

SWEN30006

Software Modelling and Design

APPLYING GOF DESIGN PATTERNS

Larman Chapter 26

The shift of focus (to patterns) will have a profound and enduring effect on the way we write programs.

—Ward Cunningham and Ralph Johnson

Objectives

On completion of this topic you should be able to:

- ❑ Apply some GoF design patterns
 - Adapter
 - Factory (not GoF)
 - Singleton
 - Strategy
 - Composite
 - Façade
 - Observer
- ❑ Recognise GRASP principles as a generalization of other design patterns.

A Brief History of Design Patterns

- ❑ **1977:** Christopher Alexander publishes: “A Pattern Language: Towns, Buildings, Construction”
- ❑ **1987:** Cunningham and Beck used Alexander’s ideas to develop a small pattern language for Smalltalk
- ❑ **1990:** The Gang of Four (Gamma, Helm, Johnson and Vlissides) begin work compiling a catalog of design patterns
- ❑ **1991:** Bruce Anderson gives first Patterns Workshop at OOPSLA
- ❑ **1993:** Kent Beck and Grady Booch sponsor the first meeting of what is now known as the [Hillside Group](#)
- ❑ **1994:** First Pattern Languages of Programs (PLoP) conference
- ❑ **1995:** The Gang of Four (GoF) publish their ground breaking book: “Design Patterns: Elements of Reusable Software”

Patterns in Architecture

"The street cafe provides a unique setting, special to cities: a place where people can sit lazily, legitimately, be on view, and watch the world go by... Encourage local cafes to spring up in each neighborhood. Make them intimate places, with several rooms, open to a busy path, where people can sit with coffee or a drink and watch the world go by. Build the front of the cafe so that a set of tables stretch out of the cafe, right into the street."

— *Christopher Alexander et al., A Pattern Language, p. 437, 439*

Gang of Four Design Patterns

Creational Patterns (5):

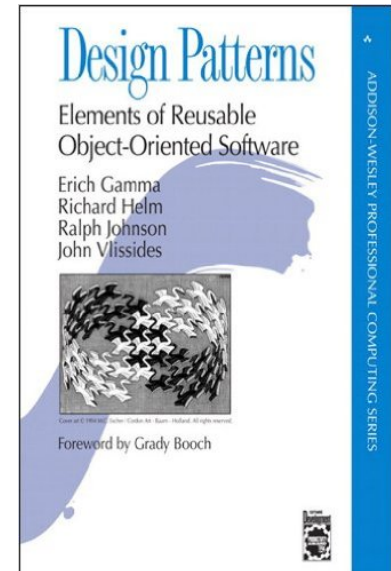
- ❑ abstract the object instantiation process.

Structural Patterns (7):

- ❑ describe how classes and objects can be combined to form larger structures.

Behavioural Patterns (11):

- ❑ are most specifically concerned with communication between objects.



GoF Design Pattern Template

- ❑ *Pattern Name and Classification:* A good , concise name for the pattern and the pattern's type.
- ❑ *Intent:* A short statement about what the pattern does.
- ❑ *Also Known As:* Other names for the pattern.
- ❑ *Motivation:* A scenario that illustrates where the pattern would be useful.
- ❑ *Applicability:* Situations where the pattern can be used.
- ❑ *Structure:* A graphical representation of the pattern.
- ❑ *Participants:* The classes and objects participating in the pattern.
- ❑ *Collaborations:* How to do the participants interact to carry out their responsibilities?
- ❑ *Consequences:* What are the pros and cons of using the pattern?
- ❑ *Implementation:* Hints and techniques for implementing the pattern.
- ❑ *Sample Code:* Code fragments for a sample implementation.
- ❑ *Known Uses:* Examples of the pattern in real systems.
- ❑ *Related Patterns:* Other patterns that are closely related to the pattern.

1. External Services with Varying Interfaces

NextGen POS (tax calc's, credit auth. services, ...)

- ❑ “To handle this problem, let's use Adapters generated from a Singleton Factory.”
- ❑ Design reasoning based on:
 - Controller
 - Creator
 - Protected Variations
 - Low Coupling
 - High Cohesion
 - Indirection
 - Polymorphism
 - Adaptor
 - Factory
 - Singleton

Adapter (GoF)

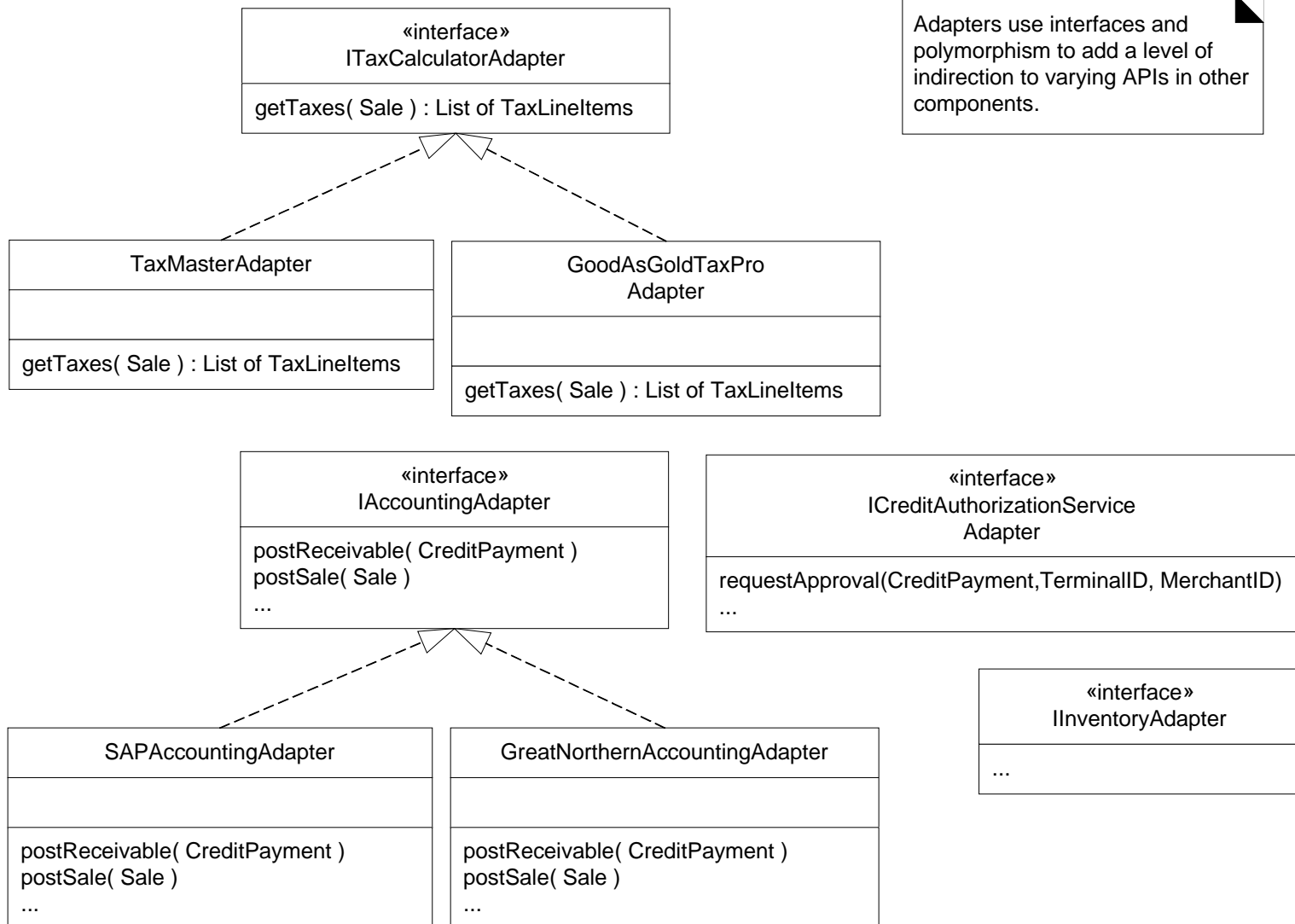
Problem:

- ❑ How to resolve incompatible interfaces, or provide a stable interface to similar components with different interfaces?

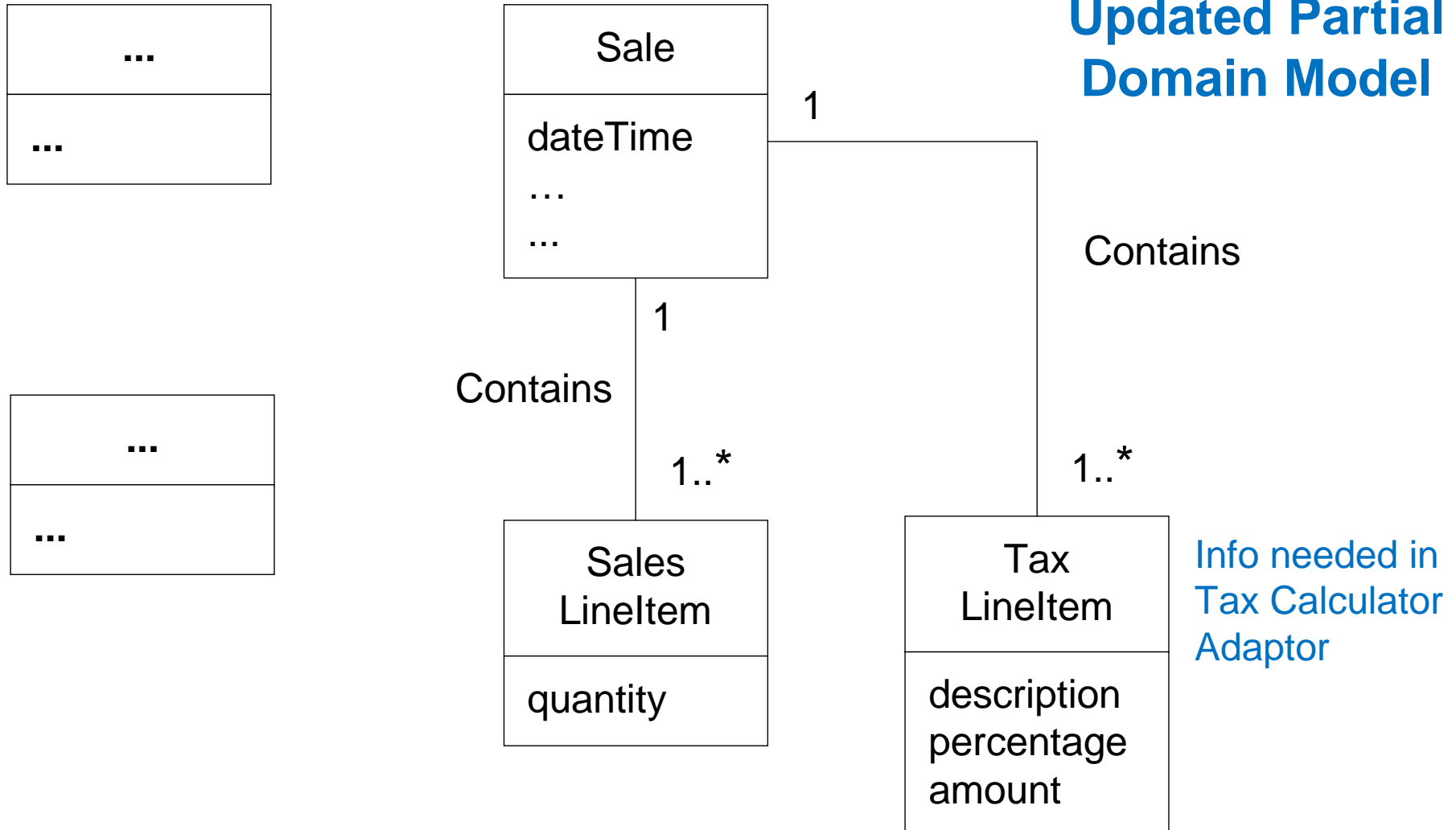
Solution (advice):

- ❑ Convert the original interface of a component into another interface, through an intermediate adapter object.

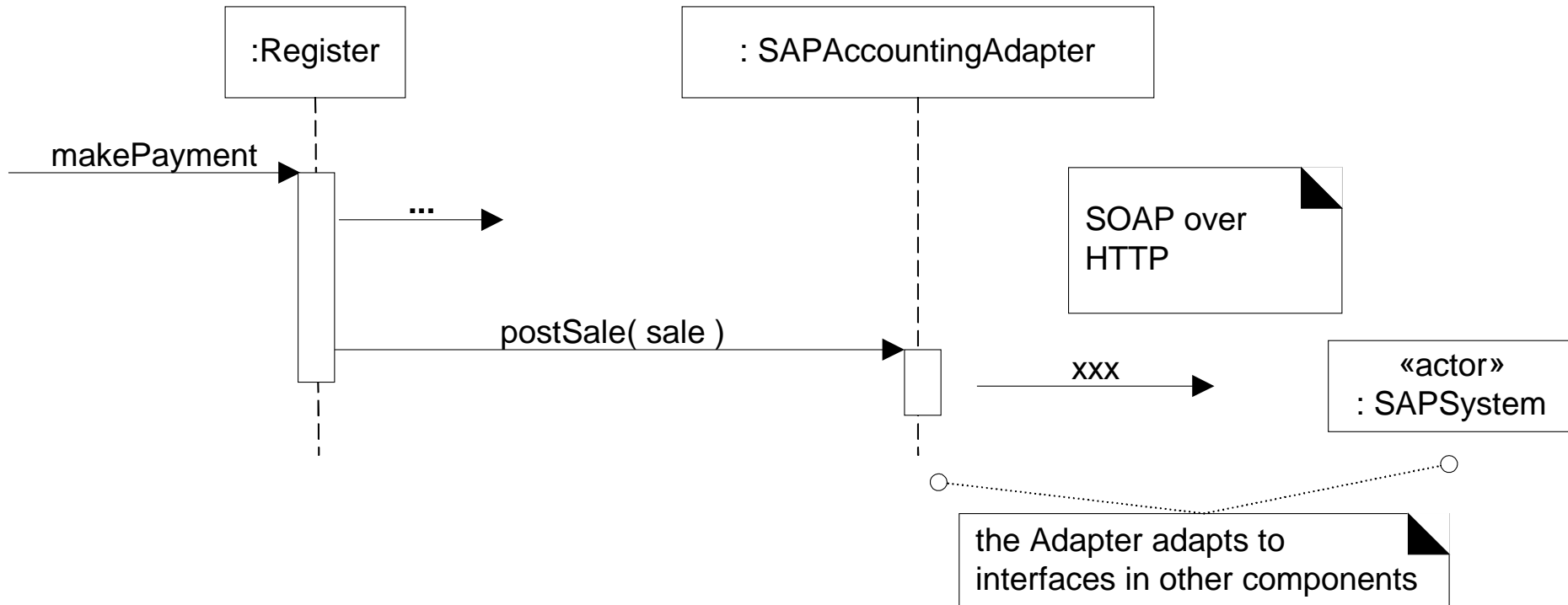
The Adapter Pattern



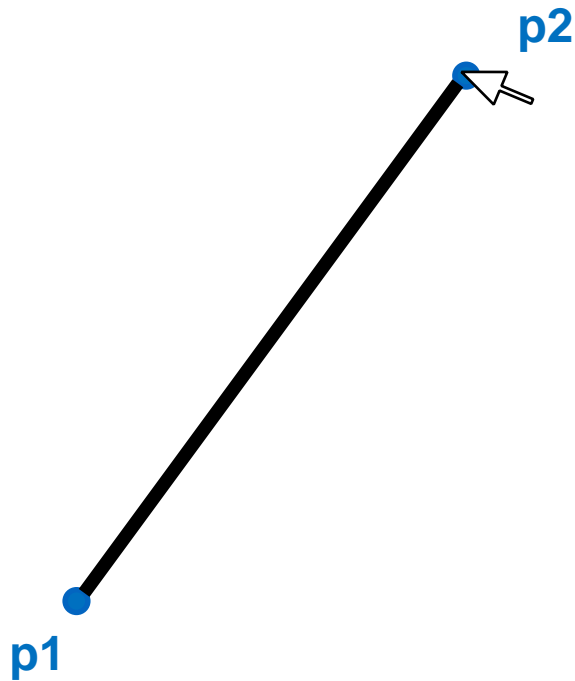
“Analysis” Discovery in Design



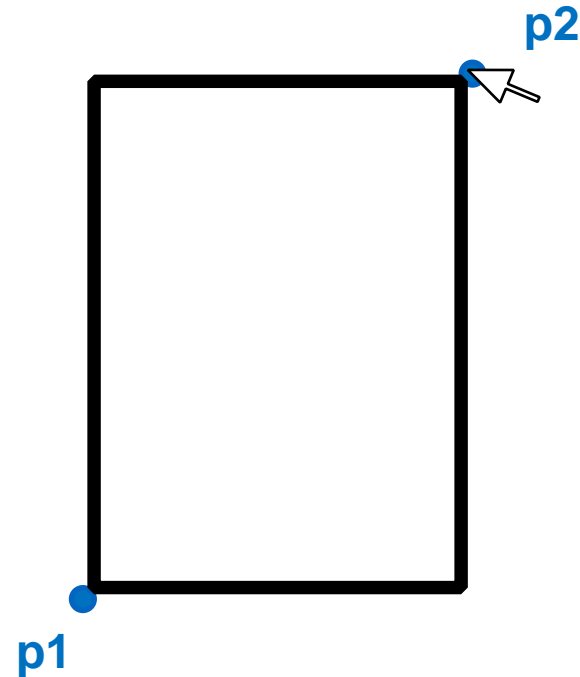
Using an Adapter



Example: Shape Drawing



Line



Rectangle

Example adapted from [SourceMaking Adapter example](#)!

Shape Drawing Library

```
class Point {  
    public int x, y;  
  
    public String toString() { return "(" + x + "," + y + " "; }  
}
```

```
class LineGraphic  
{  
    public void drawLine(Point p1, Point p2) {  
        // Stub for testing replacing real library call  
        System.out.println("line from " + p1 + " to " + p2);  
    }  
}
```

line from (10,20) to (30,60)

rectangle at (10,20) with width 20 and height 40

```
class RectangleGraphic  
{  
    public void drawRect(Point p, int width, int height)  
    {  
        // Stub for testing replacing real library call  
        System.out.println("rectangle at " + p + " with width " + width  
            + " and height " + height);  
    }  
}
```

Testing using End-points

```
public class testDirect {  
  
    void test() {  
        Object[] shapes = { new LineGraphic(), new RectangleGraphic() };  
  
        // A begin and end point from a graphical editor  
        Point p1 = new Point(10, 20);  
        Point p2 = new Point(30, 60);  
        for (int i = 0; i < shapes.length; ++i)  
            if (shapes[i].getClass().getName().equals("Line"))  
                ((LineGraphic) shapes[i]).drawLine(p1, p2);  
            else if (shapes[i].getClass().getName().equals("Rectangle"))  
                ((RectangleGraphic) shapes[i]).drawRect(  
                    new Point(Math.min(p1.x, p2.x), Math.min(p1.y, p2.y)),  
                    Math.abs(p2.x - p1.x), Math.abs(p2.y - p1.y)  
                );  
    }  
}
```

line from (10,20) to (30,60)

rectangle at (10,20) with width 20 and height 40

Introducing Adapters

```
interface iShapeAdaptor
{
    void draw(Point p1, Point p2);
}

class iLineAdapter implements iShapeAdaptor
{
    private LineGraphic adaptee = new LineGraphic();
    public void draw(Point p1, Point p2) { adaptee.drawLine(p1, p2); }
}

class iRectangleAdapter implements iShapeAdaptor
{
    private RectangleGraphic adaptee = new RectangleGraphic();
    public void draw(Point p1, Point p2)
    {
        adaptee.drawRect(new Point(Math.min(p1.x, p2.x), Math.min(p1.y, p2.y)),
            Math.abs(p2.x - p1.x), Math.abs(p2.y - p1.y));
    }
}
```

Testing with Adapters ()

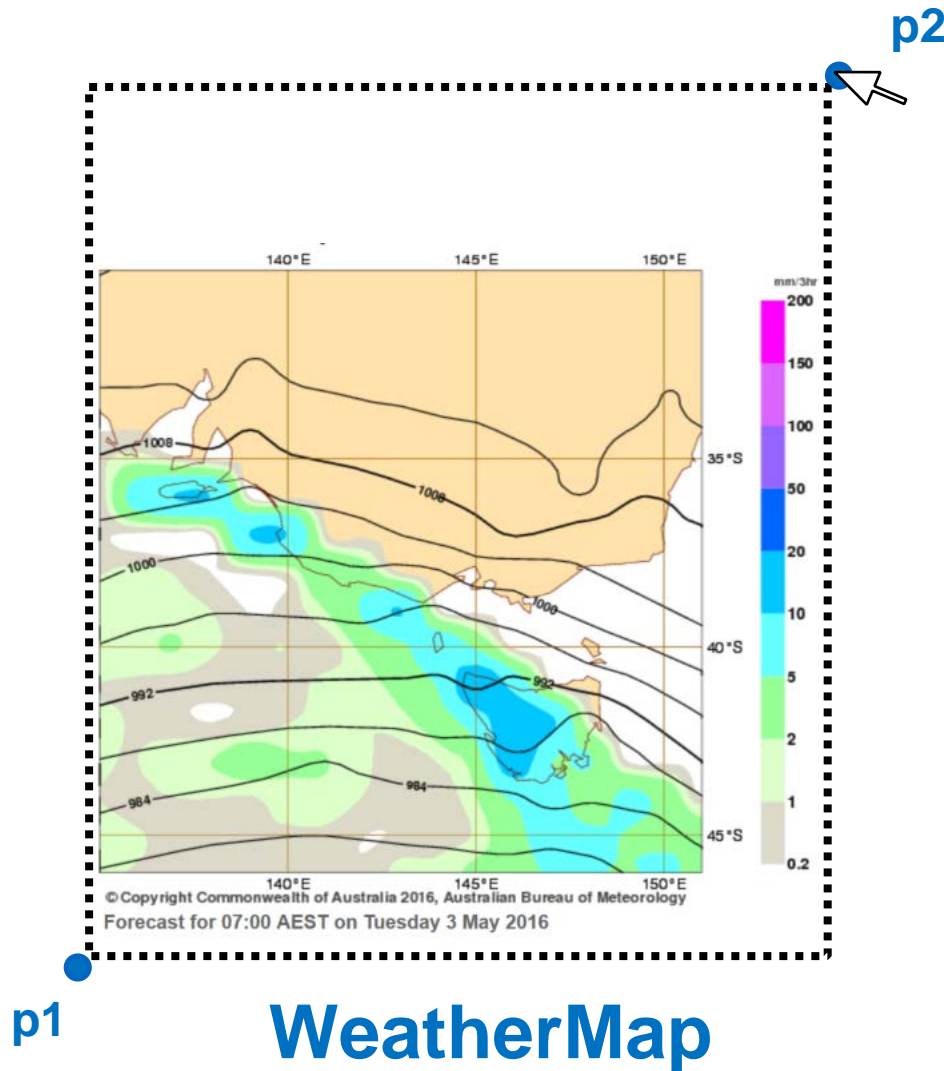
```
public class testAdapter {  
  
    void test() {  
        iShapeAdapter[] shapes = {  
            new iLineAdapter(),  
            new iRectangleAdapter(),  
            new iWeatherMapAdapter()  
        };  
  
        // A begin and end point from a graphical editor  
        Point p1 = new Point(10, 20);  
        Point p2 = new Point(30, 60);  
        for (int i = 0; i < shapes.length; ++i)  
            shapes[i].draw(p1, p2);  
    }  
}
```

line from (10,20) to (30,60)

rectangle at (10,20) with width 20 and height 40

weathermap for 3/5/2016 8:25am at (10,20) scaled to width 20

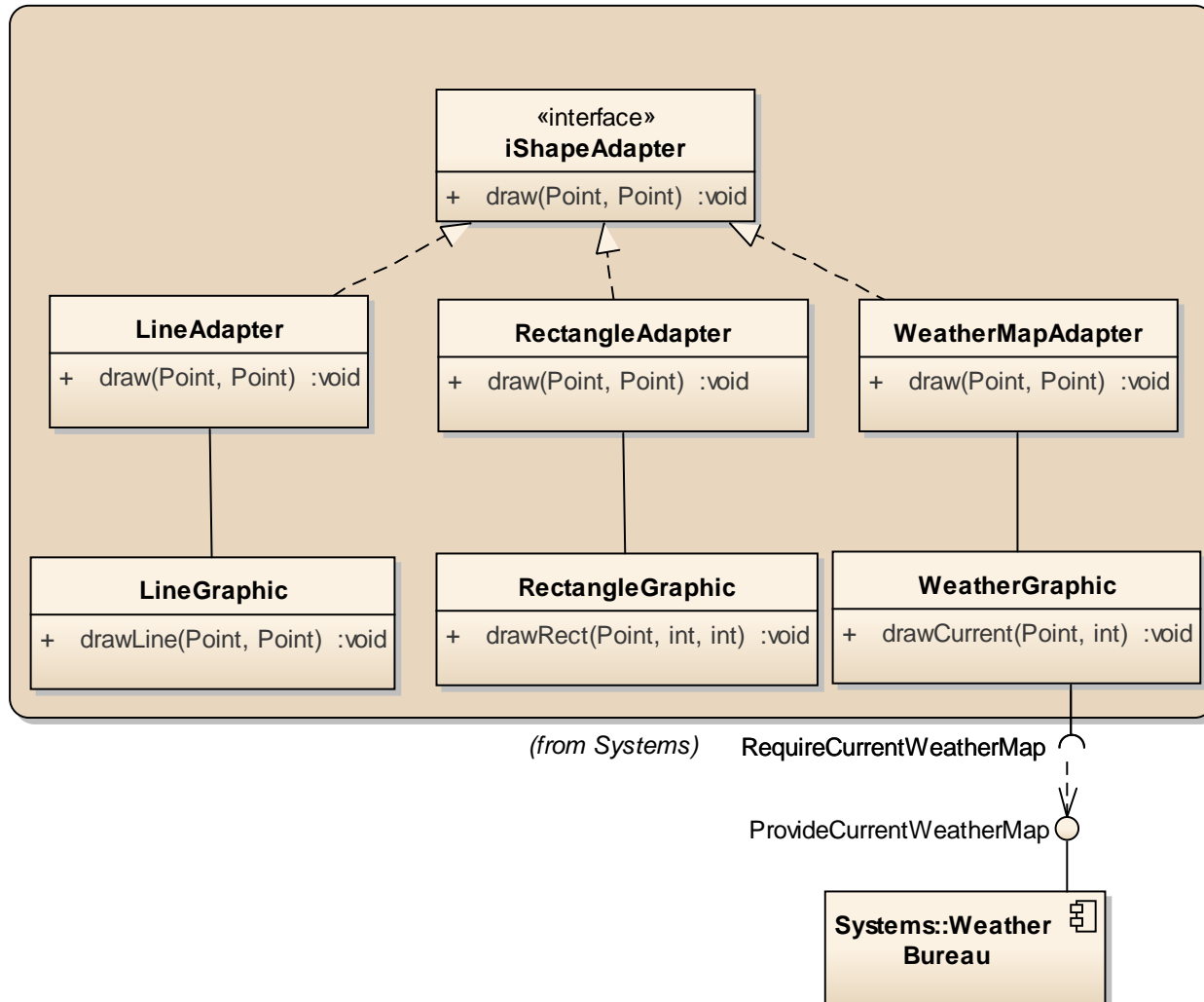
Example: Shape Drawing (2)



WeatherMap Adapter

```
class iWeatherMapAdapter implements iShapeAdapter
{
    private WeatherMapGraphic adaptee = new WeatherMapGraphic();
    public void draw(Point p1, Point p2) {
        int length = Math.abs(p2.x - p1.x);
        Point p = new Point(Math.min(p1.x, p2.x), Math.min(p1.y, p2.y));
        adaptee.drawCurrent(length, p);
    }
}
```

Adapter: Component Hiding



Component/Service Interfaces

Example: [The Open Movie Database API](#)

Adaptor: Related Patterns

- ❑ Façade (GoF) wraps access to a subsystem or system with a single object
 - C.f. Adaptor which requires polymorphism to deal with varying external systems
- ❑ GRASP: Adaptor supports *Protected Variations* w.r.t. changing external interfaces or 3rd-party packages through the use of *Indirection* object that applies interfaces and *Polymorphism*.

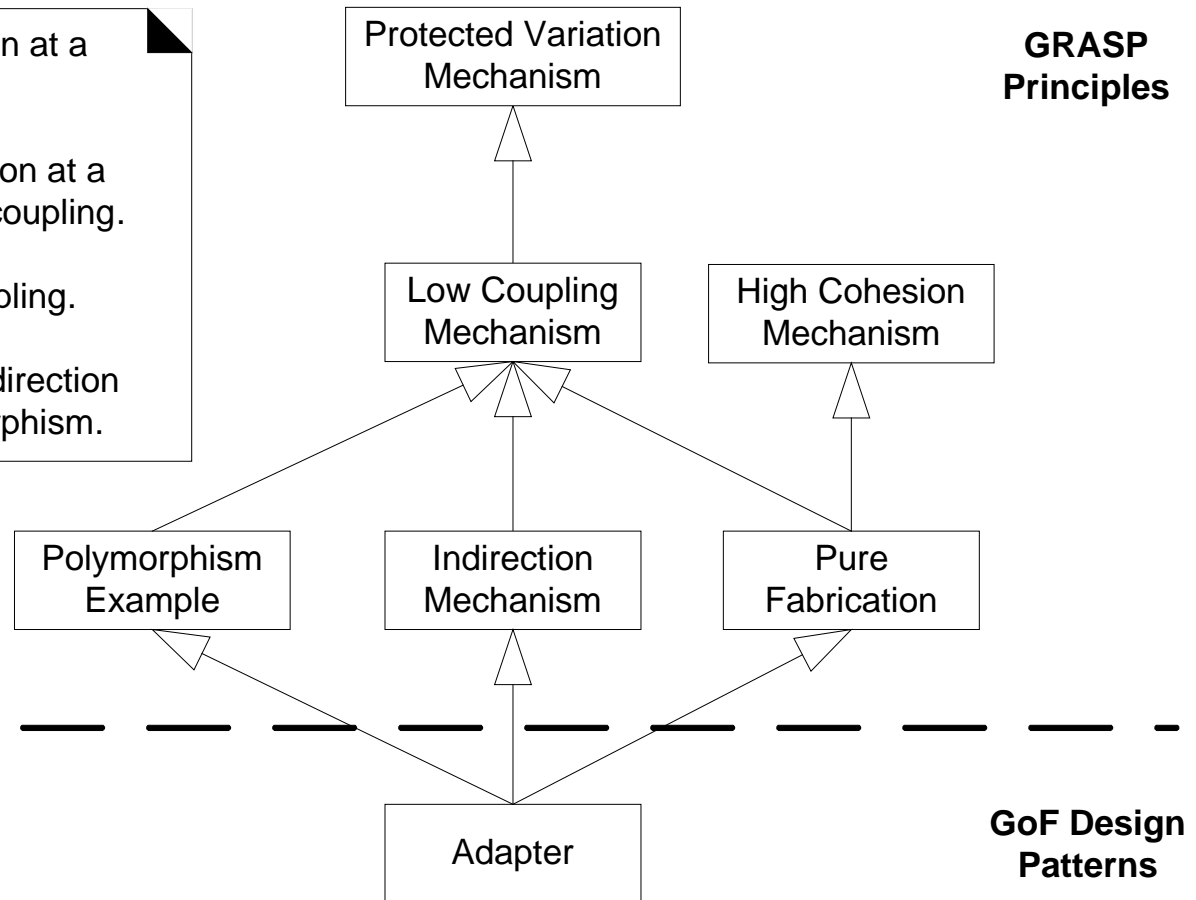
Adapter in relation to GRASP Principles

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.



Issues arising from Adapter

- ❑ Who creates the adapters?
- ❑ Which class of adapter should be created?
 - TaxMasterAdapter or GoodAsGoldTaxProAdapter

Partial Answer:

- ❑ Separation of Concerns/High Cohesion
 - Creating adapters is not pure application logic
 - Not a domain object
 - → Pure Fabrication

Factory (not GoF)

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.

Factory (not GoF) cont.

Aka **Simple Factory** or **Concrete Factory**

- ❑ Simplified GoF Abstract Factory pattern

Advantages:

- ❑ Separate responsibility of complex creation into cohesive helper objects.
- ❑ Hide potentially complex creation logic.
- ❑ Allow introduction of performance-enhancing memory management strategies, such as object caching or recycling.

The Factory Pattern

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

Properties File Entry

! taxcalculator.class.name=TaxMasterAdaptor
taxcalculator.class.name=GoodAsGoldTaxProAdaptor

```
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;
```

Issues arising from Factory

- ❑ Who creates the factory?
- ❑ How is it accessed?
 - Different adaptors (tax, inventory, ...):
access required in various places

Partial Answer:

- ❑ Only one instance of the factory is needed
- ❑ Where required, pass through to methods or initialise objects with a ref? No.
- ❑ → Single access point through global visibility

Singleton (GoF)

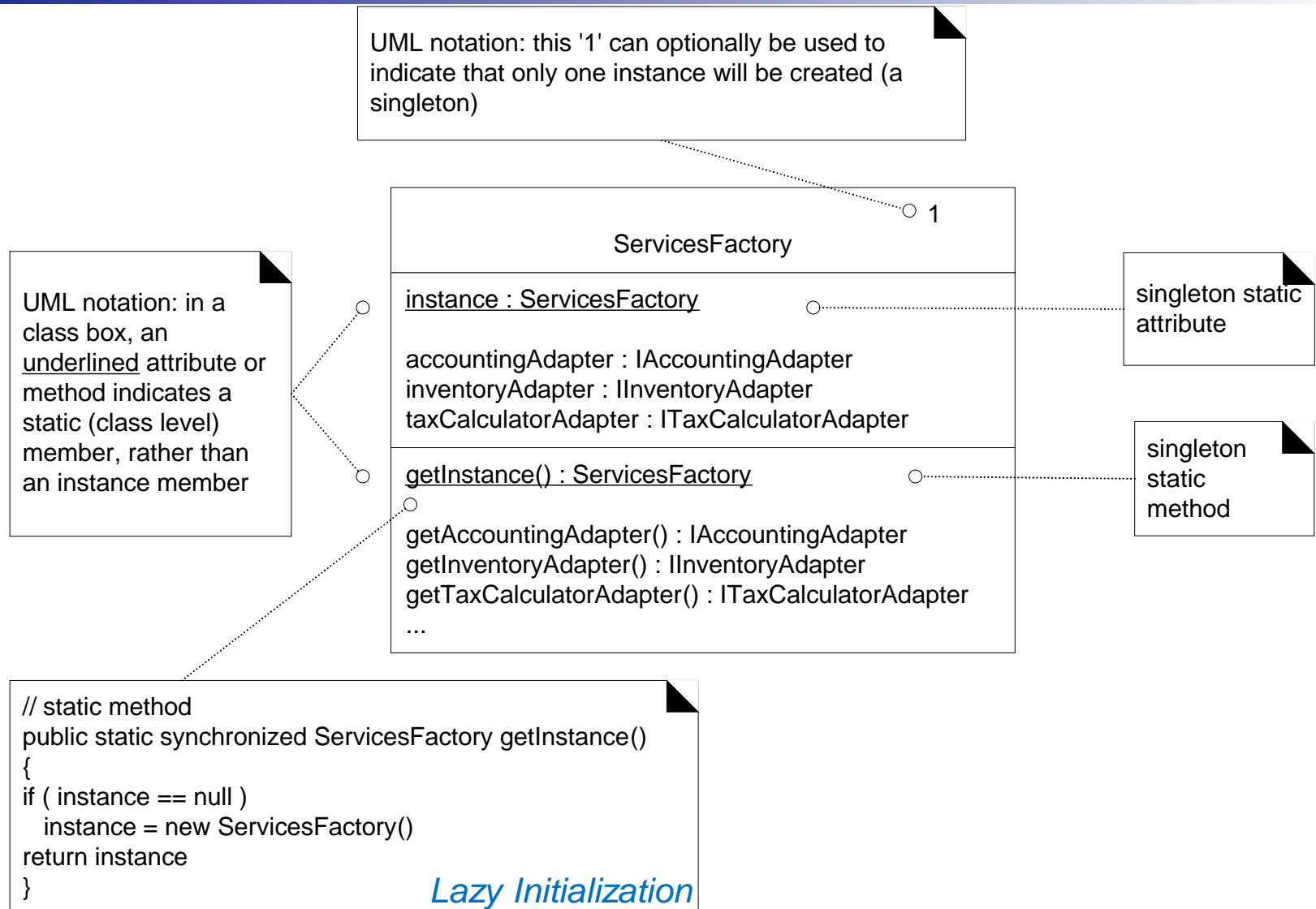
Problem:

- ❑ 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 Pattern in *ServicesFactory* Class



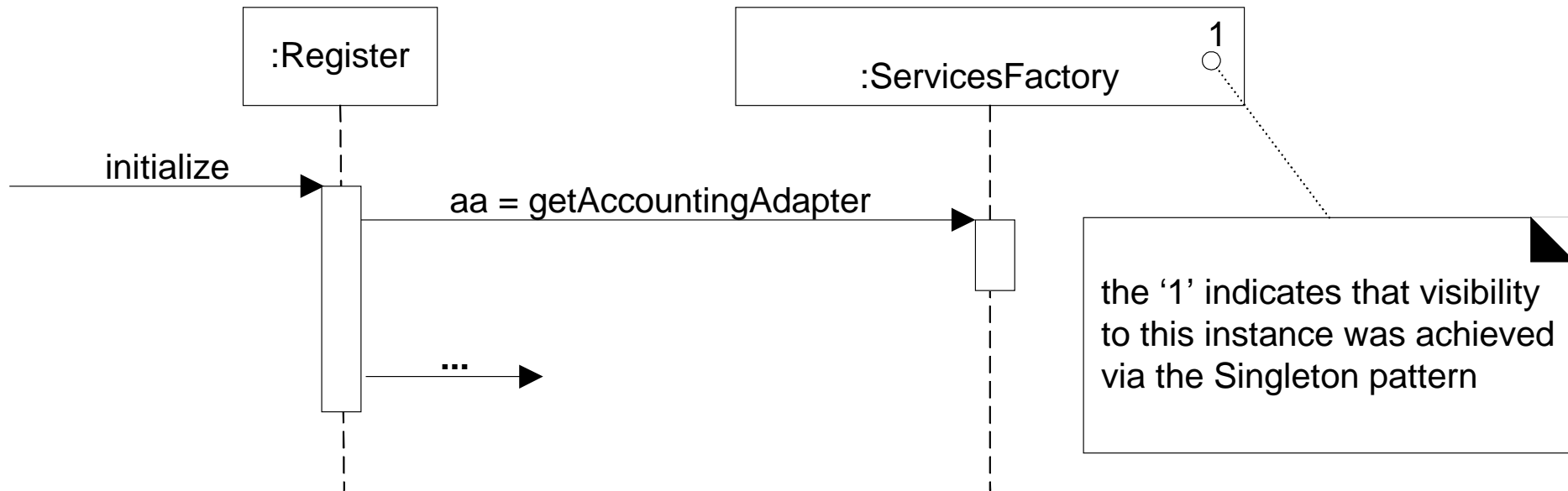
Accessing the Singleton Factory

```
public class Register
{
    public void initialize()
    {
        ... do some work ...
        // accessing the singleton Factory via the getInstance call
        accountingAdapter =
            ServicesFactory.getInstance().getAccountingAdapter();
        ... do some work ...
    }
    // other methods...
} // end of class
```



SingletonClass.getInstance()
is globally visible

Implicit *getInstance* Singleton Pattern Message



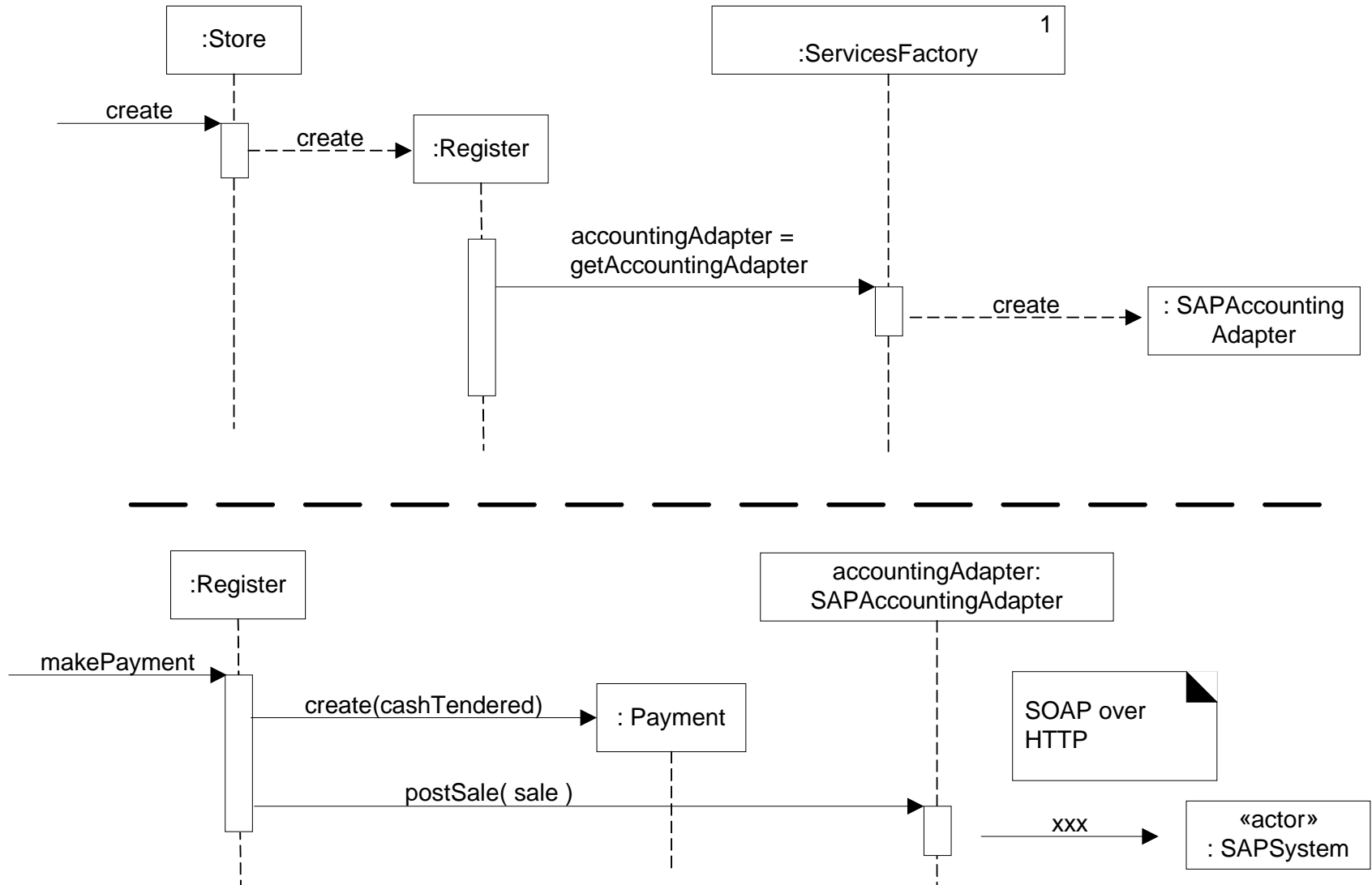
Static: why some and not all?

Why not aren't all Singleton service methods *static*?

E.g. static `getAccountingAdapter()`: ...

- ❑ May want subclasses: static methods are not polymorphic (virtual); no overriding in most languages
- ❑ Most remote communication mechanisms (e.g. Java's RMI) don't support remote-enabling of static methods.
- ❑ A class is not always a singleton in all application contexts.

Adapter, Factory, and Singleton Patterns



2. NextGen Complex Pricing Logic

- ❑ Provide more complex pricing logic, e.g.
 - store-wide discount for the day
 - senior citizen discounts
- ❑ The pricing strategy for a sale can vary, e.g.
 - one period it may be 10% off all sales
 - later it may be \$10 off if the sale total is greater than \$200
- ❑ “To handle this problem, let’s use a Composite Strategy.”

Strategy (GoF)

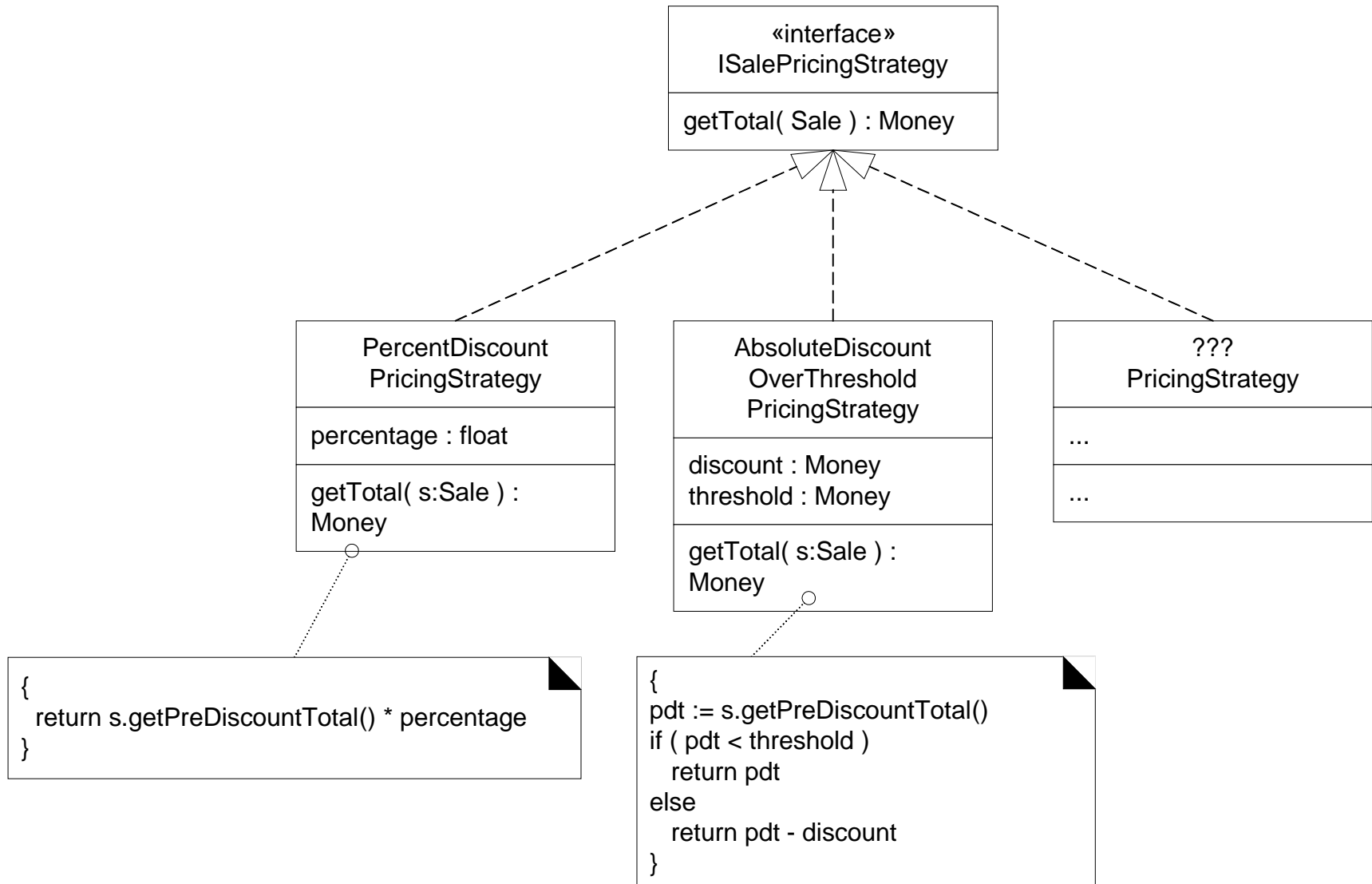
Problem:

- ❑ How to design for varying, but related, algorithms or policies?
- ❑ How to design for the ability to change these algorithms or policies?

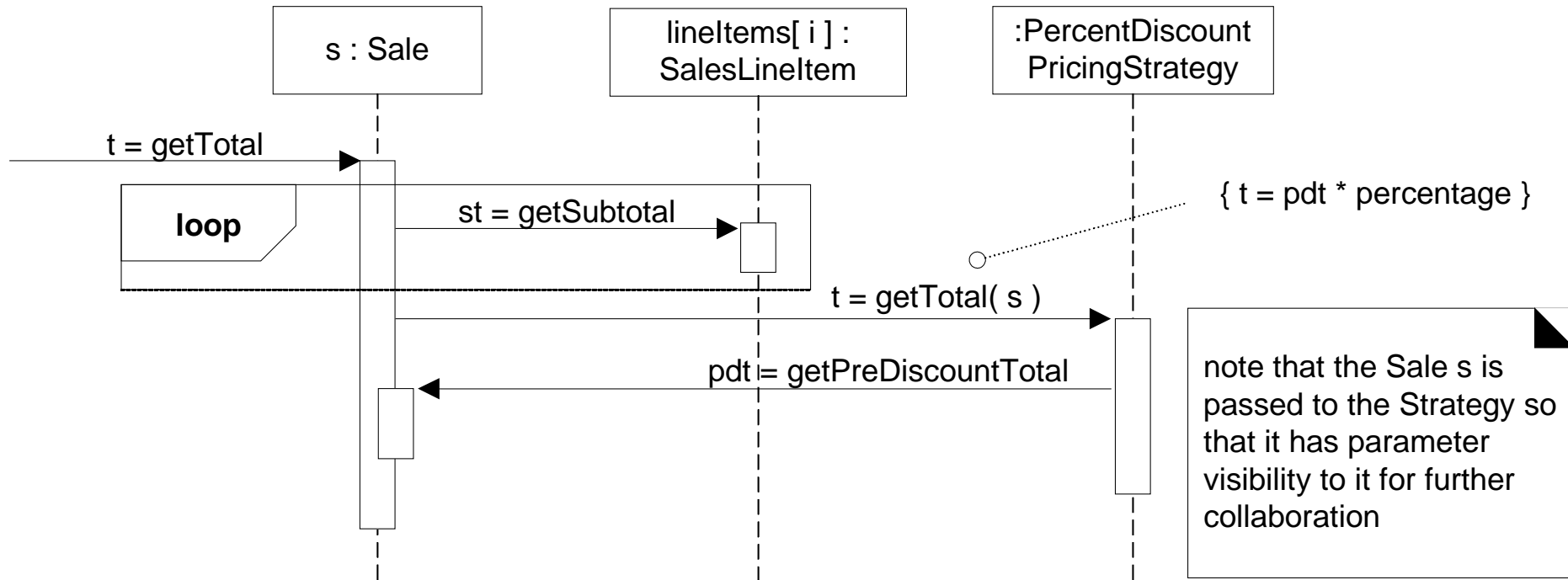
Solution (advice):

- ❑ Define each algorithm/policy/strategy in a separate class, with a common interface.

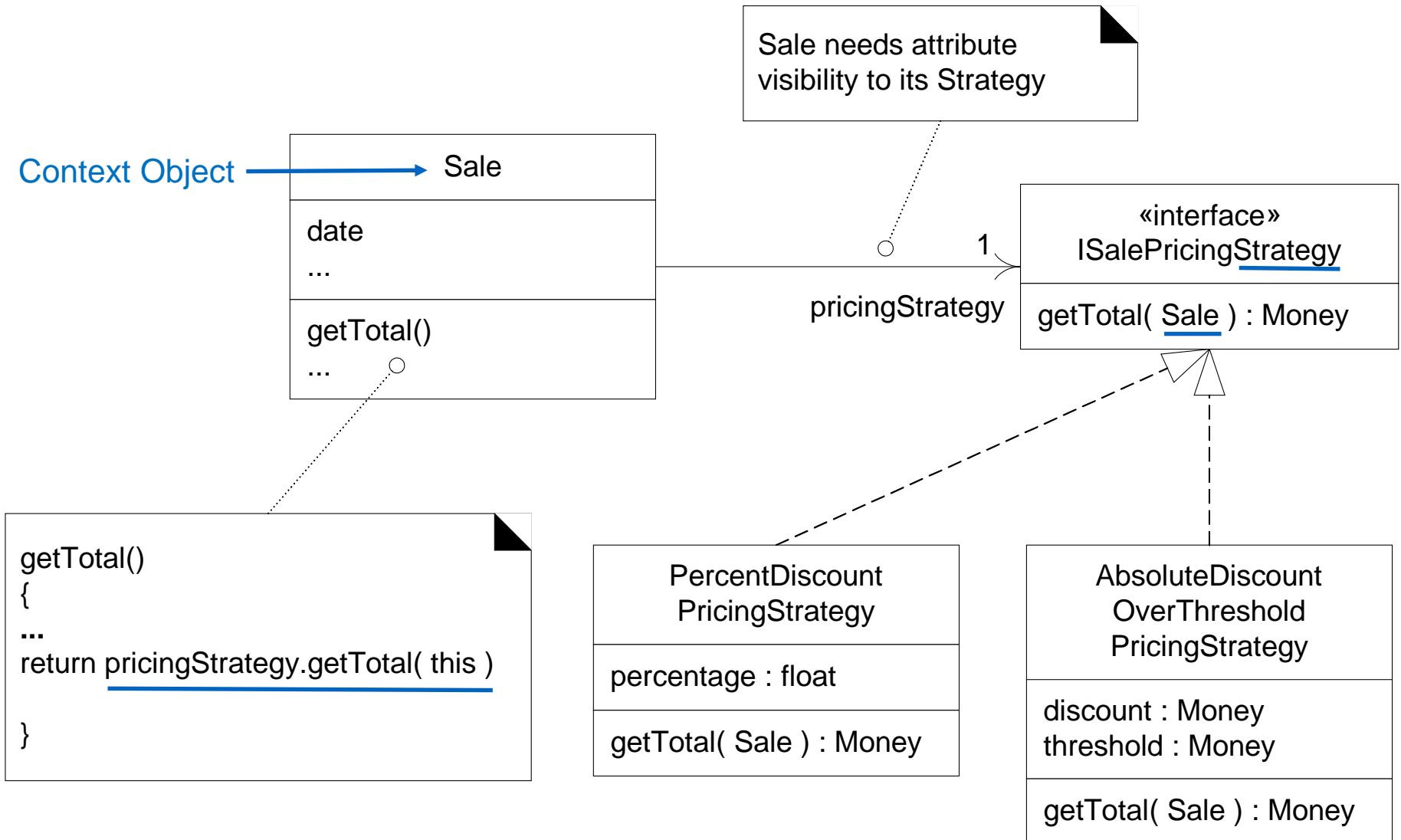
Pricing Strategy Classes



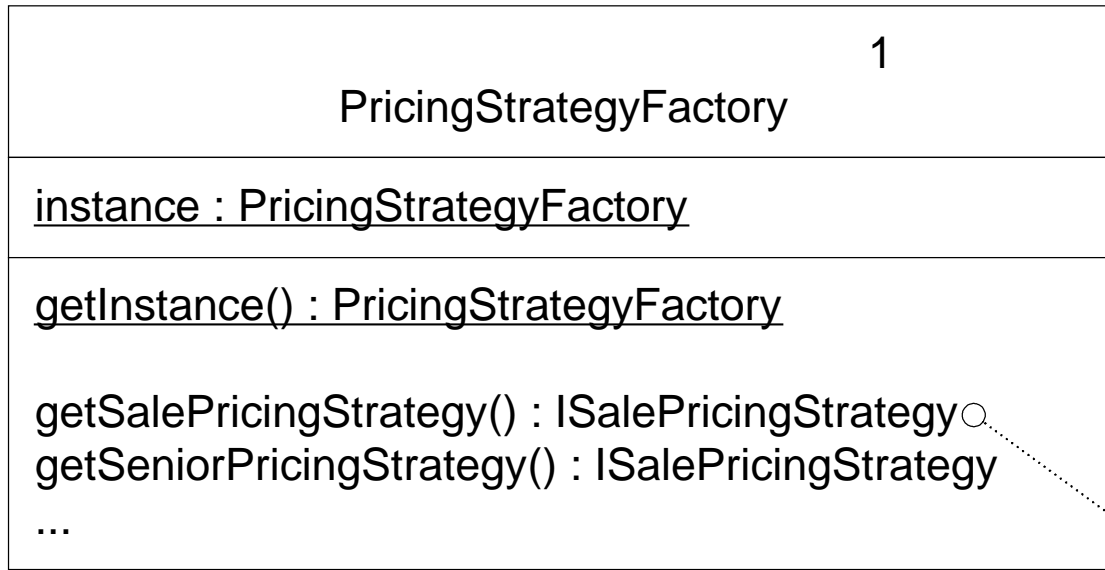
Strategy in Collaboration



Context Object Visibility to Strategy

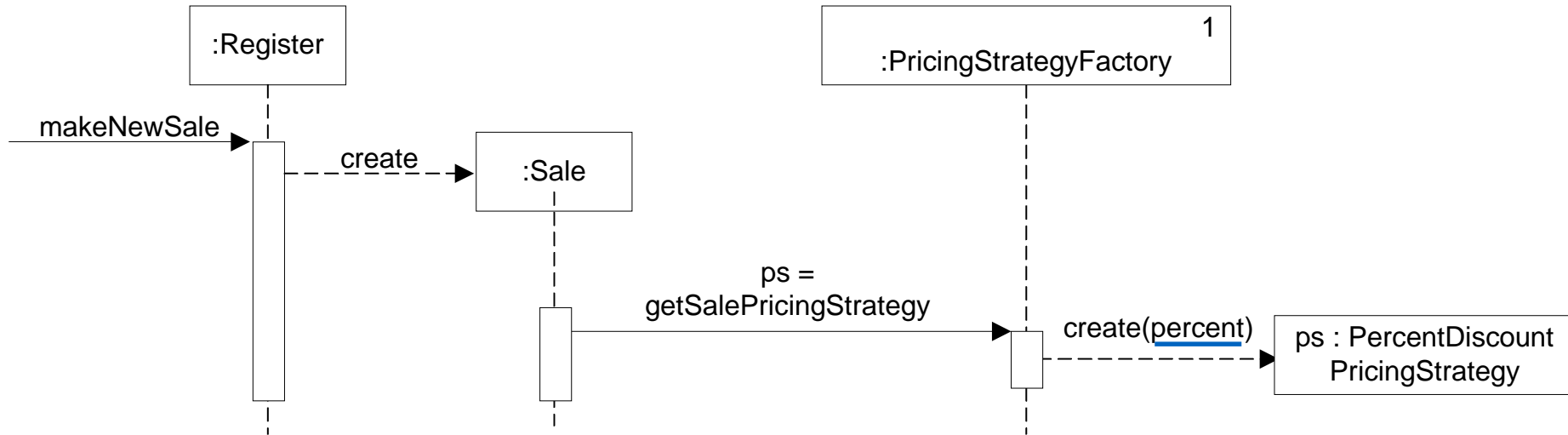


Creation: Factory for Strategies



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

Creating a Strategy



Handling Multiple Conflicting Policies

E.g. suppose the stores pricing today (Monday) is:

- ❑ 20% senior discount policy
- ❑ preferred customer discount: 15% off sales over \$400
- ❑ on Monday, there is \$50 off purchases over \$500
- ❑ buy 1 case Darjeeling tea, get 15% discount off everything

Factors:

1. customer type (senior, preferred)
2. time period (Monday)
3. line item product (Darjeeling tea)

Combining Policies

conflict resolution strategy:

- ❑ when multiple policies are applicable, how are these policies resolved?

Examples:

- ❑ “best for customer”
- ❑ “best for store”!

Composite pricing strategy:

- ❑ Determine which pricing strategies are applicable
- ❑ Apply the relevant conflict resolution strategy

Composite (GoF)

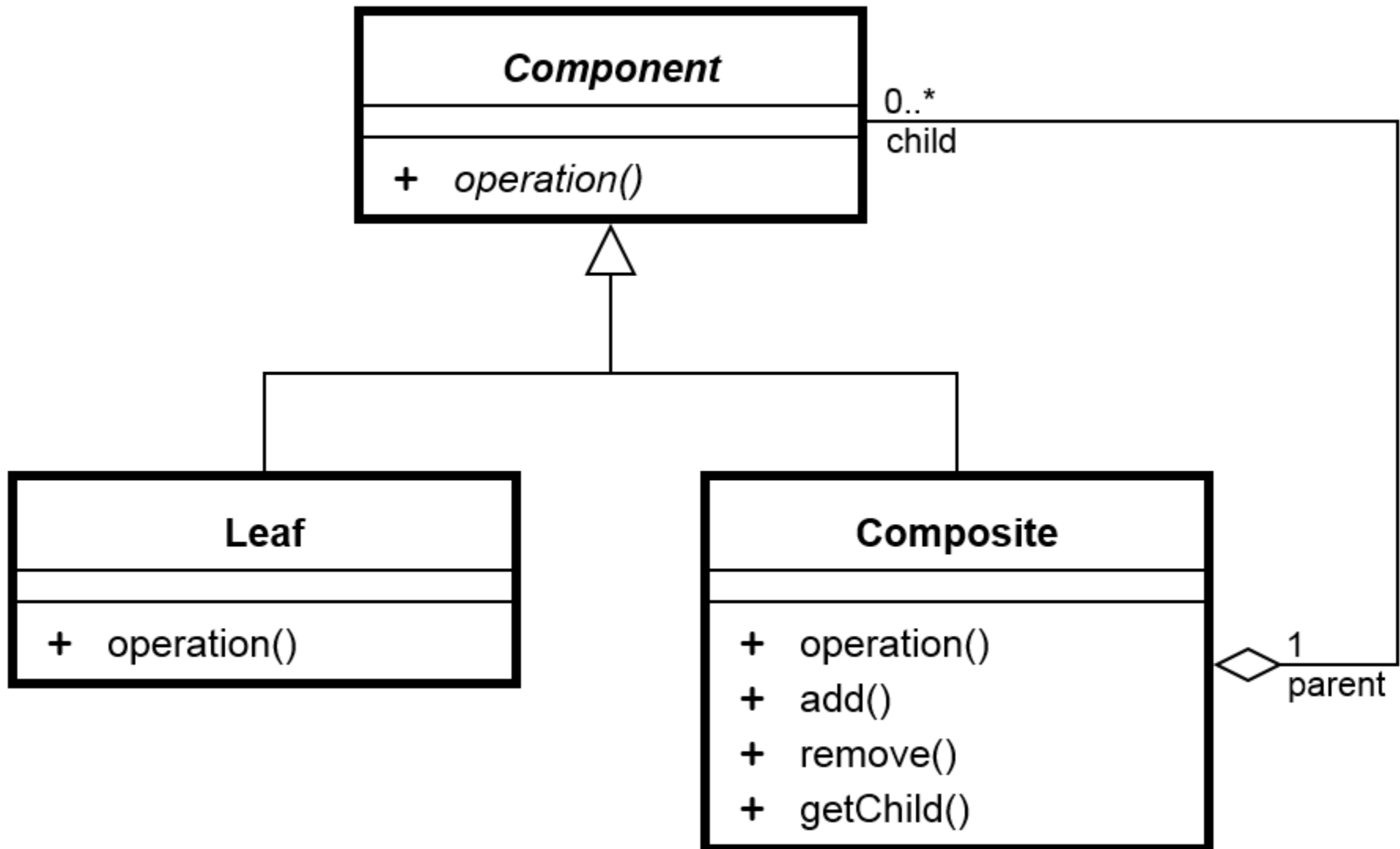
Problem:

- ❑ How to treat a group or composition structure of objects the same way (polymorphically) as a non-composite (atomic) object?

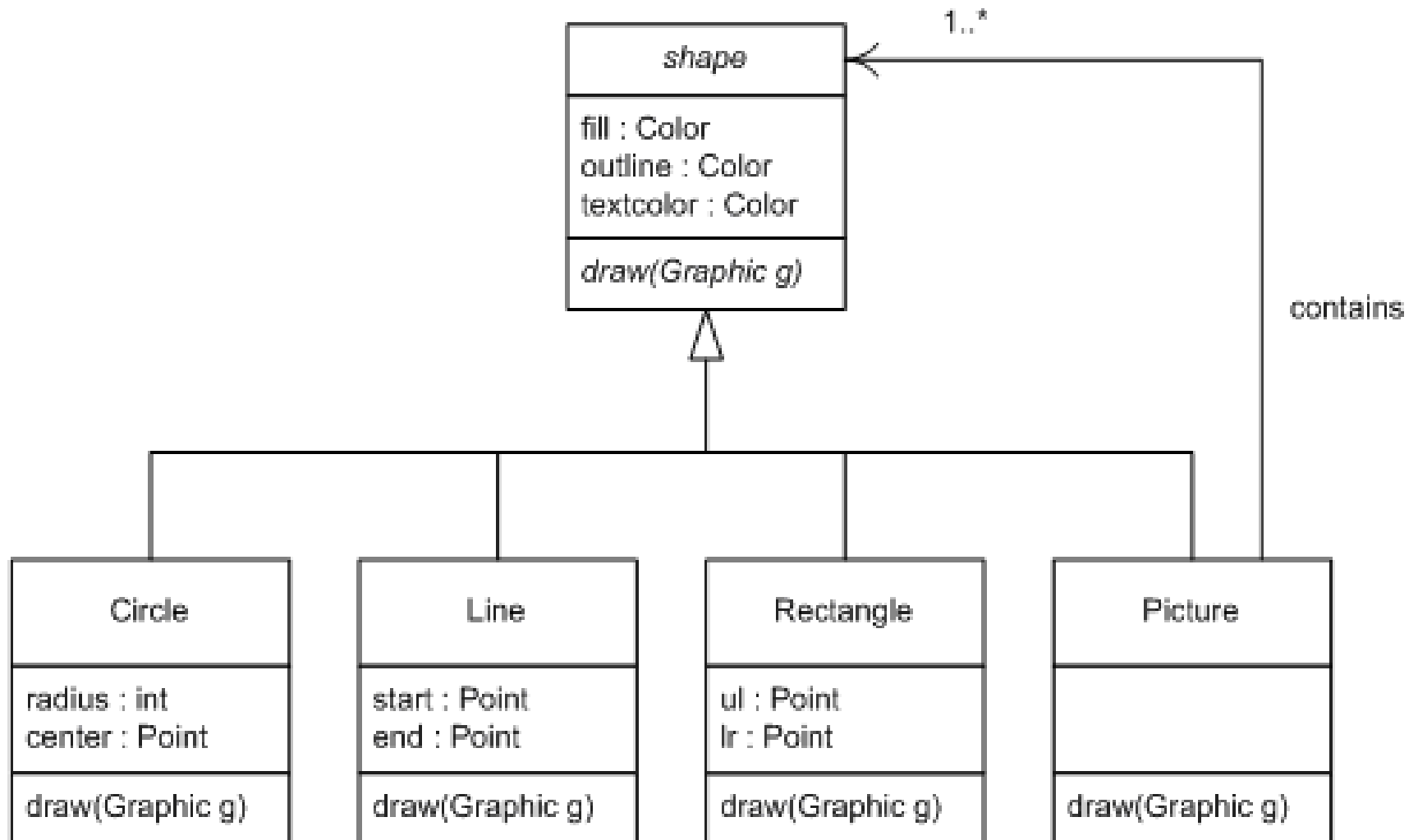
Solution (advice):

- ❑ Define classes for composite and atomic objects so that they implement the same interface.

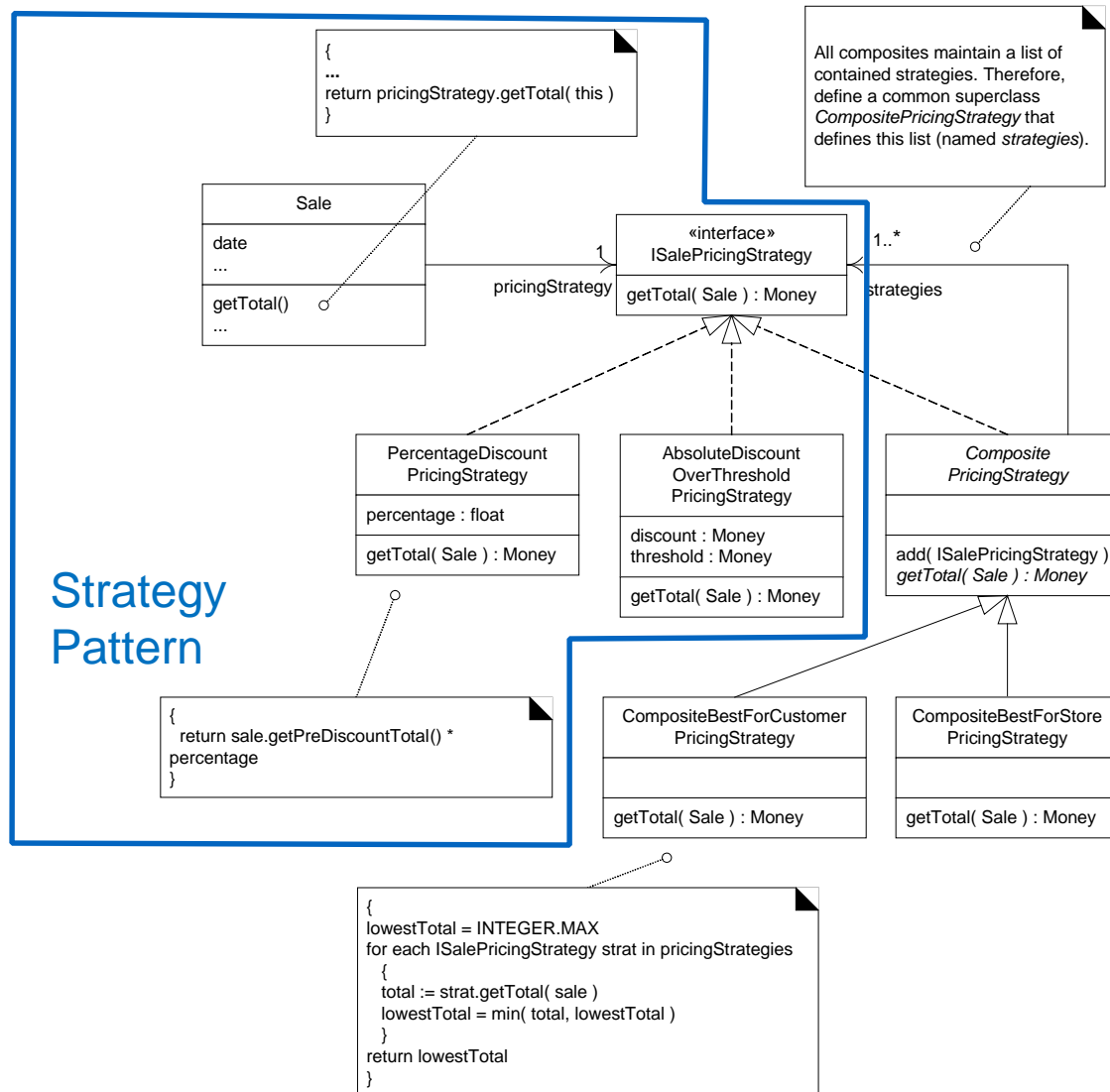
Composite: Generalised Structure



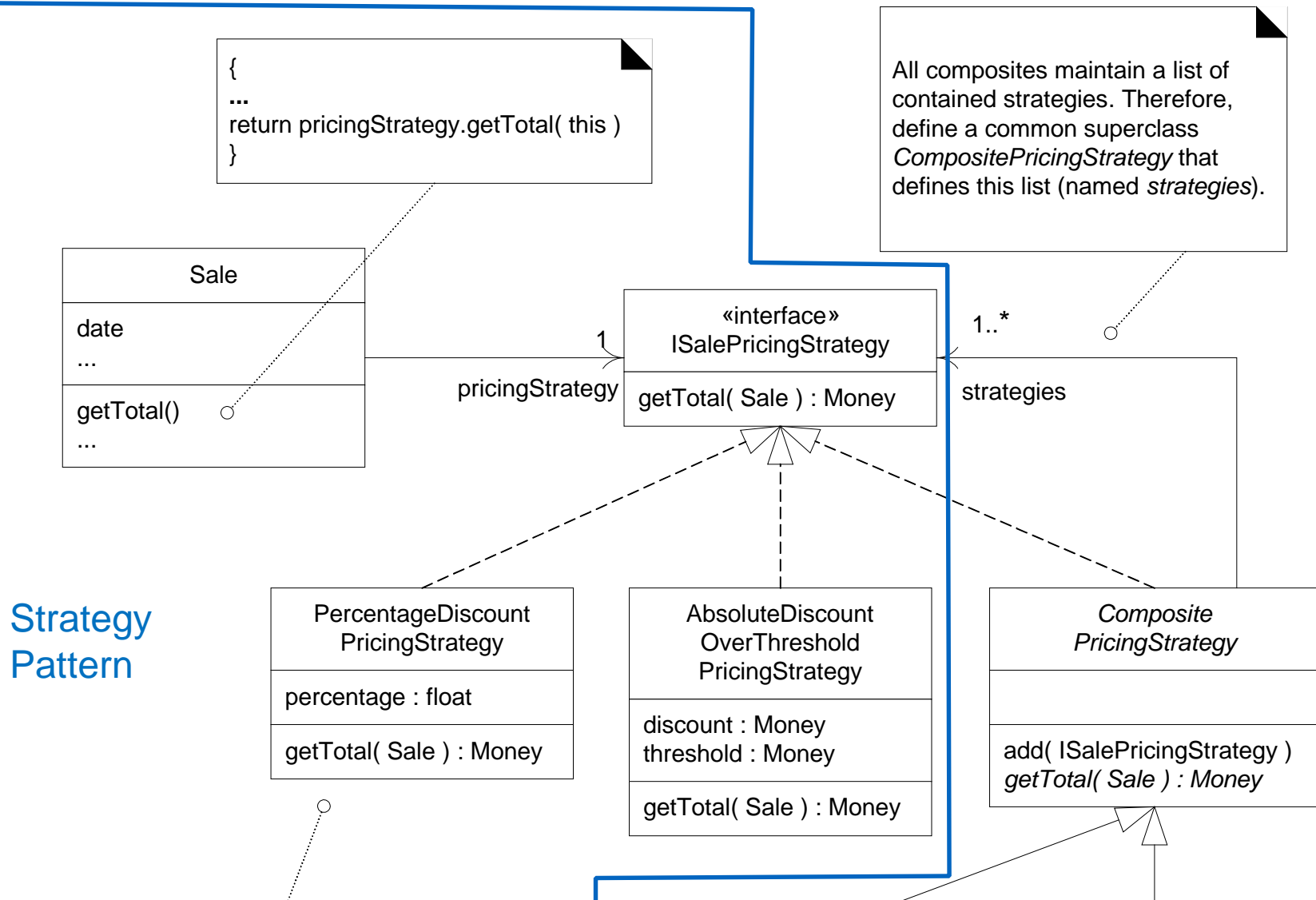
Composite: Example



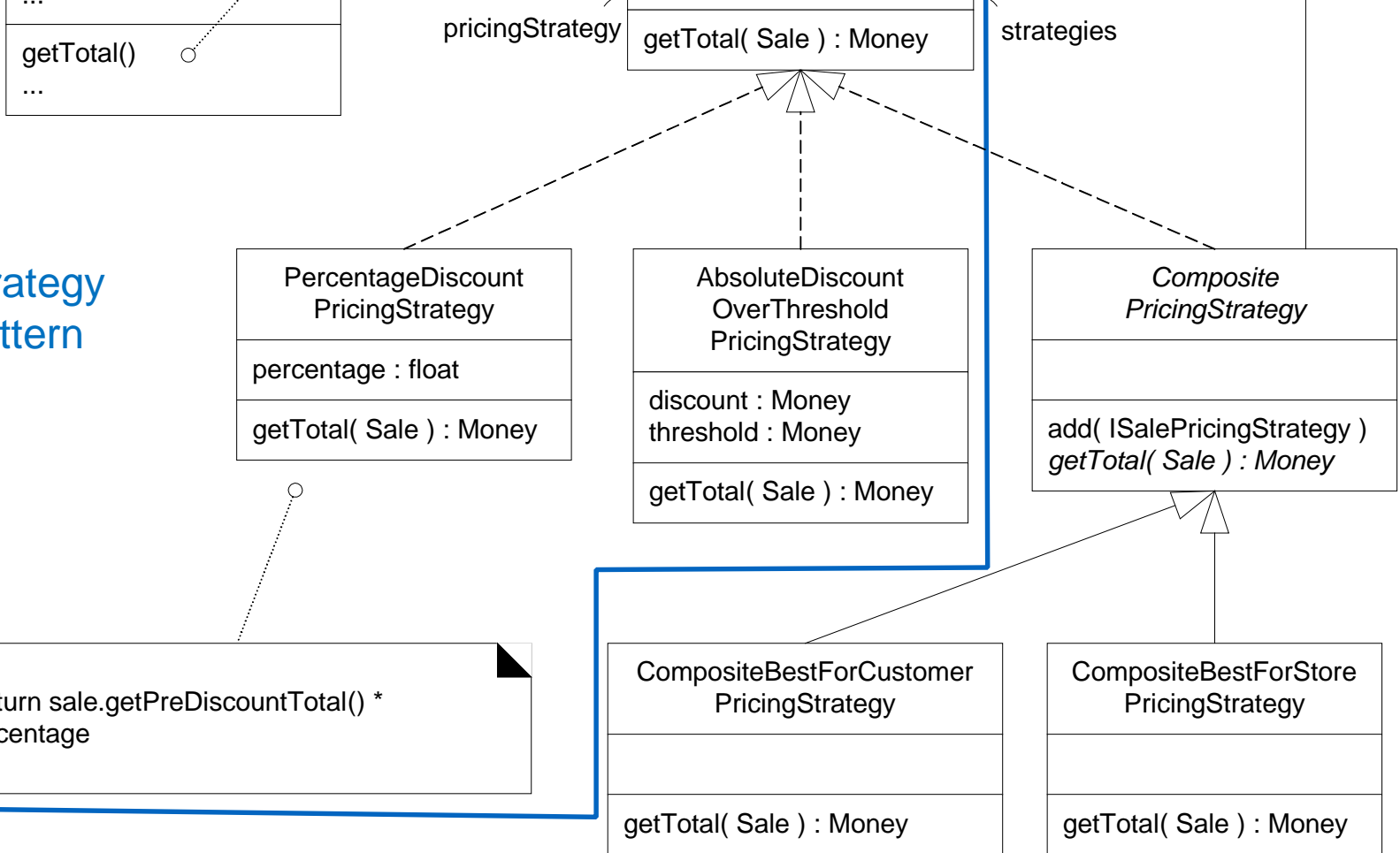
The Composite Pattern



The Composite Pattern



Strategy Pattern



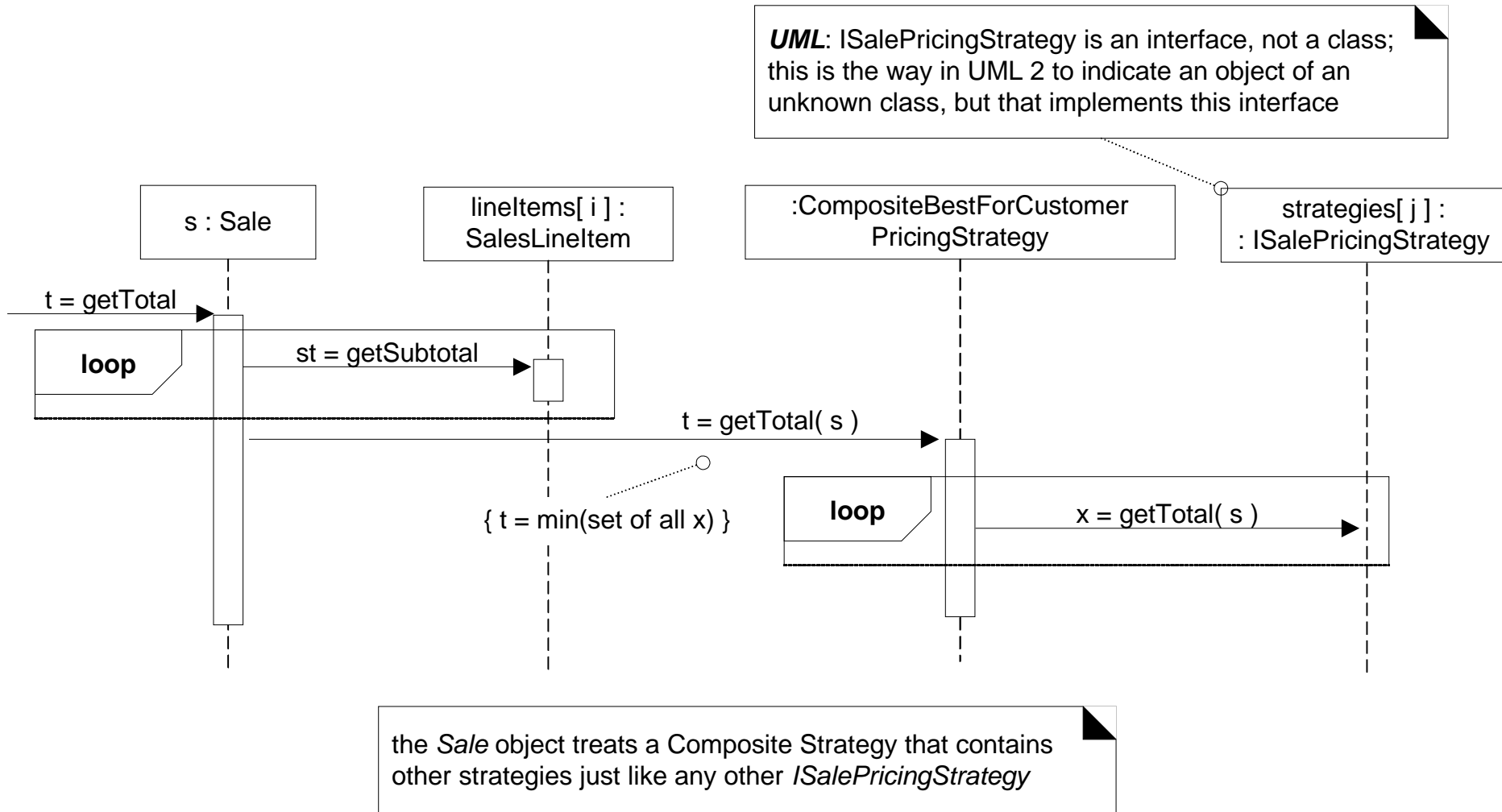
```

{
    return sale.getPreDiscountTotal() *
    percentage
}
  
```

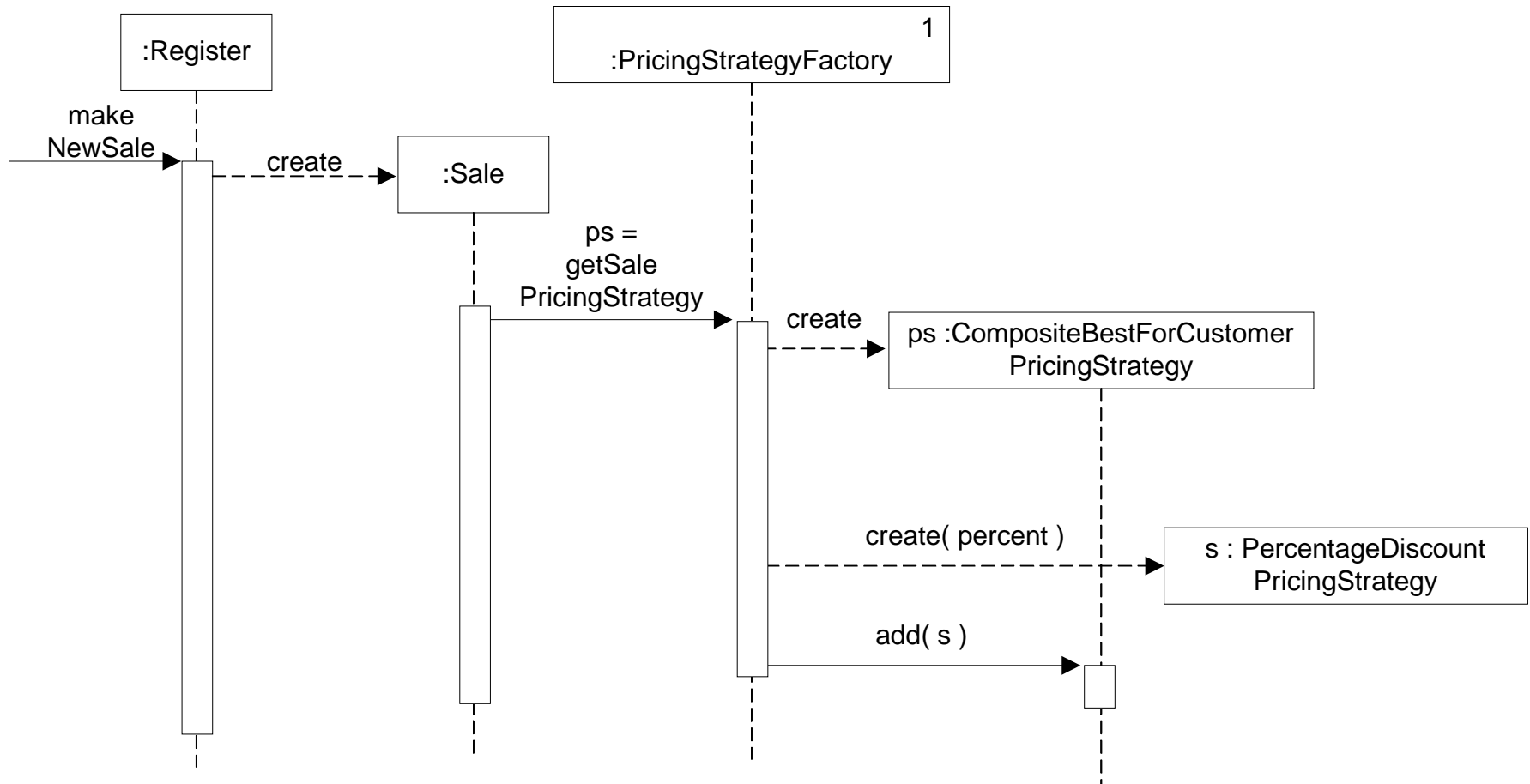
```

{
    lowestTotal = INTEGER.MAX
    for each ISalePricingStrategy strat in pricingStrategies
    {
        total := strat.getTotal( sale )
        lowestTotal = min( total, lowestTotal )
    }
    return lowestTotal
}
  
```

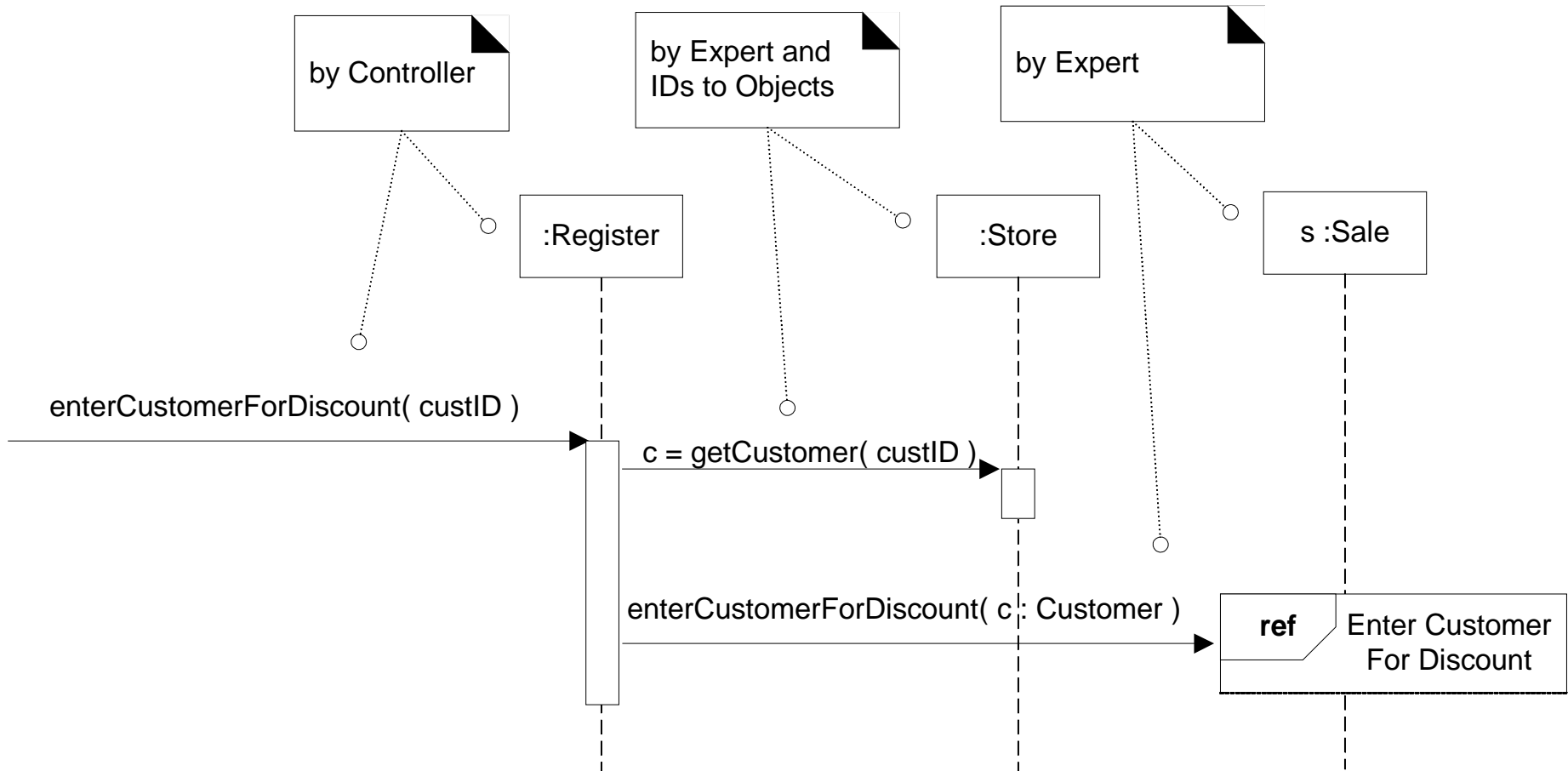

Collaboration with a Composite



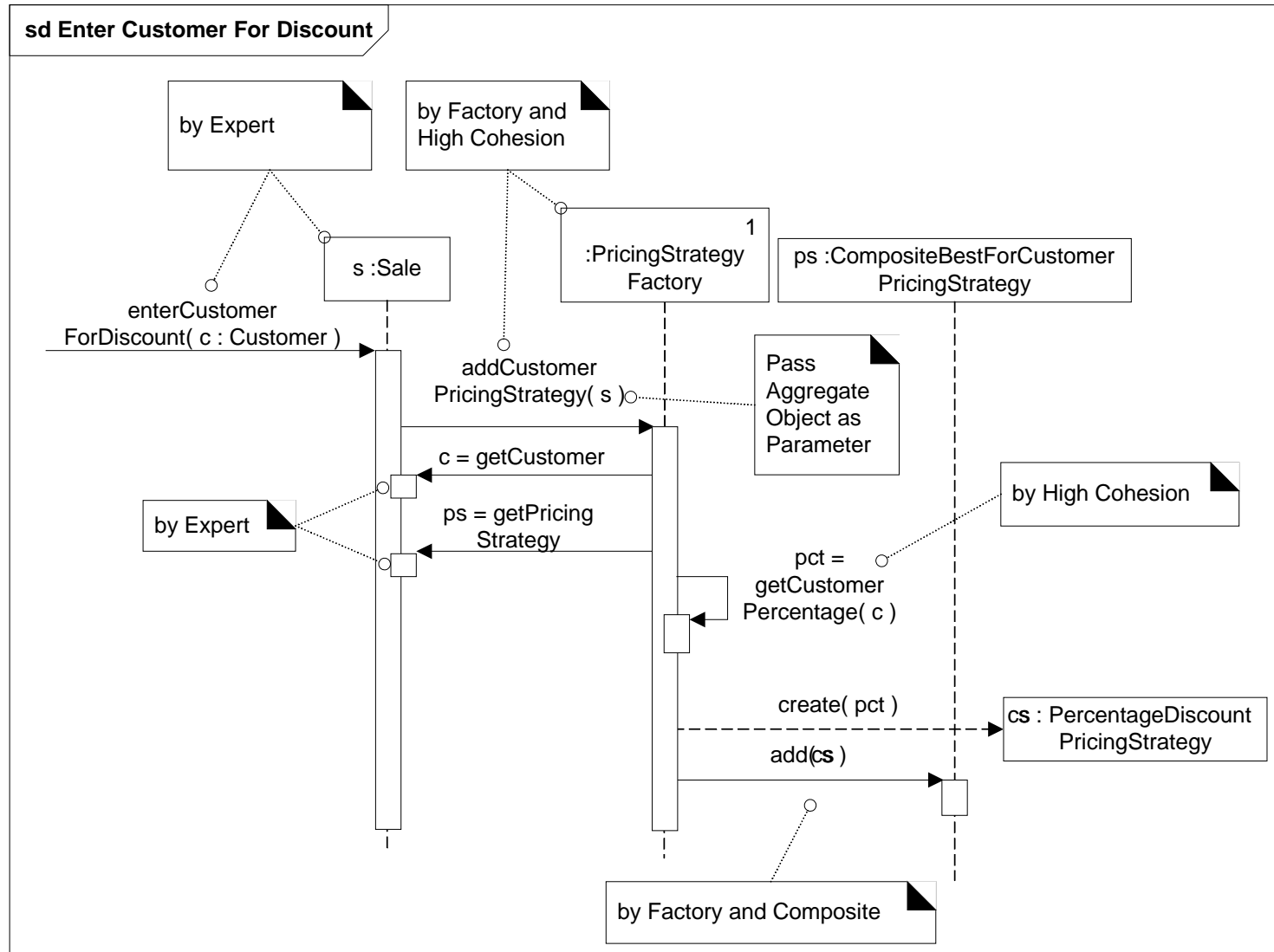
Creating a Composite Strategy



Creating Discount Pricing Strategy (1)



Creating Discount Pricing Strategy (2)



3. NextGen Pluggable Business Rules

Customise, e.g. to invalidate an action:

❑ Gift certificate sale

- Only one item and change must be gift certificate
 - Subsequent EnterItem ops invalidated
 - Request for change as cash or store a/c credit invalidated

❑ Charitable donation (by store) sale

- Cashier must be a manager and each item under \$250
 - If cashier is not manager, CreateCharitableSale invalidated
 - EnterItem ops for items over \$250 invalidated

❑ “..., let’s use a rule engine behind a facade”

Façade (GoF)

Problem:

- ❑ Require 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?

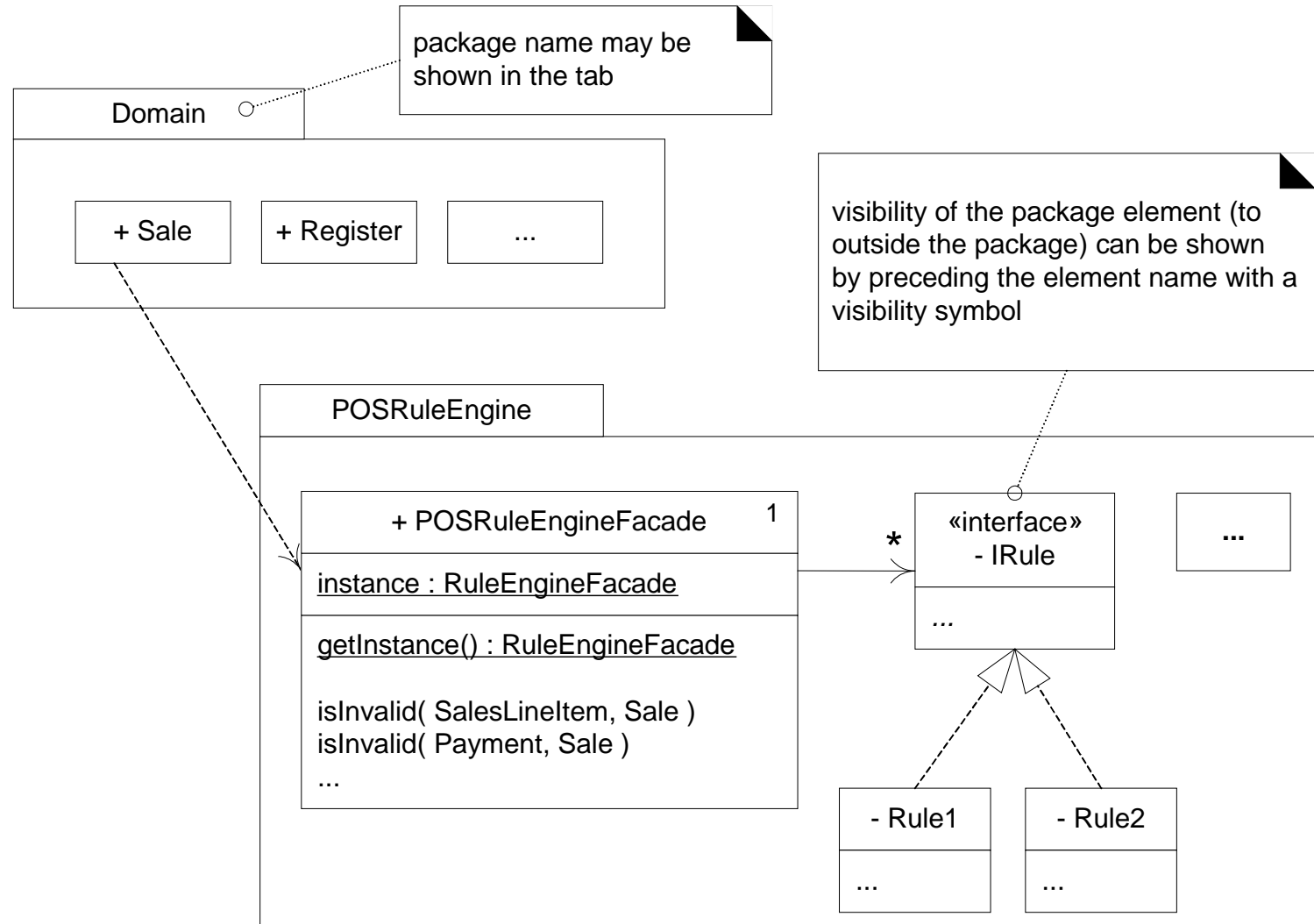
Solution (advice):

- ❑ 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.

Using the Facade

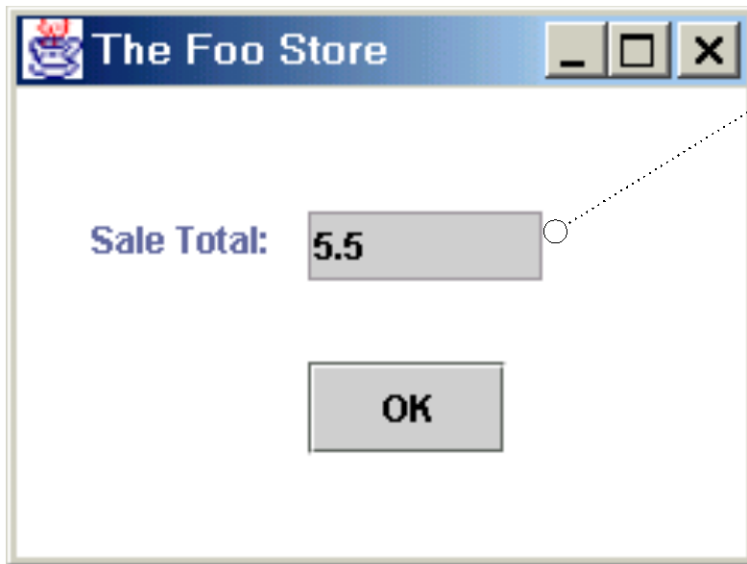
```
public class Sale {  
    public void makeLineItem( ProductDescription desc, int quantity )  
    {  
        SalesLineItem sli = new SalesLineItem( desc, quantity );  
        // call to the Facade  
        if ( POSRuleEngineFacade.getInstance().isInvalid( sli, this ) )  
            return;  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

UML Package Diagram: Facade



4. Refreshing Sales Total Interface

Goal: When the total of the sale changes, refresh the display with the new value



“To handle this problem, let’s have the SalesFrame observing the Sale object”

Observer (aka Publish-Subscribe) (GoF)

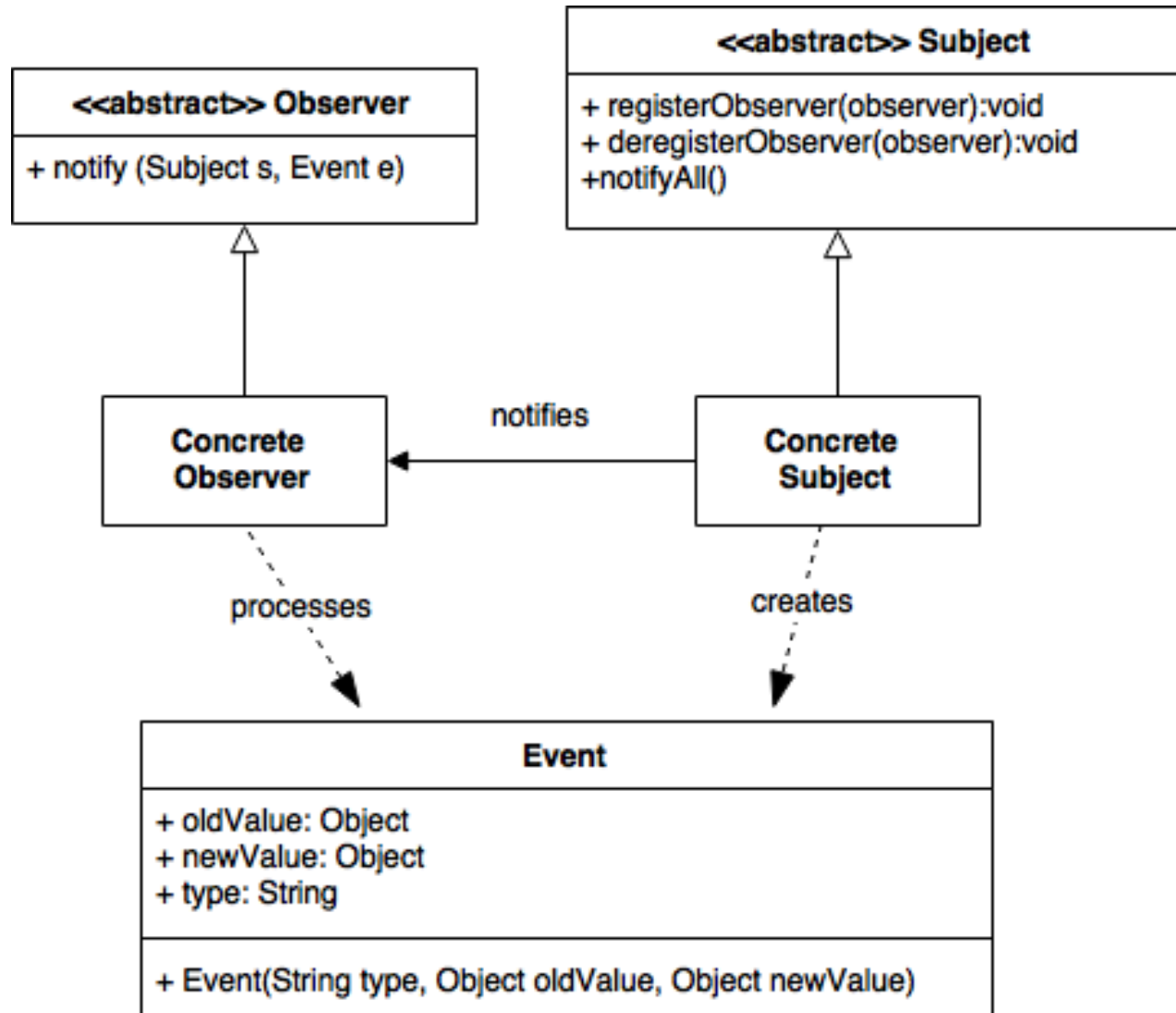
Problem:

- ❑ 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?

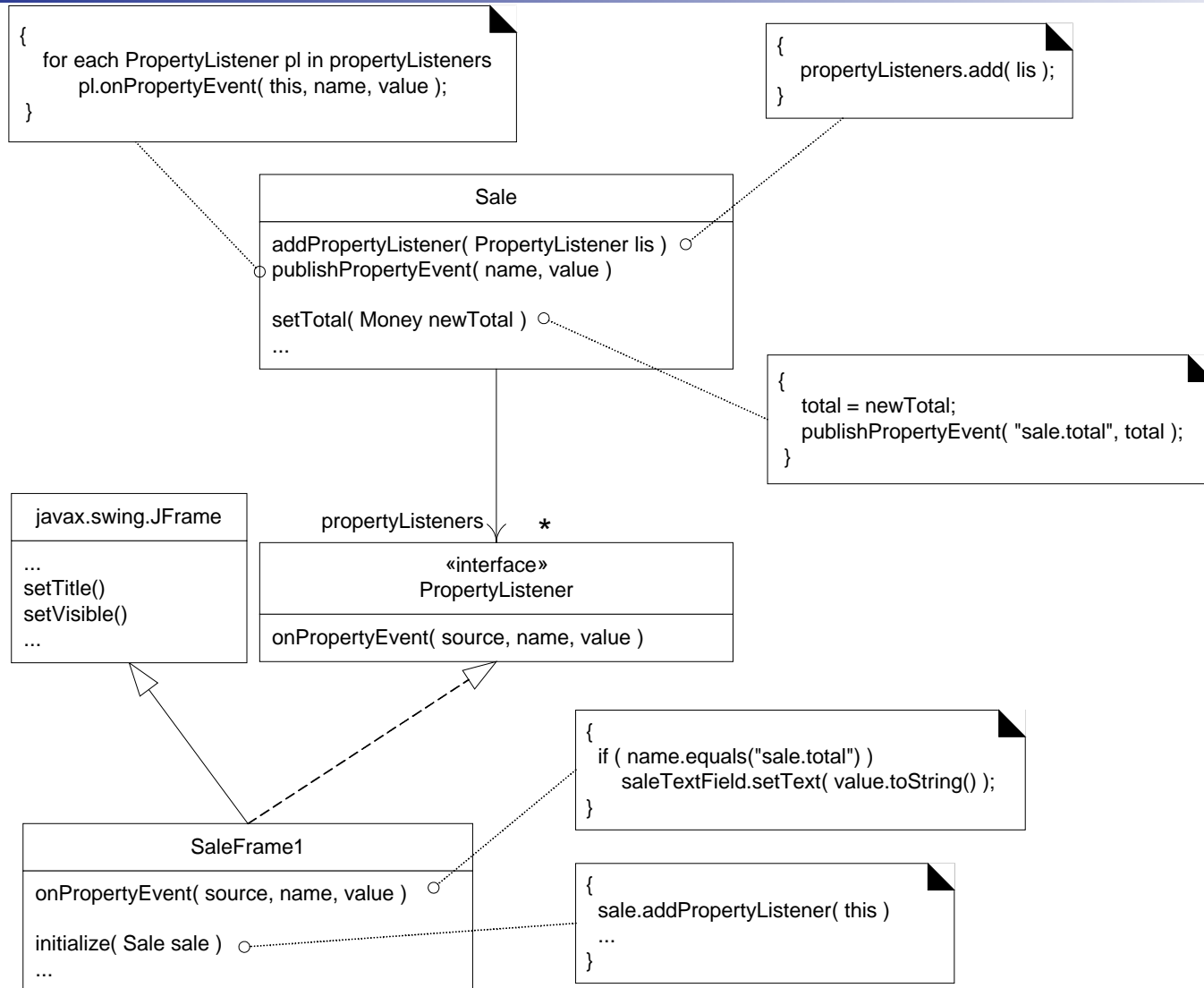
Solution (advice):

- ❑ 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.

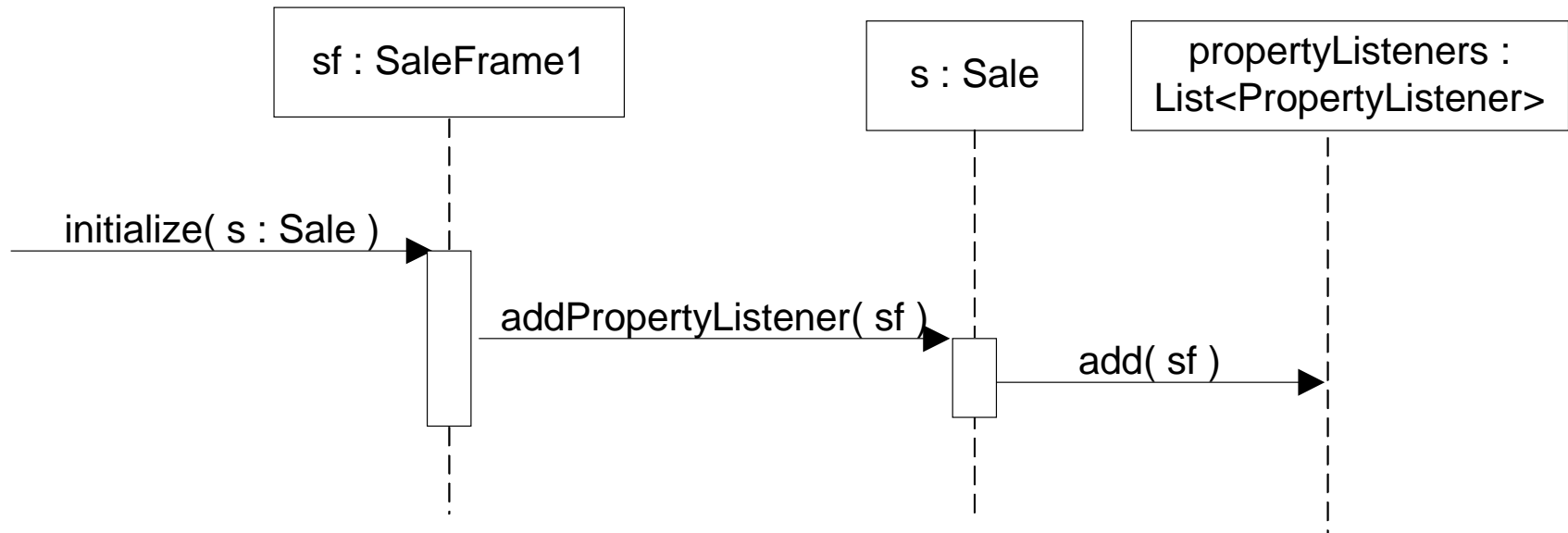
The Observer Pattern



The Observer Pattern (Sale Total)



Ob. *SalesFrame1* Subscribes to Pub. *Sale*

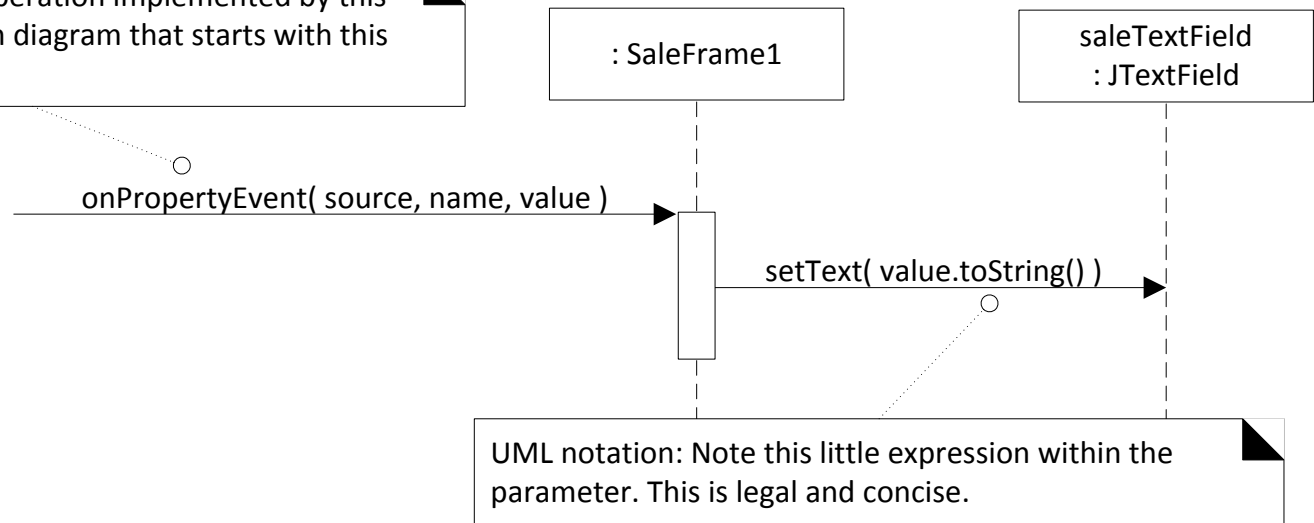


Sale Publishes Property Event to Subscribers

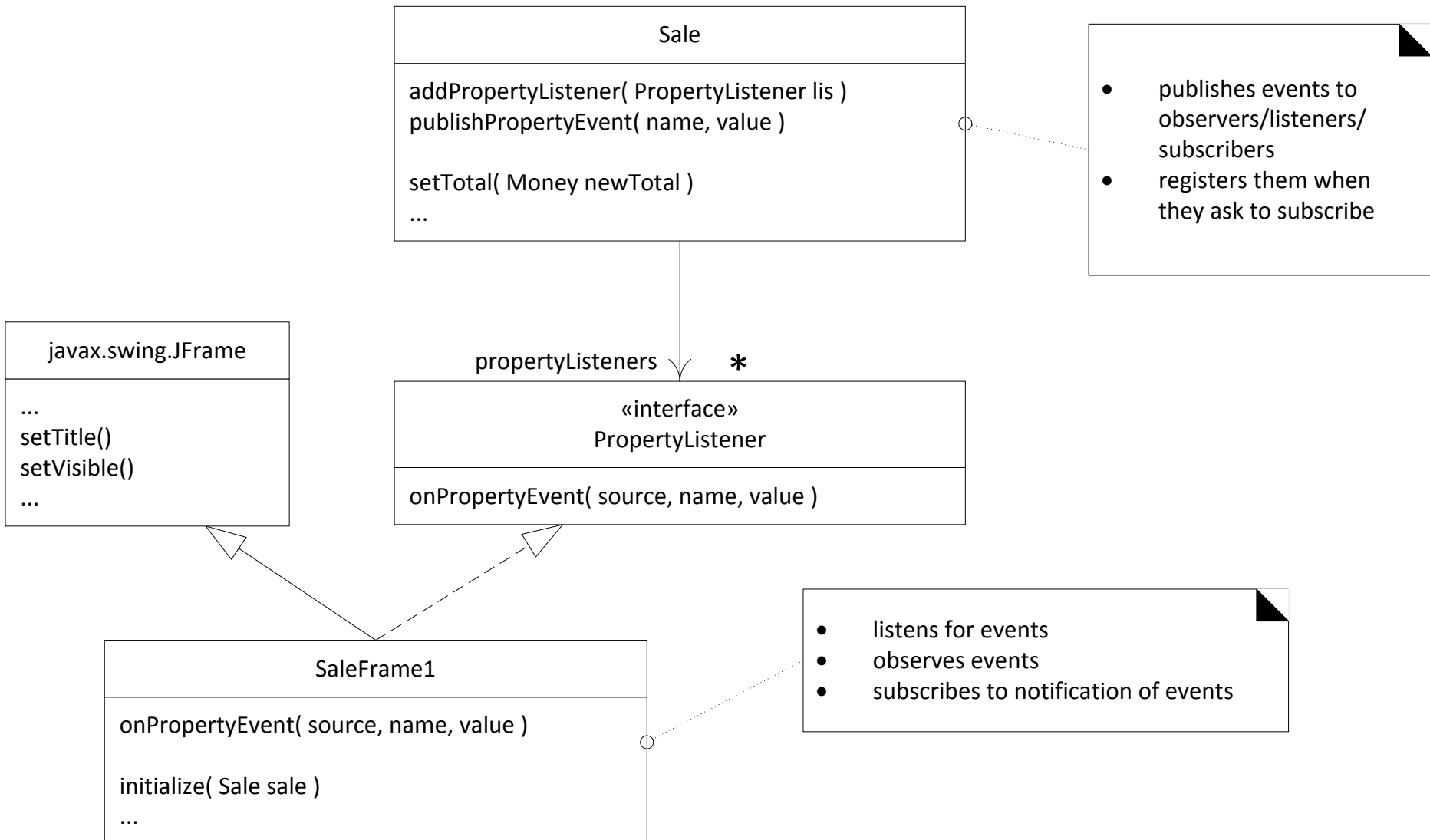


Subscriber *SaleFrame1* Receives Notification

Since this is a polymorphic operation implemented by this class, show a new interaction diagram that starts with this polymorphic version



Who is?: Observer, Listener, Subscriber, Publisher



Observer: Alarm Events, Diff. Subscribers

