Composition of refactorings for Aspects

Daniel Speicher,
Institute for Computer Science III, ROOTS Group, University of Bonn,
dsp@cs.uni-bonn.de

Problem

Refactoring needs AOP
Refactoring is the process of reorganizing code to improve understandability and maintainability without changing the programs behavior. This is accomplished especially by improving the modularization. If we do this with respect to crosscutting concerns, it means that we are doing aspect-oriented programming.

AOP needs refactoring
Introducing AOP in existing large and complex systems without changing the programs behavior is refactoring. This should be the first step on the long way to systems, which are designed with AOP from the start. As soon as there are bigger systems with AOP there will be the need for refactoring of the aspects.

Consequence: The number of beneficial refactorings increases enormously.

Solution

Tool support
Ensuring behavior preservation for large and complex systems is nearly impossible without tool support. We suggest to represent a refactoring by a conditional transformation, i.e. a condition together with a transformation. The condition checks whether the transformation is applicable and behavior preserving. Only if the condition is fulfilled the transformation is applied.

Composition of conditional transformations
To master the increasing number of conventional and aspect-oriented refactorings we suggest a composition mechanism for conditional transformations. This enables us to build bigger refactorings from smaller building blocks. The core of this composition is the derivation of a joined precondition for a sequence of conditional transformations.

A Simple Example

Composition of *Extract Aspect* from *Extract Method* and *Extract method calls* (example adapted from [L03b])

```java
public class Account {
   private String accountname;
   //...
   public void credit(float amount) {
      logger.write(acountname);
      logger.write(": credit invoked");
      // code for credit
      logger.write(acountname);
      logger.write(": credit successful");
   }
   public void debit(float amount) {
      logger.write(acountname);
      logger.write(": debit invoked");
      // code for debit
      logger.write(acountname);
      logger.write(": debit successful");
   }
}
```

Refactoring 1
Extract method

```java
public class Account {
   private String accountname;
   //...
   private void log(String s) {
      logger.write(acountname);
      logger.write(": " + s);
   }
   public void credit(float amount) {
      log("credit invoked");
      // code for credit
      log("credit successful");
   }
   public void debit(float amount) {
      log("debit invoked");
      // code for debit
      log("debit successful");
   }
}
```

Refactoring2
Extract method calls

```java
public class Account {
   private String accountname;
   //...
   private void log(String s) {
      logger.write(acountname);
      logger.write(": " + s);
   }
   public void credit(float amount) {
      // code for credit
   }
   public void debit(float amount) {
      // code for debit
   }
   private static aspect Logging{
     around (public * Account.*(..)){
       log(thisJoinPointStaticPart
          .getSignature().getName()
          + " invoked");
       proceed();
       log(thisJoinPointStaticPart
          .getSignature().getName()
          + " successful");
     }
}
```

Condition1
The code fragments to extract are similar.

Condition2'
Public methods start and end with the same code fragment.

Backward description for refactoring 1

Condition2
Public methods start and end with the invocation of the same method.

Condition 1 is the precondition of the given conventional refactoring *Extract Method*.
Condition 1 and condition 2' are checked on the initial code.
They build the joined precondition of the composite refactoring.

Condition 2' is generated automatically by the system.

Condition 2 is the precondition of the aspect-oriented refactoring *Extract method calls*. It's easier to formulate then condition 2'.

Benefits of composition

The joined precondition **checks for the whole sequence** of refactorings if it is applicable and behavior preserving.

The application of the composite refactoring is therefore **atomic.** If the precondition is true the re-factoring succeeds. **No rollback** is needed.

Conditional transformations with the no-op transformation serve as post conditions. Composition gives the precondition which **ensures the post condition**.

The set of refactorings can be **adapted** and **extended** by adding new conditional transformations as building blocks. This enables **reuse** of older refactorings.

Composite refactorings can be **edited** and **passed along** like any ordinary document. One could think about a certification process.

Approach

- Define a basic set of conditions and transformations.
- Define the backward descriptions for the transformations. It maps a condition C on a condition C' which is equivalent to C after applying the transformation. I.e. $C' = C \circ T$.
- This backward descriptions enables the composition of conditional transformations.
- Formula: $(C_1, T_1) \oplus \ldots \oplus (C_n, T_n)$ $=(C_1 \wedge B_{T_1} C_2 \wedge \ldots \wedge B_{T_1} \circ \ldots \circ B_{T_{(n-1)}} C_n, T_n \circ \ldots \circ T_1)$

Research

- Automatic generation of backward descriptions from transformations.
- Extend the backward descriptions to other functions on the program space, especially metrics.
- This would enable anticipation of program properties after transformation without applying the transformation.
- Systematically determine a basic set of conditions and transformations as building blocks. This will serve as our language, which should be minimal and as complete as possible.
- Formal model for parameter passing between refactorings.

More Applications

- In addition to building a toolbox of highly configurable refactorings for AOP,
- composition of refactorings should be helpful for facilitating the adaptation to language evolutions (e.g.:Generic Types, JDK 1.5)
- or language extensions (e.g.: LAVA [K00], LogicAJ [W03]).
- The formal model should be even applicable for transformations of other languages (e.g. XML, XMI).

Literature

University of Bonn, Computer Science Deptartment III:

[B03] Uwe Bardey: Abhängigkeitsanalyse für Programm-Transformationen (in German). Diploma thesis. February 2003.

[K00] Günter Kniesel: Dynamic Object-Based Inheritance with Subtyping. PhD Thesis. University of Bonn. July 2000.

[KK04] Günter Kniesel, Helge Koch: Static Composition of Refactorings, To appear in: *Science of Computer Programming (Special issue on "Program Transformation", edited by Ralf Lämmel)*, Elsevier Science, 2004.

[Ko00] Helge Koch: Ein Refactoring-Framework für Java (in German). Diploma thesis. April 2000.

[W03] Tobias Windeln: LogicAJ - eine Erweiterung von AspectJ um logische Meta-Programmierung (in German). Diploma thesis. August 2003.

[BSC03] Paulo Borba, Augusto Sampaio, Márcio Cornélio: A refinement algebra for object-oriented programming. Informatics center, Federal University of Pernambuco, Recife, PE, Brazil, 2003.

[F99] Martin Fowler: Refactoring – Improving the Design of Existing Code. Addison Wesley, 1999.

[HMK] Jan Hannemann, Gail Murphy, Gregor Kiczales: Dialogue-Based Aspect-Oriented Refactoring. Research project. University of British Columbia, Vancouver. http://www.cs.ubc.ca/labs/spl/projects/ao-refactoring.html

[L03a] Ramnivas Laddad: AspectJ in Action. Manning Publications Company. Jul. 2003.

[L03b] Ramnivas Laddad: Aspect Oriented Refactoring Series, 12/2003. http://www.theserverside.com/articles/article.tss?l=AspectOrientedRefactoringPart1, ...l=AspectOrientedRefactoringPart2

Related Work

[LH00] Andreas Ludwig, Dirk Heuzeroth: Meta Programming in the Large. In *2nd International Conference on Generative and Component-based Software Engineering (GCSE)*. Springer, Jan. 2000. LNCS 2177.

[R99] Don Bradley Roberts: Practical Analysis for Refactoring. PhD thesis, University of Illinois, 1999.

[TKB02] Frank Tip, Adam Kiezun, Dirk Bäumer: Refactoring for generalization using type constraint. IBM Research Report RC22662, NY10598. Dec. 4, 2002.

[ZR03] Jianjun Zhao, Martin Rinard: System Dependence Graph Construction for Aspect-Oriented Programs. MIT-LCS-TR-891, Laboratory for Computer Science, MIT, March 2003.