**1．工厂方法 Factory Method**

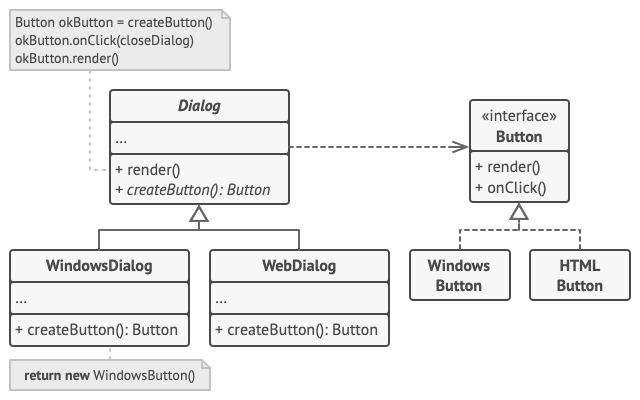


Fig 1: Factory Method

抽象类Creator中createProduct()方法被子类ConcreteCreator重写，工厂实际生产的产品类型是由子类决定的。

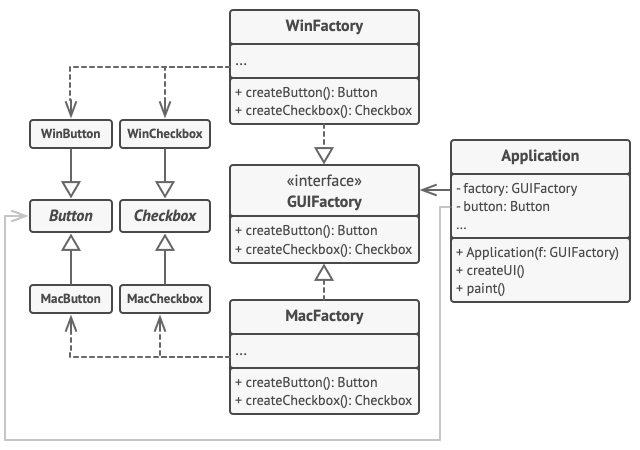
当多个不同类型对象有相同的业务逻辑时，可以使用工厂方法构建对象来减少代码修改量：把共同的业务逻辑放在父类方法someOperation()中，具体的业务逻辑由继承工厂类决定。

所生产的对象需要实现统一的接口，这样子工厂子类才可以重写createProduct()方法。

e.g.

|  |
| --- |
| Creator creator = new ConcreteCreatorA();  creator.someOperation();  //如果此时业务需要新增一个类，其业务逻辑与ConcreteCreatorA类似，我们就不需要重新写一个类，只需要写一个新的工厂子类，子类生产对应的对象。  Creator creator = new ConcreteCreatorB();  //类似creator.someOperation();等业务语句都不需要修改。 |

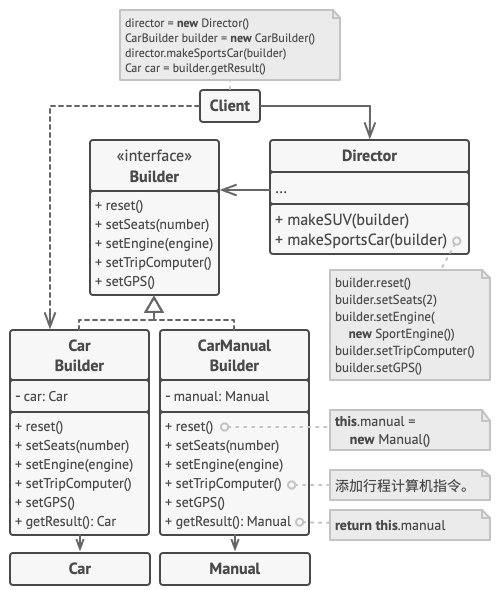
**2．抽象工厂方法 Abstract Factory**

；

抽象工厂方法类似与工厂方法，区别在于：

1. 抽象工厂只负责生产产品，不像工厂方法中还会执行一部分共同的业务逻辑。
2. 抽象工厂有多个抽象产品类，而工厂方法中只有一个抽象产品。
3. 抽象工厂自己决定实例化哪个对象，而工厂方法是由其子类自行决定实例化哪个对象。

**3．生成器模式 Builder**

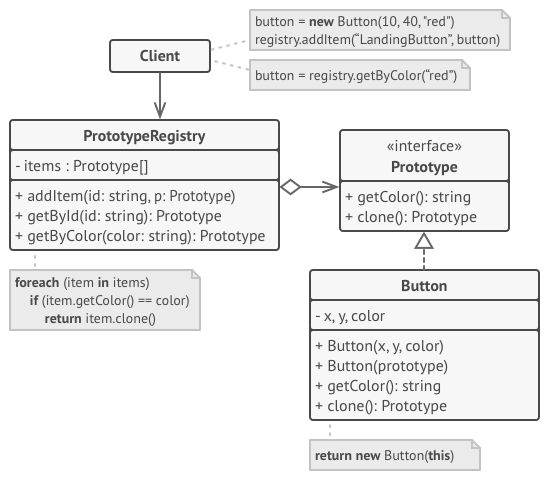


生成器模式是针对一些构造函数中参数繁多的对象的构造。通过Builder接口将产品拆分为一个个部件进行生成。具体的生成器类将会实现生成方法，并且在getResult()中可以返回不同的对象——这意味着生成器可以生成完全不同的对象。

Director（主管）中可以定义具体的生成步骤，不同的生成步骤可以生成属性多变的对象，e.g. SUV， Sports Car...

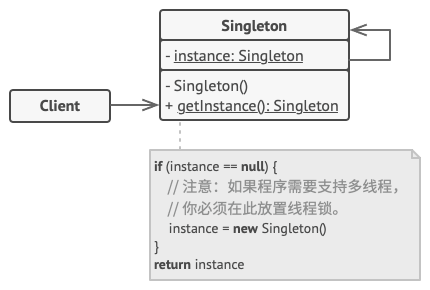
注意：生成器生成的具体对象要通过生成器子类访问，因为接口Builder和Director都不知道具体生成的对象。

1. **原型设计模式**



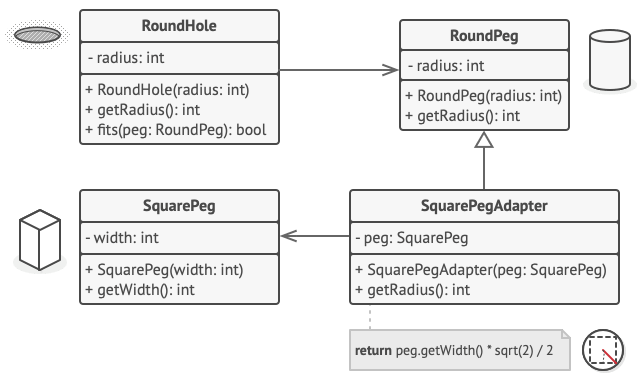
适用于需要频繁对对象进行拷贝的场景。原型接口定义抽象克隆方法，子类实现克隆方法。克隆方法通过进行类似C++中的拷贝构造方法实现。可以将一些通用的原型预定义在Registry中，客户可以方便地搜索和获取想要的原型拷贝对象。

1. **单例模式 Singleton**



一个类只能有一个实例对象。将构造函数设为私有，通过静态方法来创建实例和获取实例。

1. **适配器模式 Adapter**



适配器顾名思义，起的是中间转换器的作用。如上图所示，SquarePegAdapter可以将一个方钉经过处理后转换成圆钉，从而插入RoundHole。适配器需继承自兼容指定操作的类，在上图中继承了RoundPeg。