Aspect-Oriented Change Realization

Erasmus Mobility at Lancaster University

Lecture 1

Valentino Vranić

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

September 16–19, 2008

Overview

1. The Only Constant. . .
2. Changes as Crosscutting Concerns
3. Catalog of Changes
4. Changing a Change
5. Evaluation
6. Summary

Changes

- Change is the only constant in software development (and elsewhere, too)
- Change realization is expensive and slow
- Code modifications are usually tracked by a version control tool
- But the logic of a change as a whole vanishes without a proper support in the programming language itself

Changes as Aspects

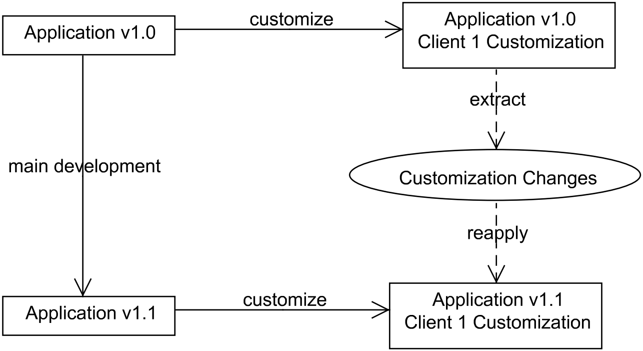
- Aspect-oriented programming enables to deal with change[1] [2] explicitly and directly at programming language level
- The logic of a change is modularized
- Changes implemented by aspects are
  - pluggable
  - reapplicable to similar applications (e.g., in a product line)

[1] V. Vranić, M. Bebjak, R. Menkyna, and P. Dolog. Developing Applications with Aspect-Oriented Change Realization. Accepted to *3rd IFIP TC2 Central and East European Conference on Software Engineering Techniques CEE-SET 2008*, October 2008, Brno, Czech Republic.

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

Motivating Example

- Customization of web applications
- A new version of the base application requires reapplication of the customization changes at the client side



Change Requests as Crosscutting Requirements

- A change is initiated by a change request
  - Specified in domain notions
  - Tends to be focused, but usually consists of several requirements
- By abstracting and generalizing the essence of a change, a *change type* can be identified
- Such a change type is applicable to a range of applications of the same domain

Crosscutting Nature of Change Realizations

- 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

Example Scenario

- Aspect-oriented change realization will be presented on an example scenario
- A merchant who runs his online music shop purchases a general affiliate marketing software to advertise at third party web sites (affiliates)
- Simplified affiliate marketing scheme:
  - A customer visits an affiliate's site which refers him to the merchant's site
  - When the customer buys something from the merchant, the provision is given to the affiliate who referred the sale
- Affiliate marketing software has to be adapted (customized) to the merchant's needs through a series of changes
- Assume the affiliate marketing software is written in Java
- We will use AspectJ to implement changes

Aspect-Oriented Programming and AspectJ

- 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

Domain Specific Changes

- Example: adding a backup SMTP server to ensure delivery of the notifications to users
  - Each time the affiliate marketing software needs to send a notification, it creates an instance of the SMTPServer class which handles the connection to the SMTP server
- A generalization:
  - An SMTP server is a kind of a resource that needs to be backed up
  - In general, it's a kind of *Introducing Resource Backup*
  - Abstract, but still expressed in a *domain specific* way—a *domain specific change type*

Domain Specific Change Implementation (1)

- The crosscutting concern identified: maintaining a backup resource that has to be activated if the original one fails
- Can be implemented in a single aspect without modifying the original code

Domain Specific Changes

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

Domain Specific Change Implementation (2)

- If we abstract from SMTP servers and resources altogether, it’s actually a class exchange
- *Class Exchange* change type based on the *Cuckoo’s Egg* aspect-oriented design pattern [3]

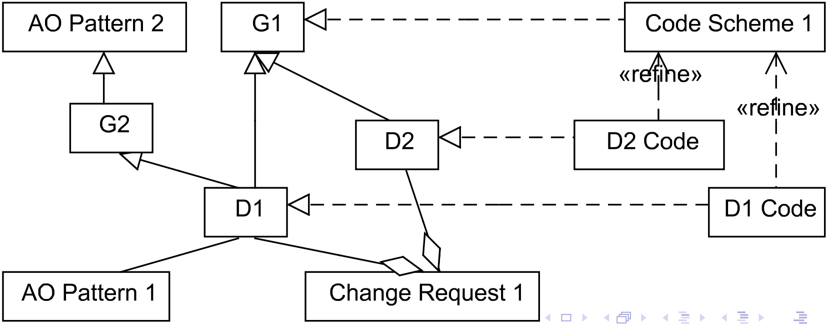
```
public class AnotherClass extends MyClass {
    . . .
}
public aspect MyClassSwapper {
    public pointcut myConstructors(): call(MyClass.new());
    Object around(): myConstructors() {
        return new AnotherClass();
    }
}
```

- *Class Exchange* is a *generally applicable* change type

[3] R. Miles. *AspectJ Cookbook*. O’Reilly, 2004.

Applying a Change Type

- How to give a hint to developer to use Cuckoo's Egg for Resource Backup?
- We have to maintain a catalog of changes
- Each domain specific change type is defined as a specialization of one or more generally applicable changes



Applying a Change Type

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

The Catalog of Changes in Web Application Domain (1)

- Integration Changes
  - One Way Integration: Performing Action After Event
  - Two Way Integration: Performing Action After Event
- Grid Display Changes
  - Adding Column to Grid: Performing Action After Event
  - Removing Column from Grid: Method Substitution
  - Altering Column Presentation in Grid: Method Substitution

The Catalog of Changes in Web Application Domain (2)

- Input Form Changes
  - Adding Fields to Form: Enumeration Modification with Additional Return Value Checking/Modification
  - Removing Fields from Form: Additional Return Value Checking/Modification
  - Introducing Additional Constraint on Fields: Additional Parameter Checking or Performing Action After Event
- Introducing User Rights Management: Border Control with Method Substitution
- User Interface Restriction: Additional Return Value Checking/Modifications
- Introducing Resource Backup: Class Exchange

Integration Changes (1)

- The affiliate marketing application has to be integrated with the newsletter software
- Newsletter has to be delivered to all affiliates
  - After an affiliate signs up, he should be added to the newsletter
  - After deletion of the affiliate account, the affiliate should be removed form the newsletter
- This corresponds to *Performing Action After Event*
- Since events are actually represented by methods, the desired action can be implemented in an after advice:

```
public aspect PerformActionAfterEvent {
    pointcut methodCalls(/∗ arguments ∗/): . . .;
    after(/∗ arguments ∗/): methodCalls(/∗ arguments ∗/) {
        performAction(/∗ arguments ∗/);
    }
    private void performAction(/∗ arguments ∗/) { /∗ action logic ∗/ }
}
```

Integration Changes (2)

- Multiple one way integrations can be seamlessly combined to integrate with several systems
- *Two Way Integration* can be seen as a double One Way Integration
- Useful in data synchronization
- Introducing a forum for affiliates with synchronized user accounts for affiliate convenience would represent a Two Way Integration

Introducing User Rights Management (1)

- A restricted administrator account is needed in our affiliate marketing application
- It should prevent the administrator from declining and deleting affiliates, and modifying the advertising campaigns and banners integrated with the web sites of affiliates
- This is an instance of *Introducing User Rights Management*

Introducing User Rights Management (2)

- Suppose all the methods for managing campaigns and banners are located in the campaigns and banners packages—a region prohibited to the restricted administrator
- The *Border Control* design pattern enables to partition an application into regions implemented as pointcuts

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

Introducing User Rights Management (3)

- We need to substitute the calls to the methods in the region with our own code that will let the original methods execute only if the current user has sufficient rights
- This can be achieved by applying *Method Substitution*
- An around advice is applied to the method call capturing pointcut to create a new logic of the methods to be substituted:

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

User Interface Restriction (1)

- It is quite annoying when a user sees, but can't access some options due to user rights restrictions
- *User Interface Restriction* should be applied
- The previous change introduced such a problem: since the restricted administrator can't access advertising campaigns and banners, he shouldn't see them in menu either

User Interface Restriction (2)

- Menu items are retrieved by a method
- To remove the banners and campaigns items, its return value should be modified →*Additional Return Value Checking/Modification*

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

Grid Display Changes (1)

- In web applications, data are often displayed in grids, and data input is usually realized via forms
- Typical changes required on a grid are
  - *Adding Column to Grid*
  - *Removing Column from Grid*
  - *Altering Column Presentation in Grid*
- If the grid is hard coded, it is difficult or even impossible to modify it using aspect-oriented techniques
- If the grid is implemented as a data driven component, we just have to modify the data passed to the grid, i.e. apply Additional Return Value Checking/Modification change
- Otherwise, a grid must be implemented either as some kind of a reusable component or generated by row and cell processing methods

Grid Display Changes (2)

- *Adding Column to Grid* can be performed *after an event* of displaying the existing columns →Performing Action After Event
- Note that the database has to reflect the change, too
- *Removing Column from Grid* requires a conditional execution of the method that displays cells →Method Substitution change

Grid Display Changes (3)

- Alterations of a grid are often necessary due to software localization
- E.g., in some occasions the surname has to be placed before the given names
- *Altering Column Presentation in Grid* requires preprocessing of all the data to be displayed in a grid before actually displaying them

Grid Display Changes (4)

- Altering Column Presentation in Grid may be easily achieved by modifying the way the grid cells are rendered →Method Substitution:

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

Input Form Changes

- Forms are often subject to modifications
  - *Adding Fields to Form*
  - *Removing Fields from Form*
  - *Introducing Additional Constraint on Fields*
- Precondition is that forms are generated (typically from a list of fields implemented by an enumeration), not hard coded in HTML
- In our scenario, assume the genre of the music promoted by affiliates has to be followed
- The genre field has to be added to the generic affiliate sign-up form and profile form →*Adding Fields to Form*
- To display the required information, we need to modify the affiliate table of the merchant panel to display genre in a new column
  - Enumeration Modification enables to add the genre field
  - Additional Return Value Checking/Modification must be used to modify the list of fields being returned

Enumeration Modification (1)

- The realization of *Enumeration Modification* depends on the enumeration type implementation
- Enumeration types are often represented as classes with a static field for each enumeration value

```
public class Genre {
    public static GenreType POP = new GenreType(1, "pop");
    public static GenreType ROCK = new GenreType(2, "rock");

    public ArrayList getGenreTypes() {
        ArrayList types = new ArrayList();
        types.add(POP);
        types.add(ROCK);
        return types;
    }
}
```

Enumeration Modification (2)

```
public class GenreType {
    public int id;
    public String name;

    public GenreType(int id, String name) {
        super();
        this.id = id;
        this.name = name;
    }
    public String toString() {
        return "["+id+","+name+"]";
    }
}
```

Enumeration Modification (3)

- We add a new enumeration value by introducing the corresponding static field:

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

Enumeration Modification (4)

- In our example:

```
public aspect NewGenre {
    // new static member of Genre class
    public static GenreType Genre.NEWGENRE =
        new GenreType(10, "new genre name");
    pointcut getGenreTypePointcut(): call(* Genre.getGenreTypes(..));
    private ArrayList retValue;

    ArrayList around() : getGenreTypePointcut() {
        retValue = proceed(); // execute original function
        processOutput();
        return retValue; // return modified output
    }
    private void processOutput() {
        retValue.add(Genre.NEWGENRE);
        // processing logic
    }
}
```

Enumeration Modification (5)

- The fields in a form are generated according to the enumeration values
- The list of enumeration values is typically accessible via a method provided by it
- This method has to be addressed by an Additional Return Value Checking/Modification change
- An Additional Return Value Checking/Modification change is sufficient to remove a field from a form
- Actually, the enumeration value would still be included in the enumeration, but this would not affect the form generation

Introducing Additional Constraint on Fields (1)

- Additional validations on the form input data to the system without a built-in validation →*Additional Parameter Checking* applied to methods that process values submitted by the form
- Key issue in Additional Parameter Checking is the pointcut: it has to capture all the calls of the affected methods along with their parameters

Introducing Additional Constraint on Fields (2)

- An around advice checks whether parameters are correct:

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

- Adding a new validator to a system that already has built-in validation is realized by simply adding it to the list of validators →Performing Action After Event: add the validator to the list of validators after the list initialization

Implementing a change of a change

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

Feasibility

- The possibility to implement a change of a change using aspect-oriented programming and without modifying the original change is given by the aspect-oriented programming language capabilities
- E.g., advices in AspectJ
  - Unnamed, so can't be referred to directly
  - **adviceexecution**() can be restricted by **within**() to a given aspect
  - If an aspect contains several advices, they have to be annotated and accessed by the **@annotation**() pointcut
  - This was impossible in AspectJ versions that existed before Java was extended with annotations

Aspect-Oriented Refactoring

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

YonBan

- We have successfully our approach to introduce changes into YonBan, a student project management system developed at Slovak University of Technology
- YonBan is based on J2EE, Spring, Hibernate, and Acegi frameworks with its architecture based on Inversion Of Control and MVC
- We implemented the following changes in YonBan:
  - Telephone number validator as Performing Action After Event
  - Telephone number formatter as Additional Return Value Checking/Modification
  - Project registration statistics as One Way Integration
  - Project registration constraint as Additional Parameter Checking/Modification
  - Exception logging as Performing Action After Event
  - Name formatter as Method Substitution
- No original code of the system had to be modified

Change Interaction

- We encountered one change interaction: between the telephone number formatter and validator
- These two changes are interrelated
    - They would probably be part of one change request
    - No surprise they affect the same method
    - No intervention was needed

Tool Support

- We managed to implement the changes easily even without a dedicated tool
- To cope with a large number of changes, such a tool may become crucial
- Even general aspect-oriented programming support tools may help
- AJDT for Eclipse
    - Shows whether a particular code is affected by advices, the list of join points affected by each advice, and the order of advice execution—important to track when multiple changes affect the same code
    - Advices that do not affect any join point are reported in compilation warnings—helps detect pointcuts invalidated by direct modifications of the application base code

The Need for a Dedicated Tool

- A change implementation can consist of several aspects, classes, and interfaces (*types*)
- The tool should keep a track of all the parts of a change
  - Some types may be shared among changes
  - Should enable simple inclusion and exclusion of changes
- Inclusion and exclusion of changes is related to change dependencies
- E.g., a change may require another change or two changes may be mutually exclusive
- But dependencies can be complex as feature dependencies in feature modeling

Feature Modeling

- Dependencies could be represented by feature diagrams and additional constraints
- Some dependencies between changes may exhibit only recommending character
- E.g., features that belong to the same change request
- Again, feature modeling can be used to model such dependencies with default dependency rules

Related Work (1)

- Maintaining change dependencies with feature modeling is similar to constraints and preferences in SIO software configuration management system[4]
- Fazekas proposed an approach that enables a kind of aspect-oriented programming on top of a versioning system[5]
    - Parts of the code that belong to one concern are marked manually in the code
    - They can be easily plug in or out
    - But concerns remain tangled in code

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

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

Related Work (2)

- Several other generally related issues have been explored
- Database schema evolution with aspects[6]
- Aspect-oriented extensions of business processes and web services with crosscutting concerns of reliability, security, and transactions[7]

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

[7] A. Charfi et al. Reliable, secure, and transacted web service compositions with AO4BPEL. In *4th IEEE European Conf. on Web Services (ECOWS 2006)*, Zürich, Switzerland, Dec. 2006. IEEE Computer Society.

Related Work (3)

- Increased changeability of components has been reported if they are implemented using
  - Aspect-oriented programming as such[8]
  - Aspect-oriented programming with the frame technology[9]
- Enhanced reusability and evolvability of design patterns has been achieved by using generic aspect-oriented languages to implement them[10]

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

[9] N. Loughran et al. Supporting product line evolution with framed aspects. In *Workshop on Aspects, Componentsand Patterns for Infrastructure Software (held with AOSD 2004, International Conference on Aspect-Oriented Software Development)*, Lancaster, UK, Mar. 2004.

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

Related Work (4)

- Other issues related to, but beyond the work presented here include
  - Automatic adaptation in application evolution, such as event triggered evolutionary actions[11]
  - Evolution based on active rules[12]
  - Adaptation of languages instead of software systems[13]

[11] 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.

[12] 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.

[13] J. Kollár et al. Functional approach to the adaptation of languages instead of software systems. *Computer Science and Information Systems Journal (ComSIS)*, 4(2), Dec. 2007.

Summary

- An approach to change realization using aspect-oriented programming
- Dealing with changes at two levels: domain specific and generally applicable change types
- Change types specific to web application domain along with corresponding generally applicable changes
- Consequences of having to implement a change of a change
- Evaluation of the approach has shown the approach can be applied even without a dedicated tool support
- But tool support is important in dealing with change dependencies
- This is a subject of our ongoing research