南京大学

研 究 生 毕 业 论 文

（申请硕士专业学位）

论　文　题　目　一种结合代码依赖和用户反馈的软件可追踪生成方法

作　者　姓　名　njucs201831

学　科、专　业　计算机技术

研　究　方　向　软件可追踪性

指　导　教　师

2018 年 4 月 15 日

学　　号：

论文答辩日期：**2018 年 5 月 26 日**

指 导 教 师：　　（签字）

Combining User Feedback with Closeness Analysis on Code to Improve IR-Based Traceability Recovery

By

njucs201831

Supervised by

A Thesis
Submitted to the Department of Computer Science and Technology
and the Graduate School
of Nanjing University
in Partial Fulfillment of the Requirements
for the Degree of
Master of Engineering

Institute of Computer Software

May 2018

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：一种结合代码依赖和用户反馈的软件可追踪生成方法

计算机技术 专业 2015 级硕士生姓名：njucs201831

指导教师（姓名、职称）：

摘 要

软件开发过程中，诸如代码、需求文档、测试集合等软件制品之间的追踪线索对于软件理解、影响分析、软件维护等活动都具有重要意义。然而，人工的建立软件制品之间的追踪线索需要耗费大量时间和精力。因此，自动化生成软件制品之间的追踪关系成为领域内的研究热点。当前建立需求到代码可追踪性的主流方法是信息检索方法，该方法通过计算需求和代码之间的文本相似度并按照该值自大到小排序形成候选追踪线索列表。然而，需求和代码之间存在的所谓词汇表失配问题使得该方法精度有限，难以支撑日常实践。针对该问题，众多研究工作提出了一系列增强策略。其中，基于代码依赖分析以及基于用户反馈的增强策略是当前的研究热点。然而，引入代码依赖分析的增强策略严重依赖于初始候选追踪线索排序表的精度，当初始列表结果精度不高时，该方法容易对之前列表造成污染。引入用户反馈的增强策略则需要用户遍历大部分列表才能取得明显效果，难以应用到日常实践中。

基于对以上相关工作的分析，为了生成需求到代码的高精度追踪线索列表，我们形成如下重要研究思路：（1）通过代码依赖紧密度分析发掘功能关系紧密的代码元素，并将其放到同一个代码域（code region）中。（2）引入用户反馈来防止直接使用代码依赖可能带来的列表污染问题，根据用户对当前代码元素与需求相关性的验证结果调整域内代码元素和域外代码元素对应候选线索的相似度值。（3）针对域内和域外代码元素对应候选追踪线索设置不同的优化策略，从而改善整个候选列表的排序。

综上所述，本文工作概括如下：

1. 结合代码依赖紧密度分析和用户反馈的可追踪性生成方法。我们提出了一种结合了代码依赖紧密度分析和用户反馈的软件可追踪生成方法。一方面通过设置代码依赖紧密度阈值划分代码域，使得功能紧密的代码元素位于同一个代码域中；另一方面，对于给定需求，将各代码域中有代表性的代码元素交由用户判断与该需求相关性，根据用户反馈结果调整相关代码元素对应候选线索的相似度值。

2. 实验数据组织及方法验证。我们用一个被领域内广泛用于可追踪方法验证的高质量数据集和三个被广泛应用于日常实践的开源系统可追踪数据集验证了我们方法的有效性和实用性。并且，我们通过对开源软件在issue-tracking工具上的软件行为信息进行分析整理，组织了其需求到代码的追踪关系。此外，我们通过运行开源系统自带的用于验证系统功能的测试用例得到了我们方法所需的代码依赖。

3. 基于代码依赖和用户反馈的软件可追踪生成工具的设计与实现。为了将我们的方法应用于日常实践，我们设计并实现软件可追踪生成工具，并集成了我们结合代码依赖紧密度分析和用户反馈的软件可追踪生成方法。

关键词： 需求可追踪性，代码依赖，信息检索，紧密度分析，用户反馈

南京大学研究生毕业论文英文摘要首页用纸

THESIS: Combining User Feedback with Closeness Analysis on Code to Improve IR-Based Traceability Recovery

SPECIALIZATION: Computer Software and Theory

POSTGRADUATE: njucs201831

MENTOR:

Abstract

A variety of software artifacts including code, requirement documents, test sets, etc. will be produced during the process of software development. The traceability between these artifacts especially the requirement-to-code make a great impact on software understanding, impact analysis, and software maintenance. However,establishing traceability between software artifacts is time consuming, tedious, and may involve unforeseen difficulties. Therefore, establish the traceability between software artifacts automatically has become one of the most representative and challenging work in the academic community.Information Retrieval(IR) is now the most widely accepted and applied technique in the research of traceability recovery.

In general, typical IR-based approaches compute the textual similarity between two software artifacts. Unfortunately, different artifacts (e.g., requirements and code) often use different terms to denote the same concept, which is known as the vocabulary mismatch problem. So it is difficult to apply IR approach in practice.To address this issue, researchers have successfully proposed enhancing strategies from different perspectives. The two types of enhancement strategies below are research hotspots in the current field. A growing body of work optimize the candidate list by combining IR techniques with code dependency analysis. However, these approaches are sensitive to the correctness of the candidate links because only correct links can help find additional correct links through code dependencies. Otherwise, the code dependency analysis offers no help or even makes the results deteriorate with the incorrect links. Recent work focused on utilizing user feedback to increase the accuracy of IR-based approaches. However, the users need to verify most of candidate list. This is infeasible in practice to improve IR-based approaches.

Based on the analysis of the above related work, we have formed the following important research ideas to generate a high performance candidate traceability list:(1) Based on closeness measure, we are able to build separate sets of code classes that closely relate each other based on their code dependencies. We suggest that each set(named as code region) implicitly represents at least one aspect of the system functionalities. (2) We try to solve the pollution of candidate list by making the candidate links verified through user feedback prior to adjust the similarity of links with code dependency analysis. (3) We employ different optimization strategies for code elements which are inside and outside region. Eventually, all links are reranked according to the combined information of IR values, user feedback, and our closeness analysis.

In summary, the contribution of this paper is summarized as follows:

1. We proposed an IR-based approach combining user feedback with closeness analysis on code dependencies. On the one hand, we build candidate regions through setting closeness threshold. On the other hand, for a given requirement, we choose the class that has the highest IR value in each region as the representative class. Then we ask user to iteratively verify these representative classes for each region and adjust the similarity of relevant candidate link base on user feedback.

2. We evaluated the above traceability recovery approach on four different case studies. We have validated the effectiveness and practicality of our approach with a high-quality data set which is widely used in the domain for traceability validation and three open source systems that are widely used in everyday practice. And we organized the traceability between requirement and code through analyzing behavioral information of the open source software on the issue-tracking tool. We obtained the code dependencies required by our method by running the test cases which are used to verify system functionality of software system self.

3. In order to apply our method to daily practice, we also developed an assistant tool for traceability recovery between requirement and code and integrating the above approach into it.

Keywords: traceability recovery, code dependencies, information retrieval, closeness analysis, user feedback

目　录

目录 ····· v

第一章　绪言 ····· 1

1.1　研究背景 ····· 1

1.2　研究现状 ····· 1

1.3　研究思路 ····· 3

1.4　本文工作 ····· 4

1.5　本文组织 ····· 5

第二章　相关工作 ····· 7

2.1　需求可追踪性 ····· 7

2.2　基于信息检索技术的追踪方法 ····· 8

2.3　基于信息检索的增强策略 ····· 11

2.3.1　代码依赖 ····· 11

2.3.2　用户反馈 ····· 11

2.4　软件可追踪数据集构造 ····· 14

2.5　本章小结 ····· 15

第三章　结合代码依赖和用户反馈的软件可追踪生成方法 ····· 17

3.1　问题定义及思路来源 ····· 17

3.2　方法概述 ····· 19

3.2.1　计算代码依赖紧密度并生成代码依赖域 ····· 20

3.2.2　基于信息检索方法生成候选追踪线索列表 ····· 23

3.2.3　结合紧密度分析和用户反馈对候选列表重排序 ····· 24

3.3　算法案例分析 ····· 26

3.4　本章小结 ····· 30

第四章 基于代码托管平台的数据组织及方法验证 ····· 31
4.1 数据组织 ····· 31
4.1.1 用于方法验证的可追踪数据集组织 ····· 31
4.1.2 用于方法输入的代码依赖数据组织 ····· 34
4.2 方法验证 ····· 39
4.2.1 实验系统 ····· 39
4.2.2 评价指标 ····· 39
4.2.3 阈值设置 ····· 40
4.2.4 实验目标与研究问题 ····· 41
4.2.5 实验结果与分析 ····· 42
4.3 本章小结 ····· 45

第五章 软件可追踪生成工具的设计与实现 ····· 47
5.1 软件可追踪生成工具应用场景 ····· 47
5.2 软件可追踪生成工具体系结构 ····· 48
5.3 软件可追踪生成工具案例介绍 ····· 50
5.4 本章小结 ····· 53

第六章 总结与展望 ····· 55
6.1 工作总结 ····· 55
6.2 研究展望 ····· 55

参考文献 ····· 57

表　格

1.1　基于信息检索的候选追踪线索排序表 …… 2

2.1　需求到代码的追踪矩阵 …… 8

3.1　与UC15相关的代码元素列表 …… 18
3.2　iTrust用例中基于信息检索的候选线索排序表 …… 20
3.3　iTrust用例中存在的数据依赖 …… 27
3.4　iTrust用例中各数据类型的idtf值 …… 28
3.5　结合用户反馈和代码依赖优化之后的候选追踪线索排序表 …… 30

4.1　数据库表内容描述 …… 34
4.2　实验系统的相关细节 …… 39
4.3　实验系统的评价指标及Wilcoxon秩和检验结果 …… 44
4.4　不同查全率下的精度对比 …… 45

插　　图

2.1　需求双向追踪 …… 7

3.1　需求文本UC15的内容 …… 18
3.2　代码依赖图 …… 19
3.3　基于用户反馈和代码依赖紧密度分析的软件可追踪生成方法流程 …… 21
3.4　代码依赖紧密度图 …… 28
3.5　候选代码域 …… 29

4.1　JIRA上的一条issue信息 …… 32
4.2　针对issue的git commit …… 33
4.3　需求到代码追踪关系整理过程 …… 35
4.4　代码依赖捕获工具结构图 …… 36
4.5　运行测试用例捕获代码依赖的过程 …… 37
4.6　precision-recall曲线 …… 43

5.1　软件可追踪生成工具的应用场景 …… 48
5.2　软件可追踪生成工具的体系结构图 …… 49
5.3　需求可追踪查询界面 …… 51
5.4　辅助用户验证信息界面 …… 51
5.5　需求查询结果显示界面 …… 52

第一章　绪言

1.1　研究背景

软件可追踪性是指在软件开发过程中建立和维护软件制品之间的关联关系，并利用这些关系对软件项目进行一系列分析的能力 [1]。我们把这种关联关系称为追踪线索。在软件开发过程中，会产生各种软件制品，包括需求文档、软件架构、设计文档、用户手册和源代码等 [2]。这些软件制品之间的追踪关系对软件利益相关者有重要价值。不同软件制品之间的追踪关系可以使得软件工程师更好的理解系统，保证软件的质量。同时也可以降低软件维护成本，有利于软件后期的维护和演化 [3–5]。因此，软件可追踪性研究成为了当前软件工程领域的研究热点。

需求是软件系统中唯一记录了人对于完整系统功能的理解与期望的软件制品 [6]，而代码则是当前软件系统运行时行为的唯一真实反映，因此两者之间的关联关系是软件开发与维护人员最关注的。Mäder等人通过对照实验发现，对于软件维护工作而言，在需求到代码可追踪性的支持下，维护的正确率提高了60%，效率提高了21% [7]。事实上，需求到代码的追踪数据描述了软件高层抽象与底层代码实现之间的对应关系。软件迭代过程中，当需求发生变更时，根据需求到代码的追踪数据可以快速找到需要更新的代码位置。此外需求到代码的追踪信息可以应用于软件生命周期以支持变更影响分析、依赖影响分析、系统验证以及安全认证等活动 [8, 9]。然而，建立和维护需求到代码的追踪关系会耗费大量的用户精力并且易出错 [10]。在工业环境中，由于软件系统复杂度高，迭代速度快，这使得人工的建立和维护需求到代码的追踪关系因代价高昂被很多组织放弃 [10, 11]。因此，如何自动化的生成需求到代码元素之间的追踪关系成为了当前研究领域内的首要目标。

1.2　研究现状

当前领域内建立需求到代码追踪关系的主要方法是信息检索方法（Information Retrieve），该方法认为如果有多个词项同时存在于需求和代码文本中，则这两个软件制品大概率是描述了相同的概念或功能，即两者存在追踪关系。该方法计算需求和代码之间的文本相似度并按照该值自大到小的顺序对候选追踪线索进行排序形成候选追踪线索列表 [3, 12–14]。用户自上而下

扫描列表，判断候选追踪线索的相关性。表 1.1 为一个候选追踪线索列表。表中每一行代表一条追踪线索，class 列为代码元素，req 列为需求元素，score 列为需求和代码之间的文本相似度。isTrace列为用户对改候选追踪线索的判断结果，‘X’表示该候选追踪线索对应的需求和代码元素具备相关性。然而，由于需求和代码元素往往采用不同的词汇表，需求会使用一些领域相关的词汇，而代码则会用一些专业术语，词汇缩写等。这就使得描述同一功能的两软件制品之间可能存在单词失配(Vocabulary Mismatch)问题，导致一些相关候选追踪线索排在列表底部。最终导致此类方法的精度有限，难以支撑日常实践。

表 1.1: 基于信息检索的候选追踪线索排序表

class	req	score	isTrace
UpdateCodesListAction	UC15	0.3524	X
UpdateNDCodesListAction	UC15	0.3124	X
DrugCodesDAO	UC15	0.2418	X
editDrugCodes_jsp	UC15	0.2112	X
editNDCInteractions_jsp	UC15	0.1816	
editNDCodes_jsp	UC15	0.1238	X
DCBeanValidator	UC15	0.1045	X
DrugInteractionAction	UC15	0.0953	
AuthDAO	UC15	0.0682	
NDCodesDAO	UC15	0.0487	X
viewResult_jsp	UC15	0.0031	

针对该问题，研究者从不同角度引入了多种增强策略（enhancing strategy）。当前领域内主流的增强策略分为两种：一种是在信息检索基础上引入了代码依赖信息（例如函数调用，继承关系）[15, 16]，该方法的思想一个需求往往是由多个代码元素通过相互调用协作完成。所以，对于与给定需求文本相似度较大的代码元素，增加与其存在代码依赖的其它代码元素与该需求的相似度值能够弥补单词失配带来的消极影响。使得一些原本位于列表底部的候选追踪线索在列表中的位置得到提升。我们之前的工作 [16]对代码元素之间的代码依赖关系进行了量化，使得能够对代码依赖进行更好的利用。然而，此类方法严重依赖信息检索的结果。如果一些相似度较大的候选追踪线索本身就是无关的，此时列表底部与其存在代码依赖的代码元素对应的无关候选追踪线索在列表中的位置可能会得到提升，进而对整个候选追踪列表造成污染。另一种增强策略是在

信息检索基础上引入用户反馈信息（user feedback） [12, 17–19]。根据用户对一些候选追踪线索的判断结果（用户反馈）调整余下候选追踪线索的相似度值。Hayes 等人提出要求用户迭代式判断追踪线索的有效性，然后根据用户反馈更改查询语句（本文中的查询语句为需求文本）的单词权重 [12]，从而改变改需求与余下代码元素组成候选追踪线索的相似度值。Panichella等人提出一种同时使用用户反馈和代码依赖的软件可追踪生成方法 [17]，该方法要求用户遍历整张信息检索方法得到的候选追踪列表，并对每条候选追踪线索的相关性做出判断。只有当前候选追踪线索被判定相关时才提升与其存在代码依赖的代码元素对应候选线索的相似度值。与以上两者迭代式方法不同，Guo等人利用深度学习处理需求到代码的软件可追踪性问题，该方法需要用户先对55%的候选追踪线索有效性做出判断，然后用其中45%的已判断候选追踪线索作为训练集，剩下10%的已判断候选追踪线索用来对通过训练集得到模型进行优化，然后对剩下45%的未判断候选追踪线索进行处理。可以看出此类方法往往需要大量的用户反馈信息，在实际中可操作性不强。

此外，当前用于验证软件可追踪方法有效性的实验系统，大多是由开发人员手工标注和维护需求到代码追踪关系的小型软件系统。例如，领域内常用的数据集iTrust系统最初用来作为学生软件测试课程的项目 [1, 20]。然而这和日常实践中的软件系统差距较大。这使得有些理论上证明有效的方法在日常实践中表现不理想 [21, 22]。当前领域内缺乏公开的用于日常实践的软件可追踪数据集 [9]，因此，使用被广泛用于日常实践的软件系统来验证软件可追踪方法的有效性和实用性是领域内面临的挑战之一。

总的来说，已有工作已经取得重大突破，但是仍有两个工作需要解决：（1）如何在利用少量的用户反馈得到相对高精度的候选追踪显示列表；（2）如何获取日常实践中的软件系统可追踪数据集，并用其验证软件可追踪方法的有效性和实用性。

1.3　研究思路

通过对上述已有工作的分析，我们逐步形成如下研究思路：（1）通过代码依赖紧密度分析识别系统功能紧密的代码元素。一个给定需求通常是由一组交互紧密的代码元素协作实现，而不是随机散布在代码各处 [23]。我们通过引入代码依赖紧密度来量化代码元素之间通过代码依赖的交互程度，并将交互紧密的代码元素放到同一个代码域中。这使得位于同一个代码域中的代码元素通常相互协作共同完成一个需求，并且代码域作为用户反馈的切入点，有效减少了用户反馈的数量。（2）引入用户反馈来防止直接使用代码依赖造成的列表污染

问题。具体来说，对于一个给定需求，取代码域中一个有代表性的代码元素交由用户验证与需求相关性。只有当用户反馈结果为该代码元素和需求相关时，才会调整域内其它代码元素和域外相关代码元素对应候选追踪线索的相似度值。（3）针对域内和域外代码元素设置不同的优化策略。对于域内代码元素对应候选线索相似度值的调整，我们考虑两个因素：域内代码元素的个数和域内其它代码元素与代表代码元素之间路径长度。我们认为域内代码元素越少，则域内代码元素之间的系统功能关系越紧密。我们定义路径长度的值为两代码元素之间所有有效路径上各代码依赖紧密度累乘的最大值。对于域外代码元素，我们根据其到域内代表代码元素之间的距离来优化其对应追踪线索的相似度值，从而改善整个候选列表的排序。

我们需要基于日常实践的高质量可追踪数据集来验证我们方法的有效性和实用性。然而，当前领域内公开的需求到代码可追踪数据集通常是一些小型系统。因此，我们整理了被广泛应用于日常实践的开源软件可追踪数据集。我们的实验系统由一个被领域内广泛用于软件可追踪方法验证的高质量数据集iTrust和三个被广泛用于日常实践的开源软件系统Maven、Pig和Infinispan组成。对于实验系统中由代码托管平台管理的开源软件，其代码托管平台记录了针对其issue-tracking工具描述软件行为变更的代码提交信息。由此，我们组织了软件行为变更（需求）到代码提交（代码元素）的相关性。此外，我们的方法需要高质量的代码依赖信息，我们通过运行软件系统自带的用于验证自身功能的测试用例得到代码依赖子集进而通过合并得到代码依赖集。实验表明，我们的方法只需少量的（3.5%）用户反馈信息在精度上即可显著优于基线方法。

为了将我们的方法应用到日常实践中，我们结合用户使用习惯实现了软件可追踪生成工具，并集成了我们提出的结合代码依赖和用户反馈的软件可追踪生成方法。该工具类似一个搜索引擎，返回与给定需求具有相关性的代码元素。在方法中的用户验证追踪线索相关性阶段，工具提供大量的辅助信息帮助用户高效的做出正确的验证。具体来说，对于被验证候选线索对应的代码元素，工具提供了与之相关的代码依赖结构拓扑图，并用不同的颜色表达图中其它代码元素与该代码元素的代码依赖紧密程度。此外，用户还能很方便的查看已被验证的与当前代码元素具有相关性的其它需求，当前代码元素与需求的相似度值。在需求文本中出现过的单词在代码元素文本中会高亮显示等。以上多种辅助信息有效节省了用户的验证时间并提高了用户验证的正确率。

1.4　本文工作

综上所述，本文完成了以下工作：

1. 结合代码依赖紧密度分析和用户反馈的软件可追踪生成方法。我们提出了一种结合了代码依赖紧密度分析和用户反馈的软件可追踪生成方法CLUSTER(CLoseness-and-USer-feedback-based TracEability Recovery)。该方法有效利用了代码依赖紧密度和用户反馈两部分信息，一方面通过设置代码紧密度阈值划分代码域，使得功能紧密的代码元素位于同一个代码域中；另一方面，将各代码域中有代表性的代码元素交由用户判断与需求相关性，根据用户反馈信息调整相关代码元素对应候选追踪线索的相似度值。

2. 实验数据组织及方法验证。我们用一个被领域内广泛用于可追踪方法验证的高质量数据集和三个被广泛应用于日常实践的开源软件系统数据集验证了我们方法的有效性和实用性。并且，我们通过对开源软件在issue-tracking工具上的行为信息进行分析整理，组织了其需求到代码的追踪关系。我们通过运行开源系统自带的用于验证系统功能的测试用例，动态捕获我们方法所需的代码依赖。

3. 基于代码依赖和用户反馈的软件可追踪生成工具的设计与实现。为了将我们的方法应用到日常实践中，我们设计并实现软件可追踪生成工具，并集成了我们基于代码依赖紧密度分析和用户反馈的软件可追踪生成方法。

1.5　本文组织

针对当前软件可追踪生成领域存在的问题，本文研究需求到代码元素可追踪性生成这一问题。综合考虑代码依赖和用户反馈两部分信息，以尽可能减少用户参与并提高结果精度。从需求到代码元素可追踪生成的场景出发，研究了代码依赖和用户反馈如何有效协作提高候选追踪列表精度。论文总共六章，第一章作为绪论，提出了本文的研究背景及概要结构，其余五章的概要内容总结如下：

第二章主要介绍了与本文研究相关的概念与工作，包括基于信息检索的软件可追踪生成方法，基于信息检索方法的增强策略以及可追踪数据集构造等相关技术。

第三章介绍基于代码依赖和用户反馈的软件可追踪生成方法。介绍如何通过代码依赖紧密度阈值划分代码域，使得功能紧密的代码元素位于同一个代码域中以及如何结合代码域和用户反馈提高软件可追踪生成精度。此外我们用一个案例对我们的方法流程进行了更详细的描述。

第四章介绍了我们对于结合代码依赖和用户反馈的软件可追踪生成方法的实验数据组织和实验结果分析。我们在四个研究案例下设计实验，验证了我们方法的有效性和实用性。我们通过分析开源项目在issue-tracking的软件行为信息组织了其需求到代码的追踪数据，通过运行其自带用于验证自身功能的测试集动态捕获到了我们方法所需要的代码依赖。

第五章介绍了软件可追踪生成工具的设计实现，并结合一个案例阐述了其在实际软件开发过程中的应用。

第六章是对全文工作的总结，并基于我们已有的研究内容提出了未来工作的探索方向。

第二章　相关工作

2.1　需求可追踪性

需求追踪是指跟踪一个需求使用期限的全过程，需求追踪包括编制每个需求同系统元素之间的联系文档，这些元素包括其他类型的需求，体系结构，其他设计部件，源代码模块，测试，帮助文件等。需求跟踪为我们提供了由需求到产品实现整个过程范围的明确查阅的能力。Gotel等人对需求追踪问题进行了一系列的实际调查和分析，并给出了需求追踪的定义：在软件整个生命周期中，对某一特定需求从前后两个方向描述和追踪的能力。如图 2.1所示需求追踪分成两个方向，前向追踪是从书写文档形式的需求追踪到需求来源，后向追踪是从需求文档软件发布过程中的各种制品（例如测试集合、代码、设计文档等）[11]。

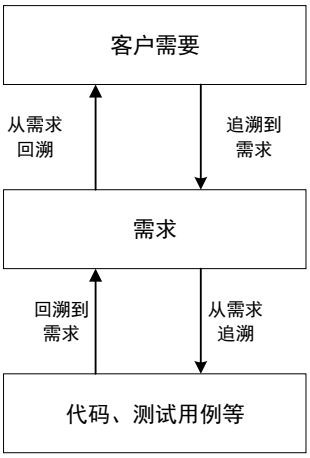

图 2.1: 需求双向追踪

需求到其它任何软件制品的追踪关系都可以用一个二维矩阵来描述。表2.1描述了需求到代码的可追踪关系。 其中行为代码项，列为需求项。每一行代表该代码项与哪些需求有关，每一列表示该需求由哪些代码项来完成。‘X’表示代码项和需求项之间存在实现关系（追踪关系）。在实际的软件生成过程中，代码的粒度通常是类或者函数，需求的粒度可以是用例（ Use Case）或声明（ Requirement Statement）。

表 2.1: 需求到代码的追踪矩阵

	$Requirement_1$	$Requirement_2$	$Requirement_3$	...	$Requirement_n$
$Code_1$	X				
$Code_2$		X			
$Code_3$	X				X
$Code_4$			X		X
...					
$Code_n$		X			

2.2　基于信息检索技术的追踪方法

信息检索技术是软件可追踪生成领域中研究最多、应用最广泛的分析方法。该方法通过计算查询文档和目标文档之间的文本相似度，进而检索出与查询文档相关的目标文档交由用户判断。一方面该方法具有人工参与少、自动化程度高、易于实现等优点；另一方面该方法严重依赖文本质量，普遍存在单词失配问题。本小节主要介绍VSM、LSI 和JS [24]三种常用的信息检索模型。VSM模型将对文本内容的处理简化为向量空间中的向量运算，用向量来表示查询和目标文档，目标文档根据向量之间的余弦距离排序；LSI是一个简单实用的主题模型，基于奇异值分解（SVD）的方法得到文本的主题，克服了VSM模型中存在的近义词和多义词问题。JS属于概率模型，估计文档和查询相关的概率。

VSM（Vector Space Model）在信息检索和搜索引擎中应用非常广泛，也是一种自然语言处理中常用的模型。该模型用向量来表示查询和目标文档，将对文本内容的处理转化为向量空间中的向量运算。查询和目标文档中的每个词项用向量中的一个维度来描述。如果文档中没有出现某个词项，此时该查询向量与该词项对应维度的值就是零。这个值用来刻画词项在文档中的权重。有很多计算词项权重的方法，其中应用比较广泛的计算方法为TF-IDF，它由词项的局部权重和全局权重两部分组成。TF（term frequency）表示当前文档中的某个词项在当前文档中的出现频率。直觉上，某一词项在文档中出现的次数越多权重应该越大，但是这里有个缺陷在于，不同文档长度不同。长文档包含的词项比较多。因此我们用某词项出现的次数除以其所在文档的总词项数来表示该词项的频率。TF计算公式如下：

$$tf_{i,j} = \frac{n_{i,j}}{\sum_k n_{k,j}} \tag{2.1}$$

其中$n_{i,j}$表示词项t_i在文档d_j中出现的次数。分母表示文档d_j中出现的总词项

数。结果$tf_{i,j}$表示词项t_i 在文档d_j中出现的频率。

对于词项的全局权重，如果一个词项普遍存在于各个文档中。极端条件下所有文档中都有该词项，则该词项没办法用来区分不同的文档，即其权重比较低。如果只有很少的文档出现了某词项，那么通过该词项很容易区分开出现过它的文档和未出现它的文档，所以此时应该赋予该词项较高的权重。IDF计算公式如下：

$$idf_i = \log \frac{|D|}{\{|j : t_i \in d_j|\}} \tag{2.2}$$

其中$|D|$ 表示总的文档个数，分母表示出现词项t_i 的文档数目

TF-IDF值由词项的局部权重值和全局权重值综合决定，其计算公式如下：

$$tfidf_{i,j} = tf_{i,j} \times idf_i \tag{2.3}$$

即词项的本地权重值乘以词项的全局权重值。

相似度计算的合理性在于，存在较多共享词项的两个文档，往往是在描述同样的信息。描述不同信息的文档往往使用不同的词项。由于向量空间模型将目标文档与查询文档用高维空间中的一个向量来表示，这两个文档之间的文本之间的相似度可以用这两个空间向量的相似度来表示，给定查询向量q, 目标文档向量d, 向量长度均为n, 那么其余弦相似度定义为：

$$sim(d_j, q) = \frac{\sum_{i=1}^{n} w_{i,j} \times w_{i,q}}{\sqrt{\sum_{i=1}^{n} w_{i,j}^2} \times \sqrt{\sum_{i=1}^{n} w_{i,q}^2}} \tag{2.4}$$

其中，w表示对应文档中由TF-IDF计算而来的词项权重。

LSI（Latent semantic indexing，潜在语义索引） [25, 26] VSM模型中并没有考虑词项之间的关联性。例如，对于分别出现了“automobile”和“car”词项的两个文档，虽然这两个词项是同义词均表示“汽车”。但是在VSM 模型下，两个文本之间的相似度不会因为有大量同义词而变大。LSI模型能够捕获词项之间的相关性，如果两个词项之间具有很强的相关性，那么当一个词项出现时，往往意味着另一个词项也会出现（同义词）；反之，如果查询文档或者目标文档中的某个词项和其它词项的相关性都不大，那么这个词项很可能表示的是另一个意思（一词多义，比如Apple 在不同语境下可能是水果也可能是指手机）。LSI采用奇异值矩阵分解（SVD） [27]对VSM模型下的词项-文档矩阵（Term-by-Document）进行分解。奇异值矩阵分解可以看作是从单词-文档矩阵中发现不相关的索引变量（因子），将原来的数据映射到语义空间内。在词

项-文档矩阵中不相似的两个文档，可能在语义空间内比较相似。SVD背后的形式化比较复杂，详细内容可参考文献 [27]。简单来说，奇异值分解是对矩阵进行分解的一种方法。它能够将一个大矩阵表示为三个小矩阵的乘积，并且与特征分解不同，奇异值分解不要求被分解矩阵为方阵。假设我们的矩阵A 是一个$m \times m$的矩阵，其中的元素全部属于域K，也就是实数域或复数域。如此则存在一个分解使得：

$$A = U\Sigma V^T \tag{2.5}$$

其中U是$m \times m$阶酉矩阵，Σ是半正定$m \times n$ 阶对角矩阵，而V^T即V的共轭转置，是$n \times n$阶酉矩阵。这样的分解就称做M的奇异值分解。Σ对角线上的元素即为矩阵A的奇异值。常见的做法是对奇异值自大到小进行排序，这样Σ可以由矩阵A唯一确定。可以直观的理解为，矩阵U的列向量（左奇异向量）组成一套对矩阵A 的正交“输入”或“分析”的基向量。这些向量是AA^T的特征向量。矩阵V的列向量（右奇异向量）组成一套对矩阵A 的“正交”输出的基向量。这些向量是$M^T M$ 的特征向量。Σ 对角线上的元素是奇异值，可视为是输入与输出间进行的标量的“膨胀控制”。这些是MM^T及$M^T M$ 的奇异值，并与U和V的列向量相对应。对奇异值自大到小排序，取前k个A的奇异值利用$X_k = U_k\Sigma_k V_k^T$构造X的秩－k近似矩阵，重建的矩阵是一个最小二乘法拟合。其中U和V 的列向量正交，即$U^T U = V^T V = I_r$，其中r是原矩阵X的秩。X_k为原矩阵X的k维最佳近似矩阵，由k个最大的奇异三元组构造而成。

LSI通过奇异值矩阵分解将原始词项文档矩阵降维，投影到一个缩小的空间中，以消除词语使用中的“噪声”。在LSI模型中，检索是基于文档的语义内容而不是其词汇内容。为了取得较好的检索结果，选择合适的K值很重要。理想情况下，我们选择一个K值能包含数据中所有的主题模型，同时能过滤掉所有噪音。实际中，选择最优K值的方法还是一个公开问题 [28]。当前K值的选择通常由实验决定。降维之后的查询、文档向量通过将VSM空间的相关向量投影至LSI子空间生成，类似VSM模型，查询语句与目标文档之间的相似度即为对应向量的余弦距离。

JS模型（Jensen-Shannon similarity model）是一种由概率方法和假设检验技术驱动的新型信息检索技术。在概率模型中，查询语句和目标文档都被假定具有潜在的概率分布。通过对查询和目标文档对应概率分布的距离进行排序的方法，达到检索的目的。根据假设检验 [29]的结论，将查询和目标文档看作是词项的概率分布，两者之间的差异通过JS散度来度量。公式如下：

$$JS(q,d) \triangleq H(\frac{\hat{p}_q + \hat{p}_d}{2}) - \frac{H(\hat{p}_q) + H(\hat{p}_d)}{2} \tag{2.6}$$

$$H(p) \triangleq \sum_{w \in W} h(p(w)),\ h(x) \triangleq -x \log x \tag{2.7}$$

其中$H(p)$代表概率分布p的熵，$\hat{p}_q$和$\hat{p}_d$ 分别为查询语句和目标文档的概率分布。我们注意到，根据定义当x 为0时，h(0)恒等于0。因此我们定义查询语句和目标文档之间的相似度为1-JS(q,d)。

2.3　基于信息检索的增强策略

针对信息检索方法的单词失配问题，领域内提出了一系列增强策略。当前主流的两种增强策略为：结合代码依赖信息和结合用户反馈信息。

2.3.1　代码依赖

在基于信息检索方法对需求到代码的可追踪性生成过程中，往往因为词汇失配问题（需求往往使用领域相关词汇，代码往往使用缩写，同义词，专业术语等）使得生成追踪列表的精度（准确率，完全率）有限。本小节主要介绍使用代码依赖信息对信息检索得到的候选追踪线索列表进行优化的一些相关工作。

对于基于信息检索方法生成的候选追踪线索列表，[15]利用代码依赖关系对列表重排序。我们的之前工作 [30] 对代码依赖细分为直接代码依赖（类之间的继承、使用和方法之间的调用）、数据依赖（类之间的数据共享），直接依赖表达的是软件的控制流，数据依赖表达的是软件的数据流。该工作表明对于需求到代码的可追踪性生成，代码元素的直接依赖和数据依赖起到互补的作用，两者相结合要比单独使用其中一种取得更好的效果。后续工作 [16] 对代码依赖之间的紧密度进行了量化。对于代码之间的代码依赖，该工作认为两个类交互越多并且其它类交互的越少，则这两个类之间的代码依赖紧密度越高。该工作通过引入代码紧密度分析，对基于信息检索方法生成的候选追踪列表进行了优化。具体来说，该方法首先利用信息检索方法生成候选追踪线索排序表（[14]位于列表顶部的候选线索大概率是具备相关性的）。接下来，根据其它代码元素与列表顶部对应代码元素之间的代码依赖紧密度值，调整它们对应候选追踪线索的相似度值，进而优化它们在排序表的位置。

2.3.2　用户反馈

如前文 2.2所述，信息检索方法严重依赖软件制品的语料质量，用户需要自上而下扫描信息检索生成的候选追踪线索列表，并验证追踪线索的相关性。

近期大量工作提出根据用户对当前候选线索相关性的验证结果来优化余下的候选线索追踪列表排序 [12, 31–33] 。hayes等人的工作中，假设q是一个查询向量，集合D_q 是用信息检索方法通过查询q 得到的文档集合。假设文档集合D 由两个子集合D_r与D_{irr}组成，其中D_r集合中的文档是用户判断结果为与查询语句相关的文档，D_{irr} 为与查询语句无关的文档，它们所包含的文档个数分别为R和S。很明显D_r集合和D_{irr}集合没有交集，并且由于这两个集合里面都是用户判断过的文档，所以它们的并集也未必是D_q，因为可能有些文档用户还没判断。使用标准Rochio [34] 反馈处理算法，在下一轮迭代中修改查询向量各词项的权重 [12]，公式如下：

$$q_{new} = \alpha \cdot q + (\frac{\beta}{R} \sum_{d_j \in D_r} d_j) - (\frac{\gamma}{S} \sum_{d_k \in D_{irr}} d_k) \tag{2.8}$$

直觉上，通过将相关文档中出现的词项加到查询语句中可以使得查询语句与其它相关文档共享更多的词项，从而提高完全率；削弱查询语句中在不相关文档中出现词项的权重，可以使得查询语句远离不相关文档，进而减小犯错的概率，增加准确率。上述公式中，可以通过调整参数α 、β 、γ 来改变原始查询语句、相关文档和不相关文档中对生成新查询语句的贡献程度。新查询语句生成之后，重新使用信息检索方法并对检索得到的相关文档按相似度排序，重复以上过程，直到用户取得满意的结果。然而，后续工作发现在做需求追踪时用Rochio反馈处理方法只有在前几次迭代中会提高准确率和完全率 [35]。Lucia等人的工作表明标准Rochio反馈处理方法对信息检索方法生成排序表的优化并不明显，尤其是当查询语句和文档语料质量比较好，使得信息检索结果的质量比较高时，Rochio反馈处理方法对信息检索的结果几乎起不到任何优化作用，对于其它软件制品之间做可追踪性（例如用例和UML图之间）使用Rochio 方法不仅不能提升排序表的效果反而可能会让效果并得更差 [31]。Zhang等人对不同查询语句进行了分类，与 [12] 使用固定的参数α 、β 、γ 不同，根据查询语句所属类别采用不同的参数并取得一定的效果 [33]。使用Rocchio算法利用用户反馈（user feedback）信息提高对查询语句的检索效果要基于两个前提条件 [36]：

1. Rocchio算法基于的假设是：查询语句与目标文档相比，词项比较少。
2. 与查询语句相关的文档之间相似度比较高，假如对查询和文档进行分析，查询和相关文档应该能聚到同一个类中（聚类假设）。

Panichella等人认为在需求追踪环境下以上两个前提假设均不成立，有些查询语句（需求的文本文档）词项比目标文档（代码文档）要多 [32]。并且，聚类假

设也不成立，根据相关文献 [37, 38]，需求实体和代码实体往往使用不同的词典，需求实体往往使用问题领域词项，代码实体则使用技术领域词项和一些缩写同义词等。综上 [32]提出一种方法，只有查询语句的词项小于等于相关文档，并且用户判断的文档中与查询语句相关的文档比不相关的文档多时才对用户反馈使用Rocchio算法，该方法取得一定效果。

与上述工作从通过用户对候选线索的反馈结果优化排序不同， [39]基于n-gram模型对查询语句进行分析，对不同的词项采取不同的权重调整策略（增大，减小，不变）。 [4, 40]先让用户判断一部分候选追踪线索，作为方法的训练集。具体来 说， [4]使用一组被判断过的追踪线索（训练集）作为贝叶斯分类器的输入，通过对测试集进行学习来提高基于概率模型的信息检索方法的准确率。[19]提出了一种使用嵌入词项和RNN技术的神经网络结构来生成软件制品之间相关性的方法。与基于信息检索的方法类似，该方法也会分析查询语句和目标文档的文本相似度，将用户已经判断过的追踪线索作为训练集作为方法的输入。相对于传统的基于VSM 和LSI模型的信息检索方法， [19] 方法得到的结果分别提高了41%和32%。但是该方法需要45%的测试集和10%的用户已经判断的追踪线索对通过训练集得到的分类模型进行优化。即用户需要判断所以候选追踪线索的55%。这在实践中会耗费大量的用户精力。

[17] 认为对于使用代码结构信息来调整候选追踪列表的方法 [15]，其最好的效果和信息检索方法得到的候选追踪线索排序表有很大关系。 [15]当某个文档和查询语句相似度比较大时，此时增大与文档存在结构依赖的其他文档与查询语句的相似度。但是在文档文本质量比较差的情况下。如果与查询语句相似度比较大的文档本身和查询语句就没有相关性，此时利用代码依赖就会将错误扩大，对信息检索方法得到的候选追踪线索排序表造成污染。 [17]提出，每次将排序表顶端的追踪线索交由用户验证，只有用户判断该追踪线索相关时才提高与该追踪线索具有代码依赖的其它追踪线索的相似度值。与 [12, 31, 32] 不同， [17]不会改变词项的权重，因此不需要每次重新计算相似度。该方法相似步骤见算法 1

其中δ为相似度奖励系数，用来调节相关追踪线索相似度的变化程度。对于不同的追踪列表，该变量应该取不同的值，考虑到当候选列表的值分布比较集中时，该值稍微大一点就会使得相关候选追踪线索相似度值调整之后的排序会发生很大的变化；当候选列表的值分布比较发散是，该值过小可能使得相关线索虽然小相似度值增大，但是对其再排序表的位置确影响不大。该方法提出自

Algorithm 1: User-Driven Combination of Structural and Textual Information —UD-CSTI

```
while not (stopping criterion) do
    Get the link (s, c_j) on top of List
    The user classifies (s, c_j)
    if s, c_j is correct then
        forall the c_t ∈ C do
            if (c_j, c_t) ∈ E then
                Sim(s, c_t) ← Sim(s, c_t) + δ × Sim(s, c_t)
    Reorder List
    Hide links already classified
```

适应的δ。

$$\delta = median\{v_i, ..., v_n\} \tag{2.9}$$

其中$v_i = (max_i - min_i)/2$，v_i与第i个查询文档对应目标文档的相似度最大值和最小值。

2.4　软件可追踪数据集构造

Cleland-Huang等人提出需求追踪领域面临的一个挑战是，当研究者想验证自己方法有效性时，需要搭建基线方法的实验运行环境，这个过程会耗费大量精力，并且有些实验用到的标准可追踪数据集往往不公开 [41]。Charrada等人认为用于方法验证的可追踪性数据集比较少，研究者为了验证自己方法有效性往往需要自己做数据集，这个过程会消耗用户大量时间。Charrada 等人公布了一个针对一个灌溉系统的数据集，包括了该系统各种软件制品之间的追踪关系。同时通过用同样的方法对该数据集和领域内常用的数据集进行处理，比较它们precision/recall图像的相似性，验证该数据集的有效性 [42]。Chen等人提出一个概率模型，以帮助用户建立可靠的数据集并公开了他们采用这种方法构造的JDK1.5 的数据集 [43]。

上述方法均为纯人工构造数据集的方法，由代码托管的开源软件通常采用issue-tracking工具来维护软件的行为信息 [44]。用户会在需求管理系统提交软件行为变更申请（issue），该issue可能是与软件相关的bug或者新的需求（new feature）等类型。研发人员会对用户所提交的issue 进行评审以决定是拒绝

该issue还是接受该issue。如果是后者，接下来会完成与issue相关的代码（可能是完成某个需求也可能是修复某个bug）。然后用户会将更改的代码提交到代码托管平台。很自然的这里issue 和提交的代码就形成了需求到代码的追踪关系。[45] 对需求管理系统中的new feature文本进做了进一步分类，使得我们对new feature 能有更好的利用。

2.5　本章小结

本章介绍了需求到代码可追踪生成的相关技术，并分析了现有方法的不足。同时本章介绍了在需求可追踪性相关领域，已有工作如何结合代码结构信息和用户反馈信息以提高信息检索所得候选追踪列表的精度。

第三章　结合代码依赖和用户反馈的软件可追踪生成方法

在第一章中我们讨论了需求到代码的可追踪性对于各种软件活动的重要意义，并在第二章相关工作中系统性的介绍了当前领域内的研究现状和不足。因此，本章详细介绍了我们提出的结合代码依赖和用户反馈的软件可追踪生成方法。在 3.1节给出了我们的问题定义并通过一个真实用例给出了我们的思路来源，在 3.2节介绍了我们所述方法的三个步骤，并且 3.3节结合具体用例详细介绍了我们方法的整个流程。

3.1　问题定义及思路来源

在软件开发过程中会衍生出大量的软件制品，例如需求文档、设计文档、测试集合和代码等。不同软件制品之间的追踪关系有利于软件后期的维护和演化。因此，建立软件制品之间的可追踪性是领域内的研究热点。其中需求到代码的可追踪性是其中最具有代表性也最具有挑战性的一类软件追踪性。其原因在于以下两点：（1）需求是软件系统中唯一记录了人对于完整系统功能的理解与期望的软件制品 [6]，而代码则是当前软件系统运行时行为的唯一真实反映，因此两者之间的关联关系是软件开发与维护人员最关注的；（2）需求到代码之间需要经过漫长的开发流程以完成语义上的转换。因此相对于任意其它两类软件制品之间的关联关系，需求和代码之间存在的语义偏差是最严重的。需求到代码可追踪性建立的最大困难在于，当需求和代码元素比较多时，人工的建立可追踪性会耗费大量的时间和精力，当前基于信息检索的自动化技术存在精度低的问题。因此，自动化生成高精度的需求到代码的追踪数据成为领域内的研究热点。接下来我们将给出我们的问题定义。

问题定义. 给定一个需求集合 R，该需求集合包含 d 个需求文本，同时给定代码集合（按粒度可分为包、类、函数，后两者为可追踪性中常用的代码粒度，我们这里的粒度是类） C，则需求到代码的可追踪性就是在$R*C$上判定任意两个r和c之间是否存在功能上相互关联。对于一个给定的需求r和给定的代码元素c，我们用布尔函数$isTrace(r,c)$来表示两者之间的关联关系，若c在代码中实现了（部分）r 的功能，则$isTrace(r,c)$=‘*trace*’(相关)。否则$isTrace$=‘*no-trace*’(不相关)。由r、c、$isTrace(r,c)$三者组成的一个三元组t就代表了r和c之间的一条追踪线索。则R 到C的可追踪性就是生成和维护两者之间的追踪线索集合$T = \{t = \{r, c, isTrace(r, c)\} | r \in R \wedge c \in C\}$。问题是：

表 3.1: 与UC15相关的代码元素列表

req	class
UC15	UpdateCodesListAction
UC15	UpdateNDCodesListAction
UC15	DrugCodesDAO
UC15	editDrugCodes_jsp
UC15	editNDCodes_jsp
UC15	DCBeanValidator
UC15	NDCodesDAO

Q：相对于传统方法，结合代码依赖和用户反馈的需求到代码的可追踪生成方法能否生成更高精度的候选追踪列表？

思路来源

我们以医疗管理系统iTrust中的需求UC15（UC15文本如图 3.1）和里面选择的11个类为例，阐述我们的思路来源。

iTrust是一个医疗管理系统，采用java语言编写。该系统包括需求和代码等制品。现在软件工程师需要得到实现需求UC15的代码元素列表。表 3.1为实现需求UC15的代码元素（类）即用户想要得到的数据。

UC15 update standards lists Use Case
...
15.2 Main Flow:
An administrator chooses to update the standards list for immunizations
[S1], diagnoses [S2], allowable drugs [S3], or allowable physical services [S4].
15.3 Sub-flows:
[S1] The administrator will maintain [add/update] a listing of allowable immunizations that an HCP can use. The administrator will store (1) the CPT code (The CPT code set accurately describes medical, surgical, and diagnostic services and is designed to communicate uniform information about medical services and procedures.
[S2] The administrator will maintain a listing of allowable diagnoses that an LHCP can use. The administrator will store (1) the ICD-9CM code (The International Statistical Classification of Diseases and Related Health
[S3] The administrator will maintain [add/update] a listing of allowable drugs that an HCP can use. The administrator will store (1) the National Drug Code (The National Drug Code (NDC) is a universal product identifier used in the United States for drugs intended for human use. National Drug Code Directory)
...

图 3.1: 需求文本UC15的内容

表 3.2为使用当前主流的信息检索方法对从代码中选取的11 个类与需求UC15做文本相似度计算得到的排序表。

该表按照相似度值递减的顺序排序， *Is Trace*项的‘X’表示两者存在追踪关系，观察排序表的结果知有些实现UC15的类如 UpdateCodesListAction、UpdateNDCodesListAction、 DrugCodesDAO 因出现了 *Update*、*Drug*、*Action*等在UC15文本中也大量出现的词项，使得它们和UC15的文本相似度比较高。但是也存在一些类虽然实现了UC15，但是因为单词失配问题和UC15的文本相似度并不高。例如 NDCodesDAO 它是一个与数据库操作相关的类，用于维护数据库中的 *national drug code*(美国使用的药品标识符，缩写为NDC), 但是UC15文本中出现比较多是 *national drug code* 而不是其缩写NDC，此外以 *DAO* 结尾的类往往与数据库操作有关，但是该词项也并未出现在UC15文本中，这就导致两者文本相似度不高。

通过观察这11个类的代码依赖图（代码依赖见图 3.2，带箭头的直线为直接依赖，虚线为数据依赖）。我们发现， NDCodesDAO和实现需求UC15的类 EditNDCodes_jsp、 UpdateCodesListAction均存在代码依赖关系。尤其是类 UpdateCodesListAction，两者既存在直接代码依赖，又共享数据。我们认为在用户确认 UpdateCodesListAction与需求UC15存在追踪关系之后，与其代码依赖紧密度比较大的 NDCodesDAO也很可能与UC15存在追踪关系。通过提升其相似度值可有效弥补单词失配带来的消极影响。基于此发现，我们提出了结合代码依赖和用户反馈来优化信息检索方法生成的候选追踪线索列表的研究思路。

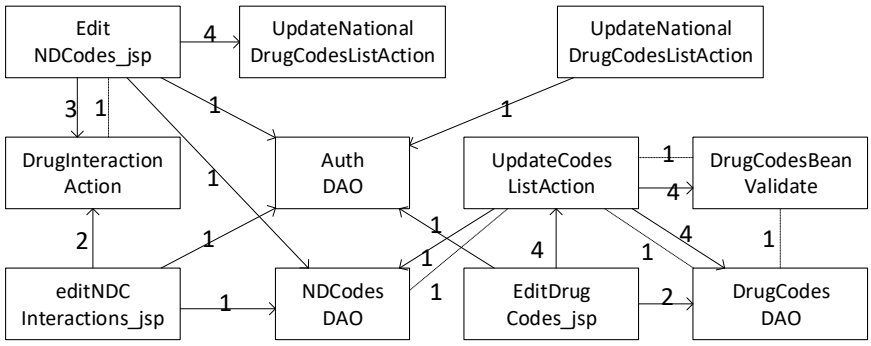

图 3.2: 代码依赖图

3.2　方法概述

在本小节，我们将介绍结合代码依赖紧密度分析和用户反馈的软件可追踪生成方法。我们把软件开发过程中产生的需求制品、代码制品以及代码依赖信息作为方法的输入，将候选追踪线索列表作为方法的输出。方法包含三个步骤（1）计算代码依赖紧密度，设置紧密度阈值生成候选代码域；（2）基于信息检

表 3.2: iTrust用例中基于信息检索的候选线索排序表

class	req	score	isTrace
UpdateCodesListAction	UC15	0.3524	X
UpdateNDCodesListAction	UC15	0.3124	X
DrugCodesDAO	UC15	0.2418	X
editDrugCodes_jsp	UC15	0.2112	X
editNDCInteractions_jsp	UC15	0.1816	
editNDCodes_jsp	UC15	0.1238	X
DCBeanValidator	UC15	0.1045	X
DrugInteractionAction	UC15	0.0953	
AuthDAO	UC15	0.0682	
NDCodesDAO	UC15	0.0487	X
viewResult_jsp	UC15	0.0031	

索方法，产生需求到代码的初始候选追踪线索列表。（3）根据用户反馈和代码依赖优化调整初始候选追踪线索列表。

算法流程图如图 3.3所示，在步骤一中，对不同的代码依赖进行分类整理，计算直接代码依赖和数据依赖紧密度，设置紧密度阈值，当两个类之间代码依赖紧密度大于阈值时两者在同一个代码域中，同一个域中的类功能比较紧密；在步骤二中，基于信息检索方法，计算类和需求之间的文本相似度，并按照文本相似度从大到小的顺序生成候选追踪线索排序表；在步骤三中，对未判断的代码域，根据各域内类与需求相似度最大值，对未判断代码域按照相似度值自大到小的顺序重排序。对于排名第一的代码域，取其域内与需求相似度最大的类交由用户判断。如果该类与需求具备相关性，则调整域内类和相关域外类与需求的相似度值。

3.2.1　计算代码依赖紧密度并生成代码依赖域

在本小节中，首先我们在 3.2.1.1节根据不同代码依赖的特点，对代码依赖进行分类。本文将代码依赖分为两类，直接代码依赖和数据依赖。接下来在 3.2.1.2节和 3.2.1.3节分别给出这两种代码依赖的计算方式。最后在 3.2.1.4节阐述代码域的产生过程。

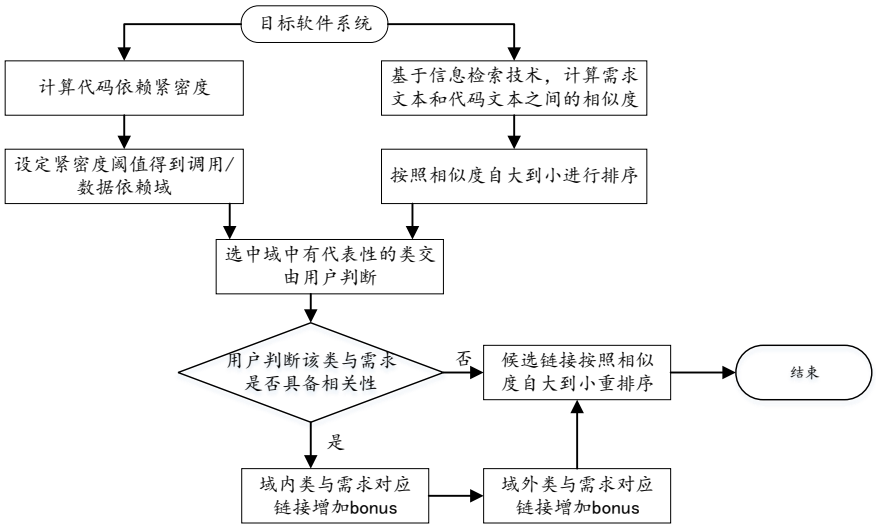

图 3.3: 基于用户反馈和代码依赖紧密度分析的软件可追踪生成方法流程

3.2.1.1　组织代码依赖关系

我们考虑类之间的四种依赖关系，类之间的方法调用、类继承、类使用和类的数据共享。C_a到C_b存在调用依赖说明至少存在一个C_a 的方法调用了C_b的方法。C_a使用C_b表明C_b是C_a的一个成员变量。两个类存在数据依赖表明两个类中存在访问同一个数据的方法，本文只关注在内存中的数据。前三种依赖可以通过静态代码分析的方法得到，与前三种代码依赖相比类之间的数据依赖捕获难度更大。然而之前工作 [16, 30]表明对于可追踪性数据生成任务，数据依赖与前三种代码依赖互补，相辅相成共同提高软件可追踪生成方法的精度。参考典型的基于信息检索使用代码依赖的方法 [16, 17]，我们把类之间的调用、继承和使用合并为一种类型的代码依赖即直接代码依赖，把类之间的数据共享看作另外一种代码依赖即数据依赖。由于前三种代码依赖的结构类似并且具有明显的方向性，并且类之间的调用、继承和使用在很大程度上是重叠的，所以本文把它们归为同一种类，用来对描述软件系统的控制流。类之间的数据依赖是指两个对象访问同一个数据，没有方向的概念，用来描述软件系统的数据流。对于以上两种不同的代码依赖，我们采用两种不同的方法计算它们的代码依赖紧密度。两种代码依赖的具体计算方式将在 3.2.1.2中介绍。

3.2.1.2　计算直接代码依赖紧密度

在本步骤中，我们将按照我们之前工作 [16]中的方法计算直接代码依赖和数据依赖紧密度。具体来说，对于类source和类sink之间的直接代码依赖，直觉上，如果两个类发生的方法调用或类使用次数比较多，则这两个类的关系比

较紧密。另一个需要考虑的因素是source总共调用了多少个类，即source的出度（out-degree）和sink被多少个类调用过，即sink的入度（in-degree）。sink的入度较小意味着类sink更专注于为类source提供服务，而类source的出度较小意味着类source 更加依赖于类sink所提供的服务。直接代码依赖计算公式如下：

$$Closeness_e = \frac{2N}{WeightOutDegree_{e.source} + WeightInDegree_{e.target}} \tag{3.1}$$

其中 N代表不同的函数调用与类引用的数量。在统计 *source*的出度和 *sink*的入度时，我们将每条直接调用依赖关系上的函数调用与类引用的数量作为出入度的权值。$WeightOutDegree_{e.source}$和$WeightInDegree_{e.sink}$ 分别表示两个加权的度量。$Closeness_{DC}$的取值范围是0到1的闭区间。

3.2.1.3　计算数据依赖紧密度

当两个对象访问同一个数据时意味着这两个对象对应的类存在数据依赖，直觉上，如果一种数据类型普遍存在于各个数据依赖中，例如一些基本数据类型，则此时不能说明访问它的类紧密度比较大；相反，如果某种数据类型只存在一个数据依赖中，则此时说明共享该数据的几个类功能比较紧密。所以在计算数据依赖紧密度时，本文认为不同的数据类型对相关类之间的数据依赖紧密度计算的重要程度不同。某种数据类型在数据依赖中出现的越少，其共享程度越强。基于此本文对引入逆向数据类型频率（idtf）对数据类型重要性进行量化，公式如下：

$$idtf = log\frac{N}{n_{dt}} \tag{3.2}$$

其中N代表所有类数据依赖的总数，而n_{dt}代表一个给定数据类型在所有数据依赖关系中出现的次数。根据*idtf*值，我们设定*idtf*的阈值$Threshold_{idtf}$来过滤*idtf*值小于阈值的数据类型。如果两个类之间共享数据类型的*idtf*值均小于阈值，则忽略两者存在的数据依赖。对这些数据类型以及相应的数据依赖关系的忽略是必要的，否则，即使这些数据依赖关系的紧密度值较小，但由于其能够将大量无关的类关联起来，仍然会对我们的算法产生不利的影响 [16]。与信息检索技术中的 *idtf*概念 [34]类似， *idtf*值实际上反映的是一个数据类型在整个代码的全局范围内被共享的程度。 *idtf* 值越高意味着该数据类型被两个类所独占的可能性就越大，也就意味着两个类之间更加紧密的交互。此外，对于类C_i 和C_j 之间的一条数据依赖，这条数据依赖中所有数据类型的数量与C_i和C_j所共享的全部数据类型的数量（各自通过其它数据依赖）之间的比值也能够反映这两个类在“本地”的数据共享程度，这个比值越高意味着交互越紧密。

由此，类数据依赖定义紧密度$Closeness_{CD}$如下：

$$Closeness_{CD} = \frac{\sum_{x\in\{DT_i\cap DT_j\}} idtf(x)}{\sum_{y\in\{DT_i\cup DT_j\}} idtf(y)} \tag{3.3}$$

其中，$idtf(x)$表示数据类型x的$idtf$值，DT_i与DT_j分别表示两个类所有数据依赖边的数据类型。$Closeness_{CD}$ 的取值范围是0到1之间。

3.2.1.4　建立候选代码域

我们定义图CDCGraph（Code Dependency Closeness Graph）为一个有序对$G =< V, E >$，其中V是代码中类的集合，并以其类名来标识。进一步的，CDCGraph 的顶点之间有两种类型的边。在两个顶点之间，E_{CD}代表一条直接代码依赖边而E_{DC}代表一条数据依赖边。CDCGraph同时在每条代码依赖关系上标注其计算得到的紧密度值。我们为两种代码依赖分别设定阈值$Threshold_{DC}$和$Threshold_{CD}$来对代码依赖图CDCGraph进行裁剪。去掉小于紧密度阈值的边形成一些联通子图即代码依赖域。我们认为每个代码域中的各个类相互紧密协作完成同一个需求。对于任意一个给定需求，我们取每个代码域中与需求相似度值最大的类作为该域的代表类。对于这些代表类按照相似度值从大到小排序，用户自上而下判断其中一些代表类与需求相关性。用户通过判断代表类与需求的相关性得出其所在代码域与需求的相关性。

3.2.2　基于信息检索方法生成候选追踪线索列表

我们利用信息检索技术在需求和类之间生成候选线索列表，这部分共包含四个步骤：

1. 创建文档库：对于代码中的每个类，我们抽取一个包含了这个类的注释、类名、方法名以及类成员名的文档。对于需求制品中的每条需求，我们抽取一个包含了其题目与内容的文档（对于有结构的需求用例我们抽取其前置条件、主要流程以及分支流程，对于无结构的需求我们直接引入所有文本信息）。

2. 标准化文档库：所有类和需求的文档都以信息检索领域内通用的标准化手段进行预处理，包括标识符拆分、骆驼分词、停用词过滤、词形还原和词根获取。

3. 索引文档库与文本相似度计算：我们使用tf-idf [34]对文档库中的每个关键词进行索引，并使用三个主流的信息检索模型来计算文档之间的相似度，即空间向量模型（VSM） [3]，隐式语义检索（LSI） [13]，以及基于概率的Jensen-Shannon模型（JS） [24]。

4. 建立候选线索列表：在生成的候选追踪线索列表中，我们按照每条候选追踪线索相似度值自大到小顺序排序。

3.2.3　结合紧密度分析和用户反馈对候选列表重排序

我们的在此步骤的增强策略受到以下三个发现的启发：（1）候选列表中相似度最大的候选线索往往是相关的 [14]。一个给定需求一般在代码的某块联通域内被实现，而不是随机散布在代码各处，这也是我们前面设置代码依赖紧密度阈值生成代码域的依据 [23]。（3）通过事先验证候选追踪线索得到的用户反馈和代码紧密度分析可以改善基于信息检索的软件可追踪生成方法的精度 [17]。因此，首先我们通过设置代码依赖紧密度阈值，在代码依赖图中去掉紧密度值小于紧密度阈值的边，形成一些联通子图，我们称这些联通子图为候选代码域。基于这些代码域我们提出一个包含四个步骤的候选追踪线索重排序算法。第一，对于未判断代码域，根据各域内类与需求相似度最大值，对未判断代码域按照相似度值自大到小排序。对于排名第一的代码域，取其域内与需求相似度最大的类交由用户判断，如果该类与需求具备相关性则执行步骤二，否则直接跳至步骤四。第二，调整域内类对应候选追踪线索相似度值。第三，调整域外类对应候选追踪线索相似度值。第四，按照相似度值从大到小的顺序对所有候选线索重排序。由用户决定是否需要继续判断候选代码域，如果用户继续判断则重复步骤一，否则算法流程结束。

3.2.3.1　用户判断候选代码域与需求相关性

对于任意一个给定需求，我们取每个代码域中与需求相似度值最大的类作为该域的代表类。对于这些代表类按照相似度值从大到小排序，用户自上而下判断其中一些代表类与需求相关性。用户通过判断代表类与需求的相关性得出其所在代码域与需求的相关性。如果相关则调整域内类和域外相关类对应候选追踪线索的相似度值。

3.2.3.2　调整域内类对应候选线索相似度值

我们考虑两个因素给予域内类对应候选追踪线索相似度奖励。（1）代码域的大小，即代码域包含类的个数。我们认为代码域中类的个数越少，类之间的

关系越紧密。（2）域内其它类与代表类C_{req}的交互紧密度程度。针对两种不同的代码依赖我们采用两种方式来量化两个类之间的交互程度

1. 直接代码依赖，我们在代码依赖图中寻找域内类C_{in} 到代表类C_{req}的路径，该路径需要满足一直调用或者一直被调用的关系。

2. 数据依赖只考虑两个类之间的直接数据依赖，不考虑传递性。

域内类对应候选线索相似度根据以上两种依赖关系可以拿到两次奖励，公式如下：

$$Bonus_{DC} = \prod_{x \in PATH} Closeness_{DC}(x) \tag{3.4}$$

公式解释：域内其它类（C_{in}）到域代表类（C_{req}）可能存在多条代码依赖路径，$PATH$为紧密度乘积最大的一条。$Closeness_{DC}(x)$ 为直接代码依赖x的紧密度值。综合考虑域大小和域内类与代表类交互程度，域内类对应候选追踪线索相似度值更新公式为：

$$IR_{in} = (IR_{current} + \frac{IR_{top}}{RegionSize})(1 + Bonus_{DC} + Closeness_{CD}(x)) \tag{3.5}$$

公式解释：$IR_{current}$是域内类C_{in}对应候选追踪线索当前的相似度值，IR_{top}为域内类对应候选追踪线索相似度最大值，即域内代表类（represent class）对应候选追踪线索相似度值。$RegionSize$为被判断代码域的大小，即代码域域内类的个数。

3.2.3.3　调整域外类对应候选线索相似度值

接下来我们对代码域域外的类对应候选追踪线索相似度值给予奖励，奖励的标准是这些类与域内代表类（C_{rep}）的交互程度。与前一步骤类似，我们针对不同的代码依赖关系采用不同的奖励方法。从直接代码依赖的角度出发，从一个代码域外的类C_{out}出发我们尝试找到一个通往被判断域内代表类C_{rep}的路径。与域内类到域内代表类之间的路径一样，域外类到域内代表类的一个合法路劲应该满足要求：这一路径是单向的，即C_{out}传递性到C_{rep} 或者C_{rep}传递性到C_{out}；域外类对应候选追踪线索相似度值更新公式为：

$$IR_{out} = IR_{current} + IR_{top}(Bonus_{DC} + Closeness_{CD}(x)) \tag{3.6}$$

其中$IR_{current}$为域外类C_{out}对应候选追踪线索当前的相似度值，IR_{top}所有类与给定需求相似度的最大值。在这一步骤中有可能所有域外类对应候选线索相似度都能得到奖励。我们希望给予与域代表类交互紧密的域外类更多的奖励以提升它们在列表中的排序。

Algorithm 2: recovery requirement traceability through user feedback and code dependency closeness analyse

Input: req,$List$,$Threshold_{DC}$,$Threshold_{CD}$

Output: $List$

```
1  irValue_top ← getTopRankedIRValue(List,req);
2  regions ← CDCGraph.prune(Threshold_DC,Threshold_CD);
3  while (notstoppingcriterion) do
4      Region_top ← regions.getTopRegion(list,req);
5      class_req ← topRegion.getRepresentativeClass(req);
6      The user verifies the link(req,class_rep)
7      if (req,class_rep is a relevant trace) then
8          foreach link in List do
9              bonus_DC ← getMaxBonusByDC(link.class,class_rep);
10             bonus_CD ← getBonusByCD(link.class,class_rep);
11             if Region_top contains link.class then
12                 link.value ← (link.value+irValue_top/Region_top.size()) (1+bonus_DC+bonus_CD);
13             else
14                 link.value ← link.value+irValue_top*(bonus_DC+bonus_CD);
15             link.value ← min(link.value,irValue_top);
16     Hide Region_top from regions
17     Hide verified links and reorder List
```

3.2.3.4　候选线索重排序

当用户没有意愿继续候选代码域与需求相关性，或者已经没有未被判断的候选代码域时，此时算法流程结束，否则进入下一轮迭代。在每一轮迭代中，有些候选追踪线索的相似度值会被更新，候选追踪列表排序会发生变化。然而，为了避免给不相关类对应追踪线索相似度太大的奖励，我们要求追踪线索相似度更新之后的值不超过候选线索相似度最大值。并且我们保证我们的算法是稳定排序。算法 2描述了我们的算法。

3.3　算法案例分析

本小节以实验系统iTrust为例，详细阐述本文方法的执行过程。iTrust是一

个医疗管理软件，采用java语言编写。该系统包括uc和src目录，其中uc目录里面是一组用于描述需求的文本文件，src目录为该项目的代码文件。接下来我们以iTrust系统的UC15需求为例，详细介绍按照我们的方法建立UC15到其相关代码元素可追踪性的过程。

计算代码依赖紧密度

图 3.2为本用例中的类形成的代码依赖图，数据依赖由没有箭头的虚线表示，虚线上的值为虚线连接的两个类之间共享数据类型的数量；带有箭头的直线为直接代码依赖，箭头的方向代表调用、继承、使用发生的方向，直线上的值代表两个类之间发生的直接代码依赖次数。例如： UpdateCodesListAction 和 DrugCodesDAO 之间存在直接代码依赖，它包括两个方法调用和两个类使用。 DrugCodesDAO 和 DCBeanValidator 存在数据依赖，在运行 UpdateCodesListAction的构造函数期间， DrugCodesDAO生成一个 DrugCodesBean 对象，并将其传递给 DCValidator的 *validate*方法。因为以上三个类共享数据类型DrugCodesBean。

表 3.3为本例子中涉及到的数据依赖

表 3.3: iTrust用例中存在的数据依赖

Source	Data Type	Target
NDCodesDAO	NDCode	UpdateCodesListAction
UpdateCodesListAction	LOINCBean	DrugCodesBeanValidator
UpdateCodesListAction	LOINCBean	DrugCodesDAO
DrugCodesBeanValidator	LOINCBean	DrugCodesDAO

根据 3.2.1的代码依赖紧密度计算公式，分别计算用例中涉及代码依赖的直接代码依赖紧密度和数据依赖紧密度。如图 3.2， editDrugCodes_jsp与 UpdateCodesListAction 之间的直接代码依赖紧密度为2*4/((1+4+2)+4)=0.62，而 editDrugCodes_jsp和 AuthDAO之间的直接代码依赖紧密度为 2*1/((1+4+2)+(1+1+1+1))=0.18，这表明与 AuthDAO相比， editDrugCodes_jsp与 UpdateCodesListAction 关系更紧密。根据 3.2.1，对于数据依赖紧密度的计算需要先计算个数据类型的idtf值。如表 3.4，展示了图 3.2 中的共享数据类型，其中“Occurrences”列代表一个数据类型在iTrust系统的所有类数据依赖关系中出现的次数。观察表 3.4可知，String 类型的出现次数远大于NDCode 和 DrugCodesBean。由于*String*类型普遍存在于各代码依赖中，所以当两个类共享数据类型*String*时，不能说明这两个类关系比较密切。我们将$Threshold_{idtf}$设置为1.4.因此图中 edit-

表 3.4: iTrust用例中各数据类型的idtf值

#	Data Type	occurrences	Idtf Value
1	LOINCBean	9	2.7310
2	NDCode	147	1.5179
3	Java.lang.String	1118	0.6368

NDCodes_jsp和 DrugInteractionAction之间的数据依赖（仅基于*String*）会被过滤掉。如图 3.2的类 UpdateCodeListAction和类 DrugCodesDAO共享数据类型 DrugCodesBean，其紧密度为2.60/(2.60+1.53)=0.63，而类 UpdateCodeListAction和类 NDCodesDAO共享数据类型 NDCode，两者之间紧密度为1.53/(1.53+2.60)=0.37。结果显示相比于类 NDCodesDAO，类 UpdateCodeListAction与类 DrugCodesDAO的关系更紧密。图 3.4为图 3.2对应的代码依赖紧密度图。

如上 3.2.1.3所述，计算类之间数据依赖之前需要先计算各数据类型的 *idtf* 值，用例中涉及数据类型的 *idtf* 值见表 3.4： 根据上述方法中的紧密度公式，计算类之间的代码依赖紧密度，代码依赖紧密度图如图 3.4所示，设置直接代码依赖阈值0.6，设置数据依赖紧密度阈值0.6，紧密度高于阈值的类之间会形成代码域。如图 3.4，虚线框表示代码域，图中有两个代码域。

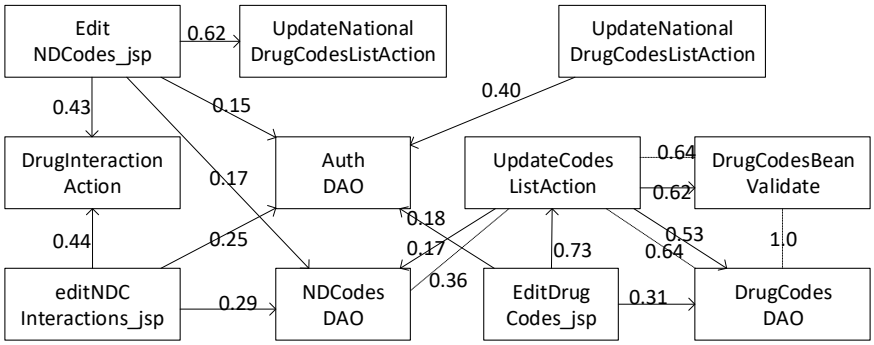

图 3.4: 代码依赖紧密度图

基于信息检索方法计算需求和代码之间的文本相似度

对需求uc和代码src中的文本，进行文本预处理。处理过程包括移除停用词，词形还原和词干提取等操作。特别的，对于从代码中抽出的文本首先需要根据骆驼命名法等变量规则进行分词。例如函数名initializeDrugCodes，首先根据驼峰规则划分为initialize、drug 和codes，然后根据进行词形还原和词干提取，分别得到init、drug 和code。对于预处理后的需求文本和代码文本，利用空间向量模

型计算文本之间的相似度，表 3.2为计算后的文本相似度从大到小排序表。最后一列isTrace表示该代码元素和需求之间是否具有相关性。‘X’表示两者之间存在相关性。

结合代码依赖紧密度分析和用户反馈对候选列表重排序

我们设置直接代码依赖阈值0.6，设置数据依赖紧密度阈值0.6，形成的代码域如图 3.5。

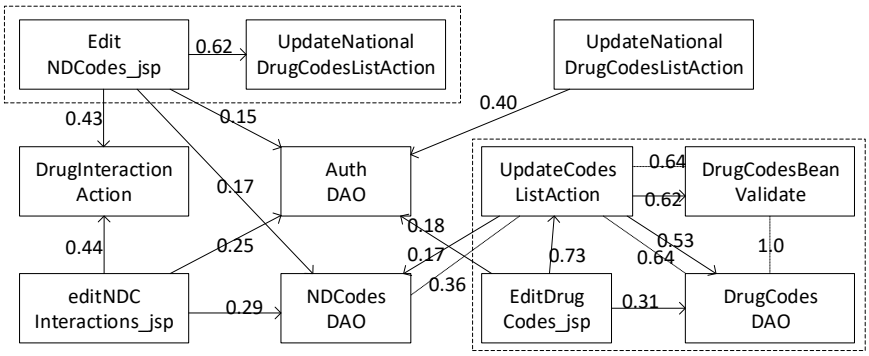

图 3.5: 候选代码域

对代码域进行排序，根据上一步骤的文本相似度计算结果，我们按照每个域中的最大值自大到小进行排序。先将第一个代码域交由用户判断即判断UpdateCodesListAction 与UC15 的相关性。由表 3.1知，两者具备相关性（实际运用中由用户判断代表类和UC15相关性）。对于UpdateCodesListAction所在的域，根据上述算法需要调整其它域内类对应候选追踪线索的相似度值。域内类有DrugCodesDAO、EditDrugCodes_jsp、DrugCodesBeanValidator，以EditDrugCodes_jsp为例，根据 3.2.3.2所述方法 EditDrugCodes_jsp与UC15相似度更新之后的值为：$min(0.3524, (0.2112 + 0.1175) * (1 + 0.73)) = 0.3524$。对于域外相关类，根据图 3.2所示代码依赖图知，只有类 NDCodesDAO与域内代表类UpdateCodesListAction存在代码依赖，并且是同时存在数据依赖和直接代码依赖，因此需要对这个类对应候选追踪线索相似度值更新。根据 3.2.3.3对域外相关类对应候选追踪线索的相似度值调整公式， NDCodesDAO与UC15相似度更新之后的值为$min(0.3524, 0.0487 + (0.17 + 0.36) * 0.3254) = 0.2355$ 同理，如果用户愿意继续判断，对由类 editNDCodes_jsp和 UpdateNDCodesListAction组成的代码域进行判断，并根据用户判断结果调整相关追踪线索相似度值。

在例子中使用我们的算法，用户判断完两个类之后，根据用户反馈调整相关候选追踪线索相似度值，候选追踪线索重排序的结果如表 3.5，由表中可以看出，本文的方法提升了实现UC15的类在排序表中的位置。

表 3.5: 结合用户反馈和代码依赖优化之后的候选追踪线索排序表

class	req	score	isTrace
UpdateCodesListAction	UC15	0.3524	X
editDrugCodes_jsp	UC15	0.3524	X
DCBeanValidator	UC15	0.3524	X
DrugCodesDAO	UC15	0.3524	X
editNDCodes_jsp	UC15	0.3524	X
UpdateNDCodesListAction	UC15	0.3124	X
NDCodesDAO	UC15	0.2954	X
editNDCInteractions_jsp	UC15	0.2468	
DrugInteractionAction	UC15	0.1845	
AuthDAO	UC15	0.1816	
viewResult	UC15	0.0031	

3.4　本章小结

在本章中，我们提出了一种结合代码依赖紧密度分析和用户反馈的软件可追踪生成方法。一方面通过设置代码依赖紧密度阈值划分代码域，使得功能紧密的代码元素位于同一个代码域中；另一方面，对于给定需求，将各代码域中有代表性的代码元素交由用户判断与该需求相关性，根据用户反馈信息调整相关代码元素对应候选线索的相似度值。并且，本章结合一个真实软件系统需求UC15的可追踪性建立过程，对方法中的流程做了详细介绍。

第四章　基于代码托管平台的数据组织及方法验证

本章通过设计实验来验证第三章所述方法的有效性和实用性。如相关工作所述，当前领域内公开用于验证可追踪方法的数据集通常是一些人工标注的小型系统数据集。本章 4.1节通过分析整理由代码托管平台（本章中的代码托管平台为github）管理的开源软件在issue-tracking 工具（本章中的issue-tracking工具为Jira）的行为信息得到了其需求到代码的追踪关系。通过运行软件系统自带的用于验证自身功能的测试用例得到本文方法所需的高质量代码依赖集。本章 4.2节中的使用一个被领域内广泛用于软件可追踪方法验证的高质量数据集iTrust [46]和三个被广泛用于日常实践的开源软件系统Maven [47]、Pig [48]和Infinispan [49]来对本文方法进行验证，该小节选取了一些领域内常用的度量指标将本文的方法效果和基线方法对比，并对实验数据进行了细致分析。实验表明，本文的方法只需少量的（3.5%）用户反馈信息在精度上即可显著优于基线方法。

4.1　数据组织

如前文所述，一方面，我们的方法需要基于日常实践的可追踪数据集来验证其实用性；另一方面，高质量的代码依赖信息是我们方法的重要输入。本节首先在 4.1.1节介绍开源软件需求到代码可追踪数据的构造过程，然后在 4.1.2节介绍了对于由不同构建工具管理的开源软件，如何通过对其配置文件进行相应配置使得能够将代码依赖捕获工具插桩到运行其测试集的虚拟机中，进而动态获得其代码依赖子集，最后，对代码依赖子集进行合并得到代码依赖。

4.1.1　用于方法验证的可追踪数据集组织

本小节主要介绍 4.2节中所用开源软件需求到代码可追踪数据集的构造过程。我们选取的Maven、Pig和Infinispan在github [50]上更新活跃，吸引了大批贡献者。当前大量Java项目由Maven构建工具来管理，Infinispan作为分布式缓存系统则被大量工业集群使用，Pig作为一个大数据处理平台被广泛用于大数据开发中。这三个开源软件由issue-tracking工具JIRA [51]来管理其软件行为变更信息（bug 反馈、需求变更和增加新需求等），被广泛用于缺陷定义、客户服务、需求收集、流程审批、任务跟踪和敏捷管理等领域。在JIRA中用issue 来描述软件行为信息。以Pig为例，图 4.1为 为实验系统Pig在JIRA上

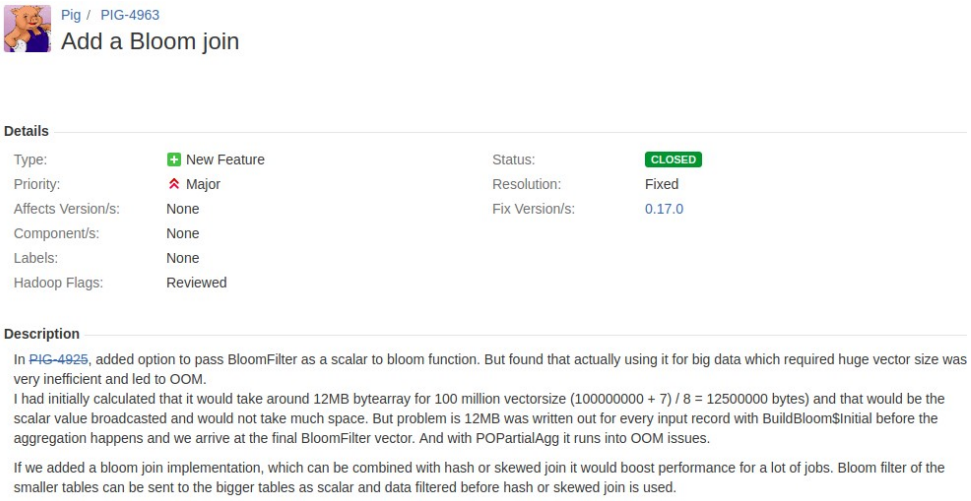

图 4.1: JIRA上的一条issue信息

的一个issue信息，根据该issue的类型我们知道该issue描述的软件行为信息是增加新功能。通过issue的描述信息我们得知该功能是用来解决大数据的处理问题，以及具体是解决方案是添加一个bloom join是实现。此外，issue还包含大量其它信息，例如issue的优先级，会影响的版本，issue当前的状态等。我们会综合利用这些信息筛选出描述需求的issue。上述开源软件由Git[52]来管理其代码变更信息，Git是一个开源的分布式版本控制系统，可以有效、高速的处理从很小到非常大的项目版本管理。通常用户在issue-tracking工具提交issue 之后，软件研发人员会对issue 进行评审以决定是拒绝还是接受该issue。如果是后者，软件研发人员会针对该issue完成相应代码变更，并将代码提交到代码托管平台。 图 4.2为针对图 4.1所示issue的一次代码提交（commit）。Git记录了该commit所引起的代码变更（我们这里只看.java结尾的文件，因为我们的对源代码的研究粒度是java类）。由图可知，这里变更的文件有 PigConfiguration.java、 MRCompiler.java和 PigCombiner.java 等。我们能很自然的得出issue PIG-4963所描述的需求与以上java 类存在追踪关系。这也是我们为Pig、Maven和Infinispan建立需求到代码可追踪数据集的思路来源。

[44]用8张数据库表对开源项目在Git和JIRA的信息进行了分类整理（表

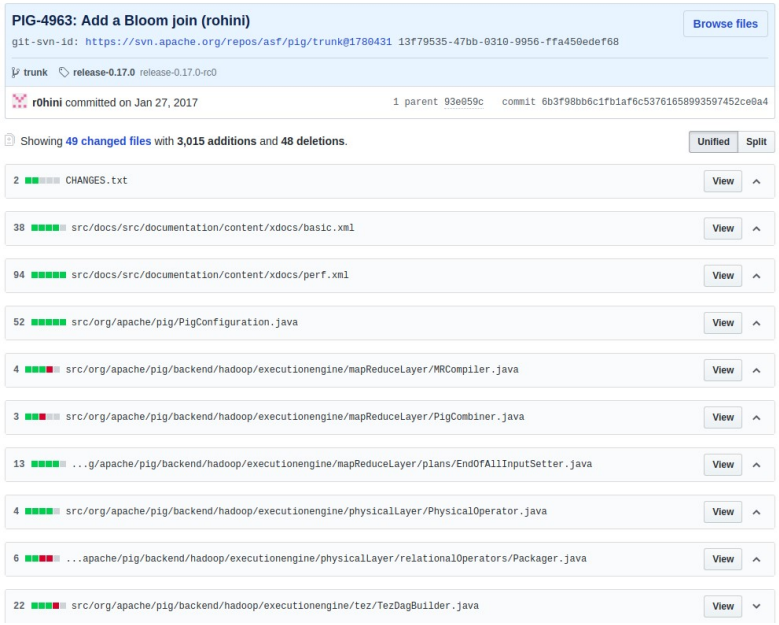

图 4.2: 针对issue的git commit

4.1描述了各数据表的内容）。

我们通过对这些数据库表进行连接与合并等相关操作（图 4.3为我们对数据库表的操作流程），最终得到需求到代码的追踪关系。具体来说，首先，我们对设置规则对issue进行过滤筛选，得到描述需求的issue，issue筛选规则如下：

1. 忽略带有关键词testing或者testcase的issue，因为这些issue往往是和测试用例有关系，不是软件系统的功能性需求。

2. issue必须已经被完成并且有与之关联的code_change，只有这样我们才能生成需求（issue）到代码的追踪关系。

3. 我们只选择优先级为Major和Critical的issue，因为这些往往是重要的系统功能性需求。

4. 我们只选择类型为New Feature（或Feature Request）的issue，这种类型的issue通常用来描述系统功能性需求 [45, 53]。并且这种需求粒度比较接近传统的需求（例如iTrust的需求），类型为Bug的issue 往往粒度太小，类型为Task的issue又往往粒度过大。

接下来，我们通过对筛选之后的数据库表 issue和数据库表 chang_set_link做连接操作，得到 issue和代码提交之间的关联关系，每一次代码提交都用一个全

表 4.1: 数据库表内容描述

table	description
issue	记录每个issue的具体信息，标识符、描述信息、类型、时间戳等
issue_link	记录issue之间的关系，例如从属、依赖、重复等
issue_component	记录改issue所属的模块
issue_fix_version	记录改issue被完成的版本
chang_set	一个代码提交的相关信息，代码提交者、时间戳、描述代码提交的文本
chang_set_link	该表记录代码提交和issue之间的对应关系
code_chang	一次代码提交引起的代码文本变更，例如增加或减少了哪些代码
project	项目的元数据，包含JIRA和git仓库的信息

局唯一的hash来表示。同时，我们通过对数据库表 change_set和 code_change做连接操作得到代码提交和代码变更之间的对应关系，然后我们通过对以上生成的两个中间数据库表进行连接操作即可得出 issue和代码变更之间的追踪关系。最后根据表 issue_link中描述的issue 之间的关系，我们将关系为从属（ part-of）、重复（ duplicate）、替代（ supersede）的 issue对应追踪线索进行合并。最终，我们得到了需求（issue 中用来描述软件行为的 description文本）和代码（Git中与issue对应的代码变更）之间的追踪关系。

4.1.2　用于方法输入的代码依赖数据组织

本小节首先对本文使用的代码依赖捕获工具 [30]进行简要介绍，然后介绍由不同构建工具管理的软件系统，通过运行其用于验证自身系统功能的测试用例得到其对应代码依赖的过程。图 4.4为我们代码依赖捕获工具的结构图，其基本原理是：使用标准JDK 提供的JVMTI接口，监听Java 虚拟机中产生的四个JVMTI 事件：类成员读取事件、类成员修改事件、函数进入事件以及函数返回事件并注册这四个事件的回调函数，在回调函数中将事件引起的函数进入、返回记录和数据访问记录保存到本地数据库中。最终，通过对这些数据进行分析处理得到我们所需的代码依赖。我们通过对软件系统所用的构建工具进行相应配置，使得在运行测试其测试用例时，代码依赖捕获工具会插桩到运行该测试用例的虚拟机中，进而得到软件系统对应的代码依赖信息。代码依赖工具得到的是方法级代码依赖，而我们实验中需要用到的是类级别的代码依赖，因此我们将方法级代码依赖按照如下规则转化为类级别代码依赖：

1. 类之间调用依赖关系从方法调用依赖关系中抽象出来，调用边的权值为发生不同方法调用的次数。

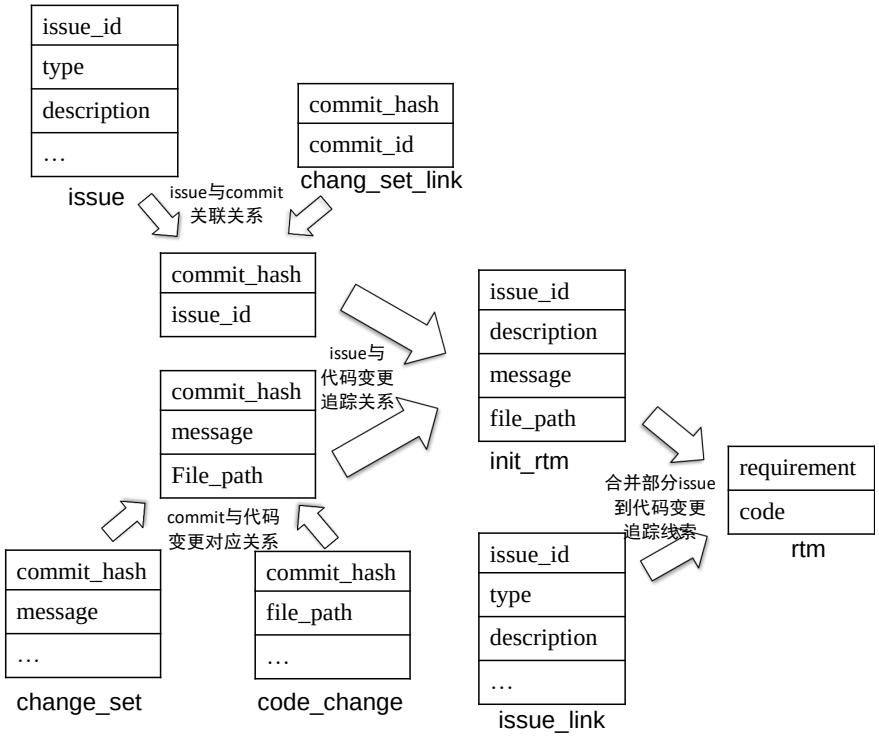

图 4.3: 需求到代码追踪关系整理过程

2. 类之间数据依赖关系从方法数据依赖关系中抽象出来，数据依赖边的权值为两个类之间共享数据类型的个数。

3. 类之间使用关系从方法之间数据依赖关系中抽象出来。

4. 类之间继承关系从方法调用中得到，子类在调用自己的构造方法之前会去调用其父类的构造方法。

我们基于如下原因选取该工具：

1. 虽然能够捕获函数调用的动态分析工具很多（例如著名的The Eclipse Test & Performance Tools Platform，TPTP；或Java Plug-in Framework，JPF），但是这些工具都不能捕获函数之间的数据共享。

2. 基于java虚拟机的运行时检测，这个工具可以捕获实际执行的代码依赖关系并正确处理多态。由于该工具是在系统运行时刻对系统实际行为的观察与记录，因此不会产生非正确的代码依赖关系。

3. 该工具可以在一次系统运行中同时获得以上四种代码依赖，由于所执行测试集合的不完备性，动态分析无法保证捕获的代码依赖关系的完整性，虽

然这一问题无法避免，但是我们认为部分确实的代码依赖关系在其缺失程度大致相同的情况下，并不会影响最终的实验结果，其原因在于我们用一个整合的工具在运行相同测试集合的时候同时捕获函数调用和数据依赖。

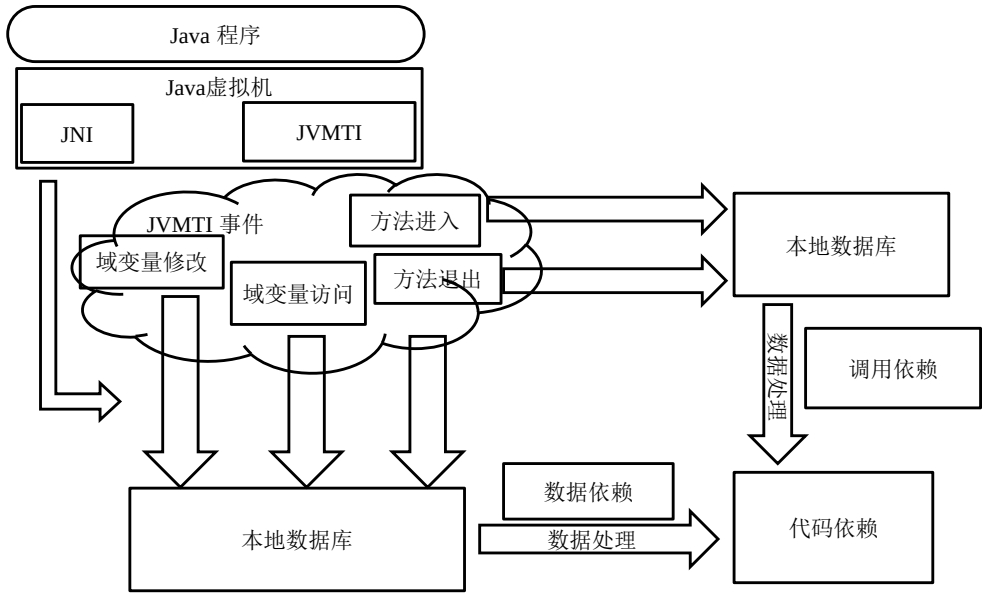

图 4.4: 代码依赖捕获工具结构图

最终我们便可得到目标系统的代码依赖信息。综上所述，通常我们获取代码依赖的方式是运行目标代码，将我们的代码依赖捕获工具插桩到运行目标代码的虚拟机中，然后我们对捕获的数据进行处理，最终得到代码依赖信息。接下来我们将阐述本文实验中的所用实验系统的代码依赖捕获过程。

本文中的实验系统分为两类：一种是iTrust这样的应用软件，它是一个医疗管理软，能独立运行。我们运行该系统同时插桩代码依赖捕获工具，根据用户手册来运行该系统具有的各种功能从而捕获尽可能完整的代码依赖。另一种是Pig、Maven、Infinispan这样的支撑软件，与应用软件不同，使用这些支撑软件的多种功能并不容易。观察得知，本文实验系统中的开源支撑软件具有如下结构特点：

1. 具有丰富用于验证自身系统功能的测试集合。（图 4.5为infinispan的结构图，每个模块都有相应测试集）

2. 采用构建工具maven或ant管理

因此，我们提出通过运行软件系统用于验证自身功能的测试用例得到方法所需代码依赖的方法。具体来说，首先，通过运行软件系统各模块的测试用例

得到代码依赖子集，然后，我们根据一些合并规则对代码依赖子集进行合并，最终得到第三者章所述软件可追踪生成方法所需的代码依赖。该方法简单高效，不需要软件系统能够独立运行，并且可以通过构建工具批量运行测试集。图4.5为该方法的流程图。

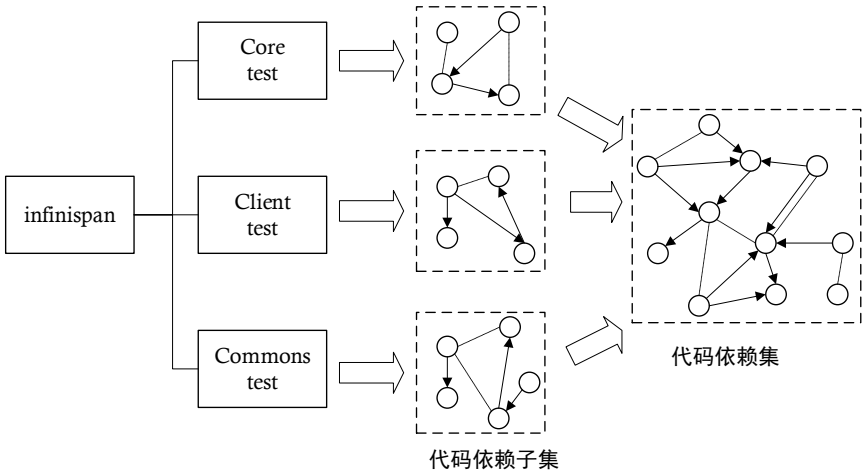

图 4.5: 运行测试用例捕获代码依赖的过程

本文实验中涉及到的基于日常实践的开源系统由两种构建工具来管理：Maven、Ant [54]。接下来，我们将在 4.1.2.1节和 4.1.2.2节分别介绍这两种构建工具下的代码依赖捕获过程。

4.1.2.1　由Maven管理的软件系统代码依赖获取

构建工具Maven是一个跨平台的项目管理工具，本身是一个Java开源项目。Maven 主要服务于基于Java平台的项目构建、依赖管理和项目信息管理。Maven帮助我们标准化构建过程，抽象了一个完整的构建生命周期模型，每个生命周期都由相应的插件来负责。Maven在运行测试集发生在测试阶段，该阶段由surefire插件负责。当执行*mvn test*命令时，surefire插件会运行所有以Test结尾的类，我们将代码依赖捕获工具插桩到运行测试集的虚拟机中即可获得测试集对应的代码依赖捕获数据。需要说明的是：在Maven 的测试周期，默认情况下Maven会开启新的虚拟机进程用来运行测试集，而不是用运行Maven的虚拟机运行测试集。即我们需要把工具插桩到运行测试集的虚拟机中。通过阅读maven-surefire-plugin [55]文档，我们通过对surefire模块两个重要参数的配置完成工具插桩进而完成代码依赖的捕获。

1. forkCount 用来指定运行测试集的虚拟机进程数目，默认为1。如果该数字以C结尾，运行虚拟机的进程数目为这个数字乘以CPU 内核数。如果该数字设置为0，则意味着不在启动新的虚拟机进程运行测试集，由主虚拟机进程（运行Maven的虚拟机）来执行测试集。

2. argLine 设置jvm启动参数

基于以上内容我们有两种方法通过运行测试集捕获代码依赖：（1）配置pom文件里的maven-surefire-plugin插件，通过*argLine*将工具插桩到运行测试集的虚拟机中。（2）通过设置forkCount参数为0，用运行maven的虚拟机进程来运行测试集，此时在启动Maven时插桩代码依赖捕获工具即可。接下来我们通过对代码依赖子集进行合并得到软件对应的代码依赖，代码依赖合并规则如下：

1. 对于直接代码依赖：在测试用例1中存在A调用B，在测试用例2中存在B调用C，由此我们可以得出A调用C。

2. 对于数据依赖：在测试用例1中存在A和B同访问类型为D的数据，在测试用例2中存在B和C同访问数据类型为D的数据，则A,B,C共享数据类型D。

4.1.2.2　由Ant管理的软件系统代码依赖获取

构建工具Ant最早用来构建著名的Tomcat，我们可以将Ant看成是一个Java版本的Make，Ant使用XML定义构建脚本，相对于 *Makefile* 更加友好。Ant可以用junit [56]框架来运行软件的测试集。在运行测试集的过程中为了捕获代码依赖我们需要将代码依赖捕获工具插桩到运行测试集的虚拟机中。与 4.1.2.1类似这里介绍以下junit的两个重要参数：fork、jvmarg。

1. fork 当开启此参数（fork=“yes”）时，Ant会启动新的java虚拟机来执行测试集

2. jvmarg 当开启参数fork时，jvmarg用于向这个新开启的虚拟机传递jvm参数

基于以上内容，与被构建工具Maven管理的项目类似，我们有两种方法通过运行用Ant管理的软件系统的测试集得到代码依赖：（1）配置文件build.xml 中的junit，开启fork参数并通过jvmarg参数向新开启的虚拟机传递jvm参数（2）配置文件build.xml中的junit，不开启fork 参数（fork=“false”），此时在启动Ant时插桩代码依赖捕获工具即可。在我们的实验中，我们都是采用第一种方法插桩

代码依赖捕获工具获取代码依赖子集，这是因为我们使用的代码依赖捕获工具为单线程，如果采用第二个方法会出现几个进程同时向数据库写数据的同步问题。

4.2 方法验证

本节我们通过实验以验证结合代码依赖紧密度分析和用户反馈的软件可追踪方法的有效性。接下来，将具体阐述我们的实验设置及实验结果与分析。

4.2.1 实验系统

我们的实验部分是基于四个现实世界、来自不同领域的软件系统。iTrust（在线医疗档案管理）、Maven（构建管理工具）、Pig（编译器）、Infinispan（数据库）。iTrust是在这个领域内被广泛使用的数据集，其软件可追踪数据集由系统开发与维护人员提供。但是该系统提供的高质量数据集是方法粒度的，即表达的是需求和方法之间的追踪关系。而本文的方法是基于类粒度的，因此在iTrust中，我们将需求和方法之间的追踪关系转化为需求和类之间的追踪关系。我们的转换规则是：若需求和方法之间有追踪关系，则我们认为需求和这个方法所在的类有追踪关系。其它三个实验系统通过 4.1.1节所述方法获取其软件可追踪数据集。表 4.2列举了这四个实验系统的基本信息。

表 4.2: 实验系统的相关细节

	iTrust	Maven	Pig	Infinispan
版本	13.0	3.5.2	0.17.0	9.2.0
编程语言	Java	Java	Java	Java
千行代码（KLoC）	43	101	365	521
代码（类）	138	94	236	388
需求（用例）	34	36	68	237
需求对应类的平均数	8	4	5	6
调用依赖	274	182	1998	1777
数据依赖	4792	1164	5405	6076
追踪关系	255	155	356	1515

4.2.2 评价指标

为了验证我们结合代码依赖和用户反馈方法的有效性和实用性，我们首先

引入领域内常用的两个重要度量，精确度（precision）和查全率（recall）。对应公式如下：

$$precision = \frac{|relevant \cap retrieved|}{|retrieved|}\% \tag{4.1}$$

$$recall = \frac{|relevant \cap retrieved|}{|relevant|}\% \tag{4.2}$$

参数解释：其中 *relevant*是相关的候选追踪线索集合。 *retrieved*表示软件可追踪生成方法所返回的候选追踪线索集合。由于自动化可追踪生成方法所返回的是一个候选追踪线索排序列表，即按照文本相似度值从高到低排序。所以一种常用的比较方法的方式是在不同的查全率下比较不同方法之间的精确度，可以由一条 *Precision-Recall*曲线展示。为了进一步衡量不同自动化软件可追踪生成方法返回结果的整体质量，我们选用了领域内另外两个常用指标：AP（Average Precision）与MAP（Mean Average Precision）。其中 *AP*用于度量全部查询（需求）所检索相关文档的排序质量，计算公式如下：

$$AP = \frac{\sum_{r=1}^{N}(Precision(r) \times isRelevant(r))}{|RelevantDocuments|} \tag{4.3}$$

参数解释： *r*表示被查询实体（类）在排序表中的位置，N表示候选追踪线索的总数。 *Precision(r)*表示对于前 *r* 个候选追踪线索的准确率。 *isRelevant()*为一个二值函数，如果这条候选线索有效则返回1，否则返回0。此外， *MAP*用于描述不同查询（需求）所检索的相关文档（类） *AP*的平均值。计算公式如下：

$$MAP = \frac{\sum_{q=1}^{Q} AP(q)}{Q} \tag{4.4}$$

参数解释： *q*表示一次查询而 *Q*表示查询的总数。为了更全面的验证方法有效性，我们同时使用 *AP*和 *MAP*两个度量指标。

4.2.3　阈值设置

我们需要校准四个阈值，$Threshold_{idtf}$、$Threshold_{DC}$、$Threshold_{CD}$和LSI的 *k* 值。根据之前的 [16, 30]案例分析，我们设置$Threshold_{idtf}$的值为1.4，用这个阈值来忽略普遍出现的数据类型。对于$Threshold_{DC}$和$Threshold_{CD}$的设置，我们首先使用3σ 标准分别去除$Closeness_{DC}$和$Closeness_{CD}$集合中的异常值。我们这里的异常值定义为：比集合标准差σ 高三倍或者低三倍的紧密度值。然后通过min-max标准化我们将剩余的代码依赖紧密度归一化到[0,1]区间。之

前被过滤掉的高异常值此时设置为1，类似的之前被过滤的低异常值被设置为0。对于$Threshold_{DC}$和$Threshold_{DC}$ 的值的选取，我们根据 4.2.2中的度量指标，在[0,1] 区间选取一组能让所有实验系统在不同信息检索模型下综合表现最好的阈值。本文中设置$Threshold_{DC}$ 为0.4，$Threshold_{CD}$为0.8。

对于代码依赖紧密度，经过min-max标准化处理之后的值，仅用于设定阈值生成代码依赖域。对于算法 2我们还是用原始的代码依赖紧密度值。对于LSI方法中的K值，我们发现当k为90时iTrust和Maven的 Precision/Recall表现最好，而对于Pig和 Infinispan，k值取200 时 Precision/Recall表现最好，从表 4.2中可以看出 iTrust和 Maven数据集中需求的个数比较接近（94和138），Pig和 Infinispan的需求比较接近（236和237）我们认为这是这两组系统在LSI模型下最优K值出现差异的原因。

4.2.4　实验目标与研究问题

我们的实验目的是分析结合我们的方法能否提高生成追踪线索列表的精度，为了达到这一目的，我们需要回答如下研究问题：
RQ: 在需求到代码的可追踪性生成场景下我们的方法能否优于基线方法？
为了回答这个问题，我们选择了以下三个基线方法：（1）只利用信息检索技术的纯信息检索方法（简称为IR-ONLY）；（2）基于信息检索方法，引入代码紧密度分析的方法（简称为TRICE [16]）；（3）基于信息检索方法，考虑代码结构信息和用户反馈信息的先驱方法（简称为UD-CSTI [17]）。我们将我们的方法命名为CLUSTER（CLoseness-and-USer-feedback-based Traceability Recovery），在对比CLUSTER和三个基线方法时，我们依次使用三种不同的主流模型（VSM、LSI和JS）。同时本方法的另一个目标是尽量减少用户参与，即使用尽量少的用户反馈。所以，对于基线方法UD-CSTI，用户需要判断所有的追踪线索（我们方法中的用户判断环节通过软件可追踪数据集来模拟，即我们假设用户判断全部都是正确的）。而实验中我们的方法里需要用户判断的追踪线索不超过全部追踪线索的3.5%。即对于给定需求，iTrust中需要判断四个类，Maven中需要判断三个类，Pig和Infinispan中需要判断八个类与给定需求的相关性。基于以上设置，通过比较UD-CSTI和CLUSTER来判断在我们的方法中是否少量的用户反馈就能起到明显的效果。除了 4.2.2中提到的指标，我们还引入了统计显著性测试来判断CLUSTER是否显著比基线方法好。通过调研 [16, 57]中所设计的显著性测试。对于候选追踪列表，我们选择在每个有效追踪线索位置的F-measure 作为我们测试的单独依赖变量。选择F-measure的原因是我们想分析相比于基线方

法，CLUTSRE方法能否同时提升查全率和准确率。F-measure计算公式如下：

$$F = \frac{2}{\frac{1}{R} + \frac{1}{P}} \tag{4.5}$$

参数解释：P表示准确率，R表示查全率，F是P和R的调和平均。从公式中可以看出准确率和查全率越大，F-measure的值越高。由于F-measure 在两个不同系统之间是成对存在的，因此我们使用Wilcoxon rank sum测试 [58]验证以下零假设：

H_0:CLUSTER和基线方法性能没有区别。

我们采用$p-value$显著性差异水平0.05作为衡量检验结果的标准。

4.2.5　实验结果与分析

根据 4.2.2提出的度量指标，表 4.3展示了4个实验系统的实验结果。我们比较了CLUSTER和基线方法在这四个实验系统上的表现（AP，MAP与显著性检验的p-value值）。在36个实验结果对比中，有30个实验结果CLUSTER 的F-measure值明显优于基线方法（p-value<0.05并且AP值大于基线方法）。这表明在绝大多数情况下，相比基线方法，CLUSTER 提高了候选追踪列表的准确率和查全率。此外，对于实验系统iTrust 和Maven，CLUSTER方法的AP和MAP几乎全部优于基线方法。只有在Maven-VSM和Maven-LSI 上CLUSTER 方法的MAP值略逊于TRICE。然而在Pig系统上CLUSTER方法只有在VSM模型下AP值优于UD-CSTI，其他情况下效果均不如基线方法UD-CSTI。但是从表 4.3 中可以看出两者差距比较小（差异小于0.5）。并且如前文所述CLUSTER 方法只需要用户判断3.5%的追踪线索，而UD-CSTI方法需要用户验证所有的跟踪线索。图4.6展示并对比了12种实验组合下的precision-recall曲线。

观察表 4.3和图 4.6，我们会发现TRICE方法的性能高度依赖通过纯信息检索方法生成候选追踪列表的质量。即TRICE方法依赖需求文本和代码文本的语料质量。在文本语料质量比较好的iTrust系统上，TRICE和UD-CSTI效果差异不大。但是在信息检索效果的AP和MAP 都比较低的Pig和Maven系统，TRICE方法明显不如UD-CSTI。在Pig-VSM和Pig-LSI，TRICE方法的性能甚至不如纯信息检索方法。没有用户反馈，TRICE方法只能把与给定需求相似度最大的类作为紧密度分析的输入，给与这个类代码依赖紧密度比较大的类对应追踪线索奖励。这种方法比较保守，然而也使得无法进一步提高基于信息检索方法的性能。例如在前面用例中，对于CLUSTER方法，用户需要判断两个域的代表类与需求相关性。如果相关会通过两个类向外扩散给与其依赖关系紧密的类

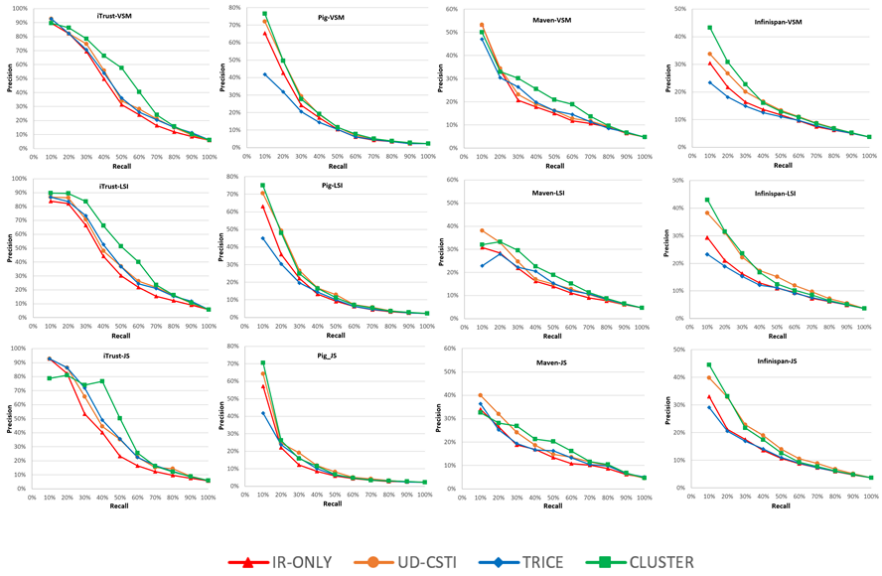

图 4.6: precision-recall曲线

对应候选线索奖励。此外，与给定需求相似度最大的类对应的追踪线索如果是无效的，在TRICE方法中由于没有用户反馈，按照TRICE方法会给与这个类紧密度比较大的类对应的追踪线索奖励。即此时这个错误会被放大最终使得被“优化之后的候选列表”准确率和查全率更低。在CLUSTER和TRICE方法中由于用户反馈的存在就避免了这种情况。实际上，我们观察到在12个实验中CLUSTER和UD-CSTI的效果均明显优于IR-ONLY，即便是在语料质量比较好的iTrust系统，情况也是如此。这表明在建立需求到代码的可追踪性时，用户反馈是一项重要的资源。

然而，使用用户反馈意味着用户需要付出额外的努力。正如前文提到的UD-CSTI 方法为了达到其最优效果需要用户判断所有的追踪线索，这是不实际的。而我们的方法CLUSTER只需要用户判断3.5%的追踪线索（即对于给定需求，iTrust中需要判断四个类，Maven中需要判断三个类，Pig中需要判断8个类，Infinispan中需要判断8个类与给定需求的相关性。）就能达到与UD-CSTI类似的效果甚至优于UD-CSTI。这在基本不影响候选追踪线索列表效果的前提下大大减少了用户需要付出的努力。为了进一步比较CLUSTER方法的成本效益（成本是指用户做出判断花费的精力，效益是用户判断后候选追踪列表准确率和查全率的提高），我们比较不同查全率对应的精度和遇到的无关追踪线索数量（如表4.4）。

从表中可以看出当查全率小于20%时，CLUSTER方法的准确率要比IR-

表 4.3: 实验系统的评价指标及Wilcoxon秩和检验结果

		VSM			LSI			JS		
		AP	MAP	p-value	AP	MAP	p-value	AP	MAP	p-value
iTrust	IR-Only	42.55	56.55	<0.01	41.21	55.73	<0.01	38.28	55.99	<0.01
	UD-CSTI	45.75	59.08	<0.01	45.42	59.35	<0.01	43.26	62.99	0.10
	TRICE	45.80	59.05	<0.01	45.18	58.76	<0.01	45.29	61.25	0.16
	CLUSTER	52.10	63.28	-	51.60	63.21	-	46.56	63.29	-
Maven	IR-Only	20.66	38.59	<0.01	17.21	42.37	<0.01	19.45	40.70	<0.01
	UD-CSTI	21.68	39.42	<0.01	18.52	43.28	<0.01	22.12	42.10	<0.01
	TRICE	21.68	41.58	<0.01	16.85	46.17	<0.01	19.30	41.69	<0.01
	CLUSTER	25.44	41.38	-	21.11	45.47	-	22.32	43.57	-
Pig	IR-Only	21.98	42.36	<0.01	19.88	42.25	<0.01	15.61	35.56	0.02
	UD-CSTI	24.19	43.49	0.49	22.40	43.67	0.68	18.90	37.82	0.27
	TRICE	16.88	41.91	<0.01	15.85	40.72	<0.01	13.83	36.79	0.18
	CLUSTER	24.64	43.32	-	22.30	43.19	-	18.88	37.57	-
Infinispan	IR-Only	14.47	22.87	<0.01	14.15	22.98	<0.01	14.98	22.59	<0.01
	UD-CSTI	16.59	24.14	<0.01	17.95	24.76	<0.01	19.57	24.56	<0.01
	TRICE	12.79	20.92	<0.01	12.96	21.84	<0.01	13.70	21.28	<0.01
	CLUSTER	18.47	23.82	-	18.61	23.91	-	19.86	23.24	-

ONLY低，这是因为CLUSTER方法是先让用户判断一些具有代表性的候选追踪线索，这些追踪线索未必是最可能有效的。相对于IR-ONLY算法本文方法刚开始会遇到更多的无效追踪线索。在查全率大于20%时，CLUSTER方法前期的用户投入开始发挥作用，如表 4.4所示在查全率为60%时，CLUSTER方法的准确度优于IR-ONLY，在iTrust-LSI下与IR-ONLY相比，CLUSTER方法的准确率提升了21.64%。从表中可以得出用户在建立153 个正确追踪线索的过程中，CLUSTER 方法检索出的无效追踪线索比UD-CSTI 少349 个。当查全率在50% 和80% 之间时，CLUSTER 方法相对于IR-ONLY 的优势尤为明显。 根据实验结果，我们还有一点额外的观察。（1）对于同一个实验系统在不同的IR模型下，CLUSTER方法的表现差异很大。这是因为CLUSTER 方法主要是通过给追踪线索奖励来实现重排序，如前所述（公式）奖励值依赖于IR值，不同的IR模型对于IR值的计算有不同的方式。（2）我们尝试过要求用户判断所有代码域与给定需求相关性，该做法对pig和Maven两个实验系统的性能提升不大。这可能是因为这两个系统的需求粒度太小。表 4.2也验证了这一点，iTrust一个需求平均由8个类完成，而Maven和Pig的一个需求分别平均由4，5个类完成。在将来的工作中，我们计划用文本聚类的方法 [59]增大需求粒度。

表 4.4: 不同查全率下的精度对比

		Recall(20%)		Recall(40%)		Recall(60%)		Recall(80%)	
		Precision	FP	Precision	FP	Precision	FP	Precision	FP
iTrust	VSM	-46.59%	+81	-0.24%	+1	+16.81%	-261	+3.93%	-428
	LSI	-46.34%	+80	+6.40%	-29	+21.64%	-349	+3.93%	-403
	JS	-46.59%	+81	+7.73%	-41	+11.48%	-381	+2.52%	-441
Maven	VSM	-6.79%	+24	+8.57%	-113	+6.78%	-287	+0.52%	-74
	LSI	-0.51%	+2	+7.12%	-117	+4.51%	-242	+1.01%	-186
	JS	+1.94%	-8	+8.94%	-127	+7.39%	-351	+1.68%	-231
Pig	VSM	-30.50%	+426	-4.72%	+319	+0.85%	-434	+0.38%	-856
	LSI	-24.04%	+410	-1.53%	+142	+0.01%	-4	+0.21%	-511
	JS	-11.70%	+367	+0.29%	-55	+0.09%	-93	+0.23%	-777
Infinispan	VSM	-5.12%	+309	+0.85%	-185	+1.06%	-677	+0.66%	-1369
	LSI	-4.85%	+309	+1.76%	-401	+0.43%	-301	+0.32%	-697
	JS	-4.32%	+264	+1.42%	-301	+0.34%	-273	+0.27%	-634

4.3　本章小结

在本章中，我们通过对开源软件在issue-tracking工具上的行为信息进行分析整理，组织了其需求到代码的追踪关系。此外，我们通过运行开源系统自带的用于验证系统功能的测试用例得到了我们方法所需的代码依赖。最终，我们用一个被领域内广泛用于可追踪方法验证的高质量数据集和三个被广泛应用于日常实践的开源系统验证了我们方法的有效性和实用性。

第五章　软件可追踪生成工具的设计与实现

在本文第四章 4.2节，我们通过实验证明了本文方法的有效性和实用性。为了将该方法应用到日常实践中，本章结合用户使用习惯实现了软件可追踪生成工具。本章 5.1节介绍了该工具的使用场景， 5.2节介绍了该工具的模块组成，以及各个模块的功能，最后在 5.3节结合真实案例详细讨论了该工具的使用流程。

在本章中，针对结合代码依赖和用户反馈的软件可追踪生成问题，我们设计了面向用户的工具。该工具集成了我们结合代码依赖紧密度分析和用户反馈的软件可追踪生成方法。我们介绍了软件可追踪生成工具的应用场景，并详细阐述了工具的设计与实现。此外，我们结合一个案例说明工具的使用流程。

5.1　软件可追踪生成工具应用场景

软件测试是软件上线之前需要完成的一个重要步骤。如果测试人员发现软件没有正确完成某个需求，此时，测试人员会将与之相关的异常信息进行整理并向软件负责人反馈。由于缺乏需求到代码的追踪矩阵，项目负责人很难快速获取与给定需求存在相关性的所有代码元素。接下来，我们将介绍需求到代码可追踪生成工具如何支持这一需求可追踪查询的过程。

软件可追踪生成工具结合代码依赖紧密度分析和用户反馈两部分信息，对通过信息检索方法形成的候选追踪线索列表进行优化调整，图 5.1 中展示了软件可追踪生成工具在软件可追踪建立场景中的作用效果。 接下来我们将简要介绍软件可追踪生成工具给用户（项目负责人）带来的益处：

1. 该工具类似一个搜索引擎（如图 5.3左侧窗体所示）根据输入需求返回与需求存在相关性的代码元素列表（如图 5.5右下角窗体），工具界面精简，操作简单。

2. 在用户对少量有代表性追踪线索相关性进行验证的阶段（如图 5.3中间窗体所示），工具提供了大量的额外信息（如图 5.4所示）辅助用户完成判断。此举能够有效减少用户付出的时间成本并有效提升用户判断的正确率。

利用软件可追踪生成工具，用户只需判断少量的候选追踪线索相关性。之前 4.2实验表明，当有3.5%候选追踪线索相关性交由用户验证时，最终得到的

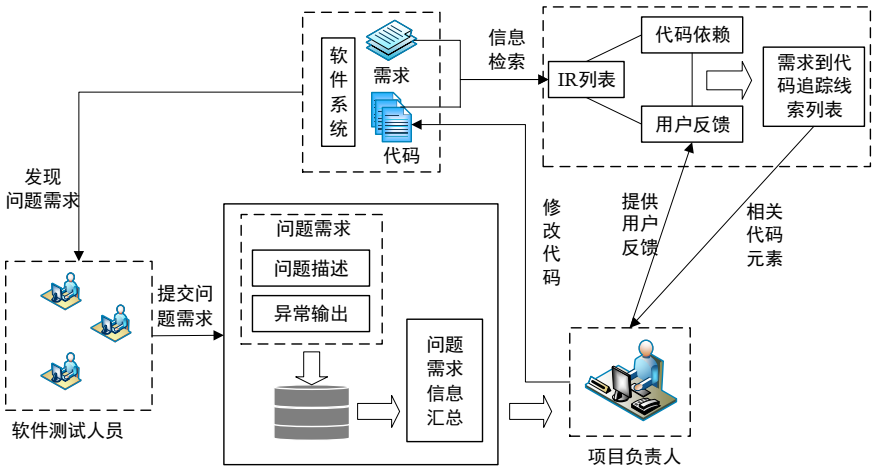

图 5.1: 软件可追踪生成工具的应用场景

候选追踪列表的精度即可明显优于基线方法。在用户验证阶段，工具通过提供代码依赖图拓扑图、特殊单词高亮等多种手段（详见 5.3）保证用户能够高效的做出给出正确的判断结果。综上所述，通过该工具用户只需很少的参与即可得到相对高精度的需求到代码的追踪列表。

5.2　软件可追踪生成工具体系结构

该工具的核心是需求到代码可追踪生成部分，图 5.2 展示该部分的体系结构，其中包括数据准备、信息检索、代码依赖捕获、用户交互以及候选线索列表优化五个模块。在本小节中，我们将对以上五个模块的设计与实现进行说明。

1. 数据准备模块：软件可追踪生成辅助工具的输入是需求和代码，这里的代码数据包含针对项目的测试集。输出是需求到代码的追踪线索列表。数据准备模块主要是提供数据导入接口。数据导入模块是其它所有模块的基础，在代码依赖捕获模块，工具会通过数据准备模块中导入的测试代码集合得到代码依赖；在用户交互模块，工具会展示需求和代码的文本内容；在检索模块，工具对导入的需求和代码做文本相似度计算并生成候选追踪列表。

2. 信息检索模块：该模块是对数据准备模块中导入的需求和代码数据进行处理。对需求文本进行文本预处理，包括移除停用词、词形还原和词干提取等操作。对于代码文本，首先需要根据命名规则进行分词，然后与需求文

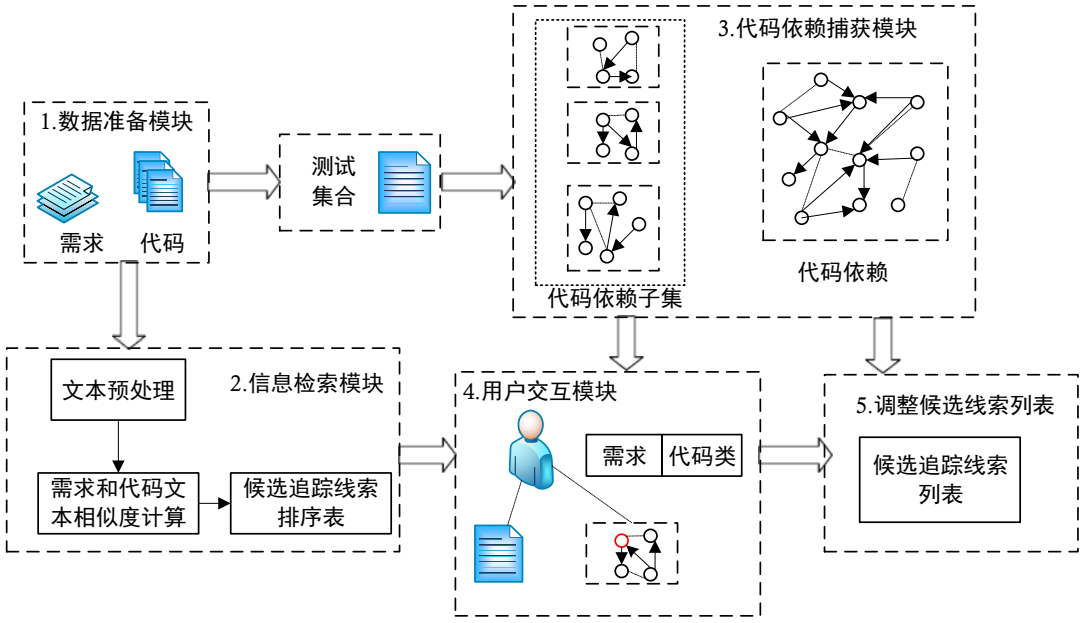

图 5.2: 软件可追踪生成工具的体系结构图

本进行同样的文本预处理；接下来基于信息检索技术，计算需求文本和代码文本集合之间的文本相似度。我们实现了VSM、LSI与JS这三个被广泛使用的信息检索模型，以VSM（向量空间模型）为例，将需求文本和代码文本用高维向量q、r表示，向量中每个维度w对应一个单词的权重，权重w可用TF-IDF公式计算，对于q、r两个高维向量，两者组成一条候选追踪线索，可利用余弦距离计算向量之间的余弦相似度，我们将其定义为这条追踪线索的相似度值。然后我们根据追踪线索的相似度值，对候选追踪线索按照相似度值从大到小的顺序排序，形成候选追踪列表。

3. 代码依赖捕获模块：该模块主要负责捕获代码依赖关系，这里的代码依赖包括直接依赖（类之间的调用、使用和继承）和数据依赖（类之间的数据共享）。虽然像TPTP（The Eclipse Test & Performance Tools Platform）和JPF（Java Plug-in Framework）等工具都能在软件系统运行过程中捕获系统中存在的方法之间的调用依赖。但是根据我们的调研，并没有一个现成的动态分析工具可以捕获方法之间的数据依赖。因此我们基于JVM提供的接口，注册函数调用数据访问等事件，并在其回调函数中存储方法进入、退出和数据访问记录，最终对该数据进一步处理得到代码依赖关系。根据项目使用的构建工具（Maven、Ant）我们采用不同的方式运行测试集，并在此过程中通过不同的方式插桩我们的代码依赖捕获工具，得到各

测试用例对应的代码依赖子集。然后我们对代码依赖子集进行合并得到代码依赖。

4. 用户交互模块：该模块首先会对代码依赖模块捕获的代码依赖进行紧密度分析，并通过代码依赖紧密度阈值生成代码域。对于每个需求，根据各域内类与需求相似度最大值，对用户为判断域按照相似度值自大到小的顺序重排序。对于排名第一的代码域，取其域内与需求相似度值最大的类交由用户判断与需求的相关性，如图 5.3右侧窗体所示。一方面在对话框左栏用户可以选择相关、不相关、跳过该判断和结束整个判断过程，右栏是当前需求的文本内容。另一方面用户可以点击帮助查看与该类代码依赖关系紧密的其它类，以及它们之间的拓扑结构，如图 5.4所示，通过点击类，用户可以查看该类的文本内容，与当前类相关的需求，出现在需求中的单词等多种辅助用户判断信息。工具会根据用户的判断结果调整候选追踪列表，继续向用户推荐下一个需要用户判断的追踪线索或者生成最终结果。

5. 候选追踪列表优化模块：该模块会根据对代码依赖紧密度分析得到的代码域，和在用户交互模块中得到的用户反馈信息对候选追踪列表进行调整。当用户判断给定的候选追踪线索具有相关性时，对于该候选线索中的类，工具会通过不同的方式提升与该类在一个域中的其它类，和在域外但是和该类存在之间或数据依赖的类对应候选追踪线索的相似度值。该过程会迭代多次直到用户交互模块中，用户选择结束判断为止。此时工具会将候选追踪列表数据持久化到磁盘中，方便软件利益相关者使用。

5.3　软件可追踪生成工具案例介绍

在本小节，我们将结合一个案例说明工具的使用方式。 对于一个即将上线的医疗管理软件iTrust，测试组发现该系统的需求UC18没有通过测试，测试人员会整理问题相关信息以及可能的异常输出信息等向软件负责人反馈。软件负责人收到该反馈信息后需尽快修复造成该问题的代码段以免造成项目延期。然而，因为项目组并没有维护需求到代码的追踪矩阵。这使得很难快速的找出与需求UC18 具有相关性的代码元素。此时，我们的工具能帮助用户（项目负责人）快速解决这个难题。

首先用户需要导入需求和代码等数据。接下来进入图 5.3左侧所示界面，

我们的工具类似一个搜索引擎，向用户返回与指定需求相关的代码元素信息。点击搜索后会进入用户交互界面（图 5.3 中间窗体所示界面），工具选择少量有代表性的追踪线索交由用户验证。本例HospitalsDAO是其所在代码域与需

图 5.3: 需求可追踪查询界面

求UC18相似度最大的类。因此，工具用HospitalsDAO与UC18的相关性来代表代码域中其它类与UC18的相关性，图 5.3左侧窗体为需求UC18的文本信息，右侧窗体为代码元素HospitlsDAO的文本信息。为了使用户能够高效的给出正确的验证结果。工具提供大量信息辅助用户完成验证，用户只需通过 帮助按钮即可获得大量辅助信息（如图 5.4所示）。

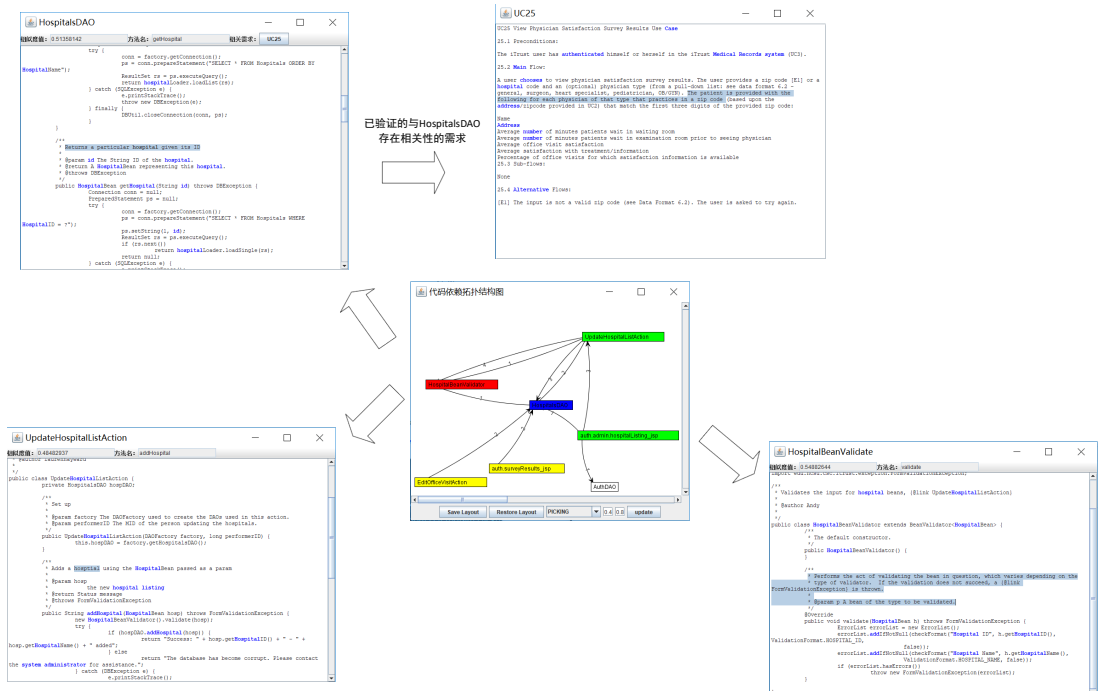

图 5.4: 辅助用户验证信息界面

图 5.4最中间窗体为代码依赖拓扑图，蓝色矩形代表的代码元素为用户需要判断的类，绿色代码元素为与当前类位于同一个代码域中的类。黄色矩形

代表与当前类存在代码依赖关系的域外类。红色矩形代表与当前需求相似度最大的类。白色为与域内其它类存在代码依赖的域外类。用户通过点击这些矩形框可以很方便的查看这些类与需求的相似度值，当前查看的方法的方法名等信息（通过点击相应矩形框，生成图 5.4 外侧窗体）。在需求中出现过的单词会在类文本中高亮显示。工具也会向用户提供已经用户验证的与当前类存在相关性的需求信息（如图 5.4左上角的窗体所示，HospitalsDAO和UC25具有相关性），同时用户也能很方便的查看该需求文本内容（如图 5.4 右上角窗体所示）。用户通过阅读UC18的文本内容知该需求描述了管理员维护医院列表信息的功能。 一方面，从图 5.4查看与HospitalsDAO 位于同一代码域的

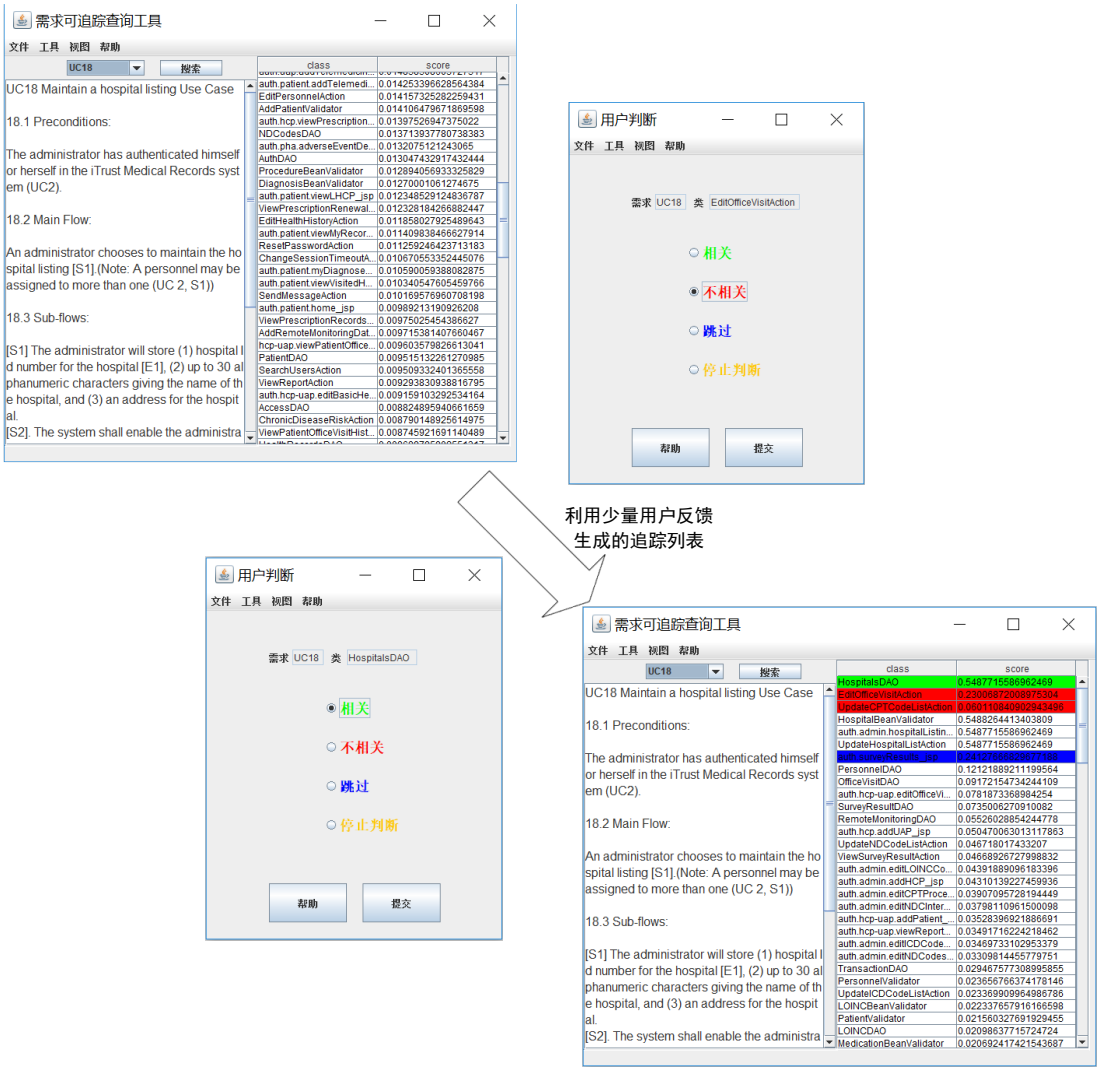

图 5.5: 需求查询结果显示界面

UpdateHospitalListAction 和与UC18 文本相似度最大的HospitalBeanValidator 的

文本内容，里面出现大量 *hospital*、*administrator* 等在UC18 中出现的单词。因此，用户很容易得出UpdateHospitalListAction 大概率与UC18具有相关。另一方面，根据代码命名规则，HospitalsDAO是负责与数据库交互。根据该类与周边类的代码依赖关系，用户很容易得出该类可能负责添加、查询医院信息等操作。与HospitalsDAO 存在相关性UC25 所描述的功能为，根据用户提供医院编码和医生类型等数据，返回患者对医生的满意度信息。这验证了用户对HospitalDAO负责与数据库交互读取与与医院相关数据表的猜测。综合与HospitalsDAO 存在代码依赖的类和与其存在相关性的需求信息，用户很容易得出该类与UC18 存在相关性。

经过少量类似的用户验证，最终，工具会向用户提供与需求UC18具有相关性的代码元素排序表。图 5.5 展示了从初始追踪线索列表经少量用户参与到生成优化后的候选追踪列表的过程。用户验证过的追踪线索排在列表最前面，其中绿色和红色背景的候选追踪线索分别代表已经用户验证与UC18 相关和不相关的代码元素，黄色代表工具推荐给用户判断但是用户选择跳过即未对进行判断与UC18相关性的代码元素（这和图 5.3中间窗台各选项颜色存在对应关系）。对于用户未验证的追踪线索，在排序表中越靠前的类，与需求存在相关性的概率越大。通常情况下，软件负责人通过review排在列表顶部的几个类，即可完成对问题代码段的修复任务。

5.4　本章小结

在本章中，我们介绍了需求到代码可追踪生成工具的设计实现与应用场景，并辅以一个具体案例解释了工具的使用流程。需求到代码可追踪生成工具能减少用户参与并且提高追踪列表的准确率和完全率。该工具可以帮助用户快速得到与指定需求存在相关性的代码元素。

第六章　总结与展望

6.1　工作总结

在软件开发过程中，随着软件规模的增加和开发人员的流动，使得软件利益相关者很难建立和维护需求到代码的可追踪性。针对这种情况，我们提出了一套结合用户反馈和代码依赖的需求到代码可追踪生成方法并取得了一定效果。本文工作主要包括以下几点：

1. 我们提出了一种结合用户反馈和代码依赖紧密度分析的需求到代码的软件可追踪生成方法。通过代码依赖紧密度分析，设置紧密度阈值将功能紧密的类绑定到同一个代码域中。在代码域中选择一个有代表性的交由用户判断域给定需求的相关性。然后根据用户判断结果（用户反馈）调整与该类关系紧密的域内其它类以及域外相关类对应候选线索的相似度值。从而在减少用户参与的情况下，提高追踪线索列表的准确率和完全率。

2. 我们在四个研究案例下设计实验，验证了上述方法的有效性和实用性。并且，我们通过对开源软件在issue-tracking工具上的行为信息进行分析整理，组织了其需求到代码的追踪关系。此外，我们通过运行开源系统自带的用于验证系统功能的测试用例得到了我们方法所需的代码依赖。

3. 我们设计并实现了一个需求到代码的可追踪生成工具，并集成了结合代码依赖紧密度分析和用户反馈的需求到代码可追踪生成方法，以辅助用户得到高质量的追踪数据。

6.2　研究展望

在未来，我们的工作主要有以下三个方向可以进行探索：

1. 考虑需求之间的关系：我们的方法引入了代码元素之间的关系（代码依赖）并取得一定效果，实际上，需求之间也并非完成独立，不同的需求可能存在依赖、从属等关系，我们可以充分挖掘和使用需求之间的关系以进一步提升生成候选追踪列表的精度。

2. 验证基于代码依赖和用户反馈的软件可追踪生成方法对维护人员的有益性：由于我们已经设计并实现了一个用户友好的软件可追踪生成工具，并

集成了基于代码依赖和用户反馈的软件可追踪生成方法。因此，我们可以设置对照实验，比较在使用或不使用该工具时，软件利益相关者完成同一软件活动所耗费的时间以及正确性等，以进一步说明方法的优越性。

3. 为开源软件建立更高质量的需求到代码可追踪数据集：当前方法通过对开源软件在issue-tracking工具上的行为信息进行分析整理，得到与软件行为对应的代码提交，进而得到其可追踪数据集。实际上，托管平台和issue-tracking工具还包含大量的其它信息，例如，代码提交者，软件行为变更请求发起者，版本信息，与代码变更相关的讨论信息等。我们可以充分利用这些信息对当前获取的可追踪数据集进行进一步优化。

参考文献

[1] Coest: Center of excellence for software traceability. http://www.CoEST.org/.

[2] Xiaofan Chen and John C. Grundy. Improving automated documentation to code traceability by combining retrieval techniques. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011*, pages 223–232, 2011.

[3] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983, 2002.

[4] G Antoniol, G Casazza, and A Cimitile. Traceability recovery by modeling programmer behavior. In *Working Conference on Reverse Engineering*, page 240, 2000.

[5] Patrick Mäder Patrick Rempel. Preventing defects: The impact of requirements traceability completeness on software quality. *IEEE Trans. Software Eng.*, 43(8):777–797, 2017.

[6] Jane Cleland-Huang. Are requirements alive and kicking? *IEEE Software*, 30(3):13–15, 2013.

[7] Patrick Mäder and Alexander Egyed. Assessing the effect of requirements traceability for software maintenance. In *28th IEEE International Conference on Software Maintenance, ICSM 2012, Trento, Italy, September 23-28, 2012*, pages 171–180, 2012.

[8] Jane Cleland-Huang, Orlena Gotel, Jane Huffman Hayes, Patrick Mäder, and Andrea Zisman. Software traceability: trends and future directions. In *Proceedings of the on Future of Software Engineering, FOSE 2014, Hyderabad, India, May 31 - June 7, 2014*, pages 55–69, 2014.

[9] Jane Huffman Hayes Michael Vierhauser Giuliano Antoniol, Jane Cleland-Huang. Grand challenges of traceability: The next ten years. *CoRR*, abs/1710.03129, 2017.

[10] Balasubramaniam Ramesh and Matthias Jarke. Toward reference models for requirements traceability. *IEEE Transactions on Software Engineering*, 27(1):58–93, 2001.

[11] Orlena CZ Gotel and Anthony CW Finkelstein. The study of methods for language model based positive and negative relevance feedback in information retrieval. In *Proceedings of the First International Conference on Requirements Engineering*, pages 94–101. IEEE, 1994.

[12] Jane Huffman Hayes, Alex Dekhtyar, and Senthil Karthikeyan Sundaram. Advancing candidate link generation for requirements tracing: The study of methods. *IEEE Transactions on Software Engineering*, 32(1):4–19, 2006.

[13] Andrian Marcus and Jonathan I Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the 25th International Conference on Software Engineering*, pages 125–135. IEEE, 2003.

[14] Jane Cleland-Huang, Raffaella Settimi, Chuan Duan, and Xuchang Zou. Utilizing supporting evidence to improve dynamic requirements traceability. In *Proceedings of the 13th IEEE International Conference on Requirements Engineering*, pages 135–144. IEEE, 2005.

[15] Collin McMillan, Denys Poshyvanyk, and Meghan Revelle. Combining textual and structural analysis of software artifacts for traceability link recovery. In *ICSE Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE)*, pages 41–48. IEEE, 2009.

[16] Hao Hu Patrick Rempel Jian Lu Alexander Egyed Patrick Mäder Hongyu Kuang, Jia Nie. Analyzing closeness of code dependencies for improving ir-based traceability recovery. In *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017*, pages 68–78, 2017.

[17] Annibale Panichella, Collin McMillan, Evan Moritz, Davide Palmieri, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. When and how using structural information to improve ir-based traceability recovery. In *17th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 199–208. IEEE, 2013.

[18] Jane Cleland-Huang, Adam Czauderna, Marek Gibiec, and John Emenecker. A machine learning approach for tracing regulatory codes to product specific requirements. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 155–164. ACM, 2010.

[19] Jin Guo, Jinghui Cheng, and Jane Cleland-Huang. Semantically enhanced software traceability using deep learning techniques. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 3–14, 2017.

[20] Paul N. Otto Aaron K. Massey and Annie I. Antón. Aligning requirements with HIPAA in the itrust system. In *16th IEEE International Requirements Engineering Conference, RE 2008, 8-12 September 2008, Barcelona, Catalunya, Spain*, pages 335–336, 2008.

[21] Lefteris Angelis Laura Phillips Yvonne Dittrich Tony Gorschek Håkan Grahn Claes Wohlin, Aybüke Aurum, Graham Low Per Rovegard Piotr Tomaszewski Christine Van Toorn Kennet Henningsson, Simon Kågström, and Jeff Winter. The success factors powering industry-academia collaboration. *IEEE Software*, 29(2):67–73, 2012.

[22] Jane Cleland-Huang. Toward meaningful industrial-academic partnerships. *IEEE Software*, 32(1):18–21, 2015.

[23] Benedikt Burgstaller and Alexander Egyed. Understanding where requirements are implemented. In *IEEE International Conference on Software Maintenance (ICSM)*, pages 1–5. IEEE, 2010.

[24] Aharon Abadi, Mordechai Nisenson, and Yahalomit Simionovici. A traceability technique for specifications. In *The 16th IEEE International Conference on Program Comprehension*, pages 103–112. IEEE, 2008.

[25] Scott Deerwester, Susan T Dumais, George W Furnas, Thomas K Landauer, and Richard Harshman. Indexing by latent semantic analysis. *Journal of the American society for information science*, 41(6):391, 1990.

[26] Susan T Dumais. Improving the retrieval of information from external sources. *Behavior Research Methods, Instruments, & Computers*, 23(2):229–236, 1991.

[27] Gerard Salton and Michael J McGill. Introduction to modern information retrieval. 1986.

[28] Thomas K. Landauer George W. Furnas Richard A. Harshman Scott C. Deerwester, Susan T. Dumais. Indexing by latent semantic analysis. *JASIS*, 41(6):391–407, 1990.

[29] Michael Gutman. Asymptotically optimal classification for multiple tests with empirically observed statistics. *IEEE Trans. Information Theory*, 35(2):401–408, 1989.

[30] Hongyu Kuang, Patrick Mäder, Hao Hu, Achraf Ghabi, LiGuo Huang, Jian Lü, and Alexander Egyed. Can method data dependencies support the assessment of traceability between requirements and source code? *Journal of Software: Evolution and Process*, 27(11):838–866, 2015.

[31] Paola Sgueglia Andrea De Lucia, Rocco Oliveto. Incremental approach and user feedbacks: a silver bullet for traceability recovery. In *22nd IEEE International Conference on Software Maintenance (ICSM2006), 24-27 September 2006, Philadelphia, Pennsylvania, USA*, pages 299–309, 2006.

[32] Andy Zaidman Annibale Panichella, Andrea De Lucia. Adaptive user feedback for ir-based traceability recovery. In *8th IEEE/ACM International Symposium on Software and Systems Traceability, SST 2015, Florence, Italy, May 17, 2015*, pages 15–21, 2015.

[33] Wenjing Zhang and Junyi Wang. The study of methods for language model based positive and negative relevance feedback in information retrieval. In *Fourth International Symposium on Information Science and Engineering*, pages 39–43, 2012.

[34] Berthier A. Ribeiro-Neto Ricardo A. Baeza-Yates. *Modern Information Retrieval*. ACM Press / Addison-Wesley, 1999.

[35] Yin Li Ye Yang Qing Wang Lingjun Kong, Juan Li. A requirement traceability refinement method based on relevance feedback. In *Proceedings of the 21st International Conference on Software Engineering & Knowledge Engineering (SEKE'2009), Boston, Massachusetts, USA, July 1-3, 2009*, pages 37–42, 2009.

[36] Christopher D Manning and Hinrich Raghavan, Prabhakar. Introduction to information retrieval. *Journal of the American Society for Information Science & Technology*, 43(3):824–825, 2008.

[37] Christopher Morrell Dawn J. Lawrie, David W. Binkley. Normalizing source code vocabulary. In *17th Working Conference on Reverse Engineering, WCRE 2010, 13-16 October 2010, Beverly, MA, USA*, pages 3–12, 2010.

[38] David W. Binkley Dawn J. Lawrie. Expanding identifiers to normalize source code vocabulary. In *IEEE 27th International Conference on Software Maintenance, ICSM 2011, Williamsburg, VA, USA, September 25-30, 2011*, pages 113–122, 2011.

[39] Nan Niu Li Da Xu Jing-Ru C. Cheng Zhendong Niu Wentao Wang, Arushi Gupta. Automatically tracing dependability requirements via term-based relevance feedback. *IEEE Trans. Industrial Informatics*, 14(1):342–349, 2018.

[40] Giuliano Antoniol Massimiliano Di Penta, Sara Gradara. Traceability recovery in RAD software systems. In *10th International Workshop on Program Comprehension (IWPC 2002), 27-29 June 2002, Paris, France*, pages 207–216, 2002.

[41] Alex Dekhtyar Olly Gotel Jane Huffman Hayes Ed Keenan Greg Leach Jane Cleland-Huang, Adam Czauderna, Yonghee Shin Andrea Zisman Giuliano Antoniol Brian Berenbach Jonathan I. Maletic, Denys Poshyvanyk, Alexander Egyed, and Patrick Mäder. Grand challenges, benchmarks, and tracelab: developing infrastructure for the software traceability research community. pages 17–23, 2011.

[42] Cédric Jeanneret Martin Glinz Eya Ben Charrada, David Caspar. Towards a benchmark for traceability. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution, EVOL/IWPSE 2011, Szeged, Hungary, September 5-6, 2011.*, pages 21–30, 2011.

[43] John Grundy Robert Amor Xiaofan Chen, John G. Hosking. Development of robust traceability benchmarks. In *22nd Australian Conference on Software Engineering (ASWEC 2013), 4-7 June 2013, Melbourne, Victoria, Australia*, pages 145–154, 2013.

[44] Patrick Mäder Michael Rath, Patrick Rempel. The ilmseven dataset. In *25th IEEE International Requirements Engineering Conference, RE 2017, Lisbon, Portugal, September 4-8, 2017*, pages 516–519, 2017.

[45] Qing Wang Shoubin Li Barry W. Boehm Lin Shi, Celia Chen. Understanding feature requests by leveraging fuzzy method and linguistic analysis. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, pages 440–450, 2017.

[46] itrust system:. http://agile.csc.ncsu.edu/iTrust/wiki/doku.php.

[47] Maven system:. http://maven.apache.org/.

[48] Pig system:. http://pig.apache.org/.

[49] Infinispan system:. http://infinispan.org/.

[50] github:. https://github.com/.

[51] Klaus Pohl. *Requirements Engineering - Fundamentals, Principles, and Techniques*. Springer, 2010.

[52] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *International Conference on Software Maintenance*, page 23, 2003.

[53] Petra Heck and Andy Zaidman. An analysis of requirements evolution in open source projects: recommendations for issue trackers. In *13th International Workshop on Principles of Software Evolution, IWPSE 2013, Proceedings, August 19-20, 2013, Saint Petersburg, Russia*, pages 43–52, 2013.

[54] ant:. https://ant.apache.org/.

[55] surefire:. https://maven.apache.org/surefire/maven-surefire-plugin/.

[56] junit:. https://junit.org/junit5/.

[57] Nasir Ali, Zohreh Sharafi, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. An empirical study on the importance of source code entities for requirements traceability. *Empirical Software Engineering*, 20(2):442–478, 2015.

[58] David J. Groggel. Practical nonparametric statistics. *Technometrics*, 42(3):317–318, 2000.

[59] Adelina Ciurumelea Sebastiano Panichella Harald C. Gall Filomena Ferrucci Andrea De Lucia Fabio Palomba, Pasquale Salza. Recommending and localizing change requests for mobile apps based on user reviews. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 106–117, 2017.