文章编号:1000-5862(2018)05-0464-06

基于版本的多重软件重构自动检测技术研究

钟林辉,黄小明,薛良波,叶海涛
(江西师范大学计算机信息工程学院,江西 南昌 330022)

摘要:软件重构的自动检测是目前软件重构领域的一个研究热点.目前,多重软件重构的自动检测方法能够检测出在不同软件版本的不同位置上实施的多重重构操作,但是对发生在不同软件版本的相同位置上的多重重构操作则无能为力.为此,该文提出了一种在函数级别,利用函数调用图,实现基于多版本的多重软件重构的自动检测方法.该方法能够自动检测出在不同软件版本中的同一个函数上发生的"函数抽取"和"函数重命名"多重重构操作.同时,通过实验验证了该方法的有效性.

关键词:软件重构;软件版本;函数抽取;函数重命名
中图分类号:TP 311 **文献标志码**:A **DOI**:10.16357/j.cnki.issn1000-5862.2018.05.05

0 引言

软件重构[1]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质量的方法.在软件重构的研究领域中,大致可以分为 2 个方向.第 1 个方向是研究可行的软件重构操作,例如为了提高函数的内聚性,设计"抽取函数"这一重构操作;第 2 个方向是研究软件重构的自动检测技术,也是目前软件重构领域的一个研究热点.软件重构的检测指的是在工具的支持下,对给定软件(通常是源程序)的 2 个版本,能自动地判断从前一个版本变化到后一个版本的过程中包含了哪些软件重构操作.重构检测的主要目的是方便程序员或者软件维护人员迅速了解程序变化的原因,以及利用检测出来的重构历史数据及时准确地更新相应的客户代码[2-3].因此,学术界提出了多种自动或半自动的重构检测方法[2,4-7].

在已有的研究中,F. F. Silva 等[8]提出了一种基于最优化算法的差异性比较工具,能准确地检查出不同粒度的代码元素移动(例如将类移动到另一个包中;将方法迁移到其他类中)而导致的软件差异.F. N. Rysselberghe 等[9-11]则利用克隆检测器来检测移动的方法;Xing Zhenchang 等[12]开发了一个支持面向对象程序差异性比较的工具——UMLDiff,能根据元素名字和结构的相似性自动匹配包、类、接口、属性等,进而推断可能存在的重构操作.B. Biegel 等[13]则分析了目前 3 种基于相似性度量(基于文本的、基于 AST 和基于词汇的)的软件重构检测方法的优点和不足.

上述这些软件重构检测方法能检测出一种重构操作或者发生在源代码的不同位置上的多个重构操作,但是对相同位置上的多重重构检测则缺乏必要的支持.本文以包含"函数抽取"和"函数重命名"这 2 种重构操作的多重重构为例,研究发生在相同位置上多重重构的检测方法.

1 多重重构的自动检测方法

1.1 问题描述及检测主要步骤

多重重构检测首先需要选取软件版本历史中的 2 个不同版本 V_m 和 V_n,这 2 个版本可以是相邻的 2 个版本,也可以是不相邻的 2 个版本.例如,对于一个整数先做加法运算、再做减法运算和最后求该整数平方的简单程序,选取其 2 个不同的版本.

版本 V_m:

```
package com. test;
```

收稿日期:2018-03-16
基金项目:国家自然科学基金(61462040,61662032,61262015,61762049),江西省自然科学基金(20142BAB207027,20171BAB202013)和江西省教育厅科学技术(GJJ170207)资助项目.
作者简介:钟林辉(1974-),男,江西赣州人,教授,博士,主要从事软件体系结构、构件化软件、软件维护与软件演化研究.E-mail:shiningto@jxnu.edu.cn

```
public class Test {
    public static int num = 0;
    public static void main( String [] agrgs) { calcu-
late( )
    }
    public static void calculate( ) {
        //加 5
        for( int i = 0; i < 5; i + + ) num = num + i;
        //减 3
        for( int i = 0; i < 3; i + + )
            fun( ) ;
            num = num - 1;
        //求平方
        num = num* num;
    }
    public static void fun( ) {
    System. out. println( ) ;
    }
}
```

版本 V_n:

```
package com. test;
public calss Test {
public static int num = 0;
    public static void main( String [] agrgs) { calcu-
lating( ) ;
        System. out. println( num) ;
    }
    //函数名发生改变
    public static void calculating( ) {
        add( ) ; //加 5
        del( ) ; //减 3
        num = num* num; //求平方
    }
    public static void add( ) {
        for( int i = 0; i < 5; i + + ) acc( ) ;
    }
    public static void del( ) {
        for( int i = 0; i < 3; i + + ) refun( ) ;
    }
    public void acc( ) {
        num = num + 1;
    }
    public static void refun( ) {
        num = num - 1;
        print( ) ;
    }
    public static void print( ) {
        System. out. println( num) ;
        if( num > 0)
            del( ) ;
    }
}
```

从版本 V_m 变化到版本 V_n 的过程中,对函数 com. test. Test. calculate 进行了"函数换名"重构(重新命名为 com. test. Test. calculating) 和"函数提取"重构(将 com. test. Test. calculate 中的部分实现抽取成 V_n 中的 com. test. Test. add 和 com. test. Test. del 函数) . 需要指出的是,这 2 个重构操作可能是直接由 V_m 变化到 V_n,即 $V_m \xrightarrow{rm,em,} V_n$,也可能是由 V_m 经中间版本 V_r 再变化到 V_n,即 $V_m \xrightarrow{rm,} V_r \xrightarrow{em} V_n$.

多重重构检测需要在给定版本 V_m 和 V_n 的前提下,能自动识别出发生了哪些多重重构. 针对上述问题,本文采用比较不同版本中函数方法体相似度的策略进行多重重构的检测,例如比较 V_m 版本中的 com. test. Test. calculate 函数和 V_n 版本中的 com. test. Test. calculating 函数方法体的相似度. 其主要步骤如下:

(i) 对给定软件的 2 个版本,依据函数名和参数列表,判断 2 个版本在函数集合上的不同之处;

(ii) 构造第 2 个版本的函数调用图,计算每个候选函数的调用图. 特别地,若调用图中存在环,则需要进行破环处理;

(iii) 基于无环调用图,实现候选函数方法体的合并. 在合并过程中,函数的方法体中需作格式化处理(例如去除注释、格式控制符号等) ,并采用自底向上的策略进行合并;

(iv) 多重重构判断,基于最长公共子序列(LCS) [9] 计算第 m 个版本中每个函数的方法体与第 n 个版本中候选函数方法体的相识度; 通过设定阈值进行多重重构的判断.

其中,需要解决的关键技术包括相似度的计算及函数方法体的合并(如 com. test. Test. calculating 函数因为调用了 com. test. Test. add 函数和 com. test. Test. del 函数,故需要将其方法体合并到 com. test. Test. calculating 函数中) .

1.2　函数方法体的相似度计算

本文采用基于最长公共子序列 L_{CS} 的相似度计算公式 $s_{imilar}(S_i, S_j)$ [23-24] .

$$s_{imilar}(S_i, S_j) = \frac{L_{CS}(S_i, S_j) \times 2}{L_{ength}(S_i) + L_{ength}(S_j)},$$

其中 S_i 表示存在于 V_m 版本中但不存在于 V_n 版本中的第 i 个函数的方法体; S_j 表示存在于 V_n 版本中但不存在于 V_m 版本中的第 j 个函数的方法体. $L_{CS}(S_i,S_j)$ 表示共同存在于 S_i 及 S_j 中的最长公共子序列,且该子序列不一定要连续地出现在 S_i 或 S_j 中,只要出现的顺序和在 S_i 和 S_j 内出现的顺序相同即可.

1.3 基于调用图的函数方法体合并

使用调用图的目的在于找到 V_n 版本中函数之间的调用关系,并利用调用关系进行函数方法体的合并. 由于调用图中可能存在循环调用,需要进行破环处理.

1.3.1 调用图

定义 1 软件版本 V_r 的调用图: 记为 $G_{V_r}=(V,E)$. 其中 V 是顶点的非空有限集合,其元素表示软件版本 V_r 中用户自定义函数,用包名、类名、函数名和参数列表的连接作为 V 中元素唯一标识; E 表示函数之间调用关系的集合. 需要指出的是,由于相同函数可能在一个函数方法体的多个位置被调用(循环语句中的多次被调用仅视为 1 次被调用),因此在边上记录被调用函数出现的位置次数(默认为 1). 例如图 1 中 2 表示在函数 a 的方法体中函数 b 出现了 2 次.

在调用图中,由于调用关系存在方向性,因此可能存在调用的环形结构,即调用环.

定义 2 调用环: 记为 $R=(e_1,e_2,\cdots,e_n)$, $n>1$,表示调用图 G 中由若干边 $(e_1,e_2,\cdots,e_n)$ 构成的特殊调用链. 其中 e_1 调用 e_2, e_2 调用 e_3, $\cdots$, e_{n-1} 调用 e_n, e_n 调用 e_1; 反之则为无调用环. 例如图 1 表示的是一个存在调用环 (d,e,f) 的调用图.

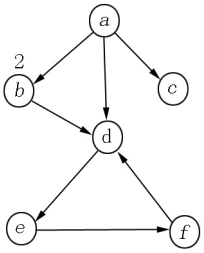

图 1 存在调用环的调用图

定义 3 候选函数 f: 对于 V_m 版本的调用图 G_{V_m} 和 V_n 版本的调用图 G_{V_n}, $f\in G_{V_n}(V)-G_{V_m}(V)$. 即存在于版本 V_n 中但不存在于版本 V_m 中的函数称为候选函数.

定义 4 候选函数的调用图 $EG_{V_r}=(V,E)$,是软件版本 V_r 的调用图 G_{V_r} 的子图,满足 $(EG_{V_r}(V)\subseteq G_{V_r}(V)\wedge EG_{V_r}(E)\subseteq G_{V_r}(E)$. 例如对于图 1 所示的调用图,假设 a、b、d、e、f 是候选函数,则其对应的候选函数调用图如图 2 所示.

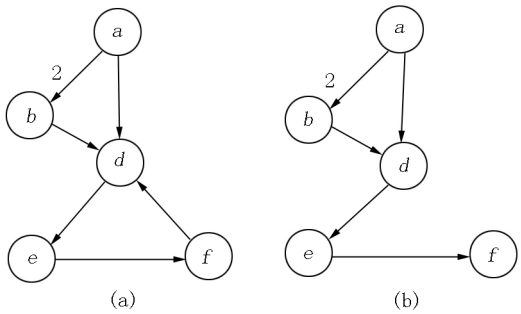

图 2 候选函数的调用图和从 a 节点出发的无环调用图

图 3 为求和问题中前一个版本(左)和后一个版本(右)的对应的调用图(图中函数均在 com. test. Test 类中).

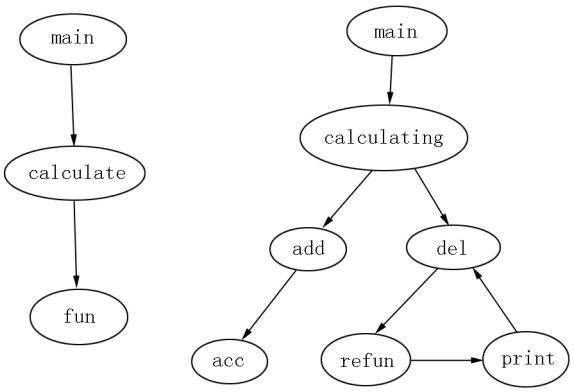

图 5 求和问题中前后版本中对应的调用图

1.3.2 调用环的破环处理

在对不同函数的方法体进行比较时,对存在函数调用关系的函数需要进行方法体的合并. 对于存在调用环的候选函数调用图,需要进行破环处理. 本文采用一种简单的处理方式,即从任意节点 v 出发,对其进行深度优先遍历,进而得到候选函数的调用树.

定义 5 候选函数 f 的无环调用图 $EG^f_{V_r}$: $EG^f_{V_r}$ 是针对 V_r 版本的候选函数调用图 $EG_r(V)$,若 $f\in G_{V_r}(V)$,则从 f 出发进行深度优先遍历中遇到的所有节点均不属于某个调用环中. 例如图 2(b)是一个无环调用图,而图 2(a)则是一个有环调用图(包含 $d\to c\to f\to d$).

为了得到一个候选函数的无环调用图,在从节点 f 出发进行遍历时,对遇到的节点 u 如果处于某个环中(例如 $u\to\cdots>t\to u$),则剔除边 $\langle t,u\rangle$. 事实上,采用这种方法,能保证从候选函数中任意节点出发,都能得到一个无环的调用关系. 算法如下:

算法: 获取候选函数的无环调用图;
输入: 候选函数调用图 g,候选函数名 f;
输出: 候选函数 f 的无环调用图 g'.

```
Function getNoCycleCallingGraph ( G g, Fun f )
{
Node t = g. getNodeByFun( f) ;
/* 从节点 t 出发,进行深度优先遍历,获取遍历过程中的所有节点* /
Node I visitedNodeSet = DFS( t) ;
/* 逐个检查每个节点是否位于一个环中,如果是则破环处理* /
for each k in visitedNodeSet Do {
    Node I cycleNodeSet;
/* cycleNodeSet 中存放从 cycleNodeSet [0] 到 cycleNodeSet [0] 的一个环,例如 a→b→a * /
/* 判断是否存在一个从 k 出发的到自己的一个环,若存在,存放在 cycleNodeSet 中. * /
if ( cylceNodeSet = g. existInCycle( k) ! = NULL)
    g. deleteEdge( k,
        cycleNodeSet [cycleNodeSet. length - 2] ) ;
}
    return g;
}
```

显然,多次调用 getNoCycleCallingGraph 能获得每个候选函数的无环调图. 例如,图 2 中候选函数 del 对应的无环调用图为 del→refun→print(此时已退化为一个单链表) .

1.3.3　函数方法体合并　基于获取的候选函数无环调用图,采用自底相上的策略逐步将被调用函数的方法体合并到调用函数中. 显然,合并后的候选函数方法体只包含对“基函数”的调用.

定义 6　基函数: 对于一个给定候选函数 f 的无环调用图 $G^f_{V_n}$,若满足 $\nexists$ (w: $w \in G^f_{V_n}(V)$ $v \in G^f_{V_n}(V)$ $\wedge$ $\langle w, v\rangle \in G^f_{V_n}(E)$) ,则 w 称为基函数. 即在无环调用图中,w 没有调用其他任何函数. 如图 2(b) 中 f 为基函数.

候选函数方法体合并算法如下:

算法: 获取合并后的候选函数方法体;
输入: 候选函数的无环调用图 g,候选函数名 f;
输出: 合并后的函数方法体.

```
Function getComposedBody( G g,Fun f)
{
/* 若函数 f 是基函数,则直接返回 f 函数方法体* /
    IF( Type( f) = = “基函数”) return f. Body;
/* 否则,依次递归获取被 f 函数调用函数的方法* /
    ELSE {
        Node t = g. getNodeByFun( f) ;
    While ( t. hasNextEdge) {
    /* 获取跟 t 具有调用关系的下一个后续函数* /
    Fun k = t. nextEdge;
    String body = getComposedBody( k) ; //递归调用
    f. body = f. body ∞ body
        }
    returnf. body;
    }
}
```

其中算子 ∞ 表示函数方法体的合并操作,如 $f1 \infty f2$ 表示在 $f1$ 中所有出现 $f2$ 的地方用 $f2$ 的方法体替换. 按此方法,对图 3 中的候选函数 calculating 进行方法体合并后得到的字符合并得到的字符串为:

for(int i = 0; i < 5; i + +) num = num + 1; for(int i = 0; i < 3; i + +) num = num - 1; System. out. println(num) ; if(num > 0) num = num* num.

1.4　多重重构判断

基于最长公共子序列(LCS) 计算 V_m 版本中每个函数的方法体与 V_n 版本中候选函数方法体的相似度; 通过设定阈值判断候选函数之间是否同时存在包含函数重命名重构和函数抽取重构的多重重构. 算法如下:

算法: 多重重构的判断;
输入: V_m 版本的调用图 $g1$,V_n 版本的调用图 $g2$,阈值 threshold;
输出: 返回包含多重重构的候选函数对.

```
Function continuousRefactoring ( G g1, G g2, double c) {
    Node f1,f2; FunPair continueRefactors = ∅;
//对 g1,g2 中候选函数合并后的方法体进行比较
For each f1 in g1 Do {
G t1 = getNoCycleCallingGraph( g1,f1) ;
String body1 = getComposedBody( t1,f1) ;
For each f2 in g2 Do{
    G t2 = getNoCycleCallingGraph( g1,f2) ;
    String body2 = getComposedBody( t1,f2) ;
    // 计算 body1 和 body2 的相似度 similar
    Similar = LCS( body1, body2) * 2/
        ( Length( body1) + Length( body2) ) ;
    /* 将每次求得的相似度与给定的阈值做比较,并给出判断结果 * /
    If similar > = threshold then
// continueRefactors 保存目前匹配成功的函数对
    continueRefactors = continueRefactors + < f1,
f2 >;
    } }
    returncontinueRefactors; }
```

当阈值为 0.7 时,V_m 版本和 V_n 版本所示的求和问题可能具有多重重构的候选函数对如表 1 所示.

表 1 阈值为 0.7 时的候选函数对

	calculating	add	acc	del	refun	print
calculate	√	×	×	×	×	×
fun	×	×	×	×	×	×

2 实验

2.1 实验对象

为了验证本文提出的方法,采用“背对背”的方式设计了一组测试案例,即测试案例的设计者根据自己对包含“函数抽取”和“函数换名”多重重构的理解,设计 27 个测试案例. 其中,真实存在多重重构的有 22 个(其中包含调用环的重构操作有 11 个),存在一个重构操作的有 4 个(其中,只包含“函数抽取”重构操作的有 2 个,只包含“函数换名”重构操作的有 2 个),不包含重构操作的有 1 个. 同时,可能存在不同类中具有相同方法的情况,在含有多重重构操作的实例中,有环和无环情况各设计一组在不同的类中包含相同方法名的实验用例.

2.2 实验过程及结果

函数方法体的相似度比较过程中设定了一个阈值,本文使用了 12 个实验例子进行调整参数,其中 10 个为真. 取相似度阈值从 0.1 到 0.9 进行了实验,得到的查准率、查全率及 F-measure 如表 2 所示. 实验中,2 个不同类中包含相同方法名的实验用例也可以使用本文方法检测出来.

图 4 更直观地展现了相似度阈值对实验结果的影响,其中横轴表示相似度阈值的变化(0.1 ~ 0.9),纵轴为对应阈值下查准率和查全率大小. 从图 4 可以看出,相似度阈值过大时会同时降低查准率和查全率,当阈值设置为 0.1 ~ 0.4 时,查准率在 70% 以上、查全率在 90% 以上; 当阈值设置为大于 0.4 时,查准率和查全率会逐渐下降.

表 2 阈值的设定

阈值	用例个数	检测为真个数	查准率/%	查全率/%	F-Measure/%
0.1	27	21	78	95	85.66
0.2	27	21	78	95	85.66
0.3	27	20	74	91	81.62
0.4	27	20	74	91	81.62
0.5	27	17	63	77	69.30
0.6	27	15	56	68	61.42
0.7	27	13	48	59	52.93
0.8	27	10	37	45	40.61
0.9	27	7	26	32	28.69

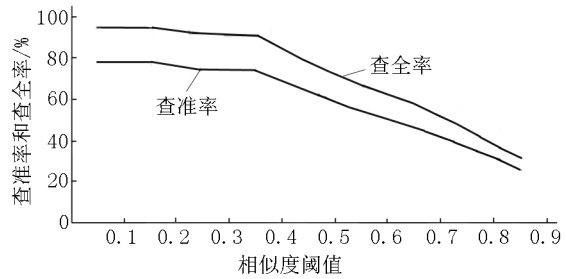

图 4 相似度阈值对查准率和查全率的影响

同时,本文与著名的重构检测工具 CCFinder[20-22] 进行了对比实验(如表 3 所示). 通过实验可以发现: 使用 CCFinder 重构检测工具对 27 组用例的检测结果分别为函数重命名检测查准率 44%、查全率 50% 和函数抽取检测查准率 48%、查全率 54%; 使用本文的方法,当阈值在 0.1 和 0.6 之间,单一函数(函数重命名和函数抽取)重构检测检测结果为查准率高于 56%、查全率高于 63%,即在此段阈值内,本方法的检测结果较好. 此外,本方法适用于多重重构检测,且查全率和查准率较高,而 CCFinder 检测工具不能检测系统版本中是否发生多重重构.

表 3 本文的算法与 CCFinder 检测方法的比较

检测方式	阈值	用例个数	检测出函数重命名个数	检测出函数抽取个数	检测出多重重构个数	单一重命名检测查准率/%	单一重命名检测查全率/%	单一函数抽取检测查准率/%	单一函数抽取检测查全率/%
OurWay	0.1	27	21	21	21	78	88	78	88
	0.2	27	21	21	21	78	88	78	88
	0.3	27	20	20	20	74	83	74	83
	0.4	27	20	20	20	74	83	74	83
	0.5	27	17	17	17	63	71	63	71
	0.6	27	15	15	15	56	63	56	63
	0.7	27	13	13	13	48	54	48	54
	0.8	27	10	10	10	37	42	37	42
	0.9	27	7	7	7	26	29	26	29
CCFinder	/	27	12	13	0	44	50	48	54

3　总结

目前对重构的检测方法众多，其主要是检测一个系统的 2 个版本中是否发生了重构，并试图检测出发生了哪些重构（例如“函数重命名”重构的检测[15-17]和“函数抽取”重构的检测[4,18-19]），但是这些方法和工具（如著名的重构检测工具 CCFinder）能够检测出来的重构都是发生在不同版本中不同的位置，若一个函数发生了多种重构，通常只能检测出其中一种重构.

因此，采用一种基于调用图的方法对一个系统 2 个版本进行重构检测，能够检测不同版本的 2 个函数是否同时发生了“函数抽取”和“函数重命名”2 种重构. 在今后的研究中，将在多重重构的检测中增加其他类型的重构（例如分解临时变量、替换算法等），并进一步增加对复杂案例的测试和完善支撑工具.

4　参考文献

[1] Fowler M，Beck K，Brant J，et al. Roberts，refactoring：improving the design of existingcode [M]. New Jersey：Addison-Wesley，1999.

[2] Dig D，Comertoglu C，Marinov D，et al. Automateddetection of refactorings in evolving components [C]. Berlin：Springer Berlin Heidelberg，2006：404-428.

[3] Soares G，Gheyi R，Murphyhill E，et al. Comparing approaches to analyze refactoring activity on software repositories [J]. Journal of Systems and Software，2013，86(4)：1006-1022.

[4] 刘阳，刘秋荣，刘辉. 函数抽取重构的自动检测方法 [J]. 计算机科学，2015，42(12)：105-107.

[5] Fontana F A，Braione P，Zanoni M. Automatic detection of bad smells in code：an experimental assessment [J]. Journal of Object Technology，2012，11(2)：1-38.

[6] Weissgerber P，Diehl S. Identifying refactorings from source-codechanges [C]. Washington：IEEE Computer Society，2006：231-240.

[7] Schmidt F，MacDonell S G，Connor A M. An automatic architecture reconstruction and refactoring framework [M]. Berlin：Springer Berlin Heidelberg，2012：95-111.

[8] Silva F F，Borel E，Lopes E，et al. Towardsa difference detection algorithm aware of refactoring-relatedchanges [EB/OL]. [2017-05-11]. 10.1109/SBES.2014.21.

[9] Hunt J W. A fastalgorithm for computing longest common subsequences [J]. Communications of the ACM，1977，20(5)：350-353.

[10] Rysselberghe F N，Demeyer S. Reconstruction of successfulsoftware evolution using clone detection [C]. Washington：IEEE Computer Society，2003：126.

[11] Roy C K，Cordy J R，Koschke R. Comparison and evaluation of code clone detection techniques and tools：a qualitative approach [J]. Science of Computer Programming，2009，74(7)：470-495.

[12] Xing Zhenchang，Stroulia E. Umldiff：an algorithm for object-oriented design differencing [EB/OL]. [2017-05-13]. 10.1145/1101908.1101919.

[13] Biegel B，Soetens Q D，Hornig W，et al. Comparison of similarity metrics for refactoring detection [EB/OL]. [2017-05-13]. http://www.st.uni-trier.de/~diehl/pubs/msr11.pdf.

[14] Malpohl G，Hunt J J，Tichy W F. Renaming detection [J]. Automated Software Engineering，2003，10(2)：183-202.

[15] Malpohl G，Hunt J J，Tichy W F. Renaming detection [EB/OL]. [2017-05-13]. https://www.researchgate.net/publication/3867122_Renaming_detection.

[16] Kim S，Pan K，Whitehead E J. When functions change their names：automatic detection of origin relationships [EB/OL]. [2017-05-13]. https://users.soe.ucsc.edu/~ejw/papers/kim-wcre2005.pdf.

[17] 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.

[18] Silva D，Terra R，Valente M T. Recommending automated extract method refactorings [EB/OL]. [2017-05-13]. http://www.ricardoterra.com.br/publications_files/2014_icpc.pdf.

[19] Prete K，Rachatasumrit N，Sudan N，et al. Template-based reconstruction of complex refactorings [EB/OL]. [2017-05-13]. http://web.cs.ucla.edu/~miryung/Publications/icsm10-reffinder-slides.pdf.

[20] Kim M，Gee M，Loh A，et al. Ref-Finder：a refactoring reconstruction tool based on logic query templates [EB/OL]. [2017-05-13]. http://web.cs.ucla.edu/~miryung/Publications/fse10-reffindertool.pdf.

[21] Prete K，Rachatasumrit N，Kim M. Catalogue of template refactoring rules [R]. UTAUSTINECE-TR-041610，University of Texas，2010.

[22] 王映龙，杨炳儒，宋泽锋，等. 基因序列相似程度的 LCS 算法研究 [J]. 计算机工程与应用，2007，43(31)：45-47.

[23] Apostolico，Guerra C. The longest common subsequence problem revisited [J]. Algorithmica，1987，2(1/2/3/4)：315-336.

[24] Deken. Some limit results for longest common subsequences [J]. Discrete Mathematics，1979，26(1)：17-31.

（下转第 472 页）

34(11):70-73.
[3] 王宇,吴炜鑫,王兴伟. "互联网+"下高校信息化建设模式的探索与研究[J]. 计算机应用与软件,2016,33(11):41-45.
[4] 李林林. 关于高校信息标准建设的若干思考[J]. 黑龙江教育高教研究与评估,2017(5):79-80.
[5] 张贤伟,陶祥亚,贾长云. 高校数字化校园建设中的信息标准探析[J]. 软件导刊,2014(8):62-64.

The Research and Implementation on Information Standard of Internet Plus Intelligent Campus

MAO Xiaohong[1], WU Zhiyi[2]
(1. Network and Education Technology Center, Guangdong Polytechnic of Science and Technology, Zhuhai Guangdong 519090, China;
2. Asset Management Department, Guangdong Polytechnic of Science and Technology, Zhuhai Guangdong 519090, China)

Abstract: Research on information standards, which serve as data dictionaries for information exchange and resources sharing, is the foundation of Internet plus intelligent campus, as well as an essential component to achieve efficiency in data sharing. The detailed solutions for development of information standards are raised and the architecture and disaggregated model of the standards are illustrated by hands-on research on information standards of colleges. The result shows that the information standard is very practical for construction of intelligent campus.
Key words: information standards; information classification; coding rules

(责任编辑: 冉小晓)

(上接第 469 页)

The Study on the Automatic Detection of Multi-Software Refactoring Based on Version

ZHONG Linhui, HUANG Xiaoming, XUE Liangbo, YE Haitao
(College of Computer and Information Engineering, Jiangxi Normal University, Nanchang Jiangxi 330022, China)

Abstract: Automatic detection of software refactoring is one of hot topics in the field of software refactoring. The current technology of automatic detection of software refactoring can detect multi-software refactoring happened at different locations in different software versions and is short of the ability to detect it at the same location in different software versions. In the paper, an automatic detection technology based on call graph is proposed at the level of method, which can automatically detect themulti-software refactoring including "extract method" and "rename method" in the same function in different software versions. Also, the effectiveness of the proposed technology is realized by the experiments.
Key words: software refactoring; software version; extract method; rename method

(责任编辑: 冉小晓)