6 Elaboration Iteration 2 More Patterns

庞雄文
Tel：18620638848
Wechat: augepang
QQ: 443121909

Contents

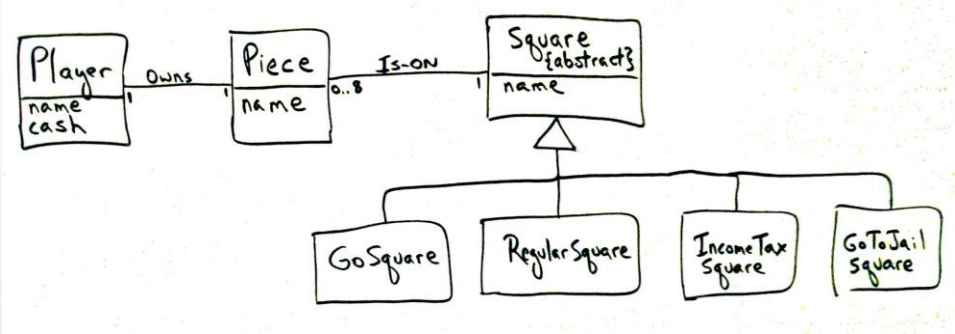
- Part 4: Elaboration Iteration 2 More Patterns
  - Quick Analysis Update
  - GRASP: More Objects with Responsibilities
  - Applying GoF Design Patterns

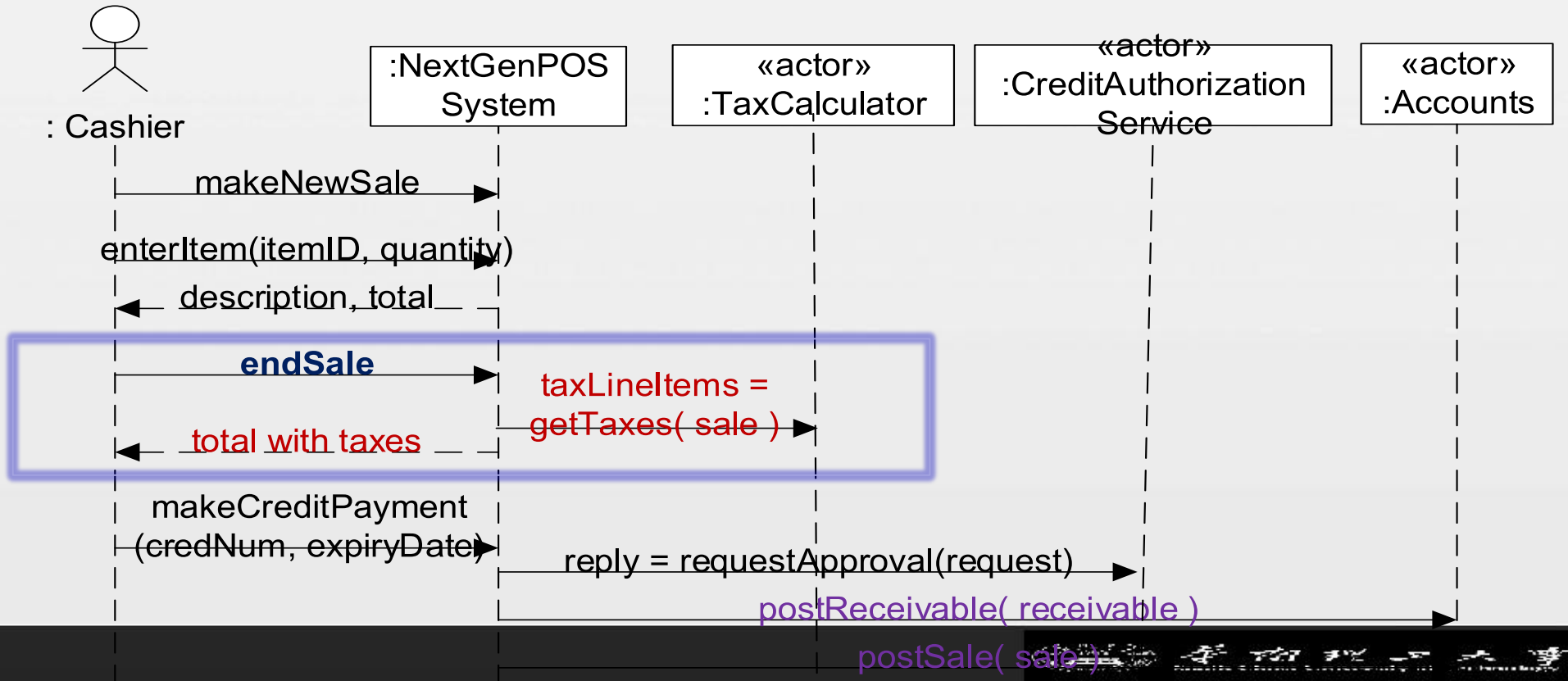
Iteration 2 More Patterns

- **From Iteration 1 to 2**
- **Iteration-2 Requirements and Emphasis**
  - **Object Design and Patterns**
  - **POS**
    - Support for variations in third-party external services
    - Complex pricing rules.
      - Employee20% off.
      - Preferred Customer10% off.
    - A design to refresh a GUI window when the sale total changes
  - **Monogame**
    - implement a basic, key scenario of the Play Monopoly Game
      moving around the squares of the board
    - Each player receives $1500 at the beginning of the game
    - When a player lands on the Go square, the player receives $2
    - When a player lands on the Go-To-Jail square, they move to t

Quick Analysis Update

- Each iteration, we should quickly update :
  - Requirement
    - Use case
  - Analysis model
    - Domain model, SSD and OC





GRASP: More Objects with Responsibilities

- **Polymorphism**
  - **Problem**
    - **How handle alternatives based on type?**
      - if-then-else means variation, but non-extensible when new variations arise
      - Example, square in the monogame
    - **How to create pluggable software components?**
      - client-server relationship: how can you replace one server component with another without affecting the client?
  - **Solution**
    - When related alternatives or behaviors vary by type, assign responsibility for the behavior – using polymorphic operations – to types for which the behavior varies
    - **Do not test for the type of an object and use conditional logic to perform varying alternatives based on type**

GRASP: More Objects with Responsib

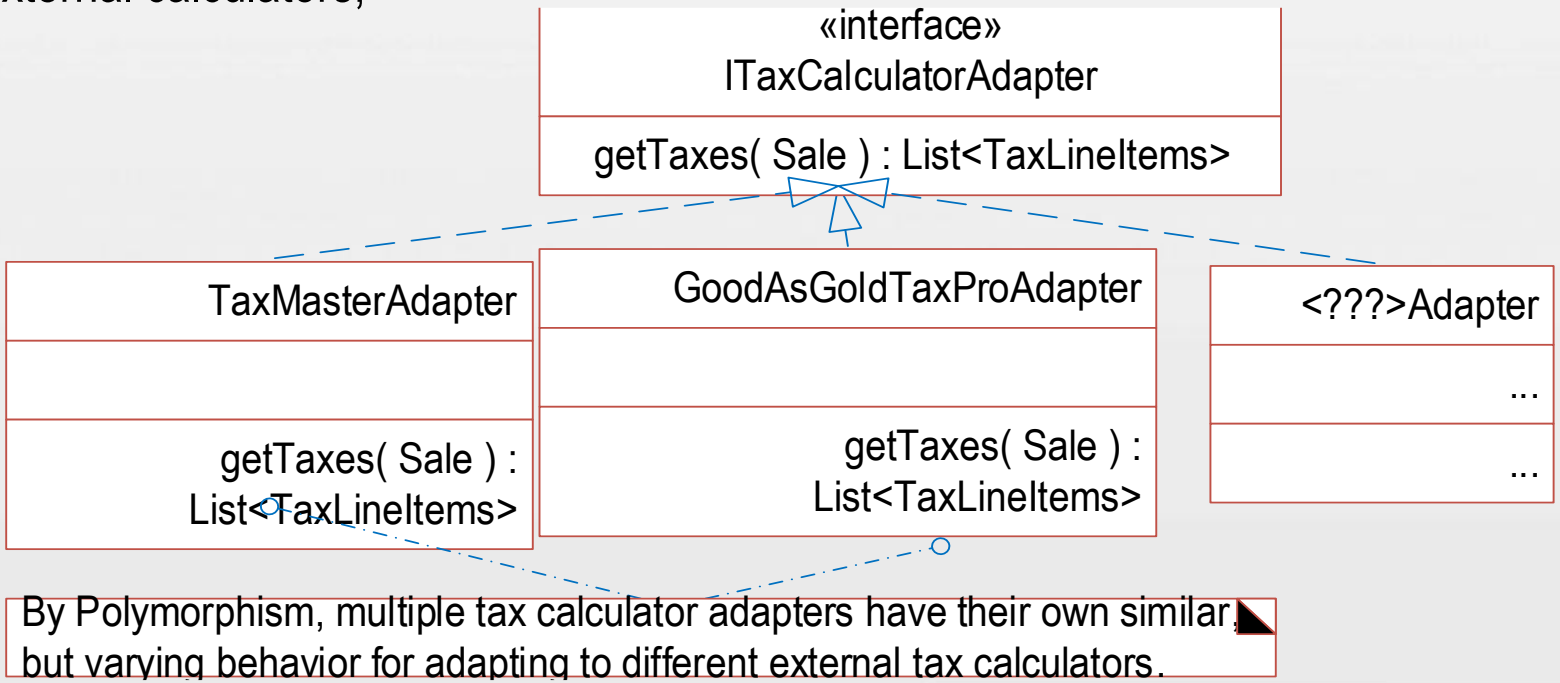
- **Polymorphism -- NextGen Problem**
  - **How Support Third-Party Tax Calculators?**
    - when you purchase, you will pay a sales tax on top of the list
    - sales tax varies by state and item category

State federal district or territory	Base sales tax	Total with max local surtax	Groceries	Prepared food	Prescription drug	Non-prescription drug	Clothing	Intangibles
Alabama	4%	13.5%						
Alaska	0%	7%						
Arizona	5.6%	10.725%						
Arkansas	6.5%	11.625%	0.125%+					
California	7.25%	10.5%						
Colorado	2.9%	10%						
Connecticut	6.35%	6.35%						1%
Delaware	0%	0%						
District of Columbia	5.75%	5.75%		10%				
Florida	6%	7.5%		9% (max)				
Georgia	4%	8%	4% (max)[39]					
Guam	4%	4%						
Hawaii	4.166%	4.712%						
Idaho	6%	8.5%			[40]			
Illinois	6.25%	10.25%	1%+	8.25%+	1%+	1%+		
Indiana	7%	7%		9% (max)				
Iowa[41]	6%	7%						
Kansas	6.5%	11.5%						

GRASP: More Objects with Responsibilities

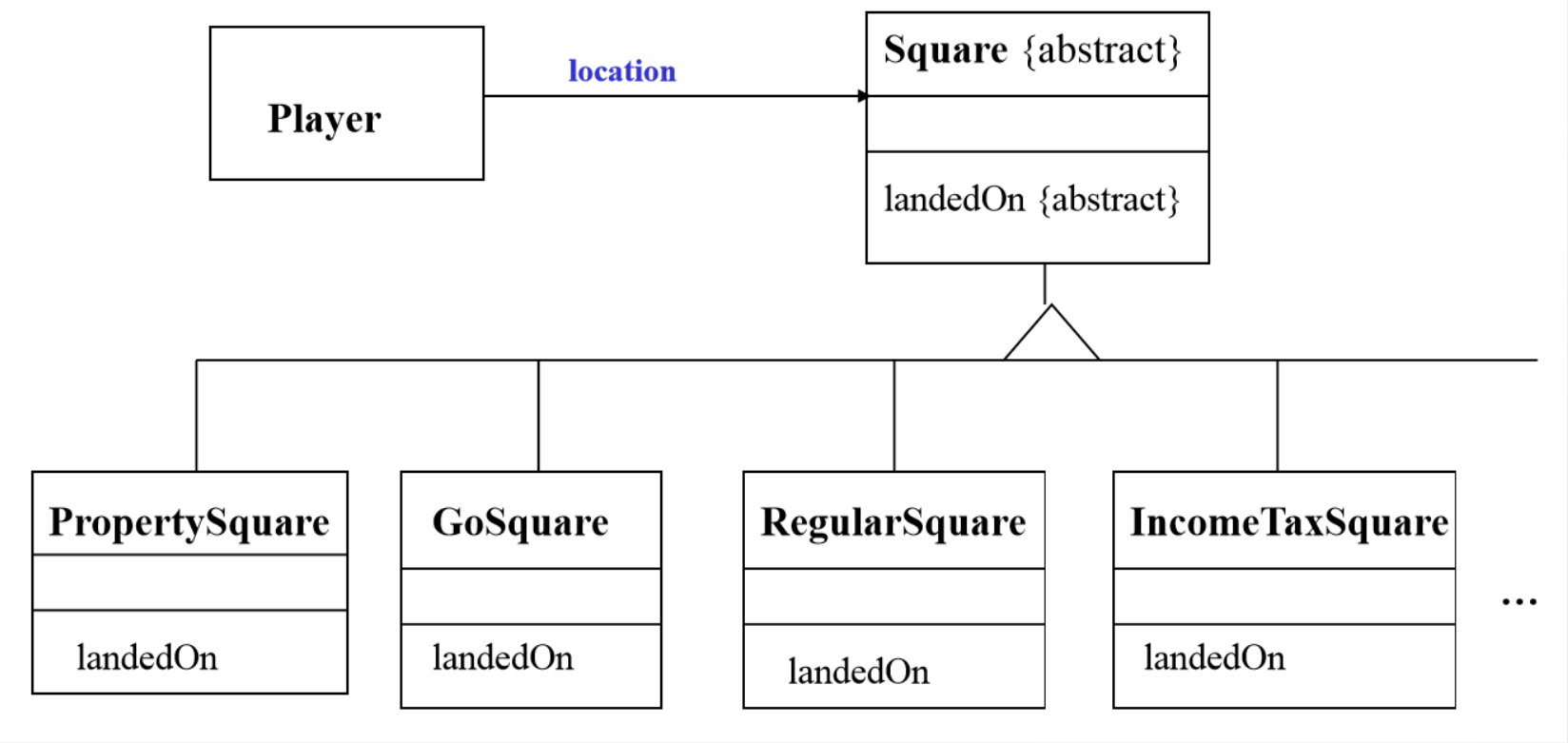
Polymorphism -- NextGen Problem

- What objects should be responsible for handling these varying external tax calculator interfaces?
  - by Polymorphism we should assign the responsibility for adaptation to different calculator
  - These calculator adapter objects are local software objects that represent the external calculators,



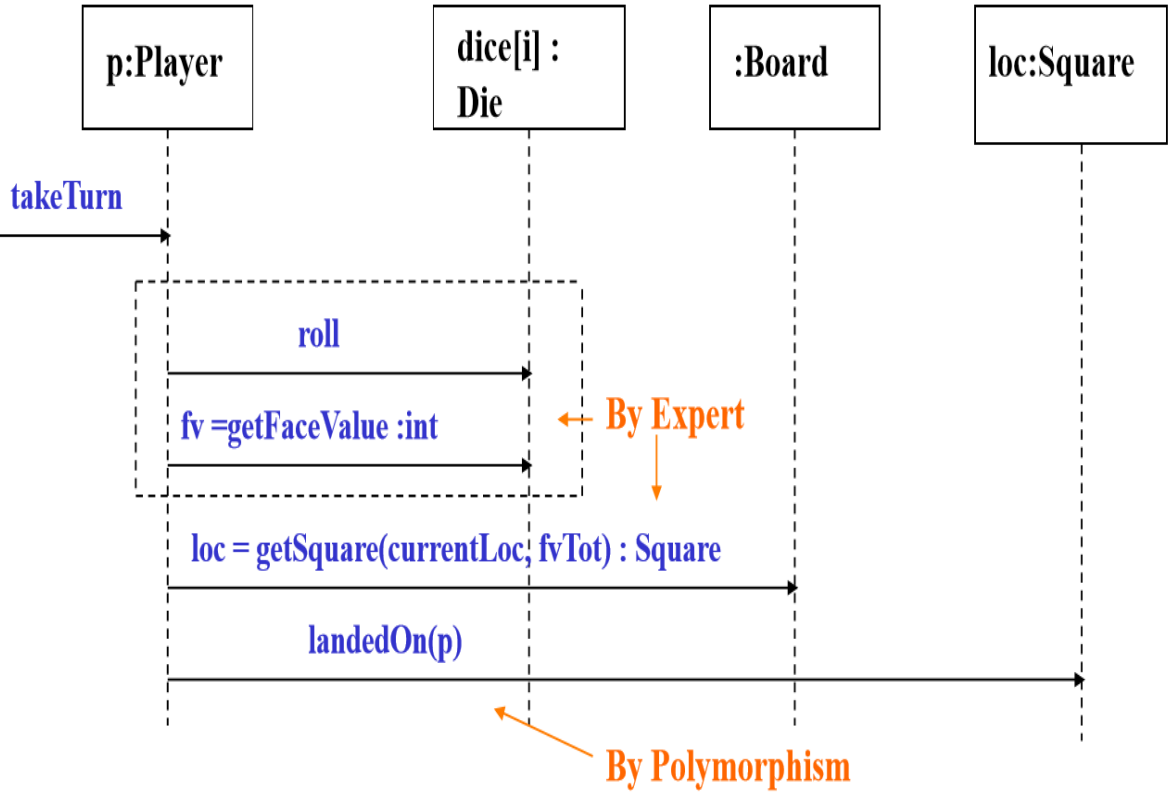
GRASP: More Objects with Responsibilities

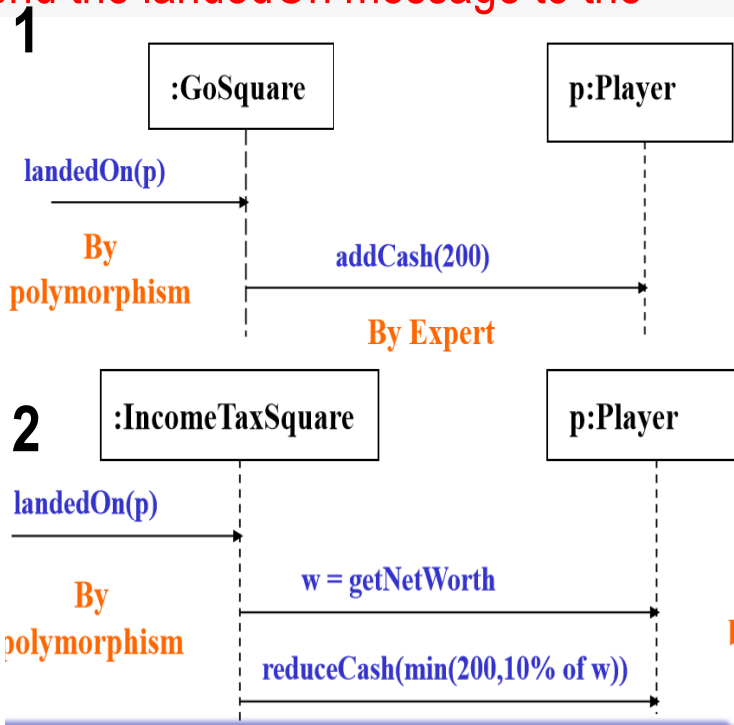
- **Polymorphism -- Monogame Problem**
  - **How to Design for Different Square Actions?**
    - when a player lands on different square, there will be different rules. **It varies for the types (classes)**
    - By polymorphism.

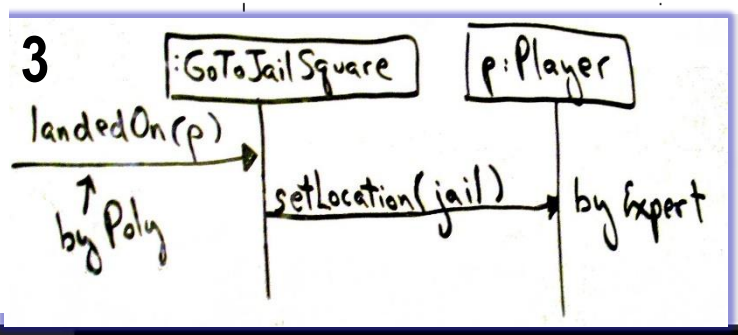


GRASP: More Objects with Responsibilities

- **Polymorphism -- Monogame Problem**
  - interaction diagrams evolve? What object should send the landedOn message to the square that a player lands on?
    - **by Low Coupling and by Expert, Player**







GRASP: More Objects with Responsibilities

- **Polymorphism – discuss**
  - **Interfaces or superclasses:**
    - Guideline: use interfaces when you want polymorphism without committing to a particular class hierarchy.
    - Liskov substitution principle – a value can be replaced by a subtype without changing important properties of program
  - **Future-proofing:**
    - if variability at a particular point is very probably, then expend the effort to accommodate flexibility.
    - •Avoid adding flexibility just because it is possible.
  - **Related Patterns**
    - Protected Variations
    - GoF design patterns rely on polymorphism, including Adapter, Command, Composite, Proxy, State, and Strategy.

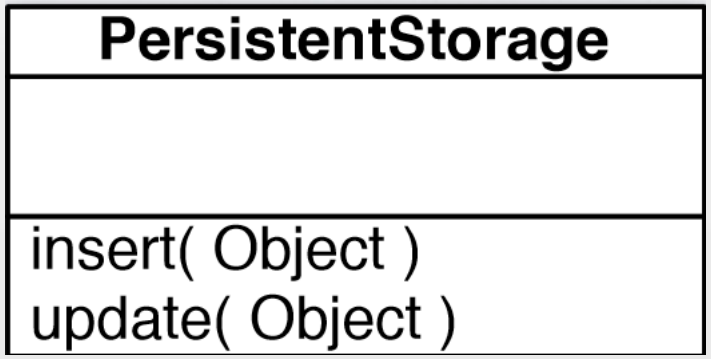
GRASP: More Objects with Responsibilities

- **PURE FABRICATION**
  - **Problem**
    - LRP. BUR assigning responsibility to domain layer software classes leads to problems in terms of poor cohesion or coupling, or low reuse potential
    - What objects should have the responsibility, when you do not want to violate High Cohesion and Low Coupling, or other goals, but solutions offered by Information Expert (for example) are not appropriate?
  - **Solution**
    - Assign a highly cohesive set of responsibilities to a convenience class, not representing a problem domain concept
    - Fabrication – made up
    - Pure – keep it clean: high cohesion, low coupling
    - "Pure fabrication" – English idiom that implies making something up.

    - **Most classes not appearing in the domain model will be pure fabrications**

GRASP: More Objects with Responsibilities

- **PURE FABRICATION -- Examples**
  - **NextGen Problem: Saving a Sale Object in a Database**
    - Need to save Sale instances in a relation database
    - Information Expert says assign functionality to Sale.
    - Implications:
      - Task needs large number of supporting database-oriented operations, none related to the concept of a Sale. Incohesion!
      - Sale becomes coupled to data base interface, so coupling goes up.
      - Saving objects in a database is a general task – many classes will need it
    - **Solution**



Understandable concept. Pure software concept. Not in domain model.

Sale unaffected
Cohesive concept.
Generic and reusable

GRASP: More Objects with Responsibilities

PURE FABRICATION -- Discussion

- **Design of objects can be broadly divided into two categories:**
  - Representational decomposition. e.g., Sale
  - Behavioral decomposition. e.g., Table Of Contents Generator.
- Pure Fabrications are often the result of behavioral decomposition.
- Often highly cohesive with high reuse potential
- **Avoid overuse**
  - functions and algorithms generally should not be represented by objects.

Related Patterns and Principles

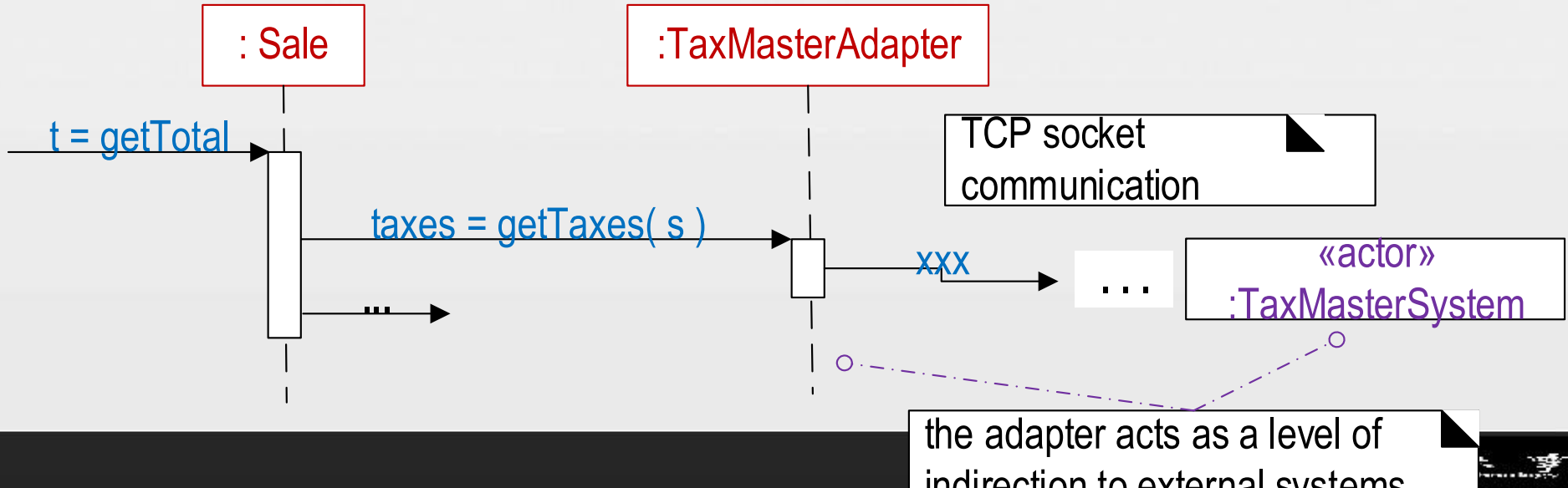
- Low Coupling、High Cohesion.
- GoF design patterns, such as Adapter, Command, Strategy, and so on, are Pure Fabrications
- all other design patterns are Pure Fabrications

GRASP: More Objects with Responsibilities

- **Indirection**
  - **Problems**
    - How to assign responsibility to avoid direct coupling between two (or more) things?
    - How to decouple objects so that low coupling is supported and reuse potential remains higher?
  - **Solutions:**
    - Assign the responsibility to an intermediate object to mediate between other components or services so that they are not directly coupled
    - The intermediary creates indirection between the other components
    - The intermediary is likely to be a pure fabrication.

GRASP: More Objects with Responsibilities

- **Indirection—EXAMPLE**
  - **TaxCalculatorAdapter**
    - The adaptor acts as a level of indirection to external systems
    - Via polymorphism, provide a consistent interface to the inner objects and hide the variations in the external APIs
    - By indirection, the adapter objects protect the inner design against variations in the external interfaces
    - **Most problems in computer science can be solved by another level of indirection**



GRASP: More Objects with Responsibilities

- **Protected Variations**
  - **Problem:**
    - How to design objects, subsystems, and systems so that the variations or instability in these elements does not have an undesirable impact on other elements?
  - **Solutions**
    - Identify points of predicted variation or instability
    - Assign responsibilities to create a stable interface around them
    - "Interface" in broadest sense – not just Java interface.

  - The ITaxCalculatorAdaptor interface (from Polymorphism) allows for future tax calculators that may not yet have been thought of.

  - Information hiding!!! Open-close principle

GRASP: More Objects with Responsibilities

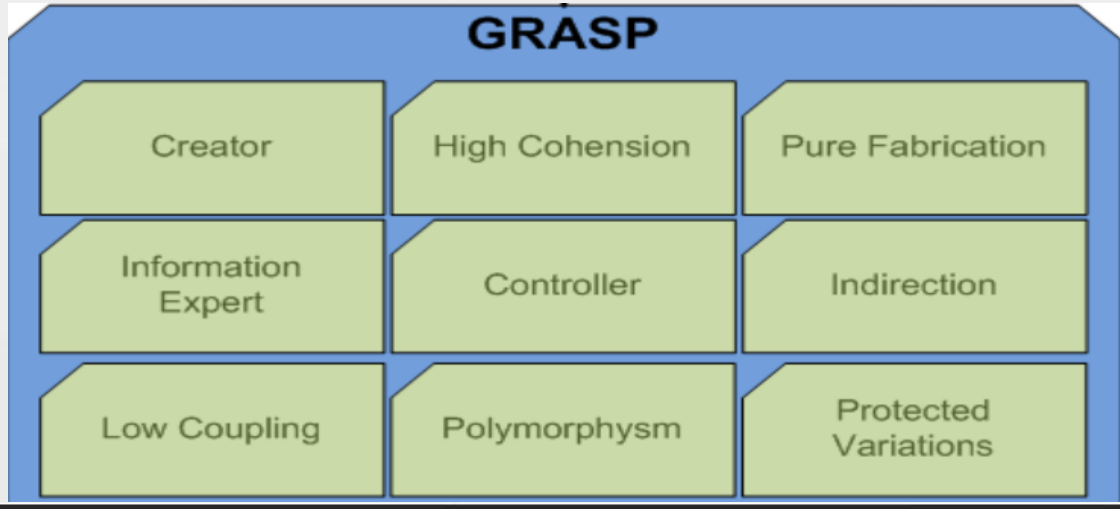
- Protected Variations
  - OTHER APPROACHES TO PROTECTED VARIATIONS
    - Core protected variation mechanisms: data encapsulation, interfaces, polymorphism, standards, virtual machines, operating systems.
    - **Data-Driven Designs**
    - Service lookup: clients look up server with stable interface via technology such as Java JINI or UDDI for Web services.
    - **Interpreter-Driven Designs**
    - **Reflective or Meta-Level Designs**
    - **Uniform Access**
    - **Standard Languages**
    - **The Liskov Substitution Principle (LSP)**
    - **Structure-Hiding Designs**
      - Law of Demeter: objects never talk to objects they are not directly connected to.

```
public void doX() {
 F someF = foo.getA().getB().getC().getD().getE().getF(); // … }
```

GRASP: More Objects with Responsibilities

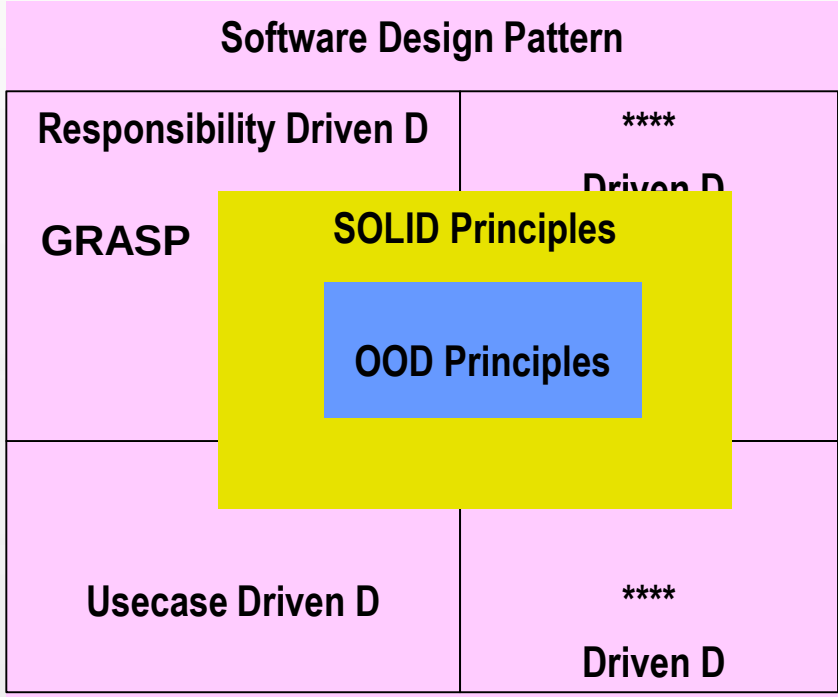
GRASP Summary

- Low coupling: How to support low dependency, low change impact, and increased reuse?
- High cohesion: How to keep objects focused, understandable, and manageable, and as a side effect, support Low Coupling?
- Creator: Who creates the Square object?
- Information expert : What is a general principle of assigning responsibilities to objects?
- Controller: What first object beyond the UI layer receives and coordinates ("controls") a system operation?
- Polymorphism
- Indirection
- Pure fabrication
- Protected variations



GRASP: More Objects with Responsibilities

- OO Design Summary
  - OOD Principles?
  - SOLID Principle
  - Software Design
    - RDD--**Responsibility Driven Design**
      - GRASP
    - **DDD—Domain Driven Design**
    - **UDD- Use case Drive Design**
    - ***** DD-- ** Driven Design**



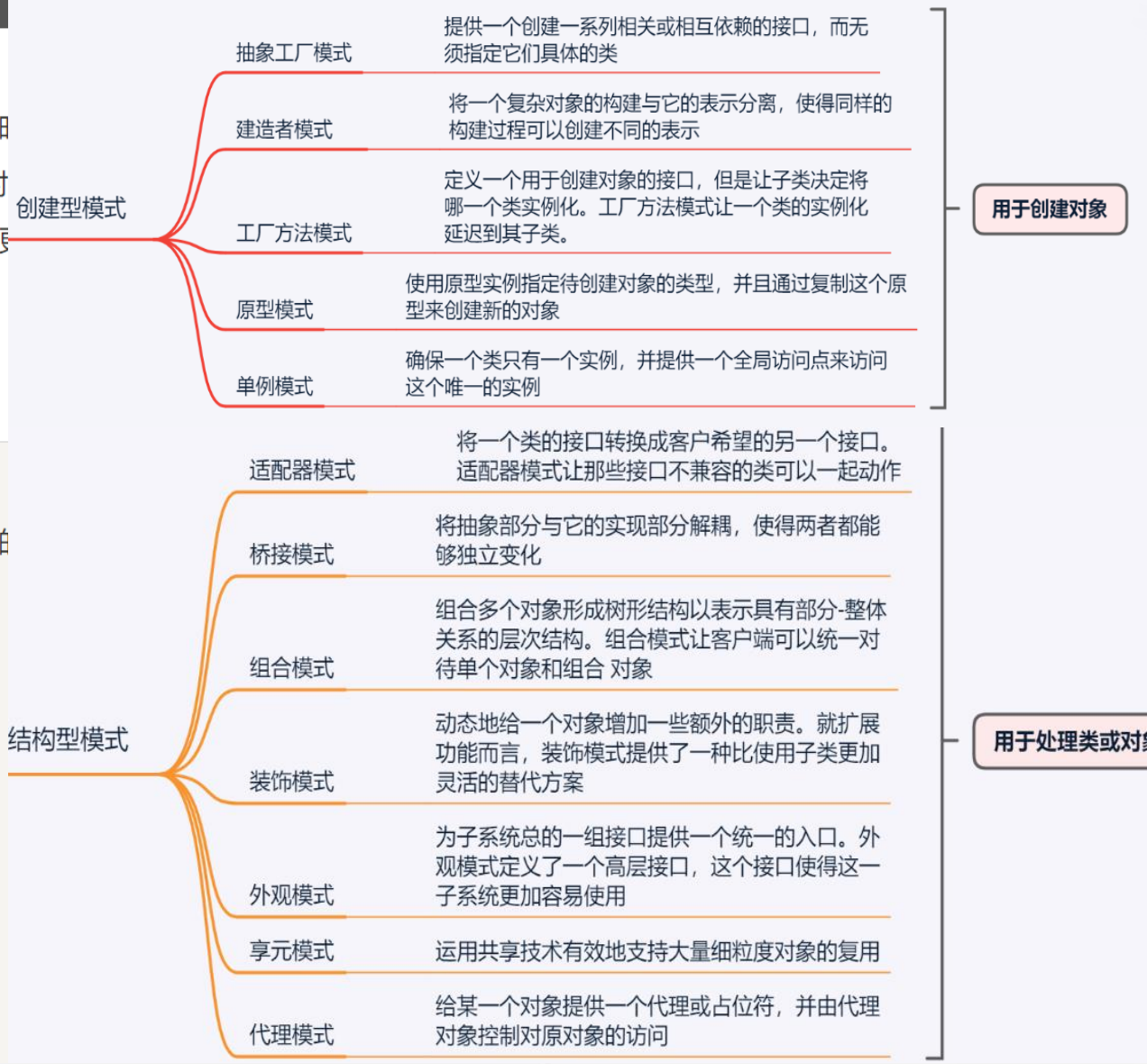
Applying GoF Design Patterns

THE 23 GANG OF FOUR DESIGN PATTERNS

C	Abstract Factory
S	Adapter
S	Bridge
C	Builder
B	Chain of Responsibility
B	Command
S	Composite
S	Decorator
S	Facade
C	Factory Method
S	Flyweight
B	Interpreter
B	Iterator
B	Mediator
B	Memento
C	Prototype
S	Proxy
B	Observer
C	Singleton
B	State
B	Strategy
B	Template Method
B	Visitor

Applying GoF Design Patterns

序号	模式 & 描述
1	**创建型模式** 这些设计模式提供了一种在创建对象的同时 式，而不是使用 new 运算符直接实例化对 断针对某个给定实例需要创建哪些对象时更
2	**结构型模式** 这些设计模式关注类和对象的组合。继承的 和定义组合对象获得新功能的方式。



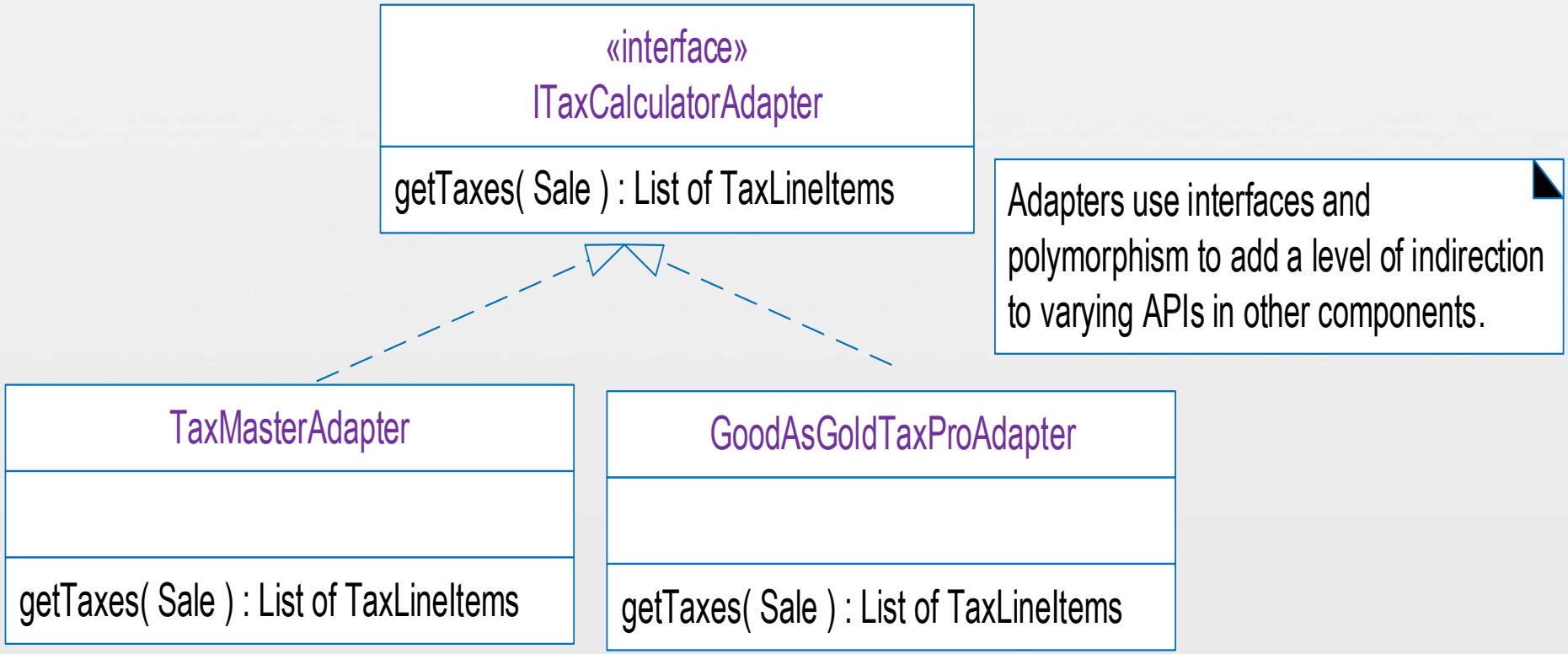
Applying GoF Design Patterns

3	**行为型模式** 这些设计模式特别关注对象之间的

Applying GoF Design Patterns

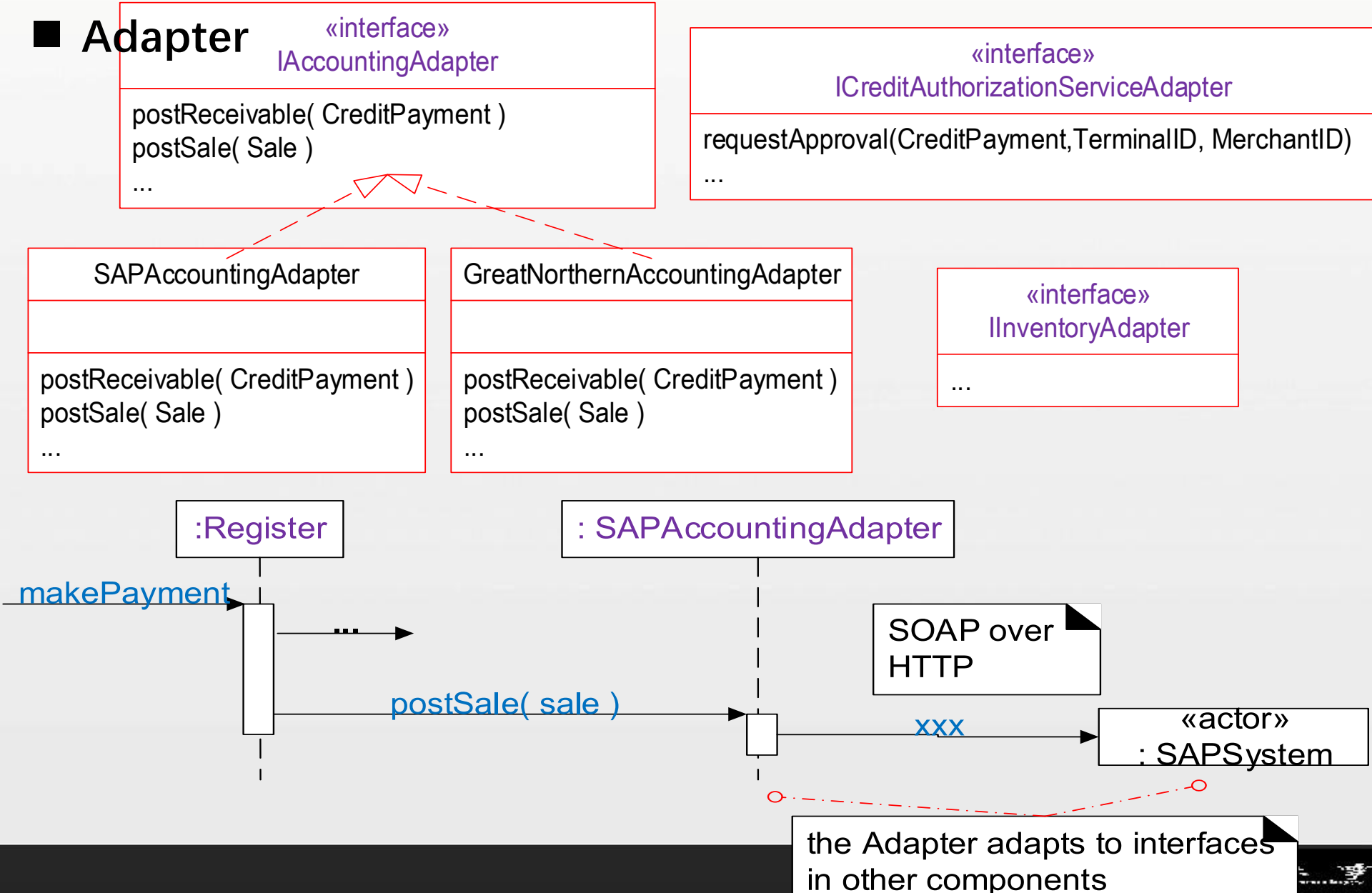
■ Adapter

Problem:	How to resolve incompatible interfaces, or provide a stable interface to similar components with different interfaces?
Solution: (advice)	Convert the original interface of a component into another interface, through an intermediate adapter object.



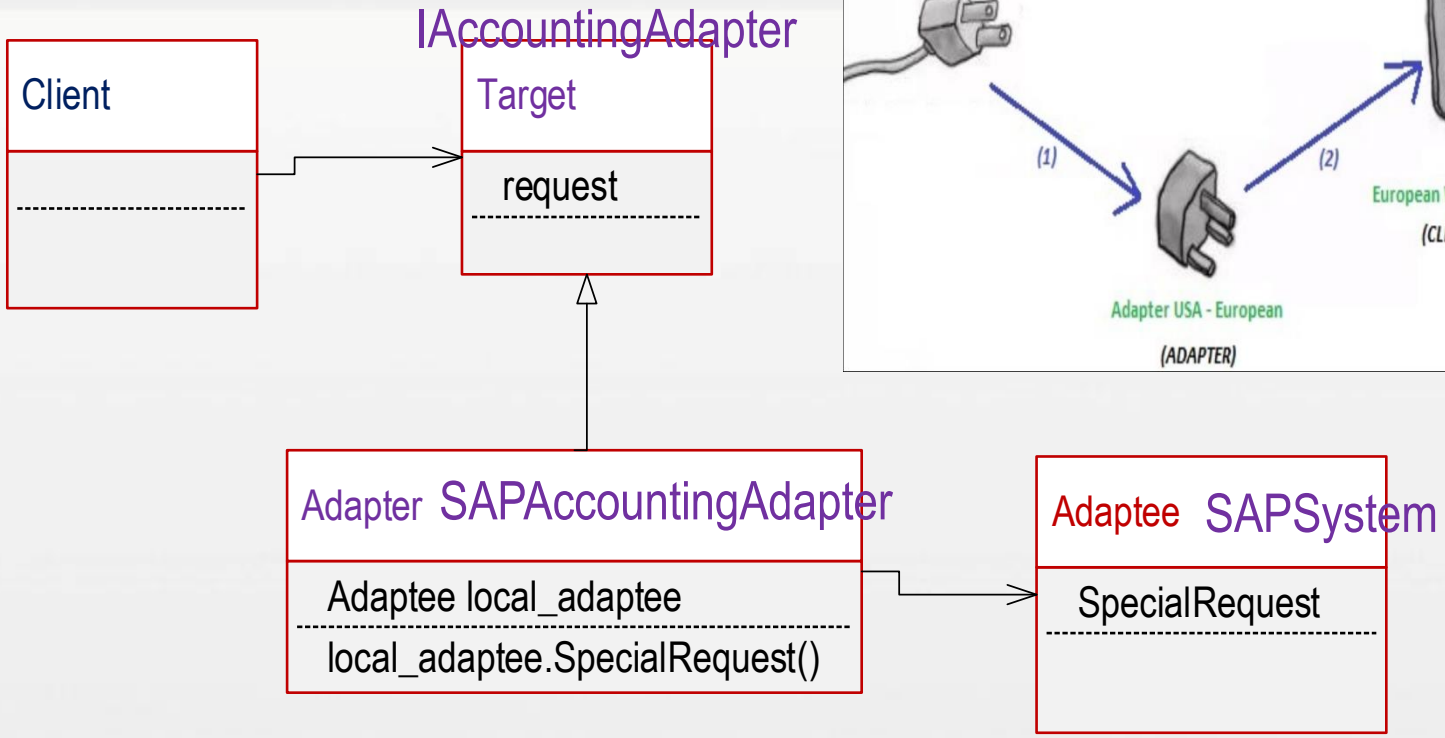
Applying GoF Design Patterns

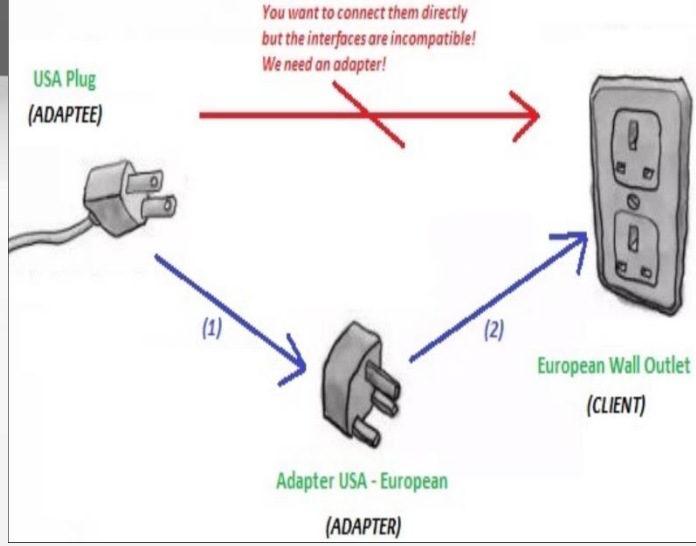
- **Adapter**



Applying GoF Design Patterns

- **Adapter**





- The classes/objects participating in adapter pattern:X
  - **Target** - defines the domain-specific interface that Client uses.
  - **Adapter** - adapts the interface Adaptee to the Target interface.
  - **Adaptee** - defines an existing interface that needs adapting.
  - **Client** - collaborates with objects conforming to the Target interface.

Applying GoF Design Patterns

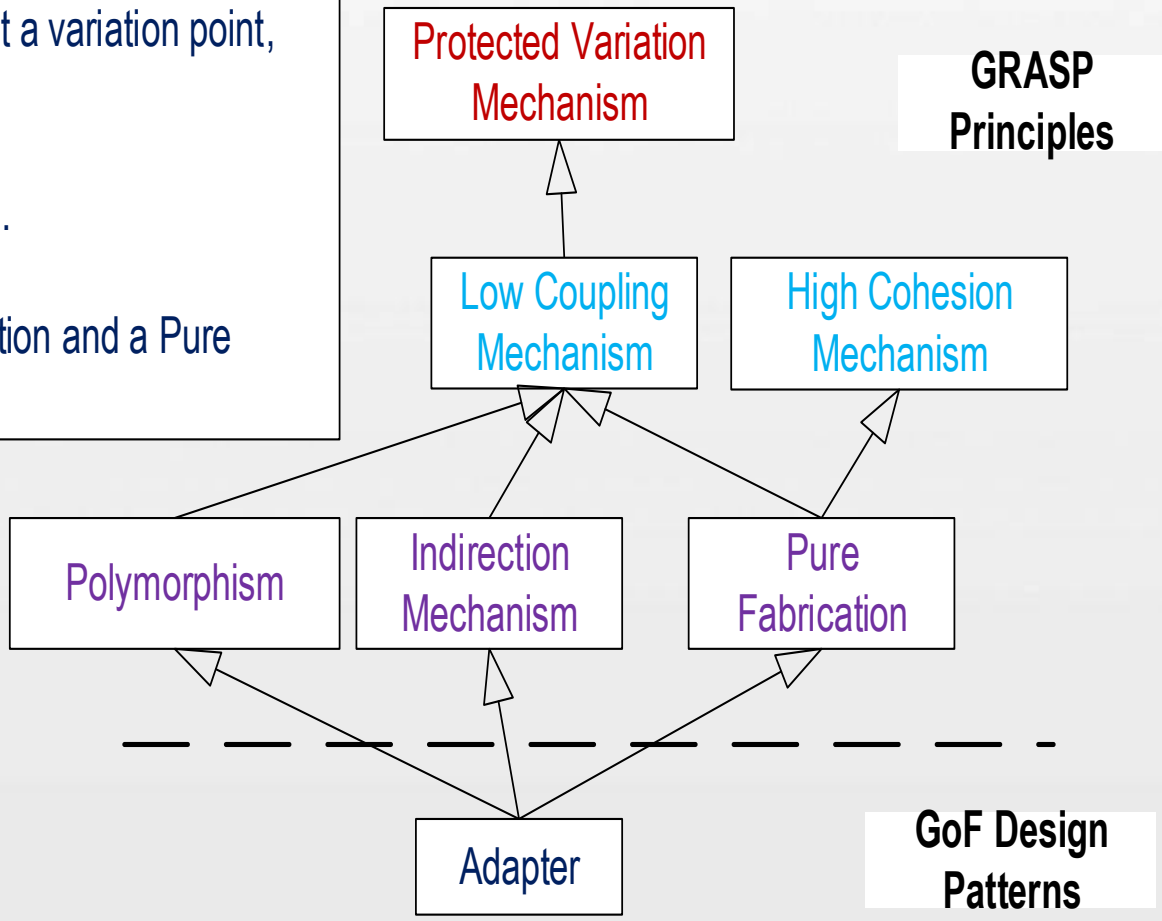
■ GRASP Principles and Design Pattern

Low coupling is a way to achieve protection at a variation point.

Polymorphism is a way to achieve protection at a variation point, and a way to achieve low coupling.

An indirection is a way to achieve low coupling.

The Adapter design pattern is a kind of Indirection and a Pure Fabrication, that uses Polymorphism.



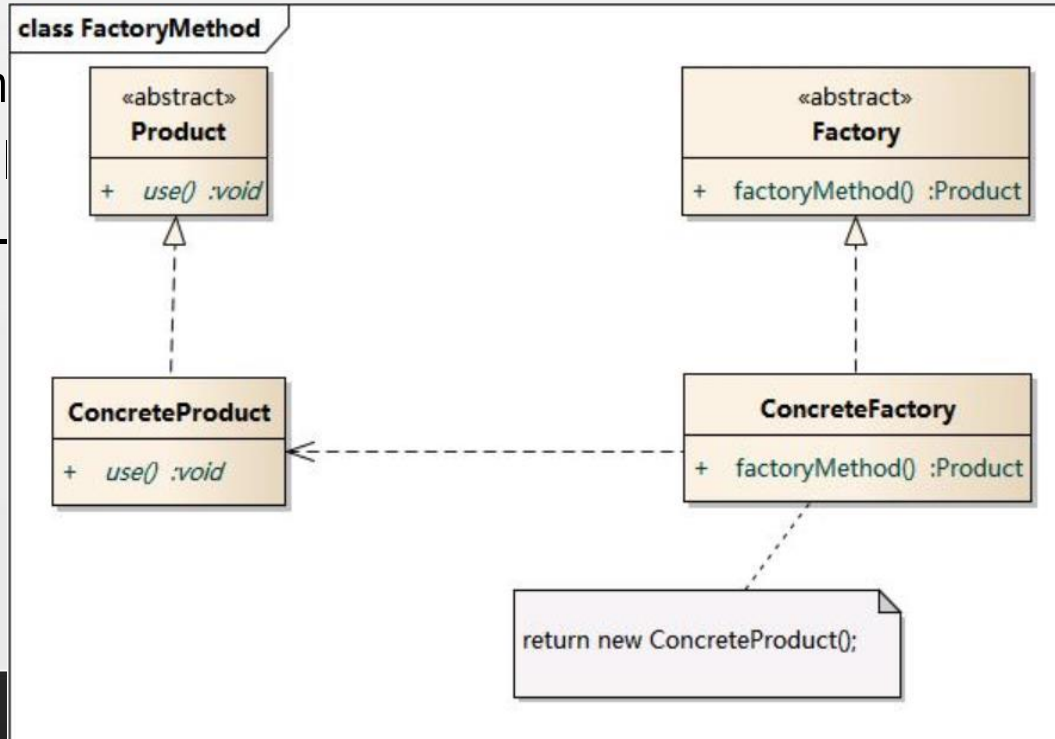
Applying GoF Design Patterns

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?
Solution: (advice)	Create a Pure Fabrication object called a Factory that handles the creation.

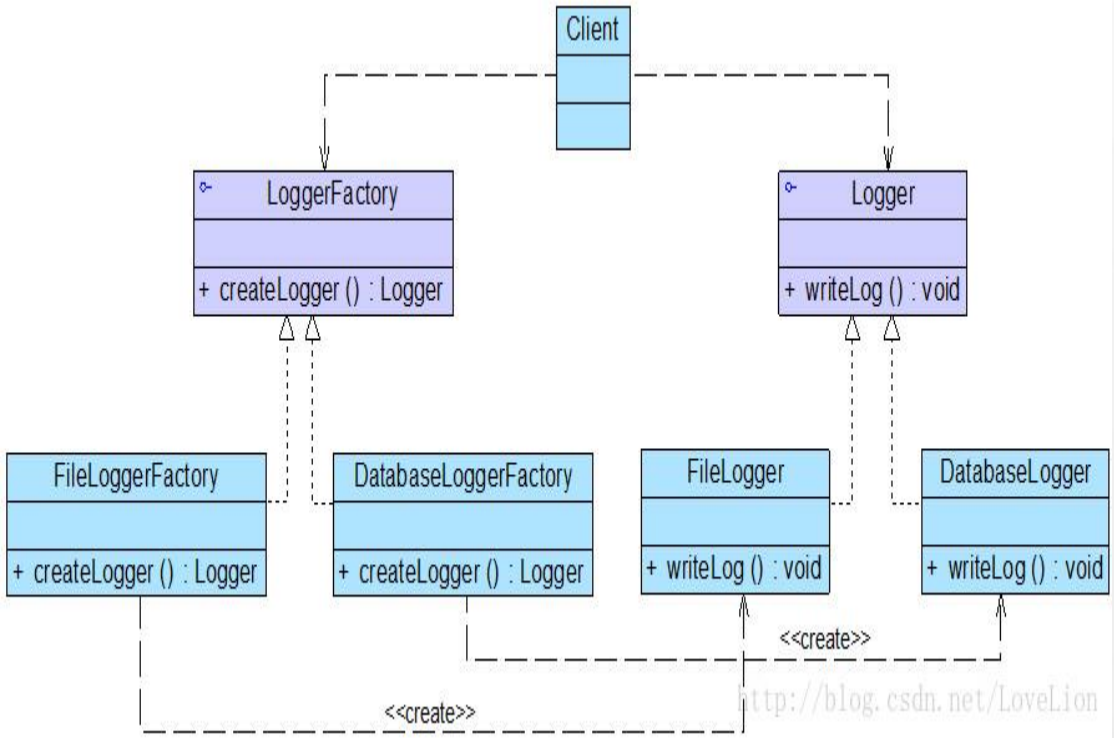
advantages:

- Separate the responsibility of com
- Hide potentially complex creation
- Allow introduction of performance-object caching or recycling.



Applying GoF Design Patterns

■ Factory



Logger接口充当抽象产品，其子类FileLogger和DatabaseLogger充当具体产品

LoggerFactory接口充当抽象工厂，其子类FileLoggerFactory和DatabaseLoggerFactory充当具体工厂

```
class Client {
    public static void main(String args[]) {
        LoggerFactory factory;
        Logger logger;
        factory = new FileLoggerFactory(); //可引入配置文件实现
        logger = factory.createLogger();
        logger.writeLog();
    }
}
```

还可以进一步隐藏工厂类的工厂方法，简化客户端的使用(将客户类方法集成到工厂类中)

```
//改为抽象类
abstract class LoggerFactory {
    //在工厂类中直接调用日志记录器类的业务方法writeLog()
    public void writeLog() {
        Logger logger = this.createLogger();
        logger.writeLog();
    }

    public abstract Logger createLogger();
}
```

JAVA的反射机制可通过类名获得类的实例。可将具体工厂名称在配置文件中，通过对配置文件的解析和的JAVA反射机制获得工厂类的实例(假设为XMLUtil.getBean)

```
//通过类名生成实例对象并将其返回
Class c=Class.forName("String");
Object obj=c.newInstance();
return obj;
```

```
LoggerFactory factory;
Logger logger;
factory = (LoggerFactory)XMLUtil.getBean();
logger = factory.createLogger();
logger.writeLog();
```

Applying GoF Design Patterns

Factory

ServicesFactory
accountingAdapter : IAccountingAdapter inventoryAdapter : IInventoryAdapter taxCalculatorAdapter : ITaxCalculatorAdapter
getAccountingAdapter() : IAccountingAdapter getInventoryAdapter() : IInventoryAdapter getTaxCalculatorAdapter() : ITaxCalculatorAdapter ...

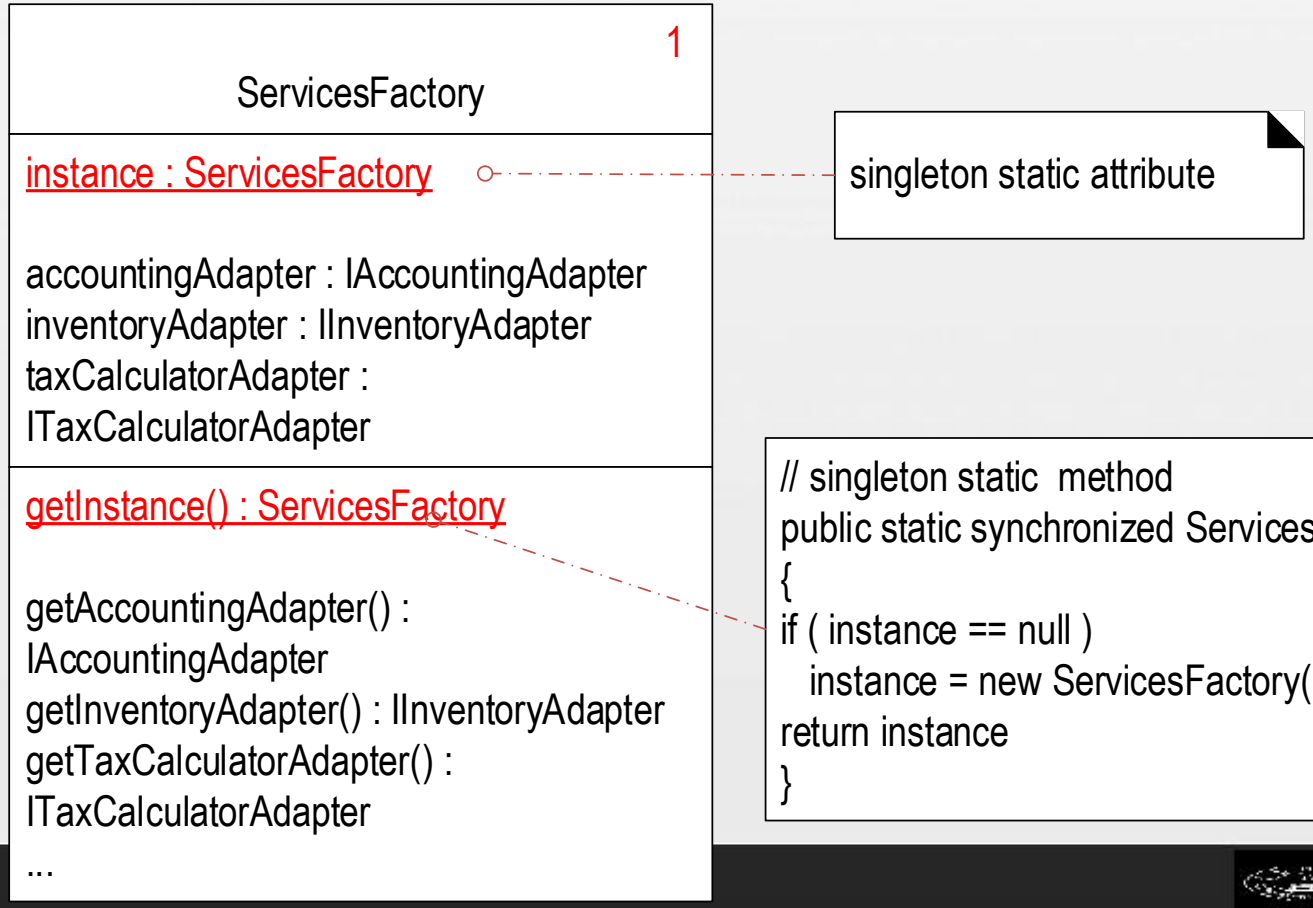
note that the factory methods return objects typed to an interface rather than a class, so that the factory can return any implementation of the interface

```
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;
```

Applying GoF Design Patterns

Singleton

Problem:	Exactly one instance of a class is allowed it is a "singleton." Objects need a global and single point of access.
Solution: (advice)	Define a static method of the class that returns the singleton.




```
// singleton static  method
public static synchronized ServicesFactory getInstance()
{
if ( instance == null )
   instance = new ServicesFactory()
return instance
}
```

Applying GoF Design Patterns

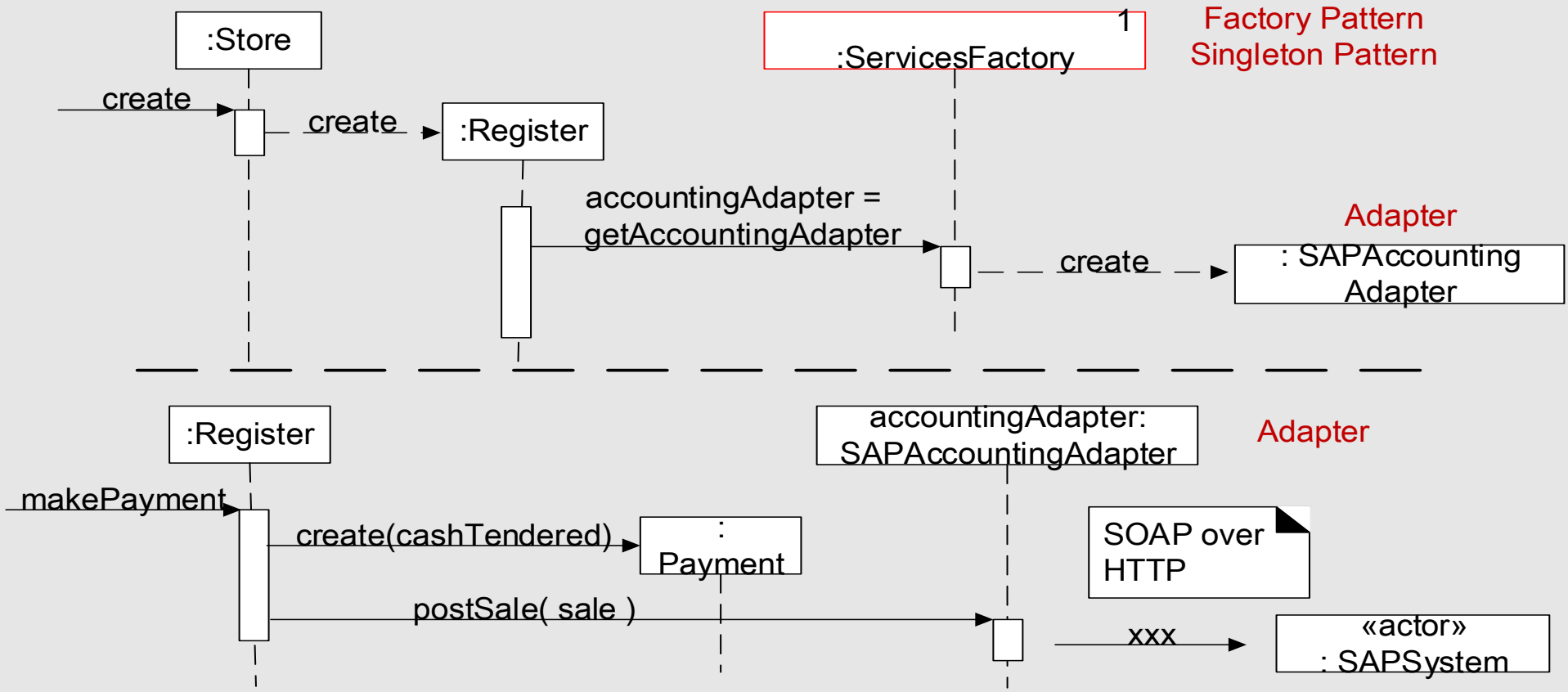
- **Singleton -- Implementation and Design Issues**
  - eager initialization
    - Create the instance when load the class
  - lazy initialization
    - Create the instance when then getinstance method is called
    - **Always wrap the method with concurrency control**
- **The Singleton pattern is often used for Factory objects and Facade objects**

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

```
public static synchronized ServicesFactory getInstance(){
   if ( instance == null )   {
      // critical section if multithreaded application
      instance = new ServicesFactory();   }
return instance;}
```

Applying GoF Design Patterns

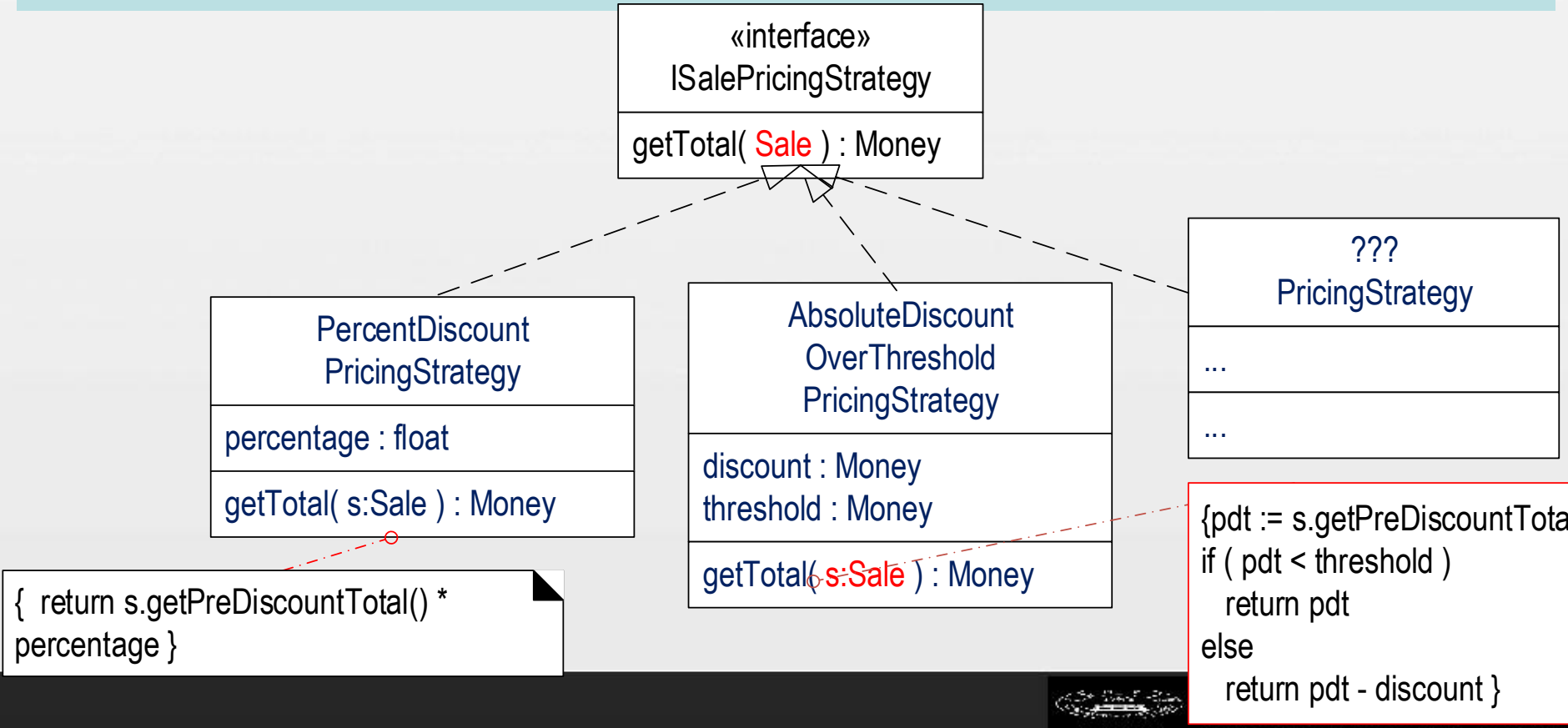
- **Conclusion of the External Services with Varying Interfaces Problem**
  - combination of Adapter, Factory, and Singleton patterns to provide Protected Variations from the varying interfaces of external service



Applying GoF Design Patterns

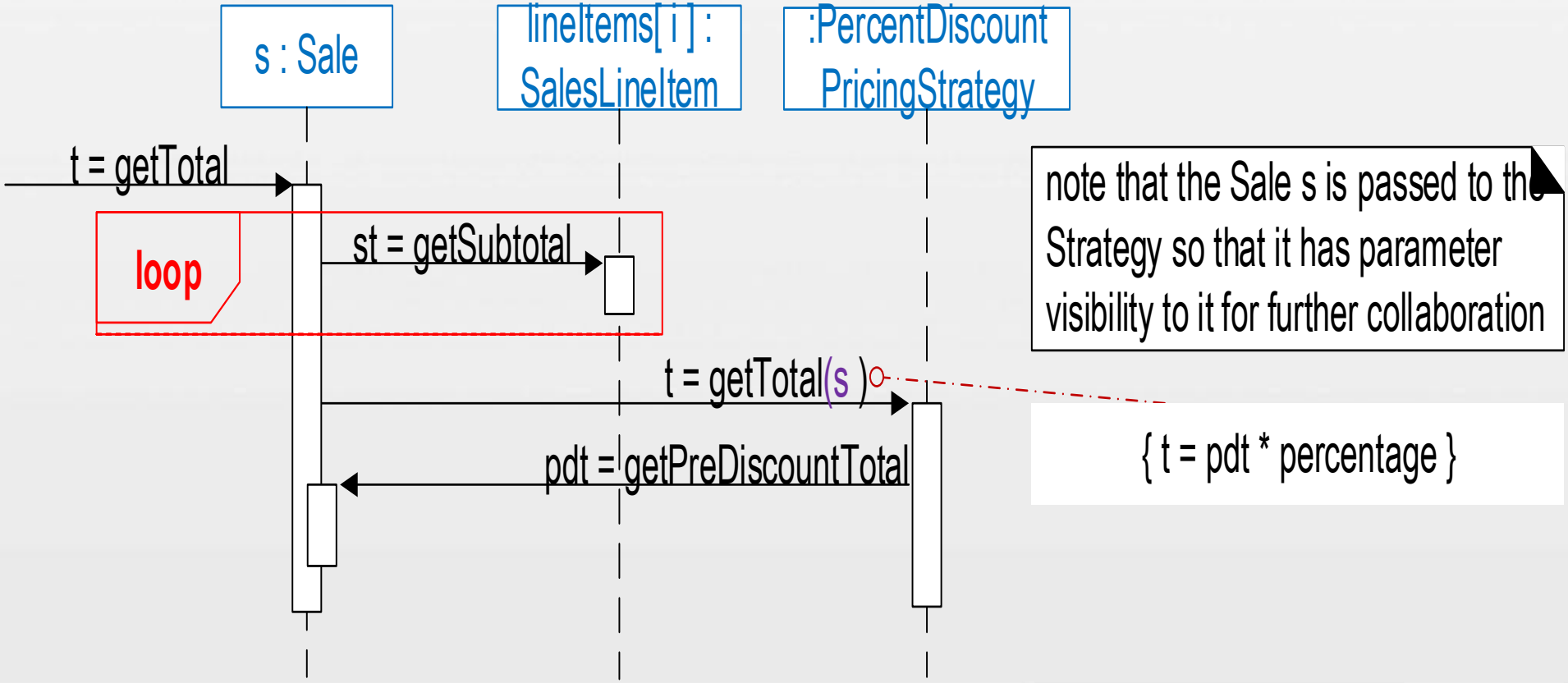
Strategy

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



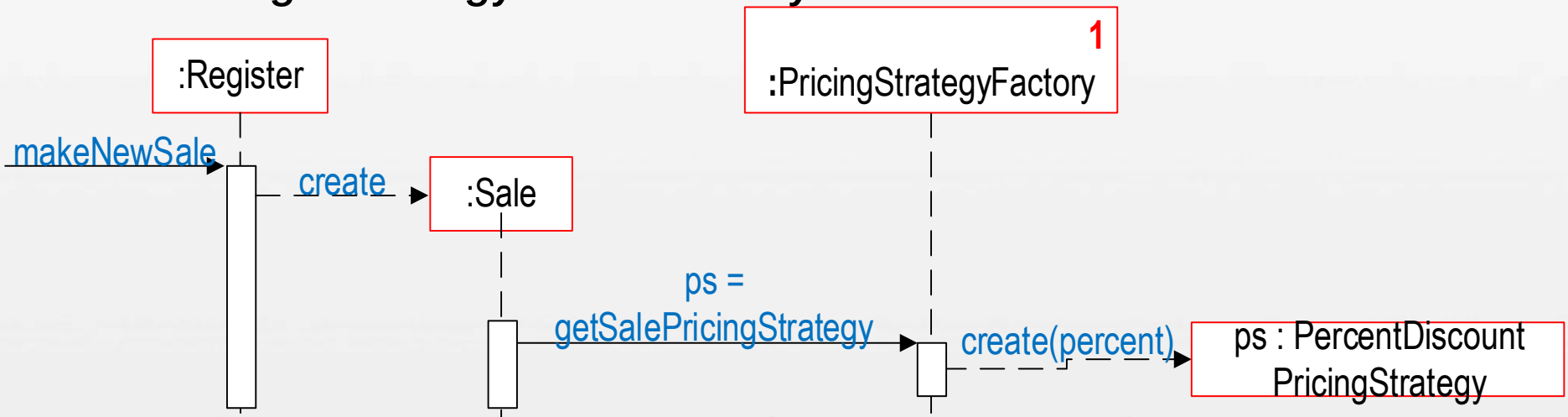
Applying GoF Design Patterns

- **Strategy**
  - **context object**
    - strategy object is attached to a context object the object to which it applies the algorithm. Sale



Applying GoF Design Patterns

- **Strategy**
  - **Creating a Strategy with a Factory**



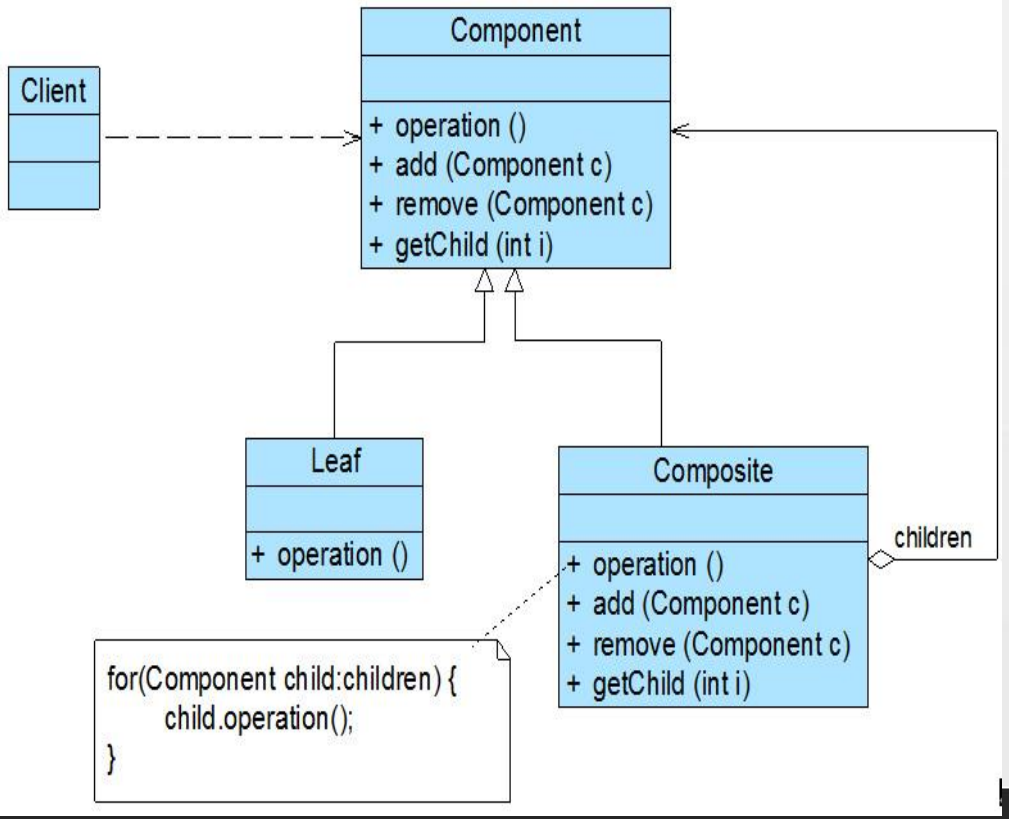
  - frequently changing pricing policy (it could be every hour), it is not desirable to cache the created strategy instance
  - Link to external data store

- Summary
  - Strategy is based on Polymorphism.
  - provides Protected Variations
  - often created by a Factory

Applying GoF Design Patterns

■ Composite

Problem:	How to treat a group or composition structure of objects the same way (polymorphically) as a non-composite (leaf) object?
Solution: (advice)	Define classes for composite and leaf objects so that they implement the same interface.



包括3个角色：

(1) Component（抽象构件）：为叶子构件和容器构件对象声明接口

(2) Leaf（叶子构件）：组合结构中表示叶子节点对象，没有子节点，实现在抽象构件中定义的行为

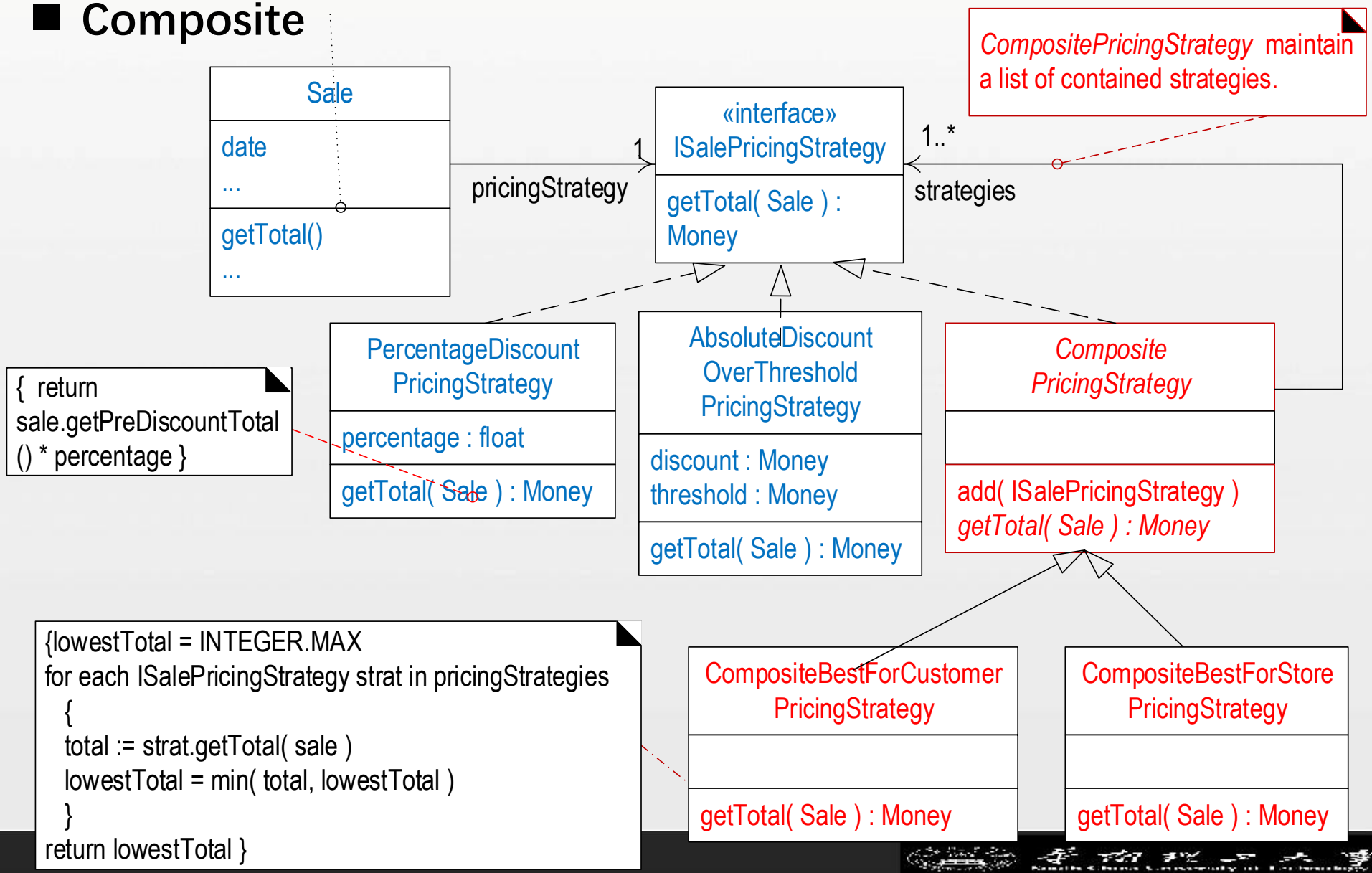
(3) Composite（容器构件）：组合结构中表示容器节点对象，包含子节点，子节点可以是叶子节点，也可以是容器节点。它提供一个集合用于存储子节点，实现在抽象构件中定义的行为

组合模式的关键是定义了一个抽象构件类，既可以代表叶子，又可以代表容器。客户端针对该抽象构件类进行编程，无须知道它到底表示的是叶子还是容器，可以对其进行统一处理

由于容器构件中可以包含容器构件，对容器构件处理时需要使用递归算法

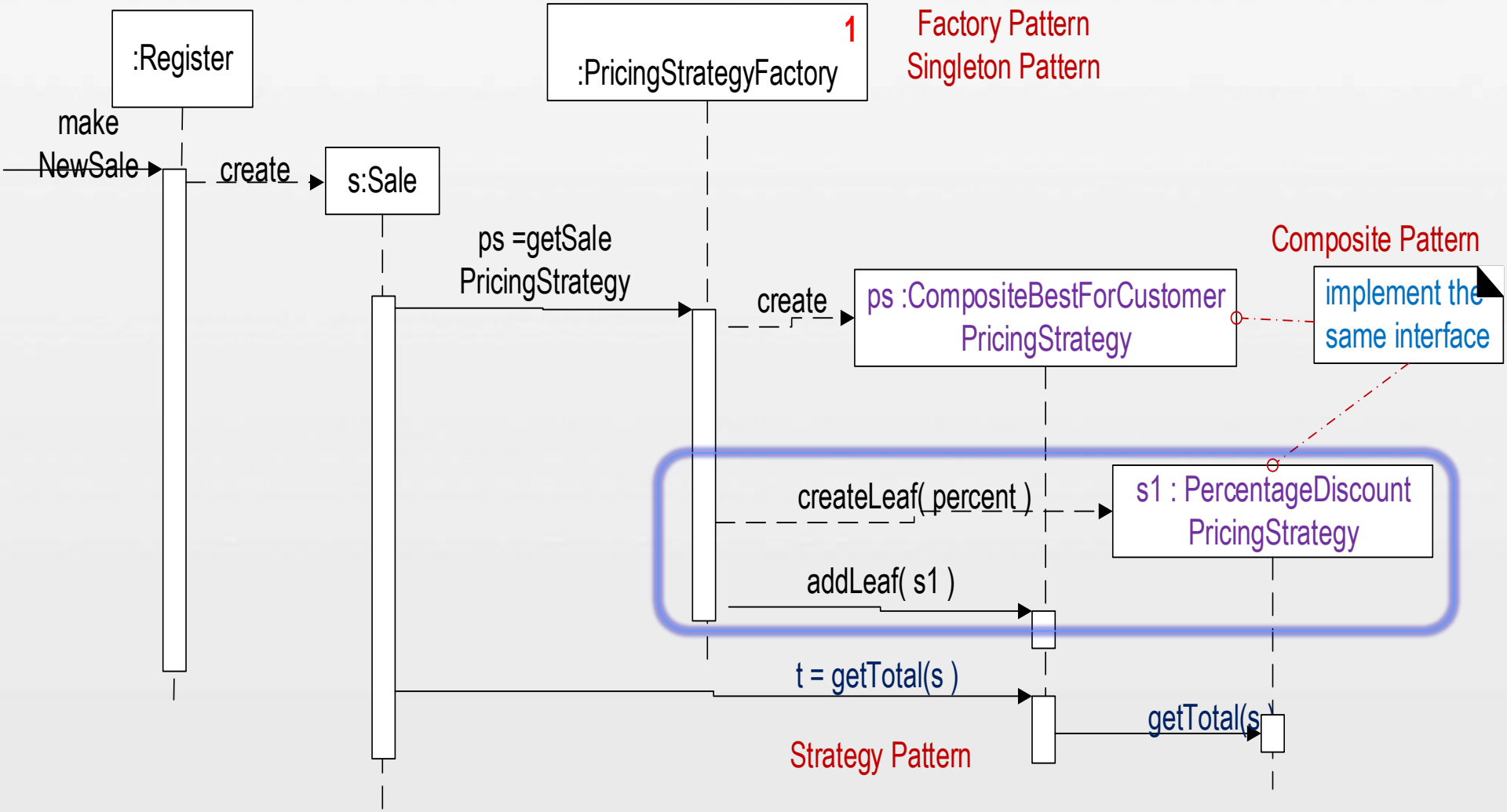
Applying GoF Design Patterns

- **Composite**



Applying GoF Design Patterns

- **Composite**

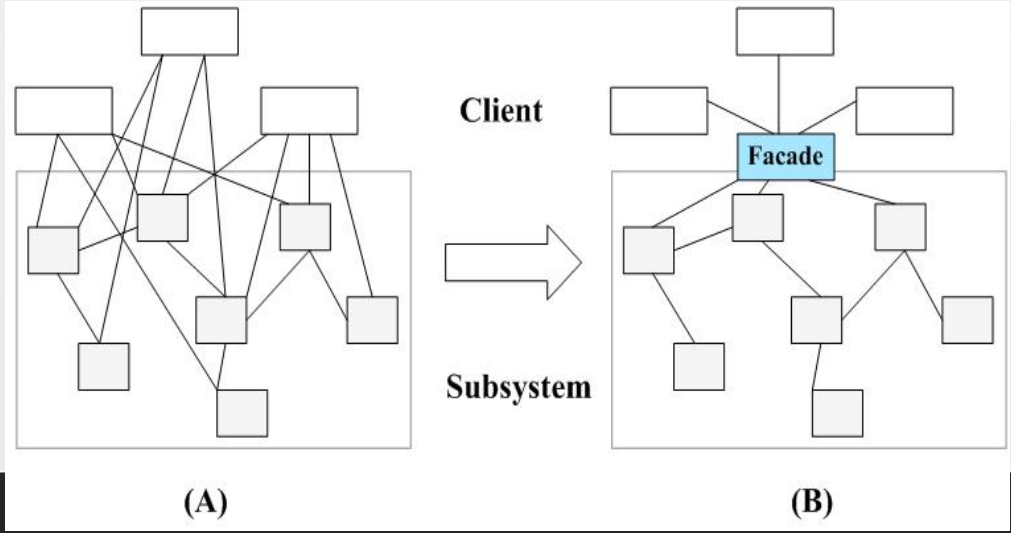


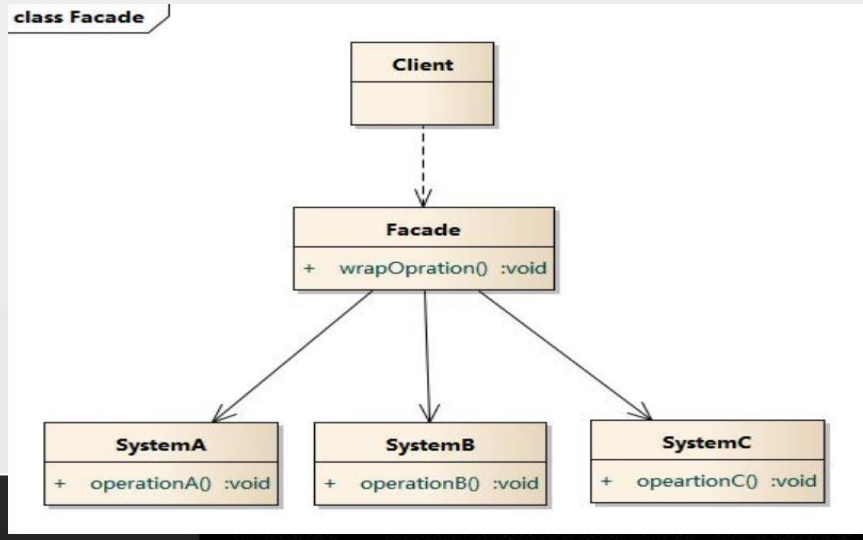
Applying GoF Design Patterns

■ Façade

Problem:	A common, unified interface to a disparate set of implementations or interfacessuch as within a subsystemis required. There may be undesirable coupling to many things in the subsystem, or the implementation of the subsystem may change. What to do?
Solution: (advice)	Define a single point of contact to the subsystema facade object that wraps the subsystem. This facade object presents a single unified interface and is responsible for collaborating with the subsystem components.

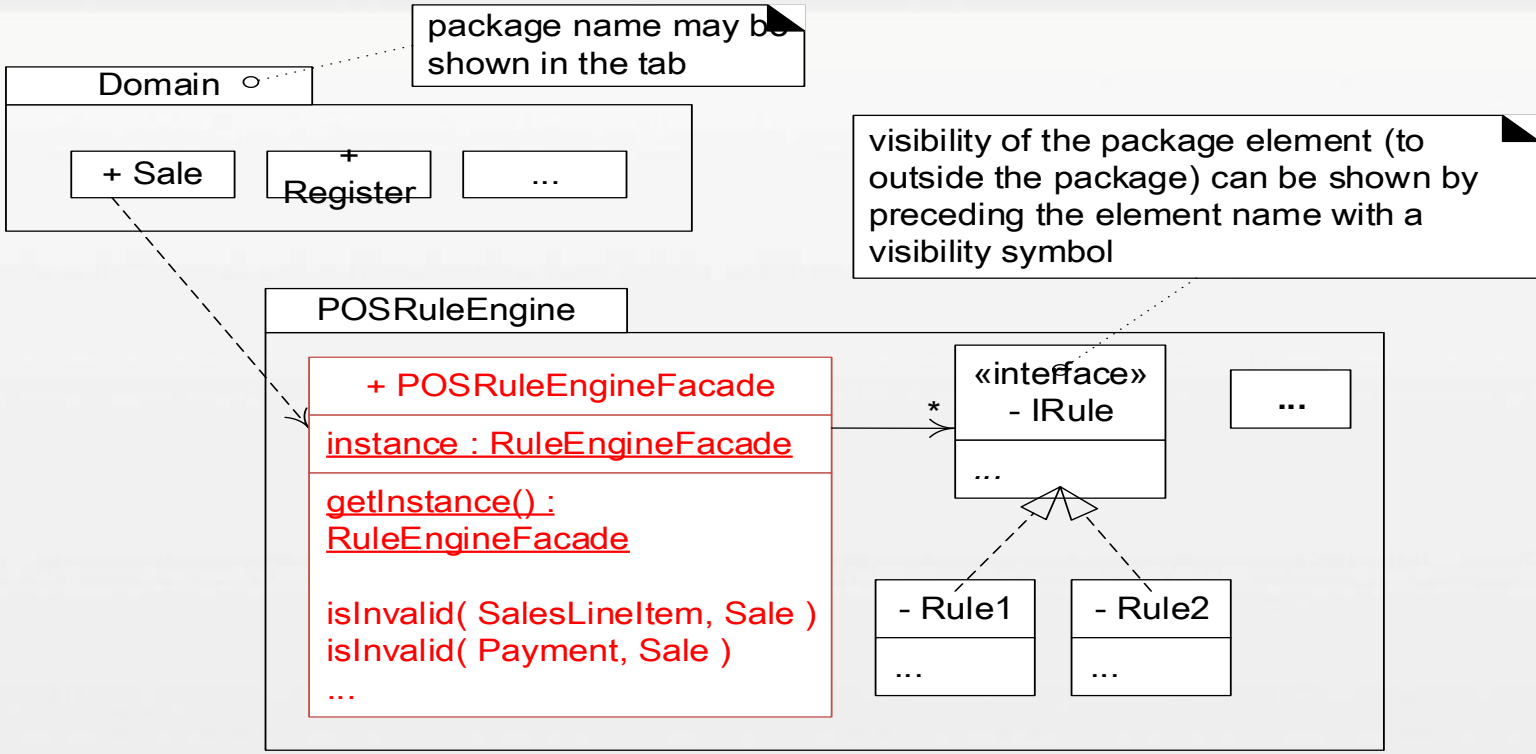
➢Facade Pattern: Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use





Applying GoF Design Patterns

- **Façade**

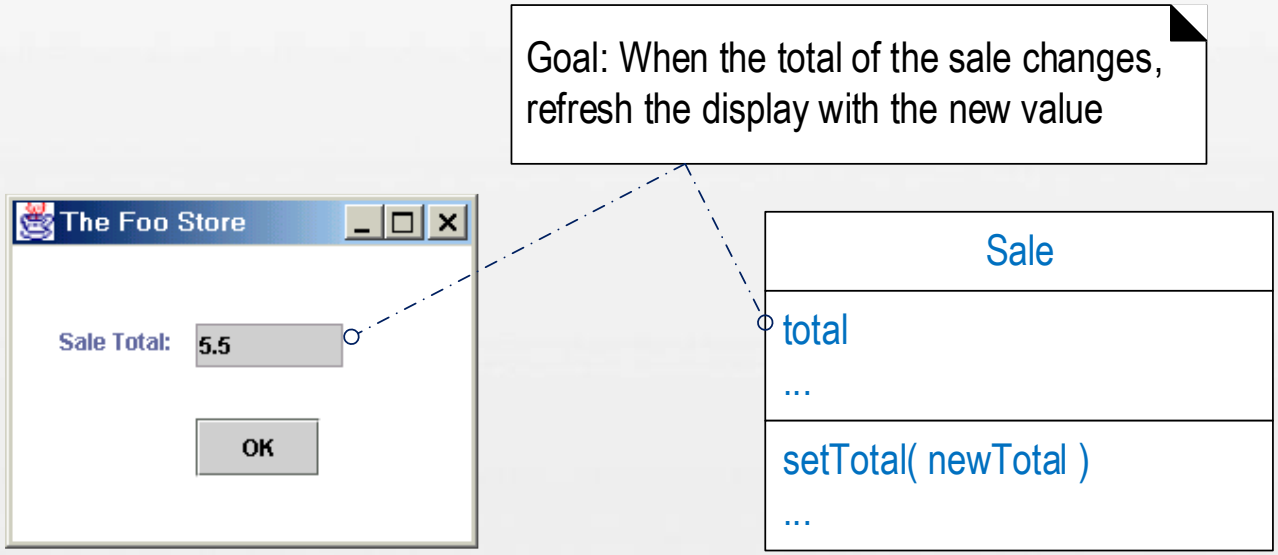



```
public class Sale {
public void makeLineItem( ProductDescription desc, int quantity )
{
        SalesLineItem sli = new SalesLineItem( desc, quantity );
        // call to the Facade
        if ( POSRuleEngineFacade.getInstance().isInvalid( sli, this ) )
        return;
        lineItems.add( sli );
} // ... } // end of class
```

Applying GoF Design Patterns

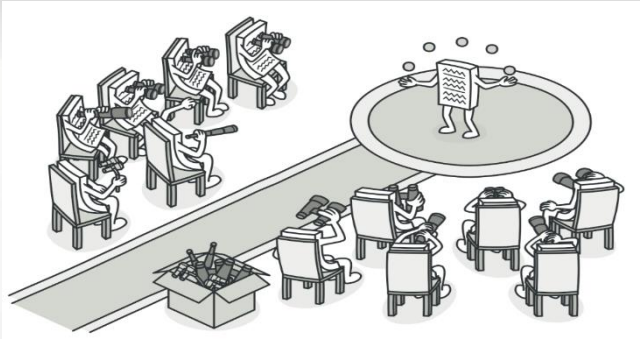
Observer/Publish-Subscribe/Delegation

- adding the ability for a GUI window to refresh its display of the sale total when the total changes

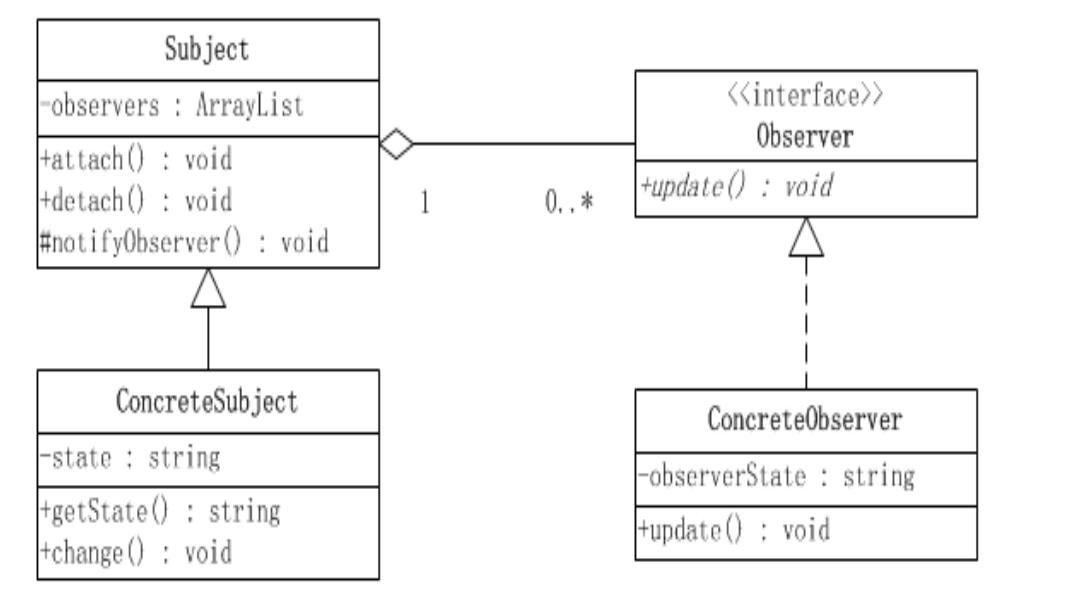


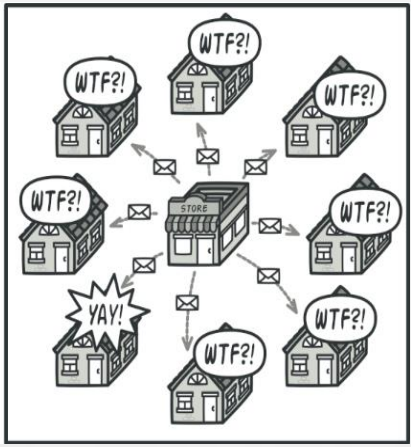
Problem:	Different kinds of subscriber objects are interested in the state changes or events of a publisher object, and want to react in their own unique way when the publisher generates an event. Moreover, the publisher wants to maintain low coupling to the subscribers. What to do?
Solution: (advice)	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.

Applying GoF Design Patterns



- **Observer/Publish-Subscribe/Delegation**

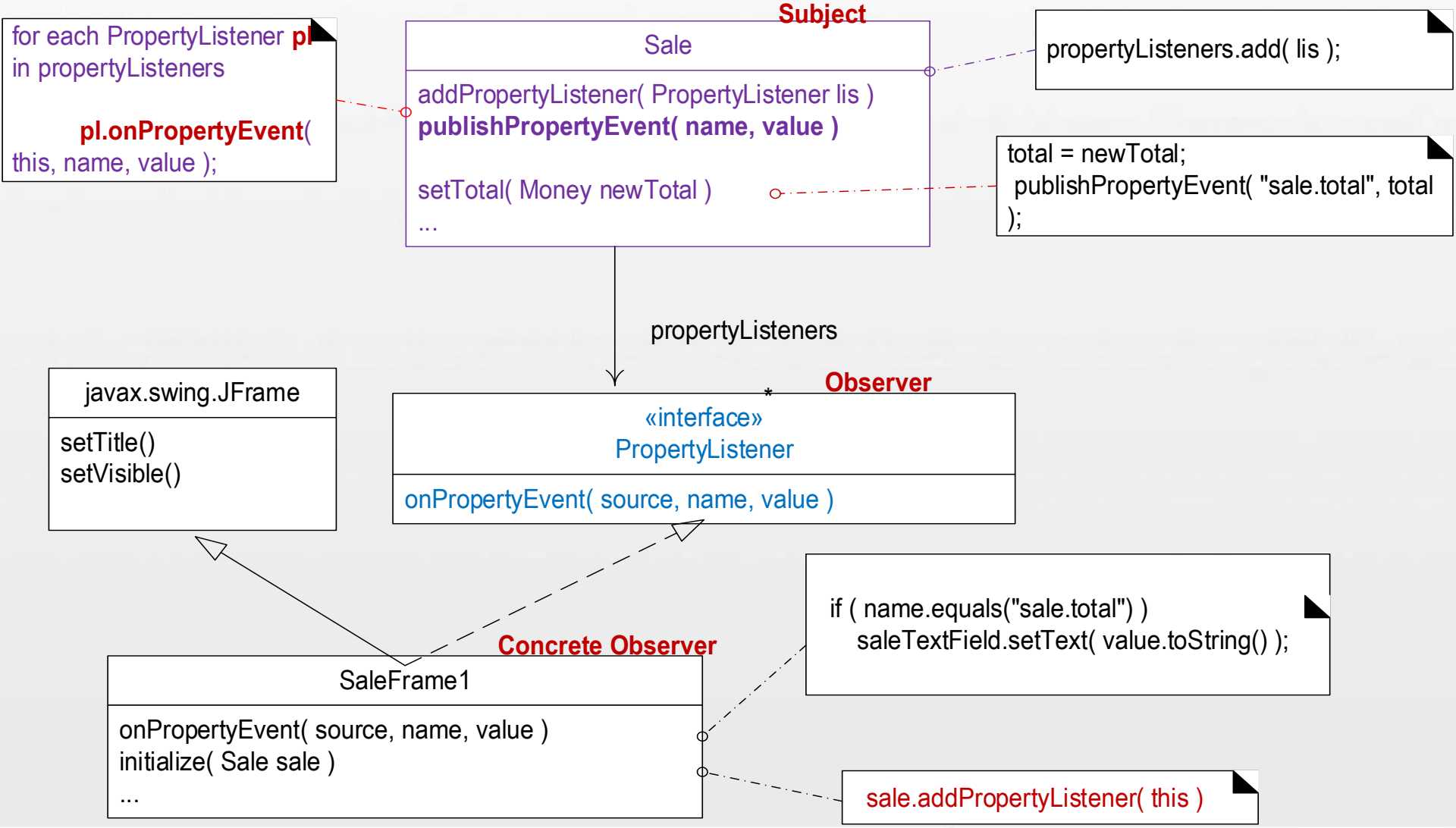




- **Subject**
  - Attach – add observer
  - notifyObserver – publish info to observer. Call the update method of observer
- **Observer**
  - Update method
  - Call the attach method to add observer to subject (visiting the store)

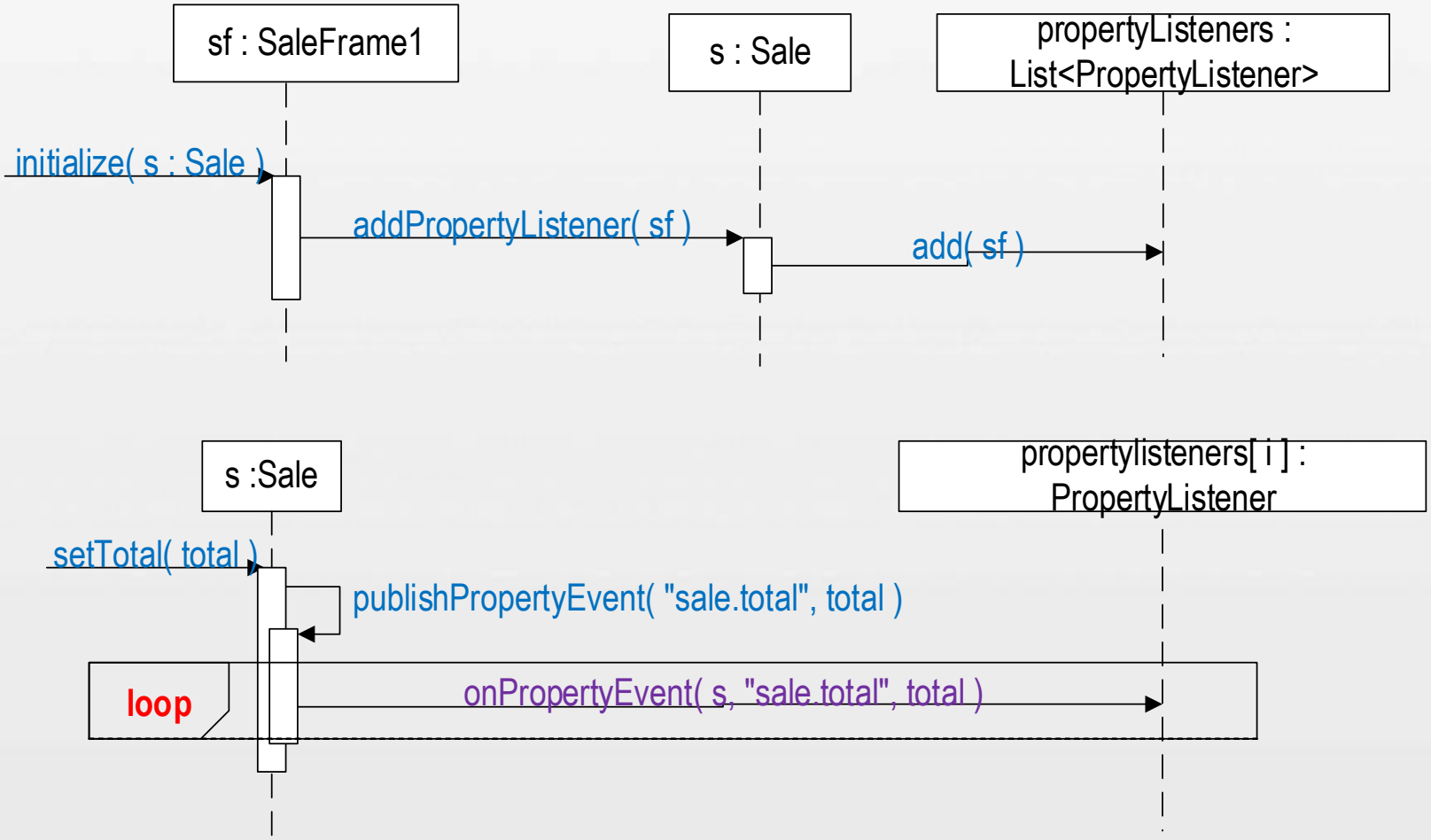
Applying GoF Design Patterns

- **Observer/Publish-Subscribe/Delegation**



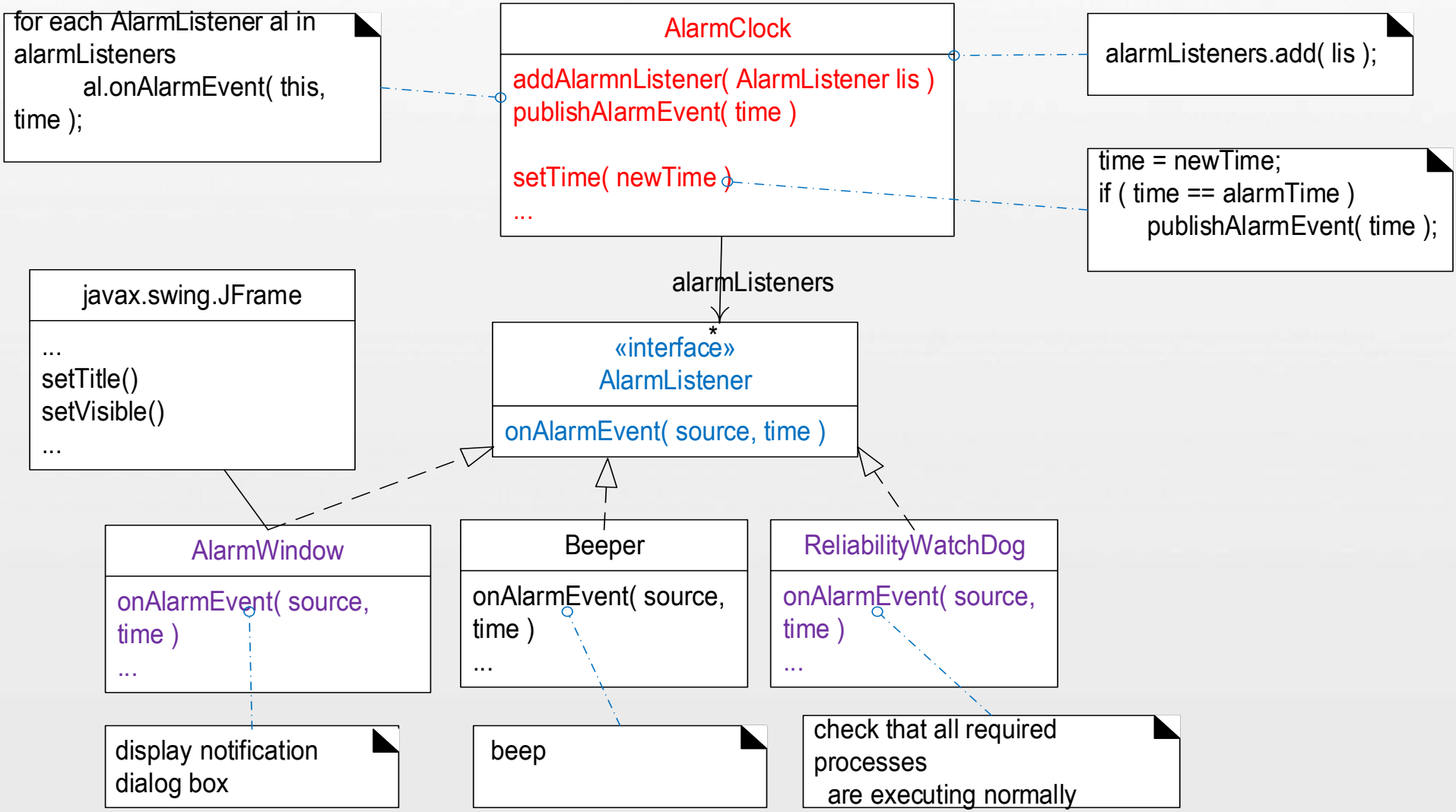
Applying GoF Design Patterns

- **Observer/Publish-Subscribe/Delegation**



Applying GoF Design Patterns

- **Observer/Publish-Subscribe/Delegation**



Applying GoF Design Patterns

- **Observer/Publish-Subscribe/Delegation**
  - **Summary**
    - Observer provides a way to loosely couple objects in terms of communication

- **Criticism of patterns**
  - **if all you have is a hammer, everything looks like a nail.**

Applying GoF Design Patterns

- reference
  - https://refactoring.guru/design-patterns/catalog