Slovenská technická univerzita v Bratislave
FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ
Študijný program: Softvérové inžinierstvo

Bc. Radoslav Menkyna

Zachytenie interakcie zmien implementovaných aspektmi modelom vlastností

Diplomová práca

Vedúci diplomového projektu: Ing. Valentino Vranić, PhD.
máj 2009

Slovak University of Technology Bratislava
FACULTY OF INFORMATICS AND INFORMATION TECHNOLOGIES
Study program: Software engineering

Bc. Radoslav Menkyna

Capturing Interaction of Changes Implemented by Aspects Using Feature Modeling

Master's Thesis

Supervisor: Ing. Valentino Vranić, PhD.
May 2009

Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií

ZADANIE DIPLOMOVEJ PRÁCE

Meno študenta: **Bc. Radoslav Menkyna**
Študijný odbor: SOFTVÉROVÉ INŽINIERSTVO
Študijný program: Softvérové inžinierstvo
Názov projektu: **Zachytenie interakcie zmien implementovaných aspektmi modelom vlastností**

Zadanie:

Implementácia zmien je dôležitou súčasťou riadenia zmien. Niektoré zmeny majú charakter pretínajúcich záležitostí, t.j. týkajú sa viacerých, inak nesúvisiacich záležitostí. Bolo preukázané, že aspektovo-orientované programovanie sa dá úspešne využiť pri implementácii takýchto zmien. Tento prístup je zvlášť výhodný pri existencii viacerých verzií systému prispôsobených rozdielnym kontextom, čo je príznačné pre rady softvérových výrobkov. Analyzujte interakciu zmien implementovaných aspektmi. Navrhnite spôsob modelovania a identifikácie interakcie zmien založený na modelovaní vlastností, technike ktorá sa používa v radoch softvérových výrobkov na zachytenie konfigurovateľnosti. Prístup demonštrujte na príkladoch.

Odporúčaná literatúra:

M. Bebjak, V. Vranić, and P. Dolog. Evolution of Web Applications with Aspect-Oriented Design Patterns. In Marco Brambilla and Emilia Mendes, editors, Proc. of ICWE 2007 Workshops, 2nd International Workshop on Adaptation and Evolution in Web Systems Engineering, AEWSE 2007, in conjunction with 7th International Conference on Web Engineering, ICWE 2007, July 19, 2007, Como, Italy.

P. Dolog, V. Vranić, and M. Bieliková. Representing Change by Aspect. ACM SIGPLAN Notices, 36(12), December 2001.

Valentino Vranić. Multi-paradigm design with feature modeling. Computer Science and Information Systems Journal (ComSIS), 2(1): 79-102, 2005.

S. O. Rashid, R. Chitchyan, A. Rashid, and R. Khatchadourian. Approach for Change Impact Analysis of Aspectual Reequirements. AOSD-Europe Deliverable D110, Technical Report, AOSD-Europe-ULANC-40, March 2008. http://www.aosd-europe.net/

Práca musí obsahovať:

Anotáciu v slovenskom a anglickom jazyku
Analýzu problému
Opis riešenia
Zhodnotenie
Technickú dokumentáciu
Zoznam použitej literatúry
Výstupy celého diplomového projektu vrátane vlastnej diplomovej práce a vytvoreného softvéru (zdrojového kódu s dokumentáciou)

Miesto vypracovania: Ústav informatiky a softvérového inžinierstva, FIIT STU, Bratislava
Vedúci projektu: Ing. Valentino Vranić PhD.

Termín odovzdania práce v letnom semestri: dňa 13. mája 2009

SLOVENSKÁ TECHNICKÁ UNIVERZITA Fakulta informatiky a informačných technológií BRATISLAVA

[illegible]

Bratislava, dňa 16. februára 2009

prof. Ing. Pavol Návrat, PhD.
riaditeľ ÚISI

LICENČNÁ ZMLUVA O POUŽITÍ ŠKOLSKÉHO DIELA

uzatvorená
podľa § 40 a nasl. zákona č. 618/2003 Z. z. o autorskom práve a právach súvisiacich s autorským právom (autorský zákon) v znení neskorších zmien a doplnení a § 51 školské dielo
medzi

Autorom:

meno a priezvisko: **Menkyna Radoslav, Bc.**

ID študenta: 20896 Dátum a miesto narodenia: 29. 06. 1984, Žilina

Trvalý pobyt: B. S. Timravy 3 , 01008 Žilina

Študent fakulty : **Fakulta informatiky a informačných technológií STU, Ilkovičova 3, 842 16 Bratislava**
Stupeň štúdia[1]: ❶ ❷ ❸

Názov študijného programu: Softvérové inžinierstvo

Názov študijného odboru: Softvérové inžinierstvo

a

nadobúdateľom:

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE (ďalej STU)
Vazovova 5, 812 43 Bratislava

Zastúpená dekanom fakulty: **prof. RNDr. Ľudovít Molnár, DrSc.**

Osoba oprávnená konať: prof. Ing. Pavol Návrat, PhD.

Čl. I
Predmet zmluvy

Predmetom tejto zmluvy je udelenie súhlasu autora školského diela (ďalej aj dielo) špecifikovaného v čl. II tejto zmluvy nadobúdateľovi na použitie školského diela (ďalej len „licencia") podľa podmienok dohodnutých v tejto zmluve.

Čl. II
Určenie školského diela

1. Autor udeľuje nadobúdateľovi licenciu k tomuto školskému dielu[1]:

- ☐ **bakalárska práca**
- ☑ **diplomová práca**
- ☐ **dizertačná práca**
- ☐ **iná práca**, špecifikovaná ako

s názvom **Zachytenie interakcie zmien implementovaných aspektmi modelom vlastností**

Zadanie práce je prílohou č. 1 tejto licenčnej zmluvy.

2. Školské dielo podľa odseku 1. bolo vytvorené jeho autorom – študentom STU, ktorá je nadobúdateľom licencie podľa tejto zmluvy, na splnennie študijné povinnosti autora vyplývajúce z jeho právneho vzťahu k nadobúdateľovi v súlade so zákonom č. 131/2002 Z. z. o vysokých školách a o zmene a doplnení niektorých zákonov v znení neskorších predpisov.

[1] vyznačte

3. Školské dielo podľa odseku 1 sa rozumie ako výsledný celok pozostávajúci z jednej alebo viacerých súčastí. Súčasťou sa rozumie textová časť publikovaná v papierovej a elektronickej podobe, softvér, hardvér, audiovizuálny záznam alebo akýkoľvek i podporný materiál, prostriedok pre vytvorenie školského diela.
4. Študent touto zmluvou dáva súhlas na zverejnenie svojho diela v zmysle § 17 ods. 1 písm. c/ Autorského zákona.
5. Prevzatím diela nadobúdateľom sa nadobúdateľ stáva oprávnený používať dielo v rozsahu a spôsobom uvedených v tejto zmluve.

Čl. III

Spôsob použitia školského diela a rozsah licencie

1. Autor udeľuje nadobúdateľovi súhlas na vyhotovenie digitálnej rozmnoženiny školského diela za účelom uchovávania a bibliografickej registrácie školského diela v súlade s § 8, ods. 2, písm. b) zákona č. 183/2000 Z. z. o knižniciach, o doplnení zákona Slovenskej národnej rady č. 27/1987 Zb. O štátnej pamiatkovej starostlivosti a o zmene a doplnení zákona č. 68/1997 Z. z. o Matici slovenskej v znení neskorších predpisov.
2. Autor udeľuje nadobúdateľovi licenciu na sprístupňovanie vyhotovenej digitálnej rozmnoženiny školského diela online prostredníctvom internetu bez obmedzenia, vrátane práva poskytnúť sublicenciu tretej osobe na študijné, vedecké, vzdelávacie a informačné účely.
3. Nadobúdateľ je oprávnený udeliť tretej osobe súhlas na použitie diela v rozsahu udelenej licencie.
4. Autor udeľuje nadobúdateľovi súhlas na použitie diela alebo jeho časti najmä na: vyhotovenie rozmnoženiny diela, verejné rozširovanie originálu diela alebo jeho rozmnoženiny alebo zaradenie diela do súborného diela.
5. Nadobúdateľ nie je oprávnený upravovať či inak meniť dielo či názov diela. Nadobúdateľ je oprávnený použiť dielo v súlade s jeho určením a za podmienok stanovených v tejto zmluve.
6. Licencia udelená autorom nadobúdateľovi podľa tejto zmluvy je nevýhradná, nie je dotknuté právo autora použiť dielo spôsobom, na ktorý nevýhradnú licenciu udelil a takisto nie je dotknuté právo autora udeliť licenciu tretej osobe pri rešpektovaní nároku autora podľa čl. III ods. 7 a 8 zmluvy s tým, že autor nesmie udeliť licenciu inému subjektu na taký účel, ktorý by bol v rozpore s oprávnenými záujmami školy.
7. Nadobúdateľ je oprávnený požadovať, aby mu autor diela zo získanej odmeny súvisiacej s použitím diela primerane prispel na úhradu nákladov vynaložených na vytvorenie diela, a to podľa okolností až do ich skutočnej výšky.
8. Autor udeľuje nadobúdateľovi licenciu na dobu trvania majetkových práv.
9. Nadobúdateľ môže školské dielo zverejniť a šíriť aj pod svojim menom.

Čl. IV

Odmena

1. Autor udeľuje nadobúdateľovi licenciu bezodplatne.

Čl. V

Pôvodnosť školského diela

1. Autor prehlasuje, že samostatnou vlastnou tvorivou činnosťou vytvoril školské dielo špecifikované v čl. II a že toto školské dielo je pôvodné.
2. Autor vyhlasuje, že pred uzavretím tejto licenčnej zmluvy neposkytol k dielu licenciu poskytovanú touto zmluvou žiadnej tretej osobe, a to ani výhradnú ani nevýhradnú.
3. Autor sa zaručuje, že všetky exempláre originálu školského diela špecifikovaného v čl. II bez ohľadu na nosič majú totožný obsah.

Čl. VI

Záverečné ustanovenia

1. Táto zmluva je vyhotovená v dvoch rovnopisoch, z ktorých po jednom vyhotovení obdržia autor a nadobúdateľ. Táto zmluva sa môže meniť alebo dopĺňať len písomným dodatkom podpísaným oboma zmluvnými stranami.
2. Táto zmluva nadobúda platnosť a účinnosť dňom jej podpisu zmluvnými stranami.
3. Na vzťahy, ktoré nie sú výslovne upravené touto zmluvou sa vzťahujú všeobecne záväzné právne predpisy platné a účinné na území Slovenskej republiky, najmä ustanovenia Autorského zákona a Občianskeho zákonníka.
4. Zmluvné strany vyhlasujú, že zmluvu uzavreli slobodne a vážne, nekonali v omyle ani v tiesni, jej obsahu porozumeli a na znak súhlasu ju vlastnoručne podpísali.

v Bratislave dňa 16. februára 2009

[illegible] [illegible]

....

autor nadobúdateľ

ANOTÁCIA

Slovenská technická univerzita v Bratislave
FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ
Študijný program: Softvérové Inžinierstvo

Autor: Radoslav Menkyna

Diplomová práca: Zachytenie interakcie zmien implementovaných aspektmi modelom vlastností

Vedúci diplomovej práce: Ing. Valentino Vranić, PhD.

máj 2009

Táto práca sa zaoberá interakciami zmien implementovaných pomocou aspektov. Uvádza novú techniku, ktorá na rozoznanie a analýzu interakcií používa modelovanie vlastností. Priama interakcia je analyzovaná na modeli vlastností zmien. Práca popisuje konštrukciu čiastočného modelu vlastností systému, ktorý slúži na analýzu ďalších, nepriamych, interakcií. Pre vyhodnotenie interakcie zmien je dôležité poznať spôsob ich implementácie. Tento je možné zistiť pomocou doménovo špecifických katalógov zmien alebo aplikovaním multi paradigmového návrhu s modelovaním vlastností. Aplikácia tohto prístupu zahŕňala rozšírenie domény riešenia jazyka AspectJ o nové priamo použiteľné paradigmy a zmenu procesu transformačnej analýzy. Využitím tohto postupu je možné identifikovať prípadné interakcie na rôznych mierach abstrakcie.

ANNOTATION

Slovak University of Technology Bratislava
FACULTY OF INFORMATICS AND INFORMATION TECHNOLOGIES
Degree course: Software engineering

Author: Radoslav Menkyna

Thesis: Capturing Interaction of Changes Implemented by Aspects Using Feature Modeling

Supervisor: Ing. Valentino Vranić, PhD.

2009, May

This thesis studies the interactions of changes implemented by aspects. A new technique which uses the feature modeling to trace the interactions is proposed. Direct interactions are analyzed on a feature model of changes. Thesis describes the construction of a partial feature model of given system, which can be used to study additional interactions present in the system. To evaluate found interactions it is needed to know how they will be implemented. This can be revealed trough the domain specific change catalogs or by applying multi-paradigm design with feature modeling. The use of this approach involved AspectJ solution domain extension and transformational analysis modification. Using approach proposed in the thesis it was possible to analyze interactions on different levels of abstraction.

Ďakujem vedúcemu mojej diplomovej práce Valentinovi Vranićovi za jeho podporu, časté diskusie a cenné pripomienky.

Čestne prehlasujem, že som diplomovú prácu vypracoval samostatne.

Contents

Chapter 1

Introduction

Aspect-oriented paradigm was designed to capture crosscutting concerns and express them in a modular way. An idea was proposed that a change should be represented using an aspect [DVB01]. In such approach a logical change can be expressed modularly and easily plugged to the system. Moreover, in case that a change is no longer required in the system it can be easily unplugged.

When a large number of changes represented by aspects is present in the system undesired interactions and problems can occur. This thesis will study interactions and discuss problems coupled with representing changes by aspects. A technique which uses feature modeling to represent changes implemented by aspects was proposed to analyze the interactions. This technique has several stages that can help to discover possible interactions of changes represented by aspects. First, interactions are analyzed on the highest level of abstraction on feature model of changes. In the next stage indirect or unforeseen interactions can be revealed. Construction of partial feature model of the underlying system is characteristic for this stage.

To evaluate the possible interactions of changes it is needed to know how they will be implemented. Implementation details can be gained trough the domain specific catalogs. This is not always possible. Another possibility is to employ the multi-paradigm design with feature modeling to get details of the implementation. To use such approach new direct usable paradigms which represent known change realizations were added to AspectJ solution domain. After modified transformation analysis process possible interactions of changes can be evaluated.

The rest of the document is organized as follows: Chapter 2 introduces the change versioning and places the used approach into the existing versioning models. Chapter 3 presents an overview of two level change realization model and possible change realization techniques. Some techniques were modified and two new techniques were proposed. In chapter 4 possible problems coupled with representing changes by aspects are discussed and their solutions are proposed. Chapter 5 introduces the technique which uses feature models of changes to find their possible interactions. Creation of partial feature model of the system which reveals additional possible interaction is described. Chapter 6 proposes an approach in which changes are considered paradigms in terms of multi-paradigm design. Transformational analysis can be than used to choose between several change realizations. Chapter 7 concludes presented work.

Chapter 2

Change Versioning

This chapter will describe the main aspects of version control in general and connect this approach with the known versioning models. The main goal of this work is not to introduce a new versioning model, but to study interactions of changes at the implementation level, focusing on approach which uses the aspects to represent changes. However, the principle of change versioning has to be understood first.

There are many approaches to the version control. One of the main difference between numerous version control models is how they define a version. Models that define a version as a state of an evolving system are called *state-based*. These models use revisions and variants to describe a state of particular system. On the other hand *change-based* models define a version as set of changes applied to a baseline. Each change represents some logical action and may carry some additional attributes which represent the nature of the change [CW98].

Another classification of the version models is derived from the way how versions are constructed. *Extensional versioning* models define a version by enumerating its members. All versions are explicit. User of system based on this model usually checks out some version (revision), performs changes on this version, and submits it as a new version. Another approach called *intensional versioning* defines a version set using a predicate. The predicate represents some condition that must be satisfied by all members of the version. In such model versions are implicit and many combinations may be created on demand [CW98].

The rest of the chapter is organized as follows: Section 2.1 summarizes concept of change based versioning. Section 2.2 describes an approach in which change is represented by aspect. Section 2.3 describes version model for aspect dependency management.

2.1 Change Based Versioning

In change based extensional versioning version set is defined explicitly by enumerating its members. Member (version) is defined as application of some changes relative to some baseline. This type of versioning is known also under term *change packets* model [CW98].

In change based intensional versioning new version is constructed as a set of freely combined changes according to actual requirements. A change is defined as partial function transferring one potential version to another. Version is then constructed by applying a sequence of changes to a baseline. This form of versioning is also known as *change set* model [CW98].

A version space (change space in the change based versioning) can be represented as a matrix or grid where lines and columns represent versions and changes (fig. 2.1a). Each change can be included or omitted in any version. Fig. 2.1b illustrates a tree based representation. These two representations explicitly name the changes included in particular version. This is a improvement compared with state based models in which the changes are not explicitly named. By merging changes, these must be deduced from graph topology, which may become difficult [CW98].

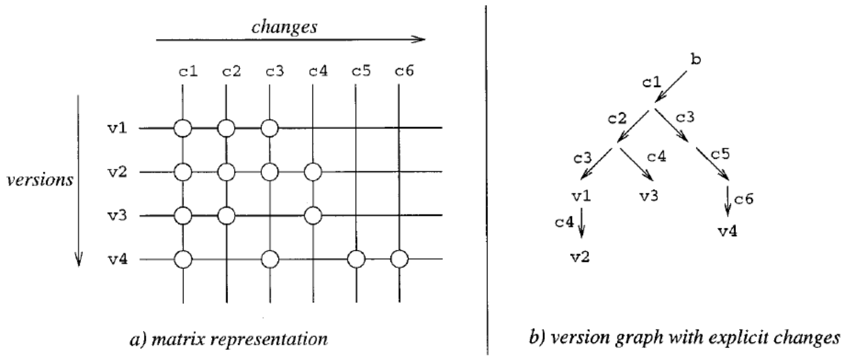


Figure 2.1: Change space. Adopted from [CW98].

2.2 Aspect-Oriented Approach to Change-Based Versioning

Aspect-oriented approach to change based versioning was presented by Dolog, P. et al. [DVB01]. Presented approach suggests that a change could be represented as an aspect. Aspect-oriented paradigm can offer new benefits to the change based model of versioning.

Main goal of aspect-oriented paradigm is a separation of crosscutting concerns. There are many approaches to this paradigm. In this work it will be described trough approach represented by AspectJ language [PARb], because its large community acceptance. To achieve the separation of concerns new language constructs were created . *Pointcut* express a certain point or set of points in control flow of a application. Upon pointcuts actions defined in *advices* can be performed. Pointcuts and advices are defined in class-like entity called *aspect* [PARa].

Using aspects to represent changes can have several benefits.

- **Localization** All the modifications coupled with a particular logical change are centralized in an aspect.
- **Modularity** The entire change logic is also represented in the aspect.
- **Pluggability** Because target is not aware of the change, the change can be easily plugged or unplugged from the system.

These benefits are important for change based versioning. For example pluggability is crucial attribute for change based intensional versioning, where changes are composed freely to construct a version.

2.3 Version Model for Aspect Dependency Management

In large systems use of aspect oriented paradigm can lead to large number of aspects to be composed into resulting system. Among these aspects various dependences can arise. These dependences can be hard to trace and lead to unexpected behavior of target system. To cope with this problem a version model for aspect dependency management was presented [PSC01].

This model describes a dependencies of grater scale and uses various granularity levels. At lowest granularity level it describes structure of aspects and on higher level dependence among them. In this model a version is a defined as a set of conditions, while condition expressed as a boolean expression. Allowed operations are conjunction, disjunction and negation e.g. $V1 = C1 \wedge (C2 \vee C3)$. At a higher level sets of aspects can be joined to model a valid system configuration $V5 = V1 \wedge V2 \wedge !V3)$[PSC01]. This work will study the interactions of implemented aspects which will represent such modeled configurations.

Chapter 3

Aspect-Oriented Change Realization Techniques

There are various techniques how a change can be represented in the aspect-oriented paradigm [Beb07, DVB01, BVD07].[1] Usually a aspect-oriented design pattern or idiom can be used to represent a change. This chapter will summarize the most important techniques which can be used to implement a particular change. All mentioned techniques are described generally which means they can be used to implement changes in the several domains. Two additional techniques were presented: Introducing Role to Class technique (Section 3.2) and Introducing Regions technique (Section 3.3). These techniques are based on aspect-oriented design patterns.

The rest of the chapter is organized as follows: Section 3.1 describes two level change realization concept. Sections 3.2 - 3.9 represent actual change realization techniques. Each technique is separated in own section.

3.1 General and Specific Changes

In this chapter techniques how to implement a changes will be described. These techniques can be considered as general change types. General change type can be altered to many specific changes which implement various change requests. They are usually based on an aspect-oriented design pattern or idiom and provide an implementation scheme. When a change request is associated with the particular general change type, its realization becomes clear.

It would be useful if the developer could get hint which general change type to use for his specific change request. This could be achieved by maintaining a catalog of changes in which each domain specific change type would

[1]This chapter is partially based on adapted text from paper Developing Applications with Aspect-Oriented Change Realization (Appendix F) to which my contribution is approximately 10 % .

be defined as a specialization of one or more generally applicable changes. Such catalogs began to emerge [BVD07]

To determine a change type to be applied, developer chooses a particular change request, identifies individual changes in it, and determines their type. Such approach is depicted in Figure 3.1. It was possible to identify domain specific changes D1 and D2 in the Change Request 1. From the previously identified and cataloged relationships between change types, it is possible to identify their generally applicable change types are G1 and G2 [VBMD08].

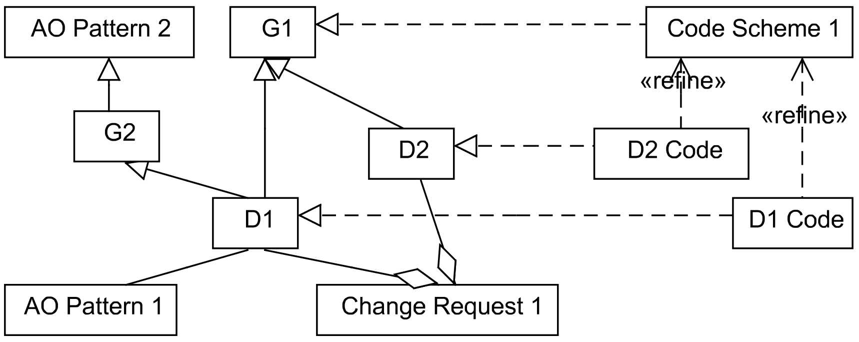


Figure 3.1: Generally applicable and domain specific changes. Adopted from [VBMD08].

A generally applicable change type is usually some aspect-oriented design pattern (consider G2 and AO Pattern 2). A domain specific change realization can also be complemented by an aspect-oriented design patterns, which is expressed by an association between them (consider D1 and AO Pattern 1) [VBMD08].

Every generally applicable change has a known domain independent implementation scheme (G2's implementation scheme was omitted from the figure). This implementation scheme has to be adapted to the context of a particular domain specific change, which may be seen as a kind of refinement (consider D1 Code and D2 Code) [VBMD08].

Building catalogs which provide described mappings is in some situations unacceptable. The problem of selecting a suitable realizing change type can be solved using multi-paradigm design [Vra05]. This will be further discussed in chapter 6.

3.2 Introducing Role To Class

Using the aspect-oriented design pattern Director it is possible to introduce one or several roles to any number of classes. This means it is possible to introduce some additional behavior to participating classes. The code of the pattern is separated in the aspects and no code is attached to the participating classes. This provides such benefits as modularity, reusability and pluggability. Using this pattern several object-oriented design patterns can be implemented [Men08].

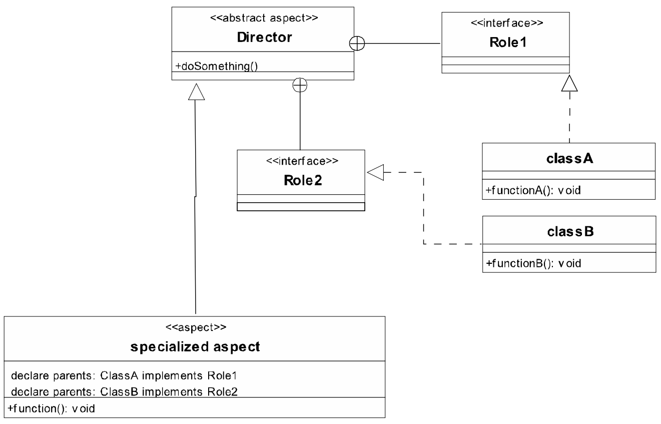


Figure 3.2: Scheme of the Director design pattern adapted from [Mil04].

Two aspects are used to implement the Director design pattern. The first aspect is abstract and it should specify the roles. This is done using Java interfaces. It also specifies any generic logic needed to support the roles. The second aspect introduces the roles to the specific classes. It can also provide any method definitions which are required and are not part of the original classes [Mil04].

3.3 Introducing Regions

This technique uses Border Control design pattern [Mil04] to define some set of regions to a system. Regions represent reasonable parts of the application and can be reused by other aspects. This ensures the aspects are used only in a correct scope. This approach is valuable also when additional changes are expected. Changing a region definition in the Border Control then affects all the aspects reusing this region which enables the region control in one location. This change type is usually not used directly but its occurrence is invoked by other change types.

Pattern uses single aspect in which regions are represented as pointcuts. Following code is a template of Border Control design pattern adapted from [Mil04]:

```
public aspect MyRegionSeparator {
        public pointcut myTypes1(): within(mypackage1.+);
        public pointcut myTypes2(): within(mypackage2.+);
        public pointcut myTypes(): myTypes1() || myTypes2();
        public pointcut myMainMethod()
```

```
                    : withincode(public void mypackage2.MyClass.main(..));
. . .
}
```

3.4 Class Exchange

Class exchange [Beb07] is a technique that uses aspect-oriented design pattern Cuckoo's Egg [Mil04]. Using the pattern it is possible to change the type of the object being instantiated on a constructor call. New object must be a subtype of the original object class, otherwise a class cast exception will be thrown on the first attempt to instantiate the original class.

Here is simple template adapted from [Mil04]:

```
public aspect ClassExchange{ // Cuckoo's Egg Aspect
        public pointcut originalClassConstructorCall( ) : <call pointcut>

        Object around( ) : originalClassConstructorCall( ){
                return new DesiredClass( );
        }
}
```

The pattern uses an aspect in which pointcut specifies join points of the object constructor calls. The advice then changes the object being created or performs some control logic over it. The pointcut can also obtain any arguments supplied to the original constructor call [Mil04]. The logic included in the advice can for example decide which object should be instantiated, return the original object or modify its arguments [Beb07].

3.5 Method Substitution

Method Substitution [Beb07] is technique similar to the Class Exchange mentioned in previous section. It is used to change, alter or even disable execution of a particular method. Technique uses an aspect in which pointcut specifies a method call and the around advice performs desired logic. Additional parameters required by advice logic (target class of method call or method arguments) can be captured by pointcut specifying the method call. Sometimes it is sufficient only to alter the method arguments, in this case it is possible to use the `proceed()` construct with the altered arguments in the advice body. This is a method substitution template adapted from [Beb07]:

```
public aspect MethodSubstition {
pointcut methodCallsPointcut(TargetClass t, int a):
        call(ReturnType TargetClass.method(int a)) && target(t) && args(a);

ReturnType around(TargetClass t, int a): methodCallsPointcut(t, a) {
        if (. . .) {
                . . .; // the new method logic
```

```
            } else {
                        proceed(t, a);
                    }
            }
      }
```

3.6 Member Introduction

Using AspectJ's static crosscutting it is possible to introduce any field or method to any class present in the system. Static crosscutting also enables to declare a class to implement some interfaces or a inheritance relationship. Such changes to class present in the system can be needed when adding a new functionality to the system by another aspect. This way existing classes can be altered and provide the additional support for new functions without affecting their code. This simple template will demonstrate the Member Introduction:

```
public aspect MemberIntroduction {
 Modifiers Type TargetClass.NewMember; \\ field introduction
 Modifiers ReturnType TargetClass.new(Arguments){ Body } \\ method introduction

 declare parents : TargetClass implements SomeInterface;
        \\ declaring a class to implemet an interface
 declare parents : TargetClass extends SomeClass;
        \\ declaring a class to extend another class
```

Static crosscutting can be also used to declare the compilation time errors, warnings and soft exceptions. Soft exceptions provide simple solution in case a code which throws a checked exception is added to some class by an aspect. This class is not aware that an exception can be thrown by the code added by the aspect so soft exception should be introduced.

3.7 Additional Parameter Checking

Additional parameter checking [Beb07] is a similar technique as the Method Substitution (Section 3.5). This technique also specifies a pointcut that defines some method call and captures required context. Around advice than checks the captured arguments and can perform suitable action, for example proceed with the method execution with modified arguments, or throw an exception which can be later handled.

3.8 Additional Return Value Checking/Modification

Sometimes it is desired to perform some additional actions or checks on method return values. This can be achieved using Additional Return Value

Checking/Modification technique [Beb07]. This technique is using around advice to obtain the return value of method represented by method call pointcut. Obtained value can be later checked or processed. Following code adopted from [Beb07] represents use of this technique.

```
public aspect AdditionalReturnValueProcessing {
        pointcut methodCallsPointcut(TargetClass t, int a): . . .;
        private ReturnType retValue;

        ReturnType around(): methodCallsPointcut(/**captured arguments **/) {
                retValue = proceed();
                processOutput(/* captured arguments */);
return retValue;
}
```

3.9 Performing Action After Event

Performing action after event [Beb07] is a simple but frequent form of the change realization. It is used when additional actions are needed after the specific event. The event is specified by pointcut, for example method or constructor execution/call or field modification. An after advice on specified pointcut is then used to perform the desired actions.

```
public aspect PerformingActionAfterEvent {
pointcut SomeEvent(arguments): . . .;

after(/captured arguments /): SomeEvent(/Captured arguments /) {
        performAction(/captured arguments /);
}

private void performAction(/arguments /) {
        // action logic
        }
}
```

Chapter 4

Solving Change Perplexity

This chapter will describe possible problems that can arise when changes are represented by aspects in the system. With the growing number of changes grows also the possibility of interaction between changes present in the system. The interaction can have negative effects and can lead to an unexpected behavior of the system. There are also other problems coupled with a large number of changes represented by aspects present in the system. These problems and their possible solutions will be discussed chapter.

The rest of the chapter is organized as follows: Section 4.1 proposes solution when change of existing change is needed. Section 4.2 describes situations in which order of aspect execution is important. Interaction can be, to some extent, seen also between general change types (Section 4.3). Some changes can rise by system evolution (Section 4.4). Section 4.5 describes logical error localization and section 4.6 outlines tool support for representing changes as aspects.

4.1 Changing a Change

In this work a change is considered as an specific logical change request. This means every change has a specific purpose that can be described and later added as metadata to the aspect which represents the change. In other words change improves or adds functionality to the system, it brings some additional value to the system. Moreover, some of the small changes such as the bug fixes cannot be represented by aspects. Main reason for this is that destinations of such small changes may be too specific to be represented by pointcuts because particular destinations in the code can not be identified with join points [Faz06]. Such destinations are for example the right side of the assignment or the cycle parameters. Thus, changes which represent only bug corrections or trivial modifications should not be represented by the aspects and applied directly to a baseline.

In case a change of the already applied change represented by aspect is needed, two approaches are possible. It is possible to implement a change of the applied change by a new change or modify the existing change in the system [Beb07].

In this case same rules mentioned at the beginning of this section should apply. If the change represents some bug fix it should be applied directly to the old change. On the other hand, if this change has some logical meaning, for example, it introduces some alternate function, but the original function may be also needed, it should be implemented as a new change and represented by the new aspect.

Bug fixes applied directly to a baseline or directly to aspect representing a logical change may be tracked by a secondary versioning system. Such versioning system may be text based. In future a unified versioning system could be created that would use versioning method according to type of change.

4.2 Order of Aspect Execution

In a large system it is possible that two independent change representations will perform an action upon the same join point. This is not a problem when one change uses the before advice and another the after advice. Problems may occur when changes represented by aspects use the same type of the advice or combination before/around, around/after advice is used. In this case it is unpredictable in which order the advices will be executed. This can cause an unwanted system behavior. Only way how to solve this problem is to explicitly specify the order in which the advices will be executed. This can be achieved by declaring precedence among the aspects with the following construct.

```
declare precedence : Pattern1, Pattern2, ..;
```

This construct explicitly declares that an aspect or aspects represented by Pattern1 have higher precedence then aspects represented by Pattern2. Use of wildcards is patterns is allowed. Only concrete aspects are considered by the matching algorithm. Precedence of the advices in one aspect is defined by lexical order in which the advices are defined.

With precedence set, order of advice execution is certain. The aspect with the higher precedence:

- executes its before advice before the one with the lower precedence
- executes its after advice after the one with the lower precedence.
- encloses the around advice in the lower-precedence aspect with his around advice.

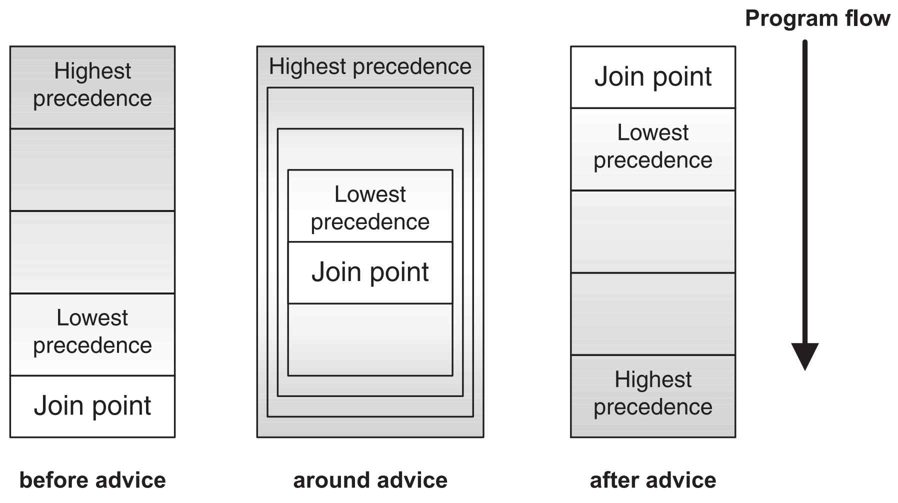


Figure 4.1: Illustration of precedence rules. Adopted from [Lad03].

These rules are depicted at the figure 4.1.

Along with adding a change to a system other changes in the system should be examined. If two or more changes are connected to the same join points a precedence of aspects, which represent the changes, should be declared. This could seem as a great drawback when a possible large number of changes in the system is considered, but development environments provide partial support for this problem. They automatically show which aspects are advising some joint point.

4.3 Interaction Between General Change Types

General changes (Section 3.1) use in most cases the aspect-oriented design patterns to achieve their purpose (Section 3). Subsequent interrelated application of the aspect-oriented design patterns to a particular problem can require additional changes to design patterns already present in the system [Men07]. Thus, by combining general changes additional changes can be required, which can be seen as an unwanted interaction. This section will try to discuss which general change combinations are problematic.

Aspect-oriented design patterns were classified according their structure into the pointcut, advice and inter-type declaration design patterns. Additional change of existing patterns is required when a pointcut design pattern is combined with an advice or inter-type declaration pattern [Men07]. Pointcut patterns are Wormhole pattern, Participant pattern and Border Control pattern.

So far only Border Control pattern, from pointcut pattern category, appears between the general change types (Section 3). It is represented by the Introducing Regions technique. General change types Introducing Role to Class (Section 3.2) and Class Exchange (Section 3.4) use advice design patterns to archive their purpose. The Member Introduction (Section 3.6) uses similar method as the Policy design pattern which is an inter-type declaration design pattern. Method Substitution (Section 3.5) uses the same principle as the Cuckoo's egg pattern which is an advice design pattern. The remaining change types do not use explicitly an aspect-oriented design pattern, but their structure is similar to the advice design patterns. Therefore, all the remaining change types can be considered to belong to the advice design pattern category. To sum up all change types except Introducing Regions belong to the advice or inter-type design pattern category.

Mentioned implies that combining the Introducing Regions change type with any other change types could cause unwanted interaction or in other words additional modification of an already applied change types. This is true only if specific changes represented by the general change types are related, otherwise no interaction occurs. Also the direction of the general change types application is important. If a design pattern from the pointcut category is applied first to the system, it can be combined with design patterns from any other category without changes [Men07]. This implies that if the Introducing Regions change type is used before the other change types no interaction occurs regardless to its relation with the other change types.

The list of general change types from section 3 is not complete. New change types can arise. For these new change types will be important to check if they are not from the pointcut design patterns category, which can mean an unwanted interaction.

4.4 Changes Invoked by System Evolution

Considering a system with already applied changes, a problems may occur when the base system implementation evolves [Bre08]. For example, consider some method names are changed, parameters added or omitted. This action could cause that the join point represented by this method is no longer captured by certain pointcuts.

This problem can be considered only as minor, because it can be solved trough development environments. Even today development environments for some aspect-oriented languages are able to help mitigate the risk such problems to minimum. Examples of the development environment functions for AspectJ:

- Every line containing a join point which is advised is highlighted with the small icon. So when the line is modified one can notice that line contains a join point that is advised. List of all the aspects which are connected with this join point can be seen in special perspective window.

- When an advice does not advices any join point developer will get a compilation warning.
- For every advice the list of advised join points can be opened. This is very useful because we can see explicitly to which join points the specific advice is connected.

Developers using aspect-oriented languages with less sophisticated development support should pay more attention to the mentioned problems, until the more sophisticated environments become available.

4.5 Logical Error Localization

In a large systems with many changes already applied to a baseline the problem with a localization of the logical error can occur [Bre08]. Consider a accounting class which uses some algorithm to create an index representing customers credibility. This algorithm is later modified by the changes represented by aspects. If testing proves that a credibility index has a wrong value, it is hard to distinguish where the logical error occurred.

Problem of logical error localization exists in all systems. In this case in order to find the error, the baseline code and related change code must be reviewed. This action is also supported by development environments. By checking baseline code the advised join points are highlighted so the changes related to these joint points can be immediately checked.

4.6 Representing Changes by Aspects—Tool Support

As mentioned in the sections 4.4, 4.5 development environments provide support for the aspect-oriented paradigm. This support varies according to used environment and also used language. Support is designed for any aspects and it is based on lexical rules. Such kind of support enables for example highlighting advised join points. While this kind of support is very useful, for aspects representing a logical change additional support may be desirable.

In the large systems complex dependences between aspects representing changes can arise. Since every change has its logical context it may be desirable to express this context in unified way. This means every aspect representing a change would contain some metadata, which could be collected and summarized as some perspective or view. Metadata could contain a description of the change represented by the aspect and also dependences to other changes represented by aspect. Such dependences would contain information about the aspect precedence or explain the logical dependence between changes (consider a change witch has meaning only if another change is present in the system). Metadata could be expressed for example in form of annotations. The perspective constructed from such metadata will provide better overview of all changes present in the system, capturing dependences among them.

Chapter 5

Interaction of Changes

This chapter will describe a technique which enables to discover possible interactions of changes represented by aspects.[1] This technique is based on the feature modeling. Prior to representing changes by the feature modeling, possibility of transforming dependency graphs into the feature models was examined. Dependency graphs can capture dependencies between the concerns in a system. The concept of partial feature model of a system, which is crucial for discovering possible interactions, will be proposed.

First, feature modeling (Section 5.1) and dependency graphs (Section 5.2) are introduced. Section 5.3 describes how to represent dependency graphs by the feature models. Also changes can be represented by the feature model (Section 5.4) on which the direct dependencies can be captured (Section 5.5). The last section 5.6 proposes a technique how to capture indirect dependences. This technique uses proposed partial feature model of a system.

5.1 Feature Modeling

Feature modeling is a conceptual domain modeling technique. It is used to express concepts by their features taking into account feature interdependencies and variability in order to capture the concept configurability [Vra05]. A domain represents an area of interest. Usually application domain and solution domain are expressed by feature modeling.

Feature modeling can be used to capture commonality and variability in the software product lines. From this point of view feature modeling is the process of identifying externally visible characteristics of products [LKL02]. An interaction between the features of products in software product lines

[1]This chapter is partially based on adapted text from my paper Aspect-Oriented Change Realizations and Their Interaction (Appendix E) to which my contribution is approximately 35 % .

can occur. Some features may require presence of other features or features are considered alternative.

Output from feature modeling process is feature model which consists of feature diagrams, constraints, default dependency rules and information associated with concepts and features. Feature diagram is a directed tree whose root represents concept and all other nodes represent concept features [Vra04b]. Feature diagrams can visually capture properties of a feature or dependencies between several features. For example features may be depicted as mandatory or optional, two features may be depicted as alternative or or-features. Additional dependencies can be captured as constraints of feature diagrams trough the predicate logic.

Changes implemented as aspects are modular and pluggable. They can be considered as the features of the system which can be included or excluded from the final configuration of the system. Therefore, feature modeling approach seams suitable for modeling changes implemented as aspects.

Feature modeling used for modeling changes implemented by aspects is based on the Feature modeling for multi-paradigm design [Vra04a] which is based on widely accepted and simple Czarnecki–Eisenecker basic notation [CE00].[2] Using feature modeling it is possible to express additional informations about the changes such as:

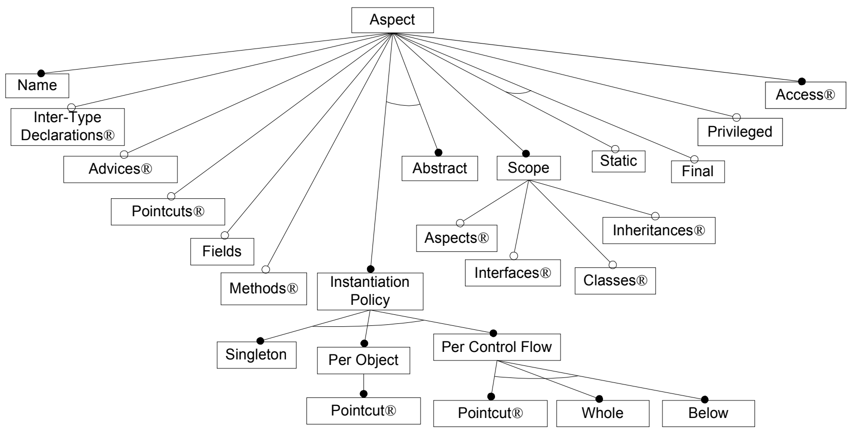
- relation of a change to the concept—mandatory, optional
- relations between the changes on the same level of the feature diagram—alternative changes, or changes
- relations between the changes on multiple levels of the feature diagram—change of change, change only applicable when other change is present in the system.
- openness for a new changes

Notation will be explained on the example of aspect paradigm from AspectJ solution domain feature model (Figure 5.1). This paradigm will be widely referenced in next chapter.

As a matter of simple fact, each aspect is named, which is modeled by a mandatory feature *Name* (indicated by a filled circle ended edge). The aspect paradigm articulates related structure and behavior that crosscuts otherwise possibly unrelated types. This is modeled by optional features *Inter-Type Declarations* ®, *Advices* ®, and *Pointcuts* ® (indicated by empty circle ended edges). These features are references to equally named auxiliary concepts that represent plural forms of respective concepts that actually represent paradigms in their own right (and their own feature models [Vra05]).

[2]This extension to the base feature modeling technique was chosen because it will be used also in chapter 6 where multi-paradigm design with feature modeling is applied on changes represented by aspects.

To achieve its intent, an aspect may—similarly to a class—employ *Methods* Ⓡ, whereas the method is yet another paradigm, and *Fields*.



final $\veebar$ *abstract*

Figure 5.1: The AspectJ aspect paradigm (adopted from [Vra05]).

An aspect in AspectJ is instantiated automatically by occurrence of the join points it addresses in accordance with *Instantiation Policy* which is either *Singleton*, *Per Object*, or *Per Control Flow*. The features that represent different instantiation policies are mandatory alternative features (indicated by an arc over mandatory features), which means that exactly one of them must selected. An aspect can be *Abstract*, in which case it can't be instantiated, so it can't have *Instantiation Policy* either, which is again modeled by mandatory alternative features.

An aspect can be declared to be *Static* or *Final*. It doesn't have to be none of these two, but it can't both either, which is modeled by optional alternative features of which only one may be selected (indicated by an arc over optional features). An aspect can also be *Privileged* over other aspects and it has its type of *Access* Ⓡ, which is modeled as another reference to a separately expressed auxiliary concept. The constraint associated with the aspect paradigm feature diagram means that the aspect is either *Final*, or *Abstract*. All the features in the Aspect paradigm are bound at source time.

5.2 Dependency Graphs

In addition to study interactions of changes implemented by aspects it is needed to capture a scope of the system in which the interactions are most likely to occur. When applying a change in traditional fashion the proposed change can interfere with many entities from a existing source code. A change realized as an aspect modularizes the essence of proposed change but still affects the system in one or several points specified by the aspect pointcuts. In both cases it is needed to identify a part of the system where proposed change can affect the existing entities. An approach of program slicing [Wei81, GL91] can be used to achieve this goal.

A program slice narrows a behavior of the program to specified subset of interest. Concern, on the other hand, represents a part of the system behavior from larger scale and higher complexity. Usually several slices that represent an elementary behavior can be grouped together to represent a concern. Analyzing these concerns and slices can lead to better understanding of dependencies among changes and their impact on system [RCRK08]. In the next section a technique that visualizes known dependencies among concerns and their slices will be described.

Dependency graphs help to visualize the change propagation and dependencies between concerns and their slices [RCRK08]. The dependency graphs are constructed from semi-formal dependency equations. Each concern can consist of temporal(S_T), conditional (S_C), business rule(S_B) and task oriented slices(S_{TO}). In the dependency graph concern slices are represented by nodes and dependencies by edges. There are three types of dependencies which can be captured between the concern slices by the dependency equations and graphs . A forward dependency means a concern slice links or results to another concern slice. A backward dependency means the concern slice uses or bases itself on previous slice. Parallel dependency expresses the concern slices occur concurrently [RCRK08].

Figure 5.2 shows a simple dependency graph that captures dependencies in case study of an tollgate system [RCRK08]. In this system owner of the vehicle first registers with the bank and activates a gizmo trough ATM. The toll is then automatically deducted when the payed motorway is used. The forward dependency is represented by arrow, the backward dependency by dashed arrow and the parallel dependency by two way arrow. Numbers in the graph express assigned weights of dependencies during change propagation evaluation. Considered concern slice is depicted as dash-dotted.

Figure shows that considered register business rule concern slice $\mathit{Register}_B$ is forward dependent on read and store gizmo information conditional concern slice $\mathit{ReadStoreInfo}_C$, which is forward dependent on debit concern's business rule and conditional slice $\mathit{Debit}_{B \wedge C}$. Correctness and compatibility concern slices $\mathit{Correctness}_C$ and $\mathit{Compatibility}_B$ are parallel dependent on debit concern slices. Calculate concern slices $\mathit{Calculate}_{B \wedge C}$ and read store gizmo information concern slice $\mathit{ReadStoreInfo}_C$ are backward dependent on debit concern slices [RCRK08].

One can notice that the read store gizmo information concern slice occurs twice in dependency graph. This suggests that dependency graphs capture also behavior. This issue will be addressed in the next section.

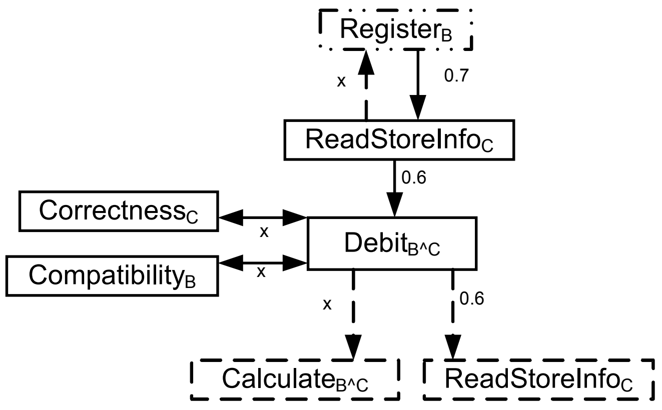


Figure 5.2: Dependency graph for *register* concern's business rule slice. Adapted from [RCRK08]

5.3 Transforming Dependency Graphs into Feature Models

Feature modeling is suitable for modeling changes implemented as aspects.[3] Dependency graphs can be used to express change propagation, and dependencies among concerns and their slices. This section will describe two different approaches of transformation of dependency graphs to feature models. The first one captures the dependencies primary by the feature model hierarchy (Section 5.3.1). The second one using the feature model constraints, which is discussed in the next section. Feature diagrams are structural while dependency graphs seam to capture behavior, too. Capturing of the behavioral component of dependency graphs will be discussed (Section 5.3.3).

5.3.1 Dependencies Captured by Feature Diagrams

In this approach, concern slices are considered as features of the system. Dependencies are modeled as relationships between features.

There are three types of dependencies between concern slices in concern dependency graphs: forward, backward, and parallel. Forward dependency represents what might follow from one concern slice. It can be understood as optionally, thus the concern slice is forward dependent on some other concern slice, this concern slice should be modeled as an optional feature of the former concern slice.

Backward dependency is expressed by the tree topology: a backward

[3]This chapter builds on adapted text from my paper Dealing with Interaction of Aspect-Oriented Change Realizations using Feature Modeling (Appendix G). I am the only author of this paper.

dependent concern slice is a subfeature of the slice it depends on. At the same time, there is a forward dependency in the opposite direction, which is in compliance with available concern dependency graphs [RCRK08].

In terms of feature modeling, parallel dependency simply poses a constraint that two features that represent parallel dependent concern slices must appear together in all possible system configurations. One way to achieve this is to model either of them as a mandatory feature of the other one.

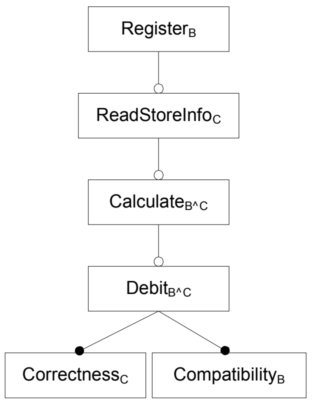


Figure 5.3: Dependencies captured by feature diagrams.

Figure 5.3 shows an example of transformed dependency graph from Section 5.2. All the forward dependencies were modeled as optional features of former concern slices. The two parallel dependencies $\mathit{Correctness}_C$ and $\mathit{Compatibility}_B$ were modeled as mandatory features of debit concern slice $\mathit{Debit}_{B \wedge C}$. The backward dependencies of $\mathit{Calculate}_{B \wedge C}$ and $\mathit{ReadStoreInfo}_C$ concern slices are expressed by tree topology.

One can notice that calculate concern slices $\mathit{Calculate}_{B \wedge C}$ were not modeled as forward dependent on read store gizmo information concern slice in dependency graph from Figure 5.2. In the feature model representation however, a forward dependency on the read store gizmo information concern slice $\mathit{ReadStoreInfo}_C$ is modeled. From our observation a forward and backward dependency very often occur together, therefore can be modeled together. In special cases where such approach is undesirable dependencies should be explicitly captured by additional constraints (Section 5.3.2).

5.3.2 Dependencies Captured by Additional Constraints

Concern slices can also be modeled in a more common style: as usual system features. This way they would form feature hierarchies that correspond to their position in the system hierarchy. However, dependencies between concern slices would have to be expressed as additional constraints. An example is depicted in Fig. 5.4. Additional constraints can be expressed using logic expressions [Vra05, Vra04b].

$Register_C \Rightarrow ReadStoreInfo_B$

$Calculate_{B\wedge C} \Rightarrow ReadStoreInfo_B, Register_C$

$Debit_{B\wedge C} \Rightarrow Calculate_{B\wedge C}, ReadStoreInfo_B, Register_C$

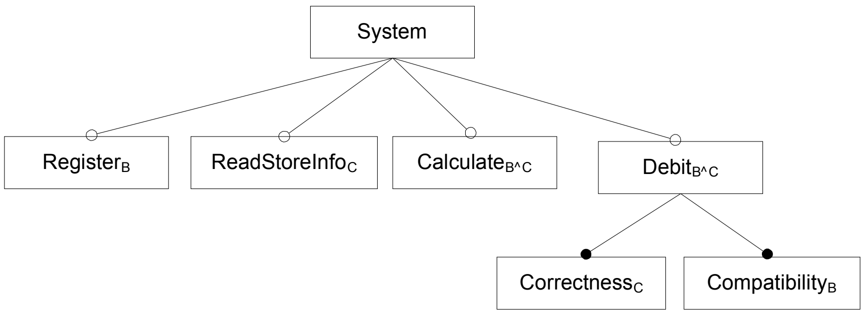


Figure 5.4: Concern slices as usual system features

This approach is much more compatible with the intended use of the transformed dependency graphs. In the same fashion changes implemented by aspects can be modeled. Therefore, feature models in this form can be used for modeling changes represented as aspects, existing concerns along with the dependencies among them. Feature models offer strong means how to express additional constraints which could occur in special occasions.

5.3.3 Capturing Behavioral Component of Dependency Graphs

Dependency graphs seam to address behavior, too. This can be seen also from example in Figure 5.2 where the read store gizmo information concern slice *ReadStoreInfo*$_C$ appears twice. This was noticed also in other studied examples. Feature models are structural and cannot address such behavior. The behavioral component of dependency graphs can be captured with state charts.

State charts are also appropriate if dependency weights which were part of the dependency graph should be preserved. These weights are assigned to dependencies in process of change propagation evaluation. Figure 5.5 shows an example of state chart for register concern's business rule slice. Concern slices are depicted as states and dependencies and their weights were depicted as transitions.

In this simple example only read store gizmo information concern slice *ReadStoreInfo*$_C$ was duplicated in the original dependency graph. One can notice that in the state chart is this concern slice represented by one state.

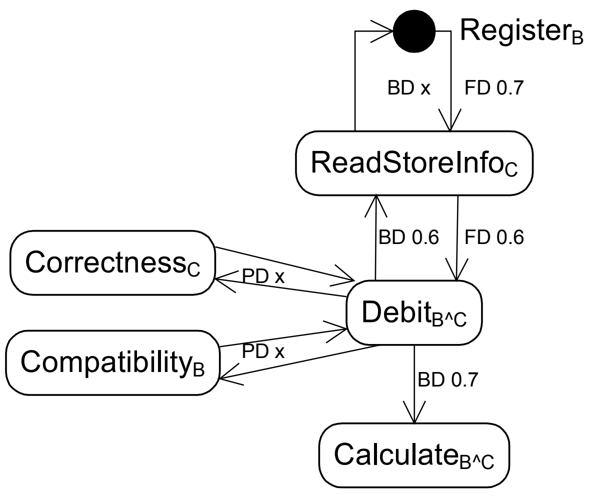


Figure 5.5: State chart for *register* concern's business rule slice

5.4 Modeling Changes Implemented by Aspects

The changes applied to existing system can be represented also by the feature modeling. This section will introduce such approach. A technique how to interpret changes by feature diagrams will be described. Feature modeling can help in several ways to discover additional constraints and interactions among the changes represented by aspects.

Some changes can interact they may be mutually dependent. Changes may also depend on the parts of the underlying system affected by other changes. With increasing number of change requests, change interaction can easily escalate into a serious problem: serious as feature interaction [VBMD08].

Change realizations in the sense of the approach presented so far actually resemble features as coherent pieces of functionality. Moreover, they are virtually pluggable and as such represent *variable* features. This brings us to feature modeling as an appropriate technique for managing variability in software development including variability among changes [VBMD08].

Modeling changes implemented by aspects using feature modeling will be demonstrated on a example of an affiliate tracking software [BVD07]. In a simplified schema of affiliate marketing, a customer visits an affiliate's site which refers him to the merchant's site. When he buys something from the merchant, the provision is given to the affiliate who referred the sale. A general affiliate marketing software enables to manage affiliates, track sales referred by these affiliates, and compute provisions for referred sales. It is also able to send notifications about new sales, signed up affiliates, etc.

The general affiliate marketing software has to be adapted (customized), which involves a series of changes. Consider the following changes and their respective realizations indicated by generally applicable change types:

- SMTP Server Backup A/B as Class Exchange: to have a backup server for sending notifications, A/B variants represent different implementations
- Newsletter Sign Up as Performing Action After Event: to sign affiliate to a newsletter when an he signs up to the tracking software
- Registration Constraint as Method substitution: to check whether the affiliate who wants to register submitted valid e-mail address
- Restricted Administrator Account as Method substitution with Introducing regions to create account with restriction to use some resources
- Hide Options Unavailable to Restricted Administrator as Method Substitution: to restrict the users interface
- User Name Display Change as Method Substitution: to alter the presentation of name
- Account Registration Statistics as Performing Action After Event: to gain statistical information about the affiliate registrations

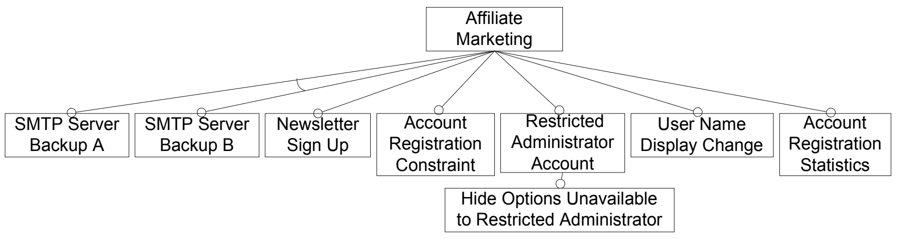

Affiliate Marketing
SMTP Server Backup A
SMTP Server Backup B
Newsletter Sign Up
Account Registration Constraint
Restricted Administrator Account
User Name Display Change
Account Registration Statistics
Hide Options Unavailable to Restricted Administrator

Figure 5.6: Feature diagram of changes in affiliate tracking software

When creating a feature diagram of the changes, the system to which changes are applied to is considered to be a concept. Every change represented by aspect is then considered a feature of the underlying system and added to the feature diagram. There are two possibilities how new change can be added to the feature diagram. Both will be demonstrated in Figure 5.6. First when there is no information that a change is in any kind of relationship with other changes and second when such information is present. In first scenario the change is connected directly to concern which represents the system (*Newsletter Sign Up*). In second scenario according to relationship information the change can be connected as subfeature of already existing change (*Hide Information Unavailable to Restricted*

Administrator), depicted as an alternative of some change (*SMTP Server backup A* and *SMTP Server backup B*) or the diagram has to be rearranged if existing change should be treated as subfeature of added change. This is basically the same approach as constructing the standard feature diagram when considering the changes as features.

By this process it is possible to create a diagram of changes present in the system. This diagram can be later analyzed and even expanded in search of possible interactions of changes. This technique will be discussed in more detail in following sections.

5.5 Direct Dependences and Interactions

Dependences and interactions expressed directly by a feature diagram of changes implemented by aspects can appear in the form of an implementation or logical dependency.

After a base diagram of changes present in the system is constructed, by following the technique described in the previous section, one can spot immediately some patterns that can indicate a dependency or interaction among depicted changes. One of most usual patterns is the feature–subfeature occurrence. When a change is listed as subfeature of another change it means that there is some known relationship between these changes. The pattern usually suggests that a change which is on the lower level in the diagram tree has only meaning only when the change on the higher level is included to the system. These situations are target of our interest and should be analyzed in search for a potential interaction.

Another pattern is the *or* relationship occurrence. Also in this scenario changes can have a similar functionality, which could lead to the interaction of the changes.

By both mentioned patterns is possible to distinguish between a several types of dependency that can occur. Two pointcuts could use a same join points which could lead to the unwanted interaction of such changes. Another type of dependency can have only logical character. In this case two changes can have a similar functionality or one change extends the functionality of the other, but they apply to different join points or even parts of the system. In this case no interaction should occur. Sometimes it is possible to distinguish between these cases immediately, otherwise is needed to proceed further with the analysis and apply technique which is also used for indirect dependences analysis (Section 5.6).

5.6 Indirect Dependences and Interactions

Additional dependences among changes can be discovered by the underlying system exploration to which the changes are introduced. To achieve this goal it is essential to know a model of the system itself. Modeling of the whole system can be difficult and time consuming task. In some cases it could be possible to find a new dependency without need to explore the whole feature model of the system. Instead it is possible to construct a partial feature diagram which would be examined to the degree sufficient to evaluate a dependency between underlying changes implemented by aspects. In the following sections technique how to achieve this goal will be described. Section 5.6.1 will describe Partial feature diagram construction. Section 5.6.2 will describe how to evaluate the found dependences and section 5.6.3 will show how to derive constraints which will be added to the feature diagram.

5.6.1 The Partial Feature Model Construction

The process of constructing partial feature model is based on the feature model in which aspect-oriented change realizations are represented by variable features that extend an existing system represented as a concept (see Section 5.4).

The concept node in this case is an abstract representation of the underlying software system. Potential dependencies of the change realizations are hidden inside of it. In order to reveal them, it is needed to factor out concrete features from the concept. Starting at the features that represent change realizations (leaves) one should proceed bottom up trying to identify their parent features until related changes are not grouped in common subtrees. Figure 5.7 depicts this process.

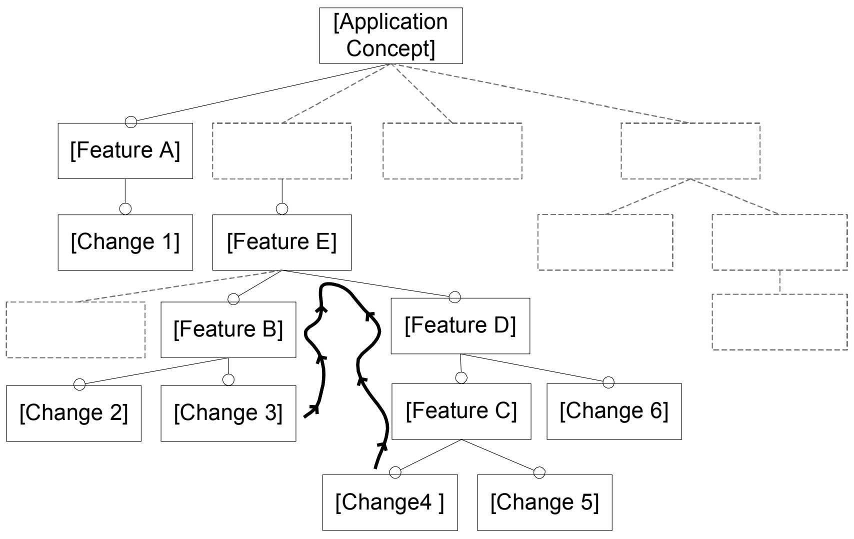


Figure 5.7: Constructing a partial feature model.

The process of partial feature model creation will be demonstrated on our affiliate marketing software example (Section 5.4). Initial step for the process is represented by model presented in figure 5.6. Following this initial stage, it is needed to identify parent features of the change realization features as the features of the underlying system that are affected by them (Figure 5.8). During analysis was found:

- *SMTP Server Backup A* affects the *SMTP Server Creation* feature
- *Newsletter Sign Up*, *Account Registration Statistics* and *Registration Constraint* change *Affiliate Sign Up*
- *Restricted Administrator Account* is changes of *Banner management* and *Campaign management*
- *User Name Display Change* is changes of *Displaying Grid Data*
- *Hide Operations Unavailable to Restricted Administrator* is changes of *Displaying Menu Items*

Knowing that *Hide Operations Unavailable to Restricted Administrator* has meaning only when *Restricted Administrator Account* change was applied to the system, following constraint was set on model.

- Hide Operations Unavailable to Restricted Administrator ⇒Restricted Administration Account

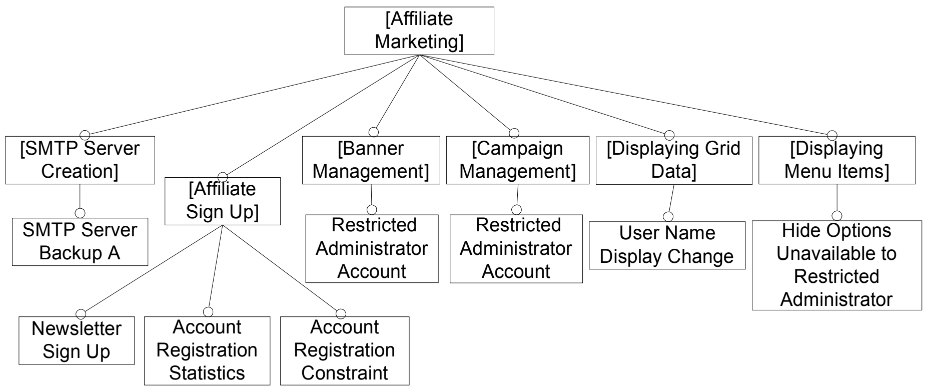


Figure 5.8: Affiliate marketing web application partial feature model.

All new features are marked as open representing that other features not captured by the model are possible. In this simple example it was not possible to find any other parent features so process of partial feature model creation finishes after this single iteration.

To better explain this process another example from real application will be described. Such system has a higher complexity and relations between features will be more complicated. System used for example is a student project management system called YonBan and it was developed at Slovak University of Technology. Consider the following changes in YonBan and their respective realizations indicated by generally applicable change types:

- Telephone Number Validating (realized as Performing Action After Event): to validate a telephone number the user has entered
- Telephone Number Formatting (realized as Additional Return Value Checking/Modification): to format a telephone number by adding country prefix
- Project Registration Statistics (realized as One Way Integration): to gain statistic information about the project registrations
- Project Registration Constraint (realized as Additional Parameter Checking/Modification): to check whether the student who wants to register a project has a valid e-mail address in his profile
- Exception Logging (realized as Performing Action After Event): to log the exceptions thrown during the program execution
- Name Formatting (realized as Method Substitution): to change the way how student names are formatted

These change realizations are captured in the initial feature diagram presented Fig. 5.9. Since there was no relevant information about direct dependencies among changes during their specification, there are no direct dependencies among the features that represent them either. The concept of the system as such is marked as open (indicated by square brackets), which means that new variable subfeatures are expected at it. The main reason for this is that only part of the analyzed system is shown, but another features are present in the system too.

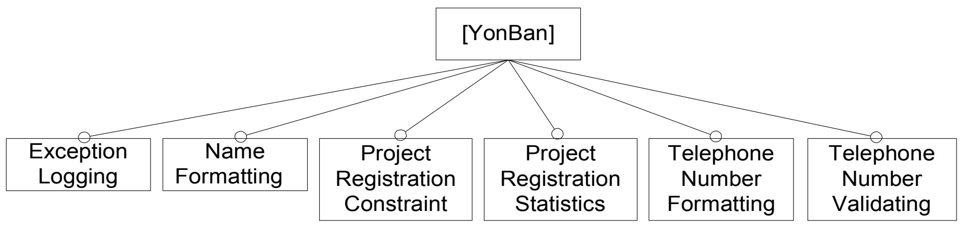

[YonBan]
Exception Logging
Name Formatting
Project Registration Constraint
Project Registration Statistics
Telephone Number Formatting
Telephone Number Validating

Figure 5.9: Initial stage of the YonBan partial feature model construction.

Figure 5.10 shows such changes identified in our case. Analysis discovered that *Name Formatting* affects the *Name Entering* feature. *Project Registration Statistics* and *Project Registration Constraint* change *User Registration. Telephone Number Formatting* and *Telephone Number Validating* are changes of *Telephone Number Entering*. *Exception Logging* affects all the features in the application, so it remains a direct feature of the concept. All these newly identified features are open because of the incompleteness of their subfeature sets.

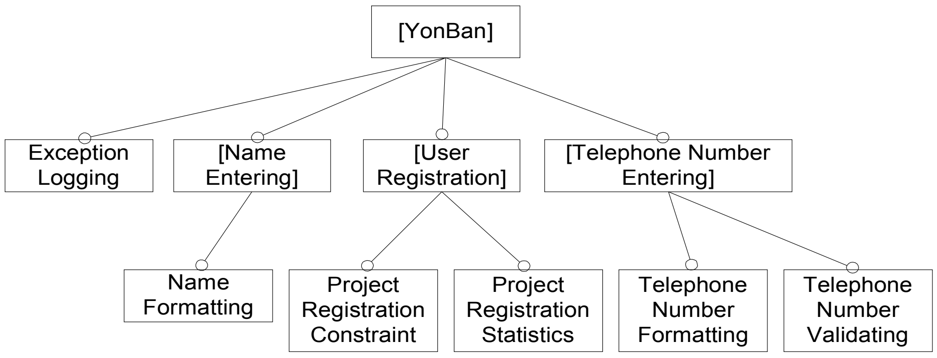

[YonBan]
Exception Logging
[Name Entering]
[User Registration]
[Telephone Number Entering]
Name Formatting
Project Registration Constraint
Project Registration Statistics
Telephone Number Formatting
Telephone Number Validating

Figure 5.10: Identifying parent features in YonBan partial feature model construction.

The process is repeated until it is possible to identify parent features or until all the changes are found in a common subtree of the feature diagram, whichever comes first. This stage is reached within the following—and thus last—iteration which is presented in Fig. 5.11: It was revealed that *Telephone Number Entering* and *Name Entering* is a part of *User Registration*.

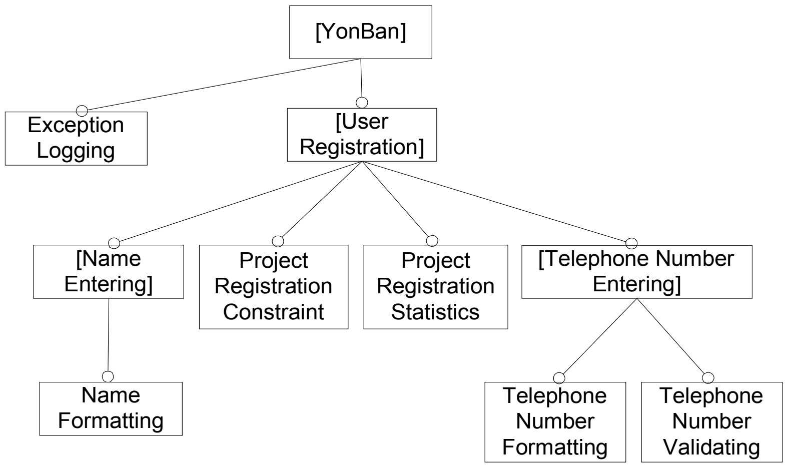

[YonBan]
Exception Logging
[User Registration]
[Name Entering]
Project Registration Constraint
Project Registration Statistics
[Telephone Number Entering]
Name Formatting
Telephone Number Formatting
Telephone Number Validating

Figure 5.11: The final YonBan partial feature model.

5.6.2 Dependency Evaluation

The partial feature model constructed according to the technique presented in previous section contains new a relationships and dependencies which should be analyzed to continue the search for a possible change interactions. If there is a knowledge how these changes are going to be implemented one can start analyzing the relationships immediately and find out which of them will cause interaction among changes. If there is no such knowledge, it is possible, in this stage, to identify the potential locations where interactions may occur. The location can be understood as a specific target in the code or as an area defined by the concern slice or several concern slices.

Generally an interaction has highest probability to occur when two or more changes are represented as direct subfeatures of one parent feature. In this case the code altered by one change can be dependent on the code altered by second change or vice versa. To deal with changes that are not direct subfeatures of one feature whole area specified with all concerns present in the subtree of the feature diagram containing analyzed changes should be examined. In this process it is possible to use dependency graphs [RCRK08] or program slicing [Wei81, GL91] to check for an undesirable interaction. The main idea is that it is needed to check all the elements from the concerns for possible interaction. The positive outcome from this modeling technique is that the area where this interaction will with most probability occur is delimited by concerns present in the subtree of the partial feature model.

If the implementation details of the changes are missing the analysis can finish with result which represents a locations where can possible interactions occur. This result is also very satisfying and can be very helpful in the implementation phase. On the other hand it is possible to go further with the analysis using the technique in which changes represented by aspects are further analyzed. This approach uses methods from multi-paradigm design with feature modeling[4] [Vra04a] and will be presented in chapter 6.

5.6.3 Deriving Constraints

All additional informations about changes gained while analyzing change interactions should be described in some way and distributed along with the partial feature model itself. It is possible to express such information by additional constraints, default dependency rules and notes. Constraints and default dependency rules use a predicate logic to express additional relations between features. The main difference between the constraints and dependency rules is in their weight. While constraints must be satisfied in each concept instance default dependency rules allow exceptions [Vra04a]. By note it is possible to introduce additional informations available about a concern.

[4] $\mathrm{MPD}_{\mathrm{FM}}$

Each of these notations should express different facts discovered through dependency evaluation of the change interactions. If the change interaction is considered strong, or hard to solve, one should consider setting a constraint upon related changes. If the discovered interaction has only logical dimension, or the problem coupled with it can be solved (Chapter 4) note should be set. Locations of possible interactions should be expressed at least by a note.

Chapter 6

Change Realization Using $\mathrm{MPD}_{\mathrm{FM}}$

In this chapter change realization techniques (Chapter 3) will be introduced as paradigms in sense of multi-paradigm design with feature modeling ($\mathrm{MPD}_{\mathrm{FM}}$).[1] These new paradigms can be than used during transformational analysis to map concepts (which represent changes) from application domain to solution domain. Details gained trough transformational analysis are crucial for final interaction evaluation.

Section 6.1 presents basic concepts of a particular multi-paradigm approach based on feature modeling. Section 6.2 will describe approach in which generally applicable changes are seen as paradigms. In Section 6.3 transformational analysis process which uses new paradigms is explained. Section 6.4 describes how to evaluate change interaction after new informations are gained trough transformational analysis.

6.1 Multi-Paradigm Design with Feature Modeling

In $\mathrm{MPD}_{\mathrm{FM}}$, paradigms are understood as *solution domain concepts* that correspond to programming language mechanisms (like inheritance or class). Such paradigms are being denoted as small-scale to distinguish them from the common concept of the (large-scale) paradigm as a particular approach to programming (like object-oriented or procedural programming) [Vra05].

In $\mathrm{MPD}_{\mathrm{FM}}$, feature modeling is used to express paradigms. A feature model consists of a set of feature diagrams, information associated with concepts and features, and constraints and default dependency rules associated

[1]This chapter builds on adapted text from my paper Aspect-Oriented Change Realization Based on Multi-Paradigm Design with Feature Modeling (Appendix D) to which my contribution is approximately 60 % .

with feature diagrams. A feature diagram is usually understood as a directed tree whose root represents a concept being modeled and the rest of the nodes represent its features [Vp06].

The features may be common to all concept instances (feature configurations) or variable, in which case they appear only in some of the concept instances. Features are selected in a process of concept instantiation. Those that have been selected are denoted as bound. The time at which this binding (or not binding) happens is called binding time. In paradigm modeling, the set of binding times is given by the solution model. In AspectJ one can distinguish among source time, compile time, load time, and runtime.

Each paradigm is considered to be a separate concept and as such presented in its own feature diagram that describes what is common to all paradigm instances (its applications), and what can vary, how it can vary, and when this happens. Recall the AspectJ aspect paradigm feature model presented in Fig. 5.1.

In modeling change types as paradigms, each change type is considered to be a concept and as such presented in a separate feature diagram created according to the solution domain related information. Paradigms that may be used in the paradigm being modeled should be referenced by it. If a paradigm enables instantiation, it should be modeled as a feature (or features). If the feature is variable, its binding time has to be selected among the binding times identified in the solution domain. If none is appropriate, a new binding time should be established.

Feature modeling is applied also to the application domain. The two feature models, the application and solution domain one, enter the process called *transformational analysis* in which application to solution domain mapping is being established. This mapping is expressed in the form of yet another feature model consisting of the paradigm instances annotated with the information about corresponding application domain concepts and features which determines the code skeleton.

Generally applicable changes may be seen as a kind of conceptually higher language mechanisms and modeled as paradigms in the sense of MPD$_{\text{FM}}$. Two change types will be presented as paradigms in next section. All general change types are presented as paradigms in Appendix A.

6.2 Generally Applicable Change Types as Paradigms

Generally applicable change types are independent of the application domain and may even apply to different aspect-oriented languages and frameworks (with an adapted code scheme, of course). The expected number of generally applicable changes that would cover all significant situations is not high. In experiments, it was possible to cope with all situations using only six generally applicable change types.

On the other hand, in the domain of web applications, eleven application specific changes was identified, and they represent only its partial coverage. Each such change requires a thorough exploration in order to discover all possible realizations by generally applicable changes and design patterns with conditions for their use, and it is not likely that someone would be willing to invest effort into developing catalog of changes apart of the actual change development.

The problem of selecting a suitable generally applicable change type resembles the problem of the selection of a paradigm suitable to implement a particular application domain concept. Here, the multi-paradigm design can be employed. In Sect. 6.2.1 and 6.2.2, paradigm models of Method Substitution and Performing Action After Event will be introduced. These and feature models of all the rest of known generally applicable change types are included in Appendix A.

6.2.1 Method Substitution

Figure 6.1 shows the Method Substitution change type feature model. Method Substitution is used to change, alter or even disable execution of a particular method or methods [BVD07]. It is implemented by an aspect (*Aspect* ®) with a pointcut specifying the calls to the methods (*Original Method Call*) to be altered by an around advice. Thus, two additional constraints were set:

- Aspect.Pointcut
- Aspect.Advice.Around

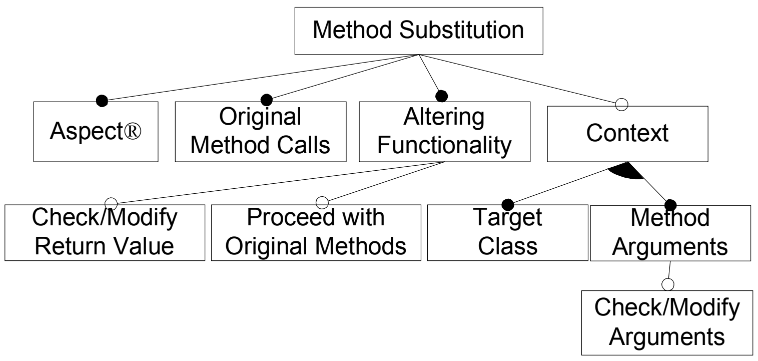


Figure 6.1: Method Substitution.

The target class of the method call and method arguments, which constitute a part of the context of the method call, are made available using appropriate primitive pointcuts *target()* and *args()*. Sometimes it is sufficient only to alter the method arguments in which case it is possible to use the *proceed()* statement with altered arguments in the advice body.

6.2.2 Performing Action After Event

Performing Action After Events [BVD07] is a simple but frequent form of the change realization. It is used when additional actions are needed after the specific event. The event is specified by pointcut, for example method or constructor execution/call or field modification. An after advice on specified pointcut is then used to perform the desired actions (Fig. 6.2). Thus, two additional constraints was set:

- Aspect.Pointcut.Call
- Aspect.Advice.After

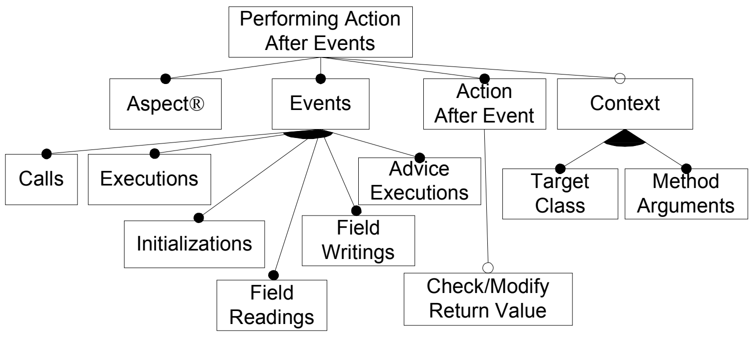


Figure 6.2: Performing action after an event paradigm.

6.3 Transformational Analysis

Transformational analysis in multi-paradigm design is a process of finding the correspondence and establish the mapping between the application and solution domain concepts. The input to transformational analysis are two feature models: the application domain one and the solution domain one [Vra05]. In this case the application domain model is represented by the application feature model (Section 5.4) and the solution domain model is represented by the AspectJ solution domain model [Vra01] which is extended by feature models of the generally applicable change types (Section 6.2). The output of transformational analysis is a set of paradigm instances annotated with application domain feature model concepts and features.

Transformational analysis was covered in $\mathrm{MPD_{FM}}$ approach [Vra05], therefore transformational analysis in this thesis covers especially features which represent changes. Transformation of changes from the application domain feature model is driven by similar rules as transformation of the features in the original $\mathrm{MPD_{FM}}$ approach. The difference is how a concept from application domain that represents a change is treated. First all new direct usable paradigms (Section 6.2) should be chosen to be instantiated over concept representing a change from application domain.

If this is not successful concept probably represents a new change type. In this case original AspectJ direct paradigms should be chosen to be instantiated over this concept, which is covered in the original transformational analysis process [Vra05][1].

As a result from stated additional transformation steps are created. These steps should be executed before the steps of original transformation process as proposed in $\mathrm{MPD_{FM}}$.

1. If selected concern C from application domain model represents a change, choose a direct usable paradigm representing generic change type from solution domain P which has not been considered for C jet.

2. Try to instantiate P over C. If this is not successful go to next step. Otherwise, record the paradigm instance created.

3. If there are no direct usable paradigms to select continue with transformational analysis according to the steps presented in $\mathrm{MPD_{FM}}$ [Vra05][1] , else return to step 1.

Transformational analysis will be described on the affiliate marketing software example (Section 5.4). Consider transformational analysis for the *Restricted Administrator Account* feature. This change should provide an additional check of access rights upon the execution of specified methods. Methods should be executed only if access is granted. This scenario suites best to the Method Substitution paradigm which can control the execution of selected methods. Moreover, in case of access violation method substitution can provide subroutines to display error message or log access violation event. Figure 6.3 represents the transformation of *Restricted Administrator Account* feature. *Target Class* and *Method Arguments* were used to capture additional context which is needed by *Proceed with Original Methods* when access is granted. One can notice that *Banner Management* and *Campaign Management* features were mapped to *Original Method Call.* Such association means that this change will affect the behavior represented by these features. Realizing and capturing such associations is crucial to change interaction evaluation. This topic will be further discussed in the next section.

[1]Original transformation analysis from $\mathrm{MPD_{FM}}$ approach is listed in appendix B.

In this particular example, one can notice that several different features were mapped to the original method call paradigm. While this causes no problem, because all these mappings can be expressed by the pointcut declared in constraint, it is possible to make these mappings reusable with the Introducing Regions paradigm. The pointcut or pointcuts would then be declared in separate aspect representing introducing regions paradigm.

Transformational analysis for the *Registration Constraint* would be very similar. Again the Method Substitution paradigm would be used. *Original Method Call* would be mapped to *Affiliate Sign Up* And the original method will be executed by the *Proceed with Original Methods* only when a valid email was provided. Two additional constraints remained set:

- Aspect.Pointcut.Call
- Aspect.Advice.Around

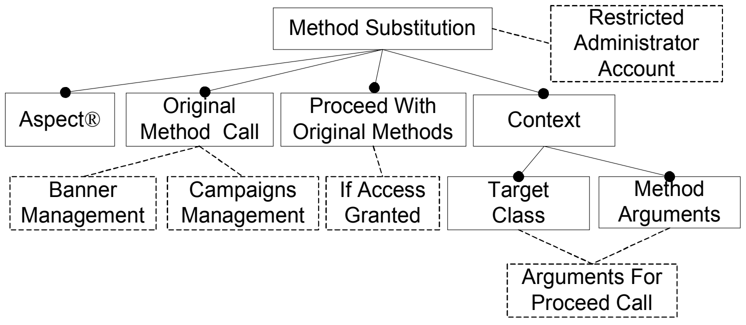


Figure 6.3: Restricted User Account transformational analysis

Another example is transformational analysis of the *Newsletter Sign Up* paradigm. This change request should add new affiliate also to the existing list of newsletter recipients. This can be best achieved as Performing Action After Event (Figure 6.4). In this case *Event* was mapped to *Affiliate Sign Up* paradigm which in this case represents execution of the affiliate sign up method. Trough the *Method Arguments* data about new affiliate can be accessed. From accessed data the e-mail address of an affiliate can be gained. Finally address is added to the newsletter recipient list trough the *Action After Event* paradigm. In similar manner also *Registration Statistics* paradigm would be transformed. Two additional constraints remained set:

- Aspect.Pointcut
- Aspect.Advice.After

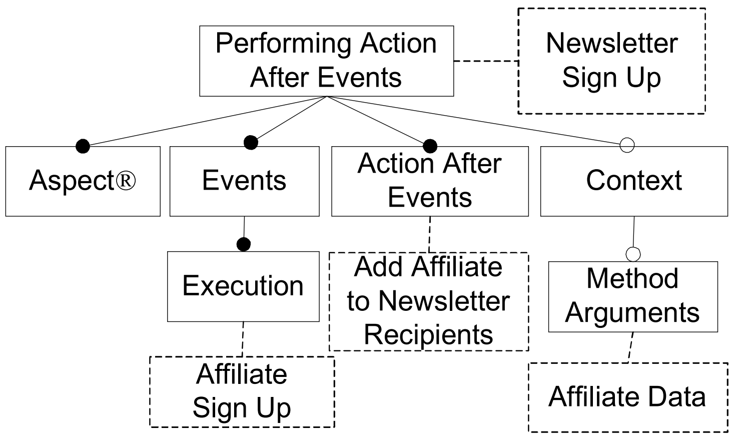


Figure 6.4: Newsletter Sign Up transformational analysis

6.4 Interaction Evaluation

In previous section an example of transformational analysis was presented. By this process concepts from application domain are mapped to solution domain paradigms. From the change interaction point of view, very important are the mappings of pointcut paradigms. These mappings reveal the destinations in application domain where change will affect the existing source code.

The highest probability of interaction occurs by the situations in which several changes affect the same destinations from an application domain. Such situations could be identified while creating the partial feature model. Notice the situation in figure 5.8 where three changes act as subfeature of *Affiliate Sign Up* feature. Every situation should be evaluated and suitable actions should be performed to avoid the interaction. Reason for further evaluation rises from fact that not every collision of pointcuts leads to change interaction. The details, which are needed to decide if interaction occurs or not, are acquired along with transformational analysis.

For example, consider the *Newsletter Sign Up* and the *Account Registration Statistics* changes, despite they share a common pointcut *Affiliate sign up* no interaction should occur. This is given by the fact that both changes were mapped to *Performing Action After Event* paradigm which uses after advice. In such situation it is important to evaluate if advices of changes should be evaluated in particular order. By *Account Registration Statistics* and *Affiliate sign up* changes such order is irrelevant.

Another problem can be caused by the *Account Registration Constraint* change which uses the same pointcut. This change was mapped to the *Method Substitution* paradigm through which it can disable the execution of the method which registers a new affiliate. If the *Newsletter Sign Up* and *Account Registration Statistics* changes use *execution* pointcut, everything is all right. On the other hand, if these changes would use *call* pointcut their advices would be still executed even when the registration method would not be executed. This would cause an undesirable system behavior. Details such as type of used pointcuts and other relevant data for interaction analysis is acquired by transformational analysis process. In this way transformational analysis makes interaction evaluation possible.

In most cases when interaction occurs it can be solved by adapting the change implementation. If unsolvable interaction should occur, constraints should be added upon application domain model which will ensure the interacting changes will not occur together in any concept instance.

Chapter 7

Conclusion and Future Work

This thesis described the problems coupled with interaction of changes represented by aspects. To implement a change by aspect a general change type (change realization technique) can be used. Important change realization techniques were summarized (Chapter 3). Some techniques were changed and two additional techniques were proposed. Work discussed several problems and their solutions coupled with the interaction of changes represented by aspects (Chapter 4).

Interactions between the changes implemented by aspects can be traced using a new approach that uses feature modeling technique. Using this technique changes can be analyzed on different levels of abstraction. First, a feature model of changes present in the system is created (Section 5.4). With this model it is possible to capture direct interactions of changes. To study additional interactions of changes, it is needed to construct a partial feature diagram of the system (Section 5.6.1. From partial feature diagram indirect interactions or locations of possible interactions can be derived.

To evaluate the possible interaction of changes it is needed to know their implementation which depends on the used change realization type. Selecting appropriate change type can be achieved using the multi-paradigm design (Section 6). In this approach the change realization techniques are considered paradigms of the AspectJ solution domain (Section 6.2). After the transformation analysis interactions can be evaluated (Section 6.4).

As future work this approach should be evaluated on a systems of larger scale. If the aspect-oriented paradigm popularity will grow it is possible that a new design patterns or idioms will emerge in it. Using these new design patterns and idioms additional change realization techniques can be created, in which case, extension of this approach would be appropriate.

Bibliography

[Beb07] Michal Bebiak. Aspektovo-orientovaná implementácia zmien vo webových aplikáciách. Master's thesis, Slovenská technická univerzita v Bratislave Fakulta Informatiky a Informacných Technológii, 2007.

[Bre08] Jakub Breier. On interaction among aspect-oriented change representation. In Mária Bieliková, editor, *IIT.SRC: Student Research Conference 2008*, pages 1–6. Slovak University of Technology, 2008.

[BVD07] Michal Bebjak, Valentino Vranić, and Peter Dolog. Evolution of web applications with aspect-oriented design patterns. In Marco Brambilla and Emilia Mendes, editors, *Proc. of ICWE 2007 Workshops, 2nd International Workshop on Adaptation and Evolution in Web Systems Engineering, AEWSE 2007, in conjunction with 7th International Conference on Web Engineering, ICWE 2007*, pages 80–86, Como, Italy, July 2007.

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programing: Methods, Tools, and Applications.* Addison-Wesley, 2000.

[CW98] Reidar Conradi and Bernhard Westfechtel. Version models for software configuration management. *ACM Computing Surveys*, 30(2):232–282, 1998.

[DVB01] Peter Dolog, Valentino Vranić, and Mária Bieliková. Representing change by aspect. *ACM SIGPLAN Notices*, 36(12), December 2001.

[Faz06] Zoltán Fazekas. *Improving Variability in Software Configuration Mmanagement by Separation of Concerns.* PhD thesis, Slovenská technická univerzita v Bratislave Fakulta Informatiky a Informacných Technológii, 2006.

[GL91] Keith Brian Gallagher and James R. Lyle. Using program slicing in software maintenance. *IEEE Trans. Softw. Eng.*, 17(8):751–761, 1991.

[Lad03] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming.* Manning, 2003.

[LKL02] Kwanwoo Lee, Kyo C. Kang, and Jaejoon Lee. Concepts and guidelines of feature modeling for product line software engineering. In *Proceedings of the Seventh International Conference on Software Reuse*, pages 62–77, April 2002.

[Men07] Radoslav Menkyna. Towards combining aspect-oriented design patterns. In Mária Bieliková, editor, *IIT.SRC: Student Research Conference 2007*, pages 1–8. Slovak University of Technology, 2007.

[Men08] Radoslav Menkyna. The director as a connection between object-oriented and aspect-oriented design patterns. In Mária Bieliková, editor, *IIT.SRC: Student Research Conference 2008*, pages 55–61. Slovak University of Technology, 2008.

[Mil04] Russell Miles. *AspectJ Cookbook.* O'Reilly, 2004.

[PARa] Xerox PARC. Aspect-oriented programming home page. http://aosd.net/. Last accessed in april 2008.

[PARb] Xerox PARC. Aspectj home page. http://www.eclipse.org/aspectj/. Last accessed in april 2008.

[PSC01] Elke Pulvermüller, Andreas Speck, and James O. Coplien. A version model for aspect dependency management. *Lecture Notes in Computer Science*, 2186:70–??, 2001.

[RCRK08] Safoora Omer Rashid, Ruzanna Chitchyan, Awais Rashid, and Raffi Khatchadourian. Approach for change impact analysis of aspectual requirements, march 2008. AOSD-Europe Deliverable D110, AOSD-Europe-ULANC-40.

[VBMD08] Valentino Vranić, Michal Bebjak, Radoslav Menkyna, and Peter Dolog. Developing applications with aspect-oriented change realization. In *Proc. of 3rd IFIP TC2 Central and East European Conference on Software Engineering Techniques CEE-SET 2008*, LNCS, Brno, Czech Republic, October 2008. Springer. Postproceedings, to appear.

[Vp06] Valentino Vranić and Miloslav Šípka. Binding time based concept instantiation in feature modeling. In Maurizio Morisio, editor, *Proc. of 9th International Conference on Software Reuse (ICSR 2006)*, LNCS 4039, pages 407–410, Turin, Italy, June 2006. Springer.

[Vra01] Valentino Vranić. AspectJ paradigm model: A basis for multi-paradigm design for AspectJ. In Jan Bosch, editor, *Proc. of 3rd International Conference on Generative and Component-Based Software Engineering (GCSE 2001)*, LNCS 2186, pages 48–57, Erfurt, Germany, September 2001. Springer.

[Vra04a] Valentino Vranić. *Multi-Pradigm Design with Feature Modeling*. PhD thesis, Slovak University of Technology in Bratislava, Slovakia, April 2004.

[Vra04b] Valentino Vranić. Reconciling feature modeling: A feature modeling metamodel. In Matias Weske and Peter Liggsmeyer, editors, *Proc. of 5th Annual International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a Networked World (Net.ObjectDays 2004)*, LNCS 3263, pages 122–137, Erfurt, Germany, September 2004. Springer.

[Vra05] Valentino Vranić. Multi-paradigm design with feature modeling. *Computer Science and Information Systems Journal (ComSIS)*, 2(1):79–102, June 2005.

[Wei81] Mark Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

Appendix A

AspectJ Solution Domain Extension

This appendix contains the feature models of change realization techniques that can be considered direct usable paradigms of the AspectJ language solution domain. By each feature model additional constraints will be listed. These constraints define the structure of referenced Aspect.

A.1 Performing Action After Event (Figure A.1)

- Aspect.Pointcut
- Aspect.Advice.After

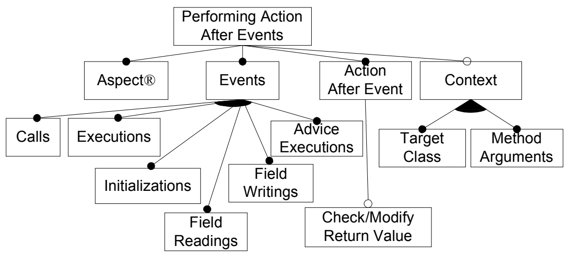


Figure A.1: Feature diagram of Action after event paradigm.

A.2 Method Substitution (Figure A.2)

- Aspect.Pointcut.Call
- Aspect.Advice.Around

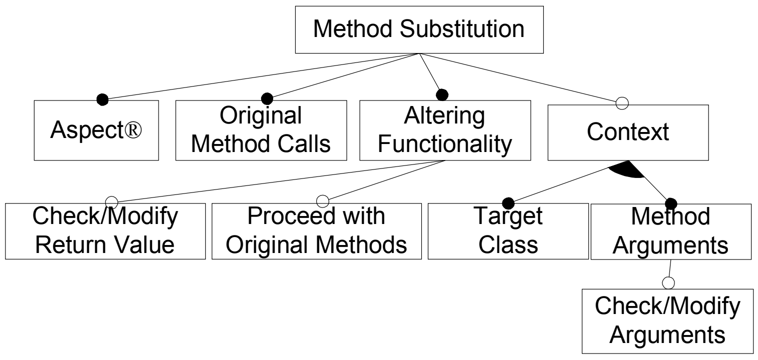


Figure A.2: Feature diagram of Method substitution paradigm.

A.3 Introducing Regions (Figure A.3)

- Aspect.Pointcut

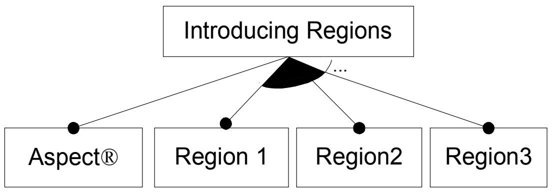


Figure A.3: Feature diagram of Introducing regions paradigm.

A.4 Class Exchange (Figure A.4)

- Aspect.Pointcut
- Aspect.Advice.Around

A.5 Introducing Role to Class (Figure A.5)

- Aspect1.Abstract
- Aspect1.Interface
- Aspect2.Inter-Type Declaration

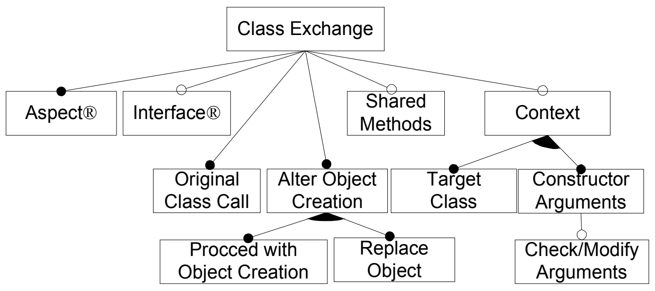


Figure A.4: Feature diagram of Class exchange paradigm.

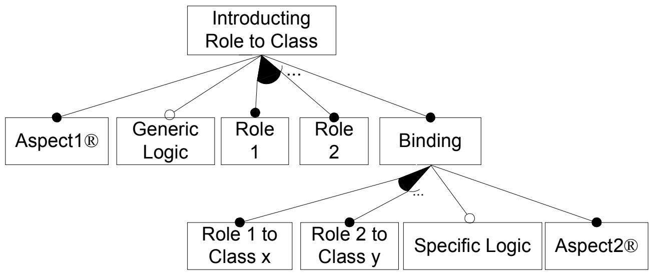


Figure A.5: Feature diagram of Introducing role paradigm.

A.6 Member Introduction (Figure A.6)

- Aspect.Inter-Type Declaration

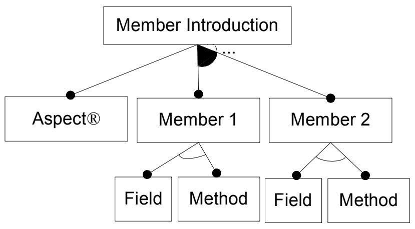


Figure A.6: Feature diagram of Member introduction paradigm.

Appendix B

The Process of Transformational Analysis

This appendix contains description of original transformational analysis process which was adopted from [Vra05].

Transformational analysis is performed as follows. For each concept C from the application domain feature model, the following steps are performed:

1. Determine the structural paradigm corresponding to C:
   (a) Select a structural paradigm P of the solution domain feature model that has not been considered for C yet.
   (b) If there are no more paradigms to select, there may be a level mismatch: C may correspond to a paradigm feature, and not to a paradigm itself. Unless C has been factored out as a concept in step 1d, continue transformational analysis considering C only as a feature of the concepts where it is referenced, and not as a concept. Otherwise, the process has terminated unsuccessfully.
   (c) Try to instantiate P over C at source time. If this couldn't be performed or if P's root doesn't match with C's root, go to step 1a. Otherwise, record the paradigm instance created.
   (d) If there are unmapped non-mediatory feature nodes of C left, factor out them as concepts (introducing concept references in place of the subtrees they headed) and perform the transformational analysis of them. Subsequently, regard them as concept references in C's feature diagram and reconsider the paradigm instance created in step 1c.
2. If there are relationships (direct or indirect ones) between the concept node of C and its non-mediatory features not yet mapped to relationships between the corresponding paradigm feature model nodes,

determine the corresponding relationship paradigms for each such a relationship:

(a) Select a relationship paradigm P of the solution domain feature model that has not been considered for a given relationship in C yet. If there are no more paradigms to select, the process has terminated unsuccessfully.

(b) Try to instantiate P over the relationship in C at source time. If this couldn't be performed or if there are no P's nodes that match with the C's relationship nodes, go to step 2a. Otherwise, record the paradigm instance created.

Paradigm instances could be presented in the overall solution instance tree. However, this is not convenient since the solution instance tree would be too big to cope with it, and it would not provide any additional benefits compared to presenting paradigm instances individually.

A successful transformational analysis results in only one of the possible solutions. Carrying out transformational analysis differently can lead to another solution. Deciding which solution is the best is out of the scope of this method.

Appendix C

Attached CD Contents

Attached CD contains electronic version of this document and four articles from Appendix D to Appendix F.

Appendix D

Aspect-Oriented Change Realization Based on Multi-Paradigm Design with Feature Modeling

This appendix contains:

> Radoslav Menkyna and Valentino Vranić. Aspect-Oriented Change Realization Based on Multi-Paradigm Design with Feature Modeling. Paper in preparation. To be submitted to *4th IFIP TC2 Central and East European Conference on Software Engineering Techniques CEE-SET 2009*, May 2009.

The paper is going to be submitted to 4th IFIP TC2 Central and East European Conference on Software Engineering Techniques CEE-SET 2009. My contribution to this paper is approximately 60 %.

Aspect-Oriented Change Realization Based on Multi-Paradigm Design with Feature Modeling

Radoslav Menkyna and Valentino Vranić

Institute of Informatics and Software Engineering
Faculty of Informatics and Information Technologies
Slovak University of Technology,
Ilkovičova 3, 84216 Bratislava 4, Slovakia
`radu@ynet.sk, vranic@fiit.stuba.sk`

Abstract. Aspect-oriented change realization based on a two-level change type framework can be employed to deal with changes so they can be realized in a modular, pluggable, and reusable way. In this paper, this original idea is extended by enabling direct change manipulation using multi-paradigm design with feature modeling. For this, generally applicable change types are considered to be (small-scale) paradigms and expressed by feature models. Feature models of the Method Substitution and Performing Action After Event change types are presented as examples. In this form, generally applicable change types enter an adapted process of transformational analysis to determine their application by their instantiation over an application domain feature model. The application of the transformational analysis in identifying the details of change interaction is presented.

1 Introduction

Changes of software applications exhibit crosscutting nature either intrinsically by being related to many different parts of the application they affect or by their perception as separate units that can be included or excluded from a particular application build. It is exactly aspect-oriented programming that can provide suitable means to capture this crosscutting nature of changes and to realize them in a pluggable and reapplicable way [13].

Particular mechanisms of aspect-oriented change introduction determine the change type. Some of these change types have already been documented [1, 13], so by just identifying the type of the change being requested, we can get a pretty good idea of its realization. This is not an easy thing to do. One possibility is to have a two-level change type model with some change types being close to the application domain and other change types determining the realization, while their mapping is being maintained in a kind of a catalog [13].

But what if such a catalog for a particular domain does not exist? To postpone change realization and develop a whole catalog may be unacceptable with respect to time and effort needed. The problem of selecting a suitable realizing change type resembles paradigm selection in multi-paradigm design [12]. This other way around—to treat change realization types as paradigms and employ multi-paradigm design to select the appropriate one—is the topic of this paper.

We will first take a look at the two-level aspect-oriented change realization model (Sect. 2). Then, a way of modeling change realization types as paradigms using feature modeling will be introduced (Sect. 3). Expressing changes in the application domain feature model will be presented, too (Sect. 4). Transformational analysis—the process of finding a suitable paradigm—tailored to change realization, will be introduced next (Sect. 5). Afterwards, it will be shown how transformational analysis results can be used to identify change interaction (Sect. 6). Related work overview (Sect. 7) and conclusions close the paper (Sect. 8).

2 Two-Level Change Realization Framework

In our earlier work [1, 13], we proposed a two-level aspect-oriented change realization framework. Changes come in the form of change requests each of which may consist of several changes. A change is understood there as a requirement focused on a particular issue that is in domain terminology perceived as indivisible.

Given a particular change, a developer determines the domain specific change type that corresponds to it. Domain specific change types represent abstractions and generalizations of changes expressed in the terminology of the particular domain. A developer gets a clue to the change realization from the cataloged mappings of domain specific changes to generally applicable change types.

Each generally applicable change type provides an example code of its realization. It can also be a kind of an aspect-oriented design pattern or a domain specific change can even be directly mapped to one or more aspect-oriented design patterns.

As an example, consider some changes in the general affiliate marketing software purchased by a merchant who runs his online music shop to advertise at third party web sites (denoted as affiliates).[1] This general affiliate marketing software tracks customer clicks on the merchant's commercials (e.g., banners) placed in affiliate sites and whether they led to buying goods from the merchant in which case the affiliate who referred the sale would get the provision.

Figure 1 shows two particular change requests with one of them demanding a change denoted as Newsletter Sign Up: integration of the affiliate marketing software with the third party newsletter used by the merchant, so that every affiliate would be a member of the newsletter. When an affiliate signs up to the affiliate marketing software, he should be signed up to the newsletter, too. Upon deleting his account, the affiliate should be removed from the newsletter, too.

This is an instance of the One Way Integration change type [1], one of web application domain specific change types. Its essence is the one way notification: the integrating application notifies the integrated application of relevant events. In this case, such events are the affiliate sign up and affiliate account deletion.

[1] This is an extended scenario originally published in our earlier work [1, 13].

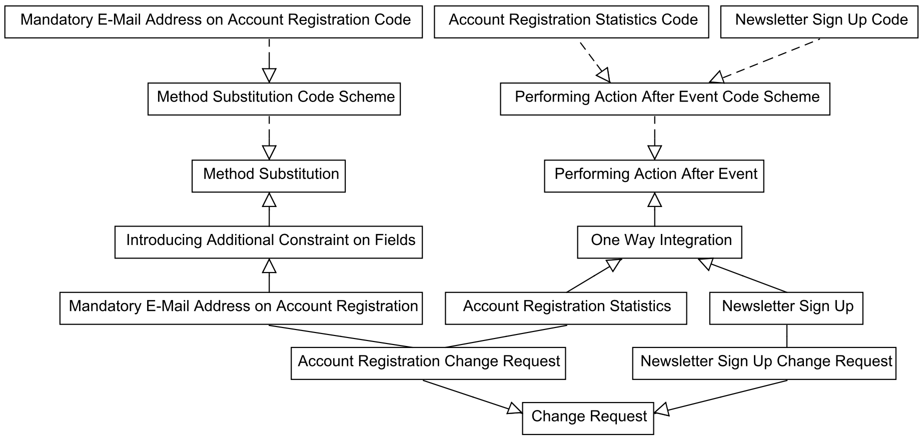

Mandatory E-Mail Address on Account Registration Code
Account Registration Statistics Code
Newsletter Sign Up Code
Method Substitution Code Scheme
Performing Action After Event Code Scheme
Method Substitution
Performing Action After Event
Introducing Additional Constraint on Fields
One Way Integration
Mandatory E-Mail Address on Account Registration
Account Registration Statistics
Newsletter Sign Up
Account Registration Change Request
Newsletter Sign Up Change Request
Change Request

Fig. 1. Some changes in the affiliate marketing software.

The catalog of changes [13] would point us to the Performing Action After Event generally applicable change type. As follows from its name, it describes how to implement an action after an event in general. Since events are actually represented by methods, the desired action can be implemented in an after advice [1]:

```
public aspect PerformActionAfterEvent {
   pointcut methodCalls(TargetClass t, int a): . . .;
   after(/* captured arguments */): methodCalls(/* captured arguments */) {
      performAction(/* captured arguments */);
   }
   private void performAction(/* arguments */) { /* action logic */ }
}
```

The after advice executes after the captured method calls. The actual action is implemented as the performAction() method called by the advice.

To implement the newsletter sign up change, in the after advice we will make a post to the newsletter sign up/sign out script and pass it the e-mail address and name of the newly signed-up or deleted affiliate.

There is another change in Fig. 1 depicted as an instance of One Way Integration: Account Registration Statistics. This change belongs to another change request comprising of changes related to account registration. Its intention is to gain statistical information about the affiliate registrations and pass to the corresponding application to perform its evaluation.

The change request related to account registration includes one more change: Mandatory E-Mail Address on Account Registration. The aim of this change is to prevent attempts to register without providing the e-mail address. This is actually an instance of Introducing Additional Constraint on Fields. There are several generally applicable change types that can be used to realize this change and choosing among them depends on the implementation of the functionality to be changed. Besides Performing Action After Event and Additional Parameter Checking, but if we assume no form validation mechanism is present, even the most general Method Substitution[2] can be used to capture method calls:

```
public aspect MethodSubstition {
   pointcut methodCalls(TargetClass t, int a): . . .;
   ReturnType around(TargetClass t, int a): methodCalls(t, a) {
      if (. . .) {
         . . . } // the new method logic
      else
         proceed(t, a);
   }
}
```

3 Generally Applicable Change Types as Paradigms

Generally applicable change types are independent of the application domain and may even apply to different aspect-oriented languages and frameworks (with an adapted code scheme, of course). The expected number of generally applicable changes that would cover all significant situations is not high. In our experiments, we managed to cope with all situations using only six generally applicable change types.

On the other hand, in the domain of web applications, we identified eleven application specific changes, yet having it only partially covered. Each such change requires a thorough exploration in order to discover all possible realizations by generally applicable changes and design patterns with conditions for their use, and it is not likely that someone would be willing to invest effort into developing a catalog of changes apart of the momentarily needs.

The problem of selecting a suitable generally applicable change type resembles the problem of the selection of a paradigm suitable to implement a particular application domain concept, which is a subject of multi-paradigm approaches [10]. Here, we will consider *multi-paradigm design with feature modeling* ($\mathrm{MPD_{FM}}$), which is based on an adapted Czarnecki–Eisenecker [5] feature modeling notation [11].

Section 3.1 presents basic concepts of $\mathrm{MPD_{FM}}$. In Sect. 3.2 and 3.3, paradigm models of Method Substitution and Performing Action After Event will be introduced.

3.1 Multi-Paradigm Design with Feature Modeling

In $\mathrm{MPD_{FM}}$, paradigms are understood as *solution domain concepts* that correspond to programming language mechanisms (like inheritance or class). Such

[2] which we haven't considered originally [13] as a possible realization of Introducing Additional Constraint on Fields

paradigms are being denoted as small-scale to distinguish them from the common concept of the (large-scale) paradigm as a particular approach to programming (like object-oriented or procedural programming) [12].

In MPD_{FM}, feature modeling is used to express paradigms. A feature model consists of a set of feature diagrams, information associated with concepts and features, and constraints and default dependency rules associated with feature diagrams. A feature diagram is usually understood as a directed tree whose root represents a concept being modeled and the rest of the nodes represent its features [15].

The features may be common to all concept instances (feature configurations) or variable, in which case they appear only in some of the concept instances. Features are selected in a process of concept instantiation. Those that have been selected are denoted as bound. The time at which this binding (or not binding) happens is called binding time. In paradigm modeling, the set of binding times is given by the solution model. In AspectJ we may distinguish among source time, compile time, load time, and runtime.

Each paradigm is considered to be a separate concept and as such presented in its own feature diagram that describes what is common to all paradigm instances (its applications), and what can vary, how it can vary, and when this happens. Consider the AspectJ aspect paradigm feature model shown in Fig. 2. Each aspect is named, which is modeled by a mandatory feature Name (indicated by a filled circle ended edge). The aspect paradigm articulates related structure and behavior that crosscuts otherwise possibly unrelated types. This is modeled by optional features Inter-Type Declarations ®, Advices ®, and Pointcuts ® (indicated by empty circle ended edges). These features are references to equally named auxiliary concepts that represent plural forms of respective concepts that actually represent paradigms in their own right (and their own feature models [12]). To achieve its intent, an aspect may—similarly to a class—employ Methods ®, whereas the method is yet another paradigm, and Fields.

An aspect in AspectJ is instantiated automatically by occurrence of the join points it addresses in accordance with Instantiation Policy which is either Singleton, Per Object, or Per Control Flow. The features that represent different instantiation policies are mandatory alternative features (indicated by an arc over mandatory features), which means that exactly one of them must selected. An aspect can be Abstract, in which case it can't be instantiated, so it can't have Instantiation Policy either, which is again modeled by mandatory alternative features.

An aspect can be declared to be Static or Final. It doesn't have to be none of these two, but it can't both either, which is modeled by optional alternative features of which only one may be selected (indicated by an arc over optional features). An aspect can also be Privileged over other aspects and it has its type of Access ®, which is modeled as another reference to a separately expressed auxiliary concept. The constraint associated with the aspect paradigm feature diagram means that the aspect is either Final, or Abstract. All the features in the Aspect paradigm are bound at source time.

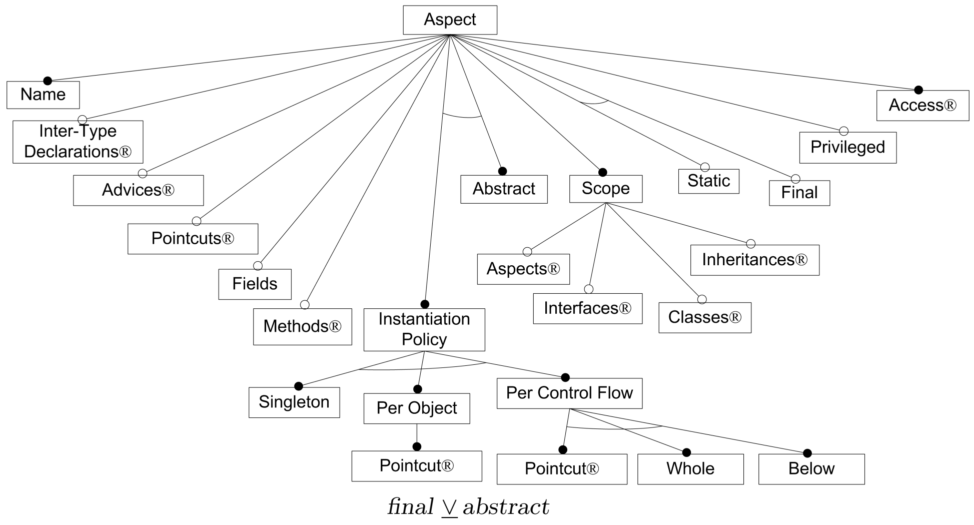


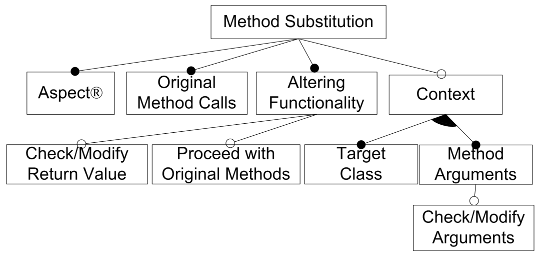
Fig. 2. The AspectJ aspect paradigm (adopted from [12]).

Feature modeling is applied also to the application domain. The two feature models, the application and solution domain one, enter the process called *transformational analysis* in which application to solution domain mapping is being established. This mapping is expressed in the form of yet another feature model consisting of the paradigm instances annotated with the information about corresponding application domain concepts and features which determines the code skeleton.

Generally applicable changes may be seen as a kind of conceptually higher language mechanisms and modeled as paradigms in the sense of MPD_{FM}. From the viewpoint of multi-paradigm design with feature modeling [12], change types represent directly usable (they do not need an "envelope" paradigm like an advice needs an aspect) relationship paradigms (they relate changing functionality with changed functionality). We will consider the two change types we presented in Sect. 2.

3.2 Method Substitution

Figure 3 shows the Method Substitution change type feature model. All the features have source time binding. This change type enables to capture calls to methods (Original Method Calls) with or without the context (Context) and to alter the functionality they implement by the additional functionality it provides (Altering Functionality) which includes the possibility of affecting the arguments (Check/Modify Return Arguments) or return value (Check/Modify Return Value), or even blocking the functionality of the methods whose calls have been captured altogether (Proceed with Original Methods).



Constraints:
Aspect.Pointcut
Aspect.Advice.Around

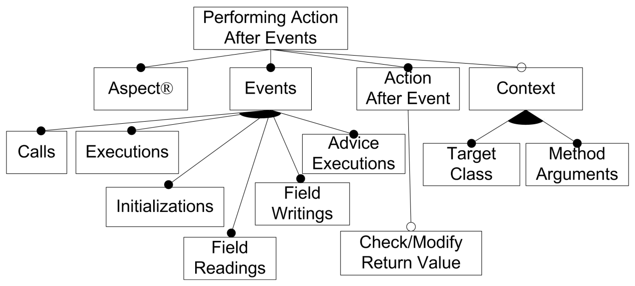
Fig. 3. Method Substitution.

Note the Context feature subfeatures. They are or-features, which means at least one them has to be selected.

Method Substitution is implemented by an aspect (Aspect Ⓡ) with a pointcut specifying the calls to the methods to be altered by an around advice, which is expressed by the constraints associated with its feature diagram (displayed in Fig. 3, too).

3.3 Performing Action After Event

Figure 4 shows the Performing Action After Event change type feature model. All the features have source time binding. This change type is used when additional actions (Actions After Event) are needed after some events (Events) of method calls or executions, initialization, field reading or writing, or advice execution (modeled as or-features mentioned in the previous section) taking or not into account their context (Context).



Constraints:
Aspect.Pointcut
Aspect.Advice.After

Fig. 4. Performing Action After Event.

Performing Action After Event is implemented by an aspect (Aspect Ⓡ) with a pointcut specifying the events and an after advice of this pointcut used to perform the desired actions, which is expressed by the constraints associated with its feature diagram (displayed in Fig. 4, too).

4 Application Feature Model

We will present how change realizations can be expressed in the application feature model on our running example of affiliate tracking software (introduced in Sect. 2).

4.1 Feature Model of Changes

In our affiliate marketing example, we may consider the following changes:

- SMTP Server Backup A/B —to have a backup server for sending notifications (with two different implementations, A and B)
- Newsletter Sign Up —to sign up an affiliate to a newsletter when he signs up to the tracking software
- Account Registration Constraint —to check whether the affiliate who wants to register submitted a valid e-mail address
- Restricted Administrator Account —to create an account with a restriction of using some resources
- Hide Options Unavailable to Restricted Administrator —to restrict the user interface
- User Name Display Change — to adapt the order of displaying the first name and surname
- Account Registration Statistics —to gain statistical information about the affiliate registrations

These changes are captured in the initial feature diagram presented in Fig. 5. The concept we model is our affiliate marketing software.[3] All the changes are modeled as optional features as they may, but don't have to be applied. We may consider the possibility of having different realizations of a change of which only one may be applied. This is expressed by alternative features. In the example, no *Affiliate Marketing* instance can contain both *SMTP Server Backup A* and *SMTP Server Backup B* [13].

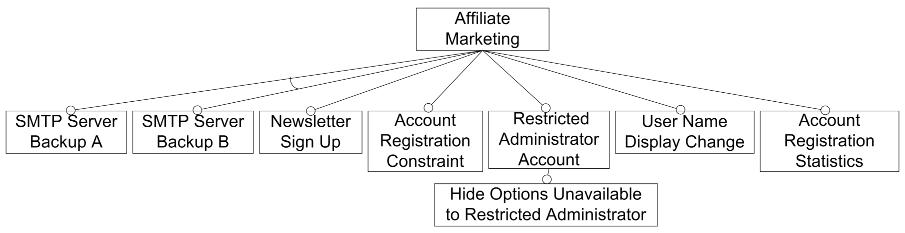

Affiliate Marketing
SMTP Server Backup A
SMTP Server Backup B
Newsletter Sign Up
Account Registration Constraint
Restricted Administrator Account
User Name Display Change
Account Registration Statistics
Hide Options Unavailable to Restricted Administrator

Fig. 5. Changes in the affiliate marketing software.

[3] In general, there may be several concepts that represent the application domain.

Some change realizations make sense only in the context of some other change realizations. In other words, such a change realization requires the other change realizations. In our scenario, hiding options unavailable to a restricted administrator makes sense only if we have introduced a restricted administrator account. This is modeled by having Hide Options Unavailable to Restricted Administrator to be a subfeature of Restricted Administrator Account. For a subfeature to be included in a concept instance, its parent feature must be included, too.

The feature–subfeature relationship represents a direct dependency between two features. Such dependency can be an early indication of a possible interaction between change realizations. However, with alternatives no interaction can occur because an application instance can contain only one change realization.

4.2 Partial Feature Model Creation

Additional dependencies among changes can be discovered by the underlying system exploration to which the changes are introduced. To achieve this goal it is essential to know a model of the system itself. Modeling of the whole system can be a difficult and time consuming task. In some cases it could be possible to find new dependencies without the need to explore the whole feature model of the system. Instead, it is possible to construct a partial feature model in which new possible dependencies could be discovered.

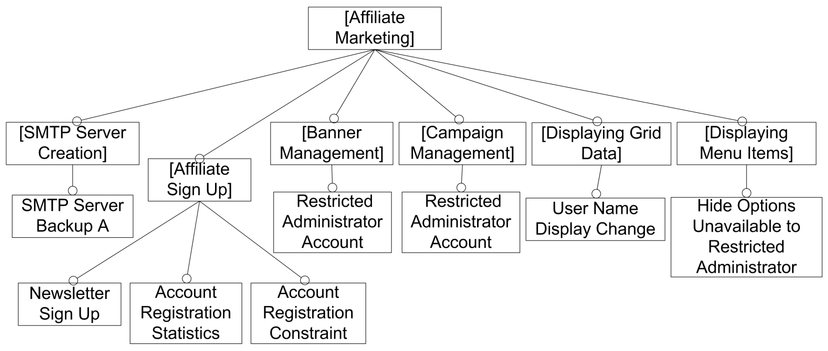
The process of constructing a partial feature model is based on the feature model in which aspect-oriented change realizations are represented by variable features that extend the existing system represented by a concept node as an abstract representation of the underlying software system. The potential dependencies of change realizations are hidden inside of it. In order to reveal them, we must factor out concrete features from the concept. Starting at the features that represent change realizations (leaves), we proceed bottom up trying to identify their parent features until related features become grouped in common subtrees [13].

In feature model from Fig. 5, we attempt to identify parent features of the change realization features as the features of the underlying system that are affected by them. The result of this is presented in Fig. 6. We found that:

– SMTP Server Backup A affects the SMTP Server Creation feature
– Newsletter Sign Up, Account Registration Statistics, and Account Registration Constraint change affect Affiliate Sign Up
– Restricted Administrator Account affects Banner Management and Campaign Management
– User Name Display Change affects Displaying Grid Data

All these newly identified features are open because we are aware of the incompleteness of the sets of their subfeatures.

At this stage, it is possible to identify potential locations at which the interaction may occur. Such locations are represented as features of the system to which changes are introduced.



Constraints:

Hide Operations Unavailable to Restricted Administrator ⇒Restricted Administration Account

Fig. 6. A partial feature model of the affiliate marketing application.

The highest probability of interaction is among sibling features (direct subfeatures of the same parent feature) which are potentially interdependent. This is caused by the fact that changes represented by such features usually use the same or similar pointcuts which can indeed lead to unwanted interaction. Such locations should represent primary targets of evaluation during transformational analysis, which is presented in the following section.

Interaction can occur also between indirect siblings or non-sibling features. However, with the increasing distance between features (that represent changes), the probability of interaction decreases.

5 Transformational Analysis

The input to transformational analysis in multi-paradigm design with feature modeling [12] are two feature models: the application domain one and the solution domain one. Transformational analysis is a process of finding the correspondence and establishing the mapping between the application and solution domain concepts. It is performed as a paradigm instantiation over application domain concepts at source time in which a paradigm is being instantiated in a bottom-up fashion with inclusion of some of the paradigm nodes being stipulated by the mapping of the nodes of one or more application domain concepts to them in order to ensure the paradigm instances correspond to these application domain concepts. The output of transformational analysis is a set of paradigm instances annotated with application domain feature model concepts and features that define code skeleton.

Transformational analysis can be used to determine how changes are to be realized. For this, we need a feature model of the application, either whole, or at least partial (see Sect. 4.2). The solution domain model would be represented by a paradigm model of the target aspect-oriented language or framework extended with generally applicable change types modeled as paradigms.

The original process of transformational analysis has to be modified to take into account change types as paradigms. They have to be considered primarily and only if they do not provide a satisfactory solution, solution domain paradigms should be considered. If this happens, the concept probably represents a new change type. In this case, original AspectJ direct paradigms should be chosen to be instantiated over this concept, which is covered in the original transformational analysis process [12].

Changes are considered to be application domain concepts. Since they are modeled as single nodes, i.e. without subfeatures, the bottom-up instantiation of paradigms over them provides no help in overcoming of the conceptual gap between changes and change types. On the other hand, the solution space is narrowed as it is known in advance that each change will be implemented as an aspect. Moreover, the actual implementation of other application domain features is known, so we can rely on them.

As an example, we will consider transformational analysis of several changes in the affiliate marketing software (introduced in Sect. 4.1) with the AspectJ paradigm model [9] extended by feature models of the generally applicable change types (see Sect. 3) as a solution domain.

The Restricted Administrator Account change provides an additional check of access rights upon execution of specified methods. Methods should be executed only if access is granted. This scenario suites best to the Method Substitution change type which can control the execution of selected methods. Moreover, in case of access violation method substitution can provide subroutines to display error message or log access violation event. Figure 7 represents the transformation of the Restricted Administrator Account change. Target Class and Method Arguments were used to capture additional context which is needed by Proceed with Original Methods when access is granted. Note that Banner Management and Campaigns Management features were mapped to Original Method Calls. Such association means that this change will affect the behavior represented by these features. Realizing and capturing such associations is crucial to change interaction evaluation (discussed in the next section).

Transformational analysis of the Account Registration Constraint would be very similar. Again, the Method Substitution paradigm would be used. Original Method Calls would be mapped to Affiliate Sign Up and the original method will be executed by the Proceed with Original Methods only if a valid e-mail address was provided.

The Newsletter Sign Up change adds a new affiliate to the existing list of newsletter recipients. This can be best realized as Performing Action After Event (Fig. 8). In this case, the Events feature is mapped to Affiliate Sign Up which represents the execution of the affiliate sign up method. Through Method Arguments, the data about a new affiliate can be accessed. From the accessed data, the e-mail address of the affiliate can be retrieved. Finally, the e-mail address

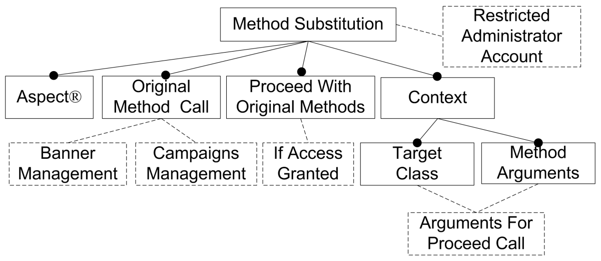


Fig. 7. Transformational analysis of the Restricted User Account change.

is added to the newsletter recipient list through Action After Event. A similar transformation would apply to the Registration Statistic change.

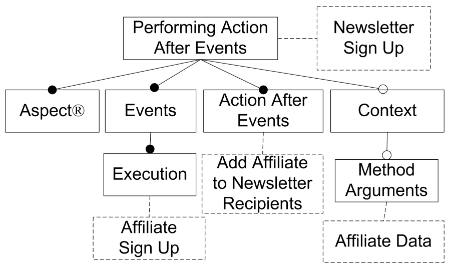


Fig. 8. Transformational analysis of the Newsletter Sign Up change.

6 Change Interaction

Change realizations can interact: they may be mutually dependent or some change realizations may depend on the parts of the underlying system affected by other change realizations [13]. It has been shown how the application domain feature model can be analyzed to identify such interactions [14]. Transformational analysis results can improve this identifications.

From the change interaction point of view, mappings to features that represent target functionality (join points) affected by changes are very important. The highest probability of interaction is when several changes affect the same target functionality. Such situations could be identified in part already during the creation of a partial feature model, but transformational analysis can reveal more details needed to enable avoiding the interaction of change realizations.

Consider, for example, the Newsletter Sign Up and Account Registration Statistics changes. Despite they share the target functionality (Affiliate Sign Up), no interaction occurs. This is because both changes are realized using the Performing Action After Event paradigm which employs an **after**() advice. In such a situation, it is important to evaluate whether the execution order of the advices is significant. In this particular case, the order is insignificant.

The Account Registration Constraint change represents a potential source of interaction with Newsletter Sign Up and Account Registration Statistics because it also targets the same functionality. This change is realized using the Method Substitution paradigm through which it can disable the execution of the method that registers a new affiliate. If the Newsletter Sign Up and Account Registration Statistics change realizations rely on method executions, not calls, i.e. they employ an **execution**() pointcut, no interaction occurs. On the other hand, if the realizations of these changes would rely on method calls, i.e. they would employ a **call**() pointcut, their advices would be executed even if the registration method haven't been executed, which is an undesirable system behavior.

In most cases, the interaction can be solved by adapting change realizations. Unsolvable change interaction should be indicated in the application domain model by constraints that will prevent affected changes of occurring together in any application configuration.

7 Related Work

The impact of changes implemented by aspects has been studied using slicing in concern slice dependency graphs [7]. Application domain feature model can be derived from concern slice dependency graphs [8] [8]. Concern slice dependency graphs provide in part also a dynamic view of change interaction that could be expressed using a dedicated notation (such as UML state machine or activity diagrams) and provided along with the feature model covering the structural view.

Applying feature modeling to maintain change dependencies is similar to constraints and preferences proposed in SIO software configuration management system [4].

Even if the original application haven't been a part of a product line, changes modeled as features of the original application tend to form a kind of a product line out of the original application. This could be seen as a kind of evolutionary development of a new product line [2].

Aspect-oriented change realization in general is related to change-based approaches in version control. It targets the problem of the lack of programming language awareness in changes realized in such approaches [6].

8 Conclusions and Further Work

The work reported here is a part of our ongoing efforts of comprehensively covering aspect-oriented change realization whose aim is to enable change realization in a modular, pluggable, and reusable way.

In this paper, we extended the original idea of having two-level change type framework to facilitate easier aspect-oriented change realization by enabling direct change manipulation using multi-paradigm design with feature modeling

(MPD_{FM}). There, we deal with generally applicable change types as (small-scale) paradigms.

We introduced the paradigm models of the Method Substitution and Performing Action After Event change types. We also developed paradigm models of other change types not presented in this paper such as Enumeration Modification with Additional Return Value Checking/Modification, Additional Return Value Checking/Modification, Additional Parameter Checking or Performing Action After Event, and Class Exchange.

We modified the process of the original transformational analysis in MPD_{FM} to include change types. We demonstrated how such transformational analysis can help in identifying the details of change interaction.

Our further work includes investigating the possibility of extending feature models of changes by expressing both application-specific and generally applicable changes in the Theme notation of aspect-oriented analysis and design [3]. We also prepare further evaluation studies.

Acknowledgements The work was supported by the Scientific Grant Agency of Slovak Republic (VEGA) grant No. VG 1/0508/09.

References

[1] M. Bebjak, V. Vranić, and P. Dolog. Evolution of web applications with aspect-oriented design patterns. In M. Brambilla and E. Mendes, editors, *Proc. of ICWE 2007 Workshops, 2nd International Workshop on Adaptation and Evolution in Web Systems Engineering, AEWSE 2007, in conjunction with 7th International Conference on Web Engineering, ICWE 2007*, pages 80–86, Como, Italy, July 2007.

[2] J. Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. Addison-Wesley, 2000.

[3] S. Clarke and E. Baniassad. *Aspect-Oriented Analysis and Design: The Theme Approach*. Addison-Wesley, 2005.

[4] R. Conradi and B. Westfechtel. Version models for software configuration management. *ACM Computing Surveys*, 30(2):232–282, June 1998.

[5] K. Czarnecki and U. W. Eisenecker. *Generative Programing: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[6] P. Dolog, V. Vranić, and M. Bieliková. Representing change by aspect. *ACM SIGPLAN Notices*, 36(12):77–83, Dec. 2001.

[7] S. Khan and A. Rashid. Analysing requirements dependencies and change impact using concern slicing. In *Proc. of Aspects, Dependencies, and Interactions Workshop (affiliated to ECOOP 2008)*, Nantes, France, July 2006.

[8] R. Menkyna. Dealing with interaction of aspect-oriented change realizations using feature modeling. In M. Bieliková, editor, *Proc. of 5th Student research Conference in Informatics and Information Technologies , IIT.SRC 2009*, Bratislava, Slovakia, Apr. 2009.

[9] V. Vranić. AspectJ paradigm model: A basis for multi-paradigm design for AspectJ. In J. Bosch, editor, *Proc. of 3rd International Conference on Generative and Component-Based Software Engineering (GCSE 2001)*, LNCS 2186, pages 48–57, Erfurt, Germany, Sept. 2001. Springer.

[10] V. Vranić. Towards multi-paradigm software development. *Journal of Computing and Information Technology (CIT)*, 10(2):133–147, 2002.

[11] V. Vranić. Reconciling feature modeling: A feature modeling metamodel. In M. Weske and P. Liggsmeyer, editors, *Proc. of 5th Annual International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a Networked World (Net.ObjectDays 2004)*, LNCS 3263, pages 122–137, Erfurt, Germany, Sept. 2004. Springer.

[12] V. Vranić. Multi-paradigm design with feature modeling. *Computer Science and Information Systems Journal (ComSIS)*, 2(1):79–102, June 2005.

[13] V. Vranić, M. Bebjak, R. Menkyna, and P. Dolog. Developing applications with aspect-oriented change realization. In *Proc. of 3rd IFIP TC2 Central and East European Conference on Software Engineering Techniques CEE-SET 2008*, LNCS, Brno, Czech Republic, Oct. 2008. Springer. Postproceedings, to appear.

[14] V. Vranić, R. Menkyna, M. Bebjak, and P. Dolog. Aspect-oriented change realizations and their interaction. Submitted to e-Informatica Software Engineering Journal, CEE-SET 2009 special issue.

[15] V. Vranić and M. Šípka. Binding time based concept instantiation in feature modeling. In M. Morisio, editor, *Proc. of 9th International Conference on Software Reuse (ICSR 2006)*, LNCS 4039, pages 407–410, Turin, Italy, June 2006. Springer.

Appendix E

Aspect-Oriented Change Realizations and Their Interaction

This appendix contains:

> Valentino Vranić, Michal Bebjak, Radoslav Menkyna, and Peter Dolog. Submitted to *e-Informatica Software Engineering Journal*, March 2009.

The paper represents special extended version of paper from appendix F. It was submitted to e-Informatica Software Engineering Journal, March 2009. Currently a review process is ongoing. My contribution to this paper is approximately 35 %.

e-Informatica Software Engineering Journal, Volume 1, Issue 1, 2007

Aspect-Oriented Change Realizations and Their Interaction

Valentino Vranić[*], Radoslav Menkyna[*], Michal Bebjak[*], Peter Dolog[**]

[*]*Institute of Informatics and Software Engineering, Faculty of Informatics and Information Technologies, Slovak University of Technology in Bratislava, Slovakia*

[**]*Department of Computer Science, Aalborg University, Denmark*

vranic@fiit.stuba.sk, radu@ynet.sk, mbebjak@gmail.com, dolog@cs.aau.dk

Abstract

With aspect-oriented programming, changes can be treated explicitly and directly at the programming language level. An approach to aspect-oriented change realization based on a two-level change type model is presented in this paper. In this approach, aspect-oriented change realizations are mainly based on aspect-oriented design patterns or themselves constitute pattern-like forms in connection to which domain independent change types can be identified. However, it is more convenient to plan changes in a domain specific manner. Domain specific change types can be seen as subtypes of generally applicable change types. These relationships can be maintained in a form of a catalog. Some changes can actually affect existing aspect-oriented change realizations, which can be solved by adapting the existing change implementation or by implementing an aspect-oriented change realization of the existing change without having to modify its source code. As demonstrated partially by the approach evaluation, the problem of change interaction may be avoided to a large extent by using appropriate aspect-oriented development tools, but for a large number of changes, dependencies between them have to be tracked. Constructing partial feature models in which changes are represented by variable features is sufficient to discover indirect change dependencies that may lead to change interaction.

1 Introduction

Change realization consumes enormous effort and time during software evolution. Once implemented, changes get lost in the code. While individual code modifications are usually tracked by a version control tool, the logic of a change as a whole vanishes without a proper support in the programming language itself.

By its capability to separate crosscutting concerns, aspect-oriented programming enables to deal with change explicitly and directly at programming language level. Changes implemented this way are pluggable and—to the great extent—reapplicable to similar applications, such as applications from the same product line.

Customization of web applications represents a prominent example of that kind. In customization, a general application is being adapted to the client's needs by a series of changes. With each new version of the base application, all the changes have to be applied to it. In many occasions, the difference between the new and old application does not affect the structure of changes, so if changes have been implemented using aspect-oriented programming, they can be simply included into the new application build without any additional effort.

Even conventionally realized changes may interact, i.e. they may be mutually dependent or some change realizations may depend on the parts of the underlying system affected by other change realizations. This is even more remarkable in aspect-oriented change realization due to pervasiveness of aspect-oriented programming as such.

We have already reported briefly our initial work in change realization using aspect-oriented programming [1]. In this paper,[1] we present our improved view of the approach to change realization based on a two-level change type model. Section 2 presents our approach to aspect-oriented change realization. Section 3 describes briefly the change types we have discovered so far in the web application domain. Section 4 discusses how to deal with a change of a change. Section 5 proposes a feature modeling based approach of dealing with change interaction. Section 6 describes the approach evaluation and outlooks for tool support. Section 7 discusses related work. Section 8 presents conclusions and directions of further work.

2 Changes as Crosscutting Requirements

A change is initiated by a change request made by a user or some other stakeholder. Change requests are specified in domain notions similarly as initial requirements are. A change request tends to be focused, but it often consists of several different—though usually interrelated—requirements that specify actual changes to be realized. By decomposing a change request into individual changes and by abstracting the essence out of each such change while generalizing it at the same time, a change type applicable to a range of the applications that belong to the same domain can be defined.

We will present our approach by a series of examples on a common scenario.[2] Suppose a merchant who runs his online music shop purchases a general affiliate marketing software [11] to advertise at third party web sites denoted as affiliates. In a simplified schema of affiliate marketing, a customer visits an affiliate's site which refers him to the merchant's site. When he buys something from the merchant, the provision is given to the affiliate who referred the sale. A general affiliate marketing software enables to manage affiliates, track sales referred by these affiliates, and compute provisions for referred sales. It is also able to send notifications about new sales, signed up affiliates, etc.

The general affiliate marketing software has to be adapted (customized), which involves a series of changes. We will assume the affiliate marketing software is written in Java, so we can use AspectJ, the most popular aspect-oriented language, which is based on Java, to implement some of these changes.

In the AspectJ style of aspect-oriented programming, the crosscutting concerns are captured in units called aspects. Aspects may contain fields and methods much the same way the usual Java classes do, but what makes possible for them to affect other code are genuine aspect-oriented constructs, namely: *pointcuts*, which specify the places in the code

[1]This paper represents an extended version of our paper presented at CEE-SET 2008 [28].
[2]This is an adapted scenario published in our earlier work [1].

to be affected, *advices*, which implement the additional behavior before, after, or instead of the captured *join point* (a well-defined place in the program execution)—most often method calls or executions—and *inter-type declarations*, which enable introduction of new members into types, as well as introduction of compilation warnings and errors.

2.1 Domain Specific Changes

One of the changes of the affiliate marketing software would be adding a backup SMTP server to ensure delivery of the notifications to users. Each time the affiliate marketing software needs to send a notification, it creates an instance of the SMTPServer class which handles the connection to the SMTP server.

An SMTP server is a kind of a resource that needs to be backed up, so in general, the type of the change we are talking about could be denoted as *Introducing Resource Backup*. This change type is still expressed in a domain specific way. We can clearly identify a crosscutting concern of maintaining a backup resource that has to be activated if the original one fails and implement this change in a single aspect without modifying the original code:

```
public class SMTPServerM extends SMTPServer {
   . . .
}
. . .
public aspect SMTPServerBackupA {
   public pointcut SMTPServerConstructor(URL url, String user, String password):
      call(SMTPServer.new(..)) && args (url, user, password);
   SMTPServer around(URL url, String user, String password):
      SMTPServerConstructor(url, user, password) {
      return getSMTPServerBackup(proceed(url, user, password));
   }
   private SMTPServer getSMTPServerBackup(SMTPServer obj) {
      if (obj.isConnected()) {
         return obj;
      }
      else {
         return new SMTPServerM(obj.getUrl(), obj.getUser(),
            obj.getPassword());
      }
   }
}
```

The **around**() advice captures constructor calls of the SMTPServer class and their arguments. This kind of advice takes complete control over the captured join point and its return clause, which is used in this example to control the type of the SMTP server being returned. The policy is implemented in the getSMTPServerBackup() method: if the original SMTP server can't be connected to, a backup SMTP server class SMTPServerM instance is created and returned.

We can also have another aspect—say SMTPServerBackupB—intended for another application configuration that would implement a different backup policy or simply instantiate a different backup SMTP server.

2.2 Generally Applicable Changes

Looking at this code and leaving aside SMTP servers and resources altogether, we notice that it actually performs a class exchange. This idea can be generalized and domain details abstracted out of it bringing us to the *Class Exchange* change type [1] which is based on the Cuckoo's Egg aspect-oriented design pattern [20]:

```
public class AnotherClass extends MyClass {
   . . .
}
. . .
public aspect MyClassSwapper {
   public pointcut myConstructors(): call(MyClass.new());
   Object around(): myConstructors() {
      return new AnotherClass();
   }
}
```

2.3 Applying a Change Type

It would be beneficial if the developer could get a hint on using the Cuckoo's Egg pattern based on the information that a resource backup had to be introduced. This could be achieved by maintaining a catalog of changes in which each domain specific change type would be defined as a specialization of one or more generally applicable changes.

When determining a change type to be applied, a developer chooses a particular change request, identifies individual changes in it, and determines their type. Figure 1 shows an example situation. Domain specific changes of the D1 and D2 type have been identified in the Change Request 1. From the previously identified and cataloged relationships between change types we would know their generally applicable change types are G1 and G2.

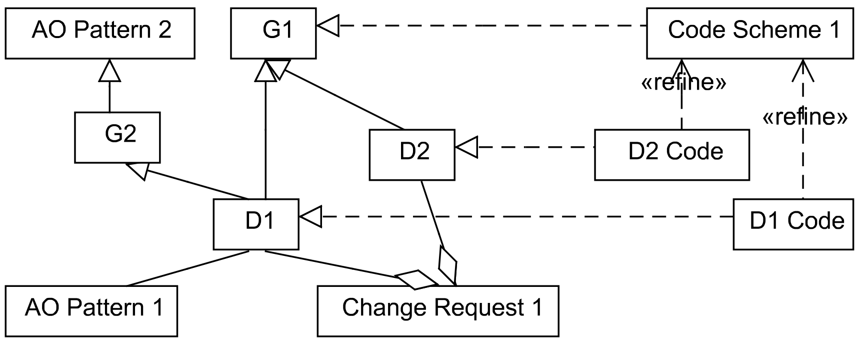


Figure 1: Generally applicable and domain specific changes.

A generally applicable change type can be a kind of an aspect-oriented design pattern (consider G2 and AO Pattern 2). A domain specific change realization can also be complemented by an aspect-oriented design pattern (or several ones), which is expressed by an association between them (consider D1 and AO Pattern 1).

Each generally applicable change has a known domain independent code scheme (G2's code scheme is omitted from the figure). This code scheme has to be adapted to the context of a particular domain specific change, which may be seen as a kind of refinement (consider D1 Code and D2 Code).

3 Catalog of Changes

To support the process of change selection, the catalog of changes is needed in which the generalization–specialization relationships between change types would be explicitly established. The following list sums up these relationships between change types we have identified in the web application domain (the domain specific change type is introduced first):

- One Way Integration: Performing Action After Event
- Two Way Integration: Performing Action After Event
- Adding Column to Grid: Performing Action After Event
- Removing Column from Grid: Method Substitution
- Altering Column Presentation in Grid: Method Substitution
- Adding Fields to Form: Enumeration Modification with Additional Return Value Checking/Modification
- Removing Fields from Form: Additional Return Value Checking/Modification
- Introducing Additional Constraint on Fields: Additional Parameter Checking or Performing Action After Event
- Introducing User Rights Management: Border Control with Method Substitution
- User Interface Restriction: Additional Return Value Checking/Modifications
- Introducing Resource Backup: Class Exchange

We have already described Introducing Resource Backup and the corresponding generally applicable change, Class Exchange. Here, we will briefly describe the rest of the domain specific change types we identified in the web application domain along with the corresponding generally applicable changes. The generally applicable change types are described where they are first mentioned to make sequential reading of this section easier. In a real catalog of changes, each change type would be described separately.

3.1 Integration Changes

Web applications often have to be integrated with other systems. Suppose that in our example the merchant wants to integrate the affiliate marketing software with the third party newsletter which he uses. Every affiliate should be a member of the newsletter. When an affiliate signs up to the affiliate marketing software, he should be signed up to the newsletter, too. Upon deleting his account, the affiliate should be removed from the newsletter, too.

This is a typical example of the *One Way Integration* change type [1]. Its essence is the one way notification: the integrating application notifies the integrated application of relevant events. In our case, such events are the affiliate sign-up and affiliate account deletion.

Such integration corresponds to the *Performing Action After Event* change type [1]. Since events are actually represented by methods, the desired action can be implemented in an after advice:

```
public aspect PerformActionAfterEvent {
   pointcut methodCalls(TargetClass t, int a): . . .;
   after(/∗ captured arguments ∗/): methodCalls(/∗ captured arguments ∗/) {
      performAction(/∗ captured arguments ∗/);
   }
   private void performAction(/∗ arguments ∗/) { /∗ action logic ∗/ }
}
```

The after advice executes after the captured method calls. The actual action is implemented as the performAction() method called by the advice.

To implement the one way integration, in the after advice we will make a post to the newsletter sign-up/sign-out script and pass it the e-mail address and name of the newly signed-up or deleted affiliate. We can seamlessly combine multiple one way integrations to integrate with several systems.

The *Two Way Integration* change type can be seen as a double One Way Integration. A typical example of such a change is data synchronization (e.g., synchronization of user accounts) across multiple systems. When a user changes his profile in one of the systems, these changes should be visible in all of them. In our example, introducing a forum for affiliates with synchronized user accounts for affiliate convenience would represent a Two Way Integration.

3.2 Introducing User Rights Management

In our affiliate marketing application, the marketing is managed by several co-workers with different roles. Therefore, its database has to be updated from an administrator account with limited permissions. A restricted administrator should not be able to decline or delete affiliates, nor modify the advertising campaigns and banners that have been integrated with the web sites of affiliates. This is an instance of the *Introducing User Rights Management* change type.

Suppose all the methods for managing campaigns and banners are located in the campaigns and banners packages. The calls to these methods can be viewed as a region prohibited to the restricted administrator. The Border Control design pattern [20] enables to partition an application into a series of regions implemented as pointcuts that can later be operated on by advices [1]:

```
pointcut prohibitedRegion(): (within(application.Proxy) && call(void *.*(..)))
   || (within(application.campaigns.+) && call(void *.*(..)))
   || within(application.banners.+)
   || call(void Affiliate.decline(..)) || call(void Affiliate.delete(..));
}
```

What we actually need is to substitute the calls to the methods in the region with our own code that will let the original methods execute only if the current user has sufficient rights. This can be achieved by applying the *Method Substitution* change type which is based on an around advice that enables to change or completely disable the execution of methods. The following pointcut captures all method calls of the method called method() belonging to the TargetClass class:

```
pointcut allmethodCalls(TargetClass t, int a):
   call(ReturnType TargetClass.method(..)) && target(t) && args(a);
```

Note that we capture method calls, not executions, which gives us the flexibility in constraining the method substitution logic by the context of the method call. The **call**() pointcut captures all the calls of TargetClass.method(), the **target**() pointcut is used to capture the reference to the target object, and the method arguments (if we need them) are captured by an **args**() pointcut. In the example code, we assume method() has one integer argument and capture it with this pointcut.

The following example captures the method() calls made within the control flow of any of the CallingClass methods:

```
pointcut specificmethodCalls(TargetClass t, int a):
   call(ReturnType TargetClass.method(a)) && target(t) && args(a)
   && cflow(call(* CallingClass.*(..)));
```

This embraces the calls made directly in these methods, but also any of the method() calls made further in the methods called directly or indirectly by the CallingClass methods.

By making an around advice on the specified method call capturing pointcut, we can create a new logic of the method to be substituted:

```
public aspect MethodSubstition {
   pointcut methodCalls(TargetClass t, int a): . . .;
   ReturnType around(TargetClass t, int a): methodCalls(t, a) {
      if (. . .) {
         . . . } // the new method logic
      else
         proceed(t, a);
   }
}
```

3.3 User Interface Restriction

It is quite annoying when a user sees, but can't access some options due to user rights restrictions. This requires a *User Interface Restriction* change type to be applied. We have created a similar situation in our example by a previous change implementation that introduced the restricted administrator (see Sect. 3.2). Since the restricted administrator can't access advertising campaigns and banners, he shouldn't see them in menu either.

Menu items are retrieved by a method and all we have to do to remove the banners and campaigns items is to modify the return value of this method. This may be achieved by applying a *Additional Return Value Checking/Modification* change which checks or modifies a method return value using an around advice:

```
public aspect AdditionalReturnValueProcessing {
   pointcut methodCalls(TargetClass t, int a): . . .;
   private ReturnType retValue;
   ReturnType around(): methodCalls(/* captured arguments */) {
      retValue = proceed(/* captured arguments */);
      processOutput(/* captured arguments */);
      return retValue;
   }
   private void processOutput(/* arguments */) {
      // processing logic
   }
}
```

In the around advice, we assign the original return value to the private attribute of the aspect. Afterwards, this value is processed by the processOutput() method and the result is returned by the around advice.

3.4 Grid Display Changes

It is often necessary to modify the way data are displayed or inserted. In web applications, data are often displayed in grids, and data input is usually realized via forms. Grids usually display the content of a database table or collation of data from multiple tables directly. Typical changes required on grid are adding columns, removing them, and modifying their presentation. A grid that is going to be modified must be implemented either as some kind of a reusable component or generated by row and cell processing methods. If the grid is hard coded for a specific view, it is difficult or even impossible to modify it using aspect-oriented techniques.

If the grid is implemented as a data driven component, we just have to modify the data passed to the grid. This corresponds to the Additional Return Value Checking/Modification change (see Sect. 3.3). If the grid is not a data driven component, it has to be provided at least with the methods for processing rows and cells.

Adding Column to Grid can be performed *after an event* of displaying the existing columns of the grid which brings us to the Performing Action After Event change type (see Sect. 3.1). Note that the database has to reflect the change, too. *Removing Column from Grid* requires a conditional execution of the method that displays cells, which may be realized as a Method Substitution change (see Sect. 3.2).

Alterations of a grid are often necessary due to software localization. For example, in Japan and Hungary, in contrast to most other countries, the surname is placed before the given names. The *Altering Column Presentation in Grid* change type requires preprocessing of all the data to be displayed in a grid before actually displaying them. This may be easily achieved by modifying the way the grid cells are rendered, which may be implemented again as a Method Substitution (see Sect. 3.2):

```
public aspect ChangeUserNameDisplay {
   pointcut displayCellCalls(String name, String value):
      call(void UserTable.displayCell(..)) || args(name, value);
   around(String name, String value): displayCellCalls(name, value) {
      if (name == "<the name of the column to be modified>") {
         . . . // display the modified column
      } else {
         proceed(name, value);
      }
   }
}
```

3.5 Input Form Changes

Similarly to tables, forms are often subject to modifications. Users often want to add or remove fields from forms or pose additional constraints on their input fields. Note that to be possible to modify forms using aspect-oriented programming they may not be hard coded in HTML, but generated by a method. Typically they are generated from a list of fields implemented by an enumeration.

Going back to our example, assume that the merchant wants to know the genre of the music which is promoted by his affiliates. We need to add the genre field to the generic affiliate sign-up form and his profile form to acquire the information about the genre to be promoted at different affiliate web sites. This is a change of the *Adding Fields to Form* type. To display the required information, we need to modify the affiliate table of the merchant panel to display genre in a new column. This can be realized by applying the Enumeration Modification change type to add the genre field along with already mentioned Additional Return Value Checking/Modification in order to modify the list of fields being returned (see Sect. 3.3).

The realization of the *Enumeration Modification* change type depends on the enumeration type implementation. Enumeration types are often represented as classes with a static field for each enumeration value. A single enumeration value type is represented as a class with a field that holds the actual (usually integer) value and its name. We add a new enumeration value by introducing the corresponding static field:

```
public aspect NewEnumType {
   public static EnumValueType EnumType.NEWVALUE =
      new EnumValueType(10, "<new value name>");
}
```

The fields in a form are generated according to the enumeration values. The list of enumeration values is typically accessible via a method provided by it. This method has to be addressed by an Additional Return Value Checking/Modification change.

For *Removing Fields from Form*, an Additional Return Value Checking/Modification change is sufficient. Actually, the enumeration value would still be included in the enumeration, but this would not affect the form generation.

If we want to introduce additional validations on form input fields in an application without a built-in validation, which constitutes an *Introducing Additional Constraint on Fields* change, an *Additional Parameter Checking* change can be applied to methods that process values submitted by the form. This change enables to introduce an additional validation or constraint on method arguments. For this, we have to specify a pointcut that will capture all the calls of the affected methods along with their context similarly as in Sect. 3.2. Their arguments will be checked by the check() method called from within an around advice which will throw WrongParamsException if they are not correct:

```
public aspect AdditionalParameterChecking {
  pointcut methodCalls(TargetClass t, int a): . . .;
  ReturnType around(/∗ arguments ∗/) throws WrongParamsException:
    methodCalls(/∗ arguments ∗/) {
    check(/∗ arguments ∗/);
    return proceed(/∗ arguments ∗/);
  }
  void check(/∗ arguments ∗/) throws WrongParamsException {
    if (arg1 != <desired value>)
      throw new WrongParamsException();
  }
}
```

Adding a new validator to an application that already has a built-in validation is realized by simply including it in the list of validators. This can be done by implementing the Performing Action After Event change (see Sect. 3.1), which would add the validator to the list of validators after the list initialization.

4 Changing a Change

Sooner or later there will be a need for a change whose realization will affect some of the already applied changes. There are two possibilities to deal with this situation: a new change can be implemented separately using aspect-oriented programming or the affected change source code could be modified directly. Either way, the changes remain separate from the rest of the application.

The possibility to implement a change of a change using aspect-oriented programming and without modifying the original change is given by the aspect-oriented programming language capabilities. Consider, for example, advices in AspectJ. They are unnamed, so can't be referred to directly. The primitive pointcut **adviceexecution**(), which captures execution of all advices, can be restricted by the **within**() pointcut to a given aspect, but if an aspect contains several advices, advices have to be annotated and accessed by the **@annotation**() pointcut, which was impossible in AspectJ versions that existed before Java was extended with annotations.

An interesting consequence of aspect-oriented change realization is the separation of crosscutting concerns in the application which improves its modularity (and thus makes easier further changes) and may be seen as a kind of aspect-oriented refactoring. For example, in our affiliate marketing application, the integration with a newsletter—identified as a kind of One Way Integration—actually was a separation of integration connection, which may be seen as a concern of its own. Even if these once separated concerns are further maintained by direct source code modification, the important thing is that they remain separate from the rest of the application. Implementing a change of a change using aspect-oriented programming and without modifying the original change is interesting mainly if it leads to separation of another crosscutting concern.

5 Capturing Change Interaction by Feature Models

Some change realizations can *interact*: they may be mutually dependent or some change realizations may depend on the parts of the underlying system affected by other change realizations. With increasing number of changes, change interaction can easily escalate into a serious problem: serious as *feature* interaction.

Change realizations in the sense of the approach presented so far actually resemble features as coherent pieces of functionality. Moreover, they are virtually pluggable and as such represent *variable* features. This brings us to feature modeling as an appropriate technique for managing variability in software development including variability among changes. This section will show how to model aspect-oriented changes using feature modeling.

5.1 Representing Change Realizations

There are several feature modeling notations [26] of which we will stick to a widely accepted and simple Czarnecki–Eisenecker basic notation [5]. Further in this section, we will show how feature modeling can be used to manage change interaction with elements of the notation explained as needed.

Aspect-oriented change realizations can be perceived as variable features that extend an existing system. Figure 2 shows the change realizations from our affiliate marketing scenario a feature diagram. A feature diagram is commonly represented as a tree whose root represents a concept being modeled. Our concept is our affiliate marketing software. All the changes are modeled as optional features (marked by an empty circle ended edges) that can but do not have to be included in a feature configuration—known also as concept instance—for it to be valid. Recall adding a backup SMTP server discussed in Sect. 2.1. We considered a possibility of having another realization of this change, but we don't want both realizations simultaneously. In the feature diagram, this is expressed by alternative features (marked by an arc), so no Affiliate Marketing instance will contain both SMTP Server Backup A and SMTP Server Backup B.

A change realization can be meaningful only in the context of another change realization. In other words, such a change realization requires the other change realization. In our scenario, hiding options unavailable to a restricted administrator makes sense only if we introduced a restricted administrator account (see Sect. 3.3 and 3.2). Thus, the Hide Options Unavailable to Restricted Administrator feature is a subfeature of the Restricted Administrator Account feature. For a subfeature to be included in a concept instance its parent feature must be included, too.

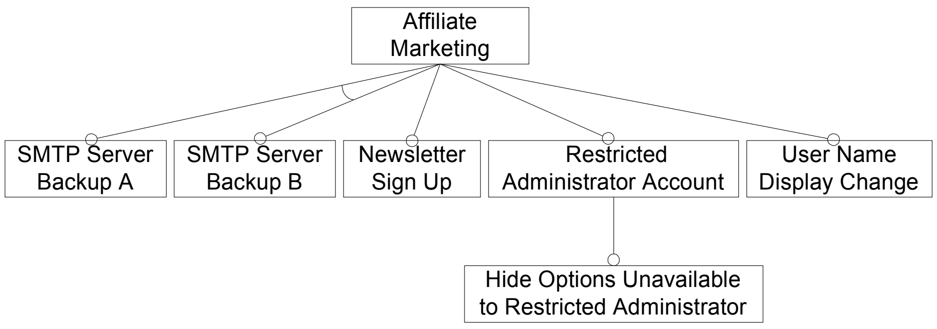


Figure 2: Affiliate marketing software change realizations in a feature diagram.

5.2 Identifying Direct Change Interactions

Direct change interactions can be identified in a feature diagram with change realizations modeled as features of the affected software concept. Each dependency among features represents a potential change interaction. A direct change interaction may occur among alternative features or a feature and its subfeatures: such changes may affect the common join points. In our affiliate marketing scenario, alternative SMTP backup server change realizations are an example of such changes. Determining whether changes really interact requires analysis of dependant feature semantics with respect to the implementation of the software being changed. This is beyond feature modeling capabilities.

Indirect feature dependencies may also represent potential change interactions. Additional dependencies among changes can be discovered by exploring the software to which the changes are introduced. For this, it is necessary to have a feature model of the software itself, which is seldom the case. Constructing a complete feature model can be too costly with respect to expected benefits for change interaction identification. However, only a part of the feature model that actually contains edges that connect the features under consideration is needed in order to reveal indirect dependencies among them.

5.3 Partial Feature Model Construction

The process of constructing partial feature model is based on the feature model in which aspect-oriented change realizations are represented by variable features that extend an existing system represented as a concept (see Sect. 5.1).

The concept node in this case is an abstract representation of the underlying software system. Potential dependencies of the change realizations are hidden inside of it. In order to reveal them, we must factor out concrete features from the concept. Starting at the features that represent change realizations (leaves) we proceed bottom up trying to identify their parent features until related changes are not grouped in common subtrees. Figure 3 depicts this process.

The process will be demonstrated on YonBan, a student project management system developed at Slovak University of Technology. We will consider the following changes in YonBan and their respective realizations indicated by generally applicable change types:

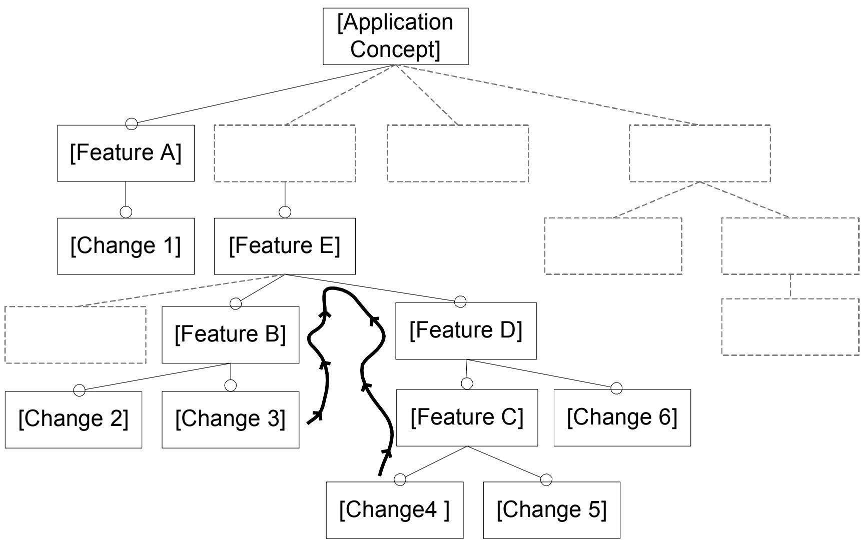


Figure 3: Constructing a partial feature model.

- Telephone Number Validating (realized as Performing Action After Event): to validate a telephone number the user has entered
- Telephone Number Formatting (realized as Additional Return Value Checking/Modification): to format a telephone number by adding country prefix
- Project Registration Statistics (realized as One Way Integration): to gain statistic information about the project registrations
- Project Registration Constraint (realized as Additional Parameter Checking/Modification): to check whether the student who wants to register a project has a valid e-mail address in his profile
- Exception Logging (realized as Performing Action After Event): to log the exceptions thrown during the program execution
- Name Formatting (realized as Method Substitution): to change the way how student names are formatted

These change realizations are captured in the initial feature diagram presented Fig. 4. Since there was no relevant information about direct dependencies among changes during their specification, there are no direct dependencies among the features that represent them either. The concept of the system as such is marked as open (indicated by square brackets), which means that new variable subfeatures are expected at it. This is so because we show only a part of the analyzed system knowing there are other features there.

Following this initial stage, we attempt to identify parent features of the change realization features as the features of the underlying system that are affected by them. Figure 5

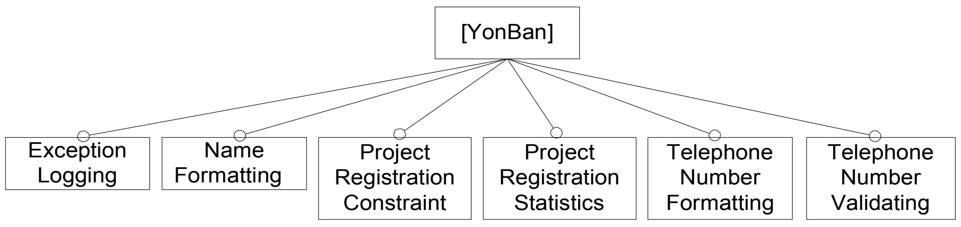

[YonBan]
Exception Logging
Name Formatting
Project Registration Constraint
Project Registration Statistics
Telephone Number Formatting
Telephone Number Validating

Figure 4: Initial stage of the YonBan partial feature model construction.

shows such changes identified in our case. We found that Name Formatting affects the Name Entering feature. Project Registration Statistic and Project Registration Constraint change User Registration. Telephone Number Formatting and Telephone Number Validating are changes of Telephone Number Entering. Exception Logging affects all the features in the application, so it remains a direct feature of the concept. All these newly identified features are open because we are aware of the incompleteness of their subfeature sets.

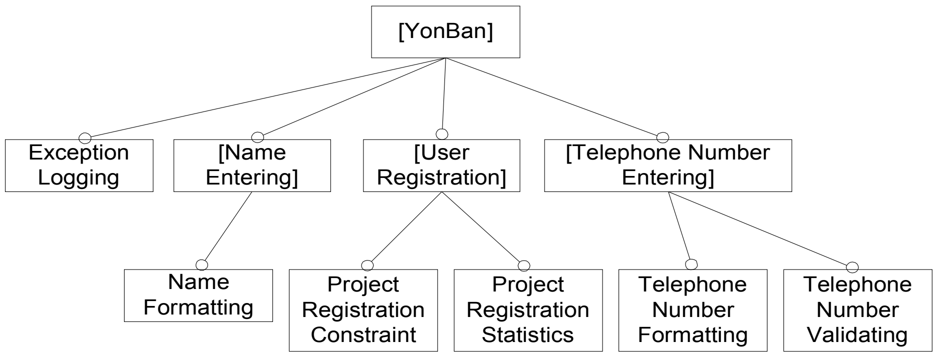

[YonBan]
Exception Logging
[Name Entering]
[User Registration]
[Telephone Number Entering]
Name Formatting
Project Registration Constraint
Project Registration Statistics
Telephone Number Formatting
Telephone Number Validating

Figure 5: Identifying parent features in YonBan partial feature model construction.

We continue this process until we are able to identify parent features or until all the changes are found in a common subtree of the feature diagram, whichever comes first. In our example, we reached this stage within the following—and thus last—iteration which is presented in Fig. 6: we realized that Telephone Number Entering is a part of User Registration.

5.4 Dependency Evaluation

Dependencies among change realization features in a partial feature model constitute potential change realization interactions. A careful analysis of the feature model can reveal dependencies we have overlooked during its construction.

Sibling features (direct subfeatures of the same parent feature) are potentially interdependent. This problem can occur also among the features that are—to say so—indirect siblings, so we have to analyze these, too. Speaking in terms of change implementation, the code that implements the parent feature altered by one of the sibling change features

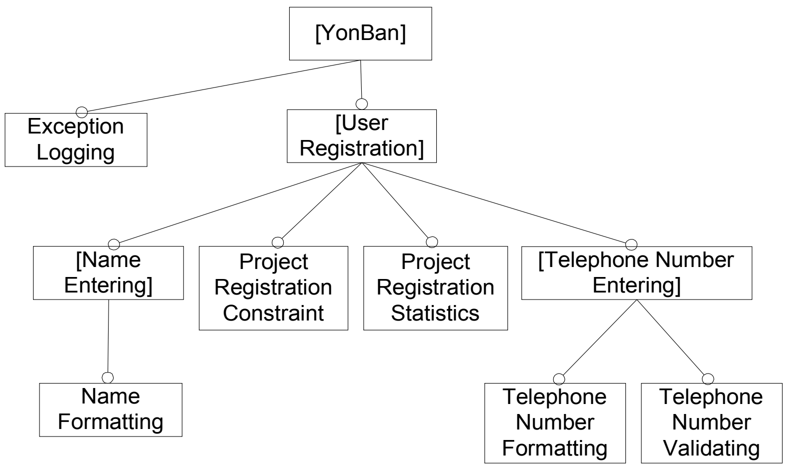

[YonBan]
Exception Logging
[User Registration]
[Name Entering]
Project Registration Constraint
Project Registration Statistics
[Telephone Number Entering]
Name Formatting
Telephone Number Formatting
Telephone Number Validating

Figure 6: The final YonBan partial feature model.

can be dependent on the code altered by another sibling change feature or vice versa. The feature model points us to the locations of potential interaction.

In our example, we have a partial feature model (recall Fig. 6) and we understand the way the changes should be implemented based on their type (see Sect. 5.3). Project Registration Constraint and Project Registration Statistic change are both direct subfeatures of User Registration. The two aspects that would implement these changes would advise the same project registration method, and this indeed can lead to interaction. In such cases, precedence of aspects should be set (in AspectJ, **dominates** inter-type declaration enables this). Another possible problem in this particular situation is that the Project Registration Constraint change can disable the execution of the project registration method. If the Project Registration Statistic change would use an **execution**() pointcut, everything would be all right. On the other hand, if the Project Registration Statistic change would use a **call**() pointcut, the registration statistic advice would be still executed even when the registration method would not be executed. This would cause an undesirable system behavior where also registrations canceled by Project Registration Constraint would be counted in statistic. The probability of a mistake when a **call**() pointcut is used instead of the **execution**() pointcut is higher if the Project Registration Statistic change would be added first.

Telephone Number Formatting and Telephone Number Validating are another example of direct subfeatures. In this case, the aspects that would implement these changes apply to different join points, so apparently, no interaction should occur. However, a detailed look uncovers that Telephone Number Formatting change alters the value which the Telephone Number Validating change has to validate. This introduces a kind of logical dependency and to this point the two changes interact. For instance, altering Telephone Number Formatting to format the number in a different way may require adapting Telephone Number Validating.

We saw that the dependencies between changes could be as complex as feature dependencies in feature modeling and accordingly represented by feature diagrams. For dependencies appearing among features without a common parent, additional constraints expressed as logical expressions [27] could be used. These constraints can be partly embedded into feature diagrams by allowing them to be directed acyclic graphs instead of just trees [10].

Some dependencies between changes may exhibit only recommending character, i.e. whether they are expected to be included or not included together, but their application remains meaningful either way. An example of this are features that belong to the same change request. Again, feature modeling can be used to model such dependencies with so-called default dependency rules that may also be represented by logical expressions [27].

6 Evaluation and Tool Support Outlooks

We have successfully applied the aspect-oriented approach to change realization to introduce changes into YonBan, the student project management system discussed in previous section. YonBan is based on J2EE, Spring, Hibernate, and Acegi frameworks. The YonBan architecture is based on the Inversion of Control principle and Model-View-Controller pattern.

We implemented all the changes listed in Sect. 5.3. No original code of the system had to be modified. Except in the case of project registration statistics and project registration constraint, which where well separated from the rest of the code, other changes would require extensive code modifications if they have had been implemented the conventional way.

As we discussed in Sect 5.4, we encountered one change interaction: between the telephone number formatting and validating. These two changes are interrelated—they would probably be part of one change request—so it comes as no surprise they affect the same method. However, no intervention was needed in the actual implementation.

We managed to implement the changes easily even without a dedicated tool, but to cope with a large number of changes, such a tool may become crucial. Even general aspect-oriented programming support tools—usually integrated with development environments—may be of some help in this. AJDT (AspectJ Development Tools) for Eclipse is a prominent example of such a tool. AJDT shows whether a particular code is affected by advices, the list of join points affected by each advice, and the order of advice execution, which all are important to track when multiple changes affect the same code. Advices that do not affect any join point are reported in compilation warnings, which may help detect pointcuts invalidated by direct modifications of the application base code such as identifier name changes or changes in method arguments.

A dedicated tool could provide a much more sophisticated support. A change implementation can consist of several aspects, classes, and interfaces, commonly denoted as types. The tool should keep a track of all the parts of a change. Some types may be shared among changes, so the tool should enable simple inclusion and exclusion of changes. This is related to change interaction, which can be addressed by feature modeling as we described in the previous section.

7 Related Work

The work presented in this paper is based on our initial efforts related to aspect-oriented change control [8] in which we related our approach to change-based approaches in version control. We concluded that the problem with change-based approaches that could be solved by aspect-oriented programming is the lack of programming language awareness in change realizations.

In our work on the evolution of web applications based on aspect-oriented design patterns and pattern-like forms [1], we reported the fundamentals of aspect-oriented change realizations based on the two level model of domain specific and generally applicable change types, as well as four particular change types: Class Exchange, Performing Action After Event, and One/Two Way Integration.

Applying feature modeling to maintain change dependencies (see Sect. 4) is similar to constraints and preferences proposed in SIO software configuration management system [4]. However, a version model for aspect dependency management [23] with appropriate aspect model that enables to control aspect recursion and stratification [2] would be needed as well.

We tend to regard changes as concerns, which is similar to the approach of facilitating configurability by separation of concerns in the source code [9]. This approach actually enables a kind of aspect-oriented programming on top of a versioning system. Parts of the code that belong to one concern need to be marked manually in the code. This enables to easily plug in or out concerns. However, the major drawback, besides having to manually mark the parts of concerns, is that—unlike in aspect-oriented programming—concerns remain tangled in code.

Others have explored several issues generally related to our work, but none of these works aims at actual capturing changes by aspects. These issues include database schema evolution with aspects [12] or aspect-oriented extensions of business processes and web services with crosscutting concerns of reliability, security, and transactions [3]. Also, an increased changeability of components implemented using aspect-oriented programming [17, 18, 22] and aspect-oriented programming with the frame technology [19], as well as enhanced reusability and evolvability of design patterns achieved by using generic aspect-oriented languages to implement them [24] have been reported. The impact of changes implemented by aspects has been studied using slicing in concern graphs [15].

While we do see potential of aspect-orientation for configuration and reconfiguration of applications, our current work does not aim at automatic adaptation in application evolution, such as event triggered evolutionary actions [21], evolution based on active rules [6], adaptation of languages instead of software systems [16], or as an alternative to version model based context-awareness [7, 13].

8 Conclusions and Further Work

In this paper, we have described our approach to change realization using aspect-oriented programming and proposed a feature modeling based approach of dealing with change interaction. We deal with changes at two levels distinguishing between domain specific and generally applicable change types. We described change types specific to web application domain along with corresponding generally applicable changes. We also discussed consequences of having to implement a change of a change.

The approach does not require exclusiveness in its application: a part of the changes can be realized in a traditional way. In fact, the approach is not appropriate for realization of all changes, and some of them can't be realized by it at all. This is due to a technical limitation given by the capabilities of the underlying aspect-oriented language or framework. Although some work towards addressing method-level constructs such as loops has been reported [14], this is still uncommon practice. What is more important is that relying on the inner details of methods could easily compromise the portability of changes across the versions since the stability of method bodies between versions is questionable.

Change interaction can, of course, be analyzed in code, but it would be very beneficial to deal with it already during modeling. We showed that feature modeling can successfully be applied whereby change realizations would be modeled as variable features of the application concept. Based on such a model, change dependencies could be tracked through feature dependencies. In the absence of a feature model of the application under change, which is often the case, a partial feature model can be developed at far less cost to serve the same purpose.

For further evaluation, it would be interesting to develop catalogs of domain specific change types of other domains like service-oriented architecture for which we have a suitable application developed in Java available [25]. Although the evaluation of the approach has shown the approach can be applied even without a dedicated tool support, we believe that tool support is important in dealing with change interaction, especially if their number is high.

By applying the multi-paradigm design with feature modeling [27] to select the generally applicable changes (understood as paradigms) appropriate to given application specific changes we may avoid the need for catalogs of domain specific change types or we can even use it to develop them. This constitutes the main course of our further research.

Acknowledgements The work was supported by the Scientific Grant Agency of Slovak Republic (VEGA) grant No. VG 1/0508/09.

References

[1] M. Bebjak, V. Vranić, and P. Dolog. Evolution of web applications with aspect-oriented design patterns. In M. Brambilla and E. Mendes, editors, *Proc. of ICWE 2007 Workshops, 2nd International Workshop on Adaptation and Evolution in Web Systems Engineering, AEWSE 2007, in conjunction with 7th International Conference on Web Engineering, ICWE 2007*, pages 80–86, Como, Italy, July 2007.

[2] E. Bodden, F. Forster, and F. Steimann. Avoiding infinite recursion with stratified aspects. In R. Hirschfeld et al., editors, *Proc. of NODe 2006*, LNI P-88, pages 49–64, Erfurt, Germany, Sept. 2006. GI.

[3] A. Charfi, B. Schmeling, A. Heizenreder, and M. Mezini. Reliable, secure, and transacted web service compositions with AO4BPEL. In *4th IEEE European Conf. on Web Services (ECOWS 2006)*, pages 23–34, Zürich, Switzerland, Dec. 2006. IEEE Computer Society.

[4] R. Conradi and B. Westfechtel. Version models for software configuration management. *ACM Computing Surveys*, 30(2):232–282, June 1998.

[5] K. Czarnecki and U. W. Eisenecker. *Generative Programing: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[6] F. Daniel, M. Matera, and G. Pozzi. Combining conceptual modeling and active rules for the design of adaptive web applications. In *Workshop Proc. of 6th Int. Conf. on Web Engineering (ICWE 2006)*, New York, NY, USA, 2006. ACM Press.

[7] F. Dantas, T. Batista, N. Cacho, and A. Garcia. Towards aspect-oriented programming for context-aware systems: A comparative study. In *Proc. of 1st International Workshop on Software Engineering for Pervasive Computing Applications, Systems, and Environments, SEPCASE'07*, Minneapolis, USA, May 2007. IEEE.

[8] P. Dolog, V. Vranić, and M. Bieliková. Representing change by aspect. *ACM SIGPLAN Notices*, 36(12):77–83, Dec. 2001.

[9] Z. Fazekas. Facilitating configurability by separation of concerns in the source code. *Journal of Computing and Information Technology (CIT)*, 13(3):195–210, Sept. 2005.

[10] R. Filkorn and P. Návrat. An approach for integrating analysis patterns and feature diagrams into model driven architecture. In P. Vojtáš, M. Bieliková, and B. Charron-Bost, editors, *Proc. 31st Conference on Current Trends in Theory and Practice of Informatics (SOFSEM 2005)*, LNCS 3381, Liptovský Jan, Slovakia, Jan. 2005. Springer.

[11] S. Goldschmidt, S. Junghagen, and U. Harris. *Strategic Affiliate Marketing*. Edward Elgar Publishing, 2003.

[12] R. Green and A. Rashid. An aspect-oriented framework for schema evolution in object-oriented databases. In *Proc. of the Workshop on Aspects, Components and Patterns for Infrastructure Software (in conjunction with AOSD 2002)*, Enschede, Netherlands, Apr. 2002.

[13] M. Grossniklaus and M. C. Norrie. An object-oriented version model for context-aware data management. In M. Weske, M.-S. Hacid, and C. Godart, editors, *Proc. of 8th International Conference on Web Information Systems Engineering, WISE 2007*, LNCS 4831, Nancy, France, Dec. 2007. Springer.

[14] B. Harbulot and J. R. Gurd. A join point for loops in AspectJ. In *Proc. of 5th International Conference on Aspect-Oriented Software Development, AOSD 2006*, pages 63–74, Bonn, Germany, 2006. ACM.

[15] S. Khan and A. Rashid. Analysing requirements dependencies and change impact using concern slicing. In *Proc. of Aspects, Dependencies, and Interactions Workshop (affiliated to ECOOP 2008)*, Nantes, France, July 2006.

[16] J. Kollár, J. Porubän, P. Václavík, J. Bandáková, and M. Forgáč. Functional approach to the adaptation of languages instead of software systems. *Computer Science and Information Systems Journal (ComSIS)*, 4(2), Dec. 2007.

[17] A. A. Kvale, J. Li, and R. Conradi. A case study on building COTS-based system using aspect-oriented programming. In *2005 ACM Symposium on Applied Computing*, pages 1491–1497, Santa Fe, New Mexico, USA, 2005. ACM.

[18] J. Li, A. A. Kvale, and R. Conradi. A case study on improving changeability of COTS-based system using aspect-oriented programming. *Journal of Information Science and Engineering*, 22(2):375–390, Mar. 2006.

[19] N. Loughran, A. Rashid, W. Zhang, and S. Jarzabek. Supporting product line evolution with framed aspects. In *Workshop on Aspects, Componentsand Patterns for Infrastructure Software (held with AOSD 2004, International Conference on Aspect-Oriented Software Development)*, Lancaster, UK, Mar. 2004.

[20] R. Miles. *AspectJ Cookbook*. O'Reilly, 2004.

[21] F. Molina-Ortiz, N. Medina-Medina, and L. García-Cabrera. An author tool based on SEM-HP for the creation and evolution of adaptive hypermedia systems. In *Workshop Proc. of 6th Int. Conf. on Web Engineering (ICWE 2006)*, New York, NY, USA, 2006. ACM Press.

[22] O. Papapetrou and G. A. Papadopoulos. Aspect-oriented programming for a component based real life application: A case study. In *2004 ACM Symposium on Applied Computing*, pages 1554–1558, Nicosia, Cyprus, 2004. ACM.

[23] E. Pulvermüller, A. Speck, and J. O. Coplien. A version model for aspect dependency management. In *Proc. of 3rd Int. Conf. on Generative and Component-Based Software Engineering (GCSE 2001)*, LNCS 2186, pages 70–79, Erfurt, Germany, Sept. 2001. Springer.

[24] T. Rho and G. Kniesel. Independent evolution of design patterns and application logic with generic aspects—a case study. Technical Report IAI-TR-2006-4, University of Bonn, Bonn, Germany, Apr. 2006.

[25] V. Rozinajová, M. Braun, P. Návrat, and M. Bieliková. Bridging the gap between service-oriented and object-oriented approach in information systems development. In D. Avison, G. M. Kasper, B. Pernici, I. Ramos, and D. Roode, editors, *Proc. of IFIP 20th World Computer Congress, TC 8, Information Systems*, Milano, Italy, Sept. 2008. Springer Boston.

[26] V. Vranić. Reconciling feature modeling: A feature modeling metamodel. In M. Weske and P. Liggsmeyer, editors, *Proc. of 5th Annual International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a Networked World (Net.ObjectDays 2004)*, LNCS 3263, pages 122–137, Erfurt, Germany, Sept. 2004. Springer.

[27] V. Vranić. Multi-paradigm design with feature modeling. *Computer Science and Information Systems Journal (ComSIS)*, 2(1):79–102, June 2005.

[28] V. Vranić, M. Bebjak, R. Menkyna, and P. Dolog. Developing applications with aspect-oriented change realization. In *Proc. of 3rd IFIP TC2 Central and East European Conference on Software Engineering Techniques CEE-SET 2008*, LNCS, Brno, Czech Republic, 2008.

Appendix F

Developing Applications with Aspect-Oriented Change Realization

This appendix contains:

> Valentino Vranić, Michal Bebjak, Radoslav Menkyna, and Peter Dolog. Developing applications with aspect-oriented change realization. In *Proc. of 3rd IFIP TC2 Central and East European Conference on Software Engineering Techniques CEE-SET 2008*, LNCS, Brno, Czech Republic, October 2008. Springer. Postproceedings, to appear.

The paper was accepted to 3rd IFIP TC2 Central and East European Conference on Software Engineering Techniques CEE-SET 2008. My contribution to this paper is approximately 10 %.

Developing Applications with Aspect-Oriented Change Realization

Valentino Vranić[1], Michal Bebjak[1], Radoslav Menkyna[1], and Peter Dolog[2]

[1] Institute of Informatics and Software Engineering
Faculty of Informatics and Information Technologies
Slovak University of Technology,
Ilkovičova 3, 84216 Bratislava 4, Slovakia
vranic@fiit.stuba.sk, mbebjak@gmail.com, radu@ynet.sk

[2] Department of Computer Science
Aalborg University
Selma Lagerlöfs Vej 300, DK-9220 Aalborg EAST, Denmark
dolog@cs.aau.dk

Abstract. An approach to aspect-oriented change realization is proposed in this paper. With aspect-oriented programming changes can be treated explicitly and directly at the programming language level. Aspect-oriented change realizations are mainly based on aspect-oriented design patterns or themselves constitute pattern-like forms in connection to which domain independent change types can be identified. However, it is more convenient to plan changes in a domain specific manner. Domain specific change types can be seen as subtypes of generally applicable change types. This relationship can be maintained in a form of a catalog. Further changes can actually affect the existing aspect-oriented change realizations, which can be solved by adapting the existing change implementation or by implementing an aspect-oriented change realization of the existing change without having to modify its source code. Separating out the changes this way can lead to a kind of aspect-oriented refactoring beneficial to the application as such. As demonstrated partially by the approach evaluation, the problem of change interaction may be avoided to the large extent by using appropriate aspect-oriented development tools, but for a large number of changes, dependencies between them have to be tracked, which could be supported by feature modeling.

Keywords: change, aspect-oriented programming, generally applicable changes, domain specific changes, change interaction

1 Introduction

To quote a phrase, change is the only constant in software development. Change realization consumes enormous effort and time. Once implemented, changes get lost in the code. While individual code modifications are usually tracked by a version control tool, the logic of a change as a whole vanishes without a proper support in the programming language itself.

By its capability to separate crosscutting concerns, aspect-oriented programming enables to deal with change explicitly and directly at programming language level. Changes implemented this way are pluggable and—to the great extent—reapplicable to similar applications, such as applications from the same product line.

Customization of web applications represents a prominent example of that kind. In customization, a general application is being adapted to the client's needs by a series of changes. With each new version of the base application all the changes have to be applied to it. In many occasions, the difference between the new and old application does not affect the structure of changes, so if changes have been implemented using aspect-oriented programming, they can be simply included into the new application build without any additional effort.

We have already reported briefly our initial efforts in change realization using aspect-oriented programming [1]. In this paper, we present our improved view of the approach to change realization and the change types we discovered. Section 2 presents our approach to aspect-oriented change realization. Section 3 introduces the change types we have discovered so far in the web application domain. Section 4 discusses how to deal with a change of a change. Section 5 describes the approach evaluation and identifies the possibilities of coping with change interaction with tool support. Section 6 discusses related work. Section 7 presents conclusions and directions of further work.

2 Changes as Crosscutting Requirements

A change is initiated by a change request made by a user or some other stakeholder. Change requests are specified in domain notions similarly as initial requirements are. A change request tends to be focused, but it often consists of several different—though usually interrelated—requirements that specify actual changes to be realized. By decomposing a change request into individual changes and by abstracting the essence out of each such change while generalizing it at the same time, a change type applicable to a range of the applications that belong to the same domain can be defined.

We will introduce our approach by a series of examples on a common scenario.[3] Suppose a merchant who runs his online music shop purchases a general affiliate marketing software [9] to advertise at third party web sites denoted as affiliates. In a simplified schema of affiliate marketing, a customer visits an affiliate's site which refers him to the merchant's site. When he buys something from the merchant, the provision is given to the affiliate who referred the sale. A general affiliate marketing software enables to manage affiliates, track sales referred by these affiliates, and compute provisions for referred sales. It is also able to send notifications about new sales, signed up affiliates, etc.

The general affiliate marketing software has to be adapted (customized), which involves a series of changes. We will assume the affiliate marketing software

[3] This is an adapted scenario published in our earlier work [1].

is written in Java and use AspectJ, the most popular aspect-oriented language, which is based on Java, to implement some of these changes.

In the AspectJ style of aspect-oriented programming, the crosscutting concerns are captured in units called aspects. Aspects may contain fields and methods much the same way the usual Java classes do, but what makes possible for them to affect other code are genuine aspect-oriented constructs, namely: *pointcuts*, which specify the places in the code to be affected, *advices*, which implement the additional behavior before, after, or instead of the captured *join point* (a well-defined place in the program execution)—most often method calls or executions—and *inter-type declarations*, which enable introduction of new members into types, as well as introduction of compilation warnings and errors.

2.1 Domain Specific Changes

One of the changes of the affiliate marketing software would be adding a backup SMTP server to ensure delivery of the notifications to users. Each time the affiliate marketing software needs to send a notification, it creates an instance of the SMTPServer class which handles the connection to the SMTP server.

An SMTP server is a kind of a resource that needs to be backed up, so in general, the type of the change we are talking about could be denoted as *Introducing Resource Backup*. This change type is still expressed in a domain specific way. We can clearly identify a crosscutting concern of maintaining a backup resource that has to be activated if the original one fails and implement this change in a single aspect without modifying the original code:

```
class AnotherSMTPServer extends SMTPServer {
  . . .
}
public aspect BackupSMTPServer {
  public pointcut SMTPServerConstructor(URL url, String user, String password):
    call(SMTPServer.new(..)) && args (url, user, password);
  SMTPServer around(URL url, String user, String password):
    SMTPServerConstructor(url, user, password) {
    return getSMTPServerBackup(proceed(url, user, password));
  }
  SMTPServer getSMTPServerBackup(SMTPServer obj) {
    if (obj.isConnected()) {
      return obj;
    }
    else {
      return new AnotherSMTPServer(obj.getUrl(), obj.getUser(),
        obj.getPassword());
    }
  }
}
```

The **around**() advice captures constructor calls of the SMTPServer class and their arguments. This kind of advice takes complete control over the captured join point and its return clause, which is used in this example to control the type of the SMTP server being returned. The policy is implemented in the getSMTPServerBackup() method: if the original SMTP server can't be connected to, a backup SMTP server class instance is created and returned.

2.2 Generally Applicable Changes

Looking at this code and leaving aside SMTP servers and resources altogether, we notice that it actually performs a class exchange. This idea can be generalized and domain details abstracted out of it bringing us to the *Class Exchange* change type [1] which is based on the *Cuckoo's Egg* aspect-oriented design pattern [16]:

```
public class AnotherClass extends MyClass {
   . . .
}
public aspect MyClassSwapper {
   public pointcut myConstructors(): call(MyClass.new());
   Object around(): myConstructors() {
      return new AnotherClass();
   }
}
```

2.3 Applying a Change Type

It would be beneficial if the developer could get a hint on using the Cuckoo's Egg pattern based on the information that a resource backup had to be introduced. This could be achieved by maintaining a catalog of changes in which each domain specific change type would be defined as a specialization of one or more generally applicable changes.

When determining a change type to be applied, a developer chooses a particular change request, identifies individual changes in it, and determines their type. Figure 1 shows an example situation. Domain specific changes of the D1 and D2 type have been identified in the Change Request 1. From the previously identified and cataloged relationships between change types, we would know their generally applicable change types are G1 and G2.

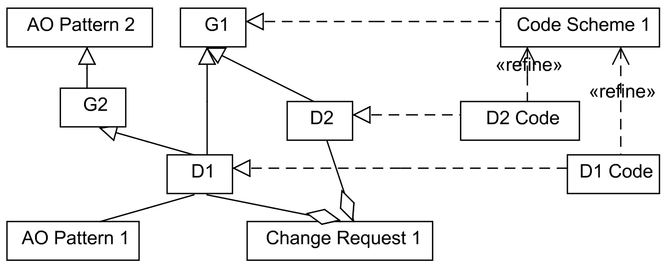


Fig. 1. Generally applicable and domain specific changes.

A generally applicable change type can be a kind of an aspect-oriented design pattern (consider G2 and AO Pattern 2). A domain specific change realization can also be complemented by an aspect-oriented design patterns, which is expressed by an association between them (consider D1 and AO Pattern 1).

Each generally applicable change has a known domain independent code scheme (G2's code scheme is omitted from the figure). This code scheme has to be adapted to the context of a particular domain specific change, which may be seen as a kind of refinement (consider D1 Code and D2 Code).

3 Catalog of Changes

To support the process of change selection, the catalog of changes is needed in which the generalization–specialization relationships between change types would be explicitly established. The following list sums up these relationships between change types we have identified in the web application domain (the domain specific change type is introduced first):

- One Way Integration: Performing Action After Event
- Two Way Integration: Performing Action After Event
- Adding Column to Grid: Performing Action After Event
- Removing Column from Grid: Method Substitution
- Altering Column Presentation in Grid: Method Substitution
- Adding Fields to Form: Enumeration Modification with Additional Return Value Checking/Modification
- Removing Fields from Form: Additional Return Value Checking/Modification
- Introducing Additional Constraint on Fields: Additional Parameter Checking or Performing Action After Event
- Introducing User Rights Management: Border Control with Method Substitution
- User Interface Restriction: Additional Return Value Checking/Modifications
- Introducing Resource Backup: Class Exchange

We have already described Introducing Resource Backup and the corresponding generally applicable change, Class Exchange. Here, we will briefly describe the rest of the domain specific change types we identified in the web application domain along with the corresponding generally applicable changes. The generally applicable change types are described where they are first mentioned to make the sequential reading of this section easier. A real catalog of changes would require to describe each change type separately.

3.1 Integration Changes

Web applications often have to be integrated with other systems. Suppose that in our example the merchant wants to integrate the affiliate marketing software with the third party newsletter which he uses. Every affiliate should be a member of the newsletter. When an affiliate signs up to the affiliate marketing software, he should be signed up to the newsletter, too. Upon deleting his account, the affiliate should be removed from the newsletter, too.

This is a typical example of the *One Way Integration* change type [1]. Its essence is the one way notification: the integrating application notifies the integrated application of relevant events. In our case, such events are the affiliate sign-up and affiliate account deletion.

Such integration corresponds to the *Performing Action After Event* change type [1]. Since events are actually represented by methods, the desired action can be implemented in an after advice:

```
public aspect PerformActionAfterEvent {
   pointcut methodCalls(TargetClass t, int a): . . .;
   after(/* captured arguments */): methodCalls(/* captured arguments */) {
      performAction(/* captured arguments */);
   }
   private void performAction(/* arguments */) { /* action logic */ }
}
```

The after advice executes after the captured method calls. The actual action is implemented as the performAction() method called by the advice.

To implement the one way integration, in the after advice we will make a post to the newsletter sign-up/sign-out script and pass it the e-mail address and name of the newly signed-up or deleted affiliate. We can seamlessly combine multiple one way integrations to integrate with several systems.

The *Two Way Integration* change type can be seen as a double One Way Integration. A typical example of such a change is data synchronization (e.g., synchronization of user accounts) across multiple systems. When a user changes his profile in one of the systems, these changes should be visible in all of them. In our example, introducing a forum for affiliates with synchronized user accounts for affiliate convenience would represent a *Two Way Integration.*

3.2 Introducing User Rights Management

In our affiliate marketing application, the marketing is managed by several co-workers with different roles. Therefore, its database has to be updated from an administrator account with limited permissions. A limited administrator should not be able to decline or delete affiliates, nor modify the advertising campaigns and banners that have been integrated with the web sites of affiliates. This is an instance of the *Introducing User Rights Management* change type.

Suppose all the methods for managing campaigns and banners are located in the campaigns and banners packages. The calls to these methods can be viewed as a region prohibited to the restricted administrator. The *Border Control* design pattern [16] enables to partition an application into a series of regions implemented as pointcuts that can later be operated on by advices [1]:

```
pointcut prohibitedRegion(): (within(application.Proxy) && call(void *.*(..)))
   || (within(application.campaigns.+) && call(void *.*(..)))
```

```
    || within(application.banners.+)
    || call(void Affiliate.decline(..)) || call(void Affiliate.delete(..));
}
```

What we actually need is to substitute the calls to the methods in the region with our own code that will let the original methods execute only if the current user has sufficient rights. This can be achieved by applying the *Method Substitution* change type which is based on an around advice that enables to change or completely disable the execution of methods. The following pointcut captures all method calls of the method called method() belonging to the TargetClass class:

```
pointcut allmethodCalls(TargetClass t, int a):
    call(ReturnType TargetClass.method(..)) && target(t) && args(a);
```

Note that we capture method calls, not executions, which gives us the flexibility in constraining the method substitution logic by the context of the method call.

The pointcut **call**(ReturnType TargetClass.method(..)) captures all the calls of TargetClass.method(). The **target**() pointcut is used to capture the reference to the target class. The method arguments can be captured by an **args**() pointcut. In the example code above, we assume method() has one integer argument and capture it with this pointcut.

The following example captures the method() calls made within the control flow of any of the CallingClass methods:

```
pointcut specificmethodCalls(TargetClass t, int a):
    call(ReturnType TargetClass.method(a)) && target(t) && args(a)
    && cflow(call(∗ CallingClass.∗(..)));
```

This embraces the calls made directly in these methods, but also any of the method() calls made further in the methods called directly or indirectly by the CallingClass methods.

By making an around advice on the specified method call capturing pointcut, we can create a new logic of the method to be substituted:

```
public aspect MethodSubstition {
    pointcut methodCalls(TargetClass t, int a): . . .;
    ReturnType around(TargetClass t, int a): methodCalls(t, a) {
        if (. . .) {
            . . . } // the new method logic
        else
            proceed(t, a);
    }
}
```

3.3 User Interface Restriction

It is quite annoying when a user sees, but can't access some options due to user rights restrictions. This requires a *User Interface Restriction* change type to be applied. We have created a similar situation in our example by a previous change implementation that introduced the restricted administrator (see

Sect. 3.2). Since the restricted administrator can't access advertising campaigns and banners, he shouldn't see them in menu either.

Menu items are retrieved by a method and all we have to do to remove the banners and campaigns items is to modify the return value of this method. This may be achieved by applying a *Additional Return Value Checking/Modification* change which checks or modifies a method return value using an around advice:

```
public aspect AdditionalReturnValueProcessing {
   pointcut methodCalls(TargetClass t, int a): . . .;
   private ReturnType retValue;
   ReturnType around(): methodCalls(/* captured arguments */) {
        retValue = proceed(/* captured arguments */);
        processOutput(/* captured arguments */);
        return retValue;
   }
   private void processOutput(/* arguments */) {
       // processing logic
   }
}
```

In the around advice, we assign the original return value to the private attribute of the aspect. Afterwards, this value is processed by the processOutput() method and the result is returned by the around advice.

3.4 Grid Display Changes

It is often necessary to modify the way data are displayed or inserted. In web applications, data are often displayed in grids, and data input is usually realized via forms. Grids usually display the content of a database table or collation of data from multiple tables directly. Typical changes required on grid are adding columns, removing them, and modifying their presentation. A grid that is going to be modified must be implemented either as some kind of a reusable component or generated by row and cell processing methods. If the grid is hard coded for a specific view, it is difficult or even impossible to modify it using aspect-oriented techniques.

If the grid is implemented as a data driven component, we just have to modify the data passed to the grid. This corresponds to the Additional Return Value Checking/Modification change (see Sect. 3.3). If the grid is not a data driven component, it has to be provided at least with the methods for processing rows and cells.

Adding Column to Grid can be performed *after an event* of displaying the existing columns of the grid which brings us to the Performing Action After Event change type (see Sect. 3.1). Note that the database has to reflect the change, too. *Removing Column from Grid* requires a conditional execution of the method that displays cells, which may be realized as a Method Substitution change (see Sect. 3.2).

Alterations of a grid are often necessary due to software localization. For example, in Japan and Hungary, in contrast to most other countries, the surname is placed before the given names. The *Altering Column Presentation in Grid* change type requires preprocessing of all the data to be displayed in a grid before actually displaying them. This may be easily achieved by modifying the way the grid cells are rendered, which may be implemented again as a Method Substitution (see Sect. 3.2):

```
public aspect ChangeUserNameDisplay {
    pointcut displayCellCalls(String name, String value):
        call(void UserTable.displayCell(..)) || args(name, value);
    around(String name, String value): displayCellCalls(name, value) {
        if (name == "<the name of the column to be modified>") {
            . . . // display the modified column
        } else {
            proceed(name, value);
        }
    }
}
```

3.5 Input Form Changes

Similarly to tables, forms are often subject to modifications. Users often want to add or remove fields from forms or perform additional checks of the form inputs, which constitute *Adding Fields to Form*, *Removing Fields from Form*, and *Introducing Additional Constraint on Fields* change types, respectively. Note that to be possible to modify forms using aspect-oriented programming they may not be hard coded in HTML, but generated by a method. Typically, they are generated from a list of fields implemented by an enumeration.

Going back to our example, assume that the merchant wants to know the genre of the music which is promoted by his affiliates. We need to add the genre field to the generic affiliate sign-up form and his profile form to acquire the information about the genre to be promoted at different affiliate web sites. This is a change of the *Adding Fields to Form* type. To display the required information, we need to modify the affiliate table of the merchant panel to display genre in a new column. This can be realized by applying the Enumeration Modification change type to add the genre field along with already mentioned Additional Return Value Checking/Modification in order to modify the list of fields being returned (see Sect. 3.3).

The realization of the *Enumeration Modification* change type depends on the enumeration type implementation. Enumeration types are often represented as classes with a static field for each enumeration value. A single enumeration value type is represented as a class with a field that holds the actual (usually integer) value and its name. We add a new enumeration value by introducing the corresponding static field:

```
public aspect NewEnumType {
    public static EnumValueType EnumType.NEWVALUE =
        new EnumValueType(10, "<new value name>");
}
```

The fields in a form are generated according to the enumeration values. The list of enumeration values is typically accessible via a method provided by it. This method has to be addressed by an Additional Return Value Checking/-Modification change.

An Additional Return Value Checking/Modification change is sufficient to remove a field from a form. Actually, the enumeration value would still be included in the enumeration, but this would not affect the form generation.

If we want to introduce additional validations on the form input data to the system without built-in validation, an *Additional Parameter Checking* change can be applied to methods that process values submitted by the form. This change enables to introduce an additional check or constraint on method arguments. For this, we have to specify a pointcut that will capture all the calls of the affected methods along with their context similarly as in Sect. 3.2. Their arguments will be checked by the check() method called from within an around advice which will throw WrongParamsException if they are not correct:

```
public aspect AdditionalParameterChecking {
   pointcut methodCalls(TargetClass t, int a): . . .;
   ReturnType around(/* arguments */) throws WrongParamsException:
      methodCalls(/* arguments */) {
      check(/* arguments */);
      return proceed(/* arguments */);
   }
   void check(/* arguments */) throws WrongParamsException {
      if (arg1 != <desired value>)
         throw new WrongParamsException();
   }
}
```

Adding a new validator to a system that already has built-in validation is realized by simply adding it to the list of validators. This can be done by implementing Performing Action After Event change (see Sect. 3.1), which would implement the addition of the validator to the list of validators after the list initialization.

4 Changing a Change

Sooner or later there will be a need for a change whose realization will affect some of the already applied changes. There are two possibilities to deal with this situation: a new change can be implemented separately using aspect-oriented programming or the affected change source code could be modified directly. Either way, the changes remain separate from the rest of the application.

The possibility to implement a change of a change using aspect-oriented programming and without modifying the original change is given by the aspect-oriented programming language capabilities. Consider, for example, advices in AspectJ. They are unnamed, so can't be referred to directly. The primitive pointcut **adviceexecution**(), which captures execution of all advices, can be restricted by the **within**() pointcut to a given aspect, but if an aspect contains several advices, advices have to be annotated and accessed by the **@annotation**()

pointcut, which was impossible in AspectJ versions that existed before Java was extended with annotations.

An interesting consequence of aspect-oriented change realization is the separation of crosscutting concerns in the application which improves its modularity (and thus makes easier further changes) and may be seen as a kind of aspect-oriented refactoring. For example, in our affiliate marketing application, the integration with a newsletter—identified as a kind of One Way Integration—actually was a separation of integration connection, which may be seen as a concern of its own. Even if these once separated concerns are further maintained by direct source code modification, the important thing is that they remain separate from the rest of the application. Implementing a change of a change using aspect-oriented programming and without modifying the original change is interesting mainly if it leads to separation of another crosscutting concern.

5 Evaluation and Tool Support Outlooks

We have successfully applied the aspect-oriented approach to change realization to introduce changes into YonBan, a student project management system developed at Slovak University of Technology. It is based on J2EE, Spring, Hibernate, and Acegi frameworks. The YonBan architecture is based on the Inversion Of Control principle and Model-View-Controller pattern. We implemented the following changes in YonBan:

- Telephone number validator as Performing Action After Event
- Telephone number formatter as Additional Return Value Checking/Modification
- Project registration statistics as One Way Integration
- Project registration constraint as Additional Parameter Checking/Modification
- Exception logging as Performing Action After Event
- Name formatter as Method Substitution

No original code of the system had to be modified. Except in the case of project registration statistics and project registration constraint, which where well separated from the rest of the code, other changes would require extensive code modifications if they have had been implemented the conventional way.

We encountered one change interaction: between the telephone number formatter and validator. These two changes are interrelated—they would probably be part of one change request—so it comes as no surprise they affect the same method. However, no intervention was needed.

We managed to implement the changes easily even without a dedicated tool, but to cope with a large number of changes, such a tool may become crucial. Even general aspect-oriented programming support tools—usually integrated with development environments—may be of some help in this. AJDT (AspectJ Development Tools) for Eclipse is a prominent example of such a tool. AJDT shows whether a particular code is affected by advices, the list of join points affected by each advice, and the order of advice execution, which all are important to track when multiple changes affect the same code. Advices that do not affect any join point are reported in compilation warnings, which may help detect pointcuts invalidated by direct modifications of the application base code such as identifier name changes or changes in method arguments.

A dedicated tool could provide a much more sophisticated support. A change implementation can consist of several aspects, classes, and interfaces, commonly denoted as types. The tool should keep a track of all the parts of a change. Some types may be shared among changes, so the tool should enable simple inclusion and exclusion of changes. This is related to *change interaction* which is exhibited as dependencies between changes. A simplified view of change dependencies is that a change may require another change or two changes may be mutually exclusive, but the dependencies between changes could be as complex as feature dependencies in feature modeling and accordingly represented by feature diagrams and additional constraints expressed as logical expressions [22] (which can be partly embedded into feature diagrams by allowing them to be directed acyclic graphs instead of just trees [8]).

Some dependencies between changes may exhibit only recommending character, i.e. whether they are expected to be included or not included together, but their application remains meaningful either way. An example of this are features that belong to the same change request. Again, feature modeling can be used to model such dependencies with so-called default dependency rules that may also be represented by logical expressions [22].

6 Related Work

The work presented in this paper is based on our initial efforts related to aspect-oriented change control [6] in which we related our approach to change-based approaches in version control. We identified that the problem with change-based approaches that could be solved by aspect-oriented programming is the lack of programming language awareness in change realizations.

In our work on the evolution of web applications based on aspect-oriented design patterns and pattern-like forms [1], we reported the fundamentals of aspect-oriented change realizations based on the two level model of domain specific and generally applicable change types, as well as four particular change types: Class Exchange, Performing Action After Event, and One/Two Way Integration.

Applying feature modeling to maintain change dependencies (see Sect. 4) is similar to constraints and preferences proposed in SIO software configuration management system [4]. However, a version model for aspect dependency management [19] with appropriate aspect model that enables to control aspect recursion and stratification [2] would be needed as well.

We tend to regard changes as concerns, which is similar to the approach of facilitating configurability by separation of concerns in the source code [7]. This approach actually enables a kind of aspect-oriented programming on top of a versioning system. Parts of the code that belong to one concern need to be marked manually in the code. This enables to easily plug in or out concerns. However, the major drawback, besides having to manually mark the parts of concerns, is that—unlike in aspect-oriented programming—concerns remain tangled in code.

Others have explored several issues generally related to our work, but none of this work aims at capturing changes by aspects. These issuse include database schema evolution with aspects [10] or aspect-oriented extensions of business processes and web services with crosscutting concerns of reliability, security, and transactions [3]. Also, an increased changeability of components implemented using aspect-oriented programming [13, 14, 18] and aspect-oriented programming with the frame technology [15], as well as enhanced reusability and evolvability of design patterns achieved by using generic aspect-oriented languages to implement them [20] have been reported. The impact of changes implemented by aspects has been studied using slicing in concern graphs [11].

While we do see potential of configuration and reconfiguration of applications, our work does not aim at automatic adaptation in application evolution, such as event triggered evolutionary actions [17], evolution based on active rules [5], or adaptation of languages instead of software systems [12].

7 Conclusions and Further Work

In this paper, we have described our approach to change realization using aspect-oriented programming. We deal with changes at two levels distinguishing between domain specific and generally applicable change types. We introduced change types specific to web application domain along with corresponding generally applicable changes. We also discussed consequences of having to implement a change of a change.

Although the evaluation of the approach has shown the approach can be applied even without a dedicated tool support, we believe that tool support is important in dealing with change interaction, especially if their number is high. Our intent is to use feature modeling. With changes modeled as features, change dependencies could be tracked through feature dependencies. For further evaluation, it would be interesting to expand domain specific change types to other domains like service-oriented architecture for which we have available suitable application developed in Java [21].

Acknowledgements The work was supported by the Scientific Grant Agency of Slovak Republic (VEGA) grant No. VG 1/3102/06. We would like to thank Michael Grossniklaus for sharing his observations regarding our work with us.

References

[1] M. Bebjak, V. Vranić, and P. Dolog. Evolution of web applications with aspect-oriented design patterns. In M. Brambilla and E. Mendes, editors, *Proc. of*

ICWE 2007 Workshops, 2nd International Workshop on Adaptation and Evolution in Web Systems Engineering, AEWSE 2007, in conjunction with 7th International Conference on Web Engineering, ICWE 2007, pages 80–86, Como, Italy, July 2007.

[2] E. Bodden, F. Forster, and F. Steimann. Avoiding infinite recursion with stratified aspects. In R. Hirschfeld et al., editors, *Proc. of NODe 2006*, LNI P-88, pages 49–64, Erfurt, Germany, Sept. 2006. GI.

[3] A. Charfi, B. Schmeling, A. Heizenreder, and M. Mezini. Reliable, secure, and transacted web service compositions with AO4BPEL. In *4th IEEE European Conf. on Web Services (ECOWS 2006)*, pages 23–34, Zürich, Switzerland, Dec. 2006. IEEE Computer Society.

[4] R. Conradi and B. Westfechtel. Version models for software configuration management. *ACM Computing Surveys*, 30(2):232–282, June 1998.

[5] F. Daniel, M. Matera, and G. Pozzi. Combining conceptual modeling and active rules for the design of adaptive web applications. In *Workshop Proc. of 6th Int. Conf. on Web Engineering (ICWE 2006)*, New York, NY, USA, 2006. ACM Press.

[6] P. Dolog, V. Vranić, and M. Bieliková. Representing change by aspect. *ACM SIGPLAN Notices*, 36(12):77–83, Dec. 2001.

[7] Z. Fazekas. Facilitating configurability by separation of concerns in the source code. *Journal of Computing and Information Technology (CIT)*, 13(3):195–210, Sept. 2005.

[8] R. Filkorn and P. Návrat. An approach for integrating analysis patterns and feature diagrams into model driven architecture. In P. Vojtáš, M. Bieliková, and B. Charron-Bost, editors, *Proc. 31st Conference on Current Trends in Theory and Practice of Informatics (SOFSEM 2005)*, LNCS 3381, Liptovský Jan, Slovakia, Jan. 2005. Springer.

[9] S. Goldschmidt, S. Junghagen, and U. Harris. *Strategic Affiliate Marketing.* Edward Elgar Publishing, 2003.

[10] R. Green and A. Rashid. An aspect-oriented framework for schema evolution in object-oriented databases. In *Proc. of the Workshop on Aspects, Components and Patterns for Infrastructure Software (in conjunction with AOSD 2002)*, Enschede, Netherlands, Apr. 2002.

[11] S. Khan and A. Rashid. Analysing requirements dependencies and change impact using concern slicing. In *Proc. of Aspects, Dependencies, and Interactions Workshop (affiliated to ECOOP 2008)*, Nantes, France, July 2006.

[12] J. Kollár, J. Porubän, P. Václavík, J. Bandáková, and M. Forgáč. Functional approach to the adaptation of languages instead of software systems. *Computer Science and Information Systems Journal (ComSIS)*, 4(2), Dec. 2007.

[13] A. A. Kvale, J. Li, and R. Conradi. A case study on building COTS-based system using aspect-oriented programming. In *2005 ACM Symposium on Applied Computing*, pages 1491–1497, Santa Fe, New Mexico, USA, 2005. ACM.

[14] J. Li, A. A. Kvale, and R. Conradi. A case study on improving changeability of COTS-based system using aspect-oriented programming. *Journal of Information Science and Engineering*, 22(2):375–390, Mar. 2006.

[15] N. Loughran, A. Rashid, W. Zhang, and S. Jarzabek. Supporting product line evolution with framed aspects. In *Workshop on Aspects, Componentsand Patterns for Infrastructure Software (held with AOSD 2004, International Conference on Aspect-Oriented Software Development)*, Lancaster, UK, Mar. 2004.

[16] R. Miles. *AspectJ Cookbook.* O'Reilly, 2004.

[17] F. Molina-Ortiz, N. Medina-Medina, and L. García-Cabrera. An author tool based on SEM-HP for the creation and evolution of adaptive hypermedia systems. In *Workshop Proc. of 6th Int. Conf. on Web Engineering (ICWE 2006)*, New York, NY, USA, 2006. ACM Press.

[18] O. Papapetrou and G. A. Papadopoulos. Aspect-oriented programming for a component based real life application: A case study. In *2004 ACM Symposium on Applied Computing*, pages 1554–1558, Nicosia, Cyprus, 2004. ACM.

[19] E. Pulvermüller, A. Speck, and J. O. Coplien. A version model for aspect dependency management. In *Proc. of 3rd Int. Conf. on Generative and Component-Based Software Engineering (GCSE 2001)*, LNCS 2186, pages 70–79, Erfurt, Germany, Sept. 2001. Springer.

[20] T. Rho and G. Kniesel. Independent evolution of design patterns and application logic with generic aspects—a case study. Technical Report IAI-TR-2006-4, University of Bonn, Bonn, Germany, Apr. 2006.

[21] V. Rozinajová, M. Braun, P. Návrat, and M. Bieliková. Bridging the gap between service-oriented and object-oriented approach in information systems development. In D. Avison, G. M. Kasper, B. Pernici, I. Ramos, and D. Roode, editors, *Proc. of IFIP 20th World Computer Congress, TC 8, Information Systems*, Milano, Italy, Sept. 2008. Springer Boston.

[22] V. Vranić. Multi-paradigm design with feature modeling. *Computer Science and Information Systems Journal (ComSIS)*, 2(1):79–102, June 2005.

Appendix G

Dealing with Interaction of Aspect-Oriented Change Realizations using Feature Modeling

Radoslav Menkyna. Dealing with interaction of aspect-oriented change realizations using feature modeling. In Mária Bieliková, editor, *IIT.SRC: Student Research Conference 2009*, pages 54–61. Slovak University of Technology in Bratislava, 2009.

The paper was accepted to IIT.SRC: Student Research Conference 2009. It was actively presented on the conference and was proposed for award from Slovak literal fund.

Dealing with Interaction of Aspect-Oriented Change Realizations using Feature Modeling

Radoslav MENKYNA*

Slovak University of Technology
Faculty of Informatics and Information Technologies
Ilkovičova 3, 842 16 Bratislava, Slovakia
`xmenkyna@is.stuba.sk`

Abstract. The realization of changes by aspects in the existing projects can lead to their interaction. A change propagation in the system can be examined using dependency graphs. This paper proposes an approach which represents the changes and their dependencies using feature modeling. This modeling technique is more suitable for the given approach because changes represented as aspects can be seen as the new features of the existing system. Two approaches how to transform dependency graph to the feature model were proposed. Intended use of this approach was outlined.

1 Introduction

Feature modeling seams suitable for modeling changes represented as aspects. In order to examine the interaction of changes with the existing system or among each other it is needed to capture scope where interaction occurs. Such scope can be expressed by the concerns and concern slices. Dependency graphs can be used to capture change propagation and dependencies among concerns present in the system [6]. If it is possible to transform dependency graphs into feature models, one modeling technique would capture aspect-oriented change realizations along with existing concerns and dependencies between them. This is crucial for studying interaction of changes.

The rest of the paper is organized as follows. Section 2 describes an approach in which the changes are represented as aspects. Section 3 discusses a possible interaction

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

of changes and its connection with the concerns and program slices. Section 4 describes dependency graphs and feature modeling. Section 5 proposes transformation of dependency graphs to the feature models. Section 7 represents a conclusion and outlines the future work.

2 Aspect-Oriented Change Realization

Changes of the existing software project can be realized by aspects [2]. To support this approach several change realization techniques were presented [1].

Main goal of aspect-oriented paradigm is a separation of crosscutting concerns. The AspectJ language can be considered as main approach to this paradigm, because its large community acceptance. To achieve the separation of concerns new language constructs were created . Pointcut expresses set of points in control flow of an application. Upon pointcuts actions defined in advices can be performed. Pointcuts and advices are defined in class-like entity called aspect.

Like crosscutting concern also change usually affects several points in existing code, therefore use of aspects to represent changes can have several benefits. All the modifications coupled with a particular change are centralized in an aspect. The entire change logic is also represented in the aspect. Because target is usually not aware of the change, the change can be easily plugged or unplugged from the system. This means aspect-oriented change realizations are modular and pluggable.

Several change realization techniques were described [1] to support the use of aspects to represent a change. A change realization technique usually uses an aspect-oriented idiom or design pattern to represent a change. These techniques are described generally which means they can be used to implement changes in several domains.

3 Interaction of Changes

Scope of system in which interaction occurs can be captured by concerns or program slices. This section will explain these terms and point out some problems coupled with the interaction of changes represented by aspects.

With the growing number of changes grows also the possibility of interaction between the changes present in the system. The interaction can have negative effects and can lead to an unexpected behavior of the system. There are various reasons why different changes represented as aspects interact. Aspects that use same pointcuts can execute in wrong order. Pointcuts no longer capture desired join points because of system evolution. Also subsequent interrelated application of the aspect-oriented design patterns to a particular problem can require additional changes to design patterns already present in the system [5]. Thus, by combining change techniques additional changes can be required, which can be seen as an unwanted interaction.

In addition to study interactions of changes implemented by aspects it is needed to capture a scope of the system in which the interactions are most likely to occur. When applying a change in traditional fashion the proposed change can interfere with many entities from a existing source code. A change realized as an aspect modularizes the essence of proposed change but still affects the system in one or several points specified by the aspect pointcuts. In both cases it is needed to identify a part of the system where proposed change can affect the existing entities. An approach of program slicing [3, 9] can be used to achieve this goal.

A program slice narrows a behavior of the program to specified subset of interest. Concern, on the other hand, represents a part of the system behavior from larger scale and higher complexity. Usually several slices that represent an elementary behavior can be grouped together to represent a concern. Analyzing these concerns and slices can lead to better understanding of dependencies among changes and their impact on system [6]. In the next section a technique that visualizes known dependencies among concerns and their slices will be described.

4 Dependency Graphs and Feature modeling

Dependency graphs help to visualize the change propagation and dependencies between concerns and their slices [6]. The dependency graphs are constructed from semi-formal dependency equations. Each concern can consist of temporal(S_T), conditional (S_C), business rule(S_B) and task oriented slices(S_{TO}). In the dependecy graph concern slices are represented by nodes and dependencies by edges. There are three types of dependencies which can be captured between the concern slices by the dependency equations and graphs . A forward dependency means a concern slice links or results to another concern slice. A backward dependency means the concern slice uses or bases itself on previous slice. Parallel dependency expresses the concern slices occur concurrently [6].

Figure 1 shows a simple dependency graph that captures dependences in case study of an tollgate system [6]. In this system owner of the vehicle first registers with the bank and activates a gizmo trough ATM. The toll is then automatically deducted when the payed motorway is used. The forward dependency is represented by arrow, the backward dependency by dashed arrow and the parallel dependency by two way arrow. Numbers in the graph express assigned weights of dependencies during change propagation evaluation. Considered concern slice is depicted as dash-dotted.

Figure shows that considered register business rule concern slice $Register_B$ is forward dependent on read and store gizmo information conditional concern slice $ReadStoreInfo_C$, which is forward dependent on debit concern's business rule and conditional slice $Debit_{B \wedge C}$. Correctness and compatibility concern slices $Correctness_C$ and $Compatibility_B$ are parallel dependent on debit concern slices. Calculate concern slices $Calculate_{B \wedge C}$ and read store gizmo information concern slice $ReadStoreInfo_C$ are backward dependent on debit concern slices [6].

One can notice that the read store gizmo information concern slice occurs twice in dependency graph. This suggests that dependency graphs capture also behavior. This issue will be addressed in the next section.

Changes implemented as aspects are modular and pluggable. They can be considered

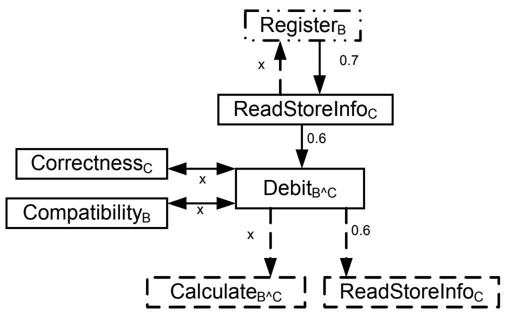


Figure 1. Dependency graph for register *concern's business rule slice. Adapted from [6]*

as the features of the system which can be included or excluded from the final configuration of the system. Therefore, feature modeling approach is suitable for modeling changes implemented as aspects.

Feature modeling can be used to capture commonality and variability in the software product lines [4]. A interaction between the features of products in software product lines can occur. Some features may require presence of other features or features are considered alternative. Aspect-oriented change representations can also be suitable for implementing variable features of software product line. It is important that these approaches share common modeling technique.

A feature model consists of feature diagrams, constraints, default dependency rules and information associated with concepts and features. Feature diagram is a directed tree whose root represents concept and all other nodes represent concept features [7]. Feature diagrams can visually capture properties of a feature or dependencies between several features. For example features may be depicted as mandatory or optional, two features may be depicted as alternative or or-features. Additional dependencies can be captured as constraints of feature diagrams trough the predicate logic.

5 Transforming Dependency Graphs into Feature Models

Feature modeling is suitable for modeling changes implemented as aspects. Dependency graphs can be used to express change propagation, and dependencies among concerns and their slices. This section will describe two different approaches of transformation of dependency graphs to feature models. The first one captures the dependencies primary by the feature model hierarchy (Section 5.1). The second one using the feature model constraints, which is discussed in the next section. Feature diagrams are structural while dependency graphs seam to capture behavior, too. Capturing of the behavioral component of dependency graphs will be discussed (Section 5.3).

5.1 Dependencies Captured by Feature Diagrams

In this approach, concern slices are considered as features of the system. Dependencies are modeled as relationships between features.

There are three types of dependencies between concern slices in concern dependency graphs: forward, backward, and parallel. Forward dependency represents what might follow from one concern slice. It can be understood as optionally, thus the concern slice is forward dependent on some other concern slice, this concern slice should be modeled as an optional feature of the former concern slice.

Backward dependency is expressed by the tree topology: a backward dependent concern slice is a subfeature of the slice it depends on. At the same time, there is a forward dependency in the opposite direction, which is in compliance with available concern dependency graphs [6].

In terms of feature modeling, parallel dependency simply poses a constraint that two features that represent parallel dependent concern slices must appear together in all possible system configurations. One way to achieve this is to model either of them as a mandatory feature of the other one.

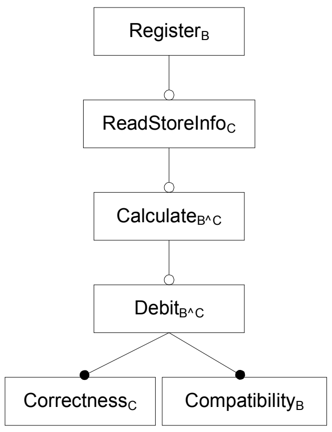


Figure 2. Dependencies captured by feature diagrams.

Figure 2 shows an example of transformed dependency graph from Section 4. All forward dependencies were modeled as optional features of former concern slices. Two parallel dependencies $\mathit{Correctness}_C$ and $\mathit{Compatibility}_B$ were modeled as mandatory features of debit concern slice $\mathit{Debit}_{B \wedge C}$. The backward dependencies of $\mathit{Calculate}_{B \wedge C}$ and $\mathit{ReadStoreInfo}_C$ concern slices are expressed by tree topology.

One can notice that calculate concern slices $\mathit{Calculate}_{B \wedge C}$ were not modeled as forward dependent on read store gizmo information concern slice in dependency graph from Figure 1. In the feature model representation however, a forward dependency on the read store gizmo information concern slice $\mathit{ReadStoreInfo}_C$ is modeled. From our observation a forward and backward dependency very often occur together, therefore can be modeled together. In special cases where such approach is undesirable dependencies should be explicitly captured by additional constraints (Section 5.2).

5.2 Dependencies Captured by Additional Constraints

Concern slices can also be modeled in a more common style: as usual system features. This way they would form feature hierarchies that correspond to their position in the system hierarchy. However, dependencies between concern slices would have to be expressed as additional constraints. An example is depicted in Fig. 3. Additional constraints can be expressed using logic expressions [7, 8].

$Register_C \Rightarrow ReadStoreInfo_B$
$Calculate_{B\wedge C} \Rightarrow ReadStoreInfo_B, Register_C$
$Debit_{B\wedge C} \Rightarrow Calculate_{B\wedge C}, ReadStoreInfo_B, Register_C$

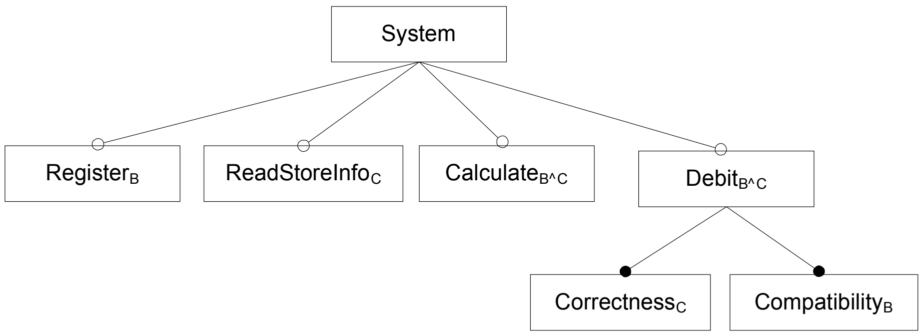


Figure 3. Concern slices as usual system features

This approach is much more compatible with the intended use of the transformed dependency graphs. In the same fashion changes implemented by aspects can be modeled. Therefore, feature models in this form can be used for modeling changes represented as aspects, existing concerns along with the dependencies among them. Feature models offer strong means how to express additional constraints which could occur in special occasions.

5.3 Capturing Behavioral Component of Dependency Graphs

Dependency graphs seam to address behavior, too. This can be seen also from example in Figure 1 where the read store gizmo information concern slice *ReadStoreInfo*$_C$ appears twice. This was noticed also in other studied examples. Feature models are structural and cannot address such behavior. The behavioral component of dependency graphs can be captured with state charts.

State charts are also appropriate if dependency weights which were part of the dependency graph should be preserved. These weights are assigned to dependencies in process of change propagation evaluation. Figure 4 shows an example of state chart for register concern's business rule slice. Concern slices are depicted as states and dependencies and their weights were depicted as transitions.

In this simple example only read store gizmo information concern slice *ReadStoreInfo*$_C$ was duplicated in the original dependency graph. One can notice that in the state chart is this concern slice represented by one state.

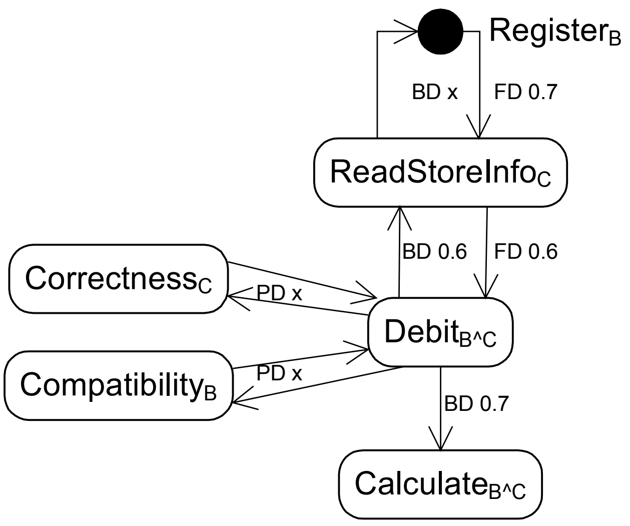


Figure 4. State chart for register *concern's business rule slice*

6 Related Work

This paper proposed two approaches how to transform the dependency graphs into feature models. This is important when dealing with the interaction of aspect-oriented change representations. No other approach of such transformation was found.

Feature models can be used also to capture commonality and variability in the software product lines [4]. They were also successfully used in multi paradigm design for modeling domains [7]. These two approaches are closely connected with aspect-oriented change representations, therefore common modeling technique is essential for future work.

7 Conclusion and Future Work

In this paper, two possible approaches of transforming dependency graphs into feature models have been proposed. The first one captures dependencies primarily by feature diagrams, while the second one does this by additional constraints. It has also been demonstrated how the behavioral component of a dependency graph and weights of dependencies can be captured by state charts.

Capturing dependencies by additional constraints is compatible with feature modeling of changes represented as aspects. This is very important for identifying and evaluating interactions of changes represented by aspects. Feature modeling approach could be used as a unifying modeling technique for modeling aspect-oriented change realizations and dependencies among them.

The future work will be focused on proposition of a feature modeling based technique for identifying interactions of aspect-oriented change realizations on different levels of abstraction during system evolution.

References

[1] Bebjak, M.: Aspektovo-Orientovaná Implementácia Zmien vo Webových Aplikáciách. Master's thesis, Slovenská technická univerzita v Bratislave Fakulta Informatiky a Informacných Technológii, 2007.

[2] Dolog, P., Vranić, V., Bieliková, M.: Representing change by aspect. *SIGPLAN Not.*, 2001, vol. 36, no. 12, pp. 77–83.

[3] Gallagher, K.B., Lyle, J.R.: Using Program Slicing in Software Maintenance. *IEEE Trans. Softw. Eng.*, 1991, vol. 17, no. 8, pp. 751–761.

[4] Lee, K., Kang, K.C., Lee, J.: Concepts and Guidelines of Feature Modeling for Product Line Software Engineering. In: *Proceedings of the Seventh International Conference on Software Reuse*, 2002, pp. 62–77.

[5] Menkyna, R.: Towards Combining Aspect-Oriented Design Patterns. In Bieliková, M., ed.: *IIT.SRC: Student Research Conference 2007*, Slovak University of Technology, 2007, pp. 1–8.

[6] Rashid, S.O., Chitchyan, R., Rashid, A., Khatchadourian, R.: Approach for Change Impact Analysis of Aspectual Requirements, march 2008, AOSD-Europe Deliverable D110, AOSD-Europe-ULANC-40.

[7] Vranić, V.: Reconciling Feature Modeling: A Feature Modeling Metamodel. In Weske, M., Liggsmeyer, P., eds.: *Proc. of 5th Annual International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a Networked World (Net.ObjectDays 2004)*. LNCS 3263, Erfurt, Germany, Springer, 2004, pp. 122–137.

[8] Vranić, V.: Multi-Paradigm Design with Feature Modeling. *Computer Science and Information Systems Journal (ComSIS)*, 2005, vol. 2, no. 1, pp. 79–102.

[9] Weiser, M.: Program slicing. In: *ICSE '81: Proceedings of the 5th international conference on Software engineering*, Piscataway, NJ, USA, IEEE Press, 1981, pp. 439–449.