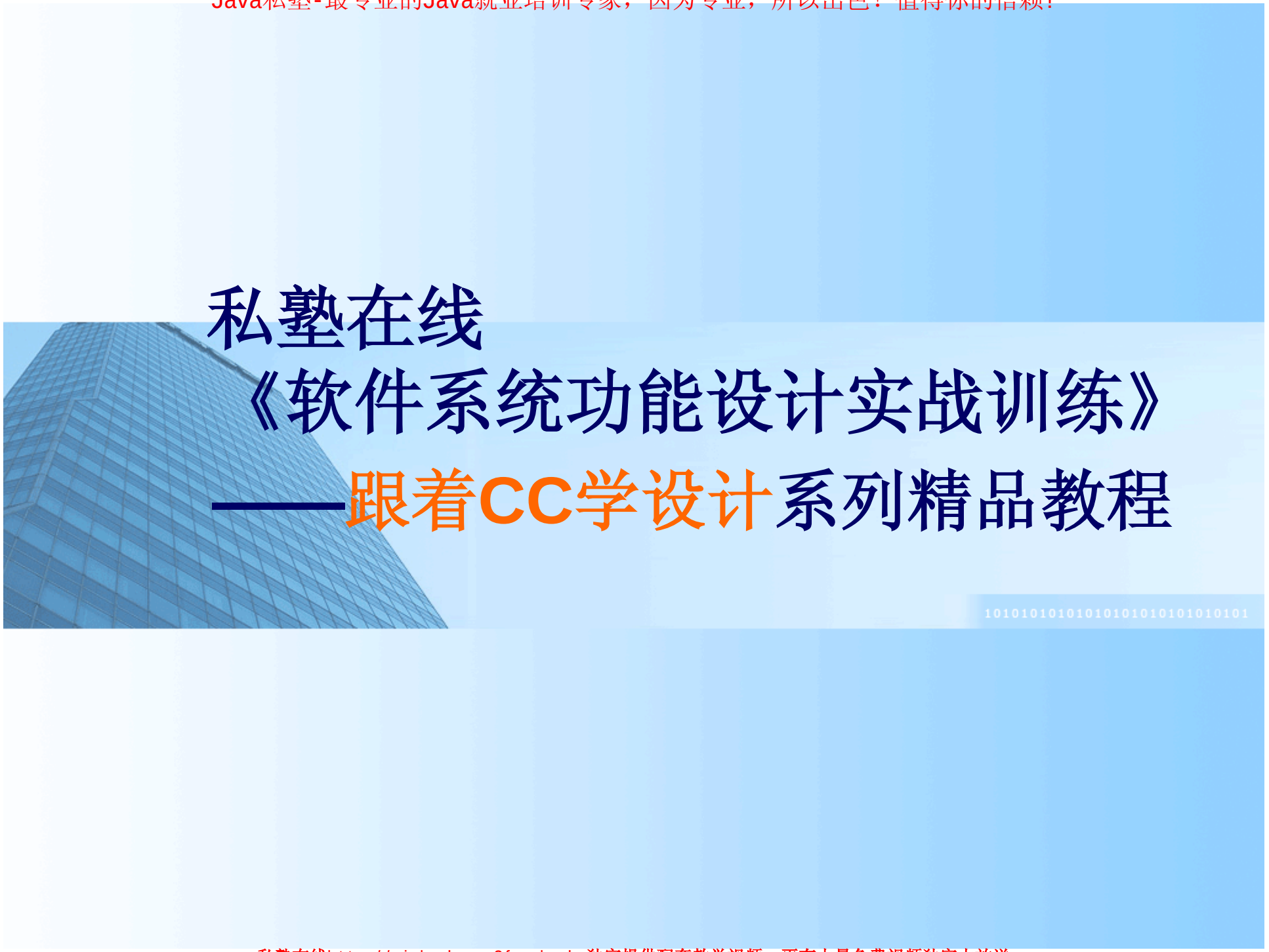


私塾在线
《软件系统功能设计实战训练》
——跟着CC学设计系列精品教程

本周设计作业

- 综合的结业测试：

  实现一个基本的通用权限系统设计

- 学习目标：

  （1）在实战中练习设计的理念和方法

  （2）学习如何实现API和SPI分离设计

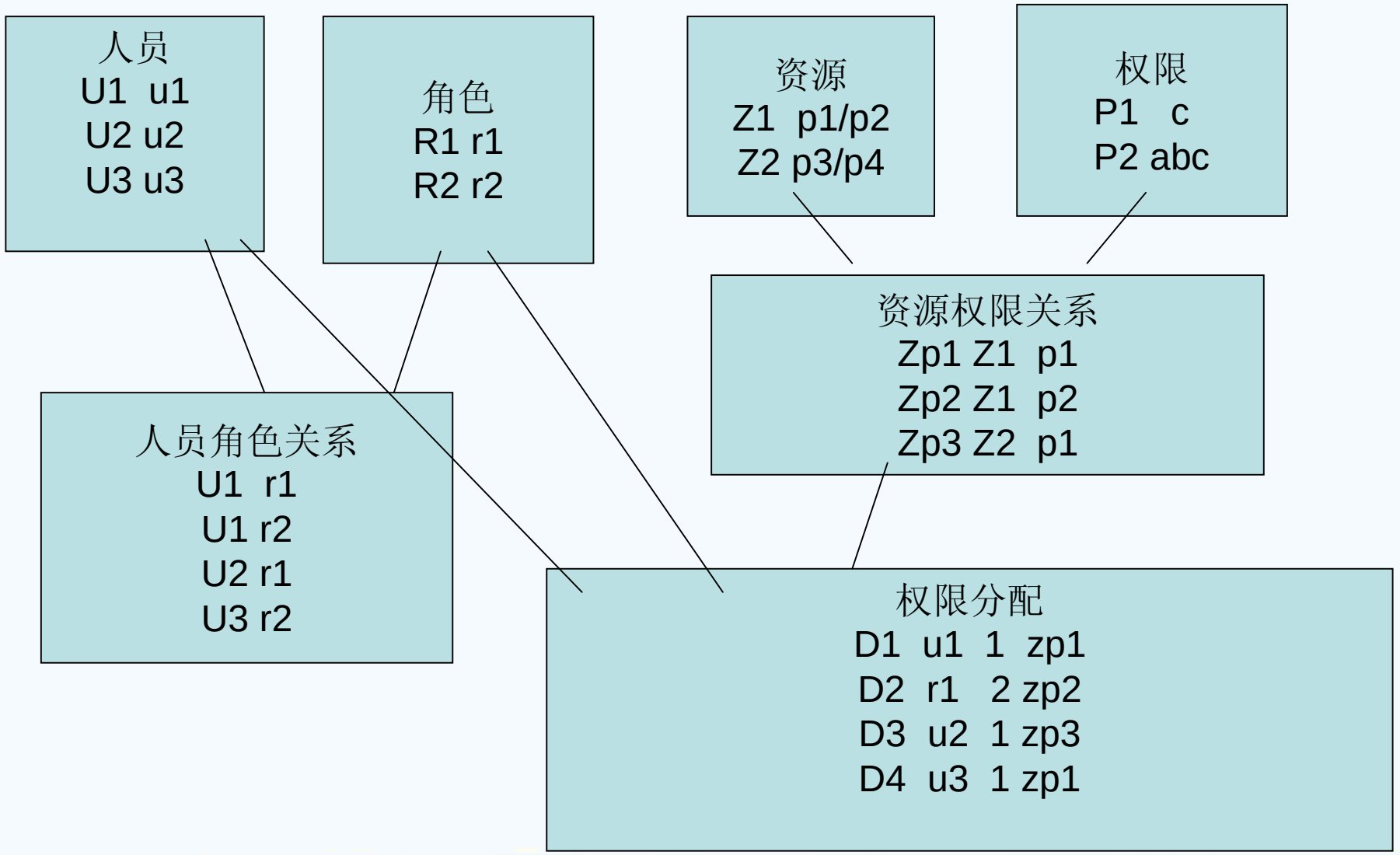
  （3）学习如何设计公共级别的接口，并提供足够的灵活性进行扩展

  （4）综合应用多种设计模式，除了前面练到的，还会有外观模式、抽象工厂模式、解释器模式、模板方法模式等的应用

  该设计方式可应用于多种有类似功能的系统，几乎所有的应用都有这样的功能……

- 分配阶段= è 就是写数据的过程
- 人员、角色、资源、权限、权限分配
- 人员角色关系、资源权限关系

- 验证阶段 è 读出数据并进行匹配的过程
- If(loginUser.hasPermit(“p1/p2”,“abc”)){
- //执行功能
- }



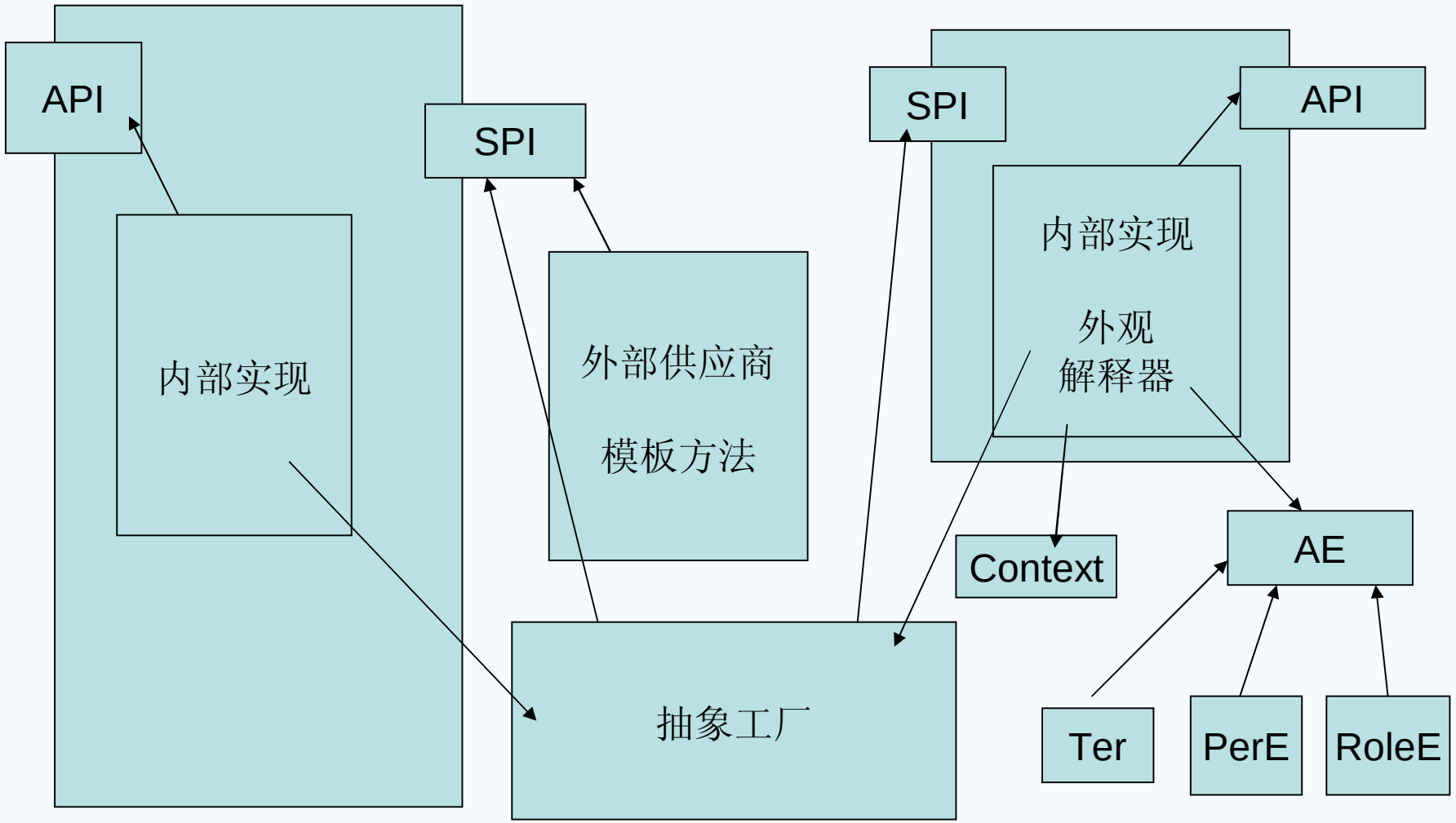
- API：应用程序接口，给客户端使用。
- 特点：功能由提供API的模块来实现，外部调用只是使用这个功能

- SPI：供应商的实现接口，给想要扩展你系统的人使用。
- 特点：功能是由其他人来实现的，但是在自己的模块里面来使用这些功能

本周设计作业

- 基本功能
  1：实现权限分配和认证的API和SPI分离设计
  2：提供权限分配和认证的实现，需要使用供应商提供的实现，引入抽象工厂模式、外观模式
  3：在SPI的实现中，引入模板方法模式、解释器模式

- 作业要求
  1：在老师给定的概要代码基础上，实现上述基本要求的功能设计
  2：重点在接口和功能实现的设计上，无须关注具体业务实现
  3：对于每个api，可以适当写点样例代码，能够调用运行更佳
  4：考虑合理的结构，职责的划分，以及设计模式的合理使用



初识抽象工厂模式

- 定义

  提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。

- 结构和说明

抽象工厂模式的知识要点

- 抽象工厂模式的知识要点

1：抽象工厂的功能是为一系列相关对象或相互依赖的对象创建一个接口，一定要注意，这个接口内的方法不是任意堆砌的，而是一系列相关或相互依赖的方法

2：抽象工厂通常实现成为interface，可以使用工厂方法来实现抽象工厂

3：抽象工厂创建的这一系列对象，可以看成是一个产品簇。这就带来非常大的灵活性，切换一个产品簇的时候，只要提供不同的抽象工厂实现就好了，也就是说现在是以产品簇做为一个整体被切换。

4：实现抽象工厂的时候，可以考虑把它实现成为可扩展的工厂

思考抽象工厂模式

- 抽象工厂模式的本质

  抽象工厂模式的本质是：**选择产品簇的实现**

- 何时选用抽象工厂模式

1：如果希望一个系统独立于它的产品的创建，组合和表示的时候，换句话说，希望一个系统只是知道产品的接口，而不关心实现的时候

2：如果一个系统要由多个产品系列中的一个来配置的时候，换句话说，就是可以动态的切换产品簇的时候

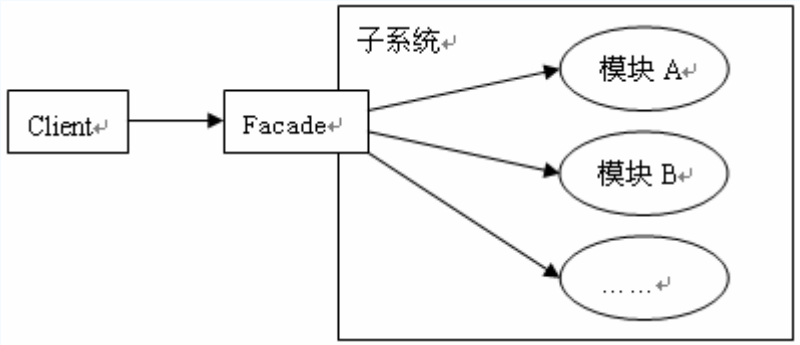
3：如果要强调一系列相关产品的接口，以便联合使用它们的时候

初识外观模式

- 定义

  为子系统中的一组接口提供一个一致的界面，Facade模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。

- 结构和说明



Facade：定义子系统的多个模块对外的高层接口，通常需要调用内部多个模块，从而把客户的请求代理给适当的子系统对象。

模块：接受Facade对象的委派，真正实现功能，各个模块之间可能有交互。
注意，Facade对象知道各个模块，但是各个模块不应该知道Facade对象。

外观模式的知识要点

- 外观模式的知识要点

1：首先要正确理解界面和接口在这里的含义。这里的界面主要指的是从一个组件外部来看这个组件，能够看到什么，这就是这个组件的界面。而这里的接口并不是指Java的interface，而是指的外部和内部交互的这么一个通道，通常是指的一些方法，可以是类的方法，也可以是interface的方法。也就是说，这里说的接口，并不等价于interface，也有可能是一个类

2：外观模式的目的不是给子系统添加新的功能接口，而是为了让外部减少与子系统内多个模块的交互，松散耦合，从而让外部能够更简单的使用子系统

3：外观是位于提供外观的模块内部的，也就是模块对外提供的外观

4：外部可以绕开Facade，直接调用某个具体模块的接口，这样就能实现兼顾组合功能和细节功能

5：外观通常实现成为类，提供缺省的功能实现，当然外观也可以实现成为interface

6：外观里面一般也不去真正实现功能，而是组合调用模块内部已有的实现，外观只是为了给客户端一个简洁的接口，并不是要外观自己来实现相应的功能

思考外观模式

- 外观模式的本质

  外观模式的本质是：**封装交互，简化调用**

- 何时选用外观模式

1：如果你希望为一个复杂的子系统提供一个简单接口的时候，可以考虑使用外观模式，使用外观对象来实现大部分客户需要的功能，从而简化客户的使用

2：如果想要让客户程序和抽象类的实现部分松散耦合，可以考虑使用外观模式，使用外观对象来将这个子系统与它的客户分离开来，从而提高子系统的独立性和可移植性

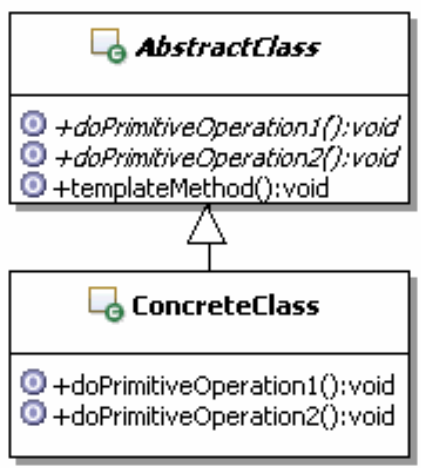
3：如果构建多层结构的系统，可以考虑使用外观模式，使用外观对象作为每层的入口，这样可以简化层间调用，也可以松散层次之间的依赖关系

初识模板方法模式

- 定义

  定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。模板方法使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。

- 结构和说明



AbstractClass：抽象类。用来定义算法骨架和原语操作，在这个类里面，还可以提供算法中通用的实现。

ConcreteClass： 具体实现类。用来实现算法骨架中的某些步骤，完成跟特定子类相关的功能。

模板方法模式的知识要点

- 模板方法模式的知识要点

1：模板方法模式的功能在于固定算法骨架，而让具体算法实现可扩展

2：模板方法模式通常实现成为抽象类，而不是接口

3：程序设计的一个很重要的思考点就是“变与不变”，模板类实现的就是不变的方法和算法的骨架，而需要变化的地方，都通过抽象方法，把具体实现延迟到子类去了，而且还通过父类的定义来约束了子类的行为，从而使系统能有更好的复用性和扩展性

4：模板方法模式体现了好莱坞法则，是一种父类来找子类的反向的控制结构

5：可以使用Java回调来实现模板方法，回调机制是通过委托的方式来组合功能，它的耦合强度要比继承低一些，这会给我们更多的灵活性。

模板方法模式的知识要点

6：要重点区分模板方法模式中的一些常见类型：

（1）模板方法：就是定义算法骨架的方法

（2）具体的操作：在模板中直接实现某些步骤的方法，通常这些步骤的实现算法是固定的，而且是不怎么变化的，因此就可以当作公共功能实现在模板里面。

（3）具体的AbstractClass操作：在模板中实现某些公共功能，可以提供给子类使用，一般不是具体的算法步骤的实现，只是一些辅助的公共功能

（4）原语操作：就是在模板中定义的抽象操作，通常是模板方法需要调用的操作，是必需的操作

（5）钩子操作：在模板中定义，并提供默认实现的操作。这些方法通常被视为可扩展的点，但不是必须的，子类可以有选择的覆盖这些方法。

（6）Factory Method：在模板方法中，如果需要得到某些对象实例的话，可以考虑通过工厂方法模式来获取，把具体的构建对象的实现延迟到子类中去

思考模板方法模式

- 模板方法模式的本质

  模板方法模式的本质是：**固定算法骨架**

- 何时选用模板方法模式

1：需要固定定义算法骨架，实现一个算法的不变的部分，并把可变的行为留给子类来实现的情况。

2：各个子类中具有公共行为，应该抽取出来，集中在一个公共类中去实现，从而避免代码重复

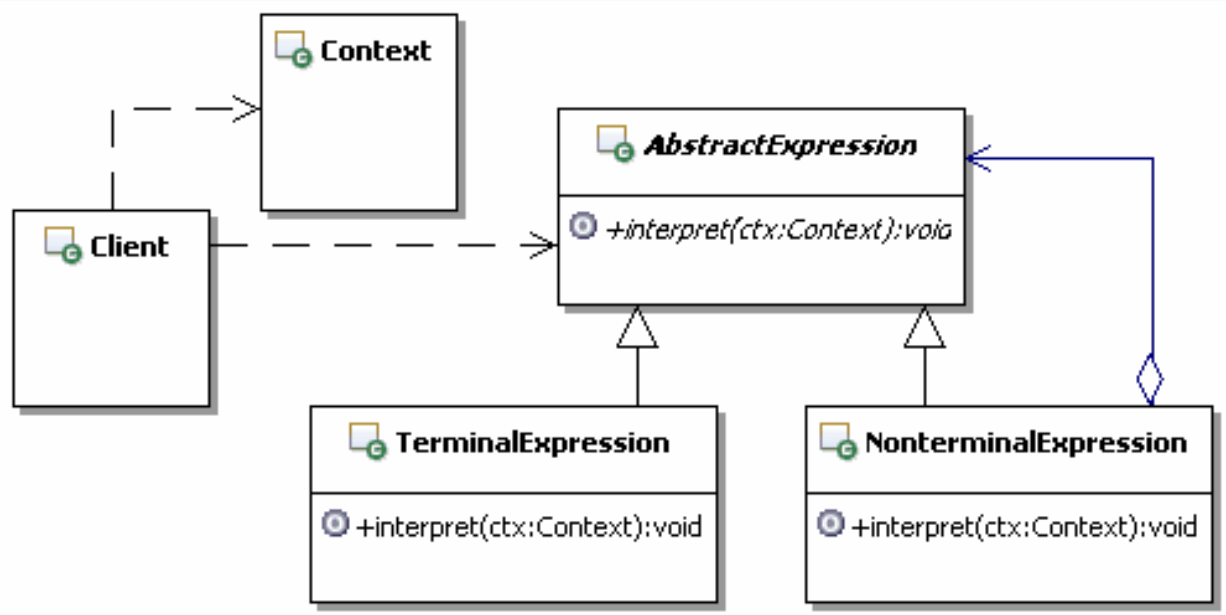
3：需要控制子类扩展的情况。模板方法模式会在特定的点来调用子类的方法，这样只允许在这些点进行扩展

初识解释器模式

- 定义

  给定一个语言，定义它的文法的一种表示，并定义一个解释器，这个解释器使用该表示来解释语言中的句子。

- 结构和说明



解释器模式的知识要点

- 解释器模式的知识要点

1：解释器模式使用解释器对象来表示和处理相应的语法规则，一般一个解释器处理一条语法规则。

2：解释器模式没有定义谁来构建抽象语法树，把这个工作交给了客户端处理

3：使用解释器模式的时候，应该先定义好相应的语法规则，并根据规则制定解释器的语法树；然后客户端在调用解释器进行解释操作的时候，需要自行构建符合语法规则要求的语法树；而解释器模式只是负责解释并执行。

4：从实质上看，解释器模式的思路仍然是分离、封装、简化，跟很多模式是一样的。

思考解释器模式

- 解释器模式的本质

  解释器模式的本质是：**分离实现，解释执行**

- 何时选用解释器模式

1：当有一个语言需要解释执行，并且可以将该语言中的句子表示为一个抽象语法树的时候，可以考虑使用解释器模式。

在使用解释器模式的时候，还有两个特点需要考虑，一个是语法相对应该比较简单，太复杂的语法不合适使用解释器模式；另一个是效率要求不是很高，对效率要求很高的情况下，不适合使用解释器模式。