CHAPTER 3

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，并保证代码可靠性。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，即模式的优缺点。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。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。

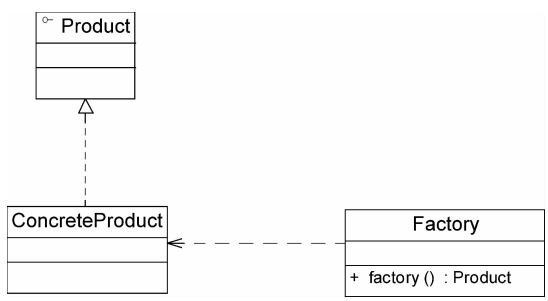

图 3-1　简单工厂模式结构图

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。

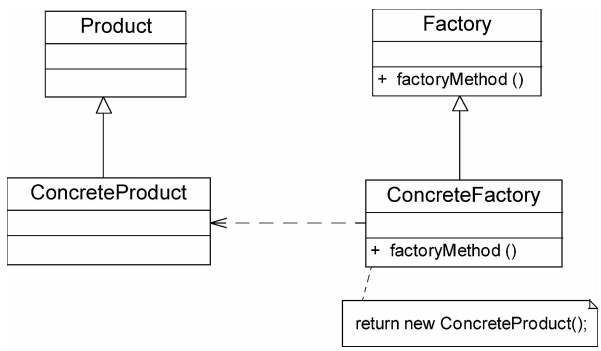

图 3-2 工厂方法模式结构图

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。

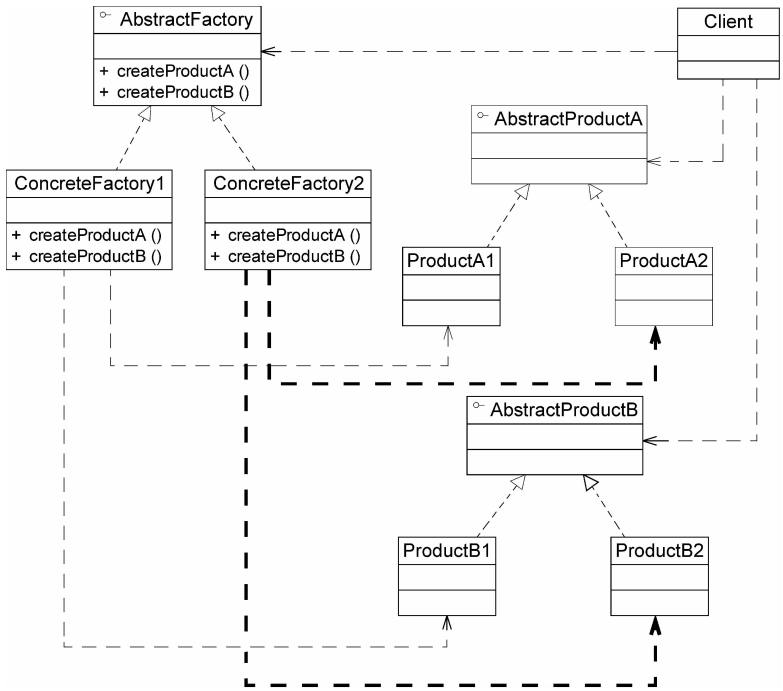

图 3-3 抽象工厂模式结构图

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。

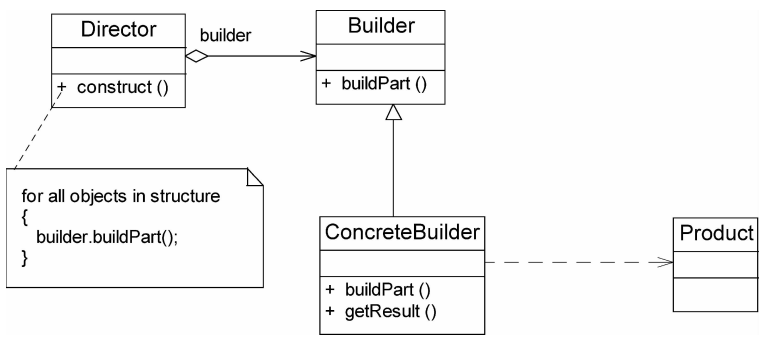

图 3-4 建造者模式结构图

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。

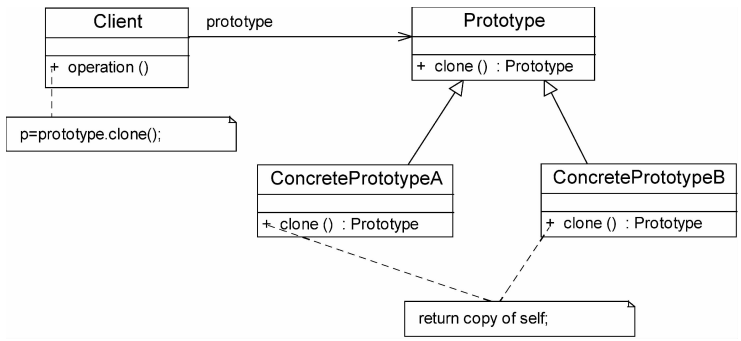

图 3-5　原型模式结构图

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，而原型模式无需这样；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

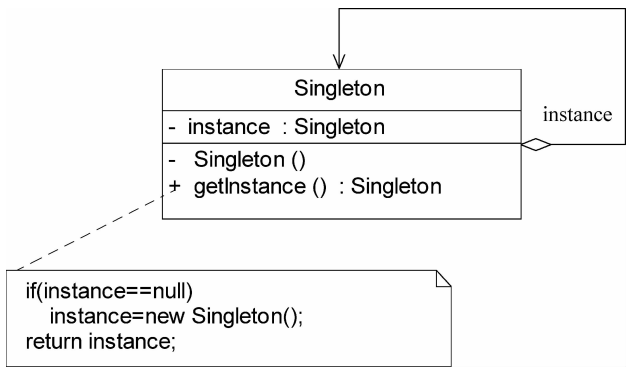

图 3-6　单例模式结构图

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。

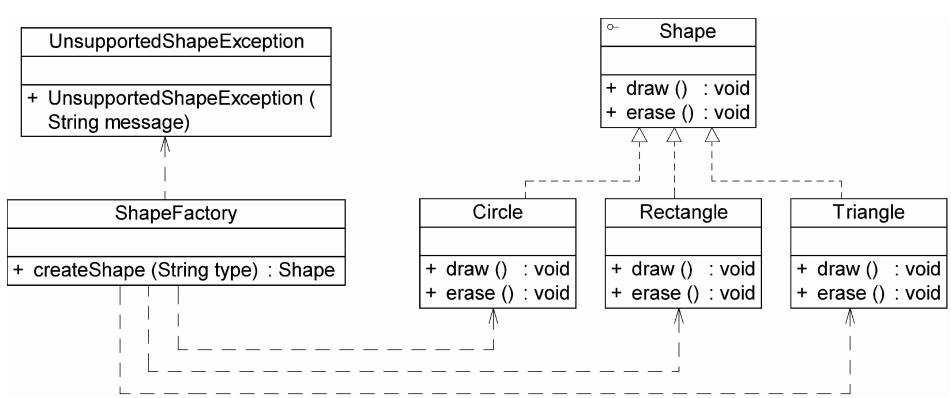

图 3-7　图形工厂实例类图

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");
    }
}
```