南京大學

研 究 生 毕 业 论 文

（申请工程硕士学位）

论 文 题 目 面向自动程序修复的人机交互式补丁审核系统的设计与实现

作 者 姓 名 徐世诚

学科、专业名称 工程硕士（软件工程领域）

研 究 方 向 软件工程

指 导 教 师 陈振宇 教授，房春荣 助理研究员

2021 年 05 月 20 日

学　　号：**MF1932209**

论文答辩日期：**2021 年 05 月 20 日**

指 导 教 师：　　**（签字）**

The Design and Implementation of Human-machine Interactive Patch Review System For Automatic Program Repair

By

Shicheng Xu

Supervised by

Professor **Zhenyu Chen**, Research Assistant **Chunrong Fang**

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 of Engineering

Software Institute

May 2021

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：面向自动程序修复的人机交互式补丁审核系统的设计与实现

工程硕士（软件工程领域） 专业 2019 级硕士生姓名： 徐世诚

指导教师（姓名、职称）： 陈振宇 教授，房春荣 助理研究员

摘 要

软件规模和复杂性的日益增大，程序代码中的错误和缺陷层出不穷，给软件测试和程序调试带来了新的挑战，自动程序修复成了解决测试和调试成本的有效手段之一。但是当前自动程序修复技术仍存在正确率不高等问题，不正确的补丁极易成为干扰因素，不仅增大了测试和调试的成本，甚至会导致错误或缺陷无法解决。

为了提高自动程序修复工具所生成补丁的正确率，本文提出一种人机交互式补丁审核方法。该方法涉及到开发者和机器两类参与者，针对自动程序修复工具提出的候选补丁，机器的职责是通过为代码构建抽象语法树，实现对程序的代码分析，提取代码的动静态特征，并采用自动化测试工具为补丁和程序扩增测试用例；开发者的职责是通过各种交互信息认识和理解程序，根据代码特征与测试预言之间的联系，对机器生成的测试用例进行修改。通过本文的设计，在开发者与机器二者的交互下，旨在构建充分性较高的测试用例，借此对候选补丁进行审核，过滤掉不正确的补丁，完成对缺陷项目的修复。

基于对当前对程序修复领域问题的分析，以及对人机交互式补丁审核方法的讨论和设计，本文将该方法实现成一个自动化插件，集成到本地 IDE 中。系统的总体设计划分为数据分析模块、测试管理模块和人机交互模块，数据分析模块用于对原始数据的分析，构建人机交互过程中所需的信息，测试管理模块支持系统对缺陷程序的测试和数据扩增，人机交互模块负责处理开发者与机器的交互。多模块相互协作，共同完成本系统的具体实现。

最后，本文从系统测试和实验两方面对人机交互式补丁审核系统进行检查。通过系统测试，验证了系统功能测试和非功能测试是符合预期的。通过在 Defects4J 通用缺陷数据集上开展实验，数据结果表明，人机交互式补丁审核方法能够过滤掉 82.61% 的不正确补丁，使得 56.25% 的参与实验的缺陷项目最终得到了正确修复，并在用户调研中得到了参与者的积极评价。人机交互式补丁审核方法能够有效提升程序测试和调试的效率，具有一定的实用性。

关键词：程序修复，代码分析，测试生成，测试预言，人机交互

南京大学研究生毕业论文英文摘要首页用纸

THESIS: The Design and Implementation of Human-machine Interactive Patch Review System For Automatic Program Repair

SPECIALIZATION: Software Engineering

POSTGRADUATE: Shicheng Xu

MENTOR: Professor **Zhenyu Chen**, Research Assistant **Chunrong Fang**

Abstract

The increasing scale and complexity of software, as well as the endless emergence of errors and defects in program code have brought new challenges to software testing and program debugging. Automatic program repair has become one of the effective means to solve the cost of testing and debugging. However, the current automatic program repair technology still has problems such as low accuracy. Incorrect patches can easily become interference factors, which not only increase the cost of testing and debugging, but also even lead to errors or defects unresolved.

In order to improve the accuracy of patches generated by automatic repair tools, this thesis proposes a Human-machine Interactive Patch Review Method. This method involves two types of participants, the developer and the machine. For the candidate patches proposed by automatic program repair tools, the machine's responsibility is to analyze the code and extract the dynamic and static characteristics of the code through building an abstract syntax tree for the code, then use the automatic testing tool to expand the test cases for the patch and the program. The developer's responsibility is to recognize and understand the program through a variety of interactive information, and modify the test cases generated by the machine according to the relationship between the code characteristics and test oracles. With the help of the design of this thesis, our purpose is to build test cases with high sufficiency under the interaction between the developer and the machine, so as to review the candidate patches, filter out the incorrect patches, and complete the repair of defective projects.

Based on the analysis of the current problems in the field of program repair, as well as the discussion and design of Human-machine Interactive Patch Review Method, this

method is implemented as an automatic plug-in and integrated into the local IDE. The overall design of the system is divided into data analysis module, test management module and human-machine interaction module. The data analysis module is used to analyze the original data and construct the information needed in the process of human-machine interaction. The test management module supports the test and data expansion of the defective program. The human-machine interaction module is responsible for the interaction between the developer and the machine. Multiple modules cooperate with each other to complete the implementation of the system.

Finally, this thesis checks the Human-machine Interactive Patch Review System For Automatic Program Repair from the aspects of system test and experiment. Through the system test, it is verified that the system function test and non-function test are in line with the expectation. Through the experiment on the general defect dataset Defects4J, the data results show that the Human-machine Interactive Patch Review Method can filter out 82.61% of the incorrect patches, and 56.25% of the defect projects involved in the experiment can be correctly repaired. In the user research, we also got the positive evaluation from the participants. Human-machine Interactive Patch Review Method can effectively improve the efficiency of program testing and debugging, and has a certain practicality.

Keywords: Program repair, Code Analysis, Test generation, Test oracle, Human-machine interaction

目录

表 目 录 ····· **x**

图 目 录 ····· **xii**

第一章 引言 ····· **1**

1.1 选题的背景和意义 ····· 1

1.2 国内外研究现状 ····· 2

1.2.1 程序修复研究现状 ····· 2

1.2.2 补丁评估研究现状 ····· 3

1.2.3 人机交互研究现状 ····· 4

1.3 本文主要工作 ····· 5

1.4 本文组织结构 ····· 6

第二章 相关技术概述 ····· **7**

2.1 程序修复技术 ····· 7

2.2 代码分析技术 ····· 9

2.2.1 程序代码切片 ····· 9

2.2.2 代码特征提取 ····· 10

2.3 测试相关技术 ····· 11

2.3.1 测试生成 ····· 11

2.3.2 测试预言 ····· 13

2.4 系统开发技术 ····· 14

2.5 本章小结 ····· 15

第三章 系统需求分析与概要设计 ····· **17**

3.1 系统整体概述 ····· 17

3.2 系统需求分析 ····· 18

3.2.1 概念介绍 ····· 18

3.2.2 功能需求 ····· 19
3.2.3 非功能需求 ····· 20
3.2.4 系统用例设计 ····· 21
3.2.5 系统用例描述 ····· 22
3.3 系统总体设计 ····· 24
3.3.1 系统架构设计 ····· 25
3.3.2 4+1 视图模型 ····· 26
3.3.3 数据实体设计 ····· 29
3.4 系统模块划分 ····· 31
3.5 本章小结 ····· 33

第四章 系统详细设计与实现 ····· 35
4.1 数据分析模块 ····· 35
4.1.1 详细设计 ····· 35
4.1.2 核心类图 ····· 37
4.1.3 顺序图 ····· 38
4.1.4 关键代码 ····· 39
4.2 测试管理模块 ····· 39
4.2.1 详细设计 ····· 39
4.2.2 核心类图 ····· 42
4.2.3 顺序图 ····· 43
4.2.4 关键代码 ····· 43
4.3 人机交互模块 ····· 44
4.3.1 详细设计 ····· 44
4.3.2 核心类图 ····· 48
4.3.3 顺序图 ····· 48
4.3.4 关键代码 ····· 50
4.4 界面展示 ····· 50
4.5 本章小结 ····· 52

第五章 系统测试与实验分析 ····· **53**
5.1 系统测试 ····· 53
5.1.1 测试准备 ····· 53
5.1.2 功能测试 ····· 53
5.1.3 非功能测试 ····· 55
5.2 系统实验 ····· 56
5.2.1 实验一：修复模拟 ····· 56
5.2.2 实验二：用户调研 ····· 60
5.2.3 有效性威胁 ····· 62
5.3 本章小结 ····· 63

第六章 总结与展望 ····· **65**
6.1 总结 ····· 65
6.2 展望 ····· 66

参考文献 ····· **67**

致谢 ····· **77**

版权与原创性说明 ····· **79**

表 目 录

3.1 系统功能需求表 ····· 20
3.2 系统非功能需求表 ····· 20
3.3 候选补丁管理系统用例 ····· 22
3.4 交互信息查看系统用例 ····· 23
3.5 测试用例扩增系统用例 ····· 23
3.6 测试预言标注系统用例 ····· 24
3.7 缺陷程序测试系统用例 ····· 24
3.8 项目数据重置系统用例 ····· 25
3.9 项目信息持久化设计 ····· 31
3.10 补丁信息持久化设计 ····· 31
3.11 方法信息持久化设计 ····· 32

4.1 EvoSuite 参数配置 ····· 41
4.2 交互信息、补丁做出的修改以及预言标注之间的设计 ····· 47

5.1 系统测试环境表 ····· 53
5.2 候选补丁管理功能测试 ····· 54
5.3 交互信息查看功能测试 ····· 54
5.4 测试用例构造功能测试 ····· 55
5.5 缺陷项目测试功能测试 ····· 55
5.6 项目数据重置功能测试 ····· 55
5.7 系统功能测试结果 ····· 56
5.8 系统性能测试结果 ····· 56
5.9 自动程序修复工具基本信息 ····· 57
5.10 待修复项目基本信息 ····· 57
5.11 程序修复实验结果 ····· 59
5.12 对比修复实验结果 ····· 59
5.13 实验任务信息 ····· 60

5.14 实验分组信息 ····· 61
5.15 分组实验结果 ····· 61

图 目 录

2.1 自动程序修复技术的一般流程 …… 8
2.2 Eclipse 体系结构 …… 14

3.1 面向自动程序修复的人机交互式补丁审核系统的工作流程 …… 17
3.2 错误代码片段与补丁 …… 18
3.3 错误补丁与测试用例 …… 19
3.4 系统用例图 …… 21
3.5 系统架构图 …… 26
3.6 逻辑视图 …… 27
3.7 开发视图 …… 28
3.8 进程视图 …… 29
3.9 物理视图 …… 30
3.10 方法信息 JSON 格式 …… 30
3.11 系统模块划分 …… 32

4.1 静态数据分析流程图 …… 35
4.2 动态数据分析流程图 …… 36
4.3 数据分析模块核心类图 …… 37
4.4 静态数据分析顺序图 …… 38
4.5 动态数据分析顺序图 …… 39
4.6 数据分析模块关键代码 …… 40
4.7 测试生成流程图 …… 41
4.8 项目测试流程图 …… 42
4.9 测试管理模块核心类图 …… 42
4.10 测试生成顺序图 …… 43
4.11 测试管理模块关键代码 …… 43
4.12 人机交互模块与测试管理模块的联系 …… 44
4.13 人机交互模块流程图 …… 45

4.14 测试预言标注示意图 …… 46
4.15 人机交互模块核心类图 …… 48
4.16 人机交互顺序图 …… 49
4.17 补丁审核顺序图 …… 50
4.18 人机交互模块关键代码 …… 51
4.19 系统界面图 …… 52

5.1 系统满意度 …… 62

第一章　引言

1.1　选题的背景和意义

随着科学技术的进步和计算机软件的迅速发展，软件产品逐渐渗透到我们生活的各个角落，对软件质量的控制也越来越受到重视 [1] 。但是如今软件规模日益增大，软件开发流程中的各个阶段都有可能出现问题，软件缺陷（Software Defects）成了软件开发中不可避免的产物。所谓的软件缺陷，是计算机软件或程序中存在的破坏正常运行能力的错误，常常被叫做 Bug。软件缺陷会在一定条件下造成程序出错，产生有违预期的运行结果，严重的情况下甚至会造成较大的损失。为了提高软件的质量，软件测试作为软件质量控制的常用手段，已经得到了广泛的关注和应用 [2] 。

软件测试是在软件工程发展高度成熟后所形成的技术之一，它是一种实际输出与预期输出之间相互比较的过程。在软件测试理论的指导下，软件开发人员通过一定数量且符合要求的测试用例来对软件进行充分的验证，来检测并排查软件中的缺陷。然而在软件规模日益增大的背景下，人工主导软件测试来修复软件中的缺陷需要耗费的资源非常庞大，数据表明，定位缺陷和修复缺陷占了将近 50% 的软件开发成本 [3] 。为了降低修复缺陷的成本，在软件测试的基础上，软件研究人员进一步提出了自动程序修复技术（Automated Program Repair，APR），自动程序修复技术能够针对含有缺陷的软件提出可能的补丁，这些补丁可以直接放入程序中使得程序能够通过测试，或者辅助软件开发人员进行缺陷定位和修复，因而该技术有效提高了软件开发人员在进行程序调试和软件测试时的效率 [4] 。目前国内外已经展开很多关于自动程序修复的研究，同时涌现出许多自动程序修复工具 [5–8] 。

自动程序修复技术首先通过缺陷定位，找到错误在程序中的具体位置；然后使用定义好的规则对错误的程序代码进行修改，直到新的代码能够通过既定的测试用例，然后将修改过的代码作为结果输出，我们称之为候选补丁；最后对生成的候选补丁做进一步验证和评估，借助特定的手段找到最合适的补丁。自动程序修复技术以及随之推出的相关工具，凭借其自动化的优势，很大程度上为软件缺陷的修复节省了时间成本和人力成本，提高了开发人员在整个软件开发工作中的效率。近年来，自动程序修复已经吸引了大量的研究工作，但是该项技术的研究在迅速发展的同时，也面临着一系列的问题和挑战 [9] 。

其中最主要的挑战来源于，程序修复技术必须具有较高的精度才能被广泛

的使用，也就是意味着生成的补丁需要具有较高的正确性。由于程序本身的复杂性和自动程序修复技术的局限性，自动程序修复工具针对缺陷程序一般情况下会提出多个候选补丁，从而造成了补丁的质量参差不齐 [10, 11]，因此确保补丁的正确性从根本上来说是很困难的。Tao 等人 [9] 通过实验分析了自动程序修复工具生成的补丁对软件开发人员的影响，补丁质量越高，则对辅助开发人员进行调试的效率提升越显著；反之甚至会给软件开发人员造成误导，增加软件开发的成本。因此，从一组候选补丁中筛选出正确的补丁，是程序修复技术中的关键环节，这就要求我们能够有效甄别出错误的补丁，排除无用的干扰信息。针对这样的问题和挑战，补丁的验证评估也成为了当前程序修复领域的热点问题。

在此背景下，本文提出了一种交互式补丁审核的方法，并将其实现成面向自动程序修复的人机交互式补丁审核系统。该系统针对给定的缺陷程序以及自动程序修复工具生成的候选补丁，通过测试工具和开发者共同构造测试用例，对候选补丁的正确性进行评估，筛选出正确补丁，从而完成缺陷程序修复的工作。本文以自动程序修复工具对缺陷程序提出的候选补丁为基础进行数据分析，利用测试生成工具对既有的测试用例进行扩增，开发者通过交互界面对工具生成的测试用例予以预言标注，完成测试用例的构建，最后执行测试验证补丁的正确性。通过迭代的测试生成、预言标注、程序测试，不断缩小正确补丁的范围，直到筛选出正确的补丁。

1.2　国内外研究现状

1.2.1　程序修复研究现状

自动程序修复技术通常由工具对错误程序进行分析，提出能够修复错误的补丁。目前大部分自动程序修复技术的研究都集中在基于测试的修复（Test Based Repair）领域中 [12]，目标是生成能够通过全部测试用例的补丁。在基于测试的自动程序修复技术中，测试用例用于描述程序的正确行为，指导补丁的生成并保证补丁至少能在配套的测试用例上正确运行。

补丁的生成是自动程序修复技术中的重要阶段，该阶段根据获取补丁的过程，补丁生成的方法主要可以分为两类：基于搜索的方法和基于语义的方法。基于搜索的方法是通过在搜索空间中查找从而获取补丁，然后借助测试用例进行验证的过程。基于搜索的方法是自动程序修复领域最早出现的一类方法，源自于基于搜索的软件工程 [13]（Search Based Software Engineering，SBSE），目前已有大量的成果并取得了较大进展。Arcuri 等人 [14] 最先提出了基于测试的修复思路，应用协同演化算法自动生成补丁，然后通过错误程序和测试用例的多轮迭代，选出正确的补丁。Weimer 等人 [15, 16] 设计了 GenProg 算法，把源程序

转化为抽象语法树，通过对语法树的节点进行增删改等操作，从而生成新的程序作为补丁，然后结合测试用例进行验证，这在自动程序修复领域是具有开创性的工作。针对 GenProg 的计算开销和补丁生成效率等问题，Weimer 等人 [17] 又进一步提出 AE（Adaptive Equivalence）方法，通过识别语义等价的候选补丁，有效减少了需要评估的候选补丁数目，从而提升了生成补丁的效率；而 Fast 等人 [18] 则从遗传规划的优化角度提升了生成补丁的效率。Qi 等人 [19] 在补丁搜索过程中考虑了随机搜索并提出了 RSRepair 方法，并证明了能更快速的找到补丁。Debroy 等人 [20] 提出了基于变异测试的修复算法，该方法借助变异测试中的变异算子的概念来获得可能的补丁。补丁的生成本质上是对代码语句进行修改，因此代码修改操作的设定决定了自动程序修复方法的搜索空间，因此 Kim 等人 [4] 通过挖掘和总结开源项目中开发人员编写的补丁所涉及的修改操作，提出了 PAR 方法，并总结出代码修改的一般模板。

基于语义的方法是通过分析语义信息来进行程序合成的，从而达到程序修复的目的。基于语义的方法是近年来自动程序修复领域的研究热点，该类方法结合了天使调试、约束求解等概念，形成了一套综合修复方法，该方法目前而言还没有基于搜索的方法成熟，但是它的修复效果普遍优于基于搜索的方法，是未来自动程序修复研究发展的方向。Nguyen 等人 [21] 提出 Semfix 方法，该方法基于天使调试的思想，对可疑语句生成正确规约，然后基于规约借助不同的代码合成方法，合成正确的表达式。Mechtaev 等人 [22] 在此基础上提出了 DirectFix 方法，该方法将缺陷定位和补丁生成融合成了一步，提高了补丁的可读性和可理解性。他们又通过结合轻量级约束提出了 Angelix 方法 [23] ，使得基于语义的方法可以修复规模更大的程序错误。还有些方法重点关注某一类型代码语句或错误，例如 Xuan 和 DeMarco 等人 [24] 重点关注条件语句缺陷的修复而提出的 Nopol 方法；Marcote 和 Monperrus 等人 [25] 针对无穷循环这一类型的缺陷提出的 Infinitel 方法；Gopinath 等人 [26] 通过生成程序对应的控制流图，并结合程序规格说明和代码约束公式，获取所有满足约束条件的参数解，该方法能识别绝大部分赋值语句和分支条件语句中的代码错误，但是在修复过程中需要人工参与，属于半自动化程序修复方法。

1.2.2　补丁评估研究现状

自动程序修复技术在补丁生成阶段为错误程序提出了多个候选补丁作为可能的解决方案，这样的补丁虽然能通过配套测试用例，但是并不代表就是正确的修复方案 [10] 。因此针对越来越丰富的补丁生成方法，准确评价补丁的正确性、筛选出有用的补丁，成为程序修复领域中的一大研究热点。

最开始研究人员将补丁成功修复错误定义为，生成的补丁能够通过所有的测试用例。但随着自动程序修复技术的发展和成熟，研究人员发现能够通过所有测试用例的补丁并非就是正确的补丁。通过对这些问题背后的原因进行观察和总结，Qi 等人 [11] 提出了 Kali 方法，该方法不仅能够针对程序提出修复方案，还考虑评估补丁的正确性，它通过对程序做简单的功能删除，来检验测试用例的完整性。Smith 等人 [27] 将配套测试用例划分成相互正交的两部分，一部分用于补丁的生成，另一部分用于补丁的评审。Jiang 等人 [28] 结合 GenProg 方法和蜕变测试，依靠蜕变测试生成的新测试用例与已有测试用例的对比，从而进行补丁的筛选，并总结出 MT-GenProg 方法。Tao 等人 [29] 从开源社区项目中分析总结出 12 种不合格的补丁类型，用于辅助自动程序修复技术中补丁的评估工作。

补丁排名是一种有效进行补丁正确性评审的手段，通过排名列表可以清晰直观的认识补丁的有效程度。目前提出的方法中，使用较多的是使用句法和语义距离 [30, 31] ，与原始程序比较从而进行排名；还有的方法从现有的规则中学习 [32] ，来构建概率模型来对补丁进行排序。补丁分类是另一种评审补丁正确性的方法，补丁分类与补丁排名是一个既相关却不相同的问题，补丁排序问题可以通过设置适当的阈值转换为补丁分类问题。Xin 和 Reiss [33] 假设存在一个完美的预言系统来对测试结果进行分类，并生成新的测试输入来识别预言系统中存在的违规情况。有的研究人员使用启发式方法对补丁进行分类，Tan 等人 [34] 提出了应用反模式的思路，捕获落入特定静态结构中的典型不正确补丁。

目前，补丁评估的研究工作仍处在研究阶段。由于绝大部分自动程序修复技术都是基于测试的，利用测试用例来验证补丁的正确性是目前的主要手段。但是，由于测试用例本身所具有的局限性，导致通过一组测试的补丁却可能在其它测试用例上失败。因此，提高测试用例的充分性，是优化补丁评审的重要思路。

1.2.3　人机交互研究现状

自动程序修复技术主要是依靠工具对错误程序进行分析并提出候选补丁，但是工具一直以来受制于自身的缺陷，造成了补丁的难以维护、难以理解等特性，是无法避免的。这使得研究人员开始考虑让开发人员以人工介入的方法参与程序修复的过程。缺陷定位是程序修复过程中的基础环节和首要步骤，其定位结果的准确性直接影响到后续步骤 [35] 。因而借助开发人员参与自动程序修复的思路首先在缺陷定位阶段就有所研究，许多交互式调试技术利用用户反馈来进行缺陷定位。最早被软件研究人员提出的方法是算法调试树 [36] ，它首先建立一棵调试树来展示测试失败的方法，然后反复要求开发人员针对调试树中的

方法回答该方法的输入输出是否正确，再根据答案修剪调试树直到缺陷的位置被确定。Li 等人 [37] 通过利用基于频谱的缺陷定位方法和动态相关性来确定可疑的方法调用顺序，从而对算法调试树做出了改进。Gong 等人 [38] 提出了一种交互式的缺陷定位技术，通过要求开发人员将语句标记为正确或错误并更新可疑语句列表来改进 Li 等人提出的算法。Ko 和 Myers 等人 [39] 提出了 Whyline，该方法允许开发人员针对程序输出提出问题，然后根据程序运行轨迹和动态切片给出可能的解释，直到开发人员找到问题的根本原因。Lin 等人 [40] 提出的 Microbat 可以允许开发人员跟踪运行模式并利用开发人员的反馈来推荐一些可疑的位置，从而提升了 Whyline 的交互效率。Ko 等人 [41] 提出了一种理解和调试程序的方法，该方法将程序行为转化为根据代码和执行情况来提出问题的集合，开发者通过选择有关输出的问题来推测程序的行为。

最近几年出现的研究尝试让开发人员参与整个程序修复的过程。Cashin 等人 [42] 提出了 PATCHPART，该方法通过程序不变性对生成的补丁进行聚类，每类中的补丁可能全部正确或全部不正确，然后让开发人员检查每类中的每一个补丁，来评估该类中补丁的正确性。Böhme 等人 [43] 设计了 LEARN2FIX 工具，该工具要求开发人员在生成补丁之前考虑补丁的过拟合问题，过拟合问题定义为生成的补丁可以通过一部分测试用例，却无法通过其它的。Liang 等人 [44] 提出一种交互式补丁评审方法，通过回答交互问题的形式让开发人员检查补丁的正确性，并将该方法集成为 InPaFer 工具。虽然人工的参与可能导致修复效率的下降，但是随着越来越多的方法和工具的出现，也证实了在适当人为参与下，自动程序修复技术能够在一定程度上保证生成补丁的正确性。

1.3　本文主要工作

本文设计并实现了面向自动程序修复的人机交互式补丁审核系统，该系统针对给定的缺陷项目以及自动程序修复工具生成的候选补丁，通过测试工具和开发者共同构造测试用例，对候选补丁的正确性进行验证，从而完成程序修复的工作。该技术旨在结合工具和人工二者的优点，通过构造充分性更高的测试用例，从一组候选补丁中筛选出正确的补丁。本文的主要工作体现在以下几个方面：

1. 提出了一种人机交互式补丁审核的方法，针对自动程序修复技术所生成的补丁进行审核和筛选。
2. 开发了应用该技术的、具有良好界面的面向自动程序修复的人机交互式补丁审核系统，供开发者轻松浏览补丁，并通过交互完成相应的工作。

3. 验证了系统的功能，并设立对比实验，从审核结果、对开发者程序调试的辅助能力等角度评估本系统的有效性。

1.4　本文组织结构

本文组织结构如下：

第一章引言，阐述了面向自动程序修复的人机交互式补丁审核系统的项目背景及意义，介绍了国内外研究现状，本文的主要研究内容及工作，并说明本文的组织架构。

第二章相关技术概述，将介绍本文在研究和开发过程中，使用到的相关技术和工具，包括程序修复技术、代码分析技术、测试相关技术和系统开发技术，以及这些技术具体如何与本文相结合。

第三章面向自动程序修复的人机交互式补丁审核系统的系统需求分析与概要设计，首先对系统进行整体概述，然后分析本系统的功能需求和非功能需求，接下来介绍系统的总体设计，最后对系统的模块进行划分。

第四章面向自动程序修复的人机交互式补丁审核系统的系统详细设计与实现，根据模块划分依次介绍每个模块的详细设计、核心类图、顺序图和关键代码，最后展示并说明系统的运行界面。

第五章面向自动程序修复的人机交互式补丁审核系统的系统测试与实验分析，通过系统测试验证本系统的功能和非功能属性，通过两个实验评估面向自动程序修复的人机交互式补丁审核系统所运用方法的有效性并收集用户的反馈。

第六章总结与展望，对本文的研究和工作进行总结，并根据研究和开发中存在的不足指出改进的方向。

第二章　相关技术概述

2.1　程序修复技术

在过去的软件研究过程中，为了降低软件开发成本，提高软件开发人员修复程序错误的效率，自动程序修复技术吸引了大量的研究工作，并且已经取得了重大进展 [5–8] 。自动程序修复技术可以针对程序中的错误自动生成一组可能的候选补丁，这些补丁既可以添加到程序中使得程序能够通过测试用例，也可以用于辅助开发人员继续改进代码直至修复错误，因此自动程序修复技术能够有效提高开发效率，较低开发成本。

目前已有的自动程序修复技术可以大致的划分为两类，基于测试的自动程序修复技术和其它类型的自动程序修复技术，大部分自动程序修复技术的研究和成果都集中在基于测试的修复领域中，这在第 1.2.1 节程序修复研究现状中已经进行了介绍。而本文涉及到的程序修复技术也使用的是基于测试的修复技术，因此，本文后续提到的程序修复技术专指基于测试的修复技术。在基于测试的自动程序修复技术中，其输入一般是缺陷程序及其配套的测试用例，并且至少有一个测试用例可以触发错误从而造成程序运行异常。该技术的输出为一个或多个候选补丁，生成零个补丁则表示该程序无法被修复。在自动程序修复技术中，测试用例在整个过程中扮演着非常重要的角色，用于描述并限定程序的正确行为，并确保补丁不会违反程序的需求说明，因此任何自动生成的补丁至少保证了其能在配套测试用例上正确运行。

自动程序修复技术的整个过程主要分为三个阶段 [6]：缺陷定位阶段、补丁生成阶段、验证评估阶段。图 2.1 展示了自动程序修复技术的一般流程。缺陷定位阶段是整个修复过程的基础，定位的结果会对后续阶段造成很大影响。缺陷定位阶段的目标是找到程序中可能含有错误，目前已有的缺陷定位方法可以被分为静态定位法和动态定位法两类。静态定位法通过静态程序分析，获取程序当中的各种依赖关系，从而确定错误语句可能出现的位置，该方法存在的最大问题就是计算开销很大，因此在自动程序修复技术中很少被用来定位缺陷。动态定位法通过程序执行测试用例得到的测试结果和执行轨迹，来确定错误语句可能出现的位置。在动态定位法中，研究人员利用可疑度（Suspiciousness）和程序四元组来描述程序包含缺陷的可能性大小，决定了该段程序在修复时被选中的可能性 [45] 。程序四元组表示为 $N(s) =< n_{ep}(s), n_{ef}(s), n_{np}(s), n_{nf}(s) >$，其中 $n_{ep}(s)$ 表示覆盖到程序 s 的成功测试用例数，$n_{ef}(s)$ 表示覆盖到程序 s 的失败测

试用例数，$n_{np}(s)$ 表示未覆盖到程序 s 的成功测试用例数，$n_{nf}(s)$ 表示未覆盖到程序 s 的失败测试用例数，根据以上定义，该段程序的可疑度与覆盖其的成功测试用例数成反比，与覆盖其的失败测试用例数成正比。当前不同自动程序修复技术在缺陷定位阶段，都是围绕以上概念设计的定位公式和算法 [4, 46]。

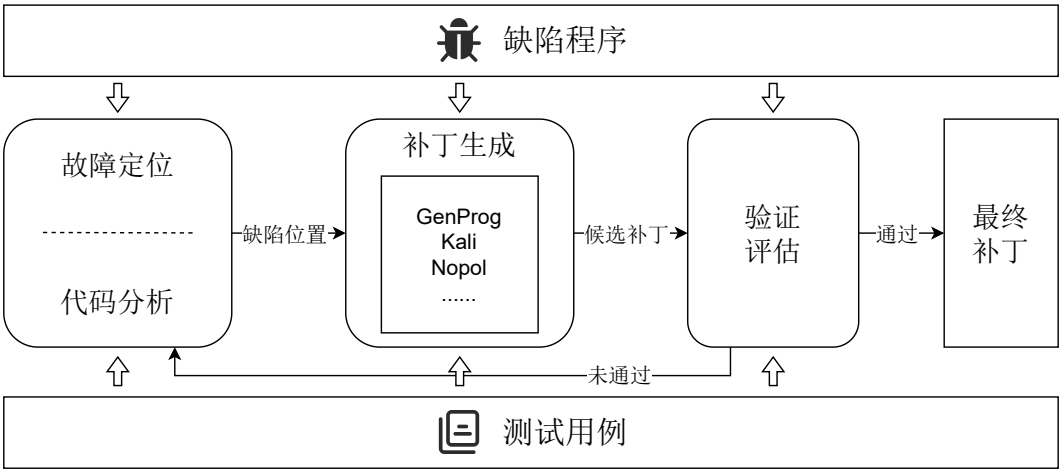

图 2.1: 自动程序修复技术的一般流程

补丁生成阶段通过预先定义好的规则对错误语句进行修改，生成新的能通过测试用例的代码，然后将其输出为补丁。补丁生成的方法主要可以分为三类，基于搜索的生成方法、基于语义的生成方法和其他类型的生成方法。首先，基于搜索的生成方法是受到基于搜索的软件工程 [13] 的启发，而被提出的用于生成补丁的方法，该方法通过在搜索空间中查找从而获取补丁，然后借助测试用例验证补丁的正确性 [15] 。搜索空间是基于搜索的生成方法中的关键，搜索空间可以包括程序自身、开源项目、问答社区等，搜索空间的设定很大程度影响了修复的效率，空间过小则难以找到最优补丁，空间过大则会拉低算法的效率。基于语义的生成方法借助了天使调试 [47] 的思路，在缺陷定位之后生成能使程序通过测试的天使值，然后从运行时信息中提取补丁的约束条件，最后通过程序合成得出补丁。基于搜索的生成方法和基于语义的生成方法各有各的优势，结合两者的特点形成综合的生成方法是未来自动程序修复技术的研究趋势。

验证评估阶段将生成的补丁传给测试用例进行验证评估，来决定该补丁能否作为结果输出。该阶段使用地主要方法就是执行测试用例，但是由于补丁生成方法众多，候选补丁同样数量多且质量不一，这一阶段的开销也不可忽略。为了更快地找到正确的补丁，研究人员在对补丁验证评估时着重关注两个方面，测试用例和补丁特征。在测试用例方面，研究人员的主要目标是构造判别能力强的测试用例，提升测试用例的健壮性，这样可以快速移除评估错误补丁的时间

花销；在补丁特征方面，研究人员旨在建立起一个通用的特征模型，优先验证符合模型公式的补丁，从而更早地选出正确补丁。

根据以上对基于测试的自动程序修复技术的概述，可以总结出补丁在自动程序修复技术中的生成轨迹。因此，根据分析补丁在自动程序修复的过程，通过优化测试用例，对补丁进行审核的思路是可行的。减少评估错误补丁的时间花销，实现更快的找到正确补丁。

2.2　代码分析技术

在面向自动程序修复的人机交互式补丁审核系统中，我们需要对缺陷项目以及补丁进行分析，提取有用的信息辅助开发者与机器的交互。因此，本节将介绍代码分析技术相关的工作，为本文技术的实现做准备。在代码分析领域，研究人员提出了许多方法来解释和分析代码的具体特性，本文将主要介绍程序代码切片和代码特征提取这两种对程序代码分析的技术手段。

2.2.1　程序代码切片

随着软件复杂度的日益增大，维护和分析复杂的程序成了一大难题。程序代码切片是一种用于将复杂程序进行分解的程序分析技术，M.Weiser 在 1979 年首次提出它的原理和方法 [48] 。程序代码切片通过删除与主体程序无关的语句，将程序分割成若干互不影响的代码片段，从而达到分解大型程序代码的目的。目前已广泛应用在软件分析和理解、软件测试、运行维护等领域。

目前来说，程序代码切片主要有三种基础方法。最开始是根据控制流图（Control Flow Graph，CFG）进行建模，M.Weiser 在当时较为流行的程序控制流图的基础上建立了数据流方程，通过对数据流的分析实现代码切片。但是根据控制流图建模的方法效率不高，并且只能解决基本的程序结构，于是 K.J.Ottenstein 和 L.M.Ottenstein 在此基础上做进一步改进，提出了根据程序依赖图（Program Dependence Graph，PDG）建模，产生了基于程序依赖图的图可达性算法，在程序代码切片技术上取得了巨大突破。后来又对该方法做进一步改进，提出了根据系统依赖图（System Dependence Graph，SDG）建模的方法 [49] ，设计了基于系统依赖图的两步遍历图可达性算法，解决了包含多个过程的代码切片问题，使得程序代码切片技术基本趋于成熟。

实现程序代码切片技术的流程一般包括如下三个步骤 [50] 。第一步，提取代码依赖关系，从代码中提取各类信息，包括基本块和执行流程等信息，形成对应的程序依赖图；第二步，制定代码切片规则，根据具体的程序分析的具体要求和需求说明设计切片规则；第三步，借助规则生成切片，根据第二步中设计的切

片规则，针对第一步中提取的依赖关系进行分析处理，从而完成程序代码的切片。本文在实现补丁解析与准备的功能时，依据该流程，对缺陷项目中和补丁中的代码进行切分，从而为后续信息的提取做铺垫。

2.2.2　代码特征提取

代码特征提取是代码分析的一个重要手段。分析代码首先要找到一个合适的数据结构来存储和表示所读取的代码，鉴于程序本身的层次型结构，软件研究人员一般选用树状结构来描述代码。抽象语法树（Abstract Syntax Tree，AST）是一种使用树状结构来表示源代码的抽象语法结构 [51]，树上的每一个节点都对应源代码中的一种语法结构，它是程序代码的一种常用且流行的中间表示形式，在代码分析、优化、编译等领域得到了广泛的应用。

抽象语法树最大的特点在于抽象，它的抽象具体表现在两个方面。第一个方面是不依赖于编程语言，当代码转换为树时并不会表示出普通源代码中出现的每一个细节；第二个方面是不依赖于具体文法，无论什么语法规则，都要求在语法分析时，构造出相同的语法树。抽象语法树的构建一般分为两个步骤，通过词法分析到语法分析，最终将源代码转化为树状表达形式。目前已经出现一些使用抽象语法树进行代码特征提取的相关工具，这些工具都具有能够将源代码转换为抽象语法树的功能，但是不同的工具对于树节点的定义是存在差异的。

Eclipse 中的 Eclipse JDT 提供了一组访问和操作 Java 源代码的 API，Eclipse AST [1] 是实现该功能的一个重要组成部分，可以实现对一个程序的抽象语法树的获取、创建、访问和修改。一个完整 Java 程序代码，通过 Eclipse AST 可以转化为十分精细的树状图。其中一些的重要节点，CompilationUnit 表示完整的 Java 程序，TypeDeclaration 表示类声明，FieldDeclaration 表示属性声明，MethodDeclaration 表示属性声明，方便了我们对语法树的调用和操作。

在 Eclipse AST 中，最重要的操作是创建、访问和修改。Eclipse AST 提供了 ASTParser 类用于解析源代码从而创建抽象语法树，它可以通过 Java Model 的形式导入源代码，或者是通过字符数组并指定代码类型。Eclipse AST 通过使用 ASTNode 类和 ASTVisitor 类来访问节点，ASTNode 类表示树的节点，而 ASTVisitor 用于查询这些节点。Eclipse AST 采用访问者模式实现了访问节点的功能，树的节点是待访问代码的具体语法结构，通过继承 ASTVisitor 来承担具体访问者的角色。抽象语法树的修改主要包括三个方面的内容：修改节点、移动节点和创建节点，在实现抽象语法树创建和访问的基础上，Eclipse AST 同样提供了一系列操作实现了对修改语法树的支持。

[1] https://www.eclipse.org/jdt/ui/astview/

借助 Eclipse AST 提供的代码分析 API，能够有效完成对缺陷项目和补丁等代码的分析，为人机交互过程做好数据准备。同时，面向自动程序修复的人机交互式补丁审核系统最终将通过 Eclipse 插件来实现，因此可以很好的兼容在一起。

2.3　测试相关技术

2.3.1　测试生成

软件测试是软件工程中的重要组成部分，在软件开发生命周期中承担着必不可少的角色。软件测试是一种在测试用例上运行程序，然后将实际结果与预期结果进行比较的过程。早期的软件测试主要由人工主导，但是人工测试是一项高强度且高成本的工作，研究表明，测试成本通常占开发成本的 50% 以上 [52] 。因此，将测试过程自动化，能够提高测试效率，降低开发成本。随着软件工程的迅速发展，目前在测试自动化的领域上已经产生大量的研究，同时许多自动化测试工具也被广泛使用。

软件测试的基本目标是花费较少的成本使用有限的测试用例来提高发现错误的能力。测试用例是软件测试的核心，它来源于被测系统（System Under Test，SUT）的源代码，以及规范、文档等软件制品，可以看作是软件需求的代码表达。测试用例通过检查潜在的输入和执行路径，来发现被测系统中的错误，对整个测试过程的有效性会产生重大影响。测试用例的生命周期包括测试用例的生成、选择、优化和评估等活动，其中测试用例的生成，是众多测试活动中最基础和关键的一项活动 [53] 。如前文所说，测试用例的生成最开始也必须手动编写，但是随着软件规模的日益复杂和软件工程的迅速发展，测试用例生成的自动化技术开始逐渐在测试实践中采用，本文称生成测试用例的自动化技术为测试生成技术。

目前，测试生成技术主要可以分为三类 [54] ，基于随机算法的生成技术，基于搜索的生成技术和基于数据挖掘的生成技术。基于随机算法的生成技术是通过随机算法生成测试用例的，实施成本较低，能够简单轻易的在被测系统中生成大量的测试用例，凭借在测试过程中引入随机性来暴露系统中可能的错误，但同时这种不可控的不确定性也成为了该技术的主要缺点，因此基于随机算法的生成技术通常用来分析评估其它技术的有效性。基于搜索的生成技术来源于基于搜索的测试，它是上文提到的基于搜索的软件工程的分支，通过优化过的算法自动搜寻指定空间的测试数据，从而最大程度地实现测试目标，同时大幅降低测试成本。基于数据挖掘的生成技术通过分析被测系统的输入输出来优化测试用例的数量，该技术极大提高了测试效率并降低了测试成本，但是它的实现

颇为复杂，仍处于发展阶段。在上述提到的三类测试生成技术中，基于搜索的生成技术具有较高的通用性和广泛的适用性，也是大量生成方法主要集中的领域 [54]，包括基于模拟退火算法（Simulated Annealing，SA）的测试生成方法、基于遗传算法（Genetic Algorithms，GA）的测试生成方法等。

针对测试生成技术相关的工具已经有很多得到了应用，针对 C 语言的工具有 DART [55]、CUTE [56] 这样稳定和广泛使用的工具；在 Java 语言中，也有 Randoop [57]、TestFul [58] 等工具。EvoSuite [59] 是用于 Java 的最新的基于搜索的测试生成工具，它是由 Sheffield 等大学联合开发的一种开源工具，在 SBST 2017 单元测试工具竞赛中，收获了所有竞争工具中的最高分，赢得了广发的青睐与关注。该工具使用一种进化方法，针对代码覆盖率标准，自动生成并优化带有断言的测试用例以及可能的测试预言，生成的测试用例全都符合 JUnit 的标准，可直接在 JUnit 中运行。EvoSuite 使用遗传算法生成测试用例，目的是达到尽可能高的代码覆盖率。遗传算法从初始随机生成的测试用例开始，然后通过应用选择、交叉分析和变异算子等逐步优化这些测试用例，直到找出达到完全覆盖率的解决方案或遇到停止条件为止 [60]。

除了满足通常被测系统所要达到的代码覆盖标准之外，EvoSuite 同样考虑了生成的测试用例的可读性，可读性也成为了 EvoSuite 进一步优化的目标。EvoSuite 采用的一些提升可读性的方法如下。一是生成较小的测试用例。众所周知，大段冗长的代码极易影响整体的可读性。EvoSuite 的搜索更偏向于较小的测试用例。同时，EvoSuite 在最小化处理时将测试用例按源代码中出现的顺序排序，这样 Junit 集合中的测试用例的顺序与被测类中方法的顺序相匹配。二是对测试用例中的值的优化。在最小化之后，所有值要么都是不必要的，要么就是实现覆盖率目标所必需的值。三是在编写 JUnit 测试代码之前会内联所有常量，为了进一步缩短测试时间并避免由于变量重用而造成的混乱。最后，EvoSuite 使用命名约定对 JUnit 测试代码中的变量名进行规范，包括常用的驼峰式命名法，使得开发者能够更清楚的知道变量的类型以及变量之间如何交互。

综上而言，EvoSuite 具有如下特点，是本文选择它作为实现测试生成功能的工具。

1. EvoSuite 生成的测试用例符合 JUnit 4 标准，这是目前被广泛使用的测试标准。它能够为选定的类生成测试用例，针对特定的类而不用涉及整个项目从而产生冗余的测试用例。
2. EvoSuite 能够优化不同的覆盖标准，综合提高覆盖率，如行覆盖、分支覆盖和变异覆盖，尽可能的减少开发者对测试用例优化的工作。

3. EvoSuite 生成的测试用例还具有很高的可读性，包括将用例个数进行精简、变量命名遵从规范等等，极大方便了开发者与本框架进行交互。

通过前文的相关内容介绍，我们可以认识到测试在程序修复中的重要作用。据我们所知，关于使用测试用例生成在程序修复中进行补丁筛选的工作还较少。根据本节对测试生成技术的概述，利用 EvoSuite 来实现面向自动程序修复的人机交互式补丁审核系统中测试生成的功能，也就是交互中机器完成的任务，是可行的方法。

2.3.2　测试预言

测试预言（Test Oracle）可以定义为，一种决定与输入关联的输出是否正确的机制。测试预言是软件测试中一个相对被忽略却必不可少的部分，尽管目前有许多能够自动生成测试用例的工具，但是受限于测试生成技术本身，由机器生成的测试用例始终没有解决测试预言问题。根据 Long 等人所做的研究调查，测试用例通常都不具有很高的充分性，比如与原有测试用例拥有类似的判断机制、未考虑特殊情况等等，以至于大多数合理的补丁都是不正确的，这使得程序修复技术很难确保补丁的正确性。而测试用例的低充分性的主要原因很大程度上来源于测试预言的问题，在软件测试中，缺少测试预言是一个长期存在的问题，这就是本文要解决的主要问题。

根据测试预言构建的知识来源，Barr 等人 [61] 将测试预言分为三大类：规范测试预言、隐式测试预言以及衍生测试预言。规范测试预言是从系统行为的正式规范中获得的，使用数学逻辑定义特定领域的测试预言，判断系统的行为是否符合形式规范。规范测试预言能够有效地发现错误，但是它的成功很大程度上取决于正式规范的可用性。隐式测试预言是在不参考任何领域知识或正式规范的情况下生成的，是依靠一般的隐式知识来区分系统的正确行为和错误行为。隐式测试预言适合建立在检测异常的过程上，例如程序崩溃或者异常终止。派生测试预言是根据从各种工件，例如文档、系统执行信息，以及被测系统的属性或者版本获得的信息，将系统的正确与错误行为区分开。在缺乏自动化构建测试预言的情况下，确定如何区分被测系统行为正确性的任务就落在了人身上 [62] 。

通常，软件研究人员使用两个属性评价测试预言的质量 [63]： 完整性和健壮性。完整性是指测试预言接受所有正确状态的性能，这表示仅对执行错误的用例发出警报，没有出现误报的情况。健壮性是指测试预言拒绝所有错误状态的能力，即表示不会遗漏任何的程序错误。为了使得测试预言达到完整性和健壮性，这就要求我们能建立合理的预言机制。

在测试预言问题被研究的同时，也向新的领域进行了开展，其中针对测试预言难以构建而提出的测试预言重用是值得关注的领域之一。基于软件产品的版本或分支中的旧预言，来构建新预言，是解决测试预言问题的可能方向之一 [61] 。

2.4　系统开发技术

为了方便面向自动程序修复的人机交互式补丁审核系统的展示和实施，我们使用 Eclipse 将其开发成一个工具，本节将介绍插件开发技术的相关工作 [64] 。Eclipse 是 IBM 公司发起开发的用于 Java 的本地 IDE 开发环境，如今围绕着 Eclipse 已经发展成为了一个庞大的 Eclipse 联盟。同时，Eclipse 作为一个开源社区的软件项目，专注于为高度集成的工具开发提供一个全功能、高品质的开发平台。Eclipse 作为一个广受欢迎的 IDE，不仅仅是一个优秀的 Java 开发环境，更是一个工具集成平台。

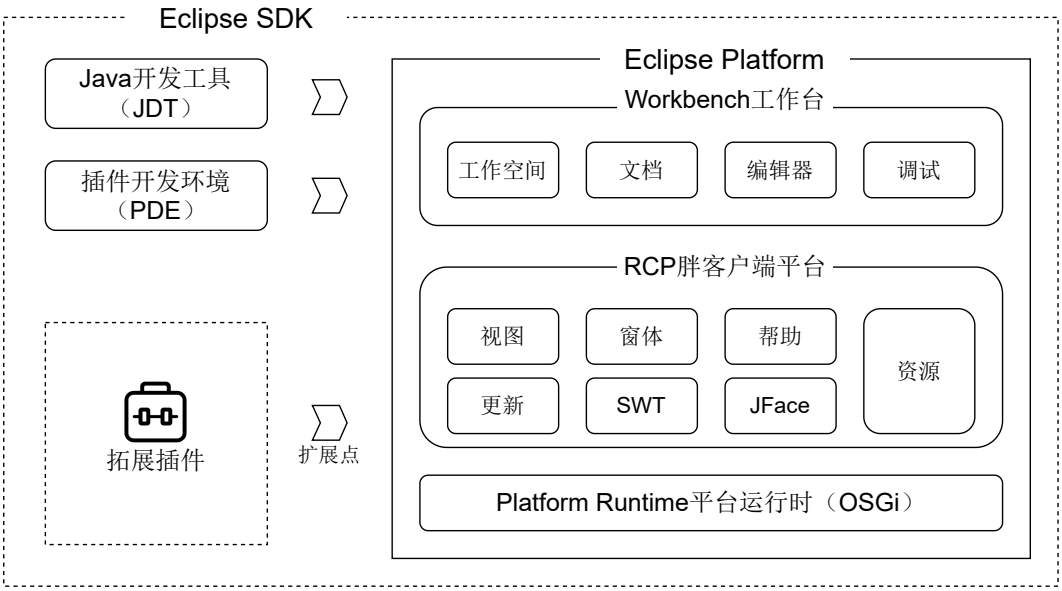

图 2.2: Eclipse 体系结构

Eclipse 最大的特性就是可扩展性。它从最初就被设计成可扩展的集成开发环境，它的可扩展性体现在可以利用插件来实现不同的功能，或者说它使用了插件架构技术。Eclipse 体系结构的基础组件是插件，插件是 Eclipse 定义新功能的核心机制，插件的大小是可以自己定义的，Eclipse 的 Java 开发环境本身就是一个大插件，可以说插件成就了今天的 Eclipse。Eclipse 的内核很小，它通过一个体系结构和一套 Java 架构对无数现有工具进行集成，从而达到了对各种语言和功能的支持。Eclipse 的体系结构是分层设计的，如图 2.2 所示，主要包括平台

运行时环境（Platform Runtime）、胖客户端平台 (Rich Client Platform，RCP）和工作台（Workbench）。其中用户接口、工作台、资源等功能不仅作为 Eclipse 的一部分来使用，同时也被作为 API 的形式提供出来，方便开发者进行自主调用。

插件之间之所以能够进行功能扩展，主要是通过扩展点的概念来实现的。扩展点描述了一个公开声明插件可扩展性的结构化方法，定义了其它插件可以向本插件添加功能的位置，如图所示。插件清单是用 XML 文件指定的，记录了插件的扩展和扩展点的定义等信息，插件清单和扩展一起描述了插件之间的关联。Eclipse 在启动时，首先找到本地安装的插件，读取插件清单中的信息，匹配扩展和扩展点，从而呈现我们所熟悉的开发环境和界面。Eclipse 为了方便插件的开发，还提供了插件开发环境（Plugin Development Environment，PDE）简化开发操作，比如 Manifest 编辑器就为插件配置文件提供了图形化的编辑功能。本文的技术框架选用 Eclipse 插件进行开发，主要源于如下几点优势。

1. Eclipse 拥有集成不同工具和应用的能力，方便后期系统进行功能扩增，添加的功能可以作为独立的插件来集成，不用过分关注通用功能部分的代码。
2. Eclipse 提供了数量庞多且有用的 API，有助于我们对文件资源进行操作，第 2.3 节也提到 Eclipse 对代码分析相关功能的支持。
3. Eclipse 是 Java 架构，拥有大量的使用者，在开发者中十分普及；其体系结构中的 SWT 为开发者提供了一个易于编程的图形界面框架，极大方便了人机交互界面的构建。
4. Eclipse 是一个稳定的项目，其基金会也一直维护着软件的持续更新，并且支持所有主流桌面平台，部署方便，节省了开发和维护的成本。

2.5　本章小结

本章介绍了面向自动程序修复的人机交互式补丁审核系统开发实现过程中所使用到的相关技术和工具，介绍这些技术的基本概念、原理和应用，阐述这些技术在本文技术框架中发挥的重要作用。本框架所用到的相关技术主要包括程序修复技术、代码分析技术、测试相关技术和系统开发技术。程序修复技术是本框架的基础，为本文提供了补丁数据，通过了解补丁在程序修复技术中的生命轨迹，能够让我们更深的认识到补丁的筛选从何入手；代码分析技术实现对缺陷项目、补丁等其中的代码分析和解释，提取其中的信息；测试生成技术作为本文人机交互中，机器自动生成测试用例的支持；使用 Eclipse 插件实现本文工具的具体实现。本文基于这些技术和相关工具，分析程序修复领域中研究的不足，从而设计并实现了面向自动程序修复的人机交互式补丁审核系统。

第三章　系统需求分析与概要设计

3.1　系统整体概述

这一章将会介绍面向自动程序修复的人机交互式补丁审核系统的需求分析和概要设计，图 3.1 展示了面向自动程序修复的人机交互式补丁审核系统的工作流程。我们将缺陷程序和配套的测试用例输入给自动程序修复工具，通过缺陷定位算法定位输入的缺陷程序中可能出错的具体位置，补丁生成算法根据定位生成一定数量的候选补丁，并保证了补丁在配套的测试用例上可以正常运行。生成补丁的过程结束后，产生的候选补丁输入到面向自动程序修复的人机交互式补丁审核系统中。面向自动程序修复的人机交互式补丁审核系统首先对待修复的项目和待审核的补丁进行数据分析与准备，提取主要的代码特征。然后进入交互式审核过程，机器利用自动化测试工具为其扩增测试数据，并进行测试分析；开发者为机器生成的测试用例做出修改和完善，主要是对测试预言进行标注，最终形成充分性较高的测试用例验证补丁的正确性。

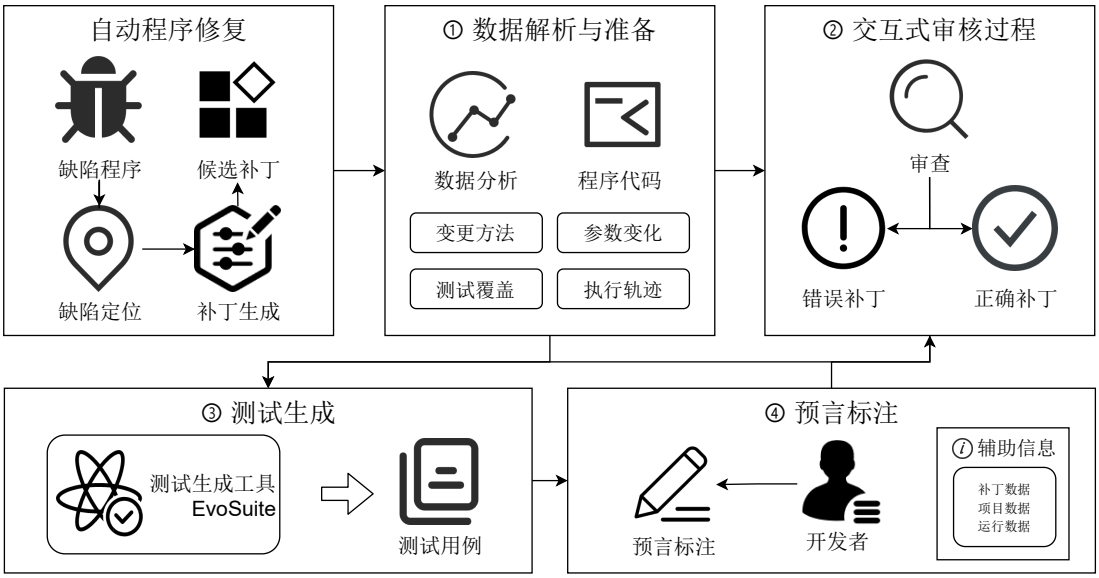

图 3.1: 面向自动程序修复的人机交互式补丁审核系统的工作流程

本文结合代码分析技术和测试相关技术，提出并设计了面向自动程序修复的人机交互式补丁审核系统，本系统面向的对象是对补丁进行审核筛选工作的开发者，作为程序修复服务过程中的一部分，接收来自上一阶段生成的候选补

丁和缺陷程序作为输入。本文将整个系统的实现分为两个过程，分别是原始数据分析过程和交补丁互审核过程，这两个过程完全描述了用户参与系统运转的一般步骤，基于此我们对系统开展需求分析和总体设计，分别在第 3.2 节和第 3.3 节进行讨论；按照业务功能划分，整个系统可以划分为三个模块，分别是数据分析模块、测试管理模块和人机交互模块，将在第 3.4 节进行讨论。

3.2　系统需求分析

3.2.1　概念介绍

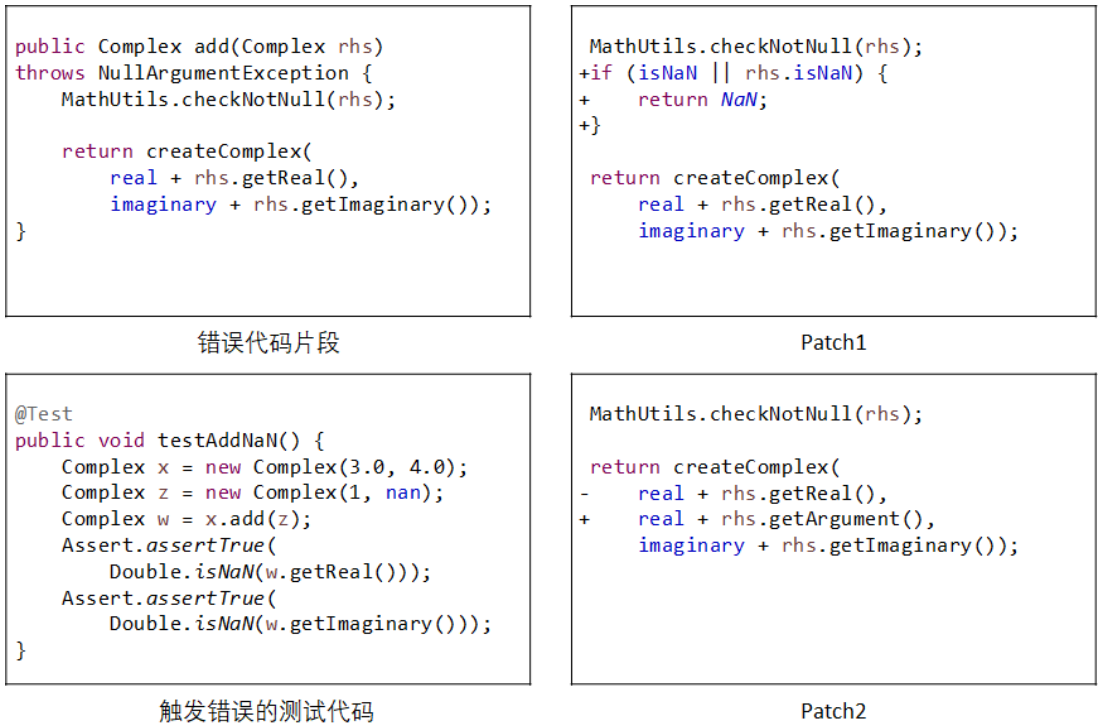

图 3.2: 错误代码片段与补丁

过拟合（Overfitting）问题经验证是自动程序修复技术中普遍存在的问题。过拟合的概念来自于机器学习的领域 [65]，在机器学习领域，当一个假设在训练数据集上能够获得比其他假设更好的拟合，但是在测试数据集上却不能如预期一样拟合数据，那么可以认为这个假设出现了过拟合问题。Smith 等人 [27] 借助这一概念，在 IntroClass 缺陷库的基础上，通过分析众多方法和工具，研究了自动程序修复技术中的过拟合问题，确认了在程序修复领域常见的方法均存在过拟合问题。将缺陷程序配套测试用例划分成两部分，训练集和测试集，前者用于生成补丁，后者用于审核补丁。如果补丁可以在训练集上执行成功，却在测试

集上执行失败，那么有理由认为该补丁存在过拟合问题。因此，可以将补丁的过拟合问题定义为过度专注所用测试用例而无法应用到其它测试用例。在本文中，我们就称该补丁是不正确的补丁。

图 3.2 展示了来自缺陷数据集 Defects4J 中的缺陷项目 Math53 的错误代码片段，触发该错误的测试用例，以及两个由自动程序修复工具生成的补丁 Patch1 和 Patch2，并且他们都可以修复 Math85 从而使其能够通过测试用例。其中 Patch1 是一个正确的补丁，它修复了项目中的缺陷，并通过了更多测试用例的验证；但是 Patch2 通过更改返回值修复错误，如果另写一个新的测试用例，那么 Patch2 则会执行失败，因此 Patch2 存在着过拟合问题，是一个不正确的补丁。

```
 MathUtils.checkNotNull(rhs);

 return createComplex(
-    real + rhs.getReal(),
+    real + rhs.getArgument(),
     imaginary + rhs.getImaginary());
```

Patch

```
@Test
public void testAddNaN() {
    Complex x = new Complex(3.0, 4.0);
    Complex z = new Complex(1, nan);
    Complex w = x.add(z);
    Assert.assertTrue(
        Double.isNaN(w.getReal()));
    Assert.assertTrue(
        Double.isNaN(w.getImaginary()));
}
```

原有测试用例

```
@Test
public void testAddNaN() {
    Complex x = new Complex(3.0, 4.0);
    Complex z = new Complex(nan, 1);
    Complex w = x.add(z);
    Assert.assertTrue(
        Double.isNaN(w.getReal()));
    Assert.assertTrue(
        Double.isNaN(w.getImaginary()));
}
```

新增测试用例

图 3.3: 错误补丁与测试用例

借助对过拟合问题的一般解决思路进行分析，我们可以认为自动程序修复技术中补丁存在过拟合问题的主要原因是测试用例的数量较少或者是质量较低。在本文中，我们将包含这样问题的测试用例称之为不充分的测试用例。图 3.3 展示了针对缺陷项目 Math53 的错误代码生成的不正确的补丁，原有的测试用例，以及新增的测试用例。我们可以发现，原有的测试用例是无法检测出该补丁是不正确的。但是通过分析方法调用，对测试用例中检测机制进行修改，变为新的测试用例，那么就可以有效的辨别出这个不正确的补丁。

3.2.2　功能需求

表 3.1: 系统功能需求表

需求编号	需求名称	需求描述
R1	原始数据处理	系统对缺陷程序和候选补丁进行代码分析，将故障方法从程序中分离出来，并提取主要的代码特征
R2	选择待审补丁	用户可以通过交互界面浏览候选补丁列表，并从候选补丁列表中选择需要审核的补丁
R3	代码特征展示	根据用户选择的补丁和处理好的数据，系统展示与之相关的代码信息
R4	测试用例生成	系统根据用户要求，针对待审补丁为故障方法所在类生成测试用例
R5	测试预言标注	用户结合代码信息修改机器生成的测试用例，完善测试用例的充分性
R6	缺陷程序测试	系统在用户完成预言标注后，为待审补丁和缺陷程序进行测试评估，并反馈测试结果
R7	选定正确补丁	用户根据测试结果从列表中将某补丁选择为最终答案
R8	删除错误补丁	用户可以从候选补丁列表中移除错误的补丁

面向自动程序修复的人机交互式补丁审核系统，以自动程序修复工具生成的候选补丁为基础，根据特定的算法流程，结合机器自动生成的测试用例和开发者辅助的测试预言，从而有效筛选出正确的补丁程序。该系统为开发者在故障定位、程序调试、缺陷修复等方面都提供了有效的帮助。作为一个软件系统，首先满足基本的功能需求。面向自动程序修复的人机交互式补丁审核系统的主要功能需求分为原始数据处理、选择待审补丁、代码特征展示、测试用例生成、测试预言标注、缺陷程序测试、选定正确补丁、删除错误补丁。系统的功能需求表整理如表 3.1 所示。

3.2.3　非功能需求

表 3.2: 系统非功能需求表

类型	非功能需求描述
可靠性	系统故障可追溯
易用性	系统有良好的交互界面以及明确的提示信息 用户能够在短时间内快速上手并熟悉系统功能并完成任务目标 系统提供的不同交互信息风格一致且易于理解
可扩展性	系统未来在面对修改的时候能保证持续提升的能力 系统可以集成新的工具，也可以添加新的数据源
可维护性	系统通过日志记录整体运行情况，及时发现问题
性能	系统主要功能加载需要在 3 秒之内完成 系统主要功能的响应时间要控制在 2 秒之内

面向自动程序修复的人机交互式补丁审核系统旨在为开发者在程序调试、

错误修复时提供帮助，在满足系统基本功能的前提下，还需要对系统的非功能需求做到严格把控。非功能需求是隐藏在功能需求背后的需求，对系统行为以及用户体验，都会产生重大影响。本系统的非功能性需求列表如表所示，主要从可靠性、易用性、可拓展性、可维护性和性能五个方面进行描述，开发者需要通过系统筛选出正确的补丁，因此特别需要考虑系统的可靠性、可维护性和相关性能；而为了降低程序修复的成本，易用性也要考虑在内；而随着修复工具、测试工具越来越丰富，系统的可扩展性也需要保证。系统的非功能需求表整理如表 3.2 所示。

3.2.4　系统用例设计

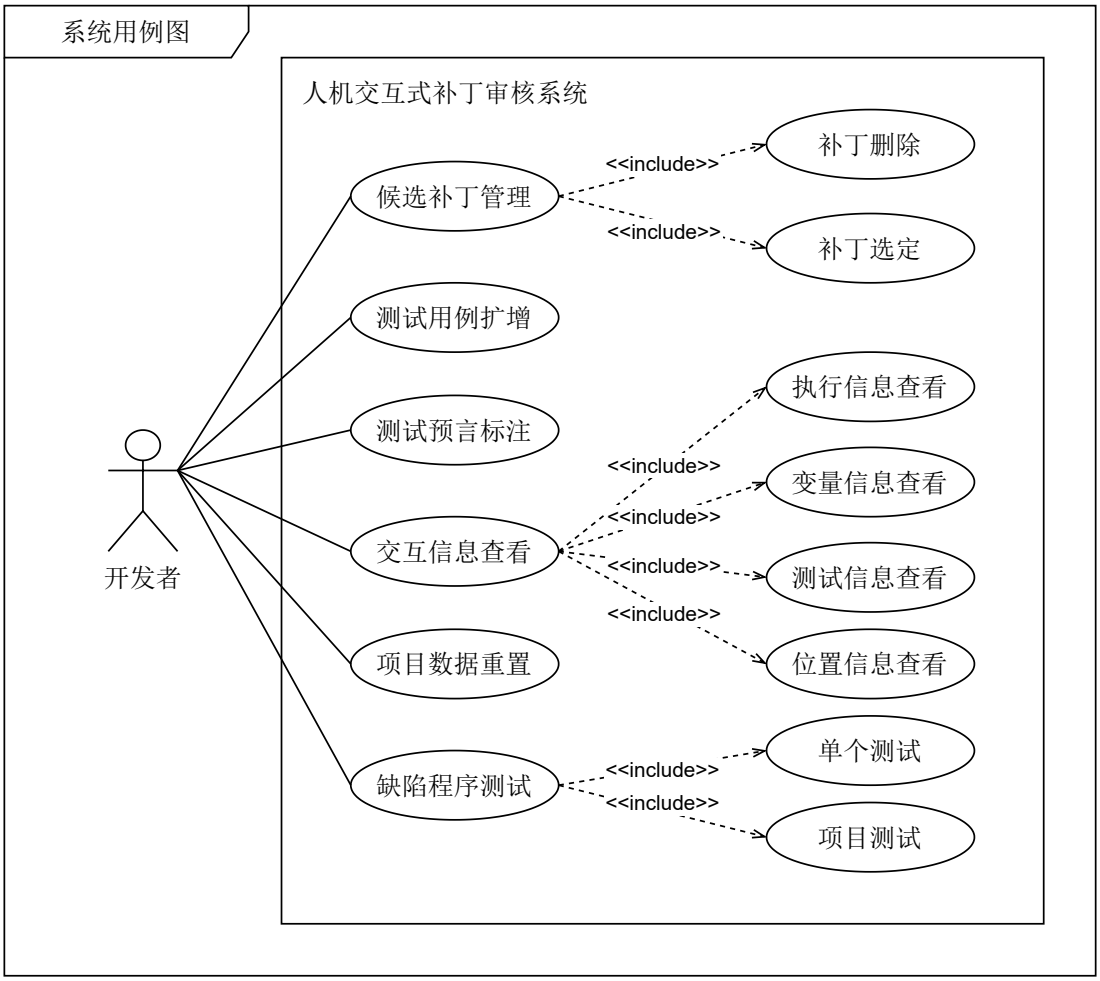

图 3.4: 系统用例图

根据上文所分析的功能性需求和非功能性需求，本系统的用户只有一类，即开发者，开发者可以与系统中的用例进行全方位的参与和交互。系统涉及到的

主要用例有 6 个，分别是候选补丁管理、交互信息查看、测试用例扩增、测试预言标注、缺陷程序测试、项目数据重置。本系统用例图如图 3.4 所示。

3.2.5　系统用例描述

本节将结合功能需求就上述 6 个系统用例进行详细描述，描述内容包括用例名称、参与者、用例描述、前置条件、基本流程、其它流程和后置条件。

表 3.3: 候选补丁管理系统用例

用例 ID	UC01
用例名称	候选补丁管理
参与者	开发者
用例描述	开发者对候选补丁列表中的补丁进行相关操作
前置条件	系统已完成对缺陷项目和补丁的分析
基本流程	1. 开发者进入系统界面； 2. 系统把所有待审核补丁罗列在候选补丁列表中显示； 3. 开发者点击展开按钮查看补丁详情； 4. 系统显示补丁详情，并提供“删除”、“生成”、“选定”选项按钮； 5. 开发者根据实际情况对补丁进行操作。
其它流程	2a 开发者点击补丁名称，进入系统用例 UC2； 4a. 开发者选择“删除”，删除该补丁； 4b. 开发者选择“生成”，进入系统用例 UC3； 4c. 开发者选择“选定”，将该补丁选择为最终补丁。
后置条件	根据开发者不同的操作，进入后续阶段

该系统涉及到的所有用例都与开发者相关，进入系统后首要系统用例是对候选补丁的管理，其用例描述如表 3.3 所示，开发者可以查看补丁详情，结合实际情况选择对补丁进行操作，主要包括删除补丁、选定补丁和为补丁生成测试用例，之后进入后续阶段。

交互信息查看的系统用例如表 3.4 所示，系统根据开发者点击的补丁，读取并解析在原始数据处理后的文件，按照故障位置、失败测试、执行轨迹、参数变化等类别归类交互信息并展示，这是开发者进行预言标注的重要依据。

测试用例扩增的系统用例如表 3.5所示，当开发者对一个补丁进行审核时，向系统下达扩增测试用例的命令，系统根据项目和补丁信息进行相关配置，对故障方法所在类执行测试生成的指令，完成后展示新增的代码文件。

测试预言标注的系统用例如表 3.6 所示，开发者根据交互信息和测试预言之间的联系，对机器生成的测试用例进行修改，完善测试用例。通过开发者和机器的共同参与，构造充分性更高的测试用例。其中，交互信息的类别以及交互信息如何与预言标注相对应，将在第四章详细设计中介绍。

表 3.4: 交互信息查看系统用例

用例 ID	UC02
用例名称	交互信息查看
参与者	开发者
用例描述	开发者查看与补丁相关的全部交互信息
前置条件	系统已在候选补丁列表中显示候选补丁
基本流程	1. 开发者点击补丁的名称； 2. 系统读取原始数据的处理信息并解析，分类呈现各种交互信息； 3. 系统打开故障方法所在文件，并跳转至指定位置； 4. 开发者浏览交互信息。
其它流程	1a. 开发者展开补丁详情后点击代码，系统做出同样的响应； 2a. 系统首先读取存储故障方法的 JSON 文件，然后读取数据动态分析的文件。
后置条件	开发者结合交互信息执行对补丁的操作

表 3.5: 测试用例扩增系统用例

用例 ID	UC03
用例名称	测试用例扩增
参与者	开发者
用例描述	开发者进行为补丁扩增测试用例的操作
前置条件	系统已在候选补丁列表中显示候选补丁
基本流程	1. 开发者点击“生成”按钮； 2. 系统定位到故障代码的具体位置，并将补丁内容与之替换； 3. 系统根据项目和补丁为测试生成工具配置参数，然后执行生成命令； 4. 系统打开新增测试用例文件； 5. 系统刷新界面，更新与测试相关的交互信息。
其它流程	无
后置条件	新增测试用例等待开发者的预言标注

缺陷程序测试的系统用例如表 3.7 所示，为扩增的测试用例运行项目测试，开发者根据测试结果决定补丁的正确性。如果测试结果为失败，则表明该补丁是不正确的，可以从候选补丁列表中移除；如果测试结果为通过，则补丁既可能是正确的也可能是不正确的，开发者可以选择为该补丁执行新一轮的测试用例扩增和预言标注，对该补丁进行迭代的审核，直到能够做出判断为止。

项目数据重置的系统用例如表 3.8 所示，考虑到开发者在操作过程中可能出现的失误判断或点击等情况，因此设置这一功能。开发者可以根据实际需要恢复所有数据到初始状态，重新开始对缺陷项目的修复流程。

表 3.6: 测试预言标注系统用例

用例 ID	UC04
用例名称	测试预言标注
参与者	开发者
用例描述	开发者对新增测试用例中的预言部分进行修改的操作
前置条件	系统已为缺陷程序扩增了测试用例
基本流程	1. 开发者进入新增测试用例的代码界面； 2. 开发者结合各种交互信息对测试预言进行更新； 3. 系统对修改过的代码进行语法检查。
其它流程	2a. 用户可以选择不修改测试数据。
后置条件	新增测试用例等待测试执行

表 3.7: 缺陷程序测试系统用例

用例 ID	UC05
用例名称	缺陷程序测试
参与者	开发者
用例描述	开发者针对新增测试用例测试缺陷程序的操作
前置条件	开发者完成测试用例扩增和测试预言标注
基本流程	1. 开发者点击“测试”按钮； 2. 系统为新增的测试用例运行 JUnit 测试； 3. 系统界面展示程序测试的结果； 4. 开发者根据测试结果对该补丁的正确性进行标记。
其它流程	2a. 系统执行测试出错，则终止此次流程； 4a. 如果测试结果为全部通过，则开发者可选择再次扩增测试用例并标注预言，然后回到正常流程 1； 4b. 开发者点击“完成”退出此次修复流程。
后置条件	系统反馈缺陷程序测试的结果

3.3　系统总体设计

通过对面向自动程序修复的人机交互式补丁审核系统要解决的问题、功能与非功能需求以及系统用例进行分析，本节将阐述面向自动程序修复的人机交互式补丁审核系统的总体设计。首先，对本系统的架构进行设计，通过系统架构图，划分系统的层次和模块，并阐述它们之间的职责和关联。然后，按照当前普遍应用的“4+1 视图”，从不同视角进一步详细介绍本系统的总体设计。最后，对系统中涉及到的实体进行持久化设计。

表 3.8: 项目数据重置系统用例

用例 ID	UC06
用例名称	项目数据重置
参与者	开发者
用例描述	开发者将所有数据恢复至初始状态
前置条件	无
基本流程	1. 开发者点击“重置”按钮； 2. 系统清除一切对缺陷项目和候选补丁的标记； 3. 系统更新交互界面。
其它流程	无
后置条件	所有候选补丁等待开发者重新开始审核

3.3.1　系统架构设计

图 3.5 描述了面向自动程序修复的人机交互式补丁审核系统的系统架构图，本系统依据 MVC 模式的设计思想，前后端分离，自上而下依次分为视图层、控制层、业务层和数据层。视图层通过 Eclipse 插件的视图元素，展示必要的交互信息，响应并收集开发者的动作；业务层主要负责处理数据运算和具体功能的实现；中间的控制层负责接受用户从视图层输入的指令，决定调用业务层中哪个模型的功能或数据，然后对其进行相应的操作产生最终结果，最后确定用哪个视图来显示返回的数据；数据层则为系统提供数据支持。图形界面和业务模型的分离使得整个系统各部分职责明确，能够有效降低系统各模块之间的耦合度，提高单个模块的重用性。

视图层中，开发者进入系统对某一缺陷程序进行修复，从候选补丁列表中选择将要进行审核的补丁，查看其交互信息，根据相关信息对补丁执行操作。首先指示机器为该补丁所更改的类生成测试用例，然后结合交互信息修改新增的测试用例，最后利用开发者和机器共同构造的测试用例，验证该补丁的正确性。视图层利用 Eclipse 插件开发技术将相关功能集成到本地 IDE 中，方便开发者的交互与使用。控制层本身不提供功能和数据运算，它根据视图层的反馈来决定调用业务层或视图层中的模块来展示和运算，并保证系统整体数据的一致性。

业务层是本系统的核心。首先为修复工作提供前期的数据分析功能，包括缺陷项目、补丁在内的原始数据会在此进行静态分析和动态分析，并将分析结果存储至对应的文件中。而与测试相关的命令调用，都在业务层中通过命令设计模式进行统一管理和调用。业务层通过对每个功能的精细划分和统一管理，为整个系统的业务逻辑提供了强有力的保障，同时为后续添加功能或是集成工具提供了接口。

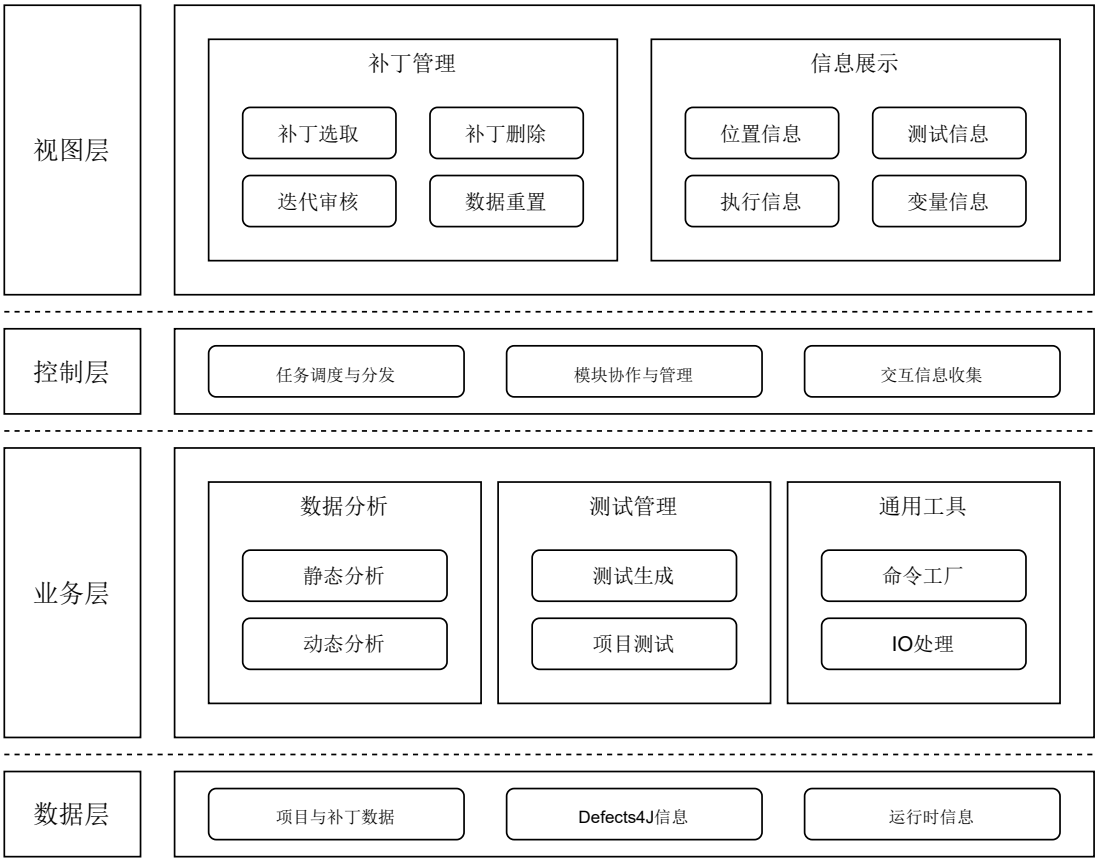

图 3.5: 系统架构图

数据层为本系统提供数据支持，所有数据可以分为三类。第一类是项目与补丁数据，其中包含本文用到的缺陷项目，以及从之前的研究中收集的补丁，经过数据清洗和验证，已经确保它们在版本上保持一致。第二类是 Defects4J 信息，这是通过 Defects4J 提供的命令获取的，包括缺陷项目信息、不同项目触发错误的测试用例信息和依赖地址信息等。第三类是运行时数据，是补丁准备模块通过动静态分析整理出来的信息，包括为方法包装的 JSON 数组、参数变化信息和执行轨迹信息。所有数据分两部分来存储，数据库和文件管理。

按模块划分，本系统主要可以分为三个模块：数据分析模块、测试管理模块和人机交互模块。其中人机交互模块对应图中的整个视图层和部分控制层，数据分析模块和测试管理模块对应图中整个业务层和部分控制层。

3.3.2　4+1 视图模型

“4+1 视图” 是当前软件工程领域普遍应用于软件系统架构划分的方法 [66] ，五个视图相结合能够完整反映出一个系统的设计和架构。其中 1 是场景视图，场景视图又称为用例视图，是系统重要的需求抽象，将前四个视图紧密联系起来，

从用户角度分析系统，对应 UML 中的系统用例图，已在 3.2.3 节中进行了详细介绍，这里不再赘述。其余四个视图分别是逻辑视图、开发视图、进程视图和物理视图，本节接下来将通过这四个视图详细介绍面向自动程序修复的人机交互式补丁审核系统的总体架构设计。

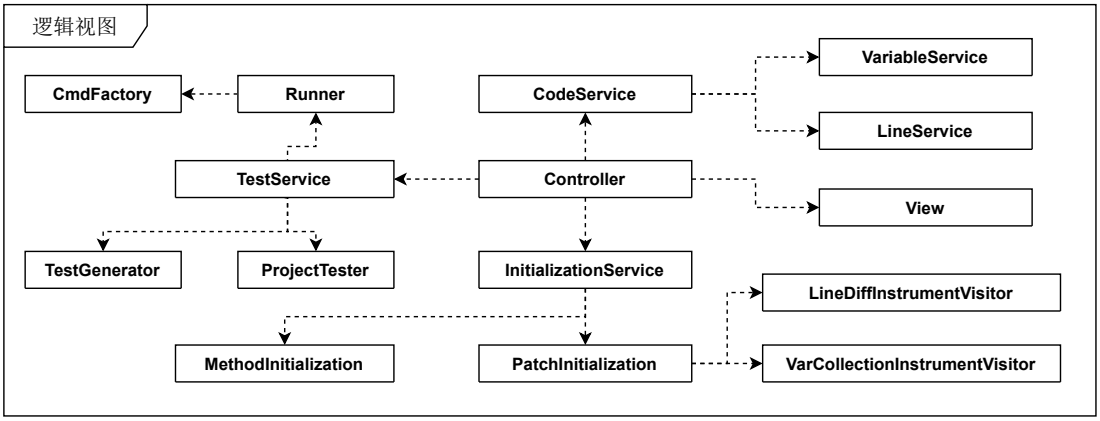

图 3.6: 逻辑视图

逻辑视图主要关注系统的功能需求，从系统最终用户的视角进行分析，将系统功能的抽象进行分解，描述系统提供给用户的功能和服务。根据面向对象的设计方法，我们通过 UML 系统类图进行描述，本系统的逻辑视图如图 3.6 所示。在本系统的逻辑视图中，Controller 为系统的核心类，负责整个系统的调度工作，通过调度多个模块的核心类完成整个系统的协作和运转。其中 InitializationService 负责系统的初始化以及原始数据的处理，它借助 ASTVisitor 对代码进行分析。TestSevice 负责系统所有测试功能，包括测试生成功能以及对所有测试命令的配置和管理。CodeService 是在人机交互模块中，解析由数据分析模块处理的信息，从而转化为开发者易于理解的样式进行交互。

开发视图主要关注系统模块之间的组织与管理，也称作模块视图，从开发人员的视角对系统的组织架构进行描述。根据面向对象的设计方法，我们通过 UML 包图进行描述，本系统的开发视图如图 3.7 所示。在本系统的开发视图中，整个系统被分为视图层、控制层、业务层和数据层。视图层主要包括与用户交互界面的前端资源，其中 plugin 包为插件入口代码包，perspective 包为视图布局包，views 包处理单个视图的信息展示，provider 包为样式化文本和图像的提供代码包。控制层是统筹整个系统的功能调度，controller 包负责控制器管理，是多个模块业务逻辑的执行入口。业务层是系统功能具体的实现，其中 service 包负责控制器下发任务的执行，initialization 包和 visitor 包提供代码分析和处理的功能，run 包管理各种命令的执行，entity 包存储系统中各个实体，util 包提供系统

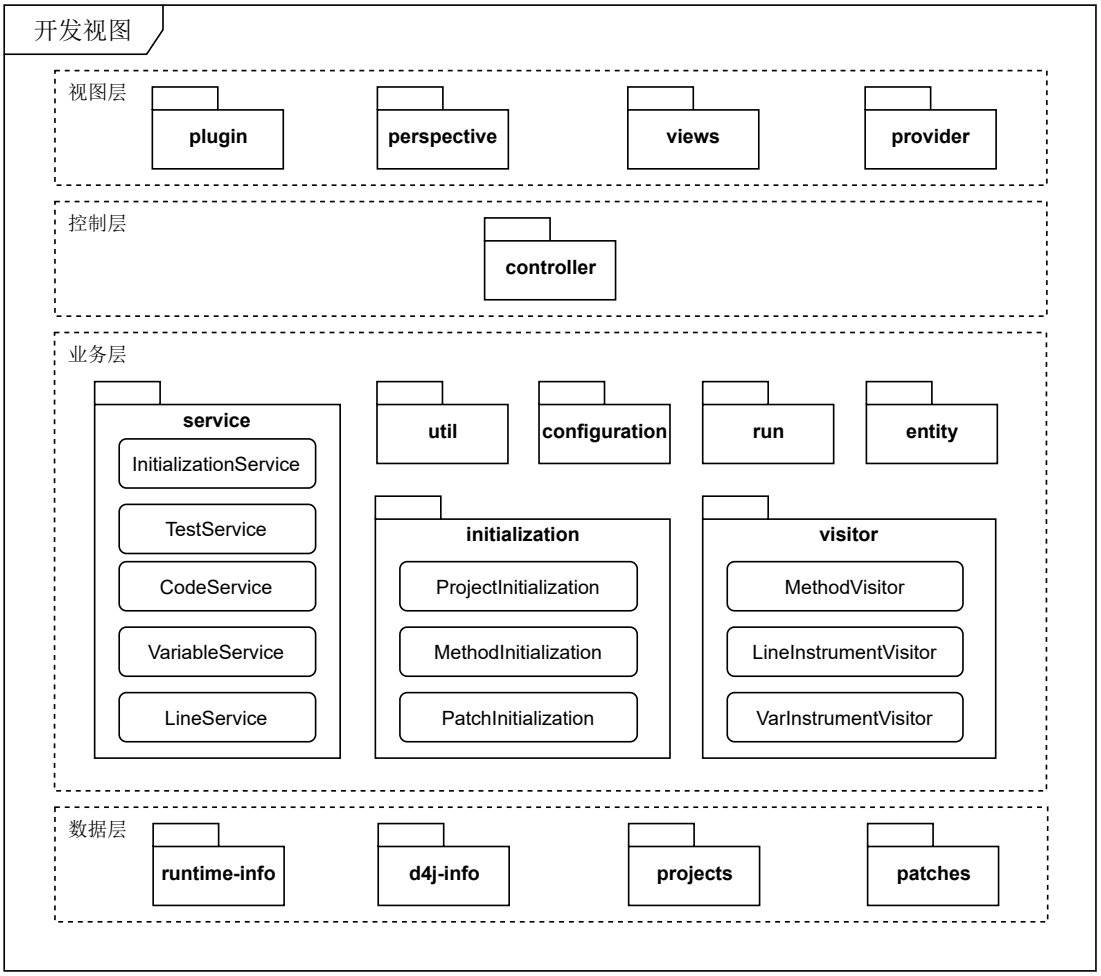

图 3.7: 开发视图

通用功能。数据层为系统的运转提供数据支持，分类存储前文提到的各种数据。

进程视图主要关注系统的非功能性需求，从设计人员的视角进行描述。侧重于功能在系统中的运行流程，描述不同功能间同步、通信等问题。根据面向对象的设计方法，我们通过 UML 系统顺序图进行描述，本系统的进程视图如图 3.8 所示。在本系统的进程视图中，首先需要对待修复的缺陷项目和及其候选补丁进行数据分析，通过数据服务收集分析结果并在后续流程中使用。在交互阶段，开发者进入系统开始审核流程，选择审核的补丁，主服务处理交互信息并展示；接下来为补丁扩增测试用例，主服务调用测试服务实现具体功能；然后开发者结合交互信息，修改测试预言；最后执行项目测试，主服务再次调用测试服务实现为新增测试用例的执行。

物理视图主要关注软件与硬件之间的映射问题，从运维人员的视角对系统物理节点件的通信与部署情况进行描述，侧重考虑系统的性能要求。根据面向对

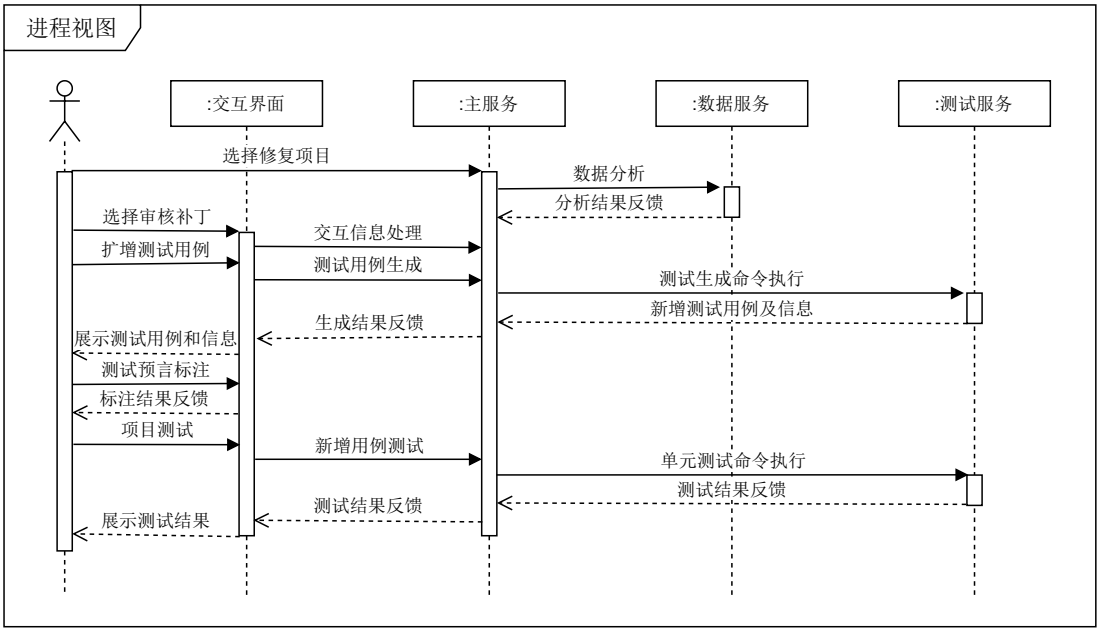

图 3.8: 进程视图

象的设计方法，我们通过 UML 部署图进行描述，本系统的物理视图如图 3.9 所示。在本系统的物理视图中，整个系统一共分为三个节点，用户访问的交互界面位于 Client 中，系统的主要实现模块位于 Server 中，运行在所需的数据和文件则放在 DBServer 中。

3.3.3　数据实体设计

为了保证数据的基础特性，以及使得数据能够在不同修复任务中得到复用，这些数据体量庞大且几乎不会变动，因此使用 MySQL 数据库进行持久化，包括项目信息、补丁信息和方法信息。每张表格描述了该实体的主要字段、在 Java 中的类型、在 MySQL 中的类型，以及相应的描述。

项目信息（Project）的持久化设计如表 3.9 所示。项目信息是本系统最基础的实体，主要存储修复项目的基础信息，包括项目名称和在 Defects4J 中的编号、源文件目录和编译目录、测试文件目录和编译目录、运行所需的依赖项和补丁列表，项目信息在数据分析、测试生成、交互展示等各个阶段都需要使用。

补丁信息（Patch）的持久化设计如表 3.10 所示。补丁信息主要存储其对应的项目、文件地址、补丁名称、变更文件、修改的具体位置、修改的内容，其中修改的内容分为三类且存储内容均为代码，考虑到有些代码长度较大，因此采用 TEXT 字段存储。

方法信息（MethodRange）的持久化设计如表 3.11所示。方法信息存储了方

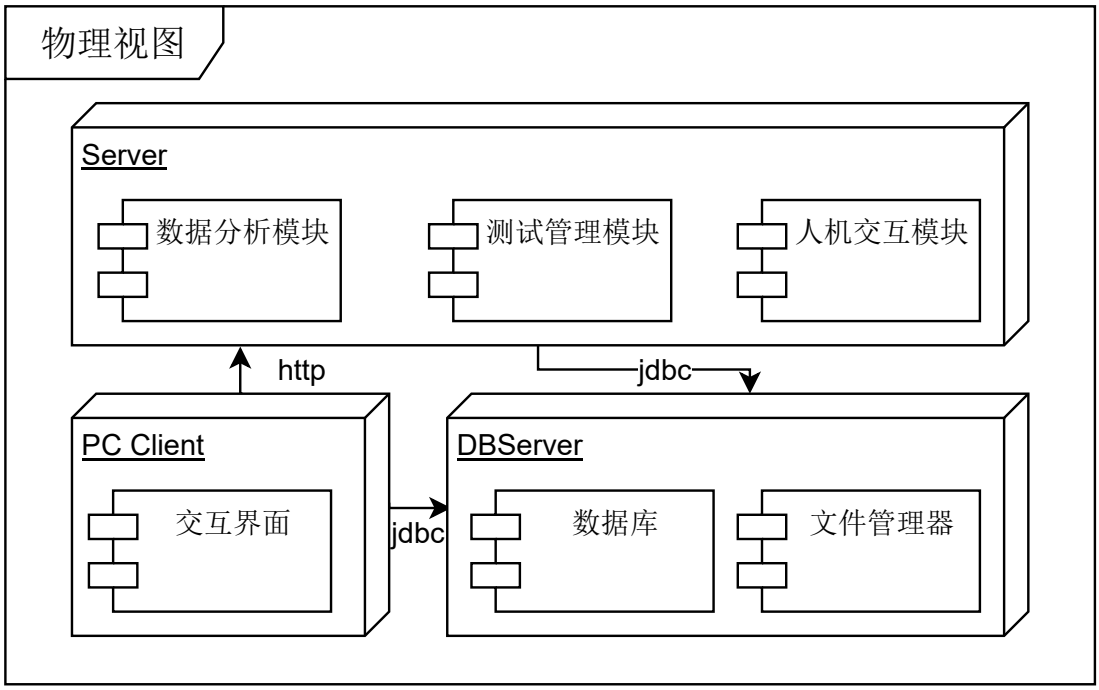

图 3.9: 物理视图

法的名称、方法起始行号和对应项目，主要用于故障方法的定位和使用。方法信息中只存储了起始行号，并没有存储具体代码，是因为并不确定方法使用的具体时间和地方，因此通过起始行号在需要使用某一方法时将其从类中切分出来。

```
{
    ......
    "org.apache.commons.math.stat.descriptive.moment.Variance#?#Variance#?,boolean": {
        "key": 124,
        "value": 127
    },
    ......
}
```

图 3.10: 方法信息 JSON 格式

方法信息中方法名称是一个经过格式化后的字符串，被“#”分为类名、返回类型、方法名和参数类型四部分，返回类型和参数类型如果为空，则用“?”代替。从数据库中读取出来后，将其转化成 JSON 对象在系统中使用，图 3.10 展示了某方法的 JSON 格式。

表 3.9: 项目信息持久化设计

字段	类型（Java）	类型（MySQL）	描述
projectid	-	INT	项目信息在数据库中的唯一标识
_name	String	VARCHAR	项目名称
_id	int	INT	项目 Bug 编号
_ssrc	String	VARCHAR	源文件目录
_sbin	String	VARCHAR	源文件编译目录
_tsrc	String	VARCHAR	测试文件目录
_tbin	String	VARCHAR	测试文件编译目录
_dependency	String	VARCHAR	运行所需的依赖项
patchList	List<Patch>	-	补丁列表

表 3.10: 补丁信息持久化设计

字段	类型（Java）	类型（MySQL）	描述
patchid	-	INT	补丁信息在数据库中的唯一标识
project	Project	INT	对应的项目，数据库中存储项目标识
patchPath	String	VARCHAR	补丁文件地址
patchName	String	VARCHAR	补丁名称
fixedFile	String	VARCHAR	被修改的文件
startLine	int	INT	补丁内容对应源文件的开始行号
modifyLine	int	INT	补丁修改的内容所在的行号
patchContent	String	TEXT	补丁内容
buggyContent	String	TEXT	有错误的内容
fixedContent	String	TEXT	修改过的内容

3.4　系统模块划分

前文已从需求分析和总体设计详述了本系统的整体功能和体系，按照模块化设计的思想，本系统依照具体的业务需求对服务进行了划分，包括数据分析模块、测试管理模块和人机交互模块，系统的模块划分如图 3.11 所示。数据分析模块对缺陷项目、补丁、配套的测试用例进行分析和数据封装，从代码中抽取隐含信息，为后续流程做好准备工作；测试管理模块负责系统中各种测试功能的执行，包括为具体的类扩增测试数据，完成交互工作中机器的部分，以及在开发者完成相应任务后响应测试，并反馈测试结果；人机交互模块处理并展示交互信息，引导开发者参与测试用例的构建，对机器生成的测试用例进行预言标注，并根据反馈的测试结果审核补丁的正确性。系统中各个模块相互协同，共同完成本系统主要功能的实施。

数据分析模块。该模块实现对项目、补丁和测试用例的分析，包括动态分析

表 3.11: 方法信息持久化设计

字段	类型（Java）	类型（MySQL）	描述
methodid	-	INT	方法信息在数据库中的唯一标识
methodName	String	VARCHAR	方法名称
startLine	int	INT	开始行号
endLine	int	INT	结束行号
project	Project	INT	对应的项目，数据库中存储项目

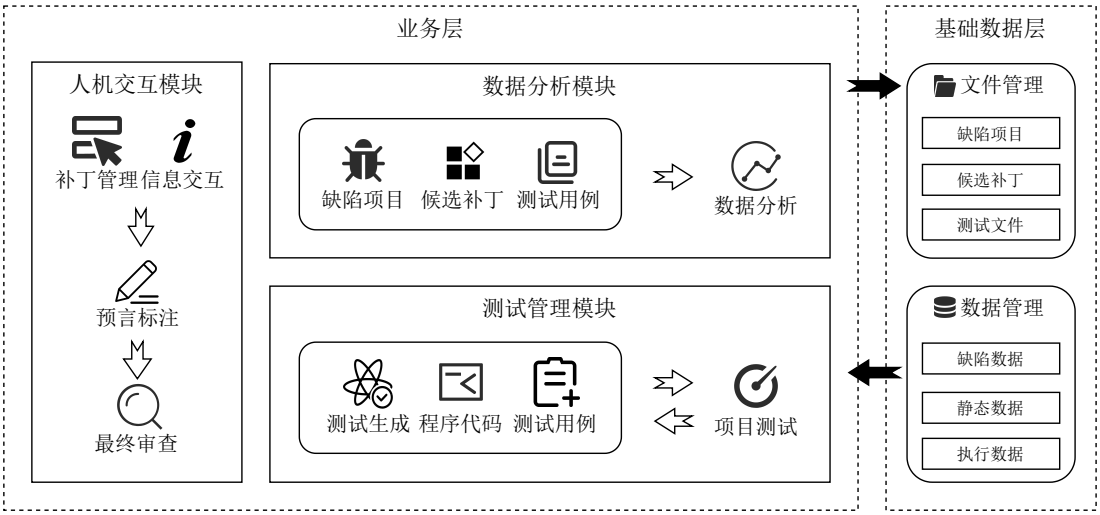

图 3.11: 系统模块划分

和静态分析。静态分析将这些实体封装成抽象的对象，并为他们之间建立联系；动态分析抽取代码的变量信息和执行信息，并把这些信息存储下来由基础数据层统一管理。一方面这些信息将在其它模块的实现中进行复用，另一方面还能作为辅助信息供开发者参考。

测试管理模块。该模块对系统中所有的测试功能进行统一管理。核心部分功能是开发者选择对某个补丁进行审核流程时，该模块根据之前准备好的数据，定位到该补丁对项目做出变更的具体位置，替换上补丁的代码，针对修改后的项目扩增测试数据；当开发者在人机交互模块完成指定的任务后，该模块为整个项目运行测试，并向开发者反馈测试结果。除此之外，在数据分析、单元测试等需要用到测试的地方，都需要测试管理模块提供支持。

人机交互模块。该模块的主要工作有两方面，一是处理需要展示的交互信息，二是引导开发者进行补丁审核。开发者选择审核的补丁，针对测试管理模块生成的测试用例，结合数据管理模块提供的交互信息，开发者为测试用例确定预言，提高测试用例的充分性，用以验证补丁的正确性。之后，根据测试管理模

块反馈的测试结果，开发者做进一步审核，决定删除补丁、或者继续审核、或者选择其作为最终答案。

3.5　本章小结

本章完成了对面向自动程序修复的人机交互式补丁审核系统的需求分析和概要设计，详细描述面向自动程序修复的人机交互式补丁审核系统的需求和总体设计。通过对程序修复领域中的两个问题进行分析，找到本文技术的切入点，以自动程序修复工具生成的补丁作为输入，完成交互式修复技术的设计和实现，并构建人机交互的图形界面。首先，简要概述本系统的工作流程；然后，分析系统的功能需求和非功能需求，并通过系统用例对需要实现的功能进行详细描述；接着，介绍了系统的总体设计，遵循“4+1 视图“的软件开发方法，并对系统进行持久化设计，从五个视角分析系统的架构设计；最后，按照业务功能对系统进行了模块划分，各个模块的具体实现将在下一章介绍。

第四章　系统详细设计与实现

4.1　数据分析模块

4.1.1　详细设计

对程序、补丁、测试文件等原始数据的分析是面向自动程序修复的人机交互式补丁审核系统的首要步骤，这也是数据分析模块的主要工作，数据分析可以进一步细分为静态数据分析和动态数据分析。静态数据分析无需运行程序，通过解析代码为各个实体封装对象和建立模型。通过对相关数据进行静态分析，我们将缺陷项目和补丁封装成对象，获取变更文件中所有方法的详细信息，找到补丁对项目的更改位置。静态数据分析流程图如图 4.1 所示。

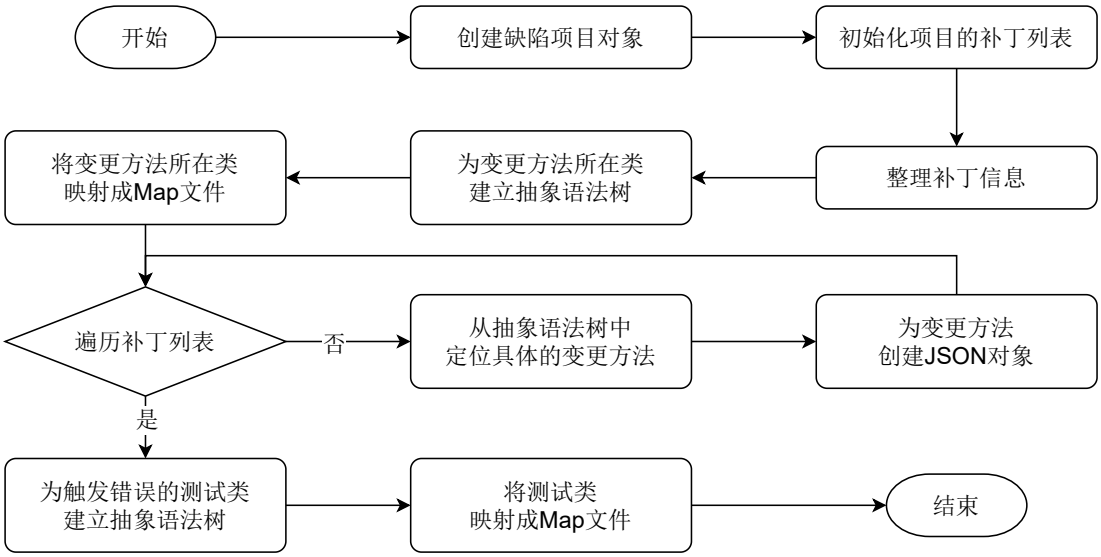

图 4.1: 静态数据分析流程图

首先，初始化缺陷项目，将项目抽象成一个项目对象，指定源程序文件路径和编译输出目录，设置项目依赖，这是本文最基础的实体类，该类的持久化设计已在上一章中介绍。然后对补丁文件展开分析，读取补丁文件列表，针对每一个补丁文件，分析文件的内容。通过文件中的内容，整理补丁对故障方法所做的修改。根据补丁的存储格式，我们按照规则对文件内容进行分解即可得到我们想要的信息。通过文件解析，我们可以得到的内容包括故障类名及地址、修改行号、具体修改的内容，我们将这些信息整理封装成补丁对象，形成一个补

丁列表存入项目对象。最后是方法分析，为补丁和更改方法建立映射关系，由于补丁中仅申明了更改方法所在的类，并没有指明具体方法，因此在该步骤中，使用抽象语法树解析每个变更方法所在的类并存储。此外，我们针对触发错误的测试类做同样的处理，方便在后续的阶段能够快速定位到具体的测试用例。经过静态数据分析的过程，我们可以得到指定缺陷项目和补丁集合的对象，以及变更文件中方法的 JSON 数组，同时为各个对象建立关联关系，将分散的实体联系起来。

动态数据分析通过运行程序代码，复现错误被触发的场景，帮助开发者更直观的认识到缺陷项目应用补丁前后的异同。一般来说，程序在运行时会展现出多种代码特征，我们需要的是能让开发者快速理解程序、快速进入交互状态的特征。因此通过对相关数据进行动态分析，我们记录下补丁应用前后变更方法中的“参数变化”和“执行轨迹”两类代码动态特征。动态数据分析流程图如图 4.2 所示。

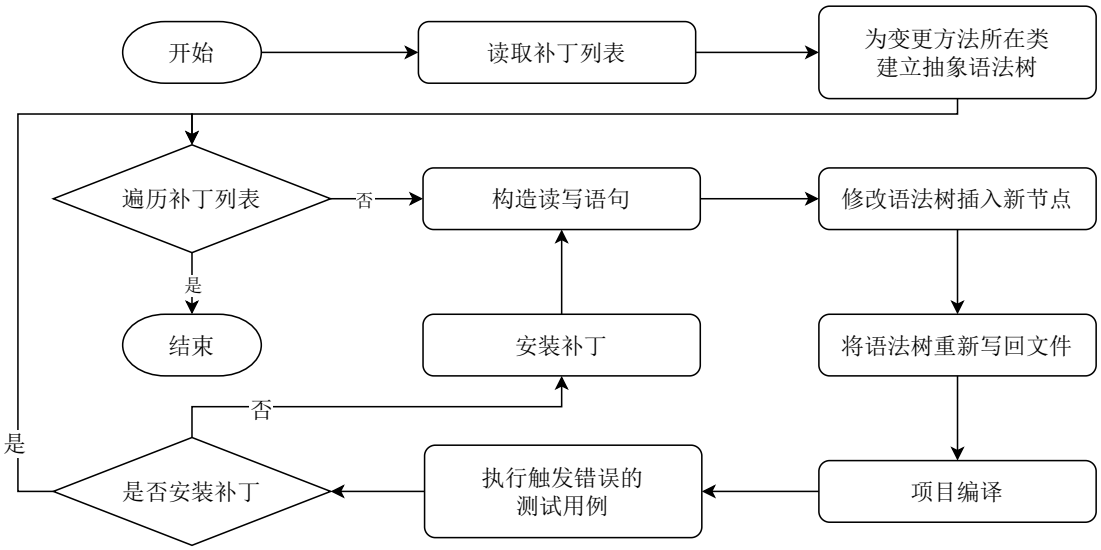

图 4.2: 动态数据分析流程图

参数变化。当一个方法中出现了修改，最显著的变化就体现在变量上面。借助黑盒测试的思想，我们此时甚至可以完全不关心方法内部的具体执行，而只关注数据经过该方法前后的变化，即该方法在功能上是否正确。因此，我们需要收集该方法参数的变化。对于任何一个方法而言，入口处、出口处以及返回值（包括抛出的异常）是整个方法最重要的几个节点。我们在紧接着方法流程开始的位置和结束之前的位置插入 write() 方法，记录数据进入该方法和离开该方法时的即时值。

执行轨迹。补丁的修改不单单在宏观上造成影响，在方法内部同样会出现变化，变化之一就是会影响数据在方法内的走向。借助白盒测试和路径覆盖等概念，为了复现补丁的改动致使方法内部执行轨迹的变化，我们在方法中每一行语句后插入 write() 方法，记录该语句的行号，如果执行到该语句，那么则将行号记录下来，从而形成一条执行的轨迹。

动态数据分析流程大致可分为三个部分，方法插桩、编译运行、信息收集。方法插桩是对所有变更的方法插入收集语句，使得我们能够获得方法中的运行数据，掌握变化的动态。该步骤首先将预先写好的 Dumper 类拷贝到项目中，随同项目在下一步中一起编译，从而发挥作用。Dumper 类定义了若干 write() 方法，支持将不同类型的变量快速写入指定的文件，同时对变量和值进行格式化，方便存取以及与开发者的交互。然后将故障方法转化为抽象语法树，定位到树的指定节点，通过对语法树实施插入操作，在不干扰方法正常执行的前提下，将 write() 方法插入到树中的指定位置。在完成对指定方法的插桩之后，需要运行程序使得这些语句发挥效力。因此我们在 Defects4J 的环境下编译项目，执行触发错误的测试用例，这样相关数据就会按照前置步骤中的设置存储到指定地方，完成信息的收集。

4.1.2　核心类图

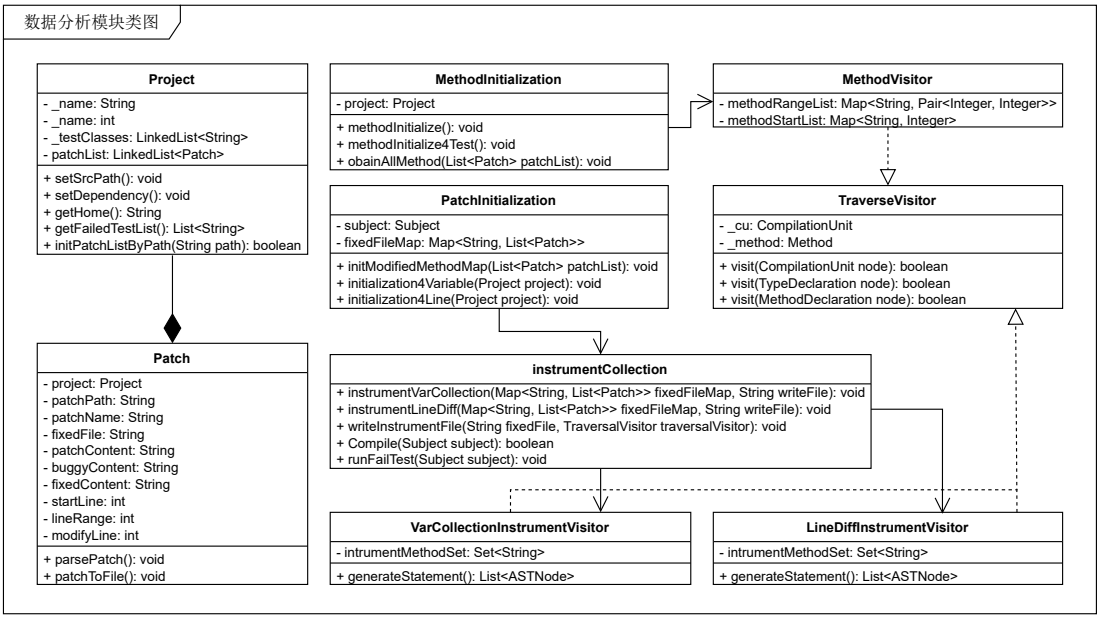

图 4.3: 数据分析模块核心类图

数据分析模块的类图如图 4.3 所示，图中包含了数据分析过程中用到的核心类。Project 和 Patch 是最重要的两个实体类，分别描述缺陷项目和补丁的具体信

息，其中 Project 中通过列表的形式聚合 Patch。数据分析模块的工作一开始就是初始化这两个实体，完成第 3.3.1 节中对缺陷项目和补丁的初始化。MethodInitialization 将项目中源代码和测试代码的各方法解析成特定格式的信息存储，它调用 MethodVisitor 对项目中与缺陷相关的每一个类文件进行分析。PatchInitialization 主要实现数据的动态分析功能，它调用 InstrumentCollection 分别收集整理补丁应用前后参数变化和执行轨迹的信息，使用 VarCollectionInstrumentVisitor 和 LineCollectionInstrumentVisitor 对源代码进行插桩从而收集信息。MethodVisitor、VarCollectionInstrumentVisitor 和 LineCollectionInstrumentVisitor 均继承自抽象类 TraverseVisitor，而 TraverseVisitor 则继承自 Eclipse AST 中的 ASTVisitor，通过将类封装成 CompilationUnit，进而转换成抽象语法树，实现对该类的访问和修改。

4.1.3　顺序图

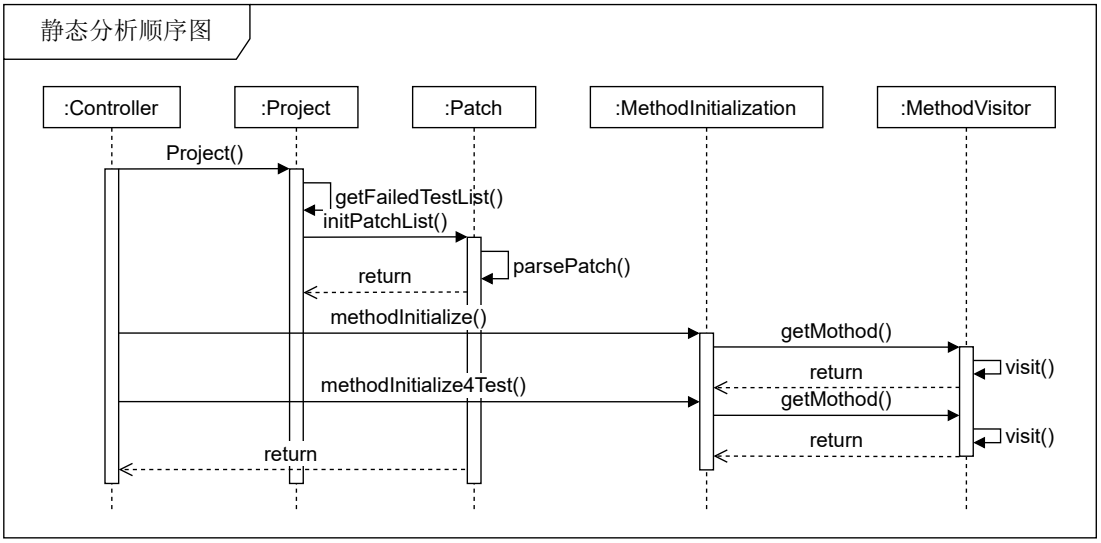

图 4.4: 静态数据分析顺序图

数据分析模块中静态数据分析的顺序图如图 4.4 所示。本部分的入口是控制类 Controller，首先为缺陷项目和补丁创建对象，然后调用 MethodInitialization 类解析源代码中的方法，将每个类封装成 CompilationUnit 后传给 MethodVisitor 建立抽象语法树，再通过 visit() 方法读取语法树中的信息。利用同样的方法，对项目中的测试代码做同样的解析，不再赘述。

数据分析模块中动态数据分析的顺序图如图 4.5 所示。在上一部分工作完成后，由 PatchInitialization 负责收集补丁应用前后参数变化和执行轨迹的信息。首先进行插桩，将变更文件对应的类封装成 CompilationUnit 后传给 CollectionInstrumentVisitor 建立抽象语法树，根据不同方法内部的实际情况构造收集语句，

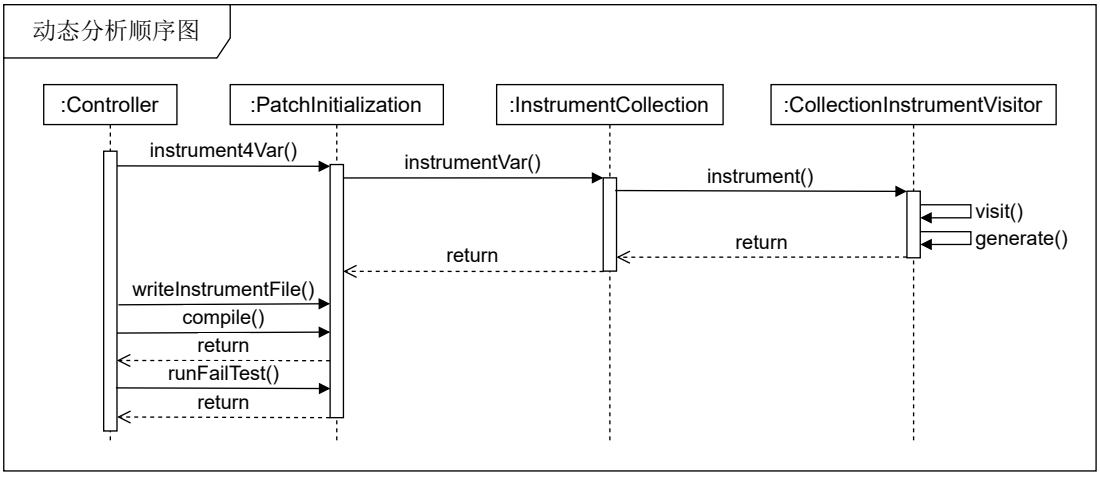

图 4.5: 动态数据分析顺序图

通过对语法树节点的修改加入信息采集的代码段，然后编译项目并执行触发错误的测试用例，把补丁应用前后的变化记录下来。参数变化和执行轨迹信息收集的顺序图基本一致，这里用一个类代替；不同地方在于前者需要精确定位每个方法的出入口，而后者只需在每个语句前添加采集代码。

4.1.4　关键代码

数据分析模块的关键代码如图 4.6所示，涉及到静态数据分析和动态数据分析两部分逻辑，分析过程为遍历每个补丁文件，通过建立抽象语法树提取指定代码特征。静态数据分析的结果按照第 3.3.3 节中数据实体设计存储进数据库，动态数据分析对比补丁应用前后收集第 4.1.1 节所述的两类代码特征。

4.2　测试管理模块

4.2.1　详细设计

测试管理模块为整个系统统一处理和提供与测试相关的功能，其中最核心的功能是测试生成和项目测试。测试生成流程图如图 4.7 所示。鉴于候选补丁的数量较多，将所有未经处理的补丁直接交给开发者去审核是不合适的。在第三章中，我们分析发现由于测试用例的不充分性导致了修复工具正确率较低，难以准确判断补丁的正确性。针对测试用例的不充分性，本节我们通过利用自动化测试工具扩增测试用例的方式来解决数量上的问题。对于每个候选补丁，首先完成一些前置工作，将补丁安装到指定位置并编译；然后对于该补丁，确定其涉及的变更文件列表，遍历变更文件的列表，测试用例生成工具会将该文件作为输入，依次为每个类生成测试用例；如果成功生成则为测试用例及相关信息

```
// init methods
for (Patch patchFile : patchList) {
    String fixedFile = patchFile.getFixedFile();
    String methodName = patchFile.getModifiedMethod();
    Method method = new Method(methodName, fixedFile);
    MethodVisitor methodVisitor = new MethodVisitor();
    methodVisitor.setMethod(method);
    CompilationUnit compilationUnit =
FileIO.genASTFromSource(FileIO.readFileToString(fixedFile), ASTParser.K_COMPILATION_UNIT);
    compilationUnit.accept(methodVisitor);
    method = methodVisitor.getMethod();
    ......
    Map<String, Pair<Integer, Integer>> methodRangeMap = methodVisitor.getMethodRange();
}

// init patches
for (Patch patchFile : patchFileList) {
    InfoCollectionService infoCollectionService = new InfoCollectionService();
    infoCollectionService.instrument(patchFile, writeFile);
    if (infoCollectionService.Compile(subject)) {
        infoCollectionService.runFailTest(subject);
    }
    if (patchFile.patchToFile()) {
        infoCollectionService.instrument(patchFile, writeFile);
        if (infoCollectionService.Compile(subject)) {
            infoCollectionService.runFailTest(subject);
         }
    }
    ......
}
```

图 4.6: 数据分析模块关键代码

进行分析，当遍历完全部变更文件后，反馈测试生成的结果，结束这一流程。

安装补丁。安装补丁是生成测试用例的必要条件。根据第二章对测试生成技术的介绍，EvoSuite 工具是一个基于字节码的测试生成工具，即源代码首先要编译生成中间码，才能使用 EvoSuite。因此需要在安装补丁并编译完成之后，才能针对指定的类生成测试用例。安装补丁需要经历如下两个步骤：内容替换，根据在准备阶段完成的对项目和补丁的解析，可以轻松定位到与补丁相关联的更改文件，通过 patch 指令利用设置修补文件的方式实现修改、更新原始文件，将 PatchFile 类中的 fixedContent 替换到文件中 buggyContent 的位置；项目编译，移除旧的 target 目录，执行编译命令，为项目生成新的 Class 文件，即为 EvoSuite 工具准备运行所需的必要文件。

测试生成。将补丁安装到指定位置并完成编译后，就是为指定的类生成测试用例。对于补丁涉及到的每个更改文件，自动化测试工具会将其作为输入，生成一组测试用例，然后运用在数据分析阶段的方法将测试文件解析成 json 格式来存储和检索。同时，还会产生一份测试报告，经过分析，我们认为其中的信息能够作为与开发者交互时辅助信息的一部分。最后，综合整理在上一阶段分

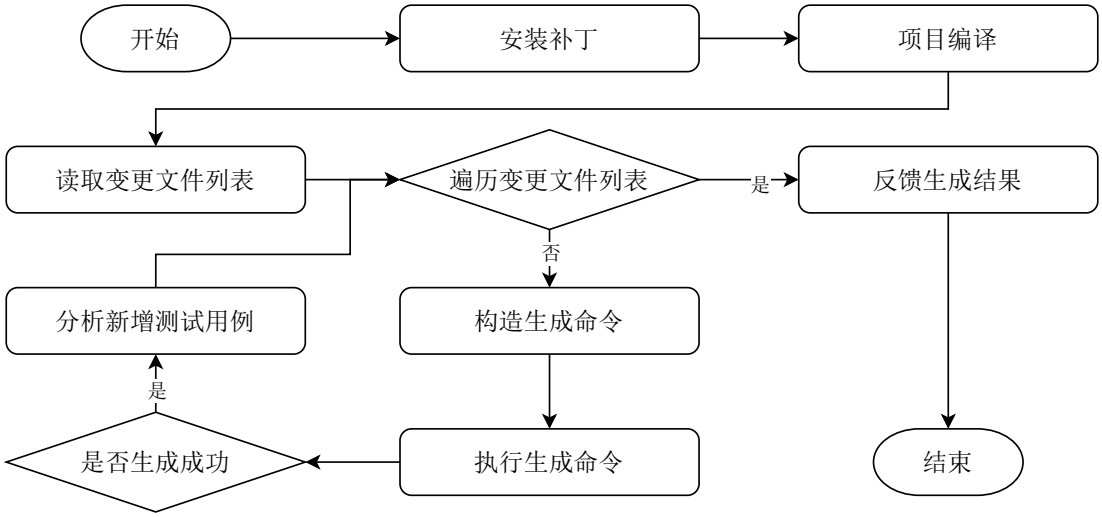

图 4.7: 测试生成流程图

析的相关信息，以及由测试工具产生的测试报告等信息，在这一步中进行集体封装，作为辅助信息通过图形界面展示给开发者，供开发者在后续流程中参考，从而完成测试用例的构造。

表 4.1: EvoSuite 参数配置

参数	**取值**	**说明**
criterion	branch line weakmutation	该参数表示 EvoSuite 在生成测试用例的过程中会尽量满足的覆盖准则。本系统选择 branch、line 和 weakmutation，使得测试用例和覆盖数据更全面和快速。
Dsearch_budget	20	该参数表示 EvoSuite 生成算法的搜索时间。本系统设置为 20s，过长极有可能影响用户体验，且经简单验证 20s 足够使得综合覆盖率达到 60% 以上。
Djunit_check	false	该参数表示 EvoSuite 是否进行 JUnit 格式检查。本系统设置为 false，即不检查。因为后续这些测试用例还需要交给开发者进行标注。
Dtarget_method	动态生成	该参数表示 EvoSuite 的目标方法。这个参数根据项目、补丁会动态配置，出于对效率的考虑，不会对全部方法生成测试用例。

参数配置。本文使用 EvoSuite 为基础来生成针对变更方法的测试用例，根据 Java 字节码 [67] 转换类型描述，对 EvoSuite 进行适当的改造和配置，将其用于对指定方法扩展测试数据。经过实验验证，综合考虑多种因素，系统对表 4.1 所示的参数进行了适合于本系统的设置。考虑到系统的效率以及交互的友好，覆盖准则仅考虑分支覆盖和行覆盖，并限制数据搜索和格式检查的时间。

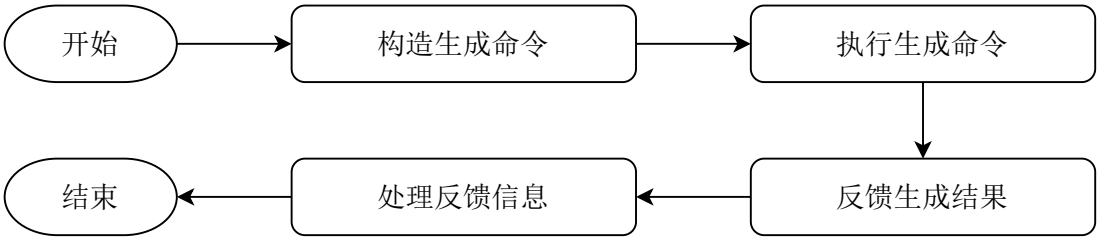

图 4.8: 项目测试流程图

项目测试的逻辑较为简单，通过对 JUnit 的配置，从而根据不同的测试需求，实现个性化测试，并反馈测试结果。项目测试的流程图如图 4.8 所示。

4.2.2　核心类图

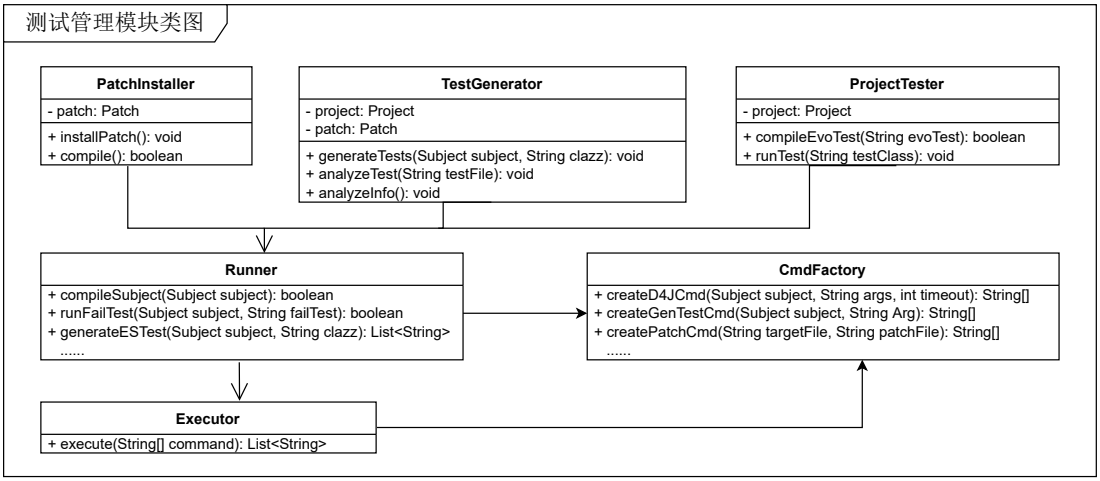

图 4.9: 测试管理模块核心类图

测试管理模块的类图如图 4.9 所示。该模块的入口 Service 类已经省去，PatchInstaller、TestGenerator 和 ProjectTester 是这个模块的核心类，分别实现了补丁安装、测试生成和测试执行。它们具体的运行都需要调用 Runner 获取命令行指令。在指令调用实现上采用了命令模式，CmdFactory 创建具体的命令，Executor 具体实施和执行每一个请求，Runner 定义了两者之间的弱耦合，负责调用 Executor 的相应操作。在具体操作中，例如在执行安装补丁命令时，仅有 Runner 对 PatchInstaller 开放，其在 Runner 中调用安装补丁的方法，Runner 使用 CmdFactory 构造完成补丁安装所需的具体命令语句，并通过 Executor 来执行。这样的设计使得指令调用更灵活，也方便新的命令添加进来。

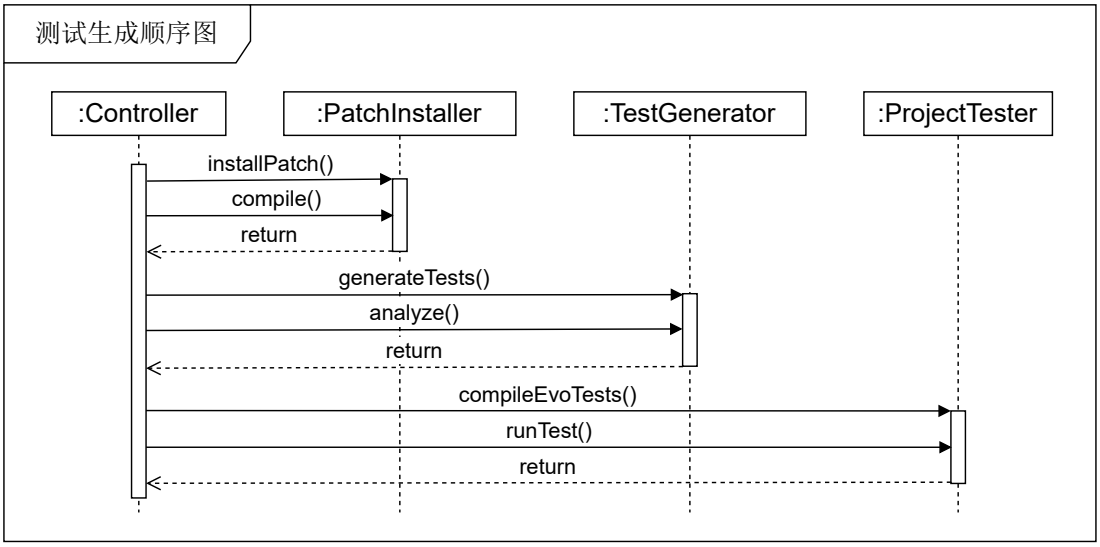

图 4.10: 测试生成顺序图

4.2.3　顺序图

测试管理模块中核心功能测试生成的顺序图如图 4.10 所示，为了展示核心功能，指令调用的部分在此省略。本部分的入口是控制类 Controller，EvoSuite 需要在字节码的基础上生成测试用例，因此首先调用 PatchInstaller 安装补丁并编译，然后调用 TestGenerator 生成测试用例和并执行文件分析，这里分析与数据分析模块对测试文件的分析所采用的方法是一样的，最后编译新的测试用例、运行并反馈结果。本部分的运行原理已在 4.2.1 节详细介绍，这里不再赘述。

4.2.4　关键代码

```
......
String clazz = fixedFile.split(subject.getSsrc())[1].split("\\.")[0].replace("/",
".").substring(1);
generateTests(subject, clazz);
......
String genArg = Constant.JAVA_HOME + "/bin/java -jar " + Constant.evosuiteJarPath + " -
class " + clazz + " -projectCP " + subject.getSbin().substring(1);
message = Executor.execute(CmdFactory.createGenTestCmd(subject, genArg));
......
ProcessBuilder builder = new ProcessBuilder(command);
Builder.redirectErrorStream(true);
process = builder.start();
......
```

图 4.11: 测试管理模块关键代码

测试管理模块的关键代码如图 4.11 所示。图示代码实现了测试生成的功能，

分为三个部分，第一部分是主要业务逻辑，即实际使用这一功能的代码；第二部分是命令工厂，通过功能需要以及传入参数组合构造不同的终端命令；第三部分是命令执行器，负责实际的命令执行并记录执行信息。

4.3　人机交互模块

4.3.1　详细设计

1）人机交互流程

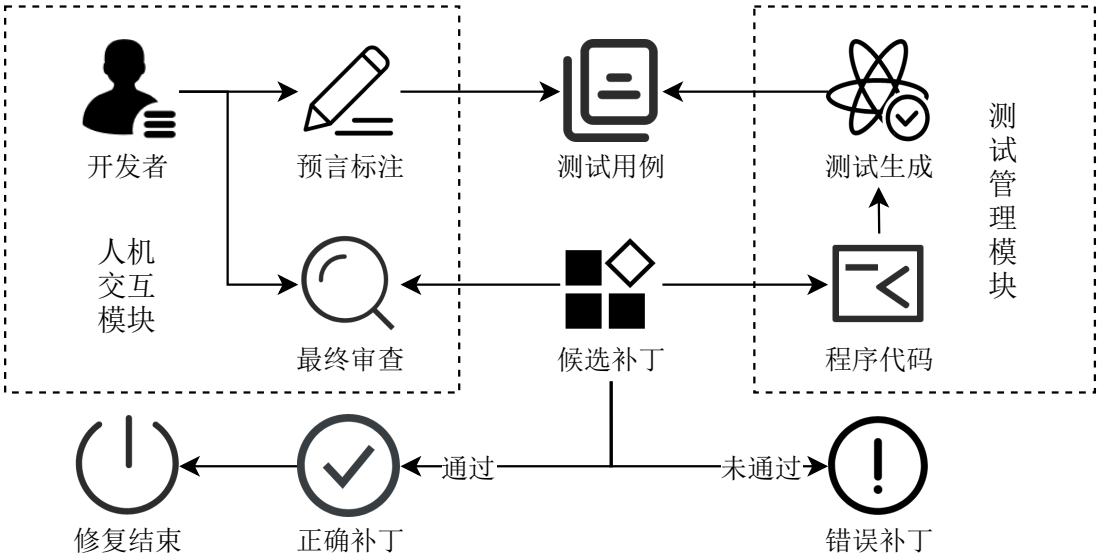

图 4.12: 人机交互模块与测试管理模块的联系

人机交互模块管理着补丁交互式审核过程的具体逻辑，这是人机交互式补丁审核方法的重要阶段，该阶段是开发者与测试工具共同构建测试用例并完成补丁筛选的过程，这一包括多个交互动作，我们会在后文中详细介绍其中的具体细节。该阶段的输入是正在修复中的完整项目，首先通过测试生成技术，针对更改过的方法生成一组测试用例；接着开发者对测试用例标注测试预言，共同完成测试用例的构建；使用扩增后的测试数据运行项目测试，以过滤掉一些看起来不合理的补丁。对于无法通过测试的补丁，则被标记为不正确，从候选补丁列表中移除；而通过测试的补丁，开发者可以选择反复执行上述的步骤，迭代的运行测试并审核补丁的正确性，直到找到正确的补丁。我们将以上描述的过程称作补丁的交互式审核过程，由开发者和测试工具进行交互，从而完成补丁筛选的目标。从系统层次的角度来看，该模块覆盖到了视图层和控制层；从模块划分的角度来看，该模块与测试管理模块有着密不可分的联系，如图 4.12 所示。

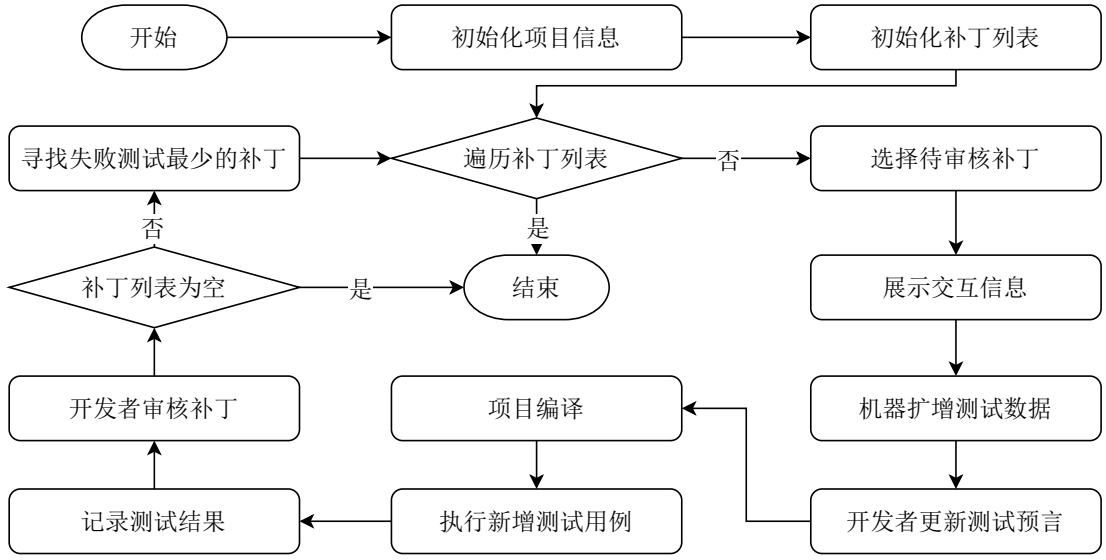

图 4.13: 人机交互模块流程图

人机交互模块流程图如图 4.13 所示，详细描述了人机交互模块的功能和交互过程。该流程的输入是一个缺陷程序和它的配套测试用例，以及一组针对缺陷程序提出的候选补丁；该流程的输出是针对缺陷程序的正确补丁。如果针对该缺陷程序没有对应的候选补丁，那么就直接返回一个空集合。否则的话，遍历候选补丁列表，开发者选择待审核补丁，查看其交互信息。由机器生成一组新的测试用例，并将其中的每一个测试用例交给开发者更新测试预言。然后，我们针对该补丁，编译并执行新增的测试用例，记录执行失败的用例个数。接下来，开发者根据交互信息和测试结果审核补丁，决定移除补丁或让其继续留在候选补丁列表中，亦或是为该补丁重复测试生成的过程。最后，如果候选补丁列表中还遗留有补丁，则会选择失败用例个数最少的补丁作为所需补丁并返回。总的来说，面向自动程序修复的人机交互式补丁审核系统希望利用开发者的主观判断，通过对失败测试和交互信息的分析，从而为该补丁做出选择。为此，本系统提供了四种选项。

- 删除：开发者认为该补丁是不正确的，将其移出候选补丁列表。
- 指定：开发者认为该补丁是正确的，将其指定为最终结果，结束任务。
- 重新审核：为当前补丁执行新一轮的交互过程，重新生成测试用例和标注预言，迭代该过程直到完成。
- 跳过：审核下一个候选补丁，跳过当前补丁，交给算法进行选择。

2）测试预言标注

由于自动生成的测试用例存在机器难以避免的问题，我们无法将测试生成工具自动生成的测试用例直接用来测试项目。因此，面向自动程序修复的人机交互式补丁审核系统要处理的核心问题就是测试预言问题，通过采用人工的方法标注预言，提高测试用例充分性，从而实现构造能够区分补丁正误的测试用例。

在测试管理模块中，使用 EvoSuite 工具生成的测试用例都是带有断言（Assertion）的。断言是一个布尔表达式，放置在程序中的某个点用来在运行时检查该点的行为。对于此次执行，当断言评估为真时，程序的行为在该断言处被视为“符合预期”；当断言评估为假时，则在程序中发现该点处发现错误。因此，可以将断言看作是测试预言的一种。我们针对由 EvoSuite 工具生成的测试用例，希望通过开发者的参与旨在完善原有的断言从而获得一个新的断言，以便新的断言为真时程序的状态恰好是正确的。我们用一张图来描述预言的质量。如图 4.14 左侧所示，程序某处的正确状态为区域 G，称之为基本事实；而目前测试预言所能覆盖到的状态是区域 O。假设我们使用断言作为此处的测试预言，区域 1 是集合 $(O-G)$，断言评估为真，程序在该处是不正确的；区域 2 是集合 $(G-O)$，断言评估为假，程序在该处是正确的；区域 3 是集合 $(O' \cap G)$，断言评估为真且程序是正确的。只有区域 3 占全部的比例越大，预言的质量才会越高。如图 4.14 右侧所示，通过预言标注，将覆盖的范围从 A 变成 O'，尽可能地使得 $(O' \cap G)$ 的面积更大。

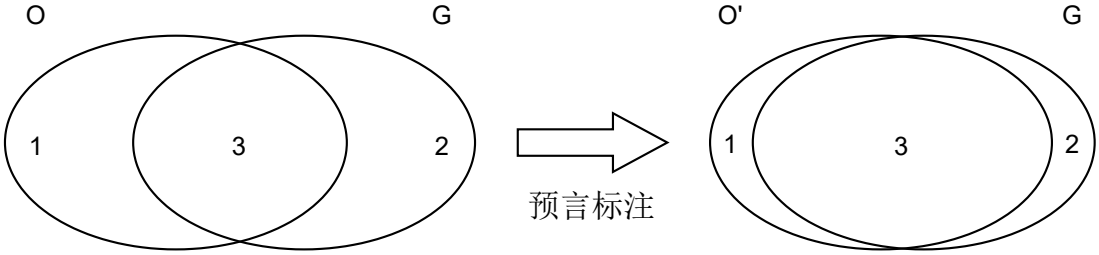

图 4.14: 测试预言标注示意图

理想情况下，开发者只需简单的点击即可修复错误，但是实际上是不太可能的，因为开发者需要理解程序的行为才能对测试用例做进一步的完善。尽管测试预言没有完全自动化，但是本文提出的方法还是从构建测试预言的方面减少了人工标注的成本。一般情况下，为了测试一个小的功能就需要大量的测试用例，这一繁琐且工作量巨大的编写工作就交给了测试生成工具。最重要的是，本文在人工参与时提供了交互信息，这些信息来自于数据分析模块对各种原始数据所做的动静态分析，包括程序缺陷的具体位置、触发错误的测试用例、补

丁应用前后参数变化和执行轨迹，以及自动生成测试用例过程中产生的新信息。本文为这些信息与预言标注措施之间建立联系，使得这些交互信息能为开发者在标注过程中提供指导和帮助。表 4.2 展示了三种交互信息、补丁做出的修改以及预言标注之间的联系。

表 4.2: 交互信息、补丁做出的修改以及预言标注之间的设计

交互信息	**补丁做出的修改**	**标注关注点**
执行轨迹	修改比较运算符	结构合理性
参数变化	增删改算术运算符或按位运算符 增删改常量或变量	功能正确性
新增测试覆盖率	-	测试充分性

执行轨迹。执行轨迹描述了程序在该方法内的运行路线，可以清晰的看出程序的数据流向，在一定条件下可以转换成程序流程图和数据流图，把程序看作是路径的集合。补丁如果修改比较运算符，会对运行路线造成很大影响。在方法的执行轨迹信息的帮助下，开发者应着重关注方法内的结构合理性，利用结构化的方法，我们可以把方法看作是路径的集合，于是对该方法的测试可以看作对方法中路径的测试。因此该属性的目标是设法让被测方法中每条路径都被执行到，使潜伏在补丁中的错误能够暴露出来。

参数变化。参数变化有两层含义，一是数据通过该方法前后的变化，二是补丁应用前后导致流经数据的变化。补丁如果对方法中的变量或运算进行增加或修改，参数变化会是显著的。结合方法的参数变化和黑盒测试的相关思想，开发者可以不关心方法的内部结构，通过对方法入口和出口值的掌控，设计测试预言验证该方法功能的正确性。同时，开发者可以将更多的注意力放在数据的选择上，通过划分等价类减少测试的数量实现更广的覆盖，或是通过分析边界值对入口和出口的边界数据进行测试，从而有针对性地对生成的测试用例标注预言，达到检查补丁是否真正实现修复的目的。

新增测试覆盖率。新增测试覆盖率描述了测试生成工具新增的测试所达到的覆盖数据，测试覆盖率是作为衡量测试是否充分的重要标准。一般来说，测试生成工具追求的是整体的覆盖目标，而不会特别关注其中某一项指标。开发者可以结合被测方法的特性，决定对该方法的关注重点，通过修改测试预言提高某一项或几项的覆盖率，以验证该补丁的修改是否正确。

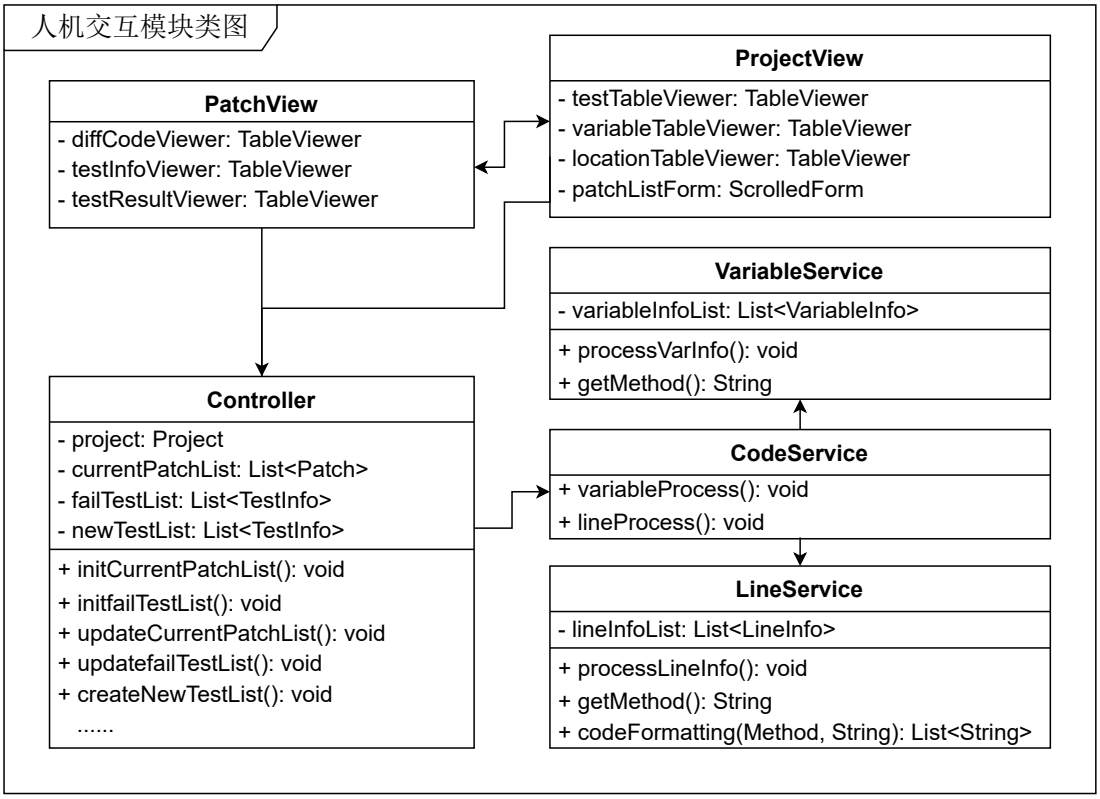

图 4.15: 人机交互模块核心类图

4.3.2　核心类图

人机交互模块的类图如图 4.15 所示，这里即体现了前后端分离的设计思想。ProjectView 和 PatchView 作为两个主要视图，是工具中用于展示信息以及与开发者交互的界面，其中设置不同的 viewer 分板块展示不同的信息，设置不同的按钮响应开发者不同的动作。两个视图在逻辑上是相互关联的，ProjectView 中的动作会触发 PatchView 中内容的刷新，使界面能够信息同步。当工具完成测试生成后，编辑器打开新的测试代码，交给开发者予以预言标注。Controller 是控制类，类中包括前端各个板块的初始化及更新操作，负责保持界面和模型的一致。在展示交互信息时，Controller 调用 CodeService 进行处理，CodeService 通过 VariableService 和 LineService 分别处理在数据分析模块收集的代码特征“参数变化”和“执行轨迹”，将两者解析成可读性较高的信息予以呈现。

4.3.3　顺序图

在人机交互模块中，补丁交互式审核过程中包含多个交互动作，对于每一个补丁的每一轮交互，我们首先用一个顺序图描述了开发者与修复技术交互的细节。如图 4.16 所示，一共产生了四次交互动作。第一次由开发者选择补丁，系

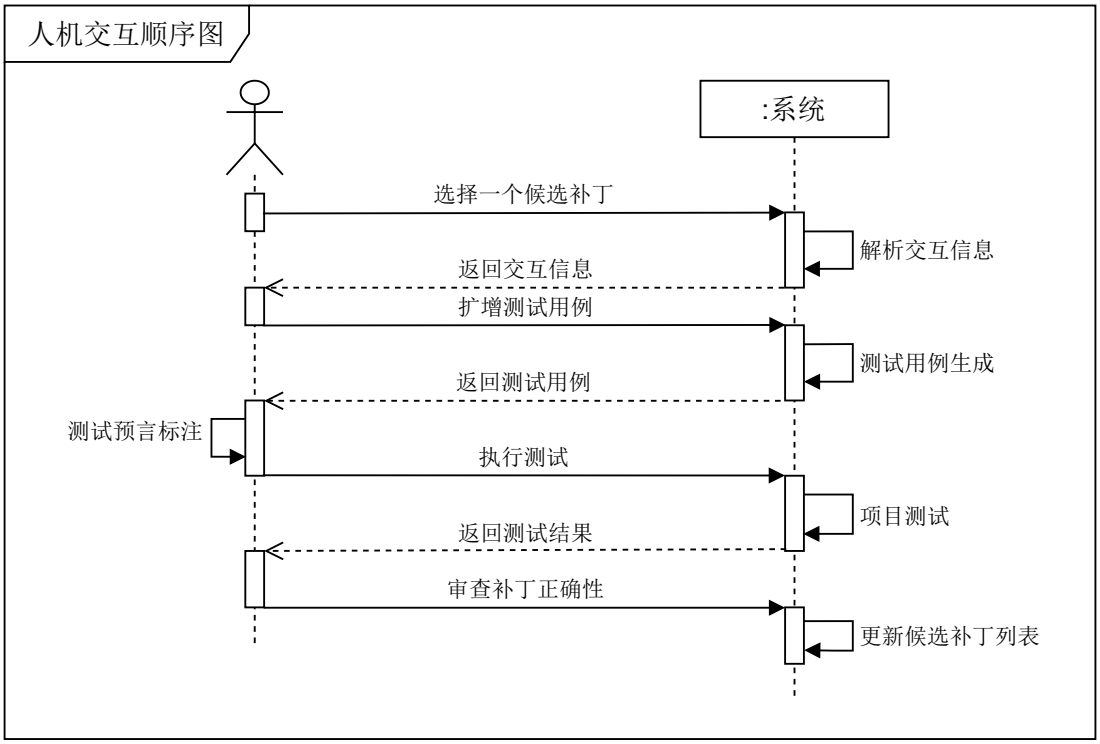

图 4.16: 人机交互顺序图

统解析交互信息并展示；第二次开发者向系统发起生成测试用例的请求，系统后台扩增测试数据并返回；第三次在开发者完成测试预言标注后，执行测试，系统响应开发者的命令并返回测试结果；第四次开发者审核补丁的正确性，决定该补丁的去留，系统更新候选补丁列表。最终，如果候选补丁列表中还遗留有补丁，系统会选择测试用例失败个数最少的补丁作为最终结果。由于缺陷项目最终只能应用一种解决方案，我们认为返回一个补丁是一个合理的选择，因此如果多个补丁具有相同数量的失败用例个数，则该算法将返回顺序靠前补丁作为最终补丁。如果一轮交互无法使开发者判得出判断结果，可以迭代该过程，直到做出判断或选择下一个补丁。

人机交互模块中补丁审核功能的顺序图如图 4.17 所示，该图以“生成”动作为例描述在交互时的开发者与机器之间的联系。开发者从候选补丁列表中选择将要进行审核的补丁，触发 CodeService 对交互信息的处理并通过 PatchView 展示出来。开发者通过点击“generate”按钮向工具传达生成动作，ProjectView 通过 Controller 完成具体功能的实现，详见之前小节。完成后返回先后更新 ProjectView、PatchView 和代码编辑器，两个视图在更新后将信息呈现给开发者，代码编辑器则打开新增测试的代码。人机交互模块中的其它动作的实现流程也与此类似，不再陈述。

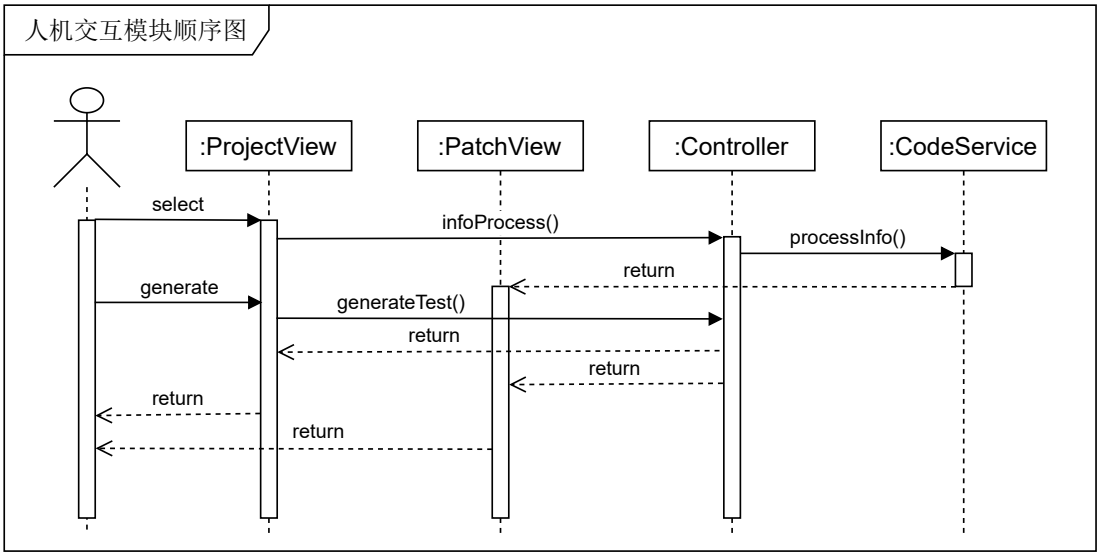

图 4.17: 补丁审核顺序图

4.3.4　关键代码

该模块除了响应开发者修复时的动作之外，还对需要对参数变化和执行轨迹这两个代码特征进行再处理。首先通过 Java 反射机制将故障方法对应的 JSON 转化成 Method 类，然后按照方法体分割动态分析记录的数据。对于参数变化，确定其触发错误时的执行，解析数据后输出到视图层。对于执行轨迹，则需要根据记录的行号，为指定的代码行打上颜色标签，Eclipse 客户端解析后输出到 view 中后就能呈现出颜色提示。人机交互模块的关键代码如图 4.18 所示。

4.4　界面展示

如图 4.19 所示，前端主要包括两个视图，项目视图（Project View）和补丁视图（Patch View），每个视图里面又包含了多个面板。项目视图展示了该缺陷项目的宏观信息，而补丁视图则是在开发者审核单个补丁时展示与特定补丁相关的信息。

从项目视图里，开发者可以获得与缺陷项目相关的整体信息，并且提供了与开发者之间的交互动作。为了更清晰的显示不同种类的信息，它进一步细分为三个面板。图 4.19 区域 1 是第一个面板，列出了待修复的缺陷项目的基础信息；下方有三个按钮，分别提供重置所有数据、执行项目测试和结束程序修复的功能；接着则是在执行项目测试后呈现测试结果，由于应用补丁之后的项目能够顺利通过原有的全部测试用例，因此原有的配套测试用例的结果不再展示。图 4.19 区域 2 是第二个面板，包含两部分信息，分别是测试信息和位置信息，测

```
// 分割
StringBuilder resultContentString = new StringBuilder();
for (String line : traceContentList) {
    if (StringUtils.isBlank(line.trim())) {
        continue;
    }
    if (line.startsWith(methodStartLine)) {
        ++currentMethodCnt;
    }
    if (currentMethodCnt == methodCnt) {
        resultContentString.append(line).append("\n");
    }
}

// 着色
for (String contentLine : colorContentString.split("\n")) {
    if (contentLine.startsWith(method.getMethodName())) {
        int number = Integer.parseInt(contentLine.split("#")[contentLine.split("#").length - 1]);
        if (number == 0) {
            continue;
        } else if (number < i) {
            continue;
        } else if (number == i) {
            line = "<span color=\"" + Cover + "\">" + line + "</span><br/>";
            covered = true;
            break;
        } else {
            line = "<span color=\"" + NotCover + "\">" + line + "</span><br/>";
            covered = true;
            break;
        }
    }
}
```

图 4.18: 人机交互模块关键代码

试信息中包含触发错误的测试用例及其当前候选补丁数量，位置信息包错误的位置及其当前的候选补丁数量。图 4.19 区域 3 是第三个板块，候选补丁列表，罗列了截至目前为止在列表中的全部补丁，每个补丁都配有三个按钮，分别提供删除补丁、生成测试和选为最终补丁的功能，点击补丁的名称即能展开看到补丁的详细代码，补丁对源代码做出的修改也用颜色进行了标注，使其更醒目。

从补丁视图里，开发者可以获得与当前审核的特定补丁相关的信息，用于辅助开发者标注测试预言。为了分别显示不同种类的信息，它进一步细分为三个面板。图 4.19 区域 4 是第一个面板，用于比较缺陷项目应用补丁之前和之后的代码运行轨迹的差异，绿色标注的部分表明该语句被执行。图 4.19 区域 5 是第二个面板，罗列了更改的方法在应用补丁之前和之后，值发生变化的参数。图 4.19 区域 6 是第三个面板，展示了为该补丁生成的测试用例所完成的覆盖指标。通过全方位展示项目与补丁的情况，开发者可以清楚地了解补丁对项目的影响，从而做出最有利的选择。

整个界面在逻辑上都是相互关联的，点击一个按钮或选择一个部分可能会触发其他位置的更新。在图 4.19 中，开发者在项目视图中候选补丁列表里选择了 patch1，补丁视图随即在区域 4 和区域 5 分别更新了代码差异和参数变化，区

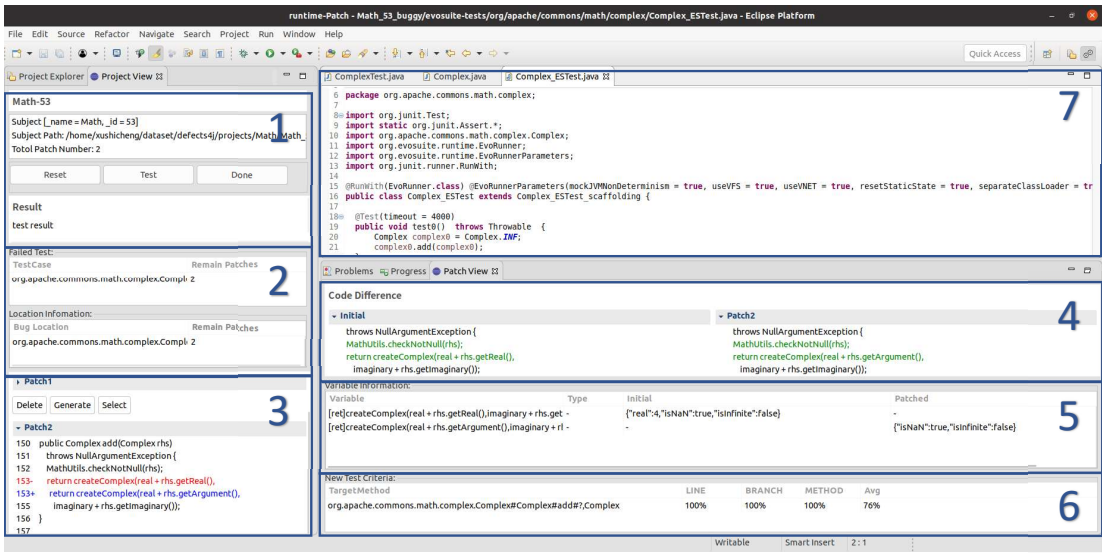

图 4.19: 系统界面图

域 7 同步打开了变更的文件；当开发者点击了“Generate”按钮，补丁视图在区域 6 更新了覆盖指标，区域 7 同时打开了新的测试文件；开发者点击“Test”按钮后，项目视图会在区域 1 更新测试结果。

4.5　本章小结

本章基于第三章系统需求分析和概要设计，对数据分析模块、测试管理模块、人机交互模块的详细设计与实现进行了阐述，每个模块首先讲述详细设计的方法，包括设计思路和流程分析；然后通过核心类图、顺序图和关键代码展示实现的具体细节；最后展示并说明本系统的运行界面。

第五章　系统测试与实验分析

5.1　系统测试

5.1.1　测试准备

测试目标。本节将开展对面向自动程序修复的人机交互式补丁审核系统的系统测试，系统测试的目的是了解系统各个模块的功能设计是否满足需求，验证系统能否正常运行，通过软件测试来保障系统的开发质量。

表 5.1: 系统测试环境表

环境名称	环境信息
操作系统	Ubuntu 20.04 LTS
测试软件	Eclipse Oxygen
数据库	MySQL 5.7.28
其它环境和工具	JDK 1.8.0_271 Git 2.25.1 SVN 1.13.0 Defects4J 2.0.0 EvoSuite 1.1.0

测试环境。表 5.1 展示了开展系统测试所用到的环境信息，系统的开发和测试全部在 Ubuntu 上实施，使用 Eclipse 将系统以插件的形式集成到本地 IDE 中运行具体的功能，使用 MySQL 数据库存储必要的基本数据。除此之外，系统的正常运转还使用到了 Git 等工具，也都列举在了测试环境表中。

5.1.2　功能测试

功能测试针对系统在需求分析阶段提出的系统功能需求，为系统用例进行逐项检查，以验证系统是否达到预期。每一个测试用例包括测试 ID、测试名称、测试目标、测试步骤和测试结果。

表 5.2 展示了候选补丁管理功能的测试详情，该测试用例对应表 3.3 描述的系统用例 UC01，主要测试内容为候选补丁的管理，开发者围绕候选补丁的一系列操作是否得到正常的响应并取得预期的效果。

表 5.3 展示了交互信息查看功能的测试详情，该测试用例对应表 3.4 描述的系统用例 UC02，主要测试内容为与交互信息相关的功能，开发者通过该功能得

表 5.2: 候选补丁管理功能测试

测试 ID	TC01
测试名称	候选补丁管理功能测试
测试目标	开发者能够浏览候选补丁、查看详情，并执行相应操作
测试步骤	1. 开发者进入系统界面，查看待审核的补丁是否按预期列出； 2. 开发者点击补丁名称，查看补丁详细信息； 3. 开发者点击“删除”，从列表中移除补丁； 4. 开发者点击“生成”，查看是否生成了新的测试用例以及界面更新情况； 5. 开发者点击“选定”，将补丁选择为最终答案。
测试结果	1. 系统显示待修复程序概要信息，并把候选补丁按序列出； 2. 系统显示补丁详细信息； 3. 系统将补丁移除出列表，并刷新系统界面； 4. 系统开始测试生成功能； 5. 系统提示“将该补丁选为最终答案”，并退出修复流程。

到与所选补丁相关的交互信息，这些信息来自于数据分析模块对原始数据的收集和处理。

表 5.3: 交互信息查看功能测试

测试 ID	TC02
测试名称	交互信息查看功能测试
测试目标	开发者能够选择自己想要查看的交互信息
测试步骤	1. 开发者点击补丁名称，查看与补丁相关的交互信息； 2. 开发者点击补丁代码，系统跳转至对应文件指定位置。
测试结果	1. 系统展开代码详情，并更新系统补丁视图，展示对应的交互信息； 2. 系统打开补丁修改的源文件，并跳转至修改的具体位置。

表 5.4 展示了测试用例构造功能的测试详情，该测试用例对应表 3.5 描述的系统用例 UC03 和表 3.6 描述的系统用例 UC04，主要测试内容为开发者与机器针对测试用例的构造，包括开发者向系统下达生成测试用例的命令，以及开发者标注测试预言。

表 5.5 展示了缺陷项目测试功能的测试详情，该测试用例对应表 3.7 描述的系统用例 UC05，主要测试内容包括系统能否在开发者的要求下正常执行项目测试，并反馈结果信息，以及之后再次进行迭代审核是否存在异常状况。

表 5.6 展示了项目数据重置功能的测试详情，该测试用例对应表 3.8 描述的系统用例 UC06，主要测试内容是项目数据重置功能能否正常使用，当开发者使用该功能时，系统的数据状态和界面状态都应响应并更新。

表 5.4: 测试用例构造功能测试

测试 ID	TC03
测试名称	测试用例构造功能测试
测试目标	开发者发起测试用例扩增并获得反馈，并与机器共同完成测试用例的构造
测试步骤	1. 开发者点击“生成”按钮，查看系统执行情况； 2. 开发者修改新增测试用例。
测试结果	1. 系统后台执行测试生成的功能，完成后打开新增文件； 2. 系统分析新增测试信息，并更新系统补丁视图。

表 5.5: 缺陷项目测试功能测试

测试 ID	TC05
测试名称	缺陷项目测试功能测试
测试目标	开发者发起缺陷项目测试并获得反馈
测试步骤	1. 开发者点击“测试”按钮，获取测试结果； 2. 开发者点击“生成”按钮，查看系统迭代扩增测试用例的功能； 3. 开发者点击“完成”按钮，退出修复流程。
测试结果	1. 系统后台执行测试，完成后将测试结果更新在系统项目视图里； 2. 系统为项目和补丁生成新一轮的测试用例； 3. 系统提示“修复完成”，并退出修复流程。

最终，我们严格按照上述测试目标和步骤，结合系统的功能需求和用例，逐项执行每个测试用例，测试的结果如表 5.7 所示，结果表明符合预期设计。通过功能测试，我们检测了系统的基本功能需求，证明了本系统的功能设计能够满足本系统的正常运转。

5.1.3　非功能测试

根据 3.2.3 节对非功能需求的分析，本节对系统的非功能特性进行测试，以验证系统是否满足预期的设计。系统的可靠性和可维护性通过 SLF4J 日志框架来记录系统运行过程中的信息，当系统运行出现故障时，可以通过日志追溯到

表 5.6: 项目数据重置功能测试

测试 ID	TC06
测试名称	项目数据重置功能测试
测试目标	开发者通过重置数据回到初始状态
测试步骤	1. 开发者点击“重置”按钮，查看系统界面变化。
测试结果	1. 系统清除全部修复动作和数据，并更新系统界面至开始状态。

表 5.7: 系统功能测试结果

测试 ID	对应用例 ID	测试结果
TC01	UC01	通过
TC02	UC02	通过
TC03	UC03、UC04	通过
TC05	UC05	通过
TC06	UC06	通过

出错的具体位置。系统通过界面的个性化信息展示和提示信息保证了系统的易用性，通过模块划分和分层设计保证了系统的可扩展性。通过对系统主要功能的运行时间进行统计，包括静态数据分析、动态数据分析、系统界面加载、测试用例生成和缺陷项目测试。表 5.8 展示了性能测试的测试结果，系统的主要功能的性能要求均符合预期。需要说明的是，测试用例生成功能耗费了 35s 的时间，同样也是符合预期的，因为 EvoSuite 需要花费较多时间去进行数据搜索和覆盖率计算。

表 5.8: 系统性能测试结果

功能	平均响应时间	测试结果
静态数据分析	381ms	符合预期
动态数据分析	3963ms	符合预期
系统界面加载	4896ms	符合预期
测试用例生成	35120ms	符合预期
缺陷项目测试	2952ms	符合预期

5.2　系统实验

5.2.1　实验一：修复模拟

1、实验准备

实验一目的是评估面向自动程序修复的人机交互式补丁审核系统的有效性，验证补丁审核方法能否筛选出正确的补丁，以及评估补丁审核方法中的各部分对于整体的影响和重要性。本实验提出了以下三个问题：（1）面向自动程序修复的人机交互式补丁审核系统所提出的方法能否有效移除不正确的补丁？（2）测试生成对面向自动程序修复的人机交互式补丁审核系统的影响如何？（3）开发者的参与对面向自动程序修复的人机交互式补丁审核系统的影响如何？

问题 1 的提出是为了评估补丁审核方法筛选补丁的有效性，理想情况下，经过几轮的交互，所有的错误补丁将被从候选补丁列表中移除，只留下正确的补

丁。为了回答问题 1，本文通过开发者与工具交互的次数，以及找到有效补丁前需要评估的候选补丁数，来验证补丁审核方法的有效性。问题 2 的提出是为了评估测试生成对补丁审核方法的影响，测试生成是补丁审核方法的重要组成部分，我们希望了解它对补丁审核方法重要性。为了回答问题 2，本文考虑两种生成策略进行对比，策略一就是按照本文的方法生成测试用例，策略二是不生成任何测试用例，然后将两种策略下补丁筛选的结果进行对比。问题 3 的提出是为了评估开发者在补丁审核方法中的作用，对比有无开发者的参与对补丁审核方法的影响。为了回答问题 3，本文同样考虑两种策略，其一就是按照本文的方法引入开发者标注预言，其二是生成测试用例后不与开发者交互，然后将两种策略下补丁筛选的结果进行对比。

2、实验设置

表 5.9: 自动程序修复工具基本信息

工具	介绍
ACS [68]	基于多个信息源，通过统计和启发式方式修复程序错误
HDRepair [32]	使用历史修复信息来指导基于搜索的修复过程
jGenProg	GenProg [15] 在 Java 语言上的重新实现，是一种基于遗传算法的修复工具
jKali	Kali [11] 在 Java 语言上的重新实现，仅通过删除功能进行修复
Nopol [24]	根据约束求解来修复不正确的条件，包括 2015 和 2017 两个版本

补丁数据。为了模拟补丁审核方法对自动程序修复工具修复能力的提升，本文使用了多种由现有工具的生成的补丁，这些工具都是针对 Java 程序中的错误。我们从之前的研究中收集了这些补丁数据 [24, 33, 68–71]，表 5.9 展示了这些工具的基本信息。所有的补丁都是为 Defects4J [1] 中的错误生成的，且都在 Defects4J 数据集上进行了实验并提供了结果。其中，并不是所有的工具都为每个错误生成了补丁，并且有些不适用于本实验的补丁已经被移除。

表 5.10: 待修复项目基本信息

标识	项目名称	错误数量	补丁数量	正确补丁数	错误补丁数
Chart	jfreechart	14	27	4	23
Lang	commons-lang	12	17	6	11
Math	commons-math	45	89	21	68
total	3	71	133	31	102

项目数据。Defects4J 是由 Just 等人提出的包含了真正错误的数据库和可扩

[1] https://github.com/rjust/defects4j

展的框架，可以在软件测试研究中进行可重复的研究。Defects4J（Defects for Java）是一个针对 Java 程序进行测试研究的缺陷数据集 [72]，每个真实的错误都附带有配套的测试用例，用于展示和模拟该错误。Defects4J 中包含了多个项目，通过收集其在开发和版本迭代中产生的错误，Defects4J 将其收录并编号。本文从中选择三个项目，项目详情如表 5.10 所示：Chart [2] 是用于显示图表的库，Lang [3] 是一组用于操作字符串等的辅助方法，以及 Math [4] 是用于科学计算的库。

实验过程。我们为 3 个项目的 71 个程序错误收集了 133 个补丁，其中 31 个正确补丁，102 个错误补丁。为了模拟真正的交互过程，在每一轮的审核中，系统随机挑选一个补丁交给开发者进行交互式修复。

3、实验结果分析

表 5.11 展示了实验一的实验结果，前三列分别是“参与实验任务”、“候选补丁个数”和“正确补丁/不正确补丁”（C = Correct，IC = InCorrect），“评估个数”列表示确定最终补丁所评估的补丁数目，“交互次数”列表示确定最终补丁所进行交互的次数，需要注意的是，每个补丁不只进行了一次交互，“剩余个数”列表示在人机交互修复后候选补丁列表中剩余的数量，最后一列说明了结果是否有效，即最终补丁是否是正确的补丁。

表 5.11 回答了问题 1，从“评估个数”可以看出，候选补丁列表中有 84.61% 的补丁都需要经过开发者的评估，个别情况下，开发者在评估某个补丁时有把握确定其正确性，则会直接确定为正确补丁，因而减少了评估的个数。从“交互次数”可以看出，平均每个补丁经历了 1.61 次的交互次数，因此我们可以总结出这样的发现，在面对不正确的补丁时，通过一轮测试生成和合理的预言标注，可以确定其正确性；而在面对正确的补丁时，则需要经过多轮交互的验证。从“剩余个数”可以看出，不正确补丁中的 82.61% 可以被过滤掉，正确补丁中的 60% 会被留下。综合来看，对于表中 16 个修复任务，56.25% 最终得到了有效的结果，成功筛选出正确补丁或过滤掉不正确补丁。但是，表 5.11 中也存在着实验任务的结果是无效的。例如，Math73 过滤掉了全部补丁，但是其候选列表中存在正确的补丁；Math81 筛选出一个最终补丁，但是其候选列表中都是不正确的。因此，本文提出的技术还是存在误报、漏报的问题。通过进一步观察发现，在候选补丁数目较多时，更有可能出现无效结果，说明补丁的数量会在一定程度上对开发者的判断产生影响。综上，对问题 1 的回答是：面向自动程序修复的人机交互式补丁审核系统针对不正确的补丁，通过开发者与机器有限次数的交

[2]https://github.com/jfree/jfreechart

[3]https://github.com/apache/commons-lang

[4]https://github.com/apache/commons-math

表 5.11: 程序修复实验结果

实验任务	补丁个数	C/IC	评估个数	交互次数	剩余个数	是否有效
Chart5	3	1/2	3	5	1	YES
Chart13	4	0/4	3	5	2	NO
Chart25	4	0/4	4	6	0	YES
Lang44	2	1/2	2	4	1	YES
Lang58	2	1/2	1	3	1	NO
Math2	3	0/3	3	5	0	YES
Math28	4	0/4	4	3	1	NO
Math41	1	0/1	1	1	0	YES
Math49	4	0/4	3	4	1	NO
Math50	5	3/2	3	6	2	YES
Math53	2	1/1	2	3	1	YES
Math73	4	1/3	2	2	0	NO
Math81	5	0/5	4	9	1	NO
Math82	5	1/4	3	4	2	NO
Math85	4	1/3	4	7	1	YES
Math105	2	0/2	2	4	0	YES
total	52	10/46	44	71	14	-

互，能够有效移除不正确的补丁，完成对缺陷程序的修复。

表 5.12: 对比修复实验结果

试验任务	策略一	策略二	策略三
Chart5	YES	NO	YES
Chart25	YES	NO	NO
Lang44	YES	NO	YES
Math2	YES	NO	NO
Math41	YES	YES	NO
Math53	YES	NO	NO
Math85	YES	NO	NO
Math105	YES	NO	NO

针对问题 2 和问题 3，我们将表 5.11 中结果有效的 8 个任务拿出来，设置三种策略并进行对比。策略一就是本文提出的面向自动程序修复的人机交互式补丁审核系统所使用的方法；策略二不考虑机器的参与，开发者仅根据补丁和项目进行判断；策略三不考虑开发者的参与，仅利用自动生成的测试用例来验证补丁的正确性。对比结果如表 5.12 所示。通过数据我们发现，在不考虑机器参与的情况下，仅有一个任务筛选出了正确补丁。在不考虑开发者参与的情况下，

有两个任务完成了对错误的修复。

综上，对问题 2 的回答是，机器生成的测试用例是补丁正确性识别的重要基础，测试生成对面向自动程序修复的人机交互式补丁审核系统具有很大的影响。对问题 3 的回答是，开发者依据代码特征对测试用例做出的修改提高了测试用例的充分性，参与其中的作用对面向自动程序修复的人机交互式补丁审核系统具有较大影响。因此，我们可以得出结论，测试生成和预言标注对本文技术的实现都十分重要，其中测试用例所造成的影响更大一些，而人为带来的影响与候选补丁的个数有一定的关联。

5.2.2　实验二：用户调研

1、实验准备

实验二目的是调查面向自动程序修复的人机交互式补丁审核系统对开发者的实际帮助如何。由于补丁审核方法是一个交互式方法，因此很有必要展开用户调查，了解补丁审核方法在实际应用中是如何运作的，以及它能为开发者修复错误提供怎样的帮助。本实验提出以下两个问题：（4）开发者如何使用本系统进行程序修复?（5）开发者对交互式修复工具的反馈如何?

2、实验设置

任务设计。本文选择了两个缺陷项目作为实验任务，两个任务中均包含正确和不正确的补丁，但是其中的数量并不一样，方便我们实施对比实验，表 5.13 展示了两个任务的详细信息。任务的参与者均是来自软件工程专业的研究生，他们都拥有至少 3 年的 Java 编程经验，熟悉 Eclipse 的使用并了解软件测试的相关理论。收集这样一批经历过专业训练的开发者对我们修复工具的反馈，其数据和结果更具参考价值。

表 5.13: 实验任务信息

	缺陷项目	候选补丁个数	C/IC
任务 1	Math50	5	3/2
任务 2	Math82	5	1/4

实施过程。为了回答问题 4 和问题 5，在本实验中，参与者被平均分为了两组，每组参与者需要完成该组中对应的两个任务，这两个任务分属于两种不同的修复方案，详细信息如表 5.14 所示。具体来说，“手动修复” 意味着参与者仅根据项目代码和原生测试判断补丁的正确性，而 “工具修复” 则使用本文的技术来审核补丁。在分组上，组 A 需要手动修复任务 1 并使用工具修复任务 2，组 B

手动修复任务 2 并使用工具修复任务 1。此外，为了让参与者正常使用本文的系统，我们让参与者在实验之前先用本系统完成一个项目的修复，直到我们认定其可以参与实验。

表 5.14: 实验分组信息

	手动修复	工具修复
组 A	任务 1	任务 2
组 B	任务 2	任务 1

在完成指定任务后，我们对每个参与者开展问卷调查并收集他们的反馈。问卷的内容包括 (a) 参与者对本系统的满意度如何？(b) 哪些信息对修复提供的帮助较大？(c) 参与者对机器生成的测试用例做了多少修改？

3、实验结果分析

表 5.15: 分组实验结果

	手动修复	工具修复
任务 1	75%	100%
任务 2	25%	75%

表 5.15 展示了两种不同方案下成功完成修复的比例，本文提出的方法（工具修复）相对于手动修复有着显著的提高。在另一方面，任务 2 比任务 1 拥有更多不正确补丁，数据表明这会对参与修复工作的开发者造成一定影响，尤其是对手动修复的工作来说，但是对于工具修复而言，这样的影响并不明显。

我们通过问卷调查，从实验的参与者处收集了他们的反馈。对于问题 (a)，本系统的满意度如图 5.1 所示，绝大多数参与者对交互式修复系统持有积极的态度，结合表的修复数据来看，工具修复确实为修复工作带来了帮助从而推动了满意度的提升；少部分参与者对于该工具的使用不是十分满意，根据我们收集来的反馈，总结有以下两点原因：一是交互时间有待控制，二是在面对复杂的代码或程序调用时，现有的交互信息有待完善以帮助他们完全理解程序。对于问题 (b)，参与者认为我们的系统为修复提供的信息中，“执行轨迹” 这一信息的帮助最大，参与者针对方法内部的运行路线，修改测试尝试使数据走向别的路径，以解决配套测试用例在路径覆盖上的不充分性；“参数变化”、“新增测试覆盖率”，包括补丁修改的内容，同样也对参与者在选择测试数据时提供了一定的帮助。对于问题 (c)，参与者表示会根据自己对程序的理解程度来决定，通常会

结合触发错误的测试用例、补丁修改的内容和辅助信息来修改由工具生成的测试用例。

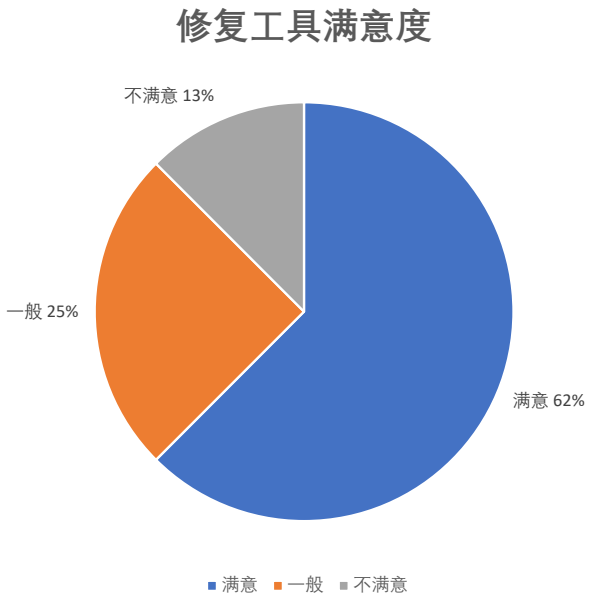

图 5.1: 系统满意度

通过对以上结果和反馈的分析，我们对问题 4 做出了回答，即开发者从候选补丁列表中选取补丁，结合各种不同的辅助信息，对为其生成的测试用例做出修改，根据测试结果决定该补丁是否正确。参与者基本按照本文设计的思路进行操作，本系统所提供的元素也都物尽其用。对于问题 5，参与者整体上对本系统的反馈是积极的，但同时也在使用中暴露了一些问题，是我们未来改进的方向之一。

5.2.3　有效性威胁

本节分析对系统实验结果的有效性造成影响的可能因素：

（1）实验项目的选择。Defects4J 是当前最全面的 Java 程序数据集，其开发目的是在软件测试研究中提供真实的错误和可重现的研究，此外它在软件测试中的研究工作以及自动程序修复研究工作中广泛用作评估主题。由于 Defects4J 数据庞大，我们只选取了一部分进行系统实验，可能导致实验结果没法推广的其它的情况下，对于更多程序缺陷的适用性，还需做进一步的研究和验证。

（2）实验补丁的选择。第二个主要威胁是本文提出的技术能否应用于不同的自动程序修复工具，我们使得选取的补丁尽量来自于不同的自动程序修复工具，同时这些工具的原理也是基于不一样的方法。但是我们仍然没有覆盖到所有的工具，以及未来提出的方法或工具是否会与当前的有所不同。

（3）开发者的选择。参与实验的开发者，至少具有一年以上的 Java 编程经验，并且也熟知软件测试等理论，我们在实验前也对每位参与者进行了引导和热身，让他们提前使用我们的工具。根据 Salman 等人 [73] 的研究，高校学生在合理的实验设置下，可以代表真实的开发者。因此，学生作为开发者，可以一定程度上代表修复工作的参与者。

5.3　本章小结

在这一章中，我们讨论了面向自动程序修复的人机交互式补丁审核系统的系统测试和系统实验，系统测试验证了本系统的可用性，分别从功能测试和非功能测试证明了系统是符合预期的。系统实验包括修复模拟和用户调查，在实验一中，我们通过在通用的缺陷数据集上进行实验，评估面向自动程序修复的人机交互式补丁审核系统在审核和过滤补丁时的有效性，同时还分别评估了开发者和工具在本系统中的重要性，二者都对补丁审核方法的效果起到了推动作用；在实验二中，我们通过引导开发者实际操作我们的工具，收集开发者在使用过程中的反馈，评估本系统的实际帮助和改进方向。

第六章　总结与展望

6.1　总结

随着软件复杂度的日益增大，软件缺陷也成了开发过程中难以避免的问题，通过传统人工主导的软件测试来排查问题、修复错误需要耗费昂贵的成本，自动程序修复技术的出现一定程度上提升了开发人员进行软件测试和调试的效率，缓解了开发成本。但是，受限于程序本身的复杂性和自动程序修复技术的局限性，自动程序修复工具的精度仍有待提高，因此对补丁的审核工作显得尤为重要。针对这一问题，本文设计并实现面向自动程序修复的人机交互式补丁审核系统，利用机器生成一定数量的测试用例，再通过数据分析引导开发者进行预言标注，实现对候选补丁的审核，从而有效过滤掉不正确的补丁，筛选出正确补丁，从而节约开发成本。本文的主要工作如下：

首先对当前自动程序修复技术进行调研，分析主要问题，提出解决方案，并从现有的自动程序修复工具中收集数据。通过 Eclipse AST 为相关的类建立抽象语法树分析指定代码，实现对代码信息的提取和修改等操作，使之能够在后续过程中充分地向开发者展示补丁对原项目做出的修改，从而帮助开发者做出决策。

其次对比市场上主流的测试生成工具，并根据我们的需求选取 EvoSuite 作为实现测试生成功能的核心工具，将其与本文技术进行适配和封装。主要使其能够为指定类生成一定数量的测试用例，并为开发者提供覆盖率指标。并借助抽象语法树提取测试用例的上下文内容，作为之后交互信息的扩充。

然后完成对该方法的集成，借助 Eclipse 设计并实现具有良好交互界面的系统。本系统分为数据分析模块、测试管理模块和人机交互模块，各模块之间相辅相成，并根据 MVC 的设计思想分离前端界面和业务模型。通过在界面上分割视图和板块，引导开发者参与交互，实现对每个缺陷项目的修复。

最后通过系统测试和实验分析对面向自动程序修复的人机交互式补丁审核系统的有效性进行验证和评估。系统测试验证了系统的功能和非功能是符合预期的。实验一模拟修复过程，通过交互次数、补丁评估个数、剩余补丁个数等指标证明本文方法是有效的，并设置对比策略，证明机器和人工在本文方法中的重要性。实验二通过用户调研，了解该开发者对本系统的满意度，以及对开发者的帮助。

本文首先阐述了研究背景和意义，并介绍相关领域的技术和工具；然后对系统进行需求分析和总体设计，并通过划分具体模块描述了系统的细节实现；最后开展系统测试和系统实验，验证本文系统和方法的有效性。

6.2 展望

本文设计并实现了面向自动程序修复的人机交互式补丁审核系统，是将测试生成和预言标注结合并应用到程序修复领域的一次探索。但是，由于时间和个人水平有限，本文提出的修复技术在未来还有很多需要优化和改进的部分。

1. 扩展修复工具。本文只选择了基于测试的自动程序修复工具所提供的补丁，而目前该领域已经出现了基于其它类型的修复技术。后续将收集来自不同类型的更多修复工具生成的补丁，让本文的技术更具适用性。

2. 扩展测试工具。本文目前选择了 EvoSuite 用于测试用例生成，该工具在测试市场上具有一定的优势，但也有一些方面不及其它工具。之后可以引入更多的测试工具完成这一功能，并观察不同工具带来的效果。

3. 扩展交互信息。目前本文为开发者交互提供的信息有三类，而补丁做出的修改可能不只这两方面。未来借助代码分析等技术，提取更多的信息，辅助开发者的交互。

4. 缩短评审时间。目前在每个补丁评审的过程中需要耗费较多时间，过长的等待极有可能影响开发者的情绪进而干扰判断。在之后的工作中，缩短评审的时间，进一步发挥机器的优势，提高开发者审核的效率。

综上，本文提出的面向自动程序修复的人机交互式补丁审核系统还存在可以继续提升、优化的点，主要的提升方向是人机交互式补丁审核方法的普适性和用户体验，会在后续工作中继续完善。

参考文献

[1] P. C. Jorgensen, Software testing - a craftsman's approach (3. ed.), Taylor & Francis, 2008.

[2] G. J. Myers, T. Badgett, T. M. Thomas, C. Sandler, The art of software testing, Vol. 2, Wiley Online Library, 2004.

[3] J. McDermid, Software engineering: A practitioner's approach [book review], Softw. Eng. J. 10 (6) (1995) 266.

[4] D. Kim, J. Nam, J. Song, S. Kim, Automatic patch generation learned from human-written patches, in: D. Notkin, B. H. C. Cheng, K. Pohl (Eds.), 35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013, IEEE Computer Society, 2013, pp. 802–811.

[5] 玄跻峰, 任志磊, 王子元, 谢晓园, 江贺, 自动程序修复方法研究进展, Journal of Software 27 (4).

[6] 王赞, 郜健, 陈翔, 傅浩杰, 樊向宇, 自动程序修复方法研究述评, 计算机学报 41 (3) (2018) 588–610.

[7] L. Gazzola, D. Micucci, L. Mariani, Automatic software repair: a survey, in: M. Chaudron, I. Crnkovic, M. Chechik, M. Harman (Eds.), Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018, ACM, 2018, p. 1219.

[8] M. Monperrus, Automatic software repair: a bibliography, CoRR abs/1807.00515.

[9] Y. Tao, J. Kim, S. Kim, C. Xu, Automatically generated patches as debugging aids: a human study, in: S. Cheung, A. Orso, M. D. Storey (Eds.), Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014, ACM, 2014, pp. 64–74.

[10] F. Long, M. C. Rinard, An analysis of the search spaces for generate and validate patch generation systems, in: L. K. Dillon, W. Visser, L. A. Williams (Eds.), Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016, ACM, 2016, pp. 702–713.

[11] Z. Qi, F. Long, S. Achour, M. C. Rinard, An analysis of patch plausibility and correctness for generate-and-validate patch generation systems, in: M. Young, T. Xie (Eds.), Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015, ACM, 2015, pp. 24–36.

[12] M. Monperrus, A critical review of "automatic patch generation learned from human-written patches": Essay on the problem statement and the evaluation of automatic software repair, CoRR abs/1408.2103.

[13] M. Harman, S. A. Mansouri, Y. Zhang, Search-based software engineering: Trends, techniques and applications, ACM Comput. Surv. 45 (1) (2012) 11:1–11:61.

[14] A. Arcuri, X. Yao, A novel co-evolutionary approach to automatic software bug fixing, in: Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2008, June 1-6, 2008, Hong Kong, China, IEEE, 2008, pp. 162–168.

[15] C. L. Goues, T. Nguyen, S. Forrest, W. Weimer, Genprog: A generic method for automatic software repair, IEEE Trans. Software Eng. 38 (1) (2012) 54–72.

[16] W. Weimer, T. Nguyen, C. L. Goues, S. Forrest, Automatically finding patches using genetic programming, in: 31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings, IEEE, 2009, pp. 364–374.

[17] W. Weimer, Z. P. Fry, S. Forrest, Leveraging program equivalence for adaptive program repair: Models and first results, in: E. Denney, T. Bultan, A. Zeller (Eds.), 2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013, IEEE, 2013, pp. 356–366.

[18] E. Fast, C. L. Goues, S. Forrest, W. Weimer, Designing better fitness functions for automated program repair, in: M. Pelikan, J. Branke (Eds.), Genetic and Evolutionary Computation Conference, GECCO 2010, Proceedings, Portland, Oregon, USA, July 7-11, 2010, ACM, 2010, pp. 965–972.

[19] Y. Qi, X. Mao, Y. Lei, Z. Dai, C. Wang, The strength of random search on automated program repair, in: P. Jalote, L. C. Briand, A. van der Hoek (Eds.), 36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014, ACM, 2014, pp. 254–265.

[20] V. Debroy, W. E. Wong, Using mutation to automatically suggest fixes for faulty programs, in: Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7-9, 2010, IEEE Computer Society, 2010, pp. 65–74.

[21] H. D. T. Nguyen, D. Qi, A. Roychoudhury, S. Chandra, Semfix: program repair via semantic analysis, in: D. Notkin, B. H. C. Cheng, K. Pohl (Eds.), 35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013, IEEE Computer Society, 2013, pp. 772–781.

[22] S. Mechtaev, J. Yi, A. Roychoudhury, Directfix: Looking for simple program repairs, in: A. Bertolino, G. Canfora, S. G. Elbaum (Eds.), 37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1, IEEE Computer Society, 2015, pp. 448–458.

[23] S. Mechtaev, J. Yi, A. Roychoudhury, Angelix: scalable multiline program patch synthesis via symbolic analysis, in: L. K. Dillon, W. Visser, L. A. Williams (Eds.), Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016, ACM, 2016, pp. 691–701.

[24] J. Xuan, M. Martinez, F. Demarco, M. Clement, S. R. L. Marcote, T. Durieux, D. L. Berre, M. Monperrus, Nopol: Automatic repair of conditional statement bugs in java programs, IEEE Trans. Software Eng. 43 (1) (2017) 34–55.

[25] S. R. L. Marcote, M. Monperrus, Automatic repair of infinite loops, CoRR abs/1504.05078.

[26] D. Gopinath, M. Z. Malik, S. Khurshid, Specification-based program repair using SAT, in: P. A. Abdulla, K. R. M. Leino (Eds.), Tools and Algorithms for the

Construction and Analysis of Systems - 17th International Conference, TACAS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings, Vol. 6605 of Lecture Notes in Computer Science, Springer, 2011, pp. 173–188.

[27] E. K. Smith, E. T. Barr, C. L. Goues, Y. Brun, Is the cure worse than the disease? overfitting in automated program repair, in: E. D. Nitto, M. Harman, P. Heymans (Eds.), Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015, ACM, 2015, pp. 532–543.

[28] M. Jiang, T. Y. Chen, F. Kuo, D. Towey, Z. Ding, A metamorphic testing approach for supporting program repair without the need for a test oracle, J. Syst. Softw. 126 (2017) 127–140.

[29] Y. Tao, D. Han, S. Kim, Writing acceptable patches: An empirical study of open source project patches, in: 30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014, IEEE Computer Society, 2014, pp. 271–280.

[30] L. D'Antoni, R. Samanta, R. Singh, Qlose: Program repair with quantitative objectives, in: S. Chaudhuri, A. Farzan (Eds.), Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II, Vol. 9780 of Lecture Notes in Computer Science, Springer, 2016, pp. 383–401.

[31] X. D. Le, D. Chu, D. Lo, C. L. Goues, W. Visser, S3: syntax- and semantic-guided repair synthesis via programming by examples, in: E. Bodden, W. Schäfer, A. van Deursen, A. Zisman (Eds.), Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017, ACM, 2017, pp. 593–604.

[32] X. D. Le, D. Lo, C. L. Goues, History driven program repair, in: IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1, IEEE Computer Society, 2016, pp. 213–224.

[33] Q. Xin, S. P. Reiss, Identifying test-suite-overfitted patches through test case generation, in: T. Bultan, K. Sen (Eds.), Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017, ACM, 2017, pp. 226–236.

[34] S. H. Tan, H. Yoshida, M. R. Prasad, A. Roychoudhury, Anti-patterns in search-based program repair, in: T. Zimmermann, J. Cleland-Huang, Z. Su (Eds.), Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016, ACM, 2016, pp. 727–738.

[35] 虞凯, 林梦香, 自动化软件错误定位技术研究进展, Ph.D. thesis (2011).

[36] R. Caballero, A. Riesco, J. Silva, A survey of algorithmic debugging, ACM Comput. Surv. 50 (4) (2017) 60:1–60:35.

[37] X. Li, S. Zhu, M. d'Amorim, A. Orso, Enlightened debugging, in: M. Chaudron, I. Crnkovic, M. Chechik, M. Harman (Eds.), Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018, ACM, 2018, pp. 82–92.

[38] L. Gong, D. Lo, L. Jiang, H. Zhang, Interactive fault localization leveraging simple user feedback, in: 28th IEEE International Conference on Software Maintenance, ICSM 2012, Trento, Italy, September 23-28, 2012, IEEE Computer Society, 2012, pp. 67–76.

[39] A. J. Ko, B. A. Myers, Designing the whyline: a debugging interface for asking questions about program behavior, in: E. Dykstra-Erickson, M. Tscheligi (Eds.), Proceedings of the 2004 Conference on Human Factors in Computing Systems, CHI 2004, Vienna, Austria, April 24 - 29, 2004, ACM, 2004, pp. 151–158.

[40] Y. Lin, J. Sun, Y. Xue, Y. Liu, J. S. Dong, Feedback-based debugging, in: S. Uchitel, A. Orso, M. P. Robillard (Eds.), Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017, IEEE / ACM, 2017, pp. 393–403.

[41] A. J. Ko, B. A. Myers, Debugging reinvented: asking and answering why and why not questions about program behavior, in: W. Schäfer, M. B. Dwyer, V. Gruhn

(Eds.), 30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, ACM, 2008, pp. 301–310.

[42] P. Cashin, C. Martinez, W. Weimer, S. Forrest, Understanding automatically-generated patches through symbolic invariant differences, in: 34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019, IEEE, 2019, pp. 411–414.

[43] M. Böhme, C. Geethal, V. Pham, Human-in-the-loop automatic program repair, in: 13th IEEE International Conference on Software Testing, Validation and Verification, ICST 2020, Porto, Portugal, October 24-28, 2020, IEEE, 2020, pp. 274–285.

[44] J. Liang, R. Ji, J. Jiang, Y. Lou, Y. Xiong, G. Huang, Interactive patch filtering as debugging aid, CoRR abs/2004.08746.

[45] 陈翔, 鞠小林, 文万志, 顾庆, 基于程序频谱的动态缺陷定位方法研究, Ph.D. thesis (2015).

[46] C. L. Goues, M. Dewey-Vogt, S. Forrest, W. Weimer, A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each, in: M. Glinz, G. C. Murphy, M. Pezzè (Eds.), 34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland, IEEE Computer Society, 2012, pp. 3–13.

[47] K. Anderson, A. Schram, Proceedings of the international conference on software engineering (2011).

[48] M. Weiser, Program slicing, IEEE Trans. Software Eng. 10 (4) (1984) 352–357.

[49] K. B. Gallagher, J. R. Lyle, Using program slicing in software maintenance, IEEE Trans. Software Eng. 17 (8) (1991) 751–761.

[50] 孙继荣, 李志蜀, 王莉, 殷锋, 金虎, 程序切片技术在软件测试中的应用, 計算機應用研究 24 (5) (2007) 210–213.

[51] I. Neamtiu, J. S. Foster, M. Hicks, Understanding source code evolution using abstract syntax tree matching, ACM SIGSOFT Softw. Eng. Notes 30 (4) (2005) 1–5.

[52] M. Ellims, J. Bridges, D. C. Ince, The economics of unit testing, Empirical Software Engineering 11 (1) (2006) 5–31.

[53] I. Hooda, R. Chhillar, A review: Study of test case generation techniques, International Journal of Computer Applications 107 (16).

[54] S. M. Mohi-Aldeen, S. Deris, R. Mohamad, Systematic mapping study in automatic test case generation, in: H. Fujita, A. Selamat, H. Haron (Eds.), New Trends in Software Methodologies, Tools and Techniques - Proceedings of the Thirteenth SoMeT '14, Langkawi, Malaysia, September 22-24, 2014, Vol. 265 of Frontiers in Artificial Intelligence and Applications, IOS Press, 2014, pp. 703–720.

[55] K. Sen, DART: directed automated random testing, in: K. S. Namjoshi, A. Zeller, A. Ziv (Eds.), Hardware and Software: Verification and Testing - 5th International Haifa Verification Conference, HVC 2009, Haifa, Israel, October 19-22, 2009, Revised Selected Papers, Vol. 6405 of Lecture Notes in Computer Science, Springer, 2009, p. 4.

[56] K. Sen, D. Marinov, G. Agha, CUTE: a concolic unit testing engine for C, in: M. Wermelinger, H. C. Gall (Eds.), Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005, ACM, 2005, pp. 263–272.

[57] C. Pacheco, M. D. Ernst, Randoop: feedback-directed random testing for java, in: R. P. Gabriel, D. F. Bacon, C. V. Lopes, G. L. S. Jr. (Eds.), Companion to the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada, ACM, 2007, pp. 815–816.

[58] L. Baresi, P. L. Lanzi, M. Miraz, Testful: An evolutionary test approach for java, in: Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7-9, 2010, IEEE Computer Society, 2010, pp. 185–194.

[59] G. Fraser, A. Arcuri, Evosuite: automatic test suite generation for object-oriented software, in: T. Gyimóthy, A. Zeller (Eds.), SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and

ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011, ACM, 2011, pp. 416–419.

[60] G. Fraser, A. Arcuri, Evosuite: On the challenges of test case generation in the real world, in: Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013, IEEE Computer Society, 2013, pp. 362–369.

[61] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, S. Yoo, The oracle problem in software testing: A survey, IEEE Trans. Software Eng. 41 (5) (2015) 507–525.

[62] M. Harman, S. G. Kim, K. Lakhotia, P. McMinn, S. Yoo, Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem, in: Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7-9, 2010, Workshops Proceedings, IEEE Computer Society, 2010, pp. 182–191.

[63] G. Jahangirova, D. Clark, M. Harman, P. Tonella, Test oracle assessment and improvement, in: A. Zeller, A. Roychoudhury (Eds.), Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016, ACM, 2016, pp. 247–258.

[64] E. Clayberg, D. Rubel, eclipse Plug-ins, Addison-Wesley, 2009.

[65] T. M. Mitchell, Machine learning. pressed by (2003).

[66] P. Kruchten, The 4+1 view model of architecture, IEEE Softw. 12 (6) (1995) 42–50.

[67] E. Bruneton, Asm 4.0-a java bytecode engineering library, 2011, URL http://download. forge. objectweb. org/asm/asm4-guide. pdf. Referenced 20 (10).

[68] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, L. Zhang, Precise condition synthesis for program repair, in: S. Uchitel, A. Orso, M. P. Robillard (Eds.), Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017, IEEE / ACM, 2017, pp. 416–426.

[69] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, M. Monperrus, Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset, CoRR abs/1811.02429.

[70] SerVal-DTF, Fl-vs-apr, https://github.com/SerVal-DTF/FL-VS-APR.

[71] Y. Xiong, X. Liu, M. Zeng, L. Zhang, G. Huang, Identifying patch correctness in test-based program repair, in: M. Chaudron, I. Crnkovic, M. Chechik, M. Harman (Eds.), Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018, ACM, 2018, pp. 789–799.

[72] R. Just, D. Jalali, M. D. Ernst, Defects4j: a database of existing faults to enable controlled testing studies for java programs, in: C. S. Pasareanu, D. Marinov (Eds.), International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014, ACM, 2014, pp. 437–440.

[73] I. Salman, A. T. Misirli, N. J. Juzgado, Are students representatives of professionals in software engineering experiments?, in: A. Bertolino, G. Canfora, S. G. Elbaum (Eds.), 37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1, IEEE Computer Society, 2015, pp. 666–676.

致　谢

在本篇论文完成之际，我想向硕士就读生涯中所有帮助和关心过我的老师、同学、朋友、亲人致以衷心的感谢。

首先，感谢我的指导老师陈振宇老师和房春荣老师对我的谆谆教诲和耐心帮助。感谢两位导师从选题到中期到最后对项目和论文的指导，帮助我整理思路，明确研究可行性的方向。感谢两位导师对项目和论文进度的不断关心和鞭策，对我按时完成论文产生了莫大的帮助。在此，对两位导师致以最高的敬意。

其次，我要感谢在毕设期间帮助过我的所有同学和朋友。感谢实验室的张犬俊学长，帮助我梳理项目思路，指导我修改论文。感谢实验室赵源学长，对我毕设论文选题和进度的督促。同时还要感谢为项目数据和实验提供帮助的同学，让我能够顺利开展论文的写作。感谢你们的热心帮助，从你们身上我学到很多具体的知识和方法以及对技术的认真和追求。

最后，我要感谢父母对我学习和生活的支持和鼓励，在我经历消极情绪时，对我予以开导和安慰，支持我完成多年的求学生涯，并为未来的工作提出指导和建议。

版权与原创性说明

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，本论文不包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。

作者签名：

日期：2021 年 05 月 20 日