Applying UML and Patterns

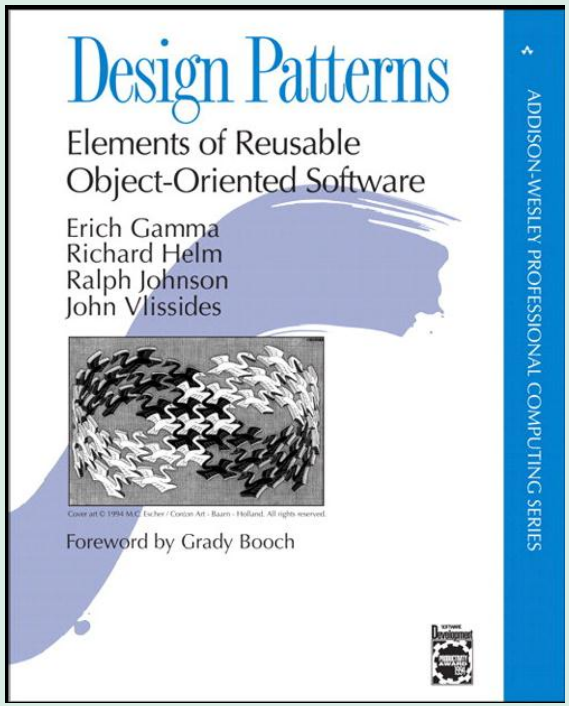
**An Introduction to
Object-oriented Analysis
and Design
and Iterative Development**

Part IV Elaboration Iteration II – More Patterns

Chap 26
Applying GoF Design Patterns

The Gang-of-Four Design Patterns

- 23 Gof Design Patterns.
- perhaps 15 are common and most useful.
- Book-- ***Design Patterns***



Catalog of Patterns: Three categories

Creational

- Abstract Factory
- Factory Method
- Builder
- Prototype
- Singleton

Structural

- Adapter
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

Behavioural

- Chain of Responsibility
- Interpreter
- Mediator
- Observer
- Strategy
- Command
- Iterator
- Memento
- State
- Template Method
- Visitor

Adapter (GoF)

- Name: Adapter
- 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.
- Related Patterns: A resource adapter that hides an external system may also be considered a Facade object
- Guideline: Include Pattern in Type Name

Adapter (GoF)

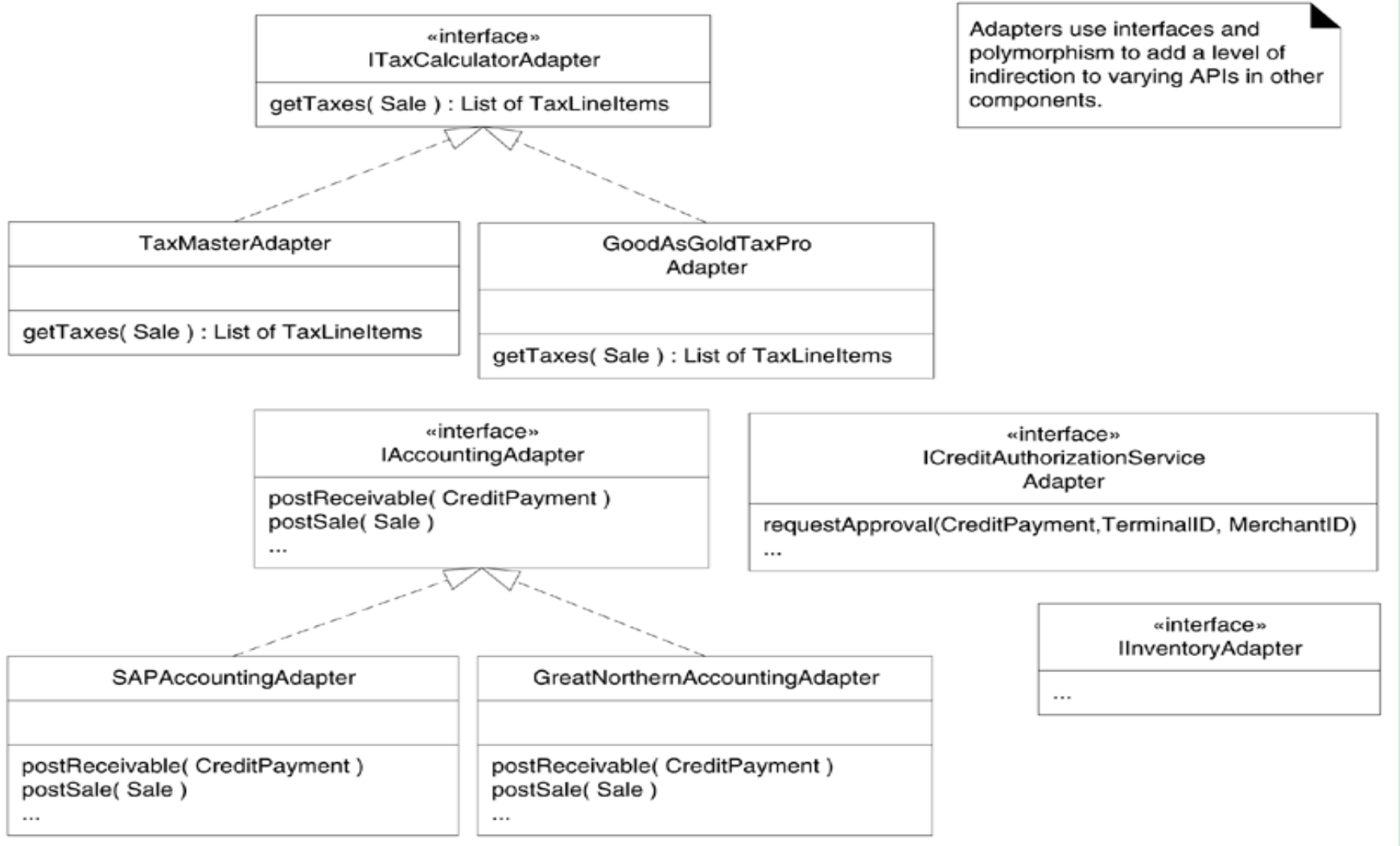


Figure 26.1. The Adapter pattern.

Adapter (GoF)

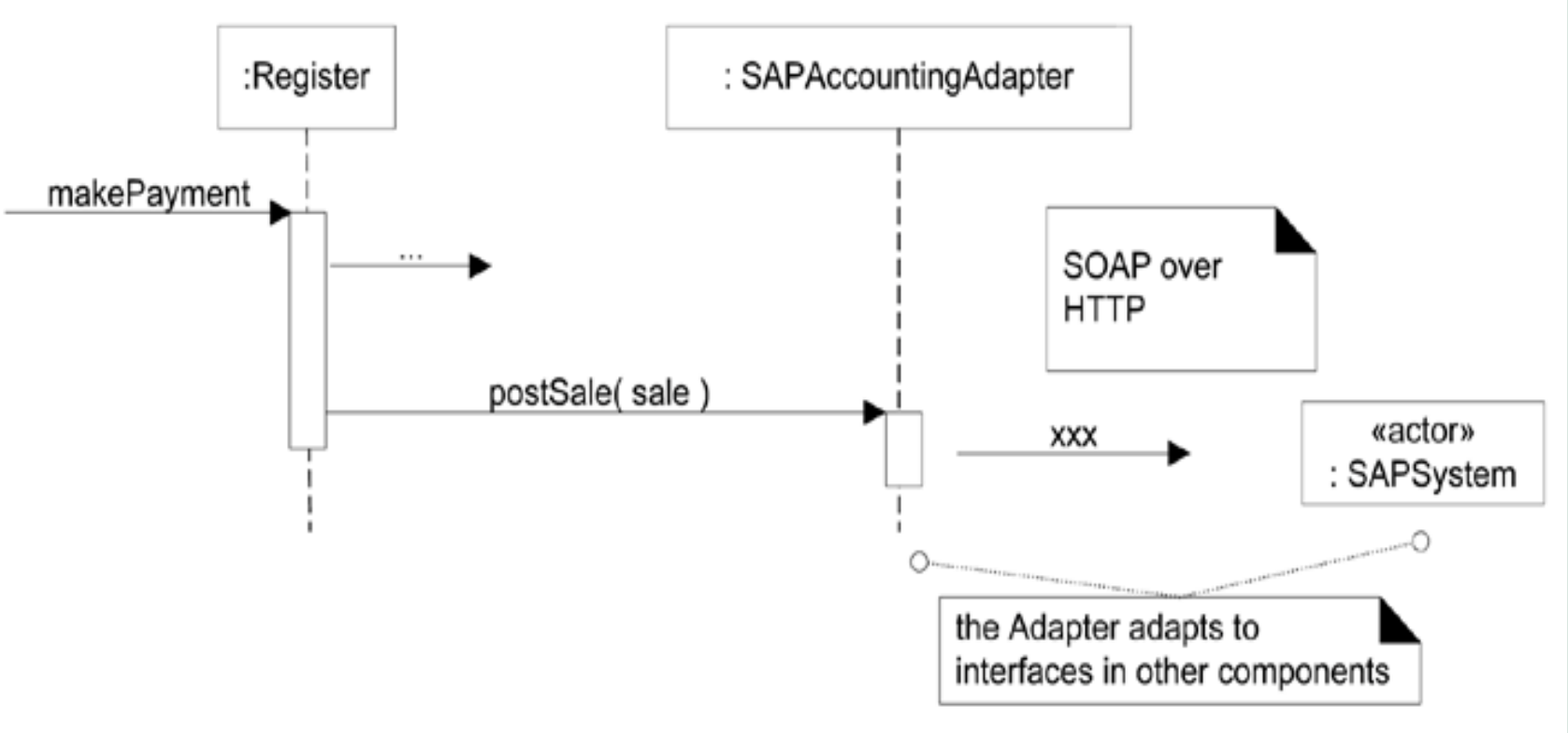
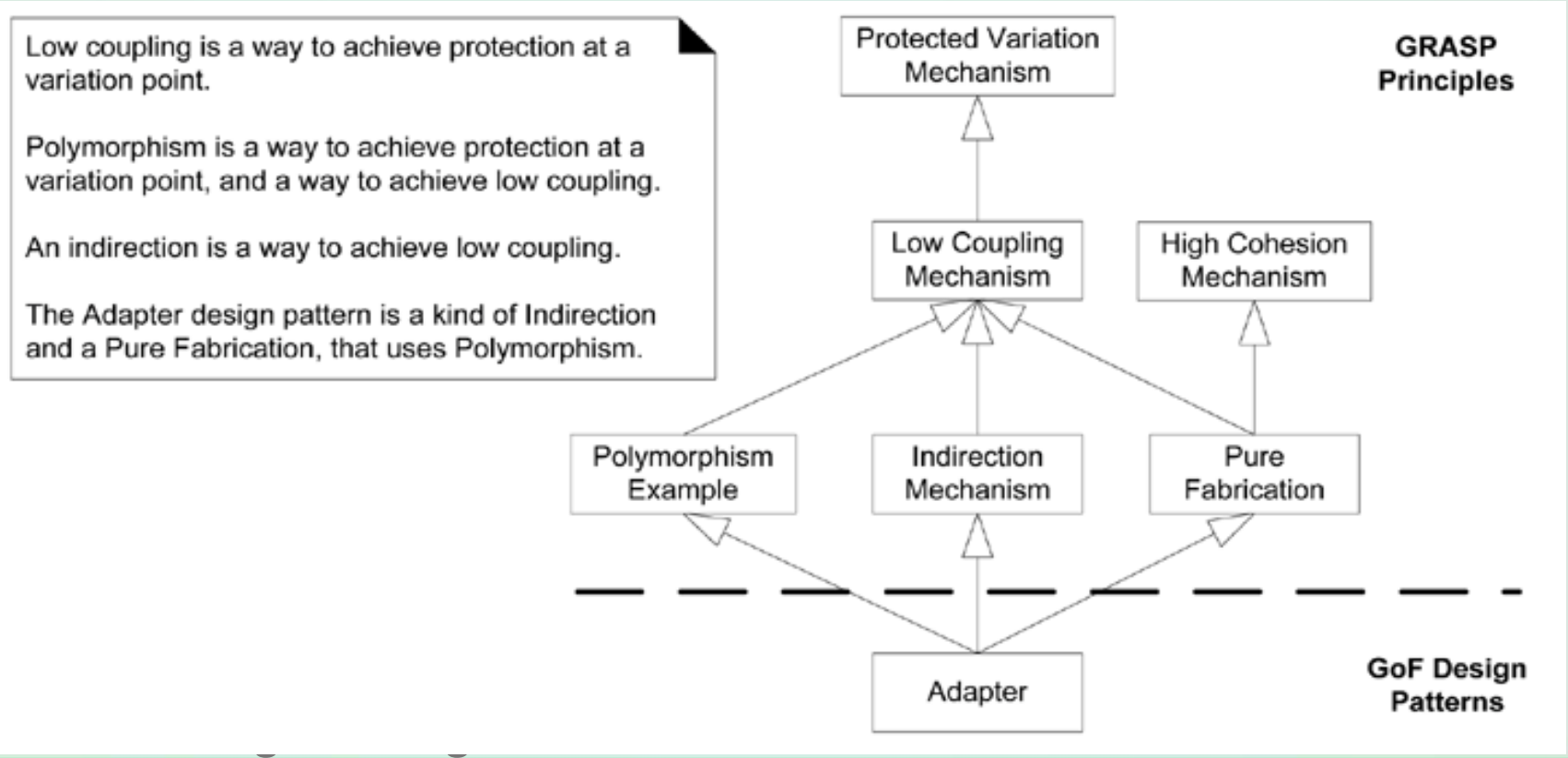


Figure 26.2. Using an Adapter

Adapter and GRASP

- Most design patterns can be seen as specializations of a few basic GRASP principles.
- Underlying themes are more important

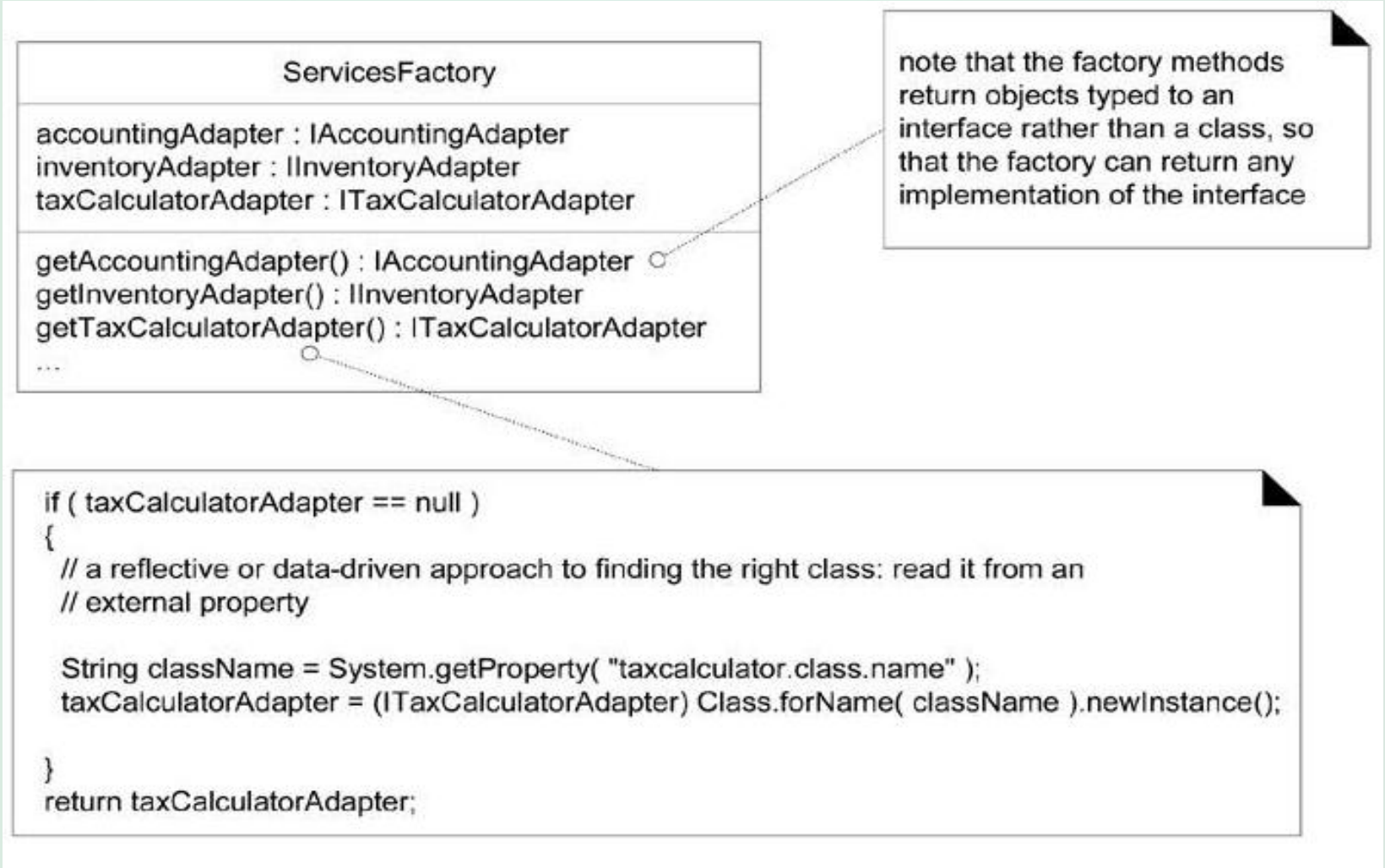


Factory

- also called *Simple Factory* or *Concrete Factory*. This pattern is not a GoF design pattern.
- It is also 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?
- Solution: Create a Pure Fabrication object called a Factory that handles the creation.
- Related Patterns: Factories are often accessed with the Singleton pattern

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;

Singleton (GoF)

- 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 allowedit is a "singleton." Objects need a global and single point of access.
- Solution: Define a static method of the class that returns the singleton.
- Related Patterns: The Singleton pattern is often used for Factory objects and Facade objectsanother GoF pattern that will be discussed.

Singleton (GoF)

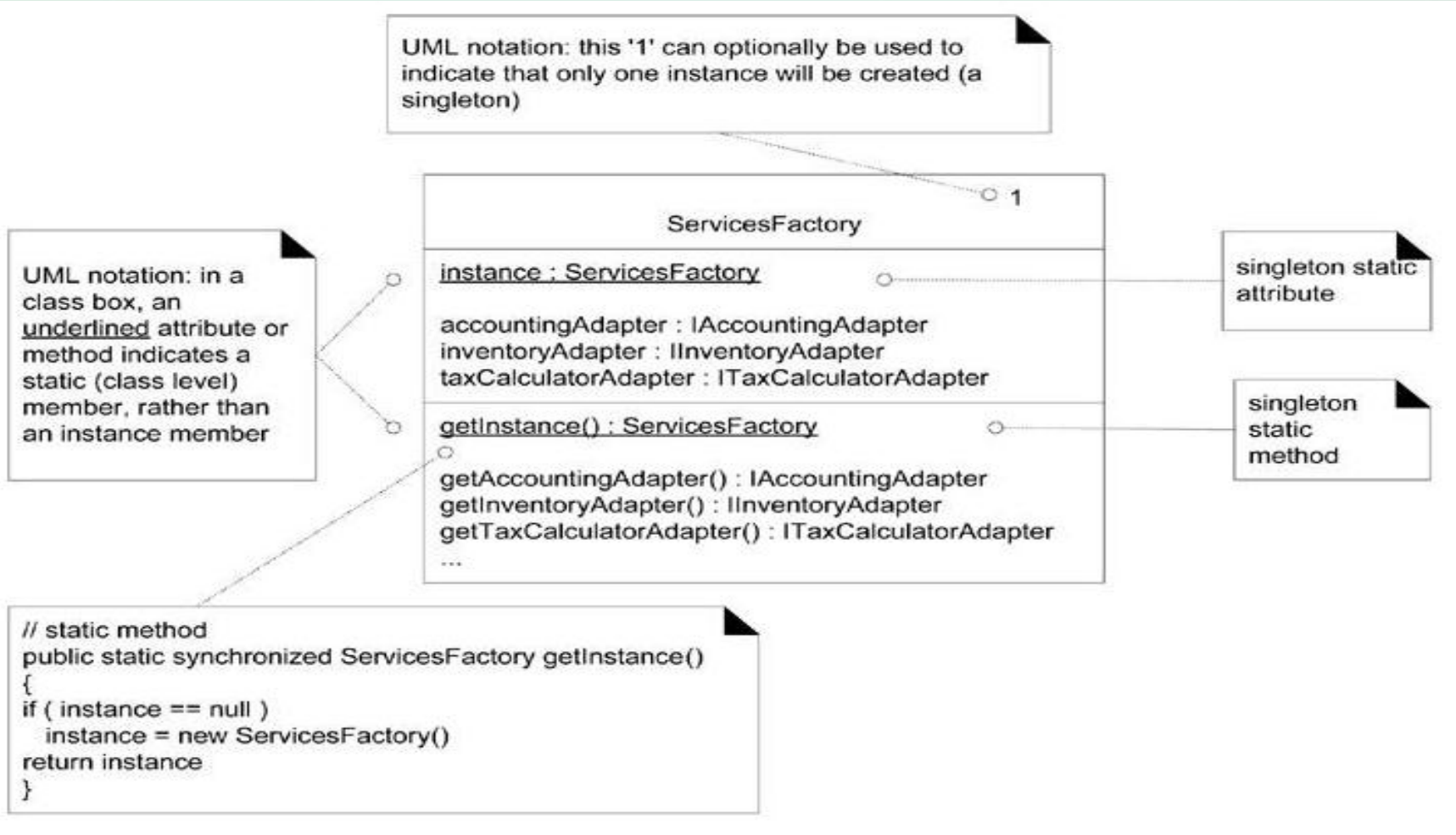


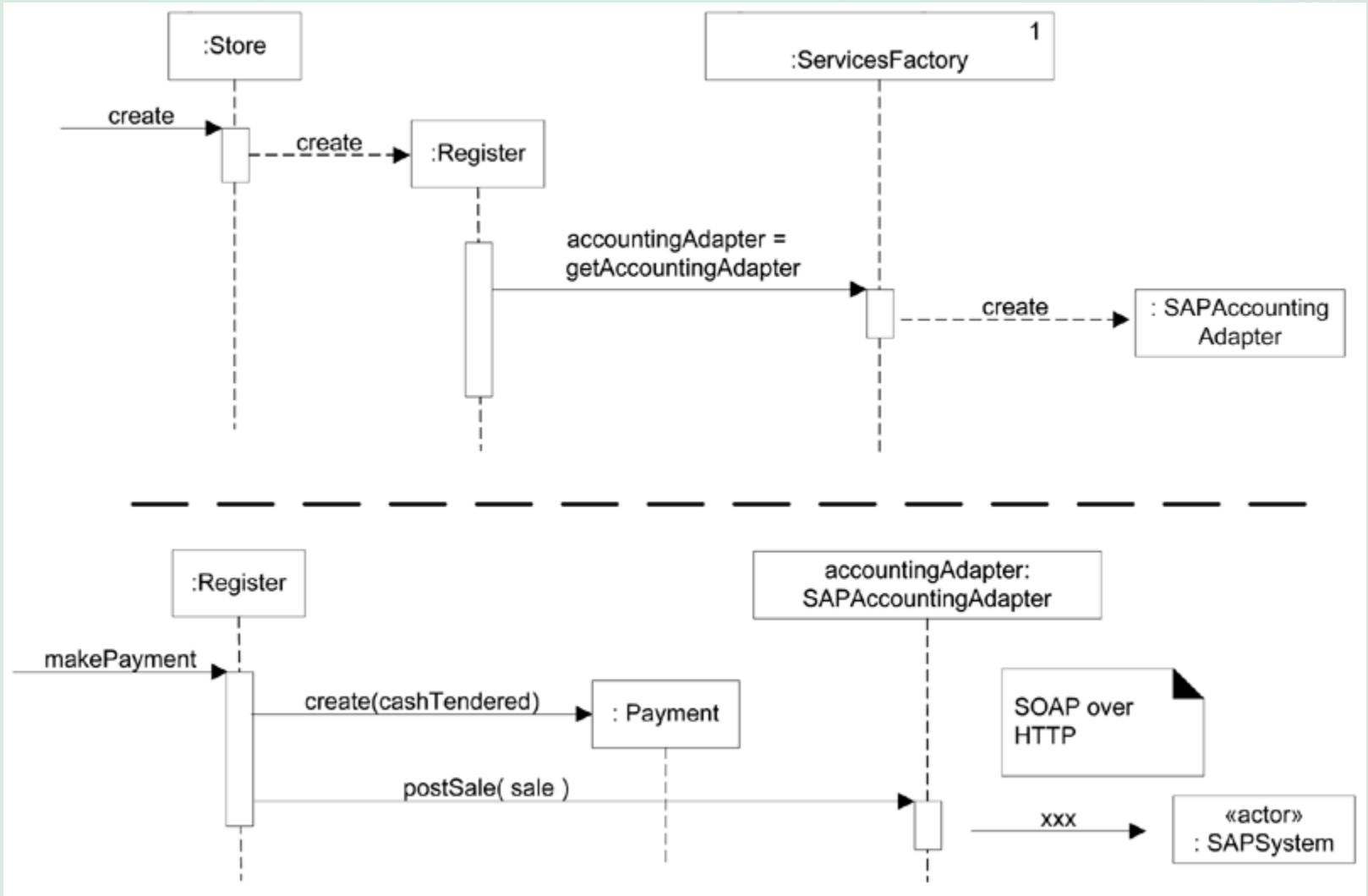
Figure 26.6. The Singleton pattern in the ServicesFactory class

Singleton (GoF)

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

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

Conclusion of the External Services with Varying Interfaces Problem



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.

Strategy (GoF)

Figure 26.9. Pricing Strategy classes

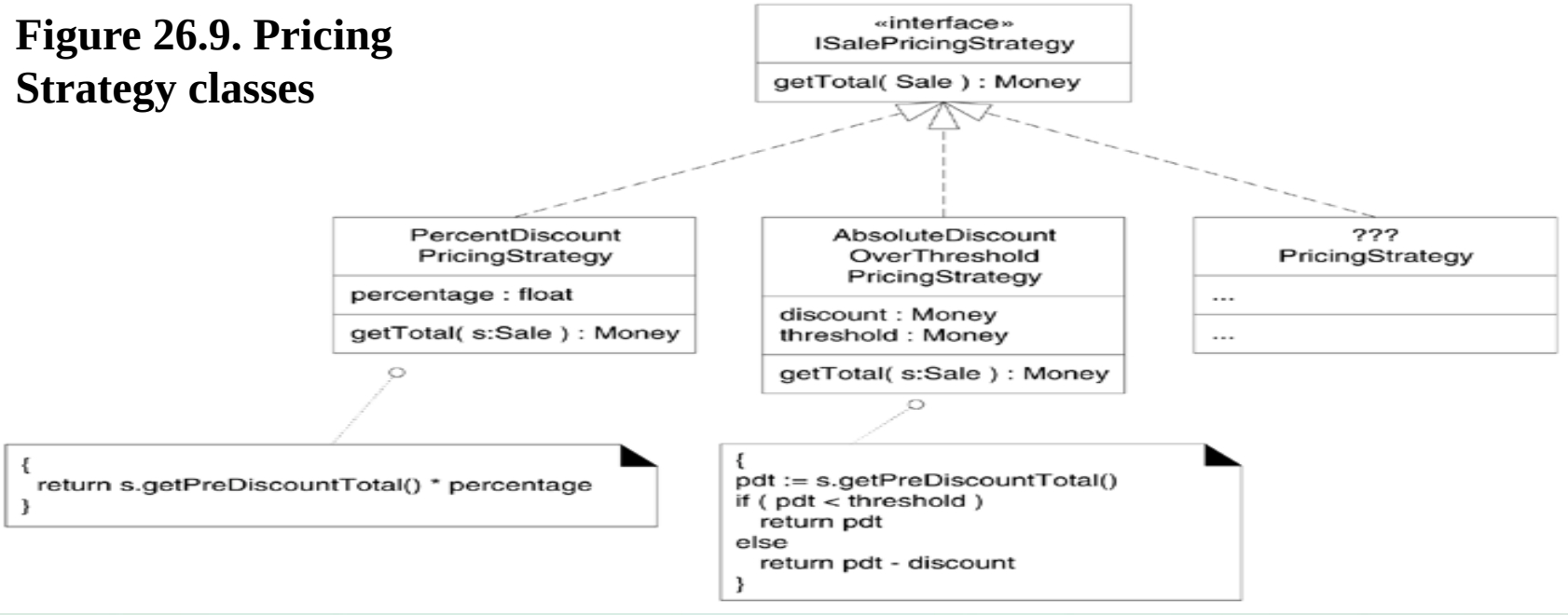
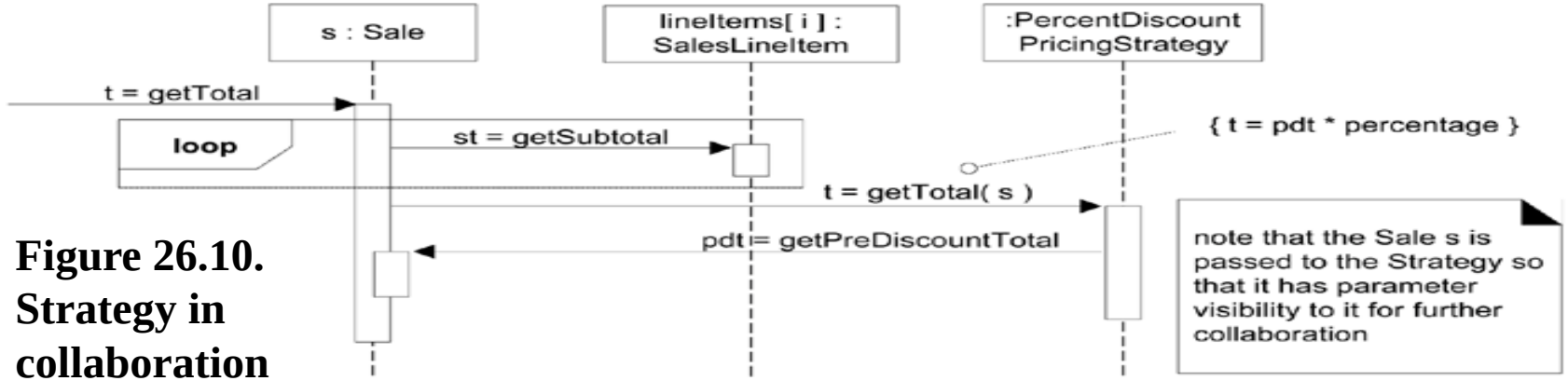


Figure 26.10. Strategy in collaboration



Strategy (GoF)

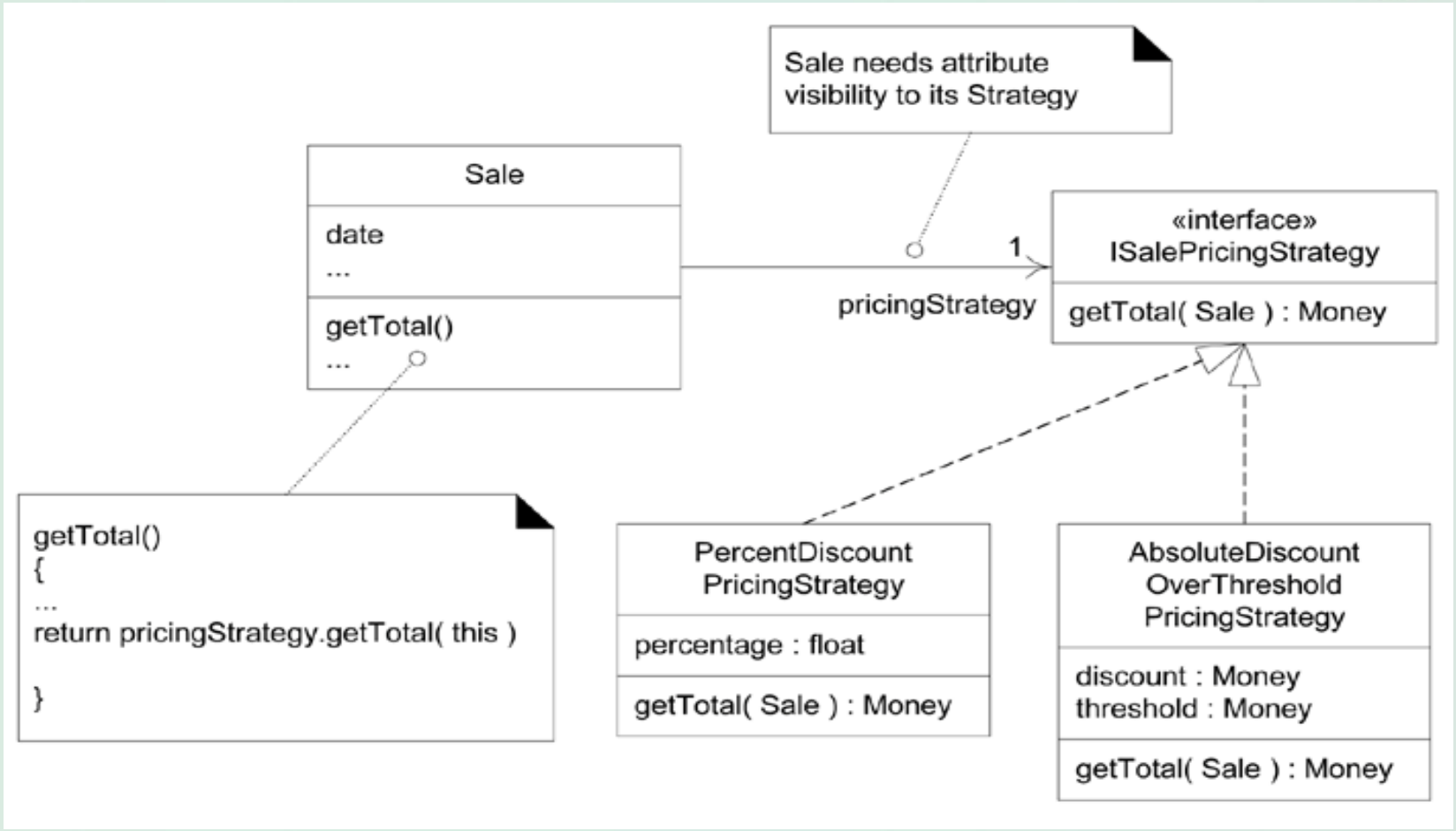


Figure 26.11. Context object needs attribute visibility to its strategy

Strategy (GoF)

- Creating a Strategy with a Factory

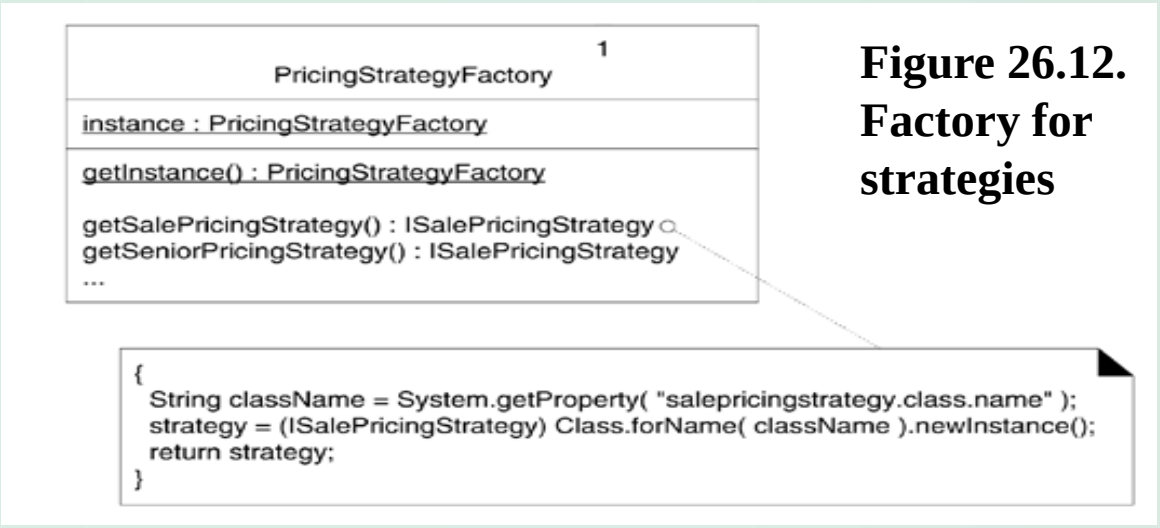


Figure 26.12. Factory for strategies

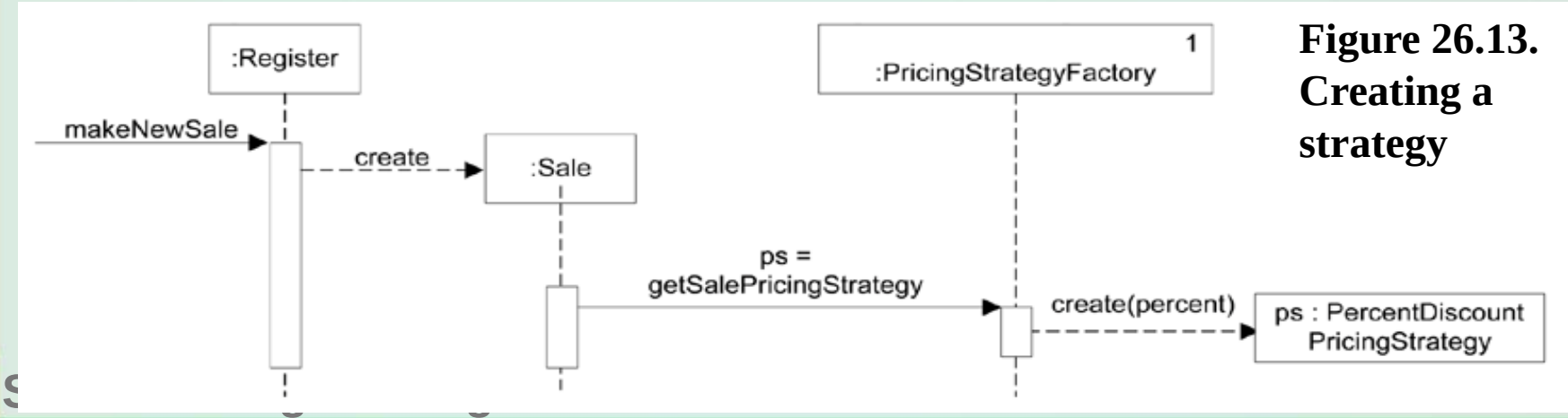


Figure 26.13. Creating a strategy

Composite (GoF) and Other Design Principles

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

- 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.

Composite (GoF) and Other Design Principles

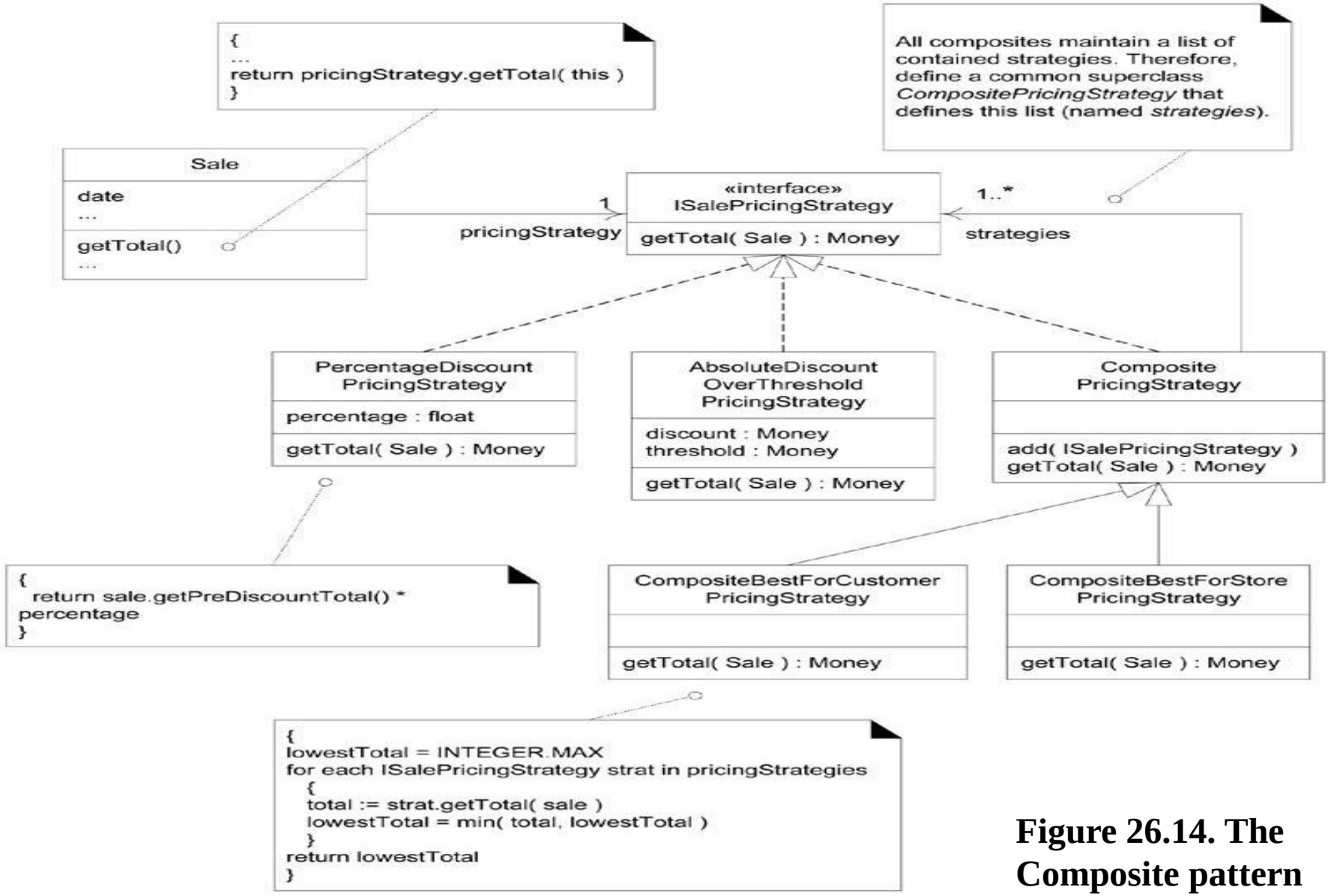


Figure 26.14. The Composite pattern

Composite (GoF) and Other Design Principles

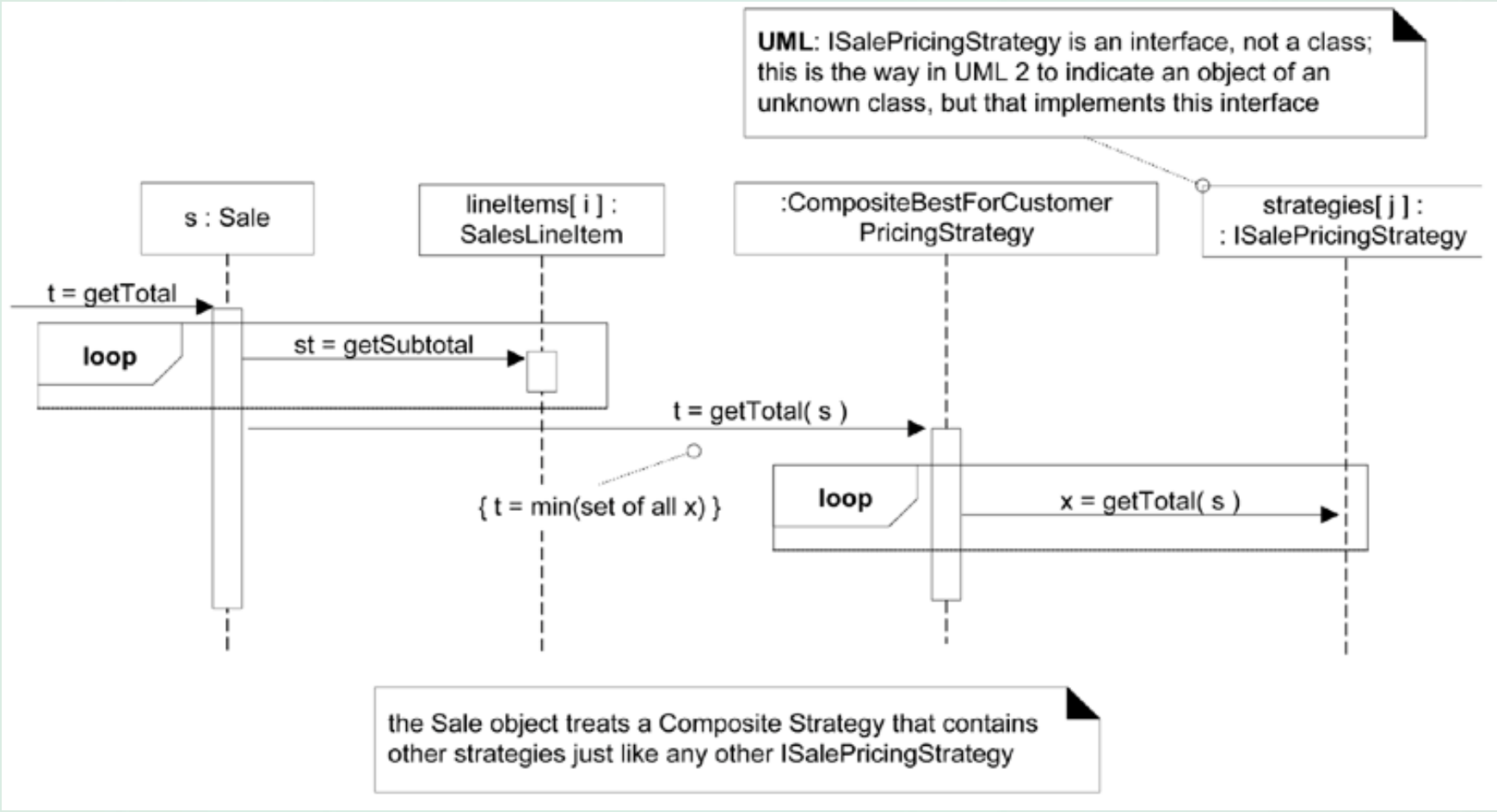


Figure 26.15. Collaboration with a Composite

Composite (GoF) and Other Design Principles

```
// 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
```

Composite (GoF) and Other Design Principles

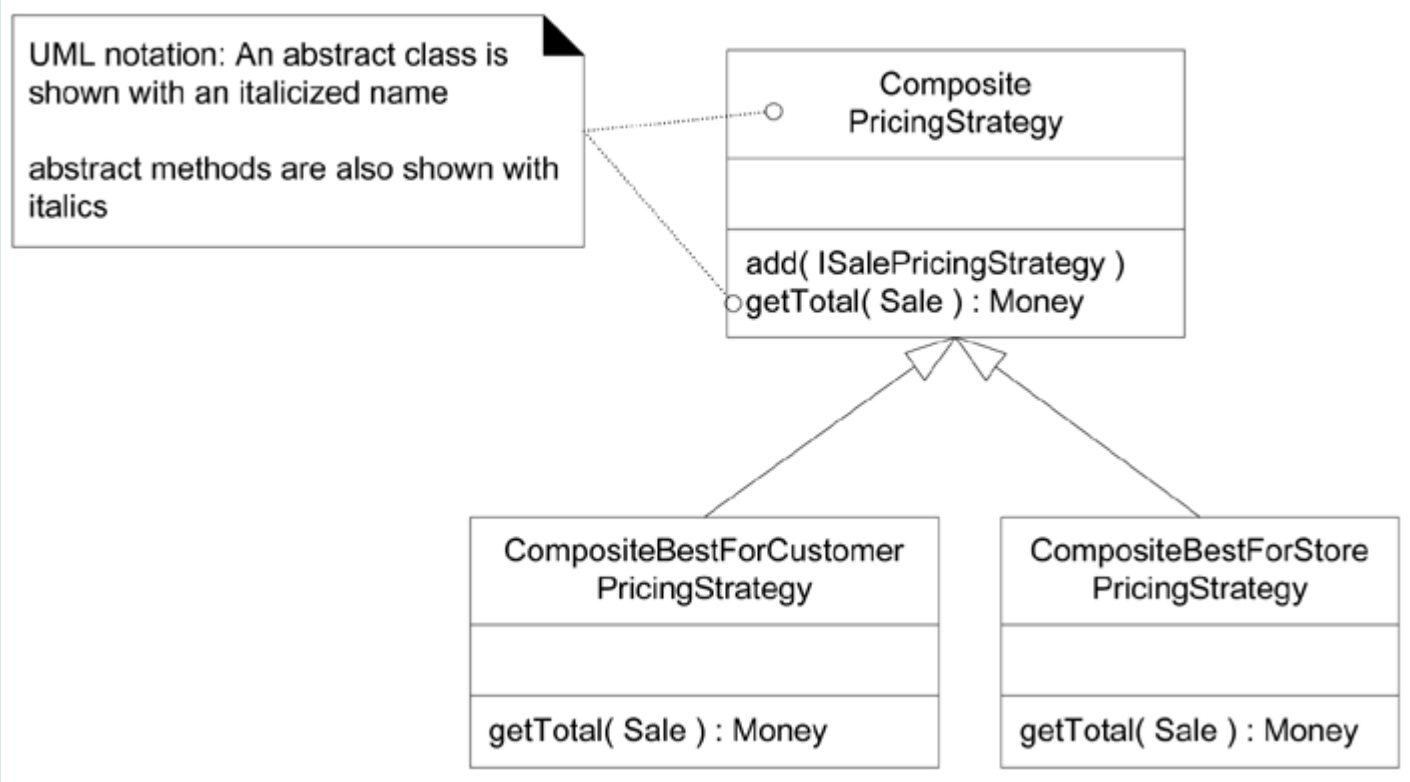


Figure 26.16. Abstract superclasses, abstract methods, and inheritance in the UML

Composite (GoF) and Other Design Principles

- Creating Multiple SalePricingStrategies

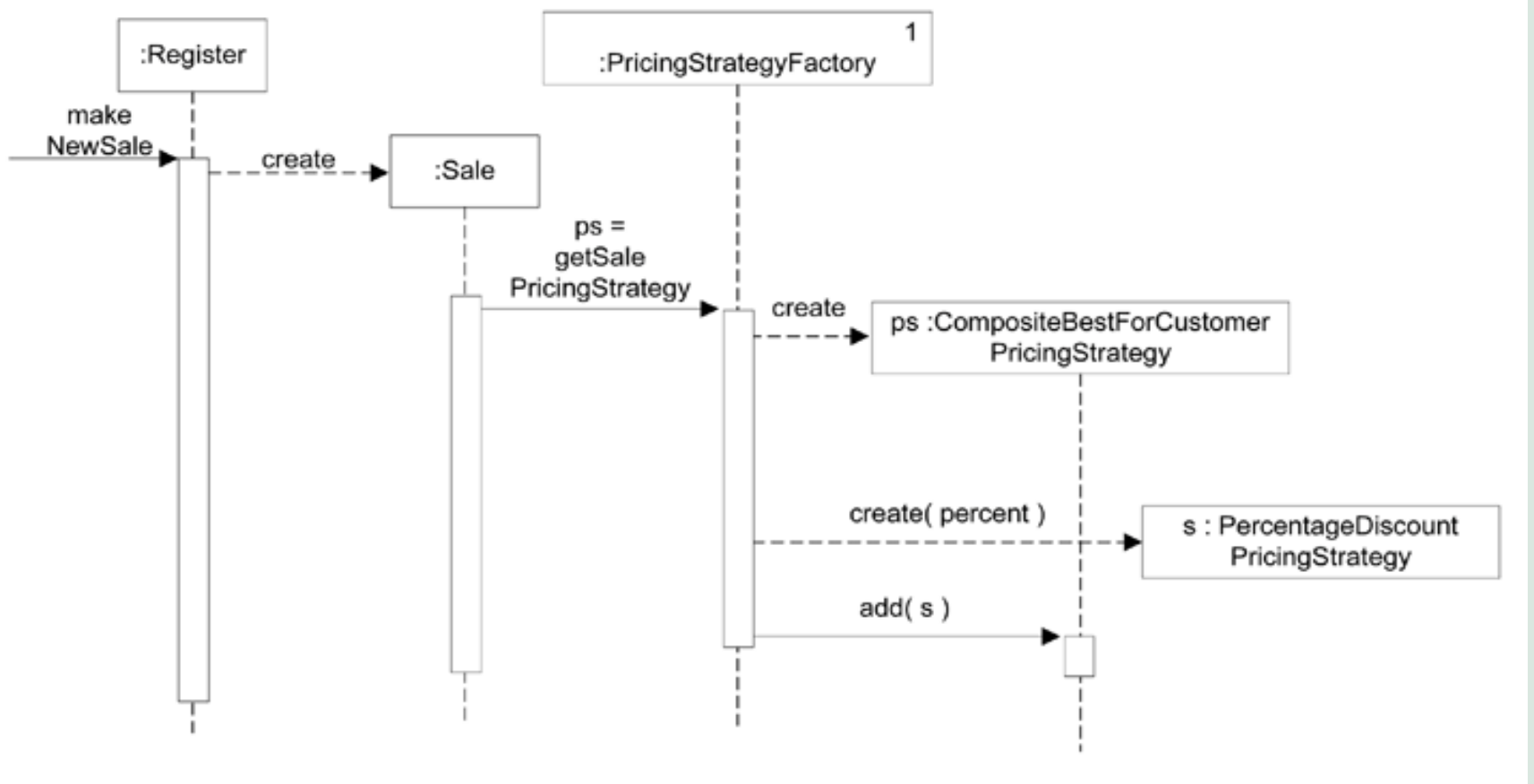
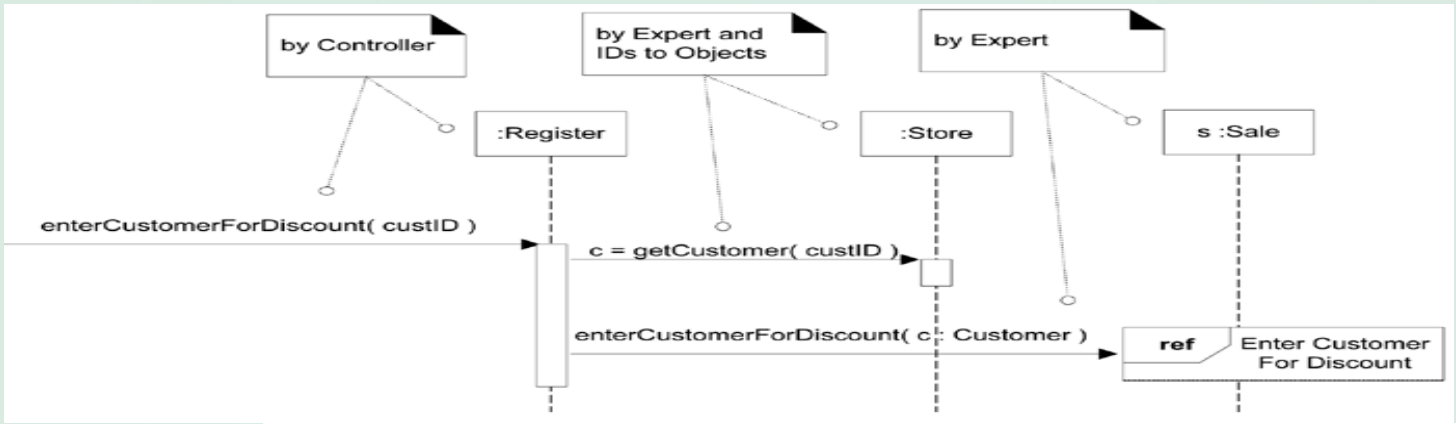
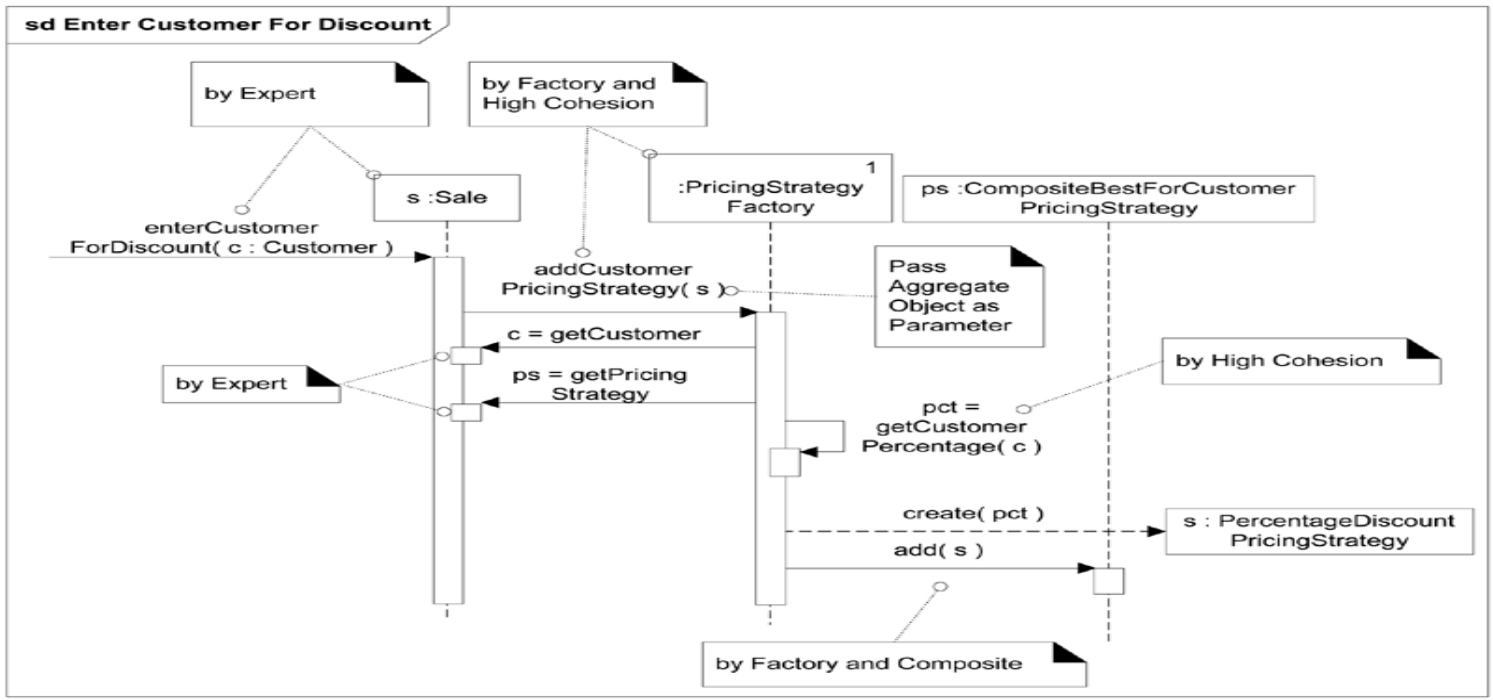


Figure 26.17. Creating a composite strategy

Composite (GoF) and Other Design Principles

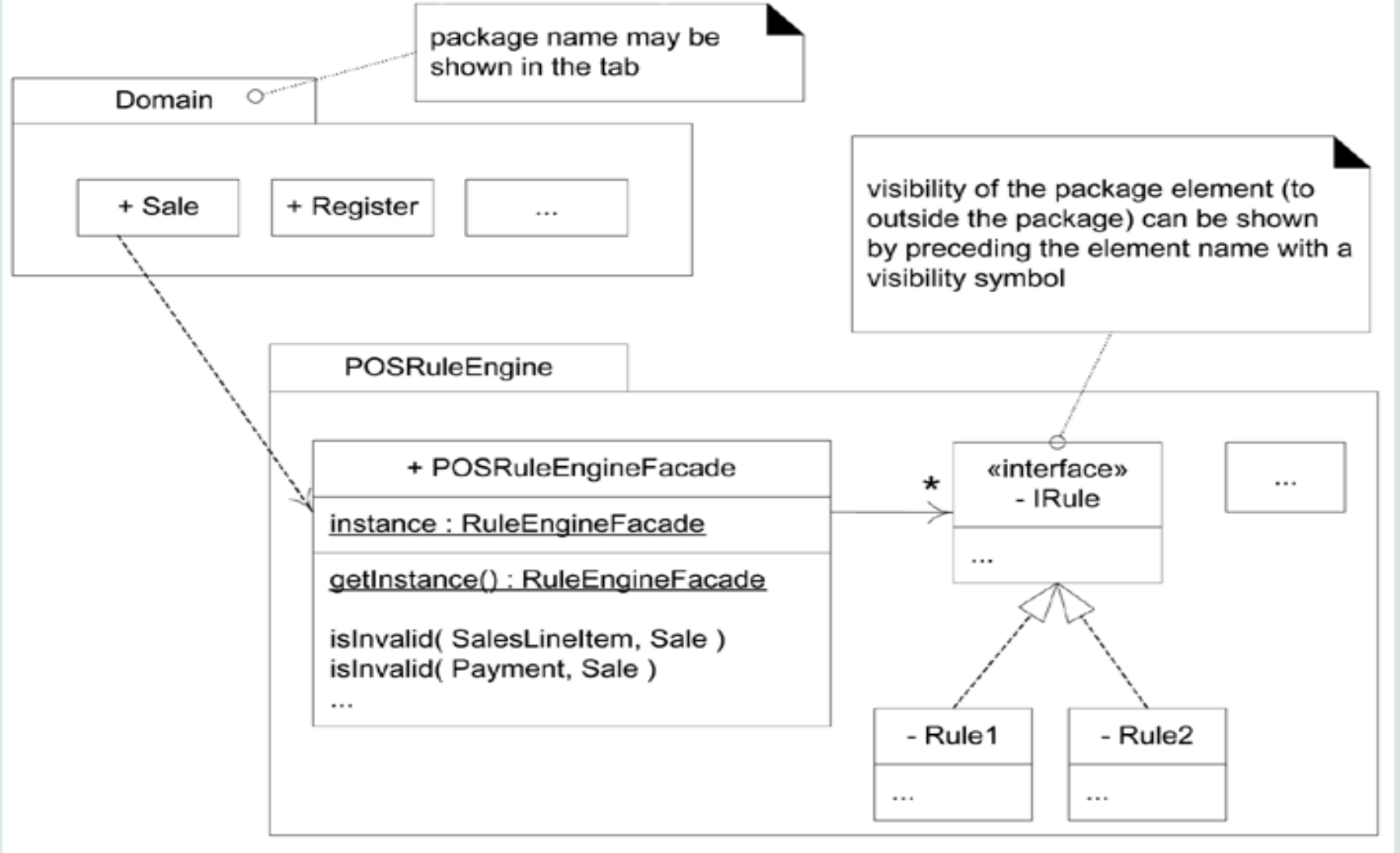




Facade (GoF)

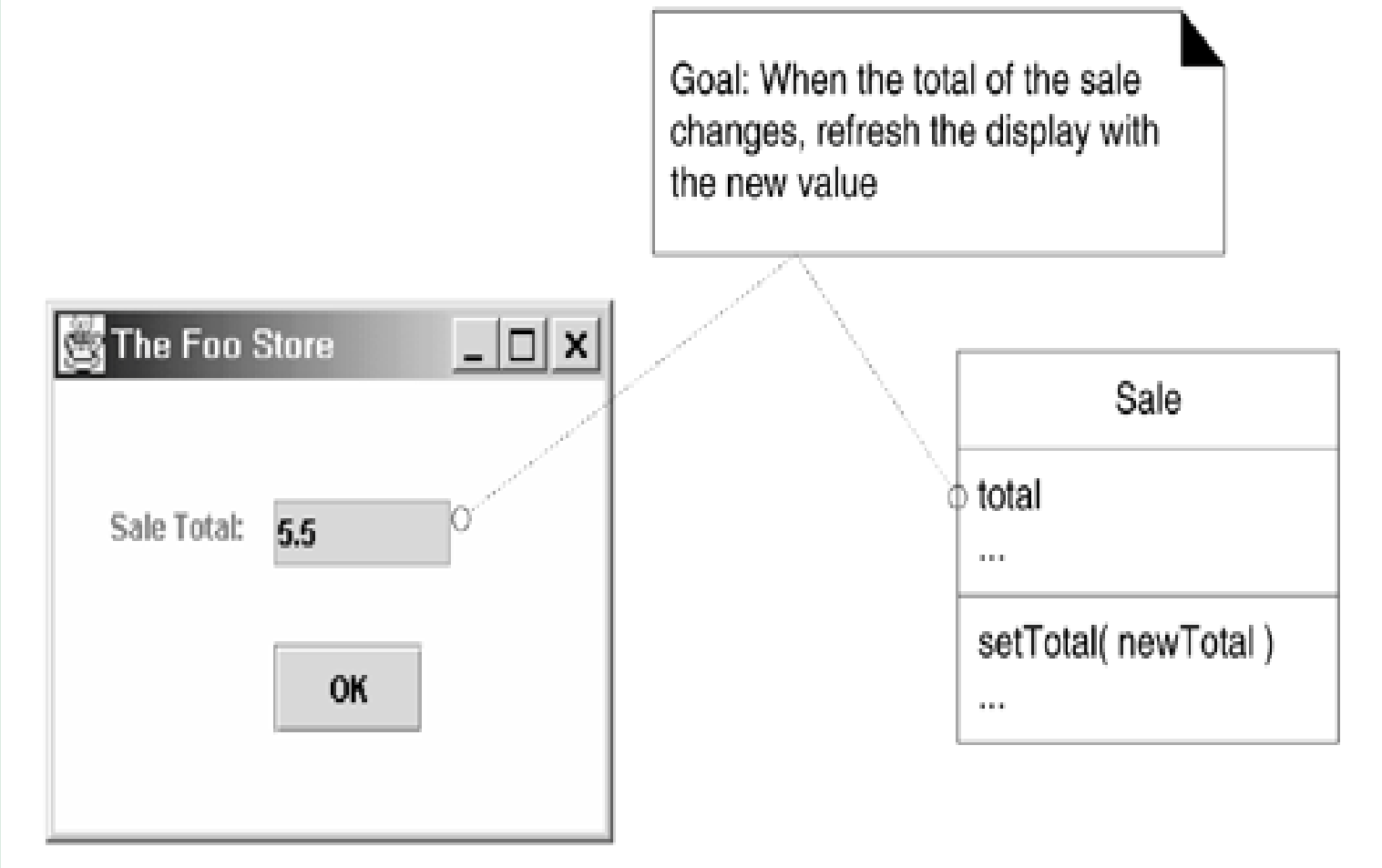
- Name: Facade
- Problem: A common, unified interface to a disparate set of implementations or interfaces such as within a subsystem is required. There may be undesirable coupling to many things in the subsystem, or the implementation of the subsystem may change. What to do?
- Solution: Define a single point of contact to the subsystem a facade object that wraps the subsystem. This facade object presents a single unified interface and is responsible for collaborating with the subsystem components.
- Summary: The Facade pattern is simple and widely used. It hides a subsystem behind an object.
- Related Patterns: Facades are usually accessed via the Singleton pattern.

Facade (GoF)


package name may be shown in the tab
Domain
+ Sale
+ Register
...
visibility of the package element (to outside the package) can be shown by preceding the element name with a visibility symbol
POSRuleEngine
+ POSRuleEngineFacade
instance : RuleEngineFacade
getInstance() : RuleEngineFacade
isInvalid(SalesLineItem, Sale)
isInvalid(Payment, Sale)
...
*
«interface»
- IRule
...
...
- Rule1
...
- Rule2
...

Observer/Publish-Subscribe/Delegation Event Model (GoF)

- Problem:



Observer/Publish-Subscribe/Delegation Event Model (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.
-

Observer/Publish-Subscribe/Delegation Event Model (GoF)

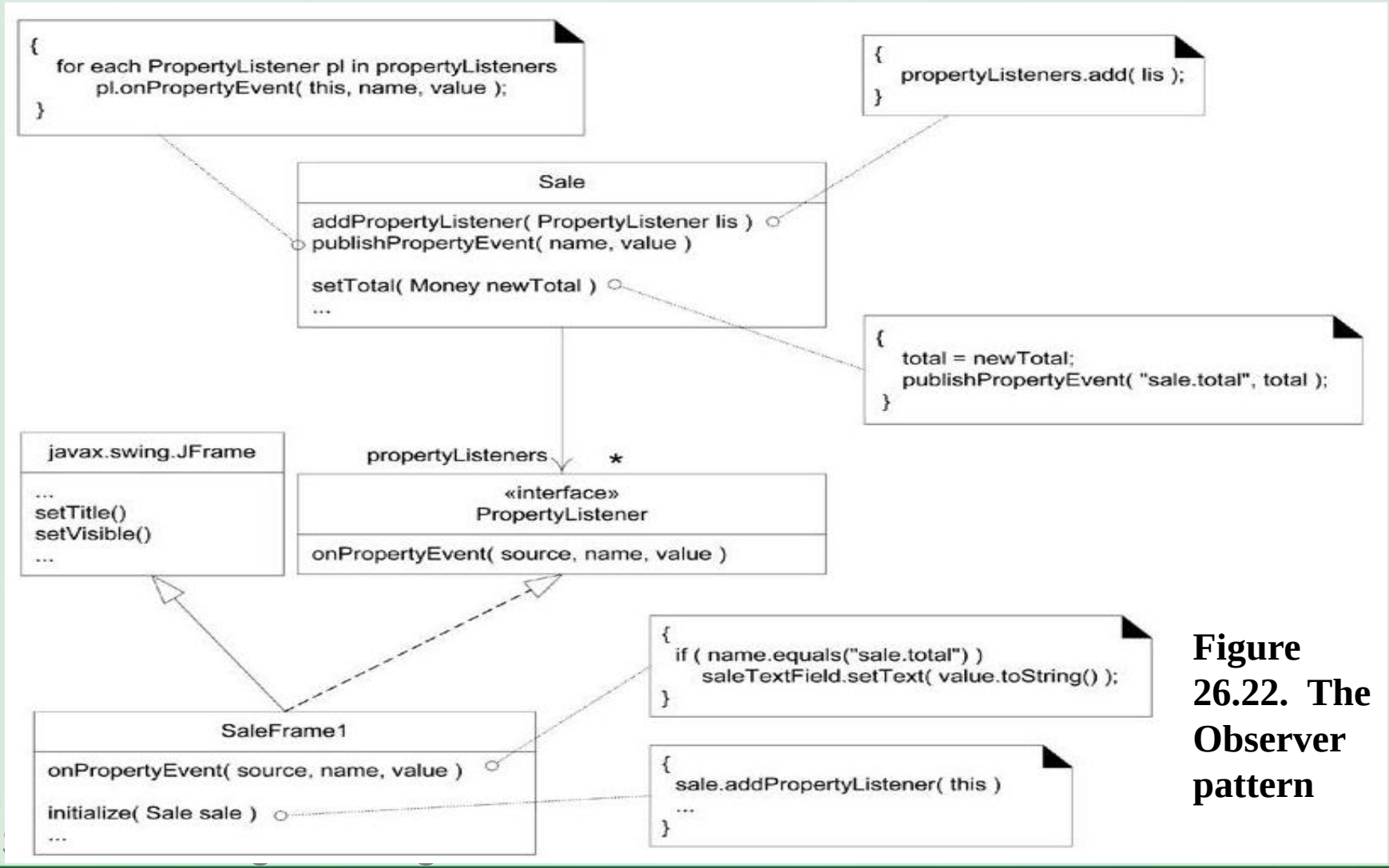


Figure 26.22. The Observer pattern

Observer/Publish-Subscribe/Delegation Event Model (GoF)

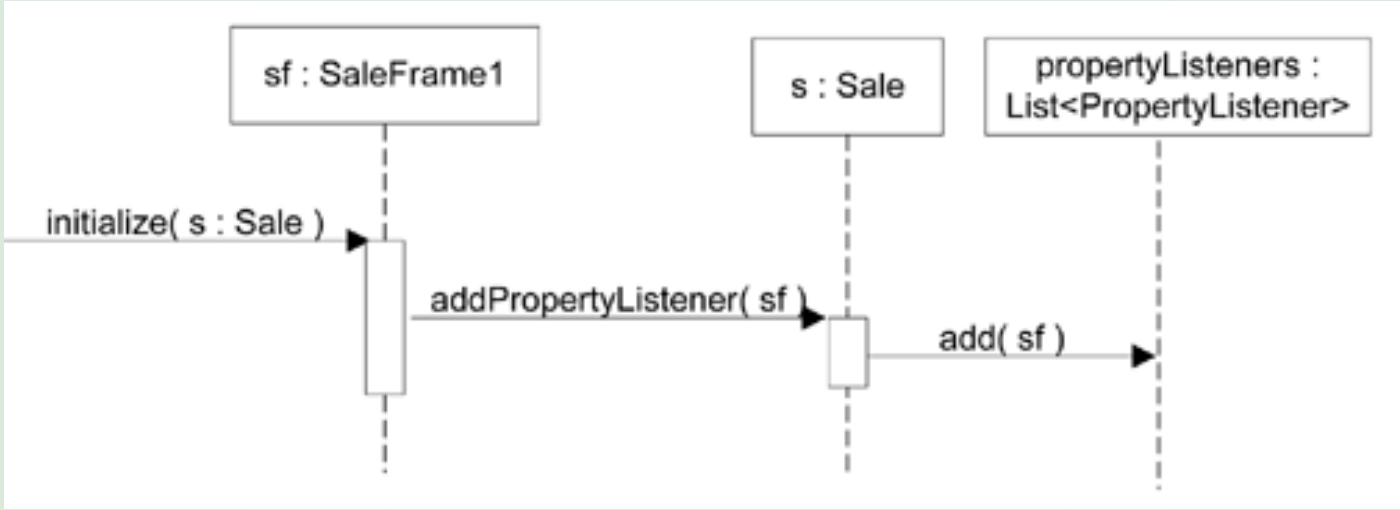


Figure 26.23. The observer SaleFrame1 subscribes to the publisher Sale

Figure 26.24. The Sale publishes a property event to all its subscribers

Observer/Publish-Subscribe/Delegation Event Model (GoF)

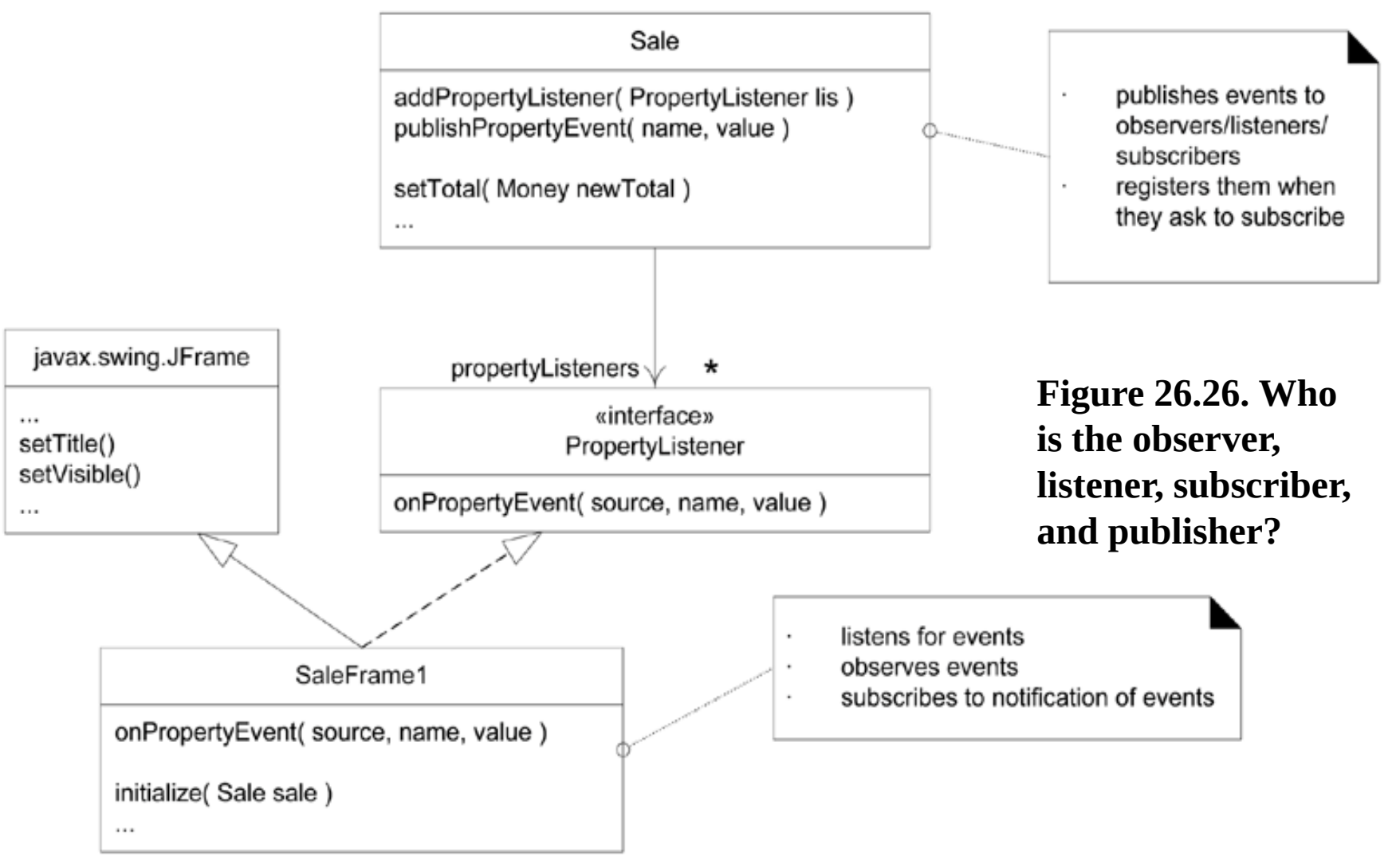


Figure 26.26. Who is the observer, listener, subscriber, and publisher?

Observer/Publish-Subscribe/Delegation Event Model (GoF)

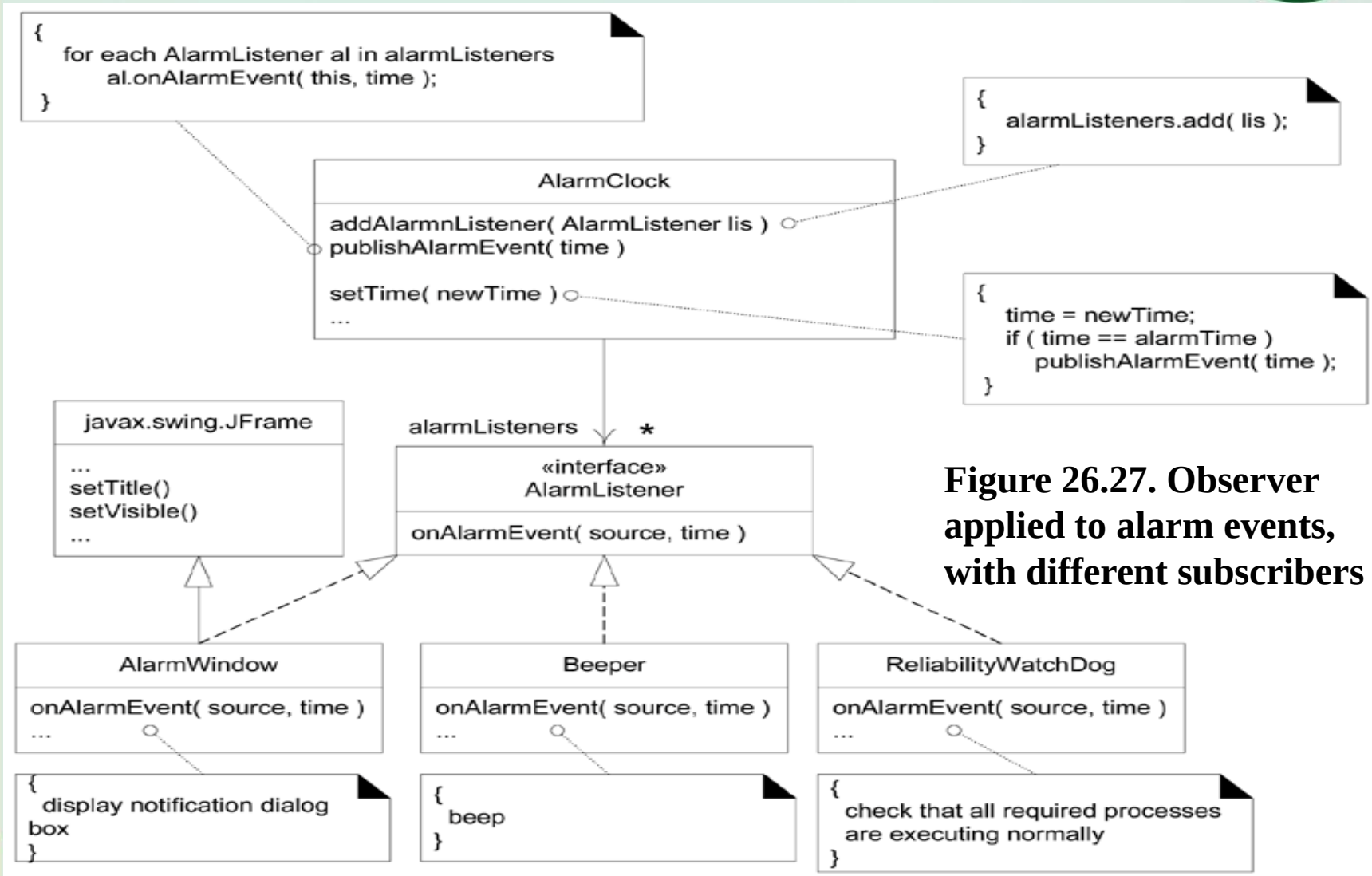


Figure 26.27. Observer applied to alarm events, with different subscribers