软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, [doi: 10.13328/j.cnki.jos.000000]

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

代码坏味对软件演化影响的实证研究*

章晓芳，朱灿

(苏州大学 计算机科学与技术学院,江苏 苏州 215006)

通讯作者：章晓芳, E-mail: xfzhang@suda.edu.cn

摘 要: 代码坏味是指程序设计中存在的不良设计模式或设计缺陷。坏味的存在被认为会阻碍软件的演化与维护。近年来，研究人员致力于探究坏味产生的影响以及坏味与软件演化之间的关系。已有研究表明，代码坏味会随着软件的演化而不断发生变化。通常，软件的演化将涉及源文件的增加、修改与删除这 3 类具体操作，了解代码坏味与软件演化中源文件操作的关系，将有助于开发者更好地计划软件开发过程和重构软件代码。因此，本文针对 13 种常见的坏味，在 8 个 Java 项目，共计 104 个版本中进行了系统的实证研究。研究发现，随着软件版本的演化，含代码坏味的文件在整个项目中的占比在不同的项目中呈现出不同的特征。另外，包含代码坏味的文件更倾向于被修改，而坏味本身与文件的添加或者删除并没有太大的关联。更进一步地，在探究的所有坏味中，有几种特定的坏味对文件的修改产生了显著的影响，且这些坏味文件间存在着明显的重叠。这些发现有助于开发人员更好地了解代码坏味,以便于更好地对软件进行维护。

关键词: 软件维护；反模式；代码坏味；实证研究

中图法分类号:

中文引用格式:
英文引用格式:

An Empirical Study of the Impact of Code Smell on Software Evolution

ZHANG Xiao-Fang, ZHU Can

(School of Computer Science and Technology, Soochow University, Suzhou 215006, China)

Abstract: Code smells refer to poor design patterns or design defects that are considered to have negative impacts on software evolution and maintenance. Many researchers have been devoted into studies on these effects and correlations in recent years. Previous researches indicated that code smells might vary with the evolution of software. In normal cases, the software evolution involves addition, modification and deletion of source files. Therefore, the understanding of the correlations between code smells and software evolution will be helpful for developers in scheduling the development process and in code refactoring. Thus, in this paper, on 8 popular Java projects with 104 released versions, we conduct an extensive empirical study to investigate 13 kinds of code smells. We find that, as the software evolves, the proportion of files that contain code smell in all files reflects different characteristics in different projects. Additionally, the files containing smells are prone to be modified while the smells are not strongly correlated with adding or deleting files. Furthermore, among all the smells we study, some certain ones have significant impact on the file changes and obvious overlap exists in these specific smelly files. These findings are beneficial for developers to achieve in-depth comprehension of code smells, which will lead to better software maintenance.

Key words: software maintenance; anti-pattern; code smell; empirical study

* 基金项目：国家自然科学基金(61772263, 61772014, 61572375)；苏州市科技发展计划（重点产业技术创新—前瞻性应用研究项目 SYG201807）；软件新技术与产业化协同创新中心资助

Foundation item: National Natural Science Foundation of China (61772263, 61772014, 61572375);
Suzhou Technology Development Plan (key industry technology innovation-prospective application research project SYG201807) ;
Collaborative Innovation Center of Novel Software Technology and Industrialization

代码坏味(Code Smell)通常代表不良的程序设计，会对程序理解与软件维护产生不良影响[1]。例如 ClassDataShouldBePrivate 这种坏味，暴露了类中的字段，违反了封装原则。为了减轻坏味对软件开发及演化的影响，近年来，研究者们对代码坏味进行了多方面的研究，包括代码坏味与开发者之间存在的关联[2]、代码坏味的演化[3-5]、坏味对代码理解与维护的影响[5-6] 、坏味与源文件的变化倾向(change-proneness)及出错倾向(fault-proneness)之间的关系[1][7] 等。

为了帮助开发人员识别出存在于不同软件中的代码坏味，研究者们已经设计并实现了多种自动检测坏味的工具。这些工具基于多种分析技术，可以帮助识别不同语言的程序中存在的坏味[8-12]。然而，即使这些工具可以相对准确并充分地检测出多种坏味，大多数的实证研究也仅仅局限在对软件已发布的单个版本进行探究。事实上，软件演化已经被认为是引起软件复杂度逐渐增加的重要因素之一[13-14]，在演化过程中，软件开发人员对源文件的操作则是代码坏味在软件生命周期中演化的根本原因 [15-16]。基于此现状，充分了解软件源文件的操作与代码坏味变化之间的关联将会有助于开发人员提升代码的设计、保证代码的质量。因此，本文系统地开展了一个详细的实证研究，以寻求存在于代码坏味与源文件操作之间的相互关联。具体来说，将源文件的改变细化为 3 类最基本的操作：增加新的文件、修改已有的文件与移除无用的文件。本文使用了广泛应用的坏味检测工具——DÉCOR[17]来检测 13 种最常见的代码坏味，对 8 个 Java 项目共 104 个版本进行了有关源文件操作与代码坏味之间关联性的实证研究。

通过实证研究，本文得到如下结论：首先，随着版本的演化，含有坏味的文件在所有文件中的占比在不同的项目中呈现出不同的变化趋势，而在所有研究项目中含有坏味的文件会发生改变的概率均较低；其次，含有代码坏味的文件在 3 类具体操作中更倾向于被修改，而文件的添加、删除操作则与代码坏味并没有很大的关联；进一步地，如果文件中包含 ComplexClass、LongParameterList 这两种坏味，源文件将会有很大的被修改倾向，另外，AntiSingleton、Blob 与 ClassDataShouldBePrivate 这几种坏味也会在一定程度上对文件的修改产生影响；最后，包含 ComplexClass、LongParameterList 这两种坏味的文件有较大的重叠，且包含这两种坏味的文件有较大的可能性包含其他类型的坏味。

1 背景与相关工作

1.1代码坏味

有关代码坏味的研究一直是软件维护领域的热点之一。Fowler 等人[18]首次提出“坏味”的概念，共描述了 22 种在软件发展与维护过程中存在设计缺陷的特殊代码结构。为了帮助开发者与研究者们识别代码坏味，很多研究人员致力于对坏味进行检测与等级划分[19-23]，并对代码坏味的处理顺序给出了建议[24]。在工具方面，Moha 等人[17]提出的 DÉCOR 工具运用 DETEX 框架自动将坏味的定义转换为坏味检测，该工具已被广泛地运用在已有的代码坏味的研究中[25]。

在坏味检测的过程中，尽管已有多种坏味被定义，但其中有 13 种坏味被认为对软件质量有着重要的影响[26]。因此本文使用 DÉCOR 工具对这 13 种坏味进行检测，表 1 给出这 13 种坏味的具体描述及其简称。

1.2 代码坏味影响的实证研究

为探究代码坏味与软件演化之间的关系，Chatzigeorgiou 等人[27]研究了存在于面向对象程序中坏味的演化，并发现对于大多数的研究实例而言，坏味持久地存在于软件的发展与演化过程中。Olbrich 等人[4]则具体针对 GodClass 和 ShotgunSurgery 这两种坏味进行研究，研究表明坏味的演化存在着不同的阶段、多样的改变频率与不同的改变规模。本文的研究不仅仅关注于坏味随着软件的演化呈现出的变化趋势，而是更深入地探究代码坏味是否与文件的添加、修改、删除这三类最基本的文件操作之间存在着某种程度上的相互关联。

针对代码坏味产生的原因，Tufano 等人[28]进行了一个大规模的实证研究来了解代码坏味何时会被引入以及为什么被引入。该研究通过统计坏味被引入之前源码文件被提交的次数，发现这些坏味通常在源码刚刚被添加到项目中时就被引入。另外，通过分析提交源码的目的、所属项目和开发者当时所处的状态，该研究

Table 1 Description of Code Smells

表 1 代码坏味的描述及其简称

坏味名称（简称）	描述
AntiSingleton(AS)	提供可变类变量的类，往往被用作全局变量。
Blob(Bb)	庞大而复杂的类，它集中了系统某一部分的行为，并仅使用其他类作为数据持有者。
ClassDataShouldBePrivate(CP)	暴露其字段的类，违反了封装原则。
ComplexClass(CC)	该类至少拥有一个具有较大的圈复杂度或是较多代码行的方法。
LargeClass(LC)	就代码行数而言，该类至少含有一个拥有较多代码行的方法。
LazyClass(YC)	具有很少方法和域的类，包含简单的方法。
LongMethod(LM)	就代码行数而言，具有过长方法的类。
LongParameterList(LPL)	具有（至少）一个方法的类，该方法具有超出平均参数数量的长参数列表。
MessageChain(MC)	当实现类的功能需要在不同的类对象之间进行长链方法调用时会发生的情况。
RefuseParentBequest(RPB)	一个使用空的主体改进继承方法的类，破坏多态性。
SpaghettiCode(SC)	一个类声明没有参数和使用全局变量的长方法。这些方法使用复杂的决策算法进行过多的交互，此类不利用和阻止使用多态和继承。
SpeculativeGenerality(SG)	一个定义为抽象但具有极少数子节点，并且这些子节点不使用其方法的类。
SwissArmyKnife(SK)	该类的方法可以划分为多种方法的析取集，从而提供许多不同的不相关的功能。

试图更进一步地探究坏味被引入的原因。研究表明：在软件开发过程中，为增强软件现有的功能或增加新的功能时，开发人员往往更容易引入代码坏味。受到该研究的启发，本文将进一步探究是否所有被添加的源文件都和坏味存在着明显的相互关联。因此，对于每一个研究项目版本，将所有添加进这个版本的文件作为研究对象之一，同时，作为文件基本操作的修改与删除也是本文的重点研究对象。

代码坏味的存在往往给软件的发展与演化带来一定的阻碍。Khomh 等人[26]探究了坏味与代码变更及代码错误倾向之间的相互关联。该研究表明，在几乎所有被研究的项目版本中，包含坏味的源码比不含坏味的源码有着更易发生改变或是发生错误的倾向。同时，该研究也探讨了一些具体类型的坏味是否有更大的倾向发生错误或被修改。在此基础上，Palomba 等人[29]开展了更大规模的实证研究，探究了代码坏味的分布及其对软件维护的影响。与上述研究不同的是，为了更好地了解代码坏味产生的影响，本文将发生改变的源码文件类型更细致地划分为新增、被修改与被移除的文件，从而分别探究这些最基本的文件操作与坏味之间的相互关联。

我们在前期的研究工作中[30-31]，探讨了代码坏味与错误倾向之间的相互关联，在研究过程中发现软件的演化实际上将细化为各个版本之间的文件变更，继而开始研究代码坏味与文件变更之间的关联。在此基础上，本文针对 8 个开源项目的共计 104 个版本，开展了更大规模的实证研究，力求能进一步验证、扩充前期研究的结论并深入探讨代码坏味与软件演化的关系，包括代码坏味与源文件操作的关系，坏味文件之间的相互关系等。

除了上述有关代码坏味对软件自身影响的相关研究，研究者们还关注到开发者们如何应对软件中存在的代码坏味[2]，例如通过开发人员的具体行为来探寻坏味存在的生命周期等[15]。

2 实证研究

2.1 预定义

本文的实证研究致力于在文件级别上探究代码坏味与软件演化之间存在的关联，其中，我们将软件的演化具体为文件的添加、修改与移除操作。为了方便解释，我们首先给出一些相关定义如下。

定义 1 项目$P = \{V_i | i = 1 \ldots n\}$，$V_i$表示项目$P$中的第$i$个发布版本。

定义 2 令存在于第i个版本中而不存在于第$i-1$版本中的文件为版本i的新增文件(added files)，记为A_i；令在版本之间发生修改且同时存在于两版本之间的文件为被修改文件(modified files)，记为M_i；令存在于第i个版本而不存在于第$i+1$个版本中的文件为被移除文件(removed files)，记为R_i。令新增文件、被修改文件与被移除文件的合集为发生改变的文件，记为C_i。

定义 3 第 i 个版本中，令所有包含坏味的文件为坏味文件，记为S_i；令所有不含坏味的文件为非坏味文件，记为N_i；令版本中所有的 Java 文件为$Allfiles_i$。

定义 4 第 i 个版本中，令含有坏味的文件在所有文件中的占比为坏味密度(smell density)，计算方法如下：

$$density_i = \frac{|S_i|}{|Allfiles_i|} \tag{1}$$

定义 5 第 i 个版本中，令含有坏味的文件发生变更的概率为坏味活跃度(smell activity)，计算方法如下：

$$activity_i = \frac{|C_i \cap S_i|}{|S_i|} \tag{2}$$

由定义 4、定义 5 可见，活跃度的值反映了坏味文件在软件演化过程中发生改变的概率，通过计算坏味密度与坏味活跃度的值，将有助于分析坏味对软件演化的影响。

定义 6 第i个版本中 ，令 ChangeType 指代新增、被修改和被移除三种不同的文件变化类型，分别计算发生三种不同类型改变的坏味文件在所有坏味文件中的占比以及发生改变的非坏味文件在所有非坏味文件中的占比，计算方法如下：

$$Correlation(ChangeType, Smelly)_i = \frac{|ChangeType_i \cap S_i|}{|S_i|} \tag{3}$$

$$Correlation(ChangeType, non-Smelly)_i = \frac{|ChangeType_i \cap N_i|}{|N_i|} \tag{4}$$

定义 7 令包含某具体坏味的文件为$smell^m$，不包含此类坏味的文件记为$non-smell^m$。第i个版本中，通过获得包含该种坏味的文件与发生某种变化类型文件的交集，再除以包含此坏味文件的数量，可以得到包含此种坏味的文件发生该变化类型的占比。类似地，可以获得不包含此坏味的文件发生各类变化的占比，计算公式如下：

$$ChangeType(smell^m)_i = \frac{|ChangeType_i \cap smell_i^m|}{|smell_i^m|} \tag{5}$$

$$ChangeType(non-smell^m)_i = \frac{|ChangeType_i \cap non-smell_i^m|}{|non-smell_i^m|} \tag{6}$$

定义 8 第i个版本中，令两种不同坏味文件的重叠率为包含坏味$smell^m$、$smell^n$的文件分别在包含这两种坏味的文件中占比，计算方法如下：

$$Overlap(smell^m, smell^n)_i = \frac{|smell_i^m \cap smell_i^n|}{|smell_i^n|} \tag{7}$$

$$Overlap(smell^n, smell^m)_i = \frac{|smell_i^m \cap smell_i^n|}{|smell_i^m|} \tag{8}$$

2.2 研究问题

代码坏味随着软件项目的演化不断变化，一方面已有研究者表明：新增文件有很大的可能性会引入低质量甚至是错误的代码[28]，因此，随着软件项目规模的扩大，含有代码坏味的文件数量也会随之上升。而另一方面，代码重构也是提升软件项目可维护性的关键因素，在演化的过程中开发者们往往会重构一些不好的程序设计，从而减少了含有代码坏味的文件数量[6]。由此可见，代码坏味的密度可能会随着软件的演化呈现波动的状态。

坏味密度和坏味活跃度可以为开发人员提供重要的信息来做出开发过程中的关键决定，例如他们何时需要重构代码设计以及哪一部分代码需要被重构，又或者通过这些信息来判断软件项目是否达到了成熟阶段等。进一步地，本文分析坏味与文件改变之间的相互关联，有助于为软件项目的维护做出更好的重构计划。此外，不同的坏味可能存在于同一个的文件中，被不同坏味影响的文件是否存在较大的重叠，了解这样的重叠有助于减轻重构时的工作量，实现更为高效的软件维护。为实现这些研究目标，本文深入分析了代码坏味对软件演化的影响，其目的在于回答以下四个研究问题：

(1) 坏味密度与坏味活跃度随着软件项目的演化如何变化？这些变化具有什么样的特征？

(2) 当文件包含代码坏味时，更倾向于发生哪一种变化？文件的变化具体包括文件的新增、修改与删除，了解这些变化与坏味之间的关系是本文的重要目标。

(3) 具体是哪几种代码坏味与文件的变化有着更为显著的关联？在第二个问题的基础上，更进一步地去探究对文件变化产生相对较大影响的坏味类型。

(4) 被不同坏味影响的文件之间是否具有较大的重叠？同一个文件可能包含不同类型的坏味，在维护资源有限的情况下，是否可以仅考虑某几种类型的坏味加以重点维护。

2.3 实验对象

本文主要对代码坏味与软件演化过程中文件不同类型的变化之间存在的关系进行探究，在实验对象选取及数据收集的过程中，需要综合考虑实验对象的多样性与数据的充分性两方面因素。所以，实验对象的选取遵循以下原则：实验对象需要具备不同的规模、属于不同领域，实现了不同的功能，分别由不同的团队进行研发，这样保证了研究对象的多样性；进一步地，这些实验对象需要具有足够多的版本、提交数量以及包含坏味文件的数量，因此优先考虑开发时间较长、发布版本较多的项目，以保证数据的充分性。文件发生改变的提交以及需要被检测的源码文件均可从 Github 上获取，最终选取了 Che、Egit、Jmeter、Xerces2-j、Jgit、Tomcat、Nifi 和 Recommenders 这 8 个开源的 Java 项目进行实证研究，这些实验对象的主要功能及简介如下：

(1) Che 是一个公共云 IDE 平台，可以在公共云、私有云中运行，也可以在任何操作系统上安装。第 1 个版本于 2016 年 2 月 16 日发布，到目前为止共有 115 个版本，最新版本为 6.9.0。

(2) Egit 是一个用于处理 Git 存储库的 Eclipse 插件，已经发布了 102 个版本。第 1 个版本于 2010 年 3 月 19 日发布，最新版本为 v5.0.2-r。

(3) Apache Jmeter 是一个性能测试框架,可用于测试具有不同负载和各种配置下的静态、动态资源和 Web 动态应用程序。第 1 个版本于 2011 年 11 月 2 日发布，到目前为止共有 102 个版本，最新版本为 v4_0。

(4) Xerces2-j 是一个高性能的完全兼容的 XML 解析器，迄今共发布了 98 个版本。第 1 个版本于 1999 年 11 月 9 日发布，最新版本是 Xerces-J_2_12_0。

(5) Jgit 是纯 Java 中 Git 版本控制系统的一个实现。共发布了 107 个版本，第 1 个版本于 2010 年 3 月 19 日发布，最新版本是 v5.0.2-r。

(6) Apache Tomcat 软件是 Java Servlet、JavaServer Pages、Java Expression Language 和 Java WebSocket 技术的开源实现。共发布了 69 个版本，第 1 个版本发布于 2009 年 2 月 28 日，最新版本为 9_0_10。

(7) Apache NiFi 是一个易于使用、功能强大且可靠的系统，用于处理和分发数据。第 1 个版本发布于 2015 年 1 月 23 日，迄今共 63 个版本，最新版本为 rel/nifi-1.7.1。

(8) Recommenders 通过一系列代码分析展示来实现代码推荐功能，共发布了 60 个版本。第 1 个版本发布于 2011 年 2 月 16 日，最新版本为 v2.5.3。

由于这 8 个项目都拥有相对较多的发布版本，本文沿用 Olbrich 等人[4]研究工作中的版本选择策略，每 5 个发布版本选择 1 个作为研究版本。此外，需要过滤掉无法使用项目管理工具自动编译和汇编的初始版本。有关实验对象的相关信息如表 2 所示，其中$|Commit|$表示第一个与最后一个版本之间所有提交的数量，$|V|$表示研究中选择出的版本数量，文件数量范围表示项目中拥有的所有 Java 文件的数量范围。

Table 2 Information of Experiment Subjects

表 2 实验对象的相关信息

项目名称	第一个版本	最新版本	$\|Commit\|$	$\|V\|$	文件数量范围
Che	4.2.3	6.9.0	5535	15	4243-6134
Egit	v1.1.0-rc	v5.0.2-r	3699	15	552-827
Jmeter	v2_5	v4_0	11000	10	730-948
Xerces2-j	Schemapointtag	Xerces-J_2_12_0	5161	18	375-737
Jgit	v0.9.3	v5.0.2-r	5369	19	500-1005
Tomcat	8_0_0_RC	9_0_10	7996	8	1355-1425
Nifi	Parent-1.0.0-incubating-rc	rel/nifi-1.7.1	3705	11	1494-3152
Recommenders	v2.0.4	v2.5.3	1792	8	349-574

2.4 实验流程

图 1 展示了本研究的实验流程。在确定好研究项目后，首先依据选择策略挑选出当前实验版本V_i。接着，根据定义 2 分别获取新增文件A_i与被移除文件R_i。随后，使用工具 DIFF 来获取版本之间（V_i与V_{i+1}之间）的被修改文件M_i；并使用 DÉCOR 区分出每一个实验版本中的坏味文件S_i和非坏味文件N_i。最后，基于上述文件数据，计算并分析坏味文件与新增文件、被修改文件及被移除文件的关系，以了解每一种具体的代码坏味产生的影响。

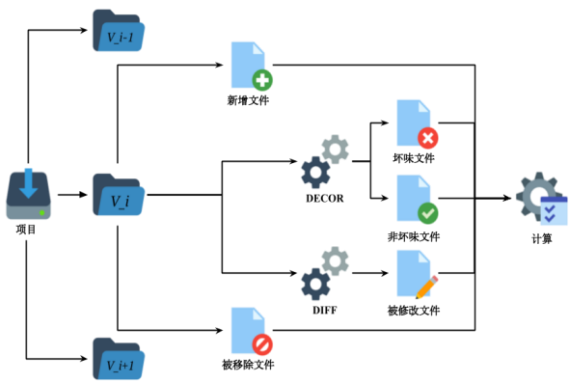


Fig.1 The process of this study

图 1 实验流程图

对于每一个研究的版本i，根据公式 3 计算发生三种不同类型改变的坏味文件在所有坏味文件中的占比，分别计为AS_i、MS_i、RS_i。相应的，根据公式 4 计算发生三种不同类型改变的非坏味文件在所有非坏味文件中的占比，表示为AN_i、MN_i、RN_i。由此，对于三种变化类型，可以分别比较坏味文件与非坏味文件发生这些变化的占比，从而分析出哪一种类型的变化与代码坏味有着显著的关联。

在了解了哪一种文件变化类型与代码坏味更为相关之后，我们进一步探究具体是哪一种或是哪几种坏味对这种变化类型的影响较为明显。根据公式 5 计算得到包含某坏味的文件发生特定类型的改变的占比，类似地，可以根据公式 6 获得不包含此坏味的文件发生各类变化的占比。

通过上述方法，可以得到代码坏味与三种文件改变类型之间的相互关联，以及具体的坏味与这些改变之间的关系。此外，根据公式 7 和公式 8 计算坏味文件之间的重叠率，进一步地探究包含不同坏味的文件之间是否存在较多的重叠，这个探究旨在提供给开发人员更多的信息，以便更为高效地进行代码重构。

2.5 分析方法

本文使用 odds ratio(OR)来评估坏味文件与非坏味文件分别发生三类基本文件操作的倾向。OR 值的计算方式如下所示:

$$OR = \frac{p/(1-p)}{q/(1-p)} \tag{9}$$

其中，p代表了在一组样例中发生某种事件的比例，而q代表了另一组样例中同样事件发生的比例。本研究中，p是指坏味文件发生改变的比例，q则是非坏味文件发生改变的比例。换言之，p值对应了由公式 3 计算得来的AS、MS、RS，而q值则对应公式 4 中的AN、MN、RN。因此，$OR > 1$表示当文件被坏味影响时有较大的可能性会发生这种类型的文件操作，而$OR < 1$则代表了相反的含义。

另外，本文有两处使用了 Wilcoxon 符号秩检验。其一是探究在AS、MS、RS数值之间是否存在较为显著的差异，其二是探究被某一种坏味影响的文件发生改变的比例是否高于未被此种坏味影响的文件。Wilcoxon 符号秩检验是一种非参数检验，因此不要求数据是正态分布的。如果检验得到的 p-value 值低于显著级，则拒绝原假设。

在此基础上，运用以下公式

$$r = \frac{Z}{\sqrt{N}} \tag{10}$$

计算效应值(effect size)，用于体现关联程度的强弱。其中 Z 表示所观察数据的方差，N 表示被观察数据的总个数。根据 Cohen's 效应值的分类，$0.1 \le r < 0.3$表示较小效应，$0.3 \le r < 0.5$表示中等效应，$r > 0.5$则表示具有较大效应。具体的，在本文中，若效应值低于 0.3 则表示该类型坏味与文件改变并无太大关联，若效应值高于 0.5 则表示该类型坏味与文件改变具有较大的关联。

3 实验结果

本节详细展示了 8 个项目的实验结果，并对实验结果做出相关讨论，以回答 4 个研究问题。

3.1 代码坏味密度及活跃度的演化

Table 3 Information of Code Smell Density

表 3 代码坏味密度信息

项目 / density	Che	Egit	Jmeter	Xerces2-j	Jgit	Tomcat	Nifi	Recommenders
density_min	0.182	0.177	0.187	0.530	0.141	0.215	0.119	0.178
density_max	0.307	0.214	0.307	0.610	0.168	0.238	0.164	0.207
density_avg	0.190	0.190	0.234	0.569	0.151	0.219	0.142	0.190

表 3 展示了研究的 8 个项目中坏味密度值的范围和均值。由表可见：代码坏味密度的值在除了 Xerces2-j 中较大外，在其他所有项目中都相对较小。图 2 为 8 个实验项目坏味密度变化的折线图，其中横坐标代表项目的版本，纵坐标代表坏味密度值。

从坏味密度变化的趋势来看，Che、Egit、Jmeter、Tomcat、Recommenders 这 5 个项目的坏味密度呈现较为一致的下降趋势，而另外 3 个项目的坏味密度则呈现出不同幅度的振荡。进一步地，坏味密度下降的 5 个项目也呈现出多种下降特征：Che 和 Egit 这两个项目密度值的下降比较平缓，这是因为这两个项目的总文件数量随着项目的演化在平稳增长；Jmeter 的总文件数量虽是同样呈现持续上升趋势，但是由于在第 5 个版本处含有 LPL 文件的数量忽然大幅下降，导致密度值在此处骤降；至于 Tomcat，在第 2 个版本处总文件数量出现较大增长且包含 AS 的文件数量大幅减少导致密度骤降；而 Recommenders 则是由于在第 4 与第 5 版本处包含文件总数量发生了相对较大的增长，从而导致密度的陡然下降。另一方面，Xerces2-j、Jgit 和 Nifi 这 3 个坏味密度振荡的项目包含文件的总数量则是随着版本的演化持续波动，并非持续增长的。

基于坏味密度呈现出不同变化的趋势，单纯从密度值难以得到关于坏味演化的一致结论，因此进一步引入坏味活跃度来反映包含坏味的文件发生变化的占比来探究坏味对文件改变的影响。表 4 给出代码坏味活跃度的范围和均值。

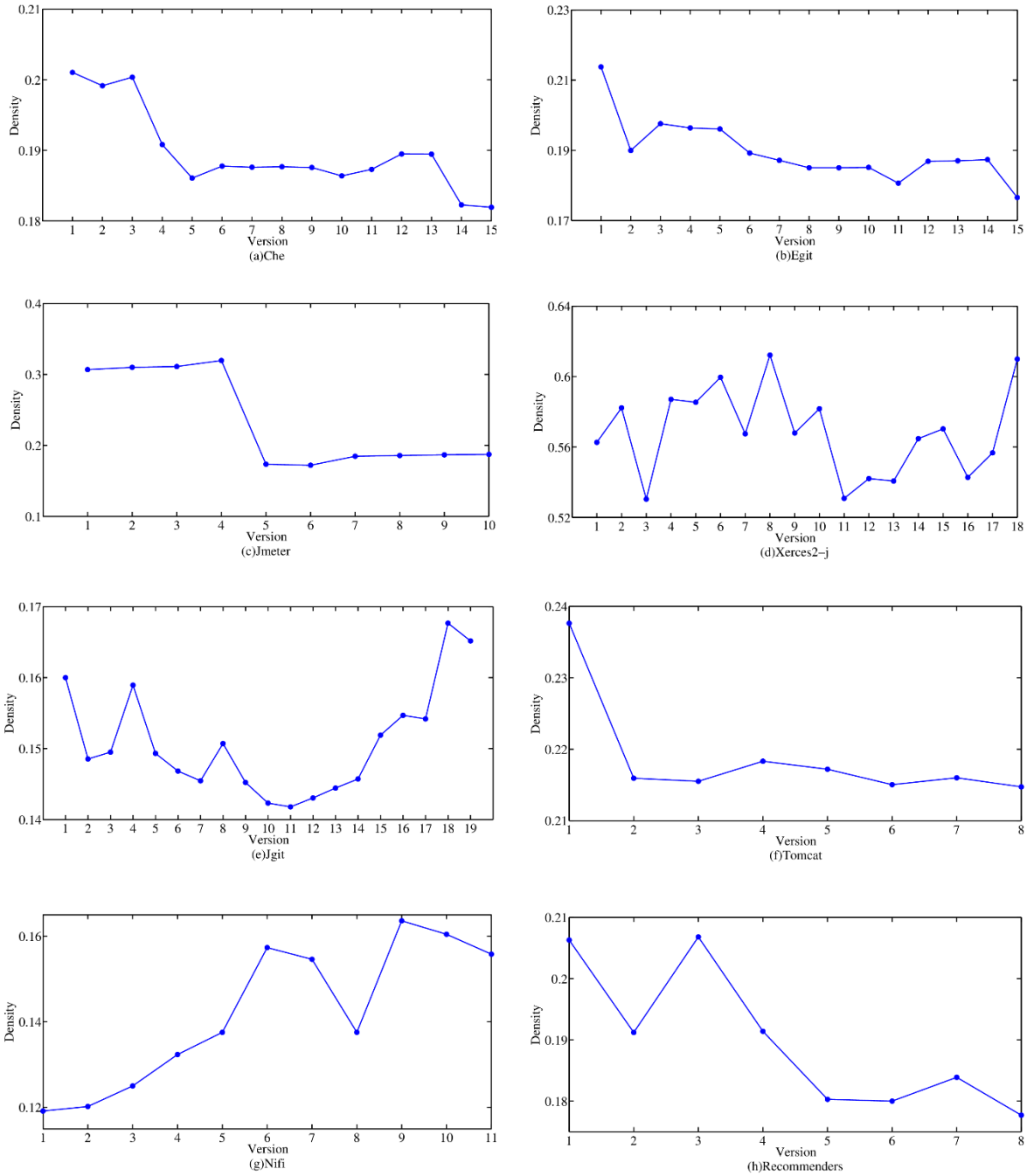


Fig.2 The evolution of code smell density

图 2　代码坏味密度的演化

Table 4 Information of Code Smell Activity

表 4 代码坏味活跃度信息

项目 / activity	Che	Egit	Jmeter	Xerces2-j	Jgit	Tomcat	Nifi	Recommenders
activity_min	0.068	0.168	0.197	0.213	0.178	0.125	0.337	0.286
activity_max	0.301	0.483	0.671	0.982	0.933	0.904	0.618	0.875
activity_avg	0.175	0.343	0.426	0.523	0.477	0.400	0.466	0.560

由表 4 可见，绝大多数项目坏味活跃度的均值低于 0.5，即在大多数情况下，只有一半不到的坏味文件将发生文件变更。其中，活跃度均值最高的 Recommenders 项目，是由于其前期版本中代码活跃度较高造成的，其后期版本中活跃度明显下降。结合坏味密度和活跃度发现，在坏味密度呈现下降的 5 个项目中，坏味活跃度也呈现出下降的趋势。

综上所述，坏味密度的变化呈现出了多种趋势，而坏味活跃度在大多项目中低于 0.5。由此，我们推测不同的项目处于不同的开发阶段，处于非成熟阶段的项目将引发诸多基础功能的改变，从而呈现不稳定的坏味密度与活跃度的演化。即使坏味密度持续下降，往往并非是因为开发人员对坏味存在的关注或是限制，而是因为总的文件数量的变化。因此，我们需要更深入地探究坏味对文件改变的具体影响。

3.2 代码坏味与不同文件变化类型的相互关联

图 3 展示了 8 个项目有关代码坏味与不同文件变化类型之间相互关联的盒图。盒图的横坐标是根据公式 3 和公式 4 计算得到的结果，分别记为AS、MS、RS、AN、MN、RN，纵坐标表示坏味文件与非坏味文件中发生 3 类不同具体操作的占比。

从图 3 可知，在所有的研究项目中，MS具有最高的中值(median value)，接着依次是MN、AN、AS、RN、RS。单纯从数值分布的角度看，无法得到一致的结论，比如MS在某些项目中具有更为稳定的分布，而在另一些项目中则情况相反。至于MS的数值范围，Che 具有最小的数值范围，坏味文件发生修改的占比均小于 0.25，而 Jgit 具有最高数值范围，最高占比大于 0.9，其次是 Xerces2-j。

从占比数值分布的角度来看，不同的研究项目具有不同的特征，但总体而言，我们有如下发现：在AS、MS、RS几个数值中，MS具有最高中值，而几乎所有RS的中值最低；考虑非坏味文件的改变，AN、MN、RN中，MN也高于AN与RN。

为了进一步探究AS、MS、RS的数值分布之间是否具有显著差异，使用 Wilcoxon 符号秩检验进行评估。设置原假设如下：

(1) H_{01}：AS值与MS值之间没有显著性差异。

(2) H_{02}：MS值与RS值之间没有显著性差异。

(3) H_{03}：AS值与RS值之间没有显著性差异。

表 5 展示了检验得到的 p-value 值，其中 A-B 表示 A、B 两组数值对比得到的 p-value 值，如 AS-MS 表示 H_{01} 的 p-value 值。若 p-value 值小于 0.05，则拒绝原假设。从表 5 可知：对于 H_{01} 与 H_{02}，所有 AS-MS、MS-RS 的 p-value 值均小于 0.05，这意味着MS分别与AS、RS之间存在显著差异。因此，拒绝假设 H_{01} 与 H_{02}。对于 H_{03}，所有 p-value 值都大于 0.05，不能拒绝假设 H_{03}，这表示AS与RS 之间不存在显著差异。

Table 5 P-values for comparison of AS, MS and RS

表 5 AS、MS 和 RS 的对比 p-value 值

项目 / 对比项	Che	Egit	Jmeter	Xerces2-j	Jgit	Tomcat	Nifi	Recommenders
AS-MS	0.002	0.001	0.007	0.022	0.000	0.012	0.026	0.025
MS-RS	0.003	0.001	0.005	0.006	0.000	0.012	0.040	0.012
AS-RS	0.080	0.328	0.051	0.078	0.053	0.500	0.091	0.128

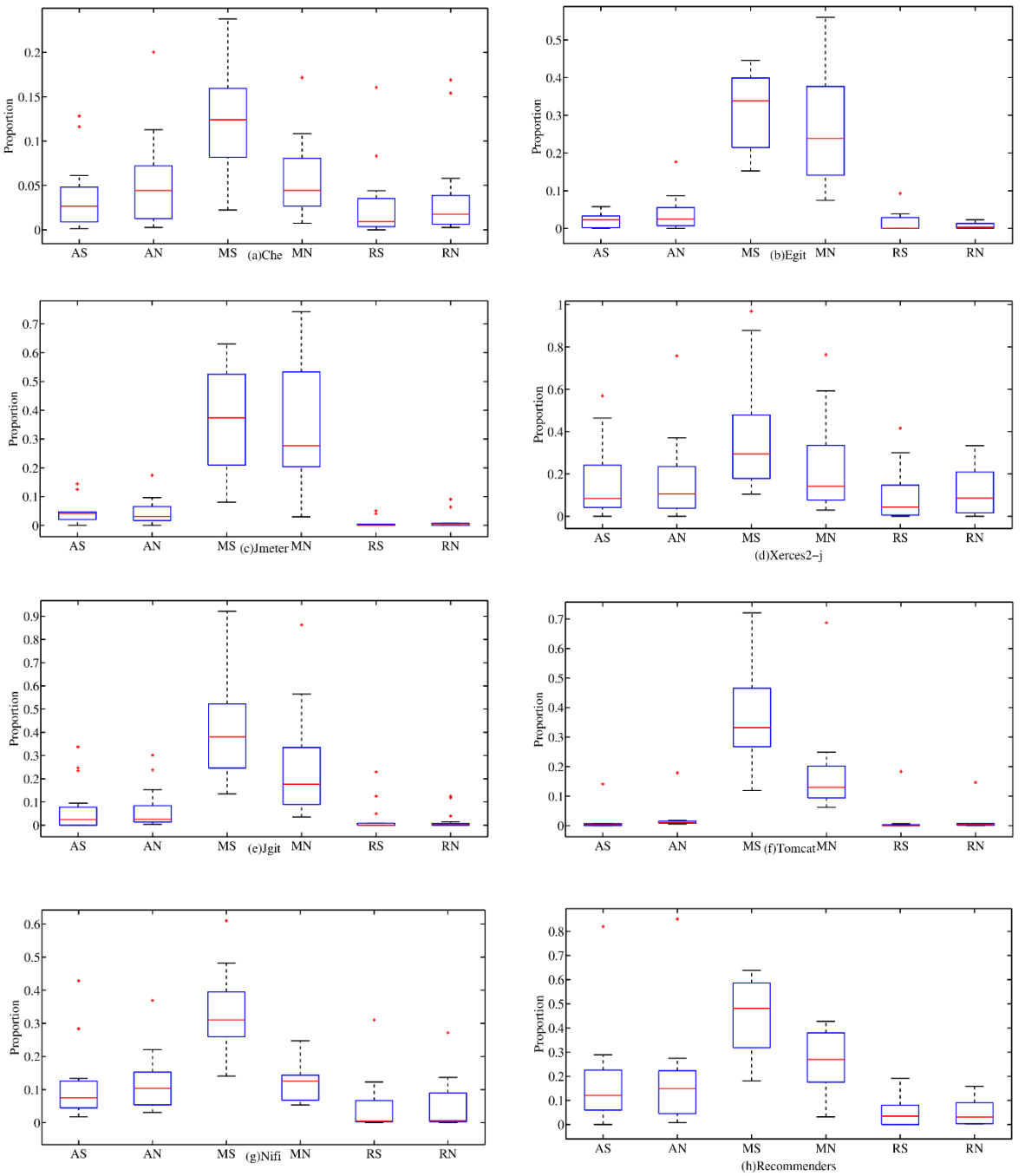


Fig.3 The proportion of different change types in smelly and non-smelly files

图 3　坏味文件与非坏味文件发生不同变化的占比

Table 6 Summary of Change ORs

表 6 OR 值汇总

	Che			Egit			Jmeter			Xerces2-j		
	A	M	R	A	M	R	A	M	R	A	M	R
$\#(OR>1)$	1	15	4	3	11	7	4	7	0	10	16	5
$\#(OR<1)$	14	0	11	12	4	8	6	3	10	8	2	13

	Jgit			Tomcat			Nifi			Recommenders		
	A	M	R	A	M	R	A	M	R	A	M	R
$\#(OR>1)$	5	19	5	0	8	2	6	11	4	2	8	4
$\#(OR>1)$	14	0	14	8	0	6	5	0	7	6	0	4

进一步地,利用 OR 值来评估坏味文件与非坏味文件发生改变的倾向。表 6 汇总了各个项目 OR>1 和 OR<1 的数量，其中 A、M、R 分别表示新增、被修改与被移除文件的 OR。表中统计了每组 OR>1 和 OR<1 的数量，分别记为#(OR>1)、#(OR<1)。

由表 6 可知：对于新增文件的 OR 值，除了 Xerces2-j 和 Nifi 的所有其他项目，#(OR<1)高于#(OR>1)。然而，Xerces2-j 中#(OR>1)为 10，对应#(OR<1)为 8，Nifi 中#(OR>1)为 6，对应#(OR<1)为 5，这两个项目#(OR>1)与#(OR<1)十分接近。对于被修改文件的 OR 值，在所有项目中，#(OR>1)高于#(OR<1)。对于被移除文件的 OR 值，除了 Recommenders 项目中存在#(OR>1)等于#(OR<1)外，在其他所有项目中有：#(OR<1)高于#(OR>1)。

OR 值体现了代码坏味与文件的具体操作类型之间的关联，其中被修改文件的 OR 值大于 1 则代表了包含坏味的文件更倾向于被修改。根据上述实验结果与分析，我们可以得出以下结论：坏味文件更倾向于被修改，而代码坏味与文件的新增或是被移除没有很大的关联。

上述实证研究结果证实了 Khomh 等人[26]关于坏味文件更具有改变倾向、错误倾向的结论，进一步得出了坏味文件更倾向于被修改的结论，并从更大规模的项目实证再次证实了我们前期的研究成果[31]。坏味文件更倾向于被修改的现象提醒开发人员，应在开发过程中重视代码坏味的检测和及时消除，以减少代码坏味对后续的开发与维护的不良影响。

3.3 特定类型的代码坏味与文件变化的相互关联

基于 3.2 节中代码坏味与文件修改更为相关的结果，本节进一步探究具体是哪一种或是哪几种坏味对文件修改产生较大的影响。因此，分别计算包含某一种坏味的文件与不包含这种坏味的文件发生修改的占比。

在对坏味进行检测的过程中发现，并非所有的 13 种坏味都存在于各个项目中。但能够检测到的坏味大多随着版本的演化一直存在于项目中，即这些坏味存在于每个项目的所有版本中。为了保证研究的可靠性，本文仅对存在于该项目所有版本中的代码坏味进行研究。这是由于要探究特定类型的坏味与文件修改之间的相互关联，若是其中某一版本中不存在受某种坏味影响的文件，不仅实验结果会产生误差，同时也因为不存在该坏味而缺失了研究的意义。

图 4 展示了被特定类型坏味影响的文件与未被该坏味影响的文件发生修改的占比。图 4 中使用了坏味名称缩写以便更加清晰地表示，如 Bb 表示 Modified(Blob)，non-Bb 表示 Modified(non- Blob)。基于图 4 的信息，表 7 汇总了特定类型坏味与文件修改的关联信息，其中“√”表示该种坏味在对应项目中表现出与文件修改有较大关联，“×”则表示关联度较小，“--”表示未检测到此种坏味。

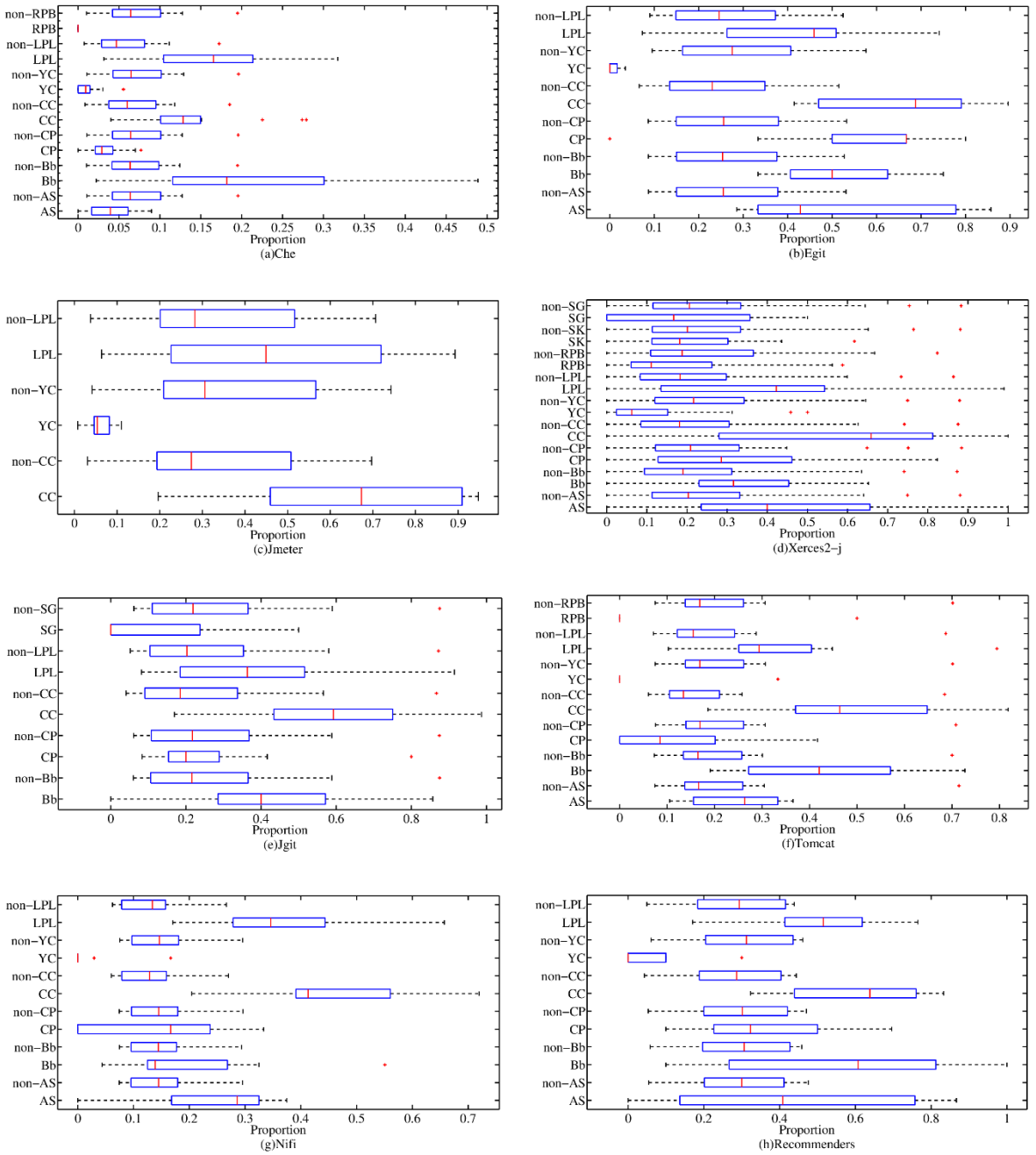


Fig.4 The proportion of modified files in certain smell files and the files that not contain the smell

图 4　包含特定类型坏味的文件与不包含该坏味的文件被修改的占比

Table 7 Correlations Between Code Smell and File Modification

表 7 坏味与文件修改的关联

项目 / 坏味类型	Che	Egit	Jmeter	Xerces2-j	Jgit	Tomcat	Nifi	Recommenders
AntiSingleton(AS)	×	√	--	√	--	√	√	√
Blob(Bb)	√	√	--	√	√	√	×	√
ClassDataShouldBePrivate(CP)	×	√	--	√	×	×	√	√
ComplexClass(CC)	√	√	√	√	√	√	√	√
LazyClass(YC)	×	×	×	×	--	×	×	×
LongParameterList(LPL)	√	√	√	√	√	√	√	√
RefuseParentBequest(RPB)	×	--	--	×	--	×	--	--
SpeculativeGenerality(SG)	--	--	--	×	×	--	--	--
SwissArmyKnife(SK)	--	--	--	×	--	--	--	--

从图 4、表 7 中可以看出，所有项目中都包含被 CC 与 LPL 影响的文件，并且这些文件都有较大的被修改倾向。相反的，YC 与文件修改不存在显著性关联。至于 AS、Bb 和 CP 这几种坏味，它们在不同的项目中对文件修改产生的影响则不尽相同：在 Che、Egit、Xerces2-j、Jgit、Tomcat、Recommenders 中包含 Bb 的文件有较大可能性被修改，而在 Nifi 中 Bb 并不呈现这样的特征；AS 在 Che 中对文件修改的影响不大，而在 Egit、Xerces2-j、Tomcat、Nifi、Recommenders 中对文件的修改影响较大；Che、Jgit、Tomcat 中被 CP 影响的文件发生修改的可能性不大，而另一些项目中被影响的文件有较大的可能性被修改。

进一步地，我们使用 Wilcoxon 符号秩检验来探究包含特定类型坏味的文件是否与文件修改有着较强的相互关联。对于每一种特定类型的代码坏味，设置原假设 H_{04}：包含该种坏味的文件发生修改的占比与不包含该坏味的文件发生修改的占比不存在显著性差异。若检验得到的 p-value 值小于 0.05，将否定原假设，并进一步计算效应值。表 8 展示了具体的 p-value 及效应值，当 p-value 值大于 0.05 时，对应数值加“*”表示。

如表 8 所示，有 5 种代码坏味需要引起我们的关注。具体来说，CC 与 LPL 这 2 种坏味，在所有 8 个项目中，均有 p-value 值小于 0.05，效应值大于 0.5，充分说明这两种坏味会对文件的修改产生较大的影响。AS、Bb、CP 则并非存在于所有的项目中，且对不同项目中文件修改的影响也不尽相同。例如，AS 在 Tomcat 和 Recommenders 中的 p-value 值大于 0.05，表示该坏味在这两个项目中与文件修改的关联度不大，而在另外的项目中则显示出较强的关联。类似地，Bb 在 Xerces2-j 和 Nifi 这 2 个项目中，CP 在 Recommenders 中的 p-value 值均大于 0.05。

Table 8 P-values and Effect Size

表 8 P-value 值与效应值

坏味类型 / 项目	AS		Bb		CP		CC		LPL	
	p-value	ES	p-value	ES	p-value	ES	p-value	ES	p-value	ES
Che			0.001	0.601			0.001	0.622	0.001	0.622
Egit	0.020	0.566	0.001	0.594	0.011	0.467	0.001	0.622	0.001	0.615
Jmeter							0.005	0.615	0.005	0.615
Xerces2-j	0.001	0.594	0.085*	--	0.020	0.410	0.000	0.622	0.000	0.601
Jgit			0.005	0.450			0.000	0.620	0.000	0.568
Tomcat	0.327*	--	0.012	0.630			0.012	0.630	0.012	0.630
Nifi	0.033	0.455	0.859*	--			0.003	0.626	0.003	0.626
Recommenders	0.327*	--	0.012	0.630	0.327*	--	0.012	0.630	0.012	0.630

通过分析上述 5 种代码坏味与文件修改之间的关联，我们发现 CC、LPL 与文件的修改之间存在较大的关联。另外我们仍需多加关注 Blob，该坏味存在于除了 Jmeter 以外的所有项目中，且在多个项目中还具有较大的效应值，对文件修改的影响也相对较大。AS 与 CP 若是存在于项目中，也需要引起一定的关注。

3.4 坏味文件的重叠

Table 9 Overlaps Between Specific Smelly Files

表 9 不同坏味文件的重叠率

项目	Che	Egit	Jmeter	Xerces2-j	Jgit	Tomcat	Nifi	Recommnders
AS_and_Bb_in_AS				0.112				
AS_and_Bb_in_Bb								
AS_and_CP_in_AS	0.924	0.660		0.225		0.113	0.466	0.883
AS_and_CP_in_CP	0.775	0.956		0.157		0.139	0.680	0.566
AS_and_CC_in_AS	0.338	0.387		0.216		0.262	0.170	0.239
AS_and_CC_in_CC				0.107				0.139
AS_and_LPL_in_AS				0.339		0.257	0.181	
AS_and_LPL_in_LPL								
Bb_and_CP_in_Bb								0.466
Bb_and_CP_in_CP								0.125
Bb_and_CC_in_Bb	0.407	0.500		0.299	0.403	0.573	0.345	
Bb_and_CC_in_CC				0.491			0.132	
Bb_and_LPL_in_Bb	0.270	0.300		0.334	0.163	0.161		0.260
Bb_and_LPL_in_LPL				0.230				
CP_and_CC_in_CP	0.299	0.562			0.182	0.300	0.294	0.187
CP_and_CC_in_CC								0.168
CP_and_LPL_in_CP				0.283		0.114	0.167	
CP_and_LPL_in_LPL								
CC_and_LPL_in_CC	0.355	0.165	0.337	0.693	0.236	0.254	0.328	0.280
CC_and_LPL_in_LPL	0.167	0.173	0.135	0.296	0.241	0.260	0.234	0.234

基于 3.3 节中坏味与文件修改之间关联度的分析，本节计算受到 5 种特定类型代码坏味(AS, Bb, CP, CC, LPL)影响的共同文件在各自坏味文件中的占比。为了便于观察与分析，表 9 仅列举各个项目中根据公式 7 和公式 8 计算得到的重叠率均值大于 0.1 的情况。

从表 9 可知，受 CC 与 LPL 影响的文件在所有项目中的均值都高于 0.1，其中同时包含这两种坏味的文件在 CC 中的占比均值最低为 0.165，最高为 0.693；在 LPL 中的占比均值最低为 0.135，最高为 0.296。这表明 CC 与 LPL 这两种坏味存在着共生现象，往往同时存在于同一项目中，且这两种坏味文件有一定程度的重叠。这些现象与这两种坏味的定义相一致，这两种坏味分别代表了具有复杂圈复杂度、较多代码行数和长参数列表的代码片段。类似地，AS 与 CP 在 6 个项目中存在着共生现象，值得注意的是，这两种坏味文件的重叠程度很大，在 AS 中的占比最高为 0.924，在 CP 中的占比最高为 0.956。这表明，项目中很大比例的文件同时受到 AS 与 CP 这两种坏味的影响。

进一步深入分析坏味文件的数量，可以发现：虽然 CC 与 LPL 的重叠率整体上并不是很高，但在所有项目中，被 CC 与 LPL 同时影响的文件数量较多，例如 Jgit 中同时包含这两种坏味的文件数量在各个版本中的均值为 13，而其余坏味文件的重叠数量均值都低于 3。对于 AS 与 CP 这两种坏味，除了 Che 中同时被这两种坏味影响的文件数量大于 65 以外，其他项目中同时被这两种坏味影响的文件数量较少，最高仅为 19。在少量重叠文件而高重叠率的情况下，同时对 AS 与 CP 这两种坏味进行代码重构，消除坏味的影响将事半功倍。

此外，被 CC 与其他坏味共同影响的文件往往占据其他坏味文件较大的比例，例如 AS_and_CC_in_AS 在 6 个项目中重叠率高于 0.1，而 AS_and_CC_in_ CC 仅在 2 个项目中重叠率高于 0.1。这说明 CC 坏味文件数量往往大于 AS 坏味文件数量。除 CC 之外，LPL 也具有类似的特征。因此，坏味文件数量较高的 CC 与 LPL 应该在重构时优先考虑，以提高重构效率。

通过上述观察可知：在代码重构过程中，应综合考虑存在共生现象的 CC 与 LPL、AS 与 CP 坏味文件，力求共同消除，特别地，应该提高 CC、LPL 这两种坏味文件的维护优先级。此外，AS、Bb 和 CP 这几种坏味也不可忽视。

3.5 讨论

存在于各个项目中不同的坏味密度与活跃度的特征促使我们探究坏味与文件改变之间的相互关联。正如

代码坏味的相关定义，坏味是指会影响代码结构与可理解性的不友好片段，我们的探究表明坏味的存在会导致文件内容发生修改，而不会影响到整个项目的基本框架。我们推测，开发者们在开发过程中更注重实现功能需求而忽视坏味的存在，往往会给后续的开发与维护带来一定的困难。从另一个角度来看，如果坏味问题得到有效的解决，那么将大大减少由坏味引起的文件修改的成本。

根据文件修改与具体坏味之间的关联与包含不同坏味文件之间的重叠率，我们的研究提供给开发者有关包含坏味文件维护的有效建议，如提升包含 ComplexClass、LongParameterList 这两种坏味文件的维护优先级、维护时可同时考虑消除 AntiSingleton 与 ClassDataShouldBePrivate 这两种坏味的影响等，以实现更为高效的代码维护。

和多数的实证研究一样，本文的工作也存在以下内部与外部因素对可靠性的影响：

(1) 内部影响：其中一个主要威胁是实验环境的准确性。首先，我们使用 DÉCOR 检测代码坏味，代码坏味的定义本身具有一定的主观性，这可能是一个对本研究可靠性的潜在的威胁。但是，作为一个被广泛应用的代码坏味自动检测工具，DÉCOR 检测坏味的召回率为 100% ，平均准确率为 60%，因此我们认为由 DÉCOR 检测的坏味是可以用于该实证研究的。另外，我们对实验环境进行了仔细的检查与测试以保证实验的有效性。

(2) 外部影响：我们仅对 8 个 Java 项目进行了实证研究。这些项目具有相对较大的规模且并应用于多个领域，并且我们共计探究了 104 个项目版本，从这些版本中得到的数据足够支持我们的研究。我们相信，这样的实验规模足以保证该研究的有效性。然而，对更大的系统进行进一步验证可以帮助推广我们的发现。此外，我们的实验是在文件级别进行的，将来我们可以将实验设置到类级别，以探索源代码和坏味之间更精确的相关性。

4 总 结

本文系统地开展了关于代码坏味对软件演化的影响的实证研究，分析了代码坏味与源文件变更之间的相关性。文件的变更细化为新增文件、修改文件和移除文件这三类具体操作，目的是探究这些不同的操作类型是否均与坏味相关。本文针对 8 个 Java 项目的 104 个版本，运用 DÉCOR 工具检测 13 种不同类型的坏味，实验结果表明：

(1) 代码坏味的密度和活跃度在不同的项目中呈现不同的特征，大部分项目的坏味密度随着软件的不断演化呈现下降趋势，且坏味活跃度通常不高。

(2) 与不含坏味的文件相比，含有坏味的文件更容易被修改。此外，代码坏味与文件的添加和移除没有明显的关联。

(3) ComplexClass 和 LongParameterList 这两类坏味与文件的修改更为相关，此外，如果项目中存在 AntiSingleton，Blob 和 ClassDataShouldBePrivate 这些坏味，仍然需要重视它们产生的影响。

(4) 软件维护的过程中可考虑 AntiSingleton 和 ClassDataShouldBePrivate 这两种坏味的特征同时进行重构；包含 ComplexClass 和 LongParameterList 这两种坏味的文件有较大的可能性包含其他坏味，且被这两种坏味影响的文件重叠率较高、重叠文件数量较多，应提高这两种坏味的重构优先级。

本文采用大规模的实证研究深入分析了代码坏味对软件演化的影响，并为开发人员在软件维护过程中如何有效地重构代码给出了相关建议：开发人员应该更多地关注包含代码气味的文件，尤其是包含 ComplexClass 或 LongParameterList 的文件，避免引入这些坏味将大大降低文件被修改的可能性，从而降低整个软件生命周期中维护成本。在未来的工作中，将分析更多的项目，并在更为细化的级别上进行深入的实证研究。此外，拟引入人的因素，如熟悉度、中心性和所有权等，以讨论代码坏味产生的影响。

References:

[1] T. Hall, M. Zhang, D. Bowes, and Y. Sun, “Some code smells have a significant but small effect on faults,” ACM Trans. Softw. Eng. Methodol., vol. 23, pp. 33:1–33:39, Sept. 2014.

[2] A. Yamashita and L. Moonen, “Do developers care about code smells? An exploratory survey,” in 2013 20th Working Conference on Reverse Engineering (WCRE), pp. 242–251, Oct 2013.

[3] L. Aversano, G. Canfora, L. Cerulo, C. Del Grosso, and M. Di Penta, “An empirical study on the evolution of design patterns,” in Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ESEC-FSE ’07, (New York, NY, USA), pp. 385– 394, ACM, 2007.

[4] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, “The evolution and impact of code smells: A case study of two open source systems,” in Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement, ESEM ’09, (Washington, DC, USA), pp. 390–400, IEEE Computer Society, 2009.

[5] S. M. Olbrich, D. S. Cruzes, and D. I. K. Sjoberg, “Are all code smells harmful? A study of god classes and brain classes in the evolution of three open source systems,” in 2010 IEEE International Conference on Software Maintenance, pp. 1–10, Sept 2010.

[6] A. Yamashita and L. Moonen, “Exploring the impact of inter-smell relations on software maintainability: An empirical study,” in 2013 35th International Conference on Software Engineering (ICSE), pp. 682–691, May 2013.

[7] W. Li and R. Shatnawi, “An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution,” Journal of Systems and Software, vol. 80, no. 7, pp. 1120 – 1128, 2007. Dynamic Resource Management in Distributed Real-Time Systems.

[8] R. Marinescu, “Detecting design flaws via metrics in object-oriented systems,” in Proceedings of 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems. TOOLS 39, pp. 173–182, 2001.

[9] R. Marinescu, “Detection strategies: metrics-based rules for detecting design flaws,” in Proceedings of 20th IEEE International Conference on Software Maintenance, 2004, pp. 350–359, Sept 2004.

[10] R. Marinescu, “Measurement and quality in object-oriented design,” in 21st IEEE International Conference on Software Maintenance (ICSM’05), pp. 701–704, Sept 2005.

[11] A. M. Fard and A. Mesbah, “Jsnose: Detecting javascript code smells,” in 2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 116–125, Sept 2013.

[12] J. Schumacher, N. Zazworka, F. Shull, C. Seaman, and M. Shaw, “Building empirical support for automated code smell detection,” in Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM ’10, (New York, NY, USA), pp. 8:1–8:10, ACM, 2010.

[13] G. Xie, J. Chen, and I. Neamtiu, “Towards a better understanding of software evolution: An empirical study on open source software,” in Software Maintenance, 2009. ICSM 2009. IEEE International Conference on, pp. 51–60, IEEE, 2009.

[14] I. Neamtiu, G. Xie, and J. Chen, “Towards a better understanding of software evolution: an empirical study on open-source software,” Journal of Software: Evolution and Process, vol. 25, no. 3, pp. 193–218, 2013.

[15] R. Peters and A. Zaidman, “Evaluating the lifespan of code smells using software repository mining,” in 2012 16th European Conference on Software Maintenance and Reengineering, pp. 411–416, March 2012.

[16] R. Arcoverde, A. Garcia, and E. Figueiredo, “Understanding the longevity of code smells: Preliminary results of an explanatory survey,” in Proceedings of the 4th Workshop on Refactoring Tools, WRT ’11, (New York, NY, USA), pp. 33–36, ACM, 2011.

[17] N. Moha, Y. G. Gueheneuc, L. Duchien, and A. F. L. Meur, “Decor: A method for the specification and detection of code and design smells,” IEEE Transactions on Software Engineering, vol. 36, pp. 20–36, Jan 2010.

[18] M. Fowler and K. Beck, “Refactoring: improving the design of existing code,” Lecture Notes in Computer Science, vol. 2418, p. 256, 1999.

[19] F. Khomh, S. Vaucher, Y. G. Guhneuc, and H. Sahraoui, “A bayesian approach for the detection of code and design smells,” in 2009 Ninth International Conference on Quality Software, pp. 305–314, Aug 2009.

[20] R. Oliveto, F. Khomh, G. Antoniol, and Y. G. Gueheneuc, “Numerical signatures of antipatterns: An approach based on b-splines,” in 2010 14th European Conference on Software Maintenance and Reengineering, pp. 248–251, March 2010.

[21] F. Palomba, A. Panichella, A. D. Lucia, R. Oliveto, and A. Zaidman, “A textual-based technique for smell detection,” in 2016 IEEE 24th International Conference on Program Comprehension (ICPC), pp. 1– 10, May 2016.

[22] N. Tsantalis and A. Chatzigeorgiou, “Identification of move method refactoring opportunities,” IEEE Transactions on Software Engineering, vol. 35, pp. 347–367, May 2009.

[23] Sheng JF, Hu PP, Wang B, “Survey of research on anti-pattern detection,” Application Research of Computers, 2013, 30(12):3525-3528. (in Chinese).

[24] 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.

[25] W. Ma, L. Chen, Y. Zhou, B. Xu, and X. Zhou, “Are anti-patterns coupled? An empirical study,” in 2015 IEEE International Conference on Software Quality, Reliability and Security, pp. 242–251, Aug 2015.

[26] F. Khomh, M. D. Penta, Y. G. Gueheneuc, and G. Antoniol, “An exploratory study of the impact of antipatterns on class change-and fault-proneness,” Empirical Software Engineering, vol. 17, no. 3, pp. 243– 275, 2012.

[27] A. Chatzigeorgiou and A. Manakos, “Investigating the evolution of bad smells in object-oriented code,” in 2010 Seventh International Conference on the Quality of Information and Communications Technology, pp. 106–115, Sept 2010.

[28] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. D. Penta, A. D. Lucia, and D. Poshyvanyk, “When and why your code starts to smell bad,” vol. 1, pp. 403–414, May 2015.

[29] F. Palomba, G. Bavota, M. D. Penta, F. Fasano, R. Oliveto, and A. D. Lucia, “On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation,” Empirical Software Engineering, pp. 1–34, Aug 2017.

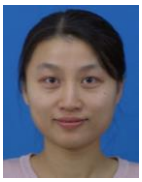
[30] X. Zhang, Y. Zhou and C. Zhu, “An Empirical Study of the Impact of Bad Designs on Defect Proneness,” 2017 International Conference on Software Analysis, Testing and Evolution (SATE), Harbin, 2017, pp. 1-9.

[31] C. Zhu, X. Zhang, Y. Feng and L. Chen, “An Empirical Study of the Impact of Code Smell on File Changes, ” 2018 IEEE International Conference on Software Quality, Reliability and Security (QRS), Lisbon, 2018, pp. 238-248.

附中文参考文献:

[23] 盛津芳,胡培培,王斌.反模式检测研究综述[J].计算机应用研究,2013,30(12):3525-3528.

[24] 高原,刘辉,樊孝忠,牛振东,邵雄忠.代码坏味的处理顺序.软件学报, 2012,23(8): 1965−1977.



章晓芳 (1980－),女,福建连江人,博士,副教授,CCF 会员,主要研究领域为软件分析与测试,众包软件工程,强化学习.

朱灿 (1995－),女,江苏通州人,硕士生,主要研究领域为软件分析与测试,代码坏味.