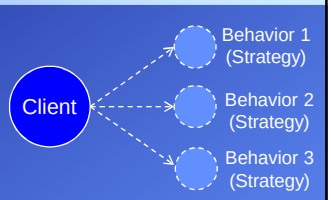
GoF Design Patterns (cont'd)

The Strategy Pattern (Behavioral)



A certain behavior of a class may change during the lifetime (runtime) of an object of this class.

Example:

Such a design problem in the example POS system is the complex pricing policy, such as discount for the day, senior citizen discounts, and so forth.

The pricing strategy (which may also be called a rule, policy, or algorithm) for a sale can vary.

For example,

- Mondays it may be 10% and Thursdays 5% off all sales,
- It may be 10TL off if the sale total is greater than 200TL,
- For customers with a loyalty card there may be other discounts.

All these different algorithms (pricing strategies) seem to be variations of the getTotal() responsibility (behavior) of the Sale class.

However, to add all these algorithms into getTotal() method of the Sale using if-then-else or switch-case statements, will cause coupling and cohesion problems.

All changes in pricing strategies will affect the Sale.

Definition: Strategy

Problem:

How to design for varying, but related, algorithms or policies?

How to design for the ability to change these algorithms or policies?

(A certain behavior of a class may change during the lifetime of an object of this class.)

Solution:

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

Solution of the problem with different pricing strategies:

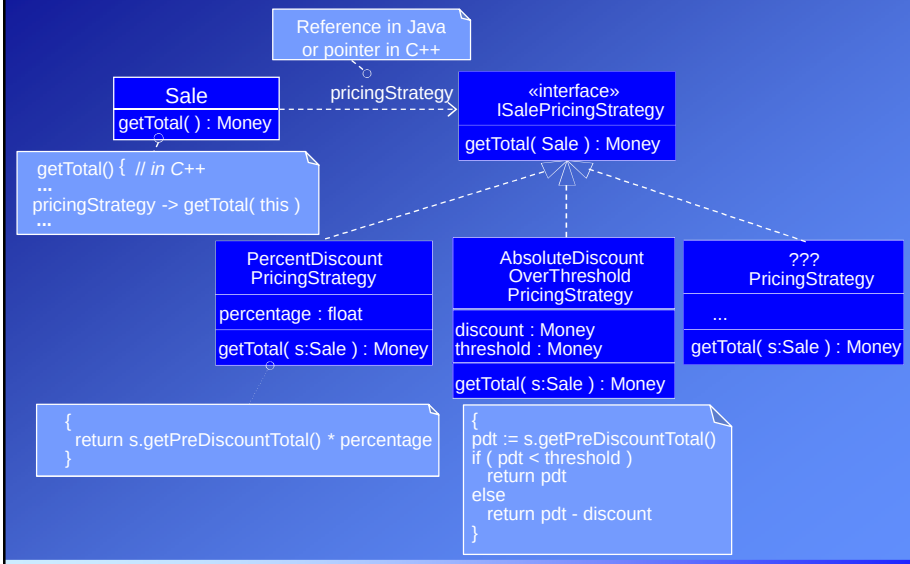
According to the strategy pattern, we create multiple SalePricingStrategy classes, for different discount algorithms, each with a polymorphic getTotal method.

The implementation of each getTotal method will be different: PercentDiscountPricingStrategy will discount by a percentage, and so on.

Each getTotal method takes the Sale object as a parameter, so that the pricing strategy object can find the pre-discount price from the Sale, and then apply the discounting rule.

Example:

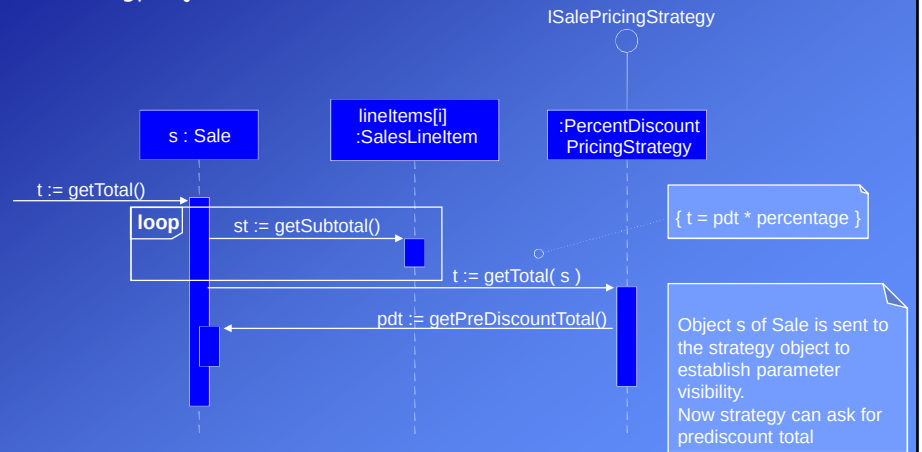
Solution of the problem about different discount policies using the Strategy.



A strategy object is attached to a context object the object to which it applies the algorithm.

In this example, the context object is the Sale.

When a getTotal message is sent to a Sale object, it delegates some of the work to its strategy object.



Underlying OO Principles of the Strategy pattern:

Principle: "*Find what varies and encapsulate it*"

This is one of the principles that the Strategy pattern is based on.

In our case study, varying parts are different discount strategies.

We separate these varying parts (pricing policies) from the stable part (Sale) of the system and encapsulate (group) them behind an abstract class (or interface in Java).

(Remember "Protected Variations")

The details (types) of these strategies are hidden for the user (Sale).

The context object (Sale) must include a reference or a pointer (C++) to the interface (Java) or base class of different strategies.

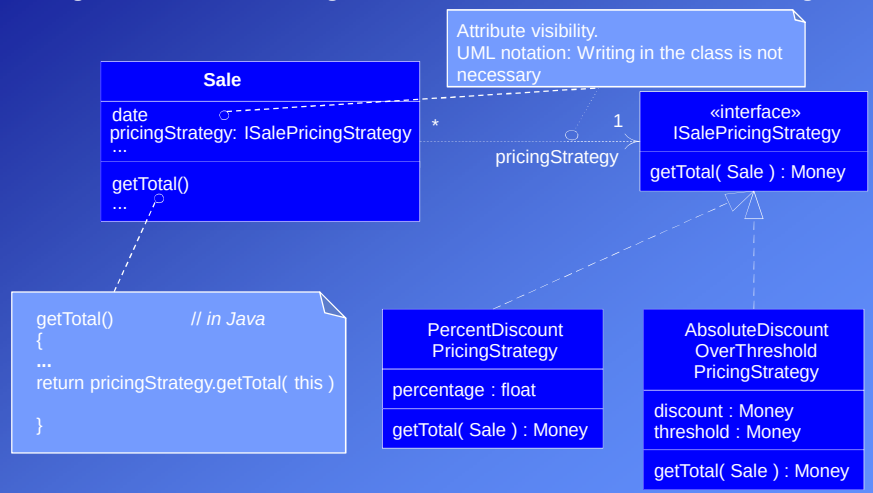
So, it gets attribute visibility to its strategy and can be connected to different strategy objects in runtime.

- Principle 1: "*Find what varies and encapsulate it*"

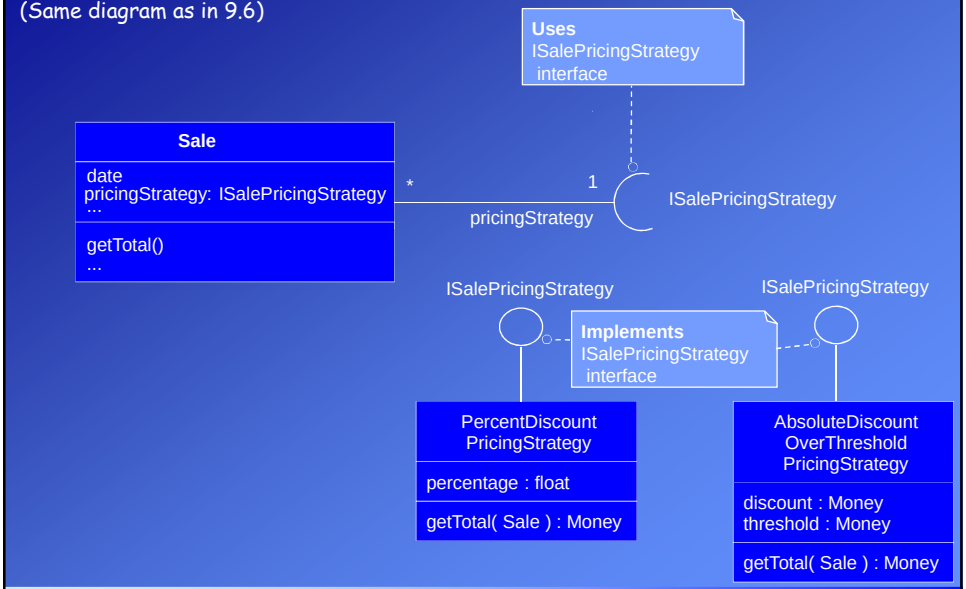
We found varying strategies and encapsulated them.

- Principle 2: "*Design to interface not to implementation*"

We designed the Sale according to common interface of different strategies.



UML 2.X notation for interface *implementation* and *usage*.
(Same diagram as in 9.6)

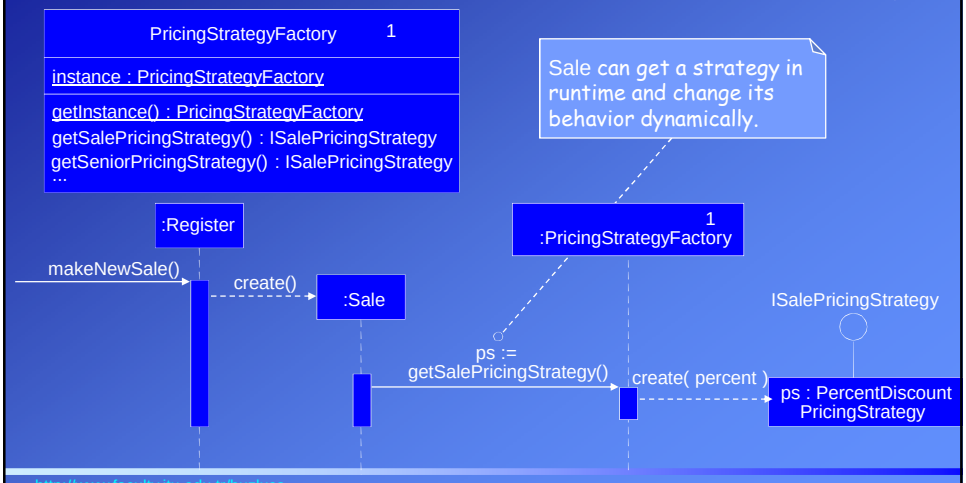


Creating strategies with a Factory:

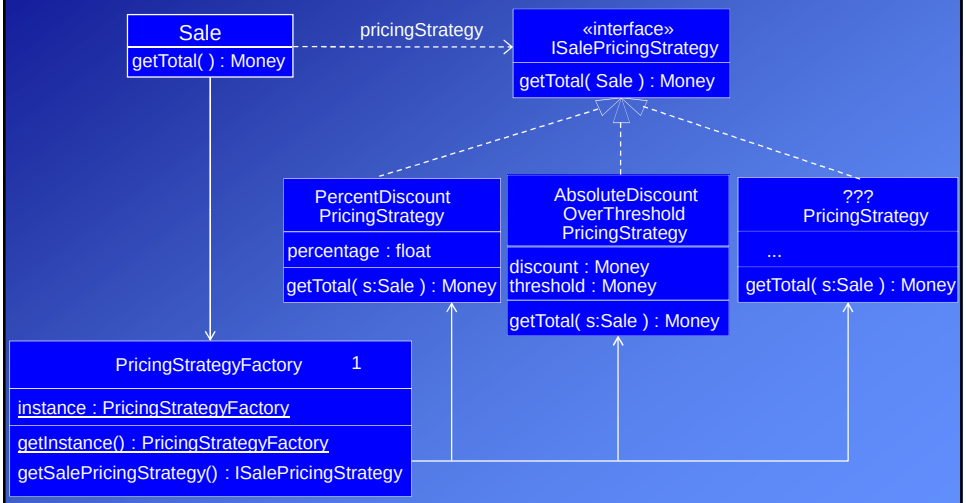
The Factory pattern can be applied to create the necessary strategy object.

A PricingStrategyFactory can be responsible for creating strategies.

The new factory is different than the ServicesFactory. This supports the goal of High Cohesion, each factory is focused only on creating a related family of objects.



Factory and strategies:

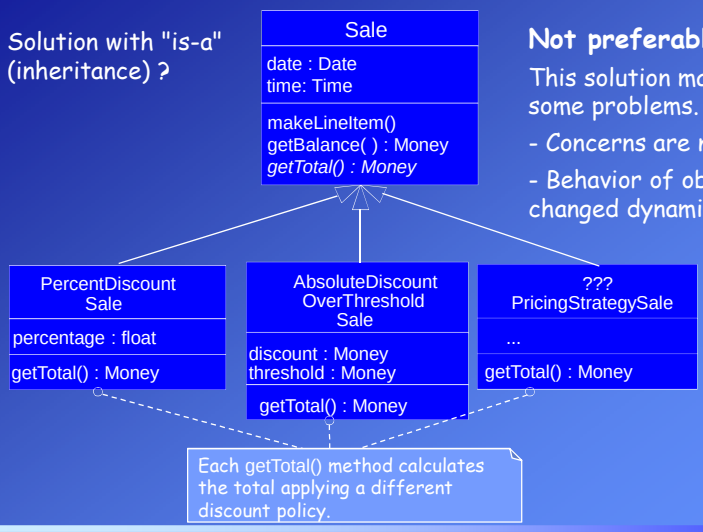


Discussion: *Favor composition over inheritance* principle

Is it possible to solve the same problem using the Inheritance?

We assume that we have different Sale classes with different pricing policies.

Solution with "is-a" (inheritance) ?



Not preferable!

This solution may work but it has some problems.

- Concerns are not separated.
- Behavior of objects cannot be changed dynamically.

Discussion: Favor composition over inheritance principle

Advantages of the solution with **composition** (has-a relation):

- Separation of concerns: Each class focuses on own task (Sale – Pricing Strategy)
- Flexibility: Sale can request a new strategy from the factory at any time and can change its behavior dynamically.

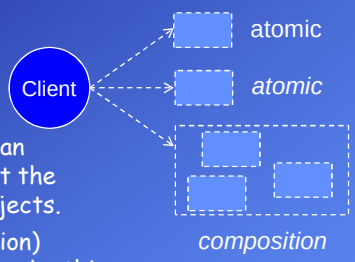
  There is a weak connection (only a pointer or reference) between the context object (Sale) and strategies.

Disadvantages of the solution with **inheritance** (is-a relation):

- Concerns are not separated : We have only Sale classes. Different tasks are mixed in the same class.
- Inflexibility: If we create a Sale object of a specific type (for example PercentDiscountSale) we cannot change its behavior dynamically.
- We must decide the pricing strategy during the creation of the sale.
- If we want to use another pricing strategy we have to delete the existing object and create a new one.

  There is a strong connection between the base class and the derived classes.

The Composite Pattern (Structural)



Sometimes a client object may get a service from an individual (atomic) object and sometimes it may get the same service from a composition (collection) of objects.

The client object treats them (atomic or composition) identically (polymorphically), and does not have to make this distinction.

Definition:

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

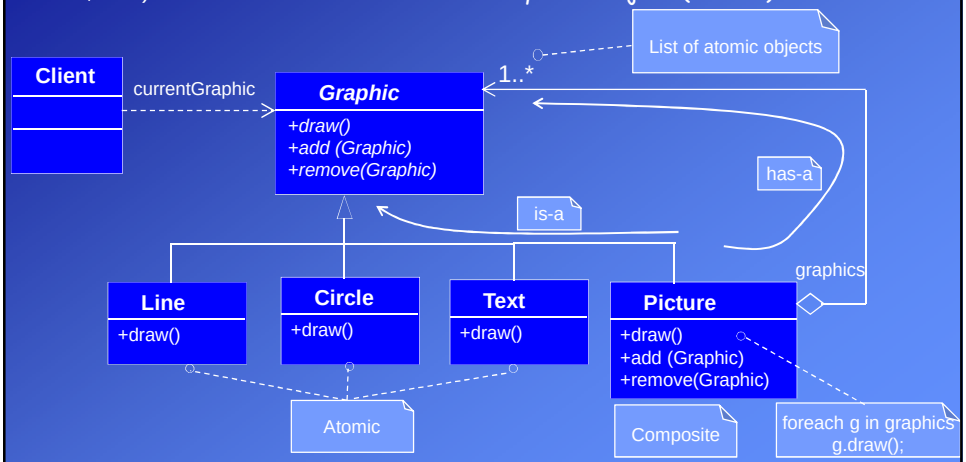
Solution: Define classes for composite and atomic objects so that they implement the same interface.

Add a list in the composite class that can include atomic objects.

Example: (From the book of the GoF)

In this example, we have *atomic* shape objects (Line, Circle, Text) and a *composite* object (Picture) that can include atomic shape objects.

The class Client can get the same service (draw) from an *atomic* object (Line, Circle, Text) and at the same time from a *composite* object (Picture).



Example: (From Larman)

How do we handle the case of multiple, conflicting pricing policies?

For example, suppose that a store has the following policies:

- On Monday, there is 10TL off purchases over 100TL
- Preferred customer discount of 15%.
- Buy the product of the day, get 5% discount off of everything.

If a preferred customer buys the product of the day and spends 150TL on Monday, what pricing policy should be applied?

Components of the problem:

1. Objects of the Sale class are sometimes connected to a single pricing strategy (atomic) and sometimes to a collection (composition) of strategies.

   The composite strategy solve this part of the problem.
2. The pricing strategies are dependent on different attributes of the Sale: Date, total, customer type, a particular line item product .
3. Different strategies are conflicting.

We need to find solutions also for 2 and 3.

Solution:

We create a composite class CompositePricingStrategy that is derived from the same base class (ISalePricingStrategy) as the atomic strategies.

This composite class can also contain other ISalesPricingStrategy objects.

A list in the CompositePricingStrategy class contains currently valid pricing strategies. (Composite pattern)

How to handle different conflicting strategies in the composite object is another strategy. (Strategy pattern again)

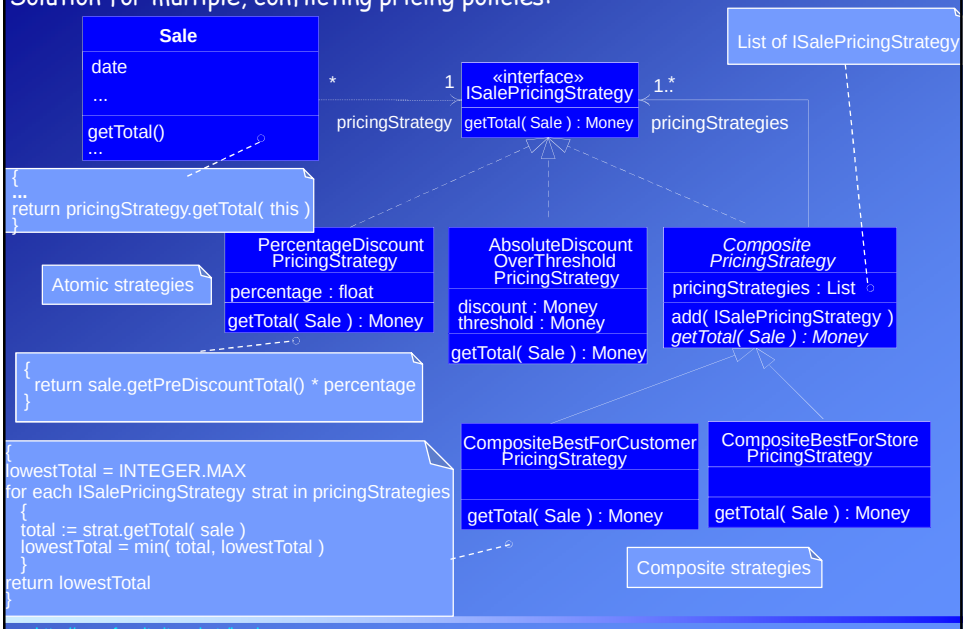
For example, the CompositeBestForCustomerPricingStrategy can try all strategies in its list and apply the strategy which returns the lowest total.

Another composite strategy (not so realistic) can be CompositeBestForStorePricingStrategy, which returns the highest total.

We can attach either a composite CompositeBestForCustomerPricingStrategy object (which contains other strategies inside of it) or an atomic PercentDiscountPricingStrategy object to the Sale object.

The Sale does not know or care if its pricing strategy is an atomic or composite; it looks the same to the Sale object, because they are all derived from the same base class ISalePricingStrategy.

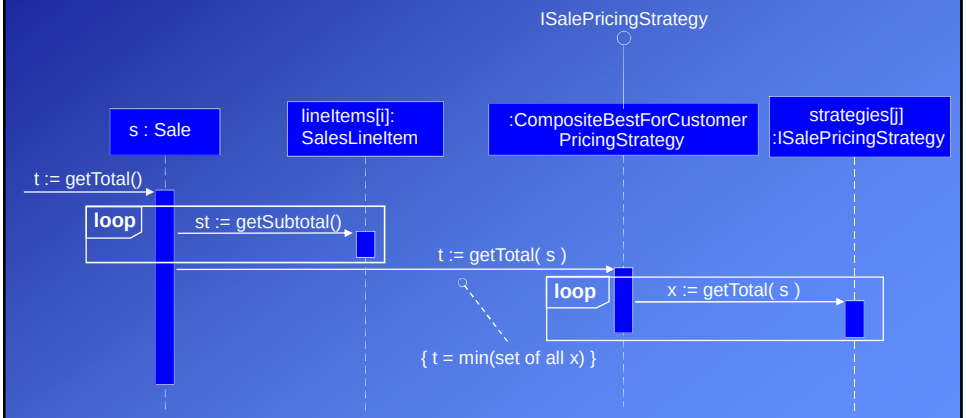
Solution for multiple, conflicting pricing policies:



Collaboration with a Composite:

Sale can be attached to any object that implements the ISalePricingStrategy interface and understands the getTotal message.

In this example, a CompositeBestForCustomerPricingStrategy object is attached to Sale.




```
// superclass so all subclasses can inherit a List of strategies
public abstract class CompositePricingStrategy implements ISalePricingStrategy
{
  protected List pricingStrategies = new ArrayList();

  public add( ISalePricingStrategy s )
  {
    pricingStrategies.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 = pricingStrategies.iterator(); i.hasNext(); )
     {
       ISalePricingStrategy strategy = (ISalePricingStrategy)i.next();
       Money total = strategy.getTotal( sale );
       lowestTotal = total.min( lowestTotal );
      }
      return lowestTotal;
   }
 } // end of class
```

Absrtract Composite

List of atomic strategies

To add a new atomic strategy to the list

Concrete Composite

This composite strategy returns the lowest total.

Creating Multiple Sale Pricing Strategies

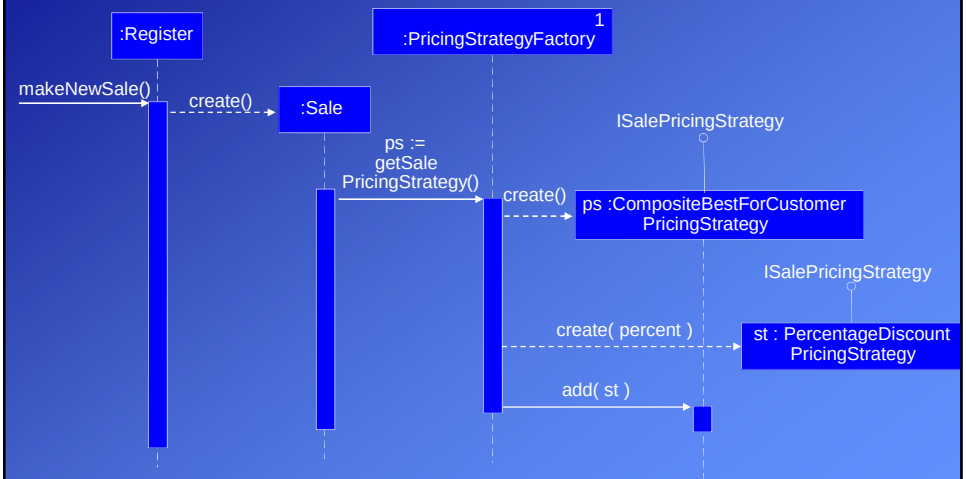
When an object of the Sale is created, it can request a strategy from the factory PricingStrategyFactory.

According to current conditions the factory can decide to create a composite strategy such as the CompositeBestForCustomerPricingStrategy.

At the beginning the factory can add the present moment's store discount policy (which could be set to 0% discount if none is active), such as some PercentageDiscountPricingStrategy to the composite object.

Then, if at a later step in the scenario, another pricing strategy is discovered (such as preferred customer discount), it will be easy to add it to the composite, using the CompositePricingStrategy.add method.

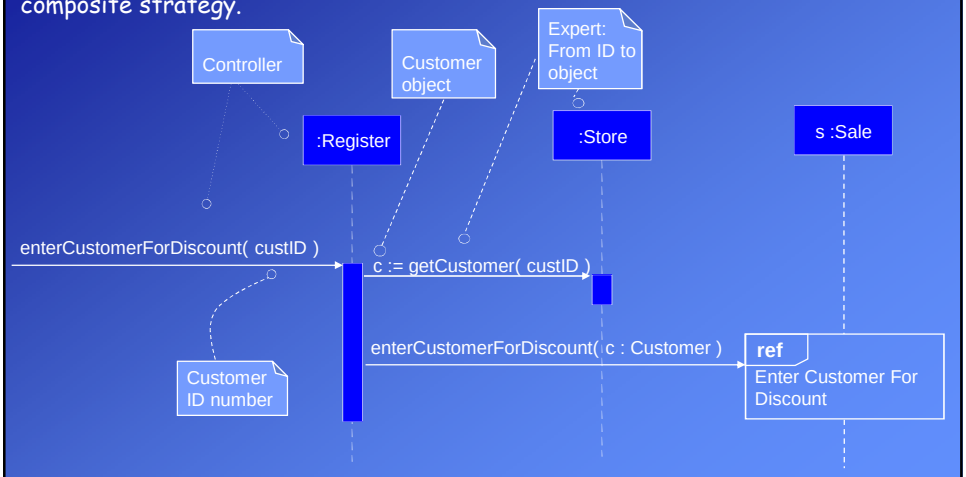
Example: Creating Multiple Sale Pricing Strategies

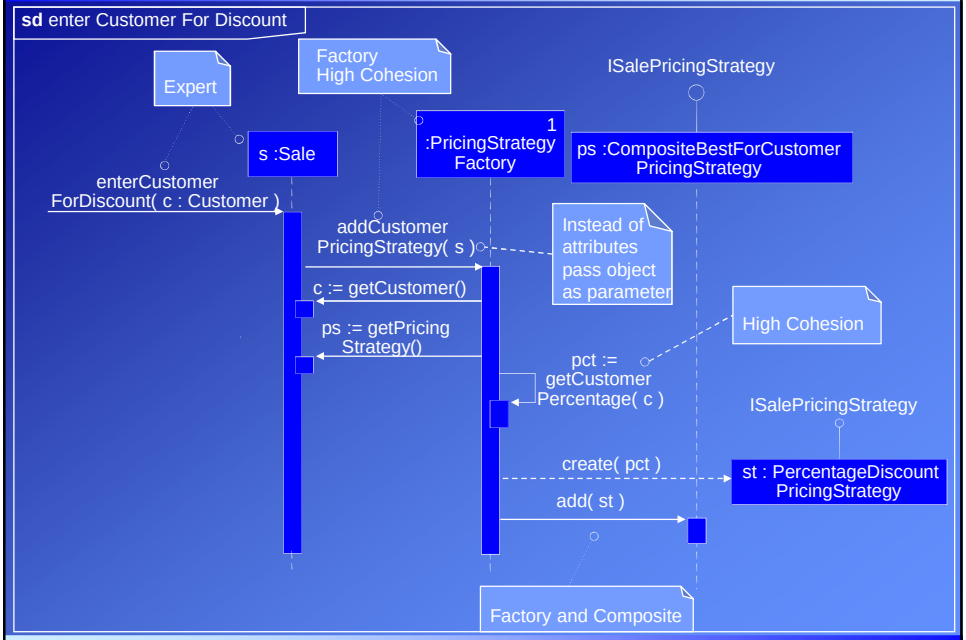


Example: Creating the pricing strategy for a preferred customer discount:

If there are preferred customers in this system we need to handle a new system operation: enterCustomerForDiscount

If there is a valid discount for this customer it needs to be added to the composite strategy.





Considering principles and patterns in the design about customer discount

- Why do not the Register send a message to the PricingStrategyFactory, to create this new pricing strategy and then pass it to the Sale?

  Reason is to support *Low Coupling*. The Sale is already coupled to the factory.

  Furthermore, the Sale is the *Information Expert* that knows its current pricing strategy.

- customerID is transformed into a Customer object by the Store .

  Reason: By *Information Expert* and the goal of *low representational gap*, the Store can know all the Customers.

  The Register asks the Store, because the Register already has attribute visibility to the Store (from earlier design work).

- Why to transform the customerID (perhaps a number) into a Customer object?

  It doesn't have a pattern name but this is a common practice in object design to transform keys and IDs for things into true objects.

  Having a true Customer object that contains information about the customer, and which can have behavior becomes beneficial and flexible as the design grows.

  Remember: itemID into a ProductDescription object in the enterItem operation.

Considering principles and patterns in the design (cont'd)

- Passing aggregate object as parameter:

  In the addCustomerPricingStrategy(s:Sale) message we pass a reference to the Sale object s to the factory, and then the factory asks for the Customer and PricingStrategy from the Sale.

  Why not to just send these two parameters to the factory?

  Principle: Instead of individual attributes or child objects, pass the aggregate object (actually the reference) that contains child objects (or attributes).

  Reason: Following this principle increases flexibility, because then the factory can collaborate with the entire Sale in ways we may not have previously foreseen as necessary.

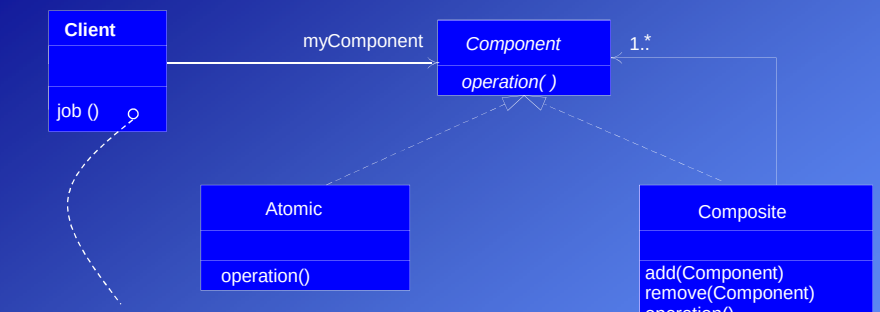
  In future steps of the design new parameters (attributes) may be necessary.

  In this case, we don't need to change interfaces of our methods; the factory can get them from Sale by calling the necessary get functions.

  Note: The composite pattern is not used only with the strategies.

  This pattern provides that a client object treats individual objects (atomic) and group of objects (composition) identically (polymorphically), and does not have to make this distinction.

General Structure of the Composite Pattern:




```
public class Client {
  private Component myComponent;  // reference (or pointer)

   public job()
   {
      ......
      myComponent.operation();  // it can be atomic or composite
   }

 }
```