南京大學

# 研 究 生 毕 业 论 文

## （申请硕士学位）

论 文 题 目 跨项目测试代码自动复用技术

作 者 姓 名 朱晨乾

专 业 方 向 软件工程

研 究 方 向 软件工程

指 导 教 师 陈振宇教授

**2022 年 05 月 20 日**

学　　号 ：**MG1932019**

论文答辩日期 ： **2022** 年 **05** 月 **20** 日

指 导 教 师 ：　　（签字）

# Crossed-Project Test Code Automatic Reuse Technology

By

**Chenqian Zhu**

Supervised by

**Professor Zhenyu Chen**

A Thesis
Submitted to the Software Institute
and the Graduate School
of Nanjing University
in Partial Fulfillment of the Requirements
for the Degree of
**Master**

Software Institute

May 2022

# 学 位 论 文 原 创 性 声 明

任何收存和保管本论文的单位和个人，未经作者本人授权，不得将本论文转借他人并复印、抄录、拍照或以任何方式传播，否则，引起有碍作者著作权益的问题，将可能承担法律责任。

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不句含其他个人或集体已经发表或撰写的作品成果。本文所引用的重要文献，均已在文中以明确方式标明。本声明的法律结果由本人承担。

论文作者签名：_____

日期：　　年　　月　　日

# 南京大学研究生毕业论文中文摘要首页用纸

毕业论文题目：跨项目测试代码自动复用技术

软件工程 专业 2019 级硕士生姓名：朱晨乾

指导教师（姓名、职称）：陈振宇教授

## 摘 要

软件测试是软件开发流程中一个极其重要的组成部分，越来越多的新技术被应用于测试开发。测试代码自动生成与测试代码复用都是很有前途的提高单元测试效率的技术，但测试代码自动生成目前在实践中不尽如人意。测试代码推荐作为一种可行的实现测试代码复用的技术，越来越受到研究人员的关注。但该技术进行跨项目测试代码推荐时，推荐的测试代码需要经过一定程度的人工修改才能运行，而这则成为了提高测试效率的瓶颈。现有关于测试代码修复的工作主要源于软件演化后测试用例失效的情况。因此，对推荐的跨项目测试代码进行自动修复，有助于实现测试代码自动复用，从而使该技术能真正有效地应用于软件测试实践。

本文提出了一种跨项目测试代码自动复用技术，它收集开源在线编程平台编程题代码及其测试代码，通过将待测方法及其测试代码进行解析，提取能够表示它们功能的关键特征，如类名、方法名、方法参数类型列表、返回类型、具体代码等，把它们的作为具有相同或相似功能待测方法的评价指标。基于这些特征构建了一个关于待测方法及其测试代码的语料库。面对需要测试代码的语料库外部的待测方法，该技术使用字符串匹配、拼写校正和相似度度量等方法，对外部待测方法与语料库中待测方法进行功能相似性匹配，对具有相同或相似功能的语料库中待测方法对应的测试代码进行推荐并修复。然后，修复的测试代码将被直接用于进行黑盒测试，减少人工修改成本，相比不进行代码修改的测试代码推荐能更好地提高测试效率。由于实现了代码语料库自动构建工具，以很方便实现语料库规模增长，提高语料库可扩展性与测试代码推荐准确性。

实验基于最新发表的跨项目测试代码推荐工具 HomoTR 所用的原始 Java 代码数据集，首先对原始数据集中测试代码进行标准化并以此为基础构建代码语料库，然后对本文提出的跨项目测试代码自动复用技术进行评估。实验证明了该技术的有效性，实现了 46% 的测试代码推荐成功率，相比 HomoTR 提高了 18%，而推荐成功的测试代码中有 87% 实现了有效修复，跨项目测试代码自动复用的真实成功率为 40%。

**关键词**：软件测试，测试代码复用，测试代码推荐，测试代码修复

# 南京大学研究生毕业论文英文摘要首页用纸

THESIS: Crossed-Project Test Code Automatic Reuse Technology

SPECIALIZATION: Software Engineering

POSTGRADUATE: Chenqian Zhu

MENTOR: Professor **Zhenyu Chen**

## Abstract

Software testing is an extremely important part of software development process. A growing number of new technologies are applied to test development. Automatic test code generation and test code reuse are promising technologies to improve the efficiency of unit testing, but automatic test code generation performs not satisfactory in practice currently. Test code recommendation, as another feasible technology to realize test code reuse, has attracted more and more attention of researchers. However, when this technology is used in recommending cross project test code, the recommended test code needs to be manually modified before running, which has become the bottleneck to improve test efficiency. The existing work on test code repair mainly comes from the failure of test cases execution after software evolution. Therefore, repairing the recommended cross project test code is helpful to realize the automatic reuse of test code, so that the technology can be effectively applied to software testing practice.

This paper proposes a cross project test code automatic reuse technology, which collects the code and its test code of programming problems from open source online programming platforms, analyzes the method under test (MUT) and its test code, and extracts the key features that can represent their functionality, such as class name, method name, the list of types of the method parameters, return type, specific code, etc, They are regarded as the evaluation metrics of the MUT with the same or similar functionality. Based on these features, a corpus about the MUT and its test code is constructed. When the MUT outside the corpus needs test code, the technology uses string matching, spelling correction and similarity measurement to match the MUT in the corpus which shares the same or similar functionality with the externel MUT, and then recommends its test code and automatically repairs it. Then, the repaired test code

can be directly used to take black-box testing, and reduce the cost of manual modification. Compared with the test code recommendation without code modification, it can better improve the test efficiency. Due to the implementation of the code corpus automatic construction tool, it can easily realize the growth of corpus scale and improve the scalability of corpus and the accuracy of test code recommendation.

The experiment is based on the original Java code data set used by the newly published cross project test code recommendation tool, HomoTR. Firstly, the test code in the original data set is standardized and the code corpus is constructed by the data set, and then the cross project test code automatic reuse technology proposed in this paper is evaluated. Experiments have proved the effectiveness of this technology, achieving a 46% success rate of test code recommendation, which is 18% higher than that of HomoTR, while 87% of the successfully recommended test codes have been effectively repaired, and the real success rate of the cross project test code automatic reuse is 40%.

**Keywords:** Software Testing, Test Code Reuse, Test Code Recommendation, Test Code Repair

# 目录

# 表 目 录

# 图 目 录

# 第一章　绪论

## 1.1　研究背景及意义

随着互联网与信息技术对传统行业赋能程度的加深，以及移动互联网设备在人群间的全面普及，软件已成为人们日常生活中不可或缺的一部分。而应用场景的多样化使软件复杂程度日趋提升，也使其越来越容易出错，而致命的软件错误可能会带来严重的经济损失。因此，作为软件开发过程中保证软件正确性和可用性的最常用技术，软件测试的重要性就愈发凸显。单元测试作为软件测试中最基本、最重要的方法之一，它的编写与执行由开发人员完成，以确保各生产代码单元按预期工作 [1]，并对后续的集成测试、系统测试、验收测试等环节以及最终的软件质量产生重大影响 [2]。

然而，编写单元测试用例不仅是一项耗时的任务，还需要具有大量测试知识的开发人员 [3–5]。此外，为了追求敏捷开发，开发人员的大部分精力都集中在生产代码开发上，对编写测试用例持不屑一顾和消极的看法 [6, 7]。一些先进的测试技术，如测试代码自动生成技术、测试用例复用技术就应运而生，以协助开发人员提高测试效率。尽管测试代码自动生成很有前途，但在实际的大规模应用中仍有许多局限性 [8]。

鉴于软件复用是提高开发人员进行软件开发效率和质量的一种重要技术 [9–11]，能够缩减软件开发和维护的花费 [12]，具有很高的经济价值，与之相似的测试代码复用也因此进入视野。测试代码复用通过利用过去软件测试过程中积累的经验，将已使用的测试用例进行标准化和存档，未来则可在检索后进行借鉴或者复用，减少从头设计测试用例的冗余时间，以提高测试效率和可靠性 [9]。而测试代码推荐作为实现测试代码复用的一种方式，也越来越受到研究人员重视。

测试代码推荐技术 [7, 13–16] 广泛采用了具有相似或相同功能的待测方法可以复用彼此测试用例的基本思想，两个待测方法 $m_1$、$m_2$ 实现了相似或相同的功能，因此可以将 $m_1$ 的测试用例 $t_1$ 推荐给新的待测方法 $m_2$，通过参考 $t_1$ 的实现，编写用于测试 $m_2$ 的新测试用例 $t_2$。一般认为，具有相似功能的两个待测方法，它们的代码也具有相似性，因此可通过度量两个待测方法体中具体代码的相似性来衡量两个待测方法的功能相似性。

然而，现有的测试代码推荐技术 [7, 13–16] 存在一些缺点。例如，只有开发人员参与的项目中的现有测试用例可以被推荐 [7]，或者由于度量的限制，如仅

使用方法签名作为度量标准 [14, 15] 而导致推荐精度下降。如果我们可以使用更多项目中的更多测试用例构建语料库，或者使用更有效的度量标准，那么测试推荐的准确性将会提高。本人之前的工作 HomoTR[16] 在后续实际使用情况中反映，仍然存在推荐的测试代码质量不佳、推荐后的测试代码仍然需要一定程度的人工修改成本等问题，而后者本身也成为提高测试效率的瓶颈。

因此，除了将 HomoTR 原始 Java 代码数据集中的测试代码进行标准化并构建新的语料库，对推荐的测试代码进行修复工作就提上日程。不过，尽管测试代码自动修复技术已经被广泛应用于 GUI 测试、Web 测试、白盒测试等场景，但它们主要面向同项目软件演化后因变更导致的测试用例在回归测试时失效的问题 [17]，而与我们先前的工作 HomoTR[16] 主要针对的跨项目测试代码推荐这一场景有所不同。我们要做的是针对跨项目推荐的测试代码进行修复，而非对同项目回归测试用例代码进行修复。

## 1.2 国内外研究现状

复用思想在软件测试领域中的应用源自其在软件开发过程中的应用。软件复用指通过利用已有软件中的有效成分用于新软件的开发，提高了新软件开发的质量和效率 [18]。而软件测试往往占据了整个开发工作一半以上的时间，因此通过测试用例复用提高测试效率也得到研究人员重视。例如，单元测试用例可以在集成测试时进行复用，旧版本软件的测试用例可以在新版本软件进行回归测试时得到复用，以及同类软件在测试用例设计、测试策略、测试数据、测试步骤相似时，将一个软件的测试用例复用给另一个软件。芮素娟等人总结了可复用测试用例设计的指导原则 [19]，包括：

(1) 测试用例间相关性降至最低；

(2) 对被测软件的依赖尽量减弱；

(3) 描述要规范化；

(4) 尽量不包含常量，输入值用变量代替；

(5) 内容完整，结构统一；

(6) 分类合理。

而王珊珊等人也给出了可复用测试用例的四项必要质量特性 [9]，包括通用性（不个性化，与场景不关联）、独立性（不依赖其他测试用例）、规范化和可维

护性。产业界已经有大量关于测试用例复用的研究和实践，主要集中在信息安全 [20, 21]、空间科技 [22–24] 和军事科技领域软件 [25–30]。有些测试用例复用通过与测试需求进行关联，构建测试用例经验库用于复用。

Zhang 等人提出了关于测试用例复用维度的概念 [31]，测试用例复用可分为四大维度，分别是：

(1) 维护测试用例复用。即演化后的软件复用演化前软件的测试用例进行测试。

(2) 回归测试用例复用。即在回归测试阶段复用回归测试用例，以在对系统进行更改（如功能修改、补丁）后发现新错误 [32]。

(3) 垂直测试用例复用。即同项目测试用例复用。可复用的测试用例可以在软件产品线中复用，这些软件产品线共享一组通用的、可管理的特性，以满足特定市场细分或任务的特定需求 [33]。垂直复用是提高软件复用水平的主要途径之一，最适合大型长寿命项目的组织。

(4) 水平测试用例复用。即跨项目测试用例复用。复用的测试用例来自不同项目而非同项目，与“垂直测试用例复用”相对。

它们通过对自己开发的测试用例管理框架进行实验，证明测试用例复用可以提高工作效率。由此可见测试用例复用的很多优点，比如缩短测试周期，提高测试效率和可靠性，降低成本，解决测试人员经验不足的问题。但实现测试用例复用也有很多难点，比如测试用例分类标准、可复用度度量标准、测试用例库的科学管理等，都有待研究和讨论。

同时，大部分测试用例复用均是基于用自然语言书写的测试用例文档或测试需求进行复用，现有研究中基于测试代码进行复用的案例较少。

### 1.2.1 代码相似性度量

测试代码复用要求未经测试的待测方法和已经经过测试、拥有测试用例的待测方法具有相同或相似的功能，即两个待测方法的代码具有相似性，因此实现测试代码复用需要对待测方法代码进行相似性度量。而在软件系统中，相似或相同的代码片段称为代码克隆 [34]，因此也可以使用代码克隆的方法进行代码相似性度量。

现在已经有大量的代码相似性度量方法被提出 [35–47]。这些方法主要从代码文本和代码结构两方面度量代码相似性。从代码文本方面进行的代码相似性度量主要依据字符串 [37, 38] 或令牌 [39–41]。从代码结构方面进行的代码相似性度量主要基于树结构 [42–44] 和图结构 [45–47]。所有这些方法优缺点并存。就

它们的代码相似性度量准确性而言，越新的成果准确性越高，但是实现的复杂度也越高。

### 1.2.2 测试代码推荐

测试代码推荐是生产代码推荐与测试代码复用相结合的产物。学术界通常使用信息检索技术进行生产代码推荐，用自然语言处理的方式将代码片段进行分析和索引。但显然程序设计语言与自然语言的语法存在显著区别，这种基于自然语言处理的搜索和推荐技术推荐效果并不尽如人意 [48–50]。

在测试代码推荐领域，Werner Janjic[14] 等人发现测试代码开发是一项劳动密集型工作，需要开发人员拥有丰富的测试知识且能保持高度注意力。通过复用已有测试用例可以减少人工劳动，进而减少软件开发成本。他们构建了一个包含大量 JUnit 测试文件的语料库以及配套的搜索工具。当外部发起一次包含方法签名的查询时，通过匹配测试方法的方法签名，将语料库中具有相似方法签名的测试方法进行推荐。

测试代码推荐也用于向缺乏测试经验的开发人员推荐有价值的上下文测试代码示例，帮助他们学习编写测试代码。R.Pham 等人 [7] 发现开发人员倾向于学习项目中已经存在的测试代码来辅助自己进行测试开发。他们也开发了一款 “Test Recommender” 工具，使用版本控制工具分析代码变更，从事先构造的测试代码语料库中检索到代码变更所在类相关的测试代码进行推荐。但若项目中缺乏其他测试用例，缺乏经验的开发人员就难以通过学习现有测试用例进行新测试用例的编写。

本人先前的工作 HomoTR[16]，主要面向缺乏测试经验的开发人员。它通过度量两种方法的同源性来实现测试代码推荐。如果待测方法与已有的有测试代码的方法具有同源性，HomoTR 将向待测方法推荐测试代码。通过集成于具有可视化结果展示（如分支覆盖率）的在线编程平台 MoocTest（该平台包含了用于测试代码推荐的待测方法及其测试代码语料库），HomoTR 能够帮助开发人员更好地了解测试过程。

### 1.2.3 测试用例修复

测试用例修复主要针对软件演化后，原有测试用例在回归测试时失效的问题 [17]，因为软件演化很可能导致原有测试方法行为变更或测试环境变化。常见的测试用例修复主要面向 GUI 测试、Web 测试和白盒测试 [51]。

现实中 GUI 程序构造的小变化就能导致大量回归测试用例不可用。Memon 等人提出的 GUI 测试修复方法主要依据事件流图解决回归测试时测试用例不可

用的情况 [52]。Grechanik 等人提出一种基于 GUI 程序对象变更进行维护黑盒测试脚本的 GUI 测试用例修复方法 [53, 54]。

当代广泛的 Web 应用也催生出对于 Web 测试用例修复的需求。Harman 等人提出了一种消除不可用的 URL 请求序列和参数集合的 Web 测试用例修复方法 [55]。Choudhary 等人提出的 Web 测试用例修复方法面向版本变更时 Web 页面元素变化导致测试用例失效的问题 [56]。

白盒测试用例修复的研究较多，Daniel 等人开发了一款工具 ReAssert 用于修复 Java 单元测试用例的 assertiion error[57, 58]，并在此基础上提出了基于符号执行的测试用例修复方法 [59]。Mirzaaghaei 等人开发的 Eclipse 插件 TestCareAssistant 用于修复方法返回值和参数改变造成的编译错误 [60]。Hao 等人将机器学习应用于测试用例失效原因分类 [61]。

在软件演化过程中，可能会出现回归测试时的部分测试用例失效的情况 [17]，这一系列测试用例的行为可分为以下四类：

(1) 测试用例执行通过；

(2) 测试用例执行时发生 runtime exception；

(3) 测试用例执行时发生 assertion error；

(4) 测试用例无法正常编译。

其中，执行时发生 runtime exception 的测试用例往往属于难以修复的过时测试用例，因此在实践中将删除这些测试用例；Daniel 等人针对测试用例执行时发生 assertion error 的问题进行了研究 [57, 58]；而导致测试用例无法编译的原因多种多样，如软件演化中包含具体代码修改、变量名修改、程序文件移动、方法声明演化等。

## 1.3 本文主要工作

为能更好地支撑后续测试代码推荐与修复工作，本文对本人先前工作 HomoTR[16] 的原始 Java 代码数据集进行了标准化工作。将一部分待测方法的代码文件进行基于集成开发环境（IDE）自带的代码格式化工作，修改或补充其对应的测试用例代码，从而能够顺利的进行程序分析并构建包含待测方法即其测试代码特征的测试代码复用语料库。对于需要进行测试但还没有测试用例的外部待测方法，本文通过程序分析获取其代码特征，利用这些代码特征在语料库中匹配具有相似或相同功能的待测方法，然后将语料库中待测方法对应的测试代

码进行推荐，再将推荐的测试代码进行修复，以尽可能减少甚至消除人工修改成本的方式，成为符合外部待测方法需求的能够直接进行黑盒测试的测试代码。同时，本文提供了一个基于跨项目测试代码推荐 Web 应用，用户可以在该 Web 应用中输入外部待测方法，获得可复用测试代码，用户可以使用该测试代码进行黑盒测试，或作为测试代码模板，在修改测试数据和待测方法执行的预期结果后获得新的测试用例代码。

本文主要贡献如下：

(1) 对原始的 Java 代码数据集中测试代码进行标准化处理，形成一套测试代码模板，方便后续测试代码自动复用工作；

(2) 实现了一个待测方法及其测试代码语料库的自动化构建工具，从而实现语料库的快速构建，方便后续语料库规模的扩展；

(3) 改进了本人先前工作 HomoTR[16] 中的测试代码推荐技术，在代码相似性度量阶段使用比 HomoTR 更合理的标准，减少了有效代码遗漏，提高了推荐成功率；

(4) 实现了对已推荐测试代码的修复工作，从而能直接使用该测试代码对外部待测方法进行黑盒测试，减少或消除人工修改的成本。

(5) 实现了基于测试代码自动复用技术的 Web 应用，用户在 Web 应用界面中可输入缺少测试用例的待测方法代码后，在 Web 应用界面上将获得可复用测试代码。

## 1.4 本文组织结构

本文的组织结构如下所示：

第一章绪论。本章介绍了软件测试与单元测试的重要性，分析了测试代码复用技术对提高测试效率的作用，简介了国内外对相关技术的研究现状，说明了本文的主要工作与组织结构。

第二章相关工作。本章介绍了与本文工作有关的方法与技术。包括程序切片技术与现有 Java 程序切片工具，代码相似性的度量方法，测试代码复用与测试代码推荐技术的主要情况，以及常见测试代码修复技术对本文中跨项目测试代码修复工作的借鉴意义。

第三章跨项目测试代码自动复用技术。本章阐述了本文提出的跨项目测试代码自动复用技术的具体实现过程，包括代码语料库自动构建、测试代码推荐和

测试代码自动修复三大部分。展示了基于跨项目测试代码自动复用技术的 Web 应用页面和使用情况。

第四章实验设计与分析。本章对本文实验设计与分析内容进行了整理。本章介绍了代码数据集的标准化工作，评估了第三章中描述的跨项目测试代码自动复用技术的有效性，对跨项目测试代码自动复用技术在各个阶段的细节进行了分析，并指出了该技术的有效性威胁。

第五章总结与展望。本章总结了本文的主要工作和成果，讨论了跨项目测试代码自动复用技术的未来改进方向。

-

# 第二章　相关工作

本章对本文中使用的部分技术、工具、方法和概念进行简要说明。

## 2.1　程序切片

在计算机编程中，程序切片是对一组程序语句的计算。程序切片可以用于调试，以便更容易地定位错误源。切片的其他应用包括软件维护、优化、程序分析和信息流控制。

程序切片主要包括静态程序切片和动态程序切片。静态程序切片应用于源代码，除了源代码之外没有其他信息。基于 Weiser 的原始定义 [62]，对于程序 $P$ 中的静态程序切片 $S$，如果其中的语句 $x$ 中含有变量 $v$，则 $S$ 包含了所有影响 $v$ 的值的语句。切片是为切片标准 $C = (x, v)$ 定义的，其中 $x$ 是程序 $P$ 中的语句，$v$ 是 $x$ 中的变量。对于任何可能的输入，静态程序切片包括所有可以影响语句 $x$ 处变量 $v$ 的值的语句。静态程序切片是通过回溯语句之间的依赖关系来计算的，更具体地说，为了计算 $(x, v)$ 的静态程序切片，我们首先在遇到语句 $x$ 之前找到所有可以直接影响 $v$ 值的语句。递归地，对于每个可以影响语句 $x$ 中 $v$ 值的语句 $y$，我们计算 $y$ 中影响 $v$ 值的所有变量 $z$ 的切片。所有这些切片的并集是 $(x, v)$ 的静态程序切片。

Bogdan Korel 和 Janusz Laski 引入了动态程序切片，它适用于程序的特定执行（对于给定执行的跟踪）[63]。动态程序切片的目的是利用有关程序特定执行的信息。如果在某次实际执行时，一些语句影响了特定语句 $x$ 中的变量 $v$ 的值，则这些语句组装成动态程序切片。但是，对于不在本次执行、而在其他执行时影响 $v$ 的值的语句，将不包含在该动态程序切片中。

下面这个例子可以说明静态程序切片和动态程序切片之间区别。考虑一小块程序单元，其中有一个包含 if-else 块的迭代块。if 和 else 块中都有一些对变量有影响的语句。在静态程序切片的情况下，由于无论程序的特定执行如何，都会查看整个程序单元，因此两个块中受影响的语句都将包含在切片中。但是，在动态程序切片的情况下，我们考虑程序的特定执行，其中 if 块被执行，而块中受影响的语句 else 不被执行。所以，这就是为什么在这个特定的执行案例中，动态程序切片将只包含 if 块中的语句。

JavaParser 是一个将 Java 代码转换成抽象语法树（AST）的开源程序静态程

序切片工具[1]。JavaParser 可对输入的 Java 代码进行词法和语法分析，然后输出一个内部结构为抽象语法树（AST）的 Compilation Unit。

AST 即通过树状结构抽象表示代码的语法结构，常用于编译器。作为一种抽象结构，它不依赖于具体文法，提高了编译器的可维护性。它也不依赖语言具体细节，当然这也导致 AST 无法表示出语法中的每一个细节。

JavaParser 生成的 Compilation Unit 内的抽象语法树上节点包含了代码中的特征信息，如图2.1所示，包括包声明 PackageDeclaration，单一类型包导入声明 SingleTypeImportDeclaration，类或接口声明 ClassOrInterfaceDeclaration、方法声明 MethodDeclaration、类变量声明 FieldDeclaration、方法调用表达式 MethodCallExpr、变量声明 VariableDeclarator 等。

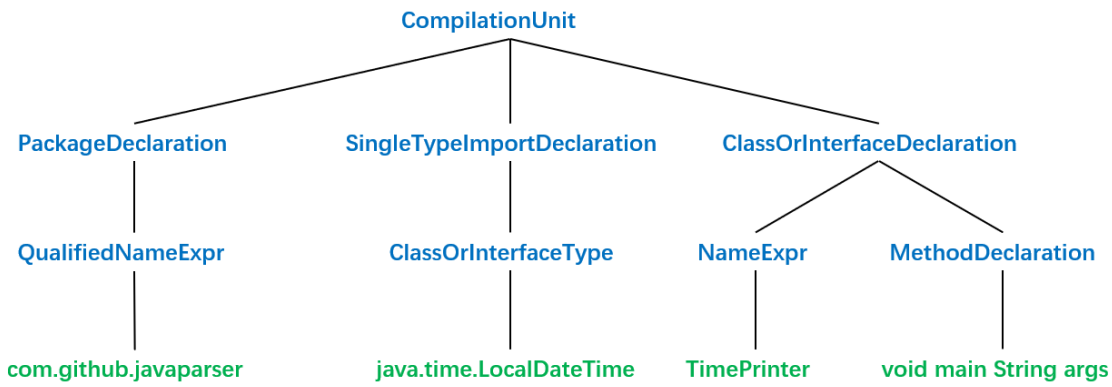


图 2.1: CompilationUnit

而访问方法声明 MethodDeclaration，如图2.2所示，还可以进一步获取到其内部的方法返回类型 Type，方法名表达式 NameExpr，方法输入参数 Parameter，块语句 BlockStmt 等的具体信息。因此，通过遍历 Compilation Unit 可获得代码包含的几乎所有信息。

由于 JavaParser 是一款开源工具，且使用方便，维护更新和社区讨论十分活跃，因此本文使用 JavaParser 作为程序切片和代码解析工具。

## 2.2 代码相似性度量

在计算机科学中，两个字符串的不同程度通过计算两者间相互转换所需的操作数进行衡量，而最小操作数即为字符串编辑距离。字符串编辑距离常用于

[1]https://github.com/javaparser/javaparser

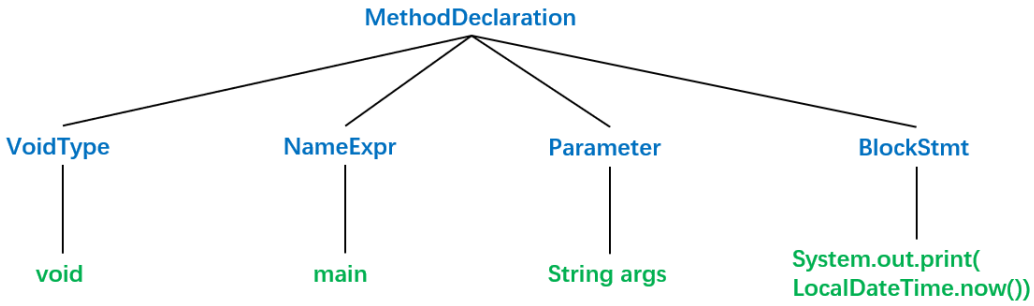


图 2.2: MethodDeclaration

自然语言处理，如拼写校正。对于一个拼写错误的单词字符串 $s$，如果能在字典中能够找到与该字符串编辑距离较近的新字符串 $s'$，则 $s'$ 就作为 $s$ 的候选校正。这种用途即为字符串编辑距离在拼写校正领域的具体应用。字符串编辑距离的不同定义使用不同的字符串操作集[2]。周汉平等人提出了一种基于 Levenshtein 距离应用于编程题自动评阅的方法 [64]，用于评价一个无法正常运行的程序的正确度，是一种将字符串编辑距离应用于代码相似性度量的尝试。本文使用了 Levenshtein 距离和 Cosine 距离。

Levenshtein 距离的衡量依据字符串中字符的新增、修改、删除操作的操作数。Levenshtein 距离主要应用于拼写检查、光学字符识别校正以及基于翻译记忆库的自然语言辅助翻译。若开发人员对一段字符串，如方法名拼写发生错误时，目标方法名与实际方法名往往仅有个别字符不同。通过拼写校正对字符串匹配进行模糊处理，可以减少因为此类错误造成的匹配结果遗漏。

若两个字符串 $a, b$ 的长度分别为 $|a|, |b|$，则 $a, b$ 之间的 Levenshtein 距离 $lev(a, b)$ 为：

$$
\mathrm{lev}(a,b) = \begin{cases} |a| & \text{if } |b| = 0, \\ |b| & \text{if } |a| = 0, \\ \mathrm{lev}(\mathrm{tail}(a), \mathrm{tail}(b)) & \text{if } a[0] = b[0], \\ 1 + \min \begin{cases} \mathrm{lev}(\mathrm{tail}(a), b) \\ \mathrm{lev}(a, \mathrm{tail}(b)) \\ \mathrm{lev}(\mathrm{tail}(a), \mathrm{tail}(b)) \end{cases} & \text{otherwise,} \end{cases}
$$

[2]https://en.wikipedia.org/wiki/Edit_distance

本文并未采用原始的 Levenshtein 距离计算方式进行拼写校正，而是使用对称删除算法改进基于 Levenshtein 距离的拼写校正。它降低了基于编辑候选字符串生成与字典查找的 Levenshtein 距离计算复杂性。相比原始 Levenshtein 距离计算需要使用新增、修改、删除操作，对称删除算法只需要进行删除。避免了原始 Levenshtein 距离在面对如中文字符串处理时，需要进行海量的字符新增和修改操作，实现了语言无关性。

由于只进行低成本的删除操作生成候选字符串生成和预计算，在最大字符串编辑距离为 3 以内，平均 5 个字母的单词有大约 300 万个可能的拼写错误，而对称删除算法秩序生成 25 个经过删除操作的候选字符串即可覆盖所有这些错误[3]。详细的基于对称删除算法在拼写校正匹配中应用的代码实现会在第三章的测试代码推荐部分给出。

本文还采用了余弦相似度用于字符串与文本相似性度量，Cosine similarity = 1 - Cosine distance。在数据分析中，对于两个数字序列之间相似度的度量结果，我们称作余弦相似度。序列被视为内积空间中的向量，余弦相似度定义为它们之间夹角的余弦值，即向量的点积除以它们长度的乘积。由此可见，余弦相似度不取决于向量的大小，而仅取决于它们的角度。余弦相似度总是属于区间 $[-1, 1]$。例如，两个比例向量的余弦相似度为 1，两个正交向量的相似度为 0，两个相反向量的相似度为-1。余弦相似度主要在正空间中使用，结果被整齐地限制在 $[0, 1]$。

在信息检索和文本挖掘中，每个单词被分配一个不同的坐标，并且文档由文档中每个单词出现次数的向量表示。然后，余弦相似度提供了一个有用的度量，可以衡量两个文档的相似程度，并且独立于文档的长度。余弦相似度的一个优点是它的复杂度低，特别是对于稀疏向量，只需要考虑非零坐标。

两个非零向量的余弦可以使用 Euclidean 点积公式导出：

$$\mathbf{A} \cdot \mathbf{B} = ||\mathbf{A}||||\mathbf{B}|| \cos\theta$$

给定两个属性向量 A 和 B，余弦相似度 $cos(\theta)$ 的计算公式为：

$$\text{cosine similarity } = S_C(A, B) := \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{||\mathbf{A}||||\mathbf{B}||} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}}$$

由于代码文本语义与自然语言语义差别很大，而在不考虑语义情况下，可通过余弦相似度较好地度量文本相似性，且余弦相似度计算速度很快，因此本文采用余弦相似度度量待测方法具体代码的相似度。

[3]https://github.com/wolfgarbe/symspell

此外，本文同样用余弦相似度度量替代了 HomoTR 中基于 Word2Vec 的语义相似度度量方法名相似性。因为本质上，采用 Word2Vec 度量字符串相似度也是把字符串单词表示为向量，然后计算向量间的余弦相似度。考虑到工具集成外部 Word2Vec 模块后耦合性较高且 Word2Vec 模型加载速度较慢，本文用余弦相似度度量方法名相似性。

如周汉平等人所述，字符串编辑距离可以用来衡量代码相似度 [64]。同时，鉴于代码克隆表示软件系统中存在相似或相同的源代码片段 [34]，因此可以使用代码克隆检测技术度量代码相似性。对于已经经过测试、拥有测试用例的待测方法 $m_1$ 和未经测试的待测方法 $m_2$，如果 $m_1$ 和 $m_2$ 具有相同或相似的功能，则 $m_1$ 和 $m_2$ 的代码往往具有相似性，因此实现测试代码复用可以使用代码相似性度量方法。

如今已经有大量关于代码克隆检测和代码相似度度量方法被提出。Bellon 等人评估了 6 个代码克隆检测工具 [35]，这些工具主要从文本、词汇、语法信息、程序依赖图等方面进行代码克隆检测。Ragkhitwetsagul 等人评估了 30 种代码相似性检测技术和工具 [36]，发现相同配置的检测技术和工具在不同数据集上性能差别可能很大。

这些代码克隆检测和代码相似性检测技术主要依据代码文本和结构进行检测。根据代码文本进行检测的工具中主要利用字符串和令牌进行代码克隆检测或相似性检测。在使用字符串的方法中，Ducasse 等人实现了一种独立于语言、不需要解析的检测大量代码重复的可视化方法 [37]；Johnson 等人使用指纹识别来对大型程序源代码文本中的重复部分进行精确检测，并构建了一套可视化的理解程序的工具 [38]。

在使用令牌的方法中，Li 等人使用数据挖掘技术有效地识别出大型软件套件中的克隆代码 [39]；Sajnani 等人使用优化的反向索引快速查询给定代码块的潜在克隆 [40]，其基于令牌排序的过滤试探法用于显著减少索引的大小、检测克隆所需的代码块比较数量，以及判断潜在克隆所需的令牌比较数量；Wise 等人则将代码相似性检测技术用于检测学生提交的计算机程序和其他文本中的涉嫌剽窃 [41]。

根据代码结构进行检测的工具主要基于树结构和图结构进行代码克隆检测或代码相似性检测。在使用树结构的方法中，Baxter 等人通过使用抽象语法树（AST）来检测程序源代码中任意程序片段上的精确克隆和未遂克隆 [42]；Jiang 等人提出了一种识别相似子树的有效算法 [43]，并将其应用于源代码的树表示，实现了一个可扩展的代码克隆检测工具 DECKARD；Zhang 等人指出，传统的基于信息检索的表示源代码的方法通常将程序视为自然语言文本，这可能会遗

漏源代码的重要语义信息，而基于 AST 的神经网络模型可以更好地表示源代码，因此他们提出了一种基于 AST 的神经网络 ASTNN 来表示源代码并成功应用于源代码分类和代码克隆检测两项程序理解任务 [44]。

在基于图结构的方法中，Komondoor 等人实现了一个使用程序依赖图（PDG）和程序切片来寻找表示克隆的同构 PDG 子图，进而检测出克隆代码的工具 [45]；Krinke 等人同样提出了一种利用程序依赖图识别程序中相似代码的方法，它既考虑了程序的语法结构，也考虑了其中的数据 [46]；Zhao 等人提出了一种新的方法 DeepSim[47]，将代码控制流和数据流编码成一个语义矩阵，其中每个元素都是一个高维稀疏二进制特征向量，并设计了一个新的深度学习模型，该模型基于这种表示来度量代码功能相似性。通过连接从代码对中学习到的隐藏表示，该新模型将检测功能相似代码的问题转化为二进制分类，可以有效地学习具有非常不同语法的功能相似代码之间的模式。

以上这些方法各有其优缺点。在度量代码相似性的准确度方面，最新的方法肯定比以往的方法拥有更高的准确度，但是其实现的复杂度也更高，检测耗时也会增多。

在本文中，跨项目测试代码自动复用技术通过度量两个待测方法的功能相似性，将已有的测试用例代码复用给没有测试用例的待测方法。考虑到本技术需要高效复用测试代码实现对待测方法的直接测试，因此使用的代码相似度度量方法将是较低复杂度、轻量级的。

## 2.3 测试用例复用

由于当今软件功能复杂度越来越高，人们对于软件质量的要求也越来越高，软件复用主要是指在新软件的开发中利用已有软件中的有效成分，积极地调动其旧软件开发的所有知识进行合理的构建，从而使软件设计更新的速度不断加快 [18]。软件复用作为一种提高软件开发质量和效率的技术引起了开发和研究人员的重视。

软件复用提高了软件质量和生产效率，同时缩短了软件产品的上市时间。测试工作消耗了所有开发工作的一半以上，是阻碍软件质量保证的重要因素之一。正如复用思想适用于软件开发过程一样，它也适用于测试过程 [65]。因此，测试人员将复用思想应用到软件测试领域，从而让缺乏经验的测试人员借鉴前人经验，提高测试效率。测试用例复用就是将一个软件已经执行过的测试用例在不同时期、不同程度地应用到同一软件新的测试或是同类软件的测试中 [9]。

目前在产业实践中已有大量关于测试用例复用的研究，在包括指挥信息系统 [29]，遥感软件 [23, 24]，信息安全软件 [20, 21]，舰船装备软件 [27]，航空机

载软件 [30]，航天测控软件 [22]，雷达软件 [25, 26, 28] 等一系列对软件质量具有极高要求的软件系统中得到应用。对于这些安全攸关的软件系统，由于同一类型的软件系统之间功能相似性很高，导致相互之间也具有很高的关联性和继承性 [66]。因此可以从测试用例的复用方面考虑解决当前安全攸关的软件测试存在的困难 [67]。

芮素娟等人给出了测试用例复用的类型 [19]，包括：

(1) 同一软件在不同测试阶段的复用。最典型的例子就是将单元测试用例用到集成测试中。

(2) 统一软件在不同时间下的测试用例复用。即随着软件演化，旧版本软件的测试用例被应用到新版本软件的测试中。这种情况就会出现下文会说到的回归测试用例失效和修复问题。

(3) 类似软件之间的测试用例复用。即同类软件在测试的用例设计、策略、数据、步骤等有相似之处，借鉴已有测试用例，编写新的测试用例，提高测试效率。这种类型即时本文所要应对的情况。

同时，芮素娟等人也总结了可复用设计用例设计指导原则 [19]，包括测试用例间相关性降至最低、对被测软件依赖尽量减弱、描述规范、不含常量、输入值替换为变量、内容完整、结构统一、分类合理。

进一步地，肖良等人提出了可复用测试用例的特性以此来判断测试用例是否可用 [68]，陈强等人则将可复用测试用例的特性归纳为标准化、独立性、通用性、有效性、小粒度 [69]，王珊珊等人的观点与之类似，即可复用测试用例的必要质量特性，包括通用性、独立性、规范化和可维护性 [9]。通用性要求复用的测试用例不能过分依赖于测试环境，不能体现有具体场景的个性化内容；独立性要求可复用测试用例的成功执行不能依赖于其他测试用例的执行状态；规范化要求依据一定规范编写可理解的测试用例设计方案；可维护性要求在一个成熟的、可靠的可复用测试用例库中的测试用例，仅需稍加修改就能应用于同一或相似领域。

Li 等人提出了一个测试用例复用模型 [70]，以解决软件测试用例设计的规范性、有效性和效率问题。测试用例复用模型是在有类似测试需求特征的测试用例集中进行提炼、提取和表达测试用例的一种设计方法，它由软件测试专家基于测试用例库进行开发，以保证测试设计的规范性和有效性。而袁松等人则根据可复用测试用例的特性，给出了一种基于层次分析法的测试用例可复用性度量方法 [71]。

Zhang 等人实现了面向复用的测试用例管理框架 [31]，确定了维护测试用例复用、回归测试用例复用、水平（跨项目）测试用例复用和垂直（同项目）测试用例复用四大维度和可用性、独立性、适应性、标准化和可信度五大特征，通过实验证明可复用的测试用例及其支持过程可以提高工作效率，激发团队成员的主观能动性。

测试用例复用的优点也由此体现 [9]，比如能够缩短测试周期，提高测试和可靠性，降低测试费用和软件成本，一定程度上解决软件开发人员测试经验不足的问题。但是难点也不容忽视，比如测试用例分类标准难以确定，对测试用例库的组织、存储和维护要做到科学管理，测试用例可复用度缺乏合理、准确的度量标准以及可复用测试用例库需要对自身的灵活性和可靠性有一定要求。

此外，也有许多将测试用例复用与测试需求相结合的例子。比如从软件测试需求层面构建软件测试经验库、进而复用测试用例 [72–74]，Noack 等人实现了需求与测试用例之间的完整跟踪与自动链接，从而实现基于复用需求的测试用例自动化复用 [75]。

然而，不管是以上关于测试用例可复用性研究，还是基于 CBR（案例推理）的测试用例复用方法 [76–78]，基于 LDA 模型 [79] 或是分词搜索 [69] 的测试用例复用方法，它们往往是对用自然语言书写的测试用例文档和设计方案进行复用，而非测试代码。这在安全攸关软件系统如军事和空间科技领域软件的测试用例复用 [20–30, 66] 也中尤为体现。张娟等人则提出基于 Z 规格说明的可复用测试用例形式化描述方法来保证测试用例描述的准确性和无二义性 [80]。

不过，也有关于具体测试代码复用的例子。Fischer 等在研究中发现，复制现有系统并定制它们以满足客户特定需求等临时做法很普遍。为了跟上这种日益增长的定制软件系统的需求，他们开发了一个基于 Eclipse 的框架工具 ECCO。ECCO 能够自动从以前开发的产品中找到可重用的部件，然后根据所需功能的选择合成新产品，实现软件复用 [81, 82]。进一步地，Fischer 基于 ECCO 开发了一种软件测试自动复用方法并进行了在这种高可配置软件中复用测试用例的实验，发现复用自动生成的测试用例变体可以实现比直接复用原有测试用例实现更高的测试成功率，证明了高可配置软件测试用例的自动重用可以大大减少调整现有测试用例的工作量 [83, 84]。

本文研究的是对测试代码的复用，构建的也是基于待测方法及其测试代码的代码语料库，而非测试用例设计方案的文档语料库。因此，本文需要做的是对代码及程序进行分析，而非理解用自然语言描述的测试用例文档。上述测试用例复用中的一些观点，比如可复用测试用例必要的质量特性等，非常值得在测试代码复用中借鉴。可复用的测试代码，其编写必定是标准化和规范化的，也

必须具备独立性、通用性、有效性等特点，因此，在进行测试代码复用前，对代码数据集进行标准化处理，并以此构建测试代码复用库就显得十分必要。此外，本文将基于两个由不同人编写、但是内容存在相似性的项目间进行测试代码复用，故本文中实现的是一种跨项目的测试代码自动复用技术。

### 2.3.1 测试代码推荐

如上所述，测试代码复用是提高测试代码编写效率的重要思想，而测试代码推荐作为一种实现测试代码复用的具体方式也逐渐受到重视。测试代码推荐主要面向两个具有相同或相似功能的待测方法 $m_1$ 和 $m_2$，如果 $m_1$ 存在测试代码 $TC_1$，而开发人员刚编写完的 $m_2$ 没有进行测试，则将 $TC_1$ 推荐给 $m_2$，以供开发人员参考来编写用于测试 $m_2$ 的测试代码。

Listing 2.1: bubbleSort1()

```
public void bubbleSort1(int[] arr) {
    for(int i = 0; i < arr.length - 1; i++){
        for(int j = 0; j < arr.length - 1 - i ; j++){
            if(arr[j] > arr[j + 1]){
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

如代码2.1和代码2.2所示，待测方法 bubbleSort1() 显然与待测方法 bubbleSort2() 均实现了对数组 arr 进行冒泡排序的功能，因此，若 bubbleSort1() 方法存在测试代码 $TC_1$，而 bubbleSort2() 方法尚未被测试，则可推荐 $TC_1$ 用于参考来编写测试 bubbleSort2() 的测试代码。

而 R.Pham 等 [7] 发现，开发人员倾向于学习项目之前存在的测试代码来学习编写新的测试代码，这也成了本文以及之前工作 HomoTR[16] 诞生的依据，即主要面向开发人员经常用来练习的在线编程题。测试代码推荐技术主要包括同项目推荐 [7, 15] 和跨项目推荐 [13, 14, 16] 两类，但现有测试推荐技术往往存在一些缺点，如同项目推荐 [7] 中只有本项目内的测试用例代码可被推荐，处于初始阶段的项目，因为测试用例代码较少，推荐效果不佳。而仅使用方法签名进行待测方法相似度度量如 Janjic 等人的工作 [14]，同样会导致推荐精度下降。

Listing 2.2: bubbleSort2()

```
public void bubbleSort2(int[] list) {
    int temp = 0;
    for (int i = 0; i < list.length - 1; i++) {
        for (int j = list.length - 1; j > i; j--) {
            if (list[j - 1] > list[j]) {
                temp = list[j - 1];
                list[j - 1] = list[j];
                list[j] = temp;
            }
        }
    }
}
```

本人先前工作 HomoTR 基于 Java 项目进行测试代码推荐，如算法1所示, 基于方法签名匹配和松弛算法，通过增加度量相似性的标准，做到了比 Janjic 等人更好的推荐精确度 [14]。但是受当时思维局限所致，部分度量标准不合理，如在基于拼写校正的匹配（第 3 行）、基于语义相似度的匹配（第 5 行）和基于通配符的匹配（第 6 行）阶段，选择直接判断参数类型列表和返回类型是否相同，遗漏了大量潜在匹配结果，因为在 Java 语言中，存在数据类型不同，但是实际作用类似的数据类型，如整型数组 int[] 和整型列表 List<Integer>，且由于装箱拆箱机制的存在，基础数据类型都有其装箱后的类型类存在，如整型 int 的装箱后类型类为 Integer。因此，HomoTR 对于参数类型的判断是有明显缺陷的。

**Algorithm 1:** HomoTR 待测方法匹配算法

```
Data: name^c, name^m, types^param, type^return, code
Result: target_method
result ← match^strict(name^c, name^m, types^param, type^return);
if size(result) = 0 then
    return ← match^spelling(name^c, name^m, types^param, type^return);
    if size(result) = 0 then
        result ← match^semantics(name^c, name^m, types^param, type^return);
        result ← result ∪ match^wildcard(name^c, name^m, types^param, type^return);
    end
end
code_similarity_map;
for method in result do
    code^r ← get_code(method);
    similarity ← cosine_similarity(code, code^r);
    key ← method;
    value ← similarity;
    code_similarity_map ← map(key,value) ∪ code_similarity_map;
end
similarity_max ← 0.0;
target_method;
for map in code_similarity_map do
    similarity^r ← get_value(map);
    if similarity^r > similarity_max then
        similarity_max ← similarity^r;
        target_method ← method^r;
    end
end
return target_method;
```

由于在实践过程中发现如上所述的推荐标准仍然不合理、被推荐后的测试代码仅能作为参考而无法直接使用等客观缺陷，因此改进 HomoTR 的这些方面就提上日程，这也成为了本文诞生的初衷。

### 2.3.2 测试代码修复

软件演化通常包括缺陷修复、功能扩展、软件重构、性能提升等行为，这些行为改变了软件行为，很可能会导致原有测试用例在回归测试时不可用，即测试用例在新版本程序中执行失败 [17]。而测试用例修复则是在测试用例执行失败后对其进行修复，使其能够重新检测出程序缺陷。现有测试用例修复主要针对回归测试时 GUI 测试 [52–54, 85],Web 测试 [55, 56, 86, 87] 和白盒测试用例失效 [57–61] 的情况。

**（1）GUI 测试用例修复**

针对 GUI 测试用例修复，Memon 等人提出了一种解决回归测试时面向 GUI 程序测试用例失效情况的修复方法 [52]，GUI 测试用例失效主要是因为起始状态错误、中间事件执行顺序不合理，前者无法修复，后者可根据 GUI 事件流图（EFG）修复 GUI 测试用例。Grechanik 等人对 GUI 程序对象变化与其对应的测试脚本进行研究。通过将两者建立关系，确定哪些对象被修改，从而找出测试脚本中哪些部分被修改所影响 [53, 54]。Huang 等人提出一种自动修复失效 GUI 测试用例的方法 [85]。目前 GUI 测试用例修复修复范围有限 [51]。

**（2）Web 测试用例修复**

由于 Web 在日常生活中应用广泛，因此通过回归测试提高 Web 应用可靠性非常常见。Web 回归测试用例修复旨在提高 Web 回归测试效率。在 Web 测试用例修复领域,Choudhary 等人定义了导致 Web 测试脚本错误的几种种原因 [56]，实现对 Web 页面元素 expected 值与 actual 值不同时测试用例执行失败的自动修复，以及 Web 页面元素位移或变化导致测试用例执行失败的自动修复。Harman 等人提出用来消除无效 URL 请求序列和错误参数集合的修复方法 [55]。与 GUI 测试用例修复相似，Web 测试用例修复方法也较为简单。

**（3）白盒测试用例修复**

研究人员对于白盒测试用例修复的关注则更高。Daniel 等人开发了一款修复 Java 单元测试用例发生 assertion error 的辅助工具 ReAssert[57, 58]，并在此基础上提出一种基于符号执行的修复方法（symbolic test repair）[59]。但若程序不能通过编译，则不能进行修复。Mirzaaghaei 等人开发出一款针对方法返回值和参数改变引起的简单编译错误进行修复的 Eclipse 插件 TestCareAssistant[60]，它使用动静态结合的程序分析技术来识别代码变更中涉及的变量及其初始化值，从而修改测试用例并保留其行为。Hao 等人则使用 best-first decision tree learning 对测试用例失效的原因进行分类 [61]。

总而言之，由于软件演化十分频繁，导致回归测试的测试用例集也不断变化，

因此大量研究人员关注回归测试时测试用例失效以及如何进行修复的问题。目前测试用例修复主要面对以下问题 [88]：（1）难以判定测试用例是否可修复；（2）现有测试用例修复应用领域狭窄；（3）缺乏有效的测试用例修复工具；（4）难以提高测试用例修复技术准确性。

张琼宇等人对测试用例修复问题进行了分析 [17]。在白盒测试用例修复场景中，软件演化导致原有的一系列测试用例中，部分测试用例不可用。所有回归测试用例的行为大致可以分为四类：

(1) 测试用例执行通过。这种情况一般不需要进行测试用例修复工作。

(2) 测试用例执行时发生了 assertion error。Daniel 等人针对这类测试用例失效问题进行了研究 [57, 58]，并提出了一种修复发生 assertion error 的测试用例的方法 ReAssert。

(3) 测试用例执行时发生 runtime exception。这往往是因为测试用例过时导致的问题。在实践中，这类难以修复的失效测试用例一般被直接删除。

(4) 测试用例无法编译。引起这类测试用例失效问题的原因很多。软件演化时可能存在诸多行为，比如方法中代码修改，局部变量重新命名，程序中某个方法声明从某文件移动到另一文件，方法发生了声明演化等等。

针对第四种情况，张琼宇等人提出了测试用例修复策略 [17]，包括识别程序中方法在演化前后的关联，分析方法声明演化类型，根据不同方法声明演化类型制定专门的修复策略。通过使用动态符号执行技术分析测试用例，并以文本形式自动生成测试用例修复建议，人工确认测试用例修复建议后对测试用例进行修改。

本文的场景是跨项目的测试代码复用，对于复用的测试代码，若不经过修复，肯定无法在新的待测方法所在环境中正常运行，这与测试代码修复有相似之处。本文中的测试代码自动修复部分参考了白盒测试用例修复的思想。

## 2.4 本章小结

本章简要介绍了本文中所涉及的技术和研究方向的相关内容，主要包括程序切片、代码相似性度量、测试代码复用、测试代码推荐和测试代码修复。第一部分介绍了程序切片技术和开源 Java 静态程序切片工具 JavaParser。第二部分介绍了代码相似性度量的各种方法，包括字符串编辑距离和其他更复杂的基于代码文本和结构的代码相似性度量方法。第三部分介绍了测试用例复用的应用现

状以及测试用例复用的分类、设计原则、质量特性和优点，指出本文中测试代码复用和测试用例复用的区别，然后介绍了测试代码推荐的定义和本人先前工作HomoTR[16] 中的测试代码推荐技术，并说明了测试用例修复的几大类型，着重讨论白盒测试用例修复，以及现有测试用例修复方法和技术与本文中需要的测试代码修复的关系。

# 第三章　跨项目测试代码自动复用技术

## 3.1　整体概述

本文中所述跨项目测试代码自动复用技术使用的代码语料库基于一个经过标准化处理的 Java 编程题代码数据集，代码数据集的标准化处理方法详见第四章第 4.1、4.2 节。在这个数据集基础上，本文实现了一个待测方法及其测试用例代码语料库自动构建工具。对于需要进行测试代码复用的外部待测方法，首先使用 JavaParser 对它进行程序切片，提取该方法的特征，如类名、方法名、输入参数类型列表、返回类型、方法中具体代码等。然后基于这些特征进行测试代码推荐，在语料库中匹配到与该外部待测方法功能最为相似或相同的待测方法，推荐其对应的测试用例代码，然后将代码进行自动修复，获得可复用的测试用例代码并输出，使其能够直接执行以测试上述外部待测方法。跨项目测试代码自动复用技术的整体架构如图 3.1 所示。此外，本文还实现了基于该技术的 Web 应用，在 Web 页面可以输入外部待测方法的类名与方法具体代码，然后在页面上获得可复用的测试用例代码。

下文将从代码语料库自动构建、测试代码推荐、测试代码自动修复和 Web 应用界面展示四部分进行叙述。在此需要介绍一些下文中会频繁提到，但是在第二章没有明确定义或描述的概念：

**待测方法**：代码文件中需要进行测试的方法定义为待测方法。特别的，下文中不存在于语料库以及构建语料库的代码数据集中，且作为实验对象、用于实验的待测方法被称为“外部待测方法”；语料库中收集的、且有对应测试用例代码的待测方法被称为“语料库中待测方法”。

**测试用例**：在软件工程中，测试用例是输入、执行条件、测试过程和预期结果的规范，它定义了为实现特定软件测试目标而执行的单个测试，例如执行特定的程序路径或验证符合特定要求 [89]。本文中，测试用例有两种含义。在第二章，它被定义为对软件进行测试任务的自然语言描述的文档，而非代码，与“测试用例代码”或“测试代码”相对。而在第四章即实验部分，它被定义为测试待测方法所需的测试对象、测试方法、测试数据、预期结果等组成的代码集合。

## 3.2　代码语料库自动构建

本文实现了一个代码语料库自动构建工具，它能够将代码数据集自动构建为代码语料库。代码语料库存储了用于测试用例代码自动复用任务的待测方法

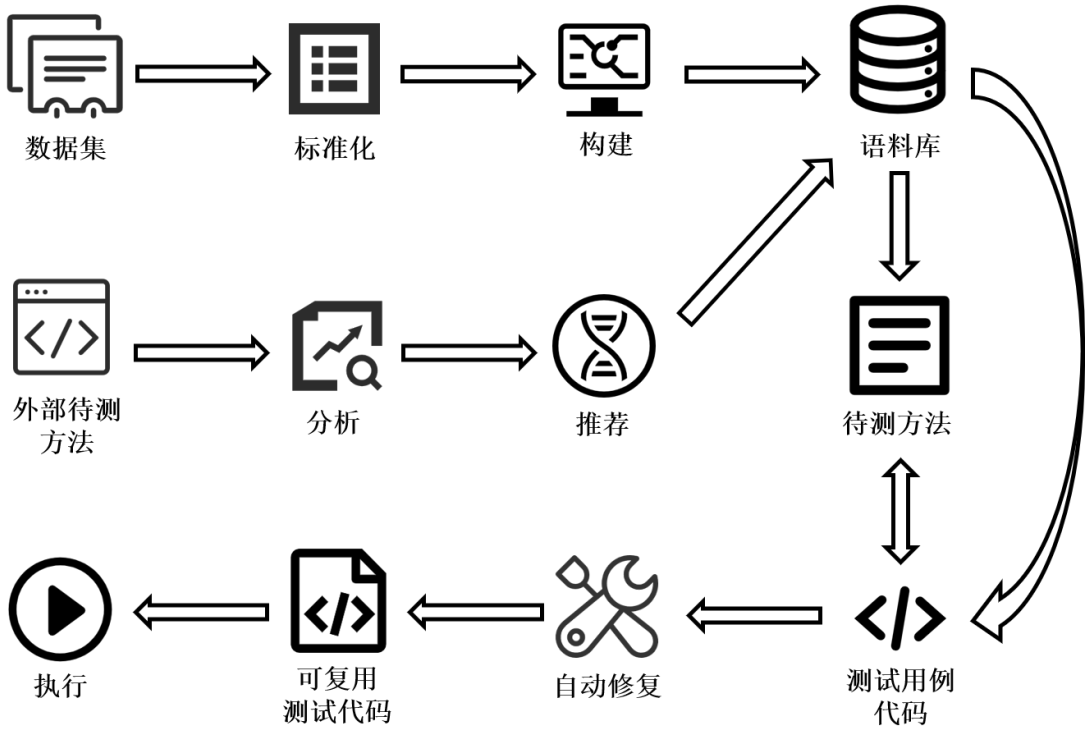


图 3.1: 整体架构

及其测试代码的主要特征信息。代码语料库自动构建工具对外提供两个接口。其一是 constructMUTDataset() 方法（MUT 即待测方法 method under test 的缩写），它实现了对代码语料库中待测方法语料的构建；其二是 constructTCCDataset() 方法（TCC 即测试用例代码 Test case code 的缩写），它实现了对代码语料库中测试用例代码语料的构建。两个方法的输入参数均为具体源码包目录的路径。

提取待测方法所在类的特征信息主要依靠 JavaParser 工具。JavaParser 将待测方法所在类表示成基于抽象语法树的 CompilationUnit（编译单元），对 CompilationUnit 中各节点可访问到的目标类型节点进行提取，然后提取待测方法所在类及其内部类 ClassOrInterfaceDeclaration（类或接口声明），内部类在待测类使用节点内部类 Node 时比较常见，然后提取该类中所有方法 MethodDeclaration（方法声明）和所有类变量 FieldDeclaration（类变量声明）。通过文件名和 ClassOrInterfaceDeclaration 可确定待测方法所在类名。

**Algorithm 2:** 确定目标待测方法

```
Data: class
Result: target_method
1  compilation_unit ← java_parser_parse(class);
2  class_variables ← find_all_class_variables(compilation_unit);
3  method_list ← find_all_method(compilation_unit);
4  main_method ← find_main_method(method_list);
5  if main_method != null and size(method_list) > 1 then
6      method_call_list ← get_all_method_call(main_method);
7      method_call_list ← reverse(method_call_list);
8      for method_call in method_call_list do
9          if has_method_call(method_call, method_list) then
10             target_method ← get_target_method(method_call, method_list);
11     end
12 else if main_method != null and size(method_list) = 1 then
13     target_method_core_model ← main_method;
14 else if main_method = null and size(method_list) > 0 then
15     union_find_set ← make_set(method_list);
16     for father_method in method_list do
17         method_call_list ← get_all_method_call(father_method);
18         for method_call in method_call_list do
19             if has_method_call(method_call, method_list) then
20                 son_method ← get_target_method(method_call,method_list);
21                 union(father_method, son_method);
22         end
23     end
24     target_method ← find_father(union_find_set);
25 return target_method;
```

确定目标待测方法的算法如算法 2 所示，它的输入即为整个待测类 class，输出为目标待测方法 target_method，具体解释如下：

首先利用 JavaParser 对待测类进行切片，生成对应的编译单元 compilation_unit（第 1 行），然后从 compilation_unit 中提取出所有的类变量 class_variables（第 2 行）、有序的类方法列表 method_list（第 3 行）和 main() 方法 main_method（第

4 行）。识别目标待测方法 target_method 要分三种情况进行：

（1）如果类中同时存在 main() 方法和其他方法（第 5 行），则提取 main() 方法中的被调用方法列表 method_call_list（第 6 行），同时将 method_list 中方法的顺序颠倒（第 7 行），这么做是为了查找 main() 方法中最后被调用的本类中声明的方法。然后遍历 method_call_list 中所有被调用方法 method_call，如果 method_call 的确为类方法列表 method_list 中的一个方法，则将 method_list 中该方法识别为目标待测方法 target_method（第 8-11 行）。

（2）如果类中除 main() 方法外不存在其他方法，则将 main() 方法作为目标待测方法（第 12-13 行）；

（3）如果类中不存在 main() 方法且存在其他方法，则要分析这些方法间的调用关系。首先依据所有声明的方法构建初始的并查集 union_find_set（第 15 行），然后遍历 method_list 中的所有方法 father_method，获取 father_method 中的被调用方法列表 method_call_list（第 17 行），然后遍历 method_call_list 中所有的被调用方法 method_call，如果它被确认为类方法列表 method_list 中的一个方法 son_method，则 father_method 与该方法 son_method 的调用与被调用关系被认为是父子关系（第 20 行），即 father_method 调用了 son_method，并将 son_method 和 father_method 合并（第 21 行）。最后获取并查集中最原始的父亲方法作为目标待测方法（第 24 行）。

**Algorithm 3:** 并查集-find_head()

```
Data: method, father_map
Result: father_method
temp_method;
while father_method != temp_method do
    temp_method ← father_method;
    father_method ← find_head(father_method);
end
key ← method;
value ← father_method;
father_map ← {key,value} ∪ father_map
return father_method;
```

需要指出的是，即便能够通过 main() 方法获取目标待测方法，但仍需要对类中除 main() 以外的方法进行调用关系解析。调用关系解析方法基于并查集，其关键点在于并查集中 find_head() 方法和 union() 方法。

find_head() 方法如算法 3 所示。该方法的目的是查找调用 method 方法的根

方法。father_map 是存储了两个方法调用关系的键值对映射集合，对于每个键值对映射 map，它的键（key）为儿子方法（被调用方法），值（value）为最原始的父亲方法，即根方法（父亲方法为调用儿子方法的方法）。需要指出的是，在并查集结构中，只有当前节点的父亲节点为本身时，当前节点才为根节点。如果父亲方法 father_method 与临时方法 temp_method 不相等（即当前父亲方法不是根方法），则递归寻找根方法，当 father_method 与 temp_method 相等时停止循环（第 2-5 行）。然后向 father_map 放入新的键值对映射，其中 key 为当前方法 method，value 为调用 method 方法的根方法 father_method（第 6-8 行）。

union() 方法如算法 4 所示。方法的输入为 a_method，b_method，size_map，father_map。其中，a_method 和 b_method 为存在调用和被调用关系的两个方法，size_map 用来保存每个方法调用链的长度。假设存在两个方法 a() 和 b()，且存在 a() 中调用了 b()，即 a() 为父亲方法，b() 为儿子方法，则 size_map 中存储了基于这个方法调用链的键值对映射，其中键（key）为最原始的父亲方法，此时是 a()，值（value）为调用链长度，此时为 2。father_map 已在上文叙述。

首先寻找 a_method 和 b_method 的根方法 a_head 和 b_head（第 3-4 行），如果 a_head 和 b_head 不相同，则分别获取各自的方法调用链长度 a_size 和 b_size（第 6-7 行）。如果 a_size <= b_size，则 b_head 为 a_head 的根方法，于是在 father_map 中存储以 b_head 为根方法、a_head 为被调用方法的调用关系（第 8-11 行），同时在 size_map 中更新以 b_head 为根方法的方法调用链长度（第 12-14 行）。如果 a_size > b_size，则在 father_map 中存储以 a_head 为根方法、b_head 为被调用方法的调用关系（第 15-18 行），同时在 size_map 中更新以 a_head 为根方法的方法调用链长度（第 19-21 行）。

需要指出的是，本文中所述的调用关系解析方法中，其 union() 方法一律按 a_method 为被调用方法，b_method 为调用 a_method 的父亲方法进行使用。

本文所述的代码语料库自动构建工具通过基于并查集的解析方法间调用关系，识别目标待测方法。然后将目标待测方法的特征信息组装成目标待测方法模型 methodCoreModel，赋予其唯一标识 methodCoreId 后存储至语料库中。

**Algorithm 4:** 并查集-union()

```
Data: a_method,b_method,size_map,father_map
if a_method = null or b_method = null then
    return;
a_head ← find_head(a_method);
b_head ← find_head(b_method);
if a_head != b_head then
    a_size ← get(size_map,a_head);
    b_size ← get(size_map,b_head);
    if a_size <= b_size then
        key1 ← a_head;
        value1 ← b_head;
        father_map ← {key1,value1} ∪ father_map;
        key2 ← b_head;
        value2 ← a_size + b_size;
        size_map ← {key2,value2} ∪ size_map;
    else
        key1 ← b_head;
        value1 ← a_head;
        father_map ← {key1,value1} ∪ father_map;
        key2 ← a_head;
        value2 ← a_size + b_size;
        size_map ← {key2,value2} ∪ size_map;
    end
```

提取测试代码主要特征信息的方式与待测方法类似，包括类变量和各个方法。为方便后续测试代码自动修复工作，对测试代码中标记有“@Before”注解的初始化方法和标记有“@Test”注解的测试任务执行方法进行特征解析时，将对方法中语句块进行拆分后存储，主要包括以下几类语句块。

(1) 包含方法调用元素 isMethodCallExpr() 的语句；

(2) 包含变量声明元素 isVariableDeclarationExpr() 的语句；

(3) 包含变量赋值元素 isAssignExpr() 的语句；

(4) 条件判断语句块 isIfStmt()；

(5) for 循环语句块 isForStmt() 和 isForEachStmt() 等。

将解析后的语句块根据代码中实际出现次序进行标记，即按 0，1，2，……开始依次标记语句出现次序，以次序为键（key）、语句块为值（value）组装成键值对，按代码中的原始顺序转换成 JSON 字符串进行存储。所有测试代码的特征信息将被组装成测试用例代码模型 testCaseModel，赋予其唯一标识 testCaseId 后存储至语料库中。然后根据待测方法及其测试用例代码的对应关系，将该待测方法的唯一标识 methodCoreId 赋值给语料库中对应测试用例代码模型的记录。

如上所述，解析待测方法及其测试用例代码中有很多共用方法。

**Algorithm 5:** 获取被调用方法的输入参数类型列表-get_mc_param_types()

```
Data: mc_parameters,global_variables,method_parameters,local_variables
Result: mc_parameter_types
1  for parameter in mc_parameters do
2      if java_parser_can_resolve_type(parameter) then
3          type ← java_parse_resolve_type(parameter);
4          type ← processing(type);
5      else if parameter in global_variables then
6          type ← get_type(parameter,global_variables;
7      else if parameter in method_parameters then
8          type ← get_type(parameter,method_parameters);
9      else if parameter in local_variables then
10         type ← get_type(parameter,local_variables);
11     mc_parameter_types ← {type} ∪ mc_parameter_types;
12 end
13 return mc_parameter_types;
```

获取被调用方法的输入参数类型列表-get_mc_param_types() 如算法 5 所示。它的输入为被调用方法的参数 mc_parameters、类变量列表 global_variables、所在方法输入参数列表 method_parameters 以及所在方法局部变量列表 local_variables。首先由 JavaParser 解析出参数类型（第 3 行），然后进行预处理，去除冗余信息（第 4 行），如果参数类型识别为“java.util.List<Integer>”，去除冗余信息后将保留“List<Integer>”。如果无法用 JavaParser 解析参数类型，则分别通过判断该参数是否存在于 global_variables、method_parameters、local_variables，如果存在，

则获得参数的类型。将所有参数的类型组装成被调用方法的输入参数类型列表 mc_parameter_types 并返回。

**Algorithm 6:** 是否为相同的方法声明-is_same_method()

```
   Data: method1, method2
   Result: method
1  flag ← false;
2  name1 ← get_name(method1);
3  name2 ← get_name(method2);
4  return_type1 = ← get_return_type(method1);
5  return_type2 = ← get_return_type(method2);
6  type_list1 ← get_type_list(method1);
7  type_list2 ← get_type_list(method2);
8  if name1 = name2 and return_type1 = return_type2 and size(type_list1) =
     size(type_list2) then
9      is_same ← true;
10     length ← size(get_type_list(method1));
11     for i ← 0:length do
12         type1 ← get(i,type_list1);
13         type2 ← get(i,type_list2);
14         if type1 != type2 then
15             is_same ← false;
16             break;
17     end
18     flag ← is_same;
19 return flag;
```

判断是否为相同的方法声明-is_same_method() 如算法 6 所示。它的输入为两个具有相同方法名的方法 $method_1$ 和 $method_2$。分别获取两个方法的方法名、返回类型和输入参数类型列表（第 2-7 行）。如果两个方法名、返回类型和参数类型列表长度均相等，则从 0 开始依次比较输入参数类型列表中的每个参数类型，如果存在一对参数类型不相等，则判断为两个方法不相同，将 false 赋值给变量 is_same，结束循环（第 15-16 行）。最后将 is_same 赋值给判断两个方法是否相同的布尔变量 flag 并返回（第 18-19 行）。

**Algorithm 7:** 获取被调用方法的方法声明-match_method_call()

```
   Data: method_call,param_types,method_list
   Result: method
1  flag ← true;
2  for method in method_list do
3      mc_name ← get_name(method_call);
4      method_name ← get_name(method);
5      method_param_types ← get_mc_param_types(method);
6      if mc_name = method_name and size(param_types) =
         size(method_param_types) then
7          length ← size(param_types);
8          for i ← 0:length do
9              type1 ← get_type(i,param_types);
10             type2 ← get_type(i,method_param_types);
11             if type1 != type2 then
12                 flag ← false;
13                 break;
14         end
15         if flag then
16             return method;
17 end
18 return null;
```

获取被调用方法的方法声明-match_method_call() 如算法 7 所示。它的输入为被调用方法 method_call 及其输入参数列表 param_types，和类声明方法列表 method_list。遍历 method_list 中的类声明方法 method，调用算法5获取类声明方法的参数列表 method_param_types（第 5 行），如果被调用方法名 mc_name 和声明方法名 method_name 相等，且被调用方法输入参数类型列表长度 size(param_types) 和类声明方法输入参数类型列表长度 size(method_param_types) 也相等，则依次序逐个对比 param_types 和 method_param_types 中的各个参数是否相同，如果存在一个参数类型不相等则结束循环并返回 null，如果都相等则返回该类声明方法 method（第 6-16 行）。

## 3.3 测试代码推荐

**Algorithm 8:** 改进后的测试代码推荐

**Data:** $name^c$,$name^m$,$types^{param}$,$type^{return}$,$code$

**Result:** target_test_code

```
types^param ← processing(types^param);
type^return ← processing(type^return);
result ← match^strict(name^c,name^m,types^param,type^return);
if size(result) = 0 then
    return ← match^spelling(name^c,name^m,types^param,type^return);
    if size(result) = 0 then
        result ← match^similarity(name^c,name^m,types^param,type^return);
        result ← result ∪ match^wildcard(name^c,name^m,types^param,type^return);
    end
end
code_similarity_map;
for method in result do
    code^r ← get_code(method);
    similarity ← cosine_similarity(code,code^r);
    key ← method;
    value ← similarity;
    code_similarity_map ← {key,value} ∪ code_similarity_map;
end
similarity_max ← 0.0;
target_method;
for map in code_similarity_map do
    similarity^r ← get_value(map);
    if similarity^r > similarity_max then
        similarity_max ← similarity^r;
        target_method ← method^r;
    end
end
method_core_id ← get_method_core_id(target_method);
target_test_code ←
  get_test_code_from_corpus_by_method_core_id(method_core_id);
return target_test_code;
```

测试代码推荐主要对本人先前工作 HomoTR[16] 进行改进。新设计的测试代码推荐算法 8 与 HomoTR 主要流程类似。该算法的输入参数为外部待测方法的主要特征，包括 $name^c$，$name^m$，$types^{param}$，$type^{return}$ 和 *code*，其中 $name^c$ 表示类名，$name^m$ 表示方法名，$types^{param}$ 表示输入参数类型列表，$type^{return}$ 表示返回类型，*code* 表示待测方法代码。

在提取了语料库中所有待测方法后，将外部待测方法与语料库中所有待测方法进行一一匹配，寻找符合要求的候选待测方法集合 result。首先需要将输入参数类型列表 $types^{param}$ 和返回类型 $type^{return}$ 进行预处理（第 1-2 行），如将“ArrayList<Integer>”修改为“List<Integer>”，这是因为语料库中所有类型均被预处理过。

接着进行严格匹配 $match^{strict}$（第 3 行）。如果没有匹配到待测方法则进行基于拼写校正的匹配 $match^{spelling}$（第 5 行）。如果还没有匹配到则依次进行基于字符串相似度的匹配 $match^{similarity}$（第 7 行）和基于通配符的匹配 $match^{wildcard}$（第 8 行），完成候选待测方法 result 的匹配。

接着将匹配结果中的待测方法代码 $code^r$ 与外部待测方法代码 *code* 进行余弦相似度度量（第 14 行），获得一个由待测方法和余弦相似度组成的键值对映射集合 code_similarity_map（第 17 行），在 code_similarity_map 中找到余弦相似度最高的键值对，将键值对中的待测方法作为最终匹配到的目标待测方法 target_method，获取其唯一标识 method_core_id（第 28 行），根据 method_core_id 从语料库中获取对应的测试用例代码（第 29 行）作为测试代码推荐结果返回。

对于本文中所述测试代码推荐方法中的严格匹配、基于拼写校正的匹配、基于字符串相似性的匹配和基于通配符的匹配四种匹配算法，它们的输入均为外部待测方法的类名 $name^c$，方法名 $name^m$，输入参数类型列表 $types^{param}$ 和返回类型 $type^{return}$，以及语料库中所有待测方法 *method_list*。相比于 HomoTR，新的待测算法匹配均增加的对类名的识别，以及对输入参数类型列表和方法名的识别做了修改。

四种匹配算法主要逻辑如算法 9 所示。在遍历语料库中所有待测方法的过程中，如果与外部待测方法的返回类型匹配（match_return_type()）结果为 false（第 4-6 行），或是输入参数类型列表匹配（match_param_types()）结果为 false（第 7-9 行），或是类名匹配（match_class_name()）结果为 false（第 10-12 行），或是方法名匹配（match_method_name()）结果为 false（第 13-15 行），均结束当前迭代，直接进入下一次迭代。如果四次匹配结果均为 true，则将这次迭代中的语料库中待测方法添加到结果集中，数量达到 10 个时结束迭代（第 16-19 行）。需要注意的是，在严格匹配阶段，对返回类型和输入参数类型列表的匹配是判断它

们是否完全相同，但是在其他三个匹配阶段进行了预处理，目的是减少遗漏有价值的语料库中待测方法。

**Algorithm 9:** 待测方法匹配算法模板

```
Data: name^c,name^m,types^param,type^return,method_list
Result: target_method_set
target_method_set;
count ← 0;
for method in method_list do
    if match_return_type(type^return,get_return_type(method) then
        continue;
    end
    if !match_param_types(types^param,get_param_types(method)) then
        continue;
    end
    if !match_class_name(name^c,get_class_name(method)) then
        continue;
    end
    if !match_method_name(name^m,get_method_name(method)) then
        continue;
    end
    if count<10 then
        target_method_set ← {method} ∪ target_method_set;
        count += 1;
    end
end
return target_method_set;
```

在基于拼写校正的匹配、基于字符串相似性的匹配和基于通配符的匹配对返回类型和输入参数类型列表的预处理均基于潜在参数类型转换列表，如表 3.1 所示。本文认为，int[] 整型数组类型和 List<Integer> 整型列表类型均可表示一组有序的整型变量排列，因此两种类型的变量很可能在实际作用上是等价的。以此类推，对于整型变量的二维数组 int[][]，与二维列表 List<List<Integer> > 的实际作用也是等价的。同时，由于 Java 语法中同时保留了基础数据类型（如 int，long，double，char 等）和继承了 Number 类型的数据类型类（如 Integer，Long，Double，Char 等）作用相似，且在实际使用中由于 Java 语言存在装箱拆箱机制，经常出现

int 与 Integer 类型的转换，因此可以认为，如 Integer[] 类型数据与 List<Integer> 类型数据在实际作用中也可能是等价的。经过对代码数据集的考察，本文认为主要存在如表 3.1 中的 18 种潜在参数类型转换。

表 3.1: 潜在参数类型转换

| | |
|---|---|
| int[] ⟺ List<Integer> | int[][] ⟺ List<List<Integer>> |
| long[] ⟺ List<Long> | long[][] ⟺ List<List<Long>> |
| double[] ⟺ List<Double> | double[][] ⟺ List<List<Double>> |
| char[] ⟺ List<Char> | char[][] ⟺ List<List<Char>> |
| String[] ⟺ List<String> | String[][] ⟺ List<List<String>> |
| Integer[] ⟺ List<Integer> | Integer[][] ⟺ List<List<Integer>> |
| Long[] ⟺ List<Long> | Long[][] ⟺ List<List<Long>> |
| Double[] ⟺ List<Double> | Double[][] ⟺ List<List<Double>> |
| Char[] ⟺ List<Char> | Char[][] ⟺ List<List<Char>> |

由于考虑了潜在参数类型转换，在新的测试代码推荐算法中基于拼写校正的匹配、基于字符串相似性的匹配以及基于通配符的匹配阶段，需要对输入参数类型列表和返回类型进行预处理。

例如，在进行返回类型匹配 match_return_type() 时，若外部待测方法 $m_1$ 的返回类型为 int[]，则将其先暂时替换为 List<Integer>，若语料库中与其功能相似的待测方法 $m_2$ 的返回类型为 List<Integer>，则 match_return_type() 执行结果为 true。同样的，对于输入参数类型列表中的每个类型均依据表 3.1 进行处理后，再进行参数类型列表匹配 match_param_types()。

新的测试代码推荐算法改进了 HomoTR 中基于拼写校正的匹配。HomoTR 的基于拼写校正的匹配，主要依据最经典的 Levenshtein 字符串编辑距离进行衡量。以编辑距离为 2 作为限制，对于两个字符串，一个进行拼写校正后是否能成为另一个字符串时，首先对一个方法名字符串分别进行新增、修改、删除一个字符的处理，将结果存入拼写校正结果集，然后对结果集中每个校正后的字符串复制出来，再次分别进行新增、修改、删除一个字符的处理，将结果同样存入拼写校正结果集。最后判断另一个字符串是否与拼写校正结果集中某个字符串相同。这种进行在线处理然后进行匹配的方式十分耗时和低效。

在新的基于拼写校正的匹配中使用了对称删除拼写校正算法，它首先构造了对目标字符串删除 2 个字符后的结果集合，如代码 3.1 所示。generateWordDeleteOne() 方法表示对字符串 word 进行删除一个字符的处理，然后将所有处理

结果汇总到集合 wordSetDeleteOne 中（第 3-6 行）。例如，对于字符串 simple，进行删除一个字符后的结果集合为 imple，smple，siple，simle，simpe，simpl。而进行两个字符删除处理的方法 generateWordDeleteTwo() 方法，则是对 wordSetDeleteOne 中的每一个字符串再次进行删除一个字符的处理（第 13 行），然后将所有处理结果收集起来作为结果集合（第 14 行）。

本文基于对称删除拼写校正算法，对语料库中所有的类名和方法名进行了删除两个字符的处理。将处理后的字符串作为键（key），而原字符串作为值（value），所有字符串的键值对将组装成一个基于键值对列表的拼写校正字典，离线存储于 JSON 文件中。

Listing 3.1: 基于对称删除拼写校正算法的字典构造

```
1  public Set<String> generateWordDeleteOne(String word) {
2      Set<String> wordSetDeleteOne = new HashSet<>();
3      for (int i = 0; i < word.length(); i++) {
4          String wordDeleteOne = word.substring(0, i) + word.substring(i + 1);
5          wordSetDeleteOne.add(wordDeleteOne);
6      }
7      return wordSetDeleteOne;
8  }
9  public Set<String> generateWordDeleteTwo(Set<String> wordSetDeleteOne) {
10     Set<String> wordSetDeleteTwo = new HashSet<>();
11     for (String wordDeleteOne : wordSetDelOne) {
12         Set<String> subWordSetDeleteTwo = generateWordDeleteOne(wordDeleteOne);
13         wordSetDeleteTwo.addAll(subWordSetDeleteTwo);
14     }
15     return wordSetDeleteTwo;
16 }
```

由于每次进行基于拼写校正的匹配都要查询拼写校正字典的 JSON 文件，为避免多次从磁盘中读取文件的 I/O 开销以及多次加载 JSON 文件产生的 Java 虚拟机堆的空间开销，本文将对键值对字典 JSON 文件的读取方法以单例模式进行构建，如代码 3.2 所示。考虑到可能存在多线程读取拼写校正字典，故拼写校正字典的读取类被设置为基于双重校验锁的单例类。通过给拼写校正字典单例变量 spellingCorrectionDictSingleton、类名字典 cnDict 和方法名字典 mnDict 添加关键字 volatile 和 static（第 2-4 行），以及在提取该单例变量的 getNameDictSingleton() 方法内代码添加同步关键字 synchronized 并在其内外同时校验 spellingCorrectionDictSingleton 是否为空来保证多线程环境下唯一（第 10-16 行）。

Listing 3.2: 拼写校正字典读取

```
1  class SpellingCorrectionDictionarySingleton {
2      private volatile static SpellingCorrectionDictionarySingleton
           spellingCorrectionDictSingleton;
3      private volatile static Map<String, Object> cnDict;
4      private volatile static Map<String, Object> mnDict;
5      private SpellingCorrectionDictionarySingleton(){
6          cnDict = JSONObject.parseObject(getData("className Dict.json"));
7          mnDict = JSONObject.parseObject(getData("methodNameDict.json"));
8      }
9      public static SpellingCorrectionDictionarySingleton getNameDictionarySingleton(){
10         if (spellingCorrectionDictSingleton == null){
11             synchronized (SpellingCorrectionDictionarySingleton.class){
12                 if (spellingCorrectionDictSingleton == null){
13                     spellingCorrectionDictSingleton = new
                           SpellingCorrectionDictionarySingleton();
14                 }
15             }
16         }
17         return spellingCorrectionDictSingleton;
18     }
19     public boolean cnDictContainsKey(String key){ return cnDict.containsKey(key); }
20     public Object cnDictGet(String key){ return cnDict.get(key); }
21     public boolean mnDictContainsKey(String key){ return mnDict.containsKey(key); }
22     public Object mnDictGet(String key){ return mnDict.get(key); }
23 }
```

当需要进行类名和方法名的拼写校正时，可调用 getNameDictionarySingleton() 方法进行拼写校正字典单例的初始化（第 13 行），于是拼写校正字典单例类将在构造方法中读取 classNameDict.json 文件和 methodNameDict.json 文件，从而对类名字典 cnDict 和方法名字典 mnDict 进行初始化（第 6-7 行）。同时，该类也提供了对类名字典和方法名字典的 containsKey() 方法和 get() 方法供外部使用（第 19-22 行）。

对方法名进行基于拼写校正的匹配如代码 3.3 所示，首先调用拼写校正字典单例 spellingCorrectionDictionary（第 1 行），进行类名和方法名字典的初始化，然后对外部待测方法的方法名 methodName 进行删除两个字符的处理，获得字符串集合 methodNameSetDeleteTwo（第 3-4 行）。遍历 methodNameSetDeleteTwo，

如果其中某个字符串是方法名字典中某个键，则该键所对应的值就是拼写校正后的方法名 methodNameSpellingCorrection（第 6-10 行），此时代表方法名匹配成功。由于基于 hash 表字典进行查找的时间复杂度为 O(1)，因此查询性能远高于在线进行对字符串增删改 2 个字符后在进行匹配的性能。对类名进行基于拼写校正的匹配与之相似。

Listing 3.3: 对方法名进行基于拼写校正的匹配

```
1  SpellingCorrectionDictionarySingleton spellingCorrectionDictionary =
       SpellingCorrectionDictionarySingleton.getNameDictionarySingleton();
2
3  //methodName为外部待测方法的方法名
4  Set<String> methodNameSetDeleteOne = generateWordDeleteOne(methodName);
5  Set<String> methodNameSetDeleteTwo =
       generateWordDeleteTwo(classNameSetDeleteOne);
6  String methodNameSpellingCorrection = null;
7  for (String methodNameDeleteTwo : methodNameSetDeleteTwo) {
8      if (spellingCorrectionDictionary.mnDictContainsKey(methodNameDeleteTwo)) {
9          methodNameSpellingCorrection = (String)
               spellingCorrectionDictionary.mnDictGet(methodNameDeleteTwo);
10     }
11 }
```

在字符串相似性匹配阶段，新的测试代码推荐算法没有采用 HomoTR 使用的基于 Word2Vec 语义相似度的方法名匹配。本质上，使用 Word2Vec 进行字符串语义相似度度量也是将字符串表示为向量，然后对向量进行余弦相似度计算。HomoTR 通过建立一个外部 Python 服务的方式，将两个字符串传输给外部服务，外部服务经计算后得出两个字符串的语义相似度并返回给 HomoTR，这就可能因为网络通信原因降低匹配性能。

Listing 3.4: 余弦相似度度量

```
import info.debatty.java. stringsimilarity .*;

private double measureCosineDistance(String string1, String string2) {
    Cosine cosine = new Cosine(cosineKeyNum);
    double cosineSimilarity = cosine. similarity (string1 , string2 ) ;
    return cosineSimilarity ;
}
```

本文中所述的待测方法匹配算法的字符串相似性匹配中，对于类名和方法名的匹配（算法 9 中的 match_class_name() 和 match_method_name），直接将两个字符串进行余弦相似度度量，如果余弦相似度大于预设的阈值时，两个字符串被认为是相似的，从而继续进行下一步的匹配。如代码 3.4 所示，本文中余弦相似性度量基于 Thibault Debatty 的 Java 字符串相似性算法库[1]进行两个字符串的余弦相似性度量。在使用余弦相似度度量时，需要设置将字符串转换为子字符串序列时的子字符串长度 cosineKeyNum。若 cosineKeyNum 设置为 2，则字符串“simple”就将先转换为“si”、“mp”、“le”三个子字符串序列，然后再转成维度为 3*1 的向量（如 [1,0,-1]）。通过在本地进行字符串相似度度量，避免了数据进行网络传输时的耗时。

Listing 3.5: 基于通配符的匹配

```
private boolean wildcardMatch(String string1, String string2){
    if(string1.contains(string2) || string2.contains(string1)){
        return true;
    }
    return false;
}
```

而基于通配符的匹配则十分简单。原则上是通过将字符串左右添加通配符“%”然后进行匹配，但在 Java 中仅需判断两个字符串是否相互包含于对方，如代码 3.5 中所示。

需要注意的是，每次进行测试代码推荐，都需要读取整个代码语料库。如果频繁进行测试代码自动复用任务，则需要经常进行与数据库的 I/O 操作，且读取的待测方法列表会极大地占用 Java 虚拟机的堆空间，因此，多次测试代码自动复用任务必须只读取一次语料库中所有待测方法然后重复使用，即语料库中待测方法列表需要存在一个单例类中。如代码 3.6 所示。与拼写校正字典读取方式类似，语料库中待测方法列表读取类采用了基于双重校验锁的单例模式，防止待测方法单例 MUTCoreModelSingleton 在多线程环境下重复初始化（第 7-16 行）。当需要读取语料库中待测方法列表 mutCoreModelList 时，通过调用 getMutCoreModelListSingleton() 方法，初始化 MUTCoreModelSingleton，通过 mutCoreDao 从数据库中读取数据到 mutCoreModelList（第 5 行），调用 getMutCoreModelList() 方法获取该列表（第 17-19 行）。

[1]https://github.com/tdebatty/java-string-similarity

Listing 3.6: 语料库中待测方法列表读取

```
class MUTCoreModelSingleton {
    private volatile static MUTCoreModelSingleton mutCoreModelListSingleton;
    private final List<MUTCoreModel> mutCoreModelList;
    private MUTCoreModelSingleton(MUTCoreDao mutCoreDao) {
        mutCoreModelList = mutCoreDao.findByIsExperiment(0);
    }
    public static MUTCoreModelSingleton
        getMutCoreModelListSingleton(MUTCoreDao mutCoreDao) {
        if (mutCoreModelListSingleton == null) {
            synchronized (MUTCoreModelSingleton.class) {
                if (mutCoreModelListSingleton == null) {
                    mutCoreModelListSingleton = new
                        MUTCoreModelSingleton(mutCoreDao);
                }
            }
        }
        return mutCoreModelListSingleton;
    }
    public List<MUTCoreModel> getMutCoreModelList() {
        return mutCoreModelList;
    }
}
```

在完成所有的待测方法匹配后，如果没有匹配到语料库中待测方法，则结束测试代码自动复用流程，即对外部待测方法的测试代码自动复用失败。如果有匹配到语料库中待测方法，则对这些待测方法代码与外部待测方法代码进行基于余弦相似度的代码相似度度量，选取相似度最高的待测方法，获取其测试代码作为目标测试代码进行自动修复。

## 3.4 测试代码自动修复

在获得推荐的测试代码后，就将进行测试代码的自动修复工作。修复工作主要解决的是，如果将语料库中待测方法 $m_1$ 的测试代码 $tc$ 进行修改，从而将其用于测试外部待测方法 $m_2$。因此，测试代码自动修复任务的重点在于如何根据 $m_1$ 和 $m_2$ 之间主要特征的差异，修改 $tc$ 中相应位置的代码。

由上文中算法 8 可知，测试代码推荐的核心部分为依据类名 $name^{class}$、方法名 $name^{method}$、输入参数类型列表 $types^{param}$ 和返回类型 $type^{return}$ 进行待测方法

的匹配，因此外部待测方法和匹配到的语料库中待测方法之间主要特征的差异就在于 $name^{class}$、$name^{method}$、$types^{param}$ 和 $type^{return}$。由此可以得到两个待测方法的主要特征差异分类：

(1) 类名差异。即 $name^{class}_{externel} \neq name^{class}_{corpus}$。

其中，$name^{class}_{externel}$ 表示外部待测方法的类名，$name^{class}_{corpus}$ 表示匹配到的语料库中待测方法的类名。

(2) 方法名差异。即 $name^{method}_{externel} \neq name^{method}_{corpus}$。

其中，$name^{method}_{externel}$ 表示外部待测方法的方法名，$name^{method}_{corpus}$ 表示匹配到的语料库中待测方法的方法名。

(3) 返回类型差异。即 $type^{return}_{externel} \neq type^{return}_{corpus}$。

其中，$type^{return}_{externel}$ 表示外部待测方法的返回类型，$type^{return}_{corpus}$ 表示匹配到的语料库中待测方法的返回类型。

(4) 输入参数类型列表差异。即 $types^{param}_{externel} \neq types^{param}_{corpus}$。

其中，$types^{param}_{externel}$ 表示外部待测方法的输入参数类型列表，$types^{param}_{corpus}$ 表示匹配到的语料库中待测方法的输入参数类型列表。

假设外部待测方法的输入参数类型列表和匹配到的待测方法的输入参数类型列表中只有第 i 个参数类型发生改变，则可表示为 $type^{i-param}_{externel} \neq type^{i-param}_{corpus}$。

由于此参数名称不同并不会导致测试用例代码编译错误，所以在现在的测试代码推荐和此处的测试代码自动修复任务均不需要考虑参数名称差异。

根据上述两个待测方法的四类差异，以下将以代码 3.7 为例，描述测试代码自动修复方法主要完成以下四大修复任务：

**修复任务一**：基于类名差异的测试代码修复。在测试代码中，与类名有关的部分包括：测试类名、被测对象声明与初始化。在代码 3.7 中体现为：

(1) 测试类名从“SearchForARangeTest”调整为“SearchRangeTest”（第 1 行）；

(2) 被测对象声明从“SearchForARange solution”调整为“SearchRange solution”（第 2 行）；

(3) 被测对象初始化从“solution = new SearchForARange()”调整为“solution = new SearchRange()”（第 7-8 行）。

Listing 3.7: 测试代码修复示例

```
1  public class SearchForARangeTest {      //调整为SearchRangeTest
2      SearchForARange solution;           //调整为SearchRange solution;
3      int [] nums;                        //调整为List<Integer> nums;
4      int target;
5      @Before
6      public void setUp() throws Exception {
7          solution = new SearchForARange();
8          //调整为solution = new SearchRange();
9          nums = new int[]{5, 7, 7, 8, 8, 10};
10         //调整为nums = Arrays.asList(5, 7, 7, 8, 8, 10);
11         target = 8;
12     }
13     @Test
14     public void searchRange() {
15         int [] expected = {3, 4};
16         //调整为List<Integer> expected = Arrays.asList(3, 4);
17         int [] actual = solution.searchForARange(nums, target);
18         //调整为List<Integer> actual = solution.searchRange(nums, target);
19         assertArrayEquals(expected, actual);
20         //调整为assertEquals(expected, actual);
21     }
22 }
```

**修复任务二**：基于方法名差异的测试代码修复。在测试代码中，与方法名有关的部分为被测对象调用待测方法。在代码 3.7 中体现为：

(1) 对待测方法实际执行结果 actual 的赋值从“solution.searchForARange(nums, target)”调整为“solution.searchRange(nums, target)”（第 17-18 行）。

**修复任务三**：基于返回类型差异的测试代码修复。在测试代码中，与返回类型有关的部分包括，被测对象执行待测方法的预期结果 expected 的声明与初始化，实际结果 actual 的声明类型，以及 Assert 系列断言方法的选择。在代码 3.7 中体现为：

(1) expected 的声明和初始化从“int[] expected = 3, 4”被调整为“List<Integer> expected = Arrays.asList(3, 4)”（第 15-16 行）。

(2) actual 的声明从“int[] actual”调整为“List<Integer>”（第 17-18 行）。

(3) Assert 系列断言方法从“assertArrayEquals(expected, actual)”调整为“assertEquals(expected, actual)”（第 19-20 行）。

**修复任务四**：基于待测方法输入参数类型列表的测试代码修复。在测试代码中，与待测方法输入参数类型列表有关的部分为待测方法输入参数的声明与初始化。在代码 3.7 中体现为：

(1) 待测方法输入参数的类型从“int[]”调整为“List<Integer>”（第 3 行）。

(2) 待测方法输入参数的初始化从“nums = new int[]5, 7, 7, 8, 8, 10”调整为“nums = Arrays.asList(5, 7, 7, 8, 8, 10)”（第 9-10 行）。

测试代码可主要分为四大部分，分别是（1）测试类名，（2）声明测试对象和待测方法输入参数的类变量，（3）由“@Before”注解标记的完成测试环境和数据准备的 setUp() 方法，以及（4）执行测试任务、声明和初始化待测方法执行预期结果 expected 和实际结果 actual、判断 expected 和 actual 是否相等的 Assert 系列断言方法的测试方法。而上述四大测试代码修复任务被零散分布在这四大部分中，因此测试代码自动修复将根据测试代码这四大部分按顺序进行。

**Algorithm 10:** 测试代码自动修复

**Data:** $method_{externel}$,$method_{corpus}$,$TCC_{corpus}$

**Result:** $TCC_{externel}$;

1 $TCC_{externel}$;

2 $name^{class}_{externel}$, $name^{method}_{externel}$, $types^{param}_{externel}$, $type^{return}_{externel}$ ← get_features($method_{externel}$);

3 $name^{class}_{corpus}$, $name^{method}_{corpus}$, $types^{param}_{corpus}$, $type^{return}_{corpus}$ ← get_features($method_{corpus}$);

// 识别四大差异

4 $diff\_name^{class} \leftarrow name^{class}_{externel} \oplus name^{class}_{corpus}$;

5 $diff\_name^{method} \leftarrow name^{method}_{externel} \oplus types^{param}_{corpus}$;

6 length ← size($types^{param}_{externel}$);

7 $diff\_type\_list^{param}$;

8 **for** $i \leftarrow 0{:}length$ **do**

9 $\quad diff\_type^{param} \leftarrow$ get(i, $types^{param}_{externel}$) ⊕ get(i, $types^{param}_{corpus}$);

10 $\quad diff\_type\_list^{param} \leftarrow \{diff\_type^{param}\} \cup diff\_type\_list^{param}$;

11 **end**

12 $diff\_type^{return} \leftarrow type^{return}_{externel} \oplus type^{return}_{corpus}$;

// 对测试代码四大部分进行修复

13 class_variables ← adjust_param_types($diff\_type\_list^{param}$, $TCC_{corpus}$);

14 before_method ← adjust_param_types($diff\_type\_list^{param}$, $TCC_{corpus}$));

15 test_method ← adjust_return_type($diff\_type^{return}$, $TCC_{corpus}$);

16 test_method ← adjust_method_name($diff\_name^{method}$, $TCC_{corpus}$);

17 $TCC_{externel}$ ← class_variables ∪ before_method ∪ test_method;

18 $TCC_{externel}$ ← adjust_class_name($diff\_name^{class}$, $TCC_{externel}$);

19 **return** $TCC_{externel}$;

如算法 10 所示，首先分别提取外部待测方法 $method_{externel}$ 和匹配到的语料库中待测方法 $method_{corpus}$ 的四大特征，即类名、方法名、输入参数类型列表和返回类型（第 2-3 行）。然后依次识别这四大特征的差异（第 4-12 行），即类名差异 $diff\_name^{class}$、方法名差异 $diff\_name^{method}$、待测方法输入参数类型列表差异 $diff\_type\_list^{param}$ 和待测方法返回类型差异 $diff\_type^{return}$。然后从测试代码由上而下的顺序，首先依据 $diff\_type\_list^{param}$ 依次进行类变量声明调整（第 13 行）和 @Before 注解标记的初始化方法 before_method 中类变量初始化调整（第 14 行），其次对于 @Test 注解标记的测试方法 test_method 进行返回类型调

整（第 15 行）和方法名调整（第 16 行）。然后将类变量声明 class_variables、before_method、test_method 拼接起来（第 17 行），然后将测试代码中所有与类名有关的元素进行调整（第 18 行）。

对于输入参数类型列表的调整，本文中所述测试代码自动修复方法首先定义了一个参数类型转换类 ParamTypeConversion，如代码 3.8，内部属性包括来自语料库的原始类型 typeFromCorpus、参数名称 name、是否要转换 needConversion、来自外部的新类型 typeFromExternel 四个。在进行具体的测试代码自动修复前，每一个输入参数均用该类对参数进行重新包装。

Listing 3.8: 参数类型转换类

```
class ParamTypeConversion {
    String typeFromCorpus;     //来自语料库的原始类型
    String name;               //参数名称
    boolean needConversion;      //是否要转换
    String typeFromExternel;   //来自外部的新类型
}
```

当类变量声明部分的待测方法输入参数类型需要调整时，遍历所有类变量声明，对需要修改的类型进行替换。而在对“@Before”注解标记的初始化方法进行调整时，由于参数的初始化行为十分多样，因此代码修复工作更为复杂。如代码 3.7 中第 9-10 行：

```
nums = new int[]{5, 7, 7, 8, 8, 10};
//调整为nums = Arrays.asList(5, 7, 7, 8, 8, 10);
```

代码 $nums = new\ int[]\{5, 7, 7, 8, 8, 10\}$ 来自语料库，其待测方法的输入参数 nums 声明的类型为 int[]。而作为被推荐的测试代码，它需要修复的外部待测方法输入参数 nums 声明的类型为 List<Integer>（显然，根据潜在变量类型转换表 3.1，这样的修复是合理的），因此 nums 的初始化语句需要进行修改，以修复外部待测方法。即将数组中所有元素 5，7，7，8，8，10 提取出来，然后用列表 Arrays.asList() 去进行包装，则修改后的 nums 初始化语句为 $nums = Arrays.asList(5, 7, 7, 8, 8, 10)$，即 nums 的声明和初始化过程要经过 $int[] \longrightarrow List < Integer >$ 的类型转换。

同样的，若来自语料库的测试代码中，待测方法的输入参数类型为 List<Integer>，且它所修复的外部待测方法同位置的输入参数类型为 int[]，则 nums 的声明和初始化过程要经过 $List < Integer > \longrightarrow int[]$ 的类型转换。

如果外部待测方法返回类型与匹配到的语料库中待测方法返回类型需要根据潜在类型转换表 3.1 进行转换，则对于测试代码中 @Test 注解标记的测试方法，其待测方法预期执行结果 expected 的类型和初始化也要进行修复，实际执行结果 actual 仅需进行声明类型修改，如代码 3.7 中第 15-18 行所示：

```
int [] expected = {3, 4};
//调整为List<Integer> expected = Arrays.asList(3, 4);
int [] actual = solution.searchForARange(nums, target);
//调整为List<Integer> actual = solution.searchRange(nums, target);
```

预期执行结果 expected 的声明类型为 int[]，可直接替换为 List<Integer>，而初始化语句为数组 3, 4，需要将数组中元素 3，4 提取出来并包装为 Arrays.asList(3, 4)。而对于实际执行结果 actual，其声明类型需从 int[] 替换为 List<Integer>，另外，由于外部待测方法与匹配到的语料库中待测方法的方法名不同，因此 actual 的初始化过程中需要对方法名进行替换。

需要注意的是，当待测方法返回类型需要进行替换时，Assert 系列断言方法很可能也需要进行修复工作。如代码 3.7 中第 19-20 行：

```
assertArrayEquals(expected, actual);
//调整为assertEquals(expected, actual);
```

因为返回类型需要经过 $int[] \longrightarrow List < Integer >$ 的转换，所以断言方法 assertArrayEquals() 也需要替换为 assertEquals()。原因是，assertArrayEquals() 方法是专门用于比较两个数组是否相等的断言方法，它首先比较两个数组对象的引用是否相等，然后判断是否为空，再判断两个数组长度是否相等，最后比较两个数组的值。

因此，如果返回类型为数组形式时，必须使用 assertArrayEquals()，否则会因为无法比较断言方法中两个元素的数组长度（尽管它们不是数组形式）而报错。但是 Arrays.asList() 本质上是集合类型，因此无法进行数组长度是否相等的判断。同样的，如果待测方法返回类型要经过 $List < Integer > \longrightarrow int[]$ 的转换，则断言方法需要从 assertEquals() 替换为 assertArrayEquals()。

在完成类变量、@Before 标记的初始化方法和 @Test 标记的测试方法关于待测方法输入参数类型、返回类型和方法名修复后，将这三大部分组装起来，成为新的测试用例代码 $TCC_{externel}$，此时仅需将该测试代码中所有和待测类有关的部分，如测试类名、被测对象的声明和初始化以及可能存在的待测方法内部类引用，如代码 3.7 中的第 1-2 行与第 7-8 行：

```
public class SearchForARangeTest{ //调整为SearchRangeTest
    SearchForARange solution;       //调整为SearchRange solution;
    ...
    solution = new SearchForARange();
    //调整为solution = new SearchRange();
```

至此，测试代码自动修复工作全部完成，获得的最终结果为可复用测试代码 $TCC_{externel}$。

## 3.5 界面展示


图 3.2: 测试代码自动复用界面

本文中所述的测试代码自动复用技术实现了一个 Web 应用，如图 3.2 所示。在左侧输入框中输入待测方法的类名和具体代码，点击中间的“测试代码复用”按钮后，右侧输入框中可自动显示可复用测试代码。

如图 3.3 中所示，分别输入待测方法的类名 ZigZag 和待测方法的具体代码，复用的测试代码如 3.4 所示，该测试代码可直接在实际项目中测试在左侧输入框中的待测方法，也可作为测试该待测方法的测试用例代码模板，通过修改其输入参数 a 和 b，以及预期结果 expected，获得新的测试用例。

**输入**

ZigZag

```
public static String convert(String a, int b) {
        if(b == 1)
            return a;
        StringBuilder result = new StringBuilder();
        int n = b + b - 2;
        for(int i = 0; i < b; i++){
            int cur = i;
            while(cur < a.length()){
                result.append(a.charAt(cur));
                cur += n;
                if(i > 0 && i < b - 1 && (cur-i-i) < a.length()){
                    result.append(a.charAt(cur - i - i));
                }
            }
        }
        return result.toString();
    }
```

图 3.3: 待测方法输入

**输出**

```
import org.junit.*;
import java.util.*;
public class ZigZagTest{
        ZigZag solution;
        String s;
        int numRows;
        @Before
        public void setUp() throws Exception {
                solution = new ZigZag();
                s = "PAYPALISHIRING";
                numRows = 3;
        }
        @Test
        public void convert() {
                String expected = "PAHNAPLSIIGYIR";
                String actual = solution.convert(s, numRows);
                assertEquals(expected, actual);
        }

}
```

图 3.4: 复用测试代码结果

## 3.6 本章小结

本章采用了总分结构，首先介绍了本文中测试代码自动复用技术的整体架构以及工作流程，包括代码语料库自动构建，改进后的测试代码推荐以及测试代码自动修复，然后解释了几个主要名词。其次详细说明了进行测试代码复用所需要的代码语料库应该如何进行自动构建，主要是如何将待测方法进行解析和特征提取，从而进行持久化存储用于测试代码复用。接着详细介绍了测试代码推荐的原理，尤其是新的测试代码推荐算法相比于本人先前工作 HomoTR 中的测试代码推荐算法的优势。然后介绍了在测试代码自动修复阶段需要执行的四大修复任务，并用一个测试代码修复的具体示例，讲解了这四大修复任务的具体要求。最后展示了测试代码自动复用技术的 Web 应用界面和执行测试代码自动复用的实际结果。

# 第四章　实验设计与结果分析

根据第三章给出的代码语料库构建与测试代码自动复用技术，本章描述了基于新构建的代码语料库的测试代码复用实验的实施与结果分析。本章首先描述了将代码数据集进行标准化以及构建语料库的情况，然后介绍了测试代码自动复用情况并对结果进行了分析。

## 4.1　实验数据准备

本文从 github 上选取了两个 Java 项目，项目中的编程题来自 LeetCode[1]、InterviewBit[2]、LintCode[3]和 GeeksForGeeks[4]。这些编程题实现的功能往往相似，可以很好地作为验证相似功能待测方法的测试用例代码能否复用的数据集。

表 4.1: leetcodegouth 项目中待测方法

| 分类 | 数量 | 分类 | 数量 |
|---|---|---|---|
| array | 51 | hashing | 16 |
| backtracking | 20 | heap | 8 |
| binary_search | 13 | linkedlist | 13 |
| bit_manipulation | 5 | math | 19 |
| breadth_first_search | 10 | reservoir_sampling | 1 |
| depth_first_search | 20 | stack | 10 |
| design | 10 | string | 39 |
| divide_and_conquer | 4 | tree | 47 |
| dynamic_programming | 52 | two_pointers | 15 |
| greedy | 16 | | |

本文选取了 leetcodegouth[5]作为构建语料库的项目，经过第四章第 4.2 节的代码标准化处理和分析后存入语料库，语料库中共收集到 369 个待测方法的主要特征信息及其对应的 369 个测试用例代码的主要特征信息，如表格 4.1 所示，

[1]https://leetcode-cn.com

[2]https://www.interviewbit.com

[3]https://www.lintcode.com

[4]https://www.geeksforgeeks.org

[5]https://github.com/gouthampradhan/leetcode

主要包括数组、回溯、二分搜索、位运算、广度优先搜索、深度优先搜索、设计、分治、动态规划、贪心算法、哈希、堆、链表、数学、蓄水池采样、字符串、树和双指针这 19 大类方法。

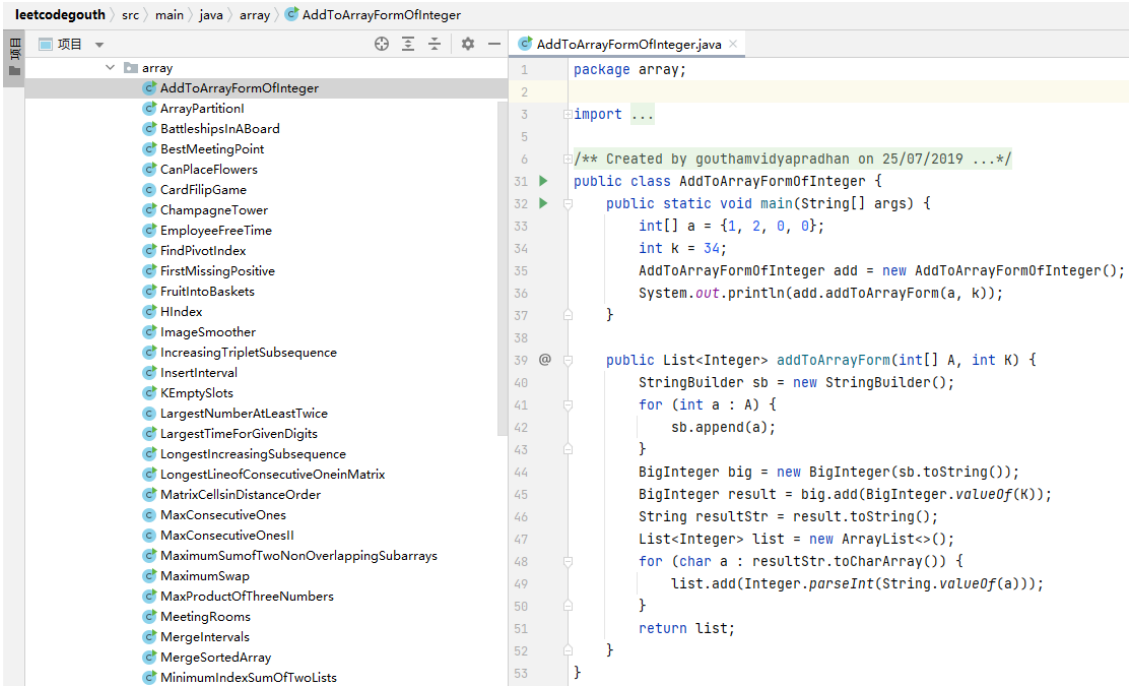


图 4.1: 代码数据集

图 4.1 展示了作为语料库的 leetcodegouth 项目的代码数据集，图中的 AddToArrayFormOfInteger 类 addToArrayForm 方法实现了数组形式的整数加法，其中数组 A 是按照从左到右的顺序表述整数数字的数组，K 是另一个整数，结果返回 A 与 K 的和，即 A=[1，2，0，0]，K=34，则 A+K 的结果为 [1，2，3，4]。

interviewnagajy[6]项目则作为实验待测项目，项目中的所有代码经过标准化处理后，共包含了 217 个待测方法，如表格 4.2 所示，所属待测方法分类与 leetcodegouth 项目相似，因此可用该项目进行相似或相同功能待测方法的测试代码自动复用任务。本人先前工作 HomoTR 也使用了 interviewnagajy 项目和 leetcodegouth 项目作为实验数据集。

## 4.2 代码数据集标准化

要进行测试代码复用，首先得拥有高质量的可复用测试代码，这对测试用例代码的规范化具有很高的要求。由于开发人员的注意力主要集中在生产代码

[6]https://github.com/nagajyothi/InterviewBit

表 4.2: interviewnagajy 项目中待测方法

| 分类 | 数量 | 分类 | 数量 |
|---|---|---|---|
| arrays | 23 | interview | 21 |
| BackTracking | 8 | Linkedin | 20 |
| BinarySearch | 13 | LinkedLists | 4 |
| BitManipulation | 6 | myMath | 8 |
| DynamicProgramming | 20 | myStrings | 22 |
| Graphs | 10 | StacksAndQueues | 8 |
| GreedyAlgorithms | 7 | Trees | 21 |
| Hashing | 11 | TwoPointers | 13 |
| HeapsAndMaps | 2 | | |

上，对测试用例代码的编写往往呈现消极态度 [6, 7]，这在原始数据集中可见一斑。因此，在构建待测方法及其测试用例代码语料库之前，首先要将测试用例代码进行标准化处理，使其成为具有规范格式的高质量测试用例代码。

Listing 4.1: 计算 x 的 n 次幂

```
public class PowXN {

    public double myPow(double x, int n) {
        if (n == 0) return 1D;
        return solve(n < 0 ? (1 / x) : x, (long) n < 0 ? ((long) n * -1) : (long) n);
    }

    public double solve(double x, long n) {
        if (n == 1) return x;
        double val = solve(x, n / 2);
        return val * val * ((n % 2) == 0 ? 1 : x);
    }

    public static void main(String[] args) throws Exception {
        System.out.println(1 / new PowXN().myPow(2.00000, -2147483648));
    }
}
```

代码 4.1 展示了一段待测方法代码示例，该方法实现了 x 的 n 次幂计算，

myPow() 方法是需要进行单元测试的目标方法，代码 4.2 是未经标准化的原始测试用例代码，代码 4.3 是经过标准化的测试用例代码。代码数据集中所有的测试用例代码全部基于 JUnit4 测试框架。测试代码标准化主要体现在以下几方面：

(1) 执行待测方法的被测对象变量以及执行测试用例时用到的所有测试输入数据和中间变量，均被声明为类变量。在标准化前的代码 4.2 中，只有执行待测方法的被测对象变量 solution 被声明为类变量。而在标准化后的代码 4.3 中，除了 solution，底数 x 和指数 n 也被声明为类变量；

Listing 4.2: 原始的 PowXN 测试用例代码

```
public class OriginalPowXNTest {
    PowXN solution;
    @Before
    public void setUp() throws Exception {
        solution = new PowXN();
    }
    @Test
    public void Test3() {
        double x = 8;
        int n = 3;
        double actual = solution.myPow(x, n);
        double expected = 512;
        assertEquals(expected, actual, 0.00001);
    }
}
```

(2) 通过“@Before”注解标记了在测试方法执行前执行的 setUp() 方法，该方法对上述类变量进行了初始化，而非只有被测对象变量 solution;

(3) 通过“@Test”注解标记了执行测试任务的具体测试方法。测试待测方法执行结果应有一个预期结果变量 expected，在具体测试方法中声明并初始化该变量。而待测方法实际执行结果则赋值给实际结果变量 actual，若待测方法返回类型为 void 类型，则将需要判断执行待测方法前后是否相等的变量赋值给 actual。在标准化前的代码 4.2 中，无论是 expected、actual，还是测试输入数据 x 和 n，均在测试方法中进行声明和初始化，而在标准化后的代码 4.3 中，测试输入数据 x 和 n 被声明为类变量，并在 setUp() 方法中得到初始化。

Listing 4.3: 标准化后的 PowXN 测试用例代码

```
public class PowXNTest {
    PowXN solution;
    double x;
    int n;
    @Before
    public void setUp() throws Exception {
        solution = new PowXN();
        x = 2.00000;
        n = 10;
    }
    @Test
    public void myPow() {
        double expected = 1024.00000;
        double actual = solution.myPow(x, n);
        assertEquals(expected, actual, 0.0);
    }
}
```

需要补充的是，在代码 4.3 中并没有展示初始化方式比较复杂的 expected 变量。如果待测方法的返回类型为一个节点，且还有其他节点连接该节点，则同样需在该测试方法中初始化其他节点并与 expected 进行连接；

(4) 使用 JUnit4 测试框架中的 Assert 系列断言方法（如 assertEquals()，assertTrue()，assertArrayEquals() 等）对执行待测方法后的预期结果和实际结果是否相同进行判断。当 JUnit4 测试框架中自带的 Assert 系列断言方法难以满足对预期结果和实际结果的判断要求时，可通过自行实现返回类型为布尔值的判断预期结果和实际结果是否相同的方法，如下列代码 4.4 所示，测试方法需要测试在移除冗余节点 t1 后，以 actual 节点作为头节点的链表是否与以 expected 节点为头节点的链表相等。由于原有待测方法没有实现判断两个链表是否相同的方法，因此在测试用例代码中自行实现 isSameList() 方法，从而判断以 head1 和 head2 为头节点的两个链表是否相等。

对于作为语料库构建项目的 leetcodegouth 项目，本文除了将该项目中已有的测试代码按照上述标准进行修改，还对所有没有对应测试用例代码的待测方法进行测试用例代码的补充，从而保证每个待测方法都有对应的测试用例代码用于进行测试代码复用。

Listing 4.4: isSameList() 方法示例

```
public boolean isSameList(ListNode head1, ListNode head2) {
    if (head1 == null & head2 == null) return true;
    if (head1 == null & head2 != null) return false;
    if (head1 != null & head2 == null) return false;
    return (head1.val == head2.val) && isSameList(head1.next, head2.next);
}
@Test
public void deleteDuplicates() {
    ListNode expected = new ListNode(1);
    expected.next = new ListNode(2);
    ListNode actual = solution.deleteDuplicates(t1);
    assertTrue(isSameList(expected, actual));    //isSameList()方法调用的地方
}
```

此外，还对 leetcodegouth 和 interviewnagajy 项目中一些数据类型及其初始化方式进行了标准化。在 Java 语言中，基于数组形式的列表是一种使用十分广泛的集合类型，也因此导致了不同代码间互不相同的初始化或使用方式。本文对所有基于数组类型的列表的初始化和使用做了标准化处理。在待测方法中，所有的方法参数类型或返回类型若为基于数组形式的列表，则统一为 “List<T>”，T 为原始代码中给出的类型，如果 T 为整型，则该参数类型为 “List<Integer>”，这样的好处是在使用测试用例进行测试时，无需因为列表类型不统一而修改代码。同样的，在所有测试用例代码中，列表类型的初始化统一使用 Arrays.asList()。

在完成代码数据集标准化后，leetcodegouth 项目将使用第三章第 3.2 节中代码语料库自动构建方法进行处理，最终得到如图 4.2 所展示的代码语料库。

图 4.2 展示了基于 leetcodegouth 项目构建的代码语料库。图中包含了类名字段 class_name、方法名字段 method_name，输入参数类型列表字段 parameter_type，方法返回类型字段 return_type，此外，图中未包含进去的语料库中重要字段还包含函数代码字段 method_code，是否是用于测试代码复用实验的字段 is_experiment 等。其中，parameter_type 字段的格式为 JSON 字符串，如 AddToArrayFormOfInteger 类 addToArrayForm 方法的输入参数类型列表为 {”0”: ”int[]”, ”1”: ”int”}，表示方法的第一个参数类型为整形数组 int[]，方法的第二个参数类型为整型 int。而所有 leetcodegouth 项目的待测方法主要特征信息在数据库中的 is_experiment 字段被赋值为 0，表示该记录作为语料库，外部待测方法通过与语料库中隶属于 leetcodegouth 的待测方法进行匹配，匹配成功后推荐其对应的测试用例代码并进行修复工作。数据库中其余字段在此不一一叙述。

| | class_name | method_name | parameter_type | return_type |
|---|---|---|---|---|
| 1 | AddToArrayFormOfInteger | addToArrayForm | {"0":"int[]","1":"int"} | List<Integer> |
| 2 | ArrayPartitionI | arrayPairSum | {"0":"int[]"} | int |
| 3 | BattleshipsInABoard | countBattleships | {"0":"char[][]"} | int |
| 4 | BestMeetingPoint | minTotalDistance | {"0":"int[][]"} | int |
| 5 | CanPlaceFlowers | canPlaceFlowers | {"0":"int[]","1":"int"} | boolean |
| 6 | CardFilipGame | flipgame | {"0":"int[]","1":"int[]"} | int |
| 7 | ChampagneTower | champagneTower | {"0":"int","1":"int","2":"int"} | double |
| 8 | EmployeeFreeTime | employeeFreeTime | {"0":"List<List<Interval>>"} | List<Interval> |
| 9 | FindPivotIndex | pivotIndex | {"0":"int[]"} | int |
| 10 | FirstMissingPositive | firstMissingPositive | {"0":"int[]"} | int |
| 11 | FruitIntoBaskets | totalFruit | {"0":"int[]"} | int |
| 12 | HIndex | hIndex | {"0":"int[]"} | int |
| 13 | ImageSmoother | imageSmoother | {"0":"int[][]"} | int[][] |
| 14 | IncreasingTripletSubsequence | increasingTriplet | {"0":"int[]"} | boolean |
| 15 | InsertInterval | insert | {"0":"List<Interval>","1":"Int… | List<Interval> |
| 16 | KEmptySlots | kEmptySlots | {"0":"int[]","1":"int"} | int |
| 17 | LargestNumberAtLeastTwice | dominantIndex | {"0":"int[]"} | int |
| 18 | LargestTimeForGivenDigits | largestTimeFromDigits | {"0":"int[]"} | String |
| 19 | LongestIncreasingSubsequence | findLengthOfLCIS | {"0":"int[]"} | int |
| 20 | LongestLineofConsecutiveOnei… | longestLine | {"0":"int[][]"} | int |
| 21 | MatrixCellsinDistanceOrder | allCellsDistOrder | {"0":"int","1":"int","2":"int"… | int[][] |

图 4.2: 代码语料库

此外，在测试代码推荐算法中，有一些数值需要设定：

(1) 在余弦相似度计算中，将字符串转换为以 2 个字符为界的序列，如字符串“ABCD”转换为序列“AB”、“CD”后，将“AB”赋值为 1，将“CD”赋值为 0，则字符串“ABCD”则被表示为向量 [1，0]。以 2 个字符为界的标准继承自 HomoTR。

(2) 基于余弦相似度计算两个字符串相似性的阈值（在测试代码推荐算法中的“基于字符串相似性的匹配”阶段）为 0.3。即，若两个字符串 $s_1$ 和 $s_2$ 经过（1）中所述方法表示为向量 $v_1$ 和 $v_2$ 后，经过公式 2.2 计算向量 $v_1$ 和 $v_2$ 的夹角余弦值 $c$ 作为字符串 $s_1$ 和 $s_2$ 的余弦相似度。若 $c < 0.3$，则认为 $s_1$ 和 $s_2$ 相似。字符串余弦相似度阈值设置为 0.3 的标准继承自 HomoTR。

## 4.3 实验设计与步骤

本文的实验，主要目标是验证测试代码复用是否有效，即复用的测试代码能否直接运行、有多少能直接运行，以及与本人先前工作 HomoTR[16] 作对比，在不考虑测试代码自动修复工作的情况下，本文中所述方法使用的测试代码推荐算法准确性相比 HomoTR 能有多大的提高。

实验将 interviewnagajy 项目作为待测项目。该项目中有 217 个有效的 Java 类文件，每一个类文件均实现了类名和待测方法中的代码作为实验的输入，即

外部类名和外部待测方法 $m$，经过第三章 3.3 节的测试代码推荐后可获得与外部待测方法 $m$ 功能相似的语料库中待测方法 $m'$ 后，可以进行本文中所述方法的测试代码推荐算法与 HomoTR 测试代码推荐算法的准确性进行比较。然后，将推荐的语料库中待测方法 $m'$ 在语料库中对应的测试代码进行测试代码自动修复工作，最后将修复结果对待测项目 interviewnagajy 中的外部待测方法方法 $m$ 进行黑盒测试，查看能否编译、运行测试用例后是否会出现断言错误。

在整个测试代码自动复用实验过程中，有以下问题需要回答：

**问题一**：本文中所述的跨项目测试代码自动复用技术的测试代码推荐算法推荐成功率相比 HomoTR 的测试代码推荐成功率能提高多少?

**问题二**：本文中所述的跨项目测试代码自动复用技术，其测试代码推荐算法是在哪个阶段实现了待测方法匹配的？这对在未来改进测试代码推荐算法有何帮助?

**问题三**：在测试代码推荐算法的基于字符串相似性的匹配阶段，阈值的设定对待测方法匹配成功率有何影响?

**问题四**：如果复用的测试用例代码在运行时出现编译错误，其原因是什么?

**问题五**：如果复用的测试用例代码在执行时出现断言错误，其原因是什么?

## 4.4 实验结果分析

经过实验验证，得到了图 4.3 的结果。由上文可知，外部待测方法总共有 217 个，其中，100 个外部待测方法实现了测试代码推荐，约占总数的 46%。而剩余 117 个外部待测方法没有实现测试代码推荐，约占总数的 54%，它们没能找到与自身功能相似的语料库中待测方法，因此导致测试代码推荐失败。

此外，使用 HomoTR 的情况下，推荐成功的个数为 61 个，仅占总数的 28%。本文中所述的测试代码复用方法，其测试代码推荐算法的成功率明显高于 HomoTR 的测试代码推荐算法。鉴于本文和 HomoTR 均是面向跨项目测试代码的复用，可认为本文中所述方法的推荐算法准确性高于 HomoTR。

通过分析本文中所述方法的测试代码推荐成功、但是 HomoTR 测试代码推荐失败的外部待测方法，发现它们均是因为输入参数类型列表和返回类型不同的。由上文可知，本文中所述方法的测试代码推荐算法应用了松弛算法的理念，在基于拼写校正的匹配阶段，以及基于字符串相似度的匹配和基于通配符的匹配阶段，放弃了严格匹配阶段中坚持的输入参数类型列表和返回类型必须相同的标准，避免遗漏大量能够进行测试代码自动复用任务的待测方法。

由表 3.1 可知，如果外部待测方法中输入参数类型列表中含有 List<Integer> 类型的参数，而语料库中实际上与该方法功能相似的待测方法输入参数类型列

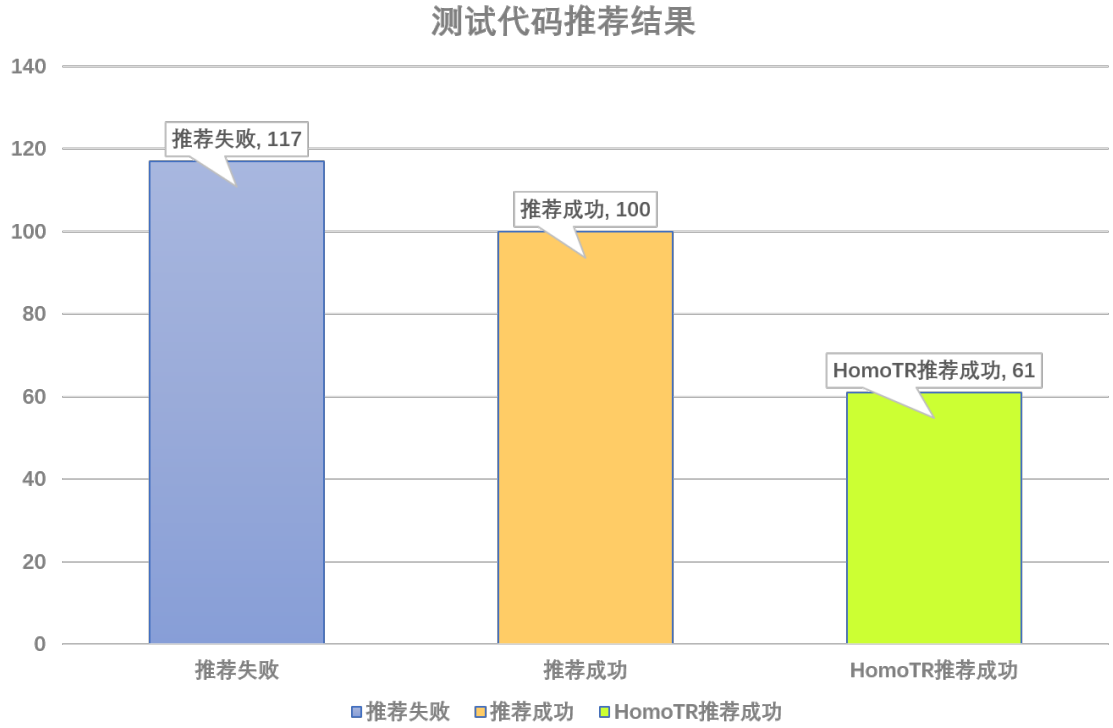


图 4.3: 测试代码推荐结果

表中，相同位置的参数类型为 int[]，则 HomoTR 无法将语料库中这个待测方法进行推荐，但本文中所述跨项目测试代码自动复用技术的测试代码推荐算法能成功进行推荐。若待测方法的返回类型也出现这种情况，同样会出现 HomoTR 推荐失败但是本文中所述方法推荐成功的情况。这是导致 HomoTR 推荐成功率低于本文中所述方法的根本原因。

**问题一的回答**：本文中所述的跨项目测试代码自动复用技术的测试代码推荐算法的推荐准确率为 46%，相比 HomoTR 的测试代码推荐准确率 28% 而言，准确率提高了 18%。

图 4.4展示的是测试代码推荐部分，针对外部待测方法，是在哪一阶段匹配到了语料库中的待测方法。由图中可见，有 5 个外部待测方法是在严格匹配阶段匹配到的，有 5 个外部待测方法是在基于拼写校正的匹配阶段匹配到的，剩余 90 个是在基于字符串相似性的匹配和基于通配符的匹配阶段匹配到的，在这 90 个中，只有 1 个是在基于字符串相似性的匹配未匹配到的情况下，在基于通配符的匹配阶段匹配到。由此可见，在整个测试代码推荐阶段，基于字符串相似性的匹配起到了最主要的作用。

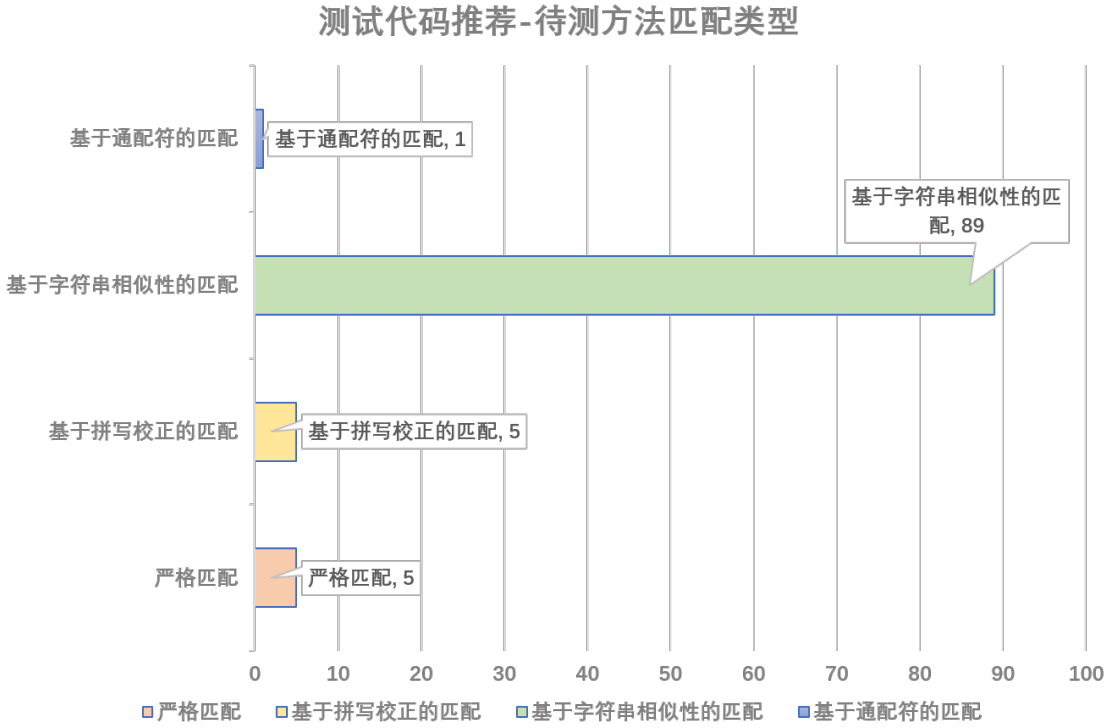


图 4.4: 测试代码推荐-待测方法匹配类型

**问题二的回答**：基于字符串相似性的匹配是外部待测方法匹配到语料库中功能相似的待测方法最多也是最重要的匹配阶段，今后提高测试代码推荐算法的准确性，重点将放在改进基于字符串相似性的匹配阶段。

图 4.5 展示了字符串相似性阈值的设置与本文中所述跨项目测试代码自动复用技术中测试代码推荐算法的待测方法匹配成功率之间的关系。由图 4.4 可知，绝大部分待测方法匹配结果均在基于字符串相似性的匹配阶段获得测试代码对应的待测方法，因此下文探讨了字符串相似性的阈值设置对待测方法匹配成功率的影响。

在图 4.5 中，横轴表示基于字符串相似性的匹配阶段设置的阈值，即两个字符串的余弦相似度高于 0.3 时被判定为相似。纵轴表示阈值从 0.3 升到 1.0 的过程中，匹配成功的数量占所有 89 个基于字符串相似性的匹配成功数量的百分比。当阈值设置为 0.3 时，能实现全部 89 个即 100% 的成功匹配，而当阈值提高到 0.96 时，这所有 89 个实验案例均无法实现待测方法匹配。

由于在严格匹配、基于拼写校正的匹配、基于通配符的匹配阶段实现待测方法匹配数量为 11，则当跨项目测试代码自动复用技术的测试代码推荐算法实

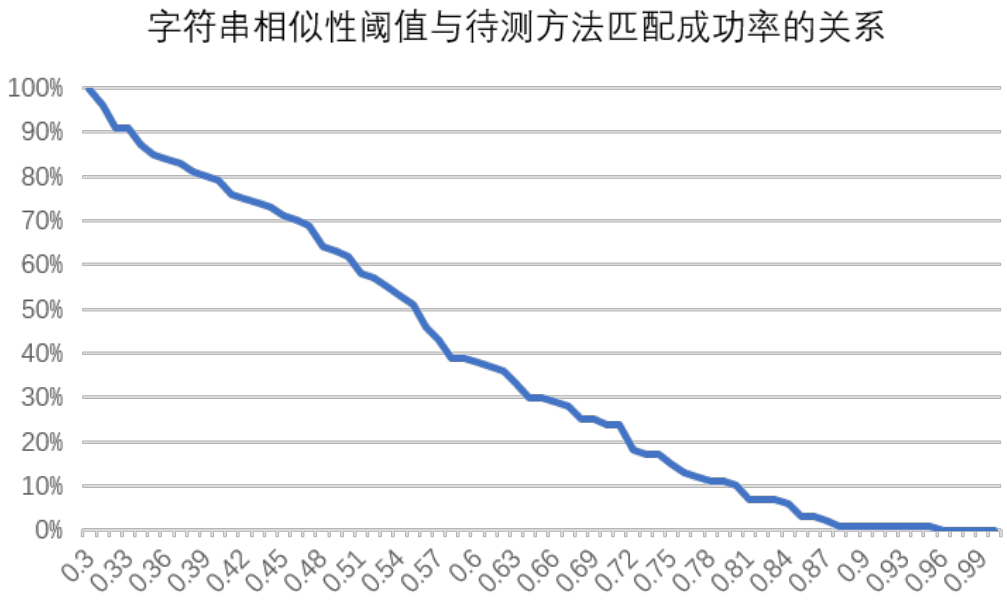


图 4.5: 字符串相似性阈值与待测方法匹配成功率的关系

现和 HomoTR 相同的待测方法匹配成功率（匹配到 61 个）时，仅需在基于字符串相似性的匹配阶段匹配 50 个，此时阈值设置约为 0.53（匹配成功 49 个）之间。需要注意的是，因此为实现比 HomoTR 更高的待测方法匹配成功率，阈值设置不能低于 0.53。需要注意的是，HomoTR 在相同阶段设置的阈值为 0.3。

考虑到阈值设置变化从 0.3 到 0.53 之间，待测方法匹配数量呈稳定下滑趋势。且当阈值设置为 0.3 时，后续实验显示有大量匹配成功的待测方法，其测试代码经过修复后能成功对外部待测方法进行测试，因此本文认为阈值设置为 0.3 仍然是合理的。

**问题三的回答**：在测试代码推荐算法的基于字符串相似性的匹配阶段，阈值设置越高，待测方法匹配成功率越低。当阈值提高到 0.53 时，与 HomoTR 的待测方法匹配成功率基本相同。由于阈值设置从 0.3 提高到 0.53 的过程中有大量有效的成功匹配的待测方法被遗漏，因此本文认为阈值仍需设置为 0.3。

图 4.6 展示了将推荐后的测试代码进行自动修复后，对外部待测方法进行测试后的情况。在这 100 个进行自动修复的测试用例代码中，有 1 个测试用例编译失败，29 个测试用例执行失败，剩下 70 个测试用例成功执行，直接复用成功率为 70%。由此可见，本文中所述方法的测试代码自动修复工作在当前待测方法项目中是行之有效的。不过，29 个测试用例执行失败并不代表本文中所述的

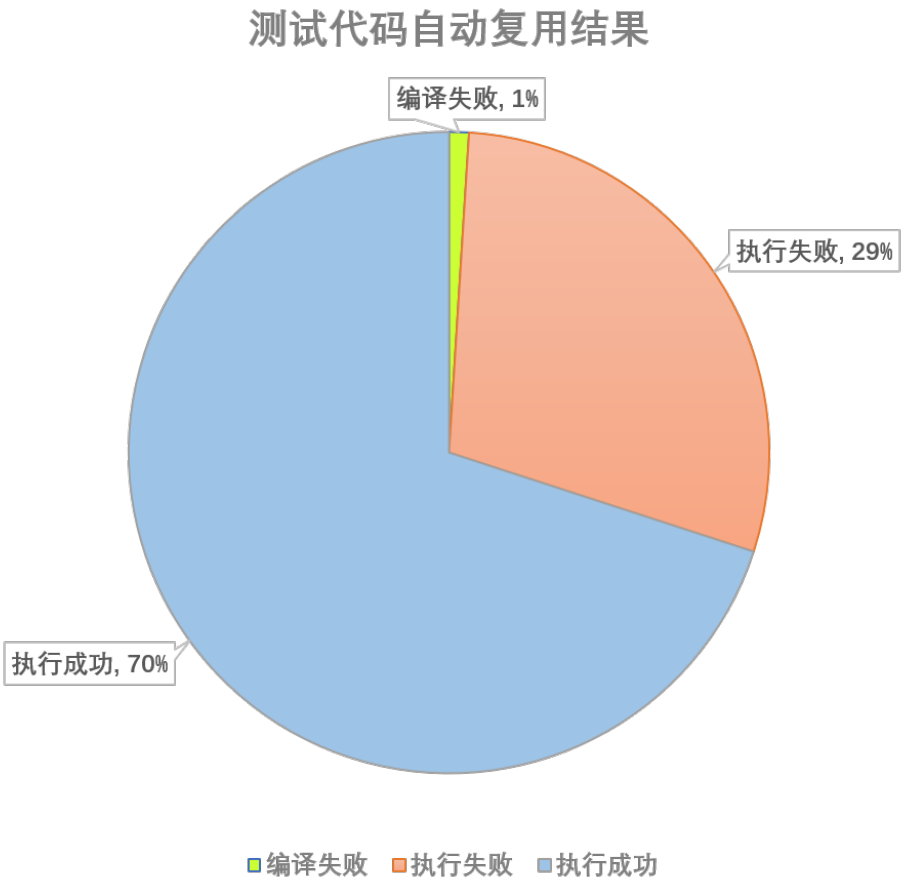


图 4.6: 测试代码复用结果

测试代码自动复用技术在应用于这些测试用例所对应的外部待测方法时没有效果，下文将详细讨论。

检查编译失败的测试用例代码后发现，两个待测方法所在的待测类均为 LRU 缓存的模拟实现。但是两个待测类是基于不同的数据结构进行实现，一个基于双向链表 DLinkList 进行实现，另一个基于节点 Node 进行实现，造成了代码实现逻辑上的不同，最终导致代码无法编译。

**问题四的回答**：造成复用的测试代码编译失败的原因为，测试代码中调用了外部待测方法所在类中不存在的方法。

图 4.7 展示了通过编译但是测试未通过的各类型测试用例的比例。由图中可知，在 29 个未通过的测试用例中，有 5 个是通过测试检测出外部待测方法中的错误，即原有的待测方法代码中存在缺陷，主要包括：

(1) 代码健壮性不足，逻辑不严谨，被特定测试用例检测出某些条件不满足；

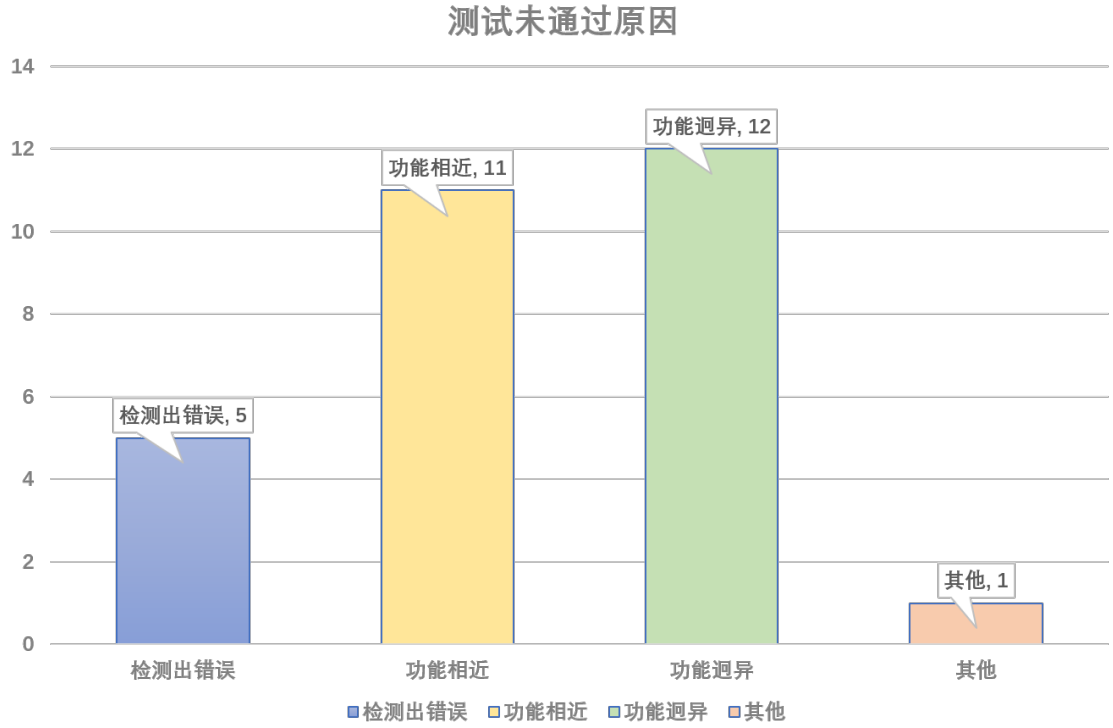


图 4.7: 测试未通过原因

(2) 方法没有正确实现预期功能（并非健壮性不足）；

(3) 定义了内部类但没有重写 hashCode() 方法和 equals() 方法，导致内部类对象无法进行正常比较。

但是，由于测试用例成功检测出了外部待测方法代码中的缺陷，本文中所述的测试代码自动复用技术针对这 5 个外部待测方法显然是有效的。

由图 4.7 可知，有 11 个测试用例出现错误的原因在于，外部待测方法和语料库中匹配到的待测方法，功能相近，但是具体条件有所不同。表 4.3 展现了一系列功能类似但是存在不同条件的外部待测方法列表。

这 11 个能够编译、功能相近但测试未通过的测试用例，待测项目 interview-nagajy 中含有与它们功能相近、但是成功通过复用的测试代码进行测试的待测方法，它们其中一部分被汇总并展示在表 4.3 中。由表 4.3 可发现，序号为 1-4、5-6、7-10、11-12、13-14 的五组待测方法，组内各方法间实现的功能极其相似，但是各自的限制条件并不相同。如不同限制条件的子数组、数字组合中数字能否重复选取、买卖股票的次数、数组中重复元素出现次数、是先序还是后序遍历

等等。这导致了即便测试用例的输入数据是相同的，其方法输出的结果也是不同的。但是测试代码复用往往给语料库中推荐的测试用例往往是同一个，这导致了测试用例的预期值与待测方法输出的实际值不同，且主要是因为预期值并非是用来测试待测方法正确实现自身功能的预期值。

分析剩下三个测试用例代码执行失败的案例。第一个是因为，外部待测方法执行结果是数组中元素的次序（从 1 开始），匹配到的语料库中待测方法执行结果是数组中元素的下标（从 0 开始），导致测试代码执行失败，及时它们实现了相同的功能。第二个是因为，外部待测方法的功能为判断是否为素数，而匹配到的语料库中待测方法的功能是统计素数个数，因此认为这两个待测方法功能相似。第三个是因为，外部待测方法的功能是找到数组中第 K 小的数，而匹配到的语料库中待测方法的功能是找到数组中第 K 大的数，因此也认为这两个待测方法功能相似。

尽管以上 11 个测试代码复用案例中测试用例代码执行失败，但这样的测试代码复用仍然是十分有价值的。在测试代码推荐领域，推荐的测试代码往往只能作为参考，但在本文中所述的测试代码复用场景下，测试代码已经完成了自动修复工作，仅仅是测试用例的预期值错误，对测试人员而言，只需要修改输入的测试数据就能就能正常对待测方法进行测试。

而在图 4.7 中通过编译、测试未通过、且功能迥异的 12 个测试用例，这些完成自动修复的测试用例代码，本质上在测试代码推荐阶段就不符合要求。在测试代码推荐阶段，测试代码推荐算法并没有在语料库中匹配到实际上功能相似或相同的待测方法，则这些待测方法所对应的测试用例代码对测试代码复用没有实际意义，因此可以理解为测试代码复用失败。

主要原因可能有以下两点：（1）语料库规模不够大，没有收集到与外部待测方法功能相同或相似的待测方法及其测试用例代码；（2）测试代码推荐算法的准确性有待提高，匹配到的语料库中功能差别很大的待测方法没能过滤。

对于图 4.7 中仅有的一个其他原因导致测试未通过的测试用例。其中一个是由 Java 语言自身特性导致的，在测试用例代码中由 Array.asList() 方法创建的列表无法直接进行元素添加操作，因为这个列表在初始化时被设置了长度，若在待测方法代码中出现向列表中添加元素的操作，会导致测试执行时出现了“不支持的操作异常”——UnsupportedOperationException。解决这个问题是十分容易的，只需将初始列表的方法从 Arrays.asList() 人工修改为 new ArrayList<>() 并一一添加元素即可。因此，这个复用的测试用例代码也可以认为是有效的。

表 4.3: 功能相似的外部待测方法列表

| 序号 | 类名 | 方法名 | 测试通过 | 方法功能 |
|---|---|---|---|---|
| 1 | MaxProductPair | maxProduct | 否 | 两个数对乘积差最大值 |
| 2 | MaxProd | maxProduct | 是 | 非空连续子数组的最大乘积 |
| 3 | MaxProduct3 | maxp3 | 否 | 长度为 3 的最大子数组乘积 |
| 4 | MaxProductSubArray | maxProduct | 是 | 非空连续子数组的最大乘积 |
| 5 | Combination | combinationSum | 是 | 和为目标数的所有数字组合(数字可重复选取) |
| 6 | Combination2 | combinationSum | 否 | 和为目标数的所有数字组合(数字不可重复选取) |
| 7 | Stocks1 | maxProfit | 否 | 一次买卖股票最大利润 |
| 8 | Stocks2 | maxProfit | 否 | 多次买卖股票最大利润 |
| 9 | Stocks3 | maxProfit | 是 | 至多两次买卖股票最大利润 |
| 10 | StocksK | maxProfit | 否 | 至多 k 次买卖股票最大利润 |
| 11 | RemoveDuplicates | removeDuplicates | 否 | 删除数组中重复元素，使其最多出现 1 次 |
| 12 | RemoveDuplicates2 | removeDuplicates | 是 | 删除数组中重复元素，使其最多出现 2 次 |
| 13 | PostOrder | postOrderTraversal | 是 | 后序遍历 |
| 14 | PreOrder | preOrderTraversal | 否 | 先序遍历 |

**问题五的回答**：复用的测试用例代码出现断言错误主要分为两种情况。假设测试用例的输入为 *input*，执行待测方法后的预期结果为 *expected*。
第一种情况为，外部待测方法 $m$ 和原始测试用例代码对应的语料库中待测方法 $m'$ 功能十分相近，但是有具体的限制条件区别，导致测试用例的预期结果 *expected* 不同，因此复用的测试用例代码只存在 *expected* 不合理的情况。
第二种则是因为外部待测方法 $m$ 和原始测试用例代码对应的语料库中待测方法 $m'$ 想要实现的功能完全不同或差别很大，导致具体的逻辑实现差距很大，因此复用的测试用例代码，其输入 *input* 和预期结果 *expected* 均不合理。
对于第一种情况，本文认为测试代码复用结果仍然有效，对于第二种情况，本文认为测试代码复用结果无效。

综上所述，对于 217 个案例，有 100 个实现了测试代码的成功推荐，测试代码推荐成功率为 46%，相比之下 HomoTR 只有 28% 能推荐成功，因此本文中所述的测试代码自动复用技术的测试代码推荐算法准确性相比 HomoTR 有显著提升。而对于 100 个成功推荐测试代码的案例，经过自动修复后，其中 87 个案例的测试代码经过修复，可以让外部待测方法成功通过测试、检测出错误或是只需经过简单的测试输入纠正就能正常使用，修复成功率为 87%。跨项目测试代码自动复用的真实成功率为 40%。

Listing 4.5: 判断是否是回文串代码（1）

```
public class Palindrome{
    public static int isPalindrome(String A) {
        if(A == null || A.length() == 1)
            return 0;

        A = A.replaceAll("[^a-zA-Z0-9]","").toLowerCase();
        int n = A.length();
        System.out.println(A);
        for(int i = 0; i < n/2; i++){
            if(A.charAt(i) != A.charAt(n-1-i))
                return 0;
        }
        return 1;
    }
}
```

**其他复用失败原因分析**：根据对 117 个在测试代码推荐阶段就失败案例进行分析，发现了按照人脑思维应该能实现测试代码自动复用，但是在实验中却没有复用成功、甚至没能进入测试代码修复阶段的案例。代码4.5表示的是对于一个给定的字符串，验证它是否是回文串，只考虑字母和数字字符，可以忽略字母的大小写。这个待测方法来自待测实验项目。我们可以看到，该方法的输入参数类型是 String，返回类型为 int。若返回 0，则表示输入的字符串 A 不是回文串，若返回 1，则表示输入的字符串 A 是回文串。

代码 4.6 展示的是来自语料库的一个待测方法，它实现的功能和代码 4.5 完全一样，唯一的不同是其方法返回类型为 boolean，即方法返回 true 时表示输入的字符串 s 是回文串，方法返回 false 表示输入的字符串 s 不是回文串。但是，待测跨项目测试代码自动复用技术无法在语料库中匹配到代码 4.6，原因是它和代码 4.5 的方法返回类型不同。

Listing 4.6: 判断是否是回文串代码（2）

```
public class ValidPalindrome {
    public boolean isPalindrome(String s) {
        if (s == null || s.isEmpty()) return true;
        s = s.toLowerCase();
        for (int i = 0, j = s.length() - 1; i < j; ) {
            char f = s.charAt(i);
            char l = s.charAt(j);
            if (!(f >= 'a' && f <= 'z') && !(f >= '0' && f <= '9')) {
                i++;
                continue;
            }
            if (!(l >= 'a' && l <= 'z') && !(l >= '0' && l <= '9')) {
                j--;
                continue;
            }
            if (f != l) return false;
            i++;
            j--;
        }
        return true;
    }
}
```

图 4.8 展示了 100 个完成测试代码自动复用（不考虑能否编译成功）的任务耗时。其中，横坐标是任务序号，从 1 到 100，纵坐标是任务耗时，单位为毫秒。由图中可知，绝大部分测试代码自动复用任务耗时在 100 毫秒到 200 毫秒之间，因此在当前语料库规模的情况下，本文中所述的测试代码自动复用技术性能是可以肯定的，小于 1 秒的复用时间并不会带来特别显著的等待。不过，若语料库规模进行了大规模扩充，且考虑了基于不稳定网络传输数据的耗时，若将该技术进行大规模落地应用，实际性能有待后续考量。

## 4.5 有效性威胁

出于对测试代码自动复用技术准确性的考量，本节对实验可能存在的偏差做出了讨论。

其一是对代码语料库规模的担忧。对于外部待测方法 $m$，只有在语料库中的确存在功能相似的待测方法 $m'$ 时，才可能正确匹配到 $m'$，然后将其对应的测

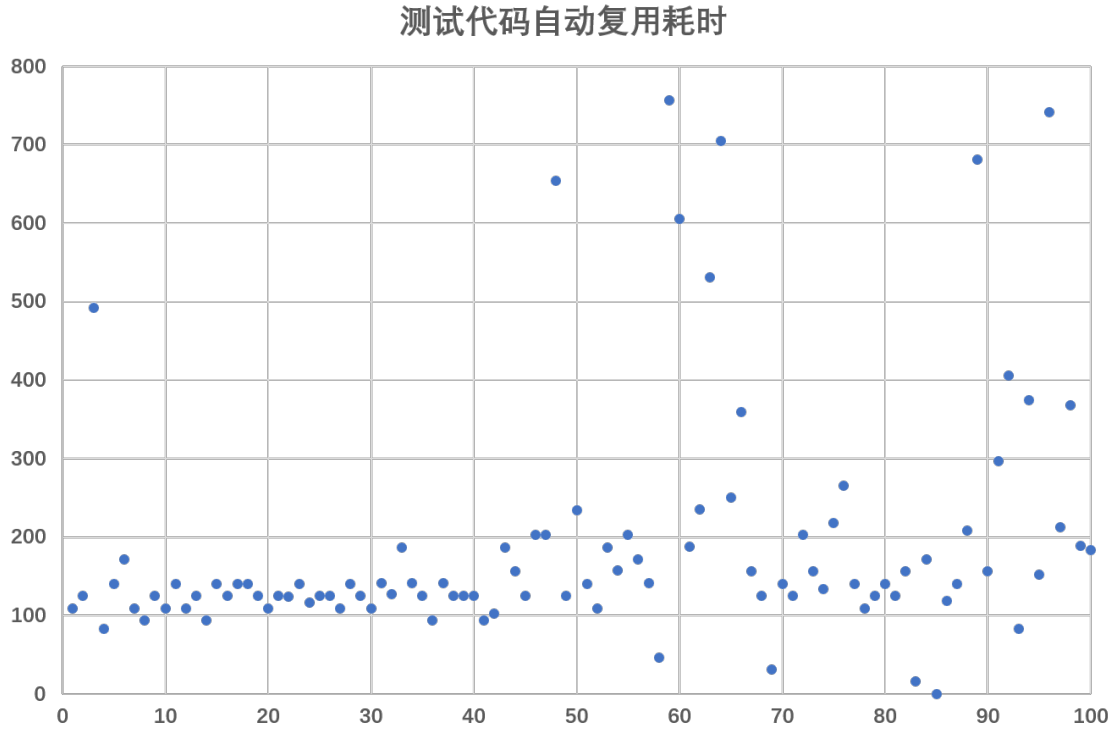


图 4.8: 测试代码自动复用耗时

试代码进行推荐和自动修复。因此，若想提升测试代码自动复用的准确性，对代码语料库规模的扩充十分必要。但是，扩充代码语料库规模后，对语料库的搜索速度会变慢。尽管当前测试代码自动复用技术已使用一些方法加快搜索速度，语料库扩充后性能仍有待考察。

其二是用余弦相似性度量两段代码相似性的有效性。即对外部待测方法代码与匹配到的语料库中待测方法代码进行余弦相似度度量，取相似度最高的待测方法的测试代码作为测试代码推荐的结果。在所有实验对象中，尽管这个方法有效的筛选掉了一部分的确不符合要求待测方法，但是在个别实验中，如表 4.3 中的第 6 个待测方法即 Combination2 类的 combinationSum() 方法 $m$，在测试代码推荐阶段，筛选掉了语料库中与其实现相同功能的待测方法，最后匹配的是与表中第 5 个待测方法即 Combination 类的 combinationSum() 方法功能相同的待测方法 $m'$。显然，$m'$ 的测试用例的测试数据无法直接复用于测试 $m$，造成 Combination2 类的 combinationSum() 方法没能通过测试用例。不过，如前所述，此次实验中复用的测试用例代码仍然是有效的，只是基于当前测试输入数据，待测方法执行的预期结果不符合实际。

## 4.6 本章小结

本章介绍了测试代码自动复用技术的实验情况。包括实验数据准备、代码数据集标准化、实验设计与步骤和实验结果分析，最后进行了有效性威胁的讨论。在实验数据准备章节，介绍了数据来源以及部分代码展示。在代码数据集标准化章节，用一个待测方法和它进行代码标准化前后两段测试用例代码的区别讲解了代码数据集标准化的具体要求。在实验设计与步骤章节简介了实验流程以及需要考虑的五个问题。在实验结果分析章节用图表展示了跨项目测试代码自动复用技术在测试代码推荐阶段的成功率以及与本人先前工作 HomoTR 相比的优势，根据测试代码自动复用流程中各阶段的具体细节，分析了参数设置对测试代码自动复用任务的影响以及复用的测试用例代码执行失败的原因，进而回答了之前需要考虑的五个问题，同时展示了测试代码自动复用技术的性能。最后讨论了测试代码自动复用技术的有效性威胁。

# 第五章　总结与展望

## 5.1　总结

进入互联网时代，软件走进千家万户。人们对各种软件需求的日益增长，也对软件开发的效率和质量提出了更高的要求。一些开发和研究人员利用软件复用的思想，提高软件的质量和开发效率，而在占据软件开发工作一半以上时间的软件测试阶段中，自然也会应用复用思想。测试复用可以让测试人员借鉴前人经验，提高测试效率和可靠性，降低软件成本，一定程度上解决测试人员经验不足的问题。但是大部分测试用例复用的案例是对测试需求和测试用例设计文档的复用。本文中所述方法和技术是对测试代码的复用，包含了用于测试代码复用的代码语料库自动构建、测试代码推荐和测试代码自动修复三部分。其中，测试代码推荐是对测试代码复用思想的一种具体实现，而测试代码自动修复是为了让复用的测试代码能够在黑盒测试中直接运行。

进行跨项目测试代码自动复用首先需要拥有高质量的代码语料库，因此需要对代码数据集进行标准化处理。本文中所用数据集仍然使用了本人先前工作 HomoTR 中使用的代码数据集，并对数据集中所有待测方法的测试用例代码按照统一的标准进行修改，同时对缺少测试用例代码的待测方法进行了测试用例代码补充的工作。

其次在代码语料库构建阶段，本文利用开源工具 JavaParser 对数据集中代码进行分析和特征提取。通过分析代码结构尤其是方法调用关系，识别出待测方法，对其进行进一步的解析，提取出它的主要特征，将这些主要特征组装成待测方法模型存入代码语料库。同时，也对待测方法的测试代码进行相似的解析工作，将其主要特征信息组装成测试代码模型存入语料库中。

然后在测试代码推荐阶段，针对 HomoTR 中存在的缺陷，本文对待测方法匹配条件做出了改进。除了严格匹配与 HomoTR 相同外，在基于拼写校正的匹配、基于字符串相似性的匹配和基于通配符的匹配阶段，都对方法输入参数类型列表和返回类型进行了松弛处理，从而避免遗漏应该在语料库中匹配到的待测方法。同时，在基于字符串相似性的匹配阶段，考虑到使用外部服务进行语义相似度匹配会提高系统不稳定性以及本质上使用 Word2Vec 进行语义相似度匹配也是基于余弦相似度，因此本文中的测试代码推荐算法直接使用余弦相似度进行字符串相似性度量。

最后在测试代码自动修复阶段，本文通过对测试代码推荐阶段获得的测试代码进行基于类名、方法名、方法输入参数类型列表和返回类型的自动修复，使得复用后的测试代码大都能正常运行，通过测试、检测出失败或是作为测试用例代码模板对待测方法进行测试用例的扩充，极大地减少了对测试代码进行复用时的人工修改成本。

在实验阶段，本文中所述跨项目测试代码自动复用技术的测试代码推荐相比 HomoTR 体现出更高的准确性。本文基于两个由不同人编写、但是内容中存在相似或相同功能的项目进行实验。其中一个项目用于构建代码语料库，另一个项目作为实验待测项目。对于所有 217 个实验案例，在测试代码推荐阶段，跨项目测试代码自动复用技术的测试代码推荐算法获得了 46% 的测试代码推荐成功率，相比 HomoTR 的 28% 具有很大的提高。在对这推荐成功的 46% 即 100 个测试用例代码进行自动修复后，有 70 个执行成功，另有 17 个虽然执行不成功但是经过核查待测方法的具体代码确认其仍然有效，即测试代码修复成功率为 87%。综上，跨项目测试代码自动复用真实成功率为 40%。

## 5.2 未来工作展望

本文实现的跨项目测试代码自动复用技术，虽然相比先前工作 HomoTR 有较大提升，但如前所述存在有效性威胁，因此仍然有可以改进的点：

第一是语料库规模可以继续扩大。跨项目测试代码自动复用技术的复用成功率，除了自动修复工作是否合理外，根本上取决于语料库中是否有和需要测试代码的外部待测方法具有相似或相同功能的待测方法，因此提高复用成功率肯定要扩充语料库规模。本文提供了极其方便的代码语料库自动构建方法，可以很方便进行语料库规模扩充。不过，提升语料库规模自然会影响测试代码推荐性能，将来视实际情况可能需要改进推荐算法提升性能。

第二是测试代码推荐的标准，现有方法主要依据待测方法的类名、方法名、输入参数类型列表、返回类型匹配到候选待测方法列表，然后对候选待测方法列表中每一个方法的代码与外部待测方法的代码进行余弦相似度度量并排序，将相似度最高的候选待测方法对应的测试代码作为将要被复用的原始测试代码进行自动修复。考虑到目前在程序理解领域有一些前沿技术正在实验中，如 Code2Vec [90]、Code2Seq [91] 等，将来这些技术若真正成熟，可用于对待测方法的功能进行理解，从而改变测试代码推荐的标准。

第三是待测方法及其测试用例代码的特征提取。目前主要通过 JavaParser 对代码进行解析，提取出关键特征如类名、方法名、方法输入参数类型列表、返回类型和关键部分代码。未来可对代码分析工作进行改进，用更合理的方式提取

出代码中的关键特征信息并以此来构建代码语料库，从而更好地支持后续的测试代码推荐过程。

第四是测试代码自动修复阶段，目前的自动修复方法并未修改原始测试数据，对于那些符合要求但是因为测试数据错误导致断言失败的测试代码仍然判断为测试代码自动复用有效。但在工业场景中，若能自动将不合理的测试数据修复为合理的测试数据，能够进一步降低人工修改成本，因此在测试数据修复等方面可以继续改进。

# 参考文献


[1] DAKA E, FRASER G. A survey on unit testing practices and problems[C] // 2014 IEEE 25th International Symposium on Software Reliability Engineering. 2014 : 201 – 211.

[2] 董威. 单元测试及测试工具的研究与应用 [J]. 微型电脑应用, 2008, 24(5) : 4.

[3] BELLER M, GOUSIOS G, PANICHELLA A, et al. When, how, and why developers (do not) test in their IDEs[C] // Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. 2015 : 179 – 190.

[4] KOCHHAR P S, THUNG F, LO D, et al. An empirical study on the adequacy of testing in open source projects[C] // 2014 21st Asia-Pacific Software Engineering Conference : Vol 1. 2014 : 215 – 222.

[5] LEE J, KANG S, LEE D. Survey on software testing practices[J]. IET software, 2012, 6(3) : 275 – 282.

[6] PHAM R, KIESLING S, LISKIN O, et al. Enablers, inhibitors, and perceptions of testing in novice software teams[C] // Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. 2014 : 30 – 40.

[7] PHAM R, STOLIAR Y, SCHNEIDER K. Automatically recommending test code examples to inexperienced developers[C] // Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. 2015 : 890 – 893.

[8] FRASER G, STAATS M, MCMINN P, et al. Does automated unit test generation really help software testers? a controlled empirical study[J]. ACM Transactions on Software Engineering and Methodology (TOSEM), 2015, 24(4) : 1 – 49.

[9] 王珊珊. 软件测试中可复用测试用例研究 [J]. 信息技术与信息化, 2015(3) : 119 – 121.

[10] FRAKES W B, NEJMEH B A. Software reuse through information retrieval[C] // ACM SIGIR Forum : Vol 21. 1986 : 30 – 36.

[11] STANDISH T A. An essay on software reuse[J]. IEEE Transactions on Software Engineering, 1984(5) : 494 – 497.

[12] GUPTA M, PRAKASH M. Possibility of reuse in software testing[C] // 6th Annual International Software Testing Conference in India. 2006.

[13] ERFANI M, KEIVANLOO I, RILLING J. Opportunities for clone detection in test case recommendation[C] // 2013 IEEE 37th Annual Computer Software and Applications Conference Workshops. 2013 : 65 – 70.

[14] JANJIC W, ATKINSON C. Utilizing software reuse experience for automated test recommendation[C] // 2013 8th International Workshop on Automation of Software Test (AST). 2013 : 100 – 106.

[15] QIAN R, ZHAO Y, MEN D, et al. Test recommendation system based on slicing coverage filtering[C] // Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. 2020 : 573 – 576.

[16] ZHU C, SUN W, LIU Q, et al. HomoTR: online test recommendation system based on homologous code matching[C] // 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE). 2020 : 1302 – 1306.

[17] 张琼宇, 杨秋辉, 张光渝, et al. 针对方法声明演化的单元测试用例修复方法 [J]. 计算机科学与探索, 2015, 9(12) : 1450 – 1458.

[18] 姚佳瑜. 软件测试中的测试用例及复用研究 [J]. 数字技术与应用, 2018(1) : 2.

[19] 芮素娟. 可复用测试用例研究 [J]. 电脑知识与技术, 2013(5X) : 3308 – 3310.

[20] 刘睿, 张彤, 丁慧. 信息安全软件测试的复用测试用例技术的研究与应用 [J]. 电子技术与软件工程, 2019.

[21] 薄慧. 软件信息安全测试的复用测试用例技术的研究与应用 [J]. 电子元器件与信息技术, 2020, 4(10) : 2.

[22] 张志国, 徐冰霖, 秦湘河. 面向复用的航天测控软件测试用例建模研究 [J]. 飞行器测控学报, 2011, 30(6) : 46 – 50.

[23] 康昊, 裴林. 数字化综合基带软件测试用例的复用及管理系统研究 [J]. 遥测遥控, 2010(6) : 47 – 51.

[24] 马贤颖, 陈青, 司倩然. 遥测软件测试用例复用技术研究及应用 [J]. 现代电子技术, 2015, 38(16) : 29 – 33.

[25] 刘末娇, 李昊, 申春妮. 基于雷达软件缺陷库的测试用例复用技术研究 [J]. 信息化研究, 2018, 44(5) : 11 – 15.

[26] 李昊, 柳溪. 基于知识图谱的雷达软件测试用例复用研究 [J]. 测控技术, 2021.

[27] 王晖, 张凯. 舰船装备软件测试用例复用技术研究 [J]. 船舶标准化与质量, 2016(4) : 39 – 44.

[28] 李昊. 雷达软件测试用例复用技术研究 [J]. 现代雷达, 2012, 34(3) : 78 – 82.

[29] 余祥, 周元璞, 王丽, et al. 指挥信息系统软件测试用例复用策略研究 [J]. 第四届中国指挥控制大会论文集, 2016.

[30] 姜蓉, 崔仕颖. 基于模型的航空机载软件测试用例复用方法研究 [J]. 工业控制计算机, 2021, 34(6) : 3.

[31] ZHANG W, ZHAO D. Reuse-oriented test case management framework[C] // 2013 International Conference on Computer Sciences and Applications. 2013 : 512 – 515.

[32] CHITTIMALLI P K, HARROLD M J. Recomputing coverage information to assist regression testing[J]. IEEE Transactions on Software Engineering, 2009, 35(4) : 452 – 469.

[33] MCGREGOR J D, MUTHIG D, YOSHIMURA K, et al. Guest editors' introduction: Successful software product line practices[J]. IEEE software, 2010, 27(3) : 16 – 21.

[34] SAINI N, SINGH S, SUMAN. Code Clones: Detection and Management[J/OL]. Procedia Computer Science, 2018, 132 : 718 – 727. https://www.sciencedirect.com/science/article/pii/S1877050918308123.

[35] BELLON S, KOSCHKE R, ANTONIOL G, et al. Comparison and evaluation of clone detection tools[J]. IEEE Transactions on software engineering, 2007, 33(9) : 577 – 591.

[36] RAGKHITWETSAGUL C, KRINKE J, CLARK D. A comparison of code similarity analysers[J]. Empirical Software Engineering, 2018, 23(4) : 2464 – 2519.

[37] DUCASSE S, RIEGER M, DEMEYER S. A language independent approach for detecting duplicated code[C] // Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360). 1999 : 109 – 118.

[38] JOHNSON J H. Identifying redundancy in source code using fingerprints[C] // Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research: software engineering-Volume 1. 1993 : 171 – 183.

[39] LI Z, LU S, MYAGMAR S, et al. CP-Miner: Finding copy-paste and related bugs in large-scale software code[J]. IEEE Transactions on software Engineering, 2006, 32(3) : 176 – 192.

[40] SAJNANI H, SAINI V, SVAJLENKO J, et al. Sourcerercc: Scaling code clone detection to big-code[C] // Proceedings of the 38th International Conference on Software Engineering. 2016 : 1157 – 1168.

[41] WISE M J. YAP3: Improved detection of similarities in computer program and other texts[C] // Proceedings of the twenty-seventh SIGCSE technical symposium on Computer science education. 1996 : 130 – 134.

[42] BAXTER I D, YAHIN A, MOURA L, et al. Clone detection using abstract syntax trees[C] // Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272). 1998 : 368 – 377.

[43] JIANG L, MISHERGHI G, SU Z, et al. Deckard: Scalable and accurate tree-based detection of code clones[C] // 29th International Conference on Software Engineering (ICSE'07). 2007 : 96 – 105.

[44] ZHANG J, WANG X, ZHANG H, et al. A novel neural source code representation based on abstract syntax tree[C] // 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). 2019 : 783 – 794.

[45] KOMONDOOR R, HORWITZ S. Using slicing to identify duplication in source code[C] // International static analysis symposium. 2001 : 40 – 56.

[46] KRINKE J. Identifying similar code with program dependence graphs[C] // Proceedings Eighth Working Conference on Reverse Engineering. 2001 : 301 – 309.

[47] ZHAO G, HUANG J. Deepsim: deep learning code functional similarity[C] // Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2018 : 141 – 151.

[48] HOLMES R, MURPHY G C. Using structural context to recommend source code examples[C] // Proceedings of the 27th international conference on Software engineering. 2005 : 117 – 125.

[49] RAHMAN M M, ROY C K. On the use of context in recommending exception handling code examples[C] // 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation. 2014 : 285 – 294.

[50] ANTUNES B, FURTADO B, GOMES P. Context-based search, recommendation and browsing in software development[G] // Context in Computing. [S.l.] : Springer, 2014 : 45 – 62.

[51] 张智轶, 陈振宇, 徐宝文, et al. 测试用例演化研究进展 [J]. Journal of Software, 2013, 24(4) : 663 – 674.

[52] MEMON A M, SOFFA M L. Regression testing of GUIs[J]. ACM SIGSOFT software engineering notes, 2003, 28(5) : 118 – 127.

[53] GRECHANIK M, XIE Q, FU C. Maintaining and evolving GUI-directed test scripts[C] // 2009 IEEE 31st International Conference on Software Engineering. 2009 : 408 – 418.

[54] GRECHANIK M, XIE Q, FU C. Experimental assessment of manual versus tool-based maintenance of GUI-directed test scripts[C] // 2009 IEEE International Conference on Software Maintenance. 2009 : 9 – 18.

[55] HARMAN M, ALSHAHWAN N. Automated session data repair for web application regression testing[C] // 2008 1st International Conference on Software Testing, Verification, and Validation. 2008 : 298 – 307.

[56] CHOUDHARY S R, ZHAO D, VERSEE H, et al. Water: Web application test repair[C] // Proceedings of the First International Workshop on End-to-End Test Script Engineering. 2011 : 24 – 29.

[57] DANIEL B, DIG D, GVERO T, et al. Reassert: a tool for repairing broken unit tests[C] // Proceedings of the 33rd International Conference on Software Engineering. 2011 : 1010 – 1012.

[58] DANIEL B, JAGANNATH V, DIG D, et al. ReAssert: Suggesting repairs for broken unit tests[C] // 2009 IEEE/ACM International Conference on Automated Software Engineering. 2009 : 433 – 444.

[59] DANIEL B, GVERO T, MARINOV D. On test repair using symbolic execution[C] // Proceedings of the 19th international symposium on Software testing and analysis. 2010 : 207 – 218.

[60] MIRZAAGHAEI M, PASTORE F, PEZZE M. Automatically repairing test cases for evolving method declarations[C] // 2010 IEEE International Conference on Software Maintenance. 2010 : 1 – 5.

[61] HAO D, LAN T, ZHANG H, et al. Is this a bug or an obsolete test?[C] // European Conference on Object-Oriented Programming. 2013 : 602 – 628.

[62] WEISER M D. Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method[D]. USA : [s.n.], 1979.

[63] KOREL B, LASKI J. Dynamic program slicing[J/OL]. Information Processing Letters, 1988, 29(3) : 155 – 163.
https://www.sciencedirect.com/science/article/pii/0020019088900543.

[64] 周汉平. Levenshtein 距离在编程题自动评阅中的应用研究 [J]. 计算机应用与软件, 2011, 28(5) : 209 – 212.

[65] TIWARI R, GOEL N. Reuse: reducing test effort[J]. ACM SIGSOFT Software Engineering Notes, 2013, 38(2) : 1 – 11.

[66] 李晓丽, 龙翔, 刘超, et al. 军用软件测试现状及对策 [J]. 装甲兵工程学院学报, 2008(5) : 5.

[67] 胡正芳. 测试用例复用技术研究 [D]. [S.l.] : 哈尔滨工程大学, .

[68] 肖良. 软件测试用例可复用性度量研究 [D]. [S.l.] : 华东理工大学, 2010.

[69] 陈强, 沈婷婷, 刘振宇. 基于分词搜索的测试用例复用 [J]. 软件产业与工程, 2015(1) : 5.

[70] LI W, DUAN M. A study on the software test case reuse model of feature oriented[C] // 2014 IEEE 3rd International Conference on Cloud Computing and Intelligence Systems. 2014 : 241 – 246.

[71] 袁松, 杨根兴, 张娟. 基于层次分析法的测试用例可复用性度量研究 [J], 2022(9).

[72] 陈平. 面向需求的嵌入式软件测试复用技术的研究 [D]. [S.l.] : 华南理工大学, 2013.

[73] 余祥, 李强, 孙胜军. 基于 TCBR 的测试用例复用方法研究 [J]. 计算机工程与设计, 2012, 33(11) : 6.

[74] 高枫, 高湘飞, 杨梦萌. 基于机器学习的软件测试经验库的构建 [J]. 长江信息通信, 2021.

[75] NOACK T, KARBE T, HELKE S. Reuse-Based Test Traceability: Automatic Linking of Test Cases and Requirements[J]. International Journal on Advances in Software, 2014, 7 : 469 – 485.

[76] 许媛媛. 基于 CBR 的测试用例复用方法研究 [J]. 软件, 2015(9) : 4.

[77] 赵中芳. 基于 CBR 的测试用例复用模型的研究与应用 [J]. 中国海洋大学, 2008.

[78] 张桂榕. 基于 CBR 的测试用例复用策略研究 [D]. [S.l.] : 西南大学, .

[79] 陈伟. 基于 LDA 模型的测试用例复用方法 [J]. 舰船电子工程, 2021, 41(2) : 5.

[80] 张娟, 童维勤, 蔡立志. 基于 Z 规格说明的可复用测试用例形式化描述 [J]. 计算机工程, 2012, 38(16) : 5.

[81] FISCHER S, LINSBAUER L, LOPEZ-HERREJON R E, et al. Enhancing clone-and-own with systematic reuse for developing software variants[C] // 2014 IEEE International Conference on Software Maintenance and Evolution. 2014 : 391 – 400.

[82] FISCHER S, LINSBAUER L, LOPEZ-HERREJON R E, et al. The ECCO tool: Extraction and composition for clone-and-own[C] // 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering : Vol 2. 2015 : 665 – 668.

[83] FISCHER S, MICHELON G K, RAMLER R, et al. Automated test reuse for highly configurable software[J]. Empirical Software Engineering, 2020, 25(6): 5295 – 5332.

[84] FISCHER S, MICHELON G K, RAMLER R R, et al. Automated Reuse of Test Cases for Highly Configurable Software Systems[J]. Software Engineering 2021, 2021.

[85] HUANG S, COHEN M B, MEMON A M. Repairing GUI test suites using a genetic algorithm[C] // 2010 Third International Conference on Software Testing, Verification and Validation. 2010: 245 – 254.

[86] DOBOLYI K, WEIMER W. Harnessing web-based application similarities to aid in regression testing[C] // 2009 20th International Symposium on Software Reliability Engineering. 2009: 71 – 80.

[87] MARíN B, VOS T, GIACHETTI G, et al. Towards testing future Web applications[C] // Proceedings of the Fifth IEEE International Conference on Research Challenges in Information Science, RCIS 2011, Gosier, Guadeloupe, France, 19-21 May, 2011. 2011.

[88] 程雪梅, 翟宇鹏. 测试用例修复的方法与工具综述 [J]. 现代计算机, 2017(02): 27 – 30+40.

[89] ANON. ISO/IEC/IEEE International Standard - Systems and software engineering – Vocabulary[J/OL]. ISO/IEC/IEEE 24765:2010(E), 2010: 1 – 418. http://dx.doi.org/10.1109/IEEESTD.2010.5733835.

[90] ALON U, ZILBERSTEIN M, LEVY O, et al. code2vec: Learning distributed representations of code[J]. Proceedings of the ACM on Programming Languages, 2019, 3(POPL): 1 – 29.

[91] ALON U, BRODY S, LEVY O, et al. code2seq: Generating sequences from structured representations of code[J]. arXiv preprint arXiv:1808.01400, 2018.

# 简历与科研成果

**基本情况** 朱晨乾，男，汉族，1997 年 3 月出生，江苏省苏州市人

**教育背景**

**2019.9 ~ 2022.6** 南京大学软件学院 硕士

**2015.9 ~ 2019.6** 南京大学软件学院 本科

**读研期间的成果（包括发表的论文及参与的专利）**

1. Chenqian Zhu, Weisong Sun, Qin Liu, Yangyang Yuan, Chunrong Fang, Yong Huang: HomoTR: Online Test Recommendation System Based on Homologous Code Matching. ASE 2020: 1302-1306

# 致　　谢

在论文最后，我要对研究生三年期间帮助过我的人表达衷心的感谢！

首先要感谢我的研究生导师陈振宇老师。陈老师带我进入了软件测试的科研领域，让我极大地丰富了软件测试的专业知识和学术科研的基础知识，让我能够发表学术论文并感受了学术会议的氛围。同时，陈老师也让我能够为软件测试教材的编写工作做出微薄的贡献。以上这些这些都是我研究生阶段一段宝贵的经历。

我也非常感谢孙伟松学长。孙伟松学长是我研究生期间科研经历的提供了最大的帮助，让我在程序分析和测试代码推荐领域进行深入的学习和研究，为我发表的论文以及这篇毕业论文提供了非常多十分有效的建议。孙伟松学长本人工作十分勤恳，乐于助人，是一位优秀的学生榜样。

我要感谢 iSE 实验室里的各位老师同学，尤其是袁阳阳学姐、房春荣老师和黄勇老师，他们对我本科毕业论文和研究生期间学术论文提供了非常多的帮助。让我了解了慕测平台的实际工作原理，这对我研究生期间的开发和科研工作有不小的帮助。

我要感谢我研究生期间的宿舍楼里的同学们，夏志龙、王越、步浩然、张晓东等，他们陪我度过了一个虽然因为疫情冲击，但是仍然不错的研究生生涯。

我要感谢实习期间认识的两位外校同学，陈朗莹和陈绍伟，他们让我在杭州经历了一段非常开心的时光！

我要感谢在蚂蚁金服实习期间的两位师兄和四位主管，师兄们对我的实习开发工作、主管们对我认识行业和思考业务问题提供了巨大帮助。

最后也是最重要的，我由衷地感谢我的爸爸妈妈爷爷奶奶等近亲。尽管与他们的争吵很多，但与他们的交流是让我能健康成长和克服学习生活中种种困难的坚实支柱和坚强动力。