接下来进入**桥接模式**

首先来看到给出的定义：将抽象部分与它的实现部分分离，使得它们都可以独立的进行扩展变化。

我们来看到一个简单的例子。我们电脑上常用的绘图软件，我们可以使用其绘制各种各样的图像，矩形，圆型等。如果要我们来设计的话，我们将会把这些图形抽象为一个个类，这些类的具体示例就是我们画布上的具体图形。

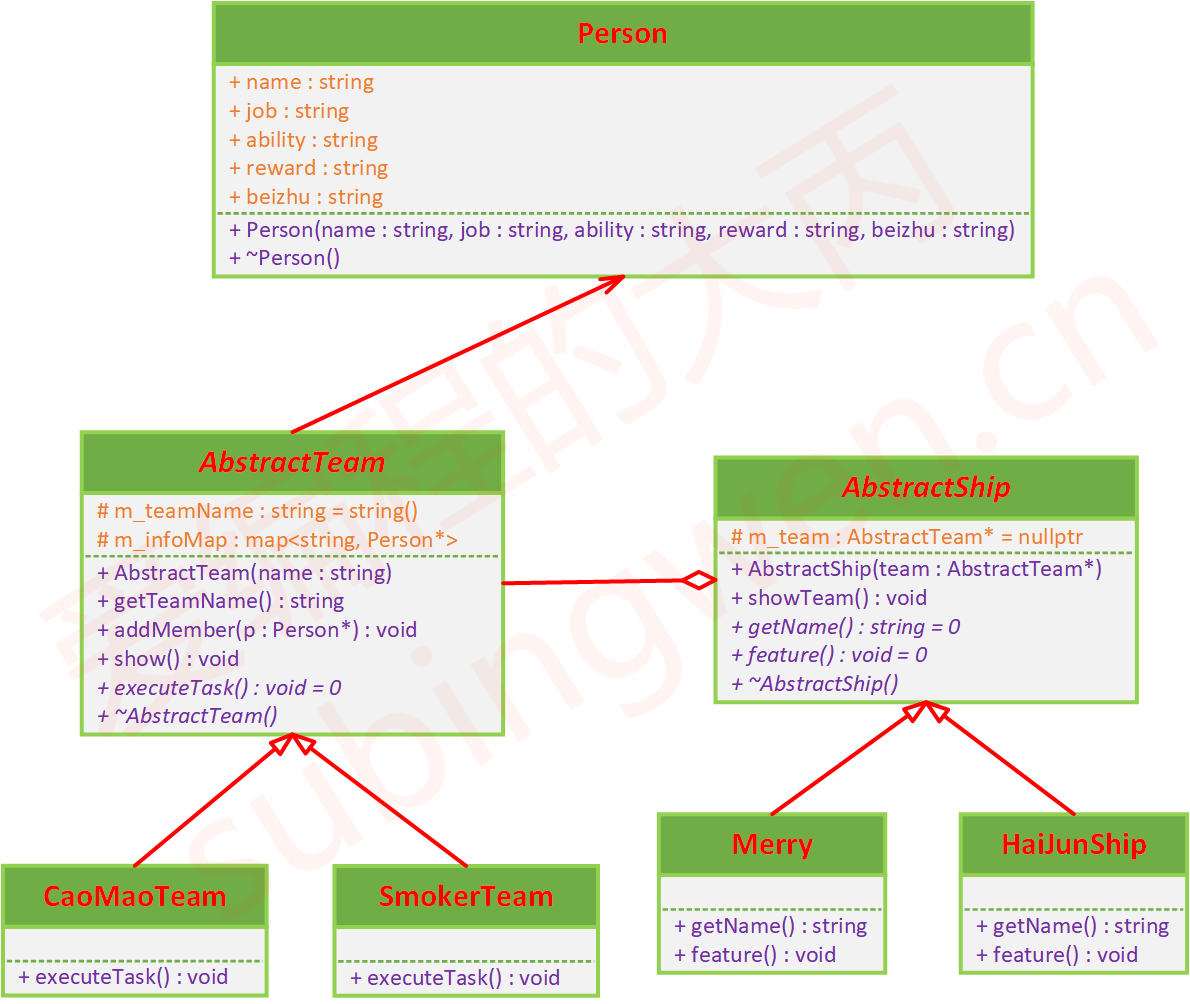
在这基础之上，我们还可以为这些个图形指定不同的颜色，就比如黄色，紫色，白色等等。那么我们如果想为我们绘制的图形也加上指定的颜色呢？在底层设计中，我们有多种做法，最简单的就是为每一个图形类都去派生出对应颜色的子类吧，这样当我们要创建一个特定颜色的矩形时，我们只需要去创建对应的对象即可了。但是吧，你应该也能感觉到这个设计上的巨大缺陷，当我们存在11个图形，11个颜色的时候，我们如果要列出所有可能的图像，我们总共需要创建121个类，这个是不可接受的。那么我们就需要一个优化的方法。

这个方法其实在现实生活中也有体现，就比如你在画画的时候，你一般是什么，你一般都是先考虑画什么图形，然后再考虑里面需要填充什么颜色。这个想法中，我们考虑画什么图形的这一步可以抽象为抽象部分，后面的填充颜色可以视为实现部分，毕竟这部分意味着你将会画出一个实际的图形出来。

接下来，我们再将这个想法建模到我们的设计中去。我们前面的做法是通过一个类去完整的标识我们需要的图像的形状，颜色等属性。但是我们这里把它们分离开来，把这些固定了组合的类给他拆分成形状类和颜色类。这就是桥接模式的核心，将一个类中可拆分的部分且不拆分会导致设计复杂的部分给他拆分出来，用前面的术语其实就是将抽象部分与实现部分给他拆分成俩个单独的类。

让我们来看看这样做的好处，在这种设计下，我们的形状类和颜色类都由一个独立的抽象类进行管理。那么我们需要怎么去实现一个具体颜色的矩形的创建呢。这时候就需要我们**桥接模式**的**桥接**俩字提供作用了。我们创建一个具体的实例类，就比如紫色矩形。这个类中应该至少保留着一个形状指针和一个颜色指针。我们前面已经提到，那个抽象部分和实现部分都应该被管理。在实际设计中，其实就是各自隶属于一个抽象类下。在我们的这一个实例类中，这写个部分一般都以一种聚合关系连接，毕竟你想，即使你这些个实例对象不见了，你的矩形属性，颜色属性，这些都应该是还存在的吧。

也就是说，桥接模式的核心，就是把一个复杂的类，甚至于不用说复杂，只要这个类中存在相对联系比较松散的属性，我们可以将这些属性给它分离开来，类中只保存这些类的一个单独实例。且这些类之间通过聚合关系联系。通过这样的设计。类间关系进行了进一步的解耦，当我们需要进行扩展时，我们不再需要对这个类进行一系列的派生。而这样在需要特定的属性类时去创建一个唯一的示例类即可，大大减少了设计的复杂度。



上面是一个桥接模式UML类图的举例。

让我们来分析下桥接模式的UML类图，其实在这其中，最关键的其实是其关联关系的几个类。这些个关联关系的类定义了整个桥接模式下的类的基本性质。

在整个桥接模式设计中，各个职责不强关联的类都被分离开来并一般都由一个抽象类进行管理。然后这一系列的抽象类间将会通过组合关系或者聚合关系进行互相通信。这种通信定义了我们最后看到的类的属性，就比如红色的矩形之类。

在桥接模式要表示的一个最后的对象中，一般就是我们这些个关联关系的属性的集合即各个抽象类的单个实例。在这些个抽象类中，能够派生出一系列的子类，但是最后的对象有且只有使用了这些个类的一个。在这里其实可以联想到我们的抽象工厂模式中的一个产品的构建，很相似。不过我们的抽象工厂模式服务的对象是一个产品的构建，其实吧，这里也是一个产品的构建，但是这个产品的构建是由其他相对来说比较大的产品来构建的，也可以看做是抽象工厂模式的产品层者一层的。