Research on Composite Service Skyline with QoS Correlations

By

Yu Du

Supervised by

Associate Professor Hao Hu

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 2016

南京大学研究生毕业论文英文摘要首页用纸

THESIS: Research on Composite Service Skyline with QoS Correlations

SPECIALIZATION: Computer Software and Theory

POSTGRADUATE: Yu Du

MENTOR: Associate Professor Hao Hu

Abstract

In the service-oriented architecture, resources distributed in the Internet are encapsulated as publically-accessible Web Services, which can be further orchestrated to make composite services with well-defined processes (also known as workflows). This process is called web service composition. The key to web service composition is how to efficiently select services from a large number of similar functionality services with satisfying the consumers' requirements.

Composite service broker constructs a composite service which satisfies a consumer's preferences and constraints of QoS. But once the preferences change, the composite service broker needs to reselect the optimal one for the user by taking a comparison in the whole candidate services which is inefficiency. Composite service skyline is a set of composite services which are not dominated by others. Composite service skyline is usually used to improve the efficiency of composite service selection. When the weights of user's preferences change, the broker just need to traverse a small space to get the optimal one. State-of-the-art approach to computing composite service skyline assumes that candidate services of different tasks are independent, and the services are running in the non-mobile devices. But QoS correlations between services are commonly existing in practical applications. In addition, along with the spread of mobile network, and the appearance of smart autonomous devices, the runtime environment of web service tend to diversification: not only be a cloud service, but also a smart autonomous device. Because of the mobility of devices, it is clearly that services provided by the mobile devices have the feature of mobility. The composite service skyline

may change, because QoS or correlated quality values of a service may change as mobile devices move.

These issues cause a lot of new challenges to composite service skyline computing. In this paper, we investigate these issues respectively, and provide a solution for each of them. Specifically, we propose a service model supports QoS correlation, and based on this model, we put forward a method to compute composite service skyline. We present several pruning criteria based on which two pruning algorithms are proposed to accelerate our approach. Then a new concept of safe value range is presented aim to improve the efficiency of computing composite service skyline in mobile environment, we also propose two methods to compute and update safe value range respectively. At last, we conduct a series of experiments to evaluate the effectiveness and efficiency of our algorithms.

Keywords: Service composition, QoS correlation, Composite service skyline, Mobile Web service, Pruning, Safe value range

目录

插图

第一章　绪言

1.1 研究背景

1.2 本文组织

第二章　相关工作

2.1　需求可追踪性的相关技术

2.1.1　需求可追踪性的基本概念

在软件工程领域，人们认为应该要追踪一个需求使用期限的全过程，编制每个需求同系统元素之间的联系文档，这样的关联关系提供了需求到产品整个过程范围的明确的查阅能力，需求可追踪性（Traceability）概念的提出，正是为了应对上述问题。1970 年，需求可追踪性被列入美国国防部合同条款；1992 年，经过软件工程领域内的广泛讨论被认为有助于软件开发实践；1994 年，需求可追踪性有了如下的明确定义 [1]：

定义：需求可追踪性是指，在软件生命周期中，对某一特定需求形成以及演变的追踪能力，既包括后向追踪，从需求文档确定后一直到软件发布过程中的各种制品之间的追踪（例如设计文档、代码和测试）又包括前向追踪，书写文档形式的需求追踪到需求的来源。

无论是正向追踪或是反响追踪，我们都可以用一个二维矩阵将对应关系固化。以需求与代码间的追踪关系为例，图 N 为对应的需求追踪矩阵。其中，列表示需求项，行表示代码项，矩阵间的标记表明对应列的需求与对应行的代码存在关联关系，既此代码负责实现该需求描述的行为，在实际的软件生产过程中，需求的粒度可以是用例（Use Case）或声明（Requirement Statement），对于面向对象的程序，代码的粒度通常是类或者函数。

表 2.1: 需求追踪矩阵（需求到代码）

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

2.1.2 研究现状

Maeder 等人 [2] 分析了需求可追踪性对于软件维护任务的有益程度，他们通过研究案例 iTrust 与 Gantt 的实验，分析认为在需求可追踪性的辅助下，维护任务平均能够提高 21% 的效率与 60% 的准确度。然而，追踪关系的建立和维护需要也需要成本，只有当收益高于成本时，追踪关系的实施才有现实意义。文章 [2] 同时分析了收益成本间的关系，分析显示随着软件演化，当维护任务达到一定数量时，在需求可追踪性的辅助下能够减少维护任务的总成本。

尽管如此, 真实世界的软件系统中通常缺少固化的追踪关系 [3], 这主要是由于（1）追踪关系在软件开发的过程中难以捕获。要求开发者在实现软件功能的同时建立和维护追踪关系代价较高，开发者实施追踪关系的意愿不高；（2）追踪关系在软件开发完成后难以重建。对于一个完成的软件系统，早期参与的开发者可能已经不在项目组中，或是遗忘系统的实现细节，导致重建追踪关系的困难。因此，领域内希望通过增强过程的自动化程度以辅助开发者完成追踪关系的生成。

目前领域内主流生成追踪关系的方法是从 Antoniol 等人 [4] 的奠基性工作发展而来，该工作利用基于 VSM（向量空间模型）的方法检索需求和代码间的追踪关系。方法核心在于利用信息检索技术估计文本间的相似度。继承这一想法的工作 [5, 6] 分别选用 LSI（潜在语义索引），JS（概率模型）检索需求和代码间的追踪关系，而比较三者模型发现，没有一种检索模型占有明显优势 [7, 8]。

尽管基于信息检索的方法能够完全自动化的生成追踪关系，但是此类方法的精度（准确率，召回率）有限。这是由于基于文本匹配的信息检索存在词汇失配（Vocabulary Mismatch）的问题。代码的文本质量低，同义词问题等情况都会影响检索的精度。因此，出现了一系列的工作对基于信息检索的追踪关系生成方法进行改进。Dasgupta 等人 [9] 通过引入与代码相关的文档扩展以扩展文本的内容；Ali 等人 [10] 利用 Eye-Tracking 设备追踪维护人员在建立追踪时的眼球轨迹, 以发现对维护人员重要的代码实体, 并根据代码实体的重要性（函数 > 注释 > 变量 > 类）为来自代码实体的词项赋权；在 Ali 等人 [11] 的另一份工作中，通过挖掘软件生产过程以提供额外的信息源（Commit，Bug Report，Mail List）参与决策；Mahmoud[12] 等人通过重构代码的方式改善了代码文本的质量，以减轻词汇失配的程度。Capobianco[13] 等人则认为在检索时名词能够提供最为关键的信息，而形容词和副词等会带来干扰；Diaz 等人 [14] 提出利用代码所有权的概念提高追踪关系的检索精度。

上述方法主要是通过引入额外文本信息或对文本赋权的方式以弥补原有需求、代码文本质量的不足。而代码与需求不同，除了文本信息之外，代码还包

含结构信息，另一类工作通过深入挖掘代码的结构信息以提高追踪关系的检索精度。McMillan 等人 [15] 结合代码的文本和结构信息用于追踪关系的生成，在基于文本信息检索的候选追踪关系之上，利于结构信息进行追踪关系的重排序；Panichella 等人 [16] 认为如果候选追踪关系本身的精度较低，则基于代码结构信息的重排序可能起到反效果，因此提出在用户反馈阶段使用代码结构信息。

2.1.3　基于信息检索的需求到代码间追踪关系生成技术

在需求与软件过程的各种制品之间的关联关系中，需求与代码的关联关系引起了开发者的主要关注，这是由于需求描述了与软件系统行为有关的高层概念，而代码负责软件系统行为的实现，两者的关联关系能够帮助开发者理解程序，分析需求变更的影响性，完成包括维护任务、二次开发，代码复用在内的一系列软件活动。

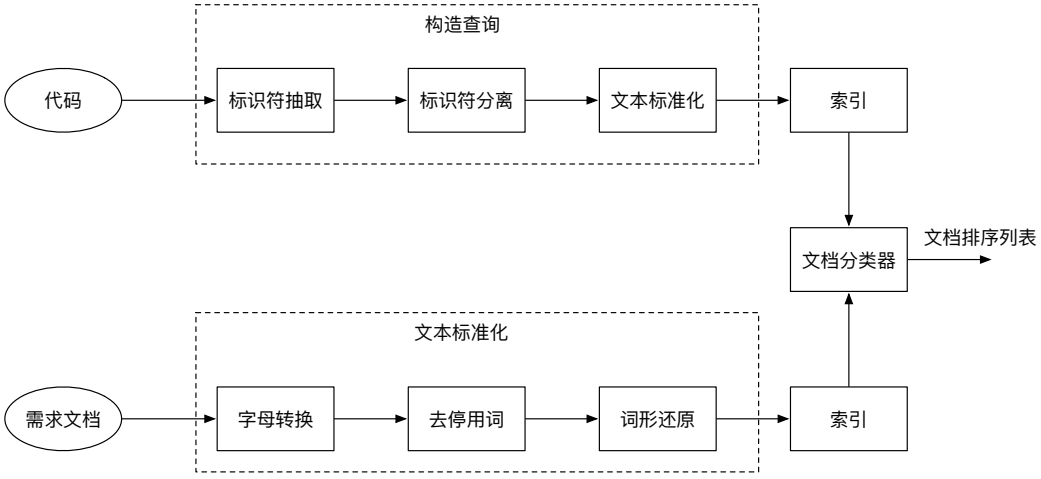


图 2.1: 追踪关系生成方法

Antoniol 等人 [4] 提出了基于信息检索的需求到代码间追踪关系生成方法。方法的核心想法是，假设代码中标识符的命名有意义，或与需求文档共用同一个术语表，因此能够通过文本匹配的方式建立需求和代码的关联关系。方法的基本过程如图 N 所示。在进行需求和代码的检索前，方法首先通过两条路径（需求文档到文档，代码到查询）构造文档与查询。

在构造文档这一路径，基于文档中抽取的词能够建立文档的索引，词的构造与索引的建立需要通过以下三个步骤的文本标准化处理：

1. 将文档中的字母统一转换为小写。

2. 根据停用词表删去文档中的停用词，通常停用词包括（标点、数字、介词等）。

3. 词形还原，包括将名词的复数形式转换为单数形式，将包含时态的动词还原为动词原形。

在构造查询这一路径，方法将每个代码组件（通常是类或函数）作为一个查询，查询的构造包括以下三个步骤：

1. 解析代码文本，提取代码组件的标识符。

2. 根据标识符的命名规则（驼峰命名法，下划线分割命名法），将标识符分隔为若干独立词项。例如，标识符 AmountDue，amount_due 将被分隔为词 amount，due。

3. 利用对文档进行标准化处理的三个步骤处理文本。

最终，通过文本分类器计算查询和文档间的相似度，并返回需求到代码的候选追踪关系列表，列表根据需求文档和代码的相似度值倒序排列。显然，排序的结果与方法采取的信息检索模型有关，我们将会对三种信息检索模型 (VSM，LSI 与 JS) 进行介绍，VSM 模型将查询和文档看作向量，文档根据计算的向量间距离排序；LSI 在 VSM 的基础上，通过奇异值分解以识别词句间隐性的关联关系的模式；JS 属于概率模型，估计文档与查询相关的概率，对文档根据关联概率进行排序。后续小节中将会对这三个检索模型进行详细阐述。

2.1.3.1　VSM

VSM（Vector Space Model）是现代信息检索系统中比较常用的检索模型，它的思想是将文档与查询都看作是高维空间中的一个向量，每个维度都是一个独立的词项。

如果某个词项出现在了文档中，那它在向量中的权重就非零。常见的权重计算方法为 TF-IDF，其中，TF（term frequency）是指给定的某一个词项在文本中的出现频率，IDF（inverse document frequency）用于度量词项的普遍重要性。根据直觉，对于查询中的某一个词项，如果在文档 A 中的出现次数多于文档 B，那么文档 A 比 B 对于查询更相关。但是这里有一个缺陷在于，内容长的文档比内容短的文档有优势，长的文档总的来讲包含的词项要多些。因此我们根据文档的长度，对词项次数进行归一化，将词项次数除以文档的总词项数，这个商称为“词项的频率”，可以作为词项的局部权重。除了局部权重以外，如果一

个词项只在很少的文档中出现，那么通过它就容易定位目标，应该赋予该词项较高的权重。反之，如果一个词项在大量文档中出现，则表明该词项缺少独特性，它的权重就应该较小。我们用“逆文本频率指数”(IDF) 来衡量词项的全局权重。TF-IDF 的计算公式如下：

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

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

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

其中，tf 指给定的某一个词项在文档中的出现频率。$n_{i,j}$ 为词项 t_i 在文档 d_j 中的出现次数，分母表示在文档 d_j 中所有词项的出现次数之和；$|D|$ 表示文档库中的文档总数，表示包含词项 t_i 的文档数目。

相似度计算是基于这样一个假设，词是文本的载体，文本的信息和词的语义是联系在一起的，也就是说同一类文本用词是相似的，不同类的文本用词各不相同。由于向量空间模型将文档与查询都看作是高维空间中的一个向量，查询与文档的相似性可基于它们在向量空间中的夹角来衡量。给定查询向量 q，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。

2.1.3.2　LSI

LSI (Latent semantic indexing，潜在语义索引) [17, 18] 在 VSM 的基础上通过分析词句的使用模式，以发现词句间潜在的联系。LSI 在自然语言文本中的应用表明 [19, 20]，LSI 不仅可以捕获独立词语的含义，而且可以捕获短语 (词组，句子) 的含义。LSI 的核心想法是，在同样的语境中出现的词语存在某些相互约束，限定了这些词之间一半具有相似的含义。在经典的文本分析应用中，LSI 使用给定的语料创建词项 - 文档矩阵 (Term-by-Document matrix)，并采用奇异值分解 [21] 的方式构造词项-文档矩阵的子空间，称之为 LSI 子空间。新的查询、文档向量通过将 VSM 空间的相关向量投影至 LSI 子空间生成。

奇异值分解背后的形式化较为复杂，详细内容可参考文献 [21]。直观来说，奇异值分解能够将一个矩阵分解为三个矩阵的乘积。给定原矩阵 X，子矩阵 U 是对原矩阵行实体分类的结果，子矩阵 V 是对原矩阵列实体分类的结果，中间的对角矩阵则表示原矩阵行实体与列实体之间的相关性，原矩阵 X 可以被此三个矩阵重建（$X = U\Sigma V^T$）。U 与 V 的列分别为左奇异向量与右奇异向量，对应的 Σ 中单调递减的对角元素称为原矩阵 X 的奇异值。当在重建矩阵过程中只使用部分奇异值因子数时，重建的矩阵是一个最小二乘法拟合。取 U，V 的前 k 列与前 k 个（最大的）X 的奇异值时，利用 $X_k = U_k\Sigma_k V_k^T$，可以构造 X 的秩 - k 近似矩阵。U 与 V 的列是正交的，故 $U^TU = V^TV = I_r$，其中 r 是原矩阵 X 的秩。X_k 由 k 个最大的奇异三元组（一个奇异值及与之对应的左右奇异向量为一个奇异三元组）构造而成，称之为原矩阵 X 的 k 维最佳近似矩阵。

对于 LSI 而言，构造以关联矩阵 X 形式表示的词项 - 文档矩阵的 k 维最佳近似矩阵 X_k。通过矩阵降维的方式，许多导致检索精度不理想的“噪音”能够被消除。为了取得较好的检索效果，选取较高的 k 值是有必要的。但需要注意的是，k 值如果过高，可能导致原矩阵 X 被重建，则词语多样性带来的干扰会造成检索精度的下降。一些研究表明，LSI 子空间的维度的选择对 LSI 的效果有较大的影响 [17, 18, 22]。在实际中，维度的选择通常由实验确定，或是参考已有工作以确定。新的查询、文档向量通过将 VSM 空间的相关向量投影至 LSI 子空间生成，查询与文档间的相似度可以由对应向量间的余弦距离计算确定。

2.1.3.3　JS

JS 模型（Jensen-Shannon similarity model）是较新的一种信息检索技术，属于概率模型。在概率模型中，假设每一个查询与文档都包含一个潜在的概率分布，则文档的排序可以由概率分布间的距离确定。JS 相似性模型将查询与文档看作词项的概率分布，概率分布间的差异度量可以由 JS 散度计算 [23] 而得，定义如下：

$$JS(q,d) \triangleq H\left(\frac{\hat{p}_q + \hat{p}_d}{2}\right) - \frac{H(\hat{p}_q) + H(\hat{p}_d)}{2} \tag{2.5}$$

$$H(p) \triangleq \sum_{w \in W} h(p(w)),\ h(x) \triangleq -x \log x \tag{2.6}$$

其中，$H(p)$ 表示概率分布 p 的熵，$\hat{p}_q$ 与 $\hat{p}_q$ 分别表示查询与文档的概率分布。根据定义可知，$h(0) \equiv 0$，故定义查询与文档间的相似度得分为 $1 - JS(q\ d)$。

2.1.4　结合代码文本与结构信息的追踪关系生成方法改进

由于词汇的多义性，代码的标识符脱离了所处的上下文信息容易具有误导性，代码注释过期 [24] 等情况的存在，相关的代码与需求的文本相似度可能较低。在本小节中，将介绍利用代码结构信息（函数调用，继承关系）改进追踪关系生成方法的相关工作 [15, 16, 25]。在基于文本匹配的候选追踪关系生成后，此类方法利用代码结构信息对候选追踪关系进行重排序。

2.1.4.1　O-CSTI

McMillan 等人 [15] 首先提出利用代码结构信息对候选追踪关系进行重排序方法。给定需求实体（例如，用例）的集合 S，代码类的集合 $C = \{C_1, \cdots, C_n\}$，$E = \{(c_i, c_j)\}$，其中 c_i 与 c_j 间存在依赖关系表示。对于某一给定的需求 s，从列表中的追踪关系 (s, c_1) 开始，如果任意 c_j 与 c_1 存在依赖关系，则给予追踪关系 (s, c_j) 的相似度值奖励 δ（常数或变量）。算法对列表中所有的追踪关系进行同样的处理，并最终根据新的相似度值对列表进行重排序，过程参见算法 N。

Algorithm 1: Optimistic Combination of Structural and Textual Information —O-CSTI

```
1 i ← 1
2 while not end of List do
3     Get the link (s, c_j) in position i of List
4     forall the c_t ∈ C do
5         if (c_j, c_t) ∈ E then
6             Sim(s, c_t) ← Sim(s, c_t) + δ × Sim(s, c_t)
7     i ← i + 1
8 Reorder List
9 The user classifies the links in List
```

2.1.4.2　UD-CSTI

在 O-CSTI 的基础上，Panichella 等人 [16] 认为利用代码依赖关系进行的重排序受制于信息检索方法的精度，如果初始的候选追踪关系排序本身质量较低，那么重排序可能会引入更多的错误。因此，作者认为应该在用户反馈阶段执行基于代码依赖关系的重排序，改进后的算法参见算法 N。同时，作者在文章中提出了一种自适应的奖励计算方式：$\delta = median\{v_i, \cdots, v_n\}$。其中 $vi = (max_i - min_i)/2$,

v_i 表示第 i 个实体相似度值的可变性，max_i 与 min_i 表示实体 i 检索结果中相似度的最高值与最低值。

Algorithm 2: 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
```

2.1.4.3　PageRank

根据代码依赖关系，Scanniello 等人 [25] 利用 PageRank 算法计算代码的结构重要性，并将需求与代码的文本相似性与该代码的结构重要性的乘积作为排序的依据。

2.2　过时需求自动检测的相关技术

在软件维护和演化的过程中，一份最新的需求规约能够提供有价值的信息以辅助维护人员完成相应的软件活动。例如，需求规约包含系统实现背后的原理，因此可以帮助维护人员理解程序，或是防止重要的决策被意外忽视。另外，需求规约通常由自然语言文本描述，可以作为用于和软件工程领域外的利益相关者讨论功能更改的基础。因此，当需求规约内的信息变得过时且不可靠时，将减弱系统的可维护性。最终，系统将会进入“维修阶段”(servicing stage) [26]，此时只可能对系统进行较少且次要的更改。

事实上，在系统演化时，维护人员通常不会及时更新需求规约 [26, 27]，这主要是由于需求的更新仍然需要人工参与，代价高昂，并伴随相应的时间成本。在实践中，维护人员需要查阅可能成百上千页的需求规约，才能够定位其中需要更新的部分。因此，根据 Lethbridge 等人 [27] 的观察，维护人员在面临维护

或演化任务时，通常直接更新代码，而不更新相应的需求。很快，需求规约将会变得过时且失效。

2.2.1 过时需求的基本概念

在本小节，我们将阐述过时需求，受影响需求，被检测需求这三者的概念与它们之间的联系 [28]。当一个属于需求规约的需求不再反映利益相关者当前的需要时，则该需求被认为是过时的。当维护人员实施一次代码变更后，导致需求规约中的一个或一组需求与新版本的代码不一致时，则这一个或一组需求是受变更影响的需求。当代码变更是用于满足利益相关者新的需要（或是已有需要的改变）时，则受影响需求同时也是过时的。然而，这两类的需求并不是完全重叠的。例如，由于利益相关者的需要发生了改变，则某个需求可以是过时需求，但代码并没有（或是此刻没有）发生更改，因此该需求不是受影响需求。相反的，如果在实施代码变更时没有进行事先的影响性分析，则代码变更很有可能影响某些需求，尽管它们并不过时。然而，在一个管理良好的软件演化过程中，代码通常与利益相关者的需要保持一致，因此过时需求与受影响需求存在很大程度上的重叠（如图 N 所示）。过时需求自动检测方法通过在代码变更时自动检测受影响需求，从而帮助维护人员识别需求规约中的过时需求。维护人员需要查看这些被检测需求以更新需求规约。在理想情况下，所有受影响需求都应该被检测到。然而，目前过时需求自动检测包含自动分类、检索等步骤，因此方法会存在误报（检测到不受影响的需求）和漏报（受影响的需求未被检测到）的情况。

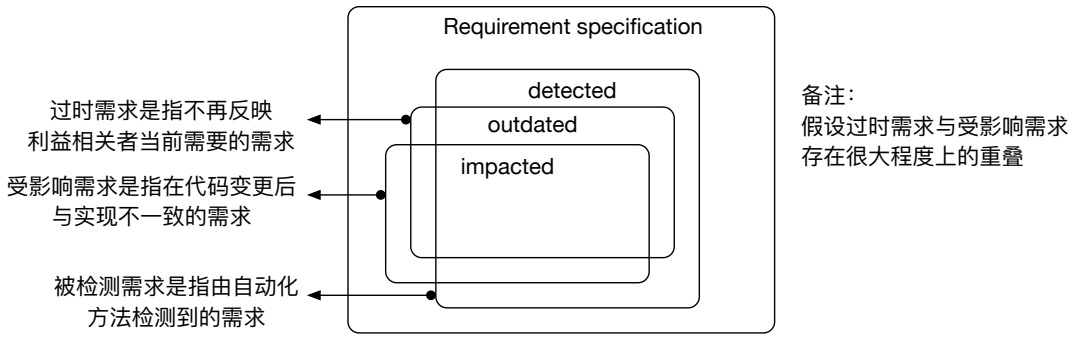


图 2.2: 过时需求，受影响需求与被检测需求

2.2.2　基于代码变更的过时需求自动检测方法

在软件系统开发的过程中，将创建各种不同的软件实体，其中，代码在软件系统的行为需要发生改变时，会被进行相应的变更。这是由于如果代码没有发生变更，那么软件系统的行为不会有实际的变化。在实施一次代码变更时，需要执行代码层的影响性分析以识别其中需要被修改的部分。因此，过时需求自动检测方法 [28] 利用执行代码层影响性分析的过程自动化检测需求中受影响的部分。

在维护人员提交代码变更后，过时需求自动检测方法能够检索潜在受影响需求的排序列表，以供维护人员核查，维护过程参考活动图 N[28]。在实施代码变更（A1）并提交至版本控制系统后（A2），代码变更将被自动化分析以判断这些代码变更是否影响需求（A3）。如果检测到影响需求的变更，则利用这些代码变更追踪需求规约（A4），然后将相关的需求以排序列表的形式向维护人员展示（A5），维护人员查看整个候选列表以确定其中的过时需求，并实施更新（A6）。

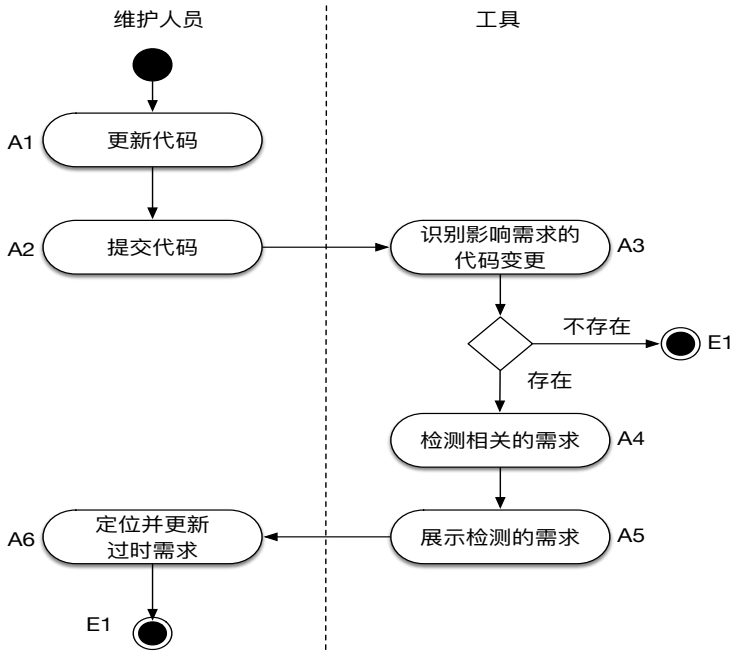


图 2.3: 活动过程：利用过时需求自动检测方法的维护过程

从实现角度出发，基于代码变更的过时需求自动检测包括三个步骤：（1）识别影响需求的代码变更；（2）检测受变更影响的需求；（3）向用户展示检测的需求。

2.2.2.1 识别影响需求的代码变更

影响需求的代码变更是指，该代码变更能够影响系统的预期行为，或是能够影响系统组件处理外界刺激，数据或功能的反馈 [29]。Charrada 等人通过人工比较 ZXing 项目 1.6 与 1.7 版本的区别，提出了以下六点观察，并根据观察提炼了相应的启发式规则：

1. 函数体内的代码变更，多数与重构、缺陷修复有关。因此，忽略函数体内的变更；
2. 新增或删除代码元素（包、类、函数、域）通常与系统功能的新增或扩展有关。因此，代码元素的新增或删除可能影响系统的外部行为；
3. 删除某一代码元素，并增加一个与其命名相似的元素，通常是重构（Rename）操作。因此，忽略与 Rename 有关的代码变更；
4. 函数签名的变更通常是重构操作。因此，忽略函数签名的变更；
5. 私有代码元素的变更会影响系统的外部行为。
6. 多个函数拥有同一个命名 (函数重载), 通常他们与同一需求有关。因此, 只选取其中一个函数进行分析。

根据上述观察得出的启发式规则，方法通过比较代码版本差异，识别其中的变更代码元素（包、类、函数、域），并过滤其中与 Rename 操作有关的变更。Rename 操作可以基于新增和删除的代码元素标识符间文本相似性来判断，相似性可通过编辑距离 [30] 等方式求得。

2.2.2.2 检测受变更影响的需求

根据步骤一中影响需求的代码变更，方法将抽取与代码变更有关的关键词，并利用关键词以追踪需求，生成受变更影响的需求排序。

2.3 本章小结

第三章　基于代码依赖紧密度分析改进需求可追踪性生成方法

3.1　引言

需求可追踪性是指，在软件生命周期中，对某一特定需求形成以及演变的追踪能力 [1]。在软件维护和演化的过程中，需求可追踪性能够追踪一个需求使用期限的全过程以辅助利益相关者完成相应的软件活动 [2]。例如，需求到代码间的追踪关系能够帮助利益相关者进行代码变更的影响性分析，以维护需求和代码的一致性。然而，在实际的软件系统中，通常存在大规模的追踪关系，且软件实体（尤其是代码）变更频繁，导致追踪关系难以建立 [31]。

为了减少生成追踪关系的人工成本，领域内已有工作集中于提高追踪关系的自动化程度。目前，信息检索技术被广泛用于解决追踪关系的生成问题 [4–6]。一般的，基于信息检索的追踪关系生成方法利用信息检索模型（如 VSM，JS 与 LSI）计算软件实体间（如需求与代码）的文本相似性，以自动化生成候选追踪关系列表，减少了实体间潜在关联的搜索空间。然而，以需求与代码为例，基于信息检索方法的效果严重依赖与需求与代码的文本质量，词汇失配（Vocabulary Mismatch）问题难以避免。为了解决这一问题，已有工作从提供额外的信息源（提交日志，缺陷修复报告，邮件列表）[11]，为代码中不同来源的信息赋权 [10]，考虑代码所有者上下文 [14] 等方面着手，以提高基于信息检索的追踪关系生成方法的精度。

同时，另一类工作关注于结合代码的文本信息与结构依赖应用于需求可追踪性生成及类似的领域，如特征定位 [32]，概念分配 [25]。此类方法首先基于信息检索技术生成初始的候选追踪关系，并基于代码依赖关系的分析对已有候选追踪关系进行过滤或重排序。近期工作通过对代码依赖的进阶分析（利用 PageRank 算法 [25]），结合代码依赖与用户反馈 [16] 等方式提高了检索的精度。然而，上述工作在分析代码依赖关系时，认为所有的代码依赖关系是同样重要的，没有对各个依赖关系的重要性加以区分，从而无法充分发挥代码依赖关系的作用。

本章中，我们提出了一种代码依赖关系的紧密度分析方法以量化代码类之间依赖的交互程度。同时，基于代码关系的紧密度分析，我们提出了一种改进的需求到代码间追踪关系的生成方法 TRICE（Traceability Recovery based on Information retrieval and ClosEness analysis）

3.2 背景

在本节，我们将说明本工作的研究动机，并给出我们的问题定义。

3.2.1 研究动机

在本小节，我们将给出一个具体的示例以解释我们的研究动机。

在医疗数据管理系统 iTrust 中，存在类 MonitorAdverseEvents。观察代码发现，类 MonitorAdverseEvents 分别与 MonitorAdverseEventAction，AuthDAO 存在调用依赖，MonitorAdverseEvents 与前者间存在三个不同的函数调用，与后者间只存在一个函数调用。此外，MonitorAdverseEventAction 只被 MonitorAdverseEvents 调用，而 AuthDAO 还被其它类所调用。比较这两种依赖关系，我们发现到 MonitorAdverseEvents 与 MonitorAdverseEventAction 之间有更强的交互，暗示着它们之间的依赖关系更紧密。类似的观察已经作为启发式规则应用于重构中类的自动提炼 [33]。

在软件系统中，一个需求通常是由代码中的若干类共同协作实现的。因此，如果两个类之间的依赖关系较为紧密（有较强的交互），则它们在功能上可能具有较高的相似性。在生成需求到代码间的追踪关系时，对代码依赖关系的紧密度分析可以发现与需求文本相似性低，但与相关类依赖关系紧密的类，以达到提高检索精度的目的。

3.2.2 问题定义

在本节，我们将定义基于代码依赖关系紧密度分析改进需求可追踪性生成方法的问题。

问题定义. 给定一个需求集合 R，该需求集合包含 m 个需求文本，同时给定代码集合 C，包含 n 个类，且类之间存在依赖关系（如调用依赖，数据依赖）。基于信息检索的需求到代码间追踪关系生成方法能够建立候选追踪关系列表 $T = \{t_1, \cdots, t_n\}$，t_i 为一个三元组 $(req, class, score)$，其中 $req \in R$，$class \in C$，$score$ 表示需求与代码类的相似度得分，列表 T 中的追踪关系根据相似性得分倒序排列。问题是：分析代码依赖关系的紧密度是否能够提高追踪关系生成方法的精度。

3.3 方法概述

在本节，我们将介绍基于代码依赖紧密度分析的需求可追踪性生成改进方法 TRICE。TRICE 的过程包括以下四个步骤：(1) 代码类间的依赖关系捕获与组

织；(2) 计算代码依赖关系的紧密度，并生成相应的图结构 CDCGraph (Code Dependencies Graph with Closeness)；(3) 基于信息检索技术生成需求到代码的候选追踪关系；(4) 利用代码依赖关系的紧密度分析，对候选追踪关系进行重排序。我们将在后续四个小节对以上步骤进行详细阐述，并辅以 iTrust 中一个实际的例子来说明相关概念。

3.4 代码类间的依赖关系捕获与组织

在本章工作中，我们考虑了以下四种类间的依赖关系，并辅以 iTrust 代码片段进行说明。

1. 调用依赖：如果两个类之间存在一个调用依赖，则表示两个类之间至少有一个函数调用。由于类 MonitorAdverseEventAction 中调用了 EmailUtil 的函数 sendEmail()，因此类 MonitorAdverseEventAction 与 EmailUtil 存在调用依赖。

2. 使用依赖：如果一个类使用了另一个类的对象，则表示两个类之间存在使用依赖。故类 MonitorAdverseEventAction 与 EmailUtil 存在类的使用依赖。

3. 继承依赖：表示两个类之间存在继承关系。

4. 数据依赖：表示两个类之间读写了同样的数据内容 [34]。观察图 N 可以发现，类 MonitorAdverseEventAction 与 EmailUtil 之间存在显式的调用依赖和使用依赖，但 EmailUtil 与 SendMessageAction 之间并没有显式的依赖关系。然而，函数 SendMessageAction.saveReceiver() 与函数 EmailUtil.sendEmail() 都传递了 Email 类型的对象作为参数，因此 EmailUtil 与 SendMessageAction 之间存在数据依赖。

为了捕获上述代码依赖，我们选用基于 JVMTI (Java Virtual Machine Tool Interface) 的动态分析工具 [34] 以捕获函数间的调用及数据依赖。由于该工具是加载于 JVM 的运行时刻进行依赖捕获，并通过测试用例驱动执行，因此能够分析真实的代码执行时刻依赖，同时正确处理多态。

根据函数间的调用及数据依赖，我们可以衍生出上述类间的代码依赖。首先，类的调用依赖可以由类之间包含多少不同的函数调用 (同方向的) 来确定。其次，如果两个属于不同类的函数间存在数据依赖，则认为函数所属的类存在

同样的数据依赖，类的使用依赖也由此确定。最后，由于子类会调用父类的构造函数，因此类的继承依赖可以从函数的调用依赖中获得。

由于类的调用依赖，使用依赖，继承依赖结构相似且有较大的重叠关系，因此我们将上述三种依赖统一表示为直接调用依赖。后续我们将介绍对类的直接调用依赖与数据依赖的紧密度计算方法。

3.5　代码依赖关系的紧密度计算

基于捕获的代码依赖关系，我们通过以下四个步骤计算代码依赖关系的紧密度，并生成相应的图结构 CDCGraph。

1. 建立代码依赖关系图（CDGraph）：首先，我们定义代码依赖关系图 CDGraph（Code Dependency Graph）为一个有序对 $CDGraph =< V, E >$，其中 V 是代码中类的集合，E 是代码依赖边的集合，其中，依赖边包括类的直接调用依赖与数据依赖。图 N 展示了 iTrust 系统示例的代码依赖关系图。其中，有方向的实线表示直接调用依赖，并标记了不同的函数调用或类的使用数量；无方向的虚线表示数据依赖，并标记了类之间共享数据类型的数量。

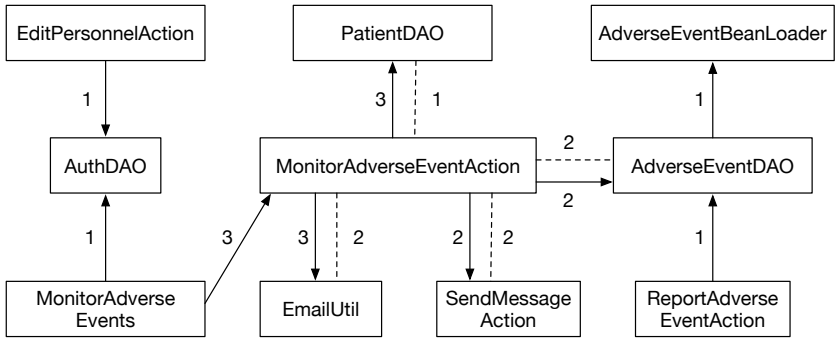


图 3.1: 与日志记录、展示功能有关的部分函数及其依赖结构

2. 直接调用依赖的紧密度计算：根据对研究动机中示例的观察，代码类之间的依赖关系表明了类之间的交互程度。在后文中，我们用源节点表示调用（使用）者，用目标节点表示被调用（使用）者。对于直接调用依赖而言，依赖的紧密程度可能与以下 3 点有关:（1）类之间包含多少不同的函数调用或类的使用；（2）源节点调用（使用）了多少其它的类，既源节点的出度；（3）目标节点被多少其它的类调用（使用），既目标节点的入度。从直觉上理解，如果两个类之间存在较多种类的函数调用或类的使用，那么这

两个类之间的交换较为紧密。同时，如果源节点的出度越小，且目标节点的入度越小，则表明源节点更专注于完成目标节点的任务，目标节点也更专注于服务源节点。基于上述两点观察，我们定义类间直接调用依赖的紧密度计算公式如下：

$$Closeness_{call} = \frac{2N}{WeightedOutDegree_{e.source} + WeightedInDegree_{e.target}} \tag{3.1}$$

其中,N 表示两个类之间依赖边上不同的函数调用与类的使用的数目,$WeightedOutDegree_{e.sour}$ 表示源节点的出度，$WeightedInDegree_{e.target}$ 表示目标节点的入度，并用每一条直接调用依赖边上，不同的函数调用与类的使用的数目，来表示出入度的权重。

3. 数据依赖的紧密度计算：数据依赖关系表明两个类之间共享了某些类型的数据。在计算数据依赖的紧密度之前，我们首先需要过滤与通用数据类型有关的数据依赖。在 iTrust 系统中，DAOFactory 作为与数据库交互的工具类，它被系统中的绝大多数类所访问。这些类相互之间虽然存在关于 DAOFactory 的数据依赖，但这样的数据依赖由于 DAOFactory 过于通用，无法用于分析类之间的紧密度分析。因此，我们引入一种数据类型的权重计算方式 IDTF (Inverse Data Type Frequency) [34]：

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

其中, N 表示数据依赖的总数, ndt 表示给定的数据类型在所有数据依赖中出现的次数。我们通过设置阈值 T - idtf 的方式，过滤 IDTF 值在阈值以下的数据类型和基于此类数据类型的数据依赖。与信息检索中的 IDF 概念类似，IDTF 反映了一个数据类型在系统整个代码中的全局重要性。IDTF 值越高，表示该数据类型越独特，如果两个类有关于该数据类型的数据依赖，则表明类之间的交互也较强。除此以外，如果两个类还分别与其它类存在数据依赖，则会降低两个类之间数据依赖的本地重要性。基于上述两点观察，我们定义类间数据依赖的紧密度计算公式如下：

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

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。

4. 生成 CDCGraph：基于上述三个步骤，我们定义 CCDGraph（Code Dependency Graph with Closeness）为一个有序对 $CCDGraph =< V, E >$，其中 V 是代码中类的集合，E 是代码依赖边的集合，其中，依赖边包括类的直接调用依赖与数据依赖，前者依赖边的权重为直接调用依赖的紧密度值，后者依赖边的权重为数据依赖的紧密度值。由图 N 衍生而来，图 N 展示了 iTrust 系统示例的 CDCGraph。

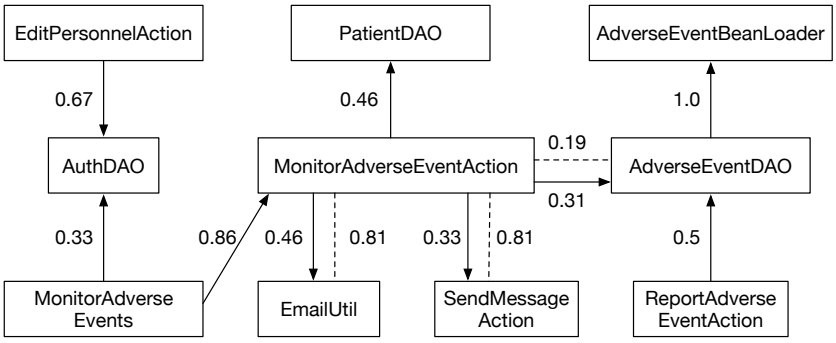


图 3.2: 与日志记录、展示功能有关的部分函数及其依赖结构

3.6 生成需求到代码的候选追踪关系

1. 生成语料库：对于代码中的每一个类，我们抽取其类名，类的注释，函数名，函数的注释，域名等信息作为代码的文本。对于每一个需求（如用例），我们抽取需求的标题及内容（在用例中，一般包括前提，主流程，子流程等）作为需求文本。

2. 标准化语料库：对于代码的文本，我们首先根据代码标识符的命名规则（驼峰命名法，下划线分割命名法），将标识符分隔为若干独立词项。然后，对需求和代码文本进行统一的预处理操作以标准化文本，预处理过程包括去停用词，词形还原和词根提取等。

3. 索引语料库并计算相似度：我们利用 TF-IDF 计算词项的权重并建立索引，在计算需求文本与代码文本时，我们使用了三个不同的检索模型：VSM[4]，LSI[5] 与 JS[7]。

4. 生成候选追踪关系列表：对于每个需求，将代码按相似度值倒序排列，生成候选追踪关系列表。

3.7　基于代码依赖关系紧密度的追踪关系重排序

在本节，我们的目的在于基于代码依赖关系改进候选追踪关系的排序。我们的想法源于以下两点观察：（1）对于给定的一个需求，候选追踪关系中 IR 值最高的类通常与该需求有关（2）对于给定的一个需求，实现该需求的代码通常是相互关联的，形成关联的需求域 [31]，而不是分散的。基于上述两点观察，我们提出了基于代码依赖关系紧密度分析的追踪关系重排序方法，包括以下两个步骤：（1）建立初始需求域，并提升初始域内所有的类；（2）对于初始域外的类，根据它们与域内类的代码依赖关系，给这些初始域外类的 IR 值以奖励。以下我们将具体阐述每个步骤。

3.7.1　建立初始需求域

我们给定阈值 T_{call} 与 T_{data}，以对 CDCGraph 进行剪枝，删去紧密度值小于 T_{call} 的直接调用依赖边与紧密度值小于 T_{data} 的数据依赖边。剪枝后 CDCGraph 中的依赖边表明较为紧密的依赖关系。对于给定的一个需求，我们选取候选追踪关系中 IR 值最高且存在依赖邻居（直接调用依赖或数据依赖）的类作为建立初始需求域的种子类，如果 IR 值最高的类没有依赖邻居，则依次寻找次高 IR 值的类。然后，我们基于 CDCGraph 引入与种子类依赖关系紧密的其他类，以建立初始域。在此过程中，对于两种不同的代码依赖关系，我们使用了两种不同的策略。在剪枝后的 CDCGraph 中，对于直接调用依赖，我们将种子类所处调用链中的其他类加入初始域；对于数据调用依赖，我们将种子类的直接邻居类加入初始域。对这些因为与种子类存在依赖关系，而被加入初始域的类，我们将其 IR 值提升至与种子类相同的 IR 值。我们建立初始需求域的具体算法如下：

3.7.2　重排序初始域外的追踪关系

在建立初始域之后，我们分析初始域外类与域内类的代码依赖关系，以给予这些初始域外类的 IR 值奖励。类似于建立初始域的过程，我们为直接调用依赖与数据依赖设计了两种不同的策略。

对于直接调用依赖，假设初始域外的类为 C_{out}，初始域内的类为 C_{in}，我们在 CDCGraph 中寻找 C_{out} 与 C_{in} 之间的有效路径。一条有效路径需要满足以下两个条件：（1）路径中的边必须是同向，指可以迭代地访问 C_{out} 的调用节点（或被调用节点）以到达 C_{in}；（2）路径中包含的节点只有一个（既 C_{in}）在初始域内（为了避免重复）。如果这样的有效路径存在，则我们计算路径中直接调用依赖边紧密度值的几何平均值，并根据如下公式更新 C_{out} 的 IR 值：

Algorithm 3: Establishing Initial Region

```
1  initialRegion ← ∅;
2  prunedGraph ← CDCGraph.setPruning(T_call,T_data);
3  topLink ← candidateList.next();
4  while prunedGraph.hasNoNeighbors(topLink.class) do
5      initialRegion.add(topLink.class);
6      topLink ← candidateList.next();
7  initialRegion.add(topLink.class);
8  initialRegion.topIRValue ← topLink.IRValue;
9  reachedClasses ← ∅;
10 reachedClasses.add(prunedGraph.getTransitiveCallers(topLink.class));
11 reachedClasses.add(prunedGraph.getTransitiveCallees(topLink.class));
12 reachedClasses.add(prunedGraph.getNeighborsByData(topLink.class));
13 foreach link in candidateList do
14     if reachedClasses.contains(link.class) then
15         link.IRValue ← initialRegion.topIRValue;
16         initialRegion.add(link.class);
17 candidateList.reorderByIRValue();
```

$$IR_{call} = IR_{origin} + (IR_{top} - IR_{origin})^{|PATH|} \sqrt{\prod_{x \in PATH} Closeness_{call}(x)} \tag{3.4}$$

其中，IR_{origin} 表示 C_{out} 原有的 IR 值，IR_{top} 表示初始域中种子类的 IR 值，$Path$ 表示 C_{out} 与 C_{in} 间路径的依赖边集合，并用 $Closeness_{call}(x)$ 表示路径中直接调用依赖边 x 的紧密度值。需要注意的是，C_{out} 与 C_{in} 间可能存在多条路径，此时我们只选取使 C_{out} 的 IR 值增幅最大的路径。

对于数据调用依赖，如果初始域外的类 C_{out} 为初始域内的类 C_{in} 的直接邻居，则基于数据依赖的紧密度值更新 C_{out} 的 IR 值，公式如下：

$$IR_{call} = IR_{origin} + (IR_{top} - IR_{origin}) Closeness_{data}(x) \tag{3.5}$$

其中，其中，IR_{origin} 表示 C_{out} 原有的 IR 值，IR_{top} 表示初始域中种子类的 IR 值，$Closeness_{data}(x)$ 表示 C_{out} 与 C_{in} 间数据依赖边的紧密度值。

基于直接调用依赖（或数据依赖），如果初始域外的类 C_{out} 到多个初始域内的类都存在有效路径（或为多个初始域内类的直接邻居），则 C_{out} 的 IR 值可以被多次奖励。此外，C_{out} 的 IR 值可以同时获得来自直接调用依赖和数据依赖的奖励，但是 IR 值提升的上限不能超过初始域内种子类的 IR 值。基于代码依赖关系紧密度重排序追踪关系的算法如下：

Algorithm 4: Re-rank Links outside Initial Region

```
1  topIRValue ← initialRegion.topIRValue;
2  foreach link in candidateList do
3      if !initialRegion.contains(link.class) then
4          foreach c in initialRegion do
5              pathList ← findValidPaths(link.class, c);
6              gMean ← 0;
7              foreach path in pathList do
8                  gMean ← max(GeometricMean(Closeness_call(path)),
                   gMean);
9              link.IRValue ← link.IRValue + gMean(topIRValue -
               link.IRValue);
10             if hasDataDependencies(c, link.class) then
11                 link.IRValue ← link.IRValue + Closeness_data(c,
                   link.class)(topIRValue - link.IRValue);
12         if link.IRValue > topIRValue then
13             link.IRValue ← topIRValue;
14 candidateList.reorderByIRValue();
```

3.8 实验与分析

在此小节，我们通过实验以验证 TRICE 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。

3.8.1 实验系统及 IR 模型

我们的实验在三个真实的中量级软件系统上进行：iTrust, GanttProject 与 jHotDraw。表 N 列举这三个系统的基本信息。这些系统都提供有效的需求规约，更重

表 3.1: My caption

	iTrust	Gantt	jHotDraw
版本	13.0	2.0.9	7.2
编程语言	Java	Java	Java
千行代码（KLoC）	43	45	72
代码（类）	131	124	144
需求（用例）	34	16	16
调用依赖	274	452	691
数据依赖	4844	1788	1815
追踪关系	248	315	221

要的是，这些系统提供标准的需求到代码间的追踪关系（RTM），用于作为实验的 Oracle。其中，iTrust 在发布时就提供了需求到函数的 RTM。对于 jHotDraw 与 GanttProject，我们招募了系统的原开发者以建立高质量的需求到类的 RTM。为了保持实验时 RTM 粒度的一致性，对于 iTrust，我们由需求到函数的 RTM 衍生出需求到类的 RTM（如果需求到某一个函数的追踪关系存在，那么需求到该函数所属类的追踪关系存在）。为了保证 TRICE 的泛化能力，我们选取了三个被广泛使用的 IR 模型，VSM，LSI 与 JS。

3.8.2 评价指标

为了验证不同追踪关系生成方法的表现，我们选用了信息检索中两个重要的指标 Precision（准确率）与 Recall（召回率）：

$$recall = \frac{|correct \cap retrieved|}{|correct|}\% \tag{3.6}$$

$$precision = \frac{|correct \cap retrieved|}{|retrieved|}\% \tag{3.7}$$

其中 *correct* 表示正确的追踪关系（在 Golden RTM 中）集合，*retrieved* 表示追踪关系生成方法所检索的追踪关系集合。一种常用的比较 IR 方法的方式是，在不同的 *Recall* 水平比较方法间的准确率，可以由 *Precision – Recall* 曲线展示。为了进一步衡量追踪关系的整体质量，我们选用了信息检索中两个常用指标：*AP*（Average Precision）与 *MAP*（Mean Average Precision）。其中，*AP* 用于度量全部查询（需求）所检索的相关文档（追踪关系）的排序质量，计算

方式如下：

$$AP = \frac{\sum_{r=1}^{N} (Precision(r) \times isRelevant(r))}{|RelevantDocuments|} \tag{3.8}$$

其中，r 表示目标实体在列表中的排序，N 表示文档的总数。$Precision(r)$ 表示在前 r 个时，列表的准确率。$isRelevant()$ 为一个二值函数，如果文档是相关的，则返回 1，若无关，则返回 0。同时，MAP 用于度量不同查询（需求）所检索的相关文档（追踪关系）AP 的平均值，计算方式如下：

$$MAP = \frac{\sum_{q=1}^{Q} AP(q)}{q} \tag{3.9}$$

其中，q 表示一次查询（需求），Q 表示查询（需求）的总数。

3.8.3 研究问题

我们在 x.x.x 节定义了如下的研究问题：

Q：分析代码依赖关系的紧密度是否能够提高追踪关系生成方法的精度。

为了回答上述研究问题，我们将 TRICE 与三种基线方法进行比较：纯粹基于信息检索的方法（IR-Only），结合文本与结构信息的方法（O-CSTI[]），利用 PageRank 算法的方法。

除了 x.x.x 节中的评价指标，我们进行显著性检验以验证 TRICE 的表现是否能够显著地优于基线方法。为了构建显著性检验，我们计算在检索到每个正确的追踪关系时的 F-measure 值，作为显著性检验的因变量。F-measure 同时考量了准确率与召回率，其计算方式如下：

$$F = \frac{2}{\frac{1}{R} + \frac{1}{P}} \tag{3.10}$$

其中 P 表示准确率，R 表示召回率，F 为 P 与 R 的调和平均。对于每种追踪关系生成方法，其正确的追踪关系数量是一致的，因此我们采用 Wilcoxon 秩和检验以检验如下的零假设：

H_0：TRICE 方法与其他基线方法的表现没有明显的区别。

我们选用 $\alpha = 0.05$ 以接受或拒绝该零假设。

3.8.4 实验结果与分析

表 N 展示了三个实验系统的实验结果，我们比较了 TRICE 与其他基线方法的效果（AP，MAP 与显著性检验的 $p-value$ 值）。在所有 27 个比较实例的 22

表 3.2: My caption

		VSM		LSI		JS	
		MAP	p-value	MAP	p-value	MAP	p-value
iTrust	IR-ONLY	58.69	0.02	59.92	0.03	56.90	<0.01
	TRICE	61.65	-	62.70	-	62.61	-
	PageRank	54.03	0.14	49.47	0.50	39.31	<0.01
	O-CSTI	44.87	0.63	42.02	0.42	34.80	<0.01
Gantt	IR-ONLY	49.80	<0.01	51.70	<0.01	46.77	0.01
	TRICE	54.00	-	52.40	-	49.70	-
	PageRank	48.84	<0.01	45.38	<0.01	43.98	<0.01
	O-CSTI	47.70	<0.01	41.18	<0.01	39.20	<0.01
JHotDraw	IR-ONLY	50.09	0.35	49.51	0.02	44.86	0.01
	TRICE	52.05	-	51.91	-	47.10	-
	PageRank	29.24	<0.01	22.52	<0.01	18.28	<0.01
	O-CSTI	23.00	<0.01	20.23	<0.01	19.92	<0.01

个中，TRICE 的 $F-measure$ 值能够明显的高于基线方法。而且，TRICE 的 AP 与 MAP 在大体上要高于其他所有基线方法，唯一的例外是在 Gantt-LSI 中，IR-Only 方法的 AP 要略微高于 TRICE（小于 0.5%）。图 N 展示了所有九个实验的 $Precision-Recall$ 曲线，说明了 4 种方法的效果。图中的实验根据实验系统与使用的检索模型以分组。

根据表 N 所示，我们发现 O-CSTI 与 PageRank 的表现并没有明显的优于 IR-Only 方法，因此我们主要关注于 TRICE 和 IR-Only 间的比较。表 N 展示了 TRICE 与 IR-Only 在不同的 Recall 水平上的 AP 与 FP（False Positive），可以看出 TRICE 能够将检索的 AP 提高 21.37%（在 Recall 水平为 40% 的 iTrust-JS 下），误报减少 445（在 Recall 水平为 80% 的 iTrust-JS 下）。同样，我们观察发现，TRICE 方法几乎没有引入额外的误报，唯一的例外是在 Recall 水平为 80% 的 jHotDraw 下，TRICE 额外引入了 185 个 FP。这些发现表明通过代码依赖关系的紧密度分析与重排序算法，追踪关系生成方法的效果能够被有效的提高，并且很少引入额外的误报。

根据实验结果，我们还有一点额外的观察。如果基于信息检索生成的候选追踪关系质量较高（例如，iTrust-JS，反之，jHotDraw-VSM），则 TRICE 能够带来较大的提高。这表明 TRICE 能够获益于追踪关系的提高，甚至与其他改进

表 3.3: My caption

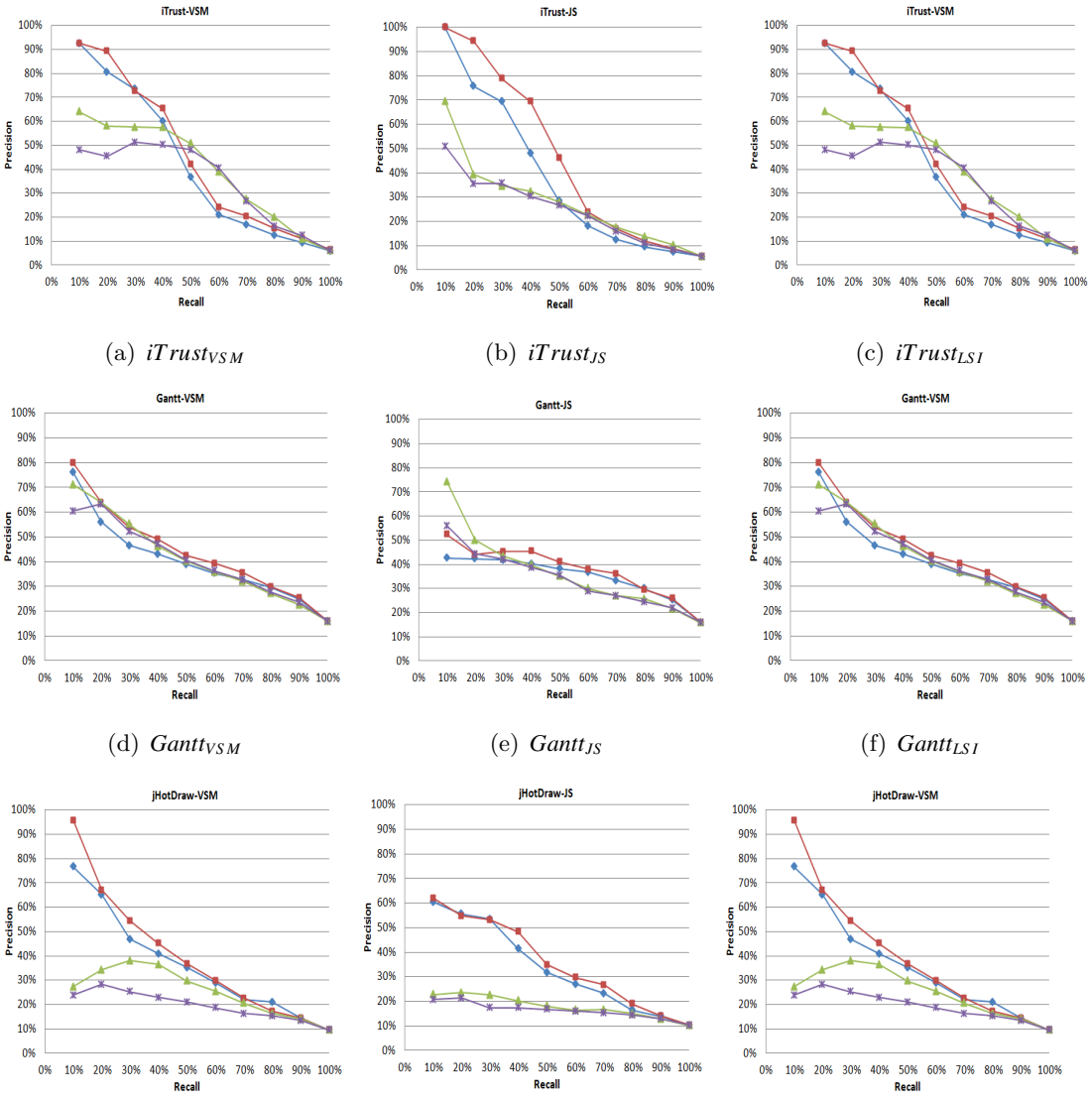
		Recall (20%)		Recall (40%)		Recall (60%)		Recall (80%)	
		Precision	FP	Precision	FP	Precision	FP	Precision	FP
iTrust	VSM	+8.64%	-6	+5.12%	-13	+3.00%	-89	+2.74%	-280
	JS	+18.58%	-13	+21.37%	-64	+5.76%	-196	+2.57%	-445
	LSI	+1.46%	-1	+4.52%	-12	+3.40%	-86	+4.39%	-425
Gantt	VSM	+7.86%	-14	+6.03%	-36	+3.81%	-52	+0.62%	-28
	JS	+1.75%	-6	+5.33%	-37	+1.11%	-15	-0.46%	+13
	LSI	-1.94%	+3	+2.92%	-20	+3.24%	-45	+2.63%	-83
jHotDraw	VSM	+1.95%	-2	+4.16%	-20	+0.91%	-14	-3.81%	+185
	JS	-0.68%	+1	+6.97%	-31	+2.54%	-42	+2.52%	-142
	LSI	-2.20%	+2	+4.66%	-21	+3.54%	-51	+0.23%	-12

方法相协作。

总体上来说，实验结果表明代码依赖关系的紧密度分析能够帮助提高基于信息检索的追踪关系生成方法的精度，当 Recall 水平在 20% 至 80% 时，提升较为明显。

3.9 本章小结

我们在本章定义了在存在 QoS 关联的场景下计算组合服务 Skyline 的问题，并给提出一种支持 QoS 关联的组合服务 Skyline 计算方法。我们提出了一系列剪枝规则来提高我们方法的效率。最后，我们通过一些列实验来分析我们方法的有效性以及效率。



(a) $iTrust_{VSM}$　(b) $iTrust_{JS}$　(c) $iTrust_{LSI}$

(d) $Gantt_{VSM}$　(e) $Gantt_{JS}$　(f) $Gantt_{LSI}$

(g) $jHotDraw_{VSM}$　(h) $jHotDraw_{JS}$　(i) $jHotDraw_{LSI}$

图 3.3: F-measure/cut curves for iTrust, AquaLush and Connect between Release.

第四章　基于代码依赖关系的过时需求自动检测与更新推荐方法

4.1 引言

在软件维护和演化的过程中，一份最新的需求规约能够提供有价值的信息以辅助维护人员完成相应的软件活动。事实上，需求规约能够描述系统的主要功能与模块，解释系统实现背后的原理，帮助维护人员理解程序，并作为讨论功能更改的基础。然而，在软件演化时，维护人员通常不会及时更新需求规约。这是由于需求的更新需要付出大量的人工和时间成本。维护人员需要遍历全部的需求文档才能够定位其中需要被更新的部分，而全部的需求文档可能包含数百页的内容；并且维护人员需要收集和组织与变更有关的信息，以完成对需求的更新。因此，在面临软件维护或演化任务时，维护人员通常直接更新代码，而不更新相应的需求，很快，需求规约将会变得过时且失效。面对上述常见的软件活动场景，我们的工作关注于如何帮助维护人员减少检测和更新过时需求的成本。

当维护人员更新代码后，已有工作通过比较代码更新前后的差异，识别其中影响需求的变更代码元素，并利用信息检索技术计算变更代码元素与需求的相似性，从而发现近似候选过时需求。然而，系统的功能是由分布在系统间的代码协作完成的，变更部分只包含局部信息，与变更部分在结构上有依赖关系的上下文信息同样有价值。已有工作并没有充分考虑这一点。同时，对于检测得到的过时需求，维护人员可能对与变更有关的知识把握不足，需要额外进行代码分析以获得更充分的信息，已有工作没有考虑推荐与变更有关的知识以辅助维护人员完成对过时需求的更新。

为了解决上述问题，我们在对过时需求的检测中引入了对代码依赖关系的分析。代码自身除了文本信息，还包含结构信息。我们假设在结构上具有紧密依赖关系的代码元素间，在功能上也有较高的相似性。因此通过分析代码依赖，能够得到与变更代码元素功能相似的其他代码元素，以补充与代码变更有关的信息。同时，在与过时需求更新有关的变更代码元素中，如果一个变更代码元素的文本包含重要的概念，且此概念在代码依赖结构上反复出现，则该变更代码元素可以作为热点元素为维护人员提供与变更有关的知识。

本章中，我们提出一种基于代码依赖关系的过时需求自动检测方法，提高了过时需求检测时的精度，并推荐与过时需求更新相关的变更代码元素，以辅助

维护人员完成对需求的更新。

4.2 背景

在本节，我们将说明本工作的研究动机，并给出我们的问题定义。

4.2.1 研究动机

在本小节，我们将给出两个具体的例子以解释我们的研究动机。

图4.1 展示了 AquaLush 系统中一次真实的代码提交所涉及的代码变更，该提交为 AquaLush 系统新增了日志记录与展示的功能，图中 buildLogScrn() 是在本次提交中新增的函数，用于建立展示历史日志的面板，图中其余的函数在此次提交前就已经存在。在代码更新后，维护人员希望通过使用自动化工具，快速定位过时需求 SRS238（需求文本内容如图 4.3 所示）。文献 [ ] 中提出的过时需求检测方法，主要关注变更代码元素 buildLogScrn()，通过抽取该函数标识符等信息构造关键词文本，并利用信息检索技术将关键词文本与需求文本进行匹配。然而，由于信息检索技术存在词汇失配的问题，其检索的效果主要取决与检索文本的质量，只考虑变更代码元素可能导致文本的描述信息不足。例如，buildLogScrn() 中的 “Scrn” 为单词 “screen” 的缩写，因此无法与需求文本 SRSxxx 中的 “screen” 相匹配，从而使得检索精度下降。

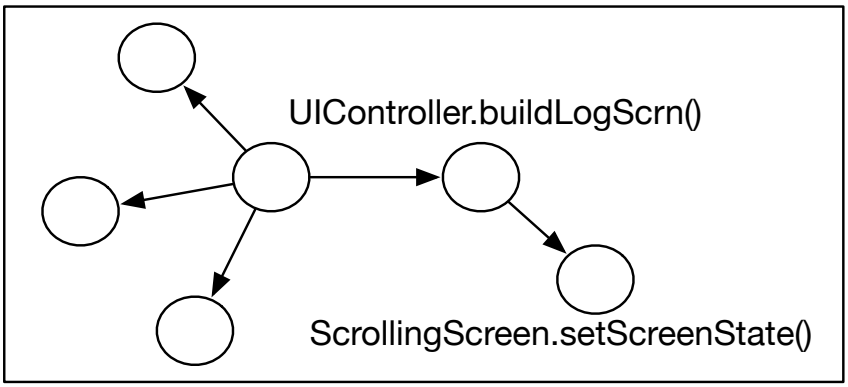


图 4.1: 与日志记录、展示功能有关的部分函数及其依赖结构

我们认为，与变更代码元素在结构上有紧密依赖关系的其他代码元素同样有价值，在这个例子中，函数 setScreenState() 与且只与函数 buildLogScrn() 存在函数调用依赖，因此我们认为这两个函数之间的关系紧密，可能用于实现相似的功能，函数 setScreenState() 的文本信息能够补充与变更有关的概念。在此

例中，函数 setScreenState() 包含了有效关键词 “screen” 和 “state”，这两个概念在 buildLogScrn() 中无法体现，从而补充了与变更有关的概念，达到提高检索精度的目的。

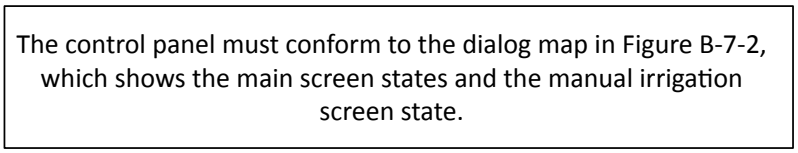


图 4.2: 需求文本 SRS238 的内容

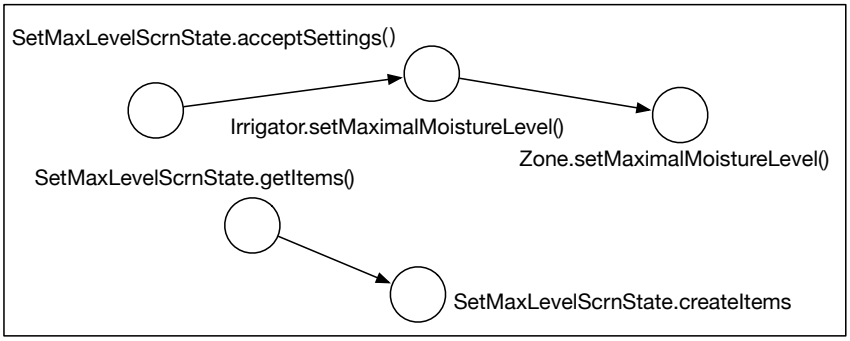


图 4.3: 与设置灌溉区域的湿度值上限功能有关的部分函数及其依赖结构

在浏览检索结果时，对于特定的某个候选过时需求，维护人员通常需要一些与候选过时需求相关的代码变更情况，从中获取与变更有关的知识，以完成对候选过时需求的更新。例如，维护人员希望获知在此次更新中，哪些关键函数对目标需求构成了影响，维护人员可以根据这些关键函数的行为判断目标需求应该如何更新。图二同样为 AquaLush 系统中一次真实的代码提交，该提交为 AquaLush 系统设置了灌溉区域的湿度值上限，图中的函数均为在此次提交中新增的函数。通过观察新增函数的文本信息和结构依赖我们发现，函数 Irrigator.setMaximalMoistureLevel() 与函数 Zone.setMaximalMoistureLevel() 协作完成了此次提交所描述的设置灌溉区域的湿度值上限这一主要行为，应该优先推荐给维护人员作为更新需求的辅助参考。与函数 getItems，createItems 相比，函数 setMaximalMoistureLevel 既包含与变更有关的重要概念，又此概念在新增函数的调用链上反复出现，因此我们认为具有这样特征的函数应该具有较高的权重，成为维护人员主要关注的函数。

4.2.2 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，同时给定新版本代码集合 NC 与旧版本代码集合 OC，分别包含 m，n 个代码实体（类），过时需求自动检测技术能够从 d 个需求文本中选择 k 个需求文本并排序，作为近似候选过时需求的排序，问题是:

Q1: 分析代码依赖关系是否能够提高过时需求自动检测方法的精度？

Q2: 对于给定的候选过时需求，能否推荐与该需求更新相关的变更代码元素？

4.3 方法概述

在本节，我们将介绍基于代码依赖关系的过时需求自动检测与更新推荐方法的流程。在维护人员完成代码提交后，我们将给定的新、旧版本代码，需求作为方法的输入，将近似候选过时需求的排序，辅以与候选过时需求相关的变更函数推荐作为输出。方法主要包含三个步骤（1）代码依赖关系构成的代码变更组识别；（2）基于变更组文本与需求文本相似性的过时需求排序生成；（3）基于代码依赖关系与文本信息的变更函数推荐。

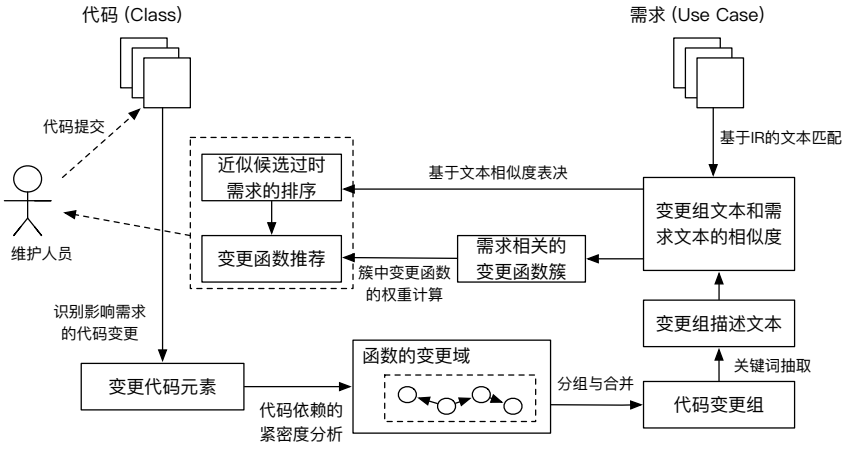


图 4.4: 基于代码依赖关系的过时需求自动检测与更新推荐方法流程

如图 N 所示，在步骤一中，方法通过比较代码元素的差异的方式识别出变更代码元素，这些变更代码元素被认为是影响需求的代码变更。然后，基于代码依赖关系的紧密度分析，能够获取与变更函数紧密依赖的其它函数元素，从而建立函数的变更域，并通过变更代码元素的分组与变更域的合并构成代码变更组；在步骤二中，对于每一个代码变更组，我们抽取变更组内代码元素的关键词

以构造变更组描述文本，并通过信息检索技术计算变更组描述文本与需求文本的相似度。然后，利用基于相似度值的表决算法生成近似候选过时需求的排序；在步骤三中，对于排序中的候选过时需求，我们能够发现与该需求有关的变更函数，然后根据函数调用依赖将变更函数划分为若干变更函数簇，并基于 TF-IDF 计算簇中变更函数的权重，从而生成变更函数推荐的排序。在后续章节中，将对上述三个步骤进行详细的阐述。

4.4 代码依赖关系构成的代码变更组识别技术

4.4.1 比较代码元素的差异

在软件演化的过程中，并非所有的代码变更都会影响需求。事实上，其中许多的代码变更与缺陷修复，重构或更改系统实现细节等方面有关。Eya 等人 [] 中提出：代码元素（包，类，函数，域）的新增或删除是影响系统外部行为，从而影响需求的代码变更。文献比较了新旧版本代码间的代码元素，以识别出新增或删除的代码元素，并过滤其中与重命名有关的变更，获得影响需求的变更代码元素。

以变更代码元素为基础，我们假定一个新增的包是有若干新增的类组成，一个新增的类是有若干新增的函数和域组成，对删除的代码元素也进行类似的处理。为描述变更代码元素，给定函数的变更集合 $CM = \{CM_1, \cdots, CM_n\}$，其中 CM_i 为新增或删除的函数，域的变更集合 $CF = \{CF_1, \cdots, CF_n\}$，其中 CF_i 为新增或删除的域。

4.4.2 构造变更组

4.4.2.1 生成 CDCGraph (Call Dependency with Closeness Graph)

我们利用 CDCGraph 以描述一个版本源代码中的函数调用依赖及其依赖的紧密度值。我们生成 CDCGraph 的算法包括以下两个步骤：

1. 建立函数的调用依赖：首先，我们定义图 CDGraph（Call Dependency Graph）为一个有序对 $CDGraph =< V, E >$，其中 V 是代码中函数的集合，E 是函数间产生的函数调用依赖集合。函数调用依赖是指，如果函数 A 调用了另一个函数 B，那么函数 A 依赖于函数 B。我们利用 Apache BCEL 库从编译后的字节码（Jar 文件）中捕获函数调用依赖，对于无法直接编译的项目，我们基于 JDT 实现了一个工具，它能够直接从项目的 Java 源文件中捕获函数的调用依赖。

2. 计算函数调用依赖的紧密度值：在 CDGraph 中，函数之间的调用依赖只有存在或不存在两种可能，存在的所有函数调用依赖都被认为是等价的，没有对依赖的紧密程度作出区分。事实上，当函数 A 调用函数 B 的时，可能意味着以下两种情况：函数 A 和函数 B 通过协作完成某个共同的任务，两者间具有较强的依赖关系；函数 A 通过调用函数 B 实现任务间的转移，两者间具有较弱的依赖关系。为了度量函数调用依赖，我们提出紧密度值以量化函数之间的相互作用程度。我们假定，对于函数调用依赖，如果调用者的出度和被调用者的入度越小，则该函数依赖越紧密。我们定义函数调用依赖的紧密度原则的计算公式如下：

$$Closeness_e = \frac{2}{OutDegree_{e.caller} + InDegree_{e.callee}} \tag{4.1}$$

其中，$OutDegree_{e.caller}$ 表示调用者的出度，$InDegreee.callee$ 表示调用者的入度。通过赋予 CDGraph 中的依赖边紧密度值的方式，可得 CDCGraph。

4.4.2.2 建立函数的变更域

对于变更的函数元素，我们考虑该元素的结构上下文信息，以补充与变更相关的知识。因此，我们将变更的函数元素及与其结构上紧密依赖的其他函数元素共同构成一个变更域。基于生成的 CDCGraph，我们给定一个阈值 k，将 CDCGraph 中所有紧密度值小于 k 的依赖边删去。剪枝后的 CDCGraph 被分解为若干个连通子图，每个连通子图表示一个区域，区域中的函数间被认为具有较强的相互作用程度。我们给定两个函数 β 和 γ：第一个函数 $\beta(CM_i)$，返回函数 CMi 在 CDCGraph 中所属的连通子图；第二个函数 $\gamma(G_i)$，返回图 G_i 中所包含顶点元素的集合。对于每个变更的函数元素，它对应的变更域可表示为：

$$CR_i = \gamma(\beta(CM_i)) \tag{4.2}$$

其中，CM_i 为集合 CM 中一个变更的函数元素。

4.4.2.3 分组策略

对于变更的函数元素，在建立其变更域后，如果孤立地考虑每一个变更域将会导致域的数量过多，并且每个变更域自身的信息可能仍不足以描述一个相对完整的系统功能。因此，我们将基于依赖关系对变更域进行合并。如果集合 CM 中的函数彼此之间存在函数调用依赖，则合并各个函数对应的变更域，合并后的域我们称之为一个变更组，给定集合 $CG = \{CG_1, \cdots, CG_n\}$ 以表示变更组的集合，其中 CG_i 表示合并后的变更组。另外的，如果变更的函数元素使用了集合

CF 中变更的域元素，则将被使用的域元素加入变更的函数元素所属的变更组。我们定义变更代码元素间的依赖关系图 $DG =< V, E >$，其中 V 表示变更代码元素，E 表示函数调用依赖或域的使用依赖。构造变更组的具体算法如下：

Algorithm 5: Change Groups Constructing

```
Input: DG, CM, CF, CR.
Output: change groups CG.
1  CG ← ∅;
2  forall the CM_i in CM do
3      if isAllocated(CM_i) = false then
4          methods M = DG.getConnectedGraphByCall(CM_i);
5          elements e ← ∅;
6          forall the m_i in M do
7              ChangeRegion CR_i = CR.findChangeRegion(m_i);
8              e.add(CR_i);
9          CG.add(e);
10         setAllocated(M);
11 forall the CG_i in CG do
12     forall the CM_i in CG_i do
13         fields F = DG.findFieldsUseage(CM_i, CF);
14         CG_i.add(F);
```

4.5 基于变更组文本与需求文本相似性的过时需求自动检测方法

4.5.1 抽取关键词以构造变更组的描述文本

在此小节，我们从变更组的代码元素中抽取关键词以构造变更组的描述文本。对于不同来源的代码元素，我们根据规则（见表 N）抽取与代码元素相关的标识符，注释等信息，可以注意到，与未变更的代码元素相比，我们会为变更代码元素抽取更多相关的信息。抽词规则确立的依据是，代码元素的标识符和注释信息通常反映了需求中与系统功能相关概念。同时，与未变更的代码元素相比，我们认为变更代码元素与概念的变更有更高的相关性。对于集合 CG 中的每一个变更组 CG_i，通过抽取关键词可以构造变更组描述文本 CGD_i，$CGD_i \in CGD$。

表 4.1: 变更组中代码元素的抽词规则

Element	Changed Type	Identifiers	Comments
Method	Added/Removed	Method, parameters, parent class	Method, parent class
	Unchanged	Method, parameters, parent class	No
Field	Added/Removed	Field, parent class	Parent class

4.5.2 基于信息检索技术生成近似候选过时需求的排序

对于给定的变更组文本集合 $CGD = \{CGD_1, \cdots, CGD_n\}$ 与需求集合 $R = \{R_1, \cdots, R_n\}$，我们通过以下两个步骤生成近似候选过时需求的排序：（1）利用信息检索技术计算变更组文本与需求文本的相似性；（2）利用基于文本相似度值的表决算法，生成近似候选过时需求的排序。我们将在本小节余下部分详细阐述每个步骤。

1. 利用信息检索技术的计算文本相似性：基于信息检索的文本匹配技术将两个文本集合作为输入，返回文本 - 文本的相似度矩阵作为输出。在本方法的场景下，我们计算的是变更组文本与需求文本间的相似性，因此，我们的输入包括变更组文本集合与需求文本集合，预期得到的输出是变更组文本 - 需求文本的相似度矩阵。在检索之前，我们需要对变更组文本及需求文本进行预处理。首先，对于词项来自代码的变更组文本，我们根据常用的代码命名模式（如驼峰命名法，下划线分割）进行分词操作。然后，我们对变更组文本和需求文本进行标准化处理，处理的过程包括除去特殊字符，词行还原，词根提取和去停用词。我们利用 $tf-idf$[11] 计算词项的权重，并利用向量空间模型计算文本相似度。

2. 基于文本相似度的表决算法：步骤一的结果是一个二维的相似度矩阵，我们基于该矩阵生成一个最终的排序列表，以向维护人员指明潜在的过时需求。我们基于文本相似度值计算最终排序列表的表决算法如下：对于每一个需求文本，累加其与变更组文本集合中各个变更组文本的相似度值，作为该需求的总得分。然后，将需求集合按需求的总得分倒叙排列，作为近似候选过时需求集合的最终排序，需求的排名越高代表该需求是过时需求的可能性越大。

4.6 基于代码依赖关系与文本信息的过时需求半自动更新推荐机制

在维护人员浏览近似候选过时需求的排序时，对于给定的某个候选过时需求，基于代码依赖关系与文本信息的过时需求半自动更新推荐机制能够推荐相关的变

更代码元素，以辅助维护人员完成对过时需求的更新。通常情况下，函数是用于表达系统某个功能实现的基本粒度。因此，本方法主要关注于对变更函数的推荐。

首先，对于给定的候选过时需求，我们基于 4.x.y 中的变更组文本 - 需求文本相似度矩阵以发现与该需求有相似性的变更组（两者的相似度值大于零），这些变更组所包含的变更函数构成候选变更函数集合 $CCM = \{CCM_1, \cdots, CCM_n\}$。然后，我们将集合 CCM 中的变更函数根据权重排序以推荐给维护人员。变更函数的排序过程分为以下两个步骤：

1. 基于代码依赖关系构造变更函数簇：利用 4.x.x 中变更代码元素间的依赖关系图 DG，我们将变更函数根据函数调用依赖划分为若干变更函数簇，其中每个变更函数簇是一个连通的函数调用依赖图，我们用集合 CMC 表示变更函数簇的集合，其中 CMC_i 由变更函数构成。

2. 基于 TF-IDF 的变更函数权重计算：我们假定变更函数的权重是基于变更函数（标识符）中所包含词的权重计算而得。因此，我们首先根据代码的文本信息计算变更函数中词的权重。我们采用 TF-IDF 加权技术以评估词的重要性，我们同时考虑词对于变更函数簇的重要性（局部重要性）和词对于代码库的重要性（全局重要性）。我们认为，如果某个词在变更函数簇中出现的频率高，并且在代码库中很少出现，则认为此词与变更概念具有较高的相关性，包含该词的变更函数适合推荐给维护人员，以提供与变更概念有关的知识。

   为此，我们首先构造变更函数簇文本集和代码库文本集。对于每一个变更函数簇，我们抽取簇中变更函数的标识符（包括函数名，及函数所属类的类名）构成变更函数簇文本。同时，我们提取更新前代码库中的所有函数，每个函数作为一个文本，文本内容同样为函数的标识符。然后，利用构造的变更函数簇文本集和代码库文本集，基于 TF-IDF 计算变更函数簇中词的权重。TF-IDF 的计算公式如下：

$$tf_{i,j} = \frac{n_{i,j}}{\sum_k n_{k,j}} \tag{4.3}$$

$$idf_i = \log \frac{|D|}{\left|j : t_i \in d_j\right|} \tag{4.4}$$

$$tfidf_{i,j} = tf_{i,j} \times idf_i \tag{4.5}$$

其中，tf 指给定的某一个词在文本中的出现频率。$n_{i,j}$ 为词 t_i 在文本 d_j 中的出现次数，分母表示在文本 d_j 中所有词的出现次数之和；idf(inverse document frequency) 用于度量词的普遍重要性，$|D|$ 表示语料库中的文本总数，表示包含词 t_i 的文本数目。在本方法的场景中，d_j 对应变更函数簇文本，语料库 D 包括变更函数簇文本和从代码库中提取的函数文本。最终，变更函数的权重由该函数标识符所包含词权重的算术平均确定，其中词的权重为该词在此变更函数所属的变更函数簇文本中的权重。我们将集合 CCM 中的变更函数根据权重倒叙排列，作为变更函数推荐的排序。

4.7 实验与分析

在此小节，我们通过实验以验证本方法的有效性。接下来，将具体阐述我们的实验设置及实验结果与分析。

4.7.1 实验目标与评价指标

我们的实验目的是为了分析基于代码依赖关系的过时需求自动检测方法是否能够减少维护人员发现过时需求的成本。为了达到这一目的，我们需要回答 4.2.1 中定义的如下两个研究问题。

Q1: 分析代码依赖关系是否能够提高过时需求自动检测方法的精度？

这个研究问题与我们方法的第一部分有关，在 x.x 节中，我们验证了基于代码依赖关系的过时需求自动检测方法的有效性。

Q1 的评价指标：Q1 是关于从整个需求集合里检测某一特定的子集（过时需求），因此我们采用信息检索中著名的两个指标 $Average\ Precision\,(AP)$ 和 $F-measure$ 以评价近似候选过时需求排序的质量。

$$AP = \frac{\sum_{r=1}^{N}\left(Precision\,(r) \times isRelevant\,(r)\right)}{|RelevantDocuments|} \tag{4.6}$$

其中，r 表示目标文本在有序文本列表中的排序，$Precision\,(r)$ 表示前 r 个排序的准确率，$isRelevant\,()$ 是一个二值函数，如果文本相关返回 1，无关则返回 0，N 表示文本的总数。同时，$F-measure$ 是准确率和召回率的调和平均值，计算方式如下：

$$F = \frac{2}{\frac{1}{R} + \frac{1}{P}} \tag{4.7}$$

其中，R 表示召回率，P 表示准确率。我们通过统计显著性检验以验证我们方法能够在不同的截止排序 n 上提高其 $F-measure$ 值。由于过时需求的总数

对于每个检测方法是一致的，因此我们采用 Wilcoxon 秩和检验 [] 以检验如下的两个零假设（null hypothesis）：

H_0：在代码提交版本间，分析代码依赖关系没有显著的提高过时需求需求自动检测方法的精度。

H_1：在代码发布版本间，分析代码依赖关系没有显著的提高过时需求需求自动检测方法的精度。

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。除了 $p-value$ 以外，我们关注的另一方面是在实践中不同方法精度间的差异幅度。为此，我们采用 *Cliff's Delta d*[68]，一种应用于序数的无参效果量（*effective size*）来衡量实验组与对照组的差异（$p-value$ 表明显著性是否存在，*effective size* 表明实践的显著程度）。*Cliff's Delta* 的范围在-1 和 1 之间，$d < 0.33$ 表示显著性较小，$0.33 \leq d < 0.474$ 表示显著性一般，$d \geq 0.474$ 表示显著性较大。

Q2: 对于给定的候选过时需求，能否推荐与该需求更新相关的变更代码元素？

这个研究问题与我们方法的第二部分有关，在 x.x.x 节中，我们验证了基于代码依赖关系和文本信息的变更函数推荐方法的有效性。

Q2 的评价指标：Q2 是关于从所有变更函数里选取某一特定的子集，并将选取的变更函数按权重排序推荐给维护人员，通常推荐排序中的每一项（变更函数）会根据其与需求更新的相关程度而被打分。因此我们采用两个衡量排序质量的指标 *DCG*（Discounted cumulative gain）和 *NDCG*（Normalized Discounted cumulative gain）用以评价我们的方法。使用 DCG 指标时有两个基本假设条件：（1）在排序列表中，相关性越高的结果排序越靠前越有益。（2）在对排序列表中的每一项标注时，评分等级高的结果比等级低的结果有益。我们的评价场景符合上述两个基本假设条件。*DCG* 的核心思想是如果评分等级较高的项却排序较后，那么在计算排序的得分时就需要对该项的得分有所惩罚，DCG 的公式如下：

$$DCG_p = rel_1 + \sum_{i=2}^{p} \frac{rel_i}{\log_2(i)} \tag{4.8}$$

其中 rel_i 表示排序为 i 项的评分。为了便于不同类型查询所得到的结果之间横向比较，并衡量当前排序与理想排序的差异，*NDCG* 通过将当前结果排序的 DCG 值除以该查询结果的理想值 *IDCG*（Ideal DCG）的方式进行归一，公式为：

$$NDCG_p = \frac{DCG_p}{IDCG_p} \tag{4.9}$$

在实际中，标定 *IDCG* 值的操作较为困难，我们在 4.4.4 节中给出了标定变更函数推荐排序 *IDCG* 值的方式。

4.7.2 研究案例与实验设置

1. iTrust

   我们第一个研究案例是医疗服务项目 iTrust[16]，iTrust 是一个医疗数据管理系统，它是由北卡罗莱纳州立大学开发并维护的一个用于教学目的的系统。iTrust 包含多个发布的代码版本和一个基于 wiki 的规约说明，规约说明中包括功能性需求，非功能性需求，术语表等信息。iTrust 是由 Java 和 Java Server Page 组合开发的 Web 应用。在我们的实验中，需求部分我们选用 iTrust 中的 39 个功能性需求，每个功能性需求是详细且定义良好的用例 (use case)。在代码部分，由于我们的原型工具暂时只支持对 Java 代码的分析，因此我们只考虑 iTrust 中 Java 部分的源码。我们使用 iTrust 版本 10（发布日期：2010.8.18）和版本 11（发布日期：2011.1.7）作为两个发布的代码版本。Eya 在文献 [] 中通过人工比较这两个版本源代码的方式，发现了 14 个概念变更，并识别了这些变更所影响的需求。我们选取了其中 10 个影响需求的变更作为代码提交。

   表 N 中展示了我们验证 Q1 时所选取的代码变更，变更的描述，以及变更所影响的需求。同时，为了验证 Q2，对于在一次变更中特定的某个过时需求，我们将除去过时需求包含语料后的变更的描述作为变更概念的文本，计算推荐的变更函数标识符中，命中变更概念文本中词的准确率，作为该变更函数的评分。根据变更函数的评分，可以计算推荐变更函数排序的 DCG 值。将变更函数按评分倒叙排列，可以得到变更函数推荐的理想排序，并标定 *IDCG* 值。由于 iTrust 中代码提交的描述信息较少，存在变更函数排序中的函数无法命中变更概念文本中词的情况，因此在实验中我们并不比较此类代码提交。

2. AquaLush

   我们第二个研究案例是由软件控制的灌溉管理系统 []，可以基于用户提供的湿度、用水量等参数自动调整灌溉输出，AquaLush 最初作为一个解释性示例在《软件设计导论》[14] 一书中被提出。AquaLush 由 Java 语言实现，代码量在 1.1 万行左右，同时，AquaLush 包含自然语言描述的结构化需求规约。网页 [] 包含 AquaLush 的代码，需求和基线等相关。在需求的部分，我们的实验提取了需求规约中文本格式的 337 个需求说明作为需求

表 4.2: iTrust 系统中的代码变更及变更所影响的需求

Change	Change description	Impacted use cases
Change 1	Reason code	UC15, UC37
Change 2	Uploading photo	UC4
Change 3	Remote monitoring (added height, weight, etc.)	UC34
Change 4	Remote monitoring (get patient data by type)	UC34
Change 5	Display patient's monitoring HCP	UC34
Change 6	Office visit form (added orc and comment)	UC11
Change 7	Enabling appointment editing	UC22
Change 8	Login (added captcha and attempts)	UC3
Change 9	Weight/height charting	UC10
Change 10	Logging (added logs)	UC5

规约集合；在代码的部分，Eya 在文献 [] 中开发了 AquaLush 的另一个版本，相比原版本，新版本包含了 8 次代码提交（3 次与系统功能有关和 5 次与缺陷修复有关）。我们选取了 3 次与系统功能有关的代码提交，并将上述两个版本作为两个代码的发布版本。表 N 中展示了我们验证 Q1 时所选取的代码变更，变更的描述，以及变更所影响的需求。我们同样采取计算推荐的变更函数标识符中，命中变更概念文本中词的准确率作为变更函数的评分以验证 Q2。

3. Connect

我们第三个研究案例是用于医疗机构间交换医疗数据的开源系统 Connect[]。Connect 项目包含大量相关的文档，如设计文档，使用手册，问题追踪，变更请求，发布记录，各代码版本的需求，需求跟踪矩阵等。Connect 由 Java 语言实现，并包含历史提交信息。在本实验中，我们采用了主项目部分的 Java 代码，它位于代码仓库的 product/production 目录下，代码量在 28 万行左右。在需求的部分，Eya 在文献 [] 中采用了需求跟踪矩阵及发布记录中 Jira 单号，其中需求跟踪矩阵关联了需求与 Jira 问题单号的关系，通过它们之间的关联，发现与相应需求有关的 26 次代码提交，我们我们选取了其中 11 个影响需求的代码提交。我们选取了由于 Connect 中没有提供明确的变更描述，因此我们在验证 Q2 时，只选用了 iTrust 和 AquaLush 系统。

表 4.3: AquaLush 系统中的代码变更及变更所影响的需求

Change	Change description	Impacted use cases
Change 1	Add a maximum moisture level: the irrigation should start when the moisture level is lower than the critical moisture level and should stop as soon as the maximal moisture level is reached. The default max level should be 50.	SRS25, SRS26 SRS28, SRS53 SRS69, SRS359
Change 2	Allow setting the water allocation for each of the zones separately	SRS29, SRS53 SRS153, SRS186 SRS32, SRS378 SRS379, SRS382
Change 3	Create a log that includes the timestamp for each of the following events: change irrigation mode, setting water allocation, setting irrigation time, setting critical, or maximum moisture level. A button show log allows the user the access the log.	SRS238, SRS262 SRS266

4.7.3 实验结果与结果分析

在本节中，以两个研究问题为指导，我们阐述了三个研究案例的实验结果并给予分析。

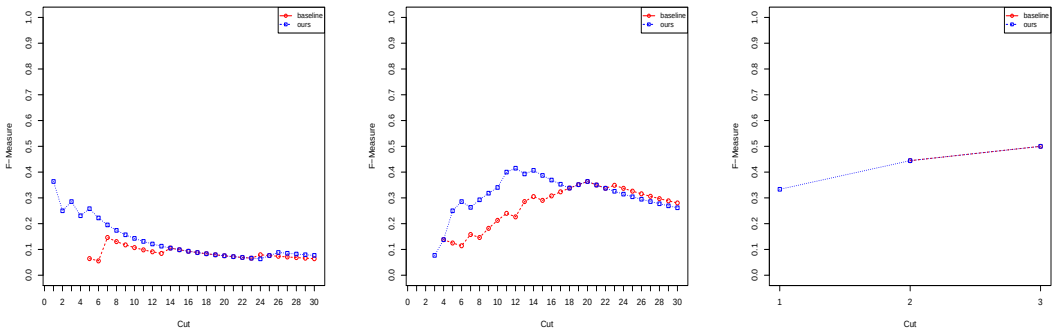
4.7.3.1 过时需求检测（Q1）

为了回答 Q1，我们在函数调用依赖的紧密度阈值 k 0.3 下进行了实验。我们检验了在前 n 个排序时的 AP 和 $F-Measure$ 值，图 N 的 $F-Measure/Cut$ 曲线展示了选取前 1 至 30 个排序时，代码提交版本间三个研究案例的 $F-Measure$，其中红线为基线方法，蓝线。图 N 为代码发布版本间的 $F-Measure/Cut$ 曲线。表 N 展示了 AP 与 $Wilcoxon$ 秩和检验结果。

iTrust：在比较代码提交版本间的过时需求检测结果时，10 个代码提交中的 5 个（Change3，4，5，8，9）在两者方法下都能取得 100% 的 AP 值，故我们主要关注在剩余的 5 个代码提交中，两者方法的表现。如图 N 中的 $iTrust_{Commit}$ 所示，我们方法的 $F-Measure/Cut$ 曲线能够明显覆盖基线方法的曲线，并且 AP 值从 5.57% 提升至 25.13%（提升约 20%，见图 N）。$F-Measure/Cut$ 曲线与基

表 4.4: 平均正确率与 Wilcoxon 秩和检验结果

System	Version	Average Precision	p-value	Cliff's Delta
iTrust	commits	25.13% (+20.00%)	< 0.01	0.49 (large)
	releases	66.55% (+31.87%)	< 0.01	0.73 (large)
AquaLush	commits	22.76% (+4.68%)	0.05	0.29 (small)
	releases	27.65% (+1.58%)	< 0.01	0.46 (medium)
Connect	commits	42.06% (+9.29%)	1	0.11 (negligible)
	releases	23.04% (+2.30%)	0.02	0.36 (medium)



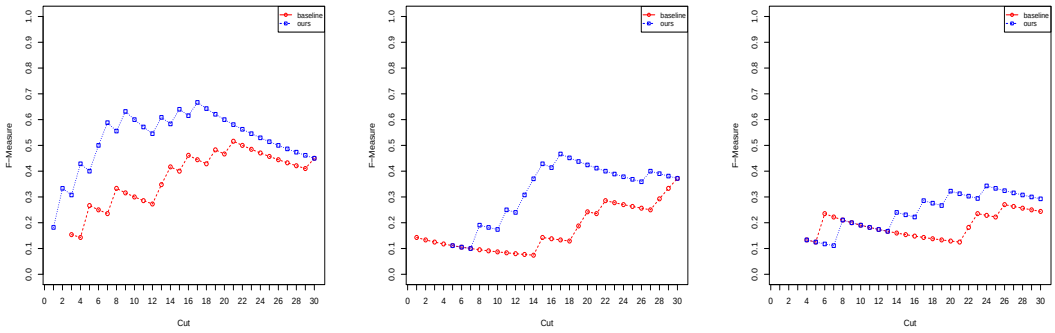
(a) $iTrust_{Commits}$　(b) $AquaLush_{Commits}$　(c) $Connect_{Commits}$

图 4.5: F-measure/cut curves for iTrust, AquaLush and Connect between Commits.

线相比，$p-value < 0.01$，故拒绝零假设 H_0，表明分析代码依赖关系能够明显提高过时需求自动检测方法的精度，在 $Cliff'sDelta$ 下为高效果量。对于代码发布版本间的过时需求检测结果的比较，我们方法的 $F-Measure$ 值仍能够明显覆盖基线方法，并且 AP 值从 34.68% 提升至 66.55%（提升约 31.87%，见图 N），$p-value < 0.01$，故拒绝零假设 H_1，在 $Cliff'sDelta$ 下为高效果量。因此对于 iTrust 系统，分析代码依赖关系能够明显提高过时需求自动检测方法的精度。

AquaLush：在比较代码提交版本和代码发布版本的过时需求检测结果时，我们方法的 $F-Measure/Cut$ 曲线值能够明显覆盖基线方法的曲线，AP 值分别提高了 4.68% 和 1.58%，$p-value$ 值均 $<= 0.05$，故拒绝零假设 H_0，H_1。对于 AquaLush 系统，分析代码依赖关系也能够提高过时需求自动检测方法的精度。

Connect：在比较代码提交版本间的过时需求检测结果时，11 个代码提交中的 8 个在两者方法下都能取得 100% 的 AP 值，故我们主要关注在剩余的 3 个



(a) $iTrust_{Releases}$　(b) $AquaLush_{Releases}$　(c) $Connect_{Releases}$

图 4.6: F-measure/cut curves for iTrust, AquaLush and Connect between Release.

代码提交中，两者方法的表现。如图 N 中的 $Connect_{Commit}$ 所示，这 3 个代码提交能够在前 3 个排序检测出所有的过时需求，AP 值从 32.78% 提升至 42.06%（提升约 9.29%），由于 $p-value > 0.05$，因此无法拒绝零假设 H_0。其主要的原因在于，与 Connect 的三个代码提交有关的过时需求数量较少，且基线方法已经能够较为准确地检测出所有过时需求（过时需求均在前 3 排序内被发现），因此该实验环境下我们的方法对检测效果的提升空间非常有限。尽管如此，我们的方法在检测时仍然将与代码提交 Change841 有关的过时需求 GATEWAY-841 提高了一个排名（从第 2 提升至第 1）。对于代码发布版本间的过时需求检测结果的比较，我们方法的 $F-Measure$ 值在大部分情况下能够明显覆盖基线方法，并且 AP 值从 20.74% 提升至 23.04%（提升约 2.3%，见图 N），$p-value < 0.01$，故拒绝零假设 H_1，在 $Cliff'sDelta$ 下为中效果量。因此对于 Connect 系统，分析代码依赖关系能够明显提高代码发布版本间过时需求自动检测方法的精度。

结合上述三个研究案例的结果与分析发现，我们的方法提高了近似候选过时需求排序的 AP 与 F-Measure，并通过了 6 个显著性测试中的 5 个，可以认为基于代码依赖关系的分析能够显著提高过时需求自动检测的精度。

4.7.3.2　**变更函数推荐（Q2）**

为了回答 Q2，对于与某一次代码提交相关的过时需求，我们检验了前 10 个推荐的变更函数排序的 DCG 与 $NDCG$ 值。图 N 展示了 iTrust 与 AquaLush 这两个研究案例中各个代码提交下，对于不同过时需求的变更函数推荐排序的平均 DCG 与 $NDCG$ 值，并与传统的字母表排序进行了比较。在计算变更函数推荐排序的 DCG 与 $NDCG$ 前，我们首先需要为排序中的变更函数标定评分（评分标定过程详见 x.x.x 节），将变更函数按评分倒叙排列，可以得到变更函数推荐

表 4.5: AquaLush 变更函数推荐排序的 DCG 与 NDCG 值

Project	Change	Rank	DCG	NDCG
AquaLush	Change1	Alphabet	0.66	0.77
		Ours	0.68	0.81
	Change2	Alphabet	0.12	0.22
		Ours	0.39	0.76
	Change3	Alphabet	1.18	0.87
		Ours	1.09	0.80

的理想排序及 *IDCG* 值。

AquaLush: 在与字母表排序的比较中, 对于 Change1, 我们的方法将 NDCG 值从 0.77 提升至 0.81（提升约 0.04，见图 N）。对于 Change2，我们的方法将 NDCG 值从 0.22 提升至 0.76（提升约 0.54）。对于 Change3, 我们方法的 NDCG 值下降了 0.03，通过分析发现，这与 Change3 中变更函数的命名方式和变更描述的质量有关。Change3 中的变更函数，如 getItems，createItems，getEvent 等包含的语义概念较为普遍，缺少独特性，表达的是与底层实现相关的细节概念，而变更描述文本更多的是在阐述与功能相关的概念，两者由于表达层次的不同，存在词汇失配的问题（Vocabulary Mismatch），此时我们用于标定变更函数评分的方式无法准确评价变更函数的重要性，从而影响评价指标的效果。

iTrust：在与字母表排序的比较中，我们的方法在多数情况下（5/7）能够提高变更函数排序的 NDCG 值（提高 0.06 ~ 0.37），如图 N 所示。

结合上述两个研究案例的结果与分析发现, 对于给定的过时需求, 我们的方法能够推荐与该需求更新相关的变更函数，且在大多数实验结果中（8/10），变更函数推荐的排序比字母表排序有更高的质量。

4.8 本章小结

在本章中，我们提出一种基于代码依赖关系的过时需求自动检测方法，提高了过时需求检测时的精度，并推荐与过时需求更新相关的变更代码元素，以辅助维护人员完成对需求的更新。最后，我们在三个研究案例下设计并完成实验，验证了我们方法的有效性。

表 4.6: iTrust 变更函数推荐排序的 DCG 与 NDCG 值

Project	Change	Rank	DCG	NDCG
iTrust	Change1	Alphabet	1.33	0.66
		Ours	1.33	0.72
	Change2	Alphabet	0.93	0.75
		Ours	1.11	0.90
	Change3	Alphabet	0.33	0.35
		Ours	0.68	0.73
	Change6	Alphabet	0.25	0.63
		Ours	0.37	0.94
	Change7	Alphabet	0.82	0.85
		Ours	0.78	0.81
	Change8	Alphabet	0.80	0.78
		Ours	0.98	0.96
	Change10	Alphabet	0.44	0.76
		Ours	0.58	1.0

第五章　总结与展望

5.1 工作总结

5.2 研究展望

简历与科研成果

基本情况 聂佳，男，汉族，1990 年 12 月出生，江苏省苏州市人。

教育背景

2013.9 ~ 2016.6　南京大学计算机科学与技术系　　硕士

2009.9 ~ 2013.6　南京大学金陵学院计算机科学与技术　　本科

攻读硕士学位期间发表的论文

1. Yu Du, Hao Hu, Wei Song, Junhua Ding and Jian Lü. Efficient Computing Composite Service Skyline with QoS Correlations. In Proceedings of International Conference on Services Computing, 2015, Accepted.

2. Yu Du, Hao Hu, Wei Song, Yuhao Gong and Jian Lü. Safety-Range Aware Mobile Composite Service Skyline. In Proceedings of International Conference on Mobile Services, 2015, Accepted.

攻读硕士学位期间申请的专利

1. 胡昊，宋巍，葛季栋，吕建，**杜宇**，“一种计算具有 QoS 关联关系的 QoS 最优组合服务限定范围的方法”，申请号：201510168726.0，已授权。

攻读硕士学位期间参与的科研课题

1. 国家 973 计划：持续演进的自适应网构软件模型、方法及服务质量保障。项目编号 No.2015CB352202。

参考文献

[1] Orlena CZ Gotel and Anthony CW Finkelstein. An analysis of the requirements traceability problem. In Requirements Engineering, 1994., Proceedings of the First International Conference on, pages 94–101. IEEE, 1994.

[2] Patrick Mader and Alexander Egyed. Assessing the effect of requirements traceability for software maintenance. In Software Maintenance (ICSM), 2012 28th IEEE International Conference on, pages 171–180. IEEE, 2012.

[3] Balasubramaniam Ramesh, Timothy Powers, Curtis Stubbs, and Michael Edwards. Implementing requirements traceability: a case study. In Requirements Engineering, 1995., Proceedings of the Second IEEE International Symposium on, pages 89–95. IEEE, 1995.

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

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

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

[7] 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.

[8] Marc Eaddy, Alfred V Aho, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. Cerberus: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. In Program Comprehension, 2008.

ICPC 2008. The 16th IEEE International Conference on, pages 53–62. IEEE, 2008.

[9] Tathagata Dasgupta, Mark Grechanik, Evan Moritz, Bogdan Dit, and Denys Poshyvanyk. Enhancing software traceability by automatically expanding corpora with relevant documentation. In 2013 IEEE International Conference on Software Maintenance, pages 320–329. IEEE, 2013.

[10] Nawazish Ali, Zohreh Sharafl, Y Gueheneuc, and Giuliano Antoniol. An empirical study on requirements traceability using eye-tracking. In Software Maintenance (ICSM), 2012 28th IEEE International Conference on, pages 191–200. IEEE, 2012.

[11] Nawazish Ali, Yann-Gael Gueneuc, and Giuliano Antoniol. Trustrace: Mining software repositories to improve the accuracy of requirement traceability links. Software Engineering, IEEE Transactions on, 39(5):725–741, 2013.

[12] Ali Mahmoud and Nan Niu. Supporting requirements traceability through refactoring. In Requirements Engineering Conference (RE), 2013 21st IEEE International, pages 32–41. IEEE, 2013.

[13] Giovanni Capobianco, Andrea De Lucia, Rocco Oliveto, Annibale Panichella, and Sebastiano Panichella. Improving ir-based traceability recovery via noun-based indexing of software artifacts. Journal of Software: Evolution and Process, 25(7):743–762, 2013.

[14] David Diaz, Gabriele Bavota, Andrian Marcus, Rocco Oliveto, Satoshi Takahashi, and Andrea De Lucia. Using code ownership to improve ir-based traceability link recovery. In Program Comprehension (ICPC), 2013 IEEE 21st International Conference on, pages 123–132. IEEE, 2013.

[15] Collin McMillan, Denys Poshyvanyk, and Meghan Revelle. Combining textual and structural analysis of software artifacts for traceability link recovery. In Traceability in Emerging Forms of Software Engineering, 2009. TEFSE'09. ICSE Workshop on, pages 41–48. IEEE, 2009.

[16] 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 Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on, pages 199–208. IEEE, 2013.

[17] 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.

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

[19] Michael W Berry. Large-scale sparse singular value computations. International Journal of Supercomputer Applications, 6(1):13–49, 1992.

[20] Thomas K Landauer and Susan T Dumais. A solution to plato's problem: The latent semantic analysis theory of acquisition, induction, and representation of knowledge. Psychological review, 104(2):211, 1997.

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

[22] Susan T Dumais et al. Latent semantic indexing (lsi) and trec-2. NIST SPECIAL PUBLICATION SP, pages 105–105, 1994.

[23] Thomas M Cover and Joy A Thomas. Elements of information theory. John Wiley & Sons, 2012.

[24] Nicolas Anquetil and Timothy Lethbridge. Assessing the relevance of identifier names in a legacy software system. In Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative Research, page 4. IBM Press, 1998.

[25] Giuseppe Scanniello, Andrian Marcus, and Daniele Pascale. Link analysis algorithms for static concept location: an empirical assessment. Empirical Software Engineering, 20(6):1666–1720, 2015.

[26] Keith H Bennett and Václav T Rajlich. Software maintenance and evolution: a roadmap. In Proceedings of the Conference on the Future of Software Engineering, pages 73–87. ACM, 2000.

[27] Timothy C Lethbridge, Janice Singer, and Andrew Forward. How software engineers use documentation: The state of the practice. Software, iEEE, 20(6):35–39, 2003.

[28] Eya Ben Charrada, Anne Koziolek, and Martin Glinz. Supporting requirements update during software evolution. Journal of Software: Evolution and Process, 27(3):166–194, 2015.

[29] Martin Glinz. On non-functional requirements. In Requirements Engineering Conference, 2007. RE'07. 15th IEEE International, pages 21–26. IEEE, 2007.

[30] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In Soviet physics doklady, volume 10, pages 707–710, 1966.

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

[32] Wei Zhao, Lu Zhang, Yin Liu, Jiasu Sun, and Fuqing Yang. Sniafl: Towards a static noninteractive approach to feature location. ACM Transactions on Software Engineering and Methodology (TOSEM), 15(2):195–226, 2006.

[33] Gabriele Bavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto. Automating extract class refactoring: an improved method and its evaluation. Empirical Software Engineering, 19(6):1617–1664, 2014.

[34] 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.