接下来看创建者模式中另外的重要的设计模式，**工厂**。这个工厂具体能细分为3类：简单工厂模式，工厂模式，抽象工厂模式。

工厂模式主要就是俩块，一块是工厂，可以将这一块理解为下订单的，它负责根据用户提供的需求来提供具体的产品。另一块是产品类，这一块负责具体产品的制造。注意的就是这里的产品制造和工厂在实际应用中都将会被抽象为类。

通过这样的设计，我们将能够值通过给工厂下达指令来获取产品。而工厂本身并不生产产品，它将生产产品的这一职能外包给了产品类。通过这样的设计。我们实现了获得产品，生产产品三个层次的解耦。从类的设计来看就是，工厂类中不涉及任何一个产品的对应的生产代码，其应该保存的是一个产品基类的指针等，工厂类可以通过这个指针去找到对应的类，而这个找到的类的类型将会由外部调用工厂类的对应函数的对应参数来决定。然后工厂类通过接口去指示对应的产品类去生产对应的产品，然后逐层返回实现产品的生产以及获取。

对于一个工厂能够生产的对象，其实是有一定的要求的。你按现实生活中的逻辑来看你应该也能够理解。毕竟在工厂模式的设计中，我们期望是根据具有一定相似度的原料和不同的要求来生成不一样的产品。在这里类的实际设计中也是这样，我们通常期望通过统一类型的参数过户，至少需要各个产品类需要的参数类型一直，数量上也有一定的规律，我们才有可能通过一定的设计规范来设计出生产产品的选择逻辑。在现实中也是如此，你总不能让一个医院去给你生产一个医疗系统吧，这个应该是IT公司的业务，所以在设计时我们需要考虑我们这个工厂想要生产的类的属性是怎么样的，具有越强关联性的产品类设计起来将会越符合规范。

在进入工厂模式之前，我们先来看一下简单的模式要求。在工厂模式中，我们需要一个工厂来作为我们获取产品的渠道。这个工厂应该能通过我们想要的产品的种类来实现对应产品的生产与提供。而我们在外部调用这个工厂进行生产的话，无疑，我们需要提供一些产品的信息，就比如产品的名称等等。最简单的就是提供一个产品的枚举名来进行制造。我们这里先不对别的参数进行讨论。

假设工厂的api能够根据我们提供的枚举类型来进行对应的产品生产，那么接下来我们需要考虑设计的就是对应的产品类。一个工厂应该能生产不止一个产品。但是这多个产品之间应该存在一些紧密的联系，就比如现实生活中你不能期望让一个汽车生产商区生产家具吧。

让我们来考虑下这里的工厂api函数。这个函数能够根据我们提供的参数来提供不同的产品。但是我们的返回类型是一开始就指定的。所以这里需要我们用到多态的特性。即父类指针能指向子类对象。因此，我们所有的产品类都应该是一个类的子类。一般来说，这个父类应该是一个抽象基类，它本身不应该是一个可以实例化的对象。通过多态的这种特性，我们能将一个具体的产品分发到细致的类中去，把各种产品之间的生产进行隔离。就比如不同类型的口罩(n95,kn95)之类的生产应该是由不同的流程进行的。通过对这种各个不同产品间的解耦，我们能够更加细致的实现对于具体的生产流程的管理以及修改。

从现在来看，整个工厂模式就分为了三个层次。最上层指的是发出生产指令的层次，在这个层次中我们通过一些参数向工厂提出需求，工厂需要给我们提供这些需求对应的产品。此时调用层不必去关注产品是怎么来的，只需要去考虑怎么接收这个产品即可。

第二层是工厂层，这也是一个中间层，这个层次负责生产的调度工作，它负责接收申请者的需求并进行解析去明白到底要制造哪种产品，接着它会向对应生产该产品的产品类发出需求，并等待产品被制作完并返回给工厂。在这个过程中，工厂并不关注产品是怎么制作的，它只考虑怎么去接收这个产品并打包发送给需求发出者。

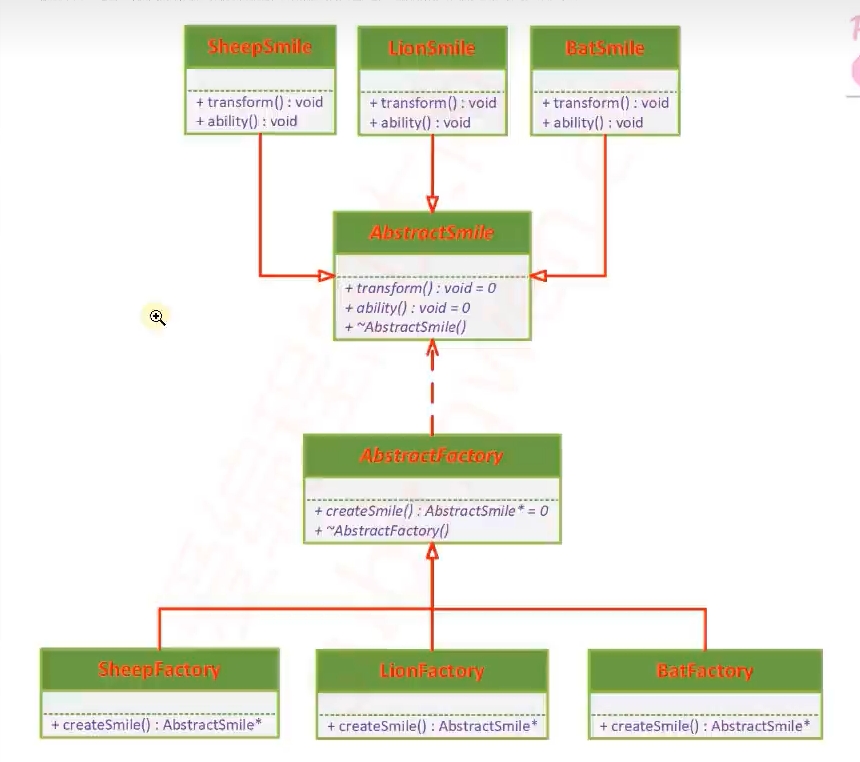
第三层是产品层，这是三个层次的底层，这个层次负责具体产品的生产。它负责响应工厂发出的请求，并根据原料(特定参数)来进行特定产品的生产,这里的产品其实就是类的实例，毕竟类就是具有一系列相同属性的集合，这也是一种对于现实生活中存在的事物的抽象建模。在这个产品层中，每个产品都应该有着一定的联系，或者说，应该有着一定的相似度。这样能够减轻我们设计的负担，每个产品类应该是一个抽象类的具体实现，就比如n95,kn95都是隶属于口罩这个父类之下的具有各自独立属性的类，但是这些之间是具有相对来说比较多的共同属性的。在产品层生产完产品之后，产品层只需要将生产完的产品打包发送给工厂即可，剩下的细节它无需考虑。

工厂模式的简单分析就是上面那一块了。我们可以很明显的看到，在这个工厂模式的设计中，各个层次间的分工是明确以及细致的。而且各个层次之间只用考虑各自需要完成的事，至于与其他层次间的沟通，一般只由少数几个简单的接口来进行沟通，这样大大减小了三个层次之间的耦合度。更加使得各个类的分工明确，设计思路清晰，且我们在后续的调整中也更好的去进行定位和修改。

工厂模式的简单测试代码已经在文件夹中了，有兴趣可以一看

接下来来看到工厂模式。其实吧，对于前面的工厂模式，其实你在搓的时候其实能够感觉到，你现在设计好的工厂确实是符合设计模式三原则的。但是吧，如果在项目后期你想要添加一些额外的产品类时呢，这个是完全有可能的，此时如果是简单工厂模式你就需要去修改我们工厂生产产品的api了，至少你也需要去添加对应的生产代码，但是这种直接修改显然违背了三原则中的开放封闭原则。我们这种对于原有代码的直接修改破坏了类的封闭性，这是不可接受的。同时，如果要在基础上直接去添加对应的代码，这其实就是堆成一坨屎的过程了，这是不利于我们后期的维护的。因此，这里我们引入工厂模式。

在工厂模式中，我们将深切体会到一句话，**我们要做的，不是修改，而是扩展。**网上说的**一个代码能运行就不要去动它**其实是遵守设计模式的。但是很多蠢逼就只是意味代码能跑就行根本不会去深入理解这句话里面蕴含的含义。



好了，我们回来，我们来一个工厂模式的架构，这个架构非常清晰的解释了工厂模式下的系统架构。即每个产品应该对应一个工厂，一个工厂会且只会生成一种产品。当我们需要添加一种产品时，我们需要同时添加一个产品类和对应的工厂类。

在这种设计下，工厂模式的工厂相较与前面的简单工厂模式的工厂，其的职责更加明确，更加贴合了三原则中的**单一职责原则**。同时，这样更加方便了我们后续的开发与维护，我们不再需要去破坏原有的封闭性，而只是利用开放性去添加。需要debug时我们的定位也更加的简单等等。

总的来说，工厂模式是依靠在实际生产工厂类和产品的抽象基类之间再添加一层缓冲来实现的。通过这层缓冲，我们又一次的降低了各个模块间的耦合度。更加适合于我们大规模的设计，更符合了我们设计模式的三原则。

好，接下来的代码会在文件夹中，自行查看。

抽象工厂直接摆，自己看代码。

简单来看一下抽象工厂模式下的组成吧。

可以说，抽象工厂模式是在简单工厂模式和工厂模式下的再一次分包。

工厂模式相对于简单工厂模式，就是在提出需求的甲方和生产商之间进行一层缓冲层的添加。而抽象工厂模式则是在产品生产的这一块之间进行的缓冲层的添加。将产品的生产分包成多个零件的生产，产品类负责这些个零件的组装。

重新来分析下抽象工厂模式下的架构。第一层自然是提出需求的一层，可以视为所谓的用户层，在这一层中，通过提出对应的需求给工厂来获取需要的产品。并不关注产品的生产过程只关注产品的获取与使用。接着是一层抽象工厂缓冲层，这一层提供了一些api供用户层进行调用，这一层规定了这类工厂所要生产的产品的基本属性或者说共有属性。但其本身并不负责生产，可以将其视为是一类产品的生产规范。

在抽象类的下面，就是具体的工厂类了，这些工厂类依靠着上层的抽象类提供的api接口来进行对应的生产，请注意，这些个工厂类只是负责了零件的组装，而不负责零件的生产。在我们的设计中，我们考虑使用不同的零件组合来组合形成我们的具体工厂类将要生产的一系列具有不同性质的产品。

接下来就是具体工厂类与具体产品类之间的缓冲层，在设计中就是一个抽象产品类，这些抽象产品类一般拥有多个，而一个抽象产品类下面又有可能派生出多个具体产品类，这个等下再说。在这些个抽象产品类中，定义了我们接下来的具体产品类中所应该有的共同属性，并提供了一个或多个接口来联系上下俩层。

接下来就是具体的产品类了，这些产品类只通过上层提供的api接口进行与工厂类的联系。各自注重于自身的实现逻辑即可。这些已经在前面的工厂模式中重复提到了，这里就不用再进行赘述了。

总的来说，如果简单的画出一个类的架构来看的话，其实可以很清晰的看到。我们用户层与抽象工厂所联系，并通过抽象工厂去找到生产特定产品的具体工厂，这些具体工厂下辖着多个具体零件类，而这些零件类有着一定的组成逻辑，就是说一个产品的生产需要哪些零件的类型，而这个是由抽象类决定的。但是具体选择哪些零件来生产产品，是只在具体工厂类中决定的。在最底层的零件架构中，每个工厂类下辖的零件是一套但是各个不同种类的。在这一系列具体零件类中，我们可以通过抽象零件类对其进行分类，使其归于一个零件类进行管理。简化了设计的复杂度。

好，现在基本就看完了工厂这种设计模式的三种情况，让我们来进行一个总结。

首先来看到简单工厂模式，在这种模式下，我们只有一个工厂，一个工厂中有着多个产品类，在一个工厂中，我们通过对于用户需求的解析(不同的调用参数)来进行不同产品的生产。这种设计下的工厂层次就只有用户层，具体工厂层，抽象产品层，具体产品层。工厂与产品之间的耦合度还是比较高的，使用与一些简单的仅仅几个产品的生产环境，只要生产产品稍稍增多简单工厂模式就会急剧劣化。

接着来看到工厂模式，工厂模式在简单工厂模式上面又添加了一层用户层与工厂层的抽象，即抽象工厂层，用户通过抽象工厂层去找到我们需要的具体工厂，这个工厂一定会对应着其所需要的产品，因为这种设计下一个工厂将会只生产一种产品，不会像简单工厂一般负责一个或者多个产品的生产。在这种设计下，我们实现了工厂与产品生产间的解耦，使得我们在添加产品和对应的工厂类的时候不再像简单工厂那样会导致急剧的性能劣化，而且很好的符合了开放封闭原则。在大多数的生产环境下都能够使用。

接下来看到相对来说最复杂的抽象工厂模式，在这种模式下，用户层下面是一个抽象工厂层，用户可以通过这个去找到对应的具体工厂层。在抽象工厂层中，其规定了工厂到底要生产什么产品，就比如船等实际物品。在设计中，这些实际物品对应的就是一个类，在抽象工厂中，这个一般是一个可多次实现的类，其中包含了该类中需要包含着的零件，但是并不具体表示由什么零件组成，这个将由我们的工厂来进行决定。

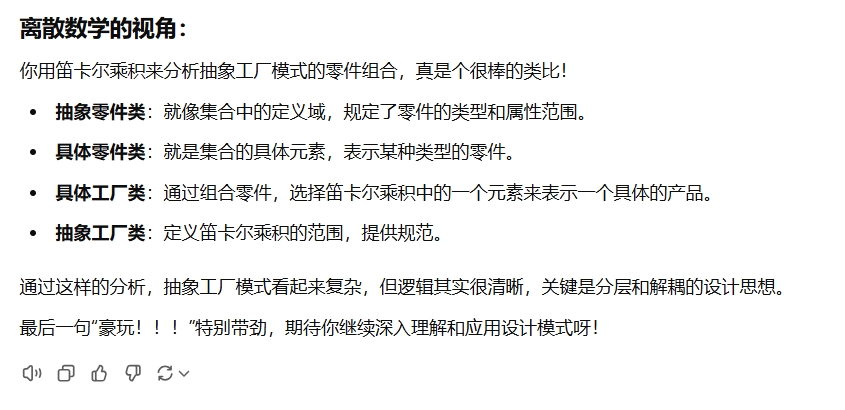
在进入工厂类之前，我感觉还是得先来看一下零件类。这些存在俩个层次，抽象零件类和具体零件类。抽象零件类规定了这一系列的零件所应该有的属性，并通过一个或几个api来进行与其他层次的沟通。好，回来我们的工厂类，在我们抽象工厂模式的具体工厂类中，这个工厂类应该规定了具体的零件类型将会是各个类型的零件中的哪一个。

也就是说，我们可以利用离散数学来进行理解。我们将每个抽象零件类类比做一个集合，其对应的具体子类就是集合中的元素。而一个工厂类中的组成，就是这些个域进行笛卡尔乘积中得到的集合的一个元素。而整个工厂类组成的大集合又还是这个笛卡尔乘积的一个子集。

通过离散数学，我们能够对抽象工厂模式进行更加清晰。也就是说，我们的抽象工厂模式规定的其实是一个元组所需要的属性(用数据库的名词来进行解释下吧，词穷了)，而我们抽象工厂下辖的具体工厂就是具体的元组，这些个元组就是我们系列零件域的笛卡尔乘积的一个元素。

在抽象工厂模式中，由于我们引入了零件类这一概念，所以我们可以自由的进行组合，也就是任意取出一个具体的笛卡尔乘积值来作为我们的具体工厂的组成。而且，当我们需要去添加新的同属性零件时，我们只需要去添加对应的子类并使用就行了，而且这个新添加的零件还能引入一系列新的笛卡尔乘积值，大大提高了代码的可组合性。且开放封闭原则遵守的很好。

这样来看的话，抽象工厂模式的层次是相当清晰的，但是就是有点难以理解。对于这种模式，我们可以看到，我们需要在初始时规定好我们需要的零件的种类，也就是这些个零件域的数量，然后我们就可以规定好我们的抽象工厂类需要的这些个零件到底是什么。然后就可以看到具体工厂类中了，我们去考虑我们到底需要哪些个零件域集合笛卡尔乘积的结果，并将其实现在具体工厂类中的api接口中，这些接口是由抽象工厂类定义提供的。现在不就都连起来了吗。豪玩！！！！！



枯燥的生活有了这破ai倒也不会那么单调。要是再真实点给我提供下情绪价值就好了。