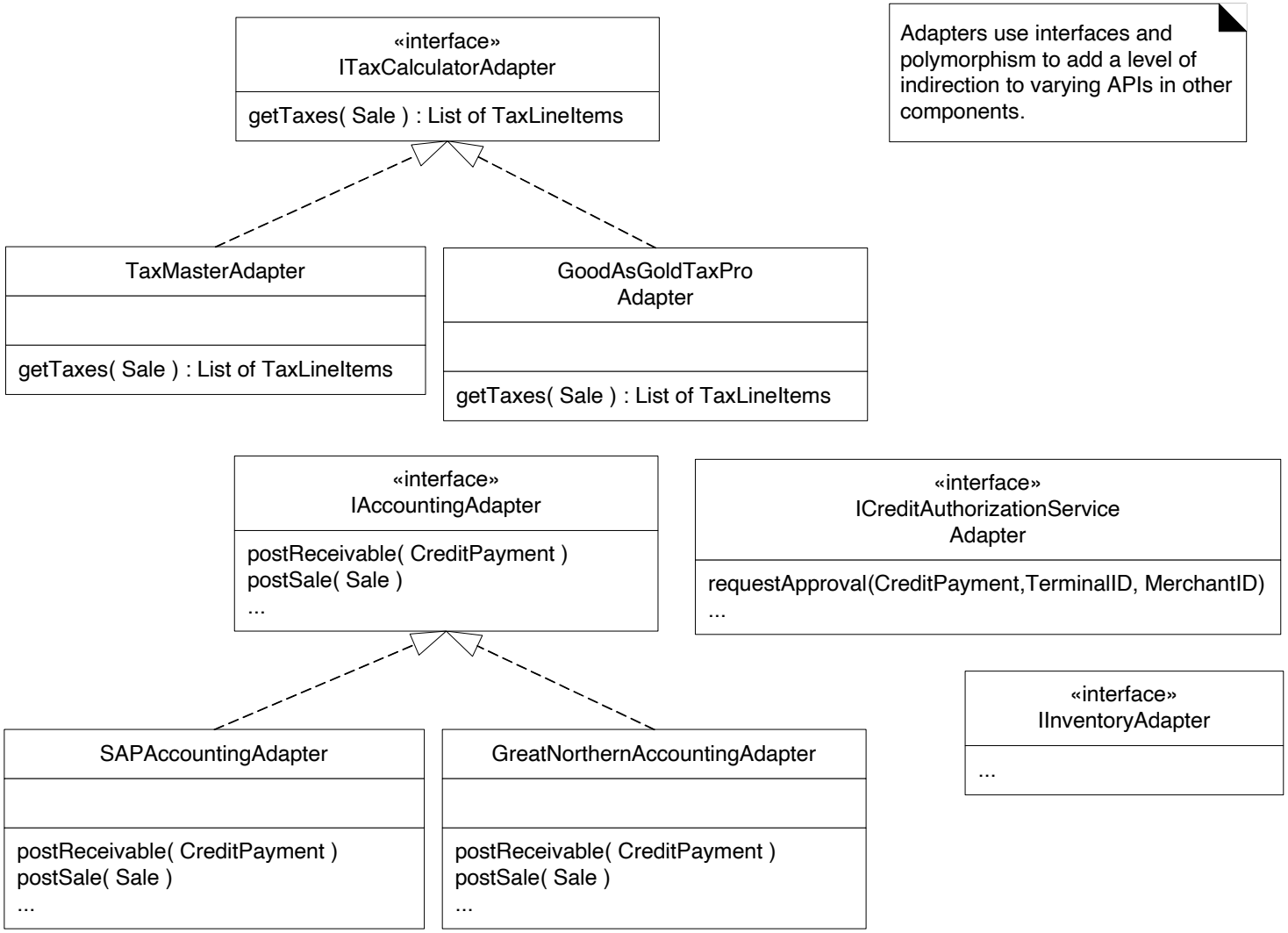
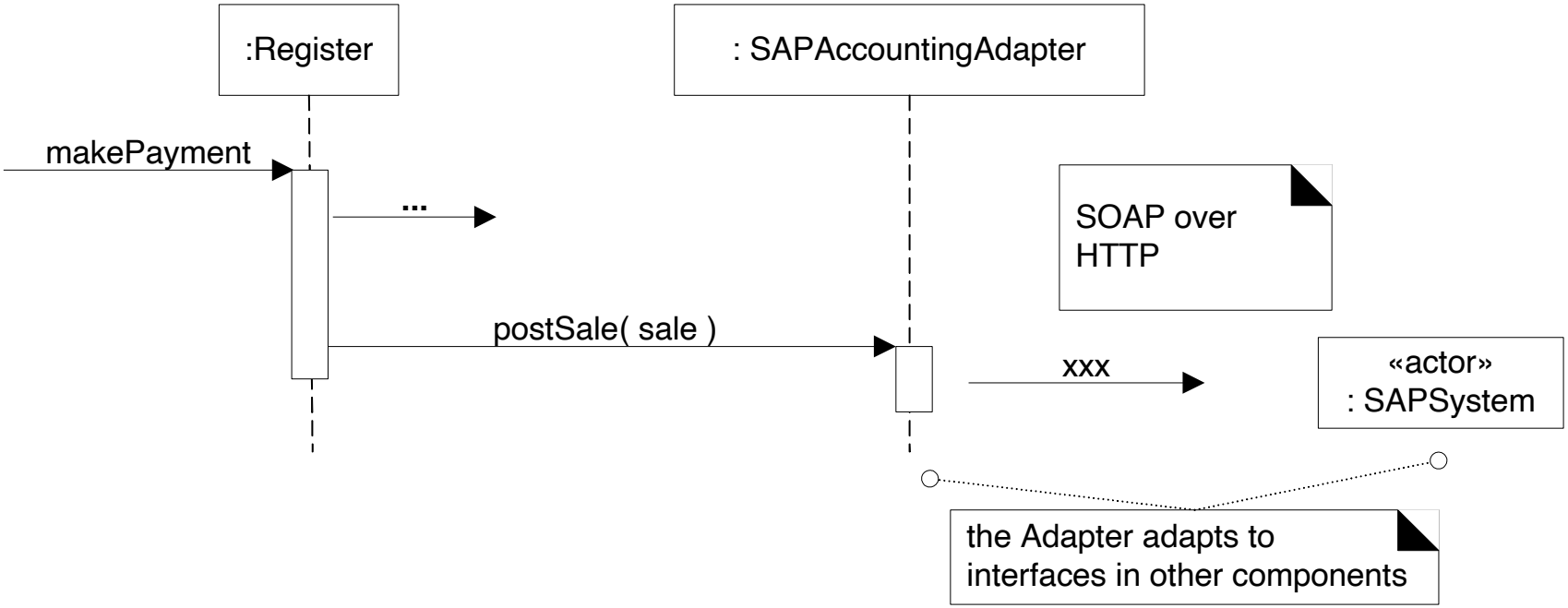
Chapter 26

GoF Design Patterns

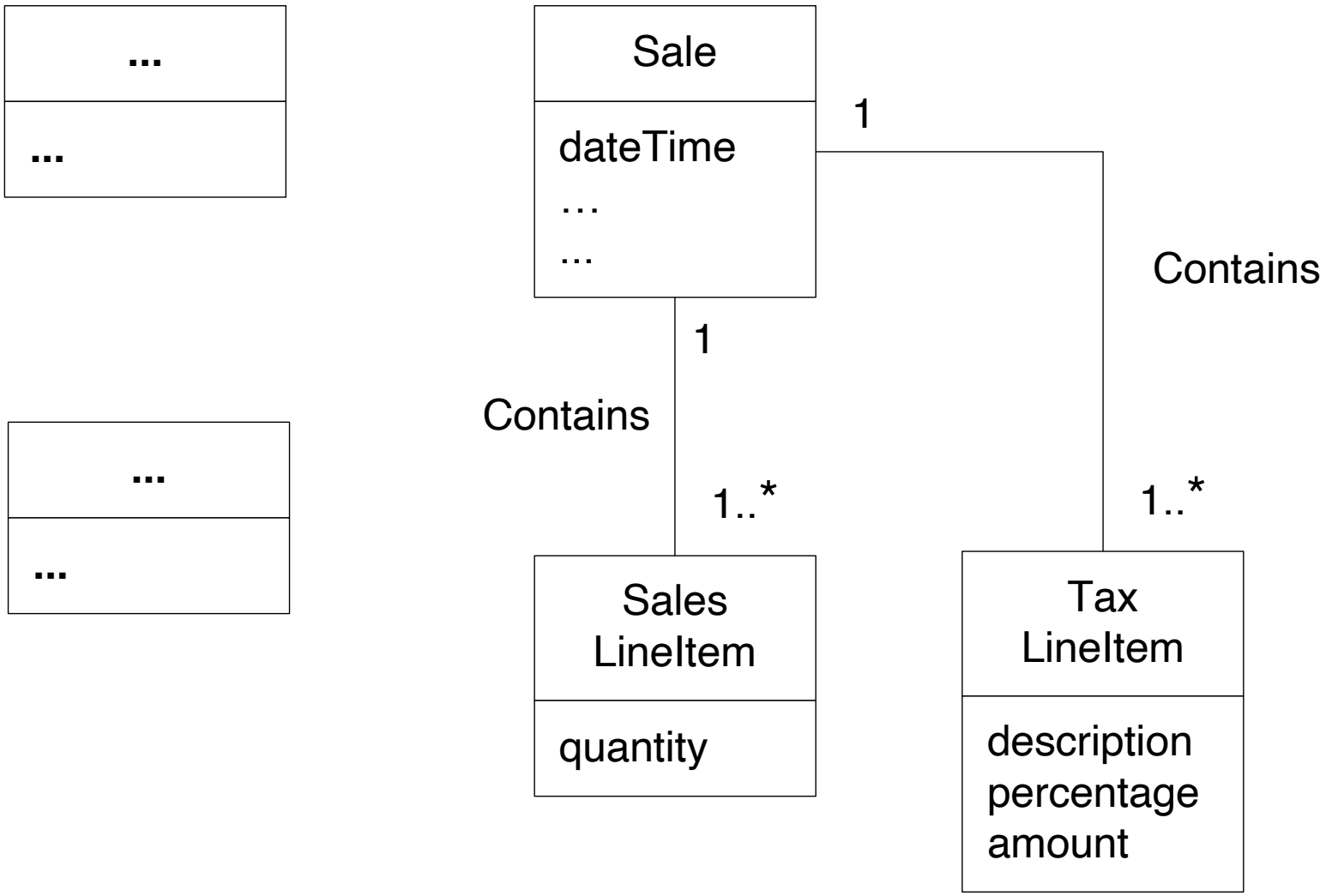
The Adapter Design Pattern



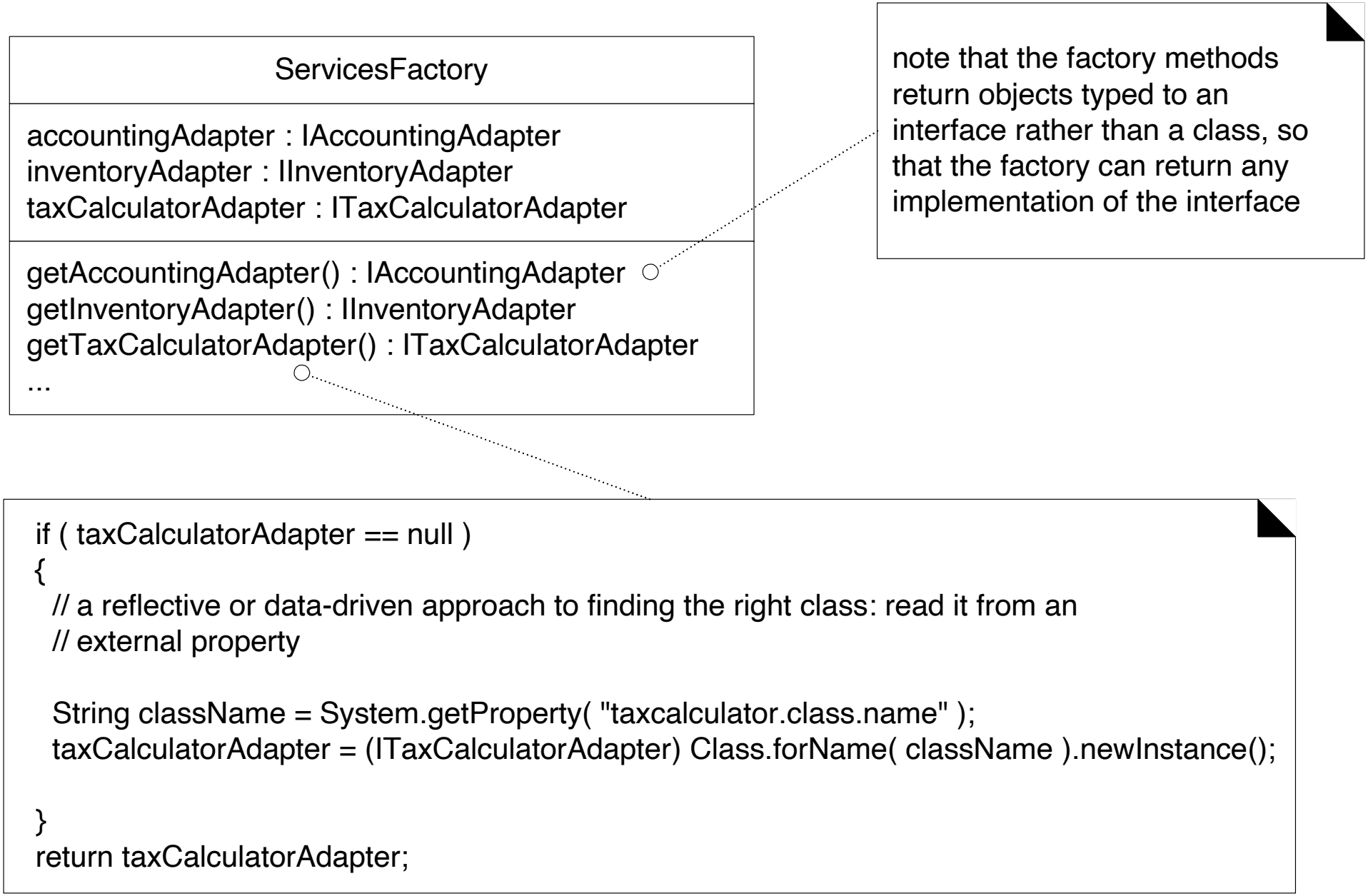
Interaction Diagram for “Adapter”



Sometimes we update domain models to reflect our more refined understanding of domain



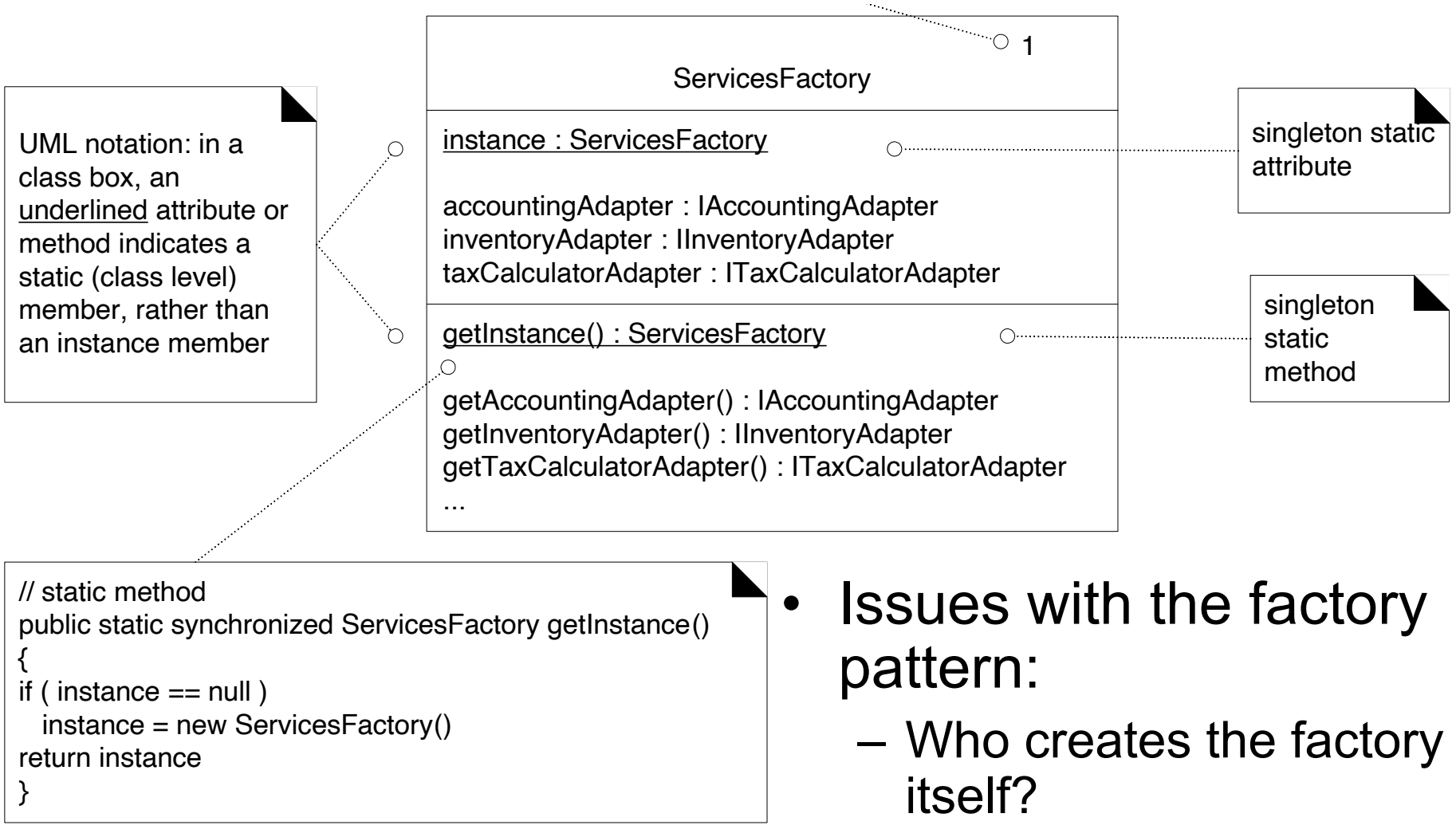
The Factory Pattern



The Factory Pattern

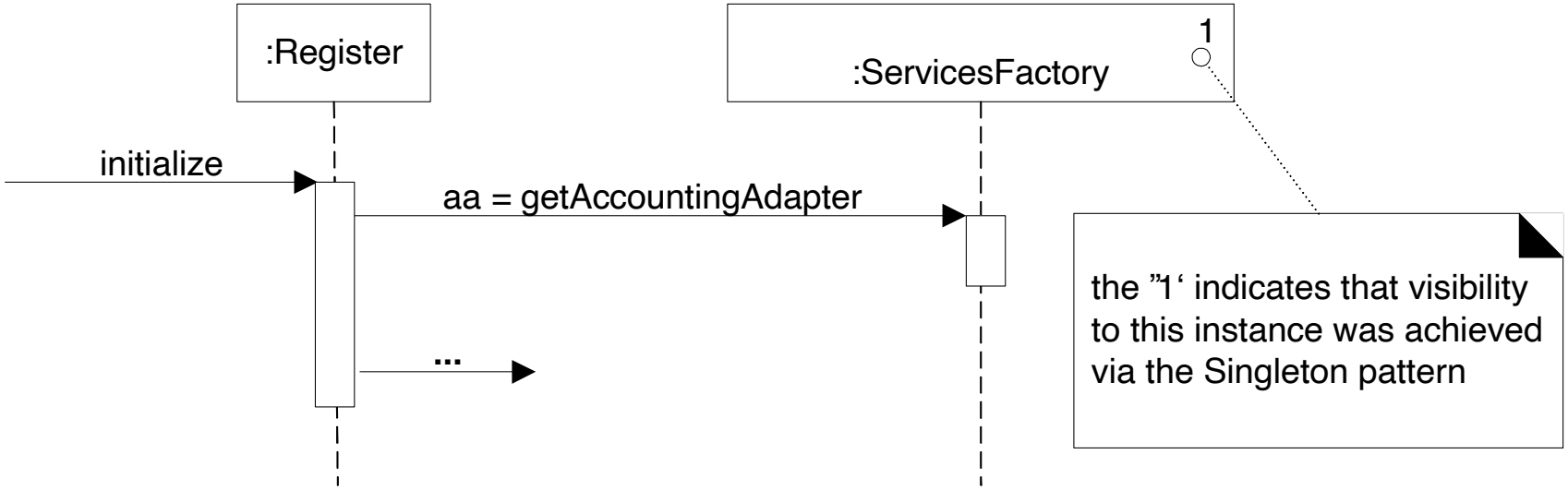
- Advantages:
  - Separate the responsibility of complex object creation into cohesive helper objects
  - Hide potentially complex creation logic
  - Allow optimizations to enhance performance
    - Object caching: Create objects beforehand and keep them in memory for quick use.
    - Re-cycling: Pool of connection threads in a web-server.
- ServicesFactory:
  - Data-driven design: Class loaded dynamically (at run time) based on system property.
  - Can create other adapter classes by changing property value

The Singleton Pattern



- Issues with the factory pattern:
  - Who creates the factory itself?
  - How do we get access to the factory class from everywhere?

Accessing the Singleton




```
public class Register {
  public void initialize() {
    ...
  aa = ServicesFactory.getInstance().getAccountingAdapter();

    ...
  }
}
```

The Singleton Pattern: Implementation and Design Issues

- Lazy initialization:

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

- Eager initialization: ???

The Singleton Pattern: Implementation and Design Issues

- Lazy initialization:

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

- Eager initialization:

```
public class ServicesFactory {
  private static ServicesFactory instance =
                                    new ServicesFactory();

  public static ServicesFactory getInstance() {
    return instance;
  }
}
```

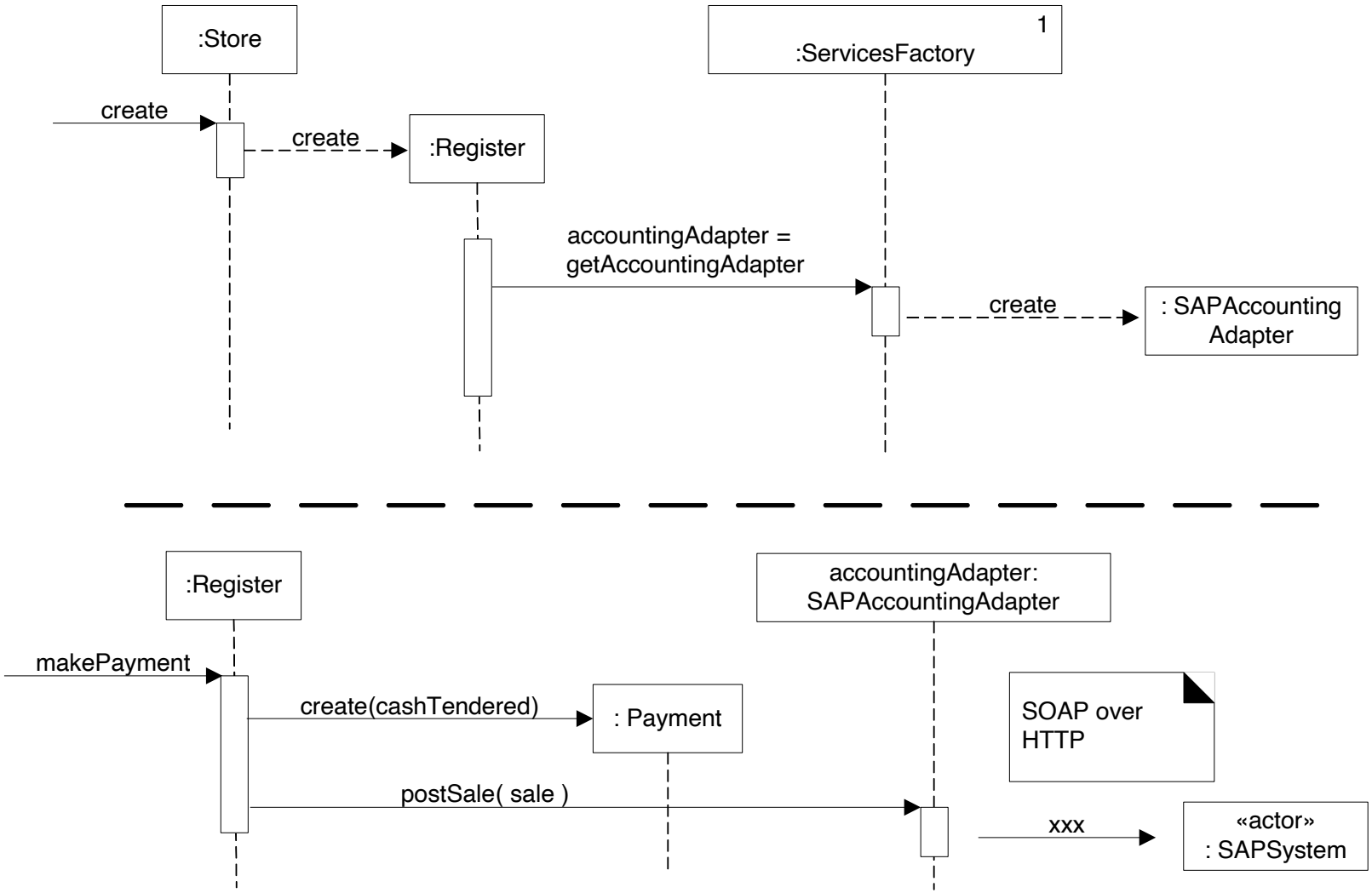
Singleton Issues

- Lazy vs. eager initialization. Which one is better?
  - Laziness, of course!
    - Creation work (possibly holding on to expensive resources) avoided if instance never created
- Why not make all the service methods static methods of the class itself?
  - Why do we need an instance of the factory object and then call its instance methods?

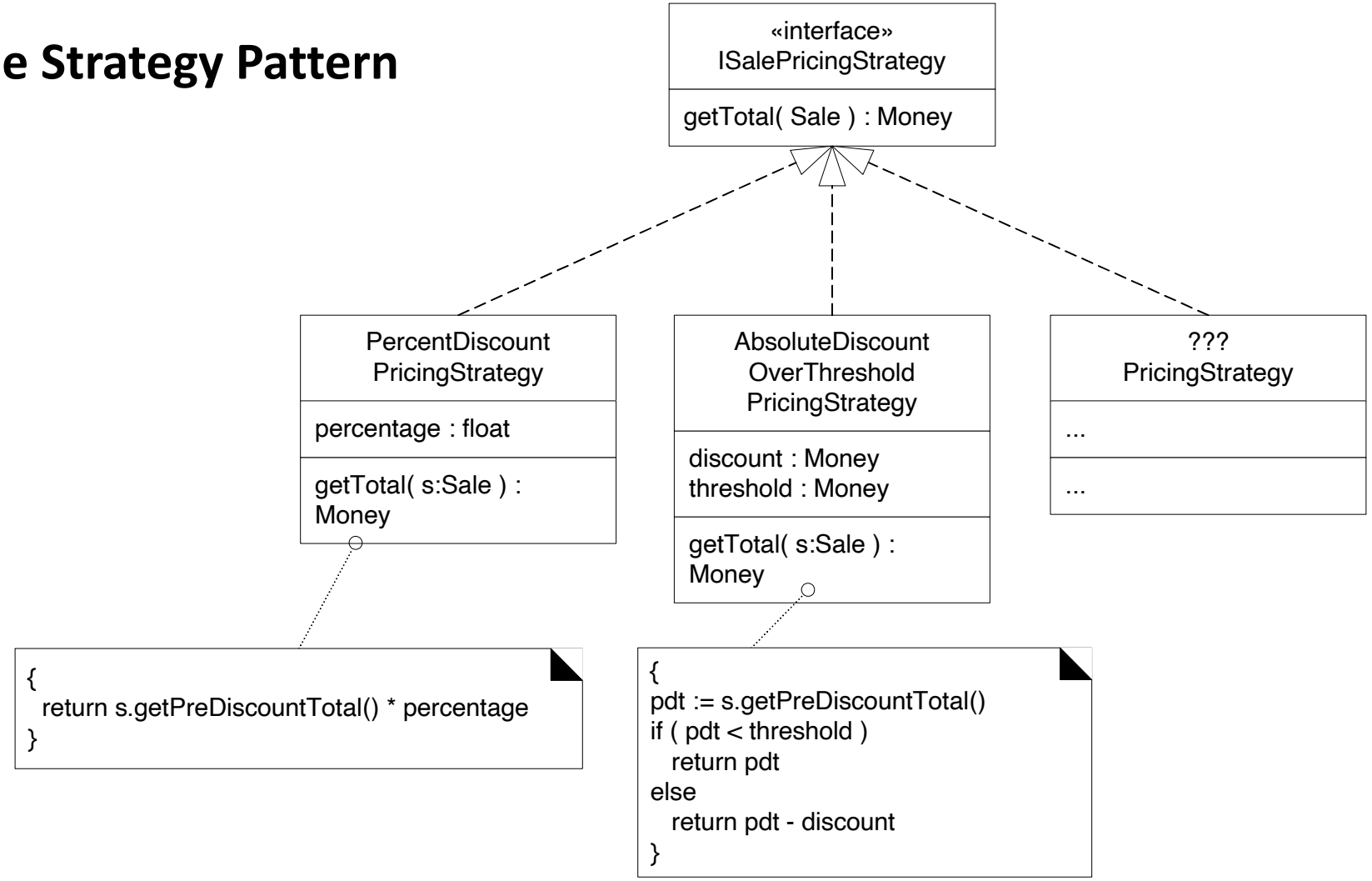
Singleton Issues

- Why not make all the service methods static methods of the class itself?
  - To permit subclassing: Static methods are not polymorphic, don't permit overriding.
  - Object-oriented remote communication mechanisms (e.g. Java RMI) only work with instance methods
    - Static methods are not remote-enabled.
  - More flexibility: Maybe we'll change our minds and won't want a singleton any more.

Design Patterns in Interaction Diagrams

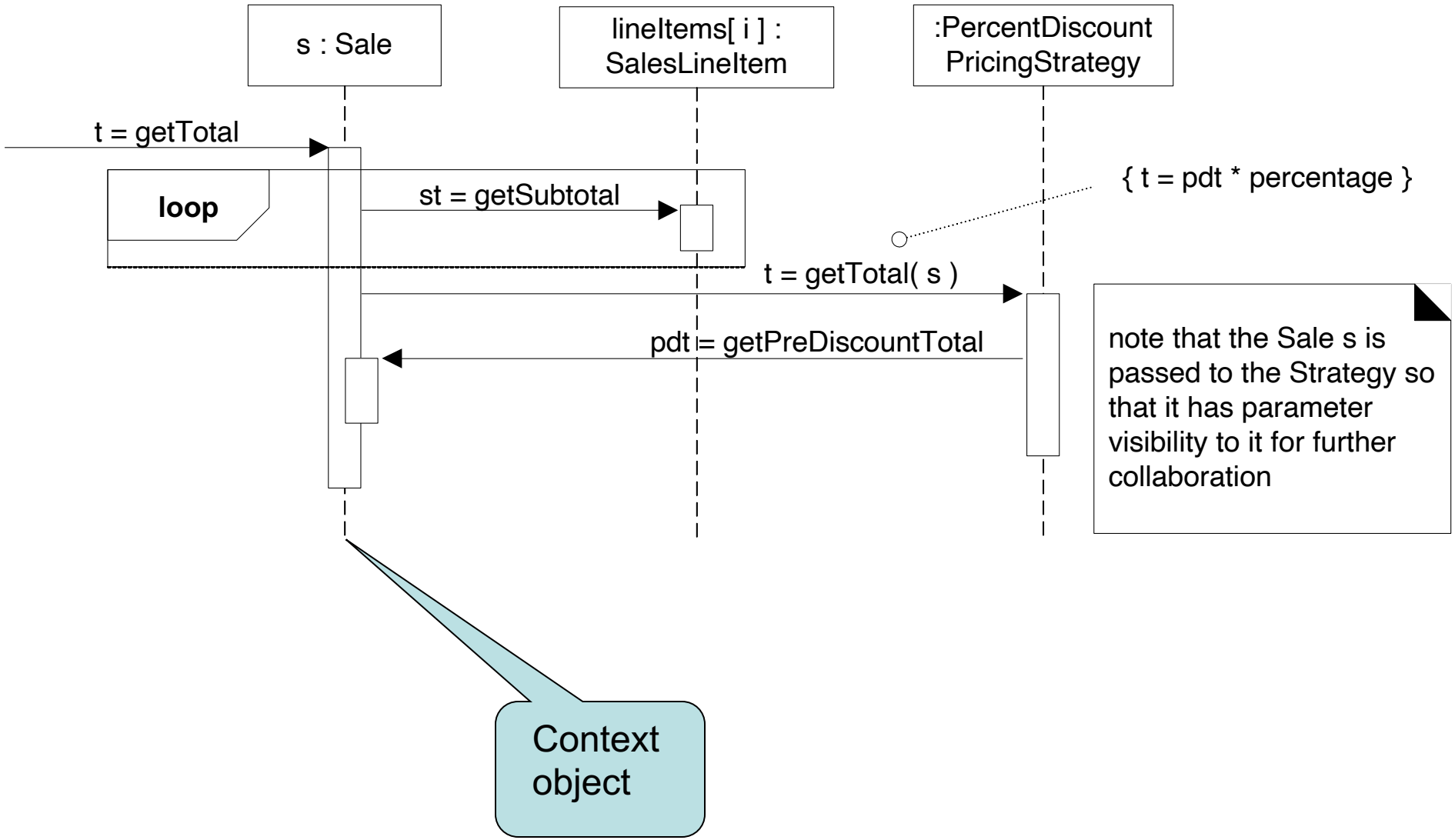


The Strategy Pattern

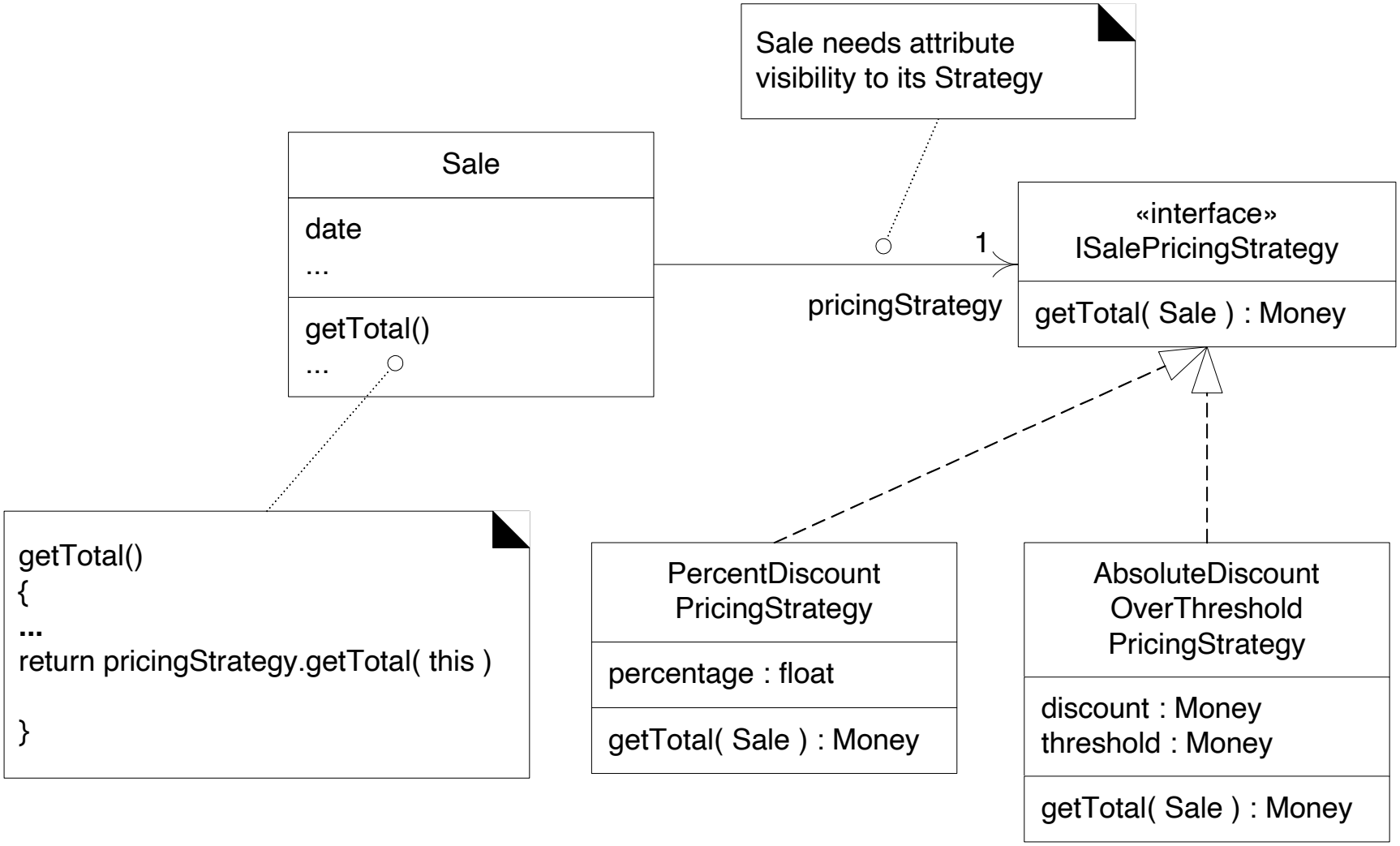


- Issue: Provide more complex pricing logic.
  - Pricing strategy varying over time
  - Example: Different kinds of sales.

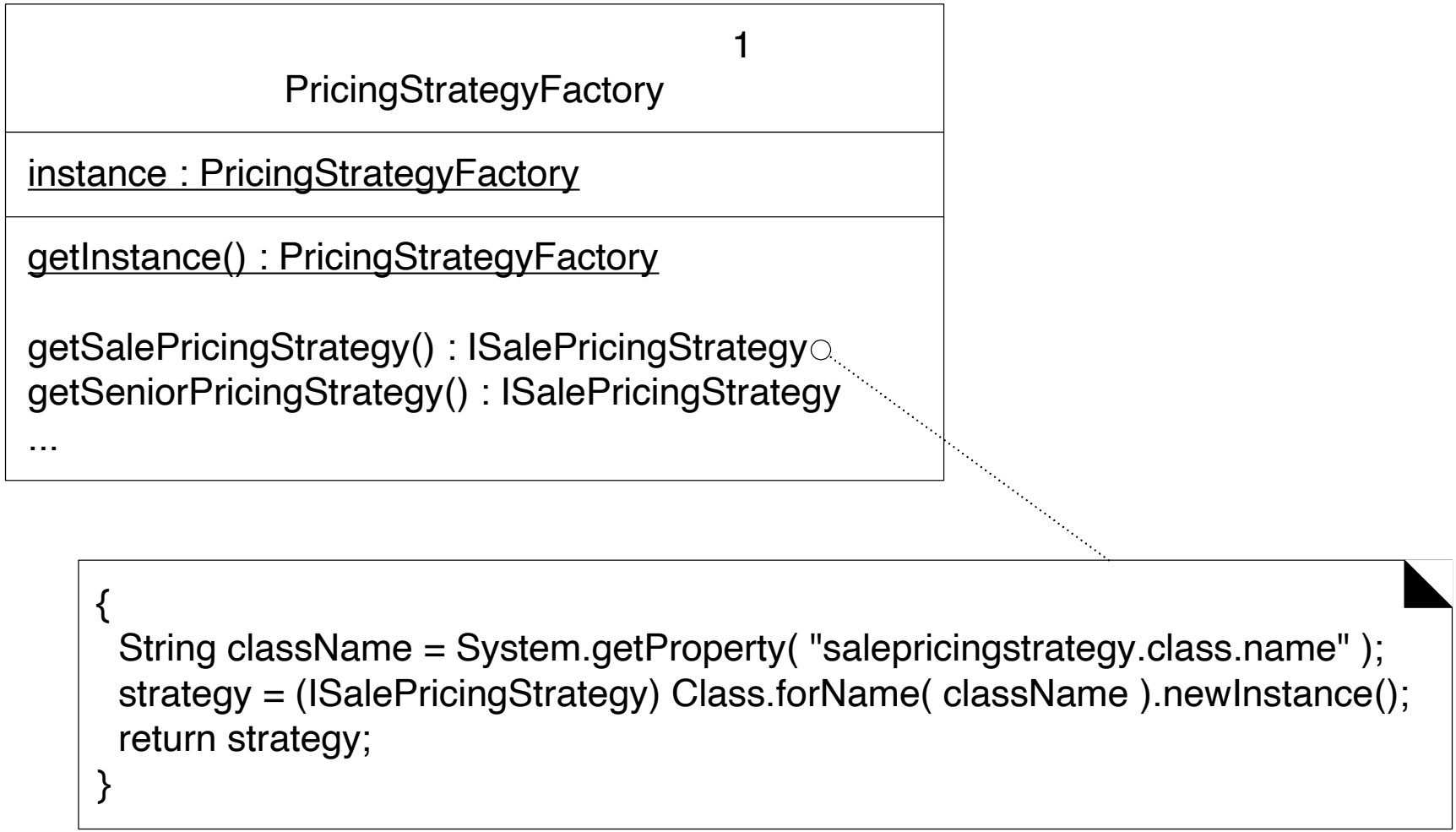
Interaction diagram for “Strategy”



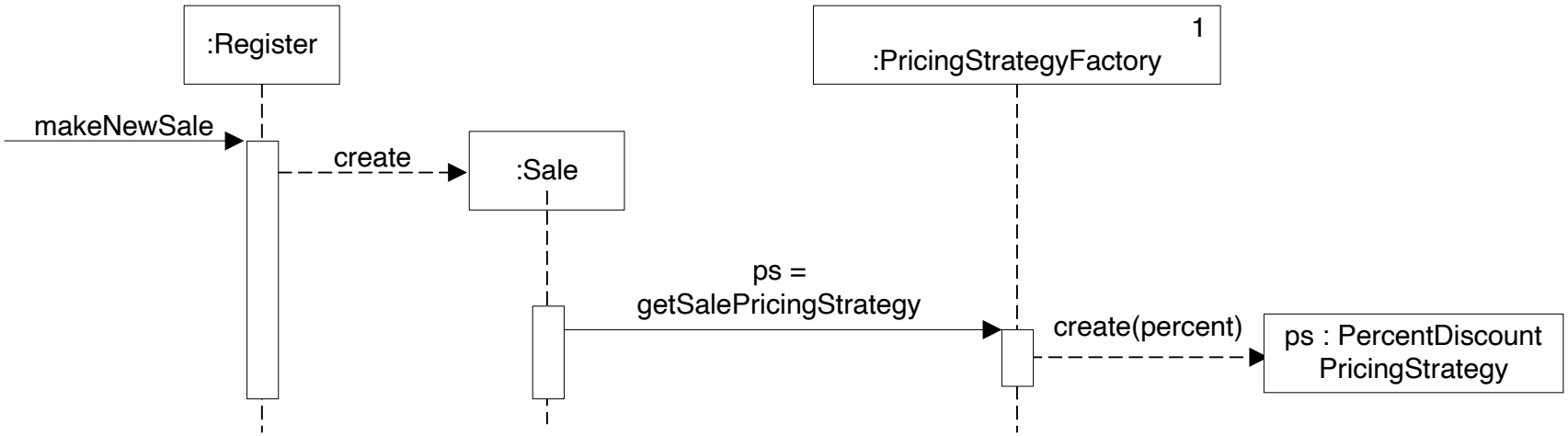
Context object has attribute visibility to its strategy



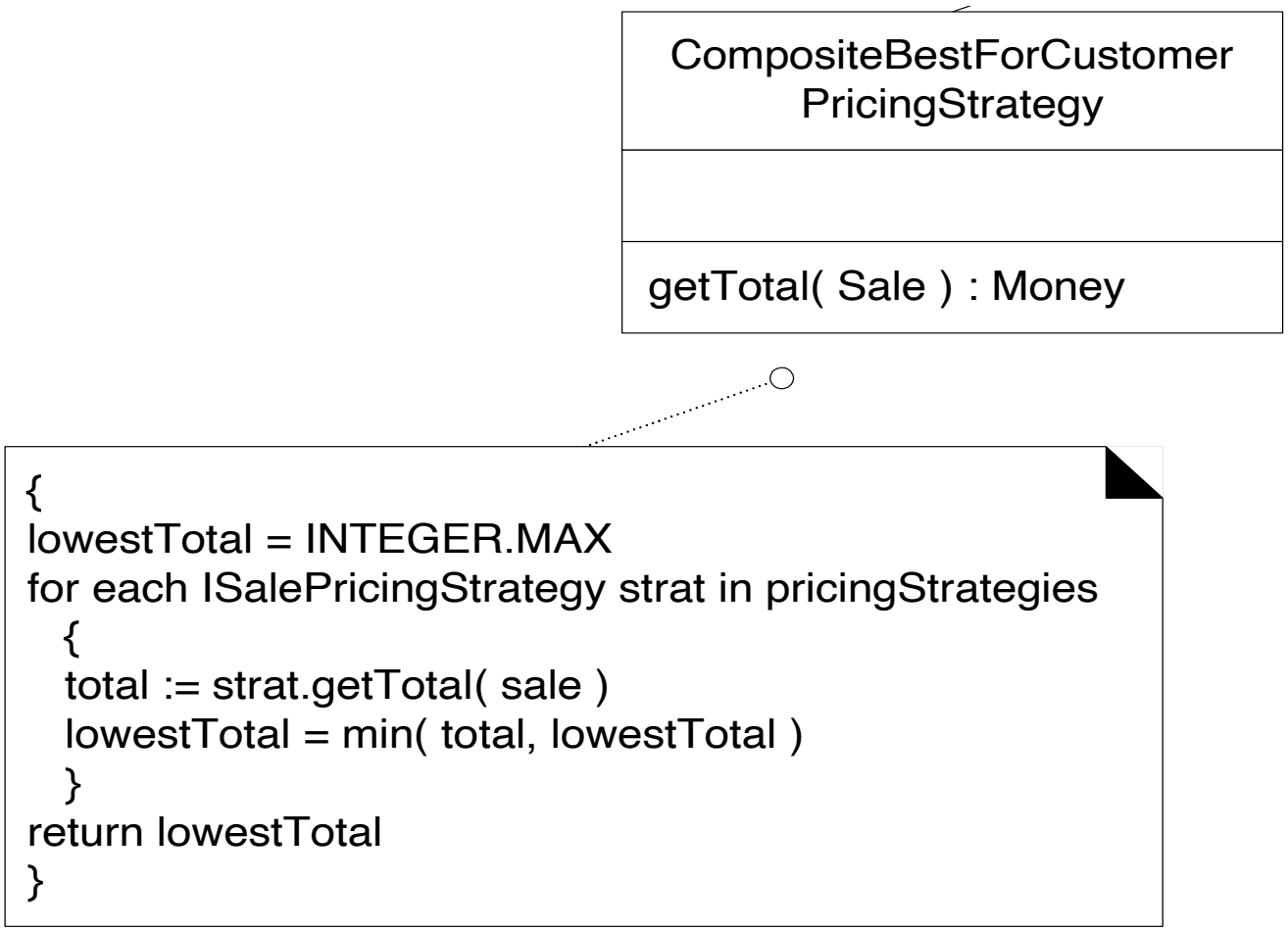
Factory for Strategy Object



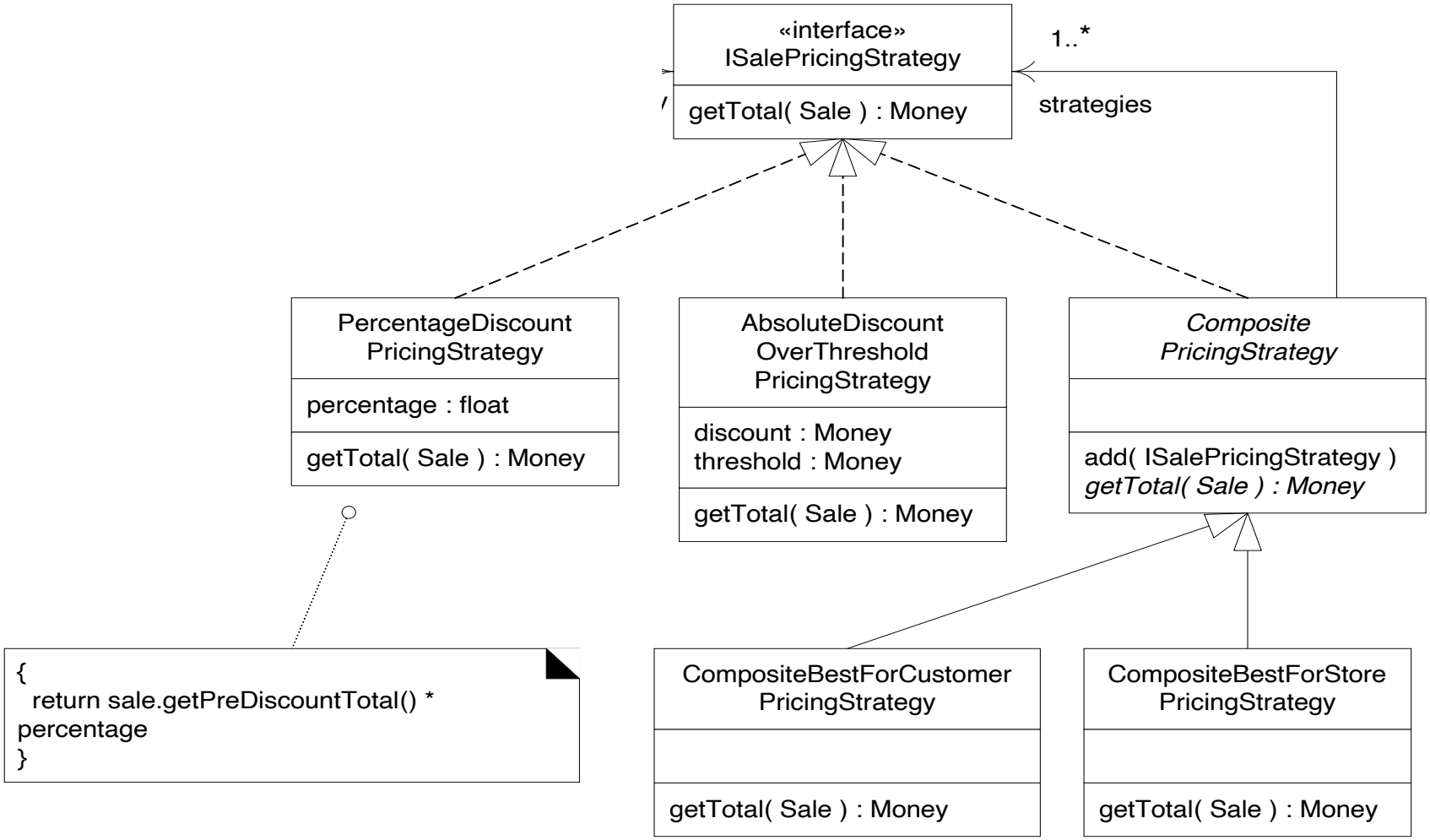
Factory Creates Strategy Object on Demand



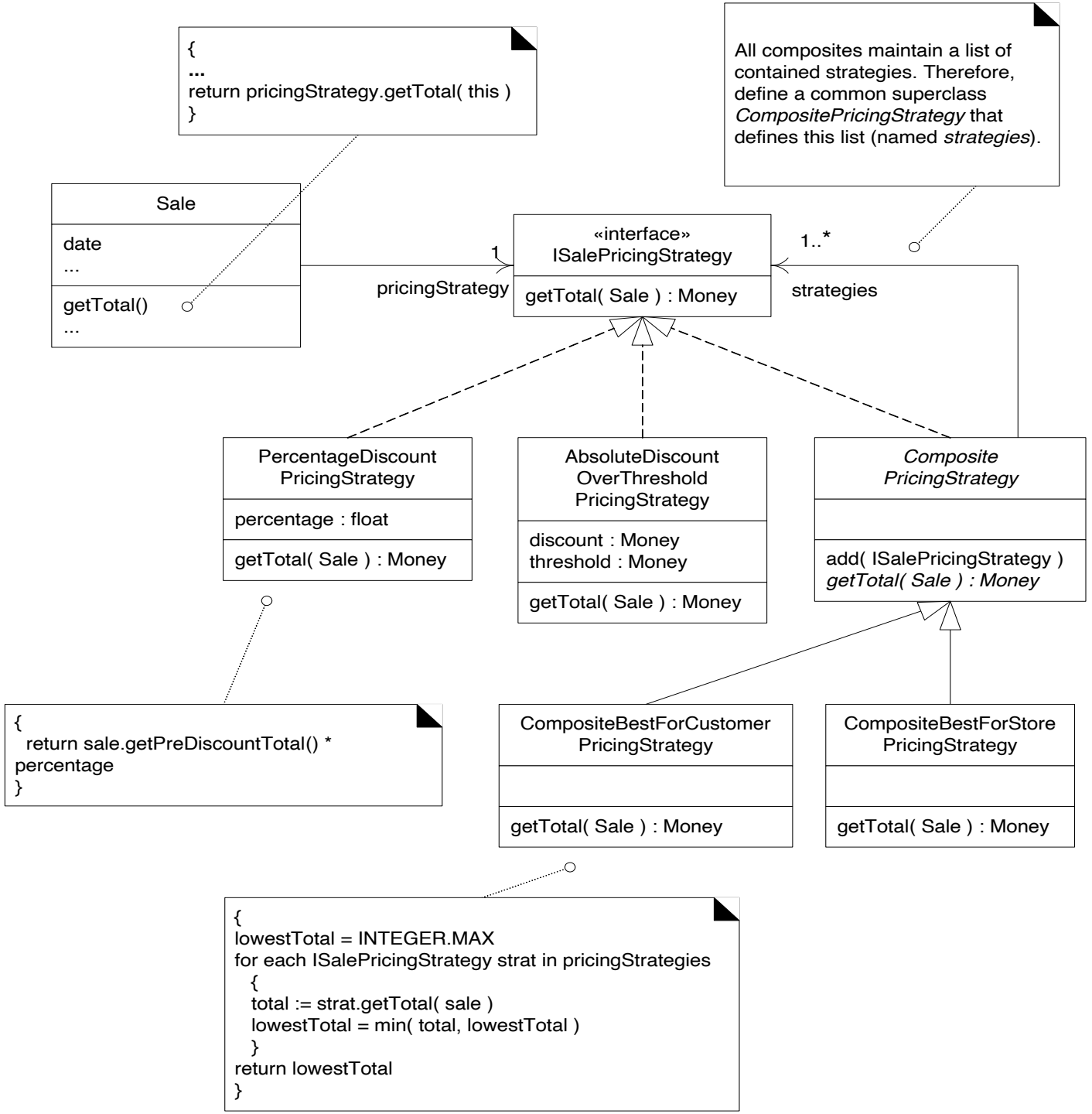
The “Composite” Design Pattern



The “Composite” Design Pattern

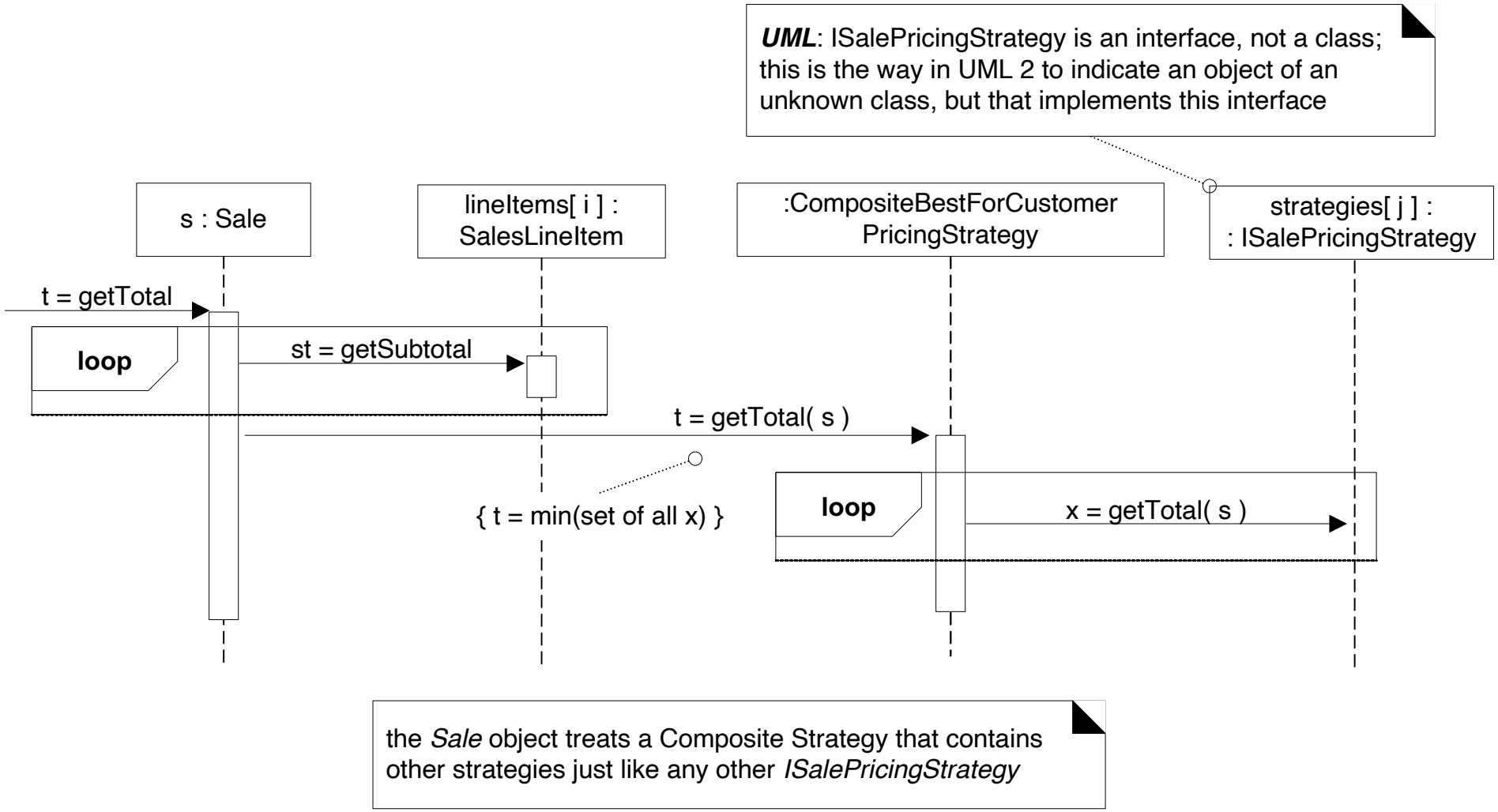


The "Composite" Design Pattern

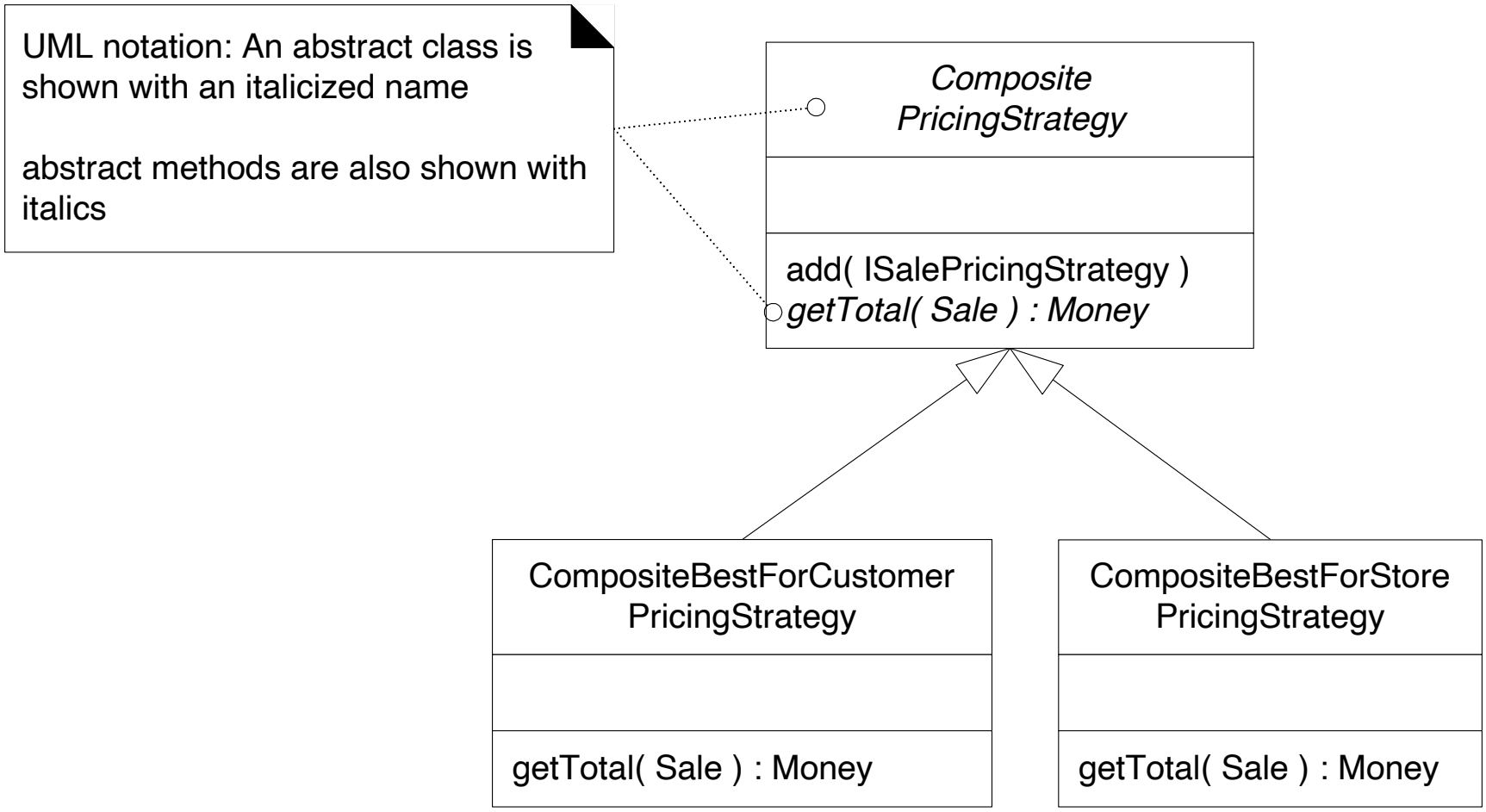


```
public abstract class CompositePricingStrategy
                        implements ISalePricingStrategy {

     protected List strategies = new ArrayList();

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

     public abstract Money getTotal( Sale sale );
}

public class ComputeBestForCustomerPricingStrategy
                       extends CompositePricingStrategy {

 Money lowestTotal = new Money( Integer.MAX_VALUE );

 for (Iterator i = strategies.iterator(); i.hasNext(); ) {
   ISalePricingStrategy strategy =
           (ISalePricingStrategy) i.next();
   Money total = strategy.getTotal( sale );
   lowestTotal = total.min( lowestTotal);
 }
 return lowestTotal;
}
```

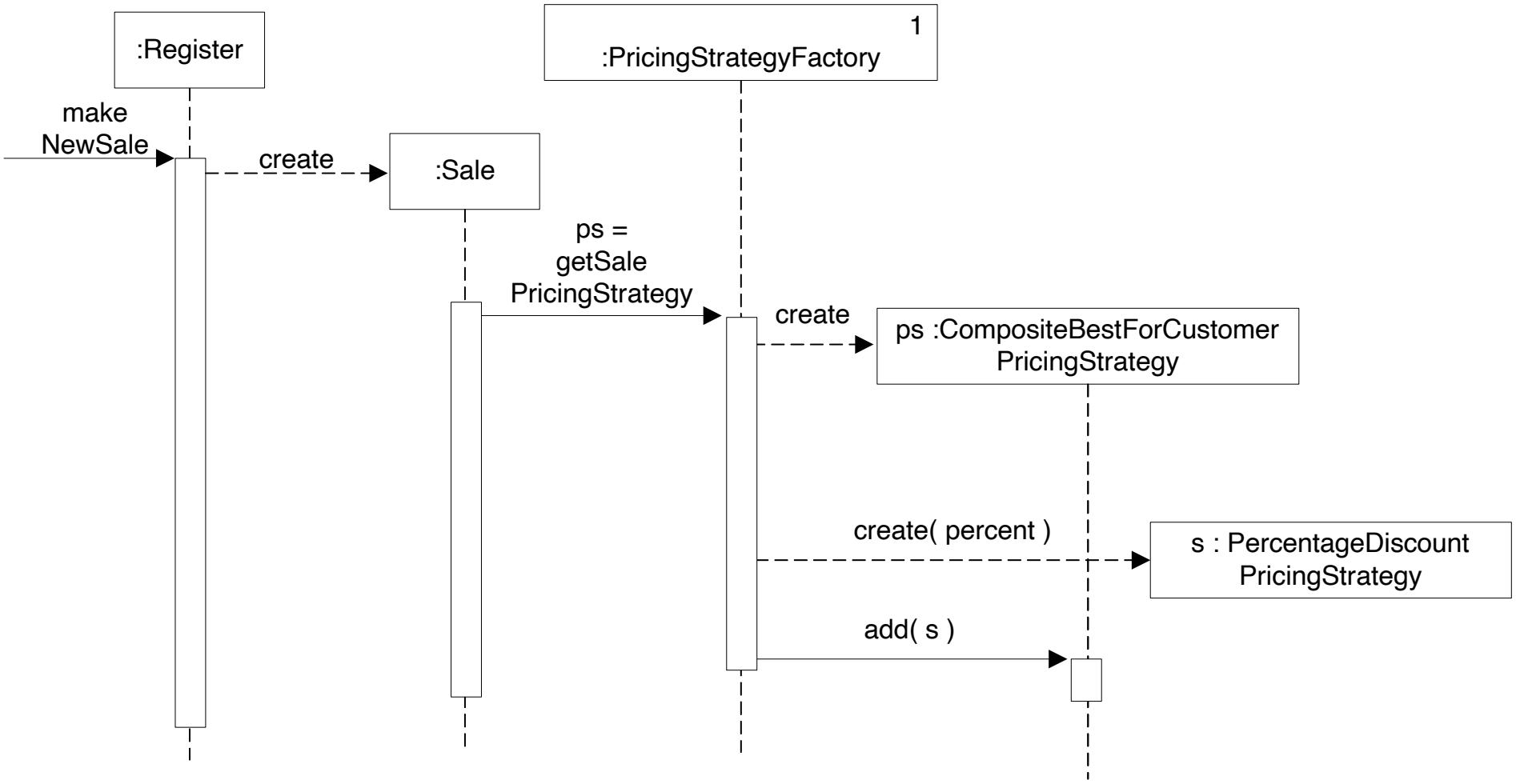
Collaboration with a Composite



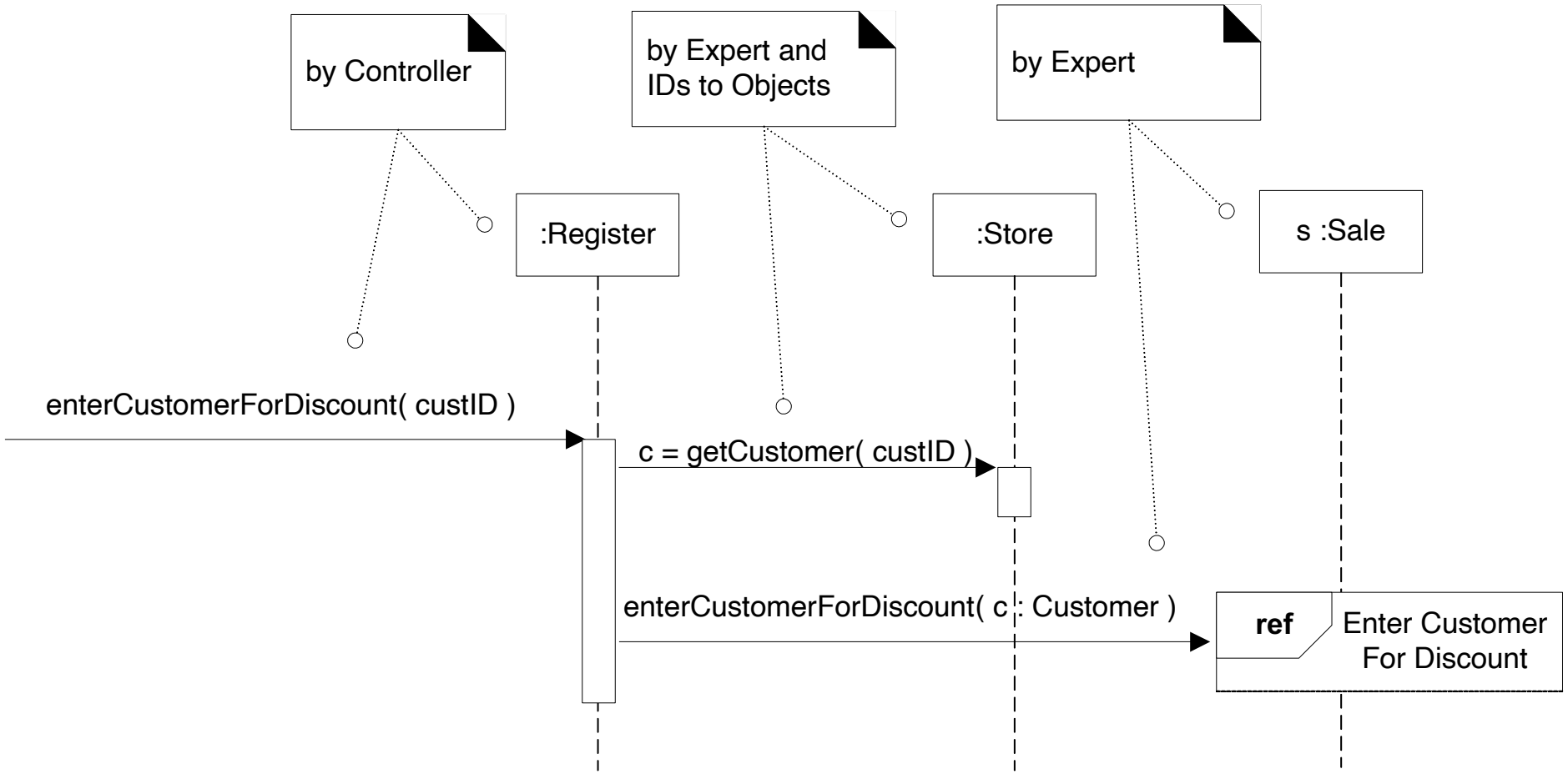
Abstract Classes and Methods in the UML



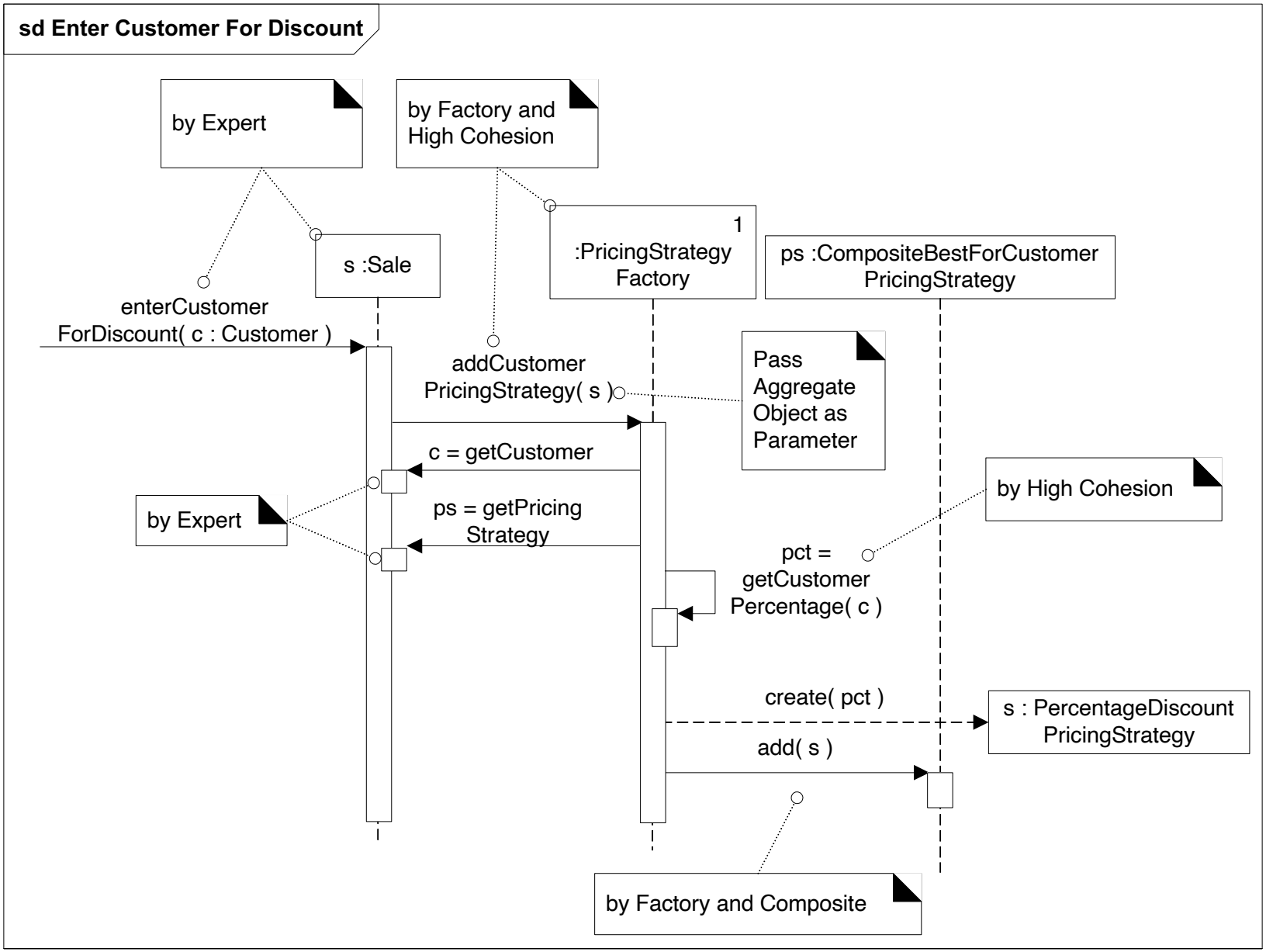
Creating a Composite Strategy



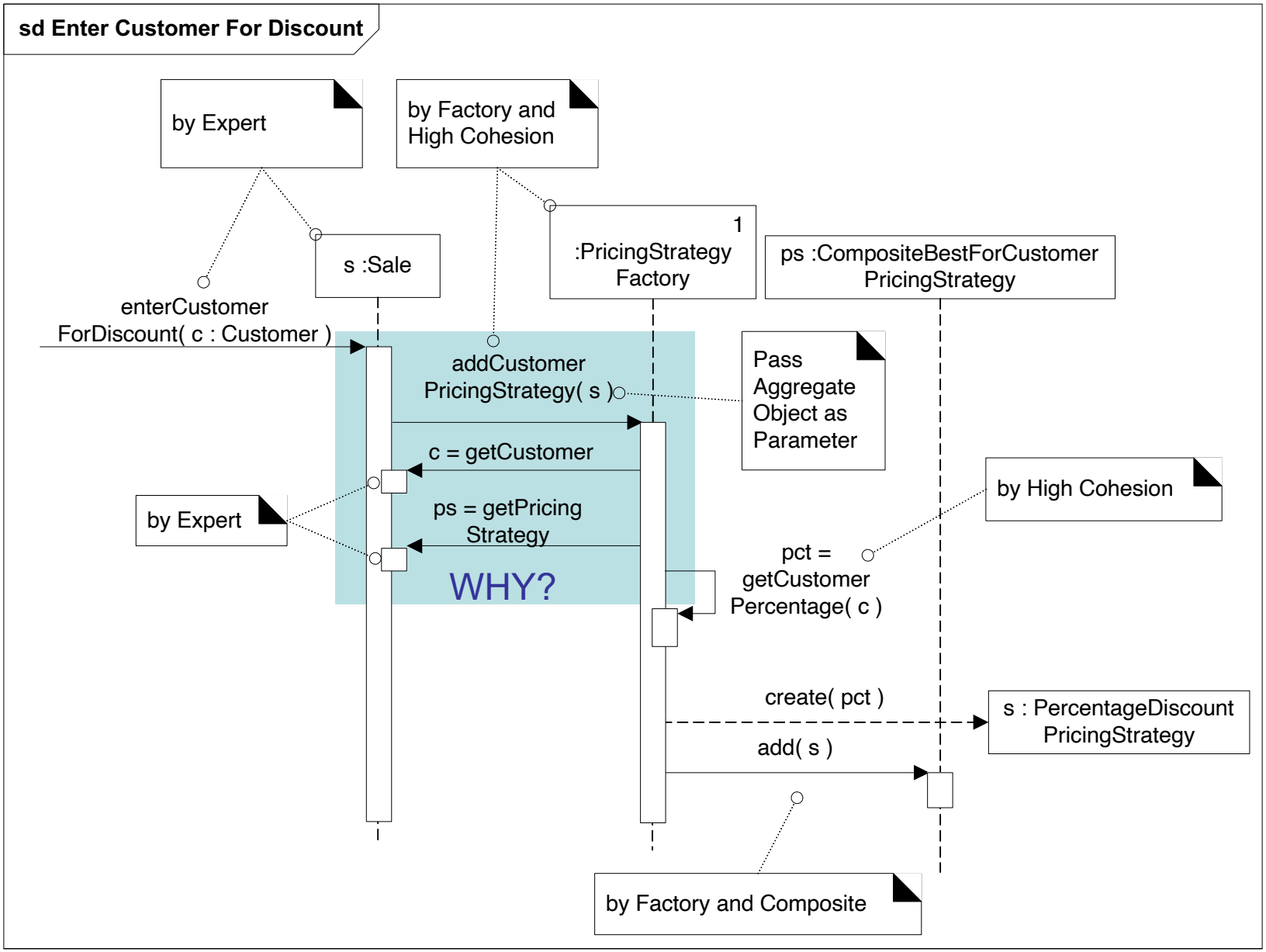
Creating the Pricing Strategy for a Customer (Part 1)



Creating the Pricing Strategy for a Customer (Part 2)



Creating the Pricing Strategy for a Customer (Part 2)



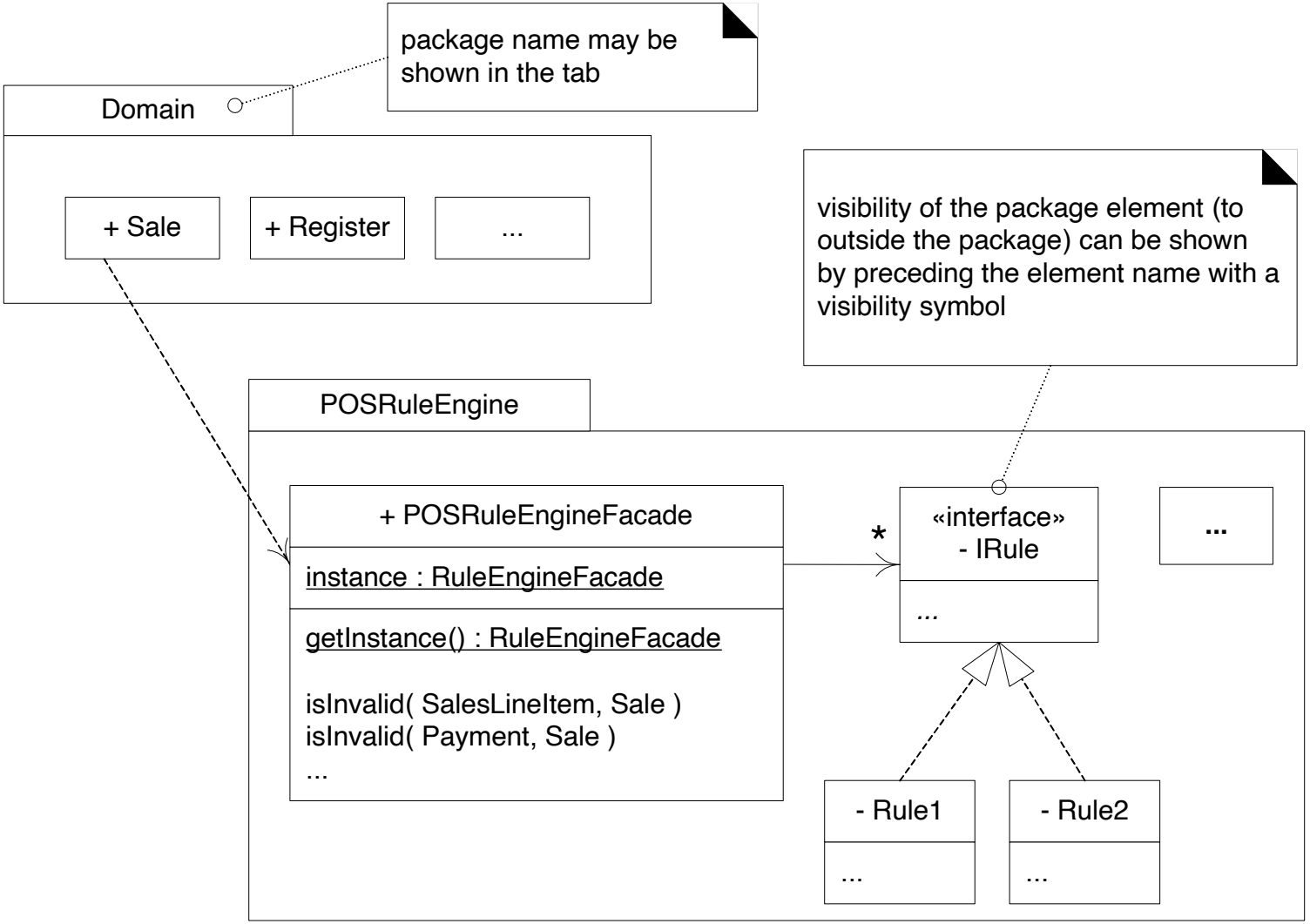
The “Façade” Design Pattern

- Additional requirement in new iteration of POS: Pluggable business rules
- Example
  - New sale might be paid by gift certificate. Only one item can be purchased by a gift certificate
  - If sale paid by gift certificate, invalidate all payments with the type of “change due back to customer” different from gift certificate.
  - Some sales might be charitable donations by the store. Limited to less than $250 each. Manager must be logged in as cashier for this transaction.
- One of the concerns: What happens to enterItem?

enterItem and Low Impact of Change

- Suppose architect wants low impact of change in pluggable business rules.
- Suppose also that the architect is not sure how to implement the pluggable business rules.
  - Wants to experiment with different ways of implementing them.
- Solution: The “Façade” design pattern
  - Problem: Need a common, unified interface to a disparate set of implementations or interfaces
  - Solution: Define a single point of contact to the subsystem containing the implementations
    - This façade object has a unified interface
    - Internal implementation details hidden
  - Example: The use-case controller objects in your project.

The “Façade” Design Pattern



Façade code example

```
public class Sale {

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

   // call to the Façade. Notice Singleton pattern

   if (POSRuleEngineFacede.getInstance().isInvalid(sli,this) )
       return;
    lineItems.add(sli);
   }
   //..
}
```

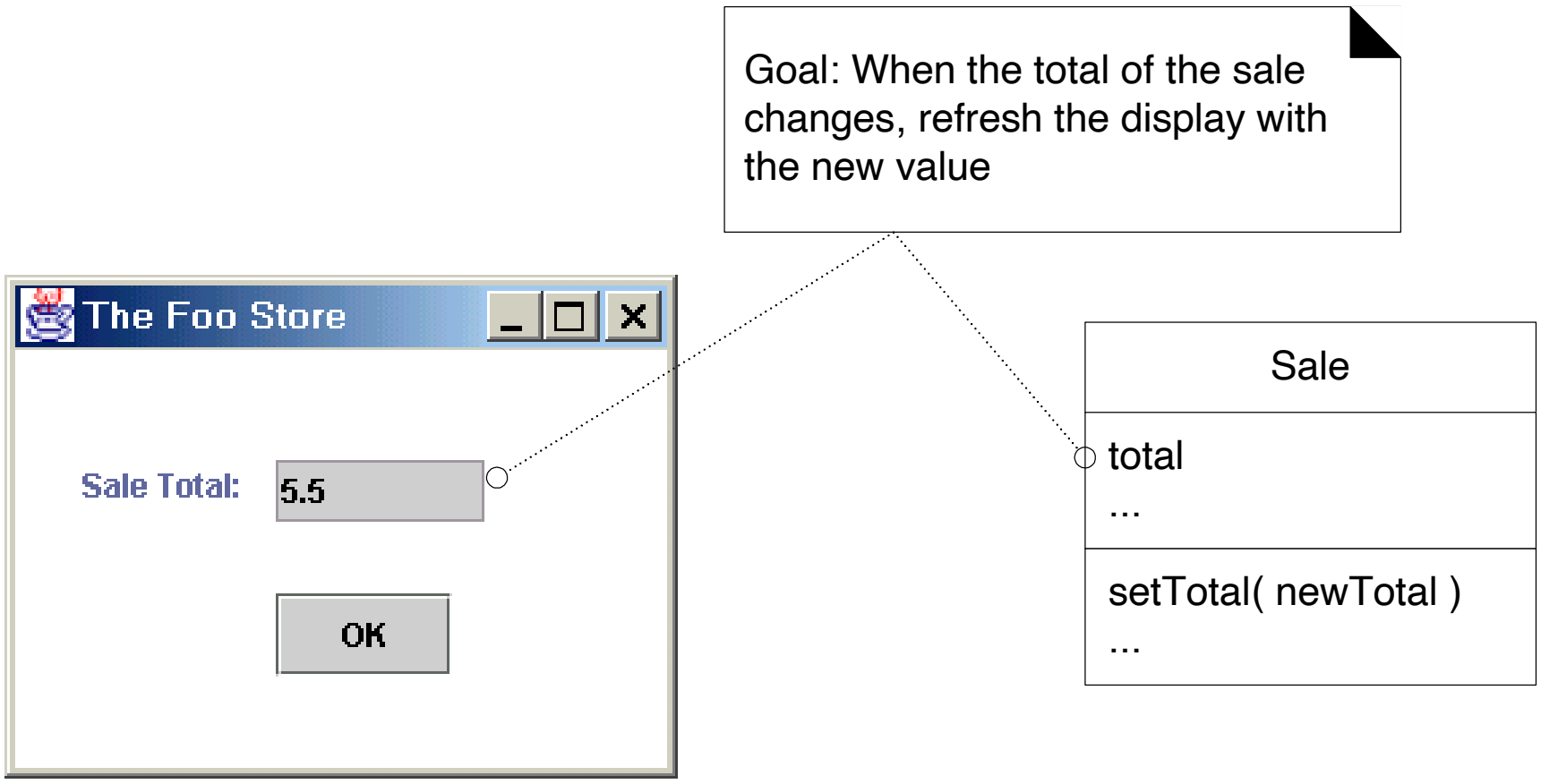
Observer/Publish-Subscribe/Delegation Event Model

- Another requirement: GUI window refreshes its display of the sales total when the total changes
  - Later: GUI updates display when other data changes as well
- What's wrong with the following?
  - When the Sale object changes its total, it sends a message to a window, asking it to refresh the display?

Publish-Subscribe Pattern

- What's wrong with the following?
  - When the Sale object changes its total, it sends a message to a window, asking it to refresh the display?
- Answer: The Model-View Separation principle discourages such solutions
  - Model objects (objects in the domain) should not know of UI objects
  - If UI changes, model objects should not need to change

The Problem


Goal: When the total of the sale changes, refresh the display with the new value
The Foo Store
Sale Total:
5.5
OK
Sale
total
...
setTotal(newTotal)
...

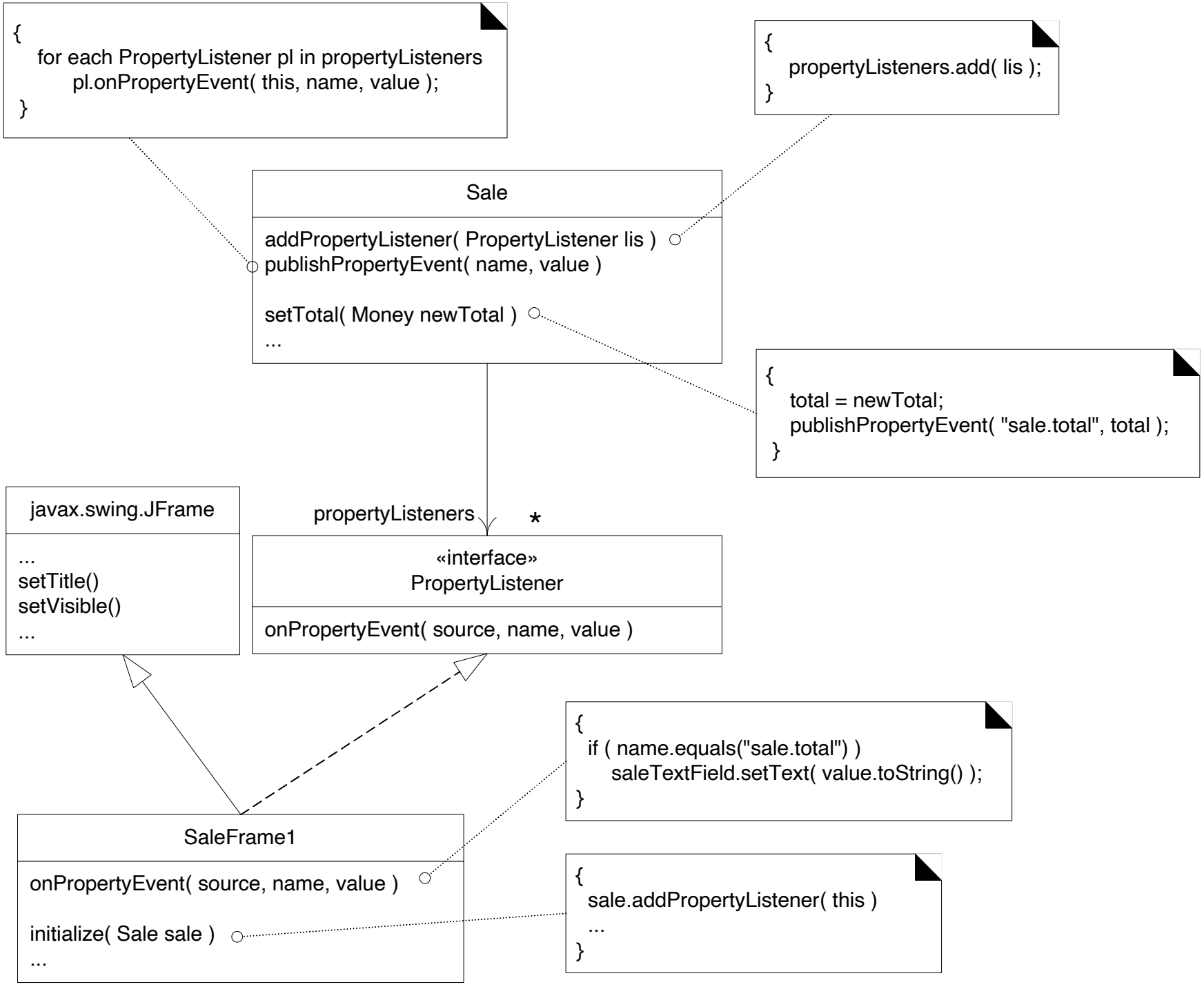

{
for each PropertyListener pl in propertyListeners
pl.onPropertyEvent(this, name, value);
}
{
propertyListeners.add(lis);
}
Sale
addPropertyListener(PropertyListener lis)
publishPropertyEvent(name, value)
setTotal(Money newTotal)
...
{
total = newTotal;
publishPropertyEvent("sale.total", total);
}
javax.swing.JFrame
...
setTitle()
setVisible()
...
propertyListeners
*
«interface»
PropertyListener
onPropertyEvent(source, name, value)
{
if (name.equals("sale.total"))
saleTextField.setText(value.toString());
}
SaleFrame1
onPropertyEvent(source, name, value)
initialize(Sale sale)
...
{
sale.addPropertyListener(this)
...
}

Fig. 26.23

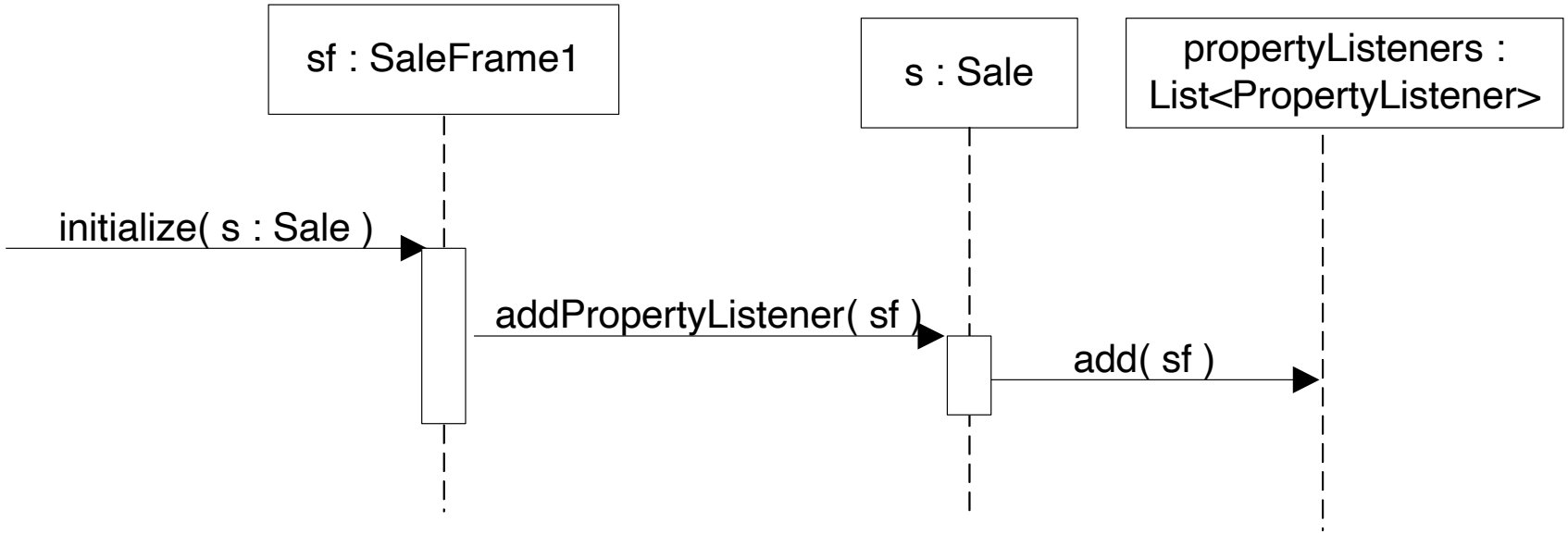

sf : SaleFrame1
s : Sale
propertyListeners :
List<PropertyListener>
initialize(s : Sale)
addPropertyListener(sf)
add(sf)

Fig. 26.24


s :Sale
propertylisteners[i] :
PropertyListener
setTotal(total)
publishPropertyEvent
("sale.total", total)
loop
onPropertyEvent(s, "sale.total", total)

Fig. 26.25

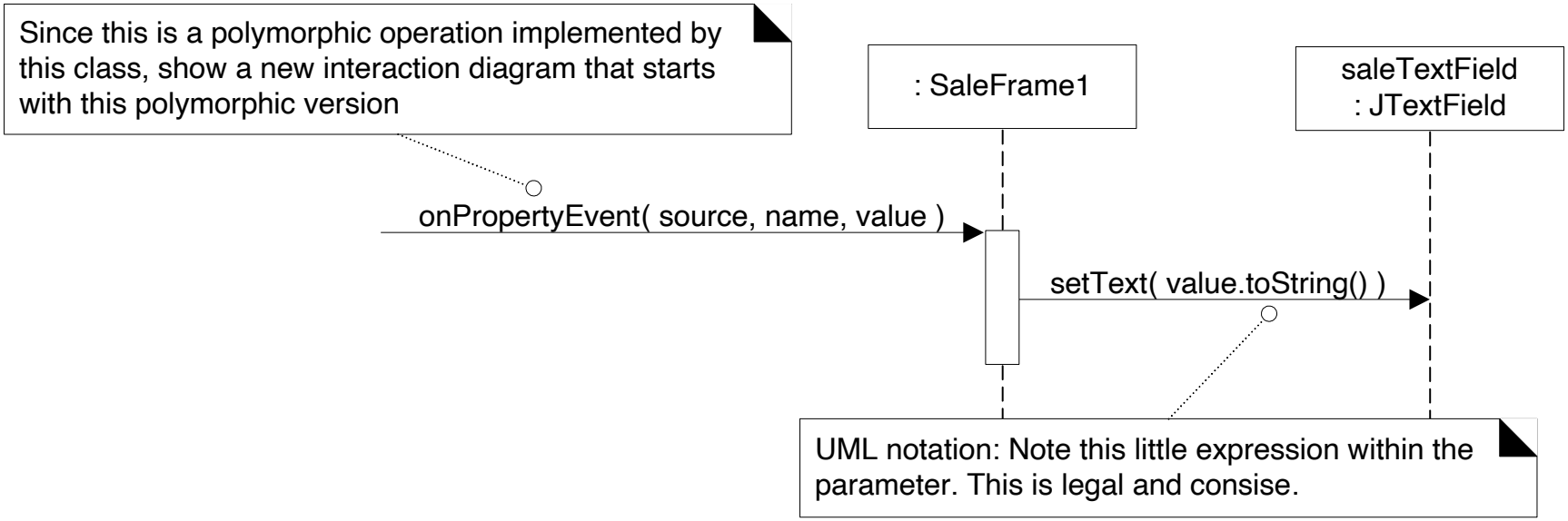

Since this is a polymorphic operation implemented by this class, show a new interaction diagram that starts with this polymorphic version
: SaleFrame1
saleTextField
: JTextField
onPropertyEvent(source, name, value)
setText(value.toString())
UML notation: Note this little expression within the parameter. This is legal and consise.

Fig. 26.26

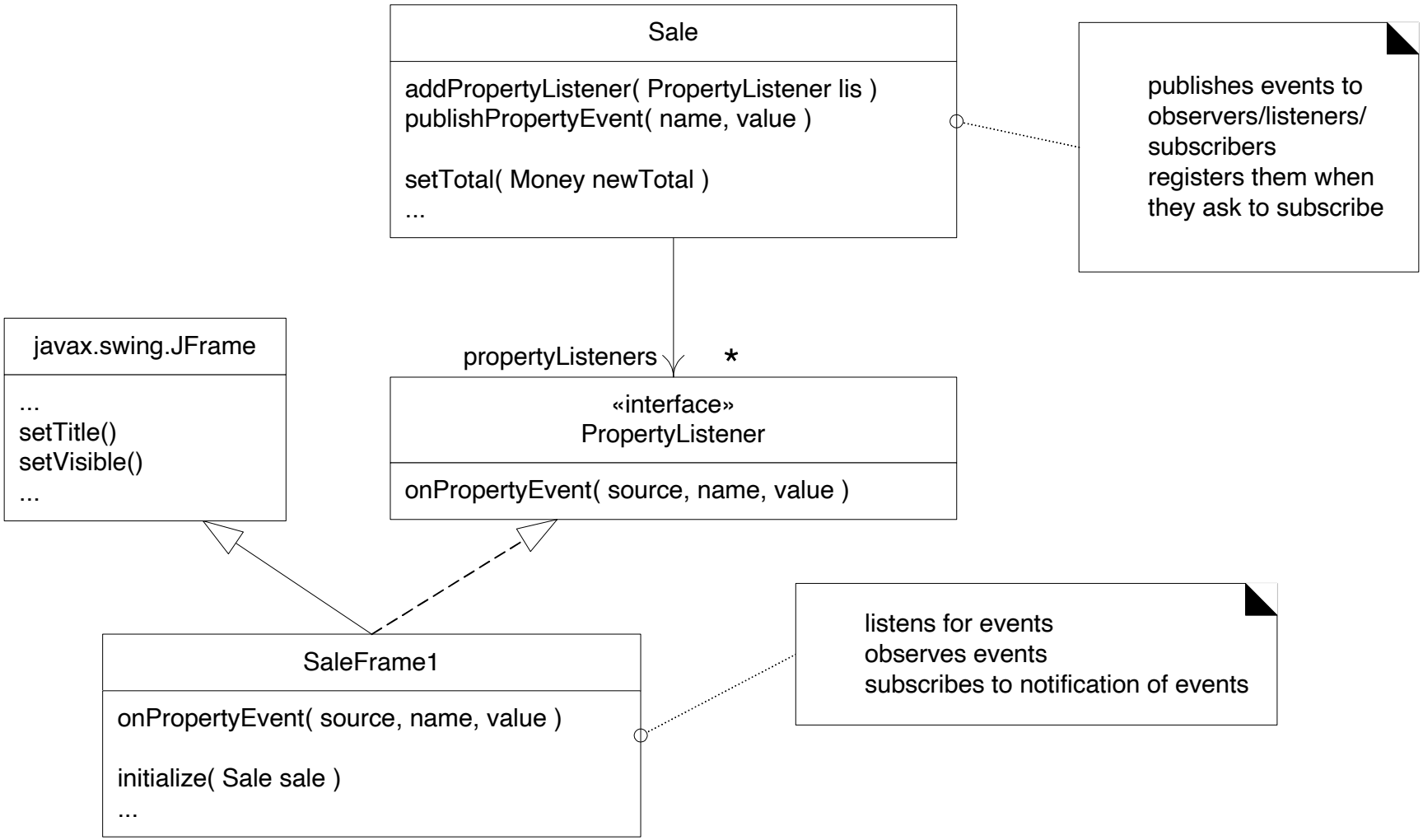
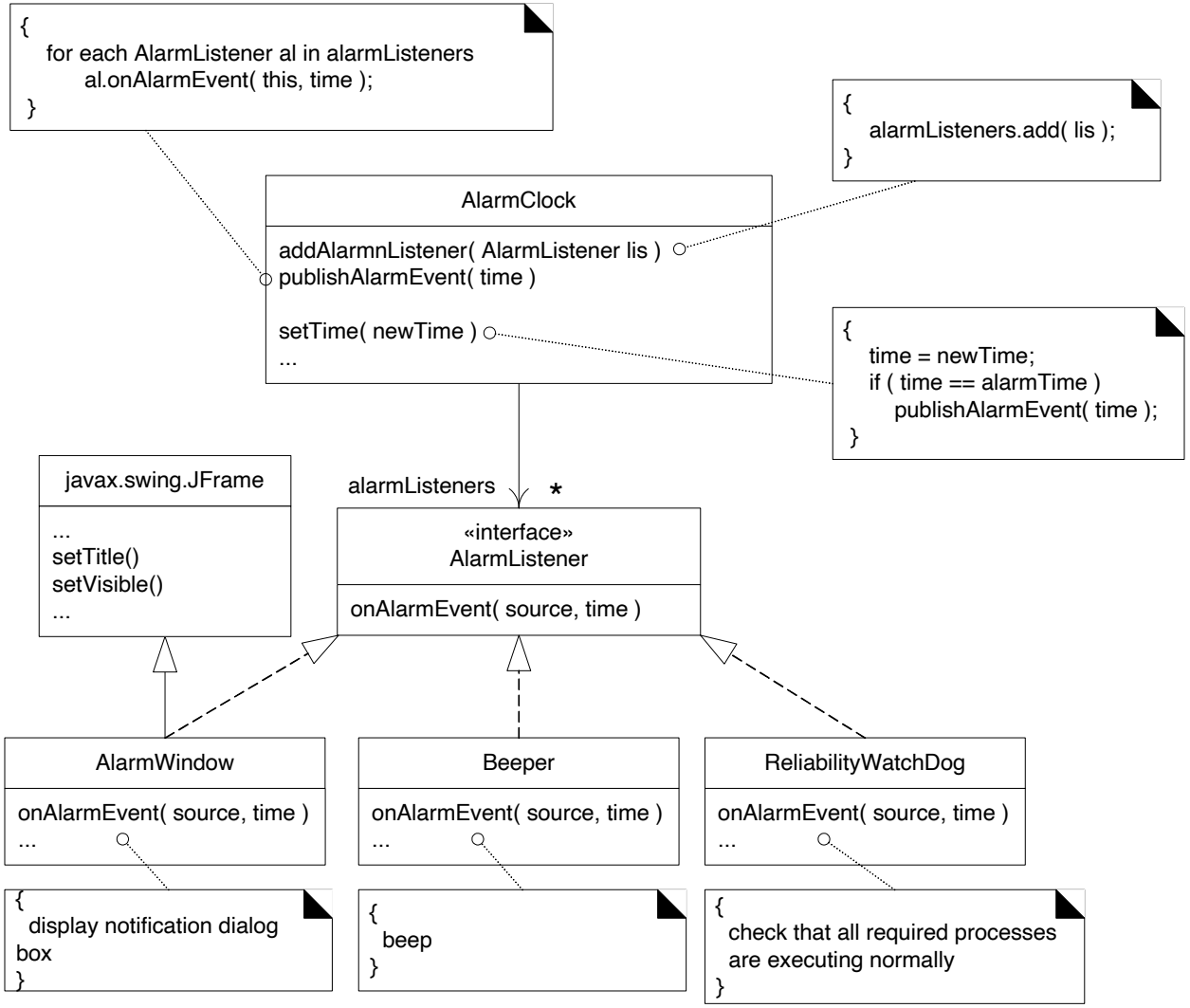

Sale
addPropertyListener(PropertyListener lis)
publishPropertyEvent(name, value)
setTotal(Money newTotal)
...
publishes events to observers/listeners/ subscribers registers them when they ask to subscribe
javax.swing.JFrame
...
setTitle()
setVisible()
...
propertyListeners
*
«interface»
PropertyListener
onPropertyEvent(source, name, value)
SaleFrame1
onPropertyEvent(source, name, value)
initialize(Sale sale)
...
listens for events
observes events
subscribes to notification of events

Fig. 26.27


{
for each AlarmListener al in alarmListeners
al.onAlarmEvent(this, time);
}
{
alarmListeners.add(lis);
}
AlarmClock
addAlarmnListener(AlarmListener lis)
publishAlarmEvent(time)
setTime(newTime)
...
{
time = newTime;
if (time == alarmTime)
publishAlarmEvent(time);
}
javax.swing.JFrame
...
setTitle()
setVisible()
...
alarmListeners
*
«interface»
AlarmListener
onAlarmEvent(source, time)
AlarmWindow
onAlarmEvent(source, time)
...
Beeper
onAlarmEvent(source, time)
...
ReliabilityWatchDog
onAlarmEvent(source, time)
...
{
display notification dialog
box
}
{
beep
}
{
check that all required processes
are executing normally
}