SWEN30006

Software Modelling and Design

Applying GoF Design Patterns

(Part 3: Façade, Observer and Decorator)

Textbook: Larman Chapter 26

Lecturer: Peter Eze

Beware of bugs in the above code; I have o
nly proved it correct, not tried it.

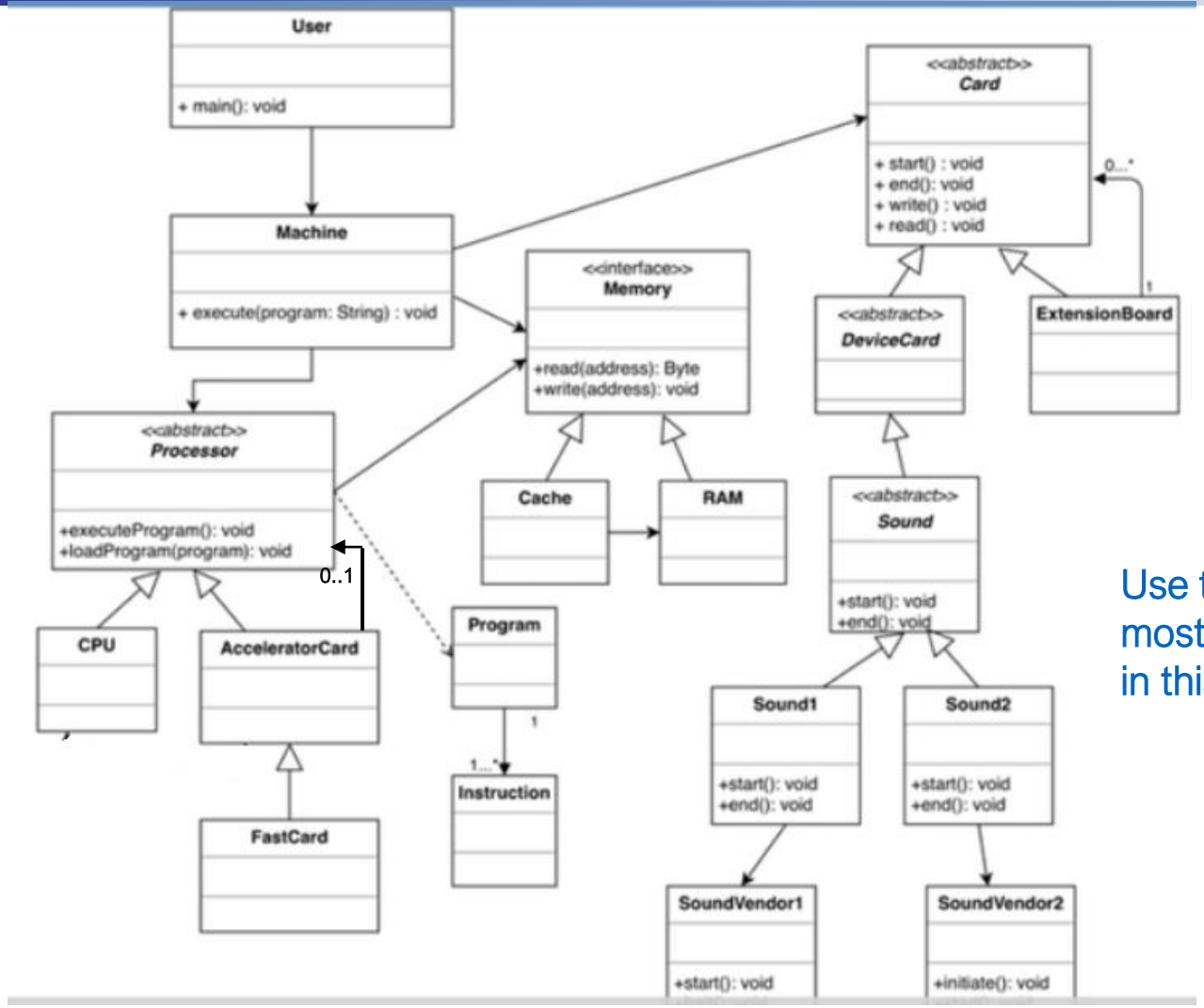
—Donald Knuth

Objectives

On completion of this topic you should be able to:

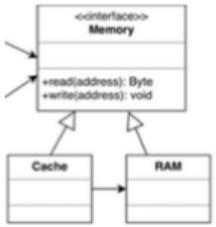
- Apply some GoF design patterns
  - Adapter
  - Factory (not GoF)
  - Singleton
  - Strategy
  - Composite
  - Façade
  - Decorator
  - Observer
- Recognise GRASP principles as a generalization of other design patterns.

Multiple Patterns in a Design

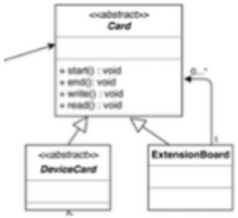


Use this diagram for most of the exercises in this lecture

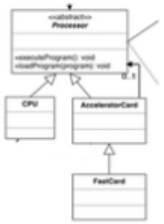
Which of the following diagrams depicts a composite pattern?



A



B



C

Problem 3: Pluggable Business Rules

- Different companies who wish to purchase the NextGen POS would like to customise its behaviour slightly.
- *E.g. to invalidate an action:*
  - **Paid by a Gift card**
    - Allow one item to be purchased -> Invalidate subsequent *EnterItem* operations
    - Invalidate request for change as cash or store a/c credit.
  - **Charitable donation (by store) sale**
    - If cashier is not manager, *CreateCharitableSale* invalidated
    - *EnterItem* operation for items over $250 invalidated.

Analysis

- This customisation should have *low impact* on the existing software components
- A "rule engine" subsystem, whose specific implementation is not yet known. It may be implemented with
  - the Strategy pattern; or
  - Free open-source rule interpreters; or
  - Commercial rule interpreters; or
  - Other solutions.

Façade (GoF)

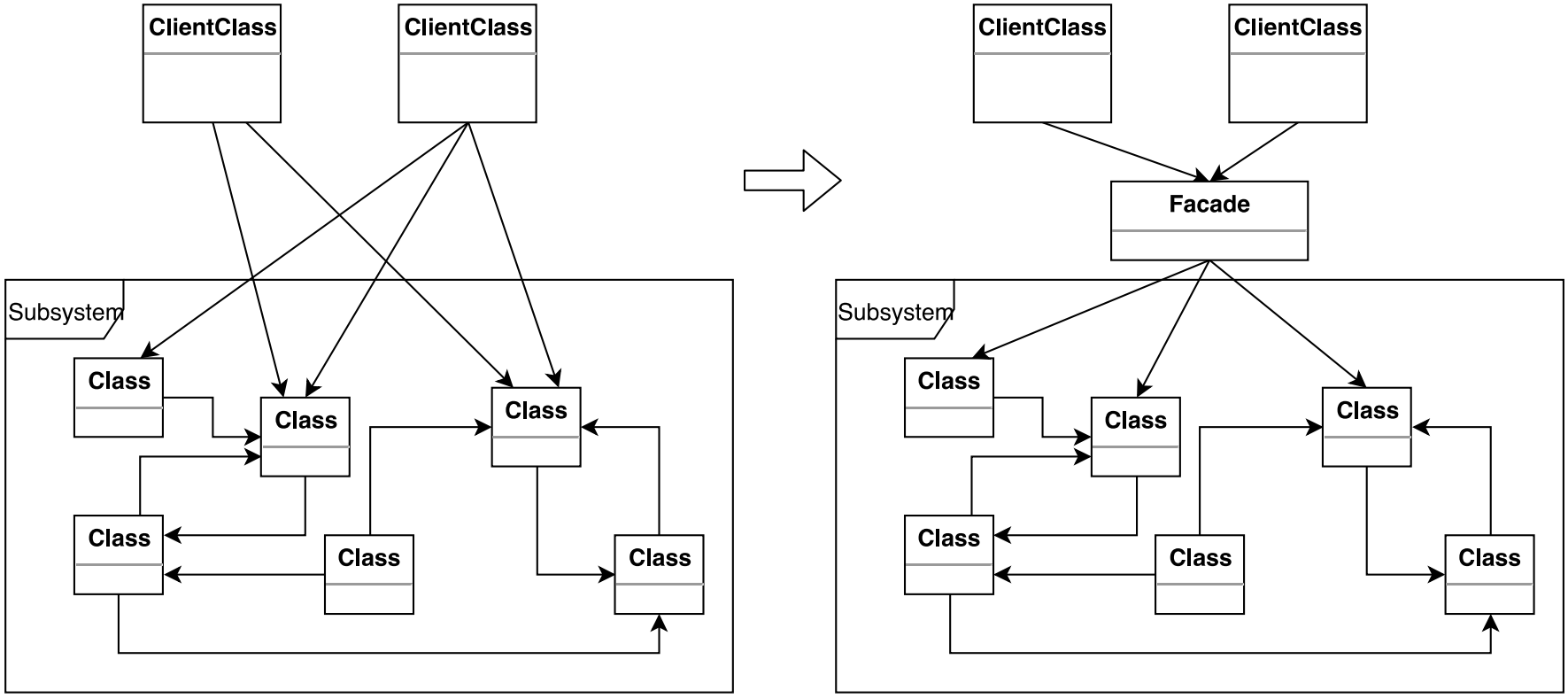
Problem:

- Require 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 objects in the subsystem, or the implementation of the subsystem may change. What to do?

Solution (advice):

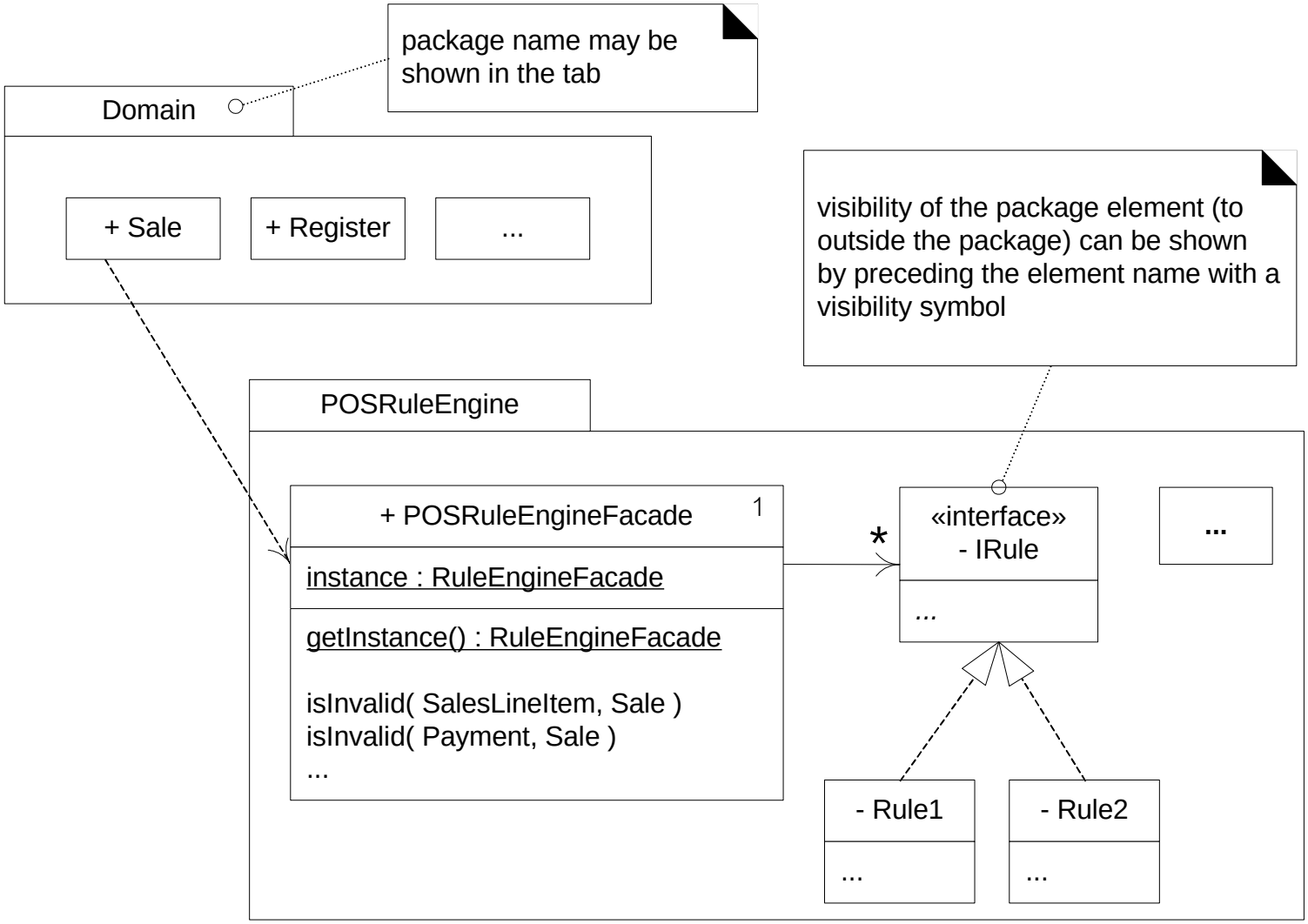
- Define a single point of contact to the subsystem—a ***facade*** object that wraps the subsystem. This facade object presents a single unified interface and is responsible for collaborating with the subsystem components.

Façade: Generalised Structure



Without Facade **With Facade**

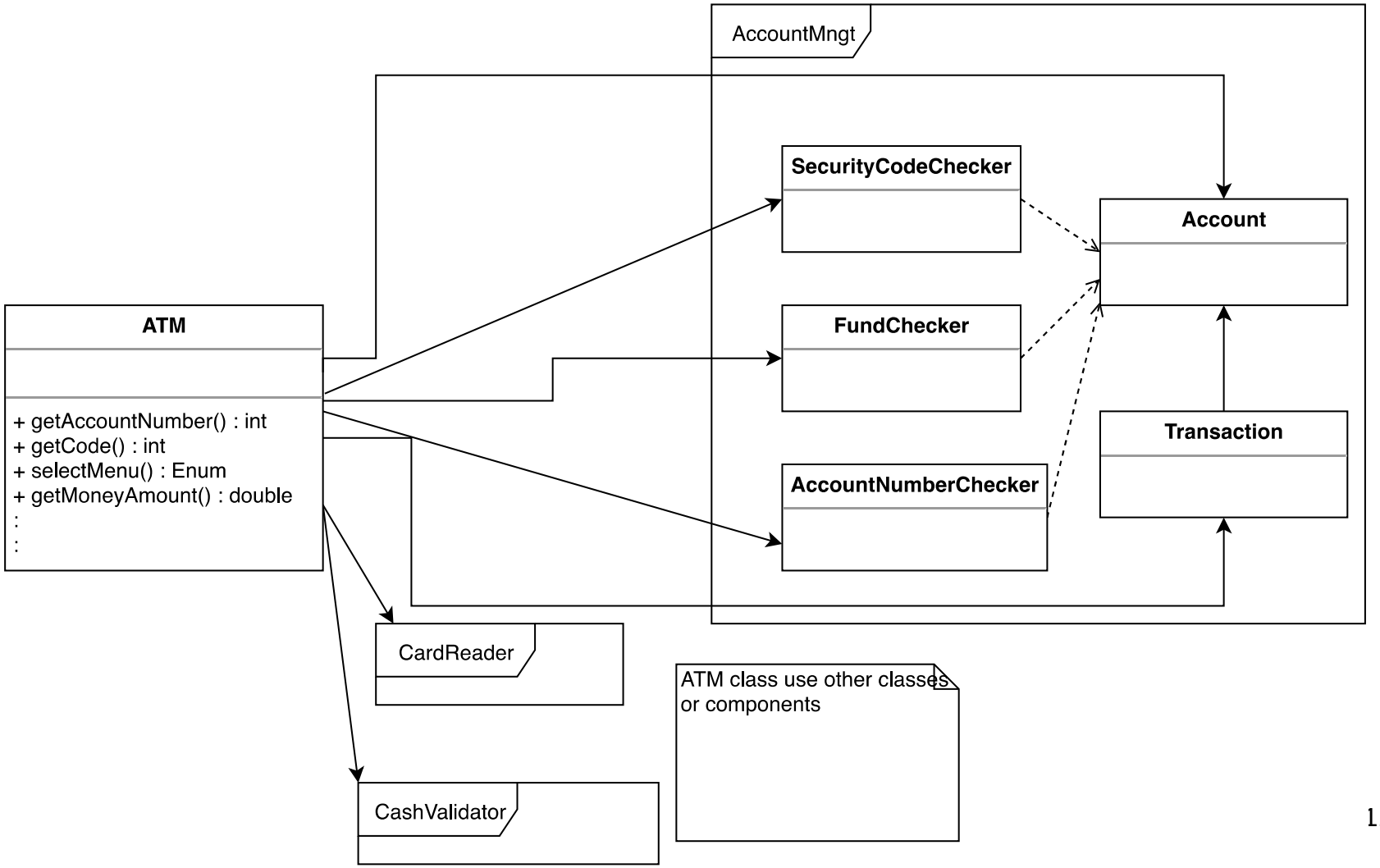
UML Package Diagram: Facade



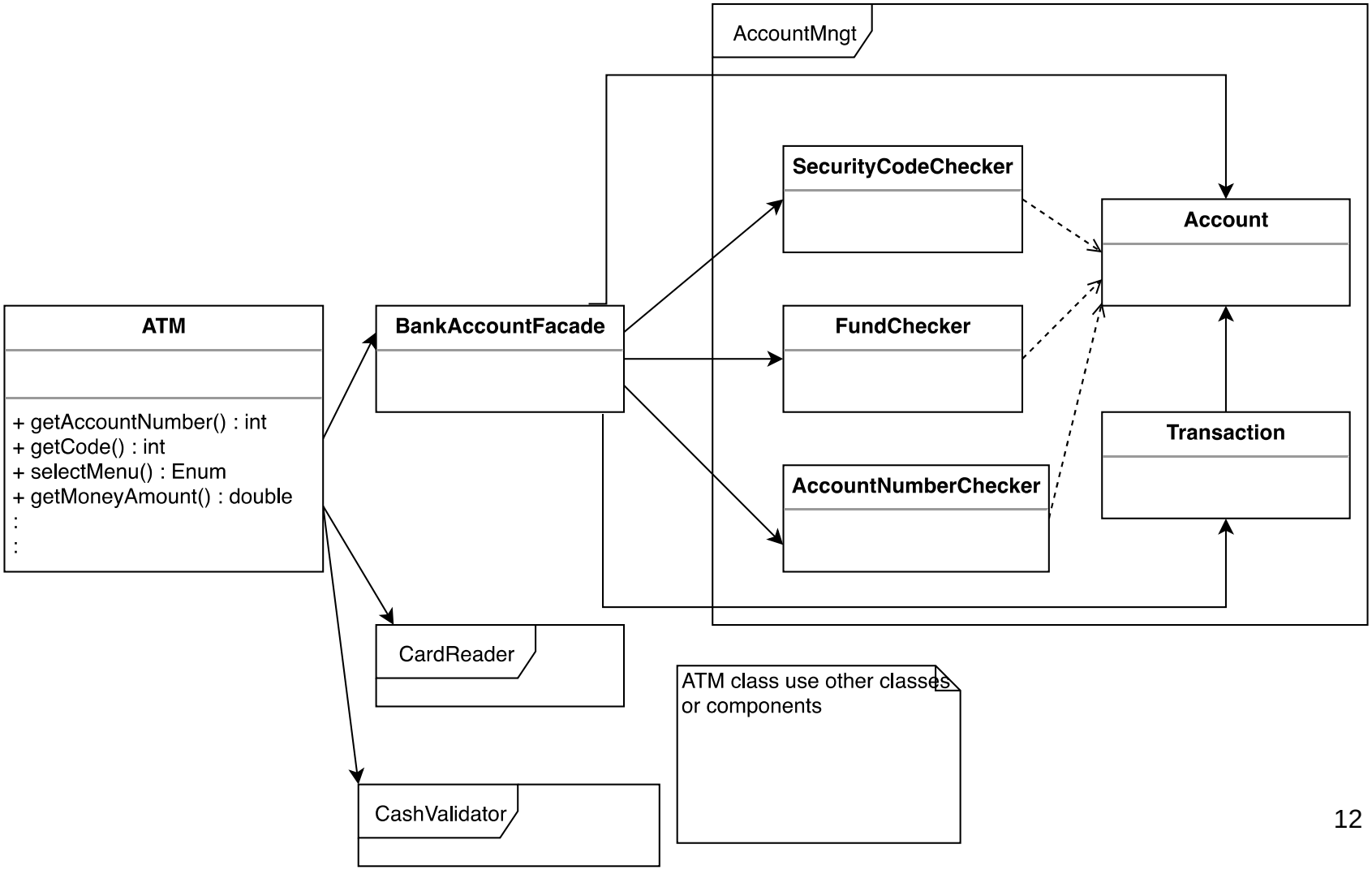
Using the Facade

```
public class Sale {
   public void makeLineItem( ProductDescription desc, int quantity )
   {
      SalesLineItem sli = new SalesLineItem( desc, quantity );
      // call to the Facade
      if ( POSRuleEngineFacade.getInstance().isInvalid( sli, this ) )
         return;
      lineItems.add( sli );
   }
// ...
} // end of class
```

Example: ATM (w/o Façade)



Example: ATM w/ Facade



Example: MyATM (Main function)

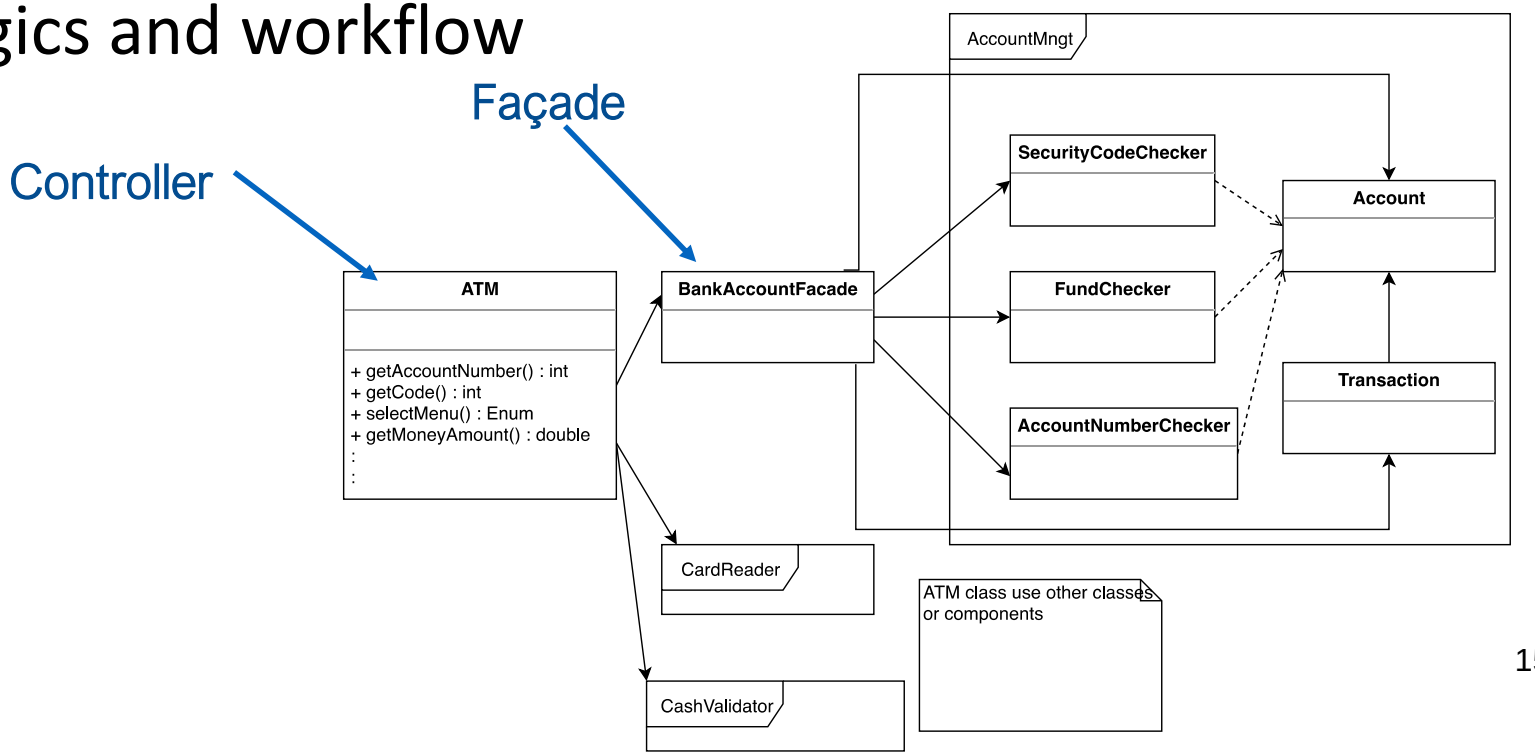
```
BankAccountFacade accessingAccount = new
                BankAccountFacade(accountNumber, code);
boolean end = false;
while(!end) {
    switch(selectMenu()) {
    case Deposit:
       double cashToDeposit = getMoneyAmount();
       accessingAccount.depositCash(cashToDeposit);
       break;
    case Withdraw:
       double cashToGet = getMoneyAmount();
       accessingAccount.withdrawCash(cashToGet);
       break;
    case Exit:
       end = true;
       break;
     }
}
```

Example: BankAccountFacade

```
public class BankAccountFacade {
  private int accountNumber;
  private int securityCode;
  private AccountNumberCheck acctChecker;
  private SecurityCodeCheck codeChecker;
  private FundsCheck fundChecker;
  private Account account;
  private Transaction transaction;

  public BankAccountFacade(int newAcctNum, int newSecCode){
     //Instantiate related objects
  }
  public void withdrawCash(double cashToGet){
   //Check account number, security code, available fund
   //If all is valid, decrease cash in account
  }
  public void depositCash(double cashToDeposit){
    //Check account number, security code
    //If all is valid, increase cash in account
  }
}
```

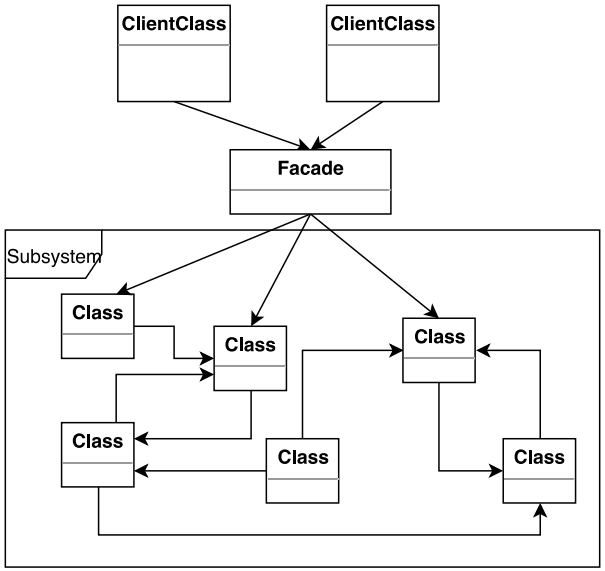
Façade VS GRASP Controller

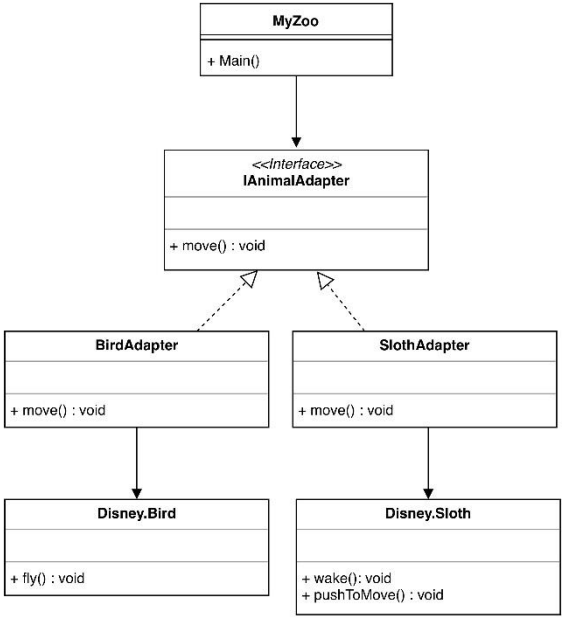
- **Façade** provides a simpler interface of a complex subsystem for a client class
- **Controller** handles user inputs based on business logics and workflow



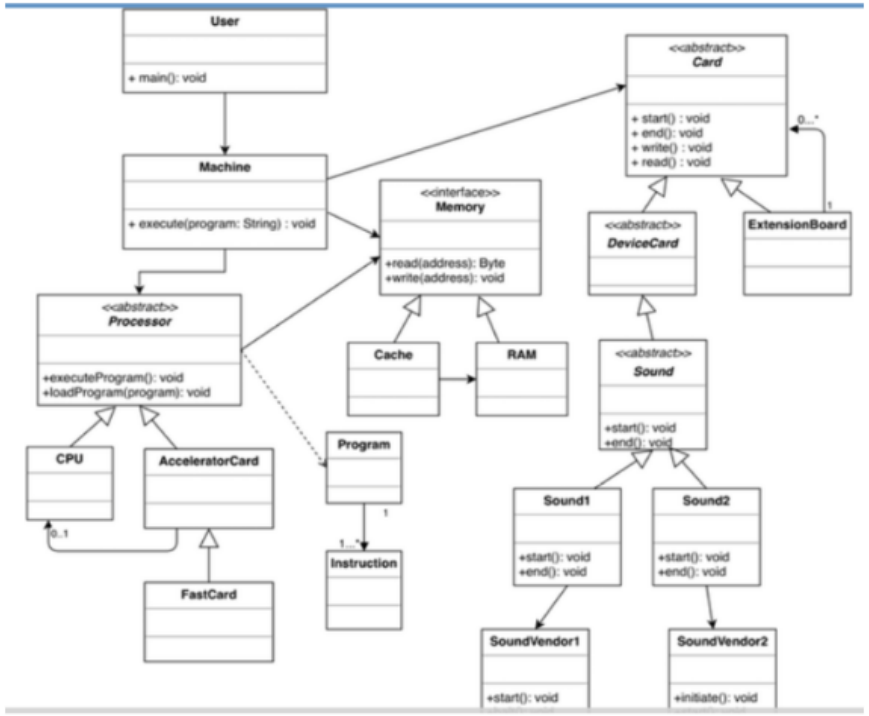
Façade VS Adapter

- **Façade** wraps access to a subsystem or system with a single object
- **Adaptor** wraps each API with varying interface to provide a single interface





Which of the classes can be considered as a facade?



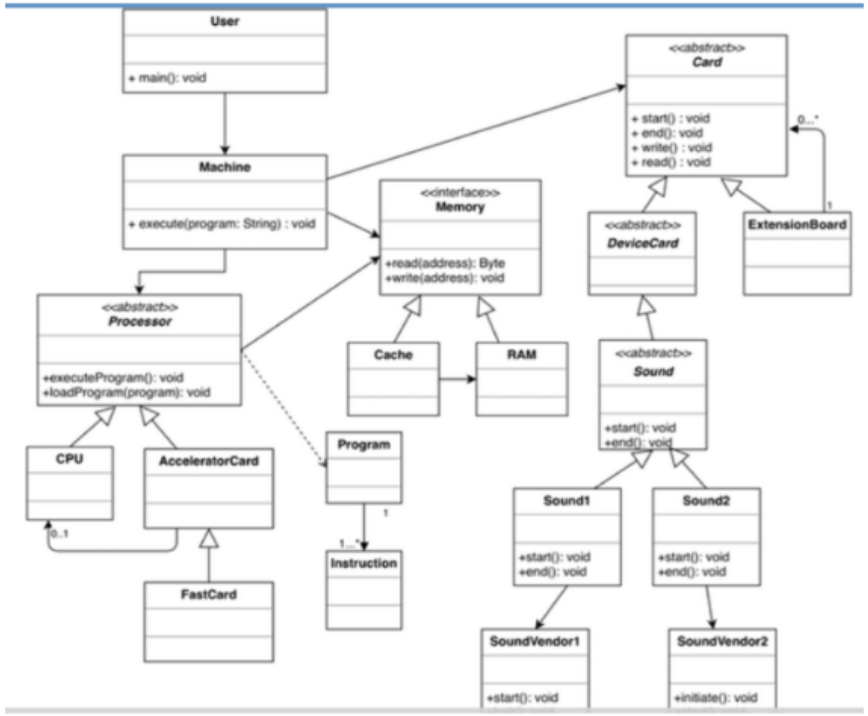
Card

Processor

Machine

Sound

Identify the adapter classes that form parts of an Adapter GoF Pattern in the diagram



CPU and AcceleratorCard

Sound1 and Sound2

DeviceCard and ExtensionCard

Cache and RAM

Problem 4: Dynamic Behavior at Run-time

- You want to add behavior or state to individual objects at run-time.
  - Irrespective of combination of number of features or behaviours added, you don't want to change the interface presented to the client.
  - Inheritance is not feasible because it is static and applies to an entire class.

Decorator (GoF)

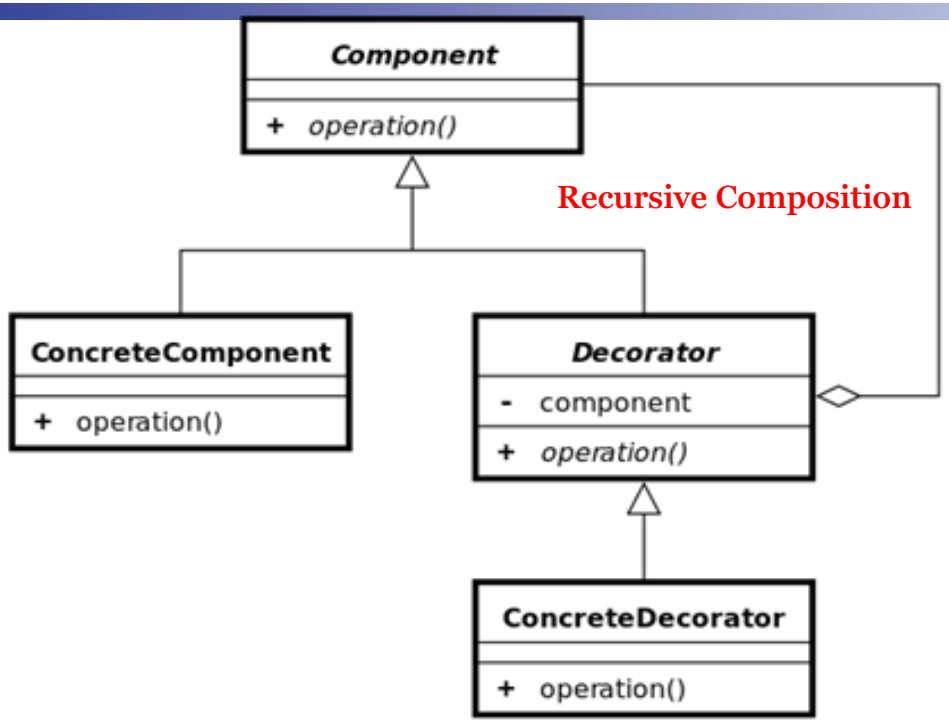
Problem:

- How to dynamically add behaviour or state to individual objects at run-time without changing the interface presented to the client. ***What to do?***

Solution (advice):

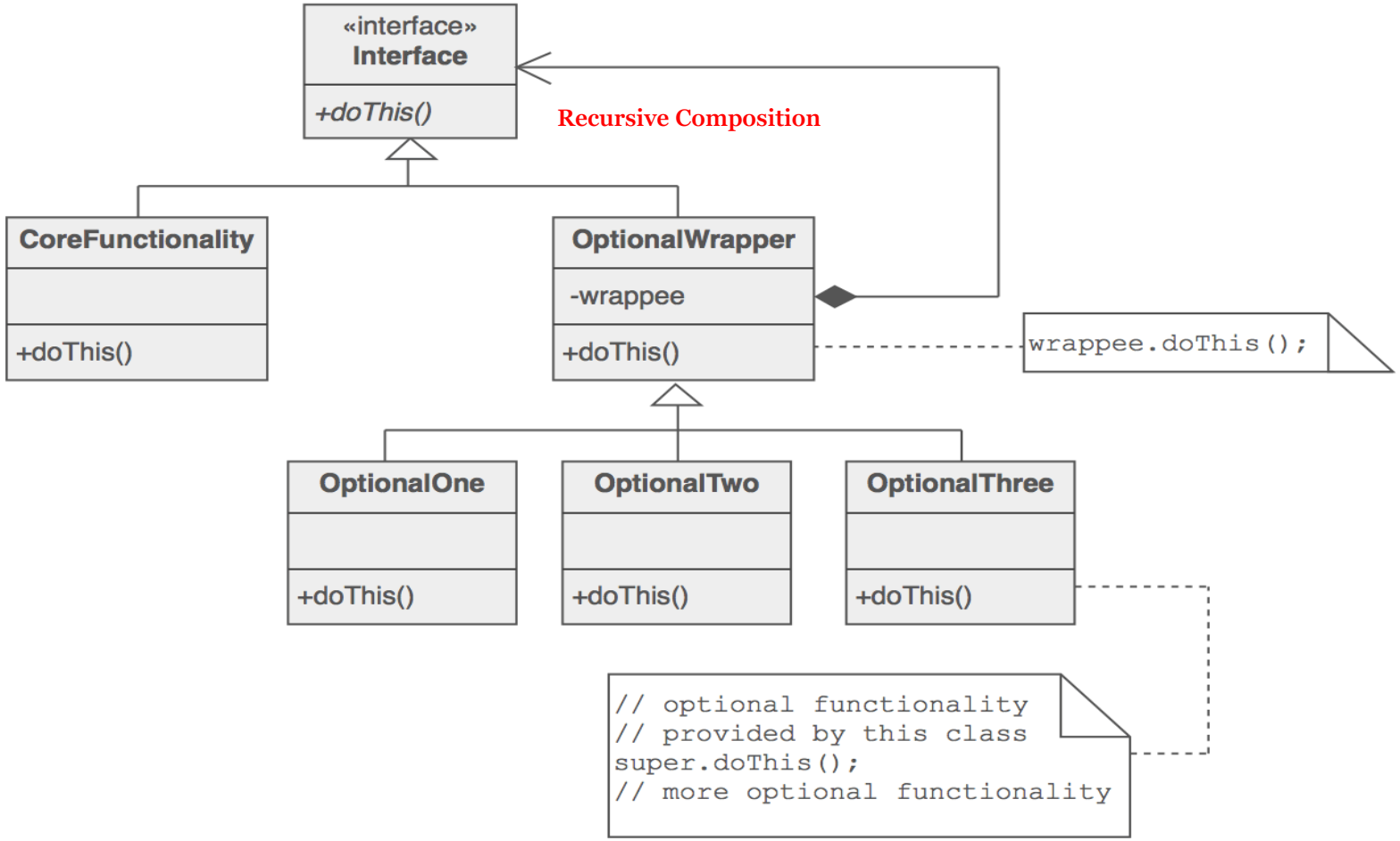
- The solution to this problem involves encapsulating the original concrete object inside an abstract wrapper interface. Then, let the **decorators** that contain the dynamic behaviours also inherit from this abstract interface. The interface will then use recursive composition to allow an unlimited number of decorator "layers" to be added to each core object.

Decorator: General Form



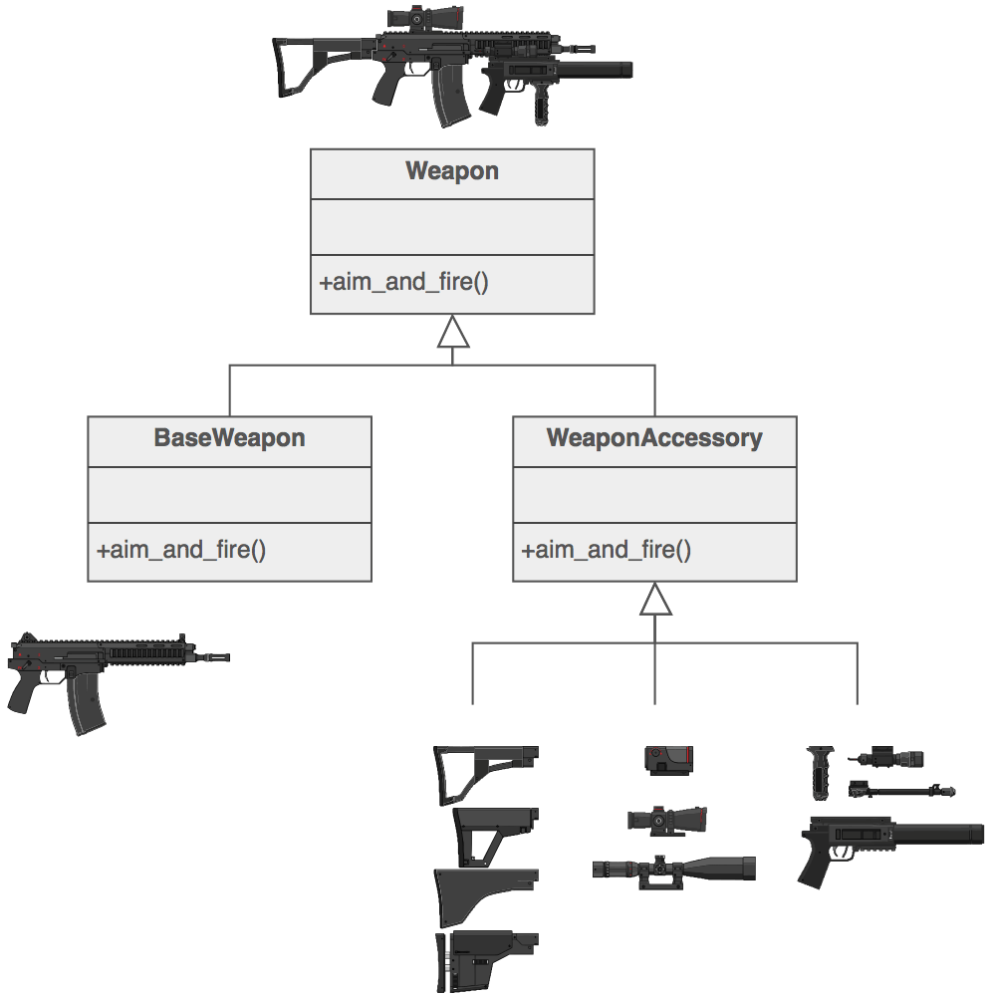
A client is always interested in ConcreteComponent.operation() but may or may not be interested in ConcreteDecorator.operation(). More optional ConcreteDecotrators can be added to provide more behaviours.

Decorator: Multiple "layers"



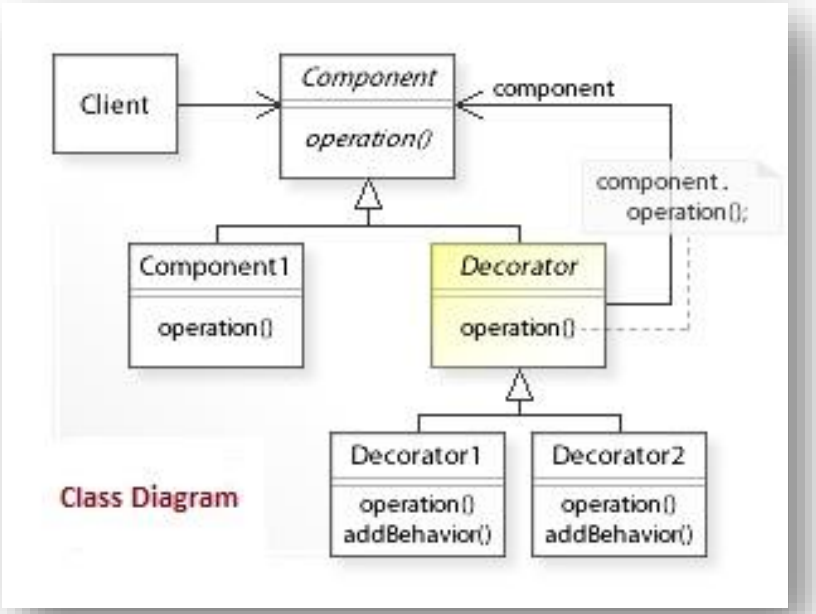
Source: https://sourcemaking.com/design_patterns/decorator

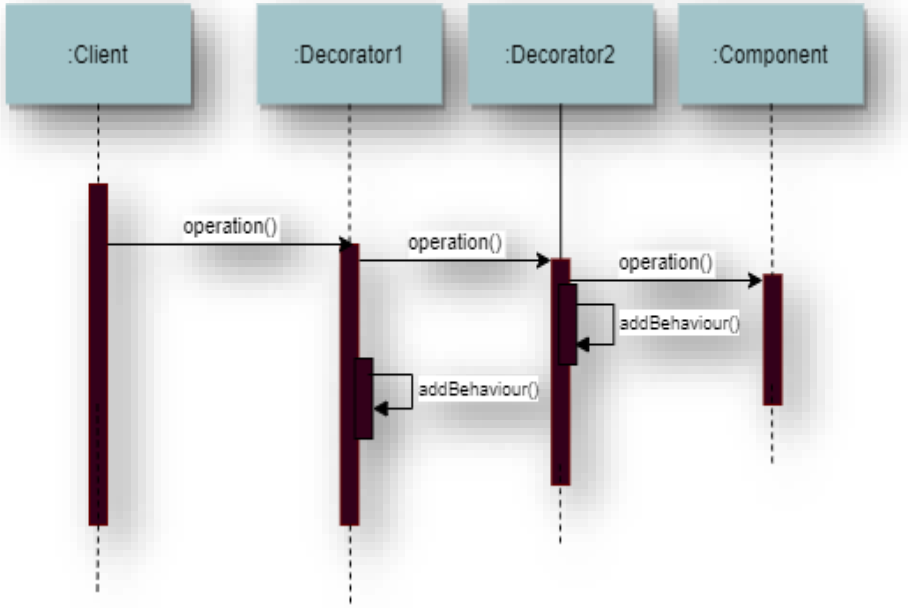
Decorator Example



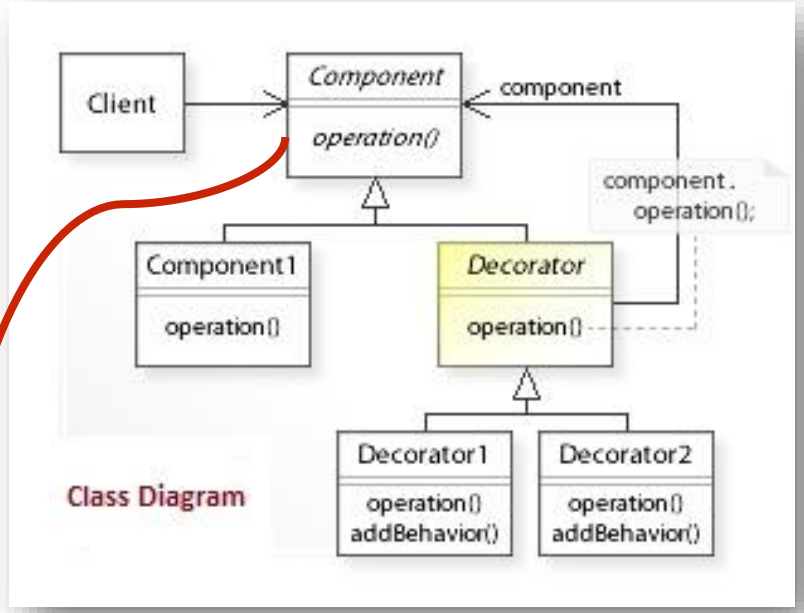
Source: https://sourcemaking.com/design_patterns/decorator

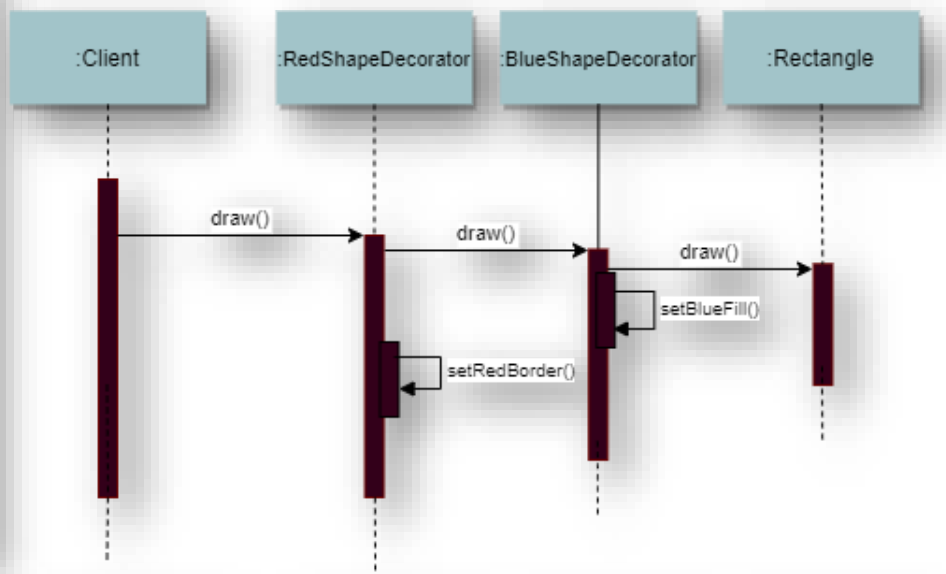
Collaboration and Sequencing





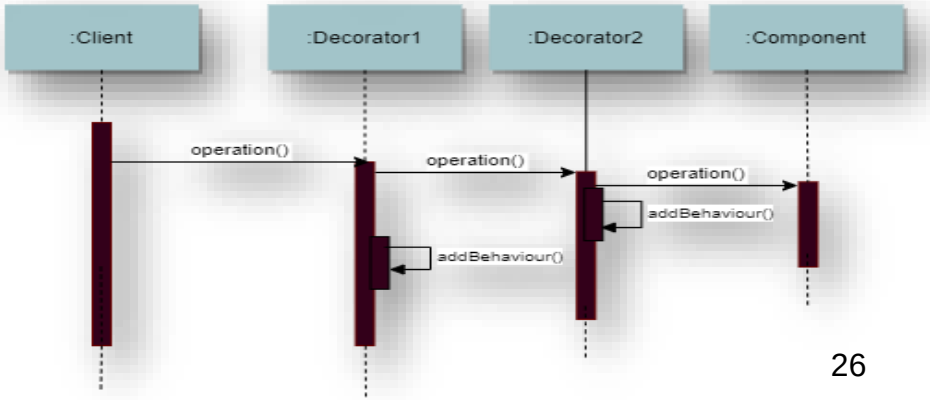
Example: Shape Decorator



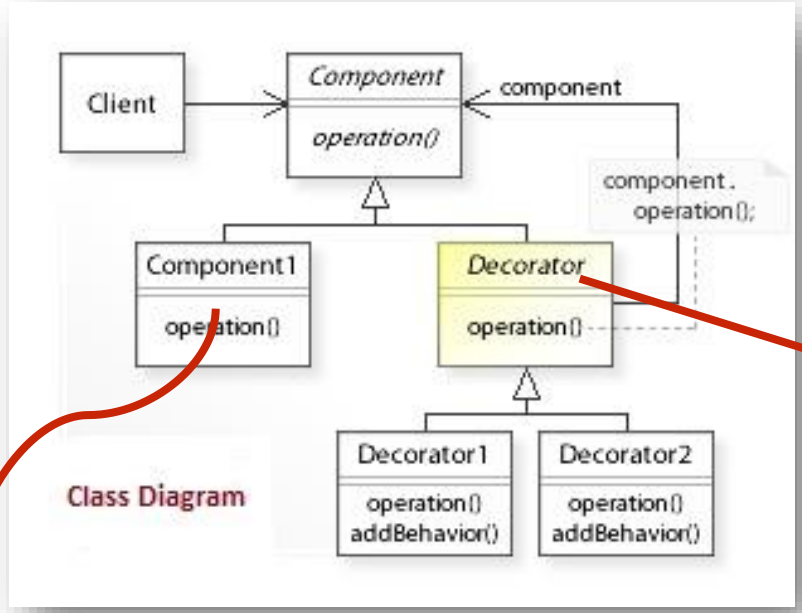


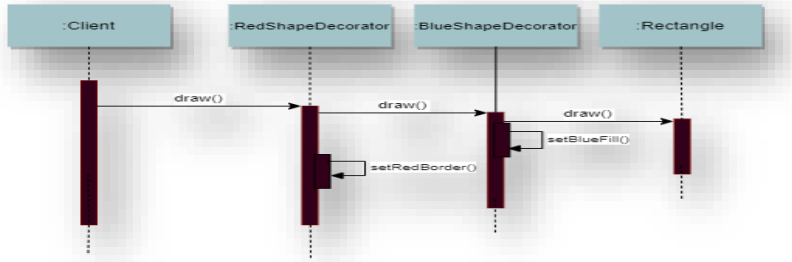
Shape.java

```
public interface Shape {
    void draw();
}
```



Example: Shape Decorator





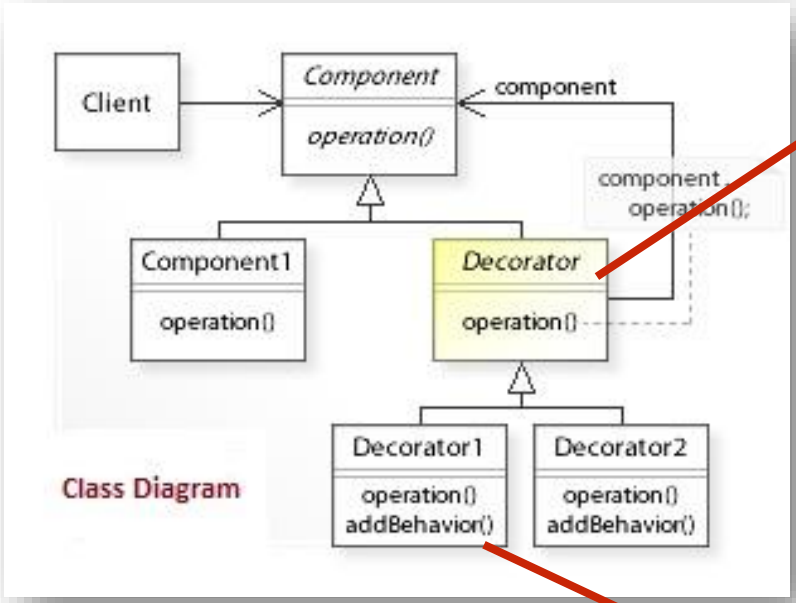
Rectangle.java

```
public class Rectangle implements Shape {

   @Override
   public void draw() {
      System.out.println("Shape: Rectangle");
   }
}
```

ShapeDecorator.java

```
public abstract class ShapeDecorator implements Shape {
   protected Shape decoratedShape;

   public ShapeDecorator(Shape decoratedShape){
      this.decoratedShape = decoratedShape;
   }

   public void draw(){
      decoratedShape.draw();
   }
}
```

Example: Shape Decorator



ShapeDecorator.java

```java
public abstract class ShapeDecorator implements Shape {
   protected Shape decoratedShape;

   public ShapeDecorator(Shape decoratedShape){
      this.decoratedShape = decoratedShape;
   }

   public void draw(){
      decoratedShape.draw();
   }
}
```

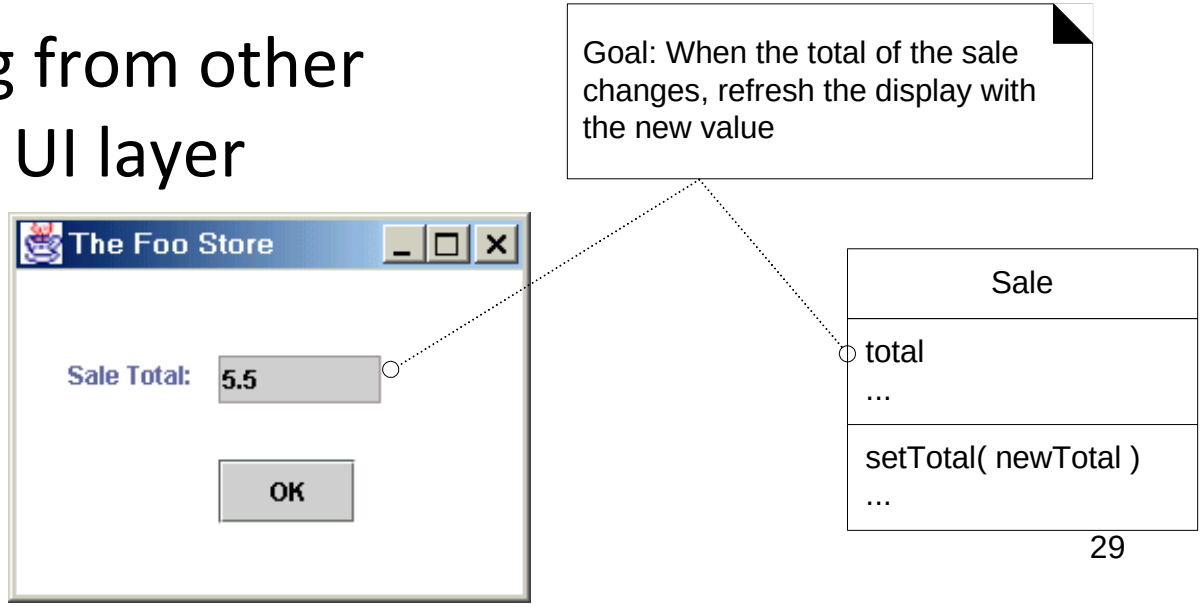
RedShapeDecorator.java

```java
public class RedShapeDecorator extends ShapeDecorator {

   public RedShapeDecorator(Shape decoratedShape) {
      super(decoratedShape);
   }

   @Override
   public void draw() {
      decoratedShape.draw();
      setRedBorder(decoratedShape);
   }

   private void setRedBorder(Shape decoratedShape){
      System.out.println("Border Color: Red");
   }
}
```

Problem 5: Refreshing Sales Total UI

- a GUI window refreshes its display of the sale total when the total changes
- *Sale* object **should not** send a message to a window, asking it to refresh its display
  - Low coupling from other layers to the UI layer

"To handle this problem, let's have an object observing the Sale object"



Observer (aka Publish-Subscribe) (GoF)

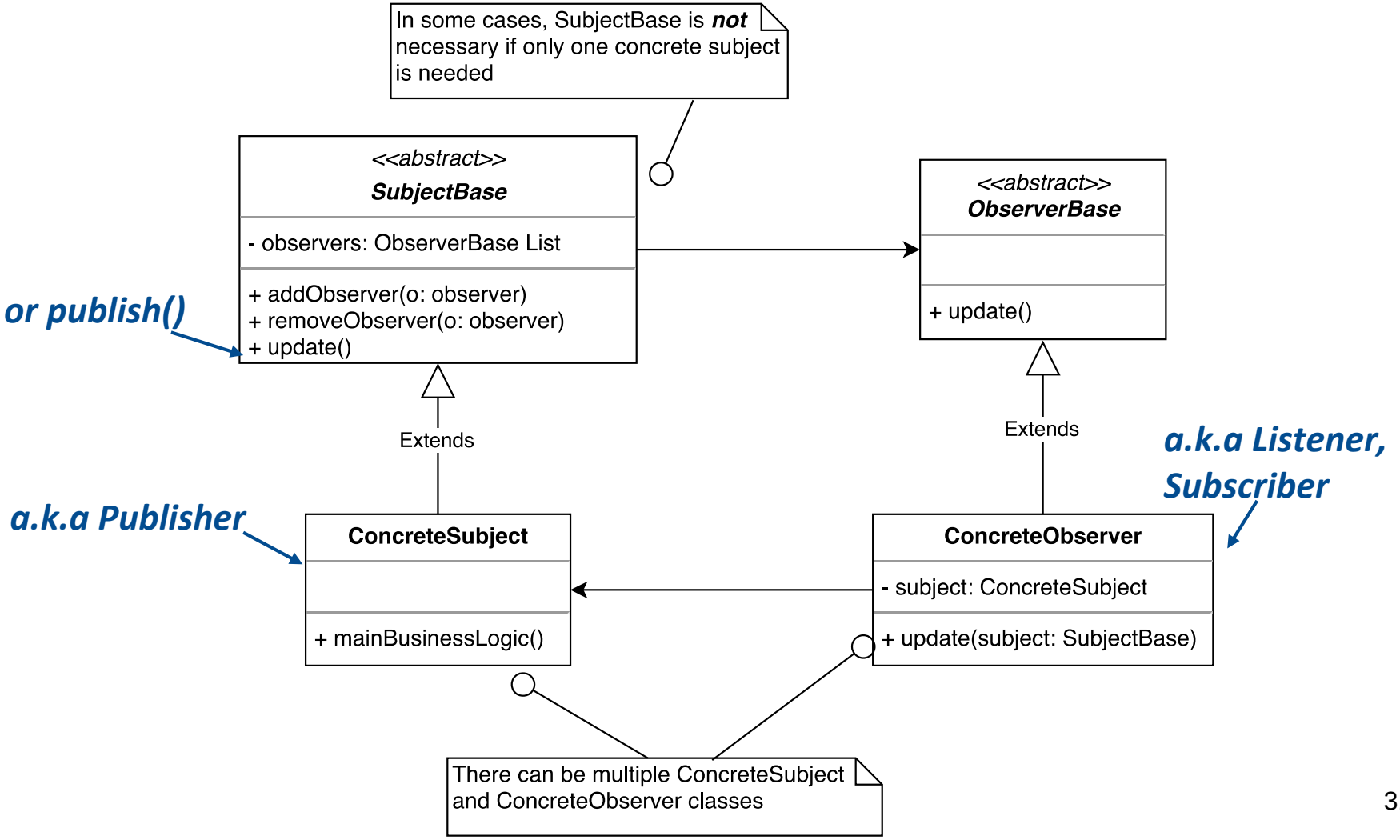
Problem:

- Different kinds of subscriber objects are interested in the state changes or events of a publisher object, and want to react in their own unique way when the publisher generates an event. Moreover, the publisher wants to maintain low coupling to the subscribers. What to do?

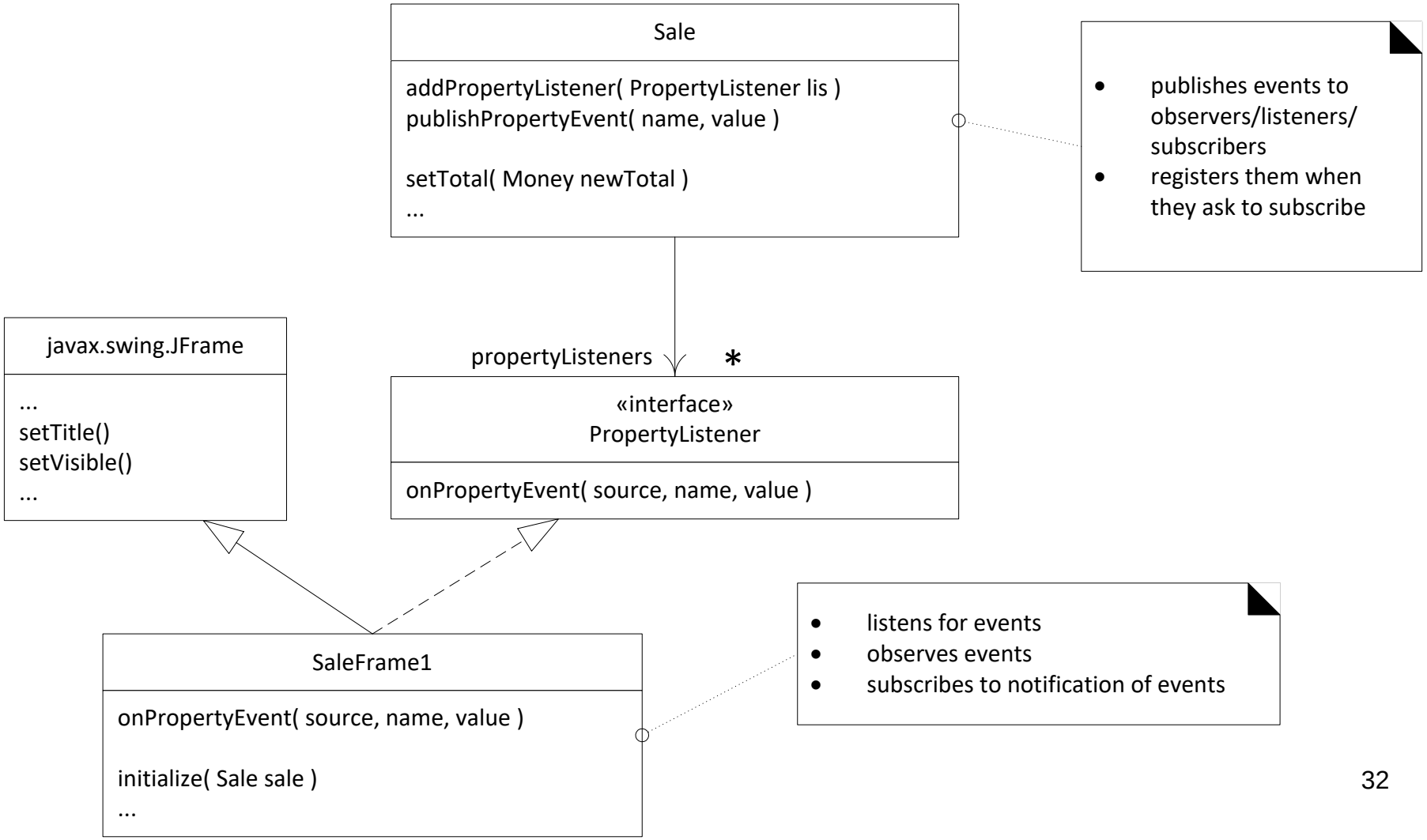
Solution (advice):

- Define a “subscriber” or “listener” interface. Subscribers implement this interface. The publisher can dynamically register subscribers who are interested in an event and notify them when an event occurs.

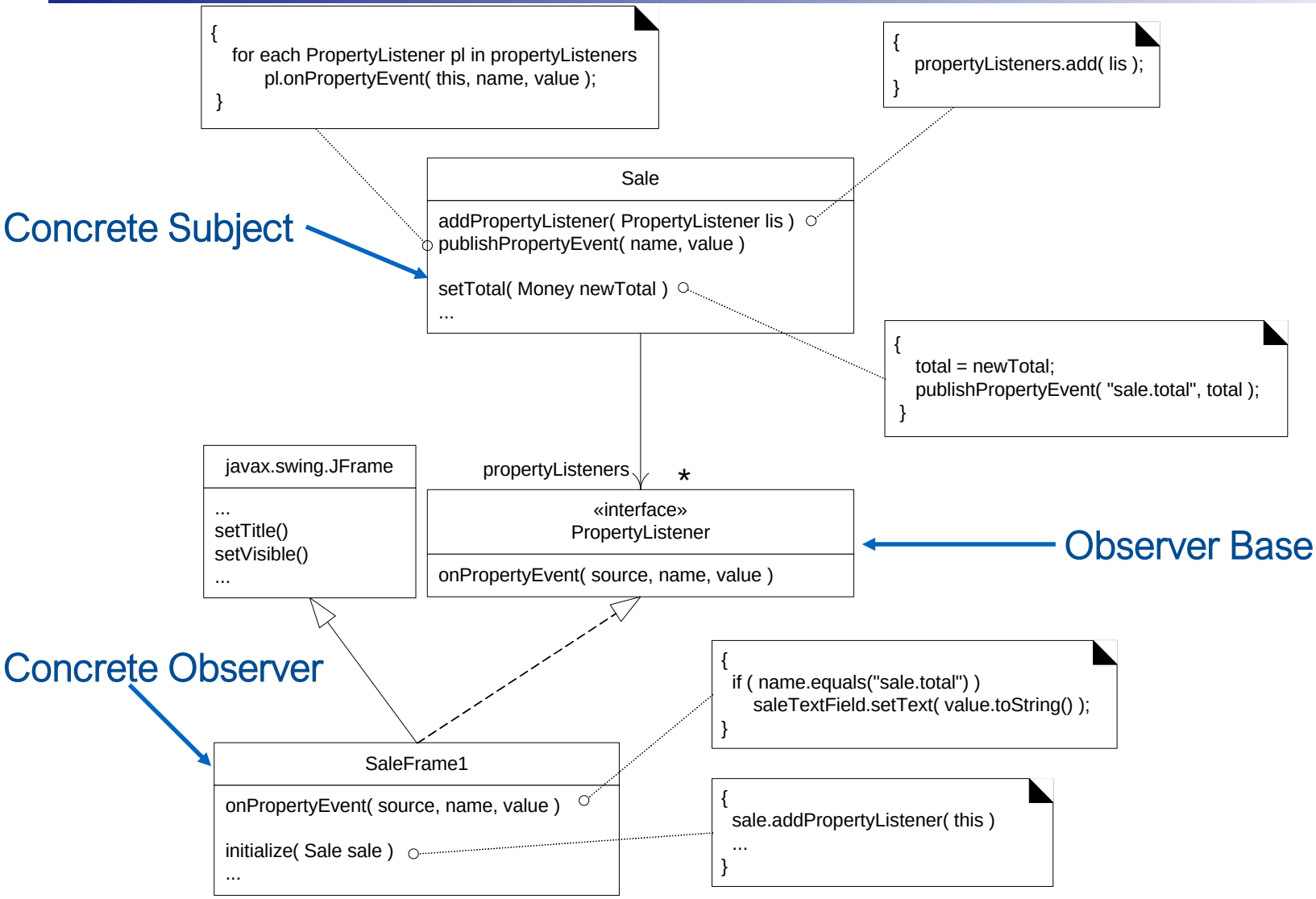
Observer: Generalised Structure



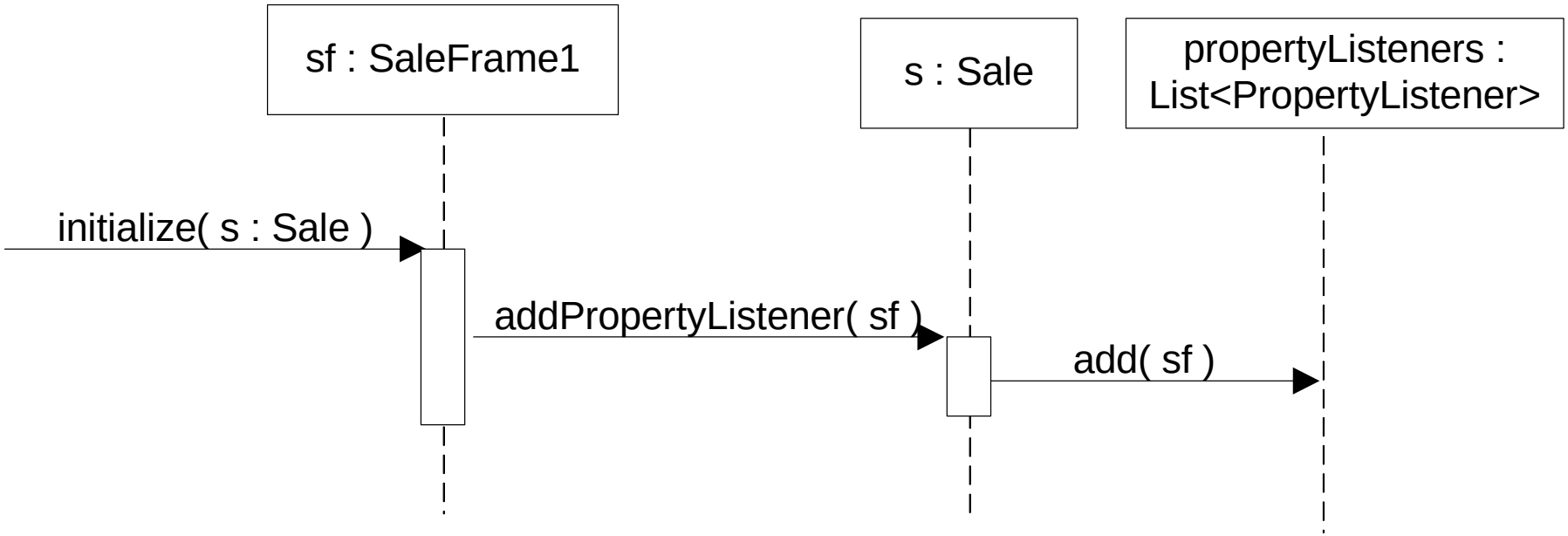
Who is?: Observer, Listener, Subscriber, Publisher



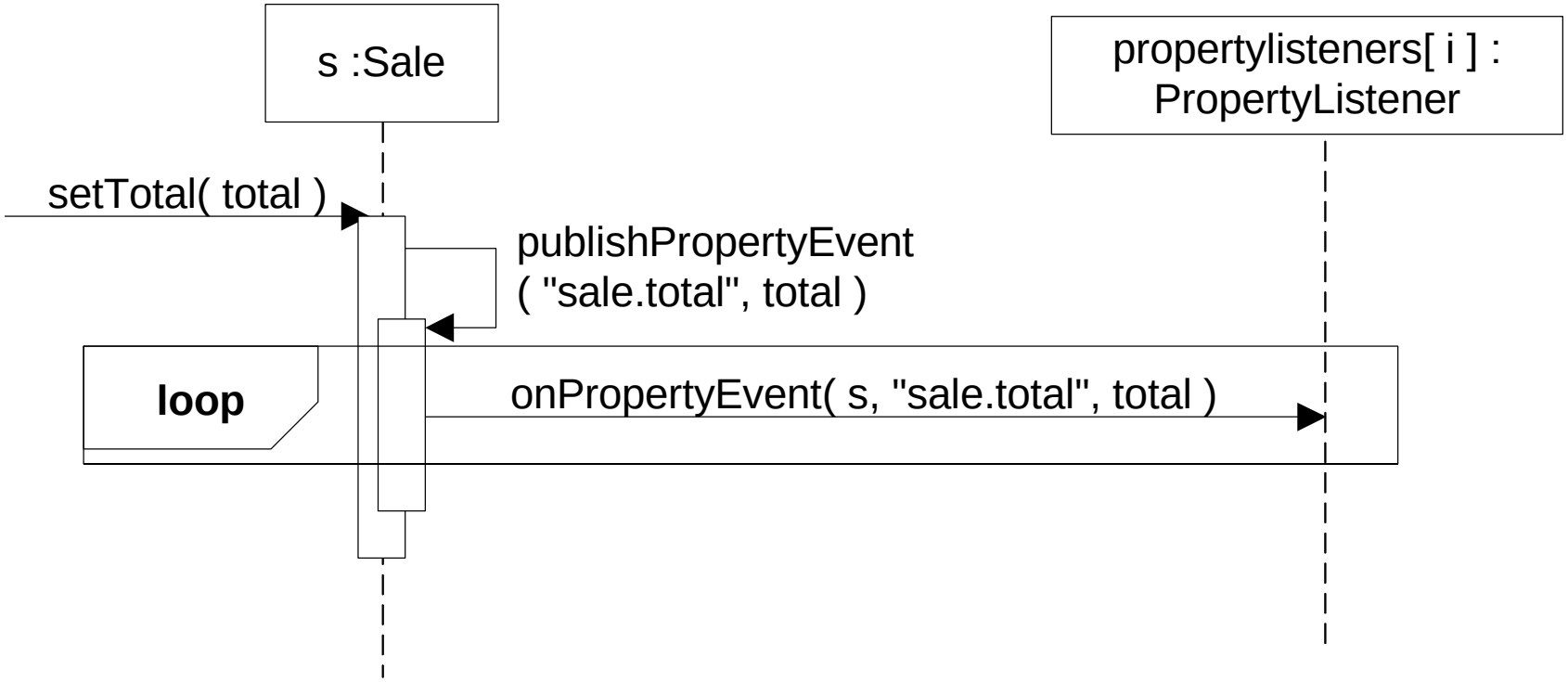
The Observer Pattern (Sale Total)



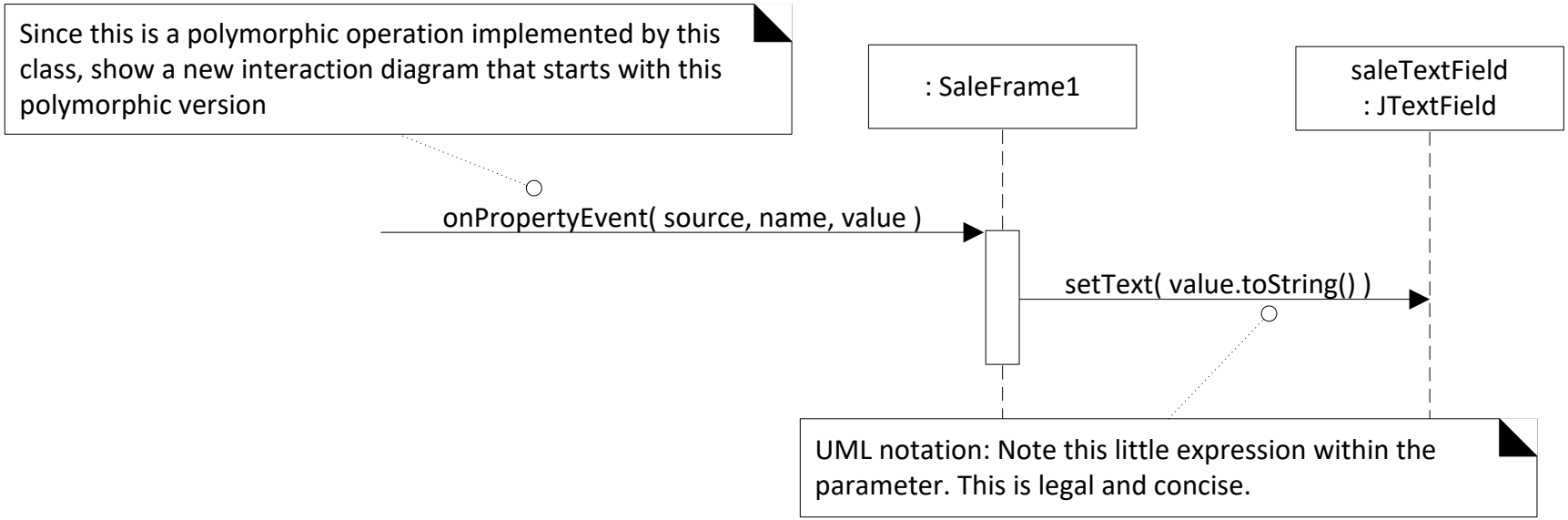
Ob. *SalesFrame1* Subscribes to Pub. *Sale*



Sale Publishes Property Event to Subscribers



Subscriber *SaleFrame1* Receives Notification



Summary of related GoF Patterns

- **Adapter** provides a different interface to its subject. **Decorator** provides an enhanced interface.
- **Composite** and **Decorator** have similar structure, reflecting the fact that both rely on recursive composition to organize an open-ended number of objects.
- **Composite**, unlike **Decorator**, is not focused on embellishment but on representation. These intents are distinct but complementary. Consequently, Composite and Decorator are often used in concert.
- **Decorator** lets you change the skin of an object. **Strategy** lets you change the guts.

GoF Patterns Understanding

To complete the survey, go to PollEv.com/petereze226

0 surveys done

0 surveys underway

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.