文章已收录我的仓库：Java学习笔记与免费书籍分享

模式类型

工厂模式属于创建者模式，与对象的创建有关，其中工厂方法模式用于类，而抽象工厂模式用于对象。创建型类模式将对象的部分创建工作延迟到子类，由子类创建对象；而创建型对象模式将它延迟到另一个对象中。

模式设计意图

工厂模式将复杂的对象创建工作隐藏起来，而仅仅暴露出一个接口供客户使用，具体的创建工作由工厂管理而对用户封装，将对象的创建和使用分离开来，降低耦合度，便于管理，能够很好的支持变化。

对于某些类而言，其对象的创建可能是需要一系列复杂的参数或大量准备代码，当这个任务由客户完成时，如果客户创建多个复杂对象，先不说这些代码可能难以编写，也会造成不必要的代码重复，交给工厂产生对象则不会造成代码重复，因为工厂中的代码写一次便可以复用无数次，能够减少代码量，并且用户无需关注这些难以编写的代码，增加可读性，例如Spring框架中的beanFactory便是如此。

此外，将对象的创建和使用分离开来，所有的创建工作由工厂管理，工厂可以管理类并适应相应业务变化，而无须对客户暴露，例如如果某个产品A升级为AA，我们可以直接在工厂中将`return new A()`改为`return new AA()`，客户仍然调用

`factory.createProductA()`而无须变更代码，如果客户是通过`new A()`的形式创建对象，那么我们还需要找到所有的代码并把它们改成`new AA()`，这在一个庞大的工程中是一项巨大的任务。

当客户使用的对象可能发生变化时，例如一开始使用A产品而后来想使用B产品时，普通的方式我们可能不得不更改所有的`new`方法，如果使用简单工厂模式，可以仅仅更改配置文件而无须更改客户代码来完成这个目的，如果使用工厂方法模式或者抽象工厂模式，那么可以调整工厂的构造函数或者使用set注入来达到这个目的，大大减少因变化而造成的不适应性。

上述所说的一切都归功于将类的使用与创建解耦，这也就是整个工厂模式的核心思想与设计意图。无论何时，高内聚、低耦合永远都是编写需要考虑到的地方。

工厂模式具体分为简单工厂、工厂方法和抽象工厂模式，这些方法都符合上述设计意图，但同时又满足不同的需求、适应不同的场景。

简单工厂(Simple Factory)

设计

定义一个创建对象的接口，创建一个子类通过传入的参数决定创建哪一种实例。

适用场景

- 工厂类负责生产的对象较少。
- 一个类不确定它所必须创建的对象的类的时候，具备不确定性。
- 客户知道需要传入工厂的参数而并不关心具体的类创建逻辑。

代码实例

假设我们具有如下需求：一家游戏厂商开发了A、B、C三种游戏，某个测试者被要求试玩相应游戏。

```
//定义游戏接口
interface Game {
    public void play();
}

//A游戏
class GameA implements Game{
    public void play(){System.out.println("Playing GameA");};
}

//B游戏
class GameB implements Game{
    public void play(){System.out.println("Playing GameB");};
}
```

```
//C游戏
class GameC implements Game{
    public void play(){System.out.println("Playing GameC");};
}

class GameFactory {
    public Game createGame(char type) {
        switch (type) {
            case 'A':
                return new GameA();

            case 'B':
                return new GameB();

            case 'C':
                return new GameC();
        }
        return null;
    }
}

//测试者(客户)类
class Tester {
    private char type;
    public Tester(char type) {
        this.type = type;
    }
    public void testGame() {
        GameFactory gameFactory = new GameFactory();
        //通过简单工厂获取游戏实例
        Game game = gameFactory.createGame(type);
        //试玩游戏
        game.play();
    }

}

//代码测试
public class Test {
    public static void main(String[] args) {
        //要求测试者试玩游戏A
        Tester tester = new Tester('A');
        tester.testGame();
    }
}
```

在测试代码Test类中我们是这样写的:

```
Tester tester = new Tester('A');
```

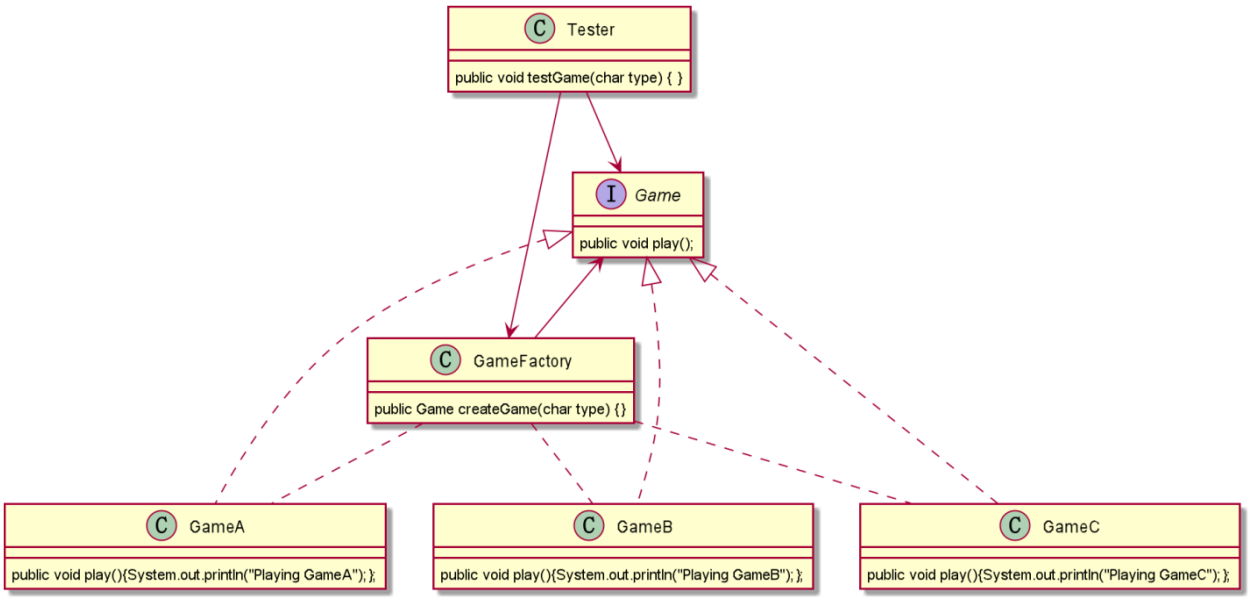
事实上在工程中我们并不直接写类型，而是导入配置文件:

```
Tester tester = new Tester(config.GAME_TYPE);
```

这样如果我们想要让测试者测试不同的游戏时，就可以修改config配置文件中的GAME_TYPE信息(还可以利用反射)，这就使得代码能够适应变化。

如果每个类的构造准备工作都是一致的初始化(上述代码没有任何准备工作而直接返回)，可以考虑采用哈希表存储参数与实例之间的关系，可以减少大量if-else语句。在这种情况下我们需要提前在哈希表中初始化并存储相关实例而没有考虑程序是否会用上，也许会造成不必要的资源浪费，但另一方面，我们每次获取实例都是返回HashMap中实例的克隆，这样比new操作效率更高——这其实就是享元设计模式。不过，大多数情况下我们仍需要对不同的实例进行不同的初始化操作，此时仍需要大量的判断语句。（如果类真的具有一致性而采用Map缓存，事实上这种设计模式就是策略模式，策略模式是编程中最基本的设计模式，我们大多都在有意无意的使用——定义算法接口并由子类实现不同算法，定义相关类以选择这些子类。）

UML类图



总结

简单工厂模式的结构非常简单，易于使用，当对象实例较少时可以考虑简单工厂模式。通过UML类图可以发现，简单工厂模式对客户封装了具体创建类的逻辑过程，所以我们可以在工厂中进行一些复杂的初始化或其他操作帮助客户减轻压力，用户仅仅需要知道传入的具体参数即可获得相应实例。简单工厂模式也可以应对适量的业务变化而不影响客户代码，符合工厂模式的设计意图。

简单工厂模式的缺点也是很明显的，正如它的名字，它仅仅适用于较为简单的场景，而对稍加复杂的情况会使得简单工厂无比庞大而难以维护，当增加或减少实例时，我们必须对工厂进行较为大量的修改，这违反了开闭原则。此外，当工厂出现BUG时，整个程序将会崩溃。

工厂方法(Factory Method)

设计

定义一个创建对象的接口，由子类决定实例化哪一个类，工厂方法将类的实例化推迟到子类实现。

适用场景

- 一个类不确定它所必须创建的对象的类的时候，具备不确定性。
- 你期望获得较高的扩展性。
- 当一个类希望由它的子类来指定它所创建的对象。
- 当类将创建对象的职责委托给多个帮忙子类的中的某一个，并且客户知道将要使用哪一个帮忙子类。

代码实例

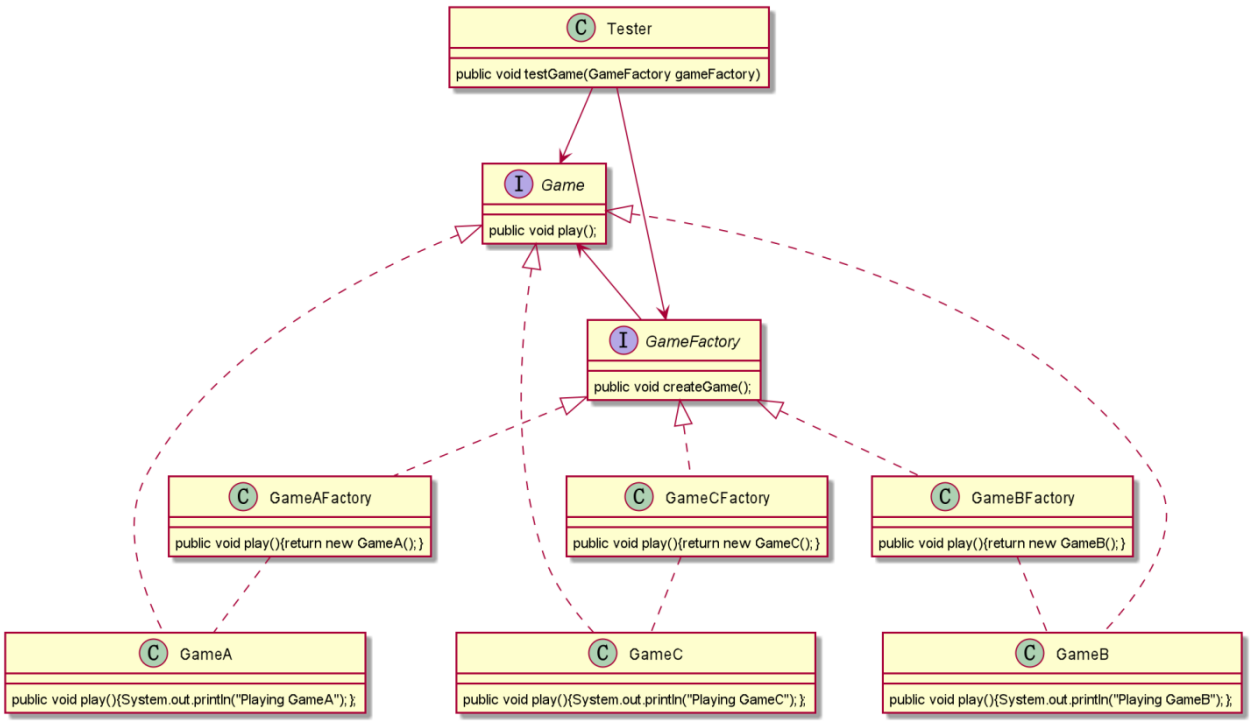
假设我们同样具有需求：一家游戏厂商开发了A、B、C三种游戏，测试者被要求试玩相应游戏。

```
//定义游戏接口
interface Game {
    public void play();
}

//A游戏
class GameA implements Game{
    public void play(){System.out.println("Playing GameA");};
}

//B游戏
class GameB implements Game{
    public void play(){System.out.println("Playing GameB");};
}

//C游戏
class GameC implements Game{
    public void play(){System.out.println("Playing GameC");};
}

//定义工厂(父类)
interface GameFactory {
    Game createGame();
}

//帮忙子类，游戏A工厂
class GameAFactory implements GameFactory {
    @Override
    public Game createGame() {
```

```
        return new GameA();
    }
}

//帮忙子类，游戏B工厂
class GameBFactory implements GameFactory {
    @Override
    public Game createGame() {
        return new GameB();
    }
}

//帮忙子类，游戏C工厂
class GameCFactory implements GameFactory {
    @Override
    public Game createGame() {
        return new GameC();
    }
}

//测试者(客户)类
class Tester {
    private GameFactory gameFactory;

    public Tester(GameFactory gameFactory) {
        this.gameFactory = gameFactory;
    }

    public void testGame() {
        //通过工厂获取游戏实例
        Game game = gameFactory.createGame();
        //试玩游戏
        game.play();
    }

}

//代码测试
public class Test {
    public static void main(String[] args) {
        //要求测试者1试玩游戏A
        GameFactory gameFactory = new GameAFactory();
        Tester tester1 = new Tester(gameFactory);
        tester1.testGame();

         //要求测试者2也试玩游戏A
        Tester tester2 = new Tester(gameFactory);
        tester2.testGame();

        //... 测试者1000也试玩游戏A
    }
}
```

我们可以通过更改构造函数或使用set方法来更改工厂，可以更改较少的代码就能让所有测试者都更改试玩游戏(最坏情况下仍然需要更改较多代码)，具有较好的灵活性。

UML类图



总结

工厂方法模式其实就是简单工厂模式一种扩展，工厂方法模式具有高扩展性，如果后续想要增加类时，直接编写一个新的帮忙子类即可，可以很方便的生产或切换产品，而无需修改现有的代码，完美符合了开闭原则·。

工厂方法模式是工程中较为理想的一种设计模式，但它的缺点也是明显的，当客户新建一个产品时，就不得不创建一个产品工厂，增加了代码量，而且对于父类接口难以进行修改，因为一旦修改接口，就必须要对众多的帮忙子类进行修改。

抽象工厂(Abstract Factory)

设计

提供一个接口以创建一系列相关或互相依赖的对象，将类的实例化延迟到子类。

适用场景

- 代码需要与多个不同系列的相关产品交互， 但是由于无法提前获取相关信息， 或者出于对未来扩展性的考虑， 你不希望代码基于产品的具体类进行构建而仅希望显示创建它们的接口。

- 需要创建的对象是一系列相互关联或相互依赖的产品族。
- 当一系列中的产品被设计为一起使用时，在一个应用中仅使用同一个系列中的对象。

代码实例

假设具有需求如下：一家品牌鞋垫可以生产大号尺寸的鞋子和对应鞋垫和小号尺寸的鞋子和对应鞋垫，现在有一个客户想要购买一双鞋子穿。

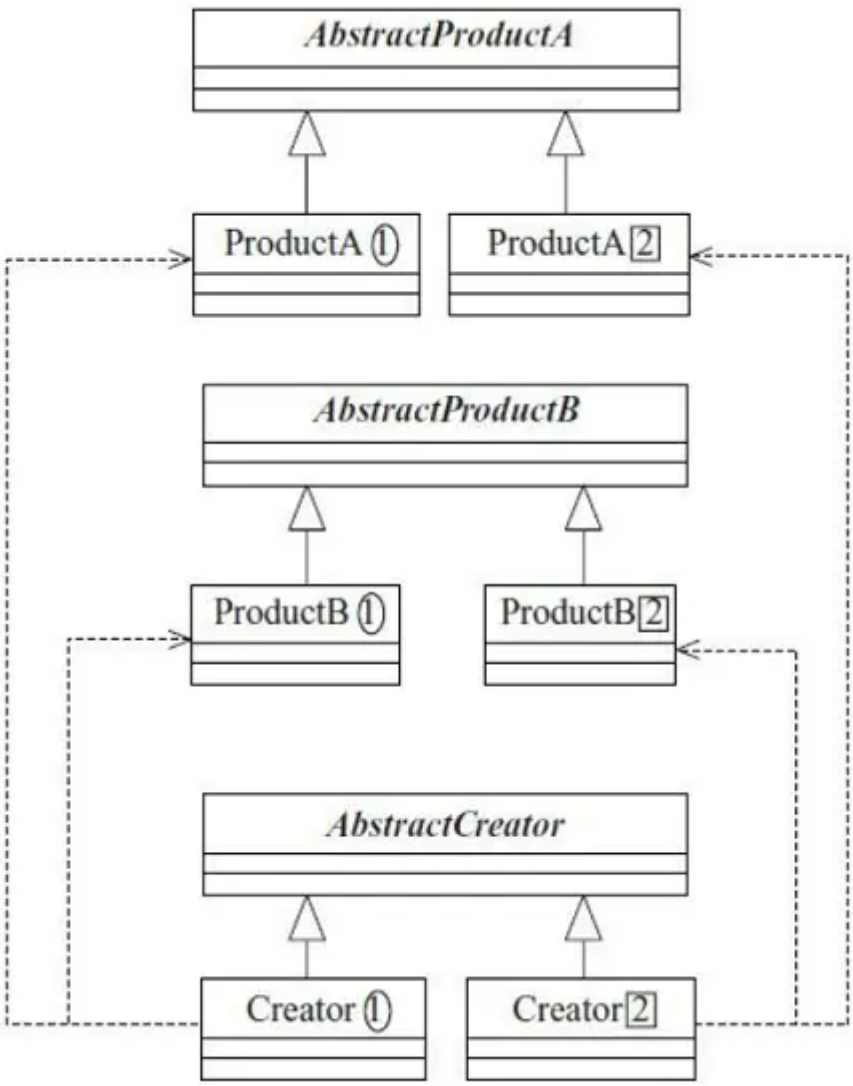
```
//定义鞋子接口
interface Shoe {
    void printShone();
}

//定义鞋垫接口
interface Insole {
    void printInsole();
}

//具体产品，大号鞋子
class LargeShoes implements Shoe {
    @Override
    public void printShone() {
        System.out.println("大号鞋子");
    }
}

//具体产品，大号鞋垫
class LargeInsole implements Insole {
    @Override
    public void printInsole() {
        System.out.println("大号鞋垫");
    }
}

//具体产品，小号鞋子
class SmallShoes implements Shoe {
    @Override
    public void printShone() {
        System.out.println("小号鞋子");
    }
}

//具体产品，小号鞋垫
class SmallInsole implements Insole {
    @Override
    public void printInsole() {
        System.out.println("小号鞋垫");
    }
}

//定义完整鞋子工厂接口，一个完整的鞋子由鞋子和鞋垫组成
interface CompleteShoeFactory {
```

```
    Shoe createShoe();
    Insole createInsole();
}

//大型鞋子工厂，生产配套的大号鞋子和鞋垫
class CompleteLargeShoeFactory implements CompleteShoeFactory {
    @Override
    public Shoe createShoe() {
        return new LargeShoes();
    }

    @Override
    public Insole createInsole() {
        return new LargeInsole();
    }
}

//小号鞋子工厂，生产配套的小号鞋子和鞋垫
class CompleteSmallShoeFactory implements CompleteShoeFactory {
    @Override
    public Shoe createShoe() {
        return new SmallShoes();
    }

    @Override
    public Insole createInsole() {
        return new SmallInsole();
    }
}

//客户类，购买鞋子
class Customer {
    private CompleteShoeFactory factory;

    public Customer(CompleteShoeFactory factory) {
        this.factory = factory;
    }

    //购买完整的鞋
    public void buyCompleteShoe() {
        Shoe myShoe = factory.createShoe();
        myShoe.printShone();

        Insole myInsole = factory.createInsole();
        myInsole.printInsole();

        System.out.println("我已经买了配套产品，终于有鞋穿了！");
    }
}

//代码测试类
public class Test {
    public static void main(String[] args) {
        //购买大号鞋子
        //这里通常使用单例模式生成工厂
```

```
        CompleteShoeFactory factory = new CompleteLargeShoeFactory();
        Customer customer = new Customer(factory);
        customer.buyCompleteShoe();
    }
}
```

理解抽象工厂需要创建的对象是一系列相互关联或相互依赖的产品族的好处，例如这里的大号鞋子配大号鞋垫，小号鞋子配小号鞋垫，这就是对应的产品族，消费者要么买大号产品，要么买小号产品，不可能买大号鞋配小号鞋垫，这就使得一个具体工厂类在一个应用中仅出现一次，因此具体工厂类通常使用单例设计模式创建，由于一个具体工厂类在一个应用中仅出现一次，那么变更产品系列将十分简单，我们仅需要修改一行代码——创建工厂时的代码，这具有极大的灵活性。

UML类图



总结

抽象工厂模式用于创建的对象是一系列相互关联或相互依赖的产品族，易于交换产品系列，能够较好的应对变化。因为一个对象的产品被设计成一起工作，因此也利于维护产品

的一致性，如果采用简单工厂模式，那么得创建四个工厂——大鞋子工厂、大鞋垫工厂、小鞋子工厂和鞋垫工厂，引入的类也增多，而且用户可能会不小心创建大鞋子和小鞋垫，使得结果不适配，采用抽象工厂模式则可以较好的解决这个问题。

抽象工厂模式的缺点在于当工厂接口的功能越来越多时，它会变得越来越笨重，因为所有的子类都必须要去实现这里接口，这就要保证不同系列的产品种类都是一致的，此外，后续想要增加新种类或者删减种类，都不得不对所有子类做出更改。

结语

工厂模式包括简单工厂模式，工厂方法模式，还是抽象工厂模式，它们在形式和特点上是极为相似的，而且最终目的都是为了解耦。在使用时，我们不必去在意这个模式到底工厂方法模式还是抽象工厂模式，因为他们之间的演变常常是令人琢磨不透的。经常你会发现，明明使用的工厂方法模式，当新需求来临，稍加修改，加入了一个新方法后，由于类中的产品构成了不同等级结构中的产品族，它就变成抽象工厂模式了；而对于抽象工厂模式，当减少一个方法使的提供的产品不再构成产品族之后，它就演变成了工厂方法模式。

所以，在使用工厂模式时，只需要设计的六大原则的目的是否达到了。