基于抽象语法树和多态机制的复杂条件语句自动重构研究

刘 伟[1]，胡志刚[1,2]，刘宏韬[2]

(1. 中南大学信息科学与工程学院 长沙 410083; 2. 中南大学软件学院 长沙 410075)

【摘要】针对源代码中复杂条件语句将增加程序的复杂性，影响代码的易理解性、可测试性、可维护性和可扩展性等问题，提出了一种基于抽象语法树和多态机制的复杂条件语句自动重构方法，用于识别源代码中复杂条件语句的重构时机并实施自动代码重构。首先将源代码转换为抽象语法树，再探测代码中的条件语句，寻找满足预定条件的条件语句，最后利用多态机制对条件语句进行自动重构，将其封装到一系列子类中。对4个开源项目进行重构时机识别和自动重构实验。实验结果表明，重构时机识别算法的精确率可达100%，并能准确地实现代码的自动重构；经回归测试，重构后的代码未引入任何错误；此外，该算法具有较好的性能，执行时间与系统规模呈线性关系，能够应用于各类不同规模的系统。

关 键 词 抽象语法树; 复杂条件语句; 重构时机识别; 多态; 重构
中图分类号 TP311.5 文献标志码 A doi:10.3969/j.issn.1001-0548.2014.05.018

Automatic Refactoring for Complex Conditional Statements Based on Abstract Syntax Tree and Polymorphism

LIU Wei[1], HU Zhi-gang[1,2], and LIU Hong-tao[2]

(1. School of Information Science and Engineering, Central South University Changsha 410083;
2. School of Software, Central South University Changsha 410075)

Abstract In order to solve the problems that complex conditional statements increase the complexity of program, and affect the understandability, testability, maintainability, and extendibility of existing code, a novel approach for automatic refactoring of complex conditional statements based on abstract syntax tree and polymorphism is proposed. The approach can be used to detect the refactoring opportunities for complex conditional statements and implement automatic refactoring. In this approach, the source code file is transformed to an abstract syntax tree at first; then all of the conditional statements are detected and the candidate statements which meet the preconditions are identified; and finally, the conditional statements are refactored automatically by introducing polymorphism, and each branch of the them is encapsulated into a subclass. Four projects are tested to identify refactoring opportunities and implement automatic refactoring. The results show that the precision of the identification algorithm for refactoring opportunities is 100%. Meanwhile, the approach can execute automatic refactoring correctly. The regression testing shows that none error is imported after refactoring. In addition, this approach has good efficiency and the execution time has a linear relationship with the size of system. It means that this approach can be used for projects of different scales.

Key words abstract syntax tree; complex conditional statements; identification of refactoring opportunities; polymorphism; refactoring

重构(refactoring)是在不改变既有代码功能的基础上对程序源代码进行调整，提高其可理解性，降低代码修改和维护成本[1]。如何自动识别重构时机以及实现源代码的自动重构已成为软件工程领域的一个重要研究方向。

文献[2-4]对移除方法(move method)、提取方法(extract method)、提取类(extract class)等常用的重构手段的重构时机识别都进行了深入研究，并提出了一系列算法来自动识别代码重构时机。文献[5-6]针对软件中的克隆代码(重复代码)，基于软件度量、程序依赖图(program dependence graph, PDG)等技术，提出了多种重构时机识别算法和自动重构方法。文献[7-8]基于博弈理论和类的内聚性提出了多种提取类(extract class)的重构时机识别算法。文献[9]研究了如何识别泛化重构时机并研发了相应的工具。通过对重构时机的识别和自动化重构来降低手工重构

收稿日期：2013– 09 – 07；修回日期： 2014 – 01 – 15
基金项目：国家自然科学基金(60970038, 61272148)
作者简介：刘伟(1982 –)，男，博士生，高级工程师，主要从事软件工程、数据挖掘方面的研究.

引入的错误，提高代码重构的效率，进而改善代码质量。此外，文献[10-11]对当前重构工具的使用情况进行了研究和分析，发现现有的重构工具没有得到充分使用的主要原因之一是开发人员并不知道何时该实施重构，即未能有效识别出重构时机。因此，自动识别出重构时机并引导开发人员实施自动重构对于软件开发和维护都具有重要意义。

文献[1]认为在程序中，复杂条件逻辑是导致程序复杂度上升的最常见因素之一。大量复杂条件语句的存在将影响到源代码的易理解性、可维护性、可重用性、可测试性和可扩展性，因此，对复杂条件语句进行合理的重构将有助于提高代码质量。本文提出的复杂条件语句自动重构方法将从程序源代码固有的特点出发，针对代码中存在的具体问题提供一种重构时机识别方法和半自动的重构方法，实现了文献[1]中提出的以多态取代条件式(replace conditional with polymorphism)和以子类取代类型码(replace type code with subclasses)两种重构手段的自动化。

1 多态机制与抽象语法树

1.1 多态机制

多态是面向对象的三大特性之一，在复杂条件语句的重构过程中，将实施以多态取代条件式(replace conditional with polymorphism)这一重构手段，将原有的包含复杂条件语句的方法进行分解，创建一系列子类，这些子类都包含与原有方法具有相同签名的方法，每一个子类的方法对应原有方法中复杂条件语句的一个分支。与重构之前的复杂条件语句相比，基于多态机制的重构具有以下优点。

1) 将分支语句分散到子类中，从而分解原有的过长方法，也避免产生过大类。

2) 将复杂条件语句分解到子类中可提高代码的可测试性，降低在测试时遗漏某一程序分支的可能性。

3) 将条件逻辑移至子类后，增加新的条件逻辑只需对应增加一个新的子类，原有代码无须任何修改，符合开闭原则，系统具备更好的可扩展性。

总之，引入多态机制的复杂条件语句重构可降低程序的复杂度，让代码更易于维护和扩展，提升程序源代码的质量。

1.2 抽象语法树

为了提高源代码分析与重构的效率，通常需要将程序源代码转换成某种中间形式，抽象语法树(abstract syntax tree, AST)是一种常用的程序代码的中间表示形式，它使用树状结构来表示源代码的抽象语法结构，树上的每一个节点都对应源代码中的一个构成元素。不同类型的节点对应不同的程序结构单元，在分析时可以根据需要来选择层次的节点，而不必每次都深入到代码的细节，从而可以缩短源代码解析时间，提高代码分析效率。

抽象语法树可以通过对源代码进行词法分析和语法分析得到。本文将以Java程序为例，详细介绍如何基于抽象语法树识别重构时机并实现自动重构。在对Java源代码进行分析时，每一个源文件(.java)文件都对应一棵抽象语法树。为了方便对Java抽象语法树进行操作，Eclipse中的Eclipse JDT提供了一组访问和操作Java源代码的API(application programming interface，应用程序接口)，其中，Eclipse AST[12]用于创建和操作抽象语法树。

本文基于Eclipse AST将需要解析的源代码转换成抽象语法树，然后对抽象语法树进行解析，判断是否存在重构时机，如果探测到符合条件的重构时机，则调用相应的程序来实现代码的自动重构。由于抽象语法树比源代码的解析过程更加简洁高效，因此，基于抽象语法树的代码重构时机识别和自动重构算法具有较高的执行效率。

2 重构时机识别研究

通过解析抽象语法树，可提取其中的条件语句，再根据预先设置的条件来识别重构时机。本文的重构时机识别主要基于对类型码(type code)的判断，类型码的数据类型包括int、short、char、byte和String，类型码可以是类的成员变量或者方法的参数。在if语句或者switch语句中通过对类型码值进行判断来决定执行哪一个分支，每一个类型码通常对应一组常量值，因此，在识别重构时机时需要检测类型码的存在。重构时机识别算法的伪代码如下所示。

Input: The name of the root directory which contains source code files for identifying refactoring opportunities.
Output: A list saved all class names, method names and type code variable names.

```
1  declare a null list named resultList
2  for each source code file in the directory
3     create an AST for the file
4     declare a String variable named className
5     assign the class name to className
6     declare a null list named fieldList
7     for each FieldDeclaration in the AST
```

```
8       if the data type of the field is in [int, short, char, byte, String] and the field is not a final field
9         get the field name and add it to fieldList
10      end if
11    end for
12    for each MethodDeclaration in the AST
13      if the method is a main() or construct or abstract method
14        break
15      end if
16      declare a String variable named methodName
17      assign the method name to methodName
18      declare a null list named parameterList
19      if the method has parameter(s)
20        if the data type of a parameter is in [int, short, char, byte, String]
21          save the parameter into parameterList
22        end if
23      end if
24      if an IfStatement node or a SwitchStatement node exists in the method
25        if the node is an IfStatement without ELSE_STATEMENT
26          break
27        end if
28        get all variables in condition expression
29        if one of the variables exists in fieldList or parameterList, and if the node is IfStatement and the variable is in an infix expression which has a "==" operator or it is a parameter of the equals() method
30          declare a RefactoringInfoObject named refInfoObj
31          assign className to refInfoObj.className
32          assign methodName to refInfoObj.methodName
33          assign variable name to refInfoObj.variableName
34          add refInfoObj to resultList
35          continue
36        end if
37      else
38        continue
39      end if
40    end for
41  end for
42  return resultList
```

算法伪代码中，第1语句用于声明一个存储所有待重构类类名及相应方法的集合对象resultList；从第2语句开始对源代码文件进行逐个分析，为每一个源代码文件创建一棵对应的抽象语法树AST：在第4语句代码中，String类型的变量className用于存储类名；第6语句～第11语句用于将所有int、short、char、byte和String类型的成员变量(非常量)存储到一个名为fieldList的集合中；第12语句开始对AST中的每一个成员方法进行遍历：第13语句～第15语句用于跳过main()函数、构造函数和抽象函数；第16语句和第17语句将方法名存储到String类型的变量methodName中；第18语句～第23语句声明一个名为parameterList的集合用于存储当前方法的所有参数，这些参数的类型也是int、short、char、byte和String中的某一种；第24语句至第39行用于检测方法中是否存在if语句(IfStatement)或Switch语句(SwitchStatement)并进行相应的处理；如果只有一个独立的if语句，则跳出循环(第25语句～第27语句)，否则判断条件表达式中是否有一个变量存在于fieldList或parameterList中，并且此变量必须位于一个包含“==”的中缀表达式中或者是equals()方法的参数(第29语句)，满足条件的变量即为类型码(Type Code)；如果存在满足条件的类型码，则将类名、方法名和变量名保存到类型为RefactoringInfoObject的变量refInfoObj中(第30语句～第33语句)；然后将refInfoObj增加到resultList集合中(第34语句)；最后再将resultList返回(第42语句)。第30语句中的RefactoringInfoObject类是一个自定义数据结构，其成员变量包括className、methodName和variableName，分别用于存储类名、方法名和类型码变量名。

对上述算法进行时间复杂度分析：对于有n个程序源代码文件的系统，一共需要创建n棵抽象语法树，因此，外层for循环需执行n次；在外层for循环中，还需要对每个类的成员变量和成员方法进行遍历，假定有m个成员变量和k个成员方法，内层for循环的执行总次数为$m+k$，整个算法的执行时间$T(n)=n(m+k)$。根据类的单一职责原则，设计良好的类的成员变量和成员方法的个数m和k通常不会太大，因此，定义一个合适的常数为C，可认为：$1\leqslant(m+k)\leqslant C$，$T(n)=n(m+k)\leqslant Cn=O(n)$，即算法的执行时间与源代码文件的个数$n$近似线性关系，源代码文件个数也近似等于类的个数(在Java、C#等语言中存在内部类机制，一个源代码文件中可能存在多个类)。总之，算法执行时间与系统规模成正比，该重构时机识别算法具有较高的执行效率。

3 代码自动重构研究

在正确识别重构时机后，将获得所有待重构的候选类的类名和方法名。软件开发人员可以进行进一步人工检查，决定是否执行自动重构。自动重构程序将对原有类进行修改，创建相应的子类，提取

原有类中的分支语句并将其分散到子类中。本文的自动重构算法只适用于一个方法的方法体全为基于类型码的条件语句的情况，在后续研究中将考虑更多更为复杂的情况。本文所提出的基于多态的自动重构算法的伪代码如下所示。

Input: A candidate class name(cName), method name(mName) and type code variable name(tName).
Output: The source code files after refactoring.

```
1  if a source code file is named cName + ".java"
2     create an AST for the file
3     for each MethodDeclaration in the AST
4        if the method name is mName
5           for each branch of conditional statements
6              if one of the variables in the condition
               expression is tName, and if the node is
               IfStatement and the variable is in an
               infix expression which has a "==="
               operator or it is a parameter of the
               equals() method
7                 create a subclass named cName +
                  type code value
8                 add a method in the new subclass, the
                  method has the same signature as the
                  method of mName
9                 copy the branch code fragment to the
                  new added method in subclass
10             end if
11          end for
12       end if
13       remove all code in method mName
14    end for
15    save all source code files
16 end if
```

算法伪代码中，首先根据类名找到对应的源文件，然后对指定的方法进行重构，从第5语句开始，对方法中的每一个条件分支语句，首先创建一个名为(cName+类型码值)的子类，再在该子类中增加一个与待重构方法保持相同方法签名的方法，将分支代码片段拷贝至子类方法中，待所有分支都处理完毕之后，再将原方法中所有代码都移除，最后保存重构之后的父类和子类。通过上述算法，将基于类型码的条件逻辑转移至一系列子类，提高代码的可理解性、可扩展性和可维护性。

通过分析不难得知，单个类的重构算法的执行时间$T=kp$，其中k为一个类所包含的方法数，p为待重构方法中包含的条件分支的个数。对于一个设计良好的类而言，其方法数k通常不会太大，可认为小于等于某一常数C_1；同理，方法中的条件分支个数p也不会太大，也可认为小于等于某一常数C_2，因此$T=kp\leqslant C_1C_2=C=O(1)$。即对于待重构的类而言，执行时间基本维持在一个时间范围内，它会随方法个数和条件分支数量的不同有一定的变化，但变化幅度并不是特别显著。因此，该重构算法的执行效率较高。

4　实验及结果分析

为了验证算法的正确性和执行效率，本文选取4个开源项目进行重构时机识别和自动重构实验。实验从3个方面开展，包括1) 重构时机识别算法的准确率，以此判断算法能否准确识别出重构时机；2) 自动重构算法的正确率，以此判断算法能否正确地对候选代码实施自动重构；3) 算法的效率，分析与评价算法的执行效率。

本文选取了4个规模不一的开源Java项目，基本信息如表1所示。

表1　待测项目基本信息表

基本信息	JHotDraw	IceHockeyManager	JRefactory	Apache Ant
版本号	5.1	0.3	2.6.24	1.9.1
代码行数	8 419	18 085	55 711	105 815
源代码文件数	144	218	563	858
类(接口)的数量	155	222	569	1 165
成员变量个数	331	1 432	1 366	5 586
方法个数	1 314	1 664	4 854	10 262

为了更好地定量评价探测结果，本文引入精确率(precision)来分析重构时机识别算法的准确性，通过人工审查来判断识别到的重构时机是否正确，精确率的计算公式为：

精确率(precision)=TP/(TP+FP)

其中，TP(true positive，真阳性)表示识别到的正确重构时机数量；FP(false positive，假阳性)表示识别到的错误重构时机数量，即人工审查后发现识别出的重构时机是错误的。通过对识别出的重构时机进行逐个审查可以得到TP和FP值。表1中4个待测项目的重构时机识别结果的精确率如表2所示。

从表2可知，4个项目的精确率都已达到100%，说明所有识别到的重构时机都是正确的，没有假阳性重构时机实例，算法具有良好的准确性。

表2　重构时机识别结果精确率一览表

项目	TP	FP	TP + FP	精确率/%
JHotDraw	1	0	1	100
IceHockeyManager	8	0	8	100
JRefactory	28	0	28	100
Apache Ant	4	0	4	100

在准确识别出重构时机之后，再对候选类执行自动重构，并对自动重构结果进行分析。通过使用JUnit进行回归测试和人工审查相结合的方法来判断重构后代码的正确性。由于自动重构算法只考虑方法体全部为条件语句的情况，如果一个方法中除条件语句中的代码外还存在其他代码则本重构算法不能实施有效重构。通过人工检查，从所识别到的候选实例中选取3个满足重构条件的实例进行自动重构实验，为每个重构候选实例设计3个测试用例，最终执行结果如表3所示。

表3 自动重构算法正确性实验结果(部分)

项目	文件名/编译单元名	方法名	分支个数	测试用例个数	重构前(通过用例数)	重构后(通过用例数)
IceHockey Manager	MatchSnapshot	finishPeriod	2	6	6	6
JRefactory	JavaParserTokenManager	jjStopStringLiteralDfa_0	11	33	33	33
Apache Ant	JavaDoc	processLine	2	6	6	6

从表3可以得知，无论是在重构之前还是在重构之后，这些测试用例都能够正确通过。由此可知，重构之后的代码并未引入新的错误，重构算法能够在保证功能不受影响的情况下对代码进行自动重构，进而提高代码质量。

除了对重构时机识别算法和自动重构算法的正确性进行分析外，本文还对算法的性能展开研究与分析。重构实验在一台预装Windows 7操作系统的PC机上进行，该PC机的配置为：双核2.67 GHz，2 GB DDR2 RAM。首先对4个项目进行重构时机识别实验，对于每一个项目，算法均执行5次，然后计算平均时间，结果如表4所示。

表4 重构时机识别执行时间

项目	源文件数	代码行数	识别重构时机数量	平均时间/ms
JHotDraw	144	8 419	1	1 915.4
IceHockeyManager	218	18 085	8	2 358.8
JRefactory	563	55 711	28	3 441.6
Apache Ant	858	105 815	4	5 041.8

由表4可知，随着系统规模的增大(源文件数量和代码行数的增加)，程序平均执行时间也逐步增加，以表4中的源代码文件数为*X*轴，重构时机识别程序的执行时间为*Y*轴，可得图1所示源文件数-执行时间图。

在图1中，图中细斜线为拟合的线性趋势线。根据前面对表1中算法的分析，由于在一个类中的成员变量和成员方法个数都不会太多，可以认为小于某一个常数*C*，因此重构时机识别算法的时间复杂度近似为$O(n)$，即执行时间与系统规模(源文件数量或类的数量)成正比，这与图1的趋势线基本一致，随系统规模的增大执行时间增加，且整体呈现线性关系，增长幅度并不大，算法具有较好的执行效率。

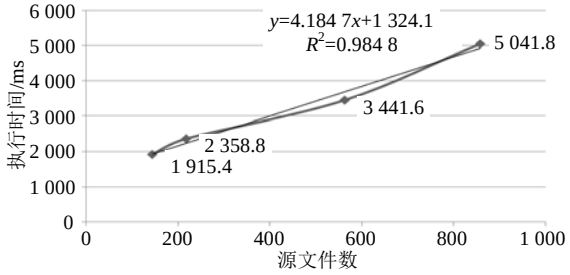

图1 重构时机识别算法源文件数-执行时间图

对表3中的3个类进行自动重构实验，每个类的重构过程执行5次，然后计算平均时间，结果如表5所示。

表5 自动重构执行时间(部分)

项目	文件名/编译单元名	成员方法数	方法名	分支个数	平均执行时间/ms
IceHockey Manager	MatchSnapshot	18	finishPeriod	2	62
JRefactory	JavaParserTokenManager	41	jjStopStringLiteralDfa_0	11	220.4
Apache Ant	JavaDoc	155	processLine	2	99.4

从表5可以得知，随着成员方法数和分支个数的增加，重构算法的执行时间将有所增加，由于一个类的成员方法数和方法中分支的个数并不会太多，因此执行时间也不会太长，执行时间与成员方法数与分支个数成正比，算法具有较好的执行性能。

5 结 论

本文提出了一种基于抽象语法树和面向对象多态机制的复杂条件语句重构时机识别方法和自动重构方法。该方法首先将源代码转换成抽象语法树，然后对抽象语法树进行解析，识别出其中的类型码和对应的条件分支，找出所有的重构候选类和方法，最后对指定的类及其方法执行自动重构算法。在对复杂条件语句进行重构时，引入了多态机制，将原本集中在一个方法中的条件分支语句分散到一系列子类中，每一个子类对应一个条件分支。使用多态来取代条件式，可提高代码的可理解性、可测试性、可维护性和可扩展性。

在后续工作中，会对算法进行进一步完善。

1) 考虑构造函数中的复杂条件语句，对构造函数实施自动重构；2) 在自动重构算法的实现中，考虑复杂条件语句只是方法一部分的情况，根据上下文环境来确定子类方法的方法签名和方法体中的代码；3) 结合策略模式、状态模式等设计模式来进一步对重构后的代码进行调整和优化[13]，实现将类型码替换为状态/策略(replace type code with state/strategy)等重构手段的自动化。

参 考 文 献

[1] FOWLER M. Refactoring: Improving the design of existing code[M]. Massachusetts: Addison-Wesley, 1999.

[2] TSANTALIS N, CHATZIGEORGIOU A. Identification of move method refactoring opportunities[J]. IEEE Transaction on Software Engineering, 2009, 35(3): 347-367.

[3] TSANTALIS N, CHATZIGEORGIOU A. Identification of extract method refactoring opportunities for the decomposition of methods[J]. Journal of Systems and Software, 2011, 84(10): 1757-1782.

[4] FOKAEFS M, TSANTALIS N, STROULIA E, et al. Identification and application of extract class refactorings in object-oriented systems[J]. Journal of Systems and Software, 2012, 85(10): 2241-2260.

[5] HIGO Y, KUSUMOTO S, INOUE K. A metric-based approach to identifying refactoring opportunities for merging code clones in a Java software system[J]. Journal of Software Maintenance and Evolution: Research and Practice, 2008, 20(6): 435-461.

[6] HOTTA K, HIGO Y, KUSUMOTO S. Identifying, tailoring, and suggesting form template method refactoring opportunities with program dependence graph[C]// Proceedings of 2012 16th European Conference on Software Maintenance and Reengineering(CSMR). Szeged: [s.n.], 2012: 53-62.

[7] BAVOTA G, OLIVETO R , DE LUCIA A, et al. Playing with refactoring: Identifying extract class opportunities through game theory[C]//Proceedings of the 2010 IEEE International Conference on Software Maintenance (ICSM). Timisoara: [s.n.], 2010: 1-5.

[8] BAVOTA G, De LUCIA A, OLIVETO R. Identifying extract class refactoring opportunities using structural and semantic cohesion measures[J]. Journal of Systems and Software, 2011, 84(3): 397-414.

[9] LIU Hui, NIU Zhen-dong, MA Zhi-yi, et al. Identification of generalization refactoring opportunities[J]. Automated Software Engineering, 2013, 20(1): 81-110.

[10] MURPHY-Hill E, PARNIN C, BLACK A P. How we refactor, and how we know it[J]. IEEE Transactions on Software Engineering, 2012, 38(1): 5-18.

[11] XI Ge, DUBOSE Q L, MURPHY-Hill E. Reconciling manual and automatic refactoring[C]//Proceedings of the 2012 34th International Conference on Software Engineering (ICSE). Zurich: [s.n.], 2012: 211-221.

[12] KUHN T, THOMANN O. Eclipse corner article abstract syntax tree [EB/OL]. (2006-11-20). http://www.eclipse.org/articles/Article-JavaCodeManipulation_AST/index.html.

[13]KERIEVSKY J. Refactoring to patterns[M]. Massachusetts: Addison-Wesley, 2004.

编 辑 蒋 晓