EKONOMICKÁ UNIVERZITA V BRATISLAVE
FAKULTA HOSPODÁRSKEJ INFORMATIKY

Evidenčné číslo: 17300/ I/2012/2137693172

ASPEKTOVÉ PROGRAMOVANIE

Diplomová práca

2012 **Eva Mačurová, Bc.**

EKONOMICKÁ UNIVERZITA V BRATISLAVE
FAKULTA HOSPODÁRSKEJ INFORMATIKY

ASPEKTOVÉ PROGRAMOVANIE

Diplomová práca

Študijný program: 6258 Manažérske rozhodovanie a informačné technológie

Študijný odbor: 3.3.24 Kvantitatívne metódy v ekonómii

Školiace pracovisko: Katedra aplikovanej informatiky

Vedúci záverečnej práce: Igor Bandurič, Ing. PhD

Bratislava 2012 **Bc. Eva Mačurová**

Zadanie záverečnej práce

Čestné vyhlásenie

Čestne vyhlasujem, že záverečnú prácu som vypracovala samostatne a že som uviedla všetku použitú literatúru.

Dátum: **.....**

Bc. Eva Mačurová

Poďakovanie

Moja vďaka za pomoc pri písaní tejto práce patrí Ing. Igorovi Banduričovi, PhD.

ABSTRAKT

MAČUROVÁ, Eva: *Aspektové programovanie.* – Ekonomická univerzita v Bratislave. Fakulta hospodárskej informatiky; Katedra aplikovanej informatiky. – Vedúci záverečnej práce: Ing. Igor Bandurič, PhD. – Bratislava: FHI, 2012, 62 s.

Cieľom práce Aspektové programovanie je definovať základné charakteristiky a pojmy v aspektovom programovaní. Popísať programové vzory a na ukážkach demonštrovať spôsob použitia aspektov pri programovaní.

Práca je rozdelená do 3 kapitol. Obsahuje 47 obrázkov a 9 tabuliek.

Prvá kapitola Súčasný stav riešenej problematiky doma a v zahraničí je venovaná úvodu do problematiky. Zaoberá sa opisom základov aspektovo-orientovanej paradigmy, aspektovo-orientovaného programovacieho jazyka AspectJ a návrhových vzorov.

V ďalšej časti Cieľ práce, metodika práce a metódy skúmania sú charakterizované hlavné a vedľajšie ciele tejto práce a postup pri jej zhotovovaní.

Záverečná kapitola Výsledky práce a diskusia je venovaná opisu vybraných návrhových vzorov v objektovo-orientovanej a v aspektovo-orientovanej verzii.

Kľúčové slová:
aspektovo-orientované programovanie, AspectJ, návrhové vzory

ABSTRACT

MAČUROVÁ, Eva: *Aspect programming.* – Economic University in Bratislava. Faculty of Business Inforamtics; Department of Business Inforamtics. – Diploma thesis supervisor: Ing. Igor Bandurič, PhD. – Bratislava: DBI, 2012, 62 p.

The main goal of the thesis Aspect programming is to define basic characteristics and concepts in aspect programming, to describe design patterns and by using examples to show the ways of usage of aspect in programming.

The diploma thesis is devided to three chapters. It contains 47 pictures and 9 tables.

First chapter Present state of solved problems at home and abroad is concentrated to intorduction to the main issues. It explains the basics of aspect-oriented paradigm, aspect-oriented programming language AspectJ and design patterns.

In next chapter The goal, methodics of work and methods of research are characterised primary and secondary goals and the way of perhorming.

The last chapter The results of the work and discusion is centred on description of several design patterns in object-oriented and aspect-oriented form.

Key Words

aspektovo-oriented programming, AspectJ, design patterns

O B S A H str.

ZOZNAM ILUSTRÁCIÍ A ZOZNAM TABULIEK

Ilustrácie

Tabuľky

ZOZNAM SKRATIEK A ZNAČIEK

angl.	anglicky
AOP	aspektovo-orientovaná paradigma, resp. aspektovo-orientované programovanie
OOP	objektovo-orientovaná paradigma, resp. objektovo-orientované programovanie
AJDT	angl. AspectJ Development Tools
JDT	angl. Java Developemnt Tools
AOSD	angl. Aspect-Oriented Software Development
AOSE	angl. Aspect-Oriented Software Engineering
TIOBE	názov spoločnosti, angl. The Importance of Being Ernest
SoC	angl. Separation of Concerns
JPM	angl. join point model
UML	angl. Unified Modelling Language
PARC	Pablo Alto Research Center
GoF	angl. Gang of Four
IDE	angl. Integrated development environment
EPL	angl. Eclipse Public License
JVM	angl. Java Virtual Machine
ECOOP	angl. European Conference on Object-Oriented Programming
PDD	angl. Pattern-Driven Development
POSD	angl. Pattern-Oriented Software Development

Úvod

Záverečná práca Aspektové programovanie sa zaoberá základnými charakteristikami aspektovo-orientovanej paradigmy, vzájomnou previazanosťou a vzťahmi medzi aspektovo-orientovanou paradigmou a objektovo-orientovanou paradigmou a softvérovými návrhovými vzormi ako potencionálnym nástrojom na porovnávanie uvedených paradigiem. Opis vybraných návrhových vzorov je jedným z primárnych cieľov tejto práce.

Ako programovací jazyk je používaný aspektovo-orientovaný jazyk AspectJ. Dôvodom pre túto voľbu bol jednoznačne fakt, že AspectJ je implementovateľný do jazyka Java, ktorý je v súčasnosti veľmi rozšírený a populárny.

Predmet záverečnej práce Aspektové programovanie je možné rozčleniť do troch základných úrovní. Prvou úrovňou je aspektovo-orientované programovanie a vývoj softvéru obsahujúcou základný kontext pre druhú úroveň, ktorou je charakteristika programovacieho jazyka AspektJ ako jedeného z aspektovo-orientovaných jazykov. Týmito dvoma úrovňami sa zaoberá 1. kapitola *Súčasný stav riešenej problematiky doma a v zahraničí*.

Tretiu úroveň predstavujú vybrané návrhové vzory implementované v jazyku AspectJ, ktoré tvoria najvýznamnejšiu časť záverečnej práce Aspektové programovanie a ktorým je venovaná 3. kapitola *Výsledky práce a diskusia*.

2. kapitola *Cieľ práce, metodika práce a metódy skúmania* uvádza postupy riešenia.

Tretia kapitola obsahuje opis siedmich návrhových vzorov. Nebolo možné vtesnať väčšie množstvo vzorov do tejto práce s obmedzeným rozsahom, preto bolo vybraných sedem reprezentatívnych vzorov. Príbuznosť jazykov AspectJ a Java bola kľúčová pri rozhodovaní, ktoré návrhové vzory majú byť spracované.

V tejto práci je zachované tradičné rozdelenie návrhových vzorov do troch skupín na: vytvárajúce, štrukturálne a behaviorálne. Vybranými návrhovými vzormi, ktoré sú podrobne analyzované v 3. kapitole, sú: Prototyp, Singleton, Flyweight, Composit, Proxy, Strategy a Observer.

Okrem popisu návrhových vzorov, ako jedného z primárnych cieľov tejto práce, je venovaná pozornosť aj porovnávaniu objektovo-orientovaných implementácií návrhových vzorov s aspektovo-orientovanými implementáciami, stanovenie ich rozdielností a súčasne všeobecnej štruktúry opisu vzoru.

Dôvodmi pre takéto porovnávanie nie je len zistenie, ktorý variantu je výhodnejšie použiť, ale aj posúdenie relevantnosti takýchto porovnávaní a ich možné využitie pre porovnávanie celých programovacích paradigiem, prinajmenšom pri porovnávaní takých blízkych paradigiem, akými sú objektovo-orientovaná a aspektovo-orientovaná paradigma, pri použití jazykov Java a AspectJ.

Obrazne, na jednej strane váh sa nachádza aspektovo-orientovaná paradigma a jazyk AspectJ, na druhej strane objektovo-orientovaná paradigma a jazyk Java a uprostred návrhové vzory, ktorých úlohou je rozhodnúť o výsledku.

1. Súčasný stav riešenej problematiky doma a v zahraničí

Už tradične sa aspektovo-orientované programovanie charakterizuje ako riešenie určitých nedostatkov objektovo-orientovanej paradigmy, čo vzbudzuje nemalý záujem o túto tému a súčasne predstavuje aj jeden z dôvodov pre napísanie práce, akou je Aspektové programovanie. Objektovo-orientované programovnie je nesporne fenoménom dnešnej doby, ktorý sa neprestajne mení, zdokonaľuje a vyvíja. Aspektovo-orientované programovanie je často označované ako nové evolučné štádium, ktoré vzišlo z objektovo-orientovanej paradigmy.

Holandská spoločnosť TIOBE Software zaoberajúca sa zisťovaním miery používanosti jednotlivých programovacích jazykov, dospela k aprílu 2012 k výsledkom uvedeným v Tabuľke č. 1 [16].

Kategórie	Hodnotenie k apr. 2012	Zmena oproti apr. 2011
Objektovo-orientované jazyky	58%	+0,4%
Procedurálne jazyky	35,9%	-1,1%
Funkcionálne jazyky	3,9%	-0,3%
Logické jazyky	2,1%	+1,0%

Tabuľka č. 1 Miera používanosti programovacích jazykov

Tieto hodnoty poukazujú na značnú obľúbenosť objektovo-orientovanej paradigmy a súčasne na veľký potenciál aspektovo-orientovanej paradigmy. Medzi oblasti využitia aspektovo-orientovaného programovania pri tvorby softvéru, zväčša pri rozsiahlejších projektoch, okrem iného patrí:

- kontrola chýb a ich odstránenie
- synchronizácia
- bezpečnosť
- autentifikácia používateľov
- kontextovo riadené správanie
- optimalizácia výkonu
- monitoring
- prihlasovanie, logging
- podpora debugovania
- multi-objektové protokoly

- uplatňovanie štandardov
- feature variations.

Spoločnou vlastnosťou väčšiny uvedených oblastí je, že, ak sú používané v objektovo-orientovanom programovaní, vyskytujú sa v rámci jedného projektu veľmi často a opakovane, čo môže viesť ku rôznym komplikáciám, ktoré sa snaží odstrániť aspektovo-orientovaná paradigma. Znovupoužiteľnosť je preto základným faktorom v aspektovo-orientovanom programovaní.

Obrázok č. 1 Predmet prvej kapitoly

Znovupoužiteľnosť je prítomná v tejto práci ešte v podobe návrhových vzorov. Cieľom tejto kapitoly je ozrejmiť kontext pre návrhové vzory implementovateľné v jazyku AspectJ. Postupne sa prechádza od najvšeobecnejšej témy aspektovo-orientovanej paradigmy, ktorá predstavuje východisko, cez programovací jazyk ako prostriedok implementácie návrhových vzorov až po tému samotných návrhových vzorov.

1.1 Aspektovo-orientovaná paradigma

1.1.1 Definícia AOP

Pri opise aspektovo-orientovanej paradigmy (AOP) môžeme vychádzať z viacerých definícií nachádzajúcich sa v odbornej literatúre. Pre potreby tejto práce je vhodná definícia uvedená v Encyklopédií informačnej vedy a informačných technológií.

„Aspektovo-orientované programovanie poskytuje špeciálne jazykové konštrukcie nazývané aspekty, ktoré sú určené na modularizáciu pretínajúcich záležitostí v konvenčných programových štruktúrach (napr. záležitosť, ktorá sa rozprestiera cez hierarchie tried v objektovo-orientovaných programoch). Prekladač nazývaný weaver je zodpovedný za zlúčenie doplnkového kódu špecifikovaného v aspektovo-orientovanom jazyku do základného jazyku v čase kompilovania.“ [1, s. 4292]

Gregor Kiczales, popredná autorita v oblasti aspektovo-orientovaného programovania a spolutvorca najpoužívanejšieho aspektovo-orientovaného jazyka AspectJ, definoval AOP v rozhovore Diskusia o aspektoch AOP [25, s. 1] nasledujúcim spôsobom.

„AOP je nová evolúcia v technológiách určených na separáciu koncernov – technológia, ktorá umožňuje, aby návrh a implementácia boli štruktúrované a aby odrážali spôsob, akým vývojári chcú myslieť o systéme. AOP je postavené na existujúcich technológiách a poskytuje dodatočné mechanizmy, ktoré umožňujú ovplyvniť implementáciu v pretínajúcom spôsobe. V AOP, jeden aspekt môže prispieť k implementácií množstva procedúr, modulov a objektov.

Prínos môže byť rovnaký, napr. poskytnutím prístupového správania, ktoré by mali všetky procedúry v určitom rozhraní nasledovať; alebo môže byť rozdielny, napr. implementovaním dvoch strán protokolu medzi dvomi rozličnými triedami.

Ako pri ostatných technológiách zameraných na pretínajúce koncerny, cieľom AOP je dosiahnuť, aby dizajn a kód boli viac modulárne, čo znamená, že koncerny sú lokalizované skôr ako roztrúsené a zabezpečiť dobre definované rozhrania s ostatnými časťami systému. Toto nám poskytuje obvyklé prínosy, vrátane umožnenia relatívnej izolácie koncernov, spravením ich (ne)oddeliteľnými, prístupnými k oddelenému vývoju, atď.

1.1.2 Základy koncepcie AOP

Aspektovo-orientovaná paradigma vo všeobecnosti poskytuje päť základných koncepcií určených na modularizáciu pretínajúcich koncernov.

Koncepcie AOP

- **model bodov spájania** (angl. join point model - JPM)
  - definujúci miesta, kde sa majú dodatočné zlepšenia, aspekty, v podobe pretínajúcich sa koncernov nachádzať
  - **bod spájania** (angl. join point) – konkrétny bod v exekúcií programu, miesto, kde je možné aplikovať aspekty
- **bodový prierez** (angl. pointcut)
  - prostriedok na identifikáciu bodov spájania, súbor bodov spájania
  - súčasť aspektu, ktorá vymedzuje aplikovateľnosť aspektu, t. j. miesto a čas, kedy sa má vykonať funkcionalita aspektu obsiahnutá v advice
- **videnie** (angl. advice)
  - prostriedok na špecifikáciu správania v bodoch spájania
  - súčasť aspektu, ktorá obsahuje funkcionalitu aspektu, t. j. čo sa má vykonať, keď je aspekt použitý
- **aspekt** (angl. aspect)
  - modularizovaný pretínajúci koncern
  - zapuzdrená jednotka kombinujúca špecifikácie bodov spájania so správaním
- **metódy na pripojenie jednotiek, aspektov, do programu**
  - angl. weaving
  - aplikácia advice do všetkých bodov spájania definovaných v bodových prierezoch

Niekedy sa v angličtine používa skratka PCA pointcut–advice, pre vyjadrenie konštrukcie pozostávajúcej z bodového prierezu a videnia.

1.1.3 História AOP

Hoci aspektovo-orientovaná paradigma je pomerne nová a nie natoľko zaužívaná, myšlienky, ktoré tvoria jej podstatu, sa objavujú od počiatkov programovania.

V sedemdesiatych rokoch minulého storočia bolo publikovaných viacero diel predstavujúcich východisko pre túto problematiku. V odborných publikáciách sa najčastejšie spomínajú práca D. L. Parnasa O kritériách používaných na dekompozíciu systémov do modulov (r. 1972, [2]) a práce E. W. Dijkstra O úlohe vedeckého myslenia (r. 1974, [3]), v ktorej je prvýkrát použité názvoslovie separácia záležitostí (resp. koncernov), angl. separation of concerns (SoC), a Disciplína programovanie (r. 1976, [4]), pričom obom spomenutým autorom sa striedavo pripisuje prvenstvo.

Princíp separácie koncernov je základom programovania vo všeobecnosti. Koncerny je možné separovať viacerými spôsobmi a vo viacerých smeroch, okrem iného napr. vertikálne, horizontálne, dátovo, behaviorálne či aspektovo. Aspektová separácia koncernov, nazývaná separácia pretínajúcich koncernov (angl. crosscutting concerns) je fundamentálnym prvkom aspektovo-orientovaného programovania.

Problematika pretínajúcich koncernov sa začala formalizovať v polovici deväťdesiatych rokov. Spoločnosť Xerox Pablo Alto Research Center (PARC) sa venovala vývoju aspektovo-orientovaného softvéru a medzi výsledky jej snaženia patrí aj aspektovo-orientovaný jazyk AspectJ.

História aspektovo-orientovanej paradigmy a aspektovo-orientovaného jazyka AspectJ sú úzko prepojené. Prvýkrát bola AOP formálne predstavená v roku 1997 na Európskej konferencii o objektovo-orientovanom programovaní (ECOOP) Gregorom Kiczalesom v príspevku Aspect-Oriented Programming [9].

1.1.4 Súčasný stav AOP

Na to, aby si aspektovo-orientovaná paradigma našla svoje pevné miesto a stala sa jednou z automaticky zvažovaných alternatív pri tvorbe softvérových projektov, je potrebný ešte nejaký čas.

Dôvody, pre ktoré nenastáva veľký rozmach tejto technológie, sú nasledovné.

V prvom rade je to osvojiteľnosť tejto paradigmy. Pri tvorbe nových systémov je nevyhnutné si osvojiť inú formu modularizácie, ktorá je častokrát komplikovanejšia ako napr. objektovo-orientovaná, a ktorá neprináša skrátenie času implementácie softvéru, pretože si vyžaduje preskúmanie všetkých oblastí, kde má byť určitý aspekt použitý,

súčasne. Proces vyčleňovania aspektov z neaspektového prostredia sa nazýva aspect mining a vyžaduje si značné programátorské a modelovacie zručnosti.

Ďalším dôvodom je efektívne použitie hlavne pri väčších projektoch, ktoré majú byť v prevádzke dlhé roky a už pri ich plánovaní sa počíta so vzniknutím potreby zmeny v budúcnosti, napr. Enterprise aplikácie, aplikačné frameworky, IDE-kompilátor integrácia. Takéto projekty sú doménou veľkých spoločností majúcich možnosť si dovoliť zaoberať sa výskumom, prípadne implementáciou jeho výdobytkov. Spoločnosti vytvárajúce rozsiahlejšie projekty sa však použitím technológií bez tradície a osvedčenia vystavujú riziku a ich ochota investovať do vývoja je porovnateľná s mierou ochoty podeliť sa s dosiahnutými výsledkami.

V neposlednom rade medzi dôvody patrí aj informačné zázemie tejto paradigmy. Hoci je publikačná činnosť so zameraním na túto tému bohatá[1], výraznú väčšinu publikácií tvoria vedecké články, ktorých obsah nie vždy možno označiť ako neredundantný v porovnaní s ostatnými článkami. Knižná publikačná činnosť je dosť obmedzená a odborné práce a učebnice je možné rátať iba na desiatky a obvykle sa nové a aktualizované vydania týchto prác neobjavujú.

Situácia na slovenskom knižnom trhu je dosť úsmevná. Počet monografií zaoberajúcich sa aspektovo-orientovanou paradigmou: nula, čoho dôsledkom je, že pri písaní prác, hlavne, čo sa týka prekladov slov z anglického jazyka, neexistuje publikácia, na ktorú by sa mohli autori odvolávať, a názvoslovie nie je unifikované.

Problematikou aspektovo-orientovaného programovania sa čiastočne zaoberá kniha Objektovo-orientované programovanie: Objekty, Java a aspekty od Valentina Vranića [6], no nie všetky preklady výrazov použitých v tejto záverečnej práci sú prevzaté z tejto publikácie.

V akademickej oblasti sa každoročne usporadúva Medzinárodná konferencia o aspektovo-orientovanom softvérovom vývoji (AOSD). Prvýkrát sa táto konferencia konala v roku 2002 a tento rok to bude už jedenásta v poradí.

[1] Práca Bibliography of Aspect-Oriented Software Development od Robert E. Filmana [5] zoskupujúca diela z oblasti AOSD obsahuje viac ako päť tisíc bibliografických odkazov

1.1.5 Trend budúcnosti a prínos vo vývoji softvéru

Vyššie spomenutá Encyklopédia informačnej vedy [1] a informačných technológií označuje aspektovo-orientovanú paradigmu ako trend budúcnosti, ktorý môže byť riešením pre rôzne problémy vo fázach pred nasadením softvérového systému, rovnako aj po nasadení.

Pred nasadením projektu do prevádzky môže aspektovo-orientované programovanie zohrať úlohu v podpore:

- udržiavateľnosti ako kvalitatívneho faktora a tak zjednodušiť modifikáciu systému
- analyzovateľnosti a zrozumiteľnosti kódu
- znovupoužiteľnosti
- adaptabilnosti

Na druhej strane po nasadení projektu do prevádzky sa môže dopomôcť

- uspokojiť časté nefunkčné požiadavky na údržbu softvérového projektu
- zjednodušiť refactoring
- zvýšiť škálovateľnosť systému.

1.1.6 Symetrický a asymetrický prístup k modularizácii aspektov

Apektovo-orientovaná paradigma rozlišuje medzi asymetrickým a symetrickým modularizovaním aspektov.

Asymetrická modularizácia je založená na tvorbe aspektov, pretínajúcich koncernov, ako oddelených a nezávislých entít od základnej funkcionality programu (najčastejšie objektovo-orientovanej), do ktorej musia byť v prípade nastatia určitej vopred definovanej udalosti začlenené. Asymetrický prístup je v súčasnosti používaný viac ako symetrický.

Symetrická modularizácia používa iba jeden druh modularizovaných komponentov a na rozdiel asymetrického prístupu nepripúšťa viac druhov modularizácie. Program teda pozostáva iba zo samostatných koncernov, aspektov. Tento prístup nie je natoľko rozšírený, pretože aspektovo-orientované programovanie je ešte iba vo svojich začiatkoch a nenadobudol a ani tak skoro nenadobudne dostatok sily, aby mohol substituovať iné paradigmy, hlavne OOP nie.

Ako to už býva, vznikajú rôzne nedorozumenia v terminológií v aspektovo-orientovanej paradigme, pretože oba spomenuté smery používajú niektoré rovnaké názvy, no ich významy sa nezhodujú.

1.1.7 Aspektovo-orientovaný vývoj softvéru a aspektovo-orientované softvérové inžinierstvo

Aspektovo-orientované softvérové inžinierstvo je jedna z disciplín softvérového inžinierstva, ktorej cieľom je separácia koncernov. Na identifikáciu koncernov je možné použiť prístup založený na pohľadoch (angl. viewpoint). Požiadavky zákazníkov a iných zainteresovaných strán prispievajú k definícii jednotlivých pohľadov a následnej identifikácií pretínajúcich koncernov.

Pretínajúce koncerny by sa nemali vynechať ani z jednej z fáz aspektovo-orientovaného vývoja softvéru a na dosiahnutie tohto želaného stavu je nutné si osvojiť koncernovo-orientovaný spôsob myslenia. Už v počiatočných fázach vývoja sa na úrovni požiadaviek dajú identifikovať kľúčové koncerny (angl. core concerns) a koncerny – rozšírenia (angl. extensions).

Koncerny je ďalej možné rozdeliť do rôznych skupín [17], ako napr.:

- funkcionálne koncerny – obsahujú špecifickú funkcionalitu
- koncerny kvality služieb – založené na nefunkčných požiadavkách
- strategické koncerny – stratégie riadenia systému
- systémové koncerny – vlastnosti systému ako celku
- organizačné koncerny – zamerané na ciele organizácie

Pri modelovaní a návrhu dochádza často k dvom negatívnym javom, na ktoré si treba dať pozor, k prepleteniu (angl. tangling) a k roztrúseniu (angl. scattering). Prepletenie spočíva v implementácii rôznych systémových požiadaviek v jednom module. Roztrúsenie nastáva vtedy, ak jeden koncern, príp. požiadavka, je implementovaná vo viacerých moduloch. Výskyt týchto javov býva obvykle dôsledkom zmeny systémových požiadaviek.

Prechod od analýzy, od definovaných požiadaviek, k návrhu je uskutočniteľný v podobe prípadov použitia, ktoré jednotlivo zaznamenávajú koncerny. Čo sa týka UML jazyka ako modelovacieho nástroja, na znázornenie aspektov sa používajú stereotypy.

Pri programovaní opäť dochádza k zmene zaužívaných postupov, okrem písania aspektov je to iný druh analýzy a čítania kódu. Na jednej strane prináša aspektovo-orientované programovanie pre pisateľa kódu lepšiu modularitu a zrozumiteľnosť, no na druhej strane, pre čitateľa kódu, ktorý sa na jeho tvorbe nepodieľal to také jednoznačné nemusí byť.

Kvôli vtkávaniu kódu nie je možné čítať zdrojový kód aspektovo-orientovaného programu sekvenčne, čo vedie k zníženiu prehľadnosti. Riešením by mohli byť podporné nástroje na čítanie kódu pracujúce na princípe vloženia kódu aspektu do bodov spájania, no častokrát neprinášajú očakávaný efekt, pretože programy môžu byť dynamické a jednému bodu spájania môže prislúchať viacero prierezových bodov.

Ian Sommerville vo svojej známej knihe Softvérové inžinierstvo predkladá názor, že problémy s testovaním sú hlavným dôvodom, prečo nie je aspektovo-orientované programovanie rozšírené vo väčšej miere.

1.1.8 Aspektovo-orientované technológie

Podpora aspektovej orientácie si nachádza cestu vo viacerých technológiách. Nasleduje výber z tých najznámejších.

- AspectJ
- JBoss-AOP
- JAC (Java Aspect Components)
- HyperJ
- CaesarJ
- AspectC++
- FeatureC++
- Aspect#
- AspectXML
- PHPaspect
- AspectR (rozšírenie pre jazyk Ruby)
- AspectL (rozšírenie pre jazyk Lisp)
- PostSharp
- AspectDNG
- AspectJS (rozšírenie pre jazyk JavaScript)
- Aspyct (rozšírenie pre jazyk Python)

- Apostle (rozšírenie pre jazyk Smalltalk)
- Pythius (rozšírenie pre jazyk Python)
- Phpaspect
- AspectS (rozšírenie pre jazyk Smalltalk)

Technológie implementujúce AOP možno rozdeliť do viacerých skupín. Jedno z rozdelení je nasledovné:

1. Kompilátorom podporované statické AOP, napr. AspectJ
2. Kompilátorom podporované dynamické AOP na úrovni aplikácie, napr. CesarJ
3. Kompilátorom podporované dynamické AOP na úrovni exekučného prostredia
4. Konfiguračne riadené AOP na úrovni exekučného prostredia
5. Dynamický AOP framework na úrovni exekučného prostredia
6. Aspektovo-orientované metaprogramovanie.

1.2 Programovací jazyk AspectJ

Aspektovo-orientovaný programovací jazyk AspectJ je jednou z najrozvinutejších a najpoužívanejších implementácií aspektovo-orientovanej paradigmy, v histórii ktorej zohráva významnú úlohu. Jazyk AspectJ je častokrát označovaný ako de facto štandard pre aspektovo-orientovanú paradigmu, podkladajúci nielen základné termíny, ale hlavne základné pravidlá a vlastnosti aspektovo-orientovanej tvorby.

1.2.1 Charakteristika AspectJ

Pôvodným zámerom spoločnosti Xerox Pablo Alto Research Center (PARC), ktorá AspectJ zverejnila v roku 1997, nebolo vytvoriť všeobecne účelový jazyk akým je AspectJ v súčasnosti, ale špecificky účelový, no zmena v zameraní jazyka sa ukázala ako veľmi prospešná. Prelomovým sa stalo spojenie so spoločnosťou Eclipse v roku 2002, ktorá prebrala AspectJ od Xerox PARC a cez Eclipse Technology Project sa z AspectJ stáva open-source softvér.

Prvá verzia AspectJ 1.0 bola vydaná v roku 2001. Súčasná stabilná verzia, v marci 2012, je 1.6.12 a posledná skúšobná verzia je 1.7.0.M1 (angl. M - milestone, slov. míľnik).

AspectJ je v súčasnosti licencovaný pomocou Eclipse Public License 1.0 (EPL) od verzie AspectJ 1.5.2; verzie AspectJ 1.1 až po AspectJ 1.5.1 sú licencované s Common Public License 1.0 a verzia AspectJ 1.0 je licencovaná s Mozilla Public License 1.1.

Verzia AspectJ 1.2 a predchádzajúce boli iba rozšírením jazyka Java.

Pomenovania verzií nie sú konzistentné, a po verzií 1.2 nasleduje 1.5, kvôli nadväznosti na jazyk Java. V Tabuľke č. 2 je uvedený prehľad verzií jazyka AspectJ, na ktorom je možné vidieť, že s každým novým vydaním stúpal počet verzií, čo svedčí o úspešnosti vývoja.

Č.	Názov	Jednotlivé verzie	Rok vydania verzie 1.x.0
1.	AspectJ 1.0	1.0.6	
2.	AspectJ 1.1	1.1.0, 1.1.1	6. 6. 2003
3.	AspectJ 1.2	1.2.0, 1.2.1	25. 5. 2004
4.	AspectJ 5	1.5.0 - 1.5.4	20. 12. 2005
5.	AspectJ 6	1.6.0 - 1.6.12	16. 4. 2008
6.	AspectJ 7	1.7.0.M1	16. 12. 2011

Tabuľka č. 2 Verzie jazyka AspectJ

Najnovšia verzia AspectJ 1.7.0.M1 používa Eclipse 3.7 Java7 kompilátor a je možné zahrnúť Java 7 do AspectJ 1.7.0.M1. Predchádzajúca verzia používala Eclipse 3.3 Java 5 kompilátor.

1.2.2 AspectJ a Java: kompatibilita a porovnanie

Jazyk AspectJ je postavený na jazyku Java a už od začiatku si kládol za svoj cieľ zachovať si kompatibilitu s Javou. Túto kompatibilitu možno rozdeliť medzi štyri základné typy [7]:

- **dopredná kompatibilita** – všetky programy napísané v jazyku Java sú aj platné programy v jazyku AspectJ
- **nástrojová kompatibilita** – nástroje jazyka Java môžu byť rozšíriteľné a využiteľné aj v jazyku AspectJ
- **platformová kompatibilita** – všetky platné programy napísané v jazyku AspectJ sú spustiteľné pomocou Java virtuálneho stroja (angl. Java Virtual Machine, JVM)
- **programovacia kompatibilita** – prechod na programovací jazyk AspectJ by mal byť plynulý pre Java programátorov

Vzťah medzi zachovaním platnosti programov je iba jednostranný. Programy platné v jazyku AspectJ nie sú platné v jazyku Java.

Okrem toho, aby mohla byť dodržaná dopredná kompatibilita, musí byť nenarušená syntax jazyka Java, t. j. kľúčové slová, ktoré sa nevyskytujú v jazyku Java, ako napr. aspect, poincut, advice atď, musia byť kontextovo-viazané. V jazyku Java sa teda môžu používať všetky kľúčové slová jazyka AspectJ ako nekľúčové.

Prehľad základných rozdielností medzi oboma jazykmi je uvedený v Tabuľke č.3 .

KRITÉRIÁ POROVNANIA	ASPECTJ	JAVA
Bezpečnosť programovacích postupov	Umožňuje písať veľmi bezpečný kód, no podpora bezpečnosti je limitovaná.	Silná podpora bezpečnosti
Objektovo-orientovaná abstrakcia	Podporuje OOP abstrakciu	Primárne OOP jazyk
Aspektovo-orientované programovanie	Primárne AOP jazyk	Dosiahnuteľné pomocou AspectJ
Funkcionálne programovanie	AspectJ implementuje profiler, ktorý zaznamenáva počty volaní metód	Sama osebe nepodporuje funkcionálne programovanie, namiesto toho je možné dosiahnuť niektoré vlastnosti funkcionálneho programovania pomocou rozhraní a vnútorných tried.
Deklaratívne programovanie	Podporuje deklaratívne programovanie pomocou anotácií	Podporuje deklaratívne programovanie prostredníctvom knižníc, ako napr. JSetL a JSolver
Web aplikácie	Podpora Web aplikácií poskytovaním bezpečnostných prvkov počas transakcií	Kvalitný, robustný a populárny nástroj na vývoj web aplikácií.
Web Service Dizajn a kompozícia	AspectJ podporuje webové služby ako napr. AXIS ale nepodporuje všetky prvky sám osebe	Vhodný pre webové služby, pre vysokú mieru prenositeľnosti a veľké množstvo podporných API
Reflection	Môže implementovať pomocou ―org.aspectj.lang.reflect.*‖	Mocný mechanizmus podporujúci reflection, ktorý umožňuje prácu s metadatami a predchádzanie komplikácií.
Batch Scripting	Sám osebe nepodporuje	Jednoduché použitie pomocou dvoch Java tried: Runtime class a Process class.

Tabuľka č. 3 Porovnanie jazykov AspectJ a Java

1.2.3 Nástroje na vývoj

Popri AspectJ ako jazyku, ktorý je sám osebe AOP nástroj na vývoj aplikácií sa stretávame ešte s AspectJ vývojovými nástrojmi (angl. AspectJ Development Tools, AJDT) a AspectJ nástrojmi (angl. AspectJ tools), ktoré je potrebné rozlišovať.

AspectJ vývojové nástroje predstavujú skupinu nástrojov určených na podporu aspektovo-orientovaného vývoja na platforme Eclipse a zabezpečenie kompatibility s Java

Developement Tools (JDT). Do Eclipse IDE sú inštalovateľné v podobe pluginu zahrňajúceho triedy a rozhrania z AspectJ knižníc.

AspectJ nástroje obsahujú:

- kompilátor ajc a bytecode weaver pre AspectJ a Java
- dokumentačný nástroj ajdoc
- ajbrowser, a crosscutting code viewer;
- podpora Ant pre ajc;

Tieto nástroje sú súčasťou knižnice AspectJ, hlavne aspectjtools.jar a aspectjrt.jar.

1.2.4 Implementácia základných koncepcií AOP v jazyku AspectJ

V predchádzajúcej podkapitole boli opísané základné koncepcie AOP, ktoré sú všetky prítomné v jazyku AspectJ. Model bodov spájania predstavuje súbor exponovaných (použiteľných) bodov spájania a ich kontextov, na ktoré sú aspekty, resp. bodové prierezy nadviazané.

Základnou programovou konštrukciou jazyka AspectJ je aspekt určený na zapuzdrenie pretínajúcej sa funkcionality. Popri aspektoch pozná AspectJ aj všetky konštrukcie obsiahnuté v jazyku Java z dôvodu dodržania doprednej kompatibility.

Deklarácia aspektu pozostáva z modifikátora prístupu, kľúčového slova **`aspect`**, identifikátora aspektu a tela aspektu. Telo aspektu sa môže skladať z bodových prierezov (angl. pointcuts) a videní (angl. advices) a súčasne môže obsahovať aj atribúty a metódy.

V nasledujúcom príklade je uvedený jednoduchý program na priblíženie základnej syntaxe jazyka AspectJ, ktorý je zložený z triedy AhojSvetTrieda.java a z aspektu AhojSvetAspekt.aj.

Triedy AhojSvetTrieda.java je napísaná výlučne v jazyku Java a neobsahuje žiadne prvky jazyka AspectJ. Aspekty v jazyku AspectJ nie je možné priamo inštanciovať, trieda AhojSvetTrieda.java nemôže obsahovať žiadnu programovú konštrukciu (napr. new ako je to pri OOP), ktorá by vytvárala aspekt z AhojSvetAspekt.aj.

```
public class AhojSvetTrieda
{
	public void vypisAhoj()
	{
		System.out.println("--- Ahoj svet z triedy! ---");
	}

	public static void main(String[] args)
	{
		new AhojSvetTrieda().vypisAhoj();
	}
}
```

Obrázok č. 2 AhojSvetTrieda.java

```
public aspect AhojSvetAspekt
{
	pointcut vypisPozdrav() :
		call (void AhojSvetTrieda.vypisAhoj(..));

	before() : vypisPozdrav()
	{
		System.out.println("Ahoj svet z aspektu - before() advice!");
	}


	void around(): vypisPozdrav()
	{
		System.out.println("Ahoj svet z aspektu - around() advice!");
		proceed();
		System.out.println("Ahoj svet z aspektu - around() advice!");
	};

	after() returning : vypisPozdrav()
	{
		System.out.println("Ahoj svet z aspektu - after() advice!");
	}

}
```

Obrázok č. 3 AhojSvetAspekt.aj

```
Ahoj svet z aspektu - before() advice!
Ahoj svet z aspektu - around() advice!
--- Ahoj svet z triedy! ---
Ahoj svet z aspektu - around() advice!
Ahoj svet z aspektu - after() advice!
```

Obrázok č. 4 Konzolový výstup

V tomto jednoduchom príklade je AhojSvetAspekt.aj na úrovni balíka, ktorý obsahuje okrem neho už len AhojSvetTrieda.java. Okrem takéhoto umiestnenia môžu byť

aspekty ešte súčasťou iného aspektu, triedy alebo rozhrania, no v týchto prípadoch musí byť aspekt deklarovaný ako static.

AhojSvetAspekt.aj je verejný (modifikátor **`public`**) a obsahuje jeden bodový prierez nazvaný **`vypisPozdrav()`** a tri videnia **`before()`**, **`around()`** a **`after()`**.

Deklarácia bodového prierezu je zložená z kľúčového slova **`pointcut`**, identifikátora bodového prierezu **`vypisPozdrav()`** operátora dvojbodky, za ktorým nasleduje definovanie bodu spájania. V tomto prípade je bod spájania určený volaním metódy, t.j. použitím kľúčového slova **`call`**, pričom za ním v okrúhlych zátvorkách sa nachádza signatúra funkcie **`void AhojSvetTrieda.vypisAhoj(..)`**, po volaní ktorej má byť vložený kód videnia, ak je s týmto bodovým prierezom nejaký asociovaný.

```
pointcut vypisPozdrav() :
	call (void AhojSvetTrieda.vypisAhoj(..));
```

Obrázok č. 5 Pointcut

Signatúry funkcií nemusia byť vždy úplne so signatúrami funkcií na mieste bodu spájania. Môžu obsahovať rôzne zástupné notácie (angl. wildcard notations), ako napr.:

- .. (dve bodky) na zastúpenie ľubovoľného počtu parametrov
- * (asterisk) na zastúpenie ľubovoľnej návratovej hodnoty
  - v ukážke by sa mohol nachádzať namiesto void
- + (znak plus) na zastúpenie podtried alebo subrozhraní

Okrem zástupných notácií sa často pri definovaní bodov spájania vyskytujú unárne a binárne operátory slúžiace na bližšie vymedzenie prierezového bodu, ktorý nemusí vždy nadväzovať na jediný bod spájania. Napríklad medzi takéto operátory patria:

- ! (výkričník) – unárny operátor, negácia
- || (dve vertikály) - binárny operátor, alebo
- && (dva ampersandy) - binárny operátor, súčasne

Uvedený bodový prierez je tzv. pomenovaný, má identifikátor **`vypisPozdrav()`**. V jazyku AspectJ je možné deklarovať aj anonymné bodové prierezy v miestach ich použitia, napr. v rámci videní. Rozlišujeme viacero druhov bodových prierezov súvisiacich s určitými elementami, medzi najpoužívanejšie patria napr.

- metóda alebo konštruktor
  - volanie (angl. call)
  - exekúcia
  -

- atribúty
  - prístup na čítanie atribútov, get
  - prístup na úpravu atribútov, set
- ošetrenia výnimiek
- videnia
  - exekúcia
- atď.

V ukážke zdrojového kódu aspektu AhojSvetAspekt.aj nasledujú po deklarácii bodového prierezu **`vypisPozdrav()`** odlišné aspektovo-orientované programové konštrukcie aspektu – videnia. Videnia slúžia na implementáciu funkcionality pretínajúcich koncernov a je možné ich rozdeliť do troch základných skupín, ktoré sú všetky zastúpené v príklade. Sú to:

- **`before()`** – kód videnia je exekuovaný pred bodom spájania

```
before() : vypisPozdrav()
{
System.out.println("Ahoj svet z aspektu - " +
+  "before() advice!");
}
```

Obrázok č. 6 Before advice

- **`around()`** – časť kódu videnia je exekuovaná pred a časť po bode spájania

```
void around(): vypisPozdrav()
{
System.out.println("Ahoj svet z aspektu - " +
+ "around() advice!");
proceed();
System.out.println("Ahoj svet z aspektu - " +
+ "around() advice!");
};
```

Obrázok č. 7 Around advice

  - **`proceed();`** zastupuje vykonanie metódy v bode spájania
- **`after()`** – kód videnia je exekvovaný po bode spájania.

```
after() returning : vypisPozdrav()
{
System.out.println("Ahoj svet z aspektu - " +
+ "after() advice!");
}
```

Obrázok č. 8 After advice

Videnie **`after()`** je ďalej možné rozdeliť na:

- **`after()`** – bez špecifikátora, vykoná sa po bode spájania v každom prípade
- **`after() returning`** – v prípade „normálneho“ priebehu
- **`after() throwing`** – v prípade výnimky

V uvedenom príklade boli programovo znázornené základné koncepcie AOP v jazyku AspectJ, no absentuje tu podstata tejto technológie. K pretínaniu dochádza v podobe pretínania chodu programu pomocou bodových prierezov v mieste spájania. Toto pretínanie nemá však nič spoločné s pretínajúcimi koncernami, ktoré sú základom AOP.

Pri takomto jednoduchom príklade je možné dosiahnuť rovnaký výstup iba s vložením príslušných **`System.out.println(..)`** metód v rovnakom poradí do metódy **`main`** a úplne vynechaním aspektu, čo by mohlo predstavovať úsporu kódu.

```
public class AhojSvetTrieda
{
	public void vypisAhoj()
	{
		System.out.println("--- Ahoj svet z triedy! ---");
	}

	public static void main(String[] args)
	{
		System.out.println("Ahoj svet z aspektu - before() advice!");
		System.out.println("Ahoj svet z aspektu - around() advice!");
		new AhojSvetTrieda().vypisAhoj();
		System.out.println("Ahoj svet z aspektu - around() advice!");
		System.out.println("Ahoj svet z aspektu - after() advice!");
	}
}
```

Obrázok č. 9 AhojSvetTrieda.java

Táto úspora však platí, iba ak sa v pôvodnej **`main`** metóde volá **`new AhojSvetTrieda().vypisAhoj();`** jeden alebo dvakrát. Pri vyššom výskyte je použitie aspektu výhodnejšie. Ďalšou výhodou je jednoznačné zníženie redundancie kódu, ktorý by bez použitia aspektu mohol byť identifikovaný ako pretínajúci koncern. Rovnako by boli zlepšené aj prehľadnosť, čitateľnosť a modifikovateľnosť.

1.2.5 Anotácie

S príchodom verzie 1.5 jazyka AspectJ bol sprístupnený nový spôsob zápisu aspektov, ktorým bolo použitie anotácií (angl. annotations). Anotácie sú označenia elementov kódu a je možné ich syntakticky rozpoznať podľa znaku zavináč pred ich identifikátorom.

V API jazyku AspectJ sú definované v rámci **`org.aspectj.lang.annotation`** balíka.

Nasleduje príklad z predchádzajúcej podkapitoly prepísaný pomocou použitia anotácií: **`@Aspect, @Pointcut, @Before, @Around`** a **`@After`**.

```
import org.aspectj.lang.annotation.Pointcut;
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.ProceedingJoinPoint;

@Aspect
public class AhojSvetAspekt
{
	@Pointcut("call (void AhojSvetTrieda.vypisAhoj(..))")
	void vypisPozdrav() {}

	@Before("vypisPozdrav()")
	public void vypisPozdravBefore()
	{
		System.out.println("Ahoj svet z aspektu - before() advice!");
	}

	@Around("vypisPozdrav()")
	public void vypisPozdravAround(ProceedingJoinPoint joinPoint)
		 throws Throwable
	{
		System.out.println("Ahoj svet z aspektu - around() advice!");
		joinPoint.proceed();
		System.out.println("Ahoj svet z aspektu - around() advice!");
	}

	@After("vypisPozdrav()")
	public void vypisPoz()
	{
		System.out.println("Ahoj svet z aspektu - after() advice!");
	}
}
```

Obrázok č. 10 AhojSvetAspekt.java

1.2.6 Medzitypové deklarácie

Medzitypové deklarácie (angl. iner-type declarations) sú ďalšou výraznou črtou aspektovo-orientovaného jazyka AspectJ. Ich úloha je v prostredí aspektu vytvoriť nový prvok a potom ho priradiť triede. Týmto prvkom môže byť rozhranie, rodič, atribút, metóda a iné.

Na Obrázkoch č. 10 a č. 11 sú uvedené príklady medzitypovej deklarácie.

```
declare parents: MojaTrieda implements MojeRozhranie;
```

Obrázok č. 11 Medzitypová deklarácia rozhrania

```
declare parents: MojaTrieda extends MojaRodicovskaTrieda;
```

Obrázok č. 12 Medzitypová deklarácia rodičovskej triedy

1.2.7 Spôsoby vtkania

Jednu zo základných koncepcií AOP predstavuje metóda na pripojenie jednotiek, aspektov, do programu. AspectJ používa na tento účel proces nazývaný vtkanie (angl. weaving).

Vtkanie aspektov môžeme rozdeliť na štyri druhy:

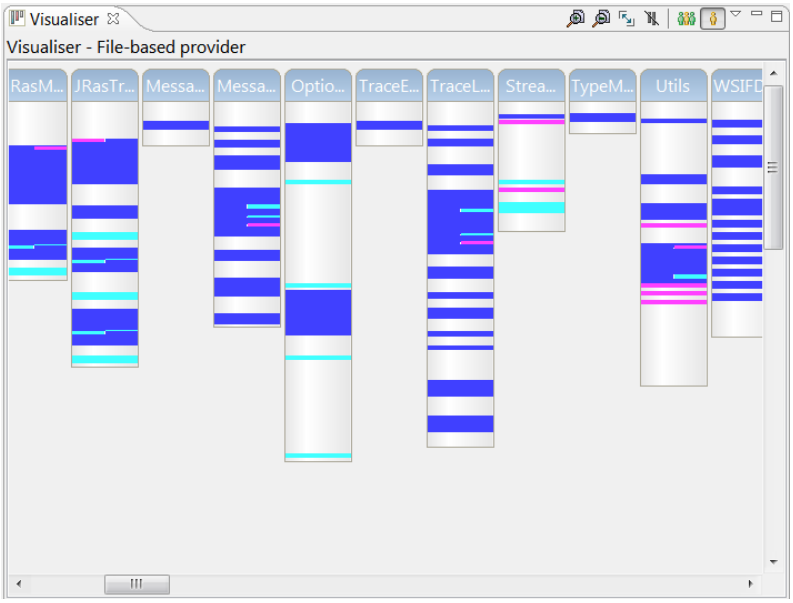
- vtkanie v čase kompilácie programu (angl. compile-time weaving)
  - zabezpečuje kompilátor ajc
- vtkanie v čase zostavenia programu (angl. link-time weaving)
  - zabezpečuje linker
- vtkanie v čase zavedenia programu (angl. load-time weaving)
  - zabezpečuje classloader
- vtkanie v čase behu programu (angl. run-time weaving).
  - zabezpečuje virtuálny stroj

1.2.8 AspectJ v Eclipse IDE

AspectJ Development Tools (AJDT) sú skupinou nástrojov umožňujúcich programovanie v jazyku AspectJ vo vývojovom prostredí Eclipse. Súčasná verzia (apríl 2012) Eclipse je 3.7.2 Indigo. Eclipse je integrované vývojové prostredie vytvorené spoločnosťou Eclipse Foundation, ktorá stojí zároveň za vývojom jazyka AspectJ.

Do tohto prostredia je zabudovaná v procese inštalácie ADJT Aspect Visualisation perspektíva, ktorý obsahuje Visualiser určený na vizuálne znázorňovanie aspektov.

Aspekty, resp. pretínajúce koncerny, sa tradične znázorňujú ako stĺpcové grafy, ktorých jednotne sfarbené časti (horizontálne orientované) reprezentujú pretínajúce koncerny.



Obrázok č. 13 Eclipse AspectJ Visualiser

Okrem Visualiseru poskytuje Eclipse IDE aj AspectJ editor, ktorý je určený na písanie kódu. Pri doprednej kompatibilite jazyka AspectJ, nemožno písať AsppectJ kód v Java editore, pretože ten nepozná AspectJ syntaktické konštrukcie a hlási pri nich Syntax error.

1.3 Návrhové vzory

Po úvode do aspektovo-orientovanej paradigmy, jednej jej implementácii, aspektovo-orientovaného jazyka AspectJ, môžeme prejsť k návrhovým vzorom. Návrhové vzory v posledných dvoch desaťročiach ovplyvnili výrazným spôsobom softvérový návrh a majú svoj vplyv aj na aspektovo-orientovaný vývoj sofftvéru.

1.3.1 Definícia a pôvod návrhových vzorov

Návrhové vzory (angl. design patterns) zaujímajú pevné miesto v oblasti softvérového návrhu. Ako častá definícia sa uvádza: „Návrhové vzory opisujú jednoduché a elegantné riešenie špecifického problému objektovo-orientovaného softvérového dizajnu. Zachytávajú riešenie, ktoré prešlo vývojom v čase.“ [12, s. 16]

Túto definíciu je možné nájsť v diele Návrhové vzory: Elementy znovupoužiteľného objektovo-orientovaného softvéru [12], ktoré je prvým významným dielom zaoberajúcim sa softvérovými návrhovými vzormi a ktoré ich preslávilo a spopularizovalo natoľko, že drvivá väčšina po ňom nasledujúcich publikácií obsahuje referencie a odvoláva sa práve na toto dielo. Erich Gamma, Richard Helm, Ralph Johnson a John Vlissides sú jeho autormi, niekedy označovaní ako Gang štyroch (angl. Gang of Four, GoF).

Pred GoF boli vzory doménou architektúry, najmä dielo z roku 1977 Jazyk vzorov od Christophera Alexandra a kol. bolo predchodcom programových vzorov. V ňom sú vzory definované ako: „Každý vzor opisuje problém, ktorý sa vyskytuje opäť, znova a zase v našom prostredí a opisuje jadro riešenia daného problému, takým spôsobom, že je možné použiť toto riešenie miliónkrát znova bez toho aby ten spôsob bol dvakrát taký istý.“ [12, s. 38]

O necelých dvadsať rokov po Jazyku vzorov GoF vydali svoje dielo Návrhové vzory (rok 1995, [12]). Počas nasledujúcich rokov až po súčasnosť si našli návrhové vzory široké pole pôsobnosti až do takej miery, že vznikla odnož softvérového inžinierstva, ktorá je celá založená na používaní vzorov, vzorovo-riadený vývoj (angl. Pattern-Driven Development, PDD), prípadne vzorovo-orientovaný softvérový vývoj (angl. Pattern-Oriented Software Developement). Samozrejme neostalo to iba pri pôvodných 23 vzoroch od GoF, ale bolo vytvorených stovky ďalších.

1.3.2 Kategorizácia návrhových vzorov

Kategorizáciu návrhových vzorov je možné posudzovať z viacerých hľadísk. Uvedieme si tri základné, ktorých vymedzenie bude urobené v 2. kapitole Cieľ práce, metodika práce a metódy skúmania a uplatnené v 3. kapitole Výsledky práce a diskusia.

Prvým druhom kategorizácie je stanovenie vlastností opisu a kritérií porovnávania vzorov, t. j. činnosť, pri ktorej sa určuje forma zápisu vzoru. Podľa GoF [12] musí takýto zápis návrhového vzoru obsahovať: názov, opis problému, opis riešenia a možné dôsledky, pričom ich vzory boli zaznamenané pomocou týchto častí: názov, význam, iné pomenovanie, motivácia, aplikovateľnosť, štruktúra, participujúci, kolaborácia, dôsledky implementácia, ukážka kódu, známe použitia a príbuzné vzory.

Ďalším druhom kategorizácie je vytvorenie katalógu vzorov z určitej oblasti softvérového inžinierstva, t. j. činnosť, pri ktorej sa zhromaždia jednotlivé opisy vzorov vo vopred definovanej unifikovanej forme. GoF [12] sa rozhodli opísať 23 všeobecne zameraných objektovo-orientovaných vzorov, ktoré budú predmetom záujmu nasledujúcich častí práce, rozdelených do troch skupín:

- vytvárajúce vzory (angl. creational patterns)
- štrukturálne vzory (angl. structural patterns)
- behaviorálne vzory (angl. behavioral patterns).

Tabuľka č. 4 obsahuje zoznam pôvodných návrhových vzorov vydaných GoF s anglickými pomenovaniami, ktoré budú používané v tejto práci a so slovenskými prekladmi ako boli uvedené v Štúdie vybraných tém programových a informačných systémov (4. diel) [10].

DRUH VZORU	Anglický názov vzoru	Slovenský názov vzoru
Vytvárajúce vzory	Abstract Factory	Abstraktná továreň
	Builder	Staviteľ
	Factory Method	Výrobná metóda
	Prototype	Prototyp
	Singleton	Unikát
Štrukturálne vzory	Adapter	Adaptér
	Bridge	Premostenie
	Composite	Zloženina
	Decorator	Dekoratér
	Facade	Fasáda
	Flyweight	Mušia váha
	Proxy	Zástupca
Behaviorálne vzory	Chain of Responsibility	Reťaz zodpovednosti
	Command	Príkaz
	Interpreter	Interpreter
	Iterator	Iterátor
	Mediator	Sprostredkovate
	Memento	Memento
	Observer	Pozorovateľ
	State	Stav
	Strategy	Stratégia
	Template Method	Šablónová metóda
	Visitor	Návštevník

Tabuľka č. 4 Slovenské názvy GoF návrhových vzorov

Kategorizácia podľa GoF [12] nie je jedinou tohto druhu. Po nej nasledovalo mnoho iných, ako napr. čo sa týka J2EE-špecifických návrhových vzorov (ktoré sa nesnažia byť všeobecné), tie rozdeľujeme podľa trojvrstvového architektonického vzoru na: návrhové vzory prezentačnej vrstvy, návrhové vzory vrstvy biznis logiky a návrhové vzory prístupu k dátam.

Tretí druh kategorizácie vychádza z predpokladu, že okrem návrhových vzorov existujú aj iné druhy softvérových vzorov, ktoré je možné rozlíšiť podľa rozsahu alebo miery abstrakcie. Poznáme tri základné kategórie [11]:

- architektonické vzory
- návrhové vzory
- idiómy.

Architektonické vzory sú určené na vyjadrenie fundamentálnej organizačnej schémy softvérových systémov. Poskytujú súbor vopred definovaných subsystémov, špecifikujú ich úlohu a zahrňujú pravidlá a odporúčania pre organizáciu vzťahov medzi nimi.

Návrhové vzory predstavujú v porovnaní s architektonickými vzormi menšie softvérové jednotky. Ich definícia bola uvedená vyššie. Aplikácia návrhového vzoru by nemala mať vplyv na softvérovú architektúru systému, no môže výrazným spôsobom ovplyvniť architektúru subsystému.

Idiómy, niekedy nazývané mikrovzory, sú kategóriou nízkoúrovňových vzorov špecifických pre konkrétny programovací jazyk a opisujú ako je možné implementovať určité komponenty alebo vzťahy medzi nimi použitím vlastností daného programovacieho jazyka. Idiómy zohrávajú dôležitú úlohu pri škálovateľnej evolučnej analýze softvérových projektov, pretože sú vo väčšej miere automaticky extrahovateľné zo zdrojového kódu ako návrhové vzory.

Doteraz spomenuté vzory slúžia ako pozitívne vzory, ako príklady, ktorých sa treba pridržiavať, aby bolo dosiahnuté vylepšenie dizajnu systému, príp. subsystému alebo implementácie.

Na druhej strane existujú negatívne vzory, tzv. antivzory dokumentujúce časté chyby a omyly softvérových dizajnérov slúžiace ako výstrahy, ktorým sa treba vyhýbať. Ako reakcia na GoF [12] bola napísaná kniha Antivzory: Refaktoring softvéru, architektúr a projektov v kríze [13], ktorá sa ako prvá zaoberá touto problematikou.

1.3.3 Prínos návrhových vzorov

Dôvodov, prečo je výhodné sa zaoberať návrhovými vzormi, je dlhý rad. Obvykle sa uvádzajú nasledujúce.

Pre človeka je prirodzené riešenie problémov odvodené od predchádzajúcich riešení, znalostí a skúseností v jeho živote. Návrhové vzory sú preto blízke človeku, lebo prešli dôkladným overovaním a preskúšaním. Pri vývoji softvéru sa vývojári nachádzajú v podobných situáciách tak často a bežne, že je možné vytvoriť všeobecný a opakovane použiteľný vzor riešenia. Znovupoužiteľnosť návrhových vzorov je ich charakteristickou črtou. Návrhové vzory disponujú značnou pridanou hodnotou, lebo ich autormi sú softvéroví experti ochotní predávať svoje skúsenosti ďalej.

Po oboznámení sa s návrhovými vzormi býva hľadanie nových riešení na novovzniknuté problémy jednoduchšie. Prístup k návrhovým vzorom je voľný; vzory nie sú autorsky chránené a ich zdieľanie a používanie nie je nijako obmedzované, čo ešte viac zvyšuje ich atraktívnosť.

V súčasnosti patria návrhové vzory medzi prirodzenú súčasť pokročilých programátorských a vývojárskych vedomostí. Uľahčujú vzájomnú komunikáciu v tíme, prácu na projektoch a pri správnej aplikácii šetria čas a peniaze.

1.3.4 Limity návrhových vzorov

Návrhové vzory si vyžadujú skúsenosti a prax so softvérovým dizajnom v procese ich tvorby, rovnako i v procese ich implementácie. Pri ich tvorbe musia byť tieto skúsenosti neporovnateľne väčšie, no ani pri ich používaní nie sú zanedbateľné. Je nutné vedieť sa orientovať vo veľkom množstve vzorov, mať schopnosti im porozumieť a vedieť správne rozhodnúť o ich nasadení, čo pri nedostatku skúseností nemusí vždy viesť k očakávaným výsledkom.

Nie je nič prekvapujúce, že návrhové vzory nie sú riešením pre všetky problémy vznikajúce v návrhu softvéru. Niektoré problémy musia byť riešené unikátne, bez prislúchajúceho návrhového vzoru. A aj v prípade, že objevíme vhodný vzor, jeho začlenenie nie je „doslovné“ prebratie vzoru. Návrhové vzory nie sú ukončené riešenia v podobe zdrojového kódu ani návody ako ho napísať. Toto prebratie sa odohráva iba na úrovni nápadu, podstaty.

2. Cieľ práce, metodika práce a metódy skúmania

2.1 Cieľ práce

Cieľom práce je definovať základné charakteristiky a pojmy v aspektovom programovaní. Popísať programové vzory a na ukážkach demonštrovať spôsob použitia aspektov pri programovaní.

Za čiastkové ciele, ktoré podmieňujú dosiahnutie hlavného cieľa, je možné považovať preskúmanie objektovo-orientovaných návrhových vzorov a OOP ako kontextu pre AOP a aspektovú implementáciu objektovo-orientovaných a aspektovo-orientovaných návrhových vzorov a zároveň poukázať na ich rozdielnosti.

2.2 Metodika práce a metódy skúmania

Objektom skúmania sú návrhové vzory a AOP. Výsledkom je súbor reprezentatívnych návrhových vzorov. Pracovný postup vytvorenia takéhoto katalógu vzorov spočíval v zhormažďovaní údajov o danom objekte skúmania a určenia všeobecných vlastností návrhových vzorov.

Spôsobom získavania údajov bolo cielené vyhľadávanie informácií v rámci publikovaných prác so zameraním na návrhové vzory a AOP. Zdrojom informácií bol najmä internet a domáce a zahraničné knižnice. Všetka použitá literatúra je uvedená na konci tejto práce.

Napriek tomu, že zoznam použitej literatúry obsahuje viacero položiek, pre túto prácu boli ako východiská dôležité tri z nich:

- Návrhové vzory: Elementy znovupoužiteľného objektovo-orientovaného softvéru, ktorá predstavuje východisko pre opis vzoru [12]
- Implementácia návrhových vzorov v Jave a AspectJ, ktorá predstavuje východisko pre implementácie [15]
- Modularizácia návrhových vzorov pomocou aspektov: kvantitatívna štúdia, ktorá predstavuje východisko pre porovnávanie implementácií [14]

Rozhodnutia ohľadom štruktúry opisu návrhového vzoru boli učinené s prihliadnutím na to, že sú tu pritomné tri varianty každého vzoru: všeobecný, objektovo-

orientovaný a aspektovo-orientovaný variant. Zoznam vlastností návrhového vzoru, ktoré boli spracované, sa nachádza v Tabuľke č. 5.

VLASTNOSŤ	Charakteristika
Názov	všeobecne známe anglické pomenovanie vzoru
Slovenský názov	slovenský preklad anglického názvu
Význam	**„**motto“ vzoru, výstižný opis vzoru
Motivácia	dôvody, prečo by mal byť návrhový vzor implmentovaný a jeho prínos
Participanti	účastníci vzoru, ktorí zohrávajú nejakú úlohu v jeho implementácii
Všeobecná štruktúra	opisuje základný model vzoru
OOP implementácia	opis základnej štruktúry objektovo-orientovanej implementácie
AOP implementácia	opis základnej štruktúry aspektovo-orientovanej implementácie
Porovnanie	zhrnutie kvalitatívneho a knantitatívneho zhodnotenie vzoru
Príbuzné vzory	vzory, ktoré majú spoločné črty s daným vzorom

Tabuľka č. 5 Štruktúra opisu návrhového vzoru

3. Výsledky práce a diskusia

Ciele práce uvedené v predchádzajúcej kapitole boli premietnuté do výsledkov práce. Táto kapitola je venovaná popisu programových vzorov a na ukážkach sa snaží demonštrovať spôsob použitia aspektov pri programovaní.

Návrhové vzory sú silným nástrojom a veľkou pomocou pre programátorov a softvérových návrhárov pri riešení opakovane sa vyskytujúcich problémov. Spracované návrhové vzory môžu byť použité ako učebné príklady, pretože vystihujú niektoré základné črty programovacích techník, ktorých priblíženie, lepšie pochopenie a ovládanie bolo jedným zo zámerov prvotnej tvorby návrhových vzorov. Po osvojení si návrhových vzorov je možné budovať na týchto znalostiach, vedome hľadať ďalšie vzory, nielen návrhové, a tak urýchliť svoje učenie a zdokonaľovať svoje dizajnérske schopnosti.

Užitočnosť spracovaných vzorov sa môže prejaviť pri rozhodovaniach týkajúcich sa použitia aspektovo-orientovanej paradigmy.

V prvom rade ide o otázku, či aspektovo-orientovaný jazyk vôbec použiť, t. j. či jeho nasadenie bude prínosné. Príčinou je, že aspektovo-orientovaná paradigma je relatívne nová a používaná skôr zriedkavo, preto je potrebné venovať jej patričnú pozornosť a písať o nej.

Návrhové vzory, v prípade, že sú známe subjektu rozhodovania v objektovo-orientovanej podobe sa stávajú jednoduchšie pochopiteľné aj v aspektovo-orientovanej podobe, a teda môžu byť veľmi vhodným prostriedkom na podporu rozhodovania. Ako východisko pri opise návrhových vzorov je preto výhodné použiť objektovo-orientované implementácie a tieto následne pretvoriť na aspektovo-orientované, čo je postup, ktorý bol dodržaný aj v tejto práci a súčasne proces, na ktorom je možné poukázať na rozdielnosti medzi oboma paradigmami.

V druhom rade vzniká otázka, ktorá úzko súvisí s predchádzajúcou, t. j. či je výhodnejšie použiť aspektovo-orientovanú verziu návrhového vzora v jazyku AspectJ ako objektovo-orientovanú, pričom podstata tkvie vo fakte, že je možné použiť v jazyku AspectJ aj objektovo orietovanú vereziu v jazyku Java, pretože tieto vzory sú kompatibilné a všetky programy napísané v jazyku Java sú aj platnými programami v jazyku AspectJ, čo znamená, že je potrebné sa rozhodnúť, ktorá verzia bude mať väčší prínos. Táto práca sa snaží byť pomôckou pri tomto type rozhodovania ohľadom návrhových vzorov, ktoré spracováva.

Na druhej strane je nutné pripomenúť, že takéto rozhodovania vo veľkej miere závisia od situácie, od typu programu a od samotného programátora alebo softvérového návrhára. Bez bližších informácií nie je možné vytvoriť jednoznačné odporúčanie použiteľné vo všetkých situáciách, ale iba poskytnúť základné charakteristiky, ktoré je možné brať do úvahy pri vyššie spomenutých rozhodovaniach.

Avšak stanovenie je takýchto charakteristík nanovo je nad možnosti tejto práce, pretože si to vyžaduje hlavne prax, rozsiahle vedomosti a prehľad. Ako východisko boli preto použité práce odborníkov. Keďže je aspektovo-orientovaná paradigma preskúmaná v tomto smere iba neveľmi, nepanuje medzi nimi všeobecne uznávaný konsenzus, ktorý by určoval charakteristiky vhodné na tento účel.

Práca Implementácia návrhových vzorov v Jave a AspectJ [15] je takmer nevyhnutne prítomná v prácach rozoberajúcich použitie návrhových vzorov v AspectJ. Najpravdepodobnejším dôvodom bude asi ten, že jej spoluautorom je Gregor Kiczales, autor jazyka AspectJ. Medzi takéto práce patria: [14, 16, 17, 18, 19, 20, 21, 22, 23, 24]. Niektoré z nich poukazujú na fakt, že charakteristiky, ktoré boli autormi tejto práce uprednostnené (lokalita, znovupoužiteľnosť, komponovateľnosť a pripojiteľnosť) sú veľmi neobvyklé, iba kvalitatívne a nimi vytvorené príklady si vyžadujú úpravy, napr. práca Alessandra Garciu a kol. Modularizácia návrhových vzorov pomocou aspektov: kvantitatívna štúdia je venovaná upraveným vzorom.

Ďalšie práce, ako napr. Zdokonalenie modularity s AOP [23] od Sérgia Brytona a O multiparadigmových softvérových metrikách komplexnosti [24] od Adama Siposa výber metrík v predchádzajúcich dielach spochybňujú a snažia sa poskytnúť iné.

Nejednotnosť v tejto oblasti je dôvodom, prečo je táto práca zameraná skôr na opis štruktúry vzorov ako na zhrnutie viacerých ich charakteristík, pričom ich však nevynecháva úplne a v rámci porovnávania je poukázané na závery prác [14] a [15]. Tieto a mnohé iné práce často poukazujú na vzory z vyššie spomenutého diela, no ich podrobnému vysvetleniu sa nevenujú.

3.1 Návrhové vzory v AspectJ - východiská

Napriek tomu, že pri novovytvorených aspektovo-orientovaných vzoroch by bolo jednoduchšie poukázať na črty aspektovo-orientovaného programovania, boli objektovo-orientované návrhové vzory uprednostnené. Jazyk AspectJ je kompatibilný s jazykom Java do takej miery, že celá syntax, všetky programy napísané v jazyku Java sú použiteľné a platné súčasne aj v jazyku AspectJ. To znamená, že všetky návrhové vzory vytvorené v jazyku Java sú implemenovateľné bez akejkoľvek zmeny aj do jazyka AspectJ.

Na druhej strane je možné ich upraviť tak, aby zahŕňali aspektovo-orientované prvky a teda aj prínos, ktorý aspektovo-orientovaná technológia ponúka, ak je teda takáto úprava možná.

Keďže doteraz v celej histórií odbornej literatúry venovanej softvérovým návrhovým vzorom nebolo napísané jediné dielo, ktoré by nevychádzalo, necitovalo alebo aspoň nekritizovalo dielo Návrhové vzory: Elementy znovupoužiteľného objektovo-orientovaného softvéru, a teda bolo by veľmi obtiažne sa mu vyhnúť v tejto práci. Je to fundamentálne dielo a doposiaľ svojím prínosom a významom nebolo prekonané.

3.1.1 GoF návrhové vzory a AspectJ

Jedným z najlepších príkladov spojenia návrhových vzorov a jazyka AspectJ je práca Implementácia návrhových vzorov v Jave a AspectJ od Gregora Kiczalesa a Jana Hannemanna zaoberajúca sa porovnaním implementácií návrhových vzorov v Jave a v AspectJ [15]. Návrhové vzory, ktoré práca skúma nie sú žiadne iné ako 23 klasických vzorov od GoF a autormi sú známi softvéroví experti, pričom Gregor Kiczales je jeden z tvorcov jazyka AspectJ.

Porovnávaním všetkých návrhových vzorov dospeli k záveru, že až pri 17-tich z nich došlo v aspektovo-orientovanej podobe k modulárnym vylepšeniam a pri jednom zostala implementácia zachovaná. Tieto výsledky je možné vidieť v Tabuľke č. 6. Vlastnosti návrhových vzorov, na ktorých bol tento výskum založený, sú: lokalita, znovupoužiteľnosť, komponovateľnosť a pripojiteľnosť.

Štyri uvedené vlastnosti sú dôležité pre ich vzťah k modularizácií, t. j. či napomáhajú aspektovosti. Lokalita (angl. locality) je vlastnosť implementácie návrhového vzoru, ktorá hovorí o umiestnení kódu do aspektov, o vzájomných vzťahoch medzi aspektmi a participujúcimi triedami, ktoré určujú mieru zlepšenia modularizácie.

Znovupoužiteľnosť (angl. reusability) ako možnosť opätovného využitia. Komponovateľnosť (angl. composition transparency) ako rozlíšiteľnosť viacerých inštancií toho istého vzoru v programe, prípadne použitie viacerých vzorov a miera ich vplyvu na celkovú zrozumiteľnosť a prehľadnosť programu. Pripojiteľnosť (angl. pluggability) ako vlastnosť participantov určujúca použitie a inštančnú špecifickosť vzorov.

Spoločným znakom vzorov, pri ktorých došlo v AOP verzii k lepším výsledkom pri porovnaní, je, že v určitej podobe obsahujú pretínajúce koncerny lokalizované v takýchto verziách do aspektov. Niektoré z týchto aspektov môžu byť dokonca použité pre viacero návrhových vzorov, najmä aspekty zamerané na implementáciu funkcionality participantov.

DRUH VZORU	Vzor	Lokalita	Znovupoužiteľnosť	Komponovateľnosť	Pripojiteľnosť
Vytvárajúce vzory	Abstract Factory	nie	nie	nie	nie
	Builder	nie	nie	nie	nie
	Factory Method	nie	nie	nie	nie
	Prototype	**áno**	**áno**	**áno**	**áno**
	Singleton	áno	áno	nedef.	áno
Štrukturálne vzory	Adapter	áno	nie	áno	áno
	Bridge	nie	nie	nie	nie
	Composite	**áno**	**áno**	**áno**	**áno**
	Decorator	áno	nie	áno	áno
	Facade	*Rovnaká implementácia*			
	Flyweight	**áno**	**áno**	**áno**	**áno**
	Proxy	áno	nie	áno	áno
Behaviorálne vzory	**Chain of Responsibility**	**áno**	**áno**	**áno**	**áno**
	Command	**áno**	**áno**	**áno**	**áno**
	Interpreter	nie	nie	nie	nie
	Iterator	**áno**	**áno**	**áno**	**áno**
	Mediator	**áno**	**áno**	**áno**	**áno**
	Memento	**áno**	**áno**	**áno**	**áno**
	Observer	**áno**	**áno**	**áno**	**áno**
	State	áno	nie	nedef.	áno
	Strategy	**áno**	**áno**	**áno**	**áno**
	Template Method	áno	nie	nie	áno
	Visitor	**áno**	**áno**	**áno**	**áno**

Tabuľka č. 6 Kvalitatívne porovnanie OOP a AOP vzorov

3.1.2 Návrhové vzory ako nástroj na porovnávanie paradigiem

Vo vyššie spomenutej práci Implementácia návrhových vzorov v Jave a AspectJ neboli uvedené kvantitatívne porovnania, iba výsledky výskumu vo áno/nie forme ako odpoveď na otázku, či prichádza k zlepšeniu vlastností modularizácie daných návrhových vzorov, založenú na odhadoch a skúsenostiach oboch expertov. Ak by sme chceli dospieť k numerickým hodnotám môžeme použiť viacero metodík, pred ich opisom je však žiaduce vysvetliť, aké sú dôvody na vytváranie takýchto analýz.

Aspektovo-orientovaná paradigma je označovaná ako nástupnícka paradigma po objektovo-orientovanej paradigme a na podporu jej rozšírenia by bolo najlepšie poskytnúť presvedčivé, nielen kvalitatívne, dôkazy. Návrhové vzory z dielne GoF [12] sa javia ako potencionálny, paradigmovo- a jazykovo-nezávislý kandidát na porovnávanie týchto paradigiem, pretože sú dostatočne známe, boli vytvorené pre OOP a AOP ich prítomnosť nijako neobmedzuje.

Na uskutočnenie komparatívnej kvantitatívnej analýzy je nevyhnutné definovať metriky, na ktorých bude porovnávanie založené. Práca Alessandra Garciu a kol. Modularizácia návrhových vzorov pomocou aspektov: kvantitatívna štúdia doplňuje Implementácia návrhových vzorov v Jave a AspectJ o kvantitatívne metriky, ktoré sú uvedené v Tabuľke č. 7.

VLASTNOSŤ	**Metriky**
Separácia koncernov	**CDC** (angl. Concern Diffusion over Components) **CDO** (angl. Concern Diffusion over Operations), **CDLOC** (angl. Concern Diffusions over Lines of Code)
Zviazanosť (angl. coupling),	**CBC** (angl. Coupling Between Components), **DIT** (angl. Depth Inheritance Tree)
Súdržnosť (angl. cohesion)	**LCOO** (angl. Lack of Cohesion in Operations)
Veľkosť (angl. size)	**LOC** (angl. Lines of Code) **NOA** (angl. Number of Attributes) **WOC** (angl. Weighted Operations per Component)

Tabuľka č. 7 Druhy metrík

Výsledky kvantitatívneho porovnania boli analyzované a pre jednotlivé vzory boli rozdelené do skupín so spoločnými vlastnosťami.

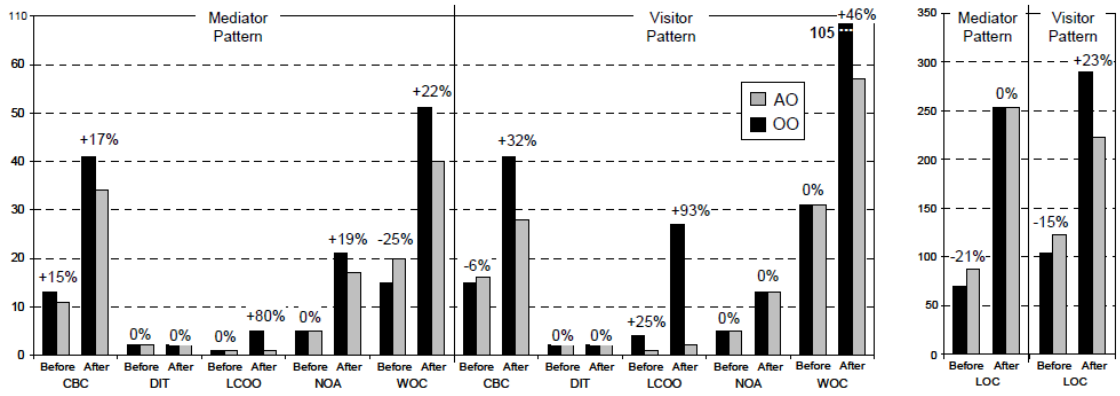
Pre separáciu koncernov boli vytvorené skupiny:

- zvýšená separácia
- znížená separácia
- bez efektu.

Pre zvyšné tri druhy vlastností návrhových vzorov boli vytvorené nasledujúce skupiny:

- lepšie výsledky pre AOP verzie
- lepšia väčšina výsledkov pre AOP verzie
- lepšie výsledky pre OOP verzie
- lepšia väčšina výsledkov OOP verzie
- bez efektu.

Na obrázku č. 13 sú znázornené percentuálne výsledky porovnania pre vzory Mediator a Visitor [14].



Obrázok č. 14 Porovnanie návrhových vzorov Mediator a Visitor

Výsledky tohto výskumu nie je možné zhrnúť do vyjadrenia typu „pri takomto počte návrhových vzorov dochádza k zlepšeniu modularizácie“, pretože jednotlivé metriky môžu byť pre jeden vzor rozdielne, t. j. môžu podporovať alebo nepodporovať takéto tvrdenie. Okrem toho, bývajú závery tejto práce niekedy spochybňované, pretože výskum nie je založený na formalizovaných metrikách.

Ďalšími prácami s témou kvantítavnych analýz sú napr. Ohodnotenie aspektovej modularizácie s použitím matice dizajnovej štruktúry a NOV (angl. net option value) od Sushila Bajracharya, Evolučný model na ohodnotenie softvérovej modularity od Yuangfanga Caia, Zdokonalenie modularity s AOP od Sérgia Brytona a O multiparadigmových softvérových metrikách komplexnosti (AV-grafy) od Adama Siposa a kol., no potýkajú sa s obdobnými nedostatkami.

3.1.3 Aspektovo-orientované návrhové vzory

Aspektovo-orientovaná paradigma nie je odkázaná iba na objektovo-orientované návrhové vzory, ale disponuje svojimi vlastnými aspektovo-orientovanými vzormi. Medzi najznámejšie patria:

- Cuckoo's Egg
- Proceed Object
- Eager Pointcut Decision
- Notbelow
- Director
- Policy
- Wormhole
- Participant
- Border Control
- Exception Introduction

3.2 Výber návrhových vzorov

Výber návrhových vzorov bol založený na jednoduchom princípe. Vybrané boli tie vzory, ktoré dosiahli najlepšie hodnotenie (v porovnaní s objektovo-orientovanými verziami) v práci Implementácia návrhových vzorov v Jave a AspectJ [15], pretože by na nich malo byť najlepšie vidieť použitie aspektov. Tieto návrhové vzory sú uvedené v Tabuľke č. 8.

DRUH VZORU	Vzor	Lokalita	Znovupoužiteľnosť	Komponovateľnosť	Pripojiteľnosť
Vytvárajúce vzory	**Prototype**	**áno**	**áno**	**áno**	**áno**
	Singleton	áno	áno	nedef.	áno
Štrukturálne vzory	**Flyweight**	**áno**	**áno**	**áno**	**áno**
	Composite	**áno**	**áno**	**áno**	**áno**
	Proxy	áno	nie	áno	áno
Behaviorálne vzory	**Strategy**	**áno**	**áno**	**áno**	**áno**
	Obsever	**áno**	**áno**	**áno**	**áno**

Tabuľka č. 8 Vybrané návrhové vzory

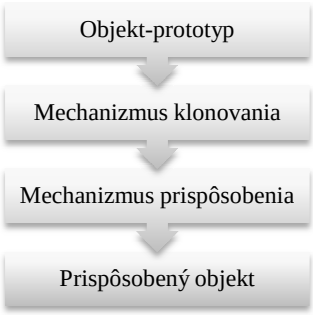
3.3 Vybrané implementácie vytvárajúcich návrhových vzorov

Pre vytvárajúce návrhové vzory (angl. creational design patterns) je charakteristické, čo naznačuje aj ich názov, že sa zaoberajú vytváraním objektov.

Podľa GoF definície ide o: „Vytvárajúce návrhové vzory konspektujú instanciačný proces. Pomáhajú systému stať sa nezávislým od toho ako sú objekty vytvárané, komponované a reprezentované.“[12, s. 81]

3.3.1 Návrhový vzor Prototype

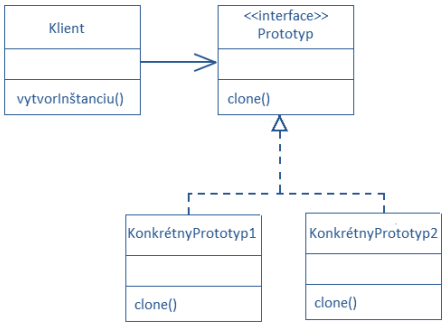
Názov	Prototype
Slov. názov	Prototyp
Význam	Vytváranie kópií objektu-protypu jednotným spôsobom.
Motivácia	Zjednodušenie vytvárania objektov, pokiaľ ich vytváranie je náročné, vyžaduje si veľa zdrojov alebo použitie operátora new je nežiaduce.



Obrázok č. 15 Symbolická štruktúra návrhového vzoru prototyp

Slúži ako spôsob inštanciácie objektov tried, ktoré nie sú známe v čase, keď je systém vyvíjaný. Nahradzuje potrebu triedy pre každý jeden druh objektu. Umožňuje oddelenie hierarchie tried, ktoré sa podieľajú na tvorbe objektov. Ukrýva komplexnosť vytvárania objektu pred klientom.

Štruktúra	Rozhranie Prototyp je spoločné pre všetky vytvárané objekty. KonkrétnyPrototyp1 a KonkrétnyPrototyp2 predstavujú jednotlivé prototypy, ktoré budú kopírované. Kopírovanie prototypu vyvoláva Klient.



Obrázok č. 16 Všeobecný návrhový vzor Prototyp

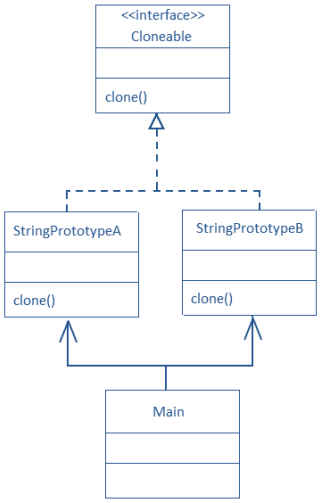
Participanti **OOP**

Klient

Prototyp rozhranie

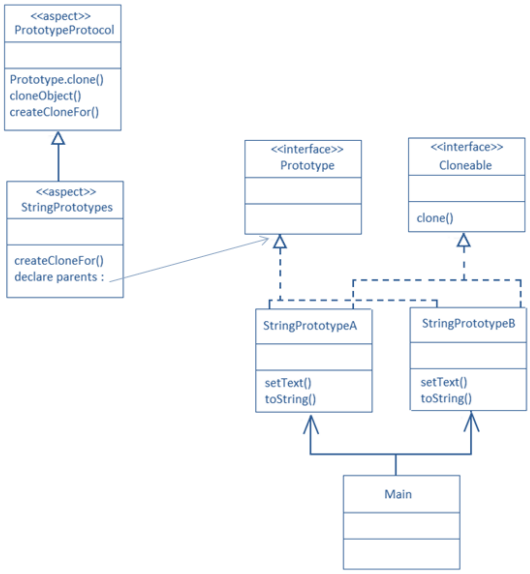
Konkrétne prototypy

OOP impl. Implementovanie metódy clone() patriacej triede java.lang.Object v rámci rozhrania Cloneable, určenej na replikáciu objektu vyvolanú objektom samotným.



Obrázok č. 17 OOP návrhový vzor Prototyp

AOP impl. Klonovanie je ako pretínajúci koncern modularizované do dvoch aspektov: PrototypeProtocol a StringPrototypes.



Obrázok č. 18 Všeobecný návrhový vzor Prototyp

Aspekt PrototypeProtocol definuje rozhranie Prototype a priraďuje mu metódu Prototype.clone(). Aspekt StringPrototypes je potomkom aspektu PrototypeProtocol a používa medzitypovú deklaráciu na deklarovanie vzťahov implementácie rozhrania Prototype pre triedy StringPrototypeA a StringPrototypeB.

```
protected interface Prototype {}

public Object Prototype.clone() throws CloneNotSupportedException
    {
        return super.clone();
    }
```

Obrázok č. 19 Ukážka z aspektu PrototypeProtocol [15]

Okrem implementácie rozhrania Cloneable s metódy clone() je možné túto metódu s modifikátorom prístupu protected prekryť použitím @Override.

Porovnanie

Jednou z výhod AOP implementácie je lokalizovanie klonovacieho mechanizmu na jednom mieste, čo môže byť v prípade viacerých StringPrototype tried, ktorých objekty sa majú klonovať, ekonomickejšie pri vzniku zmeny, prehľadnejšie a zrozumiteľnejšie.

Nevýhodou AOP implementácie je, že ak by vznikla potreba klonovať objekty rôznym spôsobom, bolo by to nemožné dosiahnuť iba za pomoci zmeny jedného aspektu.

Porovnanie OOP a AOP verzie z práce Implementácia návrhových vzorov v Jave a AspectJ je uvedená v Tabuľke č. 6 na strane 34. V kvantitatívnej štúdii Modularizácia návrhových vzorov pomocou aspektov [15] bol návrhový vzor Prototype zaradený do skupiny vzorov so zlepšením separácie koncernov pre AOP verziu. Na druhej strane bol tento vzor v hodnotení zviazanosti (angl. coupling), súdržnosti (angl. cohesion) a veľkosti zaradený do skupiny vzorov s lepšími výsledkami pre verziu OOP [14].

Tento vzor nie je možné považovať za vhodný učebný príklad AOP programu, pretože neobsahuje programovú konštrukciu bodový prierez – videnie, ale je založený na medzitypovej deklarácii.

Rovnako priradenie rozhraniu Prototype metódu clone() a implementácia rozhraní Cloneable pôsobí duplicitne.

Príbuzné vzory Abstract Factory

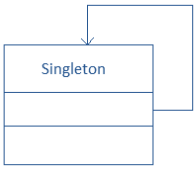
3.3.2 Návrhový vzor Singleton

Názov Singleton

Slov. názov Jedináčik

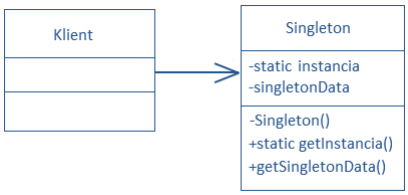
Význam Vytvorenie maximálne jednej (unikátnej) inštancie objektu.

Motivácia Použitie návrhového vzoru Singleton na dosiahnutie kontroly nad inštanciáciou triedy, pričom nie je nutný výskyt globálneho riadiaceho objektu, postačuje globálny prístupový bod, a obmedzenie platnosti volaní operátora new. Takými prípadmi môžu byť: determinácia používania určitého zdroja, bezpečnostné opatrenia, na zabránenie viacnásobného prístupu alebo zabránenie náhodnému vytvoreniu inštancie.



Obrázok č. 20 Symbolické znázornenie návrhového vzoru Singleton

Štruktúra Klient môže inštanciovať triedu Singleton iba prostredníctvom statickej metódy getInstancia() a nie je potrebné, aby sa zaoberal, či už existuje nejaká inštancia triedy Singleton.



Obrázok č. 21 Všeobecné znázornenie návrhového vzoru Singleton

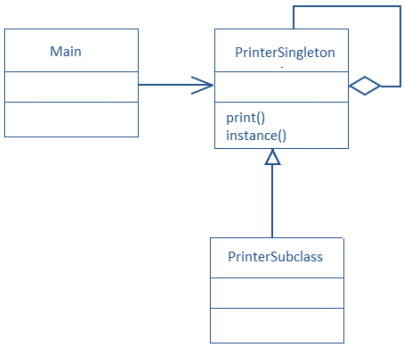
Trieda Singleton obsahuje statický člen instancia obsahujúci referenciu na objekt, ktorý má byť jediný, prvotne je mu priradená hodnota NULL. Na to, aby bolo možné vytvoriť inštanciu musí byť prítomná metóda getInstancia(), ktorá ju v prípade, že hodnota instancia je NULL, vytvorí inštanciu triedy Singleton. Prístupový modifikátor konštruktora triedy Singleton je private alebo protected, aby sa zabránilo priamej inštanciácii.

Participanti **OOP**

Singleton – trieda

Klient

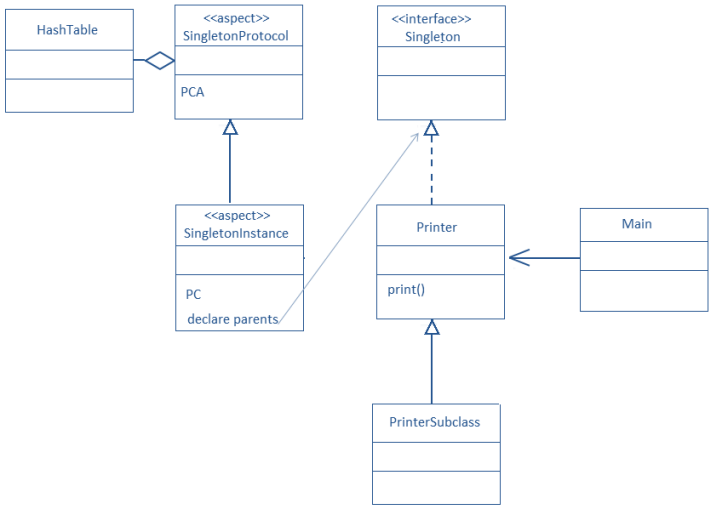
OOP impl. PrinterSingleton predstavuje Singleton triedu, ktorá by mala vytvoriť iba jednu inštanciu pomocou metódy instance().



Obrázok č. 22 Znázornenie OOP návrhového vzoru Singleton

Volanie metódy instance() zabezpečuje prácu s vždy tou istou inštanciou. Tento príklad zároveň poukazuje na funkciu modifikátora protected, ktorý umožňuje prístup ku konštruktoru z tried v rovnakom balíku ako je PrinterSubclass, podtrieda triedy PrinterSingleton schopná vytvoriť pomocou super() ďalšie inštancie triedy PrinterSingleton.

AOP impl. Do AOP implementácie boli doplnené dva aspekty: SingletonProtocol a SingletonInstance, ktorý je potomkom aspektu SingletonProtocol.



Obrázok č. 23 Znázornenie AOP návrhového vzoru Singleton

Aspekt SingletonProtocol obsahuje around() advice, naviazanú na volanie konštruktora Singleton(). Časť advice pred proceed() je podmienka určená na zistenie, či HashTable obsahuje nejakú inštanciu.

Popri aspektoch bolo pridané aj rozhranie Singleton definované v aspekte SingletonProtocol a priradené k triede Printer v aspekte SingletonInstance.

```
public interface Singleton {}

protected pointcut protectionExclusions();

 Object around(): call((Singleton+).new(..)) && !protectionExclusions()
 {
     Class singleton = thisJoinPoint.getSignature().getDeclaringType();
         if (singletons.get(singleton) == null)
         {
             singletons.put(singleton, proceed());
         }
         return singletons.get(singleton);
 }
```

Obrázok č. 24 Ukážka z aspektu SingletonProtocol [15]

Porovnanie AOP verzia presúva zodpovednosť za kontrolu množstva inštancií do aspektov. V OOP verzii boli táto kontrola roztrúsená

Nie nevyhnutne musí byť použitie AOP verzie na vytváranie jedinečných objektov, mže slúžiť aj ako továrenská metóda. Advice around() zabezpečuje jedinečnosť inštancií.

Tabuľka č. 6 na strane 34 obsahuje výsledky porovnania zo štúdie Implementácia návrhových vzorov v Jave a AspectJ [15]. Práca Modularizácia návrhových vzorov pomocou aspektov [14] zistila, že dochádza k zlepšeneniu separácie koncernov pre AOP verziu a vyššie hodnoty všetkých metrík okrem CBC a zaradila tento vzor do skupiny - lepšia väčšina výsledkov pre AOP verzie.

Príbuzné vzory Abstract Factory, Facade

3.4 Vybrané implementácie štrukturálnych návrhových vzorov

Štrukturálne návrhové vzory sú zamerané na definovanie vzťahov medzi objektmi, určujú spôsoby, akými sú objekty komponované do väčších štruktúr.

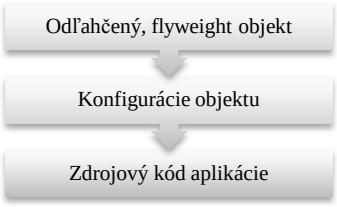
3.4.1 Návrhový vzor Flyweight

Názov Flyweight

Slov. názov Mušia váha

Význam Použitie zdieľania v systémoch s veľkým počtom objektov.

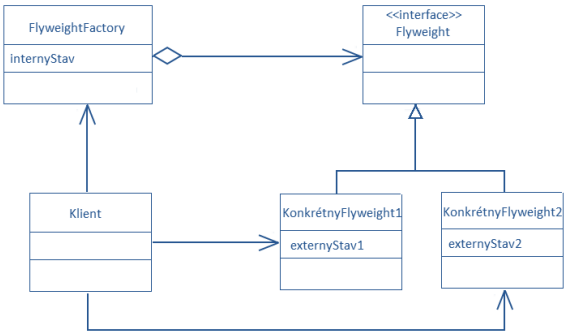
Motivácia Cieľom použitia návrhového vzoru Flyweight je zjednodušenie a zefektívnenie práce so zdrojmi založenej na zdieľaní objektov. Tento návrhový vzor rovnako umožňuje konzistenciu objektov.



Obrázok č. 25 Návrhový vzor Flyweight znázornený symbolicky

Obrázok č. 25 zjednodušene znázorňuje použitie návrhového vzoru Flyweight.

Štruktúra Každý odľahčený, flyweight, objekt je možné rozdeliť na dve časti: na externú časť (závislú na stave, externý stav) a internú časť (nezávislú na stave, interný stav). Časť závislú na stave má na starosti trieda FlyweightFactory. Časť nezávislá na stave je zdieľaná aj ostatnými odľahčenými objektmi.



Obrázok č. 26 Návrhový vzor Flyweight znázornený všeobecne

FlyweifgtFactory v podstate vytvára doplniteľnú, konfigurovateľnú šablónu objektov. Klient požaduje od FlyweifgtFactory objekt určitého typu a pokiaľ už bol vytvorený, bude mu poskytnutá referencia na neho a pokiaľ nie, bude pre neho vytvorený.

Participanti

OOP

FlyweightFactory – trieda

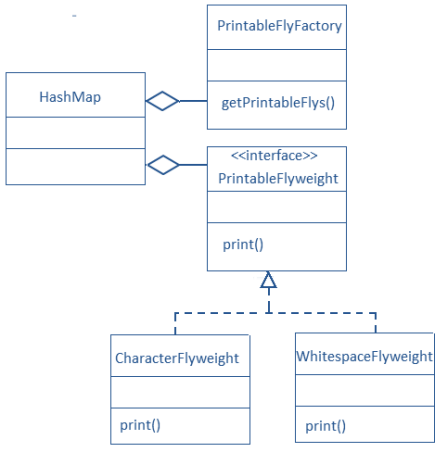
Flyweight – rozhranie

Konkrétne odľahčenia

Klient

OOP impl.

Účelom implementácie návrhového vzoru Flyweigt je vytvorenie zdieľaných objektov pre spracovávanie textu, resp. písmen, znakov, a medzier.



Obrázok č. 27 OOP návrhový vzor Flyweight

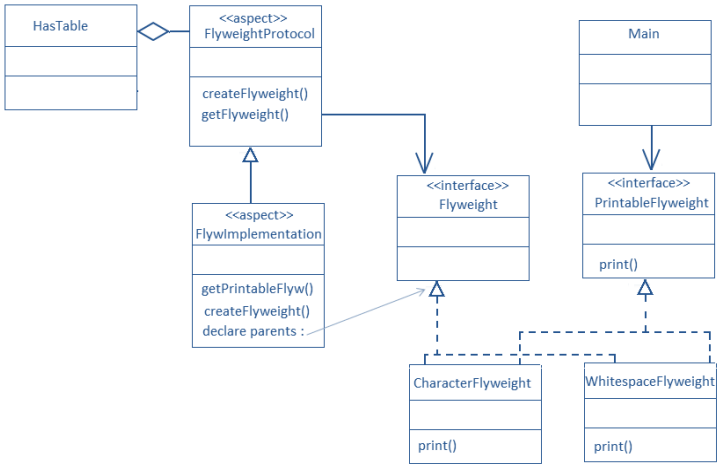
Triedy CharacterFlyweight a WhitespaceFlyweight implementujú rozhranie ProntableFlyweight. PrintableFlyweightFactory zisťuje pomocou HashTable, či v nej už daný znak nachádza, a teda či môže byť zdieľaný, alebo či musí byť do nej vložený.

Externým stavom Character objektov, t. j. vlastnosť konkrétneho znaku, je v tomto príklade veľkosť (či ide o veľké/malé písmeno).

WhitespaceFlyweight je trieda určená na spracovanie medzier.

AOP impl.

Pretínajúce koncerny boli premiestnené v AOP verzii z triedy PrintableFlyweightFactory do aspektov FlyweightImplementation a FlyweightProtocol.



Obrázok č. 28 AOP návrhový vzor Flyweight

FlyweightImplementation slúži na implementáciu rozhrania Flyweight do tried CharacterFlyweight a WhitespaceFlyweight a je potomkom aspektu FlyweightProtocol.

```
declare parents: CharacterFlyweight implements Flyweight;

declare parents: WhitespaceFlyweight implements Flyweight;
```

Obrázok č. 29 Ukážka z aspektu FlyweightImplementation [15]

FlyweightProtocol je určený na preverenie Hashtable, kam ukladá objekty Flyweight. Klientovi (triede Main) sú ale predávané PrintableFlyweight objekty.

Porovnanie

V kvalitatívnom hodnotení dopadol návrhový vzor Flyweight v AOP verzii veľmi dobre, všetky štyri kritériá (lokalita, znovupoužiteľnosť, komponovateľnosť a pripojiteľnosť) mali vyššie hodnoty ako v OOP verzii.

Kvantitatívne porovnanie [14] týchto verzií však v prospech AOP verzie nehovorí. Všetky metriky pre separáciu koncernov,

zviazanosť, súdržnosť a veľkosť sú v priemere o 33% vyššie pri OOP verzii.

To však neznamená, že tento vzor v AOP verzii je zavrhnutiahodný, iba že táto konkrétna implementácia aspektov nie je najvhodnejšia a veľmi nelíši od OOP tried.

Príbuzné vzory Strategy, State, Interpretator, Composite

3.4.2 Návrhový vzor Composite

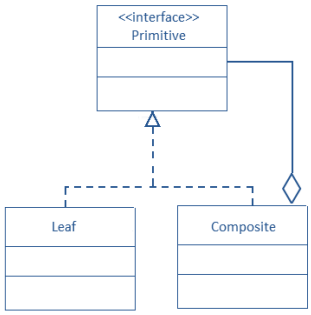
Názov Composite

Slov. názov Zloženina

Význam Kombinácia niekoľkých objektov do jedného, ktorý má rovnaké správanie ako objekty, ktoré obsahuje.

Motivácia Návrhový vzor Composite slúži na uľahčenie práce s primitívnymi objektmi, ktoré zoskupuje do jedného celku – Composite a umožňuje prácu s týmto celkom ako s primitívnym objektom.

Všeobecná štruktúra Rozhranie Primitive je abstrakciou pre primitívne objekty a je implementované aj primitívnymi Leaf objektami aj Composite objektami. Composite objekty sú tvorené primitívnymi objektmi a musia implementovať tú istú metódu deklarovanú rozhraním ako aj primitívne objekty, no v Composite objektoch musí ísť o kombináciu všetkých primitívnych.



Obrázok č. 30 Všeobecný tvar návrhového vzoru Composite

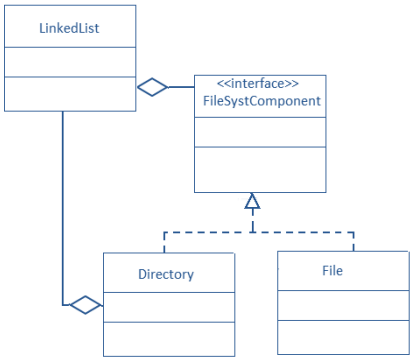
Participanti **OOP**

Primitive – rozhranie

Leaf - primitívne objekty

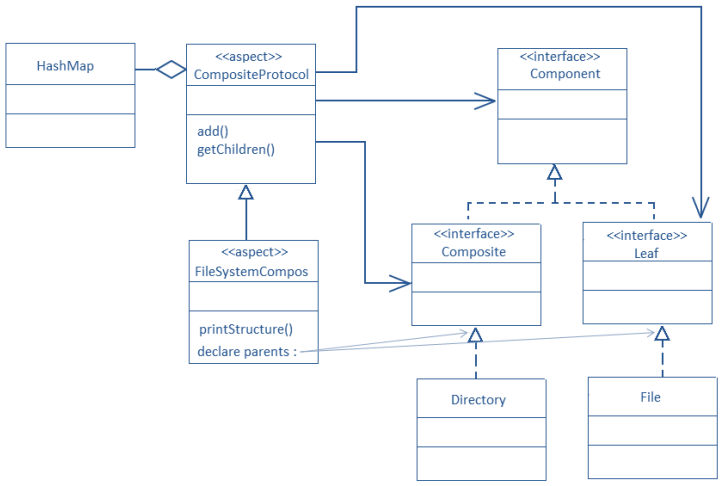
Composite objekty

OOP impl. Nasledujúca implementácia systému súborov pozostáva z rozhrania FileSystemComponent, ktoré implementujú triedy File (primitívna trieda) a Directory(Composite trieda). LinkedList je dátová štruktúra uchovávajúca obsah Directory.



Obrázok č. 31 OOP tvar návrhového vzoru Composite

AOP impl. V aspektovo-orientovanom variante sa nachádzajú iba dva aspekty: CompositeProtocol a FileSystemComposition, ktoré sú vo vzťahu rodič-potomok.



Obrázok č. 32 AOP tvar návrhového vzoru Composite

Všetky prítomné rozhrania sú deklarované v aspektoch.

CompositeProtocol deklaruje: Component, Visitor, FunctionVisitor, Composite a Leaf rozhrania, pričom Composite a Leaf sú potomkami Component rozhrania. CompositeProtocol deklaruje rozhrania Leaf pre triedu File a Coposite pre triedu Directory.

```
public interface Component {}
protected interface Composite extends Component {}
protected interface Leaf extends Component {}
```

Obrázok č. 33 Ukážka z aspektu FileSystemComposition [15]

```
declare parents: Directory implements Composite;
declare parents: File implements Leaf;

public void Component.printStructure(PrintStream s)
  {...}
public void Composite.printStructure(final PrintStream s)
  {...}
public void Leaf.printStructure(PrintStream s)
  {...}
```

Obrázok č. 34 Ukážka z aspektu FileSystemComposition [15]

AOP variant neobsahuje žiadne bodové prierezy ani videnia a je celý založený na definovaní rolí triedam pomocou rozhraní.

Porovnanie

Ako by sa mohlo z Obrázka č. 32 zdať, je AOP verzia rozsiahlejšia, čo sa týka veľkosti, no celá implementácia obsahuje iba o jeden súbor viac; no ak by sme abstrahovali od funkcionality, ktorá nie je prítomná v OOP verzii, napr. výpočet celkovej veľkosti directory a. i., je veľkosť približne rovnaká.

Na základe kvalitatívneho výskumu [15] boli všetky pozitívne výsledky na strane AOP. Pri kvantitatívnych ukazovateľoch [14] bol tento vzor v AOP verzii zaradený medzi vzory so zvýšenou separáciou a súdržnosťou a so zníženou zviazanosťou.

Príbuzné vzory

Visitor, Interpreter, Flyweight, Decorator, Iterator

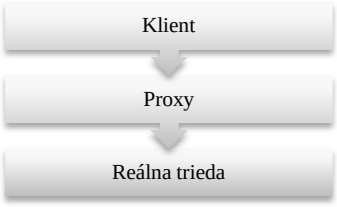
3.4.3 Návrhový vzor Proxy

Názov Proxy

Slov. názov Zástupca

Význam Riadenie prístupu k objektu prostredníctvom iného objektu – zástupcu.

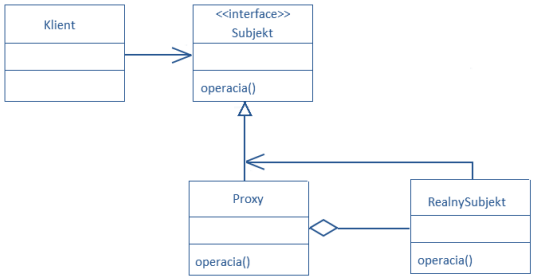
Motivácia Návrhový vzor Proxy umožňuje zamedziť priamemu prístupu k objektu, oddialiť inštanciáciu objektu a napomáha riešiť problémy so vzdialeným prístupom, rozšíriteľnosťou a výkonnosťou programov prostredníctvom zástupcu, proxy, ktorý býva pre používateľa skrytý.



Obrázok č. 35 Symbolický tvar návrhového vzoru Proxy

Ďalšia úroveň nepriameho prístupu vytvára možnosti pre rozšírenie jeho vlastností, ako napr. kontrola, distribuovanosť a komplexnosť, rovnako i potreba zmeny môže byť vykonaná iba v proxy a reálna trieda nemusí byť pozmenená. Návrhový vzor Proxy môže slúžiť na zjednodušenie rozhrania prislúchajúcemu skupine tried.

Štruktúra Trieda proxy implementuje rozhranie Subjekt a obsahuje referenciu na ReálnySubjekt alebo vie, ako ju určiť. Klient používa Proxy objekt.



Obrázok č. 36 Všeobecný tvar návrhového vzoru Proxy

Proxy volá rovnakú metódu z RealnySubjekt a môže vykonať dodatočné modifikácie, ktoré nesmú zasiahnuť klienta ani RealnySubjekt.

Participanti

OOP

Proxy – trieda

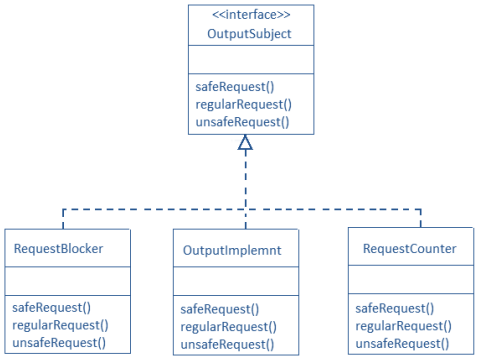
Subjekt – rozhranie

Reálny Subjekt

Klient

OOP impl.

Rozhranie OutputSubjet je implementované všetkými troma triedami: RequestBlocker, OutputImplementation a RequestCounter. Trieda OutputImplementation v tomto príklade predstavuje ReálnySubjekt a je zastupovaná pomocou proxy tried: RequestBlocker a RequestCounter.



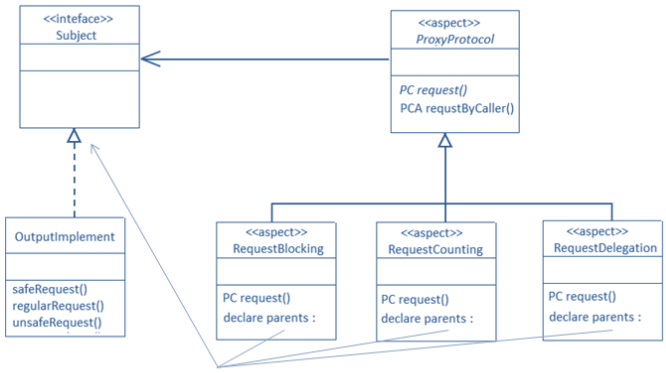
Obrázok č. 37 OOP implementácia návrhového vzoru Proxy

.

Klient, trieda Main, môže prostredníctvom objektu typu RequestBlocker, metódy na ňom použitej, vyvolať rovnomenné metódy objektov typu RequestCounter a OutputImplementation alebo prostredníctvom objektu typu RequestCounter rovnomennú metódu objektov typu OutputImplementation.

AOP impl.

Aspekty RequestBlocking, RequestCounting a RequestDelegation sú potomkami aspektu ProxyProtocol, ktorý obsahuje abstraktý pointcut request a deklaruje rozhranie Subjekt.



Obrázok č. 38 OOP implementácia návrhového vzoru Proxy

RequestBlocking, RequestCounting a RequestDelegation deklarujú pointcuty pre volanie svojej metódy: unsafe-, regular- alebo safe-Request() v OutputImplementation a RequestDelegation je určená na vyvolanie AlternateOutputImplementation v prípade volania metódy safeRequest().

```
protected interface Subject {}

protected abstract pointcut requests();

private pointcut requestsByCaller(Object caller):
      requests() && this(caller);
```

Obrázok č. 39 Ukážka z aspektu ProxyProtocol [15]

Porovnanie

AOP verzia návrhového vzoru Proxy utŕžila lepšie výsledky porovnávania. Dôvodom je najmä nezávislosť reálnej triedy od rozhrania a výhodnejším klientskym volaním, ktoré si nevyžaduje vytvorenie žiadnych proxy objektov, oni sú vtkané prostredníctvom bodov spájania do klientskeho kódu. Na druhej strane je otázne do akej miery je zabezpečená modularizácia pretínajúcich koncernov, keď v tomto príklade sú roztrúsené do proxy aspektov.

Kvalitatívne porovnanie bolo zhodné pre všetky štyri vlastnosti - zlepšenie. Kvantitatívne hodnotenie preukázalo zvýšenie separácie koncernov a tiež vyššie hodnoty väčšiny ostatných metrík, čo sa týka zviazanosti, súdržnosti a veľkosti.

Príbuzné vzory

Adapter, Decorator

3.5 Vybrané implementácie behaviorálnych návrhových vzorov

Behaviorálne návrhové vzory sú zamerané na komunikáciu medzi objektmi a prerozdelenie zodpovedností objektov prostredníctvom zapúzdrenia behaviorálnej stránky objektov a delegovanie prístupu k nej.

3.5.1 Návrhový vzor Strategy

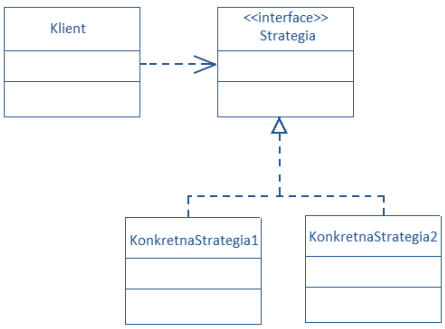
Názov Strategy

Slov. názov Stratégia

Význam Poskytuje zameniteľné varianty jedného algoritmu

Motivácia Prínos návrhového vzoru Strategy spočíva v možnosti výberu z rôznych variantov algoritmu v závislosti od kontextu. Enkapsulácia základu algoritmu prináša spôsoby ako kontrolovať podoby tohto algoritmu a biznis pravidlá. Výber správneho variantu algoritmu závisí od požiadavky klienta, presnejšie od kontextu, nie od rozhodnutia klienta alebo od druhu dát.
Z behaviorálneho hľadiska sú rôzne varianty prislúchajúce k objektom ich zodpovednosťami.

Štruktúra Rozhranie Strategia obsahuje základné kroky algoritmu a je implementované všetkými konkrétnymi algoritmami – stratégiami, ktorých každá jedna predstavuje jeden variant algoritmu, čo znamená, že algoritmy zostávajú nezávislé od klienta. t. j. separácia výberu algoritmu od jeho implementácie.



Obrázok č. 40 Návrhový vzor Strategy

Obrázok č. 40 znázorňuje zjednodušenú štruktúru návrhového vzoru Strategy.

Participanti

OOP

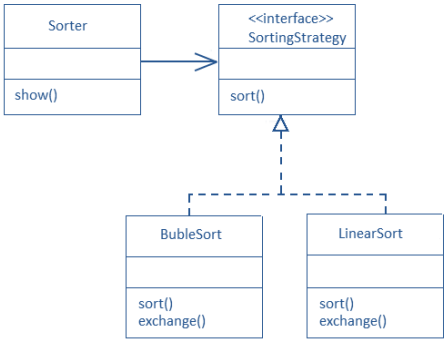
Klient – udáva kontext

Strategia – rozhranie

Konkrétne stratégie

OOP impl.

OOP Implementácia návrhového vzoru Stratégia v tomto príklade slúži na výber vhodného triediaceho algoritmu.

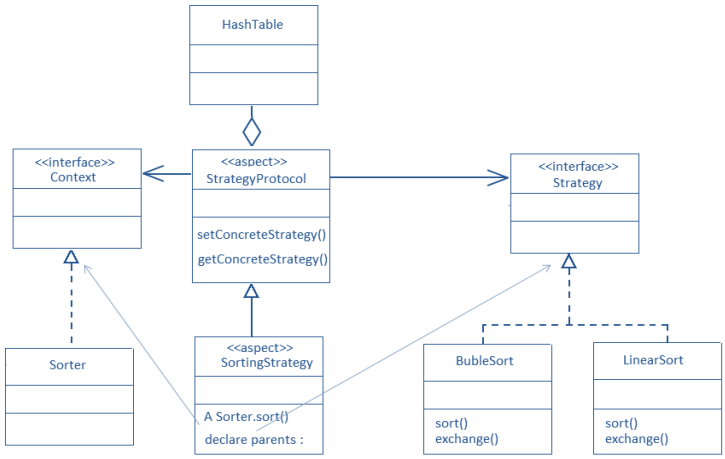


Obrázok č. 41 OOP implementácia návrhového vzoru Strategy

Triedy BubbleSort a LinearSort sú konkrétnymi podobami algorimu a implementujú rozhranie SortingStrategy. Sorter slúži ako stratégia, a vyvoláva metódu sort() objektu daného triedenia. O tom, ktorý algoritmus bude vybraný rozhoduje na základe podmienky trieda Main.

AOP impl.

V AOP implementácii zostávajú BubbleSort a LinearSort triedy zachované, trieda Sorter sa stáva prázdnou. Uvedené triediace triedy však neimplementujú svoje rozhrania priamo, ale prostredníctvom deklarácie rodiča v aspekte SortingStrategy, ktoré je potomkom aspektu StrategyProtocol, a rovnako deklaruje rozhranie Context pre triedu Sorter.



Obrázok č. 42 AOP implementácia návrhového vzoru Strategy

Aspekt StrategyProtocol deklaruje rozhrania SortingStrategy a Context a konkrétne objekty triediacich tried vkladá a vyberá z HashTable, pričom neobsahuje žiadne bodové prierezy. Tie sa nachádzajú v SortingStrategy pre volanie funkcie Sorter.sort(). Vždy, keď je táto funkcia volaná, vykoná sa videnie around(), ktoré obsahuje volania metód vybraných triediacich tried.

```
protected interface Strategy {}

protected interface Context {}

public void setConcreteStrategy(Context c, Strategy s)
 {
        strategyPerContext.put(c, s);
 }

public Strategy getConcreteStrategy(Context c)
 {
        return (Strategy) strategyPerContext.get(c);
 }
```

Obrázok č. 43 Ukážka z aspektu ProxyProtocol [15]

Porovnanie Návrhový vzor Strategy preukázal zlepšenie pre všetky kvalitatívne ukazovatele ako je možné vidieť v Tabuľke č. 6 na strane 34. Pri kvantitatívnom porovnaní sa však zaradil do skupiny s lepšími výsledkami pre väčšinu vlastností pre OO verziu.

Príbuzné vzory Flyweight

3.5.2 Návrhový vzor Observer

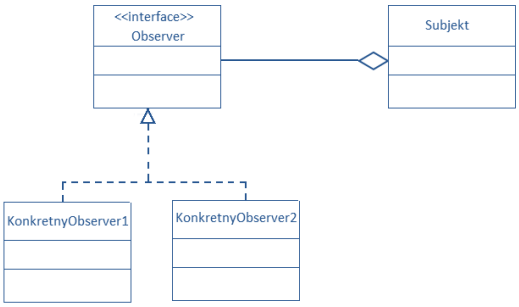
Názov Observer

Slov. názov Pozorovateľ

Význam Poskytuje spôsob, akým môže objekt upovedomiť iné objekty o udalosti, zmene

Motivácia Potreba zisťovania nastania zmeny a výskytu určitých udalostí je v programovaní veľmi rozšírená. Návrhový vzor Observer patrí medzi behaviorálne návrhové vzory a je teda určený na sprostredkovanie komunikácie medzi objektmi. Z iného pohľadu ide o vzťah závislostí medzi objektmi.

Všeobecná štruktúra Inštancie triedy subjekt predstavujú zdroje zmien a KonkretnyObserver objekty sa zaujímajú o tieto zmeny. Na to, aby im boli sprístupnené musia ich triedy implementovať rozhranie Observer.



Obrázok č. 44 Návrhový vzor Observer

Subjektu nie sú skryté objekty konkrétnych observerov, je vedomý o ich existencii a uchováva kolekciu týchto objektov. Na to používa metódu na pripojenie konkrétneho observera. Kedykoľvek nastane zmena stavu subjektu, zmena v subjekte, Subjekt upovedomí svojich konkrétnych observerov.

Participanti Subjekt

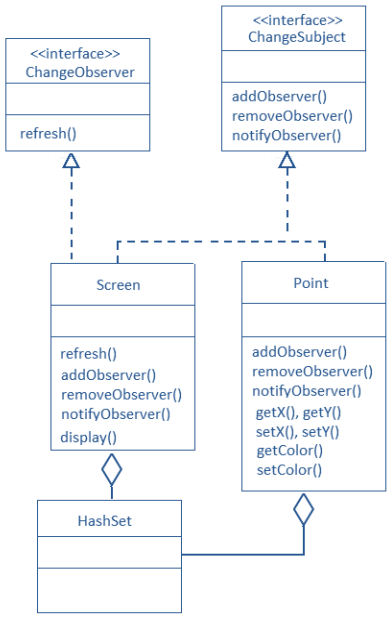
Observer – interface

Konkrétne observer triedy

OOP impl.

Návrhový vzor Observer je implementovaný za účelom pozorovania zmeny pozície alebo farby objektu triedy Point. Trieda Screen tu predstavuje observer a implementuje ChangeSubject a ChangeObserver rozhrania. Rozhranie ChangeSubject je Observer tozhranie. Trieda Point predstavuje subjekt a implementuje rovnako rozhranie ChangeSubject. V každej metóde, ktorá má na starosti zmenu polohy alebo farby, je umiestnené volanie metódy určenej na notifikáciu observerov.

Dátová štruktúra Hashset má na starosti uchovávanie konkrétnych observerov.

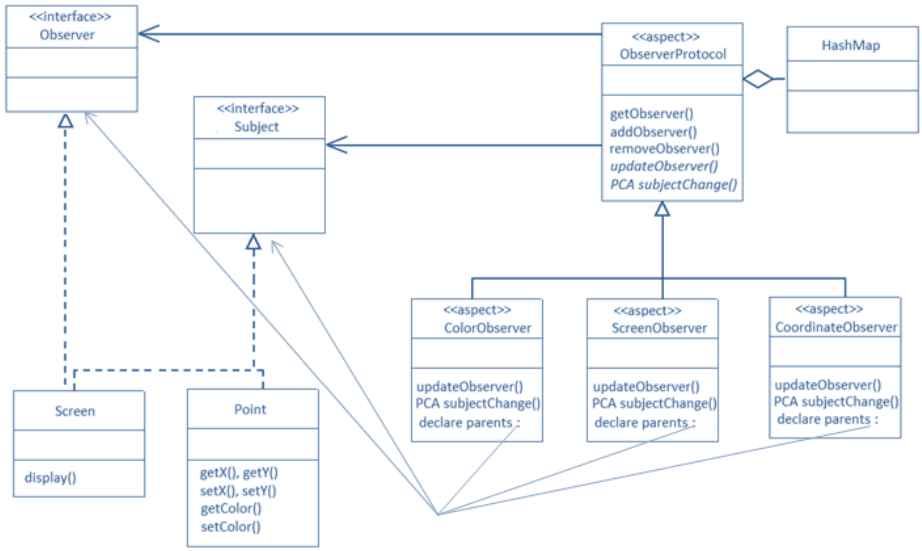


Obrázok č. 45 OOP implementácia návrhového vzoru Observer

V uvedenom príklade obsahuje aj trieda Screen prvky subjektu, ktoré nadobudla implementovaním rozhrania ChangeSubject, čo znamená, že môže pracovať s observermi a sledovať tak zmeny (ktoré sa v jej objektoch odohrávajú).

AOP impl.

Všetka funkcionalita potrebná pre upovedomovanie observerov bola vyňatá z tried Point a Screen a bola umiestnená do aspektov: ColourObserver, ScreenObserver a CoordinateObserver, ktoré všetky dedia od abstraktného aspektu ObserverProtocol a deklarujú rozhrania Subject pre triedu Point a Observer pre triedu Screen.



Obrázok č. 46 AOP implementácia návrhového vzoru Observer

Aspekt ColorObserver deklaruje bodový prierez pre volanie metódy Point.setColor, obdobne deklaruje ScreenObserver bodový prierez pre Screen.display a CoordinateObserver bodový prierez pre Point.setX a Point.setY. Všetky tieto bodové prierezy sa volajú subjectChange a je nevyhnutné implementovať, pretože bol deklarovaný ako abstract v ObserverProtocol, ktorý je rodičom týchto aspektov, pričom advice je v ňom zahrnuté tiež a nemusí byť deklarované jednotlivo v potomkoch.

```
declare parents: Point  implements Subject;
declare parents: Screen implements Observer;

protected pointcut subjectChange(Subject subject):
          (call(void Point.setX(int)) ||
           call(void Point.setY(int)) ) && target(subject);
```

Obrázok č. 47 Ukážka z aspektu CoordinateObserver [15]

Porovnanie V AOP verzii sa nepridávajú jednotlivé observer objekty prostredníctvom subjektov, ale cez aspekty. Porovnanie v podaní Kiczalesa a Hannmana [15] prinieslo lepšie výsledky pre všetky štyri porovnávané vlastnosti. Čo sa týka kvantitatívnych ohodnotení [14], návrhový vzor Observer bol zaradený do skupín vzorov so zvýšenou separáciou koncernov a lepšími výsledkami pre AOP verziu, čo bolo v zhode s kvalitatívnymi predpokladmi.

Príbuzné vzory Mediator

3.6 Diskusia

Na úvod diskusie môžeme uviesť zoznam ostatných GoF vzorov ako ďalších kandidátov na zahrnutie do pokračovania katalógu.

DRUH VZORU	Názov vzoru	Opis vzoru
Vytvárajúce vzory	Abstract Factory	Definuje metódy na tvorbu príbuzných produktov.
	Builder	Trieda - Builder vytvárajúca časti komplexného produktu a odovzdáva kompletný produkt
	Factory Method	Poskytnutie metódy, ktorá má byť prekrytá za účelom tvorby objektov rôznych druhov
	Prototype	**Vytváranie kópií objektu-prototypu**
	Singleton	**Vytvorenie maximálne jednej kópie objektu.**
Štrukturálne vzory	Adapter	Použitie triedy v kontexte, ktorý vyžaduje rôzne rozhrania.
	Bridge	Umožňuje, aby abstrakcia a jej implementácia mali oddelené hierarchie dedenia
	Composite	**Kombinácia niekoľkých objektov do jedného, ktorý má rovnaké správanie ako tieto objekty**
	Decorator	Spôsob pridania funkcionality triede bez zmeny jej rozhrania
	Facade	Spôsob ako zjednodušiť prístup k systému (pozostávajúceho z viacerých tried) prostredníctvom jednej triedy, ktorá tento prístup sprostredkuje
	Flyweight	**Použitie zdieľania v systémoch s veľkým počtom objektov**
	Proxy	**Riadenie prístupu k objektu prostredníctvom iného objektu – zástupcu.**
Behaviorálne vzory	Chain of Respon.	Priraďuje požiadavku prvému objektu v rade, ktorý je schopný ju vykonať, a ten ju buď vykoná alebo ju posunie ďalšiemu objektu v rade
	Command	Implementácia požiadaviek ako objektov, kedykoľvek požiadavky disponujú stavom a správaním.
	Interpreter	Pomocou hierarchie tried vytvára vlastné interpretačné, gramatické pravidlá.
	Iterator	Spôsob prístupu k elementu agregovaného objektu.
	Mediator	Slúži na zapúzdrenie interakcie medzi objektmi
	Memento	Zachytáva predchádzajúci stav objektu (bez toho, aby došlo k porušeniu jeho zapuzdrenosti) a umožňuje mu tak návrat k tomuto stavu
	Observer	**Udáva spôsob akým môže objekt upovedomiť iné objekty o udalosti, zmene**
	State	Rozlišuje rôzne stavy a každý umiestňuje jednotlivo do objektov
	Strategy	**Poskytnutie zameniteľných variant jedného algoritmu**
	Template Method	Poskytnutie algoritmu pre viacero typov, nezávislého od tohto typu
	Visitor	Doplnenie operácií do štruktúry objektov bez toho, aby bola zmenená

Tabuľka č. 9 Opis GoF návrhových vzorov

V rámci otvorenej diskusie je možné spochybňovať takmer všetko, niektorí autori dokonca pochybujú i o prínose návrhových vzorov do softvérového inžinierstva, no od takýchto úvah radšej upustíme. V kontexte tejto by bolo možné sa zaoberať vhodnosťou uvedených návrhových vzorov z dôvodu, že sa nejedná o implementácie z reálnych

systémov, ale iba o zjednodušené príklady. Ďalej treba prihliadať na to, že návrhové vzory nie je možné izolovať, sú súčasťou systému, ktorý im vytvára prostredie a je s nimi v interakcii.

Čo sa týka aspektovo-orientovanej paradigmy a jej potenciálu stať sa nástupníckou paradigmou za objektovo-orientovanú, rozoberané vzory tomu nenasvedčujú, pretože neprinášajú zjednodušenie implementácie, no k takýmto záverom by bol potrebný oveľa odbornejší výskum.

V práci Zrevidované návrhové vzory od Martina Kuhlemanna [26] je do porovnávania objektovo-orientovaných, aspektovo-orientovaných návrhových vzorov pridaný ešte jeden typ vzorov, a to na vlastnosti-orientované návrhové vzory (angl. feature-oriented), ktoré predstavujú vážnu konkurenciu a nie je vylúčené, že sa čoskoro objaví technológia, ktorá prekoná všetky ostatné.

Záver

Aspektovo-orientovaná paradigma sa na základe výsledkov tejto práce prejavila ako praktický nástroj na modularizáciu pretínajúcich koncernov. Výsledky práce Implementácia návrhových vzorov v Jave a AspectJ [15] sa nepotvrdili všetky a prínos nebol vždy iba pozitívny na strane aspektovo-orientovaných návrhových vzorov v porovnaní s prácou Modularizácia návrhových vzorov pomocou aspektov: kvantitatívna štúdia.

Hoci je táto práca zameraná na zisťovanie rozdielností medzi aspektovo-orientovanou paradigmou a objektovo-orientovanou paradigmou za pomoci návrhových vzorov, je možné z opisov uvedených návrhových vzorov poukázať na zachovanie ich špecifických čŕt, akými sú programovanie voči rozhraniam, ktoré je vďaka medzitypovým deklaráciam rozšírenejšie v aspektovo-orientovaných vzoroch, prítomnosť polymorfizmu a snaha o zapuzdrenie toho, čo sa mení (v AOP obvykle do aspektov).

Nejednotnosť v prístupe autorov v posudzovaní návrhových vzorov a absencia všeobecne uznávaného modelu hodnotenia návrhových vzorov v OOP a AOP verziách spôsobili, že v súčasnosti sa javia návrhové vzory skôr ako prostriedok na zlepšenie návrhu softvéru ako na nástroj porovnávania programovacích paradigiem. To však neznižuje ich význam ani potenciál na naplnenie tohto zámeru v budúcnosti.

Zoznam použitej literatúry

[1] KHOSROW-POUR, M. at al. 2009. *Encyclopedia of information science and technology*. 2. vyd. New York: Information Science Reference (IGI Global). 2009. 4292 s. ISBN 978-1-60566-027-1

[2] PARNAS, D. L. 1972. *On the Criteria To Be Used in Decomposing Systems into Modules.* In Communications of the ACM ISSN 0001-0782, 1972, roč. 15, č. 12, s. 1053 – 1058.

[3] DIJKSTRA, E. W. 1974. *On the role of scientific thought* . 2. vyd. New York: Information Science Reference (IGI Global). 2009. 4292 p. ISBN 978-1-60566-027-1

[4] DIJKSTRA, E. W. 1976. *A Discipline of Programming*. 1. vyd. New York: Prentice Hall. 1976. 217 s. ISBN 978-0132158718

[5] FILMAN, R. E. 2005. *A Bibliography of Aspect-Oriented Software Development*. 2. vyd. California: NASA Ames Research Center. 2005. 65 s.

[6] VRANIĆ, V. 2008. *Objektovo-orientované programovanie: Objekty, Java a aspekty*. 1. vyd. Bratislava: STU. 2008. 223 s. ISBN 978-80-227-2830-0

[7] *AspectJ: Frequently Asked Questions*. [online] Dostupné na internete: <http://www.eclipse.org/aspectj/doc/released/faq.php#q:whatisaj>

[8] ALEXANDER, Ch. at al. 1977. *A Pattern Language*. New York: Oxford University Press. 1977. 1171 s. ISBN 0195019199

[9] KICZALES, G. 1997. *Aspect-Oriented Programming*. In European Conference on Object-Oriented Programming. Springer Verlag 1997, roč. 11. 36 s.

[10] BIELIKOVÁ, A. et al. 2009. *Štúdie vybraných tém programových a informačných systémov (4. diel).* Bratislava: STU. 2009. 345 s. ISBN 978-80-227-3139-3

[11] BUSCHMANN, F. at al. 2001. *Pattern-Oriented Software Architecture Volume 1: A System of patterns.* New York: Wiley. 2001. 487 s. ISBN 0 471 95889 7

[12] GAMMA, E. et al. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software.* New York: Addison-Wesley. 1994. 416 s. ISBN-10: 0201633612

[13] BROWN, W. J. et al. 1998. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis.* New York: Wiley. 1998. 336 s. ISBN-10: 0471197130

[14] GARCIA, A. 2005. *Modularizing design patterns with aspects: a quantitative study*. In 4th international conference on Aspect-oriented software development ACM. 2005. 4 roč.

[15] KICZALES, G. HANNEMANN, J. 2002. *Design Pattern Implementation in Java and AspectJ*. In OOPSLA '02 Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming. 2002. 17 roč.

[16] *TIOBE Programming Community Index for April 2012.* [online] Dostupné na internete: <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>

[17] SOMMERVILLE, I. 2011. *Software engineering.* 9 vyd. New York: Wiley. 2011. 773 s. ISBN-10: 0137053460

[18] PAWLAK ,R. – SEINTURIER, L. – RETAILLÉ, J. P. 2005. *Foundations of AOP for J2EE Development.* 1. vyd. New York: Apress. 2005. 353 s.
ISBN: 1-59059-507-6

[19] MILES, R. 2005. *AspectJ cookbook.* Sebastopol: O'Reilly. 1. vyd. 2005. 360 s.
ISBN: 978-0-596-00654-9

[20] LADDAD, R. 2010. AspectJ in action. 2. vyd. 2010. Greenwich**:** Manning Publications. 567 s. ISBN 978-1-933988-05-4

[21] BUSCHMANN, F. at al. 2007. *Pattern-Oriented Software Architecture Volume 5: On Patterns and Pattern Languages.* 1. vyd. 2007. Chichester: Manning Publications. 492 s. ISBN-13: 978-0-471-48648-0

[22] ŠKURLA, M. 2009. *Analýza a realizácia vybraných návrhových vzorov použitím AspectJ*. [online] Dostupné na internete:
<http://ics.upjs.sk/~novotnyr/java/aop/vybrane-navrhove-vzory-aspectj-%28skurla,2009%29.pdf >

[23] BRYTON, S. 2008. *Modularity Improvements with Aspect-Oriented Programming*. Universidade Nova de Lisboa. 2008. Lisabon: Universidade Nova de Lisboa

[24] SIPOS, A. et al. 2008. *On Multiparadigm Software Complexity Metrics*. 1. vyd. 2008. Budapest: Eötvös Loránd University. 15. s.

[25] ELRAD, T. 2001. Discussing Aspects of Aop. In *Communications of the ACM*. 2001, roč. 44, č. 10, s. 39.

[26] KUHLEMANN, M. 2007. *Design Patterns Revisited.* 1. vyd. 2007. Magdeburg: University of Magdeburg, Germany. 41 s.