JDeodorant: Identification and Removal of Type-Checking Bad Smells

Nikolaos Tsantalis*, Theodoros Chaikalis, Alexander Chatzigeorgiou
Department of Applied Informatics, University of Macedonia
54006 Thessaloniki, Greece
{nikos, chaikalis}@java.uom.gr , achat@uom.gr

Abstract

In this demonstration, we present an Eclipse plug-in that automatically identifies Type-Checking bad smells in Java source code, and resolves them by applying the "Replace Conditional with Polymorphism" or "Replace Type Code with State/Strategy" refactorings. To the best of our knowledge there is a lack of tools that identify Type-Checking bad smells. Moreover, none of the state-of-the-art IDEs support the refactorings that resolve such kind of bad smells.

Index Terms: Software maintenance, Object oriented programming, Software quality

1. Introduction

The use of an Object-Oriented programming language does not always guarantee the employment of object-oriented design principles. According to Demeyer et al. [2], the most common problem is that programmers who have not fully understood the object-oriented paradigm use conditional statements to simulate dynamic dispatch and late binding, instead of taking advantage of polymorphism. Problems like this may also arise when a system has been repeatedly modified to satisfy constant requirement changes.

Generally, type-checking code is introduced in order to select a variation of an algorithm that should be executed, depending on the value of an attribute. Mainly it manifests itself as complicated conditional statements that make the code difficult to understand and maintain. In order to solve this problem several methodologies have been proposed [2, 4], that eliminate type-checking conditional statements by applying refactorings that introduce inheritance and polymorphism. While several mechanics have been suggested as solutions to this problem, the identification of the problem itself is a subject that has not been completely covered and would benefit from further research.

2. Methodology

Our methodology consists of two parts. The first deals with the identification of type-checking bad smells. The second concerns their elimination by applying appropriate refactorings.

2.1 Identification of Type-Checking bad smells

Concerning the type-checking bad smells, two cases can be distinguished. In the first case, there is an attribute in a class that represents state (*type field*). Depending on its value, the corresponding branch of a conditional statement is executed. If the conditional code fragment is a *switch* statement, the type field (or an invocation of its getter method) should appear in the switch expression, while the static attributes representing the different values that the type field may obtain should appear in all case expressions (Table 1a). If the conditional code fragment is an *if/else if* structure, the static attributes should be compared for equality with the type field (or an invocation of its getter method) in all conditional expressions (Table 1b). It should be noted that a switch/if statement should contain more than one case to be considered as a valid type-checking candidate since a single case is usually not regarded as a sign of possible future changes.

Table 1: Type-checking examples where an attribute represents state

a: *switch* statement

```
class Context {
  private int type;
  public void m() {
    switch(type) {
      case VALUE_0:
      //code for case 0
      case VALUE_1:
      //code for case 1
      case …
    }
  }
}
```

b: *if/else if* statement

```
class Context {
  private int type;
  public void m() {
    if(type == VALUE_0) {
      //code for case 0
    }
    else if(type == VALUE_1) {
      //code for case 1
    }
    else if(type ==…) {…}
  }
}
```

978-1-4244-2157-2/08/$25.00 © 2008 IEEE

Table 2: Type-checking examples performing RTTI

a: *instanceof*

```
class Client {
 public void m(SuperType type) {
  if(type instanceof Subclass0) {
   Subclass0 s = (Subclass0)type;
   //code for case 0
  } else
  if(type instanceof Subclass1) {
   Subclass1 s = (Subclass1)type;
   //code for case 1
  }
 }
}
```

b: *getClass*

```
class Client {
 public void m(SuperType type) {
  if(type.getClass() == Subclass0.class) {
   Subclass0 s = (Subclass0)type;
   //code for case 0
  } else
  if(type.getClass() == Subclass1.class) {
   Subclass1 s = (Subclass1)type;
   //code for case 1
  }
 }
}
```

c: The subclass type is polymorphically obtained

```
class Client {
 public void m(SuperType type) {
  if(type.getType() == STATIC_VALUE_0) {
   Subclass0 sub = (Subclass0)type;
   //code for case 0
  }
  else if(type.getType() == STATIC_VALUE_1) {
   Subclass1 sub = (Subclass1)type;
   //code for case 1
  }
 }
}
```

In the second case, there is a conditional statement that employs RunTime Type Identification (RTTI) in order to cast a reference from a base (superclass) type to the actual derived (subclass) type and invoke methods of the specific subclass. In this case the inheritance hierarchy corresponding to these class types already exists, but it is not exploited by using polymorphism. RunTime Type Identification usually appears as an *if*/*else if* statement where each conditional expression examines whether a base type reference actually points to a subclass type, using the *instanceof* keyword (Table 2a), or the *getClass* method (Table 2b), or by invoking an abstract method of the superclass (implemented by all its subclasses) that polymorphically returns the value of the static attribute corresponding to each subclass (Table 2c).

2.2 Application of "Replace Type Code with State/Strategy" refactoring

In the case where an attribute represents state (*type field*) whose value determines the corresponding branch of a conditional statement to be executed, there is an opportunity for applying the "Replace Type Code with State/Strategy" refactoring. Particularly, the class containing the type field will play the *Context* role in the State/Strategy pattern [3]. The conditional branches of the type-checking code will be moved as separate methods to the subclasses of a newly created State/Strategy inheritance hierarchy.

Concerning the construction of the State/Strategy inheritance hierarchy, an abstract class should be created that will play the role of the State/Strategy. The name of the abstract class will be the name of the type field. An abstract method having the same name and return type with the method containing the type-checking code fragment should be added to the abstract class. The number of the concrete State/Strategy subclasses that should be created is equal to the number of the conditional branches inside the type-checking code. The names of the concrete subclasses will be the names of the static attributes corresponding to each conditional expression. The concrete subclasses should implement the abstract method of the State/Strategy superclass by copying the code of the corresponding conditional branch inside the body of the overridden method.

If at least one of the copied code fragments accesses fields or methods of the class that it originally belonged to (Context class), then a parameter of Context type should be added to the signature of the abstract method (and therefore to the signature of all concrete methods implementing it), in order to enable the access of these fields/methods. Furthermore, the accessed/assigned private fields should be replaced with invocations of their getter/setter methods and the visibility modifier of the accessed private methods should be changed to public. Finally, if parameters or local variables of the method containing the type-checking code fragment are being accessed by the copied code fragments, they should be added as parameters to the signature of the abstract method.

Regarding the class that will play the Context role the main concern is the preservation of its public interface along with its original functionality, since client classes may be associated with it in the original system. The type field should become a reference to the State/Strategy abstract class. The type-checking code fragment should be replaced with an invocation of the State/Strategy abstract method through the type field reference. The setter method of the type field should be updated in order to set the value of the type field with the appropriate instance of the State/Strategy subclass that corresponds to the passed argument. All the assignments of the type field in the Context class should be replaced with an invocation of the updated setter method. Concerning the getter method of the type field, an abstract method having the same signature with it should be added in the State/Strategy superclass. The subclasses implementing it should return the corresponding static attribute that represents their state value. The original getter method should

delegate to the abstract method of the State/Strategy class through the type field reference. All the accesses of the type field in the Context class should be replaced with an invocation of the updated getter method.

2.3 Application of "Replace Conditional with Polymorphism" refactoring

In the case where the type-checking code employs RTTI, there is an opportunity for applying the "Replace Conditional with Polymorphism" refactoring. In this case the inheritance hierarchy on which polymorphism can be applied already exists. An abstract method having the same signature with the method containing the type-checking code fragment should be added to the top level class of the inheritance hierarchy. Each conditional branch should be moved as a separate method to the subclass that it is related to. The appropriate subclass is identified by the statement that casts the superclass type reference to the actual subclass type (casting statement) inside the corresponding branch. After the move of each conditional branch the casting statement is no longer necessary, since subclass methods can now be invoked directly, and thus should be removed. The type-checking code fragment should be replaced with an invocation of the abstract method of the top level class through the superclass type reference.

3. Tool Overview

The tool uses the ASTParser API of Eclipse Java Development Tools to identify switch/if statements that exhibit type-checking bad smells, and the ASTRewrite API to apply the appropriate refactorings on source code. The tool can be obtained from [1].

The user selects the "Type Checking" action from the "Bad Smells" menu item to open the corresponding view. In order to identify the bad smells, the user selects a project from Package Explorer and clicks the "Identify Bad Smells" button. The "Type Checking" view presents in table format the methods that perform type-checking and the refactorings that should be applied. By double-clicking a row in the table, the code that contains the corresponding conditional statement is automatically highlighted. A sample screenshot highlighting an identified type-checking code fragment is shown in Figure 1.

In order to apply the refactoring, the user should select the row of interest and then click the "Apply Refactoring" button. All the modified or newly created classes will automatically open in the editor. The user has the option to undo the refactoring by clicking the "Undo Refactoring" button.

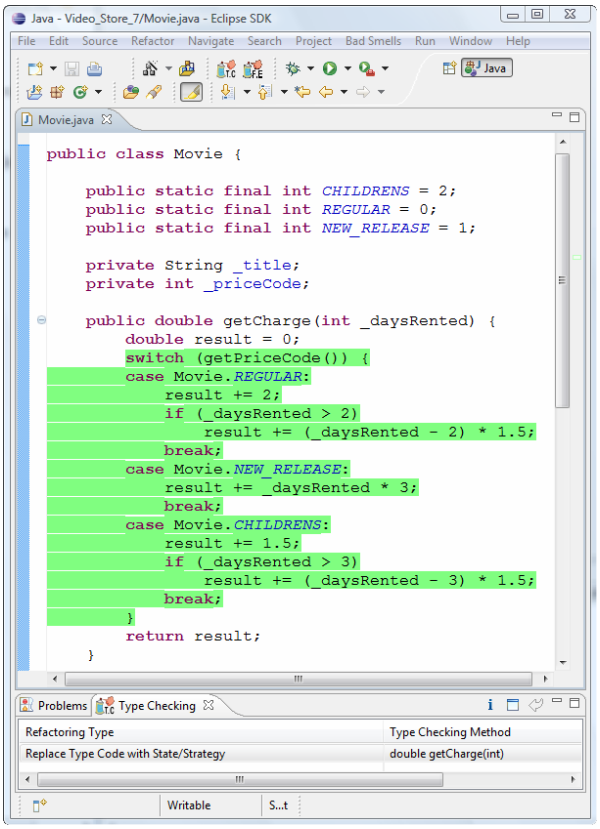

Figure 1: JDeodorant screenshot

4. Evaluation

The proposed tool has been evaluated on three teaching examples found in the textbooks by Demeyer et al. [2] and Fowler et al. [4]. The tool correctly identified the type-checking bad smells suggested by the authors of each example and also applied the refactorings in the same way they have been applied in the corresponding textbooks. A further challenge is to perform a systematic evaluation on large-scale systems in order to assess the precision and recall of the tool.

5. References

[1] Bad Smell Identification for Software Refactoring, http://www.jdeodorant.org, 2007

[2] S. Demeyer, S. Ducasse, O. Nierstrasz. *Object Oriented Reengineering Patterns*, Morgan Kaufman, 2002.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, Boston, MA, 1995.

[4] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts, *Refactoring: Improving the Design of Existing Code,* Addison Wesley, Boston, MA, 1999.