System Analysis and Design

L27. GoF Design Patterns

The Gang-of-Four Design Patterns

- GoF Book-- ***Design Patterns***
- 23 Design Patterns
- About 15 are common and most useful

GoF Design Patterns

Creational

- Factory
- Singleton

Structural

- Adapter
- Composite
- Façade

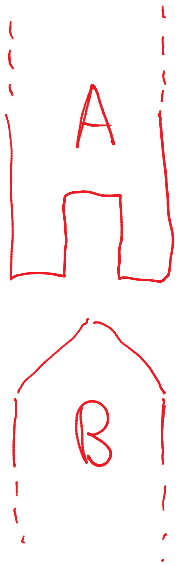
Behavioral

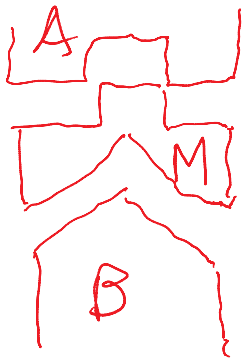
- Observer
- Strategy

适配器模式

Adapter (GoF)

- Name: Adapter 适配器模式
- **Problem:**
  How to resolve ***incompatible*** interfaces, or provide a ***stable*** interface to similar components with different interfaces?
- **Solution:**
  Convert the original interface of a component into another interface, through **an intermediate adapter object**.



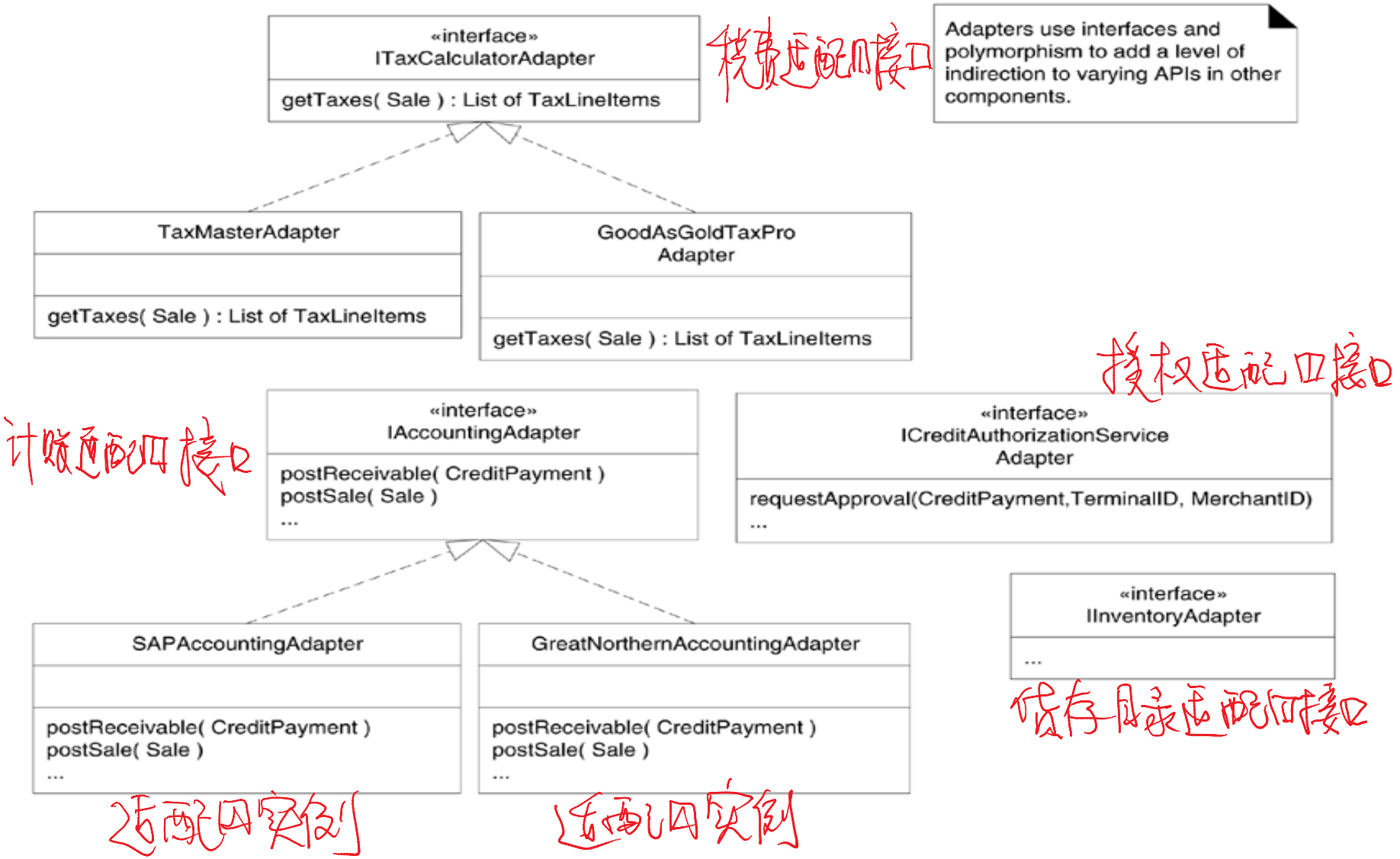


例子：

NextGen POS System

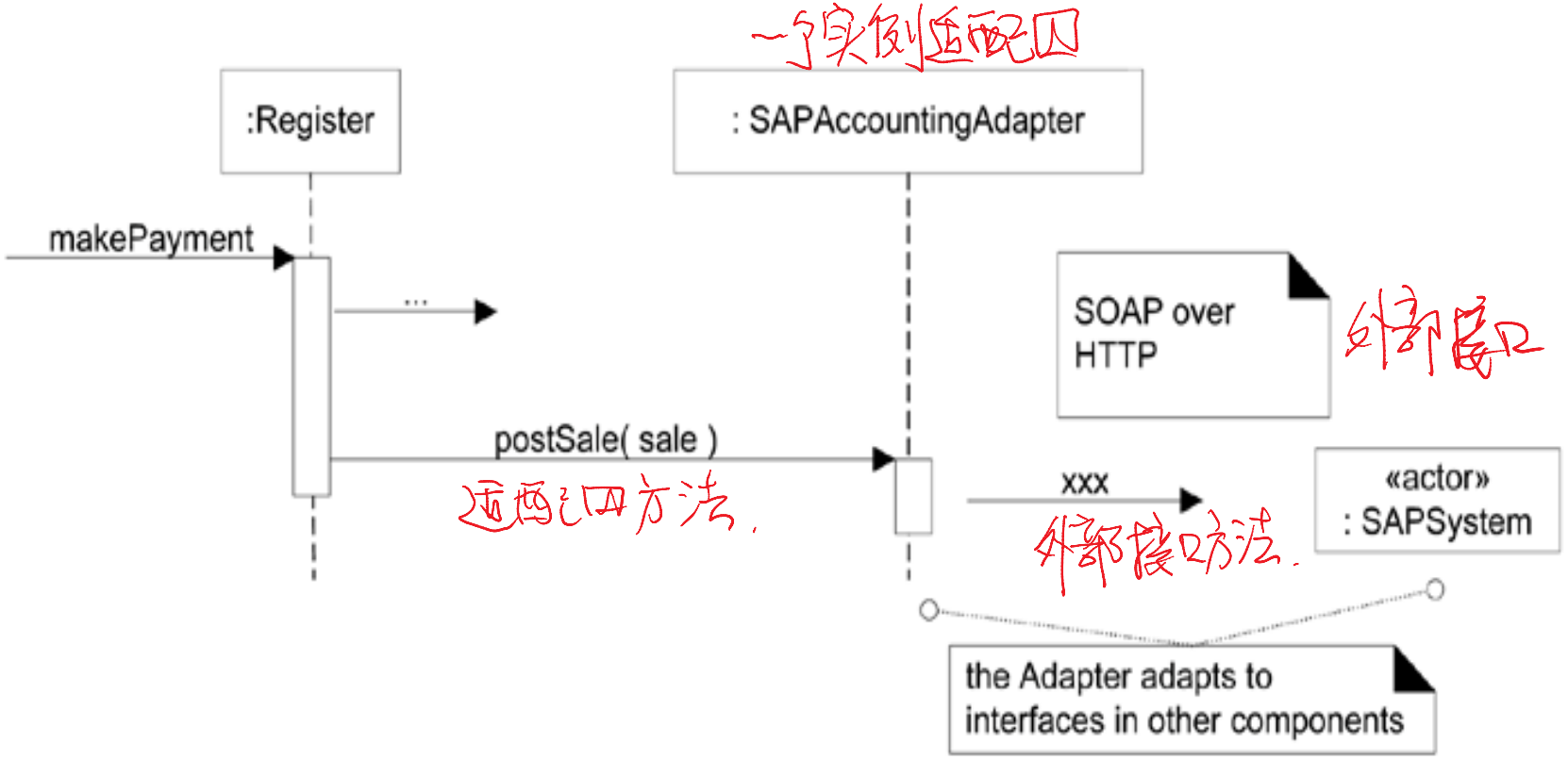
- The NextGen POS system needs to support several kinds of external third-party services, including
  - tax calculators, 税费计算
  - credit authorization services, 信用卡授权
  - inventory systems, 存货目录
  - and accounting systems, among others. 计账系统等
- Each has a different API, which can't be changed.
- A solution is to add a level of *indirection* with objects that adapt the varying external interfaces to a consistent interface used within the application.

Adapters for POS



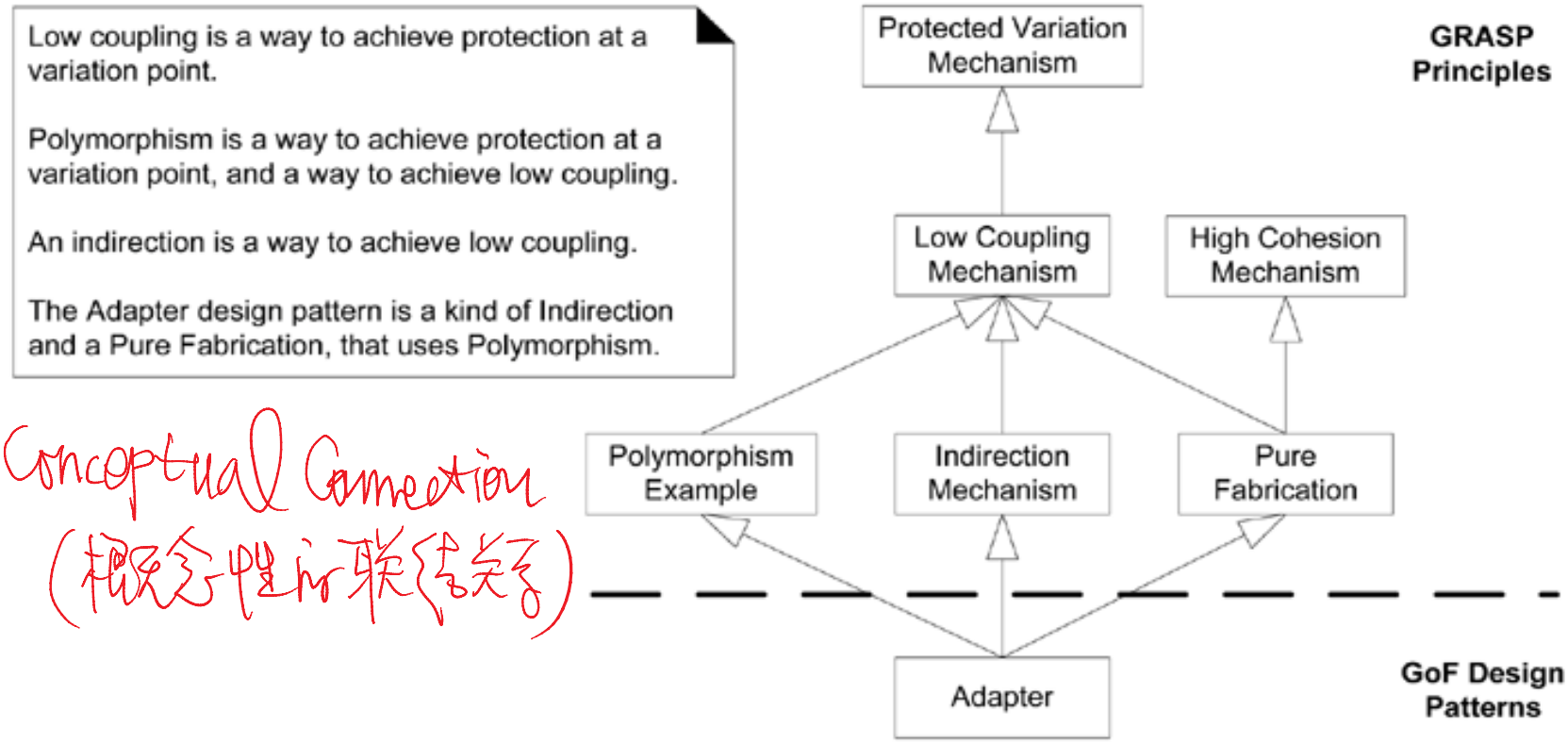
Using an Adapter

- A particular adapter instance will be instantiated for the chosen external service, such as SAP for accounting
- Adapt the *postSale* request to the external interface (SOAP interface over HTTPS offered by SAP)



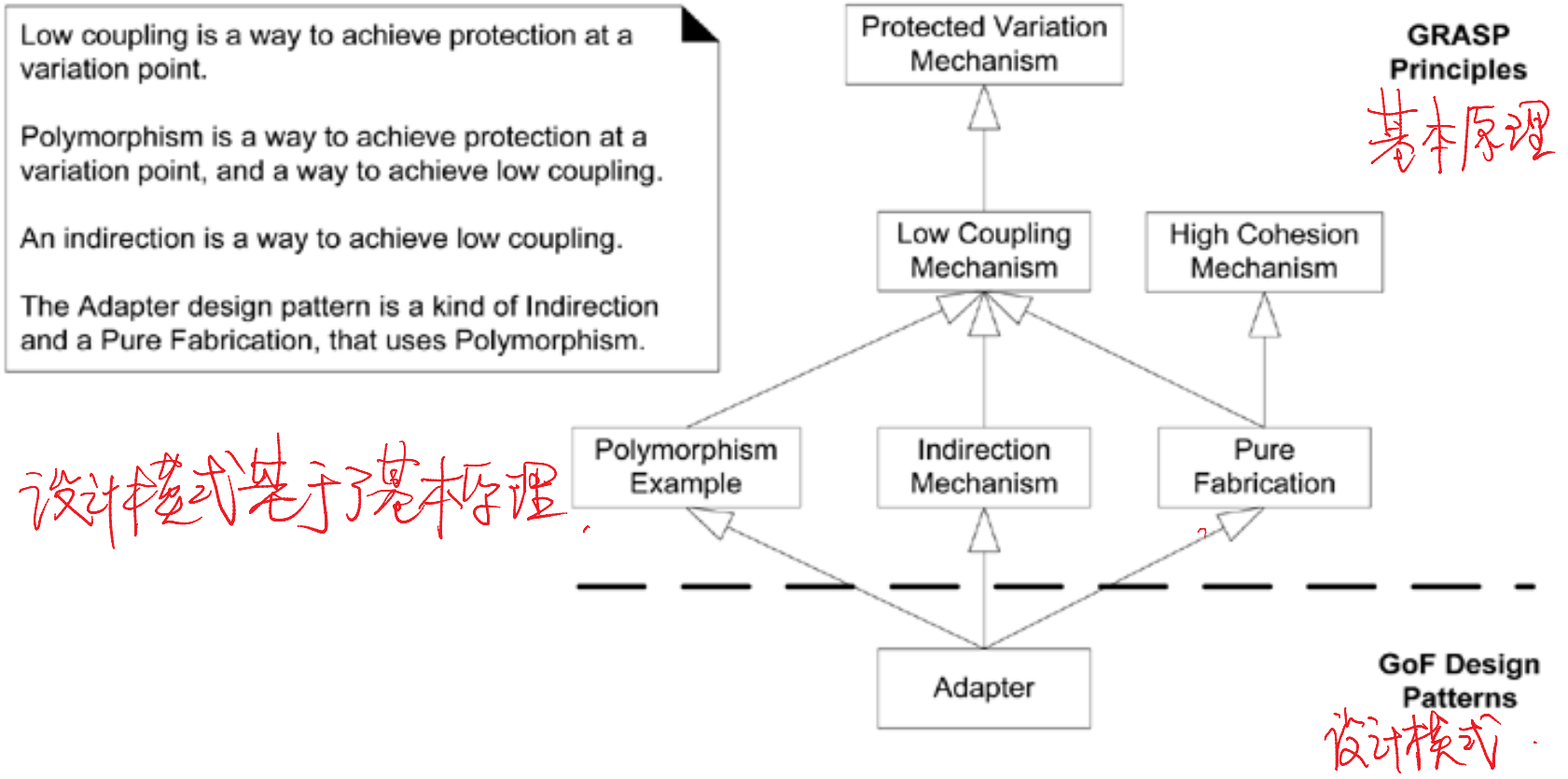
Adapter Pattern and GRASP

- Adapter supports *Protected Variations* with respect to changing external interfaces or third-party packages through the use of an *Indirection* object that applies interfaces and *Polymorphism*.



Underlying Principles

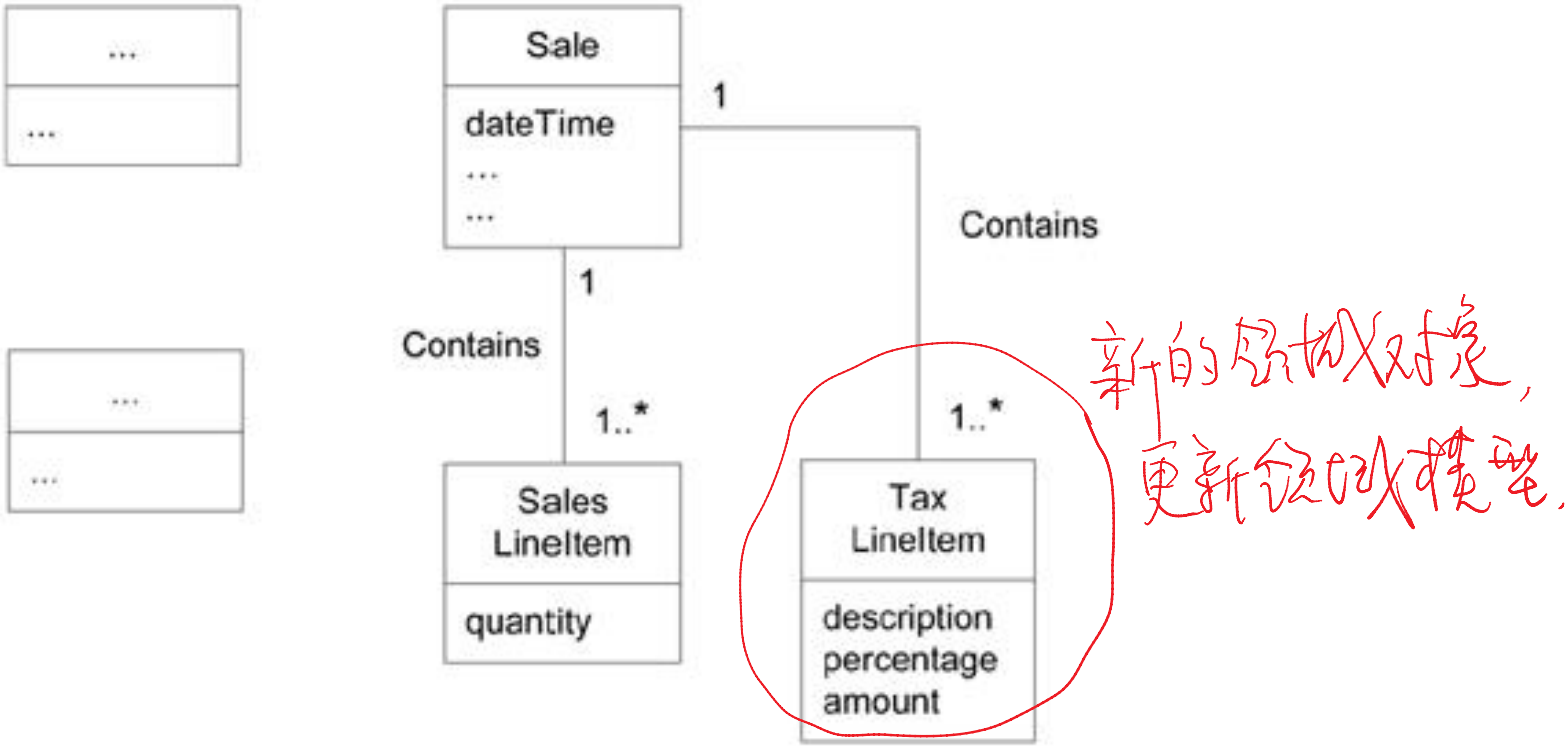
- Most design patterns can be seen as specializations of a few basic GRASP principles.
- Underlying themes are more important in design 基本原理更为重要.
  - Protected Variations, Polymorphism, Indirection, …



在设计中发现新的需求或领域概念，

Discoveries During Design

- On deeper investigation of how taxes are handled, we realized that a list of tax line items are associated with a sale, such as state tax, federal tax, and so forth.
- This is a domain concept.
- Domain Model may need be updated if necessary.



新的领域对象，
更新领域模型。

设计中一个基本的问题：

Who Creates the Adapters?

- In the prior Adapter pattern solution, who creates the adapters? And how to determine which class of adapter to create?
- If domain object creates them, the responsibilities of the domain object are going 领域对象创建Adapter
  - beyond application logic (such as sales total calculations) and
  - into other concerns related to connectivity with external components.
- Therefore, choosing a domain object (such as a *Register*) to create the adapters does not support the goal of a separation of concerns, and lowers its cohesion. 这样低内聚！

让我们考虑另一个新的模式

Factory Pattern

- A common alternative in this case is to apply the **Factory** pattern, in which a *Pure Fabrication* "factory" object is defined to create objects.
- Factory objects have several advantages:
  - Separate the responsibility of complex creation into cohesive helper objects. 用单独的类分离了职责
  - Hide potentially complex creation logic. 隐藏了复杂的创建逻辑
  - Allow introduction of performance-enhancing memory management strategies, such as object caching or recycling. 可以引入存储管理机制

Factory Pattern

- Also called *Simple Factory* or *Concrete Factory*. This pattern is not a GoF design pattern.
- It is a simplification of the GoF *Abstract Factory* pattern

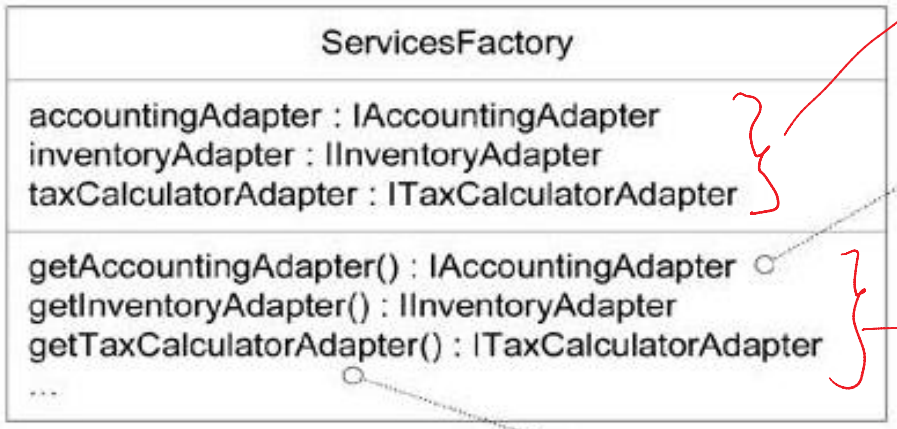
- **Name:** Factory
- **Problem:** Who should be responsible for creating objects when there are special considerations, such as complex creation logic, a desire to ***separate*** the creation responsibilities for better cohesion, and so forth? 复杂性不创建对象的职责如何分配?
- **Solution:** Create a Pure Fabrication object called a Factory that handles the creation. 用虚构件.

Factory Example

外部服务的创建(前面例中)

- In the *ServicesFactory*, the logic to decide which class to create is resolved by reading in the class name from an external source and then dynamically loading the class. 用动态的方法装入服务对象的创建.
- This is an example of a partial **data-driven design**. 近似于数据驱动设计
- This design achieves *Protected Variations* with respect to changes in the implementation class of the adapter. 设计使用了"保护的变化"原则.
- Without changing the source code in this factory class, we can create instances of new adapter classes by changing the property value Factor的代码不变,依据性质值创建
  - If Java, ensuring that the new class is visible in the Java class path for loading.

Services Factory



Three adapters

note that the factory methods return objects typed to an interface rather than a class, so that the factory can return any implementation of the interface

Three factory method.

```
if ( taxCalculatorAdapter == null )
{
  // a reflective or data-driven approach to finding the right class: read it from an
  // external property

  String className = System.getProperty( "taxcalculator.class.name" );
  taxCalculatorAdapter = (ITaxCalculatorAdapter) Class.forName( className ).newInstance();

}
return taxCalculatorAdapter;
```

读入类名 (calculator)

Create

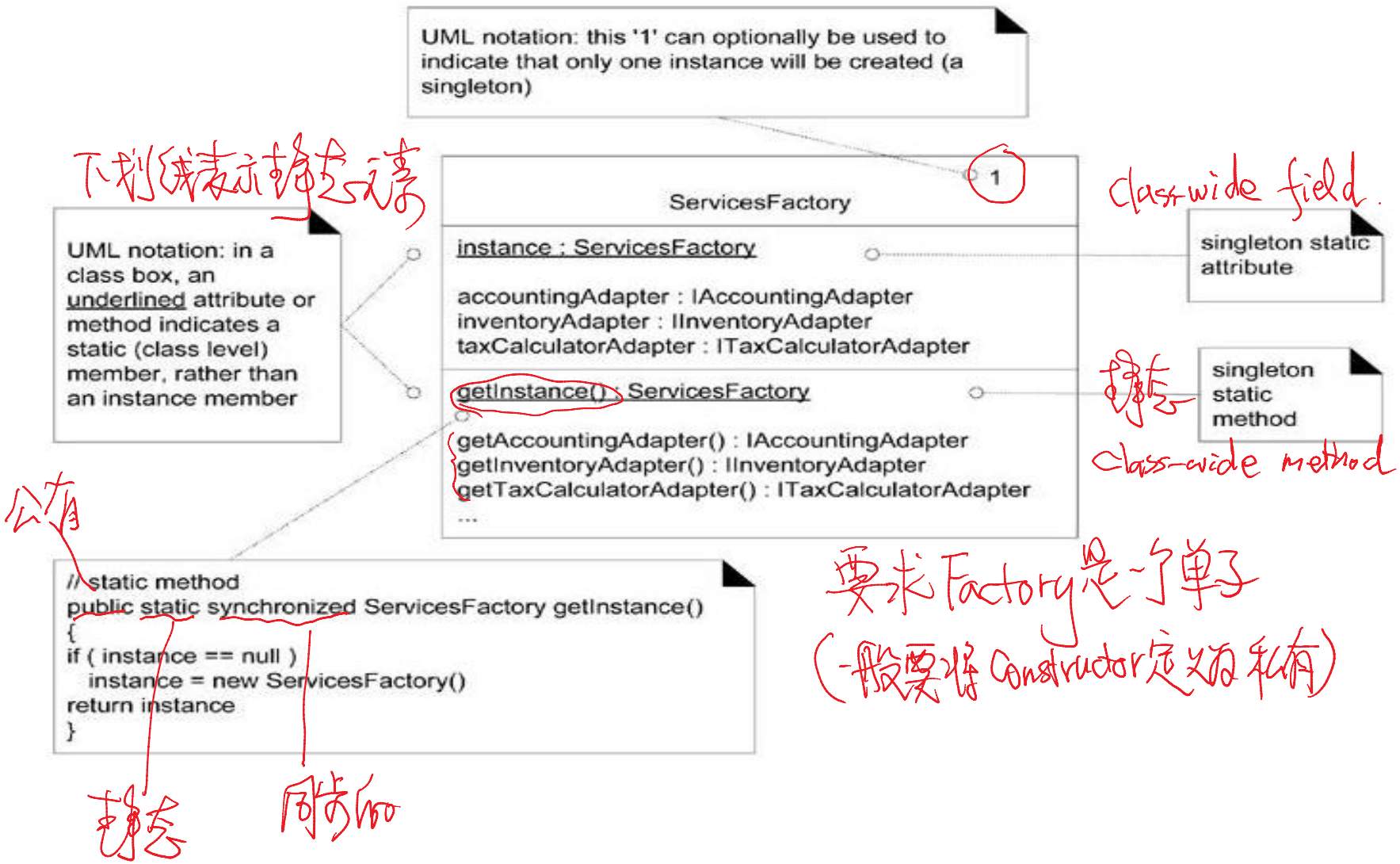
关于设计模式：

Singleton (GoF)

单子(前面提到过)

- It is desirable to support global visibility or a single access point to a single instance of a class
- **Name:** Singleton
- **Problem:**
  Exactly one instance of a class is allowed, it is a "singleton." Objects need a **global** and single point of access.
  只能创建一个实例.
- **Solution:**
  Define a static method of the class that returns the singleton.
  用静态方法返回一个实例.
- **Related Patterns:** The Singleton pattern is often used for Factory objects and Facade objects, another GoF pattern that will be discussed.

The Singleton pattern in the ServicesFactory class



The Singleton pattern in the ServicesFactory class

```
public class Register
{
public void initialize()
{
   … do some work …
   // accessing the singleton Factory via
   the getInstance call
   accountingAdapter =
ServicesFactory.getInstance().getAcco
untingAdapter();

   … do some work …
   }
    // other methods…
} // end of class
```

```
public class ServicesFactory
{
        //…
        private static ServicesFactory instance;
        public static synchronized ServicesFactory
            getInstance()
        {
          if ( instance == null )
          {
            // critical section if multithreaded
            application
            instance = new ServicesFactory();
          }
          return instance;
        }
        // other methods...
}
```

The Singleton pattern in the ServicesFactory class

```
public class Register
{
public void initialize()
{
   … do some work …
   // accessing the singleton Factory via
   the getInstance call
   accountingAdapter =
ServicesFactory.getInstance().getAcco
untingAdapter();

   … do some work…
   }
    // other methods…
} // end of class
```

```
public class ServicesFactory
{
      // eager initialization
      private static ServicesFactory instance =
        new ServicesFactory();
      public static ServicesFactory getInstance()
      {
        return instance;
      }
      // other methods...
}
```

We Prefer Lazy Initialization

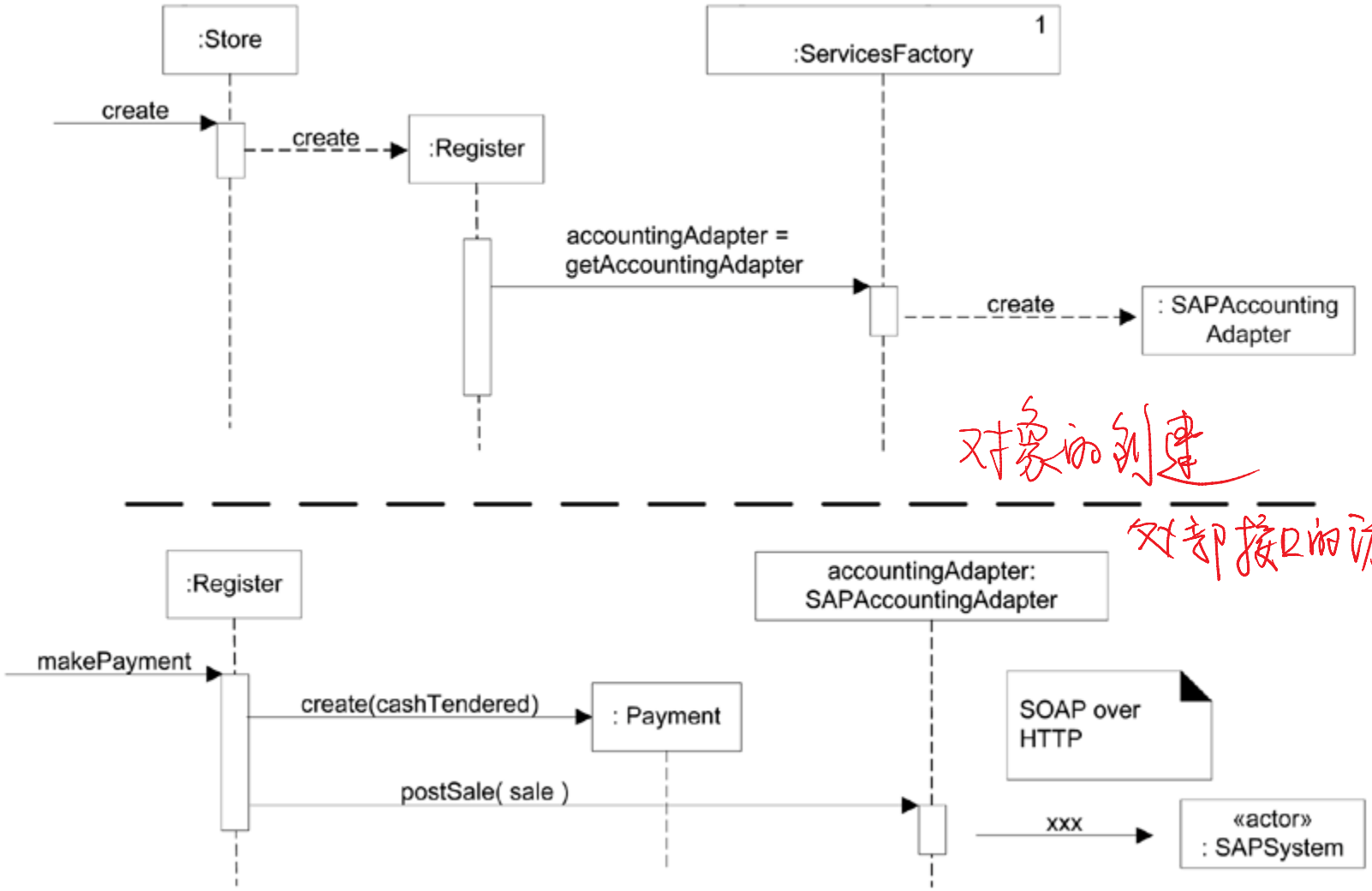
我们期望延迟实例化方法.

- Creation work (and perhaps holding on to "expensive" resources) is avoided, if the instance is never actually accessed. 如果没有用到，则不需要创建(节省资源)
- The *getInstance* lazy initialization sometimes contains complex and conditional creation logic

可以容易地实现更复杂的创建逻辑.

Conclusion of the External Services with Varying Interfaces Problem

use case realizations 表示了对于外部变化接口的整合设计



Conclusion of the External Services with Varying Interfaces Problem

- A combination of *Adapter*, *Factory*, and *Singleton* patterns have been used to provide *Protected Variations* from the varying interfaces of external tax calculators, accounting systems, and so forth.

整合的设计用了多个模式，实现了保护的变化原则

- It is important to see how the design arose from reasoning based on Controller, Creator, Protected Variations, Low Coupling, High Cohesion, Indirection, Polymorphism, Adapter, Factory, and Singleton.

重要的是要了解这些模式和原则如何推导出我们的设计.

Strategy (GoF)

- Name: Strategy
- **Problem:**
  How to design for varying, but related, algorithms or policies? How to design for the ability to change these algorithms or policies?
- **Solution:**
  Define each algorithm/policy/strategy in a separate class, with a common interface.

- Related Patterns: Strategy is based on Polymorphism, and provides Protected Variations with respect to changing algorithms. Strategies are often created by a Factory.

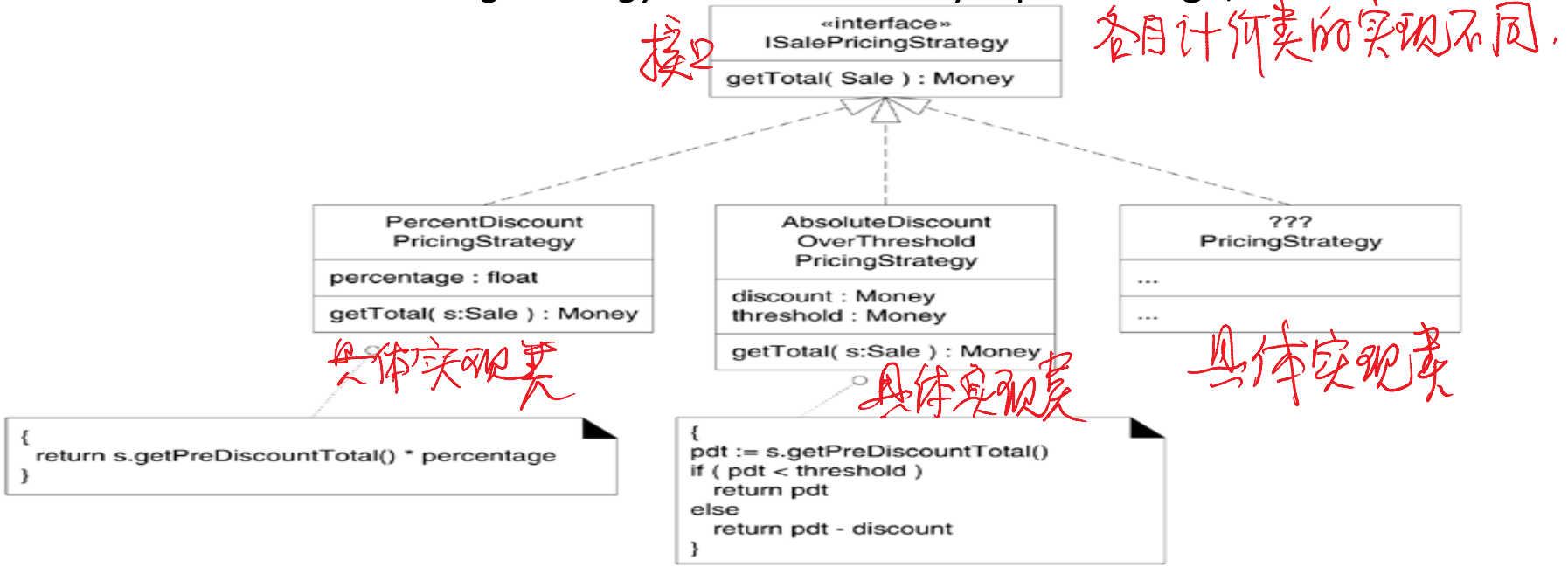
不同的记价策略

Pricing Strategy in NextGen POS

- The pricing strategy (which may also be called a rule, policy, or algorithm) for a sale can vary.
- During one period it may be 10% off all sales, later it may be $10 off if the sale total is greater than $200, and myriad other variations. 各种记价方式
- How do we design for these varying pricing algorithms? 如何设计统一的计价操作？

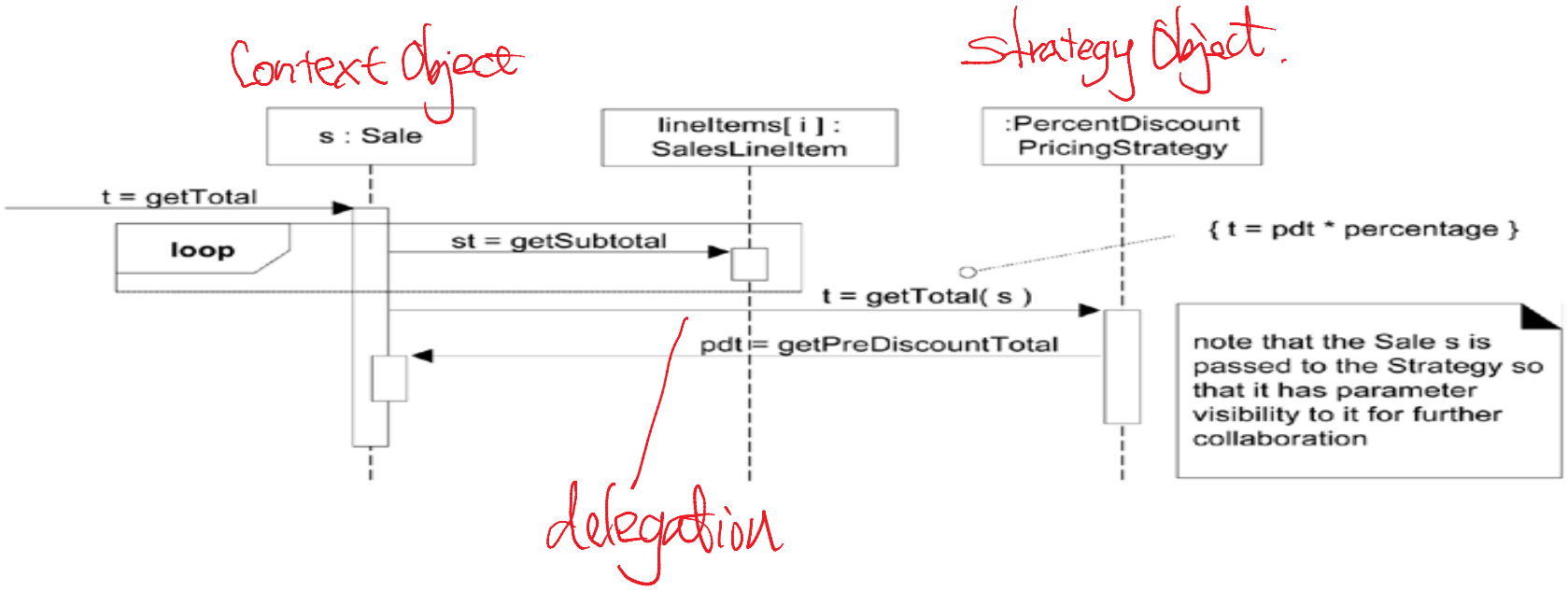
Pricing Strategy Classes

- Since the behavior of pricing varies by the strategy (or algorithm), we create multiple *SalePricingStrategy* classes, each with a polymorphic *getTotal* method.
- Each *getTotal* method takes the *Sale* object as a parameter, so that the pricing strategy object can find the pre-discount price from the *Sale*, and then apply the discounting rule.
- The implementation of each *getTotal* method will be different: *PercentDiscountPricingStrategy* will discount by a percentage, and so on.



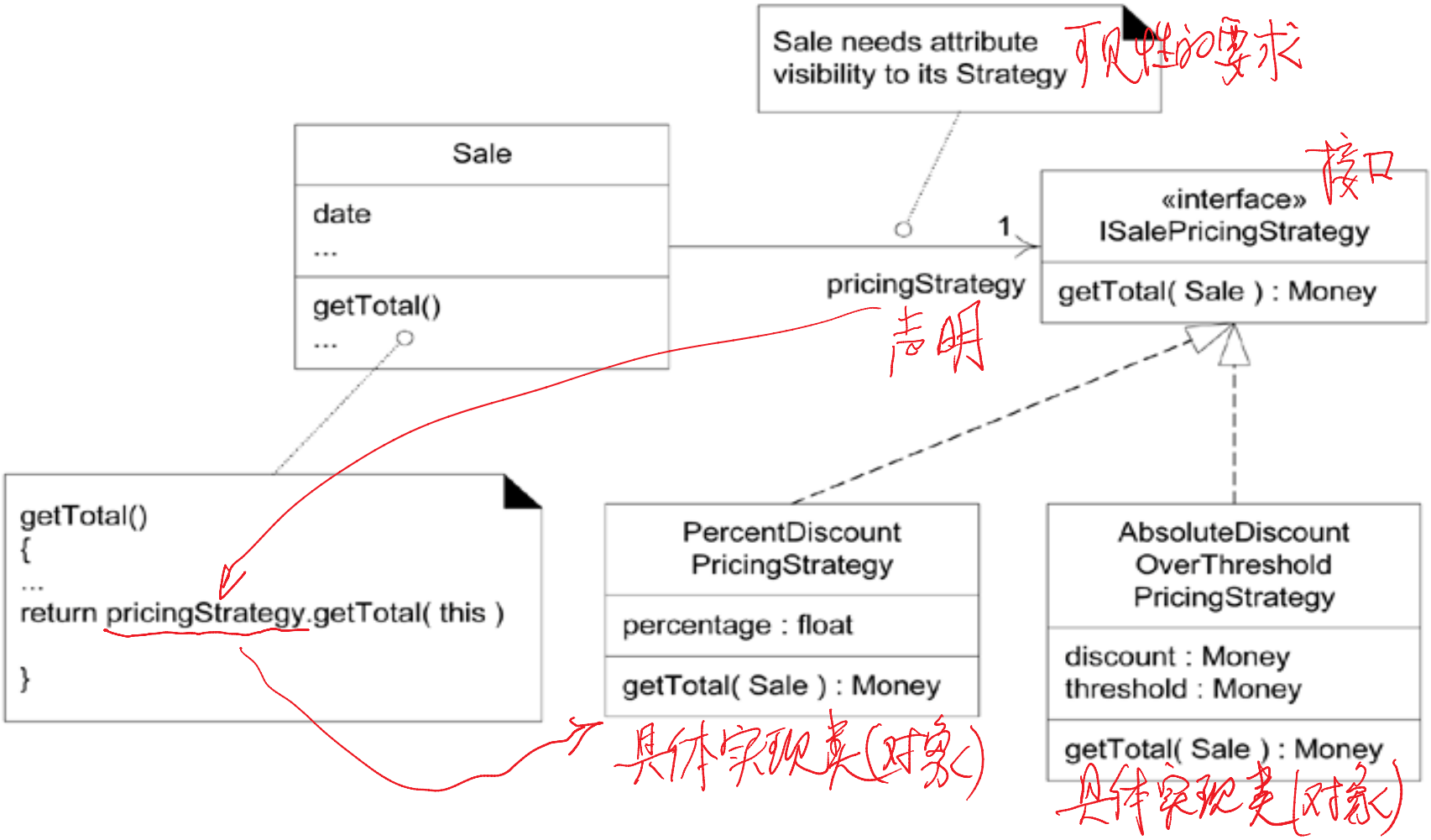
Strategy in Collaboration

- A strategy object is attached to a **context object**, the object to which it applies the algorithm.
- In this example, the context object is a *Sale*.
- When a *getTotal* message is sent to a *Sale*, it delegates some of the work to its strategy object,



Context object needs attribute visibility to its strategy

- Observe that the context object (*Sale*) needs attribute visibility to its strategy.
- This is reflected in the following DCD



Factory for Strategies

- There are different pricing algorithms or strategies, and they change over time.
- Who should create the strategy? 谁创建这些策略?
- A straightforward approach is to apply the Factory pattern again: 用工厂模式
  - A *PricingStrategyFactory* can be responsible for creating all strategies (all the pluggable or changing algorithms or policies).

策略工厂

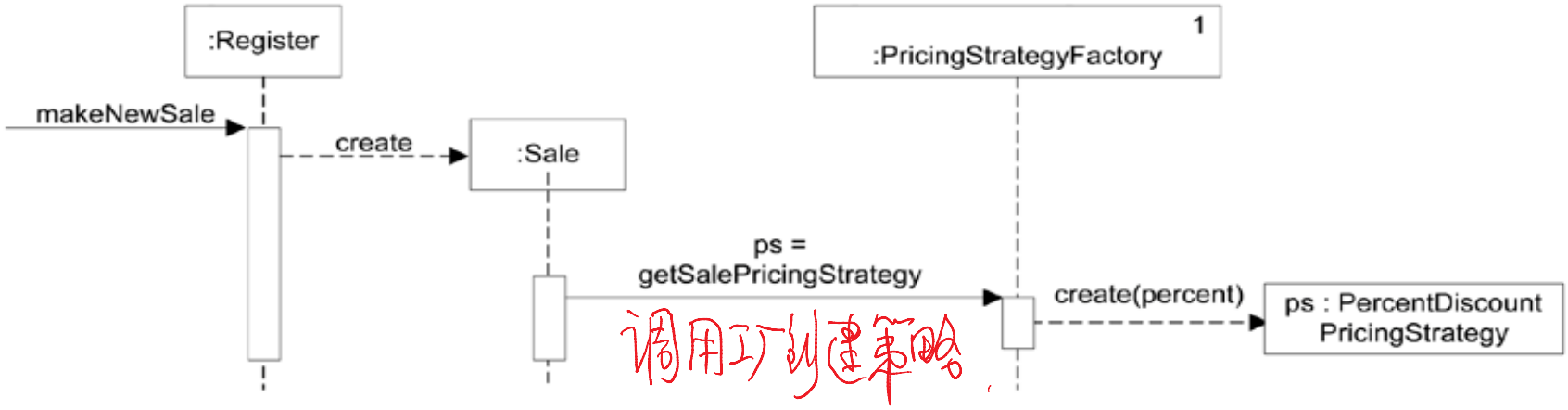
Factory for Strategies

- It can read the name of the implementation class of the pricing strategy from a system property (or some external data source), and then make an instance of it.
- This is a partial *data-driven design* (or reflective design).
- The pricing policy can be dynamically changed by specifying a different class of Strategy to create.


```
{
  String className = System.getProperty( "salepricingstrategy.class.name" );
  strategy = (ISalePricingStrategy) Class.forName( className ).newInstance();
  return strategy;
}
```

Creating a Strategy

- When a *Sale* instance is created, it can ask the factory for its pricing strategy



另一个模式

Composite (GoF) 组合模式

- **Name:** Composite
- **Problem:**
  How to treat a group or composition structure of objects the same way (polymorphically) as a non-composite (atomic) object? 如何将一个组合结构看成是一个原子对象?
- **Solution:**
  Define classes for composite and atomic objects so that they implement the same interface. 创建组合和原子对象，共实现统一接口.
- **Related Patterns:**
  Composite is often used with the Strategy and Command patterns. Composite is based on Polymorphism and provides Protected Variations to a client so that it is not impacted if its related objects are atomic or composite.

Example:

Multiple, Conflicting Pricing Policies

多个价，能决定的价格.

- Pricing strategies in Store
  1. time period (Monday) 时段
  2. customer type (senior) 客户类型
  3. a particular line item product (Darjeeling tea) 特定商品

  } 三种因素

- Percentage discount (today) 按比例减价方式
  - 20% senior discount policy
  - preferred customer discount of 15% off sales over $400
  - buy 1 case of Darjeeling tea, get 15% discount off of everything
- Absolute Discount over Threshold 按绝对值减价方式
  - on Monday, there is $50 off purchases over $500

Store's Conflict Resolution Strategy

冲突消除策略.

- Usually applies the "best for the customer" (lowest price)

通常按最优价的客户订价.

- But this is not required, and it could change.

但这可能变化.

- For example, the store may have to use a "highest price" conflict resolution strategy during a difficult financial period.

商店可能使用最高优惠的客户订价.

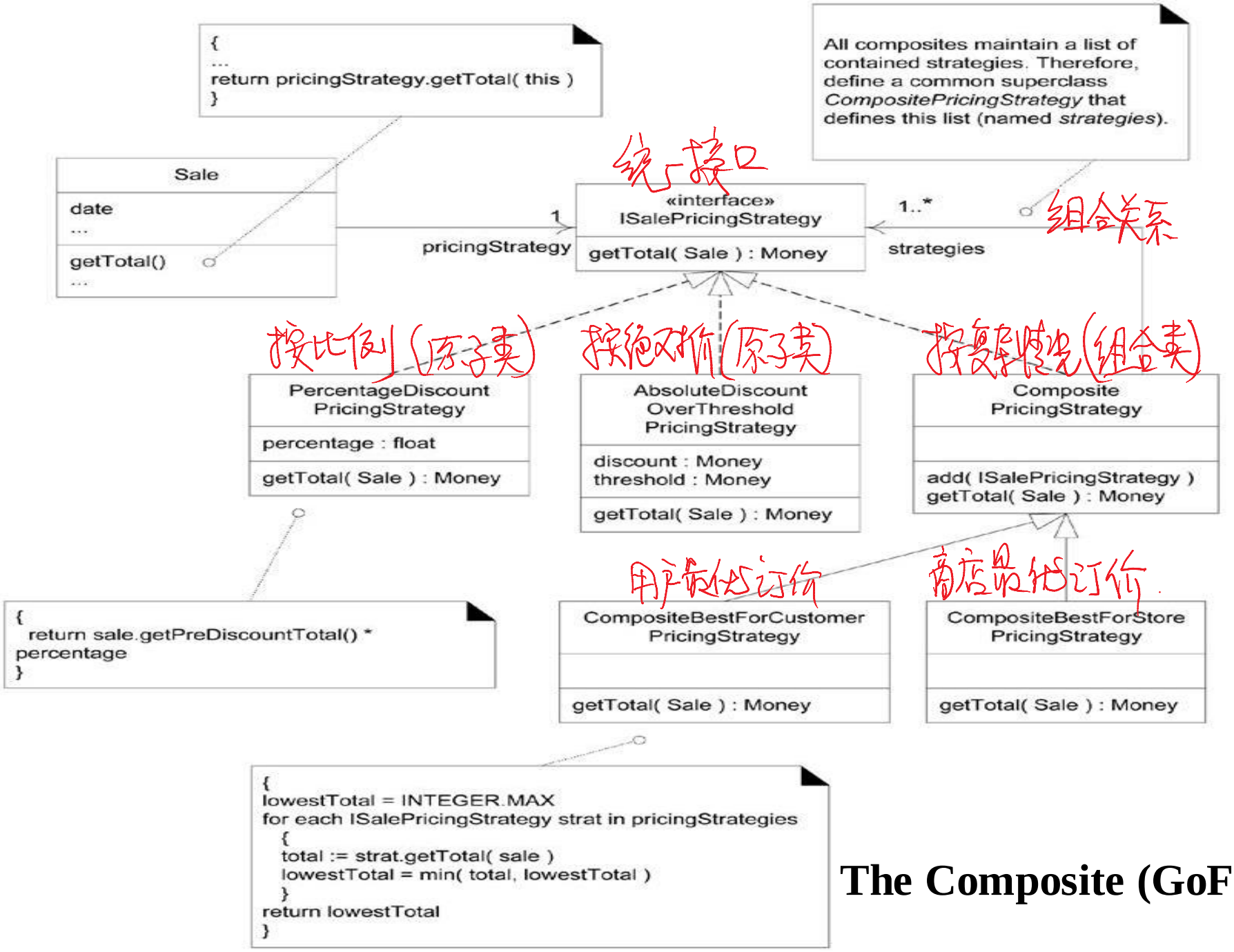
Exist Multiple Co-existing Strategies

- There can exist multiple co-existing strategies, that is, one sale may have several pricing strategies.
- A pricing strategy can be related to the type of
  - customer (for example, a senior).
    - The customer type must be known by the *StrategyFactory* at the time of creation of a pricing strategy for the customer.
  - product being bought (for example, Darjeeling tea).
    - The *ProductDescription* must be known by the *StrategyFactory* at the time of creation of a pricing strategy influenced by the product.

How do we handle the case of multiple, conflicting pricing policies?

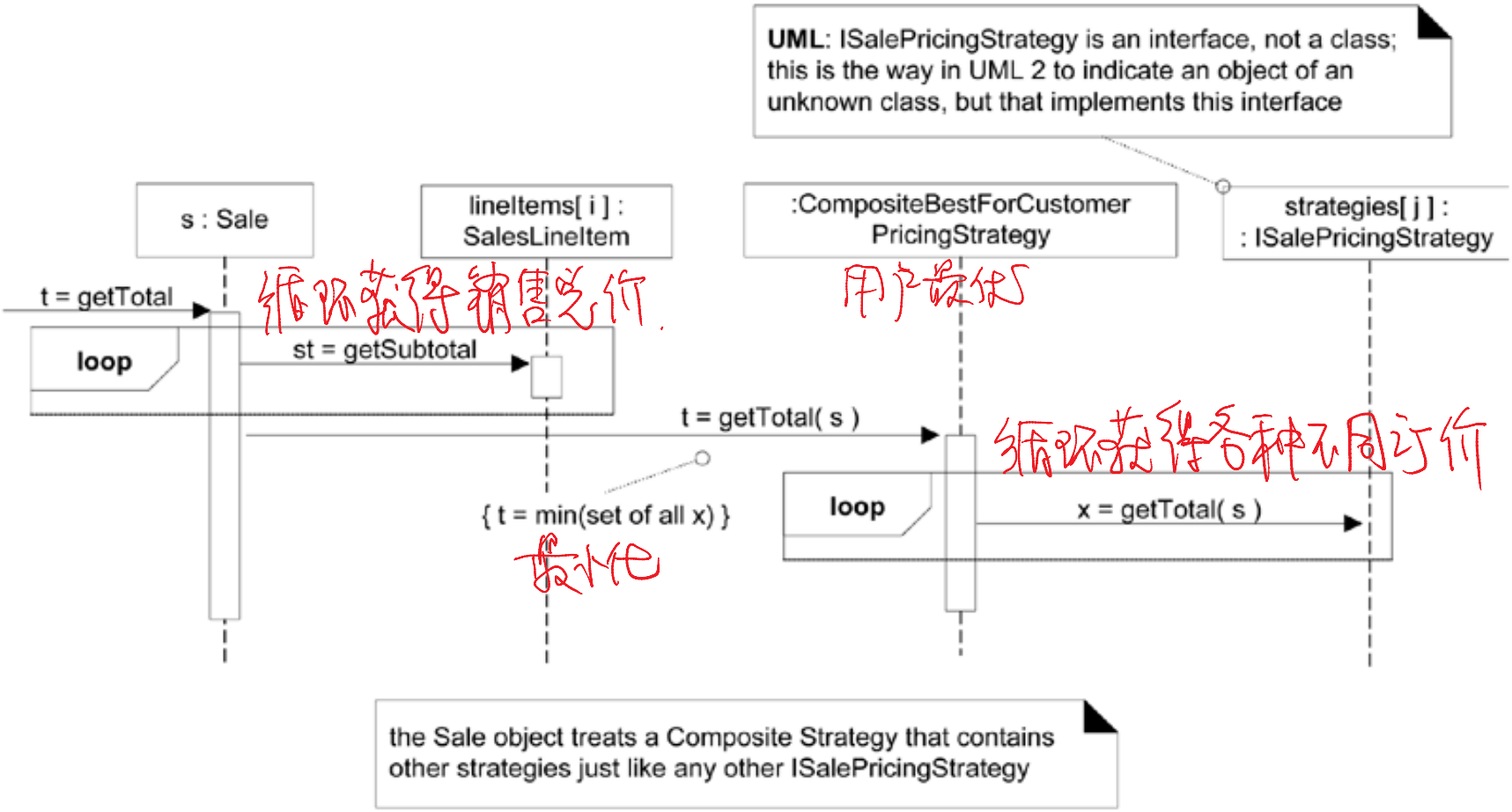
如何应对这种多种相似的情况？

- Is there a way to change the design so that the *Sale* object does not know if it is dealing with one or many pricing strategies, and also offer a design for the conflict resolution?
- Yes, with the Composite pattern. 可以使用组合模式



The Composite (GoF)

Collaboration with a Composite



CompositeBestForCustomerPricingStrategy

```
// superclass so all subclasses can inherit a
List of strategies

Public abstract  class
CompositePricingStrategy
 implements ISalePricingStrategy
{

protected List strategies = new ArrayList();

public add( ISalePricingStrategy  s )
{
  strategies.add( s );
}

public abstract Money getTotal( Sale sale );

} // end of class
```

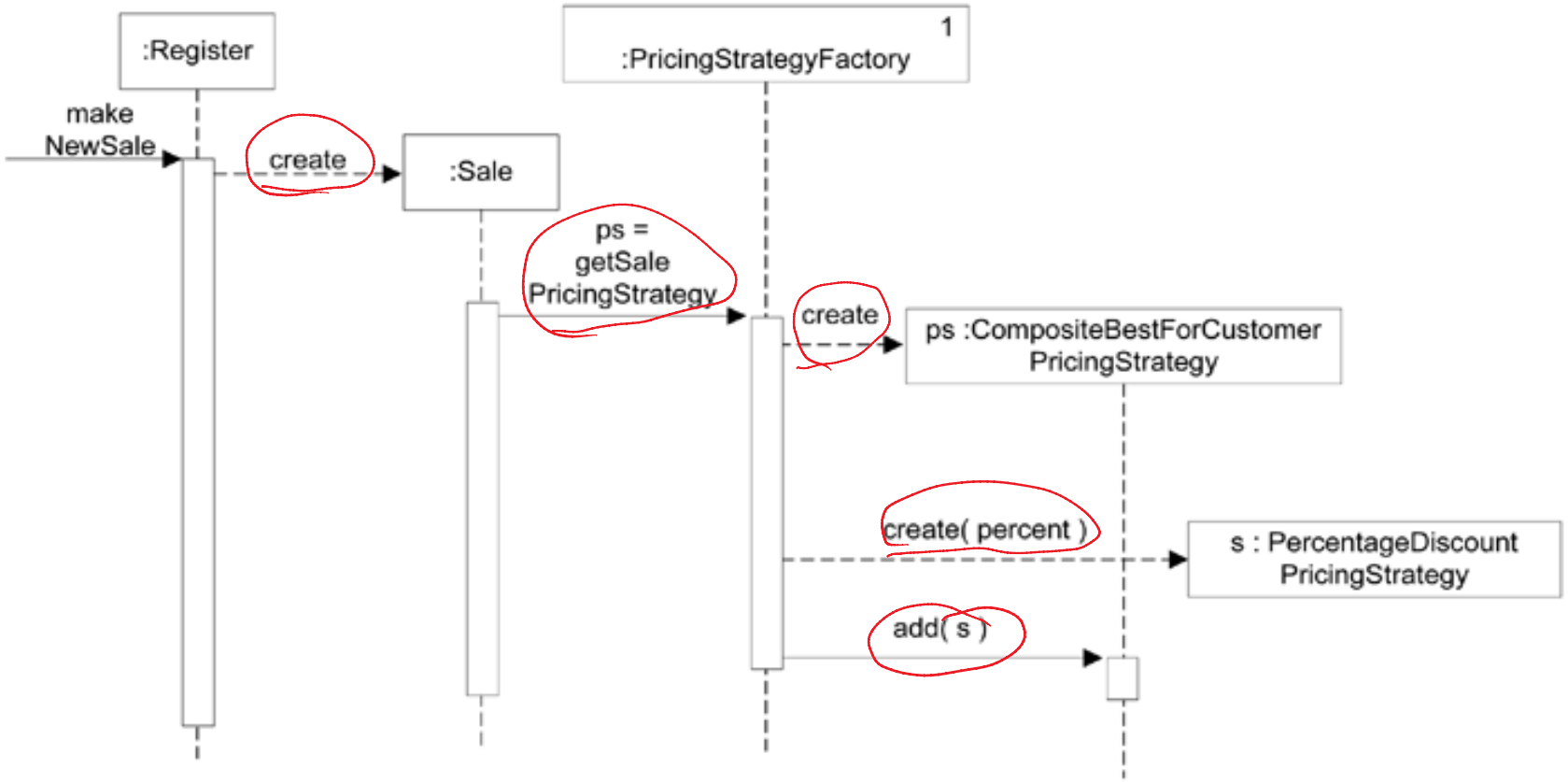
```
// a Composite Strategy that returns the lowest total
// of its inner SalePricingStrategies

public class
CompositeBestForCustomerPricingStrategy
extends CompositePricingStrategy
{
public Money getTotal( Sale sale )
{
  Money lowestTotal = new Money( Integer.MAX_VALUE );
  // iterate over all the inner strategies
  for( Iterator i = strategies.iterator(); i.hasNext(); )
  {
    ISalePricingStrategy strategy =
      (ISalePricingStrategy)i.next();
    Money total = strategy.getTotal( sale );
    lowestTotal = total.min( lowestTotal );
  }
return lowestTotal;
}
} // end of class
```

Creating Multiple SalePricingStrategies

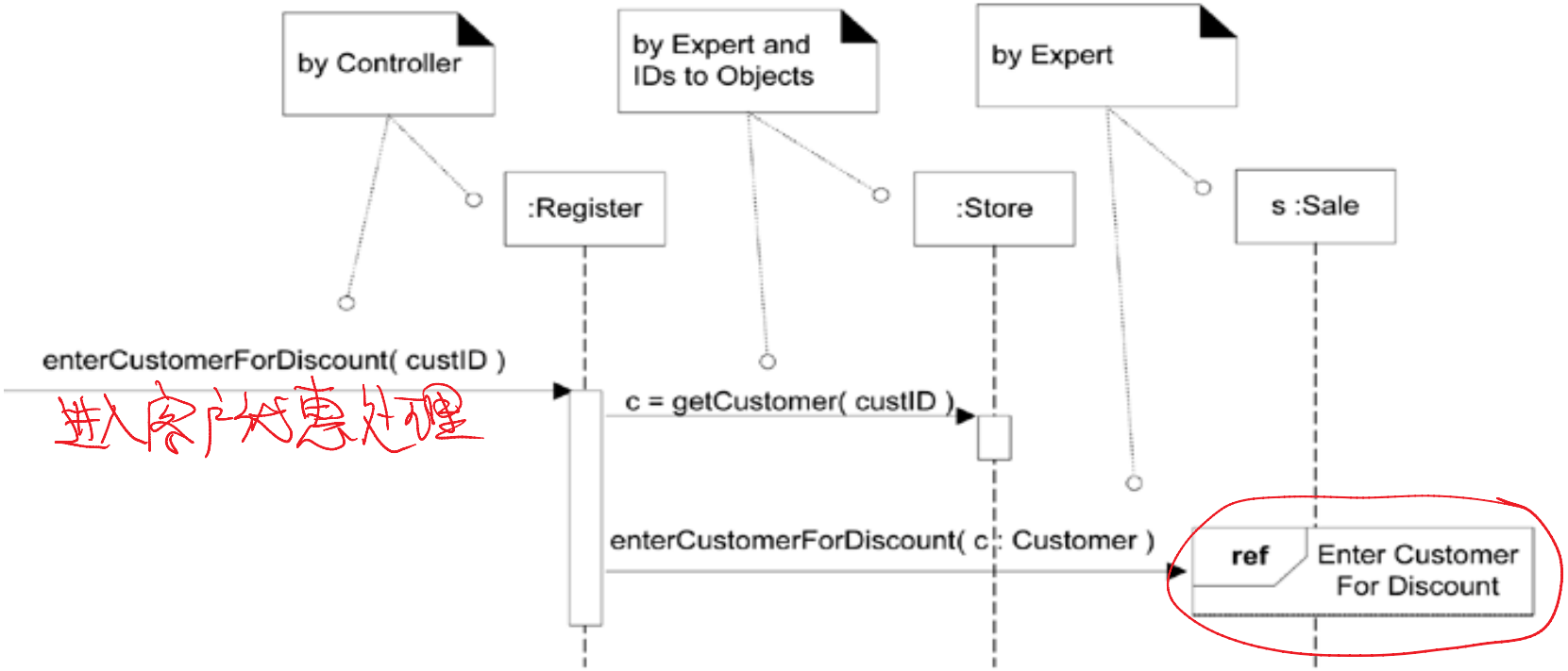
- When do we create these strategies? 什么时候创建策略对象?
- A desirable design will start by creating a Composite that contains the present moment's store discount policy. 开始组合中包含一个当前策略.
- Then, if later another pricing strategy is discovered to also apply (such as senior discount), add it to the composite using the inherited *CompositePricingStrategy.add* method. 后面应用到其它策略时再加入
- There are three points in the scenario where pricing strategies may be added to the composite: 三种情况需加入新策略:
  1. Current store-defined discount, added when the sale is created. Sale创建时加入当前商店定义的优惠策略.
  2. Customer type discount, added when the customer type is communicated to the POS. 用户类型出现时,再加入用户类型相关的策略.
  3. Product type discount (if bought Darjeeling tea, 15% off the overall sale), added when the product is entered to the sale. 当有优惠的商品出现时,加入此类优惠计价.

Creating a composite strategy for the first case

创建Sale对象时，第一次加入何策略

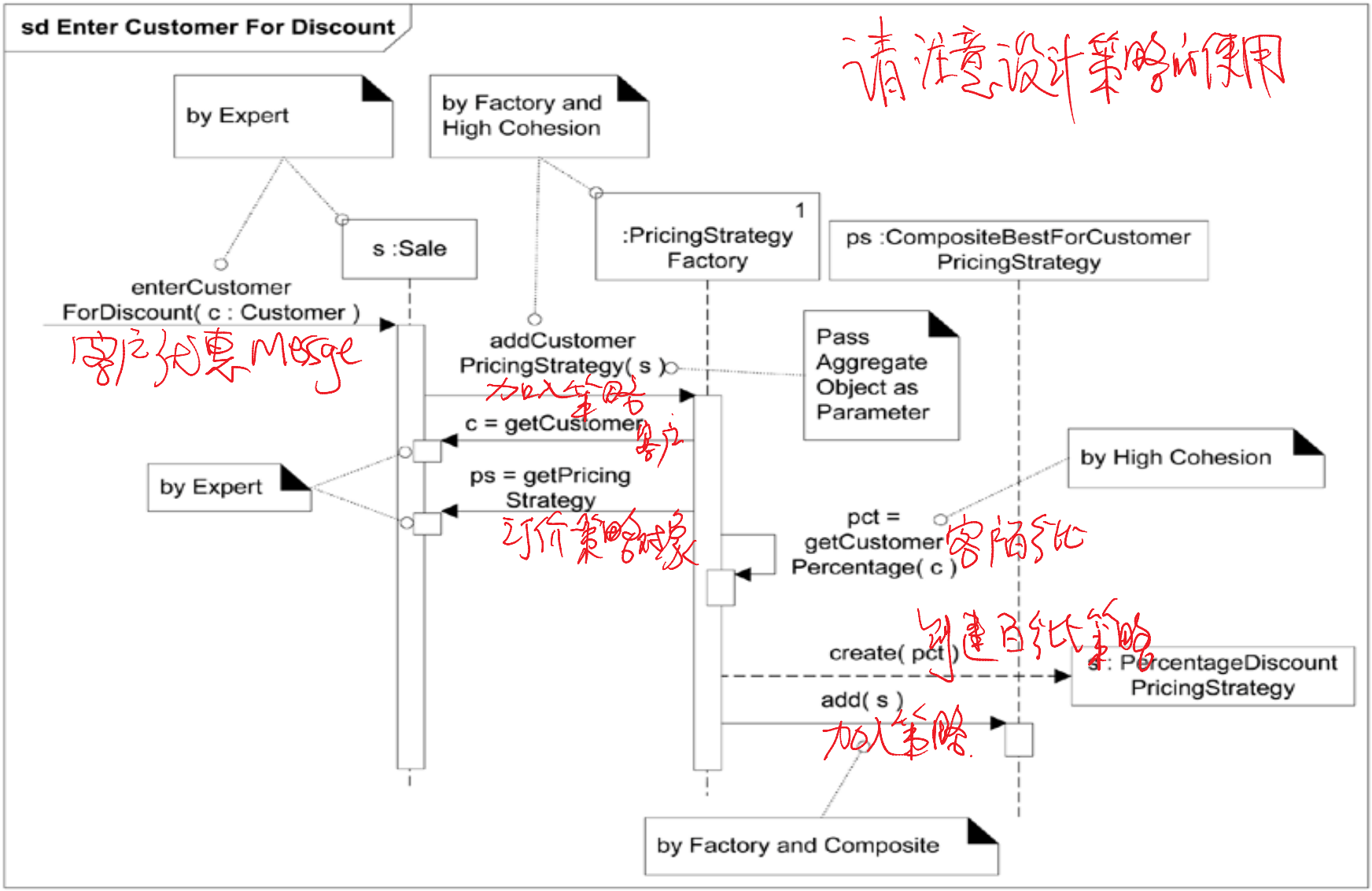


Creating the pricing strategy for a customer discount



请注意设计原则的利用.

Creating the pricing strategy for a customer discount



在设计中发现新用例

Use Case UC1: Process Sale

更新用例

...

Extensions (or Alternative Flows):

5b. Customer says they are eligible for a discount (e.g., employee, preferred customer)

客户申请价格优惠.

1. Cashier signals discount request.
   - Cashier enters Customer identification.
   - System presents discount total, based on discount rules.

This indicates a new system operation on the POS system, in addition to *makeNewSale, enterItem, endSale*, and *makePayment*. We will call this fifth system operation *enterCustomerForDiscount*; it may optionally occur after the *endSale* operation.

这意味着系统的一个新的操作.

Composite Summary

- This design problem was squeezed for many tips in object design. 设计问题给许多技巧.
- A skilled object designer has many of these patterns committed to memory through studying their published explanations, and has internalized core principles, such as those described in the GRASP family.

学习并掌握设计模式才能成为老练的设计师.

- Please note that although this application of Composite was to a Strategy family, the Composite pattern can be applied to other kinds of objects, not just strategies. Composite可以应用在很多地方.
- For example, it is common to create "macro commands“ commands that contain other commands, through the use of Composite.

宏命令可以用Composite.