

抽象工厂模式 10

本章讲解抽象工厂模式（abstract factory pattern），该模式与第9章讲解的工厂方法模式类似。不同的是，抽象工厂模式允许调用组件在不了解创建对象所需类的情况下，创建一组或一系列相关的对象。表10-1列出了抽象工厂模式的相关信息。

表10-1 抽象工厂模式的相关信息

问 题	答 案
是什么	抽象工厂模式允许调用组件创建一组相关联的对象。调用组件无需了解创建对象所使用的类，以及选择这些类的理由。这个模式与第9章讲解的工厂方法模式类似，不同的是，此模式可以为调用组件提供一组对象
有什么优点	抽象工厂模式允许调用组件不必了解创建对象使用的类，也不用知道选择这些类的原因。因此，我们可以在不修改调用组件的情况下，对其使用的类的进行修改
何时应使用此模式	如果调用组件需要使用多个相互协作的对象，同时又无需了解这些对象之间的协作方式时，就可以使用此模式
何时避免使用此模式	如果只是创建一个对象，就不要使用此模式。这种场景下，应该使用相对简单的工厂方法模式
如何确定是否正确地实现了此模式	如果调用组件可以在不了解创建对象所使用的类的情况下，正确地接收到一组对象，就说明实现是正确的。正常情况下，调用组件只有通过对象所实现的协议或者继承的基类才能使用这些对象的功能
有哪些常见的陷阱	主要的陷阱是，将调用组件所使用的类的细节暴露给调用组件。比如，调用组件对选择实现类的决策过程存在依赖，或者对具体的某个类存在依赖
有哪些相关的模式	如果只是创建一个对象，应该使用相对简单的工厂方法模式（参见第9章）。抽象工厂方法模式通常配合单例模式和原型模式一起使用（参见10.5节）

10

10.1 准备示例项目

本章所使用的示例项目，是一个名为AbstractFactory的OS X命令行工具项目。项目功能是为不同型号的汽车制造零部件。首先，新建一个名为Floorplans.swift的文件，其内容如代码清单10-1所示。

代码清单10-1 Floorplans.swift文件的内容

```
protocol Floorplan {
    var seats:Int { get }
    var enginePosition:EngineOption { get }
}
```

```

enum EngineOption : String {
    case FRONT = "Front"; case MID = "Mid";
}

class ShortFloorplan: Floorplan {
    var seats = 2;
    var enginePosition = EngineOption.MID
}

class StandardFloorplan: Floorplan {
    var seats = 4;
    var enginePosition = EngineOption.FRONT;
}

class LongFloorplan: Floorplan {
    var seats = 8;
    var enginePosition = EngineOption.FRONT;
}

```

上述代码定义了一个名为Floorplan的协议，其中定义了汽车的基本配置，包括以整型属性seats表示的座位数，以及用EngineOption枚举值表示引擎位置的enginePosition属性。这里一共定义了三个遵循Floorplan协议的类，分别对应三个不同配置的平面图：ShortFloorplan、StandardFloorplan和LongFloorplan。

然后，在另一个名为Suspension.swift的文件中重复上述过程，其内容如代码清单10-2所示。

代码清单10-2 Suspension.swift文件的内容

```

protocol Suspension {
    var suspensionType:SuspensionOption { get };
}

enum SuspensionOption : String {
    case STANDARD = "Standard"; case SPORTS = "Firm"; case SOFT = "Soft";
}

class RoadSuspension : Suspension {
    var suspensionType = SuspensionOption.STANDARD;
}

class OffRoadSuspension : Suspension {
    var suspensionType = SuspensionOption.SOFT;
}

class RaceSuspension : Suspension {
    var suspensionType = SuspensionOption.SPORTS;
}

```

上述代码中的协议名为Suspension（车辆减震用的悬架），它定义了一个名为suspensionType的属性，该属性的取值为SuspensionOption的枚举值。与前面一样，这里也定义了三个遵循Suspension协议的类，以表示三种不同的悬架。

接下来，创建一个名为Drivetrains.swift的文件，表示最后一组零部件，其内容如代码清单10-3所示。

代码清单10-3 Drivetrains.swift文件的内容

```

protocol Drivetrain {
    var driveType:DriveOption { get };
}

```

```

}
enum DriveOption : String {
    case FRONT = "Front"; case REAR = "Rear"; case ALL = "4WD";
}
class FrontWheelDrive : Drivetrain {
    var driveType = DriveOption.FRONT;
}
class RearWheelDrive : Drivetrain {
    var driveType = DriveOption.REAR;
}
class AllWheelDrive : Drivetrain {
    var driveType = DriveOption.ALL;
}

```

此协议表示的是汽车的动力传动系统，其中定义了一个用DriveOption枚举类型作为值的driveType属性。上述三个实现类分别代表了三种用于汽车生产的动力传动系统。

最后，创建一个名为CarsParts.swift的文件，其内容如代码清单10-4所示。

代码清单10-4 CarsParts.swift文件的内容

```

enum Cars: String {
    case COMPACT = "VW Golf";
    case SPORTS = "Porsche Boxter";
    case SUV = "Cadillac Escalade";
}

struct Car {
    var carType:Cars;
    var floor:Floorplan;
    var suspension:Suspension;
    var drive:Drivetrain;

    func printDetails() {
        println("Car type: \(carType.rawValue)");
        println("Seats: \(floor.seats)");
        println("Engine: \(floor.enginePosition.rawValue)");
        println("Suspension: \(suspension.suspensionType.rawValue)");
        println("Drive: \(drive.driveType.rawValue)");
    }
}

```

上述代码定义了一个名为Cars的枚举，其取值为后续将要创建的汽车的型号。此外，这里还创建了一个名为Car的结构体，代表一辆完整的汽车，它拥有前面定义的各类零部件的属性。上述代码的最后，使用了printDetails函数，将汽车的配置信息输出到控制台。

提示 在这个项目中，所有的枚举值都将使用字符串类型来表示。在真实的项目中，一般不这么做，但是就这个示例项目而言，这么做有助于将相应的值直接输出到控制台。

10.2 此模式旨在解决的问题

第9章演示了工厂方法模式允许调用组件在不了解使用了哪个实现类，以及为何选择该实现类的情况下，获取实现类实例的方式。

本章解决的问题与此类似，不同之处在于，此模式面向的是一组相互关联，却没有共同基类或协议的对象。在上一节，我分别为三种不同的汽车零部件定义了三个协议，并分别给它们创建了三个实现类。我们需要根据枚举Cars中的值，为每个类别的汽车选择合适的零部件，如表10-2所示。

表10-2 汽车型号所对应的零部件产品

Car	Floorplan	Suspension	Drivetrain
COMPACT	StandardFloorplan	RoadSuspension	FrontWheelDrive
SPORTS	ShortFloorplan	RaceSuspension	RearWheelDrive
SUV	LongFloorplan	OffRoadSuspension	AllWheelDrive

目前，调用组件若想创建一个Car对象，就必须了解上表中的部分信息，因为只有知道这些信息才能实例化它需要用到的类。代码清单10-5列出了main.swift的内容，包含创建实现类对象和配置的过程。

代码清单10-5 main.swift文件的内容

```
var car = Car(carType: Cars.SPORTS,
    floor: ShortFloorplan(),
    suspension: RaceSuspension(),
    drive: RearWheelDrive());

car.printDetails();
```

此时运行应用，将在控制台看到如下输出：

```
Car type: Porsche Boxter
Seats: 2
Engine: Mid
Suspension: Firm
Drive: Rear
```

这种方式的问题与第9章遇到的问题类似，即选择实现类的逻辑将会散布在应用的各个角落，并且调用组件对某个具体的实现类存在依赖。如果表10-2中的对应关系发生了变化，那么所有使用了相关零部件的组件都要做出相应的修改。这种修改不但繁琐、容易出错，还不好测试。

10.3 抽象工厂模式

抽象工厂模式与工厂方法模式解决的问题类似。不同的是，抽象工厂模式的作用是创建一组不存在共同协议或者基类的对象。就上述示例项目而言，为了创建一个Car对象，需要用到三个对象，而这三个对象分别实现了三个不同的协议。Floorplan、Suspension和Drivetrain。

工厂方法模式与抽象工厂模式

关于工厂方法模式与抽象工厂模式之间的区别，以及何时应该使用哪个模式，一直存在着各种争议。不同编程语言，具有不同的特性，因此在不同语言中这两种模式的实现也不同，侧重点不同的实现更是加剧了争论。

我的建议是关注意图，而不是实现。如果你有一个与表10-2类似的产品对照表，而且你需要确保跑车用的悬架不会被用来制造越野车的话，就应该使用抽象工厂模式。抽象工厂模式将对象所属的类别隐藏在具体的工厂方法类之中，而工厂方法类又是对调用组件不可见的。这么做虽然增加了复杂度，但是可以让新增产品变得简单（创建新的具体工厂即可），对现有产品的修改也会变得容易（修改具体工厂即可）。

工厂方法模式要简单许多，因为它只负责管理一个对象，而且只需隐藏选取实现类的决策逻辑。后面，我将通过把多个工厂方法合并到一个类中的方式实现抽象工厂模式。

简而言之，不必在意言语上的争论，专注于目标即可。如果想要创建一个对象，就使用工厂方法模式；需要管理一组对象，就用抽象工厂模式。

抽象工厂模式通过把决策逻辑整合到一个地方的方式，解决了决策逻辑分散的问题。此外，该模式只允许调用组件访问协议，而不能访问遵循该协议的类，防止了调用组件对某个具体的实现类产生依赖。抽象工厂模式涉及四步操作，如图10-1所示。

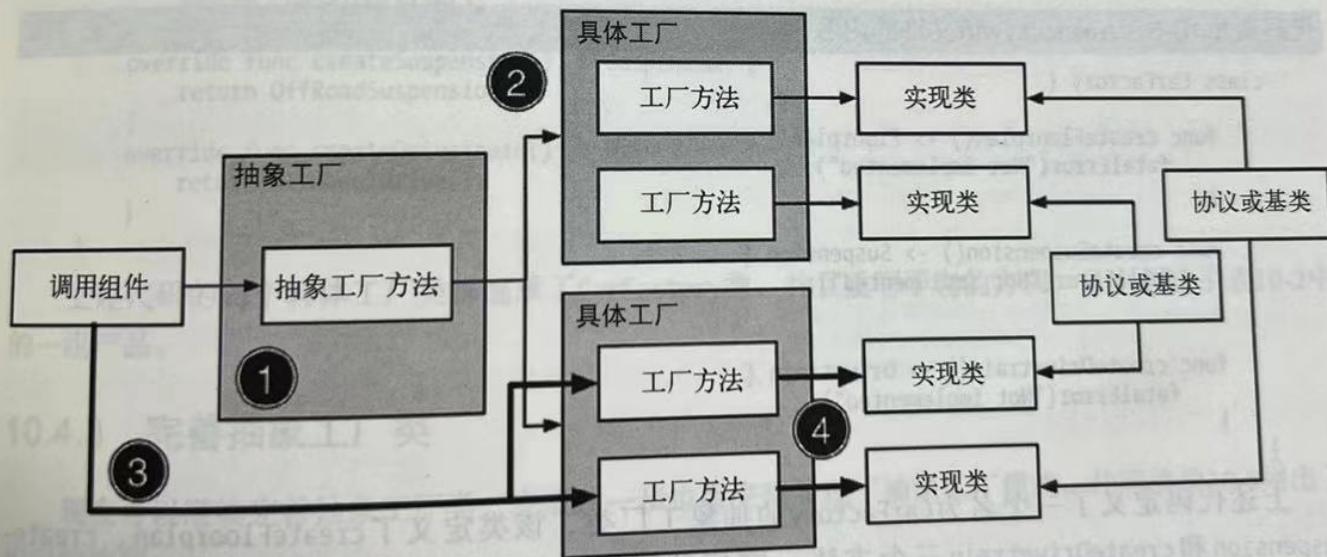


图10-1 抽象工厂模式

此模式需要用到抽象工厂类（abstract factory class），该类需要定义一个返回协议实现或者基类的方法。第一步是调用组件调用抽象工厂方法获取一个对象。

第二步操作是，抽象工厂方法检查调用组件的请求，然后选择一个具体工厂类（concrete factory class），即实现了工厂方法模式的类（参见第9章）。最后，使用工厂类创建一个实例，并将其返回给调用组件。

第三步操作是，调用组件调用具体工厂类中定义的工厂方法。这时就会触发第四步，也是最后一

步操作，即实例化实现类，并将调用组件请求的对象返回给它。

理解这个模式时，最好把注意力放在模式中各个类的信息是如何传递的。尽管这些具体工厂类实现了不同的协议，或者派生自不同的基类，但这些类知道哪些实现类属于一组，而且其工厂方法创建的对象也属于同一组。

抽象工厂虽然不知道最终会使用哪个实现类，但是对于某个请求，它知道如何选择合适的具体工厂类。

调用组件只了解实现类使用的工厂类及其实现的协议或者基类，至于实现类本身，调用组件并不了解。调用组件需要依赖抽象工厂类为其提供一个合适的具体工厂，同时需要具体工厂为其选择一个合适的实现类。

10.4 实现抽象工厂模式

后续的内容将演示如何实现抽象工厂模式，以创建本章开头所描述的汽车零部件对象。

10.4.1 创建抽象工厂类

首先，创建一个抽象工厂类。这个类是抽象工厂模式的核心，因为它将被用作具体工厂类的基类。下面为示例项目添加一个名为Abstract.swift的文件，代码清单10-6列出了该文件的内容。

代码清单10-6 Abstract.swift文件的内容

```
class CarFactory {
    func createFloorplan() -> Floorplan {
        fatalError("Not implemented");
    }

    func createSuspension() -> Suspension {
        fatalError("Not implemented");
    }

    func createDrivetrain() -> Drivetrain {
        fatalError("Not implemented");
    }
}
```

上述代码定义了一个名为CarFactory的抽象工厂类，该类定义了createFloorplan、createSuspension和createDrivetrain三个方法，它们的返回值分别为遵循Floorplan、Suspension和Drivetrain协议的对象。这个类作为具体工厂类的基类，目前实现的功能已经足够了。在定义了具体工厂类之后，再对CarFactory类进行完善，让它能够选择和使用具体工厂类。

10.4.2 创建具体工厂类

下一步是创建具体的工厂类，它的职责是创建一组可以同时使用的产品对象。首先，给示例项目添加一个名为Concrete.swift的文件，并在这个文件中定义相关类，如代码清单10-7所示。

代码清单10-7 Concrete.swift文件的内容

```

class CompactCarFactory : CarFactory {
    override func createFloorplan() -> Floorplan {
        return StandardFloorplan();
    }
    override func createSuspension() -> Suspension {
        return RoadSuspension();
    }
    override func createDrivetrain() -> Drivetrain {
        return FrontWheelDrive();
    }
}

class SportsCarFactory : CarFactory {
    override func createFloorplan() -> Floorplan {
        return ShortFloorplan();
    }
    override func createSuspension() -> Suspension {
        return RaceSuspension();
    }
    override func createDrivetrain() -> Drivetrain {
        return RearWheelDrive();
    }
}

class SUVCarFactory : CarFactory {
    override func createFloorplan() -> Floorplan {
        return LongFloorplan();
    }
    override func createSuspension() -> Suspension {
        return OffRoadSuspension();
    }
    override func createDrivetrain() -> Drivetrain {
        return AllWheelDrive();
    }
}

```

上述代码的每个具体工厂类都继承了CarFactory类，并且重写了它的方法，分别创建了表10-2中的一组产品。

10.4.3 完善抽象工厂类

现在可以继续完善抽象工厂类，完成这一步也就完整实现了抽象工厂模式。代码清单10-8列出了相应的改变。

代码清单10-8 完善Abstract.swift中的抽象工厂

```

class CarFactory {

    func createFloorplan() -> Floorplan {
        fatalError("Not implemented");
    }

    func createSuspension() -> Suspension {
        fatalError("Not implemented");
    }
}

```

```

func createDrivetrain() -> Drivetrain {
    fatalError("Not implemented");
}

final class func getFactory(car:Cars) -> CarFactory? {
    var factory:CarFactory?
    switch (car) {
        case .COMPACT:
            factory = CompactCarFactory();
        case .SPORTS:
            factory = SportsCarFactory();
        case .SUV:
            factory = SUVCarFactory();
    }
    return factory;
}
}

```

上述代码给CarFactory添加了一个名为getFactory的类方法，它接收一个参数，类型为Cars枚举。该方法的功能是选择一个合适的具体工厂类，然后实例化并将实例对象返回调用组件。对于调用组件而言，它可以通过该方法获得一个CarFactory对象，至于具体选择了哪个工厂类，以及为何选中某个类，则无从得知。

10.4.4 使用抽象工厂模式

最后一步是更新创建Car对象的代码，让它可以通过抽象工厂获得所需的产品。代码清单10-9列出了main.swift文件中的相关修改。

代码清单10-9 在main.swift文件中使用抽象工厂模式

```

let factory = CarFactory.getFactory(Cars.SPORTS);

if (factory != nil) {
    let car = Car(carType: Cars.SPORTS,
                  floor: factory!.createFloorplan(),
                  suspension: factory!.createSuspension(),
                  drive: factory!.createDrivetrain());

    car.printDetails();
}

```

上述代码没有直接实例化实现类，而是通过抽象工厂类获得一个具体工厂类，然后调用对应的方法以获得所需的对象。如果现在运行该应用，将看到以下输出：

```

Car type: Porsche Boxter
Seats: 2
Engine: Mid
Suspension: Firm
Drive: Rear

```

如代码清单10-9所示，main.swift文件中的代码与每个产品类之间并不存在依赖关系。这意味着，

如果表10-2中的对应关系发生了变化，我们只需更新对应的具体工厂类，而不用对使用这些类的组件进行修改。代码清单10-10修改了跑车的动力传动系统。

代码清单10-10 修改Concrete.swift中的一个实现类

```
... class SportsCarFactory : CarFactory {
    override func createFloorplan() -> Floorplan {
        return ShortFloorplan();
    }
    override func createSuspension() -> Suspension {
        return RaceSuspension();
    }
    override func createDrivetrain() -> Drivetrain {
        return AllWheelDrive();
    }
}
...
}
```

运行该应用，将看到以下输出。从输出内容可以看出，修改起作用了。

```
Car type: Porsche Boxter
Seats: 2
Engine: Mid
Suspension: Firm
Drive: 4WD
```

具体工厂将产品分组的决策逻辑整合到一个地方，减小了变化对应用的影响，同时也让决策逻辑更好维护和更好测试。抽象工厂则将选择具体工厂类的逻辑整合到一个地方，进一步分离了调用组件和实现类的分组逻辑。

10.5 抽象工厂模式的变体

抽象工厂模式有一些常见的变体，通过使用它们可以调整抽象工厂的实现方式。这些变体的基本机制与原来的实现并无差别，只是创建对象的方式不同。

10.5.1 隐藏抽象工厂类

第一个变体，也是最常见的变体，是将抽象工厂模式的实现隐藏在调用组件用来存储实现类对象的类或者结构体之中。就本章的示例项目而言，也就是将它隐藏到结构体Car中，代码清单10-11列出了相应的修改，修改之后的Car将直接与抽象工厂类和具体工厂类打交道。

代码清单10-11 将该模式的实现隐藏在CarParts.swift文件中

```
enum Cars: String {
    case COMPACT = "VW Golf";
    case SPORTS = "Porsche Boxter";
    case SUV = "Cadillac Escalade";
}
```

```

struct Car {
    var carType:Cars;
    var floor:Floorplan;
    var suspension:Suspension;
    var drive:Drivetrain;

    init(carType:Cars) {
        let concreteFactory = CarFactory.getFactory(carType);
        self.floor = concreteFactory!.createFloorplan();
        self.suspension = concreteFactory!.createSuspension();
        self.drive = concreteFactory!.createDrivetrain();
        self.carType = carType;
    }

    func printDetails() {
        println("Car type: \(carType.rawValue)");
        println("Seats: \(floor.seats)");
        println("Engine: \(floor.enginePosition.rawValue)");
        println("Suspension: \(suspension.suspensionType.rawValue)");
        println("Drive: \(drive.driveType.rawValue)");
    }
}

```

上述代码给结构体Car新增了一个初始化器，它将根据传入的枚举值创建相应的具体工厂。然后，便可以通过这些具体工厂获得创建汽车对象所需的Floorplan、Suspension和Drivetrain对象。修改之后，main.swift中的代码得到大幅简化，如代码清单10-12所示。

代码清单10-12 隐藏抽象工厂类之后main.swift中的代码

```

let car = Car(carType: Cars.SPORTS);
car.printDetails();

```

注意 这个方法对调用组件的意图做了两个假设，即假设调用组件想创建一个Car对象，而且它需要全部的三个对象。如果决定采用这个变体，就需要确保调用组件仍然可以访问抽象工厂，这样它才能在需要的时候（不管因为什么原因）创建相应的对象。

10.5.2 在具体工厂类中使用单例模式

另一个常见的变体是，在抽象工厂中使用单例模式。具体工厂很适合用作单例，因为它们只包含了创建实现类对象的逻辑。为了使用单例模式，首先要更新抽象工厂类，即具体工厂类的基类。代码清单10-13列出了相应的改变。

代码清单10-13 修改Abstract.swift文件以使用单例模式

```

class CarFactory {

    required init() {
        // do nothing
    }

    func createFloorplan() -> Floorplan {
        fatalError("Not implemented");
    }
}

```

```

    }
    func createSuspension() -> Suspension {
        fatalError("Not implemented");
    }
    func createDrivetrain() -> Drivetrain {
        fatalError("Not implemented");
    }
}
final class func getFactory(car:Cars) -> CarFactory? {
    var factoryType:CarFactory.Type;
    switch (car) {
        case .COMPACT:
            factoryType = CompactCarFactory.self;
        case .SPORTS:
            factoryType = SportsCarFactory.self;
        case .SUV:
            factoryType = SUVCarFactory.self;
    }
    var factory = factoryType.sharedInstance;
    if (factory == nil) {
        factory = factoryType();
    }
    return factory;
}
class var sharedInstance:CarFactory? {
    get {
        return nil;
    }
}
}
}
}

上述代码还添加了一个名为sharedInstance的计算类属性，如果具体工厂想实现单例模式，可以重写这个方法。这里还对getFactory方法的实现做了一些修改，如果sharedInstance有值，该方法将把值返回给调用组件。没有重写sharedInstance属性的具体工厂类，将继承默认实现，即创建一个新的工厂实例以响应请求。代码清单10-14列出了对其中一个具体工厂类的修改，修改之后它的实例将成为单例。

```

代码清单10-14 在Concrete.swift文件中使用单例模式

```

...
class SportsCarFactory : CarFactory {
    override func createFloorplan() -> Floorplan {
        return ShortFloorplan();
    }
    override func createSuspension() -> Suspension {
        return RaceSuspension();
    }
    override func createDrivetrain() -> Drivetrain {
        return AllWheelDrive();
    }
}
override class var sharedInstance:CarFactory? {
    get {
        struct SingletonWrapper {

```

```

        static let singleton = SportsCarFactory();
    }
    return SingletonWrapper.singleton;
}
}
...

```

上述代码重写了SportsCarFactory类的sharedInstance属性，实现了第6章介绍的单例模式。另外两个具体工厂类并没有做修改，也就是说，每次响应调用组件的请求时，那些类都会创建新的实例。相反，SportsCarFactory类则只有一个实例，所有它需要处理的请求都将使用同一个实例去处理。

10.5.3 在实现类中使用原型模式

你也可以在实现类中使用单例模式，不过这意味着所有的组件都将操作相同的一组对象。只有在没有可变状态，或者变化比较少，又或者做好了并发保护的情况下，才适合这么做。

另一个更常见的变体是，使用原型模式以克隆的方式创建实现类对象。后续的内容会演示如何在示例项目中使用原型模式。

1. 准备示例应用

第一步是更新实现类，让它们支持克隆操作。这里要做的修改可能会比想象的要多，因为Swift枚举类型不支持用于实现原型模式的NSCopying协议。为了达到目的，这里将创建一个Objective-C枚举，然后将其引入到Swift代码中。

首先，在Project Navigator中右击AbstractFactory，然后在弹出的菜单中选择New File。最后，在列表中选择Objective-C文件模板，如图10-2所示。

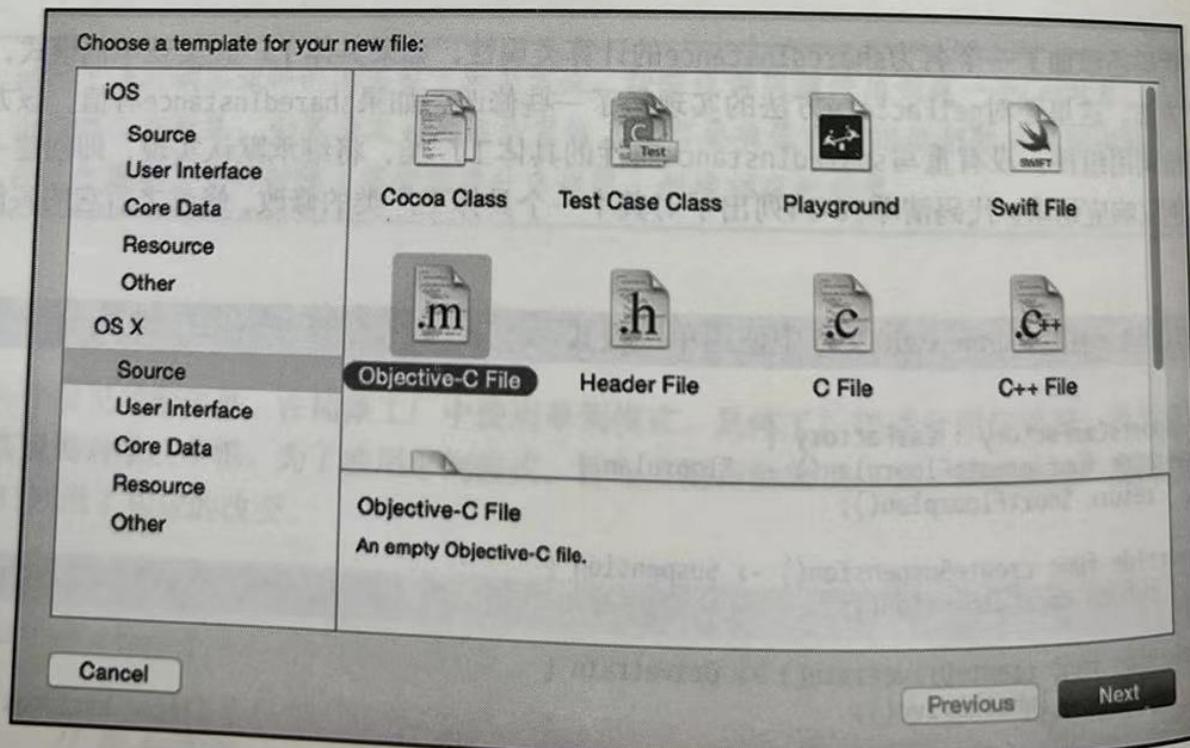


图10-2 向项目添加Objective-C文件

点击Next按钮，然后将文件名配置为SuspensionOption，如图10-3所示。

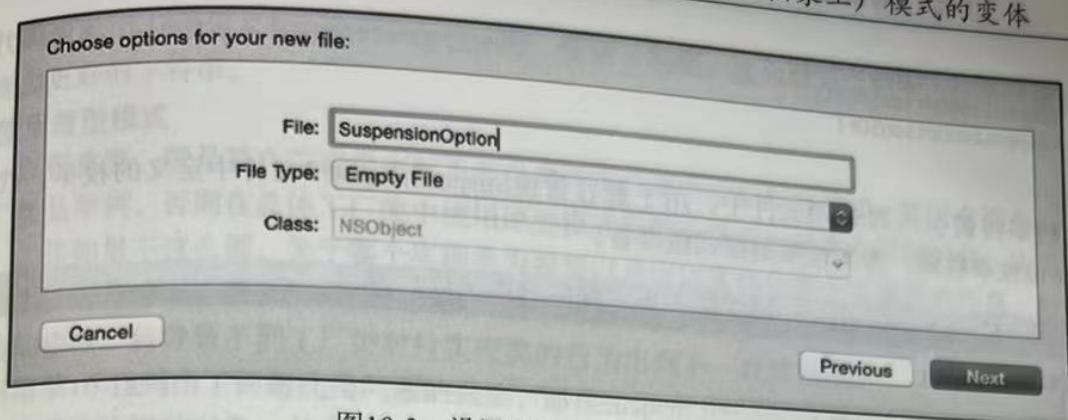


图10-3 设置Objective-C文件的名称

再次点击Next按钮，然后保存文件。保存文件时，Xcode会提示你创建一个bridging header文件，如图10-4所示。

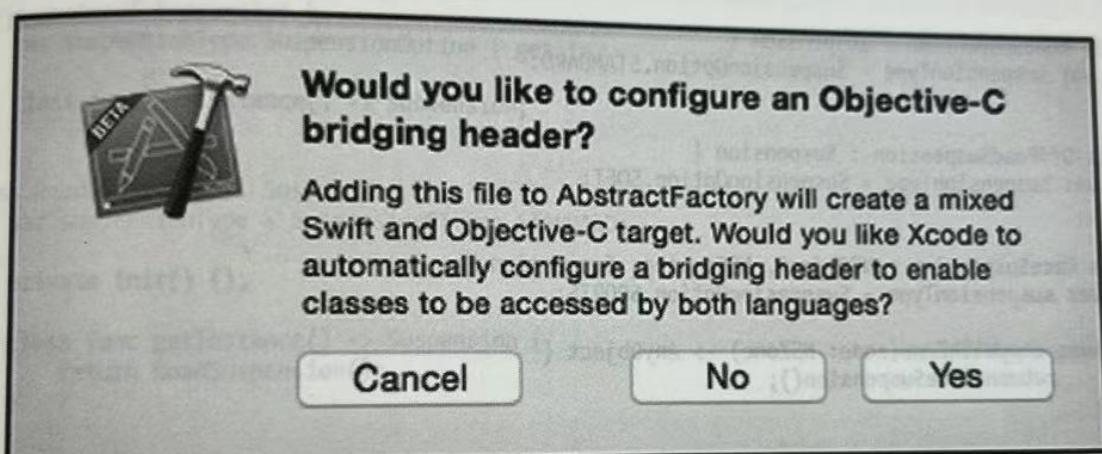


图10-4 Xcode提示创建bridging header文件

为了在Swift中使用Objective-C枚举，这里点击Yes按钮，以创建bridging header文件。Xcode会创建两个文件，其中一个是SuspensionOption.m，它是Objective-C文件，我们无需对该文件做任何修改。这个文件的作用是让Xcode配置好bridging header文件，即Xcode创建的另一个文件。bridging header文件的名称为AbstractFactory-Bridging-Header.h。代码清单10-15中列出了为定义所需枚举所做的修改。

代码清单10-15 AbstractFactory-Bridging-Header.h文件的内容

```
#import "SuspensionOption.h"
```

现在，再添加一个文件到项目中，不过这个文件只是一个名为SuspensionOption.h的头文件。然后，将代码清单10-16所示的代码输入到该文件中。

代码清单10-16 SuspensionOption.h文件的内容

```
#import <Foundation/Foundation.h>
typedef NS_ENUM(NSInteger, SuspensionOption) {
```

```

    SuspensionOptionSTANDARD,
    SuspensionOptionSPORTS,
    SuspensionOptionSOFT
}

```

这个枚举将被引入到Swift文件中，用于兼容替换Suspension.swift文件中定义的枚举。代码清单10-17将Swift枚举移除，为实现原型模式做准备。

10-17 将Swift枚举移除，为实现原型模式做准备

代码清单10-17 在Suspension.swift文件中为实现原型模式做好准备

```

import Foundation
@objc protocol Suspension {
    var suspensionType:SuspensionOption { get }
}

//enum SuspensionOption : String {
//    case STANDARD = "Standard"; case SPORTS = "Firm"; case SOFT = "Soft";
//}

class RoadSuspension : Suspension {
    var suspensionType = SuspensionOption.STANDARD;
}

class OffRoadSuspension : Suspension {
    var suspensionType = SuspensionOption.SOFT;
}

class RaceSuspension : NSObject, NSCopying, Suspension {
    var suspensionType = SuspensionOption.SPORTS;

    func copyWithZone(zone: NSZone) -> AnyObject {
        return RaceSuspension();
    }
}

```

这里在Suspension枚举前加了一个@objc修饰符，这样在具体工厂类中实现原型模式时才可以进行类型转换。上述代码将Swift SuspensionOption枚举注释了，这样它才不会与其Objective-C同类产生冲突。最后，修改RaceSuspension类，让其实现NSCopying协议，这样它才能够作为原型。

提示 由于原型模式比较简单，因此在实现类中使用原型模式所需的改动不大。如需了解原型模式更大的作用，可参见第5章。

此时运行应用，将看到如下输出：

```

Car type: Porsche Boxter
Seats: 2
Engine: Mid
Suspension: 1
Drive: 4WD

```

值得注意的是，Suspension的值是一个数字。Objective-C不支持将字符串作为枚举内部的类型，因此在10章中描述的2.有具体工的效果个工厂记做它来代码

因此在代码清单10-16中定义SuspensionOption时，使用了整型。这么做的后果是，枚举的输出值不再是描述能力更好的字符串。

2. 使用原型模式

有一点很重要，就是要在实现类中而不是在具体工厂类中使用原型模式。原因有两个。一是除非具体工厂类是单例，否则在具体工厂类中使用原型模式可能会导致出现多个原型对象，从而削弱模式的效果。二是如果不这么做，关于哪个实现类为原型以及应该实例化哪个实现类的信息，将散落在各个工厂类中，这意味着，修改某个实现类的行为，将要在使用了它的工厂类中做相应的修改。如果忘记做相应的修改，将导致不同工厂类对待实现类的行为出现不一致的情况。

代码清单10-18列出了回避此类问题的做法，即在Suspension协议中定义一个方法，工厂类可以用它来获取遵循该协议的对象，并允许实现类自行决定如何创建这些对象。

代码清单10-18 在Suspension.swift文件中使用原型模式

```
import Foundation

@objc protocol Suspension {
    var suspensionType:SuspensionOption { get };
    class func getInstance() -> Suspension;
}

class RoadSuspension : Suspension {
    var suspensionType = SuspensionOption.STANDARD;
    private init() {};
    class func getInstance() -> Suspension {
        return RoadSuspension();
    }
}

class OffRoadSuspension : Suspension {
    var suspensionType = SuspensionOption.SOFT;
    private init() {};
    class func getInstance() -> Suspension {
        return OffRoadSuspension();
    }
}

class RaceSuspension : NSObject, NSCopying, Suspension {
    var suspensionType = SuspensionOption.SPORTS;
    private override init() {};
    func copyWithZone(zone: NSZone) -> AnyObject {
        return RaceSuspension();
    }
    private class var prototype:RaceSuspension {
        get {
            struct SingletonWrapper {
                static let singleton = RaceSuspension();
            }
            return SingletonWrapper.singleton;
        }
    }
}
```

```

        }
        return SingletonWrapper.singleton;
    }

    class func getInstance() -> Suspension {
        return prototype.copy() as Suspension;
    }
}

```

提示 在真实项目中，你可以在所有的实现类中使用原型模式，但是这里只修改了suspension类，以避免重复相同的修改。

上述代码中给Suspension协议新增了一个名为getInstance的方法，每个实现类都必须定义此方法。RoadSuspension和OffRoadSuspension则只须创建新对象。RaceSuspension类中将原型对象定义成了单例，并在getInstance方法调用时创建一副本。

提示 实现类中定义的空初始化器都是private的，这样它们就无法直接被实例化。注意，在RaceSuspension类中还使用了override关键字，因为它从NSObject类继承了一个空的初始化器，而NSObject类是实现NSCopying协议必须继承的基类，详情请参加第5章。

为了反映suspension类中的变化，代码清单10-19更新了具体工厂类。

代码清单10-19 在Concrete.swift文件中所做的修改

```

class CompactCarFactory : CarFactory {
    override func createFloorplan() -> Floorplan {
        return StandardFloorplan();
    }
    override func createSuspension() -> Suspension {
        return RoadSuspension.getInstance();
    }
    override func createDrivetrain() -> Drivetrain {
        return FrontWheelDrive();
    }
}

class SportsCarFactory : CarFactory {

    override func createFloorplan() -> Floorplan {
        return ShortFloorplan();
    }
    override func createSuspension() -> Suspension {
        return RaceSuspension.getInstance();
    }
    override func createDrivetrain() -> Drivetrain {
        return AllWheelDrive();
    }

    override class var sharedInstance:CarFactory? {
        get {
    
```

```

    struct SingletonWrapper {
        static let singleton = SportsCarFactory();
    }
    return SingletonWrapper.singleton;
}

}

class SUVCarFactory : CarFactory {
    override func createFloorplan() -> Floorplan {
        return LongFloorplan();
    }
    override func createSuspension() -> Suspension {
        return OffRoadSuspension.getInstance();
    }
    override func createDrivetrain() -> Drivetrain {
        return AllWheelDrive();
    }
}

```

如此修改之后，就可以使用原型模式创建RaceSuspension类的实例。这点对于具体工厂类而言是不可见的，这意味着我们可以方便地改变各个实现类的行为，同时又不用修改具体工厂。

10.6 抽象工厂模式的陷阱

抽象工厂模式的主要陷阱是模糊了不同组件之间的界线。具体而言，抽象工厂类应只包含选择具体工厂的决策逻辑，而不应涉及选择实现类的逻辑。同样的，具体工厂应只包含选择实现类的决策逻辑，而不应提供协议所定义的功能。

当此模式与对象池模式配合使用时，还会引起另一个问题。如果试图为每个实现类管理一个单独的对象池，并让调用组件等待这些对象组合变得可用的话，这两种模式的配合将导致灾难性的后果。当调用组件请求的对象为同一类型时，对象池模式的可用性是最好的，而如果让多个组件排队访问同一对象集合，通常会导致死锁，尤其是两个调用组件中的一个组件获取到了另一个组件请求的可用对象时。如果确实想配合使用这两个模式，最好确保调用组件每次都以相同的顺序从对象池中获取对象，并且要非常谨慎，以免发生死锁。

10.7 Cocoa 中使用抽象工厂模式的示例

创建一个Cocoa对象时，很难知道使用了工厂方法模式还是抽象工厂模式。作为对象的接受者，你也无法知道获得的是一个具体工厂，还是一个根据自己的请求选择的类所创建的普通对象。如果苹果在其文档中将工厂方法模式和抽象工厂模式混为一谈也会让人感到困惑，因此你通常会看到类簇这个说法，其实就是对这两个模式的实现。

这么做其实是非常恰当的，因为这两个模式的精髓就是对调用组件隐藏决策逻辑和具体实现，区分这两个模式的唯一方法是阅读源代码。如果可以不读源码就区分这两个模式，这说明实现是有缺陷的。

10.8 在 SportsStore 应用中使用抽象工厂模式

为了演示如何在项目中使用抽象工厂模式，我们需要对SportsStore中生成产品库存总值的方式进行修改，以支持不同货币和汇率换算。后续的内容将定义用于汇率转换的协议和实现类，并讲解使用抽象工厂和具体工厂分发转换结果的方法。

10.8.1 准备示例应用

这里将直接使用第9章完成的项目，不需要其他的准备。

提示 别忘了，如果你不想自己重新创建SportsStore项目，可以前往Apress.com下载所有本书使用的源码。

10.8.2 定义实现类和协议

第一步是创建一个名为StockValueImplementations.swift的文件，并在该文件中定义实现类和协议，以实现相关功能。代码清单10-20列出了StockValueImplementations.swift文件的内容。

代码清单10-20 StockValueImplementations.swift文件的内容

```
import Foundation

protocol StockValueFormatter {
    func formatTotal(total:Double) -> String;
}

class DollarStockValueFormatter : StockValueFormatter {
    func formatTotal(total:Double) -> String {
        let formatted = Utils.currencyStringFromNumber(total);
        return "\((formatted))";
    }
}

class PoundStockValueFormatter : StockValueFormatter {
    func formatTotal(total:Double) -> String {
        let formatted = Utils.currencyStringFromNumber(total);
        return "f\((dropFirst(formatted))";
    }
}

protocol StockValueConverter {
    func convertTotal(total:Double) -> Double;
}

class DollarStockValueConverter : StockValueConverter {
    func convertTotal(total:Double) -> Double {
        return total;
    }
}
```

```
class PoundStockValueConverter : StockValueConverter {
    func convertTotal(total:Double) -> Double {
        return 0.60338 * total;
    }
}
```

上述代码定义了两个协议和两组实现类，其中两个协议的名称为 StockValueFormatter 和 StockValueConverter。StockValueConverter 协议负责汇率转换，而 StockValueFormatter 协议则负责总金额的格式化。至于两组实现类，一组负责计算总金额，另一组负责将总金额转换成英镑。

10.8.3 定义抽象与具体工厂类

接下来，在 StockValueFactories.swift 文件中定义抽象工厂类和具体工厂类，它们的内容如代码清单 10-21 所示。

代码清单 10-21 StockValueFactories.swift 文件的内容

```
import Foundation

class StockTotalFactory {
    enum Currency {
        case USD
        case GBP
    }

    private(set) var formatter: StockValueFormatter?
    private(set) var converter: StockValueConverter?

    class func getFactory(curr: Currency) -> StockTotalFactory {
        if (curr == Currency.USD) {
            return DollarStockTotalFactory.sharedInstance;
        } else {
            return PoundStockTotalFactory.sharedInstance;
        }
    }

    private class DollarStockTotalFactory : StockTotalFactory {
        private override init() {
            super.init();
            formatter = DollarStockValueFormatter();
            converter = DollarStockValueConverter();
        }

        class var sharedInstance: StockTotalFactory {
            get {
                struct SingletonWrapper {
                    static let singleton = DollarStockTotalFactory();
                }
                return SingletonWrapper.singleton;
            }
        }
    }

    private class PoundStockTotalFactory : StockTotalFactory {
```

```

private override init() {
    super.init();
    formatter = PoundStockValueFormatter();
    converter = PoundStockValueConverter();
}

class var sharedInstance: StockTotalFactory {
    get {
        struct SingletonWrapper {
            static let singleton = PoundStockTotalFactory();
        }
        return SingletonWrapper.singleton;
    }
}
}

```

StockTotalFactory类是一个抽象工厂，它可以根据传给getFactory方法的Currency枚举值，在DollarStockTotalFactory和PoundStockTotalFactory这两个类之间选择一个具体类，具体类与货币种类的对应关系如表10-3所示。

表10-3 货币种类与Formatter/Converter的对应关系

货币种类	StockValueFormatter	StockValueConverter
USD (美元)	DollarStockValueFormatter	DollarStockValueConverter
GBP (英镑)	PoundStockValueFormatter	PoundStockValueConverter

10.8.4 使用工厂与实现类

最后一步是使用抽象工厂转换和格式化库存总值。代码清单10-22列出了ViewController.swift文件中的改变。

代码清单10-22 在ViewController.swift文件中使用抽象工厂模式

```

...
func displayStockTotal() {
    let finalTotals:(Int, Double) = productStore.products.reduce((0, 0.0),
        {(totals, product) -> (Int, Double) in
            return (
                totals.0 + product.stockLevel,
                totals.1 + product.stockValue
            );
        });
}

var factory = StockTotalFactory.getFactory(StockTotalFactory.Currency.GBP);
var totalAmount = factory.converter?.convertTotal(finalTotals.1);
var formatted = factory.formatter?.formatTotal(totalAmount!);
totalStockLabel.text = "\((finalTotals.0) Products in Stock. "
+ "Total Value: \(formatted!)";
}
...

```

上述代码使用英镑作为货币种类，这意味着getFactory方法将选择处理英镑的实现类。运行该应用时，库存总值将从美元转换成英镑并在应用界面上展示，如图10-5所示。

Gold-plated, diamond-studded King
78 Products in Stock. Total Value: £246,362.43

图 10-5 转换与格式化库存总值

这里的 ViewController 类无须了解被选中的具体工厂类，以及它所提供的实现类的细节即可完成库存总值的格式化。

10.9 总结

本章中我们学习了如何使用抽象工厂模式创建属于某个对象集合，但没有遵循共同协议，也没有共同基类的对象。第11章，我们将学习建造者模式。