Composition of refactorings for Aspects

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 implementing AOP in existing large and complex systems without changing the programs behavior. This should be the first step on the long way to introducing AOP in existing large and complex systems without changing the programs behavior.

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 composition of this process 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])

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 refactoring succeeds. No rollback is needed.

Benefits of composition

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 other documentation. 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' = C
• This backward descriptions enables the composition of conditional transformations.
• Formulate (C1, T1) ⊕...⊕ (Cn, Tn) = (Cn ∨ B1,... ∨ Br1, Cn ∨ B2,... ∨ Br2, ..., Cn ∨ Brn, Tn, Tn − 1, ..., T1)

Research

• Automatic generation of backward descriptions from transformations.
• Extend the backward descriptions to other functions on the program algebra, 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], LogicJ (W03)).
• The formal model should be even applicable for transformations of other languages (e.g. XML, XMI).

Related Work


University of Bonn. Computer Science Department III.


University of Bonn, Computer Science Science Department III.