Analýza a realizácia vybraných návrhových vzorov použitím AspectJ

1. Návrhové vzory OOP

„Koľkokrát sme mali návrhové *déjá vu* – t. j. pocit, že sme už problém kedysi riešili, ale presne nevieme kde a ako? Keby sme si mohli vybaviť podrobnosti minulého problému i jeho riešenie, potom by sme mohli tieto skúsenosti znovu použiť namiesto ich opätovného objavovania.“ [1]

„Každý vzor popisuje problém, ktorý sa v našom prostredí neustále vyskytuje. Potom popisuje jadro riešenia daného problému tak, že nám umožňuje toto riešenie používať aj miliónkrát bez toho, aby sme to urobili dvakrát rovnakým spôsobom.“

Cristopher Alexander, stavebný architekt

„Návrhové vzory sú doporučené postupy riešenia často sa vyskytujúcich úloh. Môžete ich považovať za vzorce, ktoré použijete pri návrhu architektúry budúcej aplikácie.“ [2]

Myslím, že predchádzajúce citáty presne definujú na čo vlastne sú dobré, užitočné a prečo sa vôbec oplatí baviť o návrhových vzoroch. Návrhové vzory sú odporúčané postupy nielen v programovaní, ale aj v iných sférach ľudskej činnosti ako napr. stavbárstvo. Návrhové vzory pri správnom pochopení a dôkladne premyslenom použití dokážu výrazne zlepšiť kvalitu a prehľadnosť zdrojových kódov a tak nie je prekvapivé, že profesionálne firmy zaoberajúce sa návrhom a tvorbou softvéru považujú ich znalosť za jednu zo základných schopností.

Návrhové vzory boli po prvý krát spoločne sformulované a zatriedené v publikácii od GoF (Gang of Four) [1]. Na tvorbe tejto publikácie sa podieľalo veľa ľudí a táto kniha sa pre svoju kvalitu stala de facto štandardom pri procese pochopenia a učenia sa návrhových vzorov. Publikácia rozdeľuje návrhové vzory do 3 typov:

- tvorivé vzory
- štrukturálne vzory
- vzory chovania

Každá z tých skupín obsahuje množinu návrhových vzorov, ktoré sa vyznačujú spoločnými vlastnosťami, resp. účelom použitia.

2. Návrhové vzory AOP

Základnou úlohou aspektovo orientovanej paradigmy bolo riešiť základný problém objektovo orientovanej paradigmy t. j. pretínanie zámerov[1]. AOP teda vychádza z OOP a je teda akousi nadstavbou OOP. Ak sa bavíme o návrhových vzoroch v rámci AOP, dali by sa tieto vzory rozdeliť podľa primárneho vzťahu k paradigme do dvoch pomyselných skupín:

- vzory, ktoré vznikli prostým prepísaním GoF vzorov do AOP využívajúc výhodné konštrukcie AOP [3]
- vzory, ktoré sa formovali postupne so vznikom AOP [6]

Mojím cieľom záujmu bola práve druhá skupina vzorov, ktorá prináša zaujímavé riešenie problémov, ktoré by si v rámci OOP vyžiadali omnoho väčšie úsilie na ich implementáciu, resp. by boli len veľmi ťažko implementovateľné.

Tak ako OOP vzory sa delia do skupín, aj AOP vzory sa delia a to konkrétne do 2 skupín líšiacich sa predovšetkým konštrukciami, ktoré používajú:

- **Pointcut design patterns**
- **Advice design patterns**

Prvá skupina vzorov sa zaoberá výhodným návrhom *pointcutov*[2], pričom tieto vzory sú často základným stavebným kameňom ostatných vzorov. Medzi tieto návrhové vzory zaraďujeme:

- **Wormhole** – Tento návrhový vzor spája volajúceho a volaného v rámci volania metód takým spôsobom, že spoločne zdieľajú ich informačný kontext. Vytvára priame spojenie medzi dvoma úrovňami v rámci volania metód. Je veľmi vhodný v prípade nutnosti odovzdávania si kontextu, pričom nie je nutné vkladať do každej metódy v hierarchii volania parametre, alebo používať nejaký druh globálneho úložiska.
- **Participant** – Vo všeobecnosti sa aspekty väčšinou snažia pridať nejaké správanie triedam takým spôsobom, aby si toho triedy neboli vedomé. V tomto návrhovom vzore je to presne naopak. Aspekt sa snaží dosiahnuť manažovanie tried. Napríklad ak *advice*[3] ovplyvňuje iba metódy s istými charakteristikami, nie je možné rozhodnúť sa iba podľa ich mien, či ich zahrnúť do *pointcutu* alebo nie.
- **Border Control** – Tento návrhový vzor sa používa na logické oblasti v rámci aplikácie. Tieto oblasti sú ďalej znovu použité ďalšími aspektami na zaistenie aplikácie len v rámci správnych oblastí.

[1] crosscutting concerns, pozn. aut.
[2] pointcut, niekde prekladaný ako *bodový prierez*, pozn. aut.
[3] advice, niekde prekladané ako odporúčanie, pozn. aut.

Druhá skupina vzorov sa zaoberá výhodným návrhom adviceov. Tieto návrhové vzory sú podľa môjho názoru komplikovanejšie, ale dokážu riešiť určité problémy veľmi elegantne. Medzi tieto návrhové vzory zaraďujeme:

- **Cuckoo's Egg** – Tento návrhový vzor je veľmi jednoduchý, ale súčasne veľmi mocný. Ukazuje silu AOP. Používa sa na kontrolu, resp. zmenu objektov vracaných volaním konštruktora, čiže je možné vracať rôzne typy podľa určitých závislostí.
- **Director** – Tento návrhový vzor môže byť použitý na definovanie určitého správania neurčitého počtu, resp. typov tried. Určitá rola môže byť implementovaná bez priamej vedomosti na aký typ triedy sa bude aplikovať.
- **Policy** – Hlavnou myšlienkou tohto návrhového vzoru je definovať určité obmedzenia alebo pravidlá v rámci aplikácie, pričom pravidlami môžu byť rôzne rady až po varovania.
- **Worker Object** – Tento návrhový vzor má všeobecné využitie. Môže byť použitý v prípade potreby postupného volania metód, alebo ak si volanie metód vyžaduje volanie v samostatných vláknach.
- **Exception Introduction** – V niektorých prípadoch pri implementácii aspektov je nutné odchytávať kontrolované výnimky[4] vo vnútri adviceov. To je bežné pri použití metód vyhadzujúcich spomínané výnimky v adviceoch. Princípom tohto vzoru je odchytávať tieto výnimky a jednoducho ich zaobaľovať do špecifickejších nekontrolovaných[5] výnimiek, ktoré sú následne vyhadzované vyššie v hierarchii volania a môžu tam byť spracované.

Zo všetkých týchto návrhových vzorov som sa rozhodol vybrať a bližšie spracovať: *Wormhole*, *Border control* a *Cuckoo's egg.*

2.1 Border Control

Účel: chceme definovať dôležité oblasti aplikácie tak, aby naše aspekty mohli využívať tieto oblasti a ďalej chceme zaistiť aby boli aplikované len na správne oblasti aplikácie.

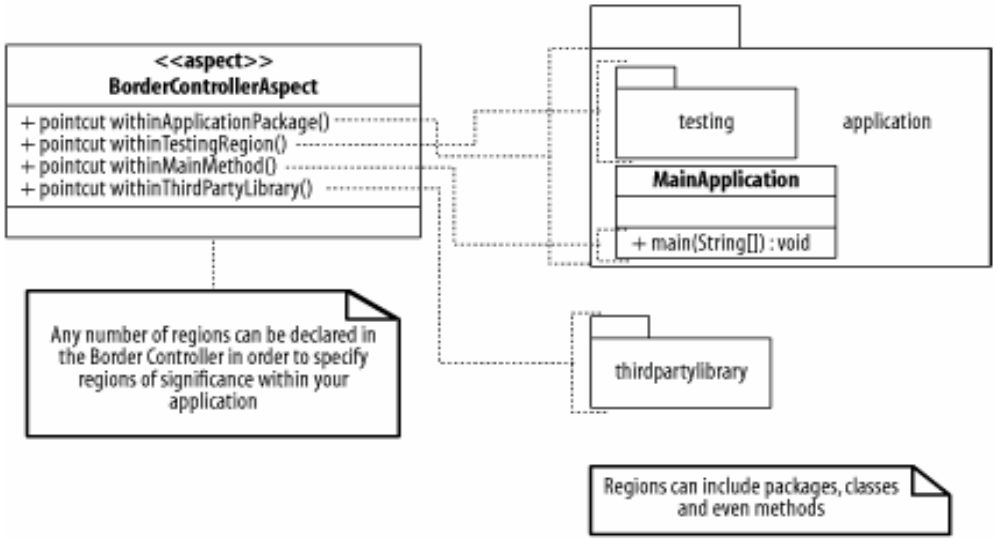
Riešenie v OOP: na podobné riešenie v OOP som neprišiel, čo však nie je problémom. Tento návrhový vzor je často využívaný inými zložitejšími AOP vzormi.

Popis funkčnosti: princípom funkčnosti je definovanie viacerých pointcutov podľa potreby, pričom tieto pointcuty sa týkajú balíkov, tried a metód. Definované pointcuty budú ďalej používané v iných aspektoch na obmedzenie použitia len vo vhodných častiach aplikácie.

---

[4] *checked exceptions*. V Jave ide o klasický prípad výnimiek dediacich od java.lang.Exception, ktoré je nutné uvádzať ich do hlavičky metódy, v prípade, že ich metóda neodchytáva.

[5] *unchecked exceptions*: Nie je nutné ich uvádzať do hlavičky metódy, v prípade ich neodchytenia sa automaticky propagujú hierarchiou volaní. V Jave ide o výnimky dediace od java.lang.RuntimeException.



Obr. 1 Štruktúra návrhového vzoru Border control

Úryvok kódu:

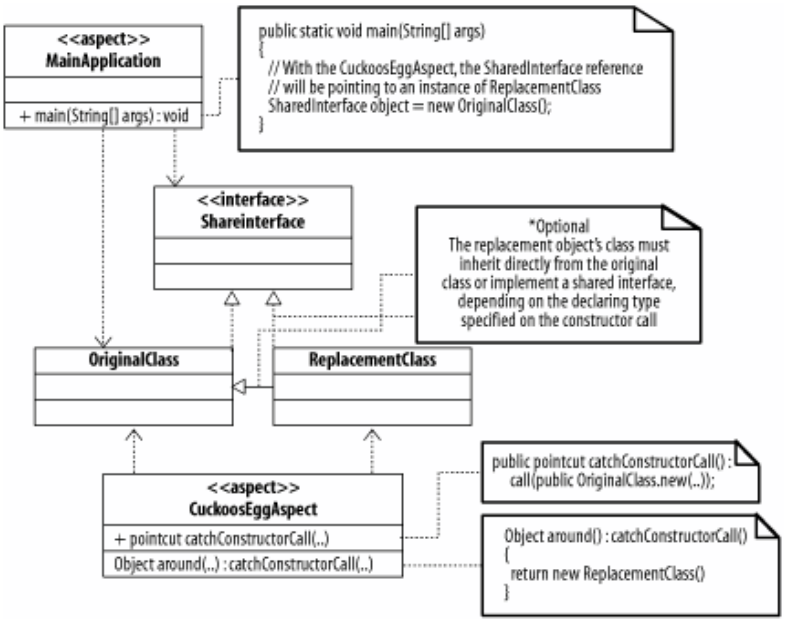
```
public aspect BorderControllerAspect  {

  public pointcut vramciMojejAplikacie( ): within(pivnaStory.*);

  public pointcut vramciInychKniznic( ): within(org.aspectj.*);

  public pointcut vramciMainMetodyMojejAplikacie( ): withincode(public void
pivnaStory.Tester.main(..));
}
```

Výhody a použitie:

- poskytuje mechanizmus na definovanie znovu použiteľných aspektov
- vhodný pri častej, resp. plánovanej zmene štruktúry aplikácie, kedy stačí zmeniť len tento aspekt a zmeny sa prejavia globálne
- vhodný ako základ na definíciu štruktúry aplikácie pri použití viacerých návrhových vzorov.

2.2 Cuckoo's egg

Účel: chceme mať možnosť ovplyvniť typ skutočného objektu vráteného konštruktorom.



Obr. 2 Štruktúra návrhového vzoru Cuckoo's egg

Riešenie v OOP: V OOP nemáme priamu možnosť ovplyvňovať typ skutočného objektu vráteného konštruktorom. Jediným možným riešením je použitie návrhového vzoru *Factory Method* [1, 2], čím síce dokážeme ovplyvniť typ vráteného objektu, ale nebude to riešené na úrovni konštruktora.

Popis funkčnosti: princípom funkčnosti je definovanie aspektu, ktorý bude odchytávať volanie určitého konštruktora, pričom volanie bude nahradené kódom definovaným v aspekte. V aspekte je možné využívať, resp. aj ignorovať argumenty dodané pri volaní konštruktora a v závislosti na ich hodnote vracať iné skutočné typy objektov. Tento mechanizmus je možný vďaka polymorfizmu. Nie je teda možné vracať akýkoľvek typ objektu, pretože medzi pôvodným objektom a nahradeným objektom musí existovať vzťah zaručujúci zameniteľnosť objektov. Zvyčajne teda nahradzované typy objektov dedia od pôvodného objektu. Ak pôvodný objekt implementuje určité rozhrania a my voláme metódy výhradne definované v týchto rozhraniach, je potom možné vracať akýkoľvek typ implementujúci dané rozhranie, resp. množinu rozhraní. V takomto prípade nie je nutné vždy implementovať všetky rozhrania (*Serializable*, *Cloneable*, ...).

Úryvok kódu:

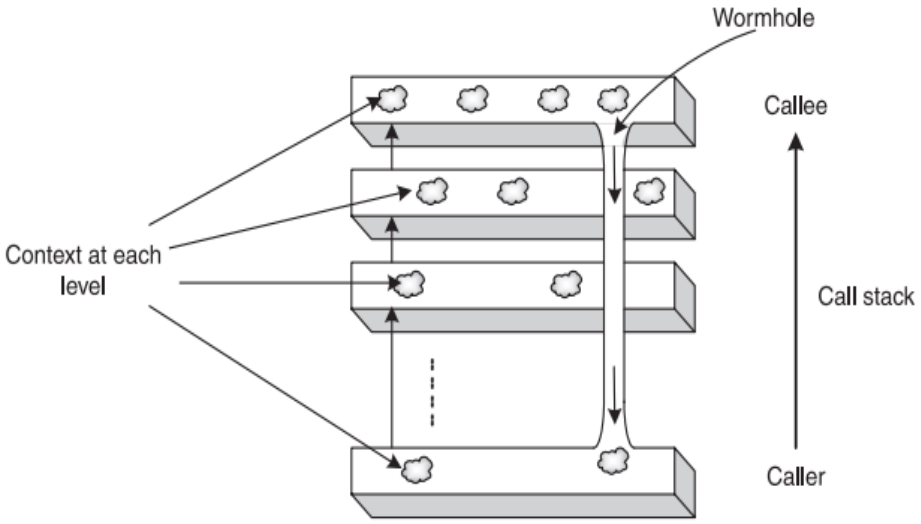
```
public aspect CuckoosEggAspect {

      public pointcut capujeme(TypVycapnika vycapnik): call(public
PivovyKrigel.new(..))                    &&                    args(vycapnik)&&
BorderControllerAspect.vramciMojejAplikacie();

      PivovyKrigel around(TypVycapnika vycapnik): capujeme(vycapnik) {
            switch (vycapnik) {
                case SLABY :
                  return new MalyPivovyKrigel();

                case NORMALNY :
                  return new NormalnyPivovyKrigel();

                case BRUTALNY :
                  return new ExtraPivovyKrigel();

                default :
                  throw new IllegalStateException("Neidentifikovateľný typ
 výčapníka ...");
            }
      }
}
```

Výhody a použitie:

- dokážeme vracať iné skutočné typy objektov ako je typ volaného konštruktora
- parametre konštruktora môžeme využívať alebo ignorovať
- vhodný na implementáciu mock objektov
- vhodný na poskytovanie proxy objektov namiesto reálnych inštancií tried

2.3 Wormhole

Účel: chceme odovzdať kontext volajúcej metódy volanej metóde v rámci hierarchie volania metód



Obr. 3 Štruktúra návrhového vzoru Wormhole

Riešenie v OOP: v rámci OOP by sme nemali veľa možností ako to dosiahnuť. Jednoduchým riešením by bolo odovzdávať kontext v podobe parametrov metód postupne cez celú hierarchiu volania metód. Problémom by však bola tzv. „API pollution" (doslova *znečistenie kódu)*, pretože by sme v metódach deklarovali parametre, ktoré by sme teoreticky vôbec nemali nepotrebovať. Okrem toho by existovala možnosť nevhodnej zmeny hodnôt týchto parametrov, čo by malo samozrejme neblahé dôsledky na funkčnosť programu

Popis funkčnosti: princípom funkčnosti je definovanie dvoch pointcutov, jeden pre volajúcu metódu s kontextom a druhý pre volanú metódu. Keďže našou snahou je sprístupniť kontext volanej metóde, ďalším krokom je definovanie ďalšieho pointcutu odchytávajúceho volanie našej volanej metódy, ale len v rámci tela volajúcej metódy. Následne pre tento pointcut definujeme advice poskytujúci kontext.

Úryvok kódu:

```
public aspect WormholeAspect {

  public pointcut vykazovanieTrzieb(Zamestnanec zamestnanec):
    execution(* Zamestnanec.vykazatTrzby(..)) && this(zamestnanec);

  public pointcut kontrolaTrzieb(double trzby) :
    execution(* Riaditel.validujTrzby(..)) && args(trzby);

  public pointcut wormhole(double trzby, Zamestnanec zamestnanec):
    kontrolaTrzieb(trzby) && cflow(vykazovanieTrzieb(zamestnanec));

  before(double trzby, Zamestnanec zamestnanec)
 : wormhole(trzby, zamestnanec) {
    // a hura mozeme kontrolovat trzby konkretneho zamestnanca...
  }
}
```

Výhody a použitie:

- nie je nutné predávanie kontextu v rámci parametrov
- vhodný na priamy prístup k volajúcemu objektu a jeho kontextu
- vhodný na jednoduché pridávanie/odoberanie ďalších kontextových informácií

3. Záver

Vďaka vypracovaniu tejto práce som si získal prehľad o návrhových vzoroch používaných v aspektovo orientovanom programovaní, tri z nich som bližšie analyzoval a vyskúšal si ich jednoduchú implementáciu. Počas tohto štúdia som prišiel na to, že väčšina týchto návrhových vzorov rieši problémy, ktoré sa nedajú vyriešiť v objektovo orientovanom programovaní, resp. dajú sa riešiť, ale nie tak čistým spôsobom a väčšinou je potrebné aj nepomerne väčšie množstvo kódu. Implementácia je však veľmi jednoduchá a tak ukazuje silu a flexibilitu aspektovo orientovaného programovania.

4. Zdroje

[1] GAMMA, Erich – HELM, Richard – JOHNSON, Ralph – VLISSIDES, John: Návrh programů pomocí vzorů. Stavební kameny objektově orientovaných programů, Grada, 2003

[2] PECINOVSKÝ, Rudolf: Návrhové vzory. 33 vzorových postupů pro objektové programování, Computer Press, 2007

[3] http://hannemann.pbwiki.com/Design-Patterns

[4] MILES, Russel: AspectJ Cookbook, O'Reilly, 2004

[5] LADDAD, Ramnivas: AspectJ in Action. Practical Aspect-Oriented Programming, Manning, 2003