More GoF Design Patterns: Composite, Façade, and Observer

Curt Clifton

Rose-Hulman Institute of Technology

GoF Pattern Taxonomy

- **Behavioral**
  - Interpreter
  - Template Method
  - Chain of Responsibility
  - Command
  - Iterator
  - Mediator
  - Memento
  - **Observer** ←
  - State
  - *Strategy*
  - Visitor
- **Creational**
  - *Factory Method*
  - Abstract Factory
  - Builder
  - Prototype
  - *Singleton*
- **Structural**
  - *Adapter*
  - Bridge
  - **Composite** ←
  - Decorator
  - **Façade** ←
  - Flyweight
  - Proxy

Composite

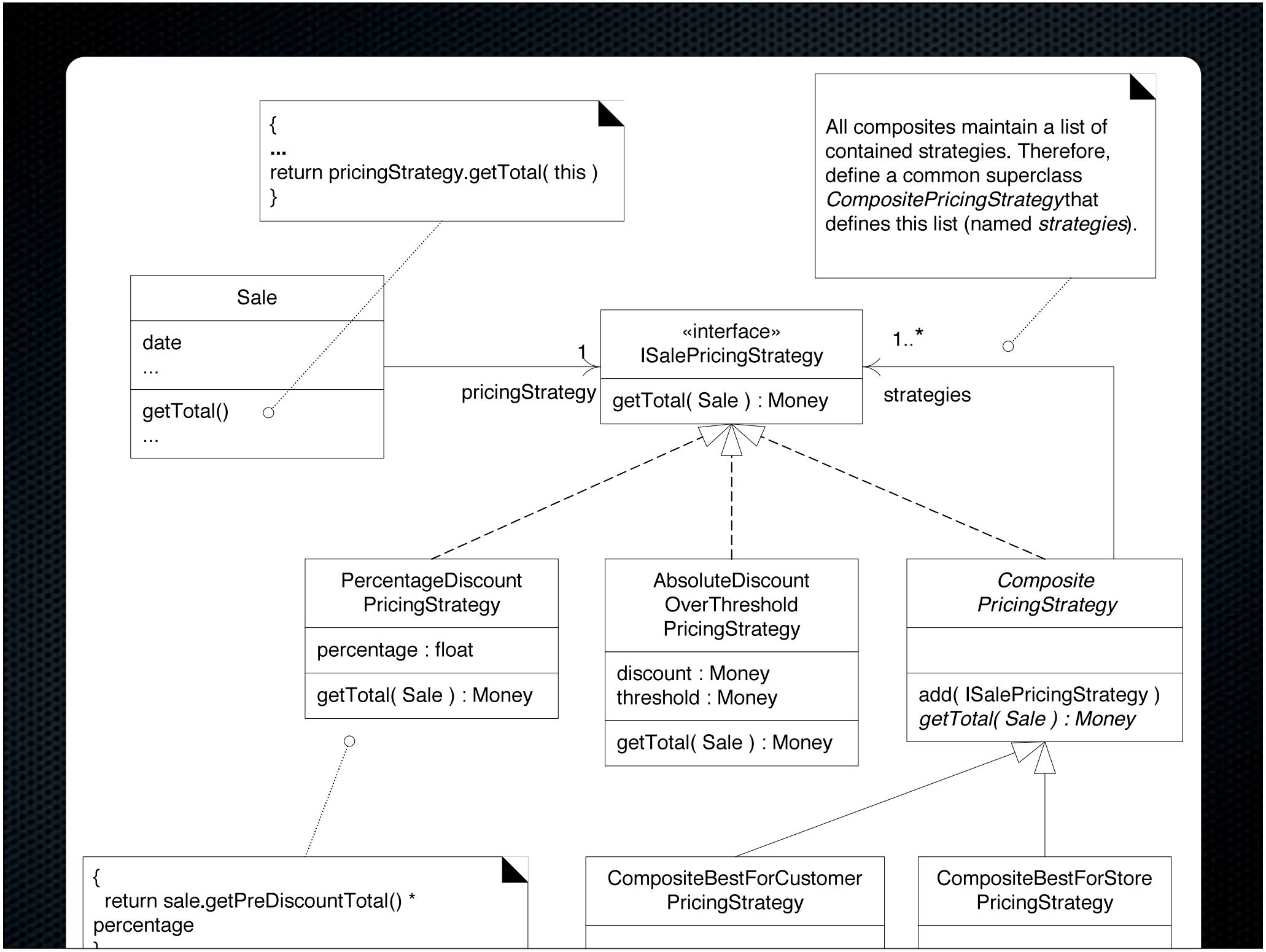
- How could we handle multiple, conflicting pricing policies?
- Such as…
  - 20% senior discount
  - Preferred customer discount, 15% off sales of $400
  - Manic Monday, $50 off purchases over $500
  - Buy 1 case of Darjeeling tea, get 15% off entire order

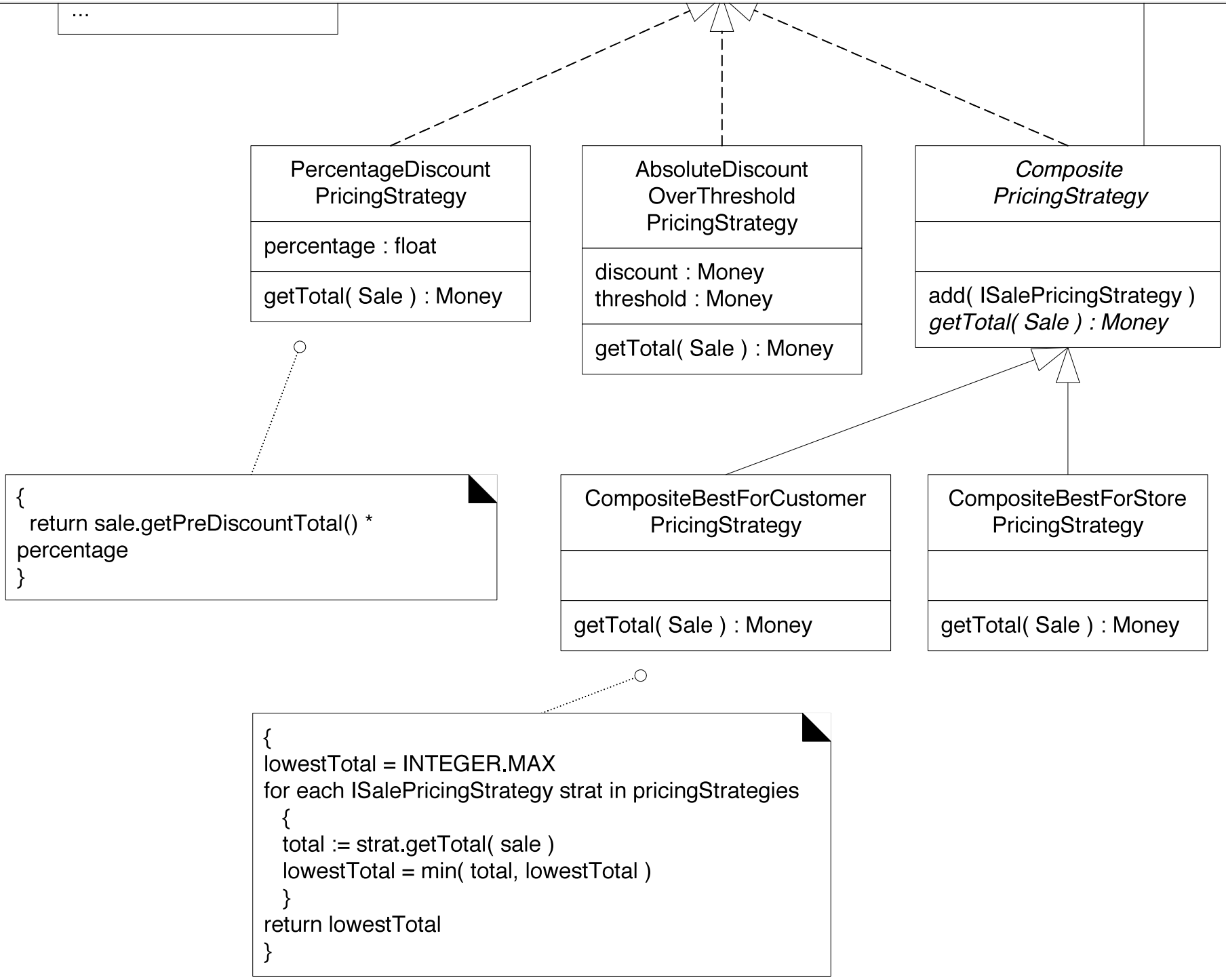
Composite Pattern

- **Problem**: How do we handle the situation where a group of objects can be combined but should still support the same polymorphic methods as any individual object in the group?
- **Solution**: Define a *composite* object that implements the same interface as the individual objects.

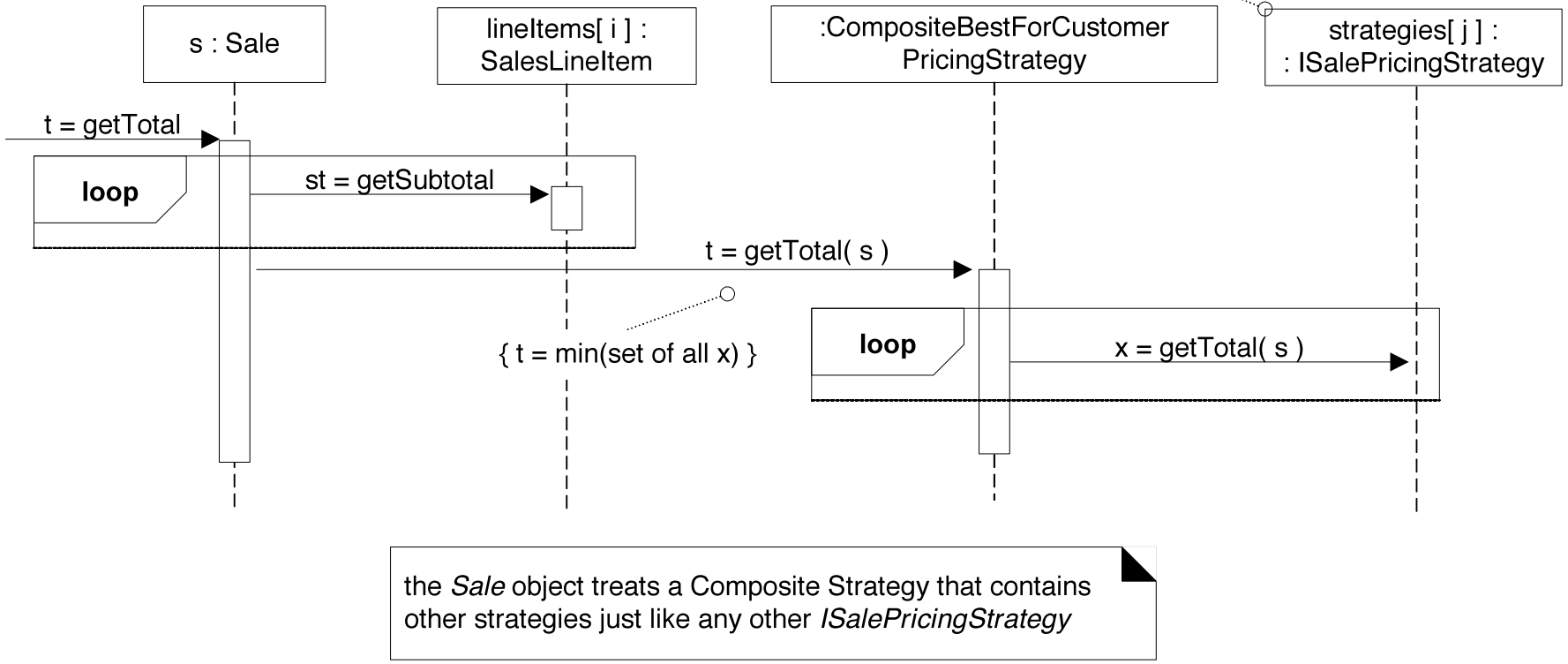
Example…

Q1,2


{
...
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
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
CompositeBestForStore PricingStrategy


...
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
}

Example Continued: Dynamic Use

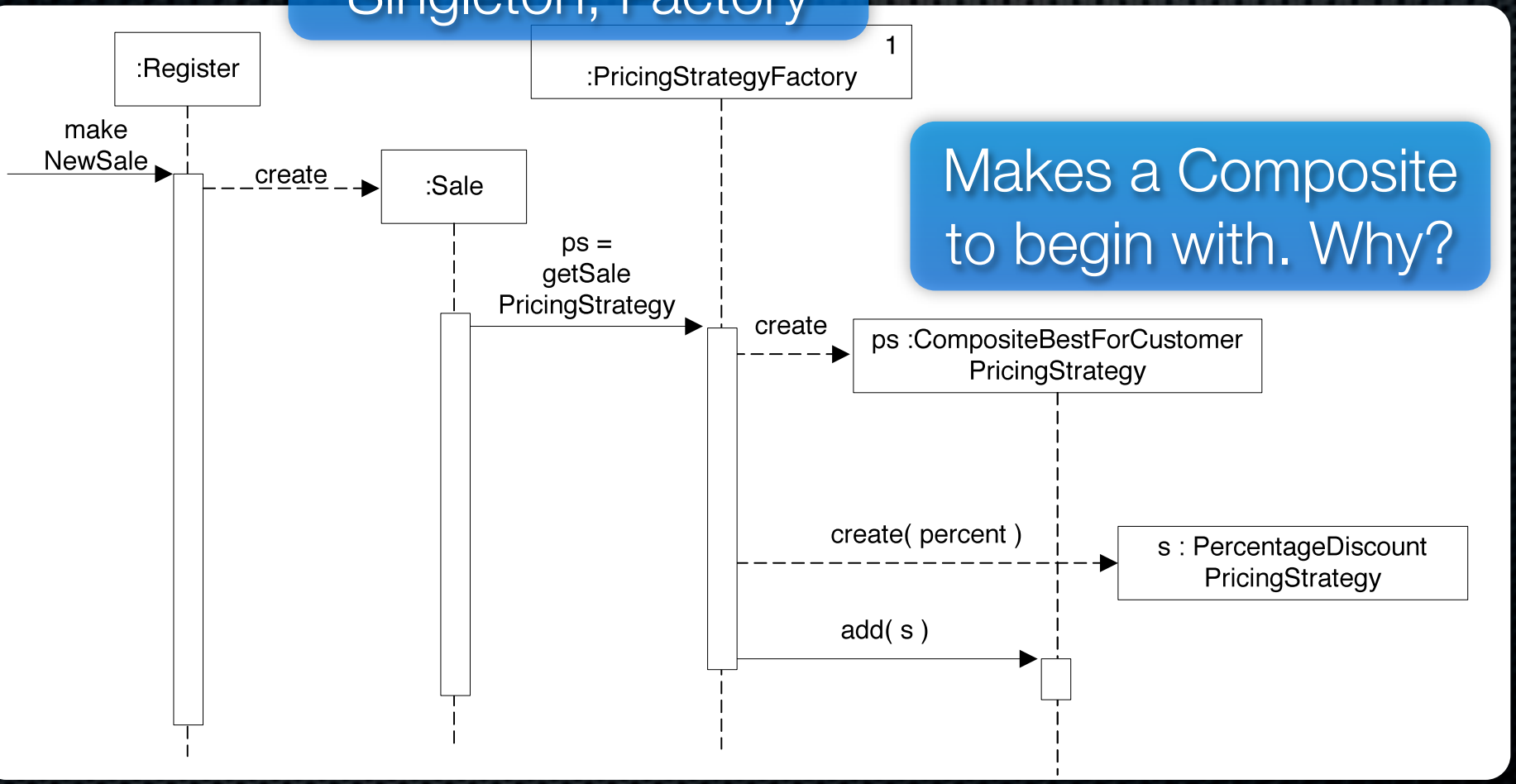


How do we build the Composite Strategy?

- Three places in example where new pricing strategies can be added:
  - When new sale is created, add store discount policy
  - When customer is identified, add customer-specific policy
  - When a product is added to the sale, add product-specific policy

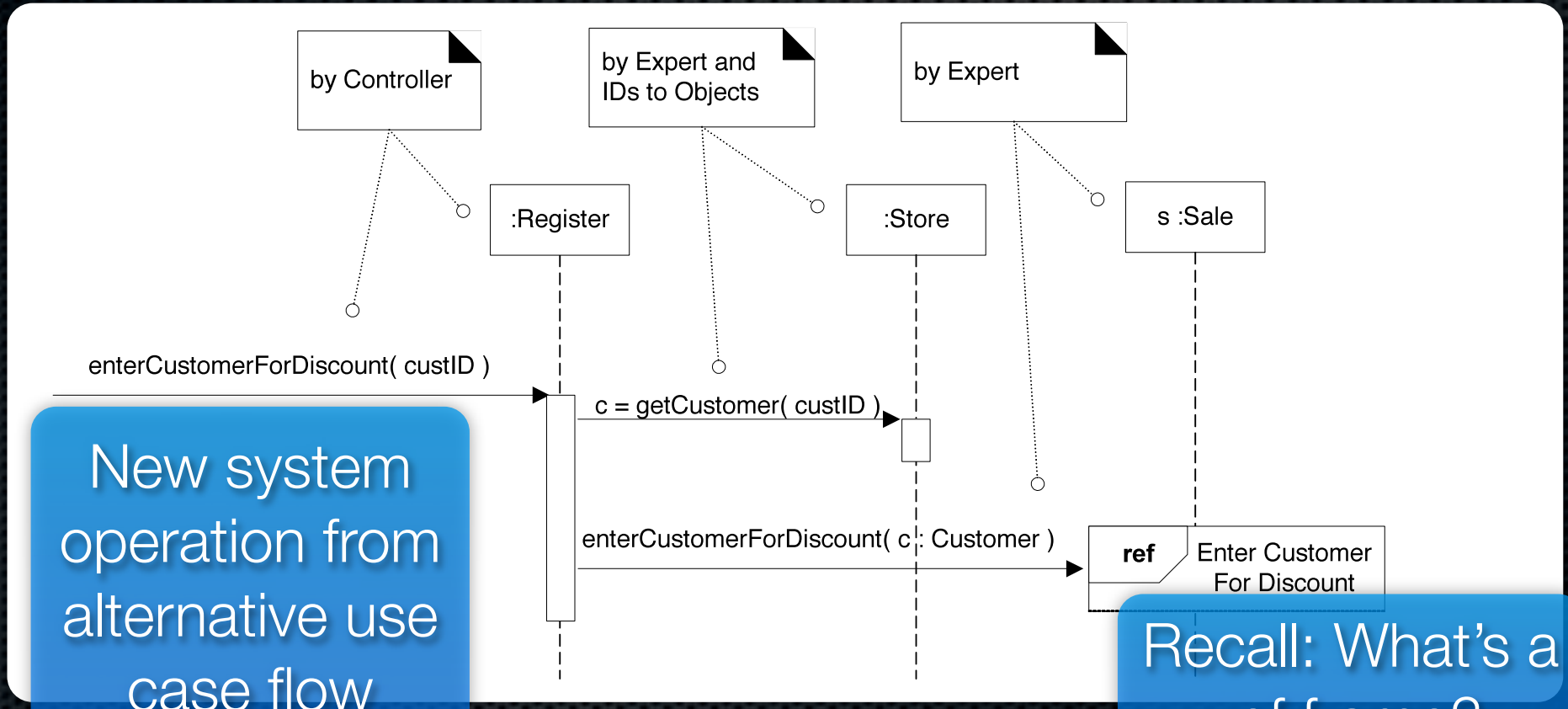
Adding Store Discount Policy

Singleton, Factory



Makes a Composite to begin with. Why?

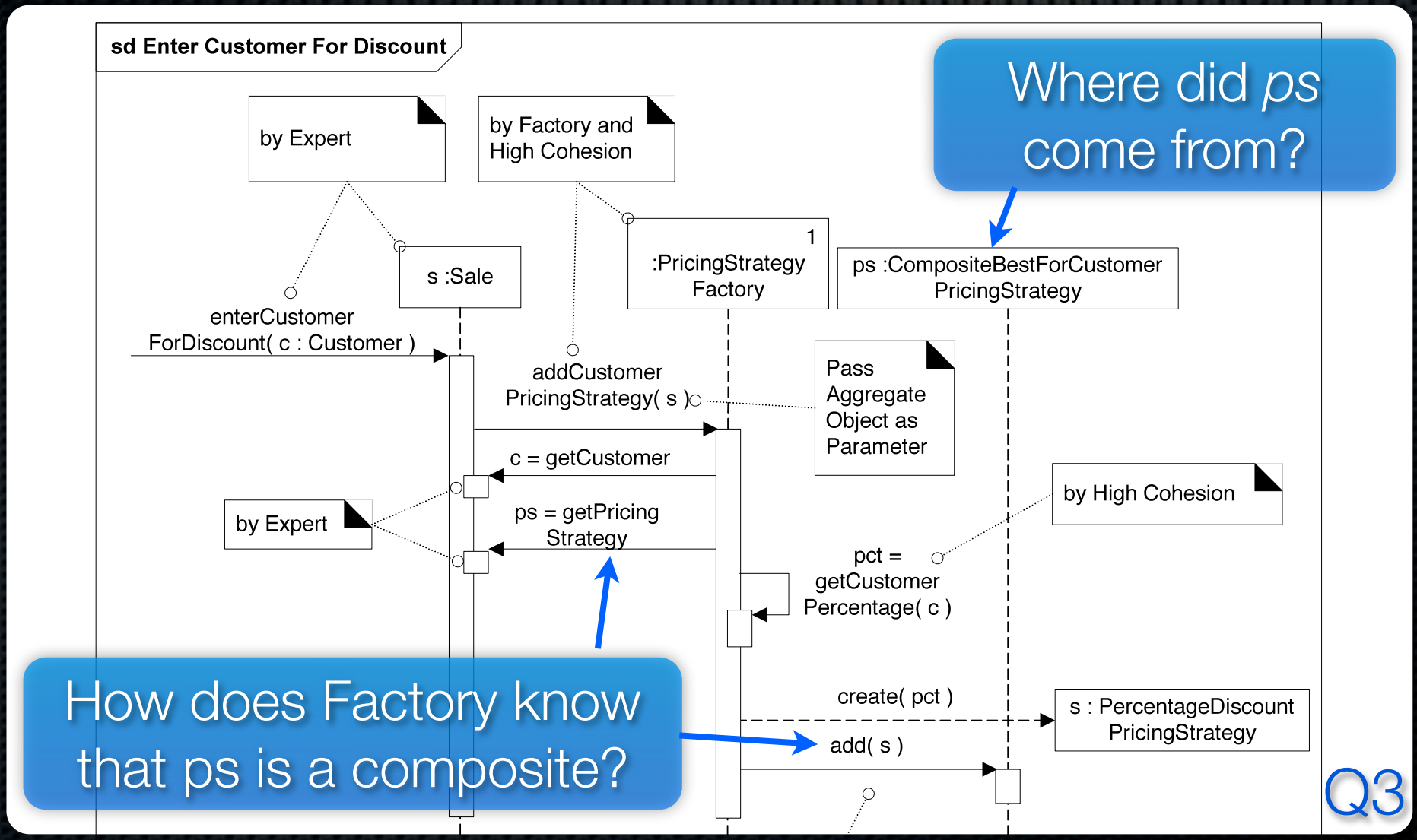
Adding Customer Specific Discount Policy



New system operation from alternative use case flow

Recall: What's a ref frame?

Adding Customer Specific Discount Policy (continued)



Q3

Applying Composite

Q4

Façade

More general than just Façade Controllers

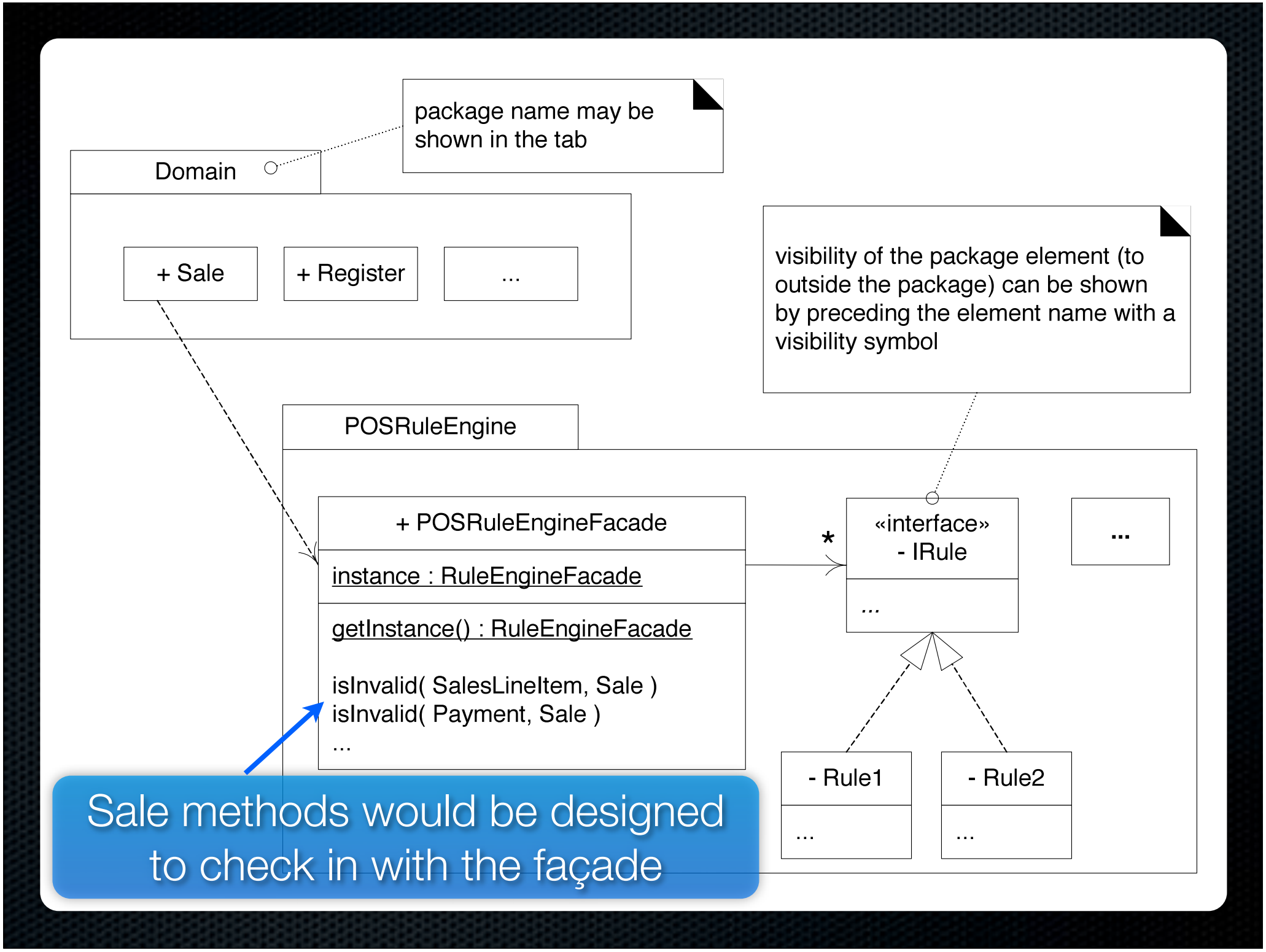
- NextGen POS needs *pluggable business rules*
- Assume rules will be able to disallow certain actions, such as…
  - Purchases with gift certificates must include just one item
  - Change returned on gift certificate purchase must be as another gift certificate
  - Allow charitable donation purchases, but max. of $250 and only with manager logged-in

Some Conceivable Implementations

- Strategy pattern
- Open-source rule interpreter
- Commercial business rule engine

Façade

- **Problem**: How do we avoid coupling to a part of the system whose design is subject to substantial change?
- **Solution**: Define a single point of contact to the variable part of the system—a *façade object* that wraps the subsystem.


package name may be shown in the tab
Domain
+ Sale
+ Register
...
visibility of the package element (to outside the package) can be shown by preceding the element name with a visibility symbol
POSRuleEngine
+ POSRuleEngineFacade
instance : RuleEngineFacade
getInstance() : RuleEngineFacade
isInvalid(SalesLineItem, Sale)
isInvalid(Payment, Sale)
...
*
«interface»
- IRule
...
...
- Rule1
...
- Rule2
...
Sale methods would be designed to check in with the façade

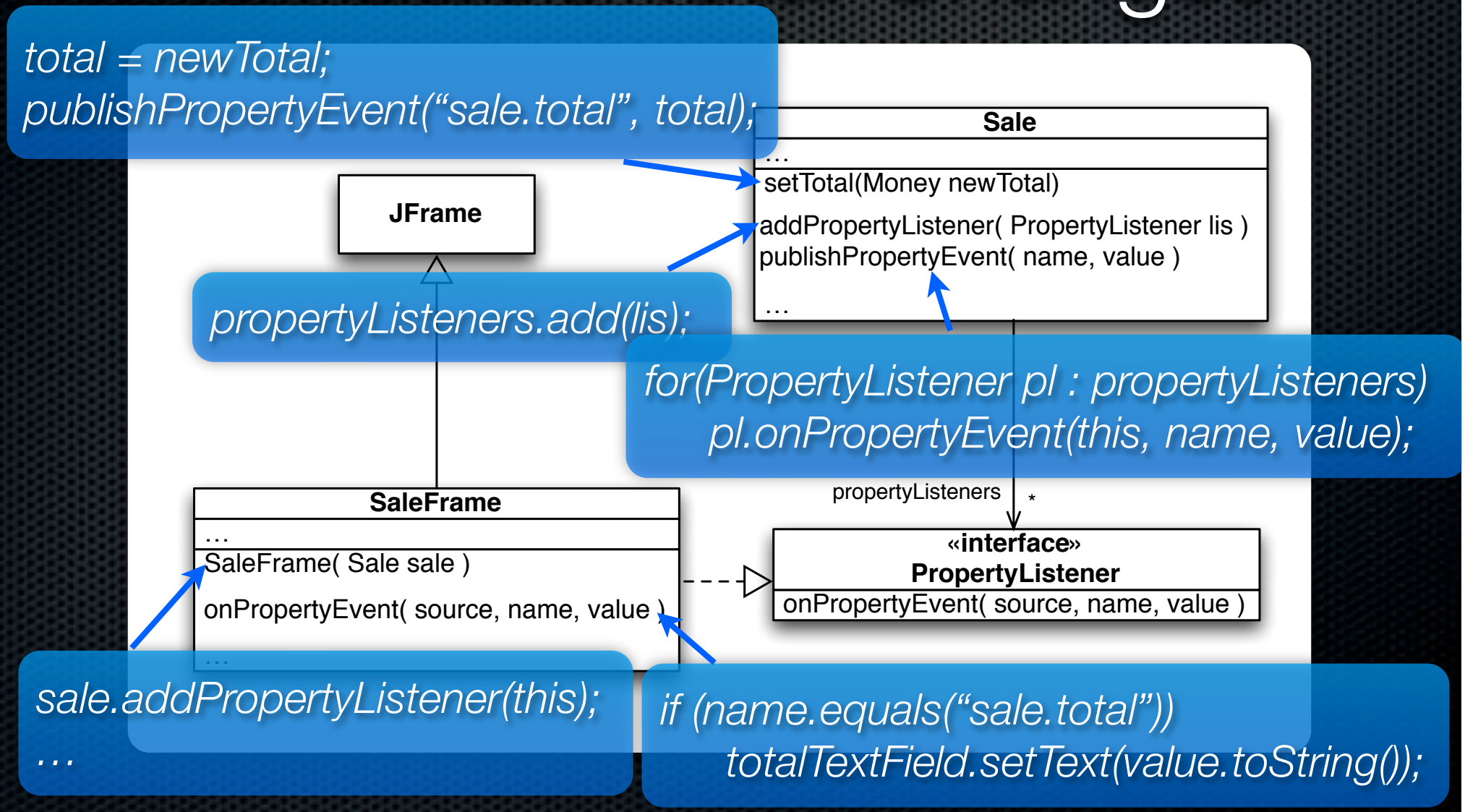
Observer

- How do we refresh the GUI display when the domain layer changes *without coupling the domain layer back to the UI layer*?

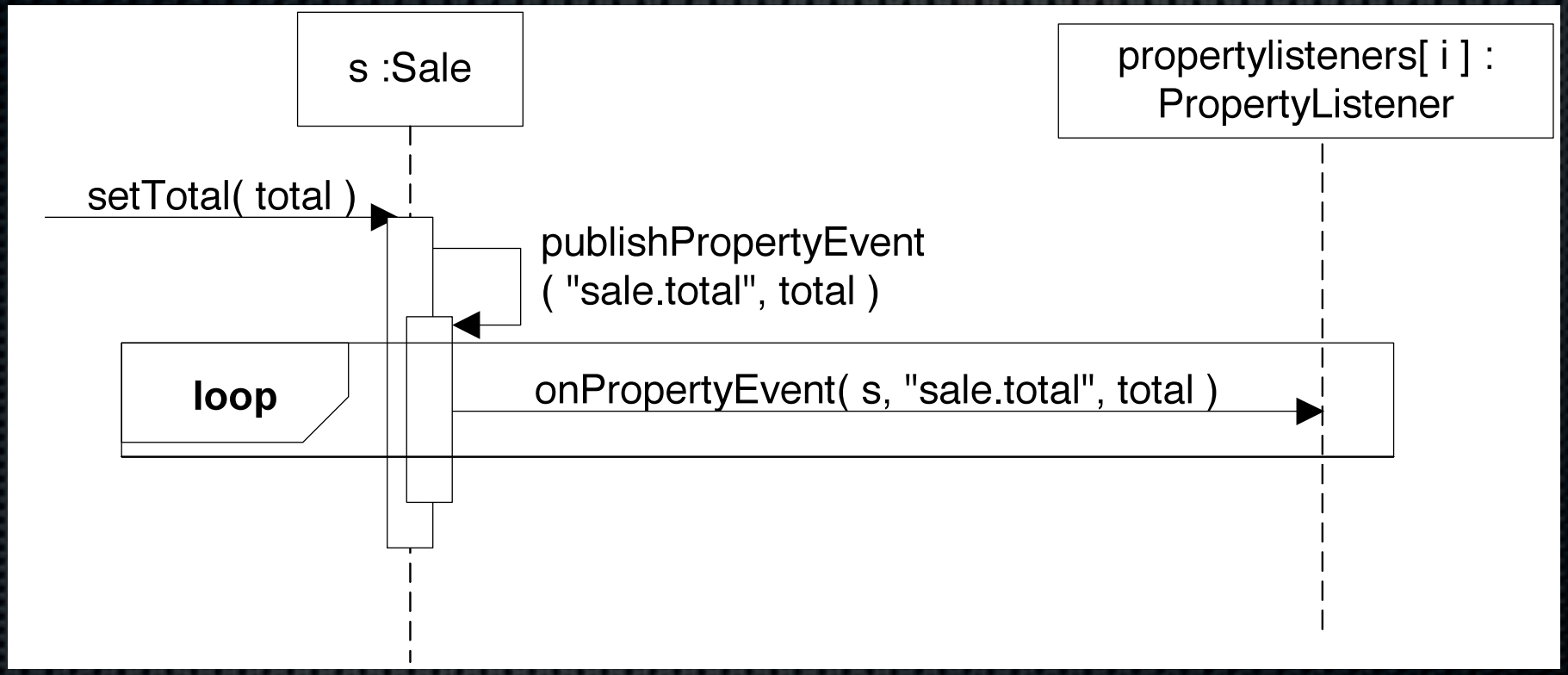
Observer (aka Publish-Subscribe)

- **Problem**: Suppose some *subscriber* objects want to be informed about events or state changes for some *publisher* object. How do we achieve this while maintaining low coupling from the publisher to the subscribers?
- **Solution**: Define an subscriber interface that the subscriber objects can implement. Subscribers register with the publisher object. The publisher sends notifications to all its subscribers.

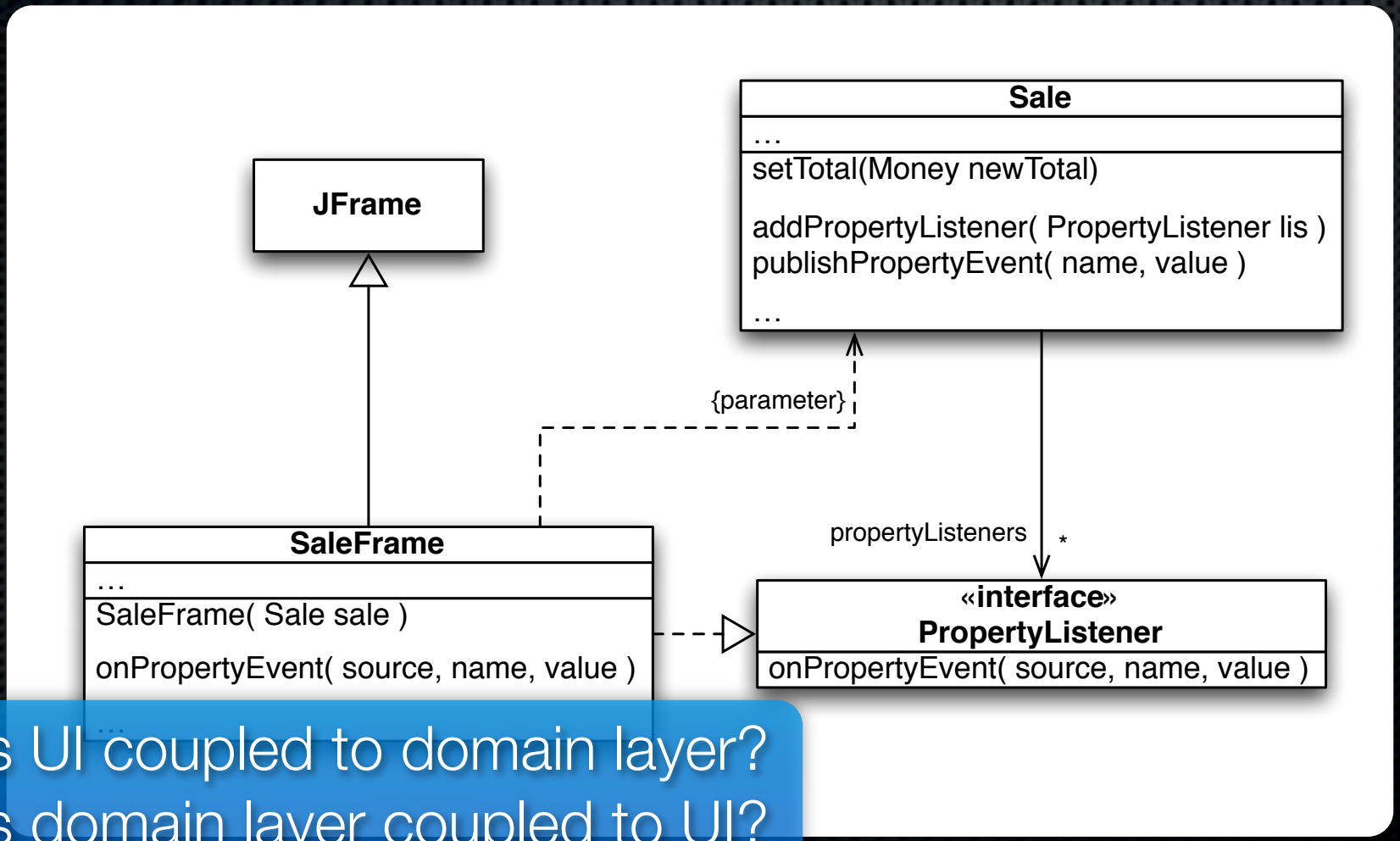
Example: Update SaleFrame when Sale's Total Changes



Example: Update SaleFrame when Sale's Total Changes



Example: Update SaleFrame when Sale's Total Changes



Is UI coupled to domain layer?
Is domain layer coupled to UI?

Q6,7

Observer: Not just for GUIs watching domain layer…

- GUI widget event handling
- Example:
  JButton startButton = new JButton("Start");
  startButton.addActionListener(new Starter());
- Publisher: *startButton*
- Subscriber: *Starter* instance

GoF Pattern Taxonomy

- **Behavioral**
  - Interpreter
  - Template Method
  - Chain of Responsibility
  - Command
  - Iterator
  - Mediator
  - Memento
  - **Observer**
  - State
  - **Strategy**
  - Visitor
- **Creational**
  - **Factory Method**
  - Abstract Factory
  - Builder
  - Prototype
  - **Singleton**
- **Structural**
  - **Adapter**
  - Bridge
  - **Composite**
  - Decorator
  - **Façade**
  - Flyweight
  - Proxy

Q9