SWEN30006

Software Modelling and Design

Applying GoF Design Patterns (Part 2: Strategy & Composite)

Textbook: Larman Chapter 26

Lecturer: Peter Eze

Anything you can do, I can do meta.

—Daniel Dennett

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 (Brief)
  - Decorator
- Recognise GRASP principles as a generalization of other design patterns.

Problem 2.1: Complex Pricing Logic

- POS provides 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

Strategy (GoF)

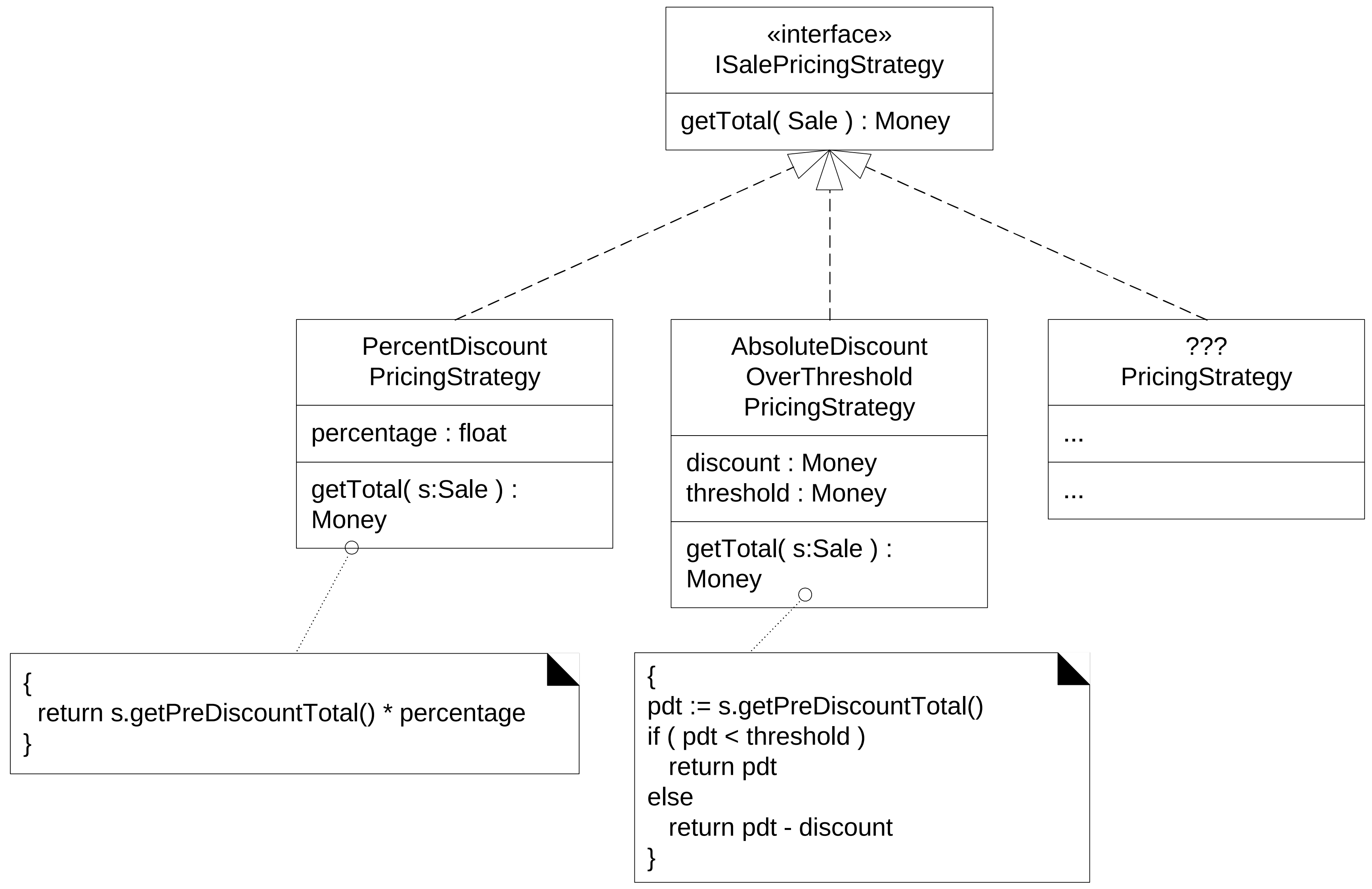
Problem:

- 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.

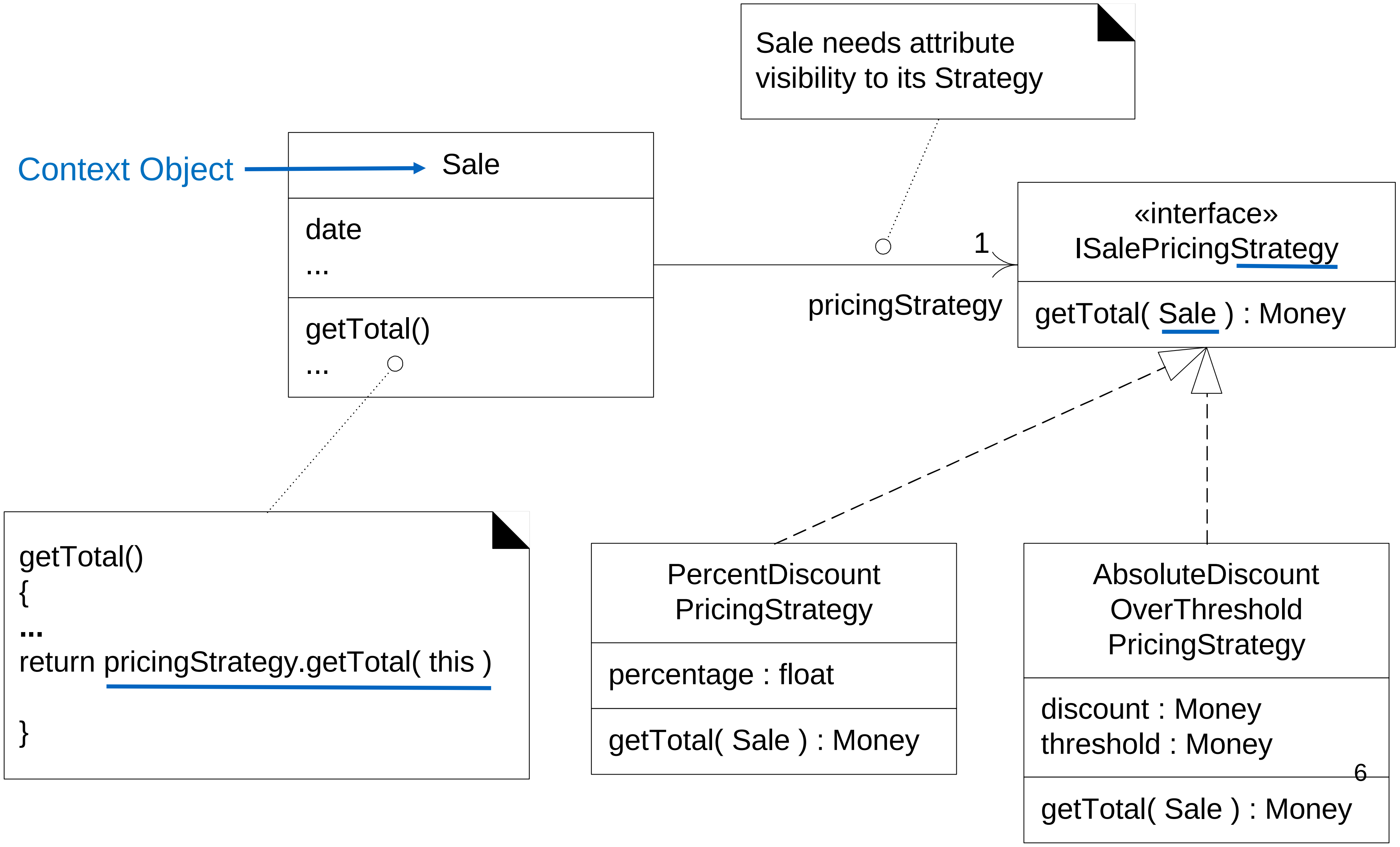
Pricing Strategy Classes



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

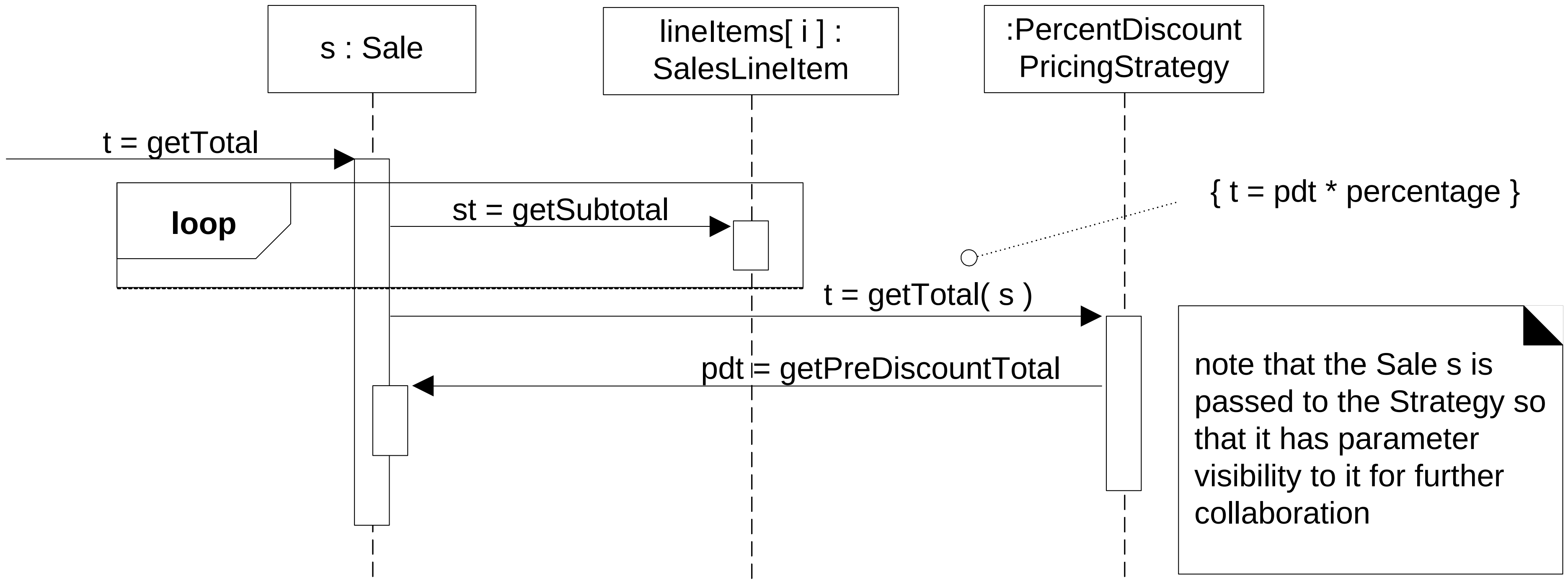
```
{
pdt := s.getPreDiscountTotal()
if ( pdt < threshold )
   return pdt
else
   return pdt - discount
}
```

Context Object Visibility to Strategy



```
getTotal()
{
...
return pricingStrategy.getTotal( this )

}
```

Strategy in Collaboration


s : Sale
lineItems[i] : SalesLineItem
:PercentDiscount PricingStrategy
t = getTotal
loop
st = getSubtotal
{ t = pdt * percentage }
t = getTotal(s)
pdt = getPreDiscountTotal
note that the Sale s is passed to the Strategy so that it has parameter visibility to it for further collaboration

Example

```java
public interface ISalePricingStrategy {
      public double getTotal(Sale s);
}
```

ISalePricingStrategy.java

```java
public class PercentDiscountPricingStrategy implements 
ISalePricingStrategy{
      private double percentage = 0.05;
      public double getTotal(Sale s) {
            return s.getPreDiscountTotal() - 
      (s.getPreDiscountTotal() * percentage);
}}
```

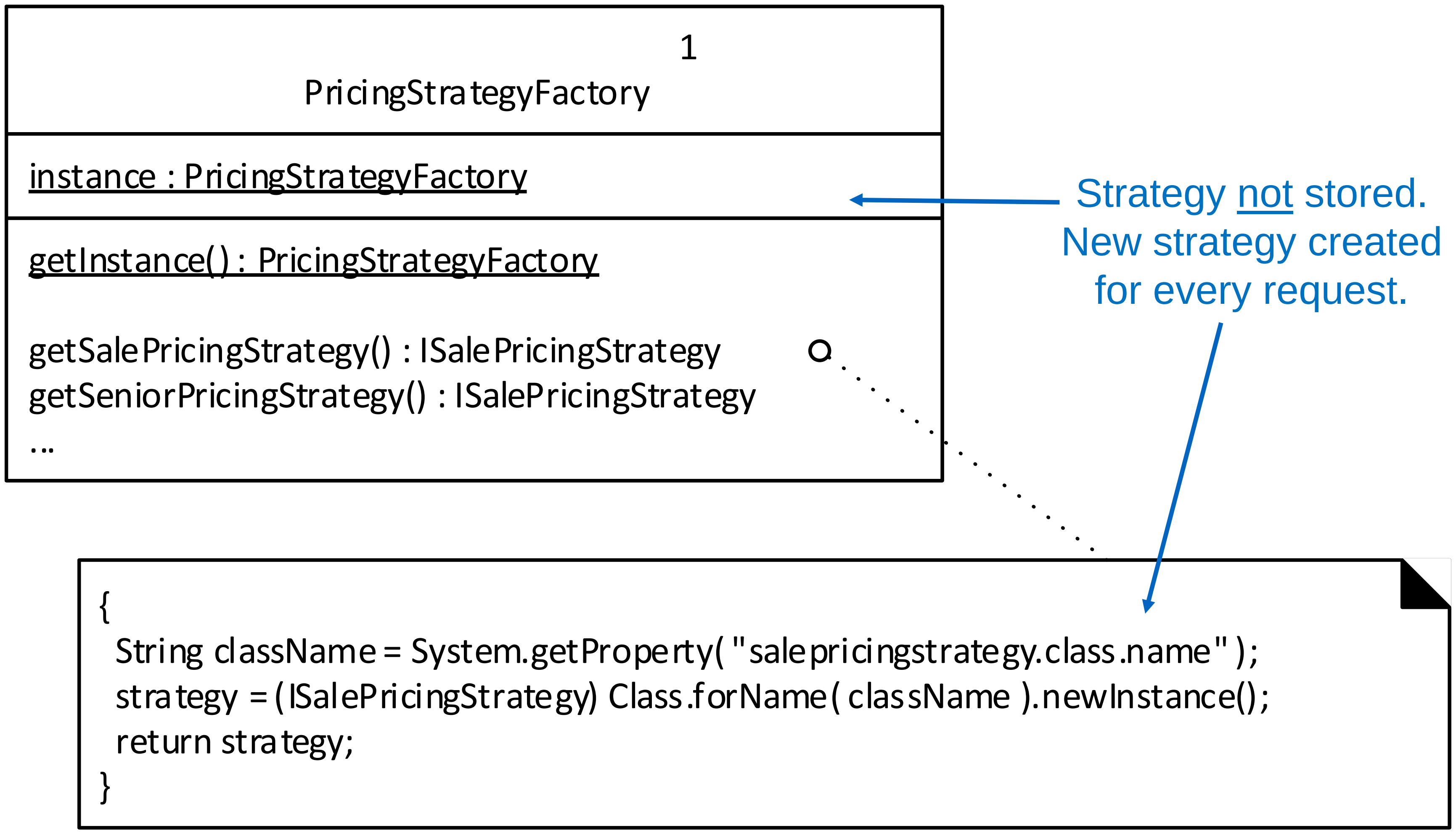
PercentDiscountPricingStrategy.java

Example (cont)

```
public class AbsoluteDiscountOverThresholdPricingStrategy
implements ISalePricingStrategy {
	private double threshold = 50;
	private double discount = 5;
	public double getTotal(Sale s) {
		double pdt = s.getPreDiscountTotal();
		if(pdt >= threshold) {
			return pdt - discount;
		}else {
			return pdt;
		}
}}
```

AbsoluteDiscountOverThethresholdPricingStrategy.java

Creation: Factory for Strategies



1
PricingStrategyFactory

instance : PricingStrategyFactory

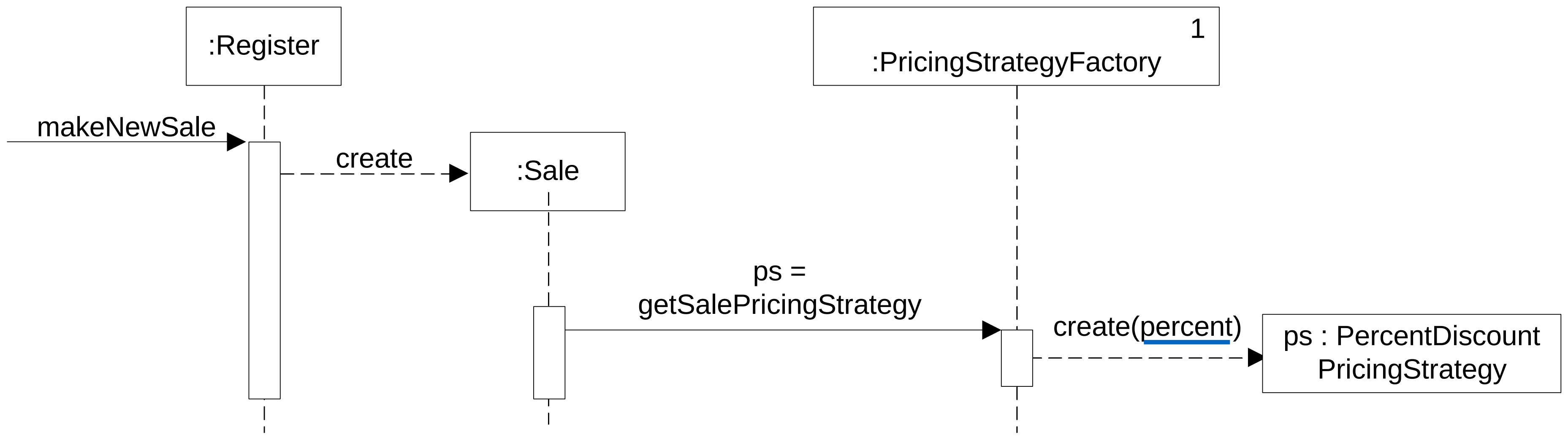
getInstance() : PricingStrategyFactory

getSalePricingStrategy() : ISalePricingStrategy
getSeniorPricingStrategy() : ISalePricingStrategy
...

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

Strategy not stored.
New strategy created
for every request.

Creating a Strategy



Which of the following statements is NOT true about Adapter and Strategy patterns:

Adapter is Structural design pattern while strategy is a behavioural design pattern

Adapter uses indirection while strategy does not.

No difference between strategy and adapter since both patterns are based on polymorphism and could be created using Singleton factory.

Both strategy and adapter patterns offer protected variation.

Problem 2.2: 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?
    - Some discounts **cannot be combined** with others
    - Possible policies: Best for customer or Best for store

Composite pricing strategy:

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

Composite (GoF)

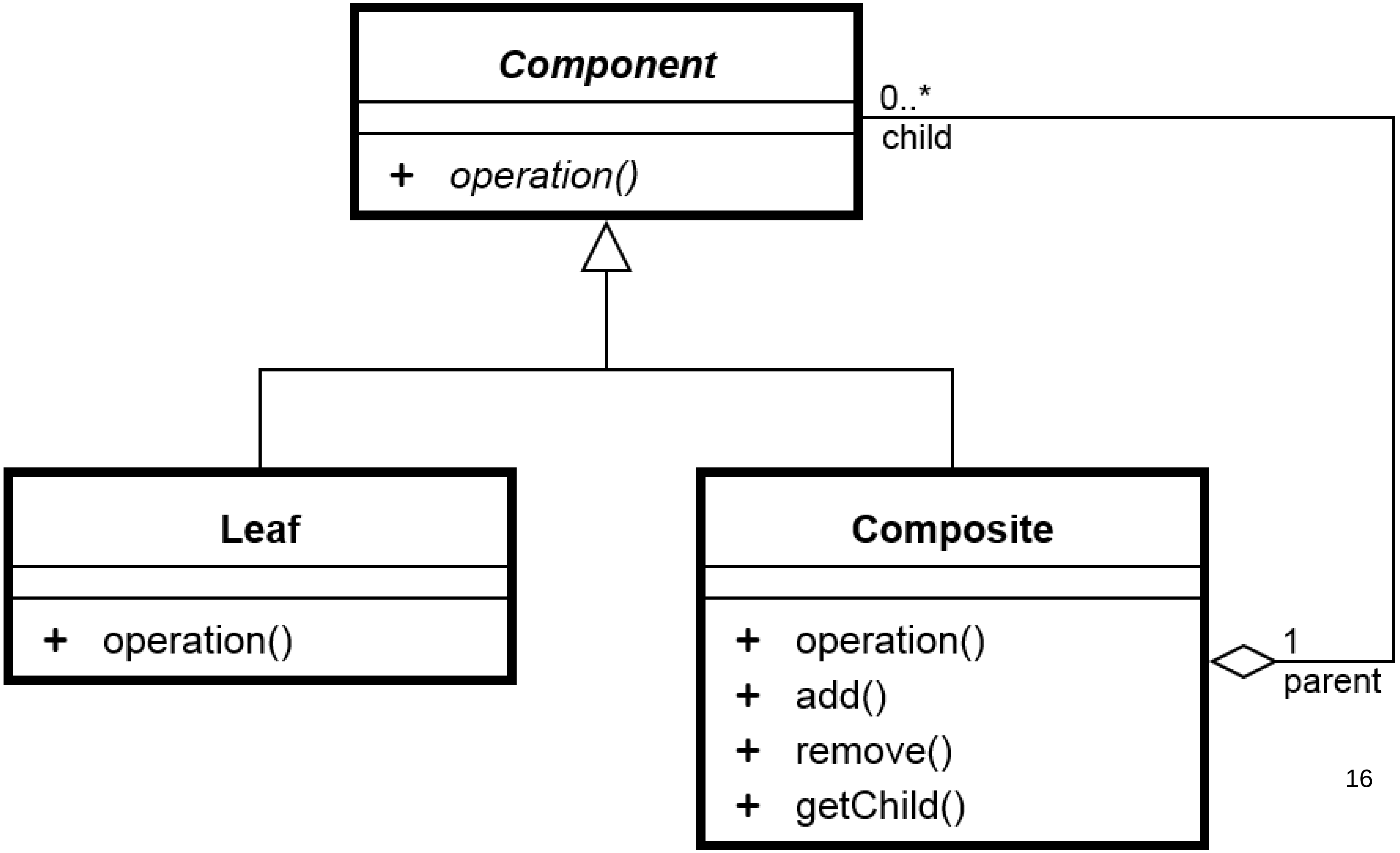
Problem:

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

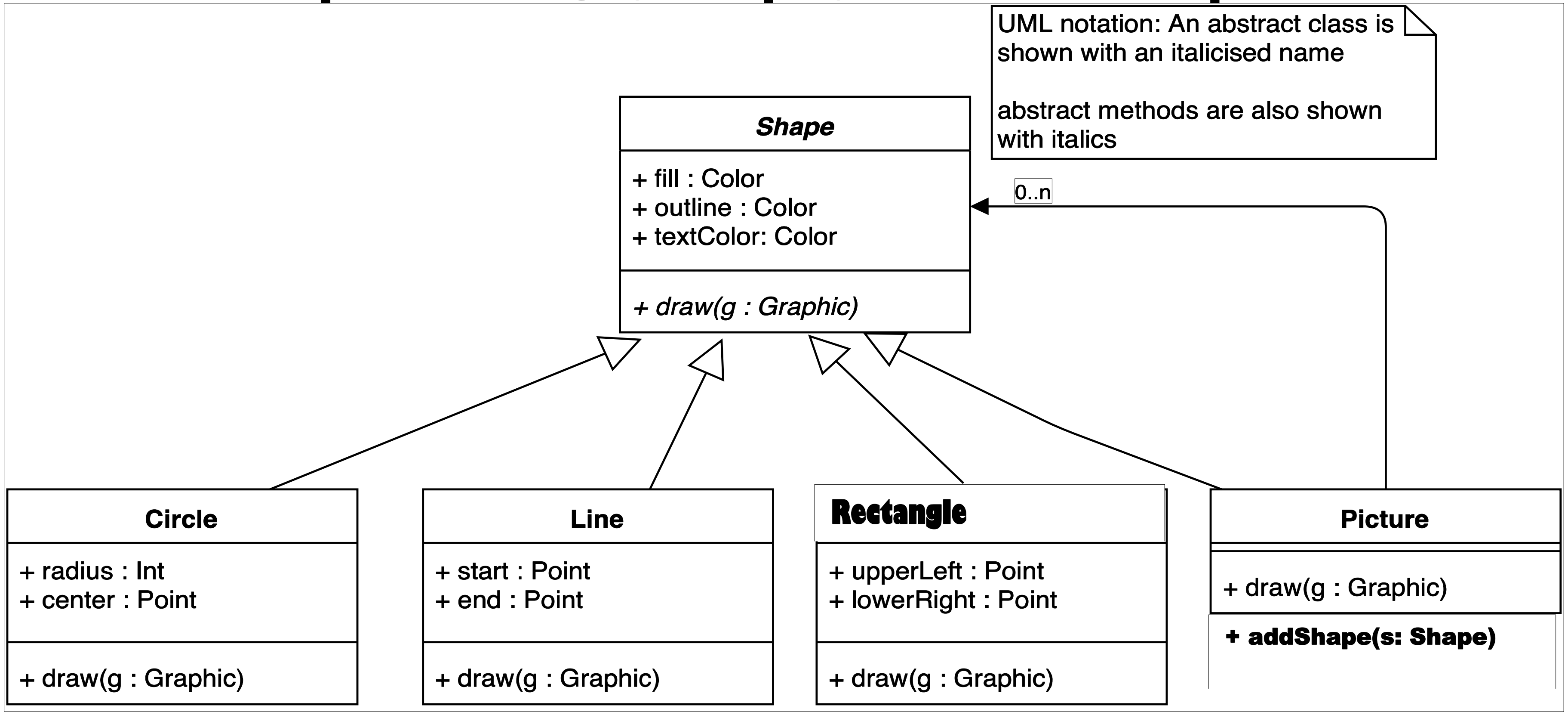
Solution (advice):

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

Composite: Generalised Structure



Example1: Composite shape



`Circle.draw(g)` `Line.draw(g)` `Rectangle.draw(g)` `Picture.draw(g)`

Example: Main

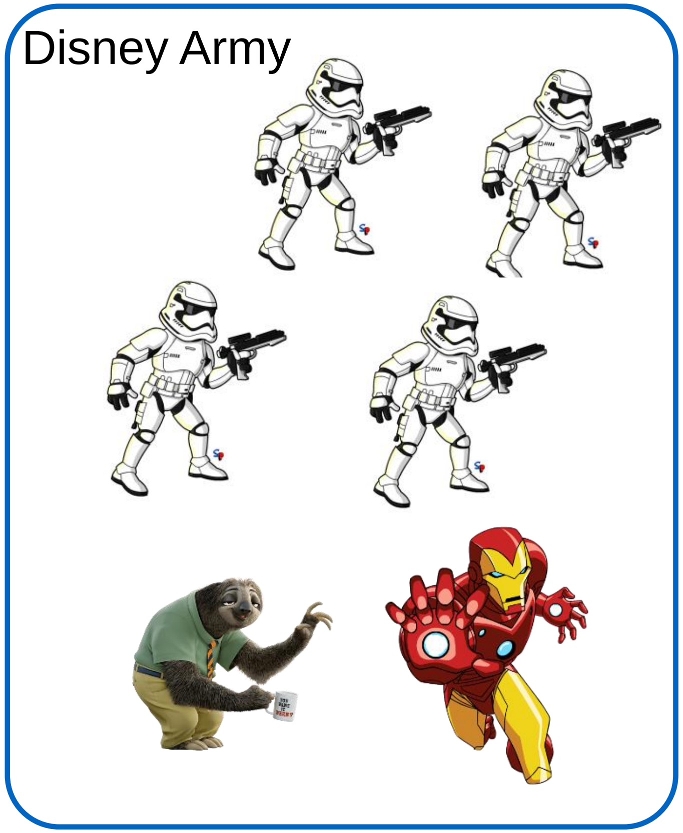
```
public class CompositeDrawing {
   public static void main(String[] args) {
   Circle circle1 = new Circle(5,new Point(0,0));
   Rectangle rectangle1 = new Rectangle(new Point(0,10),new 
Point(10,20));
   Line line1 = new Line(new Point(5,10),new Point(0,10));
   Picture myPicture = new Picture();

   myPicture.addShape(circle1);
   myPicture.addShape(rectangle1);
   myPicture.addShape(line1);
   myPicture.draw();
  }
}
```

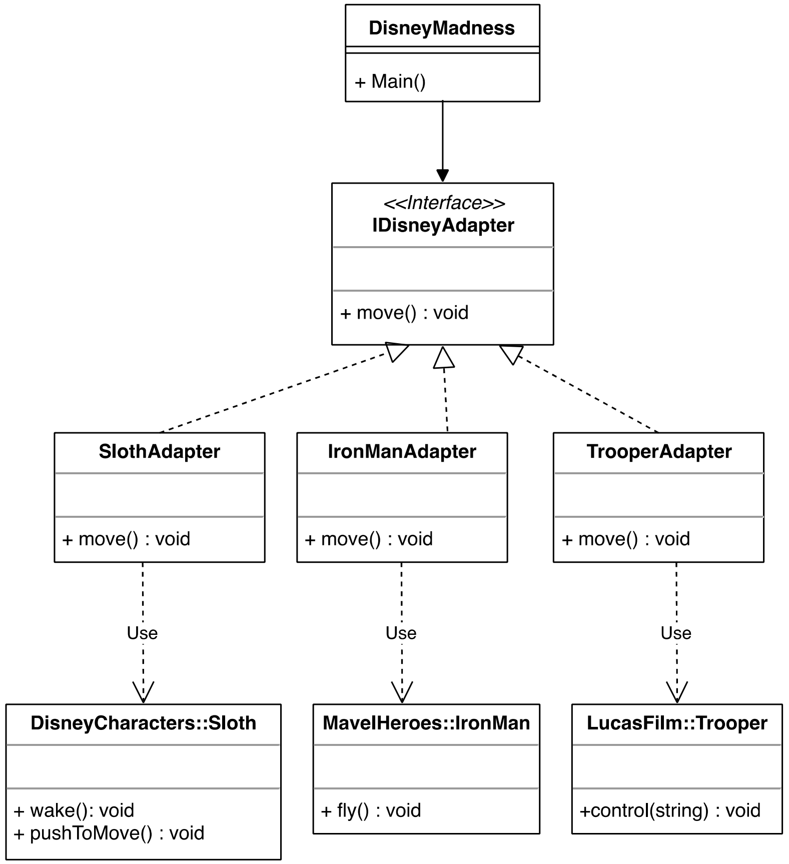
Example: Picture (Composite Shapes)

```
public class Picture extends Shape{
   ArrayList<Shape> shapes;
   public Picture() {
      shapes = new ArrayList<Shape>();
   }

   public void addShape(Shape shape) {
     shapes.add(shape);
   }

   public void draw() {
     for(Shape s: shapes) {
       s.draw();
    }
  }
}
```

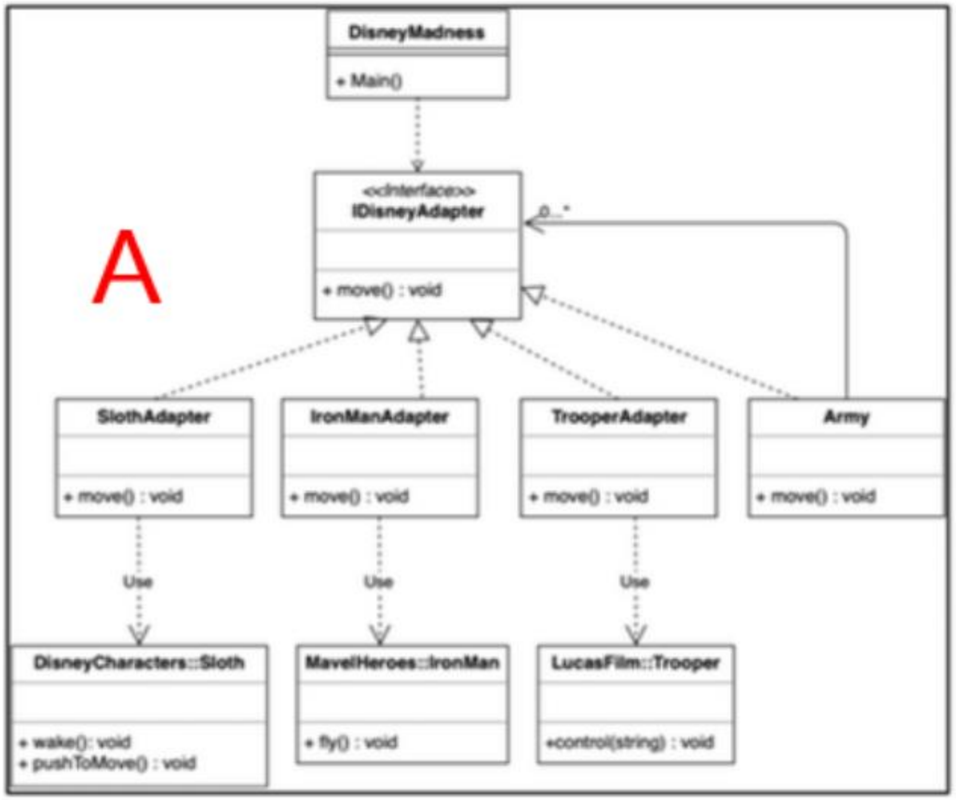
Example2: Composite characters

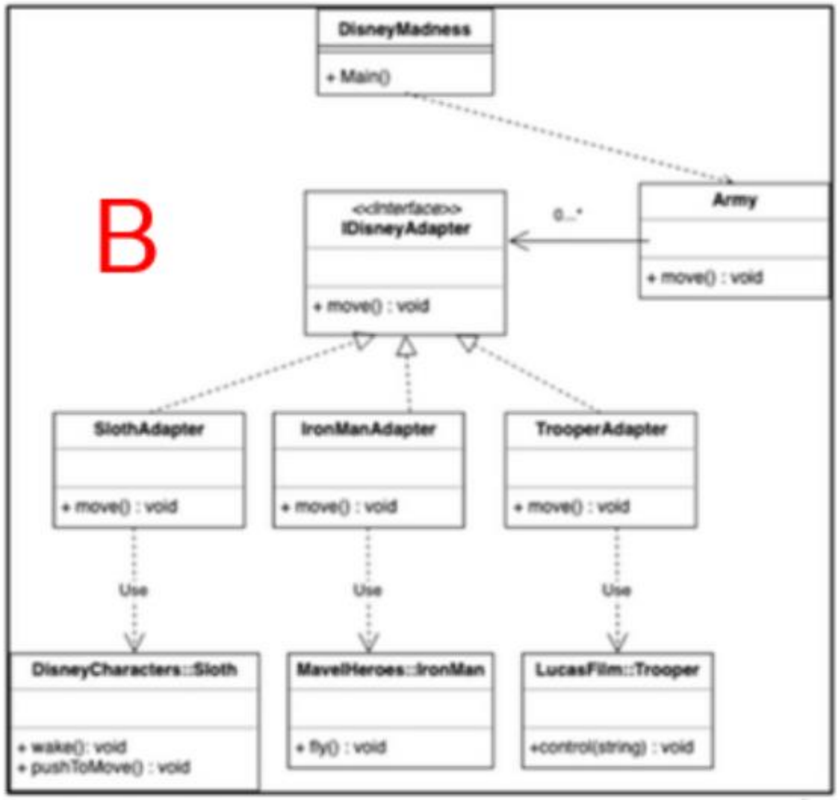
- Create a group of characters for DisneyMadness
  - Update the current diagram to use composite





Which of the following Diagrams should be for Composite Disney Characters





(Problem 2.2) Combining Policies

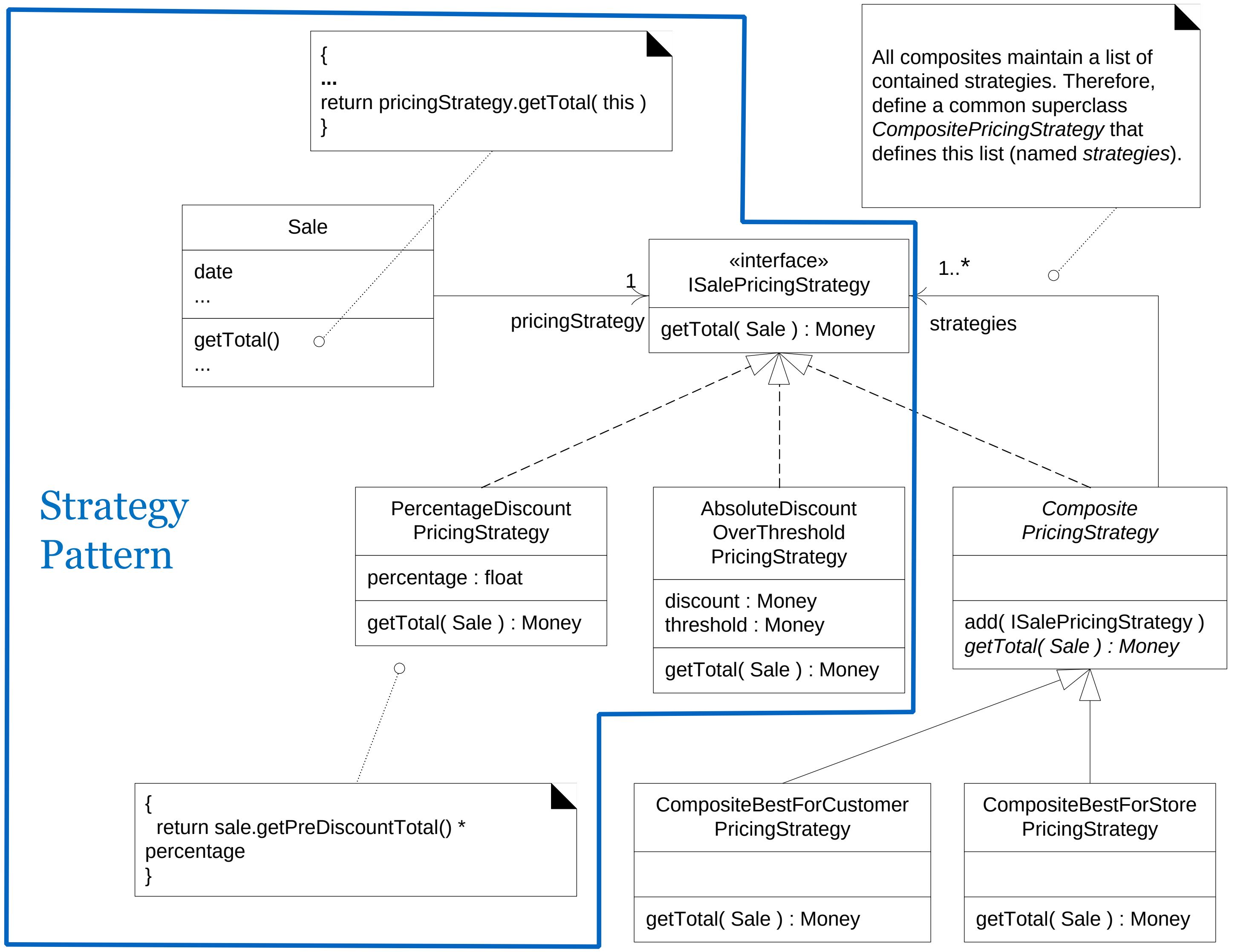
conflict resolution strategy:

- when multiple policies are applicable, how are these policies resolved?
  - Some discounts cannot be combined with others
  - **Possible policies: Best for customer or Best for store**

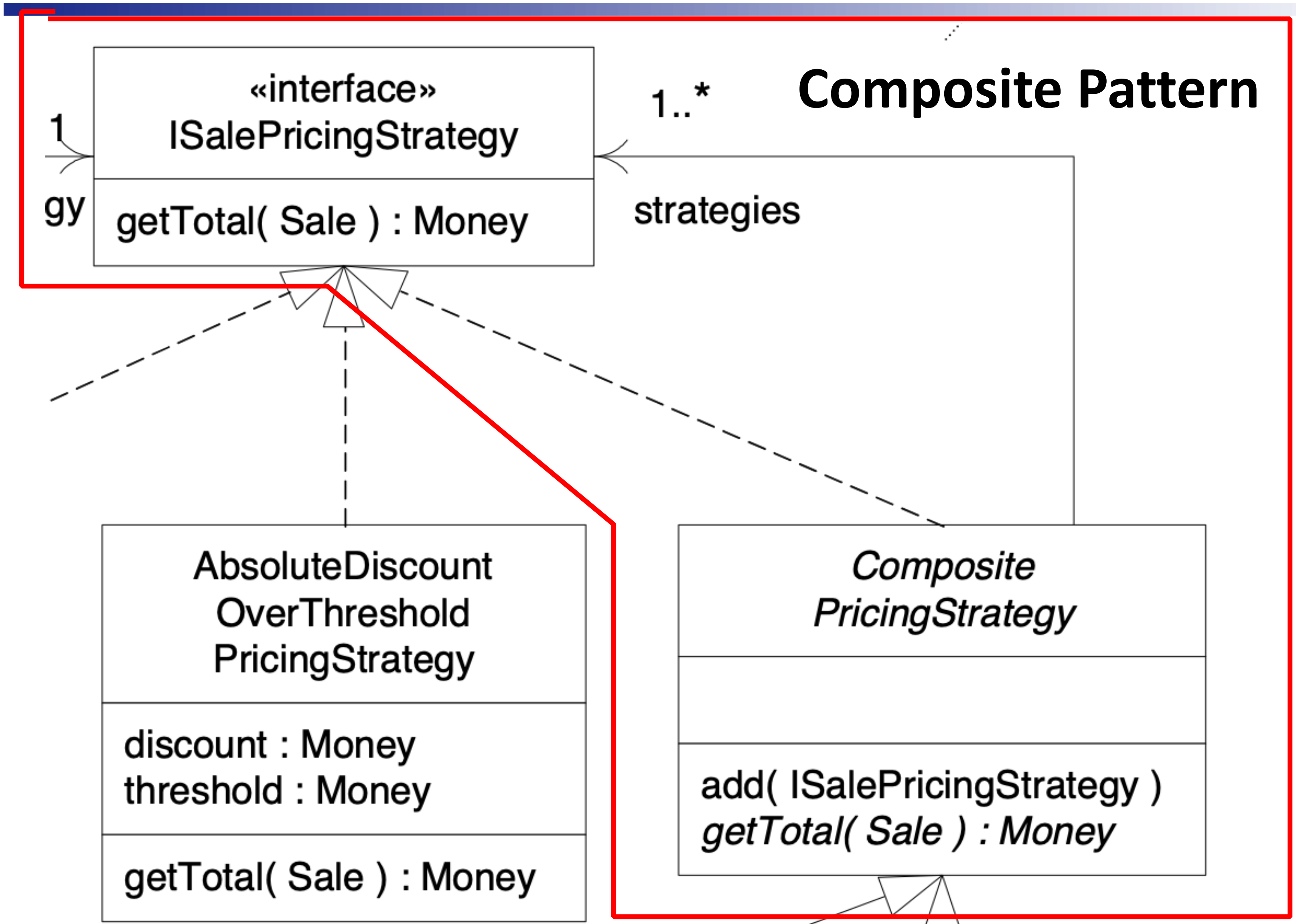
Composite pricing strategy:

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

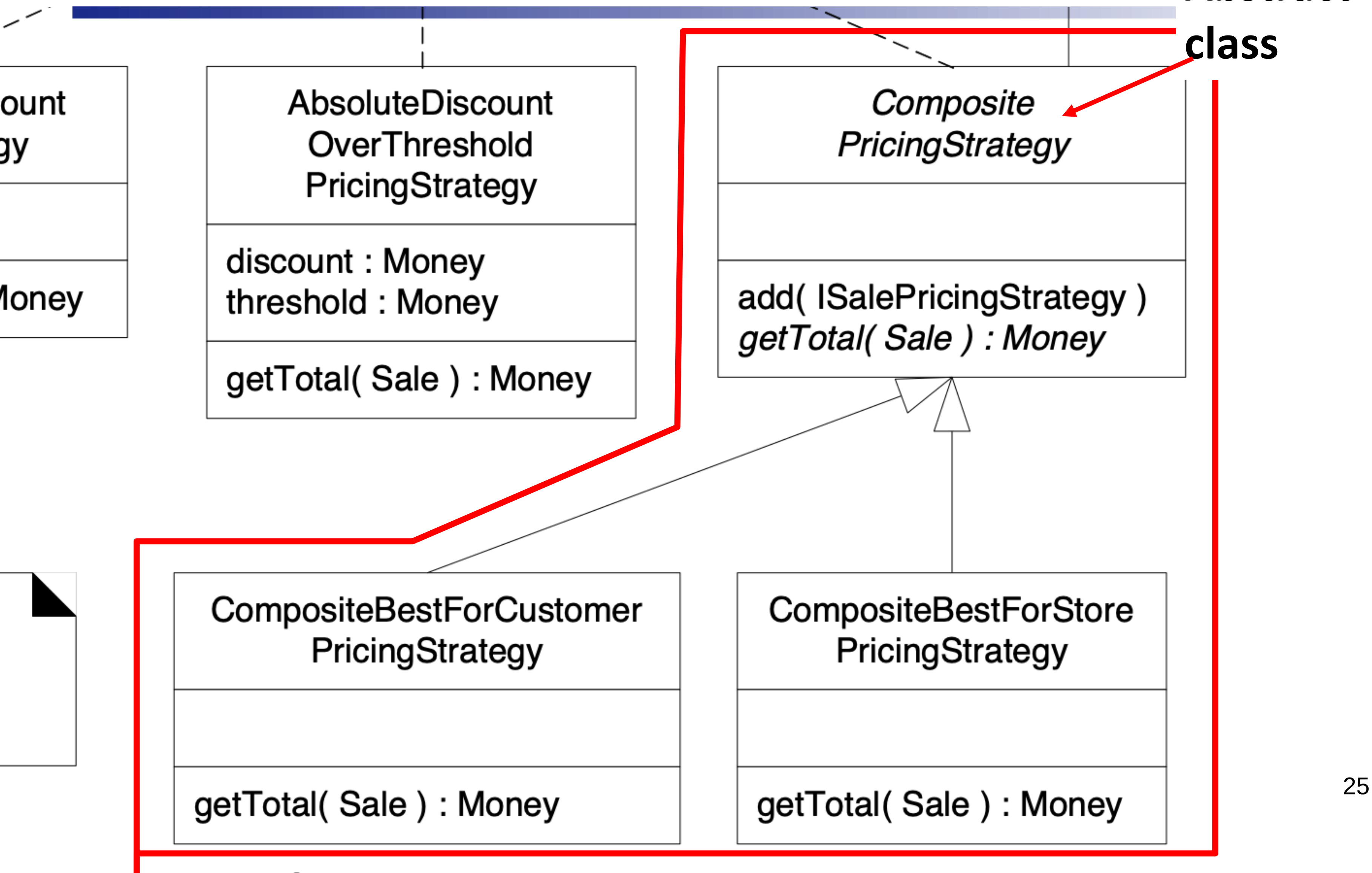
Composite Strategies


{
...
return pricingStrategy.getTotal(this)
}
All composites maintain a list of contained strategies. Therefore, define a common superclass *CompositePricingStrategy* that defines this list (named *strategies*).
Sale
date
...
getTotal()
...
1
pricingStrategy
«interface»
ISalePricingStrategy
getTotal(Sale) : Money
1..*
strategies
Strategy Pattern
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

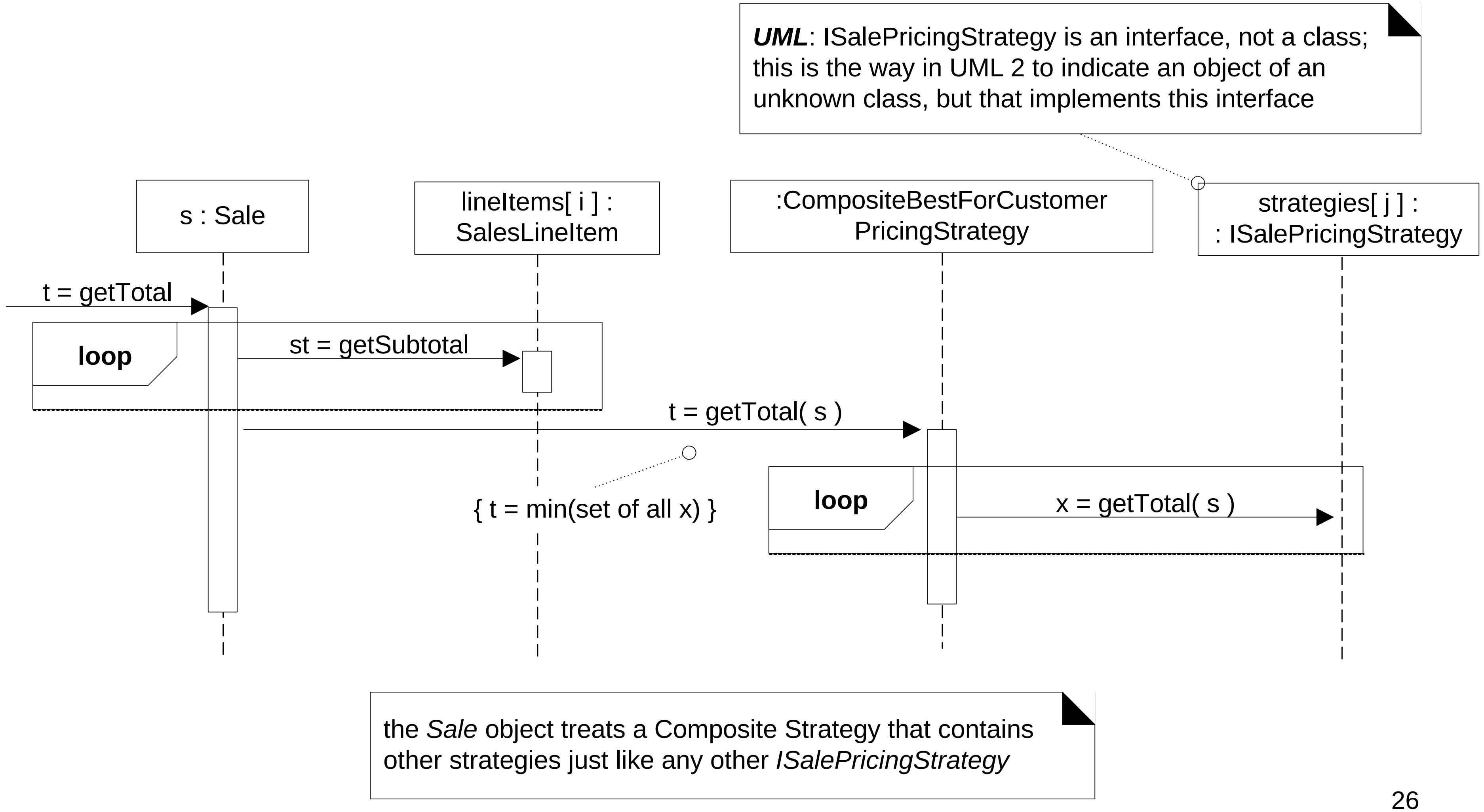
Composite Strategies



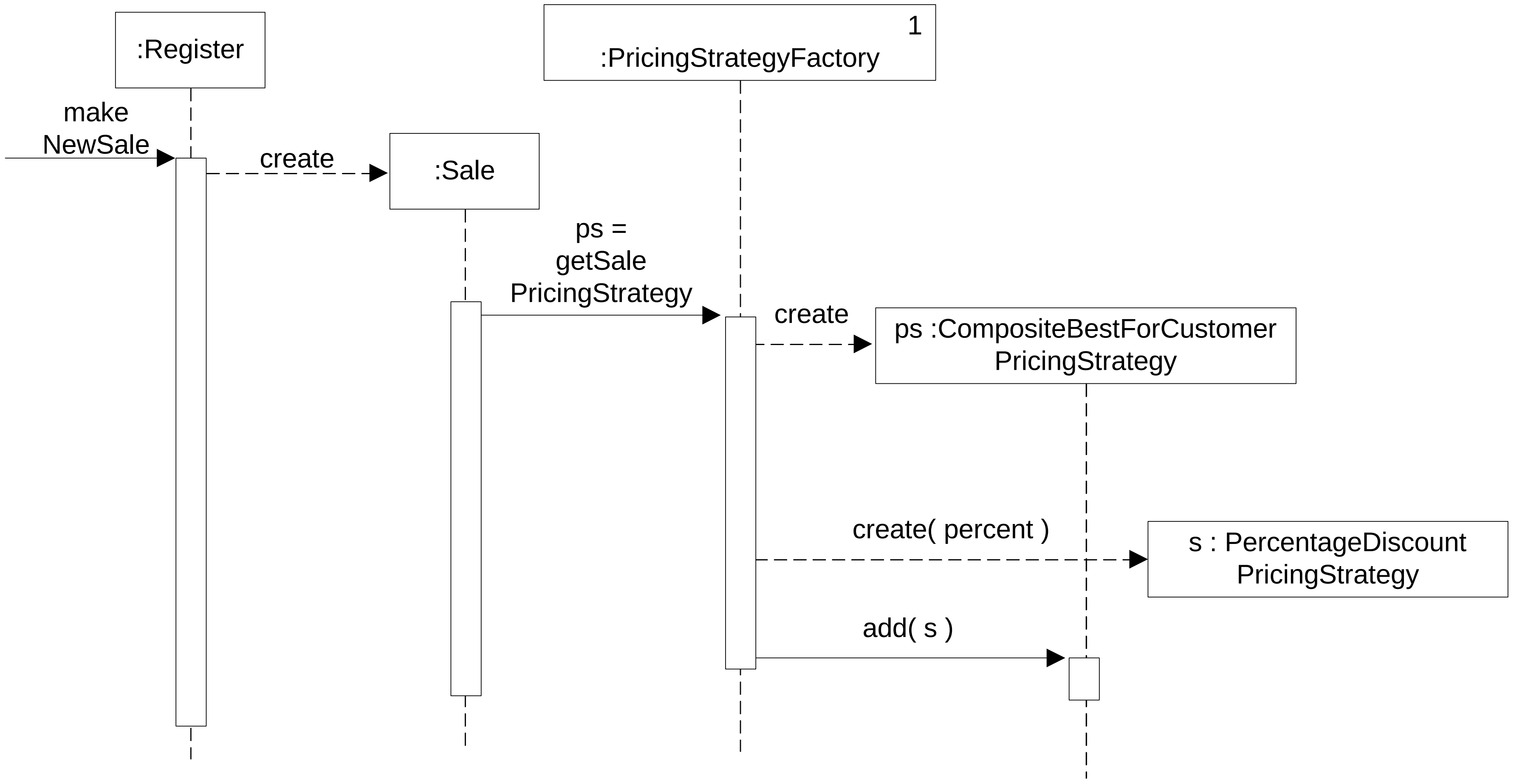
Composite Strategies


Abstract class
ount
gy
Ioney
AbsoluteDiscount OverThreshold PricingStrategy
discount : Money
threshold : Money
getTotal(Sale) : Money
Composite PricingStrategy
add(ISalePricingStrategy)
getTotal(Sale) : Money
CompositeBestForCustomer PricingStrategy
getTotal(Sale) : Money
CompositeBestForStore PricingStrategy
getTotal(Sale) : Money

Collaboration with a Composite


UML: ISalePricingStrategy is an interface, not a class; this is the way in UML 2 to indicate an object of an unknown class, but that implements this interface
s : Sale
lineItems[i] : SalesLineItem
:CompositeBestForCustomer PricingStrategy
strategies[j] : : ISalePricingStrategy
t = getTotal
loop
st = getSubtotal
t = getTotal(s)
{ t = min(set of all x) }
loop
x = getTotal(s)
the *Sale* object treats a Composite Strategy that contains other strategies just like any other *ISalePricingStrategy*

Creating a Composite Strategy


:Register
1
:PricingStrategyFactory
make NewSale
create
:Sale
ps = getSale PricingStrategy
create
ps :CompositeBestForCustomer PricingStrategy
create(percent)
s : PercentageDiscount PricingStrategy
add(s)

Example

```
public interface ISalePricingStrategy {
	public double getTotal(Sale s);
	public String getStrategyName();
}
```

ISalePricingStrategy.java

```
abstract class CompositePricingStrategy implements
ISalePricingStrategy {

	protected ArrayList<ISalePricingStrategy>
pricingStrategies = new ArrayList<ISalePricingStrategy>();

	public void add(ISalePricingStrategy strategy) {
	pricingStrategies.add(strategy);
	}

	public abstract double getTotal(Sale s);
}
```

CompositePricingStrategy.java

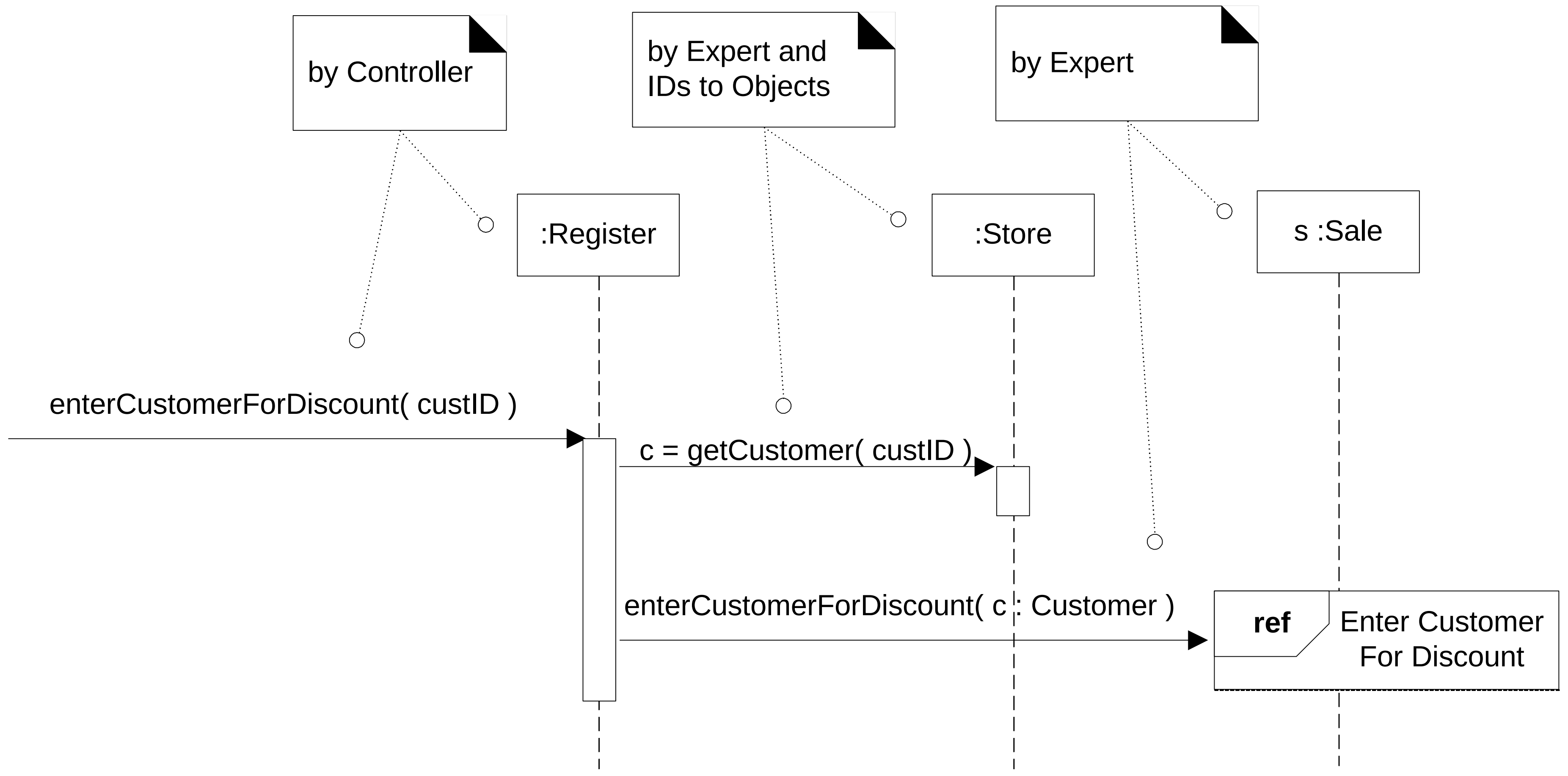
Example: Composite Strategy

```
public class CompositeBestForCustomerPricingStrategy extends
CompositePricingStrategy {
  private String selectedStrategy = null;
  //Get minimum total
  public double getTotal(Sale s) {
     double lowestTotal = s.getPreDiscountTotal();

     for(ISalePricingStrategy strat: this.pricingStrategies){
        double total = strat.getTotal(s);
        if(lowestTotal > total) {
             lowestTotal = total;
        }
      }
      return lowestTotal;
  }
}
```

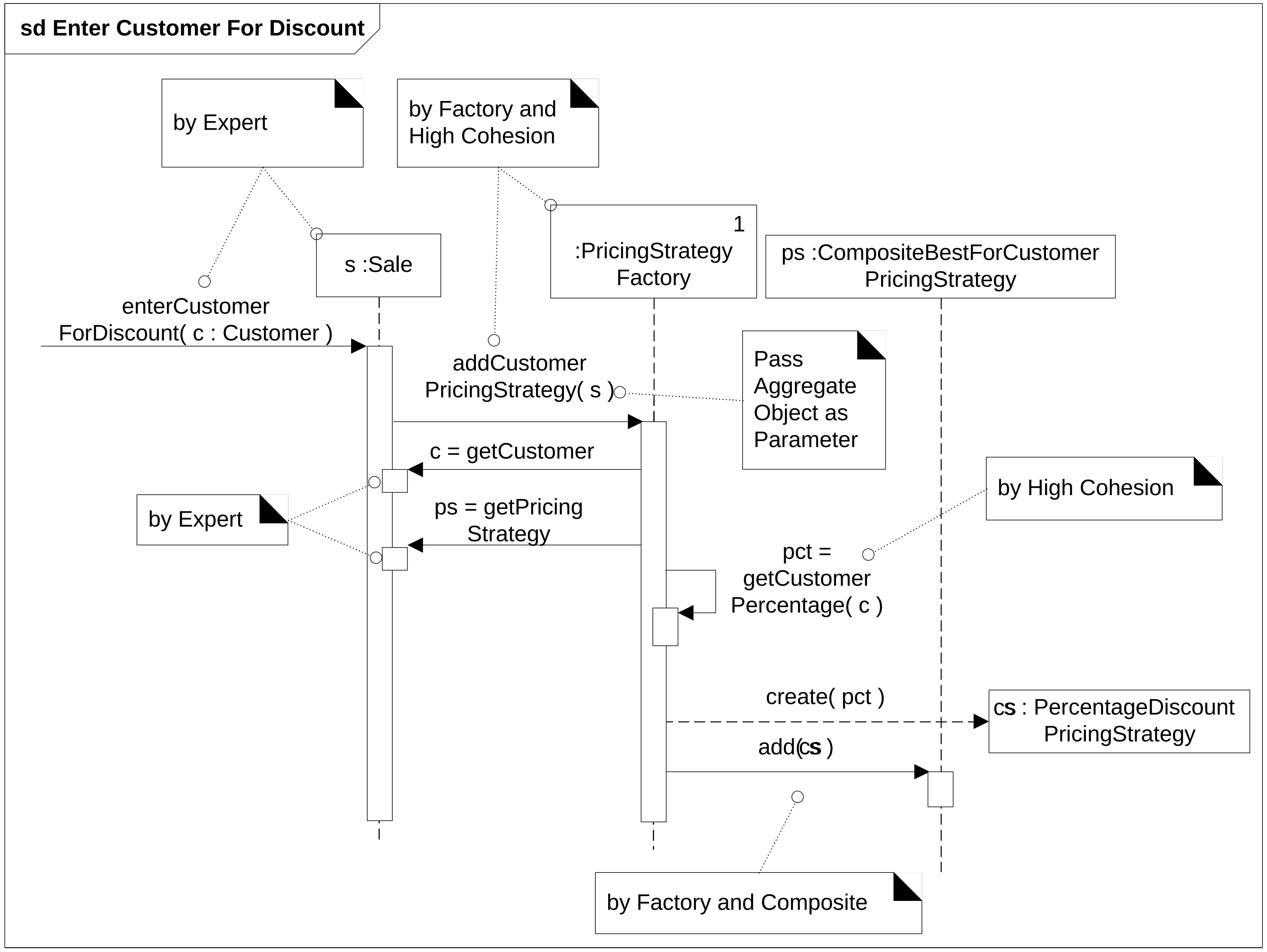
Example: Composite Strategy Factory

```
public class PricingStrategyFactory {
  public ISalePricingStrategy getCompositeSalePricingStrategy() {
   ISalePricingStrategy applicableStrategies = null;
   LocalDateTime today = LocalDateTime.now();
   switch(today.getDayOfWeek()) {
     case MONDAY:
         CompositePricingStrategy composStrat = new
                        CompositeBestForCustomerPricingStrategy();
         composStrat.add(new PercentDiscountPricingStrategy());
         composStrat.add(new TeaDiscountPricingStrategy());
         applicableStrategies = composStrat;
         break;
        case FRIDAY:
           applicableStrategies = new
                AbsoluteDiscountOverThresholdPricingStrategy();
           break;
}
return applicableStrategies;
  }
}
```

Creating Discount Pricing Strategy (1)


by Controller
by Expert and IDs to Objects
by Expert
:Register
:Store
s :Sale
enterCustomerForDiscount(custID)
c = getCustomer(custID)
enterCustomerForDiscount(c : Customer)
ref
Enter Customer For Discount

Creating Discount Pricing Strategy (2)

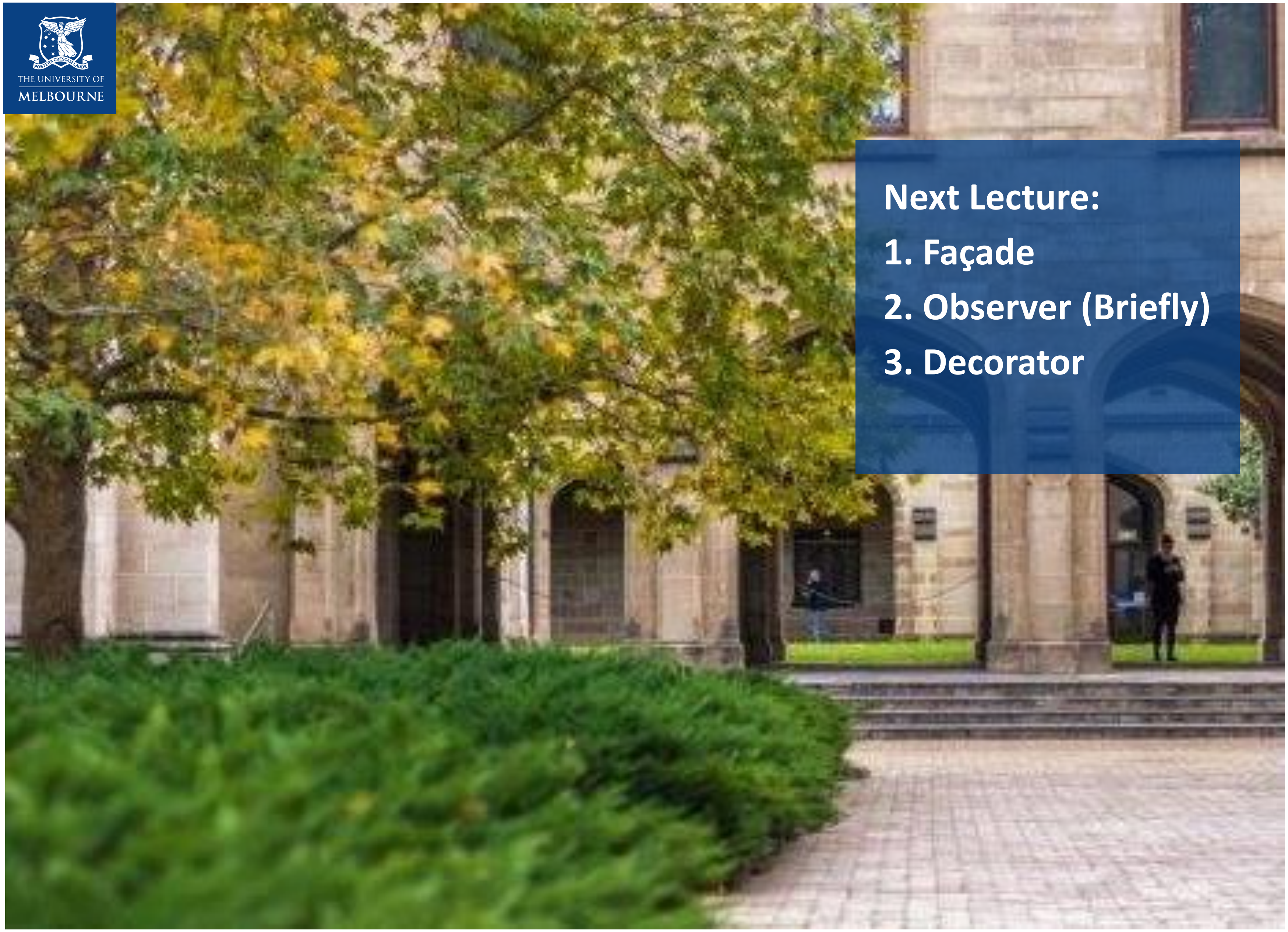

sd Enter Customer For Discount
by Expert
by Factory and High Cohesion
s :Sale
1
:PricingStrategy Factory
ps :CompositeBestForCustomer PricingStrategy
enterCustomer ForDiscount(c : Customer)
addCustomer PricingStrategy(s)
Pass Aggregate Object as Parameter
c = getCustomer
by Expert
ps = getPricing Strategy
by High Cohesion
pct = getCustomer Percentage(c)
create(pct)
cs : PercentageDiscount PricingStrategy
add(cs)
by Factory and Composite

Summary: Complex Pricing Logic

- POS has various pricing strategies/discounts and **some** of them cannot be combined

Solution:

- “To handle this problem, let’s use Composite Strategy”
- Design reasoning based on:
  - Protected Variation
  - Polymorphism
  - High Cohesion
  - Low Coupling
  - Strategy
  - Composite
  - Factory
  - Singleton



Next Lecture:

1. Façade
2. Observer (Briefly)
3. Decorator

Lecture Identification

Coordinator: Patanamon Thongtanunam

Lecturer: Peter Eze

Semester: S2 2020

These slides include materials from:

> *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*, Third Edition, by Craig Larman, Pearson Education Inc., 2005.