软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2023,34(1):150−170 [doi: 10.13328/j.cnki.jos.006431]

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

代码坏味研究综述*

田迎晨[1], 李柯君[1], 王太明[1], 焦青青[1], 李光杰[2], 张宇霞[1], 刘 辉[1]

[1](北京理工大学 计算机学院, 北京 100081)

[2](军事科学院 国防科技创新研究院 协同创新项目部, 北京 100097)

通信作者: 李光杰, E-mail: liguangjie_er@126.com; 刘辉, Email: Liuhui08@bit.edu.cn

摘 要: 代码坏味 (code smells) 是低质量的急需重构的代码片段. 代码坏味是软件工程领域的一个研究热点, 并且相关研究方向众多、时间跨度大、研究成果丰富. 为梳理相关研究思路和研究成果、分析研究热点并预判未来研究方向, 对 1990 年至 2020 年 6 月间发表的代码坏味相关的 339 篇论文进行了系统地分析和归类, 对代码坏味的发展趋势进行了分析与统计, 量化揭示了相关研究的主流与热点. 揭示了学术界关注的关键代码坏味, 并研究了工业界与学术界的关注点的差异及其影响.

关键词: 代码坏味; 软件重构; 软件质量; 度量; 缺陷

中图法分类号: TP311

中文引用格式: 田迎晨, 李柯君, 王太明, 焦青青, 李光杰, 张宇霞, 刘辉. 代码坏味研究综述. 软件学报, 2023, 34(1): 150–170. http://www.jos.org.cn/1000-9825/6431.htm

英文引用格式: Tian YC, Li KJ, Wang TM, Jiao QQ, Li GJ, Zhang YX, Liu H. Survey on Code Smells. Ruan Jian Xue Bao/Journal of Software, 2023, 34(1): 150–170 (in Chinese). http://www.jos.org.cn/1000-9825/6431.htm

Survey on Code Smells

TIAN Ying-Chen[1], LI Ke-Jun[1], WANG Tai-Ming[1], JIAO Qing-Qing[1], LI Guang-Jie[2], ZHANG Yu-Xia[1], LIU Hui[1]

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

[2](Management Department of Collaborative Innovation, National Innovation Institute of Defense Technology, Academy of Military Sciences, Beijing 100097, China)

Abstract: Code smells are low-quality code snippets that are in urgent need of refactoring. Code smell is a research hotspot in software engineering, with many related research topics, large time span, and rich research results. To sort out the relevant research approach and results, analyze the research hotspots, and predict the future research directions, this study systematically analyzes and classifies 339 papers related to code smell published from 1990 to June 2020. The development trend of code smells is analyzed and counted, the mainstream and hot spots of related research are quantitatively revealed, the key code smells concerned by the academia are identified, and also the differences of concerns between industry and academia are studied.

Key words: code smells; software refactoring; software quality; metric; defect

代码坏味 (code smells) 是 Fowler 等人[1]于 1999 年提出的一个概念, 用于表示低质量的急需重构的软件代码片段. 由于受到交付日期的压力或者开发人员的疏漏等因素的影响, 软件开发和演化过程中有可能引入代码坏味, 从而降低了代码的可理解性和可维护性. 为此, 研究人员提出了各种各样的检测/处理代码坏味的方法和工具[2−6], 辅助程序员发现并清除代码坏味. 代码坏味的自动/半自动检测是软件工程领域的一个研究热点. Fard 等人[7]提出了一种基于软件度量指标和代码静态/动态分析的代码坏味检测算法; Fakhoury 等人[8]、Liu 等人[4]以及卜依

* 基金项目: 国家自然科学基金重大项目 (61690205); 国家自然科学基金面上项目 (61772071)
收稿时间: 2020-12-31; 修改时间: 2021-03-29; 采用时间: 2021-08-10; jos 在线出版时间: 2021-10-20
CNKI 网络首发时间: 2022-11-15

凡等人[9]相继提出了基于深度神经网络的代码坏味检测方法. 此外, 研究人员还将决策树[10,11]、遗传算法[12−14]、多目标优化[15,16]、启发式[17−19]等方法应用于代码坏味的检测当中以改善检测效果. 为了深入了解代码坏味的影响, 学术界从开发人员、软件系统类型及其他多个角度, 分析论证了代码坏味对软件可维护性[20,21]、软件缺陷[22−24]、软件安全性[25,26]以及软件能耗[27−29]等软件质量的各个方面的影响. Catolino 等人[30]分析了代码坏味与代码更改之间的关系, 将代码坏味严重性信息添加到代码更改预测模型当中, 提高了对代码变化倾向的预测能力. 研究人员还进一步研究了代码坏味之间的关系, 分析了他们的相关性以及共现现象等[31−33]. 此外, 研究人员还深入分析了导致代码坏味引入的原因[34−36]、代码坏味在不同类型系统中的分布情况[37−39]以及如何自动重构代码坏味以消除其影响[5,6,40,41]等.

关于代码坏味的研究方向众多、时间跨度大、研究成果丰富. 为梳理相关研究成果和研究思路, 更好地了解和掌握研究现状, 研究人员多次尝试对相关的研究进行文献综述. Zhang 等人[42]对 2000 年至 2009 年 6 月期间发表的 39 篇与文献 [1] 总结的 22 种代码坏味的相关研究论文进行了分析, 发现重复代码坏味是最受研究人员关注的代码坏味. Zhang 等人[42]认为代码坏味影响方面的研究目前尚有不足. de Paulo Sobrinho 等人[43]分析了 1990 年至 2017 年 4 月间发表的 351 篇代码坏味相关的研究论文, 从代码坏味的类型、研究人员对坏味的关注点演变、实验设置、对坏味研究的人员/团队以及论文出版分布等 5 个角度进行了研究. Sharma 和 Spinellis[44]收集了 1999－2016 年间发表的 445 篇研究论文, 从代码坏味的定义、类型、出现原因、检测方法以及影响 5 个方面进行概述.

相关综述论文对该领域的研究进行较为深入的分析和梳理, 也给出了很多有价值的发现和研究建议. 但是学术界对于代码坏味的研究持续发展, Liu 等人[4]首先尝试使用深度学习技术检测坏味, 之后学术界进一步推进了深度学习在其他坏味检测上的应用. 此外学术界提出了众多新的代码坏味拓展了坏味目录 (如 Python 坏味[45], 持续集成反模式[46]和 iOS 性能反模式[47]等), 并且涌现出很多新的重构解决方案, 如利用模拟退火搜索策略做重构推荐[6]等, 这些研究进一步拓展和推动了人们对代码坏味的理解. 然而由于现有的文献综述研究论文发表时间较早, 没能覆盖最近几年日新月异的研究进展. 此外, 这些综述论文各有侧重点, 但都未能解决关键代码坏味遴选、工业界应用状况分析、特点方向发展趋势等关键问题. 为此, 本文针对所有代码坏味相关的研究论文进行全面的分析和梳理, 重点关注以下 4 个研究问题.

RQ1: 人们是否持续提出新的代码坏味? 新的代码坏味主要关注哪些领域/问题?

RQ2: 代码坏味相关研究的目的可以包含哪些类型? 哪些类型的研究是目前的主流和热点?

RQ3: 学术界重点关注哪些代码坏味?

RQ4: 工业界重点关注哪些代码坏味? 工业界的关注点是否与学术界一致?

RQ1 关注于代码坏味的定义, 分析各种代码坏味被首次提出的时间, 对新定义代码坏味及其发展趋势进行分析; RQ2 关注代码坏味研究的不同侧面 (如代码坏味的定义、代码坏味的检测以及代码坏味的清理等), 量化揭示本领域内的研究主流和研究热点; RQ3 则从众多代码坏味中辨识学术界的研究重点, 发掘若干关键代码坏味, 为后续研究以及工业应用聚焦提供依据; RQ4 重点关注工业界的应用情况, 比较学术界与工业界的关注点, 分析学术研究与工业应用的适配情况, 以更好地指导相关研究与软件质量保障工作的进行.

本文第 1 节介绍了系统地进行文献综述的方法. 第 2 节、第 3 节和第 4 节介绍研究结果并介绍相关结论. 第 5 节总结了本文的研究结果, 并对未来代码坏味的研究方向给出了建议. 第 6 节对全文进行了总结.

1 综述方法

本次综述按照 Kitchenham[48]的系统文献综述的标准步骤进行. 具体综述方法包括搜索范围与关键词、纳入排除规则、人工审查标准等方面.

1.1 搜索范围和关键词

代码坏味是 Fowler 等人在 1999 年提出的一个术语[1], 但是在此之前已经存在类似问题的研究. 比如 Malvean 等

人[49]于 1998 年在其著作中使用反模式 (antipattern) 一词表示违反设计模式的情况, 并且提供了对应的解决方案. 因此为了尽可能涵盖所有与代码坏味相关的论文, 本文将搜索的具体时间范围确定为 1990 年至 2020 年 6 月.

考虑到文献数据库的影响力、所收录论文的领域以及在计算机领域的权威性, 本次综述选择 ACM digital library、IEEE Xplore、Springer link、Google Scholar、Web of Science、Science Direct、中国知网、万方数据库、中国科技论文在线、中图链接等文献数据库, 对国内外针对代码坏味研究的相关文献进行检索.

对于以上选定的文献数据库, 本次综述使用统一的关键词进行文献搜索. 首先根据以往文献检索的经验, 以及通过阅读以往相关综述文章, 总结出普遍用于代码坏味相关论文搜索的关键词. 然后确定搜索关键词的替代拼写以及同义词, 例如找到 anti-pattern 的替代拼写 antipattern, code smell 的同义词 bad smell 等. 最后考虑每个搜索关键词的复数形式, 以实现更全面的覆盖范围. 在得到用于搜索的关键词后, 使用或运算符 (OR) 来连接每个关键词构成对数据库的搜索规则:

(1) 对外文文献数据库的搜索规则: “code smell” OR “bad code smell” OR “bad smell” OR “antipattern” OR “design smell” OR “design flaw” OR “design defect” OR “anti-pattern” OR “code smells” OR “bad code smells” OR “bad smells” OR “antipatterns” OR “design smells” OR “design flaws” OR “design defects” OR “anti-patterns”.

(2) 对中文文献数据库的搜索规则: “代码坏味” OR “反模式” OR “坏味” OR “设计坏味” OR “设计反模式”.

不同的文献数据库在搜索时可使用的粒度不同, 例如 IEEE 数据库中允许用户选择不同的位置搜索关键词, 包括文章标题、作者关键字以及摘要等, 并且不同位置的检测条件可以相互组合. 而 ACM 数据库中只能选择一个位置检测相应关键词. 因此针对不同的数据库需要调整检测规则, 并且尽可能使用计算机科学或软件工程之类的过滤器来优化搜索结果. 在不同的文献数据库上应用搜索规则检索后, 得到的数据库中相关的文献数量如图 1 所示.

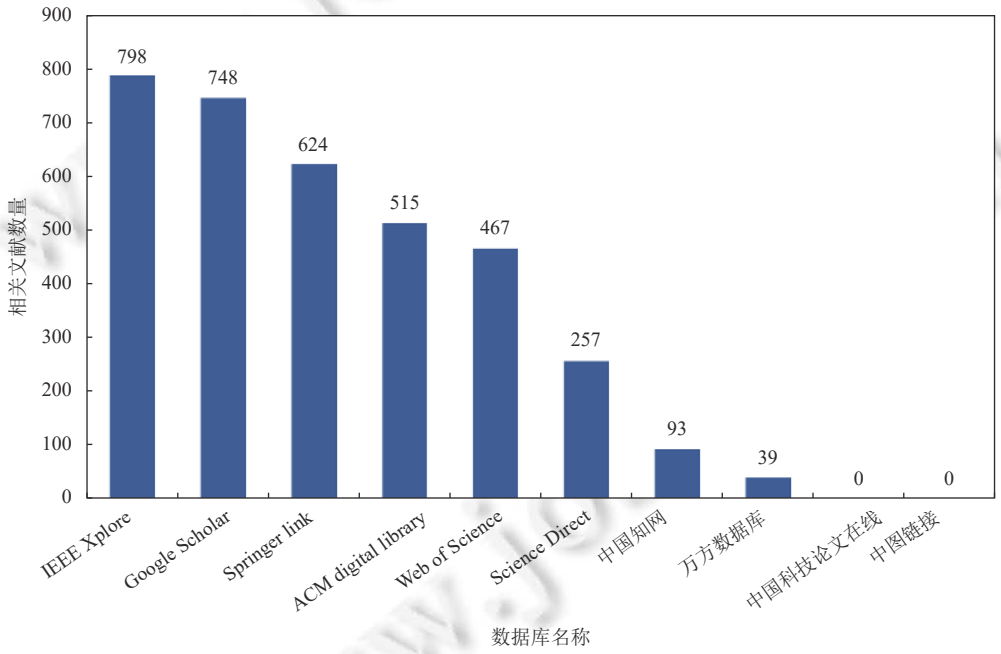

图 1 文献数据库的搜索结果

1.2 纳入排除规则

对于得到的搜索结果需由人工分析论文标题、摘要以及关键字等内容, 并应用纳入排除规则进行论文筛选, 得到需要进一步研究的论文数据集. 此外, 代码克隆的概念及其检测方法的研究远早于代码坏味以及软件重构等概念[50]. 代码克隆的检测自成体系, 其检测方法与其他代码坏味 (比如上帝类、长方法等) 有显著差异. 并且代码

克隆检测相关论文的数量远超其他代码坏味, 纳入代码克隆将导致整个综述论文偏向某一个特殊的代码坏味 (代码克隆). 因此本研究排除只研究代码克隆的论文.

由于搜索规则中使用到 bad smell、bad smells 关键字, 在搜索结果中可能存在非软件工程, 如医学、化学等领域的论文, 因此需要应用纳入规则判断论文是否属于软件工程领域. 首先判断论文是否发表在 CCF 软件工程领域推荐刊物列表中. 若是, 则纳入. 对于发表在非 CCF 软件工程领域推荐刊物列表中的论文, 由两名小组成员根据标题、摘要以及关键字, 对其是否属于软件工程领域进行判断. 如果判断未能达成一致, 则引入第三方协商讨论最终达成一致. 具体的纳入和排除规则如下.

纳入规则: ① 软件工程领域的论文; ② 以代码坏味为主要研究对象的研究工作.

排除规则: ① 文献不是英文或中文的; ② 文献无法通过数字方式获取; ③ 文献少于两页 (摘要); ④ 只对代码克隆坏味进行研究的论文; ⑤ 文献为学位论文.

为了避免遗漏任何与代码坏味相关的研究, 本研究在该步骤应用滚雪球方法[51], 选择论文参考文献中未被数据集包含的论文来扩充研究论文数据集. 由于初始数据集过大, 在具体实施时只考虑了综述性论文的参考文献, 并且只进行一次滚雪球操作, 最终得到了 1 129 篇论文作为下一阶段筛选的初始数据集.

1.3 人工审查

由于纳入排除规则仅考虑了标题、摘要以及关键字内容, 初始数据集中可能包含仅涉及代码坏味但并未深入研究的论文. 因此, 在得到初步的研究论文数据集之后, 将对数据集中的论文进行人工阅读审查, 并从中提取和概述回答研究问题所需要的信息. 此过程由 2 名博士研究生和 3 名硕士研究生组成的小组对论文进行审查, 对每篇论文除了基本信息之外还需要考虑如下方面.

(1) 复查. 进一步审查论文研究内容与结论, 按照 ① 代码坏味是文章的主要研究对象, 而不是研究问题的相关因素, ② 从软件实现而非软件管理等方面研究反模式, ③ 论文至少讨论一个代码坏味 (反模式) 这 3 个规则复查判断论文是否为代码坏味相关研究. 每篇论文有两名成员分别标注, 添加一致认同的论文, 并对标注结果不一致的论文进行讨论, 在无法达成一致结论时引入第三方仲裁, 达成最终一致. 最终得到 339 篇相关研究, 进行接下来的数据收集.

(2) 论文发表的期刊或者会议在中国计算机协会 (CCF) 推荐国际学术刊物中对应的等级;

(3) 论文的研究目的. 记录文章对代码坏味研究的目的, 包括检测、重构、研究影响等. 对研究目的分类是在论文人工审查过程中增量添加, 而非预先制定;

(4) 论文的核心观点. 阅读论文后对论文核心观点的总结摘要;

(5) 论文中涉及的代码坏味及作者对代码坏味的分类.

经过人工审查从 1 129 篇论文中筛选出 339 篇论文构成了本综述的研究数据集, 每篇论文都包括相应的基本信息、核心观点等分析数据. 完整的论文数据收集执行流程如图 2 所示.

图 2 论文收集过程

1.4 代码坏味检测工具

为了分析实际项目开发中开发人员对代码坏味的关注点. 本研究从实际开发中使用的静态代码检测工具入手, 获取工具支持的代码坏味, 进而从侧面反映开发人员的关注热点. 为了使结果能更加如实地反映真实的开发情况, 被研究的工具需满足以下条件: ① 工具是商用的; ② 工具的网址可访问; ③ 工具仍能使用. 本研究一共搜索到了 6 种用于实际项目开发过程中的商业代码质量检测工具, 构成了所要研究的工具数据集.

1.5 论文数据集

所有 339 篇论文都以代码坏味为主要研究对象, 并且通过上述排除规则剔除了只针对代码克隆 (code clone)

的研究, 但是仍然包含代码克隆 (或者含义相同的重复代码-Duplicated Code 代码坏味) 与其他坏味一起被研究的论文. 在 339 篇论文中有 121 (35.69%) 篇论文已被现有综述研究调研, 全部综述文献数据的详细信息请参阅在线文档[52]. 图 3 展示了从首次出现代码坏味的相关研究至 2020 年, 每年发表的相关研究的数量, 并且依照研究发表刊物所属的 CCF 等级, 对每年发表的研究进行区分. 由于本研究论文数据只统计到 2020 年 6 月, 因此 2020 年对应的论文数仅代表部分数据.

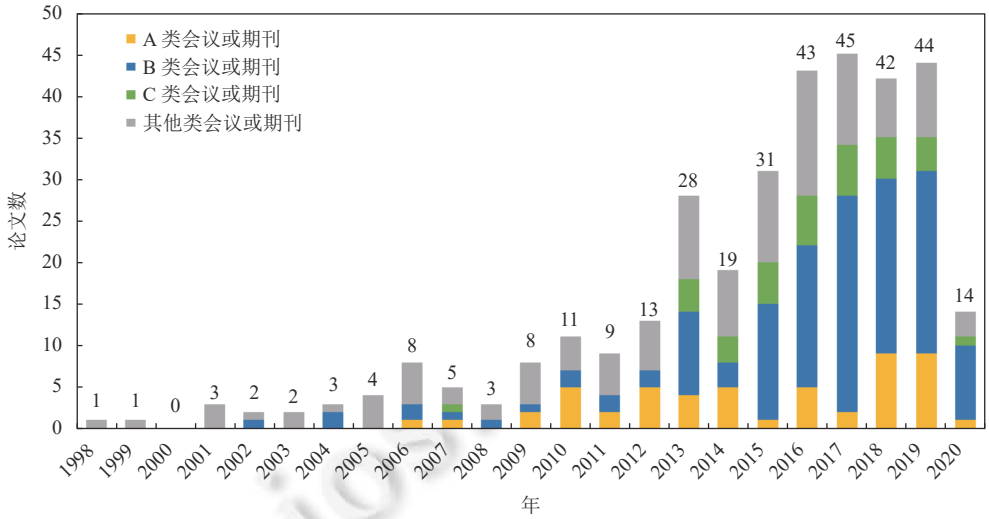

图 3 每年发表论文数

图 3 的统计结果显示, 代码坏味是一个持续的研究热点. 针对代码坏味的研究开始于 1998 年, 至 2012 年学术界针对代码坏味的研究数量增加缓慢. 自 2012 年之后研究人员对于代码坏味的关注程度显著提高, 并且相关研究数量也迅速增加. 在 2017 年相关研究数量达到顶峰后, 每年发表论文数量也一直稳定在峰值水平. 分析论文发表刊物的分区等级后发现, 代码坏味相关的高水平研究成果逐渐增多, 发表在 B 类以上刊物的论文数量, 以及在每年相关论文中的比重呈上升趋势, 其中发表在 B 类及以上刊物中的论文占到全部论文数的 55.46%. 结果表明, 近几年研究人员对于代码坏味的研究兴趣并未降低, 并且在最近几年保持着较高的关注度, 相关论文数量总体也呈上升趋势, 相关研究成果得到了学术界广泛认可.

2 RQ1: 趋势分析

本节主要从发表论文数以及代码坏味数的角度, 回答了代码坏味的持续演化趋势以及关注点的变化.

反模式指在错误的位置应用了特定的设计模式而造成负面后果的问题解决方案[49], 从较高的层次上指示了软件质量缺陷, 可以认为代码坏味的不同形式. 因此在本文中统一使用代码坏味一词指代不同的代码坏味和反模式. 本研究统计了每年发表的论文中新定义代码坏味的数量, 由于存在名称不同但实质相同的坏味 (异名同义), 在得到初始代码坏味目录之后, 本研究对异名同义的坏味进行了合并. 为了减少主观性, 合并过程由两名熟悉代码坏味定义的作者独立进行, 根据代码坏味的定义对本质相同的坏味进行标记. 对于分歧, 引入一名有代码坏味研究经验且熟悉本研究的同事作为第三方仲裁, 协商讨论最终达成一致意见. 结果如图 4 所示.

结果表明, 自 1998 年至 2004 年在代码坏味被提出和讨论的初始时期, 定义新代码坏味的热情逐渐下降, 直至 2004 年发表的 3 篇论文中只新定义了 1 种代码坏味. 而在之后的一段时间里, 对代码坏味的研究主要集中在已有的代码坏味上, 新定义代码坏味的数量一直保持在较低的水平. 而对该阶段的论文进一步分析发现, 对于代码坏味的研究主要专注于面向对象代码, 未将代码坏味的概念引申到其他领域. 只有 Spadini 等人[23]在 2001 年将代码坏味的概念引入测试代码中, 定义了 11 种新的测试代码坏味. Fowler 等人[1]和 Malveau 等人[49]针对面向对象代码

定义的代码坏味的完备程度, 是导致该时期出现代码坏味数量下降的主要原因. 其次由于对代码坏味的影响是否值得重构存在疑惑[42], 以及尚未认识到代码坏味存在于软件开发的各个阶段, 以至于人们并未关注到其他领域/对象中存在的代码坏味. 随着研究的深入, 直至 2013 年研究人员对代码坏味进行拓展并完善代码坏味列表的兴趣骤然上升, 虽然受每年发表论文数的影响, 每年研究人员定义新的代码坏味的数量有所波动, 例如 2013–2015 年新定义代码坏味数有所下降. 但是其总体呈上升趋势, 有关代码坏味的研究将在第 3.1 节详细介绍.

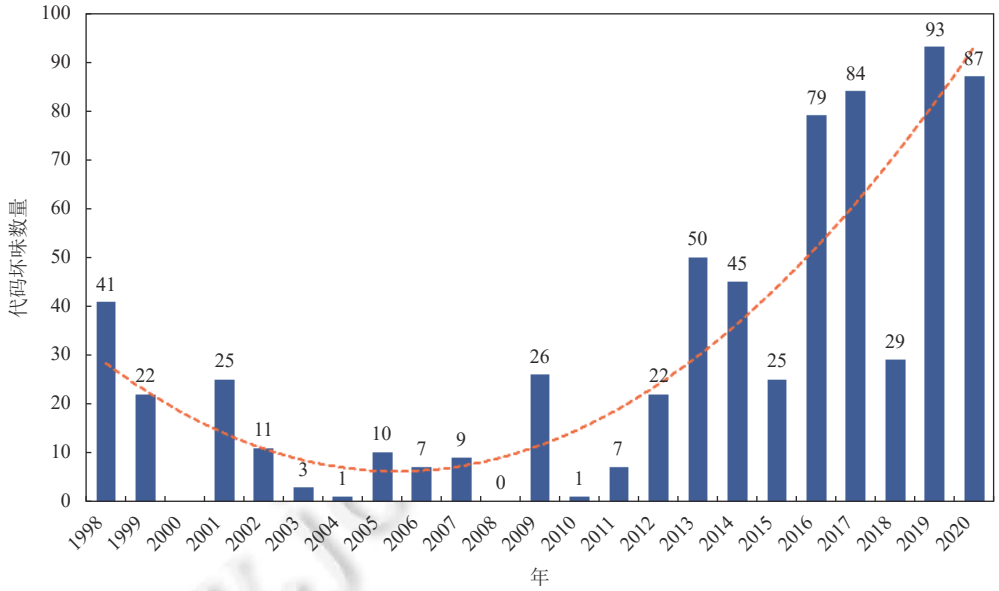

图 4 每年出现代码坏味数

现有的讨论研究了许多不同领域和针对不同对象的代码坏味. 本研究重点关注并分析代码坏味的拓展, 同时总结各种代码坏味的目录, 如安卓/iOS 应用程序[53,54]、测试[55]及 SQL[56]等代码坏味. 完备的代码坏味类型及相关坏味的目录对于代码坏味的研究有指导意义, 并且能够帮助开发人员优化代码质量. 但有关代码坏味类型的研究, 尚未形成一致的分类标准. Ganesh 等人根据代码坏味对面向对象设计原则的违背, 对出现的代码坏味进行了分类[57], 但是这种方法仅考虑了面向对象的设计原则, 无法包含其他非面向对象或更高粒度的代码坏味. Mäntylä 等人根据经验研究提出了一种基于对软件开发影响的代码坏味分类方法, 即将代码坏味分为耦合类、面向对象特征滥用、膨胀类等[58], 但是由于是一种主观的分类方法, 不同的研究人员可能得到不同的分类结果. 此外还有其他较为复杂或者针对某一方面的分类方法, 未得到广泛的认可和应用[3,59].

因此在统计过程中本研究并未对代码坏味进行分类, 只是复用相关代码坏味原作者给定的坏味类型. 同时由于没有统一标准, 不同作者定义的类型并不正交, 因此存在一个坏味属于不同类型的情况, 其次对于未指明特定适用领域或者对象的代码坏味, 默认将其分到代码具体实现过程中可能出现的代码坏味, 即实现代码坏味 (implementation code smell) 类型经过收集、分类、推断和统计, 本研究得到了以下 38 个不同的代码坏味类型, 表 1 给出了目录和对应的解释.

同时本研究收录整理了以上类型的代码坏味目录, 并统计了每种代码坏味的相关研究论文数. 表 2 中相关研究论文数指所有论文数据集中以该代码坏味为研究主体内容 (之一) 的论文数量. 由于文章空间的限制, 这里只提供了在论文中出现次数排在前十的代码坏味, 简要目录如表 2 所示.

通过对该研究问题的讨论和回答, 本研究得到了完整的代码坏味及其对应原始坏味类型目录, 详见在线文档[52]. 可以看出代码坏味已经从只代表面向对象代码中的潜在质量缺陷, 不断引申拓展到众多领域、适用不同的研究对象. 以往的研究证明代码坏味能出现在不同领域软件开发的众多阶段, 因此不止面向对象软件开发人员, 其他领域以及不同软件开发阶段的开发人员都应该及早重视并对代码坏味采取相应措施, 以保障软件质量.

表 1 代码坏味类型

代码坏味类型	介绍	代码坏味类型	介绍
Android	安卓代码坏味[53]	iOS Performance	iOS性能相关的代码坏味[47]
Android ICC Security	安卓组件间通信安全代码坏味[60]	JavaScript	JS代码坏味[7]
Android Presentation Layer	安卓表示层代码坏味[61]	Linguistic	语言代码坏味[62]
Android Security	安卓安全相关的代码坏味[26]	Logging Code*	日志代码反模式[63]
Anti-Micro-Patterns	反微模式[64]	Memory*	内存反模式[65]
Application Usability	应用程序可用性坏味[66]	Mobile App*	移动App反模式[67]
Architecture	基于组件的软件架构坏味[68]	MVC	MVC模式的代码坏味[69]
Aspect-oriented	面向方面的代码坏味[70]	Open Cloud Computing Interface*	开放云计算接口的反模式[71]
CI	持续集成代码坏味[46]	ORM performance*	面向对象映射性能反模式[72]
Communication Performance*	通信性能反模式[73]	Performance*	性能相关的反模式[74]
Design	设计代码坏味[75]	Python	Python代码坏味[45]
Design Configuration	项目配置信息的设计代码坏味[76]	Python Security	Python安全代码坏味[77]
Duplicate Log	日志中的重复代码坏味[78]	Security	安全相关代码坏味[79]
Embedded Code Smell	Web服务器端嵌入其他语言代码的代码坏味[80]	Semantic	访问控制模型语义反模式[81]
Energy	能耗代码坏味[82]	SOA*	面向服务的架构反模式[83]
Exception Handling*	异常处理反模式[84]	Software Reuse*	软件重用反模式[85]
Implementation	代码实现中的代码坏味[1]	SQL*	SQL反模式[56]
Implementation Configuration	项目配置信息的实现代码坏味[76]	Test	测试代码坏味[86]
iOS	iOS代码坏味[54]	Variability-Aware	预处理器可变性感知坏味[87]

注: *代表反模式类型的代码坏味

表 2 被研究次数在前十的代码坏味

代码坏味	坏味定义	相关研究论文数	占全部论文百分比 (%)
Feature envy[1]	当一个方法对其他类的兴趣高过对自己所处类的兴趣, 判定为该坏味.	133	39.2
Long method[1]	当方法的代码行数过长时, 出现该坏味	113	33
God class[88]	在一个好的面向对象的设计中, 系统的职能是均匀分布在顶级类中的. God class坏味违背了这一设计, 把太多的职能集中在一个类中, 导致该类负责过多的功能	95	28
Data class[1]	Data class是指一个类仅用来存储数据, 及用于访问这些数据的方法, 而没有其他的数据处理功能	85	25.1
Duplicated code[1]	在项目中由于粘贴复制、重复编码等一系列原因出现重复的代码时, 则出现Duplicated code坏味	79	23.3
Refused bequest[1]	当子类继承了父类的方法和数据, 但是却很少使用甚至不使用这些父类中定义的属性和方法, 意味着对象的继承关系设计错误	79	23.3
Blob class[49]	Blob class处理很多职责, 其属性和方法与不同的概念和过程相关, 有着很低的内聚力, 并且经常与许多数据类相关联	78	23
Shotgun surgery[1]	当只为了添加一个新的或扩展的行为片段而必须在很多的地方 (尤其是不同的类) 做小修改时, 这种情况被认为是该坏味	74	21.8
Long parameter list[1]	当一个方法具有很长的参数列表时, 就会出现该坏味	73	21.5
Spaghetti code[49]	该坏味指非结构化的难以理解和维护的代码. 这类代码包含很多复杂的控制语句来 (比如goto) 进行代码的逻辑控制, 而不是使用结构的代码 (方法、结构体) 来进行编写. 如果软件使用面向对象的语言进行开发, 而开发代码使用面向过程的方式来编写, 亦可导致本坏味	63	18.6

总体而言, 对每年发表的相关论文以及新定义代码坏味的统计结果表明, 自代码坏味提出至今, 研究人员对于代码坏味的关注程度呈稳步增长的趋势, 并且在最近 4 年一直保持着较高的关注度. 除了对已有面向对象代码坏味进行研究外, 讨论代码坏味在其他领域或者对象上的拓展, 并且定义新的代码坏味已经成为新的研究热点. 虽然代码坏味的不断引申拓展丰富了整个目录, 但由于缺少一种广泛认可的分类方法, 导致了代码坏味的类型冗繁, 增加了开发人员查找并识别感兴趣代码坏味的难度. 因此定义一种完备、一致且准确的代码坏味分类方法也是该领域需要进一步研究的问题.

3 研究热点与关键代码坏味

本节旨在对以往代码坏味相关研究的目的进行分析, 完成 RQ3 对研究目的以及趋势的回答. 同时根据收集到的信息, 量化代码坏味的关键程度, 找到学术界广泛认可的代码坏味, 完成对于 RQ4 的回答.

3.1 RQ2: 研究热点与趋势

研究中使用主题分析[89]提取并确定最终的研究目的类型. 主要包括以下几个步骤: ① 在人工审查阶段需要小组成员阅读研究内容, 了解研究工作的目的和贡献; ② 生成初始研究目的类型. 在这个阶段, 要求小组成员在对研究有完整了解的情况下, 给出概括研究目的的初始描述信息; ③ 汇总并分组. 对于上一步中得到的初始类型信息进行汇总, 并分成概念上的相似性的初始类型; ④ 合并和修改. 我们对初始类型进行了修改及合并, 保证类型之间的正交; ⑤ 最后得到最终的 10 种研究目的类型如图 5 所示, 括号中的数字代码相应研究类型的论文数量, 并给出了相应的定义. 本研究对所有论文进行了单标签分类, 涉及到两个或者多个研究内容论文, 将按照其研究的主要目的对其进行分类, 其他研究内容被视为次要目的. 为了降低主观性, 首先由两位作者根据研究目的类型及其定义, 独立地完成了每篇论文的分类. 然后计算两组结果一致性的 Cohen's Kappa[90]值, 计算得到 Kappa 为 0.88, 表明具有较强的一致性. 对于两位作者分类结果中存在的冲突, 通过引入一位有丰富定性分析经验和熟悉本研究的同事, 对差异部分进行讨论研究最终得到一致的结果. 每篇综述文献及其研究目的详见在线文档[52]. 下面对这 10 种类型进行简要的描述.

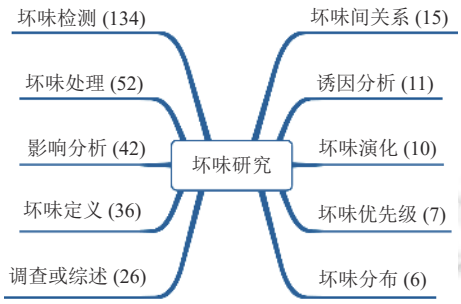

图 5 研究目的分类

(1) 坏味检测. 属于该类型的论文都以代码坏味为主要研究对象, 以提出各种检测方法、原型或者工具对所关注的代码坏味进行检测, 或者分析研究影响代码坏味检测的因素为主要目标, 进而改善对代码坏味的检测效果. 对于旨在提出检测方法的研究, 从检测方法的角度可以将其分为基于软件度量指标、基于机器学习和优化的算法、基于规则、基于图、基于树以及基于文本的检测方法研究. 基于软件度量指标的检测方法, 利用这些指标所代表的软件代码的量化特征, 从而识别代码坏味. Marinescu[91]针对使用单个度量指标进行坏味检测精度低的问题, 首先提出了一种基于度量指标的检测策略, 使用多个度量指标进行坏味定位; 基于机器学习和优化算法的方法, 主要利用机器学习和优化方法的学习和泛化能力来检测代码坏味. Maiga 等人[92,93]较早的将支持向量机 (SVM) 应用到代码坏味的检测当中, 提高了检测方法的准确率以及普适性; 基于规则的方法则利用人类专业知识、规范或者文档等生成启发式规则进行坏味检测. Moha 等人[94]则较早地提出了一种利用度量指标、结构关系、语义和结构属

性等信息定义一些设计代码坏味 (design smell), 并根据这些规范定义自动生成检测方法; 基于文本的检测方法则通过提取、匹配文本信息来定位可能存在的代码坏味. 例如 Bavota 等人[95]利用源代码和注释中标识符表示的文本信息矩阵, 以及方法之间依赖性表示的结构信息矩阵来检测功能嫉妒 (feature envy) 代码坏味; 基于图和基于树的检测方法则利用依赖关系图[96]、图数据库[97,98]和抽象语法树 (AST)[2,99,100]等来检测代码坏味. 除此之外, 其余研究则关注于影响代码坏味检测的因素, 包括开发人员对坏味的认知[101−103]、开发人员的协作[104−107]以及训练数据集的质量[69,108,109]等.

(2) 坏味处理. 属于该类型的论文旨在生成、优化重构方案, 以消除代码坏味对软件/代码质量的影响, 或者分析研究重构对软件质量带来的影响. 具体而言, 相关研究可以分为总结重构操作集合、重构推荐、研究重构影响 3 类. 早期对于代码坏味处理的研究集中关注于总结重构操作集, 以更好地支持开发人员识别到的代码坏味. 如 Deursen 等人[86]首先提出了一组特定于测试代码的重构操作, 以帮助开发人员改进测试代码. Monteiro 等人[70]则给出了一组面向方面的重构集合. 接着研究关注点转移到自动重构推荐方面, 可以进一步细分为自动重构特定代码坏味、检测重构机会以及优化重构 (或重构序列) 这 3 个方面. 如 Tsantalis 等人[110]首先提出了一种改善类型检查 (type checking) 坏味的自动重构方法. 针对重构执行前后对代码/软件影响的研究则略晚于其他两个方面, Fontana 等人[111]于 2011 年首次使用 6 个软件质量指标来观察重构操作对软件质量的影响.

(3) 影响分析. 这些研究的主要目的在于加深对代码坏味如何影响软件/代码质量以及软件发展等相关属性的了解, 有时会包含重构或者检测等次要目标. 具体而言, 研究包括分析代码坏味对软件缺陷、软件维护、软件安全性、软件性能与能耗、软件理解以及代码合并等质量和发展属性的影响. 大多数的研究都表明代码坏味会对以上属性产生负面影响, 比如拥有代码坏味的代码更容易出现错误, 或者代码坏味会增加维护所付出的努力等. 但也有部分研究认为这些影响是相对的, 并非代码坏味都与软件/代码质量的下降有关, 例如 Olbrich 等人[112]的研究表明假设每行代码所代表的功能一样多时, 上帝类 (God class) 反而是一种有效代码组织方式, 拥有更少的软件缺陷和更改. Sjøberg 等人[113] 的结果表明仅考虑代码坏味对维护工作量的影响时, 拒绝继承 (Refused bequest) 坏味反而与减少工作量相关. 并且并非所有的代码坏味都会以上属性产生影响. Hall 等人[22]则认为 Switch 惊现 (Switch statement) 坏味与软件缺陷没有关系. Palomba 等人[24]也认为部分代码坏味对于软件可维护性无关紧要. 除此之外, 部分研究利用代码坏味构建软件/代码缺陷或更改的预测模型. 例如 Soltanifar 等人[114]最早利用代码坏味信息提高其缺陷预测模型的性能.

(4) 坏味定义. 属于该类型的研究旨在分析代码坏味的特征, 进而将人们对代码坏味的主观认知, 以及代码坏味对代码质量的客观影响映射成更客观地的规则或标准, 通常表现为总结定义新的代码坏味, 代码坏味的类型详见表 1.

(5) 调查或综述. 属于该类型的研究通常通过对相关人员的问卷或访谈调查, 或者对以往研究进行综述来反映或者发现代码坏味相关的事实和规律.

(6) 坏味间关系. 属于该类型的论文主要针对代码坏味之间的联系以及这种联系对软件质量的影响进行研究. 研究内容主要包括位于同一文件/类中的代码坏味之间的共现关系, 位于不同文件/类中代码坏味之间的耦合依赖关系, 以及代码坏味之间联系对坏味检测、处理以及代码质量的影响. 例如 Pietrzak 等人[115]首次分析了代码坏味之间依赖关系, 以此来提高坏味检测过程的有效性和效率.

(7) 诱因分析. 属于该类型的论文旨在分析代码坏味的出现或引入与其他因素之间的关系. 除了人为因素的影响之外, 相关研究还探索了设计模式、度量指标以及环境因素与代码坏味存在之间的相关性. 人为因素主要表现为开发人员的经验以及开发压力等, 并且认为有经验的开发人员更偏向于引入代码坏味[36,116]. 有关设计模式对代码坏味的出现影响的研究表明, 设计模式固然能够避免或者减小代码坏味的出现[35,117], 但是仍然存在一些设计模式与代码坏味有着很高的共现率[35]. 而其他研究则从代码度量指标, 以及开发环境, 如开发社区的组织问题入手, 分析其与代码坏味出现之间的相关关系.

(8) 坏味演化. 属于该类型的论文旨在通过进行实证研究, 利用项目开发历史信息研究代码坏味随着时间的推移和软件版本的迭代如何进行演化. Olbrich 等人[118]首先研究了两种代码坏味在两个大型项目中的演化情况, 以

及对更改频率和大小的影响.

(9) 坏味优先级. 属于该类型的论文旨在研究在受不同代码坏味影响的实例中, 应该被关注/优化的代码坏味的优先级, 以及在受相同代码坏味影响的实例中, 应该被优先关注/优化的实例的优先级. 对于一组代码坏味不同的解决顺序可能需要不同的开销, 其次需要选择优先处理的代码坏味以节省开发成本, 因此进行了代码坏味优先级排序的探索[119,120], 并提出了基于严重性[121]、基于上下文[122,123]以及基于任务相关性[124]的不同排序方法. 除此之外, 为了节省精力和时间, 明智地选择实际需要重构的类, Rani 等人[125] 根据与系统中其他类的交互程度来判断类的严重程度, 并提出一种根据类的严重程度对其进行优先级排序的方法.

(10) 坏味分布. 属于该类型的研究通常将代码坏味数量视为软件/代码质量低下的指标, 研究代码坏味实例的种类、分布以及密度等情况, 进而对不同的实验对象进行比较. 研究内容主要包括分析代码坏味在不同领域[26,38,126]、不同编程语言[39]以及不同工具生成的测试代码[37,127]中的分布、密度等情况.

由图 5 可以看出, 在对代码坏味的研究中, 解决代码坏味的检测问题受到研究人员的最广泛关注, 论文数量占到全部数据集的 39.5%, 代码坏味准确且自动化的检测是进一步研究的前提条件, 首先已有代码坏味检测方法或工具在准确性以及检测范围等方面存在的缺陷, 以及新技术的不断涌现, 也成为了激励研究人员不断改善代码坏味检测的动力. 其次受到广泛关注是有关代码坏味处理、影响分析以及定义的问题.

由于以坏味间关系、诱因分析等为主要目的的研究数量较少且出现时间较晚, 以及综述和调查研究的特殊性, 本文仅针对被广泛讨论的 4 个研究方面, 从时间维度分别统计了相关论文的数量, 并使用多项式拟合得到相应的趋势线, 相应结果如图 6 所示. 从图 6(a)、(c)、(d) 可以看出, 有关代码坏味检测、分析代码坏味对软件质量的影响以及定义新的代码坏味的研究自提出以来, 相关研究数量呈总体上升趋势. 特别的, 有关代码坏味检测的研究最早在 2002 年发表, 并且直到 2019 年每年都会有相关研究结果发表, 可见对于代码坏味的检测, 学术界一直保持着较高的热情, 并且尝试利用不同的技术来改善检测质量. 而对于定义新的代码坏味, 虽然研究人员的热情在 Fowler 等人[1]给出较完整的面向对象坏味目录, 以及 Deursen 等人[86] 给出测试代码坏味目录之后有所下降, 但是随着人们意识到代码坏味可能存在于软件开发的各个阶段中后, 将代码坏味的定义拓展到其他领域, 或者针对特定对象定义新的代码坏味目录受到了研究人员的密切关注, 相关研究结果不断增加.

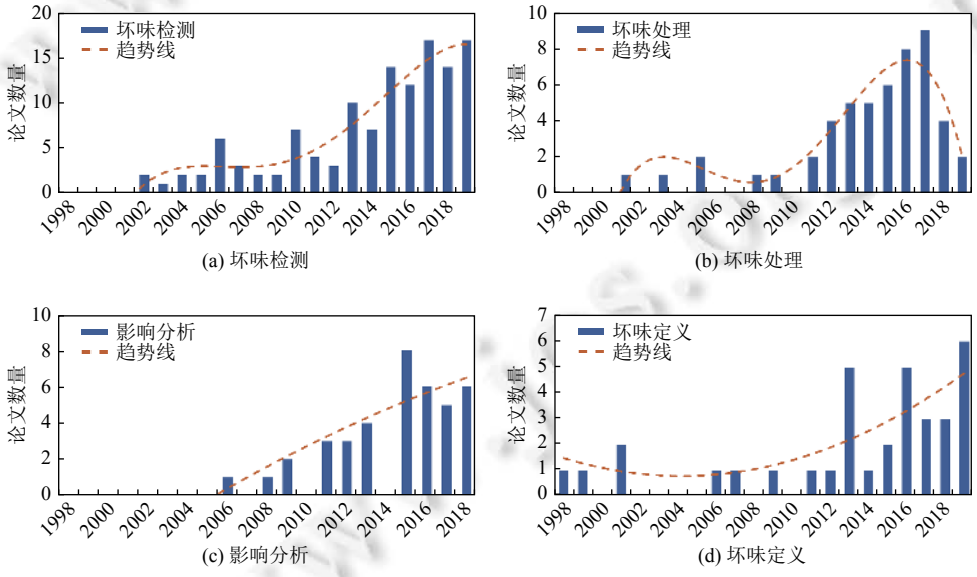

图 6 较受关注的 4 个方面的趋势

从图 6(b) 的结果中可以看出, 对于重构代码坏味以消除其影响的研究首先发表于 2001 年, 并于 2017 年相关研究数量的增长达到峰值, 之后的两年对于重构的研究热度开始下降, 具体表现为相关研究数量开始逐年下降.

本文将分析以上 4 种类型的研究数量变化趋势的可能原因. 具体来说, 快速且准确地找到代码坏味实例的自动化检测技术, 是对代码坏味其他特征展开研究的前提条件, 因此目前为止提出了许多不同的策略对代码坏味进行检测, 虽然有众多的检测方法和工具, 但是由于应用领域以及编程语言的特征不同, 代码坏味的检测工具并不能在所有情况下都达到很好的效果, 甚至在其专注的领域都未能达到较高的准确性. 比如在以往的综述中发现并没有一种方法或工具, 能够完整检测文献 [1] 定义的 22 种的代码坏味[128,129], 因此探索范围更广且能够准确识别坏味的检测方法, 可能是引起研究人员的广泛关注的原因.

而对于代码坏味对软件或代码质量的影响, 虽然 Fowler 认为代码坏味暗示了代码中可能的缺陷, 并代表了重构的机会, 但是对于代码坏味如何影响软件或代码质量缺少实际的证据支撑. 因此为了探索代码坏味是否影响以及如何影响软件质量, 研究人员展开了代码坏味与软件的缺陷、维护以及发展等相关质量属性之间关系的研究, 以加深对代码坏味的理解. 并且对代码坏味与软件缺陷关系的应用, 如将代码坏味应用于软件质量缺陷的预测模型之中进行代码缺陷预测[130], 也进一步激发了对于代码坏味影响的研究. 这些因素被认为是其研究数量保持增长的可能原因.

如上文所述, 人们认识到代码坏味的广泛存在, 是将代码坏味的概念不断拓展到其他领域最可能原因. 而对于重构处理代码坏味方面, 大部分研究仅关注被研究较多的一些代码坏味 (如, 上帝类、特征依恋、长方法等), 提出重构方案的生成或推荐方法并且得到了较好的效果. 同时由于新定义的代码坏味可能涉及到组件、包等更高的层次, 以及缺少标准化的重构操作方案, 因此对于代码坏味处理的研究在达到一个顶峰之后, 出现下降趋势.

综上所述, 研究人员对于提高代码坏味检测效果的热情仍呈现上升趋势, 同时代码坏味的影响分析, 以及定义新的代码坏味也是目前的研究热点. 虽然研究人员对于坏味处理的热度有所下降, 但是在代码坏味的重构处理方面仍然存在许多挑战, 例如提出针对特定领域所面对的代码坏味, 或者能够支持更多重构类型的自动最优重构方案的生成方法, 仍然需要进行更加深入的研究, 以支持更多的坏味或缺陷类型.

3.2 RQ3: 关键代码坏味

本节主要回答哪些代码坏味是应该被广泛认可且应该被重点关注的关键代码坏味, 并给出了代码坏味关键程度的量化方法. 研究发现, 仅从代码坏味被研究频数的角度来判断代码坏味的关键性并不可靠. 代码坏味在论文中的出现次数最多, 仅能代表代码坏味在学术界中的关注度. 但研究较多的原因有很多, 可能是该代码坏味比较容易检测, 或者是对该代码坏味的影响或处理分析比较容易进行. 因此还需要寻找额外的信息来更加准确地反映坏味的关键性程度. 而在一部分论文的研究结果中, 作者明确给出了代码坏味严重性以及广泛性的结论性描述, 因此这些作者明确给出的结论性描述, 可以作为补充信息进而反映代码坏味的关键程度. 通过统计和分析后发现, 这些结论或者观点可以分为广泛性结论、影响严重性结论两种类型, 下面分别对其进行详细介绍.

(1) 代码坏味广泛性的结论性描述. 代码坏味广泛性的结论性描述主要来源于论文实验结果中的结论性描述. 比如, 在文章所选择的应用、系统或者项目数据集中, 经过作者的实验证实, 某种坏味在这些数据集中出现次数最多, 或者扩散程度最严重. 而这些相关结论将被作为相关代码坏味广泛性的结论性描述. 结论性描述越多, 表明该坏味出现的概率越高、范围越大, 因此也更需要被重点关注. 例如 Fontana 等人[131]通过使用 iPlasma 对系统进行检查, 发现在这些系统中出现最多的坏味是: duplicated code、data class、God class、schizophrenic class 和 long method. 此外, 他们还发现这个规律在不同的软件应用领域均成立.

(2) 代码坏味影响的结论性描述. 在部分研究中, 作者会定量或者定性地描述代码坏味对软件质量的影响程度. 影响越大的代码坏味, 应该得到优先处理. 因此这些关于代码坏味影响的结论性描述, 也可以在一定程度上用于代码坏味的关键程度的评估. 在收集到的相关结论中, 对代码坏味影响的描述主要包括 4 个方面: 代码坏味与软件缺陷、软件维护工作量、软件更改频率以及软件能耗之间的关系.

代码坏味对软件缺陷的影响主要体现在包含坏味的代码在后续维护扩展的工作中更容易出现软件错误. 而如果已存在的研究表明受某种坏味影响的源代码其出错概率更高, 那么该代码坏味的影响就更为显著. 比如 Olbrich 等人[112]在 3 个开源系统及其问题跟踪系统 JIRA 和 Bugzilla 中, 调查了上帝类 (God class) 和脑类 (brain class) 与

缺陷和软件更改的关系. 他们的研究结果表明, 受上帝类和脑类代码坏味影响的类的更改更加频繁, 缺陷也更多.

代码坏味与软件维护的影响主要体现在包含坏味的代码会增加软件后续发展过程中的维护工作. 如果已有研究表明受某种坏味影响的源代码, 在后续的维护中需要付出更多的努力, 那么该代码坏味的影响就更显著. 如 Haque 等人[132]对代码坏味相关文献进行了调研和概述. 他们的研究结果表明坏味大大增加了维护阶段的工作, 尤其是 God Class 坏味显著增加了维护工作量.

代码坏味对软件更改的影响体现在包含坏味的代码在后续维护的过程中更容易发生更改. 如果相关研究表明受某种坏味影响的源代码其被更改的可能性更高, 那么该代码坏味的影响就更为显著. 如 Spadini 等人[23]研究了测试代码中的坏味与软件质量之间的关系, 他们的研究结果表明, 包含 indirect testing、eager test 和 assertion roulette 坏味更容易引起源代码的更改.

代码坏味对软件能耗的影响体现在包含代码坏味的软件会消耗更高的能耗. 如有已有相关研究对某种代码坏味与能耗之间的关系的进行描述, 那么该代码坏味的影响就更为显著. 例如 Carette 等人[27]在 5 个开源 Android 应用程序上的实证研究表明, 消除 internal getter/setter、member ignoring method 和 HashMap usage 等 3 种代码坏味可以显著降低应用程序的能耗.

以上两种代码坏味相关的结论性描述, 构成了评估其严重程度的额外信息. 本研究利用这种额外信息以及代码坏味本身的关注程度, 给出了一种量化代码坏味的严重程度的方法. 具体来讲, 代码坏味如果在相关研究中中出现次数较多, 并且在结论性描述中也出现次数较多, 则该认为该代码坏味是关键的, 并且使用代码坏味的关键程度值 (*Key Score of Code Smell*) 进行量化判断. 关键程度值的计算通过对坏味在文献中的出现频数 *NO* (number of occurrences) 和加权后的代码坏味结论性描述频数 *NC* (number of conclusive description) 求和得到. 没有对应结论性描述的坏味, 相应的 *NC* 值设为 0. 详细的计算公式如公式 (1) 所示:

$$Key\ Score\ of\ Code\ Smell = NO + \frac{NO_{\mathrm{Sum}}}{NC_{\mathrm{Sum}}} * NC \tag{1}$$

其中, NC_{Sum} 代表所有坏味在结论性描述中出现的频数之和, NO_{Sum} 代表所有坏味在论文中出现的频数之和. 利用两数的商作为结论性描述频数的权重, 以加强这些结论性额外信息的影响. 表 3 展示了根据关键程度值进行排序后的前十位代码坏味. 结果发现按照关键程度排序后的前十个代码坏味, 与单纯按照频数排序后的结果有很大差别. 而由于关键程度考虑了额外的代码坏味广泛性和严重性信息, 因此能更好地评估代码坏味的关键程度.

表 3 关键程度值在前十的代码坏味

代码坏味	坏味定义
God class	在一个好的面向对象的设计中, 系统的职能是均匀分布在顶级类中的God class坏味违背了这一设计, 把太多的职能集中在一个类中, 导致该类负责过多的功能
Feature envy	当一个方法对其他类的兴趣高过对自己所处类的兴趣, 判定为该坏味
Duplicated code	在项目中由于粘贴复制、重复编码等一系列原因出现重复的代码时, 则出现Duplicate code坏味
Long method	当方法的代码行数过长时, 出现该坏味
Long parameter list	当一个方法具有很长的参数列表时, 就会出现该坏味
Complex class	Complex class是包含复杂方法的类, 当一个类的复杂度很高时, 会出现代码难以理解等问题, 出现该坏味
Data class	Data class是指一个类仅用来存储数据, 及用于访问这些数据的方法, 而没有其他的数据处理功能
Assertion roulette	当测试方法中包含许多没有解释信息的断言且发生断言失败时, 无法判断出哪个断言发生了错误, 这种情况被认为是该坏味
Spaghetti code	该坏味指非结构化的难以理解和维护的代码. 这类代码包含很多复杂的控制语句来(比如goto)进行代码的逻辑控制, 而不是使用结构的代码(方法、结构体)来进行编写. 如果软件使用面向对象的语言进行开发, 而开发代码使用面向过程的方式来编写, 亦可导致本坏味
Blob class	Blob class处理很多职责, 其属性和方法与不同的概念和过程相关, 有着很低的内聚力, 并且经常与许多数据类相关联

4 RQ4: 工业界应用

本研究通过调查选择了 6 种用于被流行度高且维护良好代码质量检测商用工具, 例如, CppDepend[133]被 Google、惠普等 1 000 多个公司使用以提高 C/C++项目的代码质量, 并且一直被积极维护; Sonarsource[134]作为代码质量和代码安全性的保障工具, 被包括腾讯、三星在内的 6 000 多个企业所使用, 并且一直保持着对产品的更新和维护. 通过对这 6 种质量检测工具的调研, 从侧面来反映代码坏味在工业界实际开发中的现状, 并且与上节得到的关键代码坏味进行比较, 以此分析实际开发与学术研究中对于代码坏味的关注点的一致性. 表 4 对收集到的 6 种代码质量检测工具进行了简要的介绍.

表 4 代码质量检查工具

工具名	支持的语言	工具类型	支持检测的代码坏味
CppDepend[133]	C, C++	独立工具, VisualStudio拓展	Too many methods, too many fields, God method, long parameter list, low cohesion only, dead code, duplicated code
Designite[135]	C#, Java	独立工具	Long parameter list, duplicated code, deep hierarchy, cyclic dependency, cyclic hierarchy等36种
NDeend[136]	C#	VisualStudio拓展	God class, too many methods, too many fields, God method, long parameter list, low cohesion only, dead code, deep hierarchy
PMD[137]	Java	独立工具, IDE拓展	Long parameter list, dead code, duplicated code, too many fields, complex class, god class, long method等14种
SDMetrics[138]	C++, Java, Delphi, Smalltalk	独立工具	Cyclic hierarchy, duplicated code, god class, class uses multiple inheritance, dead code, long parameter list, cyclic dependency
Sonarsource[134]	C, C++, Java, Python, JavaScript等27种语言	独立工具, IDE拓展, 云工具	Long parameter list, dead code, deep hierarchy, too many fields, complex class, god method, magic number等16种

表 4 中 6 种作为代码质量检查的工具都包含众多检测规则, 可以用于代码风格、安全性和设计等方面的代码检查. 因此本研究参照各个工具检测规则的具体介绍和代码坏味的定义, 以此来确定工具支持的代码坏味. 并且由于工具检测规则的命名方式并不统一, 因此在确定其检测内容后使用含义相同的代码坏味名称来表示, 以此得到了部分代码坏味在商用代码质量检查工具中的关注度. 表 5 显示了受到 3 种及以上工具支持的代码坏味, 可以看到 long parameter list 代码坏味能够被所有 6 种工具检测, 并且同样是关键程度值在前 10 的代码坏味. 除此之外, God class、duplicated code 同样也是受到工具支持较多的且关键程度较高的代码坏味. 从总体上看, 关键程度值在前 10 代码坏味在以上 6 种工具的检测规则中只有 1 种未受到支持, 即 blob class. 除此之外, 代码质量检查工具还关注于其他 5 种关键程度值并不高的代码坏味, 如 dead code、God method 等. 进一步的分析发现, 在工业界中的检测工具中, 较多的关注于能够较明显的影响质量度量指标值的代码坏味, 或者得到较广泛论证的代码坏味, 抑或易于检测的代码坏味. 而学术界对代码坏味的研究则更具有前瞻性, 更倾向于利用新技术、新方法, 来解决检测过程中的挑战.

表 5 代码质量检查工具

代码坏味	支持检测的工具数
Long parameter list	6
Dead code	5
Duplicated code	4
Too many fields	4
Deep hierarchy	3
God class	3
God method	3
Too many methods	3

总而言之, 代码坏味在项目实际开发中会作为质量问题被开发人员关注并检测, 对于关键程度值较高的代码坏味, 学术界和工业界的关注点是一致的. 开发过程中的代码质量检查工具对其提供了很好的支持. 关键程度值最高的 10 个代码坏味, 有 9 个得到了工业界的重点关注和认可, 代码坏味 blob class 未受到支持的可能原因是其表现形式与 God class 过于相同, 因此只提供了 God class 的检测规则.

从工业界和学术界代码坏味的关注列表的角度分析, 学术界和工业界对于代码坏味的关注点存在一定差距. 工业界更关注质量、成本和效率之间的平衡, 因此倾向于关注容易检测和被广泛认可的代码坏味, 而学术界则倾向于优化坏味检测, 来解决较难/较少检测和被广泛认可的代码坏味. 具体表现为检查工具对一些关键程度值不高的代码坏味提供了支持, 并且保持了较高的关注度. 其可能的原因是代码质量检查工具通常会计算并且关注于代码的质量度量指标值, 对于高于或者低于设定阈值的质量度量指标值会认为是质量问题. 因此针对能够直接反映或者部分反映这些质量问题的代码坏味, 如 too many fields、deep hierarchy, 质量检测工具也都展现出了较高的关注度. 同时对一些易于检测的代码坏味, 如死代码 (dead code), 工具也提供了检测支持.

5 研究结果与展望

本节总结了本次综述的研究结果, 并针对综述中体现的问题, 提出了对未来代码坏味研究的建议. 首先利用获得的数据集研究了学术界对代码坏味的研究状态, 揭示了近年来软件工程领域的研究人员对代码坏味日益增长的研究兴趣, 以及不断定义新的代码坏味的研究趋势. 在不考虑仅研究代码克隆的论文的情况下, 有关代码坏味的研究论文数不断增加, 并在最近 4 年一直保持在较高水平.

研究问题 RQ1 探索了新定义代码坏味的发展趋势. 定义新的代码坏味呈现出先下降而后一直增长的趋势, 将代码坏味的概念拓展到其他领域, 并定义新的代码坏味成为现阶段的研究热点. 研究过程中发现, 目前仍然缺少对代码坏味进行分类的标准. 因此在对研究问题 RQ1 的回答中, 遵循了最初研究中对代码坏味的分类. 在现有的代码坏味目录中一共得到了共 38 种不同的代码坏味类型, 涉及安全、性能、可用性等多个方面. 但是由于缺少代码坏味的分类标准, 作者对代码坏味进行定义和分类通常遵循了不同的规则, 造成了代码坏味类型之间缺少规律, 存在包含与被包含的关系. 代码坏味类型的冗繁, 影响了研究人员以及开发人员对于代码坏味的识别, 增加了相关研究在实际开发中应用的难度. 因此在将来的研究中, 应该考虑定义一种完备、一致且准确的代码坏味分类标准, 以规范代码坏味的定义和拓展, 进一步提高代码坏味被关注的程度.

研究问题 RQ2 展示了相关研究的研究目的. 研究通过主题分析得到了彼此正交的 10 个研究目的类型, 并对每种类型的研究进行了详细的说明. Zhang 等人[42]对 2000 年至 2009 年之间的 39 篇坏味相关论文的研究目的统计分析后发现, 研究人员重点关注于代码坏味的检测, 而很少从重构角度分析代码坏味. de Paulo Sobrinho 等人[43]对 2017 年之前的 124 相关研究的研究目的进行了分析, 统计得到了 13 种代码坏味的研究目的类型, 并且发现代码坏味检测的研究数量最多, 其次是坏味影响以及坏味定义. 而本研究扩大了综述的时间跨度, 涵盖了更多更新的研究. 最终发现了两种新的研究目的, 即诱因分析和坏味分布, 同时结果表明虽然坏味检测仍受到了最多的关注, 但代码坏味处理超过了坏味影响和坏味定义, 其研究数量处于第 2 位. 在对被广泛研究的 4 种研究目的的发展趋势分析后发现, 代码坏味重构处理的相关研究在 2011 年之前相对较少, 但在之后发展中得到了研究人员的关注, 相关研究数呈现上升趋势, 在近几年的研究热度才有所下降. 除此之外学术界对于其他 3 种研究目的的关注程度都保持着增长趋势. 通过对本问题的回答, 统计得到了相关研究及其目的的完整目录[1], 将有助于研究人员更方便地了解相关研究的进展. 同时确定了以下差距和机遇: ① 对于代码坏味重构的研究热度虽然有所下降, 但是目前对于较高层次代码坏味, 以及一些非面向对象代码中的代码坏味的自动重构解决方案相对较少, 缺少更加深入的研究. 因此建议研究人员针对不同的层次或者成熟的编程语言, 研究和开发相应的代码坏味重构方案; ② 其次对于其他研究目的的论文, 比如出现原因、优先级等, 代表了对代码坏味不同特性的分析, 有助于推进该领域的研究.

研究问题 RQ3 解决了代码坏味关键程度的量化. 利用以往研究中有关代码坏味广泛性或严重性的结论性描述, 作为已被验证的补充信息来反映代码坏味的关键程度. 通过对本问题的回答, 完成了对代码坏味的关键程度的

量化, 并且可以依据其关键程度值的排序, 选择应该付出更多努力的代码坏味. 除了对代码坏味的研究, 量化之后的结果也给实际开发应该重点关注的代码坏味.

研究问题 RQ4 分析比较了学术界与工业界对代码坏味关注点的差异. 在对本问题的回答中, 利用商业代码质量检查工具从侧面反映了工业界开发中对代码坏味的关注点. 研究发现, 这些经常用于实际开发中的工具较多的关注于能够直接或间接影响软件质量指标的代码坏味, 并且一些新颖的代码坏味检测技术, 比如基于机器学习的检测方法, 并未应用于工具检测过程中. 而是更多采用模式匹配等方法, 以达到支持用户自定义规则的要求. 因此在研究代码坏味的定义以及检测时, 明确代码坏味带来的影响, 并且提供选择软件质量指标进行检测的理由, 将有利于提高开发人员对代码坏味接受程度.

6 总 结

代码坏味仍然是软件工程领域的研究热点. 本文综述了现有代码坏味相关的研究, 围绕代码坏味的持续演化, 关键代码坏味及研究热点, 和代码坏味在工业界的研究现状 3 个方面展开了研究, 完成了对代码坏味发展趋势以及研究热点的分析, 提出一种代码坏味关键程度的量化方案, 以分析其应该受关注和付出努力的优先级. 最后从工业界实际情况出发, 对实际开发过程中用于代码坏味检测的静态代码检查工具进行调查, 分析了学术界和工业界对于代码坏味关注的一致性, 并分析其可能存在的原因, 为研究人员和致力于改善软件质量的从业人员给出研究机会指明了研究机会.

References:

[1] Fowler M, Beck K. Refactoring: Improving the Design of Existing Code. Upper Saddle River: Addison-Wesley, 1999.

[2] van Emden E, Moonen L. Java quality assurance by detecting code smells. In: Proc. of the 9th Working Conf. on Reverse Engineering. Richmond: IEEE, 2002. 97–106. [doi: 10.1109/WCRE.2002.1173068]

[3] Moha N, Guéhéneuc YG, Duchien L, Le Meur AF. DECOR: A method for the specification and detection of code and design smells. IEEE Trans. on Software Engineering, 2010, 36(1): 20–36. [doi: 10.1109/TSE.2009.50]

[4] Liu H, Xu ZF, Zou YZ. Deep learning based feature envy detection. In: Proc. of the 33rd IEEE/ACM Int'l Conf. on Automated Software Engineering. Montpellier: IEEE, 2018. 385–396. [doi: 10.1145/3238147.3238166]

[5] Ouni A, Kessentini M, Sahraoui H, Inoue K, Deb K. Multi-criteria code refactoring using search-based software engineering: An industrial case study. ACM Trans. on Software Engineering and Methodology, 2016, 25(3): 23. [doi: 10.1145/2932631]

[6] Vidal S, Berra I, Zulliani S, Marcos C, Pace JAD. Assessing the refactoring of brain methods. ACM Trans. on Software Engineering and Methodology, 2018, 27(1): 2. [doi: 10.1145/3191314]

[7] Fard AM, Mesbah A. JSNOSE: Detecting JavaScript code smells. In: Proc. of the 13th IEEE Int'l Working Conf. on Source Code Analysis and Manipulation. Eindhoven: IEEE, 2013. 116–125. [doi: 10.1109/SCAM.2013.6648192]

[8] Fakhoury S, Arnaoudova V, Noiseux C, Khomh F, Antoniol G. Keep it simple: Is deep learning good for linguistic smell detection? In: Proc. of the 25th IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering. Campobasso: IEEE, 2018. 602–611. [doi: 10.1109/SANER.2018.8330265]

[9] Bu YF, Liu H, Li GJ. God class detection approach based on deep learning. Ruan Jian Xue Bao/Journal of Software, 2019, 30(5): 1359–1374 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5724.htm [doi: 10.13328/j.cnki.jos.005724]

[10] Amorim L, Costa E, Antunes N, Fonseca B, Ribeiro M. Experience report: Evaluating the effectiveness of decision trees for detecting code smells. In: Proc. of the 26th IEEE Int'l Symp. on Software Reliability Engineering. Gaithersbury: IEEE, 2015. 261–269. [doi: 10.1109/ISSRE.2015.7381819]

[11] Kreimer J. Adaptive detection of design flaws. Electronic Notes in Theoretical Computer Science, 2005, 141(4): 117–136. [doi: 10.1016/j.entcs.2005.02.059]

[12] Ouni A, Kessentini M, Sahraoui H, Boukadoum M. Maintainability defects detection and correction: A multi-objective approach. Automated Software Engineering, 2013, 20(1): 47–79. [doi: 10.1007/s10515-011-0098-8]

[13] Saranya G, Nehemiah HK, Kannan A, Nithya V. Model level code smell detection using EGAPSO based on similarity measures. Alexandria Engineering Journal, 2018, 57(3): 1631–1642. [doi: 10.1016/j.aej.2017.07.006]

[14] Kessentini M, Kessentini W, Sahraoui H, Boukadoum M, Ouni A. Design defects detection and correction by example. In: Proc. of the

19th IEEE Int'l Conf. on Program Comprehension. Kingston: IEEE, 2011. 81–90. [doi: 10.1109/ICPC.2011.22]

[15] Kessentini M, Ouni A. Detecting android smells using multi-objective genetic programming. In: Proc. of the 4th IEEE/ACM Int'l Conf. on Mobile Software Engineering and Systems. Buenos Aires: IEEE, 2017. 122–132. [doi: 10.1109/MOBILESoft.2017.29]

[16] Mansoor U, Kessentini M, Maxim BR, Deb K. Multi-objective code-smells detection using good and bad design examples. Software Quality Journal, 2017, 25(2): 529–552. [doi: 10.1007/s11219-016-9309-7]

[17] Hozano M, Garcia A, Antunes N, Fonseca B, Costa E. Smells are sensitive to developers! on the efficiency of (un)guided customized detection. In: Proc. of the 25th IEEE/ACM Int'l Conf. on Program Comprehension. Buenos Aires: IEEE, 2017. 110–120. [doi: 10.1109/ICPC.2017.32]

[18] Ujihara N, Ouni A, Ishio T, Inoue K. c-JRefRec: Change-based identification of move method refactoring opportunities. In: Proc. of the 24th IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering. Klagenfurt: IEEE, 2017. 482–486. [doi: 10.1109/SANER.2017.7884658]

[19] Salehie M, Li SM, Tahvildari L. A metric-based heuristic framework to detect object-oriented design flaws. In: Proc. of the 14th IEEE Int'l Conf. on Program Comprehension. Athens: IEEE, 2006. 159–168. [doi: 10.1109/ICPC.2006.6]

[20] Yamashita A, Moonen L. To what extent can maintenance problems be predicted by code smell detection?—An empirical study. Information and Software Technology, 2013, 55(12): 2223–2242. [doi: 10.1016/j.infsof.2013.08.002]

[21] Zhu C, Zhang XF, Feng Y, Chen L. An empirical study of the impact of code smell on file changes. In: Proc. of the 18th IEEE Int'l Conf. on Software Quality, Reliability and Security. Lisbon: IEEE, 2018. 238–248. [doi: 10.1109/QRS.2018.00037]

[22] Hall T, Zhang M, Bowes D, Sun Y. Some code smells have a significant but small effect on faults. ACM Trans. on Software Engineering and Methodology, 2014, 23(4): 33. [doi: 10.1145/2629648]

[23] Spadini D, Palomba F, Zaidman A, Bruntink M, Bacchelli A. On the relation of test smells to software code quality. In: Proc. of the 2018 IEEE Int'l Conf. on Software Maintenance and Evolution. Madrid: IEEE, 2018. 1–12. [doi: 10.1109/ICSME.2018.00010]

[24] Palomba F, Bavota G, Di Penta M, Fasano F, Oliveto R, De Lucia A. On the diffuseness and the impact on maintainability of code smells: A large scale empirical investigation. Empirical Software Engineering, 2018, 23(3): 1188–1221. [doi: 10.1007/s10664-017-9535-z]

[25] Mumtaz H, Alshayeb M, Mahmood S, Niazi M. An empirical study to improve software security through the application of code refactoring. Information and Software Technology, 2018, 96: 112–125. [doi: 10.1016/j.infsof.2017.11.010]

[26] Ghafari M, Gadient P, Nierstrasz O. Security smells in android. In: Proc. of the 17th IEEE Int'l Working Conf. on Source Code Analysis and Manipulation. Shanghai: IEEE, 2017. 121–130. [doi: 10.1109/SCAM.2017.24]

[27] Carette A, Younes MAA, Hecht G, Moha N, Rouvoy R. Investigating the energy impact of android smells. In: Proc. of the 24th IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering. Klagenfurt: IEEE, 2017. 115–126. [doi: 10.1109/SANER.2017.7884614]

[28] Palomba F, Di Nucci D, Panichella A, Zaidman A, De Lucia A. On the impact of code smells on the energy consumption of mobile applications. Information and Software Technology, 2019, 105: 43–55. [doi: 10.1016/j.infsof.2018.08.004]

[29] Lyu YJ, Alotaibi A, Halfond WGJ. Quantifying the performance impact of SQL antipatterns on mobile applications. In: Proc. of the 2019 IEEE Int'l Conf. on Software Maintenance and Evolution. Cleveland: IEEE, 2019. 53–64. [doi: 10.1109/ICSME.2019.00015]

[30] Catolino G, Palomba F, Fontana FA, De Lucia A, Zaidman A, Ferrucci F. Improving change prediction models with code smell-related information. Empirical Software Engineering, 2020, 25(1): 49–95. [doi: 10.1007/s10664-019-09739-0]

[31] Sharma T, Fragkoulis M, Spinellis D. House of cards: Code smells in open-source C# repositories. In: Proc. of the 2017 ACM/IEEE Int'l Symp. on Empirical Software Engineering and Measurement. Toronto: IEEE, 2017. 424–429. [doi: 10.1109/ESEM.2017.57]

[32] Tahmid A, Nahar N, Sakib K. Understanding the evolution of code smells by observing code smell clusters. In: Proc. of the 23rd IEEE Int'l Conf. on Software Analysis, Evolution, and Reengineering. Osaka: IEEE, 2016. 8–11. [doi: 10.1109/SANER.2016.45]

[33] Palomba F, Bavota G, Di Penta M, Fasano F, Oliveto R, De Lucia A. A large-scale empirical study on the lifecycle of code smell co-occurrences. Information and Software Technology, 2018, 99: 1–10. [doi: 10.1016/j.infsof.2018.02.004]

[34] Palomba F, Tamburri DA, Fontana FA, Oliveto R, Zaidman A, Serebrenik A. Beyond technical aspects: How do community smells influence the intensity of code smells? IEEE Trans. on Software Engineering, 2021, 47(1): 108–129. [doi: 10.1109/TSE.2018.2883603]

[35] Sousa BL, Bigonha MAS, Ferreira KAM. An exploratory study on cooccurrence of design patterns and bad smells using software metrics. Software: Practice and Experience, 2019, 49(7): 1079–1113. [doi: 10.1002/spe.2697]

[36] Habchi S, Moha N, Rouvoy R. The rise of android code smells: Who is to blame? In: Proc. of the 16th IEEE/ACM Int'l Conf. on Mining Software Repositories. Montreal: IEEE, 2019. 445–456. [doi: 10.1109/MSR.2019.00071]

[37] Bavota G, Qusef A, Oliveto R, De Lucia A, Binkley D. Are test smells really harmful? An empirical study. Empirical Software Engineering, 2015, 20(4): 1052–1094. [doi: 10.1007/s10664-014-9313-0]

[38] De Padua GB, Shang WY. Studying the prevalence of exception handling anti-patterns. In: Proc. of the 25th IEEE/ACM Int'l Conf. on Program Comprehension. Buenos Aires: IEEE, 2017. 328–331. [doi: 10.1109/ICPC.2017.1]

[39] Mateus BG, Martinez M. An empirical study on quality of android applications written in Kotlin language. Empirical Software Engineering, 2019, 24(6): 3356–3393. [doi: 10.1007/s10664-019-09727-4]

[40] Kebir S, Borne I, Meslati D. A genetic algorithm-based approach for automated refactoring of component-based software. Information and Software Technology, 2017, 88: 17–36. [doi: 10.1016/j.infsof.2017.03.009]

[41] Morales R, Saborido R, Khomh F, Chicano F, Antoniol G. EARMO: An energy-aware refactoring approach for mobile apps. IEEE Trans. on Software Engineering, 2018, 44(12): 1176–1206. [doi: 10.1109/TSE.2017.2757486]

[42] Zhang M, Hall T, Baddoo N. Code bad smells: A review of current knowledge. Journal of Software Maintenance and Evolution: Research and Practice, 2011, 23(3): 179–202. [doi: 10.1002/smr.521]

[43] de Paulo Sobrinho EV, De Lucia A, De Almeida Maia M. A systematic literature review on bad smells-5 w's: Which, when, what, who, where. IEEE Trans. on Software Engineering, 2021, 47(1): 17–66. [doi: 10.1109/TSE.2018.2880977]

[44] Sharma T, Spinellis D. A survey on software smells. Journal of Systems and Software, 2018, 138: 158–173. [doi: 10.1016/j.jss.2017.12.034]

[45] Chen ZF, Chen L, Ma WWY, Zhou XY, Zhou YM, Xu BW. Understanding metric-based detectable smells in Python software: A comparative study. Information and Software Technology, 2018, 94: 14–29. [doi: 10.1016/j.infsof.2017.09.011]

[46] Vassallo C, Proksch S, Gall HC, Di Penta M. Automated reporting of anti-patterns and decay in continuous integration. In: Proc. of the 41st IEEE/ACM Int'l Conf. on Software Engineering. Montreal: IEEE, 2019. 105–115. [doi: 10.1109/ICSE.2019.00028]

[47] Afjehei SS, Chen TH, Tsantalis N. iPerfDetector: Characterizing and detecting performance anti-patterns in iOS applications. Empirical Software Engineering, 2019, 24(6): 3484–3513. [doi: 10.1007/s10664-019-09703-y]

[48] Kitchenham B. Procedures for performing systematic reviews. Technical Report, TR/SE-0401, Keele: Keele University, 2004. 1–26.

[49] Malveau RC, Brown WJ, McCormick HW, Mowbray TJ. AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. Hoboken: Wiley, 1998.

[50] Roy CK, Zibran MF, Koschke R. The vision of software clone management: Past, present, and future (keynote paper). In: Proc. of the 2014 Software Evolution Week-IEEE Conf. on Software Maintenance, Reengineering, and Reverse Engineering. Antwerp: IEEE, 2014. 18–33. [doi: 10.1109/CSMR-WCRE.2014.6747168]

[51] Wohlin C. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In: Proc. of the 18th Int'l Conf. on Evaluation and Assessment in Software Engineering. London: ACM, 2014. 38. [doi: 10.1145/2601248.2601268]

[52] Tian YC. 2021. https://github.com/CodeSmellSurvey/CodeSmell

[53] Reimann J, Brylski M. , Aßmann U. A tool-supported quality smell catalogue for android developers. In: Proc. of the 2014 Conf. Modellierung in the Workshop Modellbasierte und Modellgetriebene Softwaremodernisierung (MMSM). 2014.

[54] Habchi S, Hecht G, Rouvoy R, Moha N. Code smells in iOS apps: How do they compare to Android? In: Proc. of the 4th IEEE/ACM Int'l Conf. on Mobile Software Engineering and Systems. Buenos Aires: HAL, 2017. 110–121.

[55] Greiler M, Van Deursen A, Storey MA. Automated detection of test fixture strategies and smells. In: Proc. of the 6th IEEE Int'l Conf. on Software Testing, Verification and Validation. Luxembourg: IEEE, 2013. 322–331. [doi: 10.1109/ICST.2013.45]

[56] Nagy C, Cleve A. A static code smell detector for SQL queries embedded in Java code. In: Proc. of the 17th IEEE Int'l Working Conf. on Source Code Analysis and Manipulation. Shanghai: IEEE, 2017. 147–152. [doi: 10.1109/SCAM.2017.19]

[57] Ganesh SG, Sharma T, Suryanarayana G. Towards a principle-based classification of structural design smells. Journal of Object Technology, 2013, 12(2). [doi: 10.5381/jot.2013.12.2.a1]

[58] Mäntylä M, Vanhanen J, Lassenius C. A taxonomy and an initial empirical study of bad smells in code. In: Proc. of the 2003 IEEE Int'l Conf. on Software Maintenance. Amsterdam: IEEE, 2003. 381–384. [doi: 10.1109/ICSM.2003.1235447]

[59] Karwin B. SQL Antipatterns: Avoiding the Pitfalls of Database Programming. Dallas: Pragmatic Bookshelf, 2010.

[60] Gadient P, Ghafari M, Frischknecht P, Nierstrasz O. Security code smells in Android ICC. Empirical Software Engineering, 2018, 24(5): 3046–3076. [doi: 10.1007/s10664-018-9673-y]

[61] Carvalho SG, Aniche M, Veríssimo J, Durelli RS, Gerosa MA. An empirical catalog of code smells for the presentation layer of android apps. Empirical Software Engineering, 2019, 24(6): 3546–3586. [doi: 10.1007/s10664-019-09768-9]

[62] Palma F, Gonzalez-Huerta J, Moha N, Guéhéneuc YG, Tremblay G. Are RESTful APIs well-designed? Detection of their linguistic (Anti)patterns. In: Proc. of the 13th Int'l Conf. on Service-oriented Computing. Goa: Springer, 2015. 171–187. [doi: 10.1007/978-3-662-48616-0_11]

[63] Chen BY, Jiang ZM. Characterizing and detecting anti-patterns in the logging code. In: Proc. of the 39th IEEE/ACM Int'l Conf. on

Software Engineering. Buenos Aires: IEEE, 2017. 71–81. [doi: 10.1109/ICSE.2017.15]

[64] Destefanis G, Tonelli R, Concas G, Marchesi M. An analysis of anti-micro-patterns effects on fault-proneness in large Java systems. In: Proc. of the 27th Annual ACM Symp. on Applied Computing. Trento: ACM, 2012. 1251–1253. [doi: 10.1145/2245276.2231972]

[65] Jezek K, Lipka R. Antipatterns causing memory bloat: A case study. In: Proc. of the 24th IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering. Klagenfurt: IEEE, 2017. 306–315. [doi: 10.1109/SANER.2017.7884631]

[66] Paternò F, Schiavone AG, Conti A. Customizable automatic detection of bad usability smells in mobile accessed Web applications. In: Proc. of the 19th Int'l Conf. on Human-computer Interaction with Mobile Devices and Services. Vienna: ACM, 2017. 42. [doi: 10.1145/3098279.3098558]

[67] El-Dahshan KA, Elsayed EK, Ghannam NE. Comparative study for detecting mobile application's anti-patterns. In: Proc. of the 8th Int'l Conf. on Software and Information Engineering. Cairo: ACM, 2019. 1–8. [doi: 10.1145/3328833.3328834]

[68] Garcia J, Popescu D, Edwards G, Medvidovic N. Identifying architectural bad smells. In: Proc. of the 13th European Conf. on Software Maintenance and Reengineering. Kaiserslautern: IEEE, 2009. 255–258. [doi: 10.1109/CSMR.2009.59]

[69] Pecorelli F, Di Nucci D, de Roover C, de Lucia A. A large empirical assessment of the role of data balancing in machine-learning-based code smell detection. Journal of Systems and Software, 2020, 169: 110693. [doi: 10.1016/j.jss.2020.110693]

[70] Monteiro MP, Fernandes JM. Towards a catalog of aspect-oriented refactorings. In: Proc. of the 4th Int'l Conf. on Aspect-oriented Software Development. Chicago: ACM, 2005. 111–122. [doi: 10.1145/1052898.1052908]

[71] Brabra H, Mtibaa A, Sliman L, Gaaloul W, Benatallah B, Gargouri F. Detecting cloud (anti)patterns: OCCI perspective. In: Proc. of the 14th Int'l Conf. on Service-oriented Computing. Banff: Springer, 2016. 202–218. [doi: 10.1007/978-3-319-46295-0_13]

[72] Chen TH, Shang WY, Jiang ZM, Hassan AE, Nasser M, Flora P. Detecting performance anti-patterns for applications developed using object-relational mapping. In: Proc. of the Int'l Conf. on Software Engineering. Hyderabad: ACM, 2014. 1001–1012. [doi: 10.1145/2568225.2568259]

[73] Wert A, Oehler M, Heger C, Farahbod R. Automatic detection of performance anti-patterns in inter-component communications. In: Proc. of the 10th Int'l ACM SIGSOFT Conf. on Quality of Software Architectures. Marcq-en-Bareul: ACM, 2014. 3–12. [doi: 10.1145/2602576.2602579]

[74] Smith CU, Williams LG. Software performance antipatterns; common performance problems and their solutions. In: Proc. of the 27th Int'l Computer Measurement Group Conf. 2001. 797–806.

[75] Dallal JA, Briand LC. A precise method-method interaction-based cohesion metric for object-oriented classes. ACM Trans. on Software Engineering and Methodology, 2012, 21(2): 8. [doi: 10.1145/2089116.2089118]

[76] Sharma T, Fragkoulis M, Spinellis D. Does your configuration code smell? In: Proc. of the 13th Int'l Conf. on Mining Software Repositories. Austin: ACM, 2016. 189–200. [doi: 10.1145/2901739.2901761]

[77] Rahman MR, Rahman A, Williams L. Share, but be aware: Security smells in python gists. In: Proc. of the 2019 IEEE Int'l Conf. on Software Maintenance and Evolution. Cleveland: IEEE, 2019. 536–540. [doi: 10.1109/ICSME.2019.00087]

[78] Li ZH, Chen TH, Yang JQ, Shang WY. DLFinder: Characterizing and detecting duplicate logging code smells. In: Proc. of the 41st IEEE/ACM Int'l Conf. on Software Engineering. Montreal: IEEE, 2019. 152–163. [doi: 10.1109/ICSE.2019.00032]

[79] Rahman A, Parnin C, Williams L. The seven sins: Security smells in infrastructure as code scripts. In: Proc. of the 41st IEEE/ACM Int'l Conf. on Software Engineering. Montreal: IEEE, 2019. 164–175. [doi: 10.1109/ICSE.2019.00033]

[80] Nguyen HV, Nguyen HA, Nguyen TT, Nguyen AT, Nguyen TN. Detection of embedded code smells in dynamic Web applications. In: Proc. of the 27th IEEE/ACM Int'l Conf. on Automated Software Engineering. Essen: IEEE, 2012. 282–285. [doi: 10.1145/2351676.2351724]

[81] Gauthier F, Merlo E. Semantic smells and errors in access control models: A case study in PHP. In: Proc. of the 35th Int'l Conf. on Software Engineering. San Francisco: IEEE, 2013. 1169–1172. [doi: 10.1109/ICSE.2013.6606670]

[82] Gottschalk M, Josefiok M, Jelschen J, Winter A. Removing energy code smells with reengineering services. In: Proc. of the Lecture Notes in Informatics (LNI), Proc. Series of the Gesellschaft fur Informatik (GI). Bonn: Gesellschaft für Informatik, 2012. 441–455.

[83] Nayrolles M, Moha N, Valtchev P. Improving SOA antipatterns detection in service based systems by mining execution traces. In: Proc. of the 20th Working Conf. on Reverse Engineering. Koblenz: IEEE, 2013. 321–330. [doi: 10.1109/WCRE.2013.6671307]

[84] Sena D, Coelho R, Kulesza U, Bonifácio R. Understanding the exception handling strategies of Java libraries: An empirical study. In: Proc. of the 13th Working Conf. on Mining Software Repositories. Austin: IEEE, 2016. 212–222.

[85] Long J. Software reuse antipatterns. ACM SIGSOFT Software Engineering Notes, 2001, 26(4): 68–76. [doi: 10.1145/505482.505492]

[86] van Deursen A, Leon Moonen, van Den Bergh A, Kok G. Refactoring test code. In: Proc. of the 2nd Int'l Conf. on Extreme Programming and Flexible Processes in Software Engineering (XP). 2001. 92–95.

[87] Fenske W, Schulze S. Code smells revisited: A variability perspective. In: Proc. of the 9th Int'l Workshop on Variability Modelling of Software-intensive Systems. Hildesheim: ACM, 2015. 3–10. [doi: 10.1145/2701319.2701321]

[88] Marinescu R. Measurement and Quality in Object-oriented Design [Ph.D. Thesis]. Politehnica University of Timisoara, 2002.

[89] Cruzes DS, Dybå T. Recommended steps for thematic synthesis in software engineering. In: Proc. of the 2011 Int'l Symp. on Empirical Software Engineering and Measurement. Banff: IEEE, 2011. 275–284. [doi: 10.1109/ESEM.2011.36]

[90] Cohen J. A coefficient of agreement for nominal scales. Educational and Psychological Measurement, 1960, 20(1): 37–46. [doi: 10.1177/001316446002000104]

[91] Marinescu R. Detection strategies: Metrics-based rules for detecting design flaws. In: Proc. of the 20th IEEE Int'l Conf. on Software Maintenance. Chicago: IEEE, 2004. 350–359. [doi: 10.1109/ICSM.2004.1357820]

[92] Maiga A, Ali N, Bhattacharya N, Sabané A, Guéhéneuc YG, Antoniol G, Aïmeur E. Support vector machines for anti-pattern detection. In: Proc. of the 27th IEEE/ACM Int'l Conf. on Automated Software Engineering. Essen: ACM, 2012. 278–281. [doi: 10.1145/2351676.2351723]

[93] Maiga A, Ali N, Bhattacharya N, Sabané A, Guéhéneuc YG, Aimeur E. SMURF: A SVM-based incremental anti-pattern detection approach. In: Proc. of the 18th Working Conf. on Reverse Engineering. Kingston: IEEE, 2012. 466–475. [doi: 10.1109/WCRE.2012.56]

[94] Moha N, Guéhéneuc YG, Leduc P. Automatic generation of detection algorithms for design defects. In: Proc. of the 21st IEEE/ACM Int'l Conf. on Automated Software Engineering. Tokyo: IEEE, 2006. 297–300. [doi: 10.1109/ASE.2006.22]

[95] Bavota G, Oliveto R, Gethers M, Poshyvanyk D, De Lucia A. Methodbook: Recommending move method refactorings via relational topic models. IEEE Trans. on Software Engineering, 2014, 40(7): 671–694. [doi: 10.1109/TSE.2013.60]

[96] Fontana FA, Pigazzini I, Roveda R, Zanoni M. Automatic detection of instability architectural smells. In: Proc. of the 2016 IEEE Int'l Conf. on Software Maintenance and Evolution. Raleigh: IEEE, 2016. 433–437. [doi: 10.1109/ICSME.2016.33]

[97] Hecht G. An approach to detect android antipatterns. In: Proc. of the 37th Int'l Conf. on Software Engineering. Florence: ACM, 2015. 766–768. [doi: 10.5555/2819009.281916]

[98] Hecht G, Rouvoy R, Moha N, Duchien L. Detecting antipatterns in Android apps. In: Proc. of the 2nd ACM Int'l Conf. on Mobile Software Engineering and Systems. Florence: ACM, 2015. 148–149. [doi: 10.5555/2825041.2825078]

[99] Tourwe T, Mens T. Identifying refactoring opportunities using logic meta programming. In: Proc. of the 7th European Conf. on Software Maintenance and Reengineering. Benevento: IEEE, 2003. 91–100. [doi: 10.1109/CSMR.2003.1192416]

[100] Liu HT, Liu W, Hu ZG. Research of automatic detection for data clumps based on abstract syntax tree. Computer Applications and Software, 2017, 34(1): 15–20 (in Chinese with English abstract). [doi: 10.3969/j.issn.1000-386x.2017.01.003]

[101] Counsell S, Hierons RM, Hamza H, Black S, Durrand M. Is a strategy for code smell assessment long overdue? In: Proc. of the 2010 ICSE Workshop on Emerging Trends in Software Metrics. Cape Town: ACM, 2010. 32–38. [doi: 10.1145/1809223.1809228]

[102] Santos JAM, De Mendonça MG, Silva CVA. An exploratory study to investigate the impact of conceptualization in god class detection. In: Proc. of the 17th Int'l Conf. on Evaluation and Assessment in Software Engineering. Porto de Galinhas: ACM, 2013. 48–59. [doi: 10.1145/2460999.2461007]

[103] Hozano M, Antunes N, Fonseca B, Costa E. Evaluating the accuracy of machine learning algorithms on detecting code smells for different developers. In: Proc. of the 19th Int'l Conf. on Enterprise Information Systems. Porto: ICEIS, 2017. 474–482. [doi: 10.5220/0006338804740482]

[104] Oliveira R. When more heads are better than one? Understanding and improving collaborative identification of code smells. In: Proc. of the 38th Int'l Conf. on Software Engineering Companion. Austin: ACM, 2016. 879–882. [doi: 10.1145/2889160.2889272]

[105] De Mello RM, Oliveira R, Garcia A. On the influence of human factors for identifying code smells: A multi-trial empirical study. In: Proc. of the 2017 ACM/IEEE Int'l Symp. on Empirical Software Engineering and Measurement. Toronto: IEEE, 2017. 68–77. [doi: 10.1109/ESEM.2017.13]

[106] Oliveira R, Sousa L, De Mello R, Valentim N, Lopes A, Conte T, Garcia A, Oliveira E, Lucena C. Collaborative identification of code smells: A multi-case study. In: Proc. of the 39th IEEE/ACM Int'l Conf. on Software Engineering: Software Engineering in Practice Track. Buenos Aires: IEEE, 2017. 33–42. [doi: 10.1109/ICSE-SEIP.2017.7]

[107] Oliveira R, De Mello R, Fernandes E, Garcia A, Lucena C. Collaborative or individual identification of code smells? On the effectiveness of novice and professional developers. Information and Software Technology, 2020, 120: 106242. [doi: 10.1016/j.infsof.2019.106242]

[108] Wang YJ, Hu SY, Yin LF, Zhou XC. Using code evolution information to improve the quality of labels in code smell datasets. In: Proc. of the 42nd Annual Computer Software and Applications Conf. Tokyo: IEEE, 2018. 48–53. [doi: 10.1109/COMPSAC.2018.00015]

[109] Di Nucci D, Palomba F, Tamburri DA, Serebrenik A, De Lucia A. Detecting code smells using machine learning techniques: Are we

there yet? In: Proc. of the 25th IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering. Campobasso: IEEE, 2018. 612–621. [doi: 10.1109/SANER.2018.8330266]

[110] Tsantalis N, Chaikalis T, Chatzigeorgiou A. JDeodorant: Identification and removal of type-checking bad smells. In: Proc. of the 12th European Conf. on Software Maintenance and Reengineering. Athens: IEEE, 2008. 329–331. [doi: 10.1109/CSMR.2008.4493342]

[111] Fontana FA, Spinelli S. Impact of refactoring on quality code evaluation. In: Proc. of the 4th Workshop on Refactoring Tools. Waikiki: ACM, 2011. 37–40. [doi: 10.1145/1984732.1984741]

[112] Olbrich SM, Cruzes DS, Sjøberg DIK. Are all code smells harmful? A study of god classes and brain classes in the evolution of three open source systems. In: Proc. of the IEEE Int'l Conf. on Software Maintenance. Timisoara: IEEE, 2010. 1–10. [doi: 10.1109/ICSM.2010.5609564]

[113] Sjøberg DIK, Yamashita A, Anda BCD, Mockus A, Dybå T. Quantifying the effect of code smells on maintenance effort. IEEE Trans. on Software Engineering, 2013, 39(8): 1144–1156. [doi: 10.1109/TSE.2012.89]

[114] Soltanifar B, Akbarinasaji S, Caglayan B, Bener AB, Filiz A, Kramer BM. Software analytics in practice: A defect prediction model using code smells. In: Proc. of the 20th Int'l Database Engineering & Applications Symp. Montreal: ACM, 2016. 148–155. [doi: 10.1145/2938503.2938553]

[115] Pietrzak B, Walter B. Leveraging code smell detection with inter-smell relations. In: Proc. of the 7th Int'l Conf. on Extreme Programming and Agile Processes in Software Engineering. Oulu: Springer, 2006. 75–84. [doi: 10.1007/11774129_8]

[116] Tufano M, Palomba F, Bavota G, Oliveto R, Di Penta M, De Lucia A, Poshyvanyk D. When and why your code starts to smell bad. In: Proc. of the 37th IEEE/ACM IEEE Int'l Conf. on Software Engineering. Florence: IEEE, 2015. 403–414. [doi: 10.1109/ICSE.2015.59]

[117] Walter B, Alkhaeir T. The relationship between design patterns and code smells: An exploratory study. Information and Software Technology, 2016, 74: 127–142. [doi: 10.1016/j.infsof.2016.02.003]

[118] Olbrich S, Cruzes DS, Basili V, Zazworka N. The evolution and impact of code smells: A case study of two open source systems. In: Proc. of the 3rd Int'l Symp. on Empirical Software Engineering and Measurement. Lake Buena Vista: IEEE, 2009. 390–400. [doi: 10.1109/ESEM.2009.5314231]

[119] Liu H, Yang LM, 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. and the ACM SIGSOFT Symp. on The Foundations of Software Engineering. Amsterdam: ACM, 2009. 265–268. [doi: 10.1145/1595696.1595738]

[120] Gao Y, Liu H, Fan XZ, Niu ZD, Shao WZ. Resolution sequence of bad smells. Ruan Jian Xue Bao/Journal of Software, 2012, 23(8): 1965–1977 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4152.htm [doi: 10.3724/SP.J.1001.2012.04152]

[121] Vidal SA, Marcos C, Díaz-Pace JA. An approach to prioritize code smells for refactoring. Automated Software Engineering, 2016, 23(3): 501–532. [doi: 10.1007/s10515-014-0175-x]

[122] Sae-Lim N, Hayashi S, Saeki M. Context-based code smells prioritization for prefactoring. In: Proc. of the 24th IEEE Int'l Conf. on Program Comprehension. Austin: IEEE, 2016. 1–10. [doi: 10.1109/ICPC.2016.7503705]

[123] Sae-Lim N, Hayashi S, Saeki M. Context-based approach to prioritize code smells for prefactoring. Journal of Software: Evolution and Process, 2018, 30(6): e1886. [doi: 10.1002/smr.1886]

[124] Sae-Lim N, Hayashi S, Saeki M. How do developers select and prioritize code smells? A preliminary study. In: Proc. of the 2017 IEEE Int'l Conf. on Software Maintenance and Evolution. Shanghai: IEEE, 2017. 484–488. [doi: 10.1109/ICSME.2017.66]

[125] Rani A, Chhabra JK. Prioritization of smelly classes: A two phase approach (reducing refactoring efforts). In: Proc. of the 3rd Int'l Conf. on Computational Intelligence & Communication Technology. Ghaziabad: IEEE, 2017. 1–6. [doi: 10.1109/CIACT.2017.7977311]

[126] Mannan UA, Ahmed I, Almurshed RAM, Dig D, Jensen C. Understanding code smells in Android applications. In: Proc. of the 2016 IEEE/ACM Int'l Conf. on Mobile Software Engineering and Systems. Austin: IEEE, 2016. 225–236. [doi: 10.1109/MobileSoft.2016.048]

[127] Grano G, Palomba F, Di Nucci D, De Lucia A, Gall HC. Scented since the beginning: On the diffuseness of test smells in automatically generated test code. Journal of Systems and Software, 2019, 156: 312–327. [doi: 10.1016/j.jss.2019.07.016]

[128] Rasool G, Arshad Z. A review of code smell mining techniques. Journal of Software: Evolution and Process, 2015, 27(11): 867–895. [doi: 10.1002/smr.1737]

[129] Gupta A, Suri B, Misra S. A systematic literature review: Code bad smells in Java source code. In: Proc. of the 17th Int'l Conf. on Computational Science and Its Applications. Trieste: Springer, 2017. 665–682. [doi: 10.1007/978-3-319-62404-4_49]

[130] Palomba F, Zanoni M, Fontana FA, De Lucia A, Oliveto R. Toward a smell-aware bug prediction model. IEEE Trans. on Software Engineering, 2019, 45(2): 194–218. [doi: 10.1109/TSE.2017.2770122]

[131] Fontana FA, Ferme V, Marino A, Walter B, Martenka P. Investigating the impact of code smells on system's quality: An empirical study

on systems of different application domains. In: Proc. of the 2013 IEEE Int'l Conf. on Software Maintenance. Eindhoven: IEEE, 2013. 260–269. [doi: 10.1109/ICSM.2013.37]

[132] Haque MS, Carver J, Atkison T. Causes, impacts, and detection approaches of code smell: A survey. In: Proc. of the 2018 ACMSE Conf. Richmond: ACM, 2018. 25. [doi: 10.1145/3190645.3190697]

[133] CppDepend. 2021. https://www.cppdepend.com/

[134] Sonarsource. 2021. https://www.sonarsource.com/

[135] Designite. 2021. https://www.designite-tools.com/

[136] NDepend. 2021. https://www.ndepend.com/

[137] PMD. 2021. https://pmd.github.io/pmd-6.29.0/index.html

[138] SDMetrics. 2021. https://www.sdmetrics.com/index.html

附中文参考文献:

[9] 卜依凡, 刘辉, 李光杰. 一种基于深度学习的上帝类检测方法. 软件学报, 2019, 30(5): 1359−1374. http://www.jos.org.cn/1000-9825/5724.htm [doi: 10.13328/j.cnki.jos.005724]

[100] 刘宏韬, 刘伟, 胡志刚. 基于抽象语法树的数据泥团自动检测研究. 计算机应用与软件, 2017, 34(1): 15–20. [doi: 10.3969/j.issn.1000-386x.2017.01.003]

[120] 高原, 刘辉, 樊孝忠, 牛振东, 邵维忠. 代码坏味的处理顺序. 软件学报, 2012, 23(8): 1965–1977. http://www.jos.org.cn/1000-9825/4152.htm [doi: 10.3724/SP.J.1001.2012.04152]

田迎晨(1997－), 男, 硕士, 主要研究领域为软件质量与重构, 开源软件生态系统.

李光杰(1980－), 女, 博士, 副研究员, 主要研究领域为软件工程, 代码质量, 软件重构, 开源软件.

李柯君(1997－), 男, 硕士, 主要研究领域为软件质量与重构.

张宇霞(1992－), 女, 博士, 特别副研究员, CCF 专业会员, 主要研究领域为开源软件生态系统, 软件仓库, 数据挖掘, 实证研究.

王太明(1996－), 男, 博士生, CCF 学生会员, 主要研究领域为软件自动化重构, 基于深度学习的软件工程.

刘辉(1978－), 男, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为智能软件工程, 数据挖掘, 深度学习.

焦青青(1996－), 女, 硕士, 主要研究领域为软件重构, 软件质量.