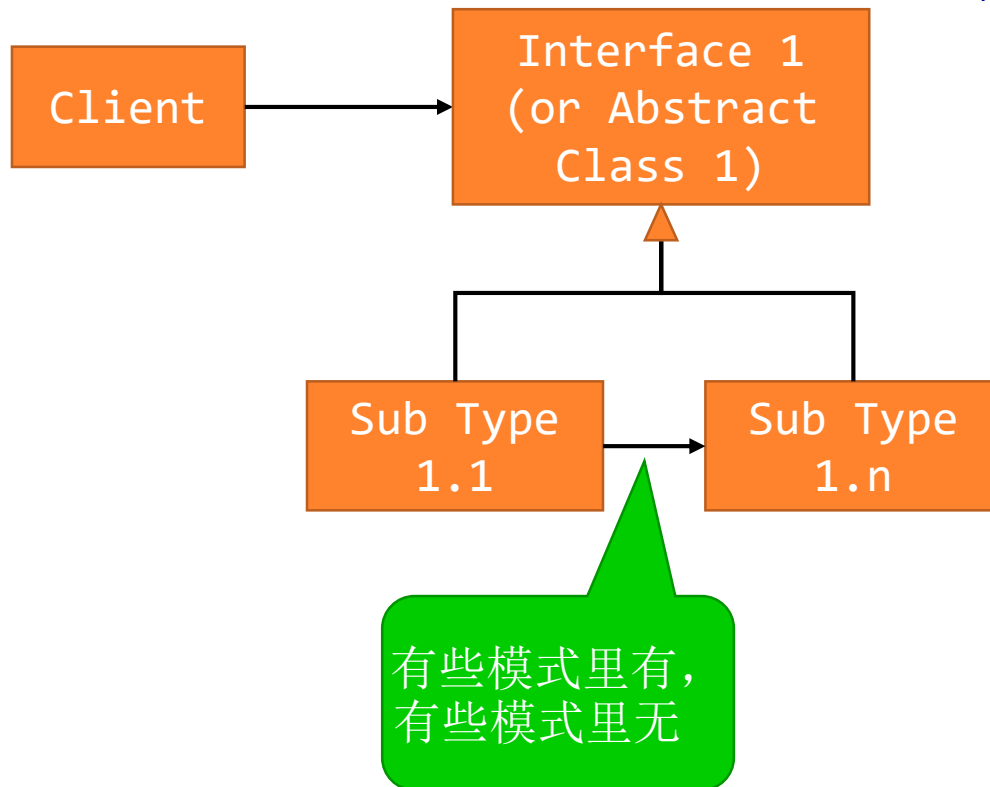


设计模式的对比：共性样式1

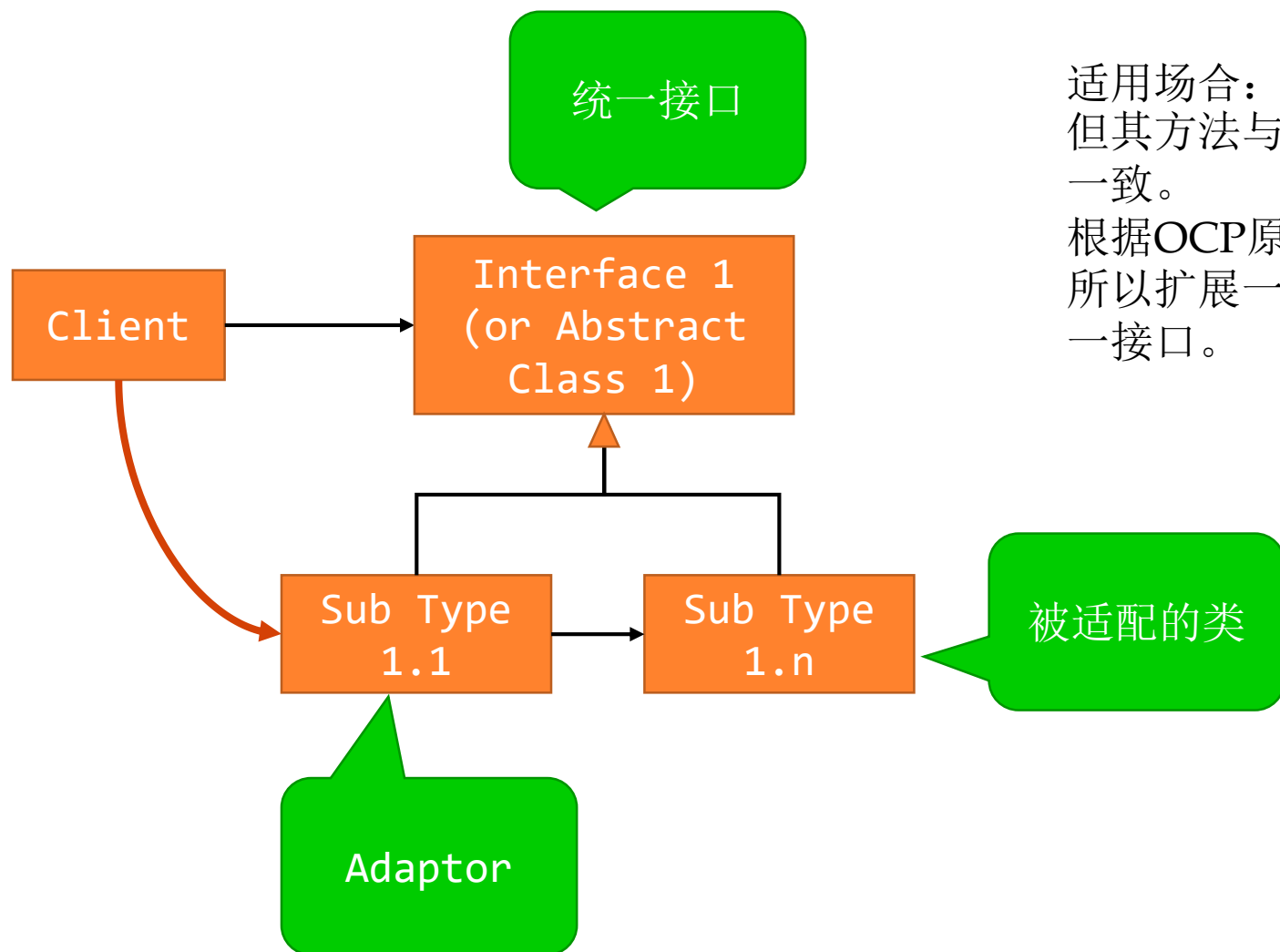


只使用“继承”，不使用“delegation”

核心思路 OCP/DIP

依赖反转. 客户端依赖抽象

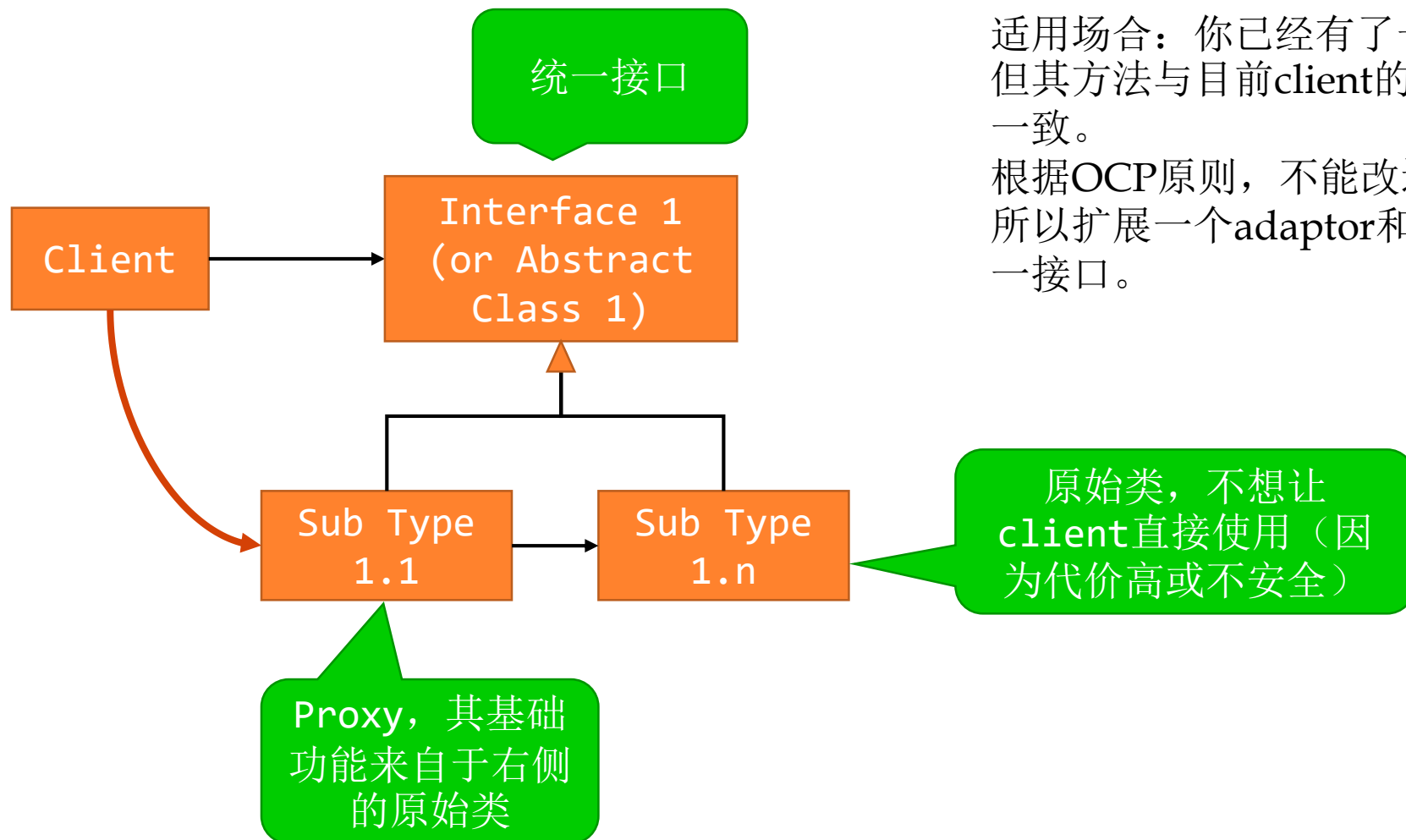
Adaptor



适用场合：你已经有了一个类，但其方法与目前client的需求不一致。

根据OCP原则，不能改这个类，所以扩展一个adaptor和一个统一接口。

Proxy



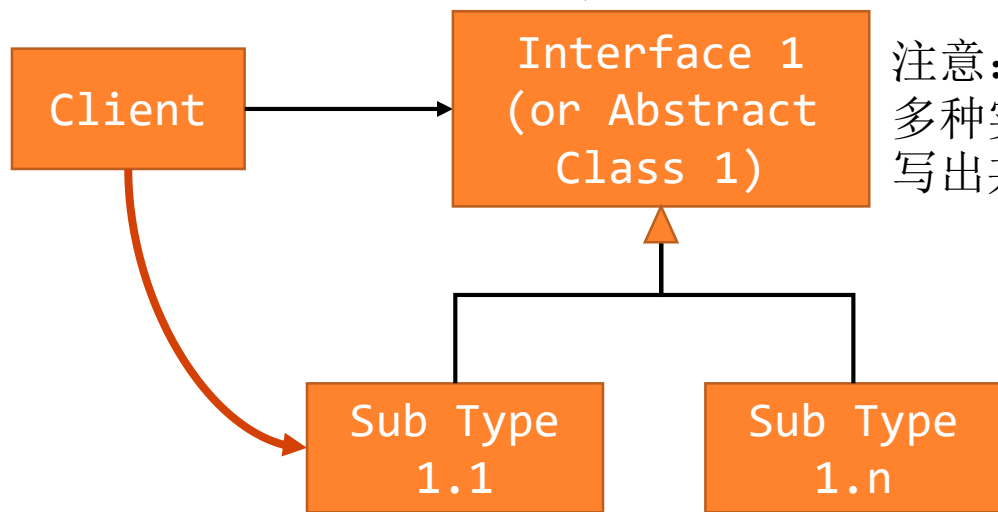
适用场合：你已经有了一个类，但其方法与目前client的需求不一致。

根据OCP原则，不能改这个类，所以扩展一个adaptor和一个统一接口。

Template

- (1) 要提供一个统一的算法方法，**final**的，按次序调用一系列代表算法步骤的**abstract**方法
- (2) 要提供一组**abstract**方法，分别代表算法的某个步骤

适用场合：有共性的算法流程，但算法各步骤有不同的实现
典型的“将共性提升至超类型，将个性保留在子类型”



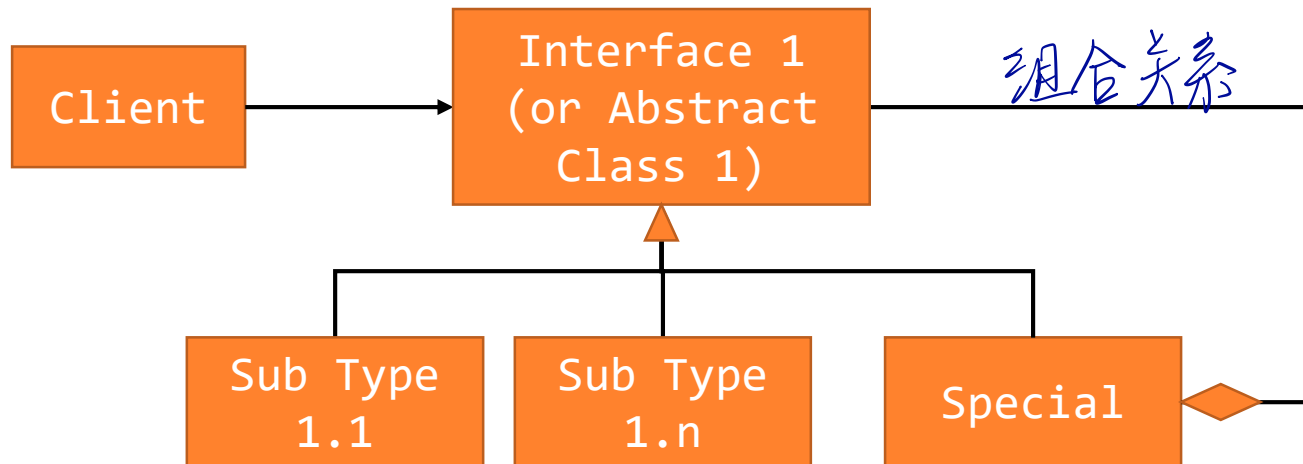
注意：如果某个步骤不需要有多种实现，直接在该抽象类里写出共性实现即可。

每个子类型，只需要实现上面的各个**abstract**方法即可。

设计模式的对比：共性样式2

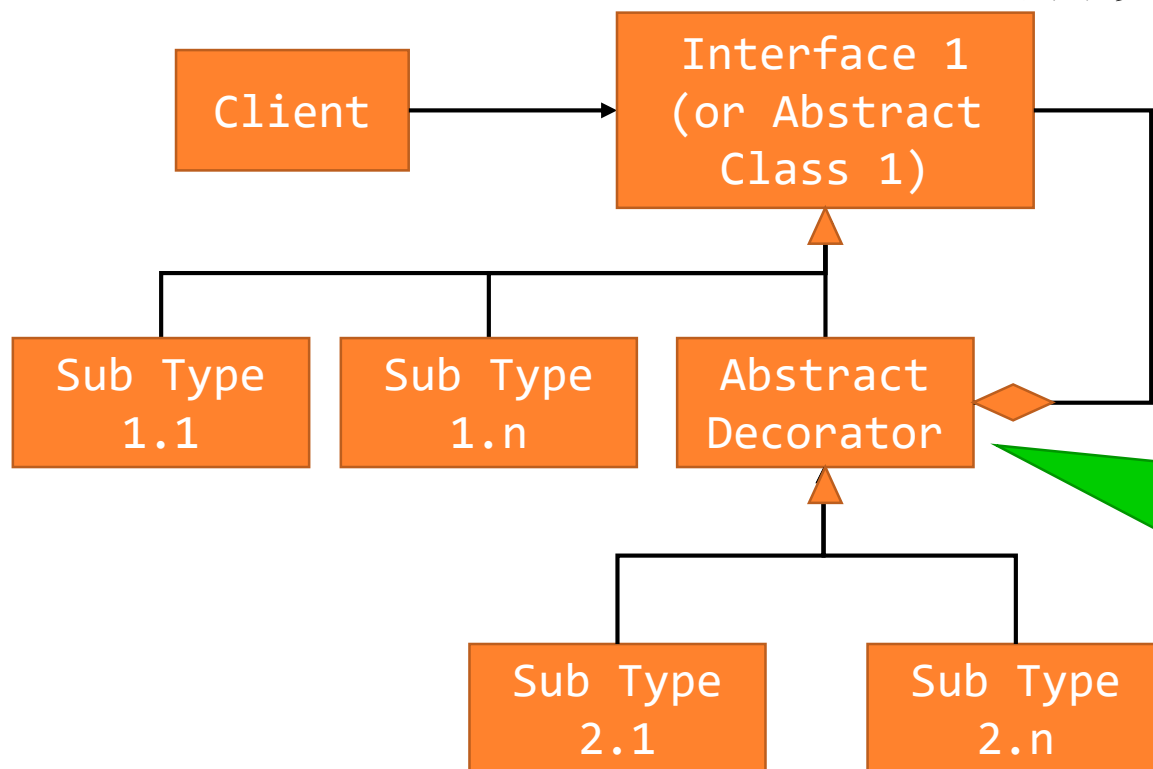
通过继承获得上层抽象接口的基本行为

通过delegation/composition实现递归



Decorator

所以这其实是个递归的设计，多个装饰器子类型可以嵌套的调用，实现对原始ADT对象的多次装饰。
但不管怎么装饰，永远是属于原始ADT的子类型

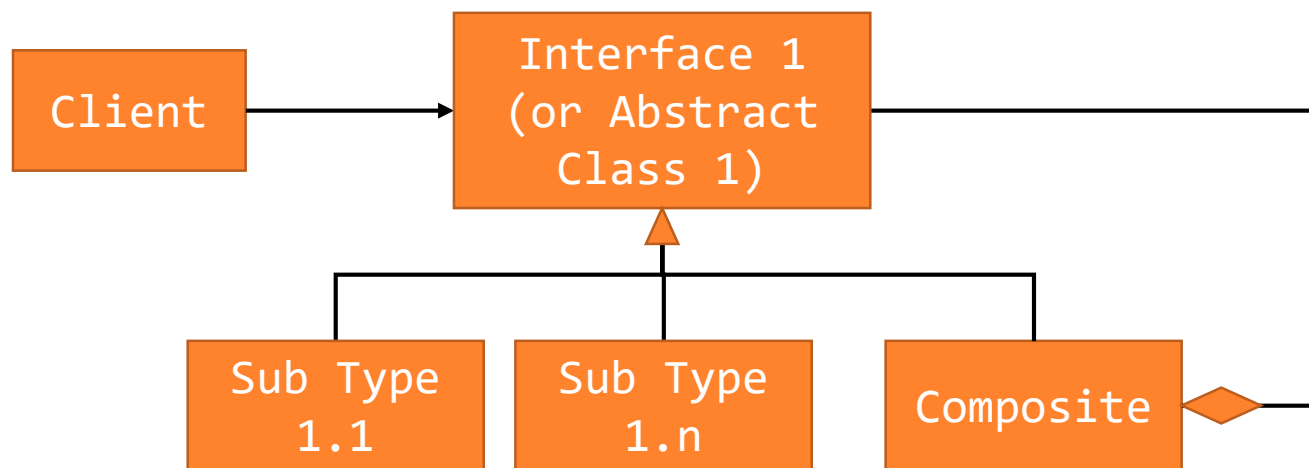


Delegation到ADT接口，所以这里既有继承，又有delegation

继承：具有ADT的所有基本功能，但通过override来扩展功能以实现装饰；
委派：具有一个上面ADT接口类型的变量，通过该变量调用基本功能。

Composite

这也是个递归的设计，每个子类型对象都可以把ADT的任何子类型放入其list当中作为下级，从而形成层次化的树形结构

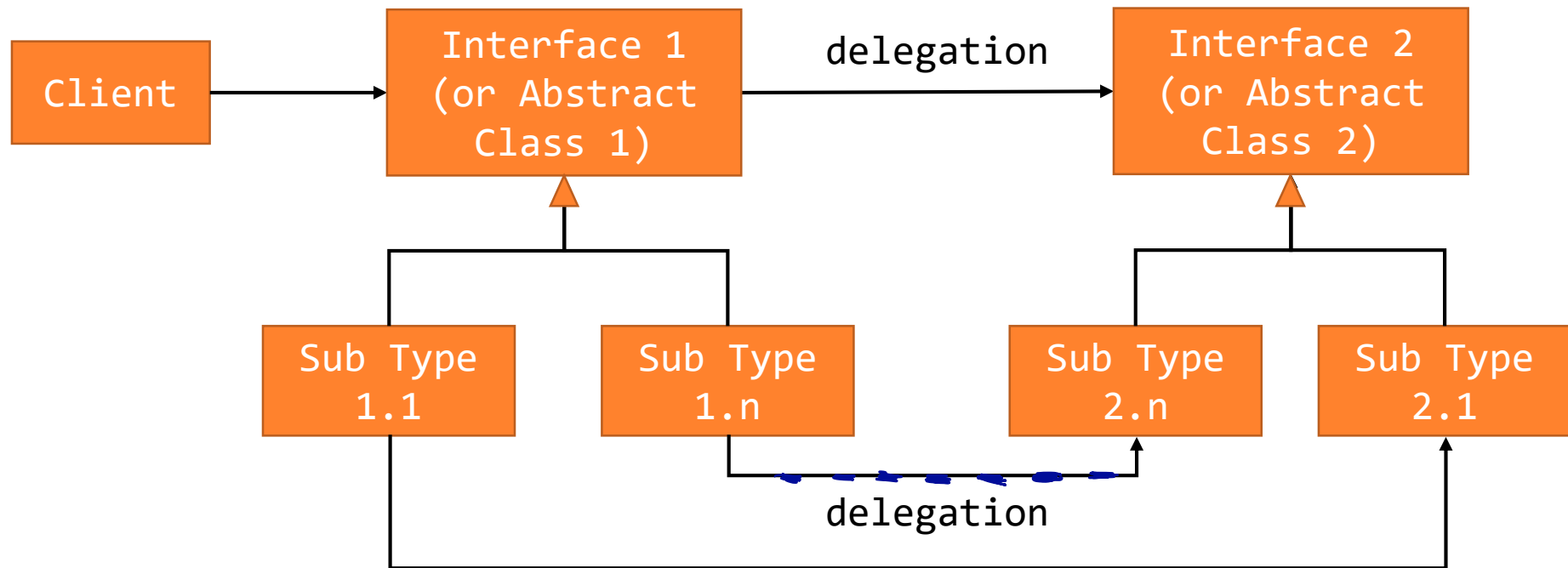


跟Decorator类似，既是继承关系，也由 delegation
(维持着一个 list: 下级对象)

靠继承关系获得 ADT 的所有其他基本功能

设计模式的对比：共性样式3

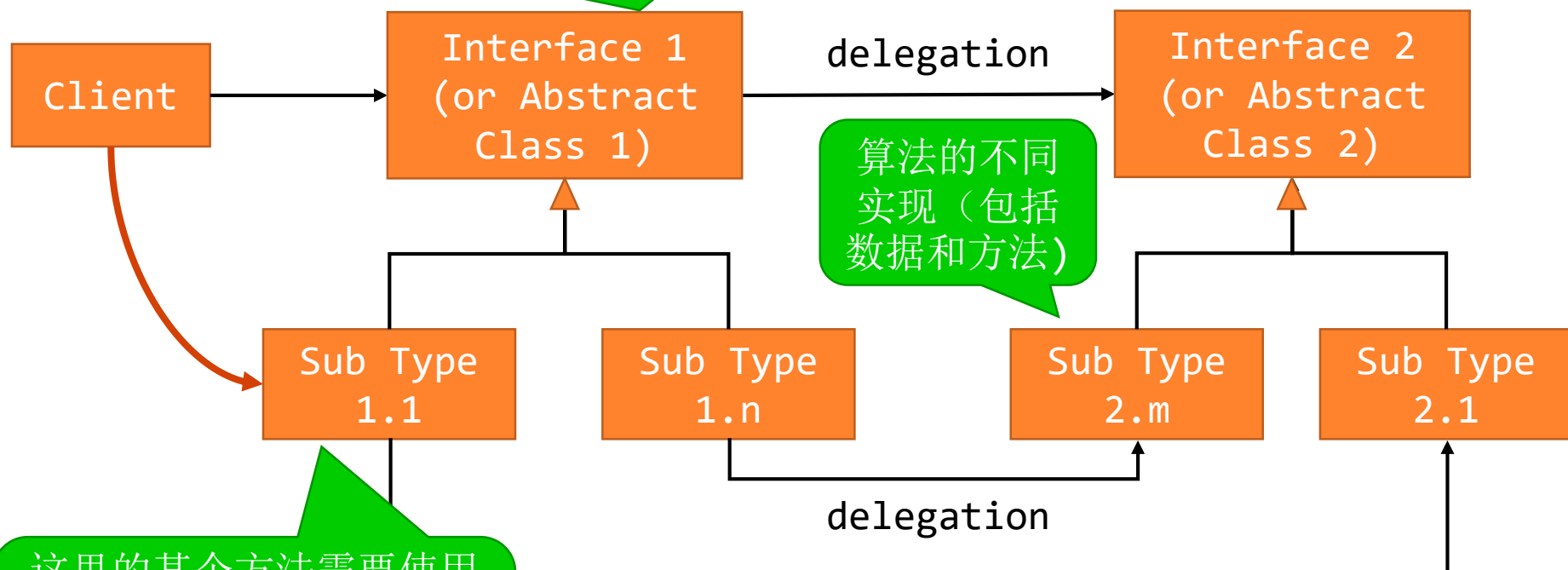
两棵“继承树”，两个层次的“delegation”



Strategy

这里的delegation，不需要永久保持，在使用算法的那个方法里动态传入subtype2.x实例即可，用完就扔掉

根据OCP原则，想有多个算法的实现，在右侧树里扩展子类型即可，在左侧子类型里传入不同的类型实例



这里的某个方法需要使用某个具体算法，运行时动态传入subtype2.x的实例，调用其具体算法

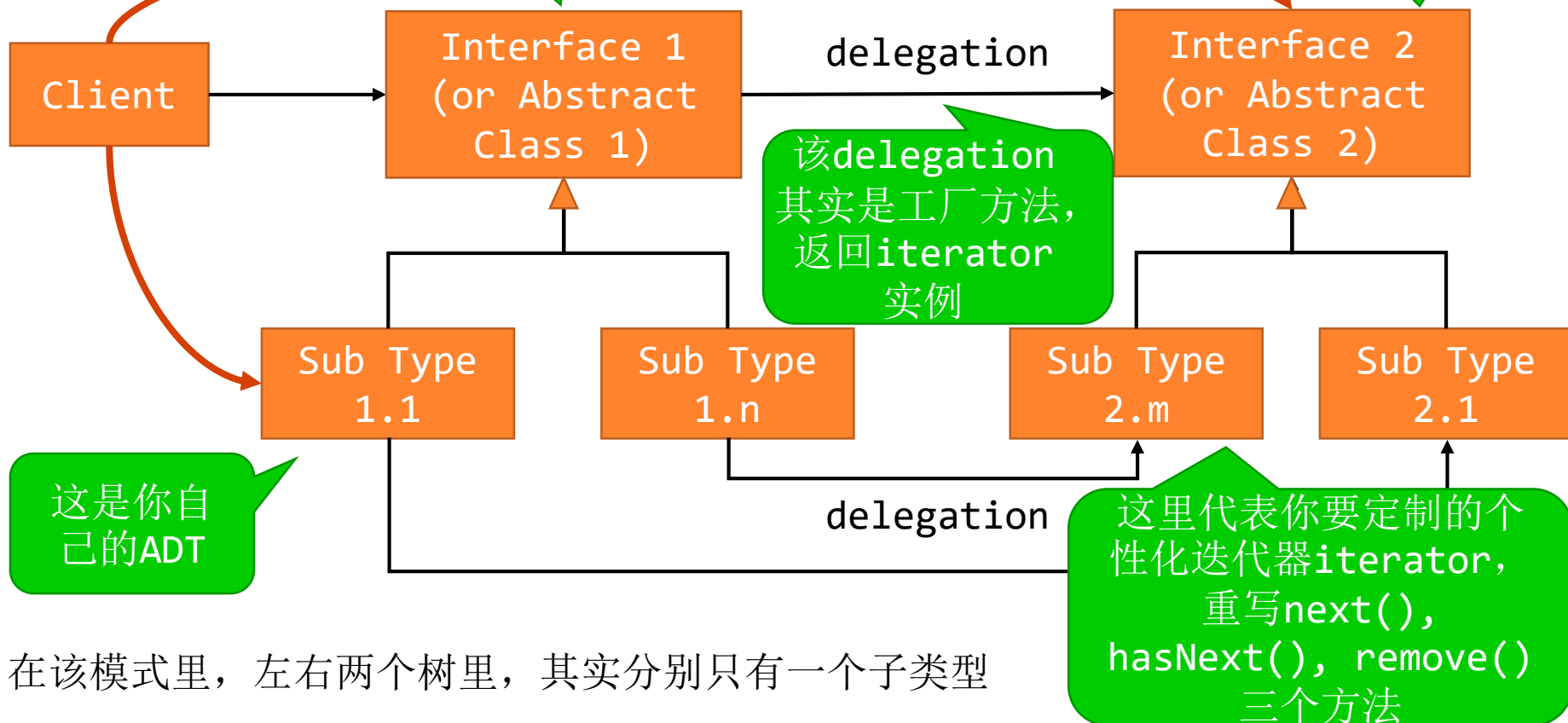
左右两侧的两棵树的子类型，不需要一一对应

Iterator

该关系是指：
client拿到
Iterator实例
之后，用
其遍历集合

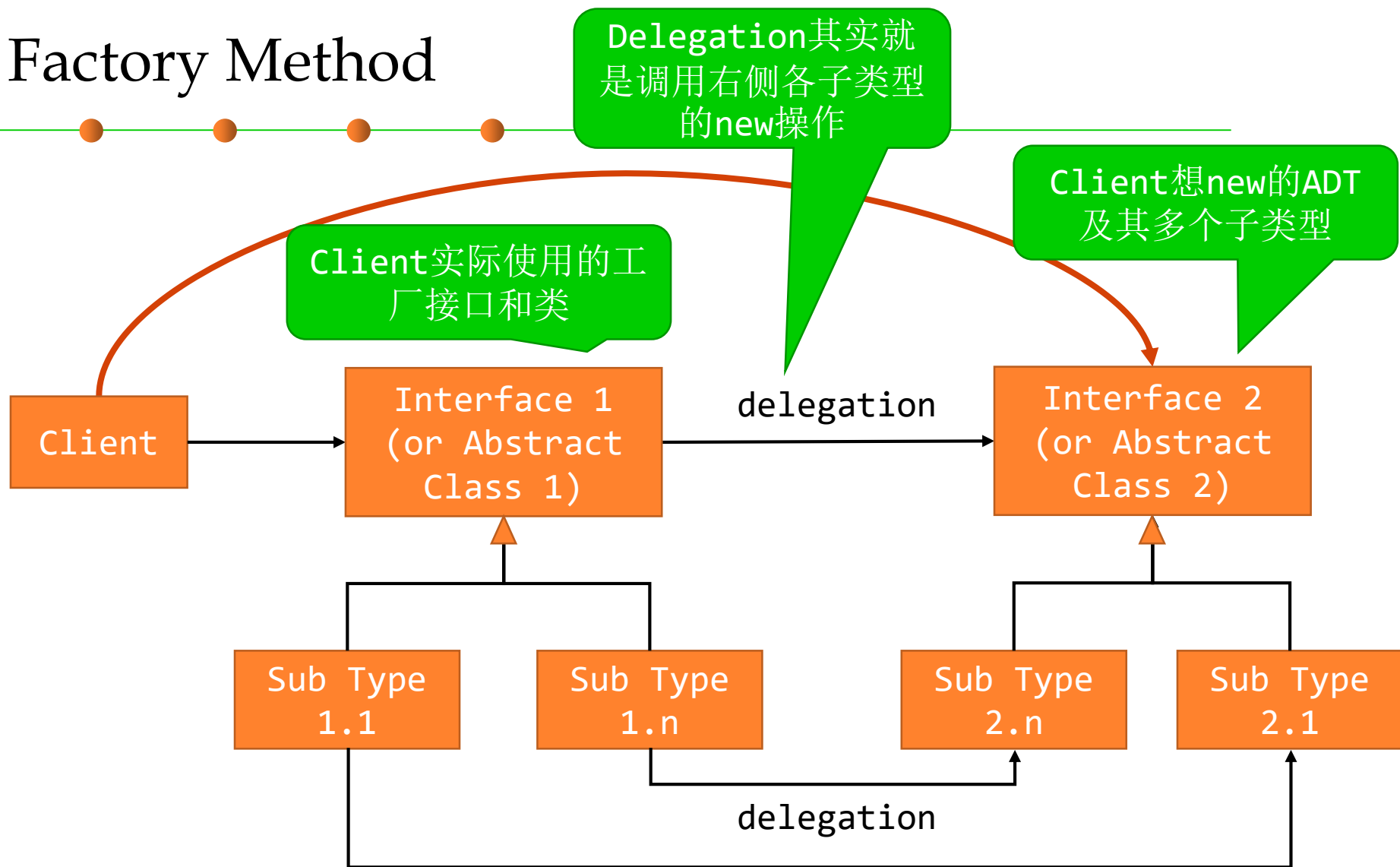
Client希望能在Collection中遍历
ADT，所以ADT要实现这个Iterable
接口，能够通过getIterator返回
迭代器实例。你不需要写这个类，
就是JDK提供的Iterable接口

这里就是Iterator接口，
你不需要写，JDK已经
提供



在该模式里，左右两个树里，其实分别只有一个子类型

Factory Method

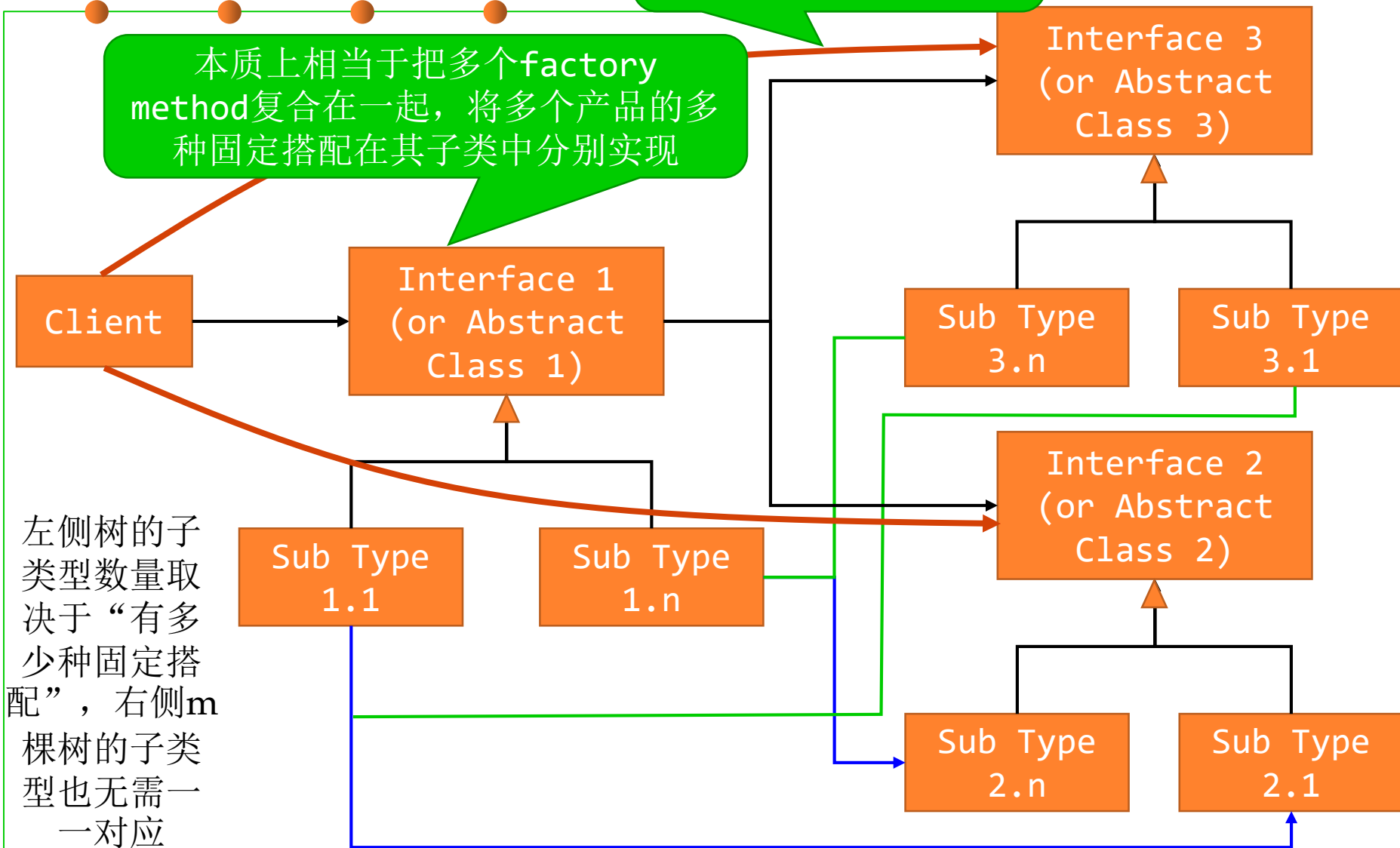


左右两棵树的子类型一一对应。如果在工厂方法里使用type表征右侧的子类型，那么左侧的子类型只要1个即可。

Abstract Factory

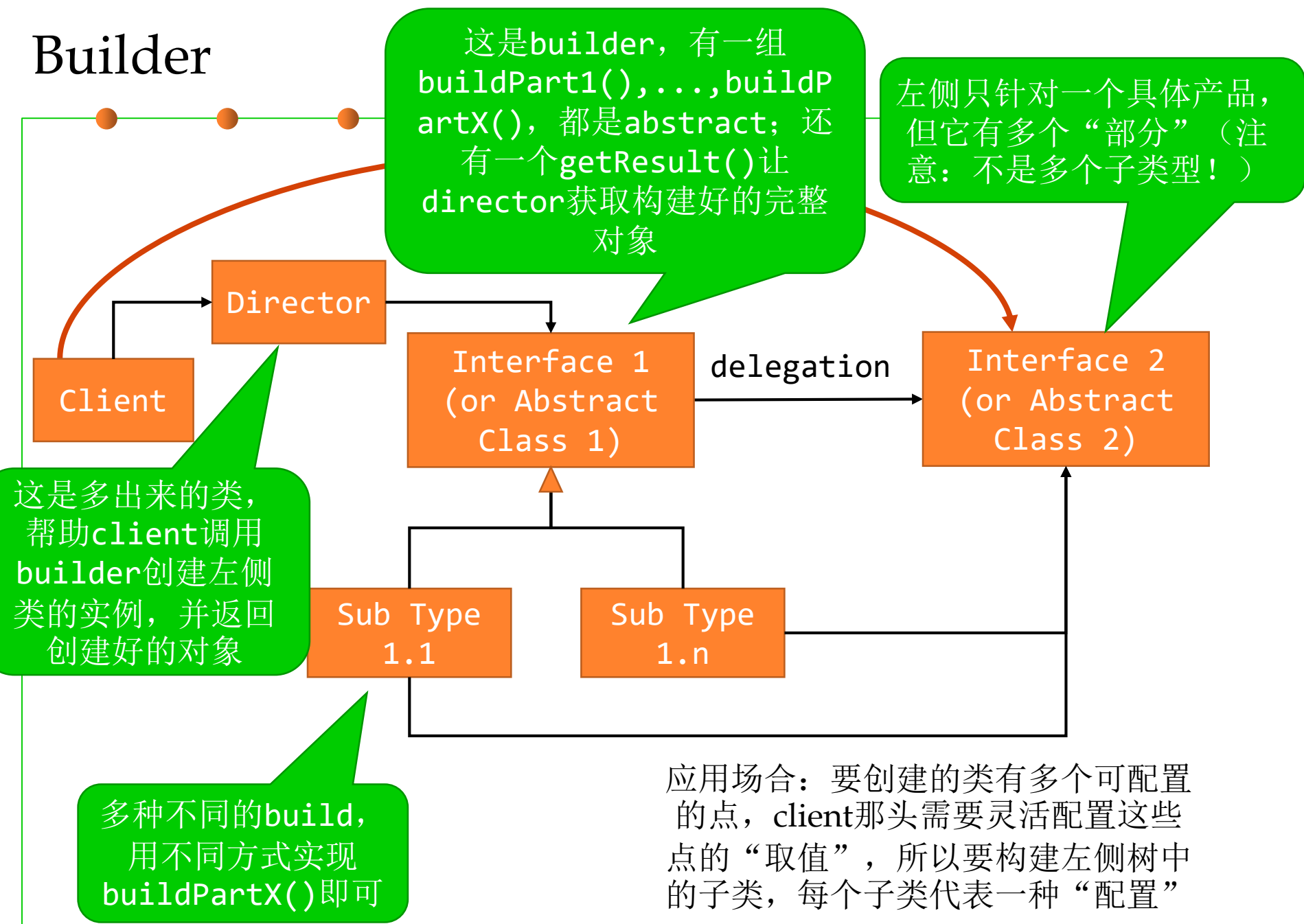
Client从抽象工厂里得到的是右侧多种对象的实例

本质上相当于把多个factory method复合在一起，将多个产品的多种固定搭配在其子类中分别实现



左侧树的子类型数量取决于“有多少种固定搭配”，右侧m棵树的子类型也无需一一对应

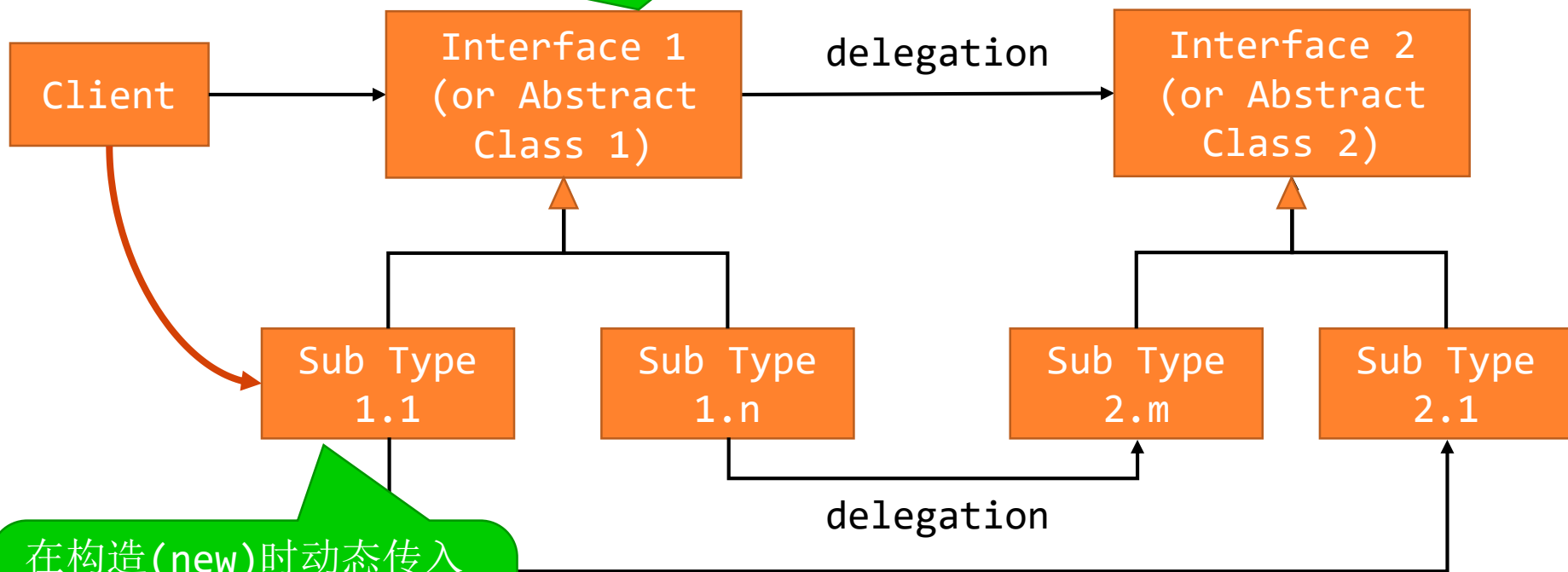
Builder



Bridge

继承树系列A，这个ADT需要持久保留右侧继承树系列B的实例，以便于随时delegate功能到B

其实跟Strategy长得完全一样儿，区分不开就不用区分了...



在构造(new)时动态传入右侧的具体子类型的实例，即可在其他各地方使用传入的实例

左右两侧的两棵树的子类型，不需要一一对应

Observer

这个不需要你自己写，就用JDK提供的Observable抽象类即可

这其实是个“双向”delegation:
(1) 右侧类调用左侧类的addObserver(), 让对方把自己加入队列;

Interface 1
(or Abstract Class 1)

delegation

Interface 2
(or Abstract Class 2)

双向delegation

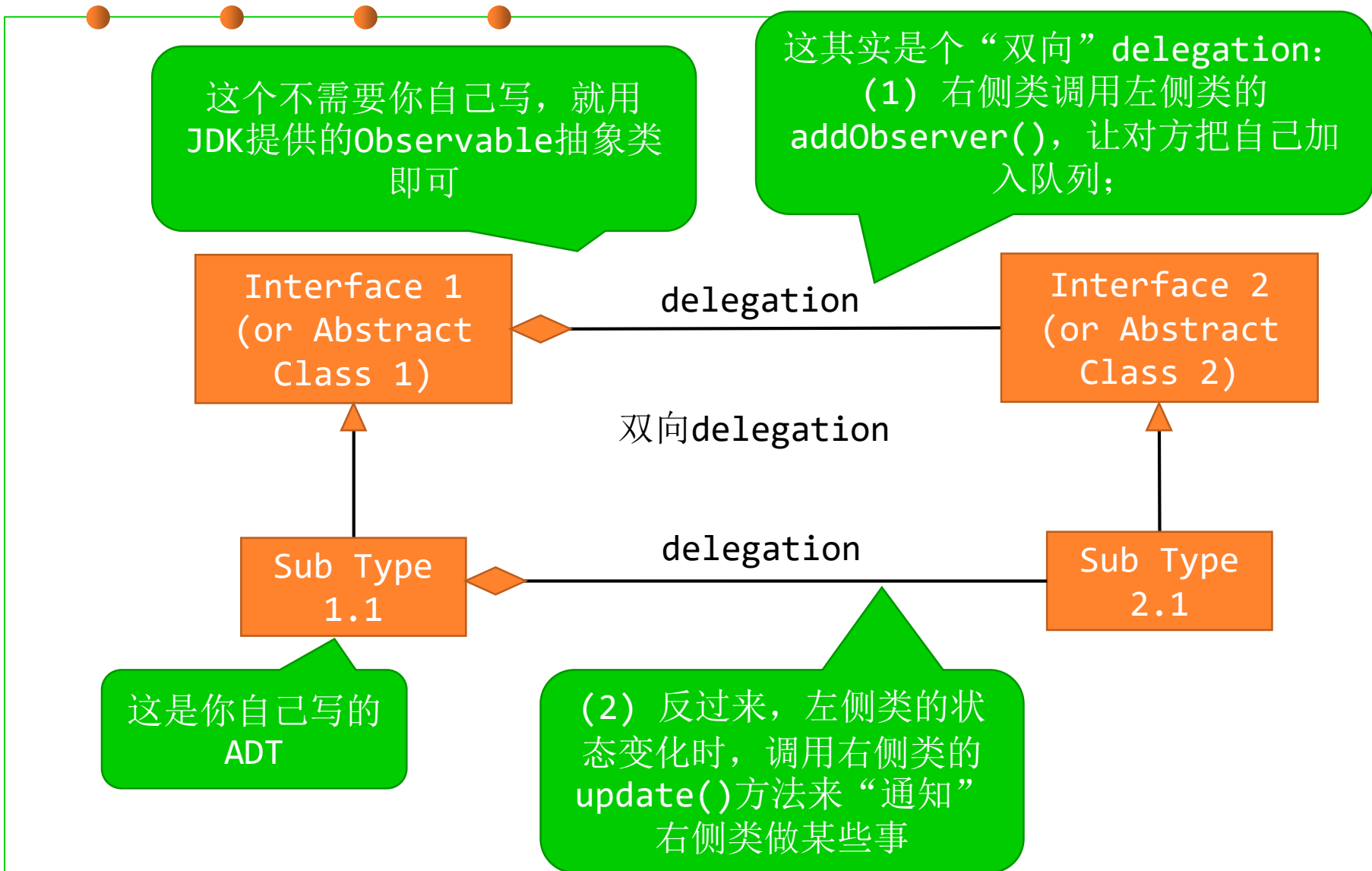
Sub Type
1.1

delegation

Sub Type
2.1

这是你自己写的
ADT

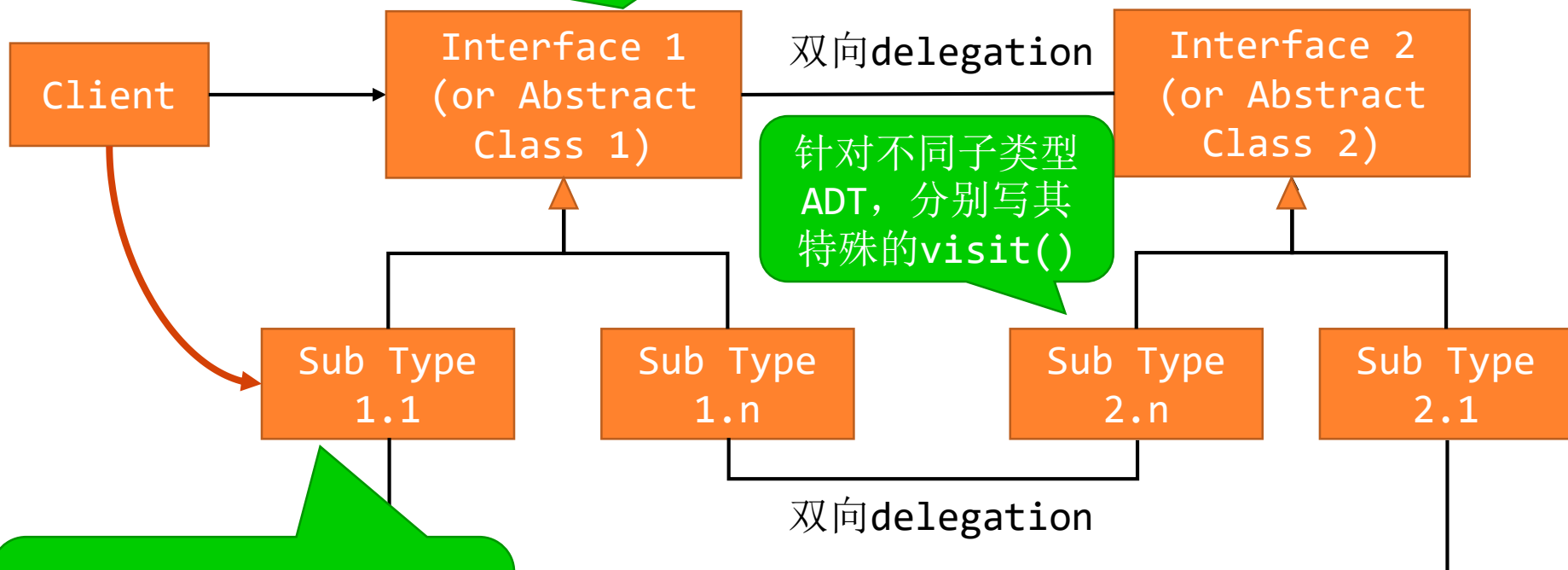
(2) 反过来，左侧类的状态变化时，调用右侧类的update()方法来“通知”右侧类做某些事



Visitor

你设计的ADT，考虑到将来可能要扩展某些操作，但根据OCP，不能再修改其代码，所以提前预留扩展点，即`accept(visitor)`

这是Visitor接口，扩展操作是`visit(ADT)`



针对不同子类型ADT，分别写其特殊的`visit()`

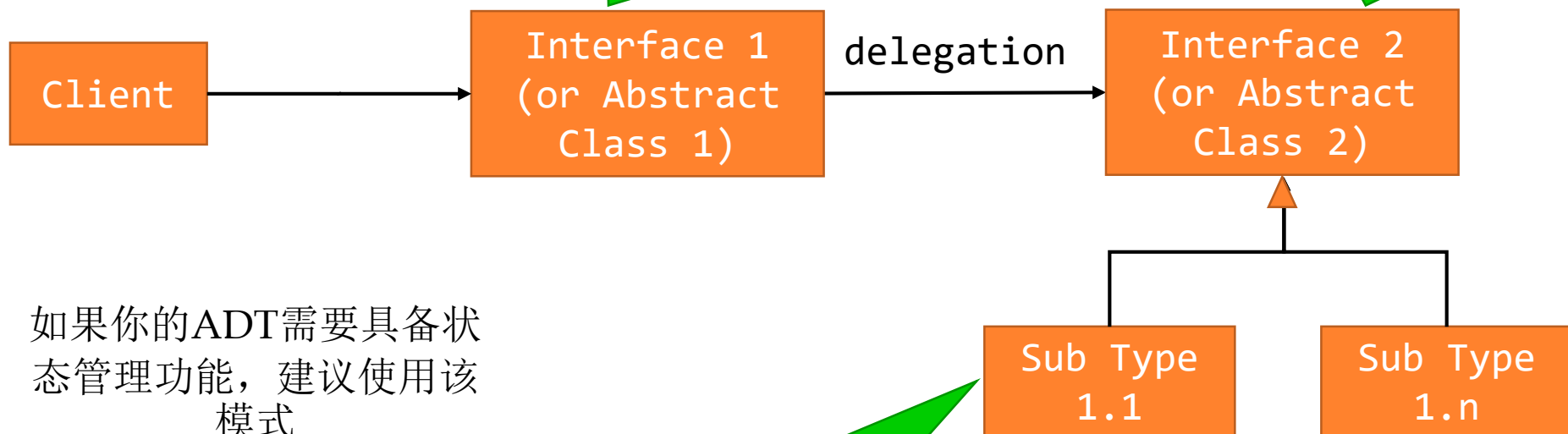
子类型的`visit()`都是同样的写法，不同子类型的`visit()`没差异

左右两侧的两棵树的子类型，基本上是一一对应，但左侧树中的不同子类型可能对应右侧树中的同一个子类型visitor

State

需要具备状态转换功能的ADT，维持一个State对象表征其当前状态，并提供一系列状态转换方法，各方法内部调用State对象的相应操作来完成。

State接口，定义一系列状态转换操作



如果你的ADT需要具备状态管理功能，建议使用该模式

每个子类型用不同方式实现状态转换操作，返回新状态的实例

Memento

但是ADT自己不维护各个备忘录，而是交给Caretaker来负责，它管理着所有的备忘录（加入新的、从中取出旧的）

增加Memento类，每次ADT要备忘时，都生成一个memento

需要备忘的ADT

这是State设计模式

State抽象接口

