第 14 章

Software Engineering: A Practitioner's Approach, Eighth Edition

构件级设计

要点浏览

概念：完整的软件构件是在体系结构设计过程中定义的。但是没有在接近代码的抽象级上表示内部数据结构和每个构件的处理细节。构件级设计定义了数据结构、算法、接口特征和分配给每个软件构件的通信机制。

人员：软件工程师完成构件级设计。

重要性：必须能够在构造软件之前就确定该软件是否可以工作。为了保证设计的正确性，以及与早期设计表示（即数据、体系结构和接口设计）的一致性，构件级设计需要以一种可以评审设计细节的方式来表示软件。它提供了一种评估数据结构、接口和算法是否能够工作的方法。

步骤：数据、体系结构和接口的设计表示构成了构件级设计的基础。每个构件的类定义或者处理说明都转化为一种详细设计，该设计采用图形或基于文本的形式来详细说明内部的数据结构、局部接口细节和处理逻辑。设计符号包括 UML 图和一些辅助表示，通过使用一系列结构化编程结构来说明过程的设计。通常的做法是采用现有可复用软件构件，而不是开发新的构件。

工作产品：每个构件的设计都以基于图形的表格或文本方式表示，这是构件级设计阶段产生的主要工作产品。

质量保证措施：采用设计评审机制。对设计执行检查以确定数据结构、接口、处理顺序和逻辑条件等是否都正确，并且给出早期设计中与构件相关的数据或控制变换。

关键概念

内聚性
构件
　适应性
　分类
　组合
　合格性
　WebApp
基于构件的开发
内容设计
耦合
依赖倒置原则
设计重用
设计准则
领域工程

体系结构设计第一次迭代完成之后，就应该开始构件级设计。在这个阶段，全部数据和软件的程序结构都已经建立起来。其目的是把设计模型转化为运行软件。但是现有设计模型的抽象层次相对较高，而可运行程序的抽象层次相对较低。这种转化具有挑战性，因为可能会在软件过程后期引入难以发现和改正的微小错误。Edsgar Dijkstra（帮助我们理解软件设计技术的主要贡献者）在其著作 [Dij72] 中写道：

> 软件似乎不同于很多其他产品，对那些产品而言，一条规则是：更高的质量意味着更高的价格。那些想要真正可靠软件的人发现他们必须找到某种方法来避免开始时的大多数错误，结果，程序设计过程的成本更低……高效率的程序员……不应该将他们的时间浪费在调试上——在开始时就不应该引入错误。

尽管这段话是在很多年前说的，但现在仍然适用。当设计模型被转化为源代码时，必须遵循一系列设计原则，以保证不仅能够完成转化任务，而且不在开始时就引入错误。

14.1 什么是构件

通常来讲，构件是计算机软件中的一个模块化的构造块。再正式一点，OMG 统一建模语言规范 [OMG03a] 是这样定义构件的：系统中模块化的、可部署的和可替换的部件，该部件封装了实现并对外提供一组接口。

关键概念
接口分离原则
Liskov 替换原则
面向对象观点
开闭原则
过程相关
传统构件
传统观点

正如第 13 章的讨论，构件存在于软件体系结构中，因而构件在完成所建系统的需求和目标的过程中起着重要作用。由于构件驻留于软件体系结构的内部，因此它们必须与其他的构件和存在于软件边界以外的实体（如其他系统、设备和人员）进行通信和合作。

对于术语构件的实际意义，软件工程师可能有不同的见解。在接下来的几节中，我们将了解关于“什么是构件以及在设计建模中如何使用构件”的三种重要观点。

引述 细节不仅是细节，它们构成设计。
Charles Eames

14.1.1 面向对象的观点

在面向对象软件工程环境中，构件包括一个协作类集合[⊖]。构件中的每个类都得到详细阐述，包括所有属性和与其实现相关的操作。作为细节设计的一部分，必须定义所有与其他设计类相互通信协作的接口。为此，设计师需要从分析模型开始，详细描述分析类（对于构件而言该类与问题域相关）和基础类（对于构件而言该类为问题域提供了支持性服务）。

关键点 以面向对象的观点来看，构件是协作类的集合。

为了说明设计细化过程，考虑为一个高级影印中心构造软件。软件的目的是收集前台的客户需求，对印刷业务进行定价，然后把印刷任务交给自动生产设备。在需求工程中得到了一个名为 PrintJob 的分析类。分析过程中定义的属性和操作在图 14-1 的上方给出了注释。在体系结构设计中，PrintJob 被定义为软件体系结构的一个构件，用简化的 UML 符号表示的该构件显示在图 14-1 中部靠右的位置[⊖]。需要注意的是，PrintJob 有两个接口：computeJob 和 initiateJob。computeJob 具有对任务进行定价的功能，initiateJob 能够把任务传给生产设备。这两个接口在图下方的左边给出（即所谓的棒棒糖式符号）。

286
~
287

构件级设计将由此开始。必须对 PrintJob 构件的细节进行细化，以提供指导实现的充分信息。通过不断补充作为构件 PrintJob 的类的全部属性和操作，来逐步细化最初的分析类。正如图 14-1 右下部分的描述，细化后的设计类 PrintJob 包含更多的属性信息和构件实现所需要的更广泛的操作描述。computeJob 和 initiateJob 接口隐含着与其他构件（图中没有显示出来）的通信和协作。例如，computePageCost() 操作（computeJob 接口的组成部分）可能与包含任务定价信息的 PricingTable 构件进行协作。checkPriority() 操作（initiateJob 接口的组成部分）可能与 JobQueue 构件进行协作，用来判断当前等待生产的任务类型和优先级。

建议 请回想一下，分析建模和设计建模都是迭代动作。细化原分析类可能需要额外的分析步骤，表示细化设计类的设计建模步骤紧随其后（构件的细节）。

对于体系结构设计组成部分的每个构件都要实施细化。细化一旦完成，要对每个属性、每个操作和每个接口进行更进一步的细化。对适合每个属性的数据结构

⊖ 在某些情况下，构件可以包含一个简单的类。

⊖ 不熟悉 UML 表示法的读者可以参考附录 1。

必须予以详细说明。另外还要说明实现与操作相关的处理逻辑的算法细节，在这一章的后半部分将要对这种过程设计活动进行讨论。最后是实现接口所需机制的设计。对于面向对象软件，还包含对实现系统内部对象间消息通信机制的描述。

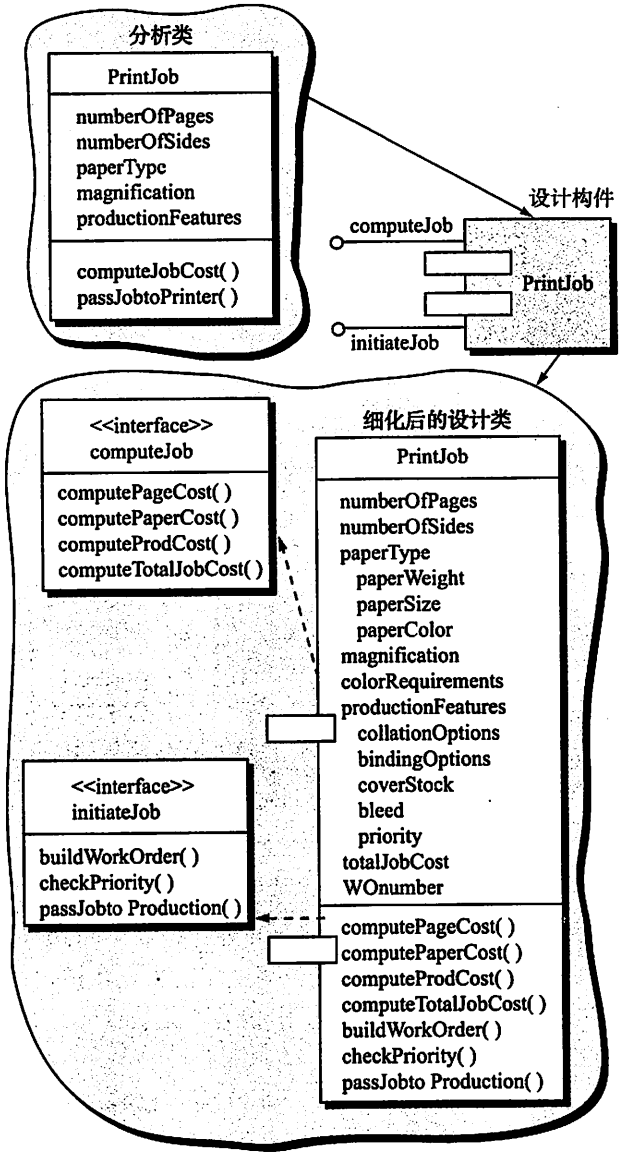


图 14-1　设计构件的细化

14.1.2　传统的观点

在传统软件工程环境中，一个构件就是程序的一个功能要素，程序由处理逻辑及实现处理逻辑所需的内部数据结构以及能够保证构件被调用和实现数据传递的接口构成。传统构件也称为模块，作为软件体系结构的一部分，它扮演如下三个重要角色之一：（1）控制构件，协调问题域中所有其他构件的调用；（2）问题域构件，完成客户需要的全部功能或部分功能；（3）基础设施构件，负责完成问题域中所需的支持处理的功能。

引述　可工作的复杂系统都是由可工作的简单系统演化来的。
John Gall

与面向对象的构件类似，传统的软件构件也来自于分析模型。不同的是在这种情况下，是以分析模型中的构件细化作为导出构件的基础。构件
288 层次结构上的每个构件都被映射为某一层次上的模块（13.6 节）。一般来讲，控制构件（模块）位于层次结构（体系结构）顶层附近，而问题域构件则倾向位于层次结构的底层。为了获得有效的模块化，在构件细化的过程中采用了功能独立性的设计概念（第 12 章）。

为了说明传统构件的细化过程，我们再来考虑为一个高级影印中心构造的软件。一个分层的体系结构的导出如图 14-2 所示。图中每个方框都表示一个软件构件。带阴影的方框在功能上相当于 14.1.1 节讨论的为 PrintJob 类定义的操作。然而，在这种情况下，每个操作都被表示为如图 14-2 所示的能够被调用的单独模块。其他模块用来控制处理过程，也就是前面提到的控制构件。

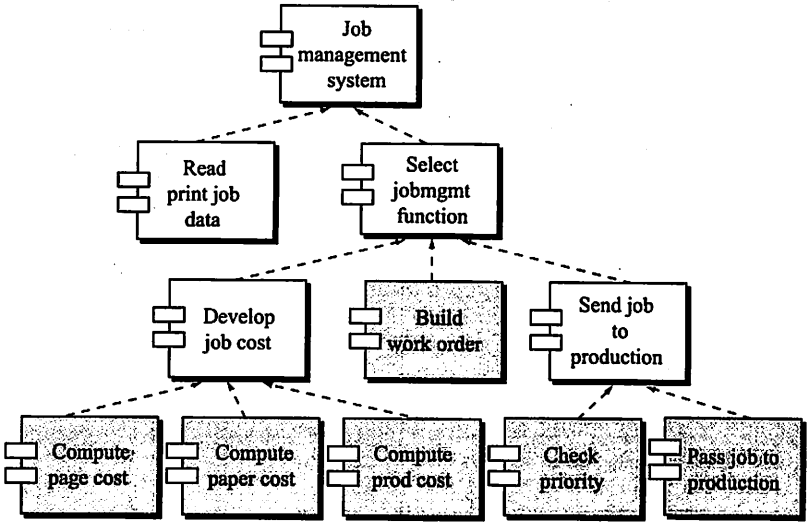


图 14-2　一个传统系统的结构图

在构件级设计中，图 14-2 中的每个模块都要被细化。需要明确定义模块的接口，即每个经过接口的数据或控制对象都需要明确加以说明。还需要定义模块内部使用的数据结构。采用第 12 章讨论的逐步求精方法设计完成模块中相关功能的算法。有时候需要用状态图表示模块行为。

建议　随着每个软件构件设计的逐步细化，设计焦点转向特定数据结构的设计和操作数据结构的过程设计。但是，不要忘了体系结构，它必须容纳构件或服务于多数构件的全局数据结构。

289 为了说明这个过程，考虑 ComputePageCost 模块。该模块的目的在于根据用户提供的规格说明来计算每页的印刷成本。为了实现该功能需要以下数据：文档的页数、文档的印刷份数、单面或者双面印刷、颜色、纸张大小。这些数据通过该模块的接口传递给 ComputePageCost。ComputePageCost 根据任务量和复杂度，使用这些数据来决定一页的成本——这是一个通过接口将所有数据传递给模块的功能。每一页的成本与任务量成反比，与任务的复杂度成正比。

图 14-3 给出了使用改进的 UML 建模符号描述的构件级设计。其中 ComputePageCost 模块通过调用 getJobData 模块（它允许所有相关数据都传递给该构件）和数据库接口 accessCostDB（它能够使该模块访问存放所

有印刷成本的数据库）来访问数据。接着，对 ComputePageCost 模块进一步细化，给出算法和接口的细节描述（图 14-3）。其中，算法的细节可以由图中显示的伪代码或者 UML 活动图来表示。接口被表示为一组输入和输出的数据对象或者数据项的集合。设计细化的过程一直进行下去，直到能够提供指导构件构造的足够细节为止。

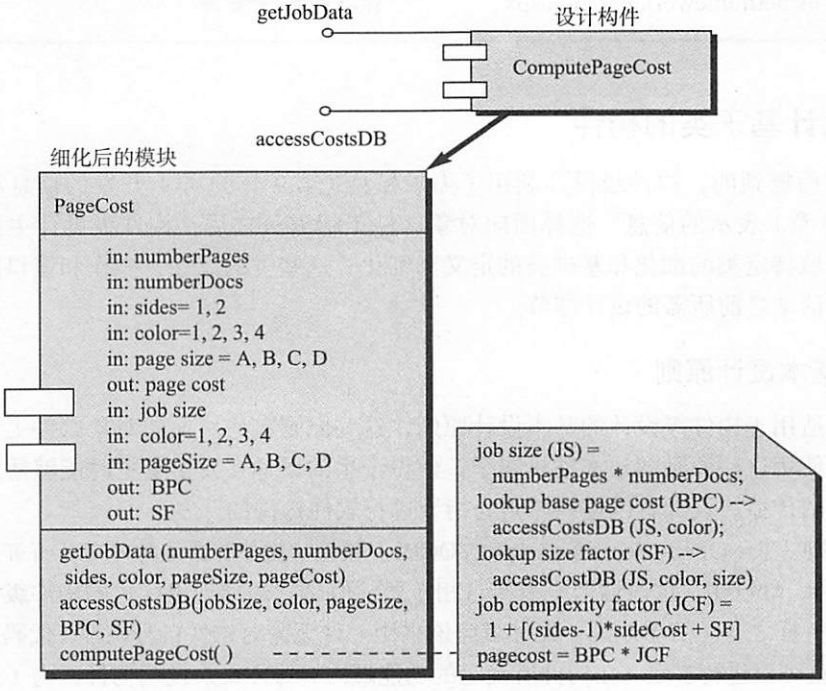


图 14-3　ComputePageCost 的构件级设计

14.1.3　过程相关的观点

290

14.1.1 节和 14.1.2 节提到的关于构件级设计的面向对象观点和传统观点，都假定从头开始设计构件。也就是说，设计者必须根据从需求模型中导出的规格说明创建新构件。当然，还存在另外一种方法。

在过去的 30 年间，软件工程已经开始强调使用已有构件或设计模式来构造系统的必要性。实际上，软件工程师在设计过程中可以使用已经过验证的设计或代码级构件目录。当软件体系结构设计完后，就可以从目录中选出构件或者设计模式，并用于组装体系结构。由于这些构件是根据复用思想来创建的，所以其接口的完整描述、要实现的功能和需要的通信与协作等对于设计者来说都是可以得到的。在 14.6 节将讨论基于构件的软件工程（Component-Based Software Engineering，CBSE）的某些重要方面。

信息栏　基于构件的标准和框架

基于构件的软件工程（CBSE）成功或者失败的重要因素之一就是基于构件标准（有时称为中间件）的可用性。中间件是一组基础构件的集合，这些基础构件使得问题域构件与其他构件通过网络进行通信，或者在一个复杂的系统中通信。对于想用基于构件的软件工程方法进行开发的软件工程师来讲，他们可以使用如下的标准：

- OMG CORBA——http://www.corba.org/。
- Microsoft COM——http://www.microsoft.com/com/default.mspx。
- Microsoft .NET——http://msdn.microsoft.com/en-us/netframework/default.aspx。
- Sun JavaBeans——http://www.oracle.com/technetwork/java/javaee/ejb/index.html。

上面提到的网站上提供了大量的教材、白皮书、工具和关于这些重要中间件标准的大量资源。

14.2　设计基于类的构件

正如前面提到的，构件级设计利用了需求模型（第 9 ～ 11 章）开发的信息和体系结构模型（第 13 章）表示的信息。选择面向对象软件工程方法之后，构件级设计主要关注需求模型中问题域特定类的细化和基础类的定义和细化。这些类的属性、操作和接口的详细描述是开始构造活动之前所需的设计细节。

291

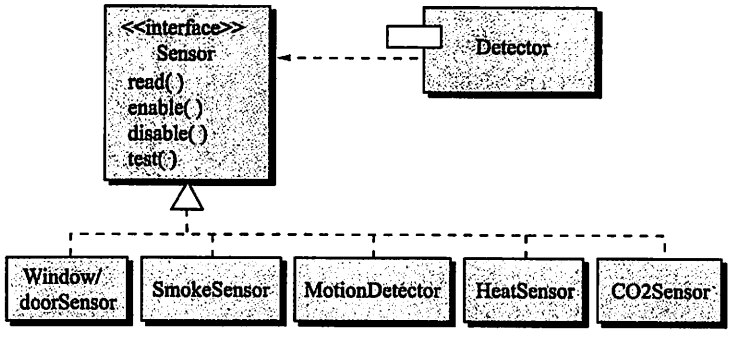
14.2.1　基本设计原则

有四种适用于构件级设计的基本设计原则，这些原则在使用面向对象软件工程方法时被广泛采用。使用这些原则的根本动机在于，使得产生的设计在发生变更时能够适应变更并且减少副作用的传播。设计者以这些原则为指导进行软件构件的开发。

开闭原则（The Open-Closed Principle，OCP）。*模块（构件）应该对外延具有开放性，对修改具有封闭性* [Mar00]。这段话似乎有些自相矛盾，但是它却体现出优秀的构件级设计应该具有的最重要特征之一。简单地说，设计者应该采用一种无需对构件自身内部（代码或者内部逻辑）做修改就可以进行扩展（在构件所确定的功能域内）的方式来说明构件。为了达到这个目的，设计者需要进行抽象，在那些可能需要扩展的功能与设计类本身之间起到缓冲区的作用。

例如，假设 SafeHome 的安全功能使用了对各种类型安全传感器进行状态检查的 Detector 类。随着时间的推移，安全传感器的类型和数量将会不断增长。如果内部处理逻辑采用一系列 if-then-else 的结构来实现，其中每个这样的结构都负责一个不同的传感器类型，那么对于新增加的传感器类型，就需要增加额外的内部处理逻辑（依然是另外的 if-then-else 结构），而这显然违背 OCP 原则。

图 14-4 中给出了一种遵循 OCP 原则实现 Detector 类的方法。对于各种不同的传感器，Sensor 接口都向 Detector 构件呈现一致的视图。如果要添加新类型的传感器，那么对 Detector 类（构件）无需进行任何改变。这个设计遵守了 OCP 原则。



292

图 14-4　遵循 OCP 原则

SafeHome　OCP 的应用

[场景] Vinod 的工作间。

[人物] Vinod 和 Shakira，SafeHome 软件工程团队成员。

[对话]

Vinod：我刚刚接到 Doug（团队经理）的一个电话，他说市场营销人员想增加一个新的传感器。

Shakira（假笑）：哎呀，别再加了。

Vinod：是啊……你永远不会相信这些家伙都提出了什么。

Shakira：确实令我很吃惊。

Vinod（大笑）：他们称之为小狗焦虑传感器（doggie angst sensor）。

Shakira：那是什么装置？

Vinod：这个装置是为了那些想把宠物留在彼此相邻很近的公寓、门廊或房子里的人设计的。狗叫致使邻里生气和抱怨，有了这种传感器，如果狗的叫声超过一定时间（比如一分钟），传感器就会向主人的手机发送特殊的报警信号。

Shakira：你在开玩笑吗？

Vinod：不是，Doug 想知道在安全功能中加入这个功能需要多长时间。

Shakira（想了想）：不用多长时间……瞧（她给 Vinod 看图 14-4），我们分离出在 sensor 接口背后的实际的传感器类。只要我们有小狗传感器的规格说明，那么把它加入其中就是一件简单的事情了。我们要做的就是为其创建一个合适的构件……哦，类。根本不用改变 Detector 构件。

Vinod：好的，我将告诉 Doug 这不是什么大问题。

Shakira：告诉 Doug，直到下一个版本发布之前，我们都要集中精力完成小狗焦虑传感器的事情。

Vinod：这不是件坏事，而如果他想让你做，你可以马上实现吗？

Shakira：是啊，我们的接口设计使得我可以毫无困难地完成它。

Vinod（想了想）：你听说过开闭原则吗？

Shakira（耸了耸肩膀）：没有。

Vinod（微笑）：不成问题。

Liskov **替换原则**（Liskov Substitution Principle，LSP）。子类可以替换它们的基类[Mar00]。最早提出该设计原则的 Barbara Liskov[Lis88] 建议，将从基类导出的类传递给构件时，使用基类的构件应该仍然能够正确完成其功能。LSP 原则要求源自基类的任何子类必须遵守基类与使用该基类的构件之间的隐含约定。在这里的讨论中，“约定”既是前置条件——构件使用基类前必须为真；又是后置条件——构件使用基类后必须为真。当设计者创建了导出类，则这些子类必须遵守前置条件和后置条件。

依赖倒置原则（Dependency Inversion Principle，DIP）。依赖于抽象，而非具体实现 [MarR00]。正如我们在 OCP 中讨论的那样，抽象可以比较容易地对设计进行扩展，又不会导致大量的混乱。构件依赖的其他具体构件（不是依赖抽象类，如接口）越多，扩展起来就越困难。

建议　如果你省去设计并且破解出代码，记住代码只是最终的“具体实现”，你违背了 DIP 原则。

293

接口分离原则（Interface Segregation Principle，ISP）。多个客户专用接口比一个通用接口要好 [Mar00]。多个客户构件使用一个服务器类提供的操作的实例有很多。ISP 原则建议设计者应该为每个主要的客户类型都设计一个特定的接口。只有那些与特定客户类型相关的操作才应该出现在该客户的接口说明中。如果多个客户要求相同的操作，则这些操作应该在每个特定的接口中都加以说明。

例如，假设 FloorPlan 类用在 SafeHome 的安全和监视功能中（第 10 章）。对于安全功

能，FloorPlan 只在配置活动中使用，并且使用 placeDevice()、showDevice()、groupDevice()、remove-Device() 等操作实现在建筑平面图中放置、显示、分组和删除传感器。SafeHome 监视功能除了需要这四个有关安全的操作之外，还需要特殊的操作 showFOV()、showDeviceID() 来管理摄像机。因此，ISP 建议为来自 SafeHome 功能的两个客户端构件定义特殊的接口。安全接口应该只包括 palceDevice()、showDevice()、groupDevice()、removeDevice() 四种操作。监视接口应该包括 placeDevice()、showDevice()、groupDevice()、removeDevice()、showFOV()、showDeviceID() 六种操作。

关键点 设计可复用的构件不仅需要优秀的技术设计，而且需要高效的配置控制机制（第 29 章）。

尽管构件级设计原则提供了有益的指导，但构件自身不能够独立存在。在很多情况下，单独的构件或者类被组织到子系统或包中。于是我们很自然地就会问这个包会有怎样的活动。在设计过程中如何正确组织这些构件？ Martin 在 [Mar00] 中给出了在构件级设计中可以应用的另外一些打包原则，这些原则如下。

发布复用等价性原则（Release Reuse Equivalency Principle，REP）。*复用的粒度就是发布的粒度* [Mar00]。当设计类或构件用以复用时，在可复用实体的开发者和使用者之间就建立了一种隐含的约定关系。开发者承诺建立一个发布控制系统，用来支持和维护实体的各种老版本，同时用户逐渐地将其升级到最新版本。明智的方法是将可复用的类分组打包成能够管理和控制的包并作为一个更新的版本，而不是对每个类分别进行升级。

共同封装原则（Common Closure Principle，CCP）。*一同变更的类应该合在一起* [Mar00]。类应该根据其内聚性进行打包。也就是说，当类被打包成设计的一部分时，它们应该处理相同的功能或者行为域。当某个域的一些特征必须变更时，只有相应包中的类才有可能需要修
294 改。这样可以进行更加有效的变更控制和发布管理。

共同复用原则（Common Reuse Principle，CRP）。*不能一起复用的类不能被分到一组* [Mar00]。当包中的一个或者多个类变更时，包的发布版本号也会发生变更。所有那些依赖于已经发生变更的包的类或者包，都必须升级到最新版本，并且都需要进行测试以保证新发布的版本能够无故障运转。如果类没有根据内聚性进行分组，那么这个包中与其他类无关联的类有可能会发生变更，而这往往会导致进行没有必要的集成和测试。因此，只有那些一起被复用的类才应该包含在一个包中。

14.2.2 构件级设计指导方针

除了 14.2.1 节中讨论的原则之外，在构件级设计的进程中还可以使用一系列实用的设计指导方针。这些指导方针可以应用于构件、构件的接口，以及对于最终设计有着重要影响的依赖和继承特征等方面。Ambler[Amb02b] 给出了如下的指导方针。

构件。对那些已经被确定为体系结构模型一部分的构件应该建立命名约定，并对其做进一步的精细化处理，使其成为构件级模型的一部分。体系结构构件的名字来源于问题域，并且对于考虑体系结构模型的所有利益相关者来说都是有意义的。例如，无论技术背景如何，FloorPlan 这个类的名称对于任何读到它的人来说都是有意义的。另一方面，基础构件或者细化后的构件级类应该以能够反映其实现意义的名称来命名。例如，对一个作为 FloorPlan 实现一部分的链表进

提问 命名构件时需要考虑些什么？

行管理时，操作 manageList() 是一个合适的名称，因为即使是非技术人员也不会误解。⊖

在详细设计层面使用构造型帮助识别构件的特性也很有价值。例如，<<infrastructure>> 可以用来标识基础设施构件；<<database>> 可以用来标识服务于一个或多个设计类或者整个系统的数据库；<<table>> 可以用来标识数据库中的表。

接口。接口提供关于通信和协作的重要信息（也可以帮助我们实现 OCP 原则）。然而，接口表示的随意性会使构件图趋于复杂化。Ambler[Amb02c] 建议：（1）当构件图变得复杂时，在较正式的 UML 框和虚箭头记号方法中使用接口的棒棒糖式记号⊜；（2）为了保持一致，接口都放在构件框的左边；（3）即使其他的接口也适用，也只表示出那些与构件相关的接口。这些建议意在简化 UML 构件图，使其易于查看。 295

依赖与继承。为了提高可读性，依赖关系自左向右，继承关系自底（导出类）向上（基类）。另外，构件之间的依赖关系通过接口来表示，而不是采用“构件到构件”的方法来表示。遵照 OCP 的思想，这种方法使得系统更易于维护。

14.2.3　内聚性

在第 12 章中，我们将内聚性描述为构件的“专一性”。在为面向对象系统进行构件级设计时，内聚性意味着构件或者类只封装那些相互关联密切，以及与构件或类自身有密切关系的属性和操作。Lethbridge 和 Laganiére[Let01] 定义了许多不同类型的内聚性（按照内聚性的级别排序⊜）。

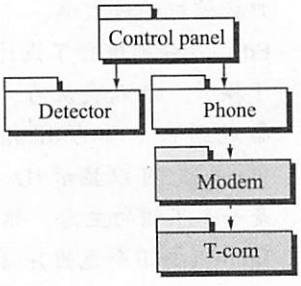


图 14-5　分层内聚

功能内聚。主要通过操作来体现，当一个模块完成一组且只有一组操作并返回结果时，就称此模块是功能内聚的。

分层内聚。由包、构件和类来体现。高层能够访问低层的服务，但低层不能访问高层的服务。例如，如果警报响起，SafeHome 的安全功能需要打出一个电话。可以定义如图 14-5 所示的一组分层包，带阴影的包中包含基础设施构件。访问都是从 Control panel 包向下进行的。

通信内聚。访问相同数据的所有操作被定义在一个类中。一般说来，这些类只着眼于数据的查询、访问和存储。

那些体现出功能、层和通信等内聚性的类和构件，相对来说易于实现、测试和维护。设计者应该尽可能获得这些级别的内聚性。然而，需要强调的是，实际的设计和实现问题有时会迫使设计者选择低级别的内聚性。

建议　尽管理解各种级别的内聚性很有益，但是更为重要的是在设计构件时对内聚性有全面的理解。尽可能保持高内聚性。 296

SafeHome　内聚性的应用

[场景] Jamie 的工作间。

[人物] Jamie 和 Ed，SafeHome 软件工程团队成员，正在实现监视功能。

[对话]

Ed：我已经完成了 camera（摄像机）构件的初步设计。

⊖ 由销售部和顾客部（非技术类型）人员检查详细的设计信息是靠不住的。

⊜ 这里指的是类似图 14-1 中的接口表示。——译者注

⊜ 一般来说，内聚性级别越高，构件实现、测试和维护起来就越容易。

Jamie：希望快速评审一下吗？

Ed：我想……但实际上，你最好输入一些信息。（Jamie 示意他继续。）

Ed：我们起初为 camera 构件定义了 5 种操作。看……

determineType() 给出用户摄像机的类型。

translateLocation() 允许用户删除设计图上的摄像机。

displayID() 得到摄像机 ID，并将其显示在摄像机图标附近。

displayView() 以图形化方式给出用户摄像机的视角范围。

displayZoom() 以图形化方式给出用户摄像机的放大率。

Ed：我分别进行了设计，并且它们非常易于操作。所以我认为，将所有的显示操作集成到一个称为 displayCamera() 的操作中——它可以显示 ID、视角和放大率等，是一个不错的主意。你不这样认为吗？

Jamie（扮了个鬼脸）：我不敢肯定。

Ed（皱眉）：为什么？所有这些小操作是很令人头疼的！

297 **Jamie**：问题是当将它们集成到一起后，我们将失去内聚性。你知道的，displayCamera() 操作不是专一的。

Ed（略有不快）：那又怎样？这样做使得我们最多只需不到 100 行的源代码。我想实现起来也比较简单。

Jamie：如果销售人员决定更改我们显示视角域的方式怎么办？

Ed：我马上跳到 displayCamera() 操作，并进行这种改变。

Jamie：那么引起的副作用怎么办？

Ed：什么意思？

Jamie：也就是说你做了修改，但是不经意之间产生了 ID 的显示问题。

Ed：我没有那么笨。

Jamie：可能没有，但是如果两年以后某些支持人员必须做这种改变怎么办。他可能并不像你一样理解这个操作，谁知道呢，也许他很粗心。

Ed：所以你反对？

Jamie：你是设计师，这是你的决定，只要确信你理解了低内聚性的后果。

Ed（想了想）：可能我们要设计一个单独的显示操作。

Jamie：好主意。

14.2.4 耦合性

建议 随着每个软件构件的细化，我们的关注点将转移到数据结构设计上，包括其相关的过程设计。但是，不要忘记体系结构，因为构件和服务于构件的全局数据结构都住在体系结构这所大房子里。

在前面关于分析和设计的讨论中，我们知道通信和协作是面向对象系统中的基本要素。然而，这个重要（必要）特征存在一个黑暗面。随着通信和协作数量的增长（也就是说，随着类之间的联系程度越来越强），系统的复杂性也随之增长了。同时，随着系统复杂度的增长，软件实现、测试和维护的困难也随之增大。

耦合是类之间彼此联系程度的一种定性度量。随着类（构件）之间的相互依赖越来越多，类之间的耦合程度亦会增加。在构件级设计中，一个重要的目标就是尽可能保持低耦合。

有多种方法来表示类之间的耦合。Lethbridge 和 Laganiére[Let01] 定义了一组耦合分类。例如，内容耦合发生在当一个构件“暗中修改其他构件的内部数据”[Let01] 时。这违反了基本设计概念当中的信息隐蔽原则。控制耦合发生在当操作 A 调用操作 B，并且向 B 传递了一个控制标记时。接着，控制标记将会指引 B 中的逻辑流程。这种耦合形式的主要问题在于，B 中的一个不相关变更往往能够导

致 A 所传递控制标记的意义也必须发生变更。如果忽略这个问题，就会引起错误。外部耦合发生在当一个构件和基础设施构件（例如，操作系统功能、数据库容量、无线通信功能等）进行通信和协作时。尽管这种类型的耦合是必要的，但是在一个系统中应该尽量将这种耦合限制在少量的构件或者类范围内。

软件必须进行内部和外部的通信，因此，耦合是必然存在的。然而，在不可避免出现耦合的情况下，设计者应该尽力降低耦合性，并且要充分理解高耦合的后果。

SafeHome　耦合的应用

[场景] Shakira 的工作间。

[人物] Vinod 和 Shakira，SafeHome 软件工程团队成员，正在实现安全功能。

[对话]

Shakira：我曾经有一个非常好的想法，但之后我又考虑了一下，觉得好像并没有那么好。最后我还是放弃了，不过我想最好和你讨论一下。

Vinod：当然可以，是什么想法？

Shakira：好的，每个传感器能够识别一种警报条件，对吗？

Vinod（微笑）：这就是我们称它为传感器的一个原因啊，Shakira。

Shakira（恼怒）：你讽刺我！你应该好好学习一下处理人际关系的技巧。

Vinod：你刚才说什么？

Shakira：我指的是……为什么不为每个传感器都创建一个名为 makeCall() 的操作，该操作能够直接和 OutgoingCall（外呼）构件协作，也就是通过 OutgoingCall 构件的接口实现协作？

Vinod（沉思着）：你的意思是让协作发生在 ControlPanel 之类的构件之外？

Shakira：是的，但接着我又对自己说，这将意味着每个传感器对象都会与 OutgoingCall 构件相关联，而这意味着与外部世界的间接耦合……我想这样会使事情变得复杂。

Vinod：我同意，在这种情况下，让传感器接口将信息传递给 ControlPanel，并且让其启动外呼，这是一个比较好的主意。此外，不同的传感器将导致不同的电话号码。在信息改变时，你并不希望传感器存储这些信息，因为如果发生变化……

Shakira：感觉不太对。

Vinod：耦合设计方法告诉我们是不太对。

Shakira：无论如何……

298

14.3　实施构件级设计

引述　如果我有更多的时间，我将写一封更短的信。

Blaise Pascal

在本章的前半部分，我们已经知道构件级设计本质上是精细化的。设计者必须将需求模型和架构模型中的信息转化为一种设计表示，这种表示提供了用来指导构件（编码和测试）活动的充分信息。应用于面向对象系统时，下面的步骤表示出构件级设计典型的任务集。

步骤 1：标识出所有与问题域相对应的设计类。使用需求模型和架构模型，正如 14.1.1 节中所描述的那样，每个分析类和体系结构构件都要细化。

步骤 2：确定所有与基础设施域相对应的设计类。在需求模型中并没有描述这些类，并且在体系结构设计中也经常忽略这些类，但是此时必须对它们进行描述。如前所述，这种类型的类和构件包括 GUI（图形用户界面）构件（通常为可复用构件）、操作系统构件以及对象

和数据管理构件等。

步骤 3：细化所有不需要作为复用构件的设计类。详细描述实现类细化所需要的所有接口、属性和操作。在实现这个任务时，必须考虑采用设计的启发式规则（如构件的内聚和耦合）。

建议　如果在非面向对象的环境下工作，那么前3步的重点在于提炼数据对象和处理功能（转换），这些功能被视为需求模型的一部分。

步骤 3a：在类或构件协作时说明消息的细节。需求模型中用协作图来显示分析类之间的相互协作。在构件级设计过程中，某些情况下有必要通
299 过对系统中对象间传递消息的结构进行说明来表现协作细节。尽管这是一个可选的设计活动，但是它可以作为接口规格说明的前提，这些接口显示了系统构件之间通信和协作的方式。

图 14-6 给出了前面提到的影印系统的一个简单协作图。ProductionJob、WorkOrder 和 JobQueue 这三个对象相互协作，为生产线准备印刷作业。图中的箭头表示对象间传递的消息。在需求建模时，消息说明如图 14-6 所示。然而，随着设计的进行，消息通过下列方式的扩展语法来细化 [Ben02]：

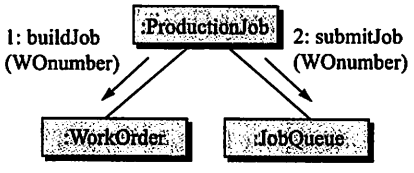


图 14-6　带消息的协作图

```
[guard condition] sequence expression (return value) :=
message name (argument list)
```

其中，[guard condition] 采用对象约束语言（Object Constraint Language，OCL）⊖来书写，并且说明了在消息发出之前应该满足什么样的条件集合；sequence expression 是一个表明消息发送序号的整数（或其他样式的表明发送顺序的指示符，如 3.1.2）；(return value) 是由消息唤醒操作返回的信息名；message name 表示唤醒的操作，(argument list) 是传递给操作的属性列表。

步骤 3b：为每个构件确定适当的接口。在构件级设计中，一个 UML 接口是“一组外部可见的（即公共的）操作。接口不包括内部结构，没有属性，没有关联……”[Ben02]。更正式地讲，接口就是某个抽象类的等价物，该抽象类提供了设计类之间的可控连接。图 14-1
300 给出了接口细化的实例。实际上，为设计类定义的操作可以归结为一个或者多个抽象类。抽象类内的每个操作（接口）应该是内聚的，也就是说，它应该展示那些关注于一个有限功能或者子功能的处理。

参照图 14-1，initiateJob 接口由于没有展现出足够的内聚性而受到争议。实际上，它完成了三个不同的子功能：建立工作单，检查任务的优先级，并将任务传递给生产线。接口设计应该重构。一种方法就是重新检查设计类，并定义一个新类 WorkOrder，该类的作用就是处理与装配工作单相关的所有活动。操作 buildWorkOrder() 成为该类的一部分。类似地，我们可能需要定义包括操作 checkPriority() 在内的 JobQueue 类。ProductionJob 类包括给生产线传递生产任务的所有相关信息。initiateJob 接口将采用图 14-7 所示的形式。initiateJob 现在是内聚的，集中在一个功能上。与 ProductionJob、WorkOrder 和 JobQueue 相关的接口几乎都是专一的。

步骤 3c：细化属性，并且定义实现属性所需要的数据类型和数据结构。描述属性的数据类型和数据结构一般都需要在实现时所采用的程序设计语言中进行定义。UML 采用下面

⊖ OCL 在附录 1 中有简明的介绍。

的语法来定义属性的数据类型：

```
name:type-expression = initial-value{property string}
```

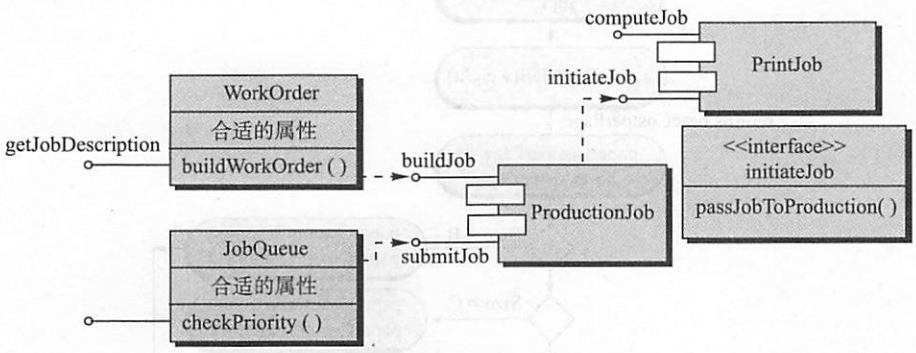


图 14-7　为 PrintJob 重构接口和类定义

其中，name 是属性名，type-expression 是数据类型，initial-value 是创建对象时属性的取值，property string 用于定义属性的特征或特性。

301

在构件级设计的第一轮迭代中，属性通常用名字来描述。再次参考图 14-1，PrintJob 的属性列表只列出了属性名。然而，随着设计的进一步细化，我们将使用 UML 的属性格式注释来定义每个属性。例如，以下列方式来定义 paperType-weight：

```
paperType-weight: string = "A" {contains 1 of 4 values-A, B, C, or D}
```

这里将 paperType-weight 定义为一个字符串变量，初始值为 A，可以取值为集合 {A, B, C, D} 中的一个。

如果某一属性在多个设计类中重复出现，并且其自身具有比较复杂的结构，那么最好是为这个属性创建一个单独的类。

步骤 3d：详细描述每个操作中的处理流。这可能需要由基于程序设计语言的伪代码或者由 UML 活动图来完成。每个软件构件都需要应用逐步求精概念（第 12 章）通过很多次迭代进行细化。

第一轮迭代中，将每个操作都定义为设计类的一部分。在任何情况下，操作应该确保具有高内聚性的特征；也就是说，一个操作应该完成单一的目标功能或者子功能。接下去的一轮迭代，只是完成对操作名的详细扩展。例如，图 14-1 中的操作 computePaperCost() 可以采用如下方式进行扩展：

```
computePaperCost(weight, size, color): numeric
```

这种方式说明 computPageCost() 要求属性 weight、size 和 color 作为输入，并返回一个数值（实际上为金额）作为输出。

如果实现 computePaperCost() 的算法简单而且易于理解，则没有必要开展进一步的设计细化。软件编码人员将会提供实现这些操作的必要细节。但是，如果算法比较复杂或者难于理解，此时则需要进一步的设计细化。图 14-8 给出了操作 computePaperCost() 的 UML 活动图。当活动图用于构件级设计的规格说明时，通常都在比源码更高的抽象级上表示。还有一种方法是，在设计规格说明中使用伪代码，这部分内容将在 14.5.3 节进行讨论。

建议　在细化构件设计时使用逐步求精的方法。经常提出这样的问题："是否存在一种方法可以简化问题并仍能达到相同的结果？"

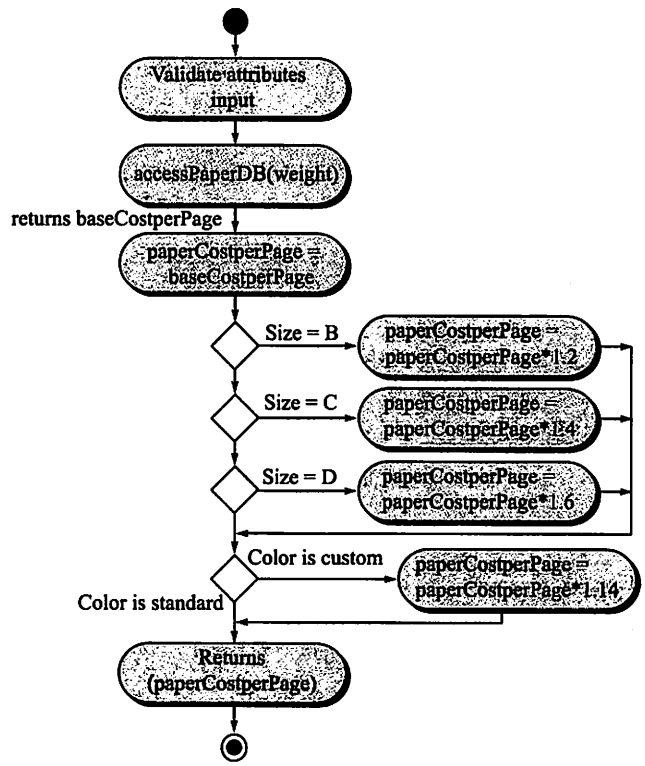


图 14-8　computePaperCost() 操作的 UML 活动图

步骤 4：说明持久数据源（数据库和文件）并确定管理数据源所需要的类。数据库和文件通常都凌驾于单独的构件设计描述之上。在多数情况下，这些持久数据存储最初都作为体系结构设计的一部分进行说明，然而，随着设计细化过程的不断深入，提供关于这些持久数据源的结构和组织等额外细节常常是有用的。

步骤 5：开发并且细化类或构件的行为表示。UML 状态图被用作需求模型的一部分，表示系统的外部可观察的行为和更多的分析类个体的局部行为。在构件级设计过程中，有些时候对设计类的行为进行建模是必要的。

对象（程序执行时的设计类实例）的动态行为受到外部事件和对象当前状态（行为方式）的影响。为了理解对象的动态行为，设计者必须检查设计类生命周期中所有相关的用例，这些用例提供的信息可以帮助设计者描述影响对象的事件，以及随着时间流逝和事件的发生对象所处的状态。图 14-9 描述了使用 UML 状态图 [Ben02] 表示的状态之间的转换（由事件驱动）。

302
~
303

从一种状态到另一种状态的转换（用圆角矩形来表示），都表示为如下形式的事件序列：

```
event-name(parameter-list)[guard-condition]/action expression
```

其中，event-name 表示事件；parameter-list 包含了与事件相关的数据；guard-condition 采用对象约束语言 OCL 书写，并描述了事件发生前必须满足的条件；action expression 定义了状态转换时发生的动作。

参照图 14-9，针对状态的进入和离开两种情形，每个状态都可以定义 entry/ 和 exit/ 两个动作。在大多数情况下，这些动作与正在建模的类的相关操作相对应。do/ 指示符提供了

一种机制，用来显示伴随此种状态的相关活动；而 include/ 指示符则提供了通过在状态定义中嵌入更多状态图细节的方式进行细化的手段。

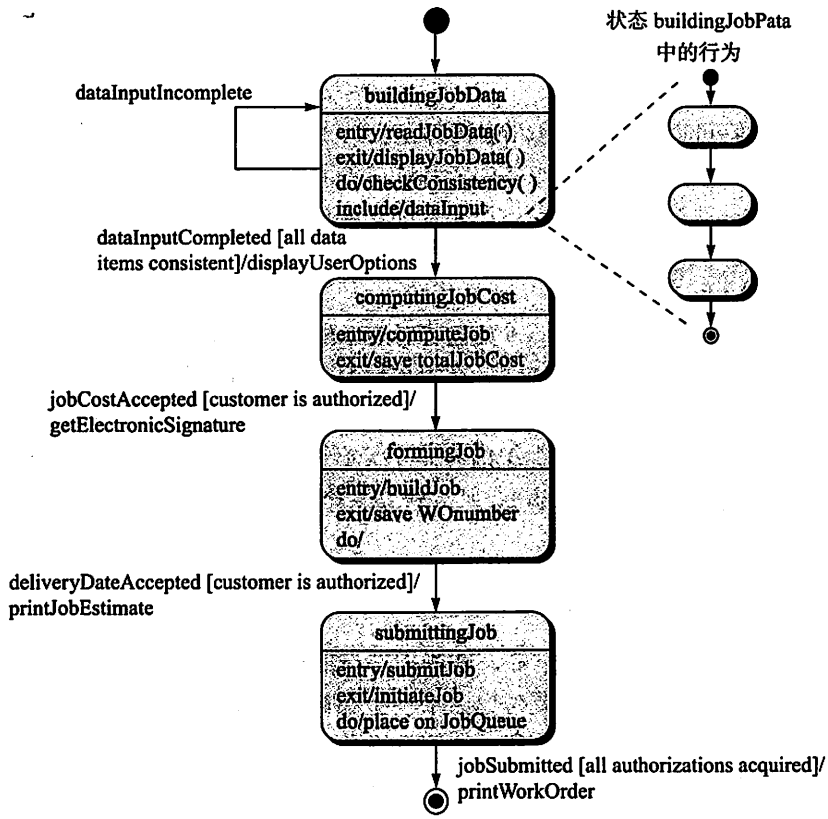


图 14-9　PrintJob 类的状态图

需要注意的重要一点是，行为模型经常包含一些在其他设计模型中不明显的信息。例如，通过仔细查看图 14-9 中的状态图可以知道，当得出印刷任务的成本和进度数据时，PrintJob 类的动态行为取决于用户对此是否认可。如果没有得到允许（警戒条件确保用户有权审核），印刷工作就不能提交，因为不可能到达 submittingJob 状态。 304

步骤 6：细化部署图以提供额外的实现细节。部署图（第 12 章）作为体系结构设计的一部分，采用描述符形式来表示。在这种表示形式中，主要的系统功能（经常表现为子系统）都表示在容纳这些功能的计算环境中。

在构件级设计过程中，应该对部署图进行细化，以表示主要构件包的位置。然而，一般在构件图中不单独表示构件，目的在于避免图的复杂性。某些情况下，部署图在这个时候被细化成实例形式。这意味着要对指定的硬件和使用的操作系统环境加以说明，而构件包在这个环境中的位置也需要确定。

步骤 7：考虑每个构件级设计表示，并且时刻考虑其他可选方案。纵观全书，我们始终强调设计是一个迭代过程。创建的第一个构件级模型总没有迭代多次之后得到的模型那么全面、一致或精确。在进行设计工作时，重构是十分必要的。

另外，设计者不能眼光狭隘。设计中经常存在其他的设计方案，在没有决定最终设计模型之前，最好的设计师会考虑所有（或大部分）的方案，运用第 12 章和本章介绍的设计原

则和概念开发其他的可选方案，并且仔细考虑和分析这些方案。

14.4　WebApp 的构件级设计

在基于 Web 的系统和应用中，内容和功能的界限常常是模糊的。因此有必要考虑什么是 WebApp 构件。

根据本章的内容可以知道，WebApp 构件是：（1）定义良好的聚合功能，为最终用户处理内容，或提供计算或数据处理；（2）内容和功能的聚合包，提供最终用户所需的功能。因
305 此 WebApp 构件级设计通常包括内容设计元素和功能设计元素。

14.4.1　构件级内容设计

构件级内容设计关注内容对象，以及包装后展示给 WebApp 最终用户的方式。构件级内容设计应该适合创建的 WebApp 的特性。在很多情况下，内容对象不需要被组织成构件，它们可以分别实现。但是，随着 WebApp、内容对象以及它们之间相互关系的规模和复杂度的增长，在更好的参考和设计方法下组织内容是十分必要的[⊖]。此外，如果内容显示出高度的动态性（如一个在线拍卖网站的内容），那么建立一个包含内容构件的、清晰的结构模型是非常重要的。

14.4.2　构件级功能设计

WebApp 是作为一系列构件交付的，这些构件与信息体系结构并行开发，以确保它们的一致性。最重要的是，在一开始就要考虑需求模型和初始信息体系结构，然后再进一步考查功能如何影响用户与系统的交互、要展示的信息以及要管理的用户任务。

在体系结构设计中，往往将 WebApp 的内容和功能结合在一起来设计应用系统的*功能体系结构*。在这里，功能体系结构代表的是 WebApp 的功能域，并且描述了关键的功能构件和这些构件是如何进行交互的。

14.5　移动 App 的构件级设计

在第 13 章中我们指出，移动 App 通常使用多层结构，包括用户界面层、业务层和数据层。如果你正在为一个基于 Web 的瘦客户端构建移动 App，那么只有那些实现用户界面所需的构件才会驻留在移动设备上。在移动设备上，一些应用可能包含用于实现业务层和数据层需求的构件，这些层会受到设备物理特性的限制。

首先考虑用户界面层，重要的一点是，较小的显示区域要求设计者必须选择要显示的内容（文本和图形）。这样可能有助于为特定的用户组定制不同内容，且仅显示每组所需要的
306 内容。业务层和数据层通常是由 Web 或云服务构件来实现的。如果提供业务和数据服务的构件完全驻留在移动设备上，那么连接问题并不是需要重点关注的问题。设计时必须考虑在网络连接间断（或丢失）时，构件如何访问当前应用驻留在网络服务器上的数据。

将桌面应用移植到移动设备上时，需要对业务层构件进行检查，看它们是否满足新平台所需的非功能性需求（例如，安全性、性能、可访问性）。目标移动设备可能缺乏必要的处理器速度、内存或显示性能。在第 18 章将更详细地介绍移动 App 的设计。

⊖ 内容构件可以在其他 WebApp 中复用。

14.6　设计传统构件

传统软件构件[⊖]的构件级设计基础在 20 世纪 60 年代早期已经形成，在 Edsgar Dijkstra（[Dij65]，[Dij76b]）及他人的著作（如 [Boh66]）中又得到了进一步的完善。20 世纪 60 年代末，Dijkstra 等人提出，所有的程序都可以建立在一组限定好的逻辑构造上。这组逻辑构造强调“对功能域的支持”，其中每个逻辑结构都是可预测的，过程流从顶端进入，从底端退出，读者很容易理解。

这些结构包括顺序型、条件型和重复型。顺序型实现了任何算法规格说明中的核心处理步骤；条件型允许根据逻辑情况选择处理的方式；重复型提供了循环。这三种结构是结构化程序设计的基础，而结构化程序设计是一种重要的构件级设计技术。

关键点　结构化程序设计是一种设计技术，该技术将程序逻辑流程限制为以下三种结构：顺序型，条件型和重复型。

结构化的构建使得软件的过程设计只采用少数可预知的逻辑结构。复杂性度量（第 30 章）表明，使用结构化的构造降低了程序复杂性，从而增加了可读性、可测试性和可维护性。使用有限数量的逻辑结构也符合心理学家所谓的人类成块的理解过程。要理解这一过程，可以考虑阅读英文的方式。读者不是阅读单个字母，而是辨认由单词或短语构成的模式或是字母块。结构化的构造就是一些逻辑块，读者可以用它来辨认模块的过程元素，而不必逐行阅读设计或是代码，当遇到了容易辨 307
认的逻辑模式时，理解力就得到了提高。

无论是对于应用领域还是对于技术复杂度，任何程序都可以只用这三种结构化的构造来设计和实现。然而，需要注意的是，教条地使用这些结构在实践中会遇到困难。

14.7　基于构件的开发

在软件工程领域，复用既是老概念，也是新概念。在计算机发展的早期，程序员就已经复用概念、抽象和过程，但是早期的复用更像是一种临时的方法。今天，复杂的、高质量的计算机系统往往必须在短时间内开发完成，所以就需要一种更系统的、更有组织性的复用方法来协助这样的快速开发。

基于构件的软件工程 (Component-Based Software Engineering，CBSE) 是一种强调使用可复用的软件构件来设计与构造计算机系统的过程。考虑到这种解释，很多问题出现了，仅仅将多组可复用的软件构件组合起来，就能够构造出一个复杂的系统吗？这种工作能够以一种高效和节省成本的方式完成吗？能否建立恰当的激励机制来鼓励软件工程师复用而不是重复开发？管理团队是否也愿意为构造可复用软件构件过程中的额外开销买单？能否以使用者易于访问的方式构造复用所必需的构件库？已有的构件可以被需要的人找到并使用吗？大家渐渐地发现了这些问题的答案，而且这些问题的答案都是“可以”。

引述　领域工程是发现系统间的共性，以识别可以应用于很多系统的构件，并识别最能充分利用那些构件的程序族。

Paul Clements

14.7.1　领域工程

领域工程的目的是识别、构造、分类和传播一组软件构件，这些构件

⊖ 传统的软件构件实现处理元素，这些处理元素涉及问题域中的功能或子功能，或者涉及基础设施域中的某种性能。通常将传统构件称为模块、程序或子程序，传统构件不像面向对象构件那样封装数据。

在某一特定的应用领域中可以适用于现有和未来的软件[⊖]。总体目标是为软件工程师建立一种机制来分享这些构件，从而在开发新系统或改造现有系统时可以共享这些构件——复用它们。领域工程包括三种主要活动：分析、构建和传播。

领域分析的总体方法通常在面向对象软件工程的环境中被赋予特色。领域分析过程中的步骤为：（1）定义待研究的领域；（2）对从领域中提取的项进行分类；（3）收集领域中有代表性的应用系统样本；（4）分析样本中的每个应用，并且定义分析类；（5）为这些类开发需求模型。值得注意的是，领域分析适用于任何软件工程范型，因此，领域分析可以应用到传统的软件开发和面向对象的软件开发中。

308

建议　这一节所讨论的分析过程主要集中于可复用构件。但是，完整的COTS系统（例如，电子商务应用、自动销售应用）分析也可以是领域分析的一部分。

14.7.2　构件的合格性检验、适应性修改与组合

领域工程提供了基于构件的软件工程（CBSE）所需的可复用构件库。某些可复用构件是自行开发的，有些可以从现有的系统中抽取得到，还可以从第三方获得。

遗憾的是，可复用构件的存在并不能保证这些构件可以很容易或很有效地被集成到为新应用所选择的的体系结构中。正因为如此，当计划使用某一构件时，要进行一系列的基于构件的开发活动。

构件合格性检验。构件合格性检验将保证某候选构件能够执行需要的功能，完全适合系统的体系结构（第13章），并具有该应用所需的质量特性（例如，性能、可靠性、可用性）。

*契约式设计*这种技术着重于定义明确的和可核查的构件接口规格说明，从而使构件的潜在用户快速了解其意图。我们将称为*前置条件*、*后置条件*和*不变式*的表述加入到构件规格说明中[⊖]。表述使得开发人员知道构件提供什么功能，以及它在一定条件下的行为方式。这种表述使开发人员更容易识别合格的构件，因而更愿意在他们的设计中信任和使用这些构件。当构件有一个“经济型接口”时，契约式设计就得到了增强。构件接口具有一组最小的必要操作，使其能够完成职责（契约）。

接口规格说明提供了有关软件构件的操作和使用的有用信息，但是，对于确定该构件是否能在新的应用系统中高效复用，它并未提供需要的所有信息。这里列出了构件合格性检验的一些重要因素[Bro96]：

- 应用编程接口（Application Programming Interface，API）。
- 构件所需的开发工具与集成工具。
- 运行时需求，包括资源使用（如内存或外存储器）、时间或速度以及网络协议。
- 服务需求，包括操作系统接口和来自其他构件的支持。
- 安全特征，包括访问控制和身份验证协议。
- 嵌入式设计假设，包括特定的数值或非数值算法的使用。
- 异常处理。

309

提问　构件合格性的重要因素有哪些？

计划使用自行开发的可复用构件时，这些因素都是比较容易评估的。如果构件开发过程

⊖ 第13章中曾经提到识别特定应用领域的体系结构类型。

⊖ 前置条件是对构件使用之前必须做验证的假设所作的叙述，后置条件是对将要交付的构件可提供的服务保证的叙述，不变式是对不会被构件变更的系统属性的叙述。这些概念将在第28章介绍。

应用了良好的软件工程实践，那么之前列出的问题就都容易回答。但是，要确定商业成品构件（Commercial Off-The-Shelf，COTS）或第三方构件的内部工作细节就比较困难了，因为能够得到的信息可能只有接口规格说明。

构件适应性修改。在理想情况下，领域工程建立构件库，构件可以很容易地被集成到应用体系结构中。“容易集成”的含义是：（1）对于库中的所有构件，都已经实现了一致的资源管理方法；（2）所有构件都存在诸如数据管理等公共活动；（3）已经以一致的方式实现了体系结构的内部接口及与外部环境的接口。

实际上，即使已经对某个构件在应用体系结构内部的使用进行了合格性检验，也可能在刚才提到的一个或多个地方发生冲突。为了避免这些冲突，经常使用一种称为构件包装 [Bro96] 的适应性修改技术。当软件团队对某一构件的内部设计和代码具有完全的访问权时（通常不是这样，除非使用开源的 COTS 构件），则可应用白盒包装技术。与软件测试中白盒测试（第 23 章）相对应，白盒包装检查构件的内部处理细节，并进行代码级的修改来消除所有冲突。当构件库提供了能够消除或掩盖冲突的构件扩展语言或 API 时，可以应用灰盒包装技术。黑盒包装技术需要在构件接口中引入预处理和后处理来消除或掩盖冲突。软件团队需要决定是否值得充分包装构件，或者考虑是否开发定制构件（对构件进行设计以消除遇到的冲突）。

建议　软件团队除了需要评估为复用所做的适应性修改是否是成本合算的，还需要评估取得所需要的功能和性能是否也是成本合算的。

构件组合。构件组合任务将经过合格性检验、适应性修改以及工程开发的构件组合到为应用建立的体系结构中。为完成这项任务，必须建立一个基础设施以将构件绑定到一个运行系统中。该基础设施（通常是专门的构件库）提供了构件协作的模型和使构件能够相互协作 310
并完成共同任务的特定服务。

由于复用和 CBSE 对软件业影响巨大，因此大量的主流公司和产业协会已经提出了软件构件标准[⊖]。这些标准包括：CCM（Corba Component Model，Corba 构件模型）[⊜]，Microsoft COM 和 .NET[⊜]，JavaBeans[⊛]，以及 OSGI（Open Services Gateway Initiative 开放服务网关协议 [OSGI3]）[⊛]。这些标准中没有哪一种在产业界独占优势。虽然很多开发者已经采用了其中的某个标准，但大型软件组织仍可能根据应用的类型和采用的平台来选择使用某种标准。

14.7.3　体系结构不匹配

广泛复用所面临的挑战之一就是*体系结构不匹配* [Gar09a]。可复用的构件设计者常常对耦合构件的有关环境进行隐式假设。这些假设往往侧重于构件控制模型、构件的连接属性（接口）、体系结构基础设施以及体系结构构建过程中的性质。如果这些假设是不正确的，就产生了体系结构不匹配的情况。

设计概念，如抽象、隐蔽、功能独立、细化和结构化程序设计、面向对象的方法、测

⊖ 为了实现 CBSE，过去和现在工业界都做了很多工作，Greg Olesn[Ols06] 对此给出了精彩的讨论。Ivica Crnkovic [Crb11] 给出了关于更多近期的工业构件模型的探讨。

⊜ 关于 CCM 的进一步资料可在 www.omg.org 找到。

⊜ 关于 COM 和 .NET 的资料，可在 www.microsoft.com/COM 及 msdn2.microsoft.com/en-us/netframework/default.aspx 找到。

⊛ 关于 javabean 的最新资料可在 java.sun.com/product/javabeans/docs/ 找到。

⊛ 关于 OSGI 的资料可在 http://www/osgi.org/Main/Homepage 找到。

试、软件质量保证（SQA）和正确性验证方法（第 28 章），所有这些都有助于创建可复用的软件构件和防止体系结构不匹配。

如果利益相关者的设想得到了明确的记载，那么在早期就能发现体系结构不匹配。此外，风险驱动过程模型的使用强调了早期体系结构原型的定义，并且指出了不匹配的区域。如果不采取一些诸如封装或适配器[⊖]的机制，往往很难修复体系结构不匹配。有时甚至需要
311 通过完全重新设计构件接口或者通过构件本身来消除耦合问题。

14.7.4 复用的分析与设计

将需求模型（第 9 ～ 11 章）中的元素与可复用构件描述进行比较的过程有时称为“规格说明匹配”[Bel95]。如果规格说明匹配指出一个现有的构件和现有应用需求相符合，那么就可以从可复用构件库中提取这些构件，并将它们应用于新系统的设计中。如果没有发现这样的构件（即没有匹配），就需要创建新构件。在这一点上，当需要建立新构件时，应该考虑可复用性设计 (Design for reuse，DFR)。

正如我们已经谈到的，DFR 需要软件工程师采用良好的软件设计概念和规则（第 12 章）。但是，也必须考虑应用领域的特点。Binder [Bin93] 提出了构成可复用设计基础的一系列关键问题[⊖]。如果应用领域定义了标准的全局数据结构，则应使用这些标准的数据结构来设计构件。在应用领域内应采用标准接口协议，并且可选取一种适合应用领域的体系结构风格（第 13 章）作为新软件体系结构设计的模板。一旦建立了标准数据、接口和程序模板，就有了进行设计所依托的框架。符合此框架的新构件将来复用的可能性就比较高。

建议 如果构件必须与遗留系统及多个系统（它们的体系结构和接口协议不一致）进行接口或者集成，DFR 可能会非常困难。

14.7.5 构件的分类与检索

考虑一个大型构件库，其中存放了成千上万的可复用构件。但是，软件工程师如何才能找到他所需要的构件呢？为了回答这个问题，又出现了另一个问题：我们如何以无歧义的、可分类的术语来描述软件构件？这些问题太难，至今还没有明确答案。

可以用很多种方式来描述可复用软件构件，但是理想的描述应包括 Tracz[Tra95] 提出的 3C 模型——概念（concept）、内容（content）和环境（context），即描述构件能够实现什么功能，如何实现那些对一般用户来讲是隐蔽的内容（只有那些想要修改或测试该构件的人才需要了解），以及将可复用软件构件放到什么样的应用领域中。

为了在实际环境中使用，概念、内容和环境必须被转换为具体的规格说明模式。关于可复用软件构件分类模式的文章很多（例如，[Nir10]，[Cec06]），所有这些分类模式都应该在可复用环境中实现，该环境应具备以下几方面的特点：

312
- 能够存储软件构件和检索该构件所需分类信息的构件数据库。
- 提供访问数据库的管理系统。
- 包含软件构件检索系统（例如对象请求代理），允许客户应用从构件库服务器中检索构件和服务。

提问 构件复用环境的关键特性是什么？

⊖ 适配器是一种软件设备，它允许具有不匹配接口的客户端访问构件，方法是将服务请求翻译成可以访问原始接口的形式。

⊖ 一般来说，DFR 准备工作应当是领域工程的一部分。

- 提供 CBSE 工具，支持将复用的构件集成到新的设计或实现中。

每种功能都与复用库交互，或是嵌入在复用库中。复用库是更大型软件库（第 29 章）的一个元素，并且为软件构件及各种可复用的工作产品（例如，规格说明、设计、模式、框架、代码段、测试用例、用户指南）提供存储设施。

软件工具　CBSE

[目标] 在软件构件的建模、设计、评审以及集成为更大系统的一部分时起辅助作用。

[机制] 工具的机制各异。一般情况下，CBSE 工具对以下一项或多项工作起辅助作用：软件体系结构的规格说明和建模，可利用的软件构件的浏览及选择，构件集成。

[代表性工具]⊖

- ComponentSource（www.componentsource.com）。提供了大量被许多不同构件标准支持的 COTS 软件构件（及工具）。
- Component Manager。由 Flashline（http://www.softlookup.com/download.asp?id=8204）开发，“它是一个应用程序，能支持、促进及测量软件构件复用”。
- Select Component Factory。由 Select Business Solution（www.selectbs.com）开发，“它是一个集成的成套产品，用于软件设计、设计审查、服务 / 构件管理、需求管理及代码生成”。
- Software Through Pictures-ACD。由 Aonix（www.aonix.com）发布，对于由 OMG 模型驱动的体系结构（一个开放的、供应商中立的 CBSE 方法），它能够运用 UML 进行广泛的建模。

14.8　小结

构件级的设计过程包含一系列活动，这些活动逐渐降低了软件表示的抽象层次。构件级设计最终在接近于代码的抽象层次上描述软件。

根据所开发软件的特点，可以从三个不同的角度来进行构件设计。面向对象的视角注重细化来自于问题域和基础设施域的设计类。传统的视角细化三种不同的构件或模块：控制模块、问题域模块和基础设施模块。在这两种视角中，都需要应用那些能够得到高质量软件的基本设计原则和概念。从过程视角考虑时，构件设计采用了可复用的软件构件和设计模式，这些都是基于构件级的软件工程的关键要素。 313

在进行类的细化时，有许多重要的原则和概念可以指导设计者，包括开闭原则、依赖倒置原则，以及耦合性和内聚性概念等，这些都可以指导工程师构造可测试、可实现和可维护的软件构件。在这种情况下，为了实施构件级设计，需要细化类，这可通过以下方式达到：详细描述消息细节，确定合适的接口，细化属性并定义实现它们的数据结构，描述每个操作的处理流程，在类或构件层次上表示行为等。在任何情况下，设计迭代（重构）都是重要活动。

传统的构件级设计需要足够详细地表示出程序模块的数据结构、接口和算法，以指导用

⊖ 这里提到的工具只是此类工具的例子，并不代表本书支持选择采用这些工具。在大多数情况下，工具名称被各自的开发者注册为商标。

编程语言书写的源代码的生成。为此，设计者采用某种设计表示方法来表示构件级详细信息，可以使用图形、表格，也可以使用文本格式。

WebApp 的构件级设计既要考虑内容，也要考虑功能性，因为它是由基于 Web 系统发布的。构件级的内容设计关注展示给 WebApp 最终用户的内容对象及这些内容对象的包装方式。WebApp 的功能性设计关注处理功能，这些处理功能可以操作内容、执行计算、查询和访问数据库，并建立与其他系统的接口。所有的构件级设计原则和指导方针都适用于此。

移动 App 的构件级设计通常使用多层体系结构，包括用户界面层、业务层和数据层。如果移动 App 通常需要执行移动设备上的业务层和数据层的构件设计，那么该设备的物理硬件特性限制将成为设计上的重要约束。

结构化程序设计是一种过程设计思想，它限制描述算法细节时使用的逻辑构造的数量和类型。结构化程序设计的目的是帮助设计师定义简单的算法，使算法易于阅读、测试和维护。

基于构件的软件工程在特定的应用领域内标识、构建、分类和传播一系列软件构件。这些构件经过合格性检验和适应性修改，并集成到新系统中。对于每个应用领域，应该在建立
314 了标准数据结构、接口协议和程序体系结构的环境中设计可复用构件。

习题与思考题

14.1 术语“构件”有时很难定义。请首先给出一个一般的定义，然后针对面向对象软件和传统软件给出更明确的定义，最后选择三种你熟悉的编程语言来说明如何定义构件。

14.2 为什么传统软件当中必要的控制构件在面向对象的软件中一般是不需要的？

14.3 用自己的话描述 OCP。为什么创建构件之间的抽象接口很重要？

14.4 用自己的话描述 DIP。如果设计人员过于依赖具体构件，会出现什么情况？

14.5 选择三个你最近开发的构件，并评估每个构件的内聚类型。如果要求定义高内聚的主要优点，那么主要优点会是什么？

14.6 选择三个你最近开发的构件，并评估每个构件的耦合类型。如果要求定义低耦合的主要优点，那么主要优点会是什么？

14.7 问题领域构件不会存在外部耦合的说法有道理吗？如果你认为没有道理，那么哪种类型的构件存在外部耦合？

14.8 完成：（1）一个细化的设计类；（2）接口描述；（3）该类中包含的某一操作的活动图；（4）前几章讨论过的某个 SafeHome 类的详细状态图。

14.9 逐步求精和重构是一回事吗？如果不是，它们有什么区别？

14.10 什么是 WebApp 构件？

14.11 选择已有程序的某一小部分（大概 50 ～ 75 行源代码），通过在源代码周围画框隔离出结构化编程构造。程序片段是否存在违反结构化程序设计原则的构造？如果存在，重新设计代码使其遵守结构化编程的构造，如果不违反，对于画出的框你注意到了什么？

14.12 所有现代编程语言都实现了结构化的程序设计构造。用三种程序设计语言举例说明。

14.13 选择有少量代码的构件，并使用活动图来描述它。

14.14 在构件级设计的评审过程中，为什么“分块”很重要？

扩展阅读与信息资源

最近几年，已经出版了许多有关基于构件的开发和构件复用的书籍。Szyperski（《 Component

Software》, 2nd ed., Addison-Wesley, 2011）强调将软件构件作为有效系统构造块的重要性。Hamlet（《Composing Software Components》, Springer, 2010）、Curtis（《Modular Web Design》, New Riders, 2009）、Apperly 和他的同事（《Service- and Component-Based Development》, Addison-Wesley, 2004）、Heineman 和 Councill（《Component Based Software Engineering》, Addison-Wesley, 2001）、Brown（《Large-Scale Component-Based Development》, Prentice-Hall, 2000）、Allen（《Realizing e-Business with Components》, Addison-Wesley, 2000）以及 Leavens 和 Sitaraman(《Foundations of Component-Based Systems》, Cambridge University Press, 2000) 编写的书覆盖了 CBSE 过程的许多重要方面。Stevens（《UML Components》, Addison-Weslsy, 2006）、Apperly 和他的同事（《Service- and Component-Based Development》, 2nd ed., Addison-Wesley, 2003）以及 Cheesman 和 Daniels(《UML Components》, Addison-Wesley, 2000）则侧重于用 UML 讨论 CBSE。

Malik（《Component-Based Software Development》, Lap Lambert Publishing, 2013) 提出了建立有效构件库的方法。Gross（《Component-Based Software Testing with UML》, Springer, 2010）以及 Gao 和他的同事（《Testing and Quality Assurance for Component-Based Software》, Artech House, 2006）论述了基于构件的系统测试和 SQA 问题。

近些年出版了大量书籍来描述产业界基于构件的标准。这些著作既强调了关于制定标准本身的工作，也讨论了许多重要的 CBSE 话题。

Linger、Mills 和 Witt 的著作（《Structured Programming—Theory and Practice》, Addison-Wesley, 1979）仍是设计方面的权威著作，里面包含很好的 PDL，以及关于结构化程序设计分支的细节讨论。其他书籍则集中在传统系统的过程设计问题方面，如 Farrell（《A Guide to Programming Logic and Design》, Course Technology, 2010）、Robertson（《Simple Program Design》, 5th ed., Course Technology, 2006）、Bentley（《Programming Pearls》, 2nd ed., Addison-Wesley, 1999）以及 Dahl（《Structured Programming》, Academic Press, 1997）等。

最近出版的书籍中，专门讨论构件级设计的书很少。一般来讲，编程语言书籍或多或少关注于过程设计，通常以书中所介绍的语言为环境进行讲解，这方面有成百上千本书。

在网上有大量关于构件级设计的信息，有关构件级设计的最新参考文献可在 SEPA 网站 www.mhhe.com/pressman 上找到。

315
~
316