More Patterns

Software Design and Analysis

CSCI 2040

Objectives

- Learn to apply the remaining GRASP patterns.
- Apply GoF design patterns.

Introduction

- We explored the first five GRASP patterns:
  - Information Expert, Creator, High Cohesion, Low Coupling, and Controller
- **The final four GRASP patterns are:**
  - **Polymorphism**
  - **Indirection**
  - **Pure Fabrication**
  - **Protected Variations**
- **Some of "gang-of-four" (GoF) design patterns**
  - such as **Strategy and Factory**

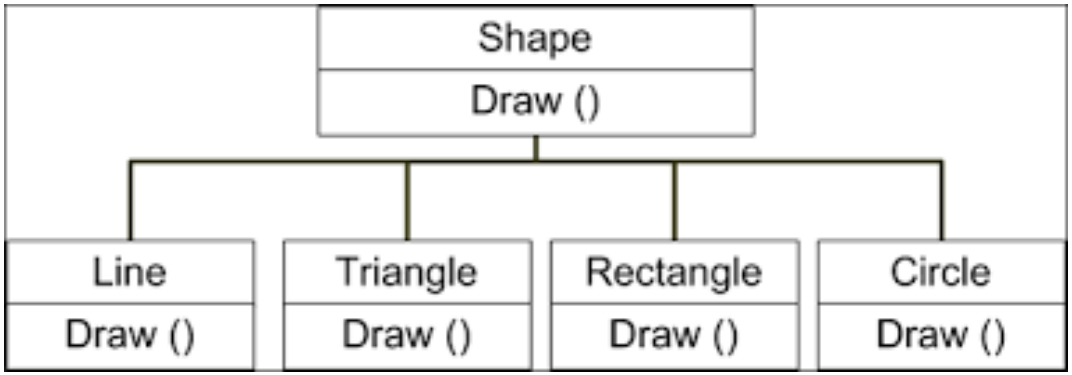
More Grasp Patterns

Polymorphism

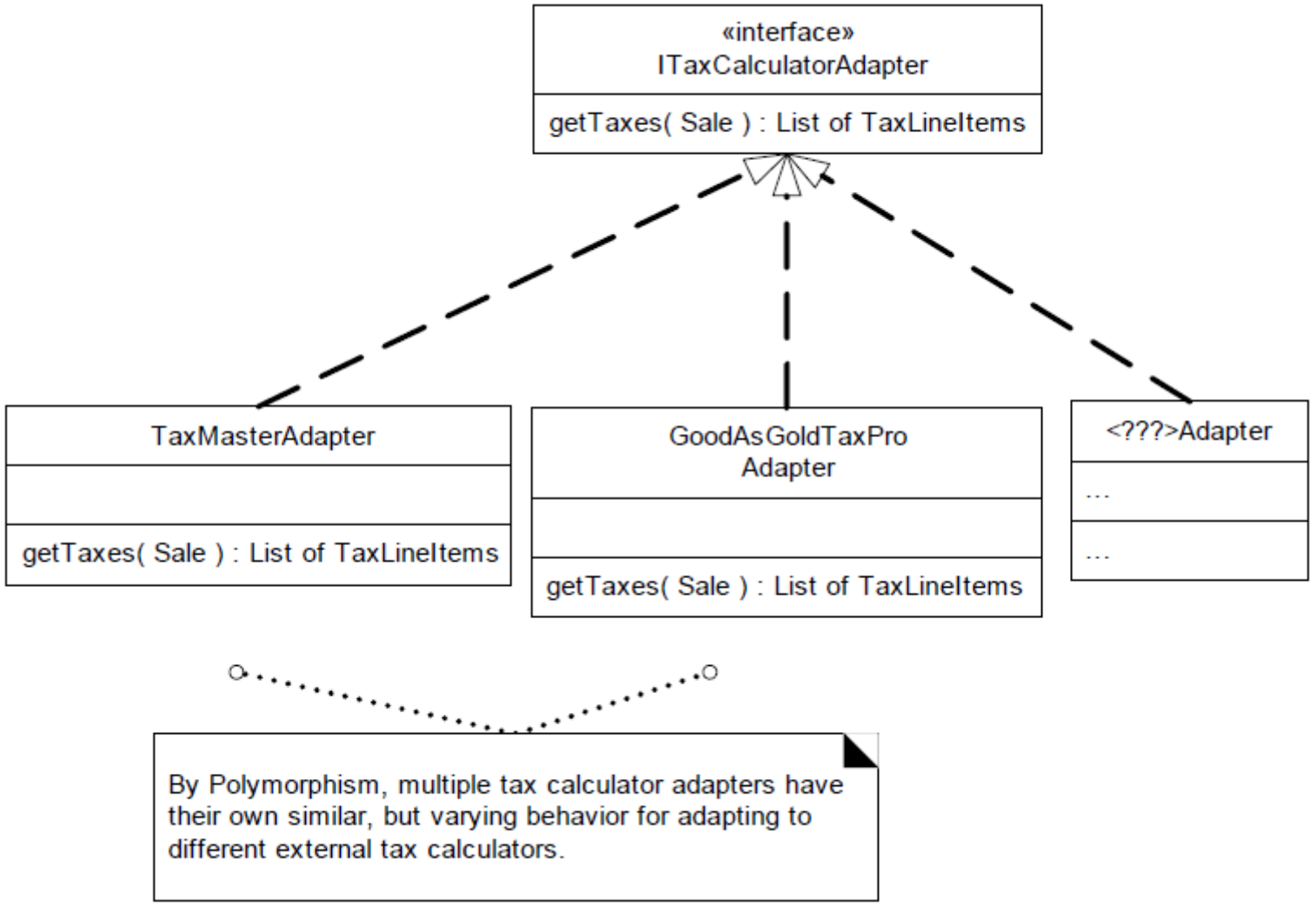
- **Problem:**
  - How to handle alternatives based on type?
  - How to create pluggable software components?
- **Example:**
  - In the NextGen POS application, there are multiple external third-party tax calculators that must be supported with different API
    - **such as Tax-Master and Good-As-Gold Tax-Pro**.
- **Solution**
  - Assign responsibility for the behavior using polymorphic operations.

Polymorphism

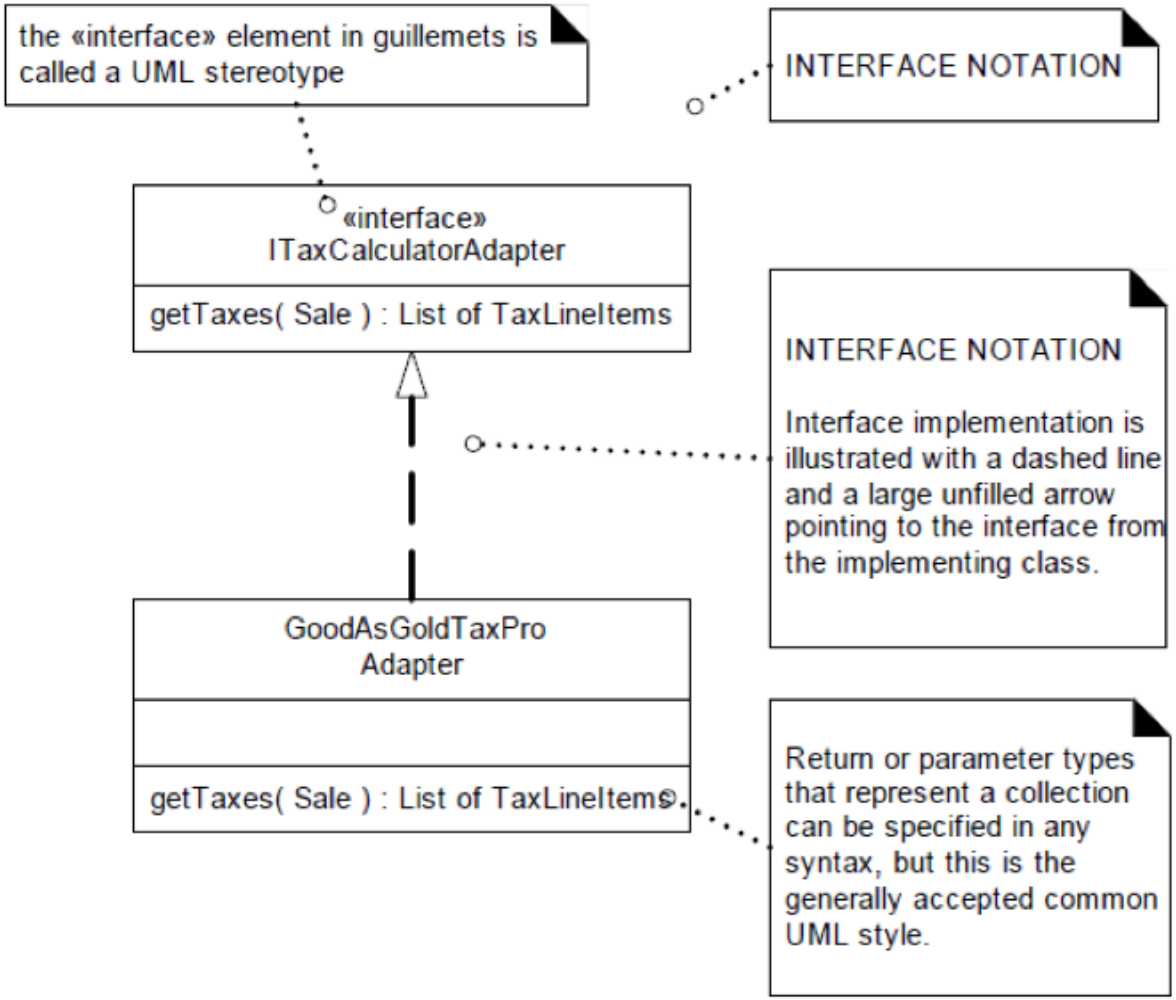
- Polymorphism - giving the same name to services in different objects.



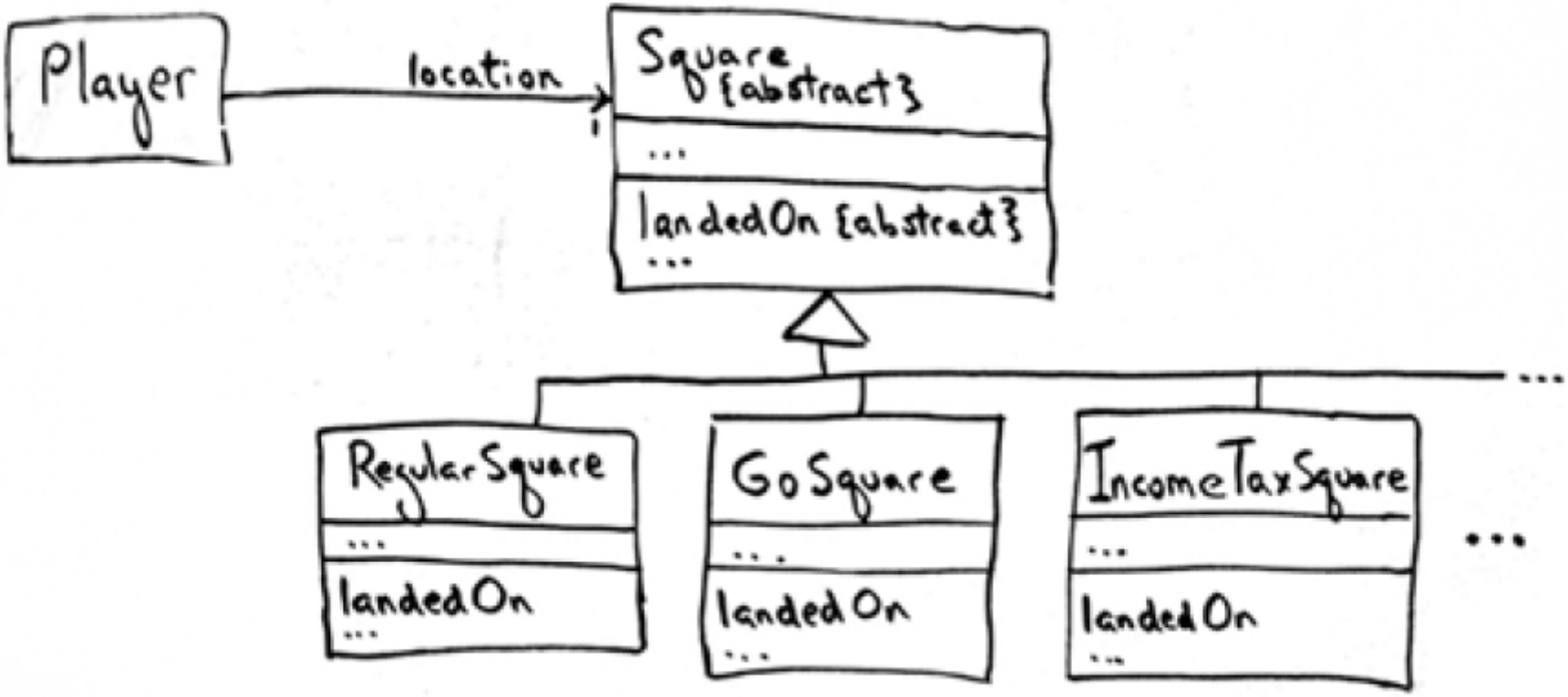
Polymorphism in Tax Calculators



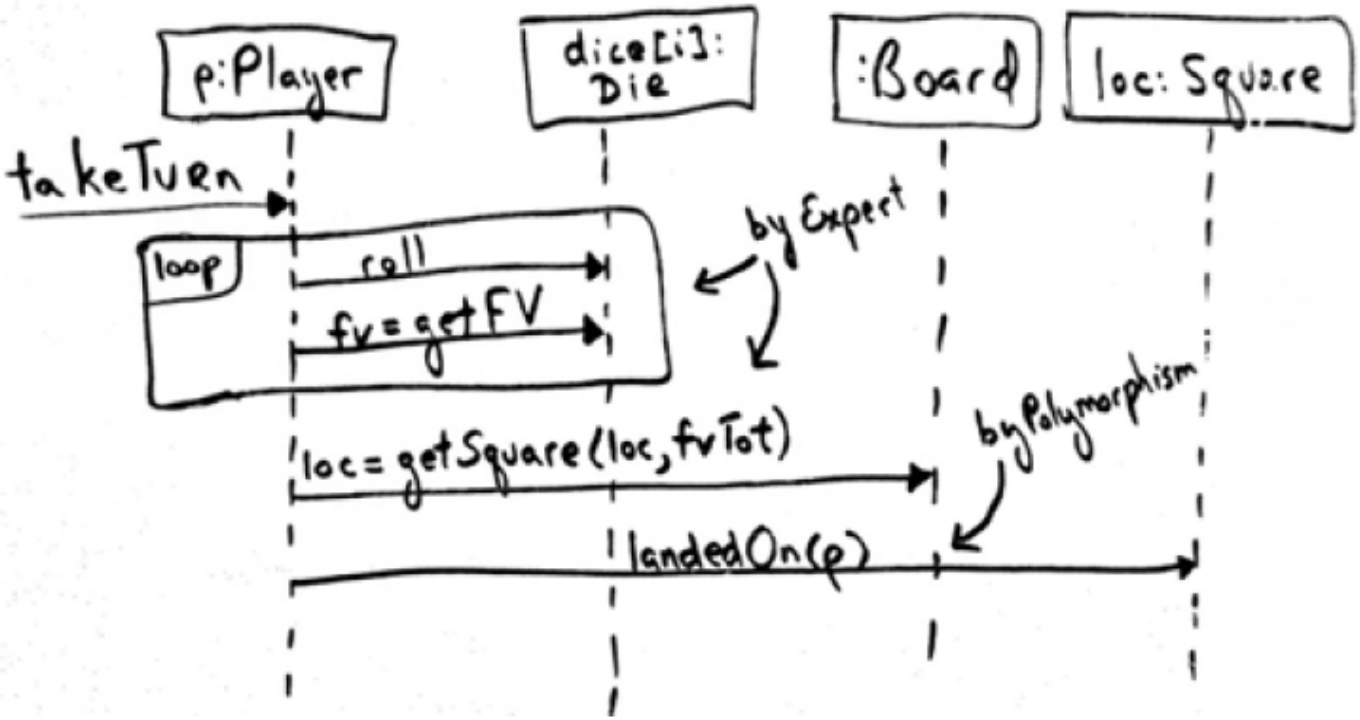
UML Notation for Interfaces and Return Types



Applying Polymorphism to Monopoly



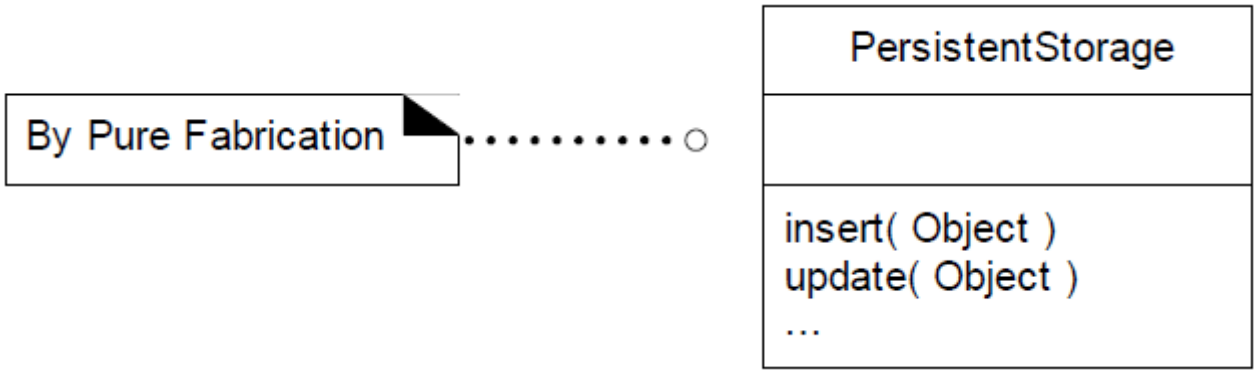
The GoSquare Case



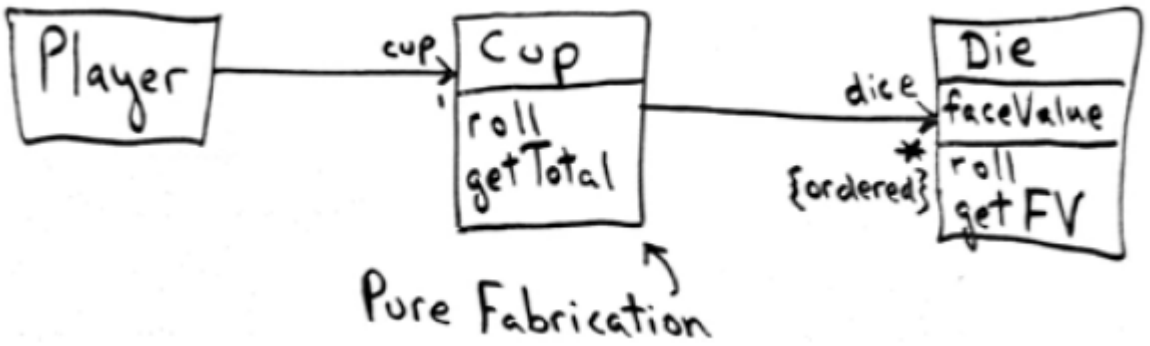
Pure Fabrication

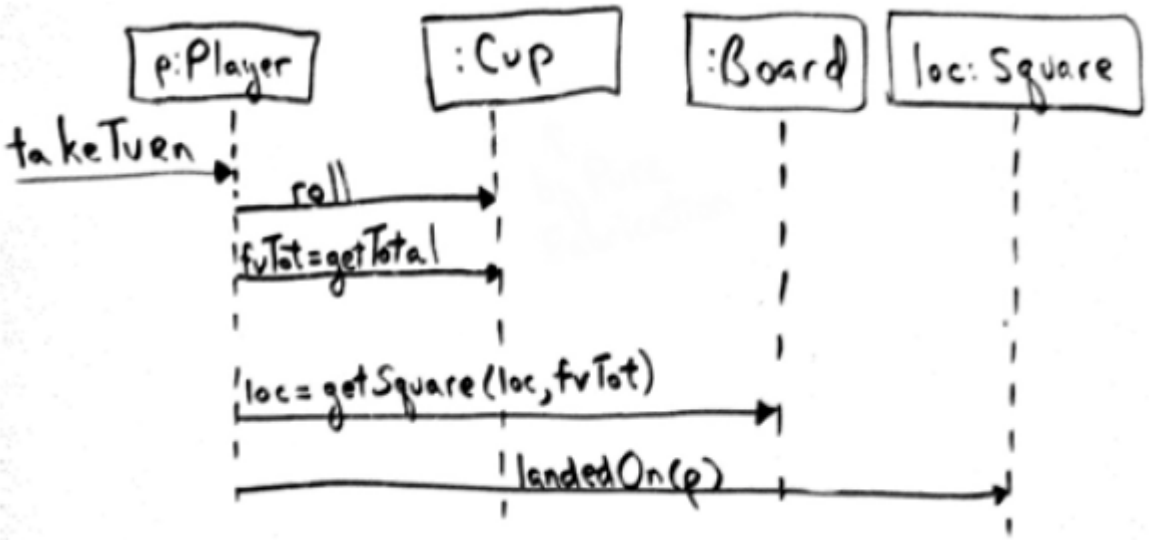
- **Problem:**
  - How to assign a highly cohesive set of responsibilities to an artificial class that does not represent a problem domain concept?
- **Example:**
  - Suppose that support is needed to save *Sale* instances in a relational database
- **Solution**
  - A reasonable solution is to create a new class that is solely responsible for the task
    - something made up, to support reuse

Pure Fabrication



Using the Cup in the Monopoly Game

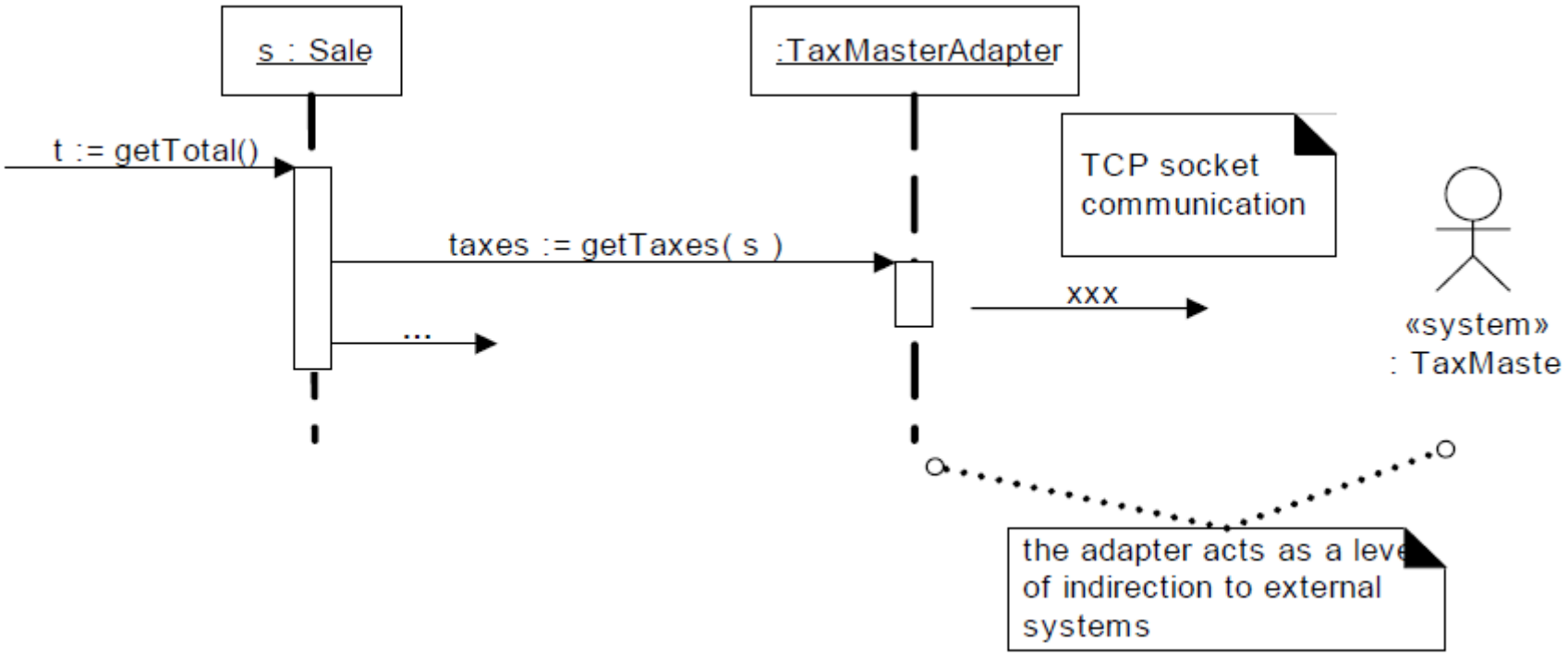




Indirection

- **Problem:**
  - Where to assign a responsibility, to avoid direct coupling between two (or more) things?
  - How to de-couple objects so that low coupling is supported and reuse
- **Solution:**
  - Assign the responsibility to an intermediate object to mediate between other components or services so that they are not directly coupled.

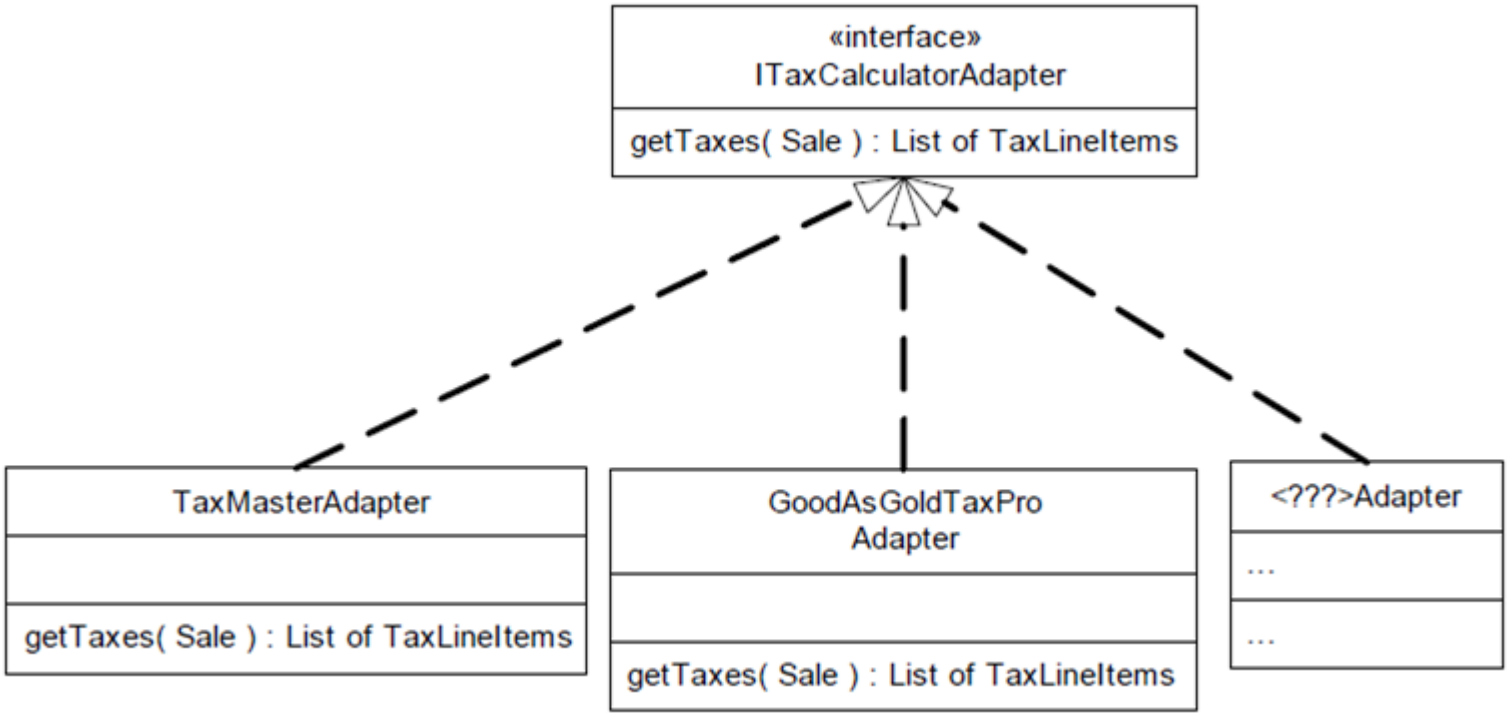
Indirection via Adapter



Protected Variations

- **Problem:**
  - How to design objects so that the variations do not have an undesirable impact?
- **Example:**
  - The point of variation is the different APIs of external tax calculators.
- **Solution**
  - Identify points of predicted variations;
  - Assign responsibilities to create a stable interface around them.

Protected Variations



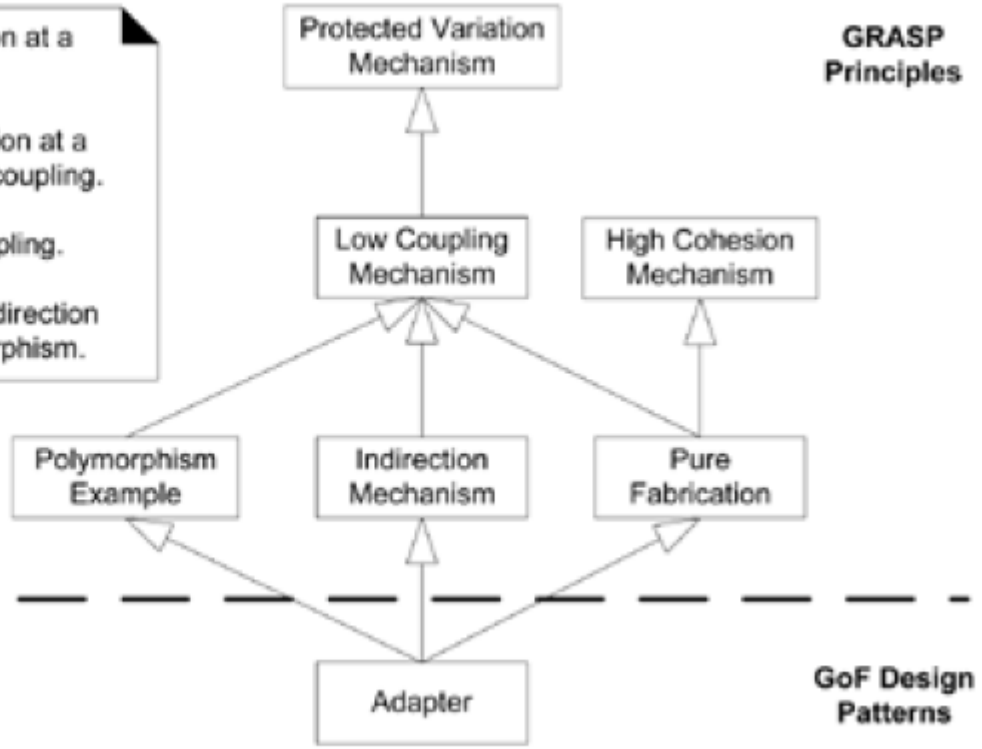
GRASP Summary

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.

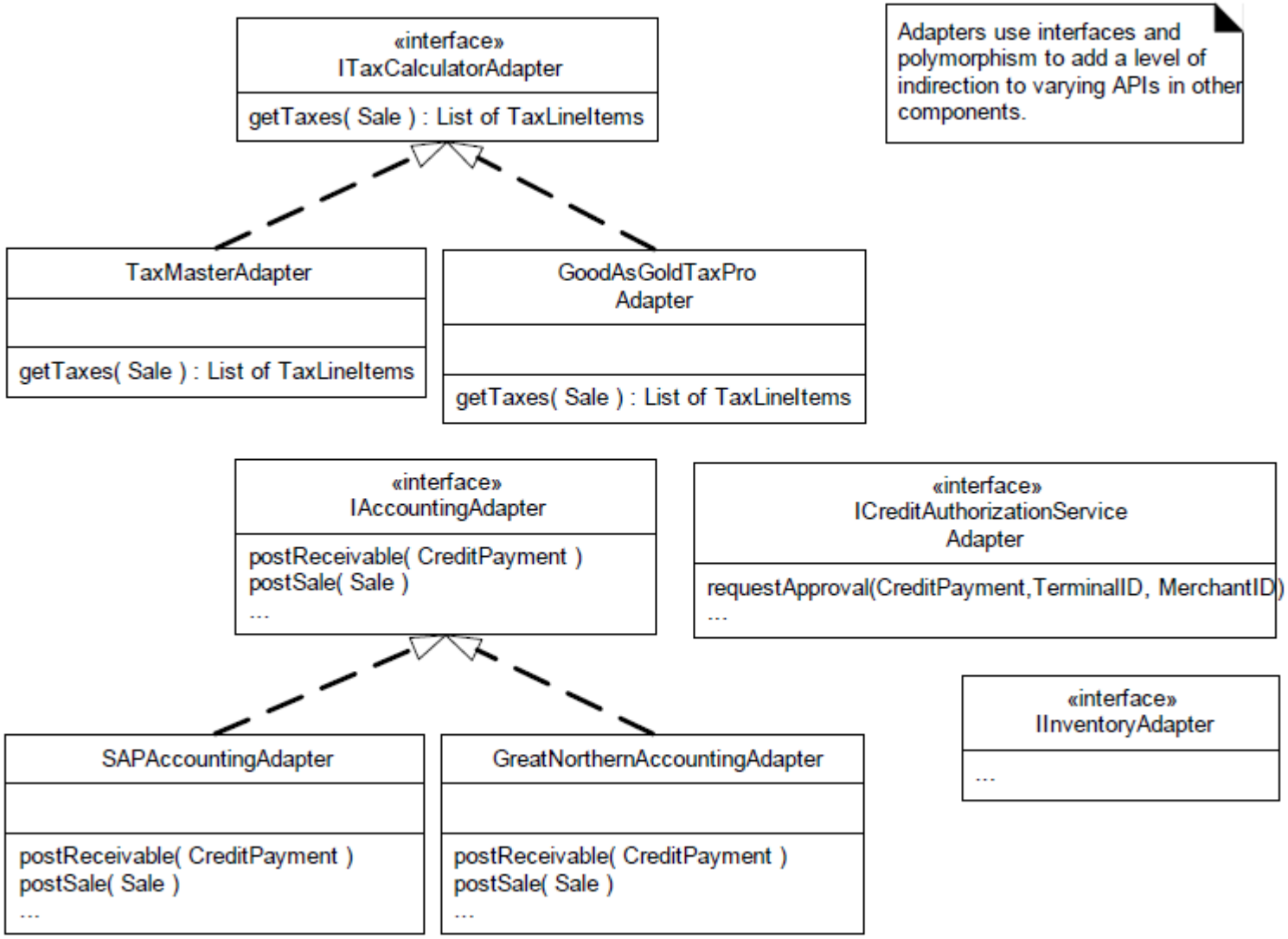
An indirection is a way to achieve low coupling.

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

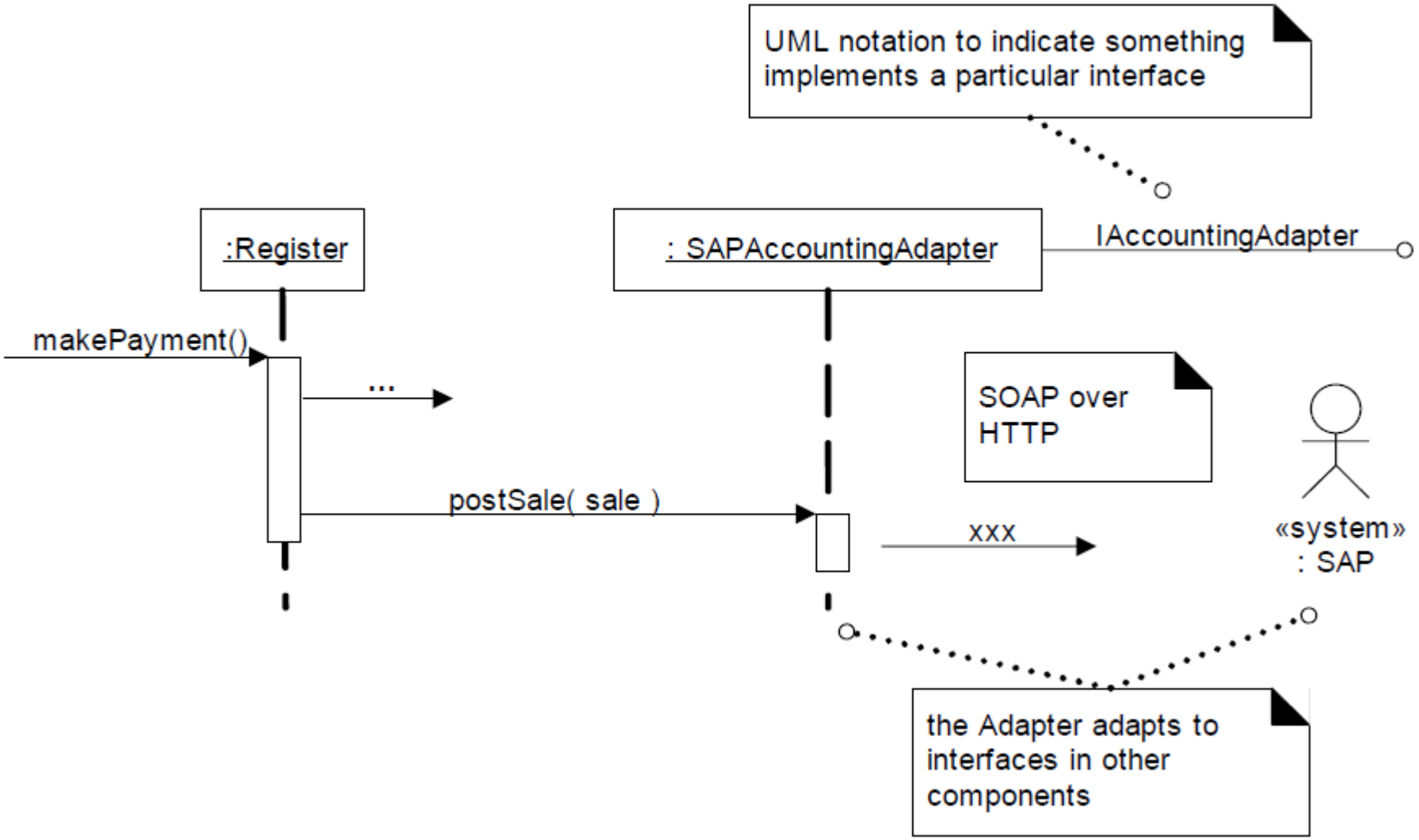


GoF Patterns

GoF Adapter Pattern

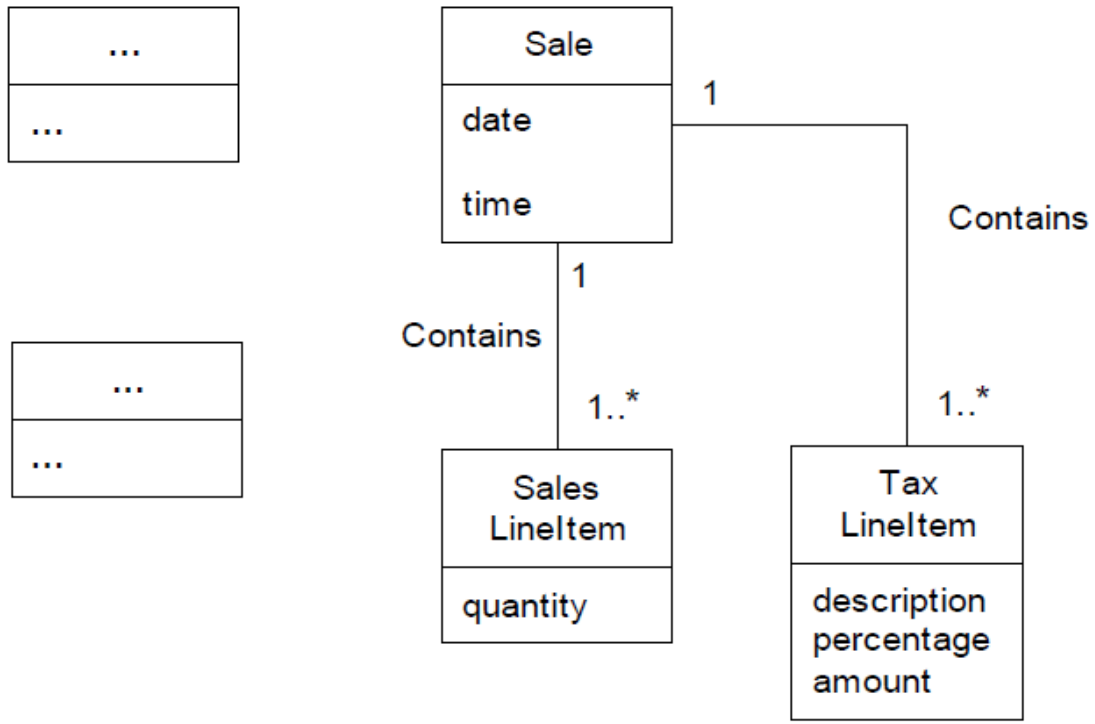


Using Adapter



Partial Domain Model

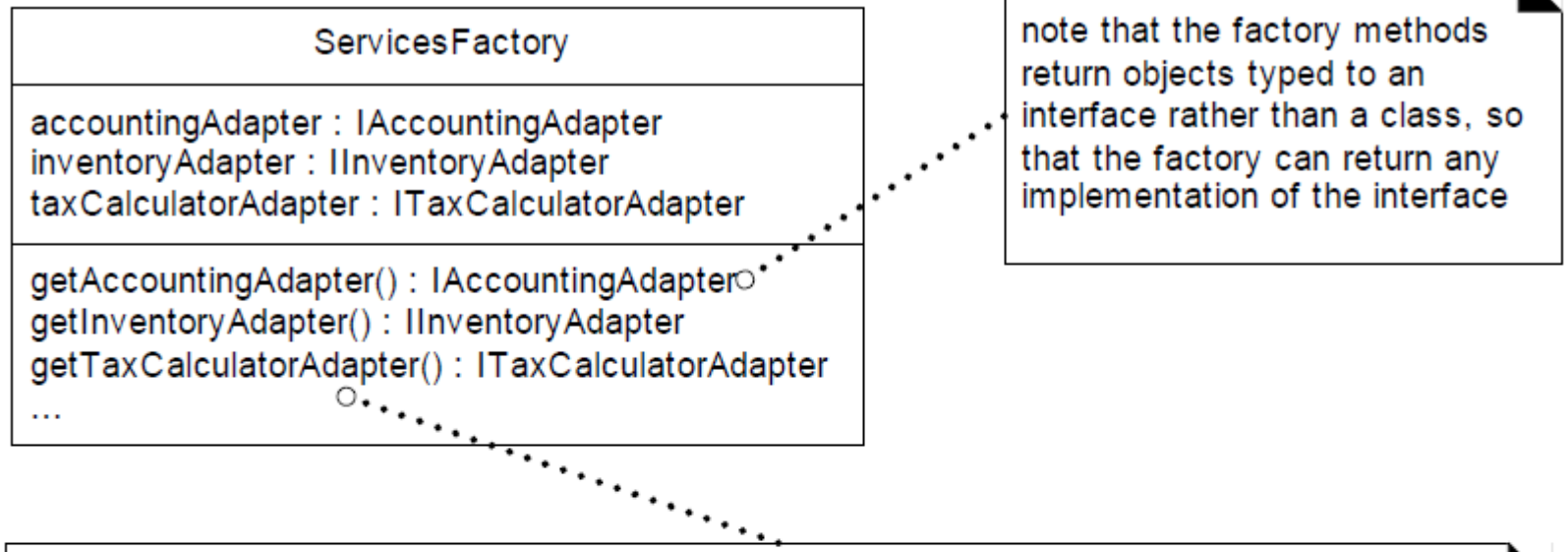
- Observe that in the Adapter design in the *getTaxes* operation returns a list of *TaxLineItems.*
- Adapter is not part of the Domain Model but used the classes from domain model
-



GoF Factory

- **Problem:**
  - Who should be responsible for creating objects when there are special considerations,
    - such as complex logic for better cohesion?
- **Solution:**
  - Create a Pure Fabrication object called a Factory that handles the creation.

Factory Pattern




```
{
  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;
}
```

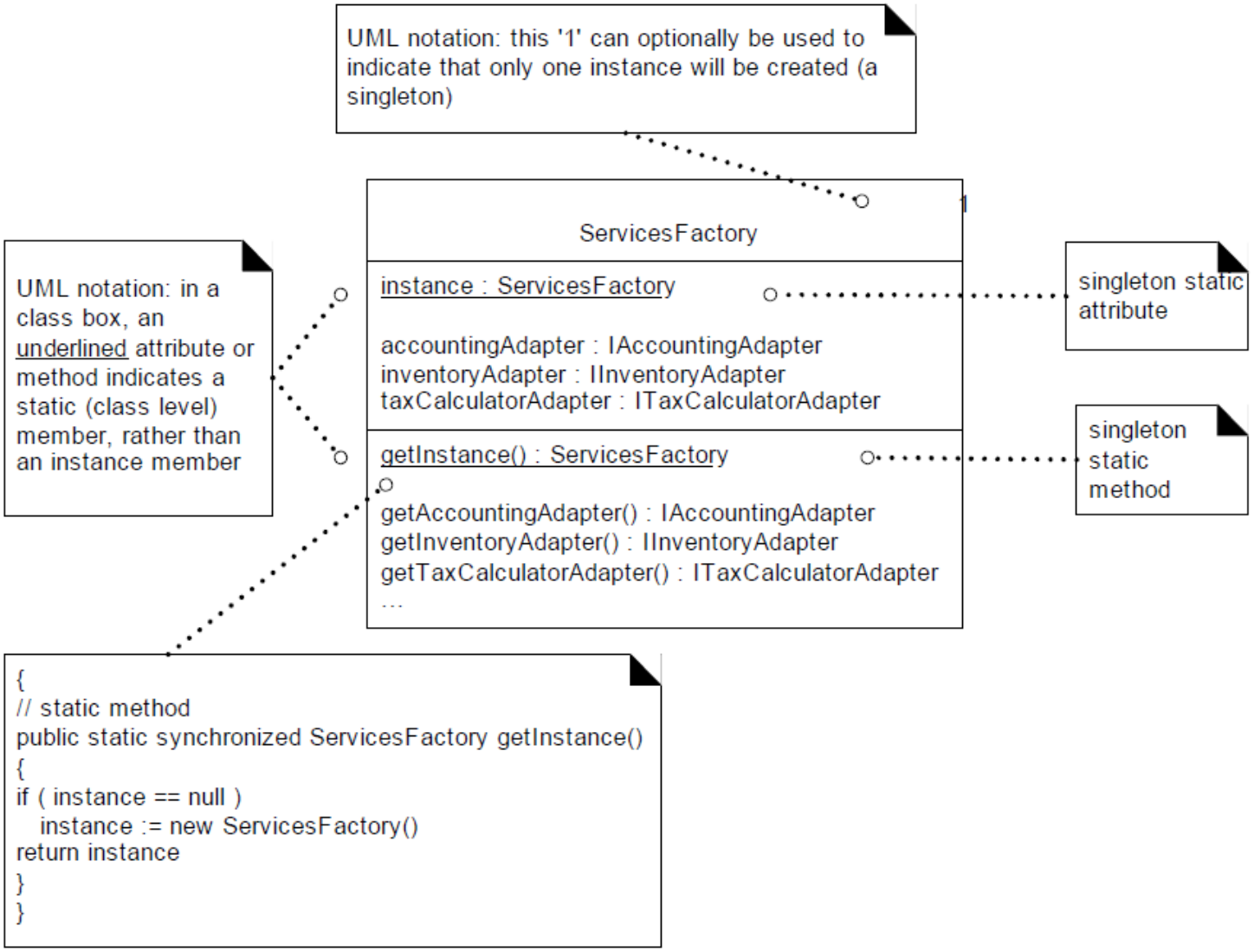
GoF Singleton

- Problem:
  - Who creates the factory itself, and how is it accessed?

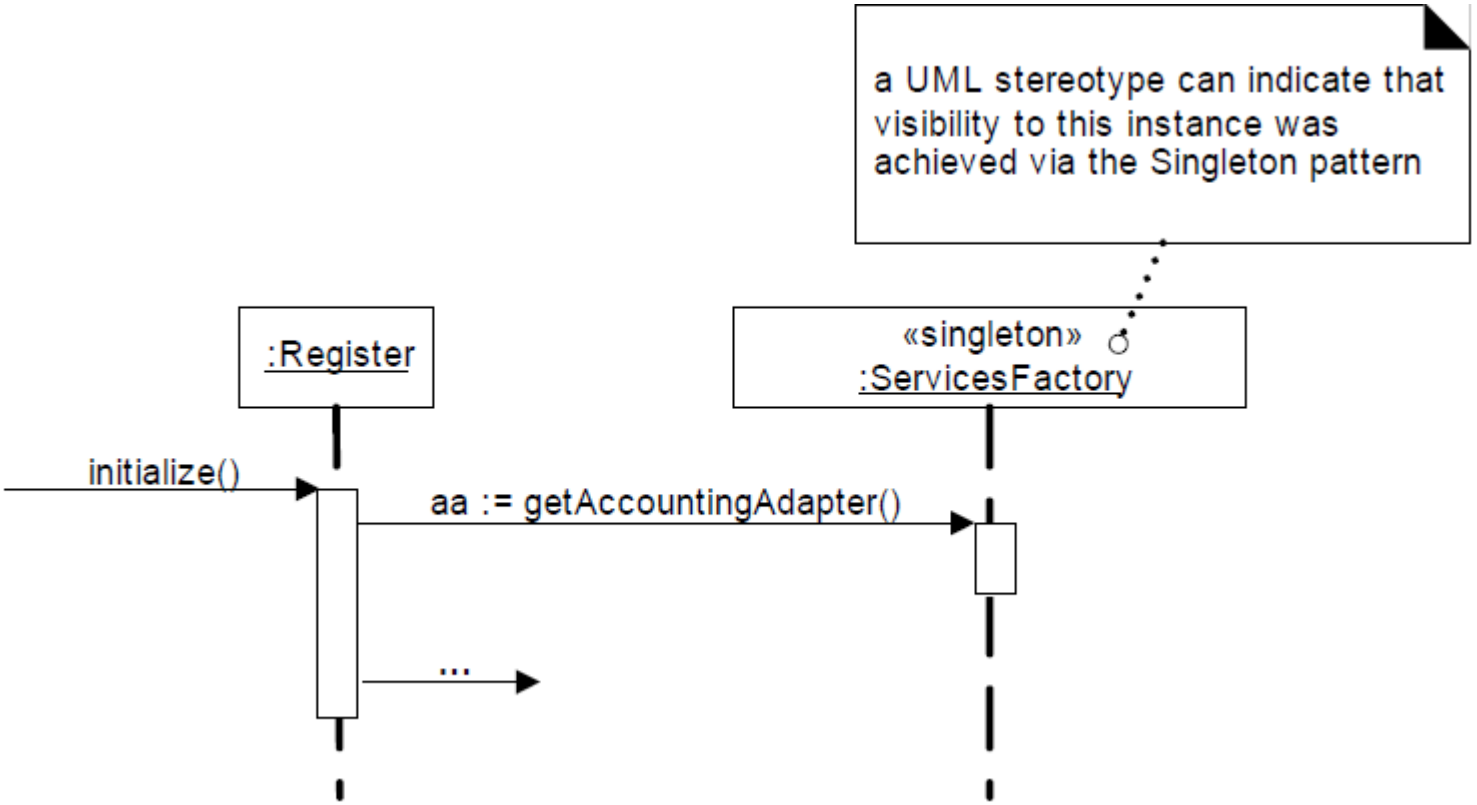
GoF Singleton

- Problem:
  - Who creates the factory itself, and how is it accessed?
- Solution
  - The key idea is that class X defines a **static method *getInstance*** that itself provides a single instance of X.

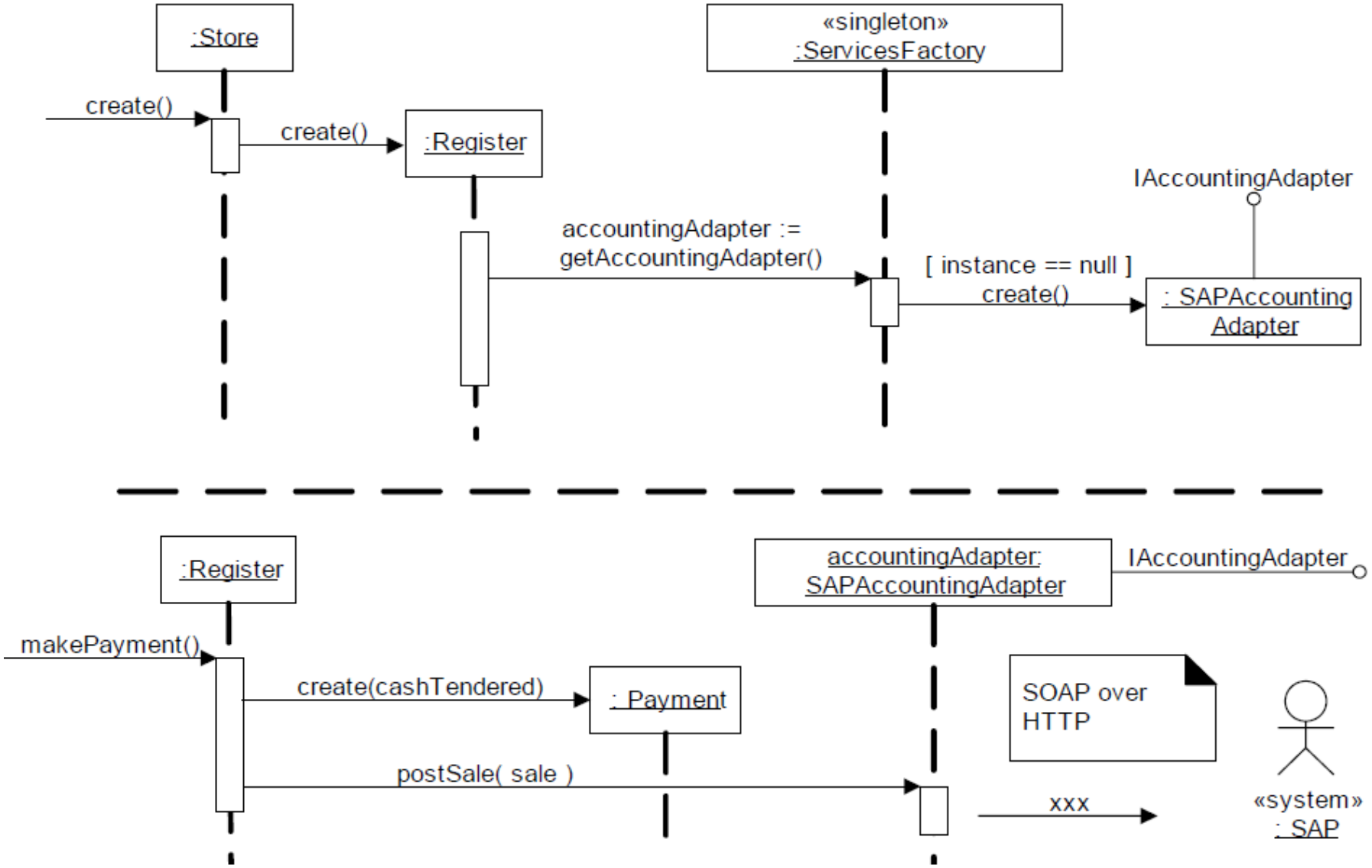
Singleton Pattern



Implicit *getInstance* Singleton Pattern



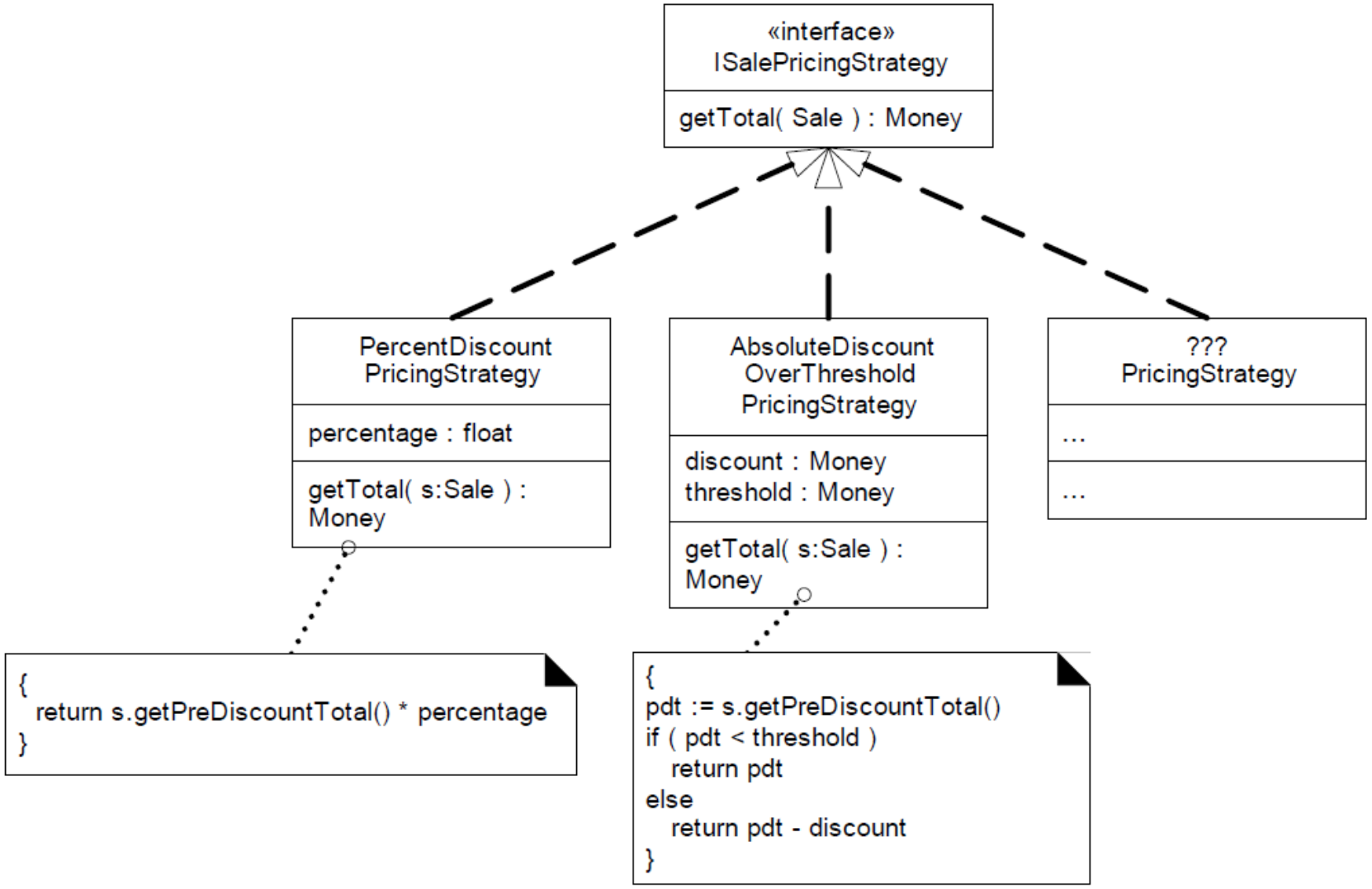
Adapter and Factory



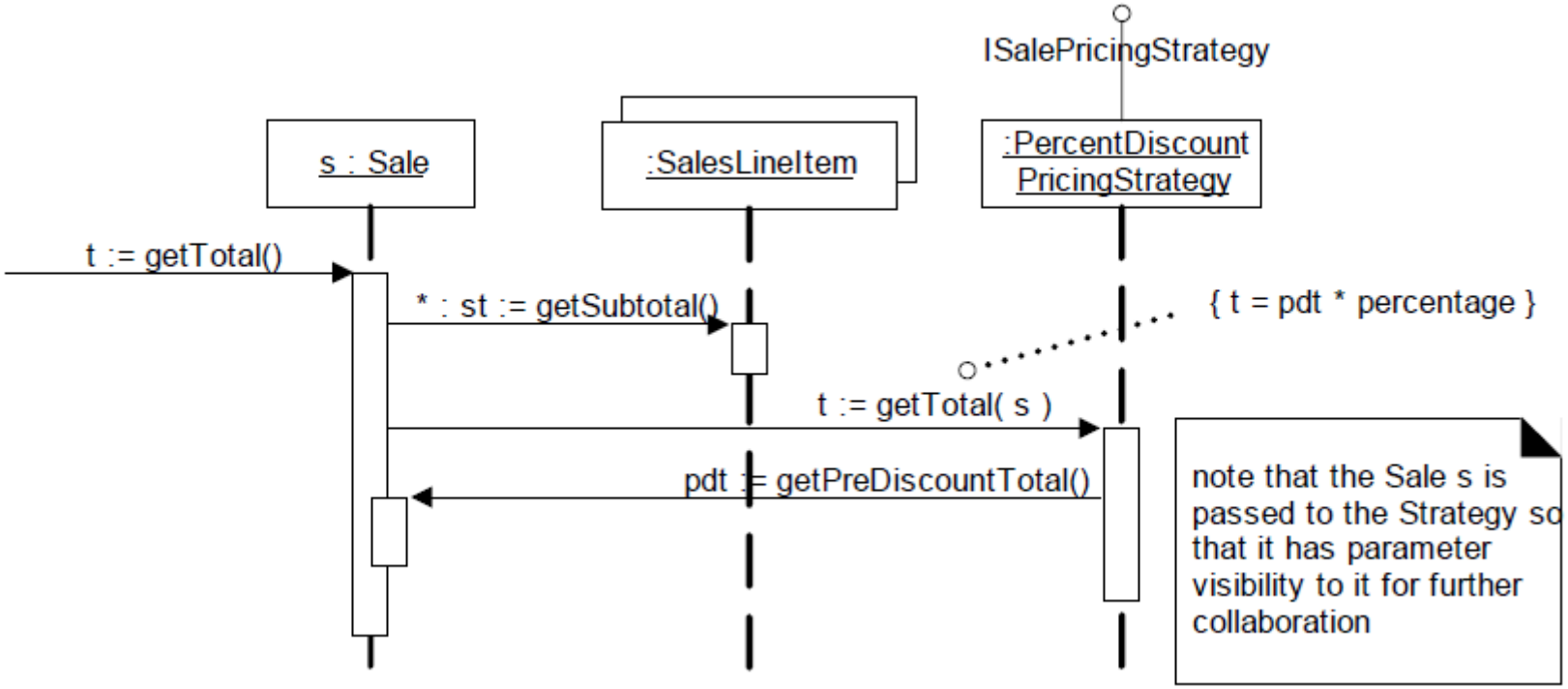
GoF Strategy

- **Problem:**
  - How to design for varying, but related, algorithms or policies?
  - such as a store-wide discount for the day, senior citizen discounts, and so forth.
- **Solution:**
  - Define each algorithm/policy/strategy in a separate class, with a common interface.

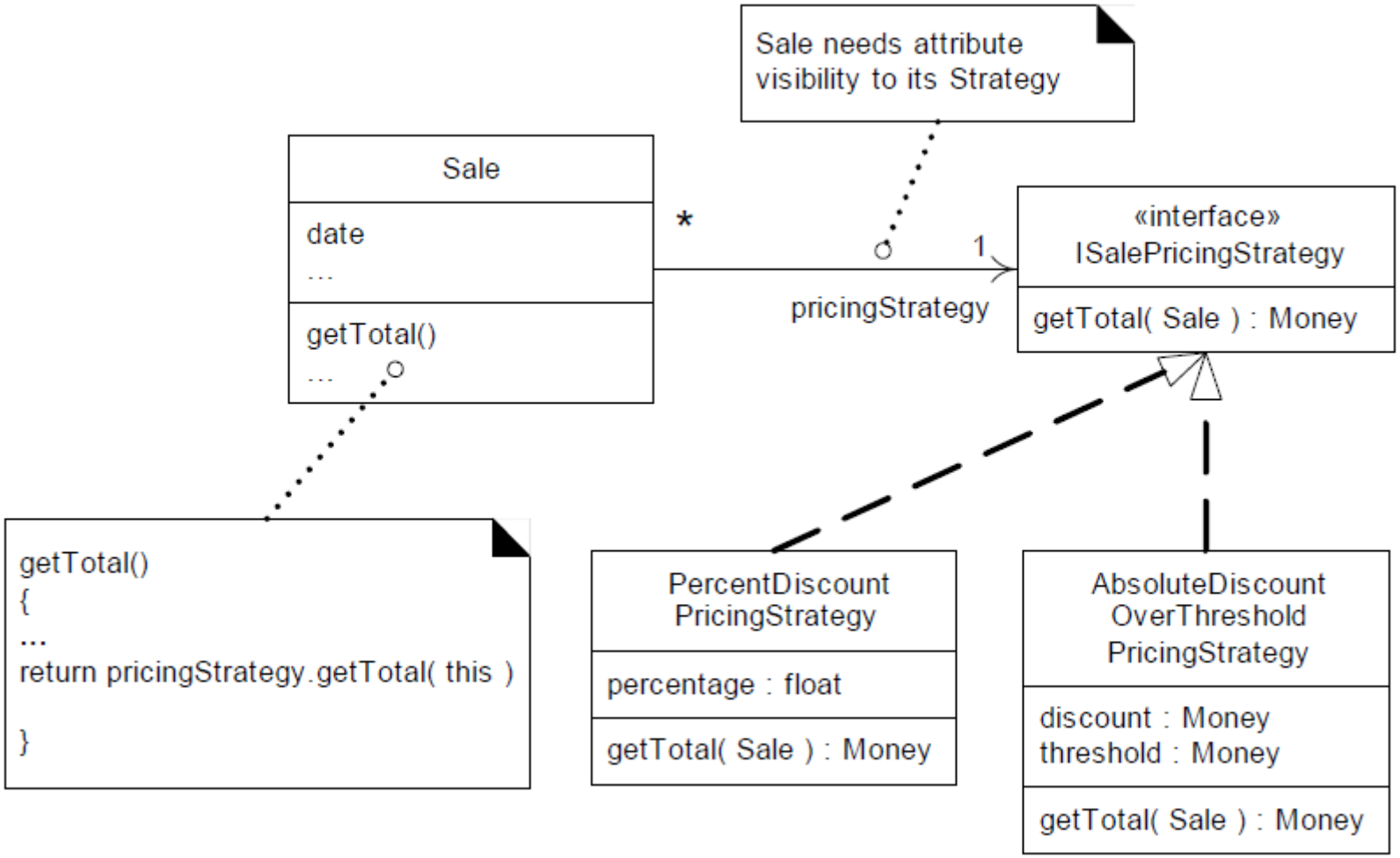
Pricing Strategy Classes



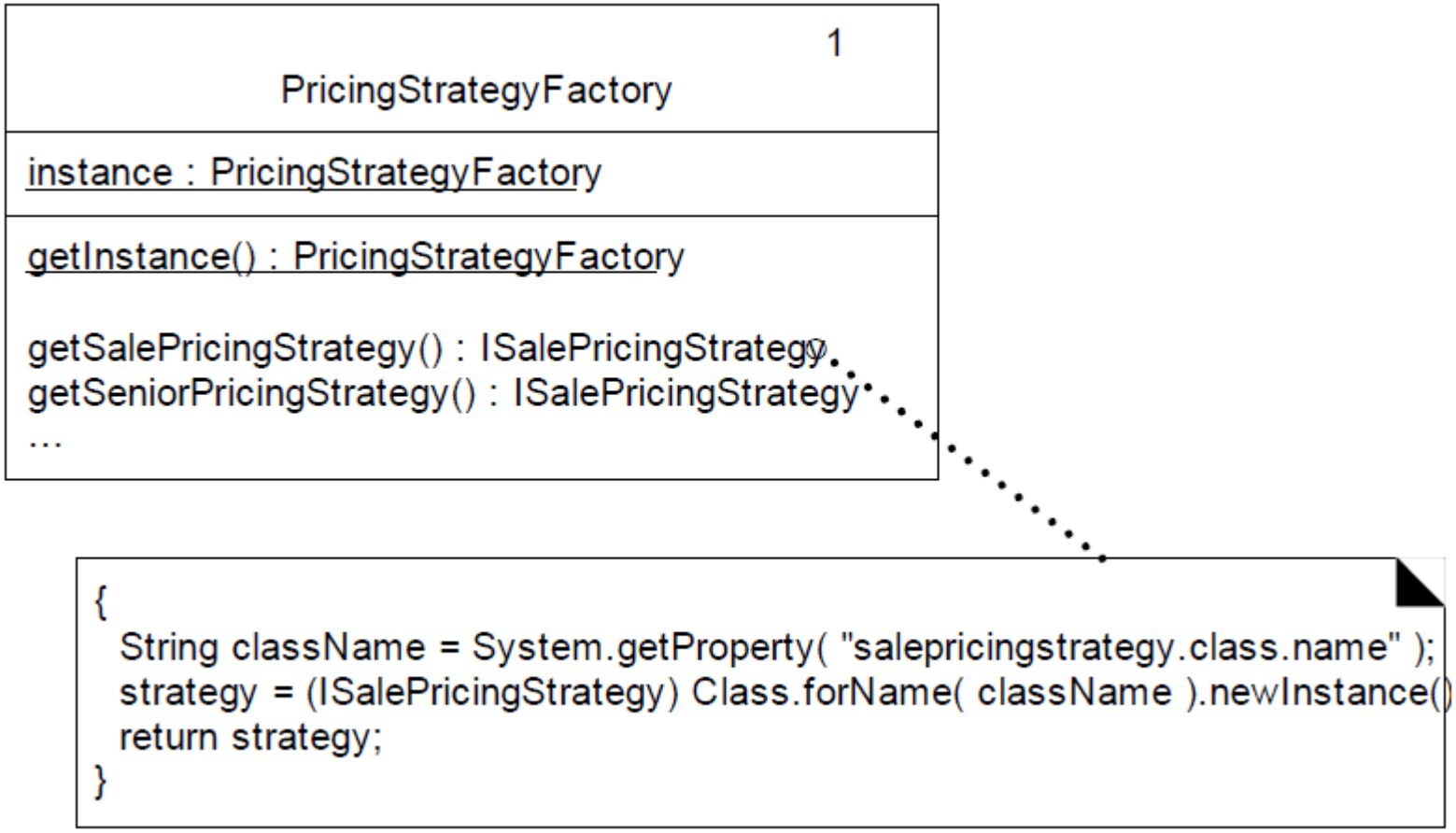
Strategy in Collaboration



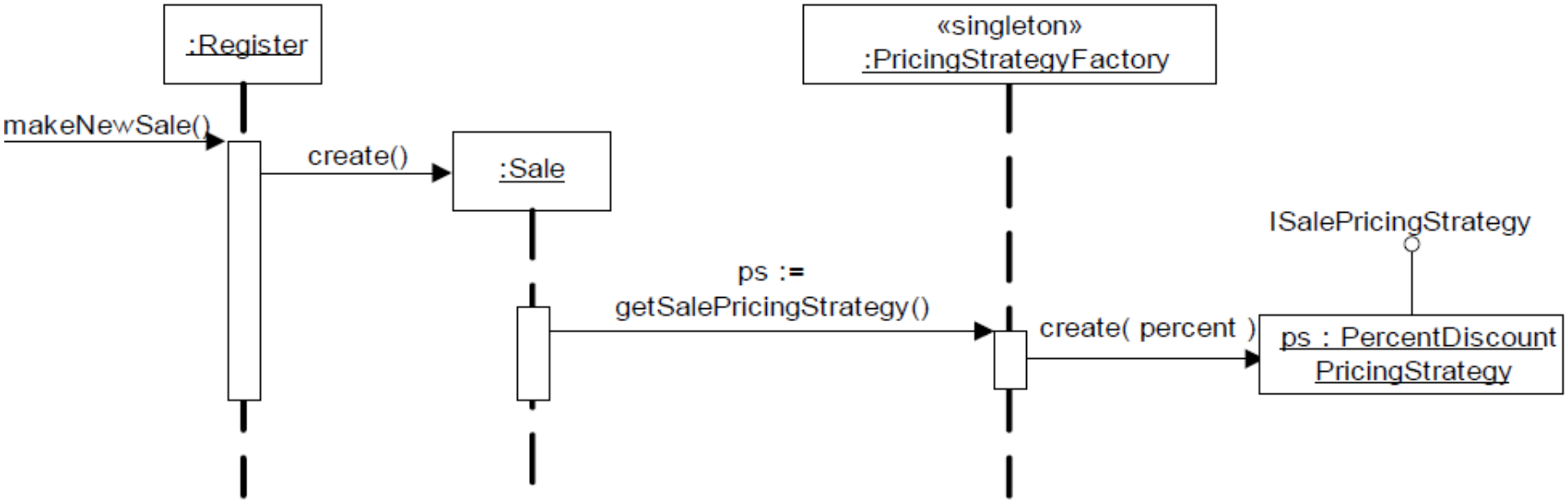
Attribute Visibility



Factory for Strategies

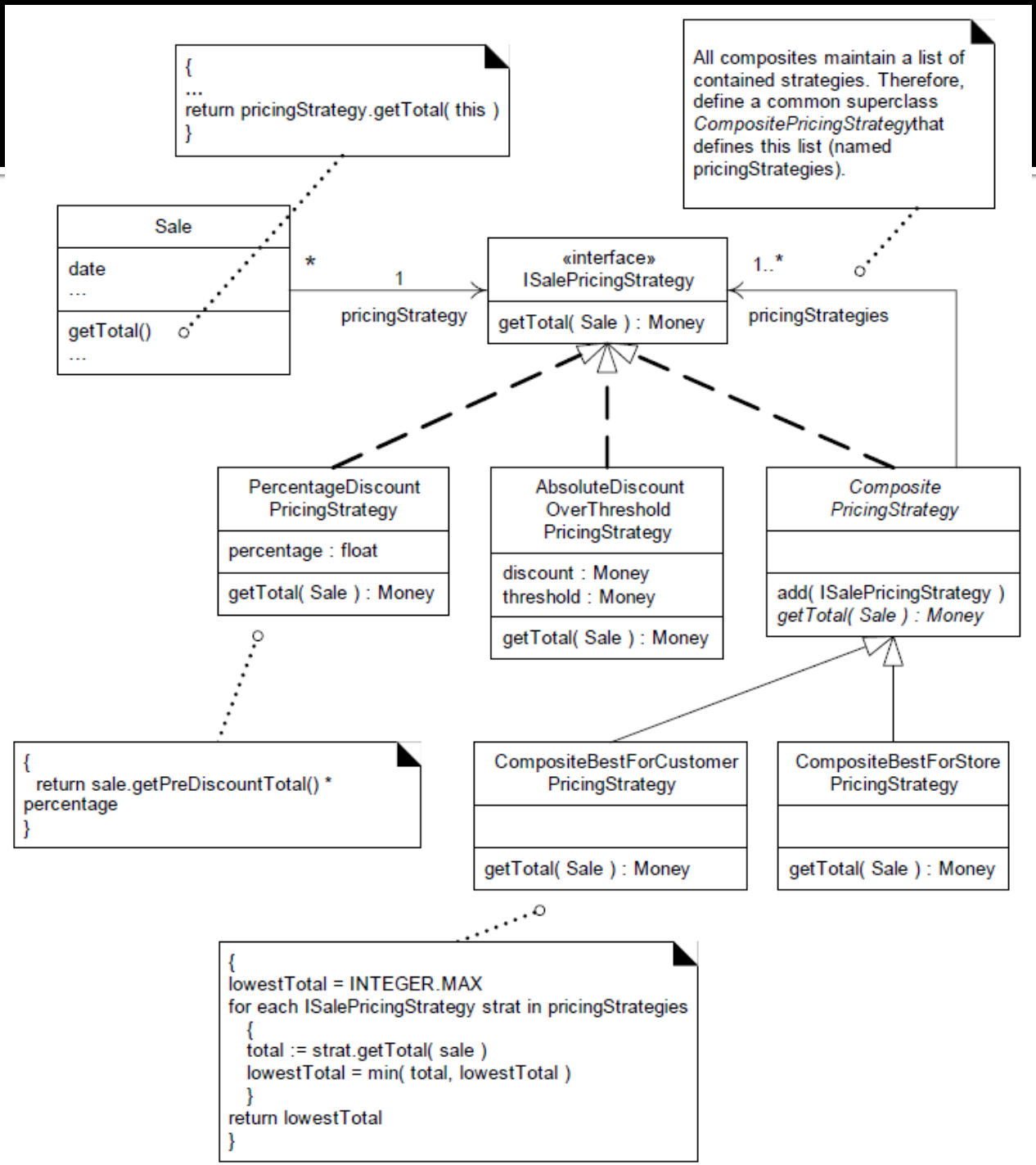


Creating a Strategy

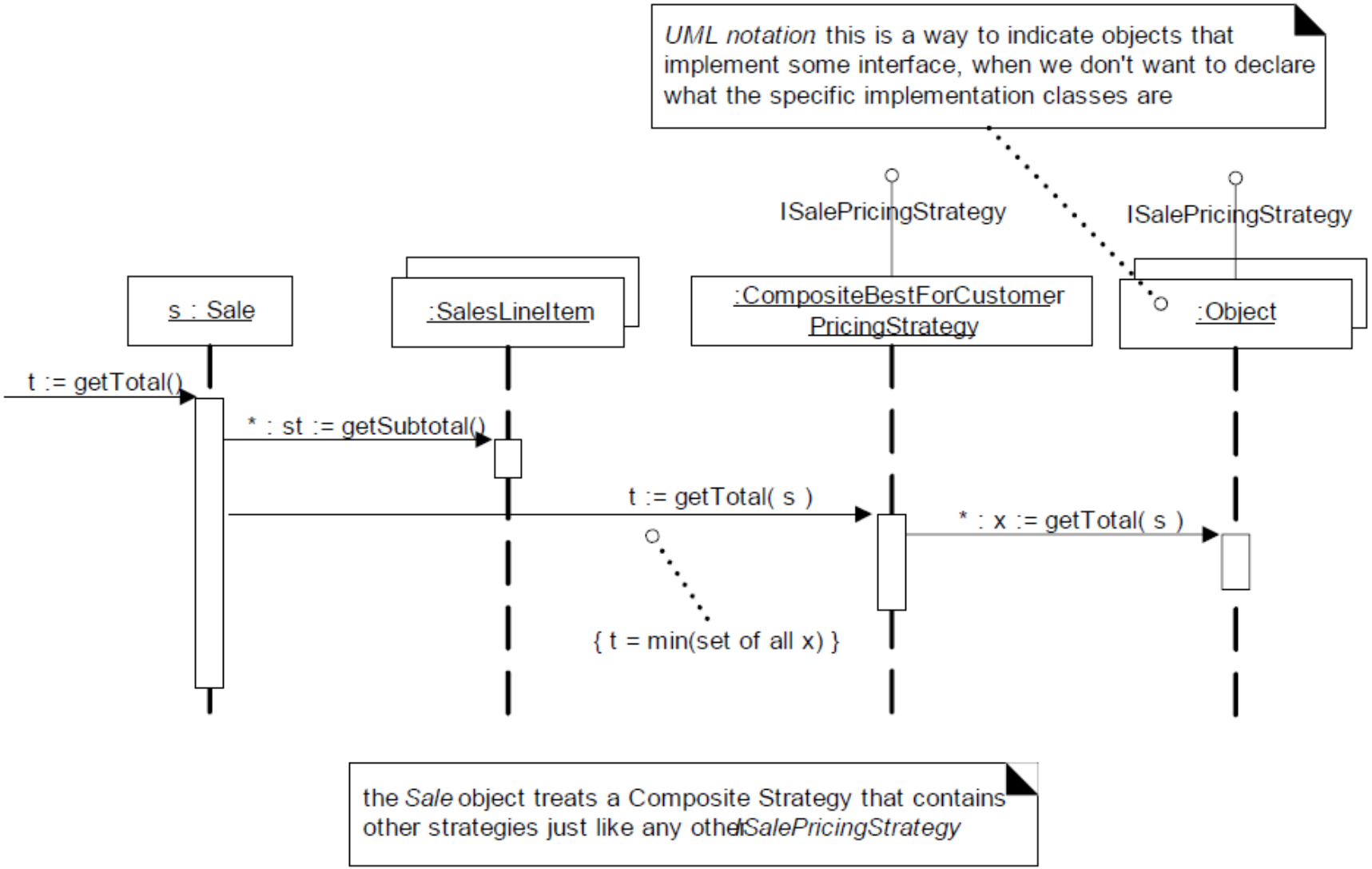


Composite GoF

- **Problem:**
  - How do we handle the case of multiple, conflicting policies?
  - e.g., conflicting pricing policies, 20% senior discount policy, on Monday, there is $50 off purchases over $500 etc.
- **Solution:**
  - Define classes for objects so that they implement the same interface.


{
...
return pricingStrategy.getTotal(this)
}
All composites maintain a list of contained strategies. Therefore, define a common superclass CompositePricingStrategythat defines this list (named pricingStrategies).
Sale
date
...
getTotal()
...
*
1
pricingStrategy
«interface»
ISalePricingStrategy
getTotal(Sale) : Money
1..*
pricingStrategies
PercentageDiscount PricingStrategy
percentage : float
getTotal(Sale) : Money
AbsoluteDiscount OverThreshold PricingStrategy
discount : Money
threshold : Money
getTotal(Sale) : Money
Composite PricingStrategy
add(ISalePricingStrategy)
getTotal(Sale) : Money
{
return sale.getPreDiscountTotal() * percentage
}
CompositeBestForCustomer PricingStrategy
getTotal(Sale) : Money
CompositeBestForStore PricingStrategy
getTotal(Sale) : Money
{
lowestTotal = INTEGER.MAX
for each ISalePricingStrategy strat in pricingStrategies
{
total := strat.getTotal(sale)
lowestTotal = min(total, lowestTotal)
}
return lowestTotal
}

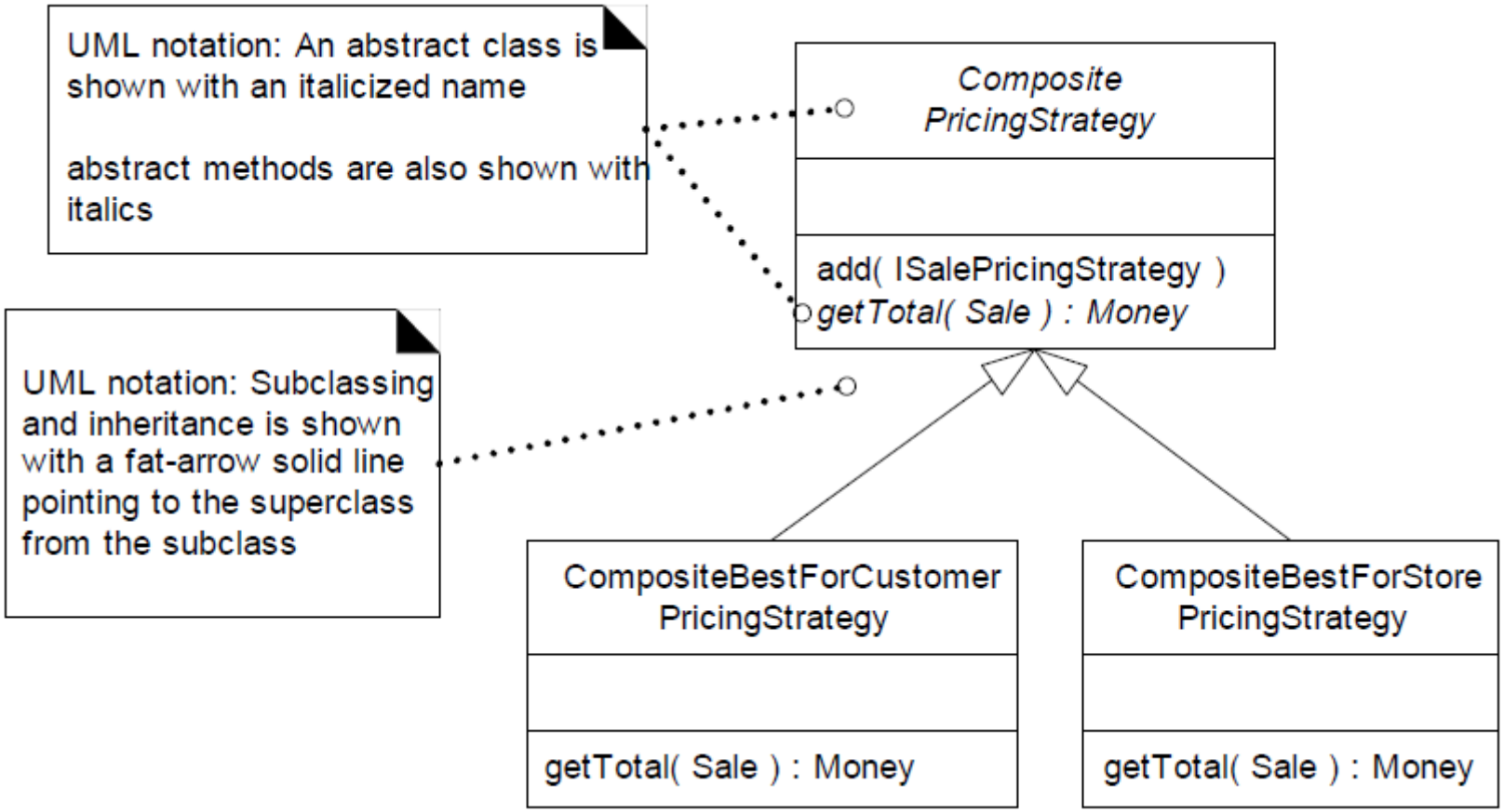
Collaborating with Composite



Mapping to Code

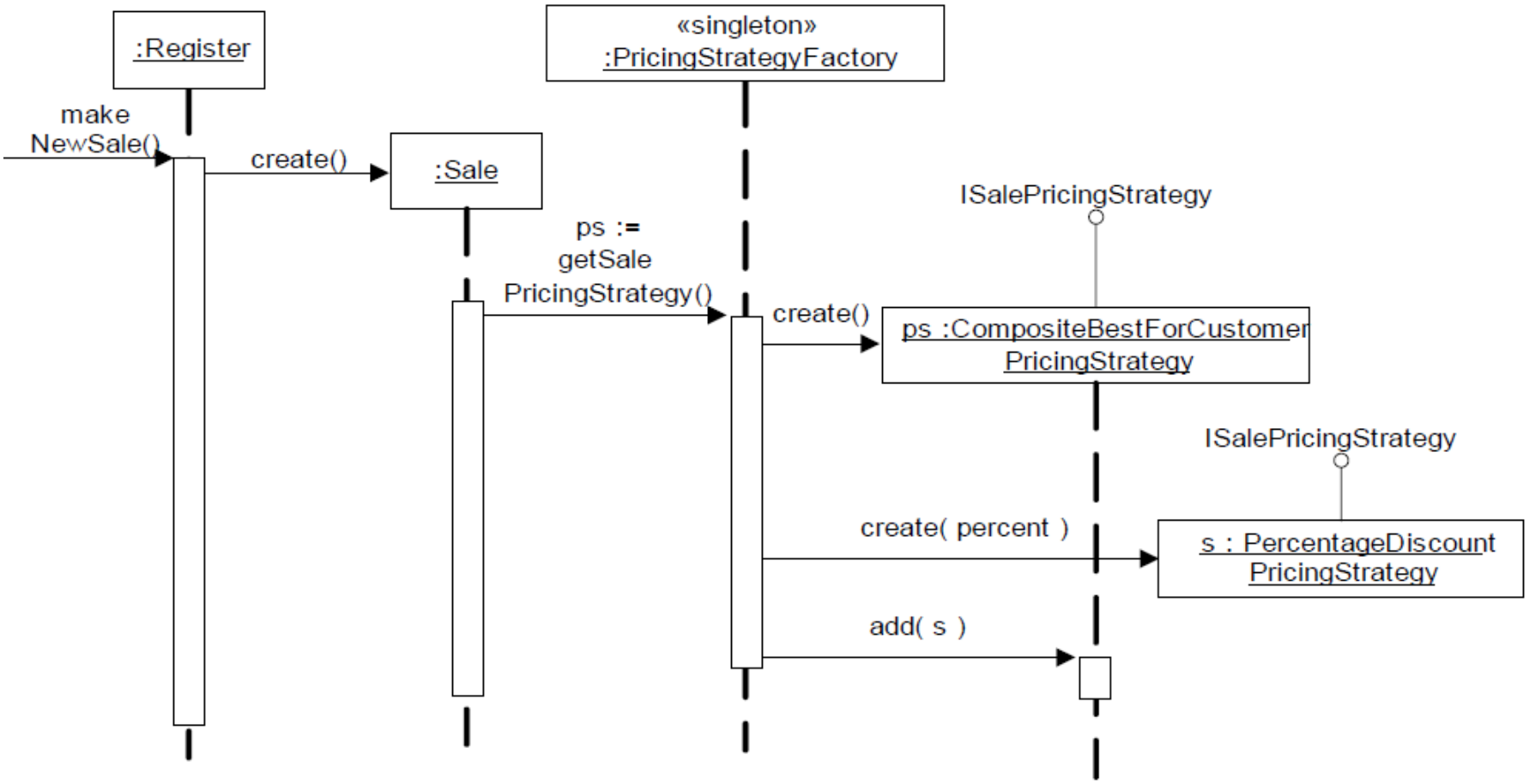
```
// a Composite Strategy that returns the lowest total
// of its inner SalePricingStrategies

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
```

Abstract Superclass and Inheritance

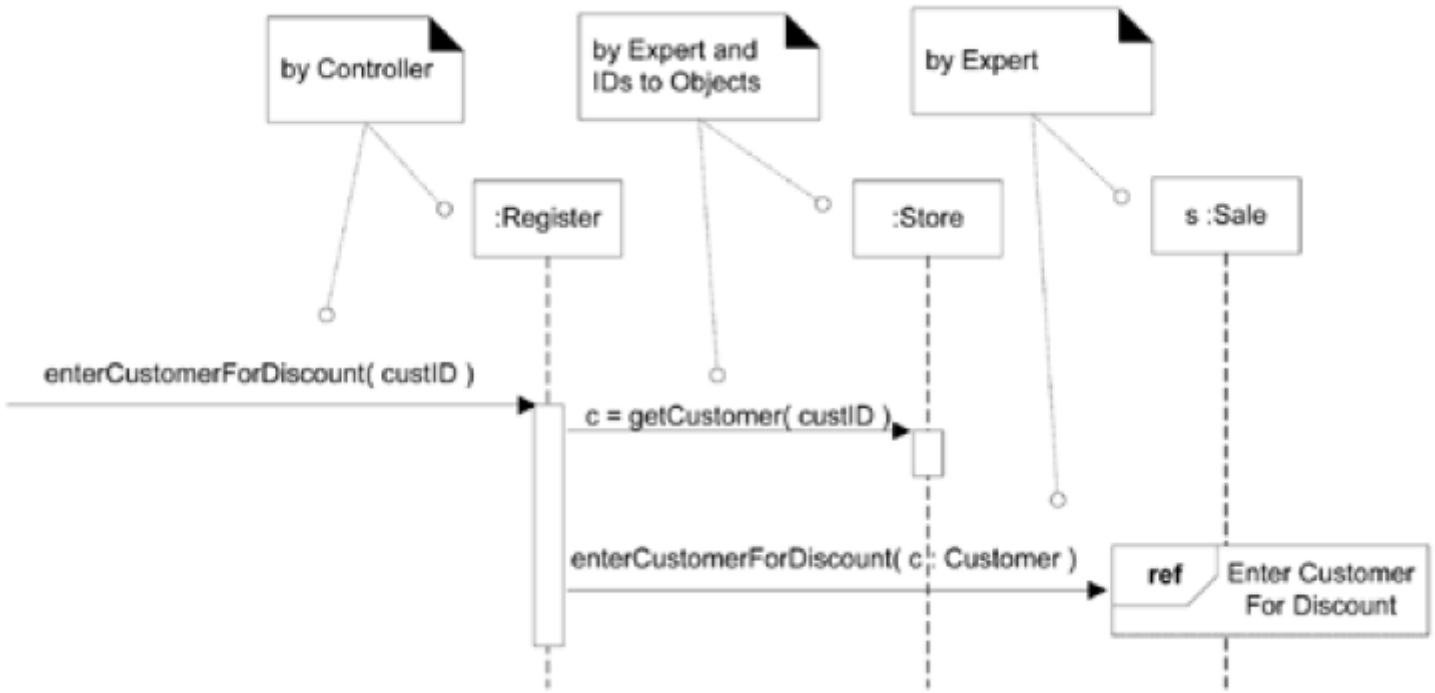


Creating Composite Strategy

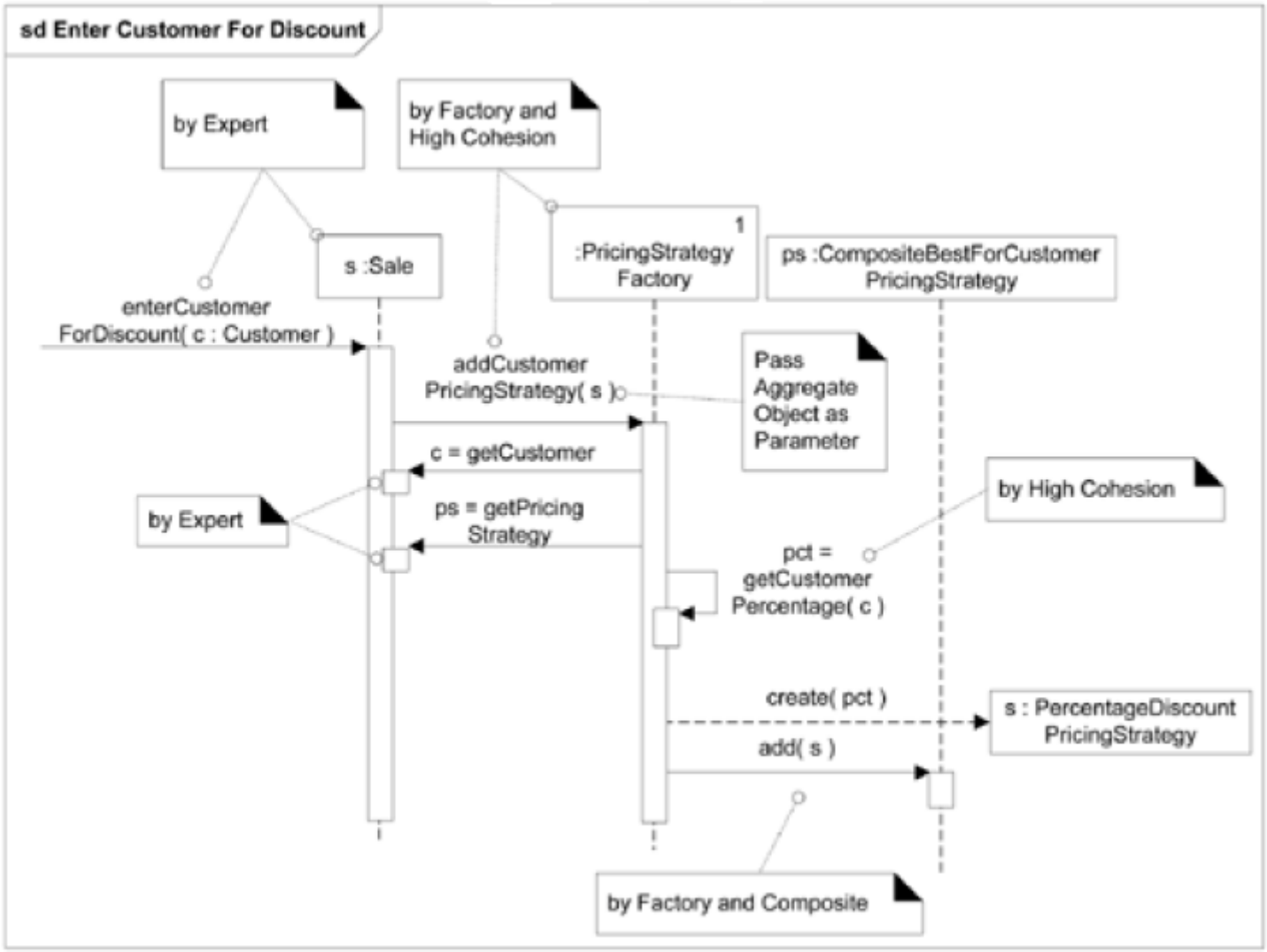
The composite object that contains the group also implements the ISalePricingStrategy interface.



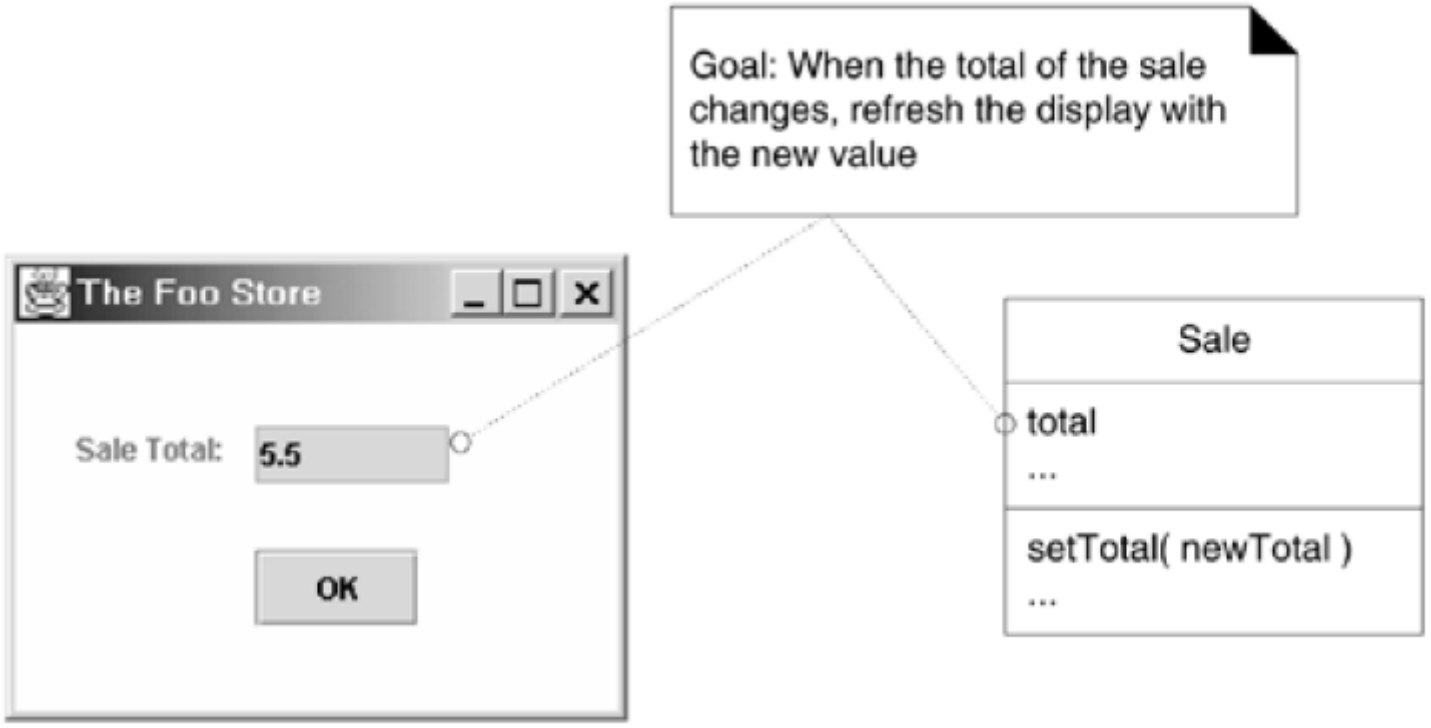
Strategy for Customer Discount (1)



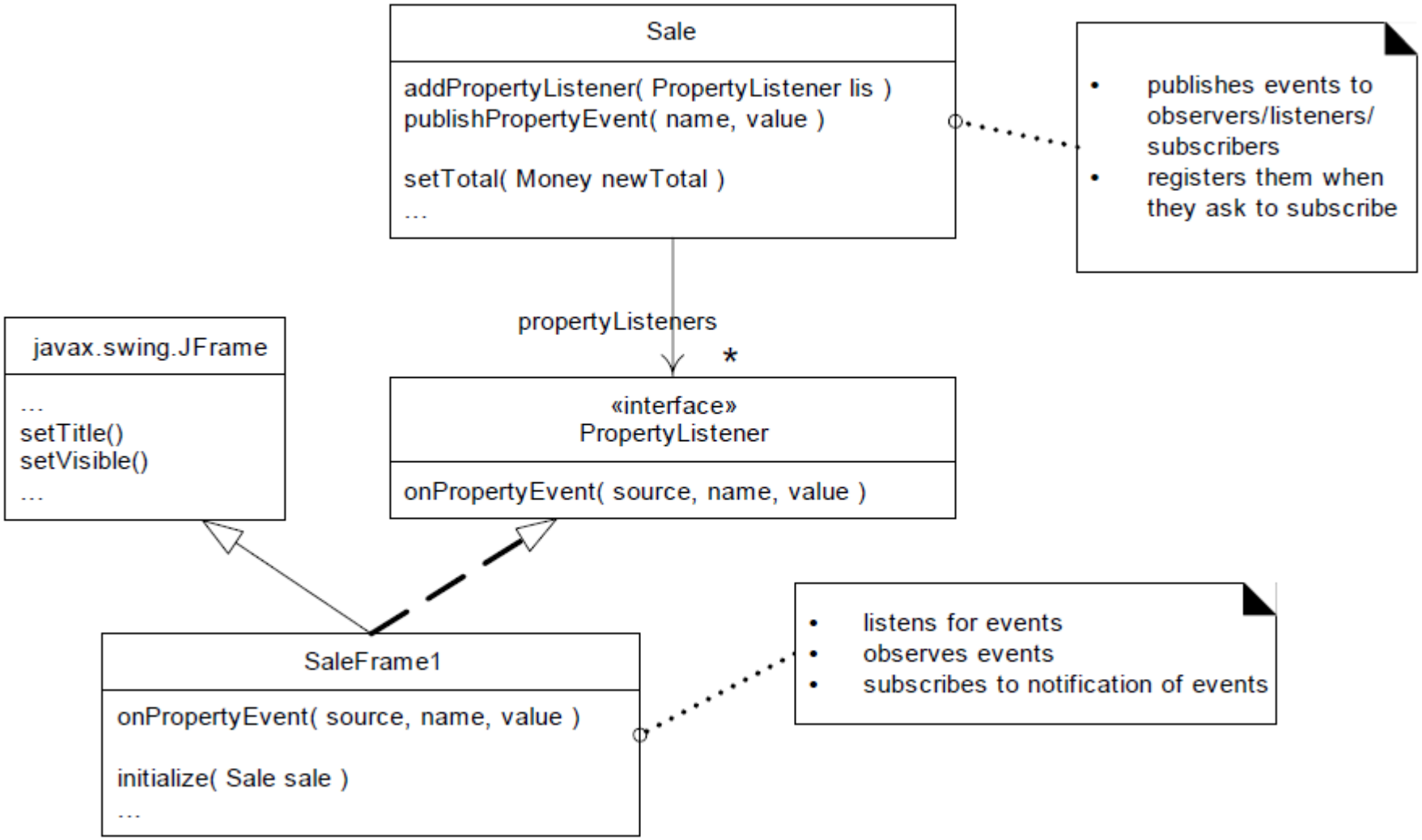
Strategy for Customer Discount (2)



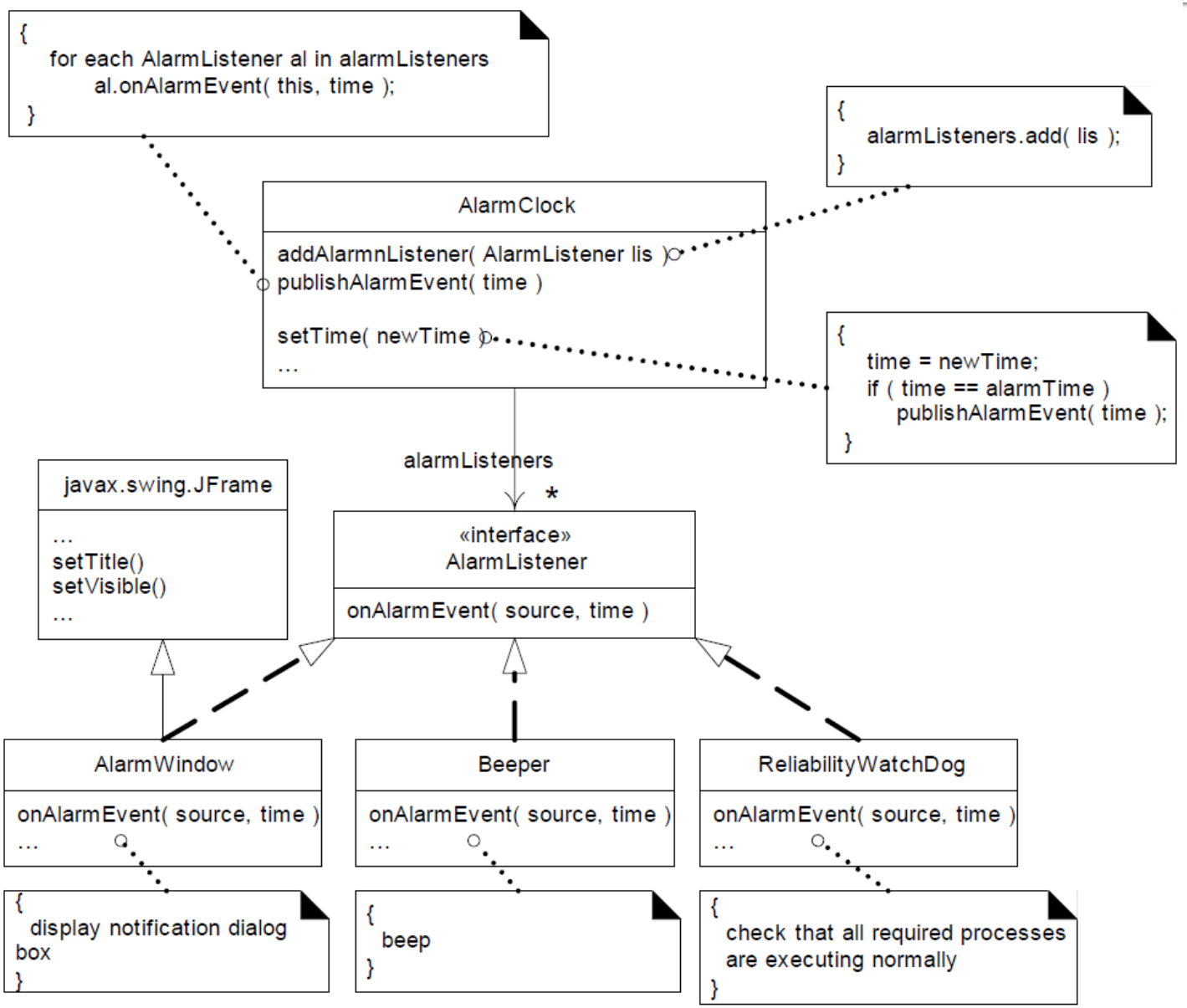
Updating Interface When Sale Total Changes



Who is Observer, Listener?



Observer Applied to Alarm Event with Different Subscribers



Quiz

- What are the final four GRASP patterns?
- Describe Indirection and Polymorphism patterns.
- What are the GoF patterns?
- Describe Singleton Pattern

Actions

- Review Slides.
- Read Chapter 22 and 23, GRASP: More Patterns for Assigning Responsibilites
  - *Applying UML and Patterns*, Craig Larman