基于相似性度量的面向对象程序方法级克隆侦测

于冬琦[1,2]，吴毅坚[1,2+]，彭 鑫[1,2]，赵文耘[1,2]

[1]上海市智能信息处理重点实验室

[2]复旦大学计算机科学技术学院，上海 200433

+通讯作者 Email: wuyijian@fudan.edu.cn

摘 要： 代码克隆侦测对于代码重构以及可复用资产抽取都有着重要的作用.现有的克隆侦测方法及工具以相似代码片段为单位，给进一步的克隆分析以及代码重构带来困难.针对这一问题，本文提出了一种基于相似性度量的面向对象程序方法级克隆侦测方法，即以方法为单位进行克隆代码侦测.该方法综合利用代码中的注释、签名以及语法相似性来度量方法代码之间的克隆程度.在此基础上合并子类中的相似方法并提取到父类中，从而实现进一步的代码重构.本文通过对 JDK 包中代码的实验分析验证了本文所提出方法的有效性.初步的实验结果表明，本文方法能够准确、有效地辅助开发者实现方法级的克隆代码侦测.

关键字： 面向对象；代码克隆；克隆侦测；逆向工程；重构

中图分类号： TP311.5 **文献标识码：** A **文章编号：** 0372-2112

Object-Oriented Program Method Level Clone Detection Based on Similarity Measurement

YU Dong-qi[1,2], WU Yi-jian[1,2+], PENG Xin[1,2], ZHAO Wen-yun[1,2]

[1]Shanghai Key Laboratory of Intelligent Information Processing

[2]School of Computer Science, Fudan University, Shanghai 200433, China

+Corresponding Author Email: wuyijian@fudan.edu.cn

Abstract: Code clone detection is important for refactoring and extraction of reusable assets. The present code clone detection method is on the basis of detection of similar code segments, which brings difficulty for further clone analysis and refactoring. Aiming at this problem, this paper offers a method level clone detection method for object-oriented program based on similarity measurement, which detects method level cloned code. Our method utilizes comments of the code, method signature and syntactic similarity to measure the degree of the clone. Based on the method mentioned above, further refactoring can be realized by combining similar methods of sub-classes into one method and pulling it up into super-class. Our method's effectiveness has been verified by analyzing the result of the experiment on the code of JDK. The initial experiment result shows that our method can help developer detect method level code clone both accurately and effectively.

Key words: object-oriented; code clone; clone detection; reverse engineering; refactoring

1 引言

现有的克隆代码侦测技术往往以代码片段为单位，从而导致侦测出的克隆代码片段往往是零散的，甚至跨越了方法边界以及类边界等语法单元.这就为进一步的克隆分析和代码重构带来了很大的问题.为了弥补这个不足，本文以典型的 Java 程序为例，提出了一种基于相似性度量的面向对象程序方法级克隆侦测技术，我们以方法为单位进行克隆侦测，就能为后续的重构提供很大便利.我们首先计算方法注释和签名的整体相似度.然后，我们只对注释和签名整体相似度超过某个阈值的方法对进行方法体内部的语法分析和对比，从而得出方法的整体相似度.然后，我们只把整体相似度超过某个阈值的方法对视为克隆的方法.最后就可以利用现有的技术对这些子类中的方法进行合并，并把合并后的方法提升到它们公共的父类中，从而实现重构.

在上述方法的基础上，我们针对 Java 代码开发了方法级克隆代码侦测工具原型，并针对 JDK1.5[1]中的克隆代码进行了自动重构实验.初步的实验结果表明，本文方法能够准确、有效地辅助开发者实现方法级的克隆代码侦测.利用方法对的注释和签名整体相似度进行筛选，可以大大减少需要进行语法分析和对比的方法对数量.

本文剩余部分将首先对克隆代码侦测的一些相

收稿日期：YYYY-MM-DD；修回日期：YYYY-MM-DD

基金项目：国家 863 高技术研究发展计划(No. 2007AA01Z125, No. 2009AA010307)；国家自然科学基金(No. 60703092, No. 60903013)；上海市科学技术委员会项目(No. 08DZ2271800, No. 09DZ2272800)

关工作进行探讨.接着，第 3 部分概述基于相似性度量的方法级克隆侦测的主要过程，并在第 4 部分着重对于度量方法注释的相似度、度量方法签名的相似度和度量方法的整体相似度等关键技术进行介绍.第 5 部分将介绍我们基于 Java 的克隆侦测工具原型，以及对 JDK1.5 中的克隆代码进行的实验.第 6 部分结合实验结果，对本文方法的优缺点以及下一步的改进方向进行了探讨.最后，第 7 部分对全文进行了总结.

2 相关工作

在代码克隆侦测方面，已经有人提出了大量的方法，主要有基于标号（token-based）的方法，基于语法树（tree-based）的方法以及基于程序依赖图的方法（PDG-based）等.

基于标号的方法以 Toshihiro Kamiya 等[2]为代表，他们提出的侦测方法包括了将源文件进行一系列转换并进行 token-by-token 的比较.他们对此方法进行了优化并开发了一个叫做 CCFinder 的工具，该工具可以侦测 C，C++，Java 的源代码中的克隆.他们同时提出了度量克隆的一些手段.他们还在 JDK 等软件系统上面进行了一系列实验，定性与定量的评价了该方法的有效性.

基于抽象语法树的克隆侦测方法以 Ira D.Baxter 等[3]为代表，他们提出了一种简单实用的克隆侦测方法.他们开发了一个工具进行克隆侦测，该工具还能够为每处侦测到的克隆产生一个宏，并且用宏调用来替换原来的克隆代码.

基于程序依赖图（PDG）的方法[4][5][6]在获取源代码的高级抽象表示方面比其他的方法更进了一步，它考虑了源代码的语义信息.得到源程序的一系列程序依赖图之后，就可以利用同构子图匹配算法来寻找相似的子图，并作为克隆返回.

相比上述的克隆侦测方法，本文提出的克隆侦测方法是面向重构的，它着眼于利用面向对象程序中方法的注释和签名中包含的文本信息以及方法体中包含的语法信息进行方法级的克隆侦测.并将侦测出的方法对视为适合进行重构的候选.

3 方法概述

本文讨论如何侦测出面向对象程序中若干个拥有共同父类的子类中的方法级克隆，其目的是将侦测出的克隆方法利用现有的技术进行重构，将重构的结果提升到它们公共的父类中，从而减少代码克隆.在对给定的某几个子类进行克隆侦测的过程中，往往由于子类中的方法可能数量众多，如果直接对每个方法体进行语法分析和对比会耗费大量的计算时间.而流行的程序设计实践要求程序员对类中方法的功能和实现方式以注释的形式加以说明，并且要求方法命名以及参数命名中所使用的单词和缩写要具有一定含义.于是，我们首先定义并计算方法注释和签名的整体相似度，并将其作为筛选的依据来决定是否继续对方法体进行语法分析和对比，进而计算出两个方法的整体相似度.最后我们对方法整体相似度超过某阈值的方法对实施重构.

本文方法的流程图如图 1 所示：

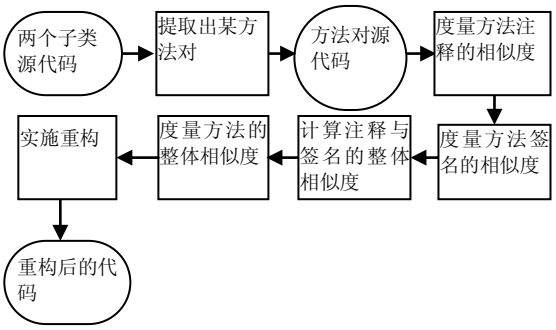

图 1 方法概览

Java 语言是一种广泛使用的面向对象程序设计语言，本文就以 Java 语言为例，并以侦测两个子类中的克隆方法为例，来描述我们的方法.

4 基于相似性度量的方法级克隆侦测

4.1 度量方法注释的相似度

一个大家所公认的注释的办法就是把某方法的注释放在这个方法签名的前面，本文假定大多数方法都是注释良好的，并认为两个相似的方法的注释也是相似的.我们忽略方法体中对于个别语句的注释.下面就给出两段注释相似度的定义：

定义 1 两段注释的相似度：设 S1 和 S2 分别为两段注释 c1 和 c2 中所使用的单词的集合，则 c1 和 c2 的相似度

$$\text{similarComment}=\begin{cases}\dfrac{|S1\cap S2|}{|S1\cup S2|}, |S1|\neq 0 \text{且} |S2|\neq 0\\ -1, |S1|=0 \text{或} |S2|=0\end{cases}$$

在求单词集合交集和并集的时候，我们一律不考虑单词中字母的大小写的区别.

例如给定如图 2 所示两段程序的注释，则可给出下面的注释相似度计算.

```
方法 1：
//show me on the screen on p
void show(Screen s, Position p)
{...}
```

```
方法 2：
//display me on the screen on p
void display(Screen s, Position p)
{...}
```

图 2 两个方法的注释

S1={‘show’,‘me’,‘on’,‘the’,‘screen’,‘p’}，S2={‘display’, ‘me’,‘on’,‘the’,‘screen’,‘p’}， S1∩S2 ={‘me’,‘on’,‘the’, ‘screen’,‘p’}，S1∪S2={‘show’,‘display’,‘me’,‘on’,‘the’, ‘screen’,‘p’}，因此 similarComment=5/7≈0.71.

在上述定义中，首先生成两段注释的单词集合，然后计算的是两段注释所包含的单词种类的相似程度.我们没有重复计算那些出现次数超过一次的单词，因为注释中有意义的单词往往是个别的名词和动词等，而英语中大量重复的其他虚词，诸如例子中出现了两次的“on”，则对于程序分析没有意义.

图 3 给出了计算两段注释相似度 similarComment 的伪代码.

```
double computeSimilarComment(String c1, String c2)
{
  String word;
  S1=Φ; S2=Φ;
  while(c1 中还有未被处理的字符)
  {
    word=下一个被空格分隔的字符串;
    if(word 为标点符号)
      continue;
    else
      {将 word 放入 S1 中;}
  }
  while(c2 中还有未被处理的字符)
  {
    word=下一个被空格分隔的字符串;
    if(word 为标点符号)
    continue;
    else
    {将 word 放入 S2 中;}
  }
  sBottom=Φ; sTop=Φ;
  sBottom=S1∪S2;
  sTop=S1∩S2;
  if(sBottom 为空 || sTop 为空)
    return -1; //-1 为特殊标记
  return  sTop.count / sBottom.count;
}
```

图 3 计算两段注释相似度 similarComment 的伪代码

4.2 度量方法签名的相似度

在 Java 语言中，方法的签名主要由四部分组成，访问权限修饰词，返回值类型，方法名和参数列表.对于访问权限修饰词，比如 public、private 和 protected 等，由于它们的作用是控制方法的访问权限，而对于方法本身功能的表达没有任何帮助，所以本文暂且忽略访问权限修饰词.则方法签名相似度的计算就应该包括返回值类型，方法名和参数列表这三部分相似度的计算.

4.2.1 度量返回值类型的相似度

由于我们进行克隆侦测的目的是要将两个克隆的方法通过重构，合并为一个方法提升至其公共的父类中，而 Java 语言要求方法的返回值具有唯一的类型.那么对于那些具有不一致的返回类型的方法则不能视为克隆的.则有如下定义：

定义 2 返回值类型的相似度：设方法 m1 和 m2 的返回值类型分别为 r1、r2，则 m1 和 m2 的返回值类型相似度 $\text{similarReturn}=\begin{cases}1, if\ \text{r1}=\text{r2}\\0, if\ \text{r1}\neq\text{r2}\end{cases}$.

4.2.2 度量方法名的相似度

方法名和注释都是通过自然语言来描述方法的功能.但是方法名中的单词和单词之间往往是紧挨着的，没有空格分隔，或者用下划线‘_’分隔，而且在流行的程序设计实践中，方法名往往是第一个单词首字母小写，后面的单词首字母大写.这就为我们提取方法名中的单词提供了极大的便利.下面就定义两个方法名的相似度：

定义 3 两个方法名 n1 和 n2 相似度：设 S1 和 S2 分别为两个方法名 n1 和 n2 中所使用的单词的集合，则 n1 和 n2 的相似度定义为：

$$\text{similarMethodName}=\frac{|S1\cap S2|}{|S1\cup S2|}$$

同计算注释相似度类似，在计算两个方法名所用单词集合的并集和交集时，我们一律不考虑单词中字母大小写的区别.例如，对于给定的两个方法名 n1=‘fightTheEnemy’和 n2=‘beat_the_enemy’计算它们的相似度，则有：S1={‘fight’, ‘the’, ‘enemy’}，S2={‘beat’, ‘the’, ‘enemy’}，S1∩S2={‘the’, ‘enemy’}，S1∪S2={‘fight’, ‘the’, ‘enemy’, ‘beat’}，则 n1 和 n2 的相似度 similarMethodName=2/4=0.5.

与计算注释的相似度不同，由于方法签名总是由至少一个单词组成，所以我们可以保证定义中的 S1∪S2 永远不为空集.

```
double computeSimilarMethodName(String n1, String n2)
{
  String word;
  S1=Φ; S2=Φ;
  while(n1 中还有未被处理的字符)
  {
   word=下一个被首字母大写标识的单词;
   去除 word 中的’_’等分隔符;
   将 word 放入 S1 中;
  }
  while(n2 中还有未被处理的字符)
  {
   word=下一个被首字母大写标识的单词;
   去除 word 中的’_’等分隔符;
   将 word 放入 S2 中;
  }
  sBottom=Φ; sTop=Φ;
  sBottom=S1∪S2;
  sTop=S1∩S2;
  return  sTop.count / sBottom.count;
}
```

图 4 计算方法名相似度 similarMethodName 的伪代码

图 4 给出了计算两段方法名相似度 similarMethodName 的伪代码。

4.2.3 度量参数列表的相似度

在 Java 语言中，参数列表中的每一项由参数类型和参数名两部分构成，即：(DATA_TYPE1 PARAMETER_NAME1，DATA_TYPE2 PARAMETER_NAME2，…，DATA_TYPEn PARAMETER_NAMEn)的形式.其中的参数名和方法名的命名习惯相同.

本文对于两个参数列表的相似度的计算分为如下两个步骤：第一步，尽量为一个列表中的每一个参数在另外一个列表中寻找一个最佳匹配参数，也就是相似度最大的参数，其中又包括求参数类型的相似度和求参数名的相似度的过程.第二步，求得在两个参数列表中相似度超过某阈值的参数个数占总的参数个数的百分比，作为两个参数列表的相似度.我们先来定义两个参数的参数类型相似度：

定义 4 两个参数的参数类型相似度：若两个参数 p1 和 p2 的参数类型分别为 t1 和 t2，则 p1 和 p2 的参数类型相似度

$$\text{similarType} = \begin{cases} 1, if\ \text{t1} = \text{t2} \\ \text{typeFactor}(\text{t1}, \text{t2}), if\ \text{t1} \neq \text{t2} \end{cases},$$

其中 typeFactor(t1,t2)∈[0,1).

上面定义中的 typeFactor 函数的定义域为 Java 语言中参数类型组合的集合，表 1 就是一个 typeFactor 函数的例子：

表 1 一个 typeFactor 函数的例子

t1	t2	typeFactor(t1,t2)
int	long	0.5
int	short	0.5
long	short	0.2
float	double	0.6
Cat	Tiger	0.3
others	*others*	0

我们可以指定某些 Java 语言基本类型的相似度，比如图中的 int 和 long.我们也可以指定某些自定义类型的相似度，比如图中的 Cat 和 Tiger.未在 typeFactor 函数中明确定义相似度的其他类型间的相似度为 0.

下面我们定义参数名的相似度：

定义 5 两个参数的参数名的相似度：若两个参数 p1 和 p2 的参数名分别为 n1 和 n2，设 S1 和 S2 分别为 n1 和 n2 中所使用的单词的集合，则 n1 和 n2 的相似度 $\text{similarParameterName} = \frac{|S1 \cap S2|}{|S1 \cup S2|}$.

可见，两个参数名的相似度的定义和两个方法名的相似度的定义完全一致.

下面给出两个参数的相似度的定义：

定义 6 两个参数的相似度：两个参数 p1 和 p2 的相似度

similarParameter = similarType*similarParameterName.

例如，根据上述 typeFactor 函数的例子，两个参数‘Cat lazyBoy’和‘Tiger lazyGirl’的类型分别为‘Cat’和‘Tiger’，其类型相似度为 similarType=0.3，它们的参数名分别为‘lazyBoy’和‘lazyGirl’，其参数名相似度为 similarParameterName =1/3≈0.33，则这两个参数的相似度 similarParameter=0.3*0.33≈0.1.

下面我们定义一个参数在某个参数列表中的最佳匹配参数：

定义 7 一个参数在某个参数列表中的最佳匹配参数：设有参数 p 和参数列表 L(p1,p2,…,pn)，则 p 在 L 中的最佳匹配参数 pk = bestMatchParameter(p,L)，k∈[0,n]，其中 pk 满足

similarParameter(p, pk) = max{similarParameter(p,pi), i ∈[0,n]}.

可以看出一个参数在某个参数列表中的最佳匹配参数为参数相似度 similarParameter 最大的那个参数.接着，我们就有两个参数列表相似度的定义：

定义 8 两个参数列表的相似度：设两个参数列表 L1(p11,p12,…,p1n)和 L2(p21,p22,…,p2m)中分别有 n 个和 m 个参数，设 thresholdParameter∈[0,1]为参数相似度阈值，设集合 S 中保存的是一系列参数对(p1i,p2j)，对于集合 S 中的每个参数对(p1i,p2j)，都有 similarParameter(p1i,p2j)> thresholdParameter，且在所有满足上述条件的集合中，集合 S 的 Σ similarParameter(p1i,p2j)达到最大值，则 L1 和 L2 的相似度

$$\text{similarParameterList} = \begin{cases} \frac{|S| \bullet 2}{m+n}, m+n>0 \\ 1, m+n=0 \end{cases}.$$

在上述定义中，我们设置了一个参数相似度阈值 thresholdParameter，我们认为超过这个阈值的参数对才是足够相似的.如果两个参数列表 L1 和 L2 都为空，那么将它们的相似度设为 1，因为两个空的参数列表是能够完全匹配的.在上述定义中，某个参数 pij 在相应的参数列表中出现的顺序对于计算结果是没有影响的.因为方法签名所约定的传参数的顺序只是方法的调用者必须遵守的规范而已，这对于计算两个参数列表的相似性没有任何影响.

还可以注意到，上述定义中对于集合 S 中所保存的某个参数对(p1i,p2j)而言，可能有

bestMatchParameter(p1i,L2) ≠ p2j，也可能有bestMatchParameter(p2j,L1)≠p1i.也就是说对于某个参数对，它们二者可能并未互相构成最佳匹配，但是最终对于集合 S 而言，其中所包含的参数对的相似度之和却是最大的，我们认为这样的集合 S 能够比较客观地反映两个参数列表的相似程度.

下面我们以 L1:(Cat lasyBoy , int sleepHours) 和 L2:(Tiger lasyGirl , int napHours)为例来说明两个参数列表相似度的计算.我们设定参数相似度阈值 thresholdParameter=0.2，并利用表 1 中 typeFactor 函数的例子，可以求得表 2 中参数相似度结果.由此可以得出集合 S={('int sleepHours', 'int napHours')}， L1 和 L2 的相似度 similarParameterList (L1,L2)=1*2/4=0.5.

表 2 参数相似度结果

参数对(P1,P2)	相似度
('Cat lasyBoy','Tiger lasyGirl')	0.3*0.33≈0.1
('Cat lasyBoy','int napHours')	0*0=0
('int sleepHours','Tiger lasyGirl')	0*0=0
('int sleepHours', 'int napHours')	1*0.33=0.33

4.2.4 计算方法签名的相似度

定义 9 方法签名的相似度：设两个方法 m1 和 m2 的签名分别为 s1 和 s2，返回值类型分别为 r1 和 r2，方法名分别为 n1 和 n2，参数列表分别为 L1 和 L2，则 s1 和 s2 的相似度为：

similarMethodSignature = similarReturn(r1,r2) * (nameFactor * similarMethodName(n1,n2) + parameterListFactor * similarParameterList(L1,L2)) / (nameFactor + parameterListFactor)，其中 nameFactor，parameterListFactor 均大于 0.

在上述定义中，nameFactor，parameterListFactor 分别代表方法名的权重和参数列表的权重.

下面我们以 m1: public void sleepyCat(Cat lasyBoy , int sleepHours) 和 m2: public void sleepyTiger(Tiger lasyGirl , int napHours)为例来说明两个方法签名相似度的计算.由于这两个方法的方法名和参数名都由一些有意义的单词组成，所以我们设定方法名的权重 nameFactor=50，参数列表的权重 parameterListFactor=50.由于 m1 的返回值类型和 m2 的返回值类型均为 void，则它们的返回值类型相似度 similarType=1.对于方法名的相似度，我们得出 'sleepyCat' 和 'sleepyTiger' 的相似度 similarMethodName=1/3≈0.33.然后我们设定参数相似度阈值 thresholdParameter=0.2，并利用表 1 中 typeFactor 函数的例子，可以求得参数列表(Cat lasyBoy , int sleepHours) 和 (Tiger lasyGirl , int napHours)的相似度为 0.5.综合上述计算结果，可以得出 m1 和 m2 的方法签名的相似度 similarMethodSignature=1*(0.33*50+0.5*50)/(50+50)= 0.415.

4.3 度量两个方法的整体相似度

在定义两个方法的整体相似度之前，我们首先定义两个方法的注释和签名的整体相似度：

定义 10 两个方法的注释和签名的整体相似度：若有两个方法 m1 和 m2，设它们的注释分别为 c1 和 c2，它们的方法签名分别为 s1 和 s2，设它们的注释和签名的整体相似度为 similarCommentSignature，则有：

若 m1 和 m2 的注释相似度 similarComment(c1,c2) ≠-1，也就是 c1 和 c2 均不为空，则有：

similarCommentSignature=(factorComment*similarComment(c1,c2) + factorMethodSignature *similarMethodSignature(s1,s2)) / (factorComment + factorMethodSignature).

其中，factorComment>0 为方法注释的权重，factorMethodSignature>0 为方法签名的权重.

若 m1 和 m2 的注释相似度 similarComment(c1,c2)=-1，也就是 c1 和 c2 至少有一个为空串，则有：similarCommentSignature = similarMethodSignature(s1,s2).

接下来我们定义两个方法的整体相似度：

定义 11 两个方法的整体相似度：若有两个方法 m1 和 m2，设它们的方法体分别为 b1 和 b2，设它们的整体相似度为 similarMethod，则有：

① 若 similarCommentSignature(m1,m2) > thresholdCommentSignature 则 similarMethod=1-percentC(b1,b2).

② 若 similarCommentSignature(m1,m2) ≤ thresholdCommentSignature 则 similarMethod=0.

其中，thresholdCommentSignature∈[0,1]为方法注释和签名整体相似度阈值，percentC(b1,b2)为将 b1 和 b2 分别作为语句段进行语法分析对比而求得的语句差异百分比.

可以看出，只有对于那些注释与签名整体相似度大于阈值的方法对我们才去比较它们的方法体.我们使用我们之前的工作中所描述的语句段差异度分析方法[7]，求得两个方法体的差异百分比 percentC，也就是两个方法体中有差异的语句数量占总的语句数量的百分比.两个方法的整体相似度可以作为度量重构后能够消除多少克隆代码的指标.

利用方法签名和注释的整体相似度进行初步的筛选是有意义的，因为那些完全克隆的方法的注释和

签名理所应当是完全相同的.而在近似克隆的情况下，由于近似克隆的方法实现的是类似的功能，多数情况下只是将代码稍加变动，所以认为它们的方法注释和方法签名具有一定的相似性，是有一定道理的.

例如，对于图 5 中的两个方法：

```
方法 1:
/*the little cat plays with toy for some minutes*/
public void catPlay(Toy toy, int minutes)
{
  int i=0;
  for(i=0;i<minutes;i++)
  {
    playOneMinute(toy);
  }
}
方法 2:
/*the baby tiger plays with toy for some minutes*/
public void tigerPlay(Toy toy, int minutes)
{
  int i=0;
  for(i=0;i<minutes;i++)
  {
    playOneMinute(toy);
  }
  goToSleep();
}
```

图 5 两个 Java 方法

可以得出两个方法的注释相似度 similarComment==7/11≈0.64.我们设定方法名的权重 nameFactor=50，参数列表的权重 parameterListFactor=50，可以计算出它们方法签名的相似度 similarMethodSignature 为 0.67.由于注释相似度不等于-1，且观察到这两个方法的方法注释和签名都能够近乎相当地表达了方法的功能和实现概貌，我们将方法注释的权重 factorComment 设定为 50，方法签名的权重 factorMethodSignature 也设定为 50，这样，我们就可以计算出方法签名和注释的整体相似度为：(0.64*50+0.67*50)/100≈0.66.如果我们设定 thresholdCommentSignature=0.2 做为方法注释与签名整体相似度阈值，则这个方法对就能够通过筛选.最后我们对其方法体进行语法分析对比，可以计算出两个方法体的差异百分比 percentC 为 0.2，进而得出这两个方法的整体相似度为 1-0.2=0.8.

计算两个方法整体相似度的伪代码如图 6 所示.两个方法的整体相似度所表达的含义是重构后能够消除多少克隆代码，对于那些重构后能够消除的克隆代码数量很少的方法对，我们就没有必要将它们提升至父类中.所以我们要设定一个方法整体相似度阈值 thresholdMethod∈[0,1]，只有那些相似度超过这个阈值的方法对，我们才可以利用现有的某种重构技术，将其重构为一个方法并提升至公共的父类中.

```
double similarMethod (Method m1, Method m2) {
  if(similarComment(m1.comment,m2.comment)!=-1)
  {
    if( (factorComment*similarComment(m1.comment,
          m2.comment) + factorMethodSignature*
         similarMethodSignature(m1.signature,
         m2.signature)) /
           (factorComment+factorMethodSignature)
            > thresholdCommentSignature)
       {return 1-percentC(m1.block,m2.block);}
     else
        return 0;
   }
   else {
     if(similarMethodSignature(m1.signature,
     m2.signature) >thresholdCommentSignature)
      {return 1-percentC(m1.block,m2.block);}
     else
        return 0;
    }
}
```

图 6 计算两个方法整体相似度 similarMethod 的伪代码

5 工具实现及实验

5.1 原型工具

我们开发了一个原型工具，该工具接受两个 Java 类文件以及一个 typeFactor 函数定义文件作为输入，并可以设定各个相似度阈值以及权重.系统首先计算给定类文件中方法对的注释与签名整体相似度阈值，把那些超过阈值的方法对显示出来，作为通过第一轮筛选的候选方法对.紧接着，系统通过语法分析和对比，计算出它们的方法整体相似度 thresholdMethod，把超过阈值的方法对显示出来，作为通过第二轮筛选的候选方法对，这些方法对是可以进行重构的.如图 7 所示.

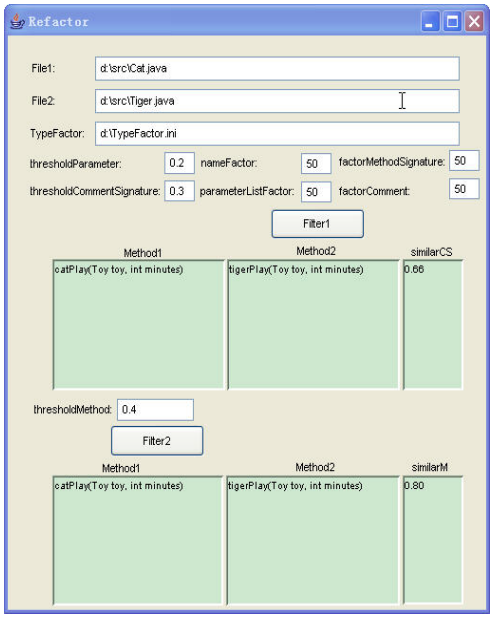

图 7 原型工具

5.2 实验：JDK1.5

Sun 公司提供的 Java 开发包 JDK1.5 中存在大量

的相似代码，我们选取了 JDK1.5 中大量的子类进行了实验，下面我们以一个实验结果示例加以说明.在这个示例中，我们选取的是 java.math 包中的 BigDecimal.java 和 BigInteger.java 这两个文件作为输入.BigDecimal 类用来表示任意精度的有符号浮点数，BigInteger 类用来表示任意精度的整数，且这两个类均继承自 Number 类.由于 JDK1.5 的源代码均为注释良好且方法名和参数名命名规范的代码，所以我们设定参数相似度阈值 thresholdParameter=0.2，并利用表 3 中所示的 typeFactor 函数来计算参数类型相似度，并设定方法名的权重 nameFactor=50，参数列表的权重 parameterListFactor=50，方法注释的权重 factorComment=50，方法签名的权重 factorMethodSignature=50，注释与签名整体相似度阈值 thresholdCommentSignature=0.3，方法整体相似度阈值 thresholdMethod=0.3.

表 3 typeFactor 函数

t1	t2	typeFactor(t1,t2)
int	long	0.5
int	short	0.5
long	short	0.4
float	double	0.6
BigInteger	BigDecimal	0.3
others	*others*	0

BigDecimal 类共有 28 个方法，BigInteger 类共有 44 个方法.经过第一轮筛选，共有 9 个方法对的注释与签名的整体相似度大于 0.3，如表 4 所示.可见，我们通过注释与签名的整体相似度的计算和筛选，大大减少了需要进行语法分析和对比的方法对的数量.最后，这 9 个方法对经过第二轮筛选，共有两个方法对的整体相似度大于 0.3，如表 5 所示.

表 4 第一轮筛选结果

方法 1（BigDecimal 中）	方法 2（BigInteger 中）	thresholdCommentSignature
public int compareTo(BigDecimal val)	public int compareTo(BigInteger val)	0.85
public boolean equals(Object x)	public boolean equals(Object x)	0.75
public int hashCode()	public int hashCode()	0.70
public int intValue()	public int intValue()	0.90
public long longValue()	public long longValue()	0.88
public int signum()	public int signum()	0.95
public String toString()	public String toString()	0.55
public double doubleValue()	public double doubleValue()	0.93
public float floatValue()	public float floatValue()	0.93

表 5 第二轮筛选结果

方法 1（BigDecimal 中）	方法 2（BigInteger 中）	thresholdMethod
public double doubleValue()	public double doubleValue()	0.67
public float floatValue()	public float floatValue()	0.67

经过仔细检查代码发现：Number 类中定义了 intValue()，longValue()，doubleValue()和 floatValue()这四个抽象方法，BigDecimal 类和 BigInteger 类均覆写了这四个方法；BigDecimal 类和 BigInteger 类中的 intValue()和 longValue()的实现方式均有很大差异，而 doubleValue()和 floatValue()的实现均只有少部分差异.我们推测上述两个类中 doubleValue()和 floatValue()中的代码克隆是由于代码中需要实现多态导致的.

6 讨论

6.1 各个权重的设定

首先，我们在计算方法签名的相似度的时候，需要人为设定两个权重：方法名权重 nameFactor 和参数列表权重 parameterListFactor，这两个权重的设定就决定了是方法名还是参数列表对于方法签名相似度的计算更具有显著意义.然后，我们在计算方法签名和注释的整体相似度的时候，人为设定了方法注释的权重 factorComment 和方法签名的权重 factorMethodSignature.这两个权重的设定就决定了是方法注释还是方法签名对于注释和签名整体相似度的计算更具有显著意义.显然，如果我们需要进行克隆侦测的子类中的方法的注释都是完备的，方法的命名以及各个参数的命名都使用了规范的办法，那么我们就可以将各个权重设定为近似相等.但是，如果子类中各个方法的注释和签名的命名情况参差不齐，们就很难以给定合适的权重.在这种情况下，我们只能依据大多数方法的注释和签名的情况给定一个大概的权重.

6.2 第一轮筛选的标准

本文将注释和签名的整体相似度作为第一轮筛选的标准，从而减少了需要进行方法体内部语法分析和对比的方法对数量.同时，这样做也带来了一个负面效应，那就是我们需要假设方法的注释和签名都是命名良好的，也就是说两个克隆的方法的注释是近似相同的，他们的方法名以及参数名也是近似相同的.如果某两个克隆方法的注释使用了不同的方式来描述方法的功能和实现，那么它们的注释中使用的单词集合有可能会有很大的不同.同理，如果某两个克隆的方法的方法名和参数名都使用了一些无意义的字母或者字母组合，那么两个克隆方法的方法名和参数名就不一定具有相似性.在这种情况下，我们计算的方法签名的相似度，其意义就值得怀疑.

6.3 第二轮筛选的标准

本文将那些注释和签名的整体相似度超过给定阈值的方法对进行语法分析和对比，从而求得方法的整体相似度.方法的整体相似度的含义是：将某两个方

法进行重构后，理论上能够消除多少克隆代码.我们将方法的整体相似度作为第二轮筛选的依据，这完全是站在尽可能地消除克隆代码的角度考虑.然而，重构是一个复杂的过程，在考察某两个方法是否有必要进行重构的时候，有许多因素需要考虑，而能够消除多少冗余的克隆代码只是其中的一个因素.

7 总结和展望

针对面向对象软件系统中普遍存在的代码克隆现象，本文提出了一种基于相似性度量的面向对象程序方法级克隆侦测技术，并可以将我们侦测的结果利用某种自动重构技术将分散在若干个子类中的克隆方法通过重构提升到其公共的父类中.本文的方法通过计算某两个方法注释的相似度，签名的相似度，以及注释和签名的整体相似度，避免了大量的方法体内部的语法分析和对比.然后，该方法通过计算经过第一轮筛选的方法对的方法整体相似度，进一步筛选出适合进行重构的方法对.在本文方法的基础上，我们实现了一个基于 Java 的方法级克隆代码侦测工具原型，并针对 JDK1.5 进行了克隆侦测实验.结果表明本文方法所提供的相似性计算技术能够准确地识别出适合进行重构的方法.

进一步的研究工作主要包括在各个权重的设定方面提供一些自动或者半自动的参考，以及将更多的用来判定某方法是否适合进行重构的判据纳入考虑范围.

参考文献：

[1] JDK1.5[CP/OL]. http://java.sun.com/javase/downloads/index.jsp,2008-03-01.

[2] Toshihiro Kamiya, Shinji Kusumoto, Katsuro Inoue. CCFinder:a multilinguistic token-based code clone detection system for large scale source code[J]. IEEE Transactions on Software Engineering, 2002, 28(7): 654-670.

[3] Ira D. Baxter, Andrew Yahin, Leonardo Moura, et al. Clone detection using abstract syntax trees[A].In:Proceedings of the International Conference on Software Maintenance[C]. Maryland, USA: IEEE CS, 1998. 368-377.

[4] Raghavan Komondoor, Susan Horwitz. Using slicing to identify duplication in source code[A].In:Proceedings of the 8th International Symposium on Static Analysis (LNCS 2126)[C]. Germany: Springer, 2001. 40-56.

[5] Jens Krinke. Identifying similar code with program dependence graphs[A]. In:Proceedings of the 8th Working Conference on Reverse Engineering[C]. Germany: ACM Press, 2001. 301-309.

[6] Chao Liu, Chen Chen, Jiawei Han, Philip S Yu. GPLAG: Detection of software plagiarism by program dependence graph analysis[A]. In: Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining[C]. Philadelphia, USA: ACM Press. 2006. 872-881.

[7] 于冬琦,彭鑫,赵文耘. 使用抽象语法树和静态分析的克隆代码自动重构方法[J].小型微型计算机系统, 2009, 30(9): 1752-1760.

作者简介：

于冬琦　男。硕士生。1984 年 12 月生于黑龙江省齐齐哈尔市。研究方向为软件再工程。

吴毅坚（通讯作者）　男。复旦大学计算机科学技术学院讲师，博士。1979 年 9 月生于上海。目前主要研究方向包括软件体系结构、软件复用和软件构件技术。Email: wuyijian@fudan.edu.cn

彭　鑫　男。复旦大学计算机科学技术学院讲师，博士，CCF 会员。1979 年 10 月生于湖北黄冈。目前主要研究方向包括软件产品线、软件构件与体系结构、软件维护与再工程。

赵文耘　男。复旦大学计算机科学技术学院教授、博导，CCF 高级会员。1964 年 9 月生于江苏常熟。目前主要研究方向包括软件复用、软件构件与体系结构。