软件学报 ISSN 1000-9825, CODEN RUXUEW
[doi: 10.13328/j.cnki.jos.007290] [CSTR: 32375.14.jos.007290]
©中国科学院软件研究所版权所有.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

软件设计模式检测技术: 现状、挑战和展望*

王 雷[1,2,3], 袁 野[1], 王国仁[1]

[1](北京理工大学 计算机学院, 北京 100081)
[2](延安大学 数学与计算机科学学院, 陕西 延安 716000)
[3](中国轻工业工业互联网与大数据重点实验室, 北京 100081)
通信作者: 袁野, E-mail: yuan-ye@bit.edu.cn; 王国仁, E-mail: wanggrbit@126.com

摘 要: 设计模式检测是软件工程领域中非常重要的研究课题. 国内外很多学者致力于设计模式检测问题的研究与解决, 取得了丰硕的研究成果. 对当前软件设计模式检测技术进行综述并展望了其前景. 首先, 简要介绍软件设计模式检测领域的发展历程, 讨论并总结了设计模式的检测对象和特征类型, 给出了设计模式检测评估指标. 然后, 总结了设计模式检测技术现有的分类方法, 引出了分类方法. 根据设计模式检测技术发展的时间线从非机器学习设计模式检测、机器学习设计模式检测、基于预训练语言模型的设计模式检测这 3 类方法出发探讨了当前软件设计模式检测技术的研究现状和最新进展, 并对当前成果进行了总结和比较. 最后, 分析了该领域存在的主要问题与挑战, 指出了今后值得进一步研究的方向以及可能的解决方案. 涵盖了从早期的非机器学习方法到利用机器学习技术, 再到现代预训练语言模型的应用, 全面系统地展现了该领域的发展历程、最新进展和未来发展前景, 对于该领域今后的研究方向和思路具有指导意义.

关键词: 设计模式检测; 研究综述; 机器学习; 预训练语言模型; 软件逆向工程; 软件开发方法; 软件设计; 软件体系结构

中图法分类号: TP311

中文引用格式: 王雷, 袁野, 王国仁. 软件设计模式检测技术: 现状、挑战和展望. 软件学报. http://www.jos.org.cn/1000-9825/7290.htm

英文引用格式: Wang L, Yuan Y, Wang GR. Software Design Pattern Detection Techniques: Current Status, Challenges and Prospects. Ruan Jian Xue Bao/Journal of Software (in Chinese). http://www.jos.org.cn/1000-9825/7290.htm

Software Design Pattern Detection Techniques: Current Status, Challenges and Prospects

WANG Lei[1,2,3], YUAN Ye[1], WANG Guo-Ren[1]

[1](School of Computer Science & Technology, Beijing Institute of Technology, Beijing 100081, China)
[2](College of Mathematics and Computer Science, Yan’an University, Yan’an 716000, China)
[3](Key Laboratory of Industrial Internet and Big Data, China National Light Industry, Beijing 100081, China)

Abstract: Design pattern detection is an essential research topic in software engineering. Many scholars both domestically and internationally have dedicated their efforts to researching and resolving design pattern detection, thereby yielding fruitful results. This study reviews the current technologies in software design pattern detection and points out their prospects. Firstly, this study briefly introduces the development history of software design pattern detection, discusses the objects of design pattern detection, summarizes the feature types of design patterns, and provides the evaluation indexes of design pattern detection. Then, the existing classification methods for design pattern detection techniques are summarized, and the classification method proposed in this study is introduced. Next, according to the development timeline of design pattern detection technologies, the research status and latest advancements of current software design

* 基金项目: 国家自然科学基金 (61932004); 陕西省教育厅专项科研计划 (23JK0724); 中国轻工业工业互联网与大数据重点实验室开放课题基金 (IIBD-2021-KF10)
收稿时间: 2024-05-11; 修改时间: 2024-06-22; 采用时间: 2024-09-03; jos 在线出版时间: 2025-03-12

pattern detection technologies are discussed from three approaches, including non-machine learning design pattern detection, machine learning design pattern detection, and design pattern detection based on pre-trained language models, with the current achievements summarized and compared. Finally, the main problems and challenges in this field are analyzed, and further research directions and potential solutions are pointed out. Covering contents from early non-machine learning methods and utilization of machine learning technologies to the application of modern pre-trained language models, this study comprehensively and systematically presents the development history, latest advancements, and prospects of this field. It provides valuable guidance for future research directions and ideas within this area.

Key words: design pattern detection; research review; machine learning; pre-trained language model; software reverse engineering; software development method; software design; software architecture

设计模式[1]是面向对象软件开发中用于解决特定问题的标准方法, 广泛应用于现代软件行业以复用成功的设计和提高系统质量[2,3]. 然而, 在实际软件开发和维护过程中, 系统中经常缺少对设计模式使用信息的记录或这些记录与系统更新不完全匹配, 尤其是一些存在多年的遗留软件 (legacy software)[4], 影响了系统的可理解性和可维护性[5,6]. 随着软件开发项目的规模和复杂度不断增加[7,8], 提高开发效率和软件质量成为产业界迫切需要解决的问题. 自动或半自动地从系统设计或源代码中检测出相应的设计模式, 可以帮助软件开发和维护人员理解大规模、高复杂性软件系统的设计思路[9,10], 进而缩短软件开发周期, 降低开发和维护成本. 因此, 设计模式检测成为目前软件逆向工程领域中的一个研究热点.

目前有一些学者对设计模式检测领域的研究进行了综述. 例如: Zhang 等人[9]从设计模式的形式定义、中间表示形式 (例如抽象语法树 (abstract syntax tree, AST), 有向图) 和设计模式挖掘方法 (分为传统的挖掘方法和基于机器学习的挖掘方法) 3 个方面阐述了设计模式检测领域的研究进展. Yarahmadi 等人[10]从数据表示、检测到的设计模式类型、不同方法的优缺点、定量结果等多个方面综述了现有的关于设计模式检测的文献. Chaturvedi 等人[11]从结构分析、行为分析和语义分析这 3 类方法出发综述了不同的设计模式检测方法和工具. Al-Obeidallah 等人[12]从检测方法 (数据库检索方法, 基于度量的方法, 基于 UML 结构、图和矩阵的方法等其他方法) 和分析风格 (结构分析方法, 行为分析方法和语义分析方法) 等几个方面对设计模式检测领域进行了综述. Mhawish 等人[13]根据分析类型、识别类型和中间表示对当前检测设计模式的技术和工具进行了分类和讨论; Rasool 等人[14]从分析类型和检测方法两个方面出发综述了设计模式检测的研究进展. Dong 等人[15]从模式特征 (结构方面、行为方面和语义方面等)、中间表示、精确/近似匹配技术、可视化、自动化或人机交互工具支撑、恢复的模式和实验 7 个角度讨论和比较了设计模式检测方法. Priya[16]从分析类型、检测步骤、特征类型、检测方法和评价标准等几个方面对当前设计模式检测技术进行了全面分析和讨论. 在文献 [17] 中, 作者将设计模式检测技术分类为基于图论的设计模式检测、基于形式化技术的设计模式检测、基于软件度量的设计模式检测和基于人工智能的设计模式检测这 4 类进行总结和评述.

本文根据软件设计检测领域发展的时间线, 将当前技术分为非机器学习的设计模式检测、机器学习的设计模式检测和基于预训练语言模型的设计模式检测这 3 类进行综述. 第 1 节概述设计模式检测领域, 包括设计模式检测的发展历程、检测对象、特征类型以及评估指标. 第 2 节总结当前根据所使用的技术、所考虑的特征类型、分析类型等不同标准的设计模式检测技术分类方法, 引出本文根据设计模式检测技术发展的时间线的分类方法. 第 3 节从上述 3 类方法出发对设计模式检测的研究现状进行探讨, 给出几个代表性的技术在 JHotDraw、JRefactory 和 JUnit 这 3 个开源项目上的应用效果并进行分析和讨论. 第 4 节分别从文献和这 3 类方法的角度对当前研究进行总结和比较. 第 5 节分析该领域存在的主要问题与挑战, 对未来研究方向进行展望. 第 6 节对全文进行总结.

本文的主要贡献如下.

(1) 发展历程回顾: 详细回顾了自 1996 年 Gamma 等人提出 23 种 GoF 设计模式以来, 设计模式检测领域随着敏捷开发、机器学习、预训练语言模型等计算机技术的出现和发展而发生的变革历程. 这为理解该领域的技术进展和演变提供了全新视角.

(2) 检测技术的分类与讨论: 不同于已有的文献综述[9−17]根据使用的技术、所考虑的特征类型、分析类型等

对设计模式检测技术进行分类, 本文根据软件设计检测领域发展的时间线对当前主要的设计模式检测技术进行了分类和讨论, 包括早期的非机器学习的设计模式检测技术, 以及现代的机器学习和基于预训练语言模型的技术, 从而可以展现该领域当前研究全貌和最新趋势. 本文特别强调了基于预训练语言模型的设计模式检测技术, 这是当前人工智能领域的前沿技术. 文中讨论了 BERT、GPT 等模型在设计模式检测中的应用, 这些内容在先前的综述中不常见.

(3) 问题与挑战分析以及前景展望: 论文分析了当前设计模式检测面临的主要问题和挑战, 如公开可用的数据集匮乏、动态行为的捕获和表示困难、新的或未知的模式类别难以识别等, 并提出了可能的解决方案, 为未来研究指明了方向.

1 软件设计模式检测领域概述

1.1 发展历程

为分析软件设计模式检测领域的发展历程，我们进行了文献检索和统计。我们对 Web of Science、Compendex、SpringerLink、Wiley Online Library、EBSCOhost、IEEE Xplore、ScienceDirect、CiteseerX Library、Google Scholar、ACM Digital Library 这 10 个主要数据库进行了设计模式检测领域文献的检索并对检索结果进行了排查, 获得了 170 篇具有代表性的高质量文献作为筛选的结果 (时间范围从 1996 年至 2023 年, 已经去除了同一篇文献被不同的数据库收录的重复现象). 将筛选得到的设计模式检测领域文献, 按照其发表年份进行统计, 结果如图 1 所示.

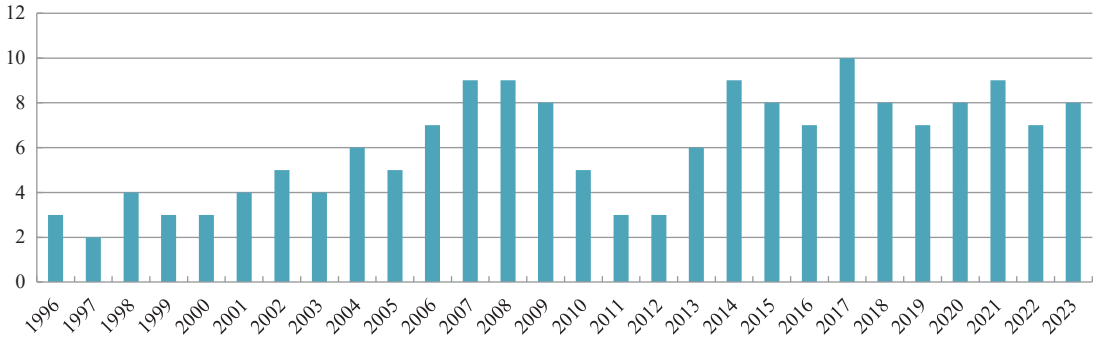


图 1 设计模式检测领域文献出版年份统计表

设计模式作为软件工程中的一个重要概念被广泛认知是从 Gamma 等人的著作[1]开始的. 这本著作详细介绍了 23 种设计模式, 也被称为四人组 (gang of four, GoF) 模式, 这些模式后来成为了软件开发中的经典. 尽管随着软件开发领域的发展已经出现了许多其他的设计模式, GoF 模式仍是应用最广泛、复用性最高、优势最为显著的核心设计模式. 因此, 目前国内外的设计模式检测方法主要针对 GoF 模式[18]. GoF 设计模式通常被分为 3 种基本类型, 这些类型反映了设计模式解决的问题的本质. 这 3 种类型是: 创建型 (creational)、结构型 (structural) 和行为型 (behavioral). 创建型模式主要涉及对象创建机制, 使得创建对象的方式更加灵活并适应程序的需要. 结构型模式主要用于设计对象和类的组合, 以形成更大、更复杂的面向对象程序结构. 行为型模式专注于对象之间的通信. GoF 设计模式的详细目录可参见文献 [1]. 另外, 近年来也有一些学者开始关注架构级设计模式 (architectural design pattern) 的检测. 架构级设计模式是一类高层次的设计模式, 用于解决系统架构设计中的常见问题和挑战. 它们提供了一组经过验证的解决方案, 帮助构建复杂系统的整体结构. 架构级设计模式关注系统的整体组织和组件之间的交互, 而不仅仅是对象和类的低层次结构和行为. 典型的架构设计模式包括 Model-View-Controller (MVC)、Model-View-Presenter (MVP)、Model-View-ViewModel (MVVM)、Model-View-Whatever (MVW)、微服务架构 (microservices architecture)、分层架构 (layered architecture)、事件驱动架构 (event-driven architecture, EDA) 等.

由图 1 可知, 在 GoF 设计模式发布后不久, 就陆续有学者对设计模式的检测问题进行了研究. 最早的研究可以追溯到 GoF 设计模式发布后的第 2 年, Krämer 等人[19]和 Brown [20]各自的工作. Krämer 等人[19]于 1996 年提出了一个名为 Pat 的系统, 该系统能从 C++头文件中提取设计信息, 并使用 Prolog 语言来表示这些信息, 进而搜索特定的结构型设计模式实例. 同年, Brown [20]研究了 Smalltalk 的设计模式检测问题. Brown [20]开发了一个名为 KT 的软件工具, 该工具可以从 Smalltalk 代码中逆向工程出设计图, 然后使用这些信息来检测 GoF 模式的 4 种知名的设计模式. 他们各自的研究展现了自动化设计模式检测的可能性, 为后期设计模式检测的研究提供了重要的方法论基础和可以借鉴的思路, 成为设计模式检测领域的奠基之作, 受到了该领域学者的广泛关注 (截至 2024 年 4 月 2 日文献 [19,20] 被引量分别为 456 次和 257 次).

为了提升软件开发团队的工作效率和快速适应变化的能力, Beck 和其他 16 位知名软件专家于 2001 年 2 月共同起草了敏捷软件开发宣言[21]. 2006 年, 包括 Google、Microsoft、IBM、Amazon 和华为在内的多家大型企业开始在大规模项目中采用敏捷方法, 引发了敏捷应用的热潮[22]. 同年, ThoughtWorks 公司在北京举办了首届中国敏捷软件开发大会 (Agile China Conference), 标志着敏捷方法在我国的正式推广[22]. 敏捷开发声明“工作的软件高于详尽的文档 (working software over comprehensive documentation)”以及“响应变化高于遵循计划 (responding to change over following a plan)”[4,21]. 以 XP、Scrum、看板 (kanban) 等为代表的敏捷开发方法的推广和大范围的应用导致系统中缺少对设计模式使用信息的记录或记录与系统更新不完全匹配, 业界对于设计模式检测技术的需求也随之增加. 由图 1 可知, 从 2006 年开始的几年里, 该领域的研究工作显著增多. 国内外学者相继将有向图、形式化方法、AST、知识表示与推理等技术应用于设计模式的检测[23–48], 设计模式检测领域得到了空前的繁荣和发展.

早期的设计模式检测技术大多是非机器学习的设计模式检测, 检测规则是通过人工从设计模式的理论描述中获取的. 设计模式理论描述的模糊性与实际工程项目中设计模式使用的灵活性之间的对立导致非机器学习的设计模式检测技术性能的提升遇到了瓶颈, 由图 1 可知 2011 年前后的几年里该领域的发文量处于较低产出水平.

随着大数据的涌现和计算机算力的提升, 机器学习领域尤其是深度学习[49]在 2012 年之后重新崛起并迅猛发展. 机器学习技术通过算法使得机器能从大量数据中学习规律或者规则, 从而对新的样本做出判断或预测, 非常适合于规则非常复杂和灵活的设计模式检测问题. 机器学习技术在设计模式检测问题上得到了成功的应用且取得了可喜的效果[29,50–71], 再次引发了设计模式检测领域的研究热潮.

近几年, BERT[72]、GPT[73,74]、BART[75]等预训练语言模型引发了人工智能以及自然语言处理领域革命性的热潮. 这些预训练模型基于大规模的自然语言以及源代码语料进行自监督的预训练, 使得这些模型本身具备了处理自然语言和源代码的能力. 这为设计模式检测问题提供了一个新的契机. 目前已经有学者尝试借助预训练语言模型检测设计模式[76–80], 极大地提升了设计模式检测的准确率以及效率、泛化能力、鲁棒性等多个方面的性能.

设计模式检测技术的发展脉络大致如图 2 所示.

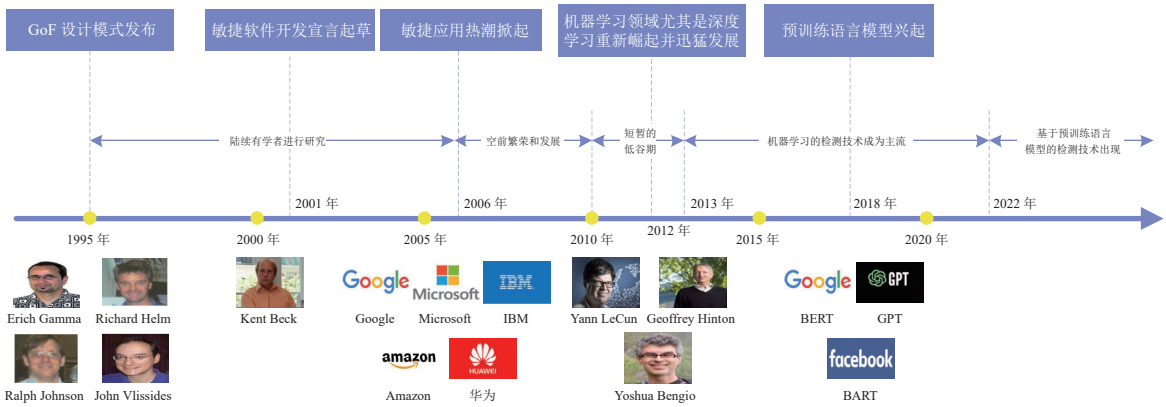


图 2 设计模式检测技术的大致发展脉络

1.2 检测对象

不同的检测方法具有不同类型的输入对象[81], 常见的输入对象可分为源代码级和系统设计级. 源代码级输入以待检测系统的源代码作为检测对象. 在当前文献中, 以源代码作为输入的最多, 主要是 C++、Java、Smalltalk、Ada、C#、Python 等面向对象语言编写的源程序. 一些文献在源代码的基础上使用代码的注释和系统的文档[25,65,66]以及 Java 字节码文件[35,68]作为辅助, 以便更准确地检测设计模式. 系统设计级输入关注待检测系统的设计模型, 例如系统的 UML 类图和序列图[25,26,28,31,33,34,42−46,48,65]、Petri 网[65].

1.3 特征类型

综合当前设计模式检测领域的文献, 作者将模式特征划分为以下 5 类.

(1) 结构特征

结构特征主要指实体 (类/接口/对象) 及实体之间的关系, 包括类之间泛化 (generalization)、组合 (composition)、聚合 (aggregation)、依赖 (dependency) 等关系, 友元 (friendship) 关系, 类与接口之间的实现 (implementation) 关系, 类和类成员的修饰符, 方法 (成员函数) 及其参数和返回类型, 属性 (成员变量) 及其数据类型, 抽象类 (abstract class) 和抽象方法 (abstract method), 对象创建 (object creation) 等.

泛化关系指的是继承 (inheritance) 关系, 子类继承父类的属性和方法. 例如, 模板方法 (template method) 模式中, 子类继承抽象父类并实现父类定义的抽象方法, 以完成具体操作步骤. 工厂方法 (factory method) 模式中, 具体工厂类继承抽象工厂类, 并实现抽象工厂类中定义的工厂方法以创建具体产品对象. 两个类之间存在关联 (association) 关系, 表明这两个类的实例之间存在语义上的联系. 组合和聚合都是关联的特例. 组合关系表示一个类包含另一个类的实例, 且被包含的实例是类的一部分, 生命周期与包含类相同. 例如, 组合 (composition) 模式中, 容器对象包含叶子对象. 聚合 (aggregation) 关系类似于组合, 但被包含的实例是独立存在的, 生命周期与包含类不同. 例如, 观察者 (observer) 模式中, 主题对象持有多个观察者对象的引用, 这是一种典型的聚合关系. 依赖关系表示一个元素依赖于另一个元素的变化, 通过依赖关系分析可以识别模式中的参与者. 例如, 观察者模式中, 主题类 (subject) 依赖于观察者类. 友元关系允许一个类访问另一个类的私有和保护成员, 常见于 C++的设计模式实现中. 例如, 在一些复杂模式中可能需要这种特殊访问权限. 类与接口之间的实现关系指的是一个类实现一个接口. 例如, 装饰者 (decorator) 模式中的组件接口和具体组件类, 策略 (strategy) 模式中的策略接口和具体策略类. 类和类成员的修饰符 (public, private, protected) 定义了类和类的属性 (成员变量)、方法 (成员函数) 的可见性. 例如, 单例 (singleton) 模式中, 构造函数通常是私有的以控制实例的创建, 类包含一个私有的静态实例变量防止直接访问, 提供一个公有的静态方法来获取实例. 方法(成员函数) 及其参数和返回类型在设计模式检测中具有重要作用, 通过分析这些特征, 可以帮助识别和区分不同的设计模式. 例如, 工厂方法通常返回一个产品 (product) 接口或抽象产品类型的实例, 返回类型用于识别该模式的具体实现; 策略 (strategy) 模式中策略方法通常具有相同的参数类型和返回类型, 策略接口定义了这些方法. 属性(成员变量) 及其数据类型在设计模式检测中起着关键作用, 因为它们能反映类的状态、行为以及类之间的关系, 从而帮助识别和验证设计模式. 例如, 单例模式中单例类通常包含一个私有的静态实例变量, 用于存储唯一实例; 装饰者模式中装饰者类通常包含一个组件类型的成员变量, 用于引用被装饰的对象. 抽象类和抽象方法在一些设计模式中也起到关键作用, 例如模板方法模式中, 抽象类定义了模板方法, 具体子类实现了抽象方法. 对象创建方式在设计模式检测中也提供了有效的支持信息, 因为不同的设计模式在对象创建方面有不同的实现方式和意图. 创建型设计模式特别关注对象的创建方式, 对象创建方式是识别这些模式的主要依据. 例如, 工厂方法模式通过工厂方法创建具体产品对象, 单例模式通过静态方法创建唯一实例.

对于架构级模式, 结构特征还要关注组件及组件之间的关系, 包括组件之间的层次结构, 数据流动模式, 组件间的交互方式, 组件的特定属性和方法等.

结构特征是设计模式最重要的特征, 通常来说, 从系统源代码或设计模型中提取结构特征相对容易些, 因此目前大多数文献在设计模式的检测中都考虑了结构特征.

(2) 行为特征

行为特征指系统的执行行为, 包括静态行为特征和动态行为特征.

静态行为特征主要指可通过静态分析方法获取的系统行为信息, 而无需执行程序. 静态行为特征通过分析源代码或设计模型中的方法调用 (invocation) 关系、控制流 (control flow)、条件分支逻辑 (conditional branch logic) 等静态信息, 提供系统的行为视图. 设计模式在实现过程中通常会有特定的方法调用序列, 静态分析可以揭示方法之间的调用关系, 帮助理解对象之间的静态交互. 例如, 通过静态分析发现在某个类中调用了工厂方法来创建产品实例, 从而识别出工厂方法模式的实现. 状态 (state) 模式中的上下文 (context) 类会调用状态的方法. 通过控制流分析可以了解代码的执行路径和分支结构, 帮助识别行为模式. 例如, 策略模式中可以通过控制流分析识别出不同策略的选择逻辑. 静态行为特征还包括条件分支逻辑, 通过条件分支逻辑分析可以识别模式中的策略选择或状态转换机制. 例如, 策略模式中, 不同策略的选择通常依赖 if-else 或 switch-case 语句, 通过检测这些条件分支结构, 可以发现潜在的策略模式实现.

动态行为特征是指源代码实际运行过程中的行为, 通常通过执行源代码获取到的方法调用、消息传递、协作方式、事件响应等信息来体现. 与静态行为特征的方法调用不同, 动态行为的方法调用是程序运行过程中实际发生的调用, 包括调用顺序、频率、参数传递、返回值等. 一些设计模式依赖于特定的消息传递机制, 例如中介者 (mediator) 模式通过中介者来协调各个协作对象 (collaboration object) 之间的通信. 对象之间的协作方式也是关键的行为特征, 例如职责链 (chain of responsibility) 模式中请求的传递方式. 动态行为特征还包括系统对事件的响应与处理方式. 通过监控系统运行时的事件处理流程, 可以识别出事件驱动的设计模式. 例如, 在观察者模式中, 可以动态监控主题对象的事件通知机制. 动态行为可以通过动态分析技术来捕捉, 例如插桩、日志记录或运行时监控.

行为特征也是检测设计模式非常重要的因素, 尤其是对于行为型模式来说更是如此. 因此部分文献在考虑结构特征的基础上进一步考虑了静态行为特征或动态行为特征, 以提升行为型模式的检测效果. 通常, 这些文献首先通过结构特征获取到行为型模式的候选实例, 然后通过行为特征对获取到的行为型模式候选实例进行进一步确认, 以此来过滤掉假阳性 (false positive) 实例, 例如文献 [35,36,38]. 此外, 还可以通过行为特征对具有相似结构的模式的实例进行区分, 例如对行为型模式中的命令 (command) 模式与结构型模式中的适配器 (adapter) 模式进行区分, 对行为型模式中的状态模式与策略模式进行区分.

(3) 度量特征

度量特征衡量软件产品或软件开发过程的特定属性, 例如属性 (成员变量) 的个数, 方法 (成员函数) 的个数, 抽象方法的个数, 继承树深度, 子类个数, 父类个数, 扇入 (fan-in), 扇出 (fan-out), 信息流度量, 代码行数 (lines of code, LOC), 圈复杂度 (cyclomatic complexity), 注释行数 (comment lines of code, CLOC) 等.

属性 (成员变量) 的个数统计类中定义的属性 (成员变量) 的数量, 帮助评估类的复杂性和职责. 例如, 状态模式中的状态类可能会有多个成员变量用于存储不同的状态信息. 方法 (成员函数) 的个数统计类中定义的方法 (成员函数) 的数量. 某些设计模式会有特定数量的方法, 如工厂方法模式中的工厂类通常只有一个创建方法. 抽象方法的个数统计抽象类或接口中定义的抽象方法的数量, 例如模板方法 (template method) 模式和策略 (strategy) 模式中, 抽象类或接口通常定义若干抽象方法, 由具体子类实现. 继承树深度衡量类的继承层次结构的深度. 继承树深度大的类通常参与复杂的继承关系, 如组合模式中的容器类和叶子类. 子类个数统计一个类的直接子类数量. 例如工厂方法模式和抽象工厂模式中, 工厂类通常有多个子类, 各自创建不同的产品对象. 父类个数指一个类直接继承的父类数量 (通常为 1, 但在多继承的语言中可能更多). 父类个数可以帮助识别通过多继承结构实现的模式实例. 例如适配器模式在某些多继承语言中 (如 C++) 可以通过多继承来实现. 在这种情况下, 适配器类继承了两个父类, 一个是目标接口, 另一个是被适配的类. 父类个数为 2 可以帮助识别这种模式. 扇入衡量有多少其他类或模块依赖于该类. 高扇入值通常表明该类在系统中具有重要作用, 如外观 (facade) 模式中的外观类. 扇出衡量该类依赖于多少其他类或模块. 高扇出值可能表示该类高度耦合, 如代理 (proxy) 模式中的代理类依赖于多个被代理对象. 信息流度量衡量信息在系统中的流动情况, 包括信息从输入到输出的路径长度和复杂度. 例如在观察者模式中, 信息流度量至关重要, 因为该模式依赖于通知机制的实现和数据流动. LOC 是一个衡量软件项目规模和复杂

度的度量标准, 表示源代码文件中非空行和非注释行的总数. 某些设计模式的实现可能会导致特定类的代码量增加, 如组合模式中的容器类. 圈复杂度衡量代码的复杂程度, 通常通过控制流图中的独立路径数量来计算. 高圈复杂度可能表明复杂的业务逻辑, 如职责链模式中的处理者类. CLOC 是指源代码文件中类或模块用于注释的行数. CLOC 较多的类或方法可能包含复杂的逻辑或重要的设计决策, 可能表明这个部分实现了某个设计模式, 需要特别注意其设计逻辑.

有学者关注带宽、响应时间、成本和冗余级别等系统的整体性能和质量指标, 作为架构级模式检测的一个因素.

一些设计模式检测方法单独通过特定的度量指标就可以辨认出软件中的设计模式, 其优点在于能够提供相对客观的设计模式识别标准, 也取得了不错的效果, 例如文献 [51,55,56]. 也有学者通过度量特征与结构特征、行为特征、语义特征相结合来检测设计模式[29,50,52−54,57,60,63,68,70,77].

(4) 语义特征

设计模式检测中的语义特征关注设计模式的语义信息, 这些特征有助于理解模式在特定上下文中的应用目的和意义, 主要包括设计意图、上下文信息、领域特定语义、命名约定等.

设计意图描述模式的设计目的和使用场景. 例如, 单例模式的设计意图是确保一个类只有一个实例, 并提供一个全局访问点. 上下文信息描述模式在具体应用中的环境和约束条件. 例如, 策略模式的上下文是需要在运行时根据不同条件选择不同算法的场景. 领域特定语义指结合具体应用领域的特定需求和规则. 例如, 在电子商务系统中, 策略模式可能用于动态调整促销策略. 命名约定是指在系统设计和实现中使用的命名规则, 包括类名、方法名、属性名等, 帮助识别模式的应用并提高代码的可读性和一致性. 例如, 在工厂方法或抽象工厂 (abstract factory) 模式中, 通常会看到类名包含“Factory”“Product”等关键词; 命令模式中的命令类通常以“Command”结尾, 便于识别; 装饰者模式中装饰类的命名通常包含“Decorator”字样.

程序代码在某种程度上与自然语言类似, 因为代码中的标识符和自然语言中的单词一样, 具有丰富的语义信息. 因此, 语义特征通常从源代码中提取. 除此之外, 注释在代码中起到了重要的说明和解释作用, 能够帮助开发者和维护人员理解代码的设计意图、逻辑结构和实现细节. 而系统文档通常详细描述了系统的各个方面, 包括设计意图、功能模块、接口定义、使用场景和约束条件. 因此用自然语言书写的注释和系统文档有时也可以成为语义特征的来源.

在非机器学习的设计模式检测技术以及基于决策树 (decision tree)、k-近邻 (k-nearest neighbor, KNN)、支持向量机 (support vector machine, SVM) 等传统机器学习算法的设计模式检测技术为主流的时期, 语义特征主要起增强检测效果的作用. 例如: Jia 等人[27]通过源代码中的注释和文档信息来理解代码的设计意图和上下文信息, 从而帮助识别那些仅靠结构信息难以检测的模式; Dong 等人[31]利用类名和方法名中的命名约定检查设计意图和动机, 进一步区分相似的设计模式; Wang 等人[65]通过结合系统上下文和启发式分析, 帮助补充传统的结构和行为特征检测, 从而更准确地识别设计模式; Thaller 等人[66]通过分析代码注释和文档提取语义信息, 起到辅助验证的作用, 提高检测的准确性和可信度. 近年来, Word2Vec、Code2Vec、Doc2Vec 等自然语言处理的嵌入 (embedding) 技术[64,69,70]以及预训练语言模型[76−80]在源代码的设计模式检测上得到应用. 嵌入技术以及在其基础上发展起来的现代预训练语言模型, 可以从源代码或自然语言书写的注释和文档中提取深层次的语义信息, 语义特征已经逐渐成为设计模式检测最重要、最直接的信息来源.

(5) 补充特征

有学者探究了不属于上述任何特征类型的特殊特征作为补充, 可能涉及代码的实现细节、代码中各部分对于识别设计模式的重要程度等. 例如 Ferenc 等人[53]通过算法特征 (例如循环和递归的数量), 公共方法对父类方法的调用, 非公共方法对父类方法的调用等代码的实现细节进一步分析和过滤适配器和策略模式候选实例; 陈时非等人[79]使用来自 Transformer 网络的注意力权重来确定代码中哪些部分对于设计模式识别最为关键. 一般情况下通过结构特征、行为特征、语义特征和度量特征就可以较好地检测大部分的模式, 补充特征主要是起辅助和优化特定模式检测能力的作用.

1.4 评估指标

设计模式检测技术的有效性评估是确保方法和工具可靠性和实用性的关键步骤. 评估指标不仅帮助研究者和开发者理解现有方法的性能, 还能指导未来方法的改进和创新.

1.4.1 常用评估指标

设计模式检测技术效果的评估指标主要有:

(1) 准确率 (*Accuracy*): 是指判断正确的样本占总样本的比例. 计算公式为

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

其中, *TP*、*TN*、*FP* 和 *FN* 分别表示真阳性、真阴性、假阳性、假阴性.

(2) 精确率 (*Precision*) : 是指正确识别为特定设计模式的实例占所有被识别为该模式实例的比例. 精确率高表示误报率低. 计算公式为

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

(3) 召回率 (*Recall*): 也称真阳性率 (true positive rate), 是指正确识别为特定设计模式的实例占所有实际为该模式实例的比例. 召回率高表示遗漏率低. 计算公式为

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

(4) F_1 分数 (F_1-score): 是准确率和召回率的调和平均, 用于衡量准确率和召回率之间的平衡[82]. 当精确率和召回率都高时, F_1 分数也会高. 计算公式为

$$F_1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$

此外, 真阴性率 (true negative rate) 指在所有实际为负类 (非特定设计模式实例) 的样本中正确识别为负类的比例.

在文献 [53] 中, 称精确率为正确性 (correctness), 称召回率为完整性 (completeness), 此外还定义了有效性 (effectiveness) 来衡量过滤错误命中的有效性, 定义了可靠性 (reliability) 评估系统对错误命中预测的准确性.

1.4.2 其他评估指标

除了基本的准确率、精确率、召回率和 F_1 分数之外, 评价设计模式检测技术还可以考虑以下指标.

(1) 鲁棒性 (*robustness*): 评估检测方法对代码变异和不同编程风格的适应能力.

(2) 效率 (*efficiency*): 评估算法执行的时间和空间复杂度, 特别是在大规模软件系统上的执行速度和内存使用情况.

(3) 可扩展性 (*scalability*): 评估检测技术在处理不同规模、复杂度或多样性的软件项目时的效能和效率. 这个指标反映了一个系统在面对扩展需求时能否保持或提升其性能的能力, 特别是在处理大型代码库、多种编程语言或频繁变更的代码时的适应性.

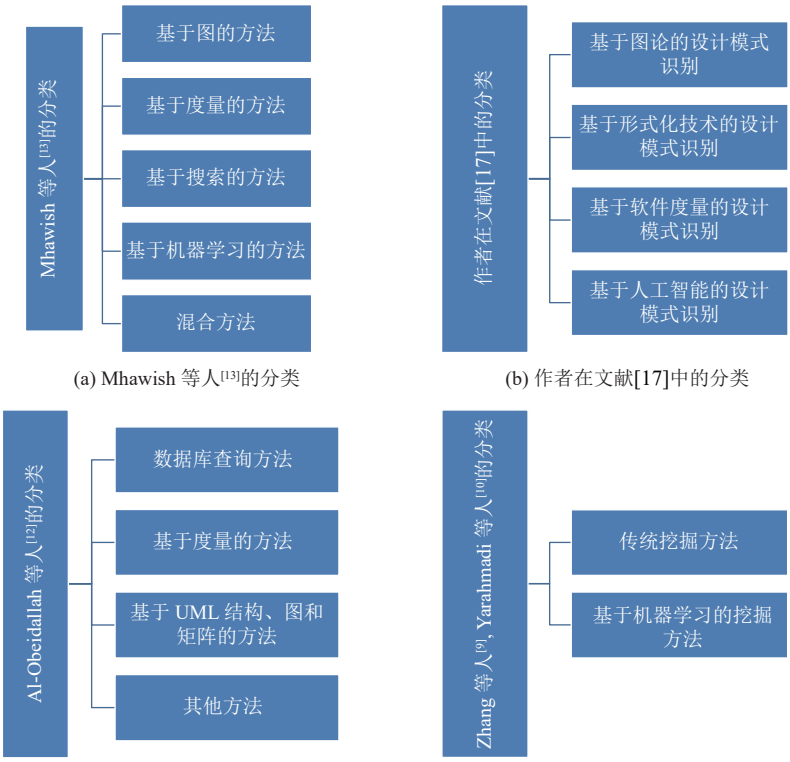
(4) 泛化能力 (*generalization*): 评估方法在不同的编程语言、不同的软件系统、不同的数据集或不同的模式类型中的适用性.

2 设计模式检测技术分类方法

研究人员根据所使用的技术、所考虑的特征类型、分析类型等不同标准, 对设计模式检测技术进行了分类, 而本文根据设计模式检测领域发展的时间线进行分类, 在该领域尚属首次.

2.1 根据所使用的技术分类

根据所使用的技术对设计模式检测技术进行分类如图 3 所示.



(a) Mhawish 等人[13]的分类 (b) 作者在文献[17]中的分类

(c) Al-Obeidallah 等人[12]的分类 (d) Zhang 等人[9], Yarahmadi 等人[10]的分类

图 3 根据所使用的技术进行分类

Mhawish 等人[13]将设计模式检测技术分为基于图的方法 (graph-based approaches)、基于度量的方法 (metric-based approaches)、基于搜索的方法 (search-based approaches)、基于机器学习的方法 (machine learning-based approaches) 和混合方法 (hybrid approaches). 作者在文献 [17] 中将设计模式检测方法分为基于图论的设计模式识别、基于形式化技术的设计模式识别、基于软件度量的设计模式识别、基于人工智能的设计模式识别. 设计模式检测技术会使用到诸如图论、形式化方法、软件度量、XML、AST、机器学习、字符串匹配、知识表示与推理等各种各样的计算机技术, 显然上述两种分类方法很难覆盖所有的设计模式检测技术.

Al-Obeidallah 等人[12]分别根据所使用的技术和所考虑的特征类型进行了分类. 根据所使用的技术, 将现有的设计模式检测方法分为数据库查询方法, 基于度量的方法, 基于 UML 结构、图和矩阵的方法, 以及其他方法. 他们将数据库查询, 度量, UML 结构、图和矩阵以外的所有方法都归类为其他方法. 虽然这种分类方式表面上可以覆盖所有的技术, 但数据库查询, 度量, UML 结构、图和矩阵并不具有非常强的代表性, 仅占设计模式检测技术的一小部分.

上述这些分类方法并没有充分考虑到设计模式检测技术的发展演进, 在体现发展历程和最新趋势方面略显不足. Zhang 等人[9]、Yarahmadi 等人[10]的分类与本文有部分重叠, 将当前设计模式检测技术分为传统挖掘方法和基于机器学习的挖掘方法. 但他们没有考虑机器学习从传统的机器学习算法到深层神经网络、图神经网络乃至如今最前沿的预训练语言模型的发展变革.

2.2 根据所考虑的特征类型分类

如图 4 所示, Chaturvedi 等人[11]、Al-Obeidallah 等人[12]、Rasool 等人[14]、Dong 等人[15]根据所考虑的特征类型将设计模式挖掘方法分为:

(1) 结构分析方法: 基于研究对象 (源代码、设计模型等) 的静态结构来检测设计模式实例.

(2) 行为分析方法: 考虑程序的执行行为.

(3) 语义分析方法: 补充结构和行为分析, 减少假阳性的数量. 例如使用命名约定和注释检索模式的角色 (role) 信息.

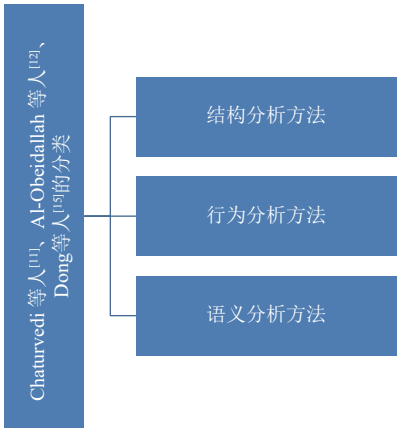


图 4 根据所考虑的特征类型进行分类

Al-Obeidallah 等人[12]的行为分析方法包括使用静态和动态分析技术提取程序的静态行为和动态行为两方面的信息, 而 Chaturvedi 等人[11]、 Rasool 等人[14]和 Dong 等人[15]所讨论的行为分析主要指的是软件运行时的动态行为.

与根据所使用的技术分类相比, 这种分类方式更为简洁明了, 且能包含绝大多数设计模式检测技术. 但根据作者在第 1.3 节中的讨论, 设计模式检测所考虑的特征除结构特征、行为特征和语义特征外, 还有非常重要的度量特征, 以及作为补充的其他特征, 所以这种分类方法也不够全面. 而且一种设计模式检测技术往往考虑不止一类特征, 很难将某一种设计模式检测技术绝对地归类于某一类. 此外与上述根据所使用的技术分类方法一样, 这种分类方式也难以体现出该领域的发展历程和最新趋势.

2.3 根据分析类型分类

从源代码中检测设计模式, 首先需要进行源代码分析来提取程序相关信息, 主要包括静态分析 (static analysis) 和动态分析 (dynamic analysis) 两种策略, 以及结合静态分析和动态分析优点的混合分析 (hybrid analysis). 如图 5 所示, Priya[16]根据分析类型将设计模式检测技术分为以下 3 类[83]:

(1) 静态方法 (static approach): 基于面向对象系统的源代码来提取设计信息. 这类方法主要利用代码结构本身, 不涉及程序执行时的动态数据.

(2) 动态方法 (dynamic approach): 侧重于实际执行源代码获取执行跟踪数据来检测设计模式.

(3) 混合方法 (hybrid approach): 结合动态信息和静态数据的分析, 从而提供一个更全面的系统分析. 这类方法试图通过结合静态结构和运行时行为来获得更准确的模式检测结果.

这种分类方法涉及到待检测系统的静态分析和动态分析以及静态分析和动态分析相结合, 可以涵盖所有源代码设计模式检测技术. 不过, 这种分类方法不适用于输入是系统设计级的文献, 而且同样无法体现该领域的发展历程和最新趋势.

2.4 根据设计模式检测技术发展的时间线分类

早期的设计模式检测技术主要是非机器学习的方法, 而后有大量的学者将机器学习技术引入到了设计模式的检测中来. 近几年预训练语言模型的出现为设计模式检测问题的解决提供了一个新的途径. 本文通过将设计模式检测技术按时间线进行分类, 展示了该领域从非机器学习到机器学习再到基于预训练语言模型的演进历程. 其中, 有监督学习方法又从传统有监督学习算法到当前最新的深度学习和图神经网络算法展开讨论. 如图 6 所示.

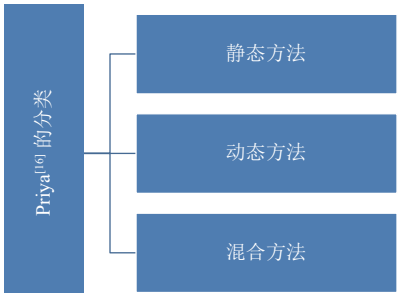


图 5　根据分析类型进行分类

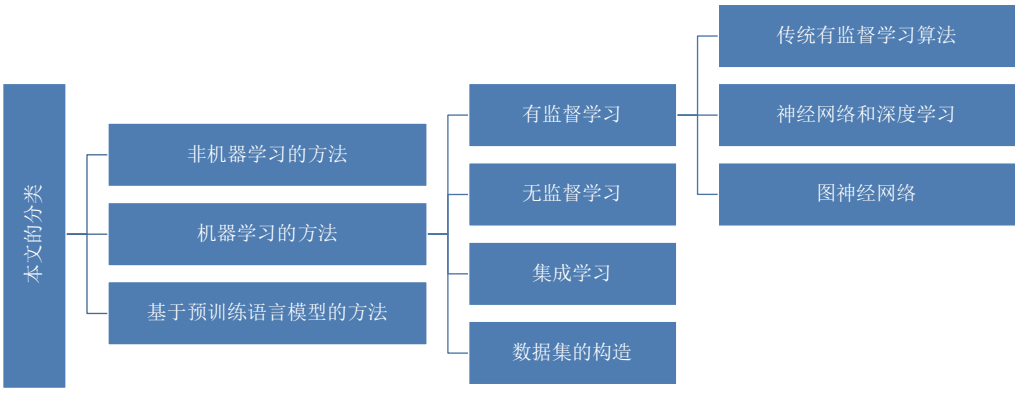


图 6　本文根据设计模式检测技术发展的时间线分类

时间线分类方法可以涵盖所有的设计模式检测技术, 无论是以源代码作为输入对象还是系统设计作为输入对象. 最重要的是, 本文方法清晰地展示了该领域的演进过程和趋势, 不仅揭示了技术的发展脉络, 强调了新技术的连续性和创新, 而且突出了当前最新发展趋势. 特别地, 本文强调了基于预训练语言模型的前沿技术, 展现了最前沿的研究动向, 同时也为未来的研究方向提供了指导.

3　软件设计模式检测技术研究现状

根据上一节的讨论, 本节将从非机器学习的设计模式检测、机器学习的设计模式检测和基于预训练语言模型的设计模式检测 3 大类方法的角度, 介绍和总结当前软件设计模式检测技术研究现状, 并给出这些技术在 JHotDraw、JRefactory 和 JUnit 这 3 个实际项目上的应用案例.

3.1　非机器学习设计模式检测方法

非机器学习的设计模式检测方法一般分为两个阶段: 检测规则获取阶段和模式匹配阶段. 在检测规则获取阶段, 人工从设计模式的理论描述中提取检测规则并提前存储起来. 模式匹配阶段的一般流程是 [84,85]: 通过静态分析或动态分析等方式, 从源代码或系统设计中提取待检测系统的静态结构、静态行为与动态行为、面向对象度量等相关信息; 根据提取到的系统信息, 将系统转换为有向图、AST、XML 等方便匹配和处理的语义形式; 将系统拆分为更小的待检测单元, 就可以根据检测规则将每个待检测单元依次与提前存储的模板设计模式进行匹配. 如果一个待检测单元与模板设计模式成功匹配, 那么该单元可被视为该设计模式的 (候选) 实例; 若匹配失败, 则将该单元排除. 非机器学习的设计模式检测方法的整体框架如后文图 7 所示.

3.1.1　有向图

将实体 (类/接口/对象) 用有向图的顶点来表示, 实体之间的关系等特征用顶点之间的边来表示, 就可以把设计模式检测问题转换为图论相关问题.

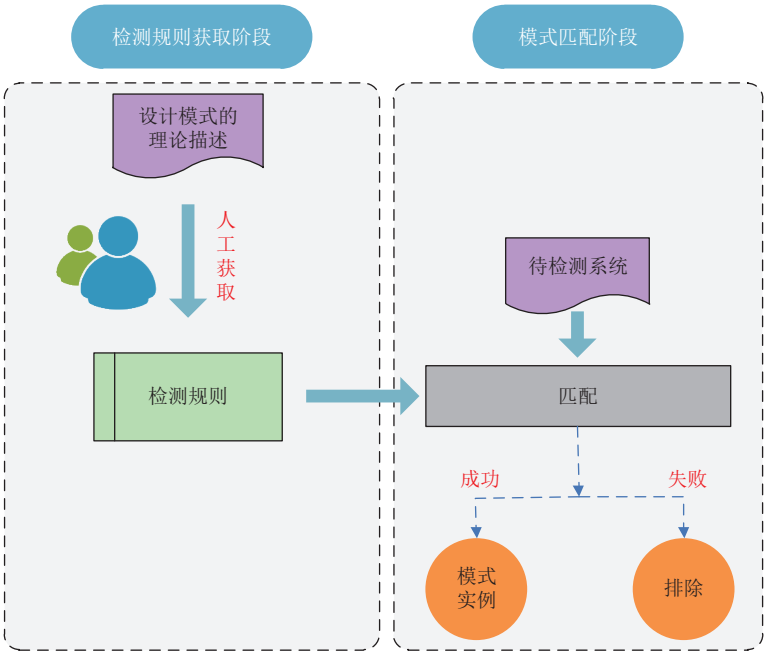


图 7 非机器学习的设计模式检测方法的整体框架

文献 [23–29] 将系统和设计模式以有向图的形式呈现, 通过图同构判定算法在系统图中寻找模式子图. 图同构算法具有较好的鲁棒性和很高的匹配精度, 但只能在系统图中找到与模板设计模式结构完全相同或仅有轻微变化的模式实例, 因此在面对系统中的设计模式变体时往往会失效, 这可能会导致精确率高而召回率过低的问题. 其中许涵斌等人[28]从类的属性、类间关系和整个模型这 3 个层次在系统图中寻找模式子图, 可以进一步提高精确率和真阴性率 (true negative rate). 但是因为匹配的条件更为苛刻, 更加限制了方法的召回率.

Dong 等人[30,31]将泛化关系、关联关系、抽象类、调用关系、聚合关系、组合关系、依赖关系和实现关系这 8 种设计特征信息编码到有向图/矩阵并将 8 个矩阵组合成一个矩阵, 使用模板匹配算法计算系统和设计模式之间的互相关值 (cross correlation value) 来寻找系统中存在的模式实例. 归一化互相关值越大, 表示设计模式与系统的匹配程度越高. 该方法可以在待考查系统中找到与模板设计模式结构不完全相同但相似度较高的模式实例. Tsantalis 等人[32]将源代码和设计模式的关联、泛化、抽象类、对象创建、抽象方法调用等信息均表示为一个单独的有向图/矩阵, 使用相似度评分算法计算各子系统与设计模式之间的相似度矩阵来寻找子系统中的模式实例. 计算得到的相似度矩阵中的元素代表两个有向图/矩阵各顶点之间的相似度得分, 因此该方法不仅可以搜索到设计模式实例, 还可以根据相似度矩阵得到实例中的类与设计模式中的角色之间的对应关系. Dewangan 等人[33,34]将 UML 类图转换为图的形式, 其中节点表示类图中的类, 边表示类之间关系. 此外, 每个节点有一些作为标签的特征, 由三元组表示, 3 个分量分别代表节点父类的个数, 节点子类的个数和节点的协作类 (collaborators) 的个数. 他们使用余弦相似度衡量待检测系统有向图和设计模式有向图的所有节点之间的相似度, 根据相似度的值的大小来判定匹配程度. 上述文献 [30−34] 没有采用精确匹配的方式, 可以检测到模式变体, 在一定程度上提升了召回率.

3.1.2 形式化方法

有向图的语义较为简单, 表达能力有限, 不太容易表达复杂的特征 (例如类的内部行为、方法的具体实现、类之间的动态交互等). 而且, 有向图更适用于描述软件系统的静态结构, 对于动态行为特征, 如运行时的调用链、方法之间的交互顺序、事件的响应与处理等, 表现能力相对有限. 此外, 图论相关算法 (如同构检测、图匹配算法) 因为组合爆炸问题、矩阵运算的复杂性、递归和回溯算法等原因可能需要较高的计算资源, 尤其在处理大型软件

系统时, 计算复杂度可能成为性能瓶颈. 形式化方法基于严格定义的数学概念和语言, 语义清晰、无歧义, 可以对软件系统进行严格的检查和分析, 通过数学的推理和证明验证软件系统是否满足某种特定性质或具有某种行为特征. 有学者借助形式化方法进行设计模式的检测, 主要用于对行为型模式候选实例进行区分和过滤. 目前在设计模式检测领域应用较多的形式化方法是模型检测 (model checking) 和有限自动机 (finite automata).

Bernardi 等人[35]将设计模式的行为属性表示为选择性 μ 演算公式, 将候选实例 (Java 文件或字节码) 转换为时序规范语言 (language of temporal ordering specification, LOTOS) 模型, 使用模型检测工具 CADP 验证候选实例是否满足 μ 演算公式, 以此来判断候选实例是否为模式实例. 上述方法尽管可以验证系统在运行期间可能发生的情况, 但并没有实际执行源代码, 没有获取在运行期间实际发生的方法调用及对象引用的具体类型等动态信息, 这会在一定程度上限制其精确率和真阴性率. Lucia 等人[36]方法和 Bernardi 等人[35]方法类似, 先使用静态结构分析得到初步的模式实例[37], 然后使用模型检测技术对其中的行为型模式候选实例进行进一步筛选. 该方法将定义设计模式正确行为的属性编码为线性时态逻辑 (linear temporal logic, LTL) 公式, 将表示候选模式实例涉及到的对象之间可能交互轨迹的顺序图转换为 Promela (模型检测工具 SPIN 的输入语言) 代码, 并使用 SPIN 验证候选实例是否满足 LTL 公式. 与 Bernardi 等人方法不同的是, 在对系统进行模型检测后, 该方法通过动态行为分析对得到的模式实例进行再次筛选. 首先通过实际执行源代码获取方法调用跟踪, 然后使用一个解析器来验证该跟踪, 该解析器能够识别定义设计模式行为的方法调用序列. 该方法将模型检测得到的模式实例进行再次筛选和区分, 可以进一步提升精确率和真阴性率, 但同时也可能会限制召回率.

Wendehals 等人[38]将行为型模式的行为特征 (主要考虑函数调用) 转换为有限自动机的状态和它们之间的转移, 并引入一个没有传出转移 (outgoing transitions) 的拒绝状态. 对于行为型模式候选实例, 使用单元测试工具 JUnit 执行源代码并获取方法调用跟踪. 若有限自动机在处理某个候选实例的方法调用跟踪时到达拒绝状态, 该候选实例被排除. 该方法通过获取类参与者的真实行为并与设计模式的有限自动机进行匹配来判断行为型候选实例是否符合特定设计模式, 从而过滤掉假阳性实例, 提升行为型模式检测的精确率.

3.1.3 AST

有向图和形式化方法都不是专门用于描述源代码结构的技术, 表示源代码时可能会存在较多语义不对称的问题, 即表达的意义和源代码实际意义之间可能存在偏差. AST, 简称语法树, 是用编程语言编写的源代码的抽象语法结构的树表示. 树的每个节点表示源代码中出现的一个语法构造. AST 在设计模式的检测中发挥着重要作用, 通过其能够详细分析代码结构, 为精确的静态分析提供基础.

Al-Obeidallah 等人[39]通过解析 Java 源代码生成 AST, 记录类之间的继承、依赖、聚合、关联和实现等关系, 并将这些关系映射到一个源代码模型中. 然后, 利用一个结构搜索模型 (structural search model, SSM), 逐步检查这些关系, 匹配和构建设计模式的结构, 以检测并提取设计模式实例. Shi 等人[40]将 GoF 模式重新划分为更适合于逆向工程的 5 类: 语言提供的模式 (language-provided patterns)、结构驱动的模式 (structure-driven patterns)、行为驱动的模式 (behavior-driven patterns)、特定领域的模式 (domain-specific patterns) 和通用概念 (generic concepts). 他们通过对源代码进行静态分析来构建 AST, 从 AST 中提取类之间的继承关系、接口实现、方法调用和属性访问等关系进行结构驱动的模式 (structure-driven patterns) 的检测. 此外, 对于行为驱动的模式 (behavior-driven patterns), 他们通过 AST 构造生成控制流图 (control flow graph, CFG) 来进行静态行为分析, 表示程序在执行过程中可能的控制流路径, 以识别方法体内的关键行为.

通过 AST, 可以详细分析类之间的关系、方法调用、访问控制等, 这些都是判断设计模式存在的关键因素. 另外基于 AST 可以在不实际执行源代码的情况下对系统的方法调用和控制流等行为进行分析和识别, 使得行为型模式的检测更为容易便捷.

3.1.4 知识表示与推理

上述这些方法都需要编写较多的程序代码来实现相应的检测算法, 而且检测条件往往预先固定在程序代码中. 人工智能的知识表示与推理技术能够在不依赖传统编程的情况下, 描述复杂的知识结构和关系并进行自动化推理, 从而检测和区分设计模式. 目前在设计模式检测中使用的知识表示与推理技术有 Prolog、回答集编程

(answer set programming, ASP) 和本体 (ontology) 等.

作为设计模式检测领域最早的工作之一, Krämer 等人[19]将待识别程序转换为 Prolog 中的事实 (fact), 将模式表示为 Prolog 规则 (rule), 然后借助第三方工具 Visual Prolog 执行 Prolog 查询 (query) 来搜索模式. 受到该工作的启发, Hayashi 等人[41]将从待识别系统中抽取到的信息表示为 Prolog 中的事实, 执行定义为 Prolog 规则的检测条件, 来推断满足设计模式条件的类结构的存在. Hayashi 等人[41]使用的 Prolog 环境是 tuProlog. 类似地, Di Martino 等人[42]通过解析 UML 图的 XMI(XML metadata interchange, XML 元数据交换) 表示生成对应的 Prolog 事实, 定义一组 Prolog 规则来描述模式特征, 应用这些 Prolog 规则在生成的事实库上进行推理, 以识别出符合设计模式的 UML 图元素. Luitel 等人[43]将待检测系统的类图和序列图表示为 ASP 中的事实, 使用 ASP 求解器输出分别遵循描述结构模式和行为模式规则的结构和行为元素[86], 从而在模型层次上识别出设计模式实例. Thongrak 等人[44]分析设计模式模板并为每种设计模式定义一组用于匹配的 SQWRL (semantic query-enhanced web rule language, 语义查询增强 Web 规则语言) 规则. 他们生成 UML 类图的 OWL (Web ontology language, 网络本体语言) 编写的本体概念 (ontology concept), 基于设计模式的 SQWRL 规则在 OWL 本体中进行搜索, 从而发现 UML 类图中的模式实例. Thongrak 等人[44]只将 UML 类图作为输入, 没有考虑设计模式的行为特征, 因此难以区分具有相似结构的设计模式, 例如状态模式与策略模式, 命令模式与适配器模式. Panich 等人[45]扩展了 Thongrak 等人[44]的检测方案来处理设计模式的行为特征, 通过结合类图和序列图的本体模型以及相关推理规则, 实现设计模式的自动检测. Panich 等人[45]通过结合类图和序列图的结构和行为特性, 部分解决了仅通过结构特性无法区分结构相似设计模式的问题.

基于有向图、基于形式化方法、基于 AST 的设计模式检测技术都需要编写较多的程序代码来实现相应的检测算法, 而基于知识表示与推理的设计模式检测技术能够在不依赖传统编程的情况下, 通过结构和行为的知识模型及推理规则来检测和区分设计模式, 实现支撑工具更加方便快捷. 此外, 基于知识表示与推理的设计模式检测技术可以方便地向知识库中添加和修改模板设计模式的约束条件, 不需要把全部检测规则都体现在固定的程序中, 这使得这些技术及其支撑工具可以非常灵活方便地新增加支持的模式或者改进检测规则以取得更好的效果.

3.1.5 其 他

除上述有向图、形式化方法、AST、知识表示与推理外, XML[46]、字符串匹配[47]、遗传算法[48]等技术也在设计模式检测中得到了应用. XML 技术在设计模式检测中的应用通常涉及利用 XML 来描述软件的结构信息, 例如类和类之间的关系. 这种方法允许研究者通过 XML 模式匹配来识别设计模式的结构特征. 字符串匹配技术在设计模式检测中主要用于识别代码中的模式特征, 如命名规则和特定代码结构. 在设计模式检测中, 遗传算法可以用来优化设计模式的匹配过程, 比如调整和选择最合适的特征集合来识别特定的设计模式.

3.1.6 总 结

非机器学习的设计模式检测技术已经取得了一定的效果. 对于结构和行为比较简单的模式 (例如只有一个角色的单例模式), 具有很好的处理能力, 且检测过程的每一步都可以追踪和溯源, 具有非常强的可解释性. 然而, 这些方法的检测规则都是人工从设计模式的理论描述中提取的, 具有模糊性和主观性. 规则编写的工作量大, 不同的编程语言的规则可移植性差, 在不同的编程语言上的泛化能力差. 设计模式的使用非常灵活, 在实际的工程项目中往往会有各种各样的模式变体, 从设计模式的理论描述中得出的规则很难涵盖所有的变体类型. 此外, 还存在着模式实例中的多个类关联同一个模式角色或一个代码片段中包含同一种模式的多个实例等复杂情况, 而设计模式的理论描述难以考虑到全部情况.

3.2 机器学习的设计模式检测方法

基于机器学习的设计模式检测方法一般分为模型训练阶段和模型应用阶段. 模型训练阶段[84,85]从实际软件系统中构造模式实例数据集, 然后使用 KNN、SVM、ANN 等机器学习算法训练得到设计模式分类器模型并存储起来. 模型应用阶段的一般流程是: 首先输入待检测系统的设计模型或源代码; 然后提取设计模型或源代码信息并将

其划分为更小的待检测单元; 最后使用训练好的设计模式分类器模型对每个待检测单元依次进行分类并对分类结果做进一步处理后输出最终的设计模式识别结果. 机器学习的设计模式检测方法的整体框架如图 8 所示.

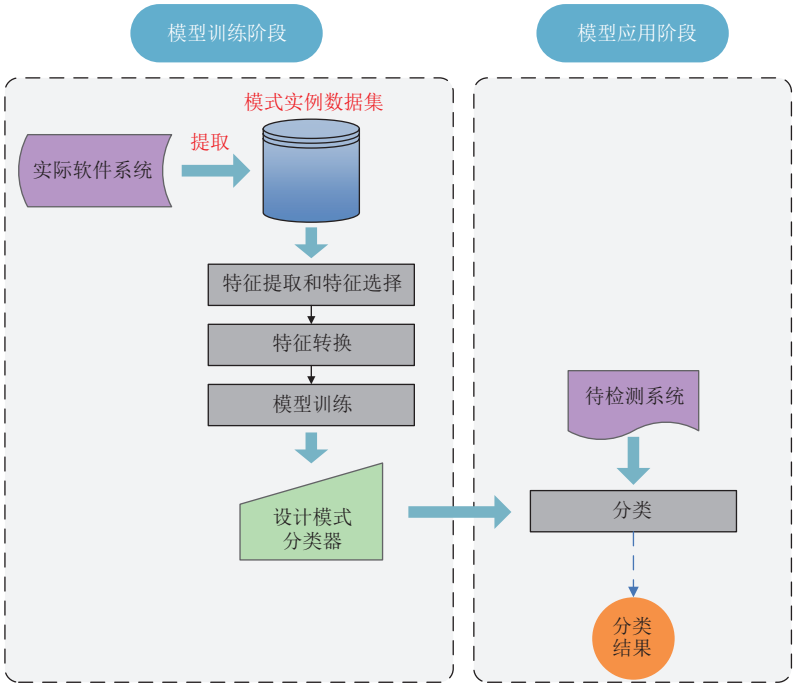


图 8 机器学习的设计模式检测方法的整体框架

3.2.1 有监督学习

设计模式检测本质上是一个类别已知的分类问题, 因此多采用有监督学习方法.

研究人员已经构建并训练了基于决策树[29,52−54,62,63,67,71], KNN[29,71], SVM[29,51,55,56,63,70,71], 关联分类[60]等传统有监督学习算法的设计模式二分类器 (预测是否属于某种模式) 或多分类器 (预测属于多种模式的哪一种). 鲁润泽等人[29]针对每种设计模式, 分别使用决策树、KNN 和 SVM 这 3 种分类方法训练模型. 他们将采用子图匹配方法得到的候选模式实例进行向量化并将得到的向量分别输入到构造的多个设计模式分类器, 由设计模式分类器判断候选模式实例是否属于某种设计模式. Chihada 等人[55]计算设计模式实例中类的度量值, 将同一角色的所有类的度量值进行聚合得到相同维度的特征向量, 使用 SVM 训练适配器、生成器 (builder)、组合、工厂方法、迭代器 (iterator) 和观察者模式的分类器. 他们将预处理阶段生成的候选类的组合发送给每种模式的分类器, 通过每个分类器得到的置信值 (confidence value) 判断该组合是否是某种模式的实例. Barbudo 等人[59]构建了 15 种设计模式的关联规则分类器. 他们使用语法引导遗传编程 (grammar-guided genetic programming, G3P) 从代码库中挖掘类关联规则 (class association rule, CAR), 并应用剪枝方法和几种分类策略来选择具有最佳检测能力的规则构建分类器. 这些传统有监督学习算法相对于深度学习, 参数相对较少, 对于数据集的规模的要求较低, 模型训练所需要的硬件配置的要求较低[87]. 但这些算法的效果在很大程度上依赖于选取的特征向量. 对于设计模式检测问题来说, 找出最合适、最有效的特征是一个非常困难的任务. 如果特征选择不佳, 算法性能可能较差或不稳定. 线性回归 (linear regression)、多项式回归 (polynomial regression) 和逻辑回归 (logistic regression)[50,56,70,71]的模型结构较为简单, 表达能力和拟合复杂关系能力不足. 因此, 这 3 种算法在文献中主要被用作基准模型, 与 SVM、ANN 等相对更复杂的算法进行对比, 展示后者在设计模式检测任务中的性能优势.

随着各种硬件基础设施的大力发展, 深度学习的研究得以展开并迅猛崛起. 人工神经网络 (artificial neural network, ANN), 尤其是卷积神经网络 (convolutional neural network, CNN)、层次循环神经网络 (layer recurrent

neural network, LRNN) 等深层神经网络逐渐应用于设计模式检测问题中[50−54,56−59,66,71]. Dwivedi 等人[50]收集源代码中的面向对象度量数据, 经过特征处理后形成数值向量, 使用逻辑回归和 ANN 模型进行训练, 以识别设计模式. Ferenc 等人[53]从源代码中提取公共方法调用 (public method calls)、非公共方法调用 (non-public method calls)、构造函数参数 (constructor parameters)、新增方法 (new methods)、继承关系 (inheritance)、算法特征 (algorithm features)、具体策略 (concrete strategy)、上下文参数 (context parameters) 等信息, 这些信息被称为预测因子 (predictors). 他们在手动标记的数据集上使用预测因子, 基于决策树和 ANN 分别训练了对象适配器模式和策略模式的决策模型. 他们将机器学习得到的模型文件集成到前期研究给出的基于模式匹配的系统[88], 以此过滤掉候选实例中的阴性命中. Alhusain 等人[57]方法分为两个阶段: 第 1 阶段针对每种模式的每个角色训练一个单标签 ANN, 为每种模式的每个角色识别一组候选类来减少搜索空间; 第 2 阶段为每种模式训练一个单独的 ANN 分类器, 检查所有相关角色的候选类的可能组合来识别模式. Uchiyama 等人[58,59]考虑了单例、适配器、模板方法、状态和策略 5 种模式和这些模式包含的 12 种角色. 他们将应用模式的程序输入到度量测量系统中来获得每个模式角色的度量值, 并将这些测量值输入到 ANN 中进行学习. 他们通过相同的方式获得待检测程序的每个类的测量值并使用训练好的模式角色 ANN 分类器识别候选角色, 然后将候选角色输入到使用模式结构定义的模式检测系统中获取最终的模式实例. 这些文献使用的 ANN 主要是全连接神经网络 (fully connected neural network, FCNN). FCNN 在层数较浅时无法学习到复杂的特征和检测规则. 通过构建多层神经网络, 可以将原始数据转换为更具表达性的特征, 但需要大量的高质量标注数据才能取得好的效果. 否则, 在层次较多时很有可能会产生非常严重的过拟合问题. 同时, 层次较多时参数规模过大导致计算效率低下, 模型训练对硬件配置的要求也更高, 所耗费的时间也更长. 因此上述文献在较为简单的模式如单例模式上效果较好, 但不适用于复杂的设计模式.

Thaller 等人[66]从源代码中提取描述类之间的结构或行为关系的高层概念 (称为微结构 (micro-structures)), 生成特征图 (feature maps), 其中, 行表示微结构, 列表示设计模式角色. 然后, 将其作为输入提供给 CNN. CNN 通过多层卷积和池化操作, 从特征图中提取和组合特征, 最终输出属于某种设计模式的概率. Thaller 等人[66]利用 CNN 的参数共享、局部连接和下采样三大优势在一定程度上解决了上述 FCNN 应用于设计模式检测问题时的不足. CNN 主要擅长处理具有空间结构的例如图像形式的数据, 而 Thaller 等人[66]构造的特征图虽然是一个二维矩阵, 但并非传统意义上的图像, 本身并不具备空间结构, 因此并未充分发挥 CNN 具备的捕捉和识别数据中局部特征和空间模式的强大能力. 文献 [52] 在文献 [49] 的基础上, 进一步应用决策树和 LRNN 算法进行模式检测. Latif 等人[54]也做了与文献 [52] 类似的工作. 与文献 [66] 研究面临相同的问题, RNN 主要擅长处理时序数据, 文献 [52, 54] 提取的源代码面向对象度量本身不具有时序性, 使用 RNN 的意义不大. 通过文献中给出的实验结果可以看到, 文献 [52] 的 LRNN 分类器检测效果与文献 [50] 的全连接神经网络分类器相比差别不大.

文献 [23–34] 使用有向图数据形式检测设计模式, 这些方法依赖于预先定义的计算规则和特征提取方法, 容易受到预定义规则的限制, 可能导致较高的误报率, 特别是在处理包含复杂行为约束的设计模式时表现不佳. 在效率方面, 这些方法计算复杂度高, 匹配过程耗时. Ardimento 等人[61]提出的方法使用图神经网络 (graph neural network, GNN) 技术, 将面向对象软件系统和设计模式表示为图, 利用 GNN 生成这些图的嵌入 (embedding) 表示, 这些嵌入表示保留了图的结构和节点之间的关系. 接着, 他们通过计算系统图的嵌入和设计模式图的嵌入的相似度, 在嵌入空间中进行高效的子图匹配, 从而实现设计模式的检测. GNN 使得复杂的图结构能够在嵌入空间中进行处理和比较, 更高效地处理图数据的复杂性, 并能学习节点之间的深层次关系, 从而提高设计模式检测的效率和准确性.

总体来说, 基于决策树和 KNN 的方法在某些特征、某些模式或某些数据集上效果好, 但是性能不稳定, 而基于 SVM 和 ANN 的方法通常表现稳定. 决策树和 KNN 在特定场景下表现出色, 但容易受训练数据变化、特征选择和参数调整等因素的影响, 性能表现出不稳定性. 例如, 决策树可能对数据的细微变化敏感, 导致不同训练集下树结构变化明显; KNN 的表现则可能取决于 k 值的选择和样本距离度量方式. SVM 和 ANN 通常表现稳定, 部分原因在于它们的算法能够自适应调整参数, 并且能够适应多种特征和模式. 因此, 在设计模式检测中, 它们往往表现出较好的泛化能力和稳定性.

3.2.2 无监督学习

Dong 等人[62]通过矩阵转换对训练样本进行聚类从而将复合记录 (类) 简化为一些基本模型, 显著减少了训练样本的规模. 然后, 他们基于聚类后的训练样例建立决策树, 使其能够高效地用于决策树分类, 训练样本分类的计算复杂度大大降低. Zanoni 等人[63]使用 k-Means 和 CLOPE [89]两种聚类算法将数据集进行聚类, 从而将样本中的每个模式实例表示为一个维度统一的特征向量, 然后训练朴素贝叶斯、规则归纳学习、随机森林 (random forest)、决策树、SVM 等多个有监督分类器. 他们使用这些分类器对基于图匹配技术提取的模式候选实例进行分类. Chaturvedi 等人[56]使用主成分分析 (principal component analysis, PCA) 技术对提取的代码特征进行降维, 以降低数据维度并突出最重要的变量, 从而为机器学习模型提供更精简和有效的输入. 他们将处理后的数据集应用于不同的机器学习算法, 包括线性回归、多项式回归、SVM 和 ANN, 进行模型训练和比较分析. 以上 3 种无监督学习的设计模式检测方法主要使用聚类或 PCA 算法进行特征转换, 而不是直接从系统中检测设计模式.

Chand 等人[64]训练 Word2Vec[90]和 Code2Vec[91]模型来理解代码的语义和上下文信息, 采用 k-Means 聚类方法对 Word2Vec 和 Code2Vec 模型提取的嵌入向量进行聚类, 将聚类结果与已经明确定义的设计模式进行比对来确定每个聚类对应哪种模式. Chand 等人[64]方法完全基于无监督的聚类算法检测设计模式, 无需标注的设计模式实例.

Wang 等人[65]使用无监督的关联分析算法分析历史架构设计数据来检测系统中的架构模式. 他们从系统的历史设计数据和架构设计数据中收集大量数据, 包括系统的各个视图产品的描述、事件日志等, 然后使用关联分析算法 (如 Apriori 算法或 FP-Growth 算法) 从预处理后的数据中挖掘出频繁项集和关联规则, 这些规则揭示了系统架构元素之间的关联关系和设计模式中的隐含规律. 基于这些关联规则, 可以对新的或现有的系统进行设计模式检测, 识别出系统中的设计模式实例.

3.2.3 集成学习

集成学习通过结合多个模型的预测结果来提高整体性能, 能够提升模型的准确率和鲁棒性, 因此不少研究者采用集成学习方法进行设计模式的检测. 设计模式检测中应用的集成学习算法主要有随机森林、堆叠 (stacking) 和提升 (boosting) 几种算法.

Dwivedi 等人[51]通过从源代码中提取特征, 并利用这些特征训练 ANN、SVM 和随机森林这 3 种机器学习模型, 模型能够有效区分和识别不同的设计模式实例. Thaller 等人[66]除使用 CNN 外, 还将特征图 (feature maps) 展平为一维向量, 作为输入提供给随机森林模型训练每种模式的二分类模型. 随机森林通过多棵决策树的集成学习, 从特征图中提取和组合特征, 最终输出属于某种设计模式的概率. Nazar 等人[69]应用 Word2Vec 算法构造 Java 嵌入模型来捕获源代码中单词 (word) 的语义以及单词之间的相关性, 将构造的词向量输入随机森林集成分类器来检测设计模式. Hamama[70]工作与 Nazar 等人[69]类似, 不同的是, Hamama[70]应用 Doc2Vec 算法构造 Java 嵌入模型, 取得了比 Nazar 等人[69]更好的检测准确率. Doc2Vec 在设计模式检测中比 Word2Vec 表现更好是因为它能够更全面地捕捉代码的语法和语义信息, 从而生成更具代表性的向量表示, 这使得训练出来的机器学习模型能够更准确地识别和分类设计模式实例. 随机森林在处理简单模式时通常表现良好, 并在一定程度上可以缓解数据不平衡问题, 但在处理复杂模式时效果有所下降, 因为作为随机森林构成单元的决策树不太擅长描述复杂特征.

冯铁等人[68]针对每种设计模式分别用度量特征和微结构特征训练出一个分类器, 然后采用模型堆叠 (stacking) 的方式训练出最终的分类器. Stacking 方法在设计模式检测中比随机森林具有更高的灵活性和综合性. 它通过结合多个基础模型 (如决策树、SVM、KNN 等) 的预测结果, 利用不同模型的优势捕捉数据的复杂特征, 从而提高整体预测的准确性和鲁棒性, 尤其在处理复杂关系时表现出色. 此外, Stacking 方法的多样性和适应性使其在应对变体识别、行为型设计模式检测及解决组合爆炸问题上具有显著优势.

Mhawish 等人[67]使用梯度提升树 (gradient boosted trees, GBT) 算法等树形机器学习算法结合软件度量训练模式角色的分类器, 从而区分和检测具有相似结构的设计模式. 他们通过特征选择和超参数调整提高模型性能. Komolov 等人[71]从 GitHub 获取使用 MVP 和 MVVM 架构的开源项目数据并提取源代码度量, 将这些度量作为输入特征来训练机器学习模型. 他们应用了 CatBoost 和可解释提升机 (explainable boosting machine, EBM) 等 9 种不

同的机器学习方法从源代码度量中检测软件设计模式. 相比于随机森林和 Stacking 方法, Boosting 方法在设计模式检测中的主要优势在于其高精度和对数据不平衡问题的良好处理能力. 通过逐步调整和优化分类器, Boosting 方法能够显著提高检测复杂设计模式的准确性, 并且在处理样本数量不均衡的问题时表现更为优越.

集成学习结合多个模型的预测结果, 从而能够有效地处理设计模式实现的多样性和复杂性. 这种方法能降低对单一模型过度依赖的风险, 减少过拟合, 并通过多个模型的综合判断抵御噪声, 从而提供更可靠、更广泛适用的设计模式检测解决方案. 集成学习在设计模式检测中通常能够提高预测性能和模型的鲁棒性, 但在集成的基础模型选择不当、参数调优不足、数据不平衡严重 (尽管 Boosting 方法在处理数据不平衡问题上具有一定优势) 等情况下, 可能会导致模型性能下降.

3.2.4 数据集的构造

机器学习的设计模式检测方法需要含有大量带标注设计模式实例的数据集.

有学者试图通过人工判断和标记的方式构建数据集. Ferenc 等人[53]首先使用基于模式匹配的设计模式识别系统[88]在开源项目 StarWriter 中找到 84 个适配器对象候选实例和 42 个策略模式候选实例, 然后对找到的实例相对应的源代码进行了手动检查, 将候选实例标记为真命中或假命中. 他们将这些手动标记的实例用作学习系统的训练集. Chaturvedi 等人[56]使用形式为 Java 项目的 71 个面向对象程序来创建数据集. Uchiyama 等人[58,59]手动判断了一组应用了模式的程序作为训练数据, 其中包括 60 个小规模程序 (small-scale programs) 和 158 个大规模程序 (large-scale programs). 小规模程序来自于一些小的用于实验的程序, 而大规模程序实例主要来自于实际使用的 Java 库、JUnit 和 Spring Framework 等开源项目. Nazar 等人[69]发现公开数据集 P-MARt 中设计模式的数量不均衡. 因此, 他们基于 GitHub Java 语料库 (GitHub Java corpus, GJC) 构建了一个新的语料库, 称为 DPD_F-Corpus, 并由设计模式方面的专业人员对语料库中的每个文件标记相应的模式. Hamama[70]通过下载并分析丢失的项目和文件、删除“.java”文件之外的所有文件、删除可能导致歧义的重复文件及双标签文件等方式扩展和改进了 DPD_F-Corpus[69], 并作为他们的数据集来源, 包含了来自 216 个项目的约 1 393 个 Java 文件. 他们通过众包 (crowdsourcing) 的方式让经验丰富的开发者对这些文件进行标注. 人工判断和标记的方式构建数据集需要耗费大量的时间和人力, 因此构建的数据集往往规模不大. 例如, Ferenc 等人[53]只标记了一个大型 C++系统 StarWriter, 构建的数据集只包含 84 个适配器对象实例和 42 个策略模式实例, 数据集的规模非常小. 另外, 这种方式需要构建者必须具备足够的设计模式知识, 而且判断和标记结果可能会因为标记人员的个人水平等因素出现错误, 导致数据集中引入一些标签噪声样本.

目前有机构和学者提供了一些公开的设计模式实例库, 例如 P-MARt [92], DPB [93], Percerons [94]和 Java-DPD-dataset [95]. 纯人工判断和标记的方式近几年已较少采用, 很多文献基于这些公开的模式实例库构造数据集. 文献 [29, 50, 51, 66, 67] 从专家手动标注的设计模式实例库 P-MARt 中提取设计模式原始实例. 文献 [29] 基于 P-MARt 包含的 9 个软件中的 6 个构建训练数据集, 其余 3 个构建测试数据集. 首先采用子图匹配的方法从这些软件中提取候选设计模式原始实例, 然后与 P-MARt 中的实例进行比对, 将比对成功的实例标上阳性标签, 比对失败的实例标上阴性标签. 这样做的目的是使得阴性标签的样本更具代表性, 因为这些比对失败的样本可能与某种模式具有相似之处, 容易被误判, 用这种样本训练模型可以增强模型的区分能力. Zanoni 等人[63]从 1 个网络上的项目以及 9 个 P-MARt 中的项目中收集设计模式实例. 冯铁等人[68]通过抽取 Zanoni 等人[63]的训练样本和网络搜集的一些训练样本构造正负模式样本集. Barbudo 等人[59]从 P-MARt 和 DPB 两个实例库中提取实例来构建数据集. 相对于完全的人工判断和标记方式, 这种方式耗费的时间和人力大大减少. 但是现成的模式库中的模式实例及支持的模式种类是有限且固定的 (例如 DBP 仅支持 5 种模式). 此外, 若将设计模式检测问题看成有监督二分类问题 (即针对每种模式使用有监督机器学习算法训练分类器, 分类器输出“属于该种模式”和“不属于该种模式”两类结果), 则每种模式需要正样本和负样本. DPB [93]提供了正负样本, 而 P-MARt [92]、Percerons [94]等实例库仅提供了正样本.

还有学者试图借助现有的设计模式检测工具来构建数据集. Chihada 等人[55]利用现有的设计模式检测工具从源代码中提取模式实例. 这些工具包括相似性评分算法[32]和文献 [63]. 文献 [57] 使用文献 [63] 等现有的设计模式检测工具对多个开源应用程序进行分析并投票, 通过票数结果来决定阳性和阴性样本. 当某个工具挖掘到一个设

计模式实例时, 就会为该实例投上一票. 最终得票数大于 1 的作为阳性样本, 得票数等于 1 的作为阴性样本. 这样做的目的是让阴性样本的数量不至于太大且具有代表性. 然而, 因忽略了得票数为 0 的实例, 导致模型对错误实例的学习不充分. 这种方式无需人工标记数据集, 避免了手动判断和标记的繁琐过程, 大幅节省了人力和时间成本. 它可以自动化分析多个开源应用程序, 从而迅速构建规模较大的数据集, 且可以方便地根据需要增加新的样本. 但是目前还没有一个公认的基准测试来验证这些工具[96], 而设计模式检测工具本身会有误判, 导致构建的样本集中标签噪声样本比例可能会比较大. Alhusain 等人[57]通过投票的方式一定程度上缓解了这个问题, 但他们简单地将投票数较少的样本 (一般取投票数为 1) 作为负样本, 可能会导致负样本中出现更多的假阴性实例. 另外这种方式受制于所选工具支持的设计模式种类, 导致数据集中包含的设计模式实例种类相对固定. 对于某些未被工具支持的设计模式, 这种方法难以获得相应的样本.

3.2.5 总 结

与非机器学习的设计模式检测技术相比, 机器学习的设计模式检测通过学习实际工程项目中实现的模式实例来提取检测规则, 无需人工编写检测规则, 也无需对每种设计模式都有深入理解和认识. 机器学习的设计模式检测技术降低了对设计模式掌握程度的门槛, 大大提升了检测准确率, 对于设计模式变体也有很好的检测效果, 目前已经成为该领域的主流方向. 但模型的性能建立在质量高规模大的设计模式实例数据集的基础之上, 尤其是深层神经网络对数据规模的要求更高. 根据第 3.2.4 节可知, 目前学术界和工程界缺乏针对设计模式检测问题的公开高质量大规模数据集. 当前文献自行构建的数据集质量参差不齐, 规模往往也比较小, 给机器学习的设计模式检测技术的模型训练、对比和评估造成困难. 此外, 决策树、KNN、SVM、关联分类模型具有较好的可解释性, 而神经网络构建的模型可能缺乏可解释性, 当达不到预期检测效果时难以追踪和溯源.

3.3 基于预训练语言模型的设计模式检测方法

基于预训练语言模型的设计模式检测方法也可以分为两个阶段: 模型准备阶段和模型应用阶段. 模型准备阶段的一般流程是首先收集大规模的自然语言以及源代码语料, 大规模语料来源包括开源文本数据集 (如 Wikipedia、Common Crawl、BooksCorpus 等), 图书和论文数据库, 开源代码库 (如 GitHub、GitLab), 编程竞赛和练习平台 (如 LeetCode、Codeforces) 等; 然后, 通过自监督学习方法训练得到预训练语言模型, 训练方法包括自回归语言模型 (autoregressive language model, ARLM), 掩码语言模型 (masked language model, MLM), 下一个句子预测 (next sentence prediction, NSP), 掩码重构 (masked reconstruction, MR) 等; 最后, 在第三方训练好的预训练语言模型 (如 OpenAI 的 GPT, Google 的 BERT, Microsoft 的 CodeBERT 等) 或自行训练的预训练语言模型的基础上, 使用少量模式实例数据集对原始的预训练语言模型进行微调得到适应设计模式检测任务的专用模型. 模型应用阶段的一般流程与机器学习的设计模式检测技术相同. 基于预训练语言模型的设计模式检测方法的整体框架如图 9 所示.

3.3.1 直接利用现有的预训练语言模型检测设计模式

目前很多知名公司和机构都训练并发布了自己的预训练语言模型. 最为知名的预训练语言模型有 Google 的 BERT[72]和 T5 [97], OpenAI 的 GPT 系列[73,74], Meta AI (原称为 Facebook AI) 的 BART[75]和 RoBERTa[98], 微软亚洲研究院的 MASS [99], 百度的 ERNIE [100,101]等. 此外还有公司和机构设计和训练了专门针对编程语言的预训练模型, 例如 Microsoft 的 CodeBERT [102], Meta AI 的 TransCoder [103]. 这些模型旨在处理与代码相关的任务, 如代码搜索、代码文档生成和代码理解等. 经过了大规模的自然语言和源代码文本数据的自监督训练, 这些模型自身已经具备了自然语言和源代码分析和处理能力, 可以直接应用于设计模式的检测或通过微调以后应用于设计模式的检测.

GPT[73,74]是一个自回归 Transformer 解码器模型. 它在预训练阶段通过自回归语言模型任务来学习序列数据. 在预训练阶段, GPT 通过逐步预测下一个词来学习序列信息, 这种方式使得 GPT 能够生成连贯的文本. GPT 可以根据提示 (prompt) 执行多种任务. 这使得 GPT 非常适合通过应用程序编程接口 (application programming interface, API) 进行访问和使用. Jánki 等人[76]基于设计模式的主要特征创建了一套详细的规则, 描述了每种 MVW 设计模式的特征和组件间的关系. 这些规则涵盖了模型、视图和控制器的交互方式, 数据传递方式以及 UI 组件的行为等. 他们将经过缩减处理 (如移除多余的空格、换行符和制表符) 后的源代码和规则转化为提示, 通过

API 与 GPT 模型 (包括 GPT-3.5 和 GPT-4) 交互, 以请求模型根据提供的规则和源代码判断是否应用了特定的设计模式. 他们没有对 GPT 进行微调以构建设计模式检测的专用模型, 研究集中于直接利用现有的 GPT 模型, 特别是 GPT-3.5 和 GPT-4 来检测设计模式. 通过这种基于规则的提示工程方法, 无需任何标注数据和模型训练, 即可访问预训练语言模型实现对复杂设计模式的检测.

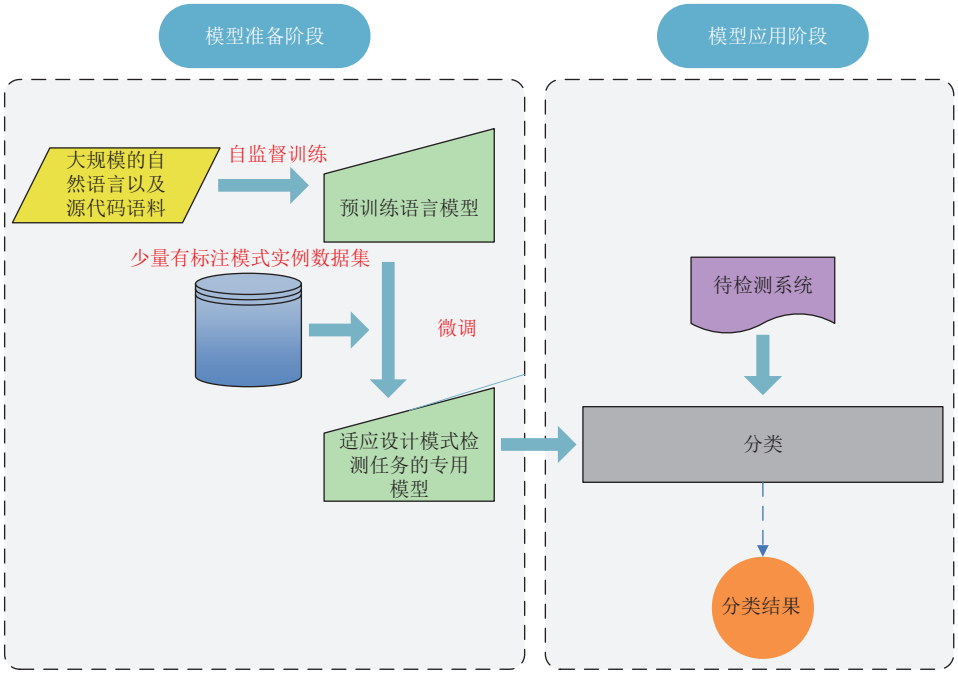


图 9 基于预训练语言模型的设计模式检测方法的整体框架

直接利用现有的预训练语言模型检测设计模式的优势在于无需或仅需极少数带标注的设计模式实例数据. 在标注数据不足或稀缺的情况下, 直接使用现有的预训练语言模型可以有效利用其在大规模无标注数据集上学到的知识, 取得较好的性能. 然而, 没有经过微调的预训练语言模型可能缺乏与设计模式检测直接相关的知识, 如果直接将其用于设计模式检测任务的预测, 可能无法达到预期的效果.

3.3.2 通过微调获取适应设计模式检测任务的专用模型

为获取设计模式检测任务专用的模型, 需要使用少量标注数据对现有的预训练语言模型进行微调.

有学者采用增加任务特定层微调 (adding task-specific layers) 的方式. CodeBERT [102]是一个由微软研究院开发的多模态预训练模型, 专为编程语言和自然语言的联合理解而设计. 这个模型基于 BERT 架构, 但进行了特定的调整和训练, 以适应源代码及其相关的自然语言文本 (如注释和文档). Dlamini 等人[77]在 CodeBERT 上增加一个梯度提升模型 CatBoost 作为分类层. 他们首先对源代码进行预处理, 移除非 Java 文件和代码中的注释及空格, 然后将处理后的代码输入到 CodeBERT 模型中生成对应的嵌入向量. 随后, 将这些嵌入向量与面向对象度量合并, 通过汇总池技术生成最终的特征向量. 生成最终特征向量后, 他们将这些特征向量作为输入训练 CatBoost 分类层进行最后的模式分类. 该方法结合深度学习和传统软件工程度量, 提高了设计模式检测的准确性和效率. TransCoder [103]是由 Facebook 预训练的跨语言程序转换模型, 利用大量未标注的代码库来学习不同编程语言之间的转换关系, 如 Java、C++ 和 Python. 与 Dlamini 等人[77]工作类似, Pandey 等人[78]在 TransCoder 上增加一个逻辑回归作为分类层. 他们利用 TransCoder 来提取程序的语义信息, 从模型的编码器中提取嵌入向量, 根据原始代码中存在的设计模式标记这些嵌入. 接下来, 他们使用标记了设计模式的嵌入创建了一个训练集, 将其输入到逻辑回归分类层以学习 C++源代码中的单例和原型 (prototype) 设计模式. 他们在 Google 上搜索了使用单例模式和使用原型模式的程

序, 这些程序已经被明确标记使用了何种模式, 然后人工检查程序的标签是否正确, 总共收集了 18 个单例模式和 8 个原型模式的 C++实现. 这种方式只对新加的层进行微调, 计算资源需求较低, 适用于小数据集. 可以看到, Pandey 等人[78]仅用 18 个单例模式和 8 个原型模式的数据微调了预训练语言模型 TransCoder 就达到了 90.0% 的准确率和 88.0% 的 F_1 分数.

还有学者采用全模型微调 (full model fine-tuning) 的方式. 陈时非等人[79]提出了一种基于 CodeBERT 的设计模式分类挖掘模型 dpCodeBERT, 对设计模式和自然语言之间的关系进行建模, 通过词嵌入技术和可扩展的微调数据集进行推理和学习, 更好地利用自然语言信息进行设计模式相关任务的辅助. 他们构建了专门的微调训练数据集, 这些数据集结合了多分类算法数据 (文本描述) 和代码搜索数据 (代码样本), 反映了设计模式的文本描述与相关代码之间的关联. 这些数据集用于 CodeBERT 模型微调, 模型利用其 Transformer 架构来分析源代码和文本描述中的结构和行为特征, 如类定义、方法调用以及代码间的交互模式. 注意力机制用于确定代码和文本中对于识别设计模式最关键的部分. 此外, 通过细致的注意力权重分析, 模型能够更精确地识别和分类软件中的设计模式, 从而提高了设计模式检测的准确率和效率. 这种方法不仅提高了设计模式识别的自动化水平, 也增强了模型对复杂代码结构的理解能力. 采用全模型微调能够更好地适应设计模式检测任务的细节, 效果通常比增加任务特定层微调更好, 但需要相对更多的计算资源和训练数据, 而且更容易过拟合.

通过微调获取适应设计模式检测任务的专用模型, 可以充分利用预训练语言模型自身具备的自然语言处理和源代码处理能力以及强大的特征提取能力, 只需少量标注数据即可训练出高质量的设计模式分类器.

3.3.3 自行构建并训练针对设计模式检测问题的预训练语言模型

上述基于预训练语言模型的设计模式检测技术都是使用第三方机构或公司训练好的预训练模型. 这些预训练模型通常由专业团队训练, 具备较高的性能和稳定性. 这种方式不需要投入大量资源进行模型的训练和维护, 可以直接或通过微调使用高质量的预训练模型, 但难以针对源代码和设计模式的特点设计和重构模型的架构和训练方式. 因此, 有学者尝试自行构建大规模源代码语料数据集, 并使用这些大规模无标注数据训练针对设计模式检测问题的预训练语言模型.

现有的预训练语言模型大多都是基于 Transformer 架构[104], Transformer 虽然在许多自然语言处理任务中表现出色, 但其计算和内存复杂度随输入序列长度的平方增长, 这使得它在处理长序列时代价非常高. 编程语言程序往往比较长, 而经典的 Transformer 架构 (如 BERT) 通常配置为处理长度在 512 至 1 024 个 token 之间的输入序列. 为了处理长序列, Parthasarathy 等人[80]使用了一种更高效的 Reformer 架构[105], 该架构结合了局部敏感哈希和可逆残差层, 能够更高效地处理长序列. 他们通过从 GitHub 公共数据集收集约 7 500 万个 C 语言文件, 构建了大规模源代码语料, 并采用 Reformer 架构在这些语料上进行预训练, 使用掩码重构任务学习程序的上下文语义. 随后, 通过在特定的汽车软件代码库上进行微调, 引入领域和设计相关知识, 最终构建了一个能高效检测控制器-处理器 (controller-handler) 设计模式合规性的预训练语言模型. 基于掩码重构任务训练代码大模型的优势在于其能够全局理解代码上下文, 捕捉长距离依赖关系, 具有更好的抗噪性和泛化能力. 这些特性使得掩码重构任务特别适合用于代码分析和设计模式检测. 相较之下, 自回归语言模型更适合代码生成任务, 在全局理解上有所不足; 掩码语言模型具有双向上下文理解的优势, 但掩码比例可能限制其对复杂模式的捕捉能力; 下一个句子预测任务在理解上下文连续性方面有优势, 但在处理代码内部复杂结构方面存在局限性.

自行构建大规模源代码语料并训练预训练语言模型进行设计模式的检测, 相对于使用第三方机构或公司训练好的预训练模型进行设计模式检测的最大优势在于, 可以自行设计模型结构并选择合适的训练方法, 从而更好地满足源代码处理和设计模式检测任务的需要. 例如 Parthasarathy 等人[80]采用了能够更好处理源代码长距离依赖的 Reformer 架构并使用掩码重构任务应对源代码的复杂结构和语义. 但是, 自行构建并训练针对设计模式检测问题的预训练语言模型也面临着诸多障碍和难题. 首先, 构建大规模的源代码语料需要获取大量的高质量代码样本, 并且需要对这些代码进行适当的标注, 以确保模型能够学习到正确的设计模式. 其次, 训练大规模预训练语言模型需要大量的计算资源, 对小型企业或个人开发者来说, 可能存在资源和成本方面的限制. 除此之外, 最重要的是预

训练语言模型在架构设计和预训练时需要深厚的机器学习和自然语言处理知识, 还需要持续的技术支持和维护以确保模型的性能和有效性, 普通开发团队可能难以完成.

3.3.4 总 结

带标注设计模式检测数据集的构建是一项非常耗时且专业性非常强的工作. 利用预训练模型强大的泛化能力和跨领域学习能力, 基于预训练语言模型的设计模式检测技术无需标注数据或仅需有限的标注数据对预训练语言模型进行调整, 就可达到良好的性能, 大幅降低对标注数据的需求. 当前预训练模型大都基于 Transformer 架构[104], Transformer 的注意力机制主要构造自然语言处理的词与词之间的关联程度, 而源代码的结构与自然语言有很大的区别. 源代码具有高度的结构化和层次化特性, 其关联不仅仅体现在词与词之间, 还体现在代码行的顺序、选择、循环结构, 函数之间的调用, 以及类之间的继承、关联、实现、依赖、创建等关系. 因此, 这些模型在挖掘源代码结构化和层次化特征方面受到限制. 此外, 预训练语言模型本质上是基于海量纯文本语料库进行预训练的, 因此在不适合文本表示的任务上表现不佳. 而设计模式的检测不仅仅是依靠纯文本形式的源代码, 源代码在执行过程中的动态行为也是检测设计模式, 尤其是对于行为型模式非常重要的因素. 这给预训练语言模型在设计模式检测任务上的应用造成了挑战.

3.4 实际项目应用案例

3.4.1 项目选择

在设计模式检测文献中, 经常使用开源项目验证所提方法. JHotDraw、JRefactory 和 JUnit 是当前文献使用较多的开源项目, 这里将给出当前设计模式检测技术在这 3 个开源项目上的实际应用案例. 表 1 列出了开源项目的基本信息[23].

表 1 JHotDraw、JRefactory 和 JUnit 项目的基本信息

项目名称	版本号	编程语言	类个数	属性个数	方法个数	代码行数(KLOC)
JHotDraw	5.1	Java	155	331	1 314	15.4
JRefactory	2.6.24	Java	569	1 364	4 234	98.5
JUnit	3.7	Java	51	111	422	5

3.4.2 代表性技术选择

这里选择几种有代表性的技术展现其在上述 3 个开源项目上的应用效果. 选择的技术方法需要具有广泛的应用性和代表性, 能够涵盖不同大类及其子类的设计模式检测方法, 并在文献中使用上述三个开源项目作为实验数据. 最终我们选择了 11 种有代表性的技术, 如表 2 所示.

表 2 代表性技术

<table>
<tr><th colspan="4">检测方法类型</th><th>代表性技术</th></tr>
<tr><td rowspan="4">非机器学习设计模式检测</td><td rowspan="2">有向图</td><td colspan="2">精确匹配</td><td>文献[23]</td></tr>
<tr><td colspan="2">非精确匹配</td><td>文献[30]</td></tr>
<tr><td colspan="3">形式化方法</td><td>文献[36]</td></tr>
<tr><td colspan="3">AST</td><td>文献[39]</td></tr>
<tr><td colspan="3">知识表示与推理</td><td>无满足条件文献</td></tr>
<tr><td rowspan="6">机器学习的设计模式检测</td><td rowspan="4">有监督学习</td><td colspan="2">传统有监督学习算法</td><td>文献[29, 55]</td></tr>
<tr><td rowspan="2">ANN</td><td>FCNN</td><td>文献[57]</td></tr>
<tr><td>LRNN</td><td>文献[52]</td></tr>
<tr><td colspan="2">GNN</td><td>文献[61]</td></tr>
<tr><td colspan="3">无监督学习</td><td>文献[63]</td></tr>
<tr><td colspan="3">集成学习</td><td>文献[68]</td></tr>
<tr><td>基于预训练语言模型的设计模式检测</td><td colspan="4">无满足条件文献</td></tr>
</table>

3.4.3 应用过程和效果分析

设计模式检测技术在 3 个开源项目中具体应用过程如下.

(1) 项目准备: 下载 JHotDraw、JRefactory 和 JUnit 的源码, 确保代码可以成功编译并正常运行. 对源码进行预处理, 如删除不必要的注释、格式化代码等.

(2) 特征提取: 通过静态分析、行为分析、嵌入等方式提取源码特征.

(3) 模型训练或规则定义: 通过人工规则定义、机器学习算法或预训练语言模型生成检测规则或模型.

(4) 模式检测: 使用预定义的规则在提取的特征信息中进行匹配或将源码特征输入模型中进行预测, 从而检测系统中的设计模式.

(5) 结果统计和验证: 对检测结果进行验证, 记录每个项目中检测到的设计模式实例, 并进行统计分析.

表 3–表 5 分别列出了对开源项目 JHotDraw、JRefactory 和 JUnit 进行设计模式识别的实验结果.

可以看到, 文献 [23,30] 这两种基于有向图的设计模式检测方法取得了不错的效果, 但主要集中在适配器、组合和装饰者等少数几种结构型模式. 文献 [30] 虽然也可以检测到行为型模式中的状态模式, 但精确率、召回率和 F1 分数都偏低. 另外有向图的方法也不擅长处理非常复杂的模式, 例如文献 [23] 在 JUnit 上的组合模式的精确率和 F_1 分数也明显降低.

文献 [36] 通过静态行为特征和动态行为特征对使用静态结构分析[37]得到的初步模式实例进行区分和过滤, 成功检测出了 JHotDraw 和 JRefactory 中的组合、观察者、状态、策略、模板方法、访问者 (visitor) 这 6 种行为型模式, 且整体表现较好, 尤其在模板方法和命令模式上表现完美. 然而, 相对更为复杂的观察者和访问者两种行为型模式在 JHotDraw 和 JRefactory 上的误报率较高. 另外, JHotDraw 上的状态和策略两种模式的精确率也很低, 可能是原因是文献 [36] 无法很好地区别这两种结构相似的行为型模式, 导致二者彼此之间的混淆.

文献 [39] 通过解析源代码生成 AST 并结合结构搜索模型逐步匹配和构建设计模式的结构, 充分有效利用了源代码中的静态结构和静态行为特征, 能够准确地识别出包括备忘录 (memento)、命令、访问者等行为型模式在内的多种设计模式. 总体而言, 文献 [39] 方法在设计模式检测中表现出色, 尤其在创建型和结构型模式的检测中显示了高精确率和召回率, 例如 JHotDraw 上的单例、原型、组合、装饰者、外观、享元 (flyweight) 几种创建型和结构型模式的精确率、召回率和 F_1 分数均为 100.0%. 然而, 由于未进行动态分析, 命令模式在 JHotDraw 和 JRefactory 上的效果都不是很好, 精确率不足 70.0%, 访问者模式在两个项目上的召回率也较低, 仅为 50.0%, 针对复杂行为模式的检测效果还有待提高. 对于状态和策略两种模式, 文献 [39] 等无法进行区别.

在 JHotDraw、JRefactory 和 JUnit 这 3 个项目上, 文献 [29] 的 SVM 分类器都获得了很好的检测效果, 但是决策树和 KNN 对于不同的项目、不同的模式表现不稳定. 对于 JHotDraw 的适配器、组合、工厂方法模式, 决策树和 KNN 的精确率、召回率和 F_1 分数都在 90.0% 以上, JUnit 的观察者的精确率、召回率和 F_1 分数也都接近于 90.0%. 但是对于 JRefactory 的适配器模式, 决策树的精确率、召回率和 F_1 分数分别仅为 64.0%, 78.0%, 70.3%, KNN 仅为 53.0%, 73.0%, 61.4%, 总体上低于文献 [23,30,39,63]. 主要原因在于 KNN 算法对样本的平衡性要求较高, 而决策树算法容易出现过拟合, 且在处理关联特征比较强的数据时表现得不是太好.

Chihada 等人[55]在 JUnit 的迭代器和观察者两种模式上均取得了 100.0% 的精确率、召回率和 F_1 分数, 超过了文献 [29,68]. 在 JHotDraw 上的表现总体上不如文献 [29,61,63,68], 在 JRefactory 的适配器模式上表现总体不如文献 [29] 的 SVM 分类器, 文献 [68] 的堆叠分类器, 文献 [63] 聚类与有监督学习算法相结合的分类器以及文献 [23,30,39] 非机器学习的方法, 但除工厂方法模式外精确率、召回率和 F_1 值都在 70.0% 以上. 因此, 总体来说在这 3 个项目上都取得了不错的效果, 且在不同模式上的表现也较为稳定, 事实上这种稳定性也是 SVM 算法的优势之一.

在 JHotDraw 上, 文献 [57] 的组合、装饰者、观察者、代理模式取得了较好的召回率, 但是大多数模式的精确率相对于其他文献较低, 其中组合和命令模式的精确率不足 20.0%, 而观察者模式仅为 1.1%. 可能的原因在于采用层次较深的 FCNN, 而数据集的规模又不是很大, 导致出现了过拟合问题. 适配器和命令模式的低精度和召回

表 3 JHotDraw 上的应用效果 (%)

设计模式	文献[23]			文献[30]			文献[36]			文献[39]			文献[29]								
													决策树			KNN			SVM		
	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
外观	—	—	—	—	—	—	—	—	—	100.0	100.0	100.0	—	—	—	—	—	—	—	—	—
适配器	96.0	100.0	98.0	90.0	85.0	87.5	—	—	—	42.0	100.0	59.2	91.0	100.0	95.2	85.0	100.0	91.9	100.0	100.0	100.0
桥接	—	—	—	—	—	—	—	—	—	71.0	100.0	83.0	—	—	—	—	—	—	—	—	—
命令	—	—	—	—	—	—	100.0	100.0	100.0	67.0	89.0	76.4	—	—	—	—	—	—	—	—	—
组合	100.0	100.0	100.0	85.0	75.0	80.0	—	—	—	100.0	100.0	100.0	97.0	100.0	98.5	99.0	100.0	99.5	99.0	100.0	99.5
装饰者	100.0	100.0	100.0	100.0	100.0	100.0	—	—	—	100.0	33.0	50.0	—	—	—	—	—	—	—	—	—
享元	—	—	—	—	—	—	—	—	—	100.0	100.0	100.0	—	—	—	—	—	—	—	—	—
工厂方法	—	—	—	—	—	—	—	—	—	—	—	—	93.0	100.0	96.4	96.0	100.0	98.0	100.0	100.0	100.0
观察者	—	—	—	—	—	—	55.0	100.0	71.0	—	—	—	—	—	—	—	—	—	—	—	—
原型	—	—	—	—	—	—	—	—	—	100.0	100.0	100.0	—	—	—	—	—	—	—	—	—
单例	—	—	—	—	—	—	—	—	—	100.0	100.0	100.0	—	—	—	—	—	—	—	—	—
状态	—	—	—	80.0	78.0	79.0	5.6	100.0	10.6	27.0	100.0	46.0	—	—	—	—	—	—	—	—	—
策略	—	—	—	—	—	—	46.5	100.0	63.5				—	—	—	—	—	—	—	—	—
模板方法	—	—	—	—	—	—	100.0	100.0	100.0	—	—	—	—	—	—	—	—	—	—	—	—
访问者	—	—	—	—	—	—	100.0	100.0	100.0	100.0	50.0	66.7	—	—	—	—	—	—	—	—	—

设计模式	文献[55]			文献[57]			文献[61]			文献[63]			文献[52]						文献[68]		
													决策树			LRNN					
	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
适配器	86.0	86.0	86.0	50.0	24.0	32.4	93.0	84.0	88.3	86.0	85.0	85.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	66.7	80.0
命令	—	—	—	19.5	61.5	29.6	94.0	92.0	93.0	—	—	—	—	—	—	—	—	—	—	—	—
组合	74.0	74.0	74.0	13.3	80.0	22.8	78.0	88.0	82.8	80.0	78.0	79.0	—	—	—	—	—	—	100.0	100.0	100.0
装饰者	—	—	—	80.0	100.0	88.9	—	—	—	86.0	85.0	85.0	—	—	—	—	—	—	—	—	—
工厂方法	61.0	61.0	61.0	—	—	—	82.0	72.0	76.8	88.0	86.0	87.0	—	—	—	—	—	—	100.0	75.0	85.7
观察者	—	—	—	1.1	100.0	2.2	85.0	73.0	78.7	—	—	—	—	—	—	—	—	—	66.7	100.0	80.0
代理	—	—	—	100.0	100.0	100.0	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
单例	—	—	—	—	—	—	79.8	79.8	79.8	88.0	86.0	87.0	—	—	—	—	—	—	100.0	100.0	100.0
抽象工厂	—	—	—	—	—	—	—	—	—	—	—	—	100.0	91.7	95.7	100.0	100.0	100.0	—	—	—

注: P表示*Precision*, R表示*Recall*, F_1表示F_1-*Score*

表 4　JRefactory 上的应用效果 (%)

设计模式	文献[23]			文献[30]			文献[36]			文献[39]			文献[29]								
													决策树			KNN			SVM		
	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
适配器	97.0	100.0	98.0	89.0	87.0	88.0	—	—	—	88.0	94.0	91.0	64.0	78.0	70.3	53.0	73.0	61.5	100.0	100.0	100.0
命令	—	—	—	—	—	—	100.0	—	—	69.0	88.0	78.0	—	—	—	—	—	—	—	—	—
组合	100.0	100.0	100.0	88.0	83.0	85.5	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
装饰者	—	—	—	—	—	—	—	—	—	100.0	100.0	100.0	—	—	—	—	—	—	—	—	—
工厂方法	—	—	—	—	—	—	—	—	—	83.0	91.0	87.0	—	—	—	—	—	—	—	—	—
观察者	—	—	—	—	—	—	100.0	—	—	—	—	—	—	—	—	—	—	—	—	—	—
单例	—	—	—	—	—	—	—	—	—	100.0	83.0	83.0	—	—	—	—	—	—	—	—	—
状态	—	—	—	78.0	74.0	76.0	100.0	—	—	73.0	73.0	62.3	—	—	—	—	—	—	—	—	—
策略	—	—	—	—	—	—	100.0	—	—				—	—	—	—	—	—	—	—	—
模板方法	—	—	—	—	—	—	100.0	—	—	—	—	—	—	—	—	—	—	—	—	—	—
访问者	—	—	—	—	—	—	50.0	—	—	100.0	50.0	66.7	—	—	—	—	—	—	—	—	—

设计模式	文献[55]			文献[57]			文献[61]			文献[63]			文献[52]						文献[68]		
													决策树			LRNN					
	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
适配器	76.0	76.0	76.0	—	—	—	—	—	—	88.0	87.0	87.0	—	—	—	—	—	—	100.0	66.2	79.7
组合	—	—	—	—	—	—	—	—	—	82.0	80.0	81.0	—	—	—	—	—	—	—	—	—
装饰者	—	—	—	—	—	—	—	—	—	88.0	87.0	87.0	—	—	—	—	—	—	—	—	—
工厂方法	—	—	—	—	—	—	—	—	—	90.0	88.0	89.0	—	—	—	—	—	—	100.0	66.7	80.0
观察者	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	66.7	100.0	80.0
单例	—	—	—	—	—	—	—	—	—	90.0	88.0	89.0	—	—	—	—	—	—	91.7	100.0	95.6

注: P表示*Precision*, R表示*Recall*, F_1表示F_1-*Score*

表 5 JUnit 上的应用效果 (%)

设计模式	文献[23]			文献[30]			文献[36]			文献[39]			文献[29]								
													决策树			KNN			SVM		
	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
适配器	100.0	100.0	100.0	95.0	92.0	93.5	—	—	—	71.0	71.0	71.0	—	—	—	—	—	—	—	—	—
命令	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
组合	50.0	100.0	66.7	100.0	90.0	95.0	—	—	—	100.0	100.0	100.0	—	—	—	—	—	—	—	—	—
装饰者	100.0	100.0	100.0	—	—	—	—	—	—	100.0	100.0	100.0	—	—	—	—	—	—	—	—	—
工厂方法	—	—	—	—	—	—	—	—	—	100.0	100.0	100.0	—	—	—	—	—	—	—	—	—
备忘录	—	—	—	—	—	—	—	—	—	100.0	100.0	100.0	—	—	—	—	—	—	—	—	—
迭代器	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
观察者	—	—	—	—	—	—	—	—	—	—	—	—	87.0	88.0	87.5	91.0	88.0	89.4	94.0	88.0	91.0
状态	—	—	—	90.0	85.0	87.5	—	—	—	100.0	100.0	100.0	—	—	—	—	—	—	—	—	—
策略	—	—	—	—	—	—	—	—	—				—	—	—	—	—	—	—	—	—
模板方法	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—

设计模式	文献[55]			文献[57]			文献[61]			文献[63]			文献[52]						文献[68]		
													决策树			LRNN					
	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
适配器	—	—	—	—	—	—	—	—	—	78.0	76.0	77.0	—	—	—	—	—	—	100.0	100.0	100.0
组合	—	—	—	—	—	—	—	—	—	73.0	71.0	72.0	—	—	—	—	—	—	100.0	100.0	100.0
装饰者	—	—	—	—	—	—	—	—	—	79.0	77.0	78.0	—	—	—	—	—	—	—	—	—
工厂方法	—	—	—	—	—	—	—	—	—	82.0	80.0	81.0	—	—	—	—	—	—	100.0	50.0	66.7
迭代器	100.0	100.0	100.0	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
观察者	100.0	100.0	100.0	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	100.0	50.0	66.7
单件	—	—	—	—	—	—	—	—	—	82.0	80.0	81.0	—	—	—	—	—	—	—	—	—

注: P表示*Precision*, R表示*Recall*, F_1表示F_1-*Score*

率可以部分归因于二者检测结果之间的混淆, 这种混淆可能是由于在借助现有的检测工具准备数据集时, 一些工具无法准确地区分这两种模式而造成的.

Dwivedi 等人[52]采用了 LRNN 而不是 FCNN 检测设计模式, 减轻了过拟合问题, 在 JHotDraw 的抽象工厂模式和适配器模式上取得了更好的效果. 但是没有讨论观察者等行为型模式, 因为他们主要是通过结构特征和度量特征检测设计模式.

在 JHotDraw 项目上, 文献 [61] 所有模式的精确率、召回率和 F_1 分数都在 80.0% 以上或接近 80.0%, 尤其是适配器和命令模式的精确率、召回率和 F_1 分数分别达到 93.0%/84.0%/88.3 和 94.0%/92.0%/93.0%. 这得益于 GNN 模型能够捕获节点 (如类、方法) 以及边 (如继承、调用关系) 之间的复杂关联, 提高了检测的精确性和鲁棒性. 虽然在个别模式上的效果不如文献 [23,30] 两种非机器学习的基于有向图的方法, 但文献 [61] 涵盖的模式种类更广, 结构型、创建型和行为型模式都可以很好地支撑, 且在不同模式上表现更为稳定.

文献 [63] 在训练分类模型前使用 k-Means 和 CLOPE 聚类算法对特征向量进行了维度统一, 在 JHotDraw、JRefactory 和 JUnit 三个项目上也有不错的效果, 不同模式上的表现比较稳定. 但检测到的模式主要集中在创建型和结构型模式上, 行为型模式的检测有所欠缺.

文献 [68] 在 JHotDraw 上大多数模式都比取得了比文献 [29,52,55,57,61,63] 的非集成分类器更高的准确率、召回率和 F_1 值, 说明模型堆叠算法起到了较好地作用. 在 JRefactory 上, 除文献 [29] 的 SVM 的适配器模式分类器外, 文献 [68] 总体上取得了比非集成分类器更好的分类效果. 在 JUnit 上, 文献 [68] 的观察者模式分类效果不如文献 [29,55], 说明集成学习可能在某些情况下产生了负作用.

3.4.4 总 结

从整体和宏观上来看, 机器学习的设计模式检测方法无论是在准确率还是支持的设计模式种类的全面程度上都优于非机器学习的设计模式检测方法. 其中, 深度学习和集成学习的设计模式检测方法又总体上优于基于决策树、KNN、SVM 等传统机器学习算法的检测方法. 但同时由于高质量大规模有标注设计模式实例数据集较为稀缺, 且在各个模式上的分布不均匀, 机器学习的设计模式检测方法所能检测的设计模式种类仍然比较有限. 相对简单的模式, 例如单例模式, 可以被很好地识别出来. 而更为复杂的设计模式, 例如组合模式、观察者模式、访问者模式的检测效果有所不足, 尤其是复杂的行为型模式. 很多方法无法很好地区别结构相近的行为型模式, 例如状态模式与策略模式.

4 当前成果总结和比较

4.1 当前设计模式检测技术文献总结与比较

根据上述讨论, 从所属大类、所使用的技术、特征类型、支持的模式类型这 4 个方面出发对当前设计模式检测技术文献进行了总结和比较, 如表 6 所示. 在表 6 中, 第 1 列是所讨论的文献; 第 2 列为文献所属大类, 包括非机器学习的设计模式检测、机器学习的设计模式检测或基于预训练语言模型的设计模式检测; 第 3 列为文献所使用的技术, 例如非机器学习的 Prolog、ASP、图同构判定等以及机器学习的决策树、SVM、ANN 等; 第 4 列为文献考虑的特征类型, 包括结构特征、行为特征、度量特征、语义特征和补充特征; 第 5 列为文献的检测对象, 包括源代码和设计模型 (例如 UML 模型、Petri 网等); 第 6 列为文献支持的模式类型, 主要包括 GoF 的结构型、创建型和行为型模式以及系统架构级的模式.

根据表 6, 可知:

(1) 目前机器学习的设计模式检测技术和基于预训练言模型的设计模式检测技术成为主流, 近 10 年来提出的设计模式检测技术大多属于这两大类. 预计未来几年中, 基于 BERT、GPT、BART 等预训练语言模型的设计模式检测技术将会受到越来越多学者的关注, 并产生大量在准确率等性能指标方面更优的研究成果.

(2) 在机器学习的设计模式检测技术中, 所选用的机器学习算法主要是有监督算法, 在有监督算法中, 决策树、

SVM 和 ANN (含 CNN、LRNN 等深层神经网络和图神经网络) 这 3 种算法最常用 (采用决策树的文献 8 篇, 采用 SVM 的文献 7 篇, 采用 ANN 的文献 11 篇), 另有少量文献采用了 KNN、线性/多项式/逻辑回归、朴素贝叶斯等算法 (采用 KNN 的文献 2 篇, 采用线性/多项式/逻辑回归的文献 4 篇, 采用朴素贝叶斯的文献 1 篇). 此外, 近 3 年, 越来越多的学者通过集成学习来获得性能更好的模型 (7 篇文献), 其中使用最多的算法是随机森林 (6 篇文献). 有少量文献尝试使用无监督算法进行特征转换 (3 篇) 或直接检测设计模式 (2 篇).

(3) 结构特征是设计模式最重要也是最容易提取的特征, 大多数文献都考虑了结构特征 (44 篇). 部分文献在此基础上进一步考虑了静态行为特征 (31 篇) 或动态行为特征 (5 篇). 有 4 篇文献在结构特征、行为特征、度量特征的基础上通过语义特征增强检测效果, 而在 8 篇基于嵌入技术以及预训练语言模型方法的文献中, 语义特征是设计模式检测最主要的信息来源. 另有部分文献单独通过度量特征或者将度量特征与结构特征和行为特征结合来检测设计模式 (18 篇). 2 篇文献进一步考虑了补充特征.

(4) 绝大部分文献的检测对象是源代码 (含 Java 字节码文件)(40 篇), 部分文献将 UML 模型 (12 篇)、Petri 网 (1 篇) 等软件设计模型作为检测对象, 2 篇文献在源代码的基础上将注释和文档作为辅助.

(5) 当前设计模式检测技术主要是针对 GoF 的 23 种设计模式, 其中结构型模式的检测相对较为容易, 几乎所有的技术都支持结构型模式. 行为型模式的检测往往需要考虑行为特征, 部分文献不支持行为型模式的检测. 此外

表 6 当前设计模式检测技术文献的总结和比较

文献	所属大类	所使用的技术	特征类型	检测对象	支持的模式类型
[23,24]	非机器学习设计模式检测	图同构判定	结构, 静态行为	源代码	创建型, 结构型, 行为型
[25]		图同构判定	结构, 静态行为	UML模型	创建型, 结构型, 行为型
[26]		图同构判定	结构, 静态行为	UML模型	创建型, 结构型, 行为型
[27]		图同构判定	结构, 静态行为, 语义	源代码为主, 注释和文档作为辅助	创建型, 结构型, 行为型
[28]		图同构判定	结构	UML模型	创建型, 结构型, 行为型
[29]		图同构判定, 后续机器学习阶段: 决策树、KNN、SVM	结构, 后续机器学习阶段: 度量	源代码	创建型, 结构型, 行为型
[30]		模板匹配	结构, 静态行为	源代码	结构型
[31]		模板匹配	结构, 静态行为, 语义	UML模型	结构型, 行为型
[32]		相似度评分	结构, 静态行为	源代码	创建型, 结构型, 行为型
[33,34]		余弦相似度	结构	UML模型	创建型, 结构型, 行为型
[35]		模型检测	结构, 静态行为	源代码, Java字节码	创建型, 结构型, 行为型
[36]		模型检测	结构, 静态和动态行为	源代码	行为型
[38]		有限自动机	结构, 动态行为	源代码	行为型
[39]		AST	结构, 静态行为	源代码	创建型, 结构型, 行为型
[40]		AST	结构, 静态行为	源代码	创建型, 结构型, 行为型
[19]		Prolog	结构, 静态行为	源代码	结构型
[41]		Prolog	结构, 静态和动态行为	源代码	结构型, 行为型
[42]		Prolog	结构, 静态行为	UML模型	创建型, 结构型, 行为型
[43]		ASP	结构, 静态行为	UML模型	结构型, 行为型
[44]		本体(Ontology)技术	结构, 静态行为	UML模型	行为型
[45]		本体(Ontology)技术	结构, 静态行为	UML模型	创建型, 结构型, 行为型
[46]		XML	结构, 静态行为	UML模型	创建型, 结构型, 行为型
[47]		字符串匹配	结构	源代码	创建型, 结构型, 行为型
[48]		遗传算法+图同构判定	结构	UML模型	创建型, 结构型, 行为型

表 6 当前设计模式检测技术文献的总结和比较 (续)

文献	所属大类	所使用的技术	特征类型	检测对象	支持的模式类型
[50]	机器学习的设计模式检测	ANN, 逻辑回归	结构, 度量	源代码	创建型, 结构型, 行为型
[51]		ANN, SVM, 随机森林	度量	源代码	创建型, 结构型, 行为型
[52]		决策树, LRNN	结构, 度量	源代码	创建型, 结构型, 行为型
[53]		决策树、ANN	结构, 静态行为, 度量, 补充	源代码	结构型, 行为型
[54]		决策树, LRNN	结构, 度量	源代码	创建型, 结构型
[55]		SVM	度量	源代码	创建型, 行为型
[56]		PCA+线性回归、多项式回归、SVM、ANN	度量	源代码	创建型, 结构型, 行为型
[57]		ANN	结构, 静态行为, 度量	源代码	创建型, 结构型, 行为型
[58,59]		ANN	度量	源代码	创建型, 结构型, 行为型
[60]		关联分类	结构, 静态行为, 度量	源代码	创建型, 结构型, 行为型
[61]		图神经网络	结构, 静态行为	源代码	创建型, 结构型
[62]		k-Means+决策树	结构	源代码	创建型, 结构型, 行为型
[63]		k-Means和CLOPE [89]+有监督机器学习算法	结构, 静态行为, 度量	源代码	创建型, 结构型
[64]		Word2Vec和Code2Vec+k-Means	结构, 静态行为, 语义	源代码	创建型
[65]		关联分析	结构, 静态和动态行为, 度量, 语义	UML模型、Petri网等	架构级
[66]		CNN、随机森林	结构, 静态行为, 语义	源代码为主, 注释和文档作为辅助	结构型, 行为型
[67]		GBT等树形机器学习算法	度量	源代码	结构型, 行为型
[68]		堆叠算法	结构, 静态行为, 度量	源代码, Java字节码	创建型, 结构型, 行为型
[69]		Word2Vec+随机森林	结构, 静态行为, 语义	源代码	创建型, 结构型, 行为型
[70]		Doc2Vec+SVM、逻辑回归、随机森林	结构, 静态行为, 度量, 语义	源代码	创建型, 结构型, 行为型
[71]		CatBoost和EBM等9种机器学习算法	度量	源代码	架构级
[76]	基于预训练语言模型的设计模式检测	GPT	结构, 静态行为, 语义	源代码	架构级
[77]		CodeBERT增加任务特定层微调	结构(间接), 静态行为(间接), 度量, 语义	源代码	架构级
[78]		TransCoder增加任务特定层微调	结构(间接), 静态行为(间接), 语义	源代码	创建型
[79]		CodeBERT全模型微调	结构(间接), 静态行为(间接), 语义, 补充	源代码	创建型, 结构型, 行为型
[80]		自行构建与训练的模型	结构(间接), 静态行为(间接), 语义	源代码	架构级

注: ‘‘检测对象’’一列中, 若是需要先将源代码转换为UML等设计模型, 则认为检测对象是设计模型而不是源代码

近年来陆续有文献关注架构级设计模式的检测 (5 篇).

4.2 软件设计模式检测三大类方法比较

在第 4.1 节的基础上, 进一步从非机器学习的设计模式检测、机器学习的设计模式检测和基于预训练语言模型的设计模式检测这 3 类方法以及各自包含的子类进行了综合总结并进行深入详细的分析和比较. 包括各类方法在准确率、效率、鲁棒性、可扩展性等方面的优势与局限性、主要适用场景以及在实际应用中可能遇到的挑战. 如表 7 所示.

表 7 软件设计模式检测三大类方法及其子类比较

检测方法类型			优势	局限性	主要适用场景	在实际应用中可能遇到的挑战
非机器学习设计模式检测	有向图	精确匹配	鲁棒性强, 精度高	变体处理能力不足, 召回率低	设计模式实现较为规范、变体较少的系统, 以及对检测精度要求较高的场景	无法有效处理结构复杂的设计模式以及行为型模式, 计算复杂度高
		非精确匹配	可以检测模式变体, 提升召回率	非精确匹配可能导致精确率降低	设计模式变体较多的系统, 以及对召回率要求较高的场景	
	形式化方法		语义清晰无歧义, 可通过过滤假阳性提高精度	模型构建过程复杂且耗时	行为型模式候选实例的区分和过滤	需要高水平的专业和数学知识
	AST		精确表示源代码结构, 便于静态行为分析	AST本身难以表示和处理运行时的动态行为信息	需要在不执行代码的情况下对源代码进行行为分析和设计模式检测的场景	缺乏对动态信息的支持, 使得行为型设计模式的检测效果可能不佳
	知识表示与推理		便于实现支撑工具, 便于检测规则的添加和修改	构建和维护知识库及规则的过程复杂且耗时	需要频繁更新和扩展检测规则的场景	需要深厚的领域知识和技术能力
机器学习的设计模式检测	有监督学习	传统有监督学习算法	模型复杂度低, 实现过程相对简单	对特征选择的依赖性强, 结果的不确定性高	数据集规模较小的场景, 计算资源受限的场景	需要进行有效的特征工程, 需要通过多次实验和调优来确保模型效果
		人工神经网络	强大的非线性建模能力	对数据量要求高, 计算资源需求高	拥有大量标注数据和高性能计算资源的场景, 需要检测复杂设计模式的场景	容易产生过拟合问题, 高质量标注数据获取成本高, 计算资源限制
		图神经网络	高效匹配, 高精度, 擅长处理节点之间的结构化和层次化信息	模型结构相对复杂, 训练和调参需要较高的技术水平和经验		
	无监督学习		无需标注数据	需要结合监督学习算法或与已经明确定义的模式比对才能得到分类结果	通过无监督学习算法进行特征转换, 从而提升处理效率和性能	主要用作特征转换, 难以直接对系统进行设计模式分类
	集成学习		提升准确率、鲁棒性和泛化能力, 减少过拟合	模型复杂性高, 训练时间长, 调参难度大	数据不平衡的场景, 需要高鲁棒性和准确性的场景	计算资源需求高, 模型融合和管理较为困难
基于预训练语言模型的设计模式检测	直接利用现有的预训练语言模型检测设计模式		无需或仅需极少数标注数据, 高效利用已有知识	缺乏专门知识, 检测效果可能不理想	缺乏充足标注数据的场景, 以及需要快速部署设计模式检测平台的场景	模型没有经过微调可能会产生较多误判, 通过提示 (prompts)和 API访问模型时需要根据设计模式的特征和上下文精心设计提示
	通过微调获取适应设计模式检测任务的专用模型		在利用已有知识的基础上, 仅需要少量标注数据即可通过模型微调进一步提高准确率	依赖初始模型的质量和架构，适应性有限	需要在现有预训练模型基础上进一步提升设计模式检测性能的场景	技术水平要求较高, 需要具备丰富的机器学习和自然语言处理经验
	自行构建大规模源代码语料并训练预训练语言模型		定制化设计, 灵活的模型架构和训练方法	高计算资源需求, 训练语料获取和处理复杂, 技术要求高	拥有充足资源和技术团队的大型企业和研究机构, 能够承担高成本和复杂的技术要求	计算资源和成本限制, 模型设计和维护困难, 技术复杂性高, 实现难度大

5 主要问题与挑战及未来研究方向

5.1 主要问题与挑战

在软件设计模式检测领域, 研究者和实践者已经进行了大量的研究和探索, 并取得了一些阶段性成果. 然而,

由于设计模式检测问题本身的复杂性, 通过前文的综述, 我们可以看到该领域仍面临着若干问题和挑战.

5.1.1 公开可用的数据集匮乏

数据集的质量高低 (是否平衡、空值和缺失值比例高低、噪声样本比例高低) 及规模大小是影响分类器性能的一个非常重要的因素. 例如: Alhusain 等人[57]的研究中，适配器和命令两种模式的精度和召回率较低, 部分原因在于数据集中这两种模式之间相互混淆; Uchiyama 等人[58,59]中大规模程序的召回率低于小规模程序, 这是因为大规模程序包含更多与模式无关的属性和操作, 而训练数据集规模不大导致模型出现了过拟合. 手写数字识别、情感分析、人脸识别等经典分类问题有很多公开可用的数据集, 而对于设计模式检测问题, 目前学术界和工程界缺乏公开可用的高质量大规模数据集. 目前机器学习的设计模式检测技术的研究主要依赖于研究人员自己构建的数据集. 构建高质量的数据集需要研究人员具备深厚的设计模式知识和丰富的实践经验, 成本高昂且费时费力. 因此这些数据集的质量参差不齐, 大多数数据集的规模都比较小. 此外这些数据集在文件形式、标注方式、样本分布等方面存在很大的差异, 数据集之间的兼容性和可比性较差. 以上这些问题给机器学习设计模式检测技术, 包括基于预训练语言模型的设计模式检测技术的模型训练和评估以及跨研究比较和综合分析带来很大的挑战和困难.

5.1.2 动态行为的捕获和表示比较困难

源代码运行过程中的动态行为是检测设计模式尤其是行为型模式的重要依据. 动态行为涉及方法调用顺序、对象的状态变化和消息传递等, 这些信息往往在运行时才会显现. 相对于获取待检测系统的静态特征, 获取动态行为特征更为困难. 因为要捕获系统的动态行为, 需要在特定的执行环境下运行程序. 这包括搭建和设置运行环境、准备输入数据、执行程序并捕获运行时数据. 这一过程需要大量的准备工作, 并且执行环境的差异可能导致行为捕获结果的不一致. 从表 1 可见目前很多文献都只是通过系统的静态结构和静态行为来检测设计模式. 此外, 源代码的动态行为是一个复杂的、动态的过程, 而检测算法和模型擅长处理的数据往往是静态的文本或图形等形式. 如何将捕获的动态行为表示为检测算法方便处理的数据形式, 也是该领域的一个难点和挑战. 特别是近年来兴起的预训练语言模型主要擅长处理文本形式的数据, 源代码的动态运行过程无法直接输入到预训练语言模型网络结构中进行处理, 这给预训练语言模型在源代码设计模式检测中的应用造成障碍.

5.1.3 新的或未知的模式类别难以识别

虽然 GoF 的 23 种设计模式是最著名、应用最广泛的设计模式, 但软件设计并不仅限于这 23 种模式. 除了 GoF 模式之外, 很多面向对象领域的科研人员和工程技术人员提出一些新的设计模式, 或者使用了一些优秀的设计范例却并未将这些范例进行总结和公开. 当前研究主要集中在检测特定已经明确定义的模式的实例, 尤其是 GoF 设计模式, 以及 MVC、MVP、MVVM 等常见的架构级模式. 当前的设计模式检测技术, 无论是非机器学习的设计模式检测技术, 机器学习的设计模式检测技术还是基于预训练语言模型的设计模式检测技术, 均需要根据模式的定义和描述构建检测规则 (人工构建或训练模型来构建). 而新的或尚未总结的设计模式通常缺乏明确的定义和详细的描述, 因此难以依靠当前设计模式检测技术来识别.

5.1.4 当前预训练语言模型挖掘源代码结构化和层次化语义能力有限

基于预训练语言模型进行设计模式检测是目前解决该问题最有潜力的一个方向. 自然语言具有线性特性, 通常以顺序的方式进行阅读和理解, 句子结构是线性和顺序的. 而源代码则不完全是线性结构, 其元素包括文件、类、函数、代码行、单词等不同层级结构, 而逻辑又可能包含嵌套、循环、分支等复杂的控制结构, 这些结构使得源代码的分析和理解需要考虑到其高度层次化和结构化的特点. 当前流行的 GPT、BERT 等预训练语言模型的架构以及训练方式主要是适用于自然语言的线性特性, 难以有效挖掘源代码的深层次语义, 限制了其理解和处理源代码能力. 有学者使用了 CodeBERT [77,79]和 TransCoder [78]等第三方训练的或自行在大规模源代码语料上训练的专门用于代码理解和生成的预训练模型[80]. CodeBERT [102]和 TransCoder [103]与针对自然语言处理的预训练语言模型在模型结构上差别不大, Parthasarathy 等人[80]自行训练的预训练模型架构主要着眼于高效地处理长序列而不是挖掘源代码的深层次特征. 在预训练方式上 CodeBERT [102]增加了替换词检测 (replaced token detection, RTD) 任务, TransCoder [103]增加了跨语言掩蔽模型 (cross-lingual masked language model, XMLM) 任务, Parthasarathy 等人[80]使用了掩码重构任务, 这些预训练方式也没有突出源代码的结构化和层次化特性. 因此, 这些预训练语言模

型主要依赖大量的源代码和对应的自然语言文档适应源代码, 是从数据的角度而不是从专门针对源代码特点的模型架构和训练方式的角度来设计和构建.

5.1.5 支撑工具不完善或未公开

设计模式检测技术的发展在很大程度上依赖于高效的支撑工具和平台. 设计模式检测支撑工具背后的原理、技术和算法较为深奥复杂, 导致其设计与实现的难度和工作量都比较大. 因此目前文献提供的支撑工具大多都还只是系统原型, 尚处于实验室研究阶段, 功能和性能上存在诸多限制. 这些工具在功能上通常比较单一, 主要着眼于实现待检测系统的导入、设计模式的检测和检测结果的输出等最基础的功能. 工具开发过程中往往更多关注算法性能和检测精度, 忽视了用户体验和易用性, 导致工具在用户界面和交互设计上不够友好, 使用门槛较高, 缺乏直观的操作和清晰的结果展示. 许多工具独立开发, 缺乏与主流集成开发环境 (IDE) 的良好集成, 使用时需要额外配置和操作, 增加了使用难度. 此外这些工具大多未公开, 其有效性只能通过文献中的描述获知, 难以对文献的实验进行重复, 也无法直接应用于工程实践. 目前商业级别的设计模式检测工具还非常少.

5.1.6 在一次寻找多个设计模式实例并确定其位置方面有所欠缺

当前的机器学习的设计模式检测技术大多是将设计模式检测问题看作是分类问题, 多采用决策树、SVM、ANN 等分类算法. 这些分类算法在设计模式检测时一次性只能处理一个模式实例. 而事实上, 设计模式检测更类似于计算机视觉中的目标检测问题, 需要一次从整个软件系统或者代码片段中寻找多个设计模式实例并确定其位置, 当前研究在这方面仍存在显著不足. 一个软件系统或者代码片段中可能存在多种设计模式, 而一种设计模式可能有多个实例, 基于分类算法的设计模式检测技术每次只能处理一个模式实例, 在寻找多个设计模式实例时容易产生误报和漏报, 影响检测结果的可靠性, 也降低了执行效率. 另外, 一个设计模式实例通常涉及多个类, 而多个模式实例之间又可能存在公共的类, 其边界和组成部分在代码中分散且不易辨识, 导致现有设计模式检测方法和工具在模式实例定位上精度不足.

5.1.7 端到端的设计模式检测技术较少

除 Pandey 等人[78]、陈时非等人[79]和 Parthasarathy 等人[80]的几种基于预训练语言模型的设计模式检测方法外, 现有的机器学习的设计模式检测技术大多严重依赖于人工特征提取和特征选择, 而找出全面且有效的特征对于设计模式识别问题是一个非常困难的任务, 这给机器学习的设计模式检测技术的推广和发展造成障碍. 深度学习是机器学习中一种基于对数据进行表征学习的方法, 具有强大的自动特征提取能力. 目前有学者探索了深度学习的设计模式检测技术, 例如 Dwivedi 等人[50]、Ferenc 等人[53]、Alhusain 等人[57]训练设计模式分类的 FCNN; Thaller 等人[66]提出了基于 CNN 的设计模式检测方法; Dwivedi 等人[52]、Latif 等人[54]在设计模式检测中引入了 LRNN 模型; Ardimento 等人[61]使用图神经网络技术检测设计模式. 但这些方法仍然是首先从系统中提取结构、行为、面向对象度量等方面的特征, 然后将这些特征输入到深度学习模型中进行学习, 未能充分发挥深度学习的强大的特征自动提取能力和端到端的处理能力. 目前直接从系统的源代码或设计模型到设计模式检测结果的端到端的检测技术还非常少, 是该领域的一个亟待解决的问题.

5.1.8 每种技术中设计模式和待检测系统的表示形式通常单一

当前设计模式检测技术采用了各种各样的数据类型和数据形式来表示设计模式或待检测系统, 例如文本形式的源代码、有向图或图神经网络、UML 模型、AST、源代码的面向对象度量等. 这些不同的表示形式各自保留了待分析系统不同层面的结构、行为和语义等方面的信息. 然而, 每种具体的检测技术通常依赖于单一的表示形式, 未能充分结合多种表示形式来全面反映系统的特征, 这限制了设计模式检测的全面性和精确性. 多模态融合在设计模式检测中的应用潜力巨大, 但其实现面临多方面的技术和实践困难. 例如, 对不同表示形式数据进行融合的技术较为复杂, 多模态融合需要结合静态分析、动态分析、自然语言处理等多种领域的知识从而对研究人员的要求较高, 以及获取和处理多种表示形式的数据需要大量的资源和时间等. 因此, 目前还没有学者尝试将这些不同表示形式进行多模态融合来提升检测效果.

5.1.9 架构级模式检测的研究处于起步阶段且难度更大

当前设计模式检测技术主要是针对 GoF 设计模式. GoF 设计模式主要用于代码的具体实现层面, 帮助开发者

解决具体的编程问题, 优化代码结构和设计. 还有一类非常重要的设计模式侧重于软件的高级结构和组织, 称为架构设计模式. 这类设计模式常用于定义整个应用程序的基础结构, 帮助开发者合理组织代码和功能模块, 使得软件易于扩展、维护和测试. 随着软件系统规模和复杂度的增加, 架构级设计模式 (如 MVC、MVP、三层架构等) 在软件设计和开发中扮演着越来越重要的角色. 现代软件开发中, 对架构级模式的检测和验证需求增加. 近几年陆续有学者从事架构级模式检测的研究[65,71,76,77,80]. 架构级设计模式检测的难度更大, 因为这些模式涉及系统级别的结构和行为, 具有更高的抽象层次和复杂度. 而目前架构级模式检测的研究仍处于起步阶段, 可供参考和借鉴的资料、文献和工具都还非常少, 给研究者造成一定的困扰的同时提供了新的研究机会, 促使他们探索新的方法和技术来解决这些挑战.

5.2 未来研究方向

根据上述主要问题与挑战, 作者认为今后该领域的研究可以从以下几个方向展开.

(1) 目前各个文献自行构建的数据集质量参差不齐, 规模往往比较小, 也没有统一的标准和规范, 这给分类器的训练和评估以及不同文献之间的对比带来困难, 在一定程度上阻碍了机器学习的设计模式检测技术的研究和发展. 鉴于此, 构建并共享规模较大质量较高的数据集具有非常重要的意义. 这将弥补当前学术界和工程界缺乏该领域公开数据集的空白, 吸引更多的学者加入机器学习的设计模式检测技术的研究, 提升设计模式检测技术的研究和应用水平. 可以通过前文所述人工判断和标记、基于公开的模式实例库、借助现有的设计模式检测工具以及这些方式的相互组合来构建数据集, 数据集可以是源代码、UML 图、特征向量等形式. 可以将构建的数据集公开到 GitHub、Kaggle、UC Irvine Machine Learning Repository 等平台, 以便更多研究者能够使用和改进这些数据集. 此外可以考虑研发数据集构建和标注的自动化工具, 降低数据集构建的成本和难度. 这些工具可以通过集成现有的设计模式检测工具, 自动从互联网上爬取设计模式实例, 自动从公式模式实例库提取实例等方式初步标注设计模式实例, 并应提供可视化界面, 方便专家进行审查和校正. 自动化工具的应用将大大提高数据集构建的效率, 数据集的质量也会显著提升.

(2) 动态行为的捕获对于设计模式检测, 尤其是行为型设计模式的检测至关重要. 当前, 由于捕获动态行为的过程复杂且成本较高, 许多研究依赖于静态特征分析. 然而, 静态特征无法全面反映系统在运行时的行为, 限制了设计模式检测的准确性和全面性. 成功捕获待检测系统的动态行为并表示为适合检测算法和模型处理的形式是提升行为型模式检测效果的关键所在. 未来的研究可以尝试借助 JUnit、TestNG、Pytest 等单元测试框架执行待检测系统, 并通过日志记录或运行时监控工具捕获动态行为. 单元测试在运行过程中可以执行特定的代码片段, 触发特定的行为, 包括方法调用、对象状态变化和消息传递等, 这些是设计模式检测特别是行为型设计模式检测所需要的关键信息. 此外, 单元测试具有高覆盖率、自动化、低侵入性和丰富的行为数据捕获能力等优势, 使其成为设计模式检测中动态行为获取的有效手段. 可以使用单元测试用例自动生成技术[106−111], 以减少编写测试用例的工作量. 在此基础上, 可以考虑将获取到的动态行为表示为方法调用序列, 日志数据, 方法调用图 (call graphs), UML 中的对象交互图 (object interaction diagrams)、状态图 (state diagrams)、活动图 (activity diagrams)、序列图 (sequence diagrams) 以及上述多种形式的混合等检测算法和模型适合处理的静态数据形式. 预训练语言模型不擅长处理非文本形式的数据. 对于预训练语言模型, 可将源代码动态行为表示为文本形式的方法调用序列或日志数据, 从而可以输入到模型中进行处理.

(3) 随着软件工程领域的不断发展, 新的设计模式和尚未总结的优秀设计实践不断涌现. 现有的设计模式检测技术主要集中在已经明确定义的 GoF 设计模式或 MVC、MVP、MVVM 等常见的架构级模式, 而对于新的或尚未总结的模式, 识别和检测仍然面临重大挑战. k-Means、基于密度的噪声应用空间聚类 (density-based spatial clustering of applications with noise, DBSCAN) 等聚类算法属于无监督学习算法, 不需要预先定义的类别标签. 它们通过对数据本身的相似性进行分析, 将数据自动分为不同的聚类. 这一特性使得聚类算法可以在没有先验知识的情况下发现新的模式, 而不仅仅局限于已经明确定义的模式. 未来研究可以使用聚类算法将代码段分组成不同的类别, 分析聚类中心的特征向量, 归纳每个聚类所代表的设计模式特征, 从而可以检测除已经明确定义的模式以外

更多的模式. 此外, 还可以通过关联规则挖掘 (例如 Apriori 算法、FP-Growth 算法)、频繁子图挖掘 (例如 gSpan 算法、Subdue 算法) 技术, 发现代码中频繁出现的结构和行为特征. 将这些频繁特征进行归纳和总结, 预期也可以识别出新的或尚未总结的设计模式.

(4) 现有的预训练语言模型架构和预训练方式主要关注文本中词与词之间的关联, 在挖掘源代码不同层级结构的语义方面有所欠缺. 为此, 需要对现有的预训练语言模型架构或预训练方式进行扩展, 在基础的预训练语言模型中引入专门针对源代码特有的结构化和层次化特性的设计, 以适应源代码处理问题. 在模型架构方面, 可以将现有的关注词级语义的模型扩展到关注词、代码行、函数和类等不同层次. 在训练方式方面也可以进行类似改进, 例如可以将现有的掩码语言模型的掩蔽单词或短语扩展到掩蔽代码行、掩蔽函数、掩蔽类, 将下一句预测任务扩展到下一代码行预测任务、下一个函数预测任务、下一个类预测任务, 而将自回归语言模型的下一个 token 预测任务扩展为更高级的层次化预测任务. 这将帮助模型更好地理解和捕捉源代码中的复杂语义关系, 提升设计模式检测的准确率和效率, 为使用预训练语言模型解决设计模式检测问题提供新的发展方向.

(5) 目前设计模式检测支撑工具存在不完善或未公开的问题, 这限制了设计模式检测技术的广泛应用和进一步研究. 设计模式检测课题与软件企业生产实际息息相关, 企业的软硬件和项目源代码等资源可以为课题的研究提供支撑, 课题的研究成果也可以应用于企业. 因此, 通过产学研合作等方式研发完善的设计模式检测工具并进行成果转化, 使得设计模式检测技术从实验室走向软件产业, 是值得广大软件行业的研究者和实践者努力的方向. 除此之外, 可以考虑建立设计模式检测工具开放发布平台, 鼓励研究人员和开发者公开工具代码和文档. 在此基础上, 可尝试搭建开源合作平台和研发联盟, 建立社区合作机制, 组织定期研讨会和合作项目, 鼓励研究人员和开发者共同开发和改进工具, 以此促进知识共享和社区合作.

(6) 研究如何一次性从整个待检测系统中寻找多个设计模式实例并定位是很有意义的方向. 作者认为将待检测系统的源代码或设计模型转化为图像形式, 然后使用 YOLO 系列、R-CNN 系列等计算机视觉领域的目标检测算法进行设计模式的检测, 可能是解决该问题的一个有效思路. 此外, 可以尝试将 YOLO 系列、R-CNN 系列等目标检测算法扩展到文本检测中, 研发针对文本的目标检测算法, 从而使其能够应用于源代码形式的待检测系统.

(7) 未来研究可以探索从系统源代码或设计模型到设计模式的端到端检测方法, 以减少对人工特征提取和选择的依赖, 提高检测效率和准确性. 端到端设计模式检测方法可以利用深度学习模型的强大表示能力, 从源代码或设计模型中自动学习并提取设计模式相关特征. 卷积循环神经网络 (convolutional recurrent neural network, ConvRNN) [112]、深度双向长短期记忆网络 (deep/stacked bidirectional LSTM, DB-LSTM) [113]、Transformer [104]对于文本形式的时序数据具有很好的处理能力, 可以尝试借助这些具有强大特征学习能力的深度学习网络模型自动提取文本形式源代码的特征, 实现从系统源代码到设计模式的端到端检测. 此外, 可以考虑将待检测系统的源代码或设计模型转换为图像形式, 借助 CNN、Vision Transformer [114]、生成对抗网络 (generative adversarial network, GAN)、CNN 与 RNN 网络结合的模型 ConvLSTM[115]等适用于图像处理的深度学习模型构建从系统设计模型到设计模式的端到端检测方法和工具.

(8) 随着软件系统复杂性的增加和对精准检测需求的提升, 将文本形式的源代码、有向图和图神经网络、AST、UML 模型等多种数据形式和类型进行多模态融合以充分利用待检测系统的各类信息变得越来越有必要. 通过多表示形式的融合, 设计模式检测技术能够更全面地捕捉系统的多层次多角度特征和信息, 提升检测的综合能力和准确性. 可以从数据融合和特征融合两个层次进行多模态融合. 在数据融合层次上, 首先将不同类型的数据转换为统一的表示形式 (例如 Token 序列或嵌入向量), 然后使用联合嵌入、协同训练、多模态注意力机制等多模态数据融合技术从数据的级别进行融合. 在特征融合层次上, 首先从不同的模态表示中提取特征向量, 然后通过将不同特征向量直接拼接, 根据特征的重要性对特征进行加权平均, 或使用深度学习模型 (如自编码器) 进行高级特征融合等方式, 将这些特征向量进行融合.

(9) 除了 GoF 设计模式外, 架构级设计模式在软件开发中也扮演越来越重要的角色. 未来研究可以着眼于 MVC、MVM、MVVM、三层架构、EDA 等系统架构级模式的检测方法和技术, 帮助软件开发者和维护者理解进而重构和优化系统架构. 一种可行的方案是将前文所述针对 GoF 设计模式的检测方法和工具扩展和改进到系

统架构级模式的检测, 可以从检测规则、数据集、源代码分析技术等方面来适应架构级模式检测. 另一种方案是针对架构级模式更高的抽象层次和复杂度, 重新构思检测思路和框架, 研发全新的专用于架构级模式检测的技术和工具.

6 结束语

设计模式是软件工程中用以解决特定设计问题的最佳实践, 通过复用这些模式, 可以显著增强软件系统的可维护性和可扩展性. 自动化设计模式检测技术能够自动识别和恢复软件系统中的设计模式, 帮助开发和维护人员理解系统的设计思路, 提高软件质量和开发效率.

本文概述了软件设计模式检测领域的发展历程、检测对象、特征类型和评估指标, 总结了设计模式检测技术分类的现有方法并引出本文根据设计模式检测技术发展的时间线的分类方法. 本文从非机器学习的设计模式检测、机器学习的设计模式检测和基于预训练语言模型的设计模式检测三大类方法出发讨论了设计模式检测技术的研究现状, 给出了实际项目应用案例, 并对当前研究成果进行了总结和比较. 在此基础上归纳了该领域存在的主要问题与挑战并展望了未来研究方向.

本文沿着时间线分类和讨论设计模式检测技术, 全面综述了该领域的发展历程和当前趋势, 尤其强调了基于预训练语言模型的前沿技术, 为解决该领域面临的挑战和推动未来发展提供了新的视角和思路.

References:

[1] Gamma E, Helm R, Johnson R, Vlissides J. Design Patterns: Elements of Reusable Object-oriented Software. Boston: Addison-Wesley, 1995.

[2] Naghdipour A, Hasheminejad SMH, Keyvanpour MR. DPSA: A brief review for design pattern selection approaches. In: Proc. of the 26th Int'l Computer Conf., Computer Society of Iran. Tehran: IEEE, 2021. 1–6. [doi: 10.1109/CSICC52343.2021.9420629]

[3] Naghdipour A, Hasheminejad SMH, Barmaki RL. Software design pattern selection approaches: A systematic literature review. Software: Practice and Experience, 2023, 53(4): 1091–1122. [doi: 10.1002/spe.3176]

[4] Pressman RS, Maxim BR. Software Engineering: A Practitioner's Approach. 9th ed., New York: McGraw-Hill Companies, 2019.

[5] Shilintsev D, Dlamini G. A study: Design patterns detection approaches and impact on software quality. In: Proc. of the 1st Int'l Conf. on Frontiers in Software Engineering. Innopolis: Springer, 2021. 84–96. [doi: 10.1007/978-3-030-93135-3_6]

[6] Wedyan F, Abufakher S. Impact of design patterns on software quality: A systematic literature review. IET Software, 2020, 14(1): 1–17. [doi: 10.1049/iet-sen.2018.5446]

[7] Robles-Aguilar A, Ocharán-Hernández JO, Sánchez-Garcia ÁJ, Limón X. Software design and artificial intelligence: A systematic mapping study. In: Proc. of the 9th Int'l Conf. in Software Engineering Research and Innovation. San Diego: IEEE, 2021. 132–141. [doi: 10.1109/CONISOFT52520.2021.00028]

[8] Jia ZJ, Zhong CX, Zhou SQ, Rong GP, Zhang C. Benefits and challenges of domian driven design patterns: Systematic review. Ruan Jian Xue Bao/Journal of Software, 2021, 32(9): 2642–2664 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6275.htm [doi: 10.13328/j.cnki.jos.006275]

[9] Zhang H, Liu JB. Research review of design pattern mining. In: Proc. of the 11th Int'l Conf. on Software Engineering and Service Science. Beijing: IEEE, 2020. 339–342. [doi: 10.1109/ICSESS49938.2020.9237742]

[10] Yarahmadi H, Hasheminejad SMH. Design pattern detection approaches: A systematic review of the literature. Artificial Intelligence Review, 2020, 53(8): 5789–5846. [doi: 10.1007/s10462-020-09834-5]

[11] Chaturvedi A. Present approaches for detection of design pattern: A survey. Int'l Journal of Computer Sciences and Engineering, 2018, 6(8): 948–958. [doi: 10.26438/ijcse/v6i8.948958]

[12] Al-Obeidallah MG, Petridis M, Kapetanakis S. A survey on design pattern detection approaches. Int'l Journal of Software Engineering, 2016, 7(3): 41–59.

[13] Mhawish MY, Gupta M. A review and classification of design pattern detection techniques. In: Proc. of the Advances in Computing, Control and Communication Technology. New Delhi: Allied Publishers, 2016. 233–239.

[14] Rasool G, Streitfdert D. A survey on design pattern recovery techniques. Int'l Journal of Computer Science Issues, 2011, 8(6): 251–260.

[15] Dong J, Zhao YJ, Peng T. A review of design pattern mining techniques. Int'l Journal of Software Engineering and Knowledge

Engineering, 2009, 19(6): 823–855. [doi: 10.1142/S021819400900443X]

[16] Priya RK. A survey: Design pattern detection approaches with metrics. In: Proc. of the 2014 IEEE National Conf. on Emerging Trends in New & Renewable Energy Sources and Energy Management. Chennai: IEEE, 2014. 22–26. [doi: 10.1109/NCETNRESEM.2014.7088733]

[17] Wang ZG, Wang L. Survey on research of automatic design pattern detection. Journal of Beijing University of Posts and Telecommunications, 2018, 41(4): 119–124 (in Chinese with English abstract). [doi: 10.13190/j.jbupt.2018-034]

[18] Wang L. Design pattern detection based on similarity scoring, FSM and machine learning [Ph.D. Thesis]. Beijing: China University of Mining & Technology (Beijing), 2019 (in Chinese with English abstract). [doi: 10.27624/d.cnki.gzkbu.2019.000034]

[19] Krämer C, Prechelt L. Design recovery by automated search for structural design patterns in object-oriented software. In: Proc. of the 4th Working Conf. on Reverse Engineering. Monterey: IEEE, 1996. 1–9. [doi: 10.1109/WCRE.1996.558905]

[20] Brown K. Design reverse-engineering and automated design-pattern detection in Smalltalk. Technology Report, Raleigh: North Carolina State University at Raleigh, 1996.

[21] Agilemanifesto. Manifesto for agile software development. 2021. https://agilemanifesto.org/

[22] Saixi Consulting. The evolutionary history of agile (Part One). 2022-12-12. https://baijiahao.baidu.com/s?id=1751992955177204890&wfr=spider&for=pc, (in Chinese).

[23] Yu DJ, Zhang P, Yang JZ, Chen ZL, Liu CF, Chen J. Efficiently detecting structural design pattern instances based on ordered sequences. Journal of Systems and Software, 2018, 142: 35–56. [doi: 10.1016/j.jss.2018.04.015]

[24] Yu DJ, Zhang YY, Chen ZL. A comprehensive approach to the recovery of design pattern instances based on sub-patterns and method signatures. Journal of Systems and Software, 2015, 103: 1–16. [doi: 10.1016/j.jss.2015.01.019]

[25] Mayvan BB, Rasoolzadegan A. Design pattern detection based on the graph theory. Knowledge-based Systems, 2017, 120: 211–225. [doi: 10.1016/j.knosys.2017.01.007]

[26] Oruc M, Akal F, Sever H. Detecting design patterns in object-oriented design models by using a graph mining approach. In: Proc. of the 4th Int'l Conf. in Software Engineering Research and Innovation. Puebla: IEEE, 2016. 115–121. [doi: 10.1109/CONISOFT.2016.26]

[27] Jia ZP, Yang XY, Cao Z. Design pattern recognition: US, 16703041. 2024-03-14.

[28] Xu HB, Zhang XL, Zheng XM, Zhang T, Li XD. UML design pattern recognition method based on structured query. Computer Science, 2014, 41(11): 50–55 (in Chinese with English abstract). [doi: 10.11896/j.issn.1002-137X.2014.11.011]

[29] Lu RZ, Zhang HP. Research on design pattern mining based on machine learning. Computer Engineering and Applications, 2019, 55(6): 119–119 (in Chinese with English abstract). [doi: 10.3778/j.issn.1002-8331.1712-0053]

[30] Dong J, Sun YT, Zhao YJ. Design pattern detection by template matching. In: Proc. of the 2008 ACM Symp. on Applied Computing. Ceara: ACM, 2008. 765–769. [doi: 10.1145/1363686.1363864]

[31] Dong J, Zhao YJ, Sun YT. A matrix-based approach to recovering design patterns. IEEE Trans. on Systems, Man, and Cybernetics — Part A: Systems and Humans, 2009, 39(6): 1271–1282. [doi: 10.1109/TSMCA.2009.2028012]

[32] Tsantalis N, Chatzigeorgiou A, Stephanides G, Halkidis S. Design pattern detection using similarity scoring. IEEE Trans. on Software Engineering, 2006, 32(11): 896–909. [doi: 10.1109/TSE.2006.112]

[33] Dewangan S, Rao RS. Design pattern detection by using correlation feature selection technique. In: Proc. of the 11th Int'l Conf. on Communication Systems and Network Technologies. Indore: IEEE, 2022. 641–645. [doi: 10.1109/CSNT54456.2022.9787569]

[34] Dewangan S, Rao RS. Design pattern detection by using cosine similarity technique. In: Proc. of the 6th Int'l Conf. on Computing Communication and Automation. Arad: IEEE, 2021. 171–175. [doi: 10.1109/ICCCA52192.2021.9666209]

[35] Bernardi ML, Cimitile M, De Ruvo G, Di Lucca GA, Santone A. Model checking to improve precision of design pattern instances identification in OO systems. In: Proc. of the 10th Int'l Joint Conf. on Software Technologies. Colmar: IEEE, 2015. 53–63.

[36] De Lucia A, Deufemia V, Gravino C, Risi M. Improving behavioral design pattern detection through model checking. In: Proc. of the 14th European Conf. on Software Maintenance and Reengineering. Madrid: IEEE, 2010. 176–185. [doi: 10.1109/CSMR.2010.16]

[37] De Lucia A, Deufemia V, Gravino C, Risi M. Design pattern recovery through visual language parsing and source code analysis. Journal of Systems and Software, 2009, 82(7): 1177–1193. [doi: 10.1016/j.jss.2009.02.012]

[38] Wendehals L, Orso A. Recognizing behavioral patterns atruntime using finite automata. In: Proc. of the 2006 Int'l Workshop on Dynamic Analysis. Shanghai: ACM, 2006. 33–40. [doi: 10.1145/1138912.1138920]

[39] Al-Obeidallah M, Petridis M, Kapetanakis S. MLDA: A multiple levels detection approach for design patterns recovery. In: Proc. of the Int'l Conf. on Compute and Data Analysis. Lakeland: ACM, 2017. 33–40. [doi: 10.1145/3093241.3093244]

[40] Shi NJ, Olsson RA. Reverse engineering of design patterns from java source code. In: Proc. of the 21st IEEE/ACM Int'l Conf. on

Automated Software Engineering. Tokyo: IEEE, 2006. 123–134. [doi: 10.1109/ASE.2006.57]

[41] Hayashi S, Katada J, Sakamoto R, Kobayashi T, Saeki M. Design pattern detection by using meta patterns. IEICE Trans. on Information and Systems, 2008, E91(4): 933–944. [doi: 10.1093/ietisy/e91-d.4.933]

[42] Di Martino B, Esposito A. A rule-based procedure for automatic recognition of design patterns in UML diagrams. Software: Practice and Experience, 2016, 46(7): 983–1007. [doi: 10.1002/spe.2336]

[43] Luitel G, Stephan M, Inclezan D. Model level design pattern instance detection using answer set programming. In: Proc. of the 8th Int'l Workshop on Modeling in Software Engineering. Austin: ACM, 2016. 13–19. [doi: 10.1145/2896982.2896991]

[44] Thongrak M, Vatanawood W. Detection of design pattern in class diagram using ontology. In: Proc. of the 2014 Int'l Computer Science and Engineering Conf. Khon Kaen: IEEE, 2014. 97–102. [doi: 10.1109/ICSEC.2014.6978176]

[45] Panich A, Vatanawood W. Detection of design patterns from class diagram and sequence diagrams using ontology. In: Proc. of the 15th Int'l Conf. on Computer and Information Science. Okayama: IEEE, 2016. 1–6. [doi: 10.1109/ICIS.2016.7550771]

[46] Fawareh HJ, Alshira'h M. Detection a design pattern through merge static and dynamic analysis using altova and lambdes tools. Int'l Journal of Applied Engineering Research, 2017, 12(19): 8518–8522.

[47] Mokaddem CE, Sahraoui H, Syriani E. A generic approach to detect design patterns in model transformations using a string-matching algorithm. Software and Systems Modeling, 2022, 21(3): 1241–1269. [doi: 10.1007/s10270-021-00936-4]

[48] Chaturvedi A, Gupta M, Kumar S. Design pattern detection using genetic algorithm for sub-graph isomorphism to enhance software reusability. Int'l Journal of Computer Applications, 2016, 135(4): 33–36. [doi: 10.5120/ijca2016908334]

[49] Lecun Y, Bengio Y, Hinton G. Deep learning. Nature, 2015, 521(7553): 436–444. [doi: 10.1038/nature14539]

[50] Dwivedi AK, Tirkey A, Rath SK. Applying learning-based methods for recognizing design patterns. Innovations in Systems and Software Engineering, 2019, 15(2): 87–100. [doi: 10.1007/s11334-019-00329-3]

[51] Dwivedi AK, Tirkey A, Rath SK. Software design pattern mining using classification-based techniques. Frontiers of Computer Science, 2018, 12(5): 908–922. [doi: 10.1007/s11704-017-6424-y]

[52] Dwivedi AK, Tirkey A, Ray RB, Rath SK. Software design pattern recognition using machine learning techniques. In: Proc. of the 2016 IEEE Region 10 Conf. Singapore: IEEE, 2016. 222–227. [doi: 10.1109/TENCON.2016.7847994]

[53] Ferenc R, Fülöp L, Lele J. Design pattern mining enhanced by machine learning. In: Proc. of the 21st IEEE Int'l Conf. on Software Maintenance. Budapest: IEEE, 2005. 295–304.

[54] Latif S, Qureshi MM, Mehmmod M. Detection and recognition of software design patterns based on machine learning techniques: A big step towards software design re-usability. In: Proc. of the 1st Int'l Conf. on Engineering Software for Modern Challenges. Johor: Springer, 2022. 3–15. [doi: 10.1007/978-3-031-19968-4_1]

[55] Chihada A, Jalili S, Hasheminejad SMH, Zangooei MH. Source code and design conformance, design pattern detection from source code by classification approach. Applied Soft Computing, 2015, 26: 357–367. [doi: 10.1016/j.asoc.2014.10.027]

[56] Chaturvedi S, Chaturvedi A, Tiwari A, Agarwal S. Design pattern detection using machine learning techniques. In: Proc. of the 7th Int'l Conf. on Reliability, Infocom Technologies and Optimization. Noida: IEEE, 2018. 1–6. [doi: 10.1109/ICRITO.2018.8748282]

[57] Alhusain S, Coupland S, John R, Kavanagh M. Towards machine learning based design pattern recognition. In: Proc. of the 13th UK Workshop on Computational Intelligence. Guildford: IEEE, 2013. 244–251. [doi: 10.1109/UKCI.2013.6651312]

[58] Uchiyama S, Washizaki H, Fukazawa Y, Kubo A. Design pattern detection using software metrics and machine learning. In: Proc. of the 1st Int'l Workshop on Model-driven Software Migration. Oldenburg, 2011. 38–47.

[59] Uchiyama S, Kubo A, Washizaki H, Fukazawa Y. Detecting design patterns in object-oriented program source code by using metrics and machine learning. Journal of Software Engineering and Applications, 2014, 7(12): 983–998. [doi: 10.4236/jsea.2014.712086]

[60] Barbudo R, Ramírez A, Servant F, Romero JR. GEML: A grammar-based evolutionary machine learning approach for design-pattern detection. Journal of Systems and Software, 2021, 175: 110919. [doi: 10.1016/j.jss.2021.110919]

[61] Ardimento P, Aversano L, Bernardi ML, Cimitile M. Design patterns mining using neural sub-graph matching. In: Proc. of the 37th ACM/SIGAPP Symp. on Applied Computing. ACM, 2022. 1545–1553. [doi: 10.1145/3477314.3507073]

[62] Dong J, Sun YT, Zhao YJ. Compound record clustering algorithm for design pattern detection by decision tree learning. In: Proc. of the 2008 IEEE Int'l Conf. on Information Reuse and Integration. Las Vegas: IEEE, 2008. 1–6. [doi: 10.1109/IRI.2008.4583034]

[63] Zanoni M, Fontana FA, Stella F. On applying machine learning techniques for design pattern detection. Journal of Systems and Software, 2015, 103: 102–117. [doi: 10.1016/j.jss.2015.01.037]

[64] Chand S, Pandey SK, Horkoff J, Staron M, Ochodek M, Durisic D. Comparing word-based and AST-based models for design pattern recognition. In: Proc. of the 19th Int'l Conf. on Predictive Models and Data Analytics in Software Engineering. San Francisco: ACM,

2023. 44–48. [doi: 10.1145/3617555.3617873]

[65] Wang F, Qiu XY, Bian WW, Xin ZF. Research on architecture design pattern mining of complex information system. In: Proc. of SPIE 12160, Int'l Conf. on Computational Modeling, Simulation, and Data Analysis. Sanya: SPIE, 2022. 1–6. [doi: 10.1117/12.2627670]

[66] Thaller H, Linsbauer L, Egyed A. Feature maps: A comprehensible software representation for design pattern detection. In: Proc. of the 26th Int'l Conf. on Software Analysis, Evolution and Reengineering. Hangzhou: IEEE, 2019. 207–217. [doi: 10.1109/SANER.2019.8667978]

[67] Mhawish MY, Gupta M. Software metrics and tree-based machine learning algorithms for distinguishing and detecting similar structure design patterns. SN Applied Sciences, 2020, 2(1): 11. [doi: 10.1007/s42452-019-1815-3]

[68] Feng T, Jin L, Zhang JC, Wang HY. Design pattern detection approach based on stacked generalization. Ruan Jian Xue Bao/Journal of Software, 2020, 31(6): 1703–1722 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5847.htm [doi: 10.13328/j.cnki.jos.005847]

[69] Nazar N, Aleti A, Zheng YK. Feature-based software design pattern detection. Journal of Systems and Software, 2022, 185: 111179. [doi: 10.1016/j.jss.2021.111179]

[70] Hamama M. Detecting design patterns by learning embedded code features [Ph.D. Thesis]. Tralee: Munster Technological University, 2021.

[71] Komolov S, Dlamini G, Megha S, Mazzara M. Towards predicting architectural design patterns: A machine learning approach. Computers, 2022, 11(10): 151. [doi: 10.3390/computers11100151]

[72] Devlin J, Chang MW, Lee K, Toutanova K. BERT: Pre-training of deep bidirectional transformers for language understanding. In: Proc. of the 2019 Conf. of the North American Chapter of the Association for Computational Linguistics. Minneapolis: ACL, 2019. 4171–4186. [doi: 10.18653/v1/N19-1423]

[73] OpenAI. GPT-4 technical report. 2023. https://cdn.openai.com/papers/gpt-4.pdf

[74] Radford A, Narasimhan K, Salimans T, Sutskever I. Improving language understanding by generative pre-training. 2018. https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf

[75] Lewis M, Liu YH, Goyal N, Ghazvininejad M, Mohamed A, Levy O, Stoyanov V, Zettlemoyer L. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In: Proc. of the 58th Annual Meeting of the Association for Computational Linguistics. Association for Computational Linguistics, 2020. 7871–7880. [doi: 10.18653/v1/2020.acl-main.703]

[76] Jánki ZR, Bilicki V. Rule-based architectural design pattern recognition with GPT models. Electronics, 2023, 12(15): 3364. [doi: 10.3390/electronics12153364]

[77] Dlamini G, Ahmad U, Kharkrang LR, Ivanov V. Detecting design patterns in android applications with CodeBERT embeddings and CK metrics. In: Proc. of the 11th Int'l Conf. on Analysis of Images, Social Networks and Texts. Yerevan: Springer, 2024. 267–280. [doi: 10.1007/978-3-031-54534-4_19]

[78] Pandey SK, Staron M, Horkoff J, Ochodek M, Mucci N, Durisic D. TransDPR: Design pattern recognition using programming language model. In: Proc. of the 2023 ACM/IEEE Int'l Symp. on Empirical Software Engineering and Measurement. New Orleans: IEEE, 2023. 1–7. [doi: 10.1109/ESEM56168.2023.10304862]

[79] Chen SF, Liu D, Jiang H. CodeBERT-based language model for design patterns. Computer Science, 2023, 50(12): 75–81 (in Chinese with English abstract). [doi: 10.11896/jsjkx.230100115]

[80] Parthasarathy D, Ekelin C, Karri A, Sun JP, Moraitis P. Measuring design compliance using neural language models: An automotive case study. In: Proc. of the 18th Int'l Conf. on Predictive Models and Data Analytics in Software Engineering. Singapore: Association for Computing Machinery, 2022. 12–21. [doi: 10.1145/3558489.3559067]

[81] Zhao S, Chen SH. Review: Traffic identification based on machine learning. Computer Engineering & Science, 2018, 40(10): 1746–1756 (in Chinese with English abstract). [doi: 10.3969/j.issn.1007-130X.2018.10.005]

[82] Chen ZY, Xu B, Wu Y, Wu KW. Overview of research on attention mechanism in medical image processing. Computer Engineering and Applications, 2022, 58(5): 23–33 (in Chinese with English abstract). [doi: 10.3778/j.issn.1002-8331.2108-0266]

[83] Maggioni S. Design pattern detection and software architecture reconstruction: An integrated approach based on software micro-structures [Ph.D. Thesis]. Milan: Milan Bicocca University, 2009.

[84] Wang L, Wang WF, Song HN, Zhang S. Design pattern recognition based on similarity scoring and secondary subsystems. Computer Engineering, 2023, 49(1): 210–222 (in Chinese with English abstract). [doi: 10.19678/j.issn.1000-3428.0063345]

[85] Wang L, Song T, Song HN, Zhang S. Research on design pattern detection method based on UML model with extended image

information and deep learning. Applied Sciences, 2022, 12(17): 8718. [doi: 10.3390/app12178718]

[86] Wang L, Song HN, Wang WF. A design pattern detection method based on similarity scoring. Journal of Hunan University (Natural Sciences), 2019, 46(12): 50–57 (in Chinese with English abstract). [doi: 10.16339/j.cnki.hdxbzkb.2019.12.007]

[87] Liu QX, Chen YH, NI JS, Luo C, Liu CY, Cao YQ, Tan R, Feng Y, Zhang Y. Survey on machine learning-based anomaly detection for industrial internet. Journal of Computer Research and Development, 2022, 59(5): 994–1014 (in Chinese with English abstract). [doi: 10.7544/issn1000-1239.20211147]

[88] Balanyi Z, Ferenc R. Mining design patterns from C++ source code. In: Proc. Int'l Conf. on Software Maintenance. Amsterdam: IEEE, 2003. 305–314. [doi: 10.1109/ICSM.2003.1235436]

[89] Yang YL, Guan XD, You JY. CLOPE: A fast and effective clustering algorithm for transactional data. In: Proc. of the 8th ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining. Edmonton: ACM, 2002. 682–687. [doi: 10.1145/775047.775149]

[90] Mikolov T, Sutskever I, Chen K, Corrado G, Dean J. Distributed representations of words and phrases and their compositionality. In: Proc. of the 26th Int'l Conf. on Neural Information Processing Systems. Lake Tahoe: Curran Associates Inc., 2013. 3111–3119.

[91] Alon U, Zilberstein M, Levy O, Yahav E. code2vec: Learning distributed representations of code. Proc. of the ACM on Programming Languages, 2019, 31(POPL): 40. [doi: 10.1145/3290353]

[92] Guéhéneuc YG, Huynh DL, Straw G. P-MARt: Pattern-like micro architecture repository. In: Proc. of the lst EuroPLoP Focus Group on Pattern Repositories. Ottawa: IEEE, 2007. 1–3.

[93] Fontana FA, Caracciolo A, Zanoni M. DPB: A benchmark for design pattern detection tools. In: Proc. of the 16th European Conf. on Software Maintenance and Reengineering. Szeged: IEEE, 2012. 235–244. [doi: 10.1109/CSMR.2012.32]

[94] Ampatzoglou A, Michou O, Stamelos I. Building and mining a repository of design pattern instances: Practical and research benefits. Entertainment Computing, 2013, 4(2): 131–142. [doi: 10.1016/j.entcom.2012.10.002]

[95] GitHub. Java-DPD-dataset: A Java-based dataset for design pattern detection. 2024. https://github.com/sdharren/Java-DPD-dataset

[96] Al-Obeidallah MG. A benchmark for design pattern recovery tools. In: Proc. of the 2023 Int'l Conf. on Software and System Engineering. Marseille: IEEE, 2023. 7–11. [doi: 10.1109/ICoSSE58936.2023.00010]

[97] Raffel C, Shazeer N, Roberts A, Lee K, Narang S, Matena M, Zhou YQ, Li W, Liu PJ. Exploring the limits of transfer learning with a unified text-to-text Transformer. The Journal of Machine Learning Research, 2020, 21(1): 140.

[98] Liu YH, Ott M, Goyal N, Du JF, Joshi M, Chen DQ, Levy O, Lewis M, Zettlemoyer L, Stoyanov V. RoBERTa: A robustly optimized BERT pretraining approach. arXiv:1907.11692, 2019.

[99] Song KT, Tan X, Qin T, Lu JF, Liu TY. MASS: Masked sequence to sequence pre-training for language generation. In: Proc. of the 36th Int'l Conf. on Machine Learning. Long Beach: PMLR, 2019. 5926–5936.

[100] Zhang ZY, Han X, Liu ZY, Jiang X, Sun MS, Liu Q. ERNIE: Enhanced language representation with informative entities. In: Proc. of the 57th Annual Meeting of the Association for Computational Linguistics. Florence: ACL, 2019. 1441–1451. [doi: 10.18653/v1/P19-1139]

[101] Sun Y, Wang SH, Li YK, Feng SK, Tian H, Wu H, Wang HF. ERNIE 2.0: A continual pre-training framework for language understanding. In: Proc. of the 34th AAAI Conf. on Artificial Intelligence. New York: AAAI Press, 2020. 8968–8975. [doi: 10.1609/aaai.v34i05.6428]

[102] Feng ZY, Guo DY, Tang DY, Duan N, Feng XC, Gong M, Shou LJ, Qin B, Liu T, Jiang DX, Zhou M. CodeBERT: A pre-trained model for programming and natural languages. In: Proc. of the Findings of the Association for Computational Linguistics. ACL, 2020. 1536–1547. [doi: 10.18653/v1/2020.findings-emnlp.139]

[103] Roziere B, Lachaux MA, Chanussot L, Lample G. Unsupervised translation of programming languages. In: Proc. of the 34th Int'l Conf. on Neural Information Processing Systems. Vancouver: Curran Associates Inc., 2020. 1730.

[104] Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser Ł, Polosukhin I. Attention is all you need. In: Proc. of the 31st Int'l Conf. on Neural Information Processing Systems. Long Beach: Curran Associates Inc., 2017. 6000–6010.

[105] Kitaev N, Kaiser L, Levskaya A. Reformer: The efficient Transformer. In: Proc. of the 8th Int'l Conf. on Learning Representations. Addis Ababa: OpenReview. net, 2020.

[106] Ding R, Dong HB, Zhang Y, Feng XB. Fast automatic generation method for software testing data based on key-point path. Ruan Jian Xue Bao/Journal of Software, 2016, 27(4): 814–827 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4971.htm [doi: 10.13328/j.cnki.jos.004971]

[107] You F, Zhao RL, Lü SS. Output domain based automatic test case generation. Journal of Computer Research and Development, 2016, 53(3): 541–549 (in Chinese with English abstract). [doi: 10.7544/issn1000-1239.2016.20148045]

[108] Albert E, Cabanas I, Flores-Montoya A, Gomez-Zamalloa M, Gutierrez S. jPET: An automatic test-case generator for Java. In: Proc. of the 18th Working Conf. on Reverse Engineering. Limerick: IEEE, 2011. 441–442. [doi: 10.1109/WCRE.2011.67]

[109] Sabanayagam S, Pradhan S, Korukoppula S, Kumar AC. System and a method for automated unit test generation: US, 10949334B2. 2021-03-16.

[110] Tufano M, Drain D, Svyatkovskiy A, Deng SK, Sundaresan N. Unit test case generation with transformers and focal context. arXiv: 2009.05617, 2020.

[111] Schäfer M, Nadi S, Eghbali A, Tip F. Adaptive test generation using a large language model. arXiv:2302.06527, 2023.

[112] Wang RS, Li Z, Cao J, Chen T, Wang L. Convolutional recurrent neural networks for text classification. In: Proc. of the 2019 Int'l Joint Conf. on Neural Networks. Budapest: IEEE, 2019. 1–6. [doi: 10.1109/IJCNN.2019.8852406]

[113] Li RN, Wu ZY, Meng H, Cai LH. DBLSTM-based multi-task learning for pitch transformation in voice conversion. In: Proc. of the 10th Int'l Symp. on Chinese Spoken Language Processing. Tianjin: IEEE, 2016. 1–5. [doi: 10.1109/ISCSLP.2016.7918466]

[114] Dosovitskiy A, Beyer L, Kolesnikov A, Weissenborn D, Zhai XH, Unterthiner T, Dehghani M, Minderer M, Heigold G, Gelly S, Uszkoreit J, Houlsby N. An image is worth 16x16 words: Transformers for image recognition at scale. In: Proc. of the 9th Int'l Conf. on Learning Representations. Vienna: OpenReview. net, 2021. 1–22.

[115] Shi XJ, Chen ZR, Wang H, Yeung DY, Wong WK, Woo WC. Convolutional LSTM network: A machine learning approach for precipitation nowcasting. In: Proc. of the 28th Int'l Conf. on Neural Information Processing Systems. Montreal: MIT Press, 2015. 1802–810.

附中文参考文献:

[8] 贾子甲, 钟陈星, 周世旗, 荣国平, 章程. 领域驱动设计模式的收益与挑战: 系统综述. 软件学报, 2021, 32(9): 2642–2664. http://www.jos.org.cn/1000-9825/6275.htm [doi: 10.13328/j.cnki.jos.006275]

[17] 王智广, 王雷. 设计模式自动识别的研究进展. 北京邮电大学学报, 2018, 41(4): 119–124. [doi: 10.13190/j.jbupt.2018-034]

[18] 王雷. 基于相似度评分、FSM 和机器学习的设计模式识别[博士学位论文]. 北京: 中国矿业大学 (北京), 2019. [doi: 10.27624/d.cnki.gzkbu.2019.000034]

[22] 赛希咨询. 敏捷的发展历史 (一). 2022-12-12. https://baijiahao.baidu.com/s?id=1751992955177204890&wfr=spider&for=pc

[28] 许涵斌, 张学林, 郑晓梅, 张天, 李宣东. 一种基于结构查询的 UML 设计模式识别方法. 计算机科学, 2014, 41(11): 50–55. [doi: 10.11896/j.issn.1002-137X.2014.11.011]

[29] 鲁润泽, 张海平. 应用机器学习方法的设计模式挖掘研究. 计算机工程与应用, 2019, 55(6): 113–119. [doi: 10.3778/j.issn.1002-8331.1712-0053]

[68] 冯铁, 靳乐, 张家晨, 王洪媛. 基于堆叠泛化的设计模式检测方法. 软件学报, 2020, 31(6): 1703–1722. http://www.jos.org.cn/1000-9825/5847.htm [doi: 10.13328/j.cnki.jos.005847]

[79] 陈时非, 刘东, 江贺. 基于 CodeBERT 的设计模式语言模型. 计算机科学, 2023, 50(12): 75–81. [doi: 10.11896/jsjkx.230100115]

[81] 赵双, 陈曙晖. 基于机器学习的流量识别技术综述与展望. 计算机工程与科学, 2018, 40(10): 1746–1756. [doi: 10.3969/j.issn.1007-130X.2018.10.005]

[82] 陈朝一, 许波, 吴英, 吴凯文. 医学图像处理中的注意力机制研究综述. 计算机工程与应用, 2022, 58(5): 23–33. [doi: 10.3778/j.issn.1002-8331.2108-0266]

[84] 王雷, 王文发, 宋慧娜, 张帅. 基于相似度评分与二级子系统的设计模式识别. 计算机工程, 2023, 49(1): 210–222. [doi: 10.19678/j.issn.1000-3428.0063345]

[86] 王雷, 宋慧娜, 王文发. 一种基于相似度评分的设计模式识别方法. 湖南大学学报 (自然科学版), 2019, 46(12): 50–57. [doi: 10.16339/j.cnki.hdxbzkb.2019.12.007]

[87] 刘奇旭, 陈艳辉, 尼杰硕, 罗成, 柳彩云, 曹雅琴, 谭儒, 冯云, 张越. 基于机器学习的工业互联网入侵检测综述. 计算机研究与发展, 2022, 59(5): 994–1014. [doi: 10.7544/issn1000-1239.20211147]

[106] 丁蕊, 董红斌, 张岩, 冯宪彬. 基于关键点路径的快速测试用例自动生成方法. 软件学报, 2016, 27(4): 814–827. http://www.jos.org.cn/1000-9825/4971.htm [doi: 10.13328/j.cnki.jos.004971]

[107] 尤枫, 赵瑞莲, 吕珊珊. 基于输出域的测试用例自动生成方法研究. 计算机研究与发展, 2016, 53(3): 541–549. [doi: 10.7544/issn1000-1239.2016.20148045]

王雷(1988－), 男, 博士, 副教授, CCF 专业会员, 主要研究领域为软件工程, 大数据, 机器学习, 区块链.

王国仁(1966－), 男, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为大数据, 区块链, 生物信息学.

袁野(1981－), 男, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为大数据, 机器学习, 区块链, 数据库.