

自承认技术债的研究: 问题、进展与挑战^{*}

郭肇强^{1,2}, 刘释然^{1,2}, 谭婷婷³, 李言辉^{1,2}, 陈林^{1,2}, 周毓明^{1,2}, 徐宝文^{1,2}

¹(计算机软件新技术国家重点实验室(南京大学), 江苏 南京 210023)

²(南京大学 计算机科学与技术系, 江苏 南京 210023)

³(北京字节网络技术有限公司, 北京 100086)

通信作者: 李言辉, E-mail: yanhuili@nju.edu.cn; 周毓明, E-mail: zhouyuming@nju.edu.cn



摘要: 技术债是一个指以牺牲长期代码质量为代价来实现短期项目目标的隐喻. 其中, 那些由开发者有意引入项目中的技术债被称为自承认技术债 (self-admitted technical debt, SATD), 通常以代码注释的形式存在于软件项目中. SATD 的存在给软件质量和鲁棒性带来了巨大挑战. 为了识别并且及时地偿还 SATD 来保障代码质量, 研究者从特性分析和识别模型两方面进行了大量研究并且取得了较大的进展. 与此同时, 相关研究工作中仍存在一些亟待解决的挑战. 对近年来国内外学者在该领域的研究成果进行系统性的总结. 首先, 描述自承认技术债的研究问题. 然后, 分别从特性分析和识别模型两方面总结相关的研究进展, 并对具体的理论和技术途径进行梳理. 接着, 简要介绍技术债的其他相关技术. 最后, 指出目前该领域研究过程中面临的挑战并给出建议的研究方向.

关键词: 技术债; 自承认技术债; 代码注释; 软件维护; 质量保障

中图法分类号: TP311

中文引用格式: 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>

英文引用格式: 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). <http://www.jos.org.cn/1000-9825/6292.htm>

Self-admitted Technical Debt Research: Problem, Progress, and Challenges

GUO Zhao-Qiang^{1,2}, LIU Shi-Ran^{1,2}, TAN Ting-Ting³, LI Yan-Hui^{1,2}, CHEN Lin^{1,2}, ZHOU Yu-Ming^{1,2}, XU Bao-Wen^{1,2}

¹(State Key Laboratory for Novel Software Technology (Nanjing University), Nanjing 210023, China)

²(Department of Computer Science and Technology, Nanjing University, Nanjing 210023, China)

³(Beijing ByteDance Co. Ltd., Beijing 100086, China)

Abstract: Technical debt is a metaphor that refers to sacrifice the long-term code quality to satisfy the short-term goals. In particular, the technical debts introduced intentionally by developers are called self-admitted technical debt (SATD), which usually exist in software projects in the form of code comments. The SATDs bring great challenges to quality and robustness of software. In order to facilitate finding and paying back them as soon as possible for assuring software quality, in recent years, great progress has been made in the field of investigating the characteristics of SATD and proposing the identification models for SATD. Nevertheless, it is still challenging to apply them in practice. This paper offers a systematic survey of recent research achievements in SATD. First, the research problems are introduced in this field. Then, the current main research work is described in detail. After that, related techniques are discussed. Finally, the opportunities and challenges in this field are summarized and the research directions in the future are outlined.

Key words: technical debt; self-admitted technical debt (SATD); code comment; software maintenance; quality assurance

* 基金项目: 国家重点研发计划 (2018YFB1003901); 国家自然科学基金 (61772259, 61872177)

郭肇强和刘释然为共同第一作者.

收稿时间: 2020-08-16; 修改时间: 2020-10-08, 2020-11-06, 2020-11-23; 采用时间: 2020-12-14; jos 在线出版时间: 2021-01-15

软件开发团队在进行软件产品的研发时有一个共同的目标, 那便是能够在指定的 (时间或者资金) 预算内交付高质量无缺陷的软件产品^[1,2]. 为了实现这一目标, 开发团队需要合理分配开发资源并且制定统一的编程规范来保证最终交付产品的质量. 然而, 现实的软件开发过程往往不尽如人意, 其中充满了各种未知的挑战 (如客户需求更改^[3]、竞争者的压力^[3,4]、截止日期提前^[1,5]等). 开发团队不得不调整开发方案, 尽可能使用简单的变通方案 (如硬编码一个参数^[6], 或者只完成部分高优先级的功能) 来快速应对上述挑战. 但与此同时, 方案的调整可能会在有意或者无意中为软件的质量和鲁棒性埋下未知的隐患, 这在很大程度上增加了后期软件演化中的维护和修复成本^[7]. 这种情况被研究者称之为技术债 (technical debt, TD)^[8].

技术债是一种隐喻, 其定义为以牺牲长期的代码质量为代价来换取实现短期的项目目标^[8]. 正如其名, 这些技术债作为软件中存在的隐患, 是需要开发者在后期维护中进行偿还的债务. 已有研究表明, 技术债在软件开发过程中是非常常见并且是不可避免的^[3,4,9], 并且会对软件质量产生负面影响^[5]. 因此, 研究技术债的特性并且及时地识别出已有的技术债对于提高软件质量意义非凡. 从 2011 年开始, 每年会举行以管理技术债为主题的国际研讨会 (Int'l Workshop on Managing Technical Debt) 来跟进技术债相关的研究^[10-17].

根据技术债的引入方式, 技术债可以分为无意引入和有意引入两个主要类别.

(1) 无意引入的技术债 (TD introduced unintentionally). 开发者由于缺乏经验而书写的低质量的代码或者是疏忽引入的技术债, 他们实际上并没有意识到已经引入了技术债^[18]. 例如, 项目管理者要求临时提供对新框架的支持. 然而开发人员对于该新引入框架的掌握并不熟练, 没有按照框架预先设计的标准来进行编程, 在这过程中便会无意间引入一些鲁棒性较差的代码债. 无意引入的技术债在现实中是不可预知的, 只有在技术债所导致的缺陷被发现后才能逆向溯源来移除它们.

(2) 有意引入的技术债 (TD introduced intentionally). 在权衡短期效益与长期维护成本之后, 开发者主动引入一类技术债来临时满足当前软件开发的需求, 例如开发者硬编码若干个参数来临时满足某个功能的正常运行并抢先上架软件来获取用户和市场. 由于开发者在引入这类技术债时明确知道它们的存在, 因而其也被称为自承认技术债 (self-admitted technical debt, SATD)^[4]. 对这类技术债进行研究可以比较容易地找出软件中明显可能会出错的地方并且及时作出决策来执行明确的软件维护任务.

自承认技术债是可以进行主动溯源的, 并且对其进行移除可以明显提升软件的质量. 此外, 这类技术债在项目普遍存在 (例如, 在 Potdar 等研究的项目中, 有超过 30% 的文件中包含 SATD^[4]). 因此, 研究者在近年来开始着重对这类技术债进行研究^[2,4,6,19,20]. 目前研究者主要从代码注释 (code comments) 中提取自承认技术债. 这是因为开发者在引入这类技术债时为了降低后期对代码进行修复时的缺陷定位成本, 通常会用注释标明包含技术债的代码. 换句话说, 自承认技术债正是开发者通过书写代码注释所承认的. 因此, 目前几乎所有的对自承认技术债的研究 (包括特性分析和识别模型) 都是在代码注释的基础上进行的.

本文主要对自承认技术债的已有研究工作进行综述. 为方便表述和阅读, 本文接下来的部分均使用 SATD 来指代自承认技术债. 从 2014 年首次提出 SATD 的概念到 2020 年 8 月为止, 学术界已有 40 多篇文献从不同的角度对 SATD 进行研究. 据我们所知, 目前 SATD 领域的文献几乎全部发表于外文国际期刊或会议. 国内学术界对该领域的介绍较为稀少, 所有的国内文献均是关于技术债 (而非 SATD) 的研究^[21-25]. 此外, 国外文献中虽然有多篇综述性的工作^[26-28]总结了技术债的研究, 但是仅有一篇介绍 SATD 的综述性文章^[20]而国内尚缺少相应的总结性的综述性工作. 为了弥补国内学术界在该领域中相对空白的认识, 同时为国内研究者后续相关研究工作的进行提供必要的参考, 本文对 2014 年提出 SATD 概念以来的研究历史进行全面地回顾, 对现有的研究进展进行分类地介绍和总结, 并在此基础上指出该领域所面临的挑战. 相比于已有的综述论文^[20], 本文中总结的文献数目接近前者的 2 倍, 细致地从 SATD 的特性分析和 SATD 的识别模型两个研究点向读者展示了该领域 (尤其是自 2019 年来) 的进展. 一方面, 介绍了研究者如何从不同的维度刻画 SATD 的特性 (如分布特征、引入、移除和对软件质量的影响等); 另一方面, 对现有的多个 SATD 识别模型的识别原理和性能比较进行总结. 更重要的是, 根据当前的研究现状总结出新的研究挑战并且指出了在未来可行的研究方向. 具体结构组织如下.

本文在第 1 节首先概述 SATD 识别领域主要的研究问题; 接着在第 2 节说明本综述的文献检索方式和文献汇总信息; 然后在第 3 节介绍对 SATD 的特性分析实证研究; 第 4 节介绍对 SATD 的识别模型的研究; 接着在第

5 节介绍技术债识别领域的其他研究方法;在第 6 节分析 SATD 领域的研究所面临的挑战;最后在第 7 节总结本文的内容。

1 研究问题

1.1 SATD 的知识概要

本小节通过提问的方式介绍 SATD 的相关知识概要,包括对 SATD 的定义、研究 SATD 的原因、SATD 的产生原因和主要的研究方式。

1.1.1 什么是 SATD?

软件项目在维护和演化过程中总是存在着一对矛盾的客体。一方面,开发团队或组织的目标和宗旨是在时效期间内能够及时地交付高质量的软件制品;另一方面,软件在被开发和迭代的过程中会由于诸多不可预期的因素而导致开发者编写无法满足实际需要的低质量代码(not-quite-right code)。Cunningham^[8]首先使用“技术债”这一术语来暗指软件中存在的这种无法避免的软件问题。正如其名,开发者在软件维护时需要技术债进行“偿还”(即移除)来保障软件正常的运作(包括功能、性能、鲁棒性等方面)。然而,这会带来额外的软件维护成本。

SATD 是技术债家族中的一种。这类技术债是被项目的技术人员(如开发者或管理者)所承认的。换句话说,他们知道项目中 SATD 的存在,甚至也知道具体的债务内容、与债务相关的代码位置、债务产生的原因以及偿还债务的解决方案等,因为这些 SATD 是被人为有意地添加到项目中的用以临时满足当前的开发需要。SATD 涵盖了项目中各方面存在的问题,包括糟糕的架构设计、不正确的测试代码、缺失的文档说明、有缺陷的代码片段、算法次优实现(如硬编码)等。为了便于后期进行 SATD 的移除,开发者会主动地在项目中(如代码里)标注这些债务。

1.1.2 为什么要研究 SATD?

结合现有研究,本小节总结出人们研究 SATD 主要的原因。

首先, SATD 的存在会给软件项目带来负面的影响^[29]。对 SATD 的研究可以有效地为软件质量提供保障。具体来说, SATD 的存在会带来以下两点影响。

(1) 降低软件质量^[7,30]。软件中的 SATD 会严重影响软件的质量及健壮性。Xavier 等人^[7]指出,包含 SATD 的代码的运行性能会出现下降现象并且更容易出现错误。此外已有研究^[30-32]指出技术债与软件制品生命周期中的各种活动(如需求、代码、设计、测试和文档等)有关联。因此, SATD 可能会在这些方面降低软件制品的质量。

(2) 增加维护成本^[5,7,33,34]。包含 SATD 的代码通常没有良好的设计逻辑,只是一种临时的解决方案。团队中其他开发人员难以对其进行阅读和理解,从而会使代码演化速度减慢^[7]。要偿还项目中的 SATD,开发者通常需要对代码进行较为复杂的变更^[5]来解决之前遗留的问题。开发团队在偿还某些 SATD 时需要进行重复性的维护工作^[7],从而会带来巨大的返工工作量^[34]。此外, SATD 的偿还利息^[33]会随着时间的推移逐渐增加,这也提高了软件的维护成本。

其次, SATD 在软件项目中扮演着重要角色,因此不能够忽略对其进行研究。其重要性体现在以下两点。

(1) 普遍性^[4,5,9,34]。SATD 并非是某个软件特有的特征,而是广泛存在于各种领域的软件项目产品中。例如, Bavota 等人^[9]对 159 个 Java 开源项目(分属于 Apache 和 Eclipse 系统)进行调研并且发现 SATD 普遍存在于每个项目中。现有研究^[1,4,5,18,34]均在其研究的项目上发现了大量的 SATD。因此,对于 SATD 的研究可以为绝大多数软件提供质量保障手段。

(2) 不可替代性^[4,6,18,35]。有些软件缺陷仅以 SATD 的形式存在于代码中,它们不容易被其它研究手段识别(如代码度量^[18])出来。对于这类缺陷,通过研究 SATD 是一种十分行之有效的解决方案。虽然可以使用自动静态分析和代码坏味道来识别 TD,但是这些方式可能不会指出相关的 TD 或者不报告开发人员认为是 TD 的代码片段^[4,35],而 SATD 所描述的软件问题基本上是确实存在的^[6]。此外,只要一个软件缺陷被开发者自己承认,那么它便会在软件项目中留下踪迹(被标注)^[4]。通过研究 SATD 来识别这类软件问题相较于其他方式更加简单和方便。这说明对 SATD 的研究无法被其他研究所替代。

1.1.3 哪些因素会导致 SATD ?

在软件演化过程中, 许多因素都会促使开发者在项目中主动引入 SATD 来暂时满足某些短期的目标. 根据对现有文献中总结, SATD 主要由如下原因造成.

- (1) 项目成本被缩减^[3,4,18]. 项目的预算成本 (包括人力和财力) 随着软件开发的进行在不断变化, 当开发成本在不可避免的情况下被缩减时会导致开发者主动搁置一些功能并作出标记. 这些被搁置的功能成为软件中的 SATD.
- (2) 快速响应客户需求^[3,4]. 客户对项目提出新的需求会带来开发成本的提升. 为了尽快满足客户的需求, 开发团队可能需要在某些功能上采用捷径 (快速解决方案) 来加速开发进程. 捷径为软件带来 SATD.
- (3) 尽快抢占市场^[3,4]. 竞争者为了抢先占领市场, 会预先着重实现主要的产品功能, 对于不重要的部分先做出简单处理. 此时会在项目中添加 SATD.
- (4) 发布日期临近^[1,5,18,34,36]. 由于实现新软件功能的截止时间已经临近, 开发人员可能会插入违反模块化 (violation of modularity) 的行为 (包含了 SATD) 来快速发布可用的软件版本.
- (5) 尽快实现新功能^[1]. 软件的开发过程充满不确定性, 经常会需要根据需求的变化进行代码变更. 为了能够快速实现某一新功能, 开发者通常会开发出简单的样例代码来测试功能的效果. 这些简单的未完善代码 (如缺少异常捕捉等) 便成为软件中 SATD.

1.1.4 如何研究 SATD ?

SATD 在被开发者引入项目中时已经被其主动地记录在项目里. 由于代码注释有助于记录、交流和理解程序中出现的各种问题, 并在程序演化中起着重要作用^[25,37-40]. 多数开发者会在包含技术债的代码实体附近用代码注释来描述 SATD 的内容. 表 1 展示了一些研究中涉及的 SATD 样例. 从样例中可以看出, 这些注释在表现技术债时是十分直观容易理解的, 它们或者会描述存在的问题^[4]; 或者会对代码提出疑问^[2]; 或者给出建议的解决方法^[9].

表 1 自承认技术债的注释样例

SATD实例	文献来源
// EATM Demand create metadata; needs to depend on processing mode...	[41]
// TODO this is such a hack it is silly	[4]
TODO: this isn't quite right but it's ok for now	[42]
/* TODO: really should be a separate class */	[1]
//I can't get my head around this; is encoding treatment needed here?	[2]
//FIXME: to be moved to ApiResponseHelper	[9]

Potdar 等人^[4]首次提出 SATD 的概念并使用源代码注释来调研软件中存在的 SATD. 随后, 绝大多数研究者们都是通过代码注释来进行 SATD 的相关研究. 例如, Maldonado 等人^[1]根据注释的所表达的不同含义将 SATD 划分为不同类型 (如设计债和文档债); Bavota 等人^[9]对不同类型的注释进行统计来研究 SATD 的类型分布情况; Zampetti 等人^[43]根据 SATD 注释在软件版本控制系统中的变化来调研 SATD 的根因和移除情况; 文献 [2,6,41,44] 的研究者通过设计不同的分类模型对软件项目中的注释进行分类来识别目标项目中的 SATD.

1.2 SATD 的研究内容

综合现有与 SATD 相关的文献, 学术界对该领域的研究主要集中在分析 SATD 的相关特性^[1,4,5,9,45,46]来提高对 SATD 的理解, 以及设计准确的 SATD 识别模型^[2,6,30,41,44,47]来识别出代码中存在的 SATD 实例这两个方面. 以下分别从这两个方面总结 SATD 领域的研究内容研究和其存在的主要挑战. 具体内容将分别在第 3 节和第 4 节详细介绍.

1.2.1 分析 SATD 相关特性 (见第 3 节)

理解 SATD 的特性能够帮助我们更有效地对其进行管理, 从而更好地对软件质量提供保障. 目前研究者们进行大量的实证研究主要从图 1 所示的 5 个维度对 SATD 展开研究.

- (1) 种类^[1,2,48,49]. 对 SATD 按照其具体含义划分为不同种类, 这将帮助开发者有针对性地为每种 SATD 采取不

同的解决方案, 提高 SATD 的偿还效率.

(2) 分布^[1,4,9,34,42]. SATD 的分布是指每种 SATD 所占的比重. 通过比重可以量化不同 SATD 的重要程度. 在有限的开发资源条件下帮助开发者优先选择处理策略.

(3) 根因^[4,9,50]. SATD 是被开发者主动引入项目中的, 通过对已有 SATD 的引入原因进行调查可以总结出常见的引入 SATD 的规律并给将来的开发者提出相应的建议尽量避免 SATD 的引入.

(4) 移除^[9,42,43,50-53]. 正确移除 (偿还) SATD 是研究者和开发者的最终目标. 通过对移除方式的调研可以总结出最佳的移除方案帮助不同开发者能够快速偿还相应的债务.

(5) 影响^[4,5,9,33,34,54]. SATD 存在于软件产品中, 会对软件造成各种影响. 通过对影响因素进行调研可以帮助开发者认知 SATD 的危害并在开发中慎重考虑引入 SATD 是否合算.



图 1 对自承认技术债的理解维度

1.2.2 构建 SATD 识别模型 (见第 4 节)

本小节对 SATD 识别方法的流程进行介绍. 根据对现有文献的总结, 多数研究者将 SATD 的识别视为文本的二分类任务^[2,6,41,51,55]. 其流程主要包含抽取注释、文本预处理、构建分类模型、评估分类模型这 4 个基本步骤, 如图 2 所示.

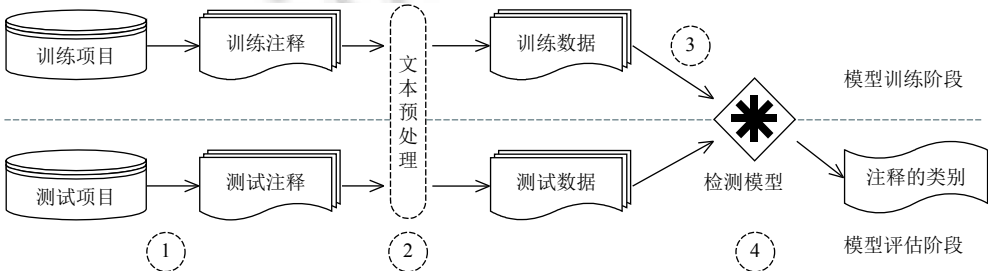


图 2 SATD 的方法流程概览

(1) 抽取注释. 这一步从项目代码中提取注释数据. 首先把项目代码下载下来; 然后使用现有工具 (如 JDeodorant^[56] 和 srcML^[57]等) 抽取项目中的注释信息, 包括注释的文本、类别和上下文信息等. 对于抽取好的注释, 现有研究是通过人工阅读注释内容的方法来手动记录注释的标签 (SATD 或者 non-SATD).

(2) 文本预处理. 代码注释大多是用自然语言文本构成, 其结构和形式复杂多样^[2,6] (如一个单词在不同的句子中可能具有多种形式). 这一步对注释中的文本进行统一化处理, 便于后续构建模型.

(3) 构建分类模型. 研究者在这一步使用各种技术来构建 SATD 分类模型, 包括但不限于有监督的自然语言处理技术^[2]、传统机器学习技术^[6]、深度学习技术^[41]以及无监督的模式匹配技术^[4,18,44]等. 多数研究设计的模式是二分类, 即仅识别注释是否包含 SATD, 而不考虑 SATD 具体的种类.

(4) 评估分类模型. 研究者在这一步使用测试数据来评估模型的有效性. 因为模型会输出分类结果, 所以常用的评估指标有精确率 (precision)、召回率 (recall) 和其调和平均数 F1 等.

经过上述步骤, SATD 识别模型会对测试项目中的每个注释实例预测出一个类别 (SATD 或者 non-SATD). 这个结果可以向开发者推荐项目中存在的 SATD 并提醒他们及时对其进行处理. 需要说明的是, 对注释进行文本预处理 (图 1 中的虚线框) 的步骤并非必须的, 有些方法 (如 Potdar 等人^[4]提出的 Pattern 方法) 在设计模型时可以直接对原始注释进行分类. 另外值得注意的是, 对于无监督模型的设计, 因为其不需要带标签的训练数据, 所以这类模型没有模型训练阶段.

2 文献检索

2.1 文献筛选与检索

对 SATD 的研究近年来成为软件质量保障和软件维护领域的热点研究问题. 本文的参考文献力图覆盖 7 年来与 SATD 相关的主要研究工作, 同时包含技术债领域的其他相关研究方法的主要文献. 具体来说, 本文使用以下的步骤来进行相关文献的检索和筛选.

(1) 检索目标: 谷歌学术搜索引擎 (Google scholar)、DBLP Computer Science Bibliography、ACM Digital Library、IEEE Xplore Digital Library 以及 Springer Link Online Library 等论文数据库. 检索内容主要包括软件工程-系统软件-程序设计语言方向的国际一流会议 (ESEC/FSE、ICSE、ASE 等) 和一流期刊 (TSE、TOSEM 等) 以及其他重要的会议和期刊如 MSR、ICSME 等录用的文章.

(2) 检索关键词: 包括“self-admitted technical debt”以及“technical debt”与“code comment(s)”的组合. 我们在目标数据库中对文献的题目使用上述关键词进行简单搜索.

(3) 检索起止时间: SATD 在 2014 年被提出, 因此我们检索 2014–2020 年共 7 年之间的文献.

(4) 文献审查: 首先对检索出来的文献进行人工筛选, 去除不符合研究主题的文献以及从不同数据库中检索出的重复文献; 然后检查选中论文中的参考文献列表补充上未检索到的文献; 最后在第 2.2 节中对所选文献进行汇总.

2.2 历史文献的汇总

经过文献的检索与筛选, 本文收集了 SATD 领域的学术论文共 41 篇. 这些研究论文将会按照其内容分类分别在第 3 节和第 4 节进行重点介绍. 对于其他种类的技术债的相关研究, 我们在第 5 节中对其进行分类和介绍.

图 3 展示了 7 年来 SATD 领域的研究论文发表的数量分布情况. 从图中可以看出, 该领域的逐年论文发表数量整体上呈现出上升的趋势, 尤其是近两年 (2019–2020 年). 这说明 SATD 这个研究领域正越来越受到研究者的重视. 截至 2020 年上半年, 相关论文数量达到 10 篇, 继续保持着较高的活跃度. 图 4 列出了 SATD 领域的研究论文在出版物上的发表分布情况. 可以看出, 在 CCF 划分的软件工程领域重点会议上有 15 篇, 包括 ICSE 论文 2 篇^[49,58]、ICSME 论文 5 篇^[4,30,45,50,52]、SANER 论文 3 篇^[5,53,54]、ICPC 论文 1 篇^[46]、MSR 论文 3 篇^[7,9,43]、APSEC 论文 1 篇^[59]. 重点期刊上有 8 篇, 包括 TSE 论文 2 篇^[2,60]、TOSEM 论文 1 篇^[41]、EMSE 论文 2 篇^[6,51]、IST 论文 1 篇^[47]、JSS 论文 2 篇^[20,42]. 另外还有其它国际出版物 (非 CCF 推荐列表) 上的论文共 18 篇, 包括 arXiv 论文 3 篇^[44,48,61]、MTD 论文 2 篇^[1,18]、TDA 论文 2 篇^[33,34]、SBES 论文 1 篇^[36]、SBQS 论文 1 篇^[62]、SBSI 论文 1 篇^[63]、IWESep 论文 1 篇^[64]、ICEIS 论文 1 篇^[65]、ITNG 论文 1 篇^[66]、ACIT 论文 1 篇^[67]、ACIT-CSII-BCD 论文 1 篇^[19]、Algorithms 论文 1 篇^[39]、SEAA 论文 1 篇^[68]、Access 论文 1 篇^[69]. 这表明对 SATD 研究是软件工程领域重要的研究方向. 图 5 列出了对 SATD 研究的进展时间轴. 现有文献着重从特性分析和识别模型两个方面进行研究. 在进展前期 (约 2018 年之前), 由于缺乏对 SATD 的理解, 研究者们着重展开实证研究来分析 SATD 的特性. 在研究近期 (约 2018 年至今), 研究者对 SATD 有一定的了解, 着重于设计模型识别项目中的 SATD.

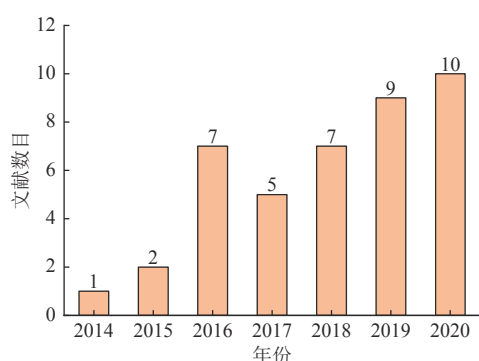


图 3 SATD 的相关研究文献的年代分布

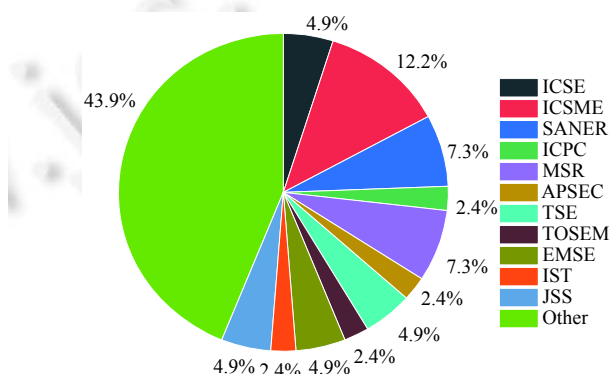


图 4 SATD 的相关研究文献的出版物分布

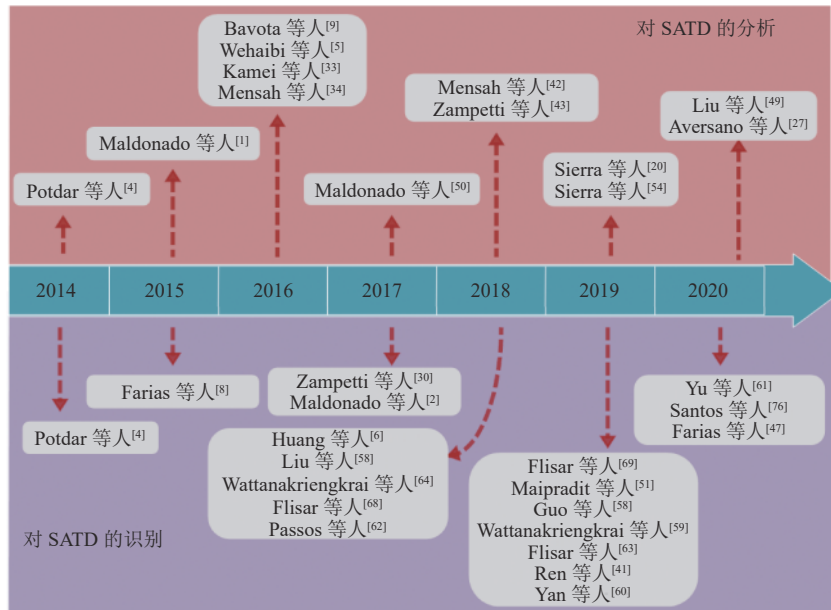


图 5 SATD 相关研究的进展时间轴

3 对 SATD 特性的分析

本节介绍对 SATD 的特性进行实证研究的进展。由于现有文献从不同的角度切入研究, 本节对其分类并主要从 SATD 的种类、分布、根因、移除和影响 5 部分展开介绍。然后汇总当前研究的用到的数据集。

3.1 SATD 的种类 (type)

对技术债的分类研究是一项重要的工作, 这是因为不同类型的 SATD 对应有不同的解决方法。在确定技术债所属的种类后可以为其量身定制的合适的解决方案来偿还该债务^[1]。此外, 它还帮助我们理解使用代码注释来识别技术债务时可能遇到的机遇和限制^[1]。

2014 年, Alves 等人^[31]依据债务的本质 (nature) 提出了划分技术债种类的定义, 这些定义描述了指定种类的技术债所具有的特征或者满足的条件, 以及偿还债务的一些可行的办法。他们的研究一共汇总了 13 种被不同研究者提出的不同类型的技术债, 为随后的研究技术债和 SATD 的学者提供了较为清晰的种类划分参考。

据我们所知, 现有已知 SATD 种类类别 (如需求债) 基本上来源于 Alves 等人^[31]的种类汇总。然而, 不同于广义的技术债, SATD 是以代码注释的形式出现在软件项目中的。因此, 研究者需要根据注释所表达的不同含义将其划分为不同的类别。在现有的 SATD 种类研究中^[1,2,9,48,49]总共出现了 10 种类型的 SATD。他们通过介绍每种类型的 SATD 具有的特征并列举出对应的注释样例来为 SATD 的种类划分提供一些指导性的建议。为了帮助读者更清晰的理解每种 SATD 的含义, 本文在表 2 以相似的方式详细介绍每种 SATD 的定义, 指明推荐的偿还方法, 并列举出对应的样例。同时在表 2 对每种 SATD 类型的划分和界定进行了更加细致清晰的总结。

(1) 需求债 (requirement debt)^[1,2,9,48,49]

定义: 这类债务注释指出某个方法、类或者程序的结构不完整, 其中有部分的需求没有被实现或者仅仅是部分被实现。同时, 一些需求虽然已经被实现但是不完全满足所有非功能性需求 (如安全性、性能等)^[31]。

偿还: 按照要求将缺失的代码补充完整。

样例: 在如下样例中, 该注释指出缺少方法 getClassname。

“/TODO no methods yet for getClassname” — [from Apache Ant]^[1]

表 2 现有研究中 SATD 的种类总结

类型 [文献]	定义概述 ^[31]	高频率关键词 ^[18]	● 出现某类SATD的场景 ^[27]	● 预防措施 & ※解决方案 ^[28]
需求债 [1, 2, 9, 48, 49]	代码中的方法、类或程序不完整	Scope, Business, Artifact, Conception, Requirements, Specification, ...	● 仅实现部分需求 ● 实现针对部分情况的需求 ● 已实现但未完全满足所有非功能性需求的需求	● 定义明确的需求 ● 遵循项目计划 ● 遵循明确的项目流程 ● 资源合理分配 ● 领域声明 ※补充缺失代码
设计债 [1, 2, 9, 48, 49]	代码中存在设计问题(如方法过长)	Component, encapsulation, design, method, metric, code, coupling, Cohesion, Class, clone, Polymorphism, Object, ...	● 违反面向对象设计原则 ● 出现代码异味 ● 复杂类或方法 ● 设计复杂度过高	● 遵循项目计划 ● TD监测 ● 定义明确的体系结构 ● 定义明确的需求 ● 风险和影响分析 ※代码重构
架构债 [48]	架构中存在的问题	Coupling, Aspect, Base, Component, Environment, Gui, Function, architecture, Software, System, Query, package, Procedure, Polymorphism, Library, Interface, Hierarchy, Connector, Layer, Link, Message, Module, Tier, ...	● 违反模块性 ● 复杂的架构行为依赖关系 ● 架构符合性问题 ● 系统级的结构质量问题 ● 体系结构策略和模式不统一 ● 缺乏处理相互依赖的资源 ● 缺乏解决非功能性需求 ● 体系结构技术的实现不成熟	● 遵循明确的项目流程 ● 遵循项目计划 ● 定义明确的体系结构 ● 培训 ● 使用合适的技术版本 ※调整代码适应架构特性
算法债 [49]	算法实现没有达到最优情况	Algorithm, optimize, ...	● 次优的算法逻辑实现	● 代码标准化 ● 采用良好的编程习惯 ● 代码评估 ※优化算法实现
代码债 [9, 48]	代码存在易读性问题	Algorithm, Code, Component, Deployment, Gui, Query, class, method, file, clone, Metric, Logic, Language, Compilation, Declaration, Variable, Exception, Legibility, Loop, Performance, Rapidity, Release, Version, Speed, ...	● 不必要的重复和复杂代码 ● 降低代码可读性的错误样式 ● 过于复杂的代码	● 代码评估 ● 定义明确的需求 ● 代码标准化 ● 采用良好的编程习惯 ● 遵循项目计划 ※按照提示修改代码
文档债 [1, 9, 48, 49]	代码中缺少说明文档	Documentation, Diagram, Use case, Template, Model, Manual, Metadata, Remarks, Survey, ...	● 丢失文档 ● 不充分文档 ● 过期文档 ● 不完整文档	● 遵循项目计划 ● 适当的任务分配 ● 培训 ● 定义明确的需求 ※添加说明文档
兼容债 [49]	代码依赖版本不兼容	Compatibility, ...	● 不成熟的依赖	● 定义明确的体系结构 ※更换依赖版本
缺陷债 [1, 9, 48, 49]	代码实现中存在缺陷	Bug, Deployment, defect, Error, Exception, ...	● 修复软件系统中缺陷的后发处理决策	● 培训 ● 版本控制 ● 定义明确的体系结构 ※定位修复已知缺陷
构建债 [48]	代码构建复杂或者耗时	Artifact, Aspect, Construction, Deployment, Environment, Polymorphism, Hierarchy, Compilation, Risk, Tool, ...	● 构建无价值代码 ● 依赖导致缓慢的构建过程 ● 手动构建过程	● 遵循项目计划 ● 定义明确的体系结构 ※修改构建文件
测试债 [1, 9, 48, 49]	缺少或者需要改进测试代码	Test, Reliability, Precision, Coverage...	● 测试覆盖率不足 ● 缺少部分测试 (如集成测试) ● 延期测试 ● 缺乏测试用例规划	● 创建测试用例 ● 遵循项目计划 ● 重构 ● 风险和影响分析 ● 培训 ● 采用良好的编程习惯 ※修改测试用例

(2) 设计债 (design debt)^[1,2,9,48,49]

定义: 这类债务注释指出了代码中存在设计问题如代码放错地方、缺少抽象层、方法冗长、实现太差、高耦合度的类或临时解决方案等. 可见, 所有的设计债都是非最优决策的结果^[48].

偿还: 通过代码重构或者对其重新实现来解决.

样例: 在如下样例中, 该注释指出该方法设计太复杂需要进行拆分.

“TODO: - This method is too complex, lets break it up” — [from ArgoUML]^[1]

(3) 架构债 (architecture debt)^[48]

定义: 这类债务指出了代码架构中存在的问题. 大多数的架构债都表现出违反模块性的特点, 少数的架构债是由于使用了过时的技术导致的^[48].

偿还: 通常来说软件架构很难改变, 开发者只能使调整项目本身代码以适应软件架构的特性.

样例: 在如下样例中, 该注释说明将代码移入自己的模块比较好.

“It would be good if these were moved into their own module...” — [from Camel]^[48]

(4) 算法债 (algorithm debt)^[49]

定义: 这类债务特指深度学习框架中的次优算法. 由于基于深度学习的项目通常比较消耗计算资源, 这类项目中的次优的算法会明显降低系统的效率^[49].

偿还: 对相应的算法进行优化.

样例: 在如下样例中, 该注释说明需要对单元 kCostPerUnit 进行优化.

“TODO (zakaria): optimize kCostPerUnit” — [from TensorFlow]^[49]

(5) 代码债 (code debt)^[9,48]

定义: 这类债务注释指出了项目中难以阅读和理解从而维护难度较高的代码. 通过检查项目的源代码并且考虑与错误的编码实践相关的问题可以识别这种债务.

偿还: 根据注释的提示对代码进行改进.

样例: 在如下样例中, 该注释指出导致代码不可维护的问题.

“This will lead to very unmaintainable code. We absolutely do not want to have nested retries for different contexts.” — [from Hadoop]^[48]

(6) 文档债 (documentation debt)^[1,9,48,49]

定义: 这类债务注释标记了那些缺少文档说明支持或者文档质量较低的程序. 虽然这些程序可以正常工作, 但不能满足软件项目的某些质量标准的文档会导致不同的开发者产生代码理解上的困难.

偿还: 在相应位置补充完善高质量的说明文档.

样例: 在如下样例中, 该注释指出需要添加文档说明该实现的原因.

“// TODO Document the reason for this” — [from Apache Jmeter]^[1]

(7) 兼容债 (compatibility debt)^[49]

定义: 这类债务注释指出某项目的代码依赖于不能提供所有合格服务的其他项目, 即依赖版本不兼容. 这类债务的存在可能导致项目在随后的演化中产生更多代码变更^[49]. 可以看出, 兼容债和架构债之间存在一定的关联性, 即他们都可能是由外部的代码引发的问题.

偿还: 开发者需要更换依赖版本来解决此债务.

样例: 在如下样例中, 该注释指出代码会在装有 CUDA 7.0.18 的 Linux 系统上运行失败.

“Moved to common.cpp instead of including boost/thread.hpp to avoid a boost/NVCC issues (#1009, #1010) on OSX. Also fails on Linux with CUDA 7.0.18.” — [from Caffe]^[49]

(8) 缺陷债 (defect debt)^[1,9,48,49]

定义: 这类债务注释指出了代码中存在缺陷, 即该段代码中的预期功能实现错误. Bavota 等人^[9]扩展了缺陷债的定义, 他们指出其不仅包括已知的缺陷, 还包括代码中可能导致缺陷的问题 (例如高访问次数导致的代码崩

溃等).

偿还: 进行代码调试来定位缺陷并且对其进行修复.

样例: 在如下样例中, 该注释就明确指出了该方法中存在缺陷 (bug).

“// Bug in above method” — [from Apache Jmeter]^[1]

(9) 构建债 (build debt)^[48]

定义: 在这类债务中, 项目的构建过程可能包含对客户来说非常不必要的代码以及包含运行定义错误的依赖项. 这些问题会导致该过程将变得不必要的缓慢.

偿还: 根据描述修正构建文件来移除这类技术债.

样例: 在如下样例中, 该注释指出了缺失显式依赖导致的问题.

“Compiling for Fedora reveals a missing declaration for javax.annotation. Nullable. This is the result of a missing explicit dependency on...” — [from Hadoop]^[48]

(10) 测试债 (test debt)^[1,9,48,49]

定义: 这类债务注释指出当前代码中与测试相关的问题, 包括未运行计划的测试用例, 或测试套件中存在已知的缺陷 (例如代码覆盖率低).

偿还: 实现缺失的测试用例或者改进有问题的测试套件.

样例: 在如下样例中, 该注释指出需要启动一些合适的测试.

“//TODO enable some proper tests!!” — [from Apache Jmeter]^[1]

根据上文对 SATD 种类的介绍以及对相关文献^[1,2,9,48,49]的总结, 今后的研究者在进行相似的 SATD 种类研究时需要了解并且关注以下要点.

(1) SATD 种类划分的依据. 在现有研究中, 研究者给出的是不同 SATD 种类的模糊定义 (表 2), 他们通过总结一些同种类注释实例共有的特征 (如, 关键词 “document” 通常意味着文档债) 来解释每种 SATD 应当符合的条件或者存在的问题. 在这些研究中, 研究者首先需要理解一条注释所表达的含义; 然后将其与表 2 所描述的每个种类的特征进行比照; 最后人工判断与该注释的描述符合的 SATD 种类. 可以看出, 现有的研究中对 SATD 种类的划分依据是模糊的语言描述而非精确的形式化定义. 尽管今后的研究者仍可以使用类似的方式对种类进行种类划分, 但是会耗费大量的人工成本. 因此, 在今后的研究中能够提出标准统一的形式化定义对于降低人工 SATD 划分的成本十分有必要.

(2) SATD 种类划分的特点. 从当前研究 SATD 种类的文献^[1,2,9,48,49]可以看出, 不同研究者在划分 SATD 种类的时候存在统一性和差异性. 统一性体现在研究者们共同定义了某些种类的 SATD, 这表明这些种类的 SATD 是比较容易进行识别和划分的. 例如, 需求债和设计债被所有 (本文引用的) 研究者定义为应当被调查研究的种类. 差异性是由于研究者在调查实际项目时不同的认知和理解角度造成. 例如, Liu 等人^[49]在研究深度学习框架的项目时将算法债作为一种独立的 SATD 种类. 从他们的角度来看, 算法实现在深度学习框架中扮演者重要角色需要被重视. 然而, Li 等人^[48]在研究非深度学习框架项目时仅仅将算法问题归为代码债中的子类别. 此外, Liu 等人^[49]对深度学习框架项目划分出架构债的种类, 然而在非深度学习框架项目中, 这些债务通常会被归为设计债. 可见, 这种种类划分的差异性难以避免, 这种情况的存在会给 SATD 种类的划分工作带来挑战. 在后续的种类研究工作中, 研究者需要根据实际的研究项目的特征进行种类划分. 因此在本节中列出的 10 种类别仅仅是给相关研究者提供一些种类划分的参考依据. 研究者仍可以提出新的 SATD 种类以适应其研究目标的需要.

(3) SATD 种类之间的关联性. 虽然本文汇总出了 10 个种类的 SATD, 但是在实际进行种类划分时存在一条注释同属于多个种类的情况, 即不同种类的 SATD 之间存在关联性. 现有文献中设计的关联性有: 1) 设计债与缺陷债. 文献^[1]指出不合理的设计会使系统产生异常的行为 (即缺陷), 因此包含设计债的注释在一些情况下也可以划分为缺陷债. 例如, “FIXME: does not throw SQLException”. 2) 架构债、兼容债与构建债. 系统架构的选择不当 (架构债) 可能导致软件产生版本不兼容的问题 (兼容债), 进而导致软件构建失败 (构建债)^[49]. 例如, “Avoid the redundant direct dependency on log4j by the components. —[Camel-4331]”. 3) 部分代码债是由算法债导致的. 例如

“#query() does O(N) calls LinkedList#get() in a loop, rather than using an iterator. This makes query O(N^2), rather than O(N).” — [Hadoop-8866]”. 对存在关联性的种类划分需要以研究目的为准进行划分. 同时, 有些种类的 SATD 相比于其他种类有着十分鲜明的特点, 在进行划分时可以很明确地找到其对应的种类, 因此不具有关联性. 例如, 文档债明确指出了项目中文档的缺失而测试债指出了项目中缺少测试用例或者需要对其进行修改^[9]. 根据现有文献对不同种类的表述, 本文在图 6 中列出了 SATD 种类之间的关联性供研究者进行参考. 其中用箭头连接的 SATD 类型存在关联, 即一条注释实例可以划分为有关联的任意一种类型.

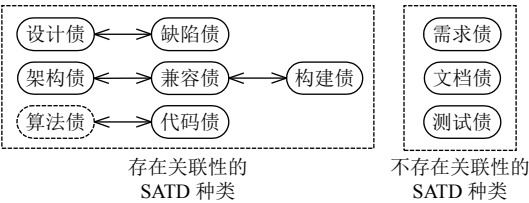


图 6 自承认技术债的种类之间的关联性

3.2 SATD 的分布 (distribution)

对 SATD 的分布进行量化有助于开发者明确地了解每种类型 SATD 对应问题的严重性、受关注程度^[4]以及是否需要优先分配稀缺的测试资源^[42]. 表 3 总结了相关研究中的种类分布信息.

表 3 SATD 在不同研究中的类型分布情况 (%)

类型	Maldonado等人 ^[1]	Bavota等人 ^[9]	Liu等人 ^[49]	Mensah等人 ^[34]	Li等人 ^[48]
代码债	—	30	—	—	39
缺陷债	4–9	20	2–7	—	3
设计债	42–84	13	24–65	57–68	5
文档债	0–5	10	0–16	—	22
需求债	5–45	20	4–31	—	3
测试债	0–7	7	0–8	—	18
兼容债	—	—	0–35	—	—
算法债	—	—	6–21	—	—
架构债	—	—	—	—	7
构建债	—	—	—	—	6

2014 年 Potdar 等人^[4]在 4 个项目上的结果显示, 大约 2.4%–31% 的文件、0.4%–33% 的 Java 类和 0.3%–2.6% 的 Java 方法会包含 SATD. Maldonado 等人^[1]指出设计债 (42%–84%) 和需求债 (5%–45%) 是最为常见的 SATD 类型. Mensah 等人在文献 [34,42] 中也证实了设计债是 SATD 中最主要的一种类型 (约占 57%–68%). 2016 年, Bavota 等人^[9]对 159 个开源项目的 SATD 进行调研, 他们发现平均每个项目包含有约 51 个 SATD 实例, 在所有注释中占比约 0.3%. 其中最多的类型是代码债 (约 30%), 之后是缺陷债和需求债 (各占约 20%). 2020 年, Liu 等人^[49]研究了深度学习框架中 (例如) 的 SATD 的分布情况. 他们的调查结果显示: 1) 在所有研究的深度学习框架中都存在大量的 SATD; 2) 其中设计债的比重最多 (24.07%–65%). 综合不同的研究结果, 设计债和代码债是比重较大的种类而架构债和构建债所占的比重较低.

如第 3.1 节中所说, 研究者在对种类进行划分时存在差异性, 因此表 3 中总结的 SATD 的分布情况很可能存在偏差. 例如 Li 等人^[48]将算法债视为代码债的子类别, 所以他们报告的占比 39% 的代码债实际上可能会包含部分算法债从而导致代码债的比例偏高. 同时需要注意的是, 不同研究者所选择的项目种类和项目数量也会对 SATD 的分布结果产生影响. 例如, Potdar 等人^[4]的结果来自 4 个开源项目, Bavota 等人^[9]的结果来自 159 个开源项目, Liu 等人^[49]的结果来自深度学习框架类型的项目.

3.3 SATD 的根因 (root cause)

SATD 由开发者主动引入项目中, 在项目中演化一段时间后可能会被开发者偿还而从项目中移除。图 7 展示了 SATD 的引入和移除的示意图。对 SATD 的引入进行研究可以帮助开发者理解它们被引入的根本原因^[4], 这对于后续的移除具有一定的指导意义。例如, 在追溯到 SATD 的引入者之后, 软件维护时分配相同的开发者对其进行移除将会提高移除效率和准确性。

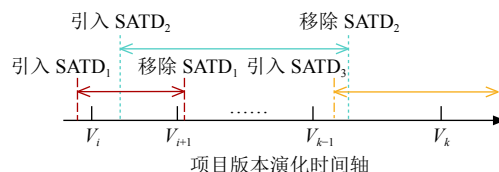


图 7 自承认技术债的引入和移除

目前, 对 SATD 根因的研究文献较少, 主要的发现来自 Potdar 等人^[4]。他们调查了开发者经验、时间和复杂度 3 个维度的可能引入 SATD 的因素。他们在 4 个 Java 开源项目 (Eclipse, Chromium OS, ArgoUML 和 Apache httpd) 中超过 800 个 SATD 实例上进行调研。结果显示: 1) 经验丰富的开发者更容易在项目引入 SATD, 并且他们在整个开发周期内的不同时段都会引入 SATD。Zampetti 等人^[30]在他们的研究中分析原因可能是有经验的开发者更能够意识到代码中存在的缺陷或者不足从而会更加主动地书写代码注释来记录这些 SATD。2) 版本发布时间的压力并不是引入 SATD 的主要原因, 仅有小部分 (<15%) 的 SATD 是在版本发布前一个月内被引入的。3) 一个文件中的 SATD 数目与代码复杂度之间的关联很弱 (通过 Spearman 相关系数检测)。由于相关研究的缺乏, 这些发现可能不具有代表性。所以在未来的研究中需要对 SATD 的根因进行更多细致的研究。相关研究推荐针对不同 SATD 的特点提出一些预防性的措施来避免 SATD 的引入, 从源头上解决部分 SATD 带来的问题。

3.4 SATD 的移除 (removal)

通常来说, SATD 在被引入项目后是需要被移除的^[39]。研究 SATD 的移除情况主要是从移除比例^[4,9,50]、移除人员^[50]、移除方法^[53]、移除的有效性^[51]等角度展开。通过这些研究, 开发者可以明白如何有效地并且精准地偿还特定的 SATD 来提升软件质量。例如, 挖掘潜在的健康技术债模式 (很久都没有被移除的技术债), 这些债务可能不需要偿还^[50]。再如, 为特定种类的 SATD 推荐合适移除方法帮助开发者决策如何偿还这些债务。

在 Potdar 等人^[4]调研的 4 个项目中, 他们发现部分 SATD 会在软件中存在很长一段时间。例如, 大约有 25.45% 的 SATD 在项目 Apache httpd 上的生存时间甚至长达 10 年。大约有 26.3%–65.5% 的 SATD 会在紧接着的下一个版本中被移除。另外有大约 9%–28% 的 SATD 会在随后的若干个版本中被陆续移除。这说明有很多 SATD 会存在于项目中的多个版本上。

根据 2016 年 Bavota 等人^[9]在 159 个项目上的调研, 约有 57% 的 SATD 会在之后的提交中被移除, 然而在被移除前长时间存在。他们发现 SATD 在移除时已经在项目中存活了很长的时间 (平均超过 1000 次代码提交) 并且随着时间推移由于新引入 SATD 导致其总数目是逐渐增加的。这说明开发有效管理 SATD 的技术或工具十分有必要。在这些被移除的 SATD 中, 大部分 (约 63%) 是被自我移除的。即, 它是被引入该技术债的开发者自己主动移除的。少部分 (约 37%) 是由更有经验的开发者移除的。

2017 年, Maldonado 等人^[50]对技术债的移除进行了深入调查。他们有如下发现: 1) 大多数 SATD 会在之后的代码变更中被移除, 约占总 SATD 数目的 74.4%; 2) 在被移除的 SATD 中, 多数 (超过 50%) 是通过自我移除来完成的, 这与 Bavota 等人^[9]的发现较为一致; 3) 相比于非自承认技术债, SATD 能够更快地被移除; 4) SATD 在被移除前在不同项目上的生存周期有较大差异。平均来说, SATD 可以在每个项目上生存 82–613 天; 5) 开发者经常使用 SATD 来跟踪未来的 bug 和需要改进的代码块。开发者通常在修复缺陷或者是添加新的特征时移除之前引入的 SATD, 很少有开发者专门为了重构或者改进代码来移除 SATD。

2018 年, Zampetti 等人^[43]对 SATD 注释的移除的真实性进行了定性和定量的分析。他们发现 1) 约有 20%–50%

的 SATD 注释是开发者在删除整个类或方法时被“意外”移除的。2) 约有 8% 的 SATD 的移除操作在代码提交日志中得到显式确认。例如, 指出该 SATD 被移除或者解释该 SATD 不对代码构成威胁因此不需要处理。3) 开发者移除 SATD 时需要源代码进行复杂地更改。同时也会通过对修改方法调用或控制逻辑实施特定更改来移除 SATD。4) 约有 55% 的 SATD 是由于代码调用的外部 API 发生特征变化或特征添加而被移除。2020 年, Zampetti 等人^[53]构建了一个多级别分类器 SARDELE 可以自动为 SATD 推荐合适的移除策略。特别地, 给定系统源代码中出现的 SATD, 它将从备选的 6 种移除策略推荐出合适的供开发者参考, 包括简单变更 API 调用、条件、方法基调、异常处理、返回语句或者指出需要更复杂的变更。SARDELE 组合了一个训练自 SATD 注释的词嵌入卷积神经网络和一个训练自受 SATD 影响的源代码的词嵌入循环神经网络。在包含 799 个实例的数据集上的测试结果来看, 该方法的 AUC 评估值可以达到 0.73。

2019 年, Maipradit 等人^[51]对 SATD 的移除进行了定性的分析。他们验证了 Zampetti 等人^[43]的发现, 大约 58% 的 SATD 并非被解决而仅仅是简单地被删除。在被移除的 SATD 实例中, 主要是通过实现新代码 (58%) 和代码重构 (15%) 被移除的。

同年, Iammarino 等人^[52]研究了 SATD 的移除与代码重构之间的关系。结果表明: 1) 代码重构操作与 SATD 移除事件在一起发生的概率很大。2) 在多数项目上只有一小部分与 SATD 移除同时发生的操作涉及到了 SATD 相关的代码。3) 在移除 SATD 时所执行的代码重构策略因项目而异, 要么涉及在类间移动操作 (operations) 和属性 (attributes), 要么从现存类中提取操作。

3.5 SATD 的影响 (impact)

研究 SATD 的影响可以对 SATD 的危害特性 (如正向利息^[33]、返工工作量^[34]等) 进行量化, 这有助于引起研究者着重研究如何识别和移除高危害的 SATD。同时, 可以帮助开发者尽量避免引入对危害程度较高的 SATD。表 4 总结了现有文献 [4,5,33,34,54] 对 SATD 的影响进行研究的主要发现。

表 4 对 SATD 影响的主要发现

年份	文献	SATD 的影响
2014	[4]	SATD 不会对代码的复杂性产生影响
2016	[5]	SATD 会对软件产生负面影响, 但是其影响并不是与缺陷有很强关系, 而是会使系统变得复杂从而导致在将来很难对其进行变更
	[33]	SATD 会给软件系统的维护带来正向利息 (positive interest, 即偿还债务的额外困难)
	[34]	SATD 会增加软件系统维护时的返工工作量 (rework effort, 即为偿还债务需要修复的代码行)
2019	[54]	SATD 注释无法对消除系统架构分歧产生显著影响

2014 年, Potdar 等人^[4]的研究显示代码复杂度 (通过圈复杂度、扇入值和扇出值来度量) 与 SATD 数目之间无明显关系, 因此 SATD 不会对影响代码的复杂性。2015 年, Bavota 等人^[9]通过计算斯皮尔曼 (Spearman) 相关系数也发现, 代码文件的内部质量度量, 包括耦合度 (CBO)、复杂度 (WMC) 和可读性, 与它们包含的 SATD 实例的数量之间没有相关性。

2016 年, Wehaibi 等人^[5]调查了 SATD 与软件质量之间的关系。他们发现 1) SATD 和软件缺陷之间没有明确的关系。具体表现为, 在研究的一部分项目中, 包含 SATD 的文件对应有更多的 bug 修复变更。然而在另一部分项目中, 不包含 SATD 的文件反而存在更多的缺陷。2) 包含 SATD 的变更比不包含 SATD 的变更在未来引入的缺陷数要少。3) 对包含 SATD 文件进行修改的变更更加难以执行, 换句话说, 很难在后期修改包含 SATD 的文件。他们得出的结论是虽然 SATD 可能会对软件产生负面影响, 但是其影响并不是与缺陷有很强关系, 而是会使系统变得复杂从而导致在将来很难对其进行变更。

Kamei 等人^[33]首次对 SATD 的利息 (即偿还债务的额外困难) 进行调查。他们使用代码的两个度量指标 (即代码行和扇入数) 来衡量 SATD 的利息。他们的基本假设是: 导致正向利息 (即代码更为复杂) 的 SATD 在将来进行移除时的难度更大。他们的样例研究显示, 在使用代码行作为度量指标时大约有 44.2% 的 SATD 会导致正向利息,

在使用扇入数作为度量指标时约有 42.2% 的 SATD 会招致正向利息。

Mensah 等人^[34]指出考虑 SATD 的返工工作量 (rework effort) 对于使发布的软件产品更加健壮和长期有效十分重要。具体来说, 返工工作量是指为偿还 SATD 需要花费的努力。根据 Zhao 等人^[70]提供的工作量指标, 为了修复 SATD, 平均每个包含 SATD 的源文件需要花费的修改工作量达到了 13–32 行代码。2018 年, Mensah 等人^[42]介绍了一个包含 6 个步骤的优先化方案以帮助提取不同种类的 SATD。该方案还可以帮助管理者在软件发布之前做出决策以尽量减少可能导致更高维护开销的 SATD 相关任务。他们发现大约 31%–39% 是主要的 SATD 任务。这些主要任务大多很复杂, 开发人员很难在软件发布之前处理这些任务。

2019 年, Sierra 等人^[54]研究了 SATD 与架构分歧 (architectural divergences) 之间的关系。其中包括两个研究点: 1) SATD 注释是否能够被用来追溯软件系统中的架构分歧; 2) 研究解决这些 SATD 注释是否可以消除对应的系统架构分歧。从调研结果来看, SATD 注释仅仅能够追溯到系统中 14% (4/29) 的分歧。虽然解决 SATD 注释中的问题可以直接对分歧进行修复, 但是 SATD 可以追溯到并进行修复的数量的数量不够具有代表性。此外, 这项任务是时间密集型的并且不会显著改进系统架构。

3.6 实证研究的数据集

上述对 SATD 进行实证研究结果是在特定的数据集上得出来的。表 5 列出了相关文献使用的调研数据集汇总信息。

表 5 进行 SATD 实证研究使用的数据集汇总

文献	项目个数	标签收集方式	调研内容					复用文献列表
			种类	分布	根因	移除	影响	
[4]	4	手工标注	√	√	√	√	—	[34,42]
[1]	5	手工标注	√	√	—	—	—	—
[9]	159	Pattern识别 ^[4]	√	√	—	√	—	—
[5]	5	Pattern识别 ^[4]	—	—	—	—	√	—
[33]	1	手工标注	—	—	—	—	√	—
[67]	4	Pattern识别 ^[4]	—	—	—	—	√	—
[50]	5	NLP识别 ^[2]	—	—	√	√	—	[51,52]
[43]	5	NLP识别 ^[2]	—	—	—	√	—	[52,53]
[54]	1	Pattern识别 ^[4]	—	—	—	—	√	—
[7]	5	手动标注	√	√	—	—	—	—
[48]	2	手动标注	√	√	—	—	—	—
[49]	7	手动标注	√	√	—	—	—	—

(1) 多样性。在不同的实证研究中, 研究者使用 SATD 数据集来调研不同的特性, 包括调研 SATD 的分布及其分类^[1,4,7,9,48,49]、引入^[4,50]、移除^[4,9,43,50]和影响^[5,33,54,67]。

(2) 差异性。对同一类特性进行调研时, 研究者通常会从不同的项目中重新收集数据。例如, 有六篇文献的研究者使用了不同的数据集来研究 SATD 种类。在不同数据集上进行调研可以使得结论更加全面。与此同时, 不同研究者在收集数据时会带来个人偏好。因此对同一特性的调研在不同的文献中表现会有差异^[1,9]。

(3) 唯一性。多数研究的数据集仅仅出现在一篇研究文献中, 仅有部分数据集 (如 Potdar 等人^[4]) 被后来的研究者 (如 Mensah 等人^[34,42]) 使用。这说明多数的研究具有唯一性, 其中的结论有待被进一步地被验证。

(4) 不可靠性。某些调研^[5,9,54,67]没有采用手工标注而是使用模型识别的方式来获取数据集的标签 (SATD 或者 non-SATD)。根据对这些识别模型的有效性验证^[6,41], 它们的分类结果在很大程度上不能够满足研究者的需要 (准确度不够高)。因为直接使用这类工具的分类结果作为类别标签可能会得出一些不可靠的调研结论。

3.7 小 结

本节介绍了近年来对 SATD 特性方面的研究, 主要从 SATD 种类、分布、根因、移除和 SATD 对软件质量

的影响 5 个方面进行介绍. 下面总结主要的发现.

- (1) 种类. 现有研究根据注释的不同语义描述划分出 10 种类型的 SATD, 分别为代码债、缺陷债、设计债、文档债、需求债、测试债、兼容债、算法债、架构债和构建债.
- (2) 分布. 在不同类型的债务中占比较多的是设计债、代码债和需求债这 3 种类型^[1,9,49], 因此它们需要在之后的研究中获取更多的关注.
- (3) 根因. 很多 SATD 是由有经验的开发者引入项目中的, 并且会在项目中长时间存在^[4,9].
- (4) 移除. 项目中有部分 (如 57%^[9]) SATD 会在之后被移除. 开发者主要会通过代码重构来移除这些 SATD. 但同时有些 SATD 不是真正的移除, 而仅仅是被删掉^[51].
- (5) 影响. SATD 会增加软件的后期维护成本, 包括增大变更难度^[5]、提高偿还利息^[33], 带来返工工作量^[34]等.
- (6) 研究者所用的数据集大多不相同, 在此基础上进行实证研究可以提高调研结果的泛化性, 但同时也会引入一些矛盾的结论^[1,9].

4 对 SATD 的识别

对 SATD 进行识别的研究是近年来 (特别是 2017 年之后) 的研究重点. 从本质上讲, 识别注释中的 SATD 是对文本进行二分类的过程^[41]. 本节首先从不同的技术角度来总结相关的 SATD 识别模型的构建. 接着, 从评估指标和数据集两个方面介绍对 SATD 识别模型的评估.

4.1 SATD 识别模型的构建

有监督模型是一类重要的 SATD 识别模型, Maldonado 等人^[2]最早开始研究这类识别模型. 这类模型会从已有的 SATD 注释中提取文本或者语义特征来训练预测模型. 目前该领域的大多数模型属于有监督模型. 表 6 列出了自承认技术债识别的代表性有监督模型汇总信息. 以下是这类模型主要的研究工作介绍.

表 6 识别 SATD 有监督模型汇总

年份	文献	重要的模型组件	识别债务类型	分类器类别
2017	[2]	NLP处理; Stanford分类器	设计债需求债	二分类
	[30]	方法级度量和静态分析工具	设计债	二分类
2018	[6]	文本挖掘; 特征选择IG; 投票机制	不区分类型	二分类
	[64]	N-gram IDF; 自动机器学习	设计债需求债	二分类
	[69]	词嵌入; 特征选择	不区分类型	二分类
2019	[41]	卷积神经网络	不区分类型	二分类
	[70]	词嵌入; 特征选择	不区分类型	二分类
	[59]	N-gram IDF; 下采样; 随机森林分类器	设计债需求债	三分类
	[51]	On hold类型的SATD; NLP分类器	不区分类型	二分类
2020	[66]	LSTM神经网络模型; 词嵌入	设计债需求债	二分类
	[61]	分治策略; 人工辅助	不区分类型	二分类

4.1.1 预处理操作

作为训练和测试数据的代码注释是用自然语言书写的. 自然语言在进行书写时用法十分灵活并且形式各异, 这种差异性会给模型构建带来挑战. 为了缓解这一问题, 很多研究^[2,6,44,59,64]都要对原始的注释文本进行预处理来获取较为整洁的数据. 其具体处理步骤如下.

- (1) 符号化 (tokenization). 这一步去除了注释中包含的无用符号, 包括标点符号 (如“:”) 和其他非字母组成的元素 (如“/”), 然后将注释文本切分为独立的单词, 同时将所有单词转换为小写形式.
- (2) 移除停用词 (stop-word removal). 停用词是指一些没有意义或者区分度的单词 (例如“I”“the”“to”等), 在这一步中这些停用词将会从备选单词中移除以降低其对预测模型的影响. 要注意, 在不同的模型中采用的停用词表

不完全相同, 这取决于具体的模型实现.

(3) 提取词干 (stemming). 最后一步将所有单词的变体转换为其原始形式 (例如将“processing”转换为“process”), 因为不同变体所具有的词意基本是一致的.

图 8 展示了一个对代码注释的进行预处理的样例. 使用上述步骤在对注释进行预处理后, 每条注释便对应一组 (有序的) 单词列表. 研究者可以在此单词列表的基础上建模分析 SATD 的特征并且构建出 SATD 分类模型.

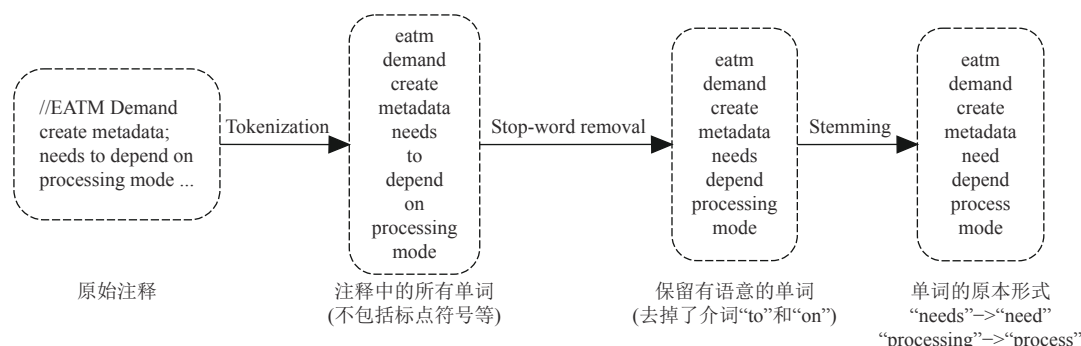


图 8 对代码注释的预处理的样例

4.1.2 基于自然语言处理技术的模型

2017 年, Maldonado 等人^[2]首次提出了 NLP 模型来进行 SATD 的识别. NLP 模型用到了最大熵分类器 Stanford Classifier, 该分类器会自动地从训练集中生成特征并且计算相应权重. 同时, 它会考虑特征之间的依赖性来训练各类别的最大化条件似然概率. 根据每种类别的概率值高低来判断最终的预测类别. 要注意的是, NLP 模型可以被用来预测两种不同类型的 SATD (design SATD 和 requirement SATD).

2018 年, Flisar 等人^[69]利用 Word2Vec 实现从代码注释中构建词嵌入 (word embedding) 模型为软件工程领域创建一个分布式表示的词向量空间模型. 接着利用该模型将单词转换为低维的实值向量并用于 SATD 的分类任务. 为了提高分类准确性, 他们对现有的传统特征选择方法进行了改进, 利用词嵌入模型中的相似度来选择最佳的分类特征. 该模型提升了 SATD 注释的分类结果. 2019 年, Zhao 等人^[70]在此基础上提出一个强化的方法. 不同之处在于他们使用大量未标记的源代码注释来训练一个单词嵌入模型, 然后对词袋模型进行强化以获得更充分、信息更全面的单词集合用于 SATD 的分类.

由于设计债和需求债在所有类型债务中的占比较多^[1], Wattanakriengkrai 等人^[59,64]关注于对这两种类型技术债的识别研究. 他们首先在 2018 年提出 N-gram IDF 和自动学习算法 auto-sklearn 组合的方法来构建分类模型^[64]. 具体来说, 该模型使用工具 Ngweight^[71]来计算每个注释中的 N-gram IDF 的权重, 使用 Matthias 等人^[72]提出的能够自动地寻找最优分类器的学习器来建模预测未知注释的标记. 虽然该模型可以对设计债和需求债进行分类, 但是需要为每种类型的技术债分别训练一个模型, 这增加了模型的维护成本 (例如参数调整). 为了解决这一问题, 他们随后在文献 [59] 中将之前的模型改进为一个使用随机森林作为学习器的三分类模型, 即输入该模型的注释会对应 3 种输出类别 (即 design SATD, requirement SATD and non-SATD). 该模型的优势在于只需训练一个模型便可以同时对设计债和需求债进行分类, 降低了模型应用和维护成本.

2019 年, Maipradit 等人^[51]定义了一种特殊的“on-hold”类型的 SATD. 这类技术债中包含一个条件来提示开发人员需要等待其他地方实现某个事件或某个更新的功能. 如样例“// TODO the following code is copied from AbstractSimpleBeanDefinitionParser // it can be removed if ever the doParse() method is not final!”. 在他们的研究样例中约有 10.4% 的实例属于“on-hold”类型的 SATD. 他们设计了一个基于自然语言处理的分类器来自动地识别这类技术债, 并且检测出开发者正在等待的特定事件. 该模型可以在等待条件满足时可以自动地向开发者推荐来移除这些 SATD.

4.1.3 基于传统机器学习方法的模型

2017 年, Zampetti 等人^[30]提出 TEDIIOUS 模型, 该模型向开发者推荐代码中可能存在技术债的方法, 并建议开发者在这些方法中主动添加相关的技术债信息 (即 SATD). TEDIIOUS 使用方法级别的度量信息 (包括结构度量和可读性度量^[73]) 和静态分析工具 (如 PMD) 所汇报的警告信息来刻画一个方法的特征, 并使用随机森林分类器来训练预测模型. TEDIIOUS 进行推荐的意义在于: 1) 可以鼓励该开发者去主动自我承认这些技术债的存在 (根据 Potdar 等人^[4]的调查, 开发者经验越丰富, 提交的 SATD 数目越多); 2) 这些推荐可以作为代码坏味道 (code smell) 检测器的替代品为改善源代码质量提供提示信息.

2018 年, Huang 等人^[6]提出了 TM 模型, 一个应用词袋模型 (bag of word, BOW^[74]) 来表示注释并使用向量空间模型 (即 VSM^[75]) 来提取注释特征的文本挖掘技术来识别 SATD. 在该模型中, 首先对注释进行预处理 (如图 8). 接着, 在每个训练项目上, 利用特征选择找出对于注释分类有效的特征 (即单词) 并且训练一个子分类器. 最后, 对每个子分类器的分类结果进行投票来确定目标项目上的注释最终的类别. 例如, 对一个目标注释, 有 5 个子分类器预测结果为 SATD, 4 个子分类器预测结果为 non-SATD, 那么该注释的最后类别将被预测为 SATD. 随后, 该 SATD 识别工具以被集成成为 Eclipse 插件^[58].

2020 年, Yu 等人^[61]根据文献^[44]的研究发现可以根据识别难度将 SATD 注释划分为两种类型: 1) “Easy”, 这类 SATD 很容易通过匹配来识别出来; 2) “Hard”, 这种 SATD 注释则需要领域专家进行人工阅读来判断其是否包含 SATD. 根据这种分类方法, Yu 等人提出了一个使用分治策略的分类模型 Jitterbug. 首先, 从训练集中提取出最常见的 SATD 模式 (即关键词) 并在测试集上对这些模式进行匹配来识别出“Easy”部分的 SATD. 然后, 去除训练集中已识别为 SATD 的注释并使用它们构建一个排序评估模型. 该模型可以评估目标项目中剩余的 SATD 的数目并且每次向开发者推荐排名靠前的 10 条注释来审查直到 SATD 的召回率达到预估的 90% 时为止. 从最终的分

4.1.4 基于深度学习的模型

2019 年, Santos 等人^[66,76]首先使用词嵌入 (word embedding) 模型来预处理注释文本. 接着应用长短期记忆网络 (long short-term network, LSTM) 模型构建分类器来识别设计债和需求债. LSTM 是一种时间递归神经网络, 它解决长序列训练过程中的梯度消失和梯度爆炸问题, 可以比普通循环神经网络 (recurrent neural network, RNN) 在更长的序列中有更好的表现. 从实验结果来看, 使用 LSTM 和词嵌入的分类方法的性能优于基准方法^[2,15].

同年, Ren 等人^[41]将深度学习中的卷积神经网络 (convolution neural network, CNN) 结构应用在注释文本特征的提取上, 提出了 CNN 模型. 作为一种深度学习模型, 它可以抽取更加丰富的文本特征. 相比已有有监督模型^[2,6], CNN 模型取得了最优的分类结果. 然而, 该模型构建参数众多, 需要进行仔细地调节参数来达到最好的效果从而增加了其应用难度. 此外, 该模型的训练时间也明显高于其他有监督模型.

4.1.5 基于模式匹配的无监督模型

相比于有监督模型, 无监督模型的构建和使用十分方便. 这类方法的最新研究展示出与有监督模型相比肩的性能. 无监督的模型主要是通过总结不同的技术债模式 (pattern) 来识别 SATD 的存在^[1,4,18,44,62,63]. 表 7 列出了自承认技术债识别的代表性无监督模型汇总信息. 以下是这类模型主要的研究工作介绍.

表 7 识别 SATD 的无监督模型汇总

年份	文献	说明	模式样例
2014	[4]	人工总结模式	todo, this is wrong
2015	[18]	上下文语境词汇表, 组合不同词类和Tag	Tag + Action Verb: TODO: test that surrogate value is legal.
2018	[62]	人工总结模式	TODO: not yet supported TODO: not documented
2019	[44]	任务标签模糊匹配	todo, fixme, hack, xxx
2020	[47]	改良的上下文语境词汇表, 考虑到词汇的重要性以及与不同种类SATD的联系	Todo: temporary Note: This is temporary must be resized need to be moved

2014 年, Potdar 等人^[4]首先提出使用模式匹配来识别 SATD. 他们通过人工阅读 101 762 条代码注释, 总结出 62 个不同的模式. 每个模式由一个标明 SATD 的关键词 (如 `todo`) 或者短语 (如 `this is wrong`) 组成. Pattern 方法通过在目标注释中匹配这些模式来识别其中是否存在 SATD. 可以看出, 只要包含这些模式的注释有很大的可能性的确存在 SATD, 但是从实验结果来看^[6], Pattern 模型的召回率很低, 这是因为这些模式太过具体化, 不能够很好地兼容随意书写的自然语言. 例如, “`this is wrong`”不能匹配到“`it is wrong`”即使这两句注释表达了相近的语义.

2015 年, Farias 等人^[18]提出一个使用上下文结构的词汇模型 CVM-TD. 该模型扩展了 Potdar 提出的 Pattern 模型 (即, 固定关键词匹配), 它考虑到不同单词词性 (名词、动词、形容词、副词) 和代码标记 (tag) 的组合来提供技术债词汇表并以此用来识别不同类型的技术债. 随后 Farias 等人^[65]通过一组对照实验来研究 CVM-TD 的整体准确性, 以及英语技能和开发人员经验因素的影响. 他们发现 CVM-TD 提出的词汇表有助于参与者理解和识别可能指出 TD 项目的评论. 他们还确定了参与者选择最多的 20 种对识别技术债起决定性作用的模式 (例如 `TODO`, `Need to` 等). 2020 年, Farias 等人^[47]对 CVM-TD 模型作出了以下改良. 1) 对词汇表中的模式进行排序来挑选对 SATD 最具判别力的模式; 2) 建立每种类型的 SATD 与模式之间的联系. 改良后模型中的词汇表能够找到与代码、设计、缺陷、文档和需求债务相关的 SATD.

2018 年, Passos 等人^[62]提供了两组新的不同模式来分别提高文档债和需求债的识别. 例如, 包含模式“`TODO: needs documenting`”、“`TODO: not documented`”和“`please document`”等的注释通常包含文档债; 而包含模式“`not implemented`”、“`TODO: not complete`”和“`not yet supported`”等的注释通常包含着需求债.

2019 年, Guo 等人^[44]对软件集成开发环境中的任务标签 (task tags) 在注释中的分布进行调研. 他们发现任务标签与 SATD 之间存在很强的相关性, 即绝大多数 SATD 中都存在任务标签 (主要的标签是“`todo`”“`fixme`”“`hack`”和“`xxx`”). 因此, 他们提出了使用任务标签模糊匹配的方法 (MAT) 来识别注释中 SATD. 与已有的有监督方法 (如 `TM`^[6]、`CNN`^[41]等) 的对比结果显示, MAT 的分类性能超过了 `TM` 并且十分接近 `CNN` 的分类性能. 这在一定程度上说明已有的有监督模型使简单的 SATD 分类问题复杂化.

4.2 SATD 识别模型的评估

本小节从评估指标、实验数据集和性能比较 3 个部分来介绍学术界对 SATD 识别模型的评估方法和结果.

4.2.1 评估指标

为了评估一个 SATD 识别模型的有效性, 研究者^[2,6,41,44,59,61,66]采用常用的分类指标对模型的预测结果进行评估. 一般来说, 一个识别模型的输入是一条待处理的注释, 输出是该条注释的预测标签 (即 SATD 或者 non-SATD).

从二分类的角度来看, 对注释进行识别的分类结果总共可以划分为以下 4 种可能情况.

- 对于一个被识别为包含 SATD 的注释, 它的确包含 SATD, 这种结果属于 TP (true positive);
- 对于一个被识别为包含 SATD 的注释, 它并不包含 SATD, 这种结果属于 FP (false positive);
- 对于一个被识别为没有 SATD 的注释, 它的确包含 SATD, 这种结果属于 FN (false negative);
- 对于一个被识别为没有 SATD 的注释, 它并不包含 SATD, 这种结果属于 TN (true negative).

现有的二分类评价指标首先基于上述 4 种情况的混淆矩阵 (见表 8) 对一个项目中的所有实例进行统计. 然后在此基础上计算各指标的值. 具体含义如下.

表 8 混淆矩阵

	预测为 SATD	预测为 non-SATD
真实为 SATD	TP	FN
真实为 non-SATD	FP	TN

(1) Precision (精度). 在预测为包含 SATD 的注释实例中, 有多少比例的注释真正包含 SATD.

$$Precision = \frac{TP}{TP + FP}.$$

(2) Recall (召回率). 在所有真正包含 SATD 的注释实例中, 有多少比例的注释被模型识别出来.

$$Recall = \frac{TP}{TP + FN}.$$

(3) F1-score (调和平均值). 精度和召回率的调和平均值, 可以更加全面客观地评价分类结果.

$$F1 = \frac{2 \times Precision \times Recall}{Precision + Recall}.$$

(4) Accuracy (准确率) 在所有分类结果中, 正确分类的注释实例 (包括正确分为 SATD 的实例和正确分为 non-SATD 的实例) 占有所有注释实例的比重.

$$Accuracy = \frac{TP + TN}{TP + FN + FP + TN}.$$

(5) MCC (Matthews correlation coefficient, 马修斯相关系数). 表示的是真实类别和预测类别的二元分类之间的相关系数.

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}.$$

在上述指标中, Precision、Recall、F1-score、Accuracy 的取值范围是 [0, 1], MCC 的取值范围是 [-1, 1]. 根据其定义, 当这些指标的值越大 (接近 1) 时模型的分类效果越好. 其中使用最多的 Precision、Recall 和 F1-score.

4.2.2 实验数据集

为了准确评估 SATD 识别模型的效果, 需要收集可靠的数据集进行验证. 综合已有研究, 评估数据集的收集主要包含以下 3 个步骤 (如图 9 所示).

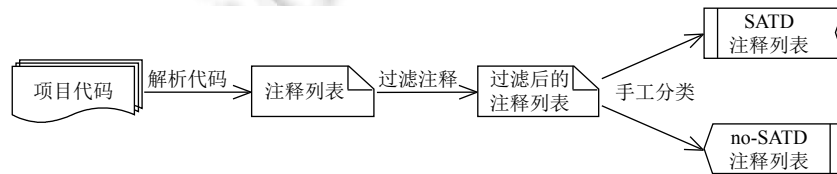


图 9 注释数据集收集过程

(1) 解析代码

这一步的目的是从项目中获取注释实例, 研究者通常使用一些自动工具来解析项目代码并抽取其中的注释部分. 例如, Maldonado 等人^[2]使用工具 JDeodorant^[56]而 Potdar 等人^[4]使用工具 srcML^[57]等.

(2) 过滤注释

直接从项目中提取的注释文本是由噪声的数据, 其中包含许多明显的特征表明它们不是 SATD 的注释. 使用这些数据进行训练和测试不能够准确地对模型进行评估^[1,2]. 因此, 需要从数据集中过滤这些干扰注释. Maldonado 等人^[2]总结了以下 5 种需要过滤的注释类型:

- 许可注释 (license comments). 这类注释通常被添加在类的声明之前并且描述了代码的许可信息. 通常这类注释中是不包含 SATD 的. 要注意的是, 如果这类注释中包含任务标签 (如“TODO”、“FIXME”或者“XXX”), 那么则不过滤这些注释, 因为这些标签通常伴随着 SATD 出现;

- 自动生成的注释 (automatically generated comments). 这类注释是由开发环境自动生成的注释, 它们没有描述任何有实际意义的内容因此不能够指明 SATD. 例如, “Auto-generated constructor stub”;

- Java 文档注释 (Javadoc comments). 这类注释的作用是解释 Java 代码中的实体的定义以及功能说明. 其中通常不会包含 SATD 除非其中包含任务标签;

- 被注释的代码段 (commented source code). 这类注释指的是代码中有些暂时不用的代码被注释掉的内容. 通常从直观上来说我们无法获得与 SATD 有关的文字描述, 因此需要过滤掉;

- 长注释 (long comments). 项目中某些长注释是由紧邻的多个单行注释组成的. 这些单行注释在形式上是分

离的,但是在内容上描述的是一个整体的事件. 对这类注释需要将这些紧邻的单行注释组合成一个完整的注释实例而不是过滤.

(3) 手工分类

这一步为过滤后的注释标记其类别 (SATD 注释或者 non-SATD). 标记过程是人工完成的. 为了降低人为因素带来的偏好,通常会由若干名研究者同时进行标注并且对有歧义的实例进行讨论后确定其类别或者先由一人进行标记并让另一人进行抽样标记之后计算 Cohen’s kappa 系数^[77],该系数大于 0.75 则表明两人标记的误差在可接受范围内^[40].

使用上述中的过程可以为模型验证提供比较可靠的数据集. Maldonado 等人^[2]花费了约 185 个小时标记了一个包含 10 个 Java 项目的数据集来评估他们 SATD 识别模型的效果. 该数据集的规模较大,总共包含 37 056 条注释并和 2812 条 SATD 注释. 表 9 列出了该数据集中所有项目的汇总信息,包括每个项目的名称、版本号、注释数目、过滤后的注释数目、SATD 的数目 (比例)、贡献者数目、类实体的数目以及代码行数. 由于在标记注释类别时需要耗费大量的人力成本,在之后的大多数研究中^[2,6,30,41,44,58,59,61,64,66,69,70,76]都使用了该数据集来验证他们的模型性能. 最近,根据 Yu 等人^[61]的调研,该数据集中存在一些错误标记. 因此,在该数据集上进行评估的 SATD 识别模型有待进一步验证其有效性.

表 9 用于评估 SATD 识别方法性能的数据集^[2]

Project	Release	#Comments	After filtering	#SATD	SATD (%)	Cont.	#classes	KLOC
Ant	1.7.0	21 587	3 052	102	0.47	74	1 475	115
ArgoUML	0.34	67 716	5 426	969	1.43	87	2 609	926
Columba	1.4	33 895	4 090	128	0.38	10	1 711	155
EMF	2.4.1	25 229	2 585	74	0.29	30	1 458	228
Hibernate	3.3.2	11 630	2 492	377	3.24	314	1 356	703
JEdit	4.2	16 991	4 644	195	1.15	57	800	310
JFreeChart	1.0.19	23 474	2 494	101	0.43	19	1 065	317
JMeter	2.1.0	20 084	4 148	282	1.40	41	1 181	354
JRuby	1.4.0	11 149	3 652	383	3.44	374	1 486	841
Squirrel	3.0.3	27 474	4 473	201	0.73	40	3 108	708
Total	—	259 229	37 056	2,812	1.08	1 046	16 249	4 657

4.2.3 性能比较

本小节汇总了现有的部分 SATD 识别模型的在上述数据集^[2]上表现的性能值的比较结果. 我们发现,并非所有的研究者报告了其模型在每个项目上详细的性能值,对于那些没有给出具体数值的模型 (例如^[58,69,70]) 将无法在这里汇总. 此外,由于多数研究只使用了 Precision、Recall 和 F1-score 作为模型的评价指标,Accuracy 和 MCC 仅在少数文献中^[30]被用到,所以本节只展示那些在前 3 个指标上报告了具体性能值的 SATD 识别模型的性能分布箱线图.

图 10 汇总了模型^[2,4,6,41,44,59,61]在识别所有种类的 SATD 时的性能比较. 可以看出,所有模型在精确度方面都表现较好 (图 10(a),性能值超过 0.5),其中基于模式匹配的无监督模型^[4,44,61]表现最好 (中位值均超过 0.85). 这说明当前的模型已经能够精准地识别出 SATD,特别是使用模式匹配的方式进行识别. 从召回率 (图 10(b)) 方面来看,现有模型在不同项目上的性能值差异较大 (性能区间约为 0.2–0.9),其中模型^[4]的召回率在每个项目上都低于 0.1. 召回率过低很难满足现实的需要,因此现有模型在召回率上仍有进一步提升的空间. 查看模型的 F1-score (图 10(c)),模型^[41,44]表现好于其他模型 (中位值接近 0.8),其中模型^[44]最新的无监督标签匹配方法,而模型^[41]是最新的基于卷积神经网络的有监督方法. 总的来说,对所有种类的 SATD 识别的模型在综合性能指标上面已有较大的进展. 因此,在未来的研究中,性能提升的空间似乎已经较小,研究者们应当在提升分类性能的同时更多地关注如何降低模型的复杂性和可解释性以帮助开发者更好地将其应用在实际的项目维护中.

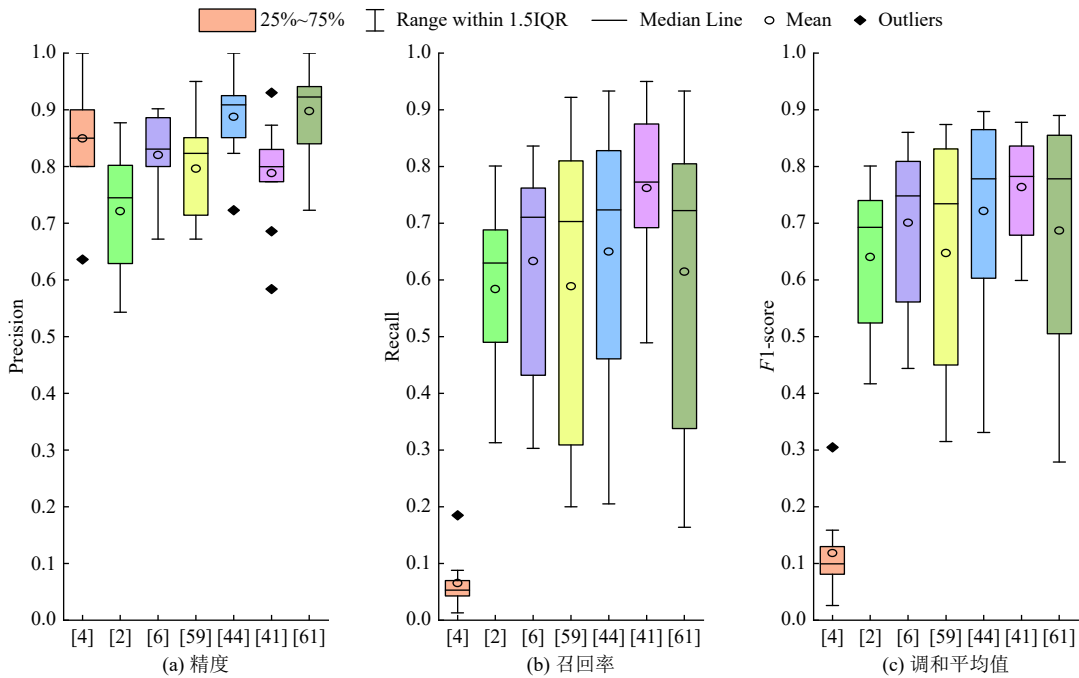


图 10 SATD 识别模型在识别所有债务时的性能比较

由于设计债和需求债在所有债务中所占比重较大并且对软件系统有着重要影响, 现有的若干研究设计模型针对这两类 SATD 进行识别. 图 11 汇总了模型^[2,64,66,76]在识别设计债和需求债种类的 SATD 时的性能比较. 从整体上来看, 现有模型识别设计债的效果好于识别需求债的效果. 值得关注的是, 这些模型基本上无法同时保证高的精确率和召回率. 例如, 模型^[66]实现了较高的精确率却只达到了很低的召回率, 而模型^[76]在精确率不理想的情况下却能够达到较高的召回率. 模型在前两个指标上的偏好性就导致了其在综合指标 F1-score 上的低效. 特别是对于模型^[66]来说, 它在识别设计债时的性能中位值在 0.4 左右, 在识别需求债时的性能中位值在 0.1 左右. 这种性能很难满足现实中的需求. 因此, 对于这种识别特定种类 SATD 的模型来说, 能够在各指标上保持均衡的性能是当前研究需要解决的挑战.

4.3 小 结

本节主要介绍了有监督模型和无监督模型两方面详细介绍了 SATD 识别模型近年的研究进展. 下面简述主要发现.

- (1) 有监督模型的比重较大, 这类模型会应用各种建模技术, 包括自然语言处理、传统机器学习和深度学习等. 这类模型的分类效果相对来说比较好, 存在的不足是这类模型比较复杂会对实用性产生负面影响.
- (2) 无监督模型主要的思想是通过模式 (即关键词) 匹配来识别注释中的 SATD. 这类模型比较简单直观, 实际操作更加方便, 不足之处是部分模型的分类效果很差并且可能需要消耗人力成本来总结模式.
- (3) 许多识别模型属于二分类模型, 它们不关心 SATD 所属于的种类 (如设计债或者需求债), 仅仅关注待测注释实例是否是 SATD, 这些模型在一定程度上简化了任务, 但不能对 SATD 的移除提供启示.
- (4) 目前对模型进行的验证的数据集比较单一. 大多数研究在 Maldonado 等人^[2]收集的数据集上进行实验, 这对已有模型的效果评估带来一定的威胁.

5 SATD 的其他研究工作

学术界对技术债的研究手段是多种多样的, 不仅仅可以使用代码注释对其进行研究和识别. 本节中简要介绍使用其他手段研究 SATD 的相关工作.

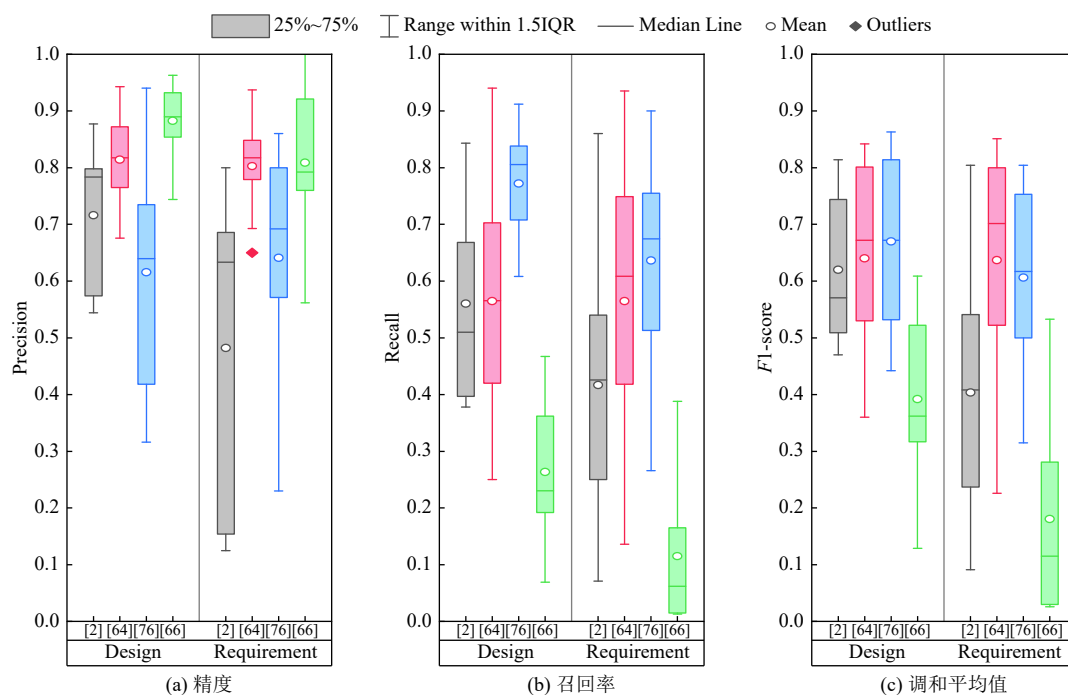


图 11 SATD 识别模型在识别设计债和需求债时的性能比较

5.1 软件变更中的 SATD

2019 年, Yan 等人^[60]指出所有的 SATD 都是通过开发者提交软件变更 (software change) 引入的. 在变更级别识别 SATD 可以帮助开发者理解该技术债的上下文信息从而更容易对其进行偿还. 因此, 他们设计了一个新的 SATD 识别模型能够预测新提交的代码变更中是否存在 SATD. 他们从扩散 (diffusion)、历史 (history) 和文本 (message) 3 个维度共提取 25 个自定义的变更度量来刻画一个变更的特征, 并使用随机森林分类器训练一个能够识别引入 SATD 变更的模型.

5.2 问题追踪系统中的 SATD

2020 年, Xavier 等人^[7]发现 SATD 不仅会出现在代码中 (称为 SATD-C), 开发者有时也会通过问题追踪系统 (issue tracking system) 中的问题 (issue) 来标明技术债 (称为 SATD-I). 他们着重对 SATD-I 进行了调研. 有如下发现: 1) SATD-I 和 SATD-C 在一定程度上是由重叠的, 大约 29% 的 SATD-I 实例可以追溯到相应的 SATD-C 实例. 2) 相比于其他类型的问题, 包含 SATD-I 的问题需要花费更多的时间才能被关闭. 3) 大部分 SATD-I 实例 (60%) 属于设计类型的债务. 4) 多数情况下 (45%) 的 SATD-I 实例是经过开发者仔细斟酌后有意引入项目中的. 5) 开发者会定期偿还 SATD-I 的主要目的是减少利息 (66%) 和清理代码 (28%).

同年, Li 等人^[48]分析了问题跟踪系统中 SATD 的类型分布并调研了软件工程师如何识别和解决这些技术债. 他们在 500 个 SATD 样例中有如下发现: 1) SATD-I 有 8 种类型, 分别为架构债 (architecture debt)、构建债 (build debt)、代码债、缺陷债、设计债、文档债、需求债和测试债. 2) 只有小部分 (15.9%) 的技术债是由其引入者自己识别的. 3) 大部分的 SATD-I 都已被偿还, 其中多数是由发现或引入它的人偿还的 (分别占 47.7% 和 44.0%).

6 现有挑战及研究方向

6.1 SATD 领域的现有挑战

从现有文献来看, 当前对 SATD 的研究还存在着若干需要解决的挑战. 本节分别介绍该领域的实证调研和识

别模型两类研究中所面临的挑战.

6.1.1 实证调研中存在的挑战

(1) 来自实证研究数据集的挑战. 数据集中的挑战主要来自两个方面: 1) 数据集标签的准确性存在问题. 有些实证研究^[5,9,43,50]使用的数据集并非人工标注, 而是使用一些识别模型(如 Maldonado 等人提出的 NLP 模型^[2])自动生成的. 然而这些识别模型本身的分类准确性并没有达到很高的水准, 因此可能会降低对 SATD 的调研结果的可靠性. 例如, Wehaibi 等人^[5]使用 Pattern 模型^[4]识别数据集中的 SATD 来进行实证研究, 他们发现 SATD 的数目很少, 这很可能是因为 Pattern 模型^[4]本身的召回率低导致的. 2) 缺少可复用的数据集. 多数实证研究使用的数据集都是研究者自己收集的并且其发布链接现在已不可访问. 由于没有明确统一的 SATD 种类定义, 不同研究者在收集数据的时候难免会因个人的偏好导致各个数据集之间存在偏差. 因此对 SATD 特性(分类、分布、引入、移除和影响)的调研结果在不同的文献中会得出不一致的结论. 这会使得后来的研究者因此产生困惑.

(2) 来自分析方法差异性的挑战. 不同研究的分析方法不完全一致, 其中的差异性会给不同研究结论一致性带来挑战. 例如, 在对 SATD 的类型进行分析时, Maldonado 等人^[1]认为存在 5 种类型的 SATD 并且得出的结论是设计债的占比最多. 而 Bavota 等人^[9]的研究中划分了 6 种 SATD 的类型, 比前者多一种代码债, 并且得出代码债是占比最多的 SATD 类型. 这之间的差异性是由不同的分析方法导致的. 虽然每种分析方法各有其合理性, 但是不同的结论会给后来的研究者带来一些干扰信息.

6.1.2 识别模型中存在的挑战

(1) 来自模型构建方法的挑战. 在构建识别模型时存在一些挑战会影响模型的性能. 1) 部分识别模型在构建时^[2,6]没有考虑到类别不平衡问题. 从现有的验证数据集^[2]来看, 注释类别之间是高度不平衡的. 其中包含 SATD 的注释仅占有所有注释的 1% 左右, 如果考虑到某个具体的 SATD 种类的比例则会更低. 已有的研究^[78,79]表明类别的不平衡的会在很大程度上影响模型的性能. 2) 一部分模型在构建时没有区分 SATD 的类型而是统一处理所有的 SATD 注释. 从现有研究来看^[1,9,34], 不同种类的 SATD 之间的注释表达方式和频繁出现的模式(即关键词)是不同的. 因此, 如果在构建识别模型时不考虑具体类型之间的差异性而是统一地将其视为 SATD 很可能给模型带来噪声从而限制了模型的性能.

(2) 来自模型有效性的挑战. 在目前所提出的各种模型中, 一方面, 有监督模型通常能够达到比较好的分类性能(其中 CNN 模型^[41]的 F1 值平均可以达到 0.76), 这在学术界的实验结果中已经表现得足够突出. 然而, 由于该模型的复杂度较高导致很难复现其性能上存在挑战, 因此还不能够广泛地应用在工业界的实际使用过程中. 另一方面, 基于模式匹配无监督模型相比于有监督模型来说有效性普遍较低^[6,41]. 主要原因在于人工总结的有限模式虽然能够准确识别出 SATD 但是无法完全匹配灵活的自然语言描述的注释文本. 因此这类模型通常具有高精度低召回率的特点.

(3) 来自模型实用性的挑战. 当前的有监督和无监督模型中都存在实用性的挑战, 这些挑战增加了开发者的使用成本因此实用性较低. 有监督模型的挑战来自于其复杂的模型设计, 其中包含了各种结构和参数的调节, 如 NLP 模型^[2]中有 14 个参数设置, CNN 模型^[41]需要调节 8 个超参数和 4 个训练参数. 虽然研究者给读者提供了推荐的参数设置, 但在实际使用中开发者根据不同项目的注释特点重新选择合适的模型配置才能发挥其效果, 这对于开发者是一种挑战. 无监督模型的挑战来自其需要花费人工成本进行总结模式. 虽然研究者提供了常用的模式, 但其可能无法匹配特定项目上的 SATD. 因此开发者需要手动重新进行总结合适的模式, 这无疑降低了模型的实用性.

(4) 来自模型验证数据集的挑战. 数据集的数量和可靠性存在问题, 目前研究者大多使用 Maldonado 等人^[2]提供的数据集来评估 SATD 识别模型的有效性. 他们通过手动的方式标注了 10 个跨领域的开源 Java 项目的注释类别, 因此该数据集十分具有代表性. 然而, 在最近 Yu 等人^[61]的研究中, 他们发现该数据集中约有 426 个注释实例的真实类别是被误标记的. 这将在一定程度上影响对识别模型有效性的评估.

6.2 SATD 领域的研究方向

针对 SATD 研究中存在的挑战以及现有研究中的结论, 我们从总结了以下几方面的研究方向可能需要得到研

究者的重点关注。

(1) 提供明确统一的 SATD 种类定义。正如 3.1 节中所说, 目前对 SATD 进行种类划分的依据是根据注释的内容进行人工分类, 具有较强的主观性和分类偏好。因此在今后的研究中, 推荐研究者能够提出明确无歧义的 SATD 种类划分定义。这将具有以下两点重要的意义: 1) 首先, 这将极大地有利于研究者根据 SATD 种类定义来编写规则脚本进行自动化地 SATD 种类划分而非耗时耗力的人工划分; 2) 其次, 这将有助于指导开发者在添加 SATD 注释时遵循明确的定义来规范注释的书写, 从而为 SATD 的后续管理提供便利。

(2) 按照种类更细化地研究 SATD。目前常见的 SATD 可以根据其含义划分为 10 个不同的种类 (如设计债、需求债等)。描述不同种类的 SATD 所用的语言和词汇有较大的差异性^[2]。例如 Passos 等人^[62]的研究中发现, 文档债中通常出现“not documented”而需求债通常伴随着“not supported”。现有的很多高效的识别模型^[6,41]在设计时不能够区分 SATD 的种类, 仅仅是判断一条注释实例是否为 SATD。因此我们推荐研究者按照种类细化地研究 SATD, 特别是识别模型的设计。这可以带来两方面的好处: 1) 能够为开发者识别出他们真正关心的 SATD 种类; 2) 专注于一个种类的识别可以避免其他种类 SATD 的数据给模型带来的噪声, 从而提高模型的分类准确性。

(3) 研究 SATD 移除方法的自动化推荐模型。对 SATD 进行研究的最终目标是要移除项目中存在的 SATD 来保障软件质量和健壮性^[4]。目前绝大多数自动化模型^[2,6,41,44]的作用是识别 SATD 的存在。能够自动为开发者推荐 SATD 移除方法的模型非常稀少。这使得开发者在识别到 SATD 之后还需要花费大量精力来决定该如何移除它们。最近, Zampetti 等人^[53]提出的推荐模型可以自动推荐移除策略, 该研究虽然为 SATD 的移除提供了一些提示性的信息, 但是其精确率 (0.55) 和召回率 (0.57) 较低。因此, 对这类模型的研究仍有较大的提升空间。

(4) SATD 管理工具的设计和开发。SATD 普遍长时间存在于项目中^[9], 然而目前没有有效的管理工具, 多数研究停留在实验阶段, 不能有应用在实际的生产中。Xavier 等人^[7]指出需要设计并实现新的工具来支持 SATD 的管理。管理工具可以用来支持 SATD 的识别、分类、重要性划分和移除方法推荐等。其中比较重要的是在每次软件发布之前都要扫描现存的 SATD 并且判断其是否需要进行偿还。

(5) 研究问题追踪系统中的 SATD。根据 Xavier 等人^[7]的调查, 问题追踪系统中也存在大量 SATD (即, SATD-I), 并且其中大部分 (约 71%) 的 SATD-I 是不能通过源代码注释来发现和解决的。目前仅有两篇文献^[7,48]涉及到这类 SATD 的研究。因为问题追踪系统是开发者进行问题发布和讨论交流的平台, 在该平台上面的发布的 SATD 可以在不同开发者的交流中被解决。我们建议研究者对这类 SATD 进行研究并鼓励开发者在问题追踪系统中创建和标记包含 TD 的问题 (issue)。

(6) 构建规范的 SATD 数据集。现有的数据集存在一些问题 (见第 6.1 节)。很多研究使用的数据现已无法在网上获取, 数据集的部分标签信息有误^[61]。为了能够获得准确的调研结果和准确地对识别模型作出评价, 我们需要对现有数据集进行验证并且构建统一规范的 SATD 数据集。数据集中最好能够包含丰富的信息 (例如注释的类别、引入者、是否被移除、移除者、存活时间、移除方法和与 SATD 关联的代码实体等) 可支持各项 SATD 研究。这对支持后续的研究工作十分重要。

(7) 支持其他软件质量保障相关的研究。SATD 是存在于软件代码中的有用实体, 其中包含部分与软件缺陷相关的信息 (如, 位置、优先级等)。研究者合理利用 SATD 提供的信息可以支持其他软件质量保障相关 (如缺陷定位^[80]、缺陷预测^[81]) 的研究。例如, SATD 在代码中的位置指明了可能存在缺陷的不同粒度的代码实体 (如源文件、类或方法)。在进行缺陷预测或者定位时提高这些包含 SATD 的代码实体的可疑度值可以优化预测或者定位结果。再如, 文档债指出的问题一般不会影响到代码的正确性, 包含这类债务的代码实体在进行缺陷可疑度排序时的优先级应当低于包含其他种类的 SATD 的代码实体。

7 结束语

当自承认技术债 (SATD) 的概念被提出后, 立即受到大量研究者的重点关注并且在该领域发表了数十篇研究成果。本文根据不同文献的侧重点围绕对 SATD 特性的调研和对 SATD 的识别两个方面对当前的研究工作进行了梳理和总结。主要工作总结如下: (1) 本文从 SATD 的种类、分布、根因、移除和影响 5 个维度对 SATD 特性

的调研进行归类 and 阐述, 并且对实证研究中的数据集信息进行了分析; (2) 本文首先根据不同的模型设计原理介绍当前的 SATD 识别模型; 然后从评价指标、实验数据集和性能比较 3 方面总结该领域的现状; (3) 本文从数据集、有效性和实用性等角度总结了当前对 SATD 的研究面临的挑战并针对性地从 7 个方面指出了未来需要重点关注的研究方向。

致 谢 感谢编辑以及两位匿名审稿专家提出的宝贵意见, 提高了这篇综述的质量!

References:

- [1] Maldonado EDS, Shihab E. Detecting and quantifying different types of self-admitted technical Debt. In: Proc. of the 7th Int'l Workshop on Managing Technical Debt. Bremen: IEEE, 2014. 9–15. [doi: [10.1109/MTD.2015.7332619](https://doi.org/10.1109/MTD.2015.7332619)]
- [2] da Silva Maldonado E, Shihab E, Tsantalis N. Using natural language processing to automatically detect self-admitted technical debt. IEEE Trans. on Software Engineering, 2017, 43(11): 1044–1062. [doi: [10.1109/TSE.2017.2654244](https://doi.org/10.1109/TSE.2017.2654244)]
- [3] Lim E, Taksande N, Seaman C. A balancing act: What software practitioners have to say about technical debt. IEEE Software, 2012, 29(6): 22–27. [doi: [10.1109/MS.2012.130](https://doi.org/10.1109/MS.2012.130)]
- [4] Potdar A, Shihab E. An exploratory study on self-admitted technical debt. In: Proc. of the Int'l Conf. on Software Maintenance and Evolution. Victoria: IEEE, 2014. 91–100. [doi: [10.1109/ICSME.2014.31](https://doi.org/10.1109/ICSME.2014.31)]
- [5] Wehaibi S, Shihab E, Guerrouj L. Examining the impact of self-admitted technical debt on software quality. In: Proc. of the Int'l Conf. on Software Analysis, Evolution, and Reengineering. Osaka: IEEE, 2016. 179–188. [doi: [10.1109/SANER.2016.72](https://doi.org/10.1109/SANER.2016.72)]
- [6] Huang Q, Shihab E, Xia X, Lo D, Li SP. Identifying self-admitted technical debt in open source projects using text mining. Empirical Software Engineering, 2018, 23(1): 418–451. [doi: [10.1007/s10664-017-9522-4](https://doi.org/10.1007/s10664-017-9522-4)]
- [7] Xavier L, Ferreira F, Brito R, Valente MT. Beyond the code: Mining self-admitted technical debt in issue tracker systems. In: Proc. of the 17th Int'l Conf. on Mining Software Repositories. Seoul: ACM, 2020. 137–146. [doi: [10.1145/3379597.3387459](https://doi.org/10.1145/3379597.3387459)]
- [8] Cunningham W. The WyCash portfolio management system. ACM SIGPLAN OOPS Messenger, 1992, 4(2): 29–30. [doi: [10.1145/157710.157715](https://doi.org/10.1145/157710.157715)]
- [9] Bavota G, Russo B. A large-scale empirical study on self-admitted technical debt. In: Proc. of the 13th Int'l Conf. on Mining Software Repositories. Austin Texas: ACM, 2016. 315–326. [doi: [10.1145/2901739.2901742](https://doi.org/10.1145/2901739.2901742)]
- [10] Kruchten P, Nord RL, Ozkaya I, Falessi D. Technical debt: Towards a crisper definition report on the 4th international workshop on managing technical debt. ACM SIGSOFT Software Engineering Notes, 2013, 38(5): 51–54. [doi: [10.1145/2507288.2507326](https://doi.org/10.1145/2507288.2507326)]
- [11] Seaman C, Nord RL, Kruchten P, Ozkaya I. Technical debt: Beyond definition to understanding report on the 6th international workshop on managing technical debt. ACM SIGSOFT Software Engineering Notes, 2015, 40(2): 32–34. [doi: [10.1145/2735399.2735419](https://doi.org/10.1145/2735399.2735419)]
- [12] Ozkaya I, Kruchten P, Nord RL, Brown N. Managing technical debt in software development: Report on the 2nd international workshop on managing technical debt, held at ICSE 2011. ACM SIGSOFT Software Engineering Notes, 2011, 36(5): 33–35. [doi: [10.1145/2020976.2020979](https://doi.org/10.1145/2020976.2020979)]
- [13] Kruchten P, Nord RL, Ozkaya I, Visser J. Technical debt in software development: From metaphor to theory report on the 3rd international workshop on managing technical debt. ACM SIGSOFT Software Engineering Notes, 2012, 37(5): 36–38. [doi: [10.1145/2347696.2347698](https://doi.org/10.1145/2347696.2347698)]
- [14] Falessi D, Kruchten P, Nord RL, Ozkaya I. Technical debt at the crossroads of research and practice: Report on the 5th international workshop on managing technical debt. ACM SIGSOFT Software Engineering Notes, 2014, 39(2): 31–33. [doi: [10.1145/2579281.2579311](https://doi.org/10.1145/2579281.2579311)]
- [15] Avgeriou P, Ernst NA, Nord RL, Kruchten P. Technical debt: Broadening perspectives report on the 7th workshop on managing technical debt (MTD 2015). ACM SIGSOFT Software Engineering Notes, 2016, 41(2): 38–41. [doi: [10.1145/2894784.2894800](https://doi.org/10.1145/2894784.2894800)]
- [16] Izurieta C, Ozkaya I, Seaman C, Snipes W. Technical debt: A research roadmap report on the 8th workshop on managing technical debt (MTD 2016). ACM SIGSOFT Software Engineering Notes, 2017, 42(1): 28–31. [doi: [10.1145/3041765.3041774](https://doi.org/10.1145/3041765.3041774)]
- [17] Fontana FA, Chatzigeorgiou A, Trumler W, Izurieta C, Avgeriou P, Nord RL. Technical debt in agile development: Report on the 9th workshop on managing technical debt (MTD 2017). ACM SIGSOFT Software Engineering Notes, 2017, 42(3): 18–21. [doi: [10.1145/3127360.3127372](https://doi.org/10.1145/3127360.3127372)]
- [18] de Freitas Farias MA, de Mendonça Neto MG, da Silva AB, Spínola RO. A contextualized vocabulary model for identifying technical debt on code comments. In: Proc. of the 7th Int'l Workshop on Managing Technical Debt. Bremen: IEEE, 2015. 25–32. [doi: [10.1109/MTD.2015.7332621](https://doi.org/10.1109/MTD.2015.7332621)]

- [19] Ichinose T, Uemura K, Tanaka D, Hata H, Iida H, Matsumoto K. ROCAT on KATARIBE: Code visualization for communities. In: Proc. of the 4th Int'l Conf. on Applied Computing and Information Technology/3rd Int'l Conf. on Computational Science/Intelligence and Applied Informatics/1st Int'l Conf. on Big Data, Cloud Computing, Data Science & Engineering. Las Vegas: IEEE, 2016. 158–163. [doi: [10.1109/ACIT-CSII-BCD.2016.040](https://doi.org/10.1109/ACIT-CSII-BCD.2016.040)]
- [20] Sierra G, Shihab E, Kamei Y. A survey of self-admitted technical debt. *Journal of Systems and Software*, 2019, 152: 70–82. [doi: [10.1016/j.jss.2019.02.056](https://doi.org/10.1016/j.jss.2019.02.056)]
- [21] Liu YJ, Li B, Li ZY, Liang P, Wu MQ. Study on technical debt management of integrated development environment. *Computer Science*, 2017, 44(11): 15–21, 40 (in Chinese with English abstract). [doi: [10.11896/j.issn.1002-137X.2017.11.003](https://doi.org/10.11896/j.issn.1002-137X.2017.11.003)]
- [22] Zhang YJ, Zhang X, Ding H, Wang X. Study on technical debt caused by requirement change. *Computer Science*, 2018, 45(9): 89–93 (in Chinese with English abstract). [doi: [10.11896/j.issn.1002-137X.2018.09.013](https://doi.org/10.11896/j.issn.1002-137X.2018.09.013)]
- [23] Yang J. Research on security assessment method based on technical debt quantification [MS. Thesis]. Beijing: Beijing Jiaotong University, 2016 (in Chinese with English abstract).
- [24] Ding H. Research on software requirement changes technical debt [MS. Thesis]. Kunming: Yunnan University, 2017 (in Chinese with English abstract).
- [25] Zhang YJ. Measuring technical debt of requirement change by marginal contribution [MS. Thesis]. Kunming: Yunnan University, 2019 (in Chinese with English abstract).
- [26] Li ZY, Avgeriou P, Liang P. A systematic mapping study on technical debt and its management. *Journal of Systems and Software*, 2015, 101: 193–220. [doi: [10.1016/j.jss.2014.12.027](https://doi.org/10.1016/j.jss.2014.12.027)]
- [27] Rios N, de Mendonça Neto MG, Spinola RO. A tertiary study on technical debt: Types, management strategies, research trends, and base information for practitioners. *Information and Software Technology*, 2018, 102: 117–145. [doi: [10.1016/j.infsof.2018.05.010](https://doi.org/10.1016/j.infsof.2018.05.010)]
- [28] Freire S, Rios N, Mendonça M, Falessi D, Seaman C, Izurieta C, Spinola RO. Actions and impediments for technical debt prevention: Results from a global family of industrial surveys. In: Proc. of the 35th Annual ACM Symp. on Applied Computing. Brno Czech Republic: ACM, 2020. 1548–1555. [doi: [10.1145/3341105.3373912](https://doi.org/10.1145/3341105.3373912)]
- [29] Zazworka N, Shaw MA, Shull F, Seaman C. Investigating the impact of design debt on software quality. In: Proc. of the 2nd Workshop on Managing Technical Debt. Honolulu: ACM, 2011. 17–23. [doi: [10.1145/1985362.1985366](https://doi.org/10.1145/1985362.1985366)]
- [30] Zampetti F, Noiseux C, Antoniol G, Khomh F, Penta MD. Recommending when design technical debt should be self-admitted. In: Proc. of the IEEE Int'l Conf. on Software Maintenance and Evolution. Shanghai: IEEE, 2017. 216–226. [doi: [10.1109/ICSME.2017.44](https://doi.org/10.1109/ICSME.2017.44)]
- [31] Alves NSR, Ribeiro LF, Caires V, Mendes TS, Spinola RO. Towards an ontology of terms on technical debt. In: Proc. of the 6th Int'l Workshop on Managing Technical Debt. Victoria: IEEE, 2014. 1–7. [doi: [10.1109/MTD.2014.9](https://doi.org/10.1109/MTD.2014.9)]
- [32] Izurieta C, Vetrò A, Zazworka N, Cai YF, Seaman CB, Shull F. Organizing the technical debt landscape. In: Proc. of the 3rd Int'l Workshop on Managing Technical Debt. Zurich: IEEE, 2012. 23–26. [doi: [10.1109/MTD.2012.6225995](https://doi.org/10.1109/MTD.2012.6225995)]
- [33] Kamei Y, Maldonado E, Shihab E, Ubayashi N. Using analytics to quantify the interest of self-admitted technical debt. In: Proc. of the 1st Int'l Workshop on Technical Debt Analytics. 2016. 68–71.
- [34] Mensah S, Keung J, Bosu MF, Bennin KE. Rework effort estimation of self-admitted technical debt. In: Proc. of the 1st Int'l Workshop on Technical Debt Analytics. 2016. 72–75.
- [35] Zazworka N, Spinola RO, Vetro A, Shull F, Seaman C. A case study on effectively identifying technical debt. In: Proc. of the 17th Int'l Conf. on Evaluation and Assessment in Software. Porto de Galinhas: ACM, 2013. 42–47. [doi: [10.1145/2460999.2461005](https://doi.org/10.1145/2460999.2461005)]
- [36] Rocha JC, Zapalowski V, Nunes I. Understanding technical debt at the code level from the perspective of software developers. In: Proc. of the 31st Brazilian Symp. on Software Engineering. Fortaleza: ACM, 2017. 64–73. [doi: [10.1145/3131151.3131164](https://doi.org/10.1145/3131151.3131164)]
- [37] Hata H, Treude C, Kula RG, Ishio T. 9.6 million links in source code comments: Purpose, evolution, and decay. In: Proc. of the IEEE/ACM 41st Int'l Conf. on Software Engineering. Montreal: IEEE, 2019. 1211–1221. [doi: [10.1109/ICSE.2019.00123](https://doi.org/10.1109/ICSE.2019.00123)]
- [38] Pascarella L. Classifying code comments in Java mobile applications. In: Proc. of the 5th Int'l Conf. on Mobile Software Engineering and Systems. Gothenburg: ACM, 2018. 39–40. [doi: [10.1145/3197231.3198444](https://doi.org/10.1145/3197231.3198444)]
- [39] Aversano L, Iammarino M, Carapella M, Del Vecchio A, Nardi L. On the relationship between self-admitted technical debt removals and technical debt measures. *Algorithms*, 2020, 13(7): 168. [doi: [10.3390/a13070168](https://doi.org/10.3390/a13070168)]
- [40] Fleiss JL, Levin B, Paik MC, Paik SM. The Measurement of Interrater Agreement. *Statistical Methods for Rates and Proportions*. 3rd ed., Hoboken: John Wiley, 2004. 212–236.
- [41] Ren XX, Xing ZC, Xia X, Lo D, Wang XY, Grundy J. Neural network-based detection of self-admitted technical debt: From performance to explainability. *ACM Trans. on Software Engineering and Methodology*, 2019, 28(3): 15. [doi: [10.1145/3324916](https://doi.org/10.1145/3324916)]
- [42] Mensah S, Keung J, Svajlenko J, Bennin KE, Mi Q. On the value of a prioritization scheme for resolving self-admitted technical debt.

- Journal of Systems and Software, 2018, 135: 37–54. [doi: [10.1016/j.jss.2017.09.026](https://doi.org/10.1016/j.jss.2017.09.026)]
- [43] Zampetti F, Serebrenik A, Penta MD. Was self-admitted technical debt removal a real removal?: An in-depth perspective. In: Proc. of the 15th Int'l Conf. on Mining Software Repositories. Gothenburg: ACM, 2018. 526–536. [doi: [10.1145/3196398.3196423](https://doi.org/10.1145/3196398.3196423)]
 - [44] Guo ZQ, Liu SR, Liu JP, Li YH, Chen L, Lu HM, Zhou YM, Xu BW. MAT: A simple yet strong baseline for identifying self-admitted technical debt. arXiv:1910.13238, 2019.
 - [45] Vassallo C, Zampetti F, Romano D, Beller M, Panichella A, Penta MD, Zaidman A. Continuous delivery practices in a large financial organization. In: Proc. of the IEEE Int'l Conf. on Software Maintenance and Evolution. Raleigh: IEEE, 2016. 519–528. [doi: [10.1109/ICSME.2016.72](https://doi.org/10.1109/ICSME.2016.72)]
 - [46] Palomba F, Zaidman A, Oliveto R, de Lucia A. An exploratory study on the relationship between changes and refactoring. In: Proc. of the IEEE/ACM 25th Int'l Conf. on Program Comprehension. Buenos Aires: IEEE, 2017. 176–185. [doi: [10.1109/ICPC.2017.38](https://doi.org/10.1109/ICPC.2017.38)]
 - [47] de Freitas Farias MA, de Mendonça Neto MG, Kalinowski M, Spinola RO. Identifying self-admitted technical debt through code comment analysis with a contextualized vocabulary. Information and Software Technology, 2020, 121: 106270. [doi: [10.1016/j.infsof.2020.106270](https://doi.org/10.1016/j.infsof.2020.106270)]
 - [48] Li YK, Soliman M, Avgeriou P. Identification and remediation of self-admitted technical debt in issue trackers. In: Proc. of the 46th Euromicro Conf. on Software Engineering and Advanced Applications. Portoroz: IEEE, 2020. 495–503. [doi: [10.1109/SEAA51224.2020.00083](https://doi.org/10.1109/SEAA51224.2020.00083)]
 - [49] Liu JK, Huang Q, Xia X, Shihab E, Lo D, Li SP. Is using deep learning frameworks free?: Characterizing technical debt in deep learning frameworks. In: Proc. of the ACM/IEEE 42nd Int'l Conf. on Software Engineering: Software Engineering in Society. Seoul: ACM, 2020. 1–10. [doi: [10.1145/3377815.3381377](https://doi.org/10.1145/3377815.3381377)]
 - [50] Da S. Maldonado E, Abdalkareem R, Shihab E, Serebrenik A. An empirical study on the removal of self-admitted technical debt. In: Proc. of the Int'l Conf. on Software Maintenance and Evolution. Shanghai: IEEE, 2017. 238–248. [doi: [10.1109/ICSME.2017.8](https://doi.org/10.1109/ICSME.2017.8)]
 - [51] Maipradit R, Treude C, Hata H, Matsumoto K. Wait for it: Identifying “On-Hold” self-admitted technical debt. Empirical Software Engineering, 2020, 25(5): 3770–3798. [doi: [10.1007/s10664-020-09854-3](https://doi.org/10.1007/s10664-020-09854-3)]
 - [52] Iammarino M, Zampetti F, Aversano L, Penta MD. Self-admitted technical debt removal and refactoring actions: Co-occurrence or more? In: Proc. of the IEEE Int'l Conf. on Software Maintenance and Evolution. Cleveland: IEEE, 2019. 186–190. [doi: [10.1109/ICSME.2019.00029](https://doi.org/10.1109/ICSME.2019.00029)]
 - [53] Zampetti F, Serebrenik A, Penta MD. Automatically learning patterns for self-admitted technical debt removal. In: Proc. of the IEEE 27th Int'l Conf. on Software Analysis, Evolution and Reengineering. London: IEEE, 2020. 355–366. [doi: [10.1109/SANER48275.2020.9054868](https://doi.org/10.1109/SANER48275.2020.9054868)]
 - [54] Sierra G, Tahmid A, Shihab E, Tsantalis N. Is self-admitted technical debt a good indicator of architectural divergences? In: Proc. of the 26th IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering. Hangzhou: IEEE, 2019. 534–543. [doi: [10.1109/SANER.2019.8667999](https://doi.org/10.1109/SANER.2019.8667999)]
 - [55] Xiao L, Cai YF, Kazman R, Mo R, Feng Q. Identifying and quantifying architectural debt. In: Proc. of the 38th Int'l Conf. on Software Engineering. Austin: ACM, 2016. 488–498. [doi: [10.1145/2884781.2884822](https://doi.org/10.1145/2884781.2884822)]
 - [56] 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](https://doi.org/10.1109/CSMR.2008.4493342)]
 - [57] Collard ML, Decker MJ, Maletic JI. Lightweight transformation and fact extraction with the srcML toolkit. In: Proc. of the IEEE 11th Int'l Working Conf. on Source Code Analysis and Manipulation. Williamsburg: IEEE, 2011. 173–184. [doi: [10.1109/SCAM.2011.19](https://doi.org/10.1109/SCAM.2011.19)]
 - [58] Liu ZX, Huang Q, Xia X, Shihab E, Lo D, Li SP. SATD detector: A text-mining-based self-admitted technical debt detection tool. In: Proc. of the 40th Int'l Conf. on Software Engineering. Gothenburg: ACM, 2018. 9–12. [doi: [10.1145/3183440.3183478](https://doi.org/10.1145/3183440.3183478)]
 - [59] Wattanakriengkrai S, Srisermphoak N, Sintoplerchaikul S, Choetkiertikul M, Ragkhitwetsagul C, Sunetnanta T, Hata H, Matsumoto K. Automatic classifying self-admitted technical debt using n-gram idf. In: Proc. of the 26th Asia-Pacific Software Engineering Conf. Putrajaya: IEEE, 2019. 316–322. [doi: [10.1109/APSEC48747.2019.00050](https://doi.org/10.1109/APSEC48747.2019.00050)]
 - [60] Yan M, Xia X, Shihab E, Lo D, Yin JW, Yang XH. Automating change-level self-admitted technical debt determination. IEEE Trans. on Software Engineering, 2019, 45(12): 1211–1229. [doi: [10.1109/TSE.2018.2831232](https://doi.org/10.1109/TSE.2018.2831232)]
 - [61] Yu Z, Fahid FM, Tu H, Menzies T. Identifying self-admitted technical debts with jitterbug: A two-step approach. arXiv: 2002.11049, 2020.
 - [62] de O. Passos AF, de Freitas Farias MA, de Mendonça Neto MG, Spinola RO. A study on identification of documentation and requirement technical debt through code comment analysis. In: Proc. of the 17th Brazilian Symp. on Software Quality. Curitiba: ACM, 2018. 21–30. [doi: [10.1145/3275245.3275248](https://doi.org/10.1145/3275245.3275248)]

- [63] de F. Farias MA, Xisto R, Santos MS, Fontes RS, Colaço M, Spinola R, Mendonça M. Identifying technical debt through a code comment mining tool. In: Proc. of the XV Brazilian Symp. on Information Systems. Aracaju: ACM, 2019. 18. [doi: [10.1145/3330204.3330227](https://doi.org/10.1145/3330204.3330227)]
- [64] Wattanakriengkrai S, Maipradit R, Hata H, Choetkiertikul M, Sunetnanta T, Matsumoto K. Identifying design and requirement self-admitted technical debt using N-gram IDF. In: Proc. of the 9th Int'l Workshop on Empirical Software Engineering in Practice. Nara: IEEE, 2018. 7–12. [doi: [10.1109/TWESEP.2018.00010](https://doi.org/10.1109/TWESEP.2018.00010)]
- [65] Farias MADF, Santos JA, Kalinowski M, Mendonça M, Spinola RO. Investigating the identification of technical debt through code comment analysis. In: Proc. of the 18th Int'l Conf. on Enterprise Information Systems. Rome: Springer, 2016. 284–309. [doi: [10.1007/978-3-319-62386-3_14](https://doi.org/10.1007/978-3-319-62386-3_14)]
- [66] Santos RM, Junior MCR, Neto MGM. Self-admitted technical debt classification using LSTM neural network. In: Latifi S, ed. Proc. of the 17th Int'l Conf. on Information Technology–New Generations (ITNG 2020). Cham: Springer, 2020. 679–685. [doi: [10.1007/978-3-030-43020-7_93](https://doi.org/10.1007/978-3-030-43020-7_93)]
- [67] Miyake Y, Amasaki S, Aman H, Yokogawa T. A replicated study on relationship between code quality and method comments. In: Lee R, ed. Applied Computing and Information Technology. Cham: Springer, 2017. 17–30. [doi: [10.1007/978-3-319-51472-7_2](https://doi.org/10.1007/978-3-319-51472-7_2)]
- [68] Flisar J, Podgorelec V. Enhanced feature selection using word embeddings for self-admitted technical debt identification. In: Proc. of the 44th Euromicro Conf. on Software Engineering and Advanced Applications. Prague: IEEE, 2018. 230–233. [doi: [10.1109/SEAA.2018.00045](https://doi.org/10.1109/SEAA.2018.00045)]
- [69] Flisar J, Podgorelec V. Identification of self-admitted technical debt using enhanced feature selection based on word embedding. IEEE Access, 2019, 7: 106475–106494. [doi: [10.1109/ACCESS.2019.2933318](https://doi.org/10.1109/ACCESS.2019.2933318)]
- [70] Zhao F, Tang YM, Yang YB, Lu HM, Zhou YM, Xu BW. Is learning-to-rank cost-effective in recommending relevant files for bug localization? In: Proc. of the IEEE Int'l Conf. on Software Quality, Reliability and Security. Vancouver: IEEE, 2015. 298–303. [doi: [10.1109/QRS.2015.49](https://doi.org/10.1109/QRS.2015.49)]
- [71] Ngweight. 2018. <https://github.com/iwnsew/ngweight>
- [72] Feurer M, Klein A, Eggenberger K, Springenberg JT, Blum M, Hutter F. Auto-sklearn: Efficient and robust automated machine learning. In: Hutter F, Kotthoff L, Vanschoren J, eds. Automated Machine Learning. Cham: Springer, 2019. 113–134. [doi: [10.1007/978-3-030-05318-5_6](https://doi.org/10.1007/978-3-030-05318-5_6)]
- [73] Buse RPL, Weimer WR. Learning a metric for code readability. IEEE Trans. on Software Engineering, 2010, 36(4): 546–558. [doi: [10.1109/TSE.2009.70](https://doi.org/10.1109/TSE.2009.70)]
- [74] Sebastiani F. Machine learning in automated text categorization. ACM Computing Surveys, 2002, 34(1): 1–47. [doi: [10.1145/505282.505283](https://doi.org/10.1145/505282.505283)]
- [75] Salton G, McGill MJ. Introduction to Modern Information Retrieval. New York: McGraw-Hill, 1983.
- [76] Santos R, Santos I, Júnior MC, Neto M. Long term-short memory neural networks and word2vec for self-admitted technical debt detection. In: Proc. of the 22nd Int'l Conf. on Enterprise Information Systems. Prague, 2020. 157–165. [doi: [10.5220/0009796001570165](https://doi.org/10.5220/0009796001570165)]
- [77] Cohen J. A coefficient of agreement for nominal scales. Educational and Psychological Measurement, 1960, 20(1): 37–46. [doi: [10.1177/001316446002000104](https://doi.org/10.1177/001316446002000104)]
- [78] Prati RC, Batista GEAPA, Monard MC. Class imbalances *versus* class overlapping: An analysis of a learning system behavior. In: Proc. of the 3rd Mexican Int'l Conf. on Artificial Intelligence. Mexico City: Springer, 2004. 312–321. [doi: [10.1007/978-3-540-24694-7_32](https://doi.org/10.1007/978-3-540-24694-7_32)]
- [79] Batista GEAPA, Prati RC, Monard MC. A study of the behavior of several methods for balancing machine learning training data. ACM SIGKDD Explorations Newsletter, 2004, 6(1): 20–29. [doi: [10.1145/1007730.1007735](https://doi.org/10.1145/1007730.1007735)]
- [80] Guo ZQ, Zhou HC, Liu SR, Li YH, Chen L, Zhou YM, Xu BW. Information retrieval based bug localization: Research problem, progress, and challenges. Ruan Jian Xue Bao/Journal of Software, 2020, 31(9): 2826–2854 (in Chinese with English abstract). <http://www.jos.org.cn/1000-9825/6087.htm> [doi: [10.13328/j.cnki.jos.006087](https://doi.org/10.13328/j.cnki.jos.006087)]
- [81] Chen X, Gu Q, Liu WS, Liu SL, Ni C. Survey of static software defect prediction. Ruan Jian Xue Bao/Journal of Software, 2016, 27(1): 1–25 (in Chinese with English abstract). <http://www.jos.org.cn/1000-9825/4923.htm> [doi: [10.13328/j.cnki.jos.004923](https://doi.org/10.13328/j.cnki.jos.004923)]

附中文参考文献:

- [21] 刘亚珏, 李兵, 李增扬, 梁鹏, 吴闽泉. 软件集成开发环境的技术债务管理研究. 计算机科学, 2017, 44(11): 15–21, 40. [doi: [10.11896/j.issn.1002-137X.2017.11.003](https://doi.org/10.11896/j.issn.1002-137X.2017.11.003)]
- [22] 张云洁, 张璇, 丁浩, 王旭. 需求变更技术债务研究. 计算机科学, 2018, 45(9): 89–93. [doi: [10.11896/j.issn.1002-137X.2018.09.013](https://doi.org/10.11896/j.issn.1002-137X.2018.09.013)]

- [23] 杨珺. 基于技术债务度量的安全评估方法的研究[硕士学位论文]. 北京: 北京交通大学, 2016.
- [24] 丁浩. 需求变更技术债务研究[硕士学位论文]. 昆明: 云南大学, 2017.
- [25] 张云洁. 基于边际贡献的需求变更技术债务研究[硕士学位论文]. 昆明: 云南大学, 2019.
- [80] 郭肇强, 周慧聪, 刘释然, 李言辉, 陈林, 周毓明, 徐宝文. 基于信息检索的缺陷定位: 问题、进展与挑战. 软件学报, 2020, 31(9): 2826–2854. <http://www.jos.org.cn/1000-9825/6087.htm> [doi: 10.13328/j.cnki.jos.006087]
- [81] 陈翔, 顾庆, 刘望舒, 刘树龙, 倪超. 静态软件缺陷预测方法研究. 软件学报, 2016, 27(1): 1–25. <http://www.jos.org.cn/1000-9825/4923.htm> [doi: 10.13328/j.cnki.jos.004923]



郭肇强(1994—), 男, 博士生, 主要研究领域为软件质量保障, 软件缺陷预测.



陈林(1979—), 男, 博士, 副教授, CCF 高级会员, 主要研究领域为软件分析测试.



刘释然(1990—), 男, 博士生, 主要研究领域为实证软件工程.



周毓明(1974—), 男, 博士, 教授, 博士生导师, CCF 专业会员, 主要研究领域为软件维护, 测试与分析.



谭婷婷(1994—), 女, 硕士, 主要研究领域为软件测试.



徐宝文(1961—), 男, 博士, 教授, 博士生导师, CCF 会士, 主要研究领域为程序设计语言, 软件分析与测试.



李言辉(1981—), 男, 博士, 助理研究员, CCF 专业会员, 主要研究领域为软件演化与维护, 软件测试.