软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software [doi: 10.13328/j.cnki.jos.006944]

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

软件中代码注释质量问题研究综述*

王 潮, 徐卫伟, 周明辉

(北京大学 计算机学院, 北京 100871)

通信作者: 周明辉, E-mail: zhmh@pku.edu.cn

摘 要: 代码注释作为辅助软件开发群体协作的关键机制, 被开发者所广泛使用以提升开发效率. 然而, 由于代码注释并不直接影响软件运行, 使其常被开发者忽视, 导致出现代码注释质量问题, 进而影响开发效率. 代码注释中存在的质量问题会影响开发者理解相关代码, 甚至可能产生误解从而引入代码缺陷, 因此这一问题受到研究者的广泛关注. 采用系统文献调研, 对近年来国内外学者在代码注释质量问题上的研究工作进行系统的分析. 从代码注释质量的评价维度、度量指标以及提升策略这 3 方面总结研究现状, 并提出当前研究所存在的不足、挑战以及建议.

关键词: 代码注释; 软件文档; 自然语言处理; 机器学习

中图法分类号: TP311

中文引用格式: 王潮, 徐卫伟, 周明辉. 软件中代码注释质量问题研究综述. 软件学报. http://www.jos.org.cn/1000-9825/6944.htm

英文引用格式: Wang C, Xu WW, Zhou MH. Survey on Quality of Software Code Comments. Ruan Jian Xue Bao/Journal of Software (in Chinese). http://www.jos.org.cn/1000-9825/6944.htm

Survey on Quality of Software Code Comments

WANG Chao, XU Wei-Wei, ZHOU Ming-Hui

(School of Computer Science, Peking University, Beijing 100871, China)

Abstract: As an essential mechanism of group collaboration in software development, code comments are widely used by developers to improve the efficiency of specific developing tasks. However, code comments do not directly affect the software operation, and developers often ignore them, which leads to poor quality of code comments and affects development efficiency. Quality issues of code comments hinder code understanding, bring misunderstanding, or even introduce bugs, which receive widespread attention from researchers. This study systematically analyzes the research work of global scholars on quality issues of code comments in recent years by literature review. It also summarizes related studies in three aspects: evaluation dimensions of code comment quality, indicators of code comment quality, and strategies to promote code comment quality and points out shortcomings, challenges, and suggestions for the current research.

Key words: code comment; software documentation; natural language processing; machine learning

开发者在软件开发的过程中, 通常采用为软件系统中的源代码书写代码注释的方式, 来帮助有着不同背景的协作者准确且高效地理解程序代码的运行逻辑与当前的开发状态 (甚至帮助未来的自己理解遗忘的复杂代码). 代码注释通常记录了特定软件代码的相关信息, 为不同背景的开发者提供相关参考. 软件系统代码文件中的代码注释, 被认为是软件文档的一种重要形式[1], 广泛存在于几乎所有的软件系统之中[2,3], 是代码开发中普遍遵循的实践. 代码注释通常以自然语言的形式书写, 包含了代码运行逻辑、参数传递规范、异常错误信息等多种多样的与软件代码相关的信息[4], 以帮助开发者理解相关代码. 同时, 代码注释中也包含了代码作者、待开发任务、代码引入版本等与软件开发活动相关的信息[4], 来辅助开发者就当前开发活动进度进行沟通.

代码注释包含丰富内容, 在群体协作的过程中为开发者完成软件维护、代码复用等开发任务提供了丰富且宝

* 基金项目: 国家自然科学基金 (61825201)
收稿时间: 2022-07-24; 修改时间: 2022-12-24; 采用时间: 2023-04-03; jos 在线出版时间: 2023-08-30

贵的信息[5], 同时也为来自不同背景的开发者理解其他开发者所书写的代码提供了十分重要的帮助[6]. 尤其是清晰、完整的代码注释, 能够帮助开发者高效且准确地理解当前代码的运行逻辑与开发现状, 从而辅助开发者高效完成相关开发任务. 因此, 规模较大的软件项目通常会要求开发者在软件开发的过程中规范准确地书写代码注释, 正如 Google 在其发布的指导软件开发的编码风格向导中所说: “代码注释对于维持代码的可读性十分重要.”[7]

然而, 代码注释并不会直接影响代码的运行——即使代码注释出现了质量问题, 相关代码仍然可以正常运行. 因此, 在软件开发的过程中, 开发者对于代码注释的重视程度往往低于相关软件代码, 使得代码注释质量问题频繁出现. 代码注释质量问题指代码注释的内容、格式、位置等方面存在缺陷, 使得开发者不能高效地利用代码注释完成相关开发活动. 代码注释质量问题有着多种表现形式, 例如代码注释内容与代码实际运行逻辑不一致、代码注释缺失、代码注释包含冗余信息等. 代码注释的质量问题给群体协作中诸多开发任务, 例如软件理解、软件维护、代码复用等, 带来了挑战. 开发者因为代码注释存在质量问题需要花费更多的时间与精力在相关的软件开发任务中, 甚至可能因为注释问题产生对软件代码的错误理解从而引入代码缺陷[8]. 例如在 Mozilla 社区中, 因为两条面向方法阻塞的错误代码注释对开发者产生的误解, 引发了包括问题报告#355409 在内的一系列代码缺陷 (https://bugzilla-dev.allizom.org/show_bug.cgi?id=363114). 而在维基百科对于代码缺陷 (software bug) 的定义中 (https://en.wikipedia.org/wiki/Software_bug), 错误或过时的代码注释本身就被认为是一种代码缺陷.

有鉴于此, 当前已有大量工作面向代码注释的质量问题开展了研究, 提出对代码注释质量进行评价的指标, 并提出了自动化方法辅助开发者提升注释质量. 为了系统化地理解当前软件开发过程中代码注释可能存在的质量问题, 以及代码注释质量的提升策略, 本文对近 20 年来的相关研究文献进行了收集与分析. 本文回答了如下问题: (1) 当前文献从什么维度对代码注释质量进行了评价? (2) 当前文献使用了何种指标对代码注释质量进行度量? (3) 当前文献提出了何种策略来提升代码注释质量?

本文接下来从代码注释的质量问题展开, 对代码注释质量问题进行定义, 分析并介绍了代码注释质量的评价维度、度量指标, 以及提升代码注释质量的方法. 本文第 1 节定义代码注释质量问题, 并介绍如何对相关文献进行收集、筛选与分析. 第 2 节对相关文献中涉及的对代码注释质量进行评价的维度进行总结. 第 3 节对相关文献中用于度量代码注释质量的指标进行了比较与总结. 第 4 节对相关文献提出的自动提升代码注释质量的策略进行了总结. 第 5 节通过对相关文献的总结, 指出代码注释质量问题相关尚待解决的问题与挑战, 并提出建议. 第 6 节对全文进行总结.

1 代码注释质量问题定义

软件开发过程中, 代码注释可能面向多种粒度的代码, 例如文件、类、方法、代码片段等. 代码注释也通常包含着多种类型的信息[4], 用于代码理解、代码维护等多种开发任务之中. 同时, 代码注释通常以自然语言书写, 自然语言表达的多样性使得代码注释的性质更为复杂. 然而, 由于代码注释并不直接影响软件代码运行, 代码注释中经常存在质量问题, 使得开发者难以利用并高效完成相关开发任务.

代码注释通常被认为由内容、格式、位置这 3 方面要素构成[9]. 因此, 若代码注释的内容、格式、位置方面存在缺陷, 使得开发者难以高效且准确地从代码注释中获取有效信息来完成相关开发任务, 此类代码注释的缺陷被称为代码注释质量问题.

代码注释的内容指代码注释内包含的相关文本信息. 大型软件项目发布的编码指南通常会要求代码注释内的文本信息能够准确地表达对应代码的相关信息 (例如代码的引入目的、运行逻辑), 或者是当前开发状态 (例如待完成的任务). 作为开发者之间沟通交流的渠道, 一方面, 代码注释的内容应该精准、全面地传达开发者沟通交流时需要传达的信息; 另一方面, 代码注释在传递表达相关信息时应该以精简且易于理解的方式. 这要求代码注释的内容不应传达错误信息或是遗漏关键信息, 与此同时, 代码注释的内容也不应该过于繁冗和复杂, 使得读者难以快速获取其关键核心信息.

代码注释的格式包括代码注释的缩进、内容组织以及语言风格等. 通常来说, 大型项目内会发布编码指南来

规范代码注释的格式[10], 例如代码注释单行的最大字符数、Javadoc 中相关标签的排列顺序、方法注释中对方法功能的描述使用第三人称祈使句等. 同时, 大型项目还通常使用静态检查工具来自动化地检测代码注释格式上的问题[10]. 此类工具一般基于文本的正则匹配, 因此只涉及格式, 而很少涉及代码注释的内容. 一致且清晰的代码注释格式能够使开发者快速地获取到其需求的信息, 而混乱的代码注释格式不利于项目内编码规范的统一管理, 也使开发者无法快速捕捉到关键信息. 此类问题本文也同样视作代码注释的质量问题.

代码注释的位置指代码注释在代码文件中所处的位置. 代码注释位置相关的质量评价, 一方面与必要代码注释在特定位置是否存在有关, 如软件项目中较为复杂的或经常被调用的接口方法, 往往需要通过为其书写方法注释的方式来提醒开发者如何正确调用特定方法. 关键代码注释的缺失可能降低开发者的开发效率, 其缺失同样属于代码注释质量问题. 另一方面, 代码注释通常描述临近代码行[9], 若代码注释并没有合理书写在对应代码的附近, 可能导致开发者无法快速定位获取有效信息, 同样属于代码注释的质量问题.

为了对代码注释的质量问题展开系统化的分析, 本文在 2023 年 1 月收集了与代码注释质量问题相关的研究文献. 为了获得相关的英文文献, 首先通过多种渠道建立了初始论文集合: (1) 使用“code comment quality”作为关键词, 在学术论文搜索引擎 Google Scholar 进行搜索, 收集前 10 页的所有已发表论文, 阅读其标题、摘要及引言部分, 并保留与代码注释相关的论文; (2) 使用 the ACM Guide to Computing Literature 这一出版文献数据库, 对所有符合下述条件的文献进行了收集——文章类型为“研究论文”, 摘要中包含“code comment”这一关键词, 发表在软件工程相关会议与期刊, 发表时间在 2012–2022 年内. 基于这一初始数据集, 本文使用“滚雪球”[11]的文献收集方式, 通过阅读获得论文的引用文献, 保留其与代码注释质量问题这一主题相关的文献, 迭代地对当前文献集合进行扩充. 对于中文文献, 使用知网对中文文献进行了检索, 收集了所有发表在中国计算机学会推荐目录中的 A 类中文学术期刊上, 且标题、摘要或主题中包含“代码注释”这一关键词的中文文献.

接着作者对以上收集得到的所有文献进行了筛选, 得到对代码注释质量问题进行了分析讨论或提出解决方法的所有相关文献. 两位作者分别阅读所有收集文献的摘要部分, 并独立判断其是否涉及代码注释质量问题这一主题. 若两位作者都认为给定文献是相关的, 则其得以保留.

最后本文一共获得了 48 篇英文论文和 2 篇中文论文. 如图 1 所示, 这些论文的发表年份分布在 2006–2022 年, 且近年来研究者对代码注释质量问题展现出了较高的热情. 图 2 展示了收集得到的文献所发表的学术会议或期刊在中国计算机学会推荐目录中所属的分类. 可以看到, 大部分论文都是发表于 A 类和 B 类的会议或期刊上, 包括 ICSE、FSE、ASE、TOSEM 等软件工程著名会议与期刊.

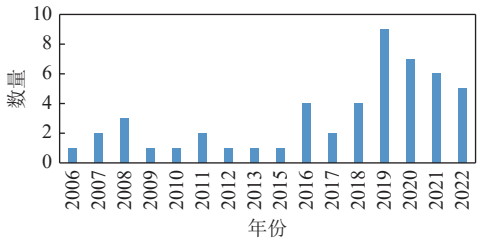

图 1 收集文献的发表年份分布

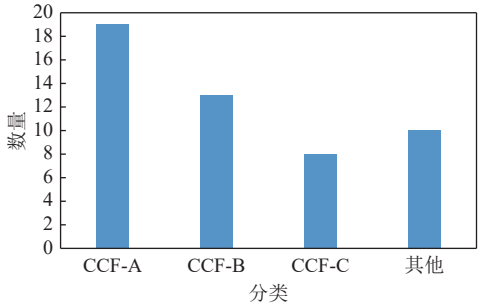

图 2 收集文献发表学术会议或期刊所属分类

为了系统化的理解代码注释质量问题, 本文作者对获取的论文进行了总结, 回答了以下 3 个问题: (1) 当前文献从什么维度对代码注释进行了评价与分析; (2) 当前文献使用了什么指标对代码注释质量进行度量; (3) 当前文献提出了什么策略来提升代码注释质量.

本文的主要贡献如下: 通过对研究代码注释质量问题的相关文献进行系统性的收集与分析, 总结了代码注释质量的评估维度, 度量技术, 以及提升方法, 并探讨了相关研究中存在的不足, 为未来的研究方向提供思路.

2 代码注释质量的评价维度

为了总结当前文献对代码注释的优劣进行评判的标准, 本文作者对所有收集得到的相关文献进行分析, 对相关文献中面向代码注释质量进行评价的句子进行摘取. 部分文献通过举例论证、问卷、访谈等定性方法对代码注释相关质量问题进行了描述与论证, 而部分文献则是通过收集相关数据, 定量地对代码注释存在的质量问题进行分析. 通过对摘取的描述代码注释质量的句子进行主题分析, 本文对代码注释质量的评价维度进行了总结, 发现当前文献对代码注释质量问题的分析主要集中在存在性、一致性、全面性、重要性、可读性、时效性、关联性等维度. 下文将介绍上述评价维度, 并对当前文献中的相关分析成果进行阐述.

2.1 存在性

代码注释的存在性指必要代码注释是否缺失——对于存在着复杂的运行逻辑的代码片段, 解释其运行逻辑的代码注释是必要的. 由于保证开发进度等原因, 开发者有时会忽视代码注释的书写, 使得必要的代码注释存在缺失的问题. 代码注释的存在性对于软件的可读性[12]与可维护性[13]至关重要, 因为关键代码片段的代码注释的缺失会使得开发者花费更多时间精力理解相关代码片段, 从而降低开发者完成相关开发任务的效率[14].

与代码注释存在性相关的质量问题在软件开发中广泛存在. Hu 等人[15]对 720 位有丰富软件开发经验的开发者开展了问卷调研, 结果显示开发者认为在软件开发过程中遇到的影响最大的代码注释质量问题为代码注释缺失 (lack of comments), 69% 的开发者认为代码注释缺失会对软件开发带来影响. 这一比例远超其他代码注释质量问题, 例如信息量缺失的代码注释 (generic comments, 62%), 过时代码注释 (outdated comments, 47%), 不一致代码注释 (inconsistent comments, 31%), 冗余代码注释 (redundant comments, 27%) 和过长注释 (too long comments, 16%). 代码注释缺失阻碍了开发者理解相关软件代码. 根据受访者的反馈, 代码注释的缺失不仅会使得相关的软件源代码无法被其他开发者理解, 甚至代码作者在一段时间后也可能无法理解自己曾写下的代码. 在没有代码注释的情况下, 为了理解代码, 开发者通常会通过阅读源代码和求助于外部信息资源的方式来获取相关信息[16], 然而这一过程通常是耗时耗力且易出错的.

为了探究代码注释缺失出现的原因, 相关工作通常使用代码注释密度作为度量指标, 分析何种因素会对代码注释存在性产生影响. 代码注释密度指代码文件中代码注释行数与总代码行数的比值, 其数值越低说明相关代码文件中可能出现代码注释缺失问题的可能性更高. 表 1 总结了相关文献中发现的与代码注释存在性的相关因素和无关因素. 相关文献[2,3,17−19]普遍发现随着项目年龄的增加, 代码注释密度会有显著的下降, 这也意味着代码注释缺失的问题可能随着项目的不断开发而逐渐严重. 对上述现象的产生, Fluri 等人[17]通过对代码与代码注释共同演化的研究给出了解释, 他们发现新引入的代码中接近一半没有被注释, 这意味着开发者可能对于新引入代码的注释情况有所忽视. 另一方面, He 等人[3]对 150 个开源软件项目的分析, 发现代码注释的存在性与项目主要使用的编程语言和项目的目的有关. 使用 Python 和 Java 的项目相较使用 C++, JavaScript 和 Go 的项目有着显著更高的代码注释密度. 他们推测这与不同编程语言的不同编码实践和工具使用情况有关. 同时, 以教育为目的软件项目有着显著最高的代码注释密度, 以复用为目的的框架、第三方库等项目次之, 直接供用户使用的应用项目有着最低的代码注释密度. 这一现象说明当开发者有想要读者理解相关代码的主观意图时, 会更有意愿书写代码注释. 同时相关文献也发现代码注释的存在性, 即代码注释密度, 与项目的团队规模 (提交代码人数) 和文件规模 (文件总行数) 并不显著相关[2,3].

表 1 影响项目内代码注释存在性的相关因素和无关因素

相关因素	无关因素
项目年龄[2,3,17−19]	项目开发团队人数[2,3]
编程语言[3]	项目文件规模[2]
项目目的[3]	

为了解决代码注释缺失这一质量问题, 首先需要了解代码文件中具有何种特征的代码需要代码注释. 这一问题的回答能为检测代码注释存在性的自动化工具奠定理论基础. 针对这一关键问题, 当前文献面向开发者以问卷、访谈的形式开展了研究. Sun 等人[20]面向方法级别的代码注释提出了两点假设, 即超过 30 行的方法更可能需要代码注释, 因为开发者对于体量较大的复杂方法理解存在困难; 以及内部代码调用了超过 3 个外部 API 的方法更可能需要代码注释, 因为开发者需要额外的知识来理解此类方法的功能. 经过基于真实数据的面向 20 位开发者的问卷调研他们验证了这两点假设, 即方法内代码行数及调用外部 API 数量可以被用于评价特定方法是否需要代码注释. Hu 等人[15]通过对开发者的问卷和访谈调研, 对开发者对于不同粒度下需要代码注释的代码类型进行了调研.

● 对于类级别的注释, 91% 的受访者认为有着复杂逻辑的难以理解的类需要注释, 而这样的类通常有以下 3 个特点: (1) 有着较长的代码长度; (2) 有较多的循环和条件控制语句 (例如 if 和 switch 语句); (3) 有较多 API 调用. 与此同时, 以下 3 种类型的类的代码对于绝大多数开发者来说也需要代码注释: (1) 使用了特殊解决方案和有着特殊设计算法的类 (classes with design patterns, 86%); (2) 提供了许多方法给其他类且经常在代码仓库中经常被复用的类 (utility classes, 85%), 这些类需要注释来减轻其他开发者进行代码复用的负担; (3) 有着开发者自己定义的异常的类 (classes with user-defined exceptions, 82%), 需要在注释里对引发异常的条件进行说明.

● 对于方法级别的注释, 受访者认为以下类型的方法需要代码注释: (1) 与上述复杂类类似的复杂方法 (complex methods); (2) 自解释性不足的方法 (non self-explanatory methods); (3) 使用了特别的算法或者可能给使用者带来困惑的易出错的方法 (tricky methods); (4) 包含着项目的核心逻辑和算法的方法 (key logic or algorithm methods); (5) 经常被其他开发者调用的接口方法 (methods in interfaces).

● 对于语句级别的注释, 受访者认为应该为以下类型的语句书写代码注释: (1) 复杂代码 (comlex code statements, 93%); (2) 处理特定代码逻辑的语句 (special statements, 89%), 例如仅接受特定格式字符串的代码语句; (3) 易出错的代码语句 (tricky statements, 89%); (4) 硬编码 (harding coding, 88%); (5) 正则表达式 (regular expressions, 81%); (6) Lambda 表达式 (Lambda expressions, 73%).

以上面向开发者的调研揭示了开发者代码注释实践中存在着共识, 例如复杂的代码是几乎所有开发者都认为需要必要的代码注释来帮助开发者理解对应代码的编写与运行逻辑, 这意味着基于一定的规则来检测代码注释缺失问题是可能的. 然而, 对于需要代码注释的代码具有何种特征的评价标准往往是模糊而主观的. 以需要代码注释的复杂方法为例, 特定方法需要达到何种复杂度才可以被视作是复杂方法并不清晰. 在不同项目开发上下文下, 都可能有着不一样的理解.

为了进一步具体定位到代码文件中需要代码注释的代码, 当前文献[21−23]通过提取海量开源软件项目中代码文件信息, 通过提取有相关注释和缺失注释的代码相关的结构、语法、语义信息作为特征, 使用机器学习模型进行训练并预测代码文件中需要代码注释的位置. 本文第 4 节对此类方法进行了总结与对比.

2.2 一致性

代码注释的一致性指代码注释的内容和其对应代码的真实运行逻辑是否一致. 若代码注释与对应代码的实际逻辑存在不一致, 说明此代码注释存在一致性上的质量问题. 代码注释内容与对应代码发生不一致主要有两类原因, 一是因为开发者在书写代码注释时由于自身疏忽或对代码理解不足而写下了错误的内容, 使得存在不一致的代码注释在代码文件中被引入; 另一类更为常见的原因则是在代码演化的过程中, 开发者对代码注释的维护不到位, 使得相关代码被更新而对应的代码注释却没有被更新. Fluri 等人[17]对代码和相关代码注释的共同演化进行分析后发现, 以版本发布为粒度, 大约有 3%–10% 的代码注释的更新滞后于对应代码, 代码注释更新的滞后性将会导致大量的不一致代码注释.

不一致的代码注释会增加开发者理解相关代码运行逻辑的时间与精力成本, 甚至可能因为错误的理解而在代码系统中引入代码缺陷. Ibrahim 等人[8]将代码注释的更新信息加入到了两个常用的代码缺陷预测模型中, 来检验代码注释的一致性与软件系统中代码缺陷的相关性. 他们发现不一致的代码注释确实更有可能在代码文件中引入代码缺陷, 尤其对于在开发历史中一直保持正确的代码注释, 如果突然出现未及时更新而引入不一致的情况, 则有

更高的风险使得开发者在相关代码中引入代码缺陷. 因此, 为了保持软件系统的正确运行, 保证代码注释的一致性十分重要.

为了避免不一致代码注释这一质量问题的引入, 当前文献从不同角度开展了分析. 在软件开发的过程中, 并不是所有代码更改的场景都需要对代码注释进行更改, 因此一类研究分析了什么类型的代码更改需要伴随着相关代码注释的更改, 从而达到可以在特定场景下提醒开发者更新代码注释的目的. 表 2 总结了相关文献得到的影响是否触发代码注释变更的代码更改的相关特征. Malik 等人[14]将特定代码更改是否需要伴随代码注释更改这一问题抽象为了分类问题, 并提取了被修改方法相关特征 (characteristics of the modified function)、代码变更细节特征 (characteristics of the change) 和时间与作者相关特征 (time and code-ownership characteristics) 这 3 方面的特征, 使用随机森林作为分类器进行训练. 他们发现对于分类器影响最大的几个参数分别为依赖方法的比例、方法的存在年龄、修改控制语句的数目以及依赖方法的数目. 这意味着如果特定方法所调用的方法被大量修改, 或者该方法已经被引入很长一段时间, 或者该方法大量控制语句被修改, 则该方法对应的代码注释也很有可能被修改. Wen 等人[24]对触发代码注释更改的细粒度的代码变更进行了分析. 他们提取了更改代码的语法信息, 即代码更改部分在抽象语法树上所处的类型, 并分析了其与是否触发代码注释更改. 他们发现大约 13%–20% 的代码更改会触发代码注释更改, 且对变量声明语句和条件控制语句的修改更有可能触发对相关代码注释的更改, 因为此类修改更有可能对代码的运行逻辑造成影响. Sondhi 等人[25]对代码注释中非直接相关内容进行了分析, 即代码注释中关注于对应代码之外的内容, 例如指向外部资源的链接、对其他关联方法的功能解释等. 他们发现代码注释中非直接相关内容在代码演化过程中常被忽视, 从而导致不一致的代码注释质量问题. 在他们收集的数据集中, 13% 的代码注释中非直接相关内容的更新是滞后于对应代码更新的.

表 2　触发代码注释变更的代码更改特征

特征	含义
对应代码特征[14]	被修改方法的名字长度、 控制语句数量、内部注释数量、参数数量、调用方法数量等
代码注释特征[25]	代码注释是否包含非直接相关内容
代码变更特征[14,24]	代码变更的语句类型, 例如变更控制语句数量、变更的变量声明语句数量
变更作者、时间特征[14]	代码变更作者和代码初始作者是否匹配, 代码变更的时间, 变更代码的存在时间、距离上一次变更的时间等

另一类文献则是基于海量项目的开发数据, 使用预定义模板或者机器学习模型等方法, 对存在不一致的代码注释进行自动的检测[26–28]和更新[29–31]. 本文第 4 节会对此类技术进行系统性地总结.

2.3 重要性

代码注释的重要性指代码注释内容是否包含了重要的额外信息, 与此同时, 代码注释不应该包含冗余、无用信息干扰开发者快速定位到核心内容. 为了满足开发者对于重要性的要求, 代码注释需要满足两方面的条件, 一是代码注释中需要包含重要的额外信息, 而不是简单地复述代码中所包含的文本信息, 使得开发者能够真正地受益于特定代码注释; 二是代码注释中应避免冗余信息与无效信息, 使得开发者能够迅速地从代码注释中定位到有用的信息.

代码注释的重要性对于代码的可读性十分重要[12], 其常作为评价软件可读性的评价标准. 在重要性上存在质量问题的代码注释, 通常表现为缺少可以帮助到相关开发活动的重要信息、仅复述相关代码的显然信息、包含大量无效冗余信息等. 此类代码注释会阻碍开发者快速有效地定位到对相关开发活动有帮助的信息[15], 从而损害开发者进行相关开发活动的效率.

一方面, 相关文献讨论了什么样的代码注释是缺乏重要的额外信息的. Nurr 等人[32]认为若代码注释的内容与相对应的代码文本过于相似, 则可以认为其是冗余的, 并没有提供更多的相关信息; 若代码注释的内容与对应的代码文本完全无关, 则代码注释可能并没有很好地描述相关代码. Jabrayilzade 等人[33]认为如果代码注释仅仅在复述具有自解释性的代码的行为, 而没有提供额外的有用信息, 则认为此类注释是冗余的. 通过对 3 个项目的代码注释的人工检查, 他们发现此类冗余注释是行间注释最容易出现的质量问题, 约占代码注释的 41%. 为了度量相关代码

注释是否包含额外信息, 相关文献通常会使用代码注释与相关代码的文本相似度[32,34,35]以及代码注释的长度[32,34]作为度量指标.

另一方面, 相关文献讨论了代码注释中什么样的信息开发者认为是冗余与无效的. Jabrayilzade 等人[33]对行间注释质量问题的研究发现开发者认为代码的作者信息是冗余的, 因为这一信息可以从版本控制系统 (例如 Git) 中获取. 同时, 他们发现开发者认为用于分割代码文件的一类起美化作用的代码注释也是冗余的, 因为此类代码注释并没有提供任何信息. Wang 等人[10]对 600 个 Java 开源项目的编码指南中与代码注释相关的部分进行了收集与分析, 他们发现开发者在代码注释中是否应该引入作者信息、代码引入时间信息、待办注释、注释代码等内容存在争议, 例如部分项目会鼓励开发者引入作者、时间等信息, 也有许多项目禁止此类注释出现在代码提交中. 与此同时, 他们还发现部分开发者认为重写方法和测试用例也不需要相关代码注释.

2.4 全面性

代码注释的全面性指代码注释是否包含了对应代码的全部重要信息, 不应因为必要信息存在缺失而引起误解. 如果代码注释的内容有所遗漏, 缺失了理解相关代码的重要信息, 则认为此代码注释在全面性上存在着质量问题. 若代码注释在全面性上存在质量问题, 则开发者需要花费更多的时间和精力用于理解相关代码, 降低相关开发任务的效率.

为了理解代码注释的全面性, 相关文献对代码注释的种类进行了分析和总结, 并使用相关数据训练得到分类器对代码注释进行分类, 从而辅助开发者确认给定代码注释是否包含了全面的信息. Pascarella 等人[4]从 6 个开源软件项目中随机抽样了 2 000 个 Java 代码文件, 并提取其中包含的代码注释进行人工标注, 得到了如表 3 所示的代码注释内容的分类系统, 包含了 6 大类型、16 种子类的代码注释. 基于得到的分类系统, Pascarella 等人对代码注释文本提取了包含词汇数目、是否包含特定 Javadoc 标签等特征, 使用决策树算法构建了对代码注释内容进行分类的分类器, 并取得了较好的检验结果, 平均真正率 (true positive rate) 达到了 0.85. Rani 等人[36]沿用了表 3 所示分类系统, 并发现该分类系统可以同样适用于其他编程语言 (Smalltalk 和 Python) 的代码注释, 并通过新增代码注释的自然语言文本模式和词嵌入向量等特征, 提升了面向代码注释内容的分类器的分类效果.

表 3 代码注释内容的分类系统[4]

类型	子类	含义
目的	总结	简要描述相关代码运行行为的代码注释
	扩展	详细描述相关代码的实现细节的代码注释
	原因	描述相关代码中的设计、实现选择的理由的代码注释
提醒	废弃	描述已废弃的代码及替代方法的代码注释
	用例	描述如何使用特定方法的代码注释
	异常	描述代码可能抛出的异常信息的代码注释
开发中	待办	描述待完成的开发任务的代码注释
	待补全	空的或者仅包含框架的需要补充的代码注释
	注释代码	代码文件中被注释掉的源代码
工具	指令	服务于开发工具的代码注释
	格式	用于分割代码文件的起到分隔符作用的代码注释
元数据	许可证	描述软件许可证信息的代码注释
	作者	描述相关代码的作者信息的代码注释
	指针	包含指向外部资源的引用的代码注释, 如url链接
其他	自动生成	由相关工具自动生成的代码注释
	噪声	未包含有意义信息的代码注释

尽管已有工作在代码注释的分类上取得了进展, 但对于保证代码注释的全面性来说, 仍有关键问题尚待回答, 即对于给定方法, 什么类型的代码注释是需要的. Hu 等人[15]面向开发者的问卷调研揭示了开发者面向不同粒度的代码注释有着不同的内容上的期望. 对于类级别的代码注释, 开发者主要期待代码功能总结和使用方法; 对于方法

级别的代码注释, 除了代码功能总结和使用方法, 开发者还期望由输入输出、设计原理、使用样例等信息; 而对于语句级别的代码注释, 开发者主要期待设计原理、功能总结和相关提醒警告. 此外, 由于项目背景的不同以及特定代码的上下文语句不同, 开发者对于特定代码注释的期待也会有所差异. Hu 等人[15]总结了整体上开发者对代码注释的期待, 但对于具体场景下开发者对于代码注释的需求的细粒度的分析仍是缺失的.

2.5 可读性

代码注释的可读性指开发者在阅读代码注释时是否存在困难. 若代码注释是清晰易懂的, 则认为此代码注释在可读性上有着好的质量; 若代码注释因为格式混乱、语言复杂等原因使得开发者难以快速理解其含义, 则认为此代码注释在可读性上存在着质量问题. 当前文献从不同角度探索了影响代码注释可读性的因素.

首先, 为了提升代码注释的可读性, 代码注释应该遵循一致的规范与格式. Steidl 等人[34]将代码注释的一致性作为衡量代码注释质量的一项重要指标, 即代码注释应该由相同语言书写 (一般项目中都是使用英语), 且代码文件应该有统一的许可证. Wang 等人[10]发现开发者通常会通过发布编码指南和使用静态检查工具的方式来保证项目内的代码注释有着统一的格式. 他们也发现在一些提升代码注释质量的代码更改中, 开发者会通过统一代码注释中所使用的术语的方式, 来提升代码注释的可读性, 避免给开发者带来不必要的困惑.

其次, 为了提升代码注释的可读性, 代码注释应该尽量使用清晰、简单的语言, 避免模糊和复杂的语言. 相关文献通常使用用来量化自然语言可读性的指标来对代码注释进行量化分析, 例如 Dale-Chall 分数[32], Flesch-Kincaid 指标[37,38]等. Nurr 等人[32]使用 Dale-Chall 分数作为指标度量了开源软件项目与公司开发的商业软件项目中的代码注释可读性, 发现二者并没有显著区别. Scalabrino 等人[37]使用 Flesch-Kincaid 指标度量了代码注释的可读性, 并使用代码注释的可读性作为新加入的特征提升了当前代码可读性的评价模型.

2.6 时效性

代码注释的时效性主要面向服务于特定开发任务的代码注释, 如待办注释 (todo comment)、注释代码等. 这类代码注释因为其服务于特定开发任务, 因此需要随着相关开发任务的进度进行更新, 例如若待办注释中所包含的待开发的任务已完成或部分完成, 此待办注释也应该同时进行删除或更新, 避免给其他开发者带来不必要的困惑. 若此类代码注释没有及时随着相关开发任务被更新, 则认为该代码注释在时效性上存在质量问题. 下文讨论了相关文献中可能出现时效性相关质量问题的代码注释类型.

相关文献发现待办注释常出现时效性的问题. 待办注释常被开发者用来记录代码开发活动中未完成或值得注意的信息, 包含了开发者沟通、变更需求、已完成任务备注、开发中任务、特征需求、位置标记、标签等丰富类型[39]. 然而由于已完成任务的相关待办注释没有被及时删除、待办注释任务未受到开发者关注与重视或是待办注释模糊不清等原因[40], 常会使得待办注释存在时效性上的质量问题. 开发者认为这些过时的待办注释会使代码杂乱无章, 掩盖了更多有意义的信息, 甚至会对代码理解产生负面影响[40]. 为了避免这一问题的发生, 相关文献[41,42]通过对待办注释和相关代码进行语义上的分析, 通过启发式的规则自动检测过时的待办注释.

相关文献发现注释代码也常出现时效性问题. 注释代码指代码文件中被注释掉的代码, 因为种种原因这些代码得以保存而没有直接被删除. 然而, 由于开发活动的进展, 一些注释代码不再有意义. Wang 等人[10]发现开发者会认为长期存在的注释代码是冗余无用的, 并会进行批量删除. 同时因为版本控制系统提供了便捷的访问历史代码版本的功能, 一些项目会禁止在代码提交中引入新的注释代码.

相关文献发现代码注释中的链接也会出现时效性问题. Hata 等人[43]在 2 500 余个 GitHub 开源项目的代码注释中收集了 960 余万条 URL 链接, 并发现这些链接极少被更新, 仅有少于 9% 的链接有更新历史. 同时他们发现 9.1% 的代码注释中的链接处于无法访问的状态, 且开发者认为此类链接应该被修复.

综上所述, 待办注释、注释代码和链接这 3 种类型的代码注释常出现时效性相关的质量问题, 开发者应该在软件演化的过程中对此类代码注释进行合理的管理与维护.

2.7 关联性

代码注释的关联性指代码注释与其附近的代码是否是紧密相关的. 若代码注释描述的并非相邻代码片段相关

信息, 而是其他位置代码片段的相关信息, 且无相关指引, 使得开发者无法快速定位到代码注释所描述的代码片段, 则认为此代码注释在关联性上存在质量问题. 代码注释在关联性上存在问题并不意味着其在一致性上也是存在问题的, 因为其描述的内容是正确的, 只是该注释未书写于合适的位置. Jabrayilzade 等人[33]认为代码注释应该关注附近代码的信息, 这样开发者才能轻松地理解代码. 若代码注释内容相关的代码位置距离此注释较远, 则认为该代码注释的位置应该被调整, 从而更好地在开发任务中服务开发者.

2.8 小 结

本文发现当前文献主要从存在性、一致性、全面性、重要性、可读性、时效性、关联性等维度对代码注释的质量进行了评价. 表 4 介绍了上述评价维度, 并列举了相关文献[2−4, 8−10, 12−15, 17−28, 30−38, 40−56], 可以看出相关文献大多关注于代码注释的存在性与一致性. 代码注释所包含的复杂内容和其起到的丰富功能使得开发者对于代码注释的质量有着不同维度的要求. 有些维度的要求是较为明确的, 例如一致性要求代码注释内容应该正确地阐释代码相关信息. 但有些维度缺乏明确的标准, 例如全面性要求代码注释应该包含所有代码相关的重要信息, 然而何种信息是需要包含在注释中的重要信息, 对于不同上下文开发者有着不一样的理解. 与此同时, 代码注释质量的评价维度在不同粒度的代码注释间也会有所差异, 例如对于类与方法层面的代码注释, 全面性和重要性更为关键, 让开发者能够迅速理解相关代码的设计与实现逻辑, 而对于行间注释, 因为书写位置的不固定与对复杂代码片段认知的差异, 其存在性更受到开发者与相关文献的关注. 如何对这些评价维度进行更细粒度的刻画并实施自动分析是很有价值的研究方向, 此类研究能加深开发者对于代码注释质量要求的理解, 同时也为服务于代码注释质量的自动化工具建立基础. 当前工作在面向代码注释质量问题研究时大多未对代码注释的类型、目的、粒度等维度进行区分, 考虑这些复杂属性所构成的上下文, 再对代码注释的质量进行分析能够得到更为细致的洞察与工具.

表 4 当前文献中对代码注释质量评价维度

评价维度	含义	相关文献
存在性	必要代码注释是否存在	[2,3,10,12−15,17−23,33,34,48−51]
一致性	代码注释内容与相关代码实现是否一致	[8,10,15,17,18,24−28,30,31,33,34,44−47,52,53]
全面性	代码注释是否包含所有必要信息	[4,9,10,15,20,32,36,54]
重要性	代码注释是否仅包含重要信息	[10,12,15,22,32−35]
可读性	代码注释是否清晰易读	[10,13,33,34,37,38,44]
时效性	任务相关代码注释是否及时维护更新	[10,33,40−43,55]
关联性	代码注释是否与附近代码紧密相关	[33,56]

3 代码注释质量的度量指标

当前文献从多种维度对代码注释质量的评价标准进行了阐述, 并使用特定指标对代码注释质量的特定维度进行了度量. 如表 5 所示, 本文对当前文献中经常使用的代码注释质量度量指标进行了总结, 并分析讨论了其应用场景与局限性.

表 5 相关文献使用的代码注释度量指标

度量指标	度量维度	文献
代码注释密度	存在性	[2,3,17−19]
代码注释长度	重要性	[32,34,44]
文本相似度	重要性、一致性	[20,32,34,35]
可读性指标	可读性	[37,38,44]
独立代码注释变更比例	存在性、一致性等	[10]

3.1 代码注释密度

因为对于具有何种特征的代码需要代码注释这一问题目前没有通用解答, 相关文献通常使用代码注释密度这

一度量指标来宽泛地度量特定项目的代码注释存在性. 通常来说, 特定项目内代码注释密度越高, 则在代码注释存在性有质量问题的可能性越小, 即需要代码注释的代码可能都已经书写了相关注释.

$$CommentDensity = \frac{CL(CommentLines)}{LOC(LinesofCode)} \tag{1}$$

代码注释密度的计算如公式 (1) 所示, 为代码文件中的注释行数与代码总行数的比值. 在计算代码总行数时, 通常会去除空行等无效代码.

代码注释密度可以从一定程度上反映代码注释的质量, 例如以教育目的的软件项目, 会有着显著更高的代码注释密度[3], 因为此类项目更加希望读者能够高效快速地理解其代码, 所以通常会包含充分的代码注释. 然而代码注释密度也会受到其他因素的影响, 例如编程语言类型[3]. 不同编程语言往往有着不同的编码实践与书写逻辑, 这使得用不同语言书写的代码在完成同一功能时可能有着不同体量的代码. 同时, 代码注释密度低并不代表着特定软件项目一定有着较低的代码注释, 拥有着较低代码注释密度的项目可能也拥有着全备的代码注释, 只是这一比例因为项目特性、编码习惯等原因而偏低. 综上, 高的代码注释密度可以反应项目有着良好的代码注释实践, 但较低的代码注释密度并不一定代表着项目内代码注释存在质量问题.

3.2 代码注释长度

代码注释长度常被用于度量代码注释的重要性. 相关文献常使用代码注释所包含词的个数作为代码注释长度用于度量代码注释质量. 相关文献认为若代码注释过短, 则说明代码注释可能并不包含全面有效的信息; 若代码注释过于冗长, 则极有可能包含无用信息, 使得开发者无法快速定位到代码注释中的宝贵信息.

然而何等长度的代码注释属于太短、何等长度属于太长并没有定论. 当前文献[32,34]通常采用 2 和 30 作为阈值对代码注释质量进行评价. Steidl 等人[34]通过提取现实代码注释样例并对开发者进行了问卷调研, 发现开发者认为包含最多两个单词的代码注释仅包含本地信息且应该被删除, 包含 2–30 个单词的代码注释既包含本地信息也包含全局信息, 而包含 30 个以上单词的代码注释通常包含全局信息.

3.3 文本相似度

文本相似度指代码注释内容与其对应代码的文本之间的相似度. 通常认为, 若文本相似度过低, 则该代码注释可能与相关代码关联性较差, 并没有提供充分的代码相关信息, 甚至可能是不一致的; 若文本相似度过高, 则该代码注释可能仅复述了相关代码的信息, 并没有包含额外的开发者需要的信息.

代码注释与代码之间的文本相似度有多种计算方法, 其中最为广泛的计算方法为计算方法名称中词汇在注释中所出现的比例[20,32,39]. 相关工作通常采用 0 和 0.5 作为阈值[20,32,35]. Steidl 等人[34]认为若文本相似度等于 0, 则认为代码注释与相关方法相关性不足, 缺失关键信息; 而若文本相似度大于 0.5, 则认为代码注释缺失代码之外的额外信息, 并通过问卷调研验证了以上两条猜测. 而 Sun 等人[20]则认为若文本相似度小于 0.5 则认为代码注释和相关方法之间相关性较差, 缺失了方法相关信息. 可以看到这一计算方法存在着许多问题, 例如当前工作对使用这一指标如何评价代码注释质量仍存在分歧; 这一方法只能用于度量方法或类所对应的代码注释, 而对于常见的行间注释则无法使用这种方式进行计算; 这一方法仅考虑相关方法的名称, 对于命名较不规范或者名字较短的方法 (例如仅包含少于两个词的方法名), 通过该种方式计算得到的文本相似度可能并不能精确地反应代码注释与代码之间的相关度.

为了解决以上所提到的问题, Aman 等人[35]提出了一种度量代码注释与相关代码文本相似度的新方法, 他们使用 Doc2Vec 这一模型来完成对代码注释和代码文本的相似度比较. Doc2Vec 模型基于 Word2Vec 模型, 是一种在自然语言处理领域被广泛运用的模型, 可以将文档转换为向量, 且有着相似含义的文档其转换而成的向量也会更加相近. 基于这一特性, Aman 等人使用从软件仓库中挖掘代码注释和对应方法训练了 Doc2Vec 模型, 并使用该模型将需要度量的代码注释和对应方法代码转换成向量, 通过计算向量余弦相似度的方式计算二者的文本相似度. 他们的验证实验发现, 生成向量的余弦相似度越大, 代码注释与相关代码的相关性更强; 余弦相似度越小, 则说明代码注释中代码相关的信息缺失的可能性更大.

然而, 尽管 Aman 等人[35]对文本相似度的计算方法进行了改进, 这一量度还有着本质上的缺陷, 即代码注释和代码文本上本身就存在层次、粒度上的差别. 代码注释通常是从高层次粗粒度地对代码行为及注意事项的总结, 理应比代码文本本身更为抽象与精炼, 因此在文本上完美匹配的代码注释与代码也可能有所差别. 对代码注释重要性的度量除了对文本的相似度的考量, 还理应加入更多语法语义层面的分析, 来确保其精确性.

3.4 文本可读性指标

为了度量代码注释文本的可读性, 相关文献借用了对自然语言文本可读性进行度量的经典指标将其用于代码注释上, 包括 Fog 指数 (Fog index)[38,44]、Flesch 阅读分数 (Flesch reading ease)[37,38,44]、Flesch-Kincaid 分数 (Flesch-Kincaid grade level)[38,44].

Fog 指数 (Fog index)[38,44]是常见的被用于检验英语文本可读性的指标, 其反映了理解给定文本所需要的教育水平. 公式 (2) 展现了 Fog 指数的计算公式, 其中 *ave_words* 表示该文本中平均每个句子所使用的词语数目, *difficult_words* 表示复杂词汇在文本中所占比例, 复杂词汇指音节大于等于 3 的英语单词. Fog 指数越高说明特定文本越难以理解, 如其值在 6 以下说明该文本符合小学学生的阅读水平, 而其值在 9–12 说明理解该文本需要高中水平的教育.

$$FogIndex = (ave\_words + difficult\_words) \times 4 \tag{2}$$

Flesch 阅读分数 (Flesch reading ease)[37,38,44]是由 Rudolph Flesch 设计的用于度量英文文本的阅读难度的指标, 类似于 Fog 指数, 其同样反映了理解给定文本所需要的教育水平. 其计算公式如公式 (3) 所示, 其中 *ave_words* 与 Fog 指数一致表示该文本中平均每个句子所使用的词语数目, *ave_syllabels* 表示平均每个词的音节数目. 类似于 Fog 指数, Flesch 阅读分数同样不同数值对应着不同的教育水平, 但 Flesch 阅读分数越低表示特定文本更为复杂.

$$FleschReadingEase = 206.835 - (1.015 \times ave\_words) - (84.6 \times ave\_syllabels) \tag{3}$$

Flesch-Kincaid 分数 (Flesch-Kincaid grade level)[38,44]是另一项由 Rudolph Flesch 设计的用于度量英文文本的阅读难度的指标, 但是使用了不一样的权重来进行计算. 公式 (4) 展示了 Flesch-Kincaid 分数的计算方式, 其中 *ave_words* 和 *ave_syllabels* 的含义与计算方式与 Flesch 阅读分数相同, 其分数越高说明特定文本更容易被理解, 90–100 分说明该文本能被平均 11 岁的学生所轻松理解, 60–70 分说明该文本能被 13–15 岁的学生所轻松理解, 而 0–30 分说明该文本需要本科生水平的阅读水平才能较好地理解.

$$FleschKincaid = (0.39 \times ave\_words) + (11.8 \times ave\_syllabels) - 15.9 \tag{4}$$

这些度量代码注释文本可读性的指标确实可以从一定程度上反映代码注释的阅读难度, 即代码注释是否使用复杂的词汇与句子书写. 但是, 用简单语句书写的代码注释并不直接等同于代码注释所需要传递的内容是被清晰地表达了, 因此这些可读性相关指标在多大程度上可以反映代码注释的可读性尚并不清楚.

3.5 独立代码注释变更比例

针对存在质量问题的代码注释难以定位这一难题, Wang 等人[10]提出了一种新颖的方法来研究代码注释的质量问题, 即面向独立代码注释变更开展对代码注释质量的分析. 独立代码注释变更指在代码提交内只对代码注释进行修改、而未修改相关代码的代码变更. 在理想状态下, 代码注释应该随着相关代码的更改一起被修改, 因此他们认为此类独立代码注释变更很有可能是开发者通过此类变更在提升代码注释质量, 从而可以面向此类代码变更来对代码注释质量进行分析. 他们通过实验验证这一猜想, 并提出了独立代码注释变更比例这一指标用于度量特定项目内的代码注释质量.

独立代码注释变更比例即为独立代码注释变更在所有代码注释变更中所占的比值, 在 Wang 等人[10]分析的 4 000 余个有着成熟开源实践的 Java 项目中, 这一比值约为 16%. 在特定项目内这一比值越大, 说明代码注释经常被独立修改提交, 代码注释可能存在着更多的质量问题. 他们使用这一指标对项目的编码指南发布情况和工具使用情况在代码注释质量上的影响进行了度量, 发现发布了何处需要代码注释或者需要何种代码注释的相关编码指南的项目相较未发布相关编码指南的项目会有着显著更低的代码注释变更比例, 同时他们发现特定静态检查工具的引入会显著降低项目的独立代码注释变更比例 (例如 Javadoc 工具), 这说明发布编码指南和使用特定静态检查工具

可以减少项目内的代码注释质量问题.

然而由于开发者开发实践的多样性也可能对这一指标的准确性产生影响, 例如一些开发者习惯于在相关代码的开发完成后, 将代码注释相关变更在另一代码提交中进行提交, 这一过程中并无存在质量问题的代码注释, 但独立代码注释变更的比例会因为这一开发习惯而存在误差. 其次, 对于代码注释和相关代码的匹配目前还没有成熟通用的算法[24], 相关文献通常是通过启发式算法对代码注释所对应的代码进行匹配[10], 这也为这一指标的计算引入了误差.

3.6 小 结

为了对代码注释的质量进行自动度量, 相关工作通常是从自然语言文本的角度提取特征, 例如代码注释的长度、与代码的文本相似度、可读性指标等. 这些代码注释质量的度量指标可以从一定程度上反映代码注释的质量, 但因为其只涉及到对代码文本的分析而具有局限性. 为了对代码注释质量有着更好的刻画与度量, 代码注释与相关代码的语法语义信息可以更多地被考虑进来. 本文建议在对代码注释质量进行度量时, 可以使用词嵌入等深度学习技术, 对代码与代码注释的语义进行学习表征, 从而加深相关方法对于当前代码注释所表达内容的一致性、全面性、重要性等方面的刻画, 达到更好的对于代码注释质量进行度量的效果.

4 代码注释质量的提升策略

本文对相关文献中提出的提升代码注释质量的策略进行了总结与分析, 包括失配代码注释检测、过时待办注释检测、代码注释自动更新以及代码注释位置预测这 4 类.

4.1 失配代码注释检测

失配代码注释检测技术指通过一系列技术手段, 定位到代码文件中其内容与对应代码的真实运行逻辑存在不一致的代码注释. 基于所使用的技术手段, 本文将相关文献分为两类, 基于预定义规则的失配代码注释检测方法和基于机器学习的失配代码注释检测方法.

4.1.1 基于预定义规则的失配代码注释检测方法

基于预定义规则的方法通常从海量代码注释中挖掘得到频繁模式, 并基于得到的模式设置规则对符合模式的失配代码进行检测.

Tan 等人[45]提出了 iComment 这一方法来自动检测特定主题的失配的描述方法调用条件的代码注释, 其主要包括以下几个步骤: 对从代码文件中提取得到的代码注释进行预处理, 包括去除特殊字符、分词、词性分析等过程; 使用特定关键词筛选出属于描述方法调用条件的代码注释; 使用混合模型 (mixture model) 对代码注释进行聚类, 得到代码注释的频繁主题 (例如在 Linux 的代码注释中, 锁为频繁主题); 为特定主题提取关键词, 并使用关键词定位与特定主题相关的代码注释; 对获得的代码注释进行取样并人工标记规则, 包括代码注释模式与相关代码需要满足的条件; 使用决策树 (decision tree) 作为分类模型, 提取了注释范围、条件规则、注释文本、应用范围等方面的特征, 使用人工标记的代码注释训练分类模型, 用于预测未标记的代码注释符合何种规则; 使用程序分析的方法检测相关代码是否匹配代码注释中所包含的规则.

Tan 等人[27]提出了@tComment 方法来检测 Javadoc 注释中与空值相关的失配代码注释, 该方法主要包含两部分, 从代码注释中提取与空值相关的内容, 以及通过生成测试的方式检验提取得到的内容. 对于第 1 部分, 该方法使用启发式规则对 Javadoc 中参数标签和异常标签文本进行分析, 与 4 种预定义好的规则进行匹配, 包括空值正常 (参数传入空值时方法可以正常运行)、空值异常 (参数传入空值会导致方法抛出不特定的异常)、空值特定异常 (参数传入空值会导致方法抛出特定的异常) 以及空值未知 (参数传入空值不确定是否会使方法抛出异常). 在第 2 部分中, 该方法对 Randoop 这一随机生成测试的算法进行扩展, 通过随机在方法的一个或多个参数传入空值, 测试程序的运行逻辑是否与注释一致的方式, 检测相关的失配代码注释.

综上所述, 此类方法的通常步骤是预先收集代码注释与相关代码的频繁模式, 基于频繁模式定义相关规则匹配相关代码注释, 并基于定义规则生成相关测试用于检测代码运行逻辑是否一致. 因此, 是否能够精确全面地定义

相关规则是此类方法能否取得好的效果的关键, 而对此类规则的定义通常要求相关领域的先验知识, 相关知识的缺乏可能会限制此类方法的开发与使用. 同时, 代码注释的复杂性质使得此类预定义规则难以具有普适性, 因此相关文献通常仅面向特定主题的代码注释, 如操作系统中的锁操作与方法参数的空值相关的代码注释.

4.1.2 基于机器学习的失配代码注释检测

此类方法通常从代码注释以及相关代码中提取特征, 使用机器学习的方法学习失配代码注释在提取特征上的差异, 来实现对失配代码注释的自动检测.

Liu 等人[57]从面向代码变更中是否出现适配代码这一问题, 从 3 个维度提取了 64 项特征用于适配代码检测, 包括: (1) 代码特征, 包括更改代码的数量、比例、类型等方面的特征; (2) 注释特征, 包括注释长度、是否包含特定关键词 (如“TODO”“bug”)、注释密度等; (3) 关系特征, 用于衡量代码注释与相关更改代码的关系, 包括代码注释与更改前后的代码之间的文本相似度、文本距离等. 该文献将失配代码注释检测问题看作二分类问题, 从 5 个开源软件项目的代码变更中提取数据集, 将更改前的代码注释视作适配注释, 并使用随机森林 (random forest) 作为分类器对数据集进行训练与预测.

Rabbi 等人[46]认为代码注释与相关代码在主题上的分布应该是一致的, 并基于这一思路提出了一套新的方法用于检测代码文件中的失配注释. 他们的方法主要包括以下几步: 对代码注释与对应代码进行预处理; 使用 LDA (latent Dirichlet allocation) 主题模型, 对代码注释和相关代码进行训练, 如公式 (5) 所示, 为每一条代码注释或者代码片段生成 k 维的向量, 其中 k 为 LDA 主题模型指定的主题数目, 向量中的每一位为给定注释或代码在特定主题上的概率; 为了消除 k, 即 LDA 主题模型的主题设定数目对结果带来的影响, 如公式 (6) 所示, 对每个主题数目不同的 LDA 模型得到代码注释和代码的向量进行拼接, 然后作为输入使用随机森林模型进行训练; 最后使用支持向量机 (support vector machine, SVM) 这一分类器, 如公式 (7) 所示, 将上述一组随机森林得到的结果进行拼接得到的向量作为输入, 得到代码注释与对应代码匹配的概率. 该方法使用 Corazza 等人提供的数据集[47]进行了验证, 发现在不同项目内都能取得令人满意的效果.

$$ConcatenatedFeature_k = LDA_k(code) \oplus LDA_k(comment) \tag{5}$$

$$score_k = RF_k(ConcatenatedFeature_k) \tag{6}$$

$$\hat{Y} = SVM(\oplus_{k=i}^{j} score_k) \tag{7}$$

在 Rabbi 等人的后续工作[26]中, 他们提出了使用孪生神经网络 (siamese recurrent network) 来度量代码注释与相关代码的相似度的方法, 来检测代码文件中的失配代码注释. 对于代码, 为代码生成抽象语法树 (abstract syntax tree, AST), 并选择其中属于特定节点类型 (如变量名和字符串等) 的节点组成代码标识符 (code tokens); 而对于代码注释, 则直接对其进行预处理, 选择其中的词作为代码注释标识符 (comment tokens). 对于代码标识符序列和代码注释标识符序列, 分别使用 RNN-LSTM 模型进行训练, 最后选取两个模型的隐藏层向量计算相似度, 来作为代码注释与相关代码的相似度, 计算公式如公式 (8) 所示, 分数越接近 0 说明给定代码注释越可能为失配代码.

$$f(H_m, H_n) = \exp(-\|H_m - H_n\|) \tag{8}$$

Panthaplackel 等人[31]同样使用了对代码注释和相关代码同时进行编码, 再通过相关神经网络结构判断对应代码注释和相关代码的向量标准是否相似. 不同的是, 他们再对相关代码进行编码时加入代码的语义特征, 即将代码解析得到的抽象语法树节点也作为特征输入到图神经网络进行对代码的编码的学习. 这样代码运行逻辑的语义信息也会在判断过程中被考虑.

可以看到, 当前文献使用机器学习技术检测失配注释时, 大部分仅使用了相关代码的文本信息, 而忽视了其语义信息. 与此同时, 因为代码注释通常是是对代码的语法与语义更高层次的抽象与总结, 因此在文本上不一定会与相关代码高度相似. 因此为了获得更精准的匹配信息, 代码的语义信息、调用信息等也可以作为特征加入到训练中.

4.2 过时待办注释检测

过时待办注释检测指通过对待办注释文本和相关代码的分析, 自动检测应删除的已完成的待办注释. 其面临的两个关键问题为如何为待办注释匹配相对应的代码范围从而完成后续的匹配, 以及如何通过分析待办注释和相关代码而检测相关任务是否已完成. 本文对相关文献面向这两个关键问题提出的解决方案进行了总结.

过时待办注释检测面临的一大难题是如何为待办注释定位到其对应开发任务所对应的代码片段, 在此基础上才能进一步分析其中包含的开发任务是否已完成. 为了解决这一问题, Sridhara 等人[41]提出了块范围 (blockscope) 这一概念, 指从待办注释所在的行一直到最近的一个结构块结束行 (结构块指 for、try 等语句所包含的所有代码行). 他们认为这一范围即为待办注释所对应的应修改的相关代码范围, 例如写在某个 try 语句后的待办注释对应整个 try 语句相关代码块. Ratol 等人[28]同样使用了待办注释应该关注于同一方法内附近的代码这一假设, 基于 Java 语法规则提出了一个在方法内部为待办注释划分界限的启发式方法.

过时待办注释检测面临的另一大难题是如何匹配待办注释中的任务相关文本与代码片段逻辑, 从而达到检测给定任务是否已完成的目的. Sridhara 等人[41]实现了 3 个检查器来对代码注释和相关代码修改关联性进行检查, 包括基于信息检索的状态检查器、基于语法的状态检查器和基于语义的状态检查器. 基于信息检索的状态检查器通过计算待办注释与代码修改间使用 TF-IDF 值加权的余弦相似度, 并设置 0.2–0.8 之间不同的阈值来判断二者是否匹配. 基于语法的状态检查器则是通过自然语言处理技术, 从待办注释文本中提取出任务的动宾结构, 并于代码修改的实际操作与对象进行匹配, 若二者匹配, 则认为待办注释对应任务已完成. 基于语义的状态检查器用于解决注释与代码的文本并不相似而语义上相似的这一问题. 这一检查器基于 4 种常见待办注释的任务类型 (即新增文档注释、新增解释注释、代码移除、代码新增), 对待办注释进行了分类并提出了对相关代码变更的对应检查方法. Ratol 等人[28]提出了语法和语义两类匹配规则来完成检测待办注释相关任务是否已完成. 他们提出的语法匹配规则同样基于代待办注释和相关实现代码应该有类似的文本这一假设, 对待办注释中的词语与代码标识符文本进行了比较, 通过一系列的规则计算其相似度. 而语义规则则是对文本进行词干还原, 并面向常见的方法名 (即以 *get* 或 *is* 开头的方法名) 提出了两条规则进行匹配. Nie 等人[42]面向待办注释提出了一套框架, 将待办注释规范为包含动作 (action) 和条件 (trigger) 两方面的文本, 并基于文本提出一系列的测试规范用于测试给定待办注释是否已经过时.

4.3 代码注释自动更新

失配代码注释大多是因为在更新代码时未及时更新对应代码注释而引入, 因此相关工作面向代码提交这一场景, 相关文献基于深度学习技术提出了基于代码修改, 自动对代码注释进行更新的方法.

Liu 等人[30]提出了 CUP 方法, 以代码的编辑序列和旧代码注释为输入, 更新后的代码注释为输出, 面向代码变更对代码注释进行了自动更新. 对于代码变更, 该方法将代码变更转换为每一位为三元组的编辑序列, 其中三元组的前两位为更改前后的代码标识符, 第 3 位为更改类型. 对于代码注释, 该方法对文本进行预处理后将其标记化, 得到对应的注释标识符序列. 该方法使用协作注意力层 (co-attention layer) 连接的两个双向 LSTM 模型 (bi-LSTM model) 分别对得到的代码编辑序列和代码注释标识符序列进行编码, 然后使用基于 LSTM 模型的解码器生成得到的更新后的注释. 该方法的纰漏之处在于并没有对代码注释是否需要更新进行细分, 在大多数情况下对代码的更新并不会影响到相关代码注释, 此时代码注释并不需要更新.

Lin 等人[29]对 CUP 方法的优点和缺点进行了深层次分析, 他们发现 CUP 更新生成的代码注释只进行了细微的更改, 96.6% 正确生成的代码注释仅修改了一个标识符; CUP 正确更新代码注释需要在训练集中有极为类似的代码变更, 超过 90% 的正确样例都是属于这种场景; CUP 经常会忽视相关的代码变更信息, 而被无关的代码变更信息误导, 使得其无法正确地更新代码注释. 基于以上对 CUP 的深度分析, Lin 等人提出了一种利用一系列启发式规则的更为简单却有效的代码注释更新方法 HEBCUP. 该方法基于代码注释常见的更新模式, 提出了一些系列替换代码注释标识符的启发式规则, 面向代码变更对代码注释进行更新. 他们提出的 HEBCUP 方法运行速度是 CUP 的 1 700 余倍, 在准确率上也超出 CUP 约 62%. 但基于启发式规则的方法在实际运用中同样面临着通用性的通病, 对于规则未能覆盖到的场景, 该方法则无法有效地完成更新代码注释的任务. 与此同时, 如何精确、全面地定义启发式规则需要相关的实证经验.

4.4 代码注释位置预测

为了避免必要的代码注释缺失这一问题的出现, 相关文献提出了代码注释位置预测的方法, 将代码是否需要

书写代码注释这一问题视为二分类问题, 提醒开发者在特定位置可能需要相关代码注释.

Wang 等人[48]面向不同类型的代码, 提取了不同类型的特征, 并使用分类器判断给定代码片段是否需要代码注释. 对于方法级别的代码, 他们提取了方法名长度、是否有返回值、是否为私有方法、参数个数等特征; 而对于循环语句代码块, 他们提取判断语句数目、是否包含 return 语句、是否包含 break 语句等特征. 基于得到的特征和从实际代码文件中收集的数据集, 该方法在多种分类器上进行了性能评价, 并取得较好的准确率. 然而此方法只面向类、方法注释, 以及条件、循环、异常代码块, 而代码文件中需要代码注释的相关代码类型不仅限于此.

Huang 等人[22]将这一问题进行了另一种形式的定义. 对于代码文件中的任意代码行, 若此代码上一行为代码注释, 则认为是正样本, 反之则为负样本. 此时代码注释位置预测问题转换为, 对于给定代码文件, 为方法内的每一行, 预测此处是否需要插入注释. 他们提出了 CommentSuggester 这一方法, 从代码中提取了 3 类特征: 结构上下文特征, 包括语句类型、标识符数目、调用信息等; 语法上下文信息, 通过对给定代码解析抽象语法树, 提取了特定节点是否包含在抽象语法树的相关特征; 语义上下文信息, 对代码进行标志化处理, 得到其标识符序列, 使用 skip-gram 模型 (continuous skip-gram model) 进行训练获得代码的向量表征. 将这 3 类特征进行拼接作为输入, 使用机器学习模型作为分类器进行训练与预测.

Huang 等人[22]将代码注释预测问题转换为给定方法内的每一代码行处是否需要插入代码注释的问题, 然而现实场景中, 代码注释可能描述其后的多行代码, 因此以代码行为粒度进行预测可能并无法精准得定位需要代码注释的位置. 为了解决这一问题, Louis 等人[23]以代码块为粒度对代码注释位置进行了预测. 他们通过空行, 对代码文件中的代码块进行了分割. 对于任意包含代码注释的代码块, 他们在收集数据集时视为正样本. 他们使用循环神经网络 (recurrent neural network, RNN) 将代码块标志化后输入, 输出该代码块是否需要代码注释.

当前文献将代码注释位置预测这一问题抽象为二分类问题并提出了相关方法, 然而因为代码注释内容的复杂性, 仅给出代码注释位置可能并无法有效地帮助开发者提升代码注释覆盖率. 例如即使指定了需要代码注释的方法, 何种代码注释在此处是需要的仍需要开发者有着相关背景知识. 因此, 本文建议未来工作在继续这一方向的探索时, 可以将代码注释的类型考虑其中, 将二分类问题拓展为多分类问题, 更好地服务于提升代码注释质量.

4.5 小 结

本文总结了当前文献面向失配代码注释检测、过时待办注释检测、代码注释自动更新以及代码注释位置预测 4 类代码注释提升策略进行了分析与总结, 并分析了当前策略所使用方法存在的不足. 失配代码注释检测和代码注释自动更新的相关技术, 通过对失配代码注释的检测与更新, 从一致性对代码注释质量进行了提升; 代码注释位置预测这一技术, 通过对必要代码注释位置的预测, 从存在性这一角度对代码注释质量进行了提升; 而过时待办注释检测技术, 则是通过对待办注释此类极易随着开发进度而失效的代码注释的分析, 从时效性这一角度对代码注释质量进行了提升. 然而对于代码注释存在的其他质量问题, 如重要性、可读性等, 此类研究仍然是缺失的. 与此同时, 当前文献面向提升代码注释质量提出技术方法时, 仍存在两大挑战. 一是当前文献使用启发式规则来对代码注释所对应的代码范围进行定位, 而适用于普适场景的代码注释对应代码精确判定方法仍然是缺失的. 研究者可以通过对代码语义和代码注释语义的匹配, 为代码注释确定其对应的相关代码范围, 这样使得面向代码注释质量的分析能更精确地定位到分析对象. 二是当前文献大多将代码注释及其对应代码视为自然语言文本, 采用自然语言处理方向的经典方法对其进行处理与分析, 这忽略了代码注释及代码中包含的丰富其他信息, 如代码的语法语义信息、代码注释的内容类型信息等. 这些信息的加入能使得相关方法更好地理解代码与代码注释的关系.

5 未来研究方向

目前, 面向代码注释质量问题的研究已经取得了一定的进展, 对代码注释的质量评价标准进行了刻画, 并提出了方法对代码注释质量进行提升. 本文通过对相关文献的分析与总结, 发现面向代码注释质量问题, 仍有许多问题和挑战尚待解决.

5.1 代码注释质量评价维度的细粒度刻画

当前工作从存在性、一致性、全面性、重要性、可读性、时效性、关联性等维度对代码注释质量进行分析.

部分代码注释质量维度有着明确的定义, 例如一致性要求代码注释中的内容应该与代码的相关逻辑内容完全一致. 而一些代码注释维度则仍未有明确的定义, 如全面性要求代码注释应该包含所有代码相关的重要信息, 然而对于何种类型的注释是被开发者需要的, 可能因为代码的特征与上下文不同, 也会有所不同. 与此同时, 由于代码注释所包含内容种类的多样性, 使得检测代码注释存在的质量问题的难度大大加深. 对于代码注释可能存在的质量问题的更细粒度的刻画, 如对不同类型的代码片段需要代码注释类型的系统性总结, 可以加深开发者对于代码注释质量问题的理解, 从而在开发过程中有意识地避免相关质量问题的引入; 也可以启示研究者提出方法对代码注释质量问题进行自动化的检测与解决.

5.2 代码注释质量度量指标和提升策略的深层次分析

相关文献对代码注释质量进行度量与分析时, 常将代码注释与相关代码视为自然语言文本, 使用常用的自然语言文本的指标和方法对其进行分析. 然而, 在对代码注释质量进行分析时, 诸如代码的语法语义信息、代码注释的类型等包含了丰富信息的相关特征时常被忽略. 因此本文建议对代码注释对应代码范围进行更精细的定位, 并从代码中提取语法语义特征、从代码注释中提取其内容类型特征用于对代码注释质量的量化与提升的相关分析中.

5.2.1 代码注释对应代码范围定位

代码注释总是面向特定代码书写, 因此对于代码注释质量进行分析时, 不可避免需要对其对应的代码也同时进行分析. 然而对于如何为给定代码注释划分对应的代码范围, 目前还没有可靠的通用方法[24]. 当前文献通常是通过基于经验的启发式规则[10,41], 来为代码注释划定对应代码的范围; 或是仅面向类与方法的注释开展研究[27,30], 因为此类代码注释所对应的代码是固定的, 即为相关类与方法的代码. 对于代码注释对应代码范围的精确定位, 能够帮助开发者更好地获取代码注释相关代码中的相关信息, 从而更有效地对代码注释的质量进行分析.

5.2.2 代码注释对应代码语法语义特征的提取

对于代码注释质量问题的分析, 无法避免需要对代码注释对应的相关代码进行分析. 例如在对代码注释的一致性进行分析时, 为了判断代码注释内容是否与代码实际行为保持一致, 不仅需要对代码注释文本进行分析, 同时也需要对相关代码的运行逻辑进行分析[27,45]. 然而, 由于代码其本身的复杂性, 相关文献在对代码注释代码进行分析时, 往往仅将其视为文本, 即仅提取其标识符中所包含的文本信息, 而忽视了代码的结构信息. 由于代码是符合特定规则体系的符号集合, 可以从中提取更多的语法语义相关信息, 用于代码注释质量问题相关分析中, 例如抽象语法树、调用关系序列等. 此类信息的引入能大大加深对相关代码逻辑的理解, 同时提升代码注释质量相关的方法的效果.

5.2.3 代码注释内容类型特征的使用

代码注释通常包含着丰富的信息, 因此相关文献对代码注释内容的分类系统进行了总结[4,10,36], 并基于得到的分类系统训练了分类器. 然而相关文献在对代码注释质量问题进行分析时, 只面向特定种类的代码注释 (例如功能总结), 或者忽略了代码注释种类. 例如代码注释位置预测的相关工作提出的方法仅考虑了特定位置代码注释是否存在[2,22,48], 而未考虑代码注释的种类. 这样即使开发者从此类方法中获得代码文件的何处可能是需要代码注释的, 但仍会困惑于应书写何种内容的代码注释. 将代码注释的内容种类的考量加入相关方法中可以使其更好地服务于软件开发的实际场景, 例如在预测需要代码注释的位置时, 同时给出该位置需要代码注释类型.

6 总 结

本文对研究代码注释质量问题的相关文献进行了收集与分析, 从代码注释质量的评价维度、度量指标和提升策略这 3 方面对研究现状进行了分析与总结, 指出了当前研究存在的问题与挑战, 并提出了建议. 本文希望我们的分析和总结能够帮助加深对代码注释质量问题的进一步理解, 促进代码注释质量问题检测技术和提升策略的增强.

References:

[1] de Souza SCB, Anquetil N, de Oliveira KM. A study of the documentation essential to software maintenance. In: Proc. of the 23rd

Annual Int'l Conf. on Design of Communication: Documenting & Designing for Pervasive Information. Coventry: ACM, 2005. 68–75. [doi: 10.1145/1085313.1085331]

[2] Arafat O, Riehle D. The comment density of open source software code. In: Proc. of the 31st Int'l Conf. on Software Engineering-companion Volume. Vancouver: IEEE, 2009. 195–198. [doi: 10.1109/ICSE-COMPANION.2009.5070980]

[3] He H. Understanding source code comments at large-scale. In: Proc. of the 27th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Tallinn: ACM, 2019. 1217–1219. [doi: 10.1145/3338906.3342494]

[4] Pascarella L, Bacchelli A. Classifying code comments in Java open-source software systems. In: Proc. of the 14th IEEE/ACM Int'l Conf. on Mining Software Repositories (MSR). Buenos Aires: IEEE, 2017. 227–237. [doi: 10.1109/MSR.2017.63]

[5] Aghajani E, Nagy C, Linares-Vásquez M, Moreno L, Bavota G, Lanza M, Shepherd DC. Software documentation: The practitioners' perspective. In: Proc. of the 42nd IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Seoul: IEEE, 2020. 590–601.

[6] Nielebock S, Krolikowski D, Krüger J, Leich T, Ortmeier F. Commenting source code: Is it worth it for small programming tasks? Empirical Software Engineering, 2019, 24(3): 1418–1457. [doi: 10.1007/s10664-018-9664-z]

[7] Weinberger B, Silverstein C, Eitzmann G, *et al*. Google C++ style guide. Section: Line Length. 2013. http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml#Line_Length

[8] Ibrahim WM, Bettenburg N, Adams B, Hassan AE. On the relationship between comment update practices and software bugs. The Journal of Systems and Software, 2012, 85(10): 2293–2304. [doi: 10.1016/j.jss.2011.09.019]

[9] Haouari D, Sahraoui H, Langlais P. How good is your comment? A study of comments in Java programs. In: Proc. of the 2011 Int'l Symp. on Empirical Software Engineering and Measurement. Banff: IEEE, 2011. 137–146. [doi: 10.1109/ESEM.2011.22]

[10] Wang C, He H, Pal U, Marinov D, Zhou MH. Suboptimal comments in Java projects: From independent comment changes to commenting practices. ACM Trans. on Software Engineering and Methodology, 2022, 32(2): 45. [doi: 10.1145/3546949]

[11] Goodman LA. Snowball sampling. The Annals of Mathematical Statistics, 1961, 32(1): 148–170. [doi: 10.1214/aoms/1177705148]

[12] Börstler J, Paech B. The role of method chains and comments in software readability and comprehension—An experiment. IEEE Trans. on Software Engineering, 2016, 42(9): 886–898. [doi: 10.1109/TSE.2016.2527791]

[13] Sukamto RA, Megasari R, Piantari E, Rischa MNF. Code comment assessment development for basic programming subject using online judge. In: Proc. of the 7th Mathematics, Science, and Computer Science Education Int'l Seminar. Bandung: EAI, 2020. 1–5. [doi: 10.4108/eai.12-10-2019.2296547]

[14] Malik H, Chowdhury I, Tsou HM, Jiang ZM, Hassan AE. Understanding the rationale for updating a function's comment. In: Proc. of the 2008 IEEE Int'l Conf. on Software Maintenance. Beijing: IEEE, 2008. 167–176. [doi: 10.1109/ICSM.2008.4658065]

[15] Hu X, Xia X, Lo D, Wan ZY, Chen QY, Zimmermann T. Practitioners' expectations on automated code comment generation. In: Proc. of the 44th Int'l Conf. on Software Engineering. Pittsburgh: ACM, 2022. [doi: 10.1145/3510003.3510152]

[16] Aghajani E, Nagy C, Vega-Márquez OL, Linares-Vásquez M, Moreno L, Bavota G, Lanza M. Software documentation issues unveiled. In: Proc. of the 41st IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Montreal: IEEE, 2019. 1199–1210. [doi: 10.1109/ICSE.2019.00122]

[17] Fluri B, Wursch M, Gall HC. Do code and comments co-evolve? On the relation between source code and comment changes. In: Proc. of the 14th Working Conf. on Reverse Engineering (WCRE 2007). Vancouver: IEEE, 2007. 70–79. [doi: 10.1109/WCRE.2007.21]

[18] Fluri B, Würsch M, Giger E, Gall HC. Analyzing the co-evolution of comments and source code. Software Quality Journal, 2009, 17(4): 367–394. [doi: 10.1007/s11219-009-9075-x]

[19] Jiang ZM, Hassan AE. Examining the evolution of code comments in PostgreSQL. In: Proc. of the 2006 Int'l Workshop on Mining Software Repositories. Shanghai: ACM, 2006. 179–180. [doi: 10.1145/1137983.1138030]

[20] Sun XB, Geng Q, Lo D, Duan YC, Liu XY, Li B. Code comment quality analysis and improvement recommendation: An automated approach. Int'l Journal of Software Engineering and Knowledge Engineering, 2016, 26(6): 981–1000. [doi: 10.1142/S0218194016500339]

[21] Huang Y, Jia N, Shu JH, Hu XY, Chen XP, Zhou Q. Does your code need comment? Software: Practice and Experience, 2020, 50(3): 227–245. [doi: 10.1002/spe.2772]

[22] Huang Y, Hu XY, Jia N, Chen XP, Xiong YF, Zheng ZB. Learning code context information to predict comment locations. IEEE Trans. on Reliability, 2019, 69(1): 88–105. [doi: 10.1109/TR.2019.2931725]

[23] Louis A, Dash SK, Barr ET, Ernst MD, Sutton C. Where should I comment my code? A dataset and model for predicting locations that need comments. In: Proc. of the 42nd IEEE/ACM Int'l Conf. on Software Engineering: New Ideas and Emerging Results. Seoul: IEEE, 2020. 21–24.

[24] Wen FC, Nagy C, Bavota G, Lanza M. A large-scale empirical study on code-comment inconsistencies. In: Proc. of the 27th IEEE/ACM Int'l Conf. on Program Comprehension (ICPC). Montreal: IEEE, 2019. 53–64. [doi: 10.1109/ICPC.2019.00019]

[25] Sondhi D, Gupta A, Purandare S, Rana A, Kaushal D, Purandare R. On indirectly dependent documentation in the context of code evolution: A study. In: Proc. of the 43rd IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Madrid: IEEE, 2021. 1498–1509. [doi: 10.1109/ICSE43902.2021.00134]

[26] Rabbi F, Siddik MS. Detecting code comment inconsistency using Siamese recurrent network. In: Proc. of the 28th Int'l Conf. on Program Comprehension. Seoul: ACM, 2020. 371–375. [doi: 10.1145/3387904.3389286]

[27] Tan SH, Marinov D, Tan L, Leavens GT. @tComment: Testing Javadoc comments to detect comment-code inconsistencies. In: Proc. of the 5th IEEE Int'l Conf. on Software Testing, Verification and Validation. Montreal: IEEE, 2012. 260–269. [doi: 10.1109/ICST.2012.106]

[28] Ratol IK, Robillard MP. Detecting fragile comments. In: Proc. of the 32nd IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). Urbana: IEEE, 2017. 112–122. [doi: 10.1109/ASE.2017.8115624]

[29] Lin B, Wang SW, Liu K, Mao XG, Bissyandé TF. Automated comment update: How far are we? In: Proc. of the 29th IEEE/ACM Int'l Conf. on Program Comprehension (ICPC). Madrid: IEEE, 2021. 36–46. [doi: 10.1109/ICPC52881.2021.00013]

[30] Liu ZX, Xia X, Yan M, Li SP. Automating just-in-time comment updating. In: Proc. of the 35th IEEE/ACM Int'l Conf. on Automated Software Engineering. Melbourne: IEEE, 2020. 585–597.

[31] Panthaplackel S, Li JJ, Gligoric M, Mooney RJ. Deep just-in-time inconsistency detection between comments and source code. In: Proc. of the 35th AAAI Conf. on Artificial Intelligence. AAAI Press, 2021. 427–435.

[32] Nuur A, Gustafsson A, Azimoh A. Analysing the differences in comment quality between open source development and industria practices: A case study. Gothenburg: University of Gothenburg, 2019.

[33] Jabrayilzade E, Gürkan O, Tüzün E. Towards a taxonomy of inline code comment smells. In: Proc. of the 21st IEEE Int'l Working Conf. on Source Code Analysis and Manipulation (SCAM). Luxembourg: IEEE, 2021. 131–135. [doi: 10.1109/SCAM52516.2021.00024]

[34] Steidl D, Hummel B, Juergens E. Quality analysis of source code comments. In: Proc. of the 21st Int'l Conf. on Program Comprehension (ICPC). San Francisco: IEEE, 2013. 83–92. [doi: 10.1109/ICPC.2013.6613836]

[35] Aman H, Amasaki S, Yokogawa T, Kawahara M. A Doc2Vec-based assessment of comments and its application to change-prone method analysis. In: Proc. of the 25th Asia-Pacific Software Engineering Conf. (APSEC). Nara: IEEE, 2018. 643–647. [doi: 10.1109/APSEC.2018.00082]

[36] Rani P, Panichella S, Leuenberger M, Di Sorbo A, Nierstrasz O. How to identify class comment types? A multi-language approach for class comment classification. Journal of Systems and Software, 2021, 181: 111047. [doi: 10.1016/j.jss.2021.111047]

[37] Scalabrino S, Linares-Vásquez M, Poshyvanyk D, Oliveto R. Improving code readability models with textual features. In: Proc. of the 24th IEEE Int'l Conf. on Program Comprehension (ICPC). Austin: IEEE, 2016. 1–10. [doi: 10.1109/ICPC.2016.7503707]

[38] Eleyan D, Othman A, Eleyan A. Enhancing software comments readability using flesch reading ease score. Information, 2020, 11(9): 430. [doi: 10.3390/info11090430]

[39] Ying ATT, Wright JL, Abrams S. Source code that talks: An exploration of Eclipse task comments and their implication to repository mining. ACM SIGSOFT Software Engineering Notes, 2005, 30(4): 1–5. [doi: 10.1145/1083142.1083152]

[40] Storey MA, Ryall J, Bull RI, Myers D, Singer J. ToDo or to bug. In: Proc. of the 30th ACM/IEEE Int'l Conf. on Software Engineering. Leipzig: IEEE, 2008. 251–260. [doi: 10.1145/1368088.1368123]

[41] Sridhara G. Automatically detecting the up-to-date status of ToDo comments in Java programs. In: Proc. of the 9th India Software Engineering Conf. Goa: ACM, 2016. 16–25. [doi: 10.1145/2856636.2856638]

[42] Nie PY, Rai R, Li JJ, Khurshid S, Mooney RJ, Gligoric M. A framework for writing trigger-action todo comments in executable format. In: Proc. of the 27th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Tallinn: ACM, 2019. 385–396. [doi: 10.1145/3338906.3338965]

[43] Hata H, Treude C, Kula RG, Ishio T. 9.6 million links in source code comments: Purpose, evolution, and decay. In: Proc. of the 41st IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Montreal: IEEE, 2019. 1211–1221. [doi: 10.1109/ICSE.2019.00123]

[44] Khamis N, Witte R, Rilling J. Automatic quality assessment of source code comments: The JavadocMiner. In: Proc. of the 15th Int'l Conf. on Application of Natural Language to Information Systems. Cardiff: Springer, 2010. 68–79. [doi: 10.1007/978-3-642-13881-2_7]

[45] Tan L, Yuan D, Krishna G, Zhou YY. /*iComment: Bugs or bad comments?*/. In: Proc. of 21st ACM SIGOPS Symp. on Operating Systems Principles. Washington: ACM, 2007. 145–158. [doi: 10.1145/1294261.1294276]

[46] Rabbi F, Haque MN, Kadir ME, Siddik S, Kabir A. An ensemble approach to detect code comment inconsistencies using topic modeling. In: Proc. of the 32nd Int'l Conf. on Software Engineering and Knowledge Engineering. Pittsburgh: KSI Research Inc., 2020. 392–395.

[47] Corazza A, Maggio V, Scanniello G. Coherence of comments and method implementations: A dataset and an empirical investigation. Software Quality Journal, 2018, 26(2): 751–777. [doi: 10.1007/s11219-016-9347-1]

[48] Wang RM, Wang T, Wang HM. Study of a code comment decision method based on structural features. In: Proc. of the 2019 Int'l Conf. on Intelligent Computing, Automation and Systems (ICICAS). Chongqing: IEEE, 2019. 570−574. [doi: 10.1109/ICICAS48597.2019.00125]

[49] Rani P, Panichella S, Leuenberger M, Ghafari M, Nierstrasz O. What do class comments tell us? An investigation of comment evolution and practices in Pharo Smalltalk. Empirical Software Engineering, 2021, 26(6): 112. [doi: 10.1007/s10664-021-09981-5]

[50] Linares-Vásquez M, Li BY, Vendome C, Poshyvanyk D. How do developers document database usages in source code? (N). In: Proc. of the 30th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). Lincoln: IEEE, 2015. 36–41. [doi: 10.1109/ASE.2015.67]

[51] Huang Y, Jia N, Zhou Q, Chen XP, Xiong YF, Luo XN. Method combining structural and semantic features to support code commenting decision. Ruan Jian Xue Bao/Journal of Software, 2018, 29(8): 2226–2242 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5528.htm [doi: 10.13328/j.cnki.jos.005528]

[52] Wang DZ, Guo Y, Dong W, Wang ZM, Liu HR, Li SS. Deep code-comment understanding and assessment. IEEE Access, 2019, 7: 174200–174209. [doi: 10.1109/ACCESS.2019.2957424]

[53] Yang Z, Keung JW, Yu X, Xiao Y, Jin Z, Zhang JY. On the significance of category prediction for code-comment synchronization. ACM Trans. on Software Engineering and Methodology, 2023, 32(2): 30. [doi: 10.1145/3534117]

[54] Li ZX, Zhong H. Understanding code fragments with issue reports. In: Proc. of the 36th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). Melbourne: IEEE, 2021. 1312–1316. [doi: 10.1109/ASE51524.2021.9678864]

[55] Guo ZQ, Liu SR, Tan TT, Li YH, Chen L, Zhou YM, Xu BW. Self-admitted technical debt research: Problem, progress, and challenges. Ruan Jian Xue Bao/Journal of Software, 2022, 33(1): 26–54 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6292.htm [doi: 10.13328/j.cnki.jos.006292]

[56] Geng MY, Wang SW, Dong DZ, Gu SZ, Peng F, Ruan WJ, Liao XK. Fine-grained code-comment semantic interaction analysis. In: Proc. of the 30th IEEE/ACM Int'l Conf. on Program Comprehension (ICPC). Pittsburgh: IEEE, 2022. 585−596. [doi: 10.1145/3524610.3527887]

[57] Liu ZY, Chen HC, Chen XP, Luo XN, Zhou F. Automatic detection of outdated comments during code changes. In: Proc. of the 42nd IEEE Annual Computer Software and Applications Conf. (COMPSAC). Tokyo: IEEE, 2018. 154−163. [doi: 10.1109/COMPSAC.2018.00028]

附中文参考文献:

[51] 黄袁, 贾楠, 周强, 陈湘萍, 熊英飞, 罗笑南. 融合结构与语义特征的代码注释决策支持方法. 软件学报, 2018, 29(8): 2226–2242. http://www.jos.org.cn/1000-9825/5528.htm [doi: 10.13328/j.cnki.jos.005528]

[55] 郭肇强, 刘释然, 谭婷婷, 李言辉, 陈林, 周毓明, 徐宝文. 自承认技术债的研究: 问题、进展与挑战. 软件学报, 2022, 33(1): 26–54. http://www.jos.org.cn/1000-9825/6292.htm [doi: 10.13328/j.cnki.jos.006292]

王潮(1996−), 男, 博士生, 主要研究领域为开源软件开发, 软件仓库挖掘.

周明辉(1974−), 女, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为软件仓库挖掘, 开源软件生态系统.

徐卫伟(1999−), 男, 博士生, 主要研究领域为软件仓库挖掘, 开源生态模式与机制.