第5章 软件设计

本章学习目标

- 理解软件设计的目标、内容以及软件设计相关的思想。
- 掌握面向对象设计过程、描述方法、典型的内聚与耦合类型，理解基本的面向对象设计原则。
- 理解面向切面编程(AOP)的基本思想和相关概念，掌握 AOP 的基本实现方式。
- 理解契约式设计的思想，了解如何在面向对象设计中定义契约。
- 理解设计模式的思想并掌握几种常用的面向对象设计模式。
- 理解演化式设计的思想，了解典型的代码坏味道类型以及常用的软件重构方法。

本章首先介绍软件设计相关的概念，包括软件设计的目标、软件设计的各个层次以及软件设计的相关思想；接着介绍面向对象软件设计方法，包括面向对象软件设计过程、设计描述、面向对象设计原则，以及作为一种横切关注点模块化方法的面向切面编程的思想、概念和实现方式；然后介绍契约式设计的思想以及面向对象设计中的契约定义方式；接下来介绍设计模式的思想并具体介绍几种常用的面向对象设计模式；最后介绍与敏捷开发相适应的演化式设计的思想以及相关的代码坏味道和软件重构方法。

需要注意的是，软件设计包含的内容十分丰富。本章主要对软件设计的内容和思想进行概述，同时围绕面向对象设计方法详细介绍组件级详细设计。接下来，第 6 章将介绍软件复用的思想和相关技术，而第 7 章将介绍软件体系结构设计。

5.1 软件设计概述

当我们面对的开发任务不再是给定接口定义的局部编码任务，而是包含多个类或文件的组件级开发任务甚至是包含多个组件的系统级开发任务时，软件设计就变得必不可少。软件设计规划一个软件解决方案的组成单元(例如组件、模块、文件或类)及其之间的关系，覆盖体系结构设计、组件级详细设计等多个不同层次，扮演着软件需求与实现代码之间的桥梁角色。同时，培养软件设计能力需要深刻理解与软件设计相关的一些思想。

5.1.1 软件设计目标

软件设计是软件需求与实现代码之间的桥梁，起着承上启下的作用。对上而言，软件设计为软件需求的实现提供了一种抽象的解决方案规划。虽然还没有具体实现，但是软件设计明确了软件需求中所定义的功能如何分配到不同的软件单元(例如组件、模块、文件或类)上，同时非功能性的质量需求(例如可扩展性、性能、可靠性等)以何种方式实现。对下而言，

软件设计明确了每一个软件单元的开发要求(例如接口定义、功能要求等),使得它们可以以一种分而治之的方式分别被实现(有时候还可以分配给不同的开发小组和开发人员来分别实现)。

从仅包含几十上百行代码的一个类,到包含几千到几万行代码的一个模块,再到包含几百万甚至上亿行代码的整个软件系统,随着软件规模的扩大,复杂性成为软件开发的一个主要挑战。人类应对复杂性的基本手段是分解加抽象,即:通过分解将复杂问题简化为一组相对简单的问题并逐一解决,通过抽象去除与问题思考无关的细节而只保留少量关键特性。对于复杂的软件开发问题而言,需要决定如何将其分解为多个组成单元(例如组件、模块、文件或类),如何定义它们对外的接口和外部属性,以及如何确定它们之间的交互关系和集成方案。而这些正是软件设计需要考虑的问题。

另一个方面,大部分软件都不可避免地要在长期的演化过程中满足需求的不断变化,例如,新增功能和特性、业务逻辑变化等。如果软件实现方案对于这种变化完全没有任何考虑,那么响应这种变化的成本可能很高,例如,需要以一种刚性的方式破坏软件设计结构并在代码中很多地方进行修改。为此,需要通过设计对未来可能发生的变化做好准备,例如,让软件的不同组成部分之间保持相对独立和松耦合以避免变化的影响扩散到很多地方、预先设计好可扩展的接口从而使得新功能和新特性的引入不破坏原有的设计结构等。

由此可见,软件设计的主要目标是面向软件需求的要求规划软件实现方案,同时为应对软件开发的复杂性和变化性提供支持,具体包括下面几个方面。

- **软件需求和实现代码之间的桥梁**:一方面通过对于软件解决方案的规划回答软件需求如何实现的问题,另一方面为各个部分的编码实现提出明确的要求。
- **应对软件开发的复杂性挑战**:通过分解和抽象将待开发的软件分为一系列组件、模块、文件或类,使得开发人员可以以一种分而治之的方式逐步完成开发工作,每次只需要专注于某一个部分的实现而只对其他部分的抽象特性(例如接口)进行了解。
- **应对软件开发的变化性挑战**:通过良好的分解和抽象使得软件的各个组成部分保持相对独立和松耦合,同时通过灵活配置和可扩展接口等手段为未来可能的需求变化做好准备。

5.1.2 软件设计层次

作为需求到实现之间的过渡,软件设计也不是一蹴而就的,而是存在一个逐步精化的过程,就像设计一栋大楼也会从整体蓝图开始然后逐步考虑细节设计。软件设计的主要层次如图 5.1 所示,自下而上包括四个主要层次。

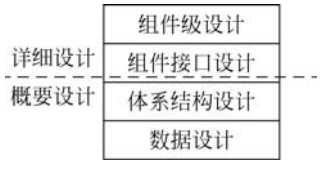

图 5.1 软件设计的各个层次

- **数据设计**:软件系统的全局数据结构设计,包括数据实体及其属性和相互之间的关系。与组件的内部数据结构不同,这些数据结构是系统的全局数据结构、由所有组件共享访问(例如,以数据库或文件的方式)。
- **体系结构设计**:表示软件系统的高层设计结构,决定了系统的高层分解结构,即组件划分、组件的外部属性以及组件间的交互关系定义。此外,体系结构设计还确定了其他一些全局性(即超出单个组件范围)的设计决策,例如,开发语言、异常处理方式等。

- **组件接口设计**：组件之间的交互接口设计，包括每个组件接口的功能和非功能性（如性能、吞吐量）要求、接口交互协议、接口操作及实现类定义等。
- **组件级设计**：组件内的具体设计方案，例如，面向对象设计类以及类之间关系的定义、组件内部的局部数据结构和算法设计等。

在正向的软件开发过程中，软件设计一般按照如图 5.1 所示的层次自下而上进行，即从数据设计、体系结构设计、组件接口设计到组件级设计。许多企业在实践中都会编写两种软件设计文档，即概要设计（Preliminary Design）文档和详细设计（Detailed Design）文档。顾名思义，概要设计是一种初步的概要性的设计，大致对应于数据设计、体系结构设计和一部分组件接口设计；详细设计是一种更加具体和详细的设计，大致对应于大部分组件接口设计和组件级设计。其中，在概要设计阶段，组件接口设计一般只是随着体系结构设计一起确定每个接口的功能和非功能性（如性能、吞吐量）要求、接口交互协议等；而在详细设计阶段，组件接口设计会详细考虑每个接口所包含的操作以及实现类的定义。

对于软件开发人员而言，在参加局部的编码工作的基础上一般都会参与详细设计，因为编码活动与文件、类级别上的详细设计联系紧密。甚至按照普遍的理解，详细设计的绝大部分活动都被认为与编码、调试等活动一起属于软件构造过程的一部分（McConnell，2006）。而概要设计则主要由架构师来进行，一般的开发人员在成长为架构师之前可能很少有机会考虑体系结构设计等概要设计内容。按照软件开发人员的技术发展路径，本章中将主要介绍详细设计层面的设计方法和技术，第 7 章将介绍软件体系结构设计。

5.1.3 软件设计思想

掌握软件设计方法和能力首先需要深刻理解一些软件设计思想。以下介绍几种最重要的软件设计思想。

1. 分解与抽象

如前所述，分解（Decomposition）和抽象（Abstraction）是人类应对复杂性的两个基本手段，也是软件设计思维的重要基础。分解比较容易理解，就是将软件不断地分解为更细粒度的代码单元，例如，从组件、模块到文件和类，直至每个单元的规模和复杂性都小到可以直接进行编码实现。而抽象则意味着忽略无关细节，只保留与当前问题相关的关键信息。例如，我们在中学物理的力学计算中经常用到的质点的概念就是一种抽象，它具有质量和位置但其他细节（例如体积、形状等）都被我们忽略了。软件设计中的接口定义就是一种抽象，基于这种抽象可以针对一个组件、模块或类的接口进行编程，此时关心的是接口操作的功能、参数、返回值、前后置条件、通信协议等接口定义方面的信息，而忽略了接口的内部数据结构、算法等实现细节。分解往往需要与抽象的思想相结合才能发挥应对复杂性的作用。如果没有抽象，那么意味着分解得到的每一个代码单元的开发人员都要了解其他相关代码单元的所有细节，这违背了我们控制复杂性的初衷。

抽象包括数据抽象和过程抽象（Pressman，2021），前者是对目标对象的数据化抽象描述，而后者则是对一系列过程性步骤和指令序列的整体抽象。好的抽象应该屏蔽底层细节，突出事物的本质特性，同时符合人的思维方式，从而实现降低复杂性的目标。与此同时，由于针对抽象编程的实现方案不依赖于许多无关细节，因此好的抽象还能极大提高程序的可迁移性。例如，针对抽象的设备（例如抽象的打印机）编程的代码可以很容易地与不同型号

的设备(例如具体的打印机)一起工作,只要这些设备都能实现同样的抽象设备接口(例如接受打印作业、返回打印机状态等)。

如图 5.2 所示,有两种针对银行账户(Account)对象的抽象方案。其中,左边的方案为账户定义了一系列操作,包括存款(deposit)、取款(withdraw)、转账(transfer)、查询余额(getBalance);右边的方案为账户定义了两个基本操作,即改变余额(changeBalance)、查询余额(getBalance)。显然,左边这个方案更符合人对于"账户"这个概念的理解,而右边的方案需要有一些更低级的操作来实现账户的各种功能(例如,通过增加或减少账户余额来实现存款、取款的目的)。除此之外,左边的抽象方案也能更好地适应变化。例如,如果银行改变了账户取款的手续费策略,那么只需要修改取款(withdraw)操作的内部实现以调整手续费扣除策略,而调用该操作的客户端无须修改代码。如果采用右边的抽象方案,那么调用改变余额(changeBalance)操作的客户端必须修改计算手续费以及调整余额的代码。

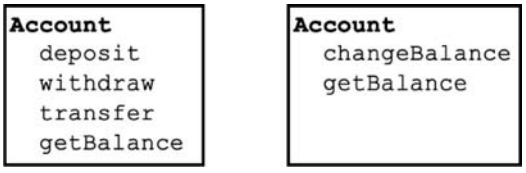

图 5.2　两种不同的账户对象抽象

2. 软件体系结构

对于大规模软件系统,特别是在网络上部署的分布式软件系统而言,考虑文件和类级别上的详细设计是不够的。这种软件系统包含大量的代码,不同部分代码之间的交互关系复杂,经常还涉及跨进程、跨网络的通信。另一方面,这种软件系统往往有着非常高的性能、可靠性、可维护性、可扩展性等非功能性质量要求。因此,这些软件系统需要在更高的抽象层次上考虑整体的设计方案,即软件体系结构设计,从而对软件的组件划分、分布式部署方式以及组件间的通信协议和交互方式进行整体性的规划,以满足各种非功能性质量要求。

软件体系结构给出了软件系统的顶层设计方案,其内容主要包括一组软件组件、软件组件的外部属性、软件组件之间的关系以及其他软件系统的全局的实现约定(例如,编程语言、异常处理策略、数据库等资源使用方式)。软件体系结构设计充分体现了分解与抽象的基本原则:一方面,体系结构设计给出了系统的分解结构,使得不同的开发小组和开发人员可以分别负责其中不同组件的开发任务;另一方面,体系结构设计也给出了系统实现方案的一种抽象表示,同时组件的外部属性描述使得每个组件的开发人员可以在无须了解实现细节的情况下理解其他组件并考虑集成关系(例如如何调用另一个组件)。

软件体系结构是软件项目分工合作的基础:不同的开发小组和开发人员任务分配的重要依据就是软件体系结构所给出的组件划分,软件组件的外部属性为每个组件的开发提出了具体要求,而组件间关系定义以及其他全局性的约定为这些组件的最终顺利集成提供了基础和保障。此外,软件体系结构设计还有其他几个方面的作用,包括确认设计方案是否能有效满足需求、为满足未来演化和复用的需要而做出规划、作为多种候选技术方案对比选择的依据、识别并降低软件实现的风险、作为相关涉众沟通和交流的基础等。

软件体系结构设计是一种高度抽象的复杂设计方案,难以通过单一的方式完整表达,因此通过多个不同视角的软件体系结构多视图描述已经成为广泛接受的共识。其中,Philippe

Kruchten(1995)提出的“4+1”视图使用最广泛，其中包括四种常用的软件体系结构视图类型，即逻辑视图、开发视图、运行视图、部署视图，以及以使用场景为纽带将其他四种视图联系到一起的用例视图。

对于设计软件体系结构的架构师而言，他们所具有的设计经验很大程度上体现为对于各种常用的体系结构风格和模式的理解和掌握。软件体系结构风格并不是具体的体系结构设计方案，而是一种抽象的设计样式，包括系统中的组件类型以及不同类型的组件之间的关系和组织方式等。典型的软件体系结构风格包括 C/S 和 B/S 风格、层次化风格、黑板风格、管道和过滤器风格，以及近几年随着云计算软件技术和应用的发展而流行起来的微服务风格。

3. 关注点分离

关注点是指软件系统中所实现的某种功能或特性，可以理解为相关涉众(即 stakeholder，表示与软件系统相关的各种人或组织，详见 8.1.1 节)所关注和关切的方面。例如，对于校园一卡通系统而言，刷卡识别师生身份、通过手机支付进行充值、刷卡付款消费等都是一卡通用户(学校师生员工)的基本关注点，而学校从系统安全角度所关注的数据加密传输、所有操作都要进行日志记录等问题也都是关注点。

关注点分离是一种重要的设计原则，其基本思想是将软件系统的整体需求分解为尽可能小的关注点并分解到不同的模块单元(例如模块、包、类、方法等)中实现，不同的关注点尽量不要混杂在一起。在模块化软件设计中，不同模块单元的划分本身就体现了关注点分离的思想。例如，在面向对象软件设计中，类作为一种基本的模块单元对关注点进行了划分，与相同或相近的关注点相关的职责和功能被分配到同一个类中；按照 MVC(即模型、视图、控制器)模式的设计要求，表示视图的用户界面应该关注于界面展示效果以及用户交互，与业务数据表示、处理数据的业务逻辑以及衔接用户界面与业务逻辑的控制逻辑相分离。

然而，软件设计中的关注点经常存在散布和混杂的问题，即与同一关注点相关的职责散布在多个模块单元中，而同一模块单元中又混杂了多个关注点的职责。例如，在面向对象软件设计中，很多类都需要执行身份认证、权限检查、日志记录等处理，这些所谓的横切关注点很难被封装在单个类中，因此造成了相关职责的散布和混杂。面向切面的编程(Aspect Oriented Programming，AOP)为这种横切关注点的封装提供了一种有效的手段(详见 5.4 节)。按照 AOP 的思想，这些横切关注点可以被封装为切面(Aspect)并通过声明式的方式编织到需要执行相关处理的地方，从而与各个类中自身的主要关注点相分离。

4. 模块化

模块化(Modularity)是指将整个产品或系统分解为大小合适、相对独立的模块。模块化的思想在制造、建筑以及计算机硬件等行业中已经得到了广泛应用。例如，汽车制造业通过整车设计将汽车分解为模块化的零部件，然后通过加工制造和外部采购等方式准备好全部零部件，最终通过组装的方式得到完整的汽车。模块化设计对于模块的独立性有很高的要求。模块独立一方面使得各个模块的生产制造可以相对独立地进行，另一方面可以在不破坏整体结构的基础上实现模块替换和扩展。

软件设计中的模块化是软件设计中的分解和抽象思想的具体体现。一个软件系统的模块结构给出了系统的分解方案，使得开发人员可以以分而治之的方式分别实现每个模块；同时，每个模块通过所声明的接口提供外部抽象，使得其他开发人员在无须了解模块内部实

现细节的情况下就可以调用模块的功能以及实现模块集成。同样，软件设计中的模块化也强调模块独立性。软件模块的独立性一般可以用内聚度和耦合度来衡量，好的模块化设计应该实现模块的高内聚和低耦合，即：模块内部紧密相关共同完成所聚焦的职责；模块之间松散关联，依赖较少，相互影响较小。模块化设计做得不好的软件通常给人的感觉就是代码中各个部分之间的依赖关系复杂，整体的代码逻辑结构及模块边界不清晰，各部分之间粘连严重。这种情况经常被形象地比喻成大泥球、毛线球或是意大利面式的代码。

需要注意的是，模块化设计的思想适用于软件设计的各个层次。例如，在高层的软件体系结构设计中，各个组件（体现为包）之间应当相对独立，体现模块化设计的思想；在组件内的详细设计中，各个类之间也应当相对独立，体现模块化设计的思想。

5. 信息隐藏

信息隐藏（Information Hiding）是指一个模块（例如组件或类）将实现细节隐藏在内部，仅通过受限的接口对外提供访问。如果没有实现信息隐藏，而是将模块内部的实现细节都对外暴露，那么即使模块分解得当也会造成不必要的模块间耦合。在面向对象软件设计与实现中，类以及类的属性和方法的访问修饰符（如 public、private 等）可以用来实现信息隐藏设置。例如，一个包（package）中作为“门面”（facade）让外部可见的一些类的访问修饰符可以设置为 public，而其他对外隐藏的类可以设置为 protected；一个类（class）中作为对外接口一部分的属性和方法（一般建议属性不要直接对外开放）可以设置为 public，而其他对外隐藏属性和方法可以设置为 private。

图 5.3 描述了校园一卡通系统中的校园卡类（CampusCard）的两种设计方案。其中，左边的设计方案将表示校园卡消费记录的数组直接暴露给外部访问，这种方式虽然简洁但会带来不必要的耦合：访问校园卡类的其他类将依赖于校园卡类中的消费记录数组（consumpList），这意味着如果校园卡类修改内部数据结构（例如，将消费记录数组改为动态数组等其他数据结构），那么访问校园卡类的其他类也将不得不修改。与之形成对比的是，如果采用右边的设计方案那么就可以避免这种不必要的耦合，因为其他类分别通过 getConsumpNum 和 getConsumptionAt 方法获得消费记录的条数以及指定序号的消费记录，因此即使校园卡类修改内部数据结构但只要依旧提供这两个方法那么其他类就无须修改。这两种设计方案的区别在于右边的设计方案对消费记录查询操作进行了抽象同时隐藏了关于保存消费记录列表的具体数据结构的信息。

信息隐藏可以带来多个方面的好处。首先，信息隐藏通过屏蔽实现细节以及暴露抽象接口的方式降低了其他模块开发者对于当前模块的认知复杂性。其次，信息隐藏通过抽象降低了内部实现细节的变化对于其他模块的影响。最后，信息隐藏通过受控接口提供访问，可以更好地实现对于内部数据和操作的保护。例如，在如图 5.3 所示的例子中，左边的设计方案将消费记录数组直接暴露出来，外部其他模块就有可能对数组中的消费记录内容进行修改；而右边的设计方案按照指定的序号返回复制的消费记录信息，从而杜绝了外部对消费记录内容的修改。

6. 重构

软件重构（Refactoring）是指在不改变代码外在行为的前提下，对代码做出修改以改进程序的内部结构，可以简单理解为代码写好之后改进它的设计（Fowler，2010）。软件重构的目的一般是提高软件的可维护性和可扩展性。重构可以在多个层面上发生，包括：在保

```
public class CampusCard{
  //消费记录数组
  public Consumption [] consumpList;
}
```

```
public class CampusCard{
  //消费记录数组
  private Consumption [] consumpList;
  //获取消费记录条数
  public int getConsumpNum(){
    …
  }
  //按照指定序号返回一条复制的消费记录
  public Consumption getConsumptionAt(int index){
    …
  }
}
```

图 5.3　软件设计中的信息隐藏

持软件系统整体行为不变的情况下对其体系结构的大规模重构；在保持软件模块整体行为不变的情况下对其类级别上的详细设计进行设计重构；在保持类行为不变的情况下对其内部设计进行代码级的重构。

软件重构可以理解为一种对于软件内部设计和实现的“整理”。就像图书馆书架上的图书以及超市货架上的商品经过一段时间之后需要进行整理一样，软件经过不断的修改之后其内部的设计和实现结构也会退化。如果任由这种趋势发展下去，那么软件将变得越来越难以维护和扩展。通过软件重构可以阶段性地改进软件设计和实现质量，缓解软件内部质量退化的趋势，因此具有重要的意义。

软件重构也是敏捷方法所推崇的一种重要的开发实践，是实现所谓的演化式设计（即随着迭代化的开发过程逐步完善软件设计）的一种重要手段。敏捷方法强调要避免过度设计，即不要在软件开发初期基于对于需求的发展变化进行预测并在此基础上做出完整的设计方案。敏捷方法推崇的是在增量和迭代化的特性实现过程中不断发现设计中的问题并通过重构不断进行改进，其背后的假设是“好的设计是演化出来的而不是提前设计出来的”。为此，软件开发人员应当持续对软件设计质量进行分析，及时发现过长的类及方法、复杂的嵌套分支结构、重复的实现代码等设计问题迹象及相应的改进机会，并通过重构实现软件设计改进。

7. 复用

软件复用（Reuse）是指在不同的软件系统中或者同一软件系统的不同部分重复使用相同或相似的软件代码、软件设计或其他相关的软件知识。最常见的软件复用形式是基于各种软件开发库（例如，编程语言自带的开发库以及第三方库）的 API 调用。除了代码层面的复用外，软件设计层面也存在多种不同的复用形式。例如，实现通用功能的软件组件可以通过明确定义的 API 直接进行复用；体现通用设计方案的设计模式提供了可复用的设计思想，在设计模式基础上结合具体问题需要进行实例化可以得到高质量的软件设计方案；面向特定类型软件应用的软件框架实现了软件的整体框架结构，同时为特定应用的定制和扩展提供了支持。这些设计级复用都需要在软件设计的过程中加以考虑，包括适合于所开发的软件系统的软件框架、设计模式和通用组件等。

软件复用包括两个方面的考虑，即面向复用的软件开发（Software Development for Reuse）以及基于复用的软件开发（Software Development with Reuse）。面向复用的软件开发是指软件开发要为未来的复用打好基础并提供可复用的软件资产。为此，软件设计应当考虑为未来的软件开发创造复用机会，例如，将通用的软件功能与特定应用的功能相分离并

实现为提供标准化接口的通用组件；面向一系列相似的软件应用设计一种通用的体系结构，在此基础上实现共性部分形成通用基础设施，同时为特定应用的定制和扩展提供途径。基于复用的软件开发是指软件开发要充分利用潜在的复用机会。为此，软件设计应当考虑适合于当前软件系统的软件框架、设计模式和软件组件。

5.2 面向对象设计

C++、Java、C#等面向对象编程语言在当前的软件开发中仍然占据着主导地位。与之相对应的面向对象设计方法也在组件级设计以及组件接口设计上扮演着重要的角色。使用面向对象语言实现的组件及其接口都需要通过面向对象设计方法来确定所需要定义的类和接口及其之间的关系，所得到的面向对象设计方案可以通过 UML 图来进行描述。按照模块化设计的思想，面向对象设计中的类应当相对独立，这种独立性可以使用类的内聚度和耦合度来衡量。面向对象设计存在一些常用的基本原则，可以为开发人员提供具体的设计指导。此外，对于横切关注点，可以以切面（Aspect）的方式对其进行封装并与相关的类解耦，从而实现更好的模块独立性。

5.2.1 面向对象设计过程

面向对象软件设计的起点是给定软件组件的设计要求，包括整体功能、对外接口等。对于规模较小的软件系统，开发人员也可能会直接使用面向对象设计方法实现系统的整体设计，此时面向对象软件设计的起点是整个系统的软件需求。面向对象软件设计一般需要通过以下过程来实现。

1. 识别设计类

在面向对象设计与实现中，类都是一个基本的组织单元。类是数据（属性）和操作（方法）的统一封装体，构成了设计、实现和测试的基本单位。需要注意的是，我们通过 Java 等面向对象编程语言所编写的程序中的类是实现类，而本节所讨论的是软件设计中所考虑的设计类。设计类在很大程度上会转换为对应的实现类，但实现类可能会根据需要增加一些实现细节（例如增加一些私有属性和方法）。

识别设计类一般可以考虑以下几个方面的来源。

- 来自问题域中的设计类：问题域是指待开发的软件系统所需要满足的现实世界问题领域，往往会在软件需求中进行刻画和描述。问题域中存在两种潜在的设计类来源。一种是名词性的业务实体对象，往往会在需求描述中以名词或名词性短语的形式出现，可以是某种物理实体（例如图书副本，即实体书）、信息实体（例如借阅记录）、人（例如读者）、组织机构（例如学院）、建筑或地点（例如图书馆分馆）等。另一种是具有业务含义的处理过程，用于表示处理过程中的状态，往往会在需求描述中以动词的名词形式出现，例如，表示学生注册的设计类可以描述当前的注册状态（例如注册进行到哪一步了）以及中间的处理过程（例如缴费通过何种方式办理以及谁经办的）。
- 位于接口上的设计类：待开发的软件系统与用户、硬件以及其他软件系统之间都有可能存在相应的接口（其中与用户的接口一般被称为用户界面），这些接口上存在一

些与接口交互以及所传递的信息相关的设计类。一方面，需要考虑实现软硬件接口包装以及用户界面自身的设计类。例如，校园一卡通系统使用第三方在线支付服务实现充值和罚款支付等功能，此时需要考虑通过一个支付接口类封装第三方在线支付服务。另一方面，这些接口上所传递的一些信息也需要一些对应的设计类来处理，例如，校园一卡通系统向第三方支付服务所发送的支付请求。

- 与基础设施相关的设计类：软件系统中的业务逻辑处理需要在计算、存储、网络等基础设施基础上才能实现，因此软件设计过程中还需要考虑与这些基础设施相关的设计类。典型的基础设施相关设计类包括实现数据库和其他存储资源访问的设计类、实现跨网络或跨进程通信的设计类等。

从以上来源发现潜在的设计类之后，还需要判断其是否符合设计类的一些基本要求，即包含多个同类对象所共有的且是当前软件系统所需要的属性及相关操作。例如，对于校园一卡通系统而言，刷卡消费点（例如校园内每一处可以刷校园卡消费的食堂和超市）是一个重要的问题域实体，但是否将其作为一个设计类则需要考虑系统的需要：如果需要对刷卡消费点进行专门的管理，例如，管理消费点的开设和关闭、统计各个消费点的消费额等，那么消费点实体具有多个系统所需要的属性和操作，可以作为一个设计类；如果刷卡消费点不需要做专门的管理而只是在消费记录中简单进行备注，那么将消费点作为其他设计类（例如刷卡请求、消费记录等）的一个字符串类型的属性即可。

2. 明确设计类职责和协作

在初步识别出的候选设计类的基础上，需要进一步考虑各个设计类的职责以及类间协作关系。每个设计类所承担的职责一部分需要利用自身所具有的能力来完成，另一部分则依赖于与其他类的协作关系。一个类所具有的能力具体体现为所包含的属性以及方法，其中，属性代表一个类自身所具有的信息和知识，而方法则代表一个类所能够执行的操作。而类间的协作关系则包括类间继承、接口实现、功能依赖、属性访问等。

类的职责分配是面向对象软件设计的一个主要难点，在很大程度上决定了软件设计质量。好的设计类职责分配应当较好地实现类之间关注点分离和相对独立性，同时能够较好地支持未来的扩展。此外，还需要综合考虑设计类的数量和类的粒度，避免职责过于集中或者过于分散。

类的职责分配确定了软件的类分解结构以及类与类之间的边界，其基本原则是关注点分离，即每个类都应该具有明确且聚焦的职责。为此，可以考虑在候选设计类的基础上按照职责单一、相对独立的原则确定类与类之间的职责分配和边界划分。其中，职责单一的认定首先需要考虑纵向的业务关注点分离。例如，在校园一卡通系统中的图书馆管理子系统中，关注书名、出版社、简介等图书基本信息及其查询的图书类应当与关注出借状态、存放位置的图书副本类相分离。此外，职责单一的认定还需要考虑横向的技术层次划分。例如，图书馆管理子系统包括用户界面层、业务层、数据层等不同层次，因此负责图书信息录入和检索的界面类应当与相应的业务逻辑类和数据实体类相分离。在此基础上，可以按照高内聚、低耦合的标准衡量每个类的独立性并进行调整，例如，将内聚度较低的类拆分成多个更小但更内聚的类，而将多个耦合度较高的类的部分职责进行合并。

为了进一步降低类间耦合，同时更好地支持未来可能的扩展，还需要进一步考虑通过引入抽象类和接口等手段进行一些共性抽象。例如，将一个类对另一个类的具体依赖变成对

一个抽象接口的依赖，这个抽象接口仅保留所依赖的操作的最小集合，这样类之间的依赖性更小同时也方便了未来的扩展（实现同一接口的类可以相互替换）。

确定所有设计类的职责之后，类与类之间的协作关系就会逐步浮现出来。每个类在实现自身的职责时，凡是通过自身能力（即属性和方法）无法完成的部分都需要请求其他类的协作。例如，一个图书服务类在实现在线预借图书功能时需要请求电子邮件类进行邮件通知发送（图 5.4）。

3. 细化设计类内部细节

确定每个设计类的职责和协作关系后，接下来就可以进一步细化类的内部细节，包括详细的属性和方法描述以及关键的内部数据结构和算法。

类属性需要细化的主要是访问修饰符、属性类型及初始值（默认值）。面向对象设计中一般都建议避免将属性设置为公开（public），对于属性的访问（读取或修改）一般应当通过方法以一种受控的方式（方法中可以内置合法性检查等控制逻辑）来实现。方法需要细化的主要是访问修饰符、参数及类型、返回值类型、前置及后置条件。面向对象设计一般都会将作为类的对外接口一部分的方法公开。方法的前置及后置条件是指方法执行之前和之后参数、返回值及类的状态变量（属性）等应当满足的条件，它们定义了方法及其调用者之间的契约（详见 5.3 节），为类间接口提供了一种精确定义其功能语义的方法。

内部数据结构和算法主要是定义类内部需要用到的局部数据结构和算法。其中，算法一般可以通过流程图、伪代码等方式进行描述。按照信息隐藏的设计原则，类内部使用的数据结构和算法一般应当对外隐藏（图 5.3），这要求类的外部接口设计应当避免依赖于内部所采用的数据结构和算法。

5.2.2 面向对象设计描述

在面向对象软件设计中，类是一个基本的设计单元。如前所述，面向对象软件设计的主要任务是确定设计类、类的职责以及类间交互关系。因此，面向对象软件设计的描述可以围绕设计类的职责和交互关系来进行。统一建模语言（Unified Modeling Language，UML）为这些描述提供了丰富的图形种类，这里主要介绍其中三种，即类图、顺序图、通信图。

UML 类图可以描述面向对象的静态设计结构。描述图书馆管理子系统部分静态结构设计的 UML 类图如图 5.4 所示。图中每个矩形组合代表一个类，上下两部分分别是类名和方法，这里省去了类的属性、方法返回值和一部分方法。其中一个特殊的类 I_Payment 是一个接口，这是利用 UML 的构造型（stereotype）机制定义的。图中的箭头表示类与类之间的各种静态结构关系，包括：虚线箭头表示类之间的依赖关系，具体表现为方法调用、属性访问等，例如，提供图书相关服务功能的 BookService 类依赖于 Email 类进行邮件发送、依赖于 I_Payment 接口实现费用支付；菱形箭头表示类之间的聚集关系，具体表现为一个类引用另一个类的实例对象作为属性，例如，表示图书拷贝的 BookCopy 类引用了图书类 Book 的实例对象；实线空心三角形箭头表示类之间的继承关系，即父类和子类的关系，例如，表示小说的 Novel 和表示教科书的 TextBook 都继承了图书类 Book；虚线空心三角形箭头表示类的接口实现关系，例如当前的两种具体支付方式 ABCPay 和 XYZPay 都实现了支付接口 I_Payment。

UML 顺序图可以描述在特定场景中类与类之间的动态交互关系，例如，类与类之间

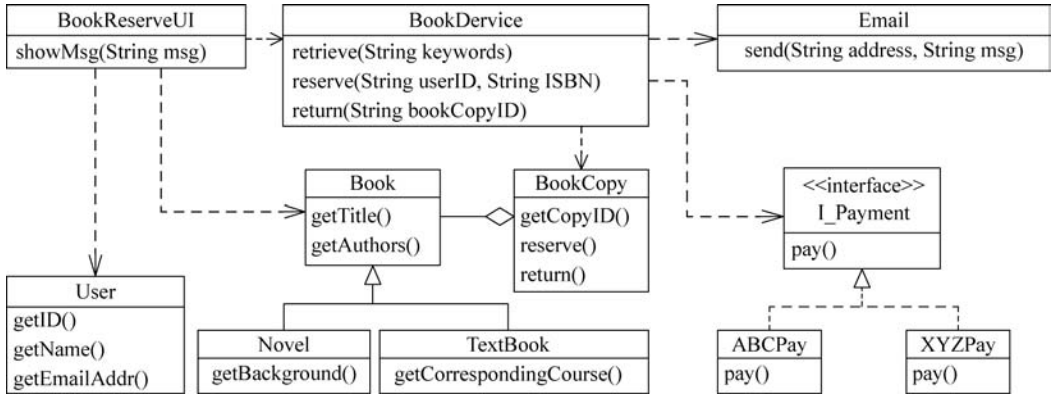

图 5.4　图书馆管理子系统静态结构设计(UML 类图)

的消息发送及其先后顺序等。描述图书馆管理子系统部分动态交互设计的 UML 顺序图如图 5.5 所示。图中顶部的圆角矩形表示交互的参与方,这里都是相关设计类的实例对象。每个参与方下面的虚线代表时间线,从上到下表示时间先后顺序。时间线上的矩形条表示激活条,表示所对应的参与方在此期间处于激活状态。不同参与方之间的横向箭头表示消息发送,其中包括多种消息类型:实线三角形箭头表示同步请求消息,即一个参与方向另一个参与方发出请求消息并在收到返回之前进行阻塞等待;虚线线型箭头表示同步调用的返回消息;实线线型箭头表示异步请求消息,即一个参与方向另一个参与方发出请求消息后不进行阻塞等待。图 5.5 描述的是用户在线预借图书的交互过程:图书预借用户界面(BookReserveUI)先后请求用户类(User)分别获得用户 ID 和邮件地址后,再请求图书服务类(BookService)进行图书预借;接下来,图书服务类请求图书拷贝类(BookCopy)进行预借,然后通过异步方式请求邮件类(Email)发送通知邮件并向图书预借用户界面返回结果;最终,图书预借用户界面显示预借结果消息。

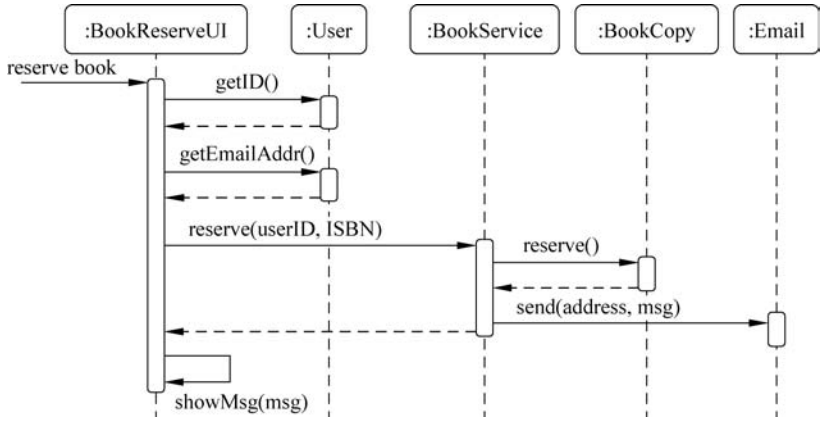

图 5.5　　图书馆管理子系统动态交互设计(UML 顺序图)

UML 状态机图可以通过状态转换的方式描述类的实例对象的行为。描述图书馆管理子系统中图书拷贝类(BookCopy)的行为设计的 UML 状态机图如图 5.6 所示。图中每个圆角矩形表示一个状态,实心圆表示一个特殊的初始状态,箭头表示状态之间的转换关系,

箭头上的方法表示触发状态转换的方法调用。通过这个状态机图可以看出，一个图书拷贝对象一开始自动处于可用状态(Available)，如果调用预借方法(reserve)则进入被预借状态(Reserved)，如果调用借书方法(borrow)则进入借出状态(Borrowed)；在被预借状态下，如果调用取书方法(take)则进入借出状态，如果调用取消预借方法(cancelReserve)则回到可用状态；在借出状态，如果调用还书方法(return)则回到可用状态。通过这种行为描述，可以理解图书拷贝类的实例对象的整体行为。

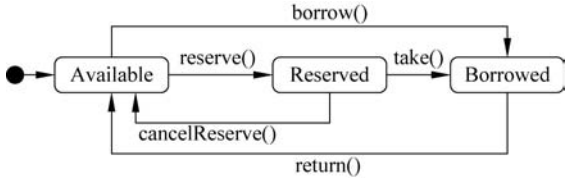

图 5.6　图书馆管理子系统 BookCopy 类行为设计(UML 状态机图)

除了以上这些图之外，UML 还有其他一些图形种类可以用于面向对象软件设计描述。例如，UML 包图可以以包的形式描述更大粒度的模块之间的静态结构关系；UML 通信图可以在展示类的实例对象之间关系的基础上描述对象间消息发送的顺序；UML 活动图可以用于描述由类方法调用构成的业务处理过程以及算法流程。

5.2.3　内聚和耦合

内聚和耦合是两个对于软件设计十分重要的概念。正如前面所提到的，作为软件设计的一个重要目标的模块独立性通常可以用内聚度和耦合度来衡量。总的来说，我们希望实现高内聚和低耦合的模块化设计，但具体内聚度能做到多高、耦合度能做到多低还是与具体的功能和业务逻辑相关。因此，需要了解一些常见的内聚和耦合的形态以及它们所适合的模块层次(如包、类、方法等)。

1. 常见的内聚形态

高内聚强调一个模块的职责应该明确且专一，而且所包含的内容都是与该职责密切相关、不可或缺的部分。Lethbridge(2001)总结了几种常见的内聚形态，如表 5.1 所示(按照内聚度从高到低排列)。

表 5.1　常见的内聚形态

内聚形态	含　义	适用层次	例　子
功能内聚	模块仅执行单个计算任务并返回结果，并且不存在任何副作用(例如修改数据库或文件、产生界面提示等)	操作(即方法或函数)	一个计算数学函数(例如 sine、cosine)的方法
层次内聚	模块中仅包含一组密切相关的功能或服务，这些功能或服务形成一个严格的层次结构，其中高层可以访问较低层次上的功能或服务，反之则不行	包、类	一个类对外提供生成成绩单的方法，内部的学生信息验证、成绩信息读取等方法构成严格的调用层次
通信内聚	模块中仅包含访问同样一组数据的操作	类	一个购物车类中所包含的都是针对购物车数据的操作，例如，添加商品、移除商品、修改商品数量等

续表

内聚形态	含　义	适用层次	例　子
顺序内聚	模块中包含一组顺序性的处理过程，每一个过程都为后一个过程提供输入	包、类	一个文本识别组件(包)包含三个顺序执行并存在输入输出关系的过程：位图区域划分、区域内字符识别、整体文本识别
过程内聚	模块中包含一组依次执行的处理过程，但过程之间并不存在输入输出关系	包、类	一个实现报表打印的类由分别打印表头、表格内容、表尾的方法组成
时间内聚	模块中包含一组在软件运行的某个阶段一起被执行的处理过程，但过程之间没有明确的顺序关系	包、类	一个负责系统初始化的类由一组分别执行不同初始化功能的方法组成
功用内聚	模块中包含一组主题相关的功能实现，这些功能可以单独被调用	包、类	一个数学函数类(如 java. lang. Math)提供一组实现常用数学函数(如三角函数)的方法

在可能的情况下，都应当尽量实现内聚性较高的内聚形态，特别是功能内聚、层次内聚和通信内聚。

图 5.7 展示了一个实现功能内聚设计的例子。图中左边的订单(Order)类设计通过一个方法实现待支付的订单金额的计算，其中包括订单总金额的计算和折扣金额的计算两部分。而右边的订单类设计则将该方法分解为两个分别计算订单总金额和折扣金额的方法，由此得到的两个更小的方法显然更符合功能内聚的要求。实现功能内聚的方法职责更单一，关注点更聚焦，也更容易维护。试想一下，如果订单折扣规则发生变化，那么开发人员只需要理解一个更小的折扣金额计算方法(calculateDiscount)并做出修改。此外，实现功能内聚的方法也更容易被复用。例如，如果另一个项目想复用订单类但折扣计算方式不同，那么订单总金额计算的方法(calculateTotal)还是可以原样被复用。

```
public class Order{
  …
  //根据订单金额和折扣规则计算需要支付的金额
  public float calculateAmountToPay(){
    …
  }
}
```

```
public class Order{
  …
  //计算订单总金额
  public float calculateTotal(){
    …
  }
  //计算折扣金额
  public float calculateDiscount(){
    …
  }
}
```

图 5.7　实现功能内聚的设计

图 5.8 展示了一个实现通信内聚设计的例子。图中左边的购物车(ShoppingCart)类包含一组访问购买商品列表(shoppingList)的方法(即添加商品、移除商品、修改商品数量)，但同时也包含一个依赖于用户(User)类的读取当前用户可用优惠券列表的方法(getCoupons)，因此不符合通信内聚的要求。为此，可以像图 5.8 右边所展示的那样将读取当前用户可用优惠券列表的方法与表示当前用户的属性(currentUser)一起从购物车类中移除，这样购物车类就符合通信内聚的要求了。事实上，原来的设计还导致了购物车类与用户类之间不必

要的耦合：无论用户是否有优惠券，购物车类都只需要记录并管理购买商品列表即可。

```
public class ShoppingCart{
  private ArrayList shoppingList;
  private User currentUser;
  …
  //添加商品
  public void addProduct(Product product){
    …
  }
  //移除商品
  public void removeProduct(Product product){
    …
  }
  //修改商品数量
  public void changeProductNum(Product product,
    int number){
    …
  }
  //读取当前用户可用优惠券列表
  public Coupon[] getCoupons(){
    …
  }
}
```

```
public class ShoppingCart{
  private ArrayList shoppingList;
  …
  //添加商品
  public void addProduct(Product product){
    …
  }
  //移除商品
  public void removeProduct(Product product){
    …
  }
  //修改商品数量
  public void changeProductNum(Product product,
    int number){
    …
  }
}
```

图 5.8　实现通信内聚的设计

2. 常见的耦合形态

低耦合强调模块之间相互依赖尽可能少，这样理解和修改一个模块时不用过多考虑其他模块，而修改后对软件其他部分的影响也比较小。此外，低耦合的软件设计中，模块也更容易被单独取出并在其他项目中进行复用。Lethbridge(2001)总结了几种常见的耦合形态，如表 5.2 所示(按照耦合度从高到低排列)。

表 5.2　常见的耦合形态

耦合形态	含　义	建　议	例　子
内容耦合	一个模块具有偷偷修改另一个模块内部数据的能力，因此对其内部的内容产生了耦合	避免出现	一个类通过公开方法将内部数组引用提供给另一个类，使其具有修改数组内容的能力
共用耦合	多个模块共同访问同一个全局变量，造成这些模块之间以及与定义全局变量的模块之间的耦合	严格限制使用	一个类声明了 public static 变量(属性)，多个其他类访问该变量。此时这个变量声明的改变或者任何一个类对于该变量的操作方式发生变化都有可能影响其他依赖于该变量的类
控制耦合	一个过程(方法或函数)通过传入“标识位”或“命令”的方式调用另一个过程并对其执行过程进行了相应控制	尽量利用多态机制消除	一个学生注册方法根据传入的学生类型进行判断，然后分别对本科生和研究生执行不同的注册过程
印记耦合	一个应用类(即实现特定应用逻辑的类)被用作一个方法的参数类型，该方法由此具有访问类的全部公开方法的能力	尽量将方法参数缩小为接口、抽象类或简单数据项	一个发送邮件的方法使用一个用户类作为参数向其发送邮件通知，而事实上该方法只需要利用用户类中的邮箱信息
数据耦合	一个方法具有多个基本数据类型或简单类(如 String)参数，导致调用该方法的类需要准备很多数据	尽量减少参数数量	一个学生综合信息打印方法要求学生学号、姓名、院系、年级、学分数、课程成绩列表等十多个不同属性

续表

内聚形态	含　义	建　议	例　子
过程调用	一个过程(方法或函数)调用另一个过程	将重复出现的调用序列封装成高层过程	一个方法依次调用分别打印表头、表格内容、表尾的三个方法来完成一个完整报表的打印
类型使用	一个类使用另一个类作为属性、方法参数或局部参数的类型	尽量将所使用的类型缩小为接口、抽象类	一个订单类包含一个类型为用户类的属性,表示订单对应的用户
文件包含和包引入	一个模块通过 include 语句引入一个文件或通过 import 语句引入一个包	消除不必要的包含和引入	一个报表打印类通过 import 语句引入打印机接口相关的包实现打印操作
外部依赖	一个模块对当前软件系统范围之外的系统(例如操作系统)产生依赖	尽量减少代码中的外部依赖	一个类通过 Java 本地化接口(Java Native Interface)调用 Windows 系统上的 DLL 文件中所实现的功能

以上这些耦合形态中的前三种,即内容耦合、共用耦合、控制耦合,需要特别引起注意。

其中,内容耦合一般都应该避免出现。内容耦合的定义中所提到的"偷偷修改"是指这种修改方式是类的定义者所没有意识到也不希望发生的。例如,图 5.8 左边的设计方案中就隐含着内容耦合。消除内容耦合的主要手段是实现信息隐藏。正如图 5.8 右边的设计方案那样,通过将内部数据隐藏起来并通过受控的接口对外提供访问,其他类除了按照预定的方式访问数据外不再具有修改其内部数据的能力了。

共用耦合是由于共享全局变量的使用而产生的。全局变量在结构化程序(例如 C 语言程序)中使用较多,在面向对象程序(例如 Java 程序)中使用较少(应当尽量避免使用 public static 变量)。共用耦合的危害主要在于其影响面、隐藏性和不可控性。共享同一全局变量的模块可能很多,因此影响面可能很大。同时,共享同一全局变量的多个模块之间的相互影响不像调用关系那样明显,因此经常被忽略。此外,全局变量的使用方式缺少必要的控制(对比一下通过受控接口对外提供间接访问的变量),一个模块如果按照自身的逻辑对变量进行了某种赋值而其他模块的开发人员又不知情,那么就有可能导致其他模块出错。

控制耦合较为常见,但也需要特别注意并考虑是否可以利用多态机制[①]进行消除。图 5.9 左边的方法显示了一个控制耦合的例子。在这个例子中,drawFigure 方法根据传入的图形数组(shapeList)中每个图形对象的类型决定如何对其进行绘制,其内部控制结构受制于外部传入对象的信息。这种控制耦合带来的问题是,drawFigure 方法需要不断根据图形种类的变化而进行修改,例如,每增加一种新的图形其内部就要增加相应的条件语句进行处理。这种控制耦合可以通过引入多态(polymorphism)来消除。如图 5.9 右边的设计方案所示,通过将图形类(Shape)变成抽象类并声明图形绘制的抽象方法(draw),可以建立图形类的继承层次,即让各种具体图形(如圆形、矩形等)都成为图形类的子类并重写(Override)图形绘制方法。这样,drawFigure 方法就可以利用多态机制直接调用抽型的图形类的绘制方法,而不需要对每个图形对象的具体类型进行判断。

① 多态是一种面向对象机制,是指按照抽象类或接口来调用方法,而实际执行的则是各个具体类中的实现。

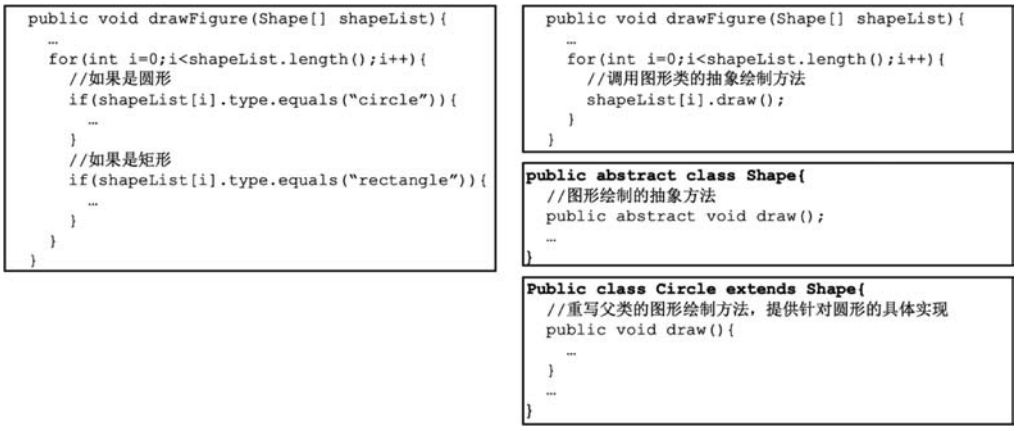

图 5.9　消除控制耦合

除了以上三种耦合形态之外，其他耦合形态（即印记耦合、数据耦合、过程调用、类型使用、文件包含和包引入、外部依赖）都属于常规的耦合形态，毕竟各种模块（例如包、类、方法）之间还是需要通过合理的协作关系构成完整的软件。对于这些常规耦合，一个基本的处理方针就是尽量降低耦合度。例如，如图 5.10 左边是一个印记耦合的例子，其中发送邮件的方法（sendEmail）将用户类（User）作为一个参数类型，从而实现向指定用户发送邮件的功能。然而，邮件发送功能只需要有邮件地址即可，并不需要用户的其他信息，因此这类的依赖和耦合有一些过多。图 5.10 右边提供了两种改进的设计方案，分别采用字符串类型（如果地址只是一个简单的字符串）和一个专门的地址类型（如果地址本身也是一个包含多种信息的对象）来取代原来的用户类。这两种改进方案都降低了原来的印记耦合度，并使得发送邮件的方法不再依赖于用户类，从而使其具有更广泛的复用性。

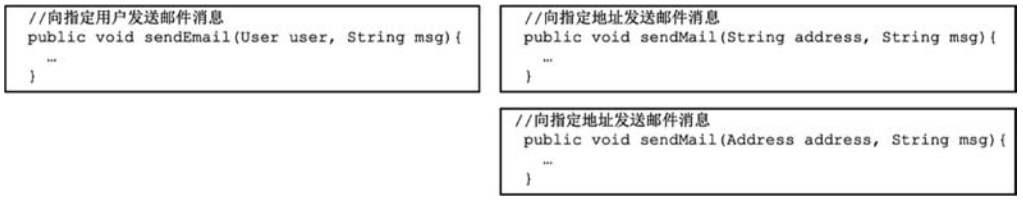

图 5.10　降低印记耦合

5.2.4　面向对象设计原则

面向对象软件设计的一个主要目标是可维护性和可扩展性，即软件设计及实现代码容易理解、修改以及扩展新功能和新特性。按照这些设计目标以及关注点分离、模块化、信息隐藏等通用设计思想，面向对象软件设计形成了一系列与面向对象设计的特点密切相关的设计原则。这些设计原则可以为面向对象设计提供具体的指导。

面向对象软件设计有一组公认的被称为 SOLID 的设计原则。这里的 SOLID 是这六种原则的英文首字母缩写的组合（其中有两个 L）。下面依次介绍这六种原则。

1. 单一职责原则

单一职责原则（Single Responsibility Principle）是指每个类、接口、方法都应该只具有

单一的职责，因此它们应该只会因为一个原因发生变化。单一职责强调的是类、接口、方法的内聚性。内聚性越高，它们的职责越单一，发生变化的原因也越集中。例如，图 5.7 右边的设计方案中计算折扣金额的方法实现了功能内聚，因此每次这个方法发生变化都应该是因为折扣金额的计算方式发生了变化。反之，图 5.7 左边的设计方案中的待支付订单金额计算方法包括订单总金额计算和折扣金额计算两部分，这两部分任何一个发生变化都有可能导致这个方法变化。当然，类与方法相比粒度更大，因此也更难实现单一职责，但可以尽量让一个类具有更少、更加聚焦的职责。

2. 开闭原则

开闭原则(Open Closed Principle)是指一个模块应该对扩展开放对修改封闭，即软件增加新功能或新特性时应当通过扩展新的代码单元(如类、方法)而非修改已有的代码单元的方式来实现。强调开闭原则的一个主要原因是理解和修改现有代码的过程中面临较大的复杂性挑战，工作量较大且容易引入代码缺陷，而扩展新的代码单元则没有这些问题。开闭原则是一个很重要也很基础的设计原则，因为它直接体现了软件设计的可扩展性的目标。

实现开闭原则的关键在于引入适当的抽象并为未来的扩展留下空间。图 5.9 中右边的改进设计方案就体现了开闭原则。图中左边的设计方案不符合开闭原则，因为每当增加一种新的图形种类，drawFigure 方法都需要增加相应的条件语句进行处理，从而导致了对已有代码的修改。而右边的改进设计方案通过引入类继承及多态机制，使得 drawFigure 方法的实现不再受到具体图形种类的影响，增加新的图形种类只需要增加相应的图形子类(即继承抽象的图形类 Shape)即可，这样就实现了对扩展开放对修改封闭。这个设计方案的 UML 类图描述如图 5.11 所示。

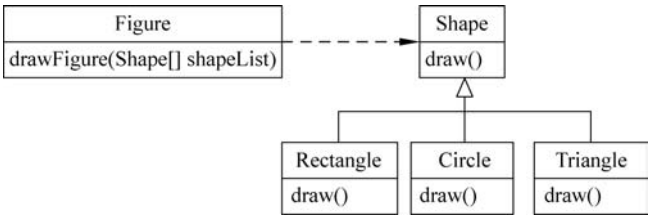

图 5.11　符合开闭原则的设计方案

3. 里氏替换原则

里氏替换原则(Liskov Substitution Principle)是指从父类派生出的子类的对象可以替换父类的对象，即子类对象可以出现在任何父类对象出现的地方。这一原则主要是为了确保子类满足父类对外的所有约定(或称为契约)，这样按照父类对外声明的行为与之交互的其他类在遇到子类的对象时才不会发生问题，而诸如多态这样的机制才能确保有效实现。例如，在如图 5.11 所示的设计方案中，如果新增加的 Shape 类的子类与 Shape 类对外的约定相违背，那么 Figure 类中按照父类 Shape 所编写的代码就有可能出错。

按照继承在概念上的含义(即表示“is a”关系)，似乎满足里氏替换原则是很自然的事情。然而，在一些继承关系中，子类除了扩展新的属性和方法之外还会对父类中已经定义的属性和方法增加额外的约束和限制，从而导致父类对外的一些约定无法满足。例如，图 5.12 中声明的正方形类(Square)继承了矩形类(Rectangle)，这一关系无疑符合概念上的继承，即正方

形是一种矩形。按照继承关系，正方形类获得了矩形类中所定义的宽度(width)、高度(height)等属性及相关方法，同时也增加了一个隐含的约束，即宽度和高度相等。这样，在一些宽度和高度设置为不同数值(例如 5 和 4)的地方，矩形对象可以出现而正方形对象则不能，否则对象内部的状态(例如正方形边长到底是 5 还是 4)以及行为(例如计算面积的结果到底是 25、16 还是 20)都会不确定。

因此，在设计继承关系时需要按照里氏替换原则验证父类与子类之间的关系，如果不符合则应该避免使用继承关系。例如，在如图 5.12 所示的设计中可以针对正方形单独定义一个新的类，而不要继承矩形类。

```
Public class Rectangle{
  private double width;
  private double height;
  public void setWidth(double width);
  public void setHeight(double height);
  public double getArea();
  …
}
```

```
Public class Square extends Rectangle{
  …
}
```

图 5.12　不符合里氏替换原则的继承关系

4. 迪米特法则

迪米特法则(Law of Demeter)强调不要和“陌生人”说话，即只与直接“朋友”交谈。在面向对象设计中，每个类都可能有一些直接的“朋友”，包括作为其属性、方法参数或返回值的对象。这个类应当尽可能只与这些直接的“朋友”打交道，而不要再跟其他类的对象打交道，例如，在方法内部实现中再引入其他类的对象作为局部变量。

迪米特法则的出发点是限制一个类与其他类的依赖关系。如果已经通过类属性、方法参数和返回值与一些对象建立了联系，那么就尽量通过这些对象满足所需要的请求，而不要再引入新的类依赖。例如，图 5.13 左边的代码中，展示图书信息的方法(displayBook)为了获取图书作者信息而在方法内部引入了对作者类(Author)的依赖，从而违反了迪米特法则。在右边的改进方案中，图书类(Book)直接提供了获取作者姓名的方法 getAuthorName，这样展示图书信息的方法就可以利用作为参数传入的图书对象来获取作者姓名，而无须与“陌生人”作者类进行直接交互。

```
//展示一本图书的信息
public void displayBook(Book book){
  …
  //展示图书作者信息
  Author author=book.getAuthor();
  display(author.getName());
  …
}
```

```
//展示一本图书的信息
public void displayBook(Book book){
  …
  //展示图书作者信息.
  display(book.getAuthorName());
  …
}
```

图 5.13　基于迪米特法则的设计改进

5. 接口隔离原则

接口隔离原则(Interface Segregation Principle)是指多个服务于特定请求方的接口好过一个通用接口,或者说不要强迫一个类依赖它不会使用的接口。在接口设计的过程中,可能存在多个相似的接口需要考虑,此时一种思路是设计一个大而全的"万能"接口来统一满足这些相似接口的需求,就像瑞士军刀将剪刀、开瓶器、螺丝刀等工具全部实现到一起那样。

使用这样的"万能"接口虽然减少了需要分别定义的接口数量,但是也会导致使用这种接口的类被迫接受一些并不需要的操作,从而增加所实现的类与接口的耦合性以及开发人员理解接口的负担,因为接口变得更大、更复杂。例如,图 5.14 左边所定义的图形(Shape)接口看起来似乎没什么问题,其中定义的方法也都和图形相关。但是仔细分析之后可以发现,这个接口中其实包含两种图形相关的职责,即支持几何计算(如求面积)的抽象图形和支持图形绘制的可视化图形,相当于从两种不同视角看图形。混合这两种职责将导致更强的耦合和不必要的依赖。试想一个只希望做抽象几何计算的图形类(例如进行土地规划)如果实现这个接口,那么它将不得不同时实现绘制图形的相关方法。同样,一个不需要图形绘制相关功能的客户端类如果通过这个接口获取相关功能,那么它的开发者将不得不了解一个更复杂的接口和一些不相关的方法(如这里的 draw 方法)。如果实现图形绘制需要一个很大的图形库,那么就会给接口的实现方和使用方引入一个不必要的依赖。由此可见,这种"通用"接口一方面不内聚,另一方面也增加了接口实现和使用的复杂性和依赖性。

```
Public interface Shape{
  //获取边的数量
  public int getNumOfSides();
  //获取第i条边的长度
  public double getLenOfSide(int i);
  //绘制图形
  public void draw();
  //获取图形面积
  public double getArea();
}
```

```
Public interface AbstractShape{
  //获取边的数量
  public int getNumOfSides();
  //获取第i条边的长度
  public double getLenOfSide(int i);
  //获取图形面积
  public double getArea();
}
```

```
Public interface DrawableShape{
  //获取边的数量
  public int getNumOfSides();
  //获取第i条边的长度
  public double getLenOfSide(int i);
  //绘制图形
  public void draw();
}
```

图 5.14　基于接口隔离原则的设计改进

图 5.14 右边给出了按照接口隔离原则所进行的设计改进。Shape 接口被分解为两个更小的接口 AbstractShape 和 DrawableShape,分别对应面向几何计算的抽象图形和面向可视化展示的可绘制图形,将两部分职责隔离。这样,实现以及使用图形接口的开发人员可以根据需要选择其中的一个,避免引入不需要的接口内容以及相应的依赖。由于 Java 等面向对象语言支持多接口实现,因此如果一个类希望同时实现这两部分职责,那也可以同时实现这两个接口。

6. 依赖转置原则

依赖转置原则(Dependence Inversion Principle)强调应该尽量依赖于抽象(例如抽象类、接口)而非具体(例如具体的实现类)。这是因为通过抽象可以只保留关键性的部分,而不用引入无关的部分(如不需要使用的方法)。此外,依赖于抽象还有利于长期的演化和维护。一方面,抽象的东西不容易变化,即使具体的实现类发生了变化,只要所实现的抽象不变,那么依赖于抽象的其他类就不会受到影响。另一方面,抽象依赖为未来的扩展提供了方便,如果有新的实现方式,那么只要实现同样的接口,其他类的代码就可以不做任何修改或仅做少量修改就能实现扩展,这也符合开闭原则的要求。如图 5.11 所示的设计方案就体现了依赖转置原则:通过依赖于抽象的图形类而非具体的图形类,Figure 类可以不做修改就支持新的具体图形种类。

5.2.5 面向切面的编程

面向对象软件设计以类为基本单位实现对于软件的模块化分解。然而,这种单一维度的模块化并不能完全满足软件设计的需求。正如 5.1.3 节中讨论关注点分离问题时所提到的,身份认证、权限检查、日志记录等横切关注点很难被封装在单个类中,因此造成了相关职责的交织和散布。在面向对象编程基础上发展起来的面向切面的编程(Aspect-Oriented Programming,AOP)方法支持横切关注点解耦和模块化封装,为进一步改进软件设计提供了支持。

1. 基本思想

面向对象软件设计以包、类和方法这样的层次化结构实现模块化分解以及相应的数据和操作封装机制,并以关注点分类作为基本的设计思想之一。然而,这种单一维度的模块化机制经常会导致以下两个问题。

- **代码交织(code tangling)**:与多个关注点相关的实现代码混杂在一个模块中。
- **代码散布(code scattering)**:与同一个关注点相关的实现代码分散在多个模块中。

如图 5.4 所示的面向对象设计结构中,类的划分及其关系定义给出了一种模块化分解方案,但其中的图书服务、用户、支付等相关的业务类中都需要在完成特定操作后进行日志记录(logging)。虽然可以通过调用公共的日志方法来实现日志记录,但是这些类中仍然需要将影响多个类(称为"横切")的日志记录关注点与自身的业务关注点(例如图书服务、用户管理、支付等)交织在一起,同时与日志记录关注点相关的代码也散布在多个类中。

面向切面编程为这种横切关注点提供了相应的模块化机制,同时允许多个切面与基本模块化单元(例如类)之间的灵活组合和集成。面向切面编程机制如图 5.15 所示。与业务相关的基本关注点按照面向对象软件设计方法产生基于类的模块化分解结构,这部分设计独立实现后形成基本程序。而横切关注点则单独以"切面"这种新引入的模块化机制来进行模块化封装和实现。在此基础上,开发人员在基本程序上定义连接点,然后相关工具自动将切面编织到对应的连接点上并形成最终的完整程序。虽然最终基本关注点和横切关注点的实现代码通过编织实现了最终集成,但从软件设计和实现的角度看横切关注点与基本关注点实现了分离,并通过"切面"这样一种新的模块化机制进行单独的封装,因此可以更好地实现关注点分离的软件设计思想。

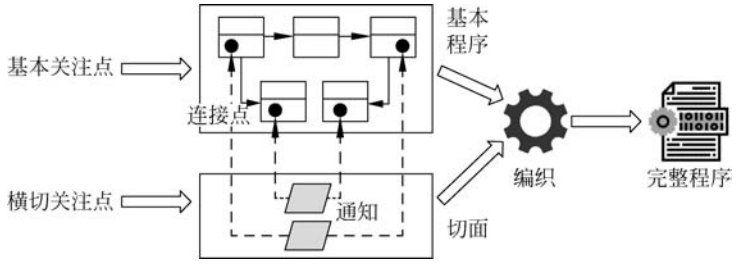

图 5.15　面向切面编程机制

2. 相关概念

面向切面编程所涉及的设计思想和实现机制建立在以下概念基础上(Filman,2006)。

- **切面(Aspect)**：切面是一种实现关注点的模块化单元。一个切面的定义包括一些实现功能逻辑的代码(称为通知)以及在哪些地方、什么时候、以何种方式调用这些代码的指令。
- **通知(Advice)**：通知是由切面所封装的、将插入到基本程序中指定的地方执行的功能代码。例如,实现日志记录的功能代码就是一个通知。通知可以被插入到程序中不同的地方,例如,目标连接点(例如某个方法调用)之前(before)、之后(after)、周围(around,即之前和之后),或者替换连接点处原有的代码(instead of)。
- **连接点(Join point)**：连接点是基本程序结构或者执行流上的一种可以明确定义的位置,在这些位置上可以插入切面所定义的新的行为。常见的连接点包括程序中的方法调用、属性的定义和访问、异常抛出等特定位置。例如,图 5.4 所示的图书馆管理子系统程序中 I_Payment 接口的 pay 方法调用就是一个连接点,可以定义一个在所有 pay 方法调用之前进行日志记录的切面。
- **切点指示器(Pointcut Designator)**：如果切面定义需要逐个枚举连接点及相应的插入位置,那么会非常麻烦而且灵活性不强。切点指示器为切面的定义提供了一种批量描述一系列连接点上的插入位置的机制,一般都是通过使用全称量词来实现。例如,可以通过切点指示器要求在某个包中所有类的公开方法被调用之前插入日志记录,或者要求实现某个接口的所有类的公开方法被调用之前插入日志记录。
- **编织(Weave)**：编织是将基本程序与切面集成到一起获得完整程序的过程。不同的面向切面编程语言的编织机制实现方式可能有所不同,既可以通过静态方法将切面代码与基本程序代码组合在一起进行编译,又可以在程序装载运行时动态地插入切面代码,还可以通过修改编译器来实现切面的插入执行。

3. 实现方式

面向切面编程思想需要借助于具体的语言或框架来实现,其中既包括定义切面(包括相关的通知实现和连接点、切入点定义等)的语言和编程规范,又包括切面编织的实现机制。当前针对 Java、C#、C++等语言都有一些较成熟的面向切面编程语言扩展及编程框架,例如,AspectJ、AspectWerkz、JBoss AOP、Spring AOP、AspectSharp、AspectC++等。下面以 Spring AOP 为例进行介绍。

Spring 是一个面向 Java 的通用软件开发框架,其中内置了 AOP 支持。Spring AOP 依

托 Spring 的核心 IOC(Inversion of Control,控制反转)容器来实现。这里的控制反转是指程序将创建对象的主动权交给 IOC 容器,自身代码中只是声明对象引用而不再创建对象,而容器则会根据所配置的规则创建对象并通过依赖注入(Dependency Injection)的方式将对象实例传入程序中。Spring AOP 通过运行时的 Java 代理机制实现切面编织。基于 Spring 框架的软件开发在第 6 章中还会详细介绍。

Spring AOP 只支持与方法调用相关的连接点,所支持的切面类型也都是与方法调用相关的,通过使用 Java 注解(annotation)机制来进行各种切面的声明,如表 5.3 所示。不同类型的切面可以实现不同的目的。例如,@Before 类型的切面的执行无法阻止目标方法的执行;@Around 类型的切面可以决定目标方法是否执行,因此可以用于权限检查(权限检查不通过则不执行目标方法);@After 类型的切面无法获得目标方法的返回值,而@AfterReturning 类型的切面可以获得返回值(可以对返回值做一些附加处理,例如记录日志),@AfterThrowing 类型的切面则可以获取所抛出的异常信息(例如在日志中记录异常)。

表 5.3　Spring AOP 所支持的切面类型

切 点 注 解	切 面 含 义
@Before	在目标方法调用之前插入通知中的实现代码
@After	在目标方法执行完成并返回前(无论正常返回或异常退出)插入通知中的实现代码
@AfterReturning	在目标方法调用正常返回之后插入通知中的实现代码
@AfterThrowing	在目标方法调用抛出异常之后插入通知中的实现代码
@Around	在目标方法调用之前和之后插入通知中的实现代码

图 5.16 展示了一个基于 Spring AOP 的切面定义的例子。这个例子中定义的切点(point 方法)表示 I_Payment 接口的 pay 方法的调用,切点表达式(@Pointcut 注解)中明确定义了该方法的完整包和类路径,同时对于参数和返回值没有任何限制,这意味着

```
@Aspect
@Component
public class PayAspectJ {

  @Pointcut("execution(* cn.setextbook.examples.I_Payment.pay(..))")
  public void point(){}

  @Before("point()")
  public void logPayment() {
    logger.info("start an online payment");
  }

  @Around("point()")
  public void checkPermission(ProceedingJoinPoint jp) {
    if(checkPermission())
      jp.proceed();
    else
      logger.info("invalid payment");
  }
}
```

图 5.16　基于 Spring AOP 的切面定义

I_Payment 接口中所有名为 pay 的方法都在覆盖范围内。在此基础上定义的@Before 类型的切面在 pay 方法执行之前插入一个日志记录方法。另一个@Around 类型的切面则在 pay 方法执行之前进行许可检查，只有检查通过才会正常执行 pay 方法（通过调用 proceed 方法），否则不执行 pay 方法并记录日志。除了使用 Java 注解进行切面定义外，Spring AOP 也支持利用配置文件进行切面的定义。

需要注意的是，这里定义的切面将通过 Spring AOP 所实现的编织机制与基本程序集成在一起。基本程序的开发人员无须了解这些切面的定义，也不需要调用这些切面中定义的功能实现。这样，原来需要分散在很多类中实现的横切关注点可以以切面的形式集中进行封装，同时与基本程序中的各个类独立并行开发，从而更好地实现了关注点分离的设计目标。

5.3 契约式设计

如前所述，面向对象设计中类的职责分配和协作关系确定是一个核心问题，也是分解和抽象原则的具体体现。这种基于类的设计分解既要使得不同类的开发者可以在明确类职责的基础上独立进行开发，又要保证不同的类分别开发完成后能够顺利集成并正确实现整体系统设计要求。因此，面向对象设计方案需要明确不同类之间的接口定义，确保不同类的开发者按照接口约定分别实现后能顺利集成。

在传统的面向对象软件设计及实现中，类之间的接口约定主要依赖于接口方法的语法声明，包括参数及其类型、返回值类型、抛出的异常等。然而，这种语法声明不足以精确定义类之间的接口语义，从而埋下了接口理解不一致的隐患。例如，图 5.17 左边所定义的计算方法根据指定操作符对传入的两个整型参数进行计算并返回结果。根据参数类型及名称，调用这个方法的开发人员很容易理解三个参数的意思，但在一些接口语义的细节上却存在模糊空间。首先，该方法支持哪些运算操作不清楚，例如，除了加减乘除是否还支持幂运算等其他操作。其次，方法的操作符参数（operator）的精确定义不清楚，例如，加减乘除分别使用 1、2、3、4 表示还是用 0、1、2、3 表示。再次，谁应该对除 0 操作进行检查不清楚，是方法的调用方进行检查把关还是方法的实现方自己进行检查。图 5.17 右边定义的整数堆栈类 IntStack 存在接口行为上的模糊性。例如，该类的出栈方法 pop 自身并没有对当前堆栈是否为空进行检查，而是根据当前的栈顶位置（topIndex）直接返回对应的元素，可以想象在堆栈为空时调用该方法会发生数组越界异常。可以看到这个类提供了一个检查堆栈是否为空的方法（isEmpty），因此可以想象这个类的开发者“假定”调用方应该先用这个方法检查一下，然后在堆栈不为空的情况下再调用出栈方法。由于这种“假定”并没有被明确定义出来，使用这个类的开发者可能会根据自己的错误理解进行编码，从而造成问题。

以上所讨论的接口定义中的模糊性除了导致不同类的开发者理解不一致之外，还会导致出错之后难以定位问题根源以及厘清责任。此外，在接口的实现方和调用方缺少明确约定的情况下，一般都会倾向于认为接口的实现方（即服务端）应当承担更多的责任对各种潜在的问题（例如不合法的参数甚至恶意的输入）进行检查，而接口的调用方（即客户端）则可以较为随意。这种做法可能导致接口实现方需要考虑的问题过多，内部逻辑过于复杂，甚至调用链上的一系列代码单元重复对某些问题进行检查。

```
//根据指定操作符进行计算并返回结果
public int calcualte(int operator,
 int a, int b){
  switch(operator){
  case 1:
    return a+b;
  case 2:
    return a-b;
  case 3:
    return a*b;
  case 4:
    return a/b;
  }
  return 0;
}
```

```
Public class IntStack{
  private int[] elements;
  private int topIndex;
  //构造方法按照指定容量构造并初始化堆栈
  public IntStack(int capability){
    elements=new int[capability];
    topIndex=-1;
  }
  //返回堆栈是否为空
  public boolean isEmpty(){
    return topIndex<0;
  }
  //执行出栈操作
  public int pop(){
    int top=elements[topIndex];
    topIndex= topIndex-1;
    return top;
  }
  …
}
```

图 5.17　接口定义中的模糊性

针对这一问题，契约式设计（Design by Contract，DBC）的思想应运而生。契约式设计的思想最早由 Bertrand Meyer 提出并在他所发明的 Eiffel 语言中进行了实现（Mitchell et al.，2003）。契约式设计，顾名思义就是要建立不同类（或者模块）之间的契约关系，以一种可检查的方式明确定义接口的实现方和调用方各自应当承担的“权利”和“义务”。“契约”在这里是一种隐喻，它将接口的实现方和调用方之间的关系比喻成商业活动中供应方和需求方之间的关系。例如，在一个采购合同中，供应方承诺交付的产品符合约定的质量要求同时按时交货，而其权利是按照约定的时间和金额获得采购款；需求方则承诺按照约定的时间和金额支付采购款，而其权利则是获得满足所约定的质量要求的产品并且能按时收货。

契约式设计中的契约应当明确定义并且可以进行验证。常见的契约形式是对程序运行状态的断言，即关于输入参数、返回值、内部属性取值范围及其关系的布尔表达式。一般的契约内容包括以下三个方面。

- **前置条件（precondition）**：调用一个方法之前应当满足的条件。
- **后置条件（postcondition）**：一个方法执行完成后应当满足的条件。
- **不变式（invariant）**：一个软件元素（例如类）在任何方法执行前后都应当一直满足的条件。

在设计契约中，前置条件是方法的调用方需要确保的，即它们只能在某个方法的前置条件满足的情况下才能对其进行调用，否则出错责任在于调用方；后置条件是方法的实现方需要确保的，即如果它们的前置条件在调用之前是满足的，那么在调用结束后它们的后置条件应该成立，否则出错责任在于被调用方；不变式则是一个软件元素所有方法的实现方都应当确保的，即任何方法的执行都不应当破坏不变式中的条件。

图 5.18 展示了一个包含契约声明的堆栈类的一部分，其中的 invariant、require、ensure 分别表示不变式、前置条件和后置条件，每个条件声明冒号之前和之后分别是条件名称和条件内容，而条件内容中的 Result 和 old 关键字分别表示方法返回值和方法执行之前的取值。

通过这些契约条件可以看到，堆栈的栈顶位置(topIndex)总是处于－1 与(数组长度－1)之间(这两个取值分别表示栈空和栈满)；创建堆栈对象时(即构造方法执行之前)所提供的容量(capability)必须是正整数，而创建成功之后内部的数组长度等于容量；查询堆栈是否为空的方法(isEmpty)的执行没有任何前置条件，而执行完之后确保返回值等于栈顶位置是否小于 0 的判断结果；出栈方法(pop)执行之前堆栈不能为空，而执行之后将返回原来栈顶的元素同时栈顶位置减 1。可以看出，这些契约条件可以帮助我们更加精确地进行接口，明确调用方和实现方的责任。例如，出栈方法的前置条件定义明确了调用方应当确保在堆栈不为空的情况下进行调用，而该方法本身无须对堆栈是否为空进行判断。

```
Public class IntStack{
  private int[] elements;
  private int topIndex;
  invariant
    topIndex_within_capability: topIndex>=-1&&topIndex<elements.length;

  public IntStack(int capability);
  require
    capability_be_positive: capability>0;
  ensure
    elements_length_equalTo_capability: elements.length==capability;

  public boolean isEmpty();
  ensure
    negative_topIndex_Indicate_Empty: Result==(topIndex<0);

  public int pop();
  require
    not_empty: isEmpty()==false;
  ensure
    return_top: Result==elements[old topIndex];
    topIndex_decrease: topIndex==old topIndex-1;
  …
}
```

图 5.18　堆栈类中的契约声明

当为一个类定义契约时需要声明哪些条件是一个需要考虑的问题。对于同一个类，不同的人可能会定义出不同的契约，其中并不存在标准的写法。针对这一问题，可以参考如下这几条契约式设计的设计原则(Mitchell et al.，2003)。

- **区分命令和查询**。一个类包含命令和查询这两种方法，其中命令能够改变对象的状态(例如属性取值)，而查询只返回结果而不改变对象状态。例如，在如图 5.18 所示的堆栈类中，pop 是命令，而 isEmpty 是查询。通过明确区分命令和查询，一个类中的查询方法可以被安全地用于契约定义，因为这些方法的执行不会改变对象状态，而命令则不能用于契约定义。
- **区分基本查询和派生查询**。基本查询是指直接对类属性和对象状态进行查询，而派生查询则是在其他查询基础上派生定义出来的。在如图 5.18 所示的堆栈类中，topIndex 的取值判断(例如 topIndex＜0)是基本查询，而 isEmpty 是派生查询(可以在 topIndex 取值判断的基础上派生出来)。
- **针对每个派生查询，定义一个使用基本查询表示的后置条件**。例如，图 5.18 中作为派生查询的 is_empty 方法有一个利用基本查询(topIndex＜0)定义的后置。

- **为每个命令定义一个后置条件，规定每个基本查询的值**。一个类中所有基本查询的值的组合可以表示这个类的整体状态。针对每个命令定义后置条件，尽量对所有基本查询的结果进行判断，确保每个命令执行之后的条件要求能够得到完整定义。
- **为每个查询和命令定义合适的前置条件**。前置条件定义了客户端在何时以及满足什么样的条件下才能调用一个查询或命令。
- **通过不变式定义对象的恒定特性**。对象的恒定特性是指一个类的对象实例在整个生存周期中都不会变化的特性。如图 5.18 所示的堆栈类中，topIndex 的取值总是要满足一个基本的取值范围要求，这通过不变式定义进行了明确。

按照契约式设计的思想，契约不仅是一种开发人员之间的约定，而且应该能够被自动检查。也就是说，契约不仅是一种提供给开发人员阅读的文本说明，而且也应该成为一种在运行时自动检查的约束。为此，一些编程语言为契约式设计提供了内置的支持（例如 Bertrand Meyer 所发明的 Eiffel 语言），允许开发人员直接在代码中声明契约，同时支持契约的运行时自动检查。而像 Java 等其他一些语言则没有这种内置的契约式设计实现机制，需要通过扩展机制来实现相关支持。例如，对于 Java 程序可以使用 Contracts for Java、iContract2、Contract4J 等扩展。此外，也可以利用断言机制简单地实现类似契约的机制，例如，在每个方法开始和结束的地方利用断言分别对前置条件和后置条件进行检查。

5.4 设 计 模 式

软件设计具有很强的经验性，而设计经验的一种主要表现形式就是设计模式。有经验的软件开发人员一般都能熟练掌握一些常用的设计模式并能够结合具体的开发要求进行实例化的应用。“模式”这一概念在很多领域中都有所体现，一般用于表示某种可借鉴的参考解决方案。在软件设计领域，设计模式是指针对一类相似设计问题的通用和参考性的设计方案，一般都经过大量的实践验证，能够较好地实现相关的设计目标。

围绕面向对象软件设计存在一些得到广泛应用的通用设计模式，特别是 Gamma 等人（Gamma et al.，2007）所总结的 23 种常用的面向对象设计模式。他们定义了设计模式的四个基本元素：

- 有意义的、能揭示设计模式目的的名称。
- 关于设计模式所针对的问题域描述，解释了该模式何时适用。
- 对于设计解决方案的描述，包括各个组成部分、各自的职责以及相互之间的关系。
- 应用这种设计模式的效果，包括可能的结果以及多方面因素的权衡，可以帮助使用者决定是否使用这种模式。

以上四个方面分别对应名称、问题、解决方案、使用效果。需要注意的是，设计模式所提供的解决方案是一种抽象的解决方案，代表的是一种设计思想，在应用时需要结合具体的问题目标和上下文进行实例化。此外，设计模式的应用存在多个方面的影响，除了有利的一面之外还可能存在不利的地方，因此效果部分需要对其中涉及的多方面因素权衡进行说明。

Gamma 等人定义的 23 种面向对象设计模式可以分为三类，即创建型模式、结构型模

式、行为型模式，如表 5.4 所示。这些模式大量应用了 5.2.4 节中所介绍的各种面向对象设计原则，其中最根本的一条是开闭原则，即通过设计模式的应用使得软件设计方案能够灵活适应需求的扩展。

表 5.4　Gamma 等人定义的设计模式分类

设计模式分类	分 类 含 义	所包含的设计模式
创建型模式	与类的实例对象创建相关的设计模式，关注于对象创建过程的抽象和封装	工厂方法（Factory Method）、抽象工厂（Abstract Factory）、单例（Singleton）、建造者（Builder）、原型（Prototype）
结构型模式	与类和对象的结构组织相关的设计模式，关注于如何实现对象的组合	适配器（Adaptor）、装饰器（Decorator）、代理（Proxy）、外观（Facade）、桥接（Bridge）、组合（Composite）、享元（Flyweight）
行为型模式	与类和对象之间的交互行为和通信相关的设计模式，关注于类和对象之间的交互关系和职责分配	策略（Strategy）、模板方法（Template Method）、观察者（Observer）、迭代器（Iterator）、责任链（Chain of Responsibility）、命令（Command）、备忘录（Memento）、状态（State）、访问者（Visitor）、中介者（Mediator）、解释器（Interpreter）

下面介绍几种常用的面向对象设计模式。其中每种模式都使用 UML 类图描述其设计思想。事实上类图仅给出了设计模式的静态设计结构，在此基础上还可以进一步使用 UML 顺序图描述各个组成部分之间的交互序列。关于表 5.4 中列举的这些设计模式的详细介绍可以参考 Gamma 等人的著作（Gamma et al.，2007）。此外，JHotDraw[①] 是一个开源 Java GUI 框架，其中包含非常丰富的设计模式实例，可以作为学习参考。

1. 单例模式

在有些应用场合中，一个类只允许有一个对象实例，例如，操作系统中的任务管理器和回收站、网站的计数器、金融交易的引擎、应用程序的日志引擎等。这要求所有需要访问这个类的对象的地方都只能获取到同一个对象引用，而不能创建不同的对象。类似的要求也可以通过定义类的静态方法来实现，即所有地方都通过访问这个类的静态方法的方式获得所需要的服务，但这样就无法使用接口、继承等面向对象机制了。

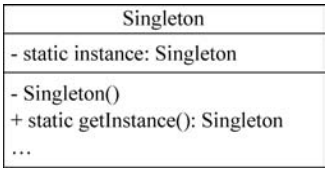

图 5.19　单例模式

单例模式为这一设计问题的解决提供了一种通用解决方案，确保了一个类只有一个对象实例并为这个对象实例提供了全局的访问入口。如图 5.19 所示，采用单例模式的类将构造方法设为私有，这样外界就无法直接创建该类的对象实例了。另一方面，该类内部包含一个类型为自身的静态成员对象（instance），同时为外界提供了一个静态方法（getInstance）用于获取这个成员对象的引用，这样就确保了外界获取到的都是同一个对象实例。

2. 适配器模式

有些时候一个类所需要的接口与另一个类所提供的接口不匹配，但功能相同或相近，此

① JHotDraw：https://sourceforge.net/projects/jhotdraw/。

时可以通过适配器在二者之间进行转换和适配。这就像不同国家的电源插座标准不一致导致所带的电器无法使用，此时需要利用转换插头将所提供的电源转换成所带的电器能够使用的插座标准。

适配器模式，顾名思义就是在客户端类所需要的接口与服务端类所提供的不匹配的接口之间进行转换和适配。如图 5.20 所示，客户端类需要的是如接口 Target 中所示的 requestA 方法，而实现相关功能的服务端类 Adaptee 提供的方法是 requestB。为了在二者之间进行适配，额外引入的适配器类（Adapter）一方面继承了 Adaptee 从而具备了该类所实现的能力，另一方面则实现了 Target 接口。这样，客户端类创建 Adapter 类的对象实例后可以将其作为接口 Target 的对象实例进行使用，而 Adapter 类内部则可以利用继承所得到的 requestB 等 Adaptee 类所提供的能力来实现所需要的 requestA 方法，从而实现转换和适配的目的。

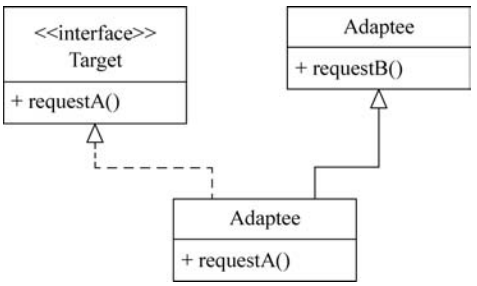

图 5.20　适配器模式

这里介绍的是使用继承实现的类的适配器。除此之外，还存在利用委托关系实现的对象适配器，即适配器 Adapter 内部包含一个 Adaptee 类型的成员对象，通过调用 Adaptee 对象的方法来实现 Target 接口所需要实现的方法。

3. 组合模式

很多时候我们需要处理的对象构成了一种表示整体部分层次的树状结构。例如，文件系统可以包含层次嵌套的目录结构，每个目录都可以包含子目录和文件，其中子目录又可以包含下一级子目录；画图工具中的图形可以形成层次嵌套的复合结构，复合图形中可以包含更小的复合图形以及基本的图形元素（如三角形、圆形）。此时，客户端代码开发人员希望可以以一种统一的方式处理对象以及对象的组合，从而避免在代码中针对不同类型的对象（复合结构或原子对象）采取不同的处理策略。例如，对于原子对象可以直接调用其操作，而对于复合结构则需要遍历其所包含的对象或下一级复合结构进行处理。

组合模式为我们提供了这样一种设计方案，一方面允许将对象组织成嵌套层次结构，另一方面允许客户端代码以统一的方式处理对象以及对象的组合。如图 5.21 所示，Leaf 表示原子对象（即层次结构中的叶子节点），而 Composite 表示复合对象，二者都是抽象的组件对象 Component 的子类。复合对象可以包含一组子对象，其中既可以有原子对象又可以有下一级的复合对象，这可以通过复合对象与抽象的组件对象之间的聚集关系（即整体部分关系）来表示。抽象的组件对象类上定义了某种操作方法 operation，原子对象可以直接给出此方法的实现，而复合对象则是进一步调用所包含的下一级原子对象或复合对象的同一操作方法来实现。采用组合模式后，客户端代码获得一个抽象的组件对象后可以直接调用其操作方法，而不必关心它是原子对象还是复合对象。

例如，在画图工具中的复合图形绘制中，抽象的组件对象是抽象的图形，原子对象是基本图形（三角形、圆形等，可以有多个），复合对象是复合图形。一个复合图形可以包含一些基本图形以及下一级的复合图形，从而构成一种层次结构。抽象图形上定义了一个绘制图形的方法 draw，每一个原子图形类可以根据自身形状实现自己的绘制方法，而复合图形则

循环调用所包含的每一个下一级图形(原子图形或复合图形)的绘制方法来实现自身的绘制。基于这种设计方案,客户端代码获取一个表示图形复合结构的根节点对象引用后,就可以按照抽象图形调用其绘制方法,从而实现整个图形复合结构的绘制,而不用关心其是一个原子图形还是复合图形。

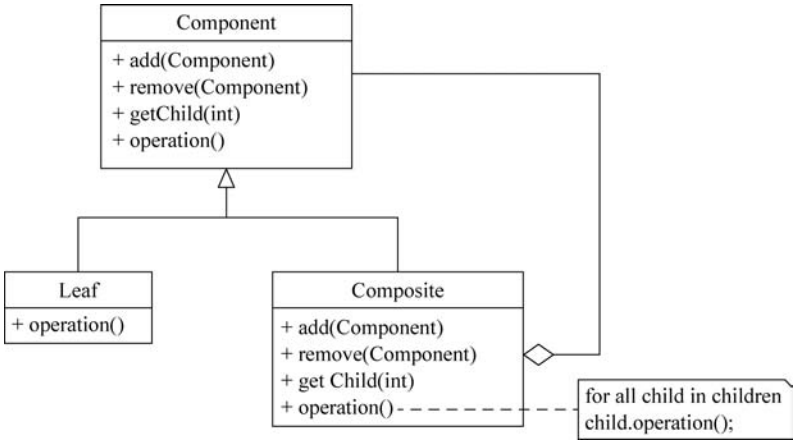

图 5.21　组合模式

4. 策略模式

一些软件功能的实现涉及算法策略选择的问题。例如,图形化软件在界面绘制和显示过程中需要使用布局算法,而其中的算法策略存在很多种不同的选择。为此,我们希望相关的功能实现与具体的算法策略之间能够实现松耦合的依赖,从而在更换算法策略或增加新的算法策略时对功能实现的影响能够最小化。

策略模式为这类问题提供了一种通用设计方案,其基本思想是对算法策略进行封装,使得算法策略与使用它们的功能代码相互独立。如图 5.22 所示,抽象的算法策略类(Strategy)对具体的算法策略进行了抽象和封装,继承该类的具体算法策略类可以提供各不相同的具体实现。另一方面,上下文类(Context)内部聚合了一个算法策略类对象,通过这个对象的算法实现(algorithm)对外提供的统一的策略方法(strategyMethod)实现相关的算法功能(例如界面布局)。这样,我们可以针对每一个算法策略开发一个 Strategy 类的子类,同时允许使用算法策略的客户端代码通过上下文类指定所需要的算法策略(通过 setStrategy 方法)并调用所需要的策略方法。

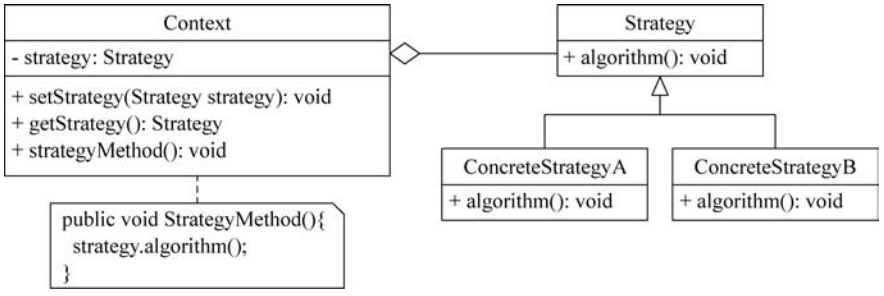

图 5.22　策略模式

策略模式突出体现了面向对象设计的开闭原则，使得算法策略的扩展变得更加容易，避免了通过多重选择语句选择不同的算法策略。同时，算法策略可以独立于客户端代码进行管理，客户端代码无须了解算法策略的实现细节。策略模式的不足是会产生很多具体策略类，需要一些额外的学习和维护开销，同时客户端代码需要明确指定使用哪种具体策略。

5. 观察者模式

在包含用户界面的应用程序中，经常存在同样的数据对象通过多种不同的形式进行可视化展示的情况。例如，关于道路交通的实时监控数据可以通过地图、表格以及各种统计图表等不同的形式来展现，而且数据的状态发生变化后各个展现视图都需要随之更新。此外，各个展现视图都有可能发生变化，未来还有可能增加新的展现方式。如果将数据管理（数据的组织和状态更新等）与数据展现混在一起，那么修改一个展现视图或者新增一个展现视图都有可能对其他已有的展现视图造成影响。按照单一职责原则以及开闭原则，我们希望数据管理与数据展现相分离，同时新增展现视图无须修改已有的代码。

观察者模式为这类问题提供了一种通用设计方案。如图 5.23 所示，负责数据管理的主题类（Subject）与负责显示逻辑的观察者（Observer）相分离，观察者通过对主题（即被观察者）的观察来实现及时的数据更新。为此，主题类中包含一个当前观察者的列表，并提供了新增（attach）和移除（detach）观察者的方法。当主题类中的数据状态发生变化时，可以调用它的通知方法（notify）来通知所有的观察者，该方法的具体实现方式是遍历观察者列表并依次调用它们的更新方法（update）。主题类和观察者类都是抽象类，在具体应用时需要定义继承自它们的具体主题类（ConcreteSubject）和具体观察者类（ConcreteObserver），其中具体观察者类的更新方法需要访问具体主题类来读取更新后的数据状态。注意，主题类依赖于抽象的观察者，而并不依赖于具体的观察者，所提供的通知方法利用多态机制调用抽象观察者的更新方法，因此新增具体观察者之后主题类和具体主题类都不需要修改。

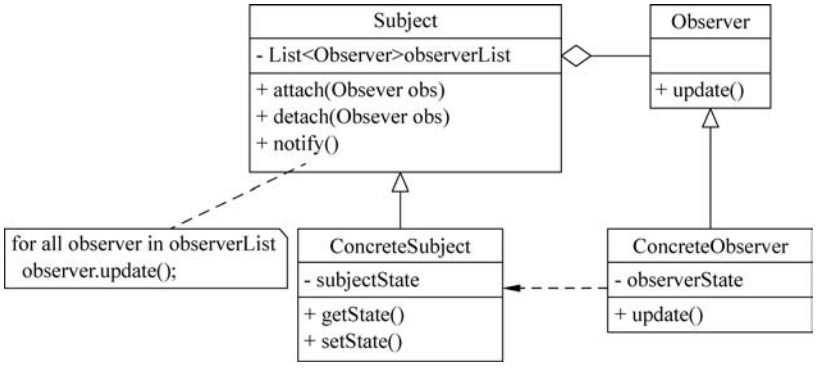

图 5.23　观察者模式

观察者模式通过主题类与具体观察者类的解耦实现了对于具体观察者类扩展的支持，但其中所引入的设计抽象也有一些不利影响。例如，主题对象的状态变化将导致所有具体观察者对象的更新操作，而其中一些更新并不是必要的，因为对应的具体观察者可能并不关注于所做的更新。

5.5 演化式设计

虽然前面介绍了很多软件设计思想、方法和原则，但是完全以事先计划的方式获得一个一直可用的“完美”设计方案却是很难的，特别是对于处于持续演化过程中的软件。在软件开发实践中，随着软件演化过程不断调整和完善软件设计方案的演化式设计一般具有更强的现实性和生命力。例如，继承结构和设计模式很多时候并不是一开始就确定的，而是随着软件演化的过程逐步引入的。这种演化式的软件设计过程往往伴随着对于各种软件设计问题（一般表现为代码坏味道）的分析以及相应的软件设计重构。

5.5.1 演化式设计与计划设计

经典软件工程信奉的是在各种工程领域中广泛采用的计划驱动的开发过程，其中的软件设计也是事先计划好的。这种计划设计（Planned Design）要求设计师在实现开始前做好高层设计并逐步细化，然后将详细的设计方案交给另一组开发者进行实现。这一过程就像建筑工程中设计师先画好图纸然后交给建筑工人进行施工。

然而，当前在软件开发实践中广泛采用的敏捷方法则强调迭代和演化式的开发过程，即通过短周期的迭代（例如一两周）不断交付可运行的软件从而持续获得用户反馈并不断调整开发计划。敏捷开发方法认为长期的计划具有很强的不确定性，因此强调以迭代化的方式制定短期计划并根据变化和反馈不断进行调整。显然，敏捷软件开发过程无法采用传统的基于事先计划的设计和实现方式，而是更倾向于采用所谓的演化式设计（Evolutionary Design），即设计随着实现过程的进展而逐步发展，而设计师的角色在一定程度上与开发者相融合。这相当于将建筑设计师的工作与建筑工人相结合，允许施工团队一边盖房子一边根据反馈持续完善和修改设计图纸。

Martin Fowler 在 *Is Design Dead*？（Fowler，2004）一文中对这两种设计方法的优缺点进行了分析和对比。

计划设计无法对软件实现过程中可能遇到的所有问题进行预见和预判，从而导致实现过程中开发者对设计产生质疑。软件设计人员如果总是忙于考虑软件设计而不参加具体的编码活动，那么软件技术的快速发展可能很快会导致软件设计人员失去编码人员的尊重。与之相对比，建筑业中设计人员和施工人员的技能界限很明确，施工人员一般不会质疑建筑设计方案，而软件开发人员很容易会优秀到足以质疑设计人员给出的设计方案的程度。此外，软件开发中的需求变更几乎总是不可避免的，这就要求软件设计必须要有灵活性，而这对于事先计划的设计是一个很大的挑战。

演化式设计中设计师和开发者角色的融合以及迭代式的设计方案演化调整很大程度上可以解决以上这些问题。然而，演化式设计不提倡太多的预先设计，而是鼓励在迭代式的演化过程中逐步考虑相关设计问题，这使得设计决策似乎主要来自开发过程中一堆即兴的决定。预先设计的缺乏有可能会使软件越来越难以应对变化，而整个软件开发似乎回到了很久以前那种“写了再改”（Code and fix）的低效模式，从而导致修复 Bug 的成本随着时间的推移越来越高。

Martin Fowler 认为这两种软件设计方法的问题都与软件变化曲线相关，即随着软件项目的进展修改软件的成本呈指数级增长（即时间越长修改越困难）（Fowler，2004）。计划设

计由于预见性的不足以及需求的变化而逐渐无法适应需要，导致软件修改越来越难；演化式设计由于缺乏事先的设计而使得软件无法为变化做好准备。为此，Martin Fowler 建议将持续集成、测试、重构相结合，实现“抚平变化曲线”的目标。其中，持续集成使得整个团队中不同开发人员的工作可以持续保持同步，使得开发人员不用担心个人所做的修改与其他开发人员的代码出现集成问题；测试可以及时暴露代码中的问题并提供反馈，同时为开发团队创造一种安全感（例如通过回归测试确认软件修改没有对其他部分造成影响）；而持续和系统性的重构则可以不断将设计思考引入软件中。

事实上，Martin Fowler 所建议的持续集成、测试、重构这三种实践已经成为当前敏捷开发实践的重要组成部分，也就是说，当前的敏捷开发实践已经可以在很大程度上实现“抚平变化曲线”的目标。在此基础上，就可以考虑将计划设计与演化式设计相结合，并找到一个合适的平衡点。一般而言，在软件编码开始之前应当做少量的预先设计，对于一些相对稳定同时涉及不同模块开发者之间约定的设计进行决策，同时将更多的设计决策留给后续开发迭代。这样在每次迭代中，开发人员在开始针对特定开发任务编码之前可以考虑是否需要通过重构引入一些设计上的考虑。

5.5.2 代码坏味道

软件重构是实现演化式设计的关键，也是在迭代化开发过程中逐步引入设计决策的重要手段。这样一来，确定软件重构的时机，即何时以及在何处开展重构，就成为一个关键性的问题。为此，Martin Fowler 和 Kent Beck 提出利用代码坏味道（bad smell）这一概念形容软件重构的时机（Fowler，2010）。他们提到：“我们看过很多很多代码，它们所属的项目从大获成功到奄奄一息的都有。观察这些代码时，我们学会了从中找寻某些特定结构，这些结构指出（有时甚至就像尖叫呼喊）重构的可能性。”这里所说到的指示着软件重构机会的特定结构就是所谓的代码坏味道。了解常见的代码坏味道可以帮助我们及时发现代码中潜在的设计问题并考虑相应的重构机会。

Martin Fowler 在他的经典著作《重构：改善既有代码的设计》（Fowler，2010）中定义了22 种常见的代码坏味道。表 5.5 中列举了其中一些典型代表，包括代码坏味道的名称、含义和重构建议。可以看出，许多代码坏味道都是由于没有很好地遵循面向对象设计思想和原则而造成的，特别是高内聚、低耦合的模块独立性原则。其中一些坏味道之间存在联系，需要加以区分。例如，发散式变化和霰弹式修改代表着相反的两个方向：前者是指同一个类由于不同的原因而发生变化，而后者则是指同一个变化导致不同的类都要进行修改。此外，需要注意的是，每种代码坏味道的评判尺度都可能包含一些主观因素。同时，每种代码坏味道对应的重构建议本身也包含着多种因素的权衡，改进一个问题的同时有可能引入其他问题，因此需要根据实际情况进行综合决策。

表 5.5 典型的代码坏味道

名　称	含　义	重构建议
重复代码 (Duplicated Code)	程序中不同的地方存在相同或相似的代码	将相似或相同的代码抽取成新的方法或类，然后将原来包含它们的地方改为对所抽取代码的调用

续表

名　　称	含　　义	重 构 建 议
过长的方法 (Long Method)	方法包含的代码过长	将方法中各个相对独立的部分抽取成多个新的方法,在原方法中调用这些新方法
过大的类 (Large Class)	一个类所承担的职责过多,例如其中包含过多的对象属性	将类中各组密切相关的对象属性抽取到多个新的类中
过长参数列表 (Long Parameter List)	一个方法所要求的传入参数过多	通过调用其他方法获得部分参数,将部分参数替换为可以提供相关信息的对象,或者将部分参数封装为新的参数对象
发散式变化 (Divergent Change)	一个类经常因为多种不同的原因在不同的方向上发生变化	将类中归属不同关注点的部分抽取成不同的类
霰弹式修改 (Shotgun Surgery)	为了实现一个变化(如需求变更)需要在许多类中进行修改	将相关的属性和方法从不同的类中抽取到同一个已有的类中或者创建一个新的类
依恋情结 (Feature Envy)	一个类中的方法对于其他类中的属性和方法的依赖高过自己所处的类	将这个方法移动到与之关系密切的类中。如果只是这个方法中的一部分存在这种情况,那么可以先将这部分抽取出来再移动
数据泥团 (Data Clumps)	多个数据项作为类属性或者方法参数在多个地方一起出现	将这些数据项封装成一个新的类,在它们原来出现的地方加入对这个新类的访问

5.5.3　软件重构

软件重构强调在不改变外部行为的情况下对内部设计进行改进,原因是不希望重构破坏外部已经通过测试甚至已经被用户使用的功能及性能等方面的非功能性表现。因为确保软件对于用户的可用性以及满足用户需求始终是第一位的,只有在不破坏这一前提的情况下对软件内部的设计改进才能得到接受和认可。因此,实施软件重构的首要前提是为其准备一个“防护网”,而这个角色一般都是由软件测试来扮演的。

通过测试构建软件重构的“防护网”要求我们为每个类准备自动化测试用例,并在软件重构过程中频繁运行这些测试,一旦测试不通过马上查找原因并解决然后再继续进行重构。在此过程中,自动化测试以一种客观可验证的方式给了我们“软件外部行为没有发生变化”的信心。同时,这也要求我们以“小步前进”的方式进行重构,即每次只执行一小部分重构然后马上通过测试来确认外部行为是否发生了变化。在此过程中,自动化测试环境以及测试用例的完整性是一个关键。对此需要注意的是,这种自动化测试能力是高质量以及高效的软件开发(特别是敏捷开发)自身的要求。正如 4.6 节中所介绍的,如果遵循测试驱动的开发方法(或者至少有意识地在每个类开发完成后添加测试用例)并使用 JUnit 等自动化测试框架,那么软件重构所需要的自动化测试保障很大程度上就已经具备了。

有了自动化测试作为保障之后,我们就可以开始考虑并实施软件重构了。如前所述,一般可以通过典型的代码坏味道来识别重构机会。但是需要注意的是,典型的代码坏味道列表并不一定能覆盖所有需要软件重构的迹象。原则上说,凡是我们感觉当前的实现方式为软件的进一步维护和演化制造了障碍,而且存在一些已知的参考设计方案可以解决问题,那么就可以考虑重构。这里所提到的对于进一步维护和演化的障碍经常与所谓的“技术债”(Technical Debt)相关。技术债是指软件实现中为了实现某种短期目标而做出的临时性的

技术决策(相当于欠下了“债务”),这种权宜之计导致后续软件修改需要付出额外的时间和成本(相当于支付“利息”)。事实上,许多代码坏味道都是这种技术债的具体体现。例如,为了快速实现并交付某个特性,开发人员可能会选择在已有的条件分支语句中增加新的条件分支,复制一段代码修改后用于另一个地方,或者在代码中“写死”一些变量和逻辑,这些问题可能导致长期维护上的额外负担。由此甚至还产生了所谓的“自承认技术债”(Self-Admitted Technical Debt)的概念,即开发人员自己承认技术债的存在并通过注释等方式进行说明,提示这些问题应当在未来改进和解决。因此,开发人员应当善于从当前代码中识别对于进一步维护和演化构成障碍的技术债,并考虑通过软件重构来进行消除。

为了体现软件重构经验并确保重构操作的安全性(即不改变软件外部行为),人们在大量的实践摸索基础上逐步形成了一系列典型的软件重构操作,其中每一种操作都有着明确的动机和关于具体做法的指导。Martin Fowler 在《重构:改善既有代码的设计》(Fowler,2010)一书中介绍了多种不同类型的软件重构操作,如表 5.6 所示。可以看到,不同类型的重构操作针对软件设计中的不同方面,例如,类间职责分配、数据的封装和访问、条件表达式、类的继承体系等。其中,大型重构是一种大规模的综合性重构,影响的范围较大,需要使用的基本重构操作较多。使用这些重构操作的好处是它们代表着经过广泛实践检验的成熟手段,它们的适用情形、实施过程等方面都有着清晰的定义。真实的软件重构过程可能会比较复杂,需要组合一系列重构操作,此时需要规划相关重构操作的执行顺序。

表 5.6　常用的软件重构操作

类　　型	含　　义	典型重构操作
重新组织方法	调整方法的内容和关系	抽取方法(Extract Method)、内联方法(Inline Method)、内联临时变量(Inline Temp)、以查询代替临时变量(Replace Temp with Query)
在对象之间移动特性	调整类的职责和类间关系	移动字段/方法(Move Field/Method)、抽取类(Extract Class)、内联类(Inline Class)、移除中间人(Remove Middle Man)
重新组织数据	调整数据的封装和访问方式	自封装字段(Self Encapsulate Field)、以对象代替数据值(Replace Data Value with Object)、以字段代替子类(Replace Subclass with Fields)
简化条件表达式	调整和优化代码中的条件表达式	分解条件表达式(Decompose Conditional)、移除控制标记(Remove Control Flag)、以多态代替条件表达式(Replace Conditional with Polymorphism)
简化方法调用	调整和优化对于方法的调用方式	重命名方法(Rename Method)、将查询与修改方法相分离(Separate Query from Modifier)、以工厂方法取代构造方法(Replace Constructor with Factory Method)
处理继承关系	调整和优化类之间的继承关系体系及相关的职责分布	上移字段/方法(Pull Up Field/Method)、下移字段/方法(Push Down Field/Method)、抽取子类/超类(Extract Subclass/Superclass)、抽取接口(Extract Interface)、以委托代替继承(Replace Inheritance with Delegation)
大型重构	在一定范围内开展大规模的软件重构	梳理并分解继承体系(Tease Apart Inheritance)、将过程化设计转换为对象设计(Convert Procedural Design to Objects)、抽取继承体系(Extract Hierarchy)

为了方便开发人员实施软件重构，一些集成开发环境（IDE）提供了自动化的软件重构操作。例如，IntelliJ IDEA（2021.2 版）[①]提供了以下常用的重构操作。

- 安全删除（Safe Delete）：在删除文件、类、方法、变量等内容时搜索并显示代码中使用这些待删除元素的情况，供开发人员确认是否需要进行必要的调整。
- 复制/移动（Copy/Move）：在不同目录或包之间复制子目录、包、文件或类，在不同路径之间移动包或类或者在不同类之间移动成员属性或方法。
- 抽取方法（Extract Method）：将一个方法中的代码片段抽取并移动到一个新的方法中，在原来方法中对应的代码替换为对新方法的调用。
- 抽取常量（Extract Constant）：将代码中的硬编码变量（如字符串）抽取为常量。
- 抽取字段（Extract Field）：将一个重复出现的变量抽取为类的属性字段。
- 抽取参数（Extract Parameter）：将方法中的变量抽取为方法的参数。
- 抽取/引入变量（Extract/Introduce Variable）：将方法中不容易理解或者重复出现的表达抽取为变量。
- 重命名（Rename）：对一个模块、目录、包、文件、类、方法或各种参数、变量进行重命名，并自动更新所有对它们的引用。
- 内联（Inline）：可以视为抽取重构的逆操作，其作用是将一个超类、匿名类、方法、参数、变量在它们被使用的地方展开。
- 修改签名（Change Signature）：修改一个类或方法的签名。

IntelliJ IDEA 为这些重构操作提供了自动化支持。开发人员使用这些重构操作时只需要选择需要重构的代码元素或片段并确定相关的重构选项，IntelliJ IDEA 将自动完成相应的重构操作。图 5.24 展示了一个基于 IntelliJ IDEA 的选择排序方法的重构示例。在这个例子中，开发人员选中选择排序方法中执行数组元素交换的一段代码（图 5.24(a)），然后利用抽取方法的重构功能得到初步的重构结果（图 5.24(b)），最后利用重命名重构功能对新抽取的方法实现了重命名（图 5.24(c)）。

```
public static void selectSort(int[] arr){
    for(int i=0; i<arr.length; i++){
        int min = i;
        for(int j=i+1; j<arr.length; j++)
            if(arr[j]<arr[min])
                min = j;
        if(i!=min){
            int temp = arr[i];
            arr[i] = arr[min];
            arr[min] = temp;
        }
    }
}
```

(a) 选择待抽取的代码片段

图 5.24　基于 IntelliJ IDEA 的软件重构示例（抽取方法并重命名）

① https://www.jetbrains.com/help/idea/refactoring-source-code.html。

```
public static void selectSort(int[] arr){
    for(int i=0; i<arr.length; i++){
        int min = i;
        for(int j=i+1; j<arr.length; j++)
            if(arr[j]<arr[min])
                min = j;
        if(i!=min){
            extracted (arr, i, min);
        }
    }
}

private static void extracted(int[] arr, int i, int min) {
    int temp = arr[i];
    arr[i] = arr[min];
    arr[min] = temp;
}
```

(b) 完成方法抽取

```
public static void selectSort(int[] arr){
    for(int i=0; i<arr.length; i++){
        int min = i;
        for(int j=i+1; j<arr.length; j++)
            if(arr[j]<arr[min])
                min = j;
        if(i!=min){
            swap (arr, i, min);
        }
    }
}

private static void swap(int[] arr, int i, int min) {
    int temp = arr[i];
    arr[i] = arr[min];
    arr[min] = temp;
}
```

(c) 对新抽取的方法进行重命名

图　5.24（续）

小　　结

软件设计覆盖体系结构设计、组件级详细设计等多个不同层次，扮演着软件需求与实现代码之间的桥梁角色。培养软件设计能力首先需要深刻理解与软件设计相关的一些思想，例如，分解与抽象、关注点分离、模块化、信息隐藏、重构、复用等。本章在概述软件设计的内容和思想的基础上，主要围绕面向对象设计方法介绍组件级详细设计。面向对象设计方法需要确定类、接口及其之间的关系，各个类之间应当相对独立。面向对象设计的一些基本原则可以为开发人员提供具体的设计指导，同时按照契约式设计的思想应当为不同类之间的

接口定义严格的契约。软件设计具有很强的经验性，而设计经验的一种主要表现形式就是设计模式。围绕面向对象软件设计存在一些得到广泛应用的通用设计模式，特别是 Gamma 等人所总结的 23 种常用的面向对象设计模式。面向对象软件设计以类为基本单位实现对于软件的模块化分解。然而，这种单一维度的模块化并不能完全满足软件设计的需求。在面向对象编程基础上发展起来的面向切面的编程方法支持横切关注点解耦和模块化封装，为进一步改进软件设计提供了支持。在软件开发实践中，随着软件演化过程不断调整和完善软件设计方案的演化式设计一般具有更强的现实性和生命力。这种演化式的软件设计过程往往伴随着对于各种软件设计问题（一般表现为代码坏味道）的分析以及相应的软件设计重构。