System Analysis and Design

L28. More Object Design with GoF Patterns

Topics

- Façade
- Observer
- Local caching
- Failover to a local service when a remote service fails

另一个设计模式：

Facade (GoF)

- Name: Facade
- **Problem:**

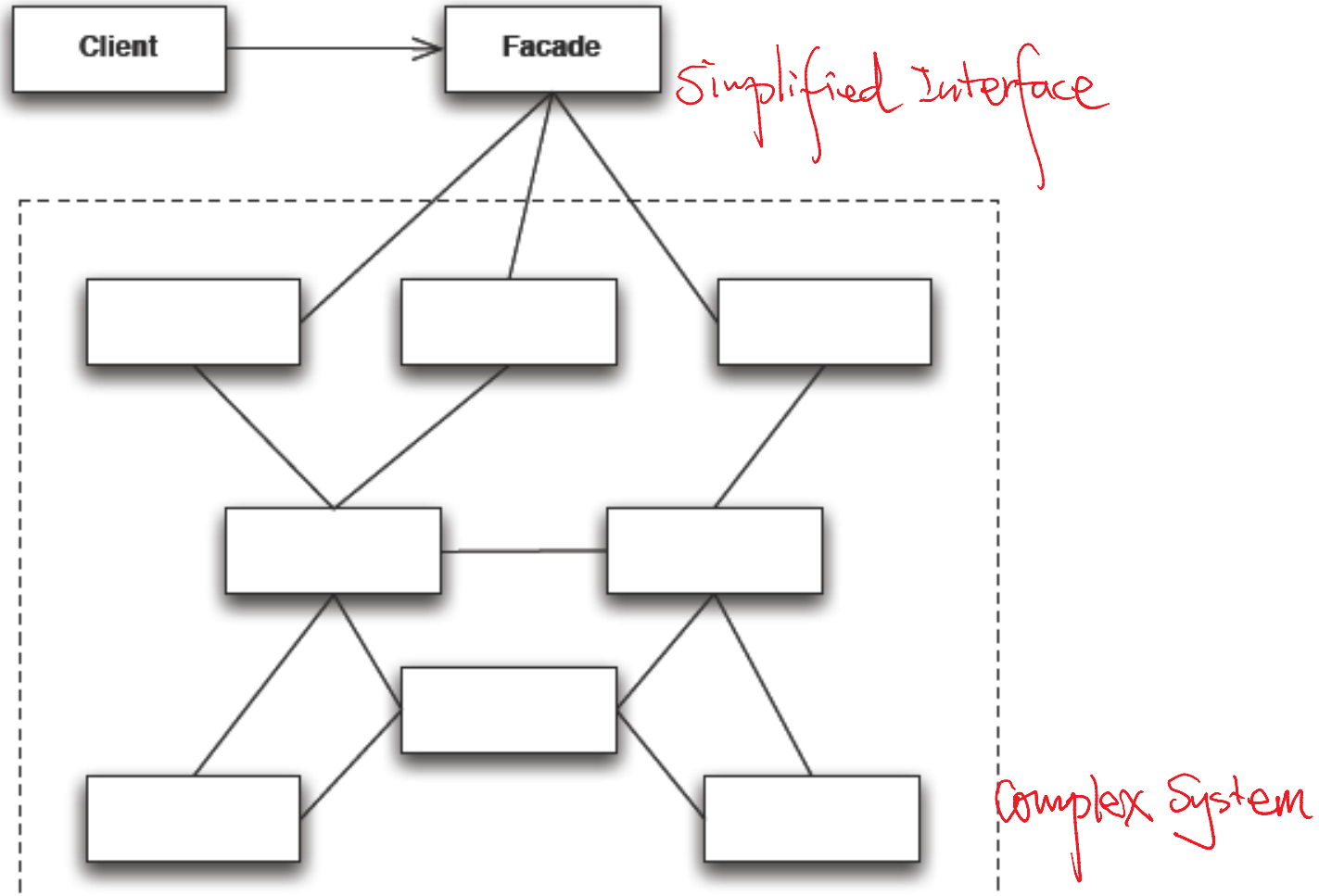
  A common, unified interface to a disparate set of implementations or interfaces, such as within a sub-system, is required. There may be undesirable coupling to many things in the subsystem, or the implementation of the subsystem may change. What to do?
- **Solution:**

  Define a single point of contact to the sub-system, a **facade object that wraps** the subsystem. This facade object presents a single unified interface and is responsible for collaborating with the subsystem components.

Facade

- “Provide a unified interface to a set of interfaces in a subsystem.
- Façade defines a higher-level interface that makes the subsystem easier to use.”
- There can be significant benefit in wrapping a complex subsystem with a simplified interface
  - If you don’t need the advanced functionality or fine-grained control of the complex system, the simplified interface makes life easy

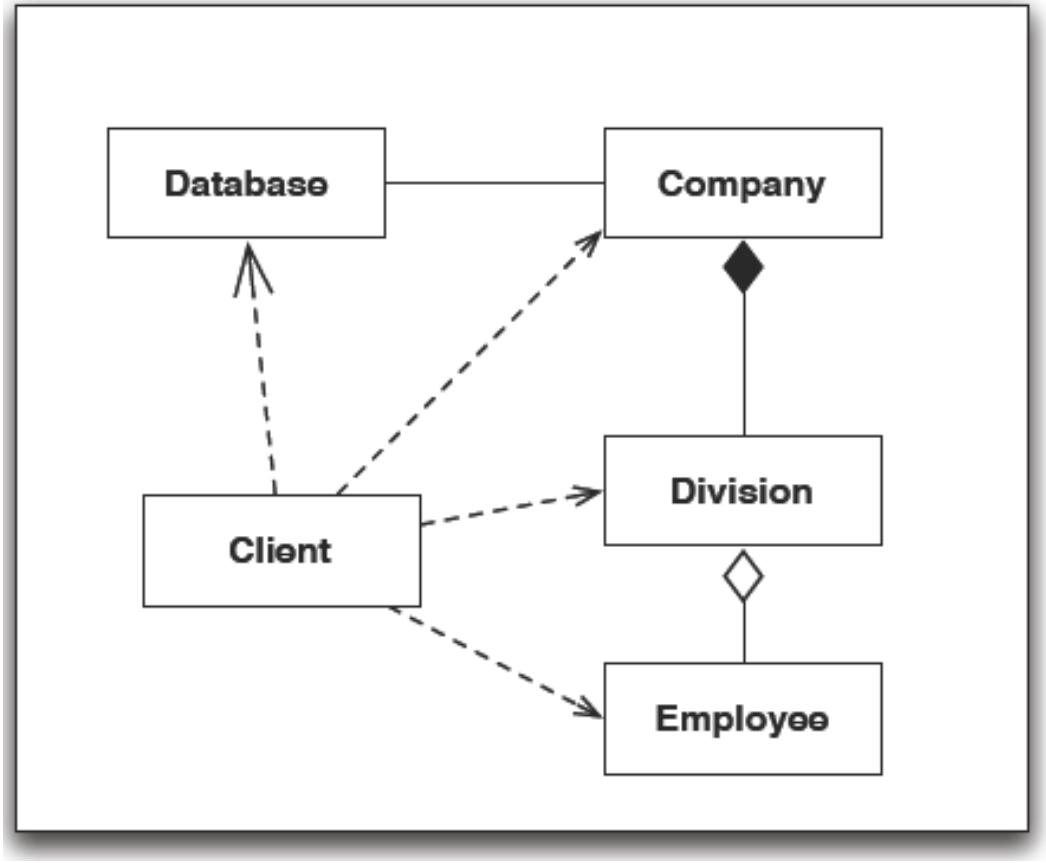
Facade Pattern: Structure



Use Facade

- Facade works best when you are accessing a subset of the subsystem's functionality
  - You can also add new features by adding it to the Facade (not the subsystem); you still get a simpler interface
- Facade not only reduces the number of methods you are dealing with but also the number of classes
  - Imagine having to pull Employees out of Divisions that come from Companies that you pull from a Database
    - A Facade in this situation can fetch Employees directly

Example (Without a Facade)



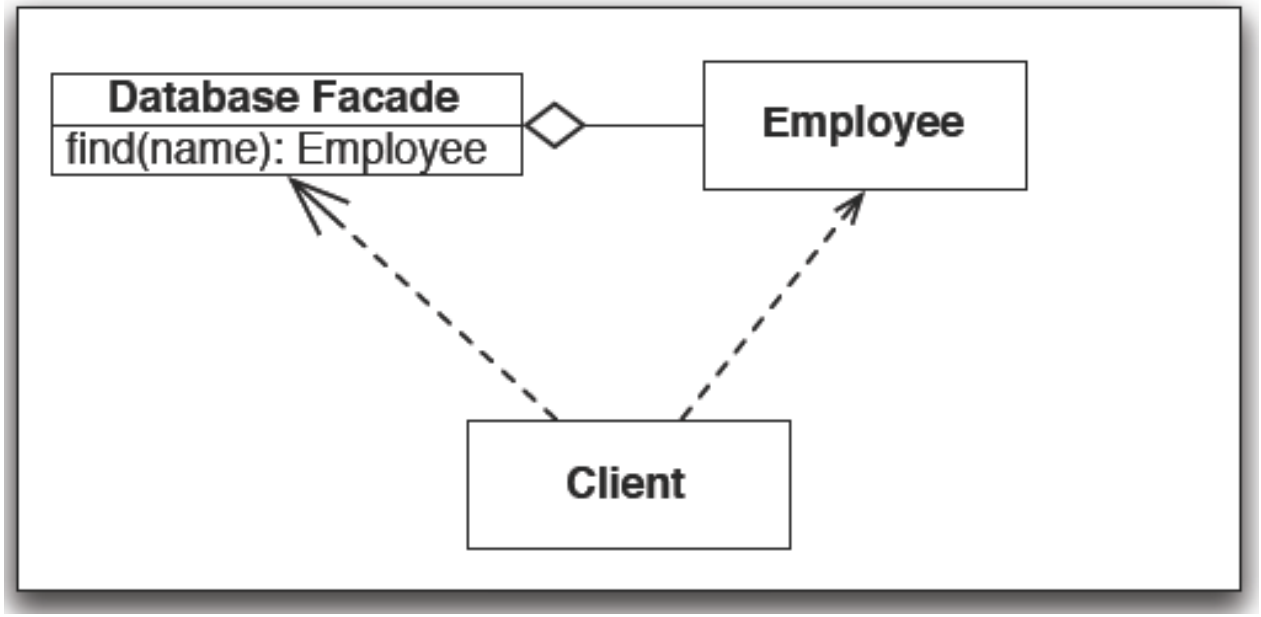
Without a Facade, Client contacts the Database to retrieve Company objects. It then retrieves Division objects from them and finally gains access to Employee objects.

It uses four classes.

无Façade时

使用到四个类

Example (With a Facade)



With a Facade, the Client is shielded from most of the classes.

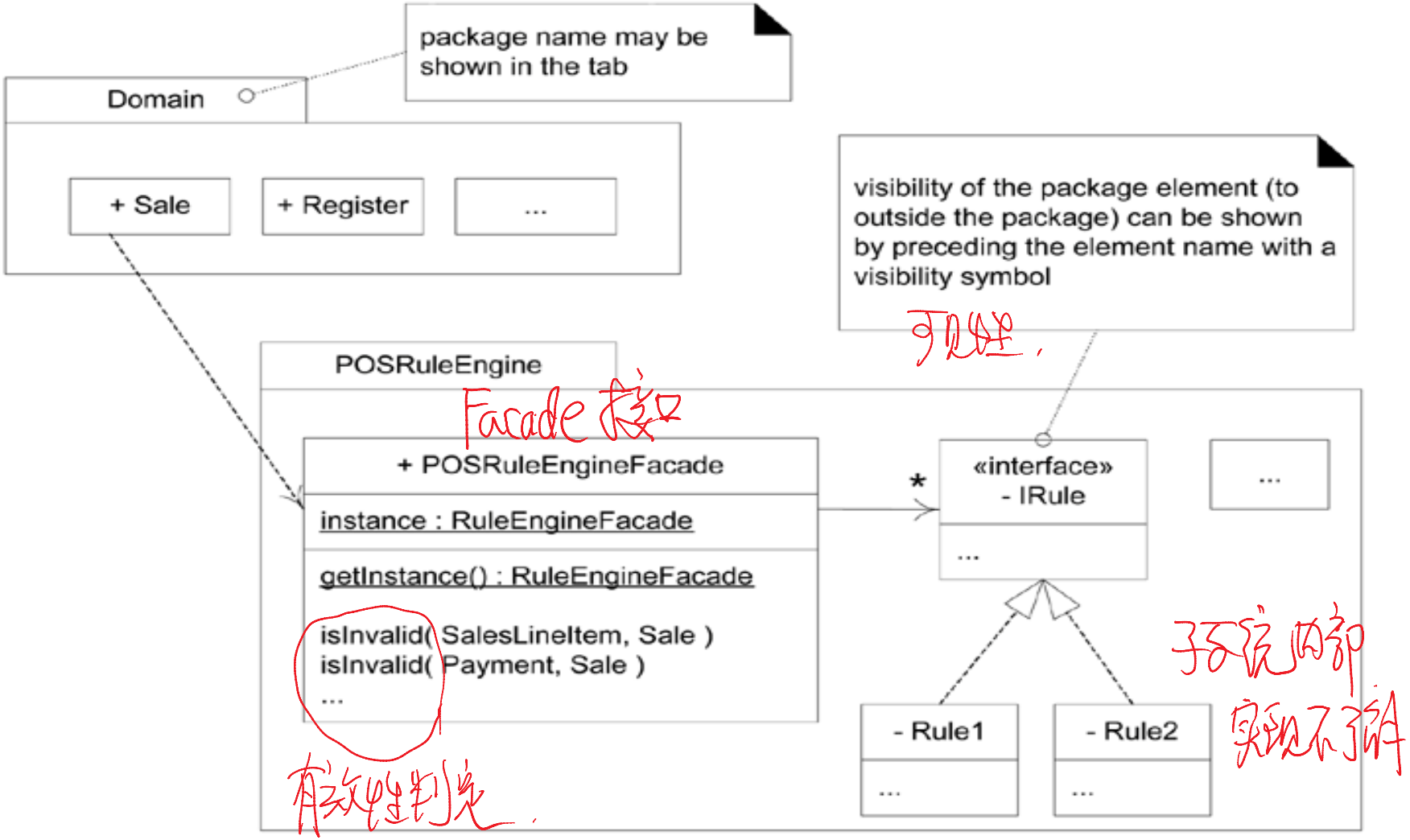
It uses the Database Facade to retrieve Employee objects directly.

客户程序与之多数类隔离。

Example: Rule Engine for POS

- We will define a "rule engine" subsystem, whose specific implementation is not yet known.
- It will be responsible for evaluating a set of rules against an operation (by some hidden implementation), and then indicating if any of the rules invalidated the operation.

POSRuleEngineFacade 门面接口



POSRuleEngineFacade

```
public class Sale
{
      public void makeLineItem( ProductDescription desc, int quantity )
      {
            SalesLineItem sli = new SalesLineItem( desc, quantity );

            // call to the Facade 调用 Facade
            if ( POSRuleEngineFacade.getInstance().isInvalid( sli, this ) )
            return;

            lineItems.add( sli );
      }
      // ...
} // end of class
```

POS设计只与POSRuleEngineFacade有关，与RuleEngine子系统无关

POSRuleEngineFacade

- With this design, the complexity and implementation of how rules will be represented and evaluated are hidden in the "rules engine" subsystem, accessed via the *POSRuleEngineFacade* facade. 规则子系统隐藏，只有一个facade
- The subsystem hidden by the facade object could contain dozens or hundreds of classes of objects, or even a non-object-oriented solution, yet as a client to the subsystem, we see only its one public access point. 规则子系统可能很复杂，甚至不是OO设计。
- A separation of concerns has been achieved to some degree, all the rule-handling concerns have been delegated to another subsystem. 分离性设计

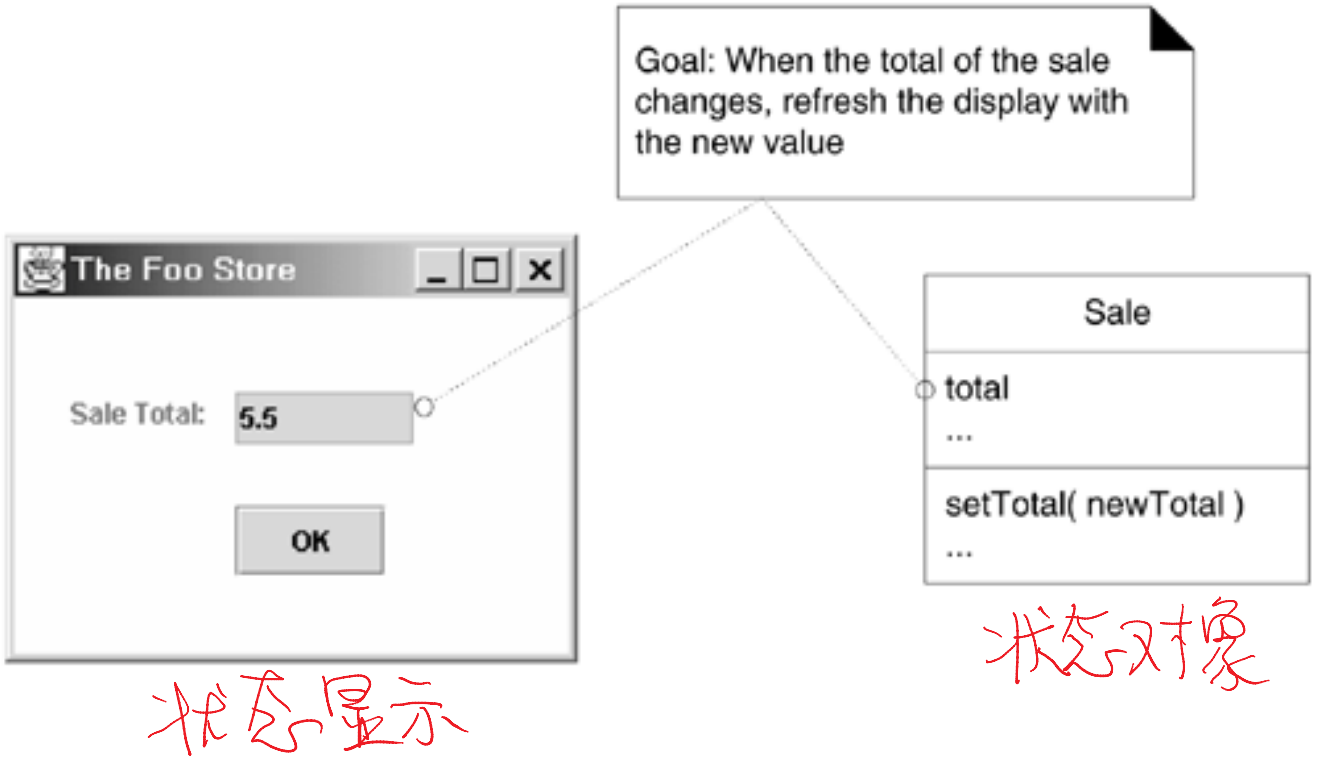
Facade (GoF)

- A Facade is a "front-end" object that is the single point of entry for the services of a subsystem
- The implementation and other components of the subsystem are private and can't be seen by external components.
- Facade provides Protected Variations from changes in the implementation of a subsystem.

另一个设计模式：

Observer (GoF)

- Observer/Publish-Subscribe/Delegation Event Model (GoF)
- The requirement is adding the ability for a GUI window to refresh its display of the sale total when the total changes



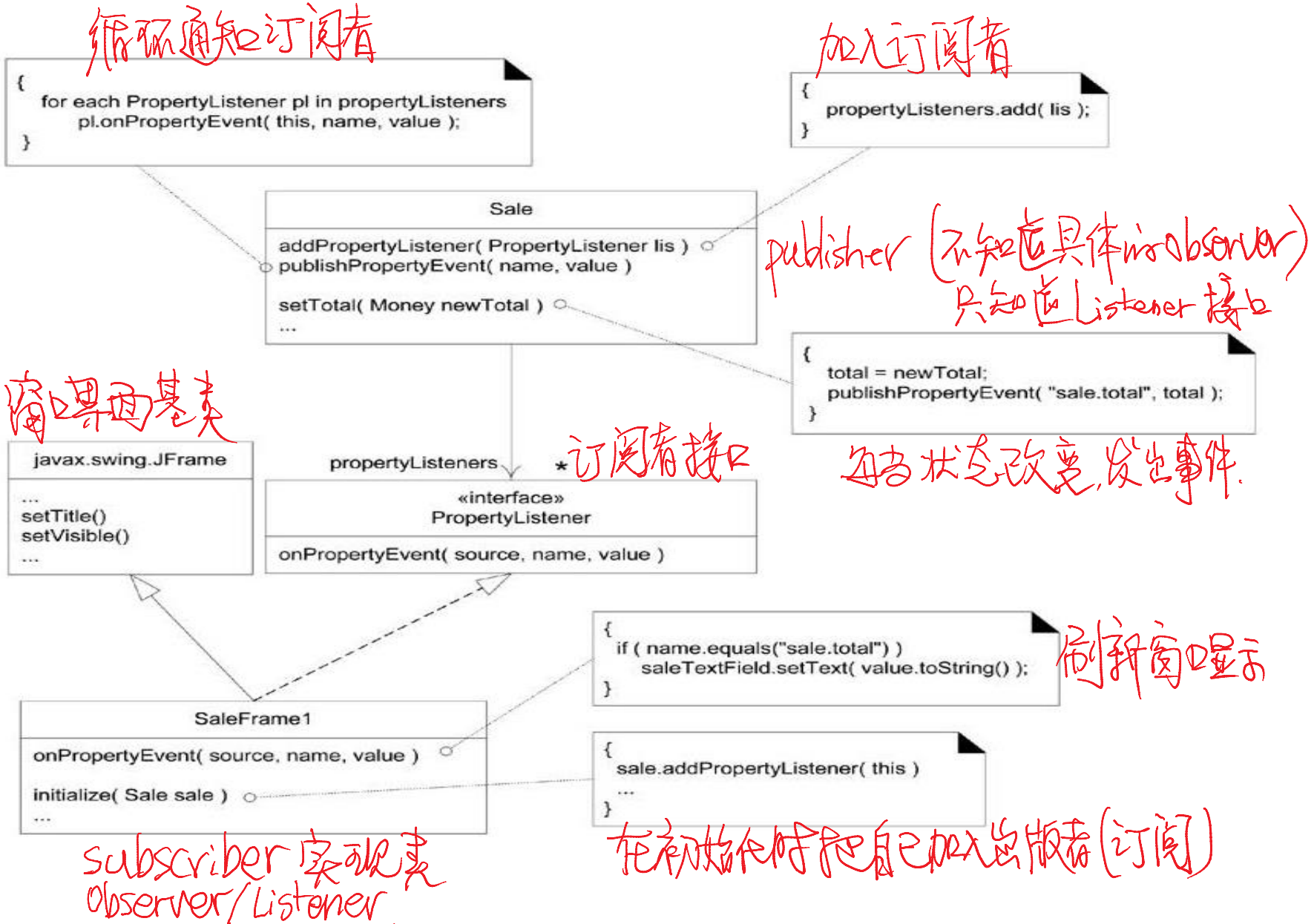
Observer (GoF)

- Why not do the following as a solution?
  When the *Sale* changes its total, the *Sale* object sends a message to a window, asking it to refresh its display.
- To review, the Model-View Separation principle discourages such solutions.
- To solve this design problem, the Observer pattern can be used.

Observer (GoF)

- Name: Observer (Publish-Subscribe)
- **Problem:**
  Different kinds of ***subscriber objects*** are interested in the ***state changes or events of a publisher object***, and want to react in their own unique way when the publisher generates an event. Moreover, the publisher wants to maintain low coupling to the subscribers. What to do?
- **Solution:**
  Define a "subscriber" or "listener" interface. Subscribers implement this interface. The publisher can dynamically register subscribers who are interested in an event and notify them when an event occurs.

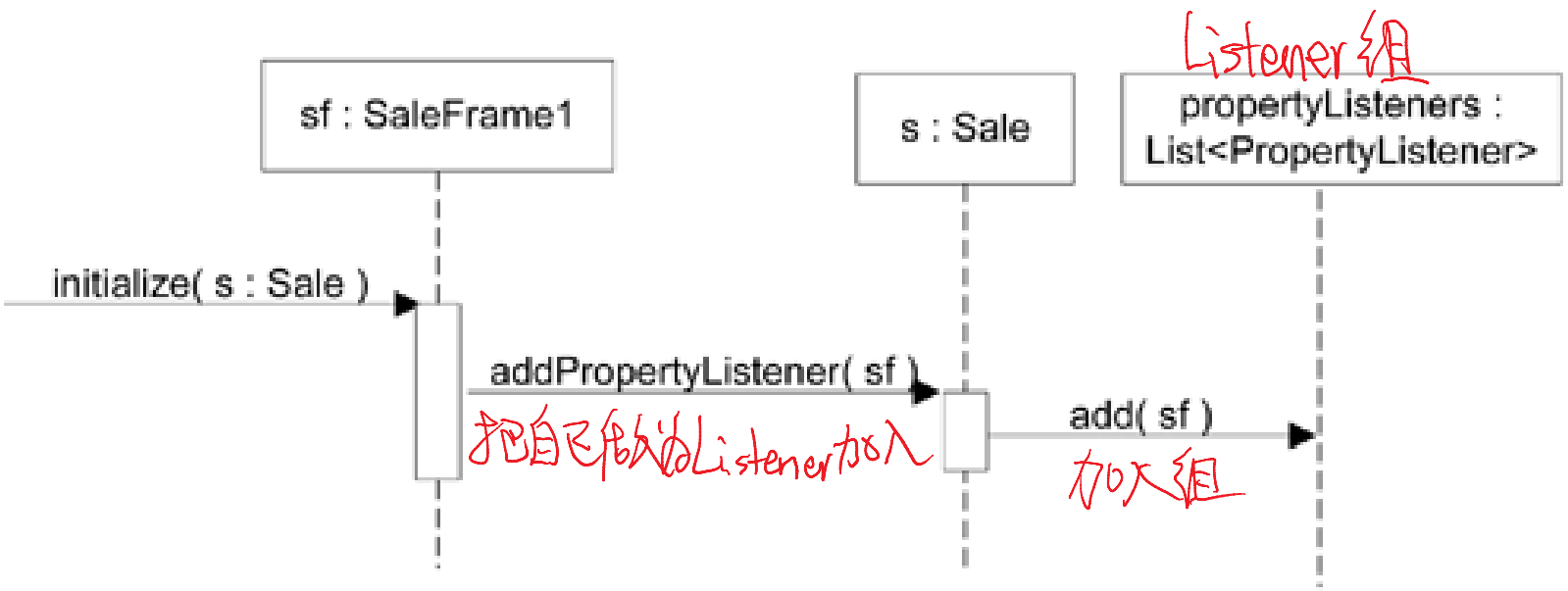
The Observer Pattern



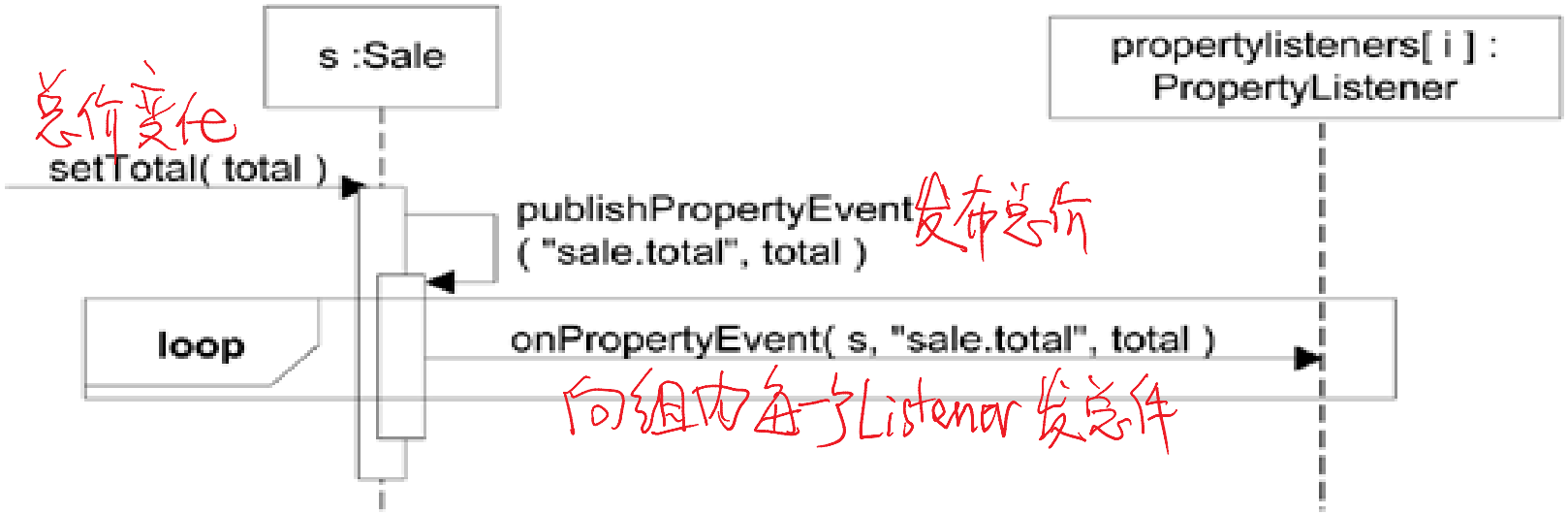
Understanding Observer

1. Interface *PropertyListener* with the operation *onPropertyEvent* is defined.
2. Define the window *SaleFrame1* to implement the interface.
3. When the *SaleFrame1* window is initialized, pass it the *Sale* instance from which it is displaying the total.
4. The *SaleFrame1* window registers or *subscribes* to the *Sale* instance via the *addPropertyListener*. That is, when a property (such as total) changes, the window wants to be notified.
5. *Sale* does not know about *SaleFrame1* objects; it only knows about objects that implement the *PropertyListener* interface. This lowers the coupling of the *Sale* to the window, the coupling is only to an interface, not to a GUI class.
6. The *Sale* instance is thus a *publisher* of "property events." When the total changes, it iterates across all subscribing *PropertyListeners*, notifying each.

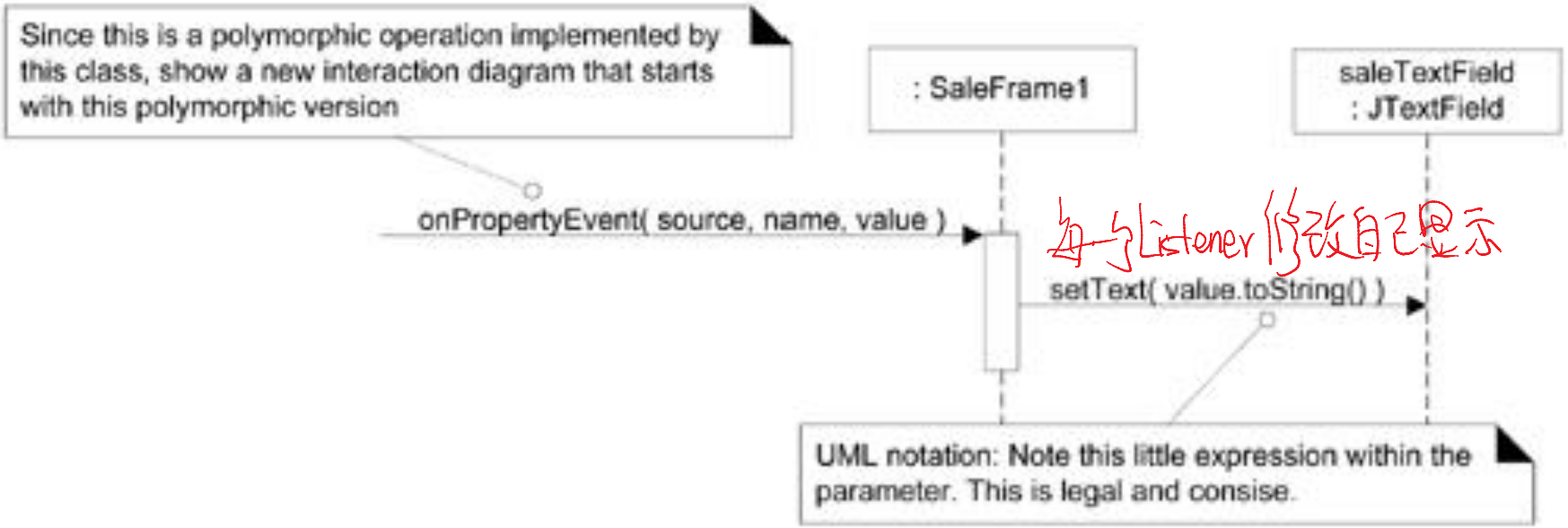
The observer SaleFrame1 subscribes to the publisher Sale



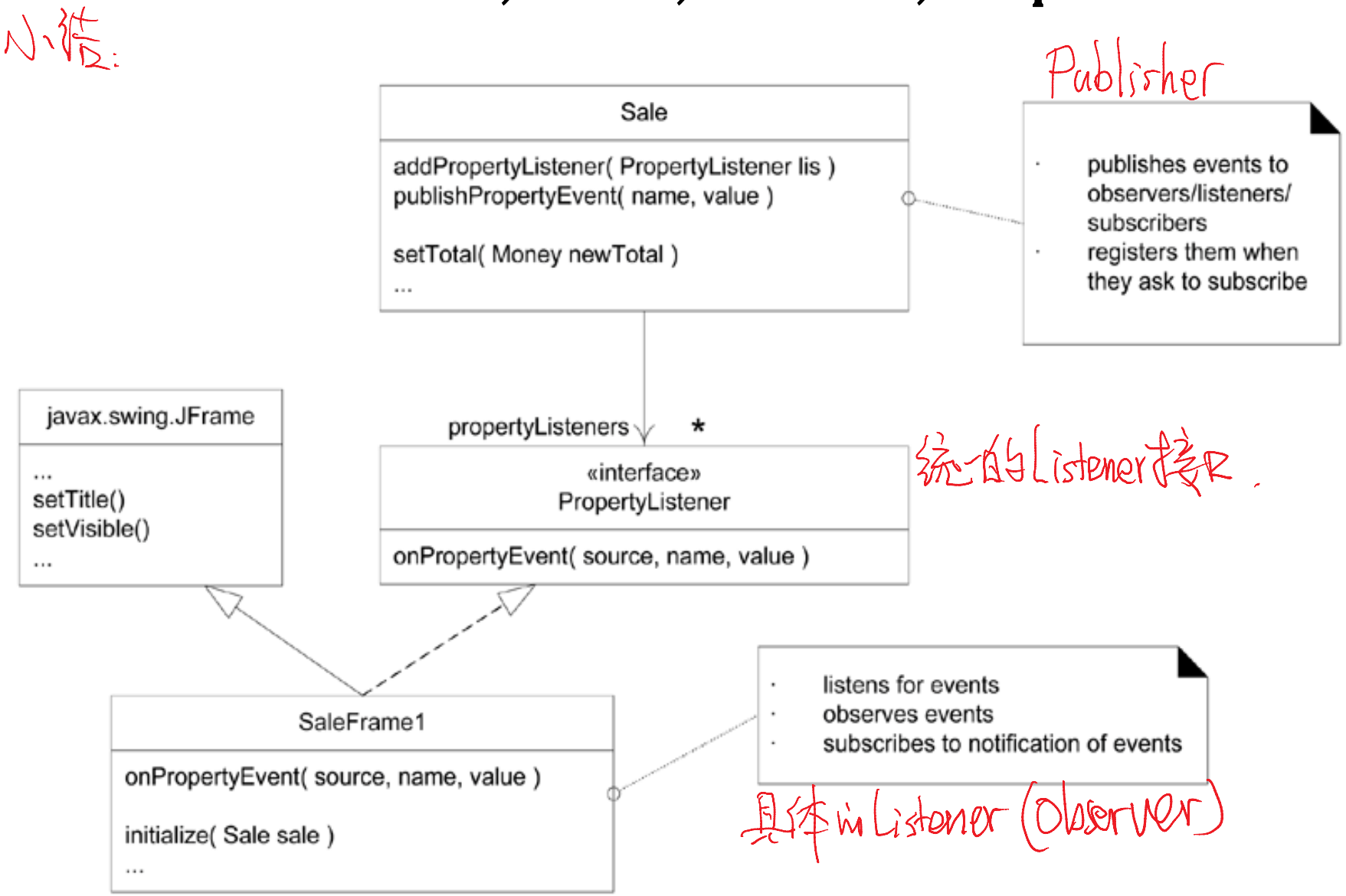
The Sale publishes a property event to all its subscribers



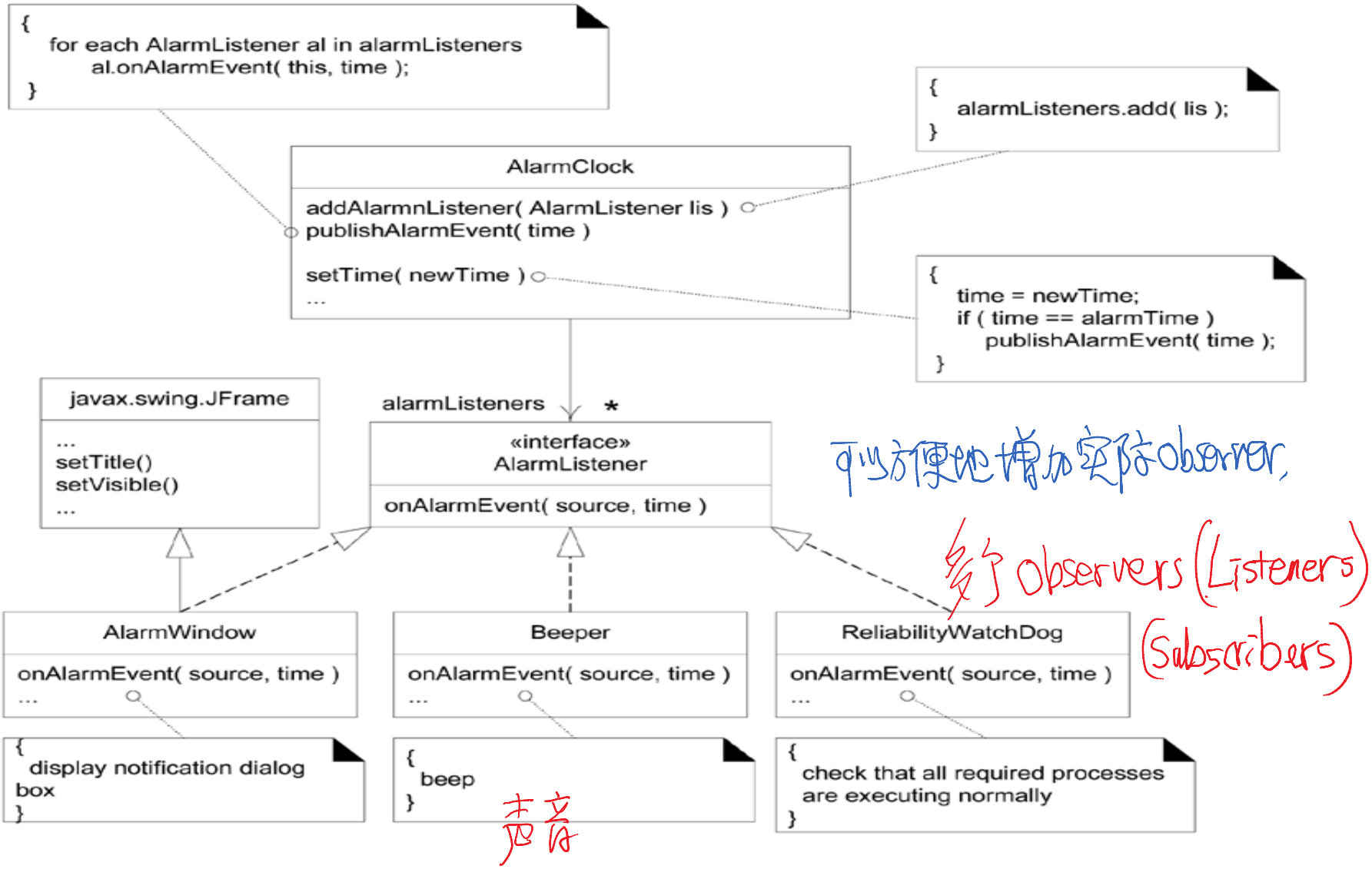
The subscriber SaleFrame1 receives notification of a published event



Who is the observer, listener, subscriber, and publisher?



Observer applied to alarm events, with different subscribers



我们讨论一些新的需求及其设计

NextGen POS Requirements

- Local caching 本地缓存.
- Failover to a local service when a remote service fails 失败支对.

} 下面讨论

Local Caching

- Some database elements and other data can be persisted locally such that if the service that provides them isn't available, you have some (limited) functionality
- For example, if the ProductDescription database is offline, the most frequently-purchased products can be found in a file on a local disk

Levels of Cache

- *In-memory* caching can keep some fixed number of elements. Sometimes this may be then entire list.
- *Local hard-drive* caching could limit by file size rather than number. (How would you organize this?)
- Which patterns does this use so far?
  Adapter and Factory

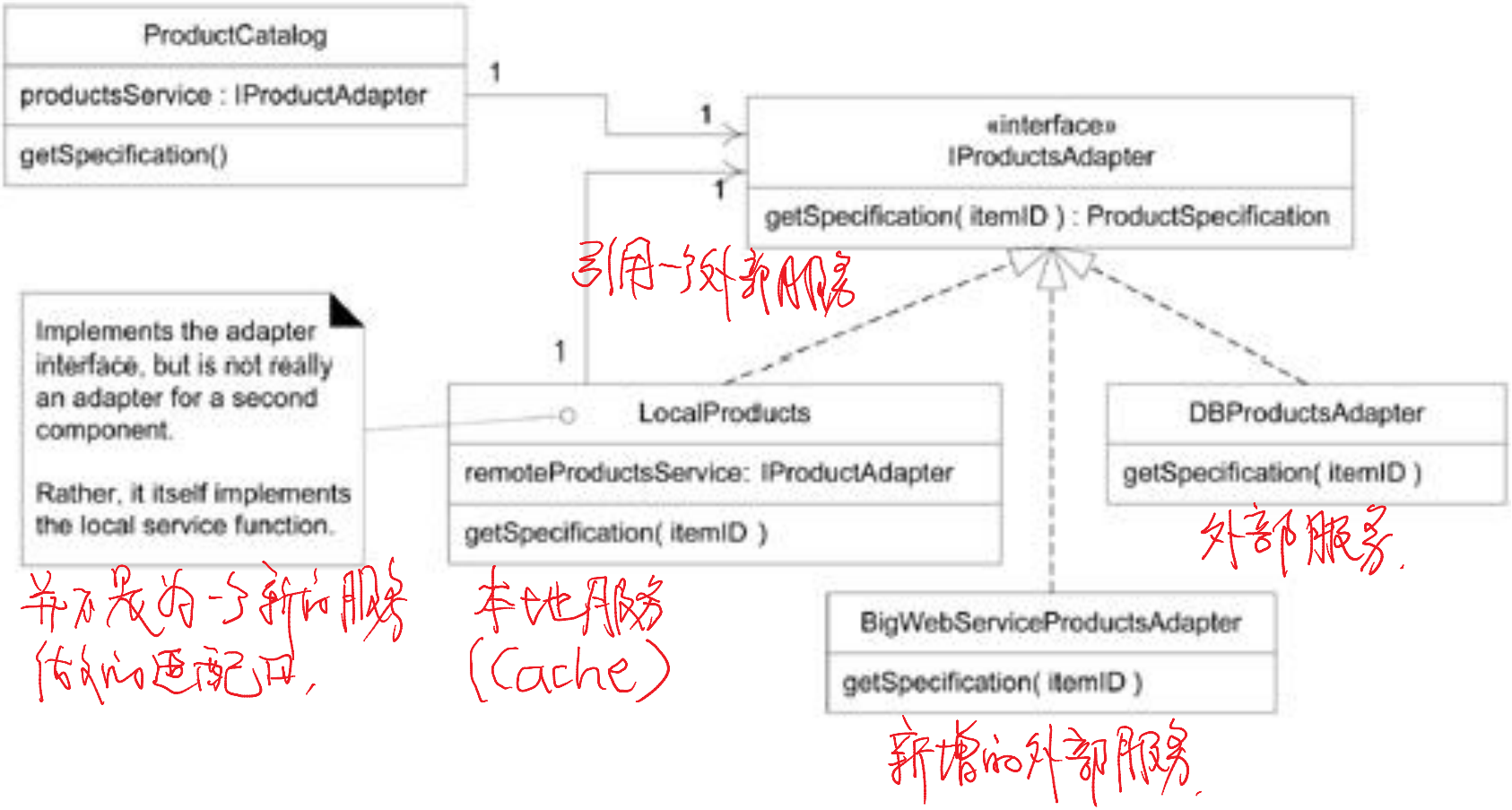
回想上次课我们在Factory中实现存储管理(Caching)

Local service with existing adapter and factory

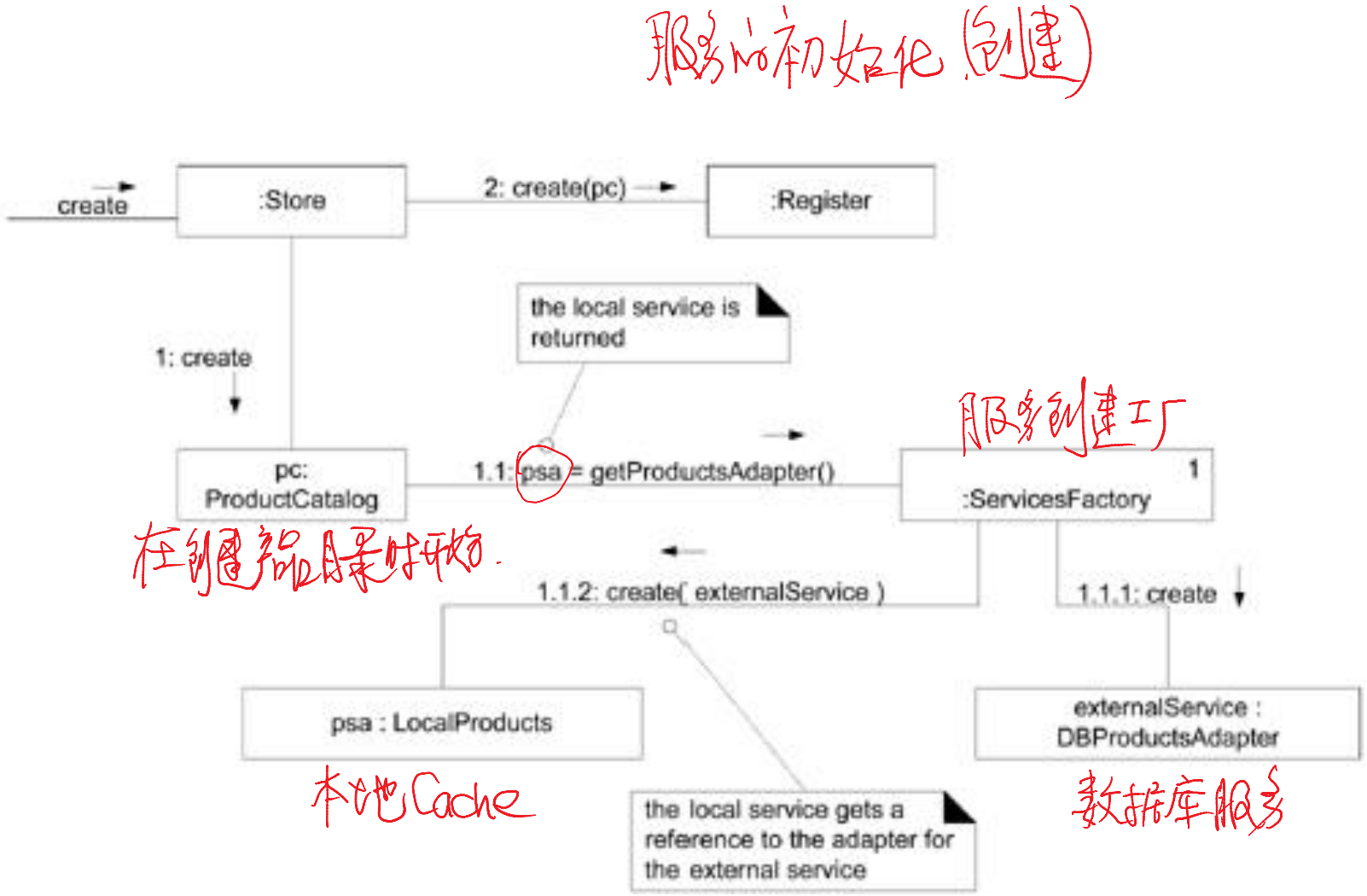
在Adapter和Factory中实现Local Cache.

1. The *ServicesFactory* will always return an adapter to a local product information service. 返回一个本地服务(cache)
2. The local products "adapter" is not really an adapter to another component. It will itself implement the responsibilities of the local service. 本地服务是一个Cache
3. The local service is initialized to a reference to a second adapter to the true remote product service. 本地服务引用一个实际的外部服务.
4. If the local service finds the data in its cache, it returns it; otherwise, it forwards the request to the adapter for the external service. 先查本地,不在本地时再查外部.

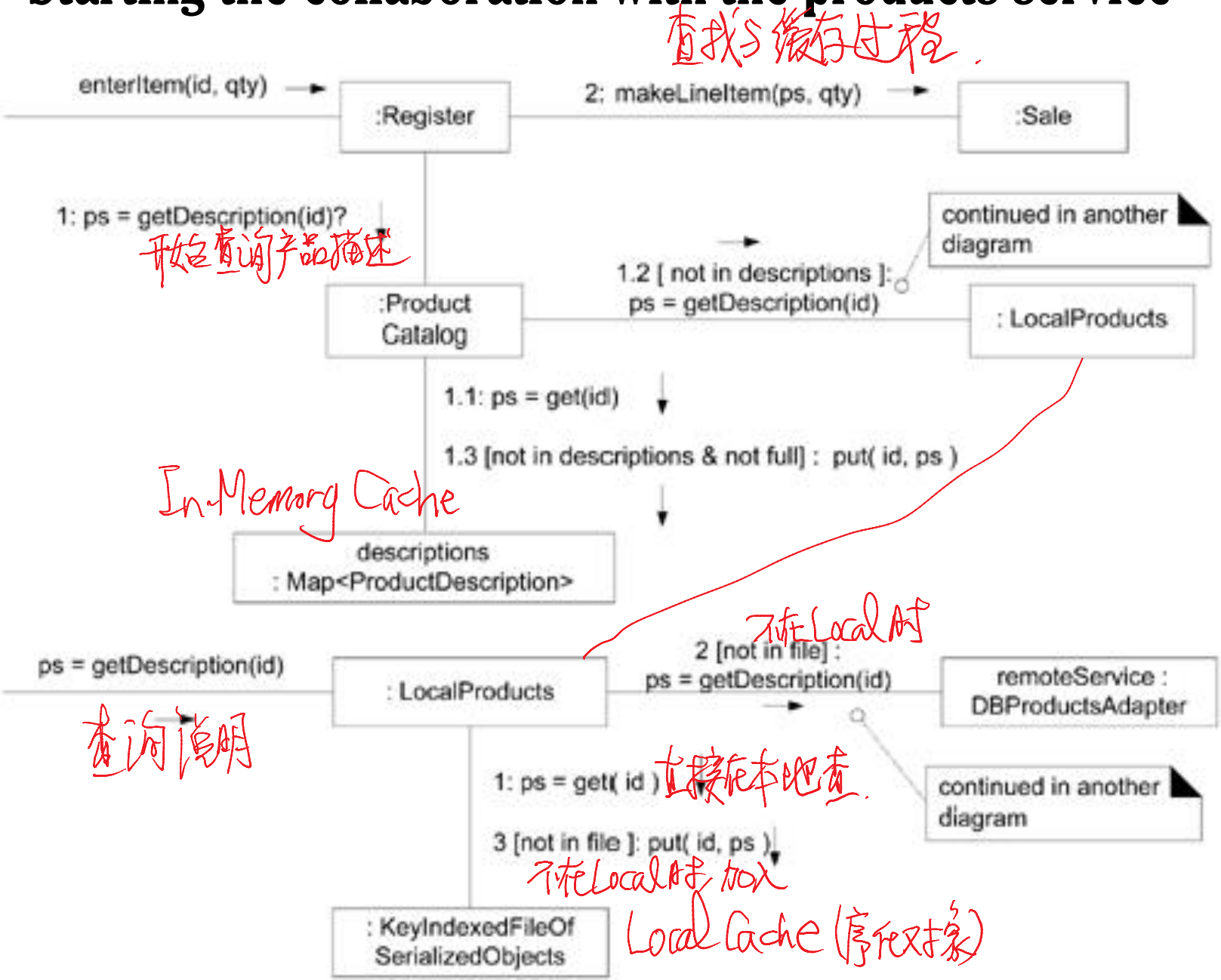
Adapters for Product Information



Initialization of the Product Information Service



Starting the collaboration with the products service



缓冲的策略（回想上次课提到）

Caching Strategies

- The alternatives for loading the in-memory *ProductCatalog* cache and the *LocalProducts* file-based cache:
  - **Lazy initialization:** Fill the in-memory cache and LocalProducts file as external product info is retrieved
  - **Eager initialization** loads them during start-up
- If the designer is unsure which approach to use and wants to experiment with alternatives, a family of different *CacheStrategy* objects based on the Strategy pattern can neatly solve the problem.

可以用策略模式创建多个对象来测试哪一种更好。

Caching and Threads

Cache陈腐的更新方法.

- Cache can go stale quickly; it should be refreshed occasionally if possible. Prices can change quickly.
- If the *LocalProducts* object is going to solve the stale cache problem with a query for updates every n minutes, one approach to the design is to make it an **active object** that owns a thread of control.

使用主动对象的线程控制更新Cache.

- The thread will sleep for n minutes, wake up, the object will get the data, and the thread will go back to sleep.

每n分钟后更新装入数据.

Dealing With Failure

- In the example, suppose an item is not in the local cache and the external service is offline.
- You could ask for a description and price, or cancel the entry.

Terminology

- **Fault:** The ultimate origin or cause of misbehavior 缺陷(根源)
- **Error:** Manifestation of the fault in the running system 错误：缺陷导致运行时的表现。
- **Failure:** Denial of services caused by an error 失败：由于错误导致的服务拒绝。

Throwing Exceptions

- A straightforward approach to signaling the failure under consideration is to throw an exception.
- Exceptions are appropriate when dealing with resource failures and other external services
- An exception will be thrown from within the persistence subsystem, where a failure to use the external products database is first detected. The exception will unwind the call stack back up to an appropriate point for its handling.
- Should an *Exception* be thrown all the way up to the presentation layer? No. It is at the wrong level of abstraction.

Pattern: Convert Exception

- Avoid showing lower-level exceptions coming from lower-level services 不要直接显示低层异常
- Convert a lower-level exception to one that is meaningful at the level of the subsystem 转换低层异常为高层语义
- Higher level exception usually wraps the lower-level exception and adds information to make it meaningful to the higher level 高层异常会附加更有意义的语义
- No one but a programmer wants to see a stack trace
- Also known as Exception Abstraction 也称为异常抽象模式

Pattern: Name the Problem

异常命名模式：异常中使用出现的问题命名异常.

- Assign a name that describes why the exception is being thrown, not the thrower.
- For example, “File not found” or “Duplicate key”
- The benefit is that it makes it easier for the programmer to understand the problem, and it highlights the essential similarity of many classes of exceptions (in a way that naming the thrower does not). 便于理解与实现.

Exceptions in UML

- UML allows the operation syntax to be in any language, such as Java:
  `Object get(key, class) throws DBUnavailableException, FatalException`
  用Java句法表示
- Default syntax allows exceptions to be defined in a property string
  用特性串表示
- *Exception* is a specialization of *Signal*, which is the specification of asynchronous communication between objects.
  看为异步消息.
- Exception handling in UML is rarely used.

Exceptions in UML

PersistenceFacade
usageStatistics : Map
Object get(Key, Class) throws DBUnavailableException, FatalException Put(Key, Object) {Exceptions=(DBUnavailableException, FatalException)}
Exceptions FatalException DBUnavailableException

Error Handling Pattern: Centralized Logging

- Use a Singleton-accessed central error logging object and report all exceptions to it.
- In a distributed system, each local singleton will collaborate with a central error logger
- Benefits:
  - Gives consistency in reporting
  - Flexible definition of output streams and formats
- Also known as Diagnostic Logger

Pattern: Error Dialog 错误会话模式

- Standard Singleton-accessed application-independent, *non-UI object* to notify users of errors. 应用程序独立的，非UI的特性化单子做为错误通知。
- It wraps one or more UI "dialog" objects and delegates notification to them. 封装多个UI对话框。
- This lets you generalize and output errors as a modal dialog, sound, calling a cell phone, etc. 这是一个一般性的机制，可以将错误输出到各种对象中。

Benefits of Error Dialog

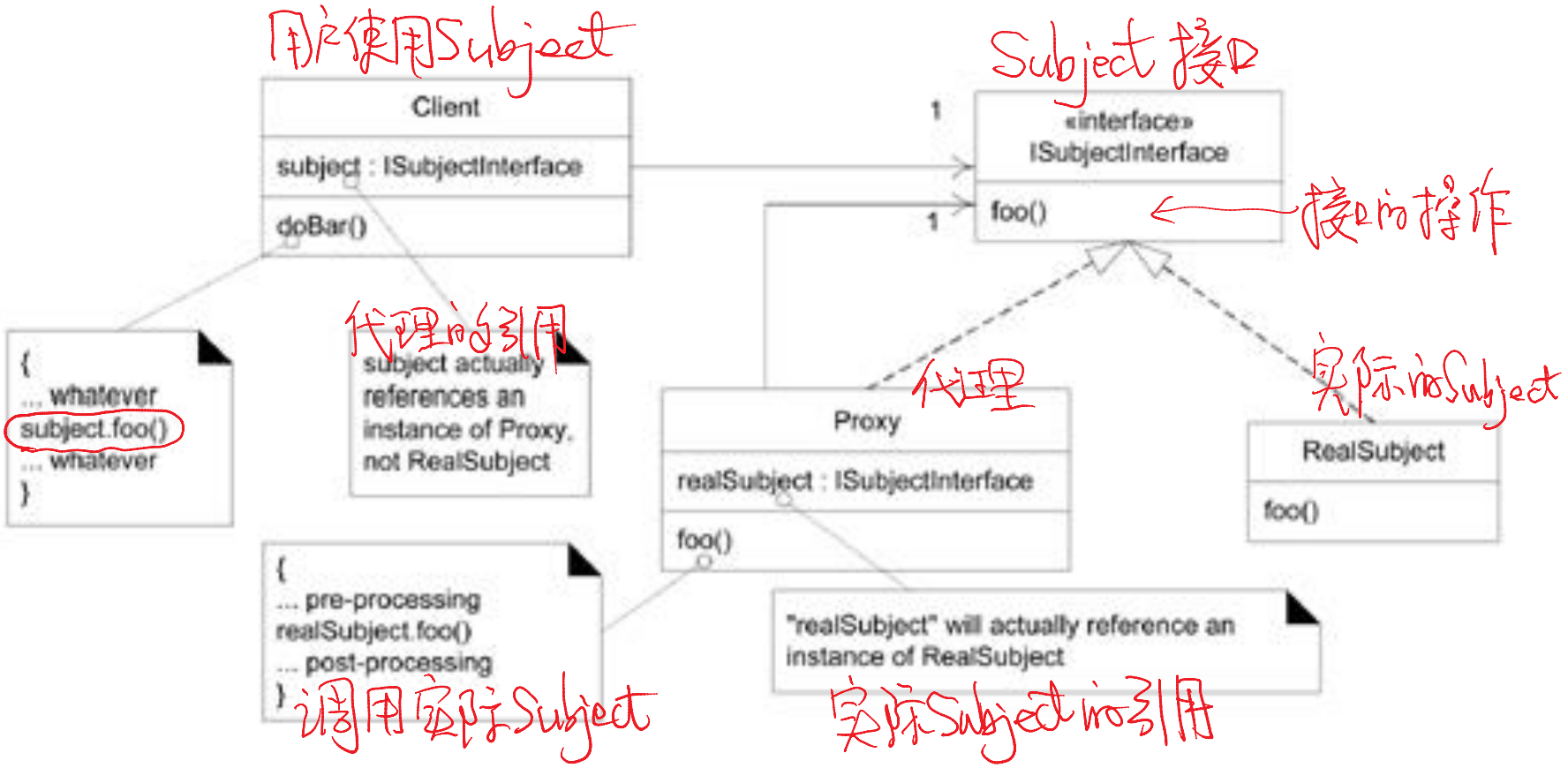
- Protected Variations with respect to output mechanism 封装了各种变异情况
- Consistent style of reporting 一致性
- Centralized control of common strategy for error notification 集中式的错误通知的通用策略
- Minor performance gain 最小的性能影响.
- In simple programs, you probably don't need to use this 简单程序中可能不需要

代理模式

Proxy (GoF)

- **Context/ Problem:**
  Direct access to a real subject object is not possible
- **Solution:**
  Add a level of indirection, as described below
- Proxy is an object that implements the same interface as the subject object, holds a reference to the real subject, and controls access to it

General Structure of the Proxy Pattern



Failover with a Proxy in NextGen Accounting

- This NextGen example use of Proxy is not the Remote Proxy variant, but rather the **Redirection Proxy** (also known as a **Failover Proxy**) variant.是一个转向代理.
  1. Send a postSale message to the proxy, as though it were the actual accounting interface 先调用Proxy.
  2. If the proxy cannot connect to the service (via its adapter) it redirects the message to a local service, which stores it locally until the remote service is available Proxy先调用实际的服务,但失败后转向本地.
     当实际服务可用后,再将记账转向实际服务.

NextGen Use of a Redirection Proxy