými prierezmi na prepojenie s klasickými Java triedami. Nakoniec kompozičné filtre som zaradil do tejto práce kvôli ich nie veľkej zložitosti a zároveň pretože sú značne rozdielne od ostatných prístupov. Definície kompozičných filtrov sú skôr deklaratívne ako procedurálne alebo objektovo orientované. Táto práca vychádza z implementácie ComposeJ [35] a čiastočne berie do úvahy aj nové prvky v implementácii Compose* [36].

6.1 Špecifikácia požiadaviek na AO model

Na overenie prístupu MDA som sa rozhodol modelovať iba základné, ale podstatné prvky z hľadiska AOP. Neskôr je možné analyzovať rozšírenie týchto modelov o zložitejšie konštrukcie a ich transformácie.

Každý model musí byť definovaný v jazyku UML. Bolo by síce postačujúce aby bol definovaný pomocou MOF, takéto modely ale nemajú podporu v nástrojoch a tak by bolo ťažké nejakým spôsobom vytvoriť príklady, prípadne funkčné transformácie takýchto modelov.

Základný jazyk by mal umožňovať modelovať:

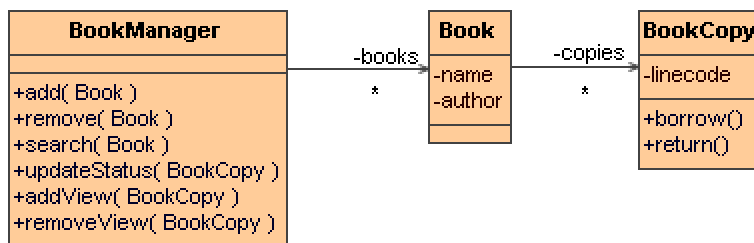
1. Medzitypové deklarácie – základné statické modifikácie určených tried, umožňuje pridávanie atribútov, metód. Ide v podstate o spojenie viacerých pohľadov na nejakú entitu v rámci programu. Je to základným prvkom symetrického prístupu keď sa spájajú rovnocenné časti špecifikované z rôznych uhlov pohľadu do jedného výsledného celku zahŕňajúceho všetky pohľady. Používajú sa ale aj v asymetrických prístupoch na modifikovanie štruktúry už existujúcich tried. Niektoré konkrétne AO jazyky neumožňujú ich použitie, a preto je potrebné aplikovať všetky tieto deklarácie už pri transformácii do konkrétneho PSM modelu.
2. Statické body spájania – statické body spájania sú väčšinou postačujúce, a niektoré jazyky iné ani nepodporujú (napr. HyperJ). Takéto body, aj keď sú statické umožňujú stále špecifikovať body vo vykonávaní programu ako vykonanie metód, prístup k atribútom, vytvorenie objektu a podobne. Všetky takéto statické elementy je možné špecifikovať pomocou diagramu tried. Modelovanie dynamických bodov spájania by bolo v UML zložité, vyžadovalo by použitie dynamických modelov ako sekvenčné alebo komunikačné či stavové diagramy, preto sa ním zatiaľ nebudem zaoberať.

Všetky navrhnuté modely v tejto práci sú reprezentované iba diagramom tried s využitím stereotypov, parametrických tried, balíkov, rozhraní a vzťahov medzi nimi. Nepoužívajú sa žiadne diagramy vyjadrujúce dynamické prvky modelovanej aplikácie. Takéto diagramy sú vhodné na modelovanie zložitejších bodových prierezov, ktoré závisia napríklad od kontextu, alebo dynamických stavov, ktoré je možné špecifikovať iba dynamickým modelom ako napr. JPDD.

Jazyk UML má vzhľadom na jeho úzku spätosť s objektovo orientovanými jazykmi rovnaké nedostatky ako samotné OOP čo sa týka roztrúsených záležitostí. Rovnako ak opri objektovo

orientovaných jazykoch ani v UML nie je možná modulárna špecifikácia roztrúsených záležitostí. Všetko v UML je vyjadrené pomocou elementov (entít, relácií, ...), a jeden element nemôže špecifikovať viacero prvkov, preto je potrebné určité rozšírenie UML, možno až nad rámec profilov. Jedinú výnimku tvoria v UML parametrické elementy ako triedy, balíky, sekvencie, ktoré určitým spôsobom umožňujú aplikovať vzorové správanie na viacero prvkov. Takéto parametrické elementy sa aj využívajú v momentálne najprepracovanejších prístupoch ako je Theme/UML a JPDD.

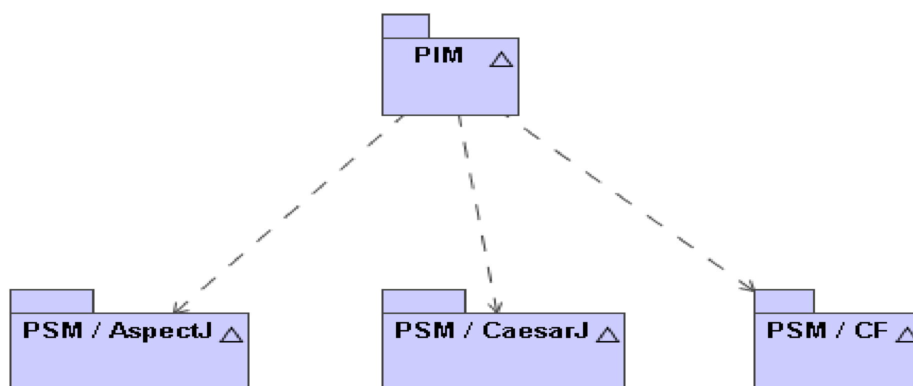
Navrhnuté modely sú prezentované pomocou príkladov nad jednoduchým modelom systému knižnice, ktorá využíva návrhový vzor observer implementovaný pomocou AO prístupu. Rovnaký príklad je uvedený vo viacerých publikáciách napr. [25][30]



Obrázok 15 - Základný model [26][30]

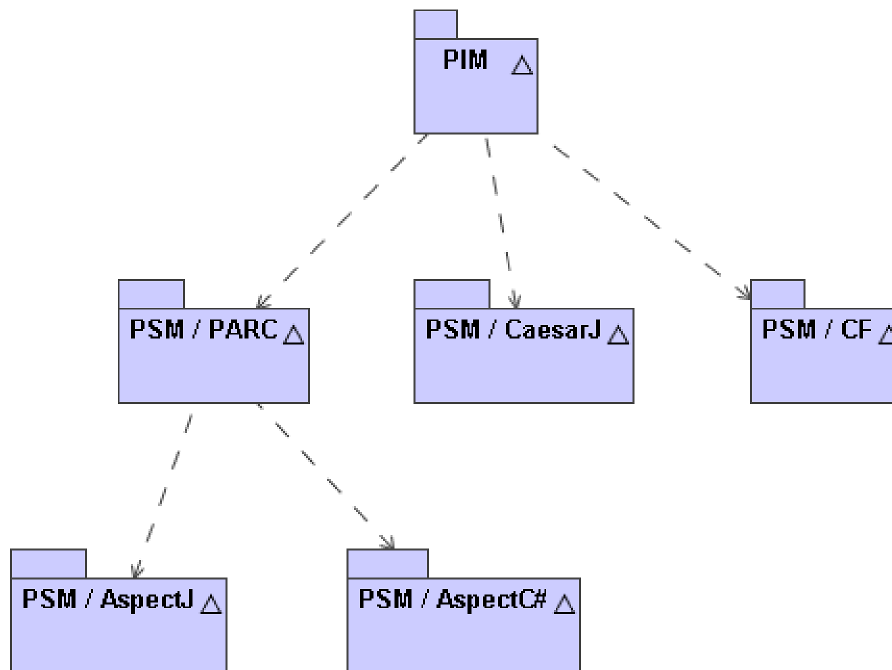
Cieľom príkladu je doplniť do aplikácie notifikáciu BookManagera pri vypožičaní a vrátení výtlačku čo je realizované volaním funkcií borrow a return triedy BookCopy.

Konkrétne PSM modely sú priamou transformáciou z PIM modelu, v prípade použitia viacerých jazykov alebo ich rôznych implementácií založených na rovnakom prístupe by bolo vhodné analyzovať použitie stromu a viacerých stupňov transformácií, čo by umožnilo presnejšie špecifikovanie a ľahšiu zámenu výslednej platformy.



Obrázok 16 - Priama jednoúrovňová transformácia z PIM do PSM

Použitie viacúrovňovej transformácie si vyžaduje identifikovať spoločné a rozdielne vlastnosti jednotlivých jazykov a vytvorenie medzistupňa, ktorý by bol spoločný pre obe cieľové platformy. Tomuto sa môže venovať ďalšia práca.

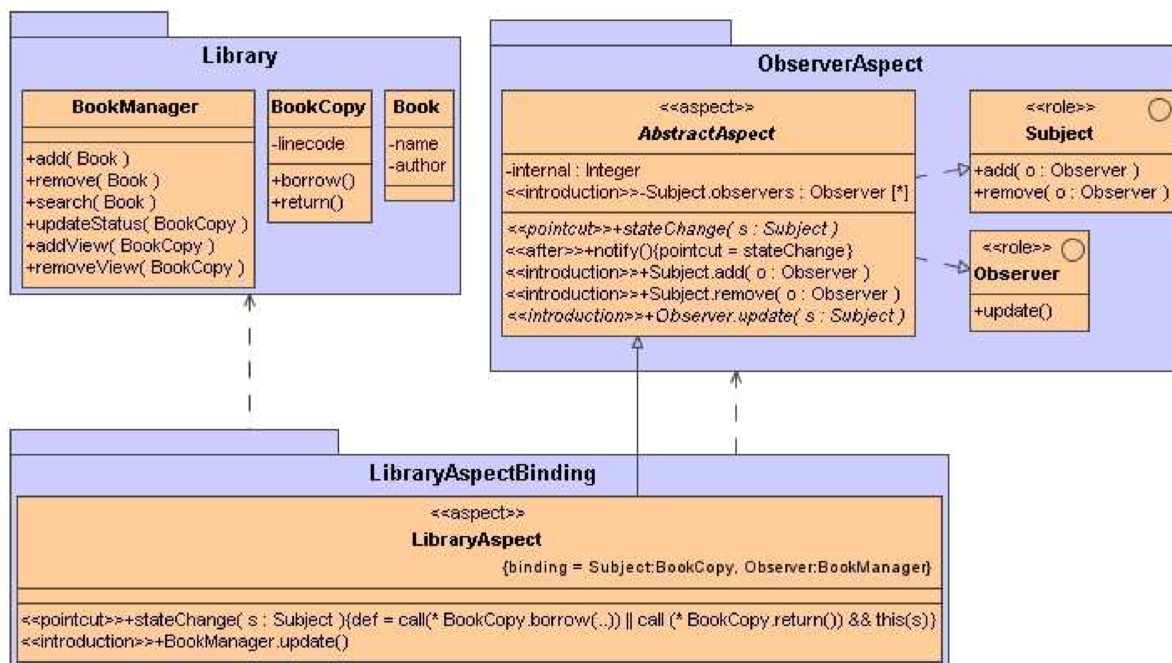


Obrázok 17 - Viacúrovňová transformácia

6.2 PSM model pre platformu AspectJ

AspectJ je úzko spätý s asymetrickým prístupom a je relatívne nízko úrovňový. Je zameraný hlavne na priamu modifikáciu základného programu. Aj keď vďaka dedičnosti spolu s medzitypovými deklaráciami s využitím rozhraní (interface) je možná dostatočná miera abstrakcie a je možné oddeliť implementáciu pretínajúcej záležitosti od jej prepojenia so základným kódom.

Navrhnutý model vychádza čiastočne z AODM [21], ale zatiaľ kvôli jednoduchosti nevyužíva sekvenčné diagramy. Model môže byť rozdelený na implementačnú a spájaciu časť podobne ako pri prístupe UFA [25], ale nie je to podmienkou a implementácia môže byť priamo v spájacej časti. Tento model v skutočnosti nerobí rozdiel medzi oboma časťami.



Obrázok 18 - Observer v AspectJ

Jednotlivé prvky jazyka AspectJ sú v modeli vyjadrené nasledovným spôsobom:

1. Aspekt – je reprezentovaný triedou so stereotypom „aspect“. Aspekty môžu byť abstraktné a umožňujú dedičnosť. Tiež môžu mať vlastné metódy a atribúty.
2. Bodové prierezy – sú definované metódami so stereotypom „pointcut“. Môžu byť abstraktné. Definícia bodového prierezu je rovnaká ako v jazyku AspectJ a je zapísaná v parametri „def“ stereotypu. Každý bodový prierez musí byť pomenovaný, čo je čiastočné obmedzenie oproti jazyku AspectJ kde je možné používať anonymné bodové prierezy. Na funkčnosti sa tým ale nič nemení.
3. Videnia – sú definované metódami so stereotypmi vyjadrujúcimi typ videnia, v profile sú definované tri takéto stereotypy a to „before“, „after“ a „around“. Rovnako ako bodové prierezy musia byť pomenované. Priradenie bodového prierezu je definované parametrom „pointcut“ stereotypu, a môže obsahovať jeden pomenovaný bodový prierez. Pôvodne som zvažoval aj použitie stereotypu „advice“ a uvedenie typu pomocou názvu metódy podobne ako je to v jazyku AspectJ, ale pre lepšiu odlíšiteľnosť pri transformácii som sa rozhodol každé videnie pomenovať. Tiež je možné vďaka tomu predefinovať videnie v odvodenom aspekte rovnako ako bodové prierezy.

4. Medzitypové deklarácie – sú vyjadrené atribútmi a metódami so stereotypom „introduction“ a zmena hierarchie dedičnosti (declare parents) realizáciou rozhraní so stereotypom „role“, konkrétny aspekt definuje priradenie tried týmto roliam pomocou parametra „binding“ stereotypu aspect.

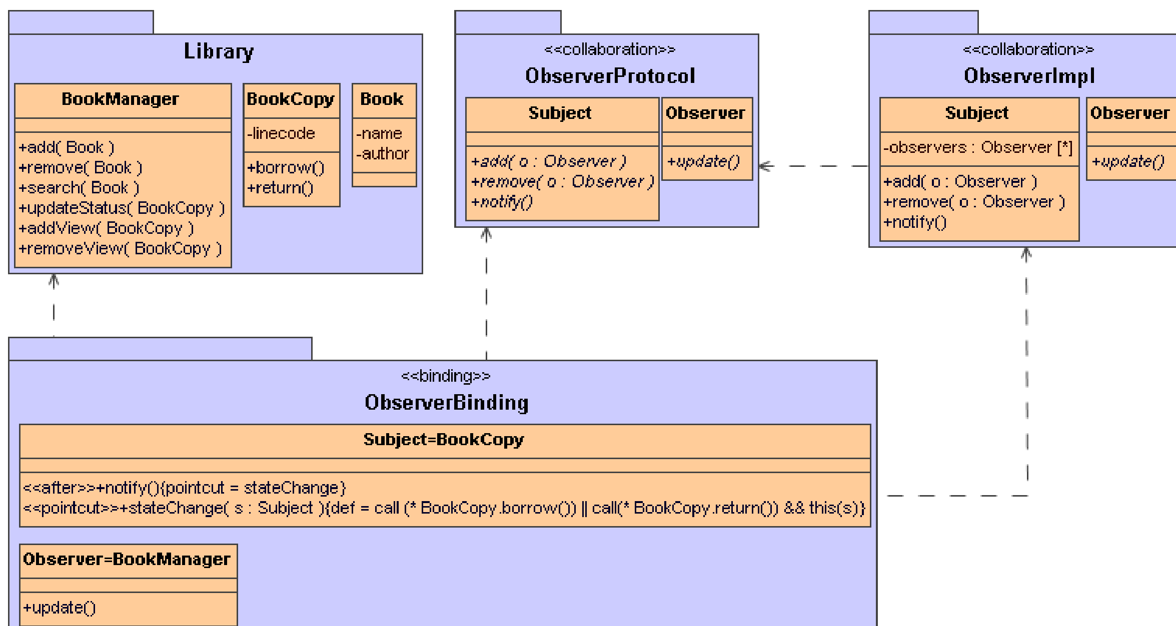
Navyše sa využívajú balíky na zachytenie súvisiacich skupín aspektov, tieto sa ale nijak nemapujú do AspectJ.

Definícia bodových prierezov pomocou syntaxe jazyka AspectJ je jednoduchým obídením problému špecifikácie bodových prierezov pomocou UML. Definície v parametri stereotypu sú vzhľadom na model iba textom, neexistuje tam preto žiadna kontrola či naozaj majú zmysel, to by bolo možné jedine použitím špeciálneho nástroja na modelovanie, ktorý by tieto definície dekódoval a nejakým spôsobom zobrazil ich vzťah k ostatným prvkom modelu, podobne ako je to v prostredí AJDT pre Eclipse. Tiež je nutné aby transformačný algoritmus bol schopný tento text analyzovať a naviazať na objekty v modeli. Aby bolo možné analyzovať vzťahy medzi elementmi v modeli a bodovými prierezmi pomocou existujúcich a štandardných transformačných nástrojov museli by byť definované v maximálnej miere prostriedkami UML. To si ale vyžaduje zložitejšie riešenie napríklad pomocou JPDD, čo ale tiež vyžaduje určitú podporu nástrojov keďže JPDD obsahujú aj špeciálne zápisy, ktoré niesú súčasťou UML.

6.3 PSM model pre platformu CaesarJ

Tento model vychádza z prístupu UFA [25][31], ktorý sa snaží ako komponenty oddeliť aspekty a ich implementáciu od prepojenia s hlavným modulom aplikácie. V [31] je porovnaný tento prístup s kompozičnými vzormi [30] kde konštatujú že oba prístupy sú značne blízke, ale argumentujú že prepojenie medzi aspektom a základným kódom by mal byť definovaný obsiahlejšie ako len reláciou šablónového prepojenia (template binding).

Navrhovaný vzor je upravený aby vyhovoval jazyku UML a tiež aby umožňoval modelovanie čo najviac prvkov jazyka CaesarJ.



Obrázok 19 - Observer v CeaserJ

Jazyk CeasarJ na rozdiel od AspectJ nepodporuje medzitypové deklarácie, podobná funkcionalita je implementovaná pomocou wrapperov spolu s bodovými prierezmi a videniami. Prvky modelu majú nasledovný význam:

1. Collaboration – je v modeli vyjadrená pomocou balíka so stereotypom „collaboration“, ide o vonkajšiu virtuálnu triedu, ktorá obsahuje vnorené virtuálne triedy, reprezentované bežnými triedami v rámci tohto balíka. Dedičnosť je vyjadrená vzťahom závislosti (dependency), pretože UML nepodporuje dedičnosť balíkov. Iná možnosť by bolo použitie vnorených tried, čo by viac korešpondovalo s modelom jazyka CaesarJ, ale väčšie odklonenie od prístupu UFA a ostatných modelov, ktoré sú v tejto práci definované. Ďalším problémom by bola podpora nástrojov, pretože modelovanie vnorených tried nemá rovnako dobrú podporu v modelovacích nástrojoch.

2. Viazanie – (binding) je definované pomocou balíka so stereotypom „binding“. Názvy tried v tomto balíku obsahujú definíciu wrapperov, bodových prierezov a videní.

3. Bodové prierezy – sú definované identicky ako v prípade modelu AspectJ, metódami so stereotypom „pointcut“ pretože oba jazyky používajú rovnakú syntax, CaesarJ ale neposkytuje všetky možnosti bodových prierezov ako AspectJ. Všetky musia byť pomenované, a nie je možné ich dedenie, aj keď v modeli to nie je nijak obmedzené. Ďalšie obmedzenia je možné do vytvoreného profilu doplniť pomocou jazyka OCL.

4. Videnia – CaesarJ tiež podporuje videnia, ktoré sú definované v modeli metódami so stereotypmi definujúcimi typ videnia „before", „after" a „around" tak ako v prípade AspectJ.

Z takéhoto modelu je možné vytvoriť kostru pre program v CaesarJ, vytváranie aspektov ale musí byť urobené programovo. V modeli nie je definované kedy a ako sa majú vytvárať inštancie aspektov.

6.4 PSM model pre Kompozičné filtre

Kompozičné filtre používajú deklaratívny spôsob popisu filtra, čo komplikuje zápis do UML, ktorý je objektový, väčšinu prvkov použitých v CF je ale možné takmer priamo zapísať do elementov triedy, s výnimkou prvkov obsahujúcich zoznamy ako sú definície filtrov (so zoznamom podmienok a pravidiel). Definície filtrov som sa rozhodol reprezentovať pomocou tried, aj keď je to zdĺhavé a vytvorí sa tak veľa tried, je možné použiť rovnaký filter viac krát, a je to jediný prehľadný spôsob.

Kompozičné filtre sa chovajú podobne ako wrappery v ponímaní OOP, rozdiel je ale v odlišnom spôsobe filtrovania správ. Zatiaľ čo wrapper musí použiť reimplementáciu metódy, filter iba deleguje volanie na iný cieľ, či už je to iný objekt alebo iná metóda.

Definícia kompozičného filtra, z ktorej vychádza model vyzerá takto:

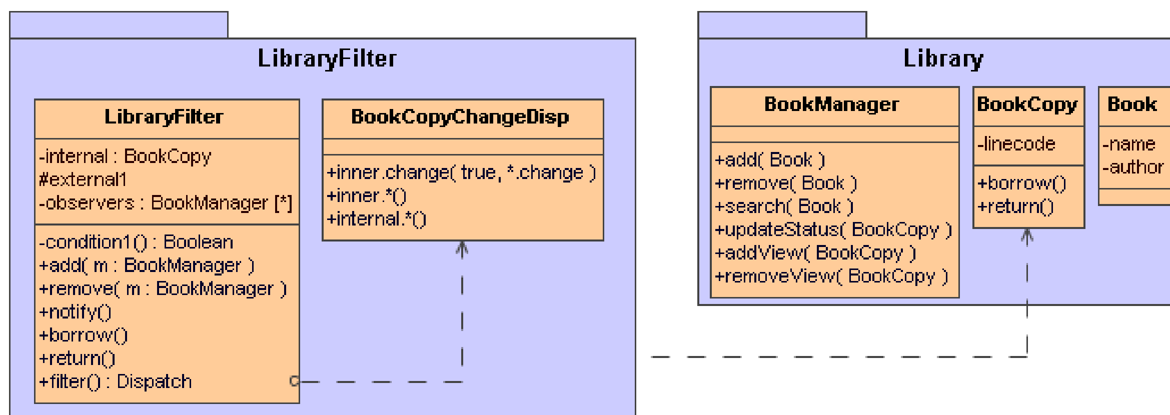
```
concern ProtectedClaimDocument begin
 filterinterface name begin
  internals // internal filtered object object
      document: ClaimDocument;
  externals // shared externals declaration
      documentManager: Manager;
  conditions
      inactiveRH; inactiveRD; inactiveMC; inactiveP;
  methods
      activeTask();
  inputfilters
      name : type = { cond -> [matchPattern]target.method };
 end filterinterface DocumentWithViews;
 implementation in Java
      // implementation of cond. and methods
 end implementation
end concern ProtectedClaimDocument;
```

Ide o syntax jazyka ComposeJ, ale aj ďalšie nové implementácie kompozičných filtrov ako Compose* používajú veľmi podobnú syntax až na minimálne rozdiely spôsobené implementáciou nových funkcií.

V definícii filtrov sú priamo použité referencie na konkrétne triedy a objekty čo nedovoľuje oddelenie definície filtra od konkrétnych objektov na ktoré bude použitý, to isté platí aj o implementácií podmienok a metód filtra.

Novšia špecifikácia kompozičných filtrov (a implementácia jazykom Compose*, ktorý ale ešte nie je dokončený, a ešte ani nemá stabilnú syntax) už umožňuje využitie parametrických definícií, podobne ako generické typy v Jave alebo parametrické klasifikátory v UML. Tiež je možné definovanie filtra prekrývajúceho viacero objektov naraz.

Tento PSM model popisuje kompozičné filtre bez prekrývania (superimposition), takže neumožňuje aplikovať filter súčasne na viaceré objekty. Rovnaký filter by ale mohol byť aplikovaný osobitne na viac objektov s minimálnymi zmenami, na čo by bolo možné využiť parametrické klasifikátory v UML, to je ale už nad rámec implementácie v ComposeJ.



Obrázok 20 - Observer cez Kompozičné filtre

Jednotlivé prvky CF sú vyjadrené v modeli nasledovne:

1. Concern – je vyjadrený balíkom obsahujúcim definíciu filtra, je to vlastne kontajner na prvky potrebné pre definíciu CF. Obsahuje triedu s rovnakým názvom, ktorá ho definuje a viacero tried definujúcich filtre.

2. Internals – ide o interné atribúty filtra, čo je totožné s privátnymi atribútmi triedy.

3. Externals – ide o referencie na vonkajšie objekty, tieto sú inicializované cez parametre konštruktora alebo ide o globálne/statické objekty. V modeli sú reprezentované atribútmi s „protected“ deklaráciou.

4. Conditions – ide o podmienky, sú reprezentované privátnymi metódami bez parametrov vracajúcimi hodnotu true alebo false.
5. Methods – klasické metódy filtra, určené na pomocné operácie sú reprezentované protected metódami hlavnej triedy v balíku.
6. Inputfilters/Outputfilters – definície filtrov sú reprezentované metódou vyjadrujúcou názov a typ filtra s väzbou na osobitnú triedu definujúcu pravidlá pre tento filter. Pravidlá sú reprezentované metódami kde názov metódy určuje cieľ volania, prvý parameter je podmienka a ďalšie parametre sú mapovacie pravidlá. Poradie týchto metód je dôležité pretože určuje poradie aplikácie filtrov. Pretože UML neprikladá poradiu žiadny význam bolo by vhodné nejakým spôsobom dodefinovať určenie poradia filtrov.
7. Implementačná časť - je modelovaná klasicky pomocou štandardného UML ako bežná OOP trieda. Ide iba o definíciu podmienok a metód.

6.5 Návrh PIM AO modelu

Viacero pokusov o definovanie modelovacieho jazyka pre AOSD je popísaných v [17] a autori deklarujú budúce rozšírenie o viacero prístupov a prípadne definíciu transformácií medzi jednotlivými prístupmi.

Aby bol PIM model navrhnutý v tejto práci čo najjednoduchší, prehľadný a mal podporu modelovacích nástrojov nezasahuje do metamodelu UML a je založený na UML profile, a umožňuje iba vysoko úrovňovú definíciu AO záležitostí. Detailnejší popis je doplnený až po transformácii na konkrétny PSM model, takto je možné úplne sa oddeliť od PSM modelov a nie je nutné do tohto abstraktného modelu vnášať prvky špecifickejších modelov. Na jednej strane je takýto PIM model jednoduchý a všeobecný, na strane druhej je ale menej presný a transformácia je možná iba v jednom smere pretože všetky dodatočné informácie doplnené do PSM už nie je možné spätne previesť do PIM, kde nie sú definovateľné. To je ale bežné aj v súčasne dostupných MDA nástrojoch, ktoré iba zriedka podporujú kvalitnú transformáciu oboma smermi, a zvyčajne sa pri nej strácajú niektoré informácie.

Cieľom modelu je poskytnúť hlavne tieto možnosti.

- Medzitypové deklarácie
- Videnia a základné statické bodové prierezy
- Oddelenie logiky aspektu od prepojenia s konkrétnym základnými časťami programu
- Jednoduché prepojenie aspektov so základným programom
- Jednoduchú transformáciu na vytvorené PSM

Obrázok 21 - Všeobecný AO model

Predchádzajúci Obrázok 21 znázorňuje navrhnutý všeobecný AO model na jednoduchom príklade správy kníh. Tento model podporuje medzitypové deklarácie, oddelenie prepojenia aspektu s hlavným programom a jeho logikou, jednoduché prepojenie videní na základnú logiku pomocou relácie závislosti.

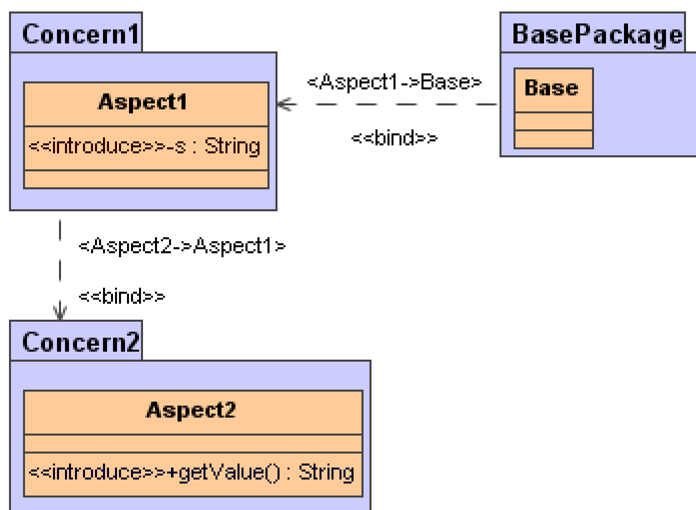
Ide síce o asymetrický prístup, ale iba z pohľadu relácie „bind“, inak sú balíky rovnocenné a je možné na seba naviazať aj dva parametrické balíky s pretínajúcimi záležitosťami. Oproti symetrickému má výhodu že nie je potrebné zložito riešiť konflikty pri spájaní pretože jeden prvok je vždy nadradený druhému.

Medzitypové deklarácie sú vyjadrené použitím parametrického balíka. Balík sa využíva pretože viaceré triedy (role) spoločne tvoria jeden celok, podobne je to aj v prístupe

Theme/UML. Na definovanie prvkov, ktoré sa majú preniesť do naviazaných objektov sa používa stereotyp „introduce“, takto označené elementy budú vložené do naviazanej triedy. Takýmto spôsobom sú riešené štrukturálne zmeny.

V parametrickom balíku vystupujú role ako plnohodnotné triedy, a je možné ich použiť priamo v modeli pretínajúcej záležitosti. V AspectJ PSM je možné pomocou kľúčových slov „declare parents“ možné definovať takéto triedy ako rozhrania a cieľovým triedam definovať že ich implementujú, takýmto spôsobom je možné priamo volať metódu „add“ triedy „Subject“ s parametrom typu „BookManager“. V prípade že PSM takéto definície neumožňuje sa po transformácii za takéto triedy môžu substituovať skutočné triedy podľa parametrov naviazania.

Je ale možné použiť naviazanie aj opačným smerom, ak pri tom nevzniknú konflikty, prípadne cykly. Je možné použiť aj medzitypové deklarácie do balíka obsahujúceho aspekty, ktoré už definujú medzitypové deklarácie do iných tried tak ako to znázorňuje nasledujúci obrázok.



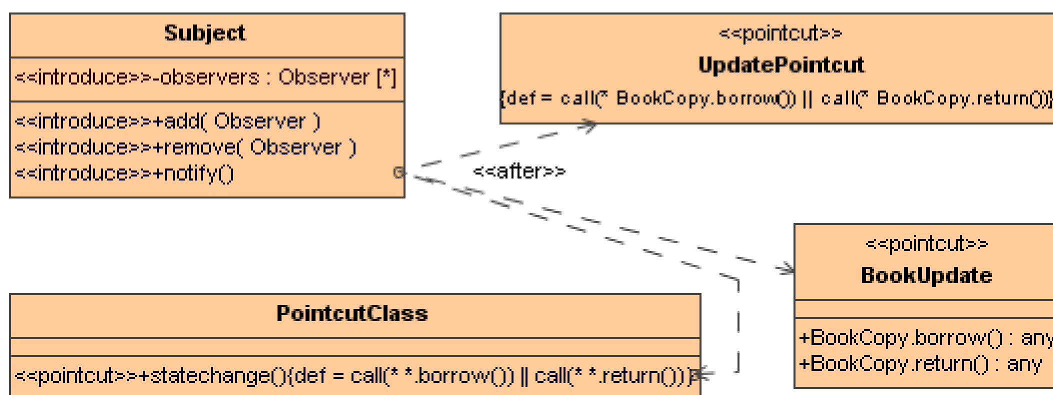
Obrázok 22 - Dvojitá medzitypová deklarácia

V tomto prípade ide o rozšírenie triedy „Aspect1“ o novú metódu „getValue“. V prípade niektorých konkrétnych jazykov to môže byť problém ak nepodporujú medzitypové deklarácie do tried zapuzdrujúcich pretínajúce záležitosti, ale v iných ako napríklad kompozičných filtroch je možné takto definovať vytvorenie filtra nad iným filtrom. V podstate ide o umožnenie aplikovania pretínajúcich záležitostí rovnako na základný kód ako aj na iné pretínajúce záležitosti. Aj preto balíky obsahujúce pretínajúce záležitosti neobsahujú žiadny stereotyp, ktorým by sa odlišovali od ostatných balíkov modelu.

Behaviorálne zásahy do hlavného kódu môžu byť vyjadrené väzbou závislosti so stereotypmi „before“, „after“ a „around“. Cieľom môže byť ľubovoľný element modelu čiže trieda, metóda aj atribúty. Vyjadruje sa ňou akým spôsobom pretínajúca záležitosť modifikuje cieľový element.

Pretože takéto explicitné definovanie každého vzťahu medzi pretínajúcimi záležitosťami a bodmi spájania je prakticky zhodné s klasickou OOP implementáciou a spôsobuje roztrúsenosť týchto väzieb po celom modeli, analyzoval som aj iné možnosti vyjadrenia bodových prierezov. Keďže jazyk UML priamo neposkytuje možnosť zápisu viacerých elementov pomocou jednej relácie, musel som vytvoriť neštandardné konštrukcie, ktoré by sa na to dali použiť.

Jedna možnosť sú JPDD diagramy [29][32], ktoré umožňujú definíciu aj dynamických bodových prierezov. Na transformáciu do konkrétnych PSM sú však príliš zložité, a preto som sa radšej rozhodol navrhnúť vlastné jednoduchšie varianty. V súčasnosti sa najviac v praxi využíva jazyk AspectJ a viaceré ďalšie AO jazyky využívajú rovnaký jazyk na definíciu bodových prierezov, preto sa prirodzene natíska otázka priameho využitia syntaxe AspectJ. Ďalšia možnosť je použitie jednoduchých diagramov tried obsahujúcich zoznam bodových prierezov vo forme čiastočne definovaných atribútov a operácií. Pretože sú založené na diagrame tried a nie sekvenčných a iných dynamických diagramoch, sú použiteľné takmer výlučne na statické bodové prierezy, nevyžadujú ale zložitú podporu nástrojov.



Obrázok 23 - Bodové prierezy

Navrhnuté modely sú vyjadrené na nasledujúcom obrázku Obrázok 23. Trieda „UpdatePointcut“ je priamym mapovaním syntaxe jazyka AspectJ do UML diagramu ako parameter triedy so stereotypom „pointcut“. Trieda „BookUpdate“ zas zobrazuje definíciu

bodového prierezu pomocou čiastočnej definície metód a atribútov. Trieda „PointcutClass" je prevzatá z PSM modelu AspectJ, ide aspekt bez videní, čisto s definíciou bodových prierezov.

Stereotyp „pointcut" je definovaný pre operácie a triedy, aby bolo v UML možné prehľadnejšie použitie rovnakého bodového prierezu z viacerých videní. Navyše je takto oddelený od definície samotného aspektu, ktorý nemusí obsahovať žiadny odkaz na konkrétne objekty modelu. Ďalšia výhoda použitia triedy je možnosť vyjadriť dedičnosť bodových prierezov štandardným spôsobom.

Druhý spôsob s použitím čiastočnej definície operácií a atribútov sprístupňuje viac informácií v štandardnom UML pre nástroje na transformáciu do PSM. Dovoľuje definovať bodové prierezy ako množinu viacerých operácií a atribútov podobným spôsobom ako syntax AspectJ. V definícii je možné v názvoch použiť ako všeobecný klasifikátor znak *, a v definícii typu typ AOP::any z UML profilu (napríklad *.get*(any) čo znamená akúkoľvek metódu začínajúcu na get s jedným parametrom ľubovoľného typu). Tiež je možné v definícii parametrov operácií použiť názov „.." (dve bodky, s rovnakým významom ako v prípade AspectJ).

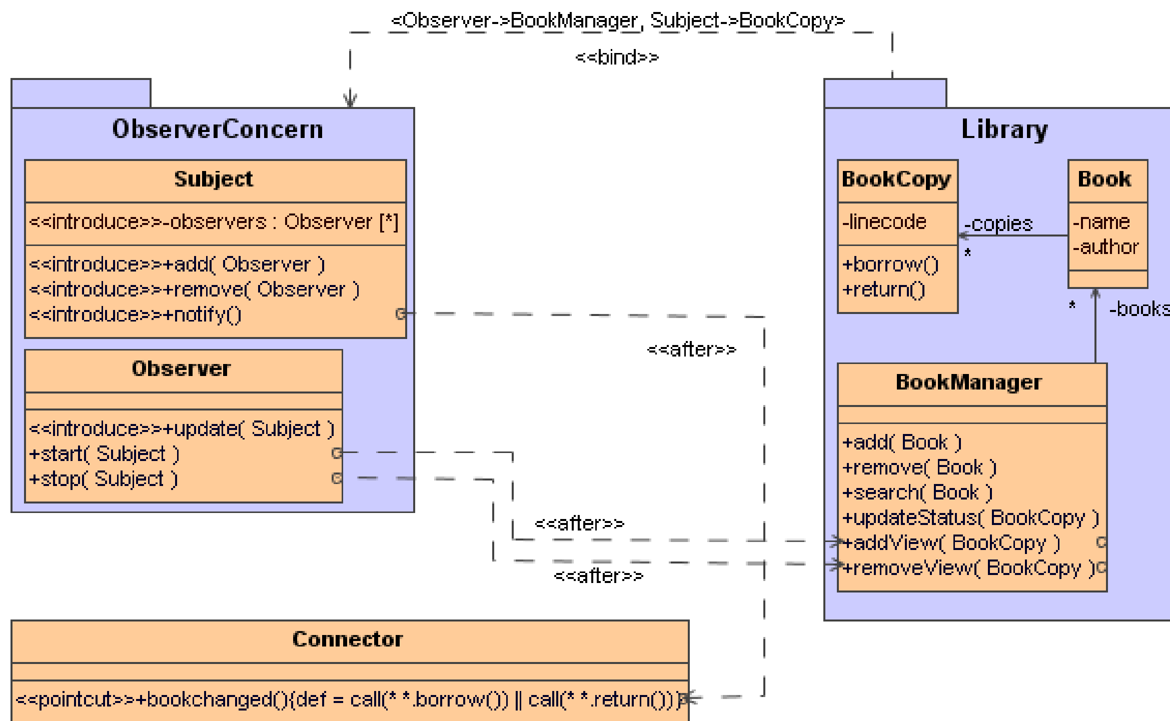
Oba prístupy vyžadujú podporu nástrojov pri modelovaní a transformácii na konkrétne PSM. Najlepšie možnosti poskytuje prístup vyjadrený „PointcutClass", ktorý poskytuje dedičnosť, možnosť abstraktných bodových prierezov a jednoduché mapovanie do väčšiny jazykov podporujúcich syntax AspectJ. V prípade že cieľový PSM nepodporuje túto syntax je nutné vytvoriť nástroj, ktorý pretransformuje tieto definície (napríklad až na jednoduché jednotlivé závislosti ako ukazuje Obrázok 21)

Takýto model je veľmi jednoduchý a pritom umožňuje generovať kostru špecifických modelov, ktoré sú po doplnení dostatočne presné. Určité spresnenie tohto modelu by bolo možné zmenou definície pretínajúcich záležitostí, pretože v takejto podobe sú ekvivalentné roztrúsenému kódu v prípade pretínajúcich záležitostí v bežnom OOP programe. Pre každé miesto v programe, ktoré je pretínané je nutné vytvoriť jednu reláciu, čo je presne problém ktorý sa AOP snaží odstrániť.

V závere [32] je porovnaných niekoľko spôsobov zápisu pretínajúcich záležitostí v UML, ich výhody a nevýhody. Prístup v tejto práci s využitím relácie závislosti má rovnaké nevýhody ako samotné OOP výsledkom čoho je množstvo roztrúsených relácií. Takýto prístup ale na druhej strane veľmi dobre graficky identifikuje miesta kde nastáva pretínanie. V kombinácii

s aspektmi obsahujúcimi bodové prierezy podľa syntaxe AspectJ PSM, je možné tieto závislosti zoskupiť a model tak sprehľadniť.

V pokračovaní práce by bolo vhodné zvážiť možnosť použit JPDD v konkrétnych PSM aj v PIM modeli čím by sa odstránila nutnosť transformácie a pokryli by sa aj dynamické bodové prierezy.



Obrázok 24 - Kompletný PIM model na príklade

6.6 Návrh transformácií z PIM do PSM

Transformácie zo všeobecného modelu budú realizované priamo do PSM konkrétneho jazyka, pretože zatiaľ neboli definované žiadne medzistupne. Konštrukcie, ktoré PSM neposkytuje by mali byť nahradené ekvivalentnou implementáciou funkčnosti pomocou iných prostriedkov ak to bude možné. Ak to možné nebude transformácia bude neúspešná a v prípade implementácie nástrojov na transformáciu by tieto mali vyhlásiť chybu a dôvod zlyhania, tak aby bolo možné upraviť zdrojový PIM model.

Navrhované transformácie sú vo forme slovne popísaných krokov, potrebných na vytvorenie PSM zo zdrojového PIM. Transformácie sú jednosmerné a po ich ukončení môže byť

vytvorený nekompletný PSM model, ktorý je potrebné došpecifikovať (podobne ako je to u mnohých súčasných nástrojov na generovanie kódu z UML modelov). Ak by sa mal transformovať PIM do kompletného PSM, musel by už PIM umožňovať všetky konštrukcie, ktoré sú potrebné pre kompletný popis všetkých PSM do ktorých sa bude transformovať. Vzhľadom na značne rozdielne AO jazyky by tento model bol následne príliš zložitý. Ak by aj PIM umožňoval detailnú špecifikáciu, pri transformácii by došlo k problému čo s definíciami, ktoré cieľový PSM nepodporuje.

Na vyjadrenie transformácií bude potrebné vytvoriť príklady, aby bolo zrejmé akým spôsobom sa budú transformovať jednotlivé prvky modelu.

6.6.1 Transformované prvky PIM

Niektoré prvky PIM modelu, ktoré majú nejaký špeciálny význam z pohľadu AO modelu sa transformujú na zodpovedajúce prvky PSM modelov podľa ich významu. Ostatné štandardné prvky ako sekvenčné diagramy metód aspektov alebo klasické atribúty tried sa prenášajú v nezmenenej podobe. Tieto špeciálne prvky PIM modelu, ktoré majú zvláštny význam sú uvedené nižšie.

- Parametrický balík – vyjadruje že sa v ňom nachádzajú triedy, ktoré obsahujú medzitypové deklarácie, tieto triedy majú rovnaký názov ako parametre.
- Atribúty a operácie so stereotypom „introduction“ – vyjadrujú že ide o medzitypové deklarácie, s cieľovými triedami podľa parametrov balíka.
- Závislosti so stereotypmi before, after a around – ide o priradenie videnia konkrétnemu bodovému prierezu, budem ich označovať ako stereotypy naviazania.
- Metóda so stereotypom „pointcut“ – ide o bodový prierez ako v prípade AspectJ

6.6.2 Transformácia do AspectJ PSM

AspectJ PSM okrem spôsobu inštanciácie aspektov a samotného kódu jednotlivých metód pokrýva všetky možnosti jazyka AspectJ, navyše definuje názvy videniam čo v tomto jazyku nie je možné. Takéto názvy môžu slúžiť napríklad ako javadoc komentár.

Pri transformácii z PIM sa môžu závislosti so stereotypom naviazania pretransformovať na videnie a vytvorí sa nový bodový prierez podľa názvu cieľového elementu. Definícia bodového prierezu bude call alebo execution ak ich cieľom je metóda, get alebo set ak ich

cieľom je atribút, call(...new()) ak ich cieľom je trieda, nezmenený bodový prierez ak cieľ má stereotyp „pointcut". Aby bolo možné definovať presne o aký druh bodového prierezu ide musel by byť všeobecný model špecifickejší a čiastočne aj obmedzenejší. Konkrétny typ bodového prierezu si môže aspekt zistiť aj cez špeciálny objekt thisJoinPoint. Metódy so stereotypom „introduce" sa vyjmú do rozhrania a zároveň ich implementácia ostane v aspekte, ktorý bude implementovať toto rozhranie. Parametre balíka sa použijú na deklarovanie implementácie rozhraní cieľovou triedou.

V postupných krokoch by sa dala takáto transformácia z PIM do AspectJ PSM definovať nasledovne

1. Do PSM sa skopírujú všetky balíky z PIM
2. Každá trieda sa skopíruje do zodpovedajúceho balíka a
3. Ak ide o triedu s názvom parametra v balíku, trieda sa skopíruje so stereotypom „aspect" a príponou Aspect (napr. ObserverAspect), vytvorí sa aj nové rozhranie s názvom triedy. Rozhranie má rovnaký názov preto aby bolo nebolo potrebné meniť aj typy parametrov jednotlivých metód. Do tohto rozhrania sa prenesú všetky metódy so stereotypom „introduce". Pre každú cieľovú triedu v hodnote parametra balíku sa vytvoria v aspekte atribúty a metódy so stereotypom „introduction", a definuje sa parameter stereotypu binding pre cieľové triedy.
4. Ak sa v triede nachádza aspoň jedna operácia, ktorá je začiatkom závislosti so stereotypom naviazania (6.6.1) triede sa definuje stereotyp „aspect" a danej operácii príslušný stereotyp podľa typu videnia. Pre každú takúto závislosť začínajúcu v triede sa vytvorí bodový prierez podľa názvu cieľového elementu a vyššie popísaných pravidiel. Ak ide o AspectJ bodový prierez reprezentovaný metódou stereotypu „pointcut" použije sa jej názov a definícia.

Pretože vo vytvorenom UML profile pre AspectJ PSM a ani PIM nie sú definované žiadne obmedzenia použitých stereotypov, je možné vytvoriť model, ktorý nebude možné pretransformovať, dokonca ani nemusí mať vôbec zmysel. Preto by bolo vhodné vytvoriť pomocou jazyka OCL vytvoriť obmedzenia, ktoré by zabezpečili iba povolené konštrukcie v rámci modelov.

6.6.3 Transformácia do CaesarJ PSM

Template binding a závislosti sa môžu pretransformovať do „binding" balíka ak existujú nejaké závislosti naviazania. Podľa template binding sa určí ktoré triedy budú wrappery. Vytvorí sa balík, ktorý bude obsahovať iba rozhranie so špecifikovanými metódami, ďalší balík bude obsahovať implementáciu metód a atribúty.

Tento PSM obsahuje špeciálne prvky ako sú „collaboration" a „binding" balíky. Tiež obsahuje definície bodových prierezov a videní ako AspectJ PSM.

Transformácia sa dá popísať nasledujúcimi krokmi.

1. Každý balík bez parametrov sa skopíruje bez zmeny
2. Parametrický balík sa skopíruje a nastaví sa mu stereotyp „collaboration", vytvorí sa aj jeho kópia s príponou „Intf", obsahujúca všetky triedy a metódy pôvodného balíka, ale budú definované ako abstraktné. Tiež sa vytvorí nový balík s príponou „Binding" a stereotypom „binding". Implementácia metód ostane v balíku s pôvodným názvom spolu s atribútmi. Medzi balíkom s implementáciou metód a tým abstraktným sa vytvorí relácia závislosti, od pôvodného k abstraktnému. Podľa parametrov balíka v PIM sa definujú triedy v balíku s príponou „Binding".
3. Ak z balíka vychádzajú nejaké závislosti naviazania, vytvorí sa nový balík s príponou „Binding" ak ešte neexistuje z predchádzajúceho kroku. Pre každú takúto závislosť sa vytvorí v príslušnej triede bodový prierez rovnako ako v prípade AspectJ PSM. Tiež sa vytvorí metóda so stereotypom podľa typu videnia a jej meta-atribút pointcut sa nastaví na vytvorený bodový prierez.

V tomto PSM môže nastať situácia keď model nebude obsahovať žiadne základné Java triedy ale iba virtuálne triedy v „collaboration" balíkoch. Toto sa dá v PIM namodelovať tak, že sa nedefinujú žiadne bodové prierezy ani medzitypové deklarácie, ale iba obojsmerná „bind" závislosť medzi balíkmi. V tomto prípade síce vzniknú dva abstraktné balíky s rozhraním virtuálnych tried, čo ale nie je problém, pretože by mali byť oba rovnaké.

6.6.4 Transformácia do CF PSM

Pretože CF podporujú v implementácii ComposeJ definovať filter iba nad jednou triedou, je nutné vytvárať filter nad každou triedou, ktorá je cieľom závislosti naviazania. Našťastie je možné vrstviť filtre na seba, takže to nie je neprekonateľný problém. Pre každú triedu v parametrickom balíku sa vytvorí samostatný CF nad triedou určenou cez template binding.

V krokoch by transformácia mala pracovať nasledovne

1. Ak PIM obsahuje definície bodových prierezov, tieto sa musia redukovať na základné bodové prierezy špecifikované závislosťami. Transformovať je možné iba definície call a execution, pretože CF dovoľuje odchytávať iba posielanie správ objektom, čiže bodové prierezy definované na metódach (ale vrátane konštruktora).
2. Balíky sa skopírujú
3. Každá trieda v neparametrickom balíku, z ktorej nevychádza žiadna závislosť naviazania sa skopíruje v nezmenenej podobe.
4. Ak je trieda definovaná ako parameter balíka, vytvoria sa triedy pre filtre nad každou triedou, ktorá je špecifikovaná v hodnote parametra. V každom filtri sa definuje internal s typom cieľovej triedy. Všetky operácie a atribúty so stereotypom „introduce“ sa transformujú na public elementy filtra, tie bez stereotypu budú private.
5. Pre každú závislosť naviazania sa vytvorí trieda definujúca pravidlá filtra, tak že najprv sa spustia interné implementácie „inner.*()“ a potom sa delegujú do interného objektu „internal.*()“. Samotné vykonanie pôvodnej operácie podľa typu videnia sa definuje v tele internej definície metódy, a to pred, za alebo vôbec.

V prípade parametrického balíka môže nastať situácia že hodnota parametra obsahuje viac ako jednu triedu, v tomto prípade pre kompozičné filtre nastáva problém, pretože nie je možné, aspoň nie v implementácii ComposeJ, filter aplikovať na viac tried. Navrhnuté transformácie takýto prípad zatiaľ neberú do úvahy a vyprodukujú v tom prípade zlý výstup.

7 Zhodnotenie

Zo začiatku sa práca zameriavala hlavne na porovnanie najpoužívanejších alebo inak zaujímavých aspektovo orientovaných jazykov. Analyzuje ich základné črty a možnosti. Pre prístup Model Driven Architecture sú veľmi dôležité možnosti aspektovo orientovaného modelovania programov, ktoré je nevyhnutným predpokladom k využitiu MDA. V práci sú predstavené niektoré najčastejšie používané spôsoby modelovania aspektovo orientovaných programov.

V práci je navrhnutý minimálny systém modelov, ktoré sa dajú použiť na modelovanie jednoduchých problémov s využitím aspektov. Tiež obsahuje návrh všeobecného modelu, ktorý pokrýva základné požiadavky a funkcionalitu niekoľkých aspektovo orientovaných jazykov, spolu s neformálnou definíciou transformácií z tohto modelu do modelov konkrétnych jazykov. Pretože ide iba o jednoduché modely, na overenie použiteľnosti MDA, pri reálnych problémoch by určite nastali situácie keď navrhnuté modely neposkytujú dostatočné možnosti, preto by si ďalšia práca vyžadovala overiť tieto modely aj na väčších problémoch, prípadne implementovať pomocné nástroje na transformáciu takýchto modelov.

Momentálne prebieha intenzívny výskum v oblasti aspektovo orientovaného návrhu softvéru, takže v tejto oblasti sa dajú očakávať výrazné zmeny, aj kvôli tomu že doposiaľ nebolo predstavené komplexné všeobecné riešenie aspektovo orientovaného návrhu a analýzy. Pre objektovo orientovaný prístup už existuje mnoho nástrojov a štandardov ako napríklad UML (Unified Modelling Language), RUP (Rational Unified Process) a podobne. Rovnako existuje mnoho nástrojov na generovanie kódu z modelov, hlavne pre oblasť OOP jazykov a objektovo-relačného mapovania. Pre oblasť aspektovo orientovaných systémov žiaľ zatiaľ neexistuje žiadny unifikovaný proces ani nástroje na analýzu, špecifikáciu a návrh systémov, ktorý by bol nezávislý na platforme.

Pretože je ale v tejto oblasti veľa zmien a nových prístupov a zároveň reálne využitie veľmi malé, je veľmi nepravdepodobné že by sa akýkoľvek súčasný model aspektovo orientovaného vývoja presadil. Určite bude trvať dlhšie kým sa tieto prístupy dostatočne rozšíria a štandardizujú, tak aby bolo možné zmysluplne sa venovať ich použitiu v MDA. Prvým krokom je práve vytvorenie jednotnej definície aspektovo orientovanej paradigmy programovania spolu s nejakým jazykom na jej špecifikáciu, tak ako to bolo v prípade objektovo orientovaného programovania a jazyka UML. Až potom má zmysel venovať sa ďalším odvíjajúcim sa krokom.

8 Literatúra

[1] A.Brown, J.Conallen: http://www-128.ibm.com/developerworks/rational/library/may05/brown/brown.html, IBM, 2005

[2] The AspectJ Project: http://eclipse.org/aspectj, October 2005

[3] CaesarJ project documentation: http://www.caesarj.org, October 2005

[4] Brichau J, Haup M.: Survey of Aspect Languages and Execution Models, Technical report of AOSD Europe, 2005.

[5] Tarr P., Ossher H.: HyperJ user and installation manual, IBM Corporation, 2000

[6] Bergmans, L., Aksit, M.: Principles and Design Rationale of Composition Filters. In R. Filman, T. Elrad, S.C.M.A., ed.: Aspect-Oriented Software Development. Addison-Wesley, 2004

[7] Lieberherr K.: Adaptive object-oriented software, The Demeter method, Northeastern university Boston, 1996

[8] K.Kiviluoma, J.Peltoniemi: Paradigms of concern separation, 2004

[9] A.Reina, J.Torres, M.Toro: Separating concerns by means of UML-profiles and metamodels in PIMs. In Proc. of the 5th Aspect-Oriented Modeling Workshop, October 2004.

[10] Robert E. Filman, Tzilla Elrad, Siobhian Clarke, and Mehmet Akssit: Aspect-Oriented Software Development. Addison-Wesley, Boston, 2005.

[11] Andrew Jackson, Siobhán Clarke: Initial Version of Aspect-Oriented Design Approach, Trinity College Dublin, Dublin, AOSD-Europe Deliverable D38, AOSD-Europe-TCD-7, 15 February 2006

[12] Siobhian Clarke, Robert J. Walker: Generic aspect-oriented design with Theme/UML. In Filman et al. [10], pages 425-458.

[13] Tzilla Elrad, Omar Aldawud, and Atef Bader: Expressing aspects using UML behavioral and structural diagrams. In Filman et al. [10], pages 459-478.

[14] Hyper/J project, http://www.research.ibm.com/hyperspace/HyperJ/HyperJ.htm, May 2006

[15] O.Aldawud, T. Elrad, A. Bader: UML profile for aspect-oriented software development. In Proceedings of the AOM workshop at AOSD, 2003

[16] O. Aldawud, M. Kand´e, G. Booch, B. Harrison, and D. Stein, editors. Third International Workshop on Aspect Oriented Modeling, Mar. 2003.

[17] A. Schauerhuber, W. Schwinger, W. Retschitzegger, and M. Wimmer: A Survey on Aspect-Oriented Modeling Approaches, AOSD-Europe, 2006

[18] C. Chavez, C. Lucena: A Theory of Aspects for Aspect-Oriented Software Development. In Proc. of the 7th Brazilian Symposium on Software Engineering, 2003.

[19] R. Chitchyan, A.Rashid, P. Sawyer, et al.: Survey of aspect-oriented analysis and design approaches. Technical Report AOSD-Europe-ULANC-9, AOSD-Europe, May 2005.

[20] S.Clarke, E.Baniassad: Aspect-Oriented Analysis and Design: The Theme Approach, Addison Wesley, March 23, 2005, ISBN: 0-321-24674-8

[21] D.Stein, S.Hanenberg, R.Unland: An UML-based Aspect-Oriented Design Notation For AspectJ, In Conference Proceedings on AOSD, 2004

[22] Han, Y., Kniesel G., Cremers A.: Towards Visual AspectJ by a Meta Model and Modeling Notation, 6th International Workshop on Aspect-Oriented Modeling at the 4th International Conference on Aspect-Oriented Software Development, Chicago, 2004

[23] O. Hachani: Extending UML meta-model for Hyper/J: HyperJ/UML, LSR-IMAG

[24] Object Management Group, www.omg.org, 2007

[25] openArchitectureWare, www.openarchitectureware.org, 2007

[26] Groher, I., Bleicher, S., Schwanninger, C.: Model-driven development for pluggable collaborations. In: 7th International Workshop on Aspect-Oriented Modeling, Montego Bay, Jamaica, Oct. 2nd, 2005

[27] A. M. Reina, J. Torres: Weaving AspectJ aspects by means of transformations, at 19th European Conference on Object-Oriented Programming (ECOOP), 2005.

[28] L. Fuentes: Elaborating UML 2.0 Profiles for AO Design. In Proc. of the 8th International Workshop on Aspect-Oriented Modeling (AOSD'06), Bonn, Germany, March 2006.

[29] Stein, D.; Hanenberg, S.; Unland, R.: On Representing Join Points in the UML; 2nd International Workshop on Aspect-Oriented Modeling with UML, UML 2002, Dresden, Germany, September 30, 2002

[30] S. Clarke, R. J. Walker: Composition patterns: An approach to designing reusable aspects. In Proceedings of the 23rd International Conference on Software Engineering, Toronto, Canada, 12--19 May 2001.

[31] S. Herrmann: Composable designs with UFA. In Workshop on Aspect-Oriented Modeling with UML at 1 st Intl. Conference on Aspect Oriented Software Development, 2002.

[32] D. Stein, S. Hanenberg, R. Unland: Expressing different conceptual models of join point selections in aspect-oriented design. In Proceedings of the 5th international conference on Aspect-oriented software development, Bonn, Germany, March 2006

[33] Joerg Evermann: A Meta-Level Specification and Profile for AspectJ in UML. In Proceedings of the 10th international workshop on Aspect-oriented modeling 2007, Vancouver, Canada, March, 2007

[34] Object Management Group (www.omg.org), Object Constraint Language Specification, version 2.0, 2007

[35] J.C. Wichman, ComposeJ: The Development of a Preprocessor to Facilitate Composition Filters in the Java Language, MSc. thesis, Dept. of Computer Science, University of Twente, December 1999.

[36] TRESE, Composition Filters & Compose* (composestar.sf.net), University of Twente, 2007

[37] Andrew Jackson, Siobhán Clarke: Towards the Integration of Theme/UML and JPDDs, in Workshop on Aspect-Oriented Modelling at AOSD, 2006

9 Prílohy

A. Elektronické médium