考点突破

根据考试大纲，本章要求考生掌握以下几个方面的知识点。

（1）分析模式与设计模式知识

（2）面向对象程序设计知识

（3）用C++语言实现常见的设计模式及应用程序。

（4）用Java语言实现常见的设计模式及应用程序。

从历年的考试情况来看，本章的考点主要集中于：设计模式基本概念、设计模式的分类、设计模式的特点与应用场合以及面向对象程序语言与设计模式思想的综合案例。

上一节　本书简介　下一节

考点精讲

1.什么是设计模式

设计模式是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结。使用设计模式可以提高代码复用度、让代码更容易被他人理解、保证代码可靠性。毫无疑问，设计模式于己、于他人、于系统都是有利的，设计模式使代码编制真正工程化，设计模式是软件工程的基石，如同大厦的一块块砖石一样。

2.设计模式的组成

一般来说，一个模式有4个基本成分，分别是模式名称、问题、解决方案和效果。

（1）模式名称

每个模式都有一个名字，帮助我们讨论模式和它所给出的信息。模式名称通常用来描述一个设计问题、它的解法和效果，由一到两个词组成。模式名称的产生使我们可以在更高的抽象层次上进行设计并交流设计思想。

（2）问题

问题告诉我们什么时候要使用设计模式、解释问题及其背景。例如，MVC（Model-View-Controler，模型-视图-控制器）模式关心用户界面经常变化的问题。它可能描述诸如如何将一个算法表示成一个对象这样的特殊设计问题。在应用这个模式之前，也许还要给出一些该模式的适用条件。

（3）解决方案

解决方案描述设计的基本要素，它们的关系、各自的任务以及相互之间的合作。解决方案并不是针对某一个特殊问题而给出的。设计模式提供有关设计问题的一个抽象描述以及如何安排这些基

本要素以解决问题。一个模式就像一个可以在许多不同环境下使用的模板，抽象的描述使我们可以把该模式应用于解决许多不同的问题。

模式的解决方案部分给出了如何解决再现问题，或者更恰当地说是如何平衡与之相关的强制条件。在软件体系结构中，这样的解决方案包括两个方面。

第一，每个模式规定了一个特定的结构，即元素的一个空间配置。例如，MVC模式的描述包括以下语句：“把一个交互应用程序划分成三部分，分别是处理、输入和输出”。

第二，每个模式规定了运行期间的行为。例如，MVC模式的解决方案部分包括以下陈述：“控制器接收输入，而输入往往是鼠标移动、点击鼠标按键或键盘输入等事件。事件转换成服务请求，这些请求再发送给模型或视图”。

（4）效果

效果描述应用设计模式后的结果和权衡。比较与其他设计方法的异同，得到应用设计模式的代价和优点。对于软件设计来说，通常要考虑的是空间和时间的权衡。也会涉及到语言问题和实现问题。对于一个面向对象的设计而言，可重用性很重要，效果还包括对系统灵活性、可扩充性及可移植性的影响。明确看出这些效果有助于理解和评价设计模式。

3. 设计模式的分类

在设计模式概念提出以后，很多人把自己的一些成功设计规范成了设计模式，一时间提出了许许多多的模式，但这些模式并没有都为众人所接受，成为通用的模式。直到ErichGamma在他的博士论文中总结了一系列的设计模式，做出了开创性的工作。他用一种类似分类目录的形式将设计模式记载下来。我们称这些设计模式为设计模式目录。根据模式的目标（所做的事情），可以将它们分成创建性模式（creational）、结构性模式（structural）和行为性模式（behavioral）。创建性模式处理的是对象的创建过程，结构性模式处理的是对象/类的组合，行为性模式处理类和对象间的交互方式和任务分布。根据它们主要的应用对象，又可以分为主要应用于类的和主要应用于对象的。

表15-1是ErichGamma等人总结的23种设计模式，这些设计模式通常被称为GoF（Gang of Four，四人帮）模式。因为这些模式是在《Design Patterns: Elements of Reusable Object-Oriented Software》中正式提出的，而该书的作者是Erich Gamma、Richard Helm、Ralph Johnson 和 John Vlissides，这几位作者常被称为“四人帮”。

表15-1设计模式目录的分类

目的	设计模式	简要说明	可改变的方面
创建型	Abstract Factory 抽象工厂模式	提供一个接口，可以创建一系列相关或相互依赖的对象，而无需指定它们具体的类	产品对象族
	Builder 生成器模式	将一个复杂类的表示与其构造相分离，使得相同的构建过程能够得出不同的表示	如何建立一种组合对象
	Factory Method* 工厂方法模式	定义一个创建对象的接口，但由子类决定需要实例化哪一个类。 工厂方法使得子类实例化的过程推迟	实例化子类的对象
	Prototype 原型模式	用原型实例指定创建对象的类型，并且通过拷贝这个原型来创建新的对象	实例化类的对象
	Singleton 单子模式	保证一个类只有一个实例，并提供一个访问它的全局访问点	类的单个实例
结构型	Adapter* 适配器模式	将一个类的接口转换成用户希望得到的另一种接口。它使原本不相容的接口得以协同工作	与对象的接口
	Bridge 桥模式	将类的抽象部分和它的实现部分分离开来，使它们可以独立地变化	对象的实现
	Composite 组合模式	将对象组合成树型结构以表示“整体-部分”的层次结构，使得用户对单个对象和组合对象的使用具有一致性	对象的结构和组合
	Decorator 装饰模式	动态地给一个对象添加一些额外的职责。它提供了用子类扩展功能的一个灵活的替代，比派生一个子类更加灵活	无子类对象的责任
	Façade 外观模式	定义一个高层接口，为子系统中的一组接口提供一个一致的外观，从而简化了该子系统的使用	与子系统的接口
	Flyweight 享元模式	提供支持大量细粒度对象共享的有效方法	对象的存储代价
	Proxy 代理模式	为其他对象提供一种代理以控制这个对象的访问	如何访问对象,对象位置
行为型	Chain of Responsibility 职责链模式	通过给多个对象处理请求的机会，减少请求的发送者与接收者之间的耦合。将接收对象链接起来，在链中传递请求，直到有一个对象处理这个请求	可满足请求的对象
	Command 命令模式	将一个请求封装为一个对象，从而可用不同的请求对客户进行参数化，将请求排队或记录请求日志，支持可撤销的操作	何时及如何满足一个请求
	Interpreter* 解释器模式	给定一种语言，定义它的文法表示，并定义一个解释器，该解释器用来根据文法表示来解释语言中的句子	语言的语法和解释
	Iterator 迭代器模式	提供一种方法来顺序访问一个聚合对象中的各个元素，而不需要暴露该对象的内部表示	如何访问、遍历聚合的元素
	Mediator 中介者模式	用一个中介对象来封装一系列的对象交互。它使各对象不需要显式地相互调用，从而达到低耦合，还可以独立地改变对象间的交互	对象之间如何交互及哪些对象交互
	Memento 备忘录模式	在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态，从而可以在以后将该对象恢复到原先保存的状态	何时及哪些私有信息存储在对象之外
	Observer 观察者模式	定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并自动更新	依赖于另一对象的数量
	State 状态模式	允许一个对象在其内部状态改变时改变它的行为	对象的状态
	Strategy 策略模式	定义一系列算法，把它们一个个封装起来，并且使它们之间可互相替换，从而让算法可以独立于使用它的用户而变化	算法
	Template Method* 模板模式	定义一个操作中的算法骨架，而将一些步骤延迟到子类中，使得子类可以不改变一个算法的结构即可重新定义算法的某些特定步骤	算法的步骤
	Visitor 访问者模式	表示一个作用于某对象结构中的各元素的操作，使得在不改变各元素的类的前提下定义作用于这些元素的新操作	无需改变其类而可应用于对象的操作

其中带*为关于类的，其他是关于对象的。

在后面的章节中，我们将详细论述23种模式的特点、应用场合及UML图。

上一节　本书简介　下一节

一点一练

试题1

在面向对象软件开发过程中，采用设计模式__(1)__。

（1）A．以复用成功的设计

B．以保证程序的运行速度达到最优值

C．以减少设计过程创建的类的个数

D．允许在非面向对象程序设计语言中使用面向对象的概念

试题2

设计模式根据目的进行分类，可以分为创建型、结构型和行为型三种。其中结构型模式用于处理类和对象的组合。__(2)__模式是一种结构型模式。

（2）A．适配器（Adapter）B．命令（Command）

C．生成器（Builder）D．状态（State）

试题3

在进行面向对象设计时，采用设计模式能够__(3)__。

（3）A．复用相似问题的相同解决方案 B．改善代码的平台可移植性

C．改善代码的可理解性 D．增强软件的易安装性

试题4

设计模式具有__(4)__的优点。

（3）A．适应需求变化 B．程序易于理解

C．减少开发过程中的代码开发工作量 D．简化软件系统的设计

上一节　本书简介　下一节

解析与答案

试题1分析

模式是一种问题的解决思路，它已经适用于一个实践环境，并且可以适用于其它环境。设计模式通常是对于某一类软件设计问题的可重用的解决方案，将设计模式引入软件设计和开发过程，其

目的就在于要重用成功的软件开发经验。

试题1答案

（1）A

试题2分析

本题考查设计模式的分类，相关分类情况请参看表15-1。

试题2答案

（2）A

试题3分析

设计模式是一种指导，在一个良好的指导下，有助于完成任务，有助于作出一个优良的设计方案，达到事半功倍的效果，而且会得到解决问题的最佳办法。采用设计模式能够复用相似问题的相同解决方案，加快设计的速度，提高了一致性。

试题3答案

（3）A

试题4分析

设计模式是用一种固定的解决方案来解决某一类问题，这种方式第一大优点是方案出错的可能性很小，因为这些方案都是经过很多人实践总结出来的；第二是适应需求变化，

试题4答案

（4）A

上一节　本书简介　下一节

常用设计模式详解

前面已经介绍了设计模式的基本概念，在本节，我们将详细分析23种模式，以及知名度非常高的MVC。对这些模式，需要了解其特点、应用场合，并需要清楚其工作原理。

上一节　本书简介　下一节

考点精讲

1. 简单工厂模式

简单工厂（Simple Factory）专门定义一个类的实例，被创建的实例通常都具有共同的父类。简单工厂模式又称为静态工厂方法（Static Factory Method）模式，属于创建型模式，通常它根据自变量的不同返回不同类的实例。

简单工厂模式的实质是由一个工厂类根据传入的参量，动态决定应该创建出哪一个产品类的实例。简单工厂模式实际上不属于GoF的23个模式，但它可以作为GoF的工厂方法模式（Factory Method）的一个引导。其UML类模型如图15-1所示，其涉及的参与者有三个：工厂角色、抽象产品角色和具体产品角色。

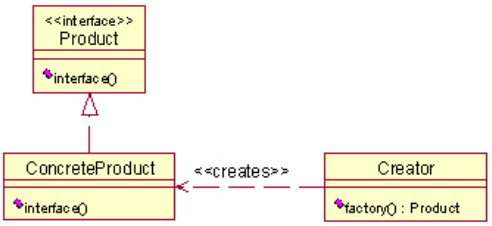

图15-1简单工厂模式的UML类模型

工厂（Creator）角色：是简单工厂模式的核心，它负责实现创建所有实例的内部逻辑。工厂类可以被外界直接调用，创建所需的产品对象。

抽象产品（Product）角色：是简单工厂模式所创建的所有对象的父类，它负责描述所有实例所共有的公共接口。

具体产品（Concrete Product）角色：是简单工厂模式的创建目标，所创建的对象都是充当这个角色的某个具体类的实例。

2. 工厂方法模式

工厂模式（Factory）又称为工厂方法模式，也叫虚拟构造函数（Virtual Constructor）模式或者多态工厂模式。在工厂模式中，父类负责定义创建对象的公共接口，而子类则负责生成具体的对象，这样做的目的是将类的实例化操作延迟到子类中完成，即由子类来决定究竟应该实例化哪一个类。

在简单工厂模式中，一个工厂类处于对产品类进行实例化的中心位置，它知道每个产品类的细节，并决定何时哪一个产品类应当被实例化。简单工厂模式的优点是能够使客户端独立于产品的创建过程，并且在系统中引入新产品时无须对客户端进行修改，缺点是当有新产品要加入到系统中时，必须修改工厂类，以加入必要的处理逻辑。简单工厂模式的致命弱点就是处于核心地位的工厂类，因为一旦它无法确定要对哪个类进行实例化，就无法使用该模式，而工厂方法模式则可以很好地解决这一问题。工厂方法模式的UML类图的如图15-2所示。

其中的类或对象之间的关系为：

产品（Product）角色：定义产品的接口。

真实产品（ConcreteProduct）角色：实现接口Product的类。

工厂（Creator）角色：声明工厂方法（FactoryMethod），返回一个产品。

真实工厂（ConcreteCreator）角色：实现工厂方法（FactoryMethod），则客户调用，返回一个产品的实例。

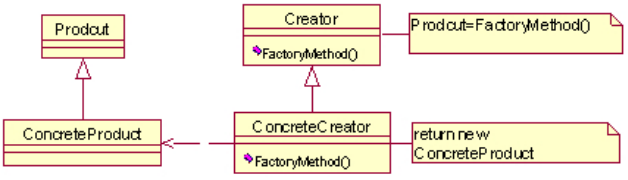

图15-2工厂模式的UML类模型

3. 抽象工厂

抽象工厂提供了一个创建一系列相关或相互依赖对象的接口，而无须指定它们具体的类。抽象工厂（Abstract Factory）模式又称为Kit模式，属于对象创建型模式。

抽象工厂模式与工厂模式最大的区别在于：工厂模式针对的是一个产品的等级结构，而抽象工厂模式则针对的是多个产品等级结构。正因为如此，在抽象工厂模式中经常会用到产品族这一概念，它指的是位于不同的产品等级结构中，并且功能相互关联的产品系列。其UML类模型如图15-3所示。其中的类或对象之间的关系为：

抽象工厂（AbstractFactory）：声明生成抽象产品的方法。

具体工厂（ConcreteFactory）：执行生成抽象产品的方法，生成一具体的产品。

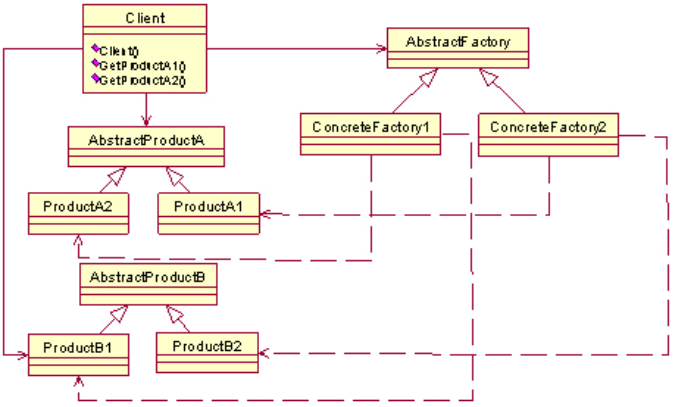

图15-3抽象工厂模式的UML类模型

4. 单例模式

单例（Singleton）模式确保其一个类只有一个实例，而且自行实例化并向整个系统提供这个实例。这个类称为单例类，它提供全局访问的方法。单例模式的要点有三个：一是某个类只能有一个实例；二是它必须自选创建这个实例；三是它必须自行向整个系统提供这个实例。UML类模型15-4所示。

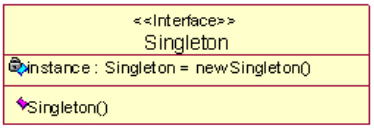

图15-4单例模式的UML类模型

类和对象之间的关系为：

单例（Singleton）：提供一个instance方法，让客户可以使用它的唯一实例。内部实现只生成一个实例。

5. **构建模式**

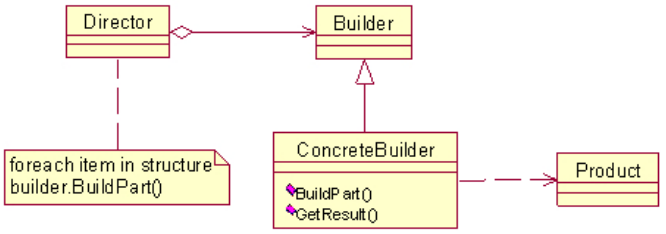

图15-5构建模式的UML类模型

构建（Builder）模式将把一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。构建模式是一步步创建一个复杂的对象，它允许用户只通过指定复杂对象的类型和内容就可以构建它们，用户不知道内部的具体构建细节。UML类模型如图8-7所示。其中的类或对象之

间的关系为：

抽象构建者（Builder）：为创建一个Product对象的各个部件指定抽象接口。

具体构建者（ConcreteBuilder）：实现Builder接口，构造和装配产品的各个部件；定义并明确它所创建的表示；提供一个返回这个产品的接口。

指挥者（Director）：构建一个使用Builder接口的对象。

产品（Product）：被构建的复杂对象，具体构建者创建该产品的内部表示并定义它的装配过程；包含定义组成部件的类，包括将这些部件装配成最终产品的接口。

6. 原型模式

原型（Prototype）模式指定创建对象的种类，并且通过复制这些原型创建新的对象。原形模式允许一个对象再创建另一个可定制的对象，根本无须知道任何创建的细节。工作原理是：通过将一个原型对象传给那个要发动创建的对象，这个要发动创建的对象通过请求原型对象复制自己来实施创建过程。UML类模型如图15-6所示。

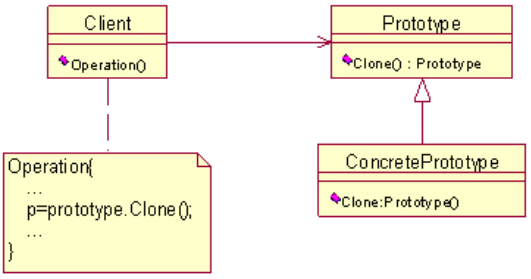

图15-6原型模式的UML类模型

其中类和对象之间的关系为：

抽象原型（Prototype）类：定义具有克隆自己的方法的接口。

具体原型（ConcretePrototype）类：实现具体的克隆方法。

客户（Client）类：通过克隆生成一个新的对象。

7. 适配器模式

适配器（Adapter）模式将一个接口转换成为客户想要的另一个接口，适配器模式使接口不兼容的那些类可以一起工作。其UML类模型如图15-7所示。

其中类的定义及其关系如下：

目标抽象（Target）类：定义客户要用的特定领域的接口。

适配器公接口（Adapter）：调用另一个接口，作为一个转换器。

适配器母接口（Adaptee）：Adapter需要接入。

客户调用（Client）类：协同对象符合Adapter适配器（公接口）。

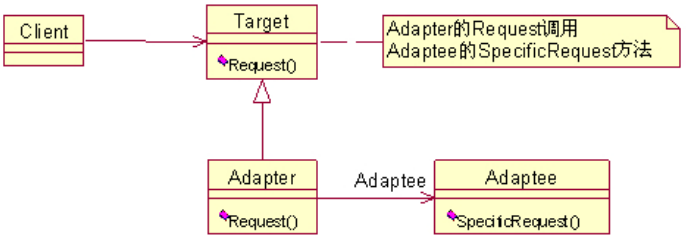

图15-7适配器模式的UML类模型

8. 合成模式

合成（Composite）模式组合多个对象形成树型结构以表示整体—部分的结构层次。合成模式

对单个对象和合成对象的使用具有一致性。其UML类模型如图15-8所示。

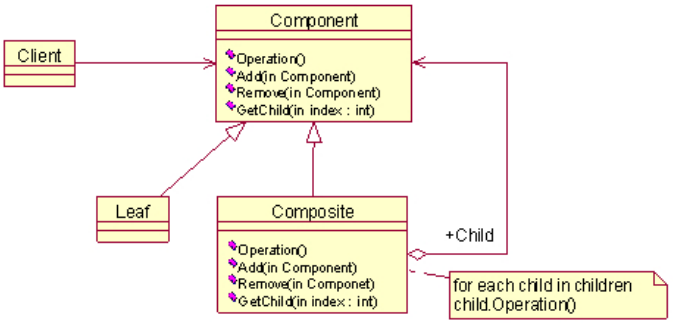

图15-8合成模式的UML类模型

其中类和对象的关系如下：

部件抽象接口（Component）：为合成的对象声明接口；某些情况下，实现从此接口派生出所有类共有的默认行为；定义一个接口可以访问及管理它的多个部分（GetChild）；如果必要也可以在递归结构中定义一个接口访问它的父节点，并且实现它。

叶子部件（Leaf）：在合成中表示叶节点对象，叶节点没有子节点；定义合成中原接口对象的行为。

合成（Composite）类：定义有子节点（子部件）的部件的行为；存储子节点（子部件）；在Component接口中实现与子部件相关的操作。

客户（Client）应用程序：通过Component接口控制组合部分的对象。

9. 装饰模式

装饰（Decorator）模式动态地给一个对象增加其他职责，就增加对象功能来说，装饰模式比生成子类实现更为灵活。UML类模型如图15-9所示。

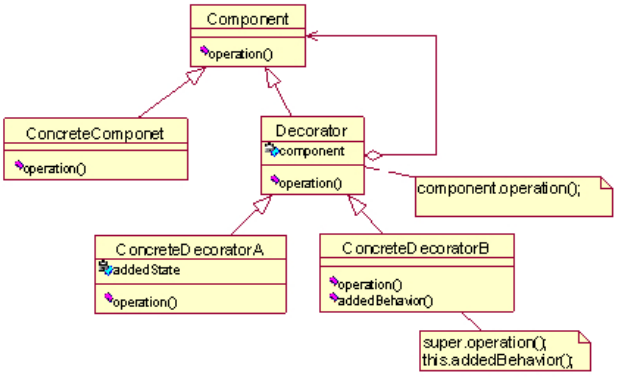

图15-9 装饰模式的UML类模型

其中类和对象的关系为：

部件（Component）：定义对象的接口，可以给这些对象动态增加职责（方法）。

具体部件（Concrete Component）：定义具体的对象，Decorator可以给它增加额外的职责（方法）。

装饰抽象（Decorator）类：维护一个内有的Component，并且定义一个与Component接口一致的接口。

具体装饰对象（ConcreteDecorator）：为内在的具体部件对象增加具体的职责（方法）。

10. 代理模式

代理（Proxy）模式为其他对象提供一个代理或地方以控制对这个对象的访问。当客户向代理对象第一次提出请求时，将实例化为真实的对象，并且将请求传给它，以后所有的客户请求都经由

proxy传给封装了的真实对象。其UML类模型如图15-10所示。

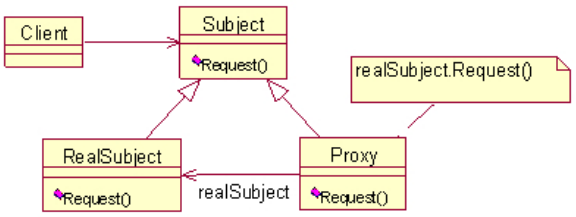

图15-10代理模式的UML类模型

其中的类和对象的关系为：

代理（Proxy）：维护一个引用使得代理可以访问实体，如果RealSubject和Subject的接口相同，Proxy会引用Subject；提供一个与Subject的接口相同的接口，使得代理可以用来替代实体，这与适配器模式类似；控制对实体的访问并且负责创建及删除它。

抽象实体（Subject）接口：为RealSubject实体及Proxy代理定义相同的接口中，使得RealSubject在任何地方都可以使用Proxy来访问。

实体（RealSubject）：定义Proxy代理的实体。

11. 享元模式

享元（Flyweight）模式运用共享技术有效地支持大量细粒度的对象。系统只使用少量的对象，而这些对象都相近、状态变化很小、对象使用次数较多。其UML类模型如图15-11所示。

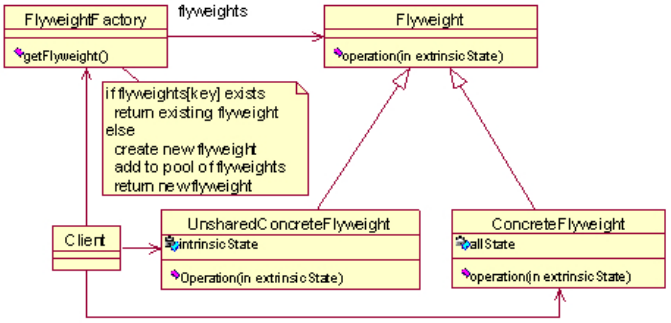

图15-11享元模式的UML类模型

其中类和对象的关系为：

享元（Flyweight）类：声明一个接口，通过它可以接收外来的参数（状态），并对新状态做出处理（作用）。

具体享元（ConcreteFlyweight）类：实现Flyweight的接口，并为内部增加存储空间。ConcreteFlyweight对象必须是可以共享的，它所存储的状态必须是内部的，即它独立存在于自己的环境中。

不共享的具体享元（UnsharedConcreteFlyweight）类：不是所有的Flyweight子类都需要被共享，Flyweight的共享不是强制的。在某些Flyweight的结构层次中，UnsharedConcreteFlyweight对象通常将ConcreteFlyweight对象作为子节点。

享元类工厂（FlyweightFactory）：创建并管理flyweight对象；确保享用flyweight，当用户请求一个flyweight对象时，FlyweightFactory提供一个已创建的实例或者创建一个实例（如果不存在）。

客户应用程序（Client）：维持一个对flyweight的引用；计算或存储一个或多个flyweight的外部状态。

12. 门面模式

门面（Facade）模式也称为外观模式，提供一个统一的接口去访问多个子系统的多个不同的接口。门面模式定义了一个高层次的接口，使得子系统更容易被使用。其UML类模型如图15-12所示。

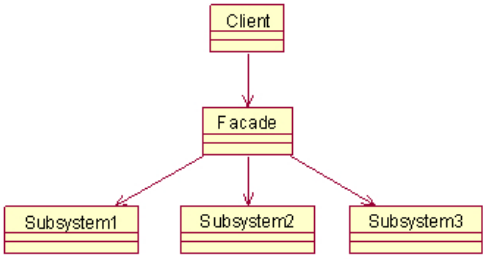

图15-12门面模式的UML类模型

其中类和对象的关系为：

门面（Facade）类：知道哪些子系统负责处理哪些请求；将客户的请求传递给相应的子系统对象处理。

子系统（Subsystem）类：实现子系统的功能；处理由Facade传过来的任务；子系统不用知道Facade，在任何地方都没有引用Facade。

13. 桥接模式

桥接（Bridge）模式将抽象部分与实现部分分离，使得它们两部分可以独立地变化。其UML类模型如图15-13所示。其中类和对象的关系为：

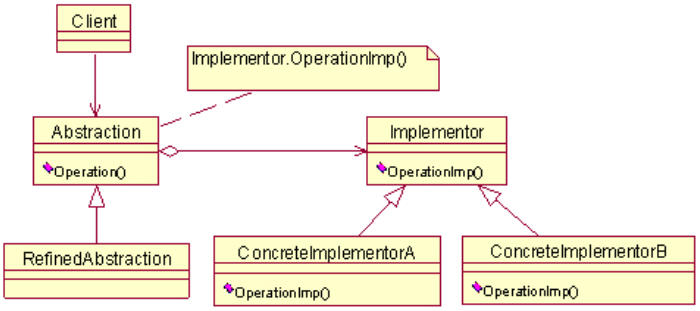

图5-13桥接模式的UML类模型

抽象（Abstraction）类：定义抽象类的接口，维护一个实现抽象（Implementor）的对象。

扩充抽象（RefinedAbstraction）类：扩充由Abstraction定义的接口。

实现（Implementor）类接口：定义实现类的接口，这个接口不一定要与Abstraction的接口完全一致，事实上这两个接口可以完全不同，一般地讲Implementor接口仅提供基本操作，而Abstraction定义的接口可能会做更多更复杂的操作。

具体实现（ConcreteImplementor）类：具体实现Implementor的接口。

14. 策略模式

策略（Strategy）模式定义一系列的算法，将每一个算法封装起来，并让它们可以相互替换。策略模式让算法独立于使用它的客户而变化。其UML类模型如图15-14所示。

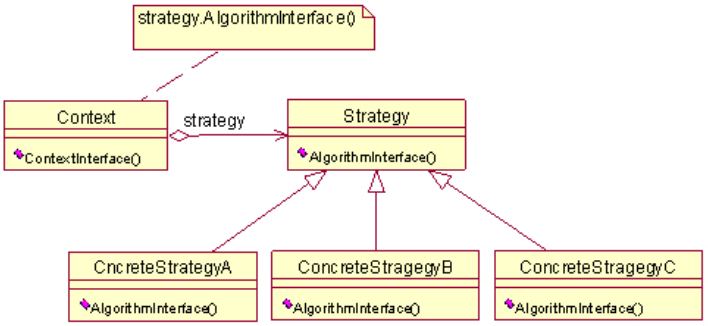

图15-14策略模式的UML类模型

其中类和对象的关系为：

抽象策略（Strategy）类：定义一个公共的接口给所有支持的算法，Context使用这个接口调用ConcreteStrategy定义的算法。

具体策略（ConcreteStrategy）类：调用Strategy接口实现具体的算法。

上下文（Context）：用ConcreteStrategy对象配置其执行环境；维护一个对Strategy的引用实例；可以定义一个接口供Strategy存取其数据。

15. 模板方法模式

模板方法（Template Method）模式定义一个操作中算法的骨架，以将一些步骤延缓到子类中实现。模板方法让子类重新定义一个算法的某些步骤而无须改变算法的结构。其UML类模型如图15-15所示。

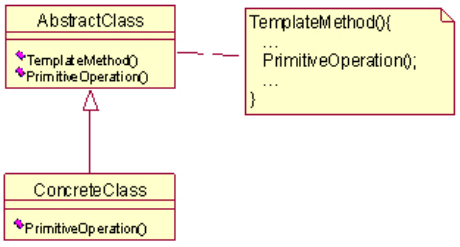

图15-15模板方法模式的UML类模型

其中类和对象的关系为：

抽象（AbstractClass）类：定义一个抽象原始的操作，其子类可以重定义实现算法的各个步骤；实现一个模板方法定义一个算法的骨架，此模板方法不仅可以调用原始的操作，还可以调用定义于AbstractClass中的方法或其他对象中的方法。

具体（ConcreteClass）类：实现原始的操作以完成子类特定算法的步骤。

16. 迭代器模式

迭代器（Iterator）模式提供一种方法可以访问聚合对象，而不用暴露这个对象的内部表示。其UML类模型如图15-16所示。

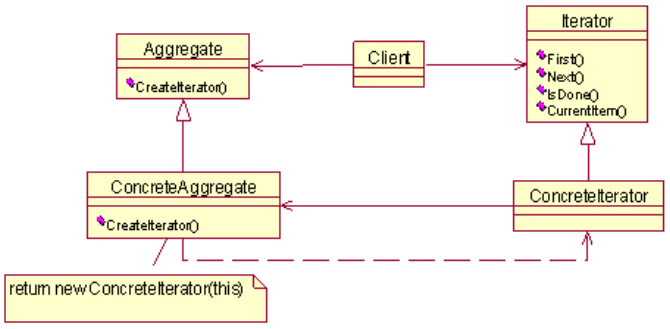

图15-16迭代器模式的UML类模型

其中类和对象的关系为：

迭代器（Iterator）：定义访问和遍历元素的接口。

具体迭代器（ConcreteIterator）：实现迭代器的接口；在遍历时跟踪当前聚合对象中的位置。

聚合（Aggregate）：定义一个创建迭代器对象的接口。

具体聚合（ConcreteAggregate）：创建迭代器对象，返回一个具体迭代器实例。

17. 责任链模式

责任链（Chain of Responsibility）模式能避免请求发送者与接收者耦合在一起，让多个对象都有可能接收接求，将这些对象连接成一条链，并且沿着这条链传递请求，直到有对象处理为止。该模式的UML类模型如图15-17所示。

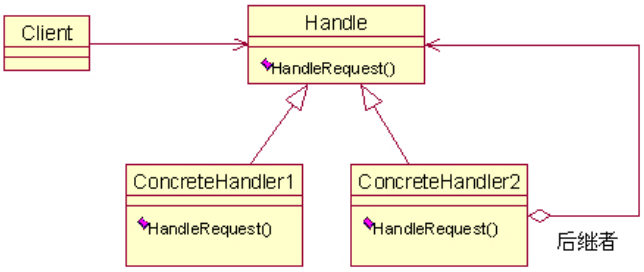

图15-17责任链模式的UML类模型

其中类和对象的关系：

传递者（Handler）接口：定义一个处理请求的接口；或实现链中下一个对象。

具体传递者（ConcreteHandler）：处理它所负责的请求；可以访问链中下一个对象；如果可以处理请求，就处理，否则将请求转发给后继者。

客户应用程序（Client）：向链中的对象提出最初的请求。

18. 命令模式

命令（Command）模式将一个请求封装成一个对象，因此可以参数化多个客户的不同请求，将请求排队，记录请求日志，并且支持撤销操作。其UML类模型如图15-18所示。

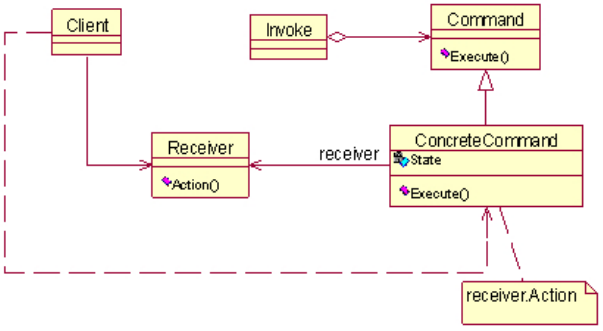

图15-18命令模式的UML类模型

其中类和对象之间的关系为：

抽象命令（Command）类：声明执行操作的一个接口。

具体命令（ConcreteCommand）类：将一个接收者对象绑定于一个动作；实现Execute方法，以调用接收者的相关操作（Action）。

客户应用程序（Client）：创建一个具体命令类的对象，并且设定它的接收者。

调用者（Invoker）：要求一个命令对象执行一个请求。

接收者（Receiver）：知识如何执行关联请求的相关操作。

19. 备忘录模式

备忘录（Memento）模式在不破坏封装的前提下，捕获并且保存一个对象的内部状态，这样可以将对象恢复到原先保存的状态。其UML类模型如图15-19所示。

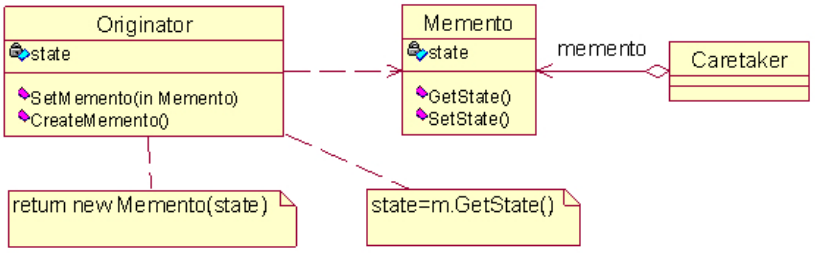

图15-19备忘录模式的UML类模型

其中类和对象的关系为：

备忘录（Memento）：保持Originator（原发器）的内部状态，根据原发器来决定保存哪些内部的状态；保护原发器之处的对象访问备忘录，备忘录可以有效地利用两个接口，看管者只能调用狭窄（功能有限）的接口，即只能传递备忘录给其他对象，而原发器可以调用一个宽阔（功能强大）的接口，通过这个接口可以访问所有需要的数据，使原发器可以返回先前的状态。理想的情况是，只允许生成本备忘录的那个原发器访问本备忘录的内部状态。

原发器（Originator）：创建一个备忘录，记录它的当前内部状态；可以利用一个备忘录来恢复它的内部状态。

看管者（Caretaker）：只负责看管备忘录；不可以对备忘录的内容操作或检查。

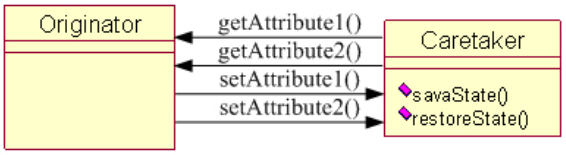

图15-20未采用备忘录模式时的UML类模型

Memento模式经常在很多应用软件出现，按Ctrl-Z会取消最后一次用户操作。如果不用Memento模式，Caretaker对象要备份Originator对象的状态，Originator就要具有所有需要访问成员的方法，当要恢复Originator对象的状态时，Caretaker更要清楚Originator内部的结构。这种严紧的凝结的结果是，发生在Originator上的任何修改，Caretaker都要作出相应的修改，这样就打破了对象封装的特点。图15-20是没有采用Memento模式时的结构。

而Memento模式是解决这种问题的最好办法，Memento类成了Originator及Caretaker的媒介，封装保存Originator的备份状态，当Originator被提出备份请求时，它就会创建一个Memento对象返回给Caretaker。Caretaker不可以看到Memento对象的内部信息，需要时Caretaker可以返回备份的Memento对象给Originator，让它恢复到备份状态。结构如图15-21所示。

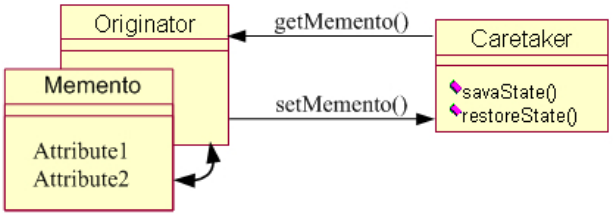

图15-21采用备忘录模式后的UML类模型

20. 状态模式

状态（State）模式能够使一个对象的内在状态改变时允许改变其行为，使这个对象看起来像是改变了其类。该模式的UML类模型如图15-22所示。

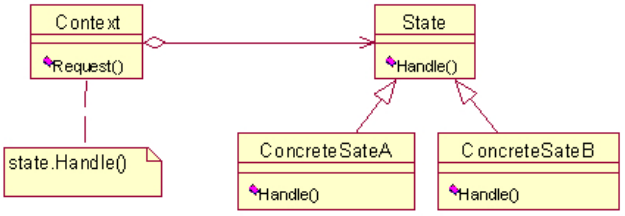

图15-22状态模式的UML类模型

其中类和对象的关系为：

上下文（Context）类：定义客户应用程序有兴趣的接口；维护一个ConcreteState（具体状态）子类的实例对象。

抽象状态（State）类：定义一个接口以封装与Context的一个特别状态（State）相关的行为。

具体状态（ConcreteState）类：每一个具体状态类实现一个Context的状态相关的行为。

21. 访问者模式

访问者（Visitor）模式说明一个操作执行于一个对象结构的成员中。访问者模式让你定义一个类的新操作而无须改变它操作的这些成员类。UML类模型如图15-23所示。

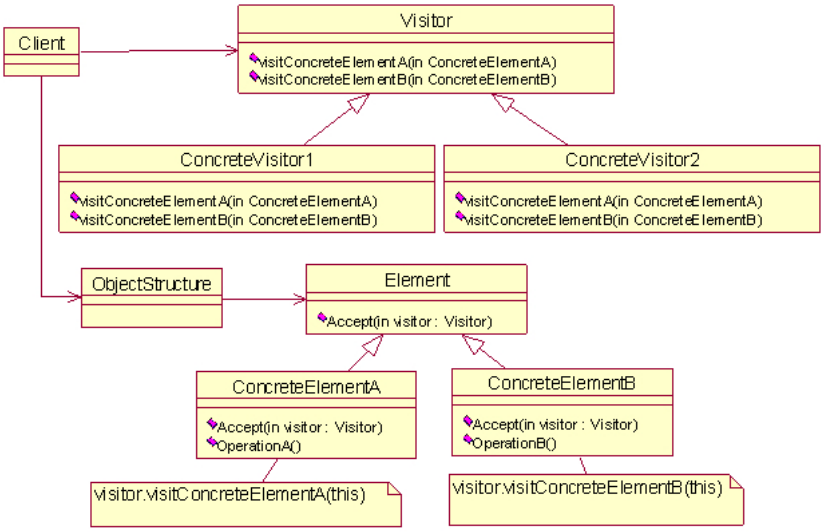

图15-23访问者模式的UML类模型

其中类和对象的关系为：

抽象访问者（Visitor）：为对象结构类中的每一个ConcreteElement的类声明一个Visit操作，这个操作的名称及标志（signature）识别传出Visit请求给访问者的类。这就使得访问者可以找到被访问的元素的具体类，也就可以直接经由其特有接口访问到元素（Element）。

具体访问者（ConcreteVisitor）：实现每个由Visitor声明的操作，每个操作实现本算法的一部分，而该算法片断乃是对应于结构中对象的类。ConcreteVisitor为该算法提供了场境并存储它的局部状态。这一状态常常在遍历该结构的过程中累积结果。

元素（Element）：定义一个Accept操作，它以一个访问者为参数。

具体元素（ConcreteElement）：实现Accept操作，该操作以访问者为参数。

对象结构（ObjectStructure）类：能枚举它的元素；可以提供一个高层的接口以允许访问者访问它的元素；可以是一个合成模式或是一个集合，如一个列表或一个无序集合。

22. 解释器模式

给出一种语言，定义这种语言的文法的一种表示，定义一个解释器（Interpreter），用它来解释使用这种语言的句子，这就是解释器模式，其UML类模型如图15-24所示。

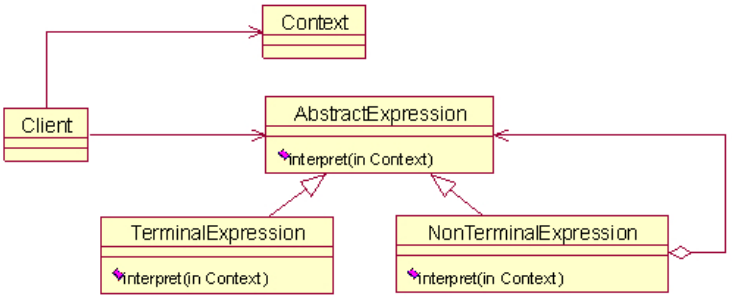

图15-24解释器模式的UML类模型

其中类和对象之间的关系为：

抽象表达式（AbstractExpression）类：定义一个接口来执行解释操作。

终结符表达式（TerminalExpression）：实现文法中关联终结符的解释操作；文句中的每个终结符都需要一个实例。

非终结符表达式（NonTerminalExpression）：文法中的每一条规则R::=R1R2...Rn都需要一个非终结符表达式类；维护每一条规则R1到Rn具有AbstractExpression接口实例；实现文法中关联非终结符的解释操作，采用递归的办法调用每一条规则R1到Rn的解释操作。

上下文（Context）：包括解释器的所有全局信息。

客户应用程序（Client）：构建由这种语言表示的句子的抽象文法树，文法树由终结符表达式或者非终结符表达式的实例组成；调用文法树中的表达式实例的解释操作。

23. 调停者模式

调停者（Mediator）模式定义一个对象封装一系列多个对象如何相互作用。Mediator使得对象之间不需要显式地相互引用，从而使其耦合更加松散。并且还让我们可以独立变化多个对象相互作用，其UML类模型如图15-25所示。其中类和对象的关系为：

抽象调停者（Mediator）：定义一个接口用于与各同事对象（Colleague）之间通信。

具体调停者（ConcreteMediator）：协调各个同事对象实现协作的行为；掌握并且维护它的各个同事对象引用。

同事（Colleague）类：每一个同事对象都引用一个调停者对象；每一个同事对象在需要和其他同事对象通信时，就与它的调停者通信。

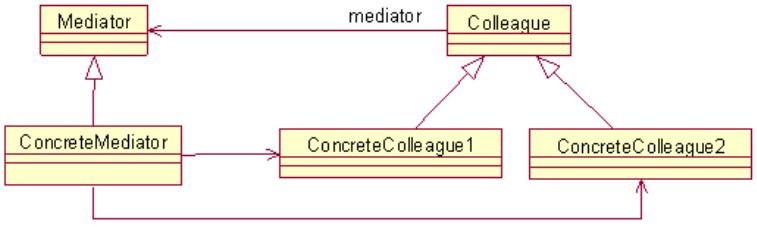

图15-25调停者模式的UML类模型

24. 观察者模式

观察者（Observer）模式定义对象间的一种一对多依赖关系，使得每当一个对象改变状态，则其相关依赖对象皆得到通知并被自动更新，其UML类模型如图15-26所示。

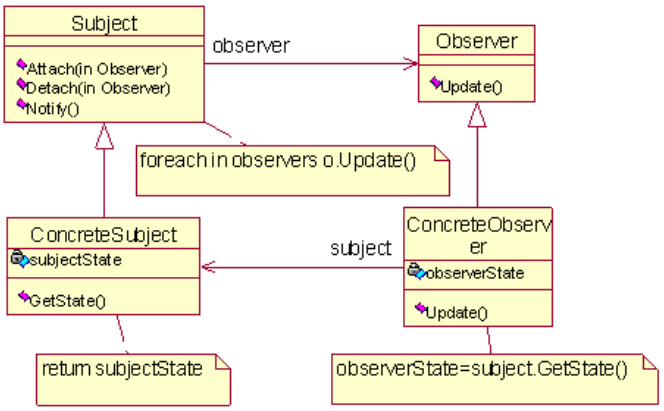

图15-26观察者模式的UML类模型

其中类和对象的关系为：

被观察对象（Subject）：了解其多个观察者，任意数量的观察者可以观察一个对象；提供一个接口用来绑定及分离观察者对象。

具体被观察对象（ConcreteSubject）：存储具体观察者（ConcreteObserver）有兴趣的状态；当其状态改变时发送一个通知给其所有的观察者对象。

观察者（Observer）：定义一个更新接口，在一个被观察对象改变时应被通知。

具体观察者（ConcreteObserver）：维护一个对ConcreteSubject对象的引用；存储状态，状态需与ConcreteSubject保持一致；实现观察者更新接口以保持其状态与ConcreteSubject对象一致。

25. MVC基础

MVC模式是“Model-View-Controller”的缩写，中文翻译为“模式-视图-控制器”。它把一个应用的输入、处理、输出流程按照Model、View、Controller的方式进行分离，这样一个应用被分成三个层——模型层、视图层、控制层。

视图（View）代表用户交互界面，对于Web应用来说，可以概括为HTML界面，但有可能为XHTML、XML和Applet。而一个应用可能有很多不同的视图，MVC设计模式对于视图的处理仅限于视图上数据的采集和处理，以及用户的请求，而不包括在视图上的业务流程的处理，业务流程的处理交予模型（Model）处理。

模型（Model）：就是业务流程/状态的处理以及业务规则的制定。业务流程的处理过程对其他层来说是黑箱操作，模型接受视图请求的数据，并返回最终的处理结果。业务模型的设计可以说是MVC最主要的核心。业务模型还有一个很重要的模型那就是数据模型。数据模型主要指实体对象的数据保存（持续化）。

控制（Controller）可以理解为从用户接收请求，将模型与视图匹配在一起，共同完成用户的请求。划分控制层的作用也很明显，它清楚地告诉你，它就是一个分发器，选择什么样的模型，选择什么样的视图，可以完成什么样的用户请求。控制层并不做任何的数据处理。

模型、视图与控制器的分离，使得一个模型可以具有多个显示视图。如果用户通过某个视图的控制器改变了模型的数据，所有其他依赖于这些数据的视图都应反映到这些变化。因此，无论何时发生了何种数据变化，控制器都会将变化通知所有的视图，导致显示的更新。这实际上是一种模型的变化-传播机制。

一点一练

试题1

设计模式中的__(1)__模式将对象组合成树形结构以表示“部分一整体”的层次结构，使得客户对单个对象和组合对象的使用具有一致性。下图为该模式的类图，其中，__(2)__定义有子部件的那些部件的行为；组合部件的对象由__(3)__通过component提供的接口操作。

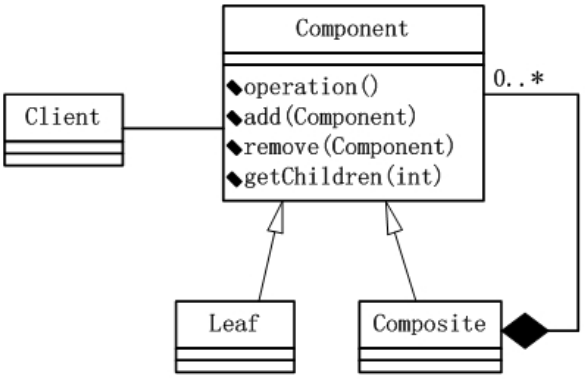

图15-27 设计模式UML类图

（1）A．代理（Proxy）B．桥接器（Bridge）

C．组合（ Composite）D．装饰器（Decorator）

（2）A．Client　　B．Component　　C．Leaf　　D．Composite

（3）A．Client　　B．Component　　C．Leaf　　D．Composite

试题2

欲动态地给一个对象添加职责，宜采用__(4)__模式。

（4）A．适配器（Adapter）B．桥接（Bridge）

C．组合（ Composite）　　D．装饰器（Decorator）

试题3

__(5)__模式通过提供与对象相同的接口来控制对这个对象的访问。

（5）A．适配器（ Adapter）　B．代理（Proxy）

C．组合（ Composite）　D．装饰器（Decorator）

试题4

__(6)__将一个类的接口转换成客户希望的另外一个接口，使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。

（6）A．Adapter（适配器）模式 B．Command（命令）模式

C．Singleton（单例）模式 D．Strategy（策略）模式

试题5

以下关于单身模式（Singleton）的描述中，正确的是__(7)__。

（7）A．它描述了只有一个方法的类的集合

B．它能够保证一个类只产生一个唯一的实例

C．它描述了只有一个属性的类的集合

D．它能够保证一个类的方法只能被一个唯一的类调用

试题6

__(8)__设计模式定义了对象间的一种一对多的依赖关系，以便当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并自动刷新。

（8）A．Adapter（适配器）B．Iterator（迭代器）

C．Prototype（原型）D．Observer（观察者）

解析与答案

试题1分析

本题考查组合设计模式。组合设计模式将对象组合成树形结构以表示“部分一整体”的层次结构，使得客户对单个对象和组合对象的使用具有一致性。

在类图中，Component为合成的对象声明接口；某些情况下，实现从此接口派生出所有类共有的默认行为，定义一个接口可以访问及管理它的多个部分（GetChild），如果必要也可以在递归结构中定义一个接口访问它的父节点，并且实现它；Leaf在合成中表示叶节点对象，叶节点没有子节点；Composite用来定义有子节点（子部件）的部件的行为，存储子节点（子部件）；Client通过Component接口控制组合部分的对象。

试题1答案

（1）C（2）D（3）A

试题2分析

本题考查设计模式的应用场合，题目中的几种设计模式在前文中已有详细描述。这几个模式的核心特点可以总结为：

适配器模式将一个接口转换成为客户想要的另一个接口，适配器模式使接口不兼容的那些类可以一起工作。

桥接模式将抽象部分与实现部分分离，使得它们两部分可以独立地变化。

组合模式组合多个对象形成树型结构以表示整体—部分的结构层次。

装饰器模式动态地给一个对象增加其他职责，就增加对象功能来说，装饰模式比生成子类实现更为灵活。

通过总结可以得知，4个备选设计模式中，装饰器模式最吻合。

试题2答案

（4）D

试题3分析

与试题2类似，本题也是考查设计模式的应用场合，关于适配器、组合、装饰器的特点前面已经介绍，不再赘述。

代理模式通过提供与对象相同的接口来控制对这个对象的访问。

试题3答案

（5）B

试题4分析

适配器模式的意图就是将一个类的接口转换成客户希望的另外一个接口，使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。

命令模式的意图是将一个请求封装为一个对象，从而使你可用不同的请求对客户进行参数化；对请求排队或记录请求日志，以及支持可撤消的操作。

单例模式的意图是确保某个类只有一个实例，且能自行实例化，并向整个系统提供这个实例。

策略模式的意图是定义一系列的算法，把它们一个个封装起来，并且使它们可相互替换，该模式使得算法可独立于使用它的客户而变化。

试题4答案

（6）A

试题5分析

此题属于纯概念题，单身模式的提出，其意图就是保证一个类仅有一个实例，并提供一个访问它的全局访问点。所以本题选B。

试题5答案

（7）B

试题6分析

本题考查内容涉及的设计模式包括：适配器、迭代器、原型、观察者。考查的正是这些模式的定义，相关知识参看本节考点精讲。

试题6答案

（8）D

上一节　本书简介　下一节

作者：希赛教育软考学院　来源：希赛网　2014年05月06日

考前冲刺

试题1

当不适合采用生成子类的方法对已有的类进行扩充时，可以采用__(1)__设计模式动态地给一个对象添加一些额外的职责；当应用程序由于使用大量的对象，造成很大的存储开销时，可以采用__(2)__设计模式运用共享技术来有效地支持大量细粒度的对象；当想使用一个已经存在的类，但其接口不符合需求时，可以采用__(3)__设计模式将该类的接口转换成我们希望的接口。

（1）A．命令（Command）B．适配器（Adapter）

C．装饰（Decorate）D．享元（Flyweight）

（2）A．命令（Command）B．适配器（Adapter）

C．装饰（Decorate）D．享元（Flyweight）

（3）A．命令（Command）B．适配器（Adapter）

C．装饰（Decorate）D．享元（Flyweight）

试题2

__(4)__设计模式允许一个对象在其状态改变时，通知依赖它的所有对象。该设计模式的类图如下图，其中，__(5)__在其状态发生改变时，向它的各个观察者发出通知。

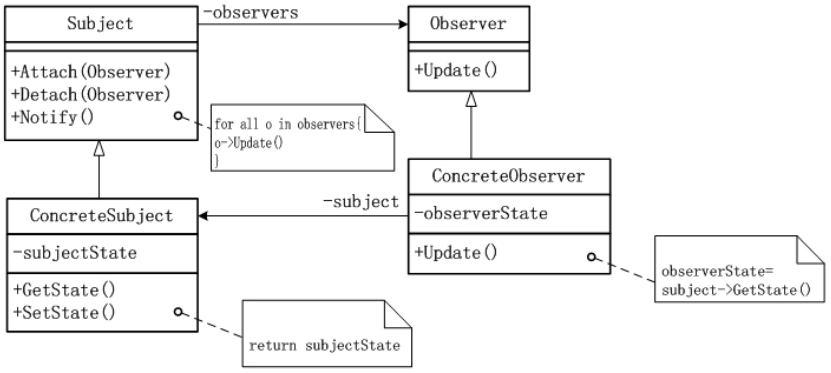

图15-28 设计模式UML类图

（4）A．命令（Command）B．责任链（Chain of Responsibility）

C．观察者（Observer）D．迭代器（Iterator）

（5）A．Subject　B．ConcreteSubject

C．Observer　D．ConcreteObserver

试题3

下面给出了4种设计模式的作用。

外观（Facade）：为子系统中的一组功能调用提供一个一致的接口，这个接口使得这个子系统更加容易使用；

装饰（Decorate）：当不能采用生成子类的方法进行扩充时，动态地给一个对象添加一些额外的功能；

单件（Singleton）：保证一个类仅有一个实例，并提供一个访问它的全局访问点；

模板方法（Template Method）：在方法中定义算法的框架，而将算法中的一些操作步骤延迟到子类中实现。

请根据下面叙述的场景选用适当的设计模式。若某面向对象系统中的某些类有且只有一个实例，那么采用__(6)__设计模式能够有效达到该目的；该系统中的某子模块需要为其他模块提供访问不同数据库系统（Oracle、SQL Server、DB2 UDB等）的功能，这些数据库系统提供的访问接口有一定的差异，但访问过程却都是相同的。例如，先连接数据库，再打开数据库，最后对数据进行查询，__(7)__设计模式可抽象出相同的数据库访问过程；系统中的文本显示类（TextView）和图片显示类（PictureView）都继承了组件类（Component），分别显示文本和图片内容。现需要构造带有滚动条，或者带有黑色边框，或者既有滚动条又有黑色边框的文本显示控件和图片显示控件，但希望最多只增加三个类，__(8)__设计模式可以实现该目的。

（6）A．外观 B．装饰 C．单件 D．模板方法

（7）A．外观 B．装饰 C．单件 D．模板方法

（8）A．外观 B．装饰 C．单件 D．模板方法

试题4

设计模式__(9)__将抽象部分与其实现部分相分离，使它们都可以独立地变化。图15-29为该设计

模式的类图，其中，__(10)__用于定义实现部分的接口。

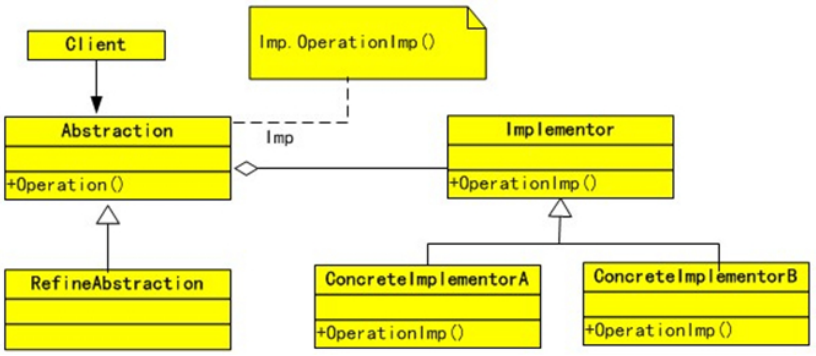

图15-29 设计模式UML类图

（9）A．Bridge（桥接）B．Composite（组合）

C．Facade（外观）D．Singleton（单例）

（10）A．Abstraction　　B．ConcreteImplementorA

C．ConcreteImplementorB　D．Implementor

试题5

以下关于Singleton（单例）模式的描述中，正确的是__(11)__。

（11）A．它描述了只有一个方法的类的集合

B．它描述了只有一个属性的类的集合

C．它能够保证一个类的方法只能被一个唯一的类调用

D．它能够保证一个类只产生唯一的一个实例

试题6

在“模型 - 视图 - 控制器”（MVC）模式中，__(12)__主要表现用户界面，__(13)__用来描述核心业务逻辑。

（12）A．视图 B．模型 C．控制器 D．视图和控制器

（13）A．视图 B．模型 C．控制器 D．视图和控制器

试题7

在面向对象软件开发过程中，采用设计模式__(14)__。

（14）A．允许在非面向对象程序设计语言中使用面向对象的概念

B．以复用成功的设计和体系结构

C．以减少设计过程创建的类的个数

D．以保证程序的运行速度达到最优值

试题8

阅读下列说明和C++代码，将应填入（n）处的字句写在答题纸的对应栏内。

【说明】

某咖啡店当卖咖啡时，可以根据顾客的要求在其中加入各种配料，咖啡店会根据所加入的配料来计算费用。咖啡店所供应的咖啡及配料的种类和价格如表15-2所示。

表15-2咖啡及配料表

咖啡	价格/杯	配料	价格/份
蒸馏咖啡（Espresso）	25	摩卡（Mocha）	10
深度烘焙咖啡（DarkRoast）	20	奶泡（Whip）	8

现采用装饰器（Decorator）模式来实现计算费用的功能，得到如图15-30所示的类图

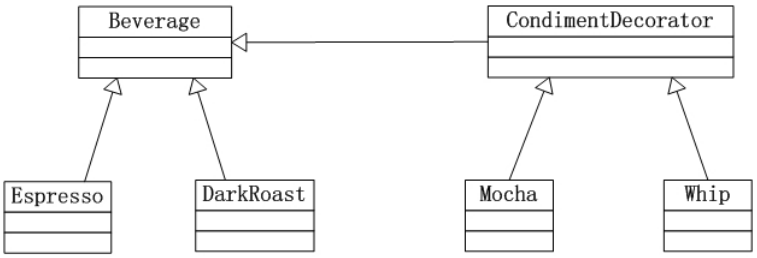

图15-30类图

【C++代码】

```
#include <iostream>
#include <string>
using namespace std;
const int ESPRESSO_PRICE = 25;
const int DRAKROAST_PRICE = 20;
const int MOCHA_PRICE = 10;
const int WHIP_PRICE = 8;
class Beverage {      //饮料
（1）:string description;
public:
（2）（){ return description; }
（3）；
};
class CondimentDecorator : public Beverage {   //配料
protected:
（4）  ;
};
class Espresso : public Beverage {      //蒸馏咖啡
public:
Espresso ( ) {description="Espresso"; }
int cost ( ){return ESPRESSO_PRICE; }
};
class DarkRoast : public Beverage {     //深度烘焙咖啡
public:
DarkRoast( ){ description = "DardRoast"; }
int cost( ){ return DRAKROAST_PRICE; }
};
class Mocha : public CondimentDecorator {  //摩卡
public:
Mocha ( Beverage*beverage ) {  this->beverage=beverage; }
string getDescription( ){ return beverage->getDescription( )+",Mocha"; }
int cost( ){ return MOCHA_PRICE+beverage->cost( ); }
```

```
};
class Whip :public CondimentDecorator {   //奶泡
public:
Whip ( Beverage*beverage ) { this->beverage=beverage; }
string getDescription( ) {return beverage->getDescription( )+",Whip"; }
int cost( ) { return WHIP_PRICE+beverage->cost( ); }
};
int main() {
Beverage* beverage = new DarkRoast( );
beverage=new Mocha( ( 5 ) ) ;
beverage=new Whip ( ( 6 ) ) ;
cout<<beverage->getDescription ( )<<"￥"<<beverage->cost( ) endl;
return 0;
}
```

编译运行上述程序，其输出结果为：

DarkRoast, Mocha, Whip ￥38

试题9

阅读下列说明和Java代码，将应填入（n）处的字句写在答题纸的对应栏内。

【说明】

某咖啡店当卖咖啡时，可以根据顾客的要求在其中加入各种配料，咖啡店会根据所加入的配料来计算费用。咖啡店所供应的咖啡及配料的种类和价格如表15-3所示。

表15-3咖啡及配料表

咖啡	价格/杯	配料	价格/份
蒸馏咖啡（Espresso）	25	摩卡（Mocha）	10
深度烘焙咖啡（DarkRoast）	20	奶泡（Whip）	8

现采用装饰器（Decorator）模式来实现计算费用的功能，得到如图15-31所示的类图

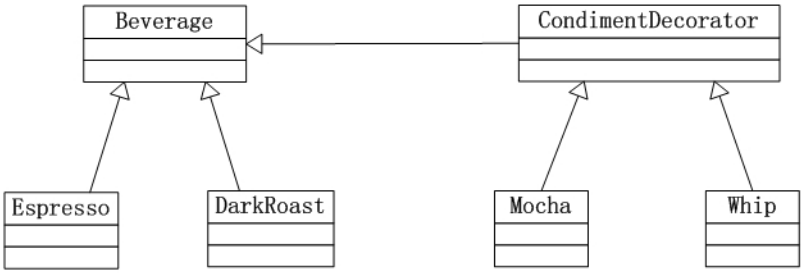

图15-31类图

【Java代码】

```
import java.util.*;
（1）class Beverage {       //饮料
String description = "Unknown Beverage";
public   （2）（ ）{return description;}
public   （3）；
}
```

```
abstract class CondimentDecorator extends Beverage {  //配料
（4）；
}
class Espresso extends Beverage {    //蒸馏咖啡
private final int ESPRESSO_PRICE = 25;
public Espress0() {   description="Espresso";  }
public int cost() {   return ESPRESSO_PRICE;   }
}
class DarkRoast extends Beverage {      //深度烘焙咖啡
private finalint DARKROAST_PRICE = 20;
public DarkRoast() { description = "DarkRoast";   }
public int cost(){ rcturn DARKROAST PRICE;   }
}
class Mocha extends CondimentDecorator {  //摩卡
private final int MOCHA_PRICE = 10;
public Mocha（Beverage beverage）{
this.beverage = beverage;
}
public String getDescription() {
return beverage.getDescription0 + ", Mocha";
}
    public int cost() {
        return MOCHA_PRICE + beverage.cost();
}
}
class Whip extends CondimentDecorator {   //奶泡
private finalint WHIP_PRICE = 8;
public Whip（Beverage beverage）{ this . beverage = beverage; }
public String getDescription（ ）{
return beverage.getDescription（ ）+", Whip";
}
public int cost() { return WHIP_PRICE + beverage.cost(); }
}
public class Coffee {
public static void main（String args[]）{
Beverage beverage = new DarkRoast();
beverage=new Mocha（5）;
beverage=new Whip （6）;
System.out.println（beverage.getDescription0 +"￥" +beverage.cost()）;
```

}

}

编译运行上述程序，其输出结果为：

DarkRoast, Mocha, Whip ￥38

试题10

阅读下列说明和C++代码，将应填入(n)处的字句写在答题纸的对应栏内。

【说明】

某大型商场内安装了多个简易的纸巾售卖机，自动出售2元钱一包的纸巾，且每次仅售出一包纸巾。纸巾售卖机的状态图如图15-32所示。

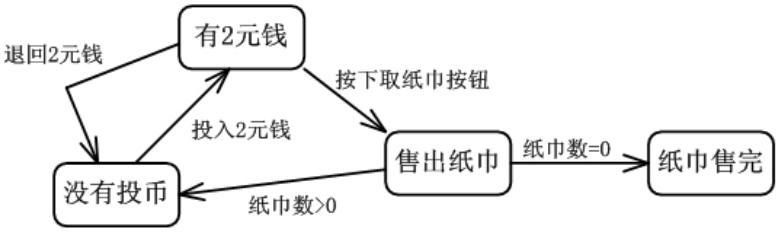

图15-32状态图

采用状态(State)模式来实现该纸巾售卖机，得到如图15-33所示的类图。其中类State为抽象类，定义了投币、退币、出纸巾等方法接口。类SoldState、SoldOutState、NoQuarterState和HasQuarterState分别对应图15-32中纸巾售卖机的4种状态：售出纸巾、纸巾售完、没有投币、有2元钱。

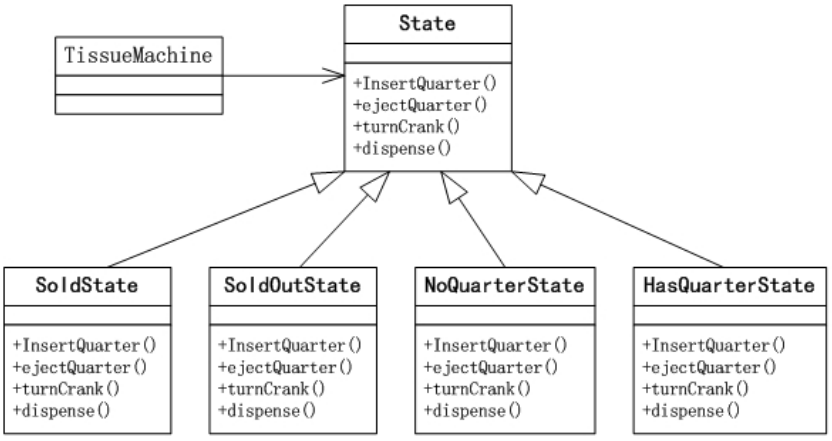

图15-33类图

【C++代码】

```
#include<iostream>
using namespace std;
//以下为类的定义部分
class TissueMachine;  //类的提前引用
class State{
public:
virtual void insertQuarter()=0;//投币
virtual void ejectQuarter()=0;//退币
virtual void turnCrank()=0;//按下“出纸巾”按钮
virtual void dispense()=0;//出纸巾
```

```
};
/*类SoldOutState. NoQuarterState. HasQuarterState. SoldState的定义省略，每个类中均定义了私有数据成员TissueMachine* tissueMachine;*/
class TissueMachine{
private:
  (1)   *soldOutState, *noQuarterState, *hasQuarterState,*soldState, *state;
int count,   //纸巾数
public:
TissueMachine(int numbers);
void setState(State* state);
State* getHasQuarterState();
State* getNoQuarterState();
State* getSoldState();
State* getSoldOutState();
int getCount();
//其余代码省略
};
//以下为类的实现部分
void NoQuarterState::insertQuarter(){
tissueMachine->setState( (2) );
}
void HasQuarterState::ejectQuarter(){
tissueMachine->setState( (3) );
}
void SoldState::dispense(){
if(tissueMachine->getCount()>0){
tissueMachine->setState ( (4) );
}
else{
tissueMachine->setState( (5) );
}
}//其余代码省略
```

试题11

阅读下列说明和Java代码，将应填入(n)处的字句写在答题纸的对应栏内。

【说明】

某大型商场内安装了多个简易的纸巾售卖机，自动出售2元钱一包的纸巾，且每次仅售出一包纸巾。纸巾售卖机的状态图如图15-34所示。

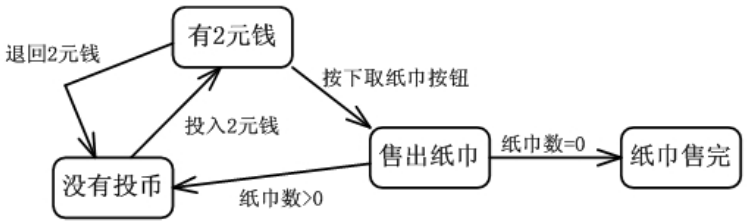

图15-34纸巾售卖机状态图

采用状态(State)模式来实现该纸巾售卖机，得到如图15-35所示的类图。其中类State为抽象类，定义了投币、退币、出纸巾等方法接口。类SoldState、SoldOutState、NoQuarterState和HasQuarterState分别对应图15-34中纸巾售卖机的4种状态：售出纸巾、纸巾售完、没有投币、有2元钱。

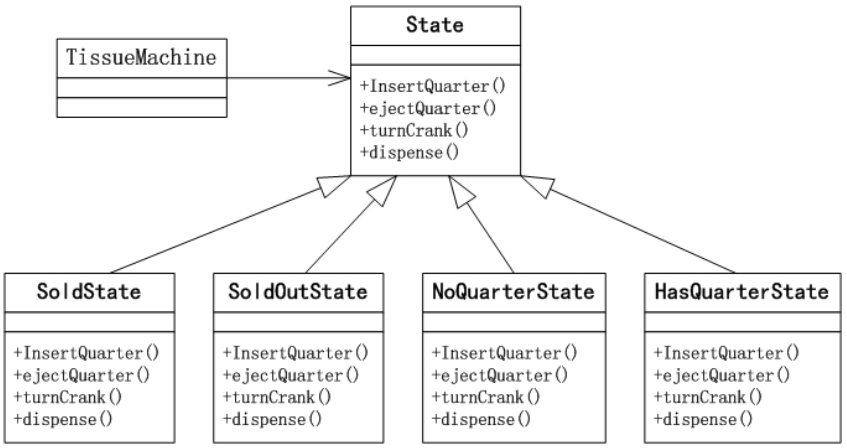

图15-35 类图

【Java代码】

```
import java.util.*;
interface State{
public void insertQuarter();   //投币
public void ejectQuarter();    //退币
public void turnCrank();//按下“出纸巾”按钮
public void dispense();    //出纸巾
}
class TissueMachine {
( 1 )       soldOutState, noQuarterState, hasQuarterState, soldState, state;
state = soldOutState;
int count = 0;                //纸巾数
public TissueMachine(intnumbers) {  /*实现代码省略 */}
public StategetHasQuarterState()   { returnhasQuarterState; }
public StategetNoQuarterState()    {  returnnoQuarterState; }
public State getSoldState()        {   return soldState;  }
public State getSoldOutState()     {   return soldOutState; }
public int getCount()              {   return count;  }
//其余代码省略
}
```

```
class NoQuarterState implements State {
TissueMachine tissueMachine;
public void insertQuarter() {
tissueMachine.setState((2));
}
//构造方法以及其余代码省略
}
class HasQuarterState implements State {
TissueMachine tissueMachine;
public void ejectQuarter() {
tissueMachine.setState ((3));
}
//构造方法以及其余代码省略
}
class SoldState implements State {
TissueMachine tissueMachine;
public void dispense() {
if(tissueMachine.getCount() > 0)  {
tissueMachine.setState ((4));
} else {
tissueMachine.setState  ((5)); }
}
}
```

试题12

阅读下列说明和C++代码，将应填入空（n）处的字句写在答题纸的对应栏内。

【说明】

某饭店在不同的时段提供多种不同的餐饮，其菜单的结构图如图15-36所示。

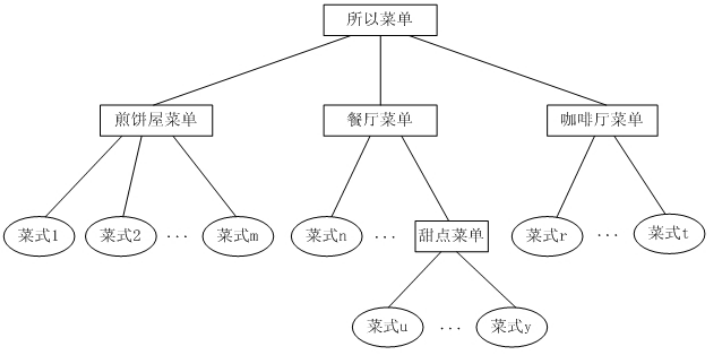

图15-36 菜单结构图

现在采用组合(Composition)模式来构造该饭店的菜单，使得饭店可以方便地在其中增加新的餐饮形式，得到如图15-37所示的类图。其中MenuComponent为抽象类，定义了添加(add)新菜单和打印饭店所有菜单信息(print)的方法接口。类Menu表示饭店提供的每种餐饮形式的菜单，如煎饼屋菜单、咖啡屋菜单等。每种菜单中都可以添加子菜单，例如图15-36中的甜点菜单。类MenuItem表

示菜单中的菜式。

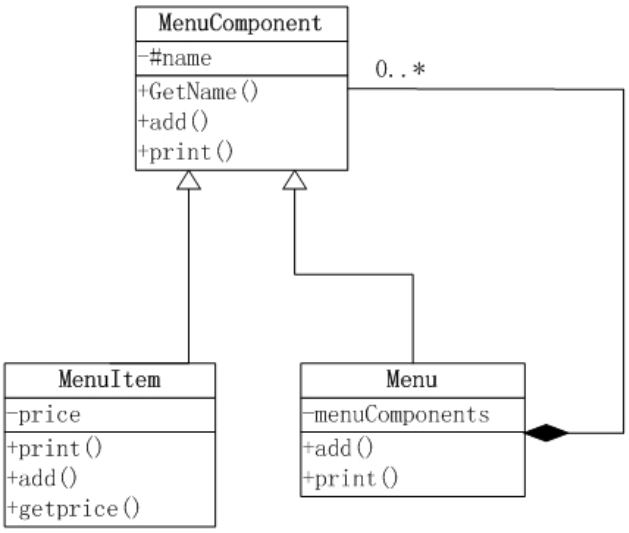

图15-37 类图

【C++代码】

```
#include<iostream>
#include<list>
#include <string>
using namespace std;
class MenuComponent{
protected:  string name;
public:
MenuComponent(string name){  this->name= name;}
string getName(){ return name;}
 （1） ；  //添加新菜单
virtual void print()=0;    //打印菜单信息
};
class MenuItem: public MenuComponent{
private:double price;
public:
MenuItem(string name, double price):MenuComponent(name)
{ this->price= price;}
    double getPrice(){ return price;}
    void add(MenuComponent* menuComponent){ return ；}//添加新菜单
    void print(){ cout<<"  "  <<getName()<<" , "<<getPrice()<<endl;}
};
class Menu:public MenuComponent{
private: list<  （2）>  menuComponents;
public:
Menu(string name):MenuComponent(name){}
void add(MenuComponent* menuComponent) //添加新菜单
```

```
{ （3）;}
void print(){
    cout<<"\n"<<getName()<<"\n--------------------"<<endl;
std::list<MenuComponent *>::iterator iter;
    for(iter= menuComponents.begin(); iter!=menuComponents.end(); iter++)
（4）->print();
}
}
void main(){
MenuComponent* allMenus= new Menu("ALL MENUS");
MenuComponent* dinerMenu= new Menu("DINER MENU");
……//创建更多的Menu对象，此处代码省略
allMenus->add(dinerMenu);  //将dinerMenu添加到餐厅菜单中
……//为餐厅增加更多的菜单，此处代码省略
(5) ->print();  //打印饭店所有菜单的信息
}
```

试题13

阅读下列说明和Java代码，将应填入（n）处的字句写在答题纸的对应栏内。

【说明】

某饭店在不同的时段提供多种不同的餐饮，其菜单的结构图如图15-38所示。

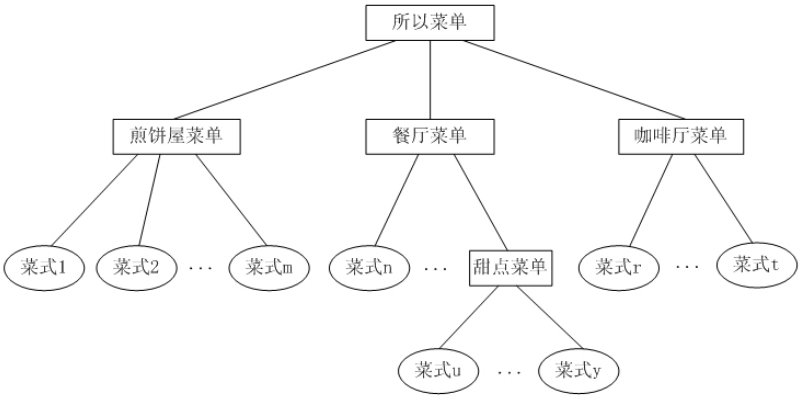

图15-38 菜单结构图

现在采用组合(Composition)模式来构造该饭店的菜单，使得饭店可以方便地在其中增加新的餐饮形式，得到如图15-39所示的类图。其中MenuComponent为抽象类，定义了添加(add)新菜单和打印饭店所有菜单信息(print)的方法接口。类Menu表示饭店提供的每种餐饮形式的菜单，如煎饼屋菜单、咖啡屋菜单等。每种菜单中都可以添加子菜单，例如图15-38中的甜点菜单。类MenuItem表示菜单中的菜式。

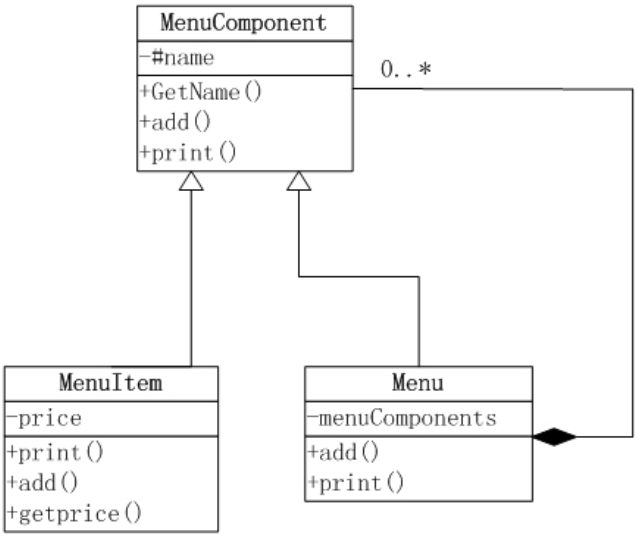

图15-39 类图

【Java代码】

```
import java.util.*;
（1）MenuComponent{
protected String name;
（2）;//添加新菜单
public abstract void print();    //打印菜单信息
public String getName(){ return name;}
}
class MenuItem extends MenuComponent{
private double price;
public MenuItem(String name, double price){
        this.name= name;  this.price= price;
}
public double getPrice(){return price;)
public void add(MenuComponent menuComponent){ return;} //添加新菜单
public void print(){
System.out.print(" "+ getName());
System.out.println(","+ getPrice());
}
}
class Menu extends MenuComponent{
private List<MenuComponent> menuComponents= new ArrayList<MenuComponent>
();
public Menu(String name){ this.name= name;)
public void add(MenuComponent menuComponent){//添加新菜单
menuComponents.（3）;
}
    public void print(){
```

```
    System.out.print("\n"+ getName());
    System.out.println(","+"---------------");
    Iterator iterator - menuComponents.iteratoro;
    while(iterator.hasNext()){
      MenuComponent menuComponent= (MenuComponent)iterator.next();
（4）;
}
}
}
class MenuTestDrive{
public static void main(String args[]){
MenuComponent   allMenus= new Menu("ALL MENUS");
MenuComponent   dinerMenu = new Menu("DINER MENU”);
……//创建更多的Menu对象，此处代码省略
allMenus.add(dinerMenu);    //将dinerMenu添加到餐厅菜单中
……//为餐厅增加更多的菜单，此处代码省略
（5）; //打印饭店所有菜单的信息
}
}
```

试题14

阅读下列说明和C++代码，将应填入（n）处的字句写在答题纸的对应栏内。

【说明】

某公司的组织结构图如图15-40所示，现采用组合（Composition）设计模式来构造该公司的组织结构，得到如图15-41所示的类图。

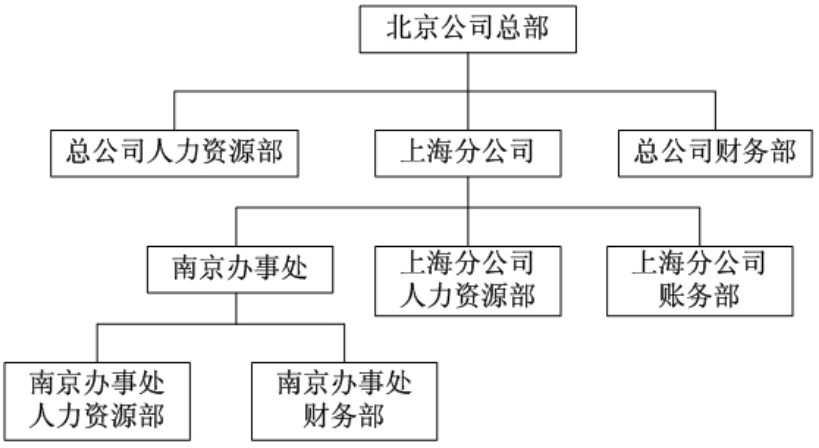

图15 -40组织结构图

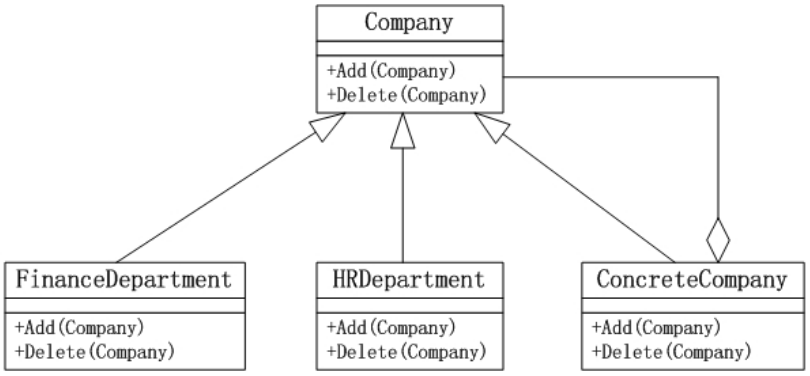

图15-41类图

其中Company为抽象类，定义了在组织结构图上添加（Add）和删除（Delete）分公司/办事处或者部门的方法接口。类ConcreteCompany表示具体的分公司或者办事处，分公司或办事处下可以设置不同的部门。类HRDepartment和FinanceDepartment分别表示人力资源部和财务部。

【C++代码】

```
#include <iostream>
#include <list>
#include <string>
using namespace std;
class Company { //抽象类
protected:
string name;
public:
    Company(string name) { (1)= name; }
(2);//增加子公司、办事处或部门
(3);//删除子公司、办事处或部门
 } ;
class ConcreteCompany:public Company {
private:
list<(4)>children;//存储子公司、办事处或部门
public:
ConcreteCompany(string name):Company(name) {  }
void Add(Company* c) { (5).push back(c); }
void Delete(Company* c) { (6).remove(c); }
 } ;
class HRDepartment:public Company {
public:
HRDepartment(string name):Company(name) {  } //其它代码省略
 } ;
class FinanceDepartment:public Company {
public:
FinanceDepartment(string name):Company(name) {  } //其它代码省略
```

```
};
void main() {
ConcreteCompany *root = new ComcreteCompany(“北京总公司”);
root->Add(new HRDepartment(“总公司人力资源部”));
root->Add(new FinanceDepartment(“总公司财务部”));
ConcreteCompany *comp = new ConcreteCompany(“上海分公司”);
comp->Add(new HRDepartment(“上海分公司人力资源部”));
comp->Add(new FinanceDepartment(“上海分公司财务部”));
(7);
ConcreteCompany *compl = new ConcreteCompany(“南京办事处”);
comp l->Add(new HRDepartment(“南京办事处人力资源部”));
comp l->Add(new FinanceDepartment(“南京办事处财务部”));
(8);//其它代码省略
}
```

试题15

阅读下列说明和Java代码，将应填入（n）处的字句写在答题纸的对应栏内。

【说明】

某公司的组织结构图如图15-42所示，现采用组合（Composition）设计模式来设计，得到如图15-43所示的类图。

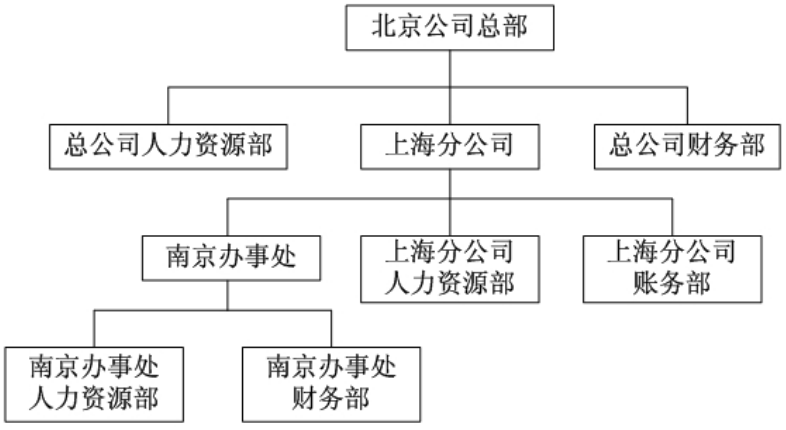

图15-42组织结构图

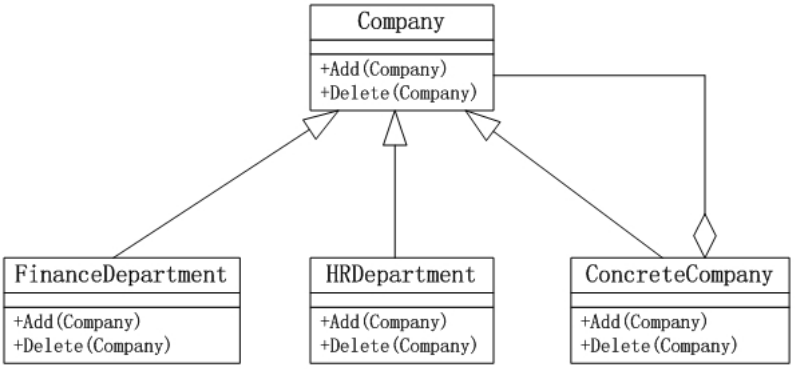

图15-43类图

其中Company为抽象类，定义了在组织结构图上添加（Add）和删除（Delete）分公司/办事处或者部门的方法接口。类ConcreteCompany表示具体的分公司或者办事处，分公司或办事处下可以设置不同的部门。类HRDepartment和FinanceDepartment分别表示人力资源部和财务部。

【Java代码】

```
import java.util.*;
(1)Company {
protected String name;
public Company(String name) { (2)=name; }
public abstract void Add(Company c);//增加子公司、办事处或部门
public abstract void Delete(Company c);//删除子公司、办事处或部门
 }
class ConcreteCompany extends Company {
private List<(3)> children = new ArrayList<(4)>();
                                        //存储子公司、办事处或部门
public ConcreteCompany(String name) { super(name); }
public void Add(Company c) { (5).add(c); }
public void Delete(Company c) { (6).remove(c); }
 }
class HRDepartment extends Company {
public HRDepartment(String name) { super(name); }
//其它代码省略
}
class FinanceDepartment extends Company {
public FinanceDepartment(String name) { super(name); }
//其它代码省略
 }
public class Test {
public static void main(String[] args) {
ConcreteCompany root = new ConcreteCompany(“北京总公司”);
root.Add(new HRDepartment(“总公司人力资源部”));
root.Add(new FinanceDepartment(“总公司财务部”));
ConcreteCompany comp = new ConcreteCompany(“上海分公司”);
comp.Add(new HRDepartment(“上海分公司人力资源部”));
comp.Add(new FinanceDepartment(“上海分公司财务部”));
(7);
ConcreteCompany comp = new ConcreteCompany(“南京办事处”);
compl.Add(new HRDepartment(“南京办事处人力资源部”));
compl.Add(new FinanceDepartment(“南京办事处财务部”);
(8);//其它代码省略
 }
 }
```

　作者：希赛教育软考学院　来源：希赛网　2014年05月06日

习题解析

试题1分析

装饰模式主要的目的是在无法生成子类的情况下给一个对象动态地增加新的职责；享元设计模式是共享大量细粒度的对象；适配器设计模式则是将已有的接口转换为系统希望的接口形式。

试题1答案

（1）C（2）D（3）B

试题2分析

观察者模式的意图是：定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。

在观察者模式类图中，Subject是被观察对象，了解其多个观察者，任意数量的观察者可以观察一个对象，提供一个接口用来绑定以及分离观察者对象。

Concrete Subject是具体被观察对象，存储具体观察者Concrete Observer有兴趣的状态。当其状态改变时，发送一个通知给其所有的观察者对象。

Observer是观察者，定义一个更新接口，在一个被观察对象改变时应被通知。

Concrete Observer是具体观察者，维护一个对Concrete Subject对象的引用。

试题2答案

（4）C（5）B

试题3分析

单件模式保证一个类仅有一个实例，并提供一个访问它的全局访问点。第（6）空的题目中讲“某些类有且仅有一个实例”，所以选择C答案。模板方法是指在方法中定义算法的框架，而将算法中的一些操作步骤延迟到子类中实现。第（7）空的数据库连接、打开、查询这是所有数据库系统操作的步骤，只是访问方式有所不同，在算法框架里没必要事先摆出来，可以放到各个子类中讨论，所以选择D答案。装饰是当不能采用生成子类的方法进行扩充时，动态地给一个对象添加一些额外的功能。第（8）空是“构造带有……”，动态地给一些对象添加额外的功能，所以选择B答案。

试题3答案

（6）C（7）D（8）B

试题4分析

在本题中，根据题目给出的图，我们不难看出该图描述的是桥接模式，它的显著特征是它将抽象部分与实现部分分离，使它们可以相互独立地变化。我们不难从题目给出的图中看出，左边的是抽象类接口，而右边都是实现类接口，显然实现了分离。抽象类接口的下面是抽象的扩充，而实现类接口的下面是具体实现，因此他们可以相互独立地变化。其中：

Abstraction：抽象类定义抽象类的接口。维护一个Implementor（实现抽象类）的对象。

RefinedAbstraction：扩充的抽象类,扩充由Abstraction定义的接口。

Implementor：实现类接口，定义实现类的接口，这个接口不一定要与Abstraction的接口完全一致，事实上这两个接口可以完全不同，一般的讲Implementor接口仅仅给出基本操作，而Abstraction接口则会给出很多更复杂的操作。

ConcreteImplementor：具体实现类，实现Implementor定义的接口并且具体实现它。

试题4答案

（9）A （10）D

试题5分析

单例模式的意图是确保某个类只有一个实例，且能自行实例化，并向整个系统提供这个实例。

试题5答案

（11）D

试题6分析

MVC模式是一个复杂的架构模式，其实现也显得非常复杂。

视图（View）代表用户交互界面，对于Web应用来说，可以概括为HTML界面，但有可能为XHTML、XML和Applet。

模型（Model）：就是业务流程/状态的处理及业务规则的制定。业务模型的设计可以说是MVC最主要的核心。

控制（Controller）可以理解为从用户接收请求，将模型与视图匹配在一起，共同完成用户的请求。

试题6答案

（12）A （13）B

试题7分析

此题考的是设计模式基本概念，要求考生清楚设计模式的优缺点。设计模式是对被用来在特写场景下解决一般设计问题的类和相互通信的对象的描述。一般而言，一个设计模式有4个基本要素：模式名称、问题（模式的使用场合）、解决方案和效果。

每一个设计模式系统地命名、解释和评价了面向对象系统中一个重要的和重复出现的设计。设计模式使人们可以更加简单、方便地复用成功的设计和体系结构；将已证实的技术变成设计模式，也会使新系统的开发者更加容易理解其设计思路。设计模式可以帮助开发者做出有利于复用的选择，避免设计时损害系统复用性。因此正确的答案为B。

试题7答案

（14）B

试题8分析

本题考查了C++语言的应用能力和装饰设计模式的应用。

第（1）空很明显，是要说明属性description在类Beverage中的类型，应该是私有的、受保护的或公有的，从后面的程序我们可以看出，子类中继承使用了该属性，因此这里只能定义为受保护的，因此第（1）空的答案为protected。

第（2）空处也很明显，是要给出一个函数的定义，并且该函数的函数体是“return description;”，从子类奶泡和摩卡中我们不难发现这个函数应该是getDescription，因此本空的答案为virtual string getDescription。

第（3）空需要结合后面各子类才能发现，在Beverage中还应该定义一个函数cost()，而这个函

数在Beverage中并没有实现，因此要定义为纯虚函数，所以第（3）空的答案为virtual int cost()=0。

第（4）空在类CondimentDecorator中，且是该类唯一的一条语句，而他的子类分别是奶泡和摩卡，在奶泡和摩卡这两个类中，都用到了Beverage*beverage，而在使用之前并没有说明，因此这就可以说明，Beverage*beverage是在父类CondimentDecorator中定义的，子类直接继承使用，因此第（4）空的答案为Beverage*beverage。

第（5）和第（6）空在主函数当中，其中第（5）空是要创建一个Mocha对象，应该调用的是类Mocha的构造函数，从类Mocha中，我们可以看出，其构造函数Mocha的参数是一个Beverage类型的对象指针，而在主函数中，开始就定义了一个Beverage类型的对象指针beverage，因此这里只需填写beverage即可。同理第（6）空的答案也是beverage。

试题8答案

（1）protected

（2）virtual string getDescription

（3）virtual int cost()=0

（4）Beverage*beverage

（5）beverage

（6）beverage

试题9分析

本题考查了JAVA语言的应用能力和装饰设计模式的应用。

第（1）空很明显，是要给类Beverage前添加定义的关键字，从整个程序来看，我们应该要将类Beverage定义为抽象类，需要在前面添加关键字abstract，因此第（1）空的答案为abstract。

第（2）空处也很明显，是要给出一个函数的定义，并且该函数的函数体是“return description;”，从子类奶泡和摩卡中我们不难发现这个函数应该是getDescription，而该函数的返回类型String，因此本空的答案为String getDescription。

第（3）空需要结合后面各子类才能发现，在Beverage中还应该定义一个函数cost()，而这个函数在Beverage中并没有实现，因此要定义为抽象函数，所以第（3）空的答案为abstract int cost()=0。

第（4）空在类CondimentDecorator中，且是该类唯一的一条语句，而他的子类分别是奶泡和摩卡，在奶泡和摩卡这两个类中，都用到了Beverage beverage，而在使用之前并没有说明，因此这就可以判定，Beverage beverage是在父类CondimentDecorator中定义的，子类直接继承使用，因此第（4）空的答案为Beverage beverage。

第（5）和第（6）空在主函数当中，其中第（5）空是要创建一个Mocha对象，应该调用的是类Mocha的构造函数，从类Mocha中，我们可以看出，其构造函数Mocha的参数是一个Beverage类型的对象引用，而在主函数中，开始就定义了一个Beverage类型的对象引用beverage，因此这里只需填写beverage即可。同理第（6）空的答案也是beverage。

试题9答案

（1）abstract

（2）String getDescription

（3）abstract int cost()

（4）Beverage beverage

（5）beverage

（6）beverage

试题10分析

本题考查基本面向对象设计模式的运用能力。

状态设计模式主要是能够使一个对象的内在状态改变时允许改变其行为，使这个对象看起来像是改变了其类。由类图可知类State是类SoldState、SoldOutState、NoQuarterState和HasQuarterState分的父类，它抽象了这四个类的共有属性和行为。在使用中，无论是这四个类中那个类的对象，都可被当作State对象来使用。

而根据题目的描述，我们可以知道一个纸巾售卖机它由4种状态，分别是售出纸巾、纸巾售完、没有投币、有2元钱。

在本题中，根据程序我们不难知道第（1）空是要定义5个对象指针，而这些对象指针都应该属于State类型，因此第一空答案为State。

而第（2）在类NoQuarterState（没有投币）的insertQuarter()函数中，而这个函数是投币函数，在该函数中，使用了tissueMachine类的setState方法，该方法是设置纸巾售卖机的当前状态，根据题目给出的纸巾售卖机状态图，我们可以知道，从没有投币状态，经过投币后，应该转换到有2元钱状态。而setState方法的参数是一个State的对象，因此第（2）空应该是一个有2元钱对象，而这里我们可以新创建一个该对象，也可以通过tissueMachine类的getHasQuarterState方法来获得这样一个对象，所以第（2）空答案应该是“tissueMachine->getHasQuarterState()”或“new HasQuarterState”。

而第（3）在类HasQuarterState（有2元钱）的ejectQuarter()函数中，而这个函数是退币函数，在该函数中，也使用了tissueMachine类的setState方法，该方法是设置纸巾售卖机的当前状态，根据题目给出的纸巾售卖机状态图，我们可以知道，从有2元钱状态，经过退币后，应该转换到没有投币状态。而setState方法的参数是一个State的对象，因此第（3）空应该是一个没有投币对象，而这里我们可以新创建一个该对象，也可以通过tissueMachine类的getNoQuarterState方法来获得这样一个对象，所以第（3）空答案应该是“tissueMachine->getNoQuarterState()”或“new NoQuarterState”。

而同样的道理，我们可以知道第（4）空的答案是“tissueMachine->getNoQuarterState()”或“new NoQuarterState”。第（5）空的答案是“tissueMachine->getSoldOutState()”或“new SoldOutState”。

试题10答案

（1）State

（2）tissueMachine->getHasQuarterState()

（3）tissueMachine->getNoQuarterState()

（4）tissueMachine->getNoQuarterState()

（5）tissueMachine->getSoldOutState()

试题11分析

本题考查基本面向对象设计模式的运用能力。

状态设计模式主要是能够使一个对象的内在状态改变时允许改变其行为，使这个对象看起来像

是改变了其类。由类图可知类State是类SoldState、SoldOutState、NoQuarterState和HasQuarterState分的父类，它抽象了这四个类的共有属性和行为。在使用中，无论是这四个类中那个类的对象，都可被当作State对象来使用。

而根据题目的描述，我们可以知道一个纸巾售卖机它由4种状态，分别是售出纸巾、纸巾售完、没有投币、有2元钱。

在本题中，根据程序我们不难知道第（1）空是要定义5个对象的引用，而这些变量都应该属于State类型，因此第一空答案为State。

而第（2）在类NoQuarterState（没有投币）的insertQuarter()函数中，而这个函数是投币函数，在该函数中，使用了tissueMachine类的setState方法，该方法是设置纸巾售卖机的当前状态，根据题目给出的纸巾售卖机状态图，我们可以知道，从没有投币状态，经过投币后，应该转换到有2元钱状态。而setState方法的参数是一个State的对象，因此第（2）空应该是一个有2元钱对象，而这里我们可以新创建一个该对象，也可以通过tissueMachine类的getHasQuarterState方法来获得这样一个对象，所以第（2）空答案应该是“tissueMachine.getHasQuarterState()”或“new HasQuarterState”。

而第（3）在类HasQuarterState（有2元钱）的ejectQuarter()函数中，而这个函数是退币函数，在该函数中，也使用了tissueMachine类的setState方法，该方法是设置纸巾售卖机的当前状态，根据题目给出的纸巾售卖机状态图，我们可以知道，从有2元钱状态，经过退币后，应该转换到没有投币状态。而setState方法的参数是一个State的对象，因此第（3）空应该是一个没有投币对象，而这里我们可以新创建一个该对象，也可以通过tissueMachine类的getNoQuarterState方法来获得这样一个对象，所以第（3）空答案应该是“tissueMachine.getNoQuarterState()”或“new NoQuarterState”。

而同样的道理，我们可以知道第（4）空的答案是“tissueMachine.getNoQuarterState()”或“new NoQuarterState”。第（5）空的答案是“tissueMachine.getSoldOutState()”或“new SoldOutState”。

试题11答案

（1）State

（2）tissueMachine.getHasQuarterState()

（3）tissueMachine.getNoQuarterState()

（4）tissueMachine.getNoQuarterState()

（5）tissueMachine.getSoldOutState()

试题12分析

本题考查基本面向对象设计模式的运用能力。

组合设计模式主要是表达整体和部分的关系，并且对整体和部分对象的使用无差别。由UML结构图知MenuComponent是MenuItem类和Menu类的父类，它抽象了两个类的共有属性和行为。在使用中，无论是MenuItem对象还是Menu对象，都可被当作MenuComponent对象来使用。另外由UML结构图可知，类MenuComponent和Menu之间存在共享关系，即Menu对象可以共享其它的Menu对象和MenuItem对象。

第（1）空是要在类MenuComponent中定义添加新菜单的方法，即add方法，而类MenuComponent是抽象类，因此该方法也应该是抽象方法，及需要加关键字virtual，因此空

习题解析

（1）应填virtual void add(MenuComponent* menuComponent)=0，而第（2）空很明显应该填add函数，而根据题目描述“每种菜单中都有本空应填

menuComponents.push_back(menuComponent)。在类Menu类下的print函数中，定义了对象iter,因此第4空可以通过这个对象来引用print()方法，因此第（4）空答案应该填（*iter）。而第（5）空为了能打印饭店所有菜单的信息，应该填allMenus。

试题12答案

（1）virtual void add(MenuComponent* menuComponent)

（2）MenuComponent *

（3）menuComponents.push_back(menuComponent)

（4）(*iter)

（5）allMenus

试题13分析

本题考查基本面向对象设计模式的运用能力。

组合设计模式主要是表达整体和部分的关系，并且对整体和部分对象的处理没有差别。由UML结构图知MenuComponent是MenuItem类和Menu类的父类，它抽象了两个类的共有属性和行为。在使用中，无论是MenuItem对象还是Menu对象，都可被当作MenuComponent对象来使用。另外由UML结构图可知，类MenuComponent和Menu之间存在共享关系，即Menu对象可以共享其它的Menu对象和MenuItem对象。

第（1）空是要在类MenuComponent定义前加一个关键字，而类MenuComponent是抽象类，因此空（1）应填abstract class或public abstract class，而第（2）空是要定义添加新菜单的方法，该方法应该是抽象方法，因此第（2）空的答案应该是public abstract void add(MenuComponent menuComponent)；

很明显应该填MenuComponent*；空（3）是要实现Menu类下的add函数，而根据题目描述“每种菜单中都可以添加子菜单”，因此这里要实现添加子菜单，因此本空应填add(menuComponent)。在空（4）需要调用print()方法，因此空（4）处应填menuComponent.print()。而第（5）空为了能打印饭店所有菜单的信息，应该填aIIMenus.print()。

试题13答案

（1）abstract class 或 public abstract class

（2）public abstract void add(MenuComponent menuComponent)

或 abstract void add(MenuComponent menuComponent)

或 protected abstract void add(MenuComponent menuComponent)

（3）add(menuComponent)

（4）menuComponent.print()

（5）allMenus.print()

试题14分析

本题考查组合模式的基本应用。组合设计模式主要是表达整体和部分的关系。由类图可知Company是ConcreteCompany类、HRDepartment 类和FinanceDepartment类的父类，它抽象

了三个类的共有属性和行为。

第（1）空是在构造函数中，被赋值为name，而name是构成函数所带的参数，那么这里是给类的一个属性name赋值，因此这空答案为：this->name。

第（2）空与第（3）空我们可以根据注释来完成，根据题目的描述，这里只提供接口，是虚方法，因此第（2）空与第（3）空分别应该为virtual void Add(Company* c) = 0和virtual void Delete(Company* c) = 0，这两个方法的参数可以从后面类的相应方法中获得。

第（4）空根据注释可以推导出应该填Company*。第（5）空与第（6）空的答案应该一致，都应该为children。

第（7）空和第（8）空在main函数中，用来创建组件结构图，根据题目提供的图，我们可以知道，创建了上海分公司接的后，应该将其添加至root下，因此第7空答案为root->Add(comp)，同样的道理，第（8）空的答案为comp->Add(comp1)。

试题14答案

（1）this->name

（2）virtual void Add(Company* c)=0

（3）virtual void Delete(Company* c)=0

（4）Company*

（5）children

（6）children

（7）root->Add(comp)

（8）comp->Add(comp1)

试题15分析

本题考查了Java语言的应用能力和组合设计模式的应用。

第（1）空在类名Company前，很明显应该要加关键字abstract class，因为题目描述的Company类是一个抽象类。第（2）空在构造函数中的赋值语句中，应该为this.name，就是给该类的一个属性name赋值，这里应该用this.name来引用这个属性。第（3）空与第（4）空的答案应该都为Company，这样要注意在java中与C++中的区别。第（5）和第（6）空的答案应该一样，一个用来增加节点，一个用来删除节点，都是使用的children对象。根据题目提供的组织结构图，我们可以知道，创建了上海分公司接的后，应该将其添加至root（北京总公司）下，因此第7空答案为root.Add(comp)，同样的道理，第（8）空的答案为comp.Add(comp1)。

试题15答案

（1）abstract class

（2）this.name

（3）Company

（4）Company

（5）children

（6）children

（7）root.Add(comp)

（8）comp.Add(comp1)