Disabling refactoring preconditions for evaluating whether they are overly strong

Melina Mongiovi, Rohit Gheyi, Gustavo Soares
Federal University of Campina Grande
Campina Grande, Brazil
melina@copin.ufcg.edu.br, {rohit,gsoares}@dsc.ufcg.edu.br

Márcio Ribeiro
Federal University of Alagoas
Maceió, Brazil
marcio@ic.ufal.br

Leopoldo Teixeira, Paulo Borba
Federal University of Pernambuco
Recife, Brazil
{lmt,phmb}@cin.ufpe.br

I. Templates

We specify a set of transformation templates that allow disabling preconditions of 10 refactoring implementations of JRRT [1] and Eclipse [2]. The implementations are of the following refactoring types: Add Parameter, Encapsulate Field, Move Method, Pull Up Field, Pull Up Method, Push Down Field, Push Down Method, Rename Class, Rename Field, and Rename Method. A template specifies a kind of transformation in the refactoring engine code that allows disabling the execution of a refactoring precondition. The left and right hand sides of a template illustrate a template of a Java program before and after the transformation, respectively.

The templates of JRRT and Eclipse have the following common meta-variables: *C* specifies a class of the refactoring engine source code (it extends a *D* class); *ds* specifies a set of class and interface declarations of the refactoring engine code; *m* specifies a method name; *T* specifies a type name, *Stmts* specifies a sequence of statements; *msg* specifies a message reported to the user by the refactoring engine when it rejects a transformation; and *cs* specifies a set of class structures, such as methods, attributes, inner classes, and static blocks. Meta-variables equal on both sides of a template means that the transformation does not change them. We create the ConditionsManagement class (*CM*) to manipulate the execution of each refactoring condition (*cond1, cond2, ..., condN*) by runtime. Next, we explain the templates of JRRT (Section I-A) and Eclipse (Section I-B).

A. Template of JRRT

JRRT rejects a refactoring transformation when a precondition is not satisfied. As JRRT does not have a graphical user interface, it only throws a *RefactoringException* (*RefExc*) to terminate the execution and report the error message to the user. To disable a refactoring precondition, we apply a transformation for preventing JRRT of throwing the *RefactoringException* when this precondition is unsatisfied. Transformation 1 illustrates the template of JRRT.

Transformation 1: ⟨Avoid throwing an exception⟩

```
ds
class C extends D {
   cs
   T m(...) {
      Stmts
      throw new RefExc(msg);
      Stmts'
   }
}
```

→

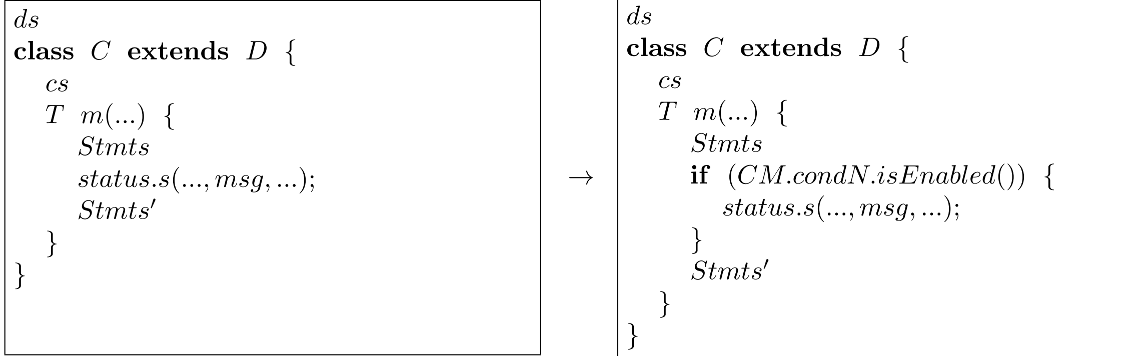
```
ds
class C extends D {
   cs
   T m(...) {
      Stmts
      if (CM.condN.isEnabled()) {
         throw new RefExc(msg);
      }
      Stmts'
   }
}
```

B. Templates of Eclipse JDT 4.5

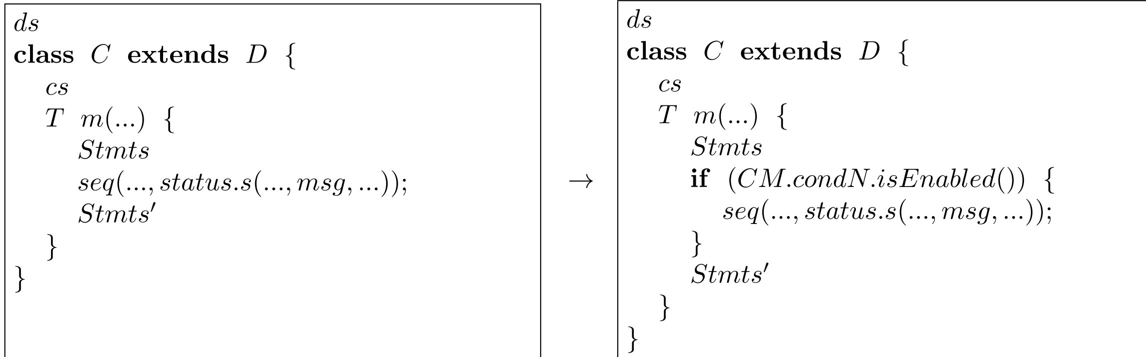
Eclipse implements a class (*RefactoringStatus*) that stores the outcome of the preconditions checking operation. It contains methods, such as *addError*, *addEntry*, *addWarning*, *createStatus*, *createFatalErrorStatus*, *createErrorStatus*, and *createWarningStatus*. Those methods receive a message and other arguments, describing one particular problem detected during the precondition checking. The methods starting with *create* return a *RefactoringStatus* object. The messages are stored in the *refactoring.properties* file. They are represented by a field of the *RefactoringCoreMessages* class. They can be directly accessed by a field call or through a variable, parameter of the method, or the return of a method call. The refactoring implementations of Eclipse check the status of a refactoring transformation, in a *RefactoringStatus* object, after evaluating the preconditions. If it contains some warning or error messages, Eclipse rejects the transformation and reports the messages to the user.

We create the templates of Eclipse by analyzing the smallest code snippet we need to disable for avoiding the engine adds a new error or warning status in a *RefactoringStatus* object. The templates of Eclipse have the following specific meta-variables: *status* specifies an object of *RefactoringStatus* type; *s* specifies a method of *RefactoringStatus*; and *seq* specifies a sequence of nested method calls, which the last method receives a *RefactoringStatus* object as parameter. Transformations 2 and 3 illustrate the templates of Eclipse.

Transformation 2: ⟨Avoid adding a refactoring status⟩



Transformation 3: ⟨Avoid adding a refactoring status within a method parameter⟩



II. IMPLEMENTATION USING ASPECTS

Aspect-Oriented Programming (AOP) allows to reduce the program complexity by breaking it into different concerns [3]. AOP languages have three critical elements: a join point model, a means of identifying join points, and a means of affecting implementation at join points [4]. A join point is an identifiable execution point in a system, such as a call to a method. AOP allows identifying joint points and alters behavior at them. AspectJ [4] is an Aspect-Oriented extension to the Java programming. It offers pointcuts to identify particular join points

by filtering out a subset of all the join points in the program flow and advices to alter behavior at join points selected by pointcuts.

Disabling refactoring preconditions can be seen as a separate concern of the refactoring engine that we want to implement. Therefore, we also implement in AspectJ the disabling precondition concern of our technique by following the specified transformation templates. We write pointcuts to collect all refactoring engine code places that match with the left hand side of the templates. We also write an *around* advice for each pointcut to disable the refactoring preconditions execution in that place. First, we write an abstract aspect (*Aspect*), which declares a general pointcut, advice and auxiliary method to both the Eclipse and JRRT aspects. *Aspect* declares an abstract pointcut *methodMsg* to collect calls to methods with a *String* parameter (*msg*). It also declares an *around* advice to allow executing only the methods collected in *methodMsg*, which the list *Messages.messages* contains *msg* (*execute* method). *Messages.messages* stores the messages related to the conditions we want to disable by runtime and *msg* is the message related to the evaluated condition. Listing 1 illustrates *Aspect*. Next, we explain the aspects of JRRT (Section II-A) and Eclipse (Section II-B).

Listing 1. Abstract aspect to disable preconditions.

```
public abstract aspect DisablingPreconditions {
  abstract pointcut methodMsg(String msg);

  void around(String msg): methodMsg(msg)
    if (executePrecond(msg)) {
      proceed(msg);
    }
  }

  public boolean executePrecond(String msg) {
    return !Messages.reportedMsgs.contains(msg);
  }
}
```

A. Aspect of JRRT

In this section, we explain the aspect we write to disable preconditions of JRRT. The goal of this aspect is to avoid JRRT rejects the transformation by throwing the *RefactoringException*. To make it possible by using aspects, we create the hook method *AST.RefactoringException.throwException* that throws a *RefactoringException*. Then, we replace in the refactoring implementation code the occurrences of "*throws new RefactoringException(msg)*" with a call to this method. Without this change, we cannot implement in aspects due to a pointcut limitation. We implement the *methodMsg* pointcut to collect all calls to this method. Therefore, the refactoring implementations of JRRT do not throws the *RefactoringException* (reject the transformation), when set the *Messages.messages* list with the messages related to the preconditions wanted to disable. Listing 2 illustrates the aspect used to disable preconditions of JRRT.

Listing 2. Aspect to disable refactoring preconditions of JRRT.

```
public aspect DisablingPreconditionsJRRT extends DisablingPreconditions {
  pointcut methodMsg(String msg):
    call (void AST.RefactoringException.throwException(String,..)) && args(msg,..);
}
```

B. Aspect of Eclipse

In this section, we explain the aspect we write to disable preconditions of Eclipse. The goal of this aspect is to avoid Eclipse of adding a new warning or error status in a *RefactoringStatus* object. The *RefactoringStatus* class declares some void methods that add a new status in a *RefactoringStatus* object (methods starting with *add*). It also declares methods that create a new *RefactoringStatus* object, add the status, and return this object (methods starting with *create*). We implement a pointcut and an advice for both kinds of methods. The *methodMsg* pointcut collects calls to the *addError*, *addWarning*, and *addEntry* void methods of *RefactoringStatus* and the *method_msg_nonvoid* pointcut collects calls to the *createStatus*, *createErrorsStatus*, *createWarningStatus*, and *createFatalErrorStatus* methods. Therefore, the refactoring implementations of Eclipse do not add or create a new status when set the *Messages.messages* list with the messages related to the preconditions wanted to disable. Listing 3 illustrates the aspect used to disable preconditions of Eclipse.

Listing 3. Aspect to disable refactoring preconditions of Eclipse.

```
import org.eclipse.ltk.core.refactoring.RefactoringStatus;

public aspect DisablingPreconditionsEclipse extends DisablingPreconditions {
  pointcut methodMsg(String msg):
    call (void RefactoringStatus.addError(String,..)) && args(msg,..) ||
    call (void RefactoringStatus.addWarning(String,..)) && args(msg,..) ||
    call (void RefactoringStatus.addEntry(int,String,..)) && args(int,msg,..);

  pointcut methodMsgNonVoid(String msg):
    call (RefactoringStatus RefactoringStatus.createErrorStatus(String,..)) && args(msg,..) ||
    call (RefactoringStatus RefactoringStatus.createWarningStatus(String,..)) && args(msg,..) ||
    call (RefactoringStatus RefactoringStatus.createFatalErrorStatus(String,..)) && args(msg,..) ||
    call (RefactoringStatus RefactoringStatus.createStatus(int,String,..)) && args(int,msg,..);

  RefactoringStatus around(String msg): methodMsgNonVoid(msg) {
    if (executePrecond(msg)) {
      return proceed(msg);
    } else {
      return new RefactoringStatus();
    }
  }
}
```

References

[1] M. Schäfer and O. de Moor, "Specifying and implementing refactorings," in *Proceedings of the 25th ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '10. ACM, 2010, pp. 286–301.

[2] Eclipse.org, "JDT Core Component," 2016, at http://www.eclipse.org/jdt/core.

[3] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, *Aspect-oriented programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 220–242.

[4] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold, "Getting started with AspectJ," *Communications of the ACM*, vol. 44, no. 10, pp. 59–65, 2001.