



## 摘要

软件架构(Software Architecture)是控制软件复杂性、提高软件系统质量、支持软件开发和复用的重要手段。然而在现实当中很多早期开发的应用系统(通常称为遗留系统)和大量新近开发的软件系统并不存在系统架构的描述文档,或者即使存在,也但难以与系统实现保持同步更新,这就使得对这类系统的维护、升级、扩充和更新十分困难。因此,深入研究从系统实现中提取、分析、推演和重构软件系统架构的有效方法,对于延长软件系统的使用寿命、提高软件系统的维护效率、降低维护成本具有重要意义。

本文对目前的软件架构重构过程模型、重构方法和工具进行了系统的研究。根据现有的重构过程模型,归纳出了一般的软件架构重构技术框架,包括架构视点的确定、系统领域信息的提取、架构信息的抽象和架构视图的展示,从而为软件架构重构系统的设计和实现提供了基本的技术向导。

对软件架构重构系统设计的关键技术进行了深入研究,给出了一种基于矩阵的架构信息分析方法。该方法将工程领域的设计结构矩阵(Design Structure Matrix 简称 DSM)应用到架构重构中,使用 DSM 表示系统模块(构件)间的依赖关系,利用矩阵的划分算法重新划分矩阵,以识别系统架构信息和违背设计规则(如环依赖)的依赖关系。

按照软件架构重构的技术框架,设计并实现了一个基于 DSM 分析方法的架构重构工具——NEL 原型系统。实例分析表明,该方法能有效的识别出系统的架构信息,对于用户输入的设计规则,NEL 可以正确的辨别系统是否违背这些设计规则,从而能有效的维护软件系统的架构。

**关键词:** 软件架构 架构重构 逆向工程 依赖结构矩阵

## Abstract

Software Architecture (SA) is an important means to control the complexity of software systems, to improve software quality and to support software development and software reuse. However, in practice the architectural documentation of large number software systems frequently does not exist and even when it does exist, it is often out of sync with the implemented system -- these increasing the overall difficulty of software maintenance. Thus, the researches on the effective methods for extracting, analyzing, reasoning and reconstructing software architecture from available evidence are very important for extending the Life of Legacy system, improving efficiency and reducing the costs of software maintenance.

A systematic research about the current generic processes, methods, and tools for reconstructing software architectures is carried out. According to current generic processes, a generic technique framework which includes architectural viewpoint, data gathering, information abstract and information interpretation is summarized. The framework provides a basic technical guideline for performing architecture reconstruction tasks.

The key technique of software architecture reconstruction (SAR) is studied in detail. Then a novel method based on matrix for SAR is presented. The dependency structure matrix (DSM) was invented for optimizing product development processes. In this thesis, the idea of the DSM to the SAR is introduced; the matrix is used to represent the dependencies between modules. A variety of algorithms such as partitioning algorithm are available to help organize the matrix in a form that reflects the architecture patterns and problematic dependencies that violate the design rules.

According to the technique framework, the analysis has been implemented in a tool called NEL. Finally, a case study and evaluation for NEL is described. The results illustrate that base on the method we proposed we can easily identify and locate violations, and keep the code and its architecture in conformance with one another during software maintenance.

**Key words:** Software Architecture, Architecture Reconstruction, Reverse Engineering, Dependency Structure Matrix

## 西北大学学位论文知识产权声明书

本人完全了解学校有关保护知识产权的规定，即：研究生在校攻读学位期间论文工作的知识产权单位属于西北大学。学校有权保留并向国家有关部门或机构送交论文的复印件和电子版。本人允许论文被查阅和借阅。学校可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。同时，本人保证，毕业后结合学位论文研究课题再撰写的文章一律注明作者单位为西北大学。

保密论文待解密后适用本声明。

学位论文作者签名：\_\_\_\_\_ 指导教师签名：\_\_\_\_\_

时凯 房新益

---

## 西北大学学位论文独创性声明

本人声明：所呈交的学位论文是本人在导师指导下进行的研究工作及取得的研究成果。据我所知，除了文中特别加以标注和致谢的地方外，本论文不包含其他人已经发表或撰写过的研究成果，也不包含为获得西北大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

学位论文作者签名：\_\_\_\_\_

时凯

# 第一章 引言

本章首先介绍了课题的研究背景和意义，从而引出本文研究的主题；随后阐述了目前该论题的国内外研究现状和本文课题研究的主要内容；最后给出了论文的章节安排。

## 1.1 研究背景及意义

“如果一个项目的系统架构(包括理论基础)尚未确定，就不应该进行此系统的全面开发。只有对架构做出明确清楚的表述，才能使之在整个开发和维护过程中加以充分的利用<sup>[1]</sup>”。

软件架构(Software Architecture 简称 SA)是控制软件复杂性、提高软件系统质量、支持软件开发和复用的重要手段之一，是目前软件工程的一个重要的研究领域。最初，SA 概念的提出是为了解决从软件需求向软件实现的平坦过渡问题，早期的软件架构的研究主要集中在软件生命周期的设计阶段，随着软件工程的发展，对 SA 的研究开始超出软件设计阶段，逐步扩展到了整个软件生命周期<sup>[2]</sup>。SA 研究热点之一为软件架构的重构，架构重构就是根据现有系统的源代码和文档等各种可利用的知识重新构建出已实现系统的软件架构的过程<sup>[3]</sup>。架构重构主要解决以下两类问题：

### I. 软件维护中的程序理解。

当今软件系统的规模越来越大，结构也越变复杂，相应的对现有软件的维护代价不断的增加，而对系统内部结构的正确理解直接影响了系统维护的总的代价。在 Fjeldstad 和 Hamlen 的报告<sup>[4]</sup>中指出：在系统性能提高和纠错任务中，分别有 42%和 62%的时间花在理解活动上。

然而许多复杂系统由于在早期开发过程中没有考虑架构的设计，或者在升级等维护过程中没有同步更新架构文档，或者架构文档丢失等等原因没有真实反映当前系统架构的文档资料，此时去维护这么一个复杂而结构不明的系统，将大大增加软件维护任务的复杂性和困难程度，大大增加了维护代价。

### II. 软件生命周期中对系统架构的维护。

软件架构阻止或者支持系统的质量属性的实现,系统能否具有所期望(或要求)的质量属性是由其架构决定的<sup>[3]</sup>。因此应该对系统的架构进行维护,在软件开发和维护过程当中,保证实现和架构设计的一致性。

然而最初的 SA 研究往往只关注较高层次的系统设计、描述和性质验证,而对缩小从软件架构层次到系统实现层次的鸿沟关注不够<sup>[2]</sup>。大型的复杂软件系统在开发和维护过程中,由于其系统元素复杂的依赖关系,高层的 SA 设计模型和底层实现往往不一致。

本文以这两类问题为出发点,试图通过对软件架构重构技术的研究以解决如下问题:如何在开发和维护过程当中及时的反馈真实的软件代码状态,以及如何保证架构与实现的一致性,以维护不断演化的软件架构。

## 1.2 研究现状

软件架构重构是逆向工程在软件领域的一种形式,而软件逆向工程的研究已经有十多年的历史,已经形成了一定的研究规模。

关于逆向工程在 1994 到 2007 年间已经连续召开了 14 次国际会议 WCRE (Working- Conference on Reverse Engineering),关于程序理解 ICPC(International Conference on Program Comprehension)在 1993 到 2007 年间已经连续召开了 15 次会议。在软件工程、软件维护、面向对象技术等方面的主流国际会议上,近年来也对这些问题设立了相应的专题进行论文交流和讨论。卡内基梅隆大学软件工程研究所成立了专门的再工程中心,致力于逆向工程的研究。

在国内,随着人们对软件的后期维护的重视,软件逆向工程的研究也逐步展开,青鸟程序理解系统 JBPAS(Jade Bird Program Analysis System)就是有影响的研究成果之一<sup>[5]</sup>。近年来一些文章关注于设计恢复如<sup>[6][7]</sup>,然而对软件架构重构的整个过程比较系统的研究还很少,目前的大多数工具也仅仅是设计的恢复而没有考虑到对软件架构的重构。

在国外,软件逆向工程是作为对软件维护的一部分出现的,主要是通过逆向工程理解程序,对系统进行维护、迁移和进化遗产系统<sup>[5]</sup>。目前在发表的文章中提出了一些支持软件架构重构的方法和技术,并且已经成功开发出很多架构重构工具,如有近 10 年历史的逆向工程工具 Rigi<sup>[8]</sup>以及获得 2006 Jolt 卓越奖

来自于 Lattix 公司的 LDM<sup>[9]</sup>等。近年来许多学者在逆向工程的架构重构应用领域上发表了文章，如<sup>[10][11]</sup>等。

### 1.3 主要工作内容

本文的工作主要以国内外近年来的相关工作为基础，针对软件架构重构过程模型、方法以及关键技术进行系统的研究，在此基础上设计并实现了以依赖结构矩阵(DSM)和关系查询技术为信息推论方法的软件架构重构系统 NEL。本文研究的主要内容包括以下几个方面：

· 架构重构的理论基础以及现实意义。包括软件架构、逆向工程、软件维护和遗留系统。

· 对现有的软件架构重构过程模型、重构方法和工具进行了系统的研究。根据现有的重构过程模型，归纳出了一般的软件架构重构技术框架，包括架构视图的确定、系统领域信息的提取、架构信息的抽象和架构视图的展示

· 在重构关键技术框架的基础之上，本文给出了一种使用依赖结构矩阵和包模式相结合的信息推论方法，该方法可以很好的提取系统架构信息并发现违背系统架构设计的规则。

· 基于以上的理论基础以及对所提出的信息推论方法的分析，设计并实现了架构重构系统 NEL。最后通过实例分析对该信息推论方法以及 NEL 进行了客观的测试以及评价。

### 1.4 论文结构及章节安排

本文接下来的章节安排具体如下：

第二章：相关背景。阐述了软件架构重构的理论基础和现实意义，包括软件架构、逆向工程、软件维护和遗留系统。

第三章：软件架构重构。详细分析 Cacophony 和 Symphony 架构重构的过程模型，对目前架构重构的方法和工具进行介绍和评估。

第四章：NEL 架构重构的关键技术。分析软件架构重构过程中的关键技术包括视点的确定、信息提取、信息推论和可视化，在这基础之上阐述了 NEL 所使用的相关技术。

**第五章：NEL 系统的设计与实现。**介绍了 NEL 系统的分析、设计以及实现，给出了一个实例研究，最后进行了系统的评价。

**第六章：总结与展望。**对本文的工作进行了系统总结，并对未来工作做了进一步的展望。

## 第二章 相关背景知识

本章介绍了软件架构重构的基础理论以及现实意义，包括软件架构、逆向工程、软件维护和遗留系统。

### 2.1 软件架构

#### 2.1.1 软件架构的定义

虽然软件架构(Software Arcitecture 简称 SA)得到了广泛的研究和应用，但是目前还没有对 SA 标准的定义，在 CMU SEI<sup>[12]</sup>上许多学者给出了软件架构的定义：

Len Bass 等著作<sup>[3]</sup>中对 SA 的定义：某个软件或计算机系统的软件架构是该系统的一个或多个结构，他们由软件元素、这些元素的外部可见属性以及这些元素之间的关系组成。

IEEE 1471<sup>[13]</sup>推荐的 SA 定义为“一个系统的基本组织，体现在组成系统的各构件、构件的相互关系、构件与环境的关系，以及指导构件设计和随时间演进的原则当中”。

Soni, Nord 和 Hofmeister 在文献<sup>[14]</sup>中提出“对于每一种结构，从不同的角度去描述系统：概念架构从系统的主要设计元素和它们之间的关系的角度去描述了该系统，互连模块架构包括系统功能的分解和层次化，运行架构描述系统的动态结构，代码架构描述了在开发环境中代码、二进制码和程序库是如何组织在一起的”。

另一方面，对于如何理解软件架构，一些学者在实践过程中提出了他们对架构的理解：

Jean Marie Favre<sup>[10]</sup>在对超大型系统的研究中得出的结论：对于大规模的软件开发，软件架构的概念要比学术上的定义要复杂的多；软件架构的本质是什么取决于公司(架构涉众团体)的文化和他们自己对软件工程的理解；这些理解一部分以非实质的形式在公司里共享，另一部分以实质的资料形式通过支持软件开发和演化的一系列开发工具表现出来。

Ralph Johnson<sup>[15]</sup>提出：“一个更好的定义应该是：大多成功的软件项目中，开发专家对系统的设计有一个共用的理解，这个理解就叫‘架构’。这些理解包括如何将系统分解为组件和组件之间是如何通过接口来联系的。”Johnson认为这个是个更好的定义，因为它清楚的指出了架构是一种‘社会构建’。

Ric Holt 文献<sup>[16]</sup>中提出：对于在系统中开发人员共享他们的方法思想，软件架构非常有用。这种观点使得我们关注如何去思考架构和如何去优化我们表述架构的方式。

本文采用 Len Bass 等对架构的定义，认为系统的架构是多结构的，他们由软件元素，这些元素的外部可见属性以及这些元素之间的关系组成。而对于架构的本质是什么，这里采用 Jean Marie Favre 和 Ralph Johnson 的理解：软件架构的本质是什么取决于公司(架构涉众团体)的文化和他们自己对软件工程的理解；开发专家对系统的设计有一个共用的理解，这个理解就叫‘架构’。

### 2.1.2 软件架构的意义

良好的架构对软件系统在软件生命期中的各个阶段都具有重要意义，这里从下列几个方面说明了软件架构的重要性<sup>[3]</sup>：

- 涉众之间的交流。软件架构是一种常见的对系统的抽象，绝大多数系统的涉众都以此作为彼此理解、协商、达成共识或相互沟通的基础。
- 早期的设计决策。软件架构是所开发系统的最早设计决策的体现，而这些决策对系统得后续开发、部署和维护具有重要的影响。这也是能够对所开发系统进行分析的最早时间点。大量实践统计表明：大规模系统软件开发中 70% 的错误是由需求和软件设计阶段引入的。
- 可传递的系统抽象。软件构架是关于系统构造及系统各元素工作机制的相对较小、却能突出反映问题的模型。这种模型可以在多个系统之间传递，特别是可以应用到具有相似质量属性和功能需求的系统中，并能够促进大规模的重用。

### 2.1.3 软件架构视图

一个软件架构为不同的架构涉众所用，不同涉众有不同的需求，他们需要的信息也不相同，如用户、客户、软件架构师，开发人员和维护人员对软件架

构有着不同的视图。用户关注于架构的功能是否能完全实现，而开发人员和维护人员关注于代码的组织和模块的关系等。尽管这些视图是不同的，但是它们从不同角度共同描述了软件的架构。

许多文献提出不同的软件架构的视图，例如：Soni 等在文献<sup>[14]</sup>中提出的概念视图、模块互连视图，运行视图和代码视图；Len Bass 等在著作<sup>[3]</sup>中提出的模块视图、组件-连接器视图和分配视图。下面给出对 Kruchten<sup>[17]</sup>提出的“4+1”视图的描述(如图 2-1)：

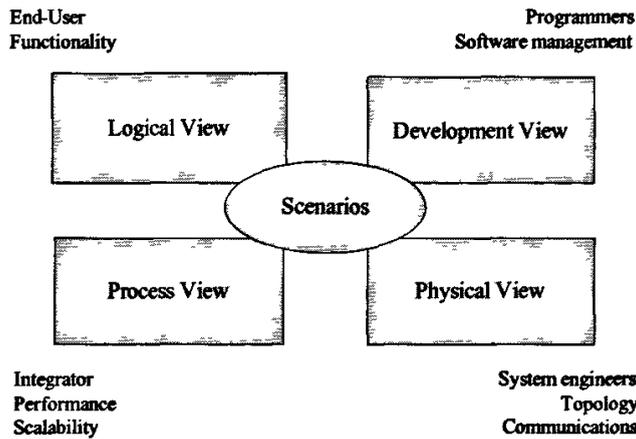


图 2-1 “4+1” 模型视图<sup>[17]</sup>

- 逻辑视图，系统被分解成为一个“关键抽象”的集合，这种“关键抽象”在面向对象中表示为对象或者对象类。这种分解并不仅仅是为了功能分析，也是为了在系统的各个不同组成部分当中识别出共同的机制和设计元素，通常用类图去表示一组类的集合和它们之间的逻辑关系。
- 开发视图，系统的开发架构由模块和子系统以及它们之间的导出(Export)和导入(Import)关系所表示。开发视图关注软件开发环境当中模块的组织结构，软件被封装在一些小模块——程序库或者子系统，这些模块可以由一个或者少数几个开发人员进行开发。开发视图可作为需求的分配、团队工作分配（甚至开发团队的组织）、支出的预估和计划，监控项目的开发进程和合理化重用的基础。
- 进程视图，主要关注一些非功能性的需求，如系统性能、可用性、并发、分布、系统完整性，容错等。这里的进程为一组可以独立运行的任务，表示了系统动态运行结构。

- 物理视图，该视图将其他元素映射到了处理和通信节点上。主要考虑非功能性需求，如可用性、可靠性，吞吐量和可扩展性等。

### 2.1.4 软件架构描述的概念模型—IEEE Std 1471

IEEE 公布了软件密集系统架构描述的建议准则，简称 IEEE 1471，该准则不说明架构应该具有什么样的视图，而只是说明该如何确定这些视图，它正成为架构描述的主要标准<sup>[18]</sup>。如图 2-2 为 IEEE 1471 定义的架构描述概念模型图。

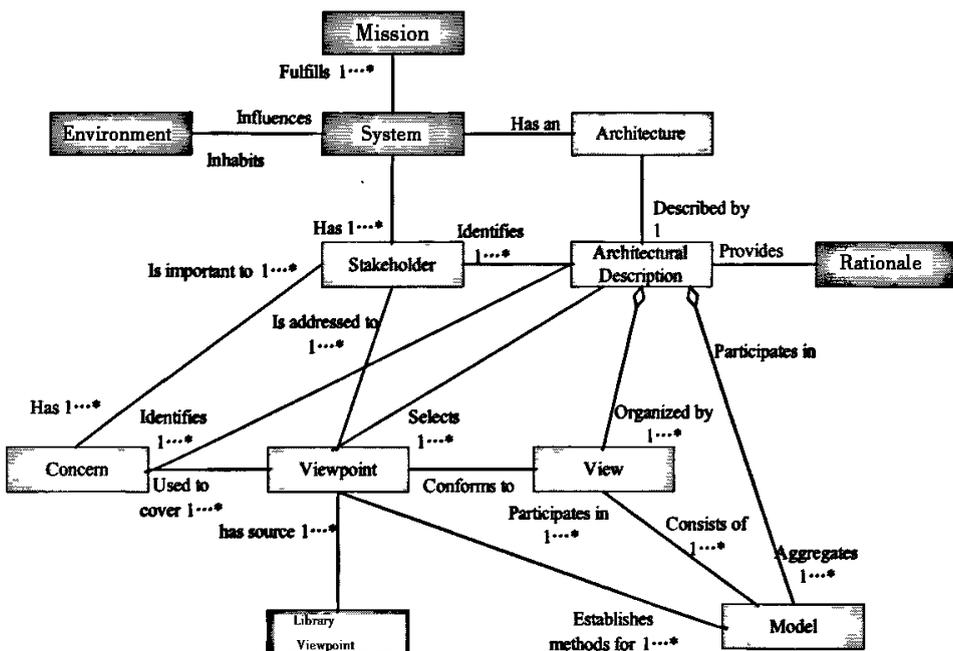


图 2-2 IEEE 1471 架构描述的概念模型图<sup>[13]</sup>

一个软件应用体系结构的规格说明成为体系结构的描述(即架构描述)<sup>[18]</sup>。

在 IEEE 1471 的架构概念模型中：

系统(System)拥有一个或者多个风险承担者(Stakeholder，本文称为架构涉众)，一些典型的涉众为，客户、用户、开发人员、设计人员、维护人员、架构师、开发商，策划人等<sup>[13]</sup>。每个涉众对系统有一个或者多个关注问题(Concern)。

系统拥有一个架构(Architecture)，架构通过架构描述(Architectural Description)来描述。架构描述被定义为“记录一个架构的产品集合<sup>[13]</sup>”，架构描述能够分解为一个或多个视图，每一视图都要处理(cover)一组相关的问题(Concerns)。视图为“当由一个选定的视点去看待一个系统时的所见<sup>[19]</sup>”，视图

由视点确定是视点的实例，视点为“一个视点为我们看待一个系统的方式。视点包括了如何去创建和分析某个特定的视图的规则。它是在多种架构描述的基础上能够重用的视图的模板<sup>[19]</sup>”。图 2-3 展示了视图和视点之间的关系：一个视图展现了视点中定义的系统架构的关注点，而视点则定义了用于描述视图的语言和方法。

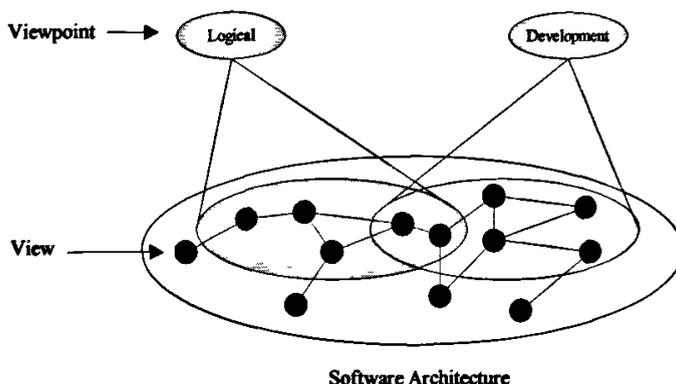


图 2-3 视点和视图之间的关系

一个架构描述选择一个或者多个视点，而视点的选择取决于架构涉众的关注点和他们需要解决的问题。视点可以和架构描述一起定义，但是也可以是定义于其他地方而在此架构描述中使用，这些外部定义的视点在 IEEE 1471 称为视点库(Library ViewPoints)。

一个视图包含了一个或者多个模型(Model)，一个模型也可以被包含于多个视图。软件系统的模型是系统知识的抽象表示，模型反映系统的某些选定的方面，如系统的结构、行为，操作或其他特征等的近似或理想化的表示，模型体现为图表、公式，文字描述或他们的组合<sup>[18]</sup>。

总的来说，对于 IEEE 1471 最主要的元素为：涉众、关注点、视点，视图以及模型。图 2-4(IEEE 1471 简化图)清楚的描述了它们之间的关系。

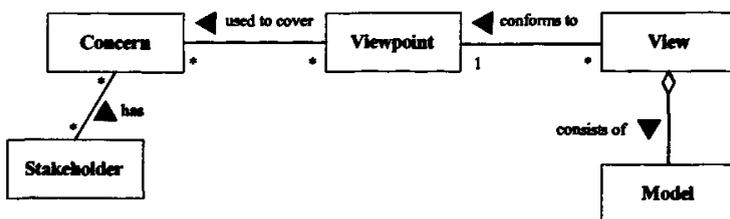


图 2-4 IEEE 1471 架构描述的简化图

## 2.2 逆向工程

软件架构的重构是逆向工程的一种形式，作为软件架构重构的理论基础本节介绍了逆向工程的定义、相关概念、规范活动以及程序理解。

### 2.2.1 逆向工程定义以及相关概念

术语“逆向工程”(Reverse Engineering)来自硬件领域，是通过检查样品开发复杂系统规约的过程，主要指研究他人的系统，发现其工作原理，以达到复制硬件系统的目的，但随着软件业的发展，“逆向工程”这一术语被引入软件工程领域，软件逆向工程可以用于描述揭示已有系统工作原理的过程(如图2-5)，或者是用于描述创建现有文档的联机文档的过程等<sup>[20]</sup>。Chikofsky 和 Cross<sup>[21]</sup>将逆向工程定义为“通过分析目标系统，识别系统各个组件以及它们之间的关系，从而抽取和创建系统的抽象和设计的信息”。

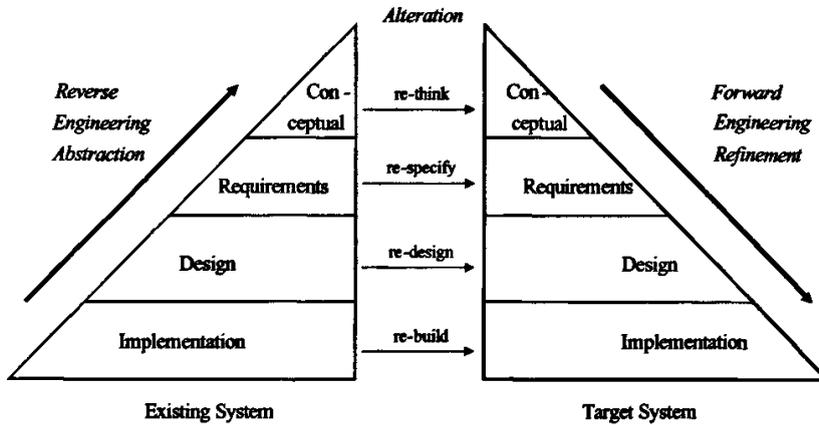


图 2-5 逆向工程模型<sup>[22]</sup>

逆向工程的主要目的是不断增加对系统的理解，任何开发人员对一个未知的系统总是需要花费大量的时间去理解。根据 Fjeldstadt 和 Hamlet 的报告<sup>[4]</sup>，一个维护系统的维护人员分别有 47%和 62%花费在理解程序上。

根据文献<sup>[21]</sup>下面列举了与逆向工程紧密联系的术语的定义：

- 正向工程(Forward Engineering)。是从系统高层抽象和独立于设计的逻辑实现转表为物理实现的过程。按照问题定义、可行性分析、需求分析、概要设计、详细设计，编码和测试的软件生命周期顺序开发系统的过程。

- 再文档(Redocumentation)。在同一抽象层次上创建或者修改相同语义的表示。是逆向工程最古老的类型之一。由于种种原因，现有的文档可能不够充分、正确和详细，因而源代码本身成了系统客观可靠的信息源。再文档就是利用已有的源代码为软件系统逆向生成精确的文档。
- 设计恢复(Design Recovery)。是结合系统的领域知识、外部信息、演绎和模糊推理去恢复更高层次的抽象表示。是从设计角度深入理解系统的方法。
- 重构(Reconstruction)。是在保持系统外部行为的前提下改变同一抽象层次上的表示。即在同一抽象层次把系统从一种表示方式转换到另一种表示方式。
- 再工程(Reengineering)。通过对目标系统的检查和改造，把现有系统重新组合成新的形式。它将逆向工程、重构和正向工程组合起来，根据对系统更深层次的理解将其重构为另一种形式的软件产品。

图 2-6 展示了这些过程在整个软件开发过程中的关系。

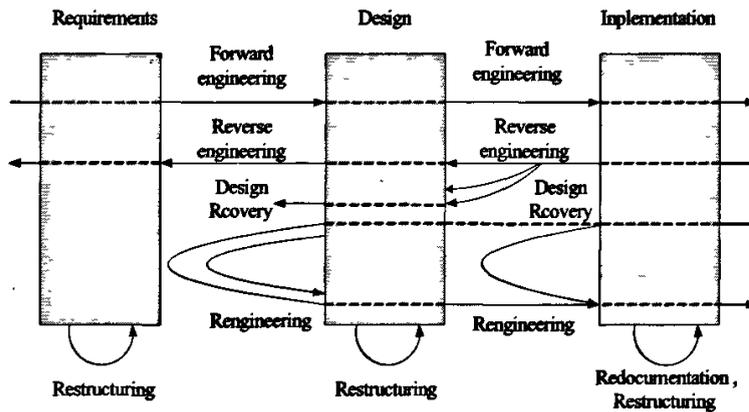


图 2-6 在软件开发周期中的逆向工程的各种活动<sup>[21]</sup>

### 2.2.2 逆向工程的规范活动

软件逆向工程并不改变目标系统，它只是一个检查的过程，而不是一个修改的过程。软件逆向工程通过标识系统相关对象并发现这些对象之间的关系，对系统做出抽象的表示，从而辅助对系统的理解。它涉及的对象可分为三类<sup>[23]</sup>：

- 数据(Data), 作为学习、推理和讨论的实际信息,
- 知识(Knowledge), 所有可知内容的总和, 包括数据和从数据中推导出的关系与规则。
- 信息(Information), 相互交织的交流知识, 这些信息包括所有的数据和知识, 以及软件工程人员的各种非正式的思路和建议。

基于这三类对象, Scott R.Tilley 等<sup>[23]</sup>总结了软件逆向工程的三个规范活动: 数据收集、知识组织和信息浏览。

### 2.2.2.1 数据收集

从目标系统进行数据收集是软件逆向工程的一项基本活动。数据收集的原始数据是构造和浏览高层抽象的基础, 所以它通常是逆向过程的第一步。数据收集时应该遵循以下原则<sup>[20]</sup>:

#### 1. 采用成熟的技术。

数据收集所采用的技术包括静态分析、动态分析和获取非正式数据(如调查)等。其中最为常用的是分析程序源代码, 构造带有语法单位及其依赖关系的抽象语法树, 在编译领域中, 语法分析和交叉索引等技术已经很成熟。采用基于编译的成熟技术, 可以得到预期的结果, 使收集的数据更加准确和可靠。

#### 2. 利用多种数据源。

一般来说遗留系统有四种数据来源: 系统源代码; 系统行为, 即系统用户所看到的系统功能和性能; 系统文档, 包括需求文档、分析文档、设计文档, 用户手册和源代码注释等; 系统涉众, 如架构师和开发人员等。系统的多种数据从不同的方面刻划了系统对象及其关系, 从而为更深入详细理解系统提供了基础。

#### 3. 过滤数据。

对于大型系统, 所收集的数据量可能是巨大的, 以至于超出了人们的理解能力。数据过滤是从丰富的数据源中抽取所选择的对象及其关系, 因而, 数据过滤在辅助系统理解中有着重要作用。

### 2.2.2.2 知识组织

对成功的程序理解而言, 所收集的数据必须用合适的数据模型保存起来,

以便实现有效的存储和检索，辅助对系统相关对象以及他们之间相互关系的分析，并反映用户对系统特性的理解。数据模型捕捉了系统的本质属性及其关系，它应该可以有效地支持知识组织，因此对数据模型有下列要求<sup>[20]</sup>：

### 1. 知识的组织。

知识的组织是面向人的而不是面向计算机的：传统的数据模型，如层次模型、网络模型和关系模型，都是从适于计算机操作的角度上建模组织数据的。而软件逆向工程的建模则要求以满足人们的理解为出发点来捕捉系统组件及其相互关系，它强调知识组织，而不是数据组织，知识组织是按照系统实体及其语义关系建模的。面向对象的数据模型所刻划的系统组件及其相互关系是面向人的，而不是面向计算机的，其抽象机制能够帮助软件工程人员有效地组织目标系统的知识。

### 2. 领域知识。

当前，逆向工程技术主要以程序分析技术为基础，但是程序结构自身并不足以反映程序所应用的问题域。领域分析为解决这个问题的有效途径之一。领域分析是指通过标识，组织和表示领域元素及其组成结构来揭示它们在问题域中的联系。因此数据模型应该按领域组织目标系统的知识，识别系统中的标准组件，从而为软件逆向工程提供有效的辅助支持作用。

### 3. 可扩展性。

对大型系统而言，软件逆向工程产生的数据是巨大而复杂的，为了实现有效的存储，检索和分析，就必须使用可扩展的软件逆向工程数据库来存储软件逆向工程中所得到的知识。可扩展的软件逆向工程数据库存储所有的数据与知识，并支持对系统的增量分析，所以它能够提高系统理解的效率，特别是在大型系统的理解中。

#### 2.2.2.3 信息浏览

因为大多数程序理解的活动都是在信息浏览时进行的，因而信息浏览可能是软件逆向工程三个规范活动中最重要的一个。信息浏览包括遍历，分析和表达三个活动<sup>[20]</sup>：

##### 1. 遍历(Navigation)。

对大型系统而言，软件逆向工程所产生的信息结构并不是线性的，而是一个相互交织的多维信息网。网中的链接代表了软件逆向工程所产生的组件之间的层次关系、继承关系、数据流，控制流和其它关系。遍历采用定向(如地图、多窗口、历史树，路径和复合节点等)和高级模式匹配等机制，辅助人们浏览软件逆向工程知识组织阶段所产生的多维信息结构<sup>[20]</sup>。

## 2. 分析(Analysis)。

分析多维信息结构是程序理解的关键。分析从原始数据中推导并抽取那些并不显式存在的信息，并产生关于系统的深层视图。为了辅助用户从多方面理解系统，可以利用编程语言机制对分析方法进行编码，并允许用户根据特定的任务开发特定的分析方法。

## 3. 表达(Presentation)。

表达是以可视化方式表示分析的结果。表达是在研究人们程序理解时所采用的认知策略基础上，创建结构化视图来表现分析结果，以达到一目了然的效果。

### 2.2.3 逆向工程中的程序理解

程序理解是逆向工程中的关键技术之一，是软件逆向工程主要的实现手段和活动，它贯穿了整个逆向工程，并且是决定软件逆向工程成败的关键。通俗地讲，程序理解就是通过一定的设施和方法来弄清楚一个程序是“做什么的”以及“如何做的”。可以把程序理解看作这样的任务：以软件维护、升级和再工程为目的，在不同的抽象级别上建立基本软件的概念模型，包括从代码本身的模型到基本应用领域的模型，即建立从问题/应用域到程序设计/实现域的映射集。

程序理解策略是程序理解在总体上所使用的方法和采用的理解过程，程序理解策略主要有<sup>[20]</sup>：

- 自底向上策略。构造从实现领域到问题领域的映射集(Mapping)，由相似(或相关)属性(或特性)的程序聚集而形成更高层的概念子系统。它特别适合于以再文档为目标的软件逆向工程，通常对新应用领域的分析使用这种策略。
- 自顶向下策略。构造从应用领域到实现领域的映射集，它可以产生几个中

间层表示。分析过程是由期望驱动的建立、确认和精化对程序所作的假设的过程，能更好地适应目标制的程序理解，例如一个指定的程序维护任务。这种策略依赖于工程师的专门领域知识，适合对已熟知领域的分析。

- 混合策略。这种策略是自底向上和自顶向下策略的结合，兼具二者的特点。混合策略有很多种，例如对不同的应用脚本都具有可适应性的同步细化策略。

## 2.3 软件维护

任何软件产品在其生命期内都会因为条件的变化而发生更改，IEEE 对软件维护的定义是“对已交付的软件系统进行修改错误、改进性能或使得其适应修改后的新环境的过程<sup>[24]</sup>”。ISO/IEC 12207 对软件维护的解释是“软件因为问题、改进或适应的需要而对代码或者相关文档的修改活动<sup>[25]</sup>”。

ISO/IEC 12207 标准中<sup>[25]</sup>，生命周期过程分成基本过程(Primary)、支持过程(Supporting)和组织过程(Organizational)三大类，其中基本过程是过程的原动力，提供生命周期的主要功能，基本过程由 5 个过程组成：获得、供应、开发、操作和维护。由此可见软件维护是软件生命周期中重要的一部分。

目前许多工程都是基于原有系统的维护和进化，维护成本也随之迅速提高。根据Koskinen的调查<sup>[26]</sup>，在2000年已经有大约2500亿行代码在维护中，每7年维护的代码量增加一倍。如表2-1显示目前维护和管理软件的演化的费用占有总的软件投入超过90%。

Year	Proportion of software maintenance costs	Definition	Reference
2000	>90%	Software cost devoted to system maintenance & evolution / total software costs	Erlikh (2000)
1993	75%	Software maintenance / information system budget (in Fortune 1000 companies)	Eastwood (1993)
1990	>90%	Software cost devoted to system maintenance & evolution / total software costs	Moad(1990)

1990	60-70%	Software maintenance / total management information systems (MIS) operating budgets	Huff(1990)
1988	60-70%	Software maintenance / total management information systems (MIS) operating budgets	Port(1988)
1984	65-75%	Effort spent on software maintenance / total available software engineering effort.	McKee (1984)
1981	>50%	Staff time spent on maintenance / total time (in 487 organizations)	Lientz & Swanson (1981)
1979	67%	Maintenance costs / total software costs	Zelkowitz <i>et al.</i> (1979)

表2-1 软件维护代价比例<sup>[26]</sup>

开发人员进行维护过程通常是从对问题和程序的理解开始的。根据 Lehman<sup>[27]</sup>的持续变化(Continuing Change)和复杂性不断增长法则(Increasing Complexity)，软件系统在期生命周期内是不断的有变化的，而且复杂性不断增长除非采取必要的措施进行维护。随着系统的不断演化，程序结构越来越难理解，根据 Fjeldstad 和 Hamlen 的报告<sup>[4]</sup>，在系统性能提高和纠错任务中，分别有 42%和 62%的时间花在理解活动上。

## 2.4 遗留系统

Brodie 和 Stonebraker<sup>[28]</sup>给出的遗留系统(Legacy System)的定义为“任何不得不修改和演化以适应不断变化的业务需求的信息系统”。Andreas.Wierda 在文献<sup>[29]</sup>中给出了遗留系统存在的问题：

- 文档缺乏或者过时。缺乏随时更新的文档会增加维护的难度，当系统经历了多次修改后，文档已经不能够反映系统当前的实现状况。
- 缺少单元测试和系统测试。缺少单元测试和系统的测试是非常危险的，因为无法去确定对系统的一个更改是否会引发其他功能模块的问题。
- 对系统的理解有限。对系统的理解主要是系统总体的结构和重要部分的实现细节，缺乏这些理解会阻碍系统的演化。缺乏理解的原因有人员流失，缺少实时更新的文档和缺少测试等，这些原因会导致系统在演化过程中质

量大大下降。

- 非期望的依赖关系。当系统需要修改时，比如修正 Bug，添加了本不应有的依赖关系，如果这些情况常常发生，那么将会导致架构的破坏，从而架构的功能属性不再起作用。
- 大量重复代码。当系统中存在大量的重复代码，导致对系统的一处更改需要在多处地方，增加了维护的难度。

## 2.5 本章小结

本章阐述了软件架构重构技术的相关背景知识，包括软件架构、逆向工程、软件维护和遗留系统。这些背景知识可分为两部分：其一为架构重构技术的理论基础，包括软件架构理论和逆向工程理论；另一部分为架构重构技术的现实意义，包括软件维护和遗留系统。

## 第三章 软件架构重构方法

软件架构重构是一个从已实现系统中得到其设计架构的过程。本章讨论架构重构的过程模型、方法和工具，并对现有的方法和工具进行评价。

### 3.1 架构重构的典型问题

在架构重构的过程当中，一个已实现系统的软件架构被恢复，恢复后的架构为“实际构建”架构(As-build architecture)。在这个过程当中模型从代码和其他相关资料中提取出来，提取出来的实体用于表述系统的高层抽象。架构重构的实施通常是因为现存系统存在不确定性。

O'Brien 等在文献<sup>[30]</sup>中总结了在实施架构重构的过程当中遇到的一些主要的问题：

- 视图集合(View set)。如何去选择一组架构的视图，从而既足够描述系统并满足所有涉众(Stakeholders)所关注的问题需求。
- 强制性的架构(Enforced architecture)。如何解决在架构设计到代码实现的过程当中，一致性信息丢失(导致设计和实现的不一致)。即在开发过程当中，如何去强制性要求开发符合设计的架构。
- 质量属性的变化(Quality-attribute changes)。如何决定架构元素和质量属性之间的关系。通常被用于决定如何使用架构模板去满足质量属性和其对系统的影响。
- 公共和可变的部分(Common and variable artifacts)。目的是在一些相似的软件系统产品中找出公共的部分,从而可以减少代价支出。
- 二进制组件(Binary components)。当使用商用的通用组件 COTS 作为信息源的时候，如何进行架构的重构？
- 混合语言(Mixed language)。当一个系统由多于一种语言实现时，如何去进行架构的重构。

目前绝大多数的软件重构方法或者重构工具都只是解决以上问题背景中的一个或者几个问题，本文主要解决强制性的架构(Enforced architecture)问题。

## 3.2 架构重构过程模型

在 Len Bass 的《软件架构实践》<sup>[3]</sup>中，软件架构重构可分为 4 个活动，这些活动以迭代方式进行：信息提取；数据库构造；视图融合；由于大多数的架构恢复过程并没有数据库构造的活动，可以将信息提取和数据库构造可以结合成一个活动，称为视图提取<sup>[29]</sup>。由于架构拥有多视图，在架构重构之前需要确定视图集合，这样有了以下 4 个活动：

- 定义目标视点，目的是确定要恢复的架构所需要包括的所有视点，这些视点应该满足涉众的需求。
- 视图提取，该活动的目的是从各种源中(例如源代码、编译文件、设计文档等)提取信息。
- 视图融合，视图融合将视图提取活动中得到的信息组合在一起，以生成该架构的一个内聚的视图。该活动包括定义和处理所提取的信息，以协调并建立元素之间的连接。
- 重构，在重构活动中，主要工作是构建数据抽象和各种表示以生成架构表示。该活动分为模式识别和可视化交互两个部分。

一些架构恢复方法和工具包含了上述四个过程，下面介绍近年来在软件架构重构领域新提出的两个相对通用的架构重构过程模型：

### 3.2.1 Cacophony 元模型驱动的架构重构

Cacophony<sup>[10]</sup>的主要目的是提供一种通用的架构重构方法过程，帮助重构人员在一个特定的环境中确定软件架构的真实含义，并且为架构重构的任务提供充分的支持。Cacophony 结合了逆向工程、IEEE 1471 Standard 的架构概念模型和对象管理组织 OMG 的模型驱动架构 MDA，提出了一种通用的架构重构过程模型——元模型驱动的架构重构模型(Metamodel-Driven)。该模型的主要思想是如何利用元模型来解决以下问题：a)确定架构视点；b)将视点和现有系统相关信息关联起来；c)驱动整个架构重构过程。

#### 3.2.1.1 Appliware 和 Metaware

在 Cacophony 中，软件(Software)可分为 metaware(M1 层)和 appliware(M2 层)两类，其中 appliware 为一般的应用软件，而 metaware 为用于开发应用软件

(appliware)和管理应用软件演化的软件。表 3-1 给出了常见的 metaware 和 appliware 的列表。即 Software = Metaware + Appliware。对应于 Metaware 和 Appliware 的实体分别为 MetawareItem 和 AppliwareItem。

	Appliware(M1)	Metaware(M2)
Programming in-the-Small	Programs, binaries...	Compilers, pretty printers, interpreters, grammarware tools, IDEs...
Programming in-the-Large	Build files, log files, release information, product descriptions, application portfolios...	Component technologies, configuration manager, architectural tools, build tools, bug tracking systems, product managers, impact analysis tools...

表 3-1 Appliware 和 Metaware<sup>[10]</sup>

### 3.2.1.2 元模型驱动架构模型

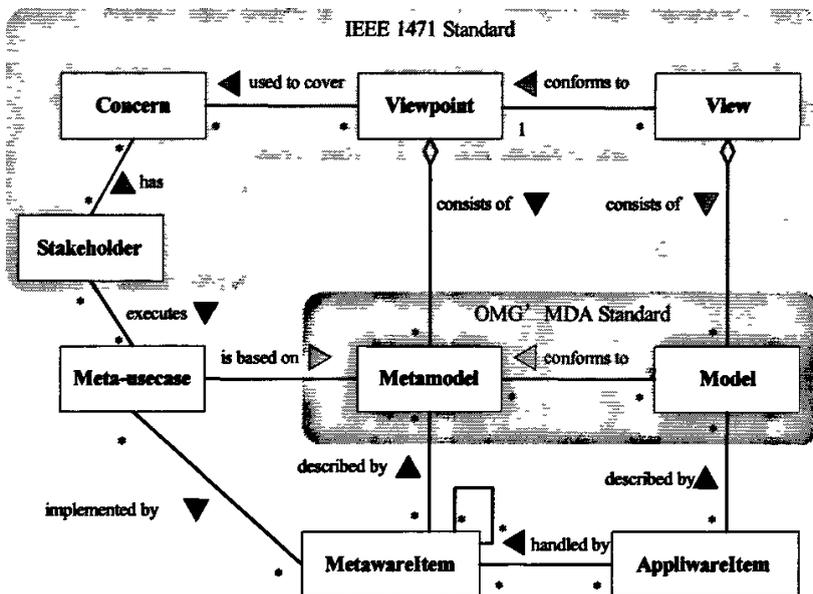


图 3-1 元模型驱动架构重构概念模型<sup>[10]</sup>

图 3-1 为 Cacophony 的元模型驱动的架构重构模型图，包含了三个部分：

#### 1. IEEE 1471 Standard 中架构的概念模型

这一部分阐述了 IEEE 1471 Standard 中对视图的概念模型，说明了架构由多个视图表示，每一个视图又由视点来约束和定义，其中一个视图可由一个或者多个模型构成；架构的多个视点又依赖于涉众以及他们的关注点。

#### 2. OMG 的模型驱动架构 MDA

MDA 是“模型驱动架构”(Model Driven Architecture)的缩写,来自 OMG。其关键之处是,模型在软件开发过程中扮演了非常重要的角色。在 MDA 中,软件开发过程是由对软件系统的建模行为驱动的。其中模型(Model)和元模型(Metamodel)是 MDA 中非常重要的两个概念,在“从古埃及到模型驱动工程”系列文章<sup>[31]</sup>中,深刻阐述了他们的概念以及关系:通过模型,我们能不用直接面对系统而够去研究它,一个模型不要求是描述了系统的方方面面,只需要满足特定的目的。元模型是建模语言的模型,在 MOF 标准中对元模型的定义“元模型是一个模型,该模型定义了一个模型的表述语言”。如图 3-2 利用具体的 C 应用程序(模型)和 C 程序语言(元模型)阐述了元模型和模型之间的关系“ComformsTo”。

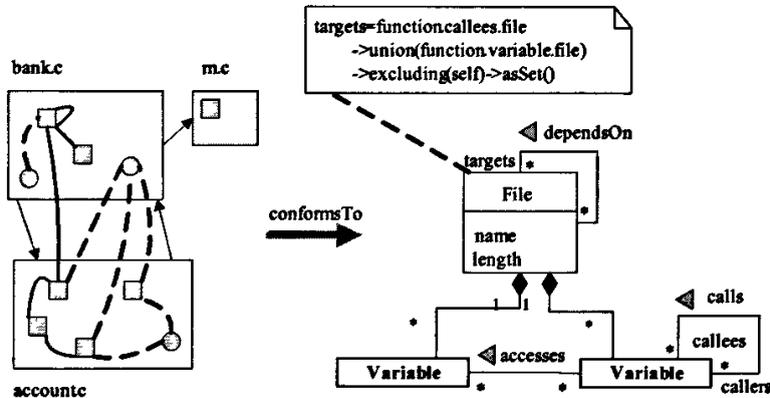


图 3-2 模型(左)与元模型(右)<sup>[10]</sup>

根据 IEEE1471 的概念模型,一个视图包括了多个模型,但是并没有提及元模型<sup>[10]</sup>。所以 Cacophony 将 IEEE1471 和 OMG 的 MDA 结合起来,提出用元模型(Metamodel)来描述架构的视点,认为当需要进行架构重构时,通过发掘架构的元模型来确定视点。

### 3. Metaware 逆向工程

前一部分提出利用 Metamodel 去描述 Viewpoint,然而这些 Metamodel 从哪里来?如何将他们和现有的资料联系在一起?软件公司是不是已经事先定义了视点和架构元模型?在现实当中,大规模的软件公司在长期的运作过程中形成软件架构的概念,并开发了一系列的“Metaware”去支持他们对架构的理解。如在图 3-2 中,左边的模型(Model)可以从应用软件(Appliware)中提取出来,这个提取过程称之为应用软件逆向过程。同样的右边的元模型(Metamodel)可以从

Metaware 中提取出来,称为 Metaware 逆向工程。在 Cacophony 中,认为 Metaware 揭示了软件公司对软件架构的概念理解,软件架构重构是基于对现存的 Metaware 进行逆向工程。

Cacophony 将 Metaware 和 Appliware 结合到 MDA 和 IEEE 1471 中(如图 3-1),提出首先利用 Metaware 获取 Metamodel,进而得到目标视点。

### 3.2.1.3 Cacophony 架构重构过程

Cacophony 架构恢复过程包括以下步骤:

- Metaware 领域分析和现有资料分析,目的为重构软件架构的元模型。
- Metaware 需求分析,架构涉众和需求的确定。
- Metaware 详细说明,详细说明了需要构建的 Metaware。
- Metaware 的实现、运行及其演化。

### 3.2.2 Symphony 视图驱动的架构重构

目前有许多技术用于特定的架构视图的重构,然而很少提到在重构过程中如何去选择视图,也很少提出一个通用的重构过程。而且这些技术仍然不能回答一些重要的问题,如驱动架构重构的问题是什么?需要恢复那一些视图?对于特定的视图需要那些技术才合适?这些视图要如何展示给用户才能更好的解决他们的问题?Symphony<sup>[11]</sup>就是在这个背景中提出的,是一种基于视图驱动的软件架构的重构的过程方法。和 Cacophony 一样,Symphony 也是将选定符合涉众需求的视点(Viewpoint)做为首先的工作。

#### 3.2.2.1 Symphony 中的架构视图

在 Symphony 中视图分为源视图(Source view)、目标视图(Target view)和假设视图(Hypothetical)。其中,源视图指的是可以直接从系统资料(如代码、文档等)中提取的视图如依赖关系,这种视图在 Symphony 中不属于严格意义上的架构视图。目标视图指的是重构过程执行后得到的描述已实现系统的架构并且包含解决问题的信息的视图。假设视图指的是描述系统的一个架构视图,但是也许并不完整,它能够做为一个参考架构视图,它可以是当前对系统的一个理解,可用于指导系统重构过程,一个典型的假设视图是软件系统开发之前的架构设计文档。



行源视图的提取、目标视图的映射和知识推论与信息解析，最后得到可用于满足涉众需求的结果。如图 3-4 该过程分为三个步骤：数据采集、知识推论和信息的解释。

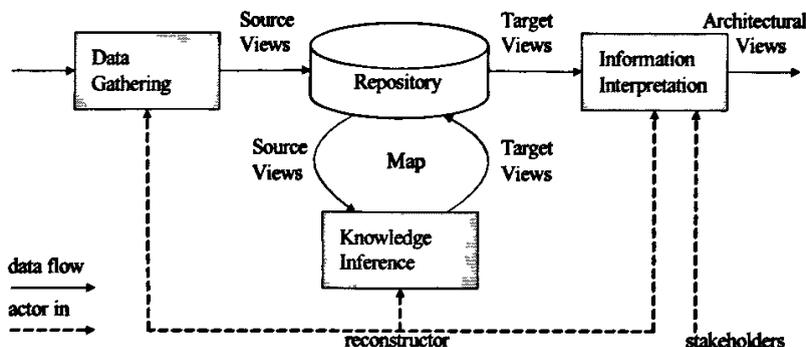


图 3-4 重构的执行过程图<sup>[11]</sup>

- 数据采集。目的是根据设计重构部分生成的架构的需求，从代码或者文档等收集架构恢复过程的数据信息。该过程产生了源视图。
- 知识推论。知识推论的目的是从源视图(通常是系统中大量关系的集合)中获取目标视图。如根据命名规则将类合并为模块以消除源视图中的多余关系信息。该过程需要源视图和领域知识相结合。
- 信息的解释。知识推论准备好了所有需要的信息，在信息解释活动中，将这些资源和信息通过某种适合于涉众解决问题的方式(如图形)展现出来。

### 3.3 架构重构方法和工具

目前已经提出很多架构重构的方法，相应的有架构重构工具去支撑着这些方法，以下为现有的一些架构重构方法和工具：

#### 1. Rigi

有近 10 年历史的逆向工程工具 Rigi<sup>[8]</sup>提供了非常友好的交互式界面，支持抽象机制，如聚合和泛化，这些功能允许用户去聚合几个节点、浏览某一节点、过滤掉边(节点间关系)和过滤节点，这样可以显示出系统的子系统结构，上述的动作利用 RCL(Rigi Command Language)可以编程实现。Rigi 的存储文件格式为 RSF(Rigi Standard Format)，Rigi 包含了 C、C++和 COBOL 的解析器，可以将提取到的信息以 RSF 格式保存，Rigi 是一个相当好的可视化工具，许多工具都使用 Rigi 作为可视化工具如下面介绍的 Riva 和 Dali。

## 2. Riva

重构工具 Riva<sup>[32]</sup>描述了一种将设计的架构影射到实现实体的方法，包括了六个步骤：

- 架构概念的定义，根据原设计架构文档和其他的信息。确定系统架构的组成部分。
- 源代码模型的提取，分析源代码并提取源代码模型。
- 抽象化，将源代码模型影射到第一步中确定的架构组成部分。
- 改进架构文档，系统架构文档化，改进架构文档。
- 分析提取的架构，产生改进架构计划。
- 代码的重新组织，根据改进架构计划进行系统的修改。

Riva 使用 Perl 脚本进行分析 C 源代码，提取的信息为 RSF 格式，最后使用 Rigi 作为可视化的工具。

## 3. Dali

SEI 开发的 Dali<sup>[33]</sup>认为“没有一个工具可以单独完成架构重构的整个过程”，所以 Dali 提供了一个轻量级的“工作平台(Workbench)”，易于与其他工具的集成。利用 Dali 进行架构重构包括四个步骤：

- 信息提取。从各种源提取信息，主要是源代码和系统运行跟踪，使用了各种工具包括：解析器(如 Imagix, SniFF+, CIA, rigiparse)、抽象语法树(AST)分析器(如 Gen++, Refine)、语法分析器(如 LSME)、剖析器(如 gprof)、代码插装工具、流行工具(例如 Grep, Perl)。
- 数据库构造。数据库构造包括将该信息转换为标准的形式。结合信息提取步骤该步骤实现了视图的提取，Dali 使用 PostgreSQL 关系数据库存储信息。通常提取的信息是 RSF 格式，需要通过 Perl 脚本转换成 SQL 命令。
- 视图融合。实现相应的架构重构活动，通过查询相应的提取信息构建特定的架构视图，查询工作基于 SQL。
- 重构。对视图进行操作，以揭示对构架概括性的、粗粒度的见解。Dali 使用 Rigi 工具作为元素和关系的层次上分解的图形提供给用户。

## 4. ARM

Guo 等<sup>[34]</sup>提出了“架构重构方法”ARM(Architecture Reconstruction Method), ARM 可以进行设计模式的识别,它使用 Dalil 作为工作平台进行架构恢复,ARM 包括四个步骤:

- 开发具体的模式识别计划,包括对模式集中的每一个模式进行形式化的描述,这些描述最终被转化为 SQL。
- 提取源模型,包括从源信息中提取结构化信息和聚合成高层的系统抽象。
- 检测并评估模式实例,在 Dali 平台之上进行模式识别,这是一个自动的过程(执行模式 SQL)。
- 重新构造并分析架构,利用 Rigi 等可视化视图,检查系统当前架构是否和设计架构相一致。

### 5. Shrimp

Shrimp<sup>[35]</sup>是一个信息可视化和信息导航系统,可以用于将系统提取出来的信息进行可视化。用于架构重构时,Shrimp 可以通过提供手动的聚合元素功能而给用户产生高层抽象的架构视图。

### 6. X-ray

X-ray<sup>[29]</sup>是一种对分布式应用程序的架构恢复方法,它可以提取组件和组件间运行时的关系。X-ray 通过结合了几种静态分析方法,从而避免了动态运行时分析的困难。X-ray 已经使用 Prolog 实现,从源代码提取出来的信息由 Prolog 事实(Facts)来表示,Prolog 谓词(Predicates)用于实现分析技术如聚类等。输出使用 Dotty 进行可视化。

## 3.4 架构重构方法的分类

本节从程序理解策略、自动化程度和结果输出三个方面对现有的架构重构方法和工具进行分类。

### 3.4.1 按程序理解策略分类

根据节 2.2.4 给出的程序理解策略,相应的架构重构方法可分为自底向上过程、自顶向下过程和混合过程。

#### 1. 自底向上过程。

从底层的知识去恢复架构，在大多数情况下由源代码模型开始，逐步地提高抽象层次以达到一个高层的易于理解的架构模型。该过程类似于 Tilley 等<sup>[23]</sup>描述的提取-抽象-展示的循环过程。如 Dali 为典型的自底向上过程：(1)从软件实现提取底层知识并存储到关系数据库；(2)使用 SQL 查询语句进行抽象操作；(3)用 Rigi 进行可视化，提供交互的界面。

## 2. 自顶向下过程。

由高层知识如需求和架构风格出发，通过形式化的概念假设并将这些假设和相应代码匹配来发现架构。纯粹的自顶向下过程的工具很少，一个典型的例子为 Murphy<sup>[36]</sup>提出的 Reflexion Model Tool(RMT)：(1)定义一个假设的系统高层概念视图；(2)确定该概念视图和代码视图的影射关系；(3) RMT 计算概念视图和代码视图的反射模型，该模型给出了概念模型和代码实现是否一致的信息。另外，SoftwareNaut<sup>[40]</sup>针对 java 程序的包结构，自顶向下进行展开和关闭包的操作，最终得到合适的视图。

## 3. 混合过程。

结合了自底向上过程和自顶向下过程，即一方面底层的知识使用不同的工具来抽象化，另一方面高层的知识和底层抽象出来的信息相比较。混合过程的一个例子为 Alborz<sup>[37]</sup>：分为三个步骤，第一，分析源代码提取信息并利用数据挖掘技术得到比代码高层的视图区域(Regions)；第二，工程师通过反复迭代得到架构的假设视图(以模式形式组合而成)；最后，将视图区域和模式相匹配。

### 3.4.2 按架构重构技术分类

按照架构重构技术的自动化程度可以将架构重构方法分为三类：准手动(Quasi-manual)；半自动化(Semi-automatic)；准自动化(Quasi-automatic)。

#### 1. 准手动(Quasi-manual)。

仅仅利用现有的简单的普通工具和领域知识对系统进行分析，使用特定的架构重构工具如 SHRiMP、Rigi 和 CodeCrawler 等进行辅助并生成可视化的结果，在这基础之上进行手工介入的架构重构活动。在论文<sup>[38]</sup>中将这些辅助技术分成了两类：

- 基于构建的技术，这些技术需要通过手工抽象化底层系统知识来重构软件

架构，相关工具包括提取底层知识工具和可视化工具：SHRiMP、Rigi、CodeCrawler<sup>[39]</sup>和 Verso 等。

- 基于浏览的技术，这些技术通过指导软件工程师从最顶层的系统实现信息出发而得到一个软件构架视图，如 SoftwareNaut<sup>[40]</sup>为工程师提供了包模式的识别，在重构过程中给出包该展开(Expend)或者关闭(Collapse)的提示。

## 2. 半自动化(Semi-automatic)。

工程师手动去引导工具如何自动的提取信息和恢复系统抽象结构。半自动化的架构重构方法帮助工程师创建一种可交互的架构视图，而通常这种可交互的架构视图的生成规则也是手工定义的。这里将半自动化方法分为两类：

- 基于查询方法，关系代数的方法允许架构重构人员去定义一种可重复的转换规则集合去获取一个特定的架构视图，如 Dali 使用 SQL 查询语言去定义聚合规则、Gupro 使用特定的查询语言 GReQL 等。
- 基于影射规则的方法，通常这种方法定义了一种在自动重构中遵循的规则，如 Rigi 使用 Tcl 脚本来定义图的转换规则，Reflexion Model<sup>[41]</sup>用规则去影射高层的实体(假设的)和源代码实体，ARM<sup>[34]</sup>定义了模式识别规则。

## 3. 准自动化(Quasi- automatic)。

准自动化(Quasi- automatic)，全自动化的架构重构方法是不存在的，准自动化架构重构工具控制整个架构重构过程，由架构重构人员进行控制重复的重构过程。一般准自动化过程使用数据挖掘技术如聚类方法和模式识别。

- 模式识别，通常使用形式概念分析(Formal Concept Analysis 简称 FCA)，Tilley 等<sup>[42]</sup>总结了形式概念分析在软件工程中的应用。
- 聚类方法，聚类方法将有某种相似关系的元素聚合起来，以识别系统的子系统、模块和组件等。

## 3.5 架构重构方法与工具评价

这里对目前的架构重构方法功能分析，以 3.1 中叙述的 O'Brien<sup>[30]</sup>等提出的架构重构主要问题为评价标准，如表 3-2，其中的符号说明如下：“×”不支持该问题场景；“√”支持该问题；“~”需要相应的修改以支持该问题；“u”为未知。

表 3-2 重构方法评估(改编自<sup>[30]</sup>)

	视图集合	强制性架构	质量属性影射	公共和可变的的部分	二进制组件	多语言支持
Manual	~	~	~	~	~	~
Rigi	~	×	×	×	×	√
Dali	~	~	~	×	×	√
Riva	~	~		×	×	√
ARM	~	~	~	×	×	√
Shrimp	~	×	×	×	×	√
X-ray	~	~	u	×	×	√

- 视图集合，目前没有方法或者工具支持显式的选择架构视图，这些视图应该能系统地重构以充分的描述系统和满足架构涉众的需求。这里假设现有的方法和工具经过修改都可以进行视图的选择。
- 强制性的架构，该问题目前很少工具支持，而且一般没有和正向工程工具整合在一起。
- 质量属性的变化，目前很少方法和工具显式的支持该问题。
- 公共和可变的的部分，目前没有工具支持该问题。
- 二进制组件，目前很少工具支持该问题。
- 混合语言，大部分工具都支持不同的程序语言，但是没有描述如何从混合语言的系统提取到的信息去构建架构视图。

### 3.6 本章小结

本章首先给出了架构重构实践过程中所遇到主要问题，然后详细分析了近年提出的两个比较全面和通用的过程模型 Cacophony 和 Symphony，介绍了现有的方法和工具并给出分类，最后对现有的架构重构工具进行了功能分析。通过评估发现目前软件架构重构领域仍然需要更进一步的工作，其中本文所关注的问题是强制性的架构。

## 第四章 架构重构工具 NEL 的关键技术

在节 3.1 介绍了 O'Brien 等<sup>[30]</sup>提出的架构重构的典型问题集并根据该问题集合对现有工具进行了评估，其中的强制性架构问题描述了系统实现与架构设计不一致的问题，在实践中发现引发该问题的原因主要有：

- 开发过程当中没有严格按照设计编写代码。对于目前大多数项目，保证架构模式的质量属性并不是放在第一位的。
- 开发人员对架构的理解程度不够。导致架构模式不恰当的实现。
- 缺乏有效的支持工具。不能很好的发现和定位违背设计的实现。
- 维护过程中很少考虑到原有架构的维持。多次的升级修改导致原有架构被破坏，同时文档也逐渐过期。

在这些问题背景下，我们设计并实现了架构重构工具——NEL。根据第三章中对架构重构的过程模型、方法和工具的分析，结合节 2.2.2 的逆向工程规范活动，本文总结了架构重构的技术框架，包括：架构视点、信息提取、信息推论和信息展示(如图 4-1)。本章将对该技术框架进行分析，在这基础之上，通过依赖结构矩阵和 HQL 查询相结合，给出了一种架构信息的识别方法，并阐述了 NEL 实现的关键技术。

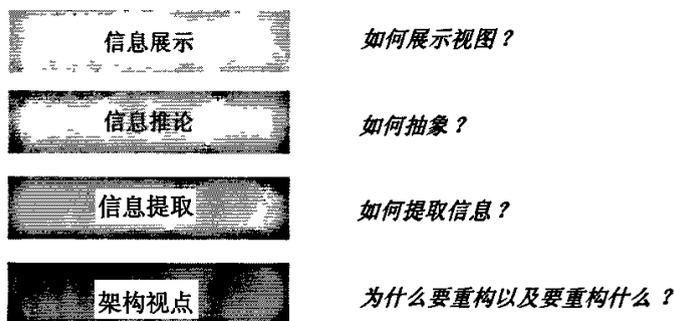


图 4-1 软件架构重构技术框架

### 4.1 架构视点

“软件架构的本质是什么取决于架构涉众团体对软件工程的理解；这些理解一部分以非实质的形式在公司里共享，另一部分以实质的资料形式通过支持

软件开发和演化的一系列开发工具表现出来<sup>[10]</sup>”。

软件架构重构由确定架构视点开始，目的是要明确“为什么要重构以及要重构什么？”，IEEE 1471<sup>[13]</sup>对声明一个视点做出了具体说明，每一个视点包括：视点名字；视点所需满足的涉众；视点需要覆盖的涉众的关注点；视点语言、建模技术以及所用的分析方法；说明视点的出处(如作者、著作或者文献引用)。

#### 4.1.1 如何获取目标视点

视点的选择至关重要，同时视点的选择又依赖于涉众以及涉众的关注点<sup>[19]</sup>，通常获取目标视点包括以下步骤<sup>[10][11]</sup>：

##### 1. 确定架构涉众列表。

即确定架构描述的概念模型 IEEE 1471 标准中的“Stakeholders”。常见的架构涉众有：客户、用户、操作员、架构师、系统工程师、开发人员、设计师、编译人员、维护人员、投资厂商、策划人员等等。

##### 2. 确定架构涉众关注问题。

即确定架构涉众的“Concern”。最终的重构结果为架构涉众解决问题所用。不同的涉众通常有不同的关注点，他们对系统有着一定需求，正是这些需求驱动了架构的重构。通常这些需求包括运行问题、高维护费用、低的可靠性和系统扩展等等。这些需求可以从以下方式获得：采访、工作会议、场景分析等。

##### 3. 领域资料分析。

涉众关注问题中隐含着他们对架构的理解，然而事实上由于当一般公司都没有很好的意识到架构的存在性，这些隐含的理解会导致模棱两可的架构概念<sup>[10]</sup>。幸运的是还有其他信息可以帮助我们更好的确定涉众对架构的概念，那就是该涉众团体所使用的开发工具和开发文档等实质的领域资料。

##### 4. 选择目标视图。

视点的来源可以是许多已经定义的视点，如在前面章节提到的：“4+1”视图(视点)、Soni 等提出的视图(视点)和 Len Bass 等提出的视图(视点)等等，也可以是为特定构架所新创建的视点。

不同的视图支持不同的目标和用途，“这就是我们基本上不采用某个特定的视图或视图集的原因<sup>[3]</sup>”。下面介绍 Len Bass 和 Clements 确定视图的方法<sup>[3]</sup>：

- 产生一个候选视图列表。涉众通常知道他们需要使用什么视图(视点), 或者至少有一些初步的想法。涉众列表, 建立一个涉众/视图表(表 5-2)。
- 组合视图。候选视图列表中可能会有很多视图, 为了减少视图数量以便管理, 接下来找出提供了多个视图所具有的信息的一个视图, 最后将它们组合单个视图。
- 划分优先级。得出一个适当的视图集合后需要去确定他们的优先级, 但是并不是完成一个视图后再开始另外一个视图, 可以从一个概要级开始, 因此宽度优先的方法通常是最佳的。

关键字: d=详细信息, s=一般细节, o=概要信息, x=有一点

涉众	模块视图				组件连接器视图	分配视图	
	分解	使用	类	层		各种	部署
项目经理	s	s	d	s		d	
开发人员	d	d	d	d	d	s	s
维护人员	d	d	s	d	d	s	s
分析人员	d	d		d	s	d	

表 4-1 涉众和视图列表例子<sup>[3]</sup>

#### 4.1.2 NEL 视点的确定

根据强制性架构的问题背景, NEL 的架构涉众为项目经理、开发小组成员、新的架构涉众、设计师、维护人员和分析人员。针对本章开始中给出的引发强制性架构问题的主要原因, 总结了 NEL 工具的需求: (a)如何理解当前系统的真实架构; (b)如何有效的管理系统各个模块(Java Package)之间的依赖关系; (c)系统在开发以及维护过程中如何发现以及定位违背设计的实现, 以做出相应的调整来维护架构。总的来说, NEL 的主要的目的就是为架构涉众提供一种实时更新的架构信息, 并且发现和定位问题的所在。

NEL 目前针对于 Java 项目, Java 项目主要的建模语言是 UML(Unified Modeling Language)<sup>[43]</sup>, NEL 使用 UML 包图和类图来描述系统组织结构。

根据以上描述的涉众关注问题, 确定 NEL 重构的目标视点为“4+1”视图的开发视图: 系统的开发架构由模块和子系统以及它们之间的导出(Export)和导入(Import)关系所表示。开发视图关注软件开发环境当中模块的组织结构, 软件被封装在一些小模块——程序库或者子系统, 这些模块可以由一个或者少数几

个开发人员进行开发。开发视图可作为需求的分配、团队工作分配（甚至开发团队的组织）、支出的预估和计划、监控项目的开发进程和合理化重用的基础。

## 4.2 信息提取

“信息提取的动机是真实架构隐藏于原始资料中<sup>[11]</sup>”，这些原始资料包括系统的源代码、编译文件、配置文件和测试用例等，其中源代码是最直接和主要的原始信息。信息提取的技术可以分为静态分析和动态分析。在架构重构过程中应该根据实际情况选择提取的信息技术以及工具。

### 4.2.1 静态分析

静态分析主要分析文本类型的原始资料，静态提取的相关方法有<sup>[11][44][45]</sup>：

- 手工分析(Manual Inspection)。在重构过程当中一些数据可以很容易利用手工获取，如程序包结构(目录结构)等等。
- 词法分析(Lexical Analysis)。词法分析的任务是对输入的字符串形式的源程序按顺序进行扫描，在扫描的同时，根据词法规则识别具有独立意义的单词(符号)，并产生与其等价的属性字符流作为输出。
- 语法分析(Syntactic Analysis)。按照语言既定的语法规则，对字符串形式的源程序进行预防检查，并识别出相应的语法成分。语法分析处理的依据是语言的文法规则，其分析结果是识别出的无语法错误的语法成分。
- 程序依赖图(Dependency Graph)，在不执行程序的情况下，收集程序数据的运行时信息，分析程序中数据对象之间的关系，程序依赖图用来表示程序中的模块、变量和函数之间依赖关系的视图。

除此之外，还可以使用剖析和分析设计模型、Build 文件、Makefiles 和可执行文件的工具根据需要提取更多的信息。例如，Build 文件和 Makefiles 包含系统中关于模块或文件的依赖性信息，这些信息可能在源代码中没有反映出来。目前，静态分析工具有很多，如 Clarkware 咨询公司的 JDepend、IBM 公司的 Sa4j、Compuware 公司的 OptimalJ、Rigipase、LSME(语法分析器)等。在架构重构实践当中，我们可以根据需求对提取工具做出合适的选择和更改。

## 4.2.2 动态分析

静态分析是在不执行目标系统的情况下对程序源代码进行分析,动态分析则是通过对目标系统的一次或多次运行进行分析<sup>[45]</sup>。由于后期绑定原因,一些信息只能在分析软件的动态行为时才能获得。动态分析技术主要有以下几类<sup>[44][45]</sup>:

- 植入技术。为了收集程序的运行时信息而修改当前程序,基本的程序植入技术是以不影响原有程序的语义为前提,在程序的关注位置插入代码。当植入后的新系统运行时,这些代码可以按照特定协议将动态信息传递到指定位置或转交给动态信息收集机制,从而提供调试信息、性能分析信息或对象之间的消息传递信息。程序植入是获取目标系统运行时信息最常用的方式。
- 部分求值 PE(Partial Evaluation),大型实时系统具有复杂的状态机体系结构,会引用大量全局变量和嵌套的条件语句。从整体上分析这类系统会很繁杂,而且效果不会太好,可以只针对系统的某些特殊行为进行分析。部分求值技术有助于进行该类大型系统的分析。PE 是一种程序转换,根据给定的不同运行参数选项,将大型系统分成较小的部分进行分析。PE 的基本过程分为两步:根据部分已知的输入数据,进行与其相关的计算,优化控制流,通过程序转换,将计算结果变换成程序代码,生成例化的程序;运行例化的滞留程序(residual program)完成其余计算。
- 动态切片,使用动态的数据流和控制流分析方法,程序语句间依赖关系是在以特定数据为输入的程序执行后确定的。一个动态切片可看作是删除原来程序的零个或多个语句得到的可执行程序,包括了“确实影响”一个值的所有语句,是静态切片的子集。动态切片的确定和程序执行历史相关,利用了程序特殊执行的信息,只考虑程序特殊执行中存在的依赖。

## 4.2.3 NEL 信息提取方法

NEL 针对的是 Java 系统,目前只支持对 Java Class 的静态分析。Java Class 文件具有特殊的结构。通过分析 Class 文件可以得到类之间的静态依赖关系。下面简单介绍 Class 文件的结构。

每个 Class 文件的前四个字节被称为它的魔数(Magic Number)。魔数的作用

在于,可以分辨出 Java Class 文件和非 Class 文件;接下来的四个字节是版本号;版本号之后是常量池(Constant Pool),常量池包含了与文件中类和接口相关的常量,它被组织为一个列表的形式;紧接着常量池后的两个字节被称为访问标志(Access Rights),表明了是类还是接口,是抽象的还是公共的等等;接下来是实现的接口(Implemented Interfaces),表示由该类直接实现或者有接口所扩展的父接口;再接下来是该类或接口所声明的字段(Fields)描述;然后是该类或接口所声明的方法(Methods)描述;最后是类的属性(Class Attributes)<sup>[46]</sup>。图 4-2 表示了“Hello World”程序的 Class 文件结构。关于更详细的 Class 文件结构请参阅 Java 虚拟机规范<sup>[47]</sup>。

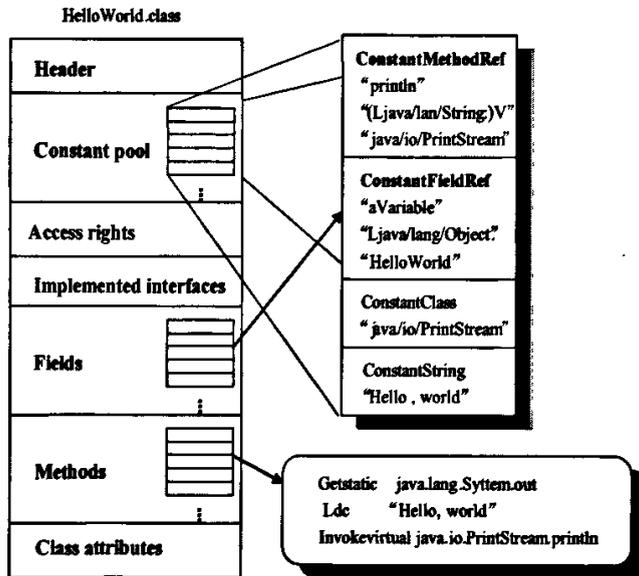


图 4-2 Java 字节码文件结构<sup>[46]</sup>

### 4.3 信息推论

对于使用信息提取技术提取出来的源信息,需要从中提取出相关的架构信息,也就是说如何抽象化提取出来的信息?这里讨论了几种方法:聚类方法、模式识别、关系查询以及设计结构矩(DSM)方法。

#### 4.3.1 聚类方法

聚类(Clustering)是根据对数据集中元素的相似性对数据集合的一个划分<sup>[48]</sup>,使得在每个划分部分中元素的相似性尽可能的大。在架构重构过程中,通

常利用聚类方法对提取的信息进行聚类处理来进行模块分解。本小节讨论聚类方法的定义、聚类任务以及架构重构过程中的聚类使用。

#### 4.3.1.1 聚类的定义

聚类的定义如下<sup>[48][49][50]</sup>：假设数据集合  $X$ ，其中  $x_i$  为数据点(或者可以成为对象、实例、模式、元组、实例和事务等)  $x_i = (x_{i1}, x_{i2}, \dots, x_{id}) \in A$ ，其中  $A$  为一个属性空间， $i \in [1, N]$ ，并且  $x_i$  的每个属性(或者称为特征、变量、维、组件、域等)  $x_{il} \in A$ ， $l \in [1, 2, \dots, d]$  既可以是数值型，也可以是枚举型。数据集合  $X$  相当于是一个  $N \times d$  矩阵。假设数据集  $X$  中有  $N$  个对象  $x_i (i = 1, \dots, N)$ 。聚类的最终目的是把数据集合  $X$  划分为  $k$  个分割  $C_m (m = 1, \dots, k)$ ，可能有些对象不属于任何一个分割，这些就是噪声  $C_n$ 。所有这些分割与噪声的并集就是数据集合  $X$ ，并且这些分割之间没有交集，即

$$\begin{cases} X = C_1 \cup \dots \cup C_k \cup C_n \\ C_i \cap C_j = \emptyset (i \neq j) \end{cases}$$

这些分割  $C_m$  称为聚类。

#### 4.3.1.2 聚类的步骤

聚类技术是一种无监督的机器学习过程，它的主要目的是把没有“标记”数据分为有意义的“组”（或者就叫聚类）。一般来说，聚类技术可以分为以下几个任务阶段<sup>[49]</sup>：

- 数据表示和特征选择。决定用什么模式来表示数据。这一阶段还包括特征选择和特征抽取。特征选择是指在所有的数据属性的集合中选择一个子集来代表数据。特征抽取则是由现有的数据属性产生新的属性。
- 相似度定义。定义如何表示数据的相识度。一般使用的是基于距离或基于相似度的表示方法。
- 聚类技术。通过多种聚类算法得到聚类，通常算法分为两类，层次分类算法和划分分类算法。
- 可选的任务阶段。聚类结果的数据抽象和评估聚类等。

### 4.3.1.3 聚类方法在架构重构中的应用

如图 4-3 是使用 NEL 对一个包含 437 个类和 1269 个关系的 Java 系统分析得到的提取结果，该图以框图形式描述了系统中所有的类以及他们之间的关系，可以看出图中表示出来的信息是难以理解的。

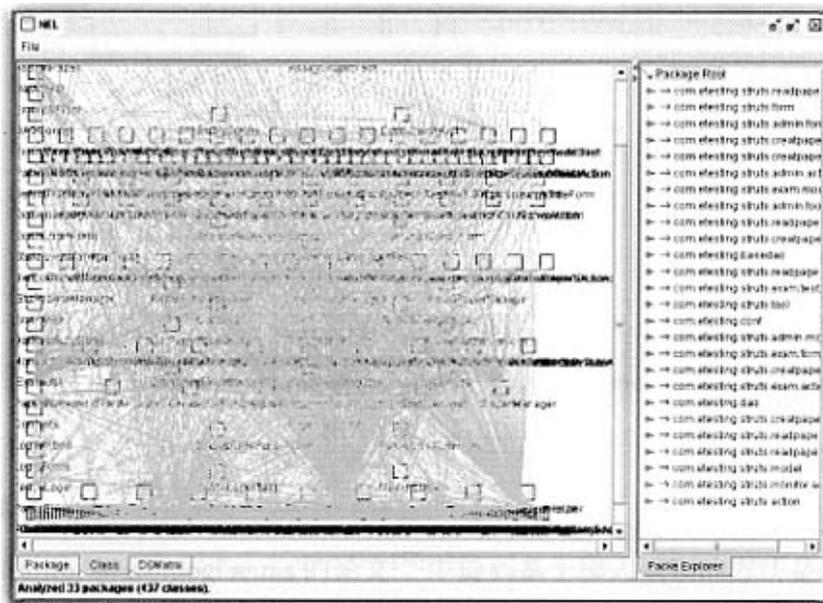


图 4-3 原始类图

聚类方法通常从提取源信息(如图 4-3 的类节点和类之间的依赖关系)开始，然后定义相似度，相似度是判断元素是否属于某一类的条件，根据相似度的定义使用聚类算法将元素分组到各个子系统(类簇)。重复该过程直到得到一个理想的系统分解。如图 4-4 为以 Java 包为相似度进行聚类后得到的包结构图。

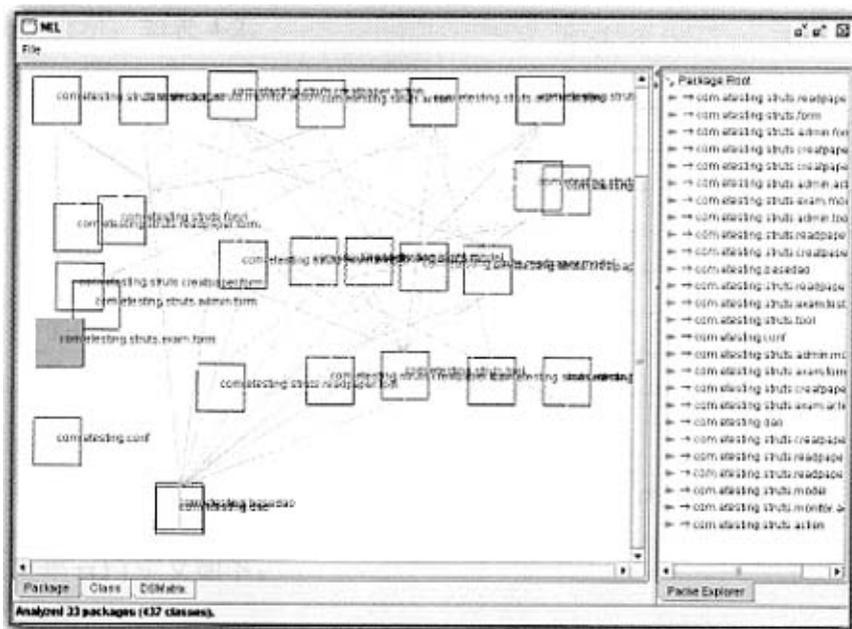


图 4-4 聚类后包图

### 4.3.2 模式识别

在 Sartipi 和 Kontogiannis 的论文<sup>[52]</sup>中提到基于模式识别的方法是软件架构重构的主要自动化方法之一。基于模式识别的架构重构技术发现实现中的公共结构和模式，通过用抽象的形式替换这些结构和模式以简化架构视图，通过展示这些抽象形式，在架构重构过程中，工程师可以很快的发现并理解这些结构和模式，减少了系统理解的时间。在架构重构领域模式识别使用的方法很多，下面给出了一种常用的数学方法——形式概念分析(Formal Concept Analysis 简称 FCA)。

#### 4.3.2.1 FCA 介绍

在过去的 10 年中，由 R. Wille 提出的 FCA 的应用领域发生了巨大的变化，其主要应用领域逐步从数学转向计算机科学<sup>[53]</sup>。FCA 被看作是概念化知识处理 (CKP, Conceptual Knowledge Processing) 数学基础。FCA 在软件工程领域应用相当的广泛<sup>[42]</sup>。

形式分析由形式上下文开始(Context)，上下文是一个三元组  $C = (O, A, R)$ ， $O$  为对象的有限集合而  $A$  为属性的有限集合， $R$  为  $O$  和  $A$  的二元关系，即有  $R \subseteq O \times A$ 。如果有  $(o, a) \in R$  那么称对象  $o$  具有属性  $a$ 。二元关系可以用一个二

维的表 T 表示，如表 4-2。

		属性				
		Fastest	Water	Running	Ball	Team
对象	Swimming	√	√			
	Soccer			√	√	√
	Waterpolo		√		√	√
	Icehockey		√			√
	Triathlon	√	√	√		
	Bobsledding	√	√	√		√

表 4-2 运动属性<sup>[29]</sup>

定义  $X \subseteq O, Y \subseteq A$ ，则定义  $X$  中对象的公共属性集合为  $\sigma(X)$  和具有共同属性  $Y$  的对象之集  $\tau(Y)$  定义如下：

$$\sigma(X) = \{a \in A \mid \forall o \in X : (o, a) \in R\}$$

$$\tau(Y) = \{o \in O \mid \forall a \in Y : (o, a) \in R\}$$

在上下文  $(O, A, R)$  中的概念  $C$  是一个二元组  $(X, Y)$ ，其中  $X \subseteq O, Y \subseteq A$ ，满足条件  $Y = \sigma(X) \wedge X = \tau(Y)$ ，称  $X$  为概念外延，记为  $ext(C)$ ，称  $Y$  为概念内涵，记为  $int(C)$ 。

从定义上可以知道，概念描述的是具有相同属性的对象构成的最大的集合，在运动例子中  $(\{soccer, waterpolo\}, \{ball, team\})$  就是一个概念，而  $(\{triathlon, bobsledding\}, \{fastest, water\})$  并不是一个概念，因为  $swimming$  也有这些属性，而  $(\{soccer, waterpolo\}, \{running, ball, team\})$  也不是概念因为  $waterpolo$  并没有  $running$  的属性。该例子的概念用表格形式表述如表 4-3。

Ct	$(\{swimming, soccer, waterpolo, icehockey, triathlon, bobsledding\}, \emptyset)$
C0	$(\{swimming, triathlon, bobsledding\}, \{fastest, water\})$
C1	$(\{soccer\}, \{running, ball, team\})$
C2	$(\{waterpolo\}, \{water, ball, team\})$
C3	$(\{icehockey, waterpolo, bobsledding\}, \{water, team\})$
C4	$(\{triathlon, bobsledding\}, \{fastest, water, running\})$

C5	((bobsledding), {fastest, water, running, team})
C6	((swimming, triathlon, bobsledding, icehockey, waterpolo), {water})
C7	((soccer, waterpolo), {ball, team})
C8	((soccer, waterpolo, icehockey, bobsledding), {team})
C9	((soccer, triathlon, bobsledding), {running})
C10	((soccer, bobsledding), {running, team})
Cb	( $\emptyset$ , {fastest, water, running, ball, team})

表 4-3 概念<sup>[29]</sup>

#### 4.3.2.2 FCA 在架构重构中的应用

在文章<sup>[53]</sup>中描述了如何利用 FCA 在面向对象的代码进行重复设计结构的识别。主要的思想是一个设计模式包括一个类集合和一个关系集合，不同的模式实例拥有相同的关系集合而用于不同的类集合。

设  $D$  为系统类集合,  $T$  为类集合中的关系类型集合, 如  $T = \{e, a\}$  表示”extend”和”association”, 类中关系集合  $P \subseteq D \times D \times T$ 。为了识别  $k$  个类组成的设计模式, 定义上下文  $C_k = (O_k, A_k, R_k)$ :

- $O_k$ , 大小为  $k$  的类序列的集合,  $O_k = \{(x_1, x_2, \dots, x_k) \mid x_i \in D \wedge i \in [1..k]\}$ 。
- $A_k$ , 在  $O_k$  中类关系的集合, 为二元组  $(x_i, x_j)_t$ , 其中  $x_i$  和  $x_j$  为类而  $t$  为关系类型  $A_k$  定义  $A_k = \{(i, j)_t \mid (x_i, x_j)_t \in P \wedge i, j \in [1..k]\}$ 。
- $R_k$ , 为  $O_k$  中元素所拥有的  $A_k$  中的属性。

根据以上的定义下面来看一个  $A_3$  的例子, 如图 4-5:

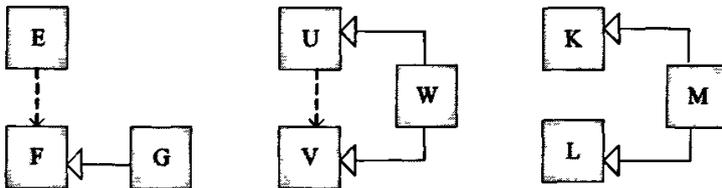


图 4-5  $A_3$  类图<sup>[29]</sup>

由以上定义可以得到关系集合  $P$  为：  
 $(E,F)_a, (G,F)_e, (U,V)_a, (W,U)_e, (W,V)_e, (M,K)_e, (M,L)_e$ 。

在该上下文中概念  $(X,Y)$  包括了一个类学列集合  $X$  和一个关系集合  $Y$ 。所以概念内涵  $Y$  表示了模式而概念外延  $X$  为模式的集合。 $A_3$  的概念表格如表 4-4(这里的图表仅仅是一个说明例子)。

		属性 $A_3$						
		$(1,2)_a$	$(1,2)_e$	$(3,2)_e$	$(3,2)_a$	$(3,1)_e$	$(2,3)_a$	$(1,3)_e$
对象 $O_3$	$(E,F,G)$	√		√				
	$(G,F,E)$		√		√			
	$(U,V,W)$	√		√		√		
	$(W,U,V)$		√				√	√
	$(W,V,U)$		√		√			√
	$(M,K,L)$		√					√
	$(M,L,K)$		√					√

表 4-4  $A_3$  概念表(例子)

### 4.3.3 设计结构矩阵

设计结构矩阵，简称 DSM(Design Structure Matrix)是表示元素(如软件组件或者设计任务)及其关系的一种方法，在产品开发过程领域得到广泛的使用。DSM 是了解设计复杂性的一个强有力的表示模型，允许被分析的设计元素之间存在依赖关系。

#### 4.3.3.1 DSM 矩阵概述

Donald Steward<sup>[54]</sup>在 1981 年引入设计结构矩阵来分析信息流，DSM 是一个  $n$  阶方阵，用于显示矩阵中的各个元素的交互关系，有利于对复杂项目进行可视化分析。如图 4-6 所示，设计结构矩阵是一个  $n \times n$  的方阵，系统的元素均以相同的顺序放在矩阵的最左边和最右边，如果元素  $i$  和元素  $j$  之间存在联系，则矩阵中  $ij$  元素为“Y”，

	1	2	3	4
1	O		Y	
2	Y	O		
3	Y		O	
4			Y	O

图4-6 DSM矩阵和框图

否则为空，对角线上的元素一般不用于描述系统，用“O”表示。这种矩阵也称为依赖关系矩阵(Dependency Structrue Martrix)<sup>[18]</sup>。

#### 4.3.3.2 DSM 矩阵的划分

利用划分(Partitioning)算法，通过聚合环中任务得到一个新的任务集合和执行次序。对 DSM 矩阵划分的方法有很多种，它们的区别在于如何识别循环任务。识别算法的过程如下<sup>[55]</sup>：

- Step1. 在 DSM 矩阵中识别空行的任务，把这些元素放在 DSM 矩阵顶部，一旦一个任务被重新排列，把它从 DSM 矩阵中移出，对所有任务重复进行该步骤。
- Step2. 在 DSM 矩阵中识别空列的任务，把这些元素放在 DSM 矩阵底部，一旦一个任务被重新排列，把它从 DSM 矩阵中移出，对所有任务重复进行该步骤。
- Step3. 如果 Step1 和 Step2 执行后在 DSM 矩阵中无剩余的任务，则此矩阵被完全分割。若有保留的任务则至少含有一个循环信息。
- Step4. 用某种方法确定环路。
- Step5. 把涉及一个环中的活动用一个聚合的任务来表示，继续 Step1。

如图 4-7 为 DSM 分解示意图：

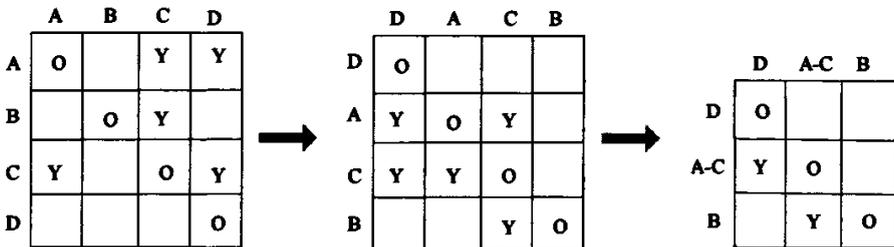


图 4-7 DSM 分解过程

#### 4.3.3.3 DSM 矩阵的分层

在论文<sup>[55]</sup>中对矩阵的分层的描述：DSM 矩阵层次划分的目的是将其中的各元素分为不同的级，假设将各项任务看成一个网络中的节点，若某些节点不能由任何其他节点到达时，这些节点被划入某一层中，即源节点。如果将这些节点从系统中删除则又可进行新的层次划分，由此可得到新源节点，这种划

分称为正向层次划分。

#### 4.3.4 关系查询

从软件架构的定义可以知道，软件架构由元素以及他们之间的关系组成。为了抽象出有价值的架构信息，在架构重构过程中需要分析大量的数据，这些数据有以下特点<sup>[56]</sup>：

- 数据量大。对于大型软件系统来说，程序源代码本身就是十分大型的数据，而在其基础上能够解析出大量的关系信息，在程序理解中动态生成的各种临时数据更是随着对程序理解的深入而无法预计其数量。
- 数据关系复杂。各种元素之间存在着复杂的关系，例如，类之间的依赖关系和包之间的依赖关系等。
- 数据在人机交互中被频繁操作。由 3.4.2 的重构方法分类可以知道，架构重构并不能达到完全自动化的程度，所以它的人机交互是十分重要的，工程师需要不断地细化完善解析出有价值的架构信息，因此数据被频繁地存储、增加、浏览、修改和删除。

基于以上的原因，目前一些架构重构方法使用数据库来进行数据组织。如架构重构工具 Dali<sup>[33]</sup>使用 PostgreSQL 关系数据库存储提取的信息，使用 SQL 查询，根据工程师期望在系统中发现的架构模式，可以建立各种查询脚本，从而可以得到各种抽象视图(图 4-8)。

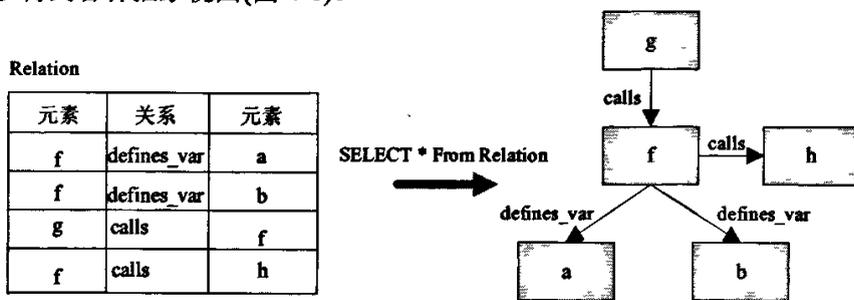


图 4-8 SQL 查询示例

#### 4.3.5 NEL 的信息推论方法

NEL 采取了设计结构矩阵架构信息分析方法，在矩阵分析方法的基础之上结合了包模式的识别，对于从源代码提取出来的原始信息构建关系数据库，利用关系查询可以定义 HQL 查询脚本以得到特定的元素和关系的集合。

#### 4.3.5.1 依赖结构矩阵

根据 4.1.2 中 NEL 视点定义为“4+1”的开发视点, NEL 使用 DSM 矩阵进行架构信息的分析, 在 NEL 中 DSM 的任务角色由系统模块(包)担当, 任务的依赖性为模块之间的依赖性, 目前我们仅仅针对 Java 系统。DSM 非常适合于表示系统模块的导出(Export)、导入(Import)关系和层次关系:

1. 与有向图表示相比, DSM 与图形表示相同的信息(如图 4-9), 但是它对整个系统元素提供整体的更为紧凑和机器更可读的形式, 即使元素数目很多也能很好的表示。有向图是一种很方便的图, 前提是元素的数目未超过浏览和直观的分析它的能力。

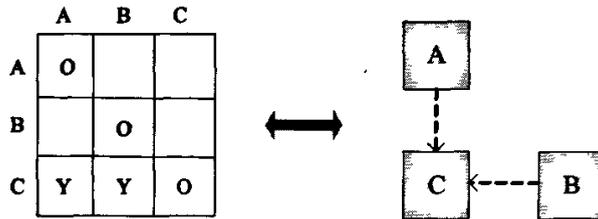


图 4-9 DSM 矩阵和框图

2. 产品开发工作流的划分目的和软件系统结构的划分目的十分类似。DSM 领域的一个很关注的去掉环依赖问题, 在软件系统当中也是存在的<sup>[57]</sup>。DSM 中的任务“分层”和软件系统的“分层”有着很大的一致性(如图 4-10)。使用 DSM 划分方法可以识别出软件系统的分层结构并发现环依赖。

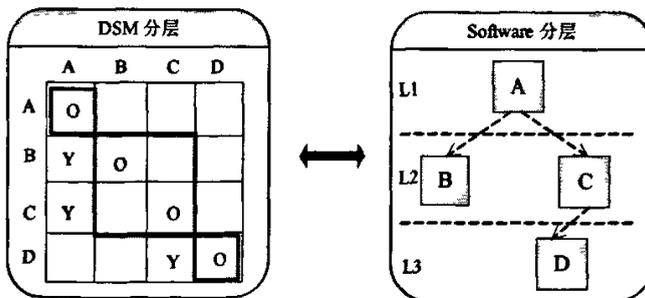


图 4-10 DSM “分层”和 Software “分层”

3. DSM 的划分算法为架构重构提供了一种自动的划分机制, 划分算法通过将环中模块聚合成子系统来去除环依赖, 执行划分算法后得出的子系统与元素顺序可以直接用于系统分层的识别, 在这个基础之上可以根据领域知识对矩阵进

行调整而得到满意的架构开发视图，划分处理后的 DSM 矩阵为架构重构过程提供了一个好的起点。

4. DSM 矩阵更易于发现一些重要的模式，如图 4-11。

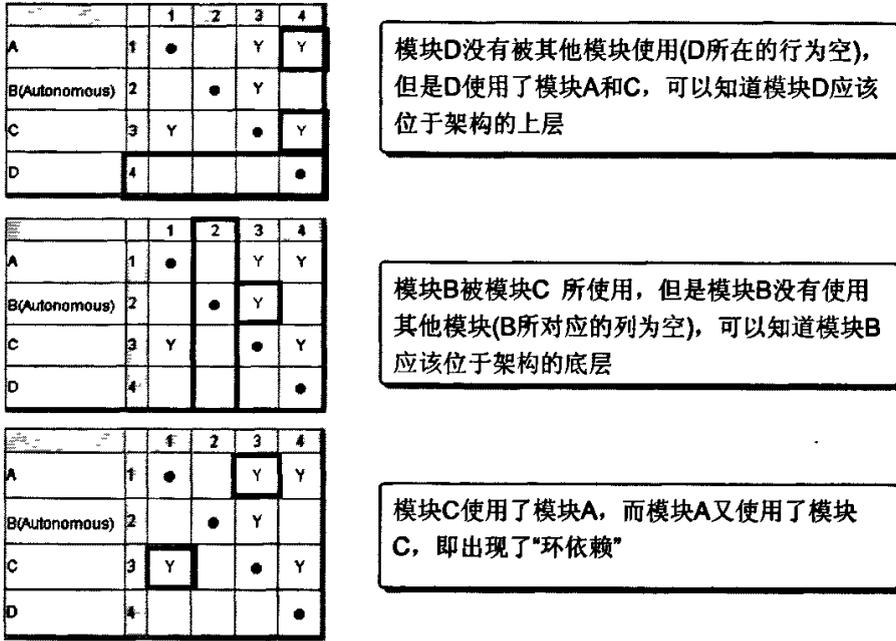


图 4-11 DSM 矩阵表示的各种有用的模式

另一方面，DSM 矩阵也有问题和不足之处：

- 当元素的数量达到一定的规模时，矩阵是难以阅读并理解的，所以通常矩阵的元素是系统较为高层元素。矩阵并不适合表示系统较低层的元素。
- 难以查看系统局部的元素及其之间的依赖关系。如在矩阵中，当发现存在违背设计原则的依赖关系时，无法确切的定位产生该依赖关系的低层元素。
- 不能根据定义的架构模式，灵活的生成特定的视图。
- 通过对系统包的展开和关闭操作，可以得到表示不同层次的 DSM 矩阵然而如何获取合适包层次？

为了弥补矩阵的不足，在 NEL 中，结合使用 HQL 查询，查询结果以 UML 静态结构图表示，以得到更有效的架构重构方法。

### 4.3.5.2 包模式的识别

Lungu<sup>[40]</sup>针对 Java 应用程序提出了一种利用 Java 包模式识别的自顶向下的架构恢复方法, 该方法的支持的工具为 SoftwareNaut。如图 4-12: 由系统顶层包为起点; 对当前的包或关系(边)进行展开、关闭和过滤操作直到得到合适的包视图;

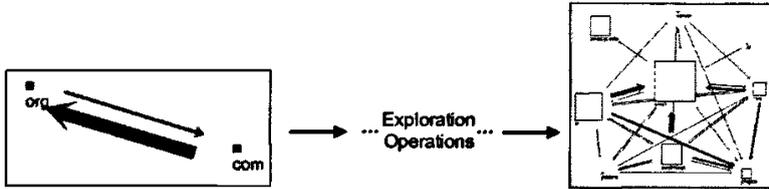


图 4-12 包模式的架构恢复方法<sup>[40]</sup>

Lungu<sup>[40]</sup>给出了一些包模式以及他们的识别方法, 并给出操作的建议, 下面给出其中常见的几种包模式:

- **Iceberg(冰山)**, 如果一个包为外部包所引用, 而其子包没有向外部包提供任何引用(可能引用外部包), 即对于外部模块来说, 其子包为隐藏的, 仅该包是可见的, 这种包称之为冰山。这种包建议不要展开。
- **Autonomous(自治)**, 一个自治的包自身和其子包对外部模块(包)没有任何引用, 仅仅对外部包提供了功能。由于对外部包没有依赖, 该包的功能性是独立的, 所以这种包建议不要展开。
- **Fall-Through(瀑布)**, 瀑布包仅仅拥有一个直接的子包, 并且该包和其他外部包没有直接的依赖关系。如 org、com 等, 建议展开。

在这里我们将 Lungu 的包模式识别方法应用到 DSM 中, 自顶向下, 根据模式识别以及操作建议, 由系统顶层包开始, 逐步进行展开、关闭和删除矩阵元素, 以得到合适的依赖结构矩阵。如图 4-13 所示, 可以看出图中识别出了 Fall-Through 和 Autonomous 两种模式的包, 通过展开和关闭操作得到合适的矩阵, 然后划分矩阵得到分析结果。

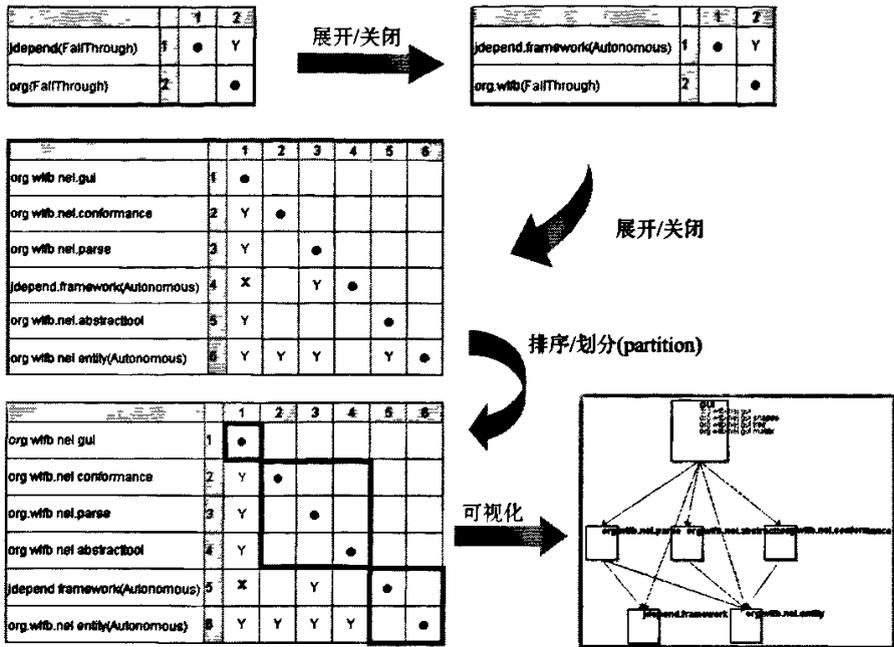


图 4-13 DSM 和包模式识别的结合

#### 4.3.5.2 HQL 查询

关系查询为信息推论方法之一，与现有工具如 Dali 不同的是，NEL 使用 Hibernate 对数据库进行操作，Hibernate 为 ORM(Object Relation Mapping)技术的一种实现。NEL 使用面向对象的查询语言 HQL(Hibernate Query Language)，图 4-14 为 HQL 完整的语法结构。

```
[select/update/delete ...] [ from ...] [ where ...] [ group by ... [ having]] [ order by ...]
```

图 4-14 HQL 语法结构<sup>[58]</sup>

可以看出 HQL 的语法和 SQL 非常相似。通过执行自定义的 HQL 脚本我们可以得到特定的元素以及关系，从而得到特定的模型视图。

HQL 基于 SQL，同时提供了完全面向对象的封装，HQL 实现了面向对象的概念：继承、多态和关联。具体的 Hibernate 描述和 HQL 语法可以参考 Hibernate 参考文档<sup>[59]</sup>。

## 4.4 信息展示

“帮助软件工程师去解决系统复杂性和提高程序员效率的一个办法是通过软件可视化，其实质是可视化表示可以使软件系统理解更容易<sup>[60]</sup>。”

### 4.4.1 软件可视化

软件可视化是作为软件工程的一个分支发展起来的，其目的是管理复杂的程序，以减少维护费用。软件可视化考虑的主要问题是软件的界面、外观和表示，而软件逆向工程的目的是程序理解，处理减少理解软件的复杂性，因此软件可视化也自然成为软件逆向工程的一个重要手段<sup>[61]</sup>。

在软件架构重构中，架构信息的可视化对架构的理解也是非常重要的，再者由于软件架构重构的非全自动化的性质，要求提取的信息能以一种可以交互的方式去展示给架构涉众。目前比较出色的可视化工具如 Rigi、Softwareaut 等。在 IEEE 的可视化会议上(IEEE Visualization Conference) Danny Holten 在文章<sup>[62][63]</sup>中论述了如何更好的可视化架构。

### 4.4.2 NEL 信息展示方式

NEL 采用了 UML(包图与类图)和设计结构矩阵(DSM)作为分析与表示系统架构的视图，我们发现 DSM 矩阵非常适合于表达依赖性和系统层次关系：当系统规模变大时，系统模型包括许多的节点(如包、类)和他们之间的依赖关系，此时方框和箭头图(如 UML 包图和类图)难以直观的分析，而在大规模的系统条件下 DSM 矩阵显示出很好的表示能力<sup>[9][18]</sup>。同时 NEL 也对框图提供了一些基本操作：删除节点/边、聚合几个节点等等，通过这些操作，工程师可以进一步构建有意义的视图。

## 4.5 本章小结

本章分析了架构重构过程中的关键技术，包括架构视点技术、信息提取技术、信息推论技术以及软件可视化技术，在这基础之上叙述了我们设计开发的架构重构系统 NEL 所应用的技术，为下一章的系统实现提供了技术方面的支持。

## 第五章 架构重构工具 NEL 的设计与实现

按照第四章中叙述的软件架构重构的技术框架，我们设计并实现了架构重构工具——NEL。该工具的开发过程，从分析、设计到实现和测试，全面运用了面向对象的方法，其开发语言为 Java，开发环境为 Eclipse。这一章将会对 NEL 系统的整个开发过程进行详细的介绍。

### 5.1 系统需求

#### 5.1.1 非功能性需求

对于 NEL 的非功能性需求主要包括两个方面：

- 可扩充性和维护性：对于不同的环境，架构重构的视点的不同，对工具的要求也不尽相同，此时需要工具易于增加或者修改当前的功能，如增加对源代码的关系的提取。
- 应用适应性：没有一个工具可以单独完成架构重构的整个过程，架构重构的特殊性使得可以兼容其他工具显得非常的重要。

#### 5.1.2 功能性需求

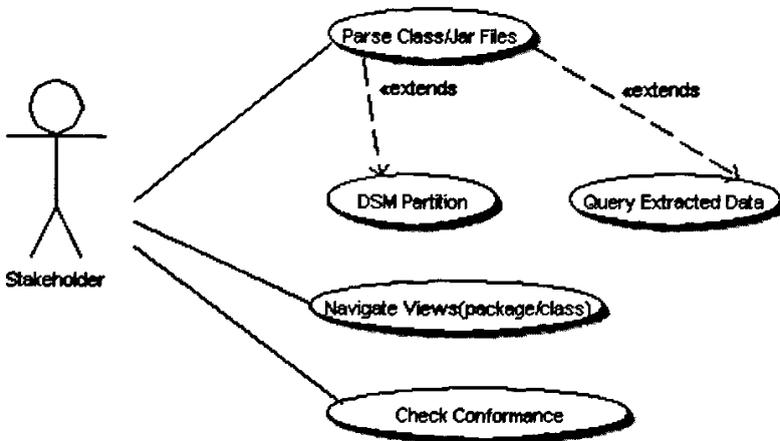


图 5-1 NEL 系统用例图

NEL 工具的主要功能包括分析 Java 系统并提取数据、信息抽象(包括 DSM 划分、HQL 查询)和浏览视图三个部分，如图 5-1，下面对图中 5 个用例进行简

单的描述:

- **分析 Class/Jar 文件(Parse Class/Jar Files)**。分析和提取目标系统信息并将分析结果保存于数据库。该用例的基本路径: a)工程师选择目标系统的二进制代码(Class 或 Jar)路径, 点击分析按钮; b)NEL 分析目标文件提取信息并存储数据库。
- **DSM 划分(DSM Partition)**。这里 DSM 的任务为 Java 包, 他们的依赖关系为包之间的“import”关系, DSM 划分为 NEL 信息抽象方式之一, 目的是利用 DSM 划分对系统进行模块分解和分层。该用例的基本路径: a)工程师选择 DSM 面板, 此时的 DSM 矩阵已划分处理; b)工程师根据需要置换矩阵行列重新排列矩阵; c)通过相应的查询生成当前 DSM 对应的包视图。
- **查询所提取的信息(Query Extracted Data)**。抽象方式之一, 利用 HQL 查询语言可以对分析系统用例中存储的数据进行不同的查询, 这些查询可展示较低层元素的各种抽象或集群的新聚合。该用例的基本路径: a)工程师输入 HQL 执行脚本, 点击查询; b)系统通过框图展示查询结果。
- **浏览视图(Navigate Views)**。在划分和查询生成视图的基础上, 工程师可以对结果视图进行浏览和操作, 包括删除节点、查看节点详细信息等等。该用例的基本路径: a)工程师对当前视图进行操作; b)系统识别操作并对视图进行相关更新。
- **检查一致性(Check Conformance)**。该用例为设计和实现的一致性检查, 其结果展示于 DSM 矩阵之中。该用例的基本路径: a)工程师输入架构设计规则信息, 点击加载; b)系统在 DSM 矩阵中使用特殊的符号展示出违反设计规则的依赖关系; c)使用 HQL 查询定位问题依赖。

## 5.2 系统分析与设计

### 5.2.1 系统协作图

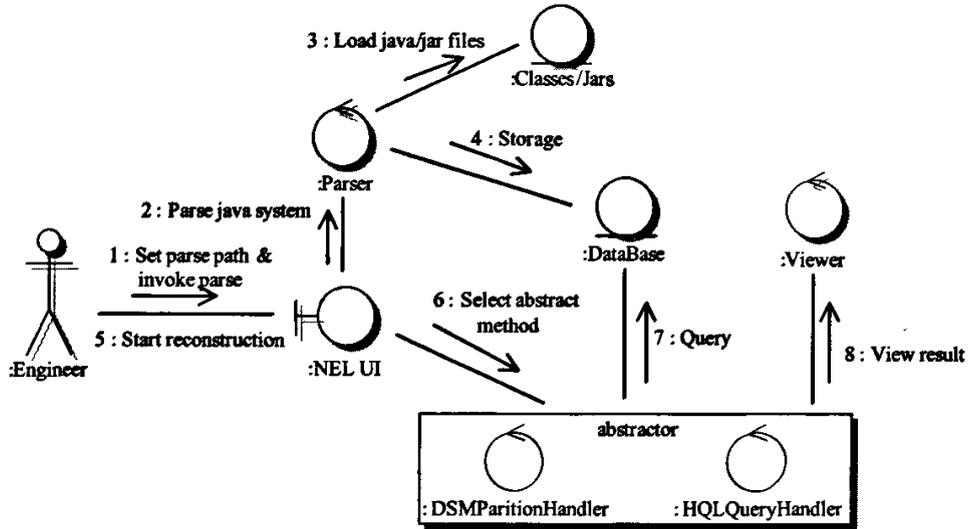


图 5-2 NEL 系统协作图

根据需求中的用例，使用协作图对这些用例进行了分析(图 5-2)：a)工程师通过 NEL 的用户界面(NEL UI)设置目标系统路径并触发分析活动(1, 2), NEL UI 调用解析器(Parser)加载目标文件(Class/Jar)并分析，最后将分析结果存储数据库(3,4)，此时系统以构建好提取数据库(DataBase)；b)工程师开始架构重构过程，NEL 提供了 DSM 划分和 HQL 查询两种信息推论方法，对于 DSM 方法，通过对 Java 包的展开和关闭可以得到不同的 DSM 矩阵，在 DSM 矩阵的基础之上根据需要进行重新排序(通过划分算法或者手工排序)，最后调用 Viewer 用图的方式表现出来(5,6,7,8)。对于 HQL 方法，通过执行不同的 HQL 查询语言，可以得到不同的关系集合，最后调用 Viewer 表示出来(6,7,8)。

### 5.2.2 系统包图

根据协作图(图 5-2)分析可以得到以下包图(图 5-3)，其中 **gui** 为系统界面，**parser** 提供提取 Class/Jar 文件的类，**conformance** 为架构设计与实现的一致性检查，**abstract** 包含两种抽象方法，**Entity** 为实体类，**jdepend.framework** 为 Clarkware<sup>[64]</sup>咨询公司的提供的分析 Class 文件的开源包。

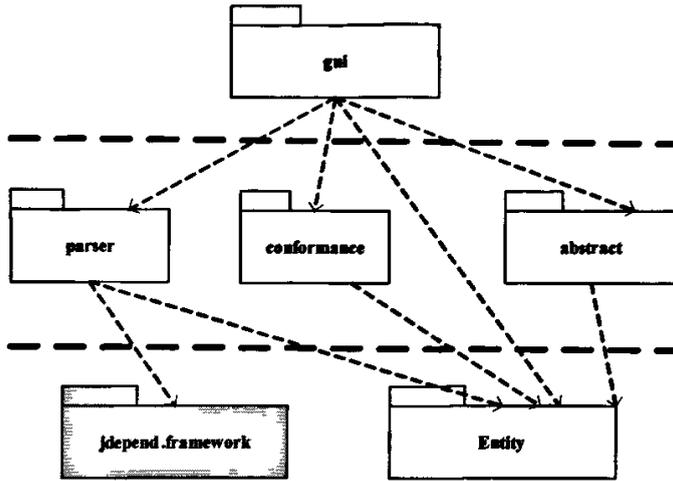


图 5-3NEL 系统包图

### 5.2.3 系统静态类图

#### 1. 数据库模型(entity 包).

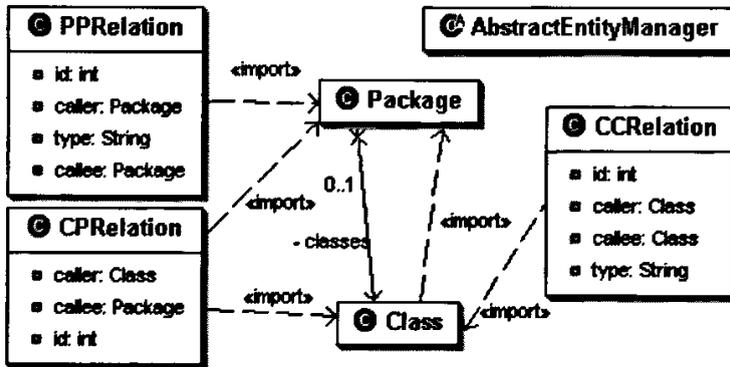


图 5-4 数据库模型(Entity 包静态类图)

Entity 包中为实体类集合，如图 5-4 为简化了的实体类图，其中 Package 类代表 Java 包(Java Package); PPRelation 代表包之间的关系，其属性 caller 和 callee 分别代表引用者与被引用者，属性 type 为关系类型，0 表示 caller 引用 callee，1 表示 caller 并没有显式的引用 callee 但是其子包中包含着对 callee 的引用; Class 代表 Java 类(Java Class); CRelation 代表类之间的关系，属性 caller 和 callee 分别代表引用者与被引用者，属性 type 为关系类型，0 表示“association”，1 表示“extends”，2 表示“implements”；CRelation 代表了类“import”包的关系; AbstractEntiyManager 为实体访问抽象类，封装了数据库访问细节，它为访问数据库提供了基础的支持。

2. 主要功能类图(Abstract、Conformnce、Parse 包)。

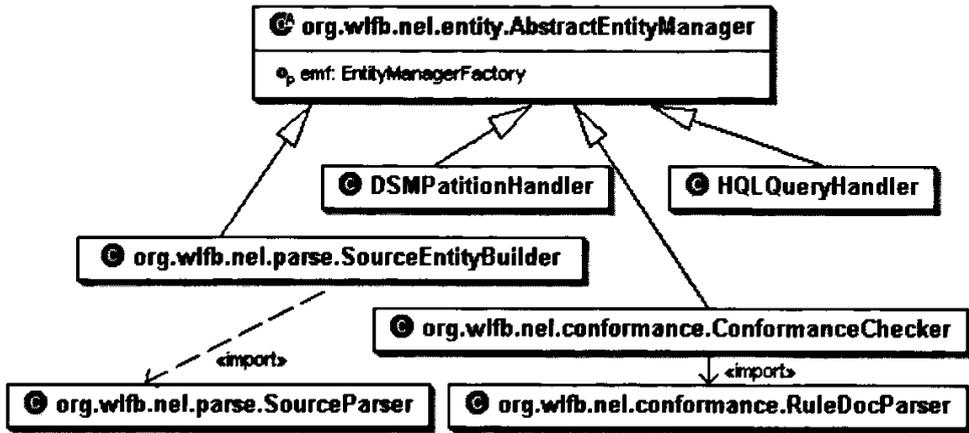


图 5-5 抽象方法主要类图(Abstract 包)

Abstract 包中提供了抽象方法的实现，DSMPatitionHandler 为 DSM 划分算法的实现，HQLQueryHandler 执行工程师查询的 HQL 脚本和一些命令并返回查询结果。他们都继承抽象类 AbstractEntiyManager。

Conformnce 包中包含了两个类 ConformanceChecker 和 RuleDocParse，其中 RuleParser 对工程师输入的架构规则进行分析，在 NEL 中架构规则为三元组{引用包，can not import，被引用包}，如 {org.company.p1，can not import，org.company.p2}表示包 p1 不能引用包 p2。ConformanceChecker 类的作用是检查数据库中违反架构规则的关系。

Parse 包主要功能为提取 Java Class/Jar 文件信息并存储到数据库，NEL 使用 JDepend 对 Java Class/Jar 文件进行分析提取，JDepend 是 Clarkware 咨询公司的一个提取 java 依赖模型的开源工具。SourceParser 为 NEL 与 JDepend 的提供了接口，通过 SourceParser 可以得到 JDepend 所提取到的系统信息，SourceEntityBuilder 将提取到的信息进行格式处理并存储数据库。

3. 用户界面主要类图(GUI 包)。

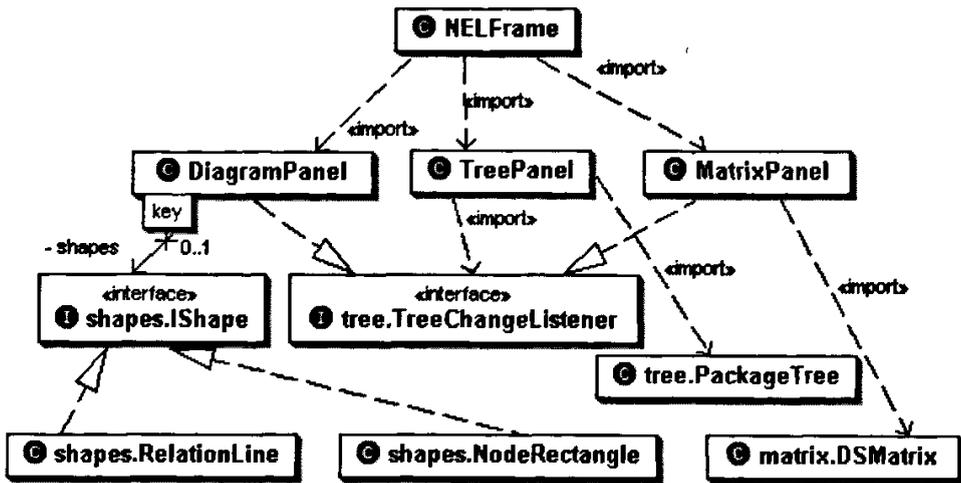


图 5-6 用户界面主要类图(GUI 包)

图 5-6 为系统用户界面主要类图,其中 NELFrame 为 NEL 界面的主框架类,它包含了三个面板: TreePanel 为包的树形结构、DiagramPanel 为画图面板、MatrixPanel 为 DSM 矩阵面板, TreeChangeListener 监听器的作用是将树形包结构的展开和关闭动作通知给矩阵和画板。另外 gui 包中包含了 shapes、matrix 和 tree 三个子包作用分别为绘制节点、绘制 DSM 矩阵和绘制包树形结构。

### 5.3 系统实现

通过需求、分析和设计我们使用 Eclipse、Hibernate 3.2.0 和 MySql 实现了 NEL。本小节主要描述 NEL 的两种信息抽象方法的实现。

#### 5.3.1 信息提取的实现

NEL 使用通过扩展 JDepend<sup>[64]</sup>对 Java Class 文件进行分析,表 5-1 为 NEL 所能提取到的 Java 系统中的关系。

表 5-1 所提取的元素和关系

源元素	关系	目标元素	描述
Class	association	Class	关联关系
Class	implements	Class(interface)	实现接口
Class	extends	Class(Class/Abstract)	继承关系
Package	contains	Class	“包”包含类
Package	contains	Package	“包”包含包
Package	efferent	Package	“包”引用(import)“包”
Class	import	Package	类“引用”包

### 5.3.2 HQL 查询的实现

NEL 采用 HQL(Hibernate Query Language)做为查询语言, 图 5-7 为 NEL 查询的运行界面, 通过在输入框中输入 HQL 脚本语言, 点击执行, 则可以在画图面板中查看查询结果, 图 5-7 所执行的脚本为“from PPRelation ppr where ppr.callee.name='org.wlfb.nel.entity'”, 其含义为从关系集合 PPRelation (包和包之间的关系集合)中获取被引用包为 org.wlfb.nel.entity 的关系集合。

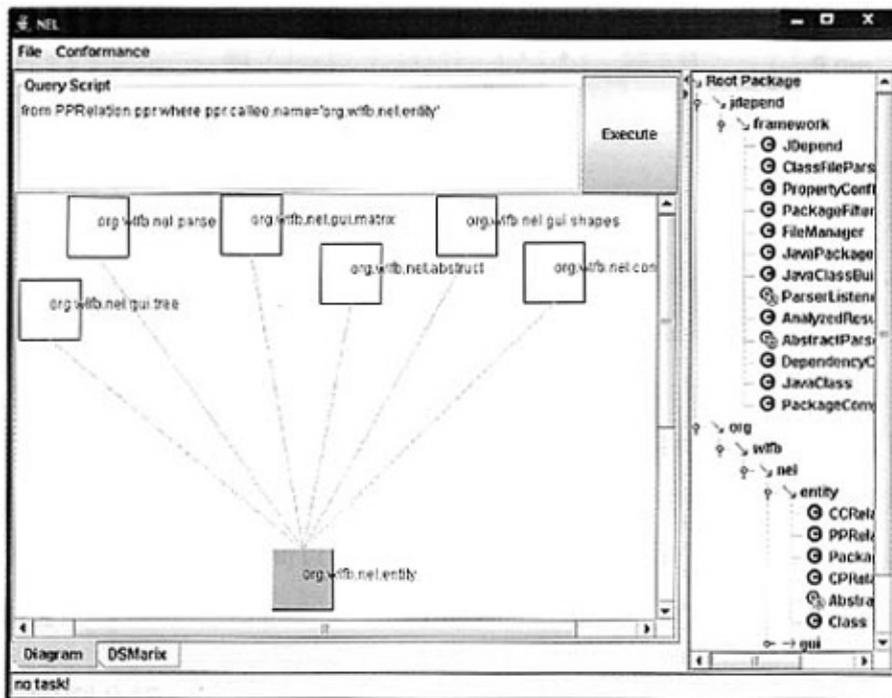


图 5-7 HQL 查询界面

### 5.3.3 设计结构矩阵的实现

图 5-8 为系统 DSM 矩阵的运行界面, NEL 采用 java 的包机制进行系统的分解, 矩阵的元素是系统的 Java 包, NEL 使用目标 Java 系统的包结构做为 DSM 分析的起点。DSM 抽象过程的一个主要步骤是 DSM 的划分, 如果一个系统具有完美(无环)的层次结构排序后的 DSM 应该是一个下三角矩阵, 然后利用 DSM 分层算法可以得到系统的层次结构。如果存在环, 根据划分后矩阵排列也能很快够通过聚合环中的包从而发现架构信息。

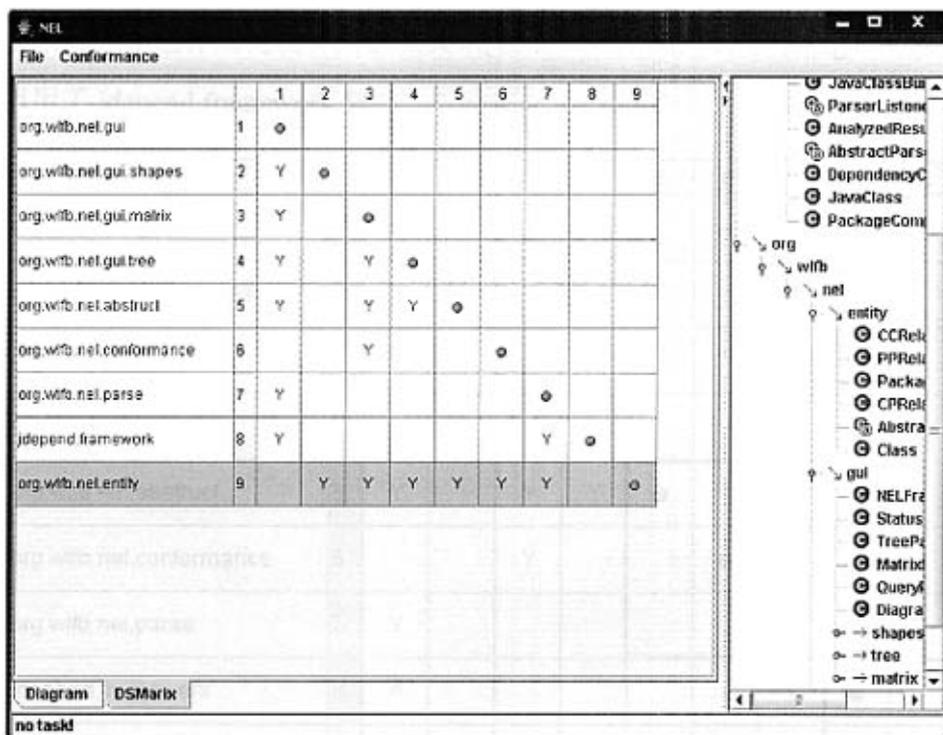


图 5-8 设计结构矩阵界面

NEL 系统实现了 DSM 划分的算法, 目前没有实现自动化的聚合方法, NEL 构建系统静态树形结构(如图 5-8 右边树形区域), 通过树形结构中包的展开(expand)和关闭(collapse)可以得到新的 DSM 矩阵, 从而可以得到不同层次的抽象结构。

## 5.4 实例分析

本节利用 NEL 对其本身进行分析, 最后将比较分析结果与 NEL 最初的设计进行比较以验证 NEL 分析方法的有效性和正确性。在这里所有的分析都是在信息数据库构建完成后开始的。

### 5.4.1 矩阵的分析

这里们输入架构规则{org.wlfb.net.gui,can not import, jdepend.framework }该架构规则表明了 gui 不能引用 jdepend。图 5-9 为经过划分处理的表示 NEL 系统包依赖关系的 DSM 矩阵, 可以看出 NEL 系统具有完美的(无环)层次结构, 根据 4.3.3.4 节描述的分层步骤和矩阵划分结果, 我们对 NEL 的包架构进行了以下的层次划分: {1}, {2, 3}, {4}, {5, 6, 7}, {8, 9}。同时我们可以发现目

前该系统存在违背架构设计的依赖关系(图 5-9 中第 9 行第 1 列): org.wifb.nel.gui 包引用了 jdepend.framework 包。

		1	2	3	4	5	6	7	8	9
org.wifb.nel.gui	1	●								
org.wifb.nel.gui.shapes	2	Y	●							
org.wifb.nel.gui.matrix	3	Y		●						
org.wifb.nel.gui.tree	4	Y		Y	●					
org.wifb.nel.abstrcut	5	Y		Y	Y	●				
org.wifb.nel.conformance	6			Y			●			
org.wifb.nel.parse	7	Y						●		
jdepend.framework	8	X						Y	●	
org.wifb.nel.entity	9		Y	Y	Y	Y	Y	Y		●

图 5-9 划分过后的 NEL 系统的 DSM 矩阵

通过对 NEL 的 DSM 划分结果的分析,我们发现,DSM 矩阵自动分层的结果还是令人满意的,但所分的层次是并不是完全的符合设计,因为根据最初设计,整个 GUI 包应该为一个层次,但是这种错误应该是可以接受的,因为重构工程师对系统的领域知识的理解可以很容易地纠正这些错误。经过重新整理我们可以得到以下的分层: {1, 2, 3, 4}, {5, 6, 7}, {8, 9}。

#### 5.4.2 HQL 查询

根据矩阵分析的结果,我们通过执行“from PPRelation”,整理得到图 5-10 展示的 NEL 系统结构,可以看出该图所表达的信息和图 5-9 的矩阵是一致的。

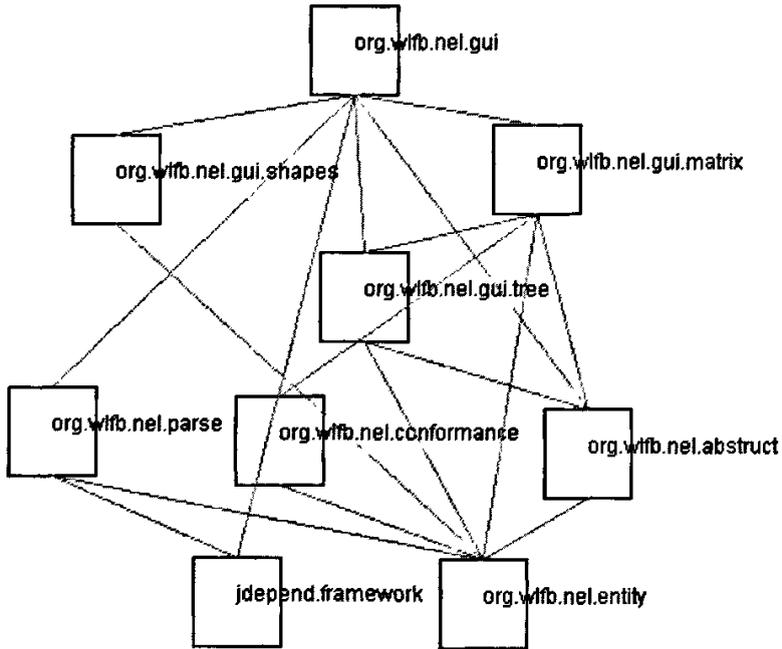


图 5-10 NEL“as-build”系统包视图

通过对节点进行聚合可以得到更加清晰的视图，如图 5-11 所示，该图所表示的系统包图和 NEL 的最初包设计(图 5-3)是基本一致的，不难看出 gui 包和 jdepend.framework 包产生了依赖关系，而在最初设计的包中它们是没有依赖关系的——这是一个违背了架构设计规则的问题依赖。

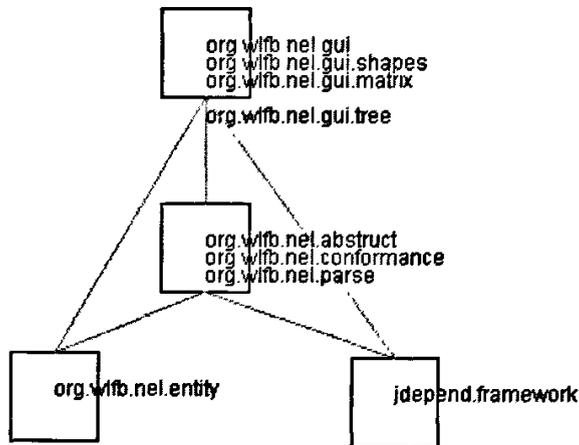


图 5-11 NEL“as-build”系统包视图(聚合)

接下来我们寻找产生违背架构设计规则的原因，执行“from CCRelation ccr

where `ccr.caller.container.name='org.wlfb.nel.gui'` and `ccr.callee.container.name='jdepend.framework'`”，其含义为在类与类的关系集合 `CCReration` 中查询 `Caller` 类属于 `gui` 包且 `callee` 类属于 `jdepend` 包的关系，我们可以得到 `gui` 包中和 `jdepend` 包产生依赖的类关系图，如图 5-12。这些关系的存在导致了 `gui` 包和 `jdepend` 包的依赖关系，我们应该进行相应的调整以去除这些关系。

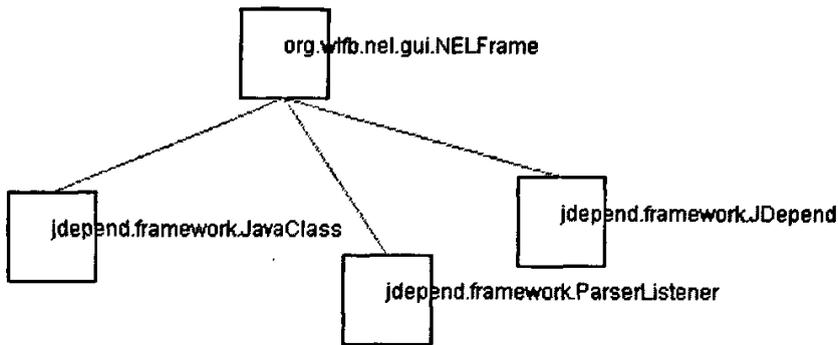


图 5-12 产生违背架构设计规则类关系图

## 5.5 系统评价

在实例分析中，NEL 的分析结果在一定程度上是令人满意的。尽管仅仅通过该实例分析无法评估系统在长期的软件开发和维护过程当中的有效性，但是我们发现 NEL 在提取架构信息、识别错误依赖、定位元素关系和系统依赖性展示等方面是很有帮助的。根据我们对 NEL 的使用，NEL 具有以下特点：

- **DSM 展示。**在系统规模比较大的情况下，DSM 可以很好的表示架构的层次结构，而对于直接从代码中提取的框图则是难以理解的。另外通过 DSM 矩阵可以很容易发现违背架构设计规则的问题依赖。
- **DSM 的划分算法。**为架构重构提供了一种自动化的机制，由划分算法处理过的 DSM 矩阵是工程师进行架构重构过程的良好开始。
- **包模式的识别**为重构过程提供了自顶向下浏览的向导，以得到合适的 DSM 矩阵。
- **HQL 的查询。**通过 HQL 脚本的执行可以：**a)**定义可重复的转换过程；**b)**在 DSM 中展示的关系，可以方便而快速地定位关系并用直观的框图显示；**c)**HQL 为面向对象的查询语言，易于编写。

另一方面，在 NEL 的使用过程中也发现一些问题：

- NEL 所使用的信息提取技术为静态分析，所以所提取到的关系均为 Class 文件中的显式引用，由于后期绑定，一些关系只有在运行当中才显示出来，这样有很多也许很重要的关系 NEL 无法提取，下一步是否有必要结合动态的分析技术？
- 在 NEL 中，矩阵元素之间存在关系仅仅用“Y”表示，是不是可以考虑用包含更多信息的数字或者某种符号来表示。
- 所提取的架构信息目前只能通过 HQL 脚本来保存，而没有很好的文档输出和错误报告。

## 5.6 本章小结

本章从分析、设计和实现的角度对架构重构系统 NEL 做了详细的介绍，最后给出了实例分析并做出了系统的评价，通过实例我们发现通过 DSM 矩阵和查询方法的结合我们可以很好的分析和重构软件构架并发现问题依赖。

## 第六章 总结与展望

### 6.1 本文工作总结

本文针对软件架构的重构技术进行了系统的研究，提出了使用依赖结构矩阵和 HQL 查询相结合的信息推论方法，在此基础之上设计并实现了架构重构工具 NEL。本文主要工作总结如下：

- 分析了现有的架构重构过程模型、重构方法和工具，并对这些方法和工具进行了评估。通过对评估结果的分析，发现目前的架构重构技术仍不能解决一些问题，如确定架构属性与架构的映射等。
- 根据现有的重构过程模型总结了软件架构重构的技术框架，包括：架构视点，解决为什么要重构以及要重构什么的问题；信息提取，解决如何提取所需信息的问题；信息推论，解决如何从大量的信息中提取有价值的架构信息；信息展示，如何以一种有效地方式向架构涉众展示架构信息。该框架为架构重构过程提供了一个很好的技术向导。
- 提出了依赖结构矩阵和包模式识别相结合的架构信息分析方法，并实现了使用这种信息推论方法的架构重构工具——NEL。最后通过实例分析证明了该方法的有效性。

### 6.2 未来的研究工作

NEL 仍然是一个实验性质的工具，有待改进，希望通过如下几个方向来扩展本文的研究工作：

- 支持多语言的架构重构。目前 NEL 仅仅支持对 Java Class 文件的分析，需要扩展 NEL 的信息提取技术，以支持其他语言如 C、C++ 等。另外有一些系统为多种语言混合实现的，提取各种语言的信息目前存在多种工具，然而如何使用从混合语言的系统提取到的信息去构建架构视图，仍然需要进一步的研究。
- 结合动态分析技术。目前 NEL 采用静态分析技术，然而由于后期绑定的原因，某些与架构相关的信息可能并不在源制品中，这就要求使用动态分析

技术去捕获系统的运行时信息，相信动态分析和静态分析的结合将会大大提高架构重构的质量。

- 扩展抽象方法。信息抽象是软件架构重构的关键，NEL 使用 Java 包对系统进行分解，然而对于其他语言如 C 等，应该通过数据挖掘技术进行聚类分析和模式识别。
- 多视图的架构重构。NEL 仅仅给架构涉众提供了开发视图，显然这不能满足某些涉众的需求，如何能提供多视图的架构重构方法也是下一步研究的重点。
- 可交互的用户界面。软件架构重构的非自动化性质，使得可交互性成为能否成功的进行架构重构的关键。本文希望通过模式推论，在整个重构过程当中给涉众提供一种指导性的浏览：当涉众对当前视图进行一个操作时，系统通过推论得到提示用户如何进行下一操作的信息。

## 参考文献

- [1] B Boehm. Engineering context(for software architecture),invited talk[C].In: Proceedings of the 1st International Workshop on Architecture for Software Systems Seattle, New York: ACM Press,1995:1~8.
- [2] 梅宏,申峻嵘. 软件体系结构研究进展[J]. Journal of Software, Vol.17, No.6, June 2006.
- [3] Len Bass, Paul Clements, Rick Kazman. Software architecture in Practice [M]. 2nd Edition, Pearson Education, Inc., publishing as Addison- Wesley 2003.
- [4] R.K. Fjeldstadt, W.T. Hamlen. Application Program Maintenance Study: Report to Our Respondents [R], 1984.
- [5] 王玉英,陈平等.软件逆向工程的研究与发展[J].西安工程科技学院学报,2006,20 卷 3 期.
- [6] 熊军.基于概念格的 Java 语言类层次分析研究与实现[D],吉林:吉林大学,2005.4.
- [7] 王振峰.基于 AOP 的逆向工程框架及工具的研究[D],西安:西安电子科技大学,2005.1.
- [8] Rigi: A visual tool for understanding legacy systems [EB/OL]. <http://www.rigi.csc.uvic.ca/>. Dec. 2004.
- [9] Lattix LDM Software for Architecture Management [EB/OL], [www.lattix.com](http://www.lattix.com).
- [10] Favre. Cacophony: Metamodel-driven software architecture reconstruction [C]. In WCRE, 2004.
- [11] A. van Deursen, C.Hofmeister, R.Koschke, L. Moonen, and C. Riva. Symphony: View-Driven Software Architecture Reconstruction [C]. In proceedings of the IEEE/IFIP Working Conference on Software Architecture (WICSA'04), 2004.
- [12] CMU SEI Architecture Definitions [EB/OL]. <http://www.sei.cmu.edu/architecture/definitions.html>.
- [13] The Institute of Electrical and Electronics Engineers (IEEE) Standards Board. Recommended Practice for Architectural Description of Software-Intensive Systems [S] (IEEE-Std-1471-2000). September 2000.
- [14] Soni, D., Nord, R., Hofmeister, C. Software Architecture in Industrial Applications [C], Proceedings of the Seventeenth International Conference on Software Engineering. ACM

- Press, 1995.
- [15] M. Fowler. Who Needs an Architect?[J], IEEE Software, July 2003
- [16] R. Holt, “Software Architecture as a Shared Mental Model”, International Workshop on Program Comprehension, 2003
- [17] Philippe Kruchten, Architectural Blueprints-The “4+1” View Model of Software Architecture [J]. IEEE Software 12(6) November 1995, pp. 42-50
- [18] Stephen T.Albin. The Art of Software and Techniques: Design Methods and Techniques [M]. Published by John Wiley & Sons.Inc 2003.
- [19] IEEE Std1471 Frequently Asked Questions [EB/OL].  
[http:// www.pithecanthropus.com/~awg/public\\_html/ieee-1471-faq.html](http://www.pithecanthropus.com/~awg/public_html/ieee-1471-faq.html).
- [20] 陈昊朋.软件逆向工程技术研究[D]. 西安: 西北工业大学, 2001.
- [21] E. J. Chikofsky and J. H. II. Cross. Reverse engineering and design recovery: a taxonomy [J]. IEEE Software, 7, 1990.
- [22] Byrne, E. A Conceptual Foundation for Software Re-engineering [C], Conference on Software Maintenance, 1992.
- [23] S.R.Tilley, S. Paul, D. B. Smith. Towards a Framework for Program Understanding [C]. In Proceedings of the 4th Workshop on Program Comprehension 1996: IEEE Computer Society Press (Order Number PR07283), March 1996.
- [24] IEEE Standard Glossary of Software Engineering Terminology [S] (R2002). IEEE Standard 610.12-1990.
- [25] R. Singh. International Standard ISO/IEC 12207: Software Life Cycle Processes [S]. June 1998.
- [26] J. Koskinen. Software Maintenance Costs [EB/OL]: University of Jyväskylä, Finland, Sept. 2004. <http://www.cs.jyu.fi/~koskinen/smcosts.htm>
- [27] M. Lehman. Laws of Software Evolution Revisited. In Proc. of the Fifth European Workshop in Software Process Technology(EWSPT'96), 1996, pp. 108-124.
- [28] Brodie, M. & Stonebrake, M: Migrating Legacy Systems [M], Morgan Kaufmann. Publishers, Inc., San Francisco, CA, 1995.
- [29] Andreas. Wierda. Architecture Reconstruction of Industrial Object-Oriented Software A

- case study [D]. Technische university Eindhoven, master thesis, August 2005.
- [30] L. O'Brien, C. Stoermer, C. Verhoef. Software Architecture Reconstruction: Practice Needs and Current Approaches [M]. SEI Technical Report CMU/SEI-2002-TR-024, Software Engineering Institute, Carnegie Mellon University, Aug. 2002.
- [31] From Ancient Egypt to Model Driven Engineering [EB/OL], series available at <http://www-adele.imag.fr/mda/>.
- [32] C. Riva. Reverse Architecting: An Industrial Experience Report [C]. In Proc. of the Seventh Working Conference on Reverse Engineering (WCRE'00), Nov. 2000, pp. 42-50.
- [33] Rick Kazman, S. Jeromy Carrière. Playing detective: constructing software architecture from available evidence [J]. Journal of Automated Software Engineering, 1999, 6(2):107-138.
- [34] G.Y. Guo, J.M. Atlee and R. Kazman. A Software Architecture Reconstruction Method [C]. In Proc. of the First Working IFIP Conference on Software Architecture (WICSA), Feb. 1999, pp 15-33
- [35] Shrimp Views [EB/OL]. [http://shrimp.cs.uvic.ca/shrimp/shrimp\\_intro.shtml/](http://shrimp.cs.uvic.ca/shrimp/shrimp_intro.shtml/) (2002).
- [36] Murphy. Lightweight structural summarization as an aid to software evolution [D]. PhD thesis, University of Washington, 1996.
- [37] Kamran Sartipi. Software Architecture Recovery based on Pattern Matching [D]. PhD thesis, University of Waterloo. Waterloo, Canada, 2003.
- [38] Damien Pollet, Stéphane Ducasse, Loïc Poyet, Ilham Alloui, Sorana Cîmpan and Hervé Verjus. Towards A Process-Oriented Software Architecture Reconstruction Taxonomy [C]. In 11th European Conference on Software Maintenance and Reengineering (CSMR), IEEE Computer Society, March, 2007.
- [39] Lanza and Ducasse. Polymetric views—a lightweight visual approach to reverse engineering [J]. IEEE TSE, 29(9), 2003.
- [40] Lungu, Lanza, and Gîrba. Package patterns for visual architecture recovery [C]. In CSMR, 2006.
- [41] Murphy, Notkin, and Sullivan. Software reflexion models: Bridging the gap between source and high-level models [J]. In FSE, 1995.

- [42] Tilley, Cole, Becker, and Eklund. A Survey of Formal Concept Analysis Support for Software Engineering Activities [C]. In Stumme, ed., ICFCA. Springer, 2003.
- [43] Jacobson, I., Booch, G., Rumbaugh, J., the Unified Software Development Process [M], Addison-Wesley, Reading MA, 1999.
- [44] 邱凤萍. 既存系统逆向工程整体自动化解决方案研究[D],北京: 北京工业大学, 2005,5.
- [45] 林炜. 逆向工程技术研究. 上海: 复旦大学 2005,5.
- [46] Apache Software Foundation [CP/OL], <http://jakarta.apache.org/bcel/manual.html>.
- [47] Tim Lindholm, Frank Yellin. The Java Virtual Machine Specification [EB/OL].<http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html>.
- [48] P. Berkhin. Survey of Clustering Data Mining Techniques [R]. Technical report, Accrue Software, San Jose, California, 2002.
- [49] 戴涛. 聚类分析算法研究[D]. 北京: 清华大学,2005.
- [50] 杨小兵. 聚类分析中若干关键技术的研究[D]. 浙江: 浙江大学,2005.
- [51] 赵健. 一种基于形式概念分析的语句级方面挖掘方法[D].吉林: 吉林大学,2005,5
- [52] K. Sartipi, K. Kontogiannis. Pattern-based Software Architecture Recovery [C]. In Proc. of the Second ASERC Workshop on Software Architecture, Feb. 2003.
- [53] P.Tonella, G.Antoniol. Object Oriented Design Pattern Inference [C]. In Proc. of the International Conference on Software Maintenance (ICSM'99), Sept. 1999, pp. 230-238.
- [54] D. V. Steward. The Design Structure System: A Method for Managing the Design of Complex System [J], IEEE Trans. on Engineering Management. August 1981, 71-74.
- [55] 徐晓刚. 设计结构矩阵研究及其在设计管理中的应用 [D]. 重庆: 重庆大学,2003.9.
- [56] 陈昊朋. 面向对象的软件逆向工程数据库系统设计[J], 计算机工程与应用, 2001 年 21 期.
- [57] Parnas, D.L. Designing software for ease of extension and contraction [J]. IEEE Transactions on Software Engineering, SE-5, Issue: 2 , on page(s): 128- 138, 1979.
- [58] 夏昕,曹晓钢,唐勇. 深入浅出 Hibernate [M]. 电子工业出版社.2005.7.
- [59] Hibernate [CP/OL]. <http://www.hibernate.org>.
- [60] T. Ball and S. E. Eick. Software Visualization in the Large [J]. IEEE Computer, April 1996. 33-43.

- [61] 张志猛. 面向理解的 OORE 关键技术研究[D]. 浙江: 浙江大学,2004,4.
- [62] D. Holten. Interactive Software Visualization within the RECONSTRUCTOR Project. Doctoral Colloquium [C], IEEE Visualization Conference, 2006
- [63] D. Holten. Hierarchical Edge Bundles: Visualization of Adjacency Relations in Hierarchical Data [C]. IEEE Transactions on Visualization and Computer Graphics 12(5):741-748, 2006 (best paper award)
- [64] Clarkware Consulting, Inc JDepend [CP/OL].  
<http://www.clarkware.com/software/JDepend.html>

## 致谢

这篇毕业论文的写作主要是在我研究生期间所做的一些工作和已完成的论文的基础上进行的，其中概括介绍了我在研究生学习阶段在软件构架相关技术问题上的一些认识和研究总结。

首先要由衷地感谢我的导师房鼎益教授。在整个论文的选题到完成，房老师自始至终都给予了特别的关心和鼓励。他严谨治学的学术态度，先进而丰富的专业知识，孜孜不倦的学习态度和淡定儒雅的君子之风，都给我留下了难以忘怀的印象，是我今后生活中的一笔宝贵财富。能成为他的弟子，我深感荣幸。

感谢网络实验室的安娜和陈晓江老师，他们在这些年来对我的关心和关照，我此生难忘。师兄汤战勇、沈静博、何路无论在生活上和学习上对我的关心和指导，感谢他们。

感谢我的好友：王义立、朱江涛、齐文华、罗养霞、于海燕、王婷等。和他们在一起渡过了短暂三年的学习历程，其间大家相互协作，相互勉励，相互帮助。这些都将成为珍贵的回忆。

特别感谢 Vicky Wang，在我最困难的时候给了我信心和帮助。

感谢所有曾经指导、教育、关心、支持、帮助过我的每一个人，祝福他们好人一生平安！

最后感谢您耐心的看完此文，并给予宝贵的意见。

## 攻读学位期间发表论文以及科研情况

### 论文发表

电子文档保护系统的设计与实现, 微电子学与计算机, 2006 年第 23 卷第 9 期, 第一作者。

### 科研项目

- 数字产品安全保护系统(陕西省国际科技合作重点项目, 2006KW-21), 项目参与开发人员。
- 基于 Linux 的短信通信平台(陕西省教育厅产业化重点项目, 05JC27), 主要开发人员。
- 分布式软件体系结构及其构造环境研究(陕西省自然科学基金, 2003F20) 项目参与人员。
- X X 军区国防动员潜力信息管理系统(横向联合项目), 项目参与开发人员。
- X X 银行短信通信管理系统(横向联合项目), 项目参与开发人员。
- X X 电子文档安全管理系统(横向联合项目), 主要设计与开发人员。

