软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2012,23(8):1965−1977 [doi: 10.3724/SP.J.1001.2012.04152]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

代码坏味的处理顺序*

高 原[1,2], 刘 辉[1,3+], 樊孝忠[1], 牛振东[1], 邵维忠[3,4]

[1](北京理工大学 计算机学院,北京 100081)

[2](第二炮兵装备研究院,北京 100085)

[3](高可信软件技术教育部重点实验室(北京大学),北京 100871)

[4](北京大学 信息科学技术学院 软件研究所,北京 100871)

Resolution Sequence of Bad Smells

GAO Yuan[1,2], LIU Hui[1,3+], FAN Xiao-Zhong[1], NIU Zhen-Dong[1], SHAO Wei-Zhong[3,4]

[1](School of Computer Science and Technology, Beijing Institute of Technology, Beijing 100081, China)

[2](The Second Artillery Equipment Research Institute, Beijing 100085, China)

[3](Key Laboratory of High Confidence Software Technologies of the Ministry of Education (Peking University), Beijing 100871, China)

[4](Software Institute, School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China)

+ Corresponding author: E-mail: liuhui08@bit.edu.cn

Gao Y, Liu H, Fan XZ, Niu ZD, Shao WZ. Resolution sequence of bad smells. *Journal of Software*, 2012,23(8): 1965−1977 (in Chinese). http://www.jos.org.cn/1000-9825/4152.htm

Abstract: This paper analyzes 10 kinds of typical bad smells in 4 aspects to recommend a resolution sequence: causes, symptom, influence and resolution. The resolution sequence has also been validated by a questionnaire survey and two experiments.

Key words: bad smell; software refactoring; resolution sequence; software maintenance; priority; schedule

摘 要: 选取了10种具有代表性的代码坏味进行研究.从每种代码坏味的产生原因、症状、对软件的影响以及相应的处理这4个方面进行分析,提出了一个代码坏味处理顺序的优先级.通过问卷调查和两个实验,对代码坏味处理顺序优先级进行了初步验证.

关键词: 代码坏味;软件重构;处理顺序;软件维护;优先级;调度

中图法分类号: TP311 **文献标识码**: A

软件在其生命周期内随着时间的变化而变化.由于需求的变更,软件也必须做出相应的改变.这种改变会打破软件原先的设计构架,使得软件结构不再清晰明了,软件的维护也变得越来越困难[1].软件重构[2]是在不改变软件外部特性的情况下,通过调整软件内部结构以提高软件的可理解性、可维护性和可扩展性.软件重构使得软件结构不断得到优化并保持最佳的状态,以适应外界需求的变化.同时,随着软件结构更加清晰,软件中存在的问题和缺陷更容易暴露,可以及时得到修改和更正,从而提高了软件的整体开发效率.软件重构的效果已经在

* 基金项目: 国家自然科学基金(61003065); 国家重点基础研究发展计划(973)(2011CB302604); 高等学校博士学科点专项科研基金(20101101120027); 新闻出版重大科技工程项目(GXTC-CZ-10015004/01); 北京理工大学优秀青年教师资助计划(2010Y0711)

收稿时间: 2011-01-10; 修改时间: 2011-08-12; 定稿时间: 2011-11-02

工业界和学术界获得广泛的认可[3,4].

代码坏味[2]则是指由于设计缺陷或坏的编码习惯而引入程序的、影响软件结构的程序代码.它总结了一些常见的软件问题模式,指出了代码中存在的潜在问题.通过代码坏味,能够发现和定位软件中存在的问题,明确应该在何处实施软件重构、实施何种重构.

在此前的工作中,刘辉等人[5,6]针对不同重构顺序会得到不同重构效果这一情况,对构成冲突的重构顺序进行了优化.通过实例证明,经过优化后的重构效果显著提高.现有的重构工具对于定位代码坏味能够提供很好的支持,但是不能在优化重构顺序之前找出所有可行的重构.因此在随后的研究中,刘辉等人[7]变换思维的角度,通过研究代码坏味之间的关联关系,提出了优化代码坏味的处理顺序的策略,以此来使重构的效果达到最大化.在此基础上,刘辉等人[8]挑选了 9 种代码坏味做进一步深入研究,分析了两两之间的相互影响,并通过两个开源的项目对提出的代码坏味处理顺序进行了评估.通过这些研究,我们逐步了解了代码坏味处理顺序和重构顺序对重构效果的影响.

此前的研究都是出于在理想状况下对代码坏味的处理顺序以及重构顺序的考虑,但在实际的软件开发过程中,会受到诸如时间和人力等一些资源的限制.尽管有辅助工具的支持,由于软件实际存在的代码坏味多种多样,软件重构依然困难重重.尤其是在项目紧迫、人员紧张的情况下,受到人力、时间等资源的约束,开发或者维护人员不可能处理软件中所有的代码坏味.为此,我们需要明确如何合理地选择代码坏味,同时依据代码坏味对软件的影响程度、修改代价等因素来决定处理的先后顺序.从而使得实施重构的效果最优化.

本文对现实存在的这些客观因素进行分析,并在此基础上对代码坏味的处理顺序做进一步研究,选取了具有代表性的 10 种代码坏味[2,9],对其形成原因和相应特征进行了分析,按照其对软件的影响程度、修改代价等因素,提出了代码坏味处理顺序的优先级.通过实验初步验证了其合理性和有效性.

本文第 1 节选取具有代表性的 10 种代码坏味,并对其进行分析.第 2 节抽取代码坏味的外部特征,按照对软件的影响程度、修改代价等因素进行分级.第 3 节对代码坏味的分级结果进行实验验证和分析.第 4 节针对实验中所采取的方法进行分析和说明.第 5 节给出结论.

1 典型代码坏味

本文选取了具有代表性的 10 种代码坏味[2,9]进行分析,见表 1.

Table 1 List of bad smells

表 1 代码坏味列表

	代码坏味
1	不合适的命名
2	重复代码
3	Switch 语句
4	特征依恋
5	平行继承体系
6	霰弹式修改
7	发散式变化
8	长方法
9	大类
10	长参数列表

(1) 不合适的命名

程序员在编程中图省事,不经推敲,简单地命名软件中的元素(包括文件名、类名、方法名、变量名等),造成软件代码的可阅读性和可理解性差.随着时间的推移,当程序员重新审视自己的软件时,根本想不起写代码时的思路.尤其当软件缺少必要的说明,维护人员面对一系列简单字符表示的名称,无从考证编程者的思路,很难对软件进行修改和维护.这种情况下,应该选择能够表达元素实际意义的名称进行命名,使得能够见名知意,增强软件的可阅读性和可理解性.

(2) 重复代码

程序员往往独立开发系统的不同部分,难免会出现一些相同或者相似的代码.更为常见的是,程序员在编写新的代码时发现,之前有相同或相似的代码可以直接使用,就进行拷贝和粘贴.重复代码使得软件更为复杂.当软件出现问题,或者功能需要调整时,必须对多处相同或相似的代码一并进行修改,容易遗漏或造成冲突.所以,应该对重复的部分进行合并,置于单独的类或方法中,使软件更为精简.

(3) Switch 语句

在软件中,经常会遇到使用 Switch 语句的情况.并非所有的 Switch 语句都是代码坏味,这里指的是以类型码为匹配对象进行选择的 Switch 语句.并且,同样的 Switch 语句会多次出现在不同的方法或类中.当需要增加新的或者修改 CASE 分支内语句时,就必须找到所有相同的 Switch 语句进行修改.随着开发规模的扩大,软件内部多处会出现重复的代码,造成系统冗余.Switch 语句可以使用多态来处理.找到与类型码有关的方法或者类,抽取相关代码到适当的类中,用子类或者状态/策略替换类型码,建立继承体系,将条件式替换为多态.

(4) 特征依恋

在软件中,某个类的方法必须通过调用其他类中大量的数据才能完成其自身的工作.即,该方法对其他类中数据的依赖远远超过其宿主类中的数据.这导致类之间的耦合度过高,增加了软件维护的难度.此类代码坏味应该通过抽取方法分离出位置不当的部分,使用移动方法和移动成员变量将软件元素置于适当的类中加以处理.

(5) 平行继承体系

在为一个类添加子类的同时,需要给另一个与其平行的类添加子类.经过多次添加之后,系统会变得复杂并难以修改.这样,在对一个类进行修改的同时必须修改与其平行的类,增加了系统的复杂性,使得系统难以维护.平行继承体系是霰弹式修改的一种特殊情况,其继承体系以一种并行的方式发展.应该使用移动方法和移动成员变量来重新分配特性,去除并行的继承体系.

(6) 霰弹式修改

在软件中,多个类之间耦合度过高.当外界发生变化时,需要同时修改软件中的多个类.此时,因为需要修改的代码过于分散,难以找出要修改的部分,导致遗漏某处重要修改,给软件留下隐患.处理的方法是利用现有类或者创造新类,通过移动方法和移动成员变量,将需要同时发生变化的代码抽取到同一个类中来处理.

(7) 发散式变化

在开发过程中,程序员往往无意识地使类承担过多的责任.当出现不同决策时,都要对同一个类进行修改以适应条件的变化,增加了类的不稳定性.即,每一次外界变化都要对类修改,并且每次修改都需要对整个类进行重新测试,以保证修改不会影响到类的其他部分.这增加了不必要的工作量,尤其当多种条件需要同时实现时,造成类内部矛盾.处理此类代码坏味,应将针对某特定变化的所有相应修改都抽取出来放在单独类中,保证每次变化需要修改的部分都在单一的类中,并且这个类内所有的方法都与这个变化相关.

(8) 长方法

软件中往往存在一个方法中有大量代码行的情况.在代码编写过程中,程序员会无意识地在方法中写入更多的代码.这增加了方法理解和修改的难度,同时也不利于重用.所以,应该通过抽取方法将原方法分解为更小的部分,并恰当地命名,使得通过名称就能理解方法实现的功能,以减少代码重复,使系统更易于理解、扩展和复用.

(9) 大 类

和长方法类似,在软件设计时,赋予一个类太多的职责,在类的内部设置了过多的成员变量或方法,增加了类的理解难度.并且,如果类中存在重复代码或者死代码,也不容易发现.处理方法是,通过对类中的成员变量进行分类和抽取,提取到新类中,或者作为该类的子类,对类进行精简.

(10) 长参数列表

在软件设计过程中,程序员往往为了减少软件模块之间的耦合关系,或者为了使模块功能更加通用化,而使用多个参数来对方法传递所需的内容.本身使用多个参数并非不可以,但是带来的主要问题是,使用这些方法需

要多个输入项,众多的参数不容易记忆和区分,增加了方法的使用难度.此类代码坏味应该通过以下方法处理:如果可以由对象得到参数则利用方法替换参数,如果可以将参数分类则通过生成参数对象来处理.

2 代码坏味处理优先级

目前,关于代码坏味形式化的研究进展缓慢,部分学者甚至认为,代码坏味不可能进行形式化[8].因此,关于代码坏味的影响程度、安全隐患等因素也难以进行定量研究.受此影响,代码坏味的分级研究也难以通过量化研究这些指标而实现.因此,综合考虑代码坏味对软件的影响程度、分布范围、安全隐患和修改代价,定性的将代码坏味分为如下 5 级(级别 1 优先级最高,级别 5 优先级最低):

(1) 级别 1

此级别的代码坏味在软件中分布广泛,影响范围大,但修改代价最小,容易以较小的代价获取高回报.不合适的命名属于此级别的代码坏味,它降低了软件的可阅读性和可理解性.当软件由不同人员来开发,彼此之间提供的接口难以理解时,就会增加软件集成的难度.尤其当团队发生变更时,继任开发人员无法读懂前任编写的代码,导致修改成本大增.该代码坏味在软件中涉及的元素多(包括文件名、类名、方法名、变量名等),分布范围广,但修改方便,只需赋予元素相应容易理解的名称.因此,对此代码坏味进行处理的投入产出比很高,应当首先处理.

(2) 级别 2

此级别的代码坏味会给软件带来安全漏洞,导致软件不稳定,影响的范围较广,而修改代价相对较小.此级别的代码坏味包括重复代码和 switch 语句.软件中时常存在多处重复的或实现功能相似的代码.Switch 语句的本质问题也在于重复.当软件出现问题或者功能需要调整时,由于各处相似的代码片段所需修改的内容基本相同,可能错过某个片段,导致软件安全漏洞或功能上的不一致.在处理方面,已有较为成熟的辅助工具.因此,其实现代价小.

(3) 级别 3

此级别的代码坏味造成类之间耦合度过高,不利于软件复用,并且使软件难以扩展和维护.此级别的代码坏味包括特征依恋、平行继承体系和霰弹式修改.此类代码坏味的修改范围限于耦合的类之间.当需要对其中的一个类进行修改时,必须将与其有关的所有类都进行修改.尤其当多个类之间都存在这种耦合关系时,软件的逻辑关系就会非常复杂,修改起来就会比较困难,并且容易出错.

(4) 级别 4

此级别的代码坏味同样会造成类之间的高耦合度,但是涉及的范围相对较小,修改的范围也较小.它破坏了类的封装性,给软件带来隐患.此级别的代码坏味包括发散式变化.对于发散式变化,由于类受到外界影响的因素较多,每次变化都要对该类进行修改.如果将受外界变化的部分单独抽取出来,修改时就不会给整个类造成影响.降低类之间的耦合程度,可以相应地缩小修改的范围.

(5) 级别 5

此级别的代码坏味增加了软件内部模块的理解难度,降低了类和方法的可复用度,影响范围最小.此级别的代码坏味包括长方法、大类和长参数列表.对于前两者,由于方法或者类中包含了太多的内容,增加了理解的难度,降低了可复用度.对其进行适当的分解,可以增强软件的可理解性和灵活性;对于后者,由于方法的参数过多,不容易记忆,造成使用不便.

通过以上分析,可将代码坏味处理优先级归纳为表 2.

Table 2 Priority of bad smell solution

表 2 代码坏味处理优先级

等级	特征	代码坏味
级别 1	• 外部理解复杂化 • 影响范围大 • 易于更改	不合适的命名
级别 2	• 导致安全漏洞 • 易于工具检测	重复代码 Switch 语句
级别 3	• 高耦合 • 软件逻辑复杂 • 难以修改	特征依恋 平行继承体系 霰弹式修改
级别 4	• 高耦合 • 破坏封装性 • 影响范围小	发散式变化
级别 5	• 内部理解复杂化 • 难以复用	长方法 大类 长参数列表

3 实验验证

为了对代码坏味分级结果进行验证[10,11],设计了问卷调查和两个实验.

3.1 问卷调查

为了了解代码坏味在实际使用中的情况,采用问卷调查的方式进行调研.调查的人群确定为 4 类:有一定开发经验的在校学生、从事软件工程教育的教师、在企业从事软件开发的人员和在研究机构从事软件工程研究的人员.将选定的代码坏味提交给调查对象,让他们根据自身的理解和在实际中的应用对这 10 种代码坏味处理的优先级进行排序,然后对回收的问卷进行统计分析.在问卷中,针对有人可能不熟悉有关代码坏味的情况,对其进行了简要的说明.问卷允许只选其中某些熟知的代码坏味进行排序.

问卷共发出 100 份,收回 98 份.这里,使用 *A*,*B* 代表代码坏味类型,以每份问卷调查代码坏味的排列顺序为一个序列,子序列 *AB* 表示代码坏味 *A* 排列在 *B* 之前.在比较 *A*,*B* 的处理优先级时,只考虑序列中同时出现 *A*,*B* 的情况,如果子序列 *AB* 的支持度大于 *BA*,认为代码坏味 *A* 的处理优先级高于 *B*.

$$support(AB)=\frac{count(AB)}{count(AB)+count(BA)} \tag{1.1}$$

$$support(BA)=\frac{count(BA)}{count(AB)+count(BA)} \tag{1.2}$$

其中,

- *support*(*AB*)为 *AB* 的支持度,表示代码坏味 *A*,*B* 均出现的情况下,*A* 优先于 *B* 的概率;
- *support*(*BA*)为 *BA* 的支持度,表示代码坏味 *A*,*B* 均出现的情况下,*B* 优先于 *A* 的概率;
- *count*(*AB*)为包含代码坏味 *A*,*B* 且 *A* 在 *B* 之前的问卷份数;
- *count*(*BA*)为包含代码坏味 *B*,*A* 且 *B* 在 *A* 之前的问卷份数.

基于上述公式,通过对问卷调查结果的统计分析,得到代码坏味处理顺序优先级概率见表 3.

表 3 中,代码坏味自上而下按照文中的分级结果进行排序,右侧对应于代码坏味之间处理顺序先后的比较数据.可以看出,在不同级别之间,高一级别的代码坏味对低一级别的代码坏味的支持度较高.例如,不合适的命名对于重复代码和 Switch 语句的支持度分别为 0.83 和 1.0;特征依恋对于发散式变化的支持度为 0.99.而同一级别内部的代码坏味之间支持度相对较低.例如,平行继承体系对于霰弹式修改的支持度为 0.53;霰弹式修改对于特征依恋的支持度为 0.66.当然也存在特殊情况,发散式变化对于长方法的支持度相对较低,只有 0.54.通过对统计结果的分析,认为是由于发散式变化在整个问卷调查中的出现的次数相对较少造成的.因此,问卷调查结果中对代码坏味的处理顺序与本文分级结果基本一致.

Table 3 Analysis of questionnaire statistics

表 3 问卷调查统计数据分析结果

等级	*A*	*B*									
		不合适的命名	重复代码	Switch 语句	平行继承体系	霰弹式修改	特征依恋	发散式变化	长方法	大类	长参数列表
1	不合适的命名		0.83	1.0	0.95	0.93	1.0	1.0	0.92	0.96	0.99
2	重复代码 Switch 语句			1.0	1.0 0.86	1.0 0.82	1.0 0.94	1.0 1.0	0.99 0.83	1.0 0.86	1.0 0.84
3	平行继承体系 霰弹式修改 特征依恋					0.53	0.81 0.66	1.0 1.0 0.99	0.80 0.83 0.80	0.85 0.97 0.84	0.85 0.90 0.84
4	发散式变化								0.54	0.85	0.81
5	长方法 大类 长参数列表									0.78	1.0 0.85

3.2 重构历史分析

3.2.1 实验数据

实验使用的数据源是基于 eclipse 官方网站公布的于 2009 年通过 Eclipse Usage Data Collector 收集的关于用户使用 eclipse 工具开发项目的历史数据[**].每条数据包含 7 项内容:UserID,Event,Kind,BundleId,BundleVersion,Description 和 Timestamp.实验关注其中的 3 项:UserID 用来区分重构用户,Description 用来描述重构类型,Timestamp 用来标明重构时间.数据示例:160853,“executed”,“command”,“org.eclipse.jdt.ui”,“3.4.0.v20080603-2000”,“org.eclipse.jdt.ui.edit.text.java.rename.element”,1230367679294.其中,160853 用以标识独特的工作空间,executed 表示发生的事件,command 表示类型,org.eclipse.jdt.ui 是插件包的标识,3.4.0.v20080603-2000 表示该插件包的版本号,org.eclipse.jdt.ui.edit.text.java.rename.element 是对该事件的具体描述,1230367679294 表示事件发生的时间.本次实验数据采集于 2008.12.1~2009.12.31,共计 219 779 位用户使用 eclipse 工具对软件实施重构,数据总量为 829 932 条.

代码坏味总是与某些重构操作相对应的[2],因此,执行了某种重构操作很大程度上意味着相对应的代码坏味的存在,通过研究整理文献[2]中的代码坏味与重构操作的关系,给出以下代码坏味与其主要重构操作之间的关系,见表 4.

Table 4 Mapping relationship between bad smells and refactorings

表 4 代码坏味与主要重构操作的对应关系

	代码坏味	主要重构操作
1	不合适的命名	重命名方法
2	重复代码	提取方法
3	Switch 语句	提取方法
4	特征依恋	提取方法、迁移(方法、值域)
5	平行继承体系	迁移(方法、值域)
6	霰弹式修改	迁移(方法、值域)
7	发散式变化	提取类
8	长方法	提取方法
9	大类	提取接口
10	长参数列表	引入参数对象

实验的设计思路:既然重构操作与某种代码坏味相对应,那么通过比对重构操作之间的执行顺序,可以分析对应代码坏味处理的先后顺序.分析的结果反映出实际重构应用中代码坏味的处理顺序,用以验证本文提出的代码坏味处理顺序优先级的合理性.

** The Eclipse Foundation. Usage Data Collector Results, June 12th, 2010. http://archive.eclipse.org/technology/phoenix/usagedata/

3.2.2 处理顺序分析

本实验中采取的策略是分析用户操作数据,找出用户执行重构操作的先后顺序,验证哪些代码坏味得到优先处理.

从数据中抽取执行重构操作活跃度最高的 1 000 名用户,划分重构的周期.以两个工作日内不实施重构操作为重构周期的分界点.用时间戳记录标定在每个周期内重构操作的先后次序.这里使用 *A*,*B* 代表重构操作,以每周期内重构操作执行顺序为一个序列,子序列 *AB* 代表重构操作 *A* 先于 *B* 出现.与问卷调查不同的是,序列中 *A*,*B* 会交叉出现.在比较 *A*,*B* 的执行优先级时,考虑序列中同时出现 *A*,*B* 的情况,如果子序列 *AB* 的支持度大于 *BA*,认为重构操作 *A* 的处理优先级高于 *B*.

$$support(AB)=P(AB)=\frac{count(AB)}{count(A\&B)} \tag{1.3}$$

$$support(BA)=P(BA)=\frac{count(BA)}{count(A\&B)} \tag{1.4}$$

其中,

- *support*(*AB*)为 *AB* 的支持度,表示在重构操作 *A*,*B* 同时出现的情况下,*A* 先于 *B* 的概率;
- *support*(*BA*)为 *BA* 的支持度,表示在重构操作 *A*,*B* 同时出现的情况下,*B* 先于 *A* 的概率;
- *count*(*A*&*B*)为重构操作 *A*,*B* 交叉出现的周期数.

基于上述公式,通过计算得出重构操作顺序优先级概率,见表 5.

Table 5 Priority probability of refactoring

表 5 重构操作优先概率

A	*B*					
	重命名方法	提取方法	迁移	提取类	提取接口	引入参数对象
重命名方法		0.802	0.875	0.983	0.990	0.997
提取方法			0.605	0.947	0.949	0.986
迁移				0.874	0.876	0.934
提取类					0.630	0.882
提取接口						0.75
引入参数对象						

因此,根据表 5 构造代码坏味与重构操作序列的对应关系[2]图,如图 1 所示.

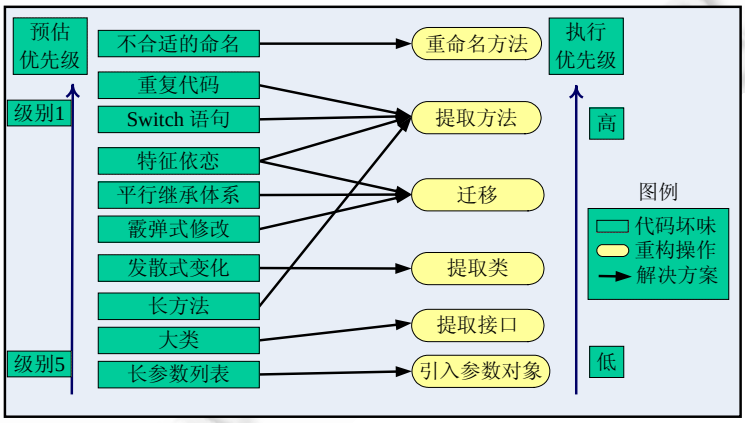


Fig.1 Mapping relationship between bad smell priority and refactoring sequence

图 1 代码坏味分级与重构操作序列的对应关系图

在图 1 中,左侧是按照文中代码坏味分级结果的顺序从上往下依次排列,右侧是将表 5 中的数据按照重构操作的相对先后顺序由上往下依次排列,图中的连线表示代码坏味与其对应的主要重构操作之间的关联.从图

中可以看出,执行重构操作的先后顺序与文中提出的代码坏味处理顺序优先级基本一致(两者之间的连线基本上是平行的,几乎没有交叉).同时也可以看出,长方法与其对应的提取方法重构操作之间在图上形成了一条长交叉线,与分级结果有较大的偏差.提取方法涉及到 4 种代码坏味:重复代码、Switch 语句、特征依恋和长方法.之所以优先级别比较高,是由于该重构操作应用于重复代码、Switch 语句和特征依恋与应用于长方法的比例约为 3:1[12].

3.3 重构效果验证

3.3.1 实验设计

为了评估本文提出的代码坏味处理顺序对重构效果的影响,本文设计了一个对比实验:将参考代码坏味分级结果的重构方式与传统的重构方式进行实验对比.

实验对象是一个企业管理系统 EMS:Enterprise Management System(http://download.csdn.net/detail/alexgy/3593675).EMS 是一个开源系统,用于中小型企业对员工的日常管理工作.现在企业对原有 EMS 系统提出了新的需求 R:补充员工信息、批量管理考勤、查询员工奖惩记录以及修正员工培训记录.实验将遵从“两顶帽子法则”,即,首先通过软件重构提高软件的可扩展性,然后再进行功能扩展以满足新的需求.考虑到项目的时间限制,将安排两个工作日进行系统重构,然后进行功能扩展.具体的实验流程如图 2 所示.

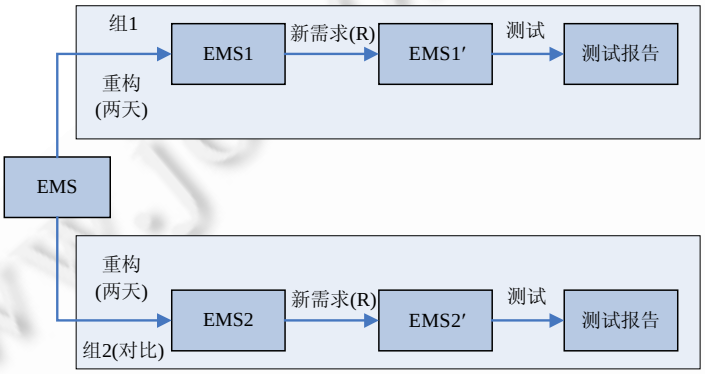


Fig.2 Flow chart of refactoring scheme comparison experiment

图 2 重构方案比对实验流程图

参加本次实验的是有一定开发经验的在校学生.选取 4 名学生,每两名分为一组.第 1 组(Group1)按照本文提出的代码坏味处理顺序对项目进行重构.对原有系统重构后得到 EMS1,在此基础上,对系统进行扩展实现新功能 R(EMS1′).作为对照,第 2 组(Group2)按照传统的重构方式对项目进行重构.在原有系统上得到 EMS2,并基于此扩展系统实现新功能 R(EMS2′)(所有 EMS,EMS1,EMS2,EMS1′,EMS2′等源码均可从网上下载***).

3.3.2 实验结果与分析

在重构过程中,Group1 总共执行重构 30 次,Group2 共执行重构 26 次.开发阶段,在 EMS2 的基础上进行扩展以实现新功能,一共耗时 11 人小时;在 EMS1 的基础上进行扩展以实现新功能,一共耗时 9 人小时,相对于前者时间上缩短了 18%.针对新的需求 R,我们设计了测试用例,统一对最终的项目进行测试.EMS2′测出的 Bug 数为 8;而 EMS1′测出的软件失效节点数为 3,数量上远远低于前者.

为了进一步找出造成两者差异的原因,我们对两组重构过程和项目代码做了进一步分析.由于对时间进行了限定,两个重构方案在规定的时间内都没有完成对所有代码坏味的处理.表 6 列出了两组重构不同和相同的部分.

*** The Eclipse Foundation. Usage Data Collector Results, June 12th, 2010. http://archive.eclipse.org/technology/phoenix/usagedata/

Table 6 Comparison of two group refactorings

表 6 两组重构对比

类型	相关类	组 1	组 2
不合适的命名	BringUpOperateI.java	√	√
	RecordOperateEditI.java	√	√
	RecordOperateNewI.java	√	√
Shotgun surgery	BringUpOperateB.java	√	√
	BringUpOperateI.java	√	√
	PersonnelSet.java	√	√
	AddUser.java	√	√
Feature envy	BringUpOperateB.java	√	√
	BringUpOperateI.java	√	√
	BringUpSelectedB.java	√	√
	BringUpSelectedT.java	√	√
	RecordOperateEditB.java	√	√
	RecordOperateEditI.java	√	√
	RecordOperateNewB.java	√	√
	RecordOperateNewI.java	√	√
Inappropriate intimacy	BringUpOperateB.java	√	√
	BringUpOperateI.java	√	√
	BringUpSelectedB.java	√	√
	BringUpSelectedT.java	√	√
	RecordOperateEditB.java	√	√
	RecordOperateEditI.java	√	√
	RecordOperateNewB.java	√	√
	RecordOperateNewI.java	√	√
Divergent change	BringUpOperateB.java	√	√
	BringUpOperateI.java	√	√
	RecordOperateEditB.java	√	√
	RecordOperateEditI.java	√	√
	RecordOperateNewB.java	√	√
	RecordOperateNewI.java	√	√
Inappropriate name	IndexFrame.java	√	×
	LandFrame.java	√	×
	BringUpOperateB.java	√	×
	BringUpSelectedB.java	√	×
	BringUpSelectedT.java	√	×
	RecordOperateEditB.java	√	×
	RecordOperateNewB.java	√	×
	RecordSelected.java	√	×
	Data.java	√	×
Duplicate code	IndexFrame.java	√	×
	LandFrame.java	√	×
	DeptAndPersonnel.java	√	×
	DeptTree.java	√	×
	BringUpOperateB.java	√	×
	BringUpOperateI.java	√	×
	RecordOperateEditB.java	√	×
	RecordOperateEditI.java	√	×
	RecordOperateNewB.java	√	×
	RecordOperateNewI.java	√	×
	Timecard.java	√	×
	AddAccountItem.java	√	×
	CreateCriterionSet.java	√	×
	CriterionSet.java	√	×
	PersonnelSet.java	√	×
	ReportForms.java	√	×
	UpdatePassword.java	√	×

Table 6 Comparison of two group refactorings (Continued)

表 6 两组重构对比(续)

类型	相关类	组 1	组 2
Shotgun surgery	DeptAndPersonnel.java	√	×
	RecordSelected.java	√	×
Long method	DeptAndPersonnel.java	×	√
	DeptTree.java	×	√
	BringUpOperateB.java	×	√
	BringUpOperateI.java	×	√
	BringUpSelectedB.java	×	√
	BringUpSelectedT.java	×	√
	RecordOperateEditB.java	×	√
	RecordOperateEditI.java	×	√
	RecordOperateNewB.java	×	√
	RecordOperateNewI.java	×	√
	Data.java	×	√
	PersonnelSet.java	×	√
	AddUser.java	×	√

两组在进行重构时的思路不同,体现在重构内容上的差异.Group1 按照本文中代码坏味处理的顺序逐个排除,所以对不合适的命名和重复代码两种代码坏味进行了较为充分的重构.Group2 则按照传统的方法先以类为单位逐个处理,然后再处理类之间存在的代码坏味.下面对 Group1 实施而 Group2 未实施的重构进行分析,探究这些重构如何影响后续的软件扩展.

Group1 对项目中的变量、方法以及类的名称进行了规范和整理,透过名称就可以理解变量、方法或类的代表的含义,而不必通过研读整段的代码.例如,在源代码中 LandFrame.java 和 IndexFrame.java 两个类分别代表登录界面和软件主界面,将名称改为 LoginFrame.java 和 MainFrame.java 能够直观地反映类代表的含义.

对重复代码进行重构,抽取代码的重复部分作为公共的类或者方法,有利于代码的重用和后续的开发.例如在 EMS 中,类 BringUpOperate.java,PersonnelSet.java 和 AddUser.java 中有一段关于生成添加员工信息的代码.在批量管理考勤和修正员工培训记录两项新需求中都需要使用该功能(类 TrainManagementOperate.java 和 TimecardOperate.java).Group1 对这段代码进行了重构,在扩展功能时就可以直接拿来使用.然而 Group2 没有进行重构.因此在扩展功能时,很难发现可重用的代码,即使找到可重用的代码也难以重用.因为在使用时必须根据当前的变量或参数对可重用部分进行调整,这势必会增加工作量,增加开发的时间.

对重复的代码进行重构还可以发现原有的 Bug,并防止 Bug 扩散.例如在 EMS 中,类 BringUpOperateB.java, RecordOperateEditB.java,RecordOperateNewB.java,RewardsAndPunishment.java 和 Timecard.java 中都有关对日期进行验证的代码,但不同类中实现的代码也不同.类 RewardsAndPunishment.java 中的验证方法会引起与 java.sql.Date 类发生冲突而导致错误.Group1 对在对该代码进行重构时,通过比对代码发现 Bug,并进行了纠正.Group2 没有进行重构,在扩展新的功能时直接使用,间接地给新功能引入了 Bug,导致软件 Bug 数增加.从减少 Bug 数目的角度来看,软件的质量得到了提高.

类似的重构包括:

- 将涉及 Record 的类重命名为 Dossier,将涉及 Bringup 的类重命名为 TrainManagement,更有利于代码的理解;
- 抽取类 DeptAndPersonnel.java 和 DeptTree.java 中的生成树模型部分成为公共部分,有利于代码的重用.

3.4 讨 论

3.4.1 调研人群

在进行问卷调查时共选取了 4 类人群,其中,在企业从事软件开发的人员和在研究机构从事软件工程研究的人员其开发的经验相对比较丰富,所接触到的项目也较多,因而在开发过程中所碰到的代码坏味的类型较为

全面,从事重构的经验积累也更多,实践经验丰富,但理论研究方面有所欠缺;有一定开发经验的在校学生和从事软件工程教育的教师在学校主要从事学习和教学工作,实际接触到的开发项目较少,实践经验不足,但在软件工程方面理论基础较为扎实,理解事物更为深刻.尤其对于学生来讲,接受新事物的能力较强,并且将成为从事软件工程的主体.因此,选取人群的互补性增强了问卷调查的可信度.

3.4.2 代码坏味与重构操作的对应关系

一种代码坏味不仅仅对应一种重构操作,在进行实验时为了避免复杂性,只是将代码坏味所对应的主要重构操作挑选出来,没有将代码坏味所对应的所有重构操作都考虑在内.因此,执行主要重构操只能近似标定某种代码坏味的存在,说明执行某种重构操作在很大程度上是为了处理某种代码坏味,但没有执行某种重构操作并不能一定说明就不存在某种代码坏味.

3.4.3 数据局限性

实验采用的数据源是eclipse UDC[****]收集的重构历史数据.eclipse工具本身提供重构功能,但是其实现重构的功能是有限的,有时可能会通过人工手动进行重构操作.这样,eclipse就不能记录所有的重构操作,也就不包含在实验使用的历史数据之中.因此,本文的分析是建立在通过使用 eclipse 工具本身的重构功能所收集的数据基础上的,使用其他数据可能结果会有偏差.

3.4.4 重构项目

用于重构效果验证所选用的项目规模较小,项目中包含的代码坏味种类和数量有限,因此对项目进行重构所需的时间较短.另外,受项目规模的影响,在其基础上新增需求的规模有限(只增加了 4 项需求),所以在重构的基础上进行功能扩展的时间较短,这样就影响了重构对于项目功能扩展的效果.为了最大限度地减少人员差异对实验的影响,从实验室软件工程课题组中选取从事软件重构研究而且技术能力相当的人员参与实验.但是个体之间不可避免会存在差异,并且参与实验的人员数量较少,这必然会影响到实验的结果.因此,该实验的结果仅仅作为实际应用的参考.

在今后的深入研究中,会着重实验方案设计的合理性,从样本的质量和数量上提高实验结果的普遍性.

4 相关工作

张家晨[13]对软件重构做了系统性的研究,对已有的软件重构方法进行了补充和完善.王忠杰等人[14]提出了一种面向复用成本优化的、基于局部性原理与实例集分解的构件重构方法.伞云飞等人[15]提出了一种利用静态分析技术自动重构 J2EE 应用软件静态模型的方法,通过该方法实现了 J2EE 应用软件静态模型重构.王青等人[16]研究和讨论了软件缺陷预测技术的起源、发展和当前所面临的挑战,对主流的缺陷预测技术进行了分类讨论和比较,并对典型的软件缺陷的分布模型给出了案例研究.景涛等人[17]给出了关联缺陷的定义,并通过一个软件实例验证了缺陷的关联关系,提出了一种缺陷放回的测试方法用来剔除关联缺陷.陈生庆等人[18]面向方面重构的程序和它的面向对象程序原型等价性的证明方法.

Mantyla 等人[19,20]通过提取特征,将文献[2]中所提及的 22 种代码坏味分为 7 类,通过实证研究证明了代码坏味之间的关联性,并对其进行了度量.研究主要侧重于代码坏味之间的关联关系.

Li 和 Shatnawi[21,22]挑选了 6 种代码坏味,通过对开源的已发布的 eclipse 项目的研究,得出了代码坏味导致面向对象系统错误的可能性以及导致系统错误的严重等级;该研究证明了代码坏味导致类错误的可能性以及实施重构确实能都减少类中的错误.

Zhang 和 Hall[23]采用专家小组评估的方式针对 5 种代码坏味进行量化,以改进甄别代码坏味的效率.Lozano 等人[24]通过对代码坏味的历史变更数据从面向对象软件系统可维护性的角度来分析哪些代码坏味使得系统更加难以维护,从而使重构的效果最大化.我们在两者采用的实验方法和数据的基础上做了进一步细化,丰富了

**** The Eclipse Foundation. Usage Data Collector Results, June 12th, 2010. http://archive.eclipse.org/technology/phoenix/usagedata/

实验内容,扩充了实验数据.

Murphy-Hill 等人[25]提出了 7 条原则,以设计代码坏味侦测工具,通过实现的工具来对选定的 11 种代码坏味进行等级标注;陈丹等人[26]为了设计和实现代码坏味自动检测工具,根据侧重点不同将代码坏味分为 4 类.该研究主要对代码坏味的特征进行了分析,致力于使自动检测工具能够更准确地识别代码坏味.

5 结 论

软件重构作为提高软件质量的一种有效手段,已经成为软件开发和维护的关键活动之一.软件重构活动与软件中存在的代码坏味息息相关.尤其在资源受限的情况下,对代码坏味进行优先级划分以决定处理的先后顺序,对于合理而有效地实施软件重构具有积极的意义.本文针对选取的具有代表性的 10 种代码坏味进行了分析,根据其对软件的影响程度、修改代价等因素进行分级.通过实验验证了文中提出的代码坏味处理顺序的合理性和有效性,从而对实施软件重构具有一定的指导意义.

References:

[1] van Gurp J, Bosch J. Design erosion: Problems & causes. Journal of Systems and Software, 2001,61(2):105−119. [doi: 10.1016/S0164-1212(01)00152-2]

[2] Fowler M, Wrote; Hou J, Xiong J, Trans. Refactoring: Improving the Design of Existing Code. Boston: Addison-Wesley Longman Publishing Co., Inc., 1999 (in Chinese).

[3] Kataoka Y, Imai T, Andou H, Fukaya T. A quantitative evaluation of maintainability enhancement by refactoring. In: Proc. of the 24th Int'l Conf on ICSM. 2002. 576−585.

[4] Ó Cinnéide M, Nixon P. Composite refactorings for Java programs. Technical Report, Department of Computer Science, University College Dublin, 2000.

[5] Liu H, Li G, Ma ZY, Shao WZ. Scheduling of conflicting refactorings to promote quality improvement. In: Proc. of the 22nd IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE 2007). 2007. 489−492. [doi: 10.1145/1321631.1321716]

[6] Liu H, Li G, Ma ZY, Shao WZ. Conflict aware scheduling of software refactorings. IET Software, 1008,2(5):446−460.

[7] Liu H, Yang L, Niu ZD, Ma ZY, Shao WZ. Facilitating software refactoring with appropriate resolution order of bad smells. In: Proc. of the 7th Joint Meeting of the European Software Engineering Conf. (ESEC) and the ACM SIGSOFT Symp. on the Foundations of Software Engineering (ESEC/FSE 2009). 2009. 265−268. [doi: 10.1145/1595696.1595738]

[8] Liu H, Ma ZY, Shao WZ, Niu ZD. Schedule of bad smell detection and resolution: A new way to save effort. IEEE Trans. on Software Engineering, 2012,38(1):220−235. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5680918&isnumber=435946 [doi: 10.1109/TSE.2011.9]

[9] Wake WC. Refactoring Workbook. Boston: Addison-Wesley Longman Publishing Co., Inc., 2003. 381−384.

[10] Murphy-Hill E, Parnin C, Black AP. How we refactor, and how we know it. In: Proc. of the 7th Int'l Conf. on Software Engineering. 2009. 287−297. [doi: 10.1109/ICSE.2009.5070529]

[11] Zimmermann T, Weiβgerber P. Preprocessing CVS data for fine-grained analysis. In: Proc. of the Int'l Workshop on Mining Software Repositories (MSR 2004). 2004. 2−6. [doi: 10.1049/ic:20040466]

[12] Counsell S, Hierons RM, Hamza H, Black S, Durrand M. Exploring the eradication of code smells: An empirical and theoretical perspective. In: Proc. of the Advances in Software Engineering. 2010. [doi:10.1155/2010/820103]

[13] Zhang JC. Research on software refactorting [Ph.D. Thesis]. Changchun: Jilin University, 2004 (in Chinese with English abstract).

[14] Wang ZJ, Xu XF, Zhan DC. Reuse cost optimization oriented component refactoring method. Journal of Software, 2005,16(12): 2157−2165 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/16/2157.htm [doi: 10.1360/jos162157]

[15] San YF, Jin MZ, Liu C, Yan L. Research on reconstructing J2EE software static model. Application Research of Computer, 2007,24(11):196−199 (in Chinese with English abstract).

[16] Wang Q, Wu SJ, Li MS. Software defect prediction. Journal of Software, 2008,19(7):1565−1580 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/19/1565.htm [doi: 10.3724/SP.J.1001.2008.01565]

[17] Jing T, Jiang CH, Hu DB, Bai CG, Cai KY. An approach for detecting correlated software defects. Journal of Software, 2005,16(1): 17−28 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/16/17.htm

[18] Chen SQ, Zhang LC, Chen GM. An equivalence proving in formal method for aspect-oriented refactory. Computer Science, 2006, 33(7):257−261 (in Chinese with English abstract).

[19] Mantyla M, Vanhanen J, Lassenius C. A taxonomy and an initial empirical study of bad smells in code. In: Proc. of the 25th Int'l Conf. on ICSM. 2003.

[20] Mantyla M. Bad smells in software—A taxonomy and an empirical study [Ph.D. Thesis]. Helsinki: Helsinki University of Technology, 2003.

[21] Li W, Shatnawi R. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. Journal of Systems and Software, 2007,80(7):1120−1128. [doi: 10.1016/j.jss.2006.10.018]

[22] Shatnawi R, Li W. An investigation of bad smells in object-oriented design. In: Proc. of the 3rd Int'l Conf. on ITNG. 2006. 161−165. [doi: 10.1109/ITNG.2006.31]

[23] Zhang M, Hall T. Improving the precision of fowler's definitions of bad smells. In: Proc. of the 32nd Int'l Conf. on Annual IEEE Software Engineering Workshop. 2008. 161−166. [doi: 10.1109/SEW.2008.26]

[24] Lozano A, Wermelinger M, Nuseibeh B. Assessing the impact of bad smells using historical information. In: Proc. of the 10th Int'l Conf. on IWPSE. 2007. 31−34. [doi: 10.1145/1294948.1294957]

[25] Murphy-Hill E, Black AP. Seven habits of a highly effective smell detector. In: Proc. of the 1st Int'l Conf. on RSSE. 2008. 36−40. [doi: 10.1145/1454247.1454261]

[26] Chen D, Yuan J, Miao HK. Automatic approach to detect structural smells between class. Computer Engineering, 2007,33(7): 59−61 (in Chinese with English abstract).

附中文参考文献:

[2] Fowler M,著;侯捷,熊节,译.重构:改善既有代码的设计.北京:中国电力出版社,2003.

[13] 张家晨.软件重构方法的研究[博士学位论文].长春:吉林大学,2004.

[14] 王忠杰,徐晓飞,战德臣.面向复用成本优化的构件重构方法.软件学报,2005,16(12):2157−2165. http://www.jos.org.cn/1000-9825/16/2157.htm [doi: 10.1360/jos162157]

[15] 伞云飞,金茂忠,刘超,闫磊.J2EE 应用软件静态模型重构技术研究.计算机应用研究,2007,24(11):196−199.

[16] 王青,伍书剑,李明树.软件缺陷预测技术.软件学报,2008,19(7):1565−1580. http://www.jos.org.cn/1000-9825/19/1565.htm [doi: 10.3724/SP.J.1001.2008.01565]

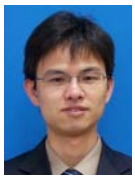
[17] 景涛,江昌海,胡德斌,白成刚,蔡开元.软件关联缺陷的一种检测方法.软件学报,2005,16(1):17−28. http://www.jos.org.cn/1000-9825/16/17.htm

[18] 陈生庆,张立臣,陈广明.面向方面软件重构等价性形式化证明方法.计算机科学,2006,33(7):257−261.

[26] 陈丹,袁捷,缪淮扣.类间结构型代码味道自动检测的研究.计算机工程,2007,33(7):59−61.

高原(1978－),男,陕西汉中人,博士生,工程师,主要研究领域为软件重构,软件测试,软件测评.

牛振东(1968－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为智能信息检索,数字图书馆,软件体系结构,计算模型,数字化教育.



刘辉(1978－),男,博士,副教授,CCF 会员,主要研究领域为软件重构,软件演化与维护,软件测试.

邵维忠(1946－),男,教授,博士生导师,CCF 高级会员,主要研究领域为软件工程,软件工程环境,面向对象技术,软件复用与构件技术.

樊孝忠(1948－),男,教授,博士生导师,主要研究领域为自然语言处理.