Larman Gang of Four Patterns

- The following slides present Larman's examples of some of the more common GoF software design patterns, from the Gang of Four book (the "bible")
  - Design Patterns by Gamma, Helm, Johnson, Vlissides (1995)

Larman's Basic GRASP Patterns/Principles

- GRASP
  - Information Expert
  - Creator
  - Controller
  - Low Coupling (evaluation pattern)
  - High Cohesion (evaluation pattern)

  - Polymorphism,
  - Pure fabrication
  - Indirection
  - Protected Variation

Gang of Four Patterns

Larman relates the GoF patterns to the underlying principles

For example, the Adapter pattern is an example of Indirection and Pure Fabrication that makes use of Polymorphism …

Adapter (GoF)

Name: Adapter

Problem:

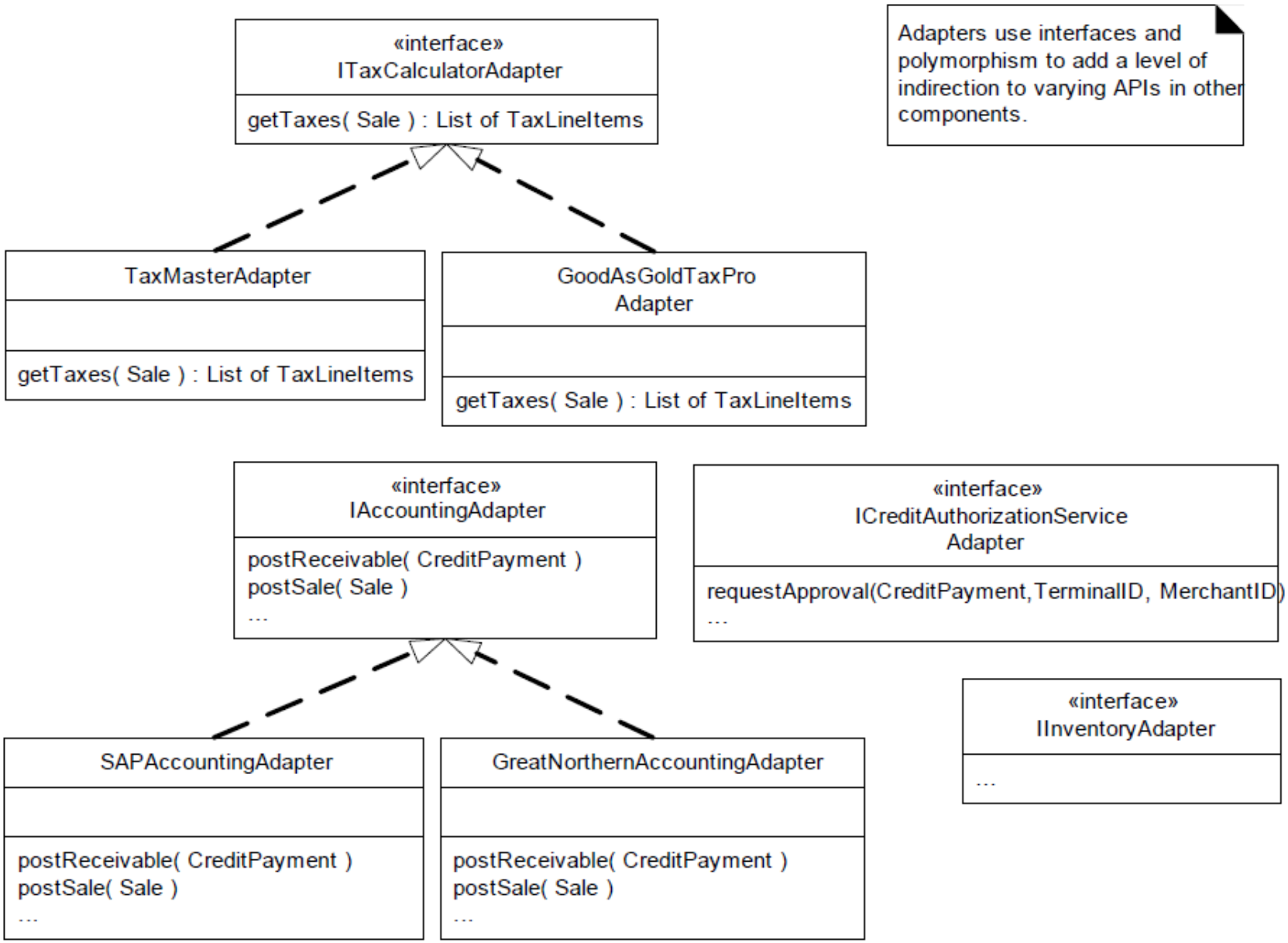
- How to resolve incompatible interfaces, or provide a stable interface to similar components with different interfaces?

Solution:

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

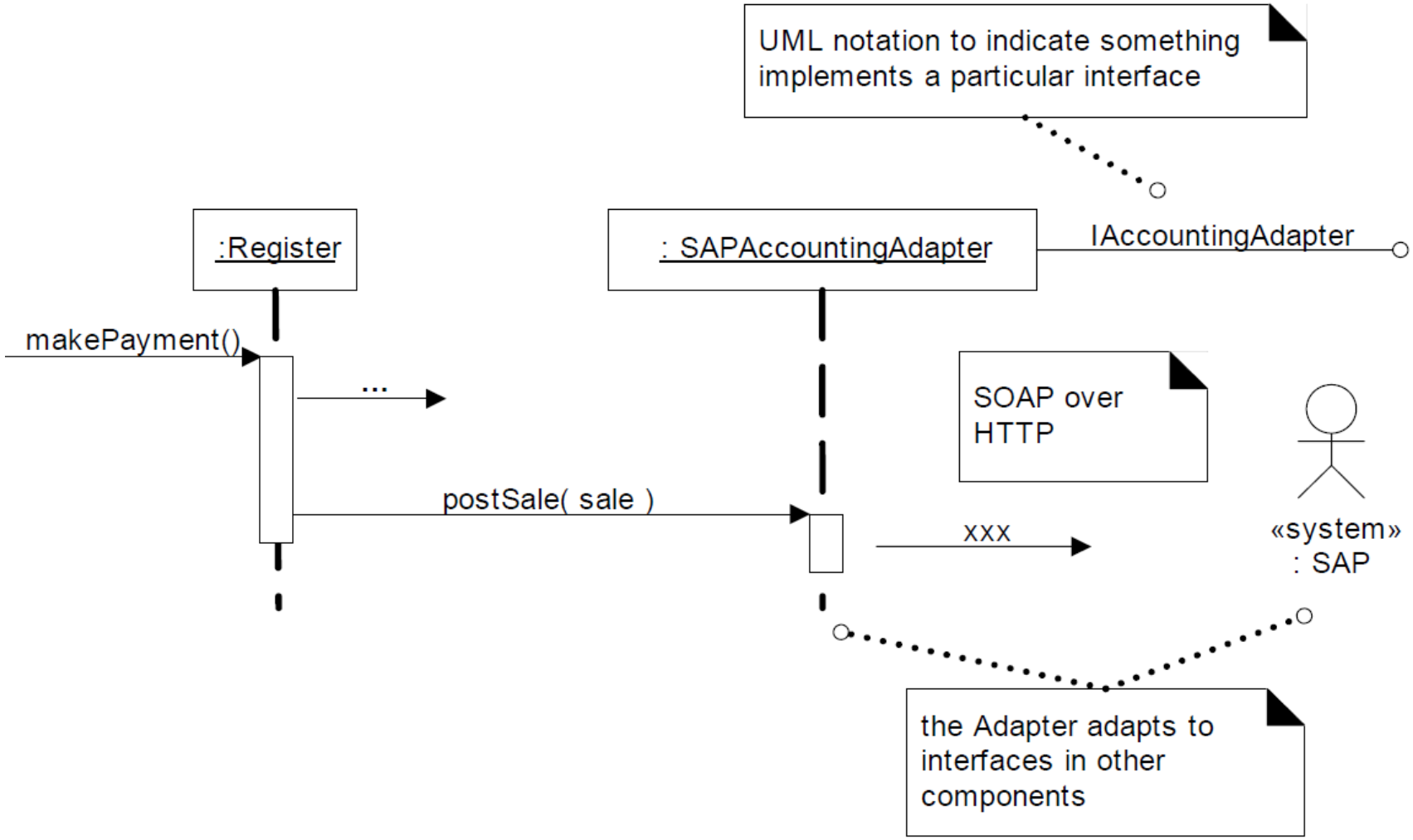
Example the Point of Sale (POS)

- The NextGen POS system needs to support several kinds of external third-party services, including tax calculators, credit authorization services, inventory systems, and accounting systems, among others. Each has a different API, which can't be changed.
- A solution is to add a level of indirection with objects that adapt the varying external interfaces to a consistent interface used within the application.


«interface»
ITaxCalculatorAdapter
getTaxes(Sale) : List of TaxLineItems
Adapters use interfaces and polymorphism to add a level of indirection to varying APIs in other components.
TaxMasterAdapter
getTaxes(Sale) : List of TaxLineItems
GoodAsGoldTaxPro Adapter
getTaxes(Sale) : List of TaxLineItems
«interface»
IAccountingAdapter
postReceivable(CreditPayment)
postSale(Sale)
...
«interface»
ICreditAuthorizationService Adapter
requestApproval(CreditPayment, TerminalID, MerchantID)
...
«interface»
IInventoryAdapter
...
SAPAccountingAdapter
postReceivable(CreditPayment)
postSale(Sale)
...
GreatNorthernAccountingAdapter
postReceivable(CreditPayment)
postSale(Sale)
...

Example

- a particular adapter instance will be instantiated for the chosen external service3, such as SAP for accounting, and will adapt the *postSale* request to the external interface, such as a SOAP XML interface over HTTPS for an intranet Web service offered by SAP.


UML notation to indicate something implements a particular interface
:Register
: SAPAccountingAdapter
IAccountingAdapter
makePayment()
...
SOAP over HTTP
postSale(sale)
xxx
«system» : SAP
the Adapter adapts to interfaces in other components

Factory

- Simplification of GoF Abstract factory

- Deals with creating objects
- Issue of domain objects doing more than just the application logic – e.g. calculations

Factory

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:

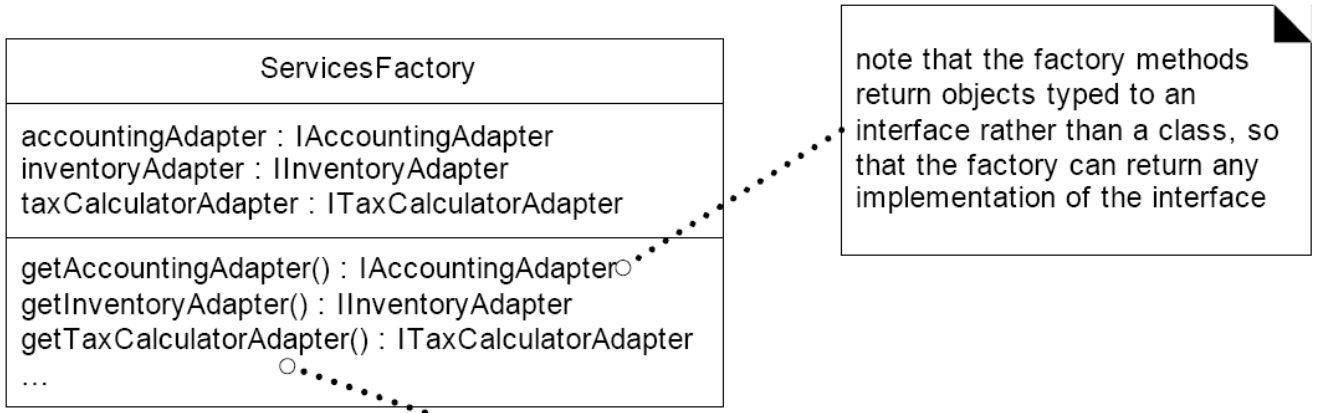
- Create a Pure Fabrication object called a Factory that handles the creation.

Factory

Benefits:

- Separate the 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

Factory




```
{
  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;
}
```

Pseudocode for getTaxCalculatorAdapter

Factory Example

- In the *ServicesFactory*, the logic to decide which class to create is resolved by reading in the class name from an external source (for example, via a system property if Java is used) and then dynamically loading the class. This is an example of a partial **data-driven design.** This design achieves Protected Variations with respect to changes in the implementation class of the adapter. Without changing the source code in this factory class, we can create instances of new adapter classes by changing the property value and ensuring the new class is visible in the Java class path for loading.

Singleton

Problem:

- Exactly one instance of a class is allowed—it is a "singleton." Objects need a global and single point of access.

Solution:

- Define a static method of the class that returns the singleton.

i.e. the basic idea is that class X defines a static method *getInstance* that itself provides a single instance of X.

Singleton

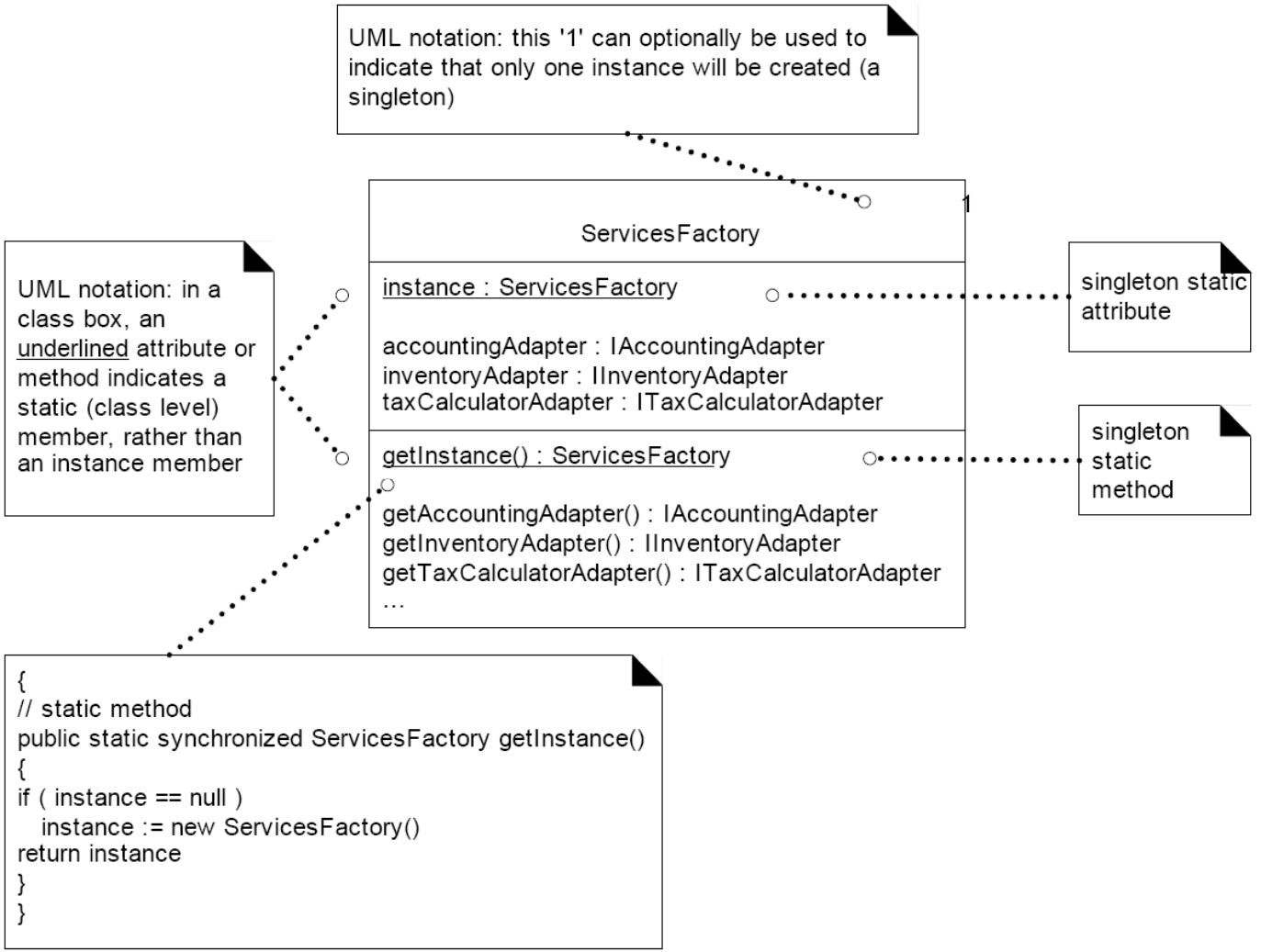
“First, observe that only one instance of the factory is needed within the process. Second, quick reflection suggests that the methods of this factory may need to be called from various places in the code, as different places need access to the adapters for calling on the external services. Thus, there is a visibility problem: how to get visibility to this single *ServicesFactory* instance?
One solution is pass the *ServicesFactory* instance around as a parameter to wherever a visibility need is discovered for it, or to initialize the objects that need visibility to it, with a permanent reference. This is possible but inconvenient; an alternative is the **Singleton pattern.**
Occasionally, it is desirable to support global visibility or a single access point to a single instance of a class rather than some other form of visibility. This is true for the *ServicesFactory* instance.”

Singleton

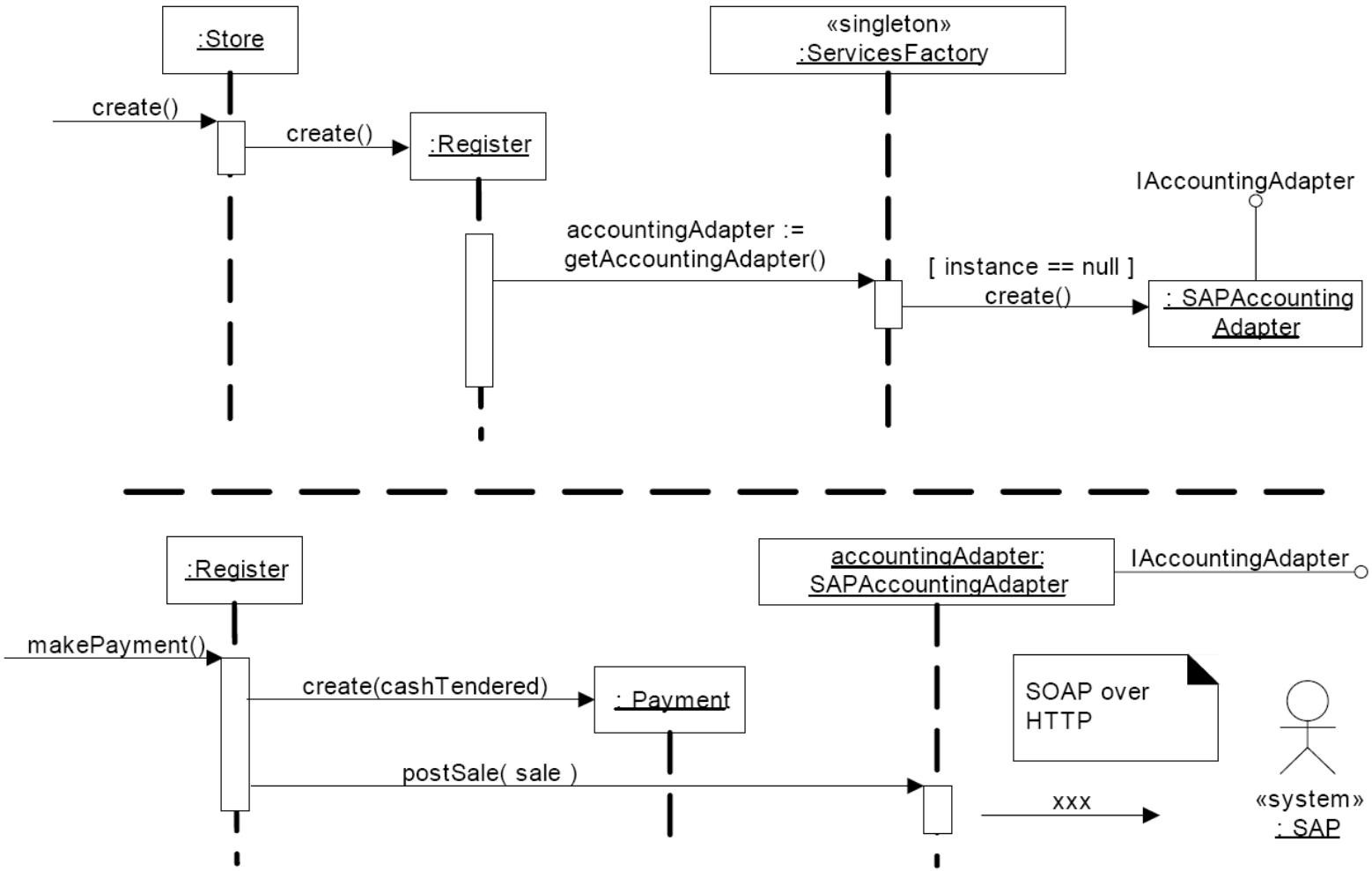
```
public class Register {
public void initialize()
{
... do some work ...
     // accessing the singleton Factory via the getInstance call
      accountingAdapter =
          ServicesFactory.getInstance().getAccountingAdapter();
   ... do some work ... }
// other methods...
}
```

Since visibility to public classes is global in scope (in most languages), at any point in the code, in any method of any class, one can write *SingletonClass.get-Instance()* in order to obtain visibility to the singleton instance, and then send it a message, such as *SingletonClass.getInstance().doFoo().*

Singleton Example



Singleton, Factory, Adapter



Strategy

Problem:

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

Solution:

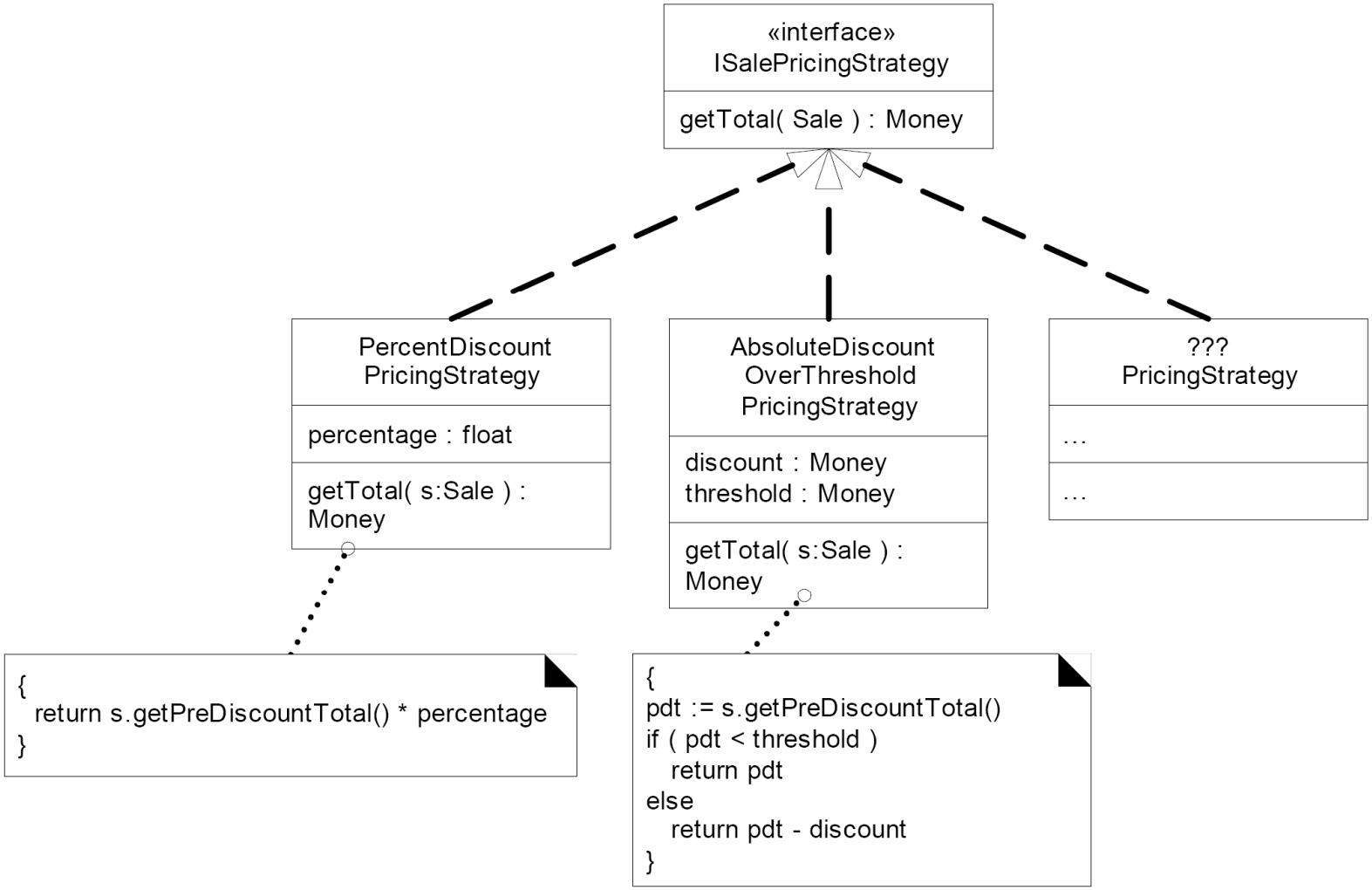
Define each algorithm/policy/strategy in a separate class, with a common interface.

Strategy

The pricing strategy for a sale can vary. During one period it may be 10% off all sales, later it may be $10 off if the sale total is greater than $200 etc. How do we design for these varying pricing algorithms?

Since the behavior of pricing varies by the strategy (or algorithm), we create multiple *SalePricingStrategy* classes, each with a polymorphic *getTotal* method. 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. The implementation of each *getTotal* method will be different: *PercentDiscountPricingStrategy* will discount by a percentage, and so on.

Strategy Example



Composite motivation

How do we handle the case of multiple, conflicting pricing policies? For example, if a store has the following policies in effect today (Monday):

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

If a senior who is also a preferred customer buys 1 case of Darjeeling tea, and $600 of veggieburgers. What pricing policy should be applied?

Composite motivation

There can exist multiple co-existing strategies, i.e. one sale may have several pricing strategies.

A pricing strategy can be related to the type of customer (for example, a senior). This has creation design implications: the customer type must be known by the *StrategyFactory* at the time of creation of a pricing strategy for the customer.

Similarly, a pricing strategy can be related to the type of product being bought. This likewise has creation design implications: the *ProductSpecification* must be known by the *StrategyFactory* at the time of creation of a pricing strategy influenced by the product.

Composite pattern: can change the design so that the *Sale* object does not know if it is dealing with one or many pricing strategies, and also offers a design for the conflict resolution.

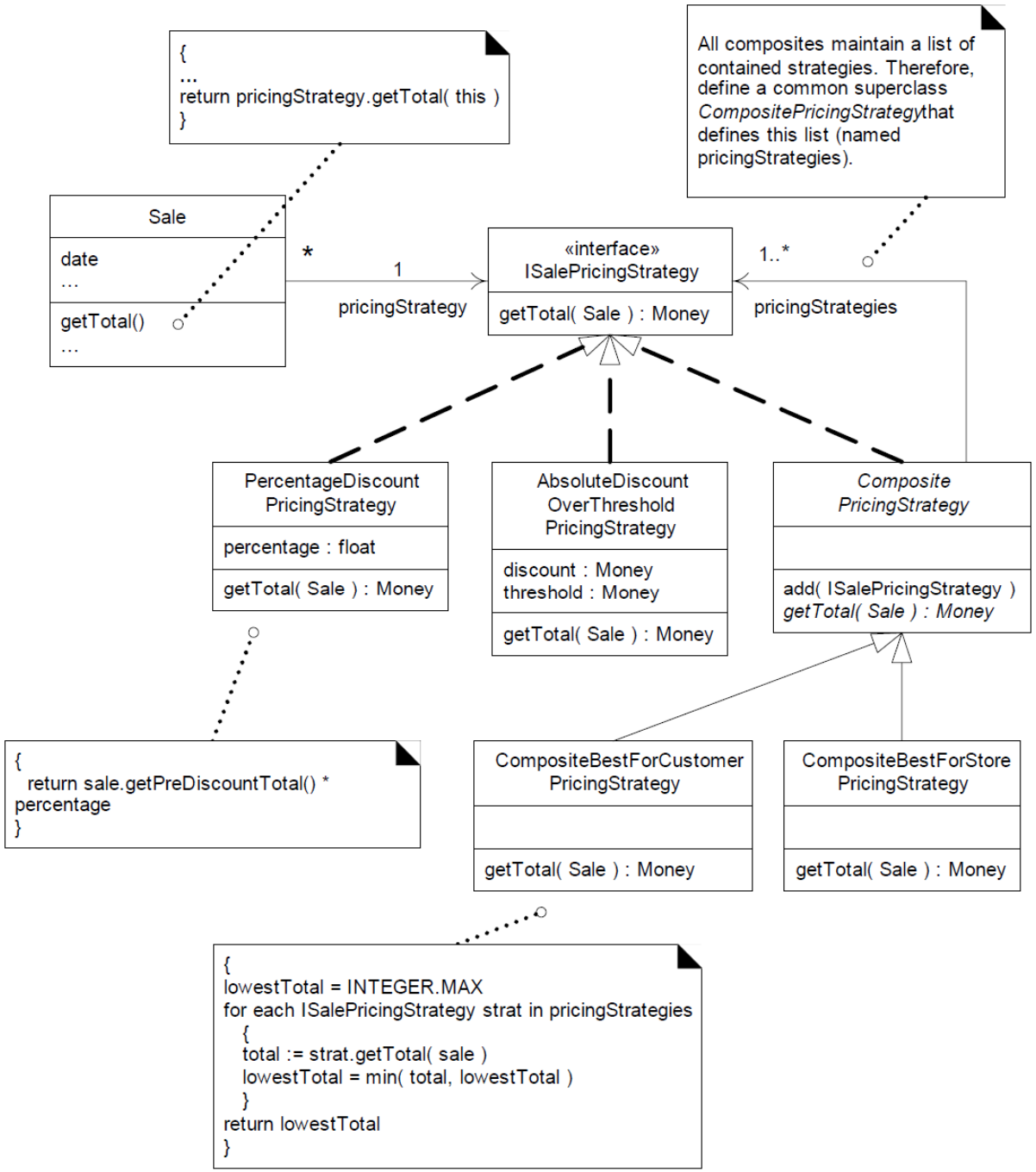
Composite

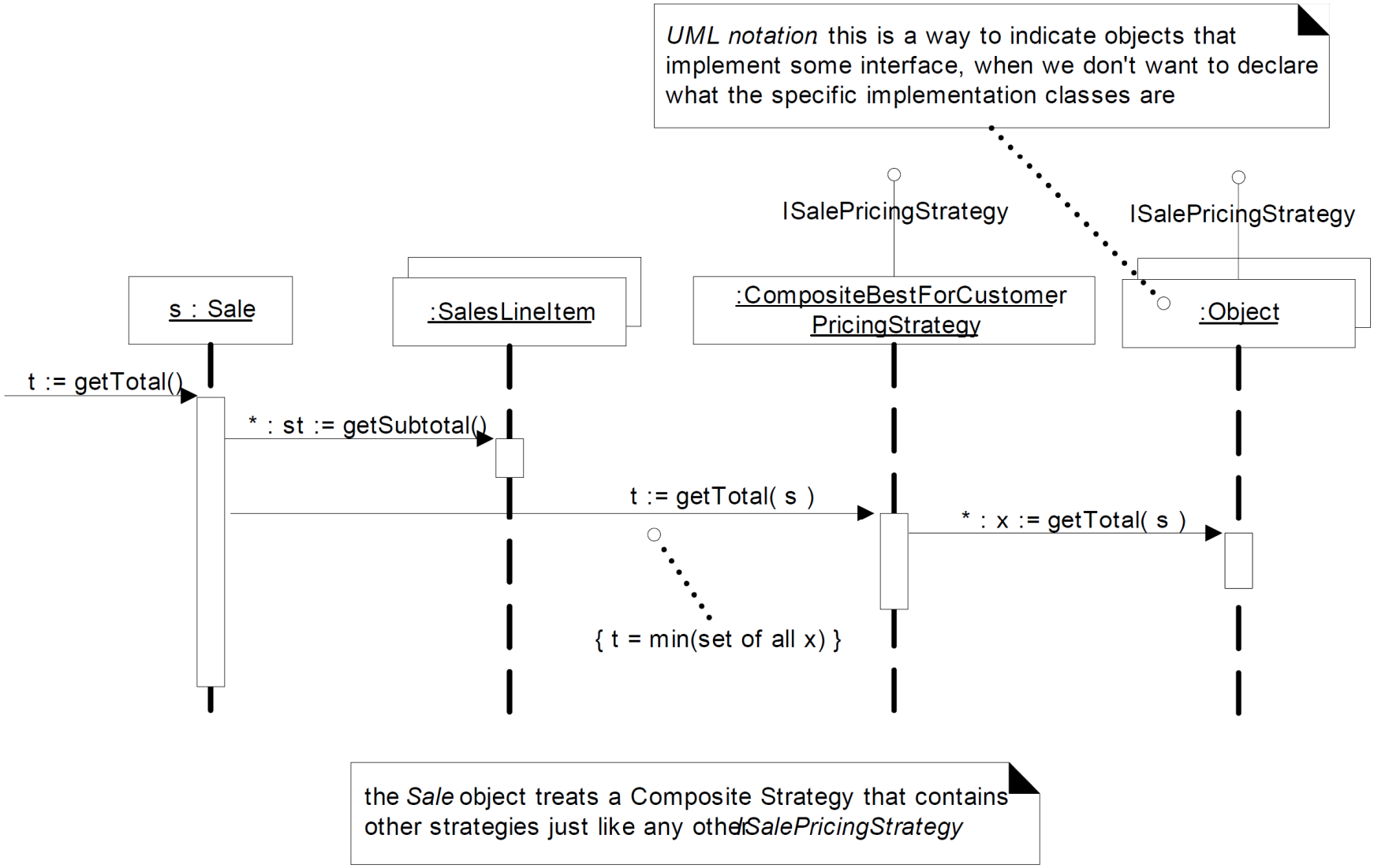
Problem:

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

Solution:

- Define classes for composite and atomic 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 CompositePricingStrategy that 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
}


UML notation this is a way to indicate objects that implement some interface, when we don't want to declare what the specific implementation classes are
ISalePricingStrategy
ISalePricingStrategy
s : Sale
:SalesLineItem
:CompositeBestForCustomer PricingStrategy
:Object
t := getTotal()
* : st := getSubtotal()
t := getTotal(s)
* : x := getTotal(s)
{ t = min(set of all x) }
the *Sale* object treats a Composite Strategy that contains other strategies just like any other *ISalePricingStrategy*

```
// 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
```

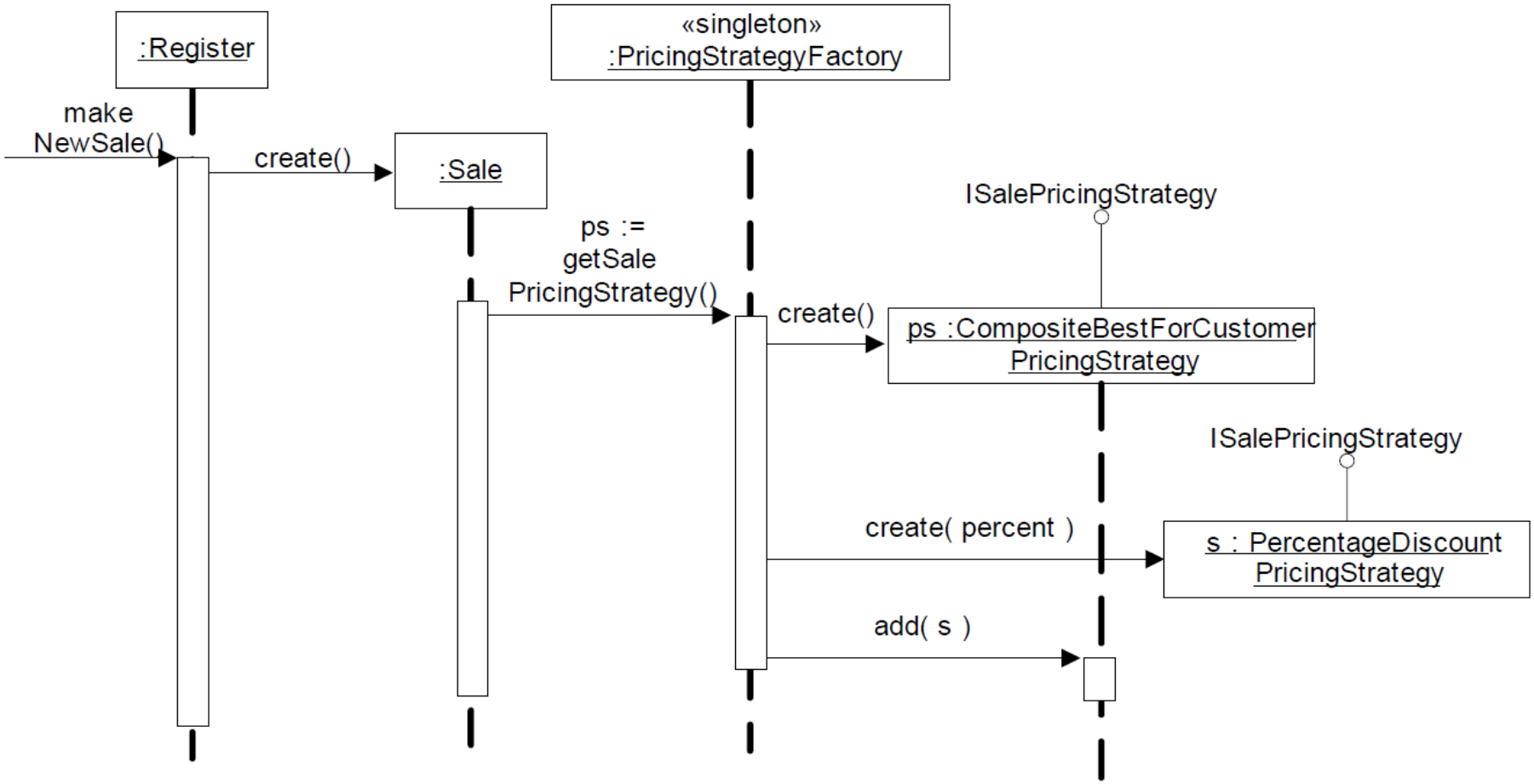
Creating Multiple SalePricingStrategies

With the Composite pattern, we have made a group of multiple (and conflicting) pricing strategies look to the *Sale* object like a single pricing strategy. The composite object that contains the group also implements the *ISalePricingStrategy* interface. The more challenging (and interesting) part of this design problem is: When do we create these strategies?

A desirable design will start by creating a Composite that contains the present moment's store discount policy (which could be set to 0% discount if none is active), such as some *PercentageDiscountPricingStrategy*. Then, if at a later step in the scenario, another pricing strategy is discovered to also apply (such as senior discount), it will be easy to add it to the composite, using the inherited *CompositePricingStrategy.add* method.

There are three points in the scenario where pricing strategies may be added to the composite:

1. Current store-defined discount, added when the sale is created.
2. Customer type discount, added when the customer type is communicated to the POS.
3. Product type discount (if bought Darjeeling tea, 15% off the overall sale), added when the product is entered to the sale.


:Register
«singleton»
:PricingStrategyFactory
make
NewSale()
create()
:Sale
ISalePricingStrategy
ps :=
getSale
PricingStrategy()
create()
ps :CompositeBestForCustomer
PricingStrategy
ISalePricingStrategy
create(percent)
s : PercentageDiscount
PricingStrategy
add(s)

Facade

Problem:

- 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 subsysten or the implementation of the subsystem may change.

Solution:

- Define a single point of contact to the subsystem – a façade object that wraps the subsystem. This façade object presents a single unified interface and is responsible for collaborating with the subsystem components

Façade example

- Define a “rule engine” subsystem for the POS application. The implementation of this is unknown. It will be responsible for evaluating a set of rules against an operation (like adding an item to the sale) and indicating if any of the rules are violated, and the operation is invalid.
- The façade object to the rule subsystem is called POSRuleEngineFacade. Calls to this façade are made at the points where pluggable rules need to be invoked

Facade

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

               // call to the Façade
     if (POSRuleEngineFacade.getInstance().isInvalid( sli, this ) )
        return;

     lineItems.add( sli );
}
//    …
} // end of class
```

Model View Controller

Model View Controller *(MVC paradigm)*

- *Used to build user interfaces*
  - *Smalltalk-80 (Alan Kay at Xerox Parc)*
- *Based on three types of objects:*

  - Model: domain objects

  - View: user interface/display objects

  - Controller*: defines how the interface responds to user input*
- *Decoupling view from the model*

Creational: Abstract Factory; Builder; Prototype; Singleton

Structural: Adapter; Bridge; Composite; Decorator; Façade; Flyweight; Proxy; Chain of Responsibility; Command

Behavioural:

Chain of Responsibility; Command; Iterator; Mediator; Memento ; Observer; State; Strategy; Visitor