Developing Applications with Aspect-Oriented Change Realization

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

Institute of Informatics and Software Engineering
Faculty of Informatics and Information Technologies
Slovak University of Technology, Bratislava, Slovakia
vranic@fiit.stuba.sk, mbebjak@gmail.com, radu@ynet.sk

Department of Computer Science
Aalborg University, Aalborg, Denmark
dolog@cs.aau.dk

CEE-SET 2008, October 13–15, 2008, Brno, Czech Republic

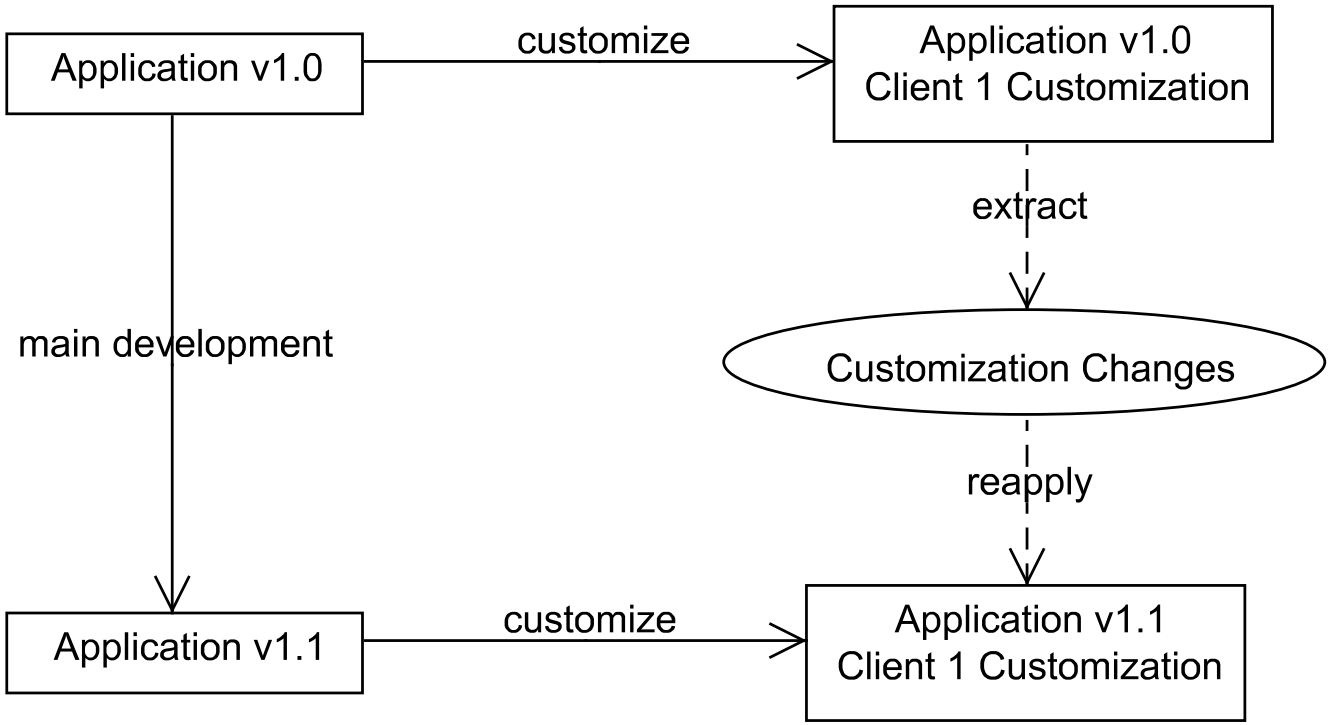
Overview

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

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 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

Why Aspect-Oriented Programming?

- Aspect-oriented programming enables to deal with change explicitly and directly at programming language level
- The logic of a change is modularized
- Changes implemented by aspects are pluggable and reapplicable to similar applications (e.g., in a product line)
- Increased changeability of components has been reported if they are implemented using
  - Aspect-oriented programming as such[1]
  - Aspect-oriented programming with the frame technology[2]
- Enhanced reusability and evolvability of design patterns has been achieved by using generic aspect-oriented languages to implement them[3]

[1] 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.

[2] 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.

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

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 passw
        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 [4]

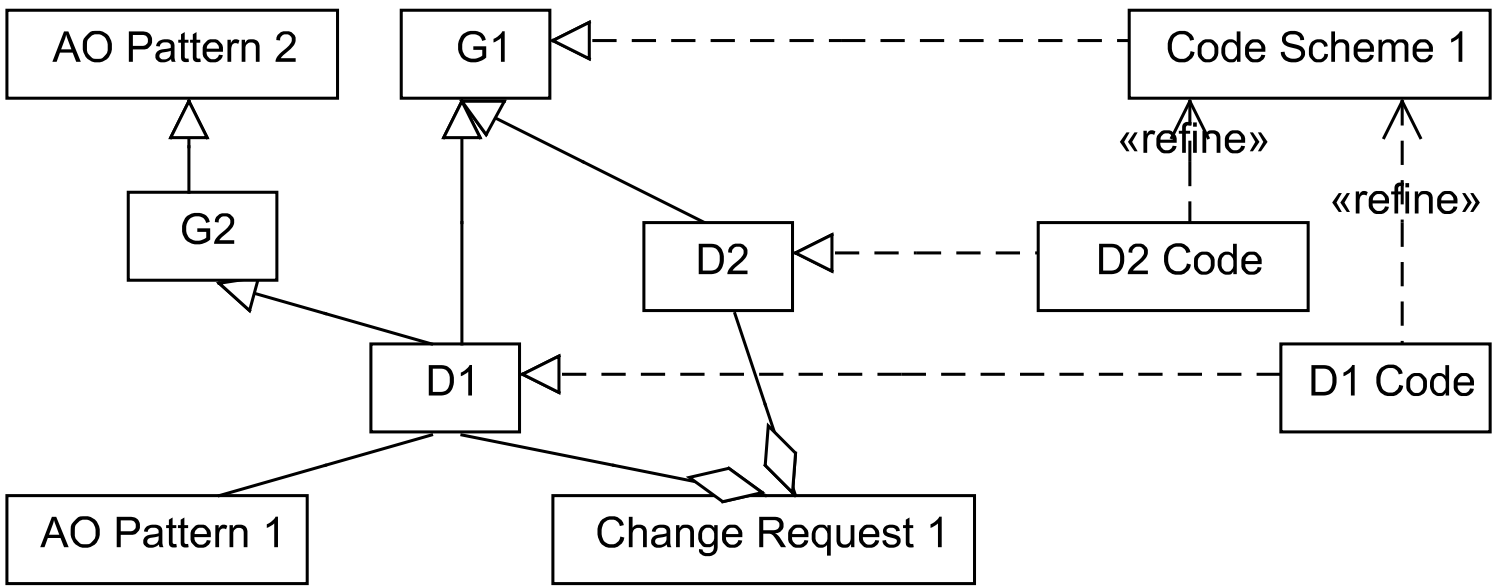
```
public class AnotherClass extends MyClass {
    ...
}
public aspect MyClassSwapper {
    public pointcut myConstructors(): call(MyClass.new());
    Object around(): myConstructors() {
        return new AnotherClass();
    }
}
```

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

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

Applying a Change Type

- How to give a hint to developer to use a particular change type?
- 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)

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

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

Changes in Affiliate Marketing Scenario

- Integrate with a newsletter: One Way Integration
- Forum for affiliates: Two Way Integration
- Add restricted administrator account: Border Control and Method Substitution
- Remove menu items in restricted administrator account: Additional Return Value Checking/Modification
- Add the genre field to the affiliate table: Adding Column to Grid
- Add the genre field to the generic affiliate sign-up form and his profile form: Adding Fields to Form

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) is actually 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 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

- The approach succesfully applied 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
- The following changes have been implemented 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 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

- Changes can be considered as features
- 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) — default dependency rules
- This is related to the approach for change impact analysis of aspectual requirements based on concern slicing[5]
- Maintaining change dependencies with feature modeling is similar to constraints and preferences in SIO software configuration management system[6]

[5] S. O. Rashid, et al. Approach for Change Impact Analysis of Aspectual Requirements. AOSD-Europe Deliverable D110, AOSD-Europe-ULANC-40, March 2008. http://www.aosd-europe.net/deliverables/d110.pdf

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

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
- Further work:
  - Apply feature modeling to deal with change interaction
  - Aspect-oriented change realization at model level
  - Tool support