|  |  |
| --- | --- |
|  | 学校代码： 10246 |
|  | 学 号： 15212010001 |
|  |  |



|  |
| --- |
| 硕 士 学 位 论 文 |

（学术学位）

**基于共同修改关系的热点检测与隐式共享依赖分析**

**Hotspot Detection and Implicit Shared Dependency Analysis based on Co-change Relation**

|  |  |
| --- | --- |
| 院 系： | 软件学院 |
| 专 业： | 计算机软件与理论 |
| 姓 名： | 陈恒 |
| 指 导 教 师： | 彭鑫 教授 |
| 完 成 日 期： | 2018年 3 月 30 日 |

指导小组成员名单

彭鑫 教授

赵文耘 教授

吴毅坚 副教授

沈立炜 副教授

目录

[中文摘要 I](#_Toc510095333)

[Abstract II](#_Toc510095334)

[第一章 引言 1](#_Toc510095335)

[1.1 研究背景 1](#_Toc510095336)

[1.2 现有工作总结与不足 1](#_Toc510095337)

[1.3 本文主要工作与贡献 2](#_Toc510095338)

[1.4 本文结构安排 3](#_Toc510095339)

[第二章 相关工作 4](#_Toc510095340)

[2.1 共同修改分析与经验研究 4](#_Toc510095341)

[2.2 变更影响分析 4](#_Toc510095342)

[2.3 设计问题检测 5](#_Toc510095343)

[2.4 本章小结 6](#_Toc510095344)

[第三章 基于共同修改关系的热点检测方法 7](#_Toc510095345)

[3.1 研究动机 7](#_Toc510095346)

[3.2 术语说明与方法概述 10](#_Toc510095347)

[3.3 共同修改文件对识别与关系挖掘 11](#_Toc510095348)

[3.3.1 数据获取与预处理 11](#_Toc510095349)

[3.3.2 共同修改文件对识别 12](#_Toc510095350)

[3.3.3 修改操作提取 13](#_Toc510095351)

[3.3.4 共同修改关系挖掘 16](#_Toc510095352)

[3.4 热点检测 17](#_Toc510095353)

[3.4.1 共同修改图生成 17](#_Toc510095354)

[3.4.2 层次聚类算法 18](#_Toc510095355)

[3.4.3 热点检测规则 20](#_Toc510095356)

[3.5 本章小结 21](#_Toc510095357)

[第四章 隐式共享依赖分析 22](#_Toc510095358)

[4.1 经验研究过程 22](#_Toc510095359)

[4.2 平行共同修改关系定义与分类 23](#_Toc510095360)

[4.3 平行共同修改关系挖掘 24](#_Toc510095361)

[4.3.1 关系判定算法 24](#_Toc510095362)

[4.3.2 修改操作语法解析 25](#_Toc510095363)

[4.3.3 修改操作判定规则 27](#_Toc510095364)

[4.4 分析方法 29](#_Toc510095365)

[4.5 本章小结 30](#_Toc510095366)

[第五章 实验与经验研究 31](#_Toc510095367)

[5.1 研究项目 31](#_Toc510095368)

[5.2 热点检测实验 32](#_Toc510095369)

[5.2.1 研究问题 33](#_Toc510095370)

[5.2.2 实验设计 33](#_Toc510095371)

[5.2.3 实验结果与分析 35](#_Toc510095372)

[5.3 隐式共享依赖经验研究 38](#_Toc510095373)

[5.3.1 统计分析 38](#_Toc510095374)

[5.3.2 案例分析 40](#_Toc510095375)

[5.3.3 演化分析 44](#_Toc510095376)

[5.4 本章小结 47](#_Toc510095377)

[第六章 总结和展望 48](#_Toc510095378)

[6.1 总结 48](#_Toc510095379)

[6.2 展望 48](#_Toc510095380)

[参考文献 49](#_Toc510095381)

[致谢 52](#_Toc510095382)

[攻读硕士学位期间发表论文 53](#_Toc510095383)

# 中文摘要

软件演化历史中蕴藏了大量有价值的信息，通过分析演化历史我们可以得到对软件开发与维护有意义的经验和结论。演化历史分析的一个重要工作是对演化耦合的研究，即对共同修改的研究。共同修改指的是两个文件在同一次提交中被修改的现象[1]。现有演化耦合相关工作通常利用共同修改信息进行变更影响分析和设计问题检测，由于对共同修改信息的挖掘和利用不够深入，所以得到的结论无法给开发人员提供更加直接而深入的帮助和指导。对于变更影响分析，开发人员不仅需要知道当某个文件被修改时，其他哪些文件同样需要修改，还需要知道如何对这些文件进行修改。对于设计问题检测，如果一个软件项目过多的文件都被检测出存在设计问题，则无法给开发人员提供更为聚焦的关注点，另一方面，开发人员也缺少细粒度的信息去分析和诊断设计问题的形成过程和根本原因。

因此本文提出了共同修改关系的概念，它是两个文件在演化历史中多次出现的共同修改类型(修改类型的二元组)，体现了文件间细粒度的隐式依赖。利用共同修改关系，我们对项目快照中的文件进行层次聚类，接着结合启发式规则在聚类结果中检测出热点。一个热点是一组存在高度演化耦合，并且具有层次结构的文件，热点中的核心文件是演化耦合最严重的文件，也是热点形成的关键。本文实验结果表明了热点文件历史上的维护开销远远高于项目中的其他文件，核心文件的维护开销也高于热点中的其他文件。因此开发人员需要及时注意到这些热点，结合核心文件、层次结构等信息，分析发现热点中存在的不良设计进行改进，防止热点进一步扩大，带来越来越高的维护开销。

在共同修改关系的基础上，本文进一步定义和挖掘了平行共同修改关系，它表示两个文件历史上至少在同一次提交中发生了相同的修改。通过经验研究，我们得到了软件项目共同修改的一些基本结论。特别的是，我们发现平行共同修改关系在软件项目中普遍存在，并且可以揭示文件间无法通过结构关系发现的隐式共享依赖，此外我们还总结了平行共同修改关系四类典型的演化趋势。本文经验研究的结论可以帮助开发人员更好地理解文件间的共同修改和隐式共享依赖。

**关键词：**演化耦合，共同修改关系，热点检测，隐式共享依赖

**中图文分类号：**TP311

# Abstract

There is a lot of valuable information in the history of software evolution. By analyzing evolution history, we can obtain meaningful experiences and conclusions on software development and maintenance. An important work of evolution history analysis is the study of evolutionary coupling, that is, the study of co-change. Co-change refers to the phenomenon that two files were modified in the same commit [1]. The existing works related to evolutionary coupling usually perform change impact analysis and design problem analysis utilizing co-change information. However, these works do not mine and utilize co-change information deeply enough, the conclusions cannot provide developers with more direct and in-depth help and guidance. For change impact analysis, developers need to know not only what files should be modified when one file is modified, but also how to modify these files. For design problem analysis, if too many files in a software project are detected to have design problems, developers cannot pay attention to a small range of files. On the other hand, they lack fine-grained information to analyze and diagnose the formation and the root cause of design problems.

Therefore, this paper proposes the concept of the co-change relation. A co-change relation is a recurring co-change type (a two-tuple of change types) in evolution history. Co-change relations can reflect fine-grained implicit dependencies between files. Utilizing co-change relations, we hierarchically cluster the files in the project snapshot and detect hotspots in the clustering results with heuristic rules. A hotspot is a group of files with high evolutionary coupling and a hierarchical structure. Each hotspot has one or more core files which have the broadest evolutionary coupling with others in the hotspot. Experimental results show that files in hotspots have much higher maintenance cost than others in the project and core files have higher maintenance cost than others in the hotspots. Developers should pay attention to these hotspots in time. After that, they can analyze and discover the bad design that exists in these hotspots with core files, hierarchical structures and other information. In this way, developers can improve the bad design to prevent further expansion of hotspots and higher maintenance cost caused by these hotspots.

On the basis of co-change relations, this paper further defines and mines parallel co-change relations. These relations indicate the same modifications had occurred in two files in at least one commit in evolution history. Through empirical study, we have obtained some basic conclusions of co-change. In particular, we have found that parallel co-change relations are common in software projects. These relations can reveal implicit shared dependencies that cannot be detected by structural relationship between files. In addition, we also summarize four typical evolutionary trends of parallel co-change relations. The conclusions drawn by our empirical study can help developers better understand the phenomenon of co-change and implicit shard dependencies between two files.

**Keywords:** Evolutionary Coupling, Co-change Relation, Hotspot Detection, Implicit Shared Dependency

**CLC Number:** TP311

# 第一章 引言

## 1.1 研究背景

随着软件需求的不断增加和变化，软件的规模和复杂度也不断增加，如果不及时发现并解决软件中存在的设计问题，软件质量会越来越差，软件也会变得越来越难维护，导致维护开销越来越昂贵。软件演化历史中蕴藏了大量有价值的信息，通过对演化历史的分析，我们可以发现软件在开发历史中是如何演化的，软件设计是如何逐渐腐化的，设计问题给软件项目带来了怎样的维护开销等经验和结论。这些经验和结论不仅可以帮助开发人员及时分析和发现软件项目中存在的设计问题，指导开发人员通过重构等手段改善不良设计，而且对后续的软件开发与维护提供了指导意义。

演化历史分析的一个重要工作是对演化耦合的研究，即对共同修改的研究。共同修改指的是两个文件在同一次提交中被修改的现象[1]。广泛应用的版本控制系统，如Git和SVN，记录了开发人员提交代码的完整历史，其中也包括共同修改的信息。两个文件历史上多次在同一次提交中被修改，意味着这两个文件间存在演化耦合，两个文件共同修改的次数越多，表明它们间的演化耦合越强，存在演化耦合的文件间，不一定存在结构关系。没有结构关系的文件间的频繁共同修改，通常可以揭示文件间的隐式依赖[2]，这样的文件被认为存在模块违约(Modularity Violation)的设计问题[3]。有结构关系的文件间的频繁共同修改，也可以揭示一些设计问题，例如不稳定的接口，不稳定的接口被认为是形成技术债务的重要原因之一[4]。

## 1.2 现有工作总结与不足

研究人员利用共同修改信息对演化耦合进行了大量的分析研究，主要包括共同修改分析与经验研究、变更影响分析以及设计问题检测。然而现有工作对演化耦合研究的粒度通常停留在文件或方法层面，仅仅计算共同修改次数(两个文件或方法在多少次提交中被一起修改)和共同修改概率(当一个文件或方法发生修改时，另一个文件或方法被修改的概率)等度量，通过关联规则挖掘、聚类和模式匹配等方法对演化耦合进行研究。由于这些工作对共同修改信息的挖掘和利用不够深入，所以得到的结论往往无法给开发人员提供更加直接而深入的帮助和指导。对于变更影响分析，开发人员不仅需要知道当某个文件被修改时，其他哪些文件同样需要修改，还需要知道如何对这些文件进行修改。对于设计问题检测，如果一个软件项目过多的文件都被检测出存在设计问题，则无法给开发人员提供更为聚焦的关注点，另一方面，开发人员也缺少细粒度的修改信息，进一步分析文件为什么以及如何被共同修改的，更加准确地定位设计问题乃至分析设计问题形成的根本原因。比如，当一个不稳定接口被检测出来，开发人员需要更详细的信息去分析和诊断它的形成过程和原因。同样地，模块违约的形成原因也有多种，比如共享数据接口，共享业务逻辑等。因此需要一种细粒度的方法对演化耦合，即共同修改进行更加深入的研究。

## 1.3 本文主要工作与贡献

基于现有工作对共同修改信息挖掘和利用不够深入的现状，本文旨在对共同修改进行更加细粒度的分析与研究，帮助开发人员更好地理解演化耦合，发现软件项目中存在的设计问题，本文的主要工作与贡献有以下三点：

1) 本文提出了共同修改关系的概念，它是两个文件在演化历史中多次出现的共同修改类型(修改类型的二元组)。相比共同修改次数和共同修改概率，共同修改关系体现了文件间细粒度的演化耦合，或者说隐式依赖。

2) 本文利用共同修改关系对软件项目当前快照中的文件进行层次聚类，得到演化耦合视角下文件的层次结构，进一步结合启发式规则在聚类结果中检测出热点。一个热点是一组存在高度演化耦合，并且具有层次结构的文件。高度的演化耦合往往是由于不良设计没有得到及时改进而逐渐形成的，这些热点会给软件项目带来昂贵的维护开销，因此开发人员需要及时注意到这些热点，分析并解决其中的设计问题。

3) 在共同修改关系的基础上，本文进一步定义了平行共同修改关系，它表示两个文件历史上至少在同一次提交中发生了同样的修改。共同修改关系体现了文件间的隐式共享依赖，这样的依赖往往无法通过结构关系发现，是造成文件频繁共同修改和产生额外维护开销的重要原因之一。

实验结果表明了本文方法检测出的热点文件历史上的维护开销确实远远高于项目中其他文件的维护开销，并且检测出的热点比其他方法(仅利用共同修改次数结合结构关系检测热点)更为聚焦，即检测出的热点文件范围更小，但是这些文件历史上维护开销更高。开发人员应该及时注意到这些热点，结合热点核心文件、层次结构和共同修改关系等信息，分析其中存在的不良设计进行改进，防止热点进一步扩大，给软件项目带来越来越高的维护开销。

本文经验研究得到了软件项目共同修改现象的一些基本结论。特别的是，我们发现平行共同修改关系在软件项目中普遍存在，并且可以揭露文件间无法通过结构关系发现的隐式共享依赖。隐式共享依赖是造成文件频繁共同修改的重要原因之一，开发人员需要及时注意到这些依赖，结合具体情况分析这些依赖的形成原因，对涉及的文件进行更好的封装和设计，避免产生额外的维护开销。此外我们还总结了平行共同修改关系四类典型的演化趋势。本文经验研究的结论可以帮助开发人员更好地理解文件间的共同修改现象和隐式共享依赖。

## 1.4 本文结构安排

本文后面的结构安排如下：第二章介绍本文的相关工作，第三章介绍基于共同修改关系的热点检测方法，第四章介绍隐式共享依赖分析，第五章是实验和经验研究，包括对第三章的方法进行实验验证，以及对共同修改现象，特别是隐式共享依赖的经验研究，第六章对本文进行总结与展望。

# 第二章 相关工作

本章主要介绍演化耦合相关工作，分为共同修改分析与经验研究、变更影响分析和设计问题检测三个方面。

## 2.1 共同修改分析与经验研究

Wong S等人[5]利用马尔科夫链模型(Markov Chain Model)形式化描述演化耦合为一个随机过程(Stochastic Process)，通过调节模型的参数，定义了一组随机依赖(Stochastic Dependency)，表示四类不同的演化路径(Evolution Path)。D’Ambros M等人[6]提出了一个可视化方法，集成了各类共同修改信息，方便开发人员更好地观察和理解演化耦合。Silva L L等人[7]根据共同修改次数对文件进行聚类，得到共同修改簇(Co-change Cluster)，共同修改簇是一组经常一起修改的文件，进一步将共同修改簇与簇中文件所在包的投影(Projection)归纳总结为三种模式(Pattern)。Beck F等人[8]通过经验研究，从开发人员的角度分析多个代码耦合概念和模块化(Modularity)之间的一致性，代码耦合概念包括结构依赖(Structural Dependency)、扇出相似度(Fan-out Similarity)、演化耦合、代码所有权(Code Ownership)、克隆代码(Clone Code)和语义相似度(Semantic Similarity)。Bavota G等人[9]从实践的角度，研究了学术界提出的结构、动态、语义和演化耦合度量值是否符合工业界，即开发人员对耦合的理解。Kirbas S等人[10]通过对工业界大型软件项目的研究，分析演化耦合跟缺陷之间的关系。这些工作对共同修改的研究停留在文件层面，仅仅分析是否发生了共同修改或者共同修改了多少次，由于缺乏细粒度的修改信息，得到的结论无法给开发人员提供更深入的指导。

还有一些学者对更细粒度的程序实体间的共同修改现象进行了研究，比如Mondal M等人[11]对方法间的共同修改进行了经验研究，并提出了修改关联(Change Correspondence)度量值，用于评估程序实体间的修改是否是关联的[12]，然而对方法共同修改信息的利用仍然只是计算共同修改次数或概率。Mondal M等人[13]还关注克隆片段(Clone Fragment)间的共同修改，通过经验研究分析克隆片段是否与非克隆片段或者来自其他克隆类(Clone Class)的克隆片段间存在演化耦合，该工作将共同修改限定在克隆代码的范围内，不具有普遍意义。Jiang Q等人[14]通过分析历史修改记录，挖掘出轨迹模式(Trajectory Pattern)，该工作同样基于对细粒度修改操作的提取，但是没有考虑演化耦合特性。

## 2.2 变更影响分析

Zimmermann T等人[1]借鉴了购物网站基于历史购买记录推荐的思想，对演化历史中的代码修改数据进行分析和挖掘，得到了一系列关联规则(Association Rule)，将这些关联规则用于预测和推荐下一步修改，防止开发人员因为修改不完全引入缺陷，以及检测出仅仅通过程序分析手段无法发现的程序实体间的演化耦合。为了提高变更影响分析的准确率，除了考虑需求文本(Textual Change Request)和代码快照(Snapshot of Source Code)，有时候还需要结合上下文信息(Contextual Information)，比如执行轨迹(Execution Trace)等，因此Gethers M等人[15]提出了一种自适应方法，计算出上下文信息与需求文本和代码快照的最佳组合，用于生成影响集合(Impact Set)。Li B等人[16]对基于代码的变更影响分析进行了综述，综述中指出修改集合(Change Set)和影响集合总是在同一个粒度上。由于较少有工作关注修改的影响范围，Hassaine S等人[17]受地震学研究的启发，建立了地震学和修改传播(Change Propagation)概念的映射，用于研究修改传播的范围。

关联规则挖掘是一种无监督学习技术，基于关联规则挖掘进行变更影响分析的效果会被多个因素影响：1)挖掘算法的参数；2)修改集合的特征；3)项目版本历史的特征，因此Moonen L等人[18]进行了经验研究，分析这些因素在多大程度上影响修改推荐的质量，并提出了一些实践指南(Practice Guideline)。这些工作关注的还是，当一个文件或方法发生修改时，其他哪些文件或方法需要同时被修改，由于缺少细粒度的修改信息，并不能进一步提示开发人员如何修改。

Nguyen H A等人[19]利用潜在狄利克雷分布(Latent Dirichlet Allocation)从程序实体的共同修改信息中捕获变更任务 (Change Task)上下文，并利用上下文进行语句级别的代码修改推荐，然而他们的工作关注的是发生在不同位置重复且相似的修改。还有一些工作研究了代码和构建配置(Build Configuration)的共同修改现象，当代码发生修改时，预测构建配置的修改[20][21][22]。本文的关注点还是代码文件的共同修改和细粒度修改信息。

## 2.3 设计问题检测

Wong S等人[3]提出方法利用共同修改信息检测由于违背设计规则(Design Rule)导致的设计缺陷(Design Defect)。Kouroshfar E等人[23]研究了共同修改分散度(Dispersion)的特征对软件质量的影响。Schwanke R等人[2]结合结构关系和演化历史，通过识别相关度量的异常值，分析发现架构问题(Architecture Issue)。Mo R等人[24]定义了一系列热点模式(Hotspot Pattern)，热点是重复出现的架构问题，并提出了结合结构关系和共同修改信息的热点检测方法，我们在实验中与此方法进行对比。Mo R等人[25]还提出了一个新的架构维护度量值，即解耦度(Decoupling Level)，用于评价软件项目多大程度上可以被解耦成小而独立的可替换模块。Xiao L等人[4]对架构债务(Architectural Debt)进行了形式化定义，并且提出了自动化识别架构债务、量化架构债务维护开销和模拟架构债务维护开销演化情况的方法。这些工作关注的是文件级别的共同修改，并且对共同修改信息的利用不够深入，仅仅计算文件的共同修改次数和共同修改概率。

Palomba F等人[26]提出了利用演化历史中的共同修改信息检测代码坏味道(Bad Smell)的方法，他们的工作虽然分析细粒度的代码修改，但是对代码坏味道的检测依赖于自定义的检测规则，并且无法揭示架构相关的设计问题。

## 2.4 本章小结

本章主要从共同修改分析与经验研究、变更影响分析和设计问题检测三个方面对演化耦合相关工作进行了介绍。

# 第三章 基于共同修改关系的热点检测方法

本章主要介绍基于共同修改关系的热点检测方法，首先结合共同修改关系和热点的例子，阐述研究动机，然后介绍相关术语在本方法场景下的说明，接着概述热点检测方法，最后对本方法的两个主要步骤分别进行详细介绍。

## 3.1 研究动机

图3-1是Cassandra项目(本文实验项目之一)中共同修改关系的3个例子。本文定义的共同修改关系是两个文件间，历史上多次出现的共同修改类型(修改类型二元组)。

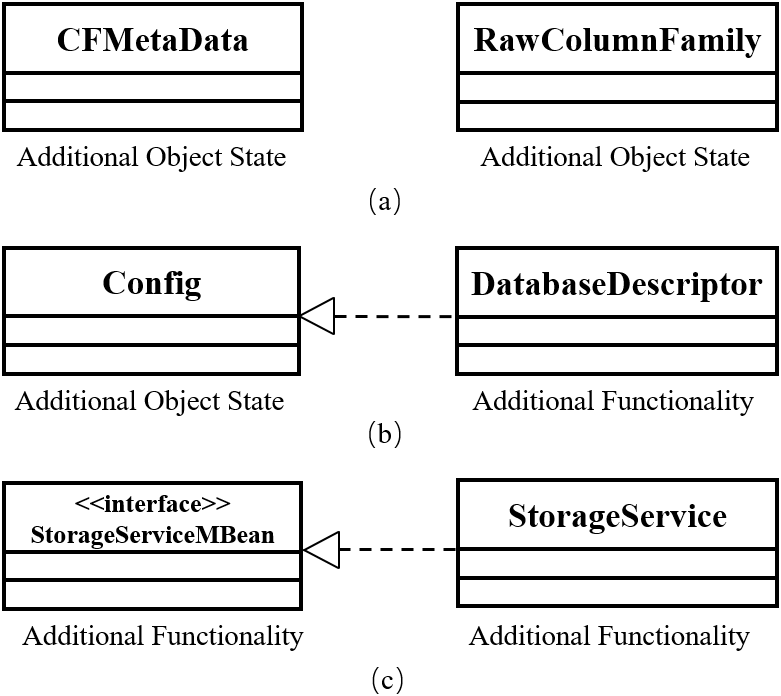


图3-1 共同修改关系例子

图3-1(a)中的两个文件分别是CFMetaData类和RawColumnFamily类，它们间不存在任何结构关系，但是这两个文件历史上一共在7次提交中被共同修改。修改类型增加属性(Additional Object State)组成的二元组(Additional Object State，Additional Object State)是一个共同修改类型，这个共同修改类型一共出现了6次，超过了一定的阈值，因此被认为是CFMetaData类和RawColumnFamily类间的共同修改关系。这个关系意味着在这6次提交中，CFMetaData类发生了增加属性的修改，同时RawColumnFamily类发生了增加属性的修改。结合修改信息，我们进一步发现，这两个文件在其中5次提交中，增加的属性的类型和标识符相同，这说明它们间存在这样的隐式依赖，即共享了多个属性。由于这两个文件间不存在任何结构关系，当它们早期只共享少量属性的时候，这样的隐式依赖较难被开发人员发现，导致在后续的开发过程中，这两个文件共享的属性越来越多，属性耦合越来越严重，没有结构关系的文件间的频繁共同修改和隐式依赖不利于项目的维护和理解。

图3-1(b)中的两个文件分别是Config类和DatabaseDescriptor类，它们间的结构关系是DatabaseDescriptor类中的一个属性的类型是Config类。这两个文件历史上一共在206次提交中被共同修改，(Additional Object State, Additional Functionality)这个共同修改关系一共出现了88次，意味着在这88次提交中，Config类发生了增加属性的修改，同时DatabaseDescriptor类发生了增加方法的修改。结合修改信息，我们进一步发现，随着Config类的属性越来越多，它逐渐形成了一个只拥有属性，而不提供其他任何功能性方法的类，乃至产生了数据类(Data Class)的代码坏味道，此外每当Config类增加一个属性，DatabaseDescriptor类就要增加对应的方法，提供对新增属性的访问和修改。这个共同修改关系体现的Config类和DatabaseDescriptor类间的隐式依赖，显然无法仅通过分析这两个文件间的结构关系发现。

图3-1(c)中的两个文件分别是StorageServiceMBean接口和StorageService类，它们间的结构关系是StorageService类实现了StorageServiceMBean接口，这两个文件历史上一共在177次提交中被共同修改，(Additional Functionality，Additional Functionality)这个共同修改关系一共出现了77次，意味着在这77次提交中，StorageServiceMBean接口发生了增加方法声明的修改，同时StorageService类发生了增加方法的修改。虽然通过这两个文件间的结构关系，我们知道每当StorageServiceMBean接口增加了方法声明，StorageService类必然要增加对应的方法实现，但这样的关系出现了88次，频繁出现说明了StorageServiceMBean接口是一个不稳定的接口，承担了过多的职责，违背了单一职责的设计原则，而StorageService类将影响继续传播给其他的文件。这说明有结构关系的文件间的共同修改关系，也可以反映出其他的设计问题。

通过这3个例子，我们发现没有结构关系的文件间可能存在共同修改关系，有结构关系的文件间也可能存在共同修改关系，并且共同修改关系体现的文件间的隐式依赖，往往无法通过结构关系发现。因此本文提出了基于共同修改关系的热点检测方法，首先我们利用共同修改关系以及关系的出现次数，定义了距离函数计算文件间的距离，距离越小，则表示文件间的隐式依赖越强，然后对项目当前快照中的文件，根据文件间的距离进行层次聚类，聚类结果是演化耦合视角下，项目文件的层次结构，最后结合启发式规则，在聚类结果中检测出热点，一个热点是符合一定规则的聚类簇。

图3-2 热点例子

图3-2展示了本文方法在Cassandra项目中检测出的一个热点的例子，从演化耦合的角度看，CassandraDaemon类与DatabaseDescriptor类关系最密切，组成了聚类簇Cluster1，Cluster1与FBUtilities类关系最密切，组成了聚类簇Cluster2，Cluster2与ByteBufferUtil类关系最密切，组成了聚类簇Cluster3，Config类和StorageProxyMBean接口关系最密切，组成了聚类簇Cluster4，Cluster3与Cluster4关系最密切，最终组成了聚类簇Cluster5，Cluster5是本文检测出的一个热点。在这个热点中，核心文件是DatabaseDescriptor类，DatabaseDescriptor类在该热点内影响力最大，影响力最大指的是与热点内最多的文件间存在共同修改关系，DatabaseDescriptor类与该热点内4个文件间都存在共同修改关系，分别是Config类、StorageProxyMBean接口、FBUtilities类以及CassandraDaemon类。

共同修改关系一定程度上体现了文件间的隐式依赖，即演化耦合视角下，文件间不同于结构关系的依赖，而热点是一组根据共同修改关系聚类出的具有层次结构的文件，因此热点是一组存在高度演化耦合，并且具有层次结构的文件，核心文件是热点内演化耦合最严重的文件，是热点形成的关键。

高度的演化耦合往往是由于不良设计导致的，通常会给软件项目带来较高的维护开销。本文实验结果表明了热点中的文件，历史上的维护开销确实远远高于其他文件，并且热点中的核心文件也比热点中其他文件的维护开销更高。因此，开发人员需要及时注意到这样的热点，结合热点的层次结构、核心文件以及文件间共同修改关系等信息，分析发现热点中存在的不良设计，通过重构等手段改善设计，将热点分解为小而独立的模块，降低不同模块间的耦合，防止热点进一步扩大，给软件维护带来越来越高的维护开销。

## 3.2 术语说明与方法概述

在介绍本文方法之前，首先对涉及的一系列术语给出在本方法场景下的说明。

共同修改(Co-change)指的是两个文件在同一次提交中发生了修改的现象。

共同修改提交(Co-change Commit)是存在共同修改现象的提交，表明这次提交修改了多于一个文件。

共同修改文件对(Co-change File Pair)是两个多次在同一次提交中发生修改的文件。我们用二元组(, )表示一个共同修改文件对，它意味着文件和文件多次在同一次提交中发生了修改。

修改操作(Change Operation)指的是代码文件在一次提交中发生的原子代码修改，修改操作的修改类型(Change Type)有增加属性(Additional Object State)，增加方法(Additional Functionality)等。根据不同的修改类型我们可以区分不同的修改操作。

共同修改类型(Co-change Type)是一个共同修改文件对包含的两个文件间，历史上出现过的修改类型的组合。对一个共同修改文件对(, )而言，它的一个共同修改类型是一个二元组(, )，其中是文件的修改类型，是文件的修改类型，这个共同修改类型意味着至少在历史上的某一次提交中，发生了修改类型为的修改操作，同时发生了修改类型为的修改操作。

共同修改关系(Co-change Relation)是多次出现的共同修改类型。由于两个文件间可能存在非常多的共同修改类型，但不是每一个共同修改类型在演化历史中都频繁出现，因此共同修改关系只表示出现次数超过一定阈值的共同修改类型。对共同修改文件对(, )而言，它的一个共同修改关系(, )表示历史上出现了多次这样的情况，即在同一次提交中，发生了修改类型为的修改操作，同时发生了修改类型为的修改操作。

共同修改图(Co-change Graph)是描述一组文件以及这组文件间共同修改关系的带权无向图，其中顶点表示文件，边表示顶点对应的文件间是否存在属于频繁关系集合(出现次数最多的共同修改关系集合)的关系，边的权重根据文件间的共同修改关系和关系出现次数计算得到。

热点是一组存在高度演化耦合，并且具有层次结构的文件。每个热点包括一个或多个核心文件，核心文件是热点中影响力最大的文件，即与该热点内最多的文件间存在共同修改关系的文件。

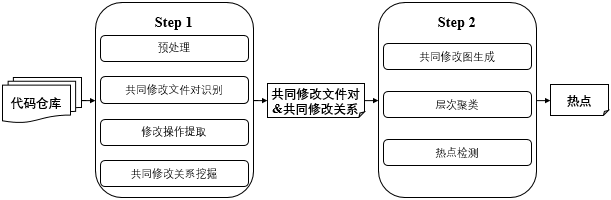


图3-3 方法流程

图3-3描述了本方法的整体流程，输入是代码仓库，输出是热点。本方法分为两个主要步骤，第一步识别和挖掘出共同修改文件对和共同修改关系，涉及的子步骤有预处理、共同修改文件对识别、修改操作提取和共同修改关系挖掘，第二步检测出热点，涉及的子步骤有共同修改图生成、层次聚类和热点检测。本章出现的所有阈值均在5.2.2节给出设置以及合理性说明。

在第一步中，首先从代码仓库中获取完整的提交记录，其次通过预处理得到一系列共同修改提交，然后识别出所有的共同修改文件对，接着对每个共同修改文件对中的两个文件，提取其在对应的每次共同修改提交中的修改操作，最后根据识别出的共同修改文件对和抽取出的修改操作挖掘出共同修改关系。在第二步中，首先根据第一步识别和挖掘出共同修改文件对和共同修改关系，生成共同修改图，然后通过层次聚类算法对图中的文件进行聚类，最后根据启发式规则在聚类结果中筛选出热点。

## 3.3 共同修改文件对识别与关系挖掘

本节首先介绍数据获取与预处理，接着介绍如何识别共同修改文件对，然后介绍如何提取修改操作，最后介绍如何挖掘共同修改关系。

### 3.3.1 数据获取与预处理

版本控制系统(Version Control System, VCS)通常用于对软件项目的版本管理，记录了软件项目完整的演化历史，提供了追溯文件修改历史的功能[27]。常用的版本控制系统有Git和SVN，每个项目在版本控制系统中对应一个代码仓库，代码仓库里记录了每一次提交(Commit)的原始信息，比如提交ID，提交时间，提交作者，修改的文件，修改的文件内容等信息。我们根据这些信息对所有的原始提交进行预处理，可以得到所有的共同修改提交。

预处理的流程为：

1) 首先，过滤掉合并的提交，即当前提交存在超过一个父提交(Parent Commit)。因为该提交仅用于将指定分支的提交合并到当前分支，合并的提交内一般不对文件进行修改。

2) 其次，对每一次提交，根据修改的文件名后缀过滤掉其中的非代码文件，如脚本文件(Script File)，自述文件(README File)等。因为本文关注的是共同修改的代码文件，且只有代码文件才可以被抽取出修改操作。

3) 然后，对每一次提交，过滤掉其中的测试文件，可以根据文件的路径和命名规则进行过滤，例如测试文件通常存在于test目录下，或者以Test为开头或结尾命名。因为开发人员通常一起修改并提交测试文件与被测试文件，测试文件通常是为了保证被测试文件的功能正确性，开发人员更关注被测试文件。

4) 接着，过滤掉修改了少于两个文件的提交。因为共同修改提交要求至少修改两个文件。

5) 最后，过滤掉修改了大量文件的提交。这些工作认为此类提交通常是高度分散的，目的是为了实现某些特殊的维护任务(比如修改了分布在大量代码文件注释中的许可证协议) [1][7][18]。设置最大修改文件数阈值为，意味着过滤掉修改文件个数多于的提交。

### 3.3.2 共同修改文件对识别

利用预处理阶段得到的共同修改提交记录，我们可以识别出所有的共同修改文件对。一个共同修改提交文件对包括两个文件，这两个文件在历史上多次在同一次提交中被修改。图3-4展示了共同修改文件对的识别过程。首先枚举出所有共同修改提交中包含的文件对，例如提交修改了、和三个文件，则对应的文件对有3个，即(, )、(, )和(, )，然后对每一个文件对，统计其共同修改次数，即该文件对在所有提交中出现的次数，例如文件对(, )一共出现了两次，分别出现在和中，最后设置最小修改次数阈值为，所有出现次数不少于的文件对被认为是共同修改文件对。如果设置为3，则该图中的共同修改文件对为(, )，因为只有文件对(, )在所有提交中的出现次数不少于3次。

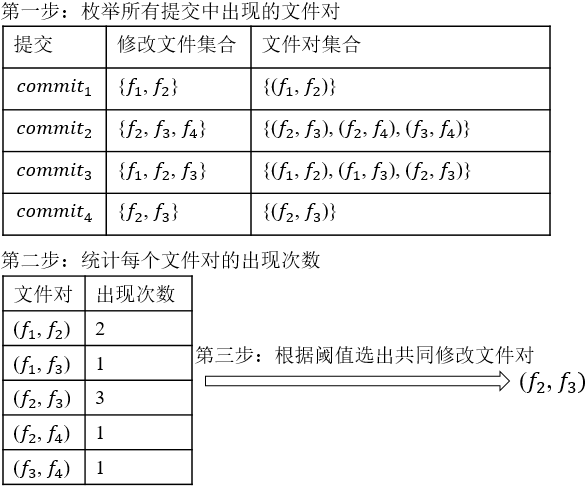


图3-4 共同修改文件对识别过程

### 3.3.3 修改操作提取

为了挖掘出每个共同修改文件对包含的两个文件间的共同修改关系，需要抽取出这两个文件在对应的共同修改提交中发生的修改操作。修改操作代表一个文件在一次提交中发生的原子代码修改，本文定义修改操作为一个六元组，属性包括：提交(Commit)，文件(File)，修改类型(Change Type)， 修改的实体类型(Changed Entity Type)，修改的实体内容(Changed Entity Content)和更新后的实体内容(New Entity Content)。其中Commit是发生修改的提交；File是发生修改的文件；Change Type是修改操作的修改类型，例如修改属性类型(Attribute Type Change)，增加参数(Parameter Insert)等；Changed Entity Type是修改的实体的类型，例如方法(Method)，属性(Field)，If语句(If Statement)等；对于增加(Insert)和删除(Delete)类的修改操作，Changed Entity Content是增加或者删除的实体内容，对于更新(Update)类的修改操作，Changed Entity Content是修改前的实体内容，New Entity Content是修改后的实体内容。

Fluri B等人定义了原子代码修改的修改类型[28]，进一步地，提出了修改蒸馏(Change Distilling)算法[29]，并提供了开源工具ChangeDistiller[30]，用于抽取一个文件在版本控制系统中两个版本间的原子代码修改，表3-1列举了所有原子代码修改的修改类型，一共47种。

表3-1 原子代码修改的修改类型

|  |  |  |  |
| --- | --- | --- | --- |
| **编号** | **修改类型** | **编号** | **修改类型** |
| 1 | Adding Attribute Modifiability | 25 | Parameter Ordering Change |
| 2 | Adding Class Derivability | 26 | Parameter Renaming |
| 3 | Adding Method Overridability | 27 | Parameter Type Change |
| 4 | Additional Class | 28 | Parent Class Change |
| 5 | Additional Functionality | 29 | Parent Class Delete |
| 6 | Additional Object State | 30 | Parent Class Insert |
| 7 | Alternative Part Delete | 31 | Parent Interface Change |
| 8 | Alternative Part Insert | 32 | Parent Interface Delete |
| 9 | Attribute Renaming | 33 | Parent Interface Insert |
| 10 | Attribute Type Change | 34 | Removed Class |
| 11 | Class Renaming | 35 | Removed Functionality |
| 12 | **Comment Delete\*** | 36 | Removed Object State |
| 13 | **Comment Insert\*** | 37 | Removing Attribute Modifiability |
| 14 | **Comment Move\*** | 38 | Removing Class Derivability |
| 15 | **Comment Update\*** | 39 | Removing Method Overridability |
| 16 | **Condition Expression Change+** | 40 | Return Type Change |
| 17 | Decreasing Accessibility Change | 41 | Return Type Delete |
| 18 | **Doc Delete\*** | 42 | Return Type Insert |
| 19 | **Doc Insert\*** | 43 | **Statement Delete+** |
| 20 | **Doc Update\*** | 44 | **Statement Insert+** |
| 21 | Increasing Accessibility Change | 45 | **Statement Ordering Change+** |
| 22 | Method Renaming | 46 | **Statement Parent Change+** |
| 23 | Parameter Delete | 47 | **Statement Update+** |
| 24 | Parameter Insert |  |  |

本文在ChangeDistiller工具抽取出的原子代码修改的基础上，做了两点改进，处理后的原子代码修改作为本文定义的修改操作，一共得到149种修改类型。

第一个改进是过滤掉注释相关的原子代码修改。表3-1中加粗且带“\*”的条目是注释相关的原子代码修改，即修改类型为Comment Delete/ Insert/ Move/ Update，以及Doc Delete/ Insert/ Update的原子代码修改。

第二个改进是对一些粒度较粗的原子代码修改，结合修改实体类型进一步细化其修改类型。表3-1中加粗且带“+”的条目是粒度较粗的原子代码修改，即修改类型为Statement Delete/ Insert/ Ordering Change/ Parent Change/ Update，以及Condition Expression Change的原子代码修改。表3-2列举了修改类型为Statement Delete/ Insert/ Ordering Change/ Parent Change/ Update的原子代码修改对应的所有修改实体类型，其中带“\*”的条目是修改类型为Condition Expression Change的原子代码修改对应的所有修改实体类型。

表3-2 粗粒度原子代码修改对应的修改实体类型

|  |  |  |  |
| --- | --- | --- | --- |
| **编号** | **修改实体类型** | **编号** | **修改实体类型** |
| 1 | Assert Statement | 13 | Method Invocation |
| 2 | Assignment | 14 | Postfix Expression |
| 3 | Break Statement | 15 | Prefix Expression |
| 4 | Catch Clause | 16 | Return Statement |
| 5 | Class Instance Creation | 17 | Switch Case |
| 6 | Constructor Invocation | 18 | Switch Statement |
| 7 | Continue Statement | 19 | Synchronized Statement |
| 8 | **Do Statement\*** | 20 | Throw Statement |
| 9 | **Foreach Statement\*** | 21 | Try Statement |
| 10 | **For Statement\*** | 22 | Variable Declaration Statement |
| 11 | **If Statement\*** | 23 | **While Statement\*** |
| 12 | Labeled Statement |  |  |

第二个改进的理由是Beat Fluri等人定义的部分原子代码修改的修改类型过于笼统，无法根据修改类型区分不同的修改实体类型。例如，对于一个修改类型是增加语句(Statement Insert)，修改实体类型是方法调用(Method Invocation)的原子代码修改，本文将修改类型中的Statement替换为对应的修改实体类型Method Invocation，认为这是一个修改类型为增加方法调用(Method Invocation Insert)的修改操作，这样便可以与修改类型是语句插入(Statement Insert)，修改实体类型是变量声明(Variable Declaration Statement)的原子代码修改进行区分。如果按照原定义，这两个原子代码修改的修改类型是一样的，无法根据修改类型对这两个原子代码修改进行区分。

### 3.3.4 共同修改关系挖掘

我们通过经验研究发现超过一半的共同修改提交只修改不超过3个文件(数据与分析结论见5.3.1节)，大量文件一起被修改的情况比较少，并且这样的修改通常是高度分散的，所以当我们考虑多个文件间的关系时，可以通过计算两两文件间的关系得到，因此本文提出的共同修改关系特指两个文件间的关系，这样有助于简化分析多个文件间的关系。

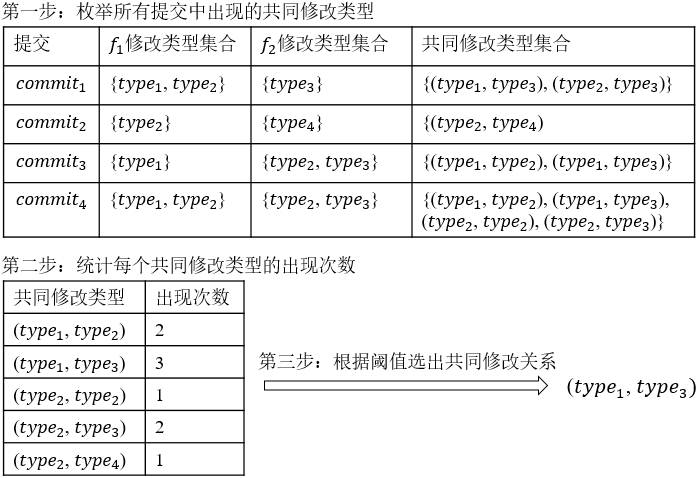


图3-5 共同修改关系挖掘流程

3.2节给出了共同修改关系的定义，图3-1也已经给出了共同修改关系的例子，因此本节重点介绍如何挖掘共同修改关系，图3-5展示了共同修改关系的挖掘过程。对于这个共同修改文件对，包含两个文件和，这两个文件对应一系列共同修改提交，提交的集合为{, , , }，和在所有这些提交中被共同修改。为了挖掘出和的共同修改关系，我们首先提取出和在每一次共同修改提交中的修改操作，根据提取出的修改操作，我们可以得到每一次共同修改提交中，和修改类型的集合，例如在提交中，修改类型的集合为{, }，意味着在中既发生了修改类型为的修改操作，也发生了修改类型为的修改操作，同样地，修改类型的集合为{, }，然后对每一次共同修改提交对应的两个修改类型集合中的元素进行组合，得到所有共同修改类型的集合，例如在提交中，出现的共同修改类型的集合为{(, ), (, ), (, ), (, )}，遍历完和所有的共同修改提交后，我们统计每个共同修改类型的出现次数，例如(, )一共出现了两次，分别出现在和中，最后我们定义最少出现次数阈值，所有出现次数不少于的共同修改类型被认为是共同修改文件对包含的两个文件和间的共同修改关系。如果设置为3，则图中存在的共同修改关系为(, )，因为只有(, )的出现次数不少于3次。

对所有的共同修改文件对挖掘出共同修改关系后，我们统计所有共同修改关系的出现次数，在这一步我们将表示共同修改类型的二元组看作无序的组合，意味着(, )和(, )被认为是同一种关系，然后按照出现次数的多少进行排序，出现次数最多的共同修改关系最具有一般性，它们一定程度上揭示了结构关系体现不出的文件间的隐式依赖，这里定义阈值，最后我们取排名前的共同修改关系组成频繁关系集合。

## 3.4 热点检测

本节主要介绍如何检测热点，首先根据识别出的共同修改文件对、共同修改文件对和频繁关系集合生成共同修改图，然后计算出文件间的距离，利用AGNES层次聚类算法对该图中的文件进行聚类，最后根据启发式规则在聚类结果中检测出热点。

### 3.4.1 共同修改图生成

共同修改图是描述一组文件和文件间共同修改关系的带权无向图。图中的每个顶点表示一个文件，边表示顶点对应的文件间是否存在属于频繁关系集合的关系，如果两个文件间不存在边，则说明它们间不存在任何属于频繁关系集合的关系，边的权重根据顶点对应的两个文件间的共同修改关系和关系出现次数计算得到，权重越大，表示两个文件的隐式依赖越强。我们以项目当前快照中的文件作为顶点生成共同修改图，下面给出两个文件间权重的计算方式：我们定义频繁关系集合为，频繁关系集合一共包括n种关系，因此，对于两个文件和，它们的共同修改次数为，意味着和一共在次提交中被共同修改，我们定义表示关系在文件和间出现了多少次，若和间不存在，则，是和间的权重函数：

(1)

公式(1)表明，如果两个文件间存在属于频繁关系集合的关系越多，并且每种关系的出现次数越多，则这两个文件间的权重越大，表示它们的关系越密切，权重的取值范围为。

我们定义共同修改图的邻接矩阵为共同修改矩阵，该矩阵的行和列表示按照相同顺序排列的文件，我们用该矩阵来存储和表示共同修改图。共同修改矩阵的第X行，第Y列的单元格的值表示文件X和文件Y间的权重。在这一步，我们首先按字典序对项目当前快照中的所有文件进行排序，对这些文件从1开始编号，编号对应共同修改矩阵中的行号和列号，然后遍历这个矩阵，对于第X行，第Y列的单元格，如果文件X和文件Y间存在一条边，则计算它们的权重作为单元格的值，否则记为0，最后得到共同修改矩阵。

### 3.4.2 层次聚类算法

在得到共同修改图和共同修改矩阵后，我们对图中的文件根据文件间的距离进行层次聚类(Hierarchical Clustering)。我们定义为文件和间的距离函数：

(2)

其中是频繁关系集合中包含的关系的个数，公式(2)表明，文件间的权重越大，则距离越小，表示两个文件间的隐式依赖越强，距离的取值范围为。当两个文件间的距离达到最大，为时，说明这两个文件间不存在任何属于频繁关系集合的关系。

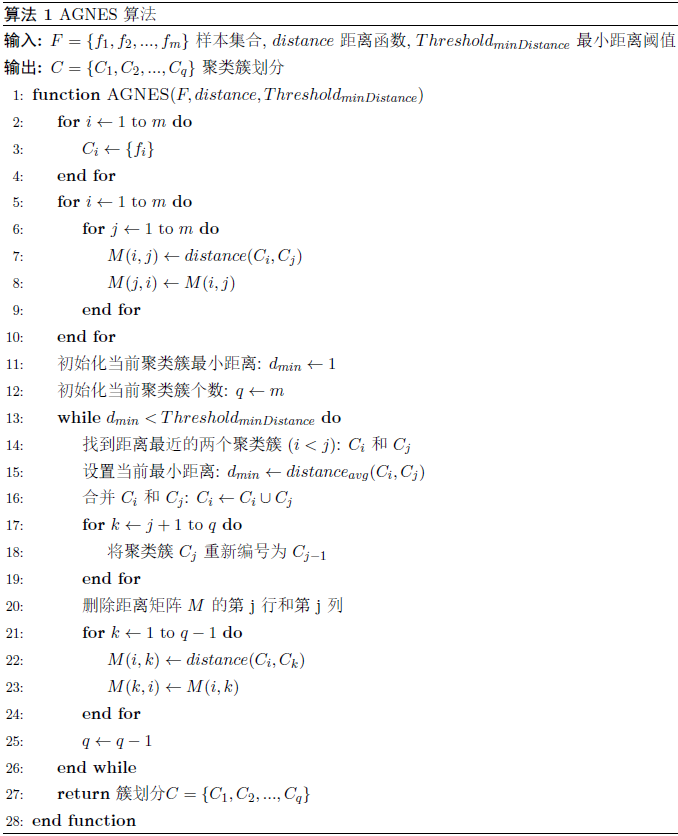


图3-6 AGNES算法伪代码

在聚类阶段，本文采取的是AGNES(Agglomerative NESting)算法，AGNES是一种自底向上采取凝聚策略的层次聚类算法。基本思想是首先将数据集中的每个样本看成一个初始聚类簇，然后每次找出距离最近的两个聚类簇进行合并，不断重复该过程，直到达到预设的条件(最小距离达到阈值，或者聚类簇个数达到阈值)。算法的关键是如何计算两个聚类簇之间的距离，通常有三种策略，最小距离策略、最大距离策略和平均距离策略。最小距离由两个簇的最近样本决定，最大距离由两个簇的最远样本决定，为了减少极端样本对聚类效果的影响，我们采取平均距离策略计算两个聚类簇之间的距离，即对这两个簇中两两样本的距离求平均值。对于两个给定的聚类簇和，定义和的平均距离函数为：

(3)

图3-6展示了AGNES算法的伪代码。输入是样本集合，每一个样本表示项目当前快照中的一个文件，距离函数用于计算两个聚类簇之间的距离，最小距离阈值用于聚类结束的控制条件，我们设置，如果当前聚类簇划分中，距离最近的两个簇间的距离达到这个值，则停止聚类，在此时的聚类簇划分中，任意两个聚类簇中的文件间都不存在任何属于频繁关系集合的关系，输出是聚类簇划分。首先将每一个样本看成一个初始聚类簇，一共有个初始聚类簇，然后根据距离函数计算两两聚类簇之间的距离，得到距离矩阵，接着初始化当前聚类簇最小距离和聚类簇个数，如果最小距离小于最小距离阈值，则继续聚类(在每次聚类中，首先找到距离最近的两个聚类簇和，计算出它们间的距离赋值给最小距离，然后合并这两个聚类，最后更新聚类簇编号，更新距离矩阵，更新当前聚类簇个数)，重复这个聚类过程直到最小距离大于等于最小距离阈值，最后输出此时的聚类簇划分作为最终的聚类结果。最终聚类结果中的每一个聚类簇都是一个树状结构，我们保留了聚类过程中所有被合并聚类簇的原始信息，将最终输出的每一个聚类簇中非叶子节点的子树看作一个独立的聚类簇。

### 3.4.3 热点检测规则

根据共同修改关系对项目当前快照中的文件进行层次聚类后，得到的聚类结果体现了演化耦合视角下，文件的层次结构，接着我们计算出每个聚类簇中影响力最大的文件作为该聚类簇的核心文件，影响力最大指的是与聚类簇中最多的文件间存在共同修改关系(属于频繁共同修改关系集合)，理论上一个热点的核心文件可能多于一个，但是不影响我们接下来的检测规则和分析。处于同一个聚类簇中的文件间存在更高的演化耦合，而核心文件与最多的文件间存在演化耦合。

我们根据下面的启发式规则从聚类结果中选出热点，每个热点是一个聚类簇：1) 热点中总文件个数不少于 。

2) 热点中与核心文件存在共同修改关系的文件个数占其他文件个数的比例不小于。

规则1)保证了筛选出的热点不是局部少数几个文件体现出的问题，而是包含了一定数量的文件，并且这些文件形成了不同于结构关系体现的层次结构，规则2)保证了核心文件与热点中一定比例的文件间存在共同修改关系，意味着核心文件是热点中演化耦合最严重的文件，也是热点形成的关键。因此热点是一组存在高度演化耦合，并且具有层次结构的文件。

高度的演化耦合往往是由于不良设计没有及时得到改进而逐渐形成的，热点通常会给软件项目带来极大的维护开销，这个结论在5.2节中通过实验得到了验证。开发人员需要及时注意到这些热点，结合热点的核心文件和分层结构等信息，分析热点中的不良设计并进行改进，防止热点进一步扩大，乃至形成技术债务，给软件项目维护带来越来越高的维护开销。

## 3.5 本章小结

本章对基于共同修改关系的热点检测方法进行了介绍，通过识别出的共同修改文件对和挖掘出的共同修改关系，对软件项目当前快照中的文件生成共同修改图，根据文件间距离利用AGNES层次聚类算法对共同修改图中的文件进行聚类，最后根据启发式规则在聚类结果中检测出热点。

# 第四章 隐式共享依赖分析

第三章的方法利用共同修改关系和关系出现次数计算出文件间距离，根据距离对项目当前快照中的文件进行层次聚类，结合启发式规则检测出热点，即存在高度演化耦合，并且具有层次结构的文件组。热点的核心文件、层次结构等信息可以帮助开发人员分析发现热点中存在的不良设计，而共同修改关系可以帮助开发人员深入理解文件间的隐式依赖。

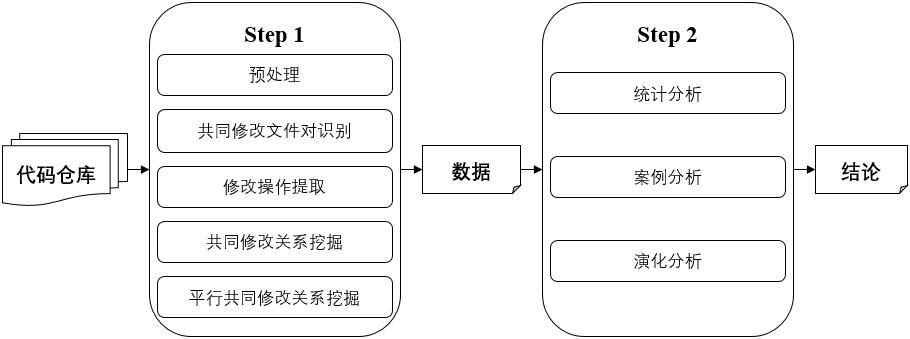
本章旨在对项目的共同修改现象进行经验研究，包括对特定的共同修改关系(平行共同修改关系)进行分析，分析其背后是否存在隐式共享依赖，以及这样的依赖是什么。首先介绍经验研究过程，接着给出平行共同修改关系(Parallel Co-change Relation)的定义和分类，平行共同修改关系表示两个文件至少在同一次提交中发生了相同的修改，一定程度上体现了文件间的隐式共享依赖，然后介绍平行共同修改关系的挖掘方法，主要介绍判断一个共同修改关系是否是平行修改关系的判定算法，以及对修改操作的语法解析和判定规则，最后介绍经验研究的分析方法，包括统计分析、案例分析和演化分析。经验研究重点分析平行共同修改关系在软件项目中的存在和分布情况，此外结合具体案例分析不同平行共同修改关系背后所体现的隐式共享依赖，以及通过可视化手段分析文件间平行共同修改关系的演化情况。通过经验研究，可以帮助开发人员更好地理解项目的共同修改现象以及平行共同修改关系背后体现的隐式共享依赖。

图4-1 经验研究过程

## 4.1 经验研究过程

图4-1展示了经验研究过程，数据源是代码仓库，第一步通过预处理、共同修改文件对识别、修改操作提取、共同修改关系挖掘以及平行共同修改关系挖掘等步骤，我们可以得到项目共同修改的相关数据，第二步对这些数据进行统计分析、案例分析和演化分析，得到共同修改现象的基本结论，以及对隐式共享依赖的深入分析，数据和结论在5.2节中详细介绍。由于预处理、共同修改文件对识别、修改操作提取、共同修改关系挖掘等步骤已经在第三章中进行过详细介绍，因此本章后面的篇幅主要介绍如何挖掘平行共同修改关系和经验研究分析方法。

## 4.2 平行共同修改关系定义与分类

本节给出平行共同修改关系的定义与分类。平行共同修改关系是符合一定条件的共同修改关系，意味着两个文件至少在同一次提交中发生了相同的修改操作。我们从属性，参数，返回值和条件表达式四个维度，对平行共同修改关系进行分类，如表4-1所示。对于每个维度，都有三类平行共同修改关系，即增加，删除和修改，由于平行共同修改关系的定义必然要求关系包含的两个修改类型相同，因此表格中只列出了一个对应的修改类型。4.3节具体介绍如何在共同修改关系的基础上挖掘出平行共同修改关系。

表4-1 平行共同修改关系分类

|  |  |  |
| --- | --- | --- |
| **维度** | **类别** | **对应的修改类型** |
| 属性 | 增加属性 | Additional Object State |
| 删除属性 | Removed Object State |
| 修改属性 | Attribute Renaming/ Type Change |
| 参数 | 增加参数 | Parameter Insert |
| 删除参数 | Parameter Delete |
| 修改参数 | Parameter Renaming/ Type Change |
| 返回值 | 增加返回值 | Return Type Insert |
| 删除返回值 | Return Type Delete |
| 修改返回值 | Return Type Change |
| 条件表达式 | 增加条件表达式 | If Statement Insert |
| 删除条件表达式 | If Statement Delete |
| 修改条件表达式 | If Statement Condition Expression Change |

## 4.3 平行共同修改关系挖掘

第三章识别和挖掘出了共同修改文件对和共同修改关系，每个共同修改文件对，对应一系列共同修改关系，每个共同修改关系，又对应一系列共同修改提交，这个共同修改文件对包含的两个文件，在每次共同修改提交里，存在一系列修改操作。在这些信息的基础上，可以挖掘出该文件对的平行共同修改关系。本节首先介绍判断一个共同修改关系是否是平行共同修改关系的判定算法，然后介绍对修改操作的语法解析，最后介绍根据语法解析结果判断两个修改操作是否相同的判定规则。

### 4.3.1 关系判定算法

对于一个共同修改文件对，它存在一系列共同修改关系，每个共同修改关系形如*()*，首先我们筛选出和相同，且是表4-1列出的修改类型之一的共同修改关系。对于这样的某个共同修改关系*)*，必然满足，我们定义出现了这个关系的共同修改提交集合为，定义文件在提交中修改类型为的修改操作集合为，这里需要注意的是文件在提交中修改类型为的修改操作可能不止一个，每一个修改操作包含3.3.3节定义的属性。

图4-2给出了判断这个共同修改关系是否是平行共同修改关系算法的伪代码。首先初始化提交集合，接着遍历出现了共同修改关系的提交集合，对于每个提交，在提交中修改类型为的修改操作集合为，同样地，在提交中修改类型为的修改操作集合为，然后双重循环遍历这两个集合，对每一组修改操作和，函数判断修改操作和是否相同，如果相同，则在集合中添加提交，这意味着文件和在提交中发生了同样的修改。最后，如果集合中提交的个数大于0，则返回，否则返回。函数根据传入的类型判断两个修改操作是否相同，首先需要对修改操作进行语法解析，语法解析在4.3.2节中详细介绍，在语法解析结果的基础上，进一步对修改操作是否相同进行判定，判定规则见4.3.3节。

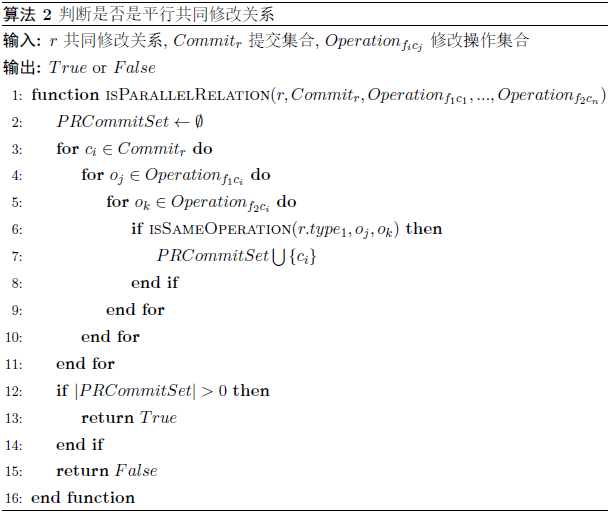


图4-2 判断是否是平行共同修改关系的伪代码

### 4.3.2 修改操作语法解析

为了判断两个修改类型相同的修改操作是否相同，首先需要对修改操作进行语法解析，得到修改操作涉及的程序实体信息，表4-2给出了每类修改操作的语法解析结果。

对于修改实体类型为属性、参数和返回值的修改操作，需要解析出的程序实体信息包括类型和标识符。对类型和标识符的处理如下：

1) 在获取类型后，如果类型是包装类型，则转化为对应的基本类型，例如包装类型Integer转化为基本数据类型int，表4-3列出了所有的包装类型及其对应的基本类型；

2) 在获取标识符后，首先将标识符中的下划线去除，然后将大写字母全部转化为小写字母，例如标识符gcGraceSeconds和gc\_grace\_seconds均转化为gcgraceseconds。

对类型和标识符这样处理后，可以通过比较表示类型或者标识符的字符串是否相同判断两个类型或者标识符是否相同。

表4-2 语法解析

|  |  |
| --- | --- |
| **修改类型** | **语法解析结果** |
| Additional Object State | 增加的属性类型和标识符 |
| Removed Object State | 删除的属性类型和标识符 |
| Attribute Renaming/ Type Change | 修改前后的属性类型和标识符 |
| Parameter Insert | 增加的参数类型和标识符 |
| Parameter Delete | 删除的参数类型和标识符 |
| Parameter Renaming/ Type Change | 修改前后的参数类型和标识符 |
| Return Type Insert | 增加的返回值类型 |
| Return Type Delete | 删除的返回值类型 |
| Return Type Change | 修改前后的返回值类型 |
| If Statement Insert | 增加的表达式树 |
| If Statement Delete | 删除的表达式树 |
| If Statement Condition Expression Change | 修改前后的表达式树 |

对于修改实体类型为If语句的修改操作，需要将涉及的条件表达式转化为表达式树，表达式树是一个二叉树，每个节点的content存储表达式的操作符(Operator)，左子节点和右子节点分别存储当前表达式的左操作数(Left Operand)和右操作数(Right Operand)。

表4-3 包装类型和对应的基本数据类型

|  |  |
| --- | --- |
| 包装类型 | 基本数据类型 |
| Byte | byte |
| Boolean | boolean |
| Short | short |
| Character | char |
| Integer | int |
| Long | long |
| Float | float |
| Double | double |

我们首先利用Eclipse提供的Java开发工具包(Java Development Tooling，JDT)将条件表达式解析为抽象语法树(Abstract Syntax Tree，AST)结构，然后将该表达式转化为二叉树，方便判断两个条件表达式是否相同。图4-3给出了将条件表达式转化为二叉树根节点的伪代码。首先初始化二叉树根节点，如果表达式的类型是中缀表达式(Infix Expression)，则提取操作符作为根节点的内容，接着利用递归的思想分别将该表达式的左操作数(Left Operand)和右操作数(Right Operand)作为输入，返回的节点分别作为根节点的左子节点和右子节点，如果表达式的类型不是中缀表达式(Infix Expression)，则根节点的内容用常量字符串填充，不再递归。最后返回根节点作为二叉树的根节点。

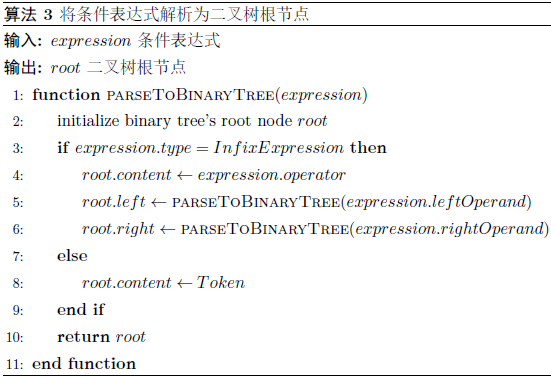


图4-3 将条件表达式解析为二叉树根节点的伪代码

### 4.3.3 修改操作判定规则

根据4.3.2节解析出的修改操作涉及的程序实体信息，可以判断两个修改操作是否相同。表4-4给出了判定规则，对修改实体类型为属性、参数和返回值的修改操作，如果表示类型或者标识符的字符串相同，则认为对应的类型或者标识符相同，否则认为不相同。例如，如果两个文件在同一次提交中分别发生了一个类型为增加属性(Additional Object State)的修改操作，并且这两个修改操作增加的属性类型和标识符相同，则认为这两个修改操作相同，也就是说这两个文件在同一次提交中发生了相同的修改。

表4-4 修改操作判定规则

|  |  |
| --- | --- |
| **修改操作类型** | **判定规则** |
| Additional Object State | 增加的属性类型和标识符相同 |
| Removed Object State | 删除的属性类型和标识符相同 |
| Attribute Renaming/ Type Change | 修改前后的属性类型和标识符相同 |
| Parameter Insert | 增加的参数类型和标识符相同 |
| Parameter Delete | 删除的参数类型和标识符相同 |
| Parameter Renaming/ Type Change | 修改前后的参数类型和标识符相同 |
| Return Type Insert | 增加的返回值类型相同 |
| Return Type Delete | 删除的返回值类型相同 |
| Return Type Change | 修改前后的返回值类型相同 |
| If Statement Insert | 增加的表达式树相同 |
| If Statement Delete | 删除的表达式树相同 |
| If Statement Change | 修改前后的表达式树相同 |

对于修改实体类型为If语句的修改操作，需要判断涉及的条件表达式对应的二叉树是否相同，图4-4描述了判断两个二叉树是否相同的伪代码。算法输入是两个二叉树的根节点和，返回值是或，表示两个二叉树是否相同。首先判断和的是否相同，若相同，则继续分别判断和的左子节点是否相同，以及它们的右子节点是否相同，若均相同返回，其余情况，均返回。

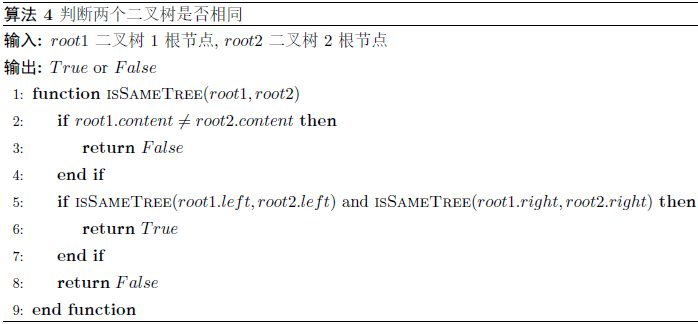


图4-4 判断二叉树是否相同的伪代码

## 4.4 分析方法

通过以上的步骤，我们可以得到平行共同修改关系的数据，然后结合共同修改提交、共同修改文件对和共同修改关系的数据进行经验研究，本节主要介绍经验研究的分析方法，包括统计分析、案例分析和演化分析。

统计分析主要包括对共同修改提交、共同修改文件对、共同修改关系和平行共同修改关系的在不同软件项目中的统计结果进行分析。通过统计结果，可以分析出软件项目中共同修改现象的基本情况和结论，包括有多少比例的提交是共同修改提交，存在多少共同修改文件对，哪些文件对存在共同修改关系，其中又有多少文件对存在平行共同修改关系，重点分析平行共同修改关系在不同项目中的存在和分布情况。

统计分析主要从宏观角度分析平行共同修改关系在不同软件项目中的存在和分布情况。在统计分析的基础上，我们结合典型案例，从微观角度分析不同的平行共同修改关系背后是否体现了文件间的隐式共享依赖，以及这样的依赖是否可以通过涉及的文件间的结构关系体现，接着分析其中是否存在设计问题，对项目维护造成了怎样的影响，最后结合具体情况给出了改进方案。

此外，本文还通过可视化手段对共同修改文件对的平行共同修改关系随时间的演化情况进行分析，总结了一些典型的演化趋势。例如，其中一些共同修改文件对，不同维度的平行共同修改关系在演化历史中交替出现；另外一些共同修改文件对，一个维度的平行共同修改关系支配了整个演化历史；还有一些共同修改文件对，平行共同修改关系随着时间推移消失了。

图4-5给出了Camel项目(本文经验研究项目之一)中，一个共同修改文件对平行共同修改关系演化的例子。我们用时间轴模拟一个共同修改文件对包含的两个文件间，平行共同修改关系的演化历史，时间轴的一个刻度表示一次共同修改提交，从左到右分别表示时间的远近，刻度数等于文件对共同修改次数。我们用红色、黄色、蓝色和绿色分别表示属性、参数、返回值和条件表达式维度的平行共同修改关系。如果一个刻度的切片中出现某个颜色，则表示对应的提交中出现了对应维度的平行共同修改关系，如果一次提交中不存在任何平行共同修改关系，则该提交对应的刻度的切片为灰色。从图中可以看到，SplitDefinition类和MulticastProcessor类总计共同修改了16次，存在属性和条件表达式维度的平行共同修改关系，关系随时间的演化趋势如图所示。

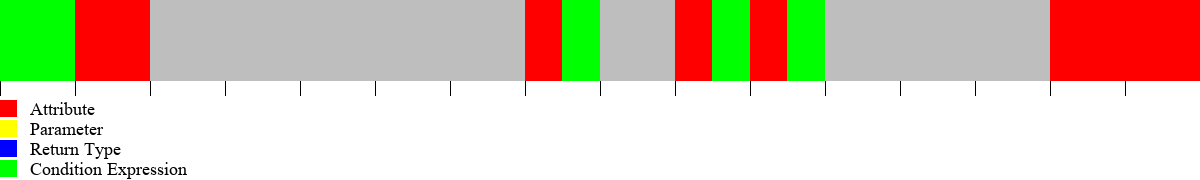


图4-5 平行共同修改关系演化例子

## 4.5 本章小结

本章首先定义了平行共同修改关系，然后介绍了如何在共同修改关系的基础上如何挖掘平行共同修改关系，最后介绍了经验研究分析方法，旨在分析得到软件项目共同修改现象的基本结论，包括平行共同修改关系在软件项目中的存在和分布情况，此外结合典型案例分析平行共同修改关系背后的隐式共享依赖，通过可视化手段分析文件间平行共同修改关系的演化情况。

# 第五章 实验与经验研究

本章主要介绍热点检测的实验和隐式共享依赖的经验研究，首先介绍研究项目，然后介绍在这些项目上的实验和经验研究。

## 5.1 研究项目

我们选取了阿帕奇软件基金会(Apache Software Foudndation)的6个软件项目作为研究对象进行实验和经验研究。选择这些项目的原因是因为这些项目均采用了版本控制系统Git对软件项目进行版本管理，记录了相对完整的提交历史，提交日志相对规范，这些项目还通过JIRA进行事务管理，有比较详细的事务描述和记录。阿帕奇软件基金会是一个非盈利性、去中心化的社区组织，它所支持的项目都遵循阿帕奇许可证(Apache License)，鼓励免费和开源软件(Free and Open-Source Software, FOSS)[31]。Git是一个开源分布式版本控制系统，用于管理、追溯文件的版本，便于协调多人合作，主要用于软件开发中的源代码管理[32]。JIRA是Atlassian公司开发的事务管理工具，提供了缺陷跟踪、事务跟踪和项目管理功能[33]。

本文选用了不同应用领域的项目，所有项目均采用Java语言编写。Camel是一个基于企业集成模式的通用开源集成框架，允许用户定义各类特定领域语言(例如基于Java的Fluent API、Spring、Blueprint XML配置文件和Scala DSL)的路由和中介规则，支持Bean绑定，与多个流行框架(例如CDI、Spring、Blueprint和Guice)无缝集成[34]。Cassandra是一个可以提供高扩展性和高可用性的分布式数据库，支持跨多个数据中心备份数据的功能，具有低延迟，能在区域性断电情况下恢复的优势[35]。CXF是一个开源服务框架，可以帮助用户使用前端编程API(例如JAX-WS和JAX-RS)构建和开发服务。这些服务支持多种协议(例如SOAP、XML / HTTP、RESTful HTTP和CORBA)，并且可以在多种传输协议(例如HTTP、JMS和JBI)下工作[36]。Hadoop是一个分布式框架，允许用户在不了解分布式底层细节的情况下，使用简单的编程模型对跨集群的大数据集进行高效的分布式处理，包括四个模块：Hadoop Common(为其他模块提供通用功能的模块)、HDFS(提供对应用程序数据高吞吐量访问的分布式文件系统)、YARN(作业调度和集群资源管理的框架)和MapReduce(基于YARN的大型数据集并行处理系统)[37]。HBase是Hadoop框架的数据库，提供分布式，可扩展的大数据存储功能，运行在Hadoop HDFS之上，为Hadoop提供类似Bigtable的功能，提供了一种容错方式来存储大量的稀疏数据[38]。Wicket是一个开源的、面向组件的、服务器端的Java Web应用程序框架，仅使用Java和HTML提供可维护的、安全的和可扩展的Web应用[39]。

表5-1描述了这些项目的基本信息，包括分析的时间段(Time)、时间段内的提交个数(#Commit)、发布版本个数(#Release)、分析的源文件个数(#File)，即项目当前快照后缀名为.java的非测试文件的个数以及缺陷个数(#BugIssue)，我们利用爬虫技术手段从JIRA网站上爬取了事务记录，统计了其中类型为缺陷(Bug)的事务个数作为缺陷个数。本文分析的这些项目在其他演化耦合相关的研究中也被广泛地作为研究对象[4][5][24][25]。

表5-1 研究项目

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Project** | **Time** | **#Commit** | **#Release** | **#File** | **#BugIssue** |
| Camel | 03/2007-06/2017 | 28,881 | 113 | 6,697 | 3,874 |
| Cassandra | 03/2009-06/2017 | 23,082 | 225 | 1,593 | 7,418 |
| CXF | 04/2008-06/2017 | 13,180 | 123 | 3,371 | 4,644 |
| Hadoop | 05/2009-06/2017 | 16,119 | 247 | 5,162 | 18,772 |
| HBase | 04/2007-06/2017 | 13,577 | 551 | 2,218 | 9,029 |
| Wicket | 09/2004-06/2017 | 19,989 | 244 | 1,822 | 3,913 |

这些项目都有较长的演化历史(8年到13年)，数以万计的提交，上百个发布版本，数千个缺陷。在这些项目中，Camel的提交最多，为28,881次，CXF的提交最少，为13,180次；HBase的发布版本最多，为551次，Camel的发布版本最少，为113次；Camel分析的源文件个数最多，为6,697个，Cassandra分析的源文件个数最少，为1,593个；Hadoop记录缺陷个数最多，为18,772个，Camel记录的缺陷个数最少，为3,874个。

## 5.2 热点检测实验

本节介绍热点检测实验，首先介绍研究问题，然后介绍实验设计，最后分析实验结果。

### 5.2.1 研究问题

本实验旨在回答以下两个问题：

**问题一：检测出的热点文件(至少出现在一个热点中的文件)是否比项目中的其他文件的维护开销更高，核心文件是否比其他热点文件的维护开销更高？**

本文检测出的热点是一组存在高度演化耦合的文件，高度的演化耦合往往是由于不良设计没有及时得到改进而逐渐形成的，热点文件在演化历史中，往往会产生较大的维护开销，此外，核心文件是热点中耦合最严重的文件，产生的维护开销往往更高。

本实验旨在证明本文方法检测出的热点文件确实产生了比其他文件更高的维护开销，并且核心文件的开销比热点中的其他文件更高。开发人员应该及时发现这样的热点，结合热点的核心文件和层次结构等信息，对涉及的文件进行解耦和重构，改善其中的不良设计，提高软件质量，避免热点进一步扩大，乃至形成技术债务，带来越来越高的维护开销。

**问题二：检测出的热点文件是否比Mo R等人的方法检测出的热点文件更加聚焦？**

Mo R等人也提出方法通过对共同修改信息的分析检测项目中的热点[24]，Mo R定义的热点是一组存在架构问题的文件。然而他们的方法仅仅计算两个文件共同修改次数，对共同修改信息的挖掘不够深入，并且对热点的形式化定义和检测很大程度上仍然依赖于结构关系，因此导致检测出的热点文件范围过大，不够聚焦，无法对开发人员提供更加针对性的指导和帮助。

本实验旨在证明本文方法通过挖掘共同修改关系，根据共同修改关系计算文件间距离进行层次聚类，检测出的热点更为聚焦。也就是说，本文检测出的热点文件个数更少，历史上产生的维护开销更高。

### 5.2.2 实验设计

本实验采取了以下阈值在这6个项目上检测热点：=30，这些工作均采取30作为阈值过滤掉修改了大量文件的提交[1][7][18]； = 6， = 3，保证了两个文件一定的共同修改次数和共同修改类型的出现次数； = 64，构成的频繁关系集中共同修改关系出现次数占所有共同修改关系出现次数的比例在6个项目中分别为67.54%、36.26%、44.81%、45.83%、32.15%和54.77%，具有一般性； = 5， = 0.2，保证了热点的规模不至于太小，核心文件有较大的影响力。

此外，我们实现了Mo R等人提出的热点检测方法，在相同的发布版本上对项目快照中的文件进行热点检测，通过比较本文方法和Mo R方法的检测结果，可以回答问题二。

为了比较本文方法检测出的热点文件和项目中其他文件的维护开销，以及比较本文方法和Mo R等人方法检测出的热点文件的维护开销，需要定义评价指标来量化维护开销。Mo R等人从易出错性(Error-proneness)和易修改性(Change-proneness)两个方面分别定义了两个文件级别的评价指标来量化维护开销[24]，因为本文在共同修改文件对的识别过程中已经引入阈值，过滤掉了共同修改次数较少的文件对，所以识别出的共同修改文件对中的文件，修改次数相对其他文件往往更多，为了保证客观性，本实验中只选用易出错性方面的两个评价指标来量化维护开销。

量化一个文件的维护开销的评价指标包括：

1) 缺陷频数(Bug Frequency，BF)：目的是修复缺陷的提交个数。

2) 缺陷修改行数(Bug Churn，BC)：目的是修复缺陷的提交中修改代码行数。

对于一组文件，其中文件的缺陷频数和缺陷修改行数分别为和，这组文件的平均缺陷频数和平均缺陷修改行数为：

1) 平均缺陷频数(Average Bug Frequency，AvgBF):

2) 平均缺陷修改行数(Average Bug Churn，AvgBC):

表5-2 Camel项目事务例子

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Project** | **Issue ID** | **Summary** | **Assignee** | **Reporter** | **Status** |
| Camel | CAMEL-1000 | Trace interceptor does not work with Spring-event component | Hadrian Zbarcea | Charles Moulliard | Closed |

为了计算这些评价指标，首先需要分析项目的事务列表和提交历史，得到提交和类型是缺陷的事务的关联关系。本文的6个项目均采用JIRA进行项目管理，JIRA中记录了完整的事务历史，我们利用爬虫技术手段爬取了JIRA网站上这6个项目的事务列表，选出其中类型是缺陷的事务。每个事务有事务ID(Issue ID)和描述(Summary)等属性。表5-2给出了一个事务的例子，这个事务属于Camel项目，事务ID(Issue ID)为CAMEL-1000，描述信息(Summary)是Spring事件组件中Trace拦截器不工作，报告人(Reporter)是Charles Moulliard，被分配人(Assignee)是Hadrian Zbarcea，当前状态(Status)是Closed。

本文实验项目的提交日志相对规范，日志中会记录本次提交涉及事务的事务 ID。表5-3是一个提交的例子。该提交的提交ID(Commit ID)为7f8456a，提交日志(Message)中记录了关联事务的Issue ID为CAMEL-11435，提交人(Committer)是Andrea Cosentino，提交时间(Committed Date)是2017-06-22 17:46:18。

表5-3 Camel项目提交例子

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Project** | **Commit ID** | **Message** | **Committer** | **Committed Date** |
| Camel | 7f8456a | CAMEL-11435 - Fix defaultValue for EvictionType to make it work with camel-caffeine-starter | Andrea Cosentino | 2017-06-22 17:46:18 |

首先我们遍历提交历史，通过正则式匹配的方法得到每次提交日志中的事务 ID，与类型是缺陷的事务的ID进行字符串匹配，可以分析出提交和类型是缺陷的事务的关联关系，接着我们通过Git命令可以得到每次提交修改的文件(git show <commit>)以及每个文件修改的代码行数(git diff <commit> <filename>)，根据这些信息可以计算出每个文件的缺陷频数和缺陷修改行数，有了这些数据，我们可以计算得到指定的一组文件的平均缺陷频数和平均缺陷修改行数。

通过比较本文方法检测出的热点文件和项目当前快照中其他文件的平均缺陷频数和平均缺陷修改行数，比较核心文件和其他热点文件的平均缺陷频数和平均缺陷修改行数，可以回答问题1。

通过比较本文方法检测出的热点文件和Mo R提出的方法检测出的热点文件的平均缺陷频数和平均缺陷修改行数，可以回答问题2。

### 5.2.3 实验结果与分析

表5-4、5-5、5-6和5-7展示了实验结果，其中表5-4和5-5的数据回答了问题1，表5-6和5-7的数据回答了问题2。

表5-4展示了项目当前快照中文件和本文方法检测出的热点文件的评价指标。第一列是项目名(Project)，第二列和第三列分别是项目当前快照中文件的平均缺陷频数(AvgBF)和平均缺陷修改行数(AvgBC)，第四列和第五列分别是本文方法检测出的热点文件的平均缺陷频数(AvgBF\*)和平均缺陷修改行数(AvgBC\*)。从表中的数据可以看到，在所有项目上，本文方法检测出的热点文件的平均缺陷频数和平均缺陷修改行数均远远高于项目当前快照中文件的平均缺陷频数和平均缺陷修改行数。这表明本文方法检测出的热点文件比项目中的其他文件的维护开销更高。

表5-4 项目快照文件和本文方法热点文件的评价指标

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Project** | **AvgBF** | **AvgBC** | **AvgBF\*** | **AvgBC\*** |
| Camel | 0.99 | 15.42 | 7.91 | 136.02 |
| Cassandra | 1.56 | 42.48 | 13.74 | 383.62 |
| CXF | 1.47 | 25.90 | 8.61 | 167.49 |
| Hadoop | 2.18 | 42.73 | 14.39 | 295.44 |
| HBase | 4.47 | 298.94 | 21.17 | 1,463.55 |
| Wicket | 1.81 | 46.86 | 9.10 | 222.19 |

表5-5展示了本文方法检测出的热点中核心文件和热点中其他文件的评价指标。第一列是项目名(Project)，第二列和第三列分别是本文方法检测出的热点中核心文件的平均缺陷频数(AvgBF)和平均缺陷修改行数(AvgBC)，第四列和第五列分别是本文方法检测出的热点中其他文件的平均缺陷频数(AvgBF\*)和平均缺陷修改行数(AvgBC\*)。从表中的数据可以看到，在所有项目上，本文方法检测出的热点中核心文件的平均缺陷频数和平均缺陷修改行数均高于本文方法检测出的热点中其他文件的平均缺陷频数和平均缺陷修改行数。这表明本文方法检测出的热点中的核心文件比热点中的其他文件的维护开销更高。

表5-5 本文方法热点核心文件和其他文件的评价指标

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Project** | **AvgBF** | **AvgBC** | **AvgBF\*** | **AvgBC\*** |
| Camel | 14.19 | 235.94 | 6.52 | 113.99 |
| Cassandra | 49.75 | 1,705.20 | 8.11 | 177.13 |
| CXF | 14.91 | 271.06 | 7.38 | 147.26 |
| Hadoop | 30.88 | 689.01 | 10.93 | 213.04 |
| HBase | 46.12 | 2,131.98 | 15.44 | 1,309.92 |
| Wicket | 17.05 | 358.36 | 7.78 | 199.67 |

表5-6展示了Mo R方法和本文方法检测出的热点文件的评价指标。第一列是项目名(Project)，第二列和第三列分别是Mo R方法检测出的热点文件的平均缺陷频数(AvgBF)和平均缺陷修改行数(AvgBC)，第四列和第五列分别是本文方法检测出的热点文件的平均缺陷频数(AvgBF\*)和平均缺陷修改行数(AvgBC\*)。从表中的数据可以看到，在所有项目上，本文方法检测出的热点文件的平均缺陷频数和平均缺陷修改行数均远远高于Mo R方法检测出的热点文件的平均缺陷频数和平均缺陷修改行数。这表明本文方法检测出的热点文件比Mo R检测出的热点文件的平均维护开销更高。

表5-6 Mo R方法热点文件和本文方法热点文件的评价指标

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Project** | **AvgBF** | **AvgBC** | **AvgBF\*** | **AvgBC\*** |
| Camel | 1.95 | 32.91 | 7.91 | 136.02 |
| Cassandra | 2.19 | 60.23 | 13.74 | 383.62 |
| CXF | 3.10 | 55.37 | 8.61 | 167.49 |
| Hadoop | 4.43 | 92.27 | 14.39 | 295.44 |
| HBase | 8.08 | 411.17 | 21.17 | 1,463.55 |
| Wicket | 3.06 | 78.90 | 9.10 | 222.19 |

表5-7展示了本文方法和Mo R方法检测出的热点文件的范围。第一列是项目名(Project)，第二列是项目当前快照中分析的源文件个数(Snapshot)，第三列和第四列分别是Mo R方法检测出的热点文件个数(Mo R方法)以及在所有文件中的占比(Ratio)，第五列和第六列分别是本文方法检测出的热点文件个数(本文方法)以及在所有文件中的占比(Ratio\*)。从表中的数据可以看出，Mo R方法检测出的热点文件范围过大，在总文件中的占比为35.86%至69.30%，而本文方法检测出的热点文件则远远小于Mo R方法检测出的热点文件在总文件中的占比，仅为5.70%至13.75%。

表5-7 Mo R方法热点文件和本文方法热点文件个数和占比

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Project** | **Snapshot** | **Mo R方法** | **Ratio** | **本文方法** | **Ratio\*** |
| Camel | 6,697 | 2,539 | 37.91% | 382 | 5.70% |
| Cassandra | 1,593 | 1,104 | 69.30% | 148 | 9.29% |
| CXF | 3,371 | 1,209 | 35.86% | 202 | 5.99% |
| Hadoop | 5,162 | 2,019 | 39.11% | 439 | 8.50% |
| HBase | 2,218 | 971 | 43.78% | 305 | 13.75% |
| Wicket | 1,822 | 871 | 47.80% | 155 | 8.51% |

综合表5-6和5-7可以发现，本文方法不仅检测出的热点文件个数远远少于Mo R方法检测出的热点文件个数，而且本文方法检测出的热点文件的平均缺陷频数和平均缺陷修改行数也远远高于Mo R方法检测出的热点文件的平均缺陷频数和平均缺陷修改行数。这表明本文方法检测出的热点文件比Mo R方法检测出的热点文件更为聚焦。

## 5.3 隐式共享依赖经验研究

本节主要介绍在这6个项目上的经验研究，包括三个部分，分别是统计分析、案例分析和演化分析。

### 5.3.1 统计分析

本节通过统计结果对共同修改提交、共同修改文件对、共同修改关系和平行共同修改关系进行分析，得到了一些基本结论。

表5-8列出了共同修改提交相关数据。第一列是项目名(Project)，第二列是共同修改提交个数(#CCo.)，第三列是共同修改提交在所有提交中的比例(#CCo./#Co.)，第四列和第五列是共同修改提交修改文件个数的平均数(#File(Mean))和中位数(#File(Median))。从表中的数据可以看到，每个项目都有几千个共同修改提交，占所有提交个数的比例为21.32%至34.68%，平均每次提交修改4到5个文件，超过一半的共同修改提交只修改不超过3个文件。

表5-8 共同修改提交

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Project** | **#CCo.** | **#CCo./#Co.** | **#File(Mean)** | **#File(Median)** |
| Camel | 8,278 | 28.66% | 4.44 | 3 |
| Cassandra | 4,920 | 21.32% | 5.23 | 3 |
| CXF | 4,045 | 30.69% | 4.39 | 3 |
| Hadoop | 5,590 | 34.68% | 5.24 | 3 |
| HBase | 4,230 | 31.16% | 5.57 | 3 |
| Wicket | 4,663 | 23.33% | 4.81 | 3 |

表5-9列出了共同修改文件对的相关数据。第一列是项目名(Project)，第二列是所有共同修改文件对的个数(#Pair)，第三列和第四列分别是共同修改文件对的修改时间跨度(第一次共同修改和最后一次共同修改相差的月数)的平均数和(Mean Span)中位数(Median Span)。从表中的数据可以看到，除了CXF项目，其他所有项目都有上千个共同修改文件对，这些文件对平均修改时间跨度都在几十个月，范围从34.18至57.53，除了Hadoop项目，其他项目共同修改文件对共同修改时间跨度的中位数都是几十个月。这些数据说明，项目中相当数量的文件间存在共同修改现象，并且共同修改的时间跨度较长。

表5-9 共同修改文件对

|  |  |  |  |
| --- | --- | --- | --- |
| **Project** | **#Pair** | **Mean Span** | **Median Span** |
| Camel | 1,981 | 53.12 | 31.00 |
| Cassandra | 2,855 | 34.18 | 42.00 |
| CXF | 833 | 45.61 | 41.00 |
| Hadoop | 3,079 | 38.14 | 7.00 |
| HBase | 2,846 | 38.20 | 45.00 |
| Wicket | 1,512 | 57.53 | 90.00 |

表5-10列出了平行共同修改关系的数据。其中第一列是项目名(Project)，第二列和第三列分别是存在共同修改关系的文件对的个数(#PairCR)以及在所有共同修改文件对中的比例(#PairCR/#Pair)，第四列和第五列分别是存在平行共同修改关系的文件对的个数(#PairPCR)以及在所有存在共同修改关系的文件对中的比例(#PairPCR/PairCR)。从表中的数据可以看到，每个项目都有46.49%至68.19%的共同修改文件对存在共同修改关系，在这些文件对里，又有14.05%至29.05%的文件对存在平行共同修改关系。也就是说，在这些项目中，共同修改关系和平行共同修改关系是普遍存在的。

表5-10 平行共同修改关系

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Project** | **#PairCR** | **#PairCR/#Pair** | **#PairPCR** | **#PairPCR/PairCR** |
| Camel | 1,046 | 52.80% | 222 | 21.22% |
| Cassandra | 1,729 | 60.56% | 243 | 14.05% |
| CXF | 568 | 68.19% | 165 | 29.05% |
| Hadoop | 2,034 | 66.06% | 381 | 18.73% |
| HBase | 1,890 | 66.41% | 442 | 23.39% |
| Wicket | 703 | 46.49% | 144 | 20.48% |

表5-11展示了四个维度的平行共同修改关系的分布情况。第一列是项目名(Project)，第二列、第三列、第四列和第五列是四个维度的平行共同修改关系出现次数的分布情况。从表中的数据可以看到，条件表达式维度的平行共同修改关系占比最高，返回值维度的平行共同修改关系占比最低，甚至在CXF项目中不存在这样的关系。在Camel和HBase项目中，属性维度的平行共同修改关系比参数维度的平行共同修改关系占比高，在其他项目中，参数维度的平行共同修改关系则比属性维度的平行共同修改关系占比高。这表明本文定义的平行共同修改关系在各个项目中基本都存在并且有不同的分布情况。

表 5-11 平行共同修改关系分布

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Project\维度** | **属性** | **参数** | **返回值** | **条件表达式** |
| Camel | 29.24% | 8.76% | 1.81% | 60.19% |
| Cassandra | 9.72% | 49.35% | 1.89% | 39.04% |
| CXF | 7.90% | 15.96% | — | 76.14% |
| Hadoop | 6.06% | 52.84% | 10.74% | 30.36% |
| HBase | 24.45% | 23.43% | 2.70% | 49.42% |
| Wicket | 1.95% | 36.28% | 5.40% | 56.37% |

### 5.3.2 案例分析

本节结合典型案例对四个维度的平行共同修改关系进行分析，旨在揭示和分析关系背后体现的隐式共享依赖。

**1) 属性维度的平行共同修改关系典型案例**

在Cassandra项目中，存在这样的共同修改文件对，包括的两个文件分别为CFMetaData类和RawColumnFamily类，这两个文件间不存在任何结构关系。通过分析它们间属性维度的平行共同修改关系可以发现，它们在历史上一共发生5次相同的修改操作，都是增加了属性。这表明这两个文件间存在这样的隐式共享依赖，即共享了许多属性，并且由于开发人员没有及时发现这个问题，对这两个文件进行解耦，导致它们共享的属性随着演化历史越来越多，属性耦合越来越严重，此外RawColumnFamily类逐渐发展成为了数据类(Data Class)。数据类指的是仅仅拥有一些属性，以及用于访问或者修改这些属性的方法，除此之外不提供任何功能性方法的类，这样的类仅仅充当了数据容器的角色。由于数据类往往拥有的属性往往非常多，甚至有的属性本不应该设计在同一个类中，除非是项目早期核心人员，否则难以理解每个属性背后的含义，也导致了对数据类的维护难度往往比较高。开发人员应该及时注意到这两个文件间的隐式共享依赖，对其中的不良设计进行改进，首先根据可见性范围对RawColumnFamily类中public修饰符修饰的属性进行封装，提供访问和修改的方法，然后定位所有其他类对这些属性的直接访问和修改，将访问和修改方式改为对属性访问和修改方法的调用，最后对这些共享属性进行拆分合并，搬移到合适的位置，保证其他类对同一个属性的访问方式一致。

在Camel项目中，存在这样的共同修改文件对，包括的两个文件分别为StreamInSession类和StreamOutSession类，这两个类继承同一个父类AbstractStreamSession类。通过分析它们间属性维度的平行共同修改关系可以发现，它们在历史上一共发生了10次相同的修改操作，其中6次增加了属性，4次删除了属性。这表明这两个文件间存在这样的隐式共享依赖，即共享了越来越多的属性，当需要删除属性时，需要在两个文件中分别删除，体现的设计问题是父类的抽象不够彻底，违背了面向对象的设计原则，且一定程度上造成了代码冗余，甚至产生了重复代码(Duplicate Code)的坏味道。开发人员应当及时对涉及的类重新设计，将子类中的共同部分提取到父类中，由子类继承父类的属性和方法，避免在子类中出现重复的属性和方法。

通过这两个案例可以发现，两个文件间属性维度的平行共同修改关系一定程度上可以揭示共享属性的隐式依赖，这样的隐式共享依赖不仅存在于父类和子类或者接口和实现类之间，也存在于没有结构关系的两个文件间。开发人员如果不及时发现这个问题并结合具体情况改进其中的不良设计，共享的属性会越来越多，文件间的耦合会越来越严重，不利于项目后续的维护。

**2) 参数维度的平行共同修改关系典型案例**

在Camel项目中，存在这样的共同修改文件对，包括的两个文件分别为MulticastProcessor类和Splitter类，这两个文件间不存在任何结构关系。通过分析它们间参数维度的平行共同修改关系可以发现，它们在历史上一共发生了10次相同的修改操作，其中6次增加了参数，4次修改了参数类型。这表明这两个文件间存在这样的隐式共享依赖，即共享了部分参数。共享的参数随着演化历史越来越多，当依赖的外界参数类型发生变化时，两个文件都需要同步修改参数类型。这样的隐式共享依赖无法通过结构关系发现，当方法契约发生变化时(依赖的外部参数类型发生变化)，往往需要修改分布在不同文件中的多个存在该参数的方法。开发人员应该及时发现这样的问题，在方法契约中明确声明隐式的共享参数依赖，方便其他开发人员理解和维护。当这样的依赖越来越多时，开发人员需要通过重构改进设计，将共享的参数提取出来，封装成一个对象，通过传递此对象来达到传递多个值的目的，降低维护开销。

在Cassandra项目中，存在这样的共同修改文件对，包括的两个文件分别为StorageServiceMBean接口和StorageService类，这两个文件间的结构关系是StorageService类实现了StorageServiceMBean接口。通过分析它们间参数维度的平行共同修改关系可以发现，它们在历史上一共发生了17次相同的修改操作，其中13次增加了参数，4次删除了参数。这表明StorageServiceMBean是一个不稳定的接口，方法契约总是发生变化，实现了该接口的其他类需要进行同样的修改，导致对这些方法的调用也要随之修改，不稳定的接口被认为是形成技术债务的重要原因之一[4]。通过本案例，我们可以发现一个接口发展成为不稳定接口的过程，以及它的影响是如何通过实现类逐渐向外扩散的。开发人员应该及时注意到这样的问题，对不稳定接口进行解耦，避免同一个接口承担过多的职责，特别是避免一个接口发生频繁的变化。

通过这两个案例可以发现，两个文件间参数维度的平行共同修改关系一定程度上可以揭示共享参数的隐式依赖，这样的隐式共享依赖不仅存在于父类和子类或者接口和实现类之间，也存在于没有结构关系的两个文件间。结合具体情况，开发人员可以分析发现其中存在的不良设计，通过重构等手段进行改进，避免共享参数太多，或者形成不稳定接口，给项目维护带来困难。

**3) 返回值维度的平行共同修改关系典型案例**

在Hadoop项目中，存在这样的共同修改文件对，包括的两个文件分别是ClientNamenodeProtocolTranslatorPB类和NameNodeRpcServer类，这两个文件间不存在任何结构关系。通过分析它们间返回值维度的平行共同修改关系可以发现，它们在历史上一共发生了19次相同的修改操作，其中4次增加了返回值类型，15次修改了返回值类型。这表明这两个文件间存在这样的隐式共享依赖，即共享了部分返回值类型。这两个文件多次增加相同的返回值类型，导致当方法的返回值类型发生变化时，需要分别修改这两个文件中对应的返回值类型，返回值类型的变化还会继续影响到项目中其他调用了这些方法的文件。因为这两个文件不是子类与父类或者实现类和接口的关系，仅仅通过结构关系无法发现这个问题。开发人员通过分析平行共同修改关系可以及时发现这个问题，进一步分析方法契约(返回值类型)频繁变化的原因，改进其中的不良设计。

在Wicket项目中，存在这样的共同修改文件对，包括的两个文件分别是ISessionStore接口和HttpSessionStore类，这两个文件间的结构关系是HttpSessionStore类实现了ISessionStore接口。通过分析它们间返回值维度的平行共同修改关系可以发现，它们在历史上一共发生了4次相同的修改操作，都是修改返回值类型。这表明这两个文件间存在这样的隐式共享依赖，即共享了返回值类型。由于HttpSessionStore类实现了ISessionStore接口，因此当接口方法的契约发生变化(返回值类型变化)，HttpSessionStore类必然要进行同样的修改，但这个关系还反映了ISessionStore接口不稳定，它的变化引起了实现类的变化，继续影响到调用了实现类方法的文件。开发人员应该及时注意到这个问题，分析返回值频繁变化的原因进行改进，甚至采取契约式编程，明确后置条件，避免后置条件的频繁变化。

通过这两个案例可以发现，返回值维度的平行共同修改关系不仅存在于父类和子类或者接口和实现类之间，也存在于没有结构关系的文件间，这样的隐式共享依赖通常无法通过结构关系发现，然而通过平行共同修改关系，开发人员可以及时发现这样的依赖，有针对性地解决其中存在的设计问题。

**4) 条件表达式维度的平行共同修改关系典型案例**

在Hadoop项目中，存在这样的共同修改文件对，包括的两个文件分别是Server类和Client类，这两个文件间不存在任何结构关系。通过分析它们间条件表达式维度的平行共同修改关系可以发现，它们在历史上一共发生了8次相同的修改操作，其中6次增加了If条件表达式，2次修改了If条件表达式。这表明这两个文件间存在这样的隐式共享依赖，即共享了部分判断条件。虽然这两个文件间没有结构关系，但是通过分析源码和修改历史可以发现，这两个文件间的联系是Client类作为客户端会向服务器端，即Server类发送连接请求，而Server类接收请求后会将处理结果返回给Client类。这两个文件间存在消息耦合，并且对消息的处理具有高度一致性，造成了同样的判断条件分别出现在两个文件里，当消息的状态或者协议发生变化，客户端和服务器端的判断逻辑都需要同步修改。

这样的隐式依赖无法通过结构关系分析发现，而通过分析本文提出的平行共同修改关系可以及时发现这个问题，开发人员在后续的开发维护中，应该注意对消息的条件处理保持一致，否则会引入漏洞或缺陷，此外及时重构代码，避免共享过多的判断条件，不利于项目的维护。

在Wicket项目中，存在这样的共同修改文件对，包括的两个文件分别是Check类和Radio类，这两个文件间不存在结构关系，但是继承同一个父类LabeledWebMarkupContainer类。通过分析它们间条件表达式维度的平行共同修改关系可以发现，它们在历史上一共发生了19次相同的修改操作，其中9次插入了If条件表达式，6次删除了If条件表达式，4次修改了If条件表达式。这表明这两个文件间存在这样的隐式共享依赖，即共享了部分判断条件。通过分析源码和修改历史可以发现，这两个类间存在大量的克隆片段(Clone Fragment)，几乎所有的判断条件都一致，而由于没有及时注意到这里的设计问题，随着演化两个文件共享的判断条件越来越多，导致对同一个判断条件的修改需要在多处进行，这体现的设计问题是父类的抽象不够彻底，同样的代码逻辑分布在多个子类中，会给软件项目后续的开发维护带来额外的开销。开发人员应该及时注意到这个问题，重新设计涉及的类，将子类中重复的属性和方法抽象到父类中去，避免相同的代码片段分散在多个子类中。

通过这两个案例可以发现，条件表达式维度的平行共同修改关系不仅存在于继承或实现了同一个父类或接口的多个文件之间，也存在于两个没有结构关系的文件间。通过平行共同修改关系，开发人员可以及时发现这样的隐式共享依赖，结合具体情况分析发现造成条件耦合的原因，改善其中的不良设计。

### 5.3.3 演化分析

我们利用可视化手段，对文件间平行共同修改关系的演化趋势进行分析，总结了四类典型的演化趋势，并结合具体案例进行介绍。

**1) 一个维度的平行共同修改关系支配整个演化历史**

图5-1展示的是Cassandra项目中，CFMetaData类和RawColumnFamily类这两个文件组成的共同修改文件对间，平行共同修改关系的演化情况。从图中可以看到，这两个文件间只存在属性维度的平行共同修改关系，并且出现在大多数共同修改提交中，该关系主导了演化历史。

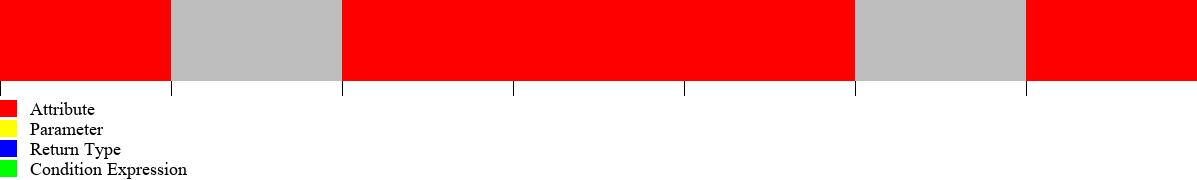
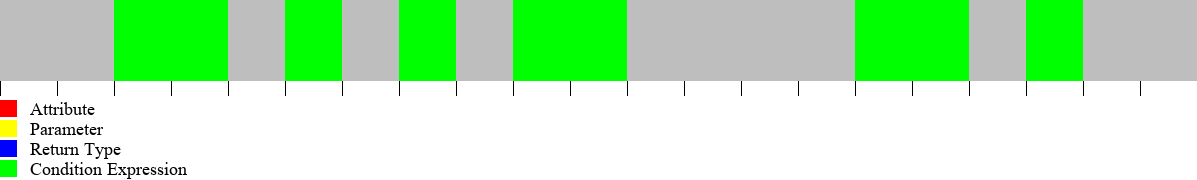


图5-1 一个维度的平行共同修改关系支配整个演化历史例子一

图5-2展示的是HBase项目中，HClient类和HMaster类这两个文件组成的共同修改文件对间，平行共同修改关系的演化情况。从图中可以看到，这两个文件间只存在条件表达式维度的平行共同修改关系，并且出现在大多数共同修改提交中，该关系主导了演化历史。

图5-2 一个维度的平行共同修改关系支配整个演化历史例子二

这两个例子体现的演化趋势是一个维度的平行共同修改关系支配了整个演化历史，说明涉及的两个文件间平行共同修改关系比较单一，并且存在于大多数的共同修改提交中，这样的关系是造成这两个文件共同修改的重要原因之一。

**2) 多个维度的平行共同修改关系交叉出现**

图5-3展示的是Camel项目中，AbstractXStreamWrapper 类和XStreamDataFormat类这两个文件组成的共同修改文件对间，平行共同修改关系的演化情况。从图中可以看到，这两个文件间存在属性和条件表达式维度的平行共同修改关系，并且交叉出现在大多数共同修改提交中。

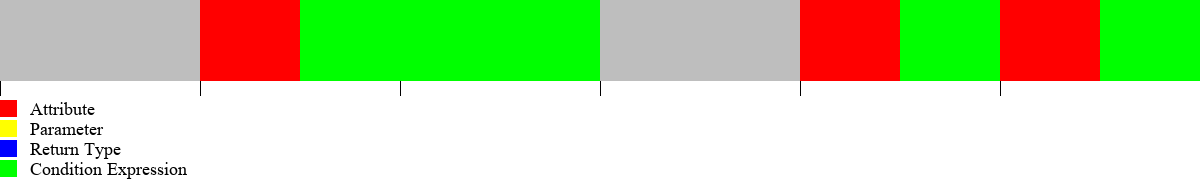


图5-3 多个维度的平行共同修改关系交叉出现例子一

图5-4展示的是Hadoop项目中，ClientProtocal类和ClientNamenodeProtocalTransalatorPB类这两个文件组成的共同修改文件对间，平行共同修改关系的演化情况。从图中可以看到，这两个文件间存在参数和返回值维度的平行共同修改关系，并且交叉出现在大多数共同修改提交中。

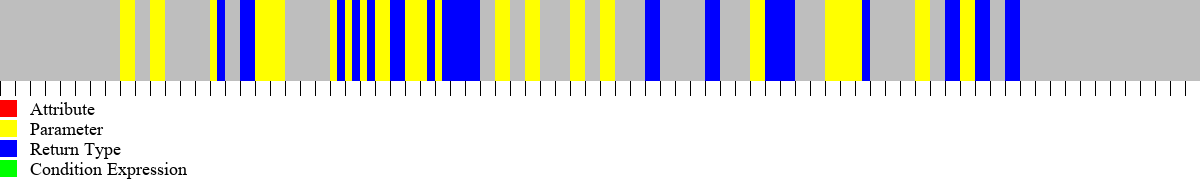


图5-4 多个维度的平行共同修改关系交叉出现例子二

这两个例子体现的演化趋势是多个维度的平行共同修改关系交叉出现，说明涉及的两个文件间存在多种平行共同修改关系，这些关系在演化历史中处于均等的地位，都是造成这两个文件频繁共同修改的原因。

**3) 平行共同修改关系存在活跃与不活跃阶段**

图5-5展示的是Camel项目中，ManagedCamelContext类和ManagedRoute类这两个文件组成的共同修改文件对间，平行共同修改关系的演化情况。从图中可以看到，这两个文件间只存在参数维度的平行共同修改关系，并且只出现在少数共同修改提交中，存在明显的活跃和不活跃阶段。

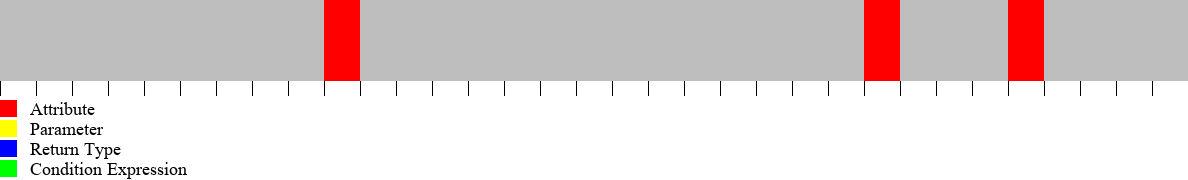


图5-5 平行共同修改关系存在活跃与不活跃阶段例子一

图5-6展示的是Wicket项目中，Component类和Page类这两个文件组成的共同修改文件对间，平行共同修改关系的演化情况。从图中可以看到，这两个文件间只存在条件表达式维度的平行共同修改关系，并且只出现在少数共同修改提交中，存在明显的活跃和不活跃阶段。

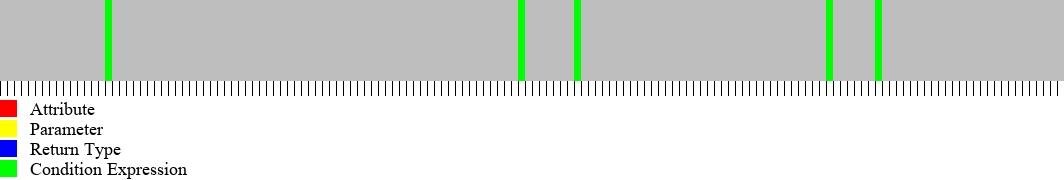


图5-6 平行共同修改关系存在活跃与不活跃阶段例子二

这两个例子体现的演化趋势是平行共同修改关系存在活跃与不活跃阶段，在某些阶段不活跃的关系并不是完全消失了，在其他阶段又会重新活跃起来。

**4) 平行共同修改关系消失**

图5-7展示的是Cassandra项目中，RackUnwareStratagy类和TokenMetadata类这两个文件组成的共同修改文件对间，平行共同修改关系的演化情况。从图中可以看到，这两个文件间只存在参数维度的平行共同修改关系，并且只出现在早期的共同修改提交中，在后面的共同修改提交中不再出现。

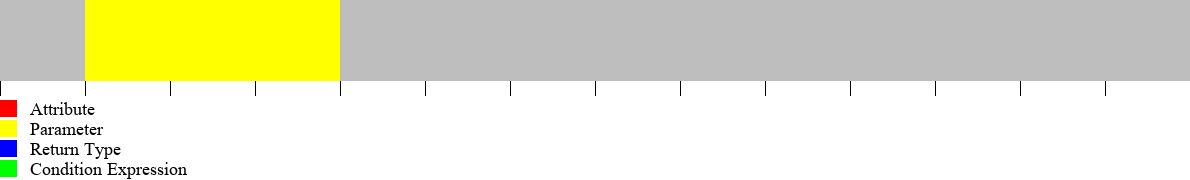


图5-7 平行共同修改关系消失例子一

图5-8展示的是CXF项目中，AsymmetricBindingHandler类和SymmetricBindingHandler类这两个文件组成的共同修改文件对间，平行共同修改关系的演化情况。从图中可以看到，这两个文件间只存在条件表达式维度的平行共同修改关系，并且只出现在前半段的共同修改提交中，在后面的共同修改提交中不再出现。

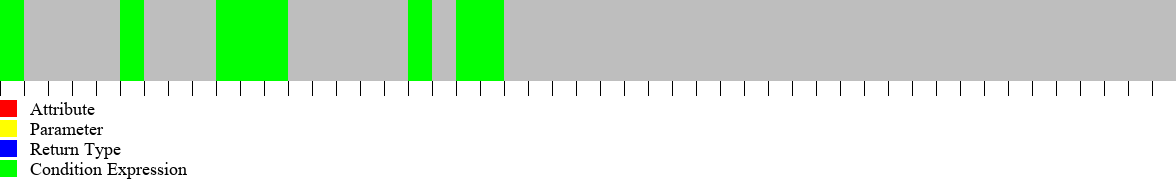


图5-8 平行共同修改关系消失例子二

这两个例子体现的演化趋势是平行共同修改关系消失，说明涉及的两个文件间平行共同修改关系趋向稳定不再出现，但是文件间仍然存在共同修改现象，关系背后的依赖是否消失需要结合具体情况分析，也有可能关系只是暂时不活跃，在未来的演化中又会重新活跃起来。

## 5.4 本章小结

本章主要介绍了热点检测的实验和隐式共享依赖的经验研究。实验结果不仅表明了本文方法检测出的热点文件比项目中其他文件造成的维护开销更高，而且表明了本文方法比Mo R等人提出的方法检测出的热点更加聚焦，即检测出的热点文件范围更小，但平均维护开销更高。经验研究部分，通过统计分析我们得到了关于项目共同修改现象的基本结论，包括平行共同修改关系在项目中的存在和分布情况，案例分析总结了不同维度的平行共同修改关系所体现的隐式共享依赖，特别是无法通过结构关系发现的依赖，并给出了具体的改进建议，通过演化分析我们总结了平行共同修改关系四类典型的演化趋势。

# 第六章 总结和展望

## 6.1 总结

针对现有工作对共同修改信息挖掘不够深入的现象，本文提出了共同修改关系，共同修改关系是两个文件间多次出现的共同修改类型，一定程度上体现了文件间的隐式依赖，本文根据共同修改关系计算出文件间距离，对项目当前快照中的文件进行层次聚类，根据启发式规则在聚类结果中检测出热点，热点是存在高度演化耦合，并且具有层次结构的文件组。我们在6个项目上进行了实验，实验结果表明本文方法检测出的热点确实比项目中其他文件历史上的维护开销更高，热点核心文件比热点中其他文件的维护开销更高，并且比Mo R等人方法检测出的热点更为聚焦。因此开发人员应该及时注意到这些热点，结合热点的核心文件和层次结构等信息，分析发现热点中存在的不良设计，对涉及的文件进行解耦和重新设计，防止热点进一步扩大，给软件项目带来越来越高的维护开销。

本文还通过经验研究对共同修改现象进行分析，特别是分析了平行共同修改关系在不同项目中的存在和分布情况，发现平行共同修改关系是普遍存在的。结合典型案例，分析了不同维度的平行共同修改关系背后所体现的隐式共享依赖，通过演化分析，总结了四类平行共同修改关系的演化趋势。本文经验研究得到的结论，可以帮助开发人员更好地理解软件项目的共同修改现象，特别是平行共同修改关系体现的文件间的隐式共享依赖。

## 6.2 展望

在后面的工作中，我们一方面将对热点进行细化，分析热点是否存在不同的模式，不同模式的热点是否体现了不一样的设计问题。另一方面对共同修改关系进行更加深入的分析，期望对大多数频繁出现的共同修改关系给出合理的解释，帮助开发人员了解背后的隐式依赖。

# 参考文献

[1] Zimmermann T, Zeller A, Weissgerber P, et al. Mining version histories to guide software changes[J]. IEEE Transactions on Software Engineering, 2005, 31(6):429-445.

[2] Schwanke R, Xiao L, Cai Y. Measuring architecture quality by structure plus history analysis[C]// International Conference on Software Engineering. IEEE, 2013:891-900.

[3] Wong S, Cai Y, Kim M, et al. Detecting software modularity violations[C]// International Conference on Software Engineering. IEEE, 2011:411-420.

[4] Xiao L, Cai Y, Kazman R, et al. Identifying and quantifying architectural debt[C]// International Conference on Software Engineering. IEEE, 2017:488-498.

[5] Wong S, Cai Y. Generalizing evolutionary coupling with stochastic dependencies[C]// International Conference on Automated Software Engineering. IEEE, 2011:293-302.

[6] D’Ambros M, Lanza M, Lungu M. Visualizing co-change information with the evolution radar[J]. IEEE Transactions on Software Engineering, 2009, 35(5):720-735.

[7] Silva L L, Tulio Valente M, De A M M, et al. Developers' perception of co-change patterns: An empirical study[C]// International Conference on Software Maintenance and Evolution. IEEE, 2015:21-30.

[8] Beck F, Diehl S. On the congruence of modularity and code coupling[C]// International Symposium on the Foundations of Software Engineering. DBLP, 2011:354-364.

[9] Bavota G, Dit B, Oliveto R, et al. An empirical study on the developers’ perception of software coupling[C]// International Conference on Software Engineering. IEEE, 2013:692-701.

[10] Kirbas S, Caglayan B, Hall T, et al. The relationship between evolutionary coupling and defects in large industrial software[J]. Journal of Software Evolution & Process, 2017, 29(4):e1842.

[11] Mondal M, Roy C K, Schneider K A. Insight into a method co-change pattern to identify highly coupled methods: an empirical study[C]// International Conference on Program Comprehension. IEEE, 2013:103-112.

[12] Mondal M, Roy C K, Schneider K A. Improving the detection accuracy of evolutionary coupling by measuring change correspondence[C]// International Conference on Software Maintenance Reengineering and Reverse Engineering. IEEE, 2014:358-362.

[13] Mondal M, Roy C K, Schneider K A. A fine-grained analysis on the evolutionary coupling of cloned code[C]// International Conference on Software Maintenance and Evolution. IEEE, 2014:51-60.

[14] Jiang Q, Peng X, Wang H, et al. Understanding systematic and collaborative code changes by mining evolutionary trajectory patterns[J]. Journal of Software Evolution & Process, 2017, 29(3):e1840.

[15] Gethers M, Dit B, Kagdi H, et al. Integrated impact analysis for managing software changes[C]// International Conference on Software Engineering. IEEE, 2012:430-440.

[16] Li B, Sun X, Leung H, et al. A survey of code-based change impact analysis techniques[J]. Software Testing Verification & Reliability, 2012, 23(8):613-646.

[17] Hassaine S, Boughanmi F, Gueheneuc Y G, et al. A seismology-inspired approach to study change propagation[C]// International Conference on Software Maintenance. IEEE, 2011:53-62.

[18] Moonen L, Alesio S D, Binkley D, et al. Practical guidelines for change recommendation using association rule mining[C]// International Conference on Automated Software Engineering. IEEE, 2016:732-743.

[19] Nguyen H A, Nguyen A T, Nguyen T N. Using topic model to suggest fine-grained source code changes[C]// International Conference on Software Maintenance and Evolution. IEEE, 2017:200-210.

[20] Macho C, Mcintosh S, Pinzger M. Predicting build co-changes with source code change and commit categories[C]// International Conference on Software Analysis, Evolution, and Reengineering. IEEE, 2016:541-551.

[21] Mcintosh S, Adams B, Nagappan M, et al. Mining co-change information to understand when build changes are necessary[C]// International Conference on Software Maintenance and Evolution. IEEE, 2014:241-250.

[22] Xia X, Lo D, Mcintosh S, et al. Cross-project build co-change prediction[C]// International Conference on Software Analysis, Evolution and Reengineering. IEEE, 2015:311-320.

[23] Kouroshfar E. Studying the effect of co-change dispersion on software quality[C]// International Conference on Software Engineering. 2013:1450-1452.

[24] Mo R, Cai Y, Kazman R, et al. Hotspot patterns: the formal definition and automatic detection of architecture smells[C]// International Working Conference on Software Architecture. IEEE, 2015:51-60.

[25] Mo R, Cai Y, Kazman R, et al. Decoupling level: a new metric for architectural maintenance complexity[C]// International Conference on Software Engineering. IEEE, 2017:499-510.

[26] Palomba F, Bavota G, Penta M D, et al. Detecting bad smells in source code using change history information[C]// International Conference on Automated Software Engineering. IEEE, 2014:268-278.

[27] Version Control System. <https://en.wikipedia.org/wiki/Version_control_system>.

[28] Fluri B, Gall H C. Classifying change types for qualifying change couplings[C]// International Conference on Program Comprehension. IEEE Computer Society, 2006:35-45.

[29] Fluri B, Wuersch M, Pinzger M, et al. Change distilling: tree differencing for fine-grained source code change extraction[J]. IEEE Transactions on Software Engineering, 2007, 33(11):725-743.

[30] Gall H C, Fluri B, Pinzger M. Change analysis with Evolizer and ChangeDistiller[J]. IEEE Software, 2009, 26(1):26-33.

[31] Apache Software Foundation, <https://en.wikipedia.org/wiki/Apache_Software_Foundation>.

[32] Git. <https://en.wikipedia.org/wiki/Git>.

[33] JIRA. <https://en.wikipedia.org/wiki/Jira>.

[34] Camel. <http://camel.apache.org>.

[35] Cassandra. <http://cassandra.apache.org>.

[36] CXF. <http://cxf.apache.org>.

[37] Hadoop. <http://hadoop.apache.org>.

[38] Hbase. <http://hbase.apache.org>.

[39] Wicket. <http://wicket.apache.org>.

# 致谢

# 攻读硕士学位期间发表论文