一种基于平均测距的重构定位方法*)

刘建宾[1] 朱学军[1] 李满玲[2] 郭伟斌[3]

(汕头大学工学院计算机系 汕头515063)[1] (零陵学院计算机系 永州425006)[2]

(汕头大学医学院第一附属医院 汕头515063)[3]

摘 要 软件会随着时间变得越来越难以理解,越来越难以维护。因而,程序员必须在软件失去控制前重新构造它。重构是在保持程序外部可观察行为不变情况下,对软件的内部结构进行重新改造和组织的技术,是获得软件质量的一项关键技术。然而,重构在实际中并没有得到预期的应用,最主要的障碍是要耗费太多的时间和精力去寻找需要重构的位置以及选择适当的重构方法。为了克服这种障碍,本文提出一种基于平均测距的重构定位方法,给出类成员与类之间的平均距离及平均距离差公式及其在实际系统中应用的方法。文中阐述了距离理论,并通过一个简单的实例展示了该理论在代码重构中的应用。

关键词 度量,距离,重构,极限编程,坏味道,移动方法,移动属性

A Location Method for Refactoring Based on Average Distance Measure

LIU Jian-Bin[1] ZHU Xue-Jun[1] LI Man-Ling[2] GUO Wei-Bin[3]

(Department of Computer Science, ShanTou University, ShanTou 515063)[1]

(Department of Computer Science, College of LingLing, YongZhou 425006)[2]

(The 1st Affiliated Hospital, Medical College, ShanTou University, ShanTou 515063)[3]

Abstract Software continuously becomes more and more difficult to understand and maintain with time elapse. Therefore, programmers must restructure the software before losing control of it. Refactoring is a key technique to acquire software quality, which is to restructure and reorganize software without changing its external observable behaviour. However, refactoring is not applied in practice as much as might be desired. One of main deterrents is the cost of detecting candidates for refactoring and choosing the appropriate refactoring transformation operation. To overcome the deterrent, a localization method for refactoring based-on average distance measure is proposed, formulas of calculating average distance and average distance difference between class members and classes, and a practical applied method in real systems are presented in this paper. This paper elaborates the theory of distance, and demonstrates application of the theory in code refactoring by a simple case.

Keywords Metrics, Distance, Refactoring, Extreme programming, Bad smell, Move method, Move attribute

1 引言

Martin Flower 在其著作[1]中是这样定义重构的:重构是对软件的内部结构所作的一种改变,这种改变在可观察行为(Observable Behaviour)不变的条件下使软件更容易理解,而且修改更廉价。重构不仅是程序维护的基本技术,也是设计和编程的基本准则,甚至是极限编程方法 XP(Extreme Programming)[2]的基本组成部分和核心技术。因此,重构是每个程序员应该熟练掌握和依靠的一项基本技术,对软件的开发维护和质量具有十分重要的意义。

"坏味道"(Bad Taste/Bad Smell)是指示程序代码的某一部分必须重写的一种隐喻[1],其鉴别始终是程序员面临的主要问题之一。在动手重构之前,程序员必须手工复查程序源代码以发现代码中感觉不佳或者可能存在潜在问题的地方。因而,"没有任何度量规矩比得上一个见识广博者的直觉"[1]。对于具有成千上万条代码行的较大规模程序而言,如果有自动探测代码"坏味道"的工具支持,那么程序员就可以克服上述障碍。目前,已出现多种"坏味道"的探测方法,常见的种类有:图形法[3]、度量法[4,5]及 UML 类图法[6,7]。

对这个课题的研究,我们主要出于两个动机:一是国内在自动重构技术方面的研究,尤其是重构定位方法方面的研究较少;二是我们认为 Frank Simon 的方法[5]有两个需要改进的地方:(1)它们呈递的重构定位工具,虽然具有计算简单,直观,易操作的优点;但是由于采用了三维虚拟现实建模语言浏览器(VRML Browse)来显示度量结果,当显示的类成员过多时,系统开销大、各类成员容易出现混乱[5],这样显然会影响重构效率;(2)"坏味道"不能直接被量化,在一定程度上,还需要借助"人的直觉"(human intuition)[1]在虚拟现实建模语言浏览器中进行判断,经常会出现重构操作"做与不做"的尴尬局面。为此,我们提出了一种快速有效的度量法—基于平均测距的重构定位方法。该方法通过计算类成员与类之间的平均距离来直接量化代码中的"坏味道",当出现"坏味道"时,就可以利用相应的重构方法对代码进行改造以消除探测到的问题,有助于实现一定程度的重构自动化。

*)广东省自然科学基金项目(032027)。刘建宾 博士,教授,主要从事软件方法与工具,软件工程,管理信息系统的研究工作。朱学军 硕士研究生,主要研究方向为软件方法学及 CASE 工具。

2 距离理论

经过多年来的研究，人们根据软件系统的特点已经提出了一系列度量技术，其中内聚性度量是软件度量学中一个重要的研究领域。在多次软件工程会议和一些软件工程杂志上发表了不少这方面的研究成果[8~11]。为了分析、探测代码中的“坏味道”，我们引入了文[5]中给出的基于内聚性的距离公式[8]，提出了类成员与类之间的平均距离及平均距离差公式。

2.1 距离

设 x,y 是类中定义的方法或属性，我们称之为成员(member)；B 是所有与 x 或 y 相关的方法和属性的集合，我们称之为特征集(set of properties)。为了便于计算成员间的距离，令 $B=B_m\cup B_a$，B_m 和 B_a[5]分别定义为：

B_m＝{既定方法，所有此方法使用的方法，所有此方法使用的属性}；

B_a＝{既定属性，所有使用此属性的方法}。

则两成员 x,y 间的距离公式为：

$$s(x,y)=1-\frac{|p(x)\cap p(y)|}{|p(x)\cup p(y)|}\text{ 其中 }p(x)=\{p_i\in B\mid x\text{ 含有 }p_i\}。\quad(1)$$

由式(1)可以得出下面三个结论：

(1) x 与 y 的距离越大，x 与 y 的聚合程度就越小；

(2) x 与 y 的距离越小，x 与 y 的聚合程度就越大；

(3) $0\leqslant s(x,y)\leqslant 1$。当 x 与 y 不相关时，$s(x,y)=1$；当 $x=y$ 时，$s(x,y)=0$。

2.2 平均距离

设 P_C 和 P_D 分别是类 C 和 D 的成员集。

$$\text{若 }x、y_j\in P_C\text{，则：}\bar{s}_{self}(x,C)=\frac{1}{|P_C|}\times\sum_{y_j\in P_C}s(x,y_j)\quad(2)$$

$$\text{若 }x\in P_C、y_j\in P_D\text{，则：}\bar{s}_{other}(x,D)=\frac{1}{|P_D|}\times\sum_{y_j\in P_D}s(x,y_j)\quad(3)$$

2.3 平均距离差

根据模块的“低耦合，高内聚”的原则，由公式②、③得：

$$\Delta S=\bar{s}_{self}(x,C)-\bar{s}_{other}(x,D)<0\quad(4)$$

其中，Δs 称为平均距离差。

式(4)的含义是：一般情况下，成员 x 在宿主类 C 中聚合的程度比在其它类 D 中聚合程度要大。这是量化“坏味道”的理论依据，当 $\Delta s>0$ 时，则暗示着代码中有存在“坏味道”的可能，这些征兆(坏味道)是开发人员以及项目经理都必须警惕的东西。

3 代码重构

软件自动重构技术是软件工程近十年来非常活跃的一个研究领域，最先对自动重构进行理论研究的是 Ralph Johnson，他把重构定义为：重构是使用各种手段重新整理一个对象设计的过程，目的是为了让设计更加灵活并且/或者更可重用。为了论证距离理论在重构中对“人的直觉”的辅助作用，我们重点研究了移动方法(Move Method)[1]和移动属性(Move Attribute)[1]这两种重构方法。

3.1 重构方法

·移动方法：是指把方法 m 从类 A 中移到使用此方法最多的类 B 中，而类 A 中的方法 m 则变换成一个简单的委托(delegation)，或者完全移去它。

·移动属性：是指把属性 a 从类 A 中移到使用此属性最多的类 B 中，同时修改属性 a 的所有使用者。

上述两种重构方法都是基于如下使用关系(use relation)：相互间大量使用的特性应聚集在同一个类中[5]。

3.2 “坏味道”

在程序开发和维护人员进行重构之前，必须明白“重构什么”和“怎样重构”。这些工作是通过探测代码中的“坏味道”和实施一定的重构操作来进行的。Martin Fowler 在他的书[1]中引用 Kent Beck 对“坏味道”隐喻，描述如何识别一种早期的警示信号，它们指示程序代码的某一部分必须重构。驱动使用“移动方法”和“移动属性”两种重构方法的“坏味道”[1]主要包括下列四种：

·弹散式修改(Shotgun Surgery)：对系统一个地方的改变涉及到其它许多地方的相关改变。这些变化率和变化内容相似的状态和行为通常应当放在同一个类中。

·依恋情节(Feature Envy)：对象的目的就是封装状态以及与这些状态紧密相关的行为。如果一个类的方法频繁用 Get 方法存取其它类的状态进行计算，那么就要考虑把行为移到涉及状态数目最多的那个类。

·不当的亲密关系(Inappropriate Intimacy)：当类过度访问其他类的私有部分，代表这个类有不当的亲密关系。可以使用移动方法及移动属性来降低亲密关系。

·数据类(Data Class)：数据类中除了属性及存取这些属性的 Set 及 Get 方法外别无它物。这种类一般多是为其它对象操作。如果其它类中常常使用到 Set 及 Get 方法，就要将这些行为用移动方法移到数据类中。

3.3 实例研究

下面，我们用一个典型的例子来说明平均测距理论和公式在程序程序中的应用。

假设有两个类 A、B：

```
Class class_A
{ static int aA1;
  static int aA2;
  public static void mA1()
  { class_B. aB1=0;
    class_B. aB2=0;
    class_B. mB1(); }
  public static void mA2()
  { class_B. aB2=0;
    aA1=0;
    aA2=0; }
  public static void mA3()
  { class_B. aB2=0;
    aA1=0;
    mA1();
    mA2(); }
  }
  Class class_B
  { static int aB1;
    static int aB2;
    public static void mB1()
    { aB1=0;
      mB2(); }
    public static void mB2()
    { aB1=0;
      mB1(); }
    public static void mB3()
    { aB1=0;
      mB1();
      mB2(); }
}
```

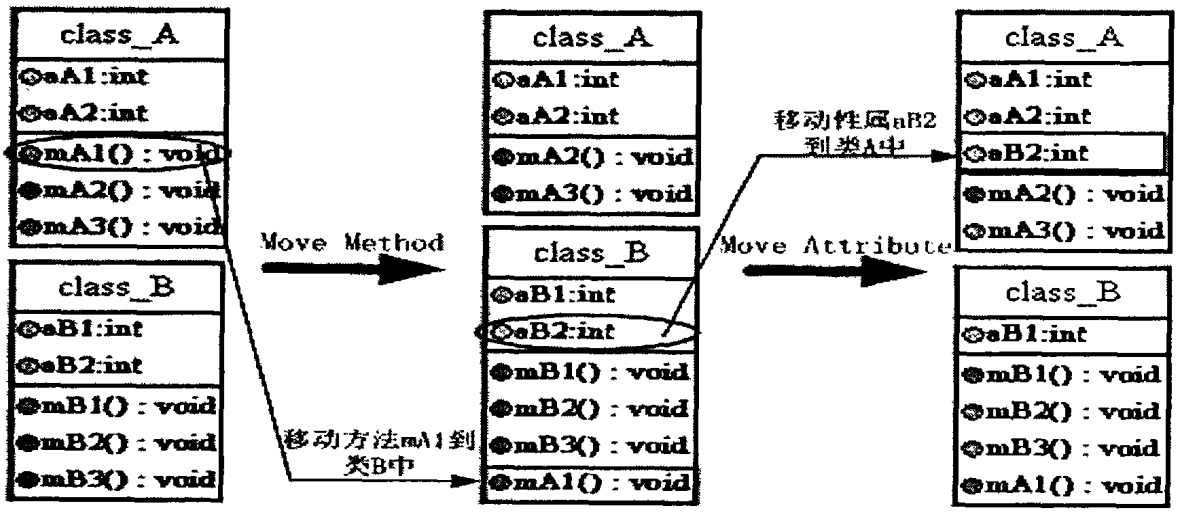
利用公式(1)～(3)可计算出成员间的距离以及成员与类之间的平均距离，其结果如表1所示。

表1 平均距离及平均距离差

		mA1	mA2	mA3	aA1	aA2			mB1	mB2	mB3	aB1	aB2
A	mA1	0	1	0.71	1	1	A	mA1	0.6	0.6	0.67	0.5	0.67
	mA2	1	0	0.5	0.6	0.5		mA2	1	1	1	1	0.67
	mA3	0.71	0.5	0	0.4	0.83		mA3	1	1	1	0.89	0.2
	aA1	1	0.6	0.4	0	0.75		aA1	1	1	1	1	0.6
	aA2	1	0.5	0.83	0.75	0		aA2	1	1	1	1	0.83
$\bar{s}_{self}(x,A)$		0.74	0.52	0.49	0.55	0.62	$\bar{s}_{other}(x,A)$		0.92	0.92	0.93	0.88	0.6
B	mB1	0.6	1	1	1	1	B	mB1	0	0	0.25	0.4	1
	mB2	0.6	1	1	1	1		mB2	0	0	0.25	0.4	1
	mB3	0.67	1	1	1	1		mB3	0.25	0.25	0	0.2	1
	aB1	0.5	1	0.89	1	1		aB1	0.4	0.4	0.2	0	0.88
	aB2	0.67	0.67	0.2	0.6	0.83		aB2	1	1	1	0.88	0
$\bar{s}_{other}(x,B)$		0.6	0.93	0.82	0.92	0.97	$\bar{s}_{self}(x,B)$		0.333	0.3	0.34	0.38	0.78

由表1的数据不难发现 $\Delta s=\bar{s}_{self}(mA1,A)-\bar{s}_{other}(mA1,B)=0.14>0$，$\Delta s=\bar{s}_{self}(aB2,A)-\bar{s}_{other}(aB2,B)=0.18>0$（黑体标识），从而违背了模块的低耦合高内聚的基本原则，这是因为类 A 中的方法 $mA1$ 与类 B 聚合得更紧，同样，类 B 中的属性 $aB2$ 与类 A 聚合得更紧。从而暗示着代码中存在两处“坏味道”，需要使用“移动方法”或“移动属性”重构方法以去除这些“坏味道”。

分别运用“移动方法”和“移动属性”对代码进行重构，将方法 $mA1$ 移到类 B 中（如图1.b 所示）、将属性 $aB2$ 移到类 A 中（如图1.c 所示）。经过两次重构操作后，代码不再有“坏味道”，从而变得更容易理解。



a. 重构前　　b. 运用 Move Method 后　　c. 运用 Move Attribute 后

图1 重构过程示意图

由于我们使用的实例忽略了部分细节，并不能完全反映真实的系统，因此，在实际的系统中，有时即便出现了某种征兆，也不一定需要重构，故可以设置一个很小的正数 ε（ε 会因类中成员个数而不同）。

1）当 $|\bar{s}_{other}(x,D)-\bar{s}_{self}(x,C)|<\varepsilon$ 时，代码味道不明显，需要借助“人的直觉”作进一步判断，重构操作可做可不做，有时还需要配合其它的重构方法；

2）当 $\varepsilon<\bar{s}_{other}(x,D)-\bar{s}_{self}(x,C)\leqslant 1$ 时，代码味道是好的，且差值越大，味道越好；

3）当 $-1\leqslant\bar{s}_{other}(x,D)-\bar{s}_{self}(x,C)<\varepsilon$ 时，代码味道是坏的，且差值越小，味道越坏。

4）如果 $(\forall D)(\bar{s}_{other}(x,D)\cong 1)$，则成员 x 远离这些类，若有多个成员 $\{x_i\}$，使得 $\bar{s}_{other}(x_i,D)\cong 1$，那么这些成员 $\{x_i\}$ 都可以从其相应的宿主类中移出，组成一个新的类，对应的重构方法为“提取方法(Extract Method)”和“提取属性(Extract Attribute)”。

通过本文中的定位方法，能较为准确地探测出代码中的“坏味道”，而且对代码进行重构后一般不会引入新的“坏味道”；

结论与展望 本文提出一种基于平均测距的重构定位方法，可以弥补“人的直觉”的缺陷，从而更有效地运用重构技术。该方法最大的好处在于计算简单，结果显示迅速，确实能够加快部分“坏味道”的识别定位速度，从而提高部分重构操作的效率，可以作为对 Simon 方法的简化、局部的改进和有益补充。不足之处在于它只能静态地分析程序源代码或部分框架代码，支持的重构方法太少，当类成员数目太大时，还会出现识别误差，有可能导致重构破坏代码的可理解性和可维护性，降低软件质量。

总体而言，目前面向对象软件度量方面的研究与面向对象的分析、设计技术以及程序设计语言的研究相比尚显薄弱，软件度量学理论还缺乏较为坚实的理论基础。因此，现阶段提出的度量技术还难以全面并准确地揭示程序中客观存在的种种问题，存在相当的局限性，有待于开展进一步的深入研究。

我们相信，随着自动重构理论、方法与工具的不断成熟，

会使重构变得更加实用，并期待着在不久的将来，重构技术真正在软件工程师团体内流行起来。

参考文献

1 Fowler M. Refactoring: Improving the Design of Existing Code. Addison Wesley Longman, Inc., Reading, Massachusetts, 1999
2 Beck K. Extreme Programming Explained. Upper Saddle River: Addison Wesley, 1999
3 Emden E van, Moonen L. Java quality assurance by detecting code smells. In: Proc. of the 9th Working Conf. on Reverse Engineering. IEEE Computer Society Press, Oct. 2002
4 Antoniol G, Fiutem R, Cristoforetti L. Using metrics to identify design patterns in object-oriented software. In: 5th Intl. Symposium on Software Metrics, March 1998
5 Simon F, Steinbruckner F, Lewerentz C. Metrics based refactoring. in CSMR, 2001. 30～38
6 Astels D. Refactoring with UML. http://www.agilealliance.org/articles/articles/RefactoringWithUML.pdf
7 Boger M, Sturm T, Fragemann P. Refactoring Browser for UML. http://www.xp2002.org/atti/Boger-Fragemann--Refactoring-BrowserforUML.pdf
8 Simon F, Löffler S, Lewerentz C. Distance Based cohesion measuring. In: proc. of the 2nd European Software Measurement Conf. (FESMA)99, Technologist Institute Amsterdam, 1999
9 Chidamber S R, Kemerer C F. A metrics suite for object oriented design. IEEE Transactions on Software Engineering, 1994
10 Bieman J M, Ott L M. Measuring functional cohesion: [Technical Report CS-93-109]. Michigan Technological University, 1993
11 Bieman J M, Kang B K. Measuring Design-Level Cohesion. IEEE Transac- tions on Software Engineering, 1998, 24(2)
12 Agile Modeling website. www.agilemodeling.com

（上接第161页）

ORC 的中间表示 WHIRL 是层次化的。WOPT 阶段的中间表示包括 WHIRL 的中层和低层。在 WOPT 之后，目标代码将经历代码生成(CG)阶段。这一个阶段主要由针对目标处理机的优化构成，它的中间表示非常接近目标机的指令。在 WOPT 的中间表示向 CG 的中间表示递降的过程中，会引入一些循环不变量。我们在代码生成阶段实现的循环不变量外提就是为了删除这些循环不变量。

SPEC 公司的 CPU2000[11]的整形测试例被用于评估在代码生成阶段的循环不变量外提(LICM)的性能。运行测试例的机器为 Itanium-2服务器。它具有 SMP 结构，每一个处理机的频率为896Mhz。

LICM 加速比如图10所示。在这里加速比被定义为：(无 LICM 时测试例的运行时间－有 LICM 时测试例的运行时间)/ 无 LICM 时测试例的运行时间 × 100%。Vortex 和 vpr 的循环小而多，因而，LICM 对这两个测试例子上能够取得比较显著的效果。gcc 和 crafty 的运行频率比较高的循环一般都比较大，循环不变量的外提的收益偏小。

表1

测试例	Loop#	Skip-Loop#	Skip-Loop%	LI#	Skip-LI#	Skip-LI%
Bzip2	180	115	64	198	116	59
Gzip	247	178	72	115	91	80
Vpr	426	298	70	277	208	75
crafty	462	367	79	191	180	94
mcf	50	22	44	45	37	82
parser	787	382	49	327	280	86
gap	2008	1765	88	350	247	71
perlbmk	808	685	85	115	79	69
eon	478	382	80	54	53	98
vortex	226	165	73	311	236	76
twolf	992	759	77	255	181	71
gcc	2839	2317	82	678	558	82
平均			72			78

表1反映了我们的代价模型的性能。从左往右各列的意义是：测试例、测试中的循环个数、被忽略的循环个数(由于循环体过大或循环的执行频率过低，详见4.1节中情况1和情况2)、被忽略的循环所占的百分比，在没有被忽略的循环中的循环不变量个数以及循环不变量中不被外提的个数，这些不外提的循环不变量所占的百分比。我们的实验结果，大约有72%的循环由于它们的循环体过大或者由于它们的循环迭代的个数较少而不进行循环不变量外提。在进行循环不变量外提的循环中，平均有78%的循环不变量不需要外提到循环之外。其中，由于 eon 的基本块较大，eon 中的循环不变量基本不需要外提到循环之外。

总结 在现代编译器中，循环不变量外提通常在中端进行。由于中端和后端的中间表示存在差别，当中端的中间表示向后端的中间表示递降时会引入循环不变量。在后端再次进行循环不变量外提是很有必要的。我们把不变量外提集成到指令调度的过程中以充分利用调度器已有的数据结构进行代价评估。实验显示我们的循环不变量外提能够提高目标代码1%的性能。在调度的同时进行循环不变量外提能够避免78%的循环不变量外提到循环之外。

参考文献

1 Click C. Global code motion/global value numbering. In: Proc. of the ACM SIGPLAN '95 Conf. on Programming Language Design and Implementation, La Jolla, California, June 1995. 246～257
2 Knoop J, Ruthing O, Steffen B. Lazy code motion. In: Proc. of the ACM SIGPLAN '92 Conf. on Programming Language Design and Implementation. SIGPLAN Notices, 1992, 27(7): 224～234
3 Cytron R, Ferrante J, Rosen B K, Wegman M N, Zadeckm F K. Efficiently Computing Static Single Assignment Form and the Control Dependench Graph. ACM Transactions on Programming Languages and Systems, 1991, 13(4): 461～486
4 Muchnick S S. Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers, San Francisco, California. 397～406
5 Coffman E G Jr. Computer and Job Shop Scheduling Theory. Wiley, New York, 1976
6 Muchnick S S. Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers, San Francisco, California. 539～543
7 Chow F C, Hennessy J L. The priority-based coloring approach to register allocation, ACM Trans. on Programming Languages and Systems, 1990, 12(4): 501～536
8 Muchnick S S. Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers, San Francisco, California, 193
9 ORC team. ORC v2.0 suite. http://sourceforge.net/projects/ipf-orc, 2001～2003
10 Intel Corp. http://www.intel.com/design/itanium/itanium/index.htm.
11 Standard Performance Evaluation Corporation. http://www.spec.org/cpu2000, 2003