摘　要

重构指在不改变软件外在行为的前提下，改善软件内部结构，从而在软件演化过程中优化软件质量，提高软件可理解性、可维护性和可扩展性等。二十多年来，人们对重构技术进行了深入地研究，许多研究成果已经在软件开发中得到了广泛应用，其重要性也得到了普遍认可。尽管如此，随着程序规模和复杂性的增加，重构依然面临诸多挑战，例如针对软件架构的重构方法偏少、重构效率较低、对动态语言的支持不足等。因此，如何对大规模软件的进行高效地重构仍是值得研究的重要课题。

为进一步促进重构在软件开发中的应用，本文对大规模软件重构中的若干关键技术进行了深入研究，从一定程度上克服了现有研究的不足。本文首先针对软件物理结构的优化问题，提出了物理重构的概念和方法体系；然后针对包结构重构，提出了两种面向重构的包内聚性度量方法，并以此为基础，讨论了度量驱动的包结构重构技术；最后针对动态语言的重构正确性评估，提出了 Python 程序的类型约束系统以及相应的约束构建算法和约束检查算法。具体而言，论文工作的主要成果表现在以下几个方面：

- 提出了物理重构的概念和方法体系，通过优化系统的物理结构，来提高大规模软件的开发和维护效率。与现有技术相比，物理重构关注于物理结构的演化，这对于大规模软件的开发、维护和重用具有重要作用。此外，还提出了一个物理重构目录，不仅为开发者提供重构时机与重构方法的指导，还有助于提高开发者交流的效率。
- 提出了两种面向重构的包内聚性度量方法，通过包所在的上下文来计算包的内聚性。与传统方法相比，本方法同时考虑了包内和包间数据依赖，能有效挖掘包在语义上的内聚程度，从而更加广泛地适应于各种类型的包。
- 提出了度量驱动的包结构重构框架以及相应的方法体系，能够自动化地提高包结构的质量。本方法首先通过内聚性、耦合性、稳定性以及抽象性等度量方法来识别存在设计缺陷的包结构，然后根据缺陷类型选择合适的重构方法进行重构。与手工重构相比，它不但显著地提高了重构效率，而且有效地降低了引入软件缺陷的可能。
- 为表达 Python 对象的类型约束，提出了类型约束系统 PyConcept，可以有效支持类型缺陷检查、文档自动生成等多种软件工程活动。在此基础上，给出了 PyConcpet 构建算法，通过分析对象使用方式，高效地生成动态语言的类型约束。进一步地，还给出了类型缺陷的检查算法，有助于提高重构过程的效率。

关键词　软件重构，软件演化，模块内聚性，物理设计，动态语言

Abstract

Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the system, to improve the system's understandability, maintainability, and extensibility in the software evolution. In the past twenty years, many refactoring techniques have been proposed, which promotes the widespread use of refactoring in the software development. It is generally accepted that refactoring is an effective technique for enhancing software quality.

However, with the increase in the scale and complexity of software, the refactoring still faces many challenges, including lack of refactorings for software architecture, low efficiency, insufficient support for dynamic language, etc. To solve these problems, this paper first proposes physical refactoring. Then, it presents two refactoring oriented cohesion metrics for package, and based on this, gives measurement driven refactoring for package structure. Finally for dynamic language, it presents the type constraint system of Python and the corresponding constraint construction and check algorithm.

The main contributions of the paper are listed as follows:

- Physical refactoring is proposed to help developers optimize the physical structure in the evolution of large-scale systems, which should be performed as an iterative process: "identify - refactor - assess". Compared to existing techniques, physical refactoring converges on the evolution of the physical structure, which is of great importance to the development, maintenance and reuse of large-scale systems. Then, a catalog of physical refactorings are put forward, which not only provides developers with guidelines on when and where to refactor physically and how best to refactor physically, but also enhances the level of communication among developers.
- Two package cohesion metrics based on package context are proposed for helping the refactoring of package structure. Compared to existing works, these metrics involve both inter- and intra- package data dependencies, which can mine the semantic couplings between classes, thereby being suitable for evaluating package cohesion.
- Measurement driven refactoring algorithm for packge structure are presented to make up the shortages of manual refactoring. It uses metrics, including cohesion, coupling, stability, abstraction, etc., to identify the undesirable package structure and select proper refactorings to improve package quality based on a set of heuristic rules, which not only increases the efficiency of refactoring of package structure, but also reduces the probability of introducing bugs.
- Type constraint system of Python is proposed to express the structure conformance, which can support any activity in software development that requires type information, including checking type bugs, developing refactoring tools. Based on this system, the construction algorithm of the type constraints is presented, which not only can reduce the cost of adopting the system, but also assit the algorithms needing type information. Futhermore, the check algorithm of the type constraints are put forward to detect the type bugs, thereby helping to improving the refactoring efficiency.

Keywords: software refactoring, software evolution, module cohesion, physical design, dynamic language

东南大学学位论文独创性声明

本人声明所呈交的学位论文是我个人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得东南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了谢意。

研究生签名：周天琳　日　期：2009.9.7

东南大学学位论文使用授权声明

东南大学、中国科学技术信息研究所、国家图书馆有权保留本人所送交学位论文的复印件和电子文档，可以采用影印、缩印或其他复制手段保存论文。本人电子文档的内容和纸质论文的内容相一致。除在保密期内的保密论文外，允许论文被查阅和借阅，可以公布（包括刊登）论文的全部或部分内容。论文的公布（包括刊登）授权东南大学研究生院办理。

研究生签名：周天琳　导师签名：徐敦　日　期：2009-9-10

第一章 引言

1.1 选题依据

为在软件演化过程中控制软件架构的复杂性，研究者提出了大量方法。其中，重构是一种被广泛使用的有效技术[1-12]。从狭义上讲，重构是一种程序变换，它在不改变软件外在行为的前提下，通过增量式的、受控的方式来逐步调整软件的设计，从而改进软件的内部结构[5, 11]。虽然重构方法的数量众多，但其核心思想均为重新组织包、类、函数以及变量等元素，为软件架构的修改和扩展奠定基础[1, 5, 11]，如提取接口[5]、引入观察者模式[12]等。从广义上讲，重构不只是程序变换，更是一个“识别-狭义重构-评估”的迭代过程。图 1.1 对这一过程进行了描述：首先根据设计原则、开发经验、软件度量以及开发需求等，识别需要重构的不良软件设计；然后选择合适的重构方法消除不良设计；最后评估重构效果，以作出设计决策。重构效果的评估包括正确性和有效性两个方面。正确性是指重构不改变软件的外在行为，如不引入软件缺陷、未增添额外功能等。正确性评估主要依赖软件构建和软件测试。此外，模型检查[13]、程序证明[14]以及程序分析（如基于静态分析的软件缺陷检查[15]）等也能够用于评估重构正确性。这些技术可以帮助发现构建和测试难以发现的软件缺陷，从而向开发者提供重构正确性的多种反馈。有效性是指重构提高了软件质量。其评估方法与“识别”阶段的评估方法相同。

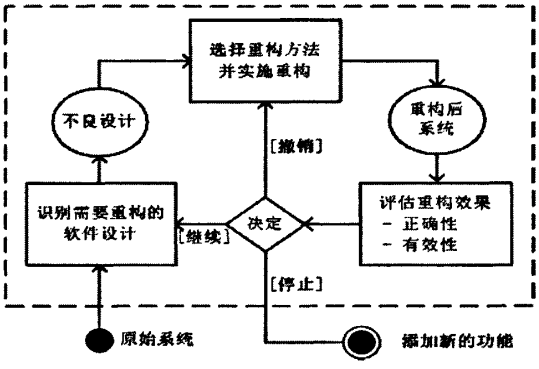


图 1.1 广义的重构过程

软件架构既包括逻辑结构，也包括物理结构。然而，传统重构技术主要调整软件的逻辑结构，对物理结构关注较少。软件的逻辑结构主要涉及类、对象、函数等逻辑实体以及继承、实现、调用等逻辑关系。而其物理结构则包含文件、目录结构以及连接和编译依赖等[16]。目前，研究如何设计软件物理结构的领域被称为物理设计[16]。由这一领域的相关研究可知，良好的物理结构可以明显提高团队的开发效率、降低软件的构建、集成、测试和发布代价，对于大规模软件的构造与维护具有重要意义。为优化物理结构的质量，我们有必要研究物理结构的重构技术。鉴于 C++被广泛地应用于大型软件系统的构造，本文主要以 C++程序为对象进行深入研究。

如何重构软件系统的包结构是物理结构重构的核心问题之一。但是，包结构通常涉及大量的文件和目录，手工重构的困难较高。一方面，包结构的重构要求开发者修改与被重构包有关的导入语句，而这些导入语句往往散布于数量庞大的文件中。在这种情况下，手工重构不仅代价较高，还容易引发错误。另一方面，单纯依赖人工评审来重构包结构的成本较大。开发者往往需要阅读大量的代码和文档来理解复杂的包结构，以发现存在设计缺陷的包结构以及检查重构的有效性。更严重地，人工评审的结果因人而异，初级开发者往往由于自身经验匮乏而做出错误判断。因而，我们需要深入研究包结构重构的自动化技术，以有

效避免手工重构的不足。

作为重要的设计属性，内聚性常被用于实现类重构的自动化。众多研究者就如何准确而方便地计算类内聚性进行了积极探索[17-44]。然而就目前来看，包内聚性的评估还主要依靠开发者的经验，这严重阻碍了内聚性作为质量指标在包结构重构中的广泛应用。虽然研究者提出了一批包内聚性度量方法[26, 45-49]，但包语义的复杂性常导致度量结果与开发经验和设计原则等相悖。因而，如何计算包内聚性已成为一个亟待解决的问题。这一问题的解决将有助于我们研究度量驱动的包结构重构技术，从而显著地提高重构效率。

由于简单和灵活的特性，动态语言得到了快速发展，并业已成为开发大规模 Web 应用程序的主流语言[50-52]。为提高这些软件的开发和维护效率，我们需要对面向动态语言的重构提供良好支持。但是，动态语言具有动态类型系统，其类型检查在运行时进行，这不利于发现重构所造成的类型缺陷（在本文中，我们称违反类型系统规则的缺陷为类型缺陷），从而给重构的正确性评估造成了困难。首先，运行时检查难以提供类型缺陷的快速反馈，不利于提高重构效率。其次，解释器不对任何未被执行的语句进行类型检查，这使得对函数、类等的类型检查依赖于执行路径。最后，运行时检查只能覆盖有限的路径和上下文，有可能遗漏类型缺陷。静态程序分析是检查和定位类型缺陷的重要技术[15]，但现有针对动态语言的静态分析技术往往难以较好地表达动态语言的类型约束[53-59]，也就难以有效地检查类型缺陷。因此，有必要进一步研究动态语言的类型缺陷检查技术，以辅助评估重构正确性。考虑到 Python 是最为流行的动态语言之一，本文关注于如何更好地检查 Python 程序的重构正确性。

本文在广泛调研的基础上，结合国家杰出青年科学基金、国家自然科学基金项目、江苏省自然科学基金项目、江苏省科技攻关项目等，试图通过研究大规模软件重构及其度量技术，从一定程度上解决上述问题。在物理设计方面，提出了物理重构的概念和方法体系，以在软件演化过程中持续优化系统的物理结构。在包结构重构方面，提出了两种新的度量方法，可以有效评估各种包的内聚程度，并在此基础上讨论了度量驱动的包结构重构技术。在动态语言方面，提出了 Python 类型约束系统以及相应的约束构建算法和约束检查算法，可为 Python 程序的重构提供良好支持。

1.2 国内外研究现状

自 1992 年 W. Opdyke 在其博士论文中引入重构概念以来[11]，重构逐渐成为了一种软件开发的最佳实践[4]。它允许开发者在软件开发的任何阶段优化软件质量，使得良好的软件设计不再是开发的前提条件，而是开发的结果[5]。鉴于此，大量研究者在重构领域进行了深入而广泛的研究，取得了一批极有价值的成果。然而，传统重构技术较少关注系统的物理结构。尽管物理设计致力于改进系统的物理结构，但它难以支持物理结构的持续演化。

模块内聚性反映了模块的内部元素为完成共同任务而紧密耦合的程度，常用于识别可重构的软件设计和评估重构的有效性，所以对于自动化重构具有重要意义。为此，研究者提出了大量的模块内聚性度量方法。现有的主要成果集中于类。

静态程序分析可以发现多种类型的软件缺陷，是验证重构正确性的重要手段之一。因此，对于 Python 程序而言，通过静态程序分析定位测试难以发现的类型缺陷是一种合理思路。但是，现有的 Python 静态分析技术在这个方面存在一定的局限性。

1.2.1 软件重构

在软件开发和维护中，软件重构作为一项行之有效的系统优化技术得到了广泛应用，并成为软件工程领域的热点研究问题。现有的主要工作包括识别坏味道、根据坏味道选择重构方法以及对总结重构方法。这里，坏味道是指需要重构的软件设计[4]。

在识别坏味道和根据坏味道选择重构方法方面，M. Flower 总结了面向对象系统中常见的坏味道，如过长函数、数据泥团等。此外，他还讨论了根据坏味道选择重构方法的启发式规则[5]。S. Ducasse 等提出了一种重复代码识别技术，可用来检查诸如发散式变化[5]之类的坏味道[60]。M. Balazinska 等通过克隆代码

识别技术来定位重复代码，并提出了对应的重构方法[61]。Y. Kataoka 等提出了利用不变量识别坏味道的方法[62]。F. Simon 等通过度量识别可重构的类设计，并实现了相应的支撑工具[63]。这些技术对于指导开发者制定重构策略以及实现重构自动化有重要作用。

在总结重构方法方面，M. Flower 提出了一个重构目录，该目录涵盖了一批被广泛使用的面向对象的原子重构方法[5]。作为 M. Flower 工作的有益补充，J. Kerievsky 提出了模式驱动的重构目录，系统地总结了通过重构引入设计模式的方法[12]。事实上，设计模式可作为重构的目标[67]。此外，在面向方面程序设计（Aspect-Oriented Programming，AOP[64]）领域，C. Zhang 等提出了利用 AOP 来重构中间件的方法，首先根据词法分析识别横切关注点，然后通过方面来封装横切关注点[65]。M. P. Monteiro 等讨论了如何将 OOP 程序重构为 AOP 程序，并提出了相应的重构目录[66]。Y. Ping 等深入研究了 Web 程序的重构问题[9]。W. Jie 等提出了一种基于重用代价优化的组件重构方法[3]。上述工作着重研究如何实施重构，不但有助于提高开发效率，还有助于扩大重构的适用范围。

然而，传统重构技术难以在软件演化过程中持续优化软件的物理结构。这是由于这些技术主要关注软件的逻辑结构，而软件还涉及物理结构。根据物理设计领域的研究，良好的物理结构有助于降低大规模软件的开发和维护成本。

1.2.2 物理设计

大量研究者对软件的物理设计进行了积极探索，取得了一批有意义的成果。虽然这些工作不能有效支持系统物理结构的持续演化，但却为我们的研究奠定了良好的基础。

在设计方法方面，J. Lakos详细地讨论了大规模C++程序的物理设计技术[16]。他首先定义了组件、包以及物理依赖（编译依赖和连接依赖），然后系统地总结了消除循环物理依赖和隔离类私有成员的方法。R. C. Martin展示了如何根据物理设计原则来组织包结构的方法[45]。此外，B. Stroupstrup、H. Sutter、S. Meyers和S. Dewhurs等提出了一批调整编译依赖的物理设计方法[68-72]。

在质量评估方面，J. Lakos讨论了无环依赖原则，并定义了累积组件依赖（Cumulated Component Dependence，CCD）来度量子系统的可测试性。进一步地，他认为编码规范和编程约定等可用于指导包的设计[16]。与此类似，S. Mcconnell也强调了编码规范和编程约定在包设计中的作用[73]。而E. Evans等则认为包结构应反映领域知识[74]。此外，R. C. Martin给出了一系列包的设计原则和支持这些原则的度量方法[45]。

在自动化工具方面，一些常用的标准工具能够高效地抽取物理依赖，如 gmake[75]、mkmf[76]以及 cdep[16]等。基于 J. Lakos 和 R. C. Martin 的工作，K. Paton 开发了度量工具 PDCHECK，它不仅可评估逻辑实体到物理实体的映射是否符合 J. Lakos 提出了的分派原则，还能检查物理依赖是否满足 R. C. Martin 提出的物理设计原则[77]。E. Hautus 等研制了工具 OptimalAdvisor，该工具能够检验系统的物理设计是否符合无环依赖原则和稳定抽象原则[78]。

1.2.3 模块内聚性度量

根据计算方法的不同，模块内聚性度量可被分为五类：基于图论的方法、基于数据流（切片）的方法、基于信息论的方法、基于共享类型的方法和基于信息检索的方法。

基于图论的方法首先用图表示被度量模块（顶点表示模块的内部成员，而边表示各内部成员之间的交互关系），然后根据图论知识定义度量[27]。典型的方法包括 S. Chidamber 的 LCOM1 度量[20]，M. Hitz 的 LCOM2 度量[21]，H. Chae 的 CO 度量[22]以及 L. Briand 的 RCI 度量[48]等。这类方法的优势在于计算和理解都较为简单，所以得到了广泛使用。但是，它们仅考虑交互数，度量结果往往与开发经验相悖。

基于数据流的方法通过数据切片来判断模块的内部成员之间的数据耦合关系，从一定程度上弥补了图论方法只考虑交互数的缺陷。例如，L. Ott 等通过函数输出参数的数据流切片的相似程度来计算函数的内聚性[24]。然而，这类方法也存在一定的局限性。其一，在度量类内聚性时，它们大多不考虑特殊方法（如构造函数，存取函数等）对类内聚性的影响。其二，由于依赖切片算法，所以它们的计算复杂性较高。

基于信息论的方法由 E. Allen 等在 2001 年提出：首先将模块的内部交互图对应的关联矩阵看作信源，

而将关联矩阵中的行看作信源发出的符号，然后定义模块内聚性为余熵[26]。虽然这类方法考虑了交互方式，但却未区分交互种类（如变量之间的交互和方法之间的交互等）。因此，仍然存在度量结果与模块实际内聚程度不一致的情况。

基于共享类型的方法认为若模块的两个内部成员使用了相同的类型，则它们之间存在语义联系[46]。这类方法的核心思想是内部成员所使用的类型从一定程度上反映了它们的语义。由于模块内聚性应体现模块的内部成员在语义上的耦合程度，因此这类方法将模块内聚性定义为其内部成员使用相同类型的程度。典型方法包括 J. Bansiya 的 CAMC 度量[35]，S. Counsel 的 NHD 度量[17, 44]等。相对于前两类方法，这类方法仅依赖模块界面，从而有助于将内聚性评估提前到软件开发的设计阶段。研究表明更早的度量是成功开发软件的一个关键因素[48]。此外，这类方法仅考虑模块所使用的类型，而、不需进行数据流依赖分析，所以使用和实现都比较简单。不过在很多情况下，模块所使用的类型名难以准确反映模块的语义。

基于信息检索的方法首先抽取源代码中的语义信息（如注释和标识符等），然后通过文档相似性评估技术来分析模块中各成员间的语义距离。典型方法是 LORM 度量[25]。与其他方法相比，这类方法更多地考虑了模块的内部成员之间的语义关系，且计算较为简单。但是，这类方法强烈地依赖于编码风格，在实际应用中存在一定的局限性。

1.2.4 Python 程序静态分析

Python 程序的静态分析主要通过扫描诸如源代码之类的静态信息来获取程序特定方面的属性，它主要包括静态标注、静态类型推导和静态检查工具等。然而，这些技术均存在一定的不足。

静态标注是指在静态时对对象属性进行描述，从而为类型检查、编译优化、形式化文档等提供良好支持。G. van Rossum 提出了可选静态类型声明来表达对象的具体类型[53]。但是，该方法需要修改 Python 语法，不能应用于既有的 Python 代码。Python3.0 对 Python2.6 进行了扩充，引入了函数标注和抽象基类[54]。其中，函数标注提供了一种标准的方式来注解函数的参数，可以是任意合法的 Python 表达式。然而，它同样需要修改 Python 语法，且只能对函数参数进行约束。而抽象基类描述了实例应具备的行为，主要用于支持函数 *isinstance* 和 *issubclass* 的重载[55]。与函数标注类似，它也是一种入侵式机制，不仅难以向后兼容大量的遗产代码，还不利于抽象基类的自动生成。更重要地，它着眼于在运行时判断实例的类型，难以处理诸如函数参数这样的非实例对象。

静态类型推导通过静态分析来推断对象的类型，以进行编译优化和类型缺陷检查等。J. Aycock 提出了一种类型推导算法 *ATI*，能够在 Python 的一个受限子集上进行类型推导[56]。B. Cannon 给出了一种局部化类型推导算法，可以推导局部变量的类型[57]。然而，Python 的动态类型决定了静态类型推导只能获取部分对象的类型。此外，这种技术难以推导函数参数的类型。

静态检查工具以静态分析为基础，可以帮助定位代码缺陷和不良软件设计。PyChecker 不仅能检查“使用未定义变量”、“传递错误的参数个数”等类型缺陷，还能判断“给定模块是否满足一批预定义的编码规则”，如在同一作用域内重定义函数等[58]。Pylint 扩展了 PyChecker，增加了检查“单行代码长度”、“变量命名是否规范”以及“接口是否实现”等特性[59]。然而，它们只能检查简单的类型缺陷。

1.3 主要研究内容

根据国内外研究现状，我们在 C++程序的物理重构、面向重构的包内聚性度量方法、度量驱动的包结构重构技术、Python 程序的重构正确性评估方法以及物理重构原型系统等方面进行了探索。具体而言，论文将在以下几个方面展开研究。

1.3.1 C++程序的物理重构

我们充分考察软件重构的基本思想和物理设计的主要技术，提出了物理重构的概念，它在不改变软件外在行为的前提下，调整软件的物理结构，从而提高软件的开发效率和可维护性等。与传统重构类似，物理重构也是一个“识别-重构-评估”的迭代过程。进一步地，还提出了一个物理重构目录。该目录不仅涵

盖了主要物理重构方法的基本步骤，还给出了根据不良物理结构的类型选择适当重构方法的启发式规则。实例研究表明物理重构能够有效地优化系统的物理结构，使开发者从多个角度持续改善软件质量。

1.3.2 面向重构的包内聚性度量方法

包对于大规模系统的构造和维护具有较大意义。作为重要的质量属性，包内聚性常被用于指导包结构的重构。然而，现有的包内聚性度量方法依赖于包内部的数据流关系，常导致度量结果与实际开发经验相悖。为解决这一问题，我们提出了两种基于上下文的包内聚性度量方法，具体研究内容如下：

(1) 包的分类

包所承担的任务是决定包内聚程度的核心因素。高内聚的包集中处理一个特定任务，服务于一致的抽象。这样的包复杂度低，具有更高的可理解性、可维护性、可重用性和可测试性。反之，低内聚的包往往试图完成过多的任务，常导致代码混乱和重复。因此，我们根据包的功能将包分为工具包、接口包、实现包和构件包：工具包旨在封装一组具备通用基础功能的类以服务于通用编程；接口包包含一组抽象类，用于表达系统规约；实现包是对系统规约的实现，由同一抽象包中的抽象类的子类组成；构件包封装一个子系统以完成相对完整的任务。

(2) 基于客户使用的包内聚性度量方法

提出了共同重用内聚性 CRC，其基本思想是若多个类总是被共同重用，则它们之间存在紧密的语义耦合。然后根据包的分类讨论了 CRC 内聚性与语义内聚性之间的关系，并在此基础上提出了计算 CRC 内聚的 HC 度量。与现有方法相比，HC 度量同时考虑了包内和包间的数据依赖，所以能有效地反映包的内部类之间的语义关系。此外，HC 度量还通过考察客户对包的使用方式来提高度量结果的可区分性。

(3) 基于相似上下文的包内聚性度量方法

提出了一种改进的包内聚性度量方法 SCC。与已有方法相比，SCC 度量通过类所在的上下文来推断类之间的耦合关系：两个类所在的上下文相似程度越高，它们的语义耦合性也越高；反之，它们的上下文相似程度越低，它们的语义耦合程度也越低。此外，与 HC 度量类似，SCC 度量同时考虑了包内和包间的数据依赖，能够有效挖掘类之间的语义联系。在本文中，我们将 HC 度量和 SCC 度量统称为基于上下文的包内聚性度量方法。

(4) 包内聚性度量方法的实验研究

通过对大型开源软件进行实验来研究各种包内聚性度量方法的效果，从而获得了一批有价值的结论。首先，本文方法比传统方法更加有效。其次，充足客户集可以保证本文方法获得稳定而有意义的度量值。再次，虽然与 HC 度量相比，SCC 度量的计算复杂性更高，但是它的优势并不明显。最后，充足客户集在保证 HC 度量值的稳定性上存在条件，即新客户不能打破客户集中已有各种客户之间的平衡。可以看出，实验研究是理论研究的有力补充。

1.3.3 度量驱动的包结构重构技术

提出了度量驱动的包结构重构框架以及相应的重构算法，可以自动化地消除存在设计缺陷的包结构，从而有效避免手工重构中存在的各种问题。它的基本过程为首先通过稳定性、抽象性、内聚性和耦合性等的度量来识别违反设计原则的不良包结构，然后根据启发式规则选择合适的重构方法进行重构，最后再次利用度量来考察重构是否提高了包结构的质量。与手工重构相比，度量驱动的包结构重构技术不仅能帮助大规模软件的开发者提高重构效率，还能较好地减少由重构引入的软件缺陷。

1.3.4 Python 程序的重构正确性评估方法

Python 的动态类型系统给 Python 程序的重构正确性评估造成了一定的困难。为此，本文首先提出了 Python 类型约束系统 PyConcept，然后基于 PyConcept，提出了一种 PyConcept 构建算法 *generate_concept* 和 PyConcept 检查算法 *check_func_call*，具体如下：

(1) Python 的类型约束系统 PyConcept

提出了 Python 的类型约束系统 PyConcept，可以通过 concept 来表达 Python 对象基于结构一致性的类型约束。由理论和实例两方面的研究可知，该系统能有效支持类型缺陷的检查、重构工具的开发以及形式化文档的生成等多种软件工程活动，对于提高大规模软件的开发效率有重要作用。

(2) PyConcept 的构建算法 *generate_concept*

提出了 PyConcept 的构建算法 *generate_concept*，可以通过分析函数参数在函数体中的使用方式来自动生成这些参数的类型约束。该算法不仅降低了 PyConcept 的使用成本，还可为其它以 PyConcept 为基础的算法提供自动生成类型约束的服务（如算法 *check_func_call*）。

(3) PyConcept 的检查算法 *check_func_call*

提出了 PyConcept 的检查算法 *check_func_call*，能够检查函数调用点是否满足 PyConcept 所规定的类型约束：对于给定调用点 *call_site*，若实参 *arg* 不满足 PyConcept 所规定的约束，则 *call_site* 存在类型缺陷。作为测试的有力补充，该算法能有效定位类型缺陷，从而有助于检查 Python 程序的重构正确性。

1.3.5 物理重构原型系统

为提高物理重构的效率，快速对比不同的物理重构技术，我们研发了一个物理重构原型系统 Alchemist，为物理重构的研究提供了一个可扩展的实验平台。Alchemist 首先构建系统的物理依赖图或逻辑依赖图。然后基于依赖图，利用度量方法来计算系统的各种属性。最后根据度量结果制定重构策略：若系统质量满足要求，则停止重构；若存在不良设计，则修改依赖图和源代码；若上一轮重构并未有效提高系统质量，则撤销修改。这里，制定重构策略既可通过手工进行，也可由算法 *refactor_package* 完成。

1.4 论文主要成果

本文从大规模软件重构技术的研究现状出发，对软件物理结构的优化、面向重构的包内聚性度量方法，度量驱动的包结构重构技术以及动态语言的重构正确性评估等领域进行了深入的研究。具体而言，论文的主要成果表现在以下几个方面：

- 提出了物理重构的概念与方法体系。物理重构是在不改变软件外在行为的前提下，对软件物理结构的再设计，可以提高软件的开发效率和可维护性等。与传统重构类似，物理重构可采用“识别-重构-评估”的迭代过程。此外，还提出了一个物理重构目录，能有效指导开发者的重构实践。
- 提出了两种面向重构的包内聚性度量方法。第一种是基于客户使用的包内聚性度量方法。其核心思想是若多个类总被共同重用，则它们具有密切的联系。第二种是基于相似上下文的包内聚性度量方法。该方法认为若两个类的上下文相似，则它们之间存在紧密耦合。虽然第二种方法的计算复杂度远远高于第一种方法，但它更加全面的利用了包的上下文信息，所以其适用范围更加广泛。与现有方法相比，本文方法同时考虑了包内和包间的数据依赖，可以通过包所在的上下文来挖掘类之间的语义关系，从而能够适用于多种类型的包。
- 利用大规模实验对各种包内聚性度量方法进行了深入研究。实验结果表明基于上下文的方法比基于数据流的方法更加有效。这进一步论证了包内数据依赖难以较好地反映包的高层语义。实验结果还表明充足客户集可以保证基于上下文的方法获得稳定而有效的度量结果。虽然上述实验所得结论与理论结论基本一致，但是也存在一定的分歧。第一，与 HC 度量相比，SCC 度量不具备明显优势。第二，充足客户集在保证 HC 度量值的稳定性上存在条件。因此，理论研究需要实验研究的支持。
- 提出了度量驱动的包结构重构框架以及相应的重构算法，可以自动化地消除存在设计缺陷的包结构，从而有效避免手工重构中存在的各种问题。其基本思路是首先通过度量来识别存在设计缺陷的包结构，然后根据缺陷类型选择合适的重构方法来提高包结构的质量，最后再依靠度量评估重构有效性。
- 提出了 Python 的类型约束系统 PyConcept，可以较好地表达结构类型系统的约束。在此基础上，提出了 PyConcept 的构建算法 *generate_concept*。进一步地，为检查 Python 程序的重构正确性，还提出

了 PyConcept 的检查算法 *check_func_call*，可以高效地检查类型缺陷。

1.5 论文结构

论文共八章，可分成五大部分：第一部分为第 1 章，是全文的概述和引言。第二部分包括第 2 章，主要研究 C++程序的物理重构技术。第三部分包括第 3、4、5 章，是对包内聚性度量以及度量驱动的包结构重构技术的研究。第四部分包括 6 章，主要讨论 Python 程序的重构正确性检查。第五部分包括第 7 章，介绍了物理重构原型系统。

第 1 章作为论文的引言，介绍了本文的研究背景和研究意义以及国内外的研究现状和仍存在的不足，描述了论文的研究内容，最后简要地介绍了论文的创新点和论文结构。

第 2 章提出了物理重构的概念，并建议采用“识别-重构-评估”的迭代过程来实施重构。此外，提出了物理重构目录，用于指导开发者进行物理重构。

第 3 章提出了两种面向重构的包内聚性度量方法。一种是基于客户使用的包内聚性度量方法，可以通过客户对包的使用方式来评估包的内聚性。另一种是基于相似上下文的包内聚性度量方法，它使用类所在的上下文来推断类之间的耦合关系。两种方法均能较好地反映包的实际内聚程度。

第 4 章通过大规模实验来对各种包内聚性度量方法进行研究。实验结果表明相对于基于数据流的方法，基于上下文的方法更加有效。此外，实验结果还表明充足客户集能有效保证基于上下文的方法获得稳定而有意义的度量结果。

第 5 章提出了度量驱动的包结构重构框架，其核心思想是使用度量来重构存在设计缺陷的包结构。在此基础上，提出了重构算法 *refactor_package*。实例研究表明该算法可以高效地提高包结构的质量，从而从较大程度上克服手工重构的局限性。

第 6 章提出了 Python 的类型约束系统 PyConcept，可以为包括类型缺陷检查在内的多种软件工程活动提供静态类型信息。基于 PyConcept，提出了 PyConcept 的构建算法 *generate_concept*，能够生成函数参数的类型约束。此外，还提出了 PyConcept 的检查算法 *check_func_call* 来定位类型缺陷，从而有力地支持了 Python 程序的重构正确性评估。

第 7 章介绍了物理重构原型系统 Alchemist 的设计与实现。

第 8 章是对整个论文的总结，综述了我们在大规模软件重构及其度量领域所获得的成果，并简单阐述了我们正在进行或将要进行的研究工作。

第二章 C++程序的物理重构

物理结构主要涉及软件系统中物理实体的关系与组织，对于大规模软件的构造和维护具有重要意义。然而，传统的重构技术和物理设计技术难以持续地优化软件的物理结构。为此，本章充分研究软件重构的基本思想和物理设计的主要技术，提出了物理重构的概念和方法体系。本章首先讨论了物理重构的实施过程，然后介绍了物理重构的常用方法，包括方法的名称、步骤以及目标。实例研究表明物理重构能够有效地优化软件的物理结构，使开发者从多个角度不断地改善软件的质量。

作为一种通用的系统开发语言，C++被广泛地应用于大型软件系统的构造[70]。因此，本章主要从大规模C++程序开发的角度来讨论物理重构及其相关技术。从下面的分析可以看到，这些技术对于使用其它语言编写的软件系统仍然具有很高的参考价值。

2.1 基本思想

良好的物理结构对于保证开发工作的独立性和一致性具有重要意义，是决定大规模软件团队开发效率的关键性因素之一。下面我们将以局域网模拟系统 *NetSimulator* 为例，讨论C++软件的物理结构对软件质量和团队开发效率的影响。*NetSimulator* 仅为示例性程序，这里只是通过它说明进行物理重构的必要性和过程，并以此讨论大规模软件开发中的一些关键性问题。

如图2.1所示，*NetSimulator* 的主要功能是根据客户的配置模拟两种常见的局域网：星型网和环形网。其软件架构采用了桥接模式[67]：类 *Net* 是对局域网的抽象，而它的子类 *Star* 与 *Ring* 分别对应于星型网和环形网；类 *Node* 及其子类 *Comp* 和 *Hub* 代表局域网中的节点，如工作站、集线器等，而类 *Line* 和它的子类 *TPW* 和 *CC* 则代表局域网中的通信链路，如双绞线等。在图2.1中，类对应的组件被封装成不同的包，而组件之间则通过编译依赖相互联系。例如，包 *PElem* 由组件 *Net*、*Star* 和 *Ring* 组成，而组件 *Net* 编译依赖于组件 *Node*。下面，我们将结合图2.1对组件、包、编译依赖等物理实体进行详细解释：

组件包含一个或多个文件，是物理设计的基本单位，如 Python 的.py 文件、Java 的.class 文件等。在C++中，组件通常记录了类的声明和实现，由一对头文件（.h 文件、.hpp 文件等）和实现文件（.cpp 文件、.cxx 文件等）构成。特别地，若组件 *c* 记录了一个抽象类，则称 *c* 为抽象组件；否则，若 *c* 包含了一个具体类，则 *c* 是具体组件。在图2.1中，组件由实线矩形表示，而抽象组件的名称则采用斜体，以和具体组件相区别。例如，*Net* 是抽象组件，而 *Star* 是具体组件。在面向对象软件系统的开发中，维持这种类和组件之间的对应关系是一种良好的编程约定。

包是语义相关的组件的集合，是开发者进行独立开发的基本单位。与组件类似，由抽象组件构成的包被称为抽象包。本章将组件和包统称为物理实体。在C++中，一个文件夹是一个包。图2.2是图2.1对应的目录结构。在图2.2中，文件夹 *PNet* 和 *PElem* 分别代表了图2.1的包 *PNet* 和 *PElem*，而头文件 *Net.h* 和实现文件 *Net.cpp* 则对应图2.1的组件 *Net*。

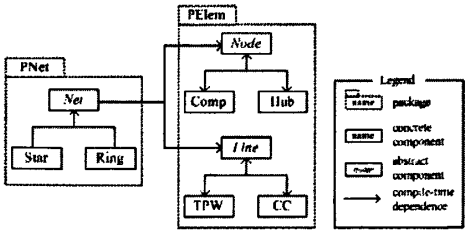


图 2.1 NetSimulator 的初始物理结构

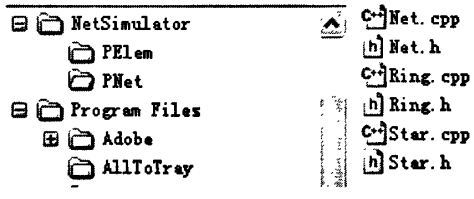


图 2.2 图 2.1 对应的目录结构

物理实体 e_1 编译依赖于物理实体 e_2，当且仅当 e_1 的编译需要 e_2 的头文件。通常，C++的编译依赖由 include 语句产生。如在图 2.1 中，组件 *Star* 在它的头文件 *Star.h* 中使用语句*#include "Net.h"*来导入组件 *Net* 的头文件 *Net.h*，所以 *Star* 编译依赖于 *Net*。普遍认为编译依赖的复杂性直接决定着系统的构建代价[16]。

由图 2.1 可知，*NetSimulator* 的初始结构难以适应持续变化的开发环境，如系统规模和复杂性的增加以及开发团队的重组等。假设在开发的初期，*NetSimulator* 的开发团队 *NetTeam* 仅由两个开发者 Mary 和 Tom 组成，他们分别负责包 *PNet* 和 *PElem*。随着系统规模和复杂性的不断上升，开发者 Jim 加入 *NetTeam* 以提高开发效率，但是图 2.1 的物理结构使得系统的开发效率很难获得预期的改善：

第一，初始结构使开发者难以独立工作。由于开发工作以包为单位进行划分，因此仅有 2 个包的系统无法适应 3 人开发团队。另外，诸如组件 *Comp*、*Hub*、*TPW* 等是包 *PElem* 的实现细节，其修改不应对包 *PElem* 外的物理实体产生影响。然而在图 2.1 的结构下，这样的修改要求 Mary 重新构建她所负责的包 *PNet*。因此，虽然在包 *PNet* 的界面（抽象类 *Node* 和 *Line*）确定之后，Tom 和 Jim 的注意力将集中在包 *PNet* 的实现细节上，但是他们的工作仍然会经常地干扰 Mary。更重要地，Tom 和 Jim 很可能出于对 Mary 的考虑而放弃某些合理的修改。

第二，初始结构不利于协调开发者之间的工作。软件构建是保证 *NetTeam* 工作一致性的重要手段。诸如接口不匹配、命名冲突等错误均可以通过构建发现。而且，高效的构建有助于更早更频繁地进行系统测试，从而及时地发现模块交互时的错误。因此，每日构建在软件开发中被广泛应用于检查软件缺陷和降低开发风险[45, 80-81]。但是在图 2.1 中，类 *Node*（*Line*）负责创建它的子类对象，这要求它依赖于它子类的定义，从而产生了组件 *Node*、*Comp* 和 *Hub*（*Line*、*TWP* 和 *CC*）之间的循环编译依赖。这种不良依赖显著增加了修改包 *PElem* 所带来的增量式构建的代价，不利于及时地同步开发者之间的工作。

由上述分析可知，为了提高团队的开发效率，我们应该持续地维护系统的物理结构，使其处于良好状态以满足软件开发的要求：

包结构 包结构从很大程度上决定着开发团队的结构[16, 45]。因此，它的质量直接影响着团队的开发效率。合理的包结构不仅使得包的内聚性较高、耦合性较低，也提供定义良好的接口，可以有效保证开发者高效而独立地工作。反之，不合理的包结构往往导致开发者之间的相互干扰。例如，包 *PNet* 的低内聚使得 Tom 和 Jim 难以独立工作。由此可见，随着项目规模和复杂性的增长，系统的目录结构（包结构）需要被持续地调整以适应项目不断变化的开发需求。

编译依赖 编译复杂性对构建成本有重大影响。虽然构建成本对于小型项目影响不大，但对于大型项目却可能成为支配开发时间的关键性因素之一[16]。例如，Microsoft Windows 2000 有大约 5 亿行代码，而这些代码分布于成千上万的文件中。这使得该项目的一次完整构建需要数台机器同时运行 19 个小时。尽管如此，由于在未成功构建的系统上工作是极不安全的，该项目的开发团队仍然坚持每日构建[73]。不难看出，构建是软件开发中不必可少的环节。进一步地，构建还是获取反馈的重要手段，能够帮助我们发现多种软件缺陷。合理的编译依赖降低了系统的构建代价，有助于协调不同开发者之间的工作。相反地，不合理的编译依赖会阻碍开发者之间的协作。例如，包 *PElem* 中的循环依赖增加了系统的增量式构建代价。基于此，消除不良编译依赖是持续优化系统物理结构的核心任务之一。

为了控制软件架构的复杂性，研究者提出了大量方法。其中，重构是一种得到广泛应用的有效技术。然而传统重构技术关注于软件的逻辑结构，难以有效解决 *NetSimulator* 所面临的问题。虽然物理设计可以优化软件的物理结构，但是该领域的大多数研究均未提及如何支持软件演化。

基于上述讨论，我们提出了物理重构的概念。与传统重构类似，物理重构在狭义上也是一种特殊的程序变换，它在保证程序行为不变的前提下，调整软件的物理结构，从而提高软件的开发效率和可维护性等。从概念的角度，物理重构是在已有代码的基础上，对物理结构进行再设计，以使其适应不断变化的软件开发需求。它允许开发者在软件生命周期的任何阶段调整系统的物理结构。从开发实践的角度，物理重构主要依靠物理设计的方法和评估标准来优化系统的物理结构，以提高团队开发效率和软件质量，如重新组织包、调整编译依赖等。它使开发者能够从物理设计的角度思考软件开发，软件质量以及软件维护等多个方面的问题。

可以看到，物理重构不是传统重构和物理设计的简单延伸，而是它们的有力补充。作为一种重构技术，它汲取了重构的基本思想，如小步骤重构、频繁测试等[5, 12]。作为一种物理设计技术，它采纳了该领域中与物理重构相关的技术，如物理设计的原则和方法等[16, 45]。为了解决诸如 *NetSimulator* 开发中所遇到的问题，我们总结这些最佳实践，将物理设计的具体技术引入了传统重构的框架中。

2.2 重构过程

根据传统重构和物理设计的特点，物理重构可采用“识别-重构-评估”的迭代过程：首先依据开发经验、设计原则等，识别需要重构的物理结构，然后选择适当的重构方法进行重构，最后评估重构后的系统以判断重构是否正确和有效。下面的章节将就这一过程进行详细讨论。

2.2.1 识别需要重构的物理结构

在重构前，我们应该对系统的物理结构进行评估以识别需要重构的物理结构，从而制定重构策略。通常，良好的物理结构应该满足物理设计原则、编码规范或编程约定等。本章主要利用表 2.1 所示的物理设计原则来指导物理重构[16, 45]。在表 2.1 中，设计原则可分为耦合性原则和内聚性原则。其中，耦合性原则主要用于评价物理实体之间的编译依赖是否合理，而内聚性原则则是组件划分的标准。因此，这些原则不仅为评估物理结构的质量提供了可依赖的标准，也为物理重构提供了目标。

表 2.1 常见的物理设计原则

类别	原则	描述
耦合性原则	无环依赖原则（ADP）	物理实体之间的依赖关系不能形成循环。
	稳定依赖原则（SDP）	物理实体应该依赖于比它稳定的物理实体。
	稳定抽象原则（SAP）	物理实体的抽象程度应该和其稳定程度一致。
	低构建代价原则（LBD）*	修改物理实体所引起的构建代价应该尽可能低。
内聚性原则	重用发布等价原则（REP）	包的重用粒度就是发布粒度。
	共同封闭原则（CCP）	包的所有组件对于同一类性质的变化应该是共同封闭的。
	共同重用原则（CRP）	包的所有组件应该是共同重用的。

注：*为本文提出的设计原则

NetSimulator 的物理结构较为简单，凭借经验和人工阅读，开发者可以发现大部分的不良设计。然而，大规模 C++系统往往包含成千上万的物理实体，而这些物理实体之间不仅具有复杂的关系，而且有可能属于不同的开发者。在这种情况下，开发者很难通过人工审查的方式理解系统复杂的物理结构，并以此发现不满足原则的设计。软件度量可以帮助开发者更为高效地发现物理结构的坏味道[16, 45]。所以与传统重构相比，物理重构更依赖于软件度量（工具）的支持。例如，R. C. Martin 在文献[45]中利用度量指标考察了薪酬系统 *Payroll* 的物理结构，并指出了度量在指导包结构设计中的重要作用。

2.2.2 重构

在定位了需要重构的物理结构之后，开发者应该针对不同情况选择恰当的重构方法对系统进行物理重构。在很多情况下，不良物理结构体现为不合理的包划分和不合理的编译依赖。因此，物理重构的主要策略是调整物理实体之间的编译依赖和重新组织系统中的包。

如表 2.2 所示，常用的 C++程序物理重构方法可以分为三类：调整组件间依赖、修改组件的实现和重新组织包结构。其中，调整组件间依赖和修改组件的实现以组件为对象，其主要目的是降低修改组件所带来的构建代价。而重新组织包结构则通过改进包间依赖和包的划分来降低包的构建代价，并帮助开发者独立地进行开发工作。从表 2.2 可以看出，很多物理重构方法同时也是传统重构方法，如提取组件[16]、引入对象工厂[45]和提取包[84]等。但作为物理重构方法，它们的主要目标是优化系统的物理结构，从而提高团队的开发效率，降低系统的构建、测试、发布等的成本。这些重构方法的详细讨论见 2.3 节。另外，很多物

理重构方法也同样适用于由其他语言编写的程序。例如，引入对象工厂、建立抽象组件[5, 16, 45]和合并包等也可用于对 Java 程序进行物理重构。

表 2.2 物理重构目录

类别	名称	别名	描述	目标
调整组件间依赖	移动成员	移动函数、移动值域	将组件中内聚性较低的部分转移到与该部分更为相关的另一个组件中。	ADP、SDP、SAP、LBP
	提取组件	提取类	将组件中内聚性较低的部分封装为一个新的组件。	ADP、SDP、SAP、LBP
	提取公共组件	升级、降级、管理类	将多个组件中较内聚的部分提取到一个新的组件。	ADP、LBP
	建立抽象组件	协议类、提取接口、依赖倒置	抽象出一个服务接口，让需要获得服务的类直接向接口发送消息。	ADP、SDP、SAP、LBP
	引入对象工厂	对象工厂模式	用对象工厂负责创建对象。	ADP、SDP、SAP、LBP
修改组件的实现	消除私有继承*		用分层技术代替私有继承。	SDP、LBP
	使用 PImpl 惯用法*	PImpl 惯用法、完全隔离的具体类	首先将私有成员提取出来封装成一个类放入实现文件中，然后在原类的头文件中添加一个指向新类的指针，最后通过指针访问私有成员。	LBP、SDP
	提供前置声明头文件*		为前置声明提供头文件，如 STL 中的头文件<iosfwd>。	LBP
	封装系统依赖*		为不同的编译系统或运行环境提供独立的头文件。	LBP
重新组织包结构	移动组件	移动类	将组件从一个包移动到另一个包。	ADP、SDP、SAP、LBP、REP、CCP、CRP
	提取包		将包中内聚性较高的组件提取到一个新的包。	ADP、SDP、SAP、LBP、REP、CCP、CRP
	提取抽象包		将抽象类提取到一个新的包。	SDP、SAP
	提取公共包		提取多个包中较为内聚的组件到一个新的包。	ADP、LBP
	合并包		将多个包合并为一个包。	ADP、REP CCP

注：*为 C++专有重构方法

2.2.3 评估重构效果

在重构后，我们需要再次对被重构的系统进行评估以确定物理重构的效果。与传统重构类似，评估的内容主要包括物理重构的正确性和有效性。

所谓正确性主要是指“对于一组给定的输入，重构前后的程序应该产生相同的输出”[11]。我们通常从两方面保证正确性。一是小步骤重构。它不仅降低了引入错误的可能性，也降低了定位和排除错误的成本。二是持续的构建和测试。它们可以及时检查重构是否改变了系统的外在行为。

所谓有效性是指所使用的物理重构方法能够提高系统物理结构的质量。由上述讨论可知，2.2.1 节所介绍的技术也同样适用于评估重构后的系统质量。通过对比重构前后的评估结果，我们可以确定重构是否切实优化了系统的物理结构。

2.3 重构方法

本节将结合 *NetSimulator* 详细讨论如何利用表 2.2 中的物理重构方法来优化系统的物理结构，从而提高团队的开发效率和软件质量。虽然 *NetSimulator* 只是一个简单的例子，但是下面展示的重构方法和重构过程也完全适用于大规模软件系统的开发和维护。

2.3.1 调整组件间依赖

调整组件间依赖通过把组件的功能在组件体系结构上重新分配来优化组件间编译依赖，以使组件能够满足无环依赖原则、稳定依赖原则、稳定抽象原则以及低构建代价原则。这里，功能是指一组语义上紧密相关的代码。当开发者遇到诸如“组件间存在循环编译依赖”、“修改某个组件所带来的构建代价较高”、“某个需要频繁修改的底层组件被大量组件依赖”之类的情况时，他们可以利用本节所介绍的重构方法来对“问题”组件进行调整。

这类方法的具体策略是将导致不良物理设计的功能转移到其他组件中（如提取组件）或为这些功能建立抽象接口（如建立抽象组件）。在实际重构过程中，重构方法的选取需要视情况而定。通常，若被重构组件的内聚性较低，则采用“拆分”方法，即将已有组件重新组织。这类方法包括移动成员[5]、提取组件、提取公共组件以及引入对象工厂等。反之，若被重构组件的内聚性较高，则可使用建立抽象组件。下面，我们结合 *NetSimulator* 对此进行说明。

在图 2.1 中，包 *PElem* 中的循环依赖违反了无环依赖原则和低构建代价原则。考虑到本例中的循环依赖由创建功能引起，我们采用引入对象工厂来分离创建子类对象的功能。重构前后的代码见图 2.3：创建工厂类 *NFac*(*LFac*)，由它负责创建类 *Node*（*Line*）的子类对象，从而消除组件 *Node* 对其子类组件的编译依赖。其中，工厂 *NFac* 的实现采用了单件模式和对象工厂模式[67, 79]：类 *NFac* 的单件对象拥有一个从类型标识符到子类创建函数的映射表 *map_*，这样客户可通过标识符来访问所需的创建函数。而向 *map_*中添加映射关系的代码定义于子类组件的实现文件中（*Comp.cpp* 和 *Hub.cpp*），从而使得工厂组件不再依赖于产品组件，而是产品组件依赖于工厂组件。

在引入对象工厂后，我们通过提取包来将组件 *NFac* 和 *LFac* 封装入一个新的包 *PFac* 中，以便于对对象工厂进行独立地开发与维护。最终重构结果如图 2.4(a)所示。由图 2.4(a)可知，重构后的包 *PElem* 满足了无环依赖原则和低构建代价原则，这有效降低了修改它所引起的增量式构建的代价。当然，引入对象工厂和提取包不一定要同时使用。具体重构策略需要开发者综合考虑物理设计原则、开发经验以及系统的实际情况来制定。有关提取包的讨论详见 2.3.3 节。

Before Refactoring

```
// In file Node.h
class Node{
public:
   // Create an object of Node's subtype
   static Node *Create(const char *typeId);
   // Other members
};
```

After Refactoring

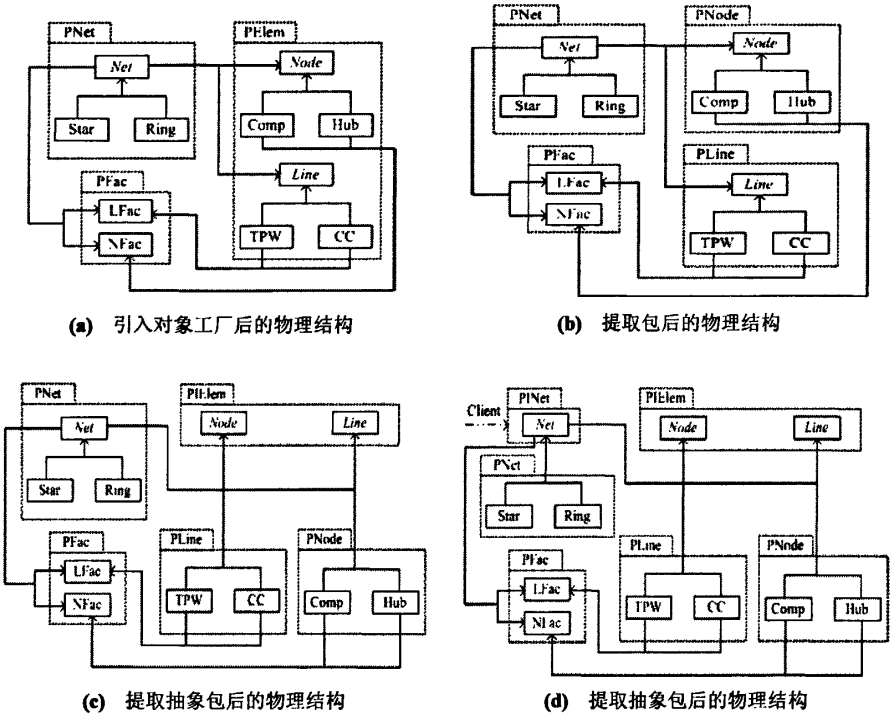
```
// In file NFac.h
#include <map>
#include <string>
// NFac is a Singleton Object Factory
class NFac{
public:
   typedef  Node* (*CreateNode)( );
   // Add (typeId, Creation function) to map_
   bool Register(const std::string &typeId,
                 CreateNode Fn);
   // Create concrete object based on typeId
   Node* Create(const std::string &typeId);
   // Return the handle of the singleton object
   static NFac& Only();
private:
   std::map<std::string, CreateNode> map_;
   …
};

// In file Hub.cpp
// Function to create a Hub's object
static Node* Hub::CreateHub(){
   return new Hub;
}
const bool registered =
   NFac::Only().Register("Hub", Hub::CreateHub);
```

图 2.3 对图 2.1 采用引入对象工厂

2.3.2 修改组件的实现

修改组件的实现主要有两种策略。第一种是通过消除物理实体对组件私有实现的依赖来降低编译依赖的强度，从而使组件满足稳定依赖原则和低构建代价原则。在 C++中，头文件中的所有内容，甚至包括类的私有成员对其客户都是可见的。换句话说，组件的客户编译依赖于组件的私有实现[69]。这使得对头文件的任何修改，即使是私有成员（对客户是不可访问的），都会导致其客户的重编译，从而降低了组件的可修改性和可维护性等。第二种是通过修改 include 语句来去除物理实体对头文件的冗余依赖，如提供前置声明头文件[68]、封装系统依赖[82]等。



(a) 引入对象工厂后的物理结构

(b) 提取包后的物理结构

(c) 提取抽象包后的物理结构

(d) 提取抽象包后的物理结构

图 2.4 对图 2.1 的进行物理重构

本节着重讨论第一种策略，它的基本思路是将类的私有实现从头文件转移到实现文件，从而允许私有实现独立于客户进行变化，这有效提高了组件的可修改性。这种策略的优点在于它不需要修改组件的客户，但它只适应于仅是私有部分经常变化的情况。对于公有部分也经常改变的组件，应该采用调整组件间依赖。在实际重构中，若只有部分的私有实现需要经常修改，则应该采用部分隔离技术，如消除私有继承[16]；反之，若所有的私有实现均需要经常修改，则应该采用完全隔离技术，如使用 PImpl 惯用法[16, 68-72]。

在图 2.4(a)中，包 *PFac* 被 *NetSimulator* 中的所有包依赖，因此对其头文件（*NFac.h* 和 *LFac.h*）的任何修改，即使是私有实现（如 *std::map<std::string, CreateNode> map_*），都将导致整个系统的重编译，这违反了低构建代价原则。为了隔离私有实现的影响，降低增量式构建代价，我们通过使用 PImpl 惯用法来对组件 *NFac* 进行物理重构（重构前后代码如图 2.5 所示）：首先，在组件 *NFac* 的头文件中声明嵌套类 *Impl* 和指向该嵌套类对象的不透明指针 *pImpl_*。其次，在组件的实现文件 *NFac.cpp* 中定义类 *NFac::Impl*。该类包括了类 *NFac* 的所有私有成员。再次，修改类 *NFac* 的公有成员函数，使它们通过指针 *pImpl_*访问类 *NFac* 的原私有成员。最后，删除类 *NFac* 的私有成员。对组件 *LFac* 采用相同的重构步骤。

在重构后，类 *NFac*（*LFac*）的私有成员全部定义在相应的实现文件中，对它们的修改将不会导致客户的重编译，只需重连接即可，这显著降低了开发者之间的互相干扰。一般地，被隔离组件具有较高的稳定性，因而依赖于这样的组件将更加安全。同时，使用 PImpl 惯用法有助于保证强异常安全和增强信息隐藏[83]。在大规模系统中，这种重构方法常被用于隔离基础组件的私有实现。这里，基础组件是指被系统中大多数组件所依赖的组件。

2.3.3 重新组织包结构

前两类重构方法着眼于降低组件的构建代价，有助于协调不同开发者之间的工作。而重新组织包结构则通过在包体系结构上重新安排组件的位置来优化包的划分和包间编译依赖，以使包满足耦合性原则和内聚性原则，对于保证开发工作的一致性和独立性有重要作用。这类方法可以用于诸如“开发者相互干扰”、

“一个开发者的工作过度依赖于另一个开发者的工作”以及“包的构建代价过高”等情况。

重新组织包结构的主要策略是通过将导致不良物理设计的组件转移到其他包中来优化系统的包结构。与前两类方法类似，我们仍然需要根据实际情况选择不同的重构方法。若包违反了稳定抽象原则，则往往采用提取抽象包。其核心思想与建立抽象组件一致，均是将对非抽象物理实体的依赖转移到抽象物理实体上。事实上，抽象物理实体对应的抽象逻辑实体代表了系统的规约，是系统的稳定部分，因而依赖于这样的物理实体是较为安全的。若包违反了其他耦合性原则（如无环依赖原则），则可利用提取公共包和合并包等来消除不合理的包间编译依赖。若包违反了内聚性原则，则应使用移动组件[84]、提取包[84]等来提高包的内聚程度。

Before Refactoring

```
// In file NFac.h
#include <map>
#include <string>
class NFac{
public:
   // public members
private:
   std::map<std::string, CreateNode> map_;
};
```

After Refactoring

```
// In file NFac.h
class NFac{
public:
   // public members
private:
   class Impl;
   Impl* pImpl_;
};

// In file NFac.cpp
#include "NFac.h"
class NFac::Impl{
   // The implementation of NFac can
   // be changed at will without
   // recompiling its clients
   std::map<std::string, CreateNode> map_;
   …
};
```

图 2.5 对图 2.4(a)应用使用 PImpl 惯用法

下面将通过重新组织包结构来改善 *NetSimulator* 的包结构以保证开发工作的独立性。在前两次重构后，*NetSimulator* 仍然存在一些问题。首先，包 *PElem* 封装了两个相对独立的子系统 *Node* 和 *Line*，这违反了包的内聚性原则，从而使得该包的开发者 Tom 和 Jim 难以并行工作。为克服这一问题，我们采用提取包将该包拆分为两个内聚的包 *PLine* 和 *PNode*：将子系统 *Node* 的组件（*Comp* 和 *Hub*）从包 *PElem* 中提取出来，封装成一个新的包 *PNode*，并将包 *PElem* 更名为 *PLine*。重构后的系统如图 2.4(b)所示。在重构后，Tom 和 Jim 可以在互不干扰的情况下高效工作，从而提高了 *NetTeam* 的开发效率。

由图 2.4(b)可知，重构后的系统仍不能完全满足开发需求。包 *PNode* 和 *PLine* 中包含了两个子系统的实现细节，依赖于这样的包违反了稳定抽象原则。在这种情况下，Mary 严重地将受到 Tom 和 Jim 的干扰。考虑到这两个包同时包含抽象组件和具体组件，我们使用提取抽象包：将抽象组件 *Node* 和 *Line* 分别从包 *PNode* 和 *PLine* 中提取出来，封装成抽象包 *PIElem*，其余不变。重构后的系统如图 2.4(c)所示。在重构后，包 *PNet* 只需依赖于抽象包 *PIElem*，不再依赖于子系统的实现细节。这样，Mary 可以独立工作，而不必受 Tom 和 Jim 的影响，她的开发效率可以得到明显提高。

在图 2.4(c)中，包 *PNet* 同时含有系统的抽象规约（类 *Net*）和系统的实现细节（类 *Star* 和 *Ring*），不满足稳定抽象原则。在开发过程中，对包 *PNet* 的修改会导致其客户的重构建。与前一次重构类似，我们仍然采用提取抽象包：将抽象组件 *Net* 从包 *PNet* 中提取出来，封装成抽象包 *PINet*，对外提供服务接口。重构后的系统如图 2.4(d)所示。经过 5 次迭代，系统的物理结构已满足开发需求，重构可以停止。

然而，包结构的改变有可能要求开发团队的组织结构进行相应的调整。即是说，开发工作可能需要重新分配。而频繁地改变团队结构不利于开发效率的提高，所以使用这类方法时应仔细考虑。

2.3.4 实例分析

上述实例表明物理重构能够显著提高团队的开发效率。一方面，物理重构使得开发者可以更加容易地并行工作。物理重构重新组织了 *NetSimulator* 的包（2.3.3 节），使得 Mary、Tom、Jim 和客户包的开发者均不依赖于易改变的物理实体，从而能够并行工作。这里虽然包 *PFac* 不是抽象包，但它由被隔离的组件构成，非常稳定。因此，依赖于它与依赖于抽象包一样安全。另一方面，物理重构降低了系统的构建、集成和测试代价，有助于协调不同开发者的工作。上述重构通过消除系统中的循环编译依赖（2.3.1 节）以及隔离工厂的私有实现（2.3.2 节）等多种重构方法来降低了 *NetSimulator* 的编译复杂度。

该实例还说明物理重构的过程在广义上与传统重构相似。其一，重构前后都应评估系统的质量。虽然 *NetSimulator* 的评估主要依赖开发经验，但是大规模系统的复杂性决定了它们的重构需要自动化工具的支持。其二，物理重构是一个迭代过程。通常，良好的物理结构难以通过单次重构获得。其三，物理重构同样采用小步骤。例如，*NetSimulator* 经过 5 次重构才满足开发需求。并且，每次重构都只进行少量修改，这有助于保证重构的正确性。

2.4 相关工作

物理重构是一种改善系统物理结构的重构技术。它以大规模软件的物理设计技术为基础，借鉴了传统重构的经验，采用“识别-重构-评估”的迭代过程。首先，利用表 2.1 中的设计原则找出需要重构的物理结构。其次，根据第一步的评估结果、系统的实际情况以及开发经验等，选择表 2.2 中的重构手法进行重构。最后，通过构建、测试和度量等方法评估重构效果。可以认为，物理重构的过程与方法具备广泛的实践基础，其基本思想是非常可靠的。更重要地，物理重构并非物理设计和传统重构的简单相加。

2.4.1 物理重构与传统重构技术

由上述以 *NetSimulator* 为例的讨论可知，物理重构并非传统重构技术的简单延伸，它们在重构的目标、方法以及评估方式等方面有着显著的区别：

目标 传统重构的主要目标是提高系统可理解性、可重用性和可修改性。而物理重构则旨在提高团队的开发效率，并降低系统的构建、测试和发布代价。以 *NetSimulator* 为例，传统重构考虑诸如“是否应该采用提取函数对类 *Star* 中某个过长函数进行拆分，以提高它的可重用性和可读性”之类的逻辑设计问题。而物理重构则关注软件的物理结构，如“是否应该利用提取包来提高包 *PElem* 的内聚性，以使 Tom 和 Jim 能并行工作”。

方法 传统重构针对软件的逻辑结构，其主要方法包括修改类、函数等逻辑实体的界面与实现。而物理重构则往往通过合理划分包以及优化编译依赖等来改善软件的物理结构，如重新安排代码的定义位置（修改组件的实现）、调整文件间的导入关系（调整组件间依赖）以及重新组织目录（重新组织包结构）等。当然，也存在一些重构方法可以同时提高物理结构和逻辑结构的质量。以 *NetSimulator* 为例，2.3.1 节的引入对象工厂不仅消除了包 *PElem* 中的循环编译依赖，还使得客户仅需通过类型标识符 *TypeId* 来创建类 *Node* 和 *Line* 的子类对象，而不需知道这些子类实现。这样，子类的创建逻辑可以独立于客户自由变化。

评估方式 传统重构主要通过针对逻辑结构的设计原则来定位需要重构的代码，而物理重构则依赖于一组不同的标准。以 *NetSimulator* 为例，传统重构更多地考虑如何识别不良的逻辑结构，如重复代码、过长函数等，而物理重构则关注与编译依赖、包结构等相关的设计原则。在图 2.4(a)中，物理重构降低了修改工厂类所引起的增量式构建的代价。

当然，逻辑结构和物理结构是软件架构的两个方面，它们共同决定着软件系统的质量，因此物理重构和传统重构不是对立技术，它们之间存在着紧密联系：

首先，传统重构是引起物理重构的重要因素。传统重构会修改逻辑结构，而逻辑结构又是物理结构的基础，所以它会导致物理结构的演化。为了在演化过程中控制物理结构的质量，我们需要物理重构。

其次，物理重构常反作用于逻辑结构。以组件为对象的物理重构方法（调整组件间依赖和修改组件的实现）会修改系统的逻辑结构。例如，为了消除包 *PElem* 中的循环编译依赖，我们创建了工厂类 *NFac* 和 *LFac*，这改变了系统的类体系结构。

再者，物理重构有助于提高传统重构的效率。物理重构可以提高系统的可测试性和团队的开发效率等，这使得开发者能够较为容易地修改和测试代码。例如，重构后的 *NetSimulator* 使得 Tom 能够自由地对他所负责的部分实施传统重构，并且高效地测试传统重构后的代码。

2.4.2 物理设计

大量研究者对物理设计进行了积极探索，并取得了一批有价值的研究成果，主要包括物理结构的设计

方法与质量评估以及相应的自动化工具等。所有这些成果构成了物理重构的研究基础。

物理结构的设计方法为物理重构提供了一批重构方法。J. Lakos详细地讨论了大规模C++程序的物理设计问题[16]。他定义了组件、包以及物理依赖（编译依赖和连接依赖）。基于此，他还提出了利用层次化方法来组织组件，并系统地总结了消除循环物理依赖和隔离类私有成员的方法。R. C. Martin展示了如何根据物理设计原则来重新组织包结构[45]。另外，B. Stroupstrup、H. Sutter、S. Meyers和S. Dewhurs等提出了一批调整编译依赖的方法，如使用PImpl惯用法[68-72]。

物理结构的质量评估包括评估标准和评估方法。评估标准主要由设计原则、编码规范和编程约定组成。它不仅提供了检查重构有效性的标准，还提供了重构的目标和理由[5, 45, 73]。而评估方法是指评价物理结构是否满足评估标准的方法。J. Lakos讨论了无环依赖原则，并定义了累积组件依赖CCD来计算子系统的可测试性。进一步地，他认为编码规范和编程约定等可用于指导包的组织[16]。与此类似，S. Mcconnell也强调了编码规范和编程约定在包结构设计中的作用[73]。而E. Evans等则认为包结构应反映领域知识[74]。此外，R. C. Martin给出了一系列包的设计原则和支持这些原则的度量方法[45]。

自动化工具可以分析系统的物理结构，从而提高质量评估的效率。一些常用的标准工具能够高效地抽取物理依赖，如gmake[75], mkmf[76]以及cdep[16]。基于J. Lakos和R. C. Martin的工作，K. Paton开发了度量工具PDCHECK，可以评估逻辑实体到物理实体的映射是否符合J. Lakos提出了的分派原则，并检查物理依赖是否符合R. C. Martin给出的物理设计原则[77]。同样地，E. Hautus等研制了度量工具OptimalAdvisor，能够检查系统的物理设计是否符合无环依赖原则和稳定抽象原则[78]。

综上所述，物理设计技术相对完整地支持了物理重构的全过程，为物理重构的研究打下了坚实的基础。在本章中，为持续优化系统的物理结构，我们全面地总结了已有的物理设计技术，并将它们应用于物理重构的合适阶段。

2.5 本章小结

就大型软件开发而言，系统的物理结构在很大程度上决定着团队的开发效率。为在软件演化过程中控制它的复杂性，我们需要一种能够持续优化物理结构的技术。然而，现有相关技术（重构和物理设计）难以有效解决这一问题。为此，我们充分研究传统重构的基本思想和物理设计的主要技术，提出了物理重构的概念，并对相应的重构过程和重构方法进行了深入讨论。可以看到，物理重构为重构和物理设计提供了全新的视角，它允许开发者从逻辑设计和物理设计两方面考虑软件质量：

- 物理重构扩展了重构的范围。与传统重构相比，物理重构着眼于物理设计，涉及编译依赖、组件和包等逻辑设计不关注的方面。因而，我们的工作是传统重构技术的有效扩展。
- 物理重构着眼于物理结构的演化。与物理设计相比，物理重构允许开发者在软件生命周期的任何阶段优化系统的物理结构。换句话说，通过物理重构，良好的物理结构不再是开发的前提条件，而是开发的结果。所以，我们的工作是现有物理设计技术的有力补充。

与传统重构类似，物理重构在本质上是一项“以人为本”的技术。物理重构中的任何决定（如选择重构方法）都需要开发者综合考虑开发的实际情况（如性能需求）、软件的各种属性（如可读性）以及软件架构等各方面的因素。本章广泛地总结了大量的相关技术，希望能为大规模软件开发提供有价值的参考。在实际开发过程中，许多简单但面向人的最佳实践往往能在物理重构中发挥重要作用，如编码风格、代码评审以及编程约定等。

第三章 面向重构的包内聚性度量方法

包结构重构是物理重构的核心内容之一。作为重要的质量指标，包内聚性可用于对存在设计缺陷的包结构实施合理的重构。为了一致而高效地计算包内聚性，研究者提出了一批包内聚性度量方法。然而，这些方法主要依赖于包内部的数据流关系，度量结果常常与包的实际内聚程度不符。为解决这一问题，本章首先根据包所承担的职责对包进行了分类。在此基础上，提出了两种基于上下文的包内聚性度量方法，并讨论了它们的适用范围。与传统方法相比，本章方法通过包所在的上下文来考察包的语义内聚程度，能够广泛适用于各种类型的包，从而良好地支持包结构的重构。

3.1 包的功能及其分类

包既是物理设计的核心元素，也是逻辑设计的重要单位。从物理设计的角度，包由组件构成，为组织代码提供更大的容器。从逻辑设计的角度，包是一组语义相关的类的集合，允许开发者在比类更高的抽象层次上对问题域进行思考。不难看出，包对于大规模软件的开发和维护具有重要意义。首先，它的质量对团队的开发效率有重要影响。其次，包结构体现了软件架构，高质量的包结构有利于提高软件的可理解性和可维护性等。最后，包是软件发布和重用的粒度，是影响软件可维护性、可重用性和可扩展性的重要因素之一[16, 45, 85]。因此，我们应该在软件开发过程中通过重构来控制包的质量。表 2.2 给出了一批可以优化包结构的重构方法（重新组织包结构）。在很多情况下，这些方法能同时改进软件的逻辑结构和物理结构。即是说，它们不仅是物理重构方法，也是传统重构方法。此外，除了表 2.2 中的重构方法外，有关包的重构方法还包括重命名包、调整包的部署等。其中，重命名包是指修改包的名称；调整包的部署是指在主机上重新部署系统中的各个包。下文重点关注包结构的重构。

包内聚性指包成员在支持一个中心目标（任务）上的紧密程度，是评估包质量的重要指标。在包结构的重构过程中，它常用于识别不良包结构和评估重构有效性。高内聚的包集中处理一个特定任务，服务于一致的抽象。这样的包复杂度低，具有更高的可理解性、可维护性、可重用性和可测试性。反之，低内聚的包由于承担过多的任务而造成代码混乱和重复。此外，内聚包的客户不会受到与它无关的包成员变化的影响。最后，内聚包对于同一种性质的变化是封闭的。也就是说，包的变化只影响该包中的类，而对该包外的类没有影响。这提高了软件的可修改性，有助于软件的演化。

由上述讨论可知，包所承担的任务是决定包内聚程度的关键性因素[45, 73, 74, 86]。为更好地探索包内聚性度量方法，我们根据包的职责将包划分为四类：工具包、接口包、实现包以及构件包。这种分类方法主要针对 C++和 Java 这样的静态语言。图 3.1 给出了这四种包的例子。其中，矩形表示包和类；有向边代表数据依赖。下面，我们结合图 3.1 对包的分类进行详细讨论：

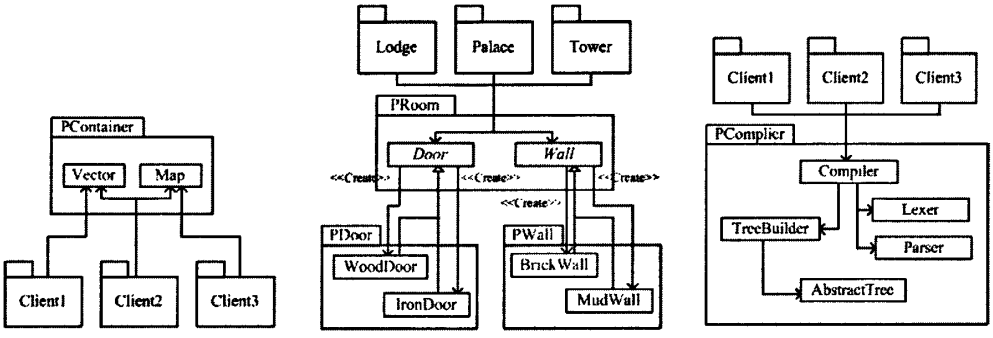
工具包旨在封装一组具备通用基础功能的类以服务于通用编程。这类包主要存在于程序库中，如 *STL*[87] 和 *Boost*[88]等。图 3.1(a)展示了一个工具包 *PContainer*，它由聚集类（*Vector* 和 *Map*）组成，可以满足软件开发对“容器”的普遍需求：开发者往往需要通过聚集类来管理一组相关对象。

接口包由一组抽象类组成，用于表达系统规约。它服从依赖倒置原则[45]，可将系统的实现细节与系统的客户隔离，从而使得实现细节可以独立于客户进行自由变化。实际上，这里的接口包与第 1 章所定义的抽象包等价，术语“接口”主要用于强调包所承担的任务。在图 3.1(b)中，包 *PRoom* 的两个抽象类 *Door* 和 *Wall* 表达了“建筑”所需的基本元素。此外，它隔离了“建筑”系统的实现细节（类 *Door* 和 *Wall* 的子类）对该系统客户（包 *Lodge*、*Palace* 和 *Tower*）的影响。

实现包实现系统特定方面的规约。在 C++这样的静态语言中，实现包通常由同一抽象包中抽象类的子类组成。如图 3.1(b)中，实现包 *PDoor* 和 *PWall* 包含了接口包 *PRoom* 的细节，分别对“建筑”系统的规约“门”和“墙”进行了实现。可以看出，实现包和接口包是一组对偶概念。

构件包对应于一个相对独立的子系统。在图 3.1(c)中，包 *PCompiler* 提供了一个用于编译的子系统，其实现采用了外观模式[67]：类 *Compiler* 是系统的“外观”，而其余类则是子系统类。

可以看出，除构件包外，其他种类的包没有或很少有的内部数据依赖，这表明包的内部数据依赖不足以表达包内部成员之间的语义关系。对于工具包、接口包和实现包而言，同一包中的类往往是相互独立的。工具包的类主要提供小而独立的功能，如图 3.1(a)中的类 *Vector* 和 *Map*。接口包的类对应于抽象的领域概念。在很多情况下，领域概念之间没有直接的数据耦合，而是通过语义相互联系。如图 3.1(b)中，抽象类 *Door* 和 *Wall* 所对应的“门”和“墙”之间不存数据依赖。实现包的类是一组相互关联的抽象类的子类，而子类之间通常不存在数据关系。同样在图 3.1(b)中，类 *WoodDoor* 和 *IronDoor* 提供了抽象类 *Door* 的两种不同实现，它们相互独立。与前三种包不同，构件包中的类常需要显式地与包中的其他类进行合作以完成一个相对完整的任务。以图 3.1(c)中的包 *PCompiler* 为例，编译器（类 *Compiler*）通过词法分析和语法分析（类 *Lexer* 和 *Parser*）来生成（类 *TreeBuilder*）抽象语法树（类 *AbstractTree*）。



(a) 工具包　　(b) 接口包和实现包　　(c) 构件包

图 3.1 包的分类

3.2 问题分析

内聚性是一种重要的设计属性，与调试、维护和修改等的代价密切相关。大量的实验研究表明内聚性可用于预测软件的外部属性，如易错性[39, 43]、可理解性[89]等。因此，高内聚一直是指导开发者识别可重构代码和评估重构效果的重要启发式规则之一。然而，人工评估效率低下，而且评估结果还因人而异。初级开发者往往由于自身经验匮乏而做出错误判断。为此，众多研究者提出了大量的模块内聚性度量方法。现有的主要成果主要集中在类上。

在包内聚性度量领域中，许多研究者对 Ada 包的内聚性度量方法进行了深入研究。S. Patel 等扩展了文档相似性概念，通过共享数据类型计算子程序的相似性。在此基础上，将包内聚性定义为子程序相似性的平均值[46-47]。L. C. Braind 等用“内部交互图”来表达设计级别的交互，并定义了一组基于该图的度量方法[48]。Z. Chen 等首先通过依赖分析讨论了包内部实体之间的依赖关系，然后利用这些关系定义了一批度量方法[49]。然而，Ada 包主要用于表达一组可被导入其他模块的逻辑声明，其作用与其他语言中（如 C++ 和 Java）的类相似。所以，它的内聚性度量方法难以体现包区别于类的特征。基于此，我们将这些方法归入类内聚性度量的范畴。

除针对 Ada 包的方法外，R. C. Martin 提出了关系内聚性（Relational Cohesiveness，RC）[45]。RC 度量被定义为包内部数据依赖的数目与包中类的数目之比。不过，RC 度量的最大值会随着包中类的数目的变化而变化，所以它不满足“最大值和最小值”属性[90-91]。E. B. Allen 等提出了基于信息论的模块内聚性度量方法（Excess Entropy Cohesiveness，EEC）[26]。他们首先给出了“模块内交互图”，然后将该图对应的关联矩阵看作信源，将关联矩阵的行看作信源发出的符号，最后将模块内聚性定义为余熵。由于 EEC 度量只需要模块内交互图，因而可以不加修改地应用于包。

然而，RC 度量和 EEC 度量均基于包的内部数据依赖，难以适应于工具包、接口包和实现包。由 3.1 节可知，除构件包外，其他种类的包通常只含有少量的内部数据依赖。考虑图 3.1，尽管包 *PContainer*、*PRoom*、*PDoor* 和 *PWall* 在语义上是内聚的，但是它们没有内部数据依赖，所以它们的 RC 值与 EEC 值均为 0。事实上，这两种方法只对包 *PCompiler* 有意义。

本质上，包的内聚程度由包的语义决定。现有方法在理解包的语义上存在困难，只能通过包内部数据依赖来推测包是否服务于一个中心目标。它们认为紧密的数据耦合意味着较高的内聚性；反之，松散的数据耦合代表着较低的内聚性。然而，紧密的数据耦合主要表明包中的类参与了共同计算，而非共同任务，这导致现有方法在很多情况下失效。因此，包的内部数据依赖对于计算包内聚性是不充分的。

我们认为包上下文能够较好地反映包的语义。实际上，开发者总是通过包上下文来设计和理解包。所谓包上下文是指包所处的“环境”，即它与系统其余部分的关系。这里，术语“关系”主要代表数据依赖。基于此，我们提出了两种考虑包间数据依赖的包内聚性度量方法：基于客户使用的包内聚性度量和基于相似上下文的包内聚性度量。下文将这两种方法统称为基于上下文的包内聚性度量方法。

类间数据依赖是构成包内和包间数据依赖的基础。为在下文中准确地定义包内聚性度量方法，我们首先给出类间数据依赖的形式化定义。一般地，类有两种基本的数据依赖：继承和使用。类 c_1 继承依赖于类 c_2，若 c_2 是 c_1 的祖先，记为 $c_1 \xrightarrow{i} c_2$。类 c_1 使用依赖于类 c_2，若实现于 c_1 中方法引用了实现于 c_2 中的方法、属性或类型，记为 $c_1 \xrightarrow{u} c_2$。这里，我们将继承自父类的方法看作其他类的方法。此外，由于动态绑定的存在，我们往往难以在静态时决议一个多态调用在运行时所调用的具体方法。为提高分析效率，我们采用“保守”策略，即考虑所有可能被调用的方法。如图 3.1(b)中，类 *WoodDoor* 和 *IronDoor* 继承依赖于类 *Door*；类 *Door* 使用依赖于类 *WoodDoor* 和 *IronDoor*。

定义 3.1 给定类 c_1 和 c_2，若 c_1 继承依赖于或使用依赖于 c_2，则称 c_1 数据依赖于 c_2，记为 $c_1 \xrightarrow{d} c_2$。若(c_1, c_2)属于数据依赖 $\xrightarrow{d}$ 的传递闭包，则称 c_1 间接数据依赖于 c_2，记为 $\xrightarrow{d+}$。

下面分别讨论基于客户使用的包内聚性度量和基于相似上下文的包内聚性度量。

3.3 基于客户使用的包内聚性度量

作为包上下文的关键部分，客户对包的使用方式能够较好地表达包的语义。在网站设计领域，客户的浏览行为是评估网站质量的重要因素，如网站的可导航性等[92]。类似地，通过客户使用来考察包内聚性具备较高的合理性。本节首先提出了一种新的包内聚性：共同重用内聚性 CRC，然后根据 3.1 节所提出的包分类对共同重用内聚性的适用情况进行了讨论，最后定义了 CRC 内聚性的计算方法 HC 度量。与现有方法相比，HC 度量同时考虑了包内和包间数据依赖。

3.3.1 包内聚性与客户使用

包内聚性可以通过客户对包的使用方式来计算。共同重用原则规定内聚包中的类应被其客户共同重用，即是说如果重用了包中的一个类，那么就应该重用该包中的所有类[45]。从逻辑设计角度，共同重用原则说明总是被共同重用的类之间存在紧密的语义联系。通常，类很少会孤立重用。为完成一个任务，它需要与作为该可重用抽象的其他类协作，如图 3.1(b)的类 *Door* 和 *Wall*。因此，内聚包的类总是向包的客户展现一致的抽象。从物理设计角度，共同重用原则能避免不必要的重新发布和重新验证。当一个包使用了另一个包时，它们之间会存在依赖关系。即使仅使用了一个类，这种关系也不会削弱。在这种情况下，被依赖包的任何修改，甚至这种修改来源于一个客户根本不关心的类，都会导致客户的重新发布和重新编译。

根据共同重用原则，若包的所有类被它的所有客户共同重用，则该包具有最高内聚性；反之，若没有类被共同重用，则该包具有最低内聚性。基于此，我们定义了一种新的包内聚性。

定义 3.2 包的共同重用内聚性（Common Reuse Cohesion，CRC）反映包的类被共同重用的程度：被客户共同重用的类越多，内聚程度越高；反之，被客户共同重用的类越少，内聚程度越低。

例如在图 3.1(c)中，包 *PCompiler* 中的所有类（类 *Door* 和 *Wall*）均被它的客户（包 *PCompiler*）共同重

用，所以它的内聚程度较高。

CRC 内聚性具有以下优点。首先，CRC 内聚性反映了包的修改和包的客户之间的关系。通常，对 CRC 内聚性较高的包而言，其修改总是与它的客户相关，这有效保证了包的客户不受无关修改的干扰。相反地，对 CRC 内聚性较低的包而言，其客户总是被无关修改所影响。此外，CRC 内聚性从客户的角度来看待类之间的耦合关系，能有效反映包的语义内聚性，因而其适用性得到了较大的提高。下面，我们根据 3.1 节所提出的包分类详细地讨论 CRC 内聚性的适用情况，即 CRC 内聚性是否与语义内聚性一致：

- 对于工具包，CRC 内聚性可能不是非常有效。一般而言，开发者会根据自身需求仅使用工具包中的部分类。以图 3.1(a)为例，在大多数情况下，类 *Vector* 和 *Map* 未被共同重用，因此工具包 *PContainer* 的 CRC 内聚性较低。但在概念上，这两个类均是对象的聚集，具有较高的语义相关性。我们不难推断其他工具包（如 *STL*）的 CRC 内聚性也不高。然而，基于客户行为审查包内聚性的思想具备较高的合理性，符合开发者理解和设计包的方式。因而，在未来工作中，我们将进一步研究 CRC 内聚性和工具包的关系。
- 对于接口包，CRC 内聚的适用性需视情况而定。对于存在于库中的接口包，CRC 内聚可能不适用。其原因与工具包有类似。对于存在于应用程序中的接口包，CRC 内聚性往往能反映包的语义内聚性。例如，在图 3.1(b)中，接口包 *PRoom* 具备较高的 CRC 内聚。
- 实现包的适用情况与接口包相同。即是说，CRC 内聚可能不适用于库中的实现包，但却能在应用程序上良好地工作。这是因为实现包是与它对应的接口包的扩展。由于采取保守策略，使用一个抽象类通常也会使用它的子类。以图 3.1(b)为例，包 *Lodge*、*Palace* 和 *Tower* 不仅使用了抽象类 *Door* 和 *Wall*，也使用了这些抽象类的子类 *WoodDoor*、*IronDoor*、*BrickWall* 和 *MudWall*。
- CRC 内聚广泛地适用于构件包。一般地，对于设计良好的构件包，其客户会直接或间接地使用它的所有类。考虑图 3.1(c)中的包 *PCompiler*，类 *Compiler* 被直接重用；类 *Lexer*、*Parser*、*TreeBuilder* 和 *AbstractTree* 被间接重用。

综上所述，就应用程序而言，CRC 内聚与语义内聚基本一致，所以它能有效地评估应用程序的质量。然而，库中的包有待于我们进一步地研究。在未来工作中，我们将选择几个流行的库，并从开源项目中收集这些库的客户，以考察 CRC 内聚与库的关系。考虑到大多数开发者从事应用程序的开发，本文将注意力集中于如何根据 CRC 内聚来评估应用程序中包的质量。

3.3.2 度量定义

由上述讨论可知，CRC 度量需要通过客户对包的使用方式来计算：若大多数客户总是使用包中的大多数类，则我们认为该包的 CRC 内聚较高；反之，若大多数客户仅使用包中的部分类，则我们认为该包的 CRC 内聚较低。为了准确地给出计算 CRC 内聚的公式，我们首先对包客户进行了形式化定义。

定义 3.3 给定包 p_1 和 p_2，如果∃类 $c_1 \in p_1$，类 $c_2 \in p_2$，满足 $c_1 \xrightarrow{*} c_2$，则 p_1 是 p_2 的客户。给定包 p，它的客户集 ClientPackage(p) = {p_i | 包 p_i 是包 p 的客户}。

由定义 3.3 可知，我们从接口包的客户集中排除了实现包。这是由于对于接口包，其对应的实现包的功能是实现接口包所描述的系统规约，而不是使用接口包所提供的服务。因而，我们不将这种包作为接口包的客户。以图 3.1(b)为例，ClientPackage(*PRoom*) = {*Lodge*, *Palace*, *Tower*}。

计算 CRC 内聚的关键在于考察客户对包的使用方式。在很多领域中，向量是表达方式（模式）的一个有效工具。基于此，我们同样利用向量来表达包的客户对包的使用方式。

定义 3.4 给定包 $p_1 = \{c_1, c_2, \ldots, c_m\}$ 以及 $p_2 \in$ ClientPackage(p)，p_2 对 p_1 的使用方式是一个向量($v_1, v_2, \ldots, v_m$)，记为 UsePattern(p_1, p_2)。其中，分量 v_i 对应于类 c_i，其值由公式 3.1 决定：

$$v_i = \begin{cases} 1 & if \ \exists c \in p_2 \wedge c \xrightarrow{d+} c_i \\ 0 & if \ \neg\exists c \in p_2 \wedge c \xrightarrow{d+} c_i \end{cases} \tag{3.1}$$

根据定义 3.4，$v_i = 1$ 表示 p_2 使用了 c_i；反之，$v_i = 0$ 代表 p_2 未使用 c_i。可以看出，UsePattern(p_1, p_2)有

效表达了 p_2 对 p_1 的使用方式。特别地，若 UsePattern(p_1, p_2)等于(1,..., 1)，则 p_2 使用了 p_1 中的所有类。鉴于此，我们将(1,..., 1)称为 p_1 的最好方式，记为 BestPattern(p_1)。

例如，对图 3.1(b)，

UsePattern(*PRoom*, *Lodge*) = UsePattern(*PRoom*, *Palace*)
= UsePattern(*PRoom*, *Tower*)
= BestPattern(*PRoom*) = (1, 1)

根据 CRC 度量，若包 p 是内聚的，则它的客户应以相似的方式使用 p。进一步地，这种相似的方式即为 p 的最好方式。因此，我们认为内聚包的客户应满足性质 3.1。

性质 3.1 给定包 p 以及 $p_i \in$ ClientPackage(p)，若 p 的 CRC 内聚较高，则 UsePattern(p, p_i)与 BestPattern(p)之间的相似性较高。

在性质 3.1 中，使用方式之间的相似性可以通过任意向量相似性度量方法进行计算。考虑到使用方式的分量属于{0, 1}，我们采用海明距离来计算两个使用方式之间的相似性。给定向量 V_1 和 V_2，它们之间的海明距离等于它们对应分量取不同值的数目[93]，记为 HamDist(V_1, V_2)。在此基础上，V_1 和 V_2 之间的相似性计算方法如公式 3.2 所示：

$$HamSim(V_1,V_2)=\frac{m-HamDist(V_1,V_2)}{m} \tag{3.2}$$

其中，m = $|V_1|$。

例如，令 V_1 = (0, 1)、V_2 = (1, 1)，则 HamDist(V_1, V_2) = 1，HamSim(V_1, V_2) = 1/2。

由性质 3.1 以及公式 3.2 可知，对于内聚包 p 和它的客户 p_i，HamSim(UsePattern(p, p_i), BestPattern(p))应接近 1。基于此，我们定义海明内聚性（Hamming Cohesiveness，HC）为最好方式与客户使用方式之间相似性的平均值，其计算公式如定义 3.5 所示。

定义 3.5 给定包 p，令 m = $|p|$、n = |ClientPackage(p)|，有

$$HC(p)=\begin{cases}\dfrac{\dfrac{m}{n}\displaystyle\sum_{p_i\in ClientPackage(p)} HamSim(UsePattern(p,p_i),BestPattern(p))-1}{m-1} & if\ \ m>1\\ 1 & if\ \ m=1\end{cases} \tag{3.3}$$

当 m = 1 时，p 仅包含一个类。在这种情况下，p 显然具有 CRC 内聚，所以我们设 HC(p)为 1。当 m > 1 时，若 p 的每个客户仅使用它的一个类，则 HC(p) = 0；若 p 的所有类均被它的客户共同重用，则 HC(p) = 1。因此，HC(p) ∈ [0, 1]。

下面以图 3.1(b)为例展示 HC 度量的计算过程。对于包 *PRoom* 以及 $p_i \in$ ClientPackage(*PRoom*)，有

m = |PRoom| = 2
n = |ClientPackage(PRoom)| = 3
HamSim(UsePattern(PRoom, p_i), BestPattern(PRoom)) = 1

则根据定义 3.5，HC(PRoom) = 1。

3.3.3 基本性质

根据上述讨论，我们可以得出 HC 度量具备以下性质：

第一，HC 度量从客户使用的角度计算包内聚性，同时考虑了包内数据依赖和包间数据依赖。通过包间数据依赖，HC 度量能够利用客户行为来挖掘类间的语义耦合。即是说，趋向于共同重用的类具有较高的语义相关性。这有助于我们从更高抽象层次上考察包的内聚程度。事实上，趋向于共同重用的类总是服务于一致的抽象，不管它们之间是否存在紧密的数据耦合。如图 3.1(b)，虽然类 *WoodDoor* 和 *IronDoor* 之间没有数据依赖，但是它们总是被共同重用。因此，它们属于相同的可重用抽象，具有紧密的语义联系。另外，包内数据依赖也对计算包的内聚性有较大贡献，它能帮助我们考虑包的内部类。所谓内部类是指不

被客户直接使用的类。例如，在图 3.1(c)中，类 *Lexer*、*Paser*、*TreeBuilder* 和 *AbstractTree* 是内部类，它们通过包内部数据依赖对 HC 值产生影响。

第二，充足客户集可以使得 HC 度量更加有效。对于包而言，充足客户集能够全面地反映客户使用包的实际情况，从而保证度量值的稳定性。由上可知，HC 度量的核心思想是利用客户行为来推测包的语义。由于全面地考察使用场景可以充分挖掘包的语义信息，所以充足客户集有助于 HC 度量较好地理解包所承担的任务。在这种情况下，HC 值不会随着新客户的加入而发生较大改变，而稳定的度量值足够辅助开发者判断包的内聚程度。因而，我们应使用充足的客户集来计算 HC 值。

在实际度量中，客户集是否充足应根据包被重用的情况来判断。对于仅用于一个项目的包而言，术语“充足”表示包的所有客户都应被考虑。同样地，对于在有限范围内重用的包，例如一个公司的项目，我们也应收集包的所有客户。然而，对于大范围重用的包，其客户总是分散在数量庞大的应用程序中，这使得收集所有客户变得十分困难，甚至不可行。典型的例子是程序库中的包。在这种情况下，充足客户集应涵盖所有典型的使用场景。进一步地，在充足客户集中，每种客户所占的比例应与它对应的场使用景的出现频率一致。以图 3.1(a)的包 *PContainer* 为例，若大约 1/3 的客户具有使用方式(1, 1)，则在用于计算 HC(*PContainer*)的客户集中，使用方式为(1, 1)的客户所占比例不能过多地偏离 1/3。

第三，HC 度量利用向量来表达包客户对包的使用方式，这使得它的可区分性高于基于计数的度量方法。所谓基于计数的度量方法是指仅通过满足某种条件的类的个数来定义包内聚性的度量方法，如 RC。在图 3.2 中，若使用被所有客户重用的类的数目来计算 CRC 内聚性，则 P_1 和 P_2 的度量结果均为 0，这与我们的开发经验相悖。然而，HC 度量能准确区分这两个包的内聚程度：$HC(P_1)=0.50$，$HC(P_2)=0.25$。

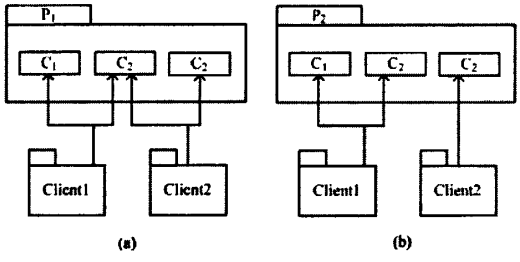


图 3.2 基于计数方法不能区分的包

第四，HC 度量具备 L. C. Briand 等所提出的良好模块内聚性度量应满足的性质：(1) 非负性及标准化；(2) 最大值和最小值；(3) 单调性；(4) 若两个模块之间不存在关系，则它们合并之后的度量值不高于它们合并之前度量值的最大值[90-91]。根据 3.3.2 节的讨论，HC 值∈[0, 1]，显然满足前两个性质。下面详细讨论 HC 度量是否满足其余两个性质。

定理 3.1 给定包 p_1，类 $c_1 \in p_1$ 以及类 c_2，若 c_2 与 p_1 之间存在关系，则增加 $c_2 \xrightarrow{d} c_1$ 后形成的新包 p_1' 应满足 $HC(p_1) \le HC(p_1')$。

证明：在 HC 度量的上下文中，c_2 与 p_1 之间存在关系，当且仅当它们满足

$$c_2 \in p_1 \vee (c_2 \in p_2 \wedge p_2 \in \text{ClientPackage}(p_1))$$

因此，由定理条件可得，

若 $c_2 \in p_2$，$p_2 \in \text{ClientPackage}(p_1)$，则对于 $\forall c \in p_1$，$c_1 \xrightarrow{d+} c$，满足 $c_2 \xrightarrow{d+} c$。根据定义 3.4，

$$\text{UsePattern}(p_1, p_2) \mid \text{UsePattern}(p_1', p_2') \ge \text{UsePattern}(p_1, p_2) \tag{3.4}$$

其中，p_2' 代表 p_2 对应的新包；符号|表示按分量与。

公式 3.4 表明增加数据依赖会增加使用方式中值为 1 的分量数。由公式 3.2，

$$\text{HamSim}(\text{UsePattern}(p_1, p_2), \text{BestPattern}(p_1)) \le \text{HamSim}(\text{UsePattern}(p_1', p_2'), \text{BestPattern}(p_1'))$$

此外，对于包 $p_3 \in \text{ClientPackage}(p_1)$，$p_3 \ne p_2$，有

$$\text{HamSim}(\text{UsePattern}(p_1, p_3), \text{BestPattern}(p_1)) = \text{HamSim}(\text{UsePattern}(p_1', p_3), \text{BestPattern}(p_1'))$$

因而，$HC(p_1) \le HC(p_1')$。

若 $c_2 \in p_1$，则对于 $\forall c_3 \in p_2$，$p_2 \in \text{ClientPackage}(p_1)$，满足 $c_3 \xrightarrow{d} c_2$，有 $c_3 \xrightarrow{d+} c_1$。这等价于直接增加 $c_3 \xrightarrow{d} c_1$，从而转化为第一种情况。因此，$HC(p_1) \le HC(p_1')$。

综上，增加 $c_2 \xrightarrow{d} c_1$，$HC(p_1) \le HC(p_1')$。 ∎

定理 3.2 给定包 p_1 和 p_2，令 $p_3 = p_1 \cup p_2$，若 p_1 和 p_2 之间没有关系，则 $HC(p_3) \le \text{Max}(HC(p_1), HC(p_2))$。

证明：在 HC 度量的上下文中，p_1 和 p_2 之间不存在关系，当且仅当它们满足

$$\neg\exists c_1 \in p_1 \wedge \neg\exists c_2 \in p_2 : c_1 \xrightarrow{d} c_2 \vee c_2 \xrightarrow{d} c_1$$

和

$$\text{ClientPackage}(p_1) \cap \text{ClientPackage}(p_2) = \varnothing$$

因此，对于 $\forall p_i \in \text{ClientPackage}(p_1)$，$c \in p_3$，若 p_i 使用 c，则 $c \in p_1$。对于 p_2，我们有相同的结论。此外，p_3 的类的数目等于 p_1 和 p_2 的类的数目之和。令 $m_1 = |p_1|$，$m_2 = |p_2|$，$n_1 = |\text{ClientPackage}(p_1)|$以及 $n_2 = |\text{ClientPackage}(p_2)|$，有

若 $m_1 = 1$ 或 $m_2 = 1$，则 $HC(p_1)$或 $HC(p_2)$等于 1。由于 $HC(p_3) \le 1$，所以 $HC(p_3) \le \text{Max}(HC(p_1), HC(p_2))$。

若 $m_1 \ne 1$ 且 $m_2 \ne 1$，则

$$HC(p_1) = \frac{\frac{m_1}{n_1} \sum_{p_i \in ClientPackage(p_1)} Similarity(UsePattern(p_1, p_i), BestPattern(p_1)) - 1}{m_1 - 1}$$

$$HC(p_2) = \frac{\frac{m_2}{n_2} \sum_{p_i \in ClientPackage(p_2)} Similarity(UsePattern(p_2, p_i), BestPattern(p_2)) - 1}{m_2 - 1}$$

对于包 $p \in \text{ClientPackage}(p_3)$，有$|\text{UsePattern}(p_3, p)| = m_1 + m_2$。又由于 p_1 和 p_2 无关，所以 $\text{UsePattern}(p_3, p)$ 由公式 3.5 决定。

$$UsePattern(p_3, p) = \begin{cases} (UsePattern(p_1, p), 0, \ldots, 0) & if \quad p \in ClientPackage(p_1) \\ (0, \ldots, 0, UsePattern(p_2, p)) & if \quad p \in ClientPackage(p_2) \end{cases} \tag{3.5}$$

因此，

$$HC(p_3) = \frac{\frac{1}{n_1 + n_2}((HC(p_1) \times (m_1 - 1) + 1) \times n_1 + (HC(p_2) \times (m_2 - 1) + 1) \times n_2) - 1}{m_1 + m_2 - 1}$$

不失一般性，设 $HC(p_1) \le HC(p_2)$，$m_1 \le m_2$，有

$$HC(p_3) - HC(p_2) \le \frac{HC(p_2) \times (m_2 - 1)}{m_1 + m_2 - 1} - HC(p_2) = \frac{-HC(p_2) \times m_1}{m_1 + m_2 - 1} \le 0$$

因此，$HC(p_3) \le \text{Max}(HC(p_1), HC(p_2))$。

综上，若两个包之间不存在关系，则它们合并之后的 HC 值不高于它们合并之前 HC 值的最大值。 ∎

3.3.4 实例分析

本节将通过两个实例来对比 RC 度量、EEC 度量和 HC 度量，以研究各种方法的效果和适用性。第一个实例是图 3.1 中的包。该实例虽然简单，但却涵盖了所有种类的包。更重要地，其简单性排除了大规模系统的实现细节，使我们将注意力集中在关键性问题上。第二个实例是演化测试框架（Evolutionary Testing Framework，ETF）中的演化计算模块 *Evolution*[94-96]。*ETF* 为演化测试研究提供了一个可配置、可扩展的实验平台，可自动生成覆盖指定语句或程序路径的测试用例。对 *ETF* 而言，*Evolution* 是一个相对独立的模块，实现了一个基

于位串编码和实数编码的遗传算法，它包含 71 个类和 15 个包。图 3.3 展示了 *Evolution* 的主要包结构和一些重要包的内部细节，如包 *Visitor*。下文将详细讨论 *Evolution* 中包的内聚性。与第一个实例相比，*Evolution* 保留了真实软件的主要特征，与产品代码更加接近，这有助于我们观察 HC 度量在真实世界中的效果。

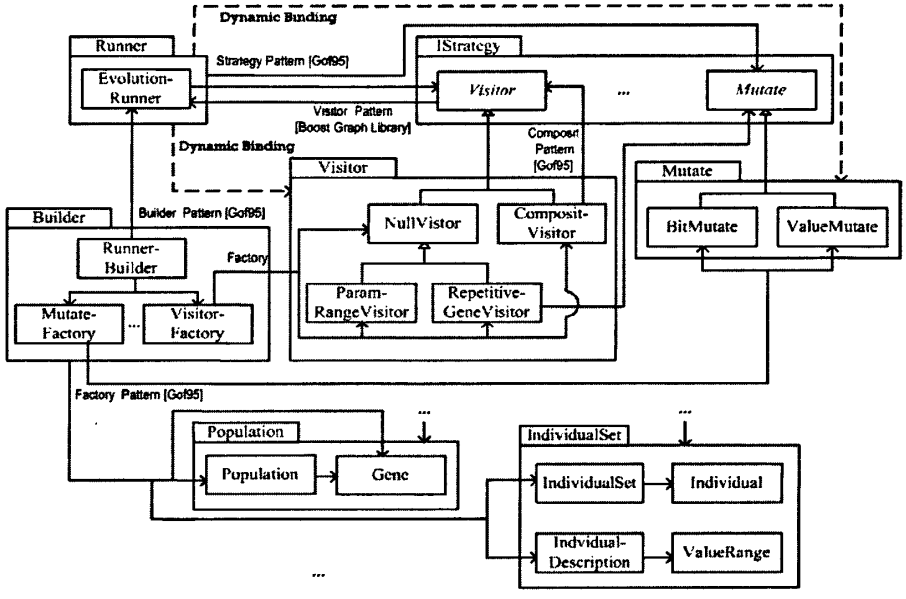


图 3.3 Evolution 的主要包结构

在实例分析中，我们收集了充足的客户集来计算两个实例的 HC 值，以保证度量结果的稳定性和有效性。表 3.1 和表 3.2 分别给出了图 3.1 和 *Evolution* 中包的 RC 值、EEC 值以及 HC 度量值。图 3.4~3.6 分别对比了工具包、接口包以及实现包的 RC 值、EEC 值以及 HC 值。由于两个实例仅包含一个工具包，我们没有给出工具包的对比图。虽然仅研究了两个实例，但它们的复杂性足以帮助我们获得一批有意义的结论：

表 3.1 图 3.1 的度量结果

包名	RC	EEC	HC	类别
PContainer	0.00	0.00	0.33	工具包
PRoom	0.00	0.00	1.00	接口包
PDoor	0.00	0.00	1.00	实现包
PWall	0.00	0.00	1.00	
PCompiler	1.00	0.46	1.00	构件包

(1) 对于工具包，RC 度量和 EEC 度量的结果常常与开发经验相悖，而 HC 度量可能也不是十分有效。

由 3.1 的分析可知，包 *PContainer* 是内聚的。然而在表 3.1 中，RC(*PContainer*) = EEC(*PContainer*) = 0.00。这是由于工具包的内部数据依赖通常较少，所以基于内部数据流关系的 RC 度量和 EEC 度量不适用于这种包。此外，HC(*PContainer*)只有 0.33，表明 HC 度量对于工具包可能不是十分有效。事实上，开发者在大多数情况下仅使用工具包中的部分类。例如，我们很少在一个项目中使用 *STL* 的所有容器。因此，工具包的 HC 值往往较低。在实际度量中，我们不能将 HC 值接近 1 作为工具包内聚的标准，而合适的判断标准有待于我们进一步研究。

(2) 对于接口包，RC 度量和 EEC 度量可能不适用，但 HC 度量却有良好表现。

在两个实例中，除包 *IStrategy* 外，其余接口包都有较高的内聚性。根据 3.1 节，包 *PRoom* 是内聚的。

表 3.2 Evolution 的度量结果

包名	RC	EEC	HC	描述	类别
IStrategy	0.00	0.00	0.57	封装了抽象的演化策略	接口包
ICommand	0.00	0.00	0.70	封装了控制演过程的抽象策略	
Coder	0.00	0.00	1.00	具体演化策略	实现包
Mutate	0.00	0.00	1.00		
Crossover	0.00	0.00	1.00		
Survive	0.00	0.00	1.00		
GeneSelector	0.00	0.00	1.00		
IndividualSetGener	0.00	0.00	1.00		
Visitor	0.75	0.23	1.00		
FitnessFuntor	0.00	0.00	1.00	控制演化过程的具体策略	
Evaluator	0.00	0.00	1.00		
Population	1.00	1.00	0.88	对演化计算中的种群进行建模	构件包
IndividualSet	0.50	0.33	0.74	对问题域的个体（解）进行建模	
Builder	1.00	0.23	1.00	创建演化计算控制器	
Runner	1.00	1.00	1.00	演化计算控制器	

对于 *Evolution*，包 *ICommand* 封装了控制演化过程的策略，服务于一致的抽象，也具有较高的语义内聚性。

然而在图 3.4 中，所有接口包的 RC 值和 EEC 值均为 0，说明这两种度量不能用于计算接口包的内聚性。事实上，构成接口包的抽象类往往表达了相对独立的领域概念，所以通常这样的类之间不存在数据依赖。即是说，接口包的内部数据依赖较少。

图 3.4 还表明客户使用方式能够有效地表达抽象类之间的语义联系。可以看到在图 3.4 中，接口包的 HC 值远远高于它们的 RC 值和 EEC 值。另外，我们还注意到在表 3.2 中，HC(*IStategy*) = 0.57，这说明包 *IStrategy* 的内聚程度不高。从语义上看，包 *IStrategy* 既包含了演化计算的策略，如变异 *Mutate*、编码 *Coder* 等，还包含了用于监控演化流程以收集研究数据的访问者 *Visitor*，而后者不属于演化计算领域，因此它的内聚性应低于包 *PRoom* 和 *ICommand*。从实现上看，包 *Visitor* 不仅是包 *IStrategy* 的扩展，也是它的客户。在图 3.3 中，包 *Visitor* 中的类 *CompositeVisitor* 和 *RepetiveGeneVisitor* 分别使用依赖于包 *IStrategy* 中的抽象类 *Visitor* 和 *Mutate*，所以 UsePattern(*IStrategy*, *Visitor*) = (1, 1, 0, 0, 0, 0, 0, 0)。这个使用方式极大地降低了 HC(*IStrategy*)。综上所述，HC 度量能反映包的实际内聚程度。

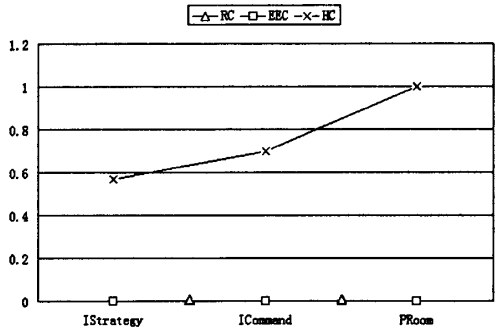


图 3.4 接口包的度量结果

(3) HC 度量能有效适用于实现包，但 RC 度量和 EEC 度量常常失效。

在图 3.1 和 *Evolution* 中，所有的实现包均由继承于同一抽象类的具体类组成，具有较高的内聚性。例

如在图 3.3 中，包 *Mutate* 封装了抽象类 *Mutate* 的子类，代表了演化计算中具体的变异策略。因此，从语义上看，它是内聚的。

在图 3.5 中除包 *Visitor* 外，所有实现包的 RC 值和 EEC 值均为 0，从而说明传统的内聚性度量方法往往难以适用于实现包。这是因为同一抽象类的不同子类表达了该抽象类的不同实现，相互之间一般不存在数据依赖关系。例如在图 3.1(b)中，类 *WoodDoor* 和 *IronDoor* 分别是“木门”和“铁门”，而实现这两种“门”不需要相互依赖。

此外，我们注意到所有实现包的 HC 值均为 1，这表示 HC 度量总是能适用于实现包。在我们的实例中，所有的子类均被共同重用。由图 3.1(b)和图 3.3 可知，实现包有两类客户。第一类是如包 *EvolutionBuilder* 这样的 factory/builder 包。为了能够创建每种类的对象，这种包显然依赖于所有子类的实现。第二类是与实现包对应的接口包的客户。由于我们采用保守方法来分析数据依赖关系，即若类 *c* 调用抽象类 *ac* 的虚函数 *vf*，则我们认为 *c* 也调用了 *ac* 的所有子类的 *vf*，所以这种客户也数据依赖于实现包的所有类。例如，类 *EvolutionRunner* 数据依赖于抽象类 *Mutate* 的所有子类 *ValueMutate* 和 *BitMutate*。

(4) 三种度量方法均能适用于构件包，但 HC 度量更加有效。

在我们的实例中，所有的构件包都具有较高的内聚性。由 3.1 节可知，包 *PCompiler* 是内聚的。对于 *Evolution*，包 *Population* 中的类代表了演化计算中被演化的个体。包 *IndividualSet* 对概念“解”进行了封装：类 *Individual* 和 *IndividualSet* 分别对应于问题的解和解集，而类 *IndividualDescription* 和 *ValueRange* 则描述了解应具有的特征。包 *Builder* 的功能是创建类 *EvolutionRunner* 的实例。以上这些包中的类均服务于一个中心目标，所以在语义上是相互联系的。此外，包 *Runner* 仅包含一个类 *EvolutionRunner*，显然具有较高内聚性。

由图 3.6 可知，构件包的 RC 值都显著地大于 0，从而表明 RC 度量能够适用于构件包。然而，RC 既缺乏最大值，也没有辅助开发者判断包是否内聚的标准，这在很大程度上降低了它的可用性。为解决这一问题，我们需要进行大规模实验，以研究 RC 度量的评估标准，即 RC 的阈值。

图 3.6 还表明 EEC 对构件包有效。在图 3.6 中，EEC 的平均值是 0.61，远大于 0。然而，从表 3.1~3.2 可得，EEC(*PComplier*)，EEC(*IndividualSet*)和 EEC(*Builder*)分别为 0.46、0.23 和 0.33，远远偏离最大值 1。这是因为 EEC 的评估标准是完全图：模块 *m* 具有最高内聚性，当且仅当 *m* 对应的模块内交互图是完全图。但在通常情况下，包内数据交互图与完全图之间存在较大距离。例如，包 *IndividualSet* 只有两条数据依赖，而对应的完全图有 6 条数据依赖。因此，我们不能用 EEC 值为 1 作为评估构件包是否内聚的标准。进一步的结论依赖于更多的大规模实验的支持。

最后对于构件包，HC 度量往往比 RC 度量和 EEC 度量有效。首先在图 3.6 中，除包 *Population* 外，HC 值均高于 RC 值和 EEC 值。虽然根据表 3.2，HC(*Population*) = 0.88，比相应的 RC 值和 EEC 值低，但已经足够我们判断包 *Population* 的内聚程度。度量的核心是辅助开发者作出正确的设计决策，而不是给出一组数值。所以在我们的实例中，0.88 与 1 同样有效。其次与 RC 度量相比，HC 度量有最大值。最后，HC 度量能在 EEC 值较低的包上良好地工作，如包 *PComplier*、*IndividualSet* 和 *Builder*。

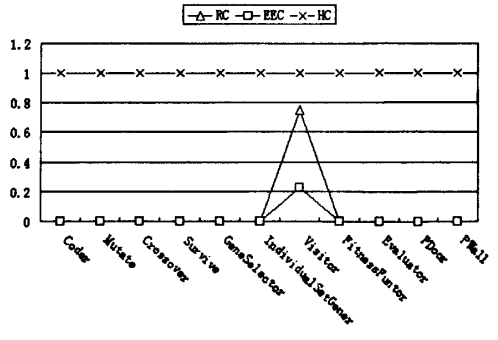


图 3.5 实现包的度量结果

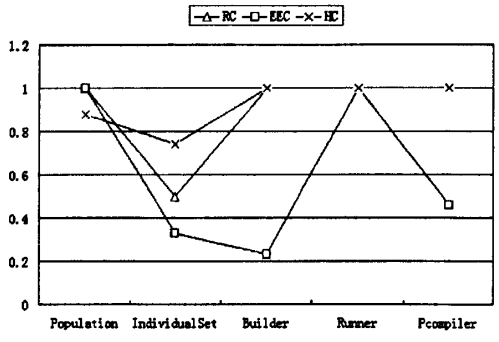


图 3.6 构件包的度量结果

综上所述，与其他包内聚性度量方法相比，HC 度量能较好地适用于应用程序，这说明 CRC 内聚从客户行为挖掘包语义的基本思想具有较高的可行性。第一，未考虑包间数据依赖的度量方法通常只适用于构件包，如 RC 度量和 EEC 度量。其根本原因是包的高层语义信息与它的低层数据流之间不一致，而这种不一致性是包内聚性度量领域的本质困难。第二，HC 度量能够通过客户使用从很大程度上获取类之间的语义关系，所以它的适用范围扩展到接口包、实现包和构件包。而对于工具包，我们还需要进一步研究。然而，我们相信 CRC 内聚的核心思想是合理的，因此我们可能需要更多以 CRC 内聚为基础的度量方法来评估工具包的内聚性，从而获得与开发经验和设计原则等一致的度量结果。

3.4 基于相似上下文的包内聚性度量

基于客户使用的包内聚性度量主要从客户角度计算包内聚性。然而，包上下文不仅包括客户，还包括包所使用的服务。为全面利用上下文信息，本节提出了另一种基于上下文的包内聚性度量方法：基于相似上下文的包内聚性度量方法 SCC。其基本思想是两个类的上下文相似程度较高，则它们的耦合性较高；反之，它们的上下文相似程度较低，则它们的耦合性较低。由于包上下文由类上下文构成，因此，SCC 度量在考察类上下文的同时，也考察了包上下文。

3.4.1 类上下文

我们认为类 c 的上下文由与 c 之间存在数据依赖关系的类组成，可分为两部分：一部分是被 c 数据依赖的类；另一部分是数据依赖于 c 的类。

定义 3.6 对于类 c，有

波动集 $S_R(c) = \{c_i \mid c_i \xrightarrow{d+} c\}$

依赖集 $S_D(c) = \{c_i \mid c \xrightarrow{d+} c_i\}$

在定义 3.6 中，$S_R(c)$是对 c 有数据依赖的类集合，代表受 c 影响的类。即是说，若 c 发生变化，该集合中的类也会发生相应的变化。$S_D(c)$包含了被 c 数据依赖的类，表示影响 c 的类。当 $S_D(c)$中的类被修改时，c 也会进行相应的调整。基于此，类的上下文被定义为由 S_R 和 S_D 构成的元组。

定义 3.7 给定类 c，其上下文是$(S_R(c), S_D(c))$，记为 $CC(c)$。

基于定义 3.7，我们可以作出如下合理的推断：若类 c_1 和 c_2 的上下文相似，则它们紧密相关。由于类上下文由 S_R 和 S_D 组成，因而计算 $CC(c_1)$和 $CC(c_2)$的相似性需要同时考虑 $S_R(c_1)$和 $S_R(c_2)$的相似性以及 $S_D(c_1)$和 $S_D(c_2)$的相似性，分别记为 $RSS(c_1, c_2)$和 $DSS(c_1, c_2)$。给定集合 S_1 和 S_2，我们将它们的相似性定义为相同元素的个数与元素总数之比，其形式化定义由公式 3.6 给出：

$$Similarity(S_1,S_2)=\begin{cases}\dfrac{|S_1\cap S_2|}{|S_1\cup S_2|} & if\ \ S_1\cup S_2\neq\varnothing\\ 0 & if\ \ S_1\cup S_2=\varnothing\end{cases} \tag{3.6}$$

当 $RSS(c_1, c_2) = 0$ 且 $DSS(c_1, c_2) = 0$ 时，$CCS(c_1, c_2)$应等于 0；当 $RSS(c_1, c_2) = 1$ 且 $DSS(c_1, c_2) = 1$ 时，$CCS(c_1, c_2)$应达最大值 1。因此，

$$CCS(c_1,c_2)=k_1RSS(c_1,c_2)+(1-k_1)DSS(c_1,c_2) \tag{3.7}$$

其中，$k_1 \in [0, 1]$。

由公式 3.7 可知，k_1 越小，则 DSS 所占的比重越大。当 $k_1 = 0$ 时，$CCS(c_1, c_2)$退化为只考虑 DSS，即类所使用的服务。k_1 越大，则 RSS 对 CCS 的贡献越大。当 $k_1 = 1$ 时，$CCS(c_1, c_2)$只关注 RSS，即类的客户。这里，类 c'是类 c 所使用的服务，若 $c' \in S_D(c)$；类 c'是类 c 的客户，若 $c' \in S_R(c)$。通常，我们认为 RSS 和 DSS 同等重要，所以将 k_1 的默认值设定为 0.5。

下面以图 3.7 为例来讨论类上下文。图 3.7 展示了一个简单的包质量评估器以及它所使用的标准库中的类。这里，包 *Controller* 负责控制评估流程；包 *IMetrics* 是度量子系统的抽象规约；包 *CohesionMetrics*

和 *CouplingMetrics* 分别计算包的内聚性和耦合性；包 *Builder* 封装了类 *Evaluator* 的创建逻辑；包 *DependenceGraph* 用于生成数据依赖图。不难看出，这些包具有语义上的内聚性。此外根据 3.1 节所提出的包分类，包 *IMetrics* 是接口包，包 *CohesionMetrics* 和 *CouplingMetrics* 是实现包，而其余包是构件包。根据公式 3.7，CCS(*SCCMetric*, *RCMetric*)的计算过程如下：

S_R(*SCCMetric*) = {*Evaluator*, *Builder*, *CohesionFactory*}
S_R(*RCMetric*) = {*Evaluator*, *Builder*, *CohesionFactory*}
S_D(*SCCMetric*) =
{*Cohesion*, *DepedenceGraph*, *DependenceGraphBuilder*, *Element*, *parser*, *os*}
S_D(RCMetric) =
{*Cohesion*, *DepedenceGraph*, *DependenceGraphBuilder*, *Element*, *parser*, *os*}
RSS(*SCCMetric*, *RCMetric*) = 1
DSS(*SCCMetric*, *RCMetric*) = 1
CCS(*SCCMetric*, *RCMetric*) = 1

CCS(c_1, c_2)能够揭示类 c_1 和 c_2 之间的语义耦合。一方面，RSS(c_1, c_2)反映了两个类被共同重用的程度。根据共同重用原则，趋向于共同重用的类总是紧密相关的。一般而言，类很少会孤立重用。为完成一个任务，它需要与作为该可重用抽象的其他类协作。事实上，这样的类总服务于一致的抽象，无论它们之间是否存在数据耦合。因此，类 c_1 和 c_2 被共同重用的频率越高，它们之间的关系就越密切。这符合 CRC 内聚性的基本思想。另一方面，DSS(c_1, c_2)体现了两个类使用共享服务的情况。类 c_1 和 c_2 很可能通过共享服务执行语义相关的操作。例如在图 3.7 中，包 *CohesionMetrics* 中的类均以数据依赖图（类 *DependenceGraph*）为基础计算包的内聚性。这与在类内聚性度量中采用的“共享类型”的想法一致[17, 35, 46]。因此，DSS(c_1, c_2)能够从较大程度上体现 c_1 和 c_2 之间的语义关系。

此外，CCS(c_1, c_2)的计算涉及了包间数据依赖。例如，计算 CCS(*SCCMetric*, *RCMetric*)需要系统中所有的包间数据依赖。这表明包内数据依赖和包间数据均对包的内聚性有贡献。仅考虑其中一种依赖不能很好地评估包的内聚性。

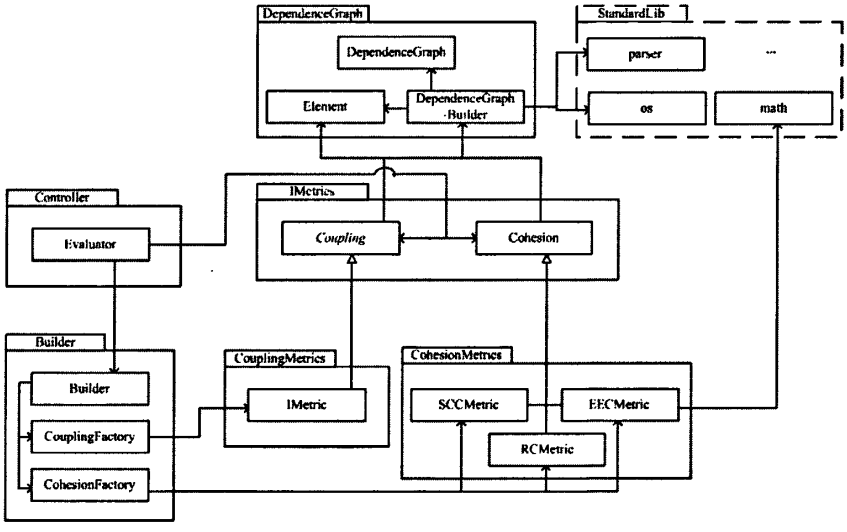


图 3.7 包质量评估器的体系结构

3.4.2 改进的包内聚度量

根据 3.4.1 节的讨论，类有两种关系：上下文关系和数据关系。给定类 c_1 和 c_2，它们之间有上下文关系，当且仅当 CCS(c_1, c_2)≠0；它们之间有数据关系，当且仅当 Dep(c_1, c_2) = 1。公式 3.8 给出了 Dep(c_1, c_2)的计算方式：

$$Dep(c_1,c_2)=\begin{cases}1 & if\ \ c_1 \xrightarrow{d} c_2 \vee c_2 \xrightarrow{d} c_1 \\ 0 & else\end{cases} \tag{3.8}$$

例如在图 3.7 中，类 *SCCMetric* 和 RCMetric 之间存在上下文关系，而类 *DependenceGraph* 和 *Element* 之间有数据关系。

定义 3.8 给定包 p，它的类带权交互图（Weighted Interaction Graph, WIG）是一个无向图 $WIG(p)=(V(p), E(p))$。其中，$V(p)=\{c \mid c\in p\}$，$E(p)=\{(c_1, c_2)\in V(p)\times V(p) \mid Wgt(c_1, c_2)>0\}$。这里，$Wgt(c_1, c_2)$的计算方式如公式 3.8 所示：

$$Wgt(c_1,c_2)=k_2 CCS(c_1,c_2)+(1-k_2)Dep(c_1,c_2) \tag{3.9}$$

其中，$k_2\in[0, 1]$。

由公式 3.9 可知，k_2 越小，则 Dep 所占的比重越大。当 $k_2=0$ 时，$CCS(c_1, c_2)$退化为只考虑数据关系。k_2 越大，则 CCS 对权的贡献越大。当 $k_2=1$ 时，$Wgt(c_1, c_2)$仅涉及上下文关系。一般地，我们将 k_2 置为 0.5，表示两者在计算过程中的地位相同。在这种情况下，若 c_1 和 c_2 间不存在数据耦合，则 $Wgt(c_1, c_2)$的最大值为 0.5，即两类之间的上下文耦合强度达到最高。

由上述讨论不难看出，$WIG(p)$的权和越高，p 的类之间的联系越紧密。因此，基于相似上下文的包内聚性度量（Similar Context Cohesiveness，SCC）被定义为 $WIG(p)$的实际权和与最大可能权和之比。

定义 3.9 给定包 p 和 $V(p)=\{c_1, c_2,..., c_m\}$，有

$$SCC(p)=\begin{cases}\dfrac{2\sum\limits_{(c_i,c_j)\in E(p)} Wgt(c_i,c_j)}{m(m-1)} & if\ \ m>1 \\ 1 & if\ \ m=1\end{cases} \tag{3.10}$$

当 $m=1$ 时，p 仅有一个类。在这种情况下，p 显然具有较高内聚性，我们设 $SCC(p)$为 1。当 $m>1$ 时，若每条边的权均为 0，即所有类之间均不存在任何关系，则 $SCC(p)=0$；若每条边的权均达到最大值 1，即所有类之间均存在数据关系和最强的上下文关系，则 $SCC(p)=1$。因此，$SCC(p)\in[0, 1]$。特别地，对于不具有内部数据耦合的包而言，$SCC(p)$的最大值为 0.5。

我们认为对于一个内聚的包 p 而言，$WIG(p)$应是一个连通图，且每条边上的权重应不小于 0.25。根据图论，一个有 m 个顶点的联通图至少有 $m-1$ 边。因此，SCC 度量的阀值应为 $1/2m$：若包的 SCC 值低于 $1/2m$，则它是不内聚的；反之，若它的 SCC 值高于 $1/2m$，则它具有较高的内聚性。例如，包 *CohesionMetrics* 的阀值应为 0.17。第 4 章将通过实验对 SCC 度量的阀值进行研究。

另外，SCC 度量对上下文的敏感性随 k_1 和 k_2 的变化而变化。一方面，根据公式 3.7，CCS 由类所使用的服务和类的客户决定，而类所使用的服务不会随上下文发生变化，即 DSS 是定值。因此，k_1 越小，DSS 所占比重越大，则 SCC 度量对上下文的敏感性越低。另一方面，由公式 3.8 可知，k_2 降低会使得 CCS 在 Wgt 中的比例降低，则 SCC 度量对上下文的敏感性也随之降低。

更重要地，与 HC 度量类似，SCC 度量同样满足 L. C. Briand 等所提出的良好模块内聚性度量应满足的性质[90-91]。根据 3.4.2 节的讨论，$SCC\in[0, 1]$，因而显然具有非负性及标准化以及最大值和最小值这两个性质。下面依次论证 SCC 度量满足其余两个性质。这些性质的详细描述见 3.3.3 节。

定理 3.3 给定包 p 以及类 c_1、$c_2\in p$，增加 $c_2 \xrightarrow{d} c_1$ 后形成的新包 p'应满足 $SCC(p)\le SCC(p')$。

证明：令增加依赖后形成的新类分别为 c_1'、c_2'，则根据定理条件有

(1) $S_R(c_1')=S_R(c_1)\cup S_R(c_2)$，因此$\forall c_i\in p$ 且 $c_i\neq c_1$，$RSS(c_1, c_i)\le RSS(c_1', c_i)$。

(2) $S_D(c_2')=S_D(c_1)\cup S_D(c_2)$，因此$\forall c_i\in p$ 且 $c_i\neq c_2$，$DSS(c_1, c_i)\le DSS(c_2', c_i)$。

(3) $Dep(c_1', c_2')=1$。

根据定义 3.8~3.9，有 $SCC(p)\le SCC(p')$。 ∎

定理 3.4 给定包 p_1、p_2，令 $p_3=p_1\cup p_2$，若 p_1 和 p_2 之间没有关系，则 $SCC(p_3)\le Max(SCC(p_1), SCC(p_2))$。

证明：根据 SCC 度量，包 p_1 和 p_2 之间不存在关系，当且仅当它们满足

$$\forall c_1 \in p_1 \wedge \forall c_2 \in p_2: \mathrm{Wgt}(c_1, c_2) = 0$$

令 $m_1 = |p_1|$、$m_2 = |p_2|$，则 $|p_3| = m_1 + m_2$，有

若 $m_1 = 1$ 且 $m_2 = 1$，则 $SCC(p_3)$ 显然为 0。因此，$SCC(p_3) \le \mathrm{Max}(SCC(p_1), SCC(p_2))$。

若 $m_1 = 1$ 且 $m_2 \neq 1$（或 $m_1 \neq 1$ 且 $m_2 = 1$），则

$$SCC(p_3) = \frac{m_2 - 1}{m_2 + 1} SCC(p_2) < SCC(p_2)$$

因此，$SCC(p_3) \le \mathrm{Max}(SCC(p_1), SCC(p_2))$。

若 $m_1 \neq 1$ 且 $m_2 \neq 1$，则

$$SCC(p_3) = \frac{m_1(m_1 - 1)SCC(p_1) + m_2(m_2 - 1)SCC(p_2)}{(m_1 + m_2)(m_1 + m_2 - 1)}$$

不失一般性，设 $SCC(p_1) \le SCC(p_2)$，有

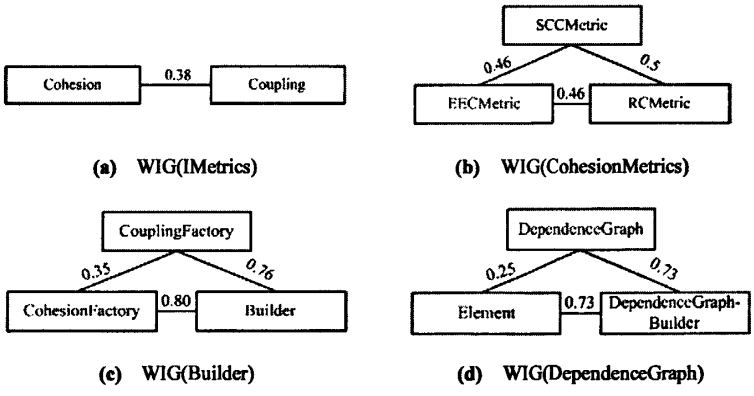
$$SCC(p_3) - SCC(p_2) \le -\frac{2m_1 m_2 SCC(p_2)}{(m_1 + m_2)(m_1 + m_2 - 1)} \le 0$$

因此，$SCC(p_3) \le \mathrm{Max}(SCC(p_1), SCC(p_2))$。

因此，若两个包之间不存在关系，则它们合并之后的 SCC 值不高于它们合并前 SCC 值的最大值。 ▌

3.4.3 实例分析

本节通过分析图 3.7 中各个包的内聚性来研究 SCC 度量的性质。图 3.8 给出了图 3.7 对应的 WIG。由图 3.7 和图 3.8 可知，包 *IMetrics* 和 *CohesionMetrics* 中的类通过上下文关系进行耦合，因此 Wgt 接近 0.5 说明它们之间有很强的上下文关系。而包 *DependenceGraph* 和 *Builder* 中的类同时具有上下文关系和数据关系，所以具有较高的 Wgt。由此可知，评估包的内聚性时应同时考虑数据关系和上下文关系。



(a) WIG(IMetrics)　(b) WIG(CohesionMetrics)

(c) WIG(Builder)　(d) WIG(DependenceGraph)

图 3.8　图 3.7 中包的 WIG

对图 3.8 的 WIG 进行了度量，得到如表 3.3 所示的度量值。在表 3.3 中，各包的 SCC 值均远大于对应的阀值，因此可认为这些包是内聚的。这一结论与上面我们根据开发经验和设计原则等所得的评估结果一致。其次，SCC 度量能够在 RC 度量和 EEC 度量不适用的接口包以及实现包上良好工作。可以看到，接口包和实现包的 RC 值与 EEC 值均为 0，而相应的 SCC 值分别为 0.38 和 0.47。因此可以得到结论：SCC 度量能够较好地挖掘类之间的语义关系，从而广泛地适用于各种类型的包。此外与 HC 度量相比，SCC 度量可能具有更好的可区分性。所谓可区分性是指对内聚程度不同的包，度量方法能给出不同的度量值。例如，包 *CoheisonMetrics* 负责内聚性度量，而包 *IMetrics* 却同时封装了内聚性度量和耦合性度量。根据开发经验，

虽然两个包均有较高的内聚程度，但前者应高于后者。根据表 3.3，它们的 SCC 值分别为 0.38 和 0.47，符合这两个包的实际情况。但是，它们的 HC 值却均为 1.00，未能较好地反映出这两个包的差异。其可能原因是 HC 度量只考虑了包的客户。最后，四种度量方法都可以用于计算构件包的内聚性。这进一步论证了现有包内聚性度量方法主要适用于构件包。不难看出，上述有关 RC 度量和 EEC 度量的结论与 3.3 节相同。

表 3.3 图 3.7 的度量结果

包名	SCC	RC	EEC	HC	SCC 对应阀值	类别
IMetrics	0.38	0.00	0.00	1.00	0.25	接口包
CohesionMetrics	0.47	0.00	0.00	1.00	0.17	实现包
Builder	0.64	0.67	0.59	1.00	0.17	构件包
DependnceGraph	0.57	0.67	0.59	1.00	0.17	

3.5 HC 度量和 SCC 度量的对比

本章提出了两种包内聚性度量方法：HC 度量和 SCC 度量。本节将从理论上对它们进行对比，从而为改进包内聚性度量提供理论依据，并辅助开发者在实际度量中选择合适的方法。

作为基于上下文的包内聚性度量方法，HC 度量和 SCC 度量有相似之处。相对于已有的包内聚性度量方法 RC 度量和 EEC 度量，两者均通过包间数据依赖来挖掘类间语义关系，从而能够更好地评估包的内聚程度。3.3.4 节和 3.4.3 节的实例分析表明 HC 度量和 SCC 度量能在只有极少数据依赖的接口包和实现包上良好工作，而 RC 度量和 EEC 度量不能。此外与 HC 度量类似，SCC 度量也可用于计算 CRC 内聚性。当 $k_1 = 0$ 且 $k_2 \neq 0$ 时，SCC 度量仅考虑类的客户。在这种情况下，两个类紧密耦合，当且仅当它们总是被共同重用或者存在数据耦合。可以看出，SCC 度量通过考察任意两个类的重用情况来估计整个包的重用情况。这符合 CRC 内聚性的基本思想。因此，SCC 度量的适用范围至少与 HC 度量一致。最后，两者均满足 L. C. Briand 等所提出的良好模块内聚性度量应满足的四个性质。

然而，SCC 度量和 HC 度量在使用上下文的方式上存在差异。第一，HC 度量只考虑包的客户，而 SCC 度量不仅考虑包的客户，还考虑了包所使用的服务，所以对包上下文的利用更加充分。第二，在 HC 度量中上下文的粒度是包，而在 SCC 度量中上下文的粒度是类。第三，HC 度量将包中的类作为整体考虑，即这些类是否被包的客户共同重用。但是，SCC 度量通过挖掘两个类之间的语义关系来推断整个包的内聚性：若任何类之间存在紧密耦合，则包的内聚程度较高。上述这些差异导致 SCC 度量从一定程度上优于 HC 度量：

首先，SCC 度量能够更好地挖掘类之间的语义关系，其适用性更好。例如在图 3.7 中，类 *Evaluator* 没有客户，则它与其他类的上下文关系需要依赖类所使用的服务来挖掘。在这种情况下，HC 度量不能工作，而 SCC 度量可以。然而，在包中的类所使用的服务较少时，SCC 度量等价于仅考虑客户，相对于 HC 度量没有优势。因此，SCC 度量可能也不适用于工具包。如图 3.1 中，类 *Vector* 和 *Map* 不存在“上文”，则 CSS(*Vector*, *Map*) = RSS(*Vector*, *Map*)。

其次，SCC 度量具有更高的可区分性。由表 3.3 可知，虽然 HC 度量能够指出图 3.7 中的各个包均具有较高的内聚程度，然而它难以较好地反映出这些包之间内聚程度的差异。

再者，SCC 度量对客户集变化的敏感度往往低于 HC 度量。换句话说，在大多数情况下，SCC 值不会随客户集的变化而产生较大波动。事实上，SCC 度量对上下文的敏感性由参数 k_1 和 k_2 的决定。当 k_1 和 k_2 降低时，SCC 度量对客户集的敏感性也随之降低。然而，对于 HC 度量而言，客户集需要满足“充足”这个性质，否则所得度量结果的稳定性和有效性可能受到影响。

不过，HC 度量的计算复杂度远低于 SCC 度量，因此有助于开发者在大规模系统中更快地获得包质量的反馈。此外，HC 度量不需要用户设定参数，而 SCC 度量却依赖于 k_1 和 k_2 的选择。以图 3.7 为例，若设 $k_1 = 0$，则类 *Element* 和 *DependenceGraph* 之间不存在任何关系，这与我们的开发经验相悖。更重要地，虽然 HC 度量对客户集敏感，但是充足客户集足以保证其度量值的有效性和稳定性。综上所述，SCC 度量的

复杂性不一定会带来相应的好处。

3.6 本章小结

随着规模的增加，软件需要通过包在更高层次上进行组织[45]。包结构重构可以在在软件演化过程中持续优化包结构的质量。作为重要的质量指标，包内聚性可用在包结构重构中识别不良的包结构以及评估重构的有效性。传统上，包内聚性度量依赖于包内部成员之间的数据依赖。然而，这种依赖不足以表达包的复杂语义。为此，我们提出了两种基于上下文的包内聚性度量方法：

第一种是基于客户使用的包内聚性度量方法 HC，它认为若多个类总是被共同重用，则这些类之间存在紧密联系。可以看出，HC 度量通过客户行为挖掘包属性的基本思路与网站度量领域通过用户浏览行为评估网站属性的想法一致[92, 98]。

第二种是基于相似上下文的包内聚性度量方法 SCC，它的核心思想是若两个类的上下文相似，则它们有紧密的语义耦合。与 HC 度量相比，SCC 度量通过类上下文对包上下文进行考察，从而更加全面地使用了包上下文的信息。

相对于 RC 度量和 EEC 度量，上述两种方法均利用包所处的“环境”来理解包的语义，因而能更好地评估包的内聚性。表 3.4 给出了 RC 度量、EEC 度量、HC 度量和 SCC 度量的对比情况。由表 3.4 可知，虽然 SCC 度量相对于 HC 度量有一定的优势，但其计算复杂度更高。

表 3.4 RC、EEC、HC 度量和 SCC 度量的对比情况

度量	包类别				数据依赖类别		客户敏感性	计算复杂度
	工具包	接口包	实现包	构件包	包内数据依赖	包间数据依赖		
RC	不适用	不适用	不适用	适用	考虑	不考虑	无	低
EEC	不适用	不适用	不适用	适用	考虑	不考虑	无	中
HC	可能不适用	适用	适用	适用	考虑	考虑	强	低
SCC	可能不适用	适用	适用	适用	考虑	考虑	随参数变化	高

第四章 包内聚性度量方法的实验研究

第 3 章提出了两种基于上下文的包内聚性度量方法，并从理论上对它们的适用性和性质进行了深入分析。研究结果表明与现有方法相比，基于上下文的包内聚性度量方法具有更加广泛的适用性。在理论研究的基础上，本章对开源 Web 框架 *Django*[99]进行了大规模实验，以进一步研究这些方法在大型程序库中的效果，从而为改进包内聚性度量方法提供定量依据。

考虑到 *Django* 是 Python 程序，与利用静态语言编写的代码有较大不同，本章首先讨论了 Python 包的特点以及相应的内聚性度量方法，然后对实验进行了设计，最后分析了实验结果。实验研究表明基于上下文的包内聚性度量方法能够适用于 Web 框架之类的程序库。

4.1 基本定义

根据所使用的信息，包内聚性度量方法可被两类：基于数据流的方法和基于上下文的方法。现有度量方法主要属于前者。它们将数据流的耦合视为语义的耦合，常导致度量结果与包的实际内聚程度相悖。典型方法包括 RC 度量和 EEC 度量。其中，RC 度量没有最大值，也缺乏根据 RC 值评估包内聚程度的启发式规则。例如，给定包 p，且 $RC(p)=0.5$，若 $|p|=2$，则 0.5 表示 p 的包内数据耦合度达到最大；若 $|p|>2$，则该值代表 p 的包内数据依赖图是非连通图，其内部数据耦合度较小。这使得开发者难以根据 RC 值作出设计决策。为此，我们提出了一种改进的 RC 度量（Refined RC，RRC），其计算方法如公式 4.1 所示：

$$RRC(p)=\begin{cases}\dfrac{RC(p)}{m-1} & if\ \ m>1\\ 1 & if\ \ m=1\end{cases} \tag{4.1}$$

其中，$m=|p|$。

当 $m=1$ 时，p 显然具有较高内聚性，我们设 $RRC(p)$为 1。当 $m>1$ 时，若 p 没有内部数据依赖，则 $RRC(p)=0$；若任意包成员之间均存在数据依赖，则 $RRC(p)=1$。因此，$RRC(p)\in[0,1]$。可以看出，RRC 度量直接反映了包内部数据耦合的程度。下文将同时对 RC 度量、RRC 度量、EEC 度量、HC 度量和 SCC 度量进行实验，以更加深入对比这两类包内聚性度量方法。

第 3 章提出了基于上下文的包内聚性度量方法。其核心思想是通过包上下文来评估包的内聚程度。根据第 3 章的理论分析，这类方法能够较好地适应包的复杂语义。作为理论研究的重要补充，本章将通过对 Python Web 框架 *Django* 的实验研究来进一步考察各种包内聚性度量方法在程序库中的适用情况和性质。

然而，Python 是动态语言，具有许多与 Java 和 C++之类的静态语言不同的性质。Python 的多态实现并非基于继承体系结构，而是基于结构一致性[100]。即是说，具体的实现类不需要继承于抽象基类，而只需要提供系统规约所要求的行为（第 5 章将对 Python 的结构类型系统进行详细讨论）。这样，软件系统不再依赖抽象基类（接口）来表达规约，从而使得 Python 的代码组织方式与静态语言有较大的区别。因此，我们应该对包内聚性度量方法作出相应的调整，以适应 Python 程序的特点：

第一，Python 应采用不同的包分类方法：低数据耦合包和高数据耦合包。由于不再利用抽象基类表达系统规约，所以 Python 程序通常没有接口包。虽然 Python 也常常通过一个包来封装实现系统特定规约的类，然而与 3.1 节所提出的实现包不同，这种包在很多情况下有大量的包内数据依赖。这是由于这种包往往同时包含父类和子类。在 Python 中，继承的主要作用是重用，而非提供抽象，因而这样的组织方式没有违反依赖倒置原则和稳定抽象原则。以 Python 项目 *Freevo*[1]为例，包 *skins.main* 封装了类 *Skin_Area* 及其子类，这些类通过函数 draw 实现了“描绘播放器的皮肤”的抽象。这里，定义父类的主要原因是子类有一

[1] Freevo 是一个 Linux 下的播放器，可将数字信号转化为电视信号，从而实现利用个人 PC 收看电视节目的功能。其主页是 http://freevo.sourceforge.net/。

批共同的属性，如 *area_name* 等。另外，这种包常利用它的初始化模块*__init__*来提供界面。即是说，在初始化模块中导入包的所有模块，从而允许客户通过它来访问包的所有功能。同样以 *Freevo* 的包 *skin.widgets* 为例，它由实现类 *ScrollableTextScreen* 和 *TextEntryScreen* 组成，这些类通过提供函数 *show()*来“显示播放器的皮肤”。在初始化模块 *skin.widgets.__init__*中，该包导入了实现类所在的模块 *skin.widgets.scrollabletext_screen* 和 *skin.widgets.textentry_screen*。这样，它的客户仅需使用指令 *import skin.widgets* 就可以使用该包的所有功能。综上所述，3.1 节的包分类方法不适用于 Python 包。

考虑到两类包内聚性度量方法的核心区别在于是否仅考虑包内数据依赖，我们依据包内数据依赖的紧密程度将 Python 包划分为低数据耦合包和高数据耦合包。低数据耦合包是指包内数据耦合程度较低的包。高数据耦合包是指包内数据耦合程度较高的包。例如在图 3.1 中，包 *PContainer* 是低数据耦合包，而包 *PCompiler* 是高数据耦合包。由于 RRC 度量是反映包内数据耦合程度的指标，所以我们通过 RRC 值来区分两类包：若包的 RRC 值小于 0.3，则它是低数据耦合包；反之，若包的 RRC 值不小于 0.3，则它是高数据耦合包。特别地，RRC 值为 0 代表被度量包没有内部数据依赖。

第二，Python 包的内聚性可以通过模块间数据依赖来评估。Python 包由模块构成，而一个模块常常包含多个紧密关联的类，所以模块间数据依赖比类间数据依赖简单。更重要地，计算 RC 值、RRC 值、EEC 值以及 SCC 值只需要包的组成成分之间的数据依赖关系。因而为了简化计算、快速实验，我们可以通过模块间数据依赖来计算 Python 包的内聚性。

定义 4.1 给定模块 c_1 和 c_2，若 c_1 引用了定义于 c_2 中的名字，则称 c_1 数据依赖于 c_2，记为 $\xrightarrow{d}$。若 (c_1, c_2)属于数据依赖 $\xrightarrow{d}$ 的传递闭包，则称 c_1 间接数据依赖于 c_2，记为 $\xrightarrow{d+}$。

此外，对数据依赖的简化同样适用于 HC 度量。根据 3.3.2 节，HC 度量区分使用依赖和继承依赖的主要目的是从接口包的客户集中排除实现包。由于不存在接口包，所以这样的区分没有意义。定义 4.2 给出了基于模块间数据依赖的包客户。

定义 4.2 给定包 p_1 和 p_2，如果∃模块 $c_1 \in p_1$，模块 $c_2 \in p_2$，满足 $c_1 \xrightarrow{d} c_2$，则称 p_1 是 p_2 的客户。给定包 p，客户集 ClientPackage(p) = $\{p_i \mid$ 包 p_i 是包 p 的客户$\}$。

4.2 实验设计

本节讨论如何设计实验以研究两类包内聚性度量方法在大型程序库，特别是 Web 框架中的适用情况和性质。第 1 节提出了实验研究的目标。第 2 节介绍了实验对象和实验数据。第 3 节讨论了实验数据的分析方法。

4.2.1 实验目标

实验研究的目的在于通过考察各种包内聚度量方法在产品代码中的效果来指导我们下一步的理论研究。根据这一总体目标，我们提出了实验研究所需回答的五个问题：

第一，与基于数据流的方法相比，基于上下文的方法是否能适用于更多种类的包？这可以帮助我们了解包间数据依赖在包内聚性度量中的作用，从而考察基于上下文评估包内聚程度的基本思想的合理性。

第二，各种包内聚性度量方法的可能阀值是多少？给定包 p 和度量方法 C，若 p 的 C 值大于 C 的阀值，则认为 p 是高内聚包；反之，若小于阀值，则认为 p 是低内聚包。换句话说，阀值可以帮助开发者依据度量结果判断包的内聚程度，对于指导开发者进行设计决策有重要意义。所以，阀值的确定是度量所需解决的核心问题之一。

第三，就基于上下文的方法而言，SCC 度量是否优于 HC 度量？包通过包间数据依赖与“环境”进行交流和协作。基于包间数据依赖，我们可以定义多种包内聚性度量方法。哪种定义方式能更好地挖掘包的语义，从而广泛地适用于各种类型的包。问题三的回答可以指导我们改进基于上下文的方法。

第四，充足客户集是否能保证基于上下文的方法获得稳定而有效的度量值？基于上下文的方法依赖于包间数据依赖，然而对于大范围重用的包（如大型程序库），包间数据依赖会随着包客户的变化而变化。也就是说，基于上下文的方法对客户集的变化是敏感的。根据第 3 章的理论分析，充足客户集能够有效保

证度量结果的稳定性和有效性。其中，有效性是指度量结果可以反映被度量包的真实内聚程度。稳定性是指度量结果不会因客户集包含新客户而产生巨大波动。本章将通过实验来研究该结论的正确性。

第五，充足客户集应满足什么条件？充足客户集是指能够保证度量值稳定而有效的客户集。第 3 章指出充足客户集应满足如下两个条件：(i) 涵盖被度量包的所有典型使用场景；(ii) 每种客户所占的比例与其对应使用场景的发生频率一致。本章将利用实验对充足客户集应满足的条件进行深入研究，以检查第 3 章的理论结论是否正确。

4.2.2 实验对象

我们的实验数据来源于 *Django v.096*。*Django* 是一个抽象层次较高的 Web 框架，它采用了 MVC 模式，具有轻便、实用与设计清晰等诸多优点，被广泛地应用于快速开发各种类型的 Web 程序[99]。目前，*Django* 包含 700 多个模块，这些模块被组织成 60 多个包。为计算这些包的 HC 值和 SCC 值，我们收集了 5 个涵盖不同应用领域的 *Django* 客户程序。这些程序总共涉及 1600 多个模块和 140 多个包。表 4.1 给出了这些客户程序的描述。

表 4.1 用于计算 HC 度量值和 SCC 度量值的 Django 客户程序

No.	项目	描述	主页
1	PyLucid	轻量级的内容管理系统	www.pylucid.org
2	Movie Vote	基于 Web 的电影投票系统	movie-vote.googlecode.com
3	Satchmo	电子商务框架	www.satchmoproject.com
4	Sphene Community Tool	用于构建社区网站的 Wiki 应用	sct.sphene.net
5	Byteflow	博客引擎	byteflow.su

4.2.3 数据收集与预处理

在本章的实验研究中，我们主要收集两种数据集来研究各种包内聚性度量方法的适用范围和性质，以回答 4.2.1 节提出的五个问题。

首先，为了检查基两种方法的有效性（上一小节所述第一至三个问题），我们收集了 *Django* 中各个包的 RC 值、RRC 值、EEC 值、HC 值以及 SCC 值。其中，计算 HC 值和 SCC 值需要涉及表 4.1 中的所有客户程序。在此过程中，我们要排除仅含一个模块的包。这种包的各种内聚性度量值均为 1，不具备可区分性，会干扰实验结果的有效性。

其次，我们还收集了基于不同客户程序获得的 HC 值和 SCC 值，以考察这两种方法的优劣和性质（上一小节所述第三至五个问题）。在实验中，我们按照表 4.1 的编号将客户程序逐一加入客户集，从而获得针对相同包的一组不同的 HC 值和 SCC 值。表 4.2 给出了每组实验包含的客户程序。通过检查在不同客户集下 HC 值和 SCC 值的变化，我们可以知道基于上下文的方法对客户集的敏感性。所谓敏感性是指在不同客户集下度量值的变化程度：变化越大，则敏感性越高；反之，变化越小，则敏感性越小。

表 4.2 用于研究 HC 度量和 SCC 度量稳定性的实验

实验	项目					
	Django	PyLucid	Movie Vote	Satchmo	Sphene Community Tool	Byteflow
1	√					
2	√	√				
3	√	√	√			
4	√	√	√	√		
5	√	√	√	√	√	
6	√	√	√	√	√	√

实验的主要目标之一是评估各种包内聚性度量方法是否能真实地反映出包的实际内聚程度。为此，我们在深入研究 *Django* 的基础上，将 *Django* 中的包分别标注为高内聚、难以判定和低内聚。标注的基本原则是单一职责原则：内聚包服务于单一目标，而不内聚的包往往承担多种任务。此外，我们还综合考虑了

R. C. Martin 所提出的包内聚性原则：共同重用原则、重用发布等价原则以及共同封闭原则。根据标注结果，*Django* 中的多数包均属于高内聚。附录 1 给出了 *Django* 中主要包的描述以及标注情况。

4.2.4 数据分析方法

本节介绍实验数据的分析方法，这些方法主要包括描述统计分析、Spearman 相关性分析、决策树分析以及 Mann-Whitney U 检验。其中，描述统计分析和 Spearman 相关性分析可以帮助我们从多个角度回答问题一和三，决策树分析能辅助解答问题一到三，而 Mann-Whitney U 检验主要针对问题四和五。

(1) 描述统计分析

通过描述统计分析，我们检查度量值的变化和分布。事实上，度量值变化较大的方法具有更好的可区分性，可以在内聚程度不同的包上获得不同的度量结果。相反地，度量值变化较小的方法不能有效区分不同的包内聚程度[39, 41-42]。此外，度量值的分布情况应符合 *Django* 中包的实际内聚程度。否则，我们认为该度量方法难以适用于 *Django*，进而可以推断它不能在 Web 框架之类的程序库上良好工作。

(2) Spearman 相关性分析

考虑到实验数据不是正态分布，我们采用 Spearman 相关性分析来考察各种包内聚性度量方法之间是否相互独立[101]。若两种方法的度量值之间的相关性高，则它们很可能依赖于相同的信息；反之，若相关性低，则说明它们考虑了被度量包的不同方面。

(3) 决策树分析

决策树是一种被广泛使用的分类学习算法，具有模型简单直观，计算效率高，且不需要假设先验概率分布等诸多优点[97]。基于此，我们利用决策树分析来建立通过度量值判断包内聚程度的启发式规则，以研究各种度量方法的阀值，并检查它们的效果。这里，决策树的具体实现采用 C4.5[102]，而相应的精度计算使用 10 折交叉检验[97]。

此外，我们还专门对低数据耦合包进行了决策树分析，从而考察基于上下文的方法是否能适用于低数据耦合包。对于这种包，基于数据流的方法往往失效。

(4) Mann-Whitney U 检验

Mann-Whitney U 检验主要用于评估两组样本之间是否存在显著差异[101]。本章利用它来评估 HC 值和 SCC 值在不同实验中的差异，从而研究这两种方法对于客户集的敏感性。

4.3 实验结论

本节根据 4.2 节的数据分析方法对实验结果进行了深入挖掘，从而回答 4.2 节所提出的五大问题：第 1 节描述了描述统计和相关性分析的结果；第 2 节是决策树分析的结果；第 3 节给出了 Mann-Whitney U 检验的结果；第 4 节总结了前 3 节的分析结果。

4.3.1 描述分析

表 4.3 是度量数据的描述统计结果。其中，Max.、75%、Med.、25%、Min.、Mean 和 Std.Dev 分别代表了最大值、四分之三位数、中位数、四分之一位数、平均值以及标准差。

表 4.3 描述统计的结果

度量	Max.	75%	Med.	25%	Min.	Mean	Std.Dev.
RC	4.620	0.857	0.500	0.125	0.000	0.834	1.114
RRC	1.000	0.683	0.250	0.015	0.000	0.378	0.374
EEC	1.000	0.249	0.072	0.001	0.000	0.204	0.330
HC	1.000	1.000	1.000	0.673	0.090	0.812	0.277
SCC	0.990	0.719	0.431	0.390	0.250	0.523	0.208

由表 4.3 可知，RRC 值的中位数较低，只有 0.250，这说明低数据耦合包在 Web 框架中广泛存在。然

而在很多情况下，这种包具有较高的语义内聚性。下面结合图 4.1 对这种情况进行分析。图 4.1 展示了包 *django.contrib.comments.views* 及其客户 *Satchmo*、*django.core* 和 *Byteflow*。在该包中，模块 *comments* 负责显示和发布评论，模块 *userflags* 通过用户 ID 标记评论，而模块 *karma* 对评论进行分级。虽然它没有内部数据依赖，但是它的内部成员均服务于网站上用户评论的管理，所以它在语义上是内聚的。

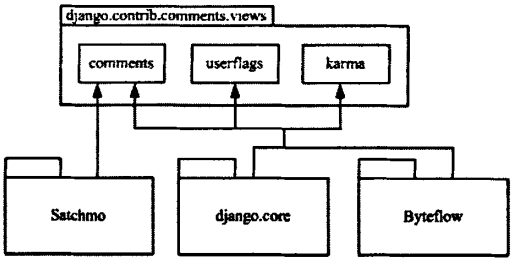


图 4.1 包 django.contrib.comments.views 及其客户

表 4.3 还表明基于数据流的方法难以适用于 Web 框架。首先，较低的 RRC 值的中位数代表被度量的包大多是不内聚的。其次，RC 值的中位数为 0.500，同样表示大多数被度量包的内聚程度较低。根据 4.1 节，给定包 p，当$|p|>2$ 时，$RC(p)=0.500$ 表示 p 不内聚。而 *Django* 中的大多数包所含有的模块数均大于 2。最后，EEC 值的四分之三位数是 0.249，同样表明 *Django* 中的大多数包的内聚性较低。这些结论与 *Django* 包的实际内聚程度不符。由此可知，在 Web 框架上，包的内部数据流不足以表达包的复杂语义，从而导致仅利用这种信息的度量方法在很多情况下失效。

最后，表 4.3 说明基于上下的方法能够获得较好的度量结果。HC 值的四分之一位数为 0.673，这个较高的数值表明大多数被度量包是内聚的。其结论与我们的开发经验和设计原则一致。此外，SCC 值的四分之三位数、中位数与四分之一位数均分布于 0.500 左右，同样符合 Django 包的实际内聚程度。低数据耦合包在 Web 框架中广泛存在。根据 3.4.2 节，低数据耦合包的最大 SCC 值应在 0.500 左右。因此，SCC 值分布于 0.500 附近说明这些包具有较高的内聚性。

在表 4.3 的基础上，我们分别对低数据耦合包和高数据耦合包的 SCC 值进行了描述统计分析，所得结果如表 4.4 所示。可以看到，两种包的 SCC 度量值主要分布在两个不同的区间：[0.000, 0.500]和[0.500, 1.000]。这说明在使用 SCC 度量时，应对低数据耦合包和高数据耦合包采用不同的评估标准。对于低数据耦合包，其 SCC 度量值通常不大于 0.500。例如在图 4.1 中，$SCC(\textit{django.contrib.comments.views}) = 0.41$。对于高数据耦合包，其数据耦合对 SCC 值的贡献度大约为 0.500。因此，这种包的 SCC 值通常高于 0.500：若高数据耦合包同时具有紧密的上下文耦合，则该包的 SCC 值接近 1.000；相反地，若高数据耦合包无上下文耦合，则它的 SCC 值接近 0.500。由于高数据耦合并非代表高内聚，因而 SCC 度量有助于我们发现具有紧密数据耦合但却不内聚的包。

表 4.4 SCC 度量值的描述统计结果

类别	Max.	75%	Med.	25%	Min.	Mean	Std.Dev.
低数据耦合包	0.470	0.315	0.410	0.315	0.250	0.366	0.065
高数据耦合包	0.990	0.815	0.755	0.620	0.470	0.7112	0.156

表 4.5 记录了各种包内聚性度量方法的 Spearman 相关性分析的结果。根据表 4.5，我们可得到以下结论。第一，基于数据流的方法之间具有较高的相关性，这说明它们依赖于相同的信息。该结论符合它们均使用包内数据依赖的实际情况。第二，除 RC 度量外，HC 度量与其余度量方法的相关程度较低，从而表明包间数据依赖在 HC 度量中占有重要地位。然而，我们注意到 HC 度量与 RC 度量成负相关，即 HC 值上升时，RC 值反而下降。根据上面的讨论，HC 度量能较好地适用于低数据耦合包，但是 RC 不能。因此，对于内聚的低数据耦合包，HC 值往往较高，而 RC 值却接近 0。以图 4.1 中的包 *django.contrib.comments.views*

为例，它的 HC 值和 RC 值分别等于 0.800 和 0.000。进一步地，低数据耦合但高内聚的包在 Web 框架中大量存在。这使得 HC 值主要分布在区间[0.600, 1.000]，但 RC 值却主要分布在[0.000, 0.600]。这种相异的分布导致两种度量方法成负相关。第三，SCC 度量与基于数据流的方法之间存在较大的相关性，但却与 HC 度量不相关。这表明当 k_2 为 0.5 时，包内数据依赖对于 SCC 值的贡献较大。这一结论与表 4.4 一致。在表 4.4 中，低数据耦合包的 SCC 值与高数据耦合包的 SCC 值之间存在较大差别，从而说明包内数据耦合可以显著提高 SCC 值。

表 4.5 Spearman 相关性分析的实验结果

	RC	RRC	EEC	HC	SCC
RC	1.000	0.665*	0.394*	-0.355*	0.559*
RRC		1.000	0.873*	-0.002	0.920*
EEC			1.000	0.091	0.767*
HC				1.000	0.063
SCC					1.000

注：置信水平 0.01（双尾）

4.3.3 决策树分析

图 4.2 分别给出了 RC 值、RRC 值、EEC 值、HC 值以及 SCC 值的决策树以及对应的 10 折交叉检验的模型精度 Per.。由图 4.2 可知，HC 度量能从较大程度上反映包的内聚程度。一方面，它的决策树精度为 0.885，说明该决策树的分类性能良好。另一方面，当包的 HC 度量值大于 0.47 时，该包被判定为高内聚。由表 4.3 可知，*Django* 中的大多数包的 HC 度量值均大于 0.47，即是说大多数包均为高内聚包。这符合开发经验和设计原则。因此，我们可认为 HC 度量的可能阀值是 0.47。然而，我们也注意到在包的 HC 值大于 0.16 且小于 0.27 时，HC 度量的决策树将其归为难以判定。这表示在 *Django* 中存在 HC 值较低却内聚的包。例如，包 *db* 负责处理数据库相关的任务，用于向领域层提供统一的数据库接口，它在语义上是内聚的。然而在大多数情况下，客户仅使用 *db* 中的部分功能，所以，HC(*db*) = 0.270。虽然这种情况在 *Django* 中较少，但是我们在未来的实验中会考察它的出现频率，从而进一步研究 HC 度量的效果。

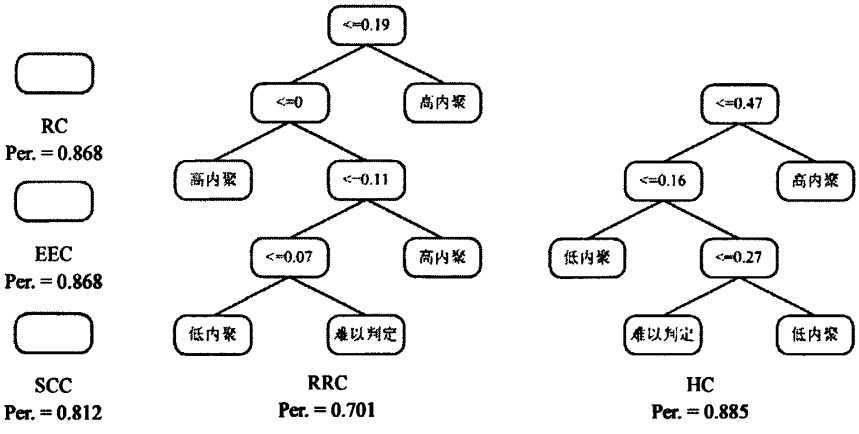


图 4.2 度量方法的决策树

此外，图 4.2 还表明 RRC 不能较好地反映包内聚性。在图 4.2 中，当被度量包的 RRC 值分布在区间[0, 0]、(0.07, 0.11]以及(0.19, 1]时，RRC 的决策树将其归为高内聚包。这是因为 *Django* 中存在大量低数据耦合但却高内聚的包，从而使得在我们的实验中低 RRC 值常对应于高内聚包。然而在很多情况下，低 RRC 值很可能对应低内聚包。例如在图 3.1 中，若合并包 *PRoom* 和 *PCompiler*，则所得包的 RRC 值仅为 0.095。然而，不难看该包不具有语义内聚性。这种相悖的情况使得过低的 RRC 值对于开发者的指导意义不强。

最后，根据图 4.2，RC 度量、EEC 度量和 SCC 度量的决策树仅含单个节点，说明它们可能难以区分不同的包内聚程度。值得注意的是，对 RC 度量和 EEC 度量的结论与 4.3.1 节相同，而对 SCC 度量却有完全相反的论断。其可能原因是在 $k_2 = 0.5$ 的情况下，SCC 度量过多地依赖于包内数据依赖，与基于数据流的方法相关性较高。在未来工作中，我们将进行更加广泛的实验研究，以深入讨论 k_2 取值对 SCC 度量的影响。不过，我们也发现这些方法的决策树的精度均大于 0.700。这是因为在 Django 中存在大量的高内聚包，从而使得决策树只需将所有包都判定为高内聚即可具备较高的精度。

为进一步研究 HC 度量和 EEC 度量对于低数据耦合包的适用性，我们对这种类型的包进行了决策树分析，其分析结果如图 4.3 所示。从图 4.3 中我们可以得到与图 4.2 类似的结论。首先，HC 度量能够较好的适用于低数据耦合包，且其可能阀值为 0.47。其次，SCC 度量可能不能较好反映低数据耦合包的内聚程度。

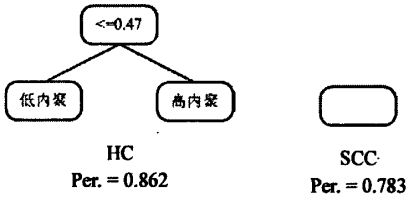


图 4.3 基于上下文的方法关于低数据耦合包的的决策树

4.3.3 Mann-Whitney U 检验

我们对 HC 度量进行了配对实验，并对实验结果进行了 Mann-Whitney U 检验，其结果记录于表 4.6。这里，一组配对实验由表 4.2 中一对相邻实验组成。例如，实验 1~2 构成一组配对实验。对于配对实验，平均秩（总秩）的差距越小，则用于实验的两组度量结果的差距越小；反之，平均秩（总秩）的差距越大，则两组度量结果的差距越大。

表4.6 HC度量的Mann-Whitney U检验结果

No.	实验	Mean Rank	Sum of Ranks	U	W	Z	p-value
1	1-2	53.790	2851.000	1389.000	2820.000	-0.106	0.042
		53.210	2820.000				
2	2-3	53.610	2841.500	1398.500	2829.500	-.0.041	0.016
		53.390	2829.500				
3	3-4	54.930	2911.500	1328.500	2759.500	-0.511	0.196
		52.070	2759.500				
4	4-5	54.370	2881.500	1358.500	2789.500	-0.305	0.120
		52.630	2789.500				
5	5-6	53.500	2835.500	1404.500	2835.500	0.000	0.000
		53.500	2835.500				

表 4.6 表明不同种类的客户会导致 HC 值的较大波动。由表 4.6 可知，在置信水平 0.050 下，第 1 组、第 2 组和第 5 组配对实验的平均秩（总秩）的差别极小。在置信水平 0.200 下，第 3 组和第 4 组配对实验的平均秩（总秩）有较大区别。这说明客户程序 *Satchmo* 代表了与其他客户程序不同的使用场景，因而包含不同的使用方式。例如，在图 4.1 中，包 *django.contrib.comments.views* 有两种客户。一种是其主要功能为允许用户自由发表评论的系统，如博客、论坛等。这种客户通常会使用包中的所有模块。客户程序 *Byteflow* 和 *django.core* 均属于这一类客户。第二种是发表评论仅作为次要功能的系统，如在线购物系统。这类客户程序通常需要对产品，而非对评论进行分级。这使得它们往往不需要使用模块 *karma*。其典型的例子是客户程序 *Satchmo*。因此，当实验 4 加入 *Satchmo* 时，HC(*django.contrib.comments.views*)从 1.000 变为 0.670。

而当实验 5 加入 *Byteflow* 时，HC 值又提高到 0.800。

因此，就 HC 度量而言，充足客户集需要满足如下两个条件：第一，充足客户集应覆盖被度量包的所有典型使用场景。否则，当加入新的客户时，HC 值会发生剧烈变化。例如在表 4.6 中，*Satchmo* 的加入引起了 HC 值的较大改变。第二，在充足客户集中，每种客户所占比例应与其对应使用场景的出现频率一致：使用场景出现越多，则与该场景对应的客户所占的比例应越大；反之，使用场景出现越少，则对应客户所占比例应越小。以图 4.1 为例，若客户集包含过多的第二种客户，即发表评论仅作为次要功能的系统，则 HC(*django.contrib.comments.views*)将下降到一个极低的值，从而使得度量结果与该包实际内聚程度不符。综上所述，充足客户集应反映被度量包的实际使用情况。在客户集充足的条件下，若新加入的客户未打破原有各种客户之间的平衡，则 HC 值不会发生较大的波动。不难看出，充足客户集可以较好地保证 HC 值的稳定性和有效性。

此外，我们还对 SCC 度量的配对实验的结果进行了 Mann-Whitney U 检验，其结果如表 4.7 所示。在置信水平 0.050 下，所有配对实验的平均秩（总秩）差距都极小，从而说明 SCC 度量对客户集的变化不敏感。根据 3.4 节的讨论，这主要有两方面的原因。一方面，当 $k_2 = 0.5$ 时，包内部数据依赖对 SCC 值的贡献较大，而 *Django* 的客户程序主要与包间数据依赖有关。另一方面，当 $k_1 = 0.5$ 时，S_R 与 S_D 在计算 SCC 值中占同等地位，而客户集的变化只会引起前者的改变。例如在图 4.1 中，将 RSS(*comment, userflags*)从 0.5 变为 1 仅会使 SCC(*django.contrib.comments.views*)增加 0.042。

表4.7 SCC度量的Mann-Whitney U检验结果

No.	实验	Mean Rank	Sum of Ranks	U	W	Z	p-value
1	1-2	53.380	2829.000	1398.000	2829.000	-0.041	0.017
		53.620	2842.000				
2	2-3	53.420	2831.000	1400.000	2831.000	-0.028	0.012
		53.580	2840.000				
3	3-4	53.380	2829.000	1401.500	2832.500	-0.019	0.017
		53.620	2842.000				
4	4-5	53.440	2832.500	1401.500	2832.500	-0.019	0.008
		53.560	2738.500				
5	5-6	53.420	2831.500	1400.500	2831.500	-0.025	0.01
		53.580	2839.500				

4.3.4 实验小结

根据 4.3.1~4.3.3 节的讨论，我们可以对 4.2.1 节所提出的问题作出如下回答：

第一，与基于数据流的方法相比，基于上下文的方法往往更加有效。描述性分析的结果表明 HC 度量和 SCC 度量的结果均符合 *Django* 中包的实际情况，所以我们可以推断基于上下文的方法能够适用于 Web 框架之类的程序库。这进一步论证了包内数据耦合不足以挖掘包的语义，从而导致基于数据流的方法在很多情况下失效。

第二，HC 度量的可能阀值为 0.47，而其余包内聚性度量方法的阀值还有待进一步研究。一方面，HC 度量的决策树具有良好的分类效果。另一方面，该决策树在被度量包的 HC 值大于 0.47 时将其归为高内聚。在未来工作中，我们将在更丰富的实验对象上使用多种的分类模型以研究各种包内聚性度量方法的阀值。

第三，目前的实验结果显示 SCC 度量稍优于 HC 度量，但不占绝对优势。首先，描述性分析的结果表明 SCC 度量能帮助开发者发现高数据耦合但却不内聚的包。事实上，在使用 SCC 度量时，我们应对低数据耦合包和高数据耦合包采用不同标准。对于低数据耦合包，内聚包的 SCC 值应接近 0.50。而对于高数据耦合包，若 SCC 度量值接近 1，则该包是内聚的；若 SCC 度量值接近 0.50，则需要开发者或其他度量方法进行进一步分析。这是由于虽然这种包具有紧密的内部数据耦合，但可能在语义上并不内聚。其次，Mann-Whitney U 检验的结果表明 HC 度量对于客户集的敏感性较高，而 SCC 度量在 $k_1 = 0.5$ 且 $k_2 = 0.5$ 的情况下，对于客户集的变化不敏感。这表明 SCC 度量更容易获得稳定而有效的度量值，而不必依赖于客

户集的选择。SCC 度量具备以上优势的可能因素是它全面地利用了包上下文。然而，决策树分析的结果却显示 HC 度量能有效评估包的内聚性，但 SCC 度量却不能。我们推测其原因可能是在 k_1 和 k_2 均为 0.5 的情况下，SCC 度量过多地依赖于包内数据耦合。此外在相同条件下，SCC 度量的计算复杂度远高于 HC 度量。最后，SCC 度量需要用户设定参数，而 HC 度量不需要。在未来工作中，我们需要更为广泛的实验以进一步研究 k_1 和 k_2 的取值对 SCC 度量的影响。

第四，充足客户集可以有效保证基于上下文的方法获得稳定而有意义的度量值。一方面，Mann-Whitney U 检验的结果显示 SCC 度量在 $k_1 = 0.5$ 且 $k_2 = 0.5$ 的情况下，对于客户集的变化不敏感。另一方面，描述性分析和决策树分析的结果表明 HC 度量能够反映 *Django* 中各种包的内聚程度。因此，我们可以认为充足客户集可以保证 HC 值的有效性。此外，Mann-Whitney U 检验说明只要新客户不打破客户集中原有各种客户之间的平衡，HC 值不会发生显著变化。即是说，若客户集是充足的，则我们不需要增加更多的客户来计算 HC 值。

第五，充足客户集表达了被度量包的实际使用情况。Mann-Whitney U 检验的结果表明术语“充足”表示客户集应覆盖被度量包的所有典型的使用场景，且每种客户在客户集中的所占比例应与其对应使用场景的实际出现频率一致。

综上所述，实验研究所得的结论与第 3 章的理论结论基本一致。然而，也存在一些差异。首先，虽然 SCC 度量更加全面地利用了上下文的信息，但它对 HC 度量的优势不明显。在客户集充足的情况下，HC 度量也能获得稳定而有意义的度量值。其次，充足客户集在保证 HC 值的稳定性上存在条件，即新客户不能打破客户集中已有客户之间的平衡。因此，实验研究是理论研究的有力补充，可以帮助我们完善理论。

4.4 本章小结

根据所使用的信息，包内聚性度量可被分为基于数据流的方法和基于上下文的方法。为研究这些方法在大型程序库中的效果，我们对开源 Python Web 框架 *Django* 进行了实验研究。首先，讨论了 Python 包的内聚性度量方法，着重指出了它们针对动态语言的特性所作出的调整。然后，讨论了实验的设计，提出了实验研究所需回答的问题。最后，对实验结果进行了分析，并获得了一批有价值的结论：(1) 基于上下文的方法比基于数据流的方法更加有效；(2) HC 度量的可能阈值为 0.47；(3) 与 HC 度量相比，SCC 度量具有一定的优势；(4) 充足客户集反映了被度量包的实际使用情况，能保证基于上下文的方法获得稳定而有效的度量值。因此，作为理论研究的重要补充，实验研究具有重大而深远的意义，能够指导我们发展理论。

第五章 度量驱动的包结构重构技术

由前面章节的讨论可知，包结构的质量对于大规模软件的开发和维护有重要影响。然而，软件的复杂性随着软件规模呈非线性增加。在没有工具辅助的情况下，开发者难以通过阅读代码来发现包结构的设计缺陷，并实施合理重构。

为解决这一问题，本章提出了度量驱动的包结构重构框架。基于该框架，提出了包结构的自动化重构算法 *refactor_package*：首先通过稳定性、抽象性、内聚性和耦合性等度量指标来评估包结构的质量，识别出违反设计原则的包结构，然后针对识别结果调用相应的算法，最后度量重构后的系统以检查重构的有效性。实例分析表明该算法能够高效地改进包结构，从而帮助大规模软件开发者提高重构效率。

5.1 包结构的重构

根据前面章节的讨论，包结构在大规模软件的构造中具有重要作用。从物理结构上讲，包结构从很大程度上决定着开发团队的结构，其质量是影响团队开发效率的关键性因素之一。另外，包是软件重用和发布的单位，所以包结构直接关系着软件的可测试性、可重用性和可扩展性等。从逻辑结构上讲，包结构体现了软件的高层架构，因而高质量的包结构有利于提高软件的可理解性和可维护性等。

鉴于上述原因，我们应该通过重构在软件开发过程中控制包结构的质量。事实上，包结构重构既是物理重构的核心工作之一，也是逻辑重构的重要部分。然而，包结构重构往往涉及大量的类（组件）和函数，这给手工重构带来了较大的困难。一方面，单纯依赖人工评审来识别需要重构的包结构以及评估重构有效性的成本较高。开发者通常需要阅读大量的代码和文档来理解错综复杂的包结构，以发现设计缺陷和选择合适的重构方法。例如，*Django* 有 700 多个组件，而这些组件不仅具有复杂的关系，还属于不同的包。若不借助工具，理解它的包结构需要花费大量的时间和精力。另一方面，包结构重构要求开发者修改与被重构包相关的导入语句，如 C++中的 include 语句、Python 中的 import 语句等。而这些导入语句往往散布于数量庞大的类（组件）中。在这种情况下，手工重构不仅代价过高，而且还容易引发错误。例如，一次包结构重构要求修改 100 个类中的导入语句，但开发者却只修改了其中的 99 个。这样的遗漏可能会导致系统的构建失败或运行时错误。

为解决这一问题，我们提出了度量驱动的包结构重构框架，它分为三步：第一步，计算包的各种度量值以识别存在设计缺陷的包结构；第二步，根据缺陷类型选择合适的重构方法进行重构；第三步，度量重构后的包结构，从而检查重构的有效性。第 2 章给出了一批可对包结构进行优化的原子重构方法（重新组织包结构）。第 3 章和第 4 章研究了包内聚性度量方法。本章将通过包括内聚性在内的多种度量指标来实现上述重构框架的自动化。

根据度量驱动的包结构重构框架，我们给出了包结构重构算法 *refactor_package*。如图 5.1 所示，该算法首先通过度量值判断包结构是否满足一组给定的设计原则，然后依据规则 5.1~5.4 调用相应的处理算法 *extract_abstract_package*、*divid_package*、*merge_coupled_package* 和 *reoganize_package*（5.1~5.3 节将对这些规则和算法进行详细讨论），最后计算被修改包的度量值来考察重构是否有效：若包结构的质量有一定的改善，则本次重构是有效的。

对于度量方法 M 和相应的度量值 v，算法 *refactor_package* 通过 v 与 M 阈值的关系来评估包结构是否满足设计原则：若 v 高于（低于）或等于阈值，则认为被度量包满足 M 对应的设计原则；反之，若 v 低于（高于）阈值，则认为被度量包违反 M 对应的设计原则。例如由第 4 章可知，HC 度量的阈值为 0.47。因此，若被度量包的 HC 值为 0.20，则可以认为该包不满足高内聚原则。

在实际重构过程中，我们有三种途径获取 M 的阈值。第一种是大规模实验分析。例如，4.2 节通过实验研究得到了 HC 度量的阈值为 0.47。第二种是开发实践或理论研究。例如，McCabe 认为圈复杂度的阈

值为 10[103]。第三种是被度量系统中包的 M 值的平均数、数学期望或中位数等。这种方法的核心思想通过考察 M 值的分布情况来发现“异常值”。在很多情况下，“异常值”由不良设计引起。例如，若一个系统的 SCC 值的数学期望是 0.40，则我们可取 SCC 度量的阀值为 0.40。这种方法常用于开发者难以通过前两种方法来获得 M 的阀值的情况。

本章主要考虑稳定抽象原则（Stable Abstractions Principle，SAP）、高内聚原则（High Cohesion Principle，HCP）和低耦合原则（Low Coupling Principle，LCP）。这些原则分别对应于图 5.1 中的算法 *is_satisfy_sap*、*is_satisfy_hcp* 和 *is_satisfy_lcp*。由于我们希望用较少的重构代价获得较好的结果，所以这些原则的考察顺序与相应的重构复杂度一致：对于设计原则 SP，若重构违反 SP 的包结构的复杂度较低，则先考察 SP；反之，若重构违反 SP 的包结构的复杂度较高，则后考察 SP。此外，为了提高重构效率，在检查重构有效性时，我们仅要求单个包的度量结果有所改善。

```
def refactor_package(p, sys, cn_m, cl_m, sap_m, cn_t, cl_t, sap_t):
    '''p: the package requiring refactored'''
    '''sys: the system where p exists'''
    '''cn_m, cl_m, sap_m: the metric respectively corresponding to hcp, lcp and sap'''
    '''cn_t, cl_t, sap_t: the thresholds of cn_m, cl_m and sap_m'''
    results = set([])                # results is modified packages by the refactoring
    # select proper rule to handle in terms of the metrics
    # Rule 5.1: handle SAP
    if not is_satisfy_sap(p, sap_m, sap_t) and is_contains_abstract_class(p):
        results = extract_abstract_package(p, sys)

        for r_p in results:
            if is_satisfy_sap(r_p, sap_m, sap_t):
                sys = sys.union(results).difference(p)
                break
    # Rule 5.2: handle HCP
    elif not is_satisfy_hcp(p, cn_m, cn_t) and is_satisfy_lcp(p, cl_m, cl_t):
        results = divid_package(p, sys, cn_m, cn_t)

        for r_p in results:
            if is_satisfy_hcp(r_p, cn_m, cn_t):
                sys = sys.union(results).difference(p)
                break
    # Rule 5.3: handle LCP
    elif not is_satisfy_lcp(p, cl_m, cl_t) and is_satisfy_hcp(p, cn_m, cn_t):
        results = merge_package(p, sys, cl_m)

        for r_p in results:
            if is_satisfy_lcp(r_p, cl_m, cl_t):
                sys = sys.union(results).difference(p)
                break
    # Rule 5.4: similar to Rule 5.3, handle lcp
    elif not is_satisfy_lcp(p, cl_m, cl_t) and not is_satisfy_hcp(p, cn_m, cn_t):
        results = reorganize_package(p, sys, cn_m, cn_t, cl_m)

        for r_p in results:
            if is_satisfy_lcp(r_p, cl_m, cl_t) and is_satisfy_ hcp(r_p, cn_m, cn_t):
                sys = sys.union(results).difference(p)
                break

    return sys
```

图 5.1　包结构重构算法 refactor_package

5.2 面向稳定抽象原则的重构

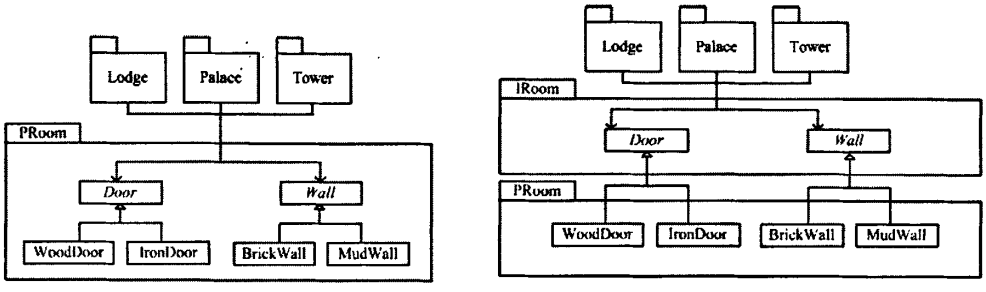
稳定抽象原则是指包的抽象性应该和其稳定性一致。这里，抽象性是指包的抽象程度；稳定性是指修改包所需要的工作量。在图 5.1 中，算法 *is_satisfy_sap* 可以通过文献[45]所提出的 D 度量来评估包是否满

足稳定抽象原则：给定包 p，若点(I(p), A(p))到直线 I + A = 1 的距离 D 小于或等于阈值，则满足稳定抽象原则；反之，若点(I(p), A(p))到直线 I + A = 1 的距离 D 大于阈值，则不满足稳定抽象原则。其中，I 度量和 A 度量分别用于计算抽象性和稳定性。本章设置 D 度量的阈值为 0.20。

规则 5.1 若包 p 违反稳定抽象原则，且含有抽象类$\{c_1,\ldots, c_s\}$，则采用提取抽象包将$\{c_1,\ldots, c_s\}$提取出来封装成一个新包 p'。

规则 5.1 将原有对 p 的部分依赖转移到 p'上，有助于重构后的包满足抽象稳定原则。根据 D 度量，转移依赖可以提高 I(p)，从而有利于降低 D(p)。更重要地，p'是抽象包，对它的依赖不会违反稳定抽象原则。

例如在图 5.2(a)中，D(*PRoom*) = 0.47 代表包 *PRoom* 违反稳定抽象原则。一方面，它封装了“门”和“墙”，被众多客户依赖，具有较高的稳定性。另一方面，它包含大量的具体类（实现一个特定的系统规约），抽象性较低。鉴于 *PRoom* 包含抽象类，我们依据规则 5.1 采用提取抽象包来进行重构。重构结果见图 5.2(b)。在图 5.2(b)中，由于 D(*IRoom*) = D(*PRoom*) = 0，所以包 *IRoom* 和 *PRoom* 均满足稳定抽象原则。



(a) 重构前的包 PRoom　　(b) 重构后的包 IRoom 和 PRoom

图 5.2 面向稳定依赖的重构

基于规则 5.1，我们提出了提取抽象包的实现算法 *extract_abstract_package*。如图 5.3 所示，该算法首先遍历 p 中的类，并将其中的抽象类添加到抽象包 *interface* 中，然后计算 p 和 *interface* 的差集，得到重构后的包 *results*，最后遍历 p 所在系统 *sys* 的类间数据依赖图，修改受影响类的导入语句。由于该算法的主要消耗在于遍历类间数据依赖图，所以它的时间复杂度为 O(n)。其中，n 为 *sys* 中类的数目。

```
def extract_abstract_package(p, sys):
    '''p: the package violating sap and containing abstract classes'''
    '''sys: the system where p exists'''
    results = set([])       # results: the refactored and new packages
    interface = set([])     # interface: the abstract or interface package
    for c in p:
        if is_abstract(c):
            interface.add(c)

    results.add(interface)
    results.add(p.difference(interface))
    modify_affected_classes(p, interface, sys) # modify the classes affected by the refactoring

    return results
```

图 5.3 算法 extract_abstract_package

5.3 面向高内聚原则的重构

高内聚原则要求包具有较高的内聚性。在图 5.1 中，算法 *is_satisfy_hcp* 可以采用第 3 章所提出的 HC 度量和 SCC 度量来计算包的内聚性。由第 4 章的讨论可知，SCC 度量在 k_1 和 k_2 均为 0.5 的情况下对低数据耦合包和高数据耦合包有不同的阈值，本章分别设它们的阈值为 0.36 和 0.60。另外，第 4 章还得出了

HC 度量的阈值为 0.47。

规则 5.2 若包 p 违反高内聚原则，但满足低耦合原则，则采用提取包将 p 拆分为一组内聚性较高的包 p_1、…、p_t。而且，p_1、…、p_t之间的耦合性不应该过高。

规则 5.2 将内聚性较低的包 p 转化为是一组内聚性较高的包 p_1、…、p_t，从而使得重构后的包满足高内聚原则。此外，p_1、…、p_t 之间的低耦合有助于这些包满足低耦合原则。事实上，内聚性和耦合性是包结构设计的一对平衡力，提高包的内聚性可能会导致包之间的耦合性过高[45]。特别地，若 p 还违反了稳定抽象原则且由具体类组成，规则 5.2 可以降低包的稳定性，从而有助于包满足稳定抽象原则。这是因为提取包将原有对 p 的依赖分散到 p_1、…、p_t上。

根据规则 5.2，一个合理的拆分思路是使得 p_i 满足高内聚原则，但 p_i 和 p_j 的并集违反高内聚原则。其中，i，j∈{1,.., t}，且 i≠j。基于这种思路，我们提出性质 5.1 来指导 p 的拆分。

性质 5.1 给定包 $p=\{c_1,\ldots,c_n\}$，且 cn_m(p) < cn_t，$\Pi=\{p_1,p_2,\ldots,p_t\}$是 p 的合理划分当且仅当 Π 满足：

$$\forall p_i\in\Pi:\ \text{cn\_m}(p_i)\geq \text{cn\_t}$$

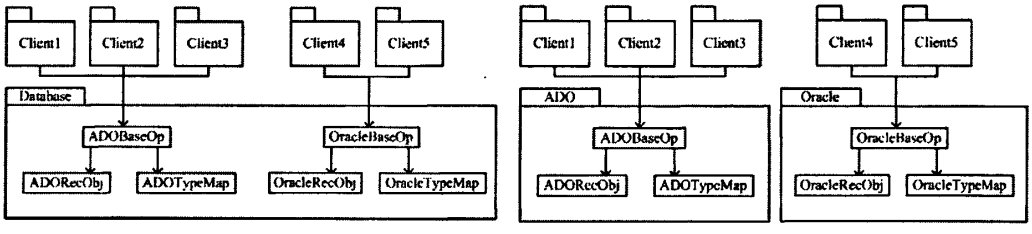
和

$$\forall p_i,p_j\in\Pi\wedge i\neq j:\ \text{cn\_m}(p_i\cup p_j)<\text{cn\_t}$$

其中，cn_m 是给定的包内聚性度量方法；cn_t 为 cn_m 的给定阈值；t > 1。

由性质 5.1 可知，p 的合理划分使得重构后的包符合规则 5.2 的要求。此外，虽然 cn_m 可以是任意的包内聚性度量方法，但第 3 章和第 4 章的研究表明与基于数据流的包内聚度量方法相比，基于上下的包内聚性度量方法往往更加有效。因此，在实际重构过程中，我们主要选择 HC 度量和 SCC 度量来获取包的合理划分。具体选择何种方法需根据被度量系统的实际情况与这两种方法的适用情况和性质来决定。例如，对于没有客户的包，我们应选用 SCC 度量，因为 HC 度量在无客户的情况下无定义。

以图 5.4(a)为例，包 *Database* 负责数据库相关的任务，可以处理 ADO 数据库和 Oracle 数据库。其中，类 *ADOBaseOp* 包装了 ADO 数据库基本操作，如查询、插入等；类 *ADORecObj* 负责 ADO 数据库中的记录和领域对象之间的相互转化；类 *ADOTypeMap* 是 ADO 数据库的数据类型和编程语言的数据类型之间的映射。与 Oracle 数据库相关的类具有相似的功能。由图 5.4(a)可知，HC(*Database*) = 0.22，即 *Database* 的大多数客户仅使用其中一种数据库，因此它不满足共同重用原则，内聚性较低。基于此，我们可利用提取包将它拆分为两个包 *ADO* 和 *Oracle*。重构结果如图 5.4(b)所示。在图 5.4(b)中，HC(*ADO*) = HC(*Oracle*) = 1，这说明重构后的包均满足共同重用原则，内聚性较高。此外，HC(*ADO*∪ *Oracle*) = 0.22，即 *ADO* 和 *Oracle* 的并集违反高内聚性原则。所以，本次重构符合规则 5.2 的要求。



(a) 重构前的包 Database　　**(b)** 重构后的包 ADO 和 Oracle

图 5.4 面向高内聚原则的重构

定理 5.1 给定包 $p=\{c_1,\ldots,c_n\}$，且 cn_m(p) < cn_t，则存在划分 $\Pi=\{p_1,p_2,\ldots,p_t\}$满足性质 5.1 的条件。其中，符号 cn_m、cn_t 和 t 的含义与性质 5.1 中的相同。

证明：设置 Π 的初始值为$\{\{c_1\},\ldots,\{c_n\}\}$，此时，Π 的任意子包的 cn_m 值为 1，显然大于等于 cn_t，即满足性质 5.1 的第一个条件，则

(1) 若 Π 符合性质 5.1 的第二个条件，则该定理得证。否则，转到(2)。

(2) 若存在 p_i，$p_j \in \Pi$，满足 cn_m($p_i \cup p_j$)≥cn_t，则合并 p_i 和 p_j，转到(1)。

由(1)-(2)可知，当合并过程停止时，Π 即为合理划分。又由于包中类的数目有限，合并过程一定能在有限步停止。 ∎

基于定理 5.1，我们提出了算法 *get_reasonable_division*，可以计算低内聚包的合理划分。如图 5.5 所示，该算法使用自底向上的方式：首先将每个类作为一个子包，其次逐步合并各个子包，直到所有子集满足性质 5.1。这里，合并采用贪心策略。换句话说，给定子包 *division[i]*，与它合并的包 *division[max_j]*应使得 cn(*division[i]*∪ *division[max_j]*)为所有 cn(*division[i]*∪ *division[j]*)中的最大值。

不难看出，算法 *get_reasonable_division* 的主要时间消耗为子集的合并和删除操作以及计算 cn_m 值。合并和删除的时间复杂性分别是 O(mα (m, m))和 O(m)。其中，m 是 p 中类的数目，α (m, m)是 Ackermann 函数的逆函数[104]。由于 α (m, m)增长极慢，因此 O(mα (m, m))可近似为 O(m)。而对于计算 cn_m 值，不同的内聚性度量方法具有不同的时间复杂度。HC 度量的计算需要依赖于 p 所在系统 *sys*，时间复杂度为 O(n)。SCC 度量同样依赖于 *sys*，其主要时间消耗为集合的并集和差集操作。由于差集操作的时间复杂度为 O(n^2)，所以计算 SCC 值的时间复杂度为 O(n^2)。

```
def get_reasonable_division(c, p, cn_m, cn_t):
    '''c: the initial class used in the division process'''
    '''p: the non-cohesive package'''
    '''cn_m: the given cohesion metric'''
    '''cn_t: the threshold of cn_m'''
    '''the algorithm is initial sensitive, so we should set the initial class'''
    original = set([]).union(p)                          # original: the original package
    division = [set([c])]                                # division: the division of p
    original.remove(c)
    division += [set([ci]) for ci in original]
    i = 0
    length = len(division)
    while i < length:
        j = i + 1
        max = cn_m(division[j])
        max_j = j
        # Find j satisfying that the union of division[i] and division[max_j] is maximal
        while j < len(division):
            if cn_m(division[i].union(division[j])) > max:
                max = cn_m(division[i].union(division[j]))
                max_j = j
            j = j + 1

        # If the union set of division[i] and division[max_j] is cohesive,
        # then merge division[i] and division[max_j]
        if cn_m(division[i].union(division[max_j])) >= cn_t:
            division[i] = division[i].union(division[l])
            division.remove(division[cn_m])
        else:
            i = i + 1

    return division
```

图 5.5 算法 get_reasonable_divsion

下面以图 5.4(a)为例展示算法 *get_reasonable_division* 使用 HC 度量对包 *Database* 进行划分的过程，其详细过程如表 5.1 所示。在前 2 次迭代中，HC(*division[0]*∪ *division[1]*)均为 1，因此，逐步合并 *division[0]* 和 *division[1]*，得到第 3 行的结果。在第 3 迭代中，HC(*division[0]*∪ *division[1]*) = HC(*division[0]*∪ *division[2]*) HC(*division[0]*∪ *division[3]*) = 0.33，小于 0.47，不进行合并。在最后 2 次迭代中，HC(*division[1]*∪ *division[2]*) = 1，所以合并这两个子包，得到第 5 行的结果。

表 5.1 图 5.4(a)中包 Database 的划分过程

迭代	division	HC	i, max_j
0	{{ADOBaseOp}, {ADORecObj}, {ADOTypeMape}, {OracleBaseOp}, {OracleRecObj}, {OracleTypeMape}}	HC(division[0]∪division[1]) = 1.00	i = 0 max_j = 1
1	{{ADOBaseOp, ADORecObj}, {ADOTypeMape}, {OracleBaseOp}, {OracleRecObj}, {OracleTypeMape}}	HC(division[0]∪division[1]) = 1.00	i = 0 max_j = 1
2	{{ADOBaseOp, ADORecObj, ADOTypeMape}, {OracleBaseOp}, {OracleRecObj}, {OracleTypeMape}}	HC(division[0]∪division[1]) = 1.00	i = 0 l = 1
3	{{ADOBaseOp, ADORecObj, ADOTypeMape}, {OracleBaseOp}, {OracleRecObj}, {OracleTypeMape}}	HC(division[0]∪division[1]) = 0.33 HC(division[0]∪division[2]) = 0.33 HC(division[0]∪division[3]) = 0.33	i = 0 max_j = 2
4	{{ADOBaseOp, ADORecObj, ADOTypeMape}, {OracleBaseOp, OracleRecObj}, {OracleTypeMape}} {OracleBaseOp, OracleRecObj, OracleTypeMape}}	HC(division[1]∪division[2]) = 1.00	i = 1 max_j = 2
5	{{ADOBaseOp, ADORecObj, ADOTypeMape}, {OracleBaseOp, OracleRecObj, OracleTypeMape}}	HC(division[1]∪division[2]) = 1.00	i = 1 max_j = 2

然而由图 5.5 可知，算法 *get_reasonable_division* 是初始敏感的，意即算法结果可能会随着输入参数 *c* 的变化而产生波动。因此，为了获得一个相对较优的合理划分，从一定程度上避免局部最优解，我们提出了算法 *divid_package* 来处理规则 5.2。图 5.6 给出了该算法的详细描述：依次将 *p* 中的每个类传递给算法 *get_reasonable_division* 来获取 *p* 的合理划分，而最终结果 *best_division* 是这些划分中使得所有子包的平均 cn_m 值达到最大的划分。由图 5.6 可知，算法 *divid_package* 的时间复杂性为 O(m*b)。其中，b 为算法 *get_reasonable_division* 的时间复杂性，m 是 *p* 中类的数目。

同样以图 5.4(a)为例，算法 *divid_package* 将包 *Database* 中的类依次传递给算法 *get_reasonable_division*。在 6 次迭代后，获得与表 5.1 相同的结果。该结果与图 5.1(b)相同。因此，算法 *divid_package* 能够有效地提高系统的包内聚程度。

```
def divid_package(p, sys, cn_m, cn_t):
    '''p: the non-cohesive package'''
    '''sys: the system where p exists'''
    '''cn_m: the given cohesion metric'''
    '''cn_t: the threshold of cn_m '''
    max_avg = 0.0       # sum: the sum of the value of cn_m of the subpackage
    best_division = []  # division: the division of a package
    # Different c may lead to different division
    # Find a reasonable division whose sum of the value of cn_m of the subpackage is maximal
    for c in p:
        division = get_reasonable_division(c, p, cn_m, cn_t)
        sum = 0.0
        for sp in division:
            sum += cn_m(p)

        avg = sum / len(division)
        if avg > max_avg:
            max_avg = avg
            best_division = division

    modify_affected_classes(p, best_division, sys)

    return best_division
```

图 5.6 算法 divid_package

5.4 面向低耦合原则的重构

低耦合原则要求包具有较低的耦合性。所谓包耦合性是指包相互协作以实现系统功能的紧密程度。在实际开发中，高内聚低耦合一直是指导开发者进行软件设计的重要启发式规则之一。然而，现有的模块耦

合性度量方法主要关注类[91]，难以支持包耦合性的评估。为此，我们在类带权交互图的基础上，提出了带权关系耦合性（Weighted Relational Coupling, WRC）。

定义 5.1 给定包的集合 $P=\{p_1,...,p_n\}$，它的类带权交互图 WIG 是一个无向图 $WIG(P)=(V(P), E(P))$。其中，$V(P)=\{c \mid c\in p_i \wedge p_i\in P\}$，$E(P)=\{(c_1,c_2)\in V(P)\times V(P) \mid Wgt(c_1,c_2)>0\}$。其中，$Wgt(c_1,c_2)$的计算方法与定义 3.8 相同。

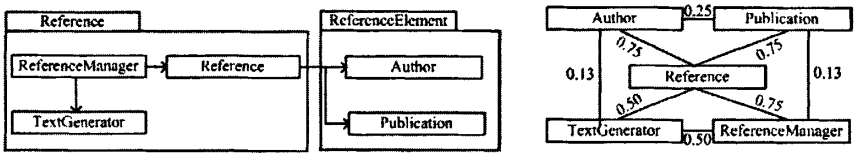
定义 5.2 给定包的集合 P，$p\in P$，有

$$WRC(p,P)=\begin{cases}\dfrac{\sum_{c_i\in p\wedge c_j\in p'\wedge p'\in P\setminus p} Wgt(c_i,c_j)}{|p|\sum_{p'\in P\setminus p}|p'|} & if\ \ P\neq\{p\}\\ 0 & if\ \ P=\{p\}\end{cases} \tag{5.1}$$

其中，符号“\”表示差集。

当 $P=\{p\}$时，P 仅含一个包，没有包间耦合问题，所以我们设 $WRC(p,P)=0$。当 $P\neq\{p\}$时，若任意 $Wgt(c_i,c_j)=0$，则 $WRC(p,P)=0$；若任意 $Wgt(c_i,c_j)=1$，则 $WRC(p,P)=1$。因此，$WRC\in[0,1]$。

例如，图 5.2(a)给出了参考文献管理系统 *Ref*：类 *Reference* 是参考文献；类 *TextGenerator* 是参考文献列表生成器；类 *ReferenceManager* 是参考文献管理器；类 *Author* 代表作者；而类 *Publication* 表示刊物。图 5.2(b)是图 5.2(a)对应的 WIG。根据定义 5.2，$WRC(Reference, Ref)=WRC(ReferenceElement, Ref)=0.38$。



(a) 参考文献管理系统 Ref　　**(b)** WIG(Ref)

图 5.7 违反低耦合原则的包

由定义 5.1~5.2 可知，$WIG(p_i)$是 $WIG(P)$的由 $E(p_i)$导出的子图。即是说，定义 3.8 是定义 5.1 的特殊情况。此外由 WRC 度量和 SCC 度量的关系可知，内聚性和耦合性之间不是独立的。在有些情况下，高内聚可能导致高耦合，而低耦合可能引起低内聚。因此，包的设计需同时考虑两个内聚性和耦合性两个因素。

图 5.1 中的算法 *is_satisfy_lcp* 可以通过 WRC 度量来考察包是否满足低耦合原则。这里，我们设定 WRC 度量的阀值为 0.07。另外，与 SCC 度量类似，k_1 和 k_2 的默认值均为 0.5。

规则 5.3 若包 p 违反低耦合原则，但满足高内聚原则，则采用合并包将 p 与 p'合并为一个新包 p''。其中，p'是系统中与 p 的耦合程度最高的包。

规则 5.3 将包间的耦合性转化为包内的耦合性。更重要地，p'与 p 之间的高耦合有助于 p''满足高内聚原则。与规则 5.2 类似，规则 5.3 也考虑了内聚性和耦合性之间的平衡。否则，降低 p 的耦合性有可能导致 p''的内聚性过低。

以图 5.7(a)为例，包 *Reference* 违反低耦合原则，但满足高内聚原则。一方面，$WRC(Reference, Ref)=0.59$，因此它不满足低耦合原则。从语义上看，作者和刊物是参考文献的一部分，这使得类 *Author* 和类 *Publication* 与 *Reference* 之间存在紧密关联，从而导致包 *Reference* 和 *ReferenceElement* 之间具有较高的耦合性。另一方面，鉴于包 *Reference* 没有客户，我们使用 SCC 度量来计算它的内聚性，可得 $SCC(Reference)=0.58$。此外，$RRC(Reference)=0.33$ 表明它是高数据耦合包，对应的 SCC 度量的阀值为 0.55。因此，包 *Reference* 满足高内聚原则。事实上，该包用于“管理参考文献”，服务于一致的抽象。考虑到 *Ref* 只有两个包，我们根据规则 5.3 使用合并包将这两个包合并为一个新包 *Reference*。重构结果如图 5.8 所示。在重构后，只有一个包的系统显然满足低耦合原则。此外，$SCC(Reference)=0.38$ 代表该包满足高内聚原则。这是因为 $RRC(Reference)=0.33$ 代表它是低数据耦合包，则其相应的 SCC 度量的阀值为 0.36。

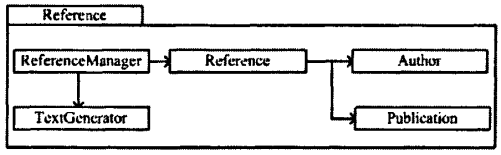


图 5.8 根据规则 5.3 重构 Ref

基于规则 5.3，我们提出算法 *merge_coupled_package*：首先查找系统 *sys* 中与 *p* 的耦合程度最高的包 *p'*，其次合并 *p* 和 *p'*，最后修改相应类的导入语句。图 5.9 给出了该算法的详细描述。由图 5.9 可知，算法的主要时间消耗在计算 cl_m 值上。由于 WRC 度量与 SCC 度量的计算复杂度相同，所以基于 WRC 的算法 *merge_coupled_package* 的时间复杂度为 $O(n^2)$。

```
def merge_coupled_package(p, sys, cl_m):
    '''p: the non-cohesive package'''
    '''sys: the system where p exists'''
    '''cl_m: the given coupling metric'''
    most_coupled = None  # most_coupled: the package that has the highest coupling with p
    max = 0.0
    # find most_coupled
    for pi in sys:
        if cl_m(p, set([p, pi])) > max:
            max = cl_m(p, set([p, pi]))
            most_coupled = pi

    results = set(p.union(most_coupled)) # merge the packages
    modify_affected_classes(p, best_division, sys)

    return results
```

图 5.9 算法 merge_coupled_package

然而，若 *p* 同时违反低耦合原则和高内聚原则，那么简单地合并两个高耦合的包会导致合并后的包仍然不满足高内聚原则。在这种情况下，我们可以依据规则 5.4 进行处理。

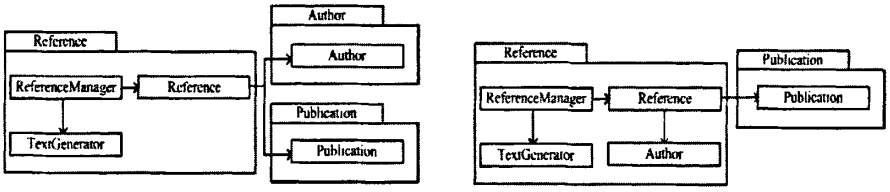
规则 5.4 若包 *p* 不仅违反低耦合原则，还违反高内聚原则，则首先采用提取包将 *p* 拆分为多个高内聚低耦合的包 p_1、...、p_t，然后通过合并包将 p_i 与 *p'*合并为一个新包 *p''*。其中，*p'*是系统中与 *p* 的耦合程度最高的包；而 p_i 是$\{p_1, p_2, ..., p_t\}$中与 *p'*的耦合程度最高的包。

在规则 5.4 的条件下，*p* 中仅有部分类与 *p'*之间的耦合性较高，否则 *p* 满足高内聚原则。这是由于类之间的关系往往是传递的。如在图 5.8 中，类 *ReferenceManager* 通过类 *Reference* 与类 *Publication* 和 *Author* 发生联系。在这种情况下，只合并 *p* 中与 *p'*耦合性较高的部分有助于 *p''*满足高内聚原则。此外根据规则 5.3，合并两个耦合性较高的包可以降低系统中包的耦合程度。

同样以图 5.7(a)为例，与包 *Reference* 类似，包 *ReferenceElement* 违反低耦合原则。此外，由于该包无包内数据依赖且 SCC 值为 0.25，所以内聚性较低。从语义角度，类 *Author* 和 *Publication* 分别是对作者和刊物的抽象，未服务于一个中心目标。根据规则 5.4，我们首先通过提取包将它拆分为包 *Author* 和 *Publication*。重构结果如图 5.10(a)所示。在图 5.10(a)中，包 *Author* 和 *Publication* 与包 *Reference* 之间的耦合性相同，可以任选其一进行合并。在本例中，我们选择包 *Author* 和 *Reference* 进行合并。图 5.10(b)给出了重构后的包结构。其中，RRC(*Reference*) = 0.25，SCC(*Reference*) = 0.44，则包 *Reference* 满足高内聚原则。此外，仅含一个类的包 *Publication* 显然内聚。因此，重构提高了包的内聚程度。此外，WRC(*Publication*, *Ref*) = 0.28，低于重构前的 0.59，意即重构降低了包的耦合程度。

然而，在图 5.10(b)中，包 *Publication* 的 WRC 值仍高于 WRC 度量的阈值，所以它违反了低耦合原则。其本质原因是代表“刊物”的类 *Publication* 和表达“参考文献”的类 *Reference* 之间有紧密的 has-a 关系。为此，我们根据规则 5.3 对采用合并包对包 *Reference* 和 *Publication* 进行合并。重构结果如图 5.8 所示。可

以看到，该结果与依照规则 5.3 所得的结果一致。即是说，对于给定系统，可能存在多种重构方式。在实际重构中，我们应选择最简单的重构方法。以图 5.7(a)的系统 *Ref* 为例，我们应采用图 5.8 而非图 5.10 的方式对它进行重构。



(a) 对图 5.7(a)使用提取包　　**(b)** 对图 5.9(a)使用合并包

图 5.10 根据规则 5.4 重构后的 Ref

基于规则 5.4，我们提出算法 *reorganize_package*。如图 5.11 所示，该算法首先查找系统 *sys* 中与 p 的耦合程度最高的包 p'，其次使用算法 *divid_package* 将 p 划分为多个内聚的子包 p_1、…、p_t，并通过算法 *merge_coupled_package* 合并 p'与 p_i，最后对受影响的类作出相应的修改。这里，p_i是 p_1、…、p_t中与 p'的耦合程度最高的包。由图 5.11 可知，该算法的时间复杂度为算法 *divide_package* 和 *merge_coupled_package* 的时间复杂性之和。

```
def reorganize_package(p, sys, cn_m, cn_t, cl_m):
   '''p: the non-cohesive package'''
   '''cn_m: the given cohesion metric'''
   '''cn_t: the threshold of cn_m '''
   '''cl_m: the given coupling metric'''

   most_coupled = None
   max = 0.0
   # Find the package that is most coupled with p
   for pi in sys:
      if cl_m(p, set([p, pi])) > max:
         max = cl_m(p, set([p, pi]))
      most_coupled = pi

   # divid_package(p,cn_m, cn_t) computes the cohesive subpackages.
   # Merge most_coupled and the package
   # that has the highest coupling in the cohesive subpackages
   results = merge_package(most_coupled, divid_package(p, cn_m, cn_t))
   modify_affected_classes(p, best_division, sys)

   return results
```

图 5.11 算法 reoganize_package

5.5 实例分析

本节通过图 5.12 来讨论如何通过算法 *refactor_package* 来提高包结构的质量，从而使其适应团队开发的需求，并降低系统的构建和发布代价。在图 5.12 中，包 *NetSimulator* 是图 2.4(a)中所有包的并集，有关它的功能的详细说明参见 2.1 节。

根据 2.1 节，图 5.12 的包结构存在一定的设计缺陷。首先，随着规模和复杂性的增加，系统可能需要更多的开发者。然而，仅含一个包的系统难以适应多人开发。再者，包 *NetSimulator* 同时封装了局域网以及局域网的组成要素（网络节点和通信链路），内聚性不高。

为解决这一问题，我们通过算法 *refactor_package* 来对包 *NetSimulator* 进行重构。鉴于系统中存在没有客户的包，我们使用 SCC 度量来计算包的内聚性。由图 5.12 可知，在包 *NetSimulator* 中，很多类主要通过上下文关系来进行联系，如类 *Comp* 和 *Hub*，因此我们将 SCC 度量的 k_1 和 k_2 分别设为 0.5 和 0.8，以突

出上下文关系在计算包内聚性中的作用。在这种情况下，SCC 度量不区分高数据耦合包和低数据耦合包，它的阀值可统一设定为 0.36。WRC 的参数设定与 SCC 度量相同，而它的阀值仍为 0.07。另外，我们仍然采用 D 度量来评估稳定抽象原则，其阀值不变。重构的详细过程如下：

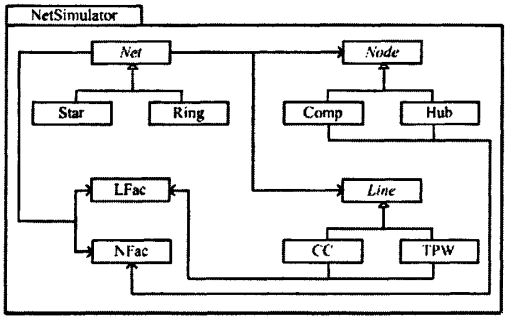
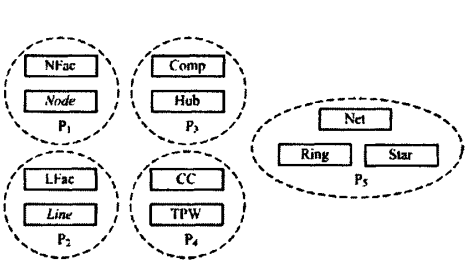
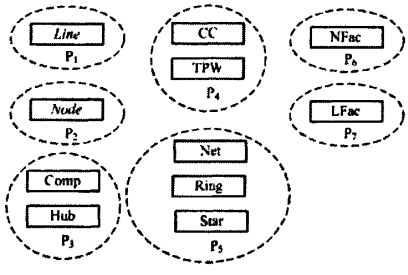


图 5.12　包 NetSimulator

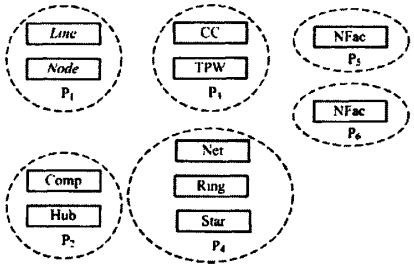
(1) 在图 5.12 中，SCC(*NetSimulator*) = 0.15，小于阀值 0.36，因此包 *NetSimulator* 违反高内聚原则。而系统仅有一个包，显然满足稳定抽象原则和低耦合原则。根据规则 5.2，调用算法 *divid_package* 将 *NetSimulator* 重构为多个内聚的子包。重构结果见图 5.13(a)。在重构后，SCC(P_1) = SCC(P_2) = SCC(P_3) = SCC(P_4) = 0.45，SCC(P_5) = 0.47。可以看到，包的内聚性得到了提高。



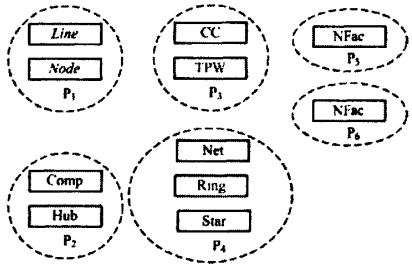
(a)　按照规则 5.2 调用算法 divid_package



(b)　按照规则 5.1 调用算法 extract_abstract_package



(c)　按照规则 5.3 调用算法 merge_package



(d)　按照规则 5.3 调用算法 merge_package

图 5.13　包 NetSimulator 的重构过程

(2) 在图 5.13(a)中，D(P_1) = 0.35，大于阀值 0.20，所以包 P_1 不满足稳定抽象原则。而且，它含有抽象类。根据规则 5.1，通过算法 *extract_abstract_package* 来提取抽象包，以消除不良包结构。对于包 P_2，算法 *refact_package* 采用相同处理方式。重构结果如图 5.13(b)所示。在重构后，D(P_1) = D(P_2) = 0。虽然 D(P_6) = D(P_7) = 0.71，但是由 2.3.2 节可知，这两个包中的类 *NFac* 和 *LFac* 被完全隔离的，易变性较低，不会给开发工作带来痛苦。因此，这次重构有效改善了包结构的质量。

(3) 在图 5.13(b)中，抽象包与系统之间的耦合仅由它们的客户引起。根据稳定抽象原则，对抽象包的依赖通常不会引起软件开发的痛苦。因此，为避免在第二步生成的抽象包与具体包合并，我们将算法 *refactor_package* 的输入参数 *sys* 设为{P_1, P_2, P_3, P_4, P_5, P_6}。在这种情况下，WRC(P_1, *sys*) = 0.08，大于阀值 0.07，即包 P_1 违反低耦合原则，但满足高内聚原则。而且，P_2 与 P_1 的耦合性最高。依据规则 5.3，使用算法 *merge_coupled_package* 合并两个包。重构结果如图 5.13(c)所示。在图 5.13(c)中，WRC(P_1, *sys*) = 0.06，因而，本次重构降低了包的耦合性。

(4) 与第三步类似，在图 5.13(c)中考察 P_5 时，我们不考虑抽象包 P_1，即 sys = {P_2, P_3, P_4, P_5, P_6}。此时，WRC(P_5, *sys*) = 0.09，所以包 P_5 违反了低耦合原则，但满足高内聚原则。而且，P_6 与 P_5 的耦合性最高。根据规则 5.3 调用算法 *merge_coupled_package* 合并 P_5 和 P_6。图 5.13(d)给出了重构结果。在图 5.13(d)中，WRC(P_5, *sys*) = 0.04，这表明本次重构使得包满足低耦合原则。

上述重构过程表明算法 *refactor_package* 可以有效地提高包结构的质量，从而有助于提高包结构重构的效率。图 5.13(d)的重构结果与图 2.4(c)的手工重构结果一致。由 2.3.3 节可知，在类 *Net* 没有客户的情况下，图 5.13(d)的包结构满足稳定依赖原则、内聚性原则以及耦合性原则，能够较好地适应团队开发的需求。即是说，重构还降低了软件的构建、重用和发布代价。

5.6 相关工作

为提高重构效率，研究者对度量驱动的重构技术进行了深入研究，获得了一批有价值的成果。据我们所知，这些成果主要集中于类和函数[91]。

在面向对象度量领域，S. Demeyer 等讨论了通过面向对象度量，如 NOM[105]等，选择合适重构方法的启发式规则[106]。F. Simon 等提出了距离驱动的重构，能通过类成员之间的距离来识别与移动类成员、提取类以及内联类等重构方法相关的不良类设计[63]。Y. Zhou 提出了一种改进的类内聚性度量 ACBMC，并研究了 ACBMC 度量在类重构上的应用[27]。

此外，M. Balazinska 等利用克隆代码识别技术定位重复代码，并提出了相应的重构方法[61]。C. Zhang 等提出了一种利用 AOP 重构中间件的方法。该方法首先利用度量识别出横切关注以及分析中间件中横切关注的散布情况，然后通过方面封装横切关注[65]。

虽然上述工作难以直接支持包结构的重构，但为我们的研究提供了有益的思路。此外，类（组件）是构成包的基础，类（组件）的设计缺陷有可能导致包的设计缺陷。因此，在包结构重构难以达到重构目标的情况下，我们应首先对类（组件）进行重构以消除类的设计缺陷。

5.7 本章小结

大规模软件的复杂性给包结构重构带来了较大的困难。单纯的手工重构不仅效率低下，而且容易引入错误，难以适应软件开发的需求。为此，本章提出了度量驱动的包结构重构框架。基于该框架，还提出了包结构的自动化重构算法 *refactor_package*：首先通过度量识别需要重构的包结构，然后根据启发式规则选择合适的重构方法进行重构，最后再次通过度量评估重构的有效性。实例研究表明该算法能够显著地提高重构效率，从而有效地弥补了手工重构的不足。

今后，我们将在以下几个方面对度量驱动的包结构重构技术进行深入研究。首先，提出更多针对包质量的度量指标，如易变性等，从而为从多角度评价包结构提供支持。其次，改进现有的度量方法。由上述讨论可知，度量方法的优劣直接影响着算法 *refactor_package* 的效果。最后，通过聚类分析算法完善现有的包划分算法 *divid_package*。

第六章 Python 程序的重构正确性评估方法

Python 是一种功能强大的动态类型语言，被广泛应用于各种大型软件的开发，如 Web 应用、科学计算等[100, 107-111]。然而，它的动态类型系统给类型缺陷的检查带来了困难，不利于评估重构的正确性。为解决这一问题，本章对 Python 程序的重构正确性评估方法进行了深入讨论。首先，结合 Python 语言的特征定义了一组与动态语言相关的基本术语。其次，分析了在 Python 程序的重构正确性评估中存在的问题，并通过 C++0x Concept 对如何通过 concept 表达结构类型系统的约束进行了讨论。在此基础上，提出了 Python 的类型约束系统 PyConcept，可以有效地表达 Python 基于结构一致性的类型约束。基于 PyConcept，提出了 PyConcept 的构建算法 *generate_concept*，它通过分析函数参数在函数体中的使用方式来生成这些参数的类型约束。进一步地，还提出了 PyConcept 的检查算法 *check_func_call*，能够高效地定位类型缺陷，从而弥补现有方法在检查 Python 程序重构正确性方面的缺陷。

6.1 基本术语

许多技术术语在不同上下文中有特定的含义。以“对象”为例，在 C++中，对象一般指由类创建的实例。而在 Python 中，对象含义的更为广泛。诸如整数、类、自由函数、模块等均为对象。为准确讨论 Python 程序的相关问题，本节对对象、名字、动态类型等基本术语进行定义。图 6.1 给出了一段涵盖这些基本术语的 Python 代码，具体说明了下列定义的含义：

- 对象是一块内存，有地址、类型和值三方面的特征[110]。
- 名字是对一个对象的称谓。一个对象可以只有一个名字，也可以没有名字或有多个名字。而动态绑定就是在运行时将对象与名字联系起来。在本章中，对象 *object* 特指名字 *object* 所指向的对象。如图 6.1 中，行 41 生成了一个 *Parrot* 类型的对象 *parrot*，行 31 的赋值语句使得名字 *flyables* 和 *collection* 同时指向相同对象，从而形成别名关系。
- 函数是指可被调用的对象，即支持函数调用运算符“*()*”的对象。Python 提供了两种定义函数的方式。第一种是以关键字 def 直接定义，如图 6.1 的函数 *Parrot.fly* 和 *fly_all*。第二种是通过定义特殊函数 *__call__* 来支持函数调用运算符*()*。以图 6.1 为例，类 *Ostrich* 定义了函数 *__call__*，所以它的实例 *ostrich* 支持函数调用运算符*()*，是一个函数。特别地，成员函数是函数的一种，它定义于类中，且第一个参数是指向该类实例的自引用。
- 数据是与函数相对的概念，指不能被调用的对象，如图 6.1 中的对象 *Bird*、*flight* 等。
- 属性分为数据属性和函数属性，是构成对象的基本成分。一般通过属性引用运算符“.”定义或访问，形如 *object.attribute*。其中，数据属性是指属性名指向一个数据的属性。相对地，函数属性指该属性对应一个函数。从属性的角度，对象可看作从属性名到属性值（对象）的字典。如图 6.1 中，类 *Bird* 的实例可看成字典{“name” : a string object, “__init__” : a function object}。其中，对象 *name* 是数据属性，对象 *__init__* 是函数属性。因此，定义属性可看作为字典中特定的名字定义值；访问属性则是在字典中查找特定的名字，并返回对应的值。在 Python 中，对象可以利用三个特殊函数 *__getattr__*、*__setattr__* 和 *__delattr__* 在运行时修改它对应的字典，以控制引用运算符的动态语义。
- 操作包括函数、运算符和控制流语句。Python 会将运算符或控制流语句转化为调用一组函数。例如，语句 *for* elem *in set: for_body* 会依次调用 *iter = set.__iter__()*、*elem = iter.next()* 和 *for_body*。所以，运算符和控制流语句可以统一表达为函数调用：*operation(args)*。例如，*1+2* 可以表示为 *__add__(1, 2)*，*for elem in set: for_body* 表示为 *__for__(set, elem, for_body)*。
- 对象的类型由对象所支持的操作决定，可在运行时修改。即是说，若两个对象在任意操作上产生相同的效果，则它们的类型相同。例如，判断对象 *i* 是否为整型的标准是：(1) 所有可应用于整型对象的

操作均可应用于 *i*，如四则运算；(2) 相同操作应返回相同结果，如 *i+1* 应返回比 *i* 大 1 的整数。

- 类是用户自定义类型，为创建对象提供原型或蓝图。用户可以以类为原型生成任意数量的对象。而这种根据类生成的对象被称为实例。由于对象支持的操作可以在运行时修改，Python 对象的类和类型常常不一致。以图 6.1 的行 51 为例，虽然对象 *ostrich* 是类 *Ostrich* 的实例，但它支持 *Ostrich* 未定义的操作 *__len__*，因而它的类型并非 *Ostrich*。
- 结构一致性是指若对象 *x* 和 *y* 支持相同的操作（具有一致的语法结构），则 *x* 与 *y* 可以互换。换句话说，任何可使用 *x* 的地方也可使用 *y*；反之亦然。例如，在图 6.1 中，对象 *ostrich* 支持操作 *ostrich.name* 和 *len(ostrich)*。因此，它与 *"person"* 对象具有一致的语法结构，可作为第二个实参传递给函数 *Parrot.talk*。这种基于结构一致性进行类型检查的类型系统被称为结构类型系统。在很多情况下，结构类型系统将语法结构的一致性视为语义层面的一致性。
- 动态类型是指对象类型在运行时决议或类型检查在运行时进行。它的典型表现形式为对象支持的操作可以在运行时发生变化。例如，在图 6.1 中，行 51 使得对象 *ostrich* 能支持内建函数 *len*，从而改变了 *ostrich* 的类型。

```
1  class Bird:
2      def __init__(self, name):
3          self.name = name

4  class Parrot(Bird):
5      '''A parrot can fly and talk'''
6      def fly(self):
7          self.talk('%s is flying!'
8                    %self.name)

9      def talk(self, content):
10         print "length: %d/n"
11               % len(content)
12         print content

13 class Ostrich(Bird):
14     '''An ostrich only can run'''
15     def run(self):
16         print "I am runnig!"

17     def __call__(self): pass

18 class Flight:
19     '''A flight has no and line'''
20     '''which can fly and talk'''
21     def __init__(self, no, line):
22         self.no = no
23         self.line = line

24     def fly(self):
25         print (self.line.start,
26                self.line.end)

27     def talk(self, person):
28         if len(person):
29             print "Dear %s/n"%person.name

30 def fly_all(flyables):
31     collection = flyables
32     for elem in collection:
33         elem.fly()

34 def fly_all_out(flyables):
35     for elem in flyables:
36         elem.fly()
37     print flyables

38 def talk_free(talkable, content):
39     talkable.talk(content)

40 if "__main__" ==__name__:
41     parrot = Parrot("Jack")
42     ostrich = Ostrich("Lily")
43     rline = create_line()
44     flight = Flight(1, rline)
45     action = raw_input("fly or talk:")

46     if action == "fly":
47         fly_all([parrot, flight])

48         # Type bug: ostrich.fly() is not allowed
49         fly_all([parrot, flight, ostrich])
50     elif action == "talk":
51         ostrich.__len__() = lambda: 1
52         talk_free(flight, ostrich)
```

图 6.1 存在类型缺陷的 Python 代码

6.2 问题分析

类型系统对于软件的正确构造有重要意义。然而，在 Python 中，动态类型使得类型系统只在运行时发挥作用，不利于检查类型缺陷，从而给 Python 程序的重构正确性评估带来了一定的困难。

第一，动态类型检查往往不能提供类型缺陷的快速反馈，从而延缓了缺陷发现的时机，导致开发成本提高。首先，对于诸如 Java 之类拥有即时编译技术的静态语言，集成开发环境（Integration Development Environment，IDE）能在第一时间发现类型缺陷，并进行报告。例如，若利用 Java 重新实现图 6.1，在开

发者输入行 49 时，Eclipse[112]将在该行代码下添加红色波浪线，以示警告。但是已有的 Python IDE 都难以实现该功能。其次，在函数被动态调用前，解释器不能对其定义进行类型检查。这导致解释器提供的错误信息往往与实际错误之间存在差距。仍以图 6.1 为例，将行 33 改为 *elem.fyl()*会引起一个拼写错误。当解释器执行到第 33 行时，它将抛出异常 *AttributeError* 来报告对象 *parrot* 没有属性 *fyl*，而非 *fyl* 拼写有误。

第二，动态类型检查只能覆盖有限的路径和上下文，有可能导致遗漏类型缺陷。在图 6.1 中，由于输入数据 *"talk"* 未覆盖行 49，因而它不能发现该行的类型缺陷。在实际开发中，动态类型检查主要通过单元测试进行。虽然高质量的单元测试可以发现大多数的类型缺陷，但是仍会存在遗漏。在软件发布后，遗留的缺陷可能导致系统在产品环境中崩溃。

除对检查类型缺陷造成负面影响外，动态类型还导致其他需要类型信息的软件开发相关工作难以进行，如增强 IDE、生成形式化文档等。这从很大程度上阻碍了软件开发效率的提高。

首先，动态类型难以为 IDE 的智能提示提供支持。IDE 是提高软件开发效率的重要工具，所以对 IDE 进行增强具有极大的价值[113]。但是，动态类型使得 IDE 难以在静态时利用类型信息提供智能提示。如在图 6.1 的行 33 处，由于缺乏参数对象 *flyables* 的类型信息，当输入 *elem.*时，IDE 不会给出任何关于对象 *elem* 所支持的操作的提示，但函数的开发者通常会假定参数具有某种类型或者支持一组特定的操作。

其次，动态类型给重构工具的开发带来困难。这是由于诸如重命名之类的许多重构方法都需要依赖类型信息来实现自动化。以图 6.1 为例，函数 *fly_all* 和 *fly_all_out* 的实参应支持一组相同的操作，即在函数的定义范围内具有结构一致性。因此，当开发者将行 33 处的 *fly* 改名为 *wing* 时，他期望行 36 也能自动更正。然而，IDE 从 Python 代码中难以获得这样的需求，从而无法实现自动替换。

最后，动态类型增加了开发者获得函数参数类型信息的成本。由于函数界面上没有类型信息，因而在不阅读函数定义或者函数相关文档的情况下，客户很难判断应传递哪种类型的实参，这从一定程度上降低了开发效率。如在图 6.1 中，若行 49 能显示函数 *fly_all* 对实参的类型约束，则能从一定能够程度上避免传递类型错误的实参。

Python 拥有结构类型系统。在进行类型检查时，Python 不关心对象的具体类型，而只关心对象所支持的操作。也就是说，类型检查主要考察对象是否符合一定的语法结构：若语法结构与对象的动态语义之间存在差距，则引起类型缺陷。例如，在图 6.1 中，函数 *fly_all* 不关心参数对象 *flyables* 的具体类型，而只要求它满足：(1) 支持操作 *flyables.__iter__()*，且该操作返回迭代子 *iterator*。(2) *iterator* 所指向的对象 *elem* 支持操作 *elem.fly()*。据此可以判断，行 49 存在类型缺陷。这是因为虽然对象 *ostrich* 属于 *Bird* 类型，但是它不支持操作 *ostrich.fly()*，所以它的动态语义与函数 *fly_all* 的静态语法结构不一致。

综上所述，我们需要一种类型约束系统来辅助开发者提高开发效率，降低维护成本。该约束系统应能提供一种简洁而灵活的方式来表达 Python 对象的类型约束。而且，这种表达方式应充分考虑 Python 类型系统的特征。在此基础上，我们可以利用该系统来检查类型缺陷，从而辅助开发者评估 Python 程序的重构正确性。另外，我们还能通过该系统还为诸如 IDE 增强这样需要类型信息的工作提供支持。

类似的，C++模板的类型系统也是以结构一致性为基础的。为表达这种基于结构一致性的类型约束，C++0x 提出了 concept 的概念，这对于我们的工作有较高的参考价值。因此，下一节将结合 C++0x 讨论利用 concept 表达结构类型系统的约束的合理性。

6.3 Concept 与结构类型系统

C++模板以结构一致性为基础对类型实参进行类型检查[114-115]。与 Python 类似，模板只关心类型实参的行为（所支持的操作），而不关心它的具体类型。图 6.2(a)给出了不受约束的模板函数 *min*。由函数 *min* 的定义可知，它的类型实参只需支持操作 *bool operator<(T, T)*即可。因此，行 8~10 是对函数 *min* 的合法调用，而行 12 将引发编译时错误。这是因为类型 *int* 和 *std::string* 均支持操作 *bool operator<(T, T>*，但类型 *v_int* 却不支持该操作。

C++98 只支持不受约束的模板。即是说，模板参数的类型约束隐式地散布于模板的定义中。如图 6.2(a)

中，虽然模板参数 *T* 需要支持操作符“<”，但是没有一种形式化的表达方式来定义这种类型约束，这给C++软件的开发带来了一系列问题[114]。为此，C++0x 提出了 concept 的概念，旨在通过描述模板参数所支持的操作来表达它的类型约束[114]。图 6.2(b)定义了名为 *LessThanComparable* 的 concept 和带约束的模板函数 *min*。其中，行 4 对参数 *T* 的类型约束进行了抽象，行 7 的 requires 子句表明函数 *min* 的模板参数 *T* 必须满足 *LessThanComparable*。

Concept 机制从诸多方面对 C++模板进行了增强。第一，提高了 C++类型系统的抽象层次，从而帮助编译器提供可读性更高的错误信息。例如，对于图 6.2(a)的行 12，编译器可以直接报告类型 *v_int* 不满足 *LessThanComparable*。第二，使得模板可以独立于客户进行类型检查。不受约束的模板只能在实例化时进行类型检查。即是说，类型检查依赖于客户。同样以图 6.2(a)为例，实例化前，编译器只对函数 *min* 进行语法检查。即使将行 5 改为 *a % b* 会导致一个类型缺陷，但编译器不会在行 8 实例化前发现该缺陷。而在图 6.2(b)中，由于 *LessThanComparable* 的存在，编译器能够直接指出这样的修改不符合该 concept 所规定的约束。第三，为模板提供与代码一致的文档。concept 可作为模板的形式化文档。更重要地，concept 属于代码的一部分，所以不会发生 concept 与代码不一致的情况，否则编译器会报告错误。例如，在图 6.2(b)中，*LessThanComparable* 是函数 *min* 的形式化文档，且 IDE 能同步两者的修改。第四，帮助 IDE 为模板编写提供智能提示。例如，在图 6.2(b)中的行 10，当输入 *x* 时，编译器会提示 *x* 支持操作符<，而这在图 6.2(a)中是难以实现的。第五，支持自动化重构工具的开发。如在图 6.2(b)中，若将行 4 改为 *bool isLess(T, T)*，则行 10 会自动替换为 *isLess(x, y) ? x : y*。

综上说述，我们的类型约束系统可以借鉴 C++0x concept。一方面，Python 和 C++模板的类型系统均为结构类型系统，而 concept 能有效表达这种类型系统的约束。例如，图 6.2(b)中的 *LessThanComparable* 明确描述了模板对参数 *T* 的类型约束，所以我们也可通过类似方式来描述图 6.1 的函数 *fly_all* 对参数 flyables 的类型约束。另一方面，concept 所解决的问题与 Python 程序开发中所面临的困难有较大的相似性。例如，与 Python 类似，编译器在实例化前不对模板进行类型检查，而 concept 能有效解决这一问题。

然而，Python 与 C++模板也存在不同之处。首先，Python 类型约束系统着重于对象，而 C++模板的 concept 关注的是类型。其次，前者是动态类型，而后者是静态类型，所以我们仅借鉴 concept 这种形式化表达方式。在此基础上，我们可以根据需求提供多种不同于 C++模板的类型检查策略，从而有效支持 Python 程序的重构正确性检查。下一节将详细讨论 Python 类型约束系统 PyConcept 的建立。

```
1  template<typename T>
2  const T& min(const T& a, const T& b)
3  {
4      // requires T supporting operator<
5      return a < b ? a : b;
6  }
7  typedef std::vector<int> v_int;
8  min(1, 2);
9  min(std::string("hello"),
10     std::string("hi"));
11 // Bug: the type v_int does not overload <
12 min(v_int(), v_int())
```

(a) 不受约束的模板函数 min

```
1  concept LessThanComparable<typename T>
2  {
3      // T should support the operation
4      bool operator<(T x, T y);
5  }
6  template<typename T>
7  requires LessThanComparable<T>
8  const T& min(const T& x, const T& y)
9  {
10     return x < y ? x : y;
11 }
```

(b) 受 LessThanComparable 约束的模板函数 min

图 6.2 模板函数 min

6.4 类型约束系统

为解决 Python 程序开发所面临的困难，我们提出了一种 Python 类型约束系统 PyConcept：首先讨论了 PyConcept 的设计目标，然后分析了 Python 的基本类型约束，并在此基础上利用 BNF 范式给出了 PyConcept 的形式化描述，最后结合 6.1 节的实例对 PyConcept 进行了分析。

6.4.1 设计目标

PyConcept 旨在通过表达 Python 对象的类型约束来支持需要类型信息辅助的软件工程活动，从而提高 Python 程序的开发效率。本章重点关注利用 PyConcept 来检查 Python 程序的类型缺陷，以改进现有的重构正确性评估方法。根据这一指导原则，我们提出了下列 PyConcept 的设计目标：

- 不进行语言扩展。在 Python 中，类型约束通过对象的使用方式来表达，隐式地分布于代码中，从而给 Python 程序的开发造成了困难。PyConcept 的主要目标是将这些约束以形式化的方式显式地表达出来，为重构正确性评估、IDE 增强等提供支持，从而提高 Python 程序的开发效率。这种形式化表达应独立于 Python 语言，且不需要对 Python 语言及其运行环境进行修改。更重要地，不扩展语言能帮助 PyConcept 更好地与现有 Python 开发环境和工具进行配合。
- 不修改 Python 类型系统。结构类型系统允许开发者从更高抽象层次上考虑算法，给 Python 带来了强大的表达能力和灵活性。例如，在实现快速排序算法时，只需考虑该算法的抽象流程，而不需关心排序对象的具体类型。所以，我们将关注于如何表达 Python 类型系统的约束，而不是修改现有的 Python 类型系统。
- 能够充分表达 Python 的类型约束。
- 主要表达语法结构。即是说，以保证语法结构的一致性为主，而将语义一致性的检查工作留给测试。实际上，语义检查属于本质复杂性[116]，在传统上主要依赖测试。我们的工作旨在从一定程度上降低偶然复杂性，使开发者能够专注于解决本质困难。
- 尽力使用 Python 机制来构建 PyConcept，以降低它实现的复杂性。例如，名字查找规则和作用域规则与 Python 一致。
- PyConcept 的语法应尽量保持简单，意即它的语法元素应尽量正交。
- PyConcept 将尽量借鉴 C++0x concept 的概念来降低学习难度，如关键字的选择。
- 采用 Python-like 的风格描述类型约束，这有利于 Python 开发者学习和接受。

6.4.2 基本类型约束

如何表达类型约束是 PyConcept 所需回答的基本问题之一。类型约束的核心是对象所支持的操作，所以下文将讨论操作对参数对象的基本类型约束。

第一种是类型，即要求对象具有特定类型。例如，内建函数 *range* 的所有实参必须为 *int* 类型。再如，控制流语句 *raise args* 要求参数 *args* 必须是类、实例或字符串。

第二种是有效属性，即要求对象拥有特定的属性。例如，操作 *getattr(object, "name")*要求 *object* 具有属性 *name*。再如，取负运算-*x* 需要 *x* 拥有函数属性 *__neg__*。在通常情况下，Python 会将对运算符和控制流语句的调用会转化为对对象特殊函数属性的调用[100, 111]。用户可通过定义这些特殊函数来修改运算符和控制流语句的语义。如在图 6.1 中，类 *Ostrich* 通过定义特殊函数 *__call__* 使其实例 *ostrich* 能支持函数调用运算符*()*。表 6.1 给出了常见运算符和控制流语句与特殊函数的对应关系。

表 6.1 常见运算符和控制流语句对应的特殊函数

操作	特殊函数	类别
x([args])	x.__call__([args])	运算符
str(x)	x.__str__()	
x[index]	x.__getitem__(index)	
x + y	x.__add__(y) 或 y.__radd__(x)	
for elem in x	x.__iter__()	控制流语句

第三种是相关对象。对象 *object* 的相关对象是指由 *object* 决定且其类型约束与 *object* 的类型约束有关联的对象。如在语句 *for elem in flyables* 中，对象 *elem* 即为对象 *flyables* 的相关对象。这是因为 *elem* 是 *flyables*

的“元素”，且 *flyables* 的类型约束包含 *elem* 的类型约束。事实上，容器元素通常是容器的相关对象。

复杂的类型约束可由上述基本类型约束组合而成。如图 6.1 中，函数 *fly_all* 的参数对象 *flyables* 的类型约束包括：(1) 相关对象 *elem* 和 *iter*；(2) 有效属性 *flyables.__iter__()*、*iter.__iter__()*、*iter.next()*以及 *elem.fly()*。下节将详细讨论类型约束的形式化描述。

6.4.3 系统定义

在 PyConcept 中，对象的类型约束是以 concept 为基础利用 requires 子句来表达的。一个 *concept* 是一组类型约束的形式化抽象，其定义包括 concept 名，对象参数列表和 concept 体。其中，参数列表不能为空。

本节首先通过一个简单的例子来概要地解释 PyConcept 的表达方式。图 6.3 定义了两个 concept: *Iterator* 和 *Iterable* 以及受约束函数 *print_all*。由于 PyConcept 不进行语言扩展，所以在 Python 代码中撰写 concept 和 requires 子句均是以 Python 注释符#加符号@开始的特殊注释。此外，为保证简单性，任意 concept 均具有模块作用域，意即 concept 不能定义于类或函数中。在图 6.3 中，*Iterator* 要求任何满足它的对象都应满足 concept 体规定的类型约束：(1) 拥有函数属性 *__iter__*，且该函数的返回对象同样满足 *Iterator*；(2) 具备函数属性 *next*；(3) 函数属性 *next* 返回相关对象 *item*。不难看出，*Iterator* 与 Python 的 iterator protocol 一致[111]。*Iterable* 规定符合它的对象应满足：(1) 具有函数属性 *__iter__*，且该属性的返回对象应满足 *Iterator*；(2) 其相关对象 *item* 是 *Iterator* 的相关对象 *item*。在 concept 体中，嵌套 requires 子句允许用户利用已有的 concept 构建新的 concept，从而提供了一种 concept 的重用机制。函数 *print_all* 的 requires 子句要求它的实参 *objects* 满足 *Iterable*。

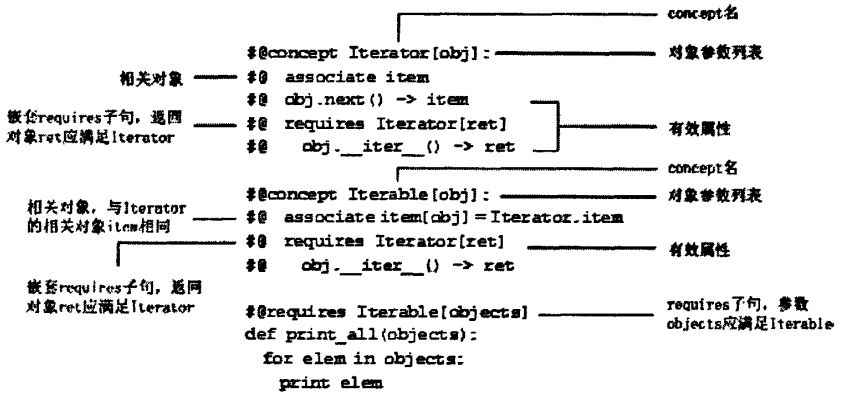


图 6.3 concept Iterator 和 Iterable 以及受约束函数 print_all

图 6.4 给出了 PyConcept 的 BNF 描述。其中，行 1~19 描述 concept 的语法，行 20 说明如何利用 requires 子句指定 Python 对象的类型约束。图 6.5 展示了如何利用 concept 指定图 6.1 中函数 *Flight.talk*，*fly_all* 和 *fly_all_out* 参数的类型约束。下面结合图 6.5 对 requires 子句和 concept 体进行详细讨论。

(1) requires 子句

requires 子句以关键字 requires 开头，用于说明对象满足已定义的 concept 或具有特定类型。如图 6.5 中，行 15 的 requires 子句表示 *Flight.talk* 的实参应满 *PersonTalk*，行 20 和行 25 说明函数 *fly_all* 和 *fly_all_out* 的实参需符合 *FlyableSeq*。

对于给定约束 req_1、…、req_n，若要求这些约束被对象同时满足，则可使用关键字 and 连接，表示为 *requires* req_1 *and* req_n；反之，若对象只需符合其中一个约束，则可使用关键字 or 连接，表示为 *requires* req_1 *or* req_n。这里，约束是指满足特定的 concept 或具有特定的类型。例如，*requires Sized[river] and Iterable[seq]* 表示对象 *river* 满足 *Sized*，同时对象 *seq* 满足 *Iterable*。而 *requires object: int or Sized[object]*则代表对象 *object* 是 *int* 类型或者满足 *Sized*。

```
1  conceptdef ::= "concept" name "[" obj_list "]" ":" NEWLINE INDENDT
2    concept_body DEDENT
3  obj_list ::= name { "," name }
4  concept_body ::= required_requirement_stmt { NEWLINE+ required_requirement_stmt }

5  required_requirement_stmt :: = [ requires_list NEWLINE INDENDT ] requirement_stmt | requires_list
6  requirement_stmt ::= message_stmt | associateobj_stmt

7  message_stmt ::= property_stmt | method_stmt
8  property_stmt ::= dotted_name "." name
9  method _stmt ::= [ dotted_name "." ] name "(" [ parameter_list ] ")" [ "->" returnobj ]
10 parameter_list ::= parameter { "," parameter } [ "*args" [ "," "**kw" ] ]
11 parameter ::= dotted_name [ "= __default__" ]
12 returnobj ::= identifier [ ":" name ]
13 associateobj_stmt ::= "associate" name [ "=" dotted_name [ "[" dotted_name_list "]" "." name ] ]

14 requires_list ::= "requires" requires_stmt { conj [NEWLINE] requires_stmt }
15 requires_stmt ::= dotted_name "[" dotted_name_list "]" | dotted_name [ ":" name ]
16 conj = "and" | "or"

17 dotted_name_list ::= dotted_name {"," dotted_name }
18 dotted_name ::= name { "." name }
19 name ::= identifier

20 constrain_stmt = requires_list NEWLINE py_stmt
```

图 6.4 concept 的 BNF 描述

返回对象是函数的输出参数。在 concept 中，有效函数属性通过符号 "->" 来指定返回对象。然而，Python 3.0 以前的 Python 版本不支持显式地表达返回对象[111]。这使得我们难以对 Python 函数的返回对象应用类型约束。为此，PyConcept 利用函数名来表示返回对象。例如，

```
#@requires foo: int
def foo(): return 1
```

这里，函数 *foo* 的 requires 子句规定 *foo* 的返回对象应为 *int* 类型。

(2) concept 体

concept 的核心是 concept 体。concept 体用于描述 concept 的参数应满足的一组类型约束，由有效属性、相关对象和嵌套 requires 子句组成。表 6.2 给出了类型约束的形式化表达方式以及对应的解释。

表 6.2 类型约束的形式化表达

类型约束	形式化表达	解释
类型	requires object : T	对象 object 的类型是 T。
concept	requires C[object]	对象 object 满足 concept C。
有效属性	requires C[attribute] or attribute : T object.attribute	对象 object 具有属性 attribute，且 attribute 满足 concept C 或具备类型 T。
	requires C_1[para] and C_2[ret] object.method(para [= __default__]) -> ret	对象 object 具有函数属性 method，且 method 的参数 para 和返回值 ret 分别满足 concept C_1 和 C_2。此外，__default__说明 para 有默认值，即 object.method()是合法调用。特别地，我们使用 *arg 和**kw 表示可变参数。
相关对象	requires C_1[ass_obj] and C_2[ass_obj] associate ass_obj [= name]	定义相关对象 ass_obj。其中，可选初始化代表 ass_obj 与 name 指向相同对象。requires 子句说明 ass_obj 应同时满足 concept C_1 和 C_2。

以图 6.5 为例，行 2~3 定义了有效函数属性 *__len__*，且 *__len__* 的返回对象为 *int* 类型。行 5 的 requires 子句表示 *FlyableSeq* 的参数 *obj* 应满足 *Iteratable*。由此可知，*FlyableSeq* 比 *Iterable* 更为严格。即是说，满

足 *FlyableSeq* 的对象必须满足 *Iterable*；反之不成立。行 6 定义了 *FlyableSeq* 的相关对象 *item*，且 *item* 以对象 *Iteratable.item* 进行初始化。行 13~14 表示对象有数据属性 *name*，且 *name* 满足 *String*。

```
1  #@concept Sized[obj]:
2  #@    requires length : int
3  #@        obj.__len__() -> length

4  #@concept FlyableSeq[obj]:
5  #@    requires Iterable[obj]
6  #@    associate item = Iterable.item
7  #@    item.fly()

8  #@concept String[obj]:
9  #@    requires string : str
10 #@        obj.__str__() -> string

11 #@concept PersonTalk[obj]:
12 #@    requires Sized[obj]
13 #@    requires String[name]
14 #@        obj.name

15 #@requires PersonTalk[person]:
16 def talk(self, person):
17     if len(person):
18         print 'Dear %s/n'
19               %person.name

20 #@requires FlyableSeq[flyables]:
21 def fly_all(flyables)
22     collection = flyables
23     for elem in collection:
24         elem.fly()

25 #@requires FlyableSeq[flyables]:
26 def fly_all_out(flyables)
27     for elem in flyables:
28         elem.fly()
29     print flyables
```

图 6.5 用 concept 约束图 6.1 的函数 Flight.talk(self, person)，fly_all(flyables)和 fly_all_out(flyables)

在 concept 体中，concept 的对象参数和相关对象具有 concept 作用域，即任何位置都能引用。而对于有效函数属性的参数，若它不是相关对象，则它只有局部作用域。换句话说，它只对有效函数属性的定义起作用。这是因为 PyConcept 允许多个有效函数属性拥有同名参数，而互不影响。例如，

```
concept C[obj]:
   requires x : str
      obj.foo(x)
   obj.goo(x)
```

这里，有效函数属性 *foo* 和 *goo* 的参数都只有局部作用域，所以两处的 *x* 不同。因此，语句 *obj.goo(x)*中的 x 不受语句 *requires x : str* 的约束。而 concept 参数 *obj* 有 concept 作用域，则 concept 体中任意位置的 *obj* 均具有相同含义。

此外，requires 子句（包括嵌套 requires 子句）的名字查找遵守 Python 的名字查找规则与作用域规则。例如，*requires SC.C[obj]*代表对象 *obj* 满足在模块 *SC* 中定义的 concept *C*。

6.4.4 系统特性

concept 通过 requires 子句相互联系，从而形成了一个类似类体系结构的 concept 体系结构。concept 之间存在两种关系：使用（Usage）和精化（Refinement）。给定 concept C_1 和 C_2，若 C_1 在有效属性或相关对象的嵌套 requires 子句中引用了 C_2，则称 C_1 使用了 C_2；若 C_1 的对象参数满足 C_2，则称 C_1 精化了 C_2。即是说，C_1 表达了比 C_2 更为严格的类型约束。图 6.6 给出了图 6.3 和图 6.5 中 concept 的体系结构。在图 6.6 中，*Iterable* 和 *PersonTalk* 分别使用了 *Iterator* 和 *String*，*FlyableSeq* 精化了 *Iterable*，而 *String* 和 *PersonTalk* 同时精化了 *Sized*。

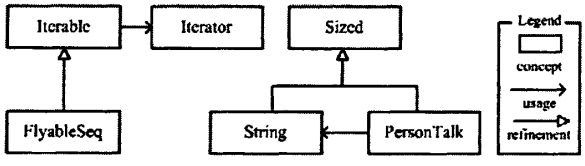


图 6.6 图 6.3 和图 6.5 中的 concept 体系结构

PyConcept 有效地表达了对象所支持的操作，是对结构类型系统约束的良好抽象。因而，它能较好地解决 6.2 节所列举的困难：

首先，PyConcept 能帮助开发者及时定位类型缺陷，从而有助于检查 Python 程序的重构正确性。如图 6.1 的行 49，对象 *ostrich* 不支持操作 *ostrich.fly()*，从而导致实参*[parrot, flight, ostrich]*不满足 *FlyableSeq*。在 concept 帮助下，IDE 有可能在开发者输入行 49 时立即报告类型缺陷。这样的检查不依赖于程序执行的路径和上下文，所以避免了遗漏类型缺陷。

其次，PyConcept 为 IDE 的智能提示提供支持。以图 6.5 的函数 *fly_all_out* 为例，当开发者输入 *elem.* 时，IDE 可根据参数 *flyables* 满足 *FlyableSeq* 的信息“智能”地给出 *elem* 支持函数 *fly* 的提示。

再次，PyConcept 可以辅助实现自动化重构。考虑图 6.5，函数 *fly_all* 和 *fly_all_out* 的实参均满足 *FlyableSeq*。因此，若将 *FlyableSeq* 中的 *item.fly()*改为 *item.flying()*，则 IDE 能够根据 *FlyableSeq* 自动地将这个重命名应用到这两个函数上。

最后，PyConcept 为函数提供了形式化的文档。如图 6.5 中，*FlyableSeq* 能清晰地表达函数 *fly_out* 的参数应具备的“特征”。

PyConcept 可由开发者手工编写，亦可由工具自动生成。为提高编写和生成的效率，PyConcept 预定义了一批基本的 concept，主要涉及容器和迭代子，包括 *Sequence*、*Map*、*Iterable* 以及 *Sized* 等。在此基础上，建立了内建操作的参数的类型约束。这里，内建操作包括内建函数、运算符和控制流语句。例如，内建函数 *len* 的参数应满足 *Sized*，for 语句的参数应满足 *Iterable*。附录 2 是 PyConcept 预定义库的核心内容。接下来的章节将依次讨论 PyConcept 的构建算法和检查算法。

6.5 函数参数 Concept 的生成

PyConcept 可以通过多种方法生成：开发者手工编写、收集历史执行信息来推测以及从抽象基类[55]转化等。本节提出了一种高效的基于源代码分析的 PyConcept 生成算法 *generate_concept*，能自动生成函数参数的 concept，从而降低使用 PyConcept 的成本。由于函数是 Python 表达计算逻辑的基本单位，所以该算法可以从一定程度上提高开发效率。另外，该算法还能为其他基于 PyConcept 的算法提供自动生成 concept 的服务，如检查类型缺陷的算法。

对于给定模块 *m*，算法 *generate_concept* 扫描 *m* 的定义体，依次生成 *m* 以及 *m* 导入模块中各函数参数的 concept。如图 6.7 所示，该算法可分为三步：计算函数参数的过程内别名、构建函数调用图以及生成函数参数的 concept。

```
1  def generate_concept(module_def):
2      '''module_def: the definition of the module requiring analyzed'''
3      # Step 1: compute the intra-procedural alias of the arguments
4      alias = compute_func_args_alias(callgraph)

5      # Step 2: build the call graph
6      call_graph = construct_callgraph(module_def)

7      # Step 3: extract the type constraints
8      module_concepts = {} # map from function to type constraints
9      # scan in the order topology on call edge
10     for func in call_graph:
11         # func will be analyzed only if it has no concept.
12         if not module_concepts.has_key(func):
13             module_concepts[func] = construct_func_concept(func, alias, call_graph)

14 return module_concepts
```

图 6.7 算法 generate_concept

在第一步中，我们采用 Steensgaard 方法[117]获取参数的过程内别名。这是因为该方法的复杂性较低，有助于降低 *generate_concept* 的时间复杂度。鉴于 Steensgaard 是成熟算法，我们不进行详细讨论。

在第二步中，为构建函数调用图，我们提出了改进的变量类型分析（Refined Variable Type Analysis，RVTA）。类型变量分析（Variable Type Analysis，VTA）通过赋值链和对象所在的类体系结构来确定对象类

型[118]，是一种高效的类型推导算法。公式 6.1 给出 VTA 的形式化描述。

$$\text{type}(obj) = \text{reaching\_types}(\text{representative}(obj)) \cap \text{hierarchy\_types}(C) \tag{6.1}$$

其中，*obj* 是对象；type(*obj*)代表 *obj* 的类型；representative(*obj*)表示 *obj* 对应的类型传播图的节点 *node*；reaching_types(*node*)是 *node* 关联的类型集，即通过赋值链可以到达 *obj* 的类型；*C* 为 *obj* 的声明类型；hierarchy_types(*C*)是以 *C* 为根的继承体系结构，包括 *C* 及其子类。

例如，若存在赋值序列 $x_1 = new\ A(); x_2 = x_1; \cdots; x_n = x_{n-1}; obj = x_n$，则 VTA 认为类型 *A* 可达对象 *obj*。

然而，Python 是结构类型系统，因此，对象可支持的操作与对象所在的类体系结构无关，只与对象的语法结构有关。以图 6.1 行 33 为例，语句 *elem.fly()*是合法的，当且仅当对象 *elem* 具有函数属性 *fly*。所以，在计算变量类型时，RVTA 不考虑类继承体系结构。公式 6.2 展示了 RVTA 的计算方法：

$$\text{type}(obj) = \text{reaching\_types}(\text{representative}(obj)) \tag{6.2}$$

其中，各符号的含义与公式 6.1 相同。

基于 RVTA，若在函数调用图中存在边$(f_1, T.f_2)$，当且仅当 f_1 和 $T.f_2$ 满足如下条件：(1) f_1 中存在调用语句 $obj.f_2$；(2) $T \in \text{type}(obj)$。

特别地，由于 Python 程序常将*__main__*部分看作模块入口，因而我们把该部分当成主函数，在调用图上生成与它对应的特殊节点。如图 6.1 中，行 41~52 是该模块的主函数*__main__*。

在第三步中，我们调用算法 *construct_func_concept* 来生成函数参数的 concept。这里，行 12 的 if 语句保证每个函数仅被分析一次。如图 6.8 所示，该算法的基本思路是分析函数参数在函数中的使用方式：扫描函数 *func* 的所有语句，若语句 *stmt* 引用了参数 *arg*，则调用算法 *extract_constraints* 生成 *arg* 的类型约束。

算法 *construct_func_concept* 将语句对参数的引用归纳为三种类型。第一种是直接引用。如图 6.1 中，行 35 直接引用了参数 *flyables*。第二种是别名引用。例如，图 6.1 的行 32 引用了参数 *flyables* 的别名 *collection*。第三种是相关对象引用。在图 6.1 中，行 36 通过相关对象 *elem* 引用了参数 *flyables*。对于第一种情况，我们直接抽取 *arg* 的类型约束。例如，图 6.1 的行 35 可以直接利用算法 *extract_constraints* 生成类型约束 *requires Iterable[flyables]*。而对于后两种情况，我们根据规则 6.1~6.2 生成相应的类型约束。

规则 6.1 (别名规则) 对于变量 *x*，若 *y* 是 *x* 的别名，且 *y* 满足类型约束 *r*，则为 *x* 生成类型约束 *r*。

以图 6.1 函数的 *fly_all* 为例，*collection* 是 *flyables* 的别名，且 *collection* 满足 *Iterable*，则为 *flyables* 生成类型约束 *requires Iterable[flyables]*。

特别地，若函数 *func* 是类 *A* 的成员函数，且 *y* 具有形式 *self.z*，则需要分析 *A* 的所有成员函数对 *y* 的使用方式。如图 6.1 中，变量 *self.name* 是 *Parrot.__init__*的参数 *name* 的别名，而行 7~8 要求 *self.name* 满足 *String*，所以应为 *name* 生成类型约束 *requires String[name]*。然而，*x* 与 *y* 互为别名，对 *y* 使用方式的分析会涉及 *x*。为消除这种循环依赖，在分析 *y* 时，我们不考虑别名引用，只分析直接引用和相关对象引用。这样做虽然会遗漏类型约束，但我们的目标是高效地生成 concept，以提供类型缺陷的快速反馈。

规则 6.2（相关对象规则）对于变量 *x*，若 *y* 是 *x* 的相关对象，且 *y* 满足类型约束 *r*，则将 *r* 加入 *x* 的 concept。特别地，若 *x* 满足 concept *C*，且 *y* 等于 *C* 的相关对象 *z*，则为 *x* 生成类型约束 *associate y = C.z*。

以图 6.1 中的函数 *fly_all_print* 为例，对象 *elem* 是参数 *flyables* 的相关对象，且 *elem* 满足有效属性 *elem.fly()*，因而将 *elem.fly()*添加到 *flyables* 对应的 concept 中。进一步地，*flyables* 满足 *Iterable*，且 *elem* 等于 *Iterable* 的相关对象 *item*，则生成类型约束 *associate elem = Iterable.item*。

算法 *extract_constraints* 根据语句 *stmt* 的类型为参数 *arg* 生成相应的类型约束：若 *stmt* 访问或调用了 *arg* 的属性，则根据属性规则（规则 6.3）生成有效属性；若 *stmt* 将 *arg* 作为实参传递给操作 *operation*，则根据调用规则（规则 6.4）产生类型约束。图 6.9 描述了该算法的详细过程。由图 6.9 不难看出，该算法会反过来调用算法 *construct_func_concept*，所以若在函数调用图中，存在调用序列$\{f_1,...,f_m,f_1\}$，意即被分析模块中存在递归调用，则算法 *construct_func_concept* 会陷入死循环。由于$\{f_1,...,f_n\}$与$\{f_1,...,f_m,f_1\}$所包含的语句相同，因而递归调用不会产生新的类型约束。基于此，为处理递归调用，我们可在函数调用图中直接删除边(f_m,f_1)。

```
1  def construct_func_concept(func_def, alias, call_graph):
2      '''func_def: the definition of the function requiring analyzed'''
3      '''alias: the intra-procedural alias information for the arguments'''
4      '''call_graph: call graph'''''
5      func_concepts = initialize()  # func_concepts is the map from argument to type constraints.
6      for stmt in func_def:
7          args = stmt.get_args()  # args is the arguments referenced by stmt.
8          for arg in args:
9              # direct reference
10             if stmt.direct_ref(arg):
11                 func_concepts[arg].union(extract_constraints(arg, stmt, call_graph))
12             # Rule 5.1: alias reference
13             elif alias.is_alias(arg):
14                 # Special situation for rule 5.1: arg's alias arg_alias is the member
15                 # of the class func_class, that is, it has the form like self.attribute.
16                 # In the case, all the methods of func_class should be analyzed.
17                 if arg_alias.is_member(func_class) and func_def.is_method(func_class):
18                     for cs in func_class:
19                         # The statement referenced directly or
20                         # by associate object should be analyzed.
21                         if stmt.direct_ref(arg_alias) or stmt.associate_ref(arg_alias):
22                             func_concepts[arg].union(
23                                 extract_constraints(arg_alias, stmt, call_graph))
24                 else:
25                     func_concepts[arg].union(extract_constraints(arg_alias, stmt, call_graph))
26             # Rule 5.2: associate object reference
27             elif stmt.associate_ref(arg):
28                 func_concepts[arg].union(extract_constraints
29                     (arg_ass, stmt, call_graph))  # arg_ass is arg's associate object
.
30                 # Special situation for rule 5.2: if arg satisfies concept arg_concept, and
31                 # arg_ass equals to arg_concept's associate object arg_concept_ass
32                 # then type constraints are generated using function type_constraints.
33                 if is_satisfied(arg, arg_concept) and is_equal(arg, arg_concept_ass):
34                     func_concepts[arg].union(
35                         type_constraints("associate arg_ass = arg_concept.arg_concept_ass"))

36     return func_concepts
```

图 6.8 算法 construct_func_concept

规则 6.3 (属性规则) 对于变量 *x*，若 *stmt* 具有形式 *x.attribute* 或 *x.method([real_args])*，则为 *x* 生成有效属性 *x.attribute* 或 *x.method([formal_args])*。其中，*formal_args* 是 *real_args* 的占位符。特别地，若 *stmt* 形如 *y = x.method([real_args])*，则为 *x* 生成如下两条类型约束：

associate y
x.method([formal_args]) -> y

例如，根据 *x.foo(1, m)*可生成 $x.foo(arg_1, arg_2)$。其中，arg_1,和 arg_2 分别对应于 *1* 和 *m*。而 *y = x.foo(1, m)* 对应于约束 *associate y* 和 $x.foo(arg_1, arg_2) \text{ -> } y$。

规则 6.4 (调用规则) 对于变量 *x*，若 *stmt* 是调用操作的语句 *call(..., x,...)*，且 *fx* 满足类型约束 *r*，则为 *x* 生成类型约束 *r*。其中，*fx* 是 *x* 对应的形参。

例如，

```
def foo(fx): fx.id
def goo(x): foo(x)
```

这里，函数 *foo* 的参数 *fx* 满足类型约束 *fx.id*，则可以为 *goo* 的参数 *x* 生成约束规则 *x.id*。在很多情况下，形参和实参是一种特殊的别名关系。

然而，Python 的动态类型使得静态确定 *stmt* 调用的操作变得十分困难。对于函数调用 *object.method()*，如果 RVTA 不能确定 *object* 的具体类型，则第二步只能获得一组可能调用的函数。下面讨论如何在这种情

况下生成类型约束。

```
1  def extract_constraints(obj, stmt, call_graph):
2      '''obj: the object needing generating type contraints'''
3      '''stmt: the statement requiring analyzed '''
4      '''call_graph: call graph'''
5      # Rule 5.3: access the obj's data attribute
6      if stmt.ref_data_attribute():
7          return type_constraint("obj.attribute")
8      # Rule 5.3: call the function attribute method([args])
9      elif stmt.ref_function_attribute():
10         return type_constraint("obj.method([args])")
11     # Special situation for rule 5.3: the return object is assigned to the object x.
12     elif stmt.assigned_ref_method_attribute():
13         ass_obj = type_constraint("associate x")
14         valid_attr = type_constraint("obj.method([args])->x")
15         return (ass_obj, valid_attr)
16     # Rule 5.4: the call operation call(…,obj,…)
17     elif stmt.is_called_operation():
18         concept = None  # concept is the concept of the formal argument corresponding to obj.
19         for callee in stmt.call_set(call_graph): # call_set(call_graph) is stmt's may call set.
20             # if the callee has no concept,
21             # then the function construct_func_concept is called.
22             if not callee.has_concept():
23                 concepts = construct_func_concept(callee)

24             # If the callee has concept,
25             # then function get_concepts is called to return the callee's concept.
26             # Here, the existed concepts can be generated by manual
27             # or by the function construct_func_concept or manual.
28             else:
29                 concepts = callee.get_concepts()

30                 # the MaxConSet of MinConSet is generated according to the requirements.
31                 join_concept(concept, concepts[formal_obj]) # formal_obj is formal arg for obj

32         return concept
```

图 6.9 算法 extract_constraints

定义 6.1 给定调用点 p: $call(x_1..., x_i, ..., x_n)$，可能调用集 CallSet($p$) = {$callee(fx_1..., fx_i, ..., fx_n) \mid p$ 可能调用 *callee*，且 fx_i 为 x_i 对应的形参}。对于 $f \in$ CallSet(p)，f 对于 x_i 的约束集 ConSet(x_i, f) = {$r \mid fx_i$ 满足 r，且 fx_i 为 x_i 对应的形参}。

定义 6.2 对于调用点 p: $call(x_1..., x_i, ..., x_n)$，有

$$MinConSet(x_i, p) = \bigcap_{f \in CallSet(p)} ConSet(x_i, f)$$
$$MaxConSet(x_i, p) = \bigcup_{f \in CallSet(p)} ConSet(x_i, f)$$

由定义 6.2 可知，MinConSet 是 p 对于 x_i 的最小约束集，违反它将导致运行时异常 *AttributeError* 或者 *TypeError*。而 MaxConSet 是 p 对于 x_i 的最大约束集，若 x 满足该约束集，则可认为不存在类型缺陷。特别地，若|CallSet(p)| = 1，即可以静态确定 p 调用的函数，则 MinConSet 等于 MaxConSet。以图 6.1 的行 39 为例，由于不能静态确定对象 *talkable* 的类型，则

CallSet(38) = {*Parrot.talk*, *Flight.talk*}
ConSet(*content*, *Parrot.talk*, 37) = {*requires Sized[content]*}
ConSet (*content*, *Flight.talk*, 37)
= {*requires Sized[content]*, *content.name*, *String[content.name]*}
MinConSet (*content*, *talkable.talk*) = {*requires Sized[content]*}
MaxConSet (*content*, *talkable.talk*)
= {*requires Sized[content]*, *content.name*, *String[content.name]*}

从上面的例子可以看出，参数 *content* 必须满足 MinConSet，否则将导致类型缺陷。若它不仅满足 MinConSet，还满足 MaxConSet，则无论 *talkable* 是哪种类型，都不存在类型缺陷。

另外，定义 6.2 表明计算 MinConSet 和 MaxConSet 需要 f 的类型约束。由图 6.9 可知，若 f 的类型约束未知，则算法 *extract_constraints* 会调用算法 *construct_func_concept* 来为 f 生成类型约束（行 22~23）。反之，若 f 的类型约束已知，则直接通过函数 *get_concepts* 获取（行 29）。在这种情况下，已存在的类型约束既可以是通过算法 *construct_func_concept* 生成的，也可以由手工编写的。

下面，我们以图 6.1 中的函数 *fly_all* 为例，展示算法 *construct_func_concept* 的计算过程：

第一步 行 32 引用了参数 *flyables* 的别名 *collection*，根据规则 5.1 应为 *flyables* 生成一条约束 *r*。其中，*r* 为 *collection* 满足的类型约束。

第二步 调用算法 *extract_constraints* 抽取 *r*。行 32 执行 for 语句，且 for 语句要求被遍历对象满足 *Iterable*，根据规则 5.4，为 *collection* 生成类型约束 *requires Iterable[collection]*。

第三步 行 33 引用了相关对象 *elem*，根据规则 5.2 调用算法 *extract_constraints* 获取 *elem* 的类型约束 *elem.fly()*。此外，由于 *elem* 等于 *Iterable.item*，生成类型约束 *associate elem = Iterable.item*。

最终的计算结果如下：

```
concept ConceptFlyables[obj]:
    associate elem = Iterable.item
    Iterable[obj]
    elem.fly()
```

不难看出，该结果与图 6.5 中手工编写的 *FlyableSeq* 一致，这表明算法 *construct_func_concept* 能有效生成函数参数的 concept。

鉴于算法 *generate_concept* 分为三步，它的时间复杂度为各步骤复杂度之和。第一步采用 Steensgaard 方法，其时间复杂度近似与被分析模块的规模 N 成线性关系[117]。其中，被分析模块包括作为算法实参的模块 *m* 和被 *m* 导入的模块。第二步以 RVTA 为基础生成函数调用图。由于 VTA 的时间复杂度近似与 N 成线性关系，且其主要时间消耗为计算 reaching_types(representative(*obj*))[114]，根据公式 6.1~6.2，RVTA 的时间复杂度也近似与 N 成线性关系。因此，第二步的时间复杂度也近似与 N 成线性关系。第三步的时间消耗主要包括遍历语句、创建 concept 以及合并 concept。遍历语句和创建 concept 的时间复杂度与 N 成正比。而合并 concept 等价于集合的 find/union 操作，其时间复杂度为 $O(N\alpha(N, N))$。因此，第三步的时间复杂度为 $O(N\alpha(N, N))$。这里，$\alpha(N, N)$为 Ackermann 的逆函数。综上所述，算法 *generate_concept* 的时间复杂度近似与 N 成线性关系。

6.6 类型缺陷的检查

Python 主要依赖于测试评估重构的正确性。然而，根据 5.2 节的讨论，Python 的动态类型不利于在测试过程中定位类型缺陷。鉴于在对象上执行对象不支持的操作（传递类型错误的实参）是一种主要的类型缺陷，本节提出了基于 PyConcept 的类型缺陷检查算法 *check_func_call*。该算法可以检查检查操作调用的合法性，从而辅助开发者定位类型缺陷。

6.6.1 检查算法

算法 *check_func_call* 在 PyConcept 的基础上检查操作调用点的合法性：若存在实参 *arg* 不满足调用点对应的 concept，则该调用点存在类型缺陷。这里的 concept 既可通过算法 *generate_concept* 自动生成，也可通过手工编写。图 6.10 给出了该算法的详细描述。

如图 6.10 所示，算法 *check_func_call* 的计算分为三步。第一步进行过程内类型推导以及构建函数调用图。第二步计算函数参数的过程内别名。这两步采用与算法 *generate_concept* 相同的策略。第三步通过算法 *check_argument* 来检查调用点的合法性。图 6.11 展示了算法 *check_argument* 的计算过程：首先调用算法 *construct_func_concept* 获得调用语句（调用点）的 concept，然后依次检查实参的合法性。

算法 *check_func_call* 的时间复杂度近似与被分析模块的规模 N 成线性关系。与算法 *generate_concept* 类似，前两步的时间复杂度近似与 N 成线性关系。第三步循环的时间消耗主要包括遍历语句、创建 concept、合并 concept 以及检查 concept。遍历语句和创建 concept 的时间复杂度与 N 成正比。而合并和检查 concept 等价于集合的 find/union 操作，其时间复杂度为 $O(N\alpha(N, N))$。。

```
1  def check_func_call(module_def):
2      '''module_def: the definition of the module requiring analyzed'''
3      # Step 1: build call graph and infer object type
4      type_info = infer_type(module_def)
5      call_graph = construct_callgraph(module_def)

6      # Step 2: compute the intra-procedural alias of the arguments
7      alias = compute_func_args_alias(call_graph)

8      # Step 3: check whether the callsite is valid or not
9      invalids = {}  # invalids is the map from invalid callsits to type bug.
10     for callsite in module_def:
11        invalids[callsite] = check_argument(callsite,
12                                            alias, call_graph, type_info)

13     return invalids
```

图 6.10 算法 check_func_call

```
1  def check_argument(callsite, alias, call_graph, type_info):
2      '''callsite: callsite'''
3      '''alias: the intra-procedural alias information for the arguments'''
4      '''call_graph: call graph'''
5      '''type_info: map from object to type information'''
6      # the callsite is a call statement, and thus,
7      # the concept can be generated in terms of rule 5.4.
8      concepts = construct_func_concept(callsite, alias, call_graph)

9      invalid2con = {}  # invalid2con is the map from invalid argument to the concept.
10     for rarg in callsite:  # rarg is the real argument of the callsite.
11        farg = rarg.get_formal_arg()  # farg is arg's corresponding formal argument.

12        # if rarg does not satisfy the corresponding concept
13        # then the call is a type bug.
14        if not tye_info[rarg].is_satisfy(concepts[farg]):
15           invalid2con[rarg] = concepts[farg]
16
17     return invalid2concept
```

图 6.11 算法 check_argument

6.6.2 实例分析

本节中以图 6.1 为例讨论算法 *check_func_call* 如何利用 PyConcept 来检查操作调用的合法性，从而定位类型缺陷。

第一步计算函数参数的过程内别名，其结果如表 6.3 所示。其中，前 3 项均具有形式 *self.attr*。因而，获取它们的类型约束需要分析对应类的所有成员函数。以第 3 项为例，虽然成员函数 *Flight.__init__* 对对象 *self.line* 没有约束，但成员函数 *Flight.fly* 要求 *self.line* 满足 Line。其中，concept *Line* 如下所示：

```
concept Line[obj]:
    obj.start
    obj.end
```

第二步进行过程内类型推导和构建函数调用图。表 6.4 给出了类型推导所得的对象类型信息。由表 6.4 可知，过程内类型推导只能获取部分对象的信息。例如，对象 *rline* 的类型由函数 *create_line* 的返回值决定，

所以不能通过过程内分析获得，这导致对象 *flight* 的类型难以完全确定。此外，在有些情况下，对于给定对象 *obj*，RVTA 只能获得 *obj* 的一组可能类型。例如，对象 *ostrich* 的可能类型包括行 42 的类型和行 51 的类型。图 6.12 是图 6.1 的函数调用图。受篇幅限制，我们只画出了与类型缺陷检查相关的部分。在图 6.12 中，静态分析难以确定函数 *talk_free* 调用的是 *Parrot.talk* 还是 *Flight.talk*。为了不遗漏错误，我们采用“保守”的分析方法，即在调用图上为两个函数都都生成调用边。

表 6.3 函数参数过程内别名

No.	参数	别名	函数
1	name	self.name	Bird.__init__(self, name)
2	no	self.no	Flight.__init__(self, no, line)
3	line	self.line	
4	flyables	collection	fly_all(flyables)

表 6.4 对象类型信息

对象	类型	支持操作	行号
parrot	Parrot	parrot.__init__(name) parrot.fly() parrot.talk(content) parrot.name	38
parrot.name	str	字符串支持的一切操作	38
ostrich	Ostrich	ostrich.__init__(name) ostrich.run() ostrich.__call__() ostrich.name	39
	匿名	ostrich.__init__(name) ostrich.run() ostrich.__call__() ostrich.__len__() ostrich.name	
ostrich.name	str	字符串支持的一切操作	39
flight	Flight	flight.__init__(name) flight.fly() flight.talk(person) flight.no flight.line	43
flight.no	int	整型支持的一切操作	43
[parrot, flight]	list	列表支持的一切操作	46
[parrot, flight, ostrich]	list	列表支持的一切操作	48

第三步调用算法 *check_argument* 依次检查 *__main__* 中各个调用点的合法性：

行 41—调用点 *Parrot("Jack")* CallSet(41) = {*Parrot.__init__*}，则 *concepts* = {*String[name]*}。由于常量*"Jack"*的类型是 *str*，因此它满足 concepts，是合法实参。

行 42—调用点 *Ostrich("Lily")* CallSet(42) = {*Ostrich.__init__*}，则 *concepts* = {}，即无类型约束。因此，该调用合法。

行 44—调用点 *Flight(1, rline)* CallSet(44) = {*Flight.__init__*}，则 *concepts* = {*Line[line]*}。然而，*rline* 的类型未知，无法判断 *rline* 是否满足 *Line*，进而难以确定该调用是否合法。

行 47—调用点 *fly_all([parrot, flight])* CallSet(46) = {*fly_all*}，则 *concepts* = {*ConceptFlyables[flyables]*}。实参 *[parrot, flight]*的类型为 *list*，且其元素均支持操作 *self.fly()*，因此它满足 *ConceptFlyables*，该调用合法。

行 49—调用点 *fly_all([parrot, flight, ostrich])* 因为实参*[parrot, flight, ostrich]*的元素 *ostrich* 不支持 *self.fly()*，所以该实参不满足 *ConceptFlyables*。不难推断行 49 存在类型缺陷。由此可知，算法 *check_func_call* 能够发现代码中存在的类型缺陷，从而辅助开发者评估重构正确性。

行 52—调用点 *talk_free(flight, ostrich)* CallSet(52) = {*talk_free*}，而调用语句 *talkable.talk(content)*的可能调用集为{*Parrot.talk, Flight.talk*}。在这种情况下，我们需要根据情况组合两个函数对应的 *concepts*：若采用最小约束集，则 *concepts* = {*Talkable[talkable], Sized[content]*}；若使用最大约束集，则 *concepts* = {*Talkable[talkable], PersonTalk[content]*}。其中，concept *Talkable* 如下所示：

```
concept Talkable[obj]:
    obj.talk(content)
```

由于对象 *flight* 的类型为 *Flight*，满足 *Talkable*，所以是合法实参。此外，最小约束集中的 *Sized* 要求对象 *ostrich* 支持内建函数 *len*。然而根据表 6.3，算法 *check_func_call* 无法判断 *ostrich* 是否支持 *len*，所以难以确定该调用是否合法。

上述实例分析表明算法 *check_func_call* 能有效定位类型缺陷，从而改进现有的 Python 程序重构正确性评估方法。可以看到，该算法准确地指出了行 49 存在类型缺陷。然而，过程内的 RVTA 难以获取所有对象的类型信息，因此可能存在无法判定的调用点。例如在行 43 处，由于无法确定对象 *rline* 的类型，所以难以判断调用点 *Flight(1, rline)*的合法性。在未来工作中，我们将进一步研究类型推导算法的精度对对类型缺陷检查的影响。

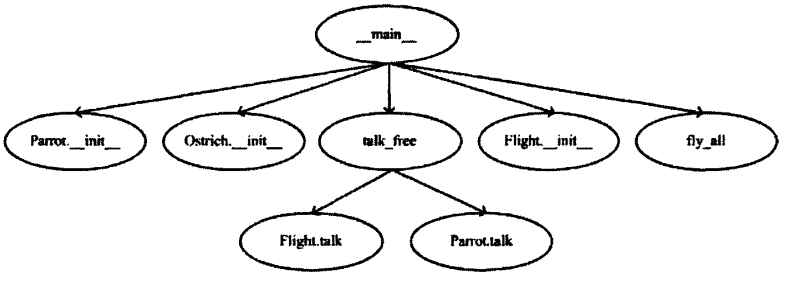


图 6.12 函数调用信息

6.7 相关工作

为提高 Python 程序开发效率，大量研究研究者对 Python 程序的静态分析技术进行了深入研究，主要包括静态标注、静态类型推导和静态检查工具等。

静态标注是指在静态时对对象属性进行描述，从而为类型检查、编译优化、文档自动生成等提供良好支持。G. van Rossum 提出了可选静态类型声明，可用于编译优化和类型缺陷定位[53]。该方法关注于对象的具体类型，而 PyConcept 关注于对象的行为。更重要地，该方法需要修改 Python 语法，不能应用于既有 Python 代码。Python 3.0 对 Python 2.6 进行了扩充，引入了函数标注和抽象基类。函数标注提供了一种标准的方式来注解函数的参数和返回值[54]。这种标注可以是任意合法的 Python 表达式，如字符串、类型等。应用程序开发者和工具开发者可以根据自身需要关注具有特定形式的标注。类似地，PyConcept 也提供了一种统一的方法来标注对象的类型约束。然而，PyConcept 可对任意对象进行标注，不限于函数参数。此外，它关注于类型约束，有统一而严格的语义。抽象基类描述了类实例应具备的行为，支持函数 *isinstance* 和 *issubclass* 的重载[55]。Python3.0 定义了一组基本的抽象基类，如 *Hashable*、*Iterable*、*Mapping*、*Sequence* 等。与此类似，PyConcept 也通过对象行为来表达类型约束。进一步地，PyConcept 的预定义库包含了基本抽象基类所表达的概念，如 *Iterable* 等。换句话说，PyConcept 的表达能力强于抽象基类。再者，与 PyConcept 相比，抽象基类是一种入侵式机制，不仅难以向后兼容大量既有代码，还导致抽象基类难以自动生成。最后，它着眼于在运行时判断实例的类型，因而难以处理非实例对象，如函数参数等。

静态类型推导通过静态分析来推断对象类型，以进行编译优化和类型缺陷检查等。J. Aycock 提出了一种类型推导算法 *ATI*，能在 Python 的一个受限子集上进行类型推导[56]。B. Cannon 提出了一种局部化类型推导算法，可推导局部变量的类型[57]。然而，Python 动态类型的本质决定了静态类型推导只能获取部分对象的类型。

此外，这种技术难以推导函数参数的类型。

静态检查工具利用静态分析技术定位代码缺陷和不良软件设计。PyChecker 不仅能检查“使用未定义变量”和“传递错误的参数个数”等类型缺陷，还能判断“给定模块是否满足一批预定义的编码规则”，如在同一作用域内重定义函数等[58]。Pylint 扩展了 PyChecker，增加了检查“单行代码长度”、“变量命名是否规范”以及“接口是否实现”等特性[59]。然而，它们只能检查简单的类型缺陷，难以发现诸如图 6.1 的行 49 这样较为复杂的情况。

我们的工作借鉴了现有技术的合理思路。首先，PyConcept 采纳了静态标注的思想，构建了预定义 concept 库，并在此基础上建立了运算符、内建函数和控制流语句的类型约束。这有效提高了 concept 的抽象性和可读性，降低了构建 concept 的难度。此外，在 PyConcept 的自动生成和检查过程中，利用了类型推导技术来高效地构建函数调用图和确定对象的类型。

除 Python 相关技术外，其他领域中也存在与 PyConcept 有相似之处的技术。接口描述语言是一个典型例子。它通过描述组件接口来使使用不同语言编写的组件之间能够相互通信[120]，如 CORBA[121]、WSDL[122]等。本质上，PyConcept 也是在表达对象接口。然而，两者之间有着完全不同的设计目标。接口描述语句独立于编程语言，在不同语言之间起着“桥”的作用，而 PyConcept 则用于表达 Python 对象的类型约束。

6.8 本章小结

传统上，Python 程序的重构正确性检查主要依赖于测试。然而，Python 的动态类型增加了在测试过程中定位类型缺陷的难度，这给重构正确性检查带来了困难。虽然静态分析可以定位类型缺陷，但需要静态类型信息的辅助。为此，我们提出了 Python 类型约束系统 PyConcept，能够有效支持类型缺陷检查、重构工具开发以及 IDE 增强等多种软件工程活动。在此基础上，提出了 PyConcept 的构建算法 *generate_concept*，它不仅降低了使用 PyConcept 的成本，还为其它以 PyConcept 为基础的算法提供自动生成类型约束的服务。进一步地，还提出了 PyConcept 的检查算法 *check_func_call*，可以较好地定位类型缺陷，从而从一定程度上克服现有方法在检查类型缺陷方面的不足。

第七章 物理重构原型系统 Alchemist 的设计与实现

物理重构通常涉及软件的多个包和文件，需要自动化工具的支持。为此，我们实现了一个物理重构原型系统 Alchemist，以提高物理重构的效率，并为各种物理重构技术提供可扩展的实验平台。本章首先讨论了 Alchemist 的设计目标与总体框架，接着依次介绍了 Alchemist 的主要模块。

7.1 设计目标与总体框架

- 在设计与实现 Alchemist 时，应保持简单和清晰。Alchemist 是一个基础性的实验平台，需要被持续地扩展与完善。在这一过程中，简单性和清晰性是最重要的，因为几乎所有其他优秀的属性都只能跟着它们而来[82]。这就要求 Alchemist 的设计简单而优雅，不包含任何重复，具有简单而整齐的代码和接口，并通过了严格的测试。
- Alchemist 应由一组可组合的工具构成。与逻辑重构类似，物理重构也是一个“识别-重构-评估”的迭代过程，其每一个环节都包含多种方法。Alchemist 将每种方法实现为一个独立的工具。而且，这些工具可通过文本文件进行通信。这样设计的好处有四点：第一是通过“分治法”降低 Alchemist 实现的复杂性[123]；第二是提高 Alchemist 的易用性。用户可以根据需求选择部分工具，而不必处理与需求无关的复杂性；第三是允许工具进行组合，以产生更为丰富的功能；第四是降低系统各模块之间的耦合性，从而有助于提高 Alchemist 的可扩展性。任何工具只需满足 Alchemist 基于文本流的通信协议，即可与已有工具进行配合。这不仅使得我们可以使用多种程序设计语言开发 Alchemist，还极大地提高了 Alchemist 的可扩展性。
- Alchemist 应具备较好的可配置性。物理重构的每个环节都可能对应多种策略，而各种策略往往存在多种配置方案。为了通过实验来快速地比较各种方案的性能与效果，Alchemist 中的各个工具应该能够在不需要重新编译和连接的情况下，改变它所使用的配置方案。
- Alchemist 中任何模块输出的结果应该易于分析。首先，这些输出应该是人工可读的。这样，用户能够直接阅读和理解输出，有助于快速地获得反馈。其次，输出应该是机器可读、可操控的。这有助于用户使用诸如 Microsoft Excel、SPSS 这样高级的数据分析工具对结果进行进一步处理。最后，输出应是可调试的。这有助于根据输出定位 Alchemist 中的软件缺陷。

根据以上目标，我们采用 MVC 模式[67]，将 Alchemist 划分为三个部分：抽象模块（模型）、分析模块（视图）以及重构模块（控制器）。图 7.1 给出了 Alchemist 的总体框架。

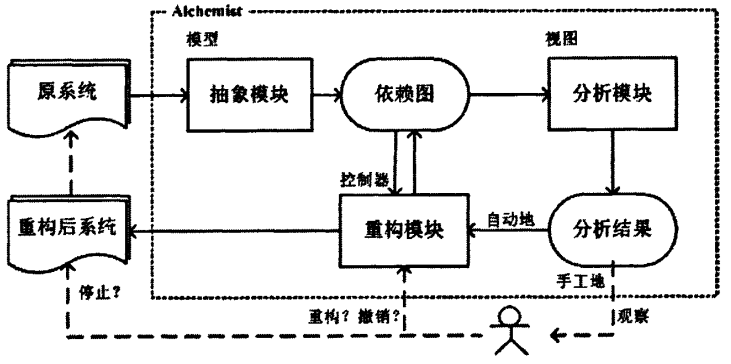


图 7.1 Alchemist 的总体框架

如图 7.1 所示，抽象模块通过静态分析生成系统的依赖图，包括物理依赖图和逻辑依赖图。基于依赖

图，分析模块采用度量等手段评估系统的各种属性。这里，存在两种重构方案：自动重构和手工重构。对于自动重构，分析结果将直接输出到重构模块。对于手工重构，分析结果将以人工可读的方式显示给用户。然后，用户根据分析结果制定重构策略：若系统质量满足要求，则停止重构；若存在不良设计，则调用重构模块修改依赖图和源代码；若上一轮重构并未有效提高系统质量，则撤销修改。在重构模块中，我们实现了包括第 2 章和第 5 章所讨论的技术在内的一批重构方法。下面依次对三个模块进行详细讨论。

7.2 抽象模块

抽象模块是一个独立工具 GraphGenerator。它根据用户配置（配置文件）分析软件源代码来生成依赖图，并将依赖图记录在具备特定格式的文本文件中。目前，GraphGenerator 主要支持 C++和 Python 这两种语言的解析。对于 C++，依赖图主要包括组件编译依赖图、类继承依赖图、类使用依赖图以及混合依赖图。混合依赖图是指同时包括多种类型依赖的依赖图。对于 Python，依赖图包括组件导入依赖图和模块数据依赖图。这里，组件导入依赖图反映组件之间的导入关系，与 C++的编译依赖图类似。在 Alchemist 中，用户可以通过配置文件指定生成特定类型的依赖图。由于依赖图的构建比较复杂，我们使用生成器模式[67]来实现依赖图生成器 *GraphBuilder*。另外，我们采用 Boost Graph Library（BGL）来实现各种依赖图[88]。这样设计的好处有两点：第一是 BGL 是开源的 C++程序库，提供了多种图的储存结构和一批高效的图论算法，有助于实现高质量的依赖图；第二是它包含一组定义良好的图访问和修改接口，允许用户定制“访问者”以在图的遍历过程中收集特定信息，从而为分析模块和重构模块提供良好的基础设施。详细过程（具体实现代码）见图 7.2。

```
DependenceGraph* GraphBuilder::Create(std::string const& inPath)
{
    fs::path rootPath(inPath, fs::native);      // 被分析项目所在路径
    dg_.reset(new DependenceGraph(rootPath));  // 指向 DependenceGraph 的 auto_ptr
    CreateElements();                           // 创建逻辑或物理实体
    AddDependences();                           // 添加依赖边

    return dg_.release();
}
```

图 7.2 依赖图建立过程

根据图 7.2，GraphBuilder 的构建过程分为两步。第一步分析系统的目录结构，根据用户指定的依赖图类型建立逻辑实体或物理实体，即依赖图的节点。在第一步中，我们通过 Boost Filesystem Library（BFL）来遍历系统中的所有目录和文件，并建立对应的实体[88]。第二步依次扫描系统中每个组件的源代码，抽取依赖关系，即依赖图的边。图 7.3 给出了第二步 *AddDependences* 的实现。

```
void GraphBuilder::AddDependences()
{
    PhysicalElements& elements = pdg_-> GetElements();       // 获取系统中所有物理实体
    DefIndentifier ind;                                       // 定义语法元素的识别器
    size_t const size = elements.size();
    for(size_t i=0; i<size; ++i)
    {
        if(elements[i].IsFile())
        {
            std::string text;
            Alchemist::LoadFile(elements[i].GetPath(), text);  // 读取文件内容
            DefIndentifier::Set s;                             // 将匹配结果存放于 s
            ind.Search(text, s);                               // 识别指定语法元素
            AddEdge(elements[i].GetPathString(), s);           // 根据 s 的结果添加依赖边
        }
    }
}
```

图 7.3 AddPhsicalDependences 的实现

在图 7.3 中，*AddDependences* 首先根据配置文件中指定的依赖图类型和被分析项目所用的语言识别相应的语法元素，然后通过进一步解析所识别的内容来添加依赖边。以继承依赖图为例，我们首先应识别继承列表，然后分析列表中使用的类以添加继承依赖。不同的语言需要不同的解析方法。对于 C++，其解析十分困难[70, 124]，且我们的研究重点在分析模块和重构模块，因此 GraphGenerator 采用正则表达式配合编码规范的策略来匹配指定的语法元素。例如，若编码标准规定数据成员以下划线 "_" 结尾，则可以通过正则表达式"^\w+_"来识别数据成员。GraphGenerator 允许在配置文件中配置编码规范以适应不同的系统。而 Python 的解析相对简单，可直接调用 CPython 的解析模块生成 Python 的抽象语法树。

由图 7.3 可知，构建依赖图的基本流程是稳定的，但是识别策略会随着被解析的语言和依赖类型而发生变化，这些策略利用条件分派的方式进行硬编码是不可取的。因此，我们采用 Strategy 模式[67]，使得具体的匹配策略可以独立于构建算法的流程而变化。这里，Strategy 模式的实现分为两层：语言和需要识别的语法元素。此外对于 C++，复杂的识别策略由基本识别策略组合而成。这里，基本识别策略对应于基本语法元素，如继承列表、类成员等。因此，我们使用组合模式[67]，使得匹配策略可以进行任意组合。图 7.4 是抽象模块的 UML 类图。其中，类 *GraphBuilder* 创建依赖图；类 *DefSearch* 及其子类负责搜索指定的语法元素；类 *DependenceGraph* 包装 BGL::graph，以对依赖图进行抽象；类 *PhysicalElement* 及其子类表达物理实体。由前面章节的讨论可知，在 C++和 Python 中，组件分别对应于类和模块，所以我们不单独建立"类"和"模块"的抽象。

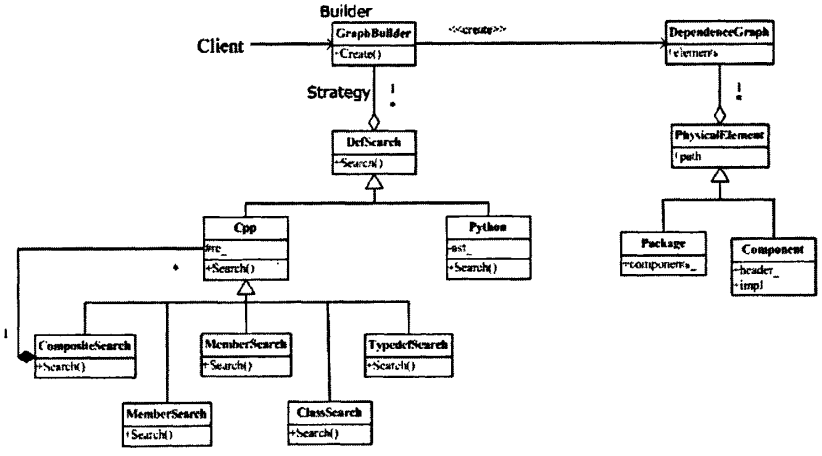


图 7.4 抽象模块的 UML 类图

7.3 分析模块

分析模块主要由一组度量工具组成。其中，每个工具对应一种分析方法。表 7.1 列举了分析模块包含的度量工具。此外，为方便 Microsof Excel 这样的高级数据分析工具对度量结果进行挖掘，分析模块的结果均以 CSV 格式存储于文本文件中。

在 Alchemist 中，度量工具首先根据抽象模块生成的依赖图文件构建依赖图。然后，遍历依赖图收集计算度量值所需的信息。这一步的实现采用了 BGL 中的访问者模式[88]。在通常情况下，BGL 通过访问者对图论算法进行控制，其原理与 STL 的仿函数类似[124]。最后，根据度量公式计算度量值。下面，我们结合度量工具 RC 对上述计算过程进行详细讨论。图 7.5 给出了 RC 度量的 Python 实现。由图 7.5 可知，访问者判断所访问的边或顶点是否满足给定的条件：若满足条件，则执行相应的操作。例如，函数 *RC* 通过对象 *rc_visitor* 收集计算所需的信息：若依赖边的两个端点都在 *package* 中，则将 *rc_visitor* 的属性 *relations* 增加 1。更为重要地，度量工具可以使用不同的语言编写。例如，除 Python 外，RC 度量的实现还能够使用 Ruby、C++、C#、Java 等。事实上，Alchemist 基于文本文件进行相互通信，任何遵守依赖图格式的度量工具均可

利用抽象模块生成的依赖图，而不必受限于 Alchemist 中其它工具所使用的编程语言。

表 7.1 分析模块中的度量工具

工具	描述	依赖图		设计原则
		C++	Python	
DS	计算对指定组件存在编译依赖或导入依赖的组件数	编译依赖图	导入依赖图	低构建代价原则
RS	计算编译依赖或导入依赖于指定组件的组件数	编译依赖图	导入依赖图	
CCD	计算指定包的累积组件依赖	编译依赖图	导入依赖图	
A	计算指定包的抽象性		无效	稳定抽象原则 稳定依赖原则
I	计算指定包的不稳定性	混合依赖图	数据依赖图	
D	计算指定包到序列 I=A 的距离	混合依赖图	无效	
RC	计算指定包的 RC 值	混合依赖图	数据依赖图	内聚性原则
EEC	计算指定包的 EEC 值	混合依赖图	数据依赖图	
HC	计算指定包的 HC 值	继承依赖图	数据依赖图	
SCC	计算指定包的 SCC 值	混合依赖图	数据依赖图	
WRC	计算指定包的 WRC 值	混合依赖图	数据依赖图	低耦合原则

```
def RC(graph_file):
    graph = dependence_graph.create_gragh_from(graph_file)  # 由依赖图文件构建图
    RCs = {}
    for e in graph.elements:
        if isinstance(e, dep_graph.Package):
            RCs[e.name] = compute_rc(e, graph)

    return RCs

def compute_rc(package, graph):
    size = len(package.modules)
    if size == 0:
        # 空包
        return 0
    elif size == 1:
        # 只含一个类的包
        return 1

    # 若两个端点都在 package 中，
    # rc_visitor.relations = rc_visitor.relations + 1
    v = rc_visitor(package)
    travers(graph, rc_visitor)     # 遍历图，利用 rc_visitor 计算 package 的包内依赖数

    return float(v.relations) / size  # 根据公式计算度量值
```

图 7.5 RC 度量的实现

7.4 重构模块

重构模块由一组重构工具组成。与分析模块类似，每个重构工具对应一个重构方法。表 7.2 列举了重构模块所包含的重构工具。由表 7.2 可知，这些工具可分为两类：原子重构工具和组合重构工具。其中，原子重构是指基本重构方法，如提取包。而组合重构由原子重构组成。例如，基于内聚性度量划分包可通过多次提取包来实现。事实上，组合重构还可以通过脚本文件（如 command shell 脚本、bash 脚本等）组合一组原子重构方法来完成。此外，第 5 章所提出的度量驱动的包结构重构可以根据分析模块的输出自动地调用重构模块，从而相当程度地提高了重构效率。最后，表 7.2 中的部分重构方法仅适用于 C++，如使用 PImpl 惯用法。

表 7.2 重构模块中的重构工具

工具	对应的重构方法	重构类型
ApplyPImpl*	使用 PImpl 惯用法	原子重构
MoveComponent	移动组件	
ExtractPackage	提取包	
ExtractAbstractPackage*	提取抽象包	
ExtractCommonPackage	提取公共包	
MergePackage	合并包	
MergeCoupledPackage	基于耦合性度量合并包	
RenamePackage	重新命名包	
DividPackage	基于内聚性度量划分包	组合重构
ReorganizePackage	基于度量重新组织包	
RefactorPackage	度量驱动的包结构重构	

注：*表示 Python 没有的重构方法

下面结合函数 *MoveComponent* 来讨论重构工具的实现，它对应的重构方法是移动组件。图 7.6 给出了 *MoveComponent* 的 C++实现。由图 7.6 可知，重构工具首先根据依赖图文件生成依赖图，然后按照重构方法的步骤修改原系统。特别地，与包相关的重构方法，如移动组件、提取包以及基于度量重新组织包等，会修改依赖图文件。这是因为这些重构方法会导致依赖图的改变。同样地，我们可以使用 Python、Ruby、Java 或 C#等来实现重构工具。不难看出，基于文本流的通信机制有效降低了 Alchemist 中的各个模块之间的耦合性，有助于提高 Alchemist 的可扩展性、可维护性与可理解性。

```
void MoveComponent(Component const&c,
                   std::string ofile, std::string nfile)
{
    auto_ptr<DependenceGraph> pdg  // 依赖图
    ReadPGraph(ofile, pdg);        // 从依赖图文件构建依赖图
    //修改原系统
    ModifyComponent(c, pdg);       // 修改 include 语句
    MoveFiles(c);                  // 移动文件
    //修改编译依赖图
    WriteGraph(nfile, pdg);
}
```

图 7.6 MoveComponent 的实现

7.5 本章小结

为了提高物理重构效率，对比不同的物理重构技术，我们研发了一个物理重构原型系统 Alchemist，为物理重构的研究提供了一个可复用的实验平台。本章首先介绍了 Alchemist 的设计目标和总体框架。然后，分别讨论了主要模块的实现和功能。由上述讨论可知，Alchemsit 由一组基于文本通讯的工具组成，具备较高的可扩展性，能够方便地增加新特性。此外，Alchemsit 还可与 Mircosoft Excel、SPSS 等高级分析工具进行良好地配合，从而增强它的数据处理能力。在未来工作中，我们将进一步集成更多的与物理重构相关技术，并为包括 Java 在内的多种语言的物理重构提供支持。

第八章 结束语

重构指在不改变软件外在行为的前提下，改善软件内部结构，从而在软件演化过程中优化软件质量，提高软件可理解性、可维护性和可扩展性等。二十多年来，人们对重构技术进行了广泛而深入的研究随着对重构技术研究的广泛与深入，取得了许多极具价值的研究成果。尽管如此，现有研究在处理大规模程序上仍然存在一些局限性。为此，本文在广泛调研的基础上，通过对物理重构、面向重构的包内聚性度量方法、度量驱动的包结构重构技术以及 Python 程序的重构正确性评估等的研究，从一定程度上解决了现有技术面临的困难。在重构方法方面，提出了新的重构方法，可以有效地优化系统的物理结构。在自动化重构方面，提出了两种面向重构的包内聚性度量方法，并以此为基础，讨论了度量驱动的包结构重构框架以及相应的方法体系，可以有效弥补手工重构的不足。在重构正确性检查方面，提出了 Python 类型约束系统以及相应的约束构建算法和约束检查算法，有助于提高动态语言的重构效率。具体而言，论文工作的主要成果表现在以下几个方面：

(1) 提出了物理重构的概念和方法体系，可在软件演化过程中优化物理结构的质量。与现有重构技术相比，物理重构关注于物理结构，后者对于大规模软件的开发、维护和重用有重要意义。与现有物理设计技术相比，它能够持续优化系统的物理结构，从而帮助系统适应不断变化的开发需求。

(2) 总结现有物理设计技术，并提出了一个物理重构目录。该目录不仅描述了重构方法的名称和步骤，还给出了根据不良物理结构的类型选择合适重构方法的启发式规则，对于开发者决定“何时”、“何地”以及“如何”进行物理重构有较强的指导意义。

(3) 提出了一种新的包内聚性 CRC，其基本思想是从客户行为考察包的内聚程度。CRC 内聚认为若多个类总是被共同重用，则它们之间存在紧密耦合。基于此，提出了基于客户使用的包内聚性度量方法 HC。与现有方法相比，HC 度量同时考虑了包内和包间的数据依赖，所以能有效地反映类之间的语义关系。此外，HC 度量通过向量来表达客户对包的使用方式，可区分性更高。

(4) 提出了基于相似上下文的包内聚性度量方法 SCC，通过类上下文来推断类之间的耦合关系。该方法认为两个上下文相似的类具有紧密的语义耦合。与已有方法相比，SCC 度量与 HC 度量均基于包上下文来计算包的内聚性，更符合包语义的复杂性。

(5) 利用大规模实验对各种包内聚性度量方法进行了深入研究。实验结果显示基于上下文的方法比基于数据流的方法更加有效，从而证明了包内数据依赖关系难以反映包的高层语义。实验结果还表明充足客户集可以保证基于上下文的方法能获得稳定而有意义的结果。

(6) 提出了度量驱动的包结构重构框架以及相应的方法体系，能够自动化地提高包结构的质量。本方法首先根据度量来识别存在设计缺陷的包结构，然后选择合适的重构方法进行重构。与手工重构相比，它不但大弧度地提高了重构效率，而且有效降低了引入软件缺陷的可能。

(7) 提出了 Python 的类型约束系统 PyConcept，可以支持类型缺陷检查等多种软件工程活动。与现有方法相比，PyConcept 能够有效表达 Python 基于结构一致性的类型约束。基于 PyConcept，给出了类型约束的构建算法，可以用于自动生成 Python 对象的类型约束。进一步地，还给出了类型约束的检查算法，能够高效地识别类型缺陷，从而有助于检查 Python 程序的重构正确性。

不过，受到时间等因素的限制，论文只是研究了物理重构、面向重构的包内聚性度量、度量驱动的包结构重构以及 Python 程序的重构正确性评估中的部分问题，对于相关理论和技术还有待进一步探索。在未来的研究中，除了对论文中的很多不足之处进行深入研究之外，还需对以下方面进行深入探索：

(1) 随着动态语言的快速发展，越来越多的大型软件使用动态语言进行开发。相对于静态语言，动态语言有着不同的代码组织方式，从而需要不同的物理设计方法和设计原则。因此，有必要对动态语言的物理重构进行研究，包括相关的物理设计的评估标准以及常用的重构方法等。

(2) 基于上下文度量模块内聚性是一种合理的研究思路。围绕这一思路，可以提出一组包内聚性度量

方法。HC 度量和 SCC 度量只是其中两种方法，还存在一定的不足。因此，还需要深入研究包的分类方法，针对不同种类的包提出不同的度量方法。此外，现有类内聚性度量主要依赖类内部数据依赖关系，同样存在很多失效情况，所以有必要研究基于上下文的类内聚性度量方法。

(3) 程序库有助于提高软件开发的效率，其设计质量对于库的普及度有极大影响。软件度量可以高效地评估程序库的质量。然而，如何更好地度量程序库中包的内聚性，还有待进一步研究。所以，有必要进行更为广泛的实验。

(4) 度量驱动的重构技术可以有效地提高重构效率，从较大程度上避免手工重构的局限性。因此，应进一步改进度量驱动的包结构重构技术。此外，类和函数的重构仍然需要度量的支持，所以可以对度量驱动的类或函数的重构进行深入研究。

(5) PyConcept 的预定义库可以从很大程度上提高编写和生成 Python 对象类型约束的效率。因此，有必要深入研究各种 Python 项目，从中抽象出一批可以广泛使用的 concept。此外，不同领域拥有不同的类型约束，所以可以开发针对特定领域的 PyConcept 库。

(6) PyConcept 的构建算法为不同参数生成的 concept 之间很可能存在精化或使用关系。然而，目前的算法尚不能自动识别。因此，可利用诸如形式概念分析[125]等方法对生成结果进行进一步优化。

(7) 程序分析技术有助于定位软件缺陷。然而对于动态语言，仅依赖静态程序分析难以获得软件的动态语义。本文通过 PyConcept 来为 Python 程序提供类型信息。另一种可行的解决方案是通过记录对象在多次执行过程中的信息来推断对象的可能类型。

致 谢

首先，衷心感谢我的导师徐宝文教授。在硕士生和博士生阶段的学习和工作中，徐老师一直给我无微不至的关心、指导和帮助。徐教授为人正直、宽厚自律、学风严谨、知识渊博、工作认真、精益求精，令我在为人处世、治学研究、协作交流等方面都受益匪浅。回想到自 2004 年师从徐教授进行软件理论的研究，往事历历在目，更觉师恩深厚，难以回报。

感谢周晓宇老师从硕士生阶段以来对我的关心和帮助，他的友善和宽容令我难忘。感谢周毓明老师在程序度量领域的帮助，他严谨的治学态度、诚恳的为人作风，令我受益匪浅。感谢陈林老师，他在程序重构方面给予了我较大的帮助。感谢卢红敏老师在生活上的相伴，她的善良和热情是我博士生活的一抹亮色。感谢师兄钱巨在程序分析领域的帮助，他扎实的专业知识以及执着的求知态度使我受到较大启发。还要感谢谢晓园同学长期以来与我一起进行学术研究方法的讨论。

感谢师兄史亮博士带领我进入程序重构和动态程序设计语言领域。多年来，我们一直进行紧密、愉快的学术合作。在漫长的求学过程中，他给予了我学术和生活上的极大支持，帮助我渡过了许多困难。

我还要感谢东南大学软件工程与理论实验室的许蕾、聂长海、陆建江、姜淑娟、戚晓芳、钱俊彦、吴军华、周吴杰、查日军、王军、张德平等老师，刘园、沈小丰、黄文伟、张舒、李平华、陈津、鞠小林、任充慧、汪鹏、曹璟、曾奕、崔自峰、章晓芳、康达周、屈波、杨彬、蒋冀翔、吴重强、谢书丹、孙莉、余斌、周超洪、李言辉、徐峻岭、万晓民、王子元、董国伟、徐洁、解凯、梁陈良、周晶晶、闵洪波、何惠贞、李亚军、周金、姜伟、陶波、张婷婷、单华茂、余超、尹恒、李磊芳、赵思奇、张实睿等同学，以及毕业的张卫丰、张迎周等学长对我的热情帮助。

感谢我的父亲和母亲。他们给予了我生命，教会了我做人做事的道理和方法。在过去的二十七年中，无论得意还是失意总有他们的默默相伴。我的每一步成长都离不开他们不计回报的支持。他们是我世上最珍贵的亲人，更是我人生的挚友。他们无私的爱值得用一生去体会和回报。

感谢所有帮助、关心过我的朋友们。

参考文献

[1] T. Mens, Member, IEEE, T. Tourwé. A Survey of Software Refactoring. IEEE Transactions on Software Engineering, 2004, 30(2):126-139.

[2] C. Zhang, H. A. Jacobsen, IEEE Member. Refactoring Middleware with Aspects. IEEE Transaction on Parallel and Distributed Systems, (14)11: 1058-1073, 2003.

[3] 王忠杰, 徐晓飞, 战德臣. 面向复用成本优化的构件重构方. 软件学报, 16(12): 2157 -2165, 2005.

[4] K. Beck. Extreme Programming Explained: Embracing Change (Third Edition). Pearson Education, 2006.

[5] M. Flower. Refactoring: Improving the Design of Existing Code. Addison Wesley, 1999.

[6] B. Foote, W. F Opdyke. Lifecycle and Refactoring Patterns That Support Evolution and Reuse. ACM Press/Addison-Wesley Publishing, 1993.

[7] S. Demeyer, S. Ducasse, O. Nierstrasz. Object-Oriented Reengineering Patterns. Morgan Kaufmann, 2002.

[8] D. Dig, R. Johnson. The Role of Refactorings in API Evolution. In Proceedings of the 21st IEEE International Conference on Software Maintenance, pp. 389 – 398, 2005.

[9] Y. Ping, K. Kontogiannis. Refactoring Web Sites to the Controller-Centric Architecture. In Proceedings of the 8th European Conference on Software Maintenance and Reengineering, pp. 204-213, 2004.

[10] G. Snelting, F. Tip. Reengineering Class Hierarchies Using Concept Analysis. In Proceedings of Foundations of Software Engineering, 1998.

[11] W. F. Opdyke. Refactoring Object Oriented Frameworks. PHD Thesis, University of Illinois, 1992.

[12] J. Kerievsky. Refactoring to Patterns. Addison Wesley, 2004.

[13] E. M. Clarke. Model checking. Springer Berlin / Heidelberg, 1999.

[14] M. Y. Vardi, P. L. Wolper. An Automata-theoretic Approach to Automatic Program Verification. International Business Machines Inc. 1986

[15] S. Muchnick. Advanced Compiler Design and Implementation. Elsevier Science, 1997.

[16] J. Lakos. Large-Scale C++ Software Design. Addison Wesley, 1996.

[17] S. Counsell, S. Swift, J. Crampton. The Interpretation and Utility of Three Cohesion Metrics for Object-Oriented Design. ACM Transactions on Software Engineering and Methodology, 15(2):123-149, 2006.

[18] L. C. Briand, J. Daly, J. Wuest. A Unified Framework for Cohesion Measurement in Object-Oriented Systems. Empirical Software Engineering, 3(1) :65-117, 1998.

[19] S. Chidamber, C. Kemerer. Towards A Metrics Suite for Object Oriented Design. In Proceedings of Conference on Object Oriented Programming Systems Languages and Applications, pp.197-211, 1991.

[20] S. Chidamber, C. Kemerer. A Metrics Suite for Object Oriented Design. IEEE Transactions on Software Engineering, 20(6): 476-493, 1994.

[21] M. Hitz, B. Montazeri. Measuring Coupling and Cohesion in Object-Oriented Systems. In Proceedings of International Symposium on Applied Corporate Computing, pp.25-27, 1995.

[22] H. Chae and Y. Kwon. A Cohesion Measure for Classes in Object-Oriented Systems. In Proceedings of the 5th International Software Metrics Symposium, pp.158-166, 1998.

[23] H. Chae, Y. Kwon, D. Bae. Improving Cohesion Metrics for Classes by Considering Dependent Instance Variables. IEEE Transactions on Software Engineering, 30(11): 826-832, 2004.

[24] L. Ott, J. Bieman, B. Kang, B. Mehra. Developing Measures of Class Cohesion for Object-Oriented Software. In Proceedings of Annual Oregon Workshop on Software Metrics, pp.1-11, 1995.

[25] A. Marcus, D. Poshyvanyk. The Conceptual Cohesion of Classes. In Proceedings of the 21st International Conference on Software Maintenance, pp.133-142, 2005.

[26] E. B. Allen, T. M. Khoshgoftaar, Y. Chen. Measuring Coupling and Cohesion of Software Modules: An Information Theory Approach. In Proceedings of the 7th International Software Metrics Symposium, pp. 124-134, 2001.

[27] Yuming Zhou. Research on Software Measurement. PHD Thesis, Southest University, 2002.

[28] Yuming Zhou, Baowen Xu, Jianjun Zhao, Hongji Yang. ICBMC: An Improved Cohesion Measure for Classes. In Proceedings of the 18th International Conference on Software Maintenance, pp.44-53, 2002.

[29] Yuming Zhou, Lijie Wen, Jianmin Wang, Yujian Chen, Hongmin Lu, Baowen Xu. DRC: A Dependence Relationships based Cohesion Measure for Classes. In Proceedings of the 10th Asia-Pacific Software Engineering Conference, pp.215-223, 2003.

[30] Yuming Zhou, Jiangtao Lu, Hongmin Lu, Baowen Xu. A Comparative Study of Graph Theory-Based Class Cohesion Measures. ACM SIGSOFT Software Engineering Notes, 29(2):1-6, 2004.

[31] Zhenqiang Chen, Yuming Zhou, Baowen Xu, Jianjun Zhao, Hongji Yang. A Novel Approach to Measuring Class Cohesion based on Dependence Analysis. In Proceedings of International Conference on Software Maintenance, pp.377-383, 2002.

[32] 陈振强, 徐宝文. 一种基于依赖性分析的类内聚度度量方法. 软件学报, 14(11):1849-1856, 2003.

[33] 李必信, 朱平, 谭毅, 李宣东, 郑国梁. 基于数据切片度量 java 内聚性. 软件学报, 2(12):1851-1858, 2001.

[34] C. Bonja, E. Kidanmariam. Metrics for Class Cohesion and Similarity between Methods. In Proceedings of the 44th Annual Southeast Regional Conference, pp.91-95, 2006.

[35] J. Bansiya, L. Etzkorn, C. Davis, W. Li. A Class Cohesion Metric for Object-Oriented Designs. Journal of Object-Oriented Programming, 11(8):47-52, 1999.

[36] J. Bieman, B. Kang. Cohesion and Reuse in An Object-Oriented System. In Proceedings of the 1995 Symposium on Software Reusability, pp.259-262, 1995.

[37] L. Briand, P. Devanbu, W. Melo. An Investigation into Coupling Measures for C++. In Proceedings of the 19th International Conference on Software Engineering, pp.412-421, 1997.

[38] L. Briand, J. Daly, V. Porter, J. Wust. A Comprehensive Empirical Validation of Design Measures for Object-Oriented Systems. In Proceedings of 15th International Software Metrics Symposium, pp.246-257, 1998.

[39] L. Briand, J. Wüst, S. Ikonomovski, H. Lounis. Investigating Quality Factors in Object-Oriented Designs: An Industrial Case Study. In Proceedings of the 21st International Conference on Software Engineering, pp.345-354, 1999.

[40] L. Briand, J. Wüst. Modeling Development Effort in Object-Oriented Systems Using Design Properties. IEEE Transactions on Software Engineering, 27(11):963-986, 2001.

[41] L. Briand, J. Wüst. The Impact of Design Properties on Development Cost in Object-Oriented Systems. IEEE Transactions on Software Engineering, 28(12):260-271, 2002.

[42] Yuming Zhou, H. Leung. Empirical Analysis of Object-Oriented Design Metrics for Predicting High and Low Severity Faults. IEEE Transactions on Software Engineering, 32(10):771-789, 2006.

[43] A. Marcus, D. Poshyvanyk, R. Ferenc. Using the conceptual cohesion of classes for fault prediction in object-oriented systems. IEEE Transactions on Software Engineering, 34(2), 2008.

[44] S. Counsell, E. Mendes, S. Swift. Comprehension of Object-oriented Software Cohesion: The Empirical Quagmire. In Proceedings of the 10th International Workshop on Program Comprehension, pp.33-42, 2002.

[45] R. Martin. Agile Software Development Principles, Patterns, and Practices. Prentice Hall, 2002.

[46] S. Patel, W. Chu, R. Baxter. A Measure for Composite Module Cohesion. In Proceedings of the 14th International Conference on Software Engineering, pp.38-48, 1992.

[47] W. C. Chu, C. Cheng, S. Patel. Data Type Based Measurement of Composite Module Cohesion. Journal of Information Science and Engineering, 13(1):63-83, 1997.

[48] L. Briand, S. Morasca, V. R. Basili. Defining and Validating Measures for Object- Based High-Level Design. IEEE Transactions on Software Engineering, 25(5): 722-743, 1999.

[49] Baowen Xu, Zhenqiang Chen, Jianjun Zhao. Measuring Cohesion of Packages in Ada95. In Proceedings of the ACM SIGAda Annual International Conference, pp.62-67, 2003.

[50] Python. http://www.python.org/. 2008.

[51] Ruby. http://www.ruby-lang.org/en/. 2008.

[52] Tcl. http://www.tcl.tk/man/. 2008.

[53] G. van Rossum. Adding Optional Static Typing to Python, http://www.artima.com/weblogs /viewpost.jsp?thread =85551, 2004.

[54] C. Winter, T. Lownds. Function Annotations. http://www.python.org/dev/peps/pep-3107/, 2007.

[55] Guido van Rossum. Introducing Abstract Base Classes. http://www.python.org/dev/peps/ pep-3119/, 2007.

[56] J. Aycock. Aggressive Type Inference. Department of Computer Science, University of Victoria, Canada, 1999.

[57] B. Cannon. Localized Type Inference of Atomic Types in Python [MS Thesis]. San Luis Obispo: California Polytechnic State University, 2005.

[58] PyChecker. http://pychecker.sourceforge.net. 2008

[59] Pylint. http://pydev.sourceforge.net/pylint.html. 2008.

[60] S. Ducasse, M. Rieger, S. Demeyer. A Language Independent Approach for Detecting Duplicated Code. In Proceedings of the 15th IEEE International Conference on Software Maintenance, pp.109-118, 1999.

[61] M. Balazinska, E. Merlo, M. Dagenais, B. Lagu, and K. Kontogiannis. Advanced Clone-Analysis to Support Object-Oriented System Refactoring. In Proceedings of the 7th Working Conference on Reverse Engineering, pp.98-107, 2000.

[62] Y. Kataoka, M. D. Ernst, W. G. Griswold, D. Notkin. Automated Support for Program Refactoring Using Invariants. In Proceedings of the 17th IEEE International Conference on Software Maintenance, pp.736-743, 2001.

[63] F. Simon, F. Steinbrückner, C. Lewerentz. Metrics Based Refactoring. In Proceedings of European Conference on Software Maintenance and Reengineering, pp.30-38, 2001.

[64] G. Kiczales, J. Lamping, A. Mendhekar, C. Maede, C. Lopes, J. M. Loingtier, J. Irwin. Aspect-oriented Programming. In Proceedings of the 11th European Conference on Object-Oriented Programming, pp.220-242, 1997.

[65] C. Zhang, H. Jacobsen, Member, IEEE. Refactoring Middleware with Aspects. IEEE Transactions on Parallel and Distributed Systems, (14)11:1058-1073, 2003.

[66] M. P. Monteiro, J. M. Fernandes. Towards A Catalog of Aspect-Oriented Refactorings. In Proceedings of the 4th International Conference on Aspect-oriented Software Development, pp.111-122, 2005

[67] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, 1998.

[68] H. Sutter. Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions. Addison Wesley, 1999.

[69] H. Sutter. A. Alexandrescu. C++ Coding Standards: 101 Rules, Guidelines, and Best Practices. Addison Wesley, 2004.

[70] B. Stroustrup. The Design and Evolution of C++. Addison-Wesley, 1994.

[71] S. Dewhurst. C++ Gotchas. Addison-Wesley, 2003.

[72] S. Meyers. Effective C++: 55 Specific Ways to Improve Your Programs and Designs (Third Edition). Addison Wesley Professional, 2005.

[73] S. McConnell. Code Complete (Second Edition). Microsoft Press, 2004.

[74] E. Evans. Domain-Driven Design: Tacking Complexity in the Heart of Software. Addison Wesley/Pearson, 2004.

[75] gmake. http://www.gnu.org/software/make/. 2008.

[76] mkmf. http://www.gfdl.gov/~vb/mkmf.html. 2007.

[77] K. Paton. Extraction and Examination of Relations in C++: Principles of Good Physical Design, Courtesy of Lakos and Martin. Dr. Dobb's Journal, 26(10): 28-34, 2001.

[78] E. Hautus. Improving Java Software Through Package Structure Analysis. In Proceedings of International Conference Software Engineering and Applications, 2002.

[79] A. Alexandresc. Modern C++ Design: Generic Programming and Design Patterns Applied. Addison Wesley, 2001.

[80] M. A. Cusumano, R. W. Selby. How Microsoft Builds Software. Communication of the ACM, 40(6):3-61, 1997.

[81] E. Karlsson, L. Andersson, P. Leion. Daily Build and Feature Development in Large Distributed Projects. In Proceedings of the 22nd International Conference on Software Engineering, pp.649-658, 2000.

[82] B. W. Kernighan, R. Pike. The Practice of Programming. Addison Wesley, 1999.

[83] H. Sutter. Exceptional C++ Style. Addison Wesley, 2004.

[84] Refactoring. http://www.refactoring.com. 2008.

[85] C. Larman. Applying Uml and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process (Second Edition), Prentice Hall, 2004.

[86] E. Yourdon, L. Constantine. Structured Design. Prentice Hall, 1979.

[87] B. Stroustrup. The C++ Programming Language (Special Edition). Person Education, 2000.

[88] Boost. http://www.boost.org/doc/libs/1_37_0/libs/libraries.htm. 2008.

[89] K. S. Mathias, J. H. Cross, T. D. Hendrix, L. A. Barowski. The Role of Software Measures and Metrics in Studies of Program Comprehension. In Proceedings of the 37th Annual Southeast Regional Conference, pp.13-70, 1999.

[90] L. C. Briand, S. Morasca, V. R. Basili. Property-Based Software Engineering Measurement. IEEE Transactions on Software Engineering, 22(1):68-85, 1996.

[91] L. C. Briand, J. W. Daly, J. K. Wüst. A Unified Framework for Coupling Measurement in Object-Oriented Systems. Transactions on Software Engineering, 25(1):91-121, 1999.

[92] Yuming Zhou, H. Leung, P. Winoto. MNav: A Markov Model-Based Web Site Navigability Measure. IEEE Transactions on Software Engineering, 33(12):869-890, 2007.

[93] R. W. Hamming. Error detecting and Error Correcting Codes. Bell System Tech Journal, 29(2):147-160, 1950.

[94] Liang Shi, Baowen Xu, Xiaoyuan Xie. An Empirical Study of Configuration Strategies of Evolutionary Testing. International Journal of Computer Science & Network Security, 6(1A):44-49, 2006.

[95] Liang Shi. Research on Test Data Automatic Generation. PHD Thesis, Southeast University, 2006.

[96] Xiaoyuan Xie, Baowen Xu, Liang Shi. A Dynamic Optimization Strategy for Evolutionary Testing. In Proceedings of the 20th Asia-Pacific Software Engineering Conference, pp.15-17, 2005.

[97] J. Han, m. Kamber. Data Mining: Concepts and Techniques. Morgan Kaufmann, 2001.

[98] J. Borges, M. Levene. An Average Linear Time Algorithm for Web Usage Mining. Information Technology and Decision Making, 3(2):307-319, 2004.

[99] Django. http://www.djangoproject.com/, 2008.

[100] A. Martelli. Python in A Nutshell (2nd Edition). O'Reilly, 2006.

[101] S. M. Ross. Introductory Statistics (Second Edition). Academic Press, 2005.

[102] J. R. Quinlan. Programs for Machine Learning. Morgan Kaufmann, 1993.

[103] T. J. McCabe. A Complexity Measure. IEEE Transactions on Software Engineering, 2(4): 308-320, 1976.

[104] R. E. Tarjan. Data Structures and Network Flow Algorithms. In Proceedings of Regional Conference Series in Applied Mathematics, 1983.

[105] M. Lorenz, J. Kidd. Object-Oriented Software Metrics: A Practical Approach. Prentice-Hall, 1994.

[106] S. Demeyer. S. Ducasse, O.Nierstrasz. Finding Refactorings via Change Metrics. ACM SIGPLAN Notices, 35(10): 166-177, 2000.

[107] W. Chun. Core Python Programming (2nd Edition). Prentice Hall PTR , 2006.

[108] J. Goerzen. Foundations of Python Network Programming. Apress, 2004.

[109] M. Lutz. Programming Python(3rd Edition). O'Reilly, 2006.

[110] Python. http://www.python.org/. 2008.

[111] G. van Rossum. Python Reference Manual. http://www.python.org/doc/2.5.2/ref/ ref.html/, 2008.

[112] D. Carlson. Eclipse Distilled. Addison Wesley/Pearson, 2006.

[113] A. Allowatt, S. Edwards. IDE Support for Test-Driven Development and Automated Grading in Both Java and C++. In Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology Exchange, pp.100-104, 2005.

[114] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. Reis, A. Lumsdaine. Concepts: Linguistic Support for Generic Programming in C++. In Proceedings of ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 291-310, 2006.

[115] A. Koenig, B. E. Moo. Templates and Duck Typing. C/C++ Users Journal June, 2005.

[116] F. P. Brooks. The Mythical Man-Month: Essays on Software Engineering. (Anniversary Edition). Addison Wesley, 1995.

[117] B. Steensgaard. Points-to Analysis in Almost Linear Time. In Proceedings of the 23rd ACM Symposium on Principles of Programming Languages, pp. 32-41, 1996.

[118] V. Sundaresan, L. Hendren, C. Razafimahefa, R. ValléeRai, P. Lam, E. Gagnon, C. Godin. Practical Virtual Method Call Resolution for Java. In Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, And applications, pp. 264-280, 2000.

[119] R. Snodgrass. The Interface Description Language: Definition and Use. Computer Science Press, 1989.

[120] CORBA. http://www.corba.org/. 2008.

[121] S. Graham, S. Simeonov, T. Boubez, G. Daniels, D. Davis, Y. Nakamura, R. Neyama. Building Web Services with Java: Making Sense of XML, SOAP, WSDL and UDDI. SAMS, 2003.

[122] E. S. Raymond. The Art of UNIX Programming. Addison-Wesley, 2004.

[123] S. B. Lippman. Inside the C++ Object Model. Addison Wesley, 1996.

[124] 侯捷. STL 源码剖析. 华中科技大学出版社, 2002.

[125] G. Bernhard, S. Gerd, W. Rudolf. Formal Concept Analysis: Foundations and Applications (Lecture Notes in Artificial Intelligence). Springer-Verlag, 2005.

附录 1 Django 中主要包的人工标注情况

包名	描述	标注
bin	由正交辅助工具组成，类似于 Windows/Linux 的 bin 目录。	低内聚
conf	负责 Django 的设置，包括 Web 应用的模板、Django 项目的模板、国际化以及 URL 设置。	难判定
conf.app_template	封装了 Web 应用的模板。	高内聚
conf.urls	主要用于设置 URL 到 Python 函数的映射。	高内聚
contrib	封装了创建 Web 应用的主要功能。	低内聚
contrib.admin	自动管理接口，允许内容提供者向网站添加内容。它包括了模板、模板标签、视图、辅助工具。	难判定
contrib.auth	处理用户权限。	高内聚
contrib.comments	封装了用户评论相关的功能。	高内聚
contrib.localflavor	实现针对不同国家和地区的不同风格，包括中国、美国、英国等。	难判定
contrib.redirects	负责维护重定向的映射。	高内聚
contrib.sessions	用于实现 session 相关的功能。	高内聚
contrib.sitemaps	负责创建和获取记录 sitmap 的 XML 文件。该文件用于向搜索引擎通知网站的结构和页面权重等。	高内聚
contrib.sites	封装了网站的管理机制。	高内聚
contrib.syndication	用于创建诸如 RSS 之类的聚合。	高内聚
core	封装了 Django 的核心基础功能。	低内聚
core.cache	封装了各种缓存处理机制，如数据库、文件等。该包中的类均支持函数 add、delete、get 和 set 等。	高内聚
core.handlers	处理请求时，用于发送或接受的信号。	高内聚
core.serializers	封装了各种序列化策略，包括 json、XML 等。该包中的类均支持函数 serialize 和 serialize 等。	高内聚
core.servers	封装了各种服务器机制，如 HTTP 和 FASTCGI 等。	高内聚
db	负责处理数据库相关功能，用于向领域层提供统一的数据库接口。	高内聚
db.backends	对各种数据库进行了包装，如 ado、oracle 和 mysql 等。	低内聚
db.backends.ado_mssql	用于包装 ado 数据库。	高内聚
db.models	对数据库的各种操作进行建模，如查询等。	高内聚
db.models.fields	对数据库的字段进行建模。	高内聚
dispatch	负责维护信号到信号处理器的映射，以实现信号发送者和处理者的解耦。	高内聚
middleware	封装了处理请求和应答的各种实现，如 gizp 压缩、事物处理等。该包中的类均支持函数 process_request 和 process_response。	高内聚
newforms	封装了处理表单的功能。	高内聚
template	负责实现 Django 的模板系统。	高内聚
template.loaders	Django 的模板加载功能。	高内聚
templatetags	实现了 Django 的模板标签	高内聚
utils	由各种独立的工具组成，如缓存、校验和、编码等。	低内聚
utils.simplejson	实现了简单 json 存储格式。	高内聚
views	实现视图功能，以向用户显示网站内容。	高内聚

附录 2 PyConcept 预定义库

预定义库能有效提高 PyConcept 的构建和检查效率。目前，预定义库主要包括描述 Container（容器）、Iterator（迭代子）以及内建操作参数的 concept。下面，我们列出预定义库的核心内容。

容器和迭代子

```
# 存在哈希值的对象：哈希表的键值
concept Hashable[obj]:
    requires ret : int
        obj.__hash__() -> ret

# 可迭代对象的接口：可利用它来遍历可迭代对象
concept Iterator[obj]:
    associate item
    obj.next() -> item
    requires Iterator[ret]
        obj.__iter__() -> ret

# 可迭代对象：可被 Iterator 遍历的对象，
# 即支持内建语句 for 的对象
concept Iterable[obj]:
    associate item = Iterator[obj].item
    requires Iterator[ret]
        obj.__iter__() -> ret


# 容器：提供了元素的测试功能，即检查元素是否
# 在 Container 中，支持内聚运算符 not in 和 in
concept Container[obj]:
    requires ret : bool
        obj.__contains__() -> ret


# 集合：可迭代有长度容器，
# 支持交、并、补、比较等操作
concept Set[obj]:
    requires Hashable[obj]
    requires Container[obj]
    requires Sized[obj]
    requires Iterable[obj]
    associate item = Iterable[obj].item
    requires Set[other] and ret : bool
        obj.__eq__(other) -> ret
    requires Set[other] and ret : bool
        obj.__le__(other) -> ret
    requires Set[other] and Set[ret]
        obj.__or__(other) -> ret
    requires Set[other] and Set[ret]
        obj.__xor__(other) -> ret
    requires Set[other] and Set[ret]
        obj.__and__(other) -> ret
    requires Set[other] and Set[ret]
        obj.__sub__(other) -> ret


# 可变集合：在 set 基础上，支持集合修改操作
concept MutableSet[obj]:
    requires Set[obj]
    associate item = Set[obj].item
    requires ret : bool
        obj.__add__(item) -> ret
    requires ret : bool
        obj.__discard__(item) -> ret
    obj.pop() -> item
    requires ret : bool
        obj.toggle() -> ret
    obj.clear()
    requires MutableSet[other]
        obj.__iand__(other)
    requires MutableSet[other]
        obj.__ixor__(other)
    requires MutableSet[other]
        obj.__ior__(other)
    requires MutableSet[other]
        obj.__isub__(obj)

# 映射：通过键值来获取对象的集合
concept Mapping[obj]:
    requires Container[obj]
    requires Sized[obj]
    requires Iterable[obj]
    associate key
    associate value
    requires Hashable[key]
        obj.__getitem__(key) -> value
    requires ret : bool
        obj.has_key(key) -> ret
    requires Set[ret]
        obj.keys() -> ret
    requires Set[ret]
        obj.values() -> ret
    requires Set[ret]
        obj.items() -> ret

# 可变映射：在 Mapping 的基础上，
# 支持映射的修改操作
concept MutableMapping[obj]:
    requires Mapping[obj]
    associate key = Mapping[obj].key
    associate value = Mapping[obj].value
    obj.__setitem__(key, value)
    obj.__delitem__(key)

# 序列：根据序号访问元素的可迭代有长度容器
concept Sequence[obj]:
    requires Container[obj]
    requires Sized[obj]
    requires Iterable[obj]
    associate item = Iterable[obj].item
    requires Int[index]
        obj.__getitem__(index) -> item

# 可变序列：在 Sequence 的基础上，
# 支持序列的修改操作
concept MutableSequence[obj]:
    requires Sequence[obj]
    associate item = Sequence[obj].item
    requires index : int
        obj.__setitem__(index, item)
    requires index : int
        obj.__delitem__(index)
    obj.append(item)
    obj.remove(item)
    requires Sequence[other]
        obj.extend(other)
    obj.pop() -> item
    requires index : int
        obj.insert(index, item)
    requires MutableSequence[obj]
        obj.reverse() -> ret
```

内建函数的参数

```
# 存在长度的对象：支持内建函数 len()的对象
concept Sized[obj]:
    requires length : int
        obj.__len__() -> length

# Int 对象：支持内建函数 int(obj)
concept Int[obj]:
    requires ret : int
        obj.__int__() -> ret

# String 对象：支持内建函数 str(obj)
concept String[obj]:
    requires ret : str
        obj.__str__() -> ret
```

```
# 函数：支持函数调用运算符()
concept Callable[obj]:
      obj.__call__(*args, ** kwds)

# 可比较对象：支持内建运算符==和!=
concept Comparable[obj]:
    requires ret : bool
        obj.__cmp__() -> ret

# 可加对象：支持内建运算符+和+=
concept Addable[obj]:
    obj.__add__(other)
    obj.__radd__(other)
    obj.__iadd__(other)
```

附录 3 攻读博士学位期间完成的论文列表与参加的科研项目列表

完成论文列表

[1] 周天琳, 史亮, 徐宝文. 重构 C++程序物理设计 (英文). 软件学报. 20(3): 597-607, 2009. **EI 收录**

[2] 周天琳, 徐宝文, 史亮, 周毓明. 基于客户程序度量包内聚性 (英文). 软件学报. 20(2): 256-270, 2009. **EI 收录**

[3] Tianlin Zhou, Liang Shi, Baowen Xu, Yuming Zhou, Lin Chen. Measuring Package Cohesion Based on Context. In Proceedings of the 1st IEEE International Workshop on Semantic Computing and Systems, 2008, pp.127-132. **EI 收录**

[4] 周天琳, 徐宝文, 史亮. 一种扩展的 C++程序重构方法. 全国第五次程序设计语言发展与学术会议, 2006, pp.3-9.

[5] Lin Chen, Baowen Xu, Tianlin Zhou, Xiaoyu Zhou. A Constraint Based Bug Checking Approach for Python. IEEE Computer Software and Applications Conference, 2009. (已录用)

[6] Lin Chen, Baowen Xu, Tianlin Zhou, Yuming Zhou. Applying Generalization Refactoring to Java Generic Programs. In Proceedings of the 1st IEEE International Workshop on Semantic Computing and Systems, 2008, pp.35-39. **EI 收录**

[7] 张婷婷, 周天琳. 一个基于 AOP 的健壮性测试框架. 第五届全国 Web 信息系统及其应用学术会议, 2008.

[8] Lin Chen, Baowen Xu, Tianlin Zhou, Liang Shi, Yanxiang He. Refactoring C++ Programs with Concepts. In Proceedings of the 9th IASTED International Conference on Software Engineering and Applications, 2005.

[9] Xiaoyu Zhou, Tianlin Zhou, Ju Qian, Baowen Xu, Yanxiang He. Identifying Use Case Relations from Codes. In Proceedings of the 9th IASTED International Conference on Software Engineering and Applications, 2005, pp.14-16.

[10] Xu Baowen, Zhou Chaohong, Zhou Tianin, Shi Liang. Aspect-Oriented Programming: Concept, Implementation and Future. Computer and Digital Engineering, 33(8): 1-10, 2005.

[11] Zhou Chaohong, Xu Baowen, Zhou Tianlin, Shi Liang. An Aspect Transformation Approach with Refactoring. In Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering, 2005, pp.318-323.

[12] 余超, 徐宝文, 周天琳. TPython: 一种扩充 Python 语言的设计与实现. 计算机与数字工程. (已录用)

参与项目列表

(1) 国家杰出青年科学基金项目(60425206)，软件工程学

(2) 国家自然科学基金重点项目(60633010)，高可信软件的测试与评估

(3) 国家重点基础研究发展规划 973(2002CB312000)，Internet 环境下基于 Agent 的软件中间件理论和方法研究

(4) 江苏省自然科学基金项目(BK2006094)软件设计模式的形式化描述及程序代码识别技术研究