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

Bc. Ján Kohut

Usmernenia pre použitie aspektovo-orientovaného prístupu v radoch softvérových výrobkov

Diplomová práca

Vedúci diplomovej práce: Ing. Valentino Vranić, PhD.

máj 2009

ANOTÁCIA

Slovenská technická univerzita v Bratislave

FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ

Študijný program: Softvérové inžinierstvo

Autor: Bc. Ján Kohut

Diplomová práca: Usmernenia pre použitie aspektovo-orientovaného prístupu v radoch softvérových výrobkov

Vedúci diplomovej práce: Ing. Valentino Vranić, PhD.

máj 2009

Táto práca sa venuje aspektovo-orientovanému prístupu k radom softvérových výrobkov. Zameriava sa na problém konfigurácie vlastností v radoch softvérových výrobkov. Aspektovo-orientovaný prístup sa javí ako užitočná technika pre konfiguráciu vlastností v radoch softvérových výrobkov. Bolo uskutočnených viacero štúdií venujúcich sa použitiu aspektov v radoch softvérových výrobkov ale nebol vytvorený návod, usmernenia, kedy použiť aspekty a kedy je lepšie použiť iný prístup. Hlavným prínosom tejto práce je návrh metodiky pre tvorbu usmernení pre použitie aspektov v radoch softvérových výrobkov a identifikovanie niektorých usmernení. Pre potreby evaluácie usmernení bola vytvorená štúdia, zhrnutie ktorej je prezentované v tejto práci. Výsledky tejto práce môžu byť použité a nápomocné na podporu vývoja radov softvérových výrobkov pomocou aspektovo-orientovaného prístupu.

ANNOTATION

Slovak University of Technology Bratislava

FACULTY OF INFORMATICS AND INFORMATION TECHNOLOGIES

Degree Course: Software engineering

Author: Bc. Ján Kohut

Thesis: Guidelines for Using Aspects in Software Product Lines

Supervisor: Ing. Valentino Vranić, PhD.

2009, May

The objective of this thesis is to contribute an aspect-oriented entry to software product lines. This thesis is focused on the problem of configuration of features in software product lines. Aspect-oriented programming is showing as a very useful approach to configure features. Many studies have already been devoted to the use aspects in software product line, but there has not been guideline, when to use aspect and when it is better use another approach. The main contribution of this thesis is the proposal of method for creating guideline for use of aspects in software product lines and clarification of some usage rules. A case study was performed to investigate whether identified guidelines are useful and results of this study are presented here. Results of the submitted thesis can be applied in development process of software product line using aspect oriented approach.

Týmto čestne prehlasujem, že som diplomovú prácu vypracoval samostatne, s použitím uvedenej literatúry.

V Bratislave 11.5.2008

Ďakujem vedúcemu mojej diplomovej práce Valentinovi Vranićovi za jeho vedenie a cenné myšlienky.

Obsah

Zoznam obrázkov a tabuliek

1 Úvod

Potreba tvorby softvéru pre určitú skupinu zákazníkov a následné upravovanie konkrétneho výrobku pre konkrétneho zákazníka si vyžiadalo vytvoriť prístup nazvaný Rady softvérových výrobkov (angl. Software Product Line). Rady softvérových výrobkov majú časť funkcionality spoločnú pre viacerých zákazníkov a časť funkcionality je nakonfigurovaná podľa konkrétnych potrieb špecifického zákazníka.

Návrh, vývoj a konfigurácia funkcionality v radoch softvérových výrobkov je netriviálny problém. Často sa používa technika modelovania vlastností, ktorá umožňuje prehľadne zobraziť jednotlivé vlastnosti, ich závislosti a vzťahy medzi nimi. Podľa týchto modelov sa potom vyvíjajú a konfigurujú jednotlivé produkty. Problém s konfiguráciou vzniká najmä v dôsledku variabilných vlastností, ktoré sa môžu, ale aj nemusia uplatniť v konfigurácii jednotlivých produktov, pretože požiadavky na softvér si môžu odporovať a vlastnosti tohto softvéru môžu byť navzájom previazané. Pri tradičnom objektovo-orientovanom prístupe môže vzniknúť roztrúsený kód, v ktorom sa jedna záležitosť opakuje v rôznych moduloch, alebo sú časti tej istej záležitosti roztrúsené vo viacerých moduloch.

Ukazuje sa, že jedným zo spôsobov ako riešiť vyššie spomínané problémy je použitie aspektovo-orientovaného prístupu. Tento prístup zjednodušuje predovšetkým konfigurovanie vlastností v rade softvérových výrobkov. Nie vždy je však použitie aspektovo-orientovaného prístupu najvhodnejším riešením.

Vhodné použitie aspektovo-orientovaného prístupu pri tvorbe radov softvérových výrobkov sa dá vyjadriť vo forme usmernení – táto práca vychádza z tejto tézy. Nachádzajú sa tu usmernenia pre použitie aspektov v radoch softvérových výrobkov. Je tu vytvorená metodika ako vytvárať dané usmernenia a niekoľko usmernení, ktoré sa podarilo identifikovať. Cieľom týchto usmernení je pomôcť návrhárovi a programátorovi sa rozhodnúť, na ktoré oblasti v rade softvérových výrobkov je vhodné použiť aspekty a kedy to nie je vhodné.

Pre potreby evaluácie konkrétneho použitia usmernení bol vyvinutý vlastný rad softvérových výrobkov. V práci je prezentovaný spôsob vývoja daného radu softvérových výrobkov, použité metriky a výsledky, ktoré boli dosiahnuté.

Problematike radu softvérových výrobkov, modelovaniu vlastností a notácií používanej pri modelovaní vlastností sa práca venuje v kapitole 2. Kapitola 3 predstavuje spôsoby prechodu od modelu vlastností k tvorbe kódu. Tvorba usmernení pre použitie aspektov v radoch softvérových výrobkov je opísaná v kapitole č. 4. Evaluácií usmernení sa venuje 5. kapitola. Porovnanie výsledkov s inými prístupmi a prácami je v kapitole 6. Záver práce je umiestnený v kapitole číslo 7.

2 Rady softvérových výrobkov a modelovanie vlastností

V tejto kapitole je predstavený rad softvérových výrobkov, modelovanie vlastností a notácie modelovania vlastností. Z notácii sú stručne predstavené Czarnecki-Eiseneckerová notácia a UML notácia.

2.1 Rady softvérových výrobkov

Rady softvérových výrobkov sú množiny softvérových systémov, ktoré majú časť spoločnej funkcionality a časť funkcionality, ktorá je variabilná. Výhodou spoločnej funkcionality je, že môže byť použitá rozličnými členmi radu softvérových výrobkov (Gomaa, 2005).

Rady softvérových výrobkov sa v súčasnosti javia ako životaschopná a dôležitá paradigma vývoja softvéru. Tieto rady dovoľujú spoločnostiam významne urýchliť vývoj nových výrobkov, znížiť ich cenu, zvýšiť produktivitu, kvalitu a pomáhajú napĺňať obchodné ciele. Urýchľujú vstup výrobkov na trh a poskytujú možnosti pre rozsiahle prispôsobovanie výrobkov potrebám zákazníkov (SEI).

Význam radov softvérových výrobkov vyplýva tiež z toho, že dovoľuje vývojárom dosiahnuť väčší prínos pri opätovnom použití softvéru použitím vhodne navrhnutej softvérovej architektúry namiesto použitia individuálnych komponentov (Gomaa, 2005). Príkladom úspešného používania radu softvérových výrobkov je napríklad firma Nokia. Ročne produkuje približne 30 modelov mobilných telefónov a predáva ich vo viac ako 130 krajinách sveta s rôznymi jazykovými mutáciami. Vo viacerých mobilných telefónoch používa podobný softvér, t.j. jadro softvéru je spoločné a voliteľné funkcie telefónov sú implementované podľa potrieb špecifického výrobku (Clements a Northrop, 2003).

2.2 Modelovanie vlastností

Architektúra radov softvérových výrobkov je architektúra pre rodinu produktov. Táto architektúra by mala opisovať spoločné a variabilné vlastnosti v tejto rodine. Podľa prístupu k vývoju (funkcionálny, objektovo-orientovaný) sa spoločné vlastnosti (angl. common features) opisujú v termínoch ako spoločné moduly, triedy alebo komponenty, a variabilné vlastnosti (angl. variable features) v radoch softvérových výrobkov sú nazývané voliteľnými vlastnosťami (modulmi). Tie zahrňujú alternatívne moduly, triedy, komponenty (Gomaa, 2005).

Modelovanie spoločných a voliteľných vlastností je dôležitou aktivitou v procese vývoja radov softvérových výrobkov. Koncept vlastností je pomerne intuitívny a dá sa aplikovať pre všetky rady výrobkov, nie len na rady softvérových výrobkov. Metodika FODA (angl. Feature Oriented Domain Analysis) organizuje tieto vlastnosti do stromu vlastností (Gomaa, 2005). Vlastnosti môžu byť:

- povinné
- voliteľné
- alternatívne
- alebo-vlastnosti

Kombináciou vyššie vymenovaných vlastností môžu vzniknúť:

- povinné alternatívne / voliteľné alternatívne vlastnosti
- povinné alebo- / voliteľné alebo-vlastnosti (Czarnecki, 1998)

Alternatívna vlastnosť predstavuje výber iba jednej vlastnosti z viacerých ponúknutých vlastností. Príkladom pre takúto vlastnosť je napríklad v doméne mobilných telefónov jazyk, v ktorom je zobrazené menu telefónu. Pri alebo-vlastnostiach si môžeme vybrať z viacerých vlastností, pričom ich môžeme použiť aj viaceré súčasne. Príkladom pre tieto vlastnosti môžu byť doplnkové programy v mobilnom telefóne, ktorých môžeme mať viac súčasne.

2.3 Notácie modelovania vlastností

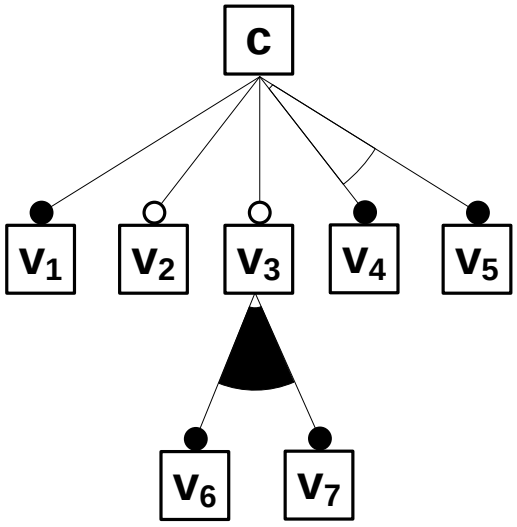
V tejto časti sú predstavené dve notácie používané v modeloch vlastností. Czarnecki-Eiseneckerová notácia sa používa pomerne často pre modelovanie vlastností, UML notácia bola vybraná z dôvodu veľkého rozšírenia UML.

Model vlastností reprezentuje povinné a voliteľné vlastnosti a zároveň závislosti medzi týmito vlastnosťami. Modely pozostávajú z diagramu vlastností a možných prídavných informácií ako napríklad krátky sémantický opis každej vlastnosti, odôvodnenie pre každú vlastnosť, príklad systému s danou vlastnosťou, obmedzenia, pravidlá implicitných závislostí, mód viazania vlastností (dynamické alebo statické viazanie) a priorita (Czarnecki, 1998).

2.3.1 Czarnecki-Eiseneckerová notácia

Czarnecki-Eiseneckerová notácia vychádza z FODA notácie pre diagramy vlastností. Diagram pozostáva z množiny uzlov, množiny hrán a množiny dekorácií hrán. Z týchto uzlov a hrán je vytvorený strom. Dekorácie hrán sú kreslené ako oblúky spájajúce podmnožiny alebo všetky hrany vychádzajúce z daného rodičovského uzla. Dekorácie hrán definujú rozdeľovanie poduzlov daného uzla do disjunktných množín. Koreň diagramu vlastností reprezentuje koncept a nazýva sa tiež konceptový uzol. Ostatné uzly v diagrame reprezentujú vlastnosti a nazývajú sa uzly vlastností (angl. feature nodes).

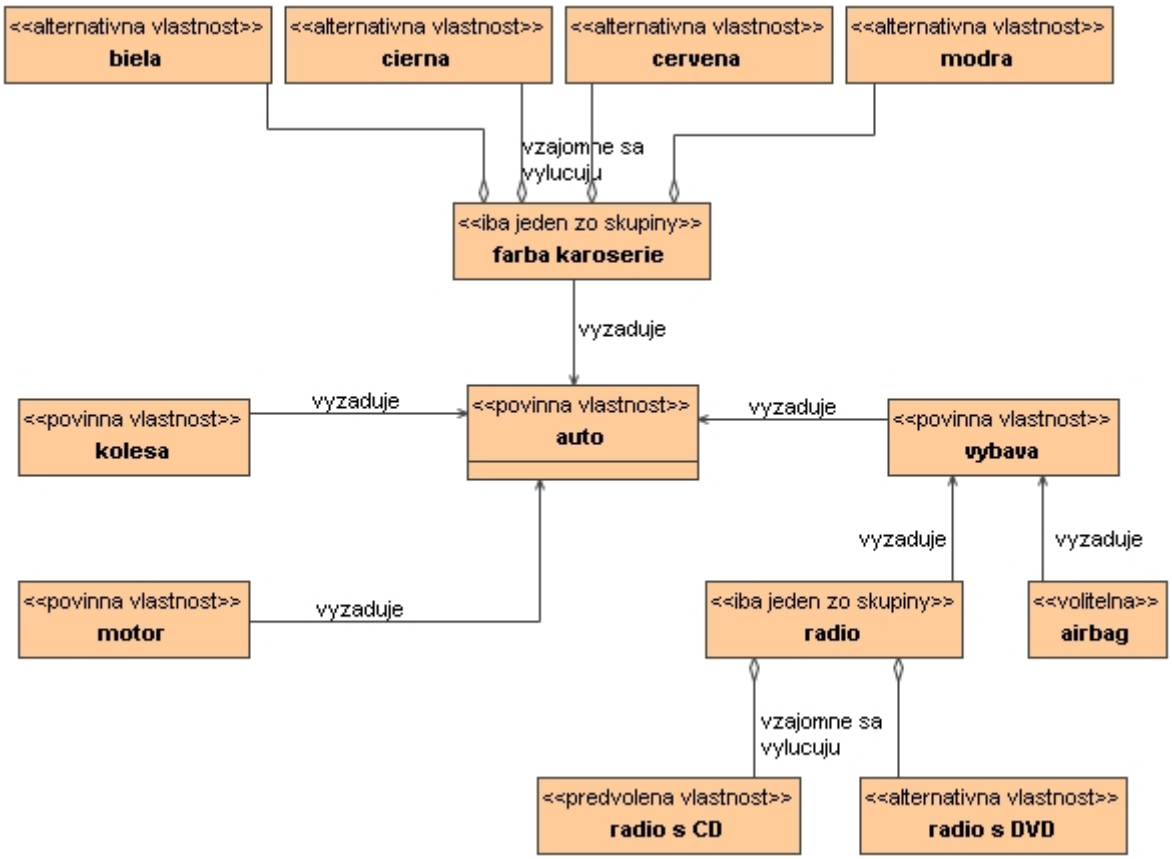
Na obr. 1 je príklad diagramu zakresleného v Czarnecki-Eiseneckerovej notácií. Povinné vlastnosti sú zakreslené jednoduchou hranou (bez dekorácie) ukončenou vyplneným krúžkom (vlastnosť v_1). Voliteľné vlastnosti sú zobrazené jednoduchou hranou ukončenou prázdnym krúžkom (v_2, v_3). Uzly množín alternatívnych vlastností sa znázorňujú hranami prepojenými prázdnym oblúkom (v_4, v_5). Alebo-vlastnosti sú znázornené uzlami, ktorých hrany sú spojené vyplneným oblúkom (v_6, v_7).



Obr. 1: Príklad diagramu v Czarnecki-Eiseneckerovej notácií.

2.3.2 UML notácia

Na obr. 2 je príklad UML notácie použitej pre model vlastností auta. Podobne ako v predchádzajúcej notácii, tak aj v tejto je možné vyjadriť typy vlastností a to pomocou stereotypov a komentárov. Predstavený príklad je len časťou metodológie založenej na UML, ktorá sa nazýva PLUS (Product Line UML-Based Software Engineering), ktorú uviedol Gamaa (2005). Metodológia PLUS približuje modelovacie metódy založené na UML, ktoré sú určené pre jednotlivé systémy, k radom softvérových výrobkov. Okrem modelovania vlastností sa metodológia PLUS používa aj na modelovanie prípadov použitia.



Obr. 2: Model vlastnosti auta v UML notácii.

2.3.3 Porovnanie Czarnecki-Eiseneckerovej a UML notácie

Model vlastností, zakreslený pomocou Czarnecki-Eiseneckerovej notácie, je strom, ktorý sa zvyčajne zakresľuje tak, že koncept je najvyšší uzol tohto stromu a vlastnosti hierarchicky tvoria vetvy tohto stromu. Model vlastností v UML notácia je podobný diagramu tried, ktorý je doplnený stereotypmi. Z môjho subjektívneho pohľadu sú modely vlastností zakreslené v Czarnecki-Eiseneckerovej notácií prehľadnejšie a podľa môjho názoru vhodnejšie na modelovanie vlastností.

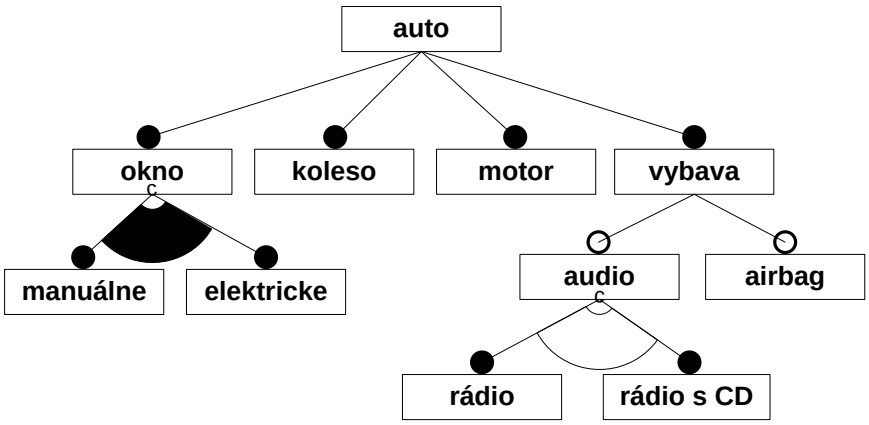
3 Spôsoby prechodu od modelu po tvorbu kódu

Diagram vlastností dovoľuje vyjadriť variabilitu na abstraktnej úrovni. Variabilita špecifikovaná diagramom vlastností je implementovaná v modeloch použitých pri analýze, návrhu, implementácií, využívajúc rôzne mechanizmy pre variabilitu. Napríklad mechanizmy variability pre prípady použitia zahŕňajú parametre, šablóny, rozšírenia vzťahov a vzťahy medzi používateľmi. Mechanizmy variability pre diagramy tried obsahujú dedičnosť, parametrizáciu, dynamické viazanie a kardinalitu (Czarnecki, 1998).

Účelom, pre ktorý sa vytvárajú modely vlastností, je uľahčiť (umožniť) vývoj radov softvérových výrobkov. Ďalším krokom, ktorý nasleduje po fáze modelovania vlastností, je samotná implementácia radov softvérových výrobkov (tvorba kódu). V tejto časti sa pozrieme na dva prístupy ako možno postupovať pri tvorbe kódu z modelu vlastností.

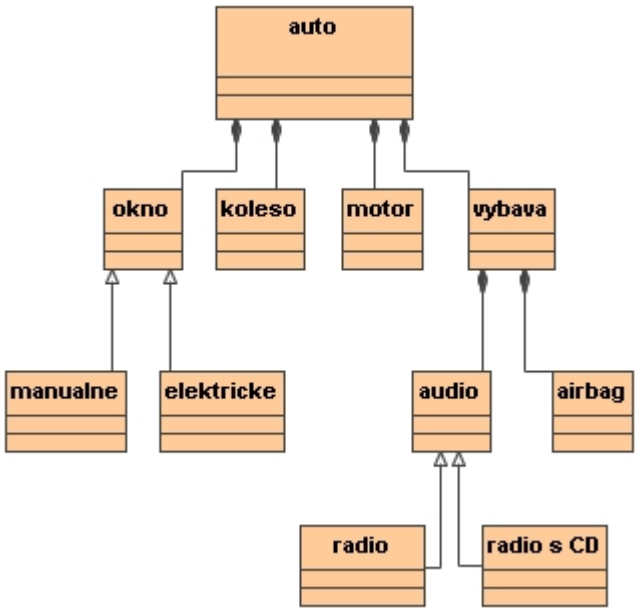
3.1 Objektovo-orientovaný návrh podľa modelu vlastností

Na nasledujúcom príklade (obr. 3) je ilustrovaná variabilita vyjadrená v diagrame vlastností. Príklad ukazuje zjednodušený model vlastností auta. Auto pozostáva z okien, kolies, motora a výbavy. Okná môžu byť manuálne, elektrické alebo obidva druhy okien spolu. Výbava auta je nepovinná a môže obsahovať audio-súpravu a airbag. Audio môže byť buď jednoduché rádio alebo rádio s CD prehrávačom.



Obr. 3: Model vlastností auta.

Na obr. 4 je ukázaná možná implementácia vyššie predstaveného modelu zobrazená ako UML diagram tried. V tomto modeli je zobrazená iba jednoduchá dedičnosť a kompozícia. V nasledujúcich častiach budú opísané dôležité mechanizmy pre variabilitu dostupné v súčasných OO programovacích jazykoch spracované podľa Czarneckého (1998). Ide menovite o mechanizmy: jednoduchá dedičnosť, viacnásobná dedičnosť, parametrizovaná dedičnosť, statická parametrizácia a dynamická parametrizácia.



Obr. 4: Možná implementácia auta zobrazená v UML diagrame tried.

3.1.1 Jednoduchá dedičnosť

Jednoduchá dedičnosť môže byť použitá ako statický, v čase prekladu prítomný mechanizmus pre variabilitu. Je vhodný pre implementáciu statických viazaní, nesimultánnych, jednoduchých bodov variácie.

Vo všeobecnosti, diagram vlastností môže byť implementovaný ako hierarchia jednoduchej dedičnosti bez duplikácie vlastností iba vtedy, ak

- diagram neobsahuje žiadne body variácií, alebo
- jeden bod variácie, ktorý je singulár , alebo
- viac ako jeden bod variácie, kde všetky tieto body sú singulár a nesimultánne.

V prípade, že použitie jednoduchej dedičnosti môže zapríčiniť duplikáciu vlastností, je vhodné zvážiť iné mechanizmy pre variabilitu (Czarnecki, 1998).

3.1.2 Viacnásobná dedičnosť

Body rozšírenia s alebo-vlastnosťami môžu byť implementované v hierarchii viacnásobnej dedičnosti. Táto dedičnosť má oveľa zložitejšie väzby medzi objektmi ako hierarchia jednoduchej dedičnosti. Oveľa flexibilnejším riešením je použitie parametrizovanej dedičnosti (Czarnecki, 1998).

3.1.3 Parametrizovaná dedičnosť

Jazyk C++ dovoľuje parametrizovať rodičovskú triedu danej triedy. Táto vlastnosť sa nazýva parametrizovaná dedičnosť.

3.1.4 Statická a dynamická parametrizácia

Parametrizované triedy (napr. šablóny v C++) sú vhodné pre implementáciu statických väzieb. Ak používame dynamickú metódu viazania objektov, variabilita môže byť reprezentovaná rôznymi triedami v čase behu programu (Czarnecki, 1998).

3.1.5 Implementačné obmedzenia

Model vlastností neobsahuje iba povinné a voliteľné vlastnosti, ale tiež závislosti medzi voliteľnými vlastnosťami. Tieto závislosti sú vyjadrené vo forme obmedzení a štandardných pravidiel pre závislosti. Obmedzenia špecifikujú správne a nesprávne kombinácie vlastností. Štandardné pravidlá pre závislosti ponúkajú prednastavené hodnoty pre nešpecifikované parametre založené na iných parametroch. Obmedzenia a prednastavené hodnoty nám dovoľujú implementovať automatickú konfiguráciu.

Pre konkrétnu implementáciu obmedzení existuje viacero možností. Obmedzenia založené na úrovní súborov je vhodné ovládať prostredníctvom konfiguračného systému. Konfigurácia obmedzení na úrovni tried a objektov sú často súčasťou znovupoužiteľného softvéru. Dynamická konfigurácia môže byť implementovaná napísaním runtime konfiguračného kódu, má podporu vo viacerých aplikačných rámcoch (napr. Spring framework). V prípade statickej konfigurácie vzniká potreba statického metaprogramovania. Statické metaprogramovanie nám dovoľuje napísať metakód, ktorý je vykonaný kompilátorom (Czarnecki, 1998).

3.2 Aspektovo-orientovaný prístup

V súčasnosti je dominantnou programovacou paradigmou objektovo-orientované programovanie (OOP). Základnou ideou tohto prístupu je, že úlohy v softvérovom systéme sú dekompozíciou rozdelené medzi objekty. Hoci je objektová orientácia skvelá myšlienka, má určité obmedzenia. Objektovo-orientované programovanie má problémy s lokalizáciu záležitostí umocňovanú globálnymi obmedzeniami, primeraným oddeľovaním záležitostí a aplikáciou na doménovo-špecifické znalosti (Elrad a kol., 2001).

Aspektovo-orientované programovanie je založené na myšlienke, že softvérové systémy sú lepšie naprogramované, ak sú oddelené rôzne záležitosti systému a niektoré rysy ich vzťahov. Po aplikovaní mechanizmov aspektovo-orientovaného programovania nad podkladovým prostredím sa vtkajú (angl. weave) alebo zostavia tieto záležitosti spolu do koherentného programu (Elrad a kol., 2001).

3.2.1 AspectJ

Jazyk AspectJ je reprezentantom aspektovo-orientovaného programovacieho jazyka a aspekovo-orientovaným rozšírením jazyka Java. Základnou konštrukciou tohto jazyka je aspekt (angl. aspect). Aspekty pridávajú do objektovo-orientovaného jazyka nový koncept, bod spájania (angl. join point) a niekoľko nových jazykových konštrukcií: bodové prierezy (angl. pointcuts), videnia (angl. advices), medzitypové deklarácie (angl. inter-type declarations) a aspekty. Bodové prierezy a videnia dynamicky pôsobia na tok programu, medzitypové deklarácie staticky pôsobia na hierarchiu tried v programe. Aspekty zapuzdrujú tieto nové konštrukcie (AspectJ, 2003).

Bod spájania je dobre definovaný bod v toku programu. Aspekty modifikujú vykonávanie programu v bodoch spájania. Tieto miesta sú určené pomocou bodových prierezov, a modifikácie v nich sú vyjadrené pomocou videní. Videnie sa vykonáva pred, po, alebo namiesto bodu spájania. Aspekty môžu taktiež dopĺňať triedy novými prvkami pomocou medzitypových deklarácií. Tie umožňujú meniť statickú štruktúru programu zmenou

premenných tried, pridávaním atribútov, metód a úpravou vzťahov medzi triedami (AspectJ, 2003).

Vtkanie aspektov sa realizuje priamo pri preklade. Výsledkom je Java bajtkód - preložený program sa vykonáva na Java virtuálnom stroji (angl. Java Virtual Machine).

Ako už bolo spomenuté vyššie, bod spájania je dobre definované miesto vo vykonávaní programu. V týchto bodoch je možné vplývať na vykonávanie programu. Exponované body spájania sú body, ku ktorým sa dá v danom jazyku pristúpiť. Body spájania sú v určitom kontexte (napr. kontext volania metódy predstavuje objekt, ktorý ju vyvolal, cieľový objekt a argumenty). Exponované body spájania v jazyku AspectJ umožňujú:

- volať a vykonávať metódy a konštruktory
- pristupovať k poliam, čítať a zapisovať do nich
- spracovanie výnimiek
- inicializovať triedy a objekty: statická inicializácia, inicializácia a predinicializácia
- vykonanie videnia

K bodom spájania sa nepristupuje jednotlivo, ale prostredníctvom bodových prierezov. Bodové prierezy predstavujú množinu bodov spájania a definujú sa pomocou primitívnych bodových prierezov (AspectJ definuje viac ako 20 primitívnych bodových prierezov). Bodové prierezy je možné skladať pomocou logických operácií *a* (&&), *alebo* (||) a *negácie* (*!*) bodového prierezu. Príkladom bodového prierezu môže byť zachytenie volaní metód. V príklade je bodový prierez na zachytenie všetkých volaní metód s názvom *mojaMetoda()* a jedným argumentom typu *int*.

```
call(void mojaMetoda(int))
```

Videnia definujú aktivity, ktoré sa vykonávajú v spojitosti so zahrnutými bodmi spájania. Zoznam argumentov videnia slúži na prenos kontextu. Jestvujú tri typy videní:

- before – pred bodmi spájania
- after – po bodoch spájania
- around - s úplným ovládaním vykonávania bodov spájania

Cez kontext bodu spájania je možné sprístupniť videnie. Na to slúžia bodové prierezy:

- args() – sprístupnenie argumentov metódy
- this() – sprístupnenie vykonávaného objektu
- target() – sprístupnenie cieľového objektu

3.2.2 Homogénne a heterogénne pretínajúce záležitosti

V literatúre, pozri (Apel a Batory, 2006), venujúcej sa aspektovo-orientovanému prístupu boli identifikované dva rozličné druhy pretínajúcich sa záležitostí:

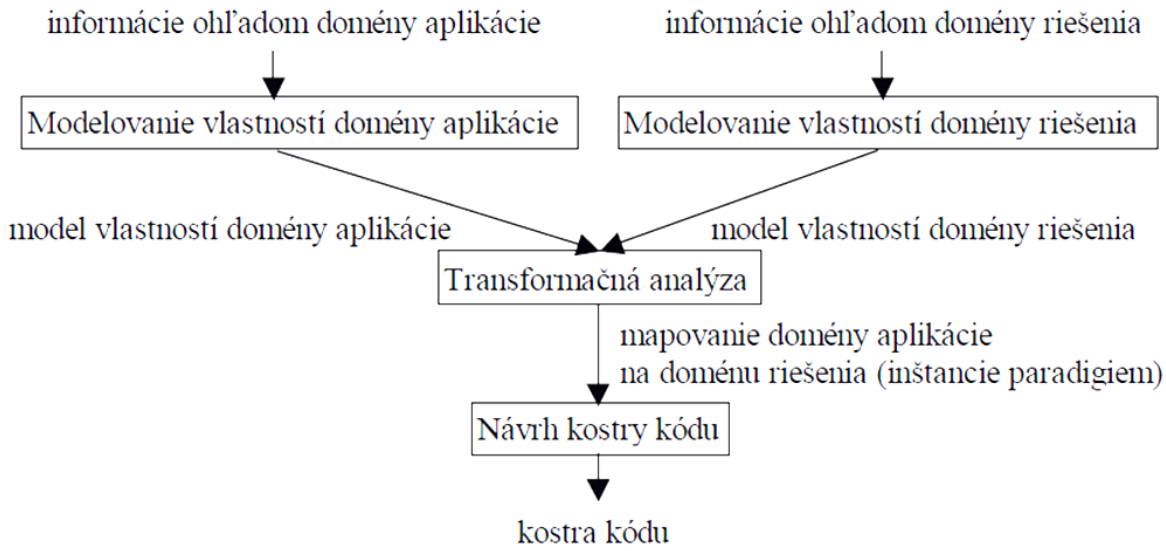
- Homogénne pretínajúce sa záležitosti
- Heterogénne pretínajúce sa záležitosti

Homogénne pretínajúce sa záležitosti majú pre viaceré body spájania rovnaké videnie. Heterogénne pretínajúce sa záležitosti majú naopak pre rozličné body spájania rozličné videnie.

3.3 Multiparadigmový návrh s modelovaním vlastností

Vranić vo svojich prácach (2004, 2005) predstavuje multiparadigmový návrh s modelovaním vlastností (MPD_{FM}, pozri obr. 5). Táto práca používa Czarnecki-Eisenckerovú notáciu na modely vlastností. Hlavným výstupom MPD_{FM} je kostra kódu. Vytvorené modely je možné neskôr opätovne využiť pri ďalších transformačných analýzach. V transformačnej analýze Vranić navrhuje zobrazenie uzlov v modeli vlastností na štruktúru jazyka (autor si vybral jazyk AspectJ). Štrukturálne paradigmy tohto jazyka sú: trieda, rozhranie, metóda a aspekt. Vzťahové paradigmy jazyka AspectJ sú: dedenie, preťaženie, medzitypové deklarácie, videnie a prierez.

Vranić definuje ako kľúčovú činnosť v multiparadigmovom návrhu transformačnú analýzu, proces hľadania korešpondencie a mapovania medzi konceptmi domény aplikácie a riešenia: "*inštanciácia paradigiem zdola nahor nad konceptmi domény aplikácie v dobe tvorby zdrojového kódu"* (Vranić, 2004). Výsledkom úspešnej transformačnej analýzy je jedno z možných riešení, keďže je to tvorivý proces. Návrh kostry kódu sa uskutočňuje prechádzaním stromami inštancií paradigiem. Najprv sa transformujú štrukturálne paradigmy a až potom vzťahové paradigmy, ktoré na nich stavajú (Vranić, 2004).



Obr. 5: Multiparadigmový návrh s modelovaním vlastností (Vranić, 2004).

3.4 Porovnanie prístupov

Czarnecki (1998) navrhol notáciu a definoval pravidlá pre tvorbu modelov vlastností. Neskôr pre vytvorené modely vlastností definoval spôsoby, ako možno z modelu vlastností vytvoriť zdrojový kód, najmä pre objektovo-orientovaný prístup.

Vranić (2004) používa notáciu zavedenú Czarneckým. Rozširuje ju napríklad o referencie konceptov, ktoré uľahčujú prácu so zložitými modelmi vlastností, parametrizáciu. Multiparadigmovým návrhom modelovania vlastností autor predstavil spôsob zobrazenia

uzlov a vzťahov z modelu vlastností na paradigmy jazyka. Autor si vybral jazyk AspectJ. Výsledkom tejto činnosti bol model jazyka.

3.5 Kombinácia modelovania vlastností a aspektovo-orientovaného prístupu

V predchádzajúcej časti je opísaný spôsob zobrazenia vlastností na aspektovo-orientovaný prístup. V aspektovo-orientovanom programovaní aspekty modifikujú vykonávanie programu staticky alebo dynamicky. Ak aspekty implementujú vlastnosti, ktoré sú nezávislé navzájom, problém nevzniká. Ak to tak nie je, variácie môžu mať dopad na ostatné časti softvéru (Lee a kol., 2006).

3.5.1 Vplyv spoločných a variabilných vlastností na návrh softvéru

Používaním aspektovo-orientovaného prístupu môžeme jasne oddeliť spoločné vlastnosti od variabilných. Spoločné vlastnosti je vhodné dávať vo fáze návrhu do modulárnych konceptov. Variabilné vlastnosti definovať ako aspektové komponenty (angl. aspectual component), ktoré upravujú alebo rozširujú základné modulárne komponenty obsahujúce spoločné vlastnosti (Lee a kol., 2006).

Spoločné vlastnosti je možné tiež definovať ako aspektové komponenty, ak majú pretínajúce sa záležitosti. Tie môžu byť homogénne (napr. logovanie), ktoré pridávajú rovnaký fragment kódu ako rozličný bod spájania viacerých komponentom, alebo heterogénne pretínajúce sa záležitosti (napr. služby), ktoré pridávajú rozličný kód. Spoločné vlastností implementované pomocou homogénnych pretínajúcich záležitostí môžu byť elegantne vyriešené ako oddelené aspekty pomocou aspektovo-orientovaného prístupu. Ak sú heterogénneho typu, môžu byť včlenené do existujúcich modulárnych konceptov, alebo definované ako oddelené aspektové komponenty. Podobne aj variabilné vlastnosti nemusia byť definované iba ako aspektové komponenty (Lee a kol., 2006).

3.5.2 Závislosti medzi vlastnosťami

Závislosti medzi vlastnosťami majú významný vplyv na vývoj radov softvérových výrobkov. Ak sa voliteľná vlastnosť pridá alebo odoberie pre niektorý produkt v radoch softvérových výrobkov, môže to ovplyvniť funkčnosť tých častí výrobku, ktoré boli na danej vlastnosti závislé. Tieto zmeny nastávajú najmä z dôvodu vzájomnej previazanosti jednotlivých variabilných vlastností. Použitím aspektovo-orientovaného prístupu môžeme riešiť tento problém skrývaním (angl. encapsulate) závislostí do implementácie aspektov (Lee a kol., 2006).

3.5.3 Spájanie závislosti podľa času

Čas, v ktorom sú variabilné vlastnosti viazané do produktu (angl. binding time), má vplyv na implementáciu týchto radov softvérových výrobkov. Jazyk AspectJ poskytuje podporu pre viazanie počas kompilácie (angl. compile time) i počas behu aplikácie (angl. run time) (Lee a kol., 2006). Pre iné programovacie paradigmy môže viazanie závislosti nastať v čase písania kódu (angl. source time), kompilácie, linkovania (angl. link time) alebo počas behu aplikácie.

3.6 Zhrnutie

Pre rady softvérových výrobkov, v ktorých používame aspektovo-orientovaný prístup, vývoj začína základnými modulárnymi komponentmi, ktoré zahŕňajú spoločné vlastnosti, a oddelením všetkých pretínajúcich sa záležitosti v aspektoch. Pre každú variabilnú vlastnosť sa musí vykonať analýza, či nepretína moduly vo viacerých komponentoch, aké sú závislosti medzi nimi a aký je čas viazania vlastností (Lee a kol., 2006).

4 Usmernenie pre použitie aspektov v radoch softvérových výrobkov

V tejto kapitole je predstavená metodika na tvorbu usmernení, definovaná forma usmernenia a je tu identifikovaných päť usmernení.

Vhodné použitie aspektovo-orientovaného prístupu pri tvorbe radov softvérových výrobkov sa dá vyjadriť vo forme usmernení (pravidiel).

Aspektovo-orientovaný prístup prináša určité výhody, ale i obmedzenia pri jeho použití vo vývoji softvéru. Je všeobecne známe, na aké typy problémov sa hodí aspektovo-orientované programovanie. Stále však prevláda určitý konzervatívny prístup k tejto programovej paradigme. Vývoj radov softvérových výrobkov nesie so sebou množinu viacerých komplexných problémov. Ako jedna z možností na riešenie určitej množiny týchto problémov sa javí aspektovo-orientovaný prístup.

Bolo už uskutočnených viacero štúdií na integráciu aspektovo-orientovaného prístupu do vývoja radu softvérových výrobkov. Autori týchto štúdií popísali niektoré pozitíva, ale aj negatíva, ktoré prinieslo aspektovo-orientované programovanie v danom rade softvérových výrobkov. Z týchto pozitív aj negatív sa dajú odvodiť odporúčania, ktoré môžu napomôcť, kedy je vhodné použiť aspektovo-orientovaný prístup pri tvorbe radov softvérových výrobkov, a kedy je vhodnejšie použiť iné prostriedky. Tieto odporúčania[1] sú v ďalšom texte.

Cieľom týchto usmernení má byť poskytnutie návrhárovi, programátorovi návodu, ako rozumne vytvoriť rad softvérových výrobkov s použitím aspektovo-orientovaného prístupu.

4.1 Forma usmernenia

V tejto časti bude definovaná forma usmernení, ktorú by sme mohli pomenovať ako návod. Existuje viacero prístupov k zápisu usmernení, pravidiel a vzorov, ktoré nemajú ustálenú formu. Mnohí autori uprednostňujú vlastné formy, pozri (Fowler, 2006). V tejto práci je definovaná forma zápisu usmernení:

- Kontext
- Problém
- Dôvody
- Riešenie
- Príklad
- Diskusia

Kontext predstavuje opis situácie, kedy je vhodné uvažovať nad aplikáciou usmernenia. *Problém* definuje problémy, ktoré môžu nastať, ak sa dané usmernenie neaplikuje. V časti *Dôvody* sú rozoberané dôvody, prekážky a ciele, prečo súčasný stav nie je uspokojujúci. *Riešenie* predstavuje spôsob ako vyriešiť situáciu. *Príklad* obsahuje buď implementovanú

[1] Anglický ekvivalent by bol guideline

alebo naznačenú konkrétnu implementáciu daného riešenia. V *Diskusii* sa môžu rozoberať viaceré aspekty, ktoré môže priniesť implementácia usmernenia.

Usmernenia je možné definovať na viacerých úrovniach abstrakcie nazerania na systém. Nie je presne definovaná úroveň abstrakcie z dôvodu, aby bolo možno pokryť čo možno najširší záber oblastí, ktorým sa dané usmernenia budú venovať. Zámerom usmernení je poskytnúť programátori, ktorý si ich naštuduje, návod a pomoc ako rozumnejšie vyvíjať rad softvérových výrobkov s prihliadnutím na aspektovo-orientovanú paradigmu.

4.1.1 Iné formy zápisov usmernení

Viacerí autori používajú pri zápise usmernení časť nazývanú *Problém* vo väčšom rozsahu, pozri napr. (Coplien, 2007). Základným a prvotným problémom, ktorý pomáha riešiť aspektovo-orientované programovanie, sú pretínajúce sa záležitosti. Podľa môjho názoru je preto zbytočné uvádzať do každého usmernenia problémy, ktoré by sa opakovali vo viacerých usmerneniach. Ďalším aspektom je, že tvorba radov softvérových výrobkov bez aspektovo-orientovaného prístupu nie je neprekonateľný problém. Definované usmernenia sú skôr odporúčania ako robiť niektoré veci lepšie. V tomto prípade, či a kedy použiť aspektovo-orientovaný prístup, v akej forme a aké to bude mať následky.

4.1.2 Názov usmernenia

Pre názov usmernenia je vhodný akčný tvar, ktorý zjednodušene povie, čo je riešením problematiky v usmernení. Základná schéma: ak ... potom urob ..., resp. podobným štýlom.

4.1.3 Aplikácia usmernení

Usmernenia popisujú okolnosti, za akých je vhodné uvažovať nad ich aplikáciou v danom kontexte. Usmernenia majú skôr odporúčací charakter ako príkazový, pretože oblasť, v ktorej sú definované, je pomerné zložitá.

4.1.4 Závislosti medzi usmerneniami

Pri definovaní usmernení je potrebné zachovať konzistentnosť s ostatnými usmerneniami tak, aby sa jednotlivé usmernenia nevylučovali. Hoci sa môže stať, že niektoré na prvý pohľad budú stať voči sebe, z kontextu, riešenia a diskusie by mal vyplývať spôsob ako skombinovať dané usmernenia.

4.1.5 Obmedzenia

Usmernenia je možné definovať na viacerých úrovniach abstrakcie: architektúra systému, konfigurácia vlastností a implementačné detaily v kóde.

Okrem kontextu samotných usmernení treba uvažovať aj akým spôsobom je daný rad softvérových výrobkov vyvíjaný (inkrementálne, z už existujúceho systému, ...), aké ma daná softvérová firma možnosti a skúsenosti s aspektovo-orientovaným vývojom a nakoľko je ochotná ísť do rizika vyvíjať softvér iným prístupom.

Samotnými usmerneniami nie je možné pokryť všetky aspekty, s ktorými sa vývojár stretne počas vývoja radu softvérových výrobkov. Naopak, pravidlá dokážu pokryť iba malú časť jeho otázok, problémov. V oblastiach, na ktoré aspektovo-orientované programovanie nemá v konečnom dôsledku pozitívny vplyv je vhodnejšie použiť iný prístup.

Definícia formy usmernení, ktorá je v tejto práci definovaná, nemusí spĺňať všetky požiadavky, ktoré pri špecifikovaní nového usmernenia môže vyžadovať jeho tvorca. Zastávam názor, podobne ako M. Fowler (2006): „Je viac cenné, ak máme množinu dobrých pravidiel, síce slabo organizovaných, ako keď máme skutočne dobrú štruktúru pravidiel s nedostatočnými a slabými vzormi.“[2]

V nasledujúcich častiach sú uvedené usmernenia, ktoré sa podarilo identifikovať. Nie je to konečná množina a verím, že sa bude rozširovať.

4.2 Rady softvérových výrobkov nevyvíjať aspektovo-orientovanou refaktorizáciou

Nie je vhodné vyvíjať rady softvérových výrobkov refaktorizáciou, pretože je problematické konfigurovať vlastnosti pomocou aspektov v systéme, ktorý nebol na to navrhnutý.

4.2.1 Kontext

Vývoj nového radu softvérových výrobkov umožňuje firme si zvoliť spôsob vývoja daného radu. Okrem spôsobu vývoja, architektúry a ďalších záležitostí, je na výber aj programovacia paradigma, ako sa budú riešiť niektoré problémy. Jednou z možností je aj aspektovo-orientovaný prístup.

4.2.2 Problém

Refaktorizácia už vyvinutých systémov s použitím aspektovo-orientovaného prístupu prináša značné problémy a nevýhody, klesá čitateľnosť a udržovateľnosť kódu (Kästner a kol., 2007).

4.2.3 Dôvody

Aplikovanie aspektovo-orientovaného prístupu na vyvíjaný systém v širšom meradle, kladie určíte požiadavky na spôsob vývoja daného systému. Vzhľadom na architektúru systému, môže aplikácia aspektovo-orientovaného prístupu priniesť problémy. Použitie aspektov izolovane v implementácií kolaborujúcich tried nemusí reflektovať objektovo-orientovanú štruktúru, ktorú má návrhár architektúry radu softvérového výrobku na mysli pri návrhu. Napríklad zapuzdrenie rôznych vlastností a ich spolupráce do jedného aspektu môže skryť nám a ostatným rozpoznanie a porozumenie vnútornej objektovo-orientovanej štruktúre a významu týchto vlastností (Apel, 2007).

4.2.4 Riešenie

Nie je vhodné použiť refaktorizáciu vlastností pomocou aspektov na systém, ktorý nebol vyvíjaný s ohľadom na konfiguráciu vlastností. Možnosťou riešenia je buď použiť iný prístup ako aspektový, alebo vyvíjať rad softvérových výrobkov postupne a so zreteľom na to, že chceme konfigurovať vlastnosti.

[2] Remember that in the end it's more valuable to have a bunch of good patterns, poorly organized than it to have a really good structure with weak patterns underneath them.

4.2.5 Príklad

Ch. Kästner v práci (Kästner a kol., 2007) opisuje refaktorizáciu Oracle Berkeley DB JE[3] systému na rad softvérových výrobkov. Daný systém refaktorovali do 38 vlastností, ktoré implementovali prostredníctvom aspektov. Táto refaktorizácia priniesla so sebou problémy najmä z pohľadu vývoja a zdrojového kódu. Aspekty bolo problematické udržiavať. Značná previazanosť kódu spôsobovala, že nebolo možné urobiť lokálnu zmenu kódu bez porozumenia všetkých aspektov, alebo bez nástroja, ktorý by poskytoval podporu pri rozšíreniach určitých kusov kódu.

Zvyčajný spôsob ako použiť aspekty pri refaktorizácií takéhoto systému môže byť vytvorenie viacerých abstraktných tried a použitie aspektového vzoru Cuckoo`s Eggs. Týmto sa však stráca znalosť objektovo-orientovaného návrhu a kódu a je problematické sledovať a pochopiť, čo vykonáva kód.

4.2.6 Diskusia

Inkrementálny vývoj radu softvérových výrobkov vzhľadom na to, že sa bude používať aspektovo-orientovaný prístup, môže priniesť za splnenia ďalších okolností určitý pozitívny prínos. Naproti tomu, použitie aspektov na konfiguráciu už vytvoreného radu softvérových výrobkov môže priniesť viac negatív ako pozitív.

Na otázku, či vyvíjať softvér inkrementálne alebo refaktorizáciou už existujúceho systému, nie je ľahké odpovedať. Je tu potrebné zohľadniť viacero faktorov. Ako bolo ukázané v príklade refaktorizácia softvéru, ktorý pôvodne nebol vyvinutý na to, aby sa v ňom jednotlivé vlastnosti dali konfigurovať, nie je najvhodnejším miestom na uplatnenie aspektovo-orientovaného prístupu. Už samotná refaktorizácia takéhoto systému, nech by bola vykonaná akýmikoľvek vývojovými prostriedkami, je pomerne náročná.

4.3 Na povinné vlastnosti v rade softvérových výrobkov bez pretínajúcich záležitostí nepoužívať aspekty

Na jadro aplikácie, ktoré je vo všetkých inštanciách radu softvérových výrobkov a je bez pretínajúcich záležitostí, nie je vhodné používať aspekty.

4.3.1 Kontext

V rade softvérových výrobkov je určitá časť vlastností daného radu povinná, nachádza sa v každej inštancii daného radu a zvyčajne tvorí jadro funkcionality.

4.3.2 Problém

Aspekty prinášajú zneprehľadnenie objektovo-orientovanej štruktúry programy.

4.3.3 Dôvody

Použitie aspektov je vhodným riešením pretínajúcich sa záležitostí v softvéri.

Architekti i vývojári softvérových systémov v súčasnosti preferujú objektovo-orientovaný prístup. Návrh softvérových systémov a radu softvérových výrobkov je zvyčajne robený ako objektovo-orientovaný. Použitie aspektovo-orientovaného prístupu na povinné vlastnosti daného radu, ktoré nemajú pretínajúce záležitosti, prináša značnú zložitosť pre

[3] http://oracle.com/technology/products/berkeley-db

pochopenie funkcionality, nečitateľnosť a pomerne výrazne ovplyvňuje implementáciu, keďže architektúra systému bola pôvodne navrhnutá ako objektovo-orientovaná (Kästner a kol., 2007).

4.3.4 Riešenie

Na povinné vlastnosti nie je vhodné použiť aspektovo-orientovaný prístup, ak je splnená väčšina nasledujúcich podmienok:

- sú to povinné vlastnosti a tvoria stálu súčasť aplikácie a radu softvérových výrobkov
- vlastnosti sú vhodne modularizované v triedach a komponentoch
- je možnosť priamo zasahovať do zdrojových kódov radu softvérových výrobkov a ich meniť, t.j. nie sme nútení používať aspektovo-orientovaný prístup iba ako nadstavbu nad existujúcim systémom, ktorého funkcionalitu chceme zmeniť, resp. rozšíriť
- ak dané záležitosti nepretínajú iné záležitosti

potom použitie aspektovo-orientovaného prístupu prináša zneprehľadnenie zdrojových kódov.

Zapúzdrenie rozličných tried a vzťahov medzi nimi do aspektov môže brániť programátorovej schopnosti rozoznať a porozumieť skrytej objektovo-orientovanej štruktúre a významom vlastností radu softvérových výrobkov. Zvlášť, ak spolupracuje veľa tried, ktoré sú zlúčené v jednom alebo viacerých aspektoch, výsledný kód môže byť náročný na čítanie a pochopenie. Príkladom nevhodného použitia aspektovo-orientovaného prístupu by bolo napríklad použitie medzitypových deklarácií.

Riešením pre úpravu povinných vlastností, pre ktoré neplatí väčšina vyššie spomenutých podmienok, môže byť klasický objektovo-orientovaný kód, alebo ďalšie prístupy ako napríklad Feature-Oriented Programming (Apel a Batory, 2006).

Úprava pretínajúcich záležitostí v povinných vlastnostiach, ktoré nesúvisia priamo s konfiguráciou vlastností a sú riešené na úrovni celého systému ako napríklad autorizácia, logovanie, perzistencia a iné, je vhodné riešiť aspektmi.

4.3.5 Príklad

Príkladom, kedy nie je vhodné použiť aspekty, je vhodne objektovo-orientovane navrhnutá a implementovaná aplikácia. V prípade radu softvérových výrobkov sú to povinné vlastnosti, ktoré tvoria jadro aplikácie. Tieto vlastnosti by mali byť bez pretínajúcich záležitostí, aj bez autorizácie, logovania, zabezpečenia perzistencie a iných pretínajúcich záležitostí.

4.3.6 Diskusia

V tomto usmernení sa opäť vynára problematika objektovo-orientovaný návrh a implementácia verzus aspektovo-orientovaný prístup. Ak si vystačíme iba s objektovo-orientovaným prístupom, potom by použitie aspektov nemuselo byť najefektívnejším riešením. Kde tradičný (objektovo-orientovaný) prístup zaostáva, môžu byť aspekty dobrou voľbou.

4.4 Pre redukciu replikovaného kódu je vhodné použiť aspekty na homogénne pretínajúce záležitosti[4]

Na homogénne pretínajúce sa záležitosti je vhodné použiť aspekty, ktoré redukujú replikovaný kód.

4.4.1 Kontext

Mnohokrát sa v rade softvérových výrobkov opakuje jedna záležitosť vo viacerých moduloch. Príkladom takejto pretínajúcej záležitosti môže byť napríklad logovanie.

4.4.2 Problém

V homogénnych pretínajúcich sa záležitostiach sa často opakuje rovnaký kód.

4.4.3 Dôvody

Našou snahou je odstrániť homogénne pretínajúce záležitosti z radu softvérových výrobkov. Ďalším cieľom je odstránenie opakujúcich sa riadkov kódu a jeho sprehľadnenie, umiestnenie funkcionality, ktorá sa opakuje vo viacerých moduloch na jedno miesto. Týmto môžeme dostať vlastnosť, ktorú je možné konfigurovať v rade.

4.4.4 Riešenie

Homogénne pretínajúce záležitosti majú pre viaceré body spájania rovnaké videnie. To nám umožňuje pre časť kódu, ktorý sa vykoná (videnie), definovať viacero bodov spájania, ktorými môžeme zachytiť časť funkcionality.

4.4.5 Príklad

Príkladom pre homogénne pretínajúce záležitosti môže byť logovanie, trasovanie, odchytávanie výnimiek, sledovanie výkonnosti aplikácie. Na zdrojovom kóde môžeme vidieť ukážku homogénnych pretínajúcich záležitostí.

```
public aspect Homogenous {
      pointcut accessAutorization(): call ...;
      pointcut accessDemilitaryZone(): call ...;
      pointcut accessCommunicationInterfaces(): call ...;
      pointcut accessApplication(): call ...;

      before ():accessAutorization() ||
                  accessDemilitaryZone() ||
                  accessCommunicationInterfaces() ||
                  accessApplication() {
            System.out.println("Access to ...");
      }
}
```

[4] Homogénne pretínajúce sa záležitosti majú pre viaceré body spájania rovnaké videnie. Heterogénne pretínajúce sa záležitosti majú naopak pre rozličné body spájania rozličné videnia. Zjednodušene rozdiel z hľadiska implementácie je najmä v tom, že viaceré homogénne pretínajúce zachytíme jedným bodovým prierezom, heterogénne pretínajúce záležitosti potrebujú pre každú záležitosť vlastný bod spájania.

4.4.6 Diskusia

Homogénne pretínajúce záležitosti majú pre viaceré body spájania rovnaké videnie. Čiže pre rôzne metódy, na ktorých zachytenie môžeme použiť aj viaceré bodové prierezy, stačí vytvoriť jedno videnie. Ak by sme chceli daný problém riešiť iným spôsobom ako aspektmi, dané riešenie by nemuselo byť tak elegantné. Iné prístupy by vyžadovali definovať pre každú metódu určitý kus kódu, pričom ten kód by bol v mnohom identický. To by viedlo k replikácií kódu a s tým spojených problémov (Apel a Batory, 2006).

Niektoré homogénne pretínajúce záležitosti alternatívne môžeme modularizovať zavedením abstraktnej triedy, ktorá zapuzdruje základnú funkcionalitu. I keď to funguje napríklad pre všetky správy alebo obsluhu správ, nefunguje to pre triedy, ktoré sú úplne nesúvisiace, napríklad logovanie. Je na programátorovom rozhodnutí, čí dané triedy sú syntakticky a sémanticky dosť blízko, aby boli združené základnou abstraktnou triedou (Apel a Batory, 2006).

Aspekty môžu ovplyvňovať kolaboráciu[5] tried medzitypovými deklaráciami a videniami. To môže viesť, ako už bolo spomenuté viackrát, k nepochopeniu problému v štruktúre tried daného programu. Ďalšou vecou, ktorú treba zvážiť, je použitie aspektov na vlastnosť v pomerne veľkom meradle, t.j. vlastnosť zloženú z veľa tried, medzi ktorými je kolaborácia. Zlučovanie všetkých participujúcich tried do jedného alebo viacerých aspektov ničí objektovo-orientovanú štruktúru spolupráce tried a zakrýva ju pred programátorom, dôsledkom je program zložitý na pochopenie (Apel a Batory, 2006).

4.5 Na zmenu povinnej vlastnosti na dve a viac alternatívnych vlastností nepoužívať aspekty

Na refaktorizáciu povinnej vlastnosti na dve a viac alternatívnych vlastností nie je vhodné používať aspekty.

4.5.1 Kontext

Pri refaktorizácií aplikácie je možnosť zmeniť povinnú vlastnosť, ktorá tvorí jadro aplikácie, na dve alebo viac alternatívnych vlastností. Ďalšou možnosťou je pridanie novej alternatívnej vlastnosti k už existujúcim alternatívnym vlastnostiam.

4.5.2 Problém

Ak sú aspekty postavené nad jadrom aplikácie, potom zmena povinnej vlastností z tohto jadra aplikácie na alternatívne vlastnosti, môže ovplyvniť aj ostatné aspekty.

4.5.3 Dôvody

Refaktorizácia aplikácie na rad softvérových výrobkov si vyžaduje zmeniť „monolitickú“ aplikáciu na množinu vlastností pospájaných určitými väzbami. Niektoré z týchto vlastností sú alternatívne a chceme ich relatívne jednoducho konfigurovať.

4.5.4 Riešenie

Zmena povinnej vlastnosti, ktorá zvyčajne tvorí jadro aplikácie, na dve alebo viac alternatívnych vlastností nie je vhodná, pretože ako uvádzajú autori (Figueiredo a kol.,

[5] Kolaborácia – spolupráca tried na dosiahnutie určitej funkcionality.

2008), táto zmena prináša do výsledného systému viac komponentov, operácií a riadkov kódu, ako by sme ho pridali napríklad podmienenou kompiláciu[6]. Je to najmä z dôvodu, že všetky aspekty obsahujú body spájania postavené na jadre aplikácie, a ak daná alternatívna vlastnosť nebude zahrnutá do konkrétnej inštancie radu softvérových výrobkov, komplikuje to použitie ďalších aspektov.

4.5.5 Príklad

Príkladom môže byť refaktorizácia povinnej vlastnosti na viacero alternatívnych vlastností. Uvedieme to na príklade radu softvérových výrobkov pre mobilné telefóny. Napríklad v pôvodnej aplikácii zmeníme vlastnosť *Média*, mobilného telefónu na viacero voliteľných vlastností *Fotoaparát*, *Video*, *Prehrávač hudby*. Kým v prvom prípade aspekt môže pracovať nad triedou, ktorá obsahuje vlastnosť *Média*, v druhom prípade je veľmi pravdepodobné, že tieto voliteľné vlastnosti budú implementované v samostatných triedach. To si vyžaduje komplikovanejšie aspekty a viac riadkov kódu v aspektoch.

Ukážka kódu povinnej vlastnosti:

```
public class Media {
	private Camera camera = new Camera();
	private Camcorder camcoder = new Camcorder();
	private MusicPlayer musicPlayer = new MusicPlayer();

	public void activateMedia(){
		camera.activate();
		camcoder.activate();
		musicPlayer.activate();
	}

	public boolean startCamera() {
		...
		return true;
	}
}
```

Ukážka kódu s aspektmi, kde vlastnosť *Fotoaparát* nie je povinná:

```
public class Media {
	private Camera camera = new Camera();
	private Camcorder camcoder = new Camcorder();
	private MusicPlayer musicPlayer = new MusicPlayer();

	public void activateMedia(){
		...
```

[6] Prístup manažujúci riadky kódu, ktoré sa skompilujú. Preprocesorová konštrukcia indikuje, ktoré kúsky kódu by sa mali alebo nemali kompilovať, podľa toho ako je nastavená preprocesorová premenná.

```
        }
}
public aspect CameraAspect {
        Camera Media.camera;
        after(): execution( Media.activateMedia()) {
                camera.activate();
                ...
        }
}
```

4.5.6 Diskusia

Pridanie novej alternatívnej vlastnosti do existujúcej množiny alternatívnych vlastností pri aspektovo-orientovanom prístupe potrebuje zmeniť menej komponentov, operácií a riadkov kódu ako neaspektová verzia. V tejto situácií, zmeny nie sú cielené na povinné vlastnosti, a preto neprichádza zmena v bodoch, na ktorých ležia už vytvorené aspekty. Hoci sa viac komponentov a operácií pridá, menej sa zmení už existujúcich (Figueiredo a kol., 2008). Pridanie novej alternatívnej vlastnosti je inkrementálny prístup. Vtedy použitie aspektov môže byť výhodné.

4.6 Na vlastnosti, ktoré nezdieľajú kód a majú pretínajúce záležitosti použiť aspekty

V aplikácií sú vlastnosti, ktoré nezdieľajú kód a majú pretínajúce záležitosti. Vtedy je vhodné použiť aspektovo-orientovaný prístup.

4.6.1 Kontext

V softvéri, a teda i rade softvérových výrobkov, sa vyskytujú pretínajúce záležitosti.

4.6.2 Problém

Rôzne záležitosti bývajú zmiešané v rámci jedného modulu (vlastnosti) alebo tieto záležitosti môžu byť roztrúsené vo viacerých moduloch.

4.6.3 Dôvody

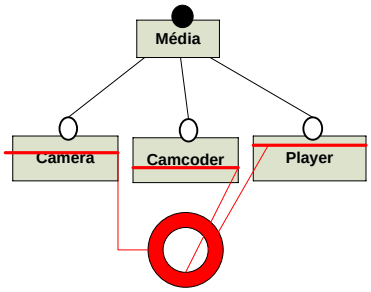
Aspektovo-orientované riešenie vykazuje vyššiu hodnotu v stabilite v oblasti prepletenia kódu a roztrúsenosti cez komponenty. Efektívnosť aspektovo-orientovaných mechanizmov lokalizovať tieto druhy vlastnosti je vďaka schopnosti realizovať voliteľné vlastnosti z tried do množiny zdedených tried a jedného alebo viacerých aspektov, ktoré to spoja (Figueiredo a kol., 2008).

4.6.4 Riešenie

Použiť aspekty na pretínajúce sa záležitosti.

4.6.5 Príklad

Na obr. 6 je zobrazenie pretínajúcich sa záležitosti vo vlastnostiach radu softvérových výrobkov. Dané vlastnosti nezdieľajú žiaden spoločný kód.



Obr. 6: Pretínajúce záležitosti vo vlastnostiach.

4.6.6 Diskusia

V tomto usmernení sa rieši problematika vlastností, ktoré nezdieľajú kód a majú pretínajúce sa záležitosti. Ako priamočiare riešenie sa ponúka aspektovo-orientovaný prístup. Treba tu zvážiť ďalšie okolnosti, viď iné usmernenia.

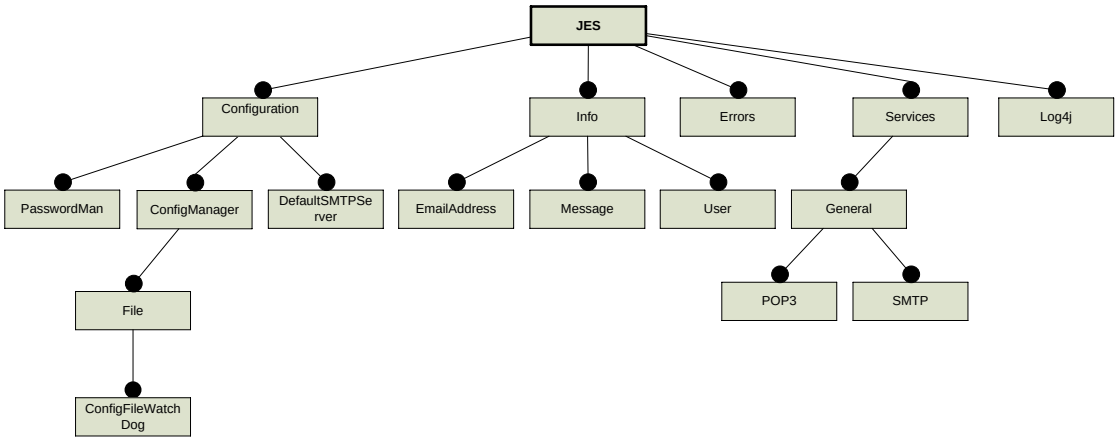
5 Evaluácia usmernení

Cieľom evaluácie usmernení je overenie usmernení pri riešení praktickej úlohy. V tejto kapitole sú prezentované zistenia týkajúce sa nadobudnutých praktických skúseností s používaním usmernení a ich konfrontáciou s reálnym použitím. Pre potreby evaluácie sa vyvíjal rad softvérových výrobkov. Daný rad vznikol z open–source projektu Java Email Server.

V časti 5.1 je predstavený vyvíjaný rad softvérových výrobkov. Časť 5.2 sa venuje zmenám, ktoré bolo potrebné vykonať pre ďalší vývoj radu. V časti 5.3 sú predstavené metriky, ktoré slúžili na vyjadrenie niektorých parametrov vytvoreného softvéru. Spôsob evaluácie je predstavený v časti 5.4. Jednotlivým usmerneniam sa venujú časti 5.5 až 5.11. Záver kapitoly patrí ďalším zisteným poznatkom, ktoré vznikli pri realizácii štúdie.

5.1 Vytvorený rad softvérových výrobkov

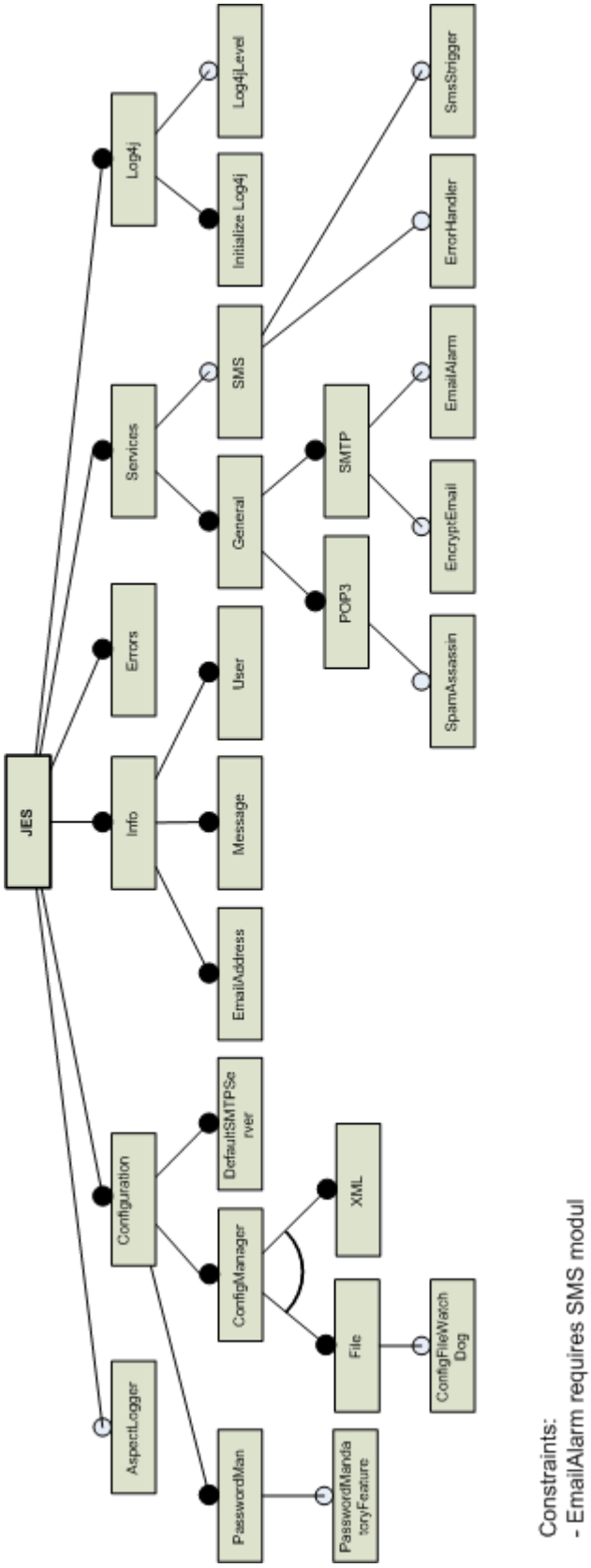
Java Email Server[7](v práci sa bude používať tento anglický názov) je Java SMTP a POP3 e-mail server. Je pod *GNU General Public Licen.*[8] Pri vývoji radu softvérových výrobkov z neho sa použila verzia 1.6.1. Samotný Java Email Server nebol vyvíjaný ako rad softvérových výrobkov. Pôvodná veľkosť projektu bola cca 5400 riadkov zdrojových kódov, 22 tried a ďalšie konfiguračné súbory. Pre potreby vytvorenia radu softvérových výrobkov z pôvodnej aplikácie bolo potrebné vykonať viaceré zmeny. Na obrázku č. 7 sú zobrazené vlastnosti, ktoré boli identifikované v pôvodnej aplikácií Java Email Server.



Obr. 7: Pôvodné vlastnosti v rade Java Email Server.

Model vlastností finálneho radu softvérových výrobkov Java Email Server je zobrazený na obr. č. 8.

[7] http://www.ericdaugherty.com/java/mailserver/
[8] http://www.ericdaugherty.com/java/mailserver/license.html



Obr. 8:Model vlastností dokončeného radu soft. výrobkov Java Email Server.

Z Java Email Server aplikácie bol vytvorený rad softvérových výrobkov s týmito vlastnosťami:

5.1.1 Povinné vlastnosti

Povinné vlastnosti tvoria jadro aplikácie a sú v každej inštancii radu softvérových výrobkov. Tvoria ich moduly, ktoré slúžia pre prácu s heslami, používateľmi, e-mailami, POP3 funkcionalitou, SMTP funkcionalitou, logovací nástroj Log4j.

5.1.2 Variabilné vlastnosti

Alternatívnymi vlastnosťami vo vytvorenom rade softvérových výrobkov je načítavanie konfiguračných parametrov z viacerých typov súborov. V rade sú vytvorené dva spôsoby načítavania týchto parametrov. Trieda *ConfigFileManager* slúži na načítavanie *.properties súborov. Trieda *ConfigXmlManager* slúži na načítanie hodnôt parametrov zo súborov vo formáte XML. Tieto dve vlastnosti sú plne zameniteľné a je k nim možné pridať aj ďalšie vlastnosti, slúžiace na načítanie parametrov z iných formátov súborov. Ďalšie vlastnosti v rade Java Email Server sú:

- *ConfigFileWatchDog* – úlohou implementácie tejto vlastnosti je zistenie zmien v konfiguračných súboroch počas behu aplikácie. Ak zistí zmenu, zavolá príslušné funkcie, ktoré znovunačítajú konfiguračné parametre zo súborov.
- *SMS* – tento voliteľný modul slúži na odosielanie SMS správ.
- *SMSTrigger* – zisťuje, či sú nejaké SMS správy pripravené na odoslanie a odošle ich.
- *SpamAssassin* – zavolá nástroj SpamAssassin[9] na zistenie spamu v doporučenej pošte.
- *EncryptEmail* – táto vlastnosť je určená na šifrovanie e-mailov.
- *EmailAlarm* – odošle SMS autentifikáciu o tom, že došiel e-mail na server.
- *ErrorAlarm* – zachytáva volanie významných výnimiek, chýb v aplikácií a odošle o tom SMS správu administrátorovi Java Email Servera.
- *InitializeLog4J* – slúži na inicializáciu logovacieho nástroja Log4j.
- *Log4jLevel* – poskytuje filtrovanie volania logovacích metód v nástroji Log4j. Má to význam najmä kvôli lepšej výkonnosti volania nástroja Log4j.

5.2 Refaktorizácia vlastností

Java Email Server nebol vyvinutý ako rad softvérových výrobkov. Z tohto dôvodu bolo potrebné refaktorovať niektoré vlastnosti, aby ich bolo možné neskôr konfigurovať. Prvotnou požiadavkou na daný softvér bolo oddelenie niektorých pretínajúcich záležitostí a ich modularizácia.

Logovací nástroj Log4j je pevne zviazaný so zdrojovým kódom. Pre potreby projektu bola inicializácia tohto nástroja presunutá do nového zdrojového súboru, aby bolo možné danú vlastnosť jednoduchšie oddeliť od ostatného zdrojového kódu a ľahšie ju konfigurovať.

Ďalšou funkcionalitou, ktorá bola refaktorovaná, bolo načítavanie konfigurácie pre aplikáciu z konfiguračných súborov. Dôvodom bolo, že pre ďalšie potreby vývoja radu softvérových výrobkov bolo potrebné si vybrať z viacerých alternatívnych vlastností, t.j. z akého formátu konfiguračných súborov si bude aplikácia načítavať konfiguráciu.

[9] http://spamassassin.apache.org/

Refaktorizáciou sa vytvorila trieda *ConfigManager*, ktorá je nezávislá od typu konfiguračných súborov. Triedy, ktoré dedia od tejto triedy, predstavujú alternatívne vlastnosti. Potomkom triedy *ConfigManager* je trieda *ConfigFileManager,* ktorá vznikla refaktorizáciu pôvodných zdrojových súborov. Táto trieda predstavuje implementáciu načítavania parametrov aplikácie z *.properties súborov.

Funkcionalita vlastnosti, ktorá testovala, či sa zmenili konfiguračné súbory aplikácie počas jej behu (*ConfigFileWatchDog)*, bola vyčlenená do vlastného súboru. Po tejto zmene je potom jednoduchšie danú vlastnosť začleňovať a konfigurovať v danom rade softvérových výrobkov.

5.3 Použité metriky

Pre potreby kvalitatívneho porovnania aspektovo-orientovaného a objektovo-orientovaného prístupu k vývoju radu softvérových výrobkov boli použité viaceré metriky. Výber metrík ovplyvnili viaceré súvislosti. Základom bola ich relevantnosť, či je ich vhodné použiť pri aspektoch a v objektovo-orientovanom kóde, aby nebola niektorá vlastnosť týchto paradigiem ignorovaná. Ďalšiu pozornosť si vyžadovalo, aby metriky zohľadňovali implementačnú rozpracovanosť niektorých vlastností, pri ktorých sa dôraz kládol najmä na naznačenie konfigurovateľnosti. Z dôvodu, že je vyvíjaný rad softvérových výrobkov, boli zohľadnené aj dopady na konfigurovateľnosť daného radu. Objektovo-orientované metriky, ktoré vyjadrujú napr. cyklomatickú zložitosť, počet parametrov, dĺžku metód a im podobné, neboli použité, pretože síce vyjadrujú kvalitu kódu z určitého pohľadu, ale nie z pohľadu previazanosti kódu a možnosti konfigurovať vlastnosti. V práci použité metriky boli vyhodnotené pomocou automatizovaných nástrojov. Existuje viacero metrík, ktoré by sa dali použiť najmä s ohľadom na oddelenie záležitostí (Figueiredo a kol., 2008) , avšak chýba použiteľný nástroj, ktorý by uľahčil ich meranie.

Použité metriky je možné členiť na tri základné okruhy. Prvým okruhom sú metriky vyjadrujúce veľkosť zmien v kóde. Tie majú význam najmä z hľadiska implementácie. Ďalšie metriky sa používajú na zachytenie previazanosti komponentov. Tretia skupina metrík vyjadruje niektoré vlastnosti priečinkov, najmä z pohľadu prenositeľnosti.

Počet riadkov kódu (ang. Lines of Code, LOC) – vyjadruje veľkosť modulov, tried, aspektov implementovaných vlastností v riadkoch zdrojových kódov. Významný je vo vzťahu k hrubému porovnaniu veľkosti komponentov a prácnosti ich implementácie (z pohľadu počtu riadkov, nie zložitosti kódu).

Počet zasiahnutých tried – vyjadruje do koľkých tried je potrebné urobiť zásah pri implementácií určitej vlastnosti.

Ďalšie metriky uvedené nižšie vychádzajú z prác Ceccato a Tonella. (2004), Chidamber a Kemerer (1994), reprezentujú metriky vhodné pre porovnanie nasadenia aspektov podľa Stichmialek (2005). Označenie modul reprezentuje triedy alebo aspekty.

Vážené operácie v module (ang. Weighted Operations in Module, WOM). Táto metrika zachytáva internú komplexnosť modulu v počte implementovaných funkcií. Moduly

s veľkým počtom operácií sú zvyčajne viac aplikačne špecifické, čo limituje ich možné znovupoužitie.

Dĺžka stromu dedenia (ang. Depth of Inheritance Tree, DIT) je dĺžka najdlhšej cesty z modulu do triedy, resp. aspektu k rodičovskému modulu v hierarchii dedenia. Podobne ako relevantná objektovo-orientovaná metrika, aj táto meria rozsah vlastností. Čím je trieda, resp. aspekt hlbšie v hierarchii dedenia, tým väčší počet operácií môže byť zdedený, a tým je viac obtiažnejšie mu porozumieť a zmeniť niektoré prvky v tomto reťazci. To, že aspekty môžu meniť vzťahy dedenia je potrebné vziať do úvahy.

Počet potomkov (angl. Number Of Children,NOC) je počet priamych podtried alebo pod-aspektov daného modulu. Počet potomkov modulu indikuje pomer potenciálnej nezávislosti modulu na vlastnostiach zdedených z daného modulu. Väčší počet potomkov indikuje väčšiu znovupoužiteľnosť, ak je dedičnosť formou znovupoužitia.

Stupeň pretínajúcich záležitostí v aspekte (angl. Crosscutting Degree of an Aspect, CDA) vyjadruje počet modulov zasiahnutých pretínajúcimi záležitosťami a medzitypovými deklaráciami daným aspektom.

Previazanosť zachytených modulov (angl. Coupling on Intercepted Modules, CIM) je počet modulov alebo rozhraní explicitne vymenovaných v bodových prierezoch patriacich danému aspektu. Vysoké hodnoty tejto metriky indikujú vysokú previazanosť aspektu s danou aplikáciou a nižšiu všeobecnosť, znovupoužiteľnosť. Vysoká hodnota *Stupňa pretínajúcich záležitostí v aspekte* a nízka hodnota *Previazanosť zachytených modulov* sú zvyčajne žiaduce. Táto metrika nebola implementovaná v dostupnom nástroji na meranie metrík, preto sa nepoužila.

Zviazanosť vykonaných videní (angl. Coupling on Advice Execution, CAE) je počet aspektov obsahujúci videnia, ktoré môžu byť spustené vykonaním operácií v danom module. Ak správanie operácie môže byť ovplyvnené videním v aspekte, je tu implicitná závislosť operácie od videnia. Takto je daný modul zviazaný s aspektom obsahujúcim videnie a neskoršia zmena v jednom z týchto prvkov môže mať dopad aj na ďalšie, ktoré sú vo väzbe.

Previazanosť volania metód (angl. Coupling of Method Call, CMC) vyjadruje počet modulov alebo rozhraní deklarovaných metód, ktoré je možné zavolať z daného modulu. Použitie vysokého počtu metód z rozličných modulov indikuje, že funkcia daného modulu nemôže byť ľahko izolovaná od ostatných. Vysoká previazanosť je asociovaná s vysokou závislosťou od funkcií v iných moduloch.

Previazanosť premenných (angl. Coupling on Field Access, CFA) je počet modulov a rozhraní deklarujúcich premenné, ktoré sú sprístupnené daným modulom.

Previazanosť medzi modulmi (angl. Coupling between Modules, CBM) predstavuje počet modulov alebo rozhraní deklarujúcich metódy alebo premenné, ktoré je možné zavolať alebo sprístupniť daným modulom. Nadbytočné viazanie medzi objektmi tried je škodlivé modulárnemu návrhu a prenositeľnosti komponentov.

Odpoveď modulu (angl. Respone For a Module, RFM) je počet metód a videní potenciálne vykonaných ako odpoveď na správu doručenú danému modulu. Táto metodika sa venuje potenciálnej komunikácií medzi danými modulmi.

Nedostatočná súdržnosť operácií (angl. Lack of Cohesion in Operations, LCO) je počet párov operácií pracujúcich na odlišných premenných tried mínus páry pracujúce na spoločných premenných. Hodnota tejto metriky bude nízka, ak všetky operácie v triede alebo aspektu zdieľajú spoločnú dátovú štruktúru s ktorou manipulujú alebo ju sprístupňujú.

Ďalšie použité metriky, merajúce najmä z pohľadu priečinku sú podľa Martin (1994) a Stochmialek (2005).

Počet typov (angl. Number of Types, NOT) určuje počet typov v danom priečinku. Je to indikátorom rozšíriteľnosti priečinku.

Abstraktnosť (angl. Abtractness, A) je pomer abstraktných modulov k celkovému počtu modulov v priečinku.

Aferentné viazanie (angl. Afferent Coupling, Ca) je počet modulov mimo priečinku, ktoré sú závislé na moduloch v priečinku. Je to indikátorom zodpovednosti priečinku.

Eferentné viazanie (angl. Efferent Coupling, Ce) je počet modulov v priečinku, ktoré sú závislé na moduloch mimo priečinku. Je to indikátor nezávislosti priečinku.

Modifikované eferentné viazanie (angl. Modified Efferent Couplings, Ce). Originálna verzia tejto metriky neindikuje, aké veľké je eferentné viazanie.

Nestabilita (angl. Instability, I) je pomer eferentného viazania (Ce) a celkového viazania (Ce + Ca). Táto metrika je indikátorom pružnosti priečinka na zmenu.

Normalizovaná vzdialenosť od hlavnej sekvencie (angl. Normalized Distance from Main Sequence, Dn) – táto metrika je indikátorom vyváženia priečinku medzi abtraktnosťou a stabilitou.

Podľa Lopez-Herrejon (2006) je vhodné deliť pretínajúce záležitosti na homogénne a nehomogénne. Ak ním definovaná hodnota metriky **Program Heterogeneity Quotient** sa blíži k hodnote 1, v programe je vhodnejšie využiť prednosti aspektov. Ak sa hodnota blíži k 0, to môže mať dve interpretácie: a) program má veľmi málo kúskov videní, ktoré pretínajú viacero tried, b) veľa záležitostí pretínajúcich triedy sú medzitypové deklarácie. Táto metrika môže napomôcť rozhodnúť, či je vhodnejšie použiť aspekty alebo objektovo-orientované postupy na konfiguráciu vlastností. Pre rátanie tejto metriky chyba vhodný nástroj, preto sa nepoužila pri evaluácii usmernení.

5.4 Spôsob evaluácie

Pre potreby evaluácie usmernení z časti č. 4 bol vytvorený rad softvérových výrobkov z Java Email Server aplikácie. Paralelne vznikli dve nezávislé implementácie radu softvérových výrobkov, vytvorené pomocou jazykov AspectJ a jazyka Java. Prvá implementácia je konfigurovaná pomocou aspektovo-orientovaného prístupu, druhá implementácia pomocou objektovo-orientovaného prístupu. Obidva prístupy

k implementácií vychádzajú z rovnakého základu (zdrojových súborov). Implementácia konfigurácie vlastností je v niektorých prípadoch iba naznačená, ale z pohľadu evaluácie postačuje. Aspektovo-orientovaná i objektovo-orientovaná implementácia daného radu sú si v mnohých ohľadoch dosť podobné. Líšia sa najmä spôsobom „pripojenia“ vlastností k radu (vo väčšine prípadov). Zdrojové súbory vyvíjaného radu softvérových výrobkov, ktoré boli použité pri evaluácií, sú na priloženom CD médiu a taktiež na tejto adrese.[10]

5.4.1 Aspektovo-orientované riešenie radu

Pri aspektovo-orientovanej implementácií radu softvérových výrobkov sa vlastnosti uplatňujú nasledujúcim spôsobom. Každá vlastnosť je riešená v samostatnom aspekte. Je vytvorený zoznam vlastností, ktoré sa pre danú inštanciu radu softvérových výrobkov použijú. Následne sa pomocou nástroja Apache Ant[11] zavolá kompilátor. Kompilátor podľa zoznamu vlastností vtká daný aspekt do kompilovaných súborov. Konfigurácia vlastností sa deje pri kompilácii radu.

5.4.2 Objektovo-orientované riešenie radu

Objektovo-orientované riešenie je urobené nasledujúcim spôsobom. Rozhodnutie, či sa daná vlastnosť uplatní je dané priradenou hodnotou pre túto vlastnosť v konfiguračnom súbore (*true*, *false* hodnota). Štartom daného radu softvérových výrobkov sa načíta konfigurácia vlastností. Ak v procese vykonávania programu sa dostaneme na miesto, kde sa môže daná vlastnosť uplatniť, pomocou podmienky *if (vlastnost == true)* sa rozhodne, či sa vlastnosť uplatní alebo nie.

```
if(OOConfig.getInstance().isInitializeLog4jEnabled()) {
      InitializeLogging.initializeLogging(directory);
}
```

5.4.3 Rozdielnosť spôsobov konfigurácie

Pri aspektovo-orientovanom spôsobe konfigurácie vlastností použitom pri evaluácií sa rad konfiguruje pri kompilácii zdrojových súborov. Počas spúšťania radu softvérových výrobkov nie je možné meniť konfiguráciu radu, pretože tá bola určená pri kompilácii a je pevne daná v štruktúre programu. Oproti tomu objektovo-orientované riešenie umožňuje dynamickú zmenu konfigurácie radu softvérových výrobkov pri štarte programu. Ak sa zmení nastavenie konfigurácie daného radu, pri opätovnom štarte tohto radu vznikne rad s odlišnou konfiguráciou. Nevýhodou tohto riešenia je, že konfigurácia je riešená dynamicky v programe aj s nárokmi spojenými s takýmto riešením (výpočtové a pamäťové nároky). Tieto závery vyplynuli z výsledkov konkrétnej implementácie pri evaluácií usmernení.

5.4.4 Použité nástroje, konfigurácia vytvoreného radu softvérových výrobkov

Rad Java Email Server bol vyvíjaný v nástroji Eclipse [12] s pluginom AJDT. [13] Pre aspektovo-orientovaný prístup sa používal jazyk AspectJ.[14] Pre daný rad boli vytvorené 4

[10] http://lapis.yweb.sk/DP/jes.zip
[11] http://ant.apache.org
[12] http://www.eclipse.org/
[13] http://www.eclipse.org/ajdt/
[14] http://www.eclipse.org/aspectj/

konfigurácie označované písmenami A, B , C, D. Tieto konfigurácie je možné použiť pri objektovo-orientovanej i aspektovo-orientovanej verzii. Pre aspektovo-orientovanú verziu radu bol vytvorený skript v nástroji Apache Ant. Metriky boli vyhodnotené pomocou nástrojov AOPmetrics[15] a Metrics[16].

5.5 Refaktorizácia radov softvérových výrobkov

Java Email Server bol vyvíjaný ako aplikácia, ktorá používa všetky svoje časti a je poskytovaná každému používateľovi v rovnakej forme a funkcionalite. Pri použití aspektov na niektoré vlastnosti sa zhoršila schopnosť programátora porozumieť štruktúre vykonávania programu, pretože pri rade softvérových výrobkov vyvíjaných refaktorizáciou aspekty nereflektujú pôvodnú objektovo-orientovanú štruktúru programu. Pre ďalšie použitie pri vývoji bola trieda *ConfigManager* objektovo-orientovane refaktorovaná, čím sa uľahčil ďalší vývoj. Bez tejto zmeny by bolo problematické viazať ďalšie vlastnosti na túto vlastnosť. Ak by sa refaktorizácia triedy *ConfigManager* uskutočnila pomocou aspektov, jej kód by sa stal pomerne nečitateľným a nezrozumiteľným pre vývojára. Samotná objektovo-orientovaná refaktorizácia je triviálnejšia operácia ako vykonanie refaktorizácie pomocou aspektov.

5.6 Použitie aspektov na povinné vlastnosti bez pretínajúcich záležitostí

Funkcionalita poskytovaná triedou *PasswordManager* bola nahradená aspektom *PasswordMandatoryFeature*. Trieda *PasswordManager* poskytovala metódu, ktorá vykonávala hash hesla a bola volaná na viacerých miestach v programe. Je to povinná vlastnosť. Nahradenie tejto funkcie inou v aspekte prinieslo znepriehľadnenie kódu a objektovo-orientovanej štruktúry. Počet zdrojových riadkov bol približne rovnaký ako pri objektovo-orientovanom kóde, pretože boli pri volaní prekryté metódy volané v pôvodnej triede. V tabuľke č. 1 je počet riadkov kódu, ktoré boli potrebné na zmenu funkcionality v povinnej vlastnosti.

Tab. 1: Počet riadkov kódu pridaných na zmenu povinnej vlastnosti.

Aspektovo-orientovaná zmena	Objektovo-orientovaná zmena
9	11

V tabuľke č. 2 sú zobrazené hodnoty príslušných aspektovo-orientovaných metrík na povinnej vlastnosti *PasswordManager*.

Tab. 2: AO metriky použité na PasswordManager.

	LOC	WOM	DIT	NOC	CFA	CMC	CBM	CDA	CAE	RFM	LCO
OO	45	3	0	0	0	1	1	0	0	5	0
AO	35	2	0	0	0	0	0	0	0	2	0

5.7 Aspekty redukujú replikovaný kód

Homogénne pretínajúce záležitosti predstavujú jedno z najvhodnejších miest na aplikáciu aspektov. Jednou z takýchto pretínajúcich záležitostí je volanie logovacieho nástroja Log4j. V aplikácií Java Email Server sa často opakoval rovnaký, resp. podobný kus kódu.

[15] http://aopmetrics.tigris.org/
[16] http://metrics.sourceforge.net/

V tabuľke č. 3 je zobrazený počet volaní jednotlivých metód v pôvodnom zdrojovom kóde. Každé jedno volanie predstavovalo aspoň jeden riadok zdrojového kódu. Tento kód je takmer identický v celom programe, zahlcuje a zneprehľadňuje logiku aplikácie.

Tab. 3: Počet volaní metód pre logovanie.

Názov metódy	Počet volaní
log.isDebugEnable()	27
log.debug()	47
log.isInfoEnable()	10
log.info	26

Pôvodná verzia:

```
if( log.isDebugEnabled() ){
      log.debug( "Loading SMTP Message " + messageFile.getName() );
}
```

Po použití aspektov.

```
log.debug( "Loading SMTP Message " + messageFile.getName() );
```

Aspekt *LoggingLevel* zredukoval počet riadkov kódu. Namiesto volania metódy *isDebugEnable()*, ktorá bola volaná 27-krát, bolo ušetrené v zdrojovom kóde volania tejto metódy. Navyše kód bol vylepšený o funkcionalitu metódy *isDebugEnable()* na ďalších 20 potenciálnych miestach, teda celkovo na 47 miestach za súčasného zmenšenia počtu riadkov. Podobne aj presunutie metódy *isInfoEnable()* do aspektu prinieslo úsporu 10-krát zavolať danú metódu v kóde (potenciálne 26-krát). Ďalšie úrovne logovania (Error, Fatal, Warn, Trace) sa v tomto prípade neuvažovali, použili sa najčastejšie vyskytujúce úrovne logovania v Java Email Server aplikácií.

Funkcionalitou, ktorá bola pridaná ako voliteľná vlastnosť do vytvoreného radu softvérových výrobkov, bola vlastnosť *ErrorAlarm*. Tá pri vyhodení pre aplikáciu významných výnimiek odošle administrátorovi SMS správu o významnej chybe, ktorá nastala počas behu aplikácie. Aspektová implementácia zachytenie výnimiek rieši jedným aspektom. Objektovo-orientované riešenie potrebuje mať v každej výnimke, o ktorej chceme dať administrátorovi správu, volanie metódy, ktorá túto správu odošle. T.j. v každej výnimke, ktorá nás zaujíma, aspoň jeden riadok kódu (pozri tabuľku č. 4). Túto zmenu je potrebné urobiť v každom súbore, kde zachytávame tieto výnimky.

```
catch (IOException ioException) {
        ErrorAlarm.error("Error loading properties ");
      ...
}
```

Tab. 4: Počet riadkov kódu potrebných pre vlastnosť ErrorAlarm.

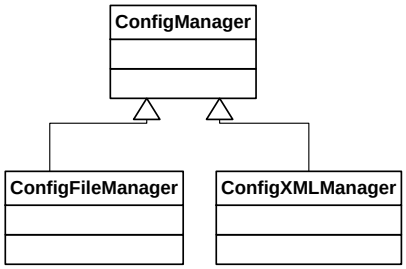
Aspektovo-orientované riešenie	Objektovo-orientovaná riešenie
9	37

Za významné výnimky sú pri daných riešeniach považované výnimky typu *IOException*, *RuntimeException* a *SocketException*.

Pri použití metrík neboli uvažované vyššie spomenuté homogénne pretínajúce záležitosti, pretože by to ovplyvnilo celkové výsledky a pozornosti by unikli ďalšie súvislosti. Pri homogénnych pretínajúcich záležitostiach sa jednoznačne ako výhodnejšie riešenie javí použitie aspektov.

5.8 Zmena povinnej vlastnosti na alternatívne

V Java Email Serveri bola realizovaná zmena povinnej vlastnosti na dve, resp. viac alternatívnych vlastností. Zmena bola vykonaná objektovo-orientovanou refaktorizáciou. Spočívala v presunutí časti funkcionality do rodičovskej triedy, ktorá bola pre tento účel vytvorená (obr. č. 10). Refaktorizáciou bol upravený zdrojový kód tak, aby ho bolo možné neskôr použiť v alternatívnych vlastnostiach. Po tejto zmene bolo pomerne jednoduché vytvoriť jednu alebo viacero alternatívnych vlastností pomocou aspektov. Bez tejto zmeny na objektovo-orientovanej úrovni by zmena pomocou aspektov bola omnoho zložitejšia, problematickejšia na ďalšie rozšírenie a priniesla by väčšie množstvo riadkov zdrojového kódu. Pomocou videní a medzitypových deklarácií je možné vytvoriť akoby refaktorovanú štruktúru tried, ale prinieslo by to zneprehľadnenie štruktúry programu a nebolo by dané riešenie tak elegantné ako objektovo-orientované.



Obr. 9: Zmena povinnej vlastnosti na alternatívne.

Pridanie ďalších alternatívnych vlastností je z pohľadu počtu pridaných riadkov kódu porovnateľné u aspektového i objektovo-orientovaného prístupu (pozri tab. č. 5).

Tab. 5: Zmeny potrebné pre zapojenie ďalšej alternatívnej vlastnosti ConfigXMLManager do existujúcej infraštruktúry programu.

	Aspektovo-orientovaná zmena	Objektovo-orientovaná zmena
Trieda, aspekt	Vytvorenie nového aspektu	Úprava kódu triedy
Počet riadkov kódu	11	3

5.9 Použitie aspektov na pretínajúce záležitosti

Pridanie nových vlastností, najmä voliteľných, prináša pri objektovo-orientovanom spôsobe vývoja radu softvérových výrobkov do kódu pretínajúce záležitosti. V objektovo-orientovanej implementácii Java Email Servera, ak bolo v aplikácií sa potrebné rozhodnúť, či sa daná vlastnosť uplatní alebo nie, bolo potrebné vložiť rozhodovací blok - podmienku *if* (pozri časť 5.4.2). Takéto riešenie zneprehľadňovalo pôvodný kód a spôsobovalo zapletenosť kódu.

Aspektovo-orientované riešenie naproti tomu nespôsobuje zapletenie kódu, pretože do pôvodných zdrojových kódov nepridáva žiaden ďalší kód. Daný spôsob umožnil pridávanie nových vlastností, pri ktorom rástla veľkosť zdrojových súborov, avšak pôvodný kód vykazoval rovnakú úroveň prepletenia kódu (pozri tab. č. 6 a 7.). Aspektovo-

orientované riešenie použilo pre každú vlastnosť nový aspekt (rástol počet súborov), ktorých pomenovanie reprezentovalo ich funkcionalitu. Je podľa môjho názoru efektívnejšie riešenie dať vlastnosť do aspektu ako hľadať v zdrojových súboroch podmienku *if*.

Tab. 6: Porovnanie hodnôt metrík pôvodnej verzie (po refaktorizácii načítavania konfigurácie), objektovo-orientovanej verzie i aspektovo-orientovanej verzie radu Java Email Server. Hodnoty vznikli aritmetickým priemerom cez všetky triedy a aspekty.

	LOC	WOM	DIT	NOC	CFA	CMC	CBM	CDA	CAE	RFM	LCO
Pôv.	124,73	8,55	0,36	0,05	0,14	3,55	3,68	0,00	0,00	15,77	54,27
OO	96,21	6,68	0,32	0,06	0,09	3,15	3,24	0,00	0,00	12,68	37,26
AO	88,17	6,00	0,28	0,06	0,08	2,64	2,72	1,00	1,00	11,92	33,22

Tab. 7: Hodnoty niektorých metrík určených na priečinky podľa (Stochmialek, 2005).

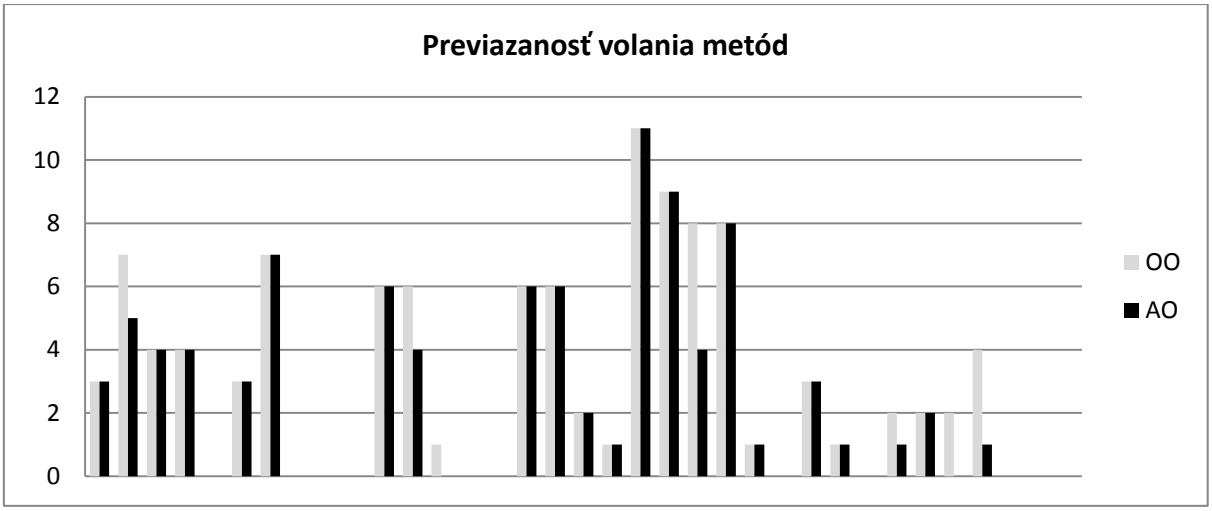
	NOT	A	RMartin Ce	RMartin Ca	RMartin I	RMartin D	Ce	Ca	I	Dn
Pôv.	3,14	0,06	2,14	4,14	0,45	0,48	7,14	4,14	0,61	0,33
OO	3,78	0,04	2,44	4,67	0,40	0,56	6,78	4,67	0,53	0,43
AO	4,00	0,05	3,00	4,22	0,43	0,53	7,33	4,22	0,56	0,40

V tabuľkách číslo 6 a 7 boli vykonané merania na kóde, v ktorom sa výraznejšie neriešili homogénne pretínajúce záležitosti opísané v časti č. 5.7 (*LoggingLevel* a *ErrorAlarm* vlastnosti).

Jednou z možností ako znížiť roztrúsenosť kódu v Java Email Serveri je zmena nástroja Log4j na logovanie pomocou aspektov. Tu treba zvážiť pozitíva i negatíva tohto kroku. V Java Email Serveri sa tento krok neuskutočnil pretože nástroj Log4j bol pomerne komplexne včlenený do funkcionality aplikácie (16 tried a aspektov) a jeho nahradenie aspektmi by si vyžadovalo rozsiahlu refaktorizáciu kódu. Ďalším dôvodom, ktorý sa bral do úvahy bolo, že síce by bol odstránený kód logovania z tried aplikácie (značné množstvo kódu, 250+ riadkov kódu), ale za cenu trochu iného formátu logovaných správ. Pri vývoji nového radu softvérových výrobkov je efektívnejšie použiť logovanie prostredníctvom aspektov ako je tomu už vo vyvinutej aplikácií, kde je logovanie pomocou nástroja Log4j previazané priamo s aplikačnou logikou.

5.10 Použitie aspektov na vlastnosti, ktoré nezdieľajú kód

Na obr. č. 10 je zobrazené porovnanie previazanosti volania metód medzi objektovo-orientovanou a aspektovou verziou radu softvérových výrobkov. Aspektovo-orientovaná verzia vykazuje nižšiu previazanosť volania metód pri niektorých triedach a tým aj relatívne lepšiu modulárnosť a znovupoužiteľnosť jednotlivých tried.



Obr. 10: Porovnanie previazanosti volania metód medzi objektovo-orientovanou a aspektovou verziou.

5.11 Pridanie nových voliteľných vlastností

Pridanie nových voliteľných vlastností bolo pomerne bezproblémové, pretože sa zväčšovala funkcionalita aplikácie. Pridanie novej voliteľnej vlastnosti pomocou aspektov, resp. objektovo-orientovaným prístupom je z pohľadu počtu riadkov zdrojových kódov porovnateľné (pozri tab. č. 8). Sú to vlastnosti: *SMSTrigger*, *SpamAssassin*, *EncryptEmail*, *EmailAlarm*, *InitializeLog4J*. Vlastnosti, u ktorých je z pohľadu počtu riadkov zdrojových kódov vhodnejšie použiť aspekty, sú *ErrorAlarm* a *Log4jLevel*. Zároveň ak by sa dané vlastnosti museli implementovať objektovo-orientovane, vyžiadalo by si to zmenu vo veľkom počte tried (11 a 22 tried).

Tab. 8: Niektoré charakteristiky vlastností v rade softvérových výrobkov. Vlastnosť môže byť povinná (P), alternatívna (A) alebo voliteľná (V). Použité usmernenie predstavuje poradové číslo použitého usmernenia. Ďalšou charakteristikou je počet riadkov kódu a počet tried, v ktorých bolo potrebné vykonať zmeny.

Vlastnosť	Typ vlastnosti	Použité usmernenie	Počet riadkov		Počet tried	
			AO	OO	AO	OO
ConfigFileManager	A	1, 4	0	3	0	1
ConfigXMLManager	A	1, 4	7	*3*	*1*	*2*
ConfigFileWatchDog	V	1	77	68	1	2
PasswordMandatoryFeature	P	2	8	7	1	1
SpamAssassin	V	5	7	9	1	2
EncryptEmail	V	5	9	9	1	2
EmailAlarm	V	5	9	37	1	1
ErrorAlarm	V	3, 5	11	37	1	11
SMSTrigger	V	5	8	38	1	2
InitializeLog4j	V	1	70	68	1	2
Log4jLevel	V	3,5	18	76	1	22

5.12 Porovnanie ďalších charakteristík medzi objektovo-orientovaným prístupom a aspektmi

Pri vývoji radu softvérových výrobkov Java Email Server poskytujú aspektový a objektovo-orientovaný prístup rozdielne spôsoby ako sa dané vlastnosti uplatňujú vo vytvorenom rade softvérových výrobkov. Niektoré detaily si je možné pozrieť v tab. 9.

Tab. 9: Rozdiely pri konfigurácií vlastností.

	AO	OO
Konfigurácia vlastností	Pri kompilácií	Pri štarte programu
Uplatnenie vlastností	Pri kompilácií	Pri behu programu
Možnosť rekonfigurácie bez kompilácie	Nie	Áno
Potreba zasahovať do pôvodných zdrojových kódov pri vytvorení novej vlastnosti	Nie	Áno

5.13 Zhrnutie evaluácie

Počas evaluácie boli overené jednotlivé usmernenia. Čiastkové výsledky sú popísané v častiach 5.5 až 5.11. Výsledky tejto štúdie ukazujú, že aspekty je zvlášť vhodné použiť na homogénne pretínajúce záležitosti. Ďalším vhodným miestom pre použitie aspektov sú voliteľné vlastnosti, najmä z pohľadu nízkej previazanosti kódu. Z hľadiska možnosti konfigurovať vlastnosti sú porovnateľné aspektovo-orientovaná i objektovo-orientovaná implementácia. Výsledky evaluácie boli ovplyvnené možnosťou k prístupu a editovaniu zdrojových kódov, čo nemusí byť možné pri každom riešenom projekte. Ak by nebola možnosť editovať zdrojové súbory, je vhodnejšie pri riešení sa prikloniť na stranu aspektovo-orientovaného prístupu.

Evaluácia ukázala opodstatnenosť jednotlivých usmernení a ich vhodnosť použitia pri vývoji konkrétnych radov softvérových výrobkov.

6 Porovnanie výsledkov s inými prístupmi a prácami

Viacero autorov, ktorí sa venujú použitiu aspektov pri vývoji softvéru a najmä pri konfigurácií radov softvérových výrobkov, porovnáva aspekty s tradičným, objektovo-orientovaným prístupom alebo aj novšími prístupmi ako napríklad Framed Aspects, Aspectual Mixin Layers, a pod. Cieľom tejto práce nebolo nájsť najlepší možný prístup aký existuje na konfiguráciu radov softvérových výrobkov, ale napísať a overiť usmernenia pre tvorbu radov pomocou aspektov.

Homogénne pretínajúce záležitosti je vhodné riešiť aspektmi. Prináša to okrem zmenšenia kódu aj jeho väčšiu prehľadnosť. K podobným výsledkom sa dopracovali viacerí autori vo svojich prácach (Apel a Batory, 2006), (Kvale a kol., 2005), (Lopez-Heron a kol., 2006). Použitie aspektov pri heterogénnych pretínajúcich záležitostiach neprináša také jednoznačné výsledky ako pri homogénnych pretínajúcich záležitostiach.

Figueiredo a kol. (2008) vo svojej práci prezentovali výsledky z kvantitatívnej štúdie, ktorá sa zamerala z viacerých pohľadov na stabilitu návrhu aspektovo-orientovanej implementácie dvoch radov softvérových výrobkov. Z výstupov ich práce vyplýva, že aspekty môžu niekedy zlepšiť vlastnosti vyvíjaných radov softvérových výrobkov, najmä ak sa použijú na voliteľné vlastnosti. Autori v práci používajú viaceré metriky na meranie kvality softvéru (použitie niektorých metrík je však podľa môjho názoru diskutabilné). Použitie všetkých nimi použitých metrík a číselných výsledkov z nich bez posúdenia ďalších okolností na vyvíjaný rad Java Email Server by viedlo k veľmi skresľujúcim výsledkom z dôvodu rozdielnych prístupov a zdrojov. Problémy vznikli aj s nástrojom, pomocou ktorého zisťovali hodnoty daných metrík.

Refaktorizácii rozsiahlej používanej aplikácie (Berkeley DB) na rad softvérových výrobkov sa venovali Ch. Kästner, S. Apel a D. Batory (2006). Výsledkom ich práce bolo, že ako vzrastal počet aspektov v závislosti od nárastu počtu vlastností, poklesla čitateľnosť a udržovateľnosť kódu. Väčšina z unikátnych a silných čŕt jazyka AspectJ nebola potrebná. Autori dokumentovali, že AspectJ je nie je vhodný na implementáciu vlastností refaktorovanej už vyvinutej aplikácie. V rade Java Email Server bolo potrebné tiež najprv refaktorovať zdrojové kódy a pripraviť ich na konfiguráciu vlastností. Autori sa venovali najmä refaktorizácii, síce pomerne rozsiahleho systému, ale už vyvinutého. Naproti tomu v rade Java Email Server bol okrem počiatočnej refaktorizácie uskutočnený inkrementálny vývoj daného radu softvérových výrobkov, čo prinieslo zaujímavejšie výsledky.

Pre konfiguráciu radov softvérových výrobkov a na uľahčenie ich vývoja boli použité viaceré prístupy. Niektoré z nich si vyžadujú podporu nástrojov, ktoré uľahčujú použitie týchto techník. *Colored IDE (CIDE)*[17] je nástroj pre rady softvérových výrobkov, ktorý pomáha analyzovať a dekomponovať kód v už vytvorených aplikáciách. Vývojár vytvára asociácie medzi kúskami kódu a vlastnosťami, t.j. značkuje kód. Výhodou tohto nástroja je, že pomáha pracovať s vlastnosťami a kódom aj na pomerne jemnej úrovni granularity. Je to podstatné pri refaktorizácií už vytvorený aplikácií na rad softvérových výrobkov. Tím

[17] http://fosd.de/cide

autorov (Kästner a kol., 2008) ho úspešne použil na vývoj dvoch radov softvérových výrobkov.

Sven Apel a Don Batory (2006) sa vo svojej štúdii pokúsili skombinovať Feature-Oriented Programming (objektovo-orientovaný prístup) a aspektovo-orientovaný prístup do prístupu Aspectual Mixin Layers, kde sa obe tieto komplementárne technológie kombinujú a prekonávajú svoje individuálne limitácie. Najväčší význam autori vidia v použití aspektov v radoch softvérových výrobkov tam, kde tradičné objektovo-orientované techniky zlyhávajú: aspekty redukujú replikovaný kód, pomáhajú modularizovať dynamické pretínajúce záležitosti (napr. pomocou *cflow*) a podporujú neskoršie pridanie vzťahov dedičnosti.

Axel A. Kvale a kol. (2005) sa zamerali na budovanie systémov založených na COST[18] komponentoch, ktoré používajú aspektovo-orientovaný prístup. Výsledky ich štúdie ukazujú, že integrácia COST pomocou aspektov môže pomôcť zvýšiť zameniteľnosť komponentov, ak pretínajúce záležitosti v „spojovacom“ kóde sú homogénne. Extrakcia heterogénnych alebo čiastkových homogénnych pretínajúcich záležitostí v „spojovacom“ kóde neposkytuje prospech. Tiež sa ukázalo, že niektoré limitácie v nástrojoch určených pre aspektovo-orientovaný vývoj sťažujú použitie aspektov pri vývoji systémov založených na COST komponentoch.

[18] Commercial-off-the-shelf: komponenty dodávané tretími stranami.

7 Zhodnotenie

V tejto práci sa podarilo vytvoriť pravidlá pre tvorbu usmernení pre použitie aspektov v radoch softvérových výrobkov. Taktiež sú identifikované niektoré usmernenia, ktoré môžu pomôcť pri tvorbe radu softvérových výrobkov pomocou aspektov. Definovaním usmernení sa podarilo dokázať, že vhodné použitie aspektovo-orientovaného prístupu pri tvorbe radov softvérových výrobkov sa dá vyjadriť vo forme usmernení. Bola vytvorená štúdia na overenie jednotlivých usmernení a jej výsledky sú prezentované v tejto práci.

Usmernenia sa použili pri vývoji radu softvérových výrobkov – Java Email Server. Boli vytvorené povinné a variabilné vlastnosti pomocou usmernení. Aspektovo-orientované riešenie sa porovnávalo voči objektovo-orientovanému riešeniu. Podrobné výsledky sú zhrnuté v kapitole *5 Evaluácia usmernení*. Hlavný prínos v použití aspektov je v redukcii kódu pri homogénnych pretínajúcich záležitostiach a odstránenie samotných pretínajúcich záležitostí z kódu. Variabilné vlastnosti je možné konfigurovať pomocou aspektov. Na diskusiu sa javí viacero oblastí medzi vzťahom aspektovo-orientovaného prístupu a objektovo-orientovaným návrhom softvéru (nielen radov softvérových výrobkov). Vytvorený rad softvérových výrobkov so zdrojovými súbormi je voľne dostupný na stiahnutie.

Okrem vyššie spomenutých pravidiel pre tvorbu usmernení, identifikovaní určitých usmernení a evaluácie týchto usmernení, sú v práci predstavené rady softvérových výrobkov, metódy modelovania vlastností i spôsoby notácie použité v týchto modeloch vlastností: Czarnecki-Eiseneckerová notácia a UML notácia. Z modelov vlastností vytvorených pomocou týchto notácií je možné generovať kód pre daný rad softvérových výrobkov. V texte je opísaný Czarneckého objektovo-orientovaný prístup i multiparadigmový návrh s modelovaním vlastností od Vranića.

Výsledky tejto práce môžu byť podľa môjho názoru použité v reálnom prostredí softvérových firiem, ktoré vytvárajú rady softvérových výrobkov a uvažujú nad zefektívnením procesu tvorby týchto radov pomocou aspektovo-orientovaného prístupu.

Použitá literatúra

Apel, Sven a Batory, Don. 2006. When to Use and Aspects? A Case Study. In: *Proceedings of the 5th international conference on Generative programming and component engineering*. ACM. s. 59-68, ISBN: 1-59593-237-2

Apel, Sven. 2007. The role of features and aspects in software development. http://edoc.bibliothek.uni-halle.de/servlets/DocumentServlet?id=4214. (20. 11 2008)

Apel, Sven, Leich, Thomas a Saake, Gunter. 2006. Aspectual Mixin Layers. http://www.infosun.fim.uni-passau.de/cl/publications/docs/TR-0508.pdf. (20. 11 2008)

AspectJ. 2003. The AspectJ Programming Guide. AspectJ Team. http://www.eclipse.org/aspectj/doc/released/progguide/index.html. (30. 04 2008)

Ceccato, Mariano a Tonella, Paolo. 2004. Measuring the Effects of Software Aspectization. In: *Cd-rom Proceedings of the 1st Workshop on Aspect Reverse Engineering (WARE 2004).* Delft, The Netherlands.

Clements, Paul a Northrop, Linda. 2003. Software Product Lines. *Carnegie Mellon Software Engineering Institute.* http://www.sei.cmu.edu/programs/pls/sw-product-lines_05_03.pdf. (30. 4 2008).

Coplien, James O. 2007. Software Patterns. *Hillside.net*. http://hillside.net/patterns/definition.html. (22. 11 2008).

Czarnecki, Krzysztof. 1998. Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models, Dizertačná práca. Ilmenau.

Elrad, Tzilla, Filman, Robert E. a Bader, Atef. 2001. Aspect-oriented programming: Introduction. http://doi.acm.org/10.1145/383845.383853. ISSN:0001-0782. (20. 04 2008)

Figueiredo, Eduardo, Cacho, Nielo a Sant'Anna, Claudio. 2008. Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability. Leipzig. ACM. ISBN 978-1-60558-079-1.

Fowler, Martin. 2006. Writing Software Patterns. http://martinfowler.com/articles/ writingPatterns.html. (11. 22 2008).

Gomaa, Hassan. 2005. Designing Software Product Lines With UML. Boston : Addison-Wesley. ISBN 0-201-77595-6.

Chidamber, Shyam. R. a Kemerer, Chris. F. 1994. A metrics suite for object oriented design. In: *IEEE Transactions on Software Engineering.* 20, s. 476-493.

Kästner, Christian, Apel, Sven a Batory, Don. 2007. A Case Study Implementing Features Using AspectJ. In: *Software Product Line Conference, 2007. SPLC 2007. 11th International*. University of Magdeburg. s. 223-232

Kästner, Christian, Apel, Sven a Kuhlemann, Martin. 2008. Granularity in software product lines. In: *Proceedings of the 30th international conference on Software engineering.* Leipzig: ACM, ISBN 978-1-60558-079-1.

Kvale, Axel Anders, Li, Jingyue a Conradi, Reidar. 2005. A case study on building COTS-based system using aspect-oriented programming. New York: ACM. ISBN 1-58113-964-0.

Lee, Kwanwoo, a kol. 2006. Combining Feature-Oriented Analysis and Aspect-Oriented Programming for Product Line Asset Development. http://doi.ieeecomputersociety.org/10.1109/SPLC.2006.13. (7. 5 2008).

Lopez-Heron, Robert a Batory, Don. 2006. From Crosscutting Concerns to Product Lines: A Function Composition Approach. s.l. University of Texas at Austin.

Lopez-Herrejon, Robert E. 2006. Towards Crosscutting Metrics for Aspect-Based Features. In: *Proceedings of the First Workshop on Aspect-Oriented Product Line Engineering (AOPLE).* Portland, Oregon.

Martin, Robert. 1994. *OO Design Quality Metrics, An Analysis of Dependencies*.

SEI. About Software Product Lines. *Software Enginering Institute*. http://www.sei.cmu.edu/productlines/about_pl.html.(30. 4. 2008).

Stochmialek, Michal. 2005. Aop metrics. http://aopmetrics.tigris.org/metrics.html. (8. 5. 2009).

Vranić, Valentino. 2004. Multi-Paradigm Design with Feature Modeling, *PhD Thesis.* s.l. : Slovenská technická univerzita, Bratislava.

Príloha A Obsah elektronického média

Na priloženom elektronickom médiu sú adresáre a súbory s nasledujúcou štruktúrou:

doc	písomná časť projektu
doc/diplomova_praca.pdf	text diplomovej práce
doc/navrh_clanku.pdf	návrh výsledkov na publikovanie
src	zdrojové súbory štúdie
src/povodny_jes	refaktorovaná aplikácia
src/oop_jes	objektovo-orientovaná verzia radu
src/aop_jes	aspektovo-orientovaná verzia radu
iit.src	dokumenty ku konferencii IIT.SRC
iit.src/iitsrc_prispevok.pdf	prezentovaný príspevok
iit.src/poster.pdf	použitý poster
merania	výsledky meraní
merania/povodny_jes.xls	pôvodný kód
merania/oop_jes.xls	objektovo-orientovaný kód
merania/aop_jes.xls	aspektovo-orientovaný kód
CitajMa.txt	pokyny k dátam na médiu

Príloha B Hodnoty nameraných metrík

V tejto prílohe sú namerané hodnoty metrík, z ktorých sa vychádzalo pri evaluácií usmernení (kapitola 5). Tabuľky č. 10, 11 a 12 zobrazujú namerané hodnoty pre metriky vhodné pre porovnanie nasadenia aspektov, výber a úpravu metrík urobil Stichmialek (2005). Hodnoty metrík pre priečinky sú v tabuľkách 13, 14 a 15. Hodnoty sú namerané pre pôvodnú verziu, objektovo-orientovanú a aspektovo-orientovanú verziu radu Java Email Server.

Tab. 10: Hodnoty AO metrík podľa Stochmialek (2005), pôvodná verzia po refaktorizácií.

Type name	Type	LOCC	WOM	DIT	NOC	CFA	CMC	CBM	CDA	CAE	RFM	LCO
ShutdownService	class	14	1	0	0	0	3	3	0	0	4	0
Mail	class	141	6	0	0	1	6	7	0	0	18	0
ServiceListener	class	83	3	0	0	0	4	4	0	0	10	0
DeliveryService	class	125	11	0	0	0	4	4	0	0	19	21
ConnectionProcessor	interface	4	2	0	0	0	0	0	0	0	0	0
DnsService	class	16	1	0	0	1	3	4	0	0	4	0
Pop3Processor	class	527	17	1	0	0	7	7	0	0	43	54
InvalidAddressException	class	5	1	2	0	0	0	0	0	0	0	0
AuthenticationException	class	5	1	2	0	0	0	0	0	0	0	0
NotFoundException	class	8	2	2	0	0	0	0	0	0	0	0
ConfigManager	class	186	39	0	1	0	4	4	0	0	43	703
PasswordManager	class	35	2	0	0	0	0	0	0	0	2	0
ConfigurationParameterContants	interface	20	0	0	0	0	0	0	0	0	0	0
DefaultSmtpServer	class	33	9	0	0	0	0	0	0	0	9	20
ConfigFileManager	class	273	6	1	0	0	6	6	0	0	26	0
User	class	115	20	0	0	0	6	6	0	0	25	129
Message	class	27	6	0	0	0	2	2	0	0	7	1
EmailAddress	class	78	12	0	0	0	1	1	0	0	9	6
SMTPRemoteSender	class	249	8	0	0	1	11	12	0	0	27	15
SMTPSender	class	255	6	0	0	0	9	9	0	0	39	13
SMTPMessage	class	219	22	0	0	0	4	4	0	0	28	184
SMTPProcessor	class	326	13	0	0	0	8	8	0	0	34	48
Average		**124,73**	**8,5455**	**0,3636**	**0,0455**	**0,1364**	**3,5455**	**3,6818**	**0**	**0**	**15,773**	**54,273**

Tab. 11: Hodnoty AO metrík podľa Stochmialek (2005), objektovo-orientovaná verzia.

Type name	Type kind	LOCC	WOM	DIT	NOC	CFA	CMC	CBM	CDA	CAE	RFM	LCO
ShutdownService	class	14	1	0	0	0	3	3	0	0	4	0
Mail	class	79	4	0	0	0	7	7	0	0	18	0
ServiceListener	class	83	3	0	0	0	4	4	0	0	10	0
DeliveryService	class	125	11	0	0	0	4	4	0	0	19	21
ConnectionProcessor	interface	4	2	0	0	0	0	0	0	0	0	0
DnsService	class	16	1	0	0	1	3	4	0	0	4	0
Pop3Processor	class	527	17	1	0	0	7	7	0	0	43	54
InvalidAddressException	class	5	1	2	0	0	0	0	0	0	0	0
AuthenticationException	class	5	1	2	0	0	0	0	0	0	0	0
NotFoundException	class	8	2	2	0	0	0	0	0	0	0	0
ConfigXMLManager	class	273	6	1	0	0	6	6	0	0	26	0
ConfigManager	class	190	39	0	2	0	6	6	0	0	46	703
OOConfig	class	99	15	0	0	0	0	0	0	0	14	71
PasswordManager	class	45	3	0	0	0	1	1	0	0	5	0
ConfigurationParameterContants	interface	20	0	0	0	0	0	0	0	0	0	0
DefaultSmtpServer	class	33	9	0	0	0	0	0	0	0	9	20
ConfigFileManager	class	273	6	1	0	0	6	6	0	0	26	0
User	class	115	20	0	0	0	6	6	0	0	25	129
Message	class	27	6	0	0	0	2	2	0	0	7	1
EmailAddress	class	79	12	0	0	0	1	1	0	0	9	6
SMTPRemoteSender	class	249	8	0	0	1	11	12	0	0	27	15
SMTPSender	class	255	6	0	0	0	9	9	0	0	39	13
SMTPMessage	class	231	22	0	0	0	8	8	0	0	35	184
SMTPProcessor	class	326	13	0	0	0	8	8	0	0	34	48
SMSSender	class	8	2	0	0	0	1	1	0	0	2	0
SMSBean	class	21	5	0	0	0	0	0	0	0	4	2
ConfigFileWatchDog	class	35	2	1	0	0	3	3	0	0	3	0
EmailAlarm	class	5	1	0	0	0	1	1	0	0	2	0
EncryptEmail	class	5	1	0	0	0	0	0	0	0	1	0

ErrorAlarm	class	8	1	0	0	0	2	2	0	0	4	0
InitializeLogging	class	65	2	0	0	1	2	3	0	0	5	0
LoggingLevel	class	14	2	0	0	0	2	2	0	0	6	0
SMSTrigger	class	24	2	1	0	0	4	4	0	0	3	0
SpamAssassin	class	5	1	0	0	0	0	0	0	0	1	0
Average		**96,206**	**6,6765**	**0,3235**	**0,0588**	**0,0882**	**3,1471**	**3,2353**	**0**	**0**	**12,676**	**37,265**

Tab. 12: Hodnoty AO metrík podľa Stochmialek (2005), aspektovo-orientovaná verzia.

Type name	Type	LOCC	WOM	DIT	NOC	CFA	CMC	CBM	CDA	CAE	RFM	LCO
ShutdownService	class	14	1	0	0	0	3	3	0	0	4	0
Mail	class	74	4	0	0	0	5	5	0	4	19	0
ServiceListener	class	83	3	0	0	0	4	4	0	2	13	0
DeliveryService	class	125	11	0	0	0	4	4	0	1	20	21
ConnectionProcessor	interface	4	2	0	0	0	0	0	0	0	0	0
DnsService	class	16	1	0	0	1	3	4	0	0	4	0
Pop3Processor	class	527	17	1	0	0	7	7	0	2	46	54
InvalidAddressException	class	5	1	2	0	0	0	0	0	0	0	0
AuthenticationException	class	5	1	2	0	0	0	0	0	0	0	0
NotFoundException	class	8	2	2	0	0	0	0	0	0	0	0
ConfigXMLManager	class	273	6	1	0	0	6	6	0	3	30	0
ConfigManager	class	186	39	0	2	0	4	4	0	2	44	703
PasswordManager	class	35	2	0	0	0	0	0	0	0	2	0
ConfigurationParameterContants	interface	20	0	0	0	0	0	0	0	0	0	0
DefaultSmtpServer	class	33	9	0	0	0	0	0	0	0	9	20
ConfigFileManager	class	273	6	1	0	0	6	6	0	3	30	0
User	class	115	20	0	0	0	6	6	0	2	28	129
Message	class	27	6	0	0	0	2	2	0	1	8	1
EmailAddress	class	78	12	0	0	0	1	1	0	0	9	6
SMTPRemoteSender	class	249	8	0	0	1	11	12	0	2	29	15
SMTPSender	class	255	6	0	0	0	9	9	0	3	43	13
SMTPMessage	class	219	22	0	0	0	4	4	0	3	31	184
SMTPProcessor	class	326	13	0	0	0	8	8	0	4	39	48

SMSSender	class	8	2	0	0	0	1	1	0	1	3	0
SMSBean	class	21	5	0	0	0	0	0	0	0	4	2
ConfigFilesWatchDog	aspect	36	3	0	0	0	3	3	1	0	3	0
ConfigFilesWatchDog	class	26	2	1	0	0	1	1	0	1	4	0
ConfigManagerForXML	aspect	8	1	0	0	0	1	1	1	0	0	0
EmailAlarm	aspect	6	1	0	0	0	1	1	2	0	0	0
EncryptEmail	aspect	5	1	0	0	0	0	0	1	0	0	0
ErrorAlarm	aspect	6	1	0	0	0	1	1	10	0	0	0
InitializeLogging	aspect	70	3	0	0	1	2	3	1	2	7	0
LoggingLevel	aspect	20	2	0	0	0	0	0	15	0	0	0
PasswordMandatoryFeature	aspect	6	1	0	0	0	1	1	3	0	0	0
SMSTrigger	aspect	6	1	0	0	0	1	1	1	0	0	0
SpamAssassin	aspect	6	1	0	0	0	0	0	1	0	0	0
Average		**88,167**	**6**	**0,2778**	**0,0556**	**0,0833**	**2,6389**	**2,7222**	**1**	**1**	**11,917**	**33,222**

Tab. 13: Hodnoty metrík pre priečinky podľa Stochmialek (2005), pôvodná verzia radu.

Package name	NOT	A	RMartin Ce	RMartin Ca	RMartin I	RMartin D	Ce	Ca	I	Dn
sk.fiit.dp.mail.server	2,00	0,00	2,00	0,00	1,00	0,00	8,00	0,00	1,00	0,00
sk.fiit.dp.mail.server.services.general	4,00	0,25	3,00	4,00	0,43	0,32	8,00	4,00	0,67	0,08
sk.fiit.dp.mail.server.services.pop3	1,00	0,00	1,00	1,00	0,50	0,50	8,00	1,00	0,89	0,11
sk.fiit.dp.mail.server.errors	3,00	0,00	0,00	6,00	0,00	1,00	0,00	6,00	0,00	1,00
sk.fiit.dp.mail.server.configuration	5,00	0,20	2,00	9,00	0,18	0,62	5,00	9,00	0,36	0,44
sk.fiit.dp.mail.server.info	3,00	0,00	3,00	8,00	0,27	0,73	5,00	8,00	0,38	0,62
sk.fiit.dp.mail.server.services.smtp	4,00	0,00	4,00	1,00	0,80	0,20	16,00	1,00	0,94	0,06
Average	**3,14**	**0,06**	**2,14**	**4,14**	**0,45**	**0,48**	**7,14**	**4,14**	**0,61**	**0,33**

Tab. 14: Hodnoty metrík pre priečinky podľa Stochmialek (2005), objektovo-orientovaná verzia radu.

Package name	NOT	A	RMartin Ce	RMartin Ca	RMartin I	RMartin D	Ce	Ca	I	Dn
sk.fiit.dp.mail.server	2,00	0,00	2,00	1,00	0,67	0,33	9,00	1,00	0,90	0,10
sk.fiit.dp.mail.server.services.general	4,00	0,25	3,00	4,00	0,43	0,32	8,00	4,00	0,67	0,08
sk.fiit.dp.mail.server.services.pop3	1,00	0,00	1,00	1,00	0,50	0,50	8,00	1,00	0,89	0,11
sk.fiit.dp.mail.server.errors	3,00	0,00	0,00	7,00	0,00	1,00	0,00	7,00	0,00	1,00
sk.fiit.dp.mail.server.configuration	7,00	0,14	3,00	14,00	0,18	0,68	5,00	14,00	0,26	0,59
sk.fiit.dp.mail.server.info	3,00	0,00	3,00	9,00	0,25	0,75	5,00	9,00	0,36	0,64
sk.fiit.dp.mail.server.services.smtp	4,00	0,00	4,00	1,00	0,80	0,20	20,00	1,00	0,95	0,05
sk.fiit.dp.mail.server.services.sms	2,00	0,00	0,00	3,00	0,00	1,00	0,00	3,00	0,00	1,00
sk.fiit.dp.mail.server.oo	8,00	0,00	6,00	2,00	0,75	0,25	6,00	2,00	0,75	0,25
Average	**3,78**	**0,04**	**2,44**	**4,67**	**0,40**	**0,56**	**6,78**	**4,67**	**0,53**	**0,43**

Tab. 15: Hodnoty metrík pre priečinky podľa Stochmialek (2005), aspektovo-orientovaná verzia radu.

Package name	NOT	A	RMartin Ce	RMartin Ca	RMartin I	RMartin D	Ce	Ca	I	Dn
sk.fiit.dp.mail.server.aspects	11,00	0,00	11,00	0,00	1,00	0,00	17,00	0,00	1,00	0,00
sk.fiit.dp.mail.server	2,00	0,00	2,00	1,00	0,67	0,33	7,00	1,00	0,88	0,13
sk.fiit.dp.mail.server.services.general	4,00	0,25	3,00	4,00	0,43	0,32	8,00	4,00	0,67	0,08
sk.fiit.dp.mail.server.services.pop3	1,00	0,00	1,00	1,00	0,50	0,50	8,00	1,00	0,89	0,11
sk.fiit.dp.mail.server.errors	3,00	0,00	0,00	7,00	0,00	1,00	0,00	7,00	0,00	1,00
sk.fiit.dp.mail.server.configuration	6,00	0,17	3,00	12,00	0,20	0,63	5,00	12,00	0,29	0,54
sk.fiit.dp.mail.server.info	3,00	0,00	3,00	9,00	0,25	0,75	5,00	9,00	0,36	0,64
sk.fiit.dp.mail.server.services.smtp	4,00	0,00	4,00	1,00	0,80	0,20	16,00	1,00	0,94	0,06
sk.fiit.dp.mail.server.services.sms	2,00	0,00	0,00	3,00	0,00	1,00	0,00	3,00	0,00	1,00
Average	**4,00**	**0,05**	**3,00**	**4,22**	**0,43**	**0,53**	**7,33**	**4,22**	**0,56**	**0,40**

Príloha C Konfigurácie vytvoreného radu softvérových výrobkov

Konfiguračný súbor pre konfiguráciu vlastností v rade Java Email Server pomocou aspektov.

```
############################################################
# Configuration file for using features is SPL
# Author: Jan Kohut
#
# File name (aspect) without comment (#) will be used in SPL
############################################################

#src/sk/fiit/dp/mail/server/optional/Logging.aj
#src/sk/fiit/dp/mail/server/optional/AspectLogger.aj
#src/sk/fiit/dp/mail/server/optional/ConfigFilesWatchDog.aj
#src/sk/fiit/dp/mail/server/optional/ConfigManagerForXML.aj
#src/sk/fiit/dp/mail/server/optional/EmailAlarm.aj
#src/sk/fiit/dp/mail/server/optional/EncryptEmail.aj
#src/sk/fiit/dp/mail/server/optional/ErrorAlarm.aj
#src/sk/fiit/dp/mail/server/optional/LoggingLevel.aj
#src/sk/fiit/dp/mail/server/optional/PasswordMandatoryFeature.aj
#src/sk/fiit/dp/mail/server/optional/SMSTrigger.aj
#src/sk/fiit/dp/mail/server/optional/SpamAssassin.aj

########## Variant A ###################################
#src/sk/fiit/dp/mail/server/optional/Logging.aj

########## Variant B ###################################
src/sk/fiit/dp/mail/server/optional/Logging.aj
src/sk/fiit/dp/mail/server/optional/ConfigManagerForXML.aj
src/sk/fiit/dp/mail/server/optional/ErrorAlarm.aj
src/sk/fiit/dp/mail/server/optional/LoggingLevel.aj

########## Variant C ###################################
#src/sk/fiit/dp/mail/server/optional/Logging.aj
#src/sk/fiit/dp/mail/server/optional/ConfigFilesWatchDog.aj
#src/sk/fiit/dp/mail/server/optional/EmailAlarm.aj
#src/sk/fiit/dp/mail/server/optional/EncryptEmail.aj
#src/sk/fiit/dp/mail/server/optional/ErrorAlarm.aj
#src/sk/fiit/dp/mail/server/optional/LoggingLevel.aj

########## Variant D ###################################
#src/sk/fiit/dp/mail/server/optional/Logging.aj
#src/sk/fiit/dp/mail/server/optional/ConfigManagerForXML.aj
#src/sk/fiit/dp/mail/server/optional/EmailAlarm.aj
#src/sk/fiit/dp/mail/server/optional/EncryptEmail.aj
#src/sk/fiit/dp/mail/server/optional/ErrorAlarm.aj
#src/sk/fiit/dp/mail/server/optional/LoggingLevel.aj
#src/sk/fiit/dp/mail/server/optional/PasswordMandatoryFeature.aj
#src/sk/fiit/dp/mail/server/optional/SMSTrigger.aj
#src/sk/fiit/dp/mail/server/optional/SpamAssassin.aj
```

Konfiguračný súbor pre konfiguráciu vlastností v rade Java Email Server, objektovo-orientovaná verzia.

```
#########################################################
# Configuration file for using features is SPL
# Author: Jan Kohut
#
# Feature with set value on true will be used in SPL.
# Other values will be ignored.
#########################################################

#spamAssassinEnabled = true
#encryptEmailEnabled = true
#emailAlarmEnabled = true
#errorHandlerEnabled = true
#sMSTrigerEnabled = true
#initializeLog4jEnabled = true
#log4jLevelEnabled = true
#xmlConfigFileEnabled = true
#configFileWatchDogEnabled = true
#anotherEncryptMethodForPasswordEnabled = true

########## Variant A ####################################
#initializeLog4jEnabled = true

########## Variant B ####################################
#initializeLog4jEnabled = true
#xmlConfigFileEnabled = true
#errorHandlerEnabled = true
#log4jLevelEnabled = true

########## Variant C ####################################
#initializeLog4jEnabled = true
#configFileWatchDogEnabled = true
#emailAlarmEnabled = true
#encryptEmailEnabled = true
#errorHandlerEnabled = true
#log4jLevelEnabled = true

########## Variant D ####################################
initializeLog4jEnabled = true
xmlConfigFileEnabled = true
emailAlarmEnabled = true
encryptEmailEnabled = true
errorHandlerEnabled = true
log4jLevelEnabled = true
anotherEncryptMethodForPasswordEnabled = true
sMSTrigerEnabled = true
spamAssassinEnabled = true
```

Príloha D Rozšírený abstrakt na konferencii IIT.SRC 2009

V tejto prílohe je publikovaný rozšírený abstrakt, ktorý bol prezentovaný na študentskej vedeckej konferencii *IIT.SRC 2009 Informatics and Information Technologies Student Research Conference*. Príspevok s názvom *Guidelines for Using Aspects in Software Product Lines* bol v sekcii *Softvérové inžinierstvo*. Príspevok bol vytvorený pod vedením Dr. Valentina Vranića.

Guidelines for Using Aspects in Software Product Lines

Ján KOHUT*

Slovak University of Technology
Faculty of Informatics and Information Technologies
Ilkovičova 3, 842 16 Bratislava, Slovakia
kohut04@student.fiit.stuba.sk

1 Extended abstract

Software product lines (SPLs) represent a common and important technology to support the derivation of a wide range of applications [2]. They have a core of functionality common for all application instances. A part of functionality is configured according to the application context most often given by customer requirements.

Design, development, and configuration of software product lines is not a trivial problem. Problems with configuration can occur as a consequence of using variable features, which can be used in software product lines, the requests on software can be antithetic, and features can be interlacing. Classical, object-oriented approach can have crosscutting concerns that lead to a problem of code tangling and scattering.

Aspect-oriented programming (AOP) can be helpful for solving these problems in software product lines mainly in feature configuration. Aspect-oriented programming cannot be considered as the best solution always. Suitability of using aspects in software product lines is possible to express by recommendations, i.e. guidelines. This is the main idea of this paper.

There are many approaches how to define guidelines, rules and patterns. In this paper the form is defined as: Context, Problem, Forces, Solution, Example and Discussion.

Context describes the situation in which it is convenient to consider using the guideline. *Problem* defines problems that can occur if the guideline is not be applied. In the *Forces* part, reasons are analyzed why the current state is not suitable. *Solution* describes the process of resolving this situation. *Example* contains an example

* Master degree study programme: Software Engineering
Supervisor: Valentino Vranić, Institute of Informatics and Software Engineering, Faculty of Informatics and Information Technologies STU in Bratislava

IIT.SRC 2009, Bratislava, April 29, 2009, pp. 1–2.

implementation or its part. In *Discussion*, the solution is discussed with several views of the guideline presented.

The following guidelines have been identified:

- *The aspects are unsuitable for implementing features of refactored legacy applications*. Feature-refactoring legacy applications is a difficult problem, because such applications do not imply that their design was amenable to feature-extensibility [3].
- *Do not use aspects in mandatory features if there are no crosscutting concerns*. Aspects flatten inherent object-oriented structure of collaboration, obscure the intent of the programmer, and the result is a program that is difficult to read [1].
- *The aspects are suitable for reduction of replicated code with homogenous crosscutting concerns*. Aspects reduce replicated code in code with homogenous crosscutting concerns.
- *Do not use aspects in transforming a mandatory feature into two or more alternative features*. Transformation a mandatory feature into two or more alternatives features AspectJ adds and change more components, lines of code as another approaches because all aspects rely on the points of intersection provided by the core [1].
- *Use aspects in features which share no code and which have crosscutting concerns*. Aspect-oriented solution present low value and superior stability in terms of tangling and scattering over components [2].

Acknowledgement: This work was partially supported by the Institute of Informatics and Software Engineering, Faculty of Informatics and Information Technologies, Slovak University of Technology in Bratislava.

References

[1] Apel, S., and Batory, D.: When to Use Features and Aspects? A Case Study. In: *Proceedings of the 5th international conference on Generative programming and component engineering*, ACM Press, (2006), pp. 59-68.

[2] Figueiredo, E., Cacho, N., and Sant'Anna, C.: Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability. In: *Proceedings of the 30th international conference on Software engineering*, ACM Press, (2008), pp. 261–270.

[3] Fowler, M.: Writing Software Patterns, (2006), [Online; accessed November 28th 2008.], Available at http://martinfowler.com/articles/writingPatterns.html.

Príloha E Príprava na publikovanie výsledkov

Táto príloha obsahuje návrh príspevku na publikovanie dosiahnutých výsledkov na niektorých zahraničných konferenciách a v odborných časopisoch.

Guidelines for Using Aspects in Software Product Lines

Ján Kohut

Slovak University of Technology
Faculty of Informatics and Information Technologies
Ilkovičova 3, 842 16 Bratislava, Slovakia
kohut04@student.fiit.stuba.sk

Abstract. Software product lines are an important approach to software development and maintenance. Design, development, and configuration of software product lines are not a trivial problem. Aspect-oriented programming can often help develop software product lines effectively, but it is not always so. In this paper, is presented an approach for creating guidelines for applying aspect-oriented programming in software product lines is presented. There is shown example of one guideline. To investigate whether identified guidelines are useful was performed a case study. Results of this study are presented there.

1 Introduction

Software product lines (SPLs) represent a common and important technology to support the derivation of a wide range of applications [6]. They have a core of functionality common for all application instances. A part of functionality is configured according to the application context most often given by customer requirements.

Design, development, and configuration of software product lines are not trivial problems. Problems with configuration can occur as a consequence of using variable features, which can be used in software product lines, the requests on software can be antithetic, and features can be interlacing. Classical, object-oriented approach can have crosscutting concerns that lead to a problem of code tangling and scattering.

Aspect-oriented programming (AOP) can be helpful for solving these problems in software product lines mainly in feature configuration. Aspect-oriented programming cannot be considered as the best solution always. Suitability of using aspects in software product lines is possible to express by recommendations, i.e. guidelines. This is the main idea of this paper.

The rest of the paper is organized as follows. Section 2 presents the form of guidelines and guidelines that have been identified so far. Section 3 describes one of the guidelines named *Aspects are suitable for reduction of replicated code with homogenous crosscutting concerns*. Evaluation of guidelines is in Section 4. Conclusion is presented in Section 5.

2 Guidelines

There are a number of publications about integrating aspect-oriented programming with software product lines and configuring features in them. These publications point out advantages and disadvantages of using aspects in software product lines from which it is possible to derive guidelines that can help developer to decide whether to use aspects in software product lines and when it is better to use another approach. The aim of these guidelines is to direct a developer towards an effective application of aspects in software product line development.

2.1 The Guideline Form

There are many approaches how to define guidelines, rules and patterns. The form of a guideline in this paper is inspired by Coplien's form of pattern description [5]. Parts of Coplien's solutions were selected and adapted because his form provides space for problem, concept, solution, and relationship between modules. In this paper, the form is defined as:

- Context
- Problem
- Forces
- Solution
- Example
- Discussion

Context describes the situation in which it is convenient to consider using the guideline. *Problem* defines problems that can occur if the guideline is not be applied. In the *Forces* part, reasons are analyzed why the current state is not suitable. *Solution* describes the process of resolving this situation. *Example* contains an example implementation or its part. In *Discussion,* the solution is discussed with several views of the guideline presented.

It is possible to define a guideline at various levels of abstraction of a particular system. Also, guidelines are not strict rules, but just recommendations because the problem of configuration of features in software product is a very complex one.

Each guideline must be consistent with other guidelines. It is possible to notice a conflicting recommendation at first sight, but from the content, solution, and discussion there should be a way how to combine these conflicting guidelines.

When applying a guideline, it is necessary to consider how the software product line has been developed (incrementally, revolutionary…), what experience software

company has with aspect-oriented programming development, and to what extent it is ready to go into risk of developing software with a new approach.

The guidelines reported in this paper need not cover all cases. But as Martin Fowler said [7]: "It's more valuable to have a bunch of good patterns, poorly organized than it is to have a really good structure with weak patterns underneath them."

2.2 Identified Guidelines

The following guidelines have been identified:

- *The aspects are unsuitable for implementing features of refactored legacy applications*. Feature-refactoring legacy applications is a difficult problem, because such applications do not imply that their design was amenable to feature-extensibility [7].
- *Do not use aspects in mandatory features if there are no crosscutting concerns*. Aspects flatten inherent object-oriented structure of collaboration, obscure the intent of the programmer, and the result is a program that is difficult to read [1].
- *The aspects are suitable for reduction of replicated code with homogenous crosscutting concerns*. Aspects reduce replicated code in code with homogenous crosscutting concerns.
- *Do not use aspects in transforming a mandatory feature into two or more alternative features*. By transforming a mandatory feature into two or more alternative features AspectJ adds and changes more components and lines of code compared to non-aspect-oriented approaches because all aspects rely on the join points provided by the core [6].
- *Use aspects in features which share no code and which have crosscutting concerns*. Aspect-oriented solution presents low value and superior stability in terms of tangling and scattering over components [6].

3 The Aspects are Suitable for Reduction of Replicated Code with Homogenous Crosscutting Concerns

In this chapter is example of one presented guideline. It says about suitability of using aspects for reduction of replicated code with homogenous crosscutting concerns. Homogenous crosscutting concerns refine multiple join points with a single piece of advice. Heterogeneous crosscutting concerns, in contrast, refine multiple join points each with a different a piece of advice [1]. Homogeneity lies in a consistent application of the same or very similar policy in multiple places [9].

Aspects reduced replicated code when implementing homogenous crosscutting concerns [1].

3.1 Context

Often, there are concerns repeated in code of software product lines.

3.2 Problem

In homogenous crosscutting concerns there is often repeated code. The change of these code requires change of code at every place, where is these code.

3.3 Forces

Our effort is to reduce a number of lines of repeated code, improve readability of code, and place the functionality into one module. Then we can get a feature into one module and enable its configuration in software product lines.

3.4 Solution

Using aspects in homogenous crosscutting concerns will enable to factor them out into separate and easily configurable modules.

3.5 Example

Examples of homogenous crosscutting concerns are logging, tracing, exception handling, etc. The code shows an example of crosscutting concerns. Aspect has a set of methods using one coherent advice:

```
public aspect Homogenous {
  pointcut accessAutorization(): call ...;
  pointcut accessDemilitaryZone(): call ...;
  pointcut accessCommunicationInterfaces(): call ...;
  pointcut accessApplication(): call ...;

  before ():accessAutorization() ||
              accessDemilitaryZone() ||
              accessCommunicationInterfaces() ||
              accessApplication() {
        System.out.println("Access to ...");
  }
}
```

3.6 Discussion

Homogenous crosscutting concerns have a single advice for multiple join points. For multiple methods that need multiple join points only one advice is needed. Other approaches are not so elegant because they require repeating almost identical pieces of code in every affected method. This solution can bring code replication and problems connected with it [2].

The aspects influence class collaboration by intertype declarations and advices. This can lead to misunderstanding object-oriented architecture and structure of classes. Subsequently, the program becomes difficult to read and understand.

4 Evaluation

Evaluation of the approach was performed on application Java Email Server.[1] Java Email Server is a Java implementation of the SMTP and POP3 e-mail server. The original application has 4500 line of code, 22 classes, and several configuration files. To evaluate the approach proposed in this paper, a software product line has been created out of this application . Figure 1 shows the feature model of the resulting product line.

4.1 Features

Mandatory features create the core of the application and they are included in all instances of the software product line. In Java Email Server are mandatory modules, which are for work with passwords, users, emails, POP3 and SMTP, logging tool Log4j.

Alternative features in created software product line is functionality for reading configuration parameters from various formats of files. There were created two kinds of reading these files. Class *ConfigFileManager* reads *.properties files, class *ConfigXMLManager* reads parameters from XML files.

In Java Email server are implemented these optional features: *ConfigFileWatchDog, SMS, SMSTrigger, SpamAssassin, EncryptEmail, EmailAlarm, ErrorAlarm, InitializeLog4j, Log4jLevel*.

[1] http://www.ericdaugherty.com/java/mailserver/

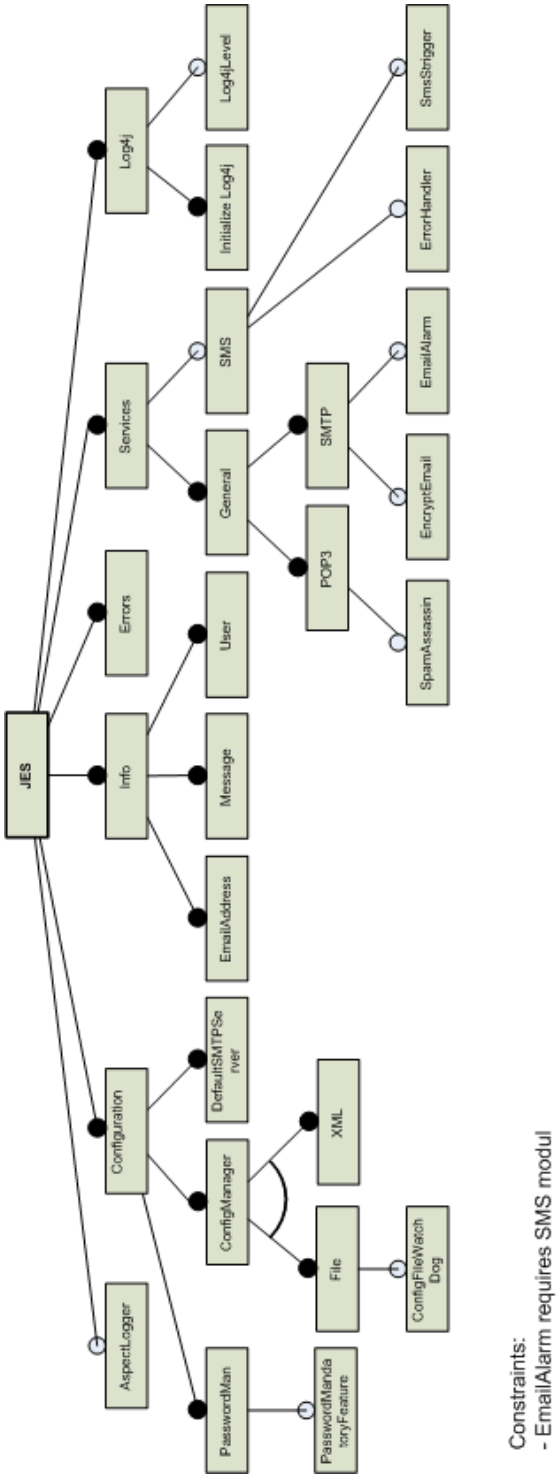


Figure 1: Java Email Server software product line.

4.2 Metrics Applied

To evaluate the application of the proposed approach to the Java Email Server product line, various kinds of metrics have been employed. One group of metrics was targeted at the size of changes:

- Lines of Code (LOC)—expresses size of modules. It is important for comparing the size of modules and labor consumption.
- Number of Affected Classes (NAC)—expresses number of classes affected by the implementation of a feature.

Another group of metrics that have been applied are suitable for both object-oriented and aspect-oriented programming [3,4,11]. Module is used as a common term for classes and aspects:

- Weighted Operations in Module (WOM)
- Depth of Inheritance Tree (DIT)
- Number of Children (NOC)
- Crosscutting Degree of an Aspect (CDA)
- Coupling of Method Call (CMC)
- Coupling of Field Access (CFA)
- Coupling between Modules (CBM)
- Response for a Module (RFM)
- Lack of Cohesion in Operations (LCO)

The metrics in the last group express package dependencies [10,11]:

- Number of Types (NOT)
- Abstractness (A)
- Afferent Couplings (Ca)
- Efferent Couplings (Ce)
- Modified Efferent Couplings (Ce)
- Instability (I)

4.3 Performing Evaluation

Two independent software product line implementations were created out of Java Email Server. In the aspect-oriented implementation, features are configured using the AspectJ language. Every feature is implemented in its own file. Features are weaved into the software product line by compilation. The object-oriented implementation is configured by a configuration file. During start of application is this configuration loading. Features are asserted during executing of program by *if clause* as shown in the code fragment below:[2]

[2] Source code is available for download at http://lapis.yweb.sk/DP/jes.zip.

```
if(OOConfig.getInstance().isInitializeLog4jEnabled()) {
        InitializeLogging.initializeLogging(directory);
}
```

4.4 Using Aspects on Mandatory Features without Crosscutting Concerns

Using aspects to implement mandatory features without crosscutting concerns enclosed object-oriented structure of program. The number of LOC used to implement the *PasswordMandatoryFeature* application feature is similar (11 and 9 LOC). In Table 1 some characteristics of *PasswordMandatoryFeature* are shown.

Table 1: Metrics on PasswordMandatory feature.

	LOC	WOM	DIT	NOC	CFA	CMC	CBM	CDA	CAE	RFM	LCO
OO	45	3	0	0	0	1	1	0	0	5	0
AO	35	2	0	0	0	0	0	0	0	2	0

4.5 Aspects Reduce Replicated Code

Homogenous crosscutting concerns are one of the most appropriate areas for using aspects. One from this crosscuting concerns is calling logging tools Log4J. In the Java Email Server application are the same or similar pieces of code appear frequently. Table 2 shows the number of calls to logging methods.

Table 2: Numbers of calling logging functions.

Name of method	Number of calls
log.isDebugEnable()	27
log.debug()	47
log.isInfoEnable()	10
log.info	26

This is the version without aspects:

```
if( log.isDebugEnabled() ){
  log.debug( "Loading SMTP Message " + messageFile.getName()
  );
}
```

and this is the version with aspects applied:

```
log.debug( "Loading SMTP Message " + messageFile.getName()
);
```

The *LoggingLevel* aspect reduced LOC saving 27 (potentially 47 times) calls of the *isDebugEnable()*method. Similarly, moving the *isInfoEnable()* method into the aspect reduced calls of this method 10 times (similarly 26 times).

4.6 Changing Mandatory Feature to Alternative Features

In Java Email Server, the change of a mandatory feature into two or more alternative features was realized. The change was realized by object-oriented refactoring. Refactoring was required to enable later use of some features. After this change, it became simple to create and configure alternative features with aspects. Adding further alternative features is from LOC comparable in aspects and object-oriented approaches, see Table 3.

Table 3: Used changes on adding alternative feature ConfigXMLManager into existing program infrastructure.

	Change	
	AO	**OO**
Class, Aspects	Creating new aspect	Change class code
LOC	11	3

4.7 Using Aspects on Crosscutting Concerns

Adding new features with object-oriented approach added into classes crosscutting concerns; *if* clause. Despite of it, aspects do not cause code tangling and crosscutting, because aspects do not add any code into original code. This method enabled adding new features, size of code has been growing, but code shows similar level of code coupling; see tables 4 and 5. In these tables were made measurements on code without implemented homogenous crosscutting concerns (*LoggingLevel* and *ErrorAlarm* features) for the reason of showing another characteristics as benefits of using aspects on homogenous crosscutting concerns.

Table 4: Comparing of metric values original version (after refactorization of reading configuration, object-oriented and aspect-oriented version of software product line Java Email Server.

	LOC	WOM	DIT	NOC	CFA	CMC	CBM	CDA	CAE	RFM	LCO
Pôv.	124,73	8,55	0,36	0,05	0,14	3,55	3,68	0,00	0,00	15,77	54,27
OO	96,21	6,68	0,32	0,06	0,09	3,15	3,24	0,00	0,00	12,68	37,26
AO	88,17	6,00	0,28	0,06	0,08	2,64	2,72	1,00	1,00	11,92	33,22

Table 5: Values of metrics for packages[11].

	NOT	A	RMartin Ce	RMartin Ca	RMartin I	RMartin D	Ce	Ca	I	Dn
Orig.	3,14	0,06	2,14	4,14	0,45	0,48	7,14	4,14	0,61	0,33
OO	3,78	0,04	2,44	4,67	0,40	0,56	6,78	4,67	0,53	0,43
AO	4,00	0,05	3,00	4,22	0,43	0,53	7,33	4,22	0,56	0,40

4.8 Using aspects on features which no share code

On figure 2 is shown comparison coupling of method call between object-oriented and aspect-oriented version of software product line. Aspect-oriented version shows lower value coupling of method call for some classes. Then this approach has better modularity and reusability of classes.

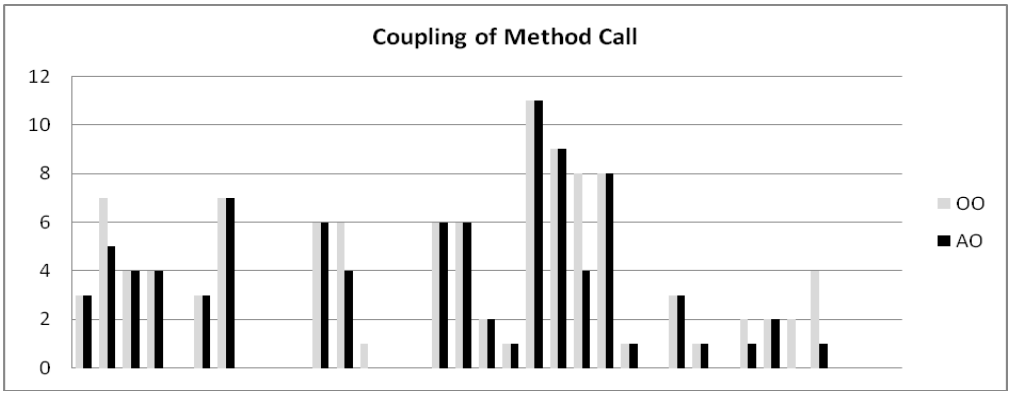


Figure 2: Coupling of method call for object-oriented and aspect oriented implementation.

5 Related works

6 Conclusions

In this paper an aspect-oriented approach to developing software product lines based on guidelines along has been proposed. An example of a complete guideline has been presented. The aim of these guidelines is to help a developer to create effectively software product lines with aspects. To evaluate usefulness of the identified guidelines, a case study has been performed comparing the object-oriented and aspect-oriented version of the same product line implementation. The results of this study are presented in the paper. After evaluation of identified guidelines is possible to say: *Is possible to express suitability of using aspects in software product lines by guidelines*.

References

[1] Apel, S., and Batory, D.: When to Use Features and Aspects? A Case Study. In: *Proceedings of the 5th international conference on Generative programming and component engineering*, ACM Press, (2006), pp. 59-68.

[2] Apel, S., Leich, T., and Saake, G.: Aspectual Mixin Layers. In: *Proceedings of the 28th international conference on Software engineering*, ACM Press, (2006), pp. 122-131.

[3] Ceccato, M., Tonella, P.: Measuring the Effects of Software Aspectization In: *Cd-rom Proceedings of the 1st Workshop on Aspect Reverse Engineering (WARE 2004)*. Delft, The Netherlands, 2004.
[4] Chidamber, S. R., Kemerer, C. F.: A metrics suite for object oriented design In: *IEEE Transactions on Software Engineering*. 1994, 20, s. 476-493.
[5] Coplien, J., O.: *Software Patterns*. http://hillside.net/patterns/definition.html
[6] Figueiredo, E., Cacho, N., and Sant'Anna, C.: Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability. In: *Proceedings of the 30th international conference on Software engineering*, ACM Press, (2008), pp. 261–270.
[7] Fowler, M.: Writing Software Patterns, (2006), [Online; accessed November 28th 2008.], Available at http://martinfowler.com/articles/writingPatterns.html.
[8] Kästner, Ch., Apel, S., Batory D.: A Case Study Implementing Features Using AspectJ. In: *Proceedings of the 11th International Software Product Line Conference*, ACM Press, (2006), pp. 223-232.
[9] Kvale, A.A., Li, J., Conradi, R.: A Case Study on Building COTS-Based System Using Aspect-Oriented Programming, *Symposium of Applied Computing*, ACM Press, (2005), pp. 1491-1498.
[10] Martin, R.; OO Design Quality Metrics, An Analysis of Dependencies. 1994.
[11] Stochmialek, M.: AOP Metrics. http://aopmetrics.tigris.org/metrics.html