Slovak University of Technology in Bratislava
Faculty of Informatics and Information Technologies

Aspect-Oriented Change Realization

Valentino Vranić

Habilitačná práca na získanie titulu docent

Habilitation thesis submitted in fulfillment of the requirements
for the Associate Professor degree

April 2010

To Brana and Sandra

Zhrnutie

Zmeny sú nevyhnutnou súčasťou životného cyklu vývoja softvéru. Keďže zmeny prichádzajú v tvare požiadaviek na zmenu, prirodzene sú dobre modularizované na úrovni špecifikácie a pýtajú nejakú formu modularizácie v ďalších fázach vývoja softvéru. Potreba modulárne vyjadriť zmeny sa stáva zjavnejšou v momente, keď zmeny majú byť odstránené alebo aplikované na inú verziu aplikácie, čo predstavuje problém dobre známy v prispôsobovaní softvéru.

Aspektovo-orientované programovanie poskytuje presne taký druh modularity, aký zmeny potrebujú: modularizáciu pretínajúcich záležitostí. Skutočne, tak ako s pretínajúcimi záležitosťami, zmeny často postihujú veľa rozdielnych miest všade v kóde a tradičná modularizácia robí ich kód roztrúseným. Realizované ako aspekty, zmeny sa stávajú zasúvateľnými a reaplikovateľnými.

Táto práca prezentuje môj výskum v oblasti aspektovo-orientovanej realizácie zmien vo forme súboru publikovaných vedeckých prác doplneného komentárom. Tento výskum pokrýva oblasť začínajúcu umiestnením aspektovo-orientovanej paradigmy v kontexte multiparadigmového vývoja softvéru cez preliminárnu štúdiu uskutočniteľnosti aspektovo-orientovanej realizácie zmien až po vývoj dvojúrovňového modelu aspektovo-orientovaných realizácií zmien a, nakoniec, späť k mutliparadigmovému vývoju softvéru v zmysle aplikácie multiparadigmového návrhu s modelovaním vlastností v aspektovo-orientovanej realizácii zmien.

Výskum sa rozširuje a prebieha ďalšia práca, ktorej jedna línia sa sústreďuje na vyjadrenie a vykonávanie aspektovo-orientovanej realizácie zmien na úrovni modelu a transformáciu modelov zmien do kódu. Ďalšia línia ide v smere rozšírenia evaluácie prístupu jeho použitím v ďalších aplikáciách s dôrazom na kolaboráciu viacerých všeobecne aplikovateľných typov zmien a návrhových vzorov a aplikácie tohto prístupu v radoch softvérových výrobkov.

Iné perspektívy zahŕňajú vývoj vyhradenej podpory nástrojom, čo je dôležité pre vysporiadanie sa s interakciou zmien, zvlášť ak je počet zmien veľký, a vývoj katalógov doménovo-špecifických typov zmien iných domén.

Abstract

Changes are an inevitable part of the software lifecycle. Since changes come packaged in the form of change requests, they are naturally well modularized at the specification level begging for some form of modularization in further phases of software development. The need to modularly express changes becomes most apparent at the moment they have to be removed or applied to another version of the application, the problem well-known in software customization.

Aspect-oriented programming provides exactly the kind of modularity changes require: modularization of crosscutting concerns. Indeed, as with crosscutting concerns, changes often affect many different places throughout the code and traditional modularization makes change code scattered. Realized as aspects, changes become pluggable and reapplicable.

This thesis presents my research in the area of aspect-oriented change realization as a collection of papers. This research spans from positioning aspect-oriented paradigm in the context of multi-paradigm software development over a preliminary feasibility study of aspect-oriented change realization towards developing the two-level aspect-oriented change realization model and, finally, back to multi-paradigm software development in the sense of applying multi-paradigm design with feature modeling in aspect-oriented change realization.

The research is expanding and there is further work going on one line of which focuses on expressing and performing aspect-oriented change realization at the model level and transformation of change models into code. Another line goes in the direction of extending the approach evaluation by employing it in further applications stressing the collaboration of multiple generally applicable change types and design patterns and applying this approach in software product lines.

Other perspectives embrace developing dedicated tool support, which is important in dealing with change interaction, especially if the number of changes is high, and developing catalogs of domain specific change types of other domains.

Acknowledgments

I am indebted to Peter Dolog for the initial idea of implementing changes using aspect-oriented programming. The research reported here would not have been possible without the contribution of Michal Bebjak and Radoslav Menkyna.

Contents

Chapter 1

Introduction

Changes are an inevitable part of the software lifecycle. Putting it as a phrase, change is the only constant in software development. Since changes come packaged in the form of change requests, they are naturally well modularized at the specification level begging for some form of modularization in further phases of software development. The need to modularly express changes becomes most apparent at the moment they have to be removed or applied to another version of the application, the problem well-known in software customization.

Aspect-oriented programming provides exactly the kind of modularity changes require: modularization of crosscutting concerns. Indeed, as with crosscutting concerns, changes often affect many different places throughout the code and traditional modularization makes change code scattered. Realized as aspects, changes become pluggable and reapplicable.

The questions that rise from applying aspect-oriented programming in change realization embrace the exact realization of changes and overcoming their interaction. Certain aspect-oriented techniques appear to be especially appropriate for aspect-oriented change realization and may be expressed as code schemes. The exact aspect-oriented technique to be applied depends on the type of a change to be realized which is one possible way to select the appropriate technique.

This thesis maps the research in the area of aspect-oriented change realization which I have conducted over the past years. The thesis is presented as a collection of papers with the main text giving a brief overview of the research referring to the appendices each of which contains one of my papers relevant to aspect-oriented change realization.[1] Figure 1.1 gives a clue of how the papers are related to each other. The edges represent the "builds upon" relationship. The letters in square brackets denote the corresponding appendices.

[1]The text of this thesis relies on the most relevant parts of the papers included in the appendices.

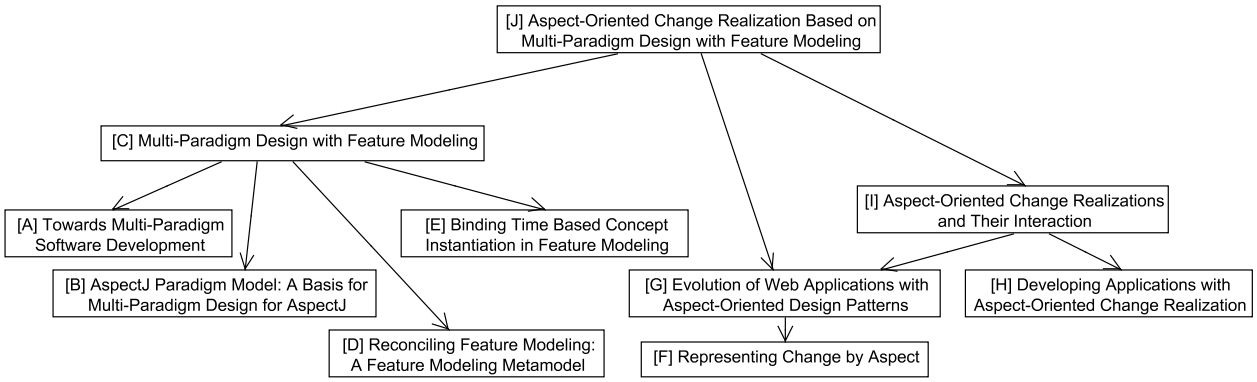


Figure 1.1: The papers included in the thesis and their dependencies.

The rest of the thesis is structured as follows. Chapter 2 gives an insight into the aspect-oriented paradigm as such and how it is related to multi-paradigm software development. Chapter 3 gives an overview of the two-level aspect-oriented change realization model and discusses change interaction. Chapter 4 describes the use of multi-paradigm design in aspect-oriented change realization. Chapter 5 compares the approach presented in this thesis to related work. Chapter 6 concludes the thesis. Appendices A–J include the paper reprints.

Chapter 2

Aspect-Oriented Paradigm

Aspect-oriented software development is a relatively new paradigm coming out from the practical programming needs as a reaction to the problem known from generalized procedure languages [KLM+97], i.e. programming languages that use the concept of the procedure to capture the functionality [Vra02].[1] In such languages the program code fragments that implement a clearly separable aspect of a system (such as synchronization) are scattered and repeated throughout the program code that becomes tangled. Aspect-oriented programming aims at factoring out such aspects into separate program units called by the same name: aspects. Aspects crosscut the base code in places called join points. These must be specified so aspects could be *woven* into the base code by the program called weaver.

This way of seeing aspect-oriented programming is known as PARC AOP (Palo Alto Research Center) and is covered by an industrial strength programming language: AspectJ. Although three other distinguished aspect-oriented approaches—subject-oriented programming, composition filters, and adaptive programming—have not had such a success in practice, they demonstrated that aspect-oriented paradigm is far more complex than the narrowed, though practical PARC AOP.

What is suppressed in PARC AOP and what comes out as essential in aspect-oriented analysis and design is the symmetric approach to aspect-oriented software development. Instead of distinguishing between the base and aspects that influence this base, which is at heart of AspectJ as the leading asymmetric aspect-oriented programming language, the symmetric approach decomposes software in a number of independently developed views none of which is declared as base. These views are then composed in various ways to get the desired functionality. This is a more general view of aspect-oriented software development as an advanced software decomposition and composition paradigm.

[1] Besides the procedural languages, these include functional and object-oriented languages as well.

Section 2.1 puts aspect-oriented software development into the context of multi-paradigm software development. Section 2.2 presents the basic idea of the AspectJ paradigm model.

2.1 Multi-Paradigm Software Development

Aspect-oriented paradigm sheds another light on what is known as multi-paradigm programming or software development in general. Being built mostly upon object-oriented paradigm puts this approach into a position of being a natural multi-paradigm approach [Vra02].

In software development, a paradigm usually denotes the essence of a software development process (often referred to as *programming*) [Vra02]. However, often it is not easy to determine what this essence exactly is and paradigm definitions change over time (not so long ago inheritance was not considered crucial for object-oriented programming).

There is another, more practical view of paradigms: they can be 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].

This perception of paradigm is apparent in Coplien's multi-paradigm design [Cop99] in which a paradigm is viewed as a configuration of commonality and variability [Cop99]. This is analogous to conjugation or declension in natural languages, where the common is the root of a word and variability is expressed through the suffixes or prefixes (or even infixes) added to the root in order to obtain different forms of the word.

Scope, commonality and variability (SCV) analysis [CHW98] can be used to describe these language level paradigms. SCV analysis of procedures paradigm illustrates the definition (based on an example by Coplien et al. [CHW98]):

S: a collection of similar code fragments, each to be replaced by a call to some new procedure P;

C: the code common to all fragments in S;

V: the "uncommon" code in S; variabilities are handled by parameters to P or custom code before or after each call to P.

Appendix A gives a more detailed view of multi-paradigm software development.

2.2 AspectJ Paradigm Model

AspectJ can be described in terms of small-scale paradigms [Vra01]. Its paradigms specific to aspect-oriented programming are aspect, advice, and inter-type declaration.

In multi-paradigm with feature modeling (MPD_{FM}), feature modeling is used to express paradigms [Vra05, Vra04]. 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 [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 choosing not to bind) 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. 2.1. 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 represent 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]). To achieve its intent, an aspect may—similarly to a class—employ Methods (with the method being 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. 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 be selected. An aspect can be Abstract, in which case it cannot be instantiated, so it cannot have Instantiation Policy either, which is again modeled by mandatory alternative features.

An aspect can be declared to be Static or Final. It does not have to be either of the two, but it cannot be both, which is modeled by optional alternative features of which only one may be selected (indicated by an arc

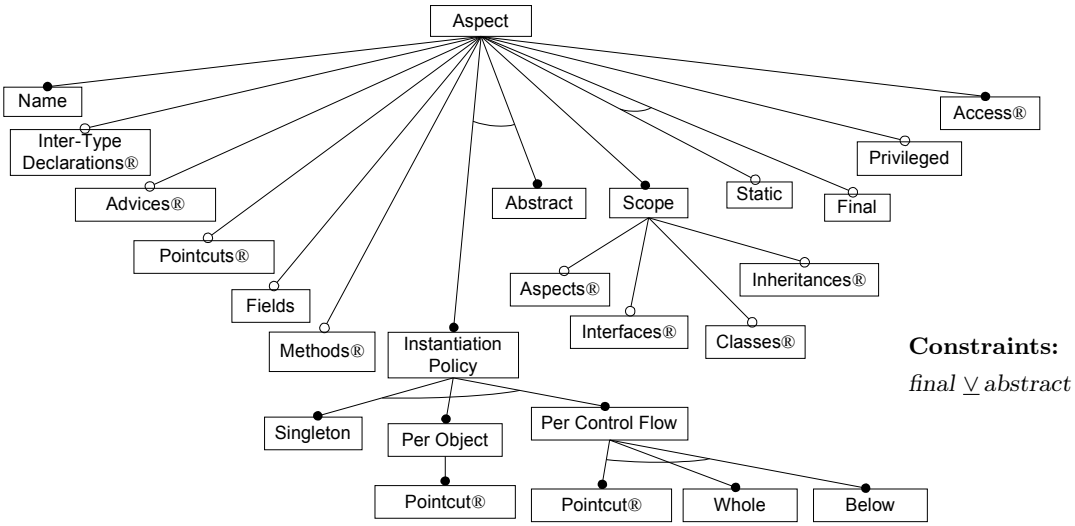


Figure 2.1: The AspectJ aspect paradigm [Vra05].

over optional features). An aspect can also be Privileged over other aspects and it has its type of Access, which is modeled as a reference to a separately expressed auxiliary concept. All the features in the aspect paradigm are bound at source time.

The constraint associated with the aspect paradigm feature diagram means that the aspect is either Final or Abstract. Here, first-order predicate logic is used to express constraints associated with feature diagrams, but OCL could be employed, too, as a widely accepted and powerful notation for such uses.

An initial AspectJ paradigm model may be found in Appendix B. Appendix C includes elaborated versions of aspect-oriented paradigms in AspectJ, and the complete AspectJ paradigm model is also available [Vra03]. Appendix D explains the details of feature modeling.

Chapter 3

Changes as Aspects

Change realization consumes enormous effort and time during software evolution. Once implemented, changes get lost in 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 [VBMD08].

By its capability to separate crosscutting concerns, aspect-oriented programming enables to deal with changes explicitly and directly at the 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.

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.

Section 3.1 presents the customization of applications as a motivating example for aspect-oriented change realization. Section 3.2 gives an overview of the two-level aspect-oriented change realization model. Section 3.3 discusses change interaction.

3.1 The Customization Problem

As a motivating example for aspect-oriented change realization, consider customization of applications. In customization, a general application is being adapted to the client's needs by a series of changes [VBMD08]. 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. Figure 3.1 shows schemati-

cally this process on an example.

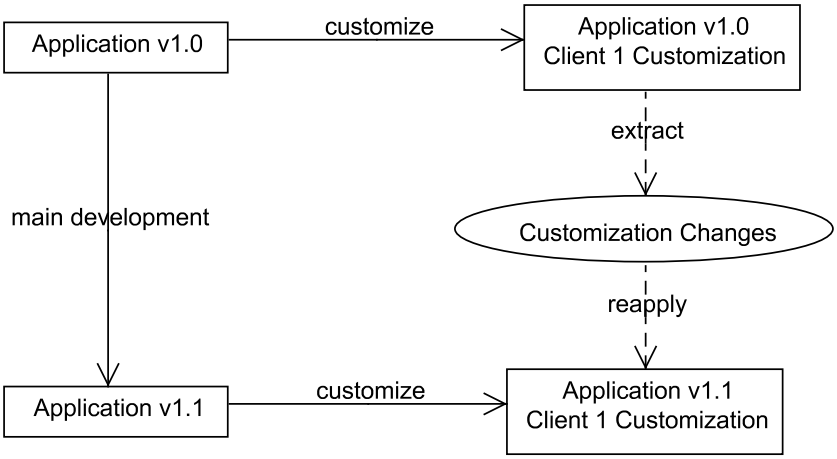


Figure 3.1: The customization problem.

It has been demonstrated how such an aspect-oriented customization may be of use in solving a problem of synchronizing a local customization with the global version of a program in script languages. For this, a small procedural aspect-oriented extension has been proposed covering the introduction of a new procedure or (global) variable into a module, extension of a procedure by a code before, after, or instead of it, and change of procedure arguments and return value [DVB01].

Appendix F presents an initial effort towards establishing an approach to aspect-oriented change realization along with the procedural aspect-oriented extension.

3.2 Two-Level Aspect-Oriented Change Realization Model

To realize changes using aspect-oriented programming effectively, a two-level aspect-oriented change realization model has been proposed [BVD07, VBMD08]. 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 3.2 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.

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,

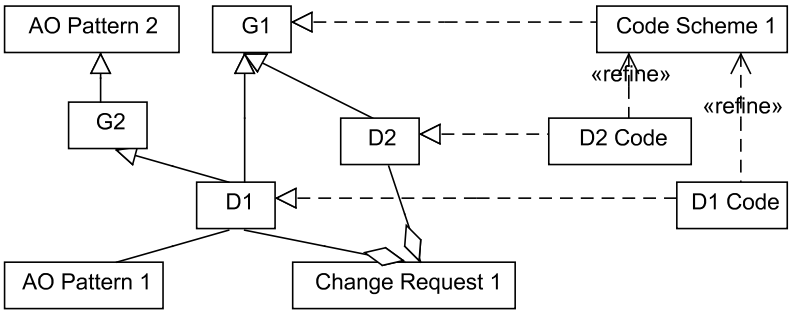


Figure 3.2: Generally applicable and domain specific changes [VBMD08].

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).

The following catalog sums up these relationships between change types that have been identified in the web application domain (the domain specific change type is introduced first) [VBMD08]:

- 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

Consider a simple scenario [VMBD09] needed to present an example of using the two level change type model. Suppose a merchant who runs his online music shop purchases a general affiliate marketing software [GJH03] 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.

Assume that there is a need to integrate an affiliate marketing software with the third party newsletter. 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 [BVD07]. 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 [BVD07]. 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 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. We can seamlessly combine multiple one way integrations to integrate with several systems.

Appendix G presents the study of aspect-oriented change realization based on the two-level aspect-oriented change realization model in web application evolution. Appendix H describes the approach to aspect-oriented change realization in detail (an extended version of the paper from this appendix may be found in Appendix I).

3.3 Change Interaction

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 an increasing number of changes, change interaction can easily escalate into a serious problem: serious as feature interaction [VMBD09].

Change realizations in the sense of the approach presented so far actually resemble features as coherent pieces of functionality [MV09]. 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.

Aspect-oriented change realizations can be perceived as variable features that extend an existing system. Figure 3.3 shows the change realizations from our affiliate marketing scenario in 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.

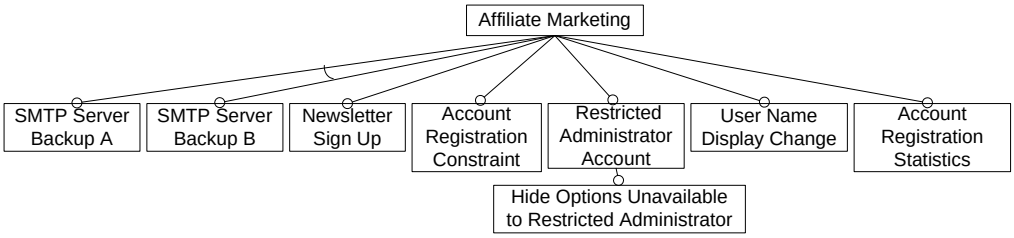


Figure 3.3: Affiliate marketing software change realizations in a feature diagram [MV09].

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.

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 to reveal indirect dependencies among

them.

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.

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 3.4 depicts this process.

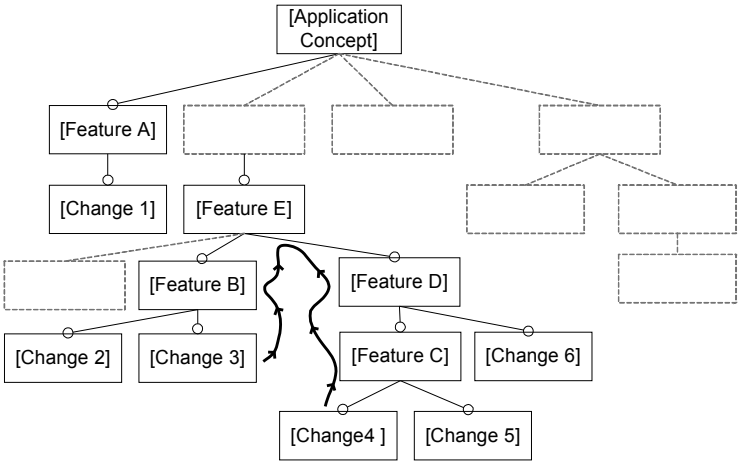


Figure 3.4: Constructing a partial feature model [VMBD09].

Figure 3.5 shows the result of this process applied to the initial affiliate marketing partial feature model.

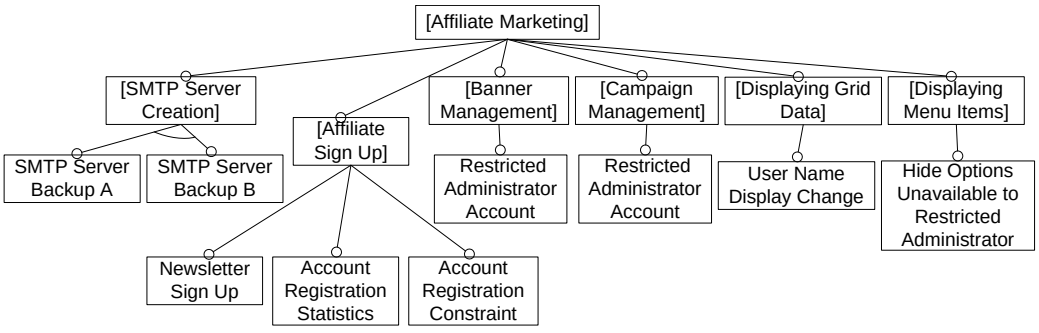


Figure 3.5: Affiliate marketing software change realizations in a feature diagram [MV09].

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 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.

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 [Vra05].

Appendix I describes the details of examining interaction of aspect-oriented change realizations, as well as the process of partial feature model construction. Appendix J develops and applies the idea of partial feature model in the context of using multi-paradigm design with feature modeling in aspect-oriented change realization.

Chapter 4

Applying Multi-Paradigm Design to Change Realization

The previous chapter presented the approach to aspect-oriented change realization based on a two-level aspect-oriented change realization 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. 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 the time and effort needed. The problem of selecting a suitable realizing change type resembles paradigm selection in multi-paradigm design mentioned (MPD_{FM}) in Chapter 2 [MV09]. This is the other way around: to treat change realization types as paradigms and employ multi-paradigm design to select the appropriate one.

Section 4.1 explains how aspect-oriented change realization types can be expressed as paradigms in MPD_{FM}. Section 4.2 describes the basics of transformational analysis as used to get to the corresponding change realization types.

4.1 Generally Applicable Change Types as Paradigms

Generally applicable change types (introduced in Section 3.2) 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 change types that would cover all significant situations is not high. In our experiments, we managed to cope with all situations using only six of them [MV09].

On the other hand, in the domain of web applications, eleven application specific change types we identified so far cover it only partially. Each such change type requires a thorough exploration in order to discover all possible

realizations by generally applicable change types 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.

In MPD_{FM}, feature modeling is used to express paradigms. 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 shown in Fig. 2.1.

Generally applicable changes may be seen as a kind of conceptually higher language mechanisms and modeled as paradigms in the sense of MPD_{FM}. As an example, consider the Performing Action After Event change type mentioned in Section 3.2. Its paradigm model is presented in Figure 4.1. All the features have source time binding. This change type is used when an additional action (Action After Event) is needed after some events (Events) of method calls or executions, initialization, field reading or writing, or advice execution (modeled as or-features) taking or not into account their context (Context).

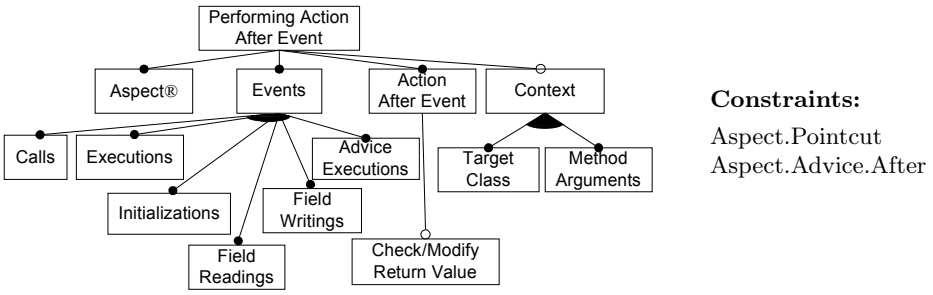


Figure 4.1: Performing Action After Event [MV09].

Performing Action After Event is implemented by an aspect (Aspect) with a pointcut specifying the events and an after advice over this pointcut used to perform the desired actions, which is expressed by the constraints associated with its feature diagram (Fig. 4.1).

Appendix J introduces the paradigm model of the Method Substitution change type, too.

4.2 Transformational Analysis of Changes

Transformational analysis is the process of finding the correspondence and establishing the mapping between the application and solution domain concepts [Cop99]. In MPD_{FM}, it is based on paradigm instantiation (feature model configuration) over application domain concepts [Vra05]. The input to the transformational analysis are two feature models: the application do-

main one and the solution domain one. The output of the transformational analysis is a set of paradigm instances annotated with application domain feature model concepts and features that define the code skeleton.

A simplified transformational analysis can be used to determine general change types that correspond to the domain specific changes. Changes presented in the application domain feature model are considered to be application domain concepts. Generally applicable change types are considered to be paradigms. Roughly, the process of transformational analysis For each change from the application domain feature model the subtree in which it resides is taken and change types are instantiated until a match for the change feature is found.

As has been discussed in Section 3.3, when dealing with changes in feature models, it is sufficient to rely on partial feature models. For the purposes of transformational analysis, the rudimentary partial feature model has to be developed further to uncover parent features of the change features as the features of the underlying system affected by them. Starting at change features, we proceed bottom up identifying their parent features until related features become grouped in common subtrees.

Figure 4.2 shows the transformational analysis of the Newsletter Sign Up change. Recall that this change adds a new affiliate to the existing list of newsletter recipients, which can be best realized as Performing Action After Event. In this case, the Events feature is mapped to the Affiliate Sign Up feature which represents the execution of the affiliate sign up method. Through Method Arguments, the data about the affiliate being added can be accessed (Affiliate Data) from which his e-mail address can be retrieved and subsequently added to the newsletter recipient list by the Action After Events feature. A similar transformation would apply to the Account Registration Statistics change. This solution corresponds to the cataloged one as presented in Section 3.2.

Appendix J presents the details of using MPD_{FM} to deal with aspect-oriented change realization along with a precise description of the simplified transformational analysis. The transformational analysis as defined for the purposes of multi-paradigm design can be found in Appendix C. Appendix D explains the details of feature modeling. Appendix E deals with the specific issue of concept instantiation in time on which is the MPD_{FM} transformational analysis based.

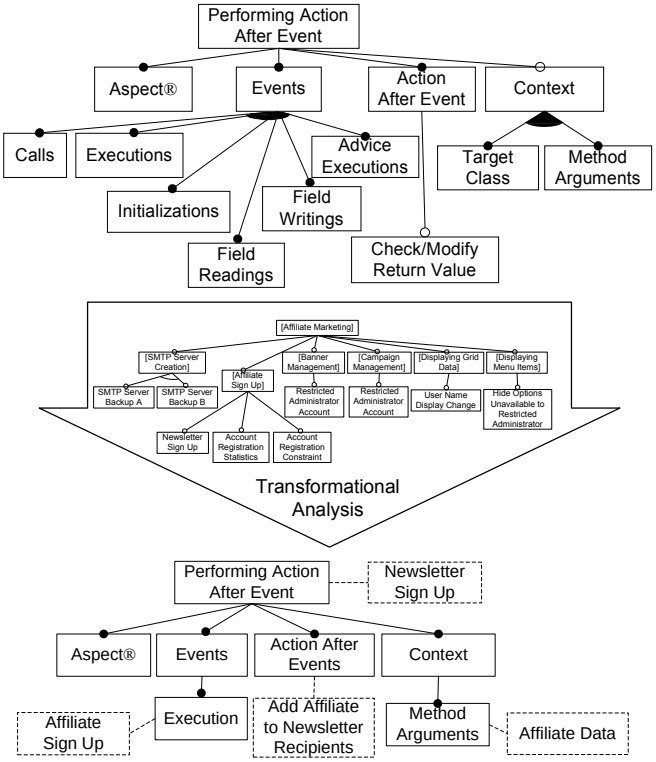


Figure 4.2: Transformational analysis of the Newsletter Sign Up change [MV09].

Chapter 5

Related Work

It has been shown that changes can be implemented using aspect-oriented programming even if the source code to be changed is not available [BB08a, BB08b]. Others have explored several issues generally related to aspect-oriented change realization, although not aiming at capturing changes by aspects. These issues include database schema evolution with aspects [GR02] or aspect-oriented extensions of business processes and web services with crosscutting concerns of reliability, security, and transactions [CSHM06]. Also, an increased changeability of components implemented using aspect-oriented programming [KLC05, LKC06, PP04] and aspect-oriented programming with the frame technology [LRZJ04], as well as enhanced reusability and evolvability of design patterns achieved by using generic aspect-oriented languages to implement them [RK06] have been reported.

The impact of changes implemented by aspects has been studied using slicing in concern slice dependency graphs [KR06]. It has been shown that the application domain feature model can be derived from concern slice dependency graphs [Men09]. 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 program slicing to features implemented as aspects with interaction understood as a slice intersection has been applied so far only to a very simplified version of AspectJ. Extension to cover complicated constructs has been identified as problematic. Even at this simplified level, it appears to be too coarse for applications in which the behavior is embedded in data structures [MBB03].

In the approach to aspect-oriented change realization presented here, changes are regarded as concerns, which is similar to the approach of facilitating configurability by separation of concerns in the source code [Faz05]. 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.

Applying feature modeling to maintain change dependencies is similar to constraints and preferences proposed in SIO software configuration management system [CW98]. However, a version model for aspect dependency management [PSC01] with appropriate aspect model that enables to control aspect recursion and stratification [BFS06] would be needed as well.

While the automatic configuration and reconfiguration of applications certainly represents a potential, this work does not aim at automatic adaptation in application evolution, such as event triggered evolutionary actions [MOMMGC06], evolution based on active rules [DMP06], or adaptation of languages instead of software systems [KPV+07a].

Even if the original application has not been a part of a product line, changes modeled as its features tend to form a kind of a product line out of it. This could be seen as a kind of evolutionary development of a new product line [Bos00].

As an alternative to the transformational analysis presented here, framed aspects [LRZJ04, LSR05] can be applied to the application domain feature model with each change maintained in its own frame in order to keep it separate. Annotations that determine the feature implementation in so-called *crosscutting feature models* [KGBL05] are similar to annotations used in the transformational analysis, but no formal process to determine them is provided.

An approach to introduce program changes by changing the interpreter instead based on grammar weaving has been reported [FK09]. With respect to suitability of aspect-oriented approach to deal with changes, it is worth mentioning that weaving—a prominent characteristic of aspect-oriented programming—has been identified as crucial for the automation of multi-paradigm software evolution [KPV+07b].

Chapter 6

Conclusions

This thesis presented my research in the area of aspect-oriented change realization. This research spans from positioning aspect-oriented paradigm in the context of multi-paradigm software development over a preliminary feasibility study towards developing the two-level aspect-oriented change realization model and, finally, back to multi-paradigm software development in the sense of applying multi-paradigm design with feature modeling in aspect-oriented change realization.

The research is expanding and there is further work going on one line of which focuses on expressing and performing aspect-oriented change realization at the model level (based on the Theme notation of aspect-oriented analysis and design [CB05]) and transformation of change models into code. Another line goes in the direction of extending the approach evaluation by employing it in further applications stressing the collaboration of multiple generally applicable change types and design patterns [MVP10] and applying this approach in software product lines [KV10].

Further perspectives embrace developing dedicated tool support, which is important in dealing with change interaction, especially if the number of changes is high, and developing catalogs of domain specific change types of other domains.

Bibliography

[BB08a] Ilona Bluemke and Konrad Billewicz. Aspect modification of an EAR application. In *Proc. of International Joint Conferences on Computer, Information, and Systems Sciences, and Engineering, CIS^{2}E 08*, Krakow, Poland, December 2008. Springer. To appear.

[BB08b] Ilona Bluemke and Konrad Billewicz. Aspects in the maintenance of complied program. In *Proc. of 5th International Conference on Dependability of Computer Systems, DepCoS-RELCOMEX 2008*, pages 253–260, Szklarska Poręba, Poland, June 2008. IEEE.

[BFS06] Eric Bodden, Florian Forster, and Friedrich Steimann. Avoiding infinite recursion with stratified aspects. In Robert Hirschfeld et al., editors, *Proc. of NODe 2006*, LNI P-88, pages 49–64, Erfurt, Germany, September 2006. GI.

[Bos00] Jan Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. Addison-Wesley, 2000.

[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.

[CB05] Siobhán Clarke and Elisa Baniassad. *Aspect-Oriented Analysis and Design: The Theme Approach*. Addison-Wesley, 2005.

[CHW98] James Coplien, Daniel Hoffman, and David Weiss. Commonality and variability in software engineering. *IEEE Software*, 15(6), November 1998.

[Cop99] James O. Coplien. *Multi-Paradigm Design for C++*. Addison-Wesley, 1999.

[CSHM06] Anis Charfi, Benjamin Schmeling, Andreas Heizenreder, and Mira 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, December 2006. IEEE Computer Society.

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

[DMP06] Florian Daniel, Maristella Matera, and Giuseppe 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.

[DVB01] Peter Dolog, Valentino Vranić, and Mária Bieliková. Representing change by aspect. *ACM SIGPLAN Notices*, 36(12):77–83, December 2001.

[Faz05] Zoltan Fazekas. Facilitating configurability by separation of concerns in the source code. *Journal of Computing and Information Technology (CIT)*, 13(3):195–210, September 2005.

[FK09] Michal Forgáč and Ján Kollár. Adaptive approach for language modification. *Journal of Computer Science and Control Systems*, 2(1):9–12, 2009.

[GJH03] Simon Goldschmidt, Sven Junghagen, and Uri Harris. *Strategic Affiliate Marketing*. Edward Elgar Publishing, 2003.

[GR02] Robin Green and Awais 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, April 2002.

[KGBL05] Uirá Kulesza, Alessandro Garcia, Fábio Bleasby, and Carlos Lucena. Instantiating and customizing aspect-oriented architectures using crosscutting feature models. In *Workshop on Early Aspects held with OOPSLA 2005*, San Diego, USA, November 2005. Available at `http://www.early-aspects.net/oopsla05ws/`.

[KLC05] Axel Anders Kvale, Jingyue Li, and Reidar 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.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Christina Vidiera Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proc. of 11th European Conference on Object-Oriented Programming (ECOOP'97)*, LNCS 1241, Jyväskylä, Finland, June 1997. Springer.

[KPV+07a] Ján Kollár, Jaroslav Porubän, Peter Václavík, Jana Bandáková, and Michal Forgáč. Functional approach to the adaptation of languages instead of software systems. *Computer Science and Information Systems Journal (ComSIS)*, 4(2), December 2007.

[KPV+07b] Ján Kollár, Jaroslav Porubän, Peter Václavík, Marcel Tóth, Jana Bandáková, and Michal Forgáč. Multi-paradigm approaches to systems evolution. In *Computer Science and Technology Research Survey*, Košice, Slovakia, 2007.

[KR06] Safoora Khan and Awais 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.

[KV10] Ján Kohut and Valentino Vranić. Guidelines for using aspects in product lines. In *Proc. of 8th International Symposium on Applied Machine Intelligence and Informatics, SAMI 2010*. IEEE, January 2010.

[LKC06] Jingyue Li, Axel Anders Kvale, and Reidar 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, March 2006.

[LRZJ04] Neil Loughran, Awais Rashid, Weishan Zhang, and Stan 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, March 2004.

[LSR05] Neil Loughran, Américo Sampaio, and Awais Rashid. From requirements documents to feature models for aspect oriented product line implementation. In *MDD for Software Product-lines: Fact or Fiction?, a workshop held with ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems, MoDELS/UML 2005)*, Montego Bay, Jamaica, October 2005.

[MBB03] Mattia Monga, Fatima Beltagui, and Lynne Blair. Investigating feature interactions by exploiting aspect oriented programming. Technical Report comp-002-2003, Lancaster University, Lancaster, UK, 2003. Available at `http://www.comp.lancs.ac.uk/computing/aose/`.

[Men09] Radoslav Menkyna. Dealing with interaction of aspect-oriented change realizations using feature modeling. In Mária Bieliková, editor, *Proc. of 5th Student Research Conference in Informatics and Information Technologies , IIT.SRC 2009*, Bratislava, Slovakia, April 2009.

[MOMMGC06] Fernando Molina-Ortiz, Nuria Medina-Medina, and Lina 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.

[MV09] Radoslav Menkyna and Valentino Vranić. Aspect-oriented change realization based on multi-paradigm design with feature modeling. In *Proc. of 4th IFIP TC2 Central and East European Conference*

on Software Engineering Techniques, CEE-SET 2009, Krakow, Poland, October 2009. Postproceedings, to appear.

[MVP10] Radoslav Menkyna, Valentino Vranić, and Ivan Polášek. Composition and categorization of aspect-oriented design patterns. In *Proc. of 8th International Symposium on Applied Machine Intelligence and Informatics, SAMI 2010*, Herľany, Slovakia, January 2010. IEEE.

[PP04] Odysseas Papapetrou and George 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.

[PSC01] Elke Pulvermüller, Andreas Speck, and James 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, September 2001. Springer.

[RK06] Tobias Rho and Günter 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, April 2006.

[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.

[VMBD09] Valentino Vranić, Radoslav Menkyna, Michal Bebjak, and Peter Dolog. Aspect-oriented change realizations and their interaction. *e-Informatica Software Engineering Journal*, 3(1):43–58, 2009.

[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.

[Vra02] Valentino Vranić. Towards multi-paradigm software development. *Journal of Computing and Information Technology (CIT)*, 10(2):133–147, 2002.

[Vra03] Valentino Vranić. *Multi-Pradigm Design with Feature Modeling.* PhD thesis in preparation, Slovak University of Technology in Bratislava, Slovakia, 2003.

[Vra04] 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.

Appendix A

Towards Multi-Paradigm Software Development

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

Towards Multi-Paradigm Software Development

Valentino Vranić

Department of Computer Science and Engineering, Faculty of Electrical Engineering and Information Technology, Slovak University of Technology in Bratislava, Slovakia

Multi-paradigm software development is a possible answer to attempts of finding the best paradigm. It was present in software development at the level of intuition and practiced as the "implementation detail" without a real support in design. Recently it is making a twofold breakthrough: several recent programming paradigms are encouraging it, while explicit multi-paradigm approaches aim at its full-scale support. In order to demonstrate this, a survey of selected recent software development (programming) paradigms (aspect-oriented approaches and generative programming) and multi-paradigm approaches (multi-paradigm programming in Leda, multi-paradigm design in C++, and intentional programming) is presented.

The survey is preceded and underpinned by the analysis of the concept of *paradigm* in computer science in the context of software development, since there is no common agreement about the meaning of this term, despite its wide use. The analysis has showed that there are two meanings of paradigm: large-scale and small-scale.

Keywords: software development, programming, large-scale, small-scale paradigm; commonality, variability analysis; multi-paradigm, aspect-oriented, generative, Leda, intentional programming; metaparadigm.

1. Introduction

The way software is developed is changing. Enforced by the need for mass production of quality software and enabled by the grown experience of the field, software development is moving towards industrialization. The question is no longer which single tool is the best one, but how to select the right tools for each task to be accomplished.

This article maps the state of the art in the field of post-object-oriented software development. Most notably, it is devoted to the promising concepts of aspect-oriented programming, generative programming and, particularly, to multi-paradigm software development.

The move towards multi-paradigm software development can be felt not only in new software development paradigms — e.g. aspect-oriented programming, which is bound to other paradigms from the first principles — it is present already in object-oriented programming. It is even more notable at the language level. It is hard to find a language that is pure in the sense of prohibiting any other than its proclaimed paradigm to be used in it.

What has been described is the implicit form of multi-paradigm software development. There are several approaches which make this idea explicit by enabling a developer not only to combine multiple paradigms, but also to choose the most appropriate one for the given feature of a system or family of systems.

The rest of this article is organized as follows. Section 2 explores the concept of paradigm in computer science in the context of software development. Section 3 is an overview of selected recent post-object-oriented paradigms, namely aspect-oriented programming approaches and generative programming. Section 4 proceeds with recent post-object-oriented approaches that exhibit explicitly the multi-paradigm character. Section 5 closes the article with conclusions and proposals for further work.

2. The Concept of Paradigm in Software Development

Paradigm is a very often used — and even more often abused — word in computer science in the context of software development. Its importance arose significantly with the appearance of so-called *multi-paradigm* approaches. Before discussing them, the concept of paradigm in software development requires a deeper examination. We must consider both the well-established meaning of paradigm in science and the actual meaning of the word in order to find out when its use in computer science is justified, and also to gain a better understanding of the concept of paradigm itself.

The term paradigm in science is strongly related to Thomas Kuhn and his essay [Kuh70], where it is used to denote a consistent collection of methods and techniques accepted by the relevant scientific community as a prevailing methodology of the specific field.

In computer science, the term paradigm denotes the essence of a software development process (often referred to as *programming*, see Section 2.1). Unfortunately, this is not the only purpose this term is used for. Probably no science has accepted this term with such an enthusiasm as computer science has; there are a lot of methods whose authors could not resist the temptation to raise them to the level of paradigm (just try a keyword "paradigm" in some citing index or digital library, e.g. [NEC]). Although not contradictory to the original meaning of the word paradigm, such an overuse causes confusion.

The basic meaning of paradigm is *example*, especially a typical one, or *pattern*, which is in a direct connection to its etymology (Greek "to show side by side") [Mer]. The meaning and etymology pose no restriction to the extent of the example or pattern it refers to. This is reflected in the common use of the word paradigm today: on the one hand, it has the meaning of a whole philosophical and theoretical framework of scientific school (akin to Kuhn's interpretation), while on the other hand, it is simply an example as in linguistics where it has the meaning of an example of conjugation or declension showing a word in all its inflectional forms [Mer].

Computer science, being a *science* whose great part is devoted to a special kind of *languages* intended for programming, hosts well both of these two interpretations of paradigm covered in more detail in the following text.

2.1. Large-Scale Paradigms

The notion of paradigm in the context of software development is used at two levels of granularity. Let us first discuss the *large-scale* meaning of paradigm, which, as it has already been mentioned, denotes the essence of a software development process. Coplien used the term large-scale paradigm to denote programming paradigms in, as he said, a "popular" sense [Cop99a].

Besides software development paradigm and software engineering paradigm, at least two more terms are used to refer to large-scale paradigm of software development: *programming paradigm* or, simply, *programming*. Although in common use (for historical reasons), one must be carful with these terms because of possible misunderstanding: programming sometimes stands for implementation only, as other phases of a software development process can also be referred to explicitly (e.g., object-oriented *analysis*, *object-oriented* design, etc.).

The name of a paradigm reveals its most significant characteristic [Vra00]. Sometimes it is derived from the central abstraction the paradigm deals with, as it is a function to functional paradigm, an object to object-oriented paradigm (according to [Mey97] it is not *object* but *class* that is the central abstraction in object-oriented paradigm), etc.

Lack of a general agreement on which name denotes which paradigm is a potential source of confusion. For example, although the term *functional paradigm* is usually used to denote a kind of application paradigm, as opposed to procedural paradigm, in [Mey97] it is used to denote exactly the procedural paradigm. It is hard to blame the author for misuse of the term knowing that the procedure is often being denoted as *function*.

It must be distinguished between the software development paradigm itself and the means used to support its realization. Unfortunately, this is another source of confusion. For example, any paradigm can be visualized by means of a visual environment and thus it makes no sense to

speak about the visual paradigm (as in [Bud95]). Making a complete classification and comparison of the software development paradigms is beyond the scope of this text; see [Náv96] for the comparison of selected programming paradigms regarding the concepts of abstraction and generalization.

A software development paradigm is constantly changing, improving, or better to say, refining. The basic principles it lays on must be preserved; otherwise it would evolve into another paradigm. Consider, for example, the simplified view on the evolution of object-oriented paradigm. First, there were commands (imperative programming). Then named groups of commands appeared, known as procedures (procedural programming). Finally, procedures were incorporated into the data it operated on yielding objects/classes (object-oriented paradigm).

However, according to Kuhn, paradigms do not evolve, although it could seem so; it is the *scientific revolution* that ends the old and starts a new paradigm [Kuh70]. A paradigm is *dominant* by definition and thus there can be only one at a time in a given field of science unless the field is in an unstable state. According to this, simultaneous existence of several software development paradigms indicates that the field of software development is either in an unstable state, or all these paradigms are parts of the one not yet fully recognized nor explicitly named paradigm. That paradigm is beyond the commonly recognized paradigms and it is about the (right) use and combination of those paradigms. Therefore it can be denoted as *metaparadigm*.

2.2. Small-Scale Paradigms

The notion of paradigm in computer science can also be considered at the small-scale level based on the programming language perspective. This perception of paradigm is apparent in James O. Coplien's multi-paradigm design [Cop99b] (covered in more detail in Section 4.2). According to Coplien et al. [CHW98], we can factor out paradigms such as procedures, inheritance and class templates. We can identify the common and variable part which together constitute a paradigm. A paradigm is then a *configuration of commonality and variability* [Cop99b]. This is analogous to conjugation or declension in natural languages, where the common is the root of a word and variability is expressed through the suffixes or prefixes (or even infixes) added to the root in order to obtain different forms of the word.

Scope, commonality and variability (SCV) analysis [CHW98] can be used to describe these language level paradigms. Here are the definitions of the three cornerstone terms in SCV analysis (instead of *entities* the word *objects* was used in [CHW98], but this could lead to a confusion with objects in the sense of object-oriented paradigm):

Scope (*S*): a set of entities;

Commonality (*C*): an assumption held uniformly across a given set of entities *S*;

Variability (*V*): an assumption true for only some elements of *S*.

SCV analysis of *procedures* paradigm illustrates the definition (based on an example from [CHW98]):

***S*:** a collection of similar code fragments, each to be replaced by a call to some new procedure *P*;

***C*:** the code common to all fragments in *S*;

***V*:** the "uncommon" code in *S*; variabilities are handled by parameters to *P* or custom code before or after each call to *P*.

In the context of the small-scale paradigms, it is hard to find a single-paradigm programming language. The relationship between the small- and large-scale paradigms is similar to that between the programming language features and programming languages; the large-scale paradigms consist of the small-scale ones. We can revise here the source of the name of a large-scale paradigm: the name of a large-scale paradigm sometimes comes from the most significant small-scale paradigm it contains. For example, object-oriented (large-scale) paradigm consists of several (small-scale) paradigms: object paradigm, procedure paradigm (methods), virtual functions, polymorphism, overloading, inheritance, etc. Lack of a common agreement what are the exact characteristics of object-oriented paradigm makes it impossible to introduce the exact list of the small-scale paradigms that object-oriented paradigm consists of.

Having an expressive programming language that supports multiple paradigms introduces another issue: a method is needed for selecting the right paradigms for the features that are to be implemented. Such a method is a *metaparadigm* with respect to the small-scale paradigms. The small-scale paradigms metaparadigm is therefore a less elusive concept than the large-scale paradigms metaparadigm. One such small-scale metaparadigm, multi-paradigm design for C++, is described in Section 4.2.

One can understand small-scale paradigms as a programming language issue exclusively, while large-scale programming paradigms seem to have a broader meaning as they are affecting all the phases of software development. Actually, small-scale paradigms have an impact on all the phases of software development as well; either with or without an explicit support in analysis and design.

3. Recent Software Development Paradigms

Among the recent software development paradigms there is a significant group of those that appeared as a reaction to the issues tackled but not satisfactorily solved by object-oriented paradigm. Many of these paradigms actually build upon object-oriented paradigm. Despite some of them are claimed not to be bound to object-oriented paradigm (and, indeed, they are more generally applicable), they are still widely applied in connection with it.

3.1. Beyond Object-Oriented Programming

Human perception of the world is to the great extent based on objects. Object-oriented programming, well-known under the acronym OOP, is based precisely on this perception of the world natural to humans. But what is OOP exactly? This question seems to be an answered one. Actually, there are plenty of answers to this question, but the trouble is that they are all different. OOP has passed a very long way of changes to reach the form in which it is known today. Yet, there is no general agreement about what its essential properties are (to some, even inheritance is not an essential property of OOP, or it is being denoted as a minor feature [Bud95]). Perhaps Bertrand Meyer's viewpoint that "'object-oriented' is not a boolean condition" [Mey97] is the best characterization of this issue.

OOP is not always the best choice among all the paradigms. This is recognized even in the OOP literature. Thus Booch points out that there is no single paradigm best for all kinds of applications. But, according to Booch, OOP has another important feature: it can serve as "the architectural framework in which other paradigms are employed" [Boo94]. Although this statement is probably overestimated in its applicability to all the paradigms, the truth is that some multi-paradigm languages (like Leda, see Section 4.1) are designed in this fashion. This reveals that OOP is multi-paradigmatic in its very nature and leaves not much space for the object-oriented purism.

The object-oriented purism comes from the dogma that everything should be modeled by objects. But not everything is an object; neither in the real world, nor in programming. Consider synchronization as a well-known example of a non-object concept; in natural language, we would probably refer to it as *aspect*. The aspects crosscut the structure of objects (or functional components, in general) making the code tangled. The pieces of code are either repeated throughout different objects or unnatural inheritance must be involved. Among other inconveniences, this "code scattering" has a bad impact on reuse.

There are also other problems with OOP, including those it was supposed to solve, which are mainly in the areas of reuse (discussed in [SN97]), adaptability, management of complexity and performance [CE00]. In the sense of the means for solution at the developer's disposal — that can be denoted as solution universe — OOP is not a universal paradigm. For example, OOP is not a universal paradigm either in C++ because it is not capable of making a full use of all of its features, or in C++ which is just a part of the solution universe of software development. OOP encompasses only a few interesting kinds of commonality and variability [Cop99a]. Other kinds are needed as well, so the non-object-oriented features of programming languages are often used even though the analysis and design were object-oriented.

```
class Point {
  int x,y;
  Point(int x, int y){...}
  void set(int x, int y){...}
  void setX(int x){...}
  void setY(int y){...}
  int getX(){...}
  int getY(){...}
}

class Line {
  int x1,y1,x2,y2;
  Line(int x1, int y1, int x2, int y2){...}
  void set(int x1, int y1, int x2, int y2){...}
  int getX1(){...}
  int getY1(){...}
  int getX2(){...}
  int getY2(){...}
}

aspect ShowAccesses {
  before(): execution(* (Point || Line).set*(..)) {System.out.println("Write");}
  before(): execution(* Point.get*(..)) {System.out.println("Read");}
  before(): execution((Point || Line).new(..)) {System.out.println("Create");}
}
```

Fig. 1. Tracking access in AspectJ.

3.2. Aspect-Oriented Programming and Related Approaches

According to one of those who stood upon the birth of the aspect-oriented programming, Gregor Kiczales, aspect-oriented programming (AOP) is a new programming paradigm that enables the modularization of crosscutting concerns [KLM+97]. We'll take a closer look at four main AOP approaches.

Xerox PARC Aspect-Oriented Programming

Most of the AOP terminology (as well as its name) later adopted by others was coined by Xerox PARC AOP group. Their research effort is concentration mainly on AspectJ [Xera], a general purpose AOP extension to Java [LK98].

AOP appeared as a reaction to the problem known from the *generalized procedure languages* [KLM+97], i.e. languages that use the concept of procedure to capture functionality (besides procedural languages, this includes functional and object-oriented languages as well). In such languages some program code fragments that implement a clearly separable *aspect* of a system (such as synchronization) are scattered and repeated throughout the program code that becomes *tangled*. AOP aims at factoring out such aspects into separate units. Aspects *crosscut* the *base* code in *join points*. These must be specified so aspects could be *woven* into the base code by a *weaver*.

A simple example written in AspectJ v1.0.0 (similar to the example in [LK98]) in Fig. 1 illustrates the idea. Two classes are presented there, `Point` and `Line`, whose methods are of three kinds: creating, writing and reading (implementation of the methods is omitted). Suppose we want to be informed what kind of access to these classes has been performed. In ordinary Java we would have to modify each method of both `Point` and `Line`. Moreover, this would result in a tangled code. In AspectJ both problems can be avoided using aspects. In our example it is the aspect `ShowAccesses` that solves the problem. Note that the original code remains unchanged.

The solution with aspects is undoubtedly more elegant than the tangled one would be. However, the information where an aspect is to be woven (i.e., join points) is included in the aspect itself. This complicates the aspect reuse. AspectJ addresses this problem with abstract aspects and named sets of join points, so-called *pointcuts*.

Adaptive Programming

Adaptive programming (AP), proposed by Demeter group [Dem] at Northeastern University in Boston, deals mainly with the traversal strategies of class diagrams. Demeter group has used the ideas of AOP several years before the name aspect-oriented programming was coined. After the collaboration with the Xerox PARC AOP group had begun, Demeter group redefined AP as "the special case of AOP where some of the building blocks are expressible in terms of graphs and where the other building blocks refer to the graphs using traversal strategies" (building block stands for aspect or component) [Lie].

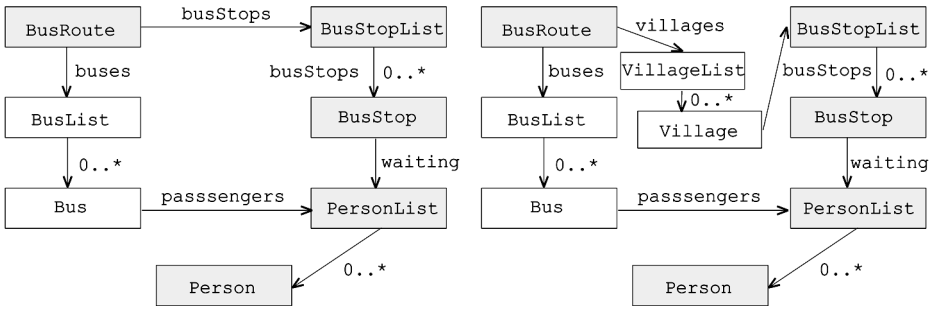


Fig. 2. Traversal strategies (from [Lie97], ©1997 Northeastern University).

The traversal strategies are partial graph specifications through mentioning a few isolated cornerstone nodes and edges, and thus they crosscut the graphs they are intended for.

An example of AP is presented in Fig. 2. The left part of the figure presents a UML class diagram of a system. Assume we would like to count the people waiting at the bus stations along the bus route. In ordinary OOP this would require either the implementation of small methods in all of the affected classes (depicted shaded) or rough breaking of the encapsulation by exposing some of the classes' private data.

If we use a traversal strategy, as it is proposed in AP, there is no need for a change in the existing classes. In this case, the traversal strategy:

from BusRoute through BusStop to Person

solves the problem of getting to objects of the class `Person` along the bus route, which is sufficient to count them. The right part of Fig. 2 demonstrates the robustness of this technique: the traversal strategy mentioned above applies in this case as well although the class diagram it was constructed for has changed.

Composition Filters

Composition filters (CF) is an aspect-oriented programming approach in which different aspects are expressed as declarative and orthogonal message transformation specifications called *filters* [AT98].

A message sent to an object is evaluated and manipulated by the filters of that object, which are defined in an ordered set, until it is discarded or dispatched (i.e., activated or delegated to another object). A filter behavior is simple: each filter can either accept or reject the received message, but the semantics of the operations depends on the filter type. For example, if an `Error` filter accepts the received message, it is forwarded to the next filter, but if it was a `Dispatch` filter, the message would be executed. A detailed description of CF can be found in [AWB+93, Koo95].

In Fig. 3 two sets of filters (written in Sina language [Koo95], which directly adopts the CF model [AT98, AWB+93]), are shown. These

```
Point
  acc: ShowAccess;
  inputfilters
    WriteAccess: Dispatch = {set, acc.WriteAccess, inner.*};
    ReadAccess: Dispatch = {getX, getY, acc.ReadAccess, inner.*};
    CreateAccess: Dispatch = {Point, acc.CreateAccess, inner.*};
    Execute: Dispatch = {true => inner.*};

Line
  acc: ShowAccess;
  inputfilters
    WriteAccess: Dispatch = {set, acc.WriteAccess, inner.*};
    ReadAccess: Dispatch = {getX, getY, getX1, getY1, acc.ReadAccess, inner.*};
    CreateAccess: Dispatch = {Line, acc.CreateAccess, inner.*};
    Execute: Dispatch = {true => inner.*};
```

Fig. 3. Tracking access example implemented using composition filters approach.

filters are attached to the `Point` and `Line` classes from Fig. 1. The existence of the class `ShowAccess` is presumed. `ShowAccess` provides three methods — `WriteAccess`, `ReadAccess` and `CreateAccess`) — that simply write out the type of the access. They are called by the three corresponding `Dispatch` filters, in case the message was accepted. Afterwards, the method of the inner object, which has actually been called, is executed (`inner.*`).

From the perspective of AOP, the class `ShowAccess` implements the aspect, while the filters represent the join points. Thus, the join points in this case are separated from the aspect, which is better regarding the aspect reuse.

Subject-Oriented Programming

A concept can be defined by naming its properties. This is sufficient to precisely define and identify mathematical concepts, but the same does not apply to natural concepts. Their definitions are *subjective* and thus never complete (more details about conceptual modeling can be found in [CE00]).

Subject-oriented programming (SOP), developed at IBM as an extension to OOP [IBM], is based on subjective views, so-called *subjects*. A subject is a collection of classes or class fragments whose hierarchy models its domain in its own, subjective way. A complete software system is then composed out of subjects by writing the *composition rules*, which specify correspondence of the subjects (i.e., namespaces), classes and members to be composed and how to combine them.

As a result of the research effort in SOP, the Watson Subject Compiler was developed [KOHK96], which allows partial (subjective) definitions of C++ program elements and automates the composition required to produce a running program. There are also other platforms SOP support was built for, such as IBM VisualAge for C++ Version 4, HyperJ and Smalltalk.

The example from Fig. 1 reimplemented in Watson Subject Compiler-like syntax (the actual syntax could by slightly different) is presented in Fig. 4. We assume that the class `ShowAccess` is implemented in `Access` namespace and that the classes `Point` and `Line` are implemented in the `Graphics` namespace. The join-points, represented by composition rules, are separated from the aspect and represented by a separate class (as in CF approach). The composition rules for the methods `getY`, `getX1`, `getY1` and `getX2` are omitted in Fig. 4 (indicated by ellipsis) since they are analogous to the rules for `getX` or `getY2`.

This is not a characteristic case of the application of SOP (such can be found in [OHBS94, KOHK96, IBM]); it is presented here in order to show how a well-known AOP example can be easily transformed into its SOP version. Nevertheless, there is no general agreement whether SOP is AOP. In [CE00] SOP is viewed as a special case of AOP where the aspects according to which the system is being decomposed are chosen in such a manner that they represent different, subjective views of the system. On the other hand, Kiczales et al. reject the very idea that SOP (which they call *subjective programming*) could be AOP, arguing that the methods from different subjects, which are being automatically composed in SOP, are components in the AOP sense, since they can be well localized in a generalized procedure (routine) [KLM+97].

```
namespace GraphicsWithAccess{
  class Point;
  class Line;}

GraphicsWithAccess.Point.Point := Merge[Graphics.Point.Point, Access.ShowAccess.CreateAccess];
GraphicsWithAccess.Line.Line := Merge[Graphics.Point.Line, Access.ShowAccess.CreateAccess];

GraphicsWithAccess.Point.set := Merge[Graphics.Point.set, Access.ShowAccess.WriteAccess];
GraphicsWithAccess.Line.set :=  Merge[Graphics.Line.set, Access.ShowAccess.WriteAccess];

GraphicsWithAccess.Point.getX := Merge[Graphics.Point.getX, Access.ShowAccess.ReadAccess];
      . . .
GraphicsWithAccess.Line.getY2 := Merge[Graphics.Line.getY2, Access.ShowAccess.ReadAccess];
```

Fig. 4. Tracking access example implemented using subject-oriented approach.

But this seems to be a more general issue, since it applies to AspectJ, too, where it has been identified as *aspectual paradox* by Liebrherr et al. [LLM99]:

> An aspect described in AspectJ, the Xerox PARC's AOP language, which has a construct for specifying aspects, is by definition no longer an aspect, as it has just been captured in a (new kind of) generalized routine.

As observed in [Cza98], SOP is close to GenVoca [BG97], a successful approach to software reuse. In GenVoca, systems are composed out of *layers* according to *design rules*: GenVoca layers can be easily simulated by subjects.

3.3. Generative Programming

Krysztof Czarnecki and Ulrich Eisenecker propose a comprehensive software development paradigm which brings together the object-oriented analysis and design methods with domain engineering methods that enable development of the families of systems: generative programming [CE00]:

> Generative programming (GP) is a software engineering paradigm based on modeling software systems families such that, given a particular requirements specification, a highly customized and optimized intermediate or end-product can be automatically manufactured on demand from elementary, reusable implementation components by means of configuration knowledge.

GP is a unifying paradigm — it is closely related to four other paradigms (see Figure 5):

- object-oriented programming, providing effective system modeling techniques,
- generic programming, enabling reuse through parameterization,
- domain-specific languages, increasing the abstraction level for a particular domain, and
- aspect-oriented programming, used to achieve the separation of concerns.

In order to be used, GP first has to be tailored to a particular domain. This process will yield a methodology for the families of systems to be developed, which can be viewed as a paradigm in its own right. This gives a certain meta-paradigm flavor to GP.

In the solution domain, GP requires metaprogramming for weaving and automatic configuration. To support domain-specific notations, syntactic extensions are needed. *Active libraries*

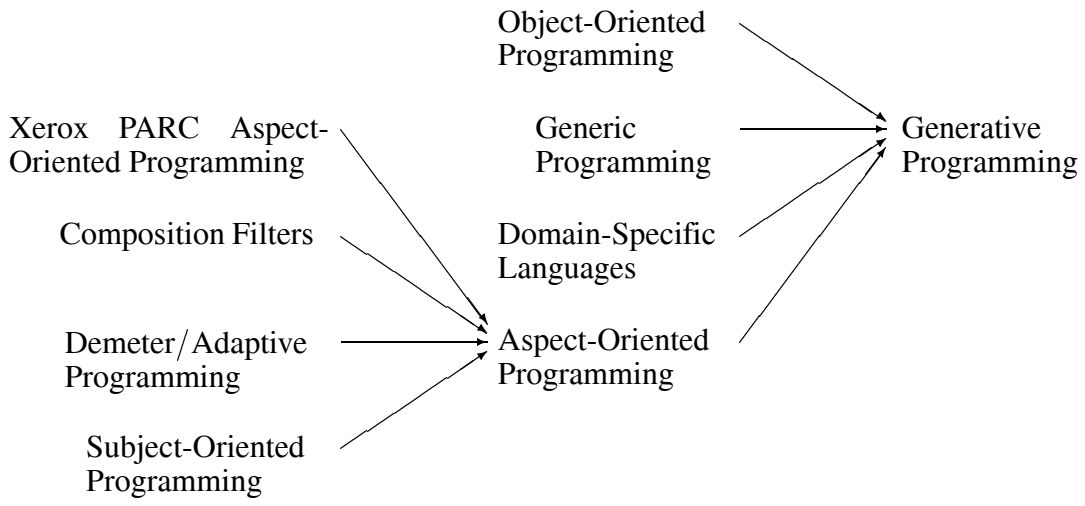


Fig. 5. Generative programming and related paradigms. The arrows represent "is incorporated into" relationship.

are proposed in [CE00], which can be viewed as *knowledgeable agents* interacting with each other to produce concrete components, as appropriate to cover this requirement.

4. Multi-Paradigm Approaches

In the survey of recent post-object-oriented software development paradigms presented in the previous section a spontaneous move towards the integration of paradigms became apparent. This section proceeds with explicit multi-paradigm approaches.

4.1. Multi-Paradigm Programming in Leda

The question how to support multi-paradigm programming at the language level can be answered simply: create a multi-paradigm language. Timothy Budd took this route by creating a multi-paradigm programming language called Leda.

According to Budd, Leda supports four programming paradigms [Bud95]: imperative (procedural, to be more precise) logic, functional, and object-oriented. The term paradigm, as used by Budd, denotes a large-scale paradigm (with respect to the classification of paradigms introduced in Section 2). This means that Leda actually supports more than four small-scale paradigms. This is clear having in mind that, for example, object-oriented paradigm breaks down into six or more small-scale paradigms, as has been shown in Section 4.2. Nevertheless, in order not to digress from the intent of this approach, just the mechanisms by which each of the four proclaimed paradigms is supported in the language will be discussed.

Leda has a Pascal-like (i.e., Algol-like) syntax and, moreover, in Leda the mechanism upon which all the four supported paradigms realization is based on are functions (procedures that return values). This makes a good background for *procedural* paradigm, denoted by Budd as *imperative*.

Logic paradigm is supported by a special type of function that returns *relation* data type and by a special assignment operator `<-`. These indicate when the inference mechanism, inherent to logic programming, is to be activated.

Functional paradigm requires no special mechanism other than those provided by functions, because Leda permits a function to be an argument of another function and to return a function. Thus, functional paradigm is achieved by using functions in the recursive fashion while refraining from assignments.

In addition to the basic mechanisms of *object-oriented paradigm*, such as classes, inheritance, encapsulation etc., Leda also supports *parameterized types* (considered by some authors a part of object-oriented paradigm [Mey97]).

Despite Leda is not widely used, it is worth consideration because it demonstrates the combination of paradigms. For example, the inference mechanism of logic programming can be used inside of a procedure.

Of course, creating a language that supports multiple paradigms and expecting it to be the best language for programming is similar to a search for the best programming paradigm. No matter how many paradigms are supported in a programming language, the number is finite and, obviously, it does not embrace future paradigms. One can argue that it is possible to extend the language with new programming mechanisms in order to support new paradigms. This is not only possible, but often practiced. Unfortunately, due to limitations set by parsing methods, programming languages cannot be extended indefinitely.

Leda is an example of a language *created* (from scratch) in order to support multiple paradigms. We can study existing interconnecting languages that support different paradigms through an interface instead of making a completely new language (a sort of language reuse). There is also a possibility of implementing one language on top of the other, but this leads to a certain degradation of performance. An example of interconnecting object-oriented and logic programming (Loops and Xerox Quintus Prolog) can be found in [KE88].

There are lots of approaches that fall into this category. Yet another approach and an overview of similar approaches, together with the discussion of the problems of paradigms integration, can be found in [VS95]. Such approaches are popular especially in the field of artificial intelligence because of the need to combine the

two paradigms traditionally used in this field, logic and functional programming, both with each other and together with OOP.

Different paradigms are expressed using different syntax. BETA language [Mad00] is supposed to overcome this inconvenience through a *unified syntax* achieved by introducing the so-called *pattern* as an abstraction of all other programming language constructs appearing in the paradigms it supports. The approach is therefore denoted as *unified paradigm*, but it is not fundamentally different from other "created to be multi-paradigm" languages.

4.2. Multi-Paradigm Design for C++

Multi-paradigm design for C++ (MPD), as proposed by Coplien [Cop99b, Cop00], has its roots in the multi-paradigm features of C++. Despite these multi-paradigm features, C++ is often considered to be just an object-oriented language. As such, C++ is used to implement the systems designed according to object-oriented paradigm. However, non-object-oriented features of C++ are widely used, but without a support in the (object-oriented) design.

MPD is a metaparadigm intended for developing families of systems, therefore akin to domain engineering approaches. It deals with choosing the appropriate paradigm for a feature being designed and implemented. MPD is based on SCV analysis (discussed in Section 2.2) or, to be more precise, SCVR analysis, where R stands for the relationship between the domains [Cop00], which are covered by variability dependency graphs (explained further in this section). On the other hand, neither SCV, nor SCVR analysis is mentioned in [Cop99b]; the term *commonality and variability analysis* is used instead to denote the same thing. Commonality analysis concentrates on common attributes while the aim of variability analysis is to parameterize the variation.

The major steps in MPD are: commonality and variability analysis of the application domain, commonality and variability analysis of the solution domain, transformational analysis and translation from the transformational analysis to the code. These steps need not to be performed sequentially. They can be performed in parallel (to some extent) and revisited as needed. Before starting the actual MPD, it is recommended to evaluate the possibility to reuse an existing (similar) design. If the commonalities and variabilities of the application domain do not fit any existing solution domain structures, creation of a new application-oriented (i.e., domain-specific) language should be considered.

Application domain analysis. Commonality analysis of the application domain (usually denoted as problem domain) starts with finding commonality domains and creating domain dictionary. It then proceeds in parallel with variability analysis, whose results — the *parameters of variation* of a given commonality domain and their characteristics — are being summarized in *variability tables* (one per each commonality domain), as depicted in the upper part of Fig. 6.

As already mentioned, *variability dependency graphs* (denoted also as *domain dependency* graphs or diagrams) are used to capture the relationship between domains and their parameters of variation, which are also domains. Variability dependency graphs are directed graphs whose nodes represent domains and the edges represent "depends on" relationship (in the direction indicated by an arrow) between the domains and their parameters of variation. Despite the simple notation, variability dependency graphs are quite useful in identifying *overlapping* domains (such domains can be merged) and *codependent* domains, i.e. the domains with circular dependencies (which must be resolved).

Solution domain analysis. The same commonality and variability analysis as applied to the application domain is applied to the solution domain, i.e. the programming language. First, a description with an example is provided of the identified small-scale paradigms, manifested through the language features, structured according to commonality, variability and binding. The analysis proceeds with exploring the *negative variability*, a variability that violates the rule of variation by attacking the underlying commonality. A *positive variability*, as opposed to the negative one, can be parameterized. The negative variability has to be kept small. If it becomes larger than the commonality, the design

Variability tables (from application domain SCVR analysis)

Text Editor Variability Analysis for Commonality domain:
TEXT EDITING BUFFERS (*Commonality: Behavior and Structure*)

Parameters of variation	Meaning	Domain	Binding	Default
Output medium *Structure, Algorithm*	...	Database, RCS file, TTY, UNIX file	Run time	UNIX file

Family table (from solution domain SCVR analysis)

Commonality	Variability	Binding	Instantiation	Language Mechanism
	. . .			
Related operations and some structure (positive variability)	. . .			
	Algorithm (especially multiple), as well as (optional) data structure and state	Compile time	Optional	Inheritance
	Algorithm, as well as (optional) data structure and state	Run time	Optional	Virtual functions

Fig. 6. Transformational analysis in MPD (according to an example from [Cop99b]).

should be refactored to reverse the commonality and variability.

The results of the solution domain commonality and variability analysis are summarized in the *family table*, as shown in the lower part of Fig. 6, and in the table used to express *features for negative variability*, where for each combination of the kind of commonality and the kind of variability the language feature for positive variability and the one for the corresponding negative variability are introduced.

Transformational analysis. The tables obtained in the preceding analyses are used in *transformational analysis*, which is, roughly speaking, a matching of application domain structures described in variability tables, with solution domain structures, i.e. paradigms, described in family tables. Figure 6 shows how this matching is performed. Prior to the matching, the commonality domain has to be generalized (e.g., TEXT EDITING BUFFERS: *behavior, structure*), as well as the parameters of variation (e.g., output medium: *structure, algorithm*).

MPD emphasizes the solution domain analysis whose underestimation in contemporary software development methodologies results in a gap between design and implementation.

To a certain extent, MPD enforces the *reuse of design*: both application and solution domain analysis can be reused independently; however, transformational analysis is not reusable. This brings MPD close to *design patterns*, as discussed in [Cop99b]. On the very cover of the design patterns cornerstone book [GHJV95] Steve Vinoski points out that a reusable design is "the real key to software reuse". This claim is being justified in the ongoing research on reuse with design patterns [SN00].

Indeed, MPD and design patterns seem to be complementary; design patterns capture designers' experience by documenting the recommended solutions for common problems in software development, while MPD relies on this experience. However, to make a full use of design patterns in MPD, and in software development in general, a better way of their representation is needed [SNB98].

Although the design patterns from [GHJV95] are inspired by Alexandrian patterns [Ale79], not all of them are the patterns in the Alexandrian sense: some of them can be formalized as configurations of commonality and variability (unlike Alexandrian patterns). As such they can be incorporated directly into MPD (by adding them to the family table), as anything else that can be formalized as a configuration of commonality and variability (i.e., other paradigms and solution domain tools that are not supported by the main programming language, like databases or parser generators) [Cop99b].

One of the problems with MPD is the unsuitability of the notation used: only a few types of tables and variability diagrams with a lot of relevant details expressed as informal text. With a better notation, like feature modeling [Vra01], transformational analysis could become more transparent. A better notation could also ease the transition to the actual program code (the program skeleton).

4.3. Intentional Programming

Programming languages with fixed syntax are limiting otherwise unlimited number of programming abstractions. Intentional programming group at Microsoft Research offered a solution to this problem as a new software development paradigm called *intentional programming* (IP) [Sim99, Sim96] (the project is on hold from Spring 2001 [Roe]). The idea behind IP is that programming abstractions, which are in IP denoted as *intentions*, could live better without their hosts, (fixed-syntax) programming languages, because of their limits in the accepted notations (due to underlying grammars).

A program in IP is represented by a so-called *intentional tree*, whose nodes represent intention instances. Each intention instance points to the corresponding intention declaration node providing a method which specifies the process of transforming the subtree headed by the intention instance. The executable program is obtained in a process called *reduction* in which the intentional tree is traversed and transformed according to the rules indicated by intention declarations until it consists only of executable nodes. Such a *reduced* tree is represented in an intermediate language. The executable code is generated from this representation.

It would be inconvenient for a human to directly maintain the intentional tree. This is being performed in a programming environment with a special graphic editor instead of the usual text editor. It enables each intention to have its own graphic representation. Of course, entering a program in such an environment is quite different from entering it in a classic text editor. A program text, as we are used to it, is a complete and an unambiguous representation of the program. In IP environment this is not so. What is presented in IP editor is only a view of the actual program. To illustrate this, consider one peculiarity: two distinct variables can have the same name (even if they reside the same scope). This is possible because the intentional tree does not rely on the names to identify intentions; the names are provided only for developers' convenience.

Although it can seem so, IP is not intended to push out the existing programming languages from the scene. It can import any program in any programming language if a parser for that language — in the form of a library — is added to IP environment.

4.4. Multi-Paradigm Approaches Compared

The three multi-paradigm approaches presented in this section are compared in Table 1 according to the selected criteria: the concept of *paradigm* the approach enforces, a programming *language* the approach is bound to, and whether the approach supports the *language extension*.

It is important to note that these three approaches are not antagonistic; they are complementary. Multi-paradigm design arms us with techniques for dealing with multiple paradigms when a multi-paradigm environment is available. Intentional programming enables such an environment to be created and maintained easier than it is the case with classical programming languages. Finally, multi-paradigm programming in Leda demonstrates how four specific programming paradigms can be combined.

	Paradigm	Language	Language extension
MP in Leda	large-scale	Leda	no
MPD	small-scale	any	not applicable
IP	small-scale	none	yes

Table 1. The three multi-paradigm approaches compared.

5. Conclusions and Further Work

The concept of paradigm in computer science in the context of software development has been analyzed in this article. Two distinct meanings of paradigm in software development have been identified and discussed: large-scale and small-scale.

A survey of selected post-object-oriented paradigms, namely aspect-oriented approaches and generative programming, has been presented. A growing multi-paradigm tendency has been identified in these approaches. This tendency has materialized into explicit multi-paradigm approaches. Three such approaches have been discussed and compared: multi-paradigm programming in Leda, multi-paradigm design for C++, and intentional programming.

Multi-paradigm approach to software development makes the question which paradigm is the best (and therefore should replace all other paradigms) a meaningless one. It has a potential of incorporating all the paradigms at disposal of the solution domain. It is a paradigm of paradigms: a metaparadigm.

However, multi-paradigm software development must be further improved and refined if it is to be used in its full strength. Among the multi-paradigm approaches considered, multi-paradigm design (for C++), described in Section 4.2, seems to be the most appropriate as the basis for the future form of multi-paradigm software development.

Multi-paradigm design can be tailored to any programming language by applying commonality and variability analysis to it. It would be particularly interesting to establish multi-paradigm design for AspectJ (see Section 3.2) since it could help to understand better the relationship between multi-paradigm design and aspect-oriented programming (although, of course, AspectJ is not the same as aspect-oriented programming in general), which Coplien denoted as "the most fully general implementation of multi-paradigm design possible" [Cop00]. An initial work towards establishing multi-paradigm design for AspectJ has been reported in [Vra01].

The notation used in multi-paradigm design, which besides informal description embraces only two types of tables and a kind of simple graphs, is not appropriate. This is apparent especially during transformational analysis. Similarly to commonality and variability analysis of multi-paradigm design, *feature modeling* also expresses commonalities and variabilities explicitly, but using a more sophisticated notation (see [CE00] for more details on feature modeling and feature diagrams). Both solution and application domains can be represented as feature models, as has been demonstrated in [Vra01]. This eases transformational analysis and brings multi-paradigm design and generative programming closer to each other.

Acknowledgments

This work was partially supported by Slovak Science Grant Agency, grant No. G1/7611/20. I would like to thank Pavol Návrat for his valuable suggestions.

References

[Ale79] C. ALEXANDER. *The Timeless Way of Building*. Oxford University Press, 1979. Cited in [Cop99b].

[AT98] M. AKSIT AND B. TEKINERDOGAN. Solving the modeling problems of object-oriented languages by composing multiple aspects using composition filters. In *Proc. of the Aspect-Oriented Programming Workshop at ECOOP'98*, 1998. Available at [Twe].

[AWB+93] M. AKSIT, K. WAKITA, J. BOSCH, L. BERGMANS, AND A. YONEZAWA. Abstracting object-interactions using composition-filters. In

Proc. of 7thEuropean Conference on Object-Oriented Programming (ECOPP'93) Workshop, LNCS 791, pages 152–184, Kaiserslautern, Germany, 1993. Springer. Available at [Twe].

[BG97] D. BATORY AND B. J. GERACI. Composition validation and subjectivity in GenVoca generators. *IEEE Transactions on Software Engineering (special issue on Software Reuse)*, pages 67–82, February 1997. Available at [Pro].

[Boo94] G. BOOCH. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley Publishing Company, second edition, 1994.

[Bud95] T. A. BUDD. *Multiparadigm Programming in Leda*. Addison-Wesley, 1995.

[CE00] K. CZARNECKI AND U. EISENECKER. *Generative Programing: Principles, Techniques, and Tools*. Addison-Wesley, 2000.

[CHW98] J. COPLIEN, D. HOFFMAN, AND D. WEISS. Commonality and variability in software engineering. *IEEE Software*, 15(6), November 1998. Available at [Cop].

[Cop] J. O. COPLIEN. Home page. `http://www.bell-labs.com/people/cope`. Accessed on November 15, 2001.

[Cop99a] J. O. COPLIEN. Multi-paradigm design and implementation in C++. Slides and notes of the tutorial given at *1st International Conference on Generative and Component-Based Software Engineering (GCSE'99)*, Erfurt, Germany, September 1999. Available at [Cop].

[Cop99b] J. O. COPLIEN. *Multi-Paradigm Design for C++*. Addison-Wesley, 1999.

[Cop00] J. O. COPLIEN. *Multi-Paradigm Design*. PhD thesis, Vrije Universiteit Brussel, Belgium, 2000. Available at [Cop].

[Cza] K. CZARNECKI. Home page. `http://www.prakinf.tu−ilmenau.de/∼czarn`. Accessed on November 15, 2001.

[Cza98] K. CZARNECKI. *Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models*. PhD thesis, Technical University of Ilmenau, Germany, 1998. Partially available at [Cza].

[Dem] Demeter group. Home page. `http://www.ccs.neu.edu/research/demeter`. Accessed on October 30, 2001.

[GHJV95] E. GAMMA, R. HELM, R. JOHNSON, AND J. VLISSIDES. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[IBM] IBM Research. Subject-Oriented Programming home page. `http://www.research.ibm.com/sop`. Accessed on August 15, 2000.

[KE88] T. KOSCHMANN AND M. W. EVENS. Bridging the gap between object-oriented and logic programming. *IEEE Software*, 60:36–42, July 1988.

[KLM+97] G. KICZALES, J. LAMPING, A. MENDHEKAR, C. MAEDA, C. V. LOPES, J.-M. LOINGTIER, AND J. IRWIN. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *Proc. of 11th European Conference on Object-Oriented Programming (ECOOP'97)*, LNCS 1241, Jyväskylä, Finland, June 1997. Springer. Available at [Xerb].

[KOHK96] M. KAPLAN, H. OSSHER, W. HARRISON, AND V. KRUSKAL. Subject-oriented design and the watson subject compiler. In *11th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'96)*, 1996. Available at [IBM].

[Koo95] P. S. KOOPMANS. On the definition and implementation of the Sina/st language. Master's thesis, Dept. of Computer Science, University of Twente, The Netherlands, August 1995. Available at [Twe].

[Kuh70] T. S. KUHN. *The Structure of Scientific Revolutions*. University of Chicago Press, Chicago, 1970. Czech translation, OIKYMENH, 1997.

[Lie] K. J. LIEBERHERR. Connections between Demeter/adaptive programming and aspect-oriented programming. Web document, College of Computer Science, Northeastern University, Boston, USA. Available at [Dem].

[Lie97] K. J. LIEBERHERR. Demeter and aspect-oriented programming: Why are programs hard to evolve? Presentation slides, *3rd Conference Smalltalk und Java in Industrie und Ausbildung (STJA 97)*, Erfurt, Germany, 1997. Available at [Dem].

[LK98] C. V. LOPES AND G. KICZALES. Recent developments in AspectJ. In *Proc. of 12th European Conference on Object-Oriented Programming (ECOPP'98) Workshops, Demos, and Posters*, LNCS 1543, Brussels, Belgium, July 1998. Springer. Available at [Xerb].

[LLM99] K. J. LIEBERHERR, D. LORENZ, AND M. MEZINI. Programming with aspectual components. Technical Report NU-CCS-99-01, College of Computer Science, Northeastern University, Boston, MA, March 1999. Available at [Dem].

[Mad00] O. L. MADSEN. Towards a unified programming language. In J. L. Knudsen, editor, *Proc. of 14th European Conference on Object-Oriented Programming(ECOOP 2000)*, Sophia Antipolis and Cannes, France, June 2000. Springer LNCS 1850.

[Mer] Merriam-Webster OnLine. Merriam-Webster's Collegiate Dictionary. `http://www.m-w.com`. Accessed on November 15, 2001.

[Mey97] B. MEYER. *Object-Oriented Analysis Software Construction*. Prentice Hall, second edition, 1997.

[NEC] NEC Research Institute. ResearchIndex: The NECI Scientific Digital Research Library. `http://citeseer.nj.nec.com`. Accessed on November 15, 2001.

[Náv96] P. NÁVRAT. A closer look at programming expertise: Critical survey of some methodological issues. *Information and Software Technology*, 38(1):37–46, 1996.

[OHBS94] H. OSSHER, W. HARRISON, F. BUDINSKY, AND I. SIMMONDS. Subject-oriented programming: Supporting decentralized development of objects. In *Proc. of 7th IBM Conference on Object-Oriented Technology*, July 1994. Available at [IBM].

[Pro] Product-Line Architecture Research group. Home page. `http://www.cs.utexas.edu/users/schwartz`. Accessed on November 15, 2001.

[Roe] L. ROEDER. Home page. `http://www.aisto.com/roeder`. Accessed on November 21, 2001.

[Sim96] C. SIMONYI. Intentional programming — innovation in the legacy age, June 1996. Presented at IFIP WG 2.1 meeting, available at [Roe].

[Sim99] C. SIMONYI. The future is intentional. *IEEE Computer*, 32(5):56–57, May 1999.

[SN97] M. SMOLÁROVÁ AND P. NÁVRAT. Software reuse: Principles, patterns, prospects. *Journal of Computing and Information Technology*, 5(1):33–48, 1997.

[SN00] M. SMOLÁROVÁ AND P. NÁVRAT. Reuse with design patterns: Towards pattern-based design. In Y. Feng, D. Notkin, and M. Gaudel, editors, *Proc. Software: Theory and Practice*, pages 232–235, Beijing, China, 2000. PHEI - Publishing House of Electronics Industry.

[SNB98] M. SMOLÁROVÁ, P. NÁVRAT, AND M. BELIKOVÁ. Abstracting and generalising with design patterns. In A. G. U. Güdükbay, T Dayar and E. Gelenbe, editors, *Proc. of 13th International Symposium on Computer and Information Sciences (ISCIS'98)*, pages 551–558, Belek-Antalya, Turkey, 1998. IOS Press.

[Twe] Twente Research and Education on Software Engineering (TRESE) group. Home page. `http://trese.cs.utwente.nl`. Accessed on November 15, 2001.

[Vra00] V. VRANIĆ. Multiple software development paradigms and multi-paradigm software development. In J. Zendulka, editor, *Proc. of 3rd International Conference on Information Systems Modelling (ISM 2000)*, pages 191–196, Rožnov pod Radhoštěm, Czech Republic, May 2000. MARQ.

[Vra01] 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, September 2001. Springer.

[VS95] S. VRANEŠ AND M. STANOJEVIĆ. Integrating multiple paradigms within the blackboard framework. *IEEE Transactions on Software Engineering*, 21(3):244–262, 1995.

[Xera] Xerox PARC. AspectJ home page. `http://aspectj.org`. Accessed on November 15, 2001.

[Xerb] Xerox PARC. Software Design Area home page. `http://www.parc.xerox.com/sda`. Accessed on November 15, 2001.

Received: February, 2001
Revised: November, 2001
Accepted: December, 2001

Contact address:
Valentino Vranić
Department of Computer Science and Engineering,
Faculty of Electrical Engineering and Information Technology
Slovak University of Technology in Bratislava, Slovakia
Ilkovičova 3
812 19 Bratislava
Slovakia
Phone: +421 (2) 602 91 548
Fax: +421 (2) 654 20 587
e-mail: `vranic@elf.stuba.sk`
WWW: `http://www.dcs.elf.stuba.sk/~vranic`

VALENTINO VRANIĆ received his Bc. (BSc.) in 1997, and his Ing. (MSc.) in 1999, both in information technology, and both from the Slovak University of Technology in Bratislava. Since 1999 he is a PhD student at the Department of Computer Science and Engineering, Faculty of Electrical Engineering and Information Technology of the Slovak University of Technology in Bratislava. His main research interests are multi-paradigm software development and aspect-oriented programming. He is a member of the Slovak Society for Computer Science.

Appendix B

AspectJ Paradigm Model: A Basis for Multi-Paradigm Design for AspectJ

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.

AspectJ Paradigm Model: A Basis for Multi-Paradigm Design for AspectJ*

Valentino Vranić

Department of Computer Science and Engineering
Faculty of Electrical Engineering and Information Technology
Slovak University of Technology, Ilkovičova 3, 812 19 Bratislava, Slovakia
vranic@elf.stuba.sk
http://www.dcs.elf.stuba.sk/~vranic

Abstract Multi-paradigm design is a metaparadigm: it enables to select the appropriate paradigm among those supported by a programming language for a feature being modeled in a process called transformational analysis. A paradigm model is a basis for multi-paradigm design. Feature modeling appears to be appropriate to represent a paradigm model. Such a model is proposed here for AspectJ language upon the confrontation of multi-paradigm design and feature modeling. Subsequently, the new transformational analysis is discussed.

1 Introduction

In this paper the AspectJ paradigm model, a basis for multi-paradigm design for AspectJ programming language (version 0.8), is proposed. AspectJ is an aspect-oriented extension to Java [6]. Multi-paradigm design for AspectJ is based on Coplien's multi-paradigm design [3] (originally applied to C++ and therefore known as multi-paradigm design *for C++*) to a different solution domain. It employs feature modeling [5] for the task Coplien's multi-paradigm design used scope, commonality, variability, and relationship analysis [4].

Scope, commonality, variability, and relationship analysis, which is basically a scope, commonality, and variability analysis [1] enhanced with the analysis of relationships between domains [4], is used to describe the paradigms (mechanisms of a programming language) provided by the solution domain (i.e., programming language), as commonality-variability pairings [4, 3]. This way of describing paradigms is compact, but not expressive enough to meet the requirements of the transformational analysis, a process of aligning problem domain structures with available paradigms.

Moreover, the paradigms are often connected, but multi-paradigm design provides no means to express how. The application of feature modeling instead of scope, commonality, variability, and relationship analysis could help solve the problems mentioned here, as will be shown in this paper.

* This work was partially supported by Slovak Science Grant Agency, grant No. G1/7611/20.

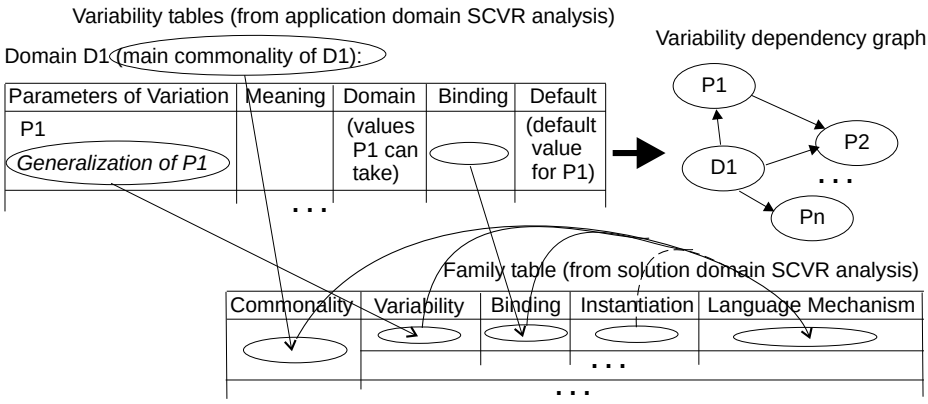


Figure 1. Transformational analysis in MPD.

Before presenting the actual AspectJ paradigm model, a critical survey of the issues regarding the multi-paradigm design for C++ (Sect. 2) and a basic information on feature modeling notation is provided (Sect. 3). Also, the relationship between feature modeling and techniques used in multi-paradigm design is analyzed (Sect. 4). AspectJ paradigm model is then presented (Sect. 5) and the impact of incorporating feature modeling into MPD on transformational analysis discussed (Sect. 6). Conclusions and further research directions close the paper (Sect. 7).

2 Multi-paradigm design for C++

Multi-paradigm design (MPD) for C++ [3] is based on the notion of small-scale paradigm [8], that can simplistically be perceived as a language mechanism (e.g., inheritance), as opposed to the (more common) notion of large-scale (programming) paradigm [2] (e.g., object-oriented programming; see [7] for a comparison of programming paradigms).

Figure 1 gives an overview of MPD. Scope, commonality, variability, and relationship (SCVR) analysis is performed on both domains, application and solution, with results summarized in variability (one for each domain) and family tables, respectively. The variability tables are incapable of capturing dependencies between the parameters of variation (that are also considered to be domains), so this is enabled by a simple graphical representation called *variability dependency graphs*. Each row of the family table represents a 4-tuple *(Commonality, Variability, Binding, Instantiation)* that determines the language mechanism.

The transformational analysis is actually a process of matching the application domain structures with the solution domain ones. First, the main commonality of the application domain, as identified by a developer, is matched with a commonality in the family table. This yields a set of rows in which the individual parameters of variation are resolved. Since parameters of variation (e.g., working

set management) are too specific to be matched with general variabilities (e.g., algorithm) in the family table, each parameter of variation must be generalized before matching. This seem as a too big step to make at once.

The generalization problem and the fact that the matching is performed between variability table *3*-tuples and family table *4*-tuples (variability table has no instantiation column), are eclipsed by another problem: some C++ language mechanisms are missing from the paradigm model proposed. For example, classes and methods (procedures) are not even mentioned. On the other hand, inheritance is embraced in the model. Maybe Coplien considered classes and methods too trivial to mention, but this has not been stated explicitly.

Moreover, C++ mechanisms listed in the family table and negative variability table[1] are inconsistent with those described in the text [3]. Yet another problem with the paradigm model in MPD is that it does not capture the dependencies between paradigms. This is an important information, since there are paradigms that make no sense without other paradigms (e.g., inheritance without classes in C++).

3 Feature Modeling

Feature modeling is a conceptual modeling technique used in domain engineering. The version of the feature modeling whose notation is described here comes from [5].

Feature diagrams are a key part of a feature model. A feature diagram is basically a directed tree with the edge decorations. The root represents a concept, and the rest of the nodes represents features. Edges connect a node with its features. There are two types of edges used to distinguish between *mandatory* features, ended by a filled circle, and *optional* features, ended by an empty circle. A concept instance *must* have all the mandatory features and *can* have the optional features.

The edge decorations are drawn as arcs connecting the subsets of the edges originating in the same node. They are used to define a partitioning of the subnodes of the node the edges originate from into *alternative* and *or-features*. A concept instance has exactly one feature from the set of alternative features. It can have any subset or all of the features from the set of or-features.

The nodes connected directly to the concept node are being denoted as its *direct features*; all other features are its *indirect features*, i.e. *subfeatures*. The indirect features can be included in the concept instance only if their parent node is included.

An example of a feature diagram with different types of features is presented in Fig. 2. Features f_1, f_2, f_3, and f_4 are direct features of the concept c, while other features are its indirect features. Features f_1 and f_2 are mandatory alternative features. Feature f_3 is an optional feature. Features f_5, f_6 and f_7 are mandatory or-features; they are also subfeatures of f_3.

[1] A table that summarizes language mechanisms corresponding to exceptions to variability.

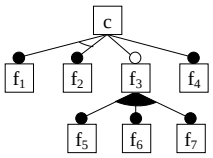


Figure 2. A feature diagram.

4 Applying Feature Modeling to Multi-Paradigm Design

Feature modeling is not unlike SCVR analysis. SCVR analysis, the heart of MPD, is based on the notions of commonality and variability (hence the name), and the notions of *common* and *variable* features is not unknown to feature modeling.

A common feature of a concept is a feature present in all concept instances, i.e. there must be a path of (pure) mandatory features leading from the concept to the feature. All other features are variable, i.e. any optional, alternative or or-feature is variable. The features to which variable features are attached are called *variation points*.

The scope in SCVR analysis, defined as a set of entities, is nothing but the concept in an *exemplar* representation.[2] The SCVR commonalities (assumptions held uniformly across the scope) and variabilities (assumptions true for only some elements in the scope) map straightforwardly to common and variable features of feature modeling, respectively.

The feature modeling enables to represent SCVR analysis commonalities and variabilities hierarchically and thus to express relationships among variabilities. For a solution domain SCVR analysis this means enabling to express how the paradigms it provides are related.

The most important results of SCVR analysis are provided in variability and family tables and variability dependency graphs.

4.1 Variability and Family Tables

Table 1 aligns the terms of feature modeling with its variability and family table counterparts (the columns). Only a fraction of the information provided usually by a feature model covers most of the needs of variability and family tables.

The parameters of variation are sometimes considered as subdomains (especially in variability dependency graphs). This is consistent with the feature modeling; the feature can be viewed as a concept.

Binding mode in feature modeling corresponds to binding time in MPD. The difference is that the set of binding times used in MPD is richer than the one used in feature modeling. This is due to a fact that the binding times in MPD are the actual binding times of a solution domain, like compile time, run

[2] The exemplar view of a concept is the one in which a concept is defined by a set of its instances [5].

Table 1. Feature modeling and MPD variability and family tables.

Feature modeling	**Variability tables**	**Family tables**
concept	commonality domain	language mechanism
common feature		commonality
variable feature		variability
variation point	parameter of variation	
alternative features	domain (of values)	
binding mode	binding	binding
semantic description, rationale	meaning	
default dependency rules	default (value)	
additional information		instantiation

time, etc. Feature modeling provides more abstract binding times, namely static, changeable, and dynamic binding. Each MPD binding time falls into one of these categories: source time and compile time bindings are static binding, link (load) time binding is a changeable binding, and run time binding is a dynamic binding.

The binding time applies only to variable features. It should be understood only as an auxiliary information to the transformational analysis. There is no notion of a unique binding time for a whole concept, as it is the case with a paradigm in MPD. Binding time should be indicated where it belongs—at variable features.

The feature modeling provides no counterpart for the family table column "instantiation", which indicates whether a language mechanism provides instantiation. This information should be provided as an attribute among the rest of the information associated with a feature model.

Possible values for instantiation in MPD are: *yes*, *no*, *not available* (*n/a*), and *optional*. It seems that *no* and *n/a* values are redundant: if a language mechanism does not provide instantiation, it can be only because the instantiation is not available for that mechanism. The *yes* value indicates that a mechanism is used only with instantiation, while *optional* means that it can be used both with instantiation and without it (a class doesn't have to be instantiated to make a use of the static fields and methods). Furthermore, the *optional* value does not make sense in the application domain—the instantiation is either needed or not.

4.2 Variability Dependency Graphs

In variability dependency graphs, the nodes represent domains and the directed edges represent the "depends on (a parameter of variation)" relationship; domain corresponds to a concept or feature (considered as a concept).

Parts of variability dependency diagrams can be derived from the feature diagrams. Commonality domain depends on its parameters of variation, or—in the feature modeling terminology—concept depends on its variation points. But, generally speaking, while the relationships between domains in variability

dependency graphs have a particular semantics, this cannot be said for the relationships in feature diagrams. Moreover, the feature diagrams are trees, not general graphs. All this suggests that variability dependency graphs should be kept as a separate notation. For each domain from the variability dependency graphs there should be a corresponding concept or feature in the feature model.

5 AspectJ Paradigms

AspectJ is an interesting programming language to explore in the sense of MPD because it supports two large-scale paradigms: object-oriented programing and aspect-oriented programming. However, large-scale view is not sufficient to make a full use of the programming language in the design. We must turn to a finer granularity and find out what small-scale paradigms, i.e. language mechanisms, AspectJ provides (referred to as *paradigms* in the following text). As was discussed in the previous sections, feature modeling will be employed to describe these paradigms.

Figure 3 shows a feature diagram of AspectJ. The paradigms in the feature diagram are indicated by a capitalization of the initial letter (e.g., Class). Binding time is indicated at variable features; if not, source time binding is assumed. Sometimes binding time of a feature depends on other features, as indicated in the diagram. In the text, the names of paradigms are typeset in the **boldface style**. The root of the feature diagram is AspectJ as a solution domain. It provides the paradigms that *can* be used, which is indicated by modeling the topmost paradigms as optional features.

The paradigm model establishes a paradigm hierarchy. Each paradigm is presented in a separate diagram as an alternative to the one big overall diagram. Wherever a root node of a paradigm tree is present, it is as if a whole tree was included there.

6 Transformational Analysis

Transformational analysis—aligning application domain structures with the solution domain ones—is the key part of MPD. The basic idea of how the transformational analysis is to be performed when these structures are represented by feature models is presented by the means of an example. Afterward, some general observations about the process of transformational analysis are given.

6.1 An Example: Text Editing Buffers

Text editing buffers[3] represent a state of a file being edited in a text editor. Text editing buffer caches the changes until user saves the text editing buffer into the file. Different text editing buffers employ different working set management

[3] The example discussed here is an adapted version of the text editing buffers example from [3].

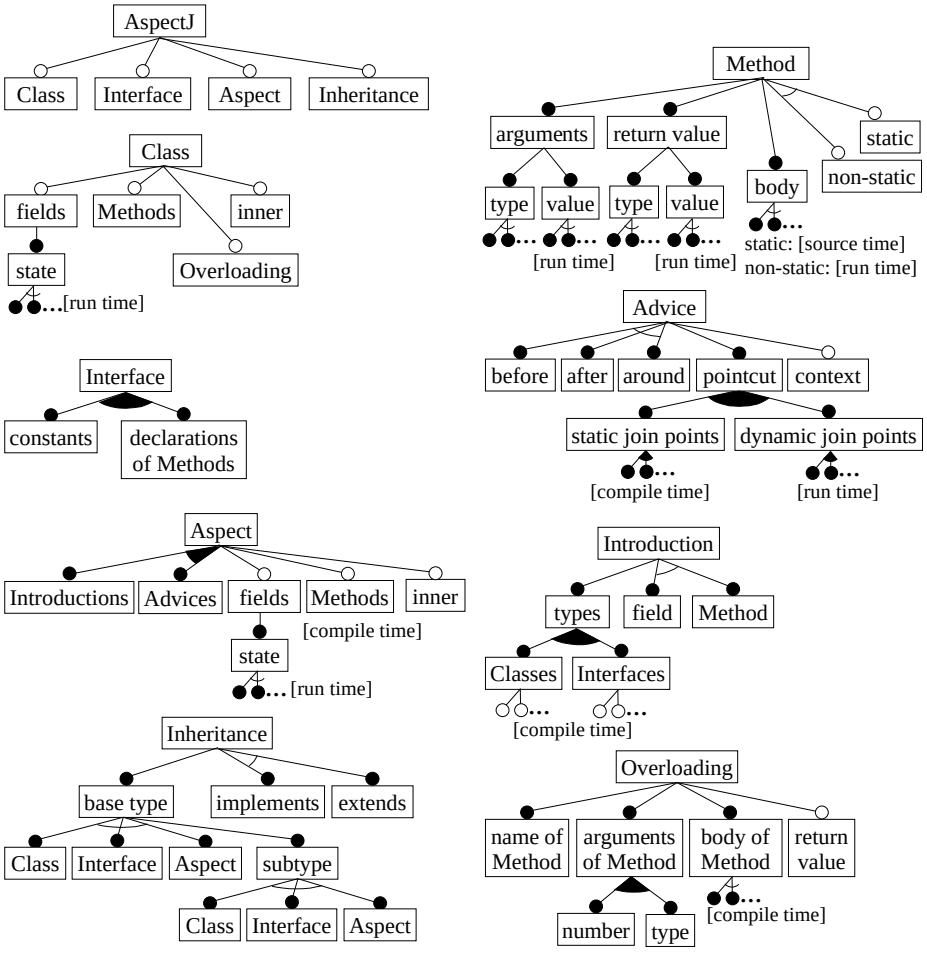


Figure 3. AspectJ paradigm model.

schemes and use different character sets. All text editing buffers load and save their contents into the file, maintain a record of the number of lines and characters, the cursor position, etc. The text editing buffer feature diagram is presented in Fig. 4. In the text, the feature names are distinguished by typesetting in the *italics style*. For simplicity, binding time and instantiation were not considered.

Now that feature models of both application and solution domains are available, we can proceed with the transformational analysis. We start with the unchangeable part of the application domain, i.e. the topmost common features. At this level a basic class or classes might be expected. The features *number of lines*, *number of characters*, and *cursor position* correspond to fields of the **class** paradigm. On the other hand, *yield data*, *replace data*, *load file*, and *save file*

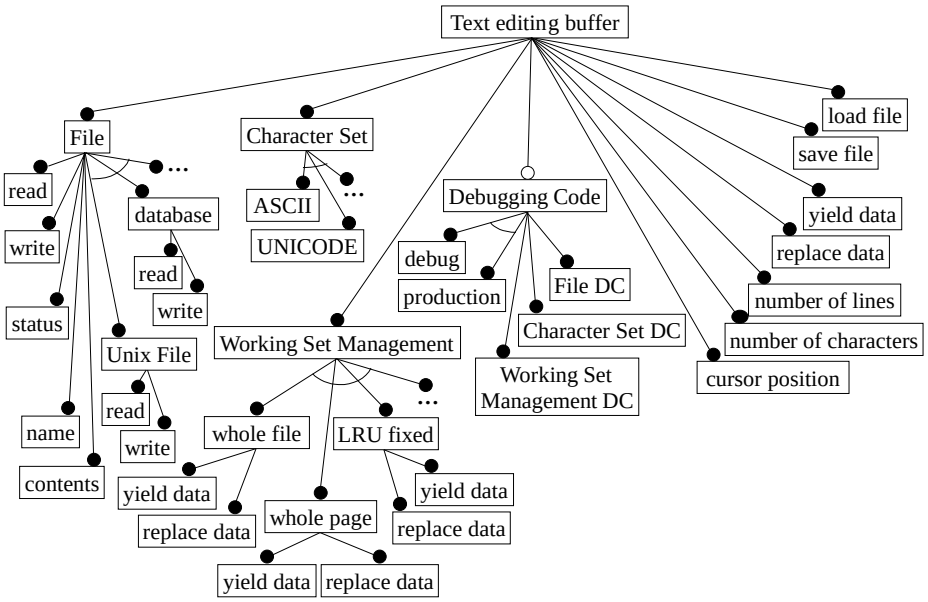


Figure 4. Text editing buffers feature diagram.

correspond to **method** paradigm. Accordingly, text editing buffer should be a class.

The rest of the topmost features are, apparently, variation points. The first one is *file*. All the files are read and written, but there are several file types and each one is read and written in a specific way. However, what is being read and written remains the same: *file name* and *contents*. We would probably expect to get the *status* of reading and writing. Thus we reached the leaves of the *file* subtree. If we compare these leaves to those of AspectJ feature diagram, they best match with arguments and return value. This brings us to **method** paradigm for *read* and *write* features.

We go up one level and discover that *database*, *RCS file*, *TTY*, and *Unix file* features match with the **class** paradigm. Accordingly, we expect that *file* would be a class too; so we match it with the **class** paradigm. The relationship between *file* and the file types matches with **inheritance**. Analogously, *character set* would be a class, and each type of it would be a subclass of that class.

The situation is similar with *working set management*: each type of *working set management* would be a class. But there is one difference: if we try to match it with **inheritance** further, we discover that we can match the whole *text editing buffer* with *base type* (because of *yield data*, *replace data*). So the *working set management* would be a primary differentiator.

Debugging code is somewhat special. It should be possible to turn it on and off easily (to obtain debug and production version, respectively). It is intended for *file*, *character set*, and *working set management* debugging; there is a special

debugging code for each of those. For example, we would like to know when the file is being read from and written to. We already matched *file* with **class** and reading and writing with **method**, so it seems we must look for such a paradigm that can influence the execution of methods. There is only one such paradigm: **advice**. As **advice** is available only in **aspect** paradigm, the *file debugging code*, *character set debugging code*, and *working set management debugging code* will be aspects. *File debugging code* will provide two **advice**s, one for reading and the other for writing a file, and *character set debugging code* only one, as only a name of character set being used has to be announced.

Things are slightly more complicated with *working set management debugging code*, as we are interested in the general operations of working set management, as well as in the specific operations of each type of it (not displayed in the feature diagram). This points us to **inheritance**: *working set management debugging code* matches with a base aspect, while each of its or-subfeatures matches with a sub-aspect.

6.2 Transformational Analysis Outline

The text editing buffers example disclosed some regularities in the process of transformational analysis. The matching was performed starting at leaves towards the root. Rarely the leaves were considered alone. Mostly, a feature was considered together with its first-level subfeatures. Multiple nodes from the application domain can match with a single solution domain node if its name is in plural. Matching of nodes is done according to the type of the nodes, e.g. the overall match of mandatory or-nodes is successful if a match has been found for one or more leaves.

The matching is interdependent. If two features depend on each other, then it matters what paradigm the first feature was matched with. In other words, matching a feature with a paradigm constrains the further design.

Up to now, nothing has been said about how the actual matching of two nodes is performed. This can be compared to the matching between the domain commonality and parameters of variation from the variability table to the commonalities and variabilities from the family table. Two nodes match if they conceptually represent the same thing; do they—it is up to the developer to decide. However, a conceptual gap is significantly smaller than in the original MPD where developer was forced to make such decisions at a very high level of abstraction.

7 Conclusions and Further Research

The table representation of the application and solution domains used in multi-paradigm design for C++ performs unsatisfactorily during the transformational analysis. Moreover, the C++ paradigm model is incomplete. The application of feature modeling instead of scope, commonality, variability, and relationship

analysis leads to a more appropriate representation—the feature model—which enables to represent relationships between paradigms.

In this paper, such a paradigm model of AspectJ is proposed. The development of AspectJ paradigm model was based on an extensive comparison of feature modeling and multi-paradigm design (for C++) presented in Sect. 4. The use of the AspectJ paradigm model—a new transformational analysis—was demonstrated on text editing buffers example (Sect. 6) and then the outline of the process was drawn. The process of transformational analysis is more visible and easier to perform with feature models than with tables.

The AspectJ paradigm model presented in this paper provides a basis for further research on multi-paradigm design for AspectJ and its subsequent improvements are expected especially regarding the transformational analysis. The relationship of negative variability tables used in multi-paradigm design and feature modeling has to be investigated. Variability dependency graphs have to be incorporated into the transformational analysis. The transformational analysis results should be noted in a more appropriate form than a textual representation is. A graphical notation would be suitable here, which points to the need for a CASE tool. Besides these immediate issues, the discussion of scope, commonality, variability, and relationship analysis and feature modeling has tackled a deeper question of the relation of multi-paradigm design and generative programming [5].

References

[1] James Coplien, Daniel Hoffman, and David Weiss. Commonality and variability in software engineering. *IEEE Software*, 15(6), November 1998. Available at `http://www.bell-labs.com/people/cope` (accessed on May 14, 2001).

[2] James O. Coplien. Multi-paradigm design and implementation in C++. In *Proc. of GCSE'99*, Erfurt, Germany, September 1999. Presentation slides and notes. Published on CD. Available at `http://www.bell-labs.com/people/cope` (accessed on May 14, 2001).

[3] James O. Coplien. *Multi-Paradigm Design for C++*. Addison-Wesley, 1999.

[4] James O. Coplien. *Multi-Paradigm Design*. PhD thesis, Vrije Universiteit Brussel, Belgium, 2000. Available at `http://www.bell-labs.com/people/cope` (accessed on May 14, 2001).

[5] Krysztof Czarnecki and Ulrich Eisenecker. *Generative Programing: Principles, Techniques, and Tools*. Addison-Wesley, 2000.

[6] Gregor Kiczales et al. An overview of AspectJ. In *Proc. of ECOOP 2001—15th European Conf. on Object-Oriented Programming*, Budapest, Hungary, June 2001. Available at `http://aspectj.org` (accessed on May 14, 2001).

[7] Pavol Návrat. A closer look at programming expertise: Critical survey of some methodological issues. *Information and Software Technology*, 38(1):37–46, 1996.

[8] Valentino Vranić. Towards multi-pradigm software development. Submitted to CIT, 2001.

Appendix C

Multi-Paradigm Design with Feature Modeling

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

UDC 681.5.015

Multi-Paradigm Design with Feature Modeling

Valentino Vranić

Institute of Informatics and Software Engineering
Faculty of Informatics and Information Technology
Slovak University of Technology, Ilkovičova 3, 84216 Bratislava 4, Slovakia
vranic@fiit.stuba.sk, http://www.fiit.stuba.sk/~vranic/

Abstract. In this article, a method for selecting paradigms, viewed as solution domain concepts, appropriate for given application domain concepts is proposed. In this method, denoted as *multi-paradigm design with feature modeling*, both application and solution domain are modeled using feature modeling. The selection of paradigms is performed in the process of feature modeling based transformational analysis as a paradigm instantiation over application domain concepts. The output of transformational analysis is a set of paradigm instances annotated with the information about the corresponding application domain concepts and features. According to these paradigm instances, the code skeleton is being designed. The approach is presented in conjunction with its specialization to AspectJ programming language. Transformational analysis performed according to the AspectJ paradigm model enables an early aspect identification.

1. Introduction

A quarter of a century since the Robert W. Floyd's Turing Award Lecture on paradigms of programming [1], there is no common agreement on the precise meaning of the term *paradigm* in the field of software development. In spite of that, it has been widely used to denote any distinctive enough approach to programming or software development in general. However, as software has finally to be expressed in the form of a program written in one of the programming languages, it is not surprising that the term paradigm is related mostly to programming languages as such.

Programming languages are often categorized according to paradigms they support. This is being done especially according to some of the more widely accepted paradigms, namely procedural, functional, logical, and object-oriented programming. Having several paradigms, each of which has some advantages over the other ones, has naturally lead to the idea of integrating or combining several programming languages, each of which supports some paradigm, into one, *multi-paradigm* programming language.

It is important to note that advantages of a paradigm are relative to the problem being solved. A multi-paradigm programming language itself does not help in multi-paradigm *design*, which is concerned with the issue of selecting a paradigm appropriate for the problem being solved. This issue is addressed by the method proposed in this article, *multiparadigm design with feature modeling* (MPD_{FM}). MPD_{FM} is based on the *small-scale paradigm* view, in which paradigms are understood as *solution domain concepts*. A solution domain is a domain in which a solution is to be expressed. Although some intermediate design notations may be considered as solution domains, too, the ultimate solution domain is a programming language. In a programming language understood as a solution domain, solution domain concepts correspond to programming language mechanisms.

By sticking to the small-scale paradigm view, MPD_{FM} avoids the problems
connected with the lack of precise definitions of the popular, *largescale* paradigms [2,3]. Small-scale paradigms can be represented as configurations of commonality and variability [3]. For this, MPD_{FM} employs *feature modeling*, which enables to explicitly deal with variability of concepts. Feature modeling is applied also to the *application domain*, the domain being solved. The two feature models, the application and solution domain one, enter *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*. The whole process is captured in Fig. 1. In a detailed design and implementation that follows MPD_{FM}, methods specific to the large-scale paradigms pointed to by the small-scale paradigms selected in transformational analysis can be employed.

The rest of the article is structured as follows. Section 2 provides the necessary information on feature modeling in MPD_{FM}. Section 3 describes solution domain feature modeling and shows its use to capture aspect-oriented mechanisms of the AspectJ programming language. Section 4 describes transformational analysis based on feature modeling and demonstrates its application using the AspectJ paradigm model. Section 5 describes briefly code skeleton design. Section 6 discusses related approaches. Section 7 concludes the article.

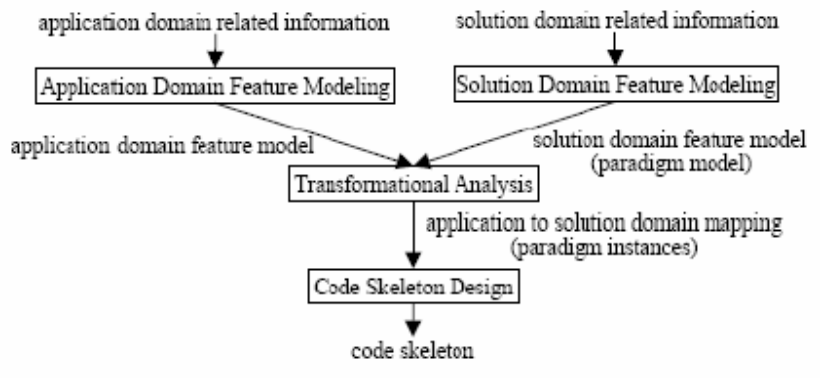


Fig.1. Multi-paradigm design with feature modeling

2. Feature Modeling for Multi-Paradigm Design

Feature modeling is a conceptual domain modeling technique in which concepts in a domaind, understood broadly as an area of interest [4,5], are being expressed by their features taking into account feature interdependencies and variability in order to capture concept configurability.

The origins of feature modeling can be traced back to FODA method [6]. Apart from the mentioned Czarnecki-Eisenecker generative programming, FODA feature modeling has been adopted and adapted by several other domain engineering approaches to software development [7,8,9,10,11,12]. Some work has been devoted primarily to extending feature modeling as such (with respect to UML) [13,14], or even to formalize it [15].

Feature modeling used in MPD_{FM} is based on the Czarnecki-Eisenecker feature modeling employed in generative programming [16,17]. It has been adapted and extended to fit the needs of MPD_{FM} by enabling concept instantiation with respect to instantiation time with concept instances represented by feature diagrams. Further, it brings in parameterization in feature models, enables to represent constraints among features by logical expressions, and introduces concept references to enable to deal with complex feature models (see [18] for details).

This section will provide the necessary information on feature modeling in MPD_{FM} invoking an example of an application domain concept on which further aspects of the method will be demonstrated. An exhaustive

description of the feature modeling for multi-paradigm design may be found in [18,19].

Feature modeling is based on the notions of concept and feature. A *concept* is an understanding of a class or category of elements in a domain. Individual elements that correspond to this understanding are called *concept instances*. A *feature* is an important property of a concept [17]. In general, a feature may be *common*, which means it is present in all concept instances, or *variable*, which means it is present only in some concept instances.

2.1. Feature Diagrams

Feature diagrams are the most important part of a feature model which also may contain information associated with concepts and features and constraints and default dependency rules associated with feature diagrams. An example of a feature diagram is presented in Fig. 2. This figure shows a feature diagram of the text editing buffer concept (adapted from [20], originally inspired by [4]). A text editing buffer represents the state of a file being edited in a text editor. This is modeled by a *mandatory* feature (*File*), which is denoted by a filled circle ended edge. Each text editing buffer employs some memory management scheme to deal with files larger than the working memory (*Memory Management*), which is also modeled by a mandatory feature. Also, each text editing buffer loads and saves its contents into a file, maintains a record of the number of lines and characters, the cursor position, etc., which is modeled by further mandatory features.

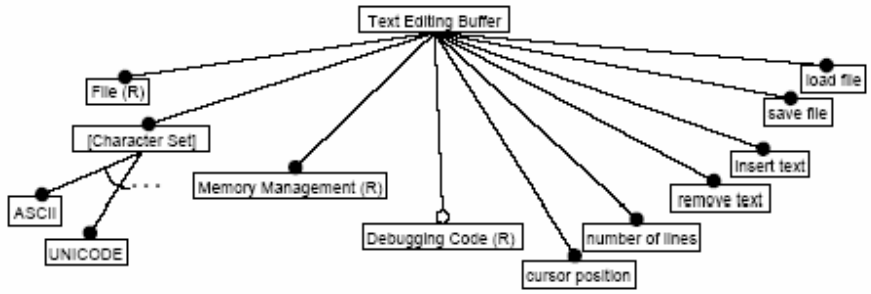


Fig.2. The feature diagram of the *Text Editing Buffer* concept

On the other hand, debugging code might be useful during the development of the text editing buffer, but would probably be undesirable in the final product. Thus, it is modeled by an *optional* feature (*Debugging Code*), which is denoted by an empty circle ended edge.

A text editing buffer will use *exactly one* of the available character sets (*Character Set*).This is specified by *alternative* features (*ASCII*, *UNICODE*...), which are denoted by an empty arc. Note the brackets around the *Character Set* feature's name. This means that it is an open feature; it is expected to have further variable subfeatures. In this case, they would represent other character sets in the group of alternative features, which is indicated by ellipsis placed at this group.

The alternative features just described are actually *mandatory alternative* features. There are also *optional alternative* features of which one or none must be selected. A mixed mandatory-optional alternative feature group is also possible, but its semantics are the same as if all the features were optional alternative.[1]

Feature diagrams may also contain *or-features*, which are denoted by a filled arc (see Fig. 3b). Any non-empty subset or all of the features can be selected from the set of or-features. Having an optional features in a group of or-features would change all its features into simple optional features.

A concept can be referenced as a feature in another or even in its own feature diagram, which is equivalent to the repetition of its feature diagram n the place of the reference. The Ⓡ mark[2] follows the names of concept references in order to distinguish them from the rest of the features. The features *Memory Management* Ⓡ, *File* Ⓡ, and *Debugging Code* Ⓡ in Fig. 2 represent concept references; Fig. 3 shows the feature diagrams of the corresponding concepts.

Note that, with exception of feature references, feature names have no absolute meaning and equally named features may represent different things.

However, no names should be repeated among sibling features, nor among concepts that belong to one feature model.

2.2. Feature Binding

For a variable feature either binding time or binding mode has to be specified. The binding time describes *when* a variable feature is to be bound, i.e. selected to become a mandatory part of a concept instance.

---

[1] This process is being denoted as feature diagram normalization [17].

[2] For technical reasons, presented as (R) in diagrams.

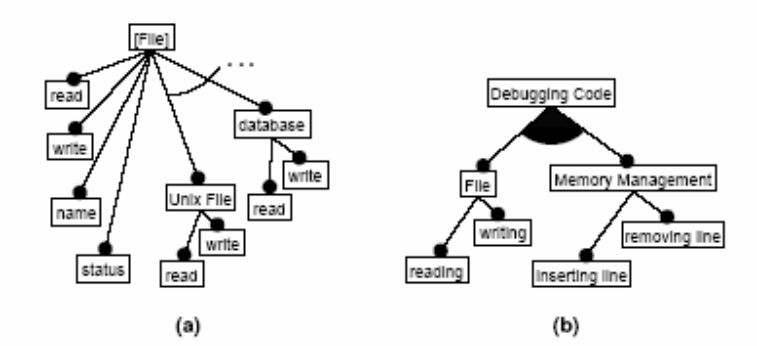


Fig.3. *File* (a) and *Debugging Code* concept (b) feature diagram

It is determined in terms of the binding times available in the solution domain. These usually include: source time, compile time, link time, and run time [4].

At the time of application domain modeling, the solution domain may be unknown or it may be undesirable to pollute the application domain feature model with solution domain details. In that case, using the binding mode instead of the binding time is more appropriate. The binding mode describes *how* a variable feature is bound from the perspective of a running program. A variable feature may be bound *statically*, in which case it cannot be unbound and rebound, or *dynamically*, in which case its binding is fully controlled at run time. Other, more specific binding modes may be defined as well, e.g. changeable binding as an optimized dynamic binding [17].

Consider again the *Text Editing Buffer* concept (presented in Fig. 2); all its variable features are statically bound. The alternative file type features of the *File* concept in Fig. 3a are bound dynamically because we need to be able to change the output file type at run time. On the other hand, it is sufficient to determine the presence of the debugging code parts at source or compile time, so the corresponding or-features in Fig. 3b are bound statically.

2.3. Constraints Associated with Feature Diagrams

Feature diagrams define the main constraints on feature combinations in concept instances. Since feature diagrams are represented as trees, in all but simplest cases it is impossible to express all the constraints solely by a feature diagram. Remaining constraints are introduced in a list of constraints associated with the feature diagram. Also, a list of default dependency rules may be associated with each feature diagram in order to specify which features should or should not appear together by default (details available in [18,19]).

To avoid ambiguities, constraints are specified by predicate logic expressions. In such an expression, a feature name *f* stands for *is in instance(f)*, a predicate which is true if *f* is embraced in the concept instance, and false otherwise. Feature names should be qualified to avoid name clashes, but since each expression is associated with a specific feature diagram, the domain and concept name are unnecessary. Some examples of constraints associated with feature diagrams will be introduced in Sect. 3.2.

2.4. Concept Instantiation

A general definition of a concept instance with respect to instantiation time is given here. An instance *I* of the concept *C* at time *t* is a *C*s specialization achieved by configuring its features which includes the *C*s concept node and in which each feature whose parent is included in *I* obeys the following conditions:

1. All the mandatory features are included in *I*.
2. Each variable feature whose binding time is earlier than or equal to *t* is included or excluded in *I* according to the constraints of the feature diagram and those associated with it. If included, it becomes mandatory for *I*.
3. The rest of the features, i.e. the variable features whose binding time is later than *t*, may be included in *I* as variable features or excluded according to the constraints of the feature diagram and those associatedwith it. The constraints (both feature diagram and associated ones) on the included features may be changed as long as the set of concept instances available at later instantiation times is preserved or reduced.
4. The constraints associated with *C*s feature diagram become associated with the *I*s feature diagram.

A concept may be instantiated in a top-down or a bottom-up fashion. The top-down instantiation starts by the inclusion of the concept node; then inclusion of each feature whose parent has been included is considered. The bottom-up instantiation starts at leaves and proceeds towards the root; a feature may be considered for inclusion only if the set of its features selected for inclusion is correct according to the feature variability defined by the feature model.

A concept instance is represented by a feature diagram derived from the feature diagram of the concept by showing only the features included in the concept instance. A concept instance is regarded as a concept and as such may be a subject of further instantiation.

During instantiation, concept references are treated as regular features. As such, they may appear in concept instances if they are not replaced by the diagrams of concepts they reference prior to instantiation.

In case of an open feature whose form of expected variable subfeatures is specified, the instance may contain any number of the subfeatures of the specified form. If this description is missing (as with the *Character Set* feature in Fig. 2), during instantiation, an open feature is considered as any other non-open feature.

3. Solution Domain Feature Modeling

This section describes how to apply feature modeling to a solution domain understood as a programming language in order to obtain its *paradigm model*, which is necessary for performing transformational analysis. Recall that the term *paradigm* in MPD$_{FM}$ denotes a solution domain concept, which, in turn, corresponds to a programming language mechanism.

Solution domain feature modeling starts with paradigm identification. The paradigms that can be used directly at the topmost level of programs, i.e. *directly usable* paradigms, are identified first, e.g. the class paradigm in AspectJ programming language [21]. All other paradigms are *indirectly usable* paradigms. In AspectJ, an example would be the method paradigm, which, unlike the class paradigm, can be used only inside of a class or aspect.

There may be several levels of indirectly usable paradigms. However, the first-level indirectly usable paradigms would probably be sufficient. This issue must be solved with respect to the purpose of the paradigm model: its use in transformational analysis. It is not feasible to model all

[3] The AspectJ paradigm model is valid for the AspectJ language definition version 1.1.1 (which remains unchanged in the version 1.2 [21]).

the language constructs as paradigms. Much of such low-level paradigms would never be used during transformational analysis because the application domain feature model would be far less detailed. For example, a method in AspectJ may contain an assignment construct, so there could be the assignment paradigm. On the other hand, an application domain feature model would hardly mention assignments, so having the assignment paradigm in the paradigm model is futile.

After identifying directly usable paradigms, binding times (see Sect. 2.2) of the solution domain should be identified. Following that, the first-level paradigm model may be created (Sect. 3.1) and the paradigms may finally be modeled (Sect. 3.2).

3.1. First-Level Paradigm Model

The directly usable paradigm references should appear as features of the solution concept. If a paradigm may appear more than once in a program, its reference should be introduced in the solution domain feature diagram in plural, otherwise in singular.[4] The variability of the paradigm references should be determined according to the restrictions posed by the programming language. If the paradigm reference is a variable feature, its binding time (usually source time) should be determined, too. Finally, initial constraints among paradigms may be determined.

As example, consider the feature diagram of the first-level AspectJ paradigms in Fig. 4. All the directly usable paradigms of AspectJ are modeled
as source time bound optional features of an AspectJ program as a solution concept. Modeling of these directly usable AspectJ paradigms leads to indirectly usable paradigms (which would appear as their features), namely method, overloading, pointcut, inter-type declaration, and advice.

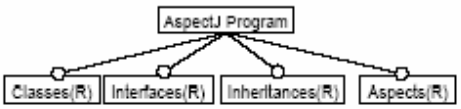


Fig.4. First-level AspectJ paradigms

[4] Plural forms should be defined with respect to singular forms (see [18,19] for details).

3.2. Modeling Paradigms

Each paradigm is considered to be a concept and thus it is 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.

After creating an initial feature model of a paradigm, feature combinations and interactions should be analyzed to determine constraints and, possibly, identify new features (as proposed in [17] for feature modeling in general).

If some feature's subtree is repeated, it should be factored out as a concept into a separate feature diagram and referenced as needed. In a solution domain feature model, this concept may be a paradigm. If it doesn't appear to be a paradigm, it may be considered as an auxiliary concept.

Much of the paradigms correspond to the main constructs, i.e. structures, of the programming language (e.g., the class in AspectJ). In transformational analysis, there *may* be an application domain concept node that matches with the root of such a *structural paradigm*. Thus, it is possible that no application domain node will match with the root of a structural paradigm. This is especially inherent to the aspect paradigm in AspectJ, which will be introduced in Sect. 3.2.[5]

Besides structural paradigms, there are also paradigms that are about the relationship between some language structures. AspectJ examples include inheritance (a relationship between classes), overloading (a relationship between methods), and advice (a relationship between the advice code, i.e. its body, and the join points it affects). In transformational analysis, no application domain node will match with the root of such a *relationship paradigm*.

Three related paradigms from the AspectJ paradigm model—the aspect, advice, and pointcut paradigm—will be presented here to illustrate the process of paradigm modeling.

[5] Examples of aspect paradigm instances without application domain nodes matching their roots may be found in [18]

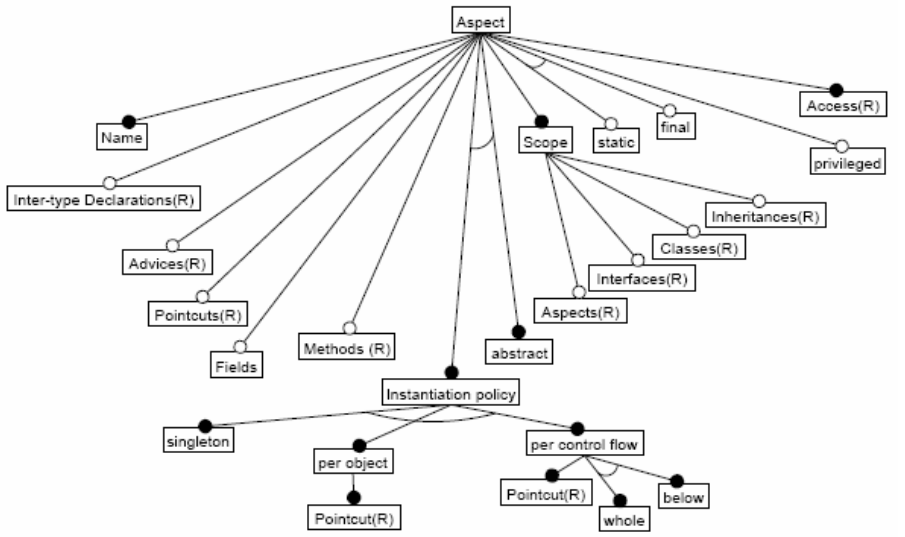


Fig.5. The aspect paradigm in AspectJ

Aspect. The aspect paradigm (see Fig. 5) enables to articulate related structure and behavior that crosscuts otherwise possibly unrelated classes, interfaces, and other aspects (only static aspects are allowed) into a named unit. An aspect is similar to a class in the sense that it also embodies related structure (fields) and behavior (methods). But this structure and behavior is used only to support the crosscutting, which is achieved by two paradigms an aspect is a container of: the advice and inter-type declaration. In addition, the pointcut paradigm is used to specify the join points (where the aspect is to be attached).

As classes, aspects can also be instantiated, but the instantiation is automatic. By default, an aspect is a singleton, i.e. there is a single aspect per Java virtual machine. Furthermore, it is possible to declare that an aspect instantiates per each of the specified objects (executing or target ones) at any of the join points specified by a pointcut or per each flow of control (as it is entered or below it) of the join points specified by a pointcut.

Aspects can be privileged in order to override the access rules of the elements they crosscut. The aspect paradigm enables employing (inside of it) the same paradigms as the class paradigm beside inter-type declarations and pointcuts, which have a special position in it.

The parts of an aspect (without considering inheritance) are known at source time, which means that all the variable features presented in Fig. 5 have source time binding.

The following constraint is associated with the aspect paradigm feature diagram:

final _abstract

which means that the aspect is either final, or abstract.

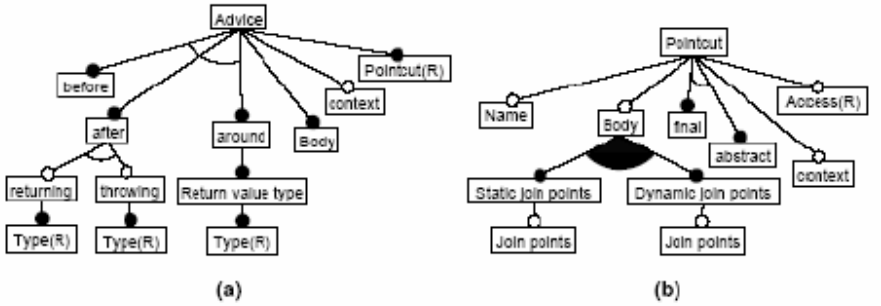


Fig.6. The advice (a) and pointcut (b) paradigm in AspectJ

Advice. Inside of an aspect, the advice paradigm (see Fig. 6a) may be used to articulate the actions to be performed in the context of the join points specified by the pointcut. An advice provides a piece of code (in its body) to be run before, after, or in place (around) of a pointcut. The body of an advice is similar to the body of a method. It can use the join points context exposed by its pointcut.

An *after* advice can run after the execution of each join point specified by the *Pointcut* Ⓡ completes normally, after it throws an exception, or after it does either one. In the last case, no matching based on the type being returned or exception being thrown can be made.

An *around* advice returns a value which will replace the original one at each join point specified by the *Pointcut* Ⓡ. The original join point return value may also be captured and returned, modified or not, by letting the original join point execute inside of the advice body. However, this AspectJ paradigm model does not go into such details as they could hardly be used in the transformational analysis.

Pointcut. The pointcut paradigm (see Fig. 6b) enables to specify the join points. Two kinds of join points exist: static and dynamic join points. Both

are specified at source time, but are really determined later; static join points, such as method calls or executions, are determined at compile time, while dynamic join points, such as all method calls performed by an object of some type, may be determined only at run time. This means that the *Static join points.Join points* feature has compile time binding, while *Dynamic join points.Join points* has run time binding.

A pointcut is a logical expression formed out of primitive pointcuts and the pointcuts already defined. It can be named or not (if it is specified directly in the place of its use). A pointcut can expose the context, i.e. an object or its fields, caught by some of the primitive pointcuts.

The following two constraints are associated with the pointcut paradigm feature diagram:

abstract _Body
Name,Access

which mean that an abstract pointcut cannot have a body (or vice versa), and that an access type can and must be specified in case a pointcut is named, respectively.

Fig.7. The type (a) and access (b) concept

The two auxiliary concepts referenced in the paradigms mentioned above are presented in Fig. 7. The variable features in Figures 5–7 whose binding time has not been explicitly introduced have source time binding.

4. Transformational Analysis

Transformational analysis in MPD_{FM} 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*. The input to transformational analysis are two feature models: the application domain one and the solution domain one. The output of transformational analysis is a set of paradigm instances annotated with the information about corresponding application domain concepts and features. Before presenting the process of transformational analysis and providing an example of it, the key issue of

it—paradigm instantiation over application domain concepts—will be explained.

4.1. Paradigm Instantiation Over Application Domain Concepts

In a paradigm instantiation over application domain concepts, a paradigm, i.e. a solution domain concept, is being instantiated in a bottom-up fashion (see Sect. 2.4) 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.

Not all nodes of application domain concepts need to be mapped. An inner[6] application domain concept node may act as an auxiliary node to ease the categorization of subfeatures. A feature represented by such a node may have no counterpart in the solution domain.[7] Such nodes will be denoted as *mediatory*.

Further, there may (and usually will) be a mismatch in detailedness between the application and solution domain feature model. If solution domain feature model is more detailed, features of some paradigms or even some indirectly usable paradigms will not be mapped to in transformational analysis, but in spite of that they may be included in paradigm instances if determined so from the application domain concept semantics. In case of the application domain feature model is more detailed, there may be no corresponding nodes of the solution domain feature model for some of the non-mediatory nodes or even whole application domain concepts.

Any other non-mediatory feature diagram node of an application domain concept has to be mapped to the corresponding node of a paradigm instance. In general, only the correspondence between the nodes of the same category may be considered, i.e. between two concepts or between two features (note that concept references are also features). Further, semantics of the two nodes have to correspond to each other.

The binding times of the nodes being mapped must correspond. For the purposes of the binding time comparison, mandatory features are treated as if they have the earliest binding time the solution domain provides (which is usually the source time, as discussed in Sect. 2.2). The binding

---

[6] An inner node is a non-root and non-leaf node.

[7] However, there may be other mappings in which such a feature would be mapped.

tme correspondence may mean equality, but it may be relaxed to mean that the binding time of the paradigm feature may not be earlier than required by the application domain concept feature (as that would "only" afect the execution time).

If binding modes were used in the application domain analysis instead of binding times, then the correspondence between the application domain binding modes and the solution domain binding times has to be established. However, in most cases, run time binding corresponds to dynamic binding mode, and the rest of binding times correspond to static binding mode.

In addition, if features are bound later than at the instantiation time, constraints on their variability must correspond, too. To a certain extent, during the instantiation of a paradigm, its constraints may accommodate to the constraints of an application domain concept (as far as they obey the rules defined in step 3 of concept instantiation introduced in Sect. 2.4).

Each mapping between the nodes should be recorded in the form of an annotation, which is graphically presented by connecting the nodes with a dashed line. Annotations other than the feature diagram nodes of an application domain concept should be introduced in dashed boxes. For example, some paradigm features may have specific values intended for use in the code skeleton design (e.g., a name of the class).

4.2. The Process of Transformational Analysis

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 Ps root doesn't match with Cs 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.

The given order of steps of transformational analysis process need not be followed strictly; the main purpose of introducing it is to precisely define the output of transformational analysis. For example, one may choose to instantiate a relationship paradigm on an application domain concept prior to actually determining its structural paradigm.

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

4.3. A Transformational Analysis Example

Consider again the text editing buffers debugging code concept whose feature diagram is shown in Fig. 3c. Assume that the *File* feature matches with the class paradigm, and that its features *read* and *write* represent methods, while *name* and *status* are its attributes. Further, assume that the file types inherit from this base file class. In this example, transformational analysis of the text editing buffer's file debugging code part will performed. For this purpose, the feature corresponding to it, *Debugging Code.File*, will be factored out as a concept.

As may be seen from Fig. 3c, the file debugging code consists of reading and writing part. *Debugging Code.File.reading* is concerned with reading files and supposed to provide an information on the type of the file before it

has been read. *Debugging Code.File.writing* should provide an information on the status of the file after it has been written to.

One could choose the method paradigm for both these features because they represent functionality. However, a more careful examination of the description of the two features given in the previous paragraph reveals that this functionality is performed in connection with some other functionality. Recalling that the debugging code should be plugable, and thus separated from the rest of the code as much as possible, brings us to another form of expressing functionality in AspectJ: the advice paradigm.

As shown in Fig. 8, both *Debugging Code.File.reading* and *Debugging Code. File.writing* match with the body of a separate advice. An advice performs its actions with respect to the join points specified by a pointcut. In both cases, the pointcut would be unnamed, as we need it only for this one application, and thus final (*Pointcut.final*). The context of the read method execution object would be needed to determine the file type in reading file advice and file status in writing file advice. Thus, the context should be exported by the pointcut (*Pointcut.context*) to be used by the advice (*Advice.context*). The reading file advice should be run before (*Advice.before*) the calls to *File.read* method, while the writing file advice should be run after (*Advice.after*) the calls to *File.write* method.

Note that Fig. 8 presents actually five paradigm instances: two pointcuts, two advices, and one aspect. Since paradigm instances are concept instances (see Sect. 2.4), and concept instances are specialized concepts, each paradigm instance could be presented in a separate diagram, as well, with enclosing paradigm referencing the enclosed paradigm instances.

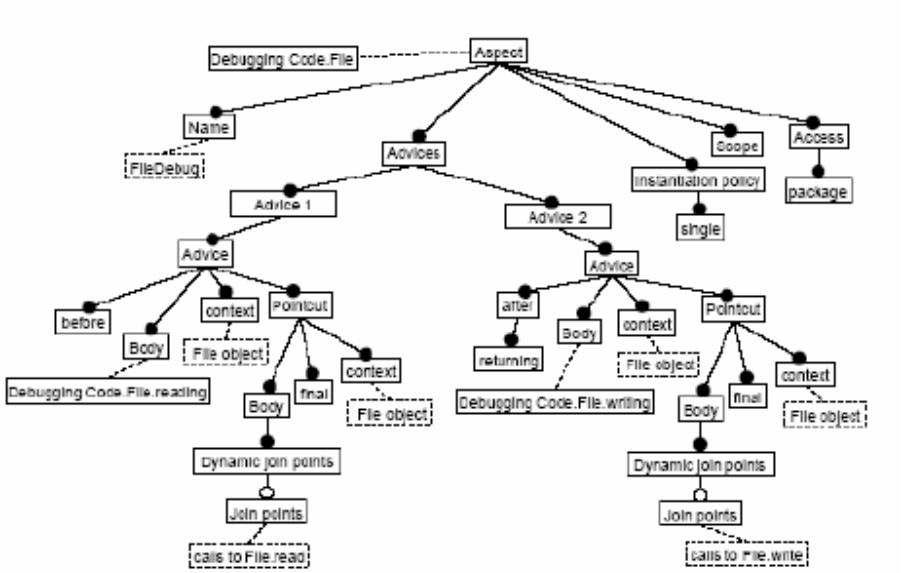


Fig.8. The file debugging code concept transformational analysis; an aspect with two advices

5. Code Skeleton Design

Code skeleton design is performed by traversing paradigm instances and writing the source code manually. The paradigm instances obtained in transformational analysis define the code skeleton, but the notes made during transformational analysis (as those accompanying the feature model element transformational analysis example) may also help mold the skeleton more accurately and make it more concrete.

In code skeleton design, first the instances of structural paradigms are transformed into code. Subsequently, the instances of relationship paradigms are transformed, too.

The first step produces the basis for the second one because relationship paradigms are usually not represented by independent syntactical structures, but rather attached to the syntactical structures representing structural paradigms.

Following the transformational analysis of the file debugging code concept presented as a paradigm instance in Fig. 8, we could write the following code:

```
aspect FileDC {
     before(File f): target(f) && call(* File.read(..)) {
          . . . }
     after(File f): target(f) && call(* File.write(..)) {
          . . . }
}
```

The code represents an aspect with two advices. The first one is being executed before reading any file, and the second one after writing each file. Both advices expose the current File object which is to be utilized in the advice bodies in order to output the file type in the first advice, and file status in the second advice.

6. Related Approaches

Conceptually, MPDFM is closest to *multi-paradigm design* (MPD) [4]. By employing feature modeling, MPD_{FM} introduces several improvements. One of the most important improvements is overcoming the MPD's problem of having to decide the conceptual correspondence between the paradigm and application domain concept at once.[8] By performing transformational analysis as a bottom-up paradigm instantiation over application domain concepts, the correspondence is decided part by part, at lower level features, which are more easily compared.

Feature modeling in MPD_{FM} also enables to visualize hierarchical relationships between the commonalities and variabilities in both application and solution domain models. In MPD, variability dependency graphs are used for this, but they are not capable of expressing variability constraints as feature diagrams are. Moreover, they are used only in application domain models, while representing hierarchical relationships between solution domain concepts, i.e. paradigms, is also needed.

While binding time in MPD is an attribute of a concept as a whole, in MPD_{FM} binding time is specified precisely where it applies: at individual variable features. Also, instantiation in MPD is just an attribute of a concept, while in MPD_{FM} it may be modeled in more details by features.

---

[8] In fact, MPD uses different terminology than MD_{FM}, e.g. a *domain* in MPD denotes a *concept* P in MPD_{FM}. See [20] for a detailed comparison.

Feature modeling enables to have a visual control over transformational analysis in MPD_{FM}. Its output, annotated paradigm instances, provide enough information about the mapping between the application and solution domain concepts to obtain the main part of the code skeleton from their trees, while in MPD, transformational analysis results are only a guide in choosing a paradigm for an application domain concept.

Negative variability, which is in MPD presented in separate tables (negative variability tables), is in feature modeling modeled by features. The negative variability features of paradigms are actually their specializations (e.g., consider the template specialization [4]).

A design method proposed in connection with *multi-paradigm programming in Leda* [22] is also related to MPD_{FM}. However, while MPD_{FM} is domain-oriented, Leda design method is concerned with the design of one system.

The substantial difference is that MPD_{FM} is performed in a bottom-up fashion, and Leda design method in a top-down fashion, which is related to the large-scale paradigm view it's being based on. The granularity of large-scale paradigms corresponds to the top level of a system or subsystem. However, the selection of the main paradigm for the system or a part of it is a hard decision to make at once. In Leda design method, a paradigm is selected based on the analysis of the application of each available paradigm impact to lower levels of the system.

Application domain feature modeling is a common activity of both *generative programming* [17] and MPD_{FM}, so it may be performed without having to decide which one of these approaches will be employed. Taking a closer look at generative programming reveals that it also aims at employing multiple paradigms. The difference is in the selection of paradigms: while in MPD_{FM} it is performed directly as a matter of the primary concern, in generative programming it can be viewed as being built into the generator.

7. Conclusions and Further Work

A new method of multi-paradigm software development called *multi-paradigm design with feature modeling* (MPDFM) has been proposed in this article. In this method, feature modeling is used to model both application and solution domain. For this purpose, Czarnecki-Eisenecker feature modeling [17] has been extended and adapted.

Consequently, transformational analysis, the key activity of multi-paradigm design, in which paradigms (solution domain concepts) appropriate for given application domain concepts are being selected, has been proposed in terms of feature modeling as a bottom-up paradigm instantiation over application domain concepts. Subsequently, code skeleton, the final output of MPD_{FM}, is obtained by traversing the trees of annotated paradigm instances, which represent the output of transformational analysis, and writing the source code manually.

To obtain the whole code skeleton, transformational analysis should be performed for each application domain concept, as explained in Sect. 4.2. It is also possible to perform transformational analysis only of some application domain concepts (e.g., the critical ones) and do the rest of the design without MPD_{FM}. The rest of the design would be restricted by such partial transformational analysis results.

Creating a feature model of a solution domain can be viewed as a specialization of MPD_{FM} with respect to transformational analysis. Parts of such a specialization of MPD_{FM} to AspectJ regarding its aspect-oriented paradigms have been presented and applied in this article; its whole paradigm
model is available in [18]. The AspectJ paradigm model has been successfully applied in transformational analysis of a feature model of the domain of feature modeling itself [18] (the feature model of feature modeling is available also in [19]).

From the viewpoint of aspect-oriented software development, transformational analysis according to the AspectJ paradigm model enables an early aspect identification. Of course, such aspects are valid in the context
of AspectJ only, but this is also the case with language-specific design notations such as [23], which have to be used due to large differences in aspect-oriented mechanisms provided by individual aspect-oriented languages. An important difference is that an application domain model expressed in such a notation is heavily language-dependent, which is not
the case with an application domain model in MPD_{FM}.

In MPD_{FM}, both application and solution domain feature models are reused as a whole: different application domains may be implemented in the same solution domain, and an application domain may be implemented in several solution domains. However, some domains overlap, and this happens even if one of them is an application domain and the other one is a solution domain. Thus, the issue of overlapping domains is worth considering as a step towards reuse of individual concepts.

The reuse of individual concepts which are similar to each other would require their generalization. Subsequently, they would appear as specializations of a more general concept. This would be particularly useful for paradigm models of related programming languages. Another interesting topic for further work would be experimenting with specialization of MPD_{FM} to design patterns or other intermediate solution domains and combinations of these in conjunction with programming languages as such.

Acknowledgements. The work was partially supported by Slovak Science Grant Agency VEGA, project No. 1/0162/03. I would like to thank Pavol N´avrat and M´ aria Bielikov´a for their valuable suggestions.

8. References

1. Floyd, R.W.: The paradigms of programming. Communications of the ACM **22** (1979) 455–460
2. Coplien, J.O.: Multi-paradigm design and implementation in C++. Slides and notes of the tutorial given at *1st International Conference on Generative and Component-Based Software Engineering (GCSE'99)*, Erfurt, Germany (1999) Available at http://www.old.netobjectdays.org/mirrors /stja.cd/Beitraege/ JimCoplien/Tutorial.ppt (accessed in June 2005).
3. Vranić, V.: Towards multi-paradigm software development. Journal of Computing and Information Technology (CIT) **10** (2002) 133–147
4. Coplien, J.O.: Multi-Paradigm Design for C++. Addison-Wesley (1999)
5. Coplien, J.O.: Multi-Paradigm Design. PhD thesis, Vrije Universiteit Brussel, Belgium (2000) Available at http://users.rcn.com/jcoplien/Mpd/Thesis/Thesis.pdf (accessed in June 2005).
6. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (FODA): A feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, USA (1990) Available at [24] (accessed in June 2005).
7. Chastek, G., Donohoe, P., Kang, K.C., Thiel, S.: Product line analysis: A practical introduction. Technical Report CMU/SEI-2001-TR-001, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, USA (2001) Available at [24] (accessed in June 2005). 8. Geyer, L.: Feature modelling using design spaces. In: Proc. of the 1st German Product Line Workshop (1. Deutscher Software-Produktlinien Workshop, DSPL-1), Kaiserslautern, Germany, IESE (2000) Available at http://wwwagss.informatik.uni-kl.de/Veroeffentl/ FeatureModelingUsingDesignSpaces.pdf (accessed in June 2005).
9. Griss, M.L., Favaro, J., d'Alessandro, M.: Integrating feature modeling with

the RSEB. In Devanbu, P., Poulin, J., eds.: Proc. of 5th International Conference on Software Reuse, Victoria, B.C., Canada, IEEE Computer Society Press (1998) 76–85 Available at http://www.favaro.net/john/home/publications/rseb.pdf (accessed in June 2005).
10. Kang, K.C., Kim, S., Lee, J., Kim, K., Shin, E., Huh, M.: FORM: A feature-oriented reuse method with domain-specific reference architectures. Annals of Software Engineering **5** (1998) 143–168
11. Simos, M.A.: Organization domain modeling (ODM): Formalizing the core domain modeling life cycle. In: Proc. of the 1995 Symposium on Software reusability, Seattle, Washington, United States, ACM Press (1995) 196–205
12. Software Engineering Institute, Carnegie Mellon University: A framework for software product line practice. (http://www.sei.cmu.edu/productlines/framework.html) Accessed in June 2005.
13. Claub, M.: Modeling variability with UML. In: Proc. of Net.ObjectDays 2001, Young Researchers Workshop on Generative and Component-Based Software Engineering, Erfurt, Germany, tranSIT (2001) 226–230
14. Riebisch, M., B¨ ollert, K., Streitferdt, D., Philippow, I.: Extending feature diagrams with UML multiplicities. In: Proc. of the 6th Conference on Integrated Design and Process Technology (IDPT 2002), Pasadena, California, USA, Society for Design and Process Science (2002) Available at http://www.theoinf.tu-ilmenau.de/»riebisch/publ/IDPT2002-paper.pdf, accessed in June 2005).
15. Jia, Y., Gu, Y.: The representation of component semantics: A feature-oriented approach. In Crnkovi´c, I., Larsson, S., Stafford, J., eds.: Proc. of the Workshop on Component-based Software Engineering: Composing Systems From Components (a part of 9th IEEE Conference and Workshops on Engineering of Computer-Based Systems), Lund, Sweden (2002) Available at http://www.idt.mdh.se/»icc/cbse-ecbs2002/jiayu.pdf (accessed in June 2005).
16. Czarnecki, K.: Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models. PhD thesis, Technical University of Ilmenau, Germany (1998) Available at http://www.prakinf.tu-ilmenau.de/»czarn/diss (accessed in June 2005).
17. Czarnecki, K., Eisenecker, U.W.: Generative Programing: Methods, Tools, and Applications. Addison-Wesley (2000)
18. Vranić, V.: Multi-Pradigm Design with Feature Modeling. PhD thesis, Slovak University of Technology in Bratislava, Slovakia (2004) Available at http://www.fiit.stuba.sk/»vranic.
19. 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) 122–137 20. Vrani´c, V.: AspectJ paradigm model: A basis for multi-paradigm design for AspectJ. In Bosch, J., ed.: Proc. of 3rd International Conference on Generative and Component-Based Software

Engineering (GCSE 2001). LNCS 2186, Erfurt, Germany, Springer (2001) 48–57
21. Eclipse.org: AspectJ project home page. (http://eclipse.org/aspectj) Accessed in June 2005.
22. Knutson, C.D., Budd, T.A., Vidos, H.: Multiparadigm design of a simple relational database. ACM SIGPLAN Notices **35** (2000) 51–61
23. Stein, D., Hanenberg, S., Unland, R.: A uml-based aspect-oriented design notation for aspectj. In Kiczales, G., ed.: Proc. of 1st International Conference on Aspect-Oriented Software Development, ACM Press (2002) 106–112
24. Software Engineering Institute, Carnegie Mellon University: Home page. (http://www.sei.cmu.edu) Accessed in June 2005.

Valentino Vranić is a researcher at Institute of Informatics and Software Engineering, Faculty of Informatics and Information Technology of the Slovak University of Technology in Bratislava. He holds a Bc. (BSc.) and Ing. (MSc.) in information technology, and PhD. in program and information systems, all from the Slovak University of Technology in Bratislava. His main research interests are multi-paradigm software development, domain engineering, and aspect-oriented programming.

Appendix D

Reconciling Feature Modeling: A Feature Modeling Metamodel

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.

Reconciling Feature Modeling: A Feature Modeling Metamodel

Valentino Vranić

Institute of Informatics and Software Engineering
Faculty of Informatics and Information Technology
Slovak University of Technology, Ilkovičova 3, 84216 Bratislava 4, Slovakia
vranic@fiit.stuba.sk, http://www.fiit.stuba.sk/~vranic

Abstract. Feature modeling, a conceptual domain modeling technique used mainly in domain engineering, proved as useful for representing configurability of concepts by dealing explicitly with commonality and variability. This paper introduces feature modeling for multi-paradigm design as an integrative approach and evaluates other approaches to feature modeling. These approaches differ mainly in the notation of feature diagrams, but there are also differences regarding the basic notions. The commonalities and variabilities of the domain of feature modeling are concisely expressed using feature modeling itself in the form of a feature modeling metamodel which may serve both for further reasoning on feature modeling and as a basis for developing feature modeling tools.

1 Introduction

Feature modeling is a conceptual domain modeling technique in which concepts are expressed by their features taking into account feature interdependencies and variability in order to capture the concept configurability [1].

A *domain* is understood here as an area of interest [2]. Two kinds of domains can be distinguished based on their role in software development: application and solution domains [2]. An *application domain*, sometimes denoted as a problem domain [2], is a domain to which software development process is being applied. A *solution domain* is a domain in which a solution is to be expressed (usually a programming language).

The origins of feature modeling are in FODA method [3], but several other approaches to feature modeling have been developed. Feature modeling has been used to represent models of application domains in many domain engineering approaches to software development beside FODA such as FORM [4], ODM [5], or generative programming [1].

Feature modeling is used also in *multi-paradigm design with feature modeling* (MPD_{FM}), a method introduced in [6] that follows the same process framework as Coplien's multi-paradigm design [2], where it was adapted to express both application and solution domain concepts in order to simplify finding a correspondence and establishing the mapping between the application and solution domain concepts in transformational analysis. Feature modeling used in MPD_{FM}

is based on Czarnecki-Eisenecker feature modeling [1]. However, it introduces the following new concepts: concept instantiation with respect to feature binding time, representing concept instances visually using feature diagrams, concept references, parameterization of feature models, expressing constraints and default dependency rules as logical expressions, and a dot convention for referring to concepts and features.

The rest of the paper is structured as follows. First, Sect. 2 introduces feature modeling for multi-paradigm design as an integrative approach to feature modeling. Next, Sect. 3 evaluates other approaches to feature modeling. Finally, based on this analysis, Sect. 4 presents a feature modeling metamodel as a feature model. Sect. 5 concludes the paper and proposes the issues for further research.

2 Feature Modeling for Multi-Paradigm Design

Feature modeling is a conceptual domain modeling technique in which concepts are being expressed by their features taking into account feature interdependencies and variability in order to capture the concept configurability [1]. Feature modeling presented in this section is based on the Czarnecki-Eisenecker feature modeling [1], which has been adapted and extended to fit the needs of $\mathrm{MPD_{FM}}$.

A *concept* is an understanding of a class or category of elements in a domain. Individual elements that correspond to this understanding are called *concept instances*.

A *feature* is an important property of a concept [1]. A feature may be *common*, in which case it is present in all concept instances, or *variable*, in which case it is present only in some concept instances. The features connected directly to a concept or feature are being denoted as its *direct features*; all other features are its *indirect features* [1].

Any feature may be isolated and modeled further as a concept, therefore being a feature is actually a relationship between two concepts. However, the concepts identified only in the context of other concepts, i.e. as their features, will be referred to as features exclusively in order to emphasize the main concepts in a domain.

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 a directed tree whose root represents a concept and the rest of the nodes represent its features.

2.1 Feature Diagrams

Each concept is presented in a separate feature diagram. A feature diagram is drawn as a directed tree with edge decorations. The root represents a concept, and the rest of the nodes represent features. Edges connect a concept with its features, and a feature with its subfeatures.

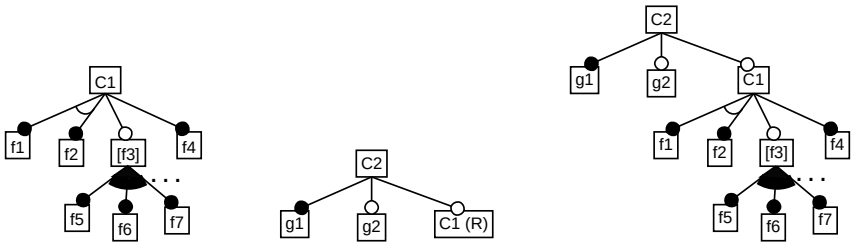
Concept instances are represented by configurations of concept features, which are achieved by a selection of the features according to their variability. A feature can be included in a concept instance only if its parent has been

included. A concept instance *must* have all the mandatory features and *can* have the optional features.

There are two types of edges used to distinguish between *mandatory* features, ended by a filled circle, and *optional* features, ended by an empty circle. A concept instance *must* have all the mandatory features and *can* have the optional features.

The edge decorations are drawn as arcs connecting disjunct subsets of the edges originating in the same node. There are two types of arcs, an empty and filled one, used to denote *alternative features* and *or-features*, respectively. Exactly one feature can be selected from the set of alternative features, and any subset or all of the features can be selected from the set of or-features. If optional, each selected alternative or or-feature may still be left out.

A concept or feature may be *open*, which means it is expected to have new direct variable subfeatures. This is indicated directly in feature diagrams by introducing the open concept or feature name in square brackets and optionally by ellipsis at its subfeatures.



(a) **(b)** **(c)**

Fig. 1. Feature diagram examples.

An example of a feature diagram with different types of features is presented in Fig. 1a. Features f_1, f_2, f_3, and f_4 are direct features of the concept C_1, while other features are its indirect features. Features f_1 and f_2 are mandatory alternative features. Feature f_3 is an optional feature. Features f_5, f_6, and f_7 are mandatory or-features. Feature f_3 is open; ellipsis indicates that new features are expected in the existing group of or-features.

A concept can be referenced as a feature in another or even in its own feature diagram, which is equivalent with the repetition of the whole feature diagram of the concept. Figure 1b presents the feature diagram of the concept C_2 that refers to the concept C_1. Figure 1c presents the same diagram, but with the reference C_1® expanded. To distinguish concept references from the rest of the features in a feature diagram, the ® mark[1] is being put after the name of a concept reference.

[1] For technical reasons, it will be presented as (R) in diagrams.

Additional information may be associated with concepts and features, which depends on the application, so it should be as configurable as possible.[2] A concept reference may be associated with its own information as any other feature, but the information associated with the concept it references applies to it, too.

2.2 Constraints and Default Dependency Rules

Feature diagrams define the main constraints on feature combinations in concept instances. Since feature diagrams are represented as trees, in all but simplest cases it is impossible to express all the constraints solely by a feature diagram. Additional constraints are expressed in a list of constraints associated with the feature diagram. Also, a list of default dependency rules is associated with each feature diagram in order to specify which features should or should not appear together by default.

Constraints and default dependency rules are specified by predicate logic expressions formed out of specific and parameterized names of concepts and features (see Sect. 2.3), and commonly used logical connectives (e.g., not $\neg$, and $\wedge$, or $\vee$, xor $\veebar$, implication $\Rightarrow$, and equivalence $\Leftrightarrow$), commonly used quantifiers (e.g., universal quantifier $\forall$ and existential quantifier $\exists$), and parentheses. A feature name *f* in constraint or default dependency rule expressions stands for *is_in_instance(f)*, a predicate which is true if *f* is embraced in the concept instance, and false otherwise.

The intention of using predicate logic to express constraints and default dependency rules is to avoid ambiguities natural language is prone to. At this stage, the automated evaluation of the constraints and default dependency rules has not been considered, although that would certainly be useful.

Feature names in expressions should be qualified to avoid name clashes, but since each expression is associated with a specific feature diagram, the domain and concept name are unnecessary. To avoid repeating long qualifications, as in $A.B.C.x \vee A.B.C.y$, the common qualification may be introduced in front of the expression, e.g. $A.B.C.(x \vee y)$.

Constraints A list of constraints associated with a feature diagram is a conjunction of the expressions it consists of. Thus, for a concept instance to be valid, all the constraints associated with the feature diagram must evaluate to true. Obviously, in case of a contradiction among the constraints, it is impossible to instantiate the concept.

Constraints express mutual exclusions and requirements among features, i.e. they determine which features cannot appear together and which must appear together, respectively. A single constraint may express both mutual exclusions and requirements.

Constraints have numerous equivalent forms, but they should be kept in the form which is as comprehensible as possible. Bearing this in mind, mutual

[2] Such a configurability has been implemented in AmiEddi, a feature modeling editor (available at [19]), through so-called metamodel editor [21, 22].

exclusions may be expressed by connecting features with xor, while requirements may be expressed as implications or equivalences, depending on whether the requirement is bidirectional or not.

As has been said, the main constraints are expressed directly in feature diagrams and thus need not be repeated in the information associated with them. However, sometimes it may be needed to change a feature diagram constraint to associated one, or vice versa. In a feature diagram, a mutual exclusion is expressed by alternative features. A requirement is expressed by a variable subfeature whose parent is also a variable feature: the subfeature *requires* its parent to be included. Also, a requirement may be expressed by or-features: at least one feature is *required* from a set of or-features.

Default Dependency Rules A list of default dependency rules associated with a feature diagram is a disjunction of an implicit (not displayed) *true* and the expressions it consists of. The implicit *true* disjunct in a list of default dependency rules assures that it always evaluates to *true*.

Default dependency rules determine which features should appear together by default. Default dependency rules are applied at the end of the process of concept instantiation if there are variable features left such that no explicit selection has been made among them. Which of these features will be included in the concept instance is decided according to the default dependency rules.

2.3 Parameterization in Feature Models

A *parameterized name* of a concept or feature has the form: $p_1 p_2 \ldots p_n$, where for each $i \in [1, n]$ p_i is either a parameter or specific string and where exists $j \in [1, n]$ such that p_j is a parameter. For each parameter, a set of possible strings that may be substituted for it has to be defined in its description. Parameters are introduced in <> brackets to distinguish them from specific strings.

Name parameterization enables to reason more generally about concepts and features. An example of a parameterized name is *Singular Form<i>*, where *<i>* is a natural number. The specific names corresponding to this parameterized name are: *Singular Form1*, *Singular Form2*, etc.

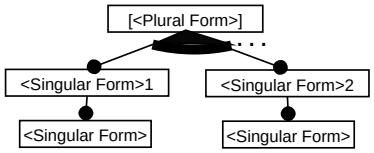


Fig. 2. Dealing with plural forms using a parameterized concept.

Name parameterization is the only way to express constraints and default dependency rules about subfeatures of an open feature because their number

is unknown. Consider the feature diagram in Fig. 2 (ignoring the parameterizations of <*Singular Form*> and <*Plural Form*> for the moment). The feature <*Plural Form*> is open; further direct variable subfeatures of the form <*Singular Form*><*i*>, where <*i*> is a natural number, are expected at it. The parameterized name <*Singular Form*><*i*> is exactly how all these features may be referred to.

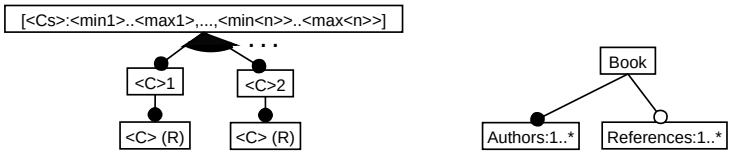
A *parameterized concept* or *feature* is a concept or feature whose name is parameterized. Parameterized features may appear only in feature diagrams of parameterized concepts; otherwise, the feature model would be inconsistent since it would define a set of different feature diagrams for a single concept. For the same reason, parameterized concepts may not be referenced in feature diagrams of specific (i.e., non-parameterized) concepts.

Figure 2 shows an example of a parameterized concept. The name <*Plural Form*> is a plural form of <*Singular Form*><*i*>.<*Singular Form*>. Using a parameterized concept, we avoided drawing a separate feature diagram for each concept.

2.4 Representing Cardinality in Feature Models

Parameterized concepts are capable of representing UML style cardinalities represented by a comma separated list of the *minimum..maximum* cardinality pairs [7]. This may be achieved by a feature diagram in Fig. 3a with the following constraint which will assure the appropriate number of features according to the specified cardinality:

$$\bigvee_{<i>=1}^{<n>} ((\max<i> \neq * \Rightarrow \bigvee_{<j>=<min<i>>}^{<max<i>>-<min<i>>+1} \bigwedge_{k=1}^{i} <C><k>) \wedge$$
$$\wedge (\max<i> = * \Rightarrow \bigwedge_{k=1}^{<min<i>>} <C><k>))$$



(a) **(b)**

Fig. 3. Parameterized concept for representing cardinality (a) and an example of its application (b).

The parameter <*Cs*> is the plural form of the parameter <*C*>. Note that parameters <*min*<*i*>> and <*max*<*i*>> are in fact doubly parameterized. This

is to enable the parameterization of the number of *minimum..maximum* cardinality pairs.

The values allowed for both minimum and maximum cardinalities are natural numbers. Also, an additional value denoted by an asterisk is allowed for the maximum cardinality value meaning "many," as in [7]. Zero cardinality is achieved by referencing the concept <*Cs*>:<*min1*>..<*max1*>,. . . ,<*min*<*n*>>..<*max*<*n*>> as an optional feature.

This parameterized concept may be applied to any domain by including it in the feature model of the domain. Next, the set of the singular and plural forms of concept names corresponding to each other (representing possible values for <*C*> and <*Cs*>, respectively) has to be defined. Obviously, a feature model must include the concepts singular form concept names refer to. Finally, specific concept name and a set of *minimum..maximum* cardinality pairs should be substituted. An example is shown in Fig. 3b; a book has at least one author, and it may have zero (modeled by the optionality of *References:1..**) or more references.

2.5 Concept Instantiation

An instance I of the concept C at time t is a configuration of C's features which includes the C's concept node and in which each feature whose parent is included in I obeys the following conditions:

1. All the mandatory features are included in I.
2. Each variable feature whose binding time is earlier than or equal to t is included or excluded in I according to the constraints of the feature diagram and those associated with it. If included, it becomes mandatory for I.
3. The rest of the features, i.e. the variable features whose binding time is later than t, may be included in I as variable features or excluded according to the constraints of the feature diagram and those associated with it. The constraints (both feature diagram and associated ones) on the included features may be changed as long as the set of concept instances available at later instantiation times is preserved or reduced.
4. The constraints associated with C's feature diagram become associated with the I's feature diagram.

A concept instance is represented by a feature diagram derived from the feature diagram of the concept by showing only the features included in the concept instance. A concept instance is regarded further as a concept and as such may be considered for further instantiation at later instantiation times. During instantiation, a concept reference may appear in a concept instance as any other feature if it is not replaced by the diagram of the concept it references prior to instantiation.

3 Other Approaches to Feature Modeling

Feature modeling originates from Software Engineering Institute (SEI), where it was used in FODA method [3] developed there, which became a part of their

MBSE method. Recently, MBSE has been replaced by PLP approach [8, 9], which also employ feature modeling. An adapted version of FODA feature modeling is also a part of FORM method [4].

Since the publishing of FODA in 1990, several approaches have adopted FODA feature modeling, often in an adapted version [10, 1, 11]. Some work has been devoted primarily to extending feature modeling as such (with respect to UML) [12, 13], or even to formalize it [14].

Czarnecki-Eisenecker feature modeling [1] generalized FODA feature modeling notation and accepted a more general notion of a feature from ODM approach in which features are associated with particular domain practitioners and domain contexts [5], i.e. a feature is any concept instance property important to some of the stakeholders [1]. Such an understanding of a feature has been adopted also by FORM [4], a direct ancestor of FODA.

Czarnecki-Eisenecker feature modeling is also more abstract than FODA or FORM feature modeling. In Czarnecki-Eisenecker feature modeling, relationships between a feature and its subfeatures don't have any predefined semantics; the relationship is fully determined by the semantics of subfeatures. FORM feature modeling defines three types of relationships of a feature to its subfeature: composed-of, generalization/specialization, and implemented-by. Moreover, each feature is classified as a capability, operating environment, domain-technology, or implementation technique feature.[3] According to their type, features are placed into one of the four layers feature diagrams are divided into. On the other hand, Matthias Riebisch argues against the classification of features according to FORM and proposes to classify features into functional, interface, and parameter features [15]. Therefore, it seems that it is better not to enforce such predefined feature categories in feature modeling.

Concept instantiation with respect to feature binding time (see Sect. 2.5) is a generalization of concept instantiation as proposed in [1]. Compared to the set representation proposed in [1], even if the features are qualified as proposed in Sect. 2, feature diagrams are a more appropriate way to represent concept instances. Moreover, they enable to represent concept instantiation in time.

The following sections discuss other solutions to referencing concepts, representing constraints and default dependency rules, and representing cardinalities.

3.1 Concept References

The problem of coping with complex feature diagrams has been recognized already in [1], where complex diagram are divided into a number of smaller diagrams, which then may be referred to in the main diagram by introducing their roots.

Concept references, introduced by MPD_{FM} feature modeling, are a logical extension of this idea. MPD_{FM} feature modeling specifies how the information

[3] This classification has been proposed already in [3], but since FODA was concerned with user visible features, it dealt only with (application) capabilities.

associated with the concept applies to its references and how it may be adapted to the needs of a particular reference.

Concept references enable a concept to reference itself (directly or indirectly). This enables feature diagrams to be viewed as trees while being in conformance with the fact that feature diagrams in general are directed graphs.

To refer to a concept or features unambiguously, a common dot convention is used in MPD_{FM} feature modeling. A similar convention is used in FeatuRSEB [10], though without taking into account domain names, which may lead to ambiguities when talking about concepts and features from several domains.

3.2 Representing Constraints and Default Dependency Rules

In MPD_{FM}, constraints and default dependency rules are expressed concisely as logical expressions. Logical expressions are capable of expressing both mutual exclusions and requirements among features. In fact, a single logical expression may encompass both types of the constraints. In FODA feature modeling, as well as in Czarnecki-Eisenecker feature modeling, constraints are expressed by explicitly stating which feature is mutually exclusive or requires which other feature.

In [16], constraints are written in an adapted version of Object Constraint Language (OCL) used in Unified Modeling Language [7]. It is merely a matter of preference whether to use OCL syntax or traditional mathematical symbols for logical connectives (e.g., implies vs. $\Rightarrow$). However, in [16], constraints are also accompanied by the information to be passed to the developer who instantiates the concepts that, for example, another feature has to be selected. This significantly reduces the readability of constraints. Better, such messages could be generated or a whole constraint could be passed instead.

Incorporating messages to developers significantly reduces the readability of such constraints. Moreover, such messages to the developer may be generated or, even better, a whole constraint may be passed instead.

The proposed form of expressing constraints and default dependency rules may be applied also to the constraints expressed directly by feature diagrams. This way, a whole feature diagram may be represented as a set of logical expressions. For the purpose of a graphical representation, a set of views of the feature diagram could be then defined. For each view, the relationships that should be shown would have to be specified with respect that the feature diagram should be a tree. The new constraints for the feature diagram could be then calculated to avoid duplicity (some of the constraints would be expressed in the feature diagram). In order to distinguish the primary relationships between the features expressed in a feature diagram from the constraints associated with it, one of the views could be denoted as primary.

The need to represent feature diagrams in a graphically independent form has been identified also in [17]. The formalized feature modeling proposed in [14] actually relinquishes the feature diagrams completely, and with them the primary relationships between the features, too.

3.3 Representing Cardinalities

In the original Czarnecki-Eisenecker feature modeling, introducing feature cardinalities was strongly avoided arguing that since the only semantics of an edge is whether to assert a feature or not, cardinality would only mean to assert it several times, which is useless [1, p. 117]. Instead, to model the cardinality as a feature was recommended. In spite of this, a later work proposes to use the UML-style cardinalities with features [18]. Also, a generalized form of alternative and or-features is introduced in which the number of features which may be included is specified also as a cardinality (which does not contradict to the original Czarnecki-Eisenecker feature modeling).[4]

As has been demonstrated in Sect. 2.4, plural forms of the concepts and cardinality in general can be specified by parameterized concepts without compromising the principles of feature modeling. If preferred, UML cardinalities can be used instead, provided they are defined as a notational extension with respect to the parameterized concept.

4 A Feature Modeling Metamodel

The domain of feature modeling is understood here as a domain of the tools that support feature modeling as a central activity in software development. The feature modeling based methods, such as generative programming, FODA, FORM, FeatuRSEB, and feature modeling for multi-paradigm design, all have in common the central role of a feature model from which traceability links to other models are provided. The variability lies in the notations of feature modeling employed by different methods. The systems built in the domain would represent feature modeling CASE tools suitable for individual methods (possibly groups of methods).

Based on the information presented in the previous sections, a metamodel of the feature modeling will be proposed in this section. The metamodel will be expressed using feature modeling itself in order to capture the variability of feature modeling notations and to describe the core concepts of feature modeling in a concise way. The purpose of this metamodel is to provide a basis both for further reasoning on feature modeling and for developing feature modeling tools. Therefore, the metamodel embraces features that express functionality, too.

The concepts identified in the domain of feature modeling are: feature model, feature diagram, node, feature, partition, associated information, AI item, AI value, constraint, default dependency rule, and link. The model also includes the parameterized concept *Plural Form* introduced in Sect. 2.3, where <*Singular Form*> <*i*>.<*Singular Form*> is a reference to one of the following concepts: *Feature Diagram*, *Node*, *Feature*, *Link*, *Constraint*, *Default Dependency Rule*, or *AI Value*. Dynamic binding of *Plural Form* features is assumed. In the rest

[4] These extensions are implemented in Captain Feature (available at [20]), in which the whole feature modeling notation should be configurable through a metamodel represented by a feature model [23, 18], but its editing is not possible.

of the concepts, dynamic binding is indicated where applies; otherwise, static binding is assumed.

4.1 Feature Model and Feature Diagram

A feature model (Fig. 4) represents the model of a domain obtained by the application of feature modeling. It consists of a set of feature diagrams (*Feature Diagram Set*). and it may have a set of links to other modeling artifacts (*Link Set*). Feature diagrams in a feature model may be normalized [1] (*Normalize*), but this applies only to those feature modeling notations that embrace or-features.

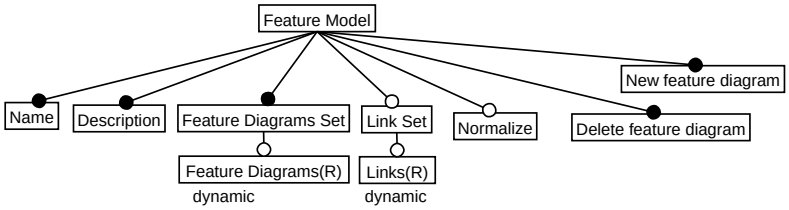


Fig. 4. *Feature Model* concept.

A feature diagram (Fig. 5) presents a featural description of a concept graphically. An additional constraint that applies to *Feature Diagram* is that a root may not be a concept reference:

$\neg$*Root.Node.Reference*

A feature diagram contains a set of nodes (*Node Set*) and a set of features (*Feature Set*). It may be represented by a directed tree (*Tree*). In this case, a feature diagram describes the features of a domain concept represented by its root node (*Root.Node*Ⓡ). An operation of adding a feature to a feature diagram represented as a tree (*Tree.Add feature*) should preserve the tree structure. A feature diagram may also be considered to be a connected directed graph.

A set of constraints (*Constraint Set*) and default dependency rules (*Default Dependency Rule Set*) may be associated with a feature diagram, which is needed by some approaches to feature modeling. Also, a feature diagram may have a set of links to other modeling artifacts (*Link Set*). A feature diagram may be normalized (*Normalize*).

4.2 Node and Feature

Feature diagram nodes (Fig. 6) represent concepts in general sense (as explained in Sect. 2), which have they own names (*Name*), and concept reference nodes (*Reference*). It may have a set of links to other modeling artifacts (*Link Set*). Some approaches to feature modeling allow feature diagram nodes to be marked as open, which means that new direct features of a node are expected (*Openness*).

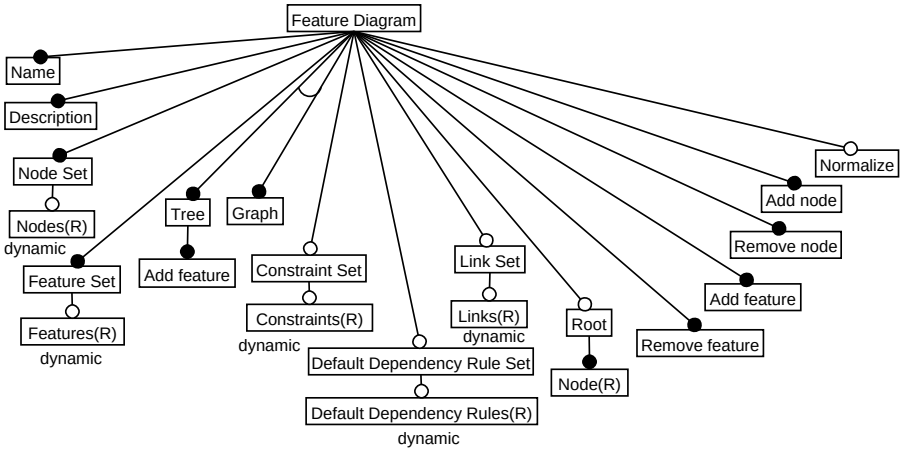


Fig. 5. *Feature Diagram* concept.

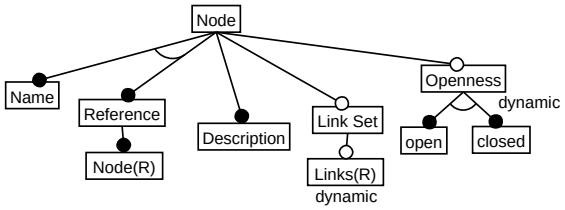


Fig. 6. *Node* concept.

A feature is a relationship between two nodes (Fig. 7). It describes the variability of a subfeature (*Subfeature*) with respect to its superfeature (*Superfeature*): the subfeature may be mandatory (*mandatory*), i.e. it must be included in a concept instance, or it may be optional (*optional*), i.e. it may be included in a concept instance.

In some approaches to feature modeling, relationships between nodes are named (*Name*) or may have a type specified (*Type*). Also, a feature may have a set of links to other modeling artifacts (*Link Set*).

4.3 Partition

Features originating in one node may be divided into a set of disjunct partitions (Fig. 8) marked by arcs in feature diagrams. The features in a partition are presumed to be alternative, i.e. to have xor semantics (as in FODA). Some approaches (e.g., Czarnecki-Eisenecker notation and MPD_{FM}) employ also or-features, so the features in a partition may be either alternative or or-features (*Type*). Other approaches (e.g. [18]) employ cardinality, which enables to specify the number of features (maximum and minimum) in a partition that may be

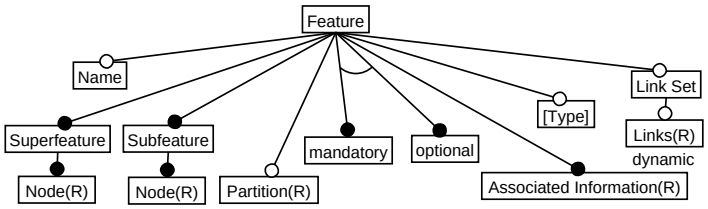


Fig. 7. *Feature* concept.

selected (*Cardinality*). Some approaches to feature modeling allow partitions to be marked as open (similarly to openness of a node in a feature diagram), which means that new direct features in a partition are expected (*Openness*).

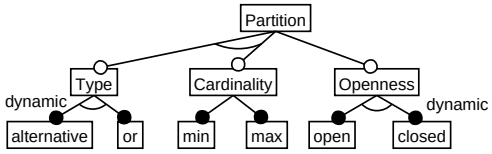


Fig. 8. *Partition* concept.

4.4 Associated Information and Related Concepts

Different approaches to feature modeling, and different applications of it, too, require different information to be associated with features. The concept of associated information (Fig. 9) captures this demand by a fully configurable set of items associated information consists of (*AI Items*®).

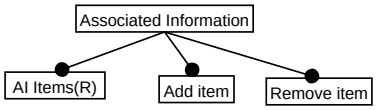


Fig. 9. *Associated Information* concept.

An associated information item (Fig. 10a) applicability may depend on the optionality of a feature with which it is associated (*Applicability*). There are two kinds of an associated information item: textually expressed ones (*Textual*) and those represented by a value selected from the extensible set of available values (*Selectable*). The concept of an associated information value (Fig. 10b) represents such a value.

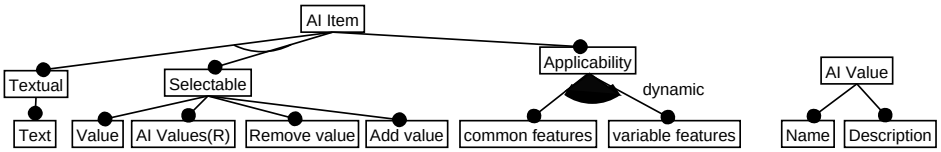


Fig. 10. *AI Item* (a) and *AI Value* (b) concepts.

4.5 Constraint and Default Dependency Rule

Constraints (Fig. 11a) express mutual exclusions and requirements among features beside those specified by the feature diagram. They may be specified either as logical expressions (*Logical expression*), textually (*Textual*), or in a FODA-like form (see Sect. 3.2).

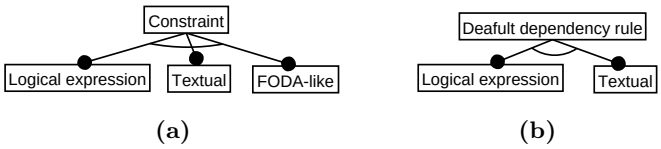


Fig. 11. *Constraint* (a) and *Default Dependency Rule* (b) concepts.

Default dependency rules (Fig. 11b) determine which features should appear together by default in concept instances. They may be specified either as logical expressions (*Logical expression*) or textually (*Textual*).

4.6 Link

A link (Fig. 12) enables to connect a feature model or its parts to its own nodes and features, or to other models. These models include feature models, in which case a link may be more specific and point to a feature diagram in that model, or a node or feature in that diagram. An additional constraint that applies to *Link* concept is that a link may not lead to a node and feature simultaneously:

$$Node \veebar Feature$$

5 Conclusions and Further Research

This paper brings several improvements into feature modeling. Concept instantiation is defined with respect to instantiation time with concept instances represented by feature diagrams. Parameterization in feature models enables to reason

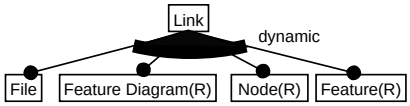


Fig. 12. *Link* concept.

more generally about concepts and features and to express constraints and default dependency rules about subfeatures of an open feature. Constraints and default dependency rules are represented by logical expressions. Concept references enable to deal with complex feature models. A dot convention enables referring to concepts and features unambiguously. A parameterized concept which enables to represent cardinality in feature modeling is introduced.

Other approaches to feature modeling have been evaluated and compared with feature modeling for multi-paradigm design. Based on this analysis, a feature modeling metamodel has been proposed. The metamodel shows how the commonalities and variabilities of the domain of feature modeling may be modeled by feature modeling itself. This metamodel may serve both for further reasoning on feature modeling and as a basis for developing feature modeling tools.

Further research topics include enhancing parameterization in feature modeling with respect to binding time/mode and expressing feature models fully in the form of constraints (as logical expressions) with defined primary constraints that are to be presented visually (in feature diagrams).

Acknowledgements The work was partially supported by Slovak Science Grant Agency VEGA, project No. 1/0162/03.

References

[1] Czarnecki, K., Eisenecker, U.W.: Generative Programing: Principles, Techniques, and Tools. Addison-Wesley (2000)
[2] Coplien, J. O.: Multi-Paradigm Design for C++. Addison-Wesley (1999)
[3] Kang, K.C., et al.: Feature-oriented domain analysis (FODA): A feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, USA (1990).
[4] Kang, K.C., Kim, S., Lee, J., Kim, K., Shin, E., Huh, M.: FORM: A feature-oriented reuse method with domain-specific reference architectures. Annals of Software Engineering 5 (1998) 143–168
[5] Simos, M.A.: Organization domain modeling (ODM): Formalizing the core domain modeling life cycle. In: Proc. of the 1995 Symposium on Software reusability, Seattle, Washington, United States, ACM Press (1995) 196–205
[6] Vranić, V.: Feature modeling based transformational analysis in multi-paradigm design. Submitted to *Computers and Informatics (CAI)*, December 2003.
[7] Object Management Group: OMG unified modeling language specification, version 1.5 (2003).
[8] Chastek, G., et al.: Product line analysis: A practical introduction. Technical Report CMU/SEI-2001-TR-001, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, USA (2001).

[9] Software Engineering Institute, Carnegie Mellon University: A framework for software product line practice — version 3.0. http://www.sei.cmu.edu/plp/framework.html. Last accessed in June 2004.
[10] Griss, M.L., et al.: Integrating feature modeling with the RSEB. In Devanbu, P., Poulin, J., eds.: Proc. of 5th International Conference on Software Reuse, Vicoria, B.C., Canada, IEEE Computer Society Press (1998) 76–85
[11] Geyer, L.: Feature modelling using design spaces. In: Proc. of the 1st German Product Line Workshop (1. Deutscher Software-Produktlinien Workshop, DSPL-1), Kaiserslautern, Germany, IESE (2000)
[12] Riebisch, M., et al.: Extending feature diagrams with UML multiplicities. In: Proc. of the 6th Conference on Integrated Design and Process Technology (IDPT 2002), Pasadena, California, USA, Society for Design and Process Science (2002).
[13] Clauβ, M.: Modeling variability with UML. In: Proc. of Net.ObjectDays 2001, Young Researchers Workshop on Generative and Component-Based Software Engineering, Erfurt, Germany, tranSIT (2001) 226–230
[14] Jia, Y., Gu, Y.: The representation of component semantics: A feature-oriented approach. In Crnković, I., Larsson, S., Stafford, J., eds.: Proc. of the Workshop on Component-based Software Engineering: Composing Systems From Components (a part of 9th IEEE Conference and Workshops on Engineering of Computer-Based Systems), Lund, Sweden (2002).
[15] Riebisch, M.: Towards a more precise definition of feature models. In M. Riebisch, J. O. Coplien, D.S., ed.: Modelling Variability for Object-Oriented Product Lines, Norderstedt, BookOnDemand Publ. Co. (2003) 64–76
[16] Streitferdt, D., et al.: Details of formalized relations in feature models using OCL. In: Proc. of the 10th IEEE Symposium and Workshops on Engineering of Computer-Based Systems (ECBS'03), Pasadena, California, USA, IEEE Computer Society (2003) 297–304
[17] Lee, K., et al.: Concepts and guidelines of feature modeling for product line software engineering. In Gacek, C., ed.: Proc. of 7th International Conference (ICSR-7). LNCS 2319, Austin, Texas, USA, Springer (2002)
[18] Czarnecki, K., et al.: Generative programming for embedded software: An industrial experience report. In Batory, D., et al., eds.: Generative Programming and Component Engineering: ACM SIGPLAN/SIGSOFT Conference, GPCE 2002. LNCS 2487, Pittsburgh, PA, USA (2002) 156—172
[19] Czarnecki, K., Eisenecker, U.W.: Generative programming — methods, tools, and applications. http://www.generative-programming.org. Last accessed in March 2004.
[20] Captain Feature: Project page. https://sourceforge.net/projects/captainfeature. Last accessed in March 2004.
[21] Blinn, F.: Entwurf und implementierung eines generators für merkmalmetamodelle. Master's thesis, Fachhochschule Zweibrücken, Fachbereich Informatik (2001) In German. Available at http://www.informatik.fh-kl.de/~eisenecker (last accessed in March 2004).
[22] Czarnecki, K., et al.: Generative programing: Methods, techniques, and applications. Slides and notes of the tutorial given at Net.ObjectDays 2003 (2003)
[23] Bednasch, T.: Konzept und implementierung eines konfigurierbaren metamodells für die merkmalmodellierung. Master's thesis, Fachhochschule Kaiserslautern, Standort Zweibrücken, Fachbereich Informatik (2002) In German. Available at http://www.informatik.fh-kl.de/~eisenecker (last accessed in March 2004).

Appendix E

Binding Time Based Concept Instantiation in Feature Modeling

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.

Binding Time Based Concept Instantiation in Feature Modeling

Valentino Vranić and Miloslav Šípka

Institute of Informatics and Software Engineering
Faculty of Informatics and Information Technology
Slovak University of Technology, Ilkovičova 3, 84216 Bratislava 4, Slovakia
vranic@fiit.stuba.sk, miloslav.sipka@gmail.com

Abstract. In this paper, we address the issue of concept instantiation in feature modeling with respect to binding time. We explain the impact of such instantiation on applying constraints among features expressed in feature diagrams and as additional constraints and propose a way to validate a concept instance under these new conditions.

1 Introduction

Feature modeling aims at expressing concepts by their features as important properties of concepts taking into account feature interdependencies and variability in order to capture the concept configurability [3]. A concept is an understanding of a class or category of elements in a domain [3]. Individual elements that correspond to this understanding are called *concept instances*. While a concept represent a whole class of systems or parts of a system, an instance represents a specific configuration of a system or a part of a system defined by a set of features. Concept instances may be used for feature model validation and manual or automatic configuration of other design models or program code of specific products in a domain [2, 3].

When designing a family of systems, we have to balance between statically and dynamically bound features. In general, dynamic binding is more flexible as we may reconfigure our system at run time, while static binding is more efficient in terms of time and space. Although they often embrace the information on feature binding time, contemporary approaches to feature modeling do not consider the time dimension during concept instantiation.

This paper focuses on the issue of concept instantiation (Sect. 2) and validation of concept instances with respect to binding time (Sect. 3). The paper is closed by a discussion (Sect. 4).

2 Concept Instantiation in Time

Binding time describes *when* a variable feature is to be *bound*, i.e. selected to become a mandatory part of a concept instance. The set of possible binding times

depend on a solution domain. For compiled languages they usually include source time, compile time, link time, and run time [1].

An instance I of the concept C at time t is a concept derived from C by selecting its features which includes the C's concept node and in which each feature f whose parent is included in I obeys the following conditions:

1. If f is a mandatory feature, f is included in I.
2. If f is a variable feature whose binding time is earlier than or equal to t, f is included in I or excluded from it according to the constraints of the feature diagram and additional constraints associated with it. If included, the feature becomes mandatory for I.
3. If f is a variable features whose binding time is later than t, f may be included in I as a variable feature or excluded from it, or the constraints (both feature diagram and additional ones) on f may be made more rigid as long as the set of concept instances available at later instantiation times is preserved or reduced.

As follows from this definition,[1] a feature in a concept instance may be bound, in which case it appears as a mandatory feature, or unbound, in which case it stays variable. Mandatory features and features bound in previous instantiations are considered as bound. A concept instance may be instantiated further at later instantiation times.

The constraints—both feature diagram and additional ones—on a variable features whose binding time is later than the instantiation time may be made more rigid as long as the set of concept instances available at later instantiation times is preserved or reduced. An example of this is a transformation of a group of mandatory or-features (Fig. 1a) into a group of alternative features (Fig. 1b).

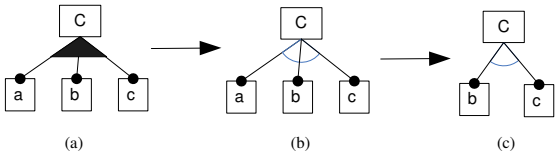


Fig. 1. Reducing the set of concept instances.

Variable features with binding times later than the instantiation time are potentially part of concept instances at later binding times. Again, such features may be excluded at instantiation times earlier than their binding times as long as the set of concept instances available at later instantiation times is preserved or reduced. Consider a group of three alternative features (Fig. 1b) with run-time binding. At source time, one of these features may be excluded (Fig. 1d). However, none of the two remaining features may be excluded since preserving

[1] The definition is based on our earlier concept instance definition [7].

only one of them will force us to make it mandatory, which is illegal, or optional, which will allow an originally unforeseen concept instance to be created: the one with no features from the group.

3 Concept Instance Validation

A concept instance is valid if its features satisfy the constraints. In general, a constraint—be it a feature diagram constraint or an additional one— may be evaluated only if all the features it refers to are bound. However, some logical expressions can be evaluated without knowing the values of all of their variables. Suppose we are instantiating a simple concept in Fig. 2a at source time (with no additional constraints). If we bind the x feature, the or-group constraint will be satisfied regardless of the y feature binding. Thus, we may omit this constraint transforming the y feature into an optional one as shown in Fig. 2b.

It is also possible to omit x. The only possibility for y is to leave it optional, as shown in Fig. 2c, but it has to be assured it will finally be bound (which can be done only at run time). For this purpose, we must add a trivial constraint to this instance: y (y has to be true, i.e. bound).

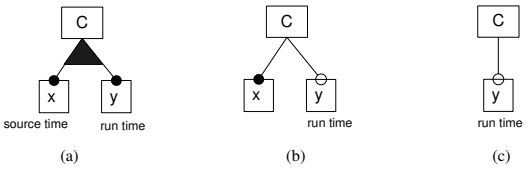


Fig. 2. Dealing with features whose binding time is later than the instantiation time.

By excluding features from feature diagrams, the feature diagram constraints are gradually relinquished. After a successful concept instance validation, all additional constraints that refer to the features whose binding time is not later than the instantiation time can be safely removed from the model. All other constraints have to be postponed for further instantiation.

4 Discussion

In this paper, we presented an approach to concept instantiation with respect to binding time. We analyzed the impact of introducing the time dimension into concept instantiation on concept instance validation with respect to both feature diagram and additional constraints. We have also developed a prototype tool that supports such instantiation (available at `http://www.fiit.stuba.sk/~vranic/fm/`).

Concept instantiation with respect to feature binding time is similar to staged configuration of feature models proposed in conjunction with cardinality-based

feature modeling [5, 6]. Although consecutive work [4] mentions a possibility of defining configuration stages in terms of the time dimension, this approach does not elaborate the issue of feature binding time with respective consequences on validation of concept specializations.

Concept instantiation with respect to binding time can be used to check for "dead-end" instances that may result into invalid configurations of a running system. Such configuration may miss some features required by other, bound features, which will lead to a system crash if such features are activated. Similarly as staged feature model configuration, concept instantiation with respect to binding time could be used for creating specialized versions of frameworks [5], which would represent a source time instantiation, and in software supply chains, optimization, and policy standards [4].

Partial validation of the constraints that incorporate unbound features may be improved by transforming them into the normal conjunctive form. This would enable to extract parts of such a constraint with bound features, while conjuncts with unbound features would be simple enough to directly determine whether they can be evaluated or not. As a further work, we plan to explore consequences of applying this approach to cardinality-based feature models [5].

Acknowledgements The work was supported by Slovak Science Grant Agency VEGA, project No. 1/3102/06, and Science and Technology Assistance Agency of Slovak Republic under the contract No. APVT-20-007104.

References

[1] James O. Coplien. *Multi-Paradigm Design for C++*. Addison-Wesley, 1999.

[2] Krzysztof Czarnecki and Michal Antkiewicz. Mapping features to models: A template approach based on superimposed variants. In Robert Glück and Michael R. Lowry, editors, *Proc. of Generative Programming and Component Engineering, 4th International Conference, GPCE 2005*, LNCS 3676, pages 422–437, Tallinn, Estonia, October 2005. Springer.

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

[4] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10:7–29, January/March 2005.

[5] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged configuration through specialization and multi-level configuration of feature models. *Software Process: Improvement and Practice*, 10:143–169, April/June 2005.

[6] Krzysztof Czarnecki and Chang Hwan Peter Kim. Cardinality-based feature modeling and constraints: A progress report. In *International Workshop on Software Factories, OOPSLA 2005*, San Diego, USA, October 2005.

[7] 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.

Appendix F

Representing Change by Aspect

Peter Dolog, Valentino Vranić, and Mária Bieliková. Representing change by aspect. *ACM SIGPLAN Notices*, 36(12):77–83, December 2001.

Representing Change by Aspect*

Peter Dolog, Valentino Vranić and Mária Bieliková
Dept. of Computer Science and Engineering
Faculty of Electrical Engineering and Information Technology
Slovak University of Technology
Ilkovičova 3, 812 19 Bratislava, Slovakia,
{dolog,vranic,bielik}@elf.stuba.sk
http://www.dcs.elf.stuba.sk/~{dologp,vranic,bielik}

Abstract. We propose the application of aspect-oriented programming to software configuration management. We believe it could improve the change control by providing a new basis for reasoning about a change. To demonstrate this, we designed an abstract-oriented extension to procedural languages where a change is represented by an aspect. Consequently, a change gains the properties of an aspect: it becomes well-localized and separated from the (unchanged) base program. This goes beyond the current capabilities of configuration management methods and tools: the aspect representing the change can be applied to other versions of the program (possibly to different programs).

Keywords: aspect-oriented programming, change control, change representation.

1 Introduction

Software systems are developed and evolved in a series of changes. Changes arise as requirements are extended, reformulated, dropped or corrected, as faults are discovered, and in many other situations. The change is often required also due to a need for adapting the product to the user's context. We are witnesses of growing cooperation among software development companies. Many (often distributed) teams work on the same release of the software system in parallel. In such a situation, change control becomes even more important.

The level of change control support provided by the existing software configuration management tools varies significantly. Hence, if two companies decide to cooperate, there is a big chance that they would have different tools that provide different repository items representation, different structure representation, etc. The companies can also have different configuration management process established, including, for example, different branching and merging strategy, what even more complicates keeping track of changes in the source code.

We propose a solution to some of these problems by treating a change at the source code level and by expressing it explicitly. To achieve this, we employ the aspect-oriented programming, a new approach to programming aiming at separation of crosscutting concerns (see Section 3).

The rest of the paper is structured as follows. First, we put our approach in the context of the existing versioning models (Section 2) and the aspect-oriented programming (Section 3). Then we present an abstract aspect-oriented extension to procedural languages (Section 4). Subsequently, we show how this extension can be concretized to VBScript language (Section 5). Finally, we draw some conclusions and point some directions for the further work (Section 6).

2 Version Models and Change

A version model defines the entities to be versioned, version identification and organization, as well as operations for retrieving existing versions and constructing new versions [4]. Several version models are described in the literature and used in existing configuration management tools. We focus on the core issue of versioning, namely the organization of version space, or to be more specific—the version description and representation. Our main interest is to improve change control.

According to the entities being handled, the version models are classified into state-based and change-based. *State-based models* focus on the states of versioned items. In such approach, versions are usually described in terms of revisions and variants [2]. A configuration item (the smallest unit of a system taken under version control) is maintained usually at the file level. The change in state-

*This work was partially supported by Slovak Science Grant Agency, grant No. G1/7611/20.

based models can be described as the difference between two versions. Many commercial systems are state-based (e.g., Microsoft Visual Source Safe, Rational ClearCase, PVCS) [3].

The problem with state-based models is that a change is maintained implicitly, during the modification of a branch. Merging can be viewed as a re-application of all the changes to the branches being merged. This requires the extraction of the changes from the branches and their subsequent application to the base.

In *change-based models* the change is treated as a first class entity and managed explicitly by a developer, either manually or by a tool. A version is considered as the result of the application of changes to a baseline. There are several commercial change-based software configuration management systems that have the ability to track the logical changes rather than individual file changes (e.g., Continuus/CM, CCC/Harvest). They treat the change at the level of the source code lines or at the level of the file versions. Accordingly, they allow to create change sets or change packages [13]. A *change set* consists of the changed code lines. A *change package* contains references to the file versions that are the compositions of logical changes.

In a change-set model, changes are combined freely to construct new versions according to the requirements. In [4] such approach is denoted as *change-based intensional versioning*. The use of change packages is denoted as *change-based extensional versioning*, because version set is defined explicitly by enumerating its members. In this case, each version is described by changes relative to some baseline.

Another change-based approach is based on change identification by language constructs. This can be denoted as *language-aware* approach: the change is handled by directives for source code inserting, deleting and editing augmented with the attributes of the change (e.g., who and when made the change, etc.). An example of this approach is VTML (Versioned Text Markup Language) [9] or conditional compilation.

The conditional compilation enables to use the preprocessor directives to control the code fragments visibilities. In this case, all changes (fragments) are stored in one file, which is hard to maintain. Management of fragments' visibilities is necessary for improving change control. This approach is used, for example, in the EPOS system [5].

The change-based systems are not so widely used as the state-based ones. The main reason is that developers think rather at the version-state level. However, nowadays many state-based systems are being extended to provide the change-based functionality (e.g., Rational ClearCase) [8]. The objective is to improve change management and traceability of the change request in a software development process.

The change representation influences the change control procedure, which consists of the four major steps [1]: checking whether the change is needed,[1] analysis of causes that led to change, planning the change, and change implementation. In the context of the change control procedure, we are concerned with the change implementation.

The problem with the surveyed change-based approaches is the granularity of the logical change. As we mentioned, some of them treat the logical change as the individual lines of the source code, while other are based on representing change by preprocessor directives.

3 Aspect-Oriented Extensions

The main idea of the aspect-oriented programming (AOP)—separation of concerns by separating the crosscutting concerns called *aspects* from the basic functionality crosscut by them—is carried throughout several independently developed approaches [10, 12]. Among them, Xerox PARC AOP [14] holds a significant position. Further in the text by the AOP we mean actually the Xerox PARC AOP.

AOP appeared as a reaction to the problem known from the *generalized procedure languages* [7], i.e. programming languages that use the concept of the procedure to capture the functionality.[2] In such languages the program code fragments that implement a clearly separable aspect of a system (such as synchronization) are scattered and repeated throughout the overall program code that, in advance, becomes *tangled*. AOP aims at factoring out such aspects into separate program units called by the same name: *aspects*. Aspects *crosscut* the *base* code in places called *join points*. These must be specified so aspects could be *woven* into the base code by the program called *weaver*.

The join points can be static or dynamic. *Static join points* can be identified in the program text itself. They can be specified in terms of a programing language syntax alone. An example of such a join point is the beginning or end of a method or procedure body. *Dynamic join points* are available at run time only. For example, a method reception by an object is a dynamic join point. In the weaving process, the static join points are resolved by a simple program code insertion, while dynamic join points can be resolved at run time only.

The special language constructs used to capture the aspects and join points are known as the *aspect-oriented extension* of the base language. The two types of aspect-oriented extension regarding its relationship to the base

[1] It is possible that some workaround for the existing activity could be more effective than the change itself.

[2] Besides the procedural languages, these include functional and object-oriented languages as well.

language can be distinguished: homogenous and heterogeneous. The homogenous extension, besides for some additional constructs, relies on the base language to the greatest possible extent, while the heterogeneous extension introduces a whole new language for capturing the aspect-oriented part of the program. In general, there can be several independent aspect-oriented extensions, handled by the same or by separate weavers.

Not unlike programming languages in general, an aspect-oriented extension (including the corresponding weaver) can be designed to solve a specific problem, such as the one presented in [7] (the filtering example), or to serve a general-purpose, as the AspectJ language [15], which is a homogenous, general-purpose aspect-oriented extension to Java. While aspect-oriented extensions provide a new way of programming, they do so only in the context of the language they extend. In other words, AOP is a multi-paradigm approach in its very nature [12], and AspectJ can be viewed as a multi-paradigm language [11].

4 Aspect-Oriented Extension for Change Representation

As it was discussed in Section 2, current configuration management approaches do not offer a satisfactory change representation regarding the change maintenance and re-applicability to different branches. The use of AOP enables to maintain changes explicitly by capturing a change into an aspect.

In order to enable change representation by aspect, the aspect-oriented extension to a given programming language should be provided. Since the changes are actually changes of the program text, all the join points will be statical. Further, the aspect-oriented extension should be *homogenous*—to preserve the base language constructs, and *general-purpose*—to cover all the types of changes (which depend on the base language). Also, the join point description should not affect the base program.

To illustrate aspect-oriented approach to change representation, we developed an aspect-oriented extension (inspired by AspectJ) to procedural languages. Proposed language constructs are presented in Fig. 1. Different type styles are used to distinguish among the `keywords`, **required parts** and *optional parts*.

The aspects are placed into modules, possibly together with the ordinary procedures which can be called from within the aspects, i.e. inside of the *block* parts. The *block* parts must be parsed either by the weaver, or by the original language parser.

The introductions are used to introduce new procedures and variables into modules (M_i). The advices enable performing a command *block* before, after or in place of the procedures determined by a specified set of join points, so-called *pointcut*. While `before` and `after` advices are simple, the `around` advice requires some explanation. It enables to run an initial block $block_i$, then to `proceed` with the next action, which is either another aspect, or the original procedure body, in case there is no other aspect affecting the procedure. The optional *return_clause* in `after` and `around` advices enables to modify the return value (if the procedure returns one) before it is actually returned to the caller.

The **pointcut_specification** is built out of the pointcut primitives (listed in the bottom of Fig. 1) using the logical operators *and* and *or*.[3] The parentheses can be used to declare the priority of subexpressions evaluation. The first two primitive pointcuts, `modules` and `withincode`, designate all the join points within the modules M_i and procedures specified by the **procedure_signature**, respectively. The `calls` pointcut primitive designates all the procedure calls specified by the **procedure_signature**. The `definitions` pointcut primitive designates the actual definitions, i.e. bodies of the procedures specified by the **procedure_signature** (see Fig. 5 for an example). A `before` advice to a `definitions` pointcut will insert its code *after* all the declarations of variables in the specified procedure(s) placed before the first non-declaration statement.

The wild cards `*` and `..` can be used in **procedure_signature** to denote any string of characters and omitted arguments, respectively. This convention is used in AspectJ, e.g. `* p*(int, *)` denotes all the methods whose name starts with `p`, with one `int` argument and one argument of any type, returning a value of any type. The most general signature—denoting all the methods—is then `* *(..)`.

Up to now we said nothing about the optional *argument_list* in advices. It is used to access the arguments of the procedures denoted by the pointcut. Suppose we want to make a `before` advice to the following C function:

```
int f(int i) {return i*i;}
```

Consider these two advices:

1. `before(): definitions(int f(int)) {i = i + 1;}`
2. `before(int x): definitions(int f(x))`
   `{x = x + 1;}`

Both advices seem to do the same thing; they add one to `f`'s argument before proceeding with the rest of `f`'s body.[4] However, if we rename `i` in function `f` to `j`, the first advice will fail to satisfy our intention (moreover, it will produce a syntax error), so the second version is obviously more robust.

[3] Since pointcuts are the sets of join points, the *and* and *or* operators have the meaning of set intersection and union, respectively.

[4] This is different from AspectJ where the method body is not visible to advices.

Introductions:

`introduction` $\mathbf{M}_1,\ldots,\mathbf{M}_n$ $\{block\}$

Advices:

`before(`*argument_list*`):` **pointcut** $\{block$

`after(`*argument_list*`):` **pointcut** $\{block$ *return_clause*$\}$

`around(`*argument_list*`):` **pointcut** $\{block_i$ `proceed...` $block_f$ *return_clause*$\}$

Pointcuts:

`pointcut` **pointcutName** `(`*argument_list*`):` **pointcut_specification**

Pointcut primitives:

`modules(`$\mathbf{M}_1,\ldots,\mathbf{M}_n$`)`

`withincode(`**procedure_signature**`)`

`calls(`**procedure_signature**`)`

`definitions(`**procedure_signature**`)`

Figure 1: Aspect-oriented extension to procedural languages.

The proposed aspect-oriented extension is capable of describing the following types of changes:

- introduction of a new procedure or (global) variable into the module;
- extension of a procedure by a code before, after, or instead of it;
- change to the procedure arguments and return value.

What all of these changes have in common is that they are all about the functionality. The changes that cannot be described at the level of functionality are very hard (or impossible) to deal with using the aspect-oriented approach. These include renaming a procedure or variable, adding a white space or comment, changing the position of a procedure in the source code, etc.

A version is obtained by weaving the aspects that capture the change into the base program. Since this version might become a subject of modification as well, it should be human readable. This is different from the AOP itself where the process of weaving yields only an intermediate product not intended to be read by a human. The aspect-oriented extension proposed principally satisfies this requirement, since it relies on static join points only.

5 Case Study: Script Customization

We will show now how the approach we proposed in the preceding section could help in solving a problem of synchronizing the local customization with the global version of a program in script languages by the means of an example. We will use VBScript-like syntax since VBscript is widely used as a language for dynamical content generation and design of the web pages. It is the core language for Microsoft's `asp` pages. Further, some software houses use VBScript as the language for customizing their products, e.g. InteractCommerce corp.'s SalesLogix.

Suppose that two teams work on one system. The teams received change requests regarding the same script, shown in Fig. 2, which is a part of the system, at the same time, i.e. before synchronization of branches, as depicted in Fig. 3. The purpose of the script is to extract the list of sales opportunities from the `opportunity` table in the `test` database. The change request received by the first team is about extending the list of opportunities by the list of products. A new recordset, as well as the SQL statement and several lines of VBScript code must be added in order to accomplish the task of extracting the records from the table and generating the page containing the data.

The modified script is presented in Fig. 4; some commands the same as in Fig. 2 have been omitted (indicated by ellipsis). The code between the `change` and `end change` comments can be separated into the aspect module, as presented in Fig. 5. The affected module is specified by the `modules` designator in both advices. The declarations of additional variables are provided in the `before` advice. The conjunction of the `definitions` and `modules` designator states that the sequence of variable declarations in the advice is to be merged *after* all the declarations in the `main` procedure which are placed before the first non-declaration statement. The sequence of the directives to be run after the `ro.close` method in-

```
sub main
  Dim con 'Connection object
  Dim s 'Select statement
  Dim ro 'Recordset object
  Set con = Server.CreateObject("ADODB.Connection")
  con.Open "Test"
  s = "SELECT * FROM opportunity"
  Set ro = con.Execute(s)
  call gener_data(ro)
  ro.Close
  con.Close
  Set ro = Nothing
  Set con = Nothing
end sub
```

Figure 2: The code base in VBScript.

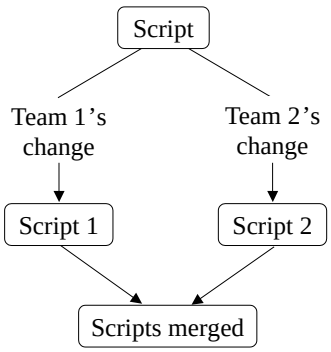


Figure 3: The branching.

vocation (specified by the `calls` designator) within the original `main` procedure (specified by the `withincode` designator) has been enclosed into the `after` advice. The result of the merging, i.e. weaving, will be the same code as displayed in Fig. 4.

The other change request addressed to the second team resulted in the script shown in Fig. 6. The second team's change consists of adding the `while` loop for updating the `applied` and the `date_of_application` fields for each record in the opportunity table, and of adding the sequence of commands that generate the list of marketing campaigns from the `marketing` table. We can apply the aspect from Fig. 5 to the code in Fig. 6 without any change.

However, that change could have been separated into the aspect, too, if both teams used the aspect-oriented approach. In that case, we would simply apply the two aspects subsequently in order to obtain both functionalities.[5]

[5]The priority of aspects is not significant here, but this is not so in general.

```
sub main
  . . .
  '***change of declarations***
  Dim rp 'Recordset object
  Dim s2 'Select statement
  Dim c 'Command object
  '***end of change***
  Set con =  . . .
  . . .
  ro.Close
  '***change***
  s2 = "SELECT * FROM product"
  Set c = Server.CreateObject("ADODB.Command")
  c.ActiveConnection = con
  c.CommandText = s2
  Set rp = c.Execute
  call gener_data(rp)
  rp.close
  rp = Nothing
  c = Nothing
  '***end of change***
  . . .
end sub
```

Figure 4: The change performed by the first team.

6 Conclusions

We proposed a new approach to change-oriented versioning based on the aspect-oriented programming. The contribution of this paper is the proposal of the technique aimed to simplify change control by reifying the changes into language-level entities: a change is represented by an aspect and maintained explicitly by a developer.

A homogenous, general aspect-oriented extension has to be provided for a given programming language first. For the purposes of our approach, it is sufficient if this extension supports static join points. Since procedural, functional and object-oriented languages are easily extended to support the AOP with static join points, this approach is low-cost. We proposed such an extension to procedural languages. Moreover, it can be expected that general aspect-oriented extensions to other programming languages will be developed and provided for the sake of the AOP itself, so no additional effort would be necessary to employ this approach in such languages. This can be denoted as *self-supported change management*: a change is represented by the constructs that are a part of the programming language itself.

We assume this as one of the main advantages of the proposed version space representation. It provides a new base level for the change control; it is a move from the change control at the line level to the one at the programming language semantics level. In small software projects it is directly usable even without a software configuration management tool. The change comprehension and ori-

```
before(): modules(script) && definitions(main)
  begin
  Dim rp 'Recordset object
  Dim s2 'Select statement
  Dim c 'Command object
end before

after(): modules(script) && withincode(main)
         && calls(ro.close)
begin
  s2 = "SELECT * FROM opportunity_product"
  Set c = Server.CreateObject("ADODB.Command")
  c.ActiveConnection = con
  c.CommandText = s2
  Set rp = c.Execute
  call gener_data(rp)
  rp.close
  rp = Nothing
  c = Nothing
end after
```

Figure 5: The change separated into the aspect.

entation in the source code is easier because the change is well-localized in the aspect and need not be searched for. A change is possibly re-applicable as is or with some adaptation of the aspect involved (white-box reuse). Actually, the aspect can be applied to a completely different module than it was intended for by a simple modification of the pointcut.

Aspect-oriented approach can be used also in post-deployment configuration management [6] for parametrization (modification of a software system to take into account the local site context). The local context can be represented by an aspect. The application of the relevant aspects provides customization of the new product version according to the local context (developed for the previous version). Obviously, new aspects will be also created, in order to customize new features in the current version of the product.

Our approach can be used with existing software configuration management tools. Moreover, our approach is independent of the model employed by software configuration management tools. An aspect is a separate item, so it can be handled in both basic version models (state-based and change-based). It also supports the implicit long transaction maintenance because aspect itself represents a change and it is up to the developer to decide when the change should be committed. As aspects can be simply plugged in or out before the compilation, adding of an individual change into a version or substracting a change from a version (similarly as in the change-set approaches) is simple. Even more, the aspects can be combined into change packages. A change request could be then directly assigned to the corresponding aspect or change package

```
sub main
  . . .
  '***change of declarations***
  Dim rm 'Recordset object
  Dim com 'Command object
  Dim str 'String - select statement
  '***end of change declarations***
  Set con =  . . .
  . . .
  call gener_data(ro)
  '***change***
  While Not ro.EOF
    ro.Fields("applied") = 'T'
    ro.Fields("date_of_application") = Now
  Loop
  str = "SELECT * FROM marketing"
  Set com = Server.CreateObject("ADODB.Command")
  com.ActiveConnection = con
  com.CommandText = str
  Set rm = com.Execute
  call gener_data(rm)
  rm.close
  rm = Nothing
  com = Nothing
  '***end of change ***
  ro.Close
  . . .
end sub
```

Figure 6: The change performed by the second team.

(indicated by the appropriate identifiers).

Our work is now oriented toward a deeper elaboration of practical use of the proposed approach. Some additional mechanisms should be added to manipulate version history and versions themselves. In order to be able to control the changes effectively, some meta-data should be stored within each change (e.g., who and when made the change). In order to follow a version history, the meta-data related to changes should be processed and interpreted.

On the other hand, we are already able to partially track the version history, but it is difficult to determine which version was created by a developer and which is just a potential version when storing only changes. The potential versions can be obtained by applying the combinations of the aspects. However, not all potential versions make sense [4].

An additional problem is that the aspect representing a change can become a subject of change, too. As a consequence, a method of dealing with the change of a change should be proposed. This problem arises in any change-based version model, of course, but a special method is needed here because our approach works at other level of change control than traditional change-based version models.

References

[1] George W. Allan. *An Holistic Model for Change Control*, pages 703–707. Plenum, New York, 1997. Available at `http://www.dis.port.ac.uk/~allangw/chng-man.htm`. Accessed on March 6, 2001.

[2] M. Bieliková and P. Návrat. An approach to automated building of software system configurations. *Int. Journal of Software Engineering and Knowledge Engineering*, 9(1):73–95, 1999.

[3] Jim Buffenbarger and Kirk Gruell. A branching/merging strategy for parallel software development. In Jacky Estublier, editor, *Proc. of 9th Int. Symposium on System Configuration Management*, pages 86–99, Tolouse, France, September 1999. Springer LNCS 1675.

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

[5] Bjorn Gulla, Even-André Karlsson, and Dashing Yeh. Change-oriented version descriptions in EPOS. *Software Engineering Journal*, 6(6):378–386, November 1991.

[6] Dennis Heimbigner and Alexander L. Wolf. Post-deployment configuration management. In Ian Sommerville, editor, *Proc. of 6th Int. Workshop on Software Configuration Management*, pages 272–276, Berlin, Germany, March 1996. Springer LNCS 1167.

[7] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Christina Vidiera Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proc. of 11th European Conf. on Object-Oriented Programming (ECOOP'97)*, Jyväskylä, Finland, June 1997. Springer LNCS 1241. Available at [15].

[8] David B. Leblang. Managing the software development process with ClearGuide. In Reidar Conradi, editor, *Proc. of 7th Int. Workshop on Software Configuration Management*, pages 66–80, Boston, USA, May 1997. Springer LNCS 1235.

[9] Fabio Vitali and David G. Durand. Using versioning to support collaboration on the WWW. In *Proc. of 4th World Wide Web Conference*, 1995. Available at `http://www.w3.org/pub/Conferences/WWW4`. Accessed on March 6, 2001.

[10] Valentino Vranić. Multiple software development paradigms and multi-paradigm software development. In J. Zendulka, editor, *Proc. of the Information Systems Modelling 2000*, pages 191–196, Rožnov pod Radhoštěm, Czech Republic, May 2000. MARQ.

[11] Valentino Vranić. AspectJ paradigm model: A basis for multi-paradigm design for AspectJ. In *Proc. of Third International Conference on Generative and Component-Based Software Engineering (GCSE 2001)*, Erfurt, Germany, September 2001. Springer. Accepted for publishing.

[12] Valentino Vranić. Towards multi-pradigm software development. Submitted to Journal of Computing and Information Technology (CIT), 2001.

[13] Darcy Wiborg Weber. Change sets versus change packages: Comparing implementation of change-based SCM. In Reidar Conradi, editor, *Proc. of 7th Int. Workshop on Software Configuration Management*, pages 25–35, Boston, USA, May 1997. Springer LNCS 1235.

[14] Xerox PARC. Aspect-Oriented Programming home page. `http://www.parc.xerox.com/aop`. Accessed on July 11, 2001.

[15] Xerox PARC. AspectJ home page. `http://aspectj.org`. Accessed on July 11, 2001.

Peter Dolog received his Bc. (BSc.) in 1998, and his Ing. (MSc.) in 2000, both in information technology, and both from Slovak University of Technology in Bratislava. Since 2000 he is a PhD student at the Department of Computer Science and Engineering, Faculty of Electrical Engineering and Information Technology of Slovak University of Technology in Bratislava. His research interests include hypermedia systems modelling, adaptive presentation of information in the Internet, and new approaches to software engineering in general. He is a member of the Slovak Society for Computer Science.

Valentino Vranić received his Bc. (BSc.) in 1997, and his Ing. (MSc.) in 1999, both in information technology, and both from Slovak University of Technology in Bratislava. Since 1999 he is a PhD student at the Department of Computer Science and Engineering, Faculty of Electrical Engineering and Information Technology of Slovak University of Technology in Bratislava. His main research interests are multi-paradigm software development and aspect-oriented programming. He is a member of the Slovak Society for Computer Science.

Mária Bieliková received her Ing. (MSc.) in 1989 from Slovak University of Technology in Bratislava, and her CSc. (PhD.) degree in 1995 from the same university. Since 1998, she is an associate professor at the Department of Computer Science and Engineering at Slovak University of Technology. Her research interests include knowledge software engineering, software development and management of versions and software configurations, adaptive hypermedia and educational systems. She is a member of the Slovak Society for Computer Science, IEE, ACM, IEEE and its Computer Society.

Appendix G

Evolution of Web Applications with Aspect-Oriented Design Patterns

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.

Evolution of Web Applications with Aspect-Oriented Design Patterns

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

[1] Institute of Informatics and Software Engineering
Faculty of Informatics and Information Technology
Slovak University of Technology,
Ilkovičova 3, 84216 Bratislava 4, Slovakia
bebjak02@student.fiit.stuba.sk, vranic@fiit.stuba.sk

[2] Department of Computer Science
Aalborg University
Fredrik Bajers Vej 7, building E, DK-9220 Aalborg EAST, Denmark
dolog@cs.aau.dk

Abstract. It is more convenient to talk about changes in a domain-specific way than to formulate them at the programming construct level or—even worse—purely lexical level. Using aspect-oriented programming, changes can be modularized and made reapplicable. In this paper, selected change types in web applications are analyzed. They are expressed in terms of general change types which, in turn, are implemented using aspect-oriented programming. Some of general change types match aspect-oriented design patterns or their combinations.

1 Introduction

Changes are inseparable part of software evolution. Changes take place in the process of development as well as during software maintenance. Huge costs and low speed of implementation are characteristic to change implementation. Often, change implementation implies a redesign of the whole application. The necessity of improving the software adaptability is fairly evident.

Changes are usually specified as alterations of the base application behavior. Sometimes, we need to revert a change, which would be best done if it was expressed in a pluggable way. Another benefit of change pluggability is apparent if it has to be reapplied. However, it is impossible to have a change implemented to fit any context, but it would be sufficiently helpful if a change could be extracted and applied to another version of the same base application. Such a pluggability can be achieved by representing changes as aspects [5]. Some changes appear as real crosscutting concerns in the sense of affecting many places in the code, which is yet another reason for expressing them as aspects.

This would be especially useful in the customization of web applications. Typically, a general web application is adapted to a certain context by a series of changes. With arrival of a new version of the base application all these changes

have to be applied to it. In many occasions, the difference between the new and the old application does not affect the structure of changes.

A successful application of aspect-oriented programming requires a structured base application. Well structured web applications are usually based on the Model-View-Controller (MVC) pattern with three distinguishable layers: model layer, presentation layer, and persistence layer.

The rest of the paper is organized as follows. Section 2 establishes a scenario of changes in the process of adapting affiliate tracking software used throughout the paper. Section 3 proposes aspect-oriented program schemes and patterns that can be used to realize these changes. Section 4 identifies several interesting change types in this scenario applicable to the whole range of web applications. Section 5 envisions an aspect-oriented change realization framework and puts the identified change types into the context of it. Section 6 discusses related work. Section 7 presents conclusions and directions of further work.

2 Adapting Affiliate Tracking Software: A Change Scenario

To illustrate our approach, we will employ a scenario of a web application throughout the rest of the paper which undergoes a lively evolution: affiliate tracking software. Affiliate tracking software is used to support the so-called affiliate marketing [6], a method of advertising web businesses (merchants) at third party web sites. The owners of the advertising web sites are called affiliates. They are being rewarded for each visitor, subscriber, sale, and so on. Therefore, the main functions of such affiliate tracking software is to maintain affiliates, compensation schemes for affiliates, and integration of the advertising campaigns and associated scripts with the affiliates web sites.

In a simplified schema of affiliate marketing a customer visits an affiliate's page which refers him to the merchant page. When he buys something from the merchant, the provision is given to the affiliate who referred the sale. A general affiliate tracking 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.

Suppose such a general affiliate tracking software is bought by a merchant who runs an online music shop. The general affiliate software has to be adapted through a series of changes. We assume the affiliate tracking software is prepared to the integration with the shopping cart. One of the changes of the affiliate tracking software is adding a backup SMTP server to ensure delivery of the news, new marketing methods, etc., to the users.

The merchant wants to integrate the affiliate tracking software with the third party newsletter which he uses. Every affiliate should be a member of the newsletter. When selling music, it is important for him to know a genre of the music which is promoted by his affiliates. We need to add the genre field to the generic affiliate signup form and his profile screen to acquire the information about the genre to be promoted at different affiliate web sites. To display it, we need to

modify the affiliate table of the merchant panel so it displays genre in a new column. The marketing is managed by several co-workers with different roles. Therefore, the database of the tracking software has to be updated with an administrator account with limited permissions. A limited administrator should not be able to decline or delete affiliates, nor modify campaigns and banners.

3 Aspect-Oriented Change Representation

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*[3], and *inter-type declarations*, which enable introduction of new members into existing types, as well as introduction of compile warnings and errors.

These constructs enable to affect a method with a code to be executed before, after, or instead of it, which may be successfully used to implement any kind of *Method Substitution* change (not presented here due to space limitations). Here we will present two other aspect-oriented program schemes that can be used to realize some common changes in web application. Such schemes may actually be recognized as aspect-oriented design patterns, but it is not the intent of this paper to explore this issue in detail.

3.1 Class Exchange

Sometimes, a class has to be exchanged with another one either in the whole application, or in a part of it. This may be achieved by employing the Cuckoo's Egg design pattern [8]. A general code scheme is as follows:

```
public aspect ExchangeClass {
   public pointcut exhangedClassConstructor(): call(ExchangedClass.new(..);
   Object around(): exhangedClassConstructor() { return getExchangingObject();}
   ExchangeObject getExchangingObject() {
      if (. . .)
         new ExchangingClass();
      else
         proceed();
   }
}
```

The exhangedClassConstructor() is a pointcut that captures the ExchangedClass constructor calls using the **call**() primitive pointcut. The around advice captures these calls and prevents the ExchangedClass instance from being created. Instead, it calls the getExchangingObject() method which implements the exchange logic. ExchangingClass has to be a subtype of ExchangedClass.

[3] Join points represent well-defined places in the program execution.

The example above sketches the case in which we need to allow the construction of the original class instance under some circumstances. A more complicated case would involve several exchanging classes each of which would be appropriate under different conditions. This conditional logic could be implemented in the getExchangingObject() method or—if location based—by appropriate pointcuts.

3.2 Perform an Action After an Event

We often need to perform some action after an event, such as sending a notification, unlocking product download for user after sale, displaying some user interface control, performing some business logic, etc. Since events are actually represented by method calls, the desired action can be implemented in an after advice:

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

4 Changes in Web Applications

The changes which are required by our scenario include integration changes, grid display changes, input form changes, user rights management changes, user interface adaptation, and resource backup. These changes are applicable to the whole range of web applications. Here we will discuss three selected changes and their realization.

4.1 Integration Changes

Web applications often have to be integrated with other systems (usually other web applications). Integration with a newsletter in our scenario is a typical example of *one way integration*. When an affiliate signs up to the affiliate tracking software, we want to sign him up to a newsletter, too. When the affiliate account is deleted, he should be removed from the newsletter, too.

The essence of this integration type is one way notification: only the integrating application notifies the integrated application of relevant events. In our case, such events are the affiliate signup and affiliate account deletion. A user can be signed up or signed out from the newsletter by posting his e-mail and name to the one of the newsletter scripts. Such an integration corresponds to the Perform an Action After an Event change (see Sect. 3.2). 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 a system with several systems.

Introducing a *two way integration* can be seen as two one way integration changes: one applied to each system. A typical example of such a change is data synchronization (e.g., synchronization of user accounts) across multiple systems. When the user changes his profile in one of the systems, these changes should be visible in all of them. For example, we may want to have a forum for affiliates. To make it convenient to affiliates, user accounts of the forum and affiliate tracking system should be synchronized.

4.2 Introducing User Rights Management

Many web applications don't implement user rights management. If the web application is structured appropriately, it should be possible to specify user rights upon the individual objects and their methods, which is a precondition for applying aspect-oriented programming.

User rights management can be implemented as a Border Control design pattern [8]. According to our scenario, we have to create a restricted administrator account that will prevent the administrator from modifying campaigns and banners and decline/delete affiliates. All the methods for campaigns and banners are located in the campaigns and banners packages. The appropriate region specification will be as follows:

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

Subsequently, we have to create an around advice which will check whether the user has rights to access the specified region. This can be implemented using the Method Substitution change applied to the pointcut specified above.

4.3 Introducing a Resource Backup

As specified in our scenario, we would like to have a backup SMTP server for sending notifications. Each time the affiliate tracking software needs to send a notification, it creates an instance of the SMTPServer class which handles the connection to the SMTP server and sends an e-mail. The change to be implemented will ensure employing the backup server if the connection to the primary server fails. This change can be implemented straightforwardly as a Class Exchange (see Sect. 3.1)

5 Aspect-Oriented Change Realization Framework

The previous two sections have demonstrated how aspect-oriented programming can be used in the evolution of web applications. Change realizations we have proposed actually cover a broad range of changes independent of the application

domain. Each change realization is accompanied by its own specification. On the other hand, the initial description of the changes to be applied in our scenario is application specific. With respect to its specification, each application specific change can be seen as a specialization of some generally applicable change. This is depicted in Fig. 1 in which a general change with two specializations is presented. However, the realization of such a change is application specific. Thus, we determine the generally applicable change whose specialization our application specific change is and adapt its realization scheme.

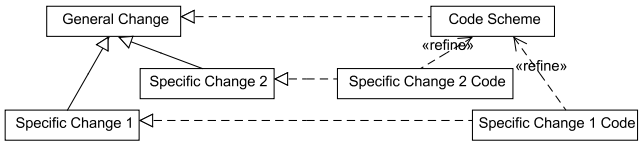


Fig. 1. General and specific changes with realization.

When planning changes, it is more convenient to think in a domain specific manner than to cope with programming language specific issues directly. In other words, it is much easier to select a change specified in an application specific manner than to decide for one of the generally applicable changes. For example, in our scenario, an introduction of a backup SMTP server was needed. This is easily identified as a resource backup, which subsequently brings us to the realization in the form of the Class Exchange.

6 Related Work

Various researchers have concentrated on the notion of evolution from automatic adaptation point of view. Evolutionary actions which are applied when particular events occur have been introduced [9]. The actions usually affect content presentation and navigation. Similarly, active rules have been proposed for adaptive web applications with the focus on evolution [4]. However, we see evolution as changes of the base application introduced in a specific context. We use aspect orientation to modularize the changes and reapply them when needed.

Our work is based on early work on aspect-oriented change management [5]. We argue that this approach is applicable in wider context if supported by a version model for aspect dependency management [10] and with appropriate aspect model that enables to control aspect recursion and stratification [1]. Aspect-oriented programming community explored several specific issues in software evolution such as database schema evolution with aspects [7] or aspect-oriented extensions of business processes and web services with crosscutting concerns of reliability, security, and transactions [3]. However, we are not aware of any work aiming specifically at capturing changes by aspects in web applications.

7 Conclusions and Further Work

We have proposed an approach to web application evolution in which changes are represented by aspect-oriented design patterns and program schemes. We identified several change types that occur in web applications as evolution or customization steps and discussed selected ones along with their realization. We also envisioned an aspect-oriented change realization framework.

To support the process of change selection, the catalogue of changes is needed in which the generalization-specialization relationships between change types would be explicitly established. We plan to search for further change types and their realizations. It is also necessary to explore change interactions and evaluate the approach practically.

Acknowledgements The work was supported by the Scientific Grant Agency of Slovak Republic (VEGA) grant No. VG 1/3102/06 and Science and Technology Assistance Agency of Slovak Republic contract No. APVT-20-007104.

References

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

[2] S. Casteleyn et al. Considering additional adaptation concerns in the design of web applications. In *Proc. of 4th Int. Conf. on Adaptive Hypermedia and Adaptive Web-Based Systems (AH2006)*, LNCS 4018, Dublin, Ireland, June 2006. Springer.

[3] A. Charfi et al. Reliable, secure, and transacted web service compositions with ao4bpel. In *4th IEEE European Conf. on Web Services (ECOWS 2006)*, pages 23–34, Zürich, Switzerland, December 2006. IEEE Computer Society.

[4] 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.

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

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

[7] 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, April 2002.

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

[9] 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.

[10] 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, September 2001. Springer.

Appendix H

Developing Applications with Aspect-Oriented Change Realization

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.

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.

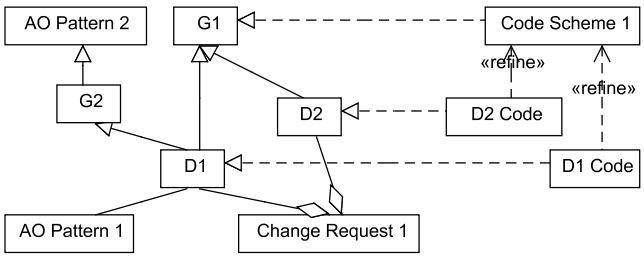


Fig. 1. Generally applicable and domain specific changes.

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 coworkers 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 I

Aspect-Oriented Change Realizations and Their Interaction

Valentino Vranić, Radoslav Menkyna, Michal Bebjak, and Peter Dolog. Aspect-oriented change realizations and their interaction. *e-Informatica Software Engineering Journal*, 3(1):43–58, 2009.

e-Informatica Software Engineering Journal, Volume 3, Issue 1, 2009

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 remark-

able 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 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

[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].

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.

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

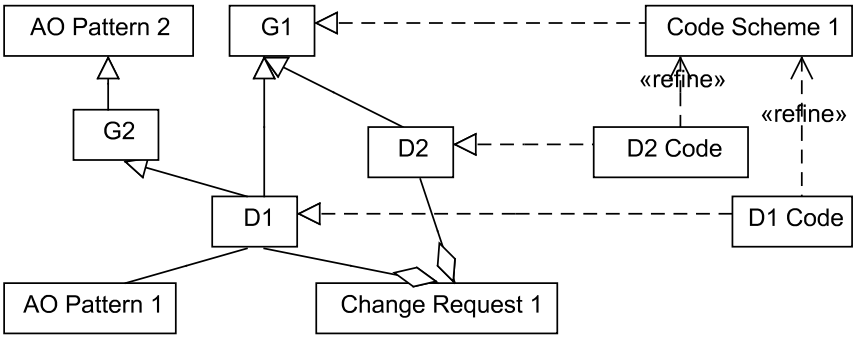


Figure 1. Generally applicable and domain specific changes

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. Fig. 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

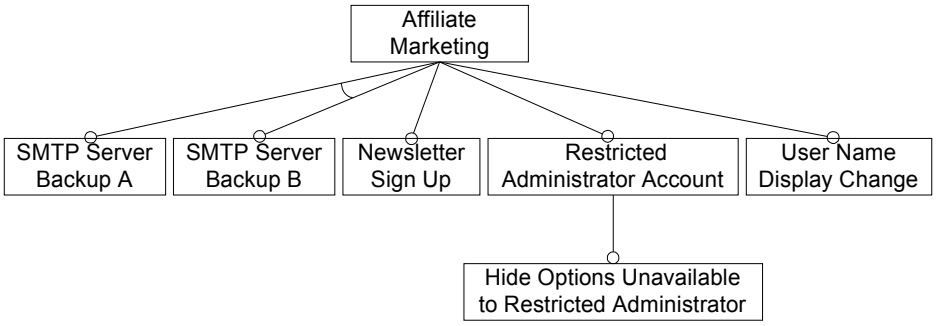


Figure 2. Affiliate marketing software change realizations in a feature diagram

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.

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 de-

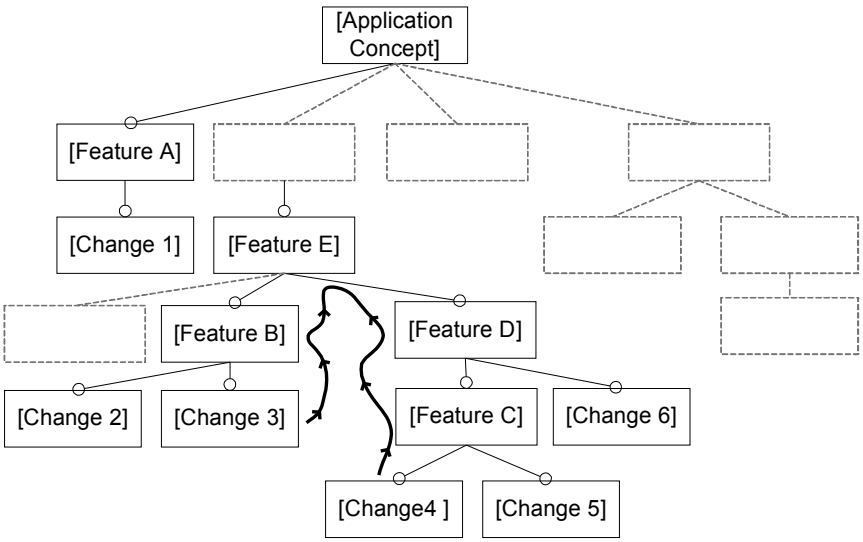


Figure 3. Constructing a partial feature model

veloped at Slovak University of Technology. We will 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. 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 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.

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.

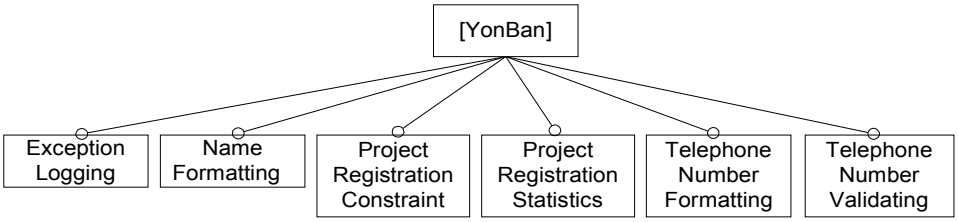


Figure 4. Initial stage of the YonBan partial feature model construction

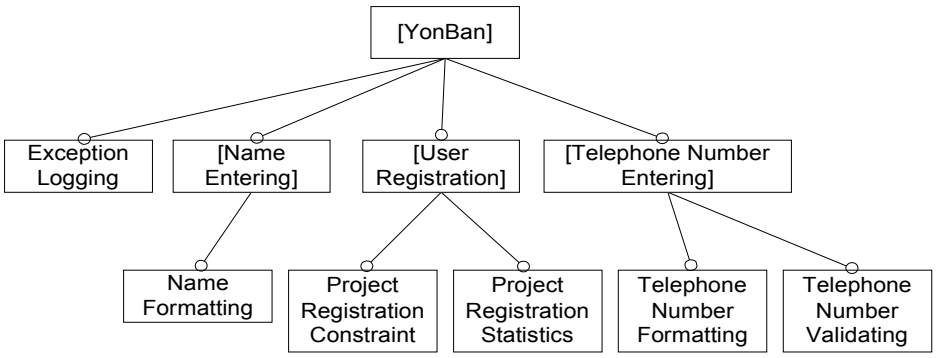


Figure 5. Identifying parent features in YonBan partial feature model construction

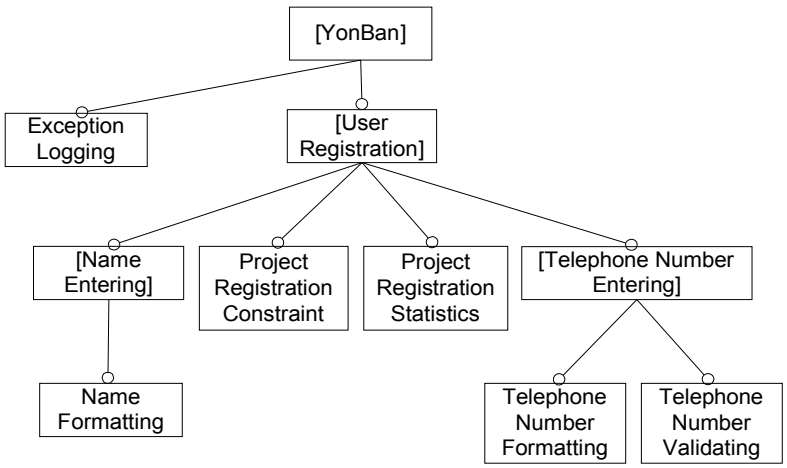


Figure 6. The final YonBan partial feature model

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 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 Registra-

tion 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 pro-

gramming 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 Ap-*

plications, 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 J

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

Radoslav Menkyna and Valentino Vranić. Aspect-oriented change realization based on multi-paradigm design with feature modeling. In *Proc. of 4th IFIP TC2 Central and East European Conference on Software Engineering Techniques, CEE-SET 2009*, Krakow, Poland, October 2009. Postproceedings, to appear.

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. It has been shown earlier that 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 idea is extended towards 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 the 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.

Keywords: change, aspect-oriented programming, multi-paradigm design, feature modeling, change interaction

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 [17].

Particular mechanisms of aspect-oriented change introduction determine the change type. Some of these change types have already been documented [1, 17], 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 [17].

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 [16]. 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 first take a look at the two-level aspect-oriented change realization model (Sect. 2). Subsequently, the approach to modeling change realization types as paradigms using feature modeling is introduced (Sect. 3). The approach employs the application domain feature model with changes expressed as features (Sect. 4). The key part of the approach is the transformational analysis—the process of finding a suitable paradigm—tailored to change realization (Sect. 5). Afterwards, it is shown how the transformational analysis results can be used to identify change interaction (Sect. 6). The approach is discussed with respect to related work (Sect. 7). Concluding notes close the paper (Sect. 8).

2 Two-Level Change Realization Framework

In our earlier work [1, 17], 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. We understand a change as a requirement focused on a particular issue perceived as indivisible from the application domain perspective.

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 vocabulary of a particular domain. A developer gets a clue to the change realization from the cataloged mappings of domain specific change types to generally applicable change types, which represent abstractions and generalizations of change realizations in a given solution domain (aspect-oriented language or framework). 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 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.

Consider a change that subsumes the 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. This is an instance of the change type called One Way Integration [1], one of the web

[1] This is an extended scenario originally published in our earlier work [1, 17].

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.

The catalog of changes [17] 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.

As another example, consider a change is needed to prevent attempts to register without providing an e-mail address. This is actually an instance of the change type called Introducing Additional Constraint on Fields [1], which can be realized using Performing Action After Event or Additional Parameter Checking, but if we assume no form validation mechanism is present, even the most general Method Substitution (which wasn't considered originally [17] for this) 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 change types that would cover all significant situations is not high. In our experiments, we managed to cope with all situations using only six of them.

On the other hand, in the domain of web applications, eleven application specific changes we identified so far cover it only partially. Each such change type requires a thorough exploration in order to discover all possible realizations

by generally applicable change types 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 [14]. Here, we will consider *multi-paradigm design with feature modeling* (MPD_{FM}), which is based on an adapted Czarnecki–Eisenecker [4] feature modeling notation [15]. Section 3.1 explains how paradigms are modeled in MPD_{FM}. Section 3.2 and 3.3 introduces two examples of change paradigm models.

3.1 Modeling Paradigms

In MPD_{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) [16].

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 [19].

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 choosing not to bind) 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. 1. 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 represent 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 [16]). To achieve its intent, an aspect may—similarly to a class—employ Methods (with the method being 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. The features that

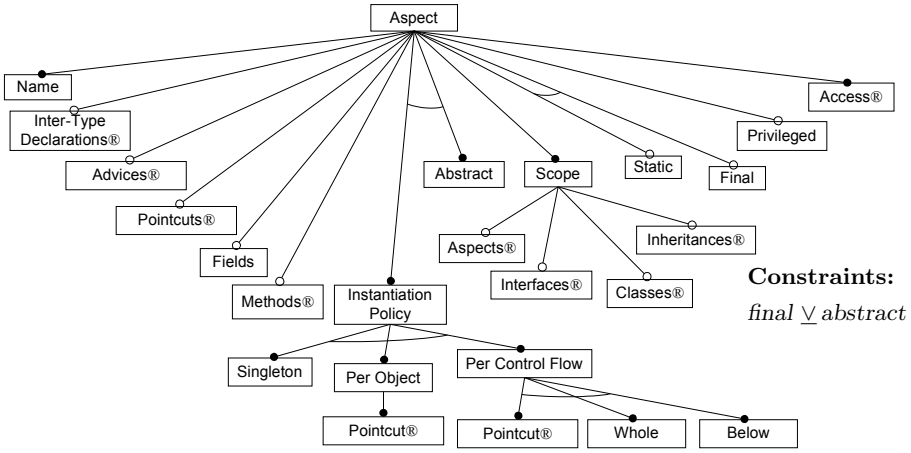


Fig. 1. The AspectJ aspect paradigm (adopted from [16]).

represent different instantiation policies are mandatory alternative features (indicated by an arc over mandatory features), which means that exactly one of them must be 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 either of the two, but it can't be both, 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 a reference to a separately expressed auxiliary concept. All the features in the aspect paradigm are bound at source time.

The constraint associated with the aspect paradigm feature diagram means that the aspect is either Final or Abstract. We use first-order predicate logic to express constraints associated with feature diagrams, but OCL could be employed, too, as a widely accepted and powerful notation for such uses (but even of wider applicability, e.g. instead of object algebras [13]).

Generally applicable changes may be seen as a kind of conceptually higher language mechanisms and modeled as paradigms in the sense of $\mathrm{MPD}_{\mathrm{FM}}$.

3.2 Method Substitution

Figure 2 shows the Method Substitution change type paradigm 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 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). Note the Context feature subfeatures. They are or-features, which means at least one them has to be selected.

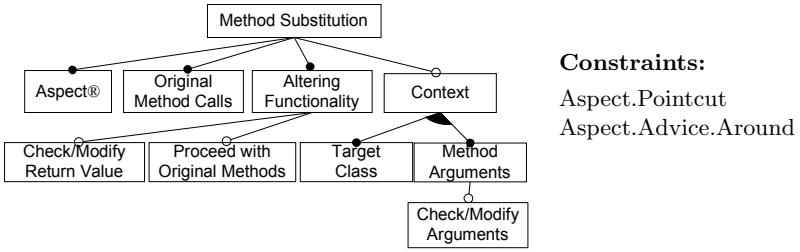


Fig. 2. Method Substitution.

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 (Fig. 2).

3.3 Performing Action After Event

Figure 3 shows the Performing Action After Event change type paradigm model. All the features have source time binding. This change type is used when an additional action (Action After Event) is needed after some events (Events) of method calls or executions, initialization, field reading or writing, or advice execution (modeled as or-features) taking or not into account their context (Context).

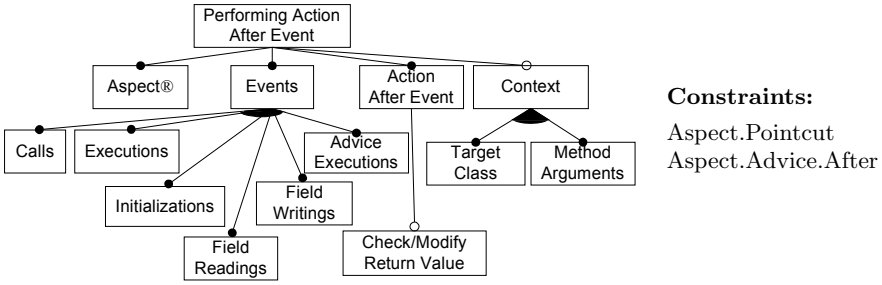


Fig. 3. Performing Action After Event.

Performing Action After Event is implemented by an aspect (Aspect) with a pointcut specifying the events and an after advice over this pointcut used to perform the desired actions, which is expressed by the constraints associated with its feature diagram (Fig. 3).

4 Feature Model of Changes

For the transformational analysis, the application domain feature model that embraces the changes is needed. We will present how changes can be expressed in the application domain feature model in our running example of affiliate tracking software.

4.1 Expressing Changes in a Feature Model

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

- SMTP Server Backup A/B —to introduce 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. 4. The concept we model is our affiliate marketing software.[2] All the changes are modeled as optional features as they can, 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*.

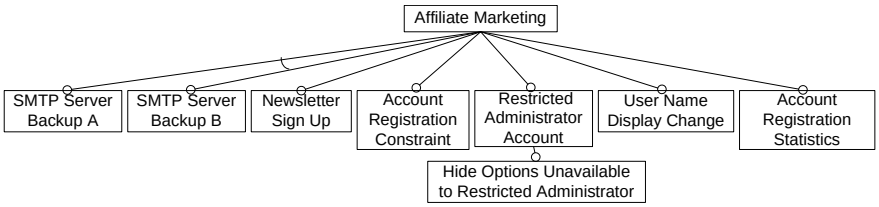


Fig. 4. Changes in the affiliate marketing software.

[2] In general, there may be several top-level concepts in one application domain.

Some change realizations make sense only in the context of some other change realizations. In other words, such change realization require 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 indication of a possible interaction between change realizations. However, with alternative features, no interaction can occur because an application instance can contain only one change realization.

4.2 Partial Feature Model

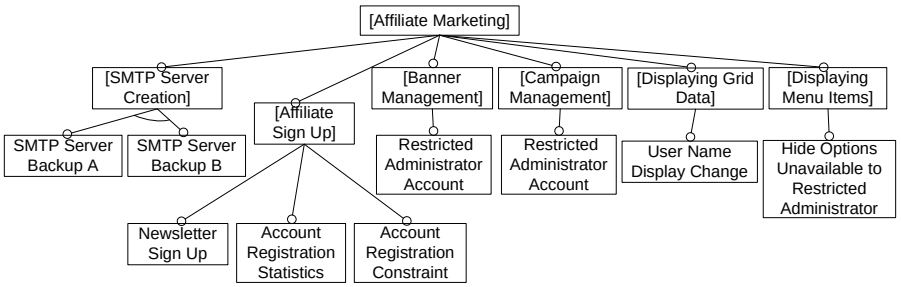
Often, no feature model of the system is available. Creating the feature model of the whole system is difficult and time consuming. Fortunately, as it has been shown [18]—for the purpose of change interaction analysis, it is a partial feature model is sufficient. The process of constructing a partial feature model starts with 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, which is exactly the model we discussed in the previous section.

In partial feature model construction, only the features that potentially take part in change interaction are being identified and modeled. Starting at change features, we proceed bottom up identifying their parent features until related features become grouped in common subtrees [18].

A partial feature model constructed from the initial feature model of the changes being introduced into our affiliate marketing software (presented in Fig. 4) is depicted in Fig. 5. All the identified change parent features are open because the sets of their subfeatures are incomplete, since we model only the changes that affect them, and since there may be other changes in the future.

At this stage, it is possible to identify potential locations of interaction. Such locations are represented as features of the system to which changes are introduced. The highest probability of interaction is among sibling features (direct subfeatures of the same parent feature) because they are potentially interdependent. This is caused by the fact that changes represented by such features usually employ the same or similar pointcuts which is generally a source of unwanted interaction. Such locations should represent primary targets of evaluation during the transformational analysis, which is the topic of the following section.

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



Constraints:

Hide Operations Unavailable to Restricted Administrator ⇒
Restricted Administration Account

Fig. 5. A partial feature model of the affiliate marketing software.

5 Transformational Analysis

The input to the transformational analysis in multi-paradigm design with feature modeling [16] are two feature models: the application domain one and the solution domain one. The output of the transformational analysis is a set of paradigm instances annotated with application domain feature model concepts and features that define the code skeleton.

A concept instance is defined as follows [16]:

> An instance I of the concept C at time t is a C's specialization achieved by configuring its features which includes the C's concept node and in which each feature whose parent is included in I obeys the following conditions:
>
> 1. All the mandatory features are included in I.
> 2. Each variable feature whose binding time is earlier than or equal to t is included or excluded in I according to the constraints of the feature diagram and those associated with it. If included, it becomes mandatory for I.
> 3. The rest of the features, i.e. the variable features whose binding time is later than t, may be included in I as variable features or excluded according to the constraints of the feature diagram and those associated with it. The constraints (both feature diagram and associated ones) on the included features may be changed as long as the set of concept instances available at later instantiation times is preserved or reduced.
> 4. The constraints associated with C's feature diagram become associated with the I's feature diagram.

5.1 Transformational Analysis of Changes

For determining change types that correspond to the changes that have to be realized, a simplified transformational analysis can be used. Changes presented in the application domain feature model are considered to be application domain concepts, and generally applicable change types to be paradigms. A complete application domain feature model may be used if available, otherwise a partial feature model has to be constructed. For each change C from the application domain feature model, the following steps are performed:

1. Select a generally applicable change type P that has not been considered for C yet.
2. If there are no more paradigms to select, the process for C has failed.
3. 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 1. Otherwise, record the paradigm instance created.

Paradigm instantiation over application domain concepts means that the inclusion of some of the paradigm nodes is 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.

If the transformational analysis fails for some change, this change is probably an instance of a new change type. The process should continue with AspectJ paradigms, which is the subject of the general transformational analysis [16].

5.2 Example

We will demonstrate the transformational analysis on several changes in the affiliate marketing software (introduced in Sect. 4.1) with the AspectJ paradigm model [16] 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, and ensure displaying an error message or logging in case of an access violation event.

Figure 6 shows the transformational analysis of the Restricted Administrator Account change. The Target Class and Method Arguments features are included to capture additional context which is needed by the Proceed with Original Methods feature when the access is granted. The If Access Granted annotation indicates the condition of proceeding with the original methods. Note that the Banner Management and Campaign Management features are mapped to the Original Method Calls feature expressed by an annotation. This means that the change affects the behavior represented by them. Such annotations are crucial to change interaction evaluation (discussed in the next section).

The transformational analysis of Account Registration Constraint would be similar. Again, we would employ the Method Substitution change type. The

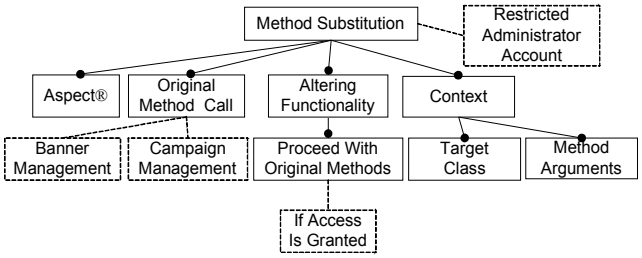


Fig. 6. Transformational analysis of the Restricted User Account change.

Original Method Calls feature would map to the Affiliate Sign Up feature and the original method will be executed only if a valid e-mail address is provided.

Figure 7 shows the transformational analysis of the Newsletter Sign Up change. Recall that this change adds a new affiliate to the existing list of newsletter recipients, which can be best realized as Performing Action After Event. In this case, the Events feature is mapped to the Affiliate Sign Up feature which represents the execution of the affiliate sign up method. Through Method Arguments, the data about the affiliate being added can be accessed (Affiliate Data) from which his e-mail address can be retrieved and subsequently added to the newsletter recipient list by the Action After Events feature. A similar transformation would apply to the Account Registration Statistics change.

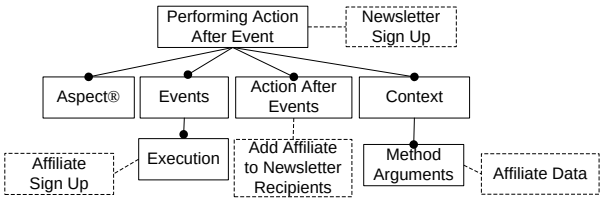


Fig. 7. Transformational analysis of the Newsletter Sign Up change.

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 [17]. The interaction is most probable if multiple changes affect the same functionality. As has been shown, such situations could be identified in part already during the creation of a partial feature model [18], but the transformational analysis can reveal more details needed to avoid 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 change type which employs an **after**() advice. In such a situation, it is important to check whether the execution order of the advices is significant. In this particular case, it is not.

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 introduced in the application domain model by constraints that will prevent affected changes from occurring together.

7 Related Work

The impact of changes implemented by aspects has been studied using slicing in concern slice dependency graphs [6]. It has been shown that the application domain feature model can be derived from concern slice dependency graphs [11]. 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 program slicing to features implemented as aspects with interaction understood as a slice intersection has been applied so far only to a very simplified version of AspectJ. Extension to cover complicated constructs has been identified as problematic. Even at this simplified level, it appears to be too coarse for applications in which the behavior is embedded in data structures [12].

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

As an alternative to our transformational analysis, framed aspects [9, 10] can be applied to the application domain feature model with each change maintained in its own frame in order to keep it separate.

Annotations that determine the feature implementation in so-called *crosscutting feature models* [8] are similar to annotations used in our transformational analysis, but no formal process to determine them is provided.

An approach to introduce program changes by changing the interpreter instead based on grammar weaving has been reported [5]. With respect to suitabil-

ity of aspect-oriented approach to deal with changes, it is worth mentioning that weaving—a prominent characteristic of aspect-oriented programming—has been identified as crucial for the automation of multi-paradigm software evolution [7].

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}) 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 generally applicable 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 adapted the process of the general transformational analysis in MPD_{FM} to work with changes as application domain concepts and generally applicable change types as paradigms. We demonstrated how such transformational analysis can help in identifying the details of change interaction.

Our further work includes extending our approach to cover the changes realized by a collaboration of multiple generally applicable change types and design patterns. We also work on improving change type models by expressing them in the Theme notation of aspect-oriented analysis and design [3].

Acknowledgements The work was supported by the Scientific Grant Agency of Slovak Republic (VEGA) grant No. VG 1/0508/09 and SOFTEC, s. r. o., Bratislava, Slovakia.

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

[5] M. Forgáč and J. Kollár. Adaptive approach for language modification. *Journal of Computer Science and Control Systems*, 2(1):9–12, 2009.

[6] 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.

[7] J. Kollár, J. Porubän, P. Václavík, M. Tóth, J. Bandáková, and M. Forgáč. Multiparadigm approaches to systems evolution. In *Computer Science and Technology Research Survey*, Košice, Slovakia, 2007.

[8] U. Kulesza, A. Garcia, F. Bleasby, and C. Lucena. Instantiating and customizing aspect-oriented architectures using crosscutting feature models. In *Workshop on Early Aspects held with OOPSLA 2005*, San Diego, USA, Nov. 2005. Available at `http://www.early-aspects.net/oopsla05ws/`.

[9] 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.

[10] N. Loughran, A. Sampaio, and A. Rashid. From requirements documents to feature models for aspect oriented product line implementation. In *MDD for Software Product-lines: Fact or Fiction?, a workshop held with ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems, MoDELS/UML 2005)*, Montego Bay, Jamaica, Oct. 2005.

[11] 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.

[12] M. Monga, F. Beltagui, and L. Blair. Investigating feature interactions by exploiting aspect oriented programming. Technical Report comp-002-2003, Lancaster University, Lancaster, UK, 2003. Available at `http://www.comp.lancs.ac.uk/computing/aose/`.

[13] M. Navarčik and I. Polášek. Object model notation. In *Proc. of 8th International Conference on Information Systems Implementation and Modelling, ISIM 2005*, Rožnov pod Radhoštěm, Czech Republic, 2005.

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

[15] 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.

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

[17] 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.

[18] 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 2008 special issue.

[19] 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.