Refactoring Opportunities for Replacing Type Code with State and Subclass

Jyothi Vedurada
IIT Madras, Chennai, India, vjyothi@cse.iitm.ac.in

V Krishna Nandivada
IIT Madras, Chennai, India, nvk@iitm.ac.in

***Abstract*—Refactoring restructures a program to improve its readability and maintainability, without changing its original behavior. One of the key steps in refactoring is the identification of potential refactoring opportunities. In this paper, we discuss about the relevance of two popular refactorings "Replace Type Code with Subclass" and "Replace Type Code with State" in real world Java applications and describe some of the challenges in automatically identifying these refactoring opportunities.**

I. Introduction

Polymorphism allows us to define supertypes that contain common behaviors and subtypes which specialize those behaviors. Even though the object-oriented languages provide mechanisms to support polymorphism, programmers often use state-checking to simulate polymorphism, with the help of conditional (`switch` and `if`) statements. Such a practice leads to three main disadvantages : 1) extendability: adding new behaviors to the subtypes require changes to the conditional statements spread across many classes; 2) maintainability: modifying the existing behaviors or fixing bugs requires that the changes do not effect the code surrounding the state-checking code; 3) readability: since the state-checking code may be spread across multiple classes, it becomes hard to reason about the behavior associated with each state.

The code with complex conditional-state-checking statements can be improved by using "replace conditional with polymorphism" (RCP) refactoring [1], [2]. Kannangara and Wijayanayake [3] also empirically show that RCP refactoring is the most effective refactoring (among the ones listed the Fowlers catalog [2]) to improve the code. RCP refactoring uses two important refactorings underneath: i) replace type code with subclasses (SC), and ii) replace type code with state (ST). We now show an example to demonstrate typical RCP refactoring opportunities found in real world programs.

Fig. 1 shows a code snippet of the class `GraphAxis` from the jOcular [4] application (an optical design software). Here, the field `log` controls if the object is a logarithmic axis or not. Thus, intuitively there are two types of GraphAxis objects: `LogGraphAxis` and `NonLogGraphAxis`, and the conditional statements at Lines 5, 12, 18 in `scale`, `unscale` and `getGridLines` methods of the class are used to control the execution of the state specific codes. This state-checking code can be seen as an SC/ST refactoring opportunity and it can be improved by applying SC/ST refactoring followed by RCP refactoring [2]; Fig. 3 shows the resulting subclasses. After the subclasses are created, the conditional-checking code

```
1  public class GraphAxis {
2   protected boolean log; ...
3   public double scale(double n){
4    double res = n;
5    if(log){ /* state-checking on log */
6     res=Math.log10(res); res-=Math.log10(min);
7     res /= Math.log10(max) - Math.log10(min);
8     } else { res -= min; res /= max - min;}
9     ... return res; }
10  public double unScale(double n){
11    double res = n; res -= .1; res /= .8;
12    if(log){ /* state-checking on log */
13      res *= Math.log10(max) - Math.log10(min);
14      res += Math.log10(min); res=Math.pow(10,res);
15    } else { res *= max - min; res += min; }
16    return res;  }
17  public ArrayList<Double> getGridLines(){
18    if (log) ...//20 lines of state-dependent code
19  } }//class GraphAxis
```

Fig. 1. Code snippet from jOcular-0.039.

can be replaced with a polymorphic function call. For example, Lines 5-8 can be replaced by "`res=scaleLog(n);`".

It is quite challenging to manually identify such refactoring opportunities in large projects. Recent works[5], [6] aim to automatically identify such opportunities. However, these approaches fail to identify many ST refactoring opportunities due to restrictions in their approaches (for example, when the state checking is not done against named constants or the state is computed via expressions involving locals and heap locations). Further, they do not differentiate between SC and ST refactoring opportunities. In this paper, we first present some important challenges involved in: (i) identifying these SC/ST refactoring opportunities effectively and (ii) classifying the identified opportunities (into SC or ST) precisely. We also present our experience of studying a number of Java applications to identify the RCP refactoring opportunities.

II. Challenges

We now present the challenges involved in identifying and classifying the SC/ST refactoring opportunities.

Identification. The first complexity in identifying refactoring opportunities is to identify the possible list of type codes [2]. Not all the fields present in (or that influence the outcome of) the conditionals can be qualified as type codes. The fields like the `log` field of `GraphAxis` (see Fig. 1), that help differentiate different forms/states of the object are called as type codes. Identifying such fields is a key step in the identification of SC/ST refactoring opportunities. However, these fields might not be explicitly present at the conditionals (the value may

```
tmp=log; ...        | if(getLog()){    | if(expr || log){
if(tmp){...}        |   ...            |  ... print (log)
else {...}          | }else {...}      | } else {...}
       (a)          |       (b)        |       (c)
```

Fig. 2. Field of a class at conditional statement

```
class LogGraphAxis extends GraphAxis{...
 public boolean getLog(){return true;}
 public double scaleLog (double res){
  res=Math.log10(res); res-=Math.log10(min);
  res/= Math.log10(max) - Math.log10(min);
  return res;} ... }
```

```
class NonLogGraphAxis extends GraphAxis{...
  public boolean getLog(){return false;}
  public double scaleLog (double res){
    res-= min; res/= max - min; return res; } ... }
```

Fig. 3. Subclasses created for the class `GraphAxis`

flow through some local variables or heap locations) or can be present as a part of complex expressions which makes the refactoring process difficult/not interesting/not possible.

The code snippets in Fig. 2 demonstrate a few such variations. In Fig. 2(a), the field `log` is not directly present in the boolean expression, but its value is reaching this conditional via one (or more) assignment statement(s). In Fig. 2(b) the value of `log` flows via a function call `getLog()`. Finding such paths (precisely) in large code bases is non-trivial.

In Fig. 2(c), although `log` is present in the expression explicitly, it is present along with another expression `expr` (joined using a || or `&&` operator). Naively extracting subclasses in such a code might not always preserve the original behaviour (note: `expr` may have side effects). Even in the absence of side effects, interesting challenges remain. It may be semantically incorrect to extract the body of the conditional as the behavior of a unique class (such as, in Fig. 2(c), extracting the body of the conditional as a method in either `LogGraphAxis` or `NonLogGraphAxis`). Identifying such a field as type code may not lead to a successful refactoring.

Classification and Resulting Class Hierarchy. Choosing the best suited refactoring between SC and ST requires checking whether the typecode is mutable [2]. If the state of an object (value of its type code) changes during its life time, then it is an ST refactoring opportunity, or else it can be seen as either SC or ST opportunity. Performing such an analysis in a scalable and precise manner in large code bases is challenging.

Based on the chosen refactoring, the inheritance structure in the refactored code varies. For example, for the code shown in Fig. 1, both SC and ST refactorings may be performed. In case of SC refactoring, it may lead to classes as shown in Fig. 3; the created subclasses extend `GraphAxis` directly. In case of ST refactoring, an intermediate state class is created and the subclasses extend the state class. In addition to the arguments shown in Fig. 3, the methods need an additional argument (of type `GraphAxis`) to access the fields of `GraphAxis`.

III. Relevance of SC/ST refactoring

We present the relevance of SC/ST refactorings on eight Java projects. Of these, jfreechart-1.0.14, jOcular-0.039, javaGeom-0.10.2, RackJ-1.05, and Unicode-Rewriter(UR)-1.0 are chosen from sourceforge [4]; these projects are alpha/pre-alpha releases. The remaining projects avrora-1.7.106, fop-0.95, sunflow-0.07.2 are taken from Dacapo [7] benchmark suite; these projects are stable and matured releases. For each of these projects, two illustrative opportunities are shown in Fig. 4, as classname, type code pairs, along with the number of conditional statements associated with each opportunity (#u). We analyze first of the two reported opportunities below.

Bench	LOC	Refactoring Opportunity	#
jOcular	31K	⟨GraphAxis, log⟩	3
		⟨BooleanProperty, m_value⟩	8
javaGeom	27K	⟨Ellipse2D, direct⟩	9
		⟨Circle2D, direct⟩	8
jfreechart	204K	⟨XYSeries, autoSort⟩	5
		⟨ChartPanel, useBuffer⟩	5
rackj	23K	⟨AlignmentRecord, forwardStrand⟩	7
		⟨ReadMapComparator, forward⟩	3
UR	11K	⟨ID3v2Frame, compression⟩	8
		⟨ID3v2ExtendedHeader, crc_present⟩	4
avrora	100K	⟨Set, delegating⟩	11
		⟨Mon, show⟩	5
fop	162K	⟨CommandLineOptions, inputmode⟩	5
		⟨BlockViewport, clip⟩	6
sunflow	25K	⟨Geometry, builtTess⟩	3
		⟨UberShader, glossyness⟩	2

Fig. 4. Sample refactoring opportunities from benchmarks.

jOcular: `GraphAxis:log` Discussed in Section I.
javaGeom: a geometrical computations library. The field `direct` in `Ellipse2D` checks if it is directed; accordingly the functionality of the object (such as, finding tangents) differs. Suggested subclasses: `DirEllipse`, `UndirEllipse`.
jfreechart: a Java chart library. Field `XYSeries:autoSort` controls whether the elements of a collection class data are sorted; accordingly, the behavior of the object varies. Suggested subclasses: `autoSorting`, `nonAutoSorting`.
rackj: analyzes RNA-sequence data. The order of processing an RNA strand (forward/backward) is decided by the field `forwardStrand` in `AlignmentRecord`. Suggested subclasses: `FwdAlignmentRec` and `BwdAlignmentRec`.
UR: converts ID3 tags to Unicode. The field `compression` in `ID3v2Frame` tells if the data is compressed. Suggested subclasses: `CompID3v2Frame` and `UncompID3v2Frame`.
avrora: a simulator for running programs on a grid of micro-controllers. The `Set:delegating` field controls the operations like add, contains, and so on. Suggested states for each `Set` object: `delegating` or `nonDelegating`.
fop: a print formatter. The type of the input (xml, image, and so on) is indicated by `CommandLineOptions:inputmode` field. Suggestion: six subclasses (one per input mode).
sunflow: image rendering software using ray-tracing. The `Geometry:builtTess` field checks the object has tesselation done. Else it builds one. Suggested subclasses: `GeometryTess` and `GeometryNoTess`.

IV. Conclusion

In this paper, we presented the relevance of the SC and ST refactorings using eight open source projects, which shows that state-checking is indeed used to simulate polymorphism. We also demonstrated the challenges involved in automatically identifying these refactoring opportunities.

REFERENCES

[1] W. F. Opdyke, "Refactoring Object-oriented Frameworks," Ph.D. dissertation, Champaign, IL, USA, 1992, uMI Order No. GAX93-05645.

[2] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[3] S. H. Kannangara and J. Wijayanayake, "An Empirical Exploration of Refactoring effect on Software Quality using External Quality Factors," *ICTer*, vol. 7, no. 2, 2014.

[4] "Soureforge: Web-based service that offers software developers a centralized online location to control and manage free and open-source software projects," https://sourceforge.net/.

[5] N. Tsantalis and A. Chatzigeorgiou, "Identification of refactoring opportunities introducing polymorphism," *Journal of Systems and Software*, vol. 83, no. 3, pp. 391–404, 2010.

[6] A. Christopoulou, E. A. Giakoumakis, V. E. Zafeiris, and V. Soukara, "Automated refactoring to the strategy design pattern," *Inf. Softw. Technol.*, vol. 54, no. 11, pp. 1202–1214, Nov. 2012. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2012.05.004

[7] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo Benchmarks: Java Benchmarking Development and Analysis," in *OOPSLA '06*. New York, NY, USA: ACM Press, Oct. 2006, pp. 169–190.