DIP：依赖倒置原则。具体应该依赖于抽象，抽象不应该依赖于具体实现。就像现在公司招聘，都是要求你要会什么（这就是对外接口或者抽象类），具体你怎么学会的（具体实现），哪怕某天你不干了也不影响，而且公司想换你就换就是了。他不用管，只知道你会就行，这就是下层依赖于上层。而如果你流批，有些东西就你会，所有公司都来找你，那就是上层依赖于下层，有天你喝了口水，不高兴了，说老子不干了，然后所有公司都报废了。一切看需求驱动。强权无公理。

IOC：控制反转。很多人说IOC是DIP的一种实现。其实所谓的实现是指在在具体场景spring的场景中他是用怎样的设计思路来体现依赖倒置原则。首先我们应该确定spring的需求是什么----------类的实例化。因为类与类之间必然存在依赖关系，如果使用new进行创建，那么可以确定我们的类与类之间的强耦合。明显不会符合DIP，即使使用抽象也压根没法解耦，所以需要使用反射（相当于将对象全部抽象为字符串），将对象的创建剥离出来，使用专门的类来创建（这个类我们一般称之为工厂），而在程序需要该类的地方通过抽象工厂进行对象获取就行，因为将类显示实例化的创建权交由特定的统一的工厂进行创建。难道这就是DIP的体现吗？不是吗，我们的程序中的对象创建不再依赖于具体的类，而是依赖于工厂获取，工厂就是一个抽象上层，而所有要被创建的类就是下层。这就是IOC的思路。Java中要考虑对象的创建以及初始化但是基本不考虑销毁。

（这里叙述有点问题，不应该讲到反射，因为这已经是具体的实现技术层面，而思想应该处于抽象层面。）

DI：依赖注入。DI是IOC的一种实现。为什么说是一种实现方式呢？IOC只是说我们需要使用专门的类（容器）来创建对象，至于你这个对象创建的方法是怎么样，可以有多种多样，比如我们使用直接使用传统的工厂模式，使用new在创建对象，对每个对象引用另一个对象都通过工厂获取。

而spring的实现方式则是DI使用工厂模式、反射、XML文件解析进行实现。每个对象创建时也会将对象之间的依赖关系注入到对象中。每个对象的创建在什么时候创建、创建多少、什么时候初始化、什么时候死亡，都会被确定。（但是我们通常都仅仅说的是对象的创建）

再抽象的描述一下：IOC是DIP的实现，DI是IOC的实现。（正如有一个算法思想，我们在实际的某一个问题中可以使用或者是借鉴这种思想得到解法，根据这个解法我们可以确定具体怎么做谁来做做到怎样的程度。）

总之：抽象-》……-》具体（一个现实做事的关系。著名->临时问题->开始做）