Automated Inference of Pointcuts in Aspect-Oriented Refactoring

Prasanth Anbalagan and Tao Xie
International Conference on Software Engineering (2007)

SeulKi Lee
2009. 8. 20
Contents

- Introduction
- Background
- Overall approach
- Implementation
- Case study
- Related work
- Conclusion
- Discussion
Current issue

- Nowadays, changeability is more important than reusability
- Aspect-oriented programming helps to easily change the structure of code

Motivation

- With the increase in size of the code and crosscutting concerns, it is hard to process of AO* refactoring
- The more pointcut expressions exist in code, the more difficult to maintain the application

*Aspect-oriented (AO)
Research goal

- Propose an automated approach for processing aspect-oriented refactoring
  - Mining aspects
  - Identifying join points in the source code
  - Clustering the join points
  - Inferring an effective pointcut
- Validate the approach on six existing applications
Aspect-oriented programming

Motivation
- Changeability is current issue, but sometimes developer cannot change the structure of code due to inter-object messages.
- Inter-object messages cause poor reusability and make tangling code.

Principle
- Modularize software system from source code by encapsulating crosscutting concerns.
Aspect-oriented refactoring

Definition

- Combination of two techniques, *software refactoring* and *aspect-oriented programming*, to restructure crosscutting elements in code.
Before being refactored

 AspectJ

- Definition
  - An aspect-oriented programming language
  - A tool for modularization of crosscutting concerns

- Components

```java
public class DrawApplication extends ... {
    public void promptNew() {
        DrawingView.toolDone();
    ...
}    
    public void promptOpen() {
        DrawingView.toolDone();
 ...... 
}

aspect beforeDrawApplication {
    private pointcut invokeTool () : (execution public void DrawApplication.prompt*(..)) || public void DrawApplication.*Selection(..))

    before() : invokeTool() {
        DrawingView.toolDone();
    }
}
```

Contains the set of actions to be taken once the pointcut matches the join points

© KAIST SE LAB 2009
Overall approach

- Consist of four main components
  - Automated process of aspect-oriented refactoring in four components except weaving
Step 1. Mining aspects

- **Purpose**
  - Use an existing aspect mining tool to identify aspects in given source code

```
public class DrawApplication extends ... {
    public void promptNew() {
        DrawingView.toolDone();
        ...
    }
    public void promptOpen() {
        DrawingView.toolDone();
        ...
    }
    public void clearSelection() {
        DrawingView.toolDone();
        DrawingView.fireSelectionChanged();
    }
}
```

**Source code**

**Extracted aspects**

- `DrawingView.toolDone()`
- `DrawingView.fireSelectionChanged()`
- `DrawingView.markTextDirty()`

- **Type-based analysis** (for method arguments)
- **Text-based analysis** (for method name)
Step 2. Identifying Join points

❖ Purpose
- Identify exactly the join points from each aspect
- Analyze the statements before and after each aspect

```java
public void promptNew() {
    DrawingView.toolDone();
    // ...
}

public void clearSelction () {
    // ...
    DrawingView.fireSelectionChanged();
}

public void read() {
    // Set boolean isTextDirty=false;
    DrawingView.markTextDirty();
    // ...
}
```

```
Join points for DrawingView.toolDone() : before
1. public void DrawApplication.promptNew()
2. public void DrawApplication.promptOpen()
3. public void DrawApplication.clearSelection()
4. public void DrawApplication.toggleSelection(Figure)

Join points for DrawingView.fireSelectionChanged() : after
5. public void DrawApplication.clearSelction()
6. public void DrawApplication.toggleSelection(Figure)

Join points for DrawingView.markTextDirty() : after
set boolean DrawApplication.isTextDirty() in
7. public void DrawApplication.read(StorableInputStream)
8. public void DrawApplication.readObject(ObjectInputStream)
```
Step 3. Clustering join points

Purpose

- Perform clustering based on attributes such as different naming parts of a join point

Clustered Data

@relation Method_cluster

@attribute Modifier {public}

@attribute return_type {void}

@attribute Method_name {promptNew, promptOpen}

@attribute Class_name {DrawApplication}

@attribute arguments {(), ()}

@data

0, public, void, DrawApplication, promptNew, (), cluster1

1, public, void, DrawApplication, promptOpen, (), cluster1

Clustered join points

Choose the method_name field of a join point as the prime factor

Join points for DrawingView.toolDone() : before
1. public void DrawApplication.promptNew()
2. public void DrawApplication.promptOpen()
3. public void DrawApplication.clearSelection()
4. public void DrawApplication.toggleSelction(Figure)

Identified join point (output from step 2)
Step 4. Inferring pointcut expressions (1/3)

- **Purpose**
  - Generate pointcut expressions given a set of clustered join points
  - Provide an additional testing mechanism to verify the strength of the inferred pointcuts
Step 4. Inferring pointcut expressions (2/3)

- **Generation of inferred pointcut expressions**
  - **Identify** the naming parts of the clustered join points
  - **Form** a regular expression for each set of naming parts
  - **Output** the pointcut expression with designator

Clustered join points
(output from step 3)

- **Cluster1**
  - DrawApplication.promptNew()
  - DrawApplication.promptOpen()

- **Cluster2**
  - DrawApplication.clearSelection()
  - DrawApplication.toggleSelection(Figure)

Separate naming parts using comma
(field separator, String aligner)

- DrawApplication, **promptNew**
- DrawApplication, **promptOpen**
- DrawApplication, **clearSelection**
- DrawApplication, **toggleSelection**

**Pointcut**
(expression + designator)

Pointcut expression:
- `execution(public void DrawApplication.prompt*())`
- `execution(public void DrawApplication.*Selection(..))`

Pointcut designator:
- `DrawingView.toolDone();`
Step 4. Inferring pointcut expressions (3/3)

- Testing framework
  - Analyze the inferred pointcut expression and the Java bytecode of the application
  - Compare actually matched join points and original set of intended join points

Solution
1. Add specific keywords to their names
2. Append with expressions like “&& (!jp)” to exclude jp from being matched by the new pointcut expression
## Used tools for each module

<table>
<thead>
<tr>
<th>Module</th>
<th>Tool</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aspect mining module</td>
<td>AMT : Aspect Mining Tool</td>
<td>✓ An analysis framework for mining aspects as well as hidden concerns</td>
</tr>
<tr>
<td>Join point identifier</td>
<td>BCEL : Byte code engineering library</td>
<td>✓ A toolkit for the static analysis of Java class files and runtime behavior under refactoring to identify the locations of aspects and their types <em>(before and after)</em></td>
</tr>
<tr>
<td></td>
<td>Java reflection APIs</td>
<td></td>
</tr>
<tr>
<td>Clustering module</td>
<td>WEKA clustering engine</td>
<td>✓ A collection of machine learning algorithms for data mining tasks</td>
</tr>
<tr>
<td></td>
<td></td>
<td>✓ Perform the clustering and output the clustered data set</td>
</tr>
<tr>
<td>Inference module</td>
<td>SimMetrics : similarity measure library</td>
<td>✓ <em>SimMetrics</em> infers pointcut expression and choosing the optimal one</td>
</tr>
<tr>
<td></td>
<td>APTE : Automated pointcut testing</td>
<td>✓ <em>APTE</em> outputs the set of join points matched by the inferred pointcut expression</td>
</tr>
</tbody>
</table>
Case study (1/3)

**Evaluation objectives**
- Impact of the # pointcut expressions to be used in the final refactored code
- Significance level of generated pointcut expressions
- Necessity for the usage of implemented tool

**Evaluation subjects**
- Assess the objectives on six applications
  - AspectJ benchmark suites (Java version)
    - Tetris, DCM, StarJPool
  - Open source application from SourceForge
    - JARP, JHotDraw, Tomcat
## Evaluation results

**Impact of reducing # pointcut expressions**
- The larger an application is, the more substantial reduction suggested approach can achieve (34% ~ 81%)

<table>
<thead>
<tr>
<th>Applications</th>
<th># Classes / LOC</th>
<th># Aspects</th>
<th>Original # pointcut expressions</th>
<th>Reduced # pointcut expressions</th>
<th>Percentage reduction %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tetris</td>
<td>17 / 1474</td>
<td>24</td>
<td>41</td>
<td>27</td>
<td>34.14</td>
</tr>
<tr>
<td>DCM</td>
<td>29 / 3384</td>
<td>62</td>
<td>351</td>
<td>196</td>
<td>44.15</td>
</tr>
<tr>
<td>StarJPool</td>
<td>191 / 16847</td>
<td>321</td>
<td>4860</td>
<td>2554</td>
<td>47.44</td>
</tr>
<tr>
<td>JARP</td>
<td>214 / 26790</td>
<td>513</td>
<td>12134</td>
<td>4375</td>
<td>63.94</td>
</tr>
<tr>
<td>JHotDraw</td>
<td>398 / 28087</td>
<td>456</td>
<td>32661</td>
<td>9091</td>
<td>72.17</td>
</tr>
<tr>
<td>Tomcat</td>
<td>455 / 45400</td>
<td>684</td>
<td>51108</td>
<td>9633</td>
<td>81.15</td>
</tr>
</tbody>
</table>

© KAIST SE LAB 2009
Evaluation results (cont’d)

- Significance level of generated pointcut expressions
  - Clustering join points is based on a common pattern in the split method-name parts of join points
    → Those are similar to expressions that developers made

- Necessity for the usage of implemented tool
  - Reducing # pointcut expressions and effectively generalizing the pointcut definition is necessary
    → Result show that usage of suggested approach would help achieve better formulation of pointcuts
<table>
<thead>
<tr>
<th><strong>Related work</strong></th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>Research goal</strong></th>
<th><strong>Limitation</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Tonella and Ceceator [TSE 05]</td>
<td>✓ Present an assessment of the effects of migrating to AO code</td>
</tr>
<tr>
<td>Binkley et al [LATE 05]</td>
<td>✓ Define a new pointcuts as the logical OR expression when there are multiple join points and pointcut expressions</td>
</tr>
<tr>
<td>Binkley et al [ICSM 05]</td>
<td>✓ Semi-automated approach to support the migration from OO code to AO code</td>
</tr>
<tr>
<td>Zhang et al [LATE 05]</td>
<td>✓ Automated aspect refactoring verification between the refactored aspects and the original sources</td>
</tr>
</tbody>
</table>
Conclusion

- Contribution
  - Aspect oriented refactoring involves automated four modules and additional testing mechanism
    - Mining aspects
    - Identifying join points for the aspects
    - Clustering join points
    - Grouping them under a pointcut definition

- Future work
  - Extend the implementation to automatically synthesize aspects and pointcut expressions to produce aspect code
Discussion

❖ Pros
  ■ Author creates automated aspect refactoring approach except synthesizing aspects and pointcut expressions

❖ Cons
  ■ Suggested approach cannot be conducted on the locations of the join points where exist repeated statement before or after the concern in the same method
Thank you
Aspect-oriented terminology

- **Concern**
  - A independent problem domain
  - Cohesive area of functionality

- **Crosscutting concern**
  - **Definition**
    - A concern that hard to modularize because those are related to multiple classes or methods
  - **Problem**
    - Crosscutting concern causes poor changeability and readability
  - **Crosscutting concern candidate**
    - Transaction, logging, error and exception handling, monitoring, statistic gathering, transaction, session management, etc.