Slovenská technická univerzita v Bratislave
Fakulta Informatiky a Informačných Technológii
Študijný program: Softvérové inžinierstvo

Bc. Michal Bebjak

Aspektovo-Orientovaná Implementácia Zmien vo Webových Aplikáciách

Diplomová práca

Vedúci diplomovej práce: Ing. Valentino Vranić, PhD.

December 2007

Slovak University of Technology in Bratislava
Faculty of Informatics and Information Technologies
Study program: Software Engineering

Bc. Michal Bebjak

Aspect-Oriented Change Implementation in Web Applications

Master's Thesis

Supervisor: Ing. Valentino Vranić, PhD.

December 2007

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

Čestne prehlasujem, že som diplomovú prácu vypracoval samostatne.

ANOTÁCIA

Slovenská technická univerzita v Bratislave
FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ
Študijný program: Softvérové inžinierstvo
Autor: Bc. Michal Bebjak
Diplomová práca: Aspektovo-orientovaná implementácia zmien vo webových aplikáciách
Vedenie diplomovej práce: Ing. Valentino Vranić, PhD.
december, 2007

V tejto práci bola analyzovaná pripravenosť webových aplikácii na zmeny vyjadrené pomocou prostriedkov aspektovo-orientovaného programovania. Na základe tejto analýzy bol vypracovaný prístup implementácie zmien pre webové aplikácie, v ktorom sú zmeny vyjadrené pomocou aspektov a sú aplikované ako inštancie všobecných apektovo-orientovaných typov zmien. V práci je identifikovaných viacero všeobecných typov zmien, ktoré sa často vyskytujú pri vývoji a úpravách podľa priania zákazníka, a tiež vzťahy medzi týmito typmi zmien a všeobecnými aspektovo-orientovanými typmi zmien. Navrhnutý prístup bol overený pomocou implemetácie viacerých zmien a výsledky ukazujú, že takéto zmeny je možné vyjadriť pomocou aspektov. Kvôli ukázaniu možnosti implementácie zmien aj v menej expresívnych aspektovo-orientovaných jazykoch a rámcoch bol predstavený rámec Seasar. Pre tento rámec boli navrhnuté postupy implementácie pokročilejších aspektovo-orientovaných konštrukcií.

ANNOTATION

Slovak University of Technology Bratislava
FACULTY OF INFORMATICS AND INFORMATION TECHNOLOGIES
Degree Course: SOFTWARE ENGINEERING
Author: Bc. Michal Bebjak
Thesis: Aspect-Oriented Change Implementation in Web Applications
Supervisor: Ing. Valentino Vranić, PhD.
2007, December

In this thesis, the possibility of expressing changes in web applications by means of aspect-oriented programming was analyzed. On the basis of this analysis, an approach to web application evolution was introduced. In this approach, changes are represented as aspects and applied as instantiations of generic aspect-oriented change types. Several change types which can occur in web applications as evolution or customization steps and correspondence between these change types and generic aspect-oriented change types were identified. This approach was evaluated by implementing several changes and results of this evaluation show that such changes can be managed as aspects. To show that proposed approach to change implementation is not limited to AspectJ and can be used also in less expressive aspect-oriented languages or frameworks, the Seasar framework was introduced and techniques for implementing advanced aspect-oriented constructs in this framework were developed.

NAMIESTO TEJTO STRANY VLOZIT ZADANIE
original resp. kopia

Contents

Chapter 1

Introduction

Changes are inseparable part of software developement. Changes take place in the process of developement as well as in the software maintenance. The goal of change control is to adapt the application to the everchanging user requirements and operating enviroment.

Huge costs and low speed of implementation are characteristic to the change implementation. Often, change implementation implies a redesign of the whole system, which is very cost inefective. The necessity of improving the software adaptability to changes is fairly evident.

Software adaptibility can be improved by employing the aspect-oriented programming. Aspect-oriented programming allows us to express changes as aspects that are being applied to the system. This possibility is investigated in this work. The stress is given on identifying and applying changes typical, though not limited to, generic web applications. Generic web applications have to be able to dynamically adapt themselves to the changing enviroment, integration with third party systems, and customization.

The rest of the work is structured as follows. Chapter 2 presents base concepts of the Aspect-oriented programming. Chapter 3 introduces the problem of change control, compares object-oriented and aspect-oriented approach to change control, and analysis web application capability for changes. Chapter 4 establishes a scenario of changes in the process of adapting affiliate tracking software used throughout next Chapters of this work. Chapter 5 proposes aspect-oriented patterns and program schemes that can be used to implement these changes. Chapter 6 identifies several interesting change types in this scenario valid, though not limited to, for the whole range of web applications. Chapter 7 envisions an aspect-oriented change realization framework and puts the identified change types into the context of it. Chapter 8 evaluates the approach proposed by the framework. Chapter 9 introduces Seasar framework which provides aspect-oriented programming for PHP language and shows that proposed approach can be implemented also in less expressive aspect-oriented languages and frameworks. Chapter 10 conclusions and directions of further work.

Chapter 2

Aspect-Oriented Programming

The goal of aspect-oriented programming is to make it possible to deal with crosscutting concerns of a system's behavior as separately as possible [8]. This is achieved by allowing programmers to first express each of the system concerns in a separate and natural form and by their subsequent automatic merging in a process called weaving.

2.1 Main Concepts of Aspect-Oriented Programming

Main concept in aspect-oriented programming is aspect [4]. An aspect is made of a pointcut selecting some join points, an advice (a code to execute).

2.1.1 Join points

The join point represents a language construction to which an advice can be connected. Join points can be function calls, variable assigments... Aspect-oriented languages usually support only subset of all possible join points.

2.1.2 Pointcuts

An aspect selects a join point by matching it with pointcut pattern. The pattern can be a term with variables matching an instruction, disjunction, conjunction, and negation of patterns. Control flow pointcut cflow(B) is a pointcut which intuitively represents all the join points which are in the control flow of a method B including the join point represented by B. cflowbelow(B) is similar but excludes the the join point represented by B.

2.1.3 Advices

Advice represents code that should be executed and type that specifies when it should be executed. Advice type can be before, after, and around.

Before advice When a before advice matches the current instruction, it is executed before this instruction.

After advice The after advice is executed after the instruction matched by pointcut has completed. To make sense, it should be applied to instructions which perform sequencing. If the instruction does not perform sequencing (e.g. it can throw exceptions), then the advice might not be executed. If the instruction is a procedure call, the advice will be executed when the procedure returns.

Around advice In order to accommodate around advice, the aspect-oriented language must contain an additional instruction **proceed**() which can be used in the code of an around advice. Typically, an around aspect starts by executing its code before the current instruction. The advice code may proceed by executing the instruction matched by the around advice using the instruction **proceed**(). An advice may also terminate without executing the current instruction.

2.2 The AspectJ Approach to Aspect-Oriented Programming

AspectJ is a general-purpose, aspect-oriented extension to the Java programming language [10]. It is considered to be a model for other aspect-oriented languages.

AspectJ implements all main aspect-oriented concepts described in Section 2.1.

2.2.1 Join Points

Avalaible join points in AspectJ are:

- method join points
- constructor join points
- field access join points
- exception handler execution join points
- class initialization join points
- object initialization join points

- object pre-initialization join points
- advice execution join points

2.2.2 Pointcuts

Pointcuts in AspectJ can select a join point that is a call/execution of a method-/constructor, and they can also capture the method's context, such as the target object on which the method was called and the method's arguments.

2.2.3 Advices

The advices in the AspectJ can be of before, after, and around type.

2.2.4 Intertype Declaratiosn

The intertype declaration is a static crosscutting instruction that introduces a changes to the classes, interfaces, and aspects of the system. It makes static changes to the modules that do not directly affect their behavior. For example, it can add a method or field to a class.

2.2.5 Compile-time Declaration

The compile-time declaration is a static crosscutting instruction that allows us to add compile-time warnings and errors upon detecting certain usage patterns.

2.3 Aspect-oriented Languages

Aspect-oriented programming is not intended to replace object-oriented programming. It extends object-oriented programming. There are many aspect-oriented extensions of existing mainly object-oriented languages. In general aspect-oriented programming can be implemented as a language extension or as a framework. AspectJ represents a language extension. It defines new language constructs which used to express aspects. If the aspect-oriented programming is done by using framework, no new language constructs need to be added to the base language. Pointcuts are usually specified in external configuration files or anotations, and aspects are represented as regular classes.

Chapter 3

Change Control

The level of change control provided by existing tools varies significantly, but most of these tools build up on version models. A version model defines entities to be versioned, version identification, as well as operations for retrieving versions and constructing new versions [3].The version models can be classified into state-based and change-based. State-based models usually describe entities in term of revisions and variants. Change in state-based model is described by difference between two versions. When new version is created, change is merged with all previous versions. Change-based models consider the change as a entity to be versioned. A version is created by applying all changes to the baseline. This allows to combine changes freely and simply make different versions by applying only selected changes. This approach is similar to aproach proposed in this report. Changes implemented using aspects can be modularized and made pluggable and reapplicable. Section 3.1 explains the categorization of changes and analysis possibilities of their implementation using aspect-oriented approach. Section 3.2 compares aspect-oriented and object-oriented approach to change implementation. Finally section 3.3 analyzes capability of web applications for changes.

3.1 Change Categorization

Changes can be divided into bug fixes, change requests, and customizations.

3.1.1 Bug Fixes

Bug fixes should not be implemented using apsect-oriented programming. It would have very bad effect on core source code. Change requests should be applied to healthy and working core. If the bug fixes were done using aspect-oriented programming, soon there would be a huge amount of changes that need to be applied to all users. No one wants a buggy system and if the change is to be applied to all system, then it is better to implement it right to the core

of the system and not as some external patch. Thus the bug fixes shouldn't be imlemented using aspect-oriented programming.

3.1.2 Change Requests

Change requests can be divided to two main categories:

1. changes applicable to all users
2. changes applicable to single user or group of users

If a change should be applicable to all users it can be implemented in two ways. If it has a character of cross cutting concern it should be implemented using aspect-oriented programming because aspect-oriented programming is better suitable for cross-cutting concerns. Aspect-oriented programming was initially intended to capture cross-cutting concers.

If the change affects only single class it could be implemented using object-oriented programming. But if the change is implemented using object-oriented programming, it can not be later easily removed from system. It is also difficult to distinguish if the change is really applicable to all users. Therefore this type of change should be implemented using aspect-oriented programmming too, even if this implementation may not be so efficient as object-oriented programming implementation.

If a change is applicable to single or group of users it should always be implemented using aspect-oriented programmming, because we want to be able to apply or remove this change from system easily.

3.1.3 Customizations

It is almost impossible to create an application that will match the requirements of all users. In order to satisfy most of the customers it is welcomed if the application can be easily customized. Most of the customers won't mind paying for customization if it can put the application nearer to their requirements and expectations.

The problem of customization is typical for generic applications. Users want to integrate application with their systems, modify it ... If some customization is developed for one user, then it will be great if the same customization could be sold to other users which may eventually need it to. Some users may also want to have more customizations made on their application. So all customizations should be implemented so that they can be used on all systems and can cooperate with other customizations.

3.2 Comparing Object-oriented and Aspect-oriented Approach to Change Control

Initially, aspect-oriented programming was designed to express cross-cutting concerns. Some recent works suggest that aspects can be also used to express

changes [5, 2, 13, 9]. In this chapter aspect-oriented and object-oriented approach is compared.

A case study by Papapetrou and Papadopoulos [13] compares aspect-oriented programming and object-oriented programming with respect to change control. Two changes to the existing web crawling system were implemented . Changes were implemented by two independent teams. Number of lines of code added, number of places to which code was added, and number of files to which code was added metric was used to compare both approaches.

Four changes were implemented and compared. These changes were logging, DNS monitoring, database monitoring, database optimizer.

change	lines of code		places to add		files to add	
	OOP	AOP	OOP	AOP	OOP	AOP
logging	126	19	73	1	8	1
DNS monitoring	15	40	3	1	1	1
database monitoring	15	40	3	1	1	1
database optimizer	45	45	3	1	1	1

Table 3.1: Comparison of aspect-oriented and object-oriented approach [13]

It can be seen that changes implemented using aspect-oriented programming are much better isolated than the ones using object-oriented programming. Logging is a great example for aspect-oriented programming. When implemented using object-oriented programming 126 lines of code needed to be added to 73 places in 8 different files. This is huge amount of work especially when we need to make modifications to these changes. On the other hand when implemented using aspect-oriented programming 19 lines of code were added to one place in one file. The main benefit of logging using aspect-oriented programming is that a whole change is located in one file. When we decide to remove logging from system, we need to remove just one file. On the other hand, when using object-oriented programming wee need to modify 8 files and look at 73 places.

Not all changes can be implemented so efficiently using aspect-oriented programming. Database monitoring and optimizer required almost 3 times more lines of code when implemented using aspect-oriented programming. But the benefit of change isolation still remains. object-oriented programming version modified 3 places in the system while aspect-oriented programming modified just 1.

Results similar to [13] can be found also in [9]. In [9] aspect-oriented programming and object-oriented programming approach was also investigated and lines of code together with number of classes changed was used as metrics.

In this case, the difference between lines of code changed using aspect-oriented programming and object-oriented programming is not so significant. In some cases change implemented by aspect-oriented programming required more lines of code than object-oriented programming version. But the benefit

change	lines of code		classes changed	
	OOP	AOP	OOP	AOP
add spam checking	36	44 (in aspect)	2	0 (aspect changed)
replace spam checking	15	15 (in aspect)	1	0 (aspect changed)
replace logging	184	162 (in aspect)	12	0 (aspect changed)

Table 3.2: Comparison of aspect-oriented and object-oriented approach [9]

of change isolation is visible. In all cases, just the aspect had to be modified. When object-oriented programming was used 1–12 classes were modifed.

Works[13, 9] stress out that changes implemented using aspect-oriented programming are very well isolated and separated from existing system. This has many benefits such as

- Changes can be easily implemented or removed from system.
- Further modification to changes are made just on one place. This reduces the problem of forgetting to implement modification on some place.
- It is not neccessary to explore whole system to understand a change.
- Change can be easily reviewed, monitored and tracked.

3.3 Capability of Web Applications for Changes

Web applications can be classified as structured and unstructured applications. Capability for changes of these two groups is discussed in Sections 3.3.1 and 3.3.2, respectively.

3.3.1 Unstructured Applications

Unstructured applications are usually small-sized applications. Application logic is tangled with the presentation logic and procedural paradigm is usually used to implement this type of applications. A source code for this type of applications usually looks like this:

```
<html>
...
...
<?php
  if ($_REQUEST['action'] == 'action1') {
    // do some stuff
    ...
    // usually many LOC
```

```
  } elseif ($_REQUEST['action'] == 'action2') {
    // do some other stuff
    ...
    //
  }
?>
...
...
Username: <?=$user['username']?>
...
...
</html>
```

To implement changes for unstructured applications is very difficult. Change in one place can cause problems elsewhere in application, source code is poorly arranged, so it is difficult even to find a place where the change should be applied.

For unstructured applications it is also very difficult to use aspect-oriented programming for change implementation. The main reason is that the join points are hard to localize and are very unstable.

Example. This example shows code snippet that inserts new user into table:

```
...
// some checks
...
$sql = 'insert into users (uname, name) values ('$uname', '$name')';
$ret = mysql_query($sql);
...
stuff that should happen after user signup
...
```

Required change. If user haven't specified his name, then the name should be set to username.

Solution. We need to add following code before command that creates the SQL statement

```
if ($name == '') $name = $uname;
```

Problems. If we want to implement this change, we have to specify an advice body and pointcut. In this case, advice body is clear (line of code above). The problem is to specify a pointcut. In order for the pointcut to be precise enough, we need to use lines of code as join points. Otherwise we wouldn't be able to implement the required change. Line of code join points are very unstable. They are specified by the line number. If someone changes the source code before line that is adviced, line number changes, and change will no longer work as it should.

3.3.2 Structured Applications

All applications should be implemented in some structured way. Especially bigger sized applications. Object-oriented paradigm is usually used for this type of applications. Structured applications are often implement by means of model-view-controller pattern and can be divided into three layers:

1. Domain layer (model)
2. Presentation layer (view and controller)
3. Persistence layer

3.3.3 Domain Layer

Domain layer is responsible for the application logic (often called bussiness logic). This layer is usually implemented using object-oriented programming. This is a good assumption for change control using aspect-oriented programming. Changes can be applied to class methods. The join points in this case are method calls.

To accurately specify a pointcut of change, there has to be enough join points in model layer. This can be ensured by fine-grained granularity of the domain layer. Therefore no method should be longer than aproximately 20 lines. If a method is longer, it can be usually divided into multiple methods. This fine-grained granularity is also good for code limpidity.

Example. Example of coarse-grained source code follows:

```
...
function addUser($uname, $name) {
  ...
  $sql = 'insert into users ...';
  mysql_query($sql);
  ...
  if ($signupBonus) {
    ...
    $sql = 'insert into provisions ...';
    mysql_query($sql);
    ...
  }
}
...
```

Required change. Send an email notification to a user when he receives a signup bonus.

Problem. When the code is coarse-grainedm, we can not specify the pointcut precisely enough. In this case we would need to execute the advice after the method addUser() is executed. In the advice body, we need to check again whether the signup bonus should be given and whether it was finally given to user.

Example. Example of fine-grained source code follows:

```
...
function saveUserToDb(...) {
  ...
  $sql = 'insert into users ...';
  mysql_query($sql);
  ...
}
...
function addSignupBonusToUser(...) {
  ...
  $sql = 'insert into provisions ...';
  mysql_query($sql);
  ...
  return true;
}
...
function addUser($uname, $name) {
  $this->saveUserToDb(...);
  if ($signupBonus) {
    $this->addSignupBonusToUser();
  }
}
...
```

Required change. The same as in previous example.

Problem: In this case we can specify the pointcut much more precisely. We need to execute the advice after the method addSignupBonusToUser() is executed. In the advice body we don't need to check if the signup bonus should be given. If the method addSignupBonusToUser() returns true, we can send an email notification.

There are more ways to specify pointcut. If the application is fine-grained, well documented, and modeled, pointcuts can be specified using UML diagrams. This approach was introduced by Gazzola et al. [2]. Pointcuts are defined using UML statechart diagram but activity diagrams can be used too.

3.3.4 Presentation Layer

The presentiation layer is responsible for presentation of data stored in applications. Presentation layer for web applictions is usually written in HTML, but

rich web applications based on JavaScript and Ajax are getting more popular and presentation layer of Ajax applications is usually component based.

Simple HTML. No visual components are used in presentation layer written in simple HTML. Tables, forms ... are generated in HTML templates.
Example:

```
...
Users list<br>
<table>
<tr><th>username</th><th>name</th></tr>
<? foreach($users as $user) { ?>
  <tr><td><?=$user['uname']?></td>
      <td><?=$user['name']?></td></tr>
<? } ?>
</table>
...
```

This type of the presentation layer is not prepared for changes. If we want to add column to the users table, we have to modify the original source code. We can't use aspect-oriented programming because there are no join points to be addressed by the pointcut of a change.

Component based presentation layer. Visual components are used for tables, forms ... in component based presentation layer.
We can see Table component generated by generate() method in following example:

```
...
Users list<br>
<?
  $table = new Table($users);
  $table->generate();
?>
...
```

If we want to add a new column, we can do that in the Domain layer. We can change the way the table is generated by applying change to the Table class, too. Component based presentation layer is well prepared for changes.

A similar component based approach can be also used for rich web applications based on JavaScript.

3.3.5 Persistence Layer

The persistence layer is responsible for storing application data. Relational databases are usually used for this purpose. Often, some object-relational mapping layer is used to communicate with database.

If the change that has to be implemented requires no database changes, we don't need to worry about the persistence layer. But some changes will require

the database changes. The database changes required by the change can not affect functionality of other system parts. Typical database changes are:

- adding column to existing table
- new table

A new table in the system shouldn't influence other parts of the system. We just need to ensure that:

- a requested table is created before the is introduced
- a created table is removed when the change is removed

A new column added to the existing table could cause problems if application uses the select commands that select all colums in table. Change could use a new table instead of a new column. This new table will contain the foreign key to the modified table and a new column. This solution shouldn't influence the rest of system as it doesn't modify existing tables.

Chapter 4

Adapting Affiliate Tracking Software: A Change Scenario

In the proposed approach, scenario of a web application will be employed throughout the rest of the thesis which undergoes a lively evolution: affiliate tracking software.[1] Affiliate tracking software is used to support the so-called affiliate marketing [7]. Affiliate marketing is a method of advertising web businesses (merchants) on 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.

A simplified schema of affiliate marketing is depicted in Figure 4.1. A customer visits an affiliate's page which refers him to the merchant page. When he buys something from the merchant, goods are sent to the customer and 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. This software is also able to send notifications about new sales, signed up affiliates, etc.

Consider such a general affiliate tracking software is bought by a merchant that runs an online music shop to support his business by building affiliate marketing. A successful adaptation of the general affiliate tracking software requires a series of changes to be applied. First, the affiliate tracking software has to be integrated with the shopping cart, so it can track sales. General integration methods and integration methods for most popular kinds of shops (including ours) are already implemented, so we do not need to implement this.

To motivate and keep affiliates informed, we want to send them e-mails about news, new marketing methods, etc. To ensure e-mail delivery, we want to

[1] This chapter is adapted from parts of my paper Evolution of Web Applications with Aspect-Oriented Design Patterns [1] to which my contribution is approximately 60 %

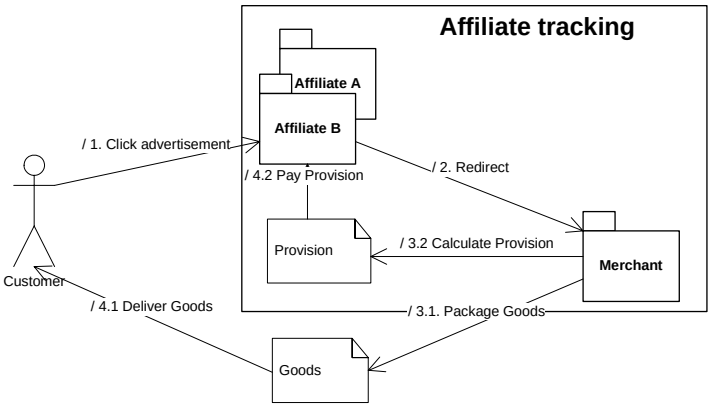

Figure 4.1: Affiliate marketing scheme.

use two SMTP servers: one as the main server, and the other one as a backup server. The affiliate tracking software does not implement this feature.

The merchant also wants to integrate the affiliate tracking software with the third party newsletter which the he uses and which fits his needs. Every affiliate should be a member of the newsletter. Different merchant has different information needs. For example when selling music, it is important for 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 managemd 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.

Chapter 5

Aspect-Oriented Change Realization

Changes can be successfully coped with using aspect-oriented programming techniques.[1] Here we will present selected aspect-oriented programming techniques that can be used to realize some common changes. Some of these techniques may actually be recognized as established aspect-oriented design patterns. As the objective of this thesis is not to search for not yet identified aspect-oriented patterns, we consider other cases as pattern-like code schemes appropriate to implement changes. We leave exploration of their potential as aspect-oriented design patterns for further work.

In the AspectJ style of aspect-oriented programming, the crosscutting concerns are captured in units called aspects that may be seen as playing the role the classes play in object-orientated programming. However, at runtime, the aspects are automatically instantiated by which they start to affect the code they are bound to. Unlike classes, whose runtime manifestations (i.e., objects) are called differently, runtime aspects are called simply aspects.

In AspectJ, aspects may contain fields and methods much the same way 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*,[2] and *inter-type declarations*, which enable introduction of new members into existing types, as well as introduction of compile warnings and errors. The complete presentation of aspect-oriented programming in AspectJ is not the intent of this thesis, so we will limit our explanation to the necessary extent put in the context of change implementation.

[1] This chapter is adapted from parts of my paper Evolution of Web Applications with Aspect-Oriented Design Patterns [1] to which my contribution is approximately 60 %

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

5.1 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 [12]. A general code scheme is as follows:

```
public aspect ExchangeClass {
  public pointcut exhangedClassConstructor(): call(ExchangedClass.new(..);

  Object around(): exhangedClassConstructor() {
    if (...) {
      return new ExchangingClass();
    } else {
      return 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 decides which if the class should be exchanged and returns instants of original or exchanging class ExchangingClass. In general, it is conditional or unconditional returning of an instance of the ExchangingClass. For this to be possible, the ExchangingClass has to be a subtype of the ExchangedClass.

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 around advice or—if location based—by appropriate pointcuts.

5.2 Method Substitution

Similarly to the class exchange, we may need to change or completely disable the execution of some methods. This can also be easily acheived with an around advice. The affected methods have to be specified by an appropriate pointcut. 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 following pointcut captures all method calls of the method called method() belonging to the TargetClass class:

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

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

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

```
pointcut specificMethodCallsPointcut(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 methodCallsPointcut(TargetClass t, int a): . . .;

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

Note that sometimes we may need to let the original method call proceed. This is done by simply letting the captured join point proceed. In the code snippet above, we supply the join point proceed with the original join point context. The context includes the captured method arguments, which may have to be adapted—and sometimes this may actually be the whole method adaptation we need—and this can be done directly in the **proceed**() call as in this example where the integer argument is halved:

```
proceed(t, a / 2);
```

5.3 Enumeration Modification

Assume we need to modify some enumeration type. Enumeration types are represented as classes with a static field per each enumartion value. A single enumeration value type is represented as a class with a field that holds the actual (usually integer) value and its name.

To add a new enumeration value, we need to introduce a static field that represents this value and initialize it:

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

A common practice is to include a method in the enumeration type class by which one may retrieve all possible enumeration values for that type. Obviously,

we would have to modify this method return value, too. We consider the return value modification as a change type of its own and as such treat it separately in Section 5.5.

5.4 Additional Parameter Checking

Often, a change involves additional check or a 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 Section 5.2. 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 methodCallsPointcut(TargetClass t, int a): . . .;

  ReturnType around(/* arguments */) throws WrongParamsException:
    methodCallsPointcut(/* arguments */) {

    check(/* arguments */);
    return proceed();
  }

  void check(/* arguments */) throws WrongParamsException {
    if (arg1 != 'desired value') {
      throw new WrongParamsException();
    }
  }
}
```

5.5 Additional Return Value Checking/Modification

When we need to implement some additional processing of the method return value, we can do it again using a similar pointcut as we did in the method substitution described in Section 5.2). The method return value is processed by an around advice:

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

  private ReturnType retValue;

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

```
  private void processOutput(/* arguments */) {
    // processing logic
  }
}
```

In the around advice, we assign the original return value (retValue = **proceed**()) to the private member of the aspect. Then this value is processed using the processOutput() method and returned by the around advice.

5.6 Performing Action After Event

We often need to perform some action after event, such as sending a notification, unlocking product download to the user after a sale, displaying a user interface control, performing some bussiness logic, etc. Since events are actually represented by method calls, we may again rely on the method call capturing pointcut we first introduced in Section 5.2). The desired action can be implemented in an after advice:

```
public aspect PerformingActionAfterEvent {
  pointcut methodCallsPointcut(TargetClass t, int a): . . .;

  after(/* captured arguments */): methodCallsPointcut(/* captured arguments */) {
    performAction(/* captured arguments */);
  }

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

The after advice executes after the captured method calls. It calls the performAction() method to perform the desired action.

5.7 Logging

We often need to log some events in the system, such as user registration, sale registration, errors, etc. Logging is similiar to Performing Action After Event change, where the action performed after event is to log the event. When implementing logging, a method call pointcut usually captures a group of methods that perform similar tasks and their call can be logged same way.

```
public aspect Logging {
  pointcut methodCallsPointcut(int a): . . .;

  after(/* captured arguments */): methodCallsPointcut(/* captured arguments */) {
    // log method call
  }
}
```

Chapter 6

Applying Changes to Web Applications

Our change scenario concerning adapting affiliate tracking software contains many typical web application changes.[1] We will go through each change and explain how it could be realized using the general aspect-oriented change schemes introduced in the previous chapter. This list of changes is not a complete list and many other changes may be appended to this list later.

6.1 Integration Changes

Web applications often have to be integrated with other systems (usually other web applications). Two main types of integration may be identified: one way and two way integration.

Integration with a newsletter from our scenario is a typical example of *one way integration*. Assume that 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 from the affiliate tracking software, it should be deleted from the newsletter, too.

The essence of this type of integration 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.

To implement this change, we first have to identify a place in the affiliate tracking software at which the integration should happen. The place for the signup to the newsletter is upon returning from the AffiliateSignup.processSignup() method. After returning from this method, a user is signed up to the affiliate tracking software. The place for removing from the newsletter is after returning from the Affiliate.delete() method.

[1] This chapter is adapted from parts of my paper Evolution of Web Applications with Aspect-Oriented Design Patterns [1] to which my contribution is approximately 60 %

The integrated application has to be notified of these events. 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 Action After Event change (see Section 5.6). In the after advice we will make a post to the newsletter sign up or 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 integration changes to integrate a system with several other 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.

Our scenario doesn't require this kind of change, but it is also very common and worth mentioning. For example we may want to have a forum, where our affiliates can discuss. To make this comfortable to affiliates, user accounts of forum and affiliate tracking system should be synchronized. Implementation of this kind of change requires the ability to modify both systems. If this requirement is fulfilled, then we can implement this change as two one way integration changes.

6.2 Grid Display Changes

It is often required to modify the way data are displayed or inserted. In web applications, data are very often displayed in grids, and data input is usually realized via forms. Such changes are also required by our scenario.

Grids usually directly display the content of a database table or collation of data from multiple tables. 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 type of reusable component or generated by row and cell processing methods. If the grid is hard coded for a specific view, it is much more 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 Section 5.5). If not, it has to be provided at least with the methods for processing rows and cells as in our affiliate tracking software scenario. These processing method calls (e.g., displayRow() and displayCell()) will be the join points for our change.

Altering column presentation in a grid is usually neccessary due to different data presentation formats at different places. For example, in Japan and Hungary, in contrast to most other countries, the surname is placed before the given names. This change requires preprocessing of all the data to be displayed in a grid before actually displaying them. A simpler solution is to modify the way the grid cells are rendered by the displayCell() method, which may be implemented again as a Method Substitution change (see Section 5.2):

```
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 == 'column name to be modified') {
      . . . // display the modified column
    } else {
      proceed();
    }
  }
}
```

Sometimes, we have to *add or remove a column from a grid*. According to our scenario, a genre field has to be added to the affiliate table. A database has to be accommodated to this change, but we also have to display the additional genere column in the affiliate table. This may be performed *after an event* of dispalying the existing columns of the affiliate table which brings us to the Performing Action After Event change (see Section 5.6).

Not all data displayed in a grid are always necessary. Sometimes, we would like to remove certain columns to make the grid more compendious. This requires a conditional execution of the displayCell() method that may be realized as a Method Substitution change (see Section 5.2). The solution is similar to altering column in grid.

6.3 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 on form inputs. In our scenario, we have to add genre column to the signup and affiliate profile form. If we want to modify forms using aspect-oriented programming, they may not be hard coded in HTML. They have to be generated from a list of fields or at least generated by a method. In such a way we can apply changes as an Enumeration Modification change (see Section 5.3).

To *add new fields*, we need to create an after advice which will be executed after the usual form processing and will store the newly added field. Forms do not have to contain all of the fields if there are default values for these fields.

The after advice can be used similarly to *remove fields from a form*. A default value can be implemented as an Additional Return Value Checking/Modification change (see Section 5.5) of the method used to retrieve list of fields in the form.

If we want to introduce *additional validations on the form input data* to the system without built-in validation, an Additional Parameter Checking change (see Section 5.4) can be applied to methods that process values submitted by the form. If we want to add new validator to systems, that already have built-in validation, the new validator usually has to be added to the list of validators. This can be done by implementing Performing Action After Event change (see

Section 5.6), which adds new validator to the list of validators after this list is initalized.

6.4 Introducing User Rights Management

Many web applications don't implement user rights management features and it has to be introduced as a change. 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 introduced using a modification of the Border Control design pattern [12]. This pattern will be employed to specify regions in our application. 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 application.campaigns and application.banners packages. The method for deleting an affiliate is Affiliate.delete() and the method for declining affiliate is Affiliate.decline(). The region specification will be as follows:

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

Then 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 method substitution (see Section 5.2) applied to the pointcut specified above.

6.5 User Interface Restriction

Most users don't make use of all the functions implemented in applications and would like to *restrict the user interface*. It is particularly annoying when a user sees options inaccessible to him due to user rights restrictions posed upon his account.s

We have a similar problem in our application with the campaigns and banners option. This option has to be removed from the menu for the restricted administrator account we have just created (see Section 6.4). Menu items are retrieved by the Menu.getMenuItems() method. We only have to modify the return value of this method which may be achieved by applying an Additional Return Value Checking/Modification change (see Section 5.5) to remove the banners and campaigns items.

6.6 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 constructor of the SMTPServer class has parameters for the server URL, user name, and password. On creation, it tries to connect to the specified server.

The change to be implemented will ensure employing the backup server if the connection to the primary server fails. Such a *resource backup* can be implemented as a Class Exchange change (see Section 5.1), where the exchange logic will be as follows:

```
SMTPServer around(): exhangedClassConstructor( ) {
  return getExchangeObject(proceed());
}

SMTPServer getExchangeObject(SMTPServer server) {
  if (server.isConnected()) {
    return server;
  } else {
    return new SMTPServer(/* alternative SMTP server params */);
  }
}
```

The around advice creates the original instance of SMTPServer class using **proceed**() method and passes it to the getExchangeObject() method which implements the exchange logic. If the original server is connected, this method returns an object of the original server type. Otherwise, it returns an object of the alternative server type.

Chapter 7

Aspect-Oriented Change Realization Framework

In Section 7.1 a change realization framework is proposed. Section 7.2 discusses how to implement change of already esxisting change.

7.1 Model of Change Realization Framework

The previous two chapters have demonstrated how aspect-oriented programming can be used in the evolution of web applications.[1] 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 Figure 7.1 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.

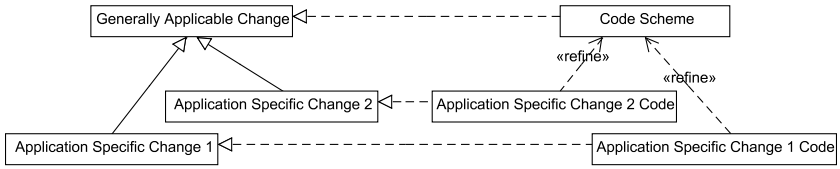

Figure 7.1: General and specific changes with realization.

When planning changes, it is more convenient to think in a domain specific

[1] This section is adapted from parts of my paper Evolution of Web Applications with Aspect-Oriented Design Patterns [1] to which my contribution is approximately 60 %

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 an SMTP server backup was needed. This is easily identified as a resource backup, which subsequently brings us to the realization in the form of class exchange.

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. The following list sums up these relationships between change types we have identified in the domain of web applications (in Chapter 6 and 5):

- One Way Integration (Section 6.1): Perform an Action After Event (Section 5.6)
- Two Way Integration (Section 6.1): Perform an Action After Event (Section 5.6)
- Adding a Column to a Grid (Section 6.2): Perform an Action After Event (Section 5.6)
- Removing a Column from a Grid (Section 6.2): Method Substitution (Section 5.2)
- Altering Column Presentation in a Grid (Section 6.2): Method Substitution (Section 5.2)
- Adding Fields to a Form (Section 6.3): Enumeration Modification (Section 5.3)
- Removing Fields from a Form (Section 6.3): Enumeration Modification (Section 5.3) with Additional Return Value Checking/Modification (Section 5.5)
- Introducing an Additional Constraint on Fields (Section 6.3): Additional Parameter Checking (Section 5.4)
- Introducing User Rights Management (Section 6.4): Border Control with Method Substitution (Section 5.2)
- User Interface Restriction (Section 6.5): Additional Return Value Checking/Modifications (Section 5.5)
- Introducing a Resource Backup (Section 6.6): Class Exchange (Section 5.1)

7.2 Implementing Changes of Changes

When implementing changes in object-oriented manner, changes become a part of the application. However, when changes are implemented using aspect-oriented techniques, they are easily distinguishable from the rest of application. When a change of change is required, it can be implemented in two ways:

- the change of change is implemented as new change (multi-layer model)
- the change which is supposed to be modified is modified using conventional techniques (two-layer model)

7.2.1 Multi-Layer Model

In the multi-layer model, each change is implemented as an independent aspect. This is depicted in Figure 7.2 in which Change 1.1.1 modifies Change 1.1 and Change 1.1 modifies Change 1. Change 1 was the first change that was implemented and modifies the application Class 1 and Class 2.

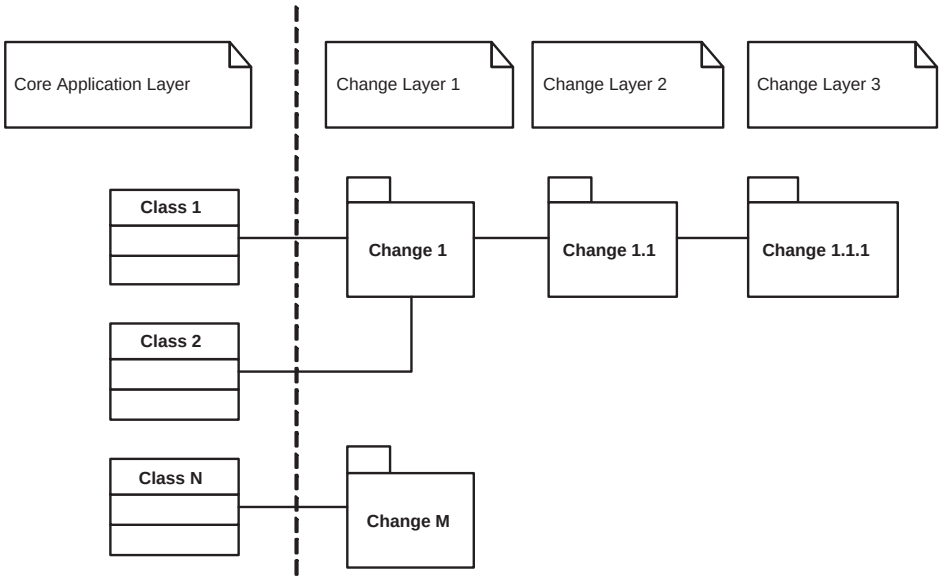

Figure 7.2: Multi-layer change model.

This approach provides better control over changes as changes are better separated, but it increases the complexity of application.

7.2.2 Two-Layer Model

In the two-layer model, the change of Change 1 is not implemented as an independent aspect, but it is implemented by changing source code of the Change 1. This is depicted in Figure 7.3 in which only two layers are shown. Core Application Layer represents classes of the application and Change Layer represents aspects, which implement changes.

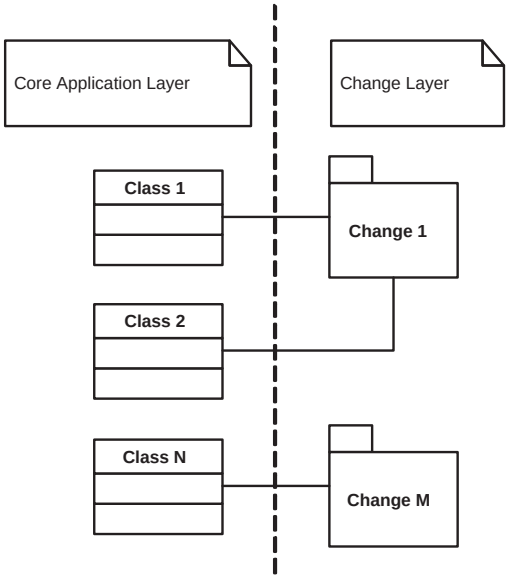

Figure 7.3: Two-layer change model.

This approach doesn't provide so precise control over changes as multi-layer model, but the complexity of application does not increase by implementing changes of changes.

Chapter 8

The Approach Evaluation

In this chapter, the approach to change control using aspect-otriented programming is evaluated by implementing and evaluating changes in the YonBan system, which is described in the Section 8.1. Implemted changes are described and evaluated in Section 8.2.

8.1 YonBan

YonBan is a system used for student management at Faculty of Informatics and Information Technologies on Slovak University of Technology in Bratislava. It is used for project (bachelor and master thesis ...) registration and project lifecycle management. It includes a module that enables uploading project results and documentation and also a module that enables project ranking.

YonBan was developed by students of Faculty of Informatics and Information Technologies as a Team Project in year 2006 [11]. This system is reimplementation of a system that is already in use at Faculty of Informatics and Information Technologies. YonBan is based on the J2EE and the Spring, Hibernate, and Acegi frameworks. The YonBan architecture is based on the Inversion Of Control and Model-View-Controller patterns. All details of YonBan will not be explained here. Instead, the parts of the system affected by the changes will be explained as needed.

8.2 Implemented Changes

The following changes have been implemented in YonBan using the proposed approach:

- telephone number validator
- telephone number formatter
- project registration statistics

- project registration constraint
- exception logging
- name formatter

These changes are described an evaluated in following Sections.

8.2.1 Telephone Number Validator

Suppose YonBan has to be extended with a telephone number validator in the user profile. This validator is supposed to check whether the entered number is a valid telephone number. This change corresponds to input form modification introduced in Section 6.3 as it modifies the way form submission is processed at the server.

YonBan is based on Model-View-Controller architecture. In this architecture, the controller processes responds to events (HTTP requests in web applications) and invokes changes in the model. On top of that, each controller in YonBan has a list of validators which are used to validate the submitted form fields. For the telephone number validator to be active, it has to be added to this list of validators.

The new validator (see Appendix A.1.2) and the aspect which adds this validator to the list of controller validators (see Appendix A.1.1) has to be implemented. This aspect is an implementation of the performing action after event change introduced in Section 5.6. Event in this case is the initialization of the controller bean. This initialization is captured by the following pointcut:

```
private pointcut controllerInitialization(SimpleDetailController controller) :
  execution(void sk.yonban.core.web.SimpleDetailController.afterPropertiesSet())
  && target(controller);
```

After the controllerInitialization() pointcut is captured, an after advice is executed. However, there is no specific controller for user profile as it uses the general class SimpleDetailController. Because of this, after advice has to check if initialized bean name equals “pouzivatelInfoController”, which is the name of the user profile controller bean:

```
after(SimpleDetailController controller) : controllerInitialization(controller) {
  if (!controller.getBeanName().equals("pouzivatelInfoController")) {
          return;
  }
  addValidator(controller);
}
```

Compared to the non aspect-oriented way of adding a validator to YonBan, the aspect-oriented approach provides better modularization by implementing change in two files and not modifying any existing file in the YonBan. Implementing telephone number validator using the non aspect-oriented approach includes creation of a new validator class, but on top of that it requires a change in two configuration files (validator.xml and controller.xml). The aspect-oriented

approach also enables further validators to be easily added to the controller class via aspects.

8.2.2 Telephone Number Formatter

In previous Scetion, the telephone number validator was implemented. This validator checks if the entered telephone number is a valid international telephone number. Most of the YonBan users do not enter their telephone number with an international calling code prefix and because of that, validator validates these numbers as incorrect (which is intended behavior). As most of YonBan users are from Slovakia, we could automatically add Slovak international calling code prefix to the number (if it is not already present) and thus making the system more user friendly.

User telephone number is being entered in a user profile form. Each form in YonBan has a corresponding command object which is filled by values submitted from this form. This command object has also methods, which allow it to load and save form data to database. To add a prefix to the telephone number, return value of getTelefon() method must be modified. This corresponds to the additional return value modification change (Section 5.5). The execution of the getTelefon() method is captured by the following pointcut:

```
private pointcut getTelefonPointcut():
  execution(String sk.yonban.model.command.PouzivatelInfoCommand.getTelefon());
```

After the getTelefonPointcut() is captured, an around advice which adds international calling code prefix to the telephone number is executed:

```
String around(): getTelefonPointcut() {
  number = proceed();
  processTelephoneNumber();
  return number;
}
```

Full implementation of the telephone number formatter can be found in Appendix A.2.

Compared to the non aspect-oriented way of adding the telephone number formatter to YonBan, aspect-oriented approach provides better modularization by implementing change in a separate file and not modifying the original YonBan source code.

The telephone number formatter change interacts with telephone number validator change (Section 8.2.1) as the telephone number validator validates the PouzivatelInfoCommand which is modified by telephone number formatter. This interaction causes no problems because the validator uses getTelefon() method to access telephone number submitted by the user profile form. Validation and formatting works like this: User enters telephone number without international calling code prefix and submits the user profile form. Upon the form submission the phone number validator is executed. It gets the telephone number value by calling the getTelefon() method. At this point, the telephone number validator around advice is executed. It calls the getTelefon() method

and checks if the return value (telephone number submitted by form) contains the international calling code prefix. If not, this code is appended to the telephone number and the new value is returned. The validator receives telephone number with international calling code prefix and validates it as correct. The telephone number is stored in the database (including the international calling code prefix as the method that stores number in the database uses getTelefon() method, too) and the form with the telephone modified number is shown back to the user.

8.2.3 Project Registration Statistics

Registering students to projects is one of the main functions of YonBan. Suppose we want to have statistics of how many students have been registered in a project during some period. This includes storing the number of students registered in a project upon each new registration or unregistration.

The project registration statistics module can be considered as a new system that has to be integrated with the YonBan. This corresponds to the one way integration change (Section 6.1). The project registration statistics module needs to be notified upon each new registration or unregistration. As described in Section 6.1, the integration change corresponds to the performing action after event change (Section 5.6).

The key point in introducing project registration statistics is to define a pointcut that would capture all changes in project registrations and to create an after advice that will be executed after these pointcuts are captured. This advice has to notify the project registration statistics module of the current number of users registered in a project.

There are two methods used for project registration in YonBan:

```
// register student "pouzivatel" in project "projekt"
public List<Error> registrujProjekt(Projekt projekt, Pouzivatel pouzivatel);

// unregister student "pouzivatel" from project "projekt"
public List<Error> odregistrujProjekt(Projekt projekt, Pouzivatel pouzivatel);
```

The pointcut that captures these two methods and the after advice which notifies Project Registration Statistics module can be found in an Appendix A.3.

The advantage of using the aspect-oriented approach compared to the non aspect-oriented approach is not so significant in this case. YonBan is well designed and all registrations are performed through two methods, so the non aspect-oriented implementation would require adding only one line of code to each one of these two methods. The advantage of the aspect-oriented approach would be much more significant if projects could be registered via multiple methods, which would require changes in every method. However, the advantage of better change modularization remains. The change is located only in one file and not spread throughout the project.

8.2.4 Project Registration Constraint

When a new student is imported to YonBan, he has no contact information filled in. If such a student registers to a project and has no contact information filled in, then the project supervisor will not able to contact him. Suppose we want to ensure that each student that registers to a project must have the email address filled in. Projects are registered through ProjektServiceImpl.registrujProjekt(Projekt projekt, Pouzivatel pouzivatel) method, which has already been identified in Section 8.2.3. To implement the desired change, we have to introduce a constraint on the method parameter pouzivatel. This corresponds to the additional parameter checking change (Section 5.4). The Pouzivatel.getEmail() method should not return an empty string (the email address is not filled). If the email address is empty, registrujProjekt() method should not be executed and the error should be returned. Execution of this method is captured by the the following pointcut:

```
private pointcut registerUser(Projekt projekt, Pouzivatel pouzivatel) :
  execution(List<Error> sk.yonban.service.ProjektServiceImpl.registrujProjekt(..))
  && args(projekt, pouzivatel);
```

After the registerUser() pointcut is captured, the around avice which checks user email address is executed:

```
List<Error> around(Projekt projekt, Pouzivatel pouzivatel) :
  registerUser(projekt, pouzivatel) {
  if (pouzivatel.getEmail().length() == 0) {
    return /* error message */;
  }
  return proceed(projekt, pouzivatel); // execute registrujProjekt() method
};
```

The full implementation of the Project Registration Constraint can be found in Appendix A.4.

The project registration constraint change interacts with the project registration statistics change (Section 8.2.3) as the project registration statistics after advice (Section A.3) advises the same method as the project registration constraint around advice (Section A.4). This interaction of changes causes no problems because if the around advice of project registration constraint change does not execute registrujProjekt() method, the after advice of project registration statistics is not executed, too, which is correct behaviour.

The advantages of the aspect-oriented approach compared to the non-aspect oriented approach are the same as with the project registration statistics change (Section 8.2.3).

8.2.5 Exception Logging

Suppose we want to have an exception logging in YonBan as exception logging can be very usefull when debugging the system. This change corresponds to the logging change (Section 5.7). Exception logging can be easily achieved by two

after advices (Appendix A.5). One advice captures and logs runtime exceptions, while the other one captures all other exceptions.

```
after() throwing(RuntimeException e): execution (* *.*(..)) {
  // log Exception
}

after() throwing(Exception e): execution(* *.*(..) throws *) {
  // log Exception
}
```

Compared to the non aspect-oriented approach, the aspect-oriented approach is much easier to implement. Implementing exception logging without aspect-oriented programming would require either to add logging to every exception constructor or to every catch block in the application. This would be very time consuming and also hard to remove in the case it is needed no more.

8.2.6 Name Formatter

Names are formatted different way in different countries. For example, in Japan and Hungary, in contrast to most other countries, the surname is placed before the given names. In YonBan, names are formatted as it is common in Slovakia (the given name is placed before the surname). Suppose we want to install YonBan in a country that uses different name formatting (e.g., Hungary). We need to find all places in the system where the names are being formatted. In YonBan this is done by the Pouzivatel.getCeleMeno() method. Non aspect-oriented approach would require a change of this method. Another possible solution is to have a system setting which would tell how the names should be formatted. By employing aspect-oriented approach this change can be regarded as a method substitution change (Section 5.2). The Pouzivatel.getCeleMeno() method has to be replaced by another method which formats the name in a desired way:

```
String around(Pouzivatel pouzivatel):
  execution(String sk.yonban.model.Pouzivatel.getCeleMeno()) && target(pouzivatel) {
    return getCeleMeno(pouzivatel);
}
```

The full implementation can be found in Appendix A.6.

8.3 Conclusion

In this chapter, the proposed approach to the aspect-oriented change implementation was evaluated. Six changes has been choosen and implemented. When the change was about to be implemented, a coresponding change was found out in the catalogue of web application specific changes. If this web application specific change was identified, the change was implemented the way it is described in this web application specific change.

However, some changes that were implemented had no corresponding web application specific changes. There are two reasons for this. First, the catalogue of web application specific changes is not complete and can be extended as new change types are identified. Second, the YonBan is based on a different architecture than the Affiliate tracking system which was used to identify web application specific changes. But even if the web application specific change could not be identified, there have always been a corresponding generally applicable change which described how to implement the change.

The result of this evaluation is that generally applicable changes can be used in different types of systems while web application specific changes are tightly coupled with the application architecture. Some web application specific changes identified in one type of systems may not applicable to the web applications based on different architecture. There could be several catalogues of web application specific changes each of which would be applicable to one type of web applications (e.g., Catalogue of web applications specifc changes for applications based on SpringMVC. This is the case of the YonBan system).

While implementing changes in YonBan, interaction of changes has been identified, too. No problems have been encountered even if changes affected same parts of system.

Chapter 9

The Seasar Framework for Aspect-Oriented Programming in PHP

To show that proposed approach to change implementation can be used also in less expressive aspect-oriented languages or frameworks, the Seasar framework is introduced. Reasons why the Seasar was selected among of other aspect-oriented extensions and Seasar's overview can be found in Section 9.1. Section 9.2 describes how control flow pointcuts required by some changes can be implemented in Seasar framework.

9.1 The Seasar Framework Overview

There are many PHP aspect-oriented frameworks and extensions such as PHPAspect, [1] Aspect-Oriented PHP, [2] and AOP Library for PHP [3]. Seasar framework [4] was selected because:

- it supports dynamic weaving (no need to install any PHP extensions)
- it supports dependency injection
- it is a multiplatform solution (avalaible also for Java and .NET)
- it is free and open source solution
- it was used in many projects

[1] `http://phpaspect.org/`
[2] `http://www.aophp.net/`
[3] `http://www.phpclasses.org/browse/package/2633.html`
[4] `http://www.seasar.org/en/php5/index.html`

9.1.1 Seasar Aspect-Oriented Programming Compared to the AspectJ Approach

The Seasar framework is based on the same aspect model as AspectJ, but it does not support all the features that AspectJ supports. The differences are explained in the following paragraphs.

Join points. In the Seasar framework, only method calls can be used as join points. Compared to AspectJ, it doesn't support field access and other join points.

Pointcuts The pointcut pattern in Seasar can consist only of regular expressions matching methods. Conjunction of these patterns is also allowed. Compared to the AspectJ, it doesn't support disjunction of patterns, **cflow**, nor cflowbellow pointcuts.

Advices In the Seasar, advice type is not explicitly defined. Each advice is of the around type. If advice doesn't do anything after the method execution, it can be considered to be a before advice. If advice doesn't do anything before method execution, it can be considered to be an after advice.

Compared to the AspectJ, Seasar doesn't support inter-type declarations nor compile-time declaration. Compile-time declaration can't be implemented since PHP is an interpreted language.

9.1.2 Seasar Basics

In the Seasar framework, aspects are defined in the S2Container configuration file (dicon file) [6]. There is no restriction on placement of the configuration file, but it is usually placed in the root folder of application.

Configuration file. Configuration file can contain an aspect tag. It weaves an aspect into a component. Interceptor is specified in a PHP statement in a BODY or in a component tag in a child tag. If several aspects are weaved into a component, they are processed in the order they have been declared.

Pointcut attribute. Several method names may be specified by delimiting the names with a comma. Not specifying any pointcut implies that all methods in an interface implemented by a component are affected. Also, regular expressions may be used to specify method names.

Example. To show how the Seasar Framework works a couple examples will be presented. These examples show basic features of Seasar framework.

The following is an example of an aspect which should advice getTime() method in Date class:

```
<?php
class Date {
  function Date() {}

  function getTime(){
    print 'getTime \n';
    return '12:00:30';
  }

  function getDate(){
    print 'getDate \n';
    return '25';
  }
}
?>
```

The following is an example that employs a regular expression to apply an aspect to all public methods in the Date class:

```
<component class="Date">
  <aspect pointcut=".*">
    <component class="MyInterceptor"/>
  </aspect>
</component>
```

The folowing is an example of the MyInterceptor aspect implementation:

```
<?php
class MyInterceptor implements MethodInterceptor {
  public function invoke(MethodInvocation $invocation){
    print "Before \n"; <-- Before invocation
    $ret = $invocation->proceed();
    print "After \n"; <-- After invocation
    return $ret;
  }
}
?>
```

The MethodInvocation object contains following properties:

```
$method;  <-- name of invocated method
$methodArgs; <-- arguments of invocated method
$target;  <-- target object of method invocation
```

When the following code is executed:

```
$date = $container->getComponent('Date');
print $date->getTime();
print "------------------------------ \n";
print $date->getDate();
```

The result is:

```
Before
getTime
```

```
After
12:00:30
------------------------------
Before
getDate
After
25
```

As it can be seen, the advice was applied to all methods of Date class. This is because the pointcut was specified by a regular expression .*.

If the configuration file is modified to following:

```
<component class="Date">
  <aspect pointcut="getDate">
    <component class="MyInterceptor"/>
  </aspect>
</component>
```

The result is:

```
getTime
12:00:30
------------------------------
Before
getDate
After
25
```

In this case, the advice was applied to getDate() method only.

9.2 Implementing More Complex Aspect-Oriented Constructs in Seasar

The Seasar framework does not support **cflow**(), and cflowbellow() pointcuts. However, when implementing changes, these pointcuts might be necessary. Here, a way to implement "cflow pointcut" in the Seasar framework will be proposed It will be shown in the following example. Suppose we want to capture all calls of Affiliate.insert() method that were called from AffiliateSignup.processSignup() method. To accomplish this, one additional advice has to be created. This advice will be an around advice of AffiliateSignup.processSignup() method and will look as follows:

```
class AffiliateSignupControlFlowInterceptor implements MethodInterceptor {
  public static $invocationCount = 0;

  public static function isInControlFlow() {
    return (self::$invocationCount > 0);
  }

  public function invoke(MethodInvocation $invocation){
```

```
    self::$invocationCount++;
    $ret = $invocation->proceed();
    self::$invocationCount--;
    return $ret;
  }
}
```

This around advice increments the $invocationCount on each call of the AffiliateSignup.processSignup() method and decrements it on return from this method. If the $invocationCount counter is greater than zero, execution is in the control flow the AffiliateSignup.processSignup() method.

In the advice for Affiliate.insert() method it is necessary to check whether it was called in the control flow of AffiliateSignup.processSignup() method. If not, proceed() is called and the advice code is not used. This advice will look as follows:

```
class AffiliateInsertInterceptor implements MethodInterceptor {

  public function invoke(MethodInvocation $invocation){
    if (!AffiliateSignupControlFlowInterceptor::isInControlFlow()) {
      return $invocation->proceed();
    }
    // ...
    // advice code
    // ...
  }
}
```

This chapter showed that even more complex aspect-oriented constructs such as control flow pointcuts that are needed by changes identified in Chapter 5 can be implemented in less expressive aspect-oriented languages or frameworks and thus the proposed approach is not limited to AspectJ.

Chapter 10

Conclusions and Further Work

In this thesis, the possibility of expressing changes in web applications by means of aspect-oriented programming was analyzed. On the basis of this analysis, an approach to web application evolution was introduced. In this approach, changes are represented as aspects and applied as instantiations of generic aspect-oriented change types. Several change types which can occur in web applications as evolution or customization steps, and correspondence between these change types and generic aspect-oriented change types were identified. This approach shows that such changes can be managed as aspects. The implementation and evaluation of changes shows that the proposed approach is applicable for change implementation in web applications and that generally applicable changes can be used in different types of systems while web application specific changes are tightly coupled with the application architecture. Some web application specific changes identified in one type of systems may not applicable to the web applications based on different architecture. There could be several catalogues of web application specific changes each of which would be applicable to one type of web applications.

To show that proposed approach to change implementation is not limited to AspectJ and can be used also in less expressive aspect-oriented languages or frameworks, the Seasar framework was introduced and techniques for implementing advanced aspect-oriented constructs in this framework were developed.

Further work should focus mainly on evaluating more interactions between changes, extending current catalogue and creating more catalogues of web application specific changes focused on a specific types of web applications.

Bibliography

[1] 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.

[2] Walter Cazzola, Sonia Pini, and Massimo Ancona. AOP for software evolution: A design oriented approach. In *2005 ACM Symposium on Applied Computing*, pages 1346–1350, Santa Fe, New Mexico, USA, 2005. ACM.

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

[4] Simplice Djoko Djoko, RA©mi Douence, Pascal Fradet, Didier LeBotlan, and Mario SA1dholt. CASB : Common Aspect Semantics Base, 2006.

[5] Peter Dolog, Valentino VraniŠ, and Mßria Bielikovß. Representing change by aspect. *ACM SIGPLAN Notices*, 36(12), December 2001.

[6] The Seasar Foundation. Seasar Home Page. `http://www.seasar.org/`.

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

[8] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[9] 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.

[10] Ramnivas Laddad. *AspectJ in Action*. Manninng, 2003.

[11] Ján Šarmír Peter Šimno Lukáš Šimon Jakub Vaňo Martin Nágl, Jozef Slezák. *Student Project Life Cycle Software Support.* Team project documentation, Slovak University of Technology in Bratislava, Slovakia, 2006. In Slovak. Available at `http://www2.dcs.elf.stuba.sk/TeamProject/2006/team09/`.

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

[13] Odysseas Papapetrou and George A. Papadopoulos. Aspect oriented programming for a componentbased real life application: A case study. In *2004 ACM Symposium on Applied Computing*, pages 1554–1558, Nicosia, Cyprus, 2004. ACM.

Appendix A

Implementation of YonBan changes

A.1 Telephone Number Validator

A.1.1 Aspect

```
package sk.yonban.core.aspectchanges;

import org.springframework.validation.Validator;
import sk.yonban.core.web.SimpleDetailController;

public aspect PouzivatelInfoValidatorAspect {
  private pointcut controllerInitialization(SimpleDetailController controller) :
    execution(void sk.yonban.core.web.SimpleDetailController.afterPropertiesSet())
    && target(controller);

  after(SimpleDetailController controller) : controllerInitialization(controller)
  {
    if (!controller.getBeanName().equals("pouzivatelInfoController")) {
      return;
    }
    addValidator(controller);
  }

  private void addValidator(sk.yonban.core.web.SimpleDetailController controller) {
    Validator validators[] = controller.getValidators();
    Validator newValidators[] = new Validator[validators.length+1];
    for (int i=0; i<validators.length; i++) {
      newValidators[i] = validators[i];
    }
    newValidators[newValidators.length-1] = new PouzivatelInfoValidator();
    controller.setValidators(newValidators);
  }
```

```
}
```

A.1.2 Validator Class

```
package sk.yonban.core.aspectchanges;

import org.springframework.validation.Errors;

import sk.yonban.model.command.PouzivatelInfoCommand;
import sk.yonban.validation.BaseValidator;

public class PouzivatelInfoValidator extends BaseValidator {

  private static final String ERROR_POUZIVATEL_INFO_TELEFON_ERROR
    = "error.pouzivatelInfo.telefonError";

  public boolean supports(Class commandClass) {
    return PouzivatelInfoCommand.class.equals(commandClass);
  }

  public void validate(Object cmd, Errors errors) {
    PouzivatelInfoCommand command = (PouzivatelInfoCommand) cmd;
    if (command == null || !validatePhoneNumber(command.getTelefon())) {
      errors.rejectValue("telefon", ERROR_POUZIVATEL_INFO_TELEFON_ERROR);
    }
  }

  private boolean validatePhoneNumber(String telefon) {
    /* Telephone number validation */
  }
}
```

A.2 Telephone Number Formatter

```
package sk.yonban.aspectchanges;

public aspect TelephoneNumberFormatter {
  private pointcut getTelefonPointcut():
    execution(String sk.yonban.model.command.PouzivatelInfoCommand.getTelefon());

  private String number;

  String around(): getTelefonPointcut() {
    number = proceed();
    processTelephoneNumber();
    return number;
  }

  private void processTelephoneNumber() {
    if (number.startsWith("+") || number.startsWith("00")) {
```

```
      return;
    }
    if (number.startsWith("0")) {
      number = number.substring(1);
      number = "+421" + number;
    }
  }
}
```

A.3 Project Registration Statistics

```
package sk.yonban.aspectchanges;

import sk.yonban.model.Projekt;
import sk.yonban.model.Pouzivatel;

public aspect RegistrationStatistics {
  after(Projekt projekt, Pouzivatel pouzivatel) :
    execution(* sk.yonban.service.ProjektServiceImpl.registrujProjekt(..))
    && args(projekt, pouzivatel) {

    // save project registration (pouzivatel, projekt) //
  }

  after(Projekt projekt, Pouzivatel pouzivatel) :
    execution(* sk.yonban.service.ProjektServiceImpl.odregistrujProjekt(..))
    && args(projekt, pouzivatel) {

    // save project unregistration (pouzivatel, projekt) //
  }
}
```

A.4 Project Registration Constraint

```
package sk.yonban.aspectchanges;

import java.util.ArrayList;
import java.util.List;
import sk.yonban.model.Pouzivatel;
import sk.yonban.model.Projekt;

public aspect RegistrationConstraint {
  private pointcut registerUser(Projekt projekt, Pouzivatel pouzivatel) :
    execution(List<Error> sk.yonban.service.ProjektServiceImpl.registrujProjekt(..))
    && args(projekt, pouzivatel);

  List<Error> around(Projekt projekt, Pouzivatel pouzivatel) :
    registerUser(projekt, pouzivatel) {

    if (pouzivatel.getEmail().length() == 0) {
```

```
    List<Error> errors = new ArrayList<Error>();
      errors.add(new Error("User email can not be empty"));
      return errors;
    }
    return proceed(projekt, pouzivatel);
  }
}
```

A.5 Exception Logging

```
package sk.yonban.aspectchanges;

public aspect ExceptionLogger {
  after() throwing(RuntimeException e): execution (* *.*(..)) {
    // log RuntimeException
  }

  after() throwing(Exception e): execution(* *.*(..) throws *) {
    // log Exception
  }
}
```

A.6 Name Formatter

```
package sk.yonban.aspectchanges;

import sk.yonban.model.Pouzivatel;

public aspect NameFormatter {

  String around(Pouzivatel pouzivatel):
    execution(String sk.yonban.model.Pouzivatel.getCeleMeno())
    && target(pouzivatel) {
      return getCeleMeno(pouzivatel);
  }

  private String getCeleMeno(Pouzivatel pouzivatel) {
    // do new name formatting
    return pouzivatel.getPriezvisko() + " " + pouzivatel.getMeno();
  }
}
```

Appendix B

Attached CD-ROM Contents

The attached CD-ROM contains the following files and directories:

- /doc/AOChangeInWebApplications.pdf — this thesis
- /doc/AOChangeArticle.pdf — Evolution of Web Applications with Aspect-Oriented Design Patterns
- /src/AOChange/ — Source code of the examples from Chapters 5 and 6
- /src/YonBan/ — Source code of the YonBan changes from Chapter 8 and Appendix A

The /src/AOChange/ directory is an Eclipse project. To build it, the following is required:

- Eclipse[1]
- AspectJ Developement Tools[2]

The changes from /src/YonBan/ directory can be directly copied to the YonBan project and will be applied right after the project is converted to AspectJ project.

[1] `http://www.eclipse.org/downloads/`
[2] `http://www.eclipse.org/ajdt/downloads/`

Appendix C

Evolution of Web Applications with Aspect-Oriented Design Patterns

Based on the main results of this Master's thesis, the following paper has been written:

> 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, July 19, 2007, Como, Italy.

My contribution to this paper is approximately 60%. I contributed mainly to Sections 1–4.

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,
Ilkovičcova 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.

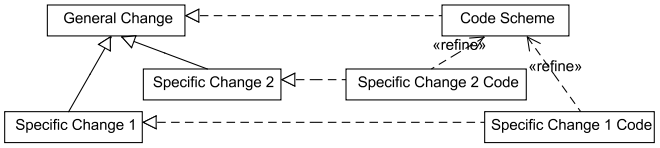

Fig. 1. General and specific changes with realization.

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.