Technická univerzita v Košiciach
Fakulta elektrotechniky a informatiky

Kompozitné anotácie a aspektovo-orientované programovanie

Diplomová práca

2015 **Matej Palfi**

Technická univerzita v Košiciach
Fakulta elektrotechniky a informatiky

Kompozitné anotácie a aspektovo-orientované programovanie

Diplomová práca

Študijný program:	Informatika
Študijný odbor:	9.2.1 Informatika
Školiace pracovisko:	Katedra počítačov a informatiky (KPI)
Školiteľ:	doc. Ing. Jaroslav Porubän, PhD.
Konzultant:	Ing. Milan Nosáľ

Košice 2015 **Matej Palfi**

Abstrakt v SJ

V jazyku Java dochádza často k využívaniu anotácií a skupín anotácií, ktoré pri opakovanom použití môžu zneprehľadňovať zdrojový kód a spomaľovať jeho písanie. Takto duplikované skupiny anotácii vyvolávajú otázku, či neexistuje mechanizmus, ako skupiny anotácií opísať niečim kratším, napríklad jedným kľúčovým slovom, ktoré by opisovalo túto skupinu anotácií a bolo plnohodnotne považované za skrátený zápis tejto skupiny anotácií. Anotáciu predstavujúcu toto kľúčové slovo nazývame kompozitnou anotáciou. Táto práca sa zaoberá analýzou problematiky opakujúcich sa skupín anotácií prostredníctvom experimentálneho overenia ich výskytu v praxi. Zároveň práca uvádza možné nové riešenie tohto problému. Novým riešením, ktoré táto práca ponúka, je využitie aspektovo orientovaného programovania za účelom tvorby kompozitných anotácií.

Kľúčové slová

anotácia, aspektovo orientované programovanie, aspekt, kompozitná anotácia, anotačný vzor, AspectJ

Abstrakt v AJ

When programming in Java programming language, annotations and groups of annotations are vastly used. If those annotation groups are used repeatedly, the code writing process can be significantly slowed and worse, source codes become highly disarranged. The problem rises a question whether a mechanism to describe an annotation group with something shorter exists, a keyword for instance, that would fully take place for the annotation group and would be recognized as the group's substitution. Such keyword would be addressed as a composite annotation. The main contribution of this thesis is the experimental study proving existence of repeating (duplicated) groups of annotations. This thesis also tries to offer a new possible solution to the problem. The new way of dealing with composite annotations that this thesis offers is based on usage of aspect oriented programming.

Kľúčové slová v AJ

annotation, aspect-oriented programming, aspect, composite annotation, annotation pattern, AspectJ

TECHNICKÁ UNIVERZITA V KOŠICIACH

FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Katedra počítačov a informatiky

ZADANIE DIPLOMOVEJ PRÁCE

Študijný odbor: **9.2.1 Informatika**

Študijný program: **Informatika**

Názov práce:

Kompozitné anotácie a aspektovo-orientované programovanie

Composite Annotations and Aspect-oriented Programming

Študent:	**Bc. Matej Palfi**
Školiteľ:	**doc. Ing. Jaroslav Porubän, PhD.**
Školiace pracovisko:	**Katedra počítačov a informatiky**
Konzultant práce:	**Ing. Milan Nosáľ**
Pracovisko konzultanta:	**Katedra počítačov a informatiky**

Pokyny na vypracovanie diplomovej práce:

1. Analyzovať problematiku kompozitných anotácií a ich významu.
2. Pripraviť štúdiu, ktorá by analýzou sady zdrojových kódov vybraných open-source projektov preukázala potenciál kompozitných anotácií.
3. Realizovať pripravenú štúdiu.
4. Analyzovať a navrhnúť prístup ku kompozitným anotáciám prostredníctvom aspektovo-orientovaného programovania.
5. Overiť vhodnosť riešenia porovnaním s existujúcimi prístupmi.

Jazyk, v ktorom sa práca vypracuje: slovenský

Termín pre odovzdanie práce: 30.04.2015

Dátum zadania diplomovej práce: 31.10.2014

doc. Ing. Jaroslav Porubän, PhD.
vedúci garantujúceho pracoviska

TECHNICKÁ UNIVERZITA V KOŠICIACH FAKULTA ELEKTROTECHNIKY A INFORMATIKY -13-

prof. Ing. Liberios Vokorokos, PhD.
dekan fakulty

Čestné vyhlásenie

Vyhlasujem, že som diplomovú prácu vypracoval samostatne s použitím uvedenej odbornej literatúry.

Košice 30. 4. 2015 .....

Vlastnoručný podpis

Poďakovanie

Ďakujem svojmu konzultantovi, Milanovi Nosáľovi za jeho odborné vedenie a cenné rady, ktoré mi poskytoval počas celej doby písania tejto diplomovej práce.

Obsah

Zoznam obrázkov

Slovník termínov

Programový element jeden zo základných prvkov programovacieho jazyka, patrí sem medzi inými trieda, metóda, atribút, rozhranie a podobne

Anotácia metaúdaj, deklarovaný nad programovým elementom

Anotačný vzor presne definovaná množina anotácií, tvoriacich jeden opakujúci sa celok

Kompozitná anotácia abstrakcia anotačného vzoru do jedného kľúčového slova; naberá význam rovný súčtu významov všetkých anotácií z ktorých je tvorená

Úvod

V informatike ako vednej disciplíne došlo za posledných niekoľko desaťročí k obrovskému pokroku. Zatiaľ čo začínala ako oblasť záujmu určitej malej skupiny ľudí, najmä vedcov a nadšencov, v súčasnosti dochádza k využívaniu niektorého z jej postupov v značnom množstve oblastí, s ktorými ešte pred niekoľkými rokmi nemusela mať nič spoločné. Je preto veľmi dôležité, aby sa informatika približovala daným oblastiam, či už pre účely zjednodušenia chápania profesionálmi danej oblasti - domény (ktorí nemusia byť nutne informatikmi), alebo z dôvodu vernejšieho (aj v prípade len čiastočného) splynutia s danou oblasťou pre zvýšenie efektivity aplikácie potrebných metód informatiky.

Na účely priblíženia informatiky k potrebnej doméne je možné využiť napríklad anotácie (označenie pochádza z jazyka Java, ekvivalentom z jazyka C# je slovo atribúty), ktorými je možné opísať objekty domény. Niekedy je potrebné jednému objektu priradiť viacero opisov. Vzniká tak otázka: "Nie je v prípade opakujúceho sa objektu s rovnakými anotáciami vhodnejšie zlúčiť tieto anotácie a na opis tohto opakujúceho sa objektu využiť získanú zlúčenú anotáciu?". Týmto sa problém podobá na problém duplikovaného kódu, keď je viacero duplikátov nahradených volaním metódy, ktorá tento kód abstrahuje. Tento problém môžu riešiť takzvané "kompozitné anotácie".

Téma kompozitných anotácií nie je novou témou. Existuje niekoľko publikácií, ktoré sa tejto téme venujú. Jedna z takýchto publikácií s názvom Daileon: A Tool for Enabling Domain Annotations [1] opisuje nástroj, rozhranie v jazyku Java nazvané Daileon. Hlavná myšlienka tejto publikácie sa opiera o takzvané "doménové anotácie" - anotácie, ktoré sa viažu na určitú doménu (oblasť). Daileon s pomocou bajtovej manipulácie zdrojového kódu dokáže takéto doménové anotácie opísať inými anotáciami, teda vytvára akúsi kompozíciu anotácií - kompozitnú anotáciu, sémanticky ekvivalentnú s danou doménovou anotáciou.

Našou snahou v tejto práci bude dokázanie, že v zdrojových kódoch programov sa nachádzajú skupiny anotácií, ktoré sa opakujú, čím by sa potvrdil význam riešenia kompozitných anotácií. Napriek tomu, že už existujú práce, ktoré sa snažia tento problém riešiť, doteraz neexistovalo experimentálne overenie toho, že k takýmto duplikátom anotácií dochádza aj v praxi. O to sa pokúsime pomocou analýzy vybranej vzorky zdrojových kódov z voľne dostupných repozitárov. Za účelom automatizácie testovania si pomôžeme vytvorením anotačého procesora, ktorý bude zdrojové kódy analyzovať. Analýzu následne vyhodnotíme, na základe jej výsledkov vyvodíme závery.

Následne navrhneme možné riešenie problému kompozitných anotácií s pomocou využitia aspektovo orientovaného programovania (ďalej ako AOP) ako náhrady bytovej manipulácie využitej v už existujúcom nástroji Daileon. Začneme opisom základných postupov pri využití AOP za účelom realizácie kompozitných anotácií, načrtneme konkrétny príklad na ktorom tieto postupy demonštrujeme a určíme hranice možností čo je a čo nie je možné s pomocou AOP v tomto smere zrealizovať. Nosným jazykom, ktorý v tejto práci využijeme, bude jazyk Java a pre realizáciu aspektov využijeme rozšírenie jazyka Java s názvom AspectJ.

Nakoniec navrhneme možné ďalšie smerovanie výskumu v otázke kompozitných anotácií.

Ciele práce

- Vysvetliť problematiku kompozitných anotácií
- Na základe analýzy zdrojových kódov existujúcich open-source projektov dokázať význam kompozitných anotácií
- Načrtnúť možné riešenie problému kompozitných anotácií pomocou, pre tento účel nového, aspektovo orientovaného prístupu

- Porovnať aspektovo orientovaný prístup s už opísaným možným riešením kompozitných anotácií, nástrojom Daileon, porovnať tieto dva prístupy a vyvodiť závery

Prínosy práce

- Potvrdenie problému opakovania sa rovnakých skupín anotácií (kapitola 2 – "Analýza výskytu anotačných vzorov vo vybraných zdrojových kódoch")
- Ponúknutie nového pohľadu na riešenie problému kompozitných anotácií pomocou aspektovo orientovaného programovania (kapitola 3 – "Kompozitné anotácie pomocou aspektovo orientovaného programovania")
- Parametrizovanie kompozitných anotácií, navrhnutie spôsobu ako na základe parametra kompozitnej anotácie dekomponovať rôzne skupiny anotácií (podkapitola 3.4 – "Parametrizované kompozitné anotácie")
- Identifikácia kompozitných anotácií, ktoré sa dekomponujú z rodičovských elementov na dcérske (pod)elementy na základe anotácie rodičovského programového elementu (podkapitola 3.5 – "Stromové kompozitné anotácie")

1 Anotácie v jazyku JAVA

Anotácie v jazyku JAVA poskytujú spôsob ako spájať lubovoľne informácie (inak nazývané ako "metaúdaje" [22] alebo "údaje o údajoch") s programovými elementmi. Syntakticky, anotácie sú používané ako modifikátory a môžu byť aplikované pri deklarovaní typov, konštruktorov, metód, atribútov či lokálnych premenných [5].

Anotácie boli zavedené verziou J2SE 5.0 vydanej v roku 2004. Špecifikácia jazyka Java, počínajúc verziou J2SE 5.0 týmto definuje šesť vstavaných anotácií, spôsob ako deklarovať anotačné typy, ako anotovať deklarácie a ako tieto anotácie neskôr čítať. Okrem týchto vstavaných anotácií je zavedená podpora pre vytváranie a použitie anotácií definovaných užívateľom. Každá anotácia je potom považovaná za modifikátor jazyka Java a môže byť použitá na anotovanie deklarácií balíčkov, typov, konštruktorov, metód, atribútov, parametrov a lokálnych premenných [17]. Pre anotácie existuje viacero základných možných využití. Medzi inými sú to informácie kompilátora (s využitím anotácií kompilátora) na potlačenie upozornení a detekciu chýb. Napríklad, vývojari môžu využívať anotáciu @SupressWarning("unchecked") na potlačenie nechcených upozornení, vyvolaných používaním API starších verzií jazyka Java. Ďalším využitím anotácií môže byť vytváranie logov alebo testovanie (vzhľadom na to že anotácie môžu byť spracovávané aj počas behu programu) či anotácia @Help("...") z ktorej možno pomocou na to určeného nástroja vygenerovať Help súbor.

Anotácie sú metaúdajmi, ktoré neovplyvňujú priamo anotovaný zdrojový kód. Z hľadiska implementácie v jazyku Java, anotácie možno považovať za špeciálny prípad rozhrania. Na odlíšenie deklarácie anotácie a deklarácie rozhrania bolo pre deklaráciu anotácie zavedené kľúčové slovo "@interface" [6], ktoré sa od kľúčového slova pre deklaráciu rozhrania "interface" odlišuje pridaním znaku "@". Zápis deklarácie novej anotácie @Table by vyzeral takto:

```
public @interface Table { ... }
```

Po deklarácii anotácie je možné vytvorenú anotáciu deklarovať nad programové elementy pomocou jej mena s pridaním prefixu "@", v našom prípade by mala deklarácia nad programovými elementmi tvar "@Table". Konkrétny príklad zápisu deklarácie anotácie @Table nad triedou Department, v zdrojovom kóde písanom v jazyku Java, by vyzeral nasledovne:

```
@Table
class Department implements DatabaseObject { ... }
```

pričom anotácia @Table by mohla byť využitá pri spracovaní napríklad k tomu, aby sa na základe nej s pomocou špecializovaného programového rámca automatizovane vytvorila v databáze tabuľka Department. Pridaním anotácií aj nad atribúty tejto triedy by mohli deklarované anotácie nad triedou aj (vybranými) atribútmi automatizovať vytvorenie tabuľky spolu so stĺpcami, ktoré by atribúty triedy opisovali. Rozšírenie o atribúty s deklarovanými anotáciami by mohlo vyzerať napríklad takto:

```
@Table
class Department implements DatabaseObject
{
   @Column @Id
   Integer id;

   @Column
   String name;

   @Column
   String code;
}
```

kde anotácia @Table by označovala že trieda nad ktorou je táto anotácia deklarovaná je v databáze z hľadiska databázových objektov tabuľkou, anotácia @Column by označovala stĺpec tabuľky a anotácia @Id by označovala že sa jedná o primárny kľúč. V poslednom zápise triedy Department možno nad atribútom "id" vidieť, že

programový element môže mať nad sebou deklarovaný variabilný počet anotácií.

Anotácie (pod označením ako "atribúty") sú využívané aj v jazyku C#, na rozdiel od jazyka Java majú však niektoré anotácie formu kľúčových slov [14].

Špecifickým prípadom využívania anotácií je situácia, keď programátor deklaruje viacero jednotlivých anotácií nad programovými elementmi jazyka Java (nad balíčkami, triedami, konštruktormi a podobne) tak, aby vyjadril funkcionalitu opísanú každou z anotácií. V tomto momente sa ponúka otázka, či a do akej miery táto činnosť vplýva na efektivitu programovania, rovnako aj na prehľadnosť zdrojových kódov. Tejto otázke sa venujú napríklad práce [1] [19]. V týchto prácach bolo navrhnuté zjednotenie rôznych, často sa opakujúcich, anotácií do celku, ktorý by sémantikou opisoval celú skupinu anotácií ktoré reprezentuje. Takáto skupina anotácií, vyjadrená jednou anotáciou, by niesla názov kompozitná anotácia.

1.1 Kompozitné anotácie

Kompozitné anotácie možno definovať ako prostriedok na zjednotenie skupín anotácií do celkov, pričom sémantika takéto jedného celku je súčtom sémantickych významov anotácií ktoré združuje. Kompozitné anotácie sú implementované ako bežné anotácie, ktorých "komponentmi" (anotáciami ktoré kompozitná anotácia združuje) sú elementy, do ktorých sa v konečnom dôsledku kompozitná anotácia "rozbalí" [21]. Hlavným prínosom kompozitných anotácií je sprehľadnenie a skrátenie zdrojového kódu, čo je dôsledkom zredukovania skupín anotácií do kompozitných anotácií. Takáto redukcia má však význam hlavne vtedy, ak dochádza k opakovaniu rovnakých skupín anotácií nad rovnakými programovými elementmi. Ilustračným príkladom anotačného vzoru, vyskytujúceho sa nad atribútmi triedy Department môže byť:

```
class Department implements DatabaseObject
{
   @Column @NotNull @Basic @Unique
   Integer id;
```

```
    @Column @NotNull @Basic
    String name;

    @Column @NotNull @Basic
    String code;
}
```

v tomto prípade kombinácia anotácií @Column @NotNull @Basic vytvára anotačný vzor, o ktorom má zmysel uvažovať z hľadiska jeho kompozície. Je tomu tak preto, lebo takýto anotačný vzor sa vyskytuje nad viacerými rovnakými programovými elementmi a v prípade jeho využitia aj pri ďalších programových elementoch kompozitný zápis sprehľadní zdrojový kód a urýchli jeho písanie.

Pokladáme za významné vyzdvihnúť dva najvýznamnejšie prípady kompozície. Prvým prípadom je kompozícia anotácií nad jedným druhom programových elementov. Univerzálny zápis takéhoto anotačného vzoru nad programovým elementom by bolo možné vyjadriť takto:

```
@Annotation1 @Annotation2 .. @AnnotationN
ProgramElement
```

pričom ProgramElement môže byť balíček, typ, konštruktor, metóda, atribút, parameter či lokálna premenná. Kompozíciou anotácií by mohla byť anotácia @CompositeAnnotation, ktorá by združovala anotácie @Annotation1, @Annotation2, .. @AnnotationN a pri jej dosadení za opísané anotácie by významom tieto anotácie nahrádzala. Zápis po nahradení kompozitnou anotáciou by vyzeral nasledovne:

```
@CompositeAnnotation
ProgramElement
```

pre ProgramElement ktorý by mohol byť balíčkom, typom, konštruktorom, metódou, atribútom, parametrom či lokálnou premennou. S náväznosťou na príklad uvedený v predchádzajúcej podkapitole, kde sme opísali deklaráciu dvoch anotácií @Column a

@Id nad atribútom, kompozíciou týchto dvoch anotácií by mohla byť tretia anotácia, nazvaná napríklad @PrimaryKeyColumn, ktorá by anotácie @Column a @Id a ich významy združovala. Pôvodná časť zdrojového kódu

```
@Table
class Department implements DatabaseObject
{
   @Column @Id
   Integer id;

   ...
}
```

by s využitím kompozície anotácií by vyzerala takto:

```
@Table
class Department implements DatabaseObject
{
   @PrimaryKeyColumn
   Integer id;

   ...
}
```

Druhým prípadom je kompozícia anotácií z dcérskych programových elementov (napríklad atribútov) na nadradený programový element (v tomto prípade triedu). Druhý prípad je však využiteľný len vtedy, ak pre všetky dcérske programové element platí to, že je nad nimi deklarovaná tá istá anotácia (prípadne skupina anotácií). Anotačný vzor nad atribútmi triedy by mal tvar:

```
ParentClass
{
   @Annotation1 @Annotation2 .. @AnnotationN
   Type1 attribute1;
```

```
   @Annotation1 @Annotation2 .. @AnnotationN
   Type1 attribute2;

   @Annotation1 @Annotation2 .. @AnnotationN
   Type1 attribute3;

   (methods)
}
```

Kompozíciou anotácií @Annotation1 @Annotation2 .. @AnnotationN by v tomto prípade vznikla kompozitná anotácia @ParentAnnotation, ktorá by bola deklarovaná nad nadradeným (rodičovským) programovým elementom, v našom prípade nad triedou. Anotácia @ParentAnnotation, deklarovaná nad triedou by zabezpečila distribúciu anotácií @Annotation1 @Annotation2 .. @AnnotationN nad podradené (dcérske) elementy. Zdrojový kód s využitím takejto kompozitnej anotácie by mal tvar:

```
@ParentAnnotation
ParentClass
{
   Type1 attribute1;

   Type1 attribute2;

   Type1 attribute3;

   (methods)
}
```

Je však dôležité znova poznamenať, že je v tomto prípade potrebné, aby boli všetky anotácie @Annotation1 @Annotation2 .. @AnnotationN definované nad všetkými

dcérskymi elementmi pre zabezpečenie správnej distribúcie smerom od kompozitnej anotácie k anotáciam ktoré združuje. Ak implementácia vyžaduje rôzne skupiny anotácií nad dcérskymi elementmi, pre využiteľnosť takejto kompozície je potrebné nájsť najmenšiu spoločnú skupinu anotácii, vyskytujúcu sa nad všetkými dcérskymi elementmi. Taktiež je potrebné poznamenať, že využitie takejto kompozície nie je obmedzené len na kombináciu triedy s jej atribútmi, ale sú možné aj kombinácie triedy a metód, či potenciálne balíčkov a tried (aj keď tento posledný prípad nebol predmetom skúmania, obsiahnutého v tejto práci).

Našim cieľom je dokázať význam kompozitných anotácií, o čo sa pokúsime pomocou analýzy zdrojových kódov programov so zameraním na výskyt kompozitných anotácií, následne bude našim ďalším cieľom rozšíriť riešenie problému kompozitných anotácií pomocou aspektovo orientovaného programovania. V nasledujúcich kapitolách sa budeme venovať už spomenutej analýze, následne prínosu, ktoré aspektovo orientované programovanie môže v tejto oblasti ponúknuť.

2 Štúdia opakovaného výskytu anotačných vzorov v praxi

Vytváranie kompozitných anotácií pomocou dostupných nástrojov, či možných nových spôsobov, má zmysel vtedy, ak je preukázateľné, že v zdrojových kódoch programov naozaj dochádza k situáciam, kedy sú vo väčšej miere využívané opakujúce sa skupiny anotácií (anotačné vzory). Aj keď najväčší prínos kompozitných anotácií má situácia keď sa opakuje čo najväčšia skupina anotácií (počet členov takejto skupiny je však vec hodná zváženia, preto túto úvahu nechávame na čitateľovi), má zmysel uvažovať aj skupiny ktoré obsahujú aspoň dva členy. V zmysle vyzdvihnutia prínosu kompozitných anotácií sme preto pokladali za dôležité spraviť štúdiu vzorky zdrojových kódov, ktoré by zmysel kompozitných anotácií potvrdili, prípadne vyvrátili.

Formulujeme hypotézu, že v zdrojových kódoch programov sa vyskytujú skupiny anotácií, ktoré sa opakujú. Platnosť tejto hypotézy potvrdí zmysel kompozitných anotácií.

Pri výbere projektov pre štúdiu zdrojových kódov sme kládli dôraz na to, aby vyberané projekty využívali programové rámce jazyka Java, pre ktoré je známe časté využívanie anotácií. Medzi takéto programové rámce patria napríklad programové rámce JPA, EJB a JSF. Projekty sme čerpali z verejne dostupných repozitárov typu GIT a SVN, pri hľadaní ktorých nám pomohol portál code.google.com. Z opisu sme vylúčili anotačné vzory, ktoré obsahujú natívne anotácie jazyka Java (anotácie balíčka java.lang) a ktoré nie sú pre svoju blízku zviazanosť s jazykom Java (a formu kľúčových slov - napríklad anotácia Override) predmetom výskumu tejto práce. Predmet výskumu tejto práce je zameraný v prvom rade na anotácie využívané v rámci externých programových ramcov tretích strán a programátorom vytvorené vlastné anotácie, pri ktorých má programátor možnosť uvažovať o vytvorení ich kompozície, vlastnej pre neho a jeho programovacie návyky.

Za účelom čo najväčšej automatizácie analýzy zdrojových kódov, pochádzajúcich z verejne dostupných repozitárov, sme navrhli nástroj tvorený anotačným procesorom, ktorý vo fáze kompilácie zdrojové kódy dokáže analyzovať, poskytnúť získané výsledky, ako aj exportovať tieto výsledky do formátu XML súboru, ktorý možno ďalej vyhodnocovať. V nasledujúcich podkapitolách opíšeme fungovanie anotačného procesora, na ktorom je náš nástroj postavený, rovnako opíšeme aj samotný nástroj a jeho štruktúru, na vybraných zdrojových kódoch programov demonštrujeme jeho činnosť a výsledky analýzy opíšeme.

V čase ukončovania písania tejto práce bol oznámený koniec portálu code.google.com, z ktorého sme čerpali analyzované projekty. Čitateľa preto odkazujeme na priložené médium, ktoré obsahuje všetky obsiahnuté projekty a ich zdrojové kódy.

2.1 Automatizácia analýzy zdrojových kódov

Vzhľadom na povahu problému, deklarovanie (skupín) anotácií skrz všetkými zdrojovými kódmi projektov, sme usúdili že analýzu zdrojových kódov bude vhodné automatizovať. Tým by sme sa vyhli zdĺhavému prezeraniu zdrojových kódov každého z analyzovaných projektov, tvoreného v niektorých prípadoch desiatkami tried. Na dosiahnutie cieľa sme využili štandardné API, poskytované v jazyku Java na spracovanie anotácií – anotačný procesor.

2.1.1 Opis fungovania anotačného procesora v jazyku Java

Zavedením nástroja zvaného "Annotation processing tool" (APT) do jazyka Java vznikol priestor na vytváranie vlastných anotačných procesorov, teda tried, ktoré slúžia na spracovanie anotácií vo vopred definovanom súbore zdrojových kódov. Anotačný procesor využíva na spracovanie programových anotácií sadu reflexívnych prostredí a podpornej infraštruktúry. APT vykonáva spracovanie vo viacerých kolách, pričom v každom kole zavolá definované anotačné procesory, ktoré následne

vykonajú svoje spracovanie. V prvom kole spracovania, APT spustí analýzu vstupných zdrojových súborov, následne spustí prehľadávanie týchto súborov a nakoniec zavolá príslušné anotačné procesory. V druhom kole APT analyzuje prípadné novo vzniknuté zdrojové súbory, ktoré sú výsledkom prvého kola spracovania, spustí prehľadávaciu procedúru nad novo vzniknutými súbormi a opäť volá príslušné anotačné procesory. Ak v druhom kole vznikli ďalšie zdrojové súbory, opísaný proces sa opakuje až do situácie keď v kole spracovania nedôjde k vzniku žiadneho nového zdrojového súboru. Po skončení posledného kola spracovania nástrojom APT, preddefinovaným ďalším postupom je zavolanie operácie javac nad originálnymi a novo vygenerovanými zdrojovými súbormi [25].

Hlavnou potrebou v našom prípade je získanie konkrétnych anotácií a programových elementov nad ktorými sa vyskytujú z obsiahnutých zdrojových kódov a priebežné vytváranie anotačných vzorov. Vytváranie anotačných vzorov sa v našom prípade skladá z krokov vyjadrených nasledujúcim pseudokódom:

- vytvor prázdnu pomocnú sadu sada_vyskytu_anotačných_vzorov
- získaj všetky programové elementy
- pre každý programový element aktuálny_programový_element sprav:
  - do aktuálna_skupina_anotácií vlož anotácie aktuálneho programového elementu
  - pre všetky programové elementy (okrem elementu aktuálny_programový_element)
    - získaj skupinu anotácií nad aktuálne prechádzaným elementom
    - nájdi najväčšiu spoločnú skupinu anotácií s anotáciami v aktuálna_skupina_anotácií
    - zisti či sa takáto skupina už nachádza v kľúčoch sady sada_vyskytu_anotačných_vzorov
    - nenachádza sa: ulož skupinu do pomocnej sady sada_vyskytu_anotačných_vzorov (kľúč = anotácie anotačného vzoru, hodnota = programové elementy nad ktorými sa anotačný vzor vyskytuje)

- nachádza sa: pridaj nájdený programový element do hodnoty opísanej kľúčom – skupinou anotácií

Výsledkom analýzy prostredníctvom našej implementáciou anotačného procesora je potom sada tvorená kľúčmi (anotačnými vzormi) a hodnotami (programovými elementami nad ktorými sa daný anotačný vzor, kľúč, vyskytuje), ktorú je následne možné ďalej spracovať. Pre tieto účely naša implementácia končí svoju činnosť prejdením celej sady anotačných vzorov a ich uložením do XML súboru, určeného na celkovú analýzu kompozitných anotácií všetkých budúcich testovaných zdrojových kódov. V nasledujúcich podkapitolách opíšeme metódy, ktoré tvoria implementáciu nášho riešenia.

2.1.2 Metódy navrhnutej implementácie anotačného procesora

Hlavnou, a zároveň povinnou, metódou, ktorou je tvorená naša implementácia je metóda *process()*. Jej vzor je nasledujúci:

```
@Override
public boolean process ( Set<? extends TypeElement> annotations,
                         RoundEnvironment roundEnv ) { ... }
```

Jej vstupnými parametrami je sada anotácií (parameter *annotations*) a parameter typu RoundEnviroment, ktorý poskytuje prístup okrem iného k programovým elementom spracovaných zdrojových kódov. Táto metóda je hlavnou metódou anotačného procesora, implementovaného v jazyku Java, je automaticky volaná anotačným procesorom pri jeho štarte. Prebieha v nej hlavná činnosť spracovania anotačným procesorom. V našej implementácii anotačného procesora je jej náplňou v prvom rade získanie všetkých programových elementov, prechádzanie všetkými elementmi spojené s priebežným vytváraním anotačných vzorov a nakoniec vytvorenie XML elementov z výslednej sady anotačných vzorov a ich uloženie do vopred definovaného súboru.

Naša metóda *getLongestMatch*, ktorá je volaná z metódy *process()*, slúži na porovnanie dvoch skupín anotácií za účelom vytvorenia najväčšieho možného anotačného vzoru z týchto skupín. Jej vzor je:

```
private List<String> getLongestMatch
( List<? extends AnnotationMirror> annotations1,
List<? extends AnnotationMirror> annotations2 ) { ... }
```

Jej vstupnými parametrami sú dva zoznamy tvorené prvkami typu rozširujúceho typ AnnotationMirror. Tento typ reprezentuje anotáciu, tvorenú jej anotačným typom (triedou, ktorej je táto anotácia inštanciou) a parametrami anotácie.

Poslednou metódou je metóda *puMatch*, slúžiaca na uloženie anotačného vzoru do pomocnej sady, zhromažďujucej anotačné vzory vo všetkých zdrojových kódoch. Jej vzor je:

```
private void putMatch ( List<String> matchList,
HashSet<Element> pairOfElements ) { ... }
```

Jej úlohou je nájsť anotačný vzor, opísaný prvým z parametrov, v kľúčoch pomocnej sady. V prípade existencie anotačného vzoru má za úlohu doplniť do hodnoty opísanej nájdeným kľúčom programové elementy, nad ktorými sa takýcto vzor vyskytuje (pričom programové elementy opisuje druhý vstupný parameter). V prípade neexistencie hľadaného kľúča (anotačného vzoru) v sade má táto metóda za úlohu vytvoriť v sade záznam, ktorej kľúčom bude anotačný vzor a hodnotou skupina programových elementov, opísaných druhým vstupným parametrom.

2.2 Analýza zdrojových kódov programov so zameraním na kompozitné anotácie

V nasledujúcich podkapitolách opíšeme desať náhodne vybraných analyzovaných projektov a opíšeme najčastejšie v nich využívané anotačné vzory. Každá z podka-

pitol bude obsahovať názov projektu, najčastejšie sa vyskytujúce anotačné vzory a tabuľku všetkých vyskytujúcich sa anotačných vzorov.

2.2.1 Projekt AutoRent-master

Projekt využíva najmä anotačné vzory:

- @Entity, @XmlRootElement - kompozícia zložená z anotácie @Entity programového rámca JPA určenej na objektovo-relačné mapovanie v databáze a anotácie @XmlRootElement využívanej pri prevode programového elementu jazyka Java do jazyka XML; kompozícia sa opakuje 9 krát
- @Id, @GeneratedValue, @Basic - kompozícia zložená z troch anotácií programového rámca JPA (objektovo-relačné mapovanie v databáze); kompozícia sa opakuje rovnako 9 krát
- @Basic, @NotNull - kompozícia zložená z anotácie @Basic programového rámca JPA (objektovo-relačné mapovanie v databáze) a anotácie @NotNull určenej na validáciu entity v databáze (constraint - obmedzenie stĺpca databázovej tabuľky); kompozícia sa opakuje 8 krát

Všetky nájdené anotačné vzory možno vidieť v tabuľke na obrázku (2 – 1):

2.2.2 Projekt InternetShop-master

Projekt využíva najmä anotačné vzory:

- @ManyToOne, @JoinColumn - kompozícia zložená z dvoch anotácií programového rámca JPA (objektovo-relačné mapovanie v databáze); kompozícia sa opakuje 4 krát
- @Basic, @Enumerated - kompozícia zložená z dvoch anotácií programového rámca JPA (objektovo-relačné mapovanie v databáze); kompozícia sa opakuje

Názov projektu: AutoRent-master		
P.č.	Anotačné vzory:	Počet výskytov:
1	@javax.persistence.Basic(optional=false) @javax.validation.constraints.NotNull	8
2	@javax.persistence.Basic(optional=false) @javax.validation.constraints.NotNull @javax.validation.constraints.Size(min=1, max=20) @javax.persistence.Column(name="haslo")	2
3	@javax.persistence.Basic(optional=false) @javax.validation.constraints.NotNull @javax.persistence.Temporal(javax.persistence.TemporalType.DATE)	2
4	@javax.persistence.Basic(optional=false) @javax.validation.constraints.NotNull @javax.validation.constraints.Size(min=1, max=20)	4
5	@javax.persistence.JoinColumn(name="oddzial_id", referencedColumnName="id_oddzial") @javax.persistence.ManyToOne(optional=false)	2
6	@javax.persistence.Basic(optional=false) @javax.validation.constraints.NotNull @javax.validation.constraints.Size(min=1, max=20) @javax.persistence.Column(name="login")	2
7	@javax.persistence.Entity @javax.xml.bind.annotation.XmlRootElement	9
8	@javax.validation.constraints.Size(max=45) @javax.persistence.Column(name="lokalizacja")	2
9	@javax.persistence.Id @javax.persistence.GeneratedValue(strategy=javax.persistence.Generation Type.IDENTITY) @javax.persistence.Basic(optional=false)	9

Obrázok 2 – 1 Anotačné vzory projektu AutoRent-master

Názov projektu: InternetShop-master		
P.č.	Anotačné vzory:	Počet výskytov:
1	@javax.persistence.Lob @javax.persistence.Basic(fetch=javax.persistence.FetchType.EAGER)	2
2	@javax.persistence.ManyToOne @javax.persistence.JoinColumn(name="ID_ITEM", referencedColumnName="ID_ITEM", nullable=false)	4
3	@javax.faces.bean.ManagedBean @javax.faces.bean.ViewScoped	3
4	@javax.persistence.Basic @javax.persistence.Enumerated(javax.persistence.EnumType.STRING)	4
5	@javax.persistence.ManyToOne @javax.persistence.JoinColumn(name="ID_CATEGORY", referencedColumnName="ID_CATEGORY", nullable=false)	2
6	@javax.ws.rs.GET @javax.ws.rs.Produces({"text/html"})	2
7	@javax.ws.rs.GET @javax.ws.rs.Produces({"application/json"})	2
8	@javax.persistence.Column(name="QUANTITY", nullable=false, insertable=true, updatable=true, length=22, precision=0) @javax.persistence.Basic	2

Obrázok 2 – 2 Anotačné vzory projektu InternetShop-master

4 krát

- @ManagedBean, @ViewScoped - kompozícia zložená z anotácií balíčka javax.faces.bean, opakujúca sa 3 krát

Všetky nájdené anotačné vzory možno vidieť v tabuľke na obrázku (2 – 2):

2.2.3 Projekt Loja_EJB-master

Projekt využíva najmä anotačné vzory:

- @Column, @NotNull - kompozícia zložená z anotácií @Column programového rámca JPA
  (objektovo-relačné mapovanie v databáze) a @NotNull (constraint - obmedzenie stĺpca databázovej tabuľky), opakujúca sa 6 krát

Názov projektu: Loja_EJB-master		
P.č.	Anotačné vzory:	Počet výskytov:
1	@javax.validation.constraints.NotNull @javax.persistence.Column(nullable=false, unique=true)	2
2	@javax.persistence.Column(nullable=false, unique=true) @javax.validation.constraints.NotNull	3
3	@javax.persistence.Column(nullable=false) @javax.validation.constraints.NotNull	6
4	@javax.persistence.Id @javax.persistence.GeneratedValue(strategy=javax.persistence. GenerationType.AUTO)	5
5	@javax.faces.bean.ManagedBean @javax.faces.bean.SessionScoped	6

Obrázok 2 – 3 Anotačné vzory projektu Loja_EJB-master

- @ManagedBean, @SessionScoped - kompozícia zložená z anotácií balíčka javax.faces.bean, upakujúca sa rovnako 6 krát
- @Id, @GeneratedValue - kompozícia zložená z anotácií programového rámca JPA
  (objektovo-relačné mapovanie v databáze), opakujúca sa 5 krát

Všetky nájdené anotačné vzory možno vidieť v tabuľke na obrázku (2 – 3):

2.2.4 Projekt esfinge-jhunt-master

V tomto projekte síce došlo k častému využitiu kompozície anotácií @SuppressWarnings a @Override, z dôvodu opísaného v rodičovskej podkapitole bola takáto kombinácia z nasledujúceho opisu vylúčená. Projekt okrem kompozície natívnych anotácií @SuppressWarnings a @Override jazyka Java využíva najmä anotačný vzor:

- @NotNull, @ManyToMany - kompozícia anotácie @ManyToMany programového rámca JPA (objektovo-relačné mapovanie v databáze) a @NotNull (constraint - obmedzenie stĺpca databázovej tabuľky), upakujúca sa 5 krát

Všetky nájdené anotačné vzory možno vidieť v tabuľke na obrázku (2 – 4):

Názov projektu: esfinge-jhunt-master		
P.č.	Anotačné vzory:	Počet výskytov:
1	@javax.validation.constraints.NotNull @javax.persistence.ManyToMany(fetch=javax.persistence.Fetch Type.LAZY)	5
2	@java.lang.SuppressWarnings({"unchecked"}) @java.lang.Override	7
3	@javax.persistence.ManyToOne @javax.persistence.JoinColumn(name="id_usuario", referencedColumnName="id_usuario")	2
4	@org.junit.runner.RunWith(org.junit.runners.Suite.class) @org.junit.runners.Suite.SuiteClasses({})	2

Obrázok 2 – 4 Anotačné vzory projektu esfinge-jhunt-master

Názov projektu: nabster_blog		
P.č.	Anotačné vzory:	Počet výskytov:
1	@javax.persistence.Id @javax.persistence.GeneratedValue(strategy=javax.persistence. GenerationType.AUTO)	4
2	@javax.faces.bean.ManagedBean @javax.faces.bean.SessionScoped	4

Obrázok 2 – 5 Anotačné vzory projektu nabster_blog

2.2.5 Projekt nabster_blog

Projekt využíva anotačné vzory:

- @Id, @GeneratedValue - kompozícia zložená z anotácií programového rámca JPA
  (objektovo-relačné mapovanie v databáze), opakujúca sa 4 krát
- @ManagedBean, @SessionScoped - kompozícia anotácií balíčka javax.faces.bean, upakujúca sa rovnako 4 krát

Všetky nájdené anotačné vzory možno vidieť v tabuľke na obrázku (2 – 5):

2.2.6 Projekt crank0.15

Projekt využíva kompozície zložené aj z natívnych anotácií jazyka Java (najmä @SuppressWarnings, @Retention a @Target), pričom ide o kompozície vyskytujúce sa v projekte 19 krát, 21 krát, 10 krát a podobne - teda v dosť významnom množstve. Aj keď takéto opakovanie je znakom opakovania sa anotačných vzorov, konkrétne tieto opakovania však do nasledujúceho opisu nezahrnieme, nakoľko tieto anotácie sú natívne anotácie jazyka Java, na ktoré nie je výskum tejto práce zameraný. Projekt z ostatných anotačných vzorov využíva najmä nasledujúce:

- @Id, @GeneratedValue(s parametrom AUTO) - kompozícia zložená z anotácií programového rámca JPA (objektovo-relačné mapovanie v databáze), opakujúca sa 15 krát
- @Id, @GeneratedValue(bez parametrov) - kompozícia zložená z anotácií programového rámca JPA (objektovo-relačné mapovanie v databáze), opakujúca sa 4 krát
- @Required, @ProperNoun - kompozícia vlastných anotácií, deklarovaných v balíčku org.crank.annotations.validation, určeních na validáciu hodnôt; kompozícia sa opakuje 4 krát

Všetky nájdené anotačné vzory možno vidieť v tabuľke na obrázku (2 – 6):

2.2.7 Projekt injection-extensions

Projekt využíva najmä anotačné vzory:

- @RunWith, @TransactionAttribute - kompozícia anotácie @RunWith balíčka org.junit a anotácie @TransactionAttribute programového rámca EJB (programový rámec určený na programovanie back-end, serverovej logiky); kompozícia sa opakuje 5 krát

Názov projektu: crank0.15		
P.č.	Anotačné vzory:	Počet výskytov:
1	@org.crank.annotations.validation.Required @org.crank.annotations.validation.Date	2
2	@org.crank.annotations.validation.Required @org.crank.annotations.validation.ProperNoun	4
3	@java.lang.SuppressWarnings({"JpaModelErrorInspection"}) @javax.persistence.Embeddable	3
4	@java.lang.SuppressWarnings({"serial"}) @javax.persistence.Entity	2
5	@java.lang.SuppressWarnings({"unchecked"}) @ExternalBean	8
6	@javax.persistence.Id @javax.persistence.GeneratedValue	4
7	@java.lang.SuppressWarnings({"deprecation"}) @org.testng.annotations.Configuration	2
8	@javax.persistence.Entity @javax.persistence.Inheritance(strategy=javax.persistence.InheritanceType.JOINED)	2
9	@Bean @ScopedProxy	4
10	@javax.persistence.Id @javax.persistence.GeneratedValue(strategy=javax.persistence.GenerationType.AUTO)	15
11	@java.lang.SuppressWarnings({"unchecked"}) @Test	10
12	@java.lang.SuppressWarnings({"unchecked"}) @java.lang.Override	2
13	@ScopedProxy @Bean	3
14	@org.crank.annotations.validation.Required @org.crank.annotations.validation.ProperNoun @org.crank.annotations.validation.Length(min=2L, max=35L)	2
15	@javax.persistence.OneToMany(cascade={javax.persistence.CascadeType.ALL}) @Cascade	2
16	@org.crank.annotations.design.DependsOnJSF @org.crank.annotations.design.DependsOnSpring	2
17	@java.lang.SuppressWarnings({"unchecked"}) @Bean	21
18	@java.lang.annotation.Retention(java.lang.annotation.RetentionPolicy.RUNTIME) @java.lang.annotation.Target({java.lang.annotation.ElementType.METHOD, java.lang.annotation.ElementType.TYPE})	19

Obrázok 2 – 6 Anotačné vzory projektu crank0.15

- @Test, @TransactionDisabled - kompozícia anotácie @Test balíčka org.junit (slúžiaca na označenie metódy pre JUnit ako vhodnej pre testy) a anotácie @TransactionDisabled, ktorá je vlastnou anotáciou, vytvorenou programátorom; kompozícia sa opakuje 4 krát
- @ContainerContext, @RunWith - kompozícia vlastnej anotácie @ContainerContext, vytvorenej programátorom a anotácie @RunWith balíčka org.junit, opakujúca sa 4 krát

Všetky nájdené anotačné vzory možno vidieť v tabuľke na obrázku (2 – 7):

2.2.8 Projekt appbudget

Projekt využíva najmä anotačné vzory:

- @Entity, @XmlRootElement - kompozícia zložená z anotácie @Entity programového rámca JPA (objektovo-relačné mapovanie v databáze) a anotácie @XmlRootElement (prevod programového elementu jazyka Java do jazyka XML); opakujúca sa 6 krát
- @Id, @GeneratedValue, @Basic - kompozícia zložená z troch anotácií programového rámca JPA (objektovo-relačné mapovanie v databáze), opakujúca sa 6 krát
- @Size, @Column - kompozícia anotácie @Size (constraint - obmedzenie stĺpca databázovej tabuľky) a anotácie @Column programového rámca JPA (objektovo-relačné mapovanie v databáze); kompozícia sa opakujúce 3 krát
- @Basic, @NotNull, @Size, @Column - kompozícia dvoch anotácií programového rámca JPA (objektovo-relačné mapovanie v databáze); anotácie @Basic a @Column) a dvoch anotácií balíčka javax.validation obsahujúceho anotácie určené na obmedzenie stĺpcov databázovej tabuľky (constraint; anotácie @NotNull a @Size); kompozícia sa opakuje 2 krát

Názov projektu: injection-extensions		
P.č.	Anotačné vzory:	Počet výskytov:
1	@org.junit.runner.RunWith(com.hrodberaht.inject.extension.transaction.junit.InjectionJUnitTestRunner.class) @javax.ejb.TransactionAttribute	5
2	@java.lang.annotation.Target({java.lang.annotation.ElementType.TYPE, java.lang.annotation.ElementType.METHOD}) @java.lang.annotation.Retention(java.lang.annotation.RetentionPolicy.RUNTIME)	2
3	@org.junit.Test(expected=com.hrodberaht.inject.extension.transaction.manager.internal.TransactionHandlingError.class) @com.hrodberaht.inject.extension.transaction.junit.TransactionDisabled	3
4	@org.junit.Test @com.hrodberaht.inject.extension.transaction.junit.TransactionDisabled	4
5	@java.lang.annotation.Retention(java.lang.annotation.RetentionPolicy.RUNTIME) @java.lang.annotation.Inherited	4
6	@java.lang.annotation.Retention(java.lang.annotation.RetentionPolicy.RUNTIME) @java.lang.annotation.Target({java.lang.annotation.ElementType.TYPE}) @java.lang.annotation.Inherited	2
7	@com.hrodberaht.inject.extension.transaction.junit.InjectionContainerContext(test.com.hrodberaht.inject.extension.transaction.example.ModuleContainerForTests.class) @org.junit.runner.RunWith(com.hrodberaht.inject.extension.transaction.junit.InjectionJUnitTestRunner.class)	3
8	@com.hrodberaht.inject.extension.transaction.junit.InjectionContainerContext(test.com.hrodberaht.inject.extension.transaction.example.ModuleContainerForTests.class) @org.junit.runner.RunWith(com.hrodberaht.inject.extension.transaction.junit.InjectionJUnitTestRunner.class) @javax.ejb.TransactionAttribute	2
9	@org.hrodberaht.inject.extension.tdd.ContainerContext(test.org.hrodberaht.inject.extension.ejbunit.ejb3.config.EJBContainerConfigExample.class) @org.junit.runner.RunWith(org.hrodberaht.inject.extension.tdd.JUnitRunner.class)	2
10	@org.hrodberaht.inject.extension.tdd.ContainerContext(test.org.hrodberaht.inject.extension.cdi.config.CDIContainerConfigExample.class) @org.junit.runner.RunWith(org.hrodberaht.inject.extension.tdd.JUnitRunner.class)	2
11	@org.hrodberaht.inject.extension.tdd.ContainerContext(test.org.hrodberaht.inject.extension.ejbunit.demo.test.config.CourseContainerConfigExample.class) @org.junit.runner.RunWith(org.hrodberaht.inject.extension.tdd.JUnitRunner.class)	4

Obrázok 2 – 7 Anotačné vzory projektu injection-extensions

Názov projektu: appbudget		
P.č.	Anotačné vzory:	Počet výskytov:
1	@javax.persistence.Basic(optional=false) @javax.validation.constraints.NotNull @javax.validation.constraints.Size(min=1, max=100) @javax.persistence.Column(name="nombre")	2
2	@javax.validation.constraints.Size(max=100) @javax.persistence.Column(name="nombre")	3
3	@javax.persistence.Entity @javax.xml.bind.annotation.XmlRootElement	6
4	@javax.persistence.Id @javax.persistence.GeneratedValue(strategy=javax.persistence.GenerationType.IDENTITY) @javax.persistence.Basic(optional=false)	6

Obrázok 2 – 8 Anotačné vzory projektu appbudget

Všetky nájdené anotačné vzory možno vidieť v tabuľke na obrázku (2 – 8):

2.2.9 Projekt proyecto-megabanco

Projekt obsahuje v podstate len dva anotačné vzory, všetky anotácie vzorov sú z programového rámca JPA, slúžia na objektovo-relačné mapovanie v databáze. Líšia sa len parametrami (odkazujem čitateľa na nižšie uvedenú referenciu na obrázok). Anotačné vzory:

- @ManyToOne, @JoinColumn - opakujú sa od dvoch až po päť opakovaní (bližšie uvedené v priloženej tabuľke)
- @Entity, @Inheritance - kompozícia opakujúca sa 5 krát

Všetky nájdené anotačné vzory možno vidieť v tabuľke na obrázku (2 – 9):

2.2.10 Projekt dsc132-marcelo

Projekt obsahuje anotačný vzor zložený z anotácií @SuppressWarning a @Override, ten však z dôvodov uvedených v rodičovskej podkapitole neberieme do úvahy. Využívajú sa však okrem nich aj anotačné vzory:

Názov projektu: proyecto-megabanco		
P.č.	Anotačné vzory:	Počet výskytov:
1	@javax.persistence.ManyToOne @javax.persistence.JoinColumn(name="idUniversidad")	2
2	@javax.persistence.ManyToOne @javax.persistence.JoinColumn(name="id_Cuenta")	4
3	@javax.persistence.ManyToOne @javax.persistence.JoinColumn(name="idCiudad")	3
4	@javax.persistence.ManyToOne @javax.persistence.JoinColumn(name="idPrograma")	3
5	@javax.persistence.ManyToOne @javax.persistence.JoinColumn(name="id_Entidad")	4
6	@javax.persistence.Entity @javax.persistence.Inheritance	5
7	@javax.persistence.ManyToOne @javax.persistence.JoinColumn(name="id_Cliente")	5
8	@javax.persistence.ManyToOne @javax.persistence.JoinColumn(name="id_Tipo")	3

Obrázok 2 – 9 Anotačné vzory projektu proyecto-megabanco

Názov projektu: dsc132-marcelo		
P.č.	Anotačné vzory:	Počet výskytov:
1	@java.lang.SuppressWarnings({"unchecked"}) @java.lang.Override	4
2	@javax.persistence.Id @javax.persistence.GeneratedValue(strategy=javax.persistence.GenerationType.AUTO)	8
3	@javax.ejb.Stateless @javax.ejb.LocalBean	9

Obrázok 2 – 10 Anotačné vzory projektu dsc132-marcelo

- @Stateless, @LocalBean - kompozícia zložená z anotácií programového rámca EJB (programový rámec určený na programovanie back-end, serverovej logiky), opakujúca sa 9 krát
- @Id, @GeneratedValue - kompozícia anotácií z programového rámca JPA (objektovo-relačné mapovanie v databáze), opakujúca sa 8 krát

Všetky nájdené anotačné vzory možno vidieť v tabuľke na obrázku (2 – 10):

2.3 Vyhodnotenie a závery

Z vyššie uvedených podkapitol, opisujúcich analýzu vybraných desiatich projektov, odvodíme vyhodnotenie najčastejšie sa opakujúcich anotačných vzorov.

- najčastejšie sa opakujúcim anotačným vzorom bola kompozícia anotácií @Id a @GeneratedValue(strategy=GenerationType.AUTO) programového rámca JPA, slúžiacich na objektovo-relačné mapovanie v databáze; celkový počet opakovaní tohto anotačného vzoru vo všetkých desiatich projektoch bol 32 krát
- ďalším najčastejšie sa opakujúcim anotačným vzorom bola kompozícia anotácií @Id a @GeneratedValue(strategy=GenerationType.IDENTITY) programového rámca JPA - kompozícia opakujúca sa 16 krát
- tretím najčastejšie sa opakujúcim anotačným vzorom bola kompozícia anotácií @Id a @GeneratedValue(strategy=GenerationType.IDENTITY), tak ako to bolo v predchádzajúcom prípade, avšak s pridaním anotácie Basic(optional=false), pochádzajúcich z programového rámca JPA - kompozícia opakujúca sa 15 krát
- štvrtým vybraným, najčastejšie sa opakujúcim, anotačným vzorom bola kompozícia anotácií @Entity (programový rámec JPA) a @XmlRootElement (balíček javax.xml.bind.annotation - prevod programového elementu jazyka Java do jazyka XML) - kompozícia opakujúca sa taktiež 15 krát

Z uvedených tabuliek z každého opísaného projektu možno štatisticky vyvodiť počet anotačných vzorov na projekt. Pre štatistiku si vyberieme len signifikantné anotačné vzory (také, ktoré nie sú tvorené ani jednou natívnou anotáciou jazyka Java a počet opakovaní anotačného vzoru v projekte je aspoň 3x). Súčtom všetkých anotačných vzorov získavame hodnotu 199 celkových opakovaní akéhokoľvek anotačného vzoru ktorý sa v ktoromkoľvek z projektov vyskytoval aspoň 3 krát, čo v priemere dáva 19 až 20 signifikantných opakovaní anotačných vzorov na projekt.

Z najčastejšie opakujúcich sa anotačných vzorov si dovolíme vyvodiť dva závery.

Prvým záverom je, že analýzou a uvedeným priemerným počtom anotačných vzorov na testovaný projekt bolo potvrdené časté opakovanie sa anotačných vzorov, čím sme potvrdili našu hypotézu. To taktiež potvrdilo zmysel zaoberať sa potrebou kompozitných anotácií, rovnako potrebu nástroja, ktorý by uľahčil prácu s opakujúcimi sa skupinami anotácií, čím by okrem iného zrýchlil vytváranie programov a sprehľadnil zdrojové kódy programov, ktoré takéto opakujúce sa skupiny anotácií využívajú.

Druhý záver spočíva v tom, že prvé tri priečky najčastejšie sa opakujúcich anotačných vzorov pochádzajú z programového rámca JPA, v niektorých prípadoch líšiacich sa len v parametri anotácie. Z vykonanej analýzy vyplynul fakt, že takéto anotačné vzory sa opakujú a opakujú sa v nezanedbateľnom počte. Námetom na budúce skúmanie by mohlo byť sústredenie sa na jeden programový rámec (či už JPA, alebo akýkoľvek iný s rovnakým potenciálom opakovania sa anotačných vzorov) a vytvorenie kompozitných anotácií v rámci tohto jedného programového rámca, ktoré by opisovali potreby programátorov. Takéto kompozitné anotácie by boli tvorené z anotácií programového rámca, ktoré majú potenciál často sa opakovať (ako napríklad v rámci programového rámca JPA vyššie opísané, často sa opakujúce anotačné vzory, anotácie @Id a @GeneratedValue).

3 Kompozitné anotácie pomocou aspektovo orientovaného programovania

Inšpiráciu pre riešenie kompozitných anotácií cez aspektovo orientované programovanie sme našli v knihe AspectJ in Action [28], časť na stranách 130–132. Tá načrtla možnosť ako v zdrojových kódoch hľadať programové elementy na základe anotácie deklarovanej nad nimi. To sa nám javilo ako základ (de)kompozície anotácií a možný nový pohľad na problém, či možné riešenie problému kompozitných anotácií.

Aspektovo orientovaný prístup ponúka iný pohľad na problém kompozitných anotácií, ako spomínané riešenie Daileon. Hlavným rozdielom je nezávislosť od zdrojového kódu programu, na ktorý takzvané "aspekty" (nástroj aspektovo orientovaného prístupu pre definovanie programových elementov na ktoré sa chceme viazať a činnosti ktorá sa má vykonať v prípade daných udalostí s takýmito programovými elementmi) chceme viazať. Aspekt reprezentuje jeden modul, jeden záujem [20]. Oddeľovaním záujmov [24], ktoré aspektovo orientovaný prístup ponúka a rozvíja, možno sprehľadniť zdrojový kód, čo napomáha pochopeniu problému aj pri budúcom rozširovaní. Z hľadiska kompozitných anotácií je využitie aspektovo orientovaného prístupu výhodné najmä kvôli tomu, že kompozitné anotácie možno sústrediť na jednom mieste, v aspektoch, kde je možné definovať kompozitnú anotáciu vzhľadom na anotácie ktoré združuje, ako aj činnosť ktorá sa má vykonať v prípade ak takáto kompozitná anotácia bude v zdrojovom kóde obsiahnutá - nahradenie kompozitnej anotácie definovanými anotáciami. Následne v kóde je možné využívať definované kompozitné anotácie, pričom keď sa vo výsledku vo fáze kompilácie zdrojového kódu pomocou definovaných aspektov kompozitné anotácie nahradia anotáciami ktoré združujú, je dosiahnutý rovnaký výsledok ako v prípade keď programátor napíše v zdrojovom kóde nad programový element namiesto kompozitnej anotácie priamo anotácie, inak združené v kompozitnej anotácii [10].

Taktiež aspektovo orientovaný prístup ponúka v rámci svojich možností oba prípady

kompozície anotácií, opísaných v podkapitole "Kompozitné anotácie", čím poskytuje dobré zázemie pre skúmanie možností v rámci problému kompozitných anotácií. A v neposlednom rade, riešenie problému kompozitných anotácií pomocou aspektovo orientovaného programovania si nevyžaduje implementáciu špecifického nástroja, ako napríklad v prípade nástroja Daileon, ale využíva samotný charakter aspektovo orientovaného programovania.

V nasledujúcich podkapitolách bližšie opíšeme aspektovo orientovaný prístup a opíšeme prostriedky ktoré ponúka pre tvorbu kompozitných anotácií.

3.1 Aspektovo orientované programovanie

Ak chceme riešiť nejaký problém pomocou objektovo orientovaného programovania (OOP), väčšinou do implementácie riešenia potrebujeme zahrnúť viacero rôznych, vzájomne spolupracujúcich tried. Zvyčajné sú dva hlavné spôsoby ako v OOP dosiahnúť dekompozíciu algoritmu na metódy. Buď sa vydať cestou vloženia celej implementácie riešenia do metód jednej triedy, alebo rozdeliť implementáciu do metód viacerých tried. Dôsledkom prvého postupu je to, že pre potrebnú funkcionalitu je potrebné naviazať veľa informácií zo štruktúry triedy (metódy, atribúty) na každú z využívaných metód tejto triedy, čím vzniká problém so zmenami v štruktúrach tried (ak sa zmení niektorá z metód, alebo atribút, prípadne viacero abtribútov, je potrebné meniť každú z metód v ktorej sa takýto programový element vyskytoval). Druhý postup je problémový z hľadiska keď chceme zmeniť implementáciu riešenia, vtedy je potrebné vo všetkých využitých triedach prispôsobiť využité metódy [7].

Aspektovo orientovaný vývoj softvéru (AOSD - aspect oriented software development [16]) vylepšuje oddeľovanie záujmov v softvérovom vývoji [23]. AOSD umožňuje modularizovanie takýchto triedami prelínajúcich sa problémov, čím zlepšuje udržiavanie a rozširovanie softvérových riešení. Výskum v oblasti AOSD sa zameriava najmä na softvérový dizajn, analýzu problémov a implementáciu v konkrétnych

jazykoch. Aj keď je známe že testovanie je procesom náročným na prácu a čas ktorý sa odzrkadľuje na polovici nákladov softvérového vývoja, vývoju v testovaní pomocou AOSD, hlavne automatizácii testovania, sa dostáva málo pozornosti [9].

Aspektovo orientované programovanie má však aj svoje nedostatky. Hlavným nedostatkom je v súčastnosti ešte stále potreba rozšírenia jazyka "prierezových bodov" (v origináli "pointcuts"), ktoré slúžia na definovanie miesta v kóde zapísanom v objektovom jazyku - môžu to byť medzi inými volania metód, deklarácie či volania konštruktorov tried, na ktoré sa aspekty viažu [8].

3.2 Návrh využitia aspektovo orientovaného prístupu pri tvorbe kompozitných anotácií

Aspektovo orientované programovanie si osvojilo viacero jazykov, či už implementáciou do jazyka alebo pomocou externých knižníc. My využijeme rozšírenie jazyka Java s názvom AspectJ [11] [13] (podobné rozšírenie bolo spracované aj pre jazyk C#, nesie názov AspectC# [15]).

AspectJ nám poskytuje nástroje pre detekciu anotácií či deklarovanie nových anotácií podľa takzvanej medzitypovej deklarácie (inter type declaration) [12] [13]. V nasledujúcich podkapitolách si opíšeme možnosti využitia aspektovo orientovaného programovania pri tvorbe kompozitných anotácií.

Vo všetkých nižšie opísaných podkapitolách využívame formulu "declare annotation" [4]. Jej univerzálny tvar je nasledujúci:

```
declare @kind : ElementPattern : Annotation ;
```

kde "@kind" môže byť @type pre triedu, @method pre metódu, @constructor pre konštruktor alebo @field pre atribút triedy, "ElementPattern" má tvar vzoru triedy, metódy, konštruktora alebo atribútu triedy a "Annotation" je nová anotácia, ktorú chceme deklarovať nad daný objekt.

Medzitypové deklarácie pomocou formuly "declare annotation" nachádzajú svoje využitie napríklad v nástroji Alice, vyvíjaného Darmstadtskou univerzitou technológií [18].

Za účelom demonštrácie kompozitných anotácií vyjadrených pomocou aspektovo orientovaného programovania tam, kde je to aplikovateľné, používame anotácie programového rámca JPA, konkrétne anotácie @Id a @GeneratedValue, ktoré boli predmetom častého opakovania v analýze v predchádzajúcej kapitole. Tieto dve anotácie sú určené pre atribúty triedy. Programový rámec JPA (Java Persistence API) slúži na objektovo-relačné mapovanie v databáze, teda spájanie programových elementov jazyka Java s objektmi databázy a ich vlastnosťami, typmi a obmedzeniami.

Anotácie, ktoré budeme v príkladoch využívať si na úvod opíšeme. Načrtneme tak dôvod, prečo potrebujú byť tieto anotácie deklarované spolu.

- anotácia @Id reprezentuje primárny kľúč entity (možno to chápať aj ako tabuľky) v databáze; atribút opísaný anotáciou @Id musí byť buď primitívneho typu jazyka Java (int, long, byte, ...), primitívnou wrapper triedou jazyka Java (prináša objektové metódy pre primitívne typy jazyka Java; napr. Integer, Long, Byte, ...), String, java.util.Date, java.sql.Date, java.math.BigDecimal alebo java.math.BigInteger [26].
- anotácia @GeneratedValue poskytuje špecifikáciu stratégie generovania primárneho kľúča [27].

Pre techniku kompozitných anotácií sú podstatné najmä prípady dekompozície kompozitných anotácií na anotácie, ktoré sú ich komponentmi. V nasledujúcich podkapitolách však najmä kvôli symetrii a ilustrácii opačného postupu opíšeme aj kompozíciu kompozitných anotácií.

3.3 Vkladanie anotácie na základe inej anotácie

Ako prvým a základným krokom ku kompozitným anotáciam je dôležité spomenúť základnú operáciu a to nahradenie anotácie inou anotáciou, čo AspectJ podporuje.

3.3.1 Anotácie nad triedou

Tvar formuly declare annotation je nasledovný:

```
declare @type : typePattern : domainAnnotation;
```

kde typePattern je vzor triedy (definuje triedu ktorú v zdrojovom kóde hľadáme) a domainAnnotation je nová anotácia, ktorú chceme nad požadovanou triedou deklarovať.

[Príklad] V príklade uvažujme imaginárnu prípadovú štúdiu, model kamenného obchodu - predajne, vyjadreného triedou Shop. Predpokladajme že trieda Shop spadá pod ekonomické oddelenie, ktoré ho má z hľadiska fakturácie a účtovníctva na starosti. Z tejto konkrétnej väzby medzi ekonomickým oddelením a obchodom ako takým pre model, triedu Shop, vyplývajú aj isté obmedzenia (ktorými sa však táto prípadová štúdia zaoberať nebude) ako aj výhody. Aby sme označili túto triedu vzhľadom na oddelenie, využijeme na to anotáciu @EconomicsDepartment. Uvažujme že jedinou funkcionalitou tejto triedy je práca s databázou, vytváranie tabuliek, manipulácia s nimi a volanie storovaných procedúr. Túto funkcionalitu vyjadríme anotáciou @DatabaseObject. Trieda Shop v tom prípade navrhnutá nasledovne:

```
@DatabaseObject
@EconomicsDepartment
public class Shop { ... }
```

Skúmaním zdrojových kódov sme zistili, že takáto kombinácia anotácií sa vyskytuje veľmi často aj nad inými triedami, pracujúcimi s databázou a prislúchajúcimi pod ekonomické oddelenie a usúdili sme, že je vhodné využiť kompozitnú anotáciu.

Takúto kompozitnú anotáciu si nazveme @EconomicsTable. S využitím AspectJ, automatickú kompozíciu na základe nájdených anotácií @DatabaseObject a @EconomicsDepartment vieme zapísať nasledovne:

```
declare @type : @DatabaseObject@EconomicsDepartment package.* :
@EconomicsTable;
```

ktorý slovne možno opísať takto: ”deklaruj anotáciu @EconomicsTable nad ľubovoľnými typmi (triedami), ktoré sa nachádzajú v balíku s názvom ”package” (časť package.*), ktoré sú anotované anotáciami @DatabaseObject a @EconomicsDepartment”.

Automatická kompozícia však z hľadiska významu kompozitných anotácií ako nástroja na uľahčenie písania zdrojových kódov a zvýšenia prehľadnosti nie je veľmi podstatná. Oveľa podstatnejšou je však opačná situácia, keď chceme kompozitnú anotáciu ”rozložiť” na pôvodné anotácie z ktorých vznikla. Tá však pri aspektovo orientovanom prístupe naráža na prekážku kedy nie je možné deklarovať viacero anotácií naraz a je potrebné rozpísať každú z nahradzovaných anotácií do osobitnej formuly.

[Príklad] Majme triedu Shop s deklarovanou kompozitnou anotáciou @EconomicsTable. Potrebujeme ju spätne rozložiť na anotácie @DatabaseObject a @EconomicsDepartment. To možno dosiahnúť nasledovným spôsobom. Povedzme, že trieda Shop je opísaná kompozitnou anotáciou nasledovne:

```
@EconomicsTable
public class Shop { ... }
```

Chceme navodiť opačnú situáciu, keď z kompozitnej anotácie spätne deklarujeme pôvodné anotácie. Ako sme uviedli, formula declare annotation nepodporuje deklarovanie viacerých anotácií naraz. Je preto potrebné vytvoriť toľko formúl, koľko anotácií potrebujeme získať. Fromuly majú v našom prípade tvar:

```
declare @type : @EconomicsTable package.* : @DatabaseObject;
```

```
declare @type : @EconomicsTable package.* : @EconomicsDepartment;
```

3.3.2 Anotácie nad metódou

Obidva spôsoby, vytvorenie kompozitnej anotácie aj jej dekompozícia, sú aplikovateľné aj v prípade metód. Rozdiel je v tvare formuly declare annotation, čo si bližšie ozrejmíme v nasledujúcom príklade.

[Príklad] Pri hlbokej úvahe celého týmu programátorov v našej imaginárnej spoločnosti sme zistili že množstvo anotácií nad metódami tried, reprezentujúcimi manipuláciu so storovanými procedúrami v databáze, je potrebné vyjadriť kompozitnou anotáciou. Jednou z takýchto metód je napríklad metóda

```
@FullDatabaseAccess
@LockingTable
public void payToCashRegister(...) { ... }
```

ktorá má plný prístup k databáze pre čítanie, úpravu, zápis a mazanie čo vyjadruje anotácia @FullDatabaseAccess a uzamyká tabuľku kvôli tomu aby zabránila kolíziam pri operácii vkladania, úpravy a mazania čo vyjadruje anotácia @LockingTable. Tieto anotácie chceme zlúčiť do anotácie @CriticalStoredProcedure.

Formula declare annotation v takomto prípade má tvar:

```
declare @method :
@FullDatabaseAccess@LockingTable * *.*(..) :
@CriticalStoredProcedure;
```

čo sa dá slovne vyjadriť nasledujúcim spôsobom: ”deklaruj anotáciu @CriticalStoredProcedure nad všetkými metódami všetkych dostupných tried s ľubovoľnými parametrami (časť *.*(..)) s ľubovoľnou viditeľnosťou (s pridaním hviezdicky pre ľubovoľnú viditeľnosť - * *.*(..)), ktoré sú anotované anotáciami @FullDatabaseAccess a @LockingTable”.

V opačnom prípade, keď chceme kompozitnú anotáciu @CriticalStoredProcedure rozložiť na anotácie z ktorých vznikla, musíme pre každú z anotácií definovať vlastnú formulu. Tie majú tvar:

```
declare @method : @CriticalStoredProcedure * *.*(..) :
@FullDatabaseAccess;
declare @method : @CriticalStoredProcedure * *.*(..) :
@LockingTable;
```

3.3.3 Anotácie nad atribútom triedy

Podobné je to aj v prípade atribútov tried. Prejdeme rovno na príklad, kde si uvedieme tvar formuly declare annotation, prísluchajúcu deklarácii atribútov tried.

[Príklad] Analýza v predchádzajúcich kapitolách potvrdila, že kompozitné anotácie majú význam, nakoľko dochádza k opakovaniu anotačných vzorov. Anotačný vzor, ktorý analýza ukázala ako najčastejšie sa opakujúci bol vzor reprezentujúci primárny kľúč v modeli, triede, opisujúcom ľubovoľnú entitu v databázovej schéme. Na definovanie primárneho kľúča tabuľky databázy boli využívané anotácie programového rámca JPA @Id a @GeneratedValue(strategy=GenerationType.AUTO). Na základe nich opíšeme tvar kompozície a (najmä) dekompozície pomocou aspektovo orientovaného prístupu.

Ako primárny kľúč našej triedy Shop sme si zvolili atribút id_shop. Pre definíciu primárneho kľúča sme usúdili, že potrebujeme využiť opísané anotácie programového rámca JPA:

```
@Id
@GeneratedValue(strategy=GenerationType.AUTO)
private int id_shop;
```

Automatickú kompozíciu opísaných anotácií do kompozitnej anotácie @GeneratedId vieme dosiahnúť pomocou formuly declare annotation nasledovne:

```
declare @field :
@Id@GeneratedValue(strategy=GenerationType.AUTO) * *.* :
@GeneratedId;
```

čo sa dá slovne opísať nasledujúcim spôsobom: ”deklaruj anotáciu @GeneratedId nad všetkými atribútmi ľubovoľných tried s ľubovoľnou viditeľnosťou, anotovanými anotáciami @Id a @GeneratedValue(strategy=GenerationType.AUTO)”.

[Príklad] Vzhľadom na to, že automatická dekompozícia je pre nás tým podstatným v otázke kompozitných anotácií, nasledujúci príklad je postavený na možnej reálnej kompozícii anotácií @Id a @GeneratedValue(strategy=GenerationType.AUTO), význam ktorej potvrdila analýza v predchádzajúcich kapitolách. Uvažujme, že kompozíciou anotácií @Id a @GeneratedValue(strategy=GenerationType.AUTO) definujeme kompozitnú anotáciu @GeneratedId.

Povedzme, že atribút id_shop je opísaný kompozitnou anotáciou nasledovne:

```
@GeneratedId
private int id_shop;
```

Chceme navodiť opačnú situáciu, keď z kompozitnej anotácie spätne deklarujeme pôvodné anotácie. Ako sme uviedli, formula declare annotation nepodporuje deklarovanie viacerých anotácií naraz. Je preto potrebné vytvoriť toľko formúl, koľko anotácií potrebujeme získať. Fromuly majú v našom prípade tvar:

```
declare @field : @PrimaryColumnAnnotation * *.* : @Id;
declare @field : @PrimaryColumnAnnotation * *.* :
@GeneratedValue(strategy=GenerationType.AUTO);
```

Takýmto spôsobom možno dosiahnúť riešenie problému, že kombinácia anotácií @Id a @GeneratedValue(strategy=GenerationType.AUTO) sa veľmi často opakovala nad atribútmi v triedach modelov, kedy atribút mal byť definovaný ako primárny kľúč, ktorý sa automaticky inkrementuje. Programátor by v takomto prípade musel vytvoriť anotáciu, ktorú by bral ako kompozitnú, napísať aspekt s formulami

declare annotation, čím by zaistil neskoršiu dekompozíciu kompozitnej anotácie na anotácie @Id a @GeneratedValue(strategy=GenerationType.AUTO) a nakoniec by v zdrojovom kóde programu stačilo využívať ním vytvorenú kompozitnú anotáciu.

3.4 Parametrizované kompozitné anotácie

Niekedy je potrebné (či vhodné) deklarovať potrebné anotácie na základe parametra doménovej anotácie. Naša prípadová štúdia uvažuje parameter typu bool s hodnotou true alebo false. Vo všetkých troch prípadoch (trieda, metóda aj atribút) budeme uvažovať anotáciu @Privacy s parametrom isPrivate. V prípade ak je parameter isPrivate anotácie @Privacy hotnota true, deklarujeme v prípade dekompozície anotácie @MemberBound a @AuthorizationRequested. Ak je parameter isPrivate anotácie @Privacy hodnota false, v prípade dekompozície deklarujeme anotácie @PublicObject a @NonAuthorizedObject. V prípade kompozície deklarujeme na základe opísaných anotácií anotáciu @Privacy s prislúchajúcim parametrom. V nasledujúcich podkapitolách uvedieme tvar formuly declare annotation pre triedu, metódu a atribút triedy ako pre kompozíciu, tak aj pre dekompozíciu.

Primy prenos parametra kompozitnej anotácie na niektorý z jej komponentov nie je v AOP podporovaný. Preto všetky možnosti parametra, ktorý chceme aby kompozitná anotácia podporovala, potrebujeme vymenovať. V prezentovaných príkladoch pre jednoduchosť použijeme iba pravdivostný parameter, ktorý môže nadobúdať iba hodnoty true a false. Rovnako by však bolo možné použiť napr. parameter typu String, avšak opäť by sme museli vymenovať v aspekte všetky možnosti, ktoré má kompozitná anotácia podporovať, a pre každú z nich vypísať hodnoty komponentových anotácií a ich parametrov. Teda uviesť niečo také ako premennú na ľavej strane "declare annotation" príkazu a potom ju použiť na pravej strane ako hodnotu pre parameter komponentovej anotácie nie je v AOP umožnené.

Napriek tomu je niektoré situácie, ktoré by si takúto vlastnosť vyžadovali, možné

vyriešiť. Ak nejaká anotácia v opakujúcej sa skupine anotácií ma variabilný parameter, ale bola by vhodným komponentom kompozitnej anotácie, je možné namiesto vytvárania novej kompozitnej anotácie využiť túto anotáciu ako kompozitnú. To je možné vďaka tomu, že AOP iba vloží nové anotácie, avšak neodstráni kompozitnú anotáciu. Ak je táto anotácia použitá aj na miestach, kde nie je spolu so svojou skupinou, tak je potrebne pri definícii toho, kedy sa majú komponentové anotácia vkladať, uviesť obmedzujúcejší výraz a nie len * ako wildcard. Inak by mohlo dôjsť k aplikovaniu komponentových anotácií aj tam, kde to nebolo žiadané.

3.4.1 Anotácie nad triedou

Tvar deklarácie anotácií má pre kompozíciu a dekompozíciu anotácií tried nasledujúce tvary:

(kompozícia ak parameter isPrivate = true)

```
declare @type :
@MemberBound@AuthorizationRequested package.* :
@Privacy(isPrivate=true);
```

(dekompozícia ak parameter isPrivate = true)

```
declare @type : @Privacy(isPrivate=true) package.* :
@AuthorizationRequested;
declare @type : @Privacy(isPrivate=true) package.* :
@MemberBound;
```

Pre hodnotu parametra isPrivate s hodnotou false anotácie @Privacy je postup podobný ako v prípade ak je parameter isPrivate hodnotou true. V takomto prípade sú však nad triedu namiesto anotácií @MemberBound a @AuthorizationRequested deklarované anotácie @PublicObject a @NonAuthorizedObject.

3.4.2 Anotácie nad metódou

Tvar deklarácie anotácií pre kompozíciu a dekompozíciu v prípade metódy s využitím vzoru metódy má nasledujúci tvar:

(kompozícia ak parameter isPrivate = true)

```
declare @method :
@MemberBound@AuthorizationRequested * *.*(..) :
@Privacy(isPrivate=true);
```

(dekompozícia ak parameter isPrivate = true)

```
declare @method : @Privacy(isPrivate=true) * *.*(..) :
@MemberBound;
declare @method : @Privacy(isPrivate=true) * *.*(..) :
@AuthorizationRequested;
```

Platí rovnaké pravidlo ako v prípade triedy, kedy parameter isPrivate má hodnotu false. Konkrétne to, že postup je podobný ako v prípade ak je parameter isPrivate hodnotou true avšak namiesto anotácií @MemberBound a @AuthorizationRequested sú deklarované anotácie @PublicObject a @NonAuthorizedObject.

3.4.3 Anotácie nad atribútom triedy

Ako posledný príklad uvedieme tvar deklarácie anotácií na príklade kompozície a dekompozície anotácií nad atribútom triedy. Nakoľko sa jedná o atribút triedy, oprieme sa znova o výsledky analýzy. Výsledky analýzy hovoria, že prvé dve miesta zaujala kombinácia anotácií @Id a @GeneratedValue, pričom anotácia @GeneratedValue sa líšila len v parametri ”strategy”, ktorý nadobúdal dve hodnoty: ”GenerationType.IDENTITY” a ”GenerationType.AUTO”. Stratégií generovania hodnôt anotácie @GeneratedValue je síce viac, ale pre účely nasledujúceho príkladu uvažujme že potrebujeme len tieto dve. Definujeme si anotáciu @PrimaryColumnAnnotation

s parametrom typu bool s názvom ”isAutoStrategy”. Ak bude nadobúdať parameter isAutoStrategy hodnotu true, dekompozíciou deklarujeme anotácie @Id a @GeneratedValue(strategy=GenerationType.AUTO). V prípade ak bude parameter isAutoStrategy nadobúdať hodnotu false, deklarujeme anotácie @Id a @GeneratedValue(strategy=GenerationType.IDENTITY).

Deklarácia anotácií má v takomto prípade tvar: (kompozícia ak parameter isAutoStrategy = true)

```
declare @field :
@Id@GeneratedValue(strategy=GenerationType.AUTO) * *.* :
@PrimaryColumnAnnotation(isAutoStrategy=true);
```

(dekompozícia ak parameter isAutoStrategy = true)

```
declare @field :
@PrimaryColumnAnnotation(isAutoStrategy=true) * *.* :
@Id;
declare @field :
@PrimaryColumnAnnotation(isAutoStrategy=true) * *.* :
@GeneratedValue(strategy=GenerationType.AUTO);
```

Pre úplnosť, v prípade kompozície a dekompozície anotácií ak isAutoStrategy = false majú formuly declare annotation nasledujúce tvary: (kompozícia ak parameter isAutoStrategy = false)

```
declare @field :
@Id@GeneratedValue(strategy=GenerationType.IDENTITY) * *.* :
@PrimaryColumnAnnotation(isAutoStrategy=false);
```

(dekompozícia ak parameter isAutoStrategy = false)

```
declare @field :
@PrimaryColumnAnnotation(isAutoStrategy=false) * *.* :
@Id;
```

```
declare @field :
@PrimaryColumnAnnotation(isAutoStrategy=false) * *.* :
@GeneratedValue(strategy=GenerationType.IDENTITY);
```

Týmto vieme docieliť flexibilitu vlastných kompozitných anotácií a ešte viac zefektívniť písanie zdrojových kódov. Ďalším smerovaním pri kompozícii, ale najmä pri dekompozícii na základe parametra anotácie môže byť rozšírenie parametra anotácie na typ, ktorý by umožňoval ešte viac možností automatického deklarovania anotácií pomocou aspektov. Parameter by mohol mať napríklad podobu enumeračného typu, ktorý by vyjadroval viacero možností (v prípade anotácie @GeneratedValue by to boli stratégie generovania hodnôt, medzi inými vynechané stratégie GenerationType.SEQUENCE a GenerationType.TABLE). Dokonca by bolo možné uvažovať už existujúce typy, ktoré sú už programátorom, ak ich často využívajú, vlastné, čím by došlo k rýchlejšiemu navyknutiu si na kompozitné anotácie (v našom prípade by parameter nebol typu bool, ale priamo typu javax.persistence.GenerationType).

3.5 Stromové kompozitné anotácie

Na rozdiel od Daileonu, AspectJ podporuje deklarovanie anotácií ”po strome”, pričom uvažujeme že koreňom je v našom prípade trieda, listami sú metódy a atribúty. Nižšie opísaným spôsobom možno na základe anotácie nad triedou anotovať špecifickou anotáciou všetky atribúty či metódy. V nasledujúcich podkapitolách si uvedieme triviálny prípad, keď triedu označíme anotáciou @Table a na základe toho sa anotácie @TableColumn (v prípade atribútov tried) a @TableStoredProcedure (v prípade metód) deklarujú nad metódami a atribútmi tried.

3.5.1 Anotácie nad metódou

Našim prvým cieľom je deklarovať na základe anotácie @Table anotácie @TableStoredProcedure nad metódy. Formula declare annotation bude mať v takomto prípade

tvar:

```
declare @method : * (@Table *).*(..) : @TableStoredProcedure;
```

čo možno slovne interpretovať takto: ”deklaruj anotáciu @TableStoredProcedure nad všetky metódy s ľubovoľnou viditeľnosťou a ľubovoľnými parametrami, ktorých triedy sú anotované anotáciou @Table”.

3.5.2 Anotácie nad atribútom triedy

Našim druhým cieľom je deklarovať anotáciu @TableColumn nad také atribúty tried, ktorých trieda (v rámci ktorej sú definované) je anotovaná anotáciou @Table. To vieme dokázať formulou declare annotation v tvare:

```
declare @field : * (@Table *).* : @TableColumn;
```

čo možno slovne interpretovať nasledujúcim spôsobom: ”deklaruj anotáciu @TableColumn nad všetky atribúty s ľubovoľnou viditeľnosťou, ktorých trieda (v rámci ktorej sú deklarované) je anotovaná anotáciou @Table”.

4 Kompozitné anotácie pomocou nástroja Daileon

Existujúcim nástrojom, ktorý v oblasti kompozitných anotácií vznikol je nástroj s názvom Daileon. Autori v práci [1] rozvíjajú potrebu náhrady opakujúcich sa skupín anotácií kompozitnými anotáciami a ponúkajú nástroj, ktorý by riešenie tohto problému realizoval. Zameriavajú sa na termíny doména a z toho odvodená doménová anotácia. Doménou nazývajú oblasť z reálneho sveta, v kontexte ktorej je daný program písaný (medicína, bankovníctvo a podobne). Doménovou anotáciou je potom podľa autorov taká anotácia, ktorá sa funkcionalitou (alebo jej menom či významom) v danom zdrojovom kóde viaže na danú doménu a kompozitná anotácia, ktorá nahrádza iné anotácie ktoré do kontextu opisovanej domény nepatria. Nástroj Daileon využíva dve techniky realizácie kompozitných anotácií.

V prvom prípade uvažuje nový programový rámec, ktorý podporuje vytváranie doménových anotácií a povoľuje anotáciam programového rámca anotovať iné anotácie. Ide hlavne o prípad, keď doménová anotácia nie je viazaná na konkrétny typ programového elementu (na triedu, metódu, atribút), tým pádom je možné anotovať ňou inú anotáciu, v našom prípade kompozitnú anotáciu. Funkcionalita Daileonu v takomto prípade spočíva vo vytvorení doménovej anotácie, v jej anotovaní anotáciami ktoré má nahrádzať a anotovaní anotáciou @DomainAnnotation. Anotácia @DomainAnnotation slúži na identifikáciu že sa jedná o doménovú anotáciu a spätné zistenie pôvodných anotácií programového rámca, ktoré táto doménová anotácia nahrádza [2].

Druhý prípad hovorí o existujúcom programovom rámci, ktorý nepodporuje vytváranie doménových anotácií v takej podobe ako k tomu dochádza vo vyššie opísanom novom programovom rámci. K využitiu tohto, druhého, prípadu dochádza najmä vtedy, ak nastane vyššie opísaný problém - kompozitnú anotáciu chceme vytvoriť z anotácií, v množine ktorých je aspoň jedna anotácia viazaná na konkrétny typ programového elementu. V takomto prípade je využívaná manipulácia bajtového kódu

(bytecode manipulácia). Najprv je potrebné vytvoriť vzorovú triedu, ktorá, ak je to v kontexte kompozitnej anotácie ktorú chceme vytvoriť potrebné, obsahuje vzorové metódy a atribúty anotované anotáciami programového rámca ktoré chceme zhrnúť do kompozitnej anotácie. Jedna vzorová metóda reprezentuje vzor (parameter pre príslušnú anotáciu kompozitnej anotácie) pre jednu kompozitnú anotáciu. Následne je potrebné doménovú anotáciu anotovať anotáciou:

- ak je cieľom kompozitnej anotácie trieda, anotáciou @ClassTemplate(annotatedClass = "VZOROVÁ TRIEDA")
- ak je cieľom kompozitnej anotácie metóda, anotáciou @MethodTemplate(annotatedClass = "VZOROVÁ TRIEDA", method = "CIEĽOVÁ METÓDA VZOROVEJ TRIEDY")
- ak je cieľom kompozitnej anotácie atribút, anotáciou @FieldTemplate(annotatedClass = "VZOROVÁ TRIEDA", field = "CIEĽOVÝ ATRIBÚT VZOROVEJ TRIEDY")

ktorá doménovú anotáciu na základe vzorovej triedy (metódy, atribútu) naviaže na anotácie programového rámca ktoré má nahrádzať. Nakoniec, po skončení kompilácie, avšak ešte pred spustením aplikácie, sa konzolovým spustením nástroja Daileon a následnou manipuláciou bajtového kódu doménové anotácie nahradia korešpondujúcimi anotáciami programového rámca [1]. Na navigovanie k cieľovým triedam obsahujúcim kompozitné anotácie využíva nástroj Daileon ako vstup špecialne prispôsobený XML súbor.

V nasledujúcich podkapitolách opíšeme na príklade rovnakých problémov, ktoré sme uviedli pri opise aspektovo orientovaného programovania, možnosti nástroja Daileon. V úvode opíšeme na vzorovom príklade všetky dostupné možnosti nahrádzania, kompozície či dekompozície, následne uvedieme reálnu situáciu na základe výsledov výskumu z predchádzajúcich kapitol. Ako reálnu situáciu si zvolíme, ako tomu bolo aj v kapitole o aspektovo orientovanom programovaní, vytvorenie kompo-

zitnej anotácie @GeneratedId z doménových anotácií (anotácií programového rámca JPA) @Id a @GeneratedValue. Túto situáciu zahrnieme ku dekompozícii v prípade atribútu triedy, vzhľadom na cieľový programový element anotácií @Id @GeneratedValue. Pri vytvorení vlastných kompozitných anotácií v nástroji Daileon sme postupovali aj podľa návodov, uvedených na oficiálnej stránke tohto nástroja [3].

4.1 Vkladanie anotácie na základe inej anotácie

Základnú operáciu náhrady jednej anotácie inou anotáciou nástroj Daileon podporuje pre všetky tri vybrané programové elementy - triedu, metódu, atribút.

4.1.1 Anotácie nad triedou

V prípade náhrady jednej anotácie triedy inou anotáciou triedy je potrebné definovať vzorovú triedu nad ktorou je deklarovaná potrebná anotácia, následne je potrebné vytvoriť kompozitnú anotáciu anotovanú anotáciou @ClassTemplate nástroja Daileon, odkazujúcou na vzorovú triedu.

[Príklad] Majme triedu Shop, nad ktorou potrebujeme mať deklarované anotácie @DatabaseObject a @EconomicsDepartment. Tieto anotácie chceme zhrnúť do jednej anotácie @EconomicsTable. Trieda Shop môže vyzerať pred využitím kompozitnej anotácie napríklad nasledovne:

```
@DatabaseObject
@EconomicsDepartment
public class Shop { ... }
```

Naším cieľom je však vytvorenie kompozície anotácií s využitím nástroja Daileon. K tomu potrebujeme vzorovú triedu, opisujúcu abstraktne aké anotácie má naša cielená trieda Shop obsahovať. Vzorová trieda môže vyzerať napríklad nasledovne:

```
@DatabaseObject
```

```
@EconomicsDepartment
public class EconomicsTableTemplateClass { ... }
```

Následne je potrebné vytvoriť anotáciu triedy, ktorá previaže vzorovú triedu EconomicsTableTemplateClass s našou triedou Shop. Anotácia môže vyzerať napríklad nasledovne:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@DomainAnnotation
@ClassTemplate(annotatedClass = "EconomicsTableTemplateClass")
public @interface EconomicsTable { }
```

Nakoniec stačí triedu Shop anotovať anotáciou @EconomicsTable, bude preto vyzerať nasledovne:

```
@EconomicsTable
public class Shop { ... }
```

Táto situácia vyjadruje však až funkcionalitu dekompozície anotácie @EconomicsTable na anotácie @DatabaseObject a @EconomicsDepartment. Kompozícia, rozoznanie skupiny anotácií a automatické vytvorenie kompozitnej anotácie v nástroji Daileon nie je, na rozdiel od aspektovo orientovaného programovania, možné. Je však nutné podotknúť že pre účely využitia kompozitných anotácií je automatická dekompozícia pomocou definovaného nástroja (či už je to Daileon alebo aspektovo orientovaný nástroj) dôležitejšia ako kompozícia, preto s náväznosťou na primárny cieľ tejto práce tento poznatok neslúži ako prekážka.

4.1.2 Anotácie nad metódou

Vytvorenie kompozitnej anotácie v prípade metódy sa v svojej podstate nelíši od kompozície v prípade triedy, rozdielom je len využitie anotácie @MethodTemplate namiesto anotácie @ClassTemplate pri definovaní vlastnej, kompozitnej anotácie

a orientovanie sa na vzorovú metódu, nad ktorou definujeme žiadané doménové anotácie, pri vytváraní vzorovej triedy.

[Príklad] Majme metódu payToCashRegister, nad ktorou potrebujeme mať definované dve anotácie, ktoré chceme vyjadriť jednou, kompozitnou anotáciou. Pred vykonaním kompozície má pôvodná metóda tvar:

```
@FullDatabaseAccess
@LockingTable
public void payToCashRegister(...) { ... }
```

Môžeme využiť vzorovú triedu z príkladu kompozitnej anotácie nad triedou, ktorá bude mať tvar:

```
public class EconomicsTableTemplateClass
{
   @FullDatabaseAccess
   @LockingTable
   public void payToCashRegisterTemplateMethod()
}
```

Následne je potrebné vytvoriť vlastnú kompozitnú anotáciu, pomenujeme si ju @CriticalStoredProcedure. Bude mať tvar:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@DomainAnnotation
@MethodTemplate(annotatedClass = "EconomicsTableTemplateClass",
method = "payToCashRegisterTemplateMethod")
public @interface CriticalStoredProcedure { }
```

Vo výsledku je potom možné metódu payToCashRegister navrhnúť v tvare:

```
@CriticalStoredProcedure
public void payToCashRegister(...) { ... }
```

Uvedený postup vyjadruje v konečnom dôsledku, vo fáze kompilácie a behu, automatickú dekompozíciu definovanej anotácie @CriticalStoredProcedure na anotácie @FullDatabaseAccess a @LockingTable. Fáza automatickej kompozície, rovnako ako pri kompozícii v prípade triedy, pri metóde nie je v nástroji Daileon realizovateľná.

4.1.3 Anotácie nad atribútom triedy

Podobný postup je možné opísať aj v prípade atribútu triedy. Rozdiel je znova len vo vzorovej triede a anotácii @FieldTemplate, ktorá nahradí anotácie @ClassTemplate a @MethodTemplate z predchádzajúcich prípadov.

[Príklad] Nasledujúci príklad obsahuje reálne využitie kompozície dvoch anotácií programového rámca JPA, ktorej význam bol podložený analýzou v predchádzajúcich kapitolách. Zdrojový kód k realizácii tejto kompozície v nástroji Daileon prikladame v zdrojových kódov, prílohách k tejto práci.

Majme atribút id_shop, nad ktorým potrebujeme mať deklarované anotácie @Id a @GeneratedValue(strategy=javax.persistence.GenerationType.AUTO). Tieto chceme zlúčiť do anotácie @GeneratedId. Pôvodná deklarácia atribútu má tvar:

```
@Id
@GeneratedValue(strategy=GenerationType.AUTO)
public int id_shop;
```

Za účelom kompozície anotácií do kompozitnej anotácie využijeme vzorovú triedu z predchádzajúcich príkladov, jej tvar bude:

```
public class PrimaryIdDomainClass
{
   @Id
   @GeneratedValue(strategy=GenerationType.AUTO)
   public int primaryId;
}
```

Naša vytvorená, kompozitná anotácia bude mať tvar:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
@DomainAnnotation
@FieldTemplate(annotatedClass = "PrimaryIdDomainClass",
field = "primaryId")
public @interface PrimaryDatabaseColumn { }
```

Výsledný návrh atribútu, nad ktorým deklarujeme kompozitnú anotáciu potom bude mať tvar:

```
@GeneratedId
public int id_shop;
```

4.2 Parametrizované kompozitné anotácie

Nástroj Daileon neponúka možnosti, ako takýto cieľ dosiahnúť. Možným riešením, aj keď nie práve efektívnym, by bolo vytvorenie viacerých kompozitných anotácií, líšiacich sa malou zmenou v názve, ktorá by vyjadrovala o aký parameter pri využití tej ktorej anotácie máme záujem.

V takom prípade by sme mohli uvažovať vzorové kompozitné anotácie @Private a @Public, vyjadrujúce obdobnú situáciu opísanú v kapitole vkladania anotácie na základe parametra aspektovo orientovaným programovaním. Aspektovo orientované programovanie realizovalo možné riešenie pomocou anotácie @Privacy s parametrom isPrivate nadobúdajúcim hodnoty true alebo false. Táto anotácia by sa viazala na všetky nami uvažované druhy programových elementov. S ňou by sme uvažovali aj súvisiace anotácie @MemberBound a @AuthorizationRequested ak parameter isPrivate = true (v našom prípade to bude v prípade ak kompozitnou anotáciou je @Private), rovnako ako anotácie @PublicObject a @NonAuthorizedObject ak parameter isPrivate = false (v našom prípade by to bola kompozitná anotácia @Public).

Takto riešená situácia by však bola len vkladaním anotácií na základe inej anotácie, čo sme už opísali v predchádzajúcej kapitole.

4.3 Stromové kompozitné anotácie

Takýto druh kompozície v nástroji Daileon nie je možný, rovnako ako nebolo možné vkladanie na základe parametra anotácie.

5 Vyhodnotenie AOP riešenia

Cieľom tejto práce bolo vysvetliť problematiku anotácií a kompozitných anotácií, vyhodnotiť zmysel kompozitných anotácií, opísať existujúce riešenia a ponúknuť nový prístup k realizácii kompozitných anotácií prostredníctvom aspektovo orientovaného programovania. Myslíme si, že každý z uvedených bodov sme splnili.

Analýzou náhodne vybraných zdrojových kódov programov sme dokázali význam kompozitných anotácií. Vybrali sme najčastejšie sa opakujúci anotačný vzor, zahrnuli ho pri vysvetľovaní princípu kompozície a dekompozície pri už existujúcom prístupe pomocou nástroja Daileon, ale aj novom prístupe pomocou aspektovo orientovaného programovania.

Následnou analýzou existujúceho aj nového prístupu k otázke kompozitných anotácií chceme ďalej tieto dva prístupy vyhodnotiť a definovať prínos nového prístupu, ktorý bol opísaný touto prácou.

Ako už bolo uvedené, možné riešenie otázky kompozitných anotácií poskytlo dva hlavné pohľady, ktoré sa nám podarilo nájsť. Jedným z pohľadov bolo využitie už existujúceho riešenia - nástroja Daileon. Ďalším bolo využitie ešte nezdokumentovaného prístupu v otázke kompozitných anotácií, využitie aspektovo orientovaného programovania. Obidva pohľady majú pozitívne aj negatívne vlastnosti. Bližšie vyhodnotíme obidva prístupy v nasledujúcich podkapitolách.

5.1 Daileon

Na prvý pohľad nástroj, ktorého zložitosť využitia sa javila byť závislá len na programátorovych znalostiach jazyka Java, sľuboval kvôli tomuto faktu jednoduchosť vytvárania kompozitných anotácií. Realita bola však trochu iná. Je zložitejší na pochopenie ako by sa na prvý pohľad mohlo zdať. Opisuje dva prístupy ako vytvárať kompozitné anotácie. Je pravda, že prvý postup je pomerne jednoduchý na

pochopenie, avšak je závislý na tom, že anotácie nie sú viazané na jeden konkrétny programový element (metódu, triedu, ...). V našom prípade bolo potrebné využiť druhý spôsob, postavený na bajtovej manipulácii a ten si vyžiadal značné množstvo času na pochopenie, po pochopení relatívne malé množstvo času na implementovanie a nakoniec znova značné množstvo času na spracovanie nástrojom. Vzhľadom na to, že manipulácia bajtového kódu nie je triviálnou operáciou, nástroj je potrebné spúšťať cez príkazový riadok, čo značne obmedzuje spätnú väzbu ak je v tvorbe kompozitnej anotácie nejaká, hoci len drobná, chyba. A to bolo dôvodom, prečo posledná fáza - spracovanie zdrojového kódu nástrojom - zabrala značné množstvo času.

Pozitívnymi vlastnosťami riešenia kompozitných anotácií pomocou nástroja Daileon boli najmä to, že definovanie dekompozície kompozitnej anotácie je postavené len na deklarovaní anotácií nad potrebným programovým elementom vo vzorovej triede. Zápis je jednoduchý a prehľadný. Nástroj Daileon ponúka aj pomocné metódy ako detegovať anotácie dosadené bajtovou manipuláciou.

Nástroj Daileon má aj pár negatívnych vlastností. Jednou z nich je, ako už bolo načrtnuté, relatívne zložitý proces akým treba prejsť pri tvorbe kompozitnej anotácie od jej vytvorenia ako anotačného rozhrania (a správneho definovania potrebných anotácií deklarovaných nad ňou, ako aj pomocnej triedy) kým je bajtovou manipuláciou dosadená nad požadovaný programový element. V neposlednom rade sú to (v porovnaní s aspektovo orientovaným prístupom) okresané možnosti nástroja Daileon, medzi ktoré patrí napríklad nemožnosť deklarovania anotácií na základe parametra kompozitnej anotácie, či nemožnosť deklarovania anotácií po strome tvorenom rodičovskými a dcérskymi programovými elementami (napríklad na základe anotácie triedy vygenerovať anotácie na všetky metódy a podobne).

5.2 Aspektovo orientované programovanie

Aspektovo orientovaný prístup pri tvorbe kompozitných anotácií, na ktorý bola mierená aj myšlienka tejto práce, priniesol nezvyčajný pohľad na automatické deklarovanie anotácií, prvotnú nedôveru v možnosti ktoré by mohol tento prístup ponúkať však vystriedali prekvapujúce výsledky a ohybnosť formuly declare annotation v aspektoch.

Pozitívne vlastnosti aspektovo orientovaného prístupu sú hlavne už v načrtnutej ohybnosti ako možno zadeklarovať na ktorý programový element sa tá ktorá deklarácia anotácie vzťahuje. Ďalej je to prehľadnosť v definovaných kompozitných anotáciach (v prípade Daileonu sú kompozitné anotácie rozdelené na také množstvo dvojíc vzoru triedy a anotácie, koľko kompozitných anotácií chceme vytvoriť; v prípade aspektovo orientovaného prístupu je možné všetky kompozitné anotácie deklarovať v jednom aspekte, na jednom mieste, a všetky kompozitné anotácie definované anotačnými rozhraniami deklarovať hoci aj v jednom balíčku určenom práve na kompozitné anotácie, čo okrem prehľadnosti jednotlivých tried dosiahnutého kompozitnými anotáciami ešte naviac zvyšuje aj prehľadnosť projektu, pretože kompozitné anotácie sústredíme v jednom aspekte). Nakoniec je to rýchlosť, akou je možné zvyknúť si na využívanie kompozitných anotácií. Aj keď je pre programátora potrebné naučiť sa písať aspekty, zaberie to však podstatne menej času ako prejsť celým procesom deklarovania a vytvorenia kompozitnej anotácie v nástroji Daileon. Rovnako, aspektovo orientované programovanie je v súčasnosti štandardne využívanou súčasťou programovacích jazykov, preto nie je pre podporu kompozitných anotácií potrebné implementovať špecializovaný nástroj, akým je napríklad nástroj Daileon. Tieto pozitívne vlastnosti možno označiť aj ako prínosy tejto práce k riešeniu otázky kompozitných anotácií.

Medzi negatívne vlastnosti možno zaradiť najmä to, že na prvý pohľad klauzula declare annotation, pre programátora ktorý sa s ňou nestretol, nedáva veľký zmysel

a je potrebný istý čas na zoznámenie sa s týmto prístupom a naučenie sa všetkých možností, ktoré ohybnosť formuly declare annotation ponúka. Ďalšou negatívnou vlastnosťou je to, že aj keď je formula declare annotation ohybná v zmysle definície vzoru programového elementu na ktorý sa má vzťahovať, stále má nedostatky v tom, že je možné naraz deklarovať len jednu anotáciu. V prípade dekompozície kompozitnej anotácie to znamená toľko deklarácií, koľko anotácií tvorí kompozitná anotácia.

5.3 Zhodnotenie prínosov práce

- Potvrdili sme problém opakovania sa rovnakých skupín anotácií naprieč zdrojovými kódmi programov
- Ponúkli sme analýzu riešenia problému kompozitných anotácií pomocou aspektovo orientovaného programovania
- Ukázali sme spôsob, ako pomocou aspektovo orientovaného programovania možno parametrizovať kompozitné anotácie
- Rovnako sme ukázali spôsob ako je možné pomocou aspektovo orientovaného programovania vytvárať stromové kompozitné anotácie
- S pomocou prehľadnosti aspektovo orientovaného prístupu sme ponúkli programátorom v otázke kompozitných anotácií také riešenie, ktoré by bolo možné v krátkom čase pochopiť a následne implementovať v praxi

6 Najdôležitejšie súvisiace práce

Prvou zo súvisiacich prác, ktoré spomenieme, bude prvá z publikácií nástroja Daileon [1], ktorý ponúka dva prístupy. Prvý prístup je použiteľný ak ani jedna z anotácií, tvoriacich mienenú kompozitnú anotáciu, nie je viazaná na konkrétny programový element (triedu, metódu, ...). Druhý prístup, využívajúci bajtovú manipuláciu, je využiteľný práve v opačnom prípade ako prvý prístup.

Ďalšou súvisiacou prácou je publikácia AspectJ in Action [28], časť ktorej bola našou inšpiráciou pri návrhu nového prístupu ku kompozitným anotáciam pomocou aspektovo orientovaného programovania. Spomínaná časť opisovala spôsob, ako vo formule declare annotation navrhnúť vzor programového elementu aj s ohľadom na anotáciu, ktorá je nad ním deklarovaná.

V neposlednom rade, súvisiacou prácou môže byť aj dokumentácia, či publikácie k nástroju jazyka Java, Annotation processing tool [25], ktorý dokáže analyzovať a spracovať anotácie, podobne ako aspektovo orientovaný prístup.

Ďalšou súvisiacou prácou môže byť napríklad práca How Annotations are Used in Java: An Empirical Study [6], ktorá na základe štúdie hovorí o tom, v akej miere a nad akým typom elementov sú anotácie najčastejšie využívané.

Taktiež uznávame za vhodné spomenúť ako súvisiacu prácu publikáciu Introduction and derivation of annotations in AOP: Applying expressive pointcut languages to introductions [10], zaoberajúcu sa zavádzaním anotácií s pomocou aspektovo orientovaného programovania, čo je technika blízko spätá s touto prácou.

7 Záver

Cieľom tejto práce bolo opísať využitie anotácií ako jednej z foriem metaúdajov, opísať spôsob deklarácie anotácií, načrtnúť situáciu kedy sa deklarované anotácie môžu nad programovými elementmi opakovať. Od tejto situácie sme prešli k myšlienke kompozitných anotácií (a s nimi súvisiacim pojmom "anotačný vzor", zameriavajúcim sa na jednotlivé anotácie tvoriace kompozíciu) ako kompozície opakujúcich sa anotácií, ich potrebu v prípade často sa opakujúcich rovnakých skupín anotácií, opísali sme existujúce riešenie.

Uznali sme za potrebné dokázať potrebu zaoberať sa kompozitnými anotáciami, za týmto účelom sme pokračovali analýzou vybraných zdrojových kódov programov, dostupných z verejných repozitárov. Projekty sme vyberali na základe využitia programových rámcov JPA, EJB a JSF, známych využívaním anotácií. S výnimkou filtrovania projektov na základe týchto troch programových rámcov sme z nájdených projektov vyberali náhodne. S využitím pripraveného anotačného procesora, ktorý nám mal pomôcť automatizovať zdrojové kódy projektov sme získali informácie o opakovaní sa anotačných vzorov v projektoch, to sme následne opísali, s priblížením jednotlivých anotácií ktoré anotačný vzor tvorili, vzhľadom aj na potrebu poukázať na význam takéhoto jedného anotačného vzoru. Nakoniec sme vo fáze analýzy zdrojových kódov zhodnotili získané výsledky, štatisticky načrtli počet anotačných vzorov na jeden projekt a vyvodili z analýzy závery.

Pokračovali sme navrhnutím iného postupu, ako náhrady za už existujúce riešenie, Daileon, priblížili sme ako by bolo možné dostať sa k rovnakému výsledku s využitím aspektovo orientovaného programovania. Konkrétne sme využili rozšírenie programovacieho jazyka Java s názvom AspectJ, pomocou neho sme opísali postupy ako je s využitím aspektov možné vytvárať kompozitné anotácie, rozkladať ich na pôvodné anotácie, opísali sme postup ako by bolo možné vytvárať kompozíciu a dekompozíciu anotácií na základe parametra a nakoniec sme načrtli ako by mohlo vyzerat

distribuovanie anotácií na objekty triedy na základe anotácie nad touto triedou.

Myslíme si, že cieľ tejto práce bol dostatočne naplnený, myslíme si tiež, že práca na základe štatisticky signifikantnej vzorky náhodne vybraných projektov zodpovedala otázku, ktorá bola načrtnutá v odkazovaných prácach, teda či má zmysel riešiť opakujúce sa skupiny anotácií (na základe analýzy sme dokázali že to zmysel má), navrhli sme vlastné riešenie. Ďalším smerovaním výskumu v tejto oblasti by mohlo byť, ako bolo načrtnuté v podkapitole "Celkové vyhodnotenie analýzy všetkých uvedených projektov", zameranie sa na jeden konkrétny programový rámec, analyzovanie problému a prípadne implementácia riešenia problému či existuje taká skupina anotácií v rámci tohto jedného programového rámca, ktorá my už natívne v rámci tohto programového rámca mohla byť vyjadrená kompozitnou anotáciou, vzhľadom na časté využitie takejto skupiny anotácií u programátorov.

Zoznam použitej literatúry

[1] PERILLO, J. R. C. – GUERRA, E. M. – FERNANDES, C. T. 2009. *A Tool for Enabling Domain Annotations* Genova (Italy): European Conference on Object-Oriented Programming, 2009. ISBN:978-1-60558-548-2

[2] PERILLO, J. R. C. – GUERRA, E. M. – SILVA, J. O. – SILVEIRA, F. F. – FERNANDES, C. T. 2009. *Metadata Modularization Using Domain Annotations* Orlando (USA): The International Conference on Object Oriented programming, Systems, Languages and Applications, 2009.

[3] DAILEON: *Daileon - A tool for enabling domain annotations* Daileon, 2015. [cit. 2015-4-5]. Dostupné na internete: http://daileon.sourceforge.net/tutorial.html.

[4] THE ECLIPSE FOUNDATION: *Declare Annotation[online]* Ottawa: The Eclipse Foundation, 2014. [cit. 2014-6-27]. Dostupné na internete: https://eclipse.org/aspectj/doc/next/adk15notebook/annotations-declare.html.

[5] FLANAGAN, D. 2005. *Java in a Nutshell.* USA : O'Reilly Media Inc, 2005, 5. vydanie. 1256 s, ISBN 978-1-44936-668-1, s. 191–202

[6] ROCHA, H. – VALENTE, M. T. 2011. *How Annotations are Used in Java: An Empirical Study.* Eden Roc Renaissance Miami Beach, (USA): The 23rd International Conference on Software Engineering and Knowledge Engineering, 2011.

[7] LIEBERHERR, K. – ORLEANS, D. – OVLINGER, J. 2001. *ASPECT-ORIENTED PROGRAMMING WITH ADAPTIVE METHODS* Northeastern University, Boston, MA (USA): Magazine Communications of the ACM, Volume 44 Issue 10, 2001. DOI: 10.1145/383845.383855

[8] CHIBA, S. – NAKAGAWA, K. 2004. *Josh: An Open AspectJ-like Language*

Lancaster (UK): AOSD '04 Proceedings of the 3rd international conference on Aspect-oriented software development, 2004.

[9] XIE, T. – ZHAO, J. 2006. *A Framework and Tool Supports for Generating Test Inputs of AspectJ Programs* North Carolina State University, Raleigh, NC (USA), Shanghai Jiao Tong University, Shanghai (China): AOSD '06 Proceedings of the 5th international conference on Aspect-oriented software development, 2006.

[10] HAVINGA, W. – NAGY, I – BERGMANS, L. 2005. *Introduction and derivation of annotations in AOP: Applying expressive pointcut languages to introductions* Brussels, (Belgium): European Interactive Workshop on Aspects in Software (EIWAS), 2005.

[11] KICZALES, G. – HILSDALE, E. – HUGUNIN, J. – KERSTEN, M. – PALM, J. – GRISWOLD, W. G. 2001. *Getting started with ASPECTJ* USA : Magazine Communications of the ACM, Volume 44 Issue 10, 2001. DOI: 10.1145/383845.383858

[12] HANENBERG, S. – UNLAND, R. 2003. *Parametric introductions* Boston, Massachusetts, (USA) : AOSD '03 Proceedings of the 2nd international conference on Aspect-oriented software development, 2003. ISBN 1-58113-660-9

[13] HILSDALE, E. – HUGUNIN, J. 2004. *ECOOP 2004 — Object-Oriented Programming - Advice Weaving in AspectJ* Lancaster (UK) : AOSD '04 Proceedings of the 3rd international conference on Aspect-oriented software development, 2004. ISBN 1-58113-842-3

[14] SCHULT, W. – POLZE, A. 2002. *Aspect-oriented programming with C# and .NET* Washington, DC (USA) : Object-Oriented Real-Time Distributed Computing (ISORC 2002), 2002. ISBN 0-7695-1558-4

[15] KIM, H. 2006. *AspectC#: An AOSD implementation for C#* Trinity College

Dublin: Department of Computer Science , 2006.

[16] FILMAN, R. – ELRAD, T. – CLARKE, S. – AKIT, M. 2004. *Aspect-oriented software development*: OAddison-Wesley Professional, 2004, 1. vydanie. ISBN 0321219767

[17] EICHBERG, M. – SCHAFER, T. – MEZINI, M. 2005. *Using Annotations to Check Structural Properties of Classes*. UK : 8th International Conference, FASE 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, (UK). ISBN 978-3-540-25420-1

[18] EICHBERG, M. – MEZINI, M. 2005. *Alice: Modularization of Middleware Using Aspect-Oriented Programming* Linz, (Austria): 4th International Workshop, SEM 2004, ISBN 978-3-540-25328-0

[19] MANCINI, F. – HOVLAND, D – MUGHAL, K. A. 2010. *Investigating the Limitations of Java Annotations for Input Validation* Krakow, (Poland): ARES'10 International Conference on Availability, Reliability and Security, 2010, ISBN 978-1-4244-5879-0

[20] NOSÁĽ, M. 2012. *Atribútovo-orientované programovanie a metaprogramovanie* Košice, (Slovakia): Písomná práca k dizertačnej skúške, 2012.

[21] PHILLIPS, A. 2009. *@Composite – Macro Annotations for Java* USA: OOPSLA '09 Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications, 2009, ISBN: 978-1-60558-768-4

[22] NOSÁĽ, M., PORUBAN, J. 2014. *XML to Annotations Mapping Definition with Patterns.* Slovakia: Computer Science and Information Systems, Vol. 11, No. 4, 1455–1477, 2014. DOI: 10.2298/CSIS130920049N

[23] KOLLÁR, J., PORUBAN, J., VÁCLAVÍK, P. 2005. *SEPARATING CON-*

CERNS IN PROGRAMMING: DATA, CONTROL AND ACTIONS Slovakia: Computing and Informatics, Vol. 24, 2005, 441–462, 2005.

[24] MAJUMDAR, D., BHATTACHARYA, S. 2010. *Aspect Oriented Requirement Engineering: A Theme Based Vector Orientation Model* INFOCOMP Journal of Computer Science, vol. 9, no.1, pp.61-69, 2010.

[25] ORACLE: *Getting Started with the Annotation Processing Tool (apt)[online]* Oracle, 2014. [cit. 2015-2-8]. Dostupné na internete: http://docs.oracle.com/javase/1.5.0/docs/guide/apt/GettingStarted.html.

[26] ORACLE: *Annotation Type Id[online]* Oracle, 2015. [cit. 2015-3-22]. Dostupné na internete: http://docs.oracle.com/javaee/6/api/javax/persistence/Id.html.

[27] ORACLE: *Annotation Type GeneratedValue[online]* Oracle, 2015. [cit. 2015-3-22]. Dostupné na internete: http://docs.oracle.com/javaee/6/api/javax/persistence/GeneratedValue.html.

[28] LADDAD, R. 2010. *AspectJ in Action.* USA : Manning Publications Co., 2010, 2. vydanie. 495 s, ISBN 978-1-933988-05-4, s. 130–132