软件学报 ISSN 1000-9825, CODEN RUXUEW
2025,36(6):2501−2514 [doi: 10.13328/j.cnki.jos.007329] [CSTR: 32375.14.jos.007329]
©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

基于大语言模型的长方法分解*

徐子懋[1], 姜艳杰[2], 张宇霞[1], 刘 辉[1]

[1](北京理工大学 计算机学院, 北京 100081)

[2](北京大学 计算机学院, 北京 100091)

通信作者: 姜艳杰, E-mail: yanjiejiang@pku.edu.cn

摘 要: 长方法及其他类型的代码坏味阻碍了软件应用程序达到最佳的可读性、可重用性和可维护性. 因此, 人们对长方法的自动检测和分解进行了广泛的研究. 虽然这些方法极大地促进了分解, 但其解决方案往往与最优方案存在很大差异. 为此, 调研公开真实长方法数据集中的可自动化部分, 探讨了长方法的分解情况, 并基于调研结果, 提出了一种基于大语言模型的新方法 (称为 Lsplitter), 用于自动分解长方法. 对于给定的长方法, Lsplitter 会根据启发式规则和大语言模型将该方法分解为一系列短方法. 然而, 大语言模型经常会拆分出相似的方法, 针对大语言模型的分解结果, Lsplitter 利用基于位置的算法, 将物理上连续且高度相似的方法合并成一个较长的方法. 最后对这些候选结果进行排序. 对真实 Java 项目中的 2 849 个长方法进行了实验, 结果表明, 相较传统结合模块化矩阵的方法, Lsplitter 的命中率提升了 142%, 相较纯基于大语言模型的方法, 命中率提升了 7.6%.

关键词: 长方法; 重构; 大语言模型; 分解

中图法分类号: TP311

中文引用格式: 徐子懋, 姜艳杰, 张宇霞, 刘辉. 基于大语言模型的长方法分解. 软件学报, 2025, 36(6): 2501–2514. http://www.jos.org.cn/1000-9825/7329.htm

英文引用格式: Xu ZM, Jiang YJ, Zhang YX, Liu H. Large Language Model-Based Decomposition of Long Methods. Ruan Jian Xue Bao/Journal of Software, 2025, 36(6): 2501–2514 (in Chinese). http://www.jos.org.cn/1000-9825/7329.htm

Large Language Model-Based Decomposition of Long Methods

XU Zi-Mao[1], JIANG Yan-Jie[2], ZHANG Yu-Xia[1], LIU Hui[1]

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

[2](School of Computer Science, Peking University, Beijing 100091, China)

Abstract: Long methods, along with other types of code smells, prevent software applications from reaching their optimal readability, reusability, and maintainability. Consequently, automated detection and decomposition of long methods have been widely studied. Although these approaches have significantly facilitated the decomposition, their solutions often differ significantly from the optimal ones. To address this, the automatable portion of the publicly available dataset containing real-world long methods is investigated. Based on the findings of this investigation, a new method (called Lsplitter) based on large language models (LLMs) is proposed in this study for automatically decomposing long methods. For a given long method, the Lsplitter decomposes the method into a series of shorter methods according to heuristic rules and LLMs. However, LLMs often split out similar methods. In response to the decomposition results of LLMs, Lsplitter utilizes a location-based algorithm to merge physically contiguous and highly similar methods into a longer method. Finally, these candidate results are ranked. Experiments are conducted on 2 849 long methods in real Java projects. The experimental results show that compared with the traditional methods combined with a modularity matrix, the hit rate of Lsplitter is improved by 142%, and compared with the methods purely based on LLMs, the hit rate is improved by 7.6%.

* 基金项目: 国家自然科学基金 (62172037, 62232003); 中国博士后科学基金 (2023M740078)
本文由“大模型下的软件质量保障”专题特约编辑王赞教授、王莹副教授、陈碧欢副教授、姚远副教授、张敏灵教授推荐.
收稿时间: 2024-08-26; 修改时间: 2024-10-14; 采用时间: 2024-11-25; jos 在线出版时间: 2024-12-10
CNKI 网络首发时间: 2025-03-19

Key words: long method; refactoring; LLM; decomposition

代码坏味是 Fowler 和 Beck 于 1999 年首次提出的概念[1], 它指的是需要软件重构的设计问题. 如果不及时解决这些问题, 就会影响软件应用程序的可读性、可扩展性和可维护性[2,3]. 目前已确认的代码坏味有几十种, 最常见的代码坏味包括特征依恋、重复代码和长方法. 值得注意的是, 代码坏味不仅是代码质量下降的指标, 也是重构机会的指标[4]. 重构 (refactoring) 由 Opdyke 提出[5], 是在不改变外部行为的情况下对现有代码进行重组. 软件重构的主要目的是通过解决代码坏味来提高源代码的可读性和可维护性[6]. 目前, 大多数主流集成开发环境都为软件重构提供了自动或半自动工具支持.

长方法是公认的代码坏味之一[1], 通常会对代码质量产生严重的负面影响[7]. 一方面, 冗长而复杂的方法难以阅读或修改[8]; 另一方面, 较短的方法往往有更大的复用机会. 因此, 长方法最好分解成几个短方法. 这种分解被称为长方法分解[9,10]. 值得注意的是, 分解一个长方法并不容易, 因为我们必须准确地确定哪些片段应被分解为新方法, 以及我们应从原始方法中分解出多少个方法. 为了促进方法的有效分解, 研究人员提出了多种解决方案[11,12]. 这些解决方案主要分为两大类: 基于代码度量的方法[12–14]和基于图的方法[11,15]. 前者计算待分解方法的数值度量, 并根据度量结果与预定义的启发式方法或人工智能模型决定哪些语句应予以分解. 后者通过从源代码中提取信息生成图形 (如抽象语法树、控制流图和程序依赖图), 并根据生成的图形推理出解决方案. 虽然这些方法大大促进了分解工作, 但它们的解决方案往往与最优方案大相径庭.

大语言模型正在成为各种文本任务 (如语言翻译[16,17]、文本分类[18]和文本摘要[19]) 的有力工具, 其也被成功用于解决一些软件工程任务, 如代码生成[20]、代码总结[21]和自动程序修复[22,23]. 大语言模型具有将冗长复杂的方法分割成较小代码片段的潜力, 因为它们擅长源代码理解[24]. 目前也有人尝试使用大语言模型对长方法进行分解[25,26].

为了进一步探究长方法分解, 本文首先对公开真实重构数据集进行调研, 并进行了总结. 根据调研结果, 提出了长方法分解的解决方案 Lsplitter. 对于给定的长方法, Lsplitter 会根据启发式规则和大语言模型分解得到一系列短方法候选方案. 值得注意的是, 其中使用大语言模型分解时经常会导致不必要的拆分, 产生一些相邻且相似的方法. 为了消除这种相似的方法, Lsplitter 利用基于位置的算法来合并连续且高度相似的方法. 最后会将得到的候选项根据设计的度量值进行排序推荐. 我们对两个 Java 项目中的 2 849 个长方法进行了实验. 实验结果表明, 相较传统方法, Lsplitter 的命中率提升了 142%, 相较纯基于大语言模型的方法, 命中率提升了 7.6%.

本文第 1 节介绍长方法的相关工作和研究现状. 第 2 节对现有长方法分解进行调研. 第 3 节介绍本文提出的基于大语言模型的长方法分解工作. 第 4 节通过对比实验验证了所提方法的有效性. 随后讨论了相关问题并总结全文.

1 相关工作

1.1 长方法分解

长方法分解是指从现有的长方法主体中提取部分代码, 创建一个新方法. 手动分解长方法既繁琐又耗时. 因此, 人们提出了一些自动或半自动的方法来分解长方法. 这些方法主要分为两类: 基于度量的方法和基于图的方法.

有些方法在长方法中计算数值度量. Xu 等人[12]利用方法的复杂性、内聚性和耦合性推出了一种工具 GEMS, 并为这项任务训练了一个机器学习分类器, 取得了很好的效果. Yamanaka 等人[14]扩展了这个工具, 并使用 code2seq 输出的方法名置信度和代码度量来训练机器学习分类器. 与 GEMS 相比, 他们提出的技术准确率更高. Shahidi 等人[13]提出了一种通过计算模块化度量来分解长方法的方案. 他们逐行分析代码, 根据行与行之间的关系形成矩阵, 通过计算聚类系数选择需要提取的代码行, 并根据矩阵值计算重构得分, 从而选择最优解, 他们还根据启发式规则为提取出的新方法命名. Fernandes 等人[27]提出了实时重构的概念, 利用代码质量矩阵实时对代码坏味进行监测并给出重构建议, 改善传统重构工具反馈较慢导致影响开发者体验的问题, 推出了 LiveRef 工具.

其他方法则利用代码生成的图形信息, 如抽象语法树、控制流图和程序依赖图, 基于这些信息对长方法进行

分析并分解. Maruyama 等人[11]提出了一种通过控制流图对长方法进行切分的方法, 分解了控制流图, 并将程序切片区域限制在特定的块内, 从而获得了有效的重构解决方案. 基于这种方法, 他们设计了一种简单的重构工具. Tsantalis 等人[28]对这一方法进行了扩展, 并推出了 JDeodorant 工具, 引入了一种自动化方法, 通过为每个变量选择块来识别与变量的完整计算和对象的状态相关的重构机会. 他们大大推进了自动化重构, 为软件设计和可维护性改进提供了实用工具. Sharma[29]提出了一种名为 DSD(数据与结构依赖图) 的新图, 用于识别分解长方法的候选方案. Silva 等人[30]提出了长方法分解的工具, 该工具按照控制流将代码划分为若干顺序语句块. 对于每个块, 它只选择块内的连续语句, 并考虑最小语句数和可以进行重构作为前提条件; 然后, 它根据结构相似性计算每个候选代码块的得分并根据得分对候选语句进行排序和筛选. 与 JDeodorant 相比, 该工具能更有效地识别方法中连续的代码. Cui 等人[15]提出了一种工具 REMS, 该工具采用了多视图的 CPG (代码属性图) 表示法, 整合了树视图和流视图表示法, 并使用机器学习分类器来识别适合方法分解的代码行.

除此之外, Pomian 等人[25]也尝试使用大语言模型来进行长方法分解指出现有的大语言模型虽然能提供出色的建议, 但是大多数方案并不可行. 他们在大语言模型分解的基础上, 辅以静态分析, 增强了基于大语言模型的长方法分解结果的准确性和可行性.

1.2 长方法检测

长方法的检测一般分为两种类型. 一种方法是基于长方法分解进行, 即对于一个代码片段, 尝试进行长方法分解, 如果有合理方案, 则视为长方法; 反之, 则不是. GEMS[12]、REMS[15]和 Shahidi[13]的方法都使用了这种方法. 另一种是使用代码度量来确定. Marinescu[31]计算了每种方法的 CYCLO(麦凯布循环复杂度) 指标, 并选择最复杂的方法作为长方法. Lanza 等人[32]使用了 4 个代码度量 (LOC (代码行)、CYCLO、MAXNESTING (最大嵌套水平) 和 NOAV (访问变量数)) 来检测长方法, 当这 4 个度量都超过预定义的阈值时, 该方法就被认为是长方法. Moha 等人[33]仅依靠代码长度来检测长方法, 认为任何长度超过预定阈值的方法都是长方法. Yoshida[34]和 Charalampidou 等人[35,36]将 COH (内聚度) 考虑在内. 他们计算 COH 和 LOC, 并确定一个方法是否被检测为长方法.

除了基于启发式的度量检测之外, Liu 等人[37]还提出了一种基于深度学习的长方法检测方法. 他们计算了方法的 LOC、方法缺乏内聚性指标 (LCOM1、LCOM2 和 LCOM4)、COH 和 CC(Class Cohesion), 构建了用于训练深度学习模型的数据集, 并使用模型的检测结果来判断方法是否为长方法. 实验结果表明, 与之前的方法相比, 该方法有了明显改进.

2 长方法分解调研

为了探究长方法的分解情况, 我们调研了 Silva 等人[38]维护的一个公开数据集. 该数据集包含了 448 个 Java 方法, 每个方法都有开发人员实际执行的长方法分解重构, 并按照 Cossette 和 Walker 的分类方法将此数据集分为 3 类[39]: 可自动化的、部分自动化的、不可自动化的. 其中, 可自动化的是直接对方法的分解, 没有其他操作; 部分自动化的是在分解的基础上做了一些简单的修改, 如更改变量名或方法名; 不可自动化是在长方法分解的基础上又增加了其他的代码和功能. 其中, 我们选择了可自动化的部分进行分析, 共包含 154 个长方法分解的重构案例.

我们逐一查看每一组分解前后的代码、GitHub 上的提交记录等, 将这些方法分为以下几类.

(1) 提取整个 block 中的代码作为新方法.

(2) 提取某一个变量的定义和初始化部分或使用部分代码.

(3) 提取代码中某一段包含注释且前后由空行分割的整块代码.

(4) 提取长方法内代码复用的片段.

(5) 不属于重构的提取, 在这个提交中还存在其他与此相关的修改.

(6) 其他类型.

前两类根据代码的语法结构进行长方法提取, 第 (1) 类是将一个完整的代码块 (block) 提取出来, 作为一个新的独立方法. 通常, 这些代码块具有逻辑上的完整性和独立性, 提取后可以提高代码的可读性和可维护性. 第 (2)

类是专门针对某个变量进行的, 提取该变量的定义、初始化或者使用的相关代码. 通过这种方式, 可以减少代码长度, 同时将变量相关的逻辑集中在一起, 提高代码的模块化程度. 第 (3) 类则是根据文本结构进行提取, 基于代码中的注释和空行, 通过这些标记将代码分成若干独立的段落, 然后将这些段落提取为新的方法. 这种方式可以利用开发人员在编写代码时的自然分段, 提高代码的结构化程度. 第 (4) 类则因为在软件原始开发过程中存在重复代码, 通过重构来优化, 提高代码的可读性. 第 (5) 类则通过程序员提交重构代码的整个提交记录进行分析, 这些是一些并非以重构为目的的长方法分解, 程序员可能是想扩展某些功能、复用一些代码, 从而将长方法中的一部分提取出来.

Silva 等人[38]维护的数据集中可自动化的长方法分解各类别方法数量如表 1 所示.

表 1　Silva 等人维护的数据集中可自动化的长方法分解分类情况

类别	数量	比例 (%)
(1) 提取整块代码	20	13.0
(2) 提取变量相关代码	18	11.7
(3) 基于注释和空行的提取	31	20.1
(4) 复用代码的提取	5	3.2
(5) 不属于重构的提取	72	46.8
(6) 其他类型	8	5.2

可以发现, 不属于重构的提取占据了最多的比例, 为 46.8%, 这表明在实际开发中, 程序员在进行长方法分解时, 常常伴随着其他的修改或者开发操作, 而不仅仅是为了重构而操作. 这也说明在实际开发过程中, 代码重构往往与功能扩展、错误修复等操作密切相关, 而很多程序员在开发过程中可能并没有重构代码、解决代码坏味的习惯, 但这些部分实际上是机器很难自动化的. 基于注释和空行的提取次之, 占 20.1%, 这说明开发人员在编写代码时, 常常利用注释和空行进行逻辑分段, 这样在后续的重构过程中就根据这些标记进行长方法分解, 这是一种良好的编写代码的习惯, 代码的模块化会使整个代码逻辑更加清晰明了.

提取整块代码和提取变量相关代码分别占据 13.0% 和 11.7%, 这些长方法分解通常基于代码的语法结构, 直接提取具有逻辑完整性的代码块或变量相关代码段. 这两类方法分解有助于提高代码的可读性和可维护性, 减少代码长度并增强代码模块化程度. 这表明, 代码结构化在长方法分解中起着重要作用, 是开发人员在重构时优先考虑的因素之一.

复用代码的提取仅占 3.2%, 但其重要性不可忽视. 重复代码的存在往往会导致代码的冗余和难以维护, 通过长方法分解将复用代码提取, 程序员可以优化代码结构, 从而提高软件质量和维护效率.

其他类型占 5.2%, 这说明了长方法分解的多样性和复杂性. 在长方法的分解中, 可能会存在一些涉及特殊情况, 包括但不限于: 直接提取某一行代码作为新方法、从 Lambda 表达式中提取方法等, 这些情况可能需要根据具体情况具体分析, 找到最优的长方法分解方案, 这是当前自动化长方法分解下的一个巨大挑战.

通过对 Silva 等人[38]维护的公开数据集中 154 个可自动化长方法分解案例的分类和分析, 我们的研究结果表明, 开发人员在进行长方法分解时, 既关注代码的结构化和模块化, 通过变量、代码块、注释和分行等信息来找到长方法分解方案, 也常常伴随着其他修改和开发操作. 我们基于这些发现, 结合大语言模型, 设计开发自动化长方法分解工具.

3　基于大语言模型的长方法分解方法 Lsplitter

我们提出的方法 Lsplitter 如图 1 所示, 其中 L 既表示大语言模型 (LLM) 中的 L, 也表示长方法 (long method) 中的 L. Lsplitter 分为以下几步.

首先, 根据前期对于真实长方法分解重构数据的调研结果, 可以发现, 除了为了功能拓展等而进行的重构以外, 大部分的重构主要是以代码结构或语法结构为基础的. 基于这一发现, 我们设计了一些启发式规则, 并提出了

一系列可行的长方法分解方案.

其次, 结合调研结果, 我们可以发现, 提取整块代码和提取变量是根据代码的语义来进行分析的, 而大语言模型更加擅长阅读和理解代码, 因此它们有可能在给定的长方法中根据规则识别出更具有内聚性的代码片段, 这些代码片段可能会完成相对独立的功能, 适合分解到一个方法中. Lsplitter 会利用大语言模型来分解给定的长方法. 根据给定的方法, Lsplitter 会按照预定义的模板自动构建提示, 并通过 API 调用将生成的提示反馈给大语言模型. 大语言模型会将给定的长方法进行分解, 然后我们对大语言模型的结果基于位置信息进行合并和处理, 选择可行的方案.

最后, 针对基于启发式规则和基于大语言模型的一系列长方法分解结果, Lsplitter 会根据设计的度量指标, 对候选重构方案进行排序推荐.

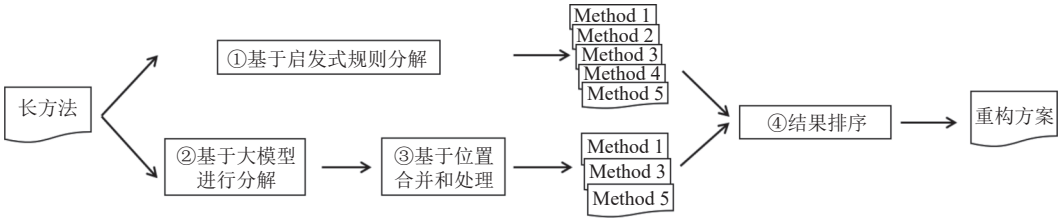

图 1 Lsplitter 方法总览

3.1 基于启发式规则的长方法分解

在长方法分解的调研中, 我们发现有 4 类情形是基于代码本体进行的, 其中提取整块代码和提取变量相关的代码是根据代码语义进行分析的, 基于注释和空行的提取是根据文本信息进行分析的, 重复代码是根据代码结构进行分析的. 基于这些发现, 我们针对这 4 类情形设计了相应的启发式规则, 以实现更加精确和有效的长方法分解.

针对整块代码的提取, Lsplitter 将枚举长方法中的常用语句类型 (for, while, if-else, try-catch) 中的所有代码块, 对于每一个代码块, 判断其是否满足以下条件, 若满足, 则将其加入候选项.

(1) 代码块中的代码不少于 3 行, 不超过方法总长度的 80%.

(2) 代码块中使用的变量在该代码块后使用的不超过 1 个.

(3) 代码块中最多只有一个出口 (return 或 throw) 且在代码块的末尾.

以上条件中, 第 1 个要求限定了被提取方法的长度, 以免选择了过长或者过短的方法, 后两个条件则是从可以提取新方法的角度考虑的, 由于 Java 代码中一个方法只能有一个返回值, 第 2 个条件则规避了这个风险, 避免代码块中对变量带来改动让在后续代码中继续使用.

针对变量相关的代码提取, Lsplitter 将枚举长方法中所有的变量, 并根据其的定义部分 (在表达式作为左值) 和使用部分 (在表达式作为右值) 划分成两部分, 对于每一个部分, 判断是否满足以下条件, 若满足, 则将其加入候选项.

(1) 变量的定义和使用部分之间代码没有交叉.

(2) 该部分的代码不少于 3 行, 不超过方法总长度的 80%.

(3) 对于定义部分, 该部分其他变量没有在此部分后的代码使用, 且不能有出口语句.

(4) 对于使用部分, 该部分使用的变量在该代码块后使用的不超过 1 个, 且最多在此部分的末尾只有一个出口.

以上 4 个条件中, 第 (1) 个条件是对于变量相关代码提取的基本要求, 定义部分和使用部分要是两个独立的部分才可以进行分解, 后 3 个条件则是对于可提取性的检测, 对于变量定义部分, 该变量本身将作为返回值返回给原方法, 因此不再接受其他在后面要使用其他变量, 而对于变量使用部分, 该变量是作为参数进行处理的, 则与普通代码块的要求一致.

针对注释和空行的提取则直接根据代码结构, 找到空行+注释+代码的部分结构, 按照判断代码块是否可以提取的 3 个条件进行判断, 如果可以, 则加入候选项.

针对重复代码的提取, 首先我们会使用深度优先搜索对抽象语法树进行一轮遍历, 标注每个节点下面的深度, 之后在同深度的节点之间进行比较, 若两个节点下有且仅有 SimpleName 属性值不一致且可以形成映射关系, 则视为他们是重复代码, 针对重复代码, 按照判断代码块是否可以提取的 3 个条件进行判断, 如果可以, 则加入候选项.

对于每一个候选项, 我们尝试将对应部分进行方法提取, 如果可以正常提取, 则将其作为候选项. 如果有多个候选项重复选择了某一个代码段, 则认为他们是相同的, 这将在后面的方案排序中多次计入推荐次数中.

3.2 基于大语言模型的长方法分解

大语言模型 (如 ChatGPT 等) 擅长阅读和理解代码, 它们更有可能识别出具有内聚性的代码片段, 这种内聚代码片段可能会完成相对独立的功能, 在方法分解时它们更有可能分配到同一个方法中. 因此, 我们可以利用这种内聚代码片段进行方法分解. 基于这一假设和针对长方法分解的调研, 我们使用大语言模型将给定的长方法分解为一组小的内聚代码片段. 而大语言模型的性能受提示的影响很大. 为此, 我们借鉴 Liu 等人[20]的工作进行了提示工程, 设计了一个由以下维度组成的提示模板.

(1) 角色提示: Suppose you are a skilled software engineer and now you should refactor your code.

(2) 任务提示: Please decompose the following long method into small ones *(based on the blocks/variables)*, but do not perform any other refactoring on the original method.

(3) 要求提示: Please give me the resulting methods with comments for each method in Javadoc format. The new methods should have unique method names. Please avoid tiny methods(less than 3 lines), empty methods, or new classes.

(4) 处理提示: Don’t generate code summary. Only give me the new method.

(5) 长方法本体: Here we present the to-be-decomposed long method in markdown format.

我们在提示中明确指定了一些约束条件来引导大语言模型进行重构. 首先, 在任务提示中, 我们会分为 3 种提示词, 分别让大语言模型直接进行长方法分解、基于代码块进行长方法分解、基于变量进行长方法分解. 后两者是根据前期调研发现的和语义相关的最常见的两种长方法分解类型, 我们将单独引导大语言模型针对这两类分解进行考虑. 其次, 我们告诉大语言模型 but do not perform any other refactoring on the original method, 因为大语言模型可能会执行除长方法分解之外的其他重构, 如方法与变量的重命名等, 这种意外的重构应该避免. 接着, 明确规定 Please avoid tiny methods (less than 3 lines), empty methods, or new classes, 以减少小方法或者空方法带来的影响; 最后, 要求 please give me the resulting methods with comment for each method in Javadoc format, 因为我们在后续的处理中, 将依赖大语言模型生成的注释来合并这些有内聚性的代码片段.

我们会对每一个长方法针对 3 种不同的类型的分解进行多次尝试, 分别处理以得到不同的候选项. 每一个执行过程中, 大语言模型都会将长方法 LM 分解成修改版本 LM' 和一系列辅助方法 m_1, m_2, …, m_n, 即

$$LM \rightarrow LM' + \{m_1, m_2, \ldots, m_n\}$$

3.3 基于位置和相似度对大语言模型分解结果合并

在大语言模型分解的结果中, 仍然会产生一些相似的方法, 如图 2 所示, verifyShardInfo、verifyCurrentNodeInfo 和 verifyUnassignedInfo 是相似的方法, 它们拥有相似的方法名和代码逻辑, 我们可以将它们合并在一起, 成为新的方法 verifyClusterAllocationExplanation.

为此, 我们在此提出了一种基于位置的合并算法, 用于合并方法体由原始方法中的连续块分解而来的相似的方法.

图 3 展示了合并这些方法的算法. 该算法以 methods 为输入, 合并相似且连续的方法. 在第 4 行, 我们计算方法之间的相似性矩阵 SimMatrix. 在第 5 行计算每个方法的聚类系数 C. 第 8–18 行, 按照聚类系数从大到小的顺序遍历每个方法, 尝试包含它的所有区间, 并评估它们是否可以合并. 如果某个区间可以合并, 我们会计算其合并得分, 最终选择得分最高的区间进行合并操作. 接下来, 我们将分别介绍合并的选择和生成.

```java
/**
 * Verifies the shard information in the cluster allocation
explanation.
 */
private void verifyShardInfo(ClusterAllocationExplanation
explanation) {
    ShardId shardId = explanation.getShard();
    boolean isPrimary = explanation.isPrimary();
    assertEquals("idx", shardId.getIndexName());
    assertEquals(0, shardId.getId());
    assertTrue(isPrimary);
}

/**
 * Verifies the current node information in the cluster allocation
explanation.
 */
private void verifyCurrentNodeInfo(ClusterAllocationExplanation
explanation) {
    ShardRoutingState shardRoutingState =
explanation.getShardState();
    DiscoveryNode currentNode = explanation.getCurrentNode();
    assertEquals(ShardRoutingState.STARTED, shardRoutingState);
    assertNotNull(currentNode);
}

/**
 * Verifies the unassigned information in the cluster allocation
explanation.
 */
private void verifyUnassignedInfo(ClusterAllocationExplanation
explanation) {
    UnassignedInfo unassignedInfo = explanation.getUnassignedInfo();
    assertNull(unassignedInfo);
}
```

合并前的方法

```java
/**
 * Verifies the shard, current node, and unassigned information in
the cluster allocation explanation.
 */
private void
verifyClusterAllocationExplanation(ClusterAllocationExplanation
explanation){
  ShardId shardId=explanation.getShard();
  boolean isPrimary=explanation.isPrimary();
  ShardRoutingState shardRoutingState=explanation.getShardState();
  DiscoveryNode currentNode=explanation.getCurrentNode();
  UnassignedInfo unassignedInfo=explanation.getUnassignedInfo();
  assertEquals("idx",shardId.getIndexName());
  assertEquals(0,shardId.getId());
  assertTrue(isPrimary);
  assertEquals(ShardRoutingState.STARTED,shardRoutingState);
  assertNotNull(currentNode);
  assertNull(unassignedInfo);
}
```

合并后的方法

图 2　相似方法合并例子

Algorithm 1: Location-Based Merging

Input: LM', *methods*

1 initialization
2 **while** *True* **do**
3 $n \leftarrow len(methods)$
4 $SimMatrix \leftarrow calcSim(methods)$
5 $C \leftarrow calcClusteringCoefficient(SimMatrix)$
6 $methods.sort(by\ clustering\ coefficient, descending)$
7 $curBest \leftarrow None$
8 **for** *m in methods* **do**
9 $id \leftarrow m.id$
10 **for** *length in* $[2, n]$ **do**
11 **for** $start \in [id - length + 1, id]$ **do**
12 $end \leftarrow start + length - 1$
13 **if** $canMerge(methods_{start,\cdots,end})$ *and* $methods_{start,\cdots,end}.score > curBest$ **then**
14 $curBest \leftarrow methods_{start,\cdots,end}$
15 **end**
16 **end**
17 **end**
18 **end**
19 **if** $curBest \neq None$ **then**
20 $merge(curBest)$
21 **else**
22 break
23 **end**
24 **end**

图 3　基于位置的合并算法

3.3.1　合并的选择

图 3 中, 算法的第 4 行计算 methods 的相似度矩阵. 两个方法 (m_1 和 m_2) 之间的相似度定义如下:

$$Similarity(m_1,m_2)=CS(m_1,m_2)+SS(m_1,m_2)+VS(m_1,m_2).$$

相似度包括 3 个部分: 内容相似度 (CS)、结构相似度 (SS) 和变量相似度 (VS). 其中, 内容相似度是根据大语言模型对每个方法分解的注释和方法名计算得出的. 结构相似度是指两个方法之间抽象语法树结构的相似性, 而变量相似度则表示其中使用的变量的重复率.

对于内容相似度, 我们使用基于 LCS (最长公共子序列) 的词级相似度, 分为两个维度: 由大语言模型生成的方法名和注释. 对于由大语言模型生成的方法名, 根据驼峰大小写规则进行分词. 对于注释, 使用 Javadoc 注释的描述. 我们对内容相似度的定义如下:

$$CS(m_1,m_2)=\max(Sim(name_1,name_2),Sim(comment_1,comment_2)),$$

$$Sim(A,B)=\frac{2\times LCS(A,B)}{len(A)+len(B)}.$$

对于结构相似度, 我们计算方法体之间抽象语法树的相似性. 提取方法中每条语句的抽象语法树结构, 不考虑 SimpleName 信息, 并计算两个方法之间相同抽象语法树结构的比例作为结构相似度.

对于变量相似性, 我们计算两个块之间共享变量的比例.

canMerge 函数用于判断是否可以合并一组方法. 如果方法满足以下条件, 我们将把它们合并为一个.

$$Mergeable(methods_{\{i,\dots,j\}})=SI(methods_{\{i,\dots,j\}})\&Score(methods_{\{i,\dots,j\}})\&Str(methods_{\{i,\dots,j\}})\&Len(methods_{\{i,\dots,j\}})$$

$$SI(methods_{\{i,\dots,j\}})=Invocations(methods_i)==1$$

$$Score(methods_{\{i,\dots,j\}})=methods_{\{i,\dots,j\}}.score>\beta$$

$$Str(methods_{\{i,\dots,j\}})=MatchStructuralRule(methods_{\{i,\dots,j\}})$$

$$Len(methods_{\{i,\dots,j\}})=TotalLength(methods_{\{i,\dots,j\}})\leqslant 80\%\cdot OriginalLength$$

(1) 这些方法仅被调用一次.

(2) 这些方法的相似度分数要达到一定的阈值.

(3) 这些方法遵守代码结构规则: 它们必须是连续的, 并且在语法上是可以合并的. 如, 不能合并 if 语句和 else 语句中的代码块.

(4) 这些方法的总长度不超过原方法的 80%.

一系列方法的相似度分数定义为其方法之间的平均相似度, 定义如下:

$$score(methods_{s,\dots,e})=\frac{\sum_{i\neq j}Similarity(methods_i,methods_j)}{(e-s+1)\times(e-s)/2}.$$

3.3.2 方法的合并

为了完成方法间的合并, 我们仍然使用大语言模型来完成. 为此, 设计了以下提示模板.

(1) 角色提示: Suppose you are a skilled software engineer and now you should refactor your code.

(2) 任务提示: Now here are some Java methods, please merge them into one method.

(3) 处理提示: Don’t generate code summary. Only give me the new method.

(4) 旧方法: Here we present the to-be-merged methods in markdown format.

通过这个模板, 大语言模型帮助我们执行了多个方法之间的合并操作. 不过, 还需要将多个函数调用语句合并为一个, 我们同样使用大语言模型来解决, 并设计了以下提示.

(1) 角色提示: Suppose you are a skilled software engineer and now you should refactor your code.

(2) 旧方法: You have merged these methods into one method. Here we present the old methods in markdown format.

(3) 新方法: You merge them into this method. Here we present the merged methods in markdown format.

(4) 任务提示: Now please give me the new method invocation for it.

(5) 旧函数调用: The original method invocations are:

(6) 处理提示: Don’t generate code summary. Only give me the new method invocation.

通过这个模板, 我们可以将多个函数调用语句合并为一个, 如图 2 的例子中, 原先的函数调用语句将被转化为 verifyClusterAllocationExplanation (explanation), 这和原始代码是等价的.

这样, 我们在大语言模型的重构结果进行优化, 作为候选项用于结果排序和推荐中.

3.4 结果排序

通过启发式规则和大语言模型, 我们得到了一系列长方法分解的候选项, 为了确定高质量建议的优先级, 我们考虑待提取片段中每一行作为候选项的次数的最大值和平均值, 作为排序的度量, 即:

$$\alpha \times \max(T(line_i)) + (1-\alpha) \times avg(T(line_i))$$

其中, $T(line_i)$ 表示第 i 行在所有候选项中出现的次数, α 为设定的一个系数. 我们认为, 针对原方法中的每一行, 如果其被推荐的次数越高, 则说明该方法越需要被提取, 因此在排序过程中考虑该候选项中每一行出现的最高次数和平均次数.

我们将其从大到小排序, 度量值越大的候选项将越靠前进行推荐.

4 实验分析

4.1 实验数据

我们在 Dorin 整理的拓展数据集上进行实验, 该数据集使用 RefactoringMiner 在 IntelliJ Community Edition 和 CoreNLP 两个开源项目上进行挖掘, 得到所有和长方法分解有关系的重构提交, 并进行筛选, 去除了提取单行代码和提取方法与主体重叠等不可进行自动化修复的情况, 最终共 2 849 个长方法分解重构实例. 这两个项目在业内都是有名的项目, 且开发团队较为成熟, 代码质量高, 同时, GitHub 社区中提交数量较多, 分别超过了 40 万和 1 万次, 代表了程序员的真实开发场景. 该数据集具体情况如表 2 所示.

表 2　实验数据集

数据集	项目	样本数量
Dorin’s Extend Corpus	IntelliJ Community Edition	1 176
	CoreNLP	1 673
	总数	2 849

4.2 评价指标及基准方法

在本文中, 我们将采用在 3% 容忍度下的长方法分解命中率作为评价指标. 也就是说如果推荐方案中存在某一个方案的代码片段和数据集中实际分解的代码片段在 3% 的误差内匹配, 则认为命中了此方案. 即, 对于长度为 len 的方法, 其数据集中实际提取了 $s1$~$e1$ 行, 存在推荐方案 $s2$~$e2$ 行, 满足

$$s1 - 3\% \times (e1 - s1 + 1) \leqslant s2 \leqslant s1 + 3\% \times (e1 - s1 + 1),$$

且

$$e1 - 3\% \times (e1 - s1 + 1) \leqslant e2 \leqslant e1 + 3\% \times (e1 - s1 + 1),$$

则认为该长方法分解命中. 我们分别比较每个工具推荐的前 5 个候选项的命中率 (记为 Recall-5) 和每个工具推荐的所有选项的命中率 (记为 Recall-All).

我们将所提方法 Lplitter 与 LiveRef[27]、Shahidi 的工作[13]和 EM-Assist[25]进行比较, 前两个方法代表了传统方案, 最后一个代表基于大语言模型的方案.

LiveRef 是基于传统算法在 IntelliJ IDEA 上实现的一个实时重构的插件, 该插件目前仅实现了长方法分解部分, 对于一个长方法, 其会选择出所有的超过 3 行且不超过 80% 原代码长度的可重构的连续代码段作为候选项. 在候选项中, 将根据循环复杂度、被提取代码行数、内聚度和认知复杂度进行排序进行推荐.

Shahidi 的工作基于方法依赖关系图进行模块化分析, 利用依赖图矩阵算出模块化矩阵, 并根据矩阵计算得到

聚类系数值, 并以此针对长方法进行提取和排序.

EM-Assist 是基于大语言模型实现的长方法分解重构工具, 其将一段代码直接交给大语言模型, 然后对大语言模型生成的结果进行静态分析, 剔除不可重构和无意义的方案, 并设计了一系列启发式规则对结果进行微调. 反复多次执行以得到多个候选结果, 通过计算每个方案的热度和流行度对候选项进行排序推荐.

4.3 实验过程

我们使用 Dorin 整理的 IntelliJ Community Edition 和 CoreNLP 两个项目作为实验数据集, 并执行长方法分解工作. 所有实验均在配备英特尔酷睿 i7 处理器和 16 GB 内存的 Windows 10 机器上进行. 为了展示大语言模型的强大性能, 我们在实验中大语言模型部分使用了 GPT-4 模型. 对于每一个长方法, 我们分别采用启发式方法和大语言模型分别进行重构, 对于基于大语言模型的重构, 每种类型最多执行 3 次并进行处理, 获得候选项, 根据排序规则进行排序.

在我们的实验中, 所有的代码都使用 Java 语言编写而成, 针对代码的分析, 采用 Eclipse 的 JDT 进行实现, 对于大语言模型的调用, 采用 plexpt 的 ChatGPT 库来实现. 对于参数的选择, 在针对基于位置信息对大语言模型的结果合并时, 选择β=2 作为阈值; 在结果排序中, 选取α=0.2; 在调用大语言模型的过程中, 根据相关工作[25]的经验表明, temperature 越高, 大语言模型在长方法分解工作中效果越高, 因此我们选用 temperature=1.0 调用 GPT-4 模型.

4.4 实验结果与分析

为了评估 LSplitter 的有效性, 我们研究了以下 4 个问题.

• RQ1: Lsplitter 是否可以获得比现有的长方法分解工具更好的结果?

• RQ2: Lsplitter 中的不同部分对结果的影响如何?

• RQ3: 程序员如何评价 Lsplitter 分解的结果?

• RQ4: 待分解的方法复杂度与 Lsplitter 效果之间有没有关系?

RQ1 研究了 Lsplitter 的性能, 这项研究将比较 Lsplitter 和基准方法之间在 3% 容忍度下的命中率差异; RQ2 研究了 Lsplitter 中启发式策略和基于大语言模型的重构方案在提高长方法分解性能的有效性, 本问题旨在评估每一部分在重构方案中带来的影响; RQ3 则是评估开发人员对于 Lsplitter 分解结果的看法, 这项研究将从专业的程序员的角度评价 Lsplitter 分解的结果质量, 我们从 GitHub 上的主流项目中抽取了 60 个方法进行重构, 并聘请了 3 位经验丰富的程序员来评估这些结果; RQ4 则利用程序员的评价探讨了待重构方法的复杂性与 Lsplitter 性能效率之间的关系, 这个问题旨在了解原始方法中不同程度的内在复杂性会如何影响 Lsplitter 执行长方法分解任务的能力.

RQ1: Lsplitter 是否可以获得比现有的长方法分解工具更好的结果?

为了验证这个问题的结果, 我们将 Lsplitter 与基于传统方法实现的 LiveRef、基于传统方法和模块化矩阵结合的 Shahidi 的工作和基于大语言模型实现的 EM-Assist 方法进行了对比实验, 并计算了前 5 个推荐方案和所有推荐方案在 3% 的容忍度下的命中率, 实验的结果见表 3.

表 3 Lsplitter 与 LiveRef 和 EM-Assist 方法性能比较

方法	Recall-5 (%)	Recall-All (%)
Lsplitter	45.3	47.8
LiveRef	6.5	8.0
Shahidi’s	18.7	24.4
EM-Assist	42.1	44.6

与基于传统方法实现的 LiveRef 相比, Lsplitter 的性能有了显著提升. Lsplitter 的 Recall-5 和 Recall-All 分别为传统方法的 6.97 倍和 5.98 倍. 这表明, 传统方法在处理长方法分解任务时存在较大的局限性, 而 Lsplitter 有效地克服了这些局限, 取得了更好的效果.

与基于传统方法和模块化矩阵结合实现的 Shahidi 的工作相比, Lsplitter 的性能也有所提升, Lsplitter 的 Recall-5 和 Recall-All 分别为其的 2.42 倍和 1.83 倍. 这表面, 即使使用了模块化矩阵等创新性工作, 长方法分解任务也有一定的困难.

与仅基于大语言模型实现的 EM-Assist 相比, Lsplitter 的性能也相对更好, Recall-5 和 Recall-All 分别较 EM-Assist 提高了 7.6% 和 7.2%. 这说明 Lsplitter 不仅在传统方法面前表现优异, 在面对先进的大语言模型方法时也能保持竞争力, 证明了其在长方法分解任务中的有效性和可靠性.

综上, Lsplitter 在长方法分解上表现出色, 这说明前期的调研为工具的设计和实现带来了正确的思路, 同时也展现了 Lsplitter 在实际应用中的潜力.

RQ2: Lsplitter 中的不同部分对结果的影响如何?

为了回答该问题, 我们将分别仅使用启发式规则和大语言模型在数据集上对长方法进行分解, 并计算了前 5 个推荐方案和所有推荐方案在 3% 的容忍度下的命中率, 实验的结果见表 4.

表 4　Lsplitter 消融实验显示启发式规则和大语言模型的贡献 (%)

方法	Recall-5	Recall-All
Lsplitter	45.3	47.8
仅使用启发式规则	28.8	30.3
仅使用大语言模型	35.5	38.5

从结果可以看出, 将启发式策略和大语言模型结合会显著高于仅使用单种策略的情况. 这表明, Lsplitter 中各部分的结合能够有效提升长方法分解的性能. 较仅使用启发式规则的结果相比, Lsplitter 的 Recall-5 和 Recall-All 分别提升了 57.3% 和 57.8%. 虽然启发式规则在分解方法中发挥了一定作用, 但其效果相对有限. 启发式规则通过简单的逻辑和固定模式识别长方法中的可分解部分, 能一定程度上提升代码的模块化和结构化, 但面对复杂的代码场景时, 其能力受限. 而仅使用大语言模型的性能相较于启发式规则有明显的提升. 这表明, 大语言模型在长方法分解中能够捕捉到更多的细节和复杂模式, 对于代码的语义能着较深的理解. 然而, 大语言模型的性能虽然优于启发式规则, 但与 Lsplitter 的整体效果相比仍有差距.

综上, 实验结果表明, Lsplitter 结合了启发式规则和大语言模型的优势, 既利用了启发式规则的简单高效, 又借助了大语言模型的强大识别能力, 显著提升分解效果. 这一发现进一步验证了 Lsplitter 设计的有效性.

RQ3: 程序员如何评价 Lsplitter 分解的结果?

软件重构是为了提高软件的可读性、结构性, 是为了程序员服务. 因此, 研究程序员眼中对分解方案的评价是非常重要的. 为了验证这个问题, 我们从 GitHub 上选择了 5 个拥有 star 最多的开源项目, 见表 5, 并使用 PMD 中的 NcssCount 规则并使用默认阈值来检测长方法. NCSS (non-commenting source statements) 是一个静态指标, 它对注释不敏感, 与代码风格无关. NCSS 值越大, 说明该方法的有效代码长度也越长, 也越可能有长方法代码坏味. 对于每个项目, 我们从检测到的长方法中选择 12 个方法. 其中一个是 NCSS 值最小的方法, 一个是 NCSS 值最大的方法, 其余方法按其 NCSS 值分为 10 组, 我们在每组中选择处于中位数的方法. 最终, 数据集共选择了 60 个方法.

表 5　RQ3 所选实验项目

项目	类的数量	方法数量	代码行数
Elasticsearch	190318	24692	2620814
Spring-framework	81407	14736	753634
Guava	58713	6866	515764
RxJava	33980	3003	314914
Ghidra	172589	19216	1816009

我们在这 60 个方法上运行 Lsplitter 进行长方法分解, 并邀请了 3 名经验丰富的程序员来对分解结果打分, 每位程序员都拥有至少 3 年的 Java 编程经验. 我们要求每个人对每个重构方案打 1–5 分, 分数越高, 表示重构建议

的质量越高, 3 分表示与程序员自身的预期相符. 实验的结果见表 6.

从结果可以看出, 程序员对 Lsplitter 分解结果的平均评分为 3.72 分, 表明 Lsplitter 的分解方案总体上得到了程序员的认可, 大多数的分解方案质量较高, 略高于程序员自身预期. 尽管 3 名程序员评分存在一定的差异, 但总体表明 Lsplitter 提供的分解方案是有价值的.

表 6 程序员打分结果

程序员	分数
1	3.62
2	3.21
3	4.34
平均	3.72

综上, 程序员的评分表明 Lsplitter 能够稳定地提供高质量的分解建议, 这说明 Lsplitter 在处理不同类型和规模的项目时, 都能保持良好的性能, 表现出其在长方法分解中的潜力和有效性.

RQ4: 待分解的方法复杂度与 Lsplitter 效果之间有没有关系?

为了研究 Lsplitter 的性能与方法复杂度之间的关系, 我们在 RQ3 的基础上, 重点研究了每个方法的 NCSS 与程序员打分之间的关系. 我们根据 NCSS 将数据分为 4 组, 计算了每组的中位数、极值、四分位数和其他统计量, 并用箱型图表示, 如图 4 所示, 图的横轴表示方法的 NCSS 范围, 纵轴表示方法的平均得分.

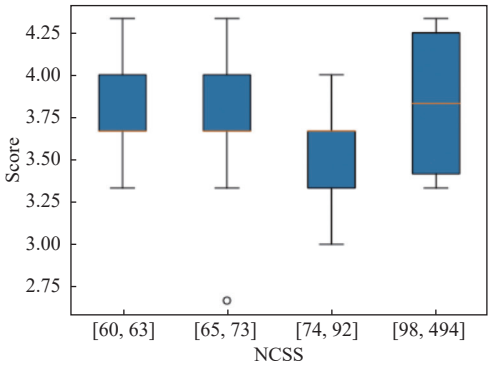

图 4 待分解方法 NCSS 和得分关系图

从结果可以看出, 各组的中位数得分在不同的 NCSS 范围内变化不大, 表明得分分布的稳定性相对较高. 前两个箱的大小几乎相同, 而较大的 NCSS 范围的分数范围更广, 这可能表明在处理更复杂的方法时, Lsplitter 的性能可能略有不稳定. 而只有第 2 个 NCSS 范围 [65, 73] 出现了一个离群值, 这表明总体上是稳定的, 只是在某些情况下偶尔会出现离群值.

与此同时, 我们还进行了 P 检验, 以研究 NCSS 与得分之间的关系. NCSS 与得分之间的皮尔逊相关系数为 0.017, 表明两者之间的线性关系很弱. 与这一相关关系相关的 P 值约为 0.9, 远高于通常的显著性水平 0.05. 这表明, 相关性在统计上并不显著. 因此, 可以得出结论: Lsplitter 的性能与方法的复杂性无关, 而且在不同的复杂性下保持稳定.

综上, Lsplitter 的性能在不同复杂程度的方法中都表现出很高的稳定性. 即使在处理复杂方法时, 它的性能也只会出现轻微的波动. P 值远大于 0.05 的统计检验表明, Lsplitter 的性能与方法复杂度并无显著相关.

5 总 结

长方法是众所周知的代码坏味, 应通过软件重构来解决. 然而, 分解长方法具有挑战性, 因此自动分解长方法是非常可取的. 为此, 我们在本文中提出了一种分解长方法的新方法 Lsplitter. 该方法根据调研前期对于实际开发过程中长方法的分解情况的结果, 利用启发式规则和大语言模型来分解长方法, 从而创造出有效的重构结果.

虽然所提出的方法并不针对特定语言, 但原型实现只接受 Java 程序. 代码分析部分 (从源代码中分析抽象语法树) 是针对特定语言的, 如果将来需要考虑其他编程语言, 则应重新实现该部分. 考虑到长方法和上帝类之间的相似性, 利用本文介绍的关键内部结构分解上帝类在未来可能会取得一定成果.

References:

[1] Fowler M, Beck K. Refactoring: Improving the Design of Existing Code. 2nd ed., Boston: Addison-Wesley Professional, 2018.

[2] Chatzigeorgiou A, Manakos A. Investigating the evolution of code smells in object-oriented systems. Innovations in Systems and Software Engineering, 2014, 10(1): 3–18. [doi: 10.1007/s11334-013-0205-z]

[3] Danphitsanuphan P, Suwantada T. Code smell detecting tool and code smell-structure bug relationship. In: Proc. of the 2012 Spring Congress on Engineering and Technology. Xi'an: IEEE, 2012. 1–5. [doi: 10.1109/SCET.2012.6342082]

[4] Hamza H, Counsell S, Hall T, Loizou G. Code smell eradication and associated refactoring. In: Proc. of the 2nd Conf. on European Computing Conf. Malta: WSEAS, 2008. 102–107.

[5] Opdyke WF. Refactoring object-oriented frameworks [Ph.D. Thesis]. Champaign: University of Illinois at Urbana-Champaign, 1992.

[6] Fowler M. Refactoring: Improving the Design of Existing Code. 2nd ed., Berkeley: Addison-Wesley Professional, 2018.

[7] McConnell S. Code Complete: A Practical Handbook of Software Construction. 2nd ed., Redmond: Microsoft Press, 2004.

[8] Haefliger S, von Krogh G, Spaeth S. Code reuse in open source software. Management Science, 2008, 54(1): 180–193. [doi: 10.1287/mnsc.1070.0748]

[9] Murphy-Hill E, Parnin C, Black AP. How we refactor, and how we know it. IEEE Trans. on Software Engineering, 2012, 38(1): 5–18. [doi: 10.1109/TSE.2011.41]

[10] Bazhenov IO, Lubkin IA. Methodology of software code decomposition analysis. In: Proc. of the 2018 Dynamics of Systems, Mechanisms and Machines. Omsk: IEEE, 2018. 1–5. [doi: 10.1109/Dynamics.2018.8601441]

[11] Maruyama K. Automated method-extraction refactoring by using block-based slicing. In: Proc. of the 2001 Symp. on Software Reusability: Putting Software Reuse in Context. Toronto: ACM, 2001. 31–40. [doi: 10.1145/375212.375233]

[12] Xu S, Sivaraman A, Khoo SC, Xu J. GEMS: An extract method refactoring recommender. In: Proc. of the 28th Int'l Symp. on Software Reliability Engineering. Toulouse: IEEE, 2017. 24–34. [doi: 10.1109/ISSRE.2017.35]

[13] Shahidi M, Ashtiani M, Zakeri-Nasrabadi M. An automated extract method refactoring approach to correct the long method code smell. Journal of Systems and Software, 2022, 187: 111221. [doi: 10.1016/j.jss.2022.111221]

[14] Yamanaka J, Hayase Y, Amagasa T. Recommending extract method refactoring based on confidence of predicted method name. arXiv: 2108.11011, 2021.

[15] Cui D, Wang QQ, Wang SQ, Chi JL, Li JN, Wang L, Li QS. REMS: Recommending extract method refactoring opportunities via multi-view representation of code property graph. In: Proc. of the 31st Int'l Conf. on Program Comprehension. Melbourne: IEEE, 2023. 191–202. [doi: 10.1109/ICPC58990.2023.00034]

[16] Xu HR, Kim YJ, Sharaf A, Awadalla HH. A paradigm shift in machine translation: Boosting translation performance of large language models. In: Proc. of the 12th Int'l Conf. on Learning Representations. OpenReview.net, 2024.

[17] Yao BW, Jiang M, Bobinac T, Yang DY, Hu JJ. Benchmarking machine translation with cultural awareness. In: Proc. of the Findings of the Association for Computational Linguistics. Miami: ACL, 2024. 13078–13096. [doi: 10.18653/v1/2024.findings-emnlp.765]

[18] Shi YC, Ma HH, Zhong WL, Tan QY, Mai GC, Li X, Liu TM, Huang JZ. ChatGraph: Interpretable text classification by converting ChatGPT knowledge to graphs. In: Proc. of the 2023 IEEE Int'l Conf. on Data Mining Workshops. Shanghai: IEEE, 2023. 515–520. [doi: 10.1109/ICDMW60847.2023.00073]

[19] Kurisinkel LJ, Chen NF. LLM based multi-document summarization exploiting main-event biased monotone submodular content extraction. arXiv: 2310.03414, 2023.

[20] Liu C, Bao XL, Zhang HY, Zhang N, Hu HB, Zhang XH, Yan M. Improving ChatGPT prompt for code generation. arXiv: 2305.08360, 2023.

[21] Ahmed T, Devanbu P. Few-shot training LLMs for project-specific code-summarization. In: Proc. of the 37th IEEE/ACM Int'l Conf. on Automated Software Engineering. Rochester: ACM, 2022. 177. [doi: 10.1145/3551349.3559555]

[22] Sobania D, Briesch M, Hanna C, Petke J. An analysis of the automatic bug fixing performance of ChatGPT. In: Proc. of the 2023 IEEE/ACM Int'l Workshop on Automated Program Repair. Melbourne: IEEE, 2023. 23–30. [doi: 10.1109/APR59189.2023.00012]

[23] Xia CS, Zhang LM. Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT. In: Proc. of the 33rd ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Vienna: ACM, 2024. 819–831. [doi: 10.1145/3650212.3680323]

[24] Yuan ZQ, Liu JW, Zi QC, Liu MW, Peng X, Lou YL. Evaluating instruction-tuned large language models on code comprehension and

generation. arXiv:2308.01240, 2023.
[25] Pomian D, Bellur A, Dilhara M, Kurbatova Z, Bogomolov E, Bryksin T, Dig D. Together we go further: LLMs and IDE static analysis for extract method refactoring. arXiv:2401.15298, 2024.
[26] Chen XP, Hu X, Huang Y, *et al*. Deep learning-based software engineering: Progress, challenges, and opportunities. Science China Information Sciences, 2025, 68(1): 111102. [doi: 10.1007/s11432-023-4127-5]
[27] Fernandes S, Aguiar A, Restivo A. LiveRef: A tool for live refactoring Java code. In: Proc. of the 37th IEEE/ACM Int'l Conf. on Automated Software Engineering. Rochester: ACM, 2022. 161. [doi: 10.1145/3551349.3559532]
[28] Tsantalis N, Chatzigeorgiou A. Identification of extract method refactoring opportunities for the decomposition of methods. Journal of Systems and Software, 2011, 84(10): 1757–1782. [doi: 10.1016/j.jss.2011.05.016]
[29] Sharma T. Identifying extract-method refactoring candidates automatically. In: Proc. of the 5th Workshop on Refactoring Tools. Rapperswil: ACM, 2012. 50–53. [doi: 10.1145/2328876.2328883]
[30] Silva D, Terra R, Valente MT. JExtract: An eclipse plug-in for recommending automated extract method refactorings. arXiv:1506.06086, 2015.
[31] Marinescu R. Detection strategies: Metrics-based rules for detecting design flaws. In: Proc. of the 20th IEEE Int'l Conf. on Software Maintenance. Chicago: IEEE, 2004. 350–359. [doi: 10.1109/ICSM.2004.1357820]
[32] Lanza M, Marinescu R. Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-oriented Systems. Berlin, Heidelberg: Springer, 2006. [doi: 10.1007/3-540-39538-5]
[33] Moha N, Guéhéneuc YG, Duchien L, Le Meur AF. DECOR: A method for the specification and detection of code and design smells. IEEE Trans. on Software Engineering, 2010, 36(1): 20–36. [doi: 10.1109/TSE.2009.50]
[34] Yoshida N, Kinoshita M, Iida H. A cohesion metric approach to dividing source code into functional segments to improve maintainability. In: Proc. of the 16th European Conf. on Software Maintenance and Reengineering. Szeged: IEEE, 2012. 365–370. [doi: 10.1109/CSMR.2012.45]
[35] Charalampidou S, Ampatzoglou A, Avgeriou P. Size and cohesion metrics as indicators of the long method bad smell: An empirical study. In: Proc. of the 11th Int'l Conf. on Predictive Models and Data Analytics in Software Engineering. Beijing: ACM, 2015. 8. [doi: 10.1145/2810146.2810155]
[36] Charalampidou S, Arvanitou EM, Ampatzoglou A, Avgeriou P, Chatzigeorgiou A, Stamelos I. Structural quality metrics as indicators of the long method bad smell: An empirical study. In: Proc. of the 44th Euromicro Conf. on Software Engineering and Advanced Applications. Prague: IEEE, 2018. 234–238. [doi: 10.1109/SEAA.2018.00046]
[37] Liu H, Jin JH, Xu ZF, Zou YZ, Bu YF, Zhang L. Deep learning based code smell detection. IEEE Trans. on Software Engineering, 2021, 47(9): 1811–1837. [doi: 10.1109/TSE.2019.2936376]
[38] Silva D, Tsantalis N, Valente MT. Why we refactor? confessions of GitHub contributors. In: Proc. of the 24th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. Seattle: ACM, 2016. 858–870. [doi: 10.1145/2950290.2950305]
[39] Cossette BE, Walker RJ. Seeking the ground truth: A retroactive study on the evolution and migration of software libraries. In: Proc. of the 20th Int'l Symp. on the Foundations of Software Engineering. Cary: ACM, 2012. 55. [doi: 10.1145/2393596.2393661]

徐子懋(2000－), 男, 博士生, CCF 学生会员, 主要研究领域为软件重构.

张宇霞(1992－), 女, 博士, 副研究员, CCF 专业会员, 主要研究领域为软件仓库挖掘, 开源软件生态系统.

姜艳杰(1993－), 女, 博士, CCF 专业会员, 主要研究领域为软件代码的自然语言处理和软件缺陷.

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