计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2018,27(9):188−192 [doi: 10.15888/j.cnki.csa.006529]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

代码变更中抽取类重构模式的识别①

孙美荣, 杨春花

(齐鲁工业大学 (山东省科学院) 信息学院, 济南 250353)
通讯作者: 孙美荣, E-mail: smr666999@163.com

摘　要: 在现代软件开发和维护中, 重构是提高软件可维护性和软件质量的常用手段. 而大量重构模式掺杂在日常的 bug 修复、功能增加等代码变更中, 使得变更理解变得非常复杂. 识别重构模式可以将重构与其它类型的代码变更隔离, 利于变更理解. 目前在识别重构模式的相关研究中, 并没有结合变更类型和相似性比较的识别重构模式的方法及工具. 为此, 提出了一种基于细粒度变更类型和文本相似性比较识别重构模式的方法. 将该方法应用于抽取类重构模式, 并在 4 个开源项目中进行了实验, 其平均准确率在 82.6% 左右.

关键词: 重构模式; 抽取类; 细粒度; 相似性比较

引用格式: 孙美荣,杨春花.代码变更中抽取类重构模式的识别.计算机系统应用,2018,27(9):188–192. http://www.c-s-a.org.cn/1003-3254/6529.html

Identifying Extract Class Refactoring Patterns from Code Changes

SUN Mei-Rong, YANG Chun-Hua

(School of Information, Qilu University of Technology (Shandong Academy of Sciences), Jinan 250353, China)

Abstract: Refactoring is a common way of improving software maintainability and software quality in modern software development and maintenance. In daily revision, refactoring patterns usually mix with code changes accomplishing other tasks such as bug fixing and feature addition, which makes the change understanding very complicated. It facilitates changes understanding to identify refactoring patterns that can isolate refactoring from other types of code changes. As far as I am concerned, there is no method and tool which have been proposed to identify code refactoring by code change types and similarity comparison. We proposed An identifying algorithm for refactoring patterns, based on fine-grained changes type and text similarity. The algorithm is used for refactoring patterns of extract class. The algorithm has been tested on 4 open source projects, with an average 82.6% accuracy.

Key words: refactoring patterns; extract class; fine-grained; similarity comparison

重构[1,2]是一种有纪律、经过训练、有条不紊的程序整理方法, 是现代软件开发和维护中用于提高软件可维护性和软件质量的常用手段, 在代码整理过程中可以将不小心引入的错误降低.

现代的软件开发一般基于版本管理系统进行, 软件工程师为了维护系统或提高系统的性能, 每天会提交大量的代码. 文献[3]指出代码变更伴随着软件系统的整个生命周期, 不断的代码变更, 会使软件的复杂度大幅度的提高[4]. 代码变更是指日常的 bug 修复、代码重构、功能增加, 这使得代码评审者[5]和软件工程师在理解代码时不得不人工对代码进行探查, 以区分哪些变更的代码是重构, 哪些不是.

而且日志描述往往反映不了代码变更的真正行为, 廖湘科等人在《大规模软件系统日志研究综述》[6]中, 通

① 基金项目: 国家自然科学基金 (61502259)
Foundation item: National Natural Science Foundation of China (61502259)
收稿时间: 2018-01-21; 采用时间: 2018-02-27; csa 在线出版时间: 2018-08-16

过对软件 (Apache, Squid, PostgreSQL, SVN 以及 Coreutils 等) 系统的失效报告进行随即检测, 发现 77% 的系统失效都是常见的错误诊断, 而 57% 的错误是没有进行日志记录.

文献[7] 对微软 54 位资深开发人员进行的调研, 发现 96% 的人认为日志在软件开发和维护中有重要作用, 认为日志是了解变更代码的主要信息来源, 然而业界人员对日志的重视度不高, 且代码变更书写不规范, 导致评审员和维护人员花费大量的时间去定位及分析这些代码变更. 因此, 为了使得变更的代码易于理解, 及提高代码质量, 将重构模式从代码变更中隔离出来是非常必要的.

重构模式的识别是在变更后的代码中寻找符合特定重构模式的代码修改, 是重构的反过程. 刘阳等人[8] 提出了一种重构检测算法, 是基于版本元素匹配原理, 对函数抽取重构进行了识别, 但是并没有涉及其它类型的重构模式.

本文通过对四个开源项目的变更代码进行探查, 抽取类是最为常见的重构模式. 因此, 本文对抽取类重构模式进行了研究, 提出了识别的算法.

1 识别方法

1.1 抽取类模式示例

抽取类 (Extract Class)[9]重构模式, 一般用于处理过长的类. 一个类如果包含过多的功能及属性, 会导致这个类过于臃肿. 为了提高类的高内聚, 低耦合, 就会将一些不必要的或不经常用的方法提炼到另一个类中, 来为这个类服务. 如图 1 是一个用类图形式表示的抽取类模式示例. 类 Person 中过多的属性 oficeAreaCode 和 officeNumber 以及功能代码 getTelephoneNumber() 被抽取到了一个新类 TelephoneNumber 中, 且在移动代码的地方对新增加类方法的引用.

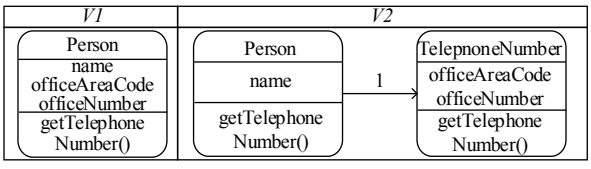

图 1 抽取类模式示例

1.2 识别方法

根据上述抽取类模式的例子, 不难发现, 抽取类模式变更具备如下 3 个特性:

① 变更文件中的某些属性及方法被删除;

② 文件中删除的属性和方法移动到其它类文件中;

③ 在原文件删除代码的位置有对被移动方法的引用.

其中, ①和③的判定需要基于变更代码块的语法信息进行识别, 因此我们借用了 ChangeDistiller[1] 工具获得一个变更文件中所有的代码变更, 如声明对象: ADDITIONAL_OBJECT_STATE、插入语句: STATEMENT_INSERT、删除语句: STATEMENT_DELETE、移除属性: REMOVED_OBJECT_STATE. 它是 BeatFluri 等人[10,11]编写的一个 Tree differ 算法, 将对变更前后抽象语法树进行对比, 获取分类变更. 它可以区别多种方法类型的变化或类等级上的变化.

而特性②的判定需要基于文本的相似性进行识别. 我们借助了 Levevshtein[2] 算法, 即文本相似性判断. Levevshtein 是一种计算两个字符串间差异程度的字符串度量 (string metric) 算法, 即一个单词变成另一个单词要求的最少单个字符编辑数量 (如: 删除、插入和替换). 那么两个字符串的相似度算法:

$$Similarity = (Max(x, y) - Levenshtein)/Max(x, y)$$

其中, x 和 y 为源串和目标串的长度, 本文 x 指删除的代码语句, y 指在新类文件中的代码行. 注意: 本文研究的是重构模式识别, 不是字符串的相似性, 所以并没有对该算法进行改进.

识别方法的具体流程如下:

1) 利用 ChangeDistiller 获取两个相邻文件的所有变更类型的详细信息如: 变更类型、变更内容、变更内容所属的双亲;

2) 根据每条变更所属双亲实体进行分组, 将相同双亲实体分到一个组内, 并对每个组内的所有删除的语句和增加的语句按行号进行排序, 其中代码行号通过读取文本行获得;

3) 依据重构模式的特性从 2) 所得分组中, 借用 Levevshtein 算法进行相似度匹配 (删除语句和新增文件的代码行), 查找相应的元素, 具体过程见下节.

1.3 抽取类模式识别

图 2 显示了该算法的框架, 具体流程如下文.

1) 代码变更抽取

通过 ChangeDistiller 获取所有的代码变更, 包括

[1]https://bitbucket.org/sealuzh/tools-changedistiller/src/
[2]https://en.wikipedia.org/wiki/Levenshtein_di stance.

声明对象、删除的语句、原方法中新增的语句等. 一个代码变更由变更类型 (ChangeType)、变更实体 (ChangeEntiy) 和变更双亲实体 (ParentEntity) 构成.

2) 代码变更分组

通过上述步骤 1), 每条变更可获得一个元组的集合 C={<*type*, *enity*, *content*,*parenten-tity* >| *type* 是变更类型, *enity* 是变更实体, *content* 是变更内容, 以及所属双亲实体 *parententity*}. 对每条代码变更根据所属双亲实体进行分组 CG, 其中每个变更分组 cg=< *parententity*, *CS*>, *CS* 指具有相同双亲的变更集合.

3) 代码块抽取判定步骤,

将每组 *CS* 分为两部分, 令删除部分为 l_{delete} 和增加部分为 l_{add}, 即 CS=$cg.\ l_{\text{delete}}$+ $cg.l_{\text{add}}$.

① 若父类实体是类等级上的变更. 判断 $cg.l_{\text{add}}$ 是否为新增类文件的声明对象, 若是则, 再判断 $cg.\ l_{\text{delete}}$ 是否为该新增类文件的属性;

② 若父类实体是方法变更. $cg.l_{\text{add}}$ 是否为新增类文件的方法, 若是则, 再判断 $cg.\ l_{\text{delete}}$ 是否为该方法的方法体;

①、②的过程我们借用了文本相似性工具 Levevshtein 对相应元素进行判定, 若以上步骤成立, 则成功识别一个抽取类重构模式.

方法扩展: 新文件 f_{new} 获取及相关操作, 1) 新增文件 f_{new} 是根据本次提交的 revision_id 与上次 revision_id–1 进行比较获得, 若 revision_id–1 中的文件集不包含文件 *f*, 则认为文件 *f* 为新增文件 f_{new}; 2) 通过 Java 的反射机制获取新增类文件的属性 f_{new}.*field*、方法 f_{new}.*method*、类名 f_{new}.*class*.

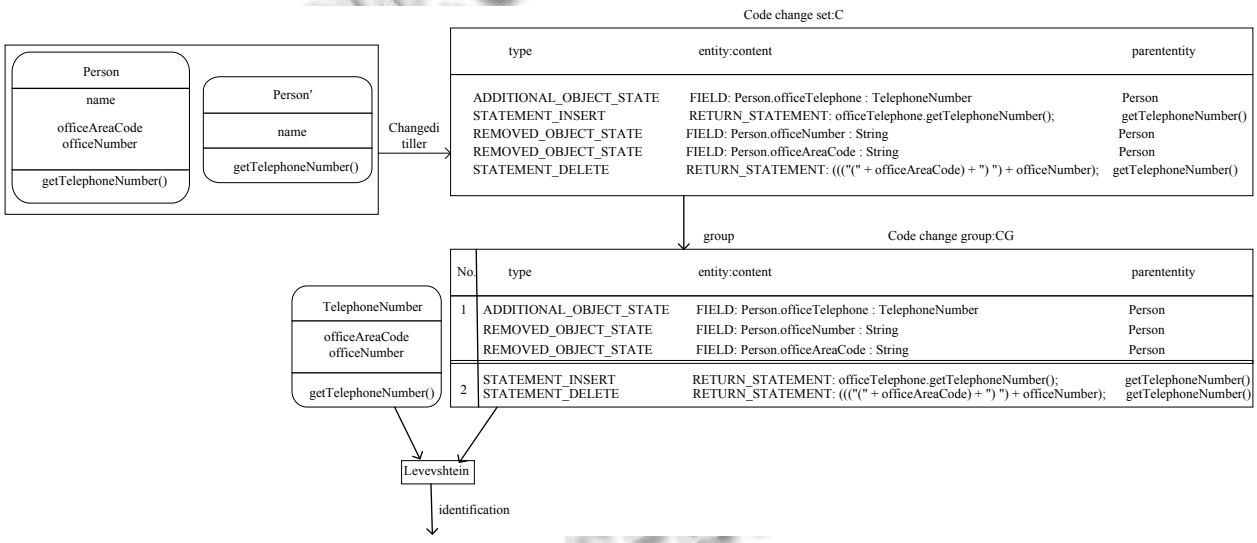

图 2 抽取类思想

2 算法

编写抽取类重构模式识别的伪代码算法.

1 输入: 一次提交的版本号 revision_id
2 输出: 重构模式集合 *P*
3 获取 revision_id 中所有的代码文件集 *F*
4 for each $f \in F$ do
5 获取 *f* 的前相邻版本 f_{old}
6 依据 f_{old} 和 *f*, 获取所有的变更集合 *C*
7 对集合 *C* 进行分组得到集合 *CG*
8 获取所有属性的变更组集合 $CG_{\text{field}} \subseteq CG$
9 获取所有方法的变更组集合 $CG_{\text{method}} \subseteq CG$
10 for each $cg \subseteq CG$ do
11 for each $f_{\text{new}} \in F$ do
12 if $cg \subseteq CG_{\text{field}}$
13 for each $cg \subseteq CG_{\text{field}}$
14 获取 $CG_{\text{field}}.CS$ 中所有的新增语句 $cg.\ CS.l_{\text{add}}$
15 获取 $CG_{\text{field}}.CS$ 中所有的删除语句 $cg.\ CS.l_{\text{delete}}$
16 if $cg.\ CS.l_{\text{add}}.entity$="FIELD"
17 ∩Levevshtein ($cg.\ CS.l_{\text{add}}.\ parententity, f_{\text{new}}.class$)
18 if $cg.\ CS.l_{\text{delete}}.entity$="FIELD"
19 ∩Levevshtein ($cg.\ CS.l_{\text{delete}}.\ conten\ t, f_{\text{new}}.field$)
20 end if
21 end if
22 end for
23 else if $cg \subseteq CG_{\text{method}}$
24 for each $cg \subseteq CG_{\text{method}}$ do
25 获取 $CG_{\text{method}}.CS$ 中所有的新增语句 $cg.\ CS.l_{\text{add}}$
26 获取 $CG_{\text{method}}.CS$ 中所有的删除语句 $cg.\ CS.l_{\text{delete}}$

27 if *cg*. *CS*.l_{add}.*entity*="RERURN_STATEMENT"
28 ∩Levevshtein (*cg*. *CS*.l_{add}. *parententity*, f_{new}.*method*)
29 if *cg*. *CS*.l_{delete}.*entity*="RERURN_STATEMENT"
30 ∩Levevshtein (*cg*. *CS*.l_{delete}. *content*, f_{new})
31 end if
32 end if
33 end for
34 end if
35 end if
36 成功识别一个抽取类模式 p
37 end for
38 end for
39 end for
40 $P=P\cup\{p\}$

算法伪代码中第 4 行指遍历集合 F 中的所有文件 *f*; 第 5 行通过 ChangeDistiller 可知 *f* 是否为变更文件; 第 8 行指获得集合 CG 中的属性集合; 第 9 行指获取集合 CG 中的方法集合; 16~19 指在变更类型为“FIELD”的情况下, 判断属性是否移动到 f_{new} 中; 27~30 指在变更类型为“RERURN_STATEMENT”情况下, 判断删除的方法及方法体是否移动到新文件 f_{new} 中, 且在删除代码的位置有对该方法的引用; 第 36 行若以上步骤为真, 则成功识别一个抽取类模式; 第 40 行返回所有识别成功的重构模式集.

3 实验验证

3.1 数据源

我们通过 minigit[3] 工具获取了四个开源项目 jEdit[4], eclipse JDT Core[5], Apache maven[6], and google-guice[7], 在某段时间的变更历史, 该工具将该时间段内所有的变更信息 (包括所有提交的版本、每个版本的日志、源文件等) 抽取到了 MySQL 数据库中.

我们前期通过人工分析变更日志和源代码探查, 判定了一些存在重构模式的版本. 项目的信息及人工判定的重构版本数据见表 1.

表 1 开源项目详细信息表

项目名称	时间段	版本数	抽取类版本数
jEdit	1998-09-27~2012-08-08	6486	24
maven	2003-09-01~2014-01-29	9723	69
goole_guice	2006-08-22~2013-12-11	1198	22
eclipse	2001-06-05~2013-10-16	19 321	6

[3] https://github.com/SoftwareIntrospectionLab/MininGit
[4] https://github.com/linzhp/jEdit-Clone
[5] https://github.com/eclipse/eclipse.jdt.core
[6] https://github.com/apache/maven/
[7] https://github.com/google/guice/

通过人工检测 4 个开源项目, 获得抽取类重构模式的数目分别为: jEdit 中含有 24 个版本, maven 中含有 69 个版本, goole_guice 中含有 22 个版本, eclipse 中含有 6 个版本.

3.2 结果及实验分析

图 3、4 是取自项目 maven 版本 68ca923 中 DefaultMetadataResolutionReques.java(left.java) 和 DefaultRepositoryRequest.java(right.java) 的部分变更内容; 其中减号表示删除代码行, 加号表示增加代码行. 图 5 是该算法对这一变更检测的信息输出. 其中 ChangeDistiller 获取声明对象 FIELD、删除代码, 以及文本读取代码块行号.

该实验对表 1 中的数据进行验证, 通过实验后检测得到的实验结果, 见表 2.

```
41      -  private ArtifactRepository localRepository  ;
42      -
43      -  private List<ArtifactRepository > remoteRepositories ;
44
45      -  private RepositoryCache cache  ;
46      -
47  41     private boolean resolveManagedVersions  ;
48  42
49      -  private boolean offline ;
    43  +  private RepositoryReqest repositoryRequest  ;
50  44
51  45     public DefaultMetadataResolutionRequest  ()
52  46
53      -   {//does nothing
    47  +  repositoryRequest =new DefaultRepositoryRequest  ();
54  48     }
55  49
56      -  public DefaultRepositoryRequest  ( RepositoryRequest request  )
    50  +  public DefaultRepositoryRequest  ( RepositoryRequest repositoryRequest  )
57  51     {
58      -  setLocalRepository ( request.getLocalRepository () );
59      -  setRemoteRepositories ( request.getRemoteRepositories () )
60      -  setCache( repositoryRequest .getCache() );
61      -  setOffline( repositoryRequest .isOffline() );
    52  +  this.repositoryRequest =new DefaultRepositoryRequest  (repositoryRequest );
    53     }
```

图 3 left.java

```
38  +  private boolean offline;
39  +  private ArtifactRepository localRepository;
40  +  private List<ArtifactRepository> remoteRepositories;
41  +  private RepositoryCache cache;
42  +  /**
43  +   * Creates an empty repository request.
44  +   */
45  +  public DefaultRepositoryRequest()
46  +  {
47  +     // enables no-arg constructor
48  +     }
49  +
50  +  /**
51  +  * Creates a shallow copy of the specified repository request.
52  +   *
53  +   * @param repositoryRequest The repository request to copy from, must not be {@code null}.
54  +   */
55  +  public DefaultRepositoryRequest( RepositoryRequest repositoryRequest )
56  +  {
57  +     setLocalRepository( repositoryRequest.getLocalRepository() );
58  +     setRemoteRepositories( repositoryRequest.getRemoteRepositories() );
59  +     setOffline( repositoryRequest.isOffline() );
60  +     setCache( repositoryRequest.getCache() );
61  +  }
```

图 4 right.java

通过表 2 可以得出, 该试验进行的抽取类模式识别, 其平均准确率为 82.6%, 准确率在 77.3%~91.6% 之

间略有波动. 准确率没有达到 100% 的原因与借用的代码相似性比较算法有关. 因为被测文本是重构代码变更文本, 所以被提取的代码块并不是简单的复制/粘贴. 程序员在进行代码重构时, 为了提高代码质量的可理解性、可维护性和可扩展性, 被提取的代码行中的有些元素可能被替换, 如新增方法中的参数、局部变量等, 但整个操作并不会改变软件功能. 因此, 对本文反过程重构模式识别中代码块提取过程会有些影响.

```
revision_id：41825
FIELD: DefaultMetadataResolutionRequest.repositoryRequest : RepositoryRequest
Key = 41, Value = private ArtifactRepository localRepository;
Key = 43, Value = private List<ArtifactRepository> remoteRepositories;
Key = 45, Value = private RepositoryCache cache;
Key = 58, Value = setLocalRepository( request.getLocalRepository() );
Key = 59, Value = setRemoteRepositories( request.getRemoteRepositories() );
Key = 60, Value = setCache( request.getCache() );
Key = 61, Value = setOffline( request.isOffline() );
commit_id file_id 变更前文件：DefaultMetadataResolutionRequest.java
41825 42923
commit_id file_id 抽取代码块文件：DefaultRepositoryRequest.java
41825 42925
```

图 5　实验结果

表 2　实验结果

项目	人工检测抽取类数	实验检抽取类测数	准确率 (%)
jEdit	24	22	91.6
maven	69	54	78.2
goole_guice	22	17	77.3
eclipse	6	5	83.3

4　结束语

我们提出了一种基于变更类型和文本相似性比较的重构模式识别方法, 并设计和实现了对抽取类 (Extract Class) 模式的识别. 通过实验验证, 该方法可以比较准确地识别 Extract Class 模式.

除了 Extract Class 模式, 该方法还适用于其它存在代码移动的重构模式, 如 Extract Superclass, Move Interface 等. 后续工作包括将该方法应用于这些模式的识别.

参考文献

1 Murphy-Hill E, Parnin C, Black AP. How we refactor, and how we know it. IEEE Transactions on Software Engineering, 2012, 38(1): 5–18. [doi: 10.1109/TSE.2011.41]

2 Fowler M. Refactoring: Improving the Design of Existing Code. Addison-Wesley Long-man Publishing Company Inc., 1999: 256.

3 原子, 于莉莉, 刘超. 面向细粒度源代码变更的缺陷预测方法. 软件学报, 2014, 25(11): 2499–2517.

4 冯欣. 代码质量控制与复杂度测量在大型软件项目中的研究及应用[硕士学位论文]. 长春: 东北师范大学, 2006.

5 井涛. 代码审查在软件工程实施中的重要性. 电子技术与软件工程, 2017, (21): 43–44.

6 廖湘科, 李姗姗, 董威, 等. 大规模软件系统日志研究综述. 软件学报, 2016, 27(8): 1934–1947. [doi: 10.13328/j.cnki.jos.004936]

7 Fu Q, Zhu J, Hu W, *et al*. Where do developers log? An empirical study on logging practices in industry. International Conference on Software Engineering. 2014. 24–33.

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

9 Fokaefs M, Tsantalis N, Stroulia E, *et al*. Identification and application of extract class refactorings in object-oriented systems. Journal of Systems & Software, 2012, 85(10): 2241–2260.

10 Fluri B, Gall HC. Classifying change types for qualifying change couplings. 2006, 1: 35–45.

11 Fluri B, Wuersch M, Pinzger M, *et al*. Change distilling: Tree differencing for fine-grained source code change extraction. IEEE Transactions on Software Engineering, 2007, 33(11): 725–743. [doi: 10.1109/TSE.2007.70731]