哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

Chapter 6: Maintainability-Oriented Software Construction Approaches

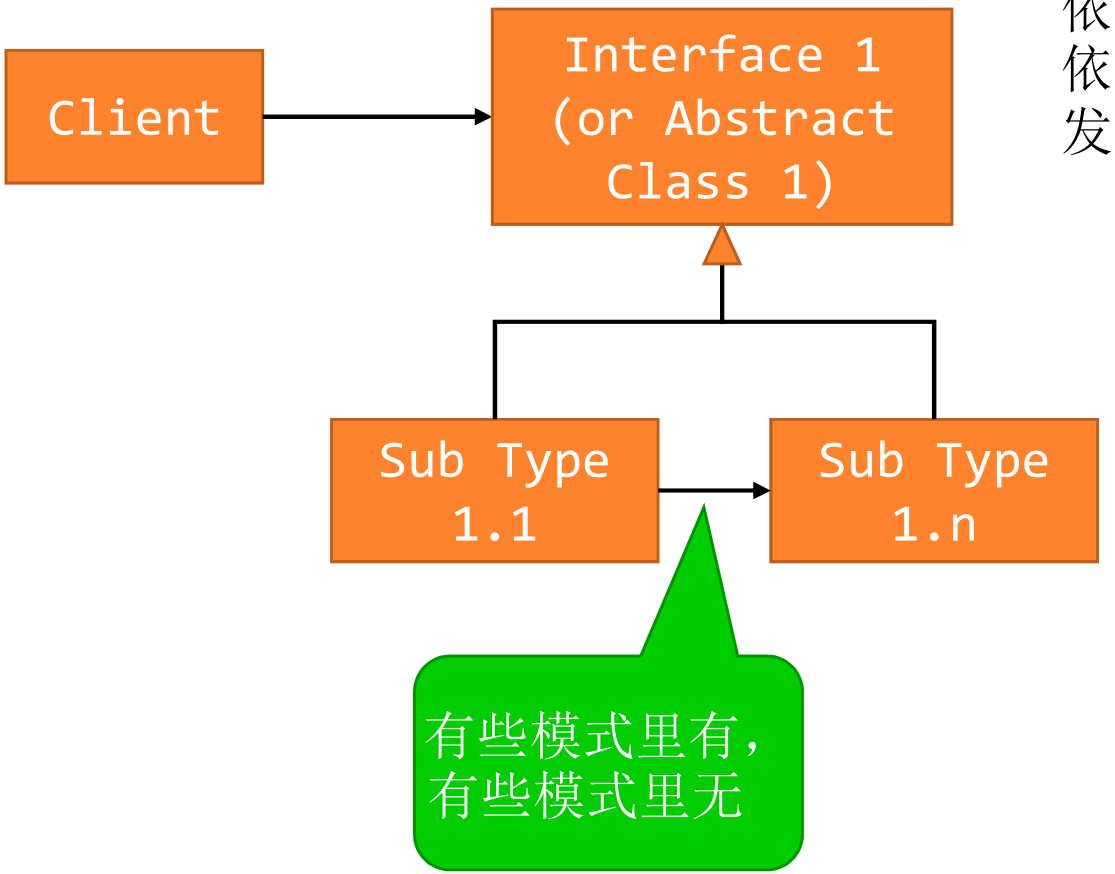
6.4 Commonality and Difference of Design Patterns

设计模式的共性和差异

Wang Zhongjie
rainy@hit.edu.cn

April 17, 2019

设计模式的对比：共性样式1



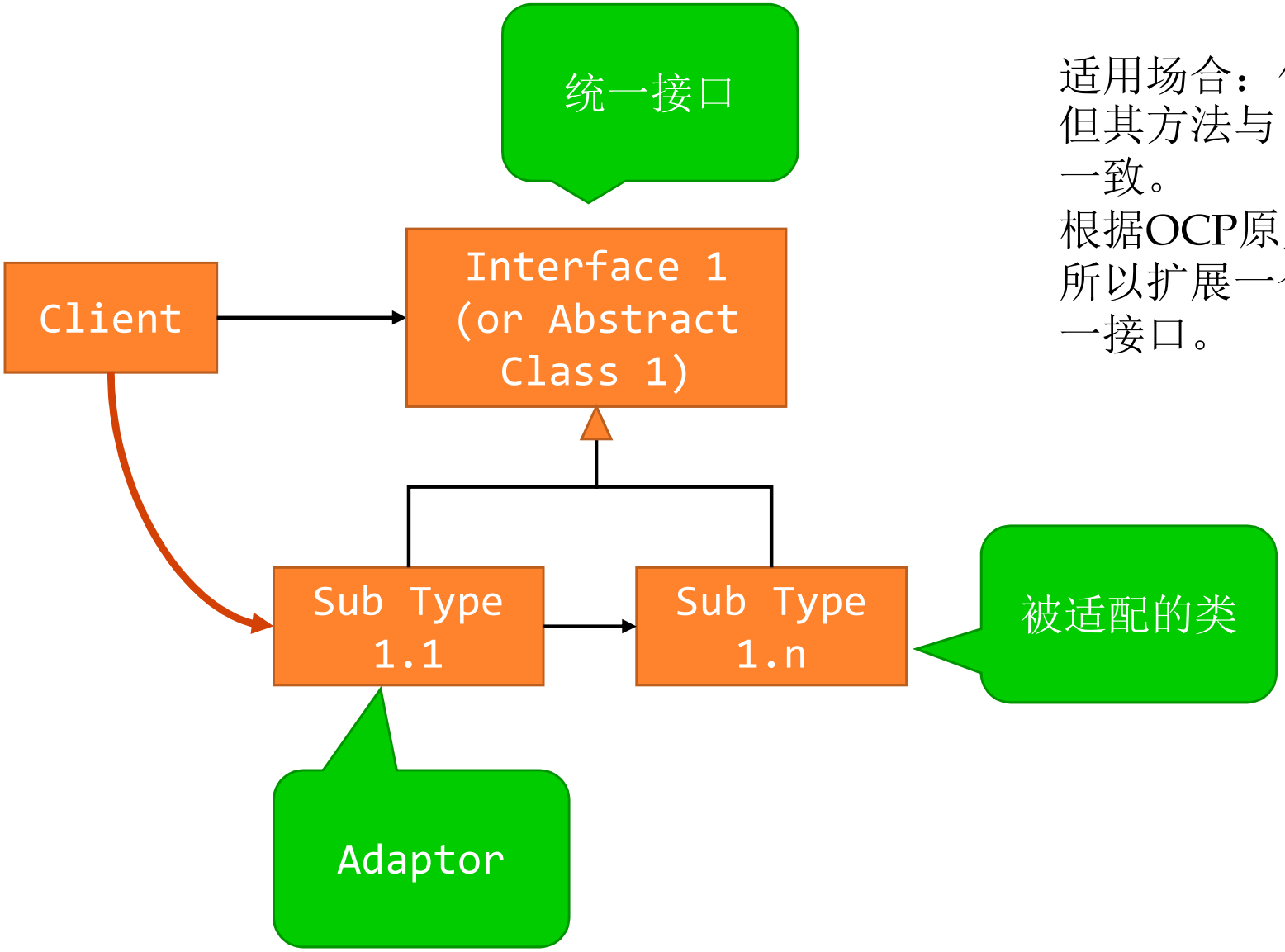
只使用“继承”，不使用“delegation”

核心思路：OCP/DIP

依赖反转，客户端只依赖“抽象”，不能依赖于“具体”

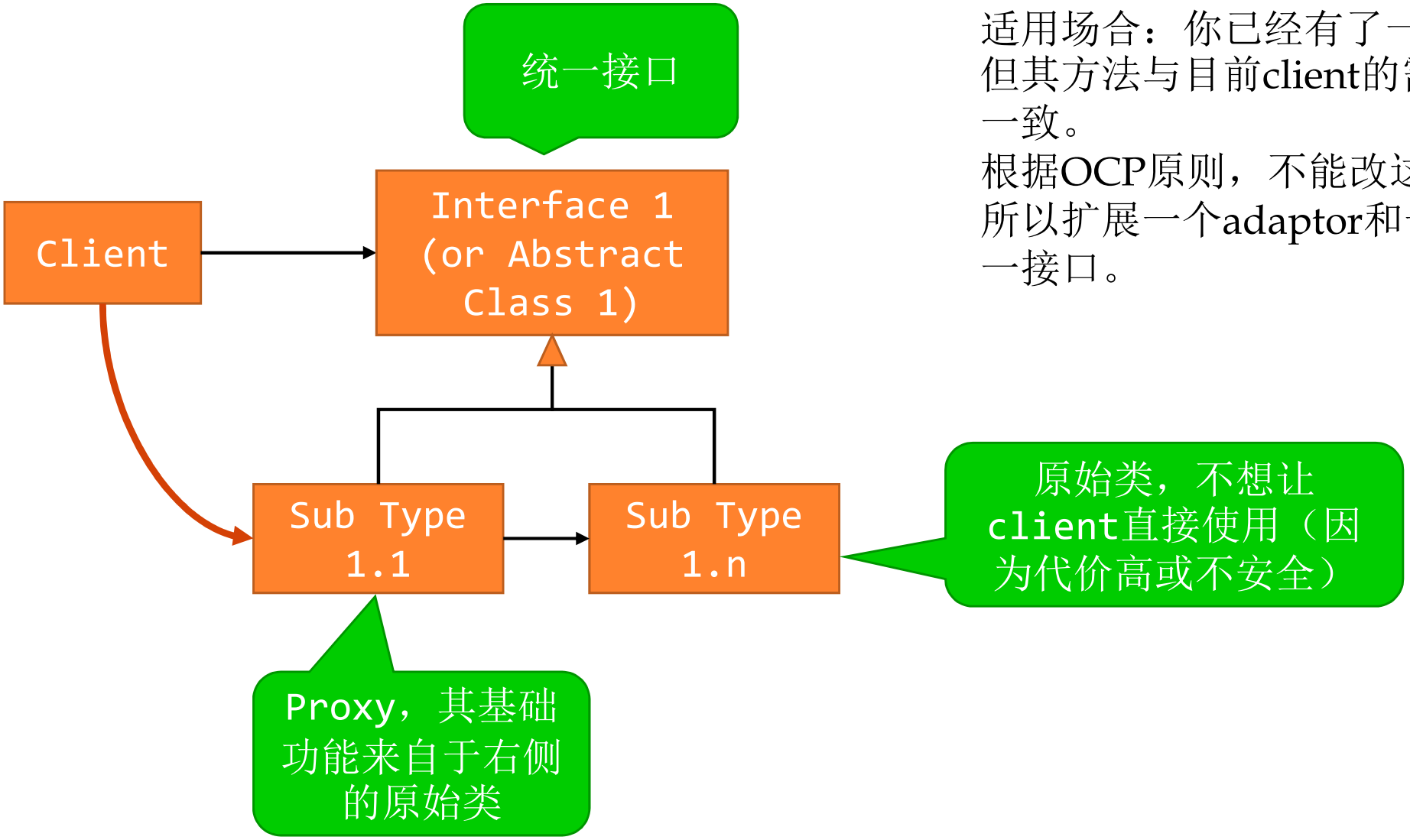
发生变化时最好是“扩展”而不是“修改”

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

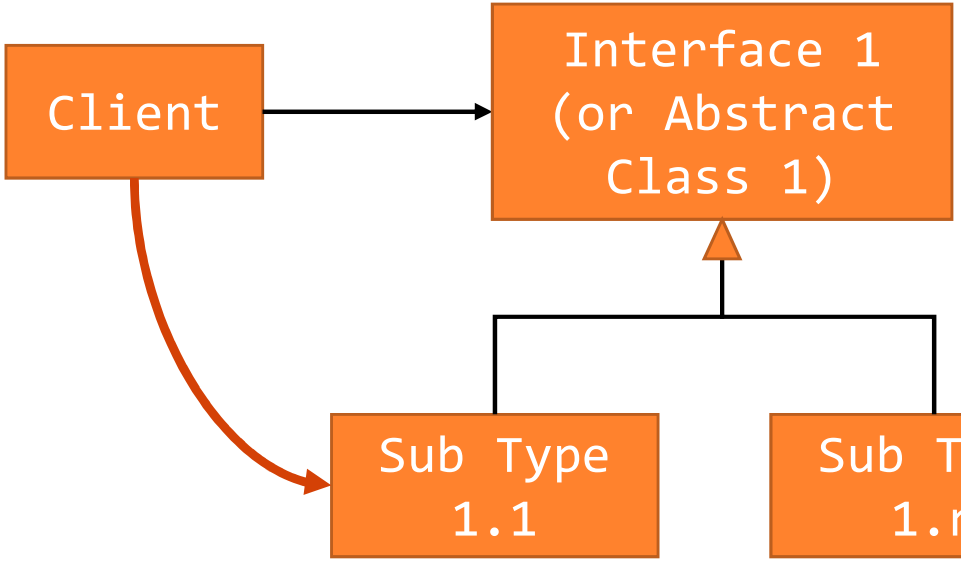


适用场合：你已经有了一个类，但其方法与目前client的需求不一致。
根据OCP原则，不能改这个类，所以扩展一个adaptor和一个统一接口。

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”



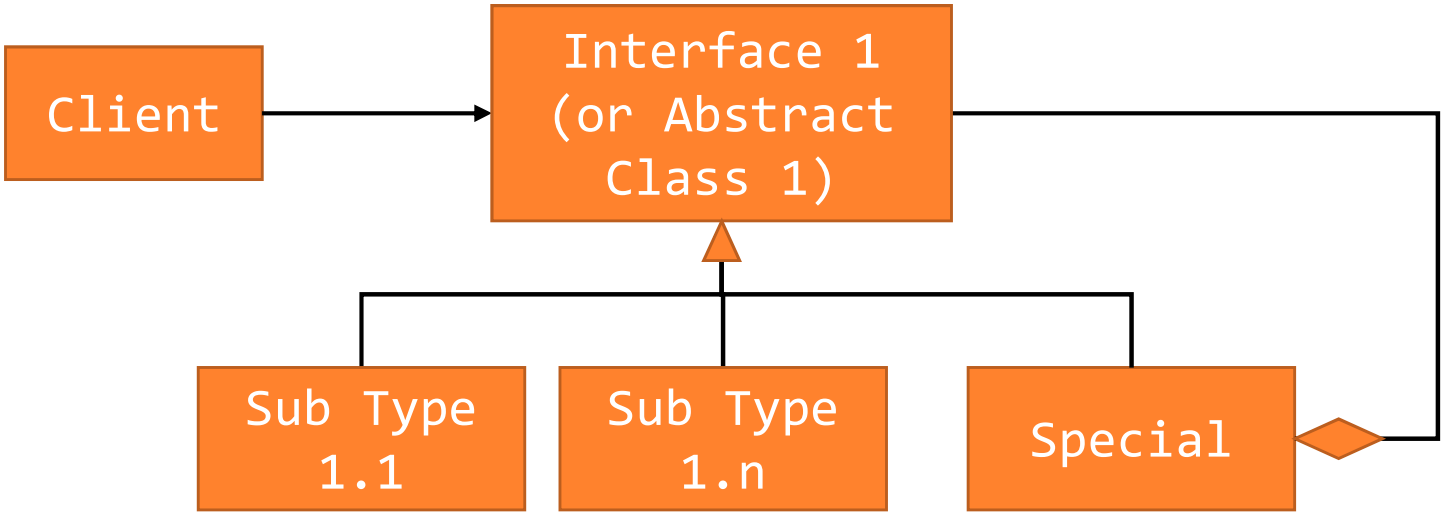
注意：如果某个步骤不需要有多种实现，直接在该抽象类里写出共性实现即可。

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

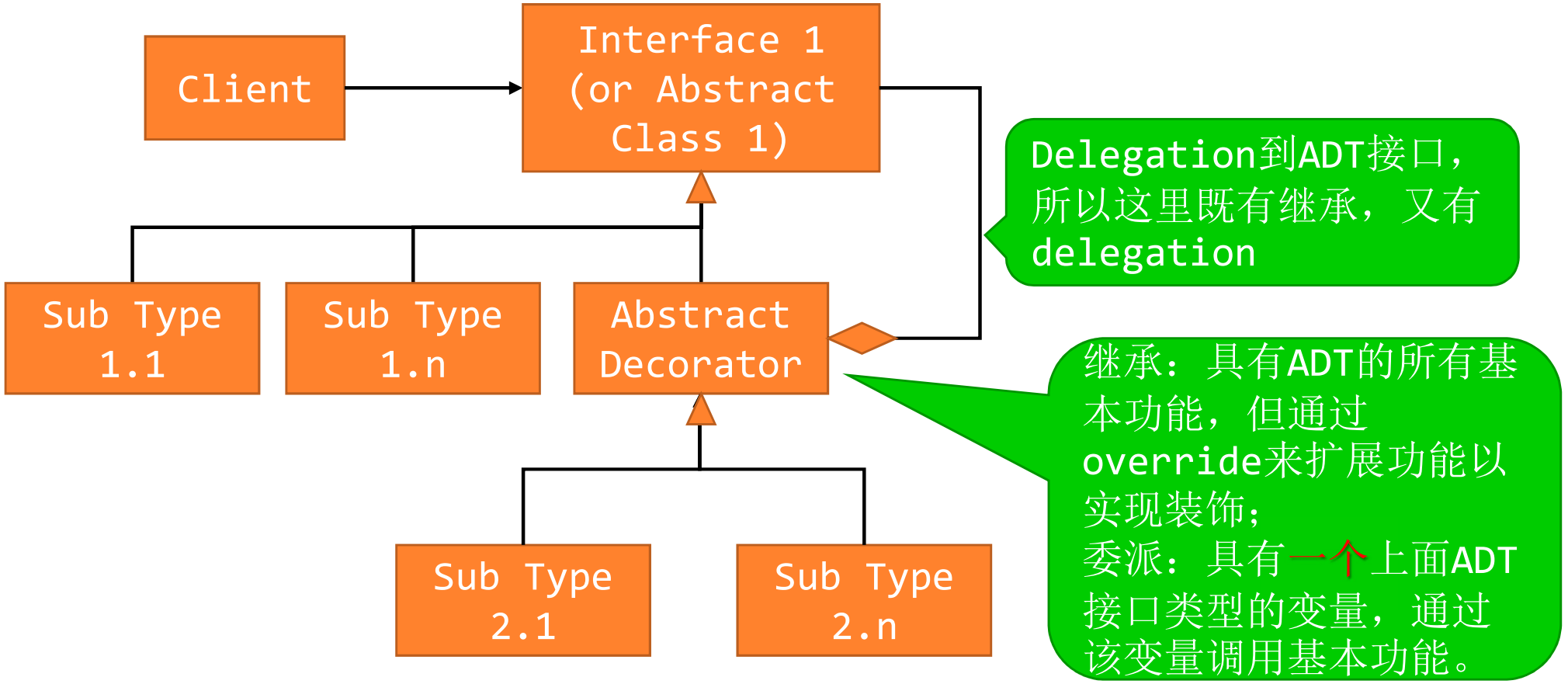
通过继承获得上层抽象接口的基本行为

通过delegation/composition实现递归



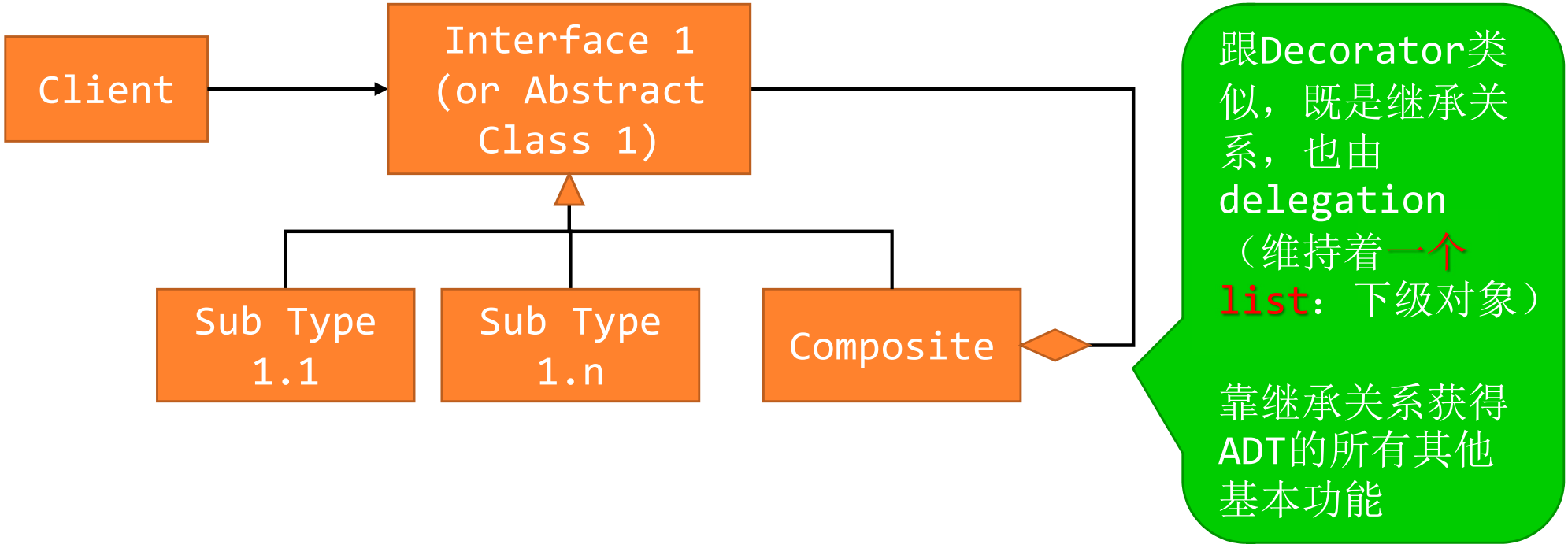
Decorator

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



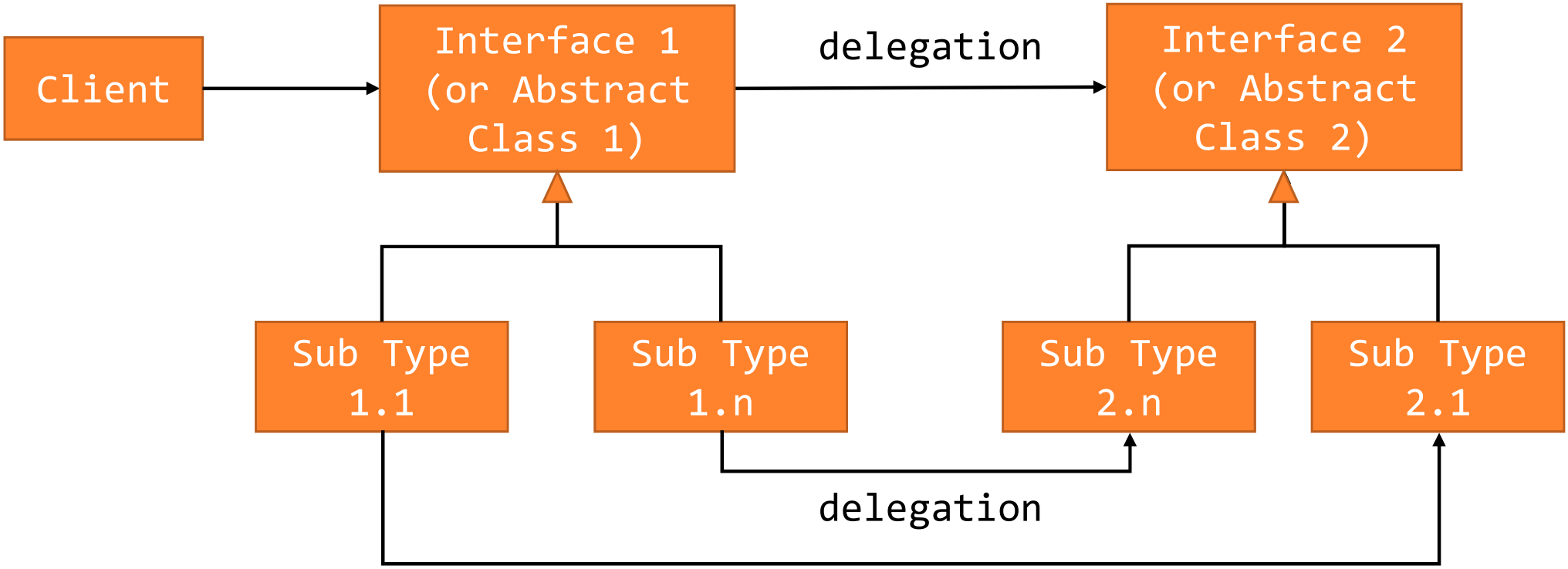
Composite

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

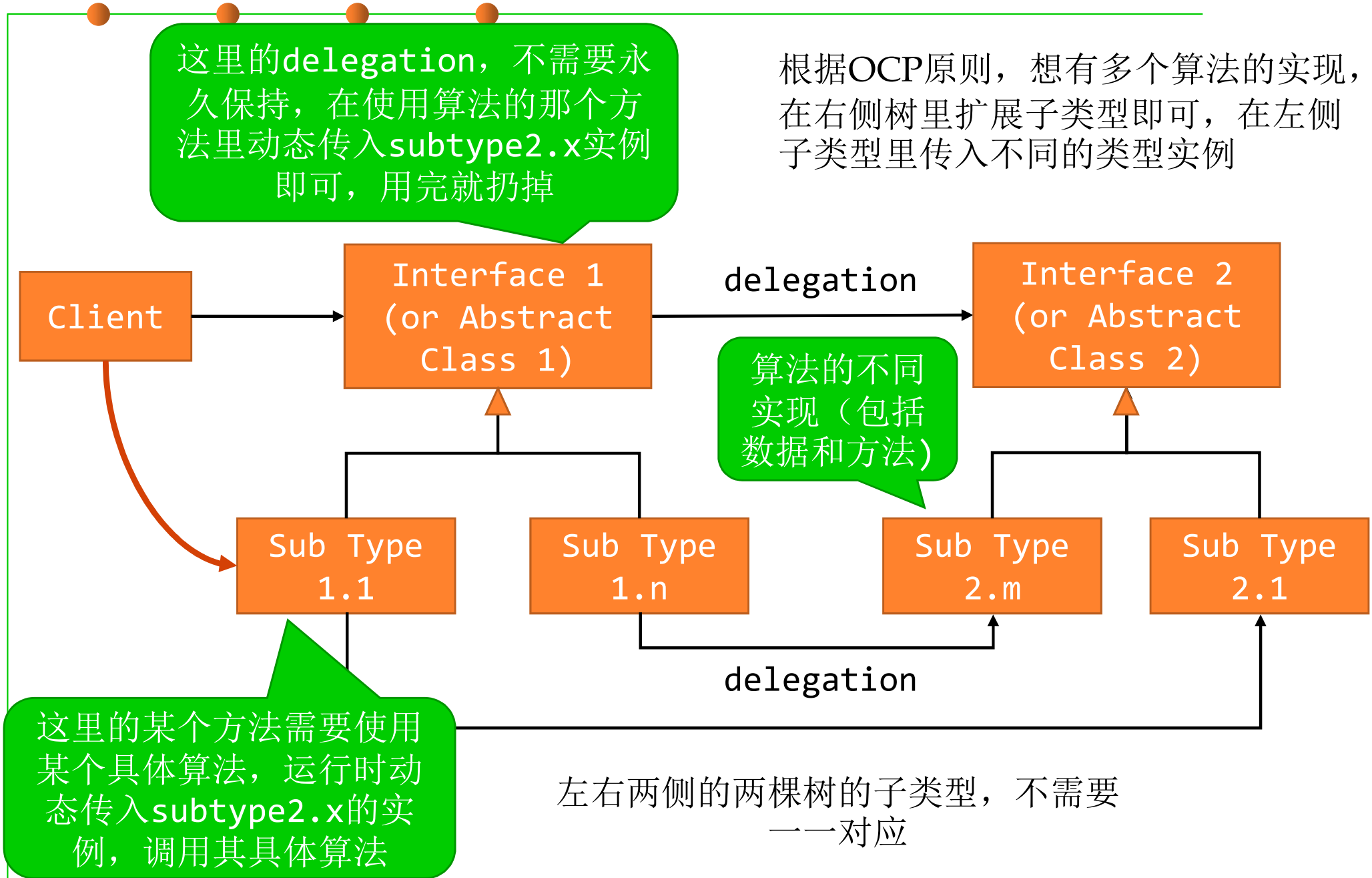


设计模式的对比：共性样式3

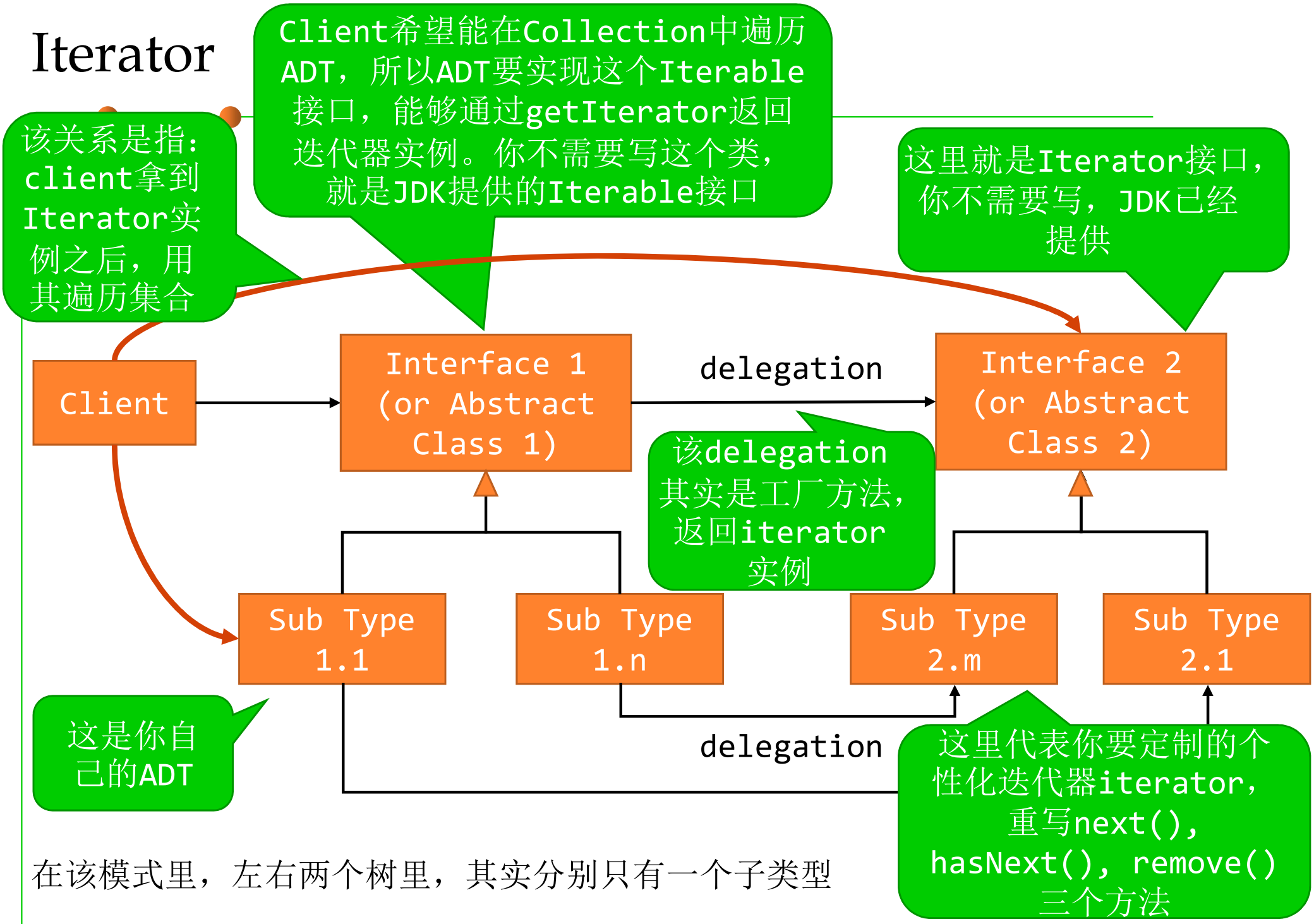
两棵“继承树”，两个层次的“delegation”



Strategy

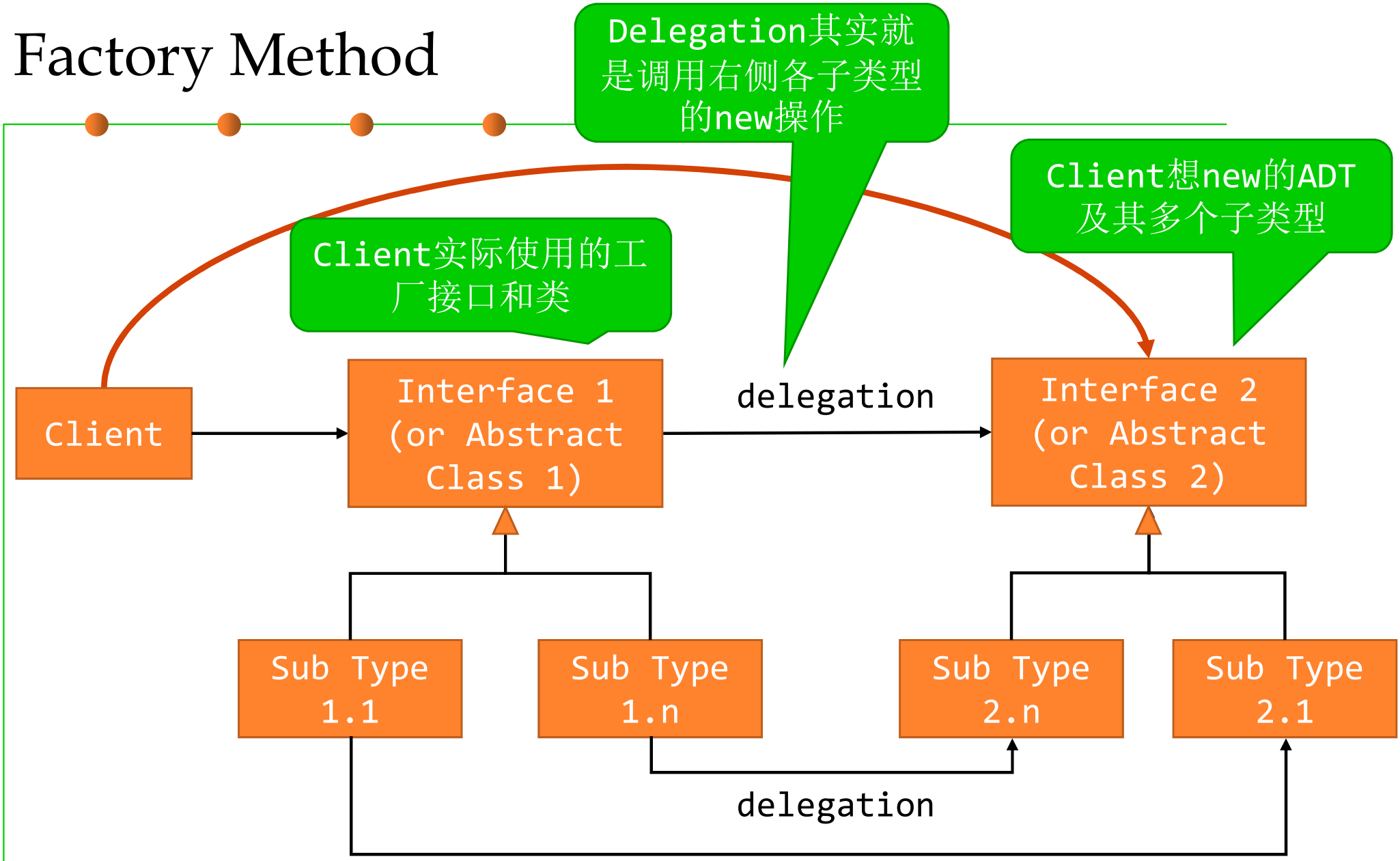

这里的delegation，不需要永久保持，在使用算法的那个方法里动态传入subtype2.x实例即可，用完就扔掉
根据OCP原则，想有多个算法的实现，在右侧树里扩展子类型即可，在左侧子类型里传入不同的类型实例
Client
Interface 1 (or Abstract Class 1)
delegation
Interface 2 (or Abstract Class 2)
算法的不同实现（包括数据和方法）
Sub Type 1.1
Sub Type 1.n
Sub Type 2.m
Sub Type 2.1
delegation
这里的某个方法需要使用某个具体算法，运行时动态传入subtype2.x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



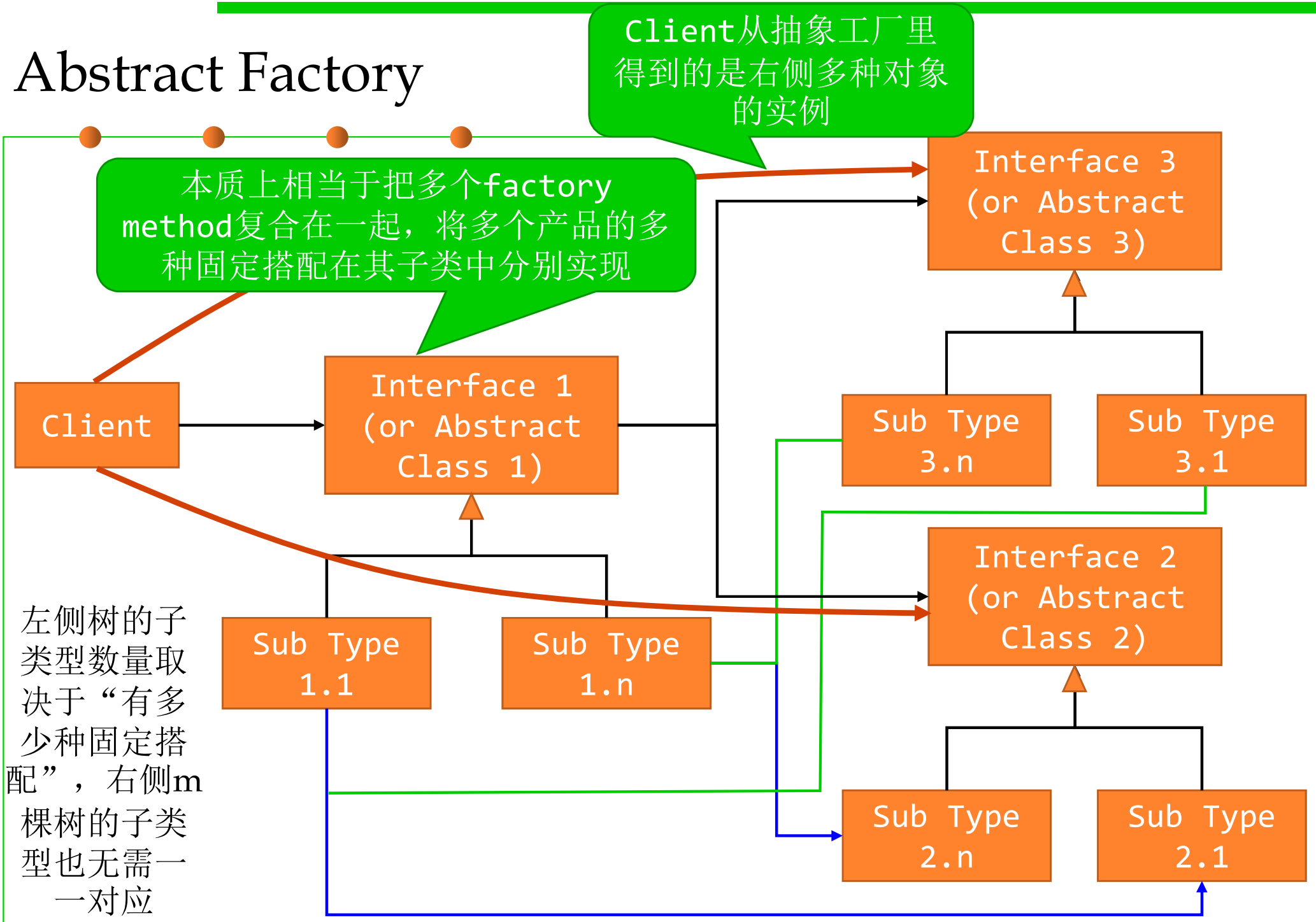
在该模式里，左右两个树里，其实分别只有一个子类型

Factory Method

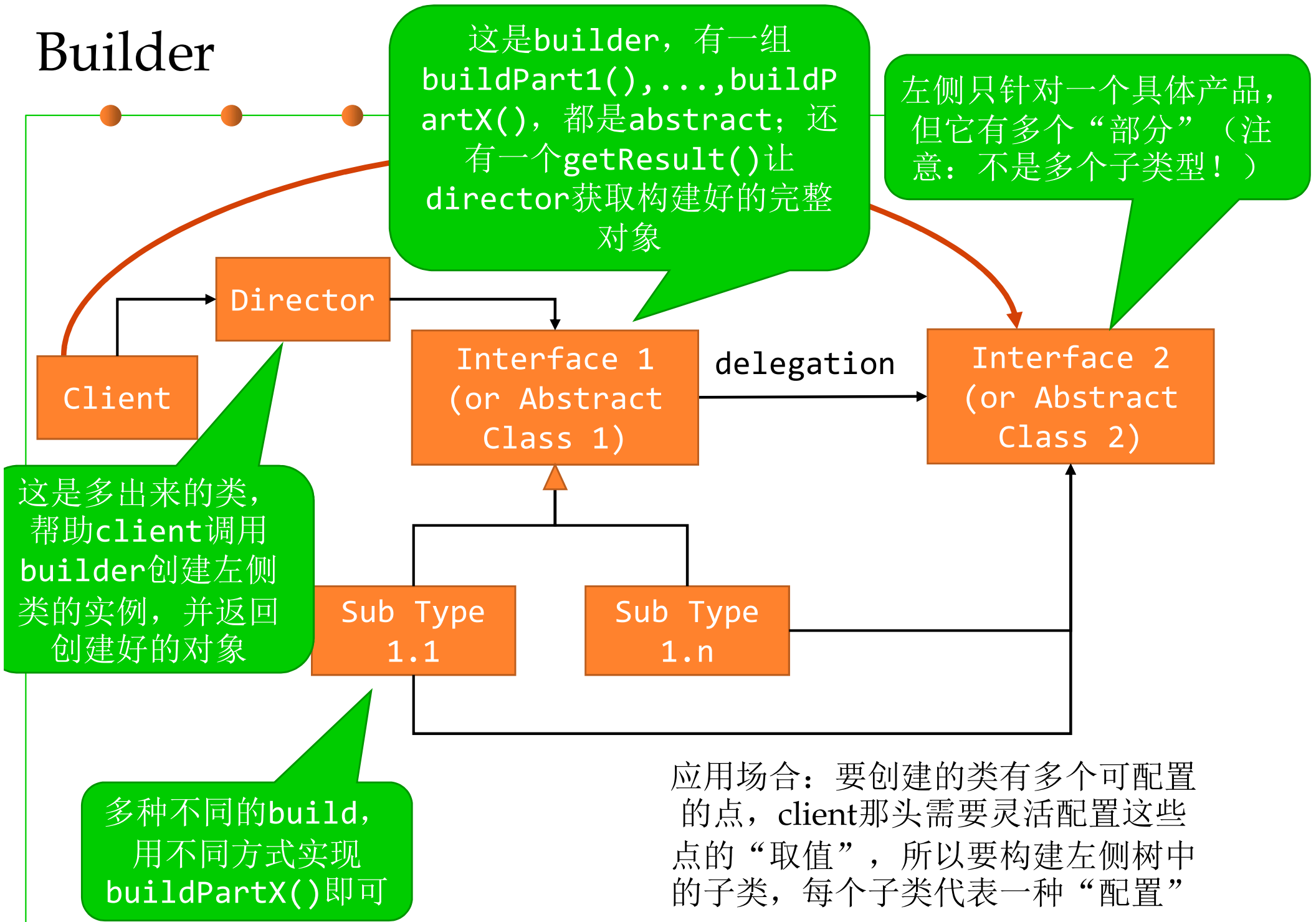


左右两棵树的子类型一一对应。如果在工厂方法里使用type表征右侧的子类型，那么左侧的子类型只要1个即可。

Abstract Factory

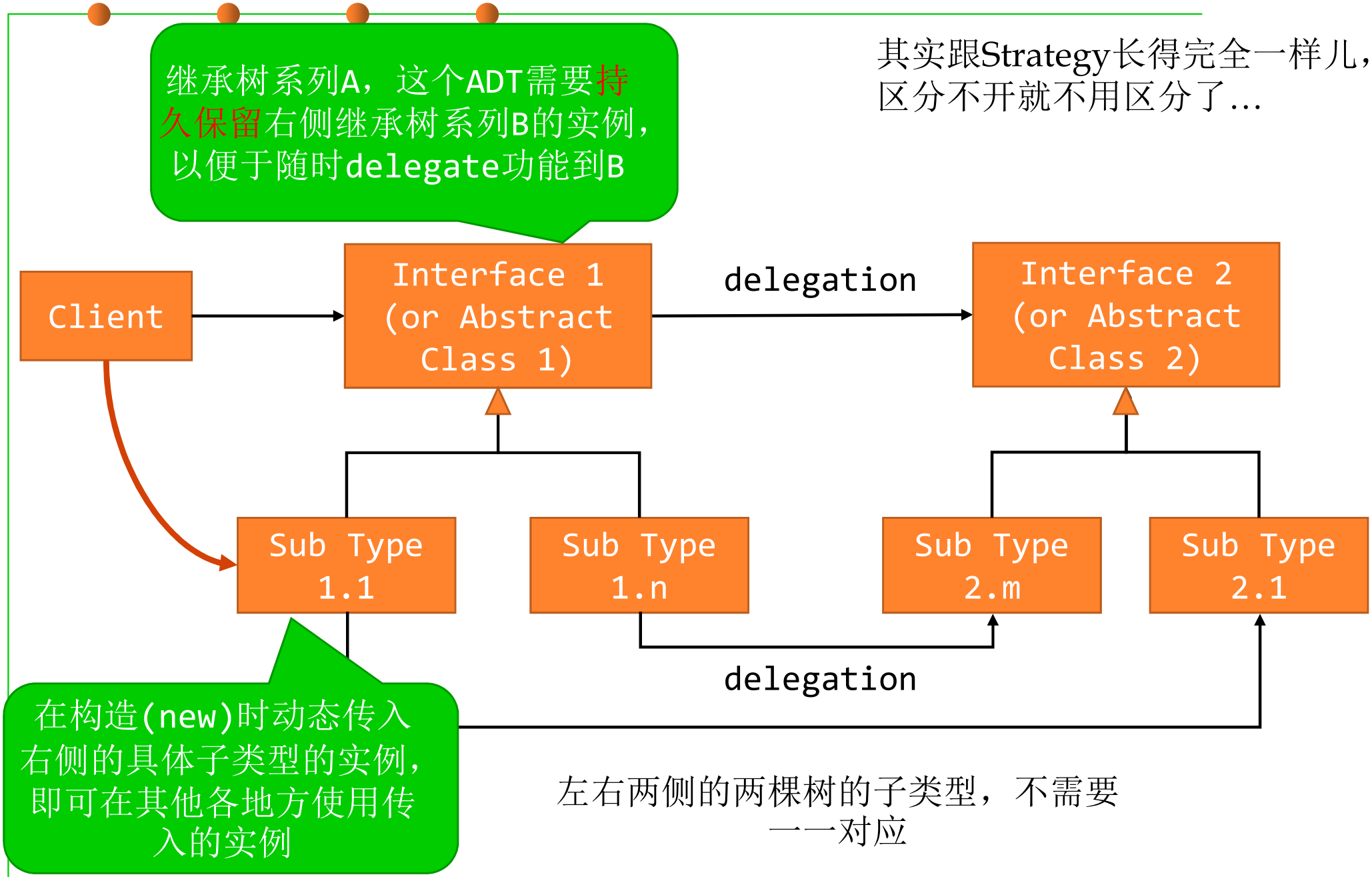


Builder

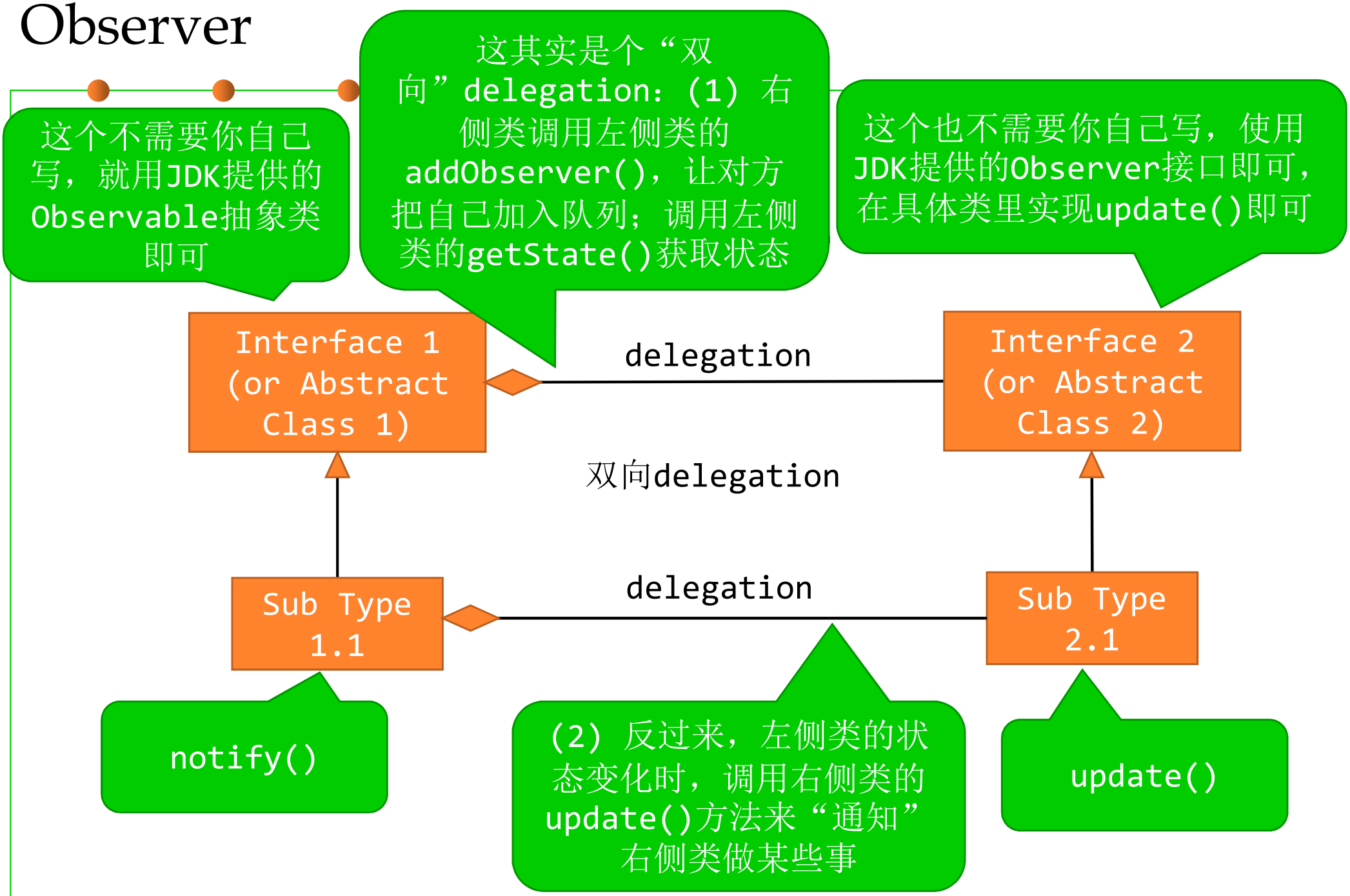


应用场合：要创建的类有多个可配置的点，client那头需要灵活配置这些点的“取值”，所以要构建左侧树中的子类，每个子类代表一种“配置”

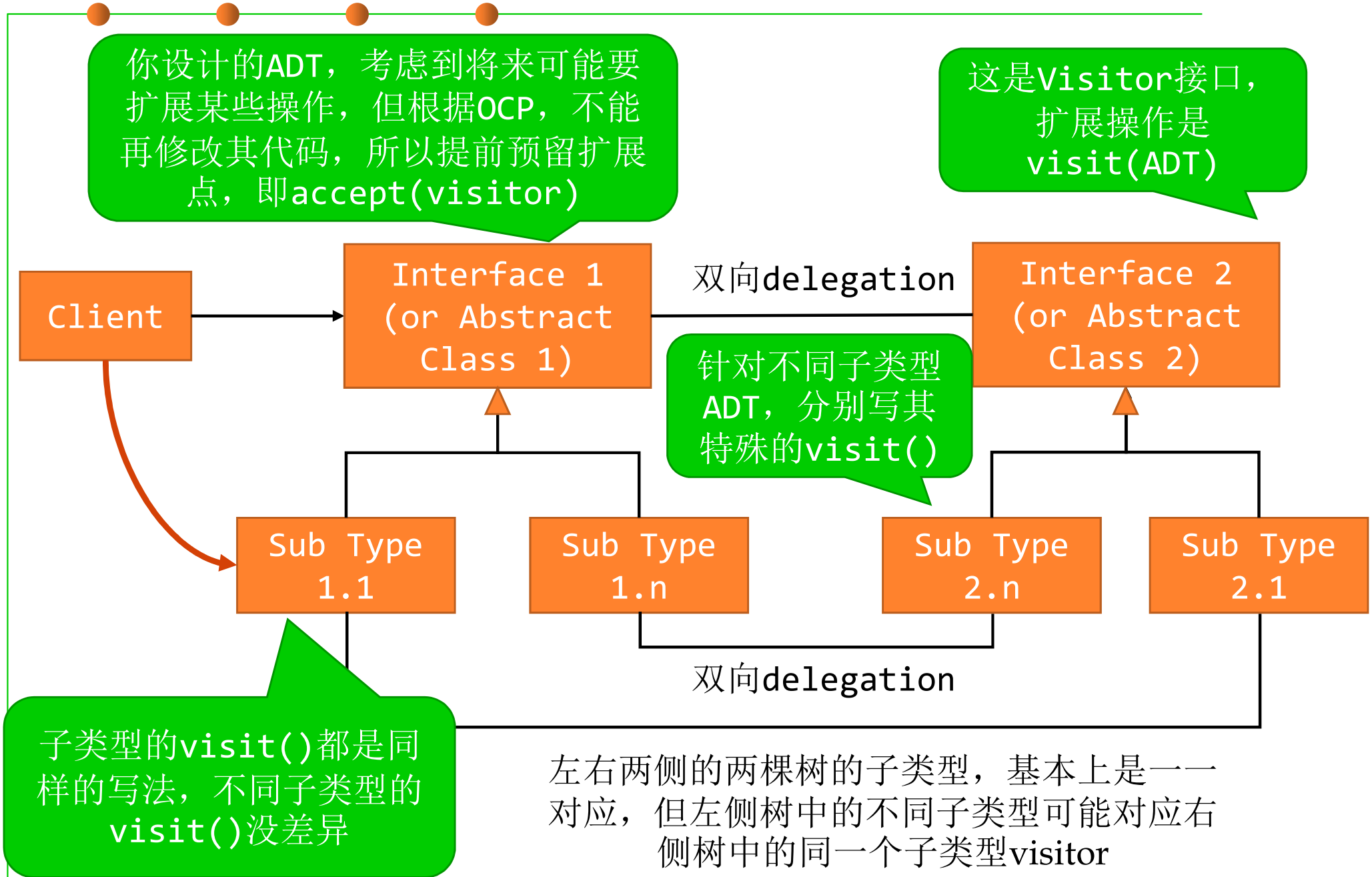
Bridge


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…
Client
Interface 1 (or Abstract Class 1)
delegation
Interface 2 (or Abstract Class 2)
Sub Type 1.1
Sub Type 1.n
Sub Type 2.m
Sub Type 2.1
delegation
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

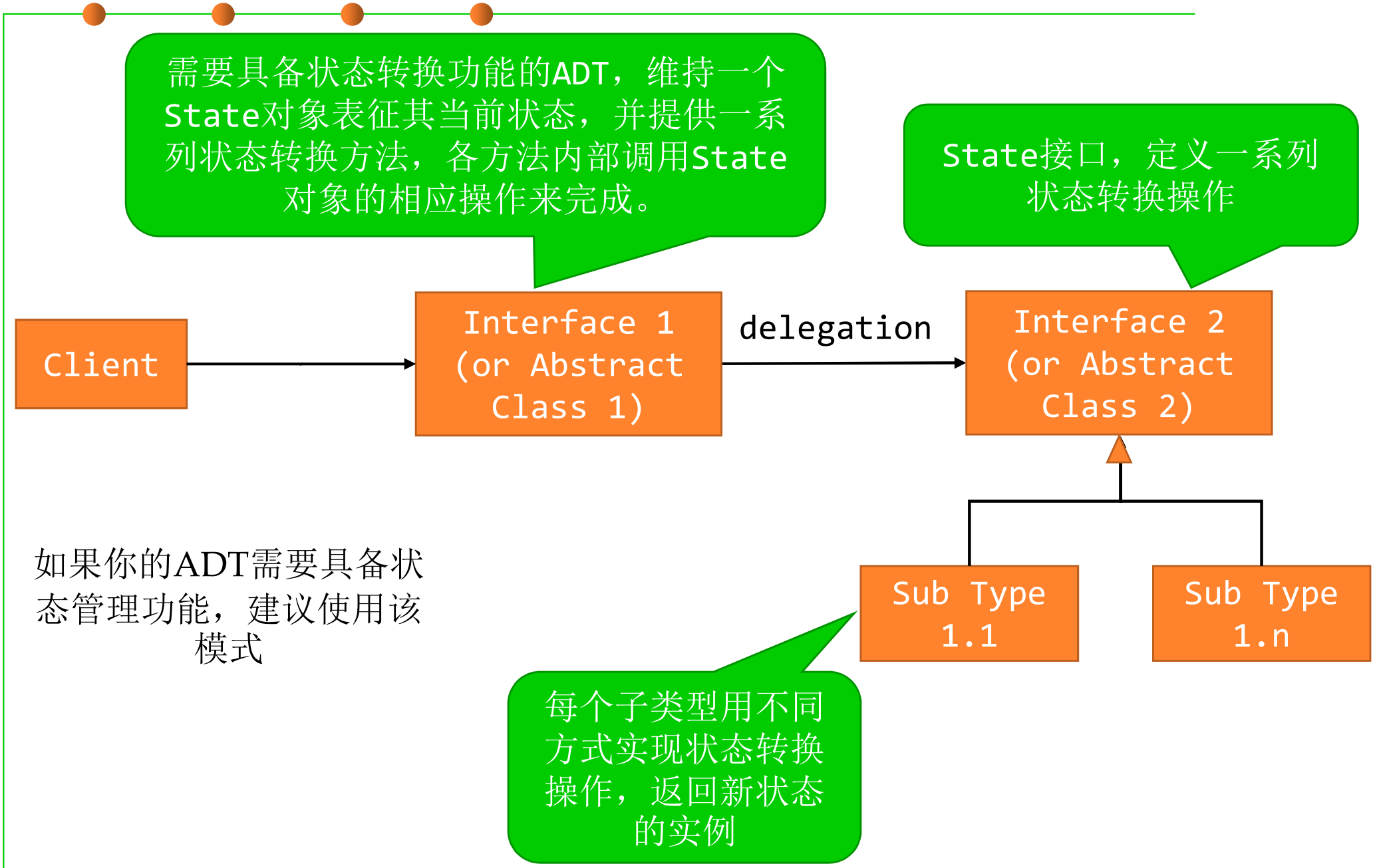
Observer


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；调用左侧类的getState()获取状态
这个也不需要你自己写，使用JDK提供的Observer接口即可，在具体类里实现update()即可
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
notify()
(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
update()

Visitor

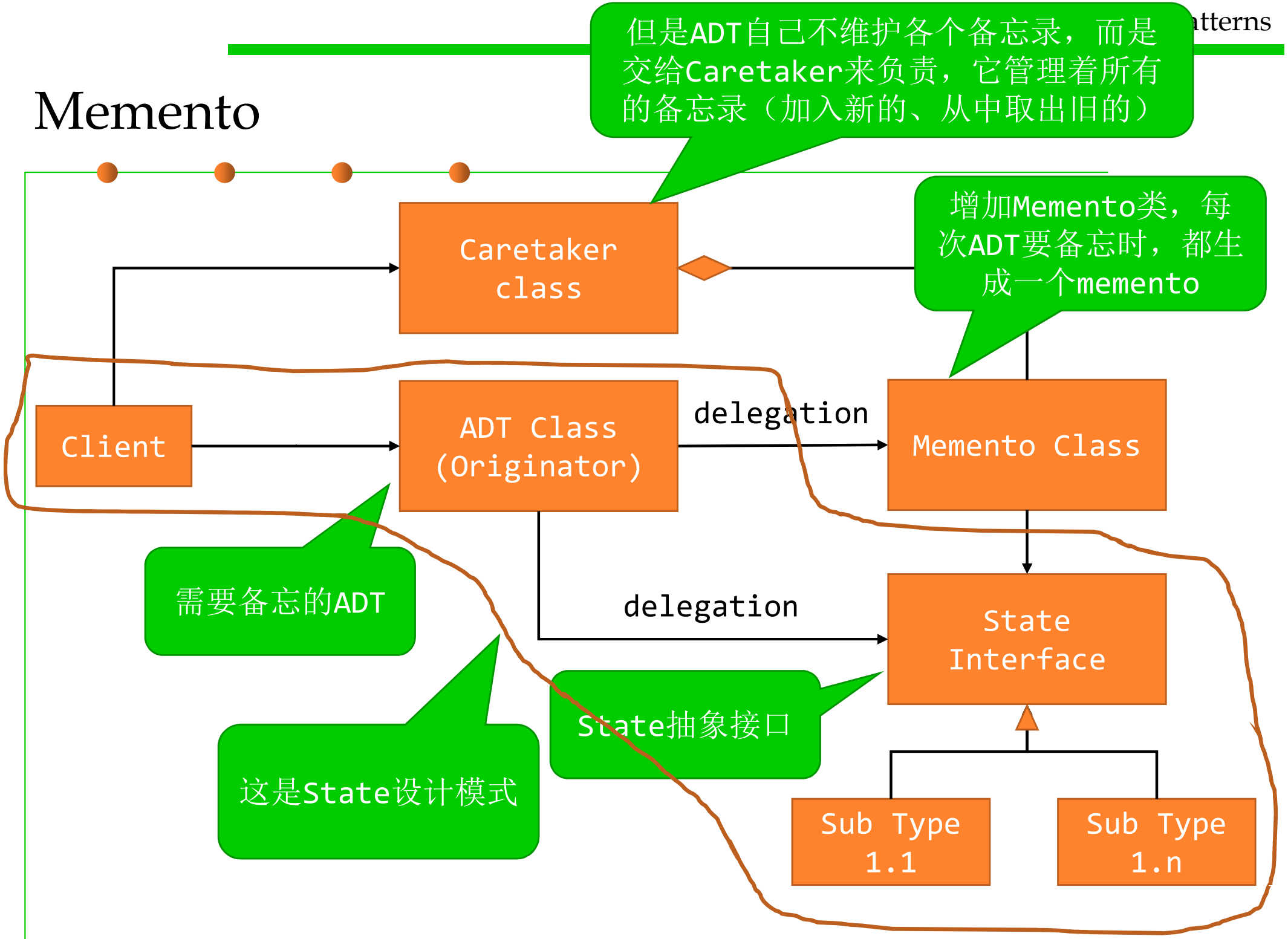

你设计的ADT，考虑到将来可能要扩展某些操作，但根据OCP，不能再修改其代码，所以提前预留扩展点，即accept(visitor)
这是Visitor接口，扩展操作是visit(ADT)
Client
Interface 1 (or Abstract Class 1)
双向delegation
Interface 2 (or Abstract Class 2)
针对不同子类型ADT，分别写其特殊的visit()
Sub Type 1.1
Sub Type 1.n
Sub Type 2.n
Sub Type 2.1
双向delegation
子类型的visit()都是同样的写法，不同子类型的visit()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



如果你的ADT需要具备状态管理功能，建议使用该模式

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）
Caretaker class
增加Memento类，每次ADT要备忘时，都生成一个memento
Client
ADT Class (Originator)
delegation
Memento Class
需要备忘的ADT
delegation
State Interface
State抽象接口
这是State设计模式
Sub Type 1.1
Sub Type 1.n

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

The end

April 17, 2019