第3章

创建型模式实训

随着面向对象技术的发展和广泛应用，设计模式不再是一个新兴名词，它已逐步成为系统架构人员、设计人员、分析人员以及程序开发人员所需掌握的基本技能之一。设计模式已广泛应用于面向对象系统的设计和开发，成为面向对象领域的一个重要组成部分。设计模式通常可以分为三类：创建型模式、结构型模式和行为型模式。

创建型模式关注对象的创建过程，是一类最常见的设计模式，在软件开发中应用非常广泛。创建型模式将对象的创建和使用分离，在使用对象时无须关心对象的创建细节，从而降低系统的耦合度，让设计方案更易于修改和扩展。

3.1 知识讲解

在GoF设计模式中包含5种创建型模式，分别是工厂方法模式(Factory Method Pattern)、抽象工厂模式(Abstract Factory Pattern)、建造者模式(Builder Pattern)、原型模式(Prototype Pattern)和单例模式(Singleton Pattern)。作为工厂模式的最简单形式，简单工厂模式(Simple Factory Pattern)也是创建型模式必不可少的成员。

3.1.1 设计模式

设计模式(Design Pattern)是前人经验的总结，它使人们可以方便地复用成功的设计和体系结构。当人们在特定的环境下遇到特定类型的问题时，可以采用他人已使用过的一些成功的解决方案，一方面降低了分析、设计和实现的难度，另一方面可以使得系统具有更好的可重用性和灵活性。

1. 模式的起源和定义

模式起源于建筑业而非软件业，模式之父——美国加利福尼亚大学环境结构中心研究所所长Christopher Alexander博士用了约20年的时间，对舒适型住宅和周边环境进行了大量的调查和资料收集工作，发现人们对舒适型住宅和城市环境存在着共同的认知规律。

他把这些规律归纳为 253 个模式,对每一个模式都从 Context(模式可适用的前提条件)、Theme 或 Problem(在特定条件下要解决的目标问题)和 Solution(对目标问题的求解方案)三个方面进行描述,并给出了从用户需求分析到建筑环境结构设计直至经典实例的过程模型。Alexander 给出模式的经典定义如下:每个模式都描述了一个在实际环境中不断出现的问题,然后描述了该问题的解决方案的核心,通过这种方式,可以无数次地使用那些已有的解决方案,无须再重复相同的工作。即模式是在特定环境中解决问题的一种方案(A pattern is a solution to a problem in a context)。

2. 软件模式与设计模式

软件模式是将"模式"的一般概念用于软件开发领域,即软件开发的总体指导思路或参照样板。最早将模式引入软件领域的是 1991 年至 1992 年以"四人组"(Gang of Four,简称 GoF,分别是 Erich Gamma,Richard Helm,Ralph Johnson,John Vlissides)自称的四位著名软件工程学者,他们在 1994 年归纳发表了 23 种设计模式,旨在用模式来统一沟通面向对象方法在分析、设计和实现之间的鸿沟。1995 年,"四人组"出版了《设计模式——可复用面向对象软件的基础》一书,这本书成为设计模式的经典书籍。

软件模式包括设计模式、体系结构模式、分析模式、过程模式等,软件生存期的各个阶段都存在着被认同的模式。软件模式是对软件开发这一特定"问题"的"解法"的某种统一表示,它和 Alexander 所描述的模式定义完全相同,即软件模式等于特定环境下的问题及其解法。

在软件模式领域,目前研究最为深入的是设计模式。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,使用设计模式的目的是提高代码的可重用性,让代码更容易被他人理解,并让代码具有更好的可靠性。毫无疑问,这些设计模式已经在前人的系统中得以证实并广泛使用,它使代码编制真正实现工程化,将已证实的技术表述成设计模式也会使新系统开发者更加容易理解其设计思路。每一种设计模式都是一种或多种面向对象设计原则的体现。

在设计模式领域,狭义的设计模式就是指 GoF 的 23 种经典模式,不过设计模式不限于这 23 种,随着软件开发技术的发展,越来越多的新模式不断诞生并得以广泛应用。

3. 设计模式关键元素

设计模式包含模式名称、问题、目的、解决方案、效果、实例代码和相关设计模式等基本要素,其中的关键元素包括以下 4 个方面。

(1) 模式名称(Pattern Name)

给模式取一个助记名,用一两个词来描述模式待解决的问题、解决方案和使用效果,以便更好地理解模式并方便设计人员及开发人员之间的交流。

(2) 问题(Problem)

描述应该在何时使用模式,即在解决何种问题时可使用该模式。在问题部分有时会包括使用模式必须满足的一系列先决条件。

(3) 解决方案(Solution)

描述设计的组成成分、它们之间的相互关系及各自的职责和协作方式。模式就像一个模板,可应用于多种不同场合,所以解决方案并不描述一个特定而具体的设计或实现,而是

提供一个问题的抽象描述和具有一般意义的元素组合(类或对象组合)。

(4) 效果(Consequences)

描述模式应用的效果以及使用模式时应权衡的问题,就是模式的优缺点。没有一种解决方案是完美的,每种设计模式都具有自己的优点,但也存在一些缺陷,它们对于评价设计选择和理解使用模式的代价及好处具有重要意义。模式效果有助于选择合适的模式,它不仅包括时间和空间的权衡,还包括对系统的灵活性、扩充性或可移植性的影响。

4. 设计模式分类

常用的设计模式分类方式有以下两种。

(1) 根据模式的目的和用途,设计模式可分为创建型模式(Creational Pattern)、结构型模式(Structural Pattern)和行为型模式(Behavioral Pattern)三种。创建型模式主要用于创建对象;结构型模式主要用于处理类或对象的组合;行为型模式主要用于描述类或对象的交互以及职责的分配。

(2) 根据模式的处理范围,设计模式可分为类模式和对象模式。类模式处理类和子类之间的关系,这些关系通过继承建立,在编译时刻就被确定下来,属于静态关系;对象模式处理对象间的关系,这些关系在运行时时刻变化,更具动态性。

5. 设计模式优点

设计模式融合了众多专家的经验,并以一种标准的形式供广大开发人员所用,它提供了一种通用的语言以方便开发人员之间的沟通和交流,使得重用成功的设计更加容易,并避免那些导致不可重用的设计方案。模式是一种指导,在一个良好的指导下,有助于做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。设计模式使得设计更易于修改,并提升设计文档的水平,使得设计更通俗易懂。

3.1.2 创建型模式概述

创建型模式(Creational Pattern)对类的实例化过程即对象的创建过程进行了抽象,能够使软件模块做到与对象的创建和组织无关。创建型模式隐藏了对象的创建细节,通过隐藏对象如何被创建和组合在一起达到使整个系统独立的目的。在掌握创建型模式时,需要回答以下三个问题:创建什么(What)、由谁创建(Who)和何时创建(When)。

创建型模式包括 6 种,其定义和使用频率如表 3-1 所示。

表 3-1 创建型模式

模式名称	定义	使用频率
简单工厂模式(Simple Factory Pattern)	定义一个类,根据参数的不同返回不同类的实例,这些类具有公共的父类和一些公共的方法。简单工厂模式不属于 GoF 设计模式,它是最简单的工厂模式	★★★★☆
工厂方法模式(Factory Method Pattern)	定义一个用于创建对象的接口,让子类决定将哪一个类实例化。工厂方法模式使一个类的实例化延迟到其子类	★★★★★

续表

模 式 名 称	定　　义	使 用 频 率
抽象工厂模式(Abstract Factory Pattern)	提供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	★★★★★
建造者模式(Builder Pattern)	将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	★★☆☆☆
原型模式(Prototype Pattern)	用原型实例指定创建对象的种类,并且通过拷贝这个原型来创建新的对象	★★★☆☆
单例模式(Singleton Pattern)	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	★★★★☆

3.1.3　简单工厂模式

简单工厂模式并不是 GoF 23 个设计模式中的一员,但是一般将它作为学习设计模式的起点。简单工厂模式又称为静态工厂方法模式(Static Factory Method Pattern),属于类创建型模式。在简单工厂模式中,可以根据参数的不同返回不同的类的实例。简单工厂模式专门定义一个类来负责创建其他类的实例,这个类称为工厂类,被创建的实例通常都具有共同的父类。简单工厂模式结构如图 3-1 所示。

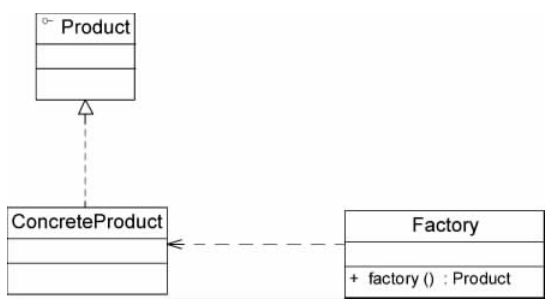

图 3-1　简单工厂模式结构图

在模式结构图中,Factory 表示工厂类,它是整个模式的核心,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。工厂类中有一个负责生产对象的静态工厂方法,系统可根据工厂方法所传入的参数,动态决定应该创建出哪一个产品类的实例。工厂方法是静态的,而且必须有返回类型,其返回类型为抽象产品类型,即 Product 类型; Product 表示抽象产品角色,它是简单工厂模式所创建的所有对象的父类,负责定义所有实例所共有的公共接口; ConcreteProduct 表示具体产品角色,它是简单工厂模式的创建目标,所有创建的对象都是充当这个角色的某个具体类的实例。一个系统中一般存在多个 ConcreteProduct 类,每种具体产品对应一个 ConcreteProduct 类。

在简单工厂模式中,工厂类包含必要的判断逻辑,决定在什么时候创建哪一个产品类的实例,客户端可以免除直接创建产品对象的责任,而仅仅“消费”产品,简单工厂模式通过这种方式实现了对责任的划分。但是由于工厂类集中了所有产品创建逻辑,一旦不能正常工作,整个系统都要受到影响; 同时系统扩展较为困难,一旦添加新产品就不得不修改工厂逻辑,违反了开闭原则,并造成工厂逻辑过于复杂。正因为简单工厂模式存在种种问题,一般

只将它作为学习其他工厂模式的入门，当然在一些并不复杂的环境下也可以直接使用简单工厂模式。

3.1.4　工厂方法模式

工厂方法模式也称为工厂模式，又称为虚拟构造器(Virtual Constructor)模式或多态模式，属于类创建型模式。在工厂方法模式中，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(创建)哪一个类，工厂方法模式结构如图 3-2 所示。

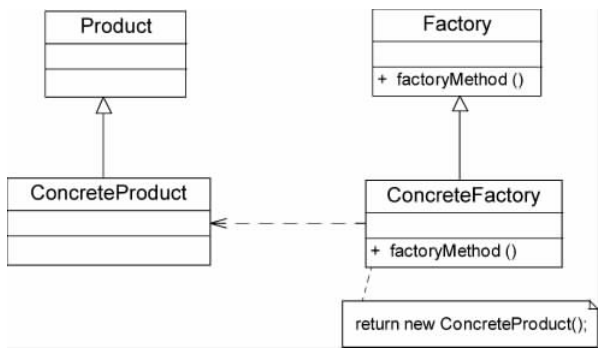

图 3-2　工厂方法模式结构图

在模式结构图中，Product 表示抽象产品，它定义了产品的接口；ConcreteProduct 表示具体产品，它实现抽象产品的接口；Factory 表示抽象工厂，它声明了工厂方法(Factory Method)，返回一个产品；ConcreteFactory 表示具体工厂，它实现工厂方法，由客户端调用，返回一个产品的实例。

在工厂方法模式中，工厂方法用来创建客户所需要的产品，同时还向客户隐藏了哪种具体产品类将被实例化这一细节。工厂方法模式的核心是抽象工厂类 Factory，各种具体工厂类继承抽象工厂类并实现在抽象工厂类中定义的工厂方法，从而使得客户只需要关心抽象产品和抽象工厂，完全不用理会返回的是哪一种具体产品，也不用关心它是如何被具体工厂创建的。在系统中加入新产品时，无须修改抽象工厂和抽象产品提供的接口，无须修改客户端，也无须修改其他具体工厂和具体产品，而只要添加一个具体工厂和具体产品即可，这样，系统的可扩展性也就变得非常好，符合开闭原则。但是在添加新产品时，需要编写新的具体产品类，而且还要提供与之对应的具体工厂类，难免会增加系统类的个数，增加系统的开销。

3.1.5　抽象工厂模式

抽象工厂模式是所有形式的工厂模式中最为抽象和最具一般性的一种形态。抽象工厂模式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。抽象工厂模式又称为 Kit 模式，属于对象创建型模式。在抽象工厂模式中，引入了产品等级结构和产品族的概念，产品等级结构是指抽象产品与具体产品所构成的继承层次关系，产品族(Product Family)是同一个工厂所生产的一系列产品，即位于不同产品等级结构且功能相关联的产品组成的家族，抽象工厂模式结构如图 3-3 所示。

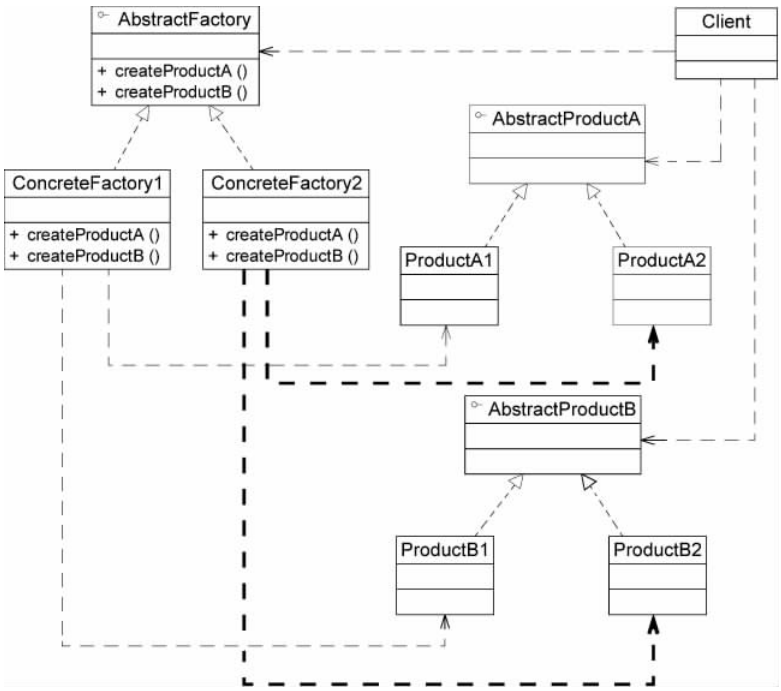

图 3-3 抽象工厂模式结构图

在模式结构图中，AbstractFactory 表示抽象工厂，用于声明创建抽象产品的方法，即工厂方法；ConcreteFactory1 和 ConcreteFactory2 表示具体工厂，它们实现抽象工厂声明的抽象工厂方法用于创建具体产品；AbstractProductA 和 AbstractProductB 表示抽象产品，它们为每一种产品声明接口；ProductA1、ProductA2、ProductB1、ProductB2 表示具体产品，它们定义具体工厂创建的具体产品对象类型，实现产品接口；Client 表示客户类，即客户应用程序，它针对抽象工厂和抽象产品编程。

抽象工厂模式是所有工厂模式最一般的形式，当抽象工厂模式退化到只有一个产品等级结构时，即变成了工厂方法模式；当工厂方法模式的工厂类只有一个，且工厂方法为静态方法时，则变成了简单工厂模式。与工厂方法模式类似，抽象工厂模式隔离了具体类的生成，使得客户类并不需要知道什么样的对象被创建。由于这种隔离，更换一个具体工厂就变得相对容易。所有的具体工厂都实现了抽象工厂中定义的那些公共接口，因此只需改变具体工厂的实例，就可以在某种程度上改变整个软件系统的行为。另外，应用抽象工厂模式可以实现高内聚低耦合的设计目的，因此抽象工厂模式得到了广泛的应用。使用抽象工厂模式的最大好处之一是当一个产品族中的多个对象被设计成一起工作时，它能够保证客户端始终只使用同一个产品族中的对象，这对于那些需要根据当前环境来决定其行为的软件系统来说，是一种非常实用的设计模式。

通过对抽象工厂模式结构进行分析可知，在抽象工厂模式中，增加新的产品族很容易，只需要增加一个新的具体工厂类，并在相应的产品等级结构中增加对应的具体产品类，但是在该模式中，增加新的产品等级结构很困难，需要修改抽象工厂接口和已有的具体工厂类。抽象工厂模式的这个特点称为开闭原则的倾斜性，即它以一种倾斜的方式支持增加新的产品，它为新产品族的增加提供方便，而不能为新的产品等级结构的增加提供这样的方便。

3.1.6 建造者模式

建造者模式(Builder Pattern)强调将一个复杂对象的构建过程与它的表示分离,使得同样的构建过程可以创建不同的表示。建造者模式描述如何一步一步地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,用户不需要知道内部的具体构建细节。建造者模式属于对象创建型模式,建造者模式结构如图 3-4 所示。

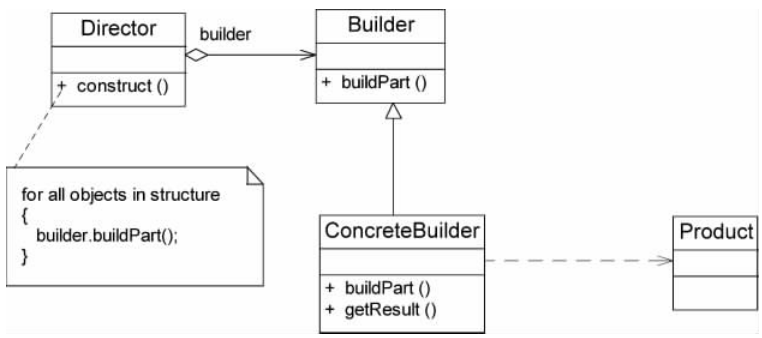

图 3-4 建造者模式结构图

在模式结构图中,Builder 表示抽象建造者,它为创建一个 Product 对象的各个部件指定抽象接口; ConcreteBuilder 表示具体建造者,它实现了 Builder 接口,用于构造和装配产品的各个部件,在其中定义并明确它所创建的产品,并提供返回产品的接口; Director 表示指挥者,它用于构建一个实现 Builder 接口的对象; Product 表示产品角色,它是被构建的复杂对象,具体建造者创建该产品的内部表示并定义它的装配过程。

建造者模式与抽象工厂模式很相似,但是 Builder 返回一个完整的产品,而 AbstractFactory 返回一系列相关的产品; 在 AbstractFactory 中,客户生成自己要用的对象,而在 Builder 中,客户指导 Director 类如何去生成对象,或是如何合成一些对象,侧重于一步步构造一个复杂对象,然后将结果返回。如果抽象工厂模式是一个汽车配件生产厂,那么建造者模式是一个汽车组装厂,通过对配件的组装返回一台完整的汽车。建造者模式将复杂对象的构建与对象的表现分离开来,这样使得同样的构建过程可以创建出不同的表现对象。

使用建造者模式时,客户端不必知道产品内部组成的细节; 每一个 Builder 都相对独立,而与其他的 Builder 无关; 同样是生成产品,工厂模式是生产某一类产品,而建造者模式则是将产品的零件按某种生产流程组装起来,它可以指定生成顺序; 建造者模式将一个复杂对象的创建职责进行了分配,它把构造过程放到指挥者方法中,装配过程放在具体建造者类中。建造者模式的产品之间一般具有共通点,但如果产品之间的差异性很大,就需要借助工厂方法模式或者抽象工厂模式。另外,如果产品的内部变化复杂,Builder 的每一个子类都需要对应到不同的产品去执行构建操作,这就需要定义很多个具体建造类来实现这种变化,将导致系统类个数的增加。

3.1.7 原型模式

在系统开发过程中,有时候有些对象需要被频繁创建,原型模式(Prototype Pattern)通过给出一个原型对象来指明所要创建的对象的类型,然后通过复制这个原型对象的办法创

建出更多同类型的对象。原型模式是一种对象创建型模式,用原型实例指定创建对象的种类,并且通过拷贝这些原型创建新的对象。原型模式允许一个对象再创建另外一个可定制的对象,无须知道任何创建的细节。其工作原理是: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原型自己来实现创建过程,原型模式结构如图 3-5 所示。

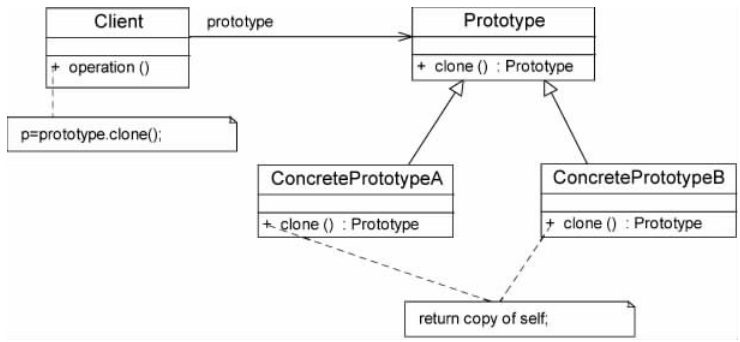

图 3-5 原型模式结构图

在模式结构图中,Prototype 表示抽象原型类,它定义具有克隆自己的方法的接口;ConcretePrototypeA 和 ConcretePrototypeB 表示具体原型类,它们实现具体的克隆方法;Client 表示客户类,它让一个原型对象克隆自身从而创建一个新的对象。

原型模式又可分为两种,分别为浅克隆和深克隆。浅克隆仅仅复制所考虑的对象,而不复制它所引用的对象,也就是其中的成员对象并不复制。在深克隆中,除了对象本身被复制外,对象包含的引用也被复制,即成员对象也将被复制。

原型模式允许动态增加或减少产品类,由于创建产品类实例的方法是产品类内部所具有的,因此增加新产品对整个结构没有影响,新产品只需继承抽象原型类并实现自身的克隆方法即可;原型模式提供了简化的创建结构,在工厂方法模式中常常需要有一个与产品类等级结构相同的工厂等级结构,而原型模式无须这样;对于创建多个相同的复杂结构对象,原型模式简化了创建步骤,在第一次创建成功后可以非常方便地复制出多个相同的对象。原型模式的最主要缺点就是每一个类必须配备一个克隆方法,在对已有系统进行改造时难度较大,而且在实现深克隆时需要编写较为复杂的代码。

3.1.8 单例模式

单例模式(Singleton Pattern)确保某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;二是它必须自行创建这个实例;三是它必须自行向整个系统提供这个实例。单例模式是一种对象创建型模式,单例模式结构如图 3-6 所示。

在模式结构图中,Singleton 表示单例类,它提供了一个 getInstance()方法,让客户可以使用它的唯一实例,其内部实现确保只能生成一个实例。

当一个系统要求一个类只有一个实例时可使用单例模式,单例模式为系统提供了对唯一实例的受控访问,并且可以对单例模式进行扩展获得可变数目的实例,即多例模式,可以

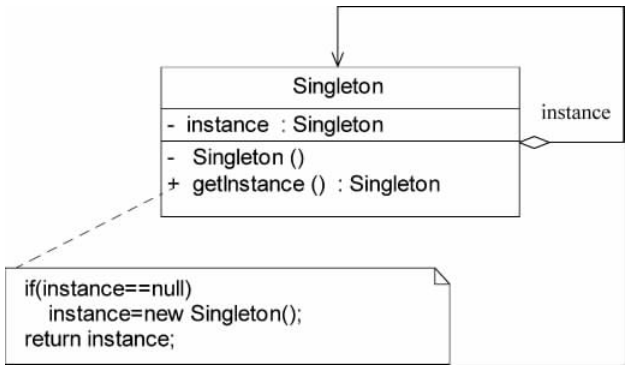

图 3-6 单例模式结构图

用与单例控制相似的方法来获得指定个数的实例。单例模式包括懒汉式单例和饿汉式单例两种实现方式,其中懒汉式单例是在第一次调用工厂方法 getInstance()时创建单例对象,而饿汉式单例是在类加载时创建单例对象,即在声明静态单例对象时实例化单例类,图 3-6 所示结构图为懒汉式单例。

3.2 实训实例

下面结合应用实例来学习如何在软件开发中使用创建型设计模式。

3.2.1 简单工厂模式实例之图形工厂

1. 实例说明

使用简单工厂模式设计一个可以创建不同几何形状(Shape)的绘图工具类,如可创建圆形(Circle)、矩形(Rectangle)和三角形(Triangle)对象,每个几何图形均具有绘制 draw()和擦除 erase()两个方法,要求在绘制不支持的几何图形时,抛出一个 UnsupportedShapeException 异常,绘制类图并编程实现。

2. 实例类图

本实例类图如图 3-7 所示。

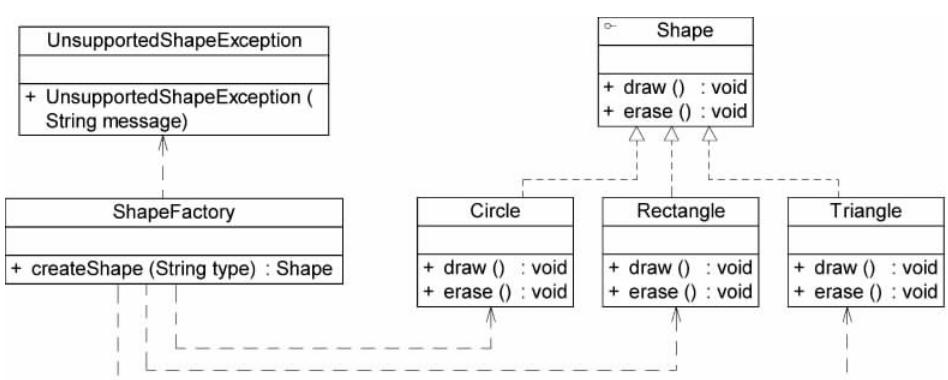

图 3-7 图形工厂实例类图

3. 实例代码

在本实例中,Shape 接口充当抽象产品,其子类 Circle、Rectangle 和 Triangle 等充当具体产品,ShapeFactory 充当工厂类。本实例代码如下:

```
//形状接口: 抽象产品
interface Shape
{
    public void draw();
    public void erase();
}

//圆形类: 具体产品
class Circle implements Shape
{
    public void draw()
    {
        System.out.println("绘制圆形");
    }
    public void erase()
    {
        System.out.println("删除圆形");
    }
}

//矩形类: 具体产品
class Rectangle implements Shape
{
    public void draw()
    {
        System.out.println("绘制矩形");
    }
    public void erase()
    {
        System.out.println("删除矩形");
    }
}

//三角形类: 具体产品
class Triangle implements Shape
{
    public void draw()
    {
        System.out.println("绘制三角形");
    }
    public void erase()
    {
        System.out.println("删除三角形");
    }
```

```
}

//形状工厂类：工厂
class ShapeFactory
{
    //静态工厂方法
    public static Shape createShape(String type) throws UnsupportedShapeException
    {
        if(type.equalsIgnoreCase("c"))
        {
            return new Circle();
        }
        else if(type.equalsIgnoreCase("r"))
        {
            return new Rectangle();
        }
        else if(type.equalsIgnoreCase("t"))
        {
            return new Triangle();
        }
        else
        {
            throw new UnsupportedShapeException("不支持该形状!");
        }
    }
}

//自定义异常类
class UnsupportedShapeException extends Exception
{
    public UnsupportedShapeException(String message)
    {
        super(message);
    }
}
```

客户端测试类代码如下：

```
//客户端测试类
class Client
{
    public static void main(String args[])
    {
         try
         {
             Shape shape;
             shape = ShapeFactory.createShape("r");
             shape.draw();
```

```
                shape.erase();
            }
            catch(UnsupportedShapeException e)
            {
                System.out.println(e.getMessage());
            }
        }
    }
```

运行结果如下：

```
绘制矩形
删除矩形
```

如果将静态方法 createShape()中的参数改为 a，则将抛出异常，输出“不支持该形状”。如果需要更换产品，只需改变静态工厂方法中的参数即可，不同的参数对应不同的产品；如果需要增加新的具体产品，则需要修改工厂中的静态工厂方法，因此增加新产品时将违背开闭原则。

3.2.2 工厂方法模式实例之日志记录器

1. 实例说明

某系统日志记录器要求支持多种日志记录方式，如文件日志记录(FileLog)、数据库日志记录(DatabaseLog)等，且用户可以根据要求动态选择日志记录方式，现使用工厂方法模式设计该系统。

2. 实例类图

本实例类图如图 3-8 所示。

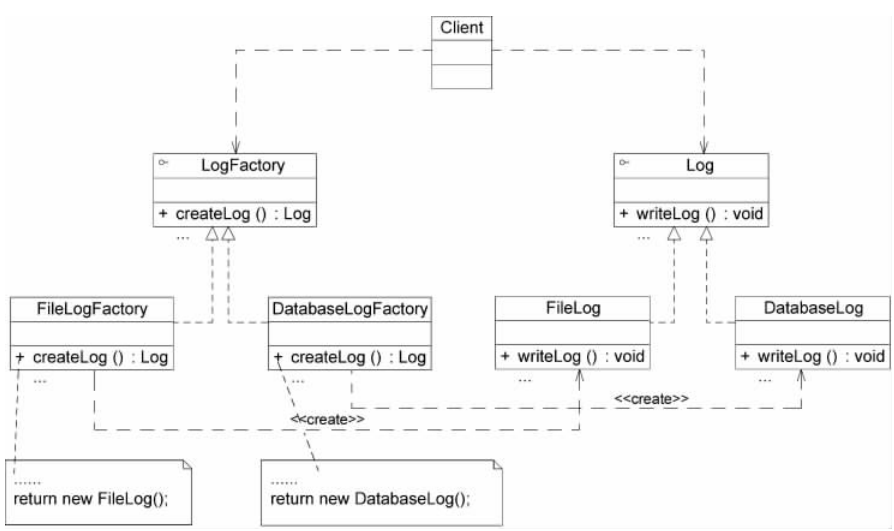

图 3-8 日志记录器实例类图

3. 实例代码

在本实例中，Log 接口充当抽象产品，其子类 FileLog 和 DatabaseLog 充当具体产品，LogFactory 接口充当抽象工厂，其子类 FileLogFactory 和 DatabaseLogFactory 充当具体工厂。本实例代码如下：

```
//日志记录器接口：抽象产品
interface Log
{
    public void writeLog();
}

//文件日志记录器：具体产品
class FileLog implements Log
{
    public void writeLog()
    {
        System.out.println("文件日志记录。");
    }
}

//数据库日志记录器：具体产品
class DatabaseLog implements Log
{
    public void writeLog()
    {
        System.out.println("数据库日志记录。");
    }
}

//日志记录器工厂接口：抽象工厂
interface LogFactory
{
    public Log createLog();
}

//文件日志记录器工厂类：具体工厂
class FileLogFactory implements LogFactory
{
    public Log createLog()
    {
        return new FileLog();
    }
}

//数据库日志记录器工厂类：具体工厂
class DatabaseLogFactory implements LogFactory
{
    public Log createLog()
```

```
        {
            return new DatabaseLog();
        }
    }
```

客户端测试类代码如下：

```
//客户端测试类
class Client
{
    public static void main(String args[])
    {
        LogFactory factory;
        Log log;
        factory = new FileLogFactory();
        log = factory.createLog();
        log.writeLog();
    }
}
```

运行结果如下：

```
文件日志记录。
```

在本实例中，可以将具体工厂类的类名存储在配置文件（如 XML 文件）中，再通过 DOM 和反射机制来生成工厂对象，增加新的具体产品只需对应增加新的具体工厂即可，如果需要更换产品，只需修改配置文件中的具体工厂类类名，无须修改源代码，符合开闭原则。

3.2.3 抽象工厂模式实例之数据库操作工厂

1. 实例说明

某系统为了改进数据库操作的性能，自定义数据库连接对象 Connection 和语句对象 Statement，可针对不同类型的数据库提供不同的连接对象和语句对象，如提供 Oracle 或 MySQL 专用连接类和语句类，而且用户可以通过配置文件等方式根据实际需要动态更换系统数据库。使用抽象工厂模式设计该系统。

2. 实例类图

本实例类图如图 3-9 所示。

3. 实例代码

在本实例中，接口 DBFactory 充当抽象工厂，其子类 OracleFactory 和 MySQLFactory 充当具体工厂，接口 Connection 和 Statement 充当抽象产品，其子类 OracleConnection、MySQLConnection 和 OracleStatement、MySQLStatement 充当具体产品。本实例代码如下：

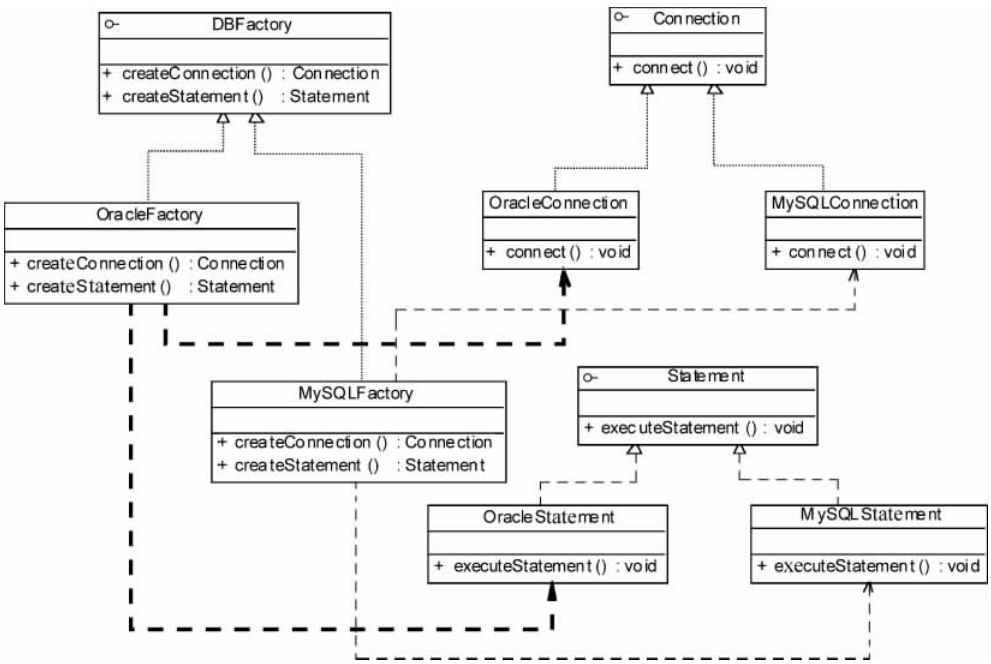

图 3-9　数据库操作工厂实例类图

```
//数据库连接接口：抽象产品
interface Connection
{
    public void connect();
}

//Oracle 数据库连接类：具体产品
class OracleConnection implements Connection
{
    public void connect()
    {
        System.out.println("连接 Oracle 数据库。");
    }
}

//MySQL 数据库连接类：具体产品
class MySQLConnection implements Connection
{
    public void connect()
    {
        System.out.println("连接 MySQL 数据库。");
    }
}

//数据库语句接口：抽象产品
```

```
interface Statement
{
    public void executeStatement();
}

//Oracle 数据库语句类：具体产品
class OracleStatement implements Statement
{
    public void executeStatement()
    {
        System.out.println("执行 Oracle 数据库语句。");
    }
}

//MySQL 数据库语句类：具体产品
class MySQLStatement implements Statement
{
    public void executeStatement()
    {
        System.out.println("执行 MySQL 数据库语句。");
    }
}

//数据库工厂接口：抽象工厂
interface DBFactory
{
    public Connection createConnection();
    public Statement createStatement();
}

//Oracle 数据库工厂：具体工厂
class OracleFactory implements DBFactory
{
    public Connection createConnection()
    {
        return new OracleConnection();
    }

    public Statement createStatement()
    {
        return new OracleStatement();
    }
}

//MySQL 数据库工厂：具体工厂
class MySQLFactory implements DBFactory
{
    public Connection createConnection()
    {
```

```
            return new MySQLConnection();
        }

        public Statement createStatement()
        {
            return new MySQLStatement();
        }
    }
```

客户端测试类代码如下：

```
//客户端测试类
class Client
{
    public static void main(String args[])
    {
        DBFactory factory;
        Connection connection;
        Statement statement;
        factory = new OracleFactory();
        connection = factory.createConnection();
        statement = factory.createStatement();
        connection.connect();
        statement.executeStatement();
    }
}
```

运行结果如下：

```
连接 Oracle 数据库。
执行 Oracle 数据库语句。
```

与工厂方法模式一样，在本实例中，可以将具体工厂类的类名存储在配置文件（如 XML 文件）中，再通过 DOM 和反射机制来生成工厂对象。在增加新的具体产品族时只需对应增加新的具体工厂即可，如果需要更换一个产品族，只需修改配置文件中的具体工厂类类名，无须修改源代码，符合开闭原则。但是如果要增加新的产品等级结构，每个工厂要生产一个新的类型的对象，则需要修改抽象工厂接口和所有的具体工厂类，从这个角度来看，抽象工厂模式违背了开闭原则。因此在使用抽象工厂模式时需要仔细分析所有的产品结构，避免在设计完成之后修改源代码，特别是抽象层的代码。

3.2.4 建造者模式实例之游戏人物角色

1. 实例说明

某游戏软件中人物角色包括多种类型，不同类型的人物角色，其性别、脸型、服装、发型等外部特性有所差异，使用建造者模式创建人物角色对象，要求绘制类图并编程实现。

2. 实例类图

本实例类图如图 3-10 所示。

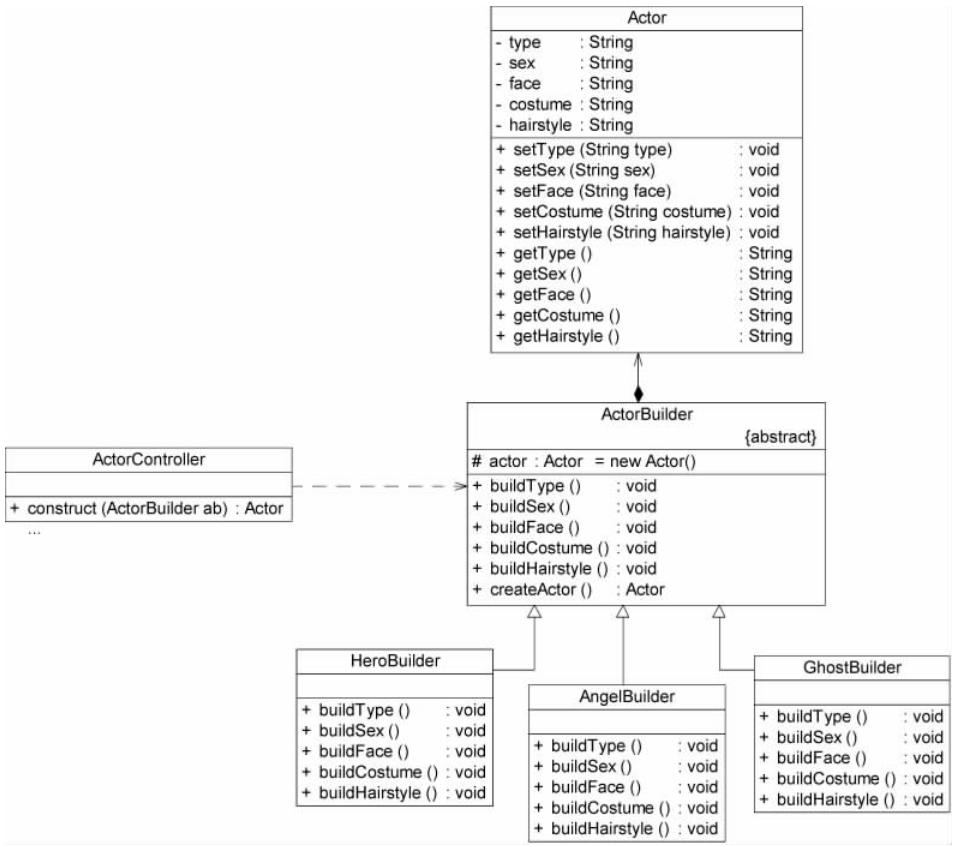

图 3-10　游戏人物角色实例类图

3. 实例代码

在本实例中，ActorController 充当指挥者，ActorBuilder 充当抽象建造者，HeroBuilder、AngelBuilder 和 GhostBuilder 充当具体建造者，Actor 充当复合产品。本实例代码如下：

```
//Actor 角色类: 复合产品
class Actor
{
    private String type;
    private String sex;
    private String face;
    private String costume;
    private String hairstyle;

    public void setType(String type) {
        this.type = type;
    }
```

```
    public void setSex(String sex) {
        this.sex = sex;
    }

    public void setFace(String face) {
        this.face = face;
    }

    public void setCostume(String costume) {
        this.costume = costume;
    }

    public void setHairstyle(String hairstyle) {
        this.hairstyle = hairstyle;
    }

    public String getType() {
        return (this.type);
    }

    public String getSex() {
        return (this.sex);
    }

    public String getFace() {
        return (this.face);
    }

    public String getCostume() {
        return (this.costume);
    }

    public String getHairstyle() {
        return (this.hairstyle);
    }
}

//角色建造器：抽象建造者
abstract class ActorBuilder
{
    protected Actor actor = new Actor();

    public abstract void buildType();
    public abstract void buildSex();
    public abstract void buildFace();
    public abstract void buildCostume();
    public abstract void buildHairstyle();
```

```
    public Actor createActor()
    {
        return actor;
    }
}

//英雄角色建造器：具体建造者
class HeroBuilder extends ActorBuilder
{
    public void buildType()
    {
        actor.setType("英雄");
    }
    public void buildSex()
    {
        actor.setSex("男");
    }
    public void buildFace()
    {
        actor.setFace("英俊");
    }
    public void buildCostume()
    {
        actor.setCostume("盔甲");
    }
    public void buildHairstyle()
    {
        actor.setHairstyle("飘逸");
    }
}

//天使角色建造器：具体建造者
class AngelBuilder extends ActorBuilder
{
    public void buildType()
    {
        actor.setType("天使");
    }
    public void buildSex()
    {
        actor.setSex("女");
    }
    public void buildFace()
    {
        actor.setFace("漂亮");
    }
    public void buildCostume()
```

```
        {
            actor.setCostume("白裙");
        }
        public void buildHairstyle()
        {
            actor.setHairstyle("披肩长发");
        }
    }

    //魔鬼角色建造器：具体建造者
    class GhostBuilder extends ActorBuilder
    {
        public void buildType()
        {
            actor.setType("魔鬼");
        }
        public void buildSex()
        {
            actor.setSex("妖");
        }
        public void buildFace()
        {
            actor.setFace("丑陋");
        }
        public void buildCostume()
        {
            actor.setCostume("黑衣");
        }
        public void buildHairstyle()
        {
            actor.setHairstyle("光头");
        }
    }
    //Actor角色创建控制器：指挥者
    class ActorController
    {
        public Actor construct(ActorBuilder ab)
        {
            Actor actor;
            ab.buildType();
            ab.buildSex();
            ab.buildFace();
            ab.buildCostume();
            ab.buildHairstyle();
            actor = ab.createActor();
            return actor;
        }
    }
```

客户端测试类代码如下：

```
//客户端测试类
class Client
{
    public static void main(String args[])
    {
        ActorController ac = new ActorController();
        ActorBuilder ab;
        ab = new AngelBuilder();
        Actor angel;
        angel = ac.construct(ab);
        String type = angel.getType();
        System.out.println(type + "的外观：");
        System.out.println("性别：" + angel.getSex());
        System.out.println("面容：" + angel.getFace());
        System.out.println("服装：" + angel.getCostume());
        System.out.println("发型：" + angel.getHairstyle());
    }
}
```

运行结果如下：

```
天使的外观：
性别：女
面容：漂亮
服装：白裙
发型：披肩长发
```

在建造者模式中，客户端只需实例化指挥者类即可，指挥者类针对抽象建造者编程，客户端根据需要传入具体建造者对象，具体建造者将一步一步构造一个完整的产品。相同的构造过程可以创建不同的产品，在本实例中，通过选择不同的具体建造者类，可以返回不同的角色。为了提高系统的可扩展性，可将具体建造者类类名存储在配置文件中。

3.2.5 原型模式实例之快速创建工作周报

1. 实例说明

在某 OA 系统中，用户可以创建工作周报，由于某些岗位每周工作存在重复性，因此可以通过复制原有工作周报并进行局部修改来快速新建工作周报。现使用原型模式来实现该功能，绘制类图并编程实现。

2. 实例类图

本实例类图如图 3-11 所示。

3. 实例代码

在本实例中，WeeklyLog 充当具体原型类，Object 类充当抽象原型类，clone()方法为原

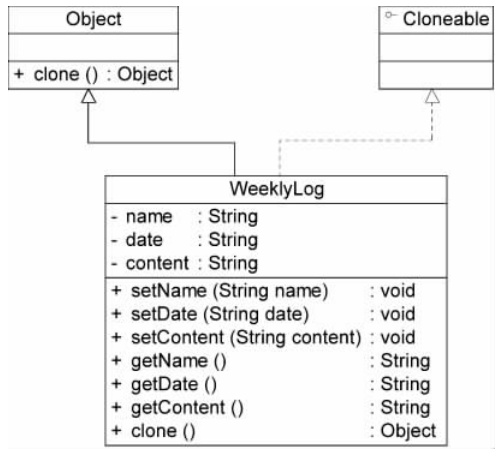

图 3-11 快速创建工作周报实例类图

型方法。本实例代码如下所示：

```
//工作周报：具体原型类
class WeeklyLog implements Cloneable
{
    private String name;
    private String date;
    private String content;
    public void setName(String name) {
        this.name = name;
    }
    public void setDate(String date) {
        this.date = date;
    }
    public void setContent(String content) {
        this.content = content;
    }
    public String getName() {
        return (this.name);
    }
    public String getDate() {
        return (this.date);
    }
    public String getContent() {
        return (this.content);
    }
    //克隆方法 clone(),此处使用 Java 语言提供的浅克隆机制
    public Object clone()
    {
        Object obj = null;
        try
        {
            obj = super.clone();
```

```
            return obj;
        }
        catch(CloneNotSupportedException e)
        {
            System.out.println("不能复制!");
            return null;
        }
    }
}
```

客户端测试类代码如下：

```
//客户端测试类
class Client
{
    public static void main(String args[])
    {
        WeeklyLog log_previous = new WeeklyLog();
        log_previous.setName("张三");
        log_previous.setDate("2017 年第 12 周");
        log_previous.setContent("这周工作很忙,每天加班!");

        System.out.println(" **** 周报 **** ");
        System.out.println(log_previous.getDate());
        System.out.println(log_previous.getName());
        System.out.println(log_previous.getContent());
        System.out.println(" ---------------------------------- ");

        WeeklyLog log_now;
        log_now = (WeeklyLog)log_previous.clone();
        log_now.setDate("2017 年第 13 周");
        System.out.println(" **** 周报 **** ");
        System.out.println(log_now.getDate());
        System.out.println(log_now.getName());
        System.out.println(log_now.getContent());
    }
}
```

运行结果如下：

```
 **** 周报 ****
2017 年第 12 周
张三
这周工作很忙,每天加班!
 ----------------------------------
 **** 周报 ****
2017 年第 13 周
```

```
张三
这周工作很忙,每天加班!
```

在本实例中使用了 Java 语言内置的浅克隆机制,通过继承 Object 类的 clone()方法实现对象的复制,原始对象和克隆得到的对象在内存中是两个完全不同的对象,通过已创建的工作周报可以快速创建新的周报,然后再根据需要修改周报,无须再从头开始创建。原型模式为工作流系统中任务单的快速生成提供了一种解决方案。

3.2.6　单例模式实例之多文档窗口

1. 实例说明

使用单例模式设计一个多文档窗口(注:在 Java AWT/Swing 开发中可使用 JDesktopPane 和 JInternalFrame 来实现),要求在主窗体中某个内部子窗体只能实例化一次,即只能弹出一个相同的子窗体,如图 3-12 所示。

图 3-12　多文档窗口示意图

2. 实例类图

本实例类图如图 3-13 所示。

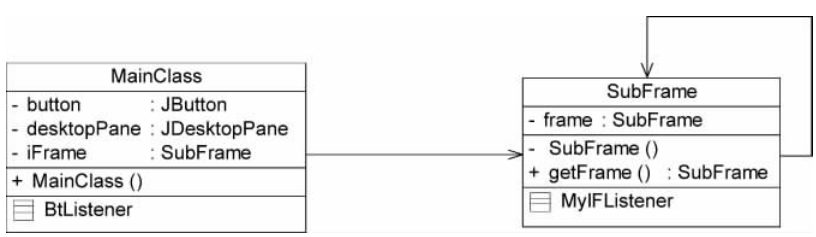

图 3-13　多文档窗口实例类图

3. 实例代码

在本实例中,SubFrame 类充当单例类,在其中定义了静态工厂方法 getFrame()。本实例代码如下:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

//子窗口：单例类
class SubFrame extends JInternalFrame
{
    private static SubFrame frame;                                  //静态实例

    //私有构造函数
    private SubFrame()
    {
        super("子窗体", true, true, true, false);
        this.setLocation(20,20);                                    //设置内部窗体位置
        this.setSize(200,200);                                      //设置内部窗体大小
        this.addInternalFrameListener(new MyIFListener());          //监听窗体事件
        this.setVisible(true);
    }

    //工厂方法,返回窗体实例
    public static SubFrame getFrame()
    {
        //如果窗体对象为空,则创建窗体,否则直接返回已有窗体
        if(frame == null)
        {
            frame = new SubFrame();
        }
        return frame;
    }

    //事件监听器
    class MyIFListener extends InternalFrameAdapter
    {
        //子窗体关闭时,将窗体对象设为 null
        public void internalFrameClosing(InternalFrameEvent e)
        {
            if(frame != null)
            {
                frame = null;
            }
        }
    }
}

//客户端测试类
class MainClass extends JFrame
{
    private JButton button;
```

```
    private JDesktopPane desktopPane;
    private SubFrame iFrame = null;

    public MainClass()
    {
        super("主窗体");
        Container c = this.getContentPane();
        c.setLayout(new BorderLayout());

        button = new JButton("单击创建一个内部窗体");
        button.addActionListener(new BtListener());
        c.add(button, BorderLayout.SOUTH);

        desktopPane = new JDesktopPane();     //创建 DesktopPane
        c.add(desktopPane);

        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setLocationRelativeTo(null);
        this.setSize(400,400);
        this.show();
    }

    //事件监听器
    class BtListener implements ActionListener
    {
        public void actionPerformed(ActionEvent e)
        {
            if(iFrame != null)
            {
                desktopPane.remove(iFrame);
            }
            iFrame = SubFrame.getFrame();
            desktopPane.add(iFrame);
        }
    }

    public static void main(String[] args)
    {
        new MainClass();
    }
}
```

在本实例中，SubFrame 类是 JInternalFrame 类的子类，在 SubFrame 类中定义了一个静态的 SubFrame 类型的实例变量，在静态工厂方法 getFrame()中创建了 SubFrame 对象并将其返回。在 MainClass 类中使用了该单例类，确保子窗口在当前应用程序中只有唯一实例，即只能弹出一个子窗口。

3.3 实训练习

1. 选择题

(1) 在面向对象软件开发过程中,采用设计模式(　　)。

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实例对象的个数

D. 以保证程序的运行速度达到最优值

(2) 设计模式具有(　　)的优点。

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

(3) 在进行面向对象设计时,采用设计模式能够(　　)。

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

(4) 以下关于简单工厂模式的叙述错误的是(　　)。

A. 简单工厂模式可以根据参数的不同返回不同的类的实例

B. 简单工厂模式专门定义一个类来负责创建其他类的实例,被创建的实例通常都具有共同的父类

C. 简单工厂模式可以减少系统中类的个数,简化系统的设计,使得系统更易于理解

D. 系统的扩展困难,一旦添加新的产品就不得不修改工厂逻辑,违背了开闭原则

(5) 在简单工厂模式中,如果需要增加新的具体产品,必须修改(　　)的源代码。

A. 抽象产品类　　B. 其他具体产品类

C. 工厂类　　D. 客户类

(6) 关于 Java 语言实现简单工厂模式中的静态工厂方法,以下叙述错误的是(　　)。

A. 工厂子类可以继承父类非私有的静态方法

B. 工厂子类可以覆盖父类的静态方法

C. 工厂子类的静态工厂方法可以在运行时覆盖由工厂父类声明的工厂对象的静态工厂方法

D. 静态工厂方法支持重载

(7) 以下代码使用了(　　)模式。

```
abstract class ExchangeMethod
{
    public abstract void process();
```

```
}

class DigitalCurrency extends ExchangeMethod
{
  public void process()
  { … }
}

class CreditCard extends ExchangeMethod
{
    public void process()
    { … }
}
⋮
class Factory
{
    public static ExchangeMethod createProduct(String type)
    {
      switch(type)
      {
        case "DigitalCurrency":
        return new DigitalCurrency(); break;
        case "CreditCard":
        return new CreditCard(); break;
        ⋮
    }
  }
}
```

A. Simple Factory　　B. Factory Method

C. Abstract Factory　　D. 未用任何设计模式

(8) 图 3-14 是(　　)模式的结构图。

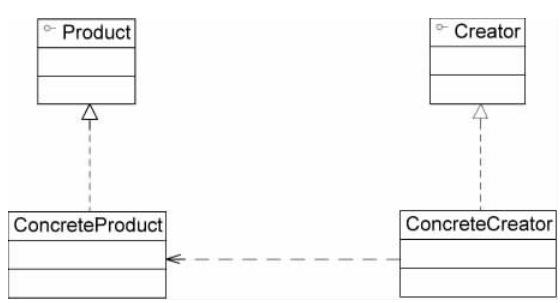

图 3-14　选择题(8)用图

A. Abstract Factory　　B. Factory Method

C. Command　　D. Chain of Responsibility

(9) 以下关于工厂方法模式的叙述错误的是(　　)。

A. 在工厂方法模式中引入了抽象工厂类,而具体产品的创建延迟到具体工厂中

实现

B. 工厂方法模式添加新的产品对象很容易，无须对原有系统进行修改，符合开闭原则

C. 工厂方法模式存在的问题是在添加新产品时，需要编写新的具体产品类，而且还要提供与之对应的具体工厂类，随着类个数的增加，会给系统带来一些额外开销

D. 工厂方法模式是所有形式的工厂模式中最为抽象和最具一般性的一种形态，工厂方法模式退化后可以演变成抽象工厂模式

(10) 某银行系统采用工厂模式描述其不同账户之间的关系，设计出的类图如图 3-15 所示。其中与工厂模式中的 Creator 角色相对应的类是(①)；与 Product 角色相对应的类是(②)。

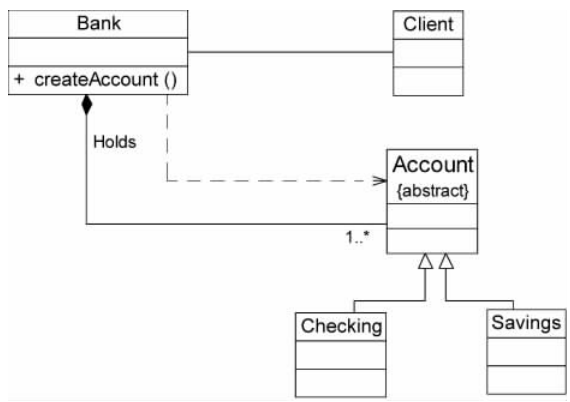

图 3-15 选择题(10)用图

① A. Bank B. Account C. Checking D. Savings

② A. Bank B. Account C. Checking D. Savings

(11) 以下选项( )可作为工厂方法模式的应用实例。

A. 曲线图创建器生成曲线图，柱状图创建器生成柱状图

B. 通过复制已有邮件对象创建新的邮件对象

C. 在网络上传输大图片时，先传输对应的文字描述，再传输真实的图片

D. 在多个界面组件类之间添加一个控制类来协调它们之间的相互调用关系

(12) 不同品牌的手机应该由不同的公司制造，苹果公司生产苹果手机，三星公司生产三星手机，该场景蕴含了( )设计模式。

A. Simple Factory B. Factory Method

C. Abstract Factory D. Builder

(13) 以下关于抽象工厂模式的叙述错误的是( )。

A. 抽象工厂模式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

B. 当系统中有多于一个产品族时可以考虑使用抽象工厂模式

C. 当一个工厂等级结构可以创建出分属于不同产品等级结构的一个产品族中的

所有对象时，抽象工厂模式比工厂方法模式更为简单、更有效率

D. 抽象工厂模式符合开闭原则，增加新的产品族和新的产品等级结构都很方便

(14) 关于抽象工厂模式的叙述，以下错误的一项是(　　)。

A. 抽象工厂模式提供了创建一系列相关或相互依赖的对象的接口，而无须指定这些对象所属的具体类

B. 抽象工厂模式可应用于一个系统要由多个产品系列中的一个来配置的时候

C. 抽象工厂模式可应用于强调一系列相关产品对象的设计以便进行联合使用的时候

D. 抽象工厂模式可应用于希望使用已经存在的类，但其接口不符合需求的时候

(15) 关于抽象工厂模式中的产品族和产品等级结构叙述错误的是(　　)。

A. 产品等级结构是从不同的产品族中任意选取产品组成的层次结构

B. 产品族是指位于不同产品等级结构、功能相关的产品组成的家族

C. 一个具体工厂可以创建出分属于不同产品等级结构的一个产品族中的所有对象

D. 工厂方法模式对应一个产品等级结构，而抽象工厂模式则需要面对多个产品等级结构

(16) 某公司欲开发一个图表显示系统，在该系统中，曲线图生成器可以创建曲线图、曲线图图例和曲线图数据标签，柱状图生成器可以创建柱状图、柱状图图例和柱状图数据标签。用户要求可以很方便地增加新的类型的图形，系统需具备较好的可扩展能力。针对这种需求，公司采用(　　)最为恰当。

A. 桥接模式　　B. 适配器模式

C. 策略模式　　D. 抽象工厂模式

(17) 关于工厂模式的陈述，以下有误的一项是(　　)。

A. 工厂模式隔离产品的创建和使用

B. 在工厂类中封装产品对象的创建细节，客户类无须关心这些细节

C. 工厂方法模式中的工厂方法可以改为静态方法

D. 工厂方法模式中抽象工厂声明的工厂方法返回抽象产品类型，不能返回具体产品类型

(18) 关于建造者模式中的 Director 类描述错误的是(　　)。

A. Director 类隔离了客户类及生产过程

B. 在建造者模式中，客户类通过 Director 逐步构造一个复杂对象

C. Director 类构建一个抽象建造者 Builder 子类的对象

D. Director 与抽象工厂模式中的工厂类类似，负责返回一个产品族中的所有产品

(19) 以下关于建造者模式的叙述错误的是(　　)。

A. 建造者模式将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

B. 建造者模式允许用户可以只通过指定复杂对象的类型和内容就可以创建它们，而不需要知道内部的具体构建细节

C. 当需要生成的产品对象有复杂的内部结构时可以考虑使用建造者模式

D. 在建造者模式中,各个具体的建造者相互之间有较强的依赖关系,可通过指挥者类组装成一个完整的产品对象返回给客户

(20) 当需要创建的产品具有复杂的内部结构时,为了逐步构造完整的对象,并使得对象的创建更具弹性,可以使用(　　)。

A. 抽象工厂模式　　B. 原型模式

C. 建造者模式　　D. 单例模式

(21) 以下关于原型模式的叙述错误的是(　　)。

A. 原型模式通过给出一个原型对象来指明所要创建的对象的类型,然后用复制这个原型对象的办法创建出更多同类型的对象

B. 浅克隆仅仅复制所考虑的对象,而不复制它所引用的对象,也就是其中的成员对象并不复制

C. 在原型模式中实现深克隆时通常需要编写较为复杂的代码

D. 在原型模式中不需要为每一个类配备一个克隆方法,因此对于原型模式的扩展很灵活,对于已有类的改造也较为容易

(22) 以下关于 Java 语言中 clone()方法的使用错误的是(　　)。

A. 对于任何对象 x,都有 x.clone()==x

B. 对于任何对象 x,都有 x.clone().getClass()==x.clone().getClass()

C. 在子类的 clone()方法中可以通过调用 super.clone()来实现自我复制

D. 支持浅克隆的类必须实现 Cloneable 接口,否则将抛出 CloneNotSupportedException 异常

(23) 某公司欲开发一个即时聊天软件,用户在聊天过程中可以与多位好友同时聊天,在私聊时将产生多个聊天窗口,在创建聊天窗口时为了提高效率,要求根据第一个窗口快速创建其他窗口。针对这种需求,采用(　　)最为恰当。

A. 享元模式　　B. 单例模式

C. 原型模式　　D. 组合模式

(24) 在(　　)时可使用单例模式。

A. 隔离菜单项对象的创建和使用

B. 防止一个资源管理器窗口被实例化多次

C. 使用一个已有的查找算法而不想修改既有代码

D. 不能创建子类,需要扩展一个数据过滤类

(25) 以下关于单例模式的描述,正确的是(　　)。

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

(26) (　　)限制了创建类的实例数量。

A. 原型模式　　B. 建造者模式

C. 策略模式　　D. 单例模式

(27) 以下(　　)不是单例模式的要点。

A. 某个类只能有一个实例　　B. 单例类不能被继承

C. 必须自行创建单个实例　　D. 必须自行向整个系统提供这个单例

(28) 某软件公司开发了一组加密类,在使用这些加密类时欲采用简单工厂模式进行设计,为了减少类的个数,将工厂类和抽象加密类合并,基本 UML 类图如图 3-16 所示。

以下陈述错误的是(　　)。

A. 在类图中,Cipher 类既充当抽象产品类,又充当工厂类

B. 工厂方法 createCipher( ) 的返回类型为 Cipher

C. 工厂方法 createCipher( ) 应定义为静态方法

D. Cipher 类中的 encrypt( ) 方法必须为抽象方法

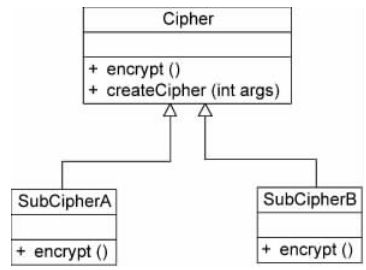

图 3-16　选择题(28)用图

(29) (　　)设计模式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。

A. Adapter(适配器)　　B. Singleton(单例)

C. Abstract Factory(抽象工厂)　　D. Template Method(模板方法)

(30) (　　)设计模式将一个复杂对象的构建与它的表示分离,使同样的构建过程可以创建不同的表示。

A. Builder(建造者)　　B. Factory Method(工厂方法)

C. Prototype(原型)　　D. Facade(外观)

2. 填空题

(1) 某系统提供一个简单计算器,具有简单的加法和减法功能,系统可以根据用户的选择实例化相应的操作类。现使用简单工厂模式设计该系统,类图如图 3-17 所示。

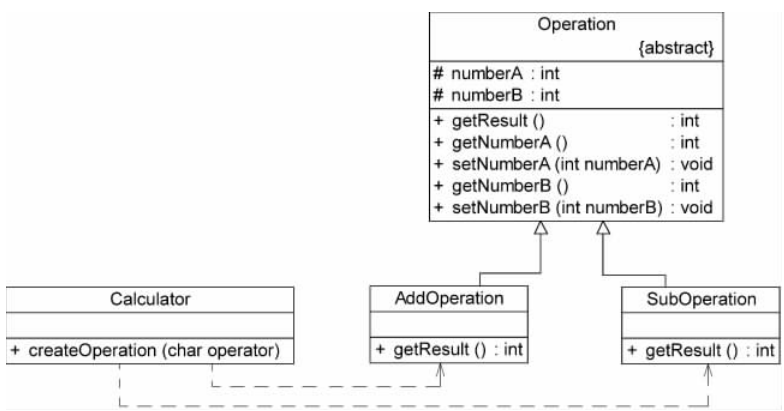

图 3-17　填空题(1)用图

在图 3-17 中,Operation 是抽象类,其中定义了抽象方法 getResult( ),其子类 AddOperation 用于实现加法操作,SubOperation 用于实现减法操作。Calculator 是简单工

厂类，工厂方法为 createOperation()，该方法接收一个 char 类型的字符参数，如果传入的参数为“+”，工厂方法返回一个 AddOperation 类型的对象，如果传入的参数为“-”，则返回一个 SubOperation 类型的对象。

Java 代码如下：

```
abstract class Operation
{
    protected int numberA;
    protected int numberB;
    //numberA 和 numberB 的 Setter 方法和 Getter 方法省略
    public    ①    int getResult();
}

class AddOperation extends Operation
{
    public int getResult()
    {
        return numberA + numberB;
    }
}

class SubOperation extends Operation
{
    public int getResult()
    {
        return numberA - numberB;
    }
}

class Calculator
{
    public    ②    createOperation(char operator)
    {
        Operation op = null;
        ③
        {
            case '+':
                op =    ④    ;
                break;
            case '-':
                op =    ⑤    ;
                break;
        }
        ⑥    ;
    }
}

class Test
```

```
{
    public static void main(String args[])
    {
        int result;
        Operation op1 = Calculator.createOperation( ' + ');
        op1.setNumberA(20);
        op1.setNumberB(10);
        result = ____⑦____;
        System.out.println(result);
    }
}
```

（2）某软件公司欲开发一个数据格式转换工具，可以将不同数据源如 TXT 文件、数据库、Excel 表格中的数据转换成 XML 格式。为了让系统具有更好的扩展性，在未来支持新类型的数据源，开发人员使用工厂方法模式设计该转换工具的核心类。在工厂类中封装了具体转换类的初始化和创建过程，客户端只需使用工厂类即可获得具体的转换类对象，再调用其相应方法实现数据转换操作，其类图如图 3-18 所示。

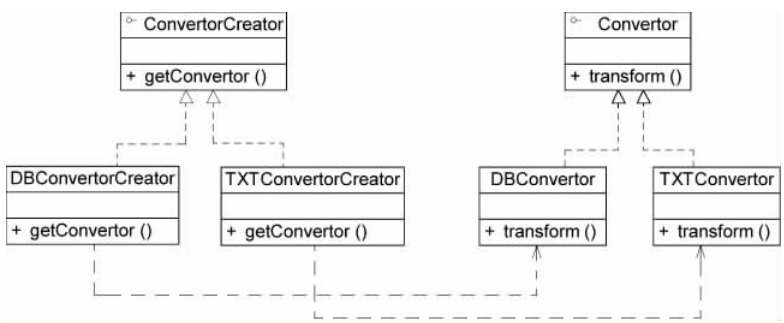

图 3-18　填空题(2)用图

在图 3-18 中，ConvertorCreator 是抽象工厂接口，它声明了工厂方法 getConvertor()，在其子类中实现该方法，用于创建具体的转换对象；Convertor 是抽象产品接口，它声明了抽象数据转换方法 transform()，在其子类中实现该方法，用于完成具体的数据转换操作。类 DBConvertor 和 TXTConvertor 分别用于将数据库中的数据和 TXT 文件中的数据转换为 XML 格式。

Java 代码如下：

```
interface ConvertorCreator
{
            ____①____;
}

interface Convertor
{
          public String transform();
```

```
}

class DBConvertorCreator implements ConvertorCreator
{
          public Convertor getConvertor()
          {
          _____②_____;
          }
}

class TXTConvertorCreator implements ConvertorCreator
{
        public Convertor getConvertor()
        {
        _____③_____;
        }
}

class DBConvertor implements Convertor
{
        public String transform()
        {
          //实现代码省略
        }
}

class TXTConvertor implements Convertor
{
        public String transform()
        {
          //实现代码省略
        }
}

class Test
{
        public static void main(String args[])
        {
          ConvertorCreator creator;
          _____④_____;
          creator = new DBConvertorCreator();
          convertor = _____⑤_____;
          convertor.transform();
    }
}
```

如果需要针对一种新的数据源进行数据转换,该系统至少需要增加_____⑥_____个类。工厂方法模式体现了以下面向对象设计原则中的_____⑦_____(多选)。

A. 开闭原则　　B. 依赖倒转原则　　C. 接口隔离原则　　D. 单一职责原则

E. 合成复用原则

(3) 某手机游戏软件公司欲推出一款新的游戏软件,该软件能够支持 Symbian、

Android 和 Windows Mobile 等多个主流的手机操作系统平台，针对不同的手机操作系统，该游戏软件提供了不同的游戏操作控制类和游戏界面控制类，并提供相应的工厂类来封装这些类的初始化。软件要求具有较好的扩展性以支持新的操作系统平台，为了满足上述需求，采用抽象工厂模式进行设计所得类图如图 3-19 所示。

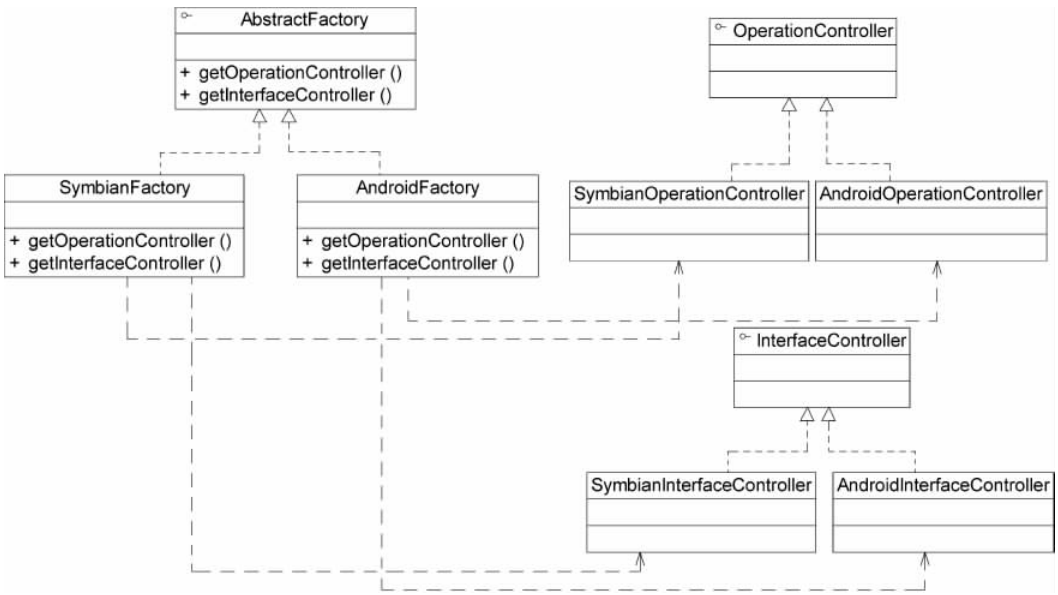

图 3-19 填空题(3)用图

在该设计方案中，具体工厂类如 SymbianFactory 用于创建 Symbian 操作系统平台下的游戏操作控制类 SymbianOperationController 和游戏界面控制类 SymbianInterfaceController，再通过它们的业务方法来实现对游戏软件的初始化和运行控制。

Java 代码如下：

```
interface AbstractFactory
{
    public OperationController getOperationController();
    public InterfaceController getInterfaceController();
}

interface OperationController
{
    public void init();
    //其他方法声明省略
}

interface InterfaceController
{
    public void init();
    //其他方法声明省略
}

class SymbianFactory implements AbstractFactory
```

```
{
    public OperationController getOperationController()
    {
    ________①________;
    }

    public InterfaceController getInterfaceController()
    {
    ________②________;
    }
}

class AndroidFactory ________③________
{
    public OperationController getOperationController()
    {
        return new AndroidOperationController();
    }

    public InterfaceController getInterfaceController()
    {
        return new AndroidInterfaceController();
    }
}

class SymbianOperationController ________④________
{
    public void init() {
        //实现代码省略
    }
    //其他方法声明省略
}

class AndroidOperationController ________⑤________
{
    public void init() {
        //实现代码省略
    }
    //其他方法声明省略
}

class SymbianInterfaceController implements InterfaceController
{
    public void init() {
        //实现代码省略
    }
    //其他方法声明省略
}

class AndroidInterfaceController implements InterfaceController
{
    public void init() {
```

```
            //实现代码省略
        }
        //其他方法声明省略
    }

    class Test
    {
        public static void main(String args[])
        {
            AbstractFactory af;
            _____⑥_____ oc;
            _____⑦_____ ic;
            af = new SymbianFactory();
            oc = _____⑧_____ ;
            ic = _____⑨_____;
            oc.init();
            ic.init();
        }
    }
```

在上述设计方案中怎样增加对 Windows Mobile 操作系统的支持？需对该设计方案进行哪些调整？简单说明实现过程。

（4）某软件公司欲开发一个音频和视频播放软件，为了给用户使用提供方便，该播放软件提供了多种界面显示模式，如完整模式、精简模式、记忆模式、网络模式等。在不同的显示模式下主界面的组成元素有所差异，如在完整模式下将显示菜单、播放列表、主窗口、控制条等，在精简模式下只显示主窗口和控制条，而在记忆模式下将显示主窗口、控制条、收藏列表等。现使用建造者模式设计该软件，所得类图如图 3-20 所示。

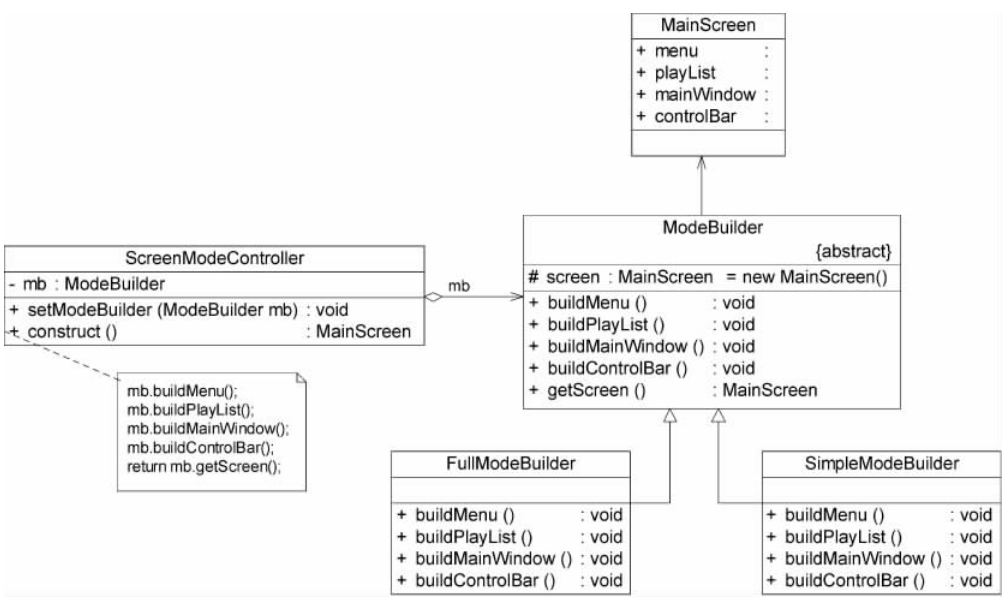

图 3-20　填空题(4)用图

在该设计方案中，MainScreen 是播放器的主界面，它是一个复合对象，包括菜单、播放列表、主窗口和控制条等成员。ModeBuilder 是一个抽象类，定义了一组抽象方法 buildXXX()用于逐步构造一个完整的 MainScreen 对象，getScreen()是工厂方法，用于返回一个构造好的 MainScreen 对象。ScreenModeController 充当指挥者，用于指导复合对象的创建，其中 construct()方法封装了具体创建流程，并向客户类返回完整的产品对象。

Java 代码如下：

```
class MainScreen
{
    public String menu;
    public String playList;
    public String mainWindow;
    public String controlBar;
}

        ①        class ModeBuilder
{
    protected MainScreen screen = new MainScreen();
    public abstract void buildMenu();
    public abstract void buildPlayList();
    public abstract void buildMainWindow();
    public abstract void buildControlBar();
    public MainScreen getScreen()
    {        ②        ; }
}

class FullModeBuilder extends ModeBuilder
{
    public void buildMenu() { //实现代码省略    }
    public void buildPlayList() { //实现代码省略 }
    public void buildMainWindow() { //实现代码省略 }
    public void buildControlBar() { //实现代码省略 }
}

class SimpleModeBuilder extends ModeBuilder
{
    public void buildMenu() { //实现代码省略}
    public void buildPlayList() { //实现代码省略 }
    public void buildMainWindow() { //实现代码省略 }
    public void buildControlBar() { //实现代码省略 }
}

class ScreenModeController
{
    private ModeBuilder mb;
    public void setModeBuilder(        ③        )
    {
        this.mb = mb;
```

```
        }
        public MainScreen construct()
        {
            MainScreen ms;
            mb.buildMenu();
            mb.buildPlayList();
            mb.buildMainWindow();
            mb.buildControlBar();
            ms = _____④_____;
            return ms;
        }
    }

    class Test
    {
        public static void main(String args[])
        {
            ScreenModeController smc = _____⑤_____;
            ModeBuilder mb;
            mb = new FullModeBuilder();  //构造完整模式界面
            MainScreen screen;
            smc.setModeBuilder(_____⑥_____);
            screen = _____⑦_____;
            System.out.println(screen.menu);
            //其他代码省略
        }
    }
```

（5）某数据处理软件需要增加一个图表复制功能，在图表对象中包含一个数据集对象，用于封装待显示的数据，可以通过界面的“复制”按钮将该图表复制一份，复制后可以得到新的图表对象，用户可以修改新图表的编号、颜色和数据。现使用原型模式设计该软件，所得类图如图 3-21 所示。

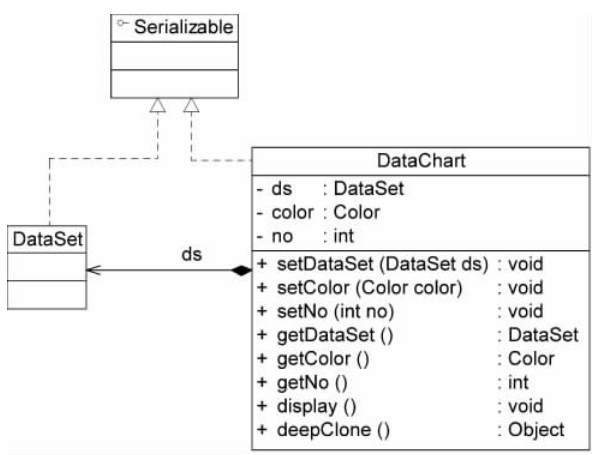

图 3-21　填空题(5)用图

在该设计方案中,DataChart 类包含一个 DataSet 对象,在复制 DataChart 对象的同时将复制 DataSet 对象,因此需要使用深克隆技术,可使用流来实现深克隆。

Java 代码如下:

```
import java.io.*;
class DataSet implements Serializable
{  //具体实现代码省略  }

class Color implements Serializable
{  //具体实现代码省略  }

class DataChart implements Serializable
{
    private DataSet ds = new DataSet();
    private Color color = new Color();
    private int no;
    //成员属性的 Getter 方法和 Setter 方法省略
    public void display() {
        //具体实现代码省略
    }
    //使用流实现深克隆,复制容器的同时复制成员
    public _____①_____ deepClone() throws IOException, ClassNotFoundException,
OptionalDataException
    {
        //将对象写入流中
        ByteArrayOutputStream bao = new ByteArrayOutputStream();
        ObjectOutputStream oos = new _____②_____;
        oos.writeObject(_____③_____);

        //将对象从流中取出
        ByteArrayInputStream bis = new ByteArrayInputStream(bao.toByteArray());
        ObjectInputStream ois = new _____④_____;
        return(_____⑤_____);
    }
}

class Test
{
    public static void main(String args[])
    {
        DataChart chart1,chart2 = null;
        chart1 = new DataChart();

        try{
            chart2 = (DataChart)chart1.deepClone();
        }
        catch(Exception e){
              e.printStackTrace();
        }
```

```
            System.out.println(chart1 == chart2);
            System.out.println(chart1.getDs() == chart2.getDs());
            System.out.println(chart1.getNo() == chart2.getNo());
        }
    }
```

编译并运行上述代码，输出结果为：___⑥___、___⑦___、___⑧___。

在本实例中，DataChart 类和 DataSet 类需要实现 Serializable 接口的原因是___⑨___。

（6）为了避免监控数据显示不一致并节省系统资源，在某监控系统的设计方案中提供了一个主控中心类，该主控中心类使用单例模式进行设计，类图如图 3-22 所示。

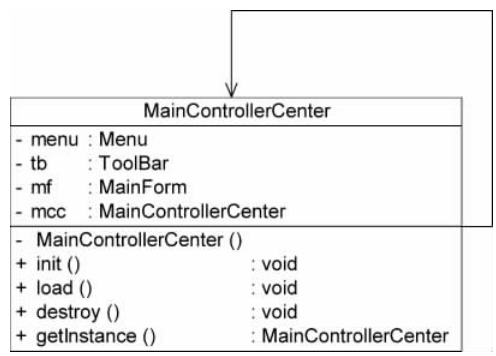

图 3-22　填空题(6)用图

在图 3-22 中，主控中心类 MainControllerCenter 是单例类，它包含一系列成员对象并可以初始化、显示和销毁成员对象，对应的方法分别为 init()、load()和 destroy()，此外还提供了静态工厂方法 getInstance()用于创建 MainControllerCenter 类型的单例对象。

Java 代码如下：

```
class MainControllerCenter
{
    private Menu menu;                  //主控中心菜单
    private ToolBar tb;                 //主控中心工具栏
    private MainForm mf;                //主控中心主窗口
    private      ①      MainControllerCenter mcc;

      ②      MainControllerCenter{
    }

    public void init()
    {
        menu = new Menu();
        tb = new ToolBar();
        mf = new MainForm();
    }

    public void load()
    {
        menu.display();
        tb.display();
        mf.display();
```

```
        }

        public void destroy()
        {
            menu.destroy();
            tb.destroy();
            mf.destroy();
        }

        public static MainControllerCenter getInstance()
        {
            if(mcc == null)
            {
            _____③_____;
            }
            return mcc;
        }
    }

    class Test
    {
        public static void main(String args[])
        {
            MainControllerCenter mcc1,mcc2;
            mcc1 = MainControllerCenter.getInstance();
            mcc2 = MainControllerCenter.getInstance();
            System.out.println(mcc1 == mcc2);
        }
    }
    //其他代码省略
```

编译并运行上述代码,输出结果为_____④_____。

在本实例中,使用了_____⑤_____(填写懒汉式或饿汉式)单例模式,其主要优点是_____⑥_____,主要缺点是_____⑦_____。

3. 综合题

(1) 使用简单工厂模式模拟女娲(Nvwa)造人(Person),如果传入参数 M,则返回一个 Man 对象,如果传入参数 W,则返回一个 Woman 对象,用 Java 语言模拟实现该场景。现需要增加一个新的 Robot 类,如果传入参数 R,则返回一个 Robot 对象,对代码进行修改并注意“女娲”的变化。

(2) 现需要设计一个程序来读取多种不同类型的图片格式,针对每一种图片格式都设计一个图片读取器(ImageReader),如 GIF 图片读取器(GifReader)用于读取 GIF 格式的图片、JPEG 图片读取器(JpgReader)用于读取 JPEG 格式的图片。图片读取器对象通过图片读取器工厂 ImageReaderFactory 来创建,ImageReaderFactory 是一个抽象类,用于定义创建图片读取器的工厂方法,其子类 GifReaderFactory 和 JpgReaderFactory 用于创建具体的图片读取器对象。使用工厂方法模式实现该程序的设计。

（3）计算机包含内存（RAM）、CPU 等硬件设备，根据下面的“产品等级结构-产品族”示意图（见图 3-23），使用抽象工厂模式实现计算机设备创建过程并绘制相应的类图。

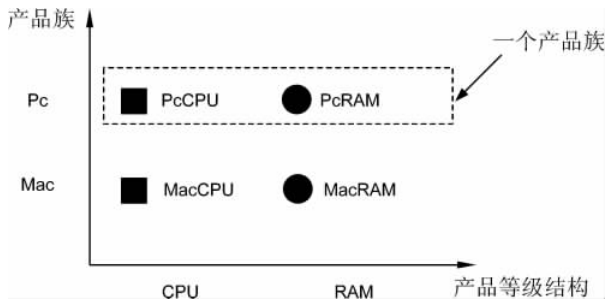

图 3-23　综合题（3）用图

（4）计算机组装工厂可以将 CPU、内存、硬盘、主机、显示器等硬件设备组装在一起构成一台完整的计算机，且构成的计算机可以是笔记本，也可以是台式机，还可以是不提供显示器的服务器主机。对于用户而言，无须关心计算机的组成设备和组装过程，工厂返回给用户的是完整的计算机对象。使用建造者模式实现计算机组装过程，要求绘制类图并编程实现。

（5）设计一个客户类 Customer，其中客户地址存储在地址类 Address 中，用浅克隆和深克隆分别实现 Customer 对象的复制并比较这两种克隆方式的异同，绘制类图并编程实现。

（6）使用单例模式的思想实现多例模式，确保系统中某个类的对象只能存在有限个，如两个或三个，设计并编写代码实现一个多例类。