Evolution of Web Applications with Aspect-Oriented Design Patterns

ICWE 2007 — Como, Italy
July 19, 2007

Michal Bebjak[1] **Valentino Vranić**[1] Peter Dolog[2]

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

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

Overview

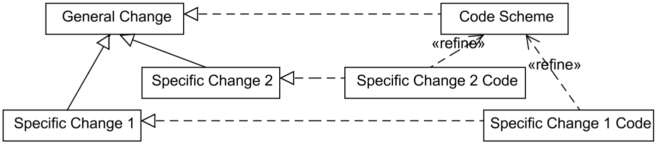
Changes

- Changes are inseparable part of software evolution
- Change implementation: expensive and slow
- A change often affects many places in the code
  - E.g., modification of selected calls of the given method
- Even if it affects a single place, we may want to keep it separate
  - To be able to revert it and reapply it
  - Especially useful in the customization of web applications
- Thus, changes can be seen as crosscutting concerns

Aspect-Oriented Programming

- Crosscutting concerns are implemented as aspects
- Variety of aspect-oriented approaches and languages
- AspectJ is the most widely used and influential aspect-oriented language
- The key issue is to identify and specify places where the crosscutting code affects the rest of the code
- Such places are called *join points* and they are specified by *pointcuts*
- Additional behavior to be performed before, after, or instead of join points is specified in *advices*
- *Inter-type declarations* enable introduction of new members into existing types, as well as introduction of compile warnings and errors

Our Approach



- Two levels of changes:
  - Application specific changes: high-level change specifications
  - General changes: change realizations
- A change to be applied is selected by its specification
  - An example:
    A backup SMTP server is needed ⇒Introducing a Resource Backup ⇒Class Exchange
- Application specific changes are implemented by adapting code schemes of the corresponding general changes

Catalogue of Changes

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

Catalogue of Changes

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

When a Change Gets Changed. . .

- Technically, it is possible for an aspect in AspectJ to affect another aspect
- Subsequent changes can be implemented in separate aspects
- Sometimes it is more desirable to either
  - transform aspect-oriented implementation of the former changes into object-oriented one, or
  - implement a subsequent change by modifying the aspect code

Other Aspect-Oriented Approaches

- No need to switch to aspect-oriented language: many frameworks support aspect-oriented programming
- We studied the Seasar PHP framework
- It is usable, though limited: supports only method calls as join points
- Seasar misses some more sophisticated primitive pointcuts: we implemented **cflow**() and **cflowbelow**()

General Change Types

- Many useful design patterns already have been identified valid for AspectJ and alike
- A successful aspect-oriented change implementation requires a structured base application
- We identified several general change types expressed in programming language terms:
    - Class Exchange
    - Method Substitution
    - Enumeration Modification
    - Additional Parameter Checking
    - Additional Return Value Checking/Modification
    - Perform an Action After an Event
- Implementation of these changes is based on aspect-oriented design patterns like Cuckoo's Egg, Policy, etc.

Class Exchange

- Sometimes, a class has to be exchanged with another one either in the whole application, or in a part of it
- Based on the Cuckoo's Egg design pattern[1]

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

    ExchangedClass around(): exhangedClassConstructor() {
        return getExchangingObject(proceed());
    }

    ExchangedClass getExchangingObject(ExchangedClass obj) {
        if (. . .) { return obj; }
        else { return new ExchangingClass(); }
    }
}
```

[1] The code scheme in the paper is incorrect. This is the correct version.

Introducing a Resource Backup

- Suppose we're adapting an affiliate marketing application
- One of its features is sending notifications about new sales, signed up affiliates, etc.
- We would like to have a backup SMTP server for sending notifications
- This change can be implemented as a Class Exchange change

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

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

Web Application Specific Changes

- High-level change descriptions
- We identified the following web application specific change types:
    - integration changes
    - grid display changes
    - input form changes
    - user rights management changes
    - user interface adaptation
    - resource backup

Conclusions

- We proposed a new approach to web application evolution
- Two levels: specification and realization
- Changes are implemented by aspect-oriented design patterns and program schemes
- We identified several change types in web applications
- We also proposed principles of an aspect-oriented change realization framework

Further Work

- Search for further change types and their realizations
- Explore change interactions and evaluate the approach practically
- Further work on the catalogue of changes
- Deal with changes at the modeling level: integrate aspect-oriented modeling approaches with web modeling approaches