设计模式学习总结

目录

[工厂方法模式Factory Method 2](#_Toc532899027)

[抽象方法模式Abstract factory pattern 4](#_Toc532899028)

[建造者模式Builder Pattern 6](#_Toc532899029)

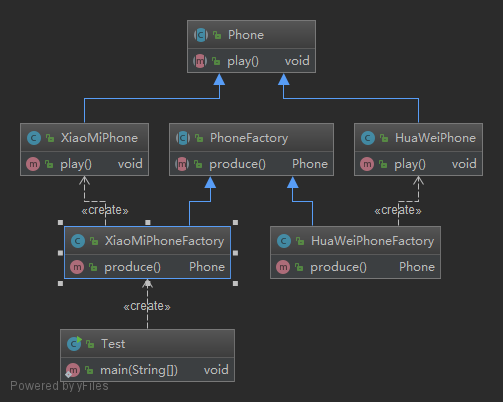
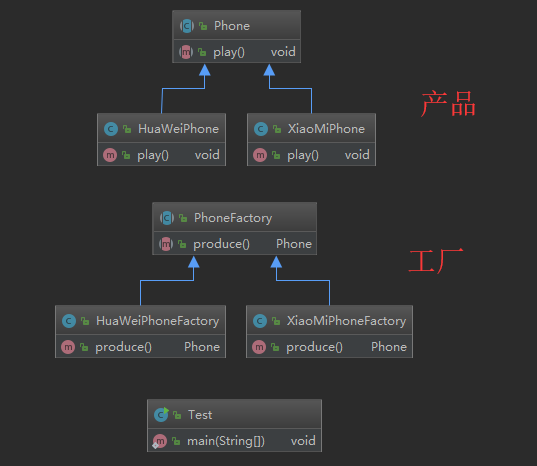
[单例模式Singleton 9](#_Toc532899030)

[原型模式Prototype pattern 13](#_Toc532899031)

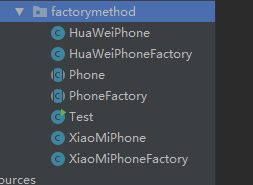
## 工厂方法模式Factory Method

主要应用场景是同一个产品等级，如小米手机，华为手机属于同一产品等级，但是属于不同的工厂，此时的设计方案应该是建立一个工厂和一个产品的抽象类或者接口，再建立相应的小米工厂子类和华为工厂子类，分别生产相应的产品。后续如果再新增同一产品等级的产品，如新增苹果手机，则只需要新建一个苹果工厂类，和一个苹果手机的产品类即可。

UML图：



文件结构：

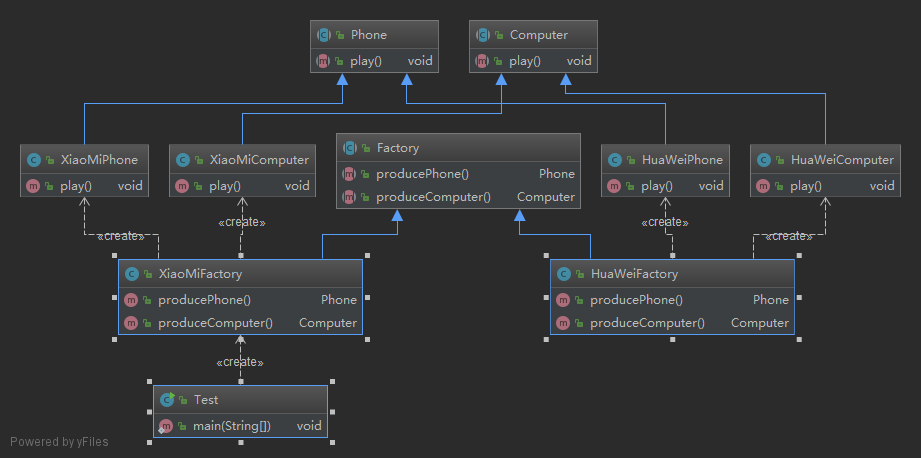
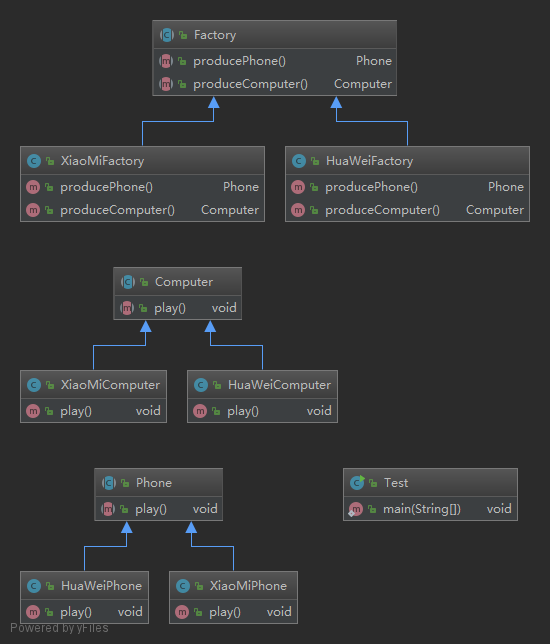


## 抽象方法模式Abstract factory pattern

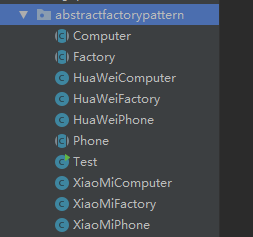
主要应用场景是同一个产品族，比如小米电视，小米手机，小米电脑，是同一产品族。

如果按照工厂方法模式来做，需要新建出来小米手机工厂，小米电脑工厂，如果业务很复杂，可能会出现类爆炸的情况。

抽象模式使用场景是产品等级不经常变动的业务场景，（比如小米工厂确定只生产手机和电脑两种产品，不会经常性的添加产品等级）而只需要其余产品族（如添加一个联想工厂，微软工厂等等），此类场景只需要添加相应工厂实现类和产品实现类即可。



文件结构：

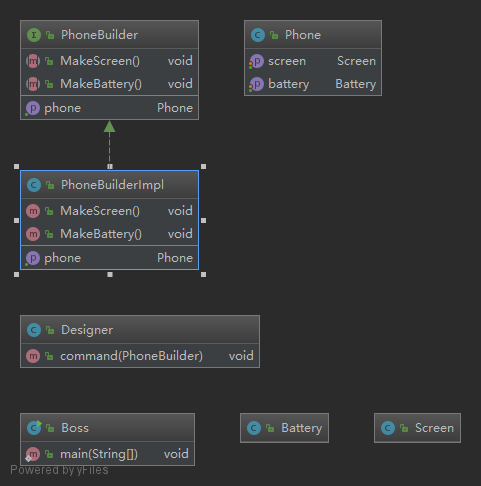


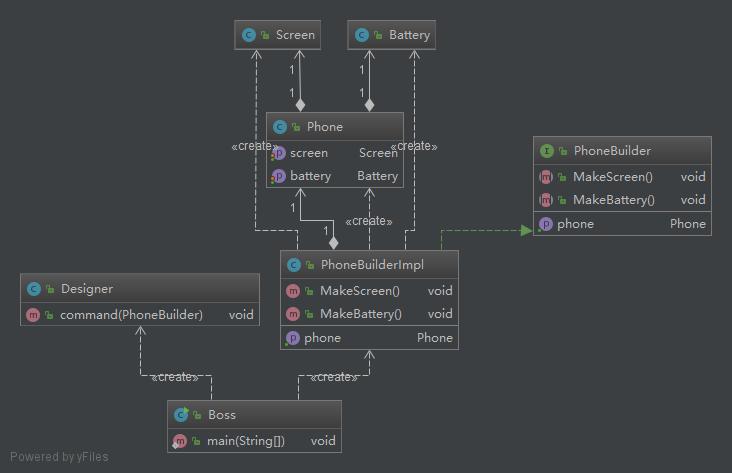
## 建造者模式Builder Pattern

建造者模式适合于一个具有较多属性的产品的创建过程，根据产品创建过程中属性的构造是否具有一致的先后顺序，分为两种形式。

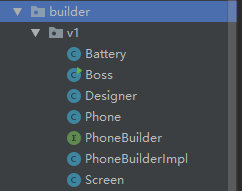
1. 通过Client、Director、Builder、Product形成的建造者模式。Builder负责构建Product类对象，Director负责指导Builder按照指定的顺序去构造Product，最后通过Builder返回建造后的结果。

场景：假设手机有屏幕Screen和电池Battery两个零件。Boss老板需要一个手机，就指派给设计师Designer一个任务说我需要一个手机，由设计师指挥命令PhoneBuilder工人去造手机，先造屏幕，再造电池，然后造出来一个手机给Designer（command方法实现），再由Designer把

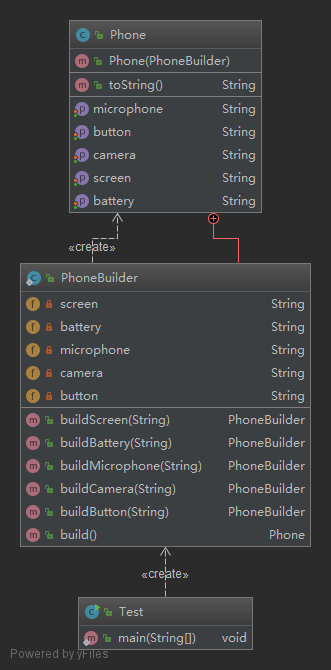


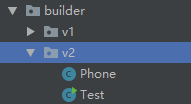


文件结构



1. 通过静态内部类方式实现的属性无序化构造，可以采用链式赋值，用到什么属性给什么属性赋值。





## 单例模式Singleton

1. 懒汉式单例模式

*/\*\*  
 \* 懒汉式单例模式  
 \* 单线程的情况下没有问题  
 \* 如果是多线程的话，可能多个线程同时判断lazySingleton==null = true，会出现隐患  
 \* 1.new了多个对象，但是返回的对象是最新产生的对象。  
 \* 2.由于线程的运行速度不同，可能会返回不同的对象。  
 \* 实现了序列化接口后需要重写readResolve方法防止序列化破坏单例模式  
 \*  
 \* 反射攻击：（无法防御反射攻击）  
 \* 由于懒汉模式是什么时候用什么时候初始化，所以私有构造器中加上非空判断也无法防止反射攻击  
 \* 因为无法确定是先创建单例，还是先反射调用  
 \*/*public class LazySingleton implements Serializable {  
 private static LazySingleton *lazySingleton* = null;  
 private LazySingleton(){  
 //只能防止先初始化后反射调用的情况  
 if (*lazySingleton* != null){  
 throw new RuntimeException("懒汉模式被反射攻击");  
 }  
 }  
  
 public static LazySingleton getInstance(){  
 if(*lazySingleton* == null){ //如果多个线程同时进入，可能出现隐患  
 *lazySingleton* = new LazySingleton();  
 }  
 return *lazySingleton*;  
 }  
 //防止序列化破坏  
 private Object readResolve(){  
 return *lazySingleton*;  
 }  
}

1. 双重检查单例模式

*/\*\*  
 \* 双重检查的懒汉模式，线程安全的写法  
 \* volatile关键字可以禁止线程内重排序  
 \* 实现了序列化接口后需要重写readResolve方法防止序列化破坏单例模式  
 \*  
 \* 反射攻击：（无法防御反射攻击）  
 \* 由于懒汉模式是什么时候用什么时候初始化，所以私有构造器中加上非空判断也无法防止反射攻击  
 \* 因为无法确定是先创建单例，还是先反射调用  
 \*/*public class LazyDoubleCheckSingleton implements Serializable {  
 private volatile static LazyDoubleCheckSingleton *lazyDoubleCheckSingleton*;  
 private LazyDoubleCheckSingleton(){  
 //只能防止先初始化后反射调用的情况  
 if (*lazyDoubleCheckSingleton* != null){  
 throw new RuntimeException("懒汉模式被反射攻击");  
 }  
 }  
  
 public static LazyDoubleCheckSingleton getInstance(){  
 if(*lazyDoubleCheckSingleton* == null){  
 synchronized (LazyDoubleCheckSingleton.class){  
 if(*lazyDoubleCheckSingleton* == null){  
 *lazyDoubleCheckSingleton* = new LazyDoubleCheckSingleton();  
 //new一个对象分三个步骤  
 //1.分配内存空间  
 //2.初始化对象  
 //3.对象指向内存空间  
 //intra-thread semantics 允许那些在单线程内，不会改变单线程程序执行结果的重排序，所以2.3可能重排序  
 //添加volatile关键字，可以禁止重排序。  
 }  
 }  
 }  
 return *lazyDoubleCheckSingleton*;  
 }  
  
 //防止序列化破坏  
 private Object readResolve(){  
 return *lazyDoubleCheckSingleton*;  
 }  
}

1. 饿汉模式

*/\*\*  
 \* 饿汉模式：类加载的时候就产生  
 \* 实现了序列化接口后需要重写readResolve方法防止序列化破坏单例模式  
 \* 添加反射攻击的代码，由于饿汉模式是在类初始化时就生成成功，所以在私有构造器中加上非空判断可以防止反射调用  
 \*/*public class HungrySingleton implements Serializable {  
 private static HungrySingleton *hungrySingleton* = new HungrySingleton();  
  
 //私有构造方法  
 private HungrySingleton(){  
 //防止反射破坏  
 if(*hungrySingleton* != null){  
 throw new RuntimeException("饿汉模式单例遭攻击");  
 }  
 }  
  
 public static HungrySingleton getInstance(){  
 return *hungrySingleton*;  
 }  
  
 //实现序列化接口后重写readResolve方法防止序列化破坏  
 private Object readResolve(){  
 return *hungrySingleton*;  
 }  
}

1. 私有静态内部类单例模式

*/\*\*  
 \* 私有静态内部类的单例模式  
 \* 实现了序列化接口后需要重写readResolve方法防止序列化破坏单例模式  
 \* 添加了防止反射攻击判断，私有静态内部类的单例模式是在类初始化时就生成成功，所以在私有构造器中加上非空判断可以防止反射调用  
 \*/*public class StaticInnerClassSingleton implements Serializable {  
 //私有构造器  
 private StaticInnerClassSingleton(){  
 //防止反射攻击  
 if(InnerClass.*staticInnerClassSingleton* != null){  
 throw new RuntimeException("静态内部类单例模式被反射攻击");  
 }  
 }  
  
 //私有静态内部类  
 private static class InnerClass{  
 private static StaticInnerClassSingleton *staticInnerClassSingleton* = new StaticInnerClassSingleton();  
 }  
  
 public static StaticInnerClassSingleton getInstance(){  
 return InnerClass.*staticInnerClassSingleton*;  
 }  
  
 //重写readResolve方法  
 private Object readResolve(){  
 return *getInstance*();  
 }  
}

1. 枚举单例模式

*/\*\*  
 \* 枚举模式，最优选择  
 \* 自身就可以防止序列化破坏和反射攻击  
 \*/*public enum EnumInstance {  
 *INSTANCE*;  
 private Object data;  
  
 public Object getData() {  
 return data;  
 }  
  
 public void setData(Object data) {  
 this.data = data;  
 }  
  
 public static EnumInstance getInstance(){  
 return *INSTANCE*;  
 }  
}

## 原型模式Prototype pattern