函数抽取重构的自动检测方法

刘 阳 刘秋荣 刘 辉
(北京理工大学计算机学院 北京 100081)

摘 要 软件重构历史的自动检测是目前软件重构领域的一个研究热点。其主要目的是方便程序员或软件维护人员理解软件演化的历史,也便于根据服务代码重构历史对其客户代码进行相应的重构操作。虽然相关研究人员已经提出了多种自动化的重构历史检测方法,但目前未见关于函数提取重构历史检测的方法或工具。为此,提出了一种基于版本比较的函数抽取重构自动检测方法,实现并验证了该方法的有效性。在8个开源项目上进行了实验验证,结果表明其查准率为65%~90%。此外,在一个小型项目上通过监控程序员的重构操作获得了全部的函数提取重构操作,进而计算出检测算法的查全率和查准率均为85%。

关键词 软件重构,重构历史,自动检测,版本,软件演化
中图法分类号 TP311.5 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2015.12.024

Automated Detection of Extract Method Refactorings

LIU Yang LIU Qiu-rong LIU Hui
(School of Computer Science and Technology, Beijing Institute of Technology, Beijing 100081, China)

Abstract Automated detection of refactorings is a hot topic in the field of software refactoring. The main purpose of the detection is to facilitate the understanding of software evolution and refactoring on clients according to changes made on their servers. Although a number of methods and tools have been proposed to automatically detect refactorings, to the best of our knowledge there is no methods or tools to detect and extract method refactorings automatically by comparing two versions of an application. To this end, we proposed an approach to detect and extract method refactorings by comparing two successive versions of a given application. We also implemented the proposed approach and validated it with open source applications. Evaluation results on 8 open-source applications suggest that the precision of the proposed approach varies from 65% to 90%. We also conducted an evaluation by monitoring developers on a small application. And evaluation results suggest that the recall and precision of the proposed approach is 85% and 85%, respectively.

Keywords Software refactoring, Refactoring history, Automated detection, Version, Software evolution

1 引言

软件重构是在不改变软件外部行为特性的情况下通过调整软件的内部结构来提高软件的质量,尤其是软件的可理解性和可扩展性[1,2]。软件重构技术已经获得了广泛的使用,主流的集成开发环境,比如Eclipse、Visual Studio、IntelliJ IDE等都在顶层菜单中提供了一个单独的重构菜单,为数十种重构提供自动化或半自动化的支持。

重构活动的自动检测是目前软件重构领域的一个研究热点。给定某个程序的两个相邻版本$\langle v_1, v_2\rangle$,重构检测工具将自行判定两个版本之间的哪些变化是由软件重构引起的,以及分别是什么类型的重构操作。重构检测的主要目的在于方便程序员或者软件维护人员迅速了解程序变化的原因[4,5]。此外,服务代码的重构历史数据也有利于程序员及时准确地更新相应的客户代码[3]。因此,提出了多种自动或半自动的重构检测方法[3-5]。

函数抽取重构(Extract Method)是一种非常常见的软件重构[6],但现有重构检测工具尚不能自动检测函数抽取重构。因此,提出了一种自动检测函数抽取重构的方法,并通过实验验证了该方法的有效性。

2 函数抽取检测算法

函数抽取检测算法每次比较的是相邻的两个版本Vnew和Vold,其中Vnew表示较新发布的版本,Vold表示较老的版本。新版本中的函数称为新函数,用*Mnew*表示新版本的所有函数。旧版本中的函数称为旧函数,用*Mold*表示旧版本的所有函数。

检测算法主要分为两步:

1)查找新版本中所有新增加的函数。用$M+$表示新版本中新增加的所有函数。因为经过函数抽取得到的函数在旧版本中必然不会存在,它们一定是新版本中新增加的函数。

2)函数抽取重构判断。对于上一步找到的每一个新增加的函数,判断其是否是经过函数抽取得到的函数。

2.1 查找新版本中新增加的函数

首先取得新、旧版本各自的函数集合*Mnew*、*Mold*,然后得到版本之间的函数匹配集*MatchTable*,新版本中不能匹配

到稿日期:2015-02-10 返修日期:2015-04-05 本文受国家自然科学基金(61272169,61472034),教育部"新世纪优秀人才支持计划"(NCET-13-0041),北京高等学校"青年英才计划"(YETP1183)资助。

刘 阳(1991-),女,硕士,主要研究方向为软件重构;刘秋荣(1988-),男,硕士,主要研究方向为软件重构;刘 辉(1978-),男,博士,副教授,主要研究方向为软件工程、软件演化与维护,E-mail:Liuhui08@bit.edu.cn(通信作者)。

到旧版本的函数即为新增加的函数。用 M 来表示函数集中的任意一个函数。函数集中的每一个函数 M 都包含了与该函数相关的信息，可以用一个四元组来表示：

$M=(Package, Class, Method, Parameter)$

其中，*Package* 表示该函数所在的包名，*Class* 表示该函数所在的类名，*Method* 代表该函数的函数名，*Parameter* 为该函数的参数列表。

版本之间首先按照函数的基本信息即四元组进行函数匹配，如果两个函数的基本信息相同，就认为它们是匹配的。但是由于重命名(rename)、移动(move)等重构操作可能会影响函数名、包名、类名，这个匹配集不是很全面，可能会遗漏某些本来可以匹配的函数。因此对于新版本中按照基本信息不能得到匹配的新函数，采用文献[11]中用来匹配版本之间函数的工具进行匹配，这个工具通过推断规则来匹配版本之间的函数。这样便可以得到全部的函数匹配集 *MatchTable*。

得到函数的匹配集 *MatchTable* 后，便可以得到新版本中可以匹配到旧版本的函数集 *Mnewmatch*，新版本中新增加的函数 $M+=Mnew-Mnewmatch$。

2.2 函数抽取重构判断

对于在新版本中找到的所有新增加的每一个函数，判断其是否是经过函数抽取(ExtractMethod)得到的函数。具体算法如下：

```
for every M in M+
get MReference
//MReference 表示所有调用该函数的函数集合
  for MRnew in MReference
  Search the method MRold who matches MRnew in MatchTable
//在 MatchTable 中查找与 MRnew 匹配的旧版函数 MRold
if (MRold =null) continue
else
{
  Compute similarity:
  //计算相似度
        Patch1=MRold-MRnew
        Patch2=M-Patch1
  similarity=1-(Total Lines of Patch2 in M)/(Total Lines of M)
  if(similarity>similarity threshold)
  M is ExtractMethod
}
```

对于新版本中新增加的每一个函数 M，首先找到在新版本内调用 M 的所有函数 *MReference*。对 *MReference* 集合中的任一元素 *MRnew*，在版本间的函数匹配集 *MatchTable* 里搜索与 *MRnew* 匹配的旧版函数 *MRold*。如果对于所有的调用函数，在匹配集里都找不到与该函数匹配的函数，那么这个新增加的函数肯定不是函数抽取(ExtractMethod)。如果于调用函数集合 *MReference* 找到了它们(MReference 中一个或多个元素均可)在旧版本中对应的匹配函数，那么接下来就计算旧版本中的函数 *MRold* 与新版本中的函数 *MRnew* 的差集和新增加的函数 M 的相似度。若相似度大于某个阈值，这个新增加的函数 M 即为函数抽取(ExtractMethod)。使用工具 Diff[7] 来比较函数体相似度的计算。首先得到 *MRold* 和 *MRnew* 之间的差集 *patch*1，然后得到 M 和 *patch*1 的差集 *patch*2，从而得到 M 与 *patch*1 不相同的代码行数，相似度定义为 M 与 *patch*1 相同的行数占 M 的比例。

3 实验验证

为了验证本文提出的方法的有效性，将该方法实现为 Eclipse插件，并在 8 个 Java 开源项目上进行了实验验证。实验结果表明所提方法具有较高的查准率和查全率。

3.1 实验对象

在 sourceforge. net 和 apache 上选择了 8 个 Java 开源项目，每个项目的版本个数为 11～73 不等。表 1 列出了这几个开源项目的详细信息。

表 1 实验项目信息

项目名称	版本个数	最旧版本函数个数	最旧版本代码行数	最新版本函数个数	最新版本代码行数
Ant	16	7736	86857	9548	107044
DoubleType	28	1115	22567	3091	47501
Findbugs	11	6299	81563	8807	123259
Hibernate	45	7627	68929	16107	176879
JEdit	33	1059	23881	6148	115919
Jfreechart	50	641	6978	8039	99375
PMD	19	3508	43063	5470	50011
Weka	73	1497	27947	18137	275519

具体实验项目简要介绍如下：

1) Ant(http://archive. apache. org/dist/ant)是一个自动将编译、测试、部署等过程进行打包的工具。本次实验 Ant 版本个数为 16，单个版本的代码行数为 86857～107044(每个版本略有变化)，单个版本的函数个数为 7736～9548(每个版本略有变化)。

2) DoubleType(http://sourceforge. net/projects/doubletype)是一种类型设计器(字体编辑器)，建立 TrueType 字体文件。本次实验 DoubleType 版本个数为 28，单个版本的代码行数为 22567～47501，单个版本的函数个数为 1115～3091。

3) Findbugs(http://sourceforge. net/projects/findbugs/)是一款静态分析工具，用来检测类或者 jar 包可能存在的问题。本次实验 Findbugs 版本个数为 11，单个版本的代码行数为 81563～123259，单个版本的函数个数为 6299～8807。

4) Hibernate (http://sourceforge. net/projects/hibernate/)是一个对象关系映射框架。本次实验 Hibernate 版本个数为 45，单个版本的代码行数为 68929～176879，总单个版本的函数个数为 7627～16107。

5) JEdit(http://sourceforge. net/projects/jedit/)是一个跨平台的文本编辑器。本次实验 JEdit 版本个数为 33，单个版本的代码行数为 23881～115919，单个版本的函数个数为 1059～6148。

6) JfreeChart(http://sourceforge. net/projects/jfree-chart/)是一个可以用来画各种不同类型图表的库。本次实验版本个数为 50，单个版本的代码行数为 6978～99375，单个版本的函数个数为 641～8039。

7) PMD(http://sourceforge. net/projects/pmd/)是一款用来静态分析 Java 源代码错误的工具。本次实验 PMD 版本个数为 19，单个版本的代码行数为 43063～50011，单个版本的函数个数为 3508～5470。

8) Weka(http://sourceforge. net/projects/weka/)是一个数据挖掘和机器学习的软件。本次实验 Weka 版本个数为 73，单个版本的代码行数为 27947～275519，单个版本的函数个数为 1497～18137。

3.2 阈值设定

本文提出的检测方法包括一个相似度阈值，用于判定某个新出现的函数是否为函数抽取重构的结果。为了设定最佳阈值，在一个测试项目上进行了调参。

这个测试项目是实验室一个科研人员曾经做过的项目，他重新查看这个项目的代码来考虑哪些地方应该进行函数抽取重构并进行记录。共进行了 20 个函数抽取重构。取相似度阈值从 0.1 到 0.9 进行了实验，得到的查准率、查全率如表 2 所列(与程序员操作记录比较)。

表 2　相似度阈值与查准率、查全率的关系

相似度阈值	检测结果总个数	为真个数	查准率	查全率
0.1	24	19	79%	95%
0.2	23	19	82%	95%
0.3	23	19	82%	95%
0.4	21	17	80%	85%
0.5	20	17	85%	85%
0.6	15	13	86%	65%
0.7	13	11	84%	55%
0.8	11	10	90%	50%
0.9	11	10	90%	50%

图 1 更直观地展现了相似度阈值的取值对实验结果的影响，其中横轴表示相似度阈值的变化，纵轴为查准率和查全率随着相似度的变化。从图中可以看出，当相似度阈值太大时，可以提高查准率但是会降低查全率，而相似度阈值太小时，查全率会增大但是查准率会下降。在相似度阈值为 0.5 时，查准率与查全率基本平衡。因此，后续实验中采用 0.5 作为相似度阈值。

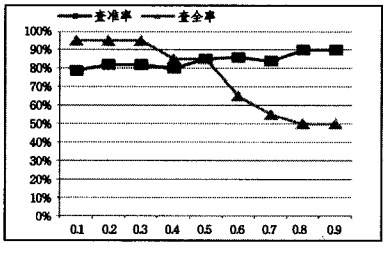

图 1　相似度阈值对查准率与查全率的影响

3.3　实验结果及分析

对以上 8 个开源项目进行了实验(相似度阈值取值为 0.5)，总共检测到 1393 个函数抽取重构。对检测的结果，由独立程序员进行核对确定检测结构是否正确。表 3 列出了每一个开源项目检测到的函数抽取重构的个数以及查准率。实验结果表明，本文提出的函数抽取重构检测方法的平均查准率约为 80%，在不同的项目上，其查准率在 65%到 90%之间略有波动。

表 3　检测结果 3

项目名称	检测到的函数抽取重构数量	查准率
Ant	229	80%
DoubleType	30	70%
Findbugs	117	85%
Hibernate	160	90%
JEdit	271	75%
Jfreechart	303	65%
PMD	45	90%
Weka	238	85%

结束语　软件重构及重构检测是软件工程领域的研究热点之一。本文提出了一种函数抽取重构的自动化检测方法，实现了相关算法并进行验证。在 8 个大型开源软件系统上的实验结果表明，该方法具有较高的准确率。

后续工作主要包括以下几个方面：首先，将该方法扩展至其他编程语言，比如 C、C++等；其次，计划在更多开源系统上进行进一步的实验验证。目前的实验采用了 8 个大型开源系统的 275 个版本，涉及的代码行数超过一千万。

参 考 文 献

[1] Mens T, Tourwe T. A survey of software refactoring [J]. IEEE Transactions on Software Engineering, 2004, 30(2): 126-139

[2] Opdyke W F. Refactoring object-oriented frameworks[D]. Llinois: University of Illinois at Urbana-Champaign, 1992

[3] Dig D, Comertoglu C, Marinov D, et al. Automatic detection of refactorings in evolving components[C]// European Conference on Object Oriented Programming. 2006: 404-428

[4] Xing Z, Stroulia E. UMLDiff: an algorithm for object-oriented design differencing[C]// Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering. 2005: 54-65

[5] Weissgerber P, Diehl S. Identifying refactorings from source-code changes[C]//21st IEEE/ACM International Conference on Automated Software Engineering. 2006: 231-240

[6] Murphy G, Kersten M, Findlater L. How are Java software developers using the eclipse IDE[J]. Software IEEE, 2006, 23(4): 76-83

[7] http://code.google.com/p/java-diff-utils/

(上接第 104 页)

[10] Hailpern B, Santhanam P. Software debugging, testing, and verification[J]. IBM Systems Journal, 2002, 41(1): 4-12

[11] Arcuri A. On the automation of fixing software bugs[C]// Proc. of the 30th International Conference on Software Engineering (ICSE). Leipzig, Germany, 2008: 1003-1006

[12] Arcuri A. Evolutionary repair of faulty software[J]. Applied Soft Computing, 2011, 11(4): 3494-3514

[13] Wei Y, Pei Y, Furia C A, et al. Automated fixing of programs with contracts[C]// Proc. of the International Symposium on Software Testing and Analysis (ISSTA). Trento, Italy, 2010: 61-72

[14] Wei Y, Pei Y, Furia C A, et al. Inferring better contracts[C]// Proc. of the 33rd International Conference on Software Engineering (ICSE). Honolulu, Hawaii, USA, 2011: 191-200

[15] Kim D, Nam J, Song J, et al. Automatic patch generation learned from human written patches[C]// Proc. of 35th International Conference on Software Engineering (ICSE). San Francisco, California, 2013: 802-811

[16] Qi Yu-hua, Mao Xiao-guang, Lei Yan, et al. Using automated program repair for evaluating the effectiveness of fault localization techniques[C]// The 22th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA). Lugano, Switzerland, 2013: 191-201

[17] Weiser M. Program slicing[J]. IEEE Transactions on Software Engineering, 1984, 10(4): 352-357

[18] Korel B, Laski J. Dynamic Program Slicing[J]. Information Processing Letters, 1988, 29(3): 155-163

[19] Agrawal H, Horgan J R. Dynamic program slicing[C]// Proceedings of the Conference on Programming Language Design and Implementation (PLDI). 1990: 246-256