南京大学

研 究 生 毕 业 论 文

（申请工程硕士学位）

论 文 题 目 基于Git仓库的团队代码质量控制系统的设计与实现

作 者 姓 名 刘锦涛

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 刘嘉 副教授

2019 年 5 月

学　　号：**MF1732083**

论文答辩日期：**2019 年 5 月 13 日**

指 导 教 师：　　（签字）

The Design and Implementation of Code Quality Control System for Development Team Based on Git

By

Jintao Liu

Supervised by

Associate Professor **Jia Liu**

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 2019

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：基于Git仓库的团队代码质量控制系统的设计与实现

工程硕士（软件工程领域） 专业 2017 级硕士生姓名： 刘锦涛

指导教师（姓名、职称）： 刘嘉 副教授

摘 要

现如今，软件开发团队越发重视控制项目代码质量，然而现有的代码质量控制工具只关注项目的源代码，少有工具考虑开发者的贡献对项目代码质量的影响。虽然现有质量控制工具能够从多个不同的代码质量维度检查项目代码质量，但是项目团队很难直接利用检查结果辅助团队决策，项目管理者也无法从质量评估结果中明确项目代码质量问题的责任划分。因此需要一个在传统代码质量控制功能基础上，从团队管理者的角度出发的代码质量控制系统。一方面整合大量的代码质量评估结果，为开发团队提供宏观的项目质量报告；另一方面结合项目开发历史，利用代码质量评估结果评价开发者和开发活动对项目的贡献质量。通过量化开发者、开发活动的贡献质量，可以有效地帮助团队管理者掌握项目开发状态，制定更加合理的开发计划。本文中所讨论的开发活动是某一个阶段内活动主体对项目提交的所有贡献的集合，其中活动主体至少由一名开发者构成。

本文主要阐述了基于Git仓库的团队代码质量控制系统的设计与实现，系统的主要功能由三个模块共同承担，包括提取项目代码质量特征的代码质量控制模块、管理和挖掘项目仓库的Git仓库控制模块和整合数据给出评估结果的团队代码质量评估模块。为了保证项目具有良好的可扩展性，模块之间通过元数据传递信息，调用发起模块通过数据适配器适配来自被调用模块的元数据。

本项目的研究方向是对衡量开发者、开发活动贡献质量的尝试，旨在通过开发者与文件之间的关系衡量开发者的贡献质量，建立一个基于Git仓库的团队代码质量控制系统。系统目前已经投入使用，并以ZooKeeper项目为对象进行了系统测试，测试结果表明系统能够准确地找出ZooKeeper项目中每个代码质量问题所对应的开发者和开发活动，并且能够给出开发者和开发活动的质量评分。

关键词： 代码质量控制，软件工程，团队协作，版本控制

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 Code Quality Control System for Development Team Based on Git

SPECIALIZATION: Software Engineering

POSTGRADUATE: Jintao Liu

MENTOR: Associate Professor Jia Liu

Abstract

Nowadays, software development teams have paid more and more attention to code quality. However, the existing code quality control tools focus only on the code itself, and hardly none of them spares attention to the impact of the developer's contribution on the quality of the project code. While existing quality control tools can check project code quality from multi-dimensions, it is either difficult for teams to directly leverage inspection results to assist decisions, nor for project managers to identify who should be responsible for which project code quality issue from quality assessment results. Therefore, we need a code quality control system that from the point of view of the team manager, more than traditional code quality control tools. On the one hand, the system integrates a large number of code quality assessment results to provide a general code quality evaluation for the development team, On the other hand, the system can combine the history of project development and use the results of code quality assessment to evaluate the quality of the contribution of developers and development activities to the project. By quantifying the output quality of developers and development activities, the system thus can effectively help team managers master the status of current development activities and make more reasonable development plans. The development activity discussed in this thesis is a collection of all contributions made by the code contributor to the project within a certain stage, in which the code contributor is composed of at least one developer.

This thesis mainly expounds the design and implementation of the code quality control system for development team based on Git. The main functions of the system are responsible by three modules together, including the code quality control module that extracts the quality characteristics of the code, the Git control module that manage

and excavates the evolution history of the project, and the team's code quality evaluation module that integrates the data to give the evaluation results. In order to ensure the well scalability of the system, the module transmits information through metadata, and invokes the initiator module to fit the metadata from the invoked module through the data converter.

The research direction of this project is an attempt to measure the quality of the contribution of developers and development activities, with the aim of building a Git based code quality control system for development team by measuring the contribution of developers through the relationship between developers and files. The system has been put into use, and the system test has been done for it on the object of ZooKeeper project, The test results show that the system can accurately identify the developers and development activities corresponding to each code quality problem in the ZooKeeper project, and can score the quality of the developer and the development activity.

Keywords: Code Quality Control, Software Engineering, Team Collaboration, Version Control

目　录

表　目　录

图　目　录

第一章　引言

1.1　团队代码质量控制的研究背景与意义

如今，软件行业在互联网的催化下高速发展，面对数量庞大且快速变化的用户需求，开发团队不得不在“速度”和“质量”之间做出权衡、取舍 [1]，项目代码质量控制和团队管理的难度也因此不断增加。在过去，进行人工代码审核是有很有价值的事情 [2]，但是如今，团队管理者把握产品质量、评估团队成员对项目的贡献 [3, 4]、分配开发任务、制定项目决策的难度在不断增加 [5]，为了能够在“速度”和“质量”之间找到平衡点，开发团队需要借助自动化工具在开发过程中保证代码质量。

开发团队通常会在工作流中引入代码质量控制系统，减少开发者在编码环节引入缺陷代码的风险，降低项目后期维护、迭代、升级的成本。在敏捷开发领域，开发团队通过在开发工作流中引入“持续集成”来提高工作效率、保障代码质量。持续集成工具通过不断集成代码产出，并利用自动化测试自动化告警功能，从多个质量维度持续对开发者进行反馈，帮助团队提高代码质量 [6]、衡量贡献 [4]。

虽然现今业界存在大量代码质量度量工具，但是，现有工具只关注代码本身，却少有工具对开发者及其开发周期内的活动进行评估，而开发者的行为对代码产生的影响往往是巨大的 [4]。现有工具往往评估的是某一个版本的项目代码质量，只评估了项目开发过程中的某一个截面。但是开发活动存在着一个相对较长的时间跨度，团队管理者很难仅从一个截面的质量评估结果看到开发周期内的详细状况。

团队管理者利用代码质量控制系统往往得到的是来自多个维度的关于整个项目的代码质量评估结果，尽管评估条目详尽，但是数量庞大的代码质量数据反而增加了团队管理者的管理负担 [7]。管理者无法根据质量评估结果直观地掌握开发周期内团队各个成员的开发活动，以及查看开发活动对项目所贡献代码的质量详细信息。

本课题实现了一套基于Git仓库的团队代码质量控制系统——BetterCode，BetterCode在传统的代码质量控制工具的基础上，评估Java项目在多个质量维度的质量状态。并利用版本控制系统——Git，将质量评估主体由传统工具关注的软件代码本身扩展到与代码相关的开发活动和活动参与者，设计并实现了针对团队开发项目的代码质量评估、控制系统。系统帮助开发团队更有效地利用代

码质量评估报告；帮助团队管理者更直观地掌握团队开发动态，更合理地安排开发活动。

系统同时也被应用于软件工程教学实践中，帮助教师或助教实时掌握学生团队中每一个开发者的开发动态，帮助教师根据每一个学生的开发质量指定有针对性的教学计划。学生之间也可以通过项目评估结果，调整开发进度，同时也增加了团队内部信息流动，促进了成员间的交流，有利于学生技能的提高[8]。本系统在高校本科生的一门软件工程实践课程中被用于管理、分析和跟踪学生开发小组的课程设计开发动态，系统给出的结果报告会被作为评估学生成绩的参考依据。

系统从“项目代码”和“团队活动”两个角度出发，拓展了传统代码质量控制系统的边界。其主要用于评估开发团队成员的代码贡献对整体代码质量的影响，分析团队成员间的协作关系对代码质量的影响，帮助团队管理者掌握团队中开发活动对项目代码质量的影响。

项目评估过程可分为“项目代码质量度量”和“团队活动质量评估”两个部分。在代码质量评估部分，BetterCode以文件为单位将静态代码文本转化为一棵抽象语法树，通过遍历语法树来收集与质量指标相关的代码特征，进而达到评估代码质量的目的，像传统代码质量控制工具一样指出项目中所有代码质量问题。

在团队活动质量评估部分，系统将团队开发活动定义为“某一个阶段内活动主体对项目提交的所有贡献的集合”，其中活动主体至少由一名开发者构成。系统利用项目的Git仓库，为项目当前版本中的每一个文件定位出最终编辑的责任人。凭此可以将当前每一个代码质量问题以文件粒度聚合后，划归到团队内具体的开发者身上，明确团队责任划分。系统除了将开发者作为评估对象外，还将在一个阶段内修改了同一文件的多个开发者作为一个开发团体，把评估对象扩展到了这类开发团体，便于掌握团队活动对项目质量的影响。

BetterCode通过追溯某一阶段内项目文件的演化过程，定义了这一阶段内所有的开发活动，并利用文件与开发者之间的归属关系，将这一阶段产出的代码质量与定义的开发活动关联起来，反映这一阶段内团队的开发活动的质量，帮助管理者进行团队决策。

1.2　国内外研究现状

代码规模、复杂度的增加意味着引入更多问题代码，导致产品缺陷的风险的增加，降低产品质量。Leveson et al.早前的研究阐述了低质量的软件不但会增

加后期项目开发、维护、升级的成本，甚至会危及人员生命、财产安全 [9]。代码质量度量是许多支持软件开发、维护的方法的核心，可以被用于代码味道检测 [10, 11]、重构推荐 [12, 13] 和缺陷预测 [14–16]等领域。

为了提高软件质量，减少问题代码的引入，Schermann et al. [1]和Vassallo et al. [6]研究了开发团队在开发过程对代码质量的控制情况。Vassallo et al.的研究表明，在敏捷开发领域，开发团队会在持续集成工作流中利用持续测试工具收集多个质量维度的质量信息，提高项目代码质量 [6]。

Vassallo et al.同时指出，当开发团队使用持续集成来保障软件产品质量时，除了需要代码仓库和进行项目构建的服务器之外，还应该有独立的代码质量控制服务，对开发者提交的代码进行质量评估 [6]。

代码质量控制服务依赖于代码质量控制工具，现有的代码质量控制工具有：来自开源社区GitHub的第三方代码质量评估系统Codacy、Code Beat；代码质量检测软件Check Style 、PMD、SonarQube。它们可以从多个质量维度评估代码好坏，帮助开发团队在开发过程中检测代码中不规范、不合理、不符合软件工程要求的“坏味道”部分，帮助开发团队掌握代码变动对代码质量造成的影响。MacDonald [3]和Rutar et al. [20]在研究中建议开发者使用多种代码质量控制工具，来获取更加全面的分析结果。

大量的分析数据，对开发团队利用分析结果制定决策产生了挑战。现有的研究表明，在获得了大量的代码质量指标数据后，开发团队反而无法从庞大的分析结果中获取有价值的信息，帮助团队开发者制定决策、分配任务。Vassallo et al.在研究结果中提到上述现象甚至会打击团队进行代码质量控制的积极性，并指出在开源社区活跃的开发团队中，工作流程中落实了持续代码质量控制的团队只占到了少数 [6]。

为了帮助开发团队更好的理解代码质量度量结果，MacDonald尝试引入机器学习的方法来筛选对团队价值最大的度量指标 [3]。Bassi et al. [4]和Wallace et al. [7]的研究指出庞杂的代码质量分析指标在实际应用中存在局限性，不能够将质量数据转化为能够为开发团队提供决策帮助的信息。

Bassi et al.尝试通过一系列代码质量度量指标来评估项目每一次提交的质量，以此反映开发者提交内容对项目质量产生的影响，过程中采用了简洁、明了、客观、健壮并且易于观测的度量指标 [4]。

West将软件度量指标分为了过程指标、项目指标和产品指标，过程指标关注软件开发过程，项目指标关注项目状态，产品指标用于度量产品在任何阶段开发阶段的属性 [21]，本课题中讨论的各个代码质量指标属于产品指标。

除了上述关于代码质量指标的相关研究外，为了建立能够直接辅助团队决

策的代码质量控制系统，本项目还需要整合具体的代码质量控制方法、开发团队管理方法和开发者开发行为特征。然而根据收集、查阅的资料表明，现阶段有大量关于代码质量控制、开发团队管理、开发者开发行为的研究，但是据已有的文献所知，没有人曾尝试过将开发者行为和开发者贡献代码结合起来用于辅助团队管理，因此本项目只能在上述三个领域已有研究成果的基础上做出新的尝试。

在软件工程中，代码评审和软件测试能够有效控制代码质量。代码评审是保证代码质量的一个重要手段，在七八十年代的开发过程中非常流行，Bacchelli et al.对代码评审在当今软件开发中的作用进行了实验研究，发现代码评审保障代码质量的功能被弱化，而更多地承担起开发人员之间知识传递、增强信息流通的功能 [17]。

软件测试是控制软件质量必要且有效的手段，但是即使代码已经通过了软件测试过程，也不能说代码不存在任何问题。Hovemeyer et al.的研究发现，即使是通过了测试的功能模块依然可能存在大量的bug [18]。为了提高测试效率高传平等人在实验中总结了软件故障模型，利用抽象语法树对软件代码进行静态分析，通过自动化的手段定位故障 [19]。在本项目中，系统利用开源工具将Java源代码转化为语法树，之后从语法树中抽取代码的质量特征。

Pantiuchina et al.在评估代码质量时，利用存储在版本控制系统中的提交记录来获取与开发者相关的信息，尝试从开发者的视角评估代码质量 [22]。Git作为当前流行的版本控制系统 [23]，可以从中获取相当详细的系统演化信息，并且由于其在软件产品中广泛使用，基于Git的分析工作可以获得丰富的数据源。系统利用了Git仓库能够记录源文件历史和其贡献者的特点，将文件和开发者关联了起来，用文件质量评估开发者的贡献质量。

由于Git天生支持分布式开发的特性，研究者们常将其拿来研究软件开发过程中的开发特征。陈丹等人尝试通过分析开源社区中开发者的协作行为，挖掘社区中开发者的开发特征 [24]。

Panichella et al.通过挖掘团队开发过程中使用的多种沟通渠道信息来定义开发者的合作关系，分别定义了社交网络中的关键开发者和在网络中开发者-代码变化之间的关系 [25]。

Lavallée et al.通过调查实验，提出了对软件质量产生影响的一组外部因素，实验结果表明，开发团队的组织结构、行为特征对最终软件质量会产生显著影响 [26]。依据上述研究成果，系统除了评估项目整体代码质量和开发者贡献质量外，还将文件关联的多个开发者看作是开发团体，评估这个开发团体的贡献质量。

1.3　本文的主要工作和组织结构

本文介绍了帮助管理者掌握团队活动对软件代码质量影响的团队代码质量控制系统BetterCode。系统利用Git仓库内开发者和源文件之间的贡献关系，定义了开发周期内开发者之间的关系，并且在项目代码质量度量的基础上，分别建立了开发者和代码质量、开发活动与代码质量之间的关系。系统从项目代码和团队活动两个方面评估了软件代码质量和开发者的贡献质量，制定了多粒度代码质量度量指标，建立了项目的文件生命周期模型，定义了开发周期内的开发活动，分别关联了开发者、开发活动与代码质量度量结果，帮助开发团队直观地了解项目代码质量状况、为制定团队决策提供帮助。BetterCode中用于挖掘Java项目代码质量特征的模块QualitySniper和用于Git仓库管理的模块GitManager均可独立运行于任何Java1.8版本的JVM上，帮助其他开发者在相似项目中集成、使用来实现代码质量度量或开发活动发现等功能。

本文的组织结构如下：

第一章引言，介绍了代码质量控制的相关背景和研究现状，指出现有代码质量控制工具并不能很好的帮助开发团队制定开发策略，阐述了基于Git仓库的团队代码质量控制系统的研发目的和意义。

第二章技术概述，介绍了基于Git仓库的团队代码质量控制系统中各个质量维度下代码质量指标的详细定义和系统中所用相关技术。

第三章基于Git仓库的代码质量控制系统的需求分析与架构设计，分析了系统的功能性需求和非功能性需求，并且基于非功能性需求对系统架构的设计和实现进行了详细的讨论。

第四章基于Git仓库的代码质量控制系统的详细设计与实现，针对系统的功能性需求，展开讨论了承担系统功能的各类模块的详细设计与实现，并阐述了设计与实现的合理性。

第五章系统测试，对系统进行了全面的测试，验证系统功能是否满足了需求分析结果，确认了系统功能被正确实现，保障了系统的可用性、可靠性。

第六章总结与展望，总结了系统实现过程中所做的工作，并对系统未来的迭代工作做了进一步展望。

第二章　技术概述

2.1　代码质量度量指标

评估代码质量作为团队代码质量控制系统的基础，是评估团队开发活动的第一步。在评估项目代码质量之前，需要建立一套可靠的度量体系。代码质量度量体系的基础是一系列可以真实反应代码本身特点并且与代码质量相关联的代码特征 [27]，通过量化这些特征，将之作为代码质量的度量指标，就可以直观地反应当前代码的质量状况。

BetterCode的代码质量度量指标制定过程中参考了Rawat et al.提出的良好的度量指标特征：可测量、客观、良好的可拓展性、较低的获得成本、可检验以及强壮等特性 [28]。

在由Java等面向对象语言开发的程序中，软件由一个个功能模块组装而成，开发工作往往也依据一个或一组相似功能来安排。根据功能细化的程度，功能模块可以是一个函数，也可以是一个类，甚至可以是一个子项目。因此，系统以“质量指标所关注的代码特征”和“模块承担一个独立功能”为参考设计每一个代码质量指标的度量粒度，建立了多种粒度指标相互补充的度量体系。

最终系统从代码规模、测试复杂度、变量控制、编程规范、和重复代码等五个代码质量维度，制定了如下度量指标。

2.1.1　代码规模

代码规模往往通过代码文件相关的语句数量反映，更进一步可以通过代码行数描述。代码规模体现了软件代码的可读性、可理解性以及可维护性等特性，合理的代码规模可以降低开发者阅读、理解代码的难度。在Java项目中，开发者往往将实现某一独立功能的代码块组织在一个函数中，函数越长，对开发者理解函数功能的成本就会增加，Buse et al.指出可读性会影响代码质量 [29]。在质量控制系统中，以函数作为代码规模维度指标的度量粒度。

系统在代码规模维度定义了如下三个函数粒度的代码质量指标：（1）可执行代码行数是模块中所有可执行语句所占的行数，最直接、客观地反映了模块规模，过长的代码块会明显降低模块的可读性和可维护性；（2）合理的注释比例可以显著提高模块的可理解性，但是过多的注释并不能提高代码的可理解性，反而会降低函数的可读性。在开发实践中 [30]建议开发者将不会用到的代码删

除，而不是将之注释掉。系统通过检测函数中异常的注释比例来发现可能存在问题的功能模块；（3）空函数是函数体为空的函数，没有做任何动作的函数没有实现任何功能，不应该出现更不应该被其他模块使用。

2.1.2　测试复杂度

测试复杂度描述了对一个模块充分测试的复杂程度。在Java开发中最常进行的，同时也是贯穿开发阶段始末的是函数的单元测试。在软件工程中会将测试分为白盒测试和黑盒测试 [31]，相较于白盒测试，黑盒测试相关的代码特征更不易收集，且较难判断函数是否容易进行黑盒测试进行。而白盒测试讨论代码覆盖的相关指标则相对更易通过分析模块的代码特征获得。

对模块进行白盒测试时，存在不同测试程度的测试方案，常见的测试方案有语句覆盖，分支覆盖和路径覆盖。其中语句覆盖的程度最弱，只需要将模块中的所有语句执行一遍即可，同时成本也是最低的；分支覆盖的程度强于语句覆盖，测试覆盖了模块中的所有逻辑分支即可；路径覆盖的程度最强，需要覆盖模块中所有存在的路径，同时也是成本最高的一种测试方案。不同的测试方案对模块中控制节点的处理方式不同，导致了测试强度、测试成本的不同，对控制节点产生的路径考虑地越细致，测试越全面，同时测试成本也会增加。即模块中的控制节点的数量直接影响了测试方案的复杂程度。

系统定义了圈复杂度和控制节点最大嵌套层数两个函数粒度的评估指标，通过收集相关代码特征来评估函数模块的测试复杂度。

圈复杂度的概念来自于图论中对图的复杂度测量 [32]，由连通图中所有线性独立路径的数量表示。用在反映代码复杂性中，表示由于控制节点而在代码中出现分支时，所有可执行路径的数量。圈复杂度越大，说明可执行路径越多，开发者需要理解和测试函数所需要的成本也越高。在函数中通常利用控制节点的数量来计算函数中存在的线性独立路径数量。

需要实现复杂功能时，控制节点之间不可避免地存在相互嵌套。但是当代码中存在大量控制节点层层嵌套时，会降低代码的可读性和可测试性。工业实践中 [30]会建议使用卫语句或其他手段来减少嵌套。

图 2.1描述了在一个代码块中控制节点对代码复杂度的影响情况，其中节点A、B、C、D均为普通的可执行语句。在这个代码块中，由于存在三个控制节点，所以代码块的圈复杂度为4。由于B、C所处的控制块和A所处的控制块均处于根控制节点之下，因此代码块的最大嵌套层数是2。

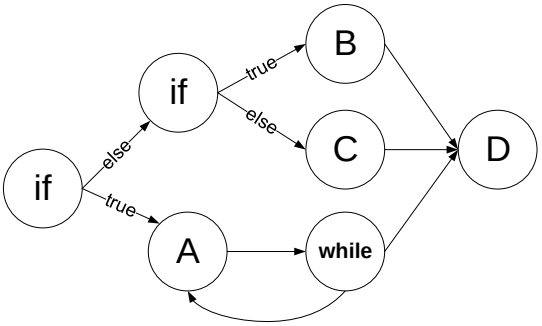

图 2.1: 代码块中的控制节点示意图

2.1.3　变量控制

函数在执行过程中，函数本身及其上下文需要维护的变量数量对函数的可理解性、可维护性和函数的执行效率都有着直接的影响。系统从函数调用和函数执行两个方面分别定义了如下两个函数粒度的代码质量度量指标：（1）函数调用时所需要的参数数量是函数上下文需要维护的变量数量的一部分，过多的函数参数会降低代码的可理解性、可维护性；（2）局部变量数量：函数的局部变量个数是在函数运行过程中需要维护的变量个数，过多的局部变量会降低代码的可读性、可理解性和可维护性。

2.1.4　编程规范

系统依据软件工程实践，挑选了一批工业界倡导的编码规范，从中选取了若干容易引入缺陷的编程特征，通过提取代码特征，制定特征收集策略，并参考了Rawat et al.给出的指标选取建议 [28]，当前版本的系统选取了“数据实体类的hashCode和equals函数需要同时重写”规范作为这一维度的质量指标。在工业界的Java开发中，类中的成员函数equals和hashCode必须同时重写 [30]。

2.1.5　重复代码

重复代码往往来源于代码编写过程中的“复制粘贴” [33]。重复代码是软件开发过程中最基本的代码复用方法 [34]，也意味着重复部分存在被抽象的可能，大量的重复代码会降低代码的可维护性和健壮性。在项目中重复代码往往存在于不同文件之间，因此重复代码这一指标被定义为文件粒度的度量指标。

检测项目中重复代码的方法很多也很成熟，Baxter曾使用抽象语法树检测项目中存在的重复代码 [35]。在本系统中，代码质量控制模块通过调用PMD的重复代码检测模块（简称CPD） [36]，解析PMD的元数据结果，集成了重复代

码检测功能。系统能够检测一定规模范围的重复代码块，把代码中每一句可执行语句中的字符，包括变量类型、变量名、保留字符等作为一个个标识，依据指定的标识数量，从文件集合中找出代码规模与指定标识数相近的重复代码。

2.2　基础技术介绍

本节会详细介绍系统开发过程中使用到的一些关键技术。系统采用前后端分离的设计模式，前后端之间采用RESTful风格的HTTP请求进行交互。服务端基于Spring Boot框架、可视化模块基于前端框架Vue，二者仅作为开发过程中的工具，其相关介绍在这里不做赘述。

2.2.1　抽象语法树

抽象语法树以树的形式来表示源代码的语法结构。语法树中的每一个节点代表了在相应代码块中出现的一个组件，这个组件可以是一个变量，也可以是一个保留字符或者是一个运算符号。经典的抽象语法树只能保留与代码结构相关的细节，无法还原出代码中出现的每一个细节。

抽象语法树将源代码文本转化为树形数据结构，由于可以使用访问树的算法来处理语法树，因此极大地提高了相关代码处理工作的效率，Neamtiu et al.曾利用抽象语法树快速比较了软件不同版本之间的差异来发现项目的演化进程 [37]。

系统的代码质量控制模块QualitySniper利用Java开源工具JavaParser，获取目标源文件的语法树。由Java代码转化的语法树会以文件或整个代码块作为整棵树的根，根节点的每个子节点代表了代码块中一个类的定义。当文件中除主类外还存在其他类的声明时，文件对应的语法树结构的一级子节点会由一棵子树转化为一个语法树森林。从根节点开始，通过遍历语法树就可以访问到文件中的每一个代码块，并收集到能够反映代码质量指标的代码特征。JavaParser是一款在GitHub社区开源的轻量级Java代码分析处理工具，实现了从Java1.0到Java12语法的全覆盖，能够将大部分Java代码块处理成一棵语法树的同时保留代码块中的所有细节。

图 2.2是Java文件HelloClass.java中HelloClass主类的部分代码转化后的语法树结构。语法树根节点的孩子包括了类名、类修饰符和类中的两个成员。图中的圆角矩形代表语法树中节点的类型，椭圆形节点表示对应代码块中的内容标识符。

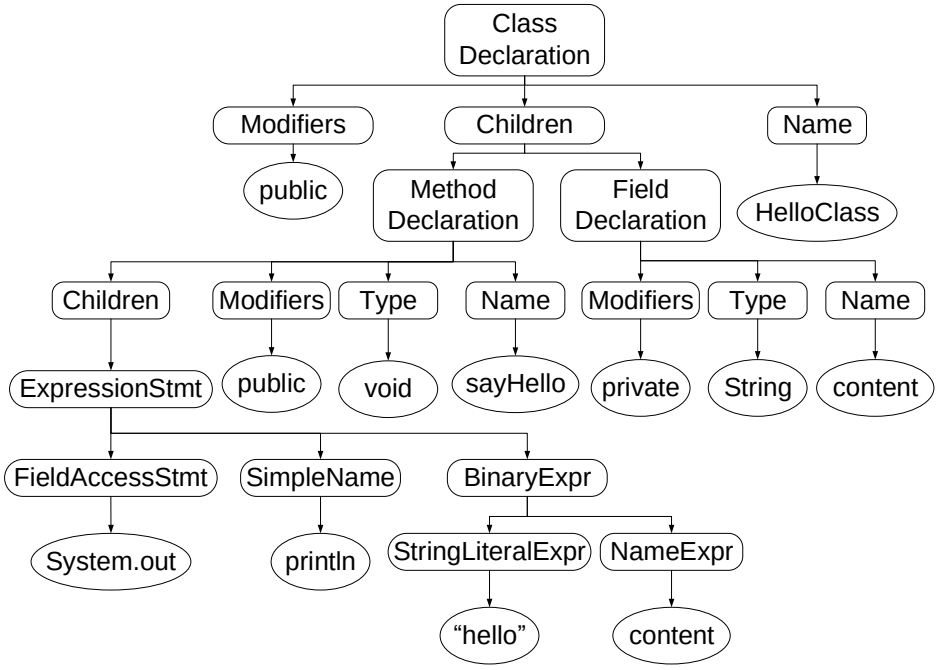

图 2.2: HelloClass类对应语法树

2.2.2　版本控制系统

版本控制系统记录了项目开发过程中每一个被纳入管理的文件的变更历史。如今为了支持以团队协作形式进行的大型软件项目开发活动，开发团队通常会选用Git来控制项目版本和支持分布式开发活动。

被Git管理的项目目录被称为Git仓库，其中记录了项目的版本信息。开发者对项目进行的每一次修改都需要通过“提交”操作将更改记录到Git仓库中，每一次提交都会产生一个项目“版本”，这个版本描述了当前项目的状态信息并且其中大多数是非正式的，并不会发布被用户使用。在一个被Git管理的项目中，Git把开发者们的一个个提交串联起来构成一个有向无环图来表示整个项目的演化历史。

在Git的对象库中存在四种基本类型的对象，他们分别是：使用二进制大对象描述文件内容的“块对象”，记录块对象和文件元数据对应关系的“目录树对象”，保存版本变化元数据和对应目录树对象的“提交对象”和为任意对象分配可读名字的“标签对象”。这四种基本对象类型共同构成了Git高级数据结构的基础。

图 2.3描述了Git管理仓库提交历史的过程，图中的一个圆形表示一次提交，一个三角形表示一个树对象，提交会指向一棵特定的树对象，这棵树对象在提

交时在仓库中被创建。图中的每一个矩形表示一个块对象，每个圆角矩形表示一个分支名，当前最长的提交链的分支名为master [38]。

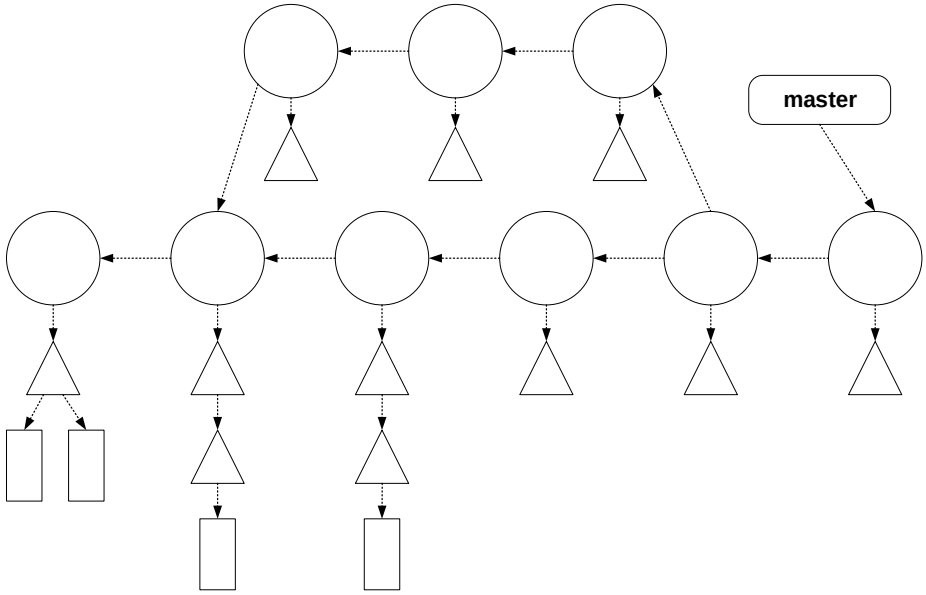

图 2.3: Git版本控制示意图

在Git中每进行一次提交，都会自动创建一棵目录树对象来记录当前提交的内容块对象和对应文件名的关联。由于每个提交都有唯一的“提交者”与之对应，保证了可以利用Git找到系统中任何一个文件在某一次提交时的内容和与内容有直接关系的最终提交者的信息。

系统的Git仓库管理模块GitManager集成了由Eclipse基金会提供的开源工具JGit。JGit是一款由Java编写的功能相对完善的Git管理工具，支持使用者在Java程序中访问Git项目仓库，将访问结果处理为可在Java中直接操作的数据对象实例。相比于在Java中通过命令行界面（以下简称CLI）访问Git仓库，减少了文本处理、并发控制等困难，虽然JGit本身只支持部分Git的功能，而CLI方式则可以提供更大的操作空间，但是JGit所提供的功能已经足以支撑系统实现仓库管理功能。

2.2.3 MongoDB

MongoDB是由美国MongoDB 公司开发的一款开源分布式文档数据库。与基于数据关系模型的关系型数据库 [39]不同的是，MongoDB被认为是非关系型（NoSQL）数据库。在关系型数据库看来，现实世界的事物实体和实体之间的关系都可以使用关系模型来表示，并将实体和实体关系按照预先设计好的模式

以一张张二维表的形式保存在数据库中。非关系型数据库则是以数据原有的组织形式，将数据存储在数据库中，这个过程中不需要去设计表的模式 [40]，因此无模式数据也是非关系型数据库的一大特征。如今非关系型数据库的种类大致可以分为如下四类：HBase所属的面向列的数据库，MongoDB所属的文档数据库，ZooKeeper、Redis所属的键值对数据库，Neo4j所属的图数据库。

MongoDB这一类文档数据库被用于存储和管理文档类型的数据，即半结构化数据。MongoDB采用了一种类似JSON的二进制文档组织形式BSON。在MongoDB中，每一条数据被认为是一个“文档（document）”，文档内容的组织形式和JSON结构类似，由一组键值对构成。一组相关的文档在数据库中可以被组织为一个“集合（collection）”。但是因为集合是无模式的，所以每一条文档不需要保持一致的数据结构，因此MongoDB中同一个数据集合下的不同文档可能存在不同的数据结构。虽然MongoDB的文档对象不能超过4MB，但是数据库对文档的数量没有限制，因此，数据库可以很容易的支持分布式扩展。

系统使用MongoDB作为数据存储、管理工具，在编程过程中使用Spring Boot提供的MongoDB依赖包，可以将代码质量分析、评估和仓库信息挖掘过程中涉及到的Java数据类映射为数据库中的集合，将数据对象映射为集合中的文档。由于MongoDB中数据组织形式和JSON类似，因此通过合理的数据结构设计，可以简化系统工作过程中数据查询后数据组装的工作。

2.2.4　RabbitMQ

RabbitMQ是一款轻量级开源消息队列实现，作为网络应用层的消息中间件，RabbitMQ支持AMQP协议。RabbitMQ服务程序使用Erlang编写，同时拥有良好的客户端库可以支持当前主流的编程语言。

消息队列一般由三个最基本的元素组成，分别是：负责产生数据的生产者、负责传递数据的消息队列和负责消费数据的消费者。RabbitMQ的核心思想是生产者不直接向消息队列传递数据，而是通过一个被称为交换器（exchange）的组件进行数据传递 [41]。在这种设计思路下，RabbitMQ支持六种消息队列模式，分别是：

简单模式，最简单的队列模式，由一个生产者、一个消费者和一个消息队列构成；

工作队列模式，由一个生产者、一个消息队列和多个消费者构成。消息队列会将一组任务平均分发给多个任务消费者，避免短时间接受大量任务造成任务堵塞；

发布/订阅模式，由一个生产者、多个队列和多个消费者构成。交换器的类型为fanout，交换器会将任务广播给所有已经与之绑定的消息队列 [42]。

路由模式，由一个生产者、多个队列和多个消费者构成，交换器类型为direct。在路由模式下，队列与交换器绑定时会指明队列的路由标识，同时生产者产生的消息也会指明一个路由标识，消息会被交换器发送到路由标识一致的队列中。

主题模式，由一个生产者、多个队列和多个消费者构成，交换器类型为topic。主题模式中，交换器绑定的路由标识由一组词构成，交换器将与队列路由字符串模式匹配的消息发送到相应队列。

RPC模式，RabbitMQ的RPC模式支持客户端向远程计算机发起计算请求并获取计算结果。

系统中，由于耗时任务种类较少，因此使用路由模式，为每一个耗时任务建立了独立的任务队列和独立的任务消费者。

2.3　本章小结

本章介绍了系统中主要涉及的底层技术概念和工具，以及它们在系统中所起到的作用。基于这些技术和工具，建立起了系统的上层功能，并为系统的性能优化、功能拓展提供了方向。

第三章　基于Git仓库的团队代码质量控制系统的需求分析与架构设计

3.1　系统概述

BetterCode是一款基于项目Git仓库来度量项目代码质量、挖掘开发者和代码之间关系、评估团队开发活动对代码质量影响的代码质量控制系统。为了达到这一目标，系统需要实现如下四类功能：负责提取代码质量特征的代码质量控制功能，负责挖掘项目Git仓库历史信息的Git仓库控制功能，负责制定评估标准、评定质量特征和定义、评估开发活动的团队代码质量评估功能和负责沟通系统和用户、处理系统业务逻辑的业务处理功能。

3.1.1　代码质量控制功能

代码质量控制功能包含了处理源文件和提取代码特征这两个基本功能。代码质量控制功能保证系统能够将Java源码转化为一棵能够表达所有代码细节的语法树，通过遍历转化后的语法树，收集代码的质量特征完成对Java源码质量的静态分析。

代码质量控制功能不包含代码质量特征数据评估功能，实现质量控制功能的模块应该按照约定好的规则从语法树中收集特定的语句节点，把代码语句处理成能够描述特定代码质量维度的代码特征，并将这些属于同一份文件的质量特征数据组织成一份“元数据”。元数据并不对项目的代码质量下任何结论，而是直接反映了在这一质量维度下，项目中对应模块的代码状态。如，对于代码规模维度的“可执行代码行数”指标，其对应的元数据包含了函数代码块中所有可执行语句在源文件中的位置信息，但是元数据并没有评判函数的可执行代码行数这一特征的优劣，而是把“指标定性”这一工作交给功能的调用者来完成，在本项目中这一工作是交由质量评估功能的相关模块来完成。

3.1.2　Git仓库控制功能

Git仓库控制功能包含Git仓库管理和历史信息挖掘两个基本功能。在分布式开发活动中，尽管开发团队中的每个开发者都有一份保存在本地的项目仓库副本，但是为了同步开发进度，还必须在团队的Git服务器上设置一个远端的中央仓库，在中央仓库中保存有最完备的团队开发活动记录。Git仓库管理功能简

化了JGit获取远端仓库的流程，通过登陆Git服务器或使用身份验证标识符就可以在Java代码中执行远端仓库的获取操作。

在项目的Git仓库中，能直接反映开发活动的数据就是开发者在仓库中的提交记录。提交记录包括了提交内容的作者、提交者、提交时间、与提交对应的文本描述和修改内容列表。Git仓库控制功能提供了收集项目Git仓库中所有的提交记录和特定项目版本下文件内容提取的功能。Git仓库控制功能的原则和代码质量控制功能一致，仅提供Git操作和收集信息，不对收集到的数据做定性或定制化的处理，而是选择将其保存为元数据，描述项目某一个快照版本的开发状态。

3.1.3　团队代码质量评估功能

团队代码质量评估功能包括了：数据定性、开发活动发现、代码质量评估、活动质量评估等功能。质量评估功能需要依据指定的代码质量评定标准对代码质量特征元数据进行定性处理，为项目的每一个质量指标给出评估结果。

把开发活动和代码质量关联起来的关键在于如何定义活动中文件和一个或多个开发者之间的关系。为此需要从提交记录中抽取项目中每个文件的演化历史，并以此建立文件的生命周期，记录文件的演化过程，将文件的每一次变更和开发者直接关联起来。利用这种关联关系，系统可以获得每位开发者在某一版本中编辑的所有源文件，并用源文件的代码质量来表示开发者在当前版本的开发质量。

在当前流行的敏捷开发模式下，团队往往以一个迭代周期为单位来安排开发活动。因此开发活动具有时间跨度，只根据某一个版本快照无法还原出真实的团队活动状况，也就无法对活动做出合理的评估。在Git中一个提交记录代表项目的一个版本快照，项目在一个迭代周期中的经历就是从一个稳定的版本演化到另一个稳定版本。质量评估功能选取迭代开始和结束的提交作为开发周期的起点和终点，从提交历史纪录中截取一组特定的提交记录。同时，以“是否向同一文件贡献代码”为标准定义这一阶段的临时协作小组，将被多个开发者贡献的文件的代码质量归属到对应的临时协作小组中，并由这些开发小组来代表开发活动。最终评估功能可以给出这一阶段内所有开发活动的评估结果。

3.1.4　业务处理功能

系统中的代码质量控制功能、Git仓库控制功能和团队代码质量评估功能都是系统的核心功能。在系统中，核心功能只负责执行特定领域的计算任务，不需要考虑用户交互相关的业务功能，保证系统具备了良好的可扩展性、可用性

和可替换性。代码质量控制功能和Git仓库控制功能的实现模块甚至可以作为独立的分析工具，提供CLI操作方式。

系统的业务处理功能保证了系统能够与用户交互，业务处理功能包括系统可视化和业务逻辑处理两部分。业务处理功能为用户提供了图形化交互界面，关联了用户和系统核心功能。业务处理功能具备处理如下四个业务场景的能力：管理用户仓库、管理项目质量、管理项目问题和管理项目生命周期。

3.1.5　项目目标与可行性分析

BetterCode的目标是在代码质量分析的基础之上帮助开发团队快速掌握开发活动的质量，相较于现有代码质量控制工具，能够帮助开发团队更加高效地制定开发策略。为此系统应该具备三个核心功能点：评估软件项目的代码质量、定义开发周期内的团队活动和评估团队活动的质量。

项目可行性可从如下三个方面来阐述。（1）代码仓库的获取，在代码托管平台中，大多数开源社区对社区内开源的公共项目并没有设置访问限制，而对于私有代码托管平台或开源社区中的个人项目可以通过设置远程登陆密码的方法从Git服务器中获取远程仓库。（2）多人协作活动的定义，虽然项目的提交历史提供的信息无法完全还原出真实开发状态下开发者的开发活动，但是足够挖掘出在开发周期内开发者之间的协作关系 [43]，这种关系可以作为评估协作活动的质量的依据。（3）模块之间的耦合，系统以功能为边界划分各个模块，模块间使用元数据通信，模块核心功能的实现不直接依赖另一模块的函数、功能，质量评估模块通过适配器将来自下层模块的元数据转化为自身数据对象，并不直接调用下层模块的业务函数，因此系统也具备了良好的可拓展性。

3.2　系统需求分析

根据系统描述和系统目标，基于Git仓库的团队代码质量控制系统应该建立在三个核心功能的基础之上：静态分析目标项目仓库的源代码、管理目标项目的Git仓库并从中收集团队开发信息、整合数据评估团队开发活动的质量。根据负责的功能不同，可以将系统划分为三个功能模块：代码质量控制模块、Git仓库控制模块和团队代码质量评估模块。同时系统还需要具备能够调用功能模块为用户服务的业务处理功能，业务处理功能由业务模块提供，业务模块包块可视化模块和系统业务中台。

如图 3.1所示，是系统用例图，描述了系统所有的用例，其中项目仓库拉取用例是创建纳管任务用例的扩展，提取代码特征、代码质量评估、挖掘项目历

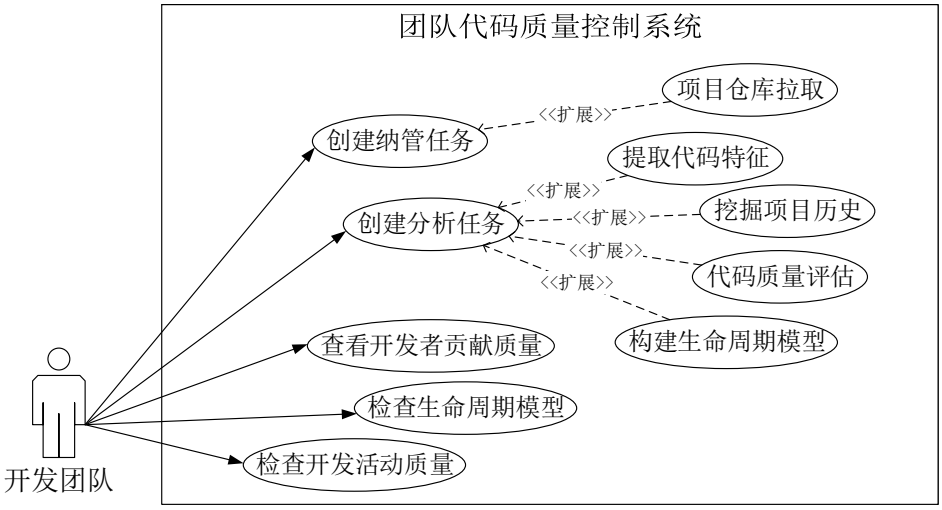

图 3.1: 团队代码质量控制系统用例图

史和构建生命周期模型用例是创建分析任务用例的扩展。上述五个用例都是系统的自动化流程，与系统的核心功能模块相关。系统用例图中的用例会在接下来的章节中，根据所涉及到的模块进行详细描述。

3.2.1　业务模块需求分析

系统的可视化模块和业务中台处于用户和系统功能模块之间，共同组成了系统的业务模块。可视化模块为用户提供可视化操作接口，业务中台为系统处理业务逻辑，将用户请求和系统提供的功能一一对应。本节将可视化模块和业务中台作为一个整体的业务模块，讨论系统需要满足的业务需求。在系统用例中所有与用户操作相关的用例都由业务模块处理。

表 3.1是用户创建仓库纳管任务的用例描述，用户可以从自己所关联开源社区平台的仓库中选择目标仓库，也可以通过手动方式，向系统提供远程仓库url来纳管新仓库。系统根据用户提供的仓库Git地址自动化提取项目相关信息，同时用户也可以手动修改提取信息中的错误。

表 3.2是用户创建分析任务的用例描述，用户从已关联的项目仓库中选择任意仓库作为分析对象，创建已关联项目的分析任务，分析任务包括代码质量分析任务和项目历史挖掘任务。另外，当仓库纳管任务完成后，纳管任务执行者会自动为项目仓库创建分析任务。

表 3.3是查看开发者贡献质量的用例描述，用户可以选择某个已纳管到自己账号下的项目，查看项目的代码质量和项目开发者的贡献质量。如果用户选择了一个未关联项目，系统会提示项目没有被纳管，并建议用户纳管目标项目。

表 3.1: 创建纳管任务用例描述

项目	内容描述
ID	R1
名称	创建纳管任务
参与者	用户/开发团队
用例描述	用户将新项目仓库与自己的账号关联
前置条件	用户需要登陆，并且提供了目标项目的远程仓库信息
后置条件	关联用户和目标仓库；任务加入任务队列
正常流程	1. 用户从仓库列表中选择待纳管的项目 2. 系统提取项目仓库基本信息 3. 用户核对系统提取的仓库相关信息是否正确 4. 用户确认创建纳管任务 5. 系统创建项目仓库纳管任务 6. 关联用户和目标仓库
扩展流程	1a. 用户直接提供待纳管项目的远程地址 1a.1 执行步骤2 3a. 如果系统提取的仓库信息不正确： 3a.1. 删除错误信息 3a.2. 填写正确的仓库信息，执行步骤5

当因为关联项目异常而无法获得质量结果时，用户可以选择重新纳管项目仓库或对项目仓库进行重新分析。

表 3.4是检查生命周期模型的用例描述，登陆用户可以选择任意已经关联的项目仓库，向系统发起查询项目文件生命周期的请求，系统验证用户和目标仓库已经关联且目标仓库已经被分析后，返回目标仓库中指定文件从被加入仓库中开始至项目被纳管时所有的修改记录。

表 3.5是检查开发活动质量的用例描述，用户通过指定开发周期的起始提交ID，向系统发起开发活动贡献质量查询请求，系统根据用户提交的ID返回在这一阶段内所有的开发活动组的质量贡献。

3.2.2　代码质量控制模块需求分析

代码质量控制模块负责对源代码内容进行静态分析，并获取能够代表源码质量的代码特征元数据。为了完成上述功能，质量控制需要具备以下功能点。

（1）处理代码文本，模块能够根据传入的源代码，利用JavaParser将对应源代码处理为一棵语法树，供后续代码特征提取步骤使用。根据实际使用场景的要求，需要至少保证能够处理如下两种应用场景，能够直接处理源代码的输入流和通过文件名处理文件系统中的文件内容。

表 3.2: 创建分析任务用例描述

项目	内容描述
ID	R2
名称	创建分析任务
参与者	用户/开发团队
用例描述	用户创建新项目的分析任务
前置条件	用户需要登陆，并且已经与目标项目关联
后置条件	将分析任务加入任务队列
正常流程	1. 用户选择特定的已纳管项目 2. 系统验证仓库是否已经与用户关联 3. 系统创建已纳管项目的分析任务
扩展流程	2a. 用户选择了未纳管的项目仓库，提示异常信息，返回步骤1 3a. 首次纳管时由纳管任务消费者创建项目分析任务

（2）提取代码特征，模块遍历处理后的代码语法树，提取代码质量指标对应的代码特征。各个代码特征之间应该尽可能地保持独立、减少依赖关系，降低未来在项目升级过程中添加、删除代码质量指标或添加新的代码质量分析工具的风险。

（3）传递特征数据，代码质量控制模块的最终产出是一组用于描述源文件代码质量维度的质量特征数据。

（4）模块调用，提供一组通用的代码质量特征提取接口作为外部模块调用功能的API。同时，提供命令行界面来保证模块可以作为独立的代码质量度量工具被用户使用。

代码质量控制模块是系统用例中提取代码特征用例的具体执行模块，当分析任务建立完毕后，分析任务中的代码质量分析子任务首先由质量控制模块处理。表 3.6是系统用例中“提取代码特征”用例描述，代码质量分析任务的消费者从消息队列中获取任务后开始执行分析任务。之后由质量控制模块抽取项目的代码质量特征元数据，最后质量分析消费者调用质量评估模块的数据适配器将处理后的结果保存至数据库中。在任务执行前后由任务消费者分别更新质量分析子任务的状态为“分析开始”和“分析结束”。

3.2.3　Git仓库控制模块需求分析

Git仓库控制模块负责操作系统中所有被管理的Git项目仓库，和收集项目的演化历史信息。为了完成上述功能，模块需要具备以下功能点。

（1）把目标仓库纳入管理，系统分析项目代码质量和开发活动的前提是系统可以找到对应的代码仓库。针对不同的使用场景，模块需要具备应对以下场

表 3.3: 查看开发者贡献质量用例描述

项目	内容描述
ID	R3
名称	查看开发者贡献质量
参与者	用户/开发团队
用例描述	用户查看关联项目的代码质量和项目开发者贡献质量
前置条件	用户需要登陆，并且关联项目已被分析
后置条件	返回查询结果
正常流程	1. 用户选择已分析的关联项目 2. 系统检查用户是否和项目已经关联 3. 系统返回项目代码质量报告和开发者贡献质量报告
扩展流程	2a. 用户选择了未纳管的项目仓库，提示异常信息，返回步骤1 3a. 质量报告获取异常，或无法获取项目质量报告： 3a.1. 向用户反馈异常信息 3a.2. 建议重新建立分析任务，返回步骤1

景的能力，在命令行界面中使用时，通过项目根路径将项目仓库纳入管理；在线上环境下，通过指定项目远程仓库地址来获取项目并将项目纳入管理。

（2）项目仓库非侵入式管理，对于纳管的项目仓库，模块不应该执行侵入式的操作，即模块不修改项目内容并且不将修改内容提交到项目仓库。系统对纳管项目的“写操作”应严格控制在“拉取”、“克隆”操作范围内。

（3）提取项目修改历史，模块通过遍历项目仓库的提交历史纪录，还原出项目的演化历史。并且能够获取特定项目版本，通过回溯提交记录，获取某一特定提交版本的文件内容。

（4）模块调用，提供一组通用的Git仓库管理接口作为外部模块使用模块功能的API，同时，提供命令行界面来保证模块可以作为独立的Git仓库管理工具被用户使用。

Git仓库控制模块负责项目仓库拉取用例和挖掘项目历史用例。当纳管任务建立后，由Git仓库控制模块的仓库管理功能负责执行纳管任务；当分析任务建立后，模块的项目历史挖掘功能负责执行分析任务中的项目历史挖掘子任务。

表 3.7是项目仓库拉取用例的详细描述，在纳管任务创建后，仓库纳管任务消费者调用Git仓库控制模块的仓库管理功能来获取项目的远程仓库，模块依据任务消费者提供的信息为项目仓库建立本地路径，并将仓库拉取到指定的本地路径下。在任务执行过程前后，任务消费者分别修改任务的状态为“运行”和“结束”，当任务执行失败时，模块返回相应的错误码，任务消费者依据错误码修改任务状态。

表 3.4: 检查生命周期模型用例描述

项目	内容描述
ID	R4
名称	检查生命周期模型
参与者	用户/开发团队
用例描述	用户查看项目中某一文件的生命周期
前置条件	用户需要登陆，并且关联项目已经被分析
后置条件	返回查询结果
正常流程	1. 用户选择已分析的关联项目 2. 系统检查用户是否和项目已经关联且项目已经被分析 3. 用户选定目标文件 4. 系统返回目标文件生命周期
扩展流程	2a. 系统验证失败，提示异常信息，返回步骤1

表 3.8是挖掘项目历史用例的详细描述，当分析任务创建后，分析任务中的项目历史挖掘子任务的消费者调用模块的项目历史挖掘功能。模块通过访问项目的Git仓库，获取项目的提交历史集合，并且依据两个提交记录间的承接关系，收集两次提交之间的变更实体集合，并将所有提交的变更实体集合整合为元数据交由评估模块处理后保存到数据库中。在任务执行前后，任务消费者分别修改任务状态为“挖掘开始”和“挖掘结束”。

3.2.4　团队代码质量评估模块需求分析

团队代码质量评估模块是系统的核心模块，负责实现处理项目分析元数据、定义质量指标评估标准、评估代码质量特征数据、定义和发现团队开发活动、定义开发活动和代码质量的关系和评估开发活动的质量等核心功能，模块包含了如下功能点。

（1）数据收集与转化，负责接收来自下层代码质量控制模块和Git仓库控制模块的数据，并将收集到的数据转化为自身的内部数据结构。

（2）定义代码质量指标评估标准，模块需要制定默认的代码质量特征评估规则，同时支持根据实际情况自定义配置指标评估规则。指标评估规则主要包括：根据每个指标的实际代码特征统计情况，制定统计结果的严重性分级标准，任意模块的任意指标成绩由其严重性等级决定；制定每个指标的严重性等级所占该指标总成绩的权重；制定项目的综合成绩评估标准。模块能够根据质量指标评估标准评估来自质量控制模块的代码质量特征数据。

（3）使用处理后的项目提交历史记录建立当前项目版本下所有文件的生命周期模型，以文件粒度描述在一定阶段内开发活动对文件造成的变化。

表 3.5: 检查开发活动质量用例描述

项目	内容描述
ID	R5
名称	检查开发活动质量
参与者	用户/开发团队
用例描述	用户查看在某一开发阶段项目开发活动的贡献质量
前置条件	用户需要登陆，并且关联项目已被分析
后置条件	返回查询结果
正常流程	1. 用户选择已分析的关联项目 2. 系统检查用户是否和项目已经关联 3. 用户指定开发周期的起点ID 4. 系统验证提交ID的格式 5. 获得项目开发周期内开发活动的贡献质量
扩展流程	4a. 当指定的提交ID与仓库最近提交ID相同时，返回项目最新版本的开发者贡献质量

（4）模块能够通过项目提交记录建立起开发者和项目文件之间的关联，由此间接定义开发主体和代码质量之间的关系。同时还可以挖掘某一特定阶段内的提交历史，利用在某一提交版本下开发者和被修改文件唯一对应的特点定义开发者之间的协作关系。

团队代码质量评估模块负责构建生命周期模型用例和代码质量评估用例。构建生命周期模型和代码质量评估分别在项目历史挖掘任务和代码质量分析任务完成后自动执行，属于项目分析任务的第二阶段。两任务并发执行，执行过程互不干扰，虽然模块在执行代码质量评估任务过程中，需要调用Git仓库控制模块获取每个源文件最终提交者的信息，但是并不需要等待历史信息挖掘任务执行结束之后再执行评估任务。

表 3.9是构建生命周期模型的用例描述。项目开发历史信息保存至数据库之后，进入项目历史挖掘任务的第二阶段，历史挖掘任务消费者调用模块，开始生命周期模型构建任务。模块利用收集到的提交历史，构建项目的生命周期模型，并将获得的项目生命周期模型保存至数据库中。在任务执行前后，由任务消费者分别修改任务状态为“构建开始”和“构建结束”。

表 3.10是系统代码质量评估的用例描述。代码质量特征提取完毕后，进入代码质量分析任务的第二阶段，质量分析任务消费者调用团队代码质量评估模块，开始代码质量评估任务。代码质量评估模块依据前一阶段收集的代码质量特征，需要完成评估每个特征数据、计算项目中每个文件的质量评分、计算项目整体评分和利用Git仓库控制模块计算每个开发者的质量评分等流程。在质量

表 3.6: 提取代码特征用例描述

项目	内容描述
ID	R6
名称	提取代码特征
参与者	任务调度队列、代码质量评估模块
用例描述	代码质量分析任务创建后，由对应任务消费者调用代码质量控制模块的功能收集代码质量特征信息
前置条件	系统中存有目标项目的代码仓库，代码质量分析任务消费者接收到分析任务
后置条件	保存经质量评估模块转化后的代码质量特征数据到MongoDB中
正常流程	1. 代码质量分析任务消费者调用质量控制模块分析接口 2. 任务消费者修改代码质量分析任务状态为“分析开始” 3. 扫描项目仓库中的Java源文件，并转化为语法树 4. 抽取语法树中的函数节点和类节点，并从节点中收集代码质量特征信息 5. 封装质量特征信息为元数据 6. 评估模块利用数据适配器将特征元数据处理后保存到数据库中 7. 任务消费者修改代码质量分析任务状态为“分析结束”
扩展流程	3a. 语法树无法转化时，记录并跳过问题源文件，继续执行下一文件的语法树转化过程

评估任务执行前后，由任务消费者分别修改任务状态为“评估开始”和“评估结束”。

3.2.5　系统非功能需求分析

基于Git的团队代码质量控制系统由三个负责不同功能的模块组合而成，为了降低未来系统维护、升级的难度，系统应该满足良好的人机交互、可靠灵活和易于维护等要求，因此系统的架构设计上应该满足如下要求。

首先，系统应该是易用的，能够提供良好的人机交互，可以及时向用户反馈用户请求的结果。

其次，系统应该具有良好的可维护性，系统中对数据的操作应该保证幂等性，模块内部的状态不会对模块功能调用产生影响，模块之间是相互独立的，被调用模块的状态变化都应该对调用模块透明。

最后，系统应该满足可扩展性，系统内各个模块之间可以自由组合，提供数据的代码质量度量模块和Git仓库控制模块可以很容易地被替换。

3.3　系统架构设计

通过系统需求分析，明确了系统所需要具备的功能性需求和非功能性需求。系统的功能性需求主要由系统的三个功能模块和业务模块承担，系统的非功能

表 3.7: 项目仓库拉取用例描述

项目	内容描述
ID	R7
名称	项目仓库拉取
参与者	任务调度队列
用例描述	仓库纳管任务创建成功后，由纳管任务消费者执行具体的仓库纳管任务，将项目从远程仓库拉取到本地
前置条件	项目纳管任务被成功创建
后置条件	本地获取项目仓库的最新开发版本
正常流程	1. 仓库挖掘任务消费者调用Git仓库控制模块 2. 仓库控制模块获取项目的远程仓库地址 3. 任务消费者修改仓库纳管任务状态为“运行”状态 4. 检查本地仓库根路径状态 5. 拉取远程项目仓库 6. 任务消费者修改仓库纳管任务状态为“结束”状态
扩展流程	4a. 本地仓库的根路径不存在 4a.1. 递归创建本地仓库的根路径 4a.2. 创建成功后跳转到步骤5 4a.3. 创建失败后结束任务，任务消费者修改任务状态为“仓库异常” 5a. 项目拉取失败结束任务，任务消费者修改任务状态为“拉取异常”并结束

性需求涉及到系统的业务模块，并对功能模块之间的协调调度提出了要求。本节从非功能性需求出发，从宏观角度设计系统的架构方案。

3.3.1　系统模块协作设计

根据系统的非功能性需求分析的结果，系统应该具备较低的响应时延、模块功能具有幂等性和较低的模块拓展代价等特点。为了达到上述效果，为系统架构做出了如下设计。

第一，要求系统能够及时响应用户请求。及时反馈用户的请求可以显著提高用户体验，为了达到降低用户请求时延的目的，首先需要分析时延的组成部分。从用户像系统发起请求到请求结果被返回给用户这段时间包括了请求和结果的网络传递时间、数据查询时间和任务计算时间。其中网络传递时间主要受网络影响，数据查询时间主要受数据库查询影响，这两部分时延可分别通过缩小数据传输体积、优化数据库查询逻辑等方式进行优化，即二者均可通过优化算法和数据结构将时延压缩到可以接受的范围内。但是还有一些用户请求如远程纳管项目仓库、项目代码质量分析、项目开发历史挖掘等请求，由于本身需要大量计算或严重依赖网络，在某些情况下需要消耗大量时间，导致无法只通过编程手段将响应时延压缩到用户可以接受的范围。

表 3.8: 挖掘项目历史用例描述

项目	内容描述
ID	R8
名称	挖掘项目历史
参与者	任务调度队列、代码质量评估模块
用例描述	历史信息挖掘任务创建后，由对应任务消费者调用Git仓库控制模块的功能收集项目开发历史信息
前置条件	系统中存有目标项目的代码仓库，项目历史挖掘任务消费者接收到任务
后置条件	保存经质量评估模块转化后的开发历史数据到MongoDB中
正常流程	1. 仓库挖掘任务消费者调用Git仓库控制模块 2. 仓库控制模块获取项目的Git仓库地址 3. 任务消费者修改任务状态为“挖掘开始” 4. 模块遍历Git仓库提交历史，获取具有承接关系的提交记录之间的变更实体，并处理为元数据 5. 模块获取初始提交包含的文件内容，并处理为元数据 6. 代码质量评估模块利用数据适配器将元数据处理后保存到数据库中 7. 任务消费者修改任务状态为“挖掘结束”
扩展流程	4a. 仓库中只包含一个初始提交，跳转至步骤5

对于无法只依赖编程手段来压缩时延的用户请求，系统引入了消息异步处理机制来及时反馈用户。系统为可能产生较长时延的任务建立了独立的任务队列，当用户向系统请求这一类任务时，系统会将请求信息保存在任务队列中，由对应的任务消费者不间断的处理任务请求，同时在接收到用户请求后及时向用户反馈请求入队的结果，当计算完毕后将计算结果异步地反馈给用户。通过在系统内添加消息队列中间件来提高系统的响应速度，增加了人机交互友好度，提高了系统的任务处理效率。

图 3.2描述了系统处理的各类请求的过程，请求类型分为耗时任务请求和简单查询请求。所有的耗时任务都需要被加入到任务队列中，由任务执行组件的特定任务消费者获取任务并执行，实际的执行过程需要调用对应的功能模块。在与耗时任务相关的系统界面，可视化模块以指定时间间隔向服务端查询任务执行状态，并将任务状态实时反馈给用户。任务执行完毕后，系统异步地将执行结果通过可视化模块展示。而所有查询请求等非耗时任务均由业务模块直接调用数据库服务，获取请求结果，并将结果同步反馈给用户。

系统中的任务调度队列就是一组消息队列，负责记录各个耗时任务的执行状态，耗时任务由业务模块的业务中台创建并添加进队列。任务执行组件是在系统核心功能模块之上的一组功能调用组件，是任务队列的消费者。每一个调用组件通过调用一个或多个功能模块，完成了一个具体的任务逻辑。

表 3.9: 构建生命周期模型用例描述

项目	内容描述
ID	R9
名称	构建生命周期模型
参与者	任务调度队列
用例描述	评估模块利用收集到的项目提交历史构建项目生命周期模型
前置条件	目标项目的提交历史被收集、处理并被保存
后置条件	将构建的项目生命周期模型保存至数据库
正常流程	1. 项目历史挖掘任务消费者调用团队代码质量评估模块 2. 任务消费者修改任务状态为“构建开始” 3. 模块获取项目基本信息 4. 模块获取项目的所有提交记录，提取提交记录中的文件实体，并按其新老文件名和变更类型确定关系 5. 模块构建文件生命周期 6. 保存生命周期数据到数据库中 7. 任务消费者修改任务状态为“构建结束”
扩展流程	无

第二，模块对外提供的功能应该具有幂等性。在系统中，在同一功能、同一计算对象的前提下，模块提供的功能运行若干次，应该获得同样的结果。即，模块对外服务应该和模块内部状态无关，同时使得模块之间保持了一个较低的耦合度。

为了保证下层模块内部状态对上层模块透明，降低上下层模块之间的耦合度，模块之间需要传递无状态数据。在系统实现中，下层模块为上层模块提供描述项目当前代码质量特征、提交历史记录的元数据。上层获取到来自下层的数据时，不需要判断下层模块的运行状态。

第三，系统应该对模块更新、替换、扩展等操作提供良好的支持。系统中模块之间的依赖关系来自于，质量评估模块在评估团队开发活动的贡献质量时需要质量控制和Git仓库控制模块提供相应的数据，其中产生了上层模块对下层模块的数据依赖和功能依赖。

在系统实现中，模块间通过元数据传递信息，上层模块只需要处理收集到的描述目标对象的属性值，不需要关心下层模块内部状态，即对上层模块屏蔽了下层模块的内部细节，下层模块完全透明。同时，上层模块的数据结构不应该依赖下层模块的数据结构，上下层模块之间通过一个层间的数据适配器从下层模块的数据中剥离出上层模块数据所需要的属性。这样可以保证系统在未来出现新的需求需要变更底层模块时，可以尽可能地减小底层模块变化对上层核心模块产生的影响。

表 3.10: 代码质量评估用例描述

项目	内容描述
ID	R10
名称	代码质量评估
参与者	任务调度队列
用例描述	评估模块对项目代码质量进行评估
前置条件	代码质量特征数据已被收集、处理完毕
后置条件	保存项目代码质量的评估结果
正常流程	1. 代码质量分析任务消费者调用团队代码质量评估模块 2. 任务消费者修改任务状态为“评估开始” 3. 模块获取代码质量特征评估指标，并评估代码质量特征数据 4. 计算文件代码质量成绩 5. 计算当前版本开发者的贡献质量 6. 计算项目整体代码质量 7. 保存项目代码质量评估结果 8. 任务消费者修改任务状态为“评估结束”
扩展流程	无

系统中，代码质量控制模块利用JavaParser将Java源代码转化为语法树后，抽取大部分质量指标的代码特征来实现代码质量分析功能，当未来需要将其替换为一个全新的模块时，只需要在上层模块中为新的下层模块添加对应的数据适配器即可。在当前系统中，重复代码检测功能的实现依赖于PMD的CPD，质量控制模块将检测结果组织为元数据交给质量评估模块的数据适配器处理。

除了需要保证上下层模块间没有直接的数据依赖外，需要为底层模块制定一套接口规范，定义统一的上下层模块交互接口。上层模块不用关心下层模块的具体实现，只需要通过统一的功能接口从下层模块获取数据，并将下层模块的数据结构转化为自身数据结构即可，便于未来拓展底层数据模块。

综上所述，这里给出了系统物理图，如图 3.3所示，描述了系统的部署架构。业务模块中的可视化部分BetterCode Frontend作为系统的前端服务，需要和Nginx应用一起部署到前端服务器中，其中Nginx起到反向代理的作用。业务模块中的业务中台、代码质量评估模块、Git仓库控制模块和团队代码质量评估模块作为系统的服务端BetterCode Application部署在应用服务器中，应用服务器中还保存了被纳管的项目仓库。系统前端和服务端之间通过RESTful风格的HTTP接口来进行数据通信。同时系统还需要有承载RabbitMQ队列管理应用的节点，本项目中使用了RabbitMQ的Docker镜像在应用服务器中建立了消息队列容器，系统和消息队列之间通过AMQP协议通信。最后系统还需要数据服务器用于部署承担数据存储任务的MongoDB应用，系统和MongoDB之间的交互

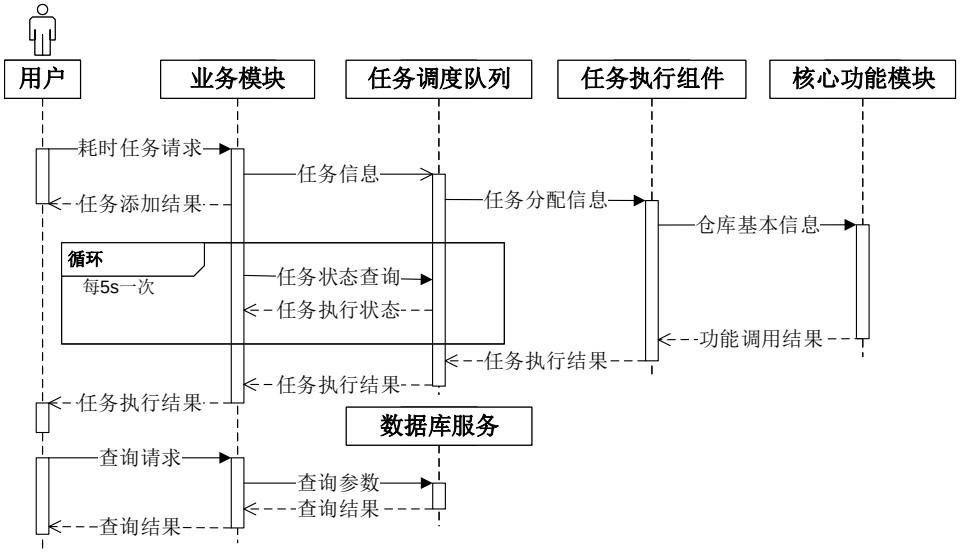

图 3.2: 系统时序图

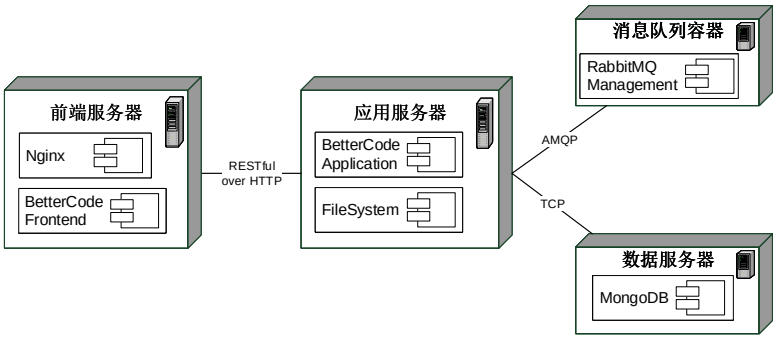

图 3.3: 系统部署图

通过TCP协议传输。图 3.3中的各个部署节点是逻辑节点，具体实现时可以把逻辑节点分派给多台真实服务器或虚拟机，也可以由一台服务器承担所有节点的功能。

其中需要注意的是，虽然在逻辑上可视化模块和业务中台共同组成了业务模块。但是在具体实现中，可视化模块和业务中台需要分别实现、分别部署，二者之间通过RESTFul风格的接口通信。

3.3.2　系统工作流程设计

根据需求分析结果，系统需要具备处理如下工作流的能力：用户纳管目标项目并分析项目质量进而获取项目质量分析结果，用户指定项目提交版本区间从中获取团队活动和活动的质量评估报告。

当系统接受来自用户的仓库纳管请求时，由Git仓库控制模块判断目标仓库是否存在于本地，选择从远端仓库“克隆”或“拉取”等不同的操作，获得项目的最新版本。之后由代码质量控制模块分析本地目标仓库的代码质量特征，抽取相应代码质量元数据，并将元数据交予评估模块。最终由评估模块从元数据中抽取并保存当前项目版本的代码质量指标详细信息，并依据代码质量指标评估标准对质量指标度量数据定性，将定性结果保存并生成项目代码质量评估报告后返回给用户。

在代码质量控制模块提取代码质量特征的同时，由Git仓库控制模块从已纳管的项目仓库抽取项目提交历史并将提交历史元数据传递至评估模块。评估模块利用数据适配器从提交历史元数据中抽取出项目提交记录并保存至数据库中，之后根据项目提交记录建立起了追踪项目中所有文件演化历史的文件生命周期模型。如图 3.4所示，描述了从用户发起纳管仓库请求到完成项目质量评估并建立项目生命周期模型的系统活动状态。

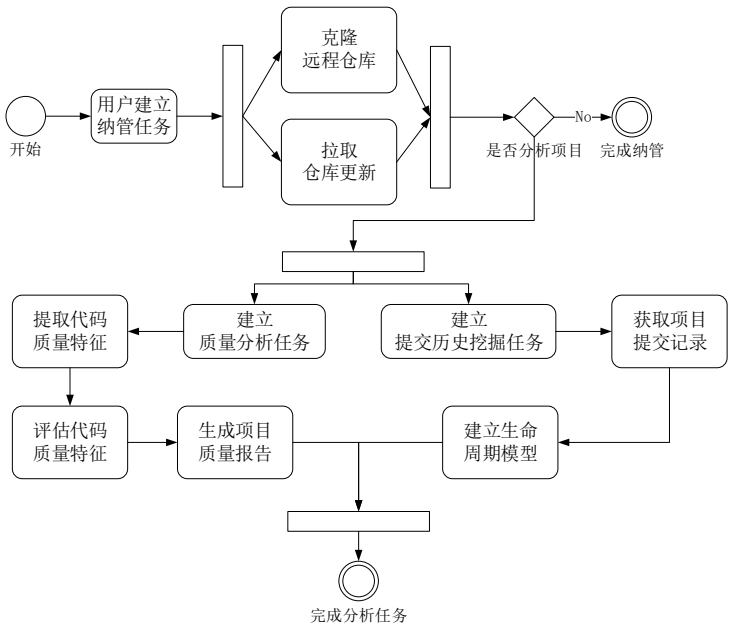

图 3.4: 项目仓库分析活动图

用户可以向系统请求查询被纳管项目的提交历史，用户通过指定开发阶段对应的起始提交节点，来截取当前这一开发阶段内的所有提交记录。评估模块依据这一组提交记录，从项目的文件生命周期模型中提取出在这一开发阶段内所有开发者与其所贡献的文件之间的关系。凭借开发者与文件之间的关系，可以间接定义开发者与开发者之间存在“共同为某一文件贡献内容”的协作关系。

通过判断在这一阶段内开发者之间是否存在协作关系，系统就可以找出这一阶段内所有开发活动的行为主体，即每个独立的开发者或协作小组，并用这些开发主体来代表团队开发活动。利用提交记录和文件生命周期模型，系统获得了开发周期内所有的团队活动。在开发活动中开发者和项目中的文件存在直接的内容贡献关系，通过开发者和文件之间的关系，系统将开发活动与相关文件的代码质量进行一一对应，用代码质量间接表示团队活动的质量。如图 3.5所示，描述了用户查询已纳管项目某一开发阶段开发活动的贡献质量的系统活动。

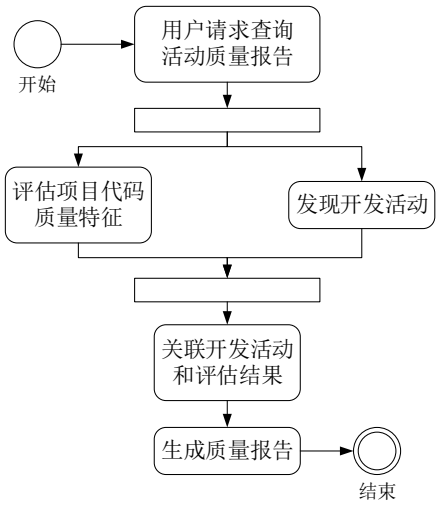

图 3.5: 开发活动的贡献质量评估活动图

3.3.3　系统架构设计

综上所述，可以给出基于Git仓库的团队代码质量控制系统架构的逻辑视图，如图 3.6所示。业务模块处于系统的顶层，在系统和用户之间起沟通作用。业务模块中的可视化模块为用户提供了图形化界面，业务中台连接了可视化模块与系统服务。由于业务模块和系统业务直接相关，因此业务模块负责从请求执行结果中过滤掉评估结果中不需要被用户感知的条目，提取用户真正关心的数据。

与业务中台直接相连的是系统中不需要消耗大量计算时间的查询服务，如团队代码质量查询、文件生命周期查询等服务，此类功能大多由代码质量评估模块提供。系统中需要消耗大量计算时间的服务，如纳管项目仓库、仓库质量分析、仓库演化历史挖掘等服务，则需要业务中台将任务交予消息队列来管理，由消息队列中的任务消费者调用系统功能来完成任务。业务中台向可视化模块

反馈任务进入消息队列的结果，之后由可视化模块周期性请求任务进度，直至任务完成。

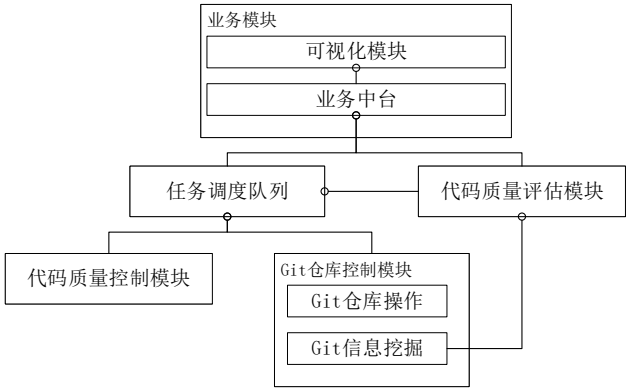

图 3.6: 系统逻辑视图

前文中为了方便介绍模块之间的数据依赖关系，从数据流动的角度分析，将Git仓库控制模块和代码质量控制模块称为质量评估模块的下层模块。而从功能调用角度来讲，在具体的系统实现中，由于仓库纳管、执行分析任务属于耗时请求，此类请求涉及到的相关功能都做了异步处理。质量评估模块只依赖了Git仓库控制模块中历史信息挖掘的部分功能，Git仓库操作功能、项目提交历史分析功能由Git操作相关的消息队列调用。而质量控制模块负责的代码质量特征收集功能同样由消息队列调用。

在系统中传递的数据由三个部分组成，分别是项目仓库中的文件、经过系统分析后的结果数据和业务中台交付给可视化模块的可视化数据。其中业务中台提供的可视化数据是临时数据，不需要存储载体。系统利用文件系统保存项目仓库的文件，利用Spring Boot提供的类似对象关系模型（ORM）的第三方依赖，将分析结果数据保存至MongoDB。

为了指导接下来的开发工作，综合系统的需求分析和架构设计结果，这里给出了系统的开发视图，如图 3.7所示。开发视图关注的是系统中需要实现的模块或组件，在开发视图中，系统被分为前端和服务端两个部分。前端部分包括了实现系统交互功能的四个页面组件，前端的每一个页面组件都在服务端的业务中台有一组业务处理逻辑与之对应。在服务端，除了上述的业务中台外，还包括系统的核心功能组、消息队列组、数据库接口和文件读写接口。核心功能组包括了代码质量控制模块、Git仓库控制模块和团队代码质量评估模块，负责实现系统核心功能。消息队列组包括了系统中各个耗时任务的任务队列，包括接受仓库纳管任务的仓库管理队列、接受代码质量分析任务的质量分析队列和

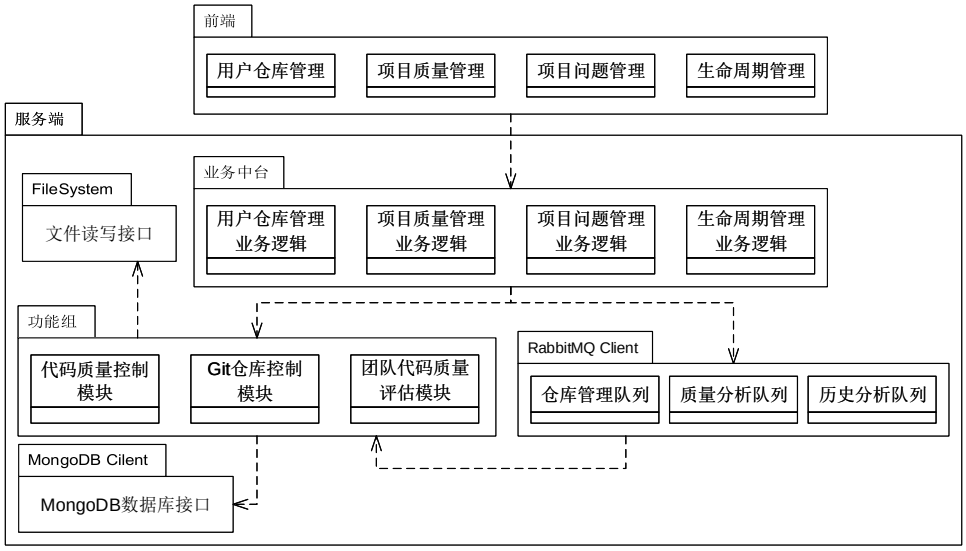

图 3.7: 系统开发视图

接受历史信息挖掘任务的历史分析队列。数据库接口和文件读写接口负责实现系统内持久化数据的增、删、改、查等操作。

3.4　本章小结

本章主要阐述了三个部分的内容：

一、项目概述，这一部分主要阐述了系统及其各个子系统所实现的功能，阐明了项目的意义与目标，并对项目的可行性进行了分析；

二、需求分析，这一部分分析了系统的功能性需求和非功能性需求，对系统的功能性需求进行分类，按照分类后的功能性需求对系统进行模块划分，并详细罗列了每个模块所承担的具体功能性需求。

三、架构设计，这一部分从非功能需求出发，设计了系统各个模块之间的协作模式和工作流程，最终给出了系统的架构设计方案，为后续工作打下良好基础。

第四章　基于Git仓库的团队代码质量控制系统的详细设计与实现

4.1　代码质量控制模块详细设计与实现

代码质量控制模块根据预先约定的质量维度从目标仓库中提取用于描述质量维度的代码特征，并将特征提取结果组织成一组元数据供外部模块使用。在项目代码质量特征提取过程中，主要的流程是：首先对静态代码文本格式化处理成一棵语法树，之后从语法树中提取描述代码质量维度的特征数据。

在系统中，代码质量控制模块根据约定的代码质量维度，需要从Java源文件中分别提取描述代码规模、测试复杂度、变量控制和编程规范等维度的代码特征，通过集成开源代码质量检测工具PMD的CPD功能来收集重复代码质量维度的代码特征。

4.1.1　源代码格式化

抽象语法树基于编程语法，将代码文件转化为可直接处理的树形数据结构，降低了源代码静态分析的难度，对获取的语法树进行深度优先遍历即可将一棵语法树重新转化为可执行的源代码。

代码质量控制模块需要依据代码质量维度描述的代码块粒度，确定语法树中的目标节点，并收集约定的代码特征。系统关注的质量维度中代码规模、测试复杂度和变量控制这三个维度都将度量粒度细化到函数级别，而系统当前版本的编程规范维度在类的粒度上进行代码质量分析。所以，代码质量控制模块需要从语法树中获得所有函数声明节点和类声明节点。

代码质量控制模块使用了开源的代码分析工具JavaParser把静态Java文件源代码转化为语法树，因此关于文件和语法树之间转换的细节在本文不做讨论。质量控制模块基于深度优先遍历算法，从转化后的语法树中获得特定粒度的子树以供后续分析。

算法 1描述了利用深度优先遍历的思想从代码语法树中提取函数节点和类节点的过程。第一至四行判断节点的类型，当节点属于函数节点或类节点时将其添加到结果列表中，第五至七行递归遍历当前节点的子节点。

Algorithm 1: 语法树深度优先遍历DFS

Input: T 语法树

Output: *MethodList* 函数结点列表 *ClassList*类结点列表

```
if T.type is ClassDeclaration then
    ClassList.add(T);
if T.type is MethodDeclaration then
    MethodList.add(T);
subTrees ← T.getChildren();
foreach child in subTrees do
    DFS(child);
```

4.1.2 代码质量度量

完成特定粒度的语法树节点收集工作之后，就可以从函数节点列表和类节点列表中提取每个节点的相关代码质量特征来描述对应代码质量维度。

图 4.1是代码质量控制模块下所有质量维度元数据的类图。所有质量维度均继承自基础抽象类Metric，其中按照是否能以文件组织元数据信息，将各个质量维度元数据分为文件级别和项目级别，除了重复代码维度属于项目级别外，其余维度均属于文件级别。

代码规模反映了一个模块的大小，可以由模块对应代码块所占据的代码行数来表示。一个模块所占据的行数由模块中所有可执行语句的代码行数和所有注释行数构成。为了描述“代码规模”质量维度的代码特征，代码质量控制模块从函数节点中提取了函数内所有的可执行语句和注释的范围以及函数是否属于抽象函数等信息，并将信息组织成元数据交予上层评估模块处理。

语句位置信息$Location_{Stmt}$由起始行Row_S和终止行Row_E来表示，Row_S记录了控制节点所包含的代码块在源代码文件中出现的首行首字符的行列坐标，Row_E表示代码块在源文件中结束的尾行最后一个字符的行列坐标。

评估模块通过统计函数所有可执行语句和注释的位置跨度，计算函数模块的代码规模。通过元数据中的抽象函数标识符与可执行语句数量来判断函数是否是一个没有任何动作的空函数。

函数模块中的逻辑控制节点直接影响了函数测试的复杂度，通过记录函数内各个控制节点的位置信息即可描述函数的测试复杂度。代码质量控制模块记录了语法树中所有函数节点的逻辑控制节点的位置信息，代码质量评估模块通

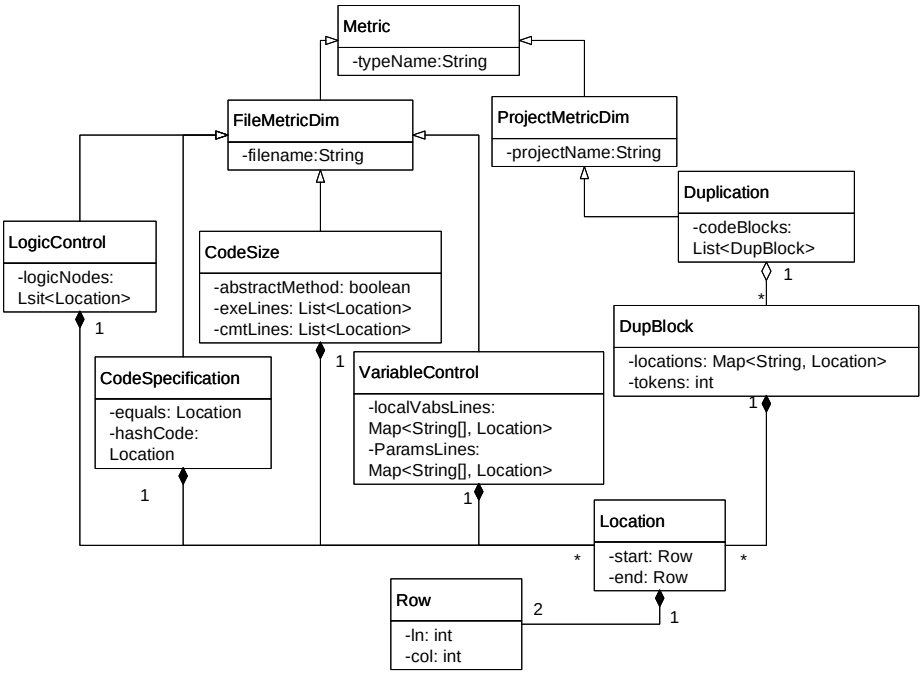

图 4.1: 代码质量维度类图

过控制节点的位置信息来计算函数测试复杂度的质量指标“圈复杂度”和“控制节点最大嵌套层数”。

函数的圈复杂度代表了在函数模块中所有线性无关路径的数量，系统按照如下公式计算函数的模块圈复杂度：

$$CyclomaticComplexity(Method) = N_{control} + 1\,(N_{control} \geq 0) \tag{4.1}$$

其中$Method$表示目标函数的控制节点集合，$N_{control}$表示函数中所有控制节点的数量，函数的圈复杂度即是所有控制节点数量加一。计算公式的现实意义是，当函数中不存在任何控制节点时，函数中只存在一条执行路径，此时函数的圈复杂度为一，当函数中每增加一个控制节点时，函数中都会增加一条线性无关的执行路径，此时函数的圈复杂度为函数中所有控制节点的数量加一。

函数中控制节点的最大嵌套深度更易理解，只需要从节点的位置信息中统计出控制节点嵌套的最大层数即可：

$$MaxNested(Method) = Max(Nested_{control})\,(Nested_{control} \geq 0) \tag{4.2}$$

基于Java语言的语法特点，存在嵌套关系的控制节点在代码块中也必然存在位置嵌套。通过计算函数控制节点的位置信息中包含的嵌套关系即可算出函数控制节点的最大嵌套深度，计算过程如算法 2所示，通过不断统计节点的嵌套深度，来获得函数控制节点中存在的最大嵌套深度。

Algorithm 2: 统计最大嵌套深度

Input: *nodes* 函数中控制节点列表

Output: *MaxNested* 控制节点最大嵌套深度

```
1 i ← 1, MaxNested ← 0, nested ← 0, preNode ← nodes[0];
2 while i < nodes.length do
3     node ← nodes[i++];
4     if node.isSubNode(preNode) then
5         nested++;
6     else
7         MaxNested ← Max(nested, MaxNested);
8         nested ← 0;
9     preNode ← node;
```

在“变量控制”维度中讨论的对象包含了函数本身的局部变量和在函数执行过程中函数上下文参与调用活动的变量。在当前版本，系统的代码质量控制模块提取了函数节点的参数列表和局部变量列表，记录了函数中每一个参数和局部变量的位置、类型信息。评估模块基于质量控制模块提供的变量元数据，分别统计了函数的参数数量和局部变量数量。

“编程规范”维度从当前业界所推荐一系列的软件工程实践中，以“稳定、易于观察、具有代表性”为标准，在当前版本挑选了“类中必须同时重写equals函数和hashCode函数”规范作为这一维度的代表。代码质量控制模块从语法树的类节点中提取了每个类的equals函数和hashCode函数的位置信息，作为描述编程规范质量维度的元数据。系统对这一指标的度量标准是：元数据中equals函数和hashCode函数的位置信息应该同时存在或同时不存在，不应一者存在而另一者不存在。

代码质量控制模块内部集成了开源工具PMD实现了对文件集合中重复代码块的检测功能。模块通过调用PMD的CPD功能接口，获取来自PMD的检测结果，并将检测结果处理为模块间传递的元数据格式。“重复代码”维度的每一条元数据包含了文件集合中的一个重复代码块的行数、标识符数和其在相关文件

中出现的首行行号信息。系统可以通过文件路径和重复代码块行号、行数信息快速从被纳管项目仓库中提取出重复代码内容。评估模块依据重复代码元数据，以文件为单位统计了每个文件中出现的重复代码块数量、位置和每个代码块在文件集合中出现的次数。

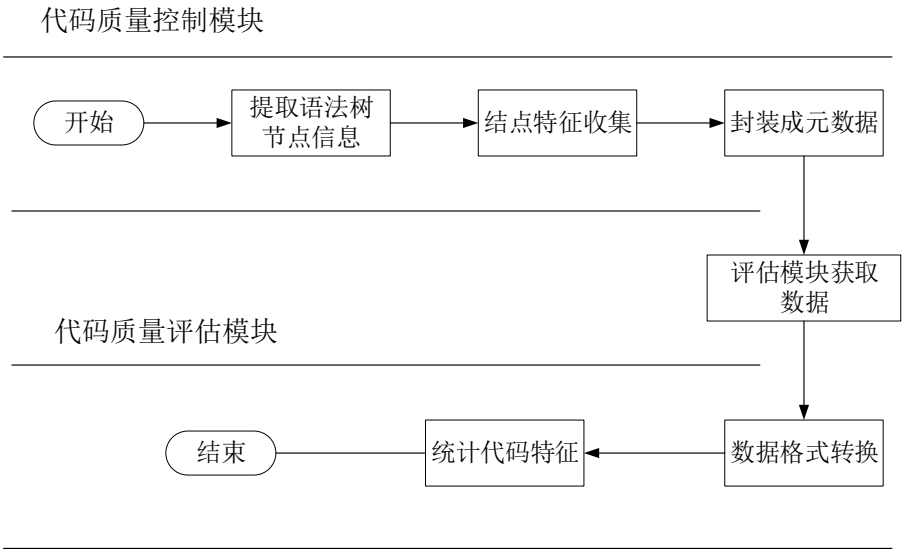

图 4.2: 代码质量特征收集、处理流程图

图 4.2描述了系统中代码质量特征元数据的收集、处理过程。代码质量控制模块从源文件中收集各个质量维度的代码质量特征，将收集到的特征数据封装为元数据后交予数据适配器。经适配器处理后的数据与代码质量控制模块解耦，可直接被质量评估模块使用。

算法 3描述了数据适配器对代码质量特征元数据的默认处理过程。评估模块的每一个统计实体FileMetricStat包含了描述一个文件各个质量维度下所有指标的统计性数据。其中第一第二行是统计实体的初始化过程；第三至十八行根据前文提到的统计逻辑，统计了特征列表中各个维度的质量特征元数据，并将统计结果保存在统计实体的对应属性中。

4.2　Git仓库控制模块详细设计与实现

系统中所有的分析任务都是在本地项目仓库中进行的。管理项目仓库和挖掘项目仓库演化历史的功能由系统的Git仓库控制模块提供。

Algorithm 3: 代码质量元数据处理过程

Input: *MetaMetricList* 质量特征元数据列表

Output: *FileMetricStat* 源文件内所有质量特征的集合

```
1  FileMetricStat ← getNewInstance();
2  FileMetricStat.setBasicInfo();
3  foreach MetaMetric in MetaMetricList do
4      type ← MetaMetric.getType();
5      if type == MethodSize then
6          countAndAddLoc(MetaMetric, FileMetricStat);
7          checkAndAddEmpty(MetaMetric, FileMetricStat);
8          countAndAddLocmt(MetaMetric, FileMetricStat);
9      else if type == VariableControl then
10         countAndAddVabs(MetaMetric, FileMetricStat);
11         countAndAddParams(MetaMetric, FileMetricStat);
12     else if type == LogicControl then
13         countAndAddCyclomaticComplexity(MetaMetric, FileMetricStat);
14         countAndAddMaxNested(MetaMetric, FileMetricStat);
15     else if type == CodeSpecification then
16         checkAndAddOverride(MetaMetric, FileMetricStat);
17     else if type == Duplication then
18         countAndAddDuplication(MetaMetric, FileMetricStat);
```

4.2.1 管理和挖掘项目仓库

系统对项目仓库的管理属于非侵入式的，即Git仓库控制模块不会对项目仓库内容做出修改。在项目管理过程中，模块除了从其他平台拉取、克隆远程仓库等必须的“写操作”外，其他的项目仓库管理操作都属于“读操作”。

仓库控制模块对开源Git管理软件JGit功能进行了组合和封装，实现了系统内需要用到的一系列Git仓库管理操作。如模块实现了从代码托管平台获取远程仓库的功能。

此外，模块实现了项目仓库历史信息挖掘功能，并将其封装为可直接被外部模块调用的功能接口，向调用模块返回对应操作的历史信息元数据，免去了外部模块使用命令行接口调用Git命令需要处理线程并发和文本解析的问题。项

目仓库历史信息挖掘功能包括获取开发者提交的文件修改信息和还原项目文件内容到某一特定版本。

模块通过JGit获取到目标仓库的提交链，比较提交链中任意两个提交对象所指向的目录树对象，即可获得两次提交间的一组差异实体，每一个差异实体代表了目录树中一个具体的文件。当比较的两个提交对象存在前后继承关系时，获得的差异实体集合就是项目在后一次提交版本中被修改的文件集合。

项目历史信息挖掘功能对外返回一组描述项目提交记录的元数据，每条元数据代表了一个提交记录，包括了提交记录的SHA-1标识ID、前驱提交ID列表、提交者信息、作者信息、提交时间、提交描述和一组提交文件的统计性信息。文件的统计性信息记录了该次提交中被修改的文件的修改状态、内容行数变化以及新老文件名变化情况等。

项目历史信息挖掘功能不关注每次提交中文件内容的变化，而是通过项目提交历史的有向无环图，从宏观角度观察项目的演化过程。当需要还原某次提交版本下的文件内容时，使用模块提供的另一挖掘功能即可实现。模块根据特定的提交ID从项目仓库中还原出目标版本下的文件内容，并将文件内容和文件描述信息组织为元数据反馈给调用模块。

算法 4描述了仓库控制模块从项目仓库中提取提交历史的过程。第一行利用了JGit中的Git仓库对象，从仓库对象中获得所有提交对象的列表。第三至十五行描述了项目提交历史的提取过程，其中第六至九行描述了提交历史中非初始提交节点的提取过程，通过比较提交节点和其第一个父节点的差异获取差异实体，并统计差异实体的内容获得两次提交之间修改内容的统计性数据。由于在Git提交历史中一个提交记录可能继承自多个提交，这里模块和Git中的规则保持一致，比较当前提交和从Git数据库中获得的第一个提交之间的差异。第十一至十四行描述了提取初始提交的过程，由于初始提交没有父辈节点，因此无法通过比较两次提交之间的差异来获取差异实体。但是在初始节点中只存在文件被添加进仓库这一事件，因此模块获取了项目在初始提交时所有文件的内容作为差异实体，直接统计内容来获得初始提交的统计性数据。

4.2.2　建立文件生命周期模型

在系统中，代码质量评估任务和项目变更历史信息挖掘任务都是以文件为单位展开的。代码质量控制模块获取了每一个源文件中出现的不同粒度的代码质量特征；Git仓库控制模块挖掘的每一条项目提交记录由一个个代表文件的差异实体组成。因此，系统需要一个从文件角度来分析代码质量和描述演化历史的模型。

Algorithm 4: 仓库提交历史挖掘

Input: *GitRepo* 指向目标仓库的Git实例

Output: *CommitHistory* 项目仓库提交历史

```
1  AllCommits ← GitRepo.RefDatabase.Refs;
2  CommitHistory ← [];
3  foreach commit in AllCommits do
4      Record ← getNewCommitEntity(commit.info);
5      if commit.parents != null then
6          DiffEntities ← ExtractDiffEntity(commit, commit.parents[0]);
7          foreach entity in DiffEntities do
8              EntityStat ← StatisticEntity(entity);
9              Record.entities.add(EntityStat);
10     else
11         CommitContents ← ExtractContent(commit, GitRepo);
12         foreach content in CommitContents do
13             EntityStat ← StatisticContent(content);
14             Record.entities.add(EntityStat);
15     CommitHistory.add(Record);
```

系统提出了文件生命周期模型的概念，文件生命周期模型记录了被纳管项目从仓库初始化版本到系统纳管时的快照版本之间出现过的所有文件实体。生命周期模型中的一个文件实体代表了项目仓库中存在或曾经存在但是已经被删除的一个文件，文件实体的生命周期从被添加到Git仓库开始至被Git仓库删除为止，开发者对文件的每一次修改都会在文件实体的生命周期中产生一个对应类型的事件。

事件描述了文件在特定提交版本下的统计性信息、事件类型、事件所属提交者和事件对应的提交ID，不记录特定版本下的文件内容。一个文件的生命周期事件有四种不同类型，分别是“ADD”类型代表文件被添加到仓库中、“MODIFY”类型代表文件内容被修改、“RENAME”类型代表文件被重命名、“DELETE”类型代表文件从仓库中删除。文件生命周期的起点和终点分别对应ADD类型的事件和DELETE类型的事件。

不同的事件类型除了存在类型标识符的差异外，事件对应文件的新老文件名也存在差异。事件中老文件名标识事件发生前的文件路径，新文件名标识事

件发生后的文件路径。ADD事件的老文件名和DELETE事件的新文件名均不存在，这里系统和Git保持风格一致，使用“dev/null”表示在仓库中不存在的文件路径。MODIFY事件表示开发者只修改了文件内容，因此事件的新老文件名需要保持一致。系统对文件生命周期中重命名事件的判定方法继承自Git和JGit。当开发者重命名一个文件时，往往还会对文件的内容做出修改，如何界定文件是被重命名而不是被删除后在系统中添加新文件，是所有版本控制系统面临的难题 [38]。在Git中，提交对象的目录树中只保存了文件名和对应的二进制对象块，并不存在文件重命名的概念，Git通过计算文件内容的相似度来判断两个版本间的任意两个文件是否存在重命名的可能。JGit认为当不同提交版本间两个文件内容相似度超过60%即可判定新文件是由旧文件经过重命名修改得到的，此时系统认定同一文件在生命周期中发生了RENAME类型的事件，并将文件实体的文件名更新为RENAME事件中的新文件名。

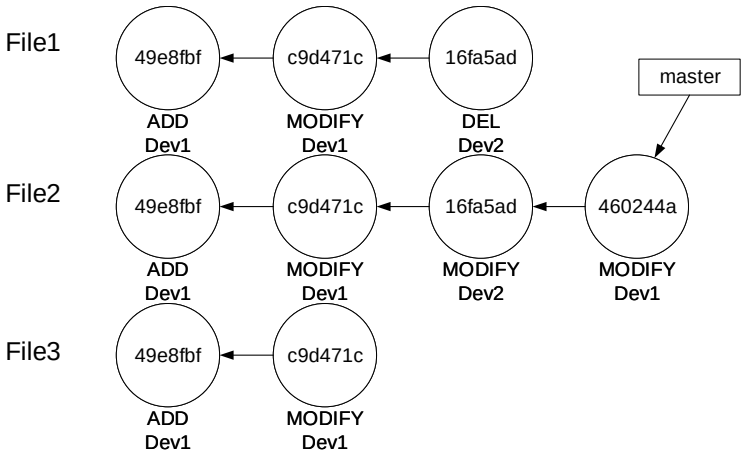

图 4.3: 项目生命周期模型

图 4.3是一个只包含了三个文件的生命周期模型示意图，文件后的每一个圆代表了文件在其生命周期中的一个事件，每一个事件对应一次提交、文件变更类型和变更责任人。这个生命周期模型描述了在当前项目版本中，最新提交460244a记录了开发者Dev1只修改了文件File2，当前版本中处于活跃状态的文件只有File2和File3，文件File1在上一个提交16fa5ad中被开发者Dev2删除。

Git仓库管理模块为系统提供了项目仓库提交历史元数据，团队代码质量评估模块通过数据适配器将提交历史转化并保存，之后由质量评估模块利用提交历史信息构建项目中每个文件的生命周期模型。利用文件生命周期模型，系统可以快速还原出文件的演化过程，确定文件与开发者之间的关系，为后续定义团队开发活动、确定开发活动和文件质量的关系奠定良好的基础。

Algorithm 5: 提取生命周期模型

Input: *CommitHistory* 项目提交历史信息

Output: *LifeCircles* 文件生命周期模型

```
1 LifeCircles ← new HashMap();
2 idx ← CommitHistory.length - 1;
3 while idx ≥ 0 do
4     Commit ← CommitHistory[idx−−];
5     for file in Commit.Entities do
6         lifecircle ← LifeCircles.get(file.getOldName());
7         updateLifeCircle(lifecircle, file);
8         LifeCircles.remove(file.getOldName());
9         LifeCircles.put(file.getNewName(), lifecircle);
```

算法 5描述了文件生命周期模型的构建过程。生命周期模型还原了项目中出现过的每一个文件的演化过程，因此必须从项目初始提交记录开始构建。在第二行代码将指针指向初始提交记录，之后逆序遍历提交列表。第五至九行基于提交记录和记录中的差异实体建立文件的生命周期事件。文件在开发过程中可能发生重命名事件，所以生命周期模型中应该存储最新的文件名，当发生重命名事件时需要更新生命周期模型中的索引。

图4.4描述了评估模块利用处理后的提交信息构建文件生命周期的设计类图。从图中可以看出，文件生命周期中描述每一个事件的数据结构和提交中描述文件变化的数据结构类似。文件生命周期构建的过程就是将每一个文件的修改历史从提交列表中抽出并重新组合的过程。

4.3　团队代码质量评估模块详细设计与实现

团队代码质量评估模块依赖代码质量控制模块和Git仓库控制模块，使用与下层模块适配的数据适配器将收集到的元数据预处理成能够被使用的数据。评估模块利用已处理的质量特征数据和生命周期模型，评估项目开发者和开发活动的贡献质量。

本节主要介绍模块的三个核心功能的设计与实现：基于处理后的质量特征数据给出项目质量评估结果、给出项目快照版本下团队成员的开发质量报告、定义开发周期中存在的开发活动并评估活动质量。

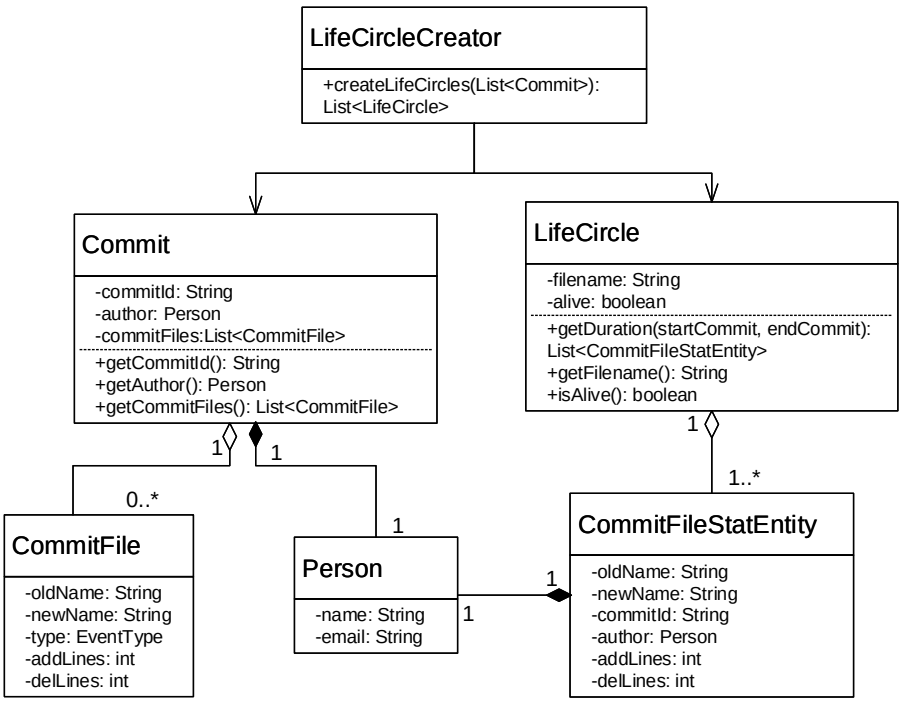

图 4.4: 提交信息-生命周期类图

4.3.1　项目代码质量评估

代码质量评估作为度量开发者和开发活动贡献质量的第一步，是系统的基础功能。评估模块通过预处理操作把来自代码质量控制模块的代码特征元数据量化为描述代码质量指标的统计性数值。系统根据软件工程的实践经验，为每一种质量指标制定了相应的评估标准。

虽然代码特征元数据通过预处理被量化，但是表示指标的数值依旧无法反映代码质量的好坏并直接被开发团队使用。系统需要对这些指标数值定性，屏蔽代码质量指标的细节，使得开发团队可以直接使用定性结果来掌握团队的贡献质量状态，制定开发策略。

在当前版本中，系统划分了三个等级来表示质量指标状态对代码质量产生负面影响的可能性，它们分别是“INFO”、“WARNING”和“ERROR”，表示对代码质量产生负面影响的可能性依次增强。INFO等级表示代码特征的当前状态符合软件工程规范与实践，对代码质量产生负面影响的可能性很小；WARNING等级表示当前的代码特征存在设计不合理的地方，可能会对代码质量产生影响；ERROR等级的质量指标状态说明当前模块的实现方式可能对软件的质量产生比较严重的影响。

根据软件工程理论和实践，质量评估模块为系统中各质量维度的指标制定了定性标准，并依据定性标准将定性结果反馈给用户。

代码规模维度的各个指标数值代表了模块的可执行代码行数和注释行数。现代软件工程实践中，往往认为一块电脑屏幕所能容纳的行数应该是一个功能模块的最大代码行数。在阅读代码时，翻屏会影响代码的可读性和可理解性。根据显示器的高度，可以设定一块屏幕最多可容纳的代码行数。由于可执行代码行数没有统计模块中的空行数量，因此模块在源代码文件中实际的总行数是略大于可执行代码行数的，系统需要设定模块中的空行数占模块总行数的比例。关于可执行代码行数指标的定性标准如下：

$$State_{LOC}=\begin{cases}WARNING & LOC > SLOC\times(1-\alpha)\\ ERROR & LOC \geq 1.5\times SLOC\times(1-\alpha)\\ INFO & \text{ELSE}\end{cases} \tag{4.3}$$

其中LOC表示可执行的代码行数，α表示模块中空行数所占的比例，系统中默认取1/20，$SLOC$表示显示屏所能容纳的最大行数，这里将$SLOC\times(1-\alpha)$称为最大屏显行数，表示一块屏幕中最多能显示的可执行语句行数。

在模块中数量合理的注释可以帮助开发者理解程序代码，过多或过少的注释都表明函数实现不具有良好的可读性和可理解性，函数存在风险的可能性随之上升。系统定义了注释行数指标的定性标准：

$$LOCMT_R=\frac{LOCMT}{LOCMT+LOC} \tag{4.4}$$

$$State_{LOCMT}=\begin{cases}WARNING & LOCMT_R > \alpha_1 \ \ OR \ \ (LOC > 10 \ \ AND \ \ LOCMT_R < \alpha_2)\\ ERROR & LOCMT_R > \alpha_3\\ INFO & ELSE\end{cases} \tag{4.5}$$

其中$LOCMT$表示模块中的注释行数，$LOCMT_R$表示注释行数在模块所有非空行数中所占的比例，α_1、α_2和α_3分别表示注释比例指标不同问题等级的阈值，在系统中分别默认取1/3、1/10和3/4。

代码中存在空函数是开发者不规范编程导致的，空函数的存在会影响代码的可理解性，当空函数被其他函数调用时可能会产生严重后果。系统通过如下标准对空函数指标定性：

$$State_{Empty}=\begin{cases}ERROR & \text{isEmpty is true}\\ INFO & \text{isEmpty is false}\end{cases} \tag{4.6}$$

定性标准表示当模块中存在空函数时，这一指标就被定性为ERROR，反之被定性为INFO，不存在中间状态。

测试复杂度的各个指标数值代表了函数模块中控制节点嵌套和执行路径的复杂程度。系统制定了控制节点最大嵌套层数和圈复杂度的指标定性标准：

$$State_{Nested}=\begin{cases}WARNING & Nested>\beta_1\\ ERROR & Nested\geq\beta_2\\ INFO & \text{ELSE}\end{cases} \tag{4.7}$$

$$State_{CC}=\begin{cases}WARNING & CC>\beta_3\\ ERROR & CC\geq\beta_4\\ INFO & \text{ELSE}\end{cases} \tag{4.8}$$

其中*Nested*代表函数中控制节点的最大嵌套数量，*CC*表示函数的圈复杂度，β_1、β_2代表最大嵌套层数不同等级的阈值，系统中默认取3、5，β_3、β_4表示圈复杂度不同等级的阈值，默认取4、6。

变量控制维度的指标数值代表了函数调用上下文所必须维护的变量数量和函数自身所需要维护的变量数量。函数参数个数和局部变量个数指标的定性标准如下：

$$State_{Params}=\begin{cases}WARNING & Params>\gamma_1\\ ERROR & Params\geq\gamma_2\\ INFO & \text{ELSE}\end{cases} \tag{4.9}$$

$$State_{Vabs}=\begin{cases}WARNING & Vabs>\gamma_3\\ ERROR & Vabs\geq\gamma_4\\ INFO & \text{ELSE}\end{cases} \tag{4.10}$$

其中*Params*代表函数声明的参数的个数，*Vabs*表示函数局部变量的数量，γ_1、γ_2代表函数参数个数不同等级的阈值，系统中默认取3、6，γ_3、γ_4代表函局部变量个数不同等级的阈值，默认取10、15。

系统目前只收录了编程规范维度中Java类函数重写规范，判断Java类实现中是否同时重写了equals和hashCode函数。这一指标的定性标准如下：

$$State_{Override}=\begin{cases}ERROR & Override_{equals()}\ XOR\ Override_{hashCode()}=1\\ INFO & Override_{equals()}\ XOR\ Override_{hashCode()}=0\end{cases} \tag{4.11}$$

当equals和hashCode中只有一个函数被重写时，质量特征被定性为Error，当二者同时重写或都没有被重写时，质量特征被定性为INFO。

重复代码质量维度的指标数值表示文件中的重复代码块在整个文件集合中出现的最大次数。当文件中不存在重复代码块时，指标数值为0。重复代码指标定性标准如下：

$$State_{Dups}=\begin{cases}ERROR & Dups>\theta_1\\ WARNING & Dups\geq\theta_2\\ INFO & \text{ELSE}\end{cases} \tag{4.12}$$

其中*Dups*表示文件中存在的重复代码块在项目中出现的次数，θ_1、θ_2分别表示指标不同等级的阈值，系统中默认为3、5。

图 4.5是代码质量特征评估的设计类图，其中主要负责质量特征评估的组件是MetricStatistic和MetricAssessor，MetricStatistic负责将各个质量维度的代码特征元数据进行统计处理，MetricAssessor负责将各个统计后的指标数值进行评级分类。

对目标项目中各个指标数值定性分级完成了代码质量评估工作的第一步。定性工作使得质量数据离散化，但是数据规模并没有变化，开发者仍然无法直接使用大量指标等级，所以系统需要给出项目的整体质量状况。

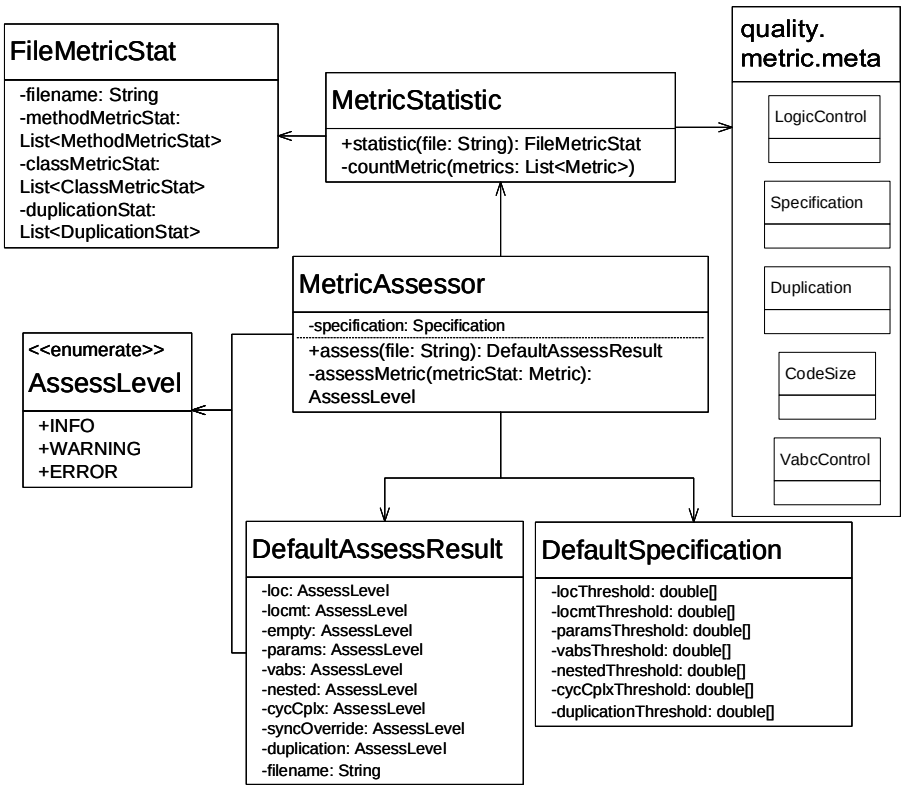

图 4.5: 代码质量特征评估类图

项目整体质量状况以数值的形式来表示。项目整体质量得分来自于项目中所有代码文件综合质量得分的平均值，而文件综合质量得分来自于文件中每一个质量指标的加权平均。计算公式如下所示：

$$Score_{project} = \frac{\sum_{i=1}^{n} Score_{file_i}}{N} \tag{4.13}$$

$$Score_{file} = \frac{\sum_{j=1}^{m} Score_{metric_j} \times W_{metric_j}}{M} \tag{4.14}$$

其中$Score_{project}$表示项目的整体代码质量得分；$Score_{file_i}$表示项目中第i个文件的综合代码质量得分；$Score_{metric_j}$表示文件中第j个质量指标的总分，在系统中，所有指标的总分默认均为1；w_{metric_j}表示文件中第j个质量指标对应评级在该指标总分中的占比，取值在0-1之间；N表示系统中所有的文件数量；M表示文件所有指标总分的总和，在系统中默认等于文件中所有指标的数量。

由于代码质量指标的定性标准和项目质量的评分标准存在较大的主观因素，并且随着软件工程的发展，软件质量的评判标准也在不断演化。所以，质量评估模块并不保存项目质量得分。系统只保留了项目在某次版本下的质量指标数值、指标定性标准和得分计算过程中所需要的各项权重，为了提高效率保存了质量特征定性结果。同时系统在运行过程中，可以根据实际情况对计算规则进行实时的调整，保证系统的可用性。

4.3.2　开发者的贡献质量评估

评估模块对项目中每一条代码指标进行定性分级，并给出了项目的总体代码质量评分和项目中所有文件的综合评分。项目质量评分可以帮助开发团队对项目整体质量有所把握，但是团队管理者无法掌握每一名开发者的开发质量。评估模块需要将开发者和其所贡献的代码质量关联起来，用与开发者相关的文件质量综合评分来衡量开发者的贡献质量。

在当前的系统版本下，文件生命周期的最后一个节点描述了文件最后一次被修改的状态。通过项目的生命周期模型，评估模块可以确定项目当前版本中所有处于“非DELETE”状态文件的最后修改责任人。此时项目中所有文件都有唯一开发者与之对应，模块将每个文件的综合质量评分归属到对应开发者身上，由此可以获得代表开发者的贡献质量评分。

开发者的贡献质量评分公式如下：

$$Score_{Dev_I} = Avg\left(Score_{file_i}\right)(file_i \in Files) \tag{4.15}$$

其中Dev_I表示团队中的任意一个开发者I，$file_i$表示所有被开发者I最后修改的文件集合中的任意一个文件，$Files$表示当前项目中所有处于非DELETE状态的文件集合。开发者的贡献质量评分是所有开发者最后修改的文件的综合质量评分的平均值。

上述开发者的贡献质量计算方法在“将文件质量直接与文件最终修改者关联”这一点上可能会存在争议。在实际开发情况中，开发者很可能需要修改其他开发者所添加的文件，以达到自己的开发目的。此时在文件修改历史中会存在两个或两个以上开发者的修改记录，文件的质量也直接被这些开发者影响。但是，两个开发者对同一文件进行修改时，后者必定需要理解和使用前人留下的代码，即后者不但需要对自己贡献的部分负责，也要为现有代码负责。故评估模块将文件质量和文件最后修改者直接关联。

算法 6描述了评估开发者贡献质量的实现过程。第二行获得了项目当前所有处于活跃状态的文件集合。第三至八行匹配开发者和文件质量的关系。第九行计算“开发者-文件质量”关系集合中每一个开发者的最终成绩。

Algorithm 6: 开发者贡献质量评估

Input: $FileScores$ 项目文件代码质量结果集合
$FileLifeCircles$ 所有文件的生命周期

Output: $DevScores$ 开发者贡献质量报告

```
1 DevsQuality ← new HashMap();
2 Files ← findAllAliveFiles(FileLifeCircles);
3 foreach file in Files do
4     dev ← findLastModifier(file);
5     devQuality ← DevsQuality.get(dev);
6     fileScore ← FileScores.get(file);
7     devQuality.add(fileScore);
8     DevsQuality.put(dev, devQuality);
9 DevScores ← map(DevsQuality, calculateAvg());
```

4.3.3 定义开发活动

前文介绍了衡量项目总体质量的方法，并利用源文件的综合质量评分结果评估了开发者的贡献质量。但是在衡量开发者的贡献质量时，只选取了团队在某一个快照版本下的开发活动作为评估对象。一个项目版本只是项目演化过程的一个剖面，并不能反映项目周期内的所有开发活动。并且，只是分析一个剖面上的开发活动不能评估团队中所有开发者的贡献质量。

图 4.6是某一文件的生命周期示意图，开发者Dev1修改了文件File后，产生提交节点c9d471c。开发者Dev2在之后也对文件File做出修改并提交，并且来自Dev2的提交16fa5ad是文件File的最终提交。如果开发Dev1者在这一阶段只对文件File进行了修改，那么团队管理者就无法获取开发者Dev1的贡献质量评分，也就无法评估开发者Dev1在开发周期中的活动对项目质量产生的影响。

团队开发活动评估是团队代码质量评估模块的一个核心功能，评估模块将评估对象从项目的某一个版本扩展到项目某一个开发周期的演化历史，从评估开发者个人的贡献代码质量拓展到评估开发周期内的所有开发活动的贡献代码质量。

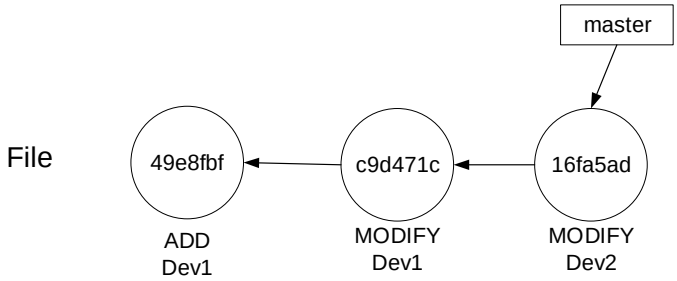

图 4.6: 单个文件生命周期

系统将开发活动定义为：在某一个开发阶段内，活动主体对项目提交的所有贡献的集合。其中活动主体由这一阶段内为项目开发贡献了代码的每一位开发者组成，活动主体可以是一个开发者，也可以是多个开发者组成的临时开发小组。开发活动的客体就是项目中所有在开发阶段内被修改过的文件。通过“开发者是否向文件贡献内容”来确定活动主体和活动客体之间的关联。开发活动具有时间跨度，开发活动发生在一个开发阶段内，开发阶段的时间边界就是所有活动的边界，并且每一个开发活动至少包含一次提交。

由此可以建立开发活动的实体关系图，如图 4.7所示。每个开发者在一个开发周期中都会产生若干个提交，每个提交只会对应唯一开发者；一次提交涉及零个或多个文件生命周期事件，每个文件的生命周期中至少包含了一个提交。

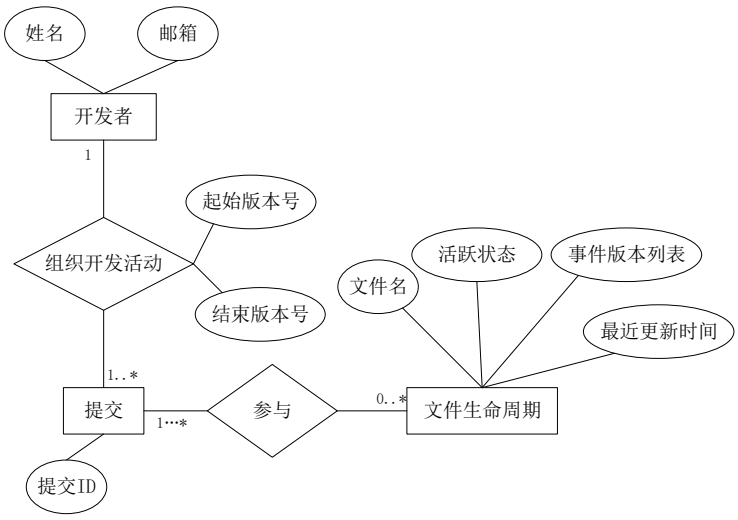

图 4.7: 开发活动实体关系图

开发活动的贡献质量评估建立在项目代码质量评估的基础上，通过评估开发活动产出代码的质量来间接评估开发活动的质量。与开发者的贡献质量评估

不同，评估开发活动前模块需要从项目仓库的提交历史中找出开发周期内所有的开发活动。

在项目的Git仓库中，开发者需要将每一次修改的内容提交到仓库中，除了仓库初始化提交外，所有提交均建立在已有提交的基础上，因此所有的提交记录构成了一个有向无环图，项目的演化历史由这个有向无环图来描述。图 4.8描述了项目在某一个开发阶段的“提交链”。其中提交A和提交F分别是开发阶段的起点和终点，提交V和提交F分别是上文中提到的Dev1和Dev2对文件File的提交c9d471c和16fa5ad，文件File生命周期中的第一个事件节点不属于这一开发阶段。开发者的贡献质量评估功能只考虑了文件生命周期最后一个事件所关联的开发者，评估结果反映了开发者个人的代码质量状况，并不具备时间跨度，不能反映团队开发活动的质量状况。

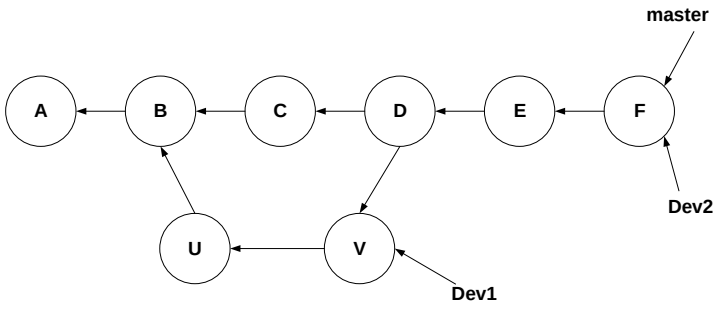

图 4.8: 项目仓库提交历史

系统只讨论项目的某一历史提交版本到当前纳管版本的开发活动质量，并不分析文件演化历史中每一个提交的综合代码质量，因为这是没有意义的。文件演化过程中的大多数提交版本并不会被用于最终生产环境中为用户服务，甚至某些版本只具备一个框架，而没有实现内容，讨论这种文件的代码质量是没有意义的。评估模块根据项目文件演化历史中涉及到的开发者将开发周期中的开发活动分类分为两类，分别是只有一个开发者参与的个人开发活动和多个开发者参与的协作开发活动。

个人开发者活动表示开发活动中的活动主体是一个开发者，对应的客体是这一开发阶段内所有只被该开发者修改过的文件的集合。协作开发活动的活动主体由多个开发者组成，彼此间存在协作关系，这些协作者也被称为一个“临时开发小组”，协作开发活动的客体是这一阶段内临时开发小组贡献的文件的集合。综上所述，评估开发活动贡献质量的第一步就是识别开发活动主体和关联活动主体与客体。

模块可以通过文件生命周期模型获得文件的全部生命周期事件，通过项目提交历史获取某一阶段的提交列表。之后通过检索源文件在提交列表内的生命周期事件详情，即可获得与文件相关的开发活动主体集合。遍历所有项目中源文件，重复上述过程，即可获得某一开发阶段内所有开发活动和活动贡献的源文件集合。

Algorithm 7: 活动发现

Input: $CommitSet$ 开发阶段中所有提交的集合
$FileLifeCircles$ 所有文件的生命周期

Output: *ActivityMap* 开发活动集合

```
1  ActivityMap ← new HashMap();
2  foreach file in FileLifeCircles do
3      events ← filterByCommits(file.lifecircle, CommitSet);
4      if events.length == 0 then
5          continue;
6      committers ← [];
7      foreach event in events do
8          committer ← event.committer;
9          committers.add(committer);
10     Committers2FileSet ← ActivityMap.get(committers);
11     Committers2FileSet.add(file);
```

算法 7是活动发现的具体执行过程。算法的第三行中filterByCommits函数用于过滤掉生命周期中不属于提交记录集合的事件。第四、五行表示模块不考虑在指定阶段内没有发生修改事件的文件。第七至九行表示从过滤结果中抽取与文件相关的开发者，并将这些开发者组成一个开发活动主体。第十、十一行表示更新活动主体和文件的关系表。算法 7的最终结果ActivityMap就是评估模块从提交记录和文件生命周期模型中获得的特定开发阶段的开发活动集合。

4.3.4　团队开发活动的贡献质量评估

确定了开发阶段内所有的开发活动后，就可以利用开发活动所关联文件的代码质量来评估开发活动的贡献质量。评估开发活动只使用了所有开发阶段内被修改文件的代码质量，在指定开发周期内没有被修改的文件的代码质量不会对开发活动的评估结果造成影响。

图4.9描述了评估团队开发活动贡献质量的设计类图，其中DefaultFileScorer和DevActivityScorer是负责计算活动贡献质量的主要组件。DefaultFileScorer负责根据指标评分标准计算已评级文件的质量评分，DevActivityScorer负责从文件生命周期中提取“开发者-文件”关系建立开发活动，并计算开发活动的贡献质量评分。

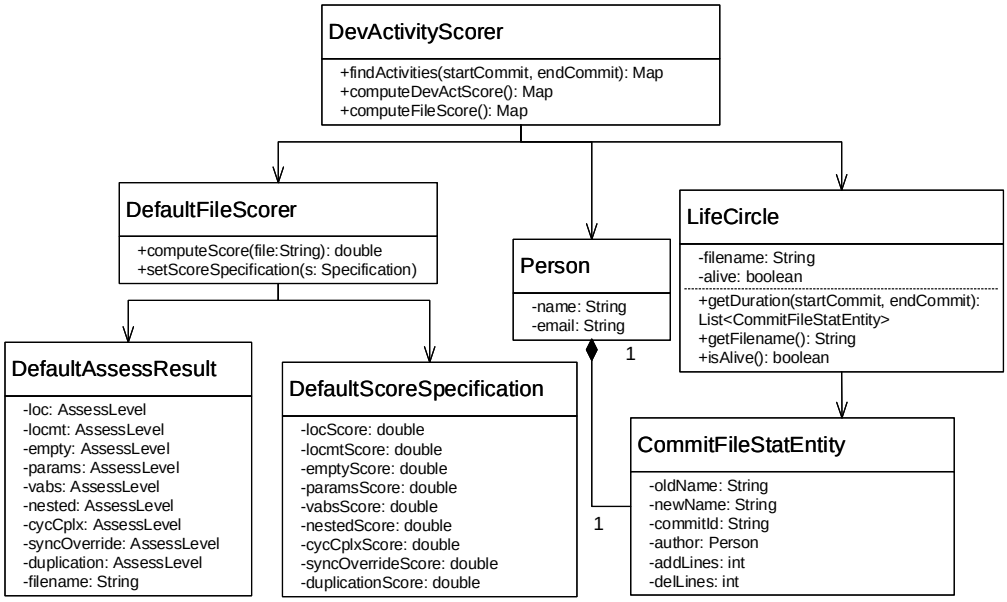

图 4.9: 开发活动的贡献质量评估类图

开发活动的贡献质量计算公式和开发者的贡献质量计算公式相似，公式中$Activity_X$表示在开发周期内的任意开发活动X，$file_x$是活动X在开发周期中所修改的文件集合$Files_{M_X}$中的一个文件，$Score_{file_x}$是文件$file_x$在开发结束时的文件代码质量，所有$Files_{M_X}$组成了开发周期内被修改的文件集合$Files_M$。$Files_M$只统计在开发周期内被修改的文件，是当前版本的项目中所有活跃状态的文件集合$Files$的子集，并且在大多数情况下是其真子集。

$$Score_{Activity_X} = Avg\left(Score_{file_x}\right)(file_x \in File_{M_X}, Files_M \subseteq Files) \tag{4.16}$$

团队开发活动质量评分和开发者的贡献质量评分的计算方式十分相似，两者都使用源文件的代码质量来衡量开发者的贡献质量，但是二者功能并不重复并且以互补关系存在。开发者的贡献质量评估关注项目中每个文件的最终责任人所产生的内容的质量，是对开发者个人能力的衡量。开发活动质量评估关注

开发周期内在开发者之间的独立开发活动或协作开发活动所产生的内容的质量，反映了开发者的活动风格以及这种风格对项目质量产生的影响。

4.4　业务模块详细设计与实现

业务模块由可视化模块和业务中台共同组成，可视化模块提供了系统的功能接口，业务中台负责实现功能接口对应的业务逻辑。用户通过可视化模块的图形化界面向系统发起业务请求，业务中台通过调用一组系统功能服务实现相应的业务逻辑，并将请求执行结果返回给可视化模块。根据第三章的系统需求分析，业务模块被划分为如下四个组件：用户仓库总览组件、项目质量面板组件、项目问题面板组件、项目生命周期面板组件。

如图 4.10业务模块组件视图所示，业务模块中的每个组件由可视化视图和业务逻辑部分组成，视图与逻辑之间通过RESTful接口连接实现功能调用。在系统实现过程中，可视化部分在前端框架Vue的基础上实现，业务逻辑部分基于Spring Boot和RabbitMQ实现。

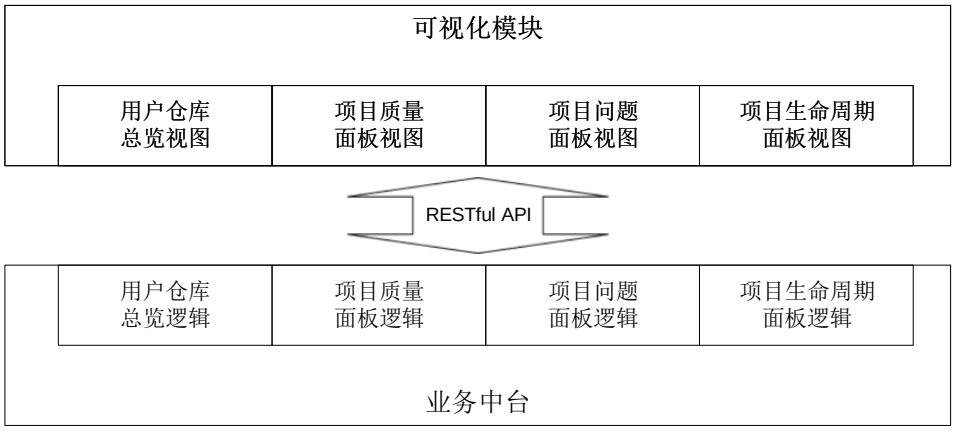

图 4.10: 业务模块组件视图

4.4.1　用户仓库总览组件

用户仓库总览组件为用户提供了未关联仓库管理、已关联仓库管理、新仓库纳管和任务管理等功能接口。系统支持第三方平台登陆，用户可以在仓库总览视图中查询到关联的第三方开源社区中的仓库纳管状态。新仓库纳管功能支持用户将未纳管的项目仓库与自己关联，同时组件提供了任务查询接口，实时更新项目纳管、分析任务的执行进度。

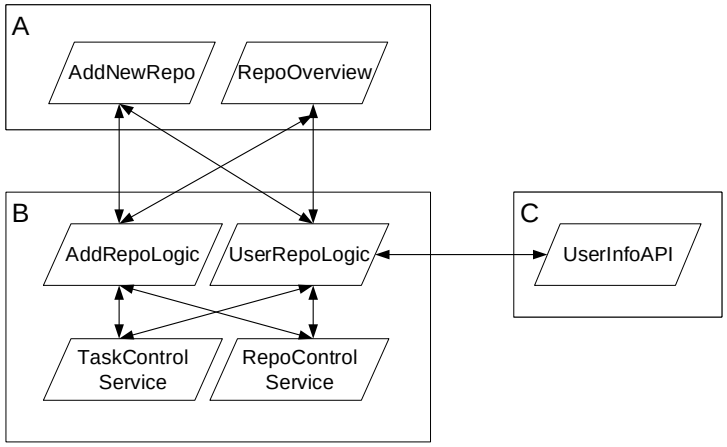

图 4.11: 仓库管理功能调用

图 4.11是仓库总览组件中与仓库管理相关的功能调用逻辑图。其中A、B、C分别代表了不同的载体，A表示载入了可视化模块的用户浏览器，B代表了对外提供服务的系统服务器，C代表了提供用户基本信息的第三方开源社区。当可视化模块请求获得用户相关仓库信息时，向服务端发起RepoOverview请求。业务中台的UserRepoLogic接收请求，并向调用第三方社区的用户信息API获取用户拥有的仓库列表L_1，同时调用仓库管理服务获取用户已关联和正在关联的仓库列表L_2，L_1与L_2的差集，就是用户未关联项目的列表。当用户关联新的项目仓库时，可视化模块向服务端发起AddNewRepo请求。业务中台的AddRepoLogic接收请求，并调用任务管理服务创建项目纳管任务，同时修改项目仓库和用户之间的关联关系。

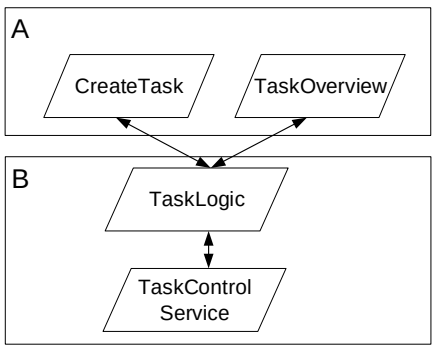

图 4.12: 任务管理功能调用

图 4.12是仓库总览组件中与任务管理相关的功能调用逻辑图。当用户通过可视化模块发起任务创建请求或可视化模块定时发起任务状态查询请求时，均

由业务中台的TaskLogic处理。TaskLogic调用任务管理服务创建任务或获取任务状态，并为可视化模块返回请求结果。

4.4.2　项目质量面板组件

项目质量面板组件为用户提供了查询项目代码质量、查询开发者贡献质量、查询开发活动贡献质量、查询项目开发历史等功能，组件为用户提供了一份项目代码质量报告。

图 4.13-a是项目质量面板组件中查询项目代码质量和开发者的贡献质量的功能调用逻辑图。可视化模块向服务端发起QualityDetails请求来获得关于项目代码质量和开发者的贡献质量信息。业务中台的QualityLogic捕获请求并进行处理，业务中台调用质量评估模块的质量评估器QualityAssessor，再由质量评估器为文件代码质量指标定性结果评分。同时业务中台通过生命周期相关服务获取文件和最终贡献者之间的关系。由QualityLogic将代码质量评估结果和“文件-开发者”关系整合后返回项目代码质量和开发者的贡献质量信息。

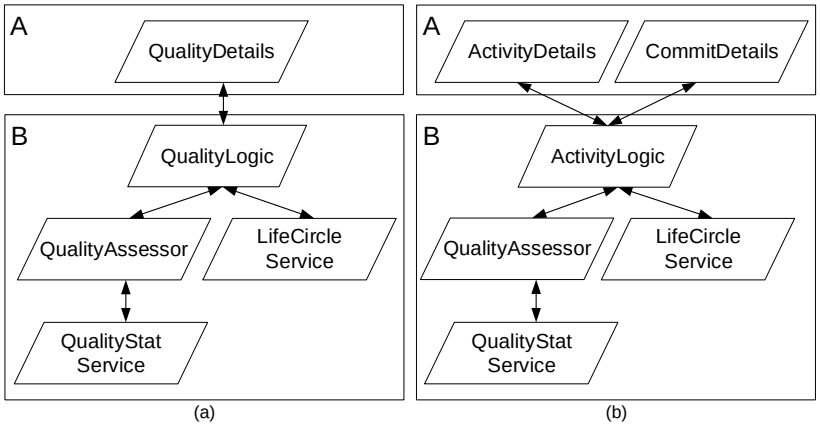

图 4.13: 项目质量面板功能调用

图 4.13-b是项目质量面板组件中关于查询项目开发历史和开发活动贡献质量的功能调用图。可视化模块向服务端请求查询项目提交历史或某一阶段的开发活动时，由ActivityLogic负责处理。当请求查询项目提交历史时，ActivityLogic只调用生命周期服务获取项目的提交历史数据。当请求某一开发阶段开发活动的贡献质量时，ActivityLogic需要调用生命周期服务获取指定阶段内的被修改的文件集合、并找出所有的开发活动主体；同时需要调用质量评估模块获取项目代码质量评估结果。

4.4.3　项目问题面板组件

项目问题面板组件为用户提供代码质量问题查询功能，和责任定位功能，项目问题面板会过滤掉所有INFO级别的质量指标，留下WARNING和ERROR级别的问题指标。

图 4.14是问题面板的功能调用图。问题面板帮助用户追踪开发者贡献质量问题和开发活动贡献质量问题。可视化模块请求开发者的贡献质量报告时，业务中台关联问题指标与文件的最终编辑人；请求开发活动的贡献质量报告时，业务中台关联问题指标和该指标所属文件的所有编辑人集合。

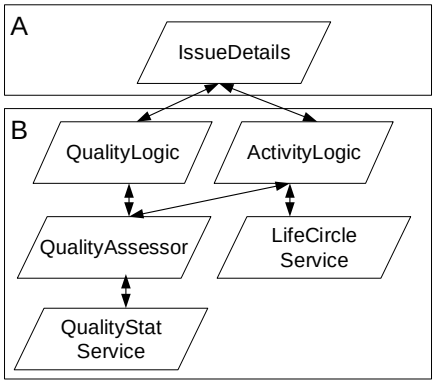

图 4.14: 问题面板功能调用

4.4.4　项目生命周期面板组件

项目生命周期模板提供了可视化的文件生命周期演化过程。可视化模块将用户指定的查询文件名通过FileLife请求交给服务端，业务中台中的LifeCircleLogic通过调用生命周期相关服务将文件的演化历史封装后返回给可视化模块。生命周期面板的功能调用逻辑如图 4.15所示。

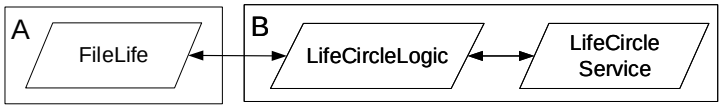

图 4.15: 生命周期面板功能调用

4.4.5　业务模块示例

图 4.16描述了系统的用户仓库面板界面的主要组件，图 4.16-a中项目描述后的对号标记表示项目已经被纳管并且已经完成分析任务，用户关联账号未

纳管的项目条目会使用问号标记。用户通过双击ZooKeeper项目卡片，即可查看ZooKeeper仓库的项目代码质量。图 4.16-b为用户仓库总览界面的搜索面板，用户可以在远程仓库输入栏中输入目标仓库的远程项目地址。输入完成后，通过点击“提取信息”按钮，即可获得系统从项目地址中获取的仓库基本信息，通过点击“获取”按钮，即可创建仓库纳管任务。仓库纳管、分析任务的执行状态会实时更新在图 4.16-c面板中。

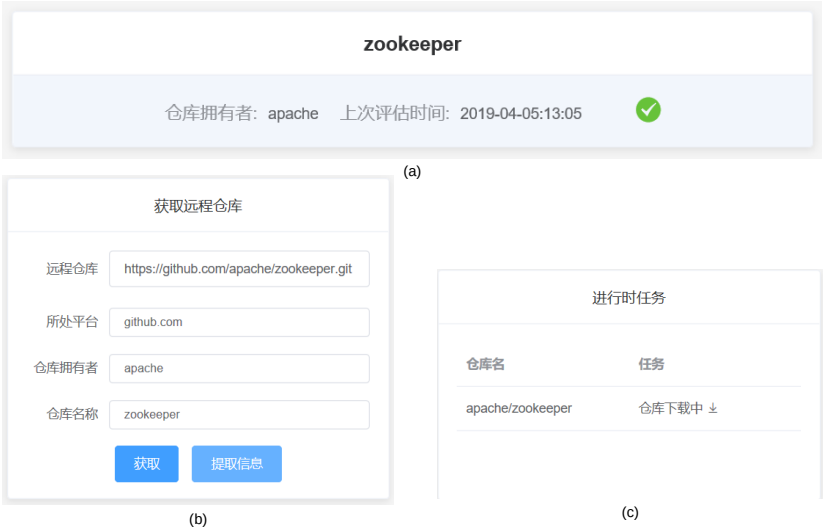

图 4.16: 用户仓库总览界面

图 4.17是项目质量面板界面的每日提交折线图，用于可视化开发活动查询，用户从折线图中选择提交日期即可查看对应日期的提交列表。当用户需要查询某一阶段的开发活动质量时，从列表中选择提交节点，即可通过查询表单查询所选的两个提交之间所存在的所有开发活动和开发活动所对应的贡献质量，在项目中，开发活动提交节点的其中一个端点固定为项目被纳管时的最近一次提交。

从每日提交折线图中选择2019-2-3的第一个提交97e51a4和项目被纳管时2019-4-3的最后一个提交7335efc作为查询区间，即可查询两个月内apache/ZooKeeper项目的开发活动详情。

图 4.18是开发活动质量评估详情页面的可视化组件，开发活动评估结果如图 4.18-a所示，图 4.18-a为开发活动查询结果界面的成绩面板的部分截图，描述了开发阶段内所有开发活动的贡献质量评估结果，开发者贡献质量评估可视化组件与图 4.18-a相似。从图 4.18-a中可以看出在两个月的开发周期内，存在由三个开发者构成的临时协作小组进行的协作活动。通过点击评

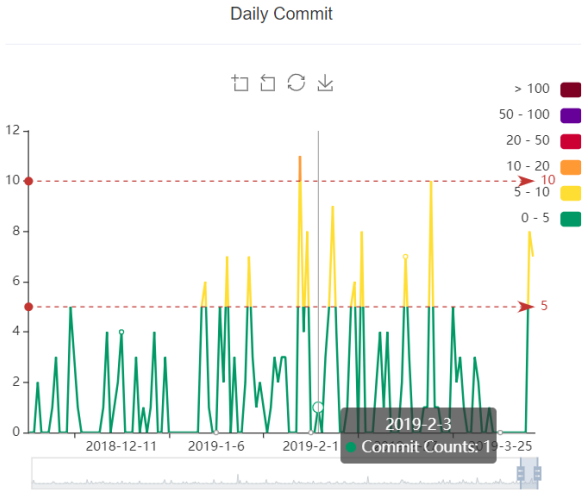

图 4.17: 项目提交历史图

估列表中的特定开发者，即可查看开发者或开发活动相关的质量问题列表，如图 4.18-b所示。本文截取了与开发者Andor Molnar和开发者Norbert Kalmar相关的质量问题列表片段，根据图 4.18-b可知，与两人有关的代码质量问题包括ClientRequestTimeoutTest类的testClientRequestTimeout函数的圈复杂度和变量数量等指标，问题都是WARNING等级。在问题源文件的实现代码中，问题函数testClientRequestTimeout的圈复杂度为5，局部变量个数为12。

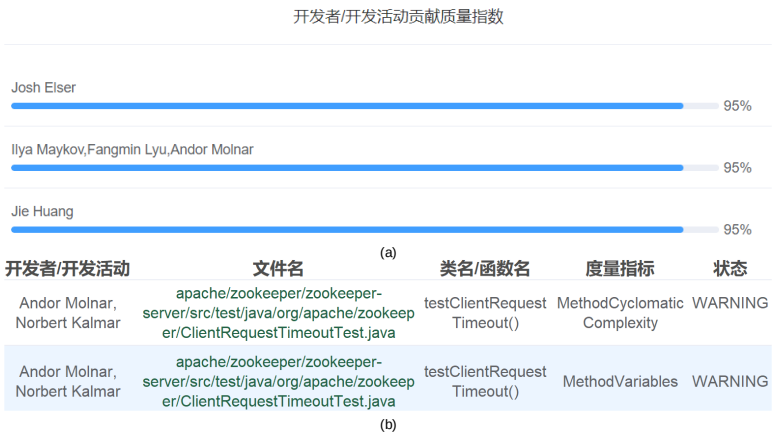

图 4.18: 开发者/开发活动贡献质量评估

4.5　本章小结

本章阐述了系统中各个模块的具体设计与实现细节、代码质量度量体系的详细设计与应用。

在代码质量控制部分讲述了系统从静态源代码中抽取质量特征的详细过程和代码质量特征元数据的预处理方式。

在Git仓库控制部分讲述了系统管理、挖掘项目仓库信息的具体方法和利用仓库提交历史构建项目生命周期的详细过程。

在团队代码质量评估部分首先讲述了代码质量评估标准的细节和项目质量、开发者贡献质量的计分方式，之后阐述了开发活动的发现方法和活动贡献质量的评估算法。

在业务模块部分讲述了系统实现的四个负责处理用户与系统交互的业务功能组件，以及组件和系统中各个功能模块的调用关系。

第五章　系统测试

第四章给出了系统各个模块的详细设计与具体实现，本章将依据第三、第四章的内容对基于Git仓库的团队代码质量控制系统展开系统测试。

5.1　测试环境与测试准备

5.1.1　测试环境

测试工作开始前需要准备测试环境，由于系统在Windows系统下开发，而系统的服务端需要部署到Linux服务器上，所以需要对系统所依赖的软件环境进行严格的控制。系统测试工作在阿里云的轻量应用服务器上展开。

表 5.1是系统的测试环境，为了保证系统能够正常对外提供服务，服务端需要开放的端口详情如下：

（1）Nginx负责监听80端口，对外暴露系统可视化模块；

（2）Java虚拟机负责运行系统的服务端，由于服务端开发使用的Spring Boot自带Tomcat服务器，由服务端监听8080端口对外提供系统功能服务；

（3）MongoDB负责监听27017端口，并在测试过程中打开远程登陆配置，便于远程访问数据库；

（4）使用RabbitMQ-management的Docker镜像，并且映射RabbitMQ服务端口5672和远程管理端口15672到宿主机的5672、15672端口。由于数据库中保存有任务进度详情，因此不需要为容器添加额外数据挂卷。

表 5.1: 测试环境表

参数	参数详情
机型	阿里云轻量应用服务器
CPU	1
内存	2GB
硬盘	40GBSSD云盘
操作系统	CentOS7.5
JVM	Java版本不低于1.8
Nginx	1.14.2
MongoDB	4.0.6
RabbitMQ	docker.io/rabbitmq:management

5.1.2　测试准备

测试是保障软件质量的重要方式，验证系统能否按照预期目的运行，确认系统是否正确实现了需求。由于系统的代码质量控制模块和Git仓库控制模块可以作为独立地Java应用程序被使用，故选取二者的可执行程序作为功能测试的对象。系统测试过程中使用到的测试源代码来自BetterCode系统本身和Apache顶级Java项目ZooKeeper，项目的提交版本为7335efc。

本节以系统的需求分析结果为基础，为系统的各个模块设计了功能测试用例套件，测试系统是否实现了预期的需求设计。测试套件的设计思路从需求出发，测试代码质量控制模块是否能够提取详细的代码质量维度元数据；测试Git仓库控制模块能否执行正确的Git仓库控制操作；测试代码质量评估模块是否能够准确地建立项目生命周期模型、还原开发活动历史、评估代码质量；测试系统的业务模块，验证系统是否具有良好地可用性。测试用例套件表如表5.2所示，每一个测试套件包括了一批与模块特定功能有关的测试用例。

表 5.2: 系统测试用例套件表

相关模块	套件编号	套件描述
代码质量控制模块	TUS1	指定一组文件集合，提取出文件集合中所有代码质量特征元数据。
Git仓库控制模块	TUS2	指定项目的远程仓库，拉取项目远程仓库到服务器本地。
Git仓库控制模块	TUS3	指定本地仓库路径，其中重命名检测相似度指数设置为60%，提取项目演化历史。
代码质量评估模块	TUS4	指定项目的所有代码质量特征数据，评估项目代码质量，并将评估数据按所属文件进行聚类整合。
代码质量评估模块	TUS5	获取指定项目的提交历史数据，构建项目的文件生命周期模型。
代码质量评估模块	TUS6	基于文件生命周期数据和项目代码质量评估结果，获得开发者的贡献质量报告。
代码质量评估模块	TUS7	基于文件生命周期数据、项目代码质量评估结果和给定开发阶段，获得开发活动的贡献质量报告。
业务模块	TUS8	用户进入仓库总览界面，添加项目仓库。
业务模块	TUS9	用户进入项目质量面板界面，查看开发者的贡献质量。
业务模块	TUS10	用户进入项目质量面板界面，查看开发活动的贡献质量。
业务模块	TUS11	用户进入项目问题面板界面，查看开发者的质量问题。
业务模块	TUS12	用户进入项目问题面板界面，查看开发活动的质量问题。
业务模块	TUS13	用户进入项目生命周期面板界面，查看某一文件的生命周期。

5.2　功能测试

本节将以模块为单位，展开系统的功能测试工作，主要流程是：首先在需求分析的基础上为测试套件开发测试用例，之后执行测试用例，最终记录测试

用例通过情况。测试用例执行过程也因每个模块所具有的不同适用场景而分为三种类型。由于代码质量控制模块和Git仓库控制模块可以作为独立工具对外提供完整功能，因此为上述两个模块编写shell自动化测试脚本进行测试。代码质量评估模块只在系统中为业务模块提供支持，因此需要根据测试用例为评估模块编写Junit测试代码。而业务模块则需要根据测试用例在系统界面中执行操作流程。

5.2.1　代码质量控制模块测试

代码质量控制模块功能测试的测试目的是为了保证质量控制模块能够正确地从文件集合中收集其中所有Java源文件的代码质量特征元数据。测试用例的设计思路是通过向模块提供不同种类的文件集合，测试模块应对不同场景的能力。测试用例提供的源代码文件集合包括如下五类：全部都是Java源文件集合、空集合、未包含任何源文件集合、包含部分源文件集合和包含语法错误的源文件集合。从文件类型和文件内容两个角度检测模块的可用性和健壮性。其中包含部分源文件和包含语法错误的源文件是系统在实际运行过程中最经常遇到的场景。

```
1 /bin/bash
2 echo Start testing QualitySniper Module
3 java qualitysniper.jar -files /test/repos/onlyJavas -out /test/TUS1/out -desc TUS1-1
4 java qualitysniper.jar -files /test/repos/empty -out /test/TUS1/out -desc TUS1-2
5 java qualitysniper.jar -files /test/repos/BetterCode -out /test/TUS1/out -desc TUS1-3
6 java qualitysniper.jar -files /test/repos/ZooKeeper -out /test/TUS1/out -desc TUS1-4
7 java qualitysniper.jar -files /test/repos/error.java -out /test/TUS1/out -desc TUS1-5
8 java qualitysniper.jar -files /test/repos/ctn-errors -out /test/TUS1/out -desc TUS1-6
```

图 5.1: 代码质量控制模块测试脚本

图 5.1是代码质量控制模块的测试脚本，其中-files为输入源文件或文件集合的路径，-out为输出报告的根路径，-desc为任务描述信息，会被用作输出日志的文件名。输出日志作为代码质量分析结果，包含了文件集合中总文件数、被分析的源文件数量和异常源文件数量以及异常文件对应的异常信息。

在图5.1中，路径/onlyJavas中只包含十个随机选取自apache/zookeeper的Java源文件。/empty为空文件夹，未包含任何文件；/BetterCode是系统的JavaDoc根路径，包含文件904个，文件夹169个，不包含Java类型的文件。/ZooKeeper获取自apache/zookeeper的远程仓库，其中的Java源文件共有383个。error.java是

主类定义结尾少一个大括号的简单HelloWorld类。/ctn-error则是将error.java放于/ZooKeeper下所得。

5.2.2　Git仓库控制模块测试

本模块功能测试的目的是验证Git仓库控制模块是否具备提供正确的Git仓库管理功能的能力，测试内容包括了远程Git仓库获取和仓库历史信息挖掘。结合需求分析结果，仓库获取功能测试用例的设计思路是通过控制远程仓库的公有属性、私有仓库的用户验证信息和仓库在本地的存在状态来验证模块是否可以正常获取远程项目仓库。历史信息挖掘共测试用例的设计思路是通过控制输入路径是否是Git仓库来检验模块的仓库历史信息挖掘功能。

```
1 /bin/bash
2 echo Start testing GitManager Module
3 java gitmanager.jar -task clone -url https://github.com/apache/zookeeper.git \
  -dir /test/TUS2/repos -desc tus2-1
4 java gitmanager.jar -task clone -url https://github.com/apache/zookeeper.git \
  -dir /test/TUS2/repos -desc tus2-2
5 java gitmanager.jar -task clone -url https://gitlab.com/GitMining/bettercode.git \
  -dir /test/TUS2/repos -u <username> -p <password> -desc tus2-3
6 java gitmanager.jar -task clone -url https://gitlab.com/GitMining/bettercode.git \
  -dir /test/TUS2/repos -desc tus2-4
```

图 5.2: Git仓库获取功能测试脚本

图 5.2是远程Git仓库获取功能的测试用例执行脚本，其中-task是模块待执行的任务种类，有clone和analyze两种；-url是项目仓库远程地址；-dir是仓库拉取任务的本地路径；-desc是任务描述，会被用于输出日志的文件名；-u和-p分别是和私有仓库相关的用户名和密码。第三行验证了模块可以获取公有仓库，第四行验证了模块可以在本地项目已存在的情况下更新项目版本。第五、第六行验证了模块可以利用项目用户身份信息获取私有项目仓库。

图 5.3是仓库历史信息挖掘功能的测试用例执行脚本，仓库提交历史纪录会被输出到-out所对应的输出路径下名为-desc对应值的输出文件中，输出内容包括项目的提交历史和异常告警信息。脚本中第三行-dir对应路径是模块获取的ZooKeeper远程仓库，第四行中/single-commit仓库中只包含一次初始化提交。第五行/no-git路径为普通项目路径，其中不包含.git文件夹，第六行路径/test2020在系统中并不存在。

```
1 /bin/bash
2 echo Start testing GitManager Module
3 java gitmanager.jar -task analyze -dir /test/TUS2/repos/apache/zookeeper \
  -out /test/TUS2/out -desc tus3-1
4 java gitmanager.jar -task analyze -dir /test/TUS2/repos/single-commit \
  -out /test/TUS2/out -desc tus3-2
5 java gitmanager.jar -task analyze -dir /test/TUS2/repos/no-git -out /test/TUS2/out \
  -desc tus3-3
6 java gitmanager.jar -task analyze -dir /test2020 -out /test/TUS2/out -desc tus3-4
```

图 5.3: 仓库历史信息挖掘功能测试脚本

5.2.3 团队代码质量评估模块测试

团队代码质量评估模块的测试用例如表 5.3所示。模块计算的数据由代码质量控制模块和Git仓库控制模块提供，因此，模块所面对的数据相较前两个模块更为干净，格式也更加统一。质量评估模块测试用例的设计目的是验证模块实现的正确性和处理异常数据的可靠性。

根据需求分析结果和测试用例套件，需要对团队代码质量评估模块的四个功能进行测试，分别是代码特征数据评估、生命周期提取、评估开发者贡献质量和评估开发活动贡献质量。每一个功能的测试用例都包括一个正常输入和一个异常输入的测试用例，通过观察测试用例的对应输出，验证模块是否实现了预期设计。

表 5.3: 团队代码质量评估模块测试用例表

用例编号	用例描述	输入/步骤	预期输出
TUS4-1	评估正确特征数据	输入ZooKeeper的代码质量特征数据提取结果。	输出项目下所有源文件的质量评估结果；输出评估的源文件数量为383
TUS4-2	评估含有异常值的特征数据	1. 获得ZooKeeper的代码质量特征数据提取结果。 2. 复制，并修改其中任一条数据，产生条目重复但值不一致的数据	忽略异常条目； 输出项目下所有源文件的质量评估结果； 输出评估的源文件数量为383； 输出存在异常数据的文件名和条目所属质量维度。
TUS5-1	计算正确历史提交记录	输入ZooKeeper的历史提交记录	输出项目中所有文件的生命周期提取结果。

见下页

表 5.3 – 团队代码质量评估模块测试用例表（续表）

用例编号	用例描述	输入/步骤	预期输出
TUS5-2	计算异常历史提交记录	1. 获取ZooKeeper的历史提交记录； 2. 删除数据库中项目的第一条提交记录。	输出项目中所有文件的生命周期提取结果； 输出没有ADD事件的异常文件列表。
TUS6-1	评估开发者的贡献质量	1. 获取TUS4-1结果； 2. 获取TUS5-1结果； 3. 生成开发者贡献质量报告。	输出作为文件最后编辑人的开发者所贡献的代码质量。
TUS6-2	评估开发者的贡献质量	1. 获取TUS4-2结果； 2. 获取TUS5-2结果； 3. 生成开发者贡献质量报告。	输出作为文件最后编辑人的开发者所贡献的代码质量。
TUS7-1	评估开发活动贡献质量	TUS6-1的基础上，输入97e51a4和7335efc作为开发周期起止ID。	输出开发周期内开发活动所贡献的代码质量。
TUS7-2	评估开发活动贡献质量	TUS6-1的基础上，输入97e51a4和97e51a4作为开发周期起止ID。	输出最近提交下开发者所贡献的代码质量。

5.2.4 业务模块测试

业务模块的功能测试用例如表 5.4所示。业务模块主要负责沟通用户和系统，并且系统的各项功能均在前文中进行了测试，因此验证模块的可用性和处理用户的异常输入是业务模块测试用例开发的主要设计目的。

根据需求分析结果和测试用例套件，需要对业务模块的四个组件进行功能测试。功能测试除了对每个组件的基本功能进行测试外，还需要测试每个组件处理异常操作的能力。异常操作主要包括：非法访问、错误内容输入、异常内容输入。除了非法访问操作外，每个组件处理异常操作的方式均不相同。限于篇幅限制，测试用例表中只给出了仓库总览组件和项目质量面板组件的非法访问测试用例。

表 5.4: 业务模块测试用例表

用例编号	用例描述	输入/步骤	预期输出
TUS8-1	拦截未登录用户	直接通过浏览器请求仓库总览界面。	页面跳转至登陆界面。

见下页

表 5.4 – 业务模块测试用例表（续表）

用例编号	用例描述	输入/步骤	预期输出
TUS8-2	添加远程项目仓库	1. 登陆并进入用户的仓库总览界面； 2. 填写真实存在的远程仓库的URL。	输出项目仓库基本信息； 更新当前项目纳管进度； 更新用户关联仓库列表。
TUS8-3	添加远程项目仓库	1. 登陆并进入用户的仓库总览界面； 2. 填写错误的远程仓库地址。	提示项目仓库不存在。
TUS9-1	拦截未登录用户	直接通过浏览器请求质量面板界面。	页面跳转至登陆界面。
TUS9-2	查看项目代码质量	1. 登陆并进入用户仓库总览界面； 2. 选择已关联的仓库。	输出项目中处于各个级别的质量指标数量； 输出开发者的贡献质量； 输出项目的提交历史。
TUS10-1	查看开发活动贡献质量	1. 登陆用户选择已关联的项目仓库； 2. 通过提交历史视图选择与项目最近提交不同的提交记录。	页面跳转至质量问题界面； 输出开发活动的贡献质量； 输出开发活动的贡献中存在的质量问题。
TUS10-2	查看开发活动贡献质量	1. 登陆用户选择已关联的项目仓库； 2. 通过提交历史视图选择与项目最近提交相同的提交记录。	页面跳转至质量问题界面； 输出当前版本开发者的贡献质量； 输出开发者的贡献中存在的质量问题。
TUS11-1	查看开发者贡献质量问题	1. 登陆用户选择已关联的项目仓库； 2. 从开发者贡献质量评估列表中选择开发者。	页面跳转至质量问题界面； 输出当前版本开发者的贡献质量； 输出开发者的贡献中存在的质量问题。
TUS12-1	查看开发活动所贡献质量问题	1. 登陆用户选择已关联的项目仓库； 2. 选择开发周期起始提交点。	页面跳转至质量问题界面； 输出开发活动的贡献质量； 输出开发活动的贡献中存在的质量问题。
TUS13-1	查看文件生命周期	1. 登陆用户选择已关联的项目仓库； 2. 进入生命周期面板； 3. 选择文件树中的任意叶子节点。	输出叶子节点所代表的文件的修改历史。

5.3　本章小结

本章基于系统需求分析结果和系统部署硬件环境，为系统的每一个模块设计配套的测试用例套件，在测试用例套件基础上结合模块需求开发了相应测试用例，并对测试目的和测试用例输出进行了讨论，保证了项目实现的正确性。

第六章　总结与展望

6.1　总结

随着软件规模不断增长，复杂度不断变大，开发团队不但需要实时把握软件产品的质量，还需要掌握开发周期内开发活动的状态来制定开发策略。为了帮助开发团队理解代码质量评估的结果和开发活动的状态，本文提出了基于Git仓库的团队代码质量控制系统。

系统核心功能包括代码质量控制部分、Git仓库控制部分和团队代码质量评估部分。首先，代码质量控制部分按照传统代码质量分析系统的模式，从多个代码质量维度出发收集代码特征，并将特征组织为元数据供评估模块调用。在收集代码特征的过程中，质量控制模块利用了基于抽象语法树的代码静态分析方法，将源代码转化为语法树，通过遍历语法树来获取代码特征。其次，Git仓库控制部分封装了Git仓库操作的基础命令集合，并实现了对项目仓库提交历史的挖掘。最后，团队代码质量评估模块除了为代码质量特征评分外，还根据项目提交历史，还原了项目的开发活动。系统基于开发者和文件之间存在的内容贡献关系，用文件质量评估结果衡量了开发者和开发活动的贡献质量。此外，系统将功能模块与具体业务功能解耦，在业务和功能模块之间设计了专门的业务模块处理系统需要应对的业务场景。

系统具有良好的可拓展性，模块与模块之间都是松耦合的，上层模块不直接依赖下层模块的数据结构，而是通过元数据传递信息。元数据中只包含对内容的描述，并不包含对内容的定性，所有的定性工作均由获得元数据的上层模块来完成。下层模块可以随时添加新的特性，或者增加新的模块，只需要满足统一的接口规范，并通过元数据向上层模块传递信息即可。上层模块只需要为新增元数据增加适配器即可实现对下层模块功能的调用。

系统已经上线，可以获取用户指定的远程项目仓库，能够准确定位出团队在某一阶段的开发活动，可以有效评估项目仓库的代码质量、开发者和团队开发活动的贡献质量。

6.2　展望

下一步，基于Git的团队代码质量控制系统需要在更高可用性、鲁棒性的基础上增加代码质量评估的质量维度，提高评估标准的合理性。

虽然系统按照不同的功能需求进行了模块划分，但是系统内的各个功能模块还处在同一个应用内，存在单点故障的风险。未来可以将现有的功能模块划分出去作为独立的服务，以微服务的形式更加灵活、可靠地组织系统架构。

在当前版本，评估标准中各个指标的阈值、权重参考了软件工程实践经验和业界同类产品的评估标准，具有较大的主观性。在未来，应该根据实际情况不断调整评估标准，让评估标准更加合理。

当前版本的代码质量评估模块只提取了五个代码质量维度的代码特征，在未来系统可以提取更多的代码特征，根据需要集成更多的代码质量分析工具。

现阶段模块间以元数据的方式传递信息，在将系统各个模块转化为微服务之后，为了提高模块之间调用效率，降低网络时延，可以引入如gRPC等更高效的框架来提高效率。

参考文献

[1] G. Schermann, J. Cito, P. Leitner, H. C. Gall, Towards quality gates in continuous delivery and deployment, in: 24th IEEE International Conference on Program Comprehension, ICPC 2016, Austin, TX, USA, May 16-17, 2016, 2016, pp. 1–4.
URL https://doi.org/10.1109/ICPC.2016.7503737

[2] A. F. Ackerman, L. S. Buchwald, F. H. Lewski, Software inspections: An effective verification process, IEEE Software 6 (3) (1989) 31–36.
URL https://doi.org/10.1109/52.28121

[3] R. MacDonald, Software defect prediction from code quality measurements via machine learning, in: Advances in Artificial Intelligence - 31st Canadian Conference on Artificial Intelligence, Canadian AI 2018, Toronto, ON, Canada, May 8-11, 2018, Proceedings, 2018, pp. 331–334.
URL https://doi.org/10.1007/978-3-319-89656-4_35

[4] P. R. de Bassi, G. M. Puppi, P. H. Banali, E. C. Paraiso, Measuring developers' contribution in source code using quality metrics, in: 22nd IEEE International Conference on Computer Supported Cooperative Work in Design, CSCWD 2018, Nanjing, China, May 9-11, 2018, 2018, pp. 39–44.
URL https://doi.org/10.1109/CSCWD.2018.8465320

[5] E. Bouwers, J. Visser, A. van Deursen, Getting what you measure, Commun. ACM 55 (7) (2012) 54–59.
URL https://doi.org/10.1145/2209249.2209266

[6] C. Vassallo, F. Palomba, A. Bacchelli, H. C. Gall, Continuous code quality: are we (really) doing that?, in: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018, 2018, pp. 790–795.
URL https://doi.org/10.1145/3238147.3240729

[7] L. G. Wallace, S. D. Sheetz, The adoption of software measures: A technology acceptance model (TAM) perspective, Information & Management 51 (2) (2014)

249–259.
URL https://doi.org/10.1016/j.im.2013.12.003

[8] C. R. B. de Souza, D. F. Redmiles, The awareness network, to whom should I display my actions? and, whose actions should I monitor?, IEEE Trans. Software Eng. 37 (3) (2011) 325–340.
URL https://doi.org/10.1109/TSE.2011.19

[9] N. G. Leveson, C. S. Turner, Investigation of the therac-25 accidents, IEEE Computer 26 (7) (1993) 18–41.
URL https://doi.org/10.1109/MC.1993.274940

[10] M. Lanza, R. Marinescu, Object-Oriented Metrics in Practice - Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems, Springer, 2006.
URL https://doi.org/10.1007/3-540-39538-5

[11] N. Moha, Y. Guéhéneuc, L. Duchien, A. L. Meur, DECOR: A method for the specification and detection of code and design smells, IEEE Trans. Software Eng. 36 (1) (2010) 20–36.
URL https://doi.org/10.1109/TSE.2009.50

[12] K. Praditwong, M. Harman, X. Yao, Software module clustering as a multi-objective search problem, IEEE Trans. Software Eng. 37 (2) (2011) 264–282.
URL https://doi.org/10.1109/TSE.2010.26

[13] M. W. Mkaouer, M. Kessentini, S. Bechikh, M. Ó. Cinnéide, K. Deb, On the use of many quality attributes for software refactoring: a many-objective search-based software engineering approach, Empirical Software Engineering 21 (6) (2016) 2503–2545.
URL https://doi.org/10.1007/s10664-015-9414-4

[14] T. Gyimóthy, R. Ferenc, I. Siket, Empirical validation of object-oriented metrics on open source software for fault prediction, IEEE Trans. Software Eng. 31 (10) (2005) 897–910.
URL https://doi.org/10.1109/TSE.2005.112

[15] A. Marcus, D. Poshyvanyk, R. Ferenc, Using the conceptual cohesion of classes for fault prediction in object-oriented systems, IEEE Trans. Software Eng. 34 (2)

(2008) 287–300.
URL https://doi.org/10.1109/TSE.2007.70768

[16] T. Zimmermann, N. Nagappan, Predicting defects using network analysis on dependency graphs, in: 30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, 2008, pp. 531–540.
URL https://doi.org/10.1145/1368088.1368161

[17] A. Bacchelli, C. Bird, Expectations, outcomes, and challenges of modern code review, in: 35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013, 2013, pp. 712–721.
URL https://doi.org/10.1109/ICSE.2013.6606617

[18] D. Hovemeyer, W. Pugh, Finding bugs is easy, in: Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, October 24-28, 2004, Vancouver, BC, Canada, 2004, pp. 132–136.
URL https://doi.org/10.1145/1028664.1028717

[19] 高传平, 谈利群, 宫云战, 基于抽象语法树的代码静态自动测试方法研究, 北京化工大学学报(自然科学版) 34 (s1) (2007) 25–29.

[20] N. Rutar, C. B. Almazan, J. S. Foster, A comparison of bug finding tools for java, in: 15th International Symposium on Software Reliability Engineering (ISSRE 2004), 2-5 November 2004, Saint-Malo, Bretagne, France, 2004, pp. 245–256.
URL https://doi.org/10.1109/ISSRE.2004.1

[21] M. West, Object-oriented metrics: Measures of complexity, by brian henderson-sellers, prentice hall, 1996 (book review), Softw. Test., Verif. Reliab. 6 (3/4) (1996) 255–256.
URL https://doi.org/10.1002/(SICI)1099-1689(199609/12)6:3/4255::AID-STVR110\mskip\medmuskip3.0.CO;2-R

[22] J. Pantiuchina, M. Lanza, G. Bavota, Improving code: The (mis) perception of quality metrics, in: 2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018, 2018, pp. 80–91.
URL https://doi.org/10.1109/ICSME.2018.00017

[23] Git, https://git-scm.com/site.

[24] 陈丹, 王星, 何鹏, 曾诚, 开源社区中已有开发者的合作行为分析, 计算机科学43 (S1).

[25] S. Panichella, G. Bavota, M. D. Penta, G. Canfora, G. Antoniol, How developers' collaborations identified from different sources tell us about code changes, in: 30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014, 2014, pp. 251–260.
URL https://doi.org/10.1109/ICSME.2014.47

[26] M. Lavallée, P. N. Robillard, Why good developers write bad code: An observational case study of the impacts of organizational factors on software quality, in: 37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1, 2015, pp. 677–687.
URL https://doi.org/10.1109/ICSE.2015.83

[27] I. Stamelos, L. Angelis, A. Oikonomou, G. L. Bleris, Code quality analysis in open source software development, Inf. Syst. J. 12 (1) (2002) 43–60.
URL https://doi.org/10.1046/j.1365-2575.2002.00117.x

[28] M. S. Rawat, A. Mittal, S. K. Dubey, Survey on impact of software metrics on software quality, International Journal of Advanced Computer Science and Applications 3 (1).

[29] R. P. L. Buse, W. Weimer, Learning a metric for code readability, IEEE Trans. Software Eng. 36 (4) (2010) 546–558.
URL https://doi.org/10.1109/TSE.2009.70

[30] Alibaba Java Coding Guidelines, https://alibaba.github.io/Alibaba-Java-Coding-Guidelines.

[31] 骆斌, 软件工程与计算. 卷2, 软件开发的技术基础.

[32] T. J. McCabe, A complexity measure, IEEE Trans. Software Eng. 2 (4) (1976) 308–320.
URL https://doi.org/10.1109/TSE.1976.233837

[33] A. Sæbjørnsen, J. Willcock, T. Panas, D. J. Quinlan, Z. Su, Detecting code clones in binary executables, in: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA 2009, Chicago, IL, USA, July 19-23, 2009, 2009, pp. 117–128.
URL https://doi.org/10.1145/1572272.1572287

[34] A. B. Mohammed, A. Shahanawaj, Code cloning: The analysis, detection and removal, International Journal of Computer Applications 20 (7) (2011) 34–38.

[35] I. D. Baxter, A. Yahin, L. M. de Moura, M. Sant'Anna, L. Bier, Clone detection using abstract syntax trees, in: 1998 International Conference on Software Maintenance, ICSM 1998, Bethesda, Maryland, USA, November 16-19, 1998, 1998, pp. 368–377.
URL https://doi.org/10.1109/ICSM.1998.738528

[36] PMD, https://pmd.github.io.

[37] I. Neamtiu, J. S. Foster, M. Hicks, Understanding source code evolution using abstract syntax tree matching, ACM SIGSOFT Software Engineering Notes 30 (4) (2005) 1–5.
URL https://doi.org/10.1145/1082983.1083143

[38] J. Loeliger, Version Control with Git - Powerful techniques for centralized and distributed project management, O'Reilly, 2009.
URL http://www.oreilly.de/catalog/9780596520120/index.html

[39] E. F. Codd, A relational model of data for large shared data banks, Commun. ACM 13 (6) (1970) 377–387.
URL http://doi.acm.org/10.1145/362384.362685

[40] N. Leavitt, Will nosql databases live up to their promise?, IEEE Computer 43 (2) (2010) 12–14.
URL https://doi.org/10.1109/MC.2010.58

[41] M. Rostanski, K. Grochla, A. Seman, Evaluation of highly available and fault-tolerant middleware clustered architectures using rabbitmq, in: Proceedings of the 2014 Federated Conference on Computer Science and Information Systems, Warsaw, Poland, September 7-10, 2014., 2014, pp. 879–884.
URL https://doi.org/10.15439/2014F48

[42] P. Dobbelaere, K. S. Esmaili, Kafka versus rabbitmq: A comparative study of two industry reference publish/subscribe implementations: Industry paper, in: Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems, DEBS 2017, Barcelona, Spain, June 19-23, 2017, 2017, pp. 227–238.
URL https://doi.org/10.1145/3093742.3093908

[43] W. Zhang, L. Nie, H. Jiang, Z. Chen, J. Liu, Developer social networks in software engineering: construction, analysis, and applications, SCIENCE CHINA Information Sciences 57 (12) (2014) 1–23.
URL https://doi.org/10.1007/s11432-014-5221-6

简历与科研成果

基本情况 刘锦涛，男，汉族，1994 年 11 月出生，河南省洛阳市人。

教育背景

2017.9～2019.6 南京大学软件学院 硕士

2013.9～2017.6 中国药科大学理学院 本科

读研期间的成果

1. 刘嘉，**刘锦涛**，方文强，邹卫琴，李玉莹“一种基于Star信息和README文档的GitHub相似仓库推荐方法”，专利申请号：201810092877.

致　　谢

两年的研究生生活转瞬即逝，眨眼就到了毕业的时候。首先我想感谢我的研究生导师刘嘉老师，感谢刘老师在这两年来的悉心指导，并且让我有机会作为助教参与到《软件工程与计算III》的课程教学中，给我的毕业项目提供宝贵的实战机会，在实践中不断提高自己的知识储备和技术水平。

在此，我还要感谢陈振宇教授，陈老师给予我的不仅仅是iSE实验室良好的科研、开发氛围，在陈老师严格的要求下，也培养了我严格、细致的习惯。

此外，我还要感谢所有给予过我帮助的朋友们、同学们，感谢大家的陪伴，我才能不断地鼓起勇气在这条路上走下去。

最后，我要感谢我的父母，虽然他们不能帮我写代码，不能帮我写报告、不能帮我做测试，但是他们会在我最疲惫的时候给我力量，他们给予我的帮助是无可比拟的。

版权及论文原创性说明

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：　　年　　月　　日