


# 构建有效软件重用存储库的渐进方法

SCOTT HENNINGER

University of Nebraska—Lincoln

---

 软件重用存储库面临两个相互关联的问题：（1）获取知识以初步构建存储库；（2）修改存储库以满足软件开发组织不断发展和动态的需求。当前的软件存储库方法严重依赖于分类，这加剧了获取和演化问题，因为在有效使用存储库之前，需要进行昂贵的分类和领域分析工作。本文概述了一种通过选择一种检索方法来避免这些问题的方法，该方法利用最小的存储库结构来有效地支持查找软件组件的过程。该方法通过一对概念验证原型进行了演示：PEEL 是一种半自动识别可重用组件的工具，CodeFinder 是一种检索系统，通过扩展激活检索过程来弥补显式知识结构的不足。CodeFinder 还允许在用户搜索信息时修改组件表示。此机制适应存储库中不断变化的信息性质，并在人们使用存储库时逐步改进存储库。这些技术的结合为设计软件存储库提供了潜力，可以最大限度地降低前期成本，有效地支持搜索过程，并随着组织不断变化的需求而发展。

General Terms: Design

Additional Key Words and Phrases: Component repositories, information retrieval, software reuse

---

## 1. INTRODUCTION

随着可重用软件组件库的不断增长，从软件库中检索组件的问题引起了软件重用社区的关注 [Burton et al. 1987; Devanbu et al. 1991;

---

This research was supported in part by grant MDA903-86-C0143 from the Army Research Institute.

Author's address: Department of Computer Science and Engineering, 115 Ferguson Hall, CC 0115, University of Nebraska-Lincoln, Lincoln, NE 68588-0115; email: scotth@cse.unl.edu.

Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.

© 1997 ACM 1049-331X/97/0400-0111 \$03.50

Frakes and Gandel 1990; Frakes and Nejme 1987; Frakes and Pole 1994; Maarek et al. 1991; Prieto-Díaz and Freeman 1987; Sommerville and Wood 1986]. 尤其是在基于组件的重用方法中, 开发人员将软件部件组合到应用程序中, 组件库是实现软件重用所必需的。基于组件的软件重用面临着一个固有的困境: 为了使该方法有用, 存储库必须包含足够的组件来支持开发人员, 但当有许多示例可用时, 查找和选择合适的示例变得很麻烦。为了解决查找相关组件的问题, 已经采用了各种检索技术, 如枚举分类[Booch 1987], 剖面分类法[Prieto-Díaz 1985; Prieto-Díaz and Freeman 1987], 基于框架的分类[Ostertag et al. 1992], 自由文本索引[Frakes and Nejme 1987], 关系数据库[Burton et al. 1987], and 规格说明法[Chen et al. 1993] have been employed to address the problem of finding relevant components. 但是, 涉及如何构建、填充和演化有效的存储库以满足开发组织不断变化的需求的问题受到了相当少的关注。

大多数检索算法要求在设计人员能够有效搜索存储库之前, 有一个预定义的结构。至少, 组件在放入存储库之前必须进行分类。这种构建软件存储库的增值方法造成了许多组织无法克服的实际和智力资本投资障碍[Biggerstaff and Richter 1987]. 大多数开发组织都面临交付特定产品的压力, 无法承担对单个项目中创建的工作产品进行概括和分类所需的额外工作。需要的方法能够以最小的索引和结构化工作提供足够的检索效率, 使组织能够利用从以前的开发工作中积累的宝贵资产, 而无需大量的前期投资。

除了成本高昂之外, 检索所需的结构通常是静态的, 无法适应动态的开发环境。一旦创建了结构, 它们就会一成不变, 只能通过复杂的过程进行更改, 通常需要重新设计整个存储库。这是一个重要的问题, 不仅因为事实证明, 要在第一时间将结构正确化非常困难, 而且因为该领域在不断变化的技术、项目动态和客户需求的流动性的压力下不断发展。

**本文的目的有两个:** 描述软件存储库的索引和检索问题, 并提供一组旨在支持组件存储库增量优化的工具。在我们的方案中, 初始组件表示是以最小的结构化工作创建的, 允许以最小的前期工作构建存储库。然后使用检索技术来补偿最小的检索结构, 并帮助开发人员找到可重用的组件。当开发人员使用存储库时, 采用自适应索引技术来增强检索结构, 并使存储库朝着表示其用户对开发问题进行推理的方式发展

以下各节首先概述了软件组件检索方法的当前研究。介绍了构建存储库的进化方法，然后介绍了PEEL和CodeFinder的存储库种子、检索和自适应索引功能，这是一种实验原型，旨在研究创建和维护软件存储库的问题。最后，我们基于CodeFinder有限的实证研究对未来的工作进行了讨论。

## 2. 使用组件存储库支持软件重用

有两种基本技术已应用于软件重用方法中：基于组合的重用（也称为基于部件或基于组件的重用）和生成技术（基于语言的重用）[Biggerstaff and Richter 1987; Frakes and Pole 1994]. 在组合技术中，组件是自包含的实体，例如库例程、数据结构、程序、对象等。重用这些组件是通过开发人员发现、理解这些组件并使其适应新的应用程序来实现的。用于编程语言的Unix管道和代码库就是组合技术的示例。生成方法与编译器技术密切相关，在编译器技术中，用户选择调用参数化代码片段来创建定制的应用程序。虽然生成性技术可能存在库问题【Batory等人，1994年】，但对于组件技术来说，问题更为关键，因为必须存在足够的组件才能使存储库有用。

### 2.1 组件存储库

存储库的结构是获得良好检索结果的关键。无论匹配算法多么“智能”，如果组件被索引或结构不良，则很难获得良好的检索性能。直观且广泛接受的假设是，在构建存储库方面的前期投资会使组件重用的简易性按比例增加[Barnes and Bollinger 1991, p. 16]. 虽然这一假设有些道理，但已经创建了可以有效使用低成本存储库结构的检索方法。信息检索技术在30多年的研究中得到了迅速发展[Salton and McGill 1983], 但用于软件存储库的检索方法可分为三类：枚举分类、分面和自由文本索引[Frakes and Gandel 1990]. 超文本系统也已被使用，尽管某种形式的检索对于使超文本在大型存储库中有效是必要的[Halasz 1988; Thompson and Croft 1989].

枚举分类是杜威十进制系统和ACM计算评论分类系统使用的一种著名检索方法。在这种方法中，信息被放置在通常以子类别层次结构构成的类别中，这与Unix文件系统非常相似。

ACM Transactions on Software Engineering and Methodology, Vol. 6, No. 2, April 1997.

分类方案的吸引力在于能够迭代地将信息空间划分为更小的部分，从而减少需要仔细阅读的信息量。使用枚举分类涉及的问题包括其固有的不灵活性和理解大型层次结构的问题。使用枚举分类涉及的问题包括其固有的不灵活性和理解大型层次结构的问题。一些领域将适合许多小型类。其结果是，不熟悉其结构的用户将迷失在可能的类的泥沼中 [Halasz 1988]。其他领域的类别很少，但必须包含许多成员。在这种情况下，选择类只是检索过程中的第一步，因为用户随后必须搜索大量类别成员以获取相关信息。另一个问题是，一旦层次结构就位，它只提供存储库的一个视图。对该视图的更改可能会在整个分类法中产生反响，从而导致对类结构的广泛重新设计，这可能会对存储库的整个内容产生影响。

枚举分类要求用户了解存储库的结构和内容，以便有效地检索信息。即使是经过深思熟虑的分类系统，如国会图书馆使用的分类系统，对用户来说也是有问题的 [Blackshaw and Fischhoff 1988]。在与基于分类的检索系统 Helgon 进行的一项研究中 [Fischer and Nieper-Lemke 1989]，受试者难以区分计算机科学文献参考数据库中的分类标签，如 Proceedings 和 Inproceedings [Foltz and Kintsch 1988]。这导致了对类内容的误解和无效的检索行为。问题的一部分在于，大多数信息检索系统都假设一个信息空间可以用一个单独的分类来充分表示。但没有一种分类在所有情况下都是正确的，原则上也不可能识别出大型信息空间中所有可能的相关特征 [Furnas et al. 1987]。即使是自然发生的信息空间，如生物分类学，也需要不止一个结构来满足生物学和自然科学研究人员的信息需求 [Lakoff 1987]。但这些结构经过多年的演变，枚举分类的劳动密集型性质仍然是创建多重结构的一个重大障碍。

刻面分类法通过定义可以用不同术语实例化的属性类，避免了层次结构中组件定义的枚举 [Prieto-Díaz and Freeman 1987]。这是关系模型的一种变体，在关系模型中，术语被分组为固定数量的互斥方面。用户通过为每个面指定一个术语来搜索组件。在每个方面中，分类技术用于帮助用户选择适当的术语。这与人工智能中许多基于帧的检索技术中使用的属性值结构非常相似 [Brachman et al. 1991; Devanbu et al. 1991; Patel-Schneider et al. 1984]，except that faceted techniques use a fixed

number of facets (attributes) per domain, and no such restriction exists for attribute-value methods [Frakes and Pole 1994].

刻面比枚举方案更灵活, 因为可以重新设计各个面, 而不会影响其他面。但一些可用性问題仍然存在。虽然刻面使合成和组合术语来表示组件变得很容易, 但用户很难找到正确的术语组合来准确描述信息需求, 尤其是在大型或复杂的信息空间中[Frakes and Gandel 1990]. 该方法还要求用户了解库和术语的结构, 并了解每个方面的重要性以及该方面中使用的术语[Curtis 1989]. 刻面检索系统的现场使用表明, 需要培训人们有效地使用刻面, 设计分面信息域需要更广泛的培训[Prieto-Díaz 1991].

自由文本检索(自动索引)方法使用文档中的文本进行索引。文档文本应用于“停止列表”, 以删除频繁出现的单词, 如“and”和“the”其余文本用作文档的索引。用户使用应用于索引的关键字指定查询, 以查找匹配的文档。虽然有时使用人工索引器来增加自动提取的索引项, 但不需要进行分类。匹配标准的范围从布尔匹配到更复杂的方法, 如向量模型, 这些方法使用统计度量对检索到的信息进行排序[Salton and McGill 1983].

自由文本检索构建和检索起来很简单, 但依赖于语言文本中的规律性, 这些规律性需要大量文本才能达到统计上的准确性。源代码的非语言性质以及工作代码不需要清晰准确的文档这一事实使得这些方法对软件组件存储库的吸引力不如对文本文档的吸引力。自由文本检索最适用于包含大量文档的领域, 如Unix手册页[Frakes and Pole 1994; Maarek and Smadja 1989]. 将大多数源代码描述为针对这些方法进行了充分记录是不准确的。尽管但构建存储库的低成本加上足够的性能, 使得这种方法在商业文本检索系统和万维网引擎(如Yahoo或Alta Vista)中广受欢迎。方法的检索效率在文本密集型领域受到质疑, 如Law[Blair and Maron 1985], 但构建存储库的低成本加上足够的性能, 使得这种方法在商业文本检索系统和万维网引擎(如Yahoo或Alta Vista)中广受欢迎。

## 2.2 存储库结构和检索效率

这些方法定义了一个连续体, 从需要广泛结构的枚举分类到需要较少结构的刻面分类, 再到不需要结构的自由文本索引。分类方案在计算上很容易检索(尽管存在可用性问題), 但很难构建。设计师必须获得“正确”的分类法“正确”的定义是难以捉摸的、特定于情况的, 并且

取决于个人的需要 [Harter 1992]. 刻画分类和属性值方法遭受相同问题的简化形式-必须正确获取属性; 然后, 就变成了一个简单的问题, 即在刻画中选择术语。自由文本索引涉及很少的前期成本, 但在有大量语言文本可用时最有效。

使用复杂信息结构的方法的优势在于, 人们可以使用结构中包含的知识来引导他们获得相关信息。缺点是, 如果信息的结构与用户期望不一致, 则不可能提供支持。创建专门的信息是一个劳动密集型且成本高昂的过程, 在资源紧张或涉及大型信息空间的情况下是禁止的。虽然良好的结构对检索系统的有效性至关重要, 但人们已经使用了复杂的信息检索技术来处理存储库结构和检索方法之间的关系。接下来的问题是: 结构与有效检索之间的关系是什么? 使用最小结构的存储库是否可以实现足够的检索效率?

在接下来的内容中, 通过研究在很少假设专业知识存在的情况下可以完成多少工作, 同时有效地利用可以廉价派生的信息结构, 可以达到一个折中的效果。使用的检索方法只需要很少的结构, 但可以产生有效的检索性能。然后在使用上下文中对结构进行扩充。当人们发现信息时, 他们与系统的交互被跟踪并用于改进现有结构。其结果是检索和存储库构建方法的灵活集成, 从而在使用存储库时改进底层结构。

### 3. CODEFINDER: 软件重用的信息访问

图1显示了如何以最少的前期工作构建存储库, 然后随着存储库的使用而逐步改进

。在构建存储库的初始阶段, 使用组件填充存储库是很重要的, 即使这些组件不是为重用而明确设计的[Caldiera and Basili 1991, p. 63]. 在我们的方法中, 一个存储库最初是用一种低成本的存储库构建方法“播种”的 [Fischer et al. 1994], 这种方法可以半自动地用术语和短语索引组件。本文描述的原型系统称为 **PEEL (Parse and Extract Emacs Lisp)**, 它从文本文件中提取组件, 并通过自动提取和交互式用户支持的组合对其进行索引。

生成的存储库使用最小的检索结构创建, 索引可能不完整且不一致, 使得关键字匹配算法难以检索相关信息。采用了软匹配和查询重新格式化技术的组合, 以补偿在设计存储库时所做的最小努力,

Fig. 1. Evolutionary construction of repositories.

使查找信息更容易。**CodeFinder**系统是一个原型检索工具，它使用多种检索技术帮助用户找到可重用的软件[Henninger 1994]。

即使使用有效的检索工具，存储库的有效性也与索引和检索结构的质量成正比。相关反馈和自适应索引是两种可以用来改进存储库结构的方法，以反映开发人员在搜索组件时使用的心理模型。实证研究表明，随着时间的推移，在人们有相似关注点和使用相似词汇的团队环境中，不断发展的网络结构可以设计得更加有效[Belew 1987]此外，这样的系统可以适应组织软件开发环境和业务需求的变化。

这种方法的总体优势是，成本是根据需要递增的，而不需要大量的前期存储库设计工作。最初，组件只需经过成为生产系统一部分所需的任何认证过程然后，在重用组件时，后续工作可以逐步为组件增加价值。文献中对组件增值技术进行了广泛研究，包括可重用性认证程序等方法[Caldiera and Basili 1991]以及各种领域分析方法[Prieto-Díaz and Arango 1991; Simos et al. 1995]。这里报告的工作的目的是构建一个能够容纳这些技术的存储库基础设施，并以增量方式增加价值，以便找到满足开发人员手头任务的组件。

为了有用，存储库需要（1）创建初始信息结构的工具，（2）灵活的搜索和

浏览存储库, 以及 (3) 在用户使用存储库时优化和调整信息的工具。**CodeFinder** 和 **PEEL** 的结合通过支持源代码组件半自动提取、查询构造技术和自适应索引设施的方法实现了这一目标。图1显示了这些工具如何协同工作以支持软件存储库的发展。

### 3.1 CodeFinder PEEL存储库

**CodeFinder** 和 **PEEL** 旨在研究与构建和使用存储库来支持设计过程相关的成本/收益权衡。这些问题已经在GNU Emacs文本编辑器的Emacs Lisp定制领域进行了研究[Cameron and Rosenblatt 1991; Stallman 1981]. 组件是Emacs Lisp函数、变量和常量, 它们定义了可以从Emacs文本编辑器环境执行的两个电子邮件和三个网络新闻阅读器应用程序。Emacs Lisp是Lisp的一个变体, 它具有专门的构造和原语, 用于在windows上编辑和显示文本。使用PEEL从FTP站点和新闻组下载的文件构建存储库, 以提取组件并将其转换为CodeFinder 表示 [Henninger 1993]. 源代码是从专门讨论Emacs和Emacs LISP问题的各种新闻组中提取的, 包括 *comp.emacs*, *gnu.emacs.sources*, 以及更专业的团体, 比如: *gnu.emacs.gnus* and *gnu.emacs.vm.info*。通过对PEEL的适度努力, 创建了一个由1800多个Emacs LISP函数、变量、宏和常量以及大约2900个不同术语组成的存储库。

## 4. 为存储库设定种子

种子库是通过使用关键术语和短语对组件表示进行索引来创建组件表示的问题。组件可以采用任何大小或形式, 具体取决于存储库用户的需要。**PEEL** 是一种再工程工具, 它将Emacs Lisp文件转换为基于框架的知识表示语言Kandor中的单个可重用组件 [Devanbu et al. 1991; Patel-Schneider et al. 1984], **CodeFinder** 使用该语言索引组件并创建基于框架的检索概念层次结构 [Henninger 1994]. **PEEL** 从源代码文件中提取函数、变量、常量和宏的源代码定义。从每个组件中提取信息, 并将其转换为Kandor对象。Kandor允许在定义的框架的成员上表达约束。

<sup>1</sup> At this point, the repository only contains source code functions and data structures. Nothing in CodeFinder prevents placing other types of artifacts such as larger -grained components or supporting documentation . While it is desirable to support higher -level reuse , such as subsystems [Wirfs -Brock and Johnson 1990 ], architectures [Shaw and Garlan 1996 ], or megaprogramming [Biggerstaff 1992; Wiederhold et al. 1992], the functionality and level of

granularity described here are characteristic of current software reuse repositories , which are

designed to find and reuse functions and objects to create new programs.

ACM Transactions on Software Engineering and Methodology, Vol. 6, No. 2, April 1997.



Fig. 2. Highlighting and other indexing operations.

框架定义按层次组织，并通过对超框架的限制进行定义。图4显示了Emacs Lisp存储库的部分层次结构（稍后显示）。CodeFinder【Henninger 1994年】以及LaSSIE【Devanbu等人1991年】、Argon【Patel Schneider等人1984年】和Helgon【Fischer和NieperLemke 1989年】都使用了继承和分类的推断能力来检索Kandor表示的信息。Kandor的推断能力在其他地方有详细描述【Devanbu等人，1991年；Patel Schneider等人，1984年】。出于我们的目的，Kandor表示可以被视为一组属性/值槽，其中包含有关给定组件的信息。PEEL用于捕获该信息并将其放置在Kandor插槽中

知识表示系统的好坏取决于它所包含的知识。自动提取当然是经济高效的，但也有一些重要的属性需要扩充，例如术语、作者和描述时隙。如图2所示，对于每个组件，PEEL将显示源代码（图2左侧的窗口），并允许用户编辑部分表示（右侧的窗口）。用户可以使用任何被认为有用的术语和短语来扩充表示。

考虑到非单词和深奥的缩写在源代码中非常常见，术语对于检索软件组件尤其重要。例如，PEEL解析的一个邮件系统在整个系统的源代码和文档中使用缩写“mp”作为“消息指针”这是一个需要捕获的重要缩写。PEEL使用三步程序处理术语，类似于在Basili的软件工厂提取组件的过程[Caldiera and Basili 1991]。第一步是完全自动化的。术语是从函数名和定义内或定义前的任何字符串或注释中提取的。对提取的术语应用公共停止列表，以删除语义意义较小的常用词，如“and”“of”“the”等[Fox 1992]。PEEL用户也可以通过以下方式扩充此停止列表或将术语添加到无意义单词列表表中在图2下半部分所示的窗口中，操作“将此术语放在停止词列表上”和“将此术语放在无意义词列表上”。早期版本的PEEL从变量名和函数名中提取了术语，但事实证明这些术语并不是很具有描述性。

三步处

Fig. 3. The Kandor representation of an object translated by PEEL.

第二步是向用户显示术语。用户可以删除术语，截断列表（这一特性对于许多只有名称是描述性的组件非常有用），并在源代码窗口中突出显示术语，以显示术语的使用方式和位置（请参见图2下部的弹出窗口）。第三步允许用户添加自己的术语，包括短语（用空格分隔的单词）和任何其他标点符号。

图3显示了名为“scribe environment short form”的组件的Kandor表示（源代码如图2所示）。大部分信息都是自动提取的，尽管有些信息是为正在解析的源代码文件输入一次的。其中包括文件名和组件所属的框架（“文字处理器”是用于从文件“scribe.el”中提取的所有组件的框架，包括“scribe environment short form”）。其余插槽通过从E-Lisp源组件提取信息来填充。例如，E-Lisp语法允许在参数后加一个字符串，许多程序员使用该字符串作为函数的描述。这将被提取并放置在描述槽中，PEEL用户可以在其中编辑它。

CodeFinder还创建指向组件调用的所有函数的超链接（请参见图3中的“函数调用”属性）。这些函数显示在CodeFinder界面中的鼠标敏感文本中（参见图4中检索项目窗格的示例）。因此，只需单击鼠标即可显示组件的直接子系统。

<sup>2</sup>Personal conversation with Tom Landauer and Lynn Streeter of Bellcore. One reason for this phenomena may be that variables are closely tied to the implementation of a function and lose sight of what the function does. They are more descriptive of *how* the function works instead of *what* the function does. Since a designer's primary interest is in what functions exist to help perform a task, variable names will generally be less useful descriptors than function names and comments.

此时，PEEL没有尝试评估组件的可重用性。软件质量考虑因素表明，我们可能希望只提取在各种度量中得分良好的组件，如Halstead和McCabe指标【Caldiera和Basili 1991】或其他【Poulin和Caruso 1993】。这些度量具有自动的优点，但存在基于表面信息的假设，例如组件中的运算符和变量的数量。这里描述的工作定义了一个基础架构，该基础架构可以容纳这些指标，同时提供工具来查找具有特定功能的组件。

## 5. 补偿不完整和不一致的索引

信息检索接口需要解决两个主要问题：第一个是设计问题的本质，在设计过程中，人们对需要什么以及应该如何解决问题有一个不明确的概念 [Belkin et al. 1982a; 1982b; Borgman 1989; Henninger 1994]。第二个问题是，文档索引通常不一致且不完整。研究表明，人类索引者对于索引文档的术语会产生分歧，同一索引者会在不同的时间使用不同的术语索引同一文档 [Salton and McGill 1983]。其他研究表明，人们使用一组令人惊讶的不同描述符来描述相同的对象 [Furnas et al. 1987]。

因此，建立在索引质量是信息检索服务效率的首要因素这一假设基础上的检索系统将无法支持其用户。搜索过程不仅仅是找到精确匹配 [Biggerstaff and Richter 1987; Henninger 1994]。它包括找到为问题提供部分或类似解决方案的组件 [Maiden and Sutcliffe 1992]。不一致的索引总是存在的，因此有必要使用超越简单的关键字匹配方案的检索方法和算法 [Belkin and Croft 1987]。传统匹配策略的缺点是，查询被视为用户需求的精确规范，文档表示被视为存储库对象的精确描述。另一种更现实的策略是在文档和查询表示中假设一定程度的不确定性 [Mozer 1984]。

CodeFinder（见图4）旨在通过使用智能检索方法和支持查询构造来解决这些问题。“智能”检索由关联扩散激活检索算法提供【Mozer 1984】，该算法扩展了精确匹配范式，以检索与查询关联的项目。通过重新格式化检索（Williams 1984）支持查询构造，这是一种允许用户在探索信息空间时以增量方式构造查询的技术。实证研究表明这种技术的结合在面部获得了良好的表现索引问题和定义不清的信息需求【Henninger 1994】，并可以有效地与PEEL创建的最小结构存储库协作

**Fig. 4.** CodeFinder用户界面。在界面中，“类别层次结构”窗口显示存储库的图形层次结构。查询窗格显示当前查询。查询的顶部指定要在其中搜索的两个类别（“THING”和“E-mail”）。底部部分指定了关键术语和相关组件。“检索项目”窗格按与查询的相关性顺序显示与当前查询匹配的所有组件。检索项目窗格的示例显示了顶部匹配组件或用户选择的组件的完整条目。“书签”窗格包含已查看对象的历史视图。“相关术语”窗格显示查询检索到的术语。

Fig. 5. An associative network. In CodeFinder the network consists of two layers of nodes: terms (represented by circles) and components (rectangles). Boxed lines are connections between components and terms, as defined by the *Terms* field of the component. Link weights are shown in link boxes.

关联网络。在CodeFinder中，网络由两层节点组成：术语（由圆表示）和组件（矩形）。方框线是组件和术语之间的连接，由组件的“术语”字段定义。链接权重显示在链接框中

## 5.1 扩展激活检索

PEEL创建的存储库结构是基于框架的属性/值对组织中的组件表示的集合（见图3）。给定这种结构，可以通过将查询术语与存储库中的术语字段相匹配来检索组件（CodeFinder还使用参数和描述字段）。CodeFinder使用这种表示来构建一个关联网络（见图5）。然后，将基于连接主义松弛检索过程的关联扩展激活过程应用于网络以检索信息【Mozzer 1984】。用户指定由术语节点和组件节点之一或两者组成的查询。查询节点的激活值为1.0，该值在扩展激活过程中保持为该值。如果节点具有正激活值，则该值通过其每个链接传递；否则不会通过激活。每个节点计算由链接权重修改的传入激活值的总和。然后，通过扇入、衰减和其他参数对接收到的激活值的总和进行调制。结果值被输入到挤压函数中，以使激活值保持在0.2和1的边界之间（需要不对称值来鼓励正激活流【McClelland and Rumelhart 1981】）。

例如，在图5中指定术语remove的查询将向组件节点“vm quit”和“vm delete message”传播值0.499（通过下面描述的各种参数修改）在下一个周期中，“vm quit”和“vm delete message”将把计算出的激活值传播到它们所连接的所有节点。例如，“vm delete message”将激活message和delete。在第三个周期中，所有先前激活的节点将继续传播其值。

Table I. Associative Spreading Activation Parameters

Symbol	Definition
$n_D$	number of documents in the repository
$n_T$	number of distinct terms in the repository
$w_{ij}$	strength of connection between node $i$ and node $j$
IDF	inverse document frequency
$\Theta_D$	document decay rate
$\Theta_T$	term decay rate
$M$	maximum activity level (1.0)
$m$	minimum activity level (-0.2)
$Dt_i$	number of terms in document $i$
$df_j$	number of documents indexed with term $j$
$\overline{Dt}$	average number of terms per document in collection
$\overline{df}$	average number of documents per term in collection
$\alpha_D$	document fan-in
$\alpha_T$	term fan-in

请注意，message和delete将一起激活“mail delete forward”和“gnus kill”，检索这些组件，即使它们没有直接按照查询中使用的术语编制索引。这通常被称为“查询扩展”[Salton and Buckley 1988]. 还要注意的，delete被标识为remove的同义词。关键字通过索引项动态关联，通过共现关系补偿不一致的索引。

允许系统通过上述程序循环，直到达到稳定或达到最大循环次数。当循环导致节点激活值发生微小变化时，会发生稳定。与所有非布尔检索系统一样，通过扩展激活，无限数量的文档可能在某种程度上“相关”。但随着激活从查询节点传递得更远，活动值很快消失。由于计算限制，CodeFinder 允许用户选择最大循环数[Henninger 1993]. 四到五个周期通常允许部分稳定，同时仍然检索从初始查询中删除的大量项目的关联

5.1.1 计算激活值。从形式上讲，扩散激活是由文档3的活动级别和对应于周期的离散时间单位的术语定义的。 Given the parameter definitions found in Table I, the activity level of a document  $D_i$  at time  $t + 1$  is given by

$$D_i(t + 1) = \begin{cases} \delta_{D_i}(t) + \Psi_{D_i}(t)(M - D_i(t)) & \text{if } \Psi_{D_i}(t) > 0 \\ \delta_{D_i}(t) + \Psi_{D_i}(t)(D_i(t) - m) & \text{if } \Psi_{D_i}(t) \leq 0, \end{cases}$$

<sup>3</sup>To remain consistent with standard information retrieval technology, “document” will be used in place of “component” in this section.

where  $\delta_{D_i}(t)$  is the previous activation valued reduced by the document decay rate

$$\delta_{D_i}(t) = (1 - \Theta_D) D_i(t),$$

and  $\Psi_{D_i}(t)$  is the net input to document  $i$  at time  $t$ , given by

$$\Psi_{D_i}(t) = \left( \frac{Dt}{Dt_i} \right)^{\alpha_D} \sum_{j=1}^{n_T} w_{ji} U(T_j(t)),$$

where  $U(x)$  is the zero-threshold identity function that does not allow negative activation values to propagate:

$$U(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{if } x > 0. \end{cases}$$

The activity level of a term unit  $T_j$  is found in the same manner, substituting term for document parameters:

$$T_j(t+1) = \begin{cases} \delta_{T_j}(t) + \Psi_{T_j}(t)(M - T_j(t)) & \text{if } \Psi_{T_j}(t) > 0 \\ \delta_{T_j}(t) + \Psi_{T_j}(t)(T_j(t) - m) & \text{if } \Psi_{T_j}(t) \leq 0, \end{cases}$$

where  $\delta_{T_j}(t)$  is the previous activation valued reduced by the document decay rate,

$$\delta_{T_j}(t) = (1 - \Theta_T) T_j(t),$$

and  $\Psi_{T_j}(t)$  is the net input to term  $j$  at time  $t$ , given by

$$\Psi_{T_j}(t) = \left( \frac{df}{df_j} \right)^{\alpha_T} \sum_{i=1}^{n_D} w_{ij} U(D_i(t)).$$

**5.1.2 扩展激活算法的参数** 节点的活动级别是由衰减因子减少的先前激活和由先前激活值与最大值或最小值之间的差值缩放的净输入的组合. 保留前一个活动的一部分可以减少活动在不同时间到达节点时发生的剧烈波动, 本质上集成了流程中所有周期中节点的证据[Belew 1987]. 然后将该值输入到 $\arctan()$ 函数中, 以将值保持在最大值和最小值之间. 衰减因子 $\Theta$ 会导致节点在每个循环中失去固定百分比的激活, 从而导致激活值随时间呈指数衰减. 接收到一些初始激活, 但缺乏来自激活节点的正反馈形式的确证证据的节点, 将随着时间的推移逐渐衰减到静止值0。

这可以防止“惯性”现象，即接收激活的节点将其永久保留。因为术语的数量通常会超过文档，而且术语作为查找相关文档的多个来源，所以术语的衰减率需要设置为高于文档。实验表明，项（QT）的衰减率为0.25，文件（QD）的衰减率为0.1，效果良好[Henninger 1993; Mozer 1984].

网络输入 $\Psi$ 由扇入因子 $x^a$ 修改，以防止具有大量连接的节点接收过多的激活。将每个文档的平均术语数除以给定节点的术语数，会增加术语少于平均值的节点的净输入，并减少术语多于平均值的节点的净输入。通过扇入指数 $a$ 提高该因子，可以放大效果，并提供一个可以调整的参数，直到消除具有大量连接的节点的偏差。CodeFinder存储库的实验发现  $\omega = 0.3$  and  $\alpha = 0.2$  有效地消除了扇入偏差[Henninger 1993].

**5.1.3 链接权重.** 扩展激活系统结构的一个关键部分是关联网络中节点之间的链路权重。这控制了扩散激活周期中传递的信息量。以前的扩展激活系统定义了节点之间的恒定链路权重[Mozer 1984]. 这是基于一个错误的猜测，即所有术语都与一个项目同等相关。例如，如果术语“reply”和“mail”用于索引项目 $vm$  reply，那么可以肯定地说，“reply”更能描述该功能。“Reply”是一个更好的鉴别器，它可以更好地描述项目，并且应该以比“mail”更高的概率检索 $vm$ 回复这种鉴别可以在关联网络中通过改变与项的鉴别值成比例的权重来实现，更好的鉴别器具有更高的权重，这将导致更大激活值的扩散。

术语到项目的区分很难用一般化的方式定义，也很难应用于大型存储库中的每个术语-组件关系。幸运的是，经验观察表明，频率较低的术语有倾向于更精确（在回忆和精确性方面），这导致了有效的近似。[Furnas et al. 1987; Sparck-Jones 1972] Furnas et al. 1987; Sparck-Jones 1972 ] 如果一个术语的频率较低（在信息空间中很少出现），那么它是一个很好的项目识别器。例如，如果一个项目由术语A和B索引，A索引了100个其他项目，而B只索引了2个其他项目，那么B对该项目来说是非常独特的，应该以相当高的概率检索它。B这种称为反向文档频率（IDF）的方法在许多域的检索系统中都非常有效[elkin and Croft 1992; Salton and Buckley 1988].

在CodeFinder中，从文档到描述符的连接的权重

ACM Transactions on Software Engineering and Methodology, Vol. 6, No. 2, April 1997.



$w_{ji}$ 由(重量对称)给出

$$w_{ji} = w_{ij} = \begin{cases} 0 & \text{if no connection between } i \text{ and } j \\ \text{IDF} = \frac{\log(n_D/df_j)}{\log(n_D)} & \text{if } i \text{ and } j \text{ connected.} \end{cases}$$

这是标准IDF加权方案的一种修改形式, 即 $\log(n_D/df_j)$  [Sparck-Jones 1973]. (i.e 修改将标准IDF除以文档总数的日志, 将权重值标准化为介于0和1.0之间。请注意, 索引一个文档的术语的权重为1.0.,  $\log(n_D)/\log(n_D)$ ).

此方案为存储库中的每个术语赋予一个权重。On更好的判别标准是认识到某些术语对于某些项目来说是更好的判别者, 并且不应在其索引的所有项目中具有统一的权重。此term frequency (TF) 的一个度量值, 即给定项目或文档中术语的频率。[Salton and Buckley 1988]. CodeFinder只使用IDF度量, 因为软件源代码对象没有TF度量所依赖的大型文本文档的语言规则性。CodeFinder通过相关反馈机制实现单个术语到文档的加权(see Section 6.3).

5.1.4通过内容诱导结构定义关系. PEEL和CodeFinder的结合的威力在于, 对于CodeFinder来说, 索引不需要详尽无遗, 就可以有效地定位软件对象。PEEL的最小索引与CodeFinder的扩展激活和浏览工具相结合, 充分支持定位相关源代码的过程, 这在用户研究中已得到验证 [Fischer and Nieper-Lemke 1989; Foltz and Kintsch 1988; Henninger 1994]. [关联扩展激活模型使用关键字和组件之间的公共连接来“诱导”组件和术语之间的关系[Mozer 1984], 对使用不正确术语的查询进行补偿

传播激活用于检索信息和归纳组件之间关系的是存储库的结构, 而不是预定义的术语或组件关系。演示此效果的一个特别突出的方法是显示通过单字查询检索到的相关关键字。图6 (a) 显示了查询delete将检索大量同义词尚未将删除和检索到的组件之间的语义关系编程到系统中。尚未将delete和检索到的组件之间的语义关系编程到系统中。考虑到存储库的结构, 其中术语delete、remove和kill在组件之间出现, 传播激活能够诱导这些单词之间的语义关系。这被称为“内容诱导结构”[Halasz 1988]. 图6 (c) 显示, 当指定了术语open时, 与具有开放操作的事物相关的术语, 例如

图6: 通过内容诱导结构捕获单词关系。相关术语*span*中的最高比率术语（与查询最相关）按照从最高（顶部）到最低（底部）激活值的顺序出现，在语义上与查询相关；（a）显示仅指定了删除的查询结果；（b）使用术语*kill*；（c）使用*open*。

除了启动和打开等同义词外，还检索服务器、连接、主机等（请注意，这些术语通常不被视为同义词，而是在程序命名成语中经常出现的术语）。图6所示的屏幕图像直接从PEEL创建的Emacs Lisp存储库中检索，没有对存储库进行任何增强，只是通过传播激活可以派生的语义关系的样本。只要存在适当的联想结构，就会发现这种关系。

扩散激活不仅可以找到与语义相关的单词，还可以捕获存储库中独特的术语使用情况。例如，在图6(A)中，DELETE和KILL之间的关系抓住了一个编程成语，其中术语KILL通常指删除对象此外，其中一个程序，View Mail(由Kyle Jones编写)，在删除邮件时使用术语Delete，在涉及删除文件时使用Kill(图6(B))。This因此，如图所示，DELETE查询检索了与消息相关的词语，而KILL查询则找到了许多与文件相关的词语。传统的基于词的等价类的同义词词典方法会遗漏一种特殊的术语用法，但对于生成惯用术语来描述复杂概念的领域来说，如软件开发、医学领域等，这是很重要的。扩散激活使用存储库的结构来查找惯用的词关系，而不需要额外的劳动密集型工作来预定义词关系。然后，这些计算的单词关系可以显示在“相关关键词”窗格中（见图4），为重新格式化查询提供线索。

## 5.2 通过改写进行检索

除了软匹配算法，检索系统还可以通过支持查询构造过程来减少不良存储库结构的影响。重构检索是一种通过构建查询结果来支持增量查询形成的方法[Williams 1984]。每次用户指定一个查询时，系统都会用查询重新格式化提示进行响应，这些提示会向用户指示存储库的结构以及用于索引对象的术语。然后，用户可以通过对以前查询的结果进行批评性分析来逐步改进查询。这通过积累在搜索过程中获得的知识，同时缩小解决方案的范围，支持细化信息需求。Rabbit [Williams 1984] and Helgon [Fischer and Nieper-Lemke 1989] 是基于重构检索范式的检索系统的例子。

CodeFinder通过在接口中提供大量检索线索，支持通过重新格式化进行检索。其中最重要的是显示在“检索项目示例”窗格中的示例组件（见图4）。此窗格显示组件的完整表示形式，允许用户选择要在查询中包括或排除的任何属性。通过重新配方进行检索是通过鼠标操作完成的，该操作允许用户只需指向并单击要包含在查询中的组件。例如，如果用户想在图4中添加组件“vm-kill-subject”的“条款”属性中显示的术语“主题”，他们只需单击该术语-left。该术语将放在查询窗格的术语字段中，命令窗格中的“检索”按钮将以粗体显示，表明查询已更改。然后，选择“检索”按钮将根据新的查询检索信息

query.<sup>4</sup>

CodeFinder通过三种方式重新格式化范式来增强检索。首先，项目会自动放置在查询的适当部分。与查询语言不同，用户对查询的结构和内容负责，用户在决定将属性、类别或术语放在哪里时免受认知开销。其次，CodeFinder对检索集添加一个排名标准，自动显示查询的评分最高的项目。这很重要，因为示例提供了线索，描述了查询检索的哪种项目以及如何改进它。示例越好，就越容易收敛到满意的解。选择一个好的例子没有被以前的重新表述系统检索到。例如，Helgon按照字母顺序组织了检索集，并按此顺序显示了第一个示例[Fischer and Nieper-Lemke 1989]。第三，CodeFinder在相关条款窗格中显示术语

<sup>4</sup>The two -step process was used because retrieval will take a nontrivial amount of time in large repositories. If users wish to make multiple changes to a query, they would have to wait each time a change is made. The “Retrieve” button puts users in control of when this potentially time - consuming operation takes place.

Fig. 7. Situation and system models.

(如图4和图6所示)。这些术语还按照与查询相关的顺序进行排序，并可用于细化查询。

## 6. 支持组件存储库的增量提细化工作

一旦用最小的结构捕获了组件，就必须演化该结构以满足开发组织的需要。这不仅是为了改善存储库的索引和结构，也是为了随着组织知识的发展保持信息的最新状态。[Henninger et al. 1995].即使使用创新和智能的检索工具，存储库的有效性也会受到索引质量的影响。问题在于，索引“质量”是一个相对的衡量标准，这取决于由不同的信息需求和视角引起的检索上下文，这些需求和观点对不同时间具有不同信息需求的个人以及不同背景的不同用户都不同[Harter 1992].因此，不能预先设计有效的存储库结构；必须允许它们在使用的环境中发展。

组件存储库的最终目标是创建一个“类似于大多数在应用程序领域工作的程序员所拥有的知识结构的索引方案”[Curtis 1989]迄今为止，很少有研究来描述这些知识结构的本性。一个潜在有用的研究方向是研究人们如何建模问题和解决方案空间之间的关系[Kintsch and Greeno 1985].该理论区分了情景模型(应用程序领域的设计者的模型以及他或她所面临的特定问题)和系统模型(例如，由系统、编程语言和环境提供的一组资源)。如图7所示，这些模型之间的关系是一种转换关系。用户对问题的初始高级概念化需要被翻译成用于索引所需项目的语言。通过最小化情况和系统模型之间的不匹配，可以提高存储库的有效性.这可能是一个棘手的问题，如图7所示，这是从一项实证研究中得出的，该研究要求人们输入关键字来描述给定的对象[Henninger 1994].

请注意，虽然用户通常会根据其应用程序目标将检索问题概念化，但系统术语指的是对象是如何实现的。用户不熟悉实现模型将难以找到所需的功能。仅仅设置一些结构是不够的；它必须是一个用户可以理解和使用的结构，一种支持用户系统模型的结构。

Curtis指出，实证研究表明，随着程序员获得经验，他们对特定领域知识的理解倾向于共同结构[Curtis 1989]。经验还表明，适应性系统中的索引结构在兴趣相似的群体中向共同主题趋同[Belew 1987]。虽然很难预先预测所有的设计需求或设计制品的用途，但在适度同质的环境中工作的一群用户将倾向于使用类似的术语来处理问题。研究表明，项目中的大量努力旨在让开发人员对问题产生相互理解[Walz et al. 1993]。捕获典型使用情形的最佳机会是提供在使用期间轻松修改存储库的方法。这为设计师如何看待他们的领域提供了批判性的洞察力——他们情况模型的特色。

## 6.1 在使用环境中调整存储库

通过记录用户操作并根据这些操作增量更改表示，应用了改进存储库的自适应技术。当用户与系统交互时，系统将使用通常由用户确定的各种应用目标向索引方案迁移。总体目标是让系统逐步构建用户知识的共识表示[Rose and Belew 1991，第3页]，该表示可以随着组织知识的发展而改变。

Bellcore对人们选择术语来描述常见对象、烹饪食谱、编辑器命令和其他项目的研究显示，两个人为这些对象选择相同关键字的概率在10%到20%之间[Furnas et al. 1987]。使用15个别名可以达到60%-80%的一致性，使用30个别名可以获得高达90%的一致性。这导致了“无限别名”的概念，其中用户应用于描述对象的任何术语都应用作该对象的描述符（索引项）[Furnas et al. 1987]。请注意，别名与使用术语等价类的传统同义词词典方法不同，因为在别名中，术语直接指向对象，而不是同义词中的其他术语。

无限混叠方法的问题之一是可能会出现精度问题。如果更频繁地使用术语来描述存储库中的对象，则系统检索不需要的对象的概率会增加。对使用大量别名的检索系统的研究表明，精度问题没有理论预期的那么严重[Gomez et al. 1990]。

发生这种情况是因为在实践中，每个对象的更多项不一定意味着每个项的对象更多，在交互式检索环境中，精度的理论概念（检索的无关对象与总检索集的比率）不如召回（检索的相关对象与总检索集的比率）。换句话说，只要检索到相关对象，并且人们能够找到并阅读这些对象，检索多少无关对象就不那么重要了。

实现无限制混叠而不存在不必要的精度问题的一种方法是自适应索引。[Furnas 1985].这种方法通过在文档满意的搜索中使用对象时将术语添加到对象表示中，以交互式方式收集术语使用数据。CodeFinder通过跟踪检索会话期间的术语使用并允许用户将这些术语与所选项目的索引表示相关联，从而支持无限制的别名和自适应索引。在构造查询和浏览信息空间的过程中，用户可能从查询中删除多个术语，并输入系统词典中没有的术语（在这种情况下，发布警告该缺陷的错误消息）。这些术语可以作为情景模型的证据，这是思考检索问题的初始方式。当前查询中也可能有一些术语不在项目的当前表示形式中。将所有这些术语添加到所选项目中可能会增强该项目未来的可检索性，但我们必须防止检索会话中使用拼写错误或不良术语的情况。

CodeFinder 通过向用户显示信息以供检查和可能的修改来解决此问题。当用户单击“检索项目示例”窗格中的“选择此”按钮时，这些术语会显示给用户（见图4中的匹配项目窗格示例）。单击此按钮可以具有多种语义，例如将示例放置在编辑器缓冲区中或调用其他CASE工具来分析组件 [Fischer et al. 1991]. 发生这种情况时，将调用自适应索引会话，如图8所示。用户可以选择将这些项添加到所选示例的当前表示中，并允许根据需要输入其他术语。

存储库用户还可以通过下拉菜单修改CodeFinder中基于Kandor的部分使用的存储库结构，如图3中间顶部所示的菜单 [Henninger 1995]. 这种修改可能最好由存储库管理员来处理，但它是支持存储库进化的重要工具。

## 6.2 通过相关性反馈调整链接权重

除了为组件表示分配新术语外，代码查找器还使用相关性反馈来调整术语组件链接权重。相关反馈通常用于重新表述查询，通常是通过将文档的表示添加到查询中 [Salton and Buckley 1990].

Fig. 8. Adaptive indexing in CodeFinder.

CodeFinder通过允许用户单击检索项目窗格右侧的圆圈来实现这一目标（见图4）。选择的每个项目都放在查询中（请参阅查询窗格的相关项目部分），在那里它被用作扩展激活查询的一部分。

CodeFinder还使用此操作来表明所选组件与查询窗格的术语部分中显示的术语相关。加强这些组件和术语之间的链接权重将增加未来类似查询获得更好检索结果的可能性。局部强化[Belew 1987]学习技术通常调整权重，以便激活一些输入节点导致输出上的目标向量，用于加强权重。在CodeFinder中使用用户反馈来代替目标。我们希望加强的关联是查询中的术语和相关性反馈中选择的组件之间的关联。因此，如果在相关反馈中选择的组件由查询中的任何词语索引，则每个权重都根据以下学习规则进行调整[Rose and Belew 1991, p. 21]:

$$w'_{ij} = w_{ij} + \eta f_i a_j,$$

其中  $a_j$  是文档节点  $j$  的激活值;  $f_i$  用户节点给予节点  $i$  的反馈 (a binary yes/no value in CodeFinder); and  $\eta$  is the learning rate, usually a value between 0 and 1.0. The values are adjusted by an  $\arctan()$  squashing function to keep values between  $-1.0$  and  $1.0$ . 因为“知识在链接权重”[Smolensky 1988], 这可能对传播激活的有效性做出重要贡献。不幸的是，评估这种方法的有效性遇到了“信用分配”问题，其中的权重是负责结果的，应该改变哪个权重来提高性能。

$\eta$




Fig. 9. CodeFinder architecture.

然而，实证研究表明，检索性能确实会随着时间的推移而提高，尤其是在兴趣相同的群体中使用时[Belew 1987].

## 7. 编解码器的设计与评价

### 7.1 实施

CodeFinder的一般架构如图9所示。CodeFinder的界面建立在Symbolics Lisp机器上。为了效率和扩展存储库的原因，扩展激活计算被设置为在Unix机器上执行的C例程。套接字接口用于在不同操作系统之间传递信息。PEEL将E-lisp源文件作为输入，并生成Kandor对象。要创建矩阵，CodeFinder从Kandor对象中提取项，并创建一系列形式为{term, item}的元组，每个数据库对象中的每个项一个元组。在Unix端，每个元组被插入稀疏矩阵表示中，并为扩展激活参数积累了统计信息。

ACM Transactions on Software Engineering and Methodology, Vol. 6, No. 2, April 1997.



扩展激活的实现是简单的矩阵乘法，并对算法参数进行调整。术语和文档之间的连接以 $n \times n$ 连接矩阵表示，其中 $n$ 是包含所有文档和术语的向量。初始查询以长度为 $n$ 的向量表示，查询节点（术语或文档）设置为1.0。该向量与矩阵相乘，得到一个向量，该向量对连接到查询的所有节点的净输入求和。然后通过算法参数调整节点。生成的向量乘以连接矩阵来计算下一个周期。矩阵乘法，尤其是大矩阵乘法，是一种资源需求大的运算。幸运的是，关联网络稀疏地填充矩阵，从而允许使用稀疏矩阵乘法算法来提高计算效率。CodeFinder中使用的结构化关联网络尤其如此，它不允许术语之间和组件之间的链接。通过使用DECstation 5000和基本的稀疏矩阵乘法器，Emacs Lisp存储库实现了不到一秒的响应时间。该算法的计算复杂度部分支持用于实现扩展激活的稀疏矩阵乘法算法。更快的硬件和更好的稀疏矩阵乘法算法可以轻松容纳具有可接受性能的数万个项目的存储库。

CodeFinder还允许在Unix端存储矩阵表示，因此可以省略矩阵创建步骤。通过自适应索引设施(见第6节)对矩阵的更新是通过将元组发送到矩阵创建模块来处理的，该模块将元组插入到矩阵中并更新参数。如果新术语是唯一的，即没有出现在存储库中，则没有其他操作。但是，如果将现有项添加到组件中，新的项项关联会更改该链接权重（请记住，权重由IDF度量决定）。必须根据与该术语关联的每个组件调整权重。

## 7.2 CodeFinder的经验评估

进行了一项实验，以评估CodeFinder的接口和具有最小结构(即，从Peel派生的结构而没有自适应索引的优势)的检索方法的有效性[Henninger 1993; 1994]。研究人员给受试者分配的任务包括查找软件组件和功能。任务是根据人们在Emacs LISP环境中开发软件的观察而改编的。CodeFinder与另外两个软件组件检索系统进行了比较，这两个系统使用的是直接匹配检索算法，但重新制定查询的能力有限。评估了这些系统在帮助受试者根据问题定义（精细、定向和定义不当）和词汇匹配（匹配、不匹配）维度解决问题的能力进行了评估。

结果表明，当任务属于定义不明确或词汇不匹配的类别时，系统之间的差异更为明显

[Henninger 1994] CodeFinder在这些类别中得分最高, 受试者找到更多项目的速度更快。结果表明, 当问题定义不明确或词汇表不匹配时, 受试者使用直接匹配算法的系统存在问题。CodeFinder取得更大程度的成功表明, 软匹配检索算法, 如扩展激活, 加上查询构建技术, 对于充分支持开发人员搜索可重用组件信息通常遇到的任务是必要的。

重要的是要强调, 这些结果是CodeFinder整体接口的连接, 而不仅仅是扩展激活算法。该实验旨在评估接口, 而不是检索算法, 因此评估标准与传统的召回和精确度量[Salton和McGill 1983]不同, 而是专注于受试者使用系统功能查找相关信息的能力。因此, 我们的结果与其他研究一致, 这些研究得出结论, 仅靠检索方法不足以获得卓越的性能 [Frakes and Pole 1994]. 因为我们能够通过重构控制传播激活和检索的影响[Henninger 1994], 我们的假设是CodeFinder更好的性能至少部分归功于这些方法的耦合。需要进行更多的研究来验证这些结果, 并提高我们对观察结果的理解。

## 8. 结论和未来方向

本文中描述的工作源于对软件开发工具的需求, 这些工具支持查找要重用的组件的过程。尽管完美构建和索引的组件存储库确实使相关组件很容易找到, 但现实情况是, 实用的和固有的环境阻止了此类存储库的创建。设计问题的定义不明确性就是这样一个问题, 使得很难为组件进行索引以备将来使用。另一个问题是, 软件设计过程中经常使用的陌生和深奥的词汇使潜在的重用者难以找到有用的组件。

因此, 预测使用给定组件的所有需求既困难又昂贵。即使可以实现完美, 人们使用不同的词汇来描述不同的信息需求, 这一事实将阻止用户始终找到合适的组件。除了成本高昂之外, 检索所需的结构通常是静态的, 无法适应动态的开发环境。一旦创建了结构, 它们就“一成不变”, 只能通过复杂的过程进行更改, 而这些过程通常涉及重新设计整个存储库。需要的方法是: (1) 以最少的索引和结构化工作提供足够的检索效率, 允许组织利用从以前的开发工作中积累的宝贵资产, 而无需在可重用性方面进行大量的前期投资

(2) 帮助该设备随着软件开发组织中人们不断变化的需求而发展。

本文介绍了一种支持存储库的“生命周期”的方法。存储库最初通过PEEL（一种从Emacs Lisp源代码中提取存储库信息的工具）播种结构和索引术语。CodeFinder通过重新配方和扩展激活支持检索过程，然后用于帮助用户在不太完美的存储库结构和索引时找到组件。在使用CodeFinder查找组件时，用户有机会向存储库添加结构和索引项，从而改进存储库以反映存储库用户通常遇到的各种问题。此过程的优点是避免了昂贵的存储库填充工作，同时捕获可能对将来遇到类似信息需求的人有用的信息。

经验研究表明，即使使用PEEL执行的最小结构和索引，CodeFinder也能够充分支持找到相关软件组件的过程[Henninger 1993;1994] 这项研究的结果只是为CodeFinder提供的存储库工具的有效性提供了一些证据。需要进行更多的研究，以便在软件开发的背景下更好地理解检索过程的性质。还需要在软件开发组织的背景下进行纵向研究，以评估CodeFinder中自适应工具的实用性和局限性。这种性质的研究有望更好地理解如何在解决问题的环境中满足信息需求，以及存储库如何随着时间的推移变得更加有效。

研究还指出，在PEEL/CodeFinder工具集成为发展组织的可行工具之前，需要对其进行一些改进。目前，尚未尝试评估组件的可重用性或捕获其他组件度量。目前正在将本文中描述的方法应用于更大收益的软件组件和非源代码工件[Henninger 1997;Henninger et al.1995]。还需要进行更多的研究，以将CodeFinder与其他存储库检索方法进行比较，如分面分类[Prieto-Díaz和Freeman 1987]，尽管这些方法的前期存储库建设成本使此类比较变得困难。

为了使软件重用方法的可行和有效，需要以现有代码模块的形式利用开发组织中存在的宝贵资产的方法，这些代码模块已经开发并保存，而无需考虑重用。必须开发用于构建存储库的工具，以提取这些模块，并以适合特定检索工具集的存储库形式表示它们。这里介绍的工作为解决这些问题采取了一些步骤。未来需要工作来完善这些想法，并对其进行严格的实证研究，以评估其有效性，并了解更多关于为支持软件重用技术而构建的组件存储库的性质。

## ACKNOWLEDGMENTS

感谢科罗拉多大学格哈德·菲舍尔人机交流小组的成员，特别是彼得·福尔茨、格哈德·菲舍尔、沃尔特·金奇、大卫·雷德迈尔斯和科特·史蒂文斯，他们就本文中讨论的观点进行了许多讨论。Helga Nieper Lemke和Jonathan Bein对可重用软件的慷慨捐赠为CodeFinder提供了最初的开端。

## REFERENCES

- BARNES, B. H. AND BOLLINGER, T. B. 1991. Making reuse cost-effective. *IEEE Softw.* 8, 1, 13–24.
- BATORY, D., SINGHAL, V., THOMAS, J., DASARI, S., GERACI, B., AND SIRKIN, M. 1994. The GenVoca model of software-system generators. *IEEE Softw.* 11, 5, 89–94.
- BELEW, R. K. 1987. Adaptive information retrieval: Machine learning in associative networks. Ph.D. dissertation, Tech. Rep. 4, Univ. of Michigan, Ann Arbor, Mich.
- BELKIN, N. J. AND CROFT, W. B. 1987. Retrieval techniques. *Ann. Rev. Inf. Sci. Tech.* 22, 109–145.
- BELKIN, N. J. AND CROFT, W. B. 1992. Information filtering and information retrieval: Two sides of the Same Coin? *Commun. ACM* 35, 12 (Dec.), 29–38.
- BELKIN, N. J., ODDY, R. N., AND BROOKS, H. M. 1982a. Ask for information retrieval: Part 1. *J. Doc.* 38, 2, 61–71.
- BELKIN, N. J., ODDY, R. N., AND BROOKS, H. M. 1982b. Ask for information retrieval: Part 2. *J. Doc.* 38, 3, 145–163.
- BIGGERSTAFF, T. J. 1992. An assessment and analysis of software reuse. *Adv. Comput.* 34, 1–57.
- BIGGERSTAFF, T. J. AND RICHTER, C. 1987. Reusability framework, assessment, and directions. *IEEE Softw.* 4, 2, 41–49.
- BLACKSHAW, L. AND FISCHOFF, B. 1988. Decision making in online searching. *J. Am. Soc. Inf. Sci.* 39, 6, 369–389.
- BLAIR, D. C. AND MARON, M. E. 1985. An evaluation of retrieval effectiveness for a full-text document-retrieval system. *Commun. ACM* 28, 4 (Apr.), 289–299.
- BOOCH, G. 1987. *Software Components with ADA*. Benjamin Cummings, Menlo Park, Calif.
- BORGMAN, C. L. 1989. All users of information retrieval systems are not created equal: An exploration into individual differences. *Inf. Process. Manage.* 25, 3, 237–251.
- BRACHMAN, R. J., MCGUINNESS, D. L., PATEL-SCHNEIDER, P. F., RESNICK, L. A., AND BORGIDA, A. 1991. Living with CLASSIC: When and how to use a KL-ONE-like language. In *Principles of Semantic Networks: Explorations in the Representation of Knowledge*, J. F. Sowa, Ed. Morgan Kaufmann, San Mateo, Calif., 401–456.
- BURTON, B. A., ARAGON, R. W., BAILEY, S. A., KOEHLER, K. D., AND MAYES, L. A. 1987. The reusable software library. *IEEE Softw.* 4, 4, 25–33.
- CALDIERA, G. AND BASILI, V. R. 1991. Identifying and qualifying reusable software components. *Computer* 24, 2, 61–70.
- CAMERON, D. AND ROSENBLATT, B. 1991. *Learning GNU Emacs*. O'Reilly and Assoc., Sebastopol, Calif.
- CHEN, P. S., HENNICKER, R., AND JARKE, M. 1993. On the retrieval of reusable software components. In *Advances in Software Reuse (Proceedings of the 2nd International Workshop on Software Reusability)*. IEEE Computer Society Press, Los Alamitos, Calif., 99–108.
- CURTIS, B. 1989. Cognitive issues in reusing software artifacts. In *Software Reusability*. Vol. 2, *Applications and Experience*, T. J. Biggerstaff and A. J. Perlis, Eds. Addison-Wesley, Reading, Mass., 269–287.
- DEVANBU, P., BRACHMAN, R. J., SELFRIDGE, P. G., AND BALLARD, B. W. 1991. LaSSIE: A knowledge-based software information system. *Commun. ACM* 34, 5 (May), 34–49.

- FISCHER, G. AND NIEPER-LEMKE, H. 1989. HELGON: Extending the retrieval by reformulation paradigm. In *Human Factors in Computing Systems, CHI '89 Conference Proceedings*. ACM, New York, 333–352.
- FISCHER, G., HENNINGER, S. R., AND REDMILES, D. F. 1991. Cognitive tools for locating and comprehending software objects for reuse. In the *13th International Conference on Software Engineering*. ACM, New York, 318–328.
- FISCHER, G., MCCALL, R., OSTWALD, J., REEVES, B., AND SHIPMAN, F. 1994. Seeding, evolutionary growth and reseeding: Supporting the incremental development of design environments. In *Proceeding of the Conference on Computer-Human Interaction (CHI '94)*. ACM, New York, 292–298.
- FOLTZ, P. W. AND KINTSCH, W. 1988. An empirical study of retrieval by reformulation on HELGON. Tech. Rep. 88-9, Inst. of Cognitive Science, Univ. of Colorado, Boulder, Colo.
- FOX, C. 1992. Lexical analysis and stoplists. In *Information Retrieval: Data Structures and Algorithms*, W. B. Frakes and R. Baeza-Yates, Eds. Prentice-Hall, Englewood Cliffs, N.J., 102–130.
- FRAKES, W. B. AND GANDEL, P. B. 1990. Representing reusable software. *Inf. Softw. Tech.* 32, 10, 653–664.
- FRAKES, W. B. AND NEJMEH, B. A. 1987. Software reuse through information retrieval. In *The 20th Hawaii International Conference on System Sciences*. IEEE Computer Society Press, Los Alamitos, Calif., 530–535.
- FRAKES, W. B. AND POLE, T. 1994. An empirical study of representation methods for reusable software components. *IEEE Trans. Softw. Eng.* 20, 8, 617–630.
- FURNAS, G. W. 1985. Experience with an adaptive indexing scheme. In *Human Factors in Computing Systems, CHI '85 Conference Proceedings*. ACM, New York, 131–135.
- FURNAS, G. W., LANDAUER, T. K., GOMEZ, L. M., AND DUMAIS, S. T. 1987. The vocabulary problem in human-system communication. *Commun. ACM* 30, 11 (Nov.), 964–971.
- GOMEZ, L. M., LOCHBAUM, C. C., AND LANDAUER, T. K. 1990. All the right words: Finding what you want as a function of richness of indexing vocabulary. *J. Am. Soc. Inf. Sci.* 41, 8, 547–559.
- HALASZ, F. G. 1988. Reflections on notecards: Seven issues for the next generation of hypermedia systems. *Commun. ACM* 31, 7 (July), 836–852.
- HARTER, S. P. 1992. Psychological relevance and information science. *J. Am. Soc. Inf. Sci.* 43, 9, 602–615.
- HENNINGER, S. 1994. Using iterative refinement to find reusable software. *IEEE Softw.* 11, 5.
- HENNINGER, S. 1995. Information access tools for software reuse. *J. Syst. Softw.* 30, 3, 231–247.
- HENNINGER, S., LAPPALA, K., AND RAGHAVENDRAN, A. 1995. An organizational learning approach to domain analysis. In the *17th International Conference on Software Engineering*. ACM Press, New York, 95–104.
- HENNINGER, S. R. 1993. Locating relevant examples for example-based software design. Ph.D. dissertation, Univ. of Colorado, Boulder, Colo.
- KINTSCH, W. AND GREENO, J. G. 1985. Understanding and solving word arithmetic problems. *Psychol. Rev.* 92, 109–129.
- LAKOFF, G. 1987. *Women, Fire, and Dangerous Things: What Categories Reveal about the Mind*. The University of Chicago Press, Chicago, Ill.
- MAAREK, Y. S. AND SMADJA, F. A. 1989. Full text indexing based on lexical relations, an application: Software libraries. In *Proceedings of SIGIR '89*. ACM, New York, 198–206.
- MAAREK, Y. S., BERRY, D. M., AND KAISER, G. E. 1991. An information retrieval approach for automatically constructing software libraries. *IEEE Trans. Softw. Eng.* 17, 8, 800–813.
- MAIDEN, N. A. AND SUTCLIFFE, A. G. 1992. Exploiting reusable specifications through analogy. *Commun. ACM* 35, 4 (Apr.), 55–64.
- MCCLELLAND, J. L. AND RUMELHART, D. E. 1981. An interactive activation model of context effects in letter perception: Part 1: An account of basic findings. *Psychol. Rev.* 88, 5, 375–407.

- MOZER, M. C. 1984. Inductive information retrieval using parallel distributed computation. ICS Rep. 8406, Inst. for Cognitive Science, Univ. of California—San Diego, La Jolla, Calif.
- OSTERTAG, E., HENDLER, J., PRIETO-DÍAZ, R., AND BRAUN, C. 1992. Computing similarity in a reuse library system: An AI-based approach. *ACM Trans. Softw. Eng. Methodol.* 1, 3, 205–228.
- PATEL-SCHNEIDER, P. F., BRACHMAN, R. J., AND LEVESQUE, H. J. 1984. ARGON: Knowledge representation meets information retrieval. In *Proceedings of the 1st Conference on Artificial Intelligence Applications (CAIA '84)*. IEEE Computer Society Press, Los Alamitos, Calif., 280–286.
- POULIN, J. S. AND CARUSO, J. M. 1993. A reuse metrics and return on investment model. In *Advances in Software Reuse*. IEEE Computer Society Press, Los Alamitos, Calif., 152–166.
- PRIETO-DÍAZ, R. 1985. A software classification scheme. Ph.D. dissertation, Tech. Rep. 85-19, Univ. of California—Irvine, Irvine, Calif.
- PRIETO-DÍAZ, R. 1991. Implementing faceted classification for software reuse. *Commun. ACM* 35, 5 (May).
- PRIETO-DÍAZ, R. AND ARANGO, G. 1991. *Domain Analysis and Software Systems Modeling*. IEEE Computer Society Press, Los Alamitos, Calif.
- PRIETO-DÍAZ, R. AND FREEMAN, P. 1987. Classifying software for reusability. *IEEE Softw.* 4, 1, 6–16.
- ROSE, D. E. AND BELEW, R. K. 1991. A connectionist and symbolic hybrid for improving legal research. *Int. J. Man Mach. Stud.* 35, 1, 1–33.
- SALTON, G. AND BUCKLEY, C. 1988. Term weighting approaches in automatic text retrieval. *Inf. Process. Manage.* 24, 3, 513–525.
- SALTON, G. AND BUCKLEY, C. 1990. Improving retrieval performance by relevance feedback. *J. Am. Soc. Inf. Sci.* 41, 4, 288–297.
- SALTON, G. AND MCGILL, M. J. 1983. *Introduction to Modern Information Retrieval*. McGraw Hill, New York.
- SHAW, M. AND GARLAN, D. 1996. *Software Architecture: Perspectives on an Emerging Domain*. Prentice-Hall, Englewood Cliffs, N.J.
- SIMOS, M., CREPS, D., KLINGER, C., AND LEVINE, L. 1995. Organization domain modeling (ODM) guidebook. STARS-VC-A023/011/00, Unisys Corp., Reston, Va.
- SMOLENSKY, P. 1988. On the proper treatment of connectionism. *Behav. Brain Sci.* 11, 1–23.
- SOMMERVILLE, I. AND WOOD, M. 1986. A software components catalogue. In *Intelligent Information Systems: Progress and Prospects*. R. Davies, Ed. Ellis Horwood Limited, Chichester, U.K., 13–32.
- SPARCK-JONES, K. 1972. A statistical interpretation of term specificity and its application in retrieval. *J. Doc.* 28, 1, 11–21.
- SPARCK-JONES, K. 1973. Index term weighting. *Inf. Storage Retrieval* 9, 619–633.
- STALLMAN, R. M. 1981. EMACS, the Extensible, Customizable, Self-Documenting Display Editor. *ACM SIGOA Newslett.* 1, 1/2, 147–156.
- THOMPSON, R. H. AND CROFT, W. B. 1989. Support for browsing in an intelligent text retrieval system. *Int. J. Man Mach. Stud.* 30, 639–668.
- WALZ, D. B., ELAM, J. J., AND CURTIS, B. 1993. Inside a software design team: Knowledge acquisition, sharing, and integration. *Commun. ACM* 36, 10 (Oct.), 62–77.
- WIEDERHOLD, G., WEGNER, P., AND CERI, S. 1992. Toward megaprogramming. *Commun. ACM* 35, 11 (Nov.), 89–99.
- WILLIAMS, M. D. 1984. What makes RABBIT run? *Int. J. Man Mach. Stud.* 21, 333–352.
- WIRFS-BROCK, R. J. AND JOHNSON, R. E. 1990. Surveying current research in object-oriented design. *Commun. ACM* 33, 9 (Sept.), 105–124.

Received July 1995; revised February 1996; accepted November 1996