Design Patterns

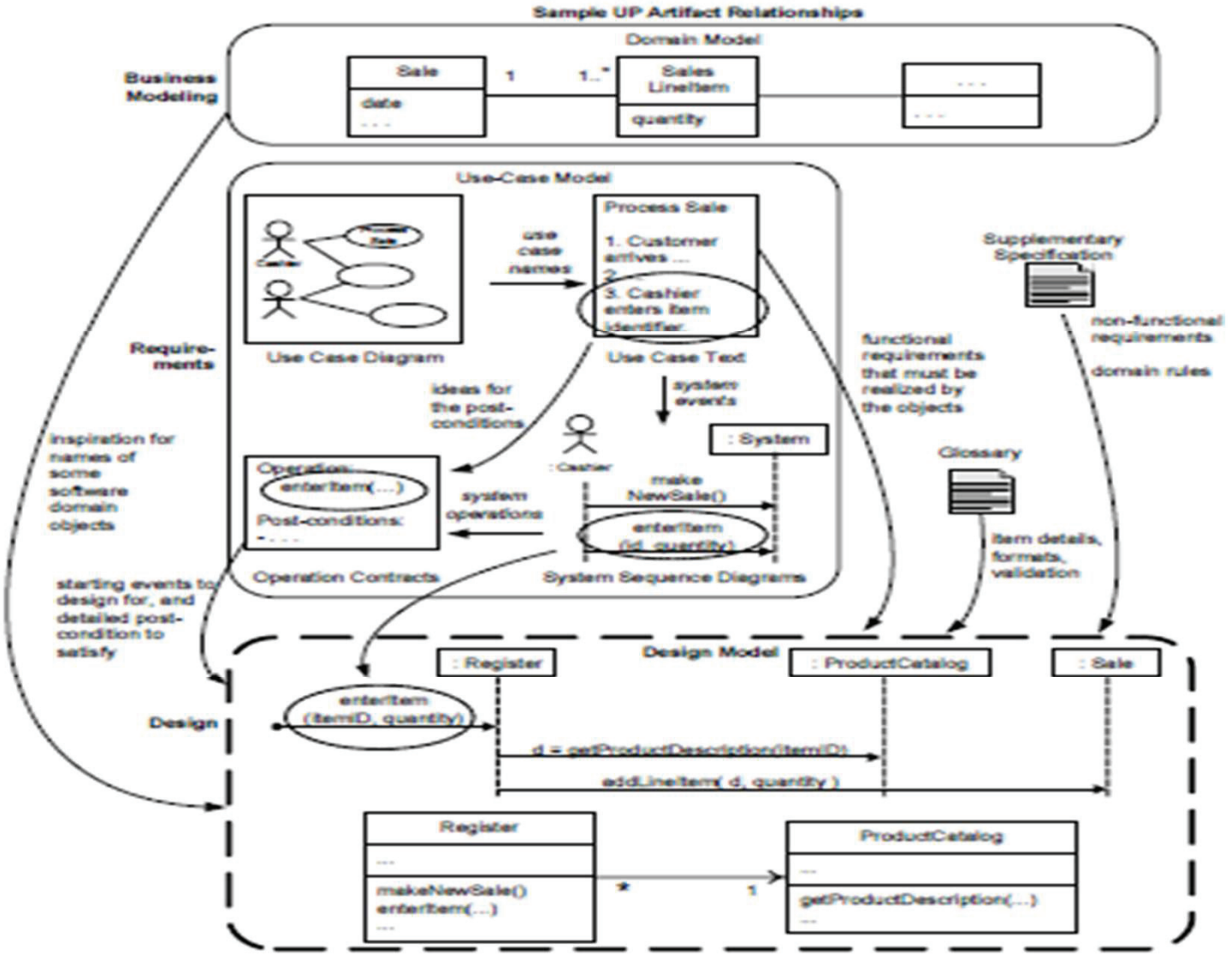
Syllabus

- Responsibility Driven Design (RDD)
- GRASP
  - Creator
  - Information Expert
  - Controller
  - Low coupling
  - High Cohesion
- GOF Patterns
  - Creational
    - Factory
  - Structural
    - Bridge
    - Adapter
  - Behavioral
    - Strategy
    - Observer

Design Patterns - Introduction

- A simple definition: In the analysis part, you have identified use cases and created use case descriptions to get the requirements created and refined the domain concept model.
- Now in order to make a piece of object design, you assign methods to software classes
- Design how the classes collaborate (i.e. send messages) in order to fulfil the functionality stated in the use cases.
- Central tasks in design are:
  1. Deciding what methods belong where
  2. How the objects should interact
- Object Oriented Design is primarily about assigning responsibilities to objects.

Design Patterns - Introduction



Larman (2005), Fig. 17.1

Responsibilities Driven Development (RDD)

- In design model, responsibilities are obligations of an object in terms of its behaviour.
- Responsibility-driven design concerns object responsibilities, roles, and collaborations. There are two types of responsibilities:

1. Doing responsibilities concern performing a task such as creating an object, performing a calculation, coordinating activities, and so on.
   - Doing something itself, such as creating an object or doing a calculation
   - Initiating action in other objects
   - Controlling & Coordinating activities in other objects
2. Knowing responsibilities concern knowing about private data, related objects, data needed for calculations, and so on.

Responsibilities Driven Development (RDD)

- Responsibilities are implemented by means of methods that either act alone or collaborate with other methods and objects.
- “Knowing” responsibilities will often be inspired by the domain model. Given a domain object Sale having a time attribute, according to the principle of minimal representational gap, we would probably say that the Sale software object should be responsible for knowing its time.
- Collaborations mean the objects that cooperate to get work done. Collaboration example: a Sale object might invoke the getSubtotal method on multiple SaleLineItem objects to fulfil the responsibility of calculating a total

Responsibilities and Interaction diagrams

- Responsibilities are assigned to objects during object design while creating interaction diagrams i.e., Sequence diagrams and Collaboration diagrams
- Examples: "a Sale is responsible for creating SalesLineItems" (a doing), or
- "a Sale is responsible for knowing its total" (a knowing).

Design Patterns

- Design patterns were introduced by an architect Christopher Alexander in architecture. He studied the problem: What is present in a good quality design is not present in poor quality design?
- Alexander observed buildings, towns, streets, and virtually every aspect of humans have built for themselves
- Structures that solve similar problems [school, hospital buildings] even though they look different, they have similarities if their designs are of high quality.
- He called these similarities as Patterns
- He defined a pattern as a solution to a problem in a context
- Alexander works influenced software development and led to development of patterns.

Software Design Patterns

- Software designers face common [similar] problems in different projects
- Experienced designers reuse solutions that have worked in the past.
- Patterns describe solutions discovered by experienced software developers for common problems in software design.
- A pattern is a recurring solution to a standard problem in a context – Alexander
- In OO design, a pattern is a named description of a problem and solution that can be applied in new contexts
- A pattern advises us on how to apply the solution in various circumstances and considers the forces and trade-offs.

Responsibilities Driven Development (RDD) - GRASP

- Acronym for General Responsibility Assignment Software Patterns
- Collection of some principles and basic patterns
- Describe fundamental principles of object design and responsibility
- Composed by larman as a learning aid.
- When drawing interaction diagrams, you are implicitly assigning responsibilities to particular objects.
- Use GRASP when drawing interaction diagrams and coding

Responsibilities Driven Development (RDD) – Fundamental GRASP principles

5 fundamental GRASP principles / patterns

1. Creator
2. Information Expert
3. Controller
4. Low Coupling
5. High Cohesion

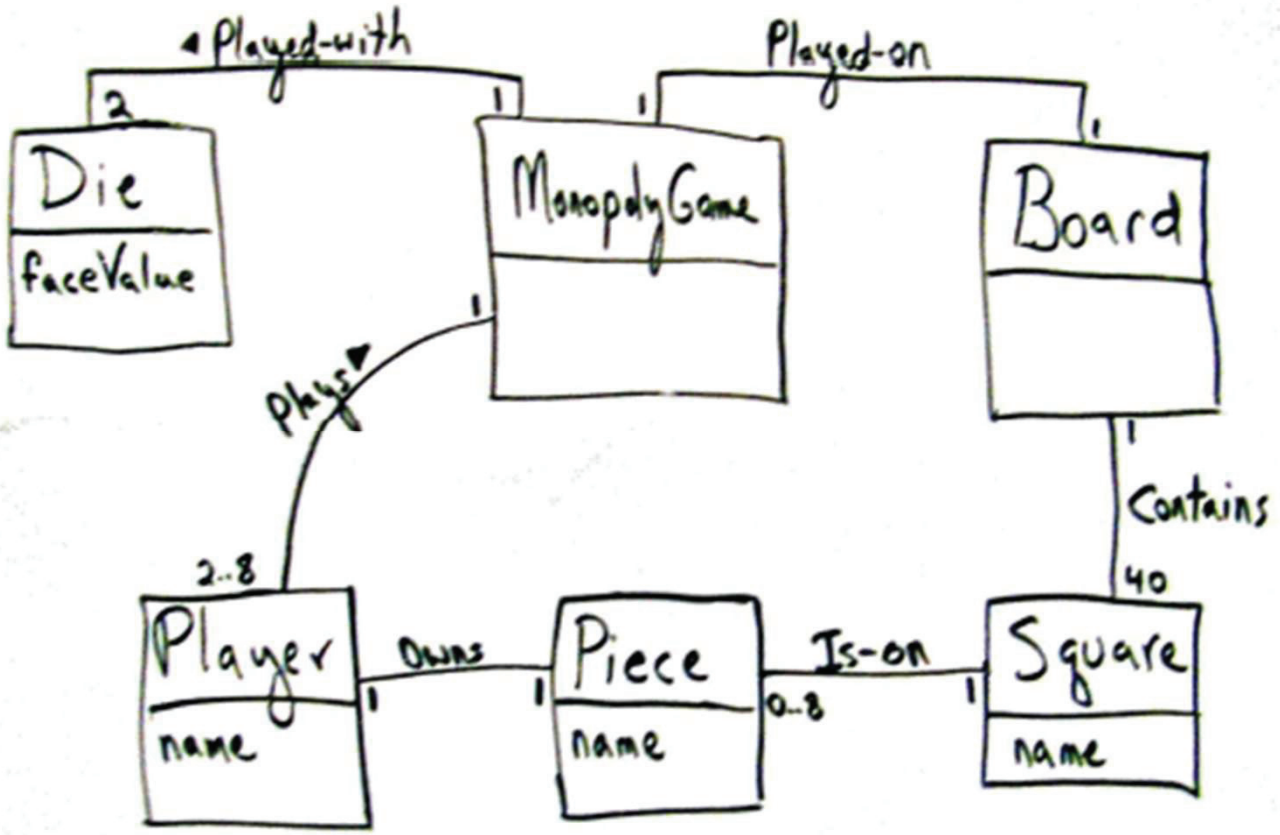
GRASP

- Creator: The answer of “who creates the object X?”
- Information Expert: Assign a responsibility to the class that has the information necessary to fulfill the responsibility
- Controller: Put a controller object between two layers
- Low coupling: Assign a responsibility so that coupling remains low
- High Cohesion: A class with high cohesion has a relatively small number of methods, with highly related functionality and does not do too much work.

Creator Pattern

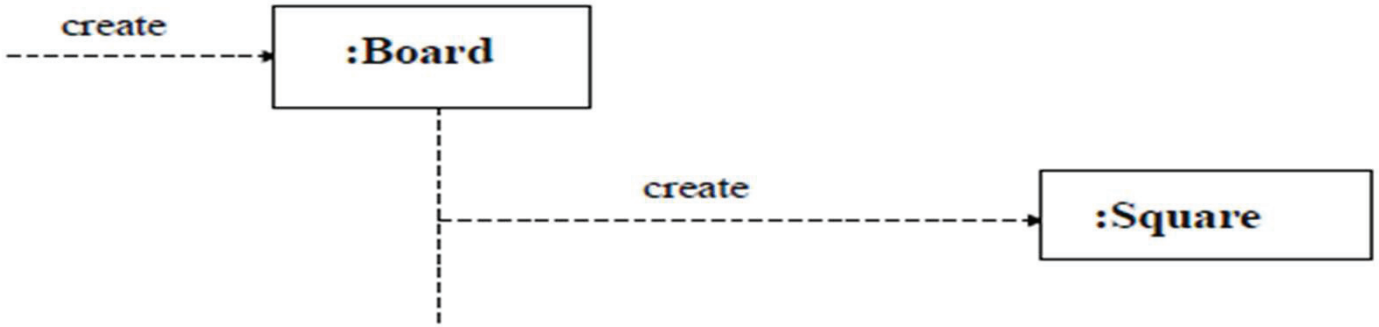
- Creation of objects is one of the most common activities in an object-oriented system.
- Which class is responsible for creating objects is a fundamental property of the relationship between objects of particular classes.
- Name: Creator Problem: Who creates an instance of A?
- Solution: Assign class B the responsibility to create an instance of class A if one of these is true. B creates A if
- Instances of B contains or aggregates instances of A (in a collection – aggregation / composition)
- Instances of B records instances of A
- Instances of B closely uses instances of A
- Instances of B have the initializing information for instances of A and pass it on creation

Implementation of Creator Pattern in Monopoly



Implementation of Creator Pattern in Monopoly

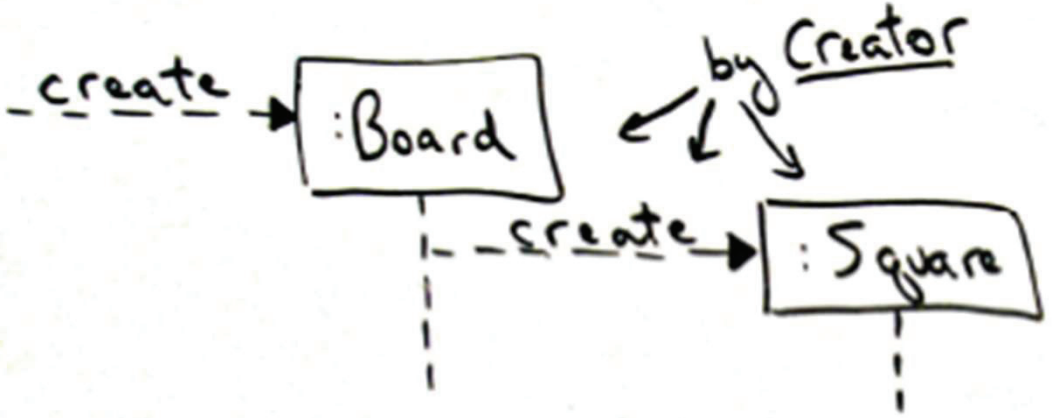
Applying the Creator pattern in Static Model



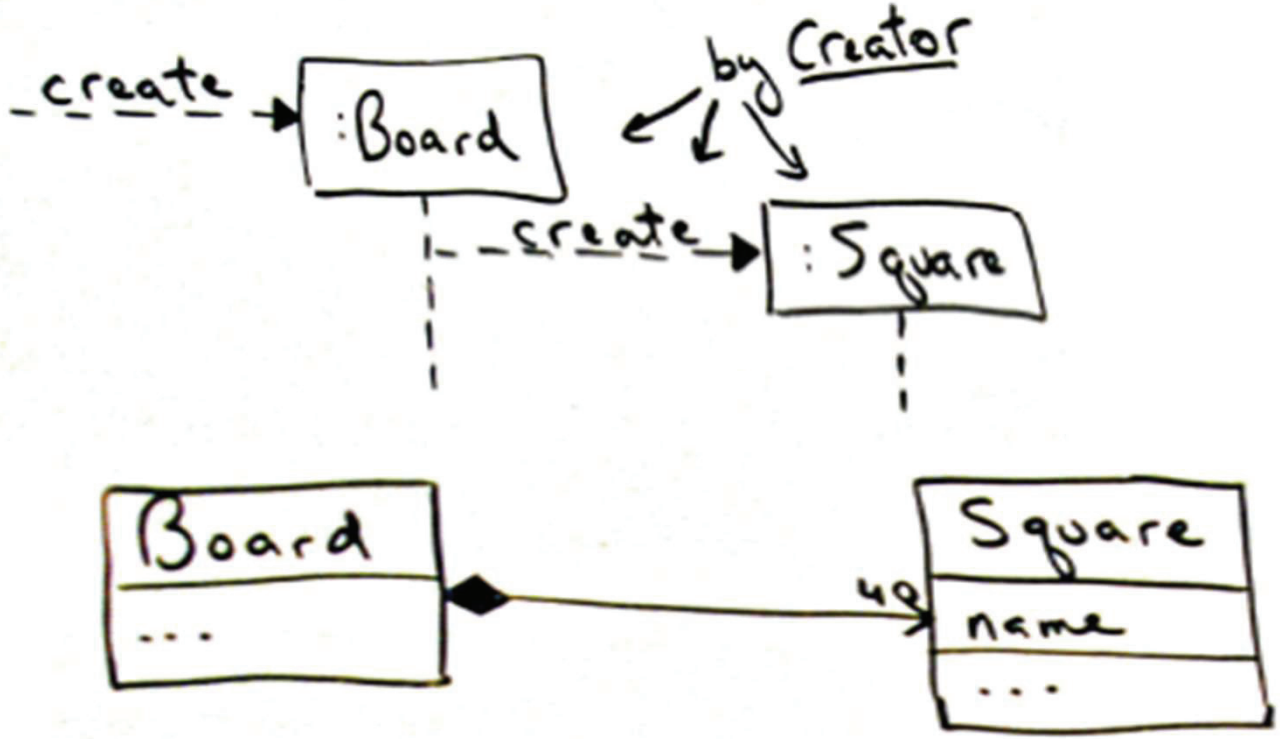
Dynamic Model – illustrates Creator Pattern

Implementation of Creator Pattern in Monopoly

- The board object contains Square objects, so it is a good candidate. This gives us clue that the relationship should be composition

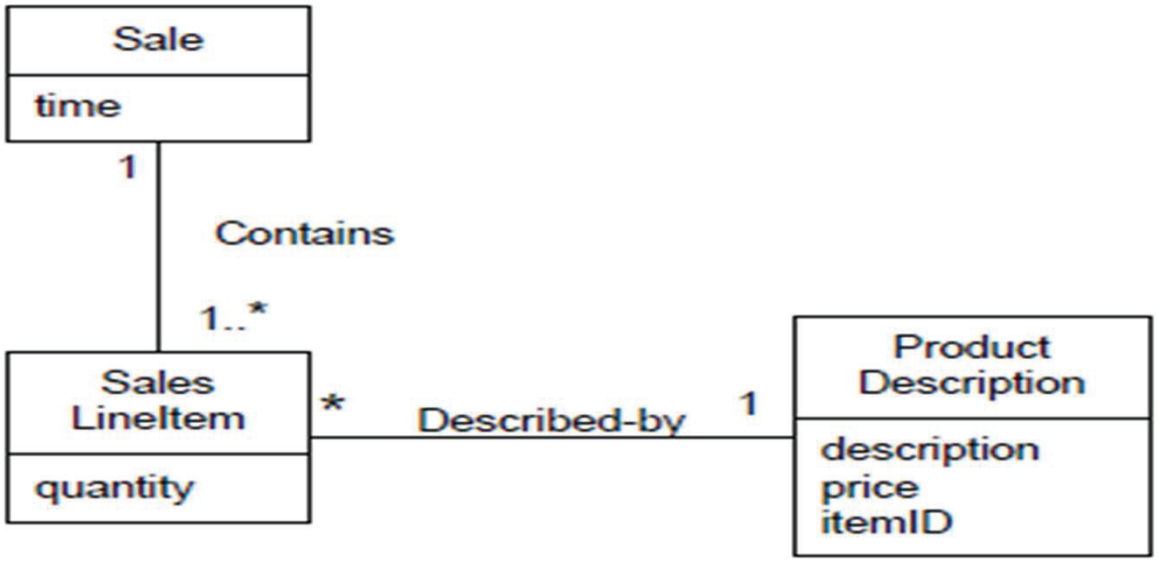


Implementation of Creator Pattern in Monopoly



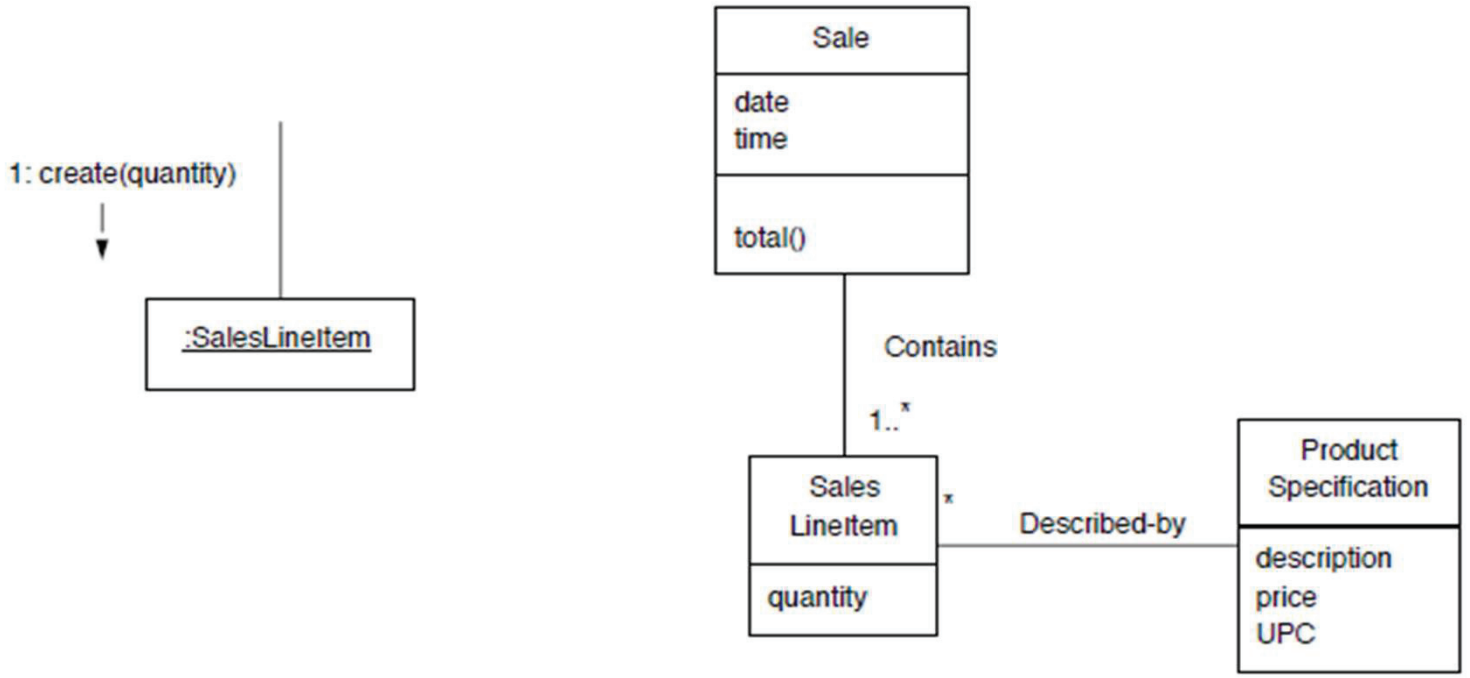
Implementation of Creator Pattern in POS

- In POS system who should create the SalesLineItem objects?
- Domain model



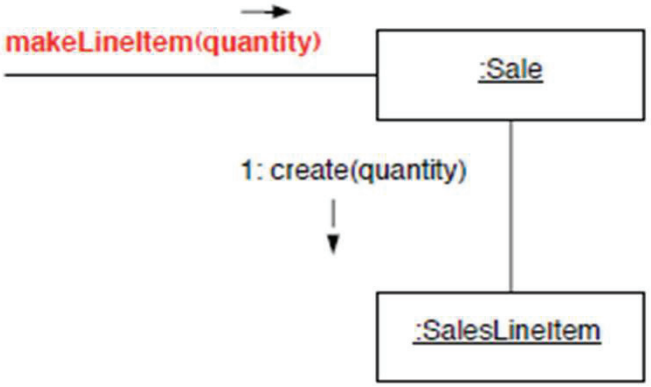
Implementation of Creator Pattern in POS

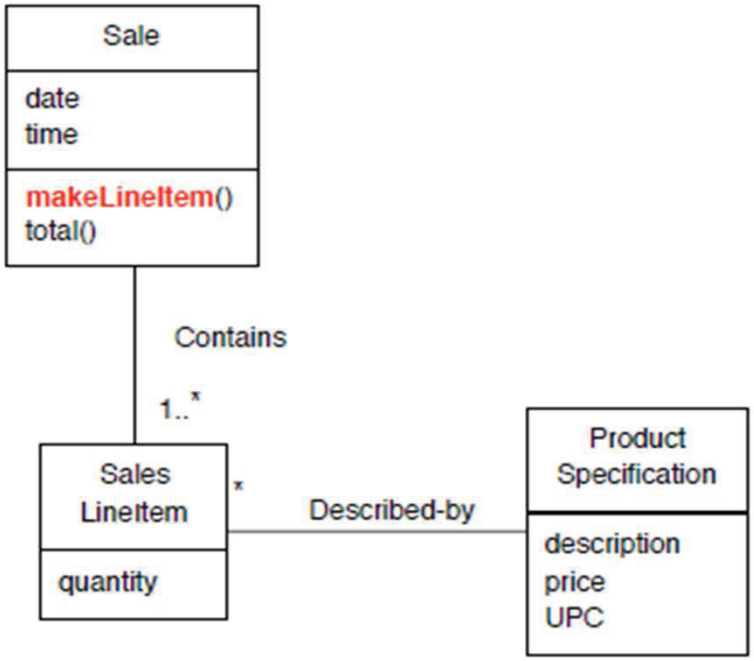
- Creation of “SalesLineItem” Instances



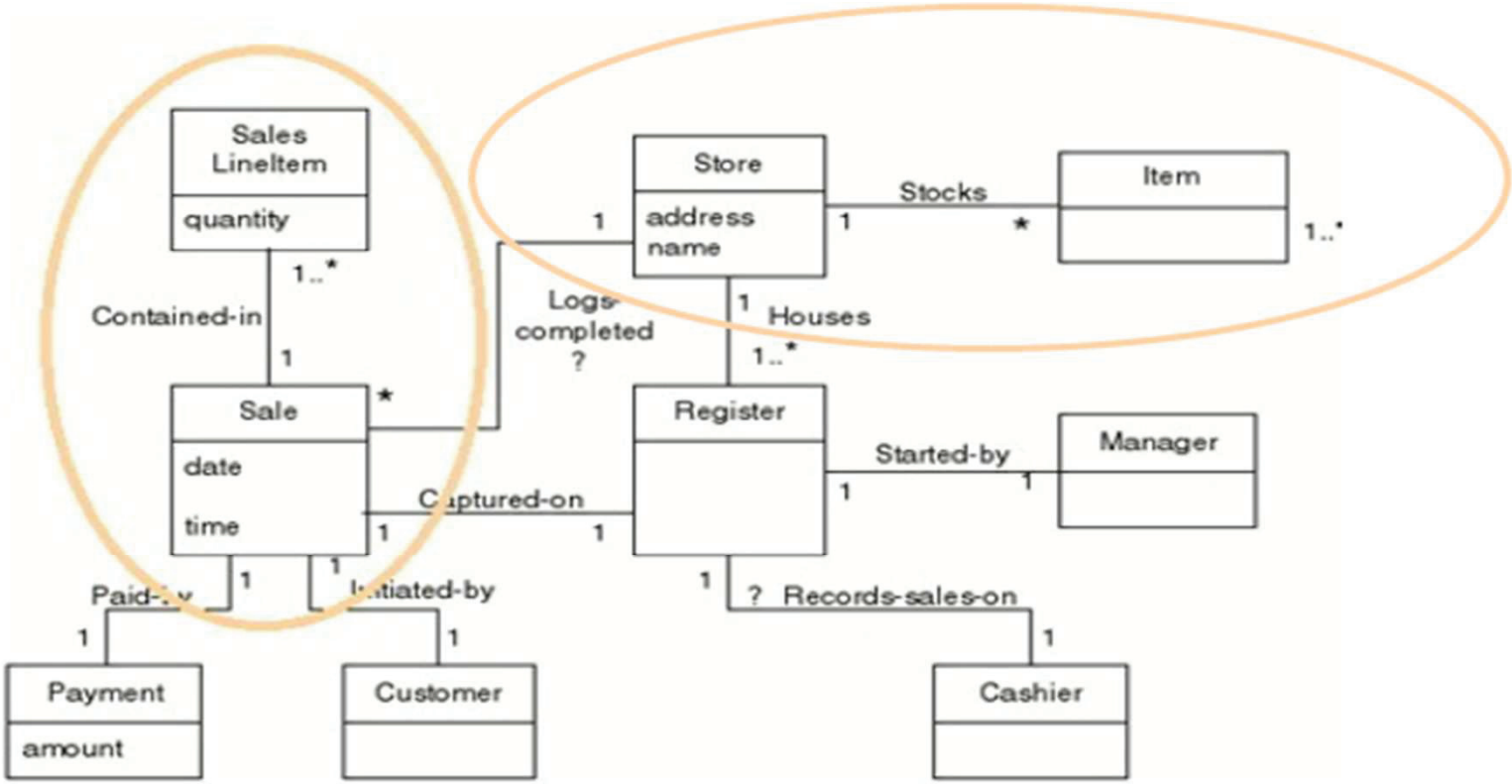
Implementation of Creator Pattern in POS

- Creation of "SalesLineItem" Instances





Implementation of Creator Pattern in POS – Inspired from domain model



Information Expert

- Information Expert is a principle used to determine where to delegate responsibilities. These responsibilities include methods, computed fields and so on.
- Using the principle of Information Expert a general approach to assigning responsibilities, is to look at a given responsibility, determine the information needed to fulfil it, and then determine where that information is stored.

Information Expert

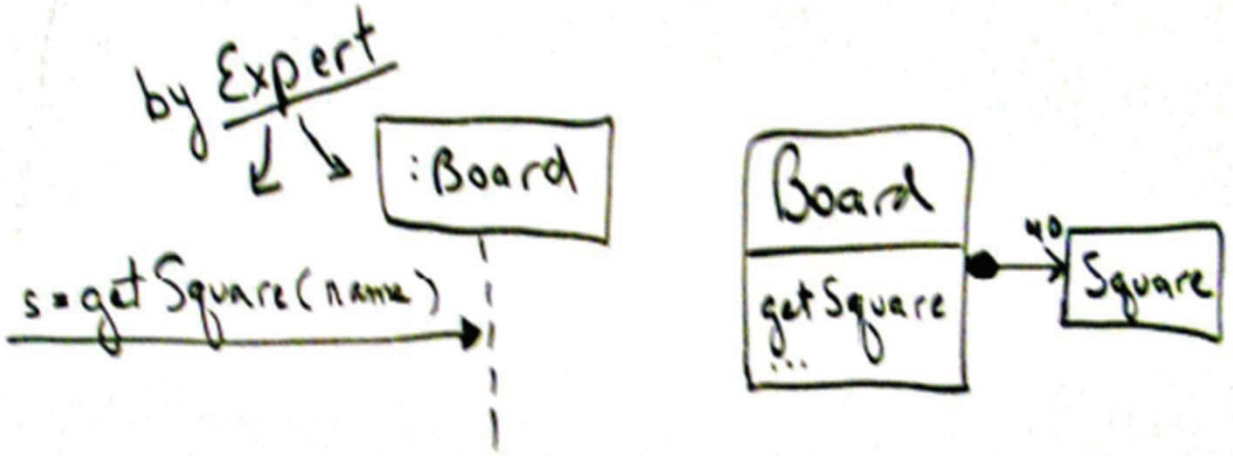
- Name: Information Expert Problem: What is a basic principle by which to assign responsibilities to an object
- Solution: Assign a responsibility to the class that has the information needed to respond to it.
- Who should be responsible for knowing about instances of a class?
- We assign knowing responsibilities to the class having the information needed to fulfil the responsibility.
- Suppose objects need to be able to reference a particular Square, given its name.
- Who knows about a Square object, given a key?

Information Expert

- When assigning “knowing" responsibilities, first look in the design model for an appropriate class.
- If not found in the design model, look in the domain model

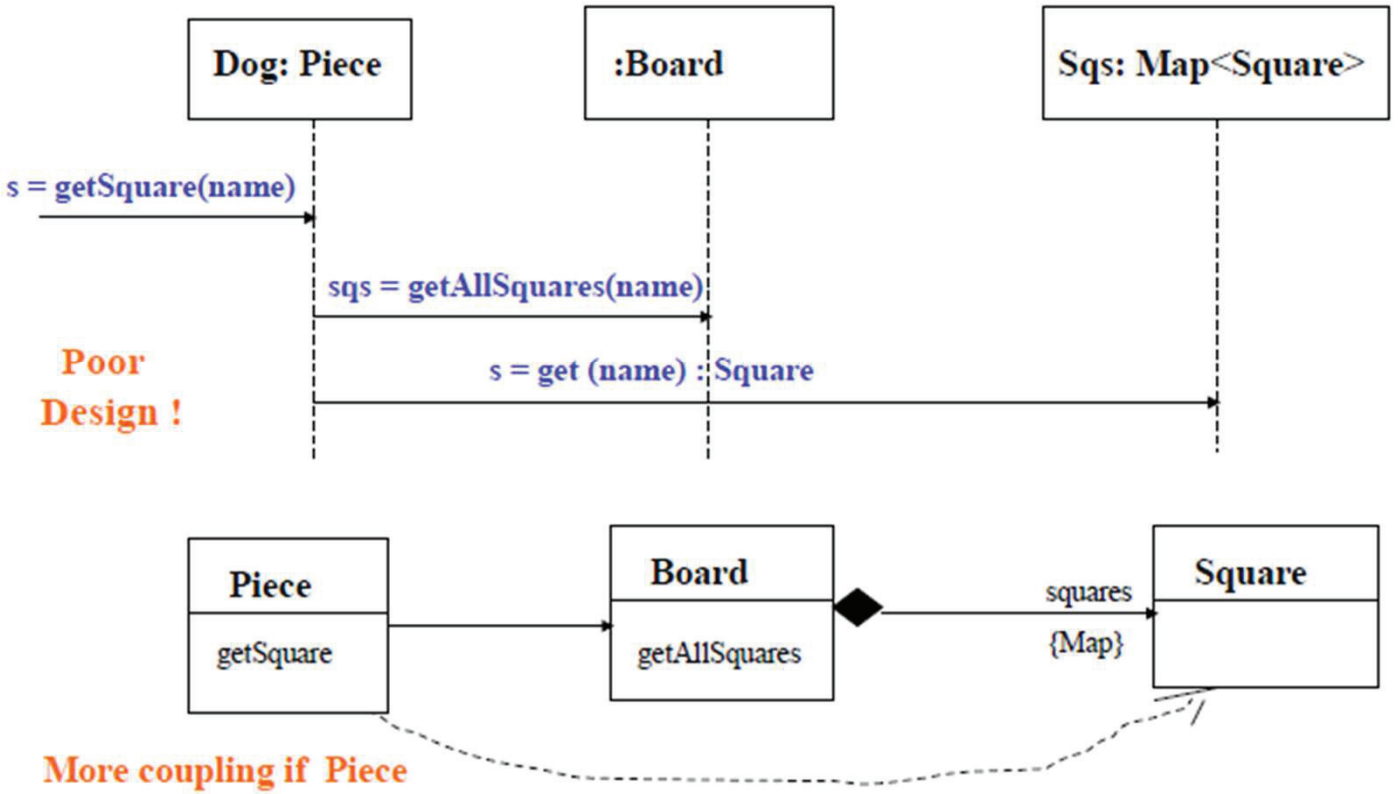
Information Expert – Monopoly game

- The Board aggregates all of the Squares, so the Board has the Information needed to fulfil this responsibility.
- So Board should have the responsibility to know about square instances



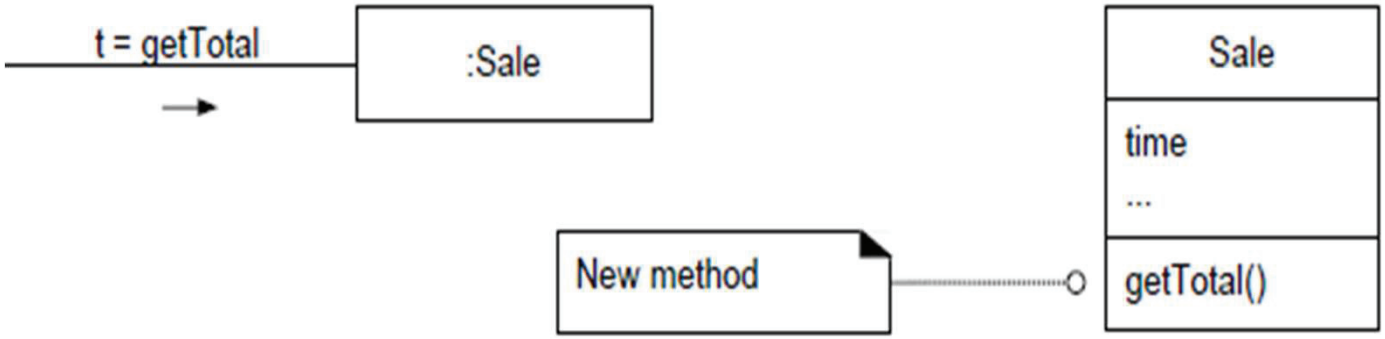
Information Expert – Monopoly game

- Alternative Design – More coupling if Piece has getSquare()



Information Expert – POS

- What information is needed to determine the grand total?
- It is necessary to know about all the SalesLineItem instances of a sale and the sum of their subtotals.
- A Sale instance contains these; therefore
  - by the guideline of Information Expert, Sale is a suitable class of object for this responsibility



Information Expert – POS

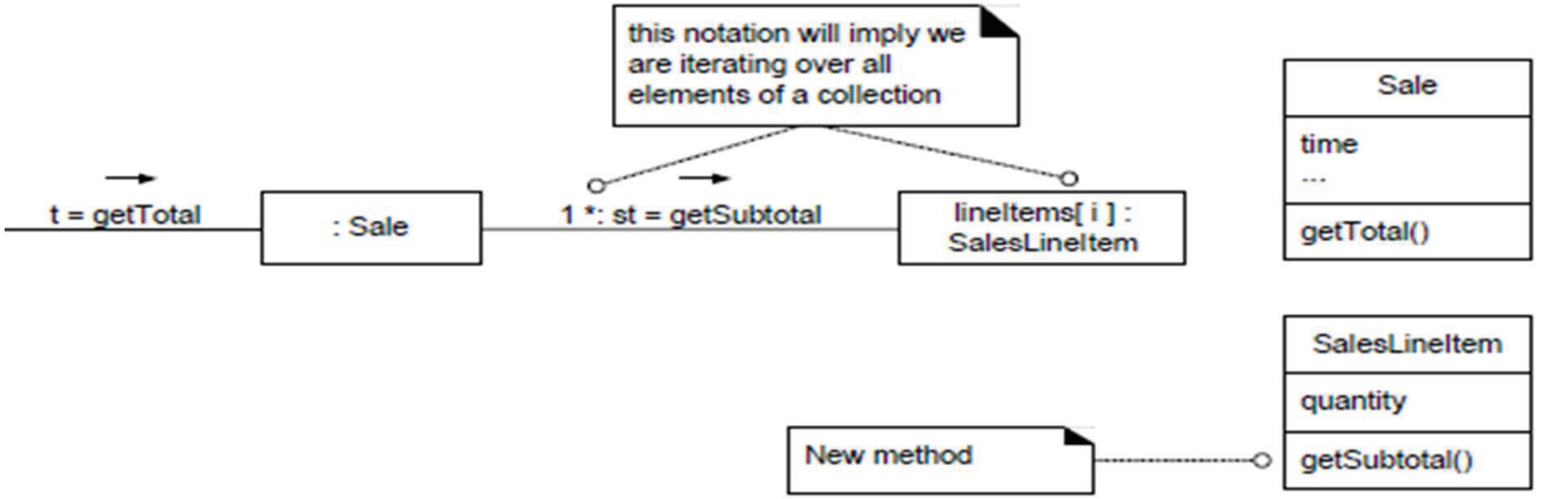
- What information is needed to determine the line item subtotal?
  - by Expert, SalesLineItem should determine the subtotal
- To fulfil this responsibility, a SalesLineItem needs to know the product price.
  - By Expert, the ProductDescription is an information expert on answering its price
- In conclusion, to fulfil the responsibility of knowing and answering the sale's total, three responsibilities were assigned to three design classes

Information Expert – POS

Design Class	Responsibility
Sale	Knows Sale total
SalesLineItem	Knows line item subtotal
ProductSpecification	Knows product price

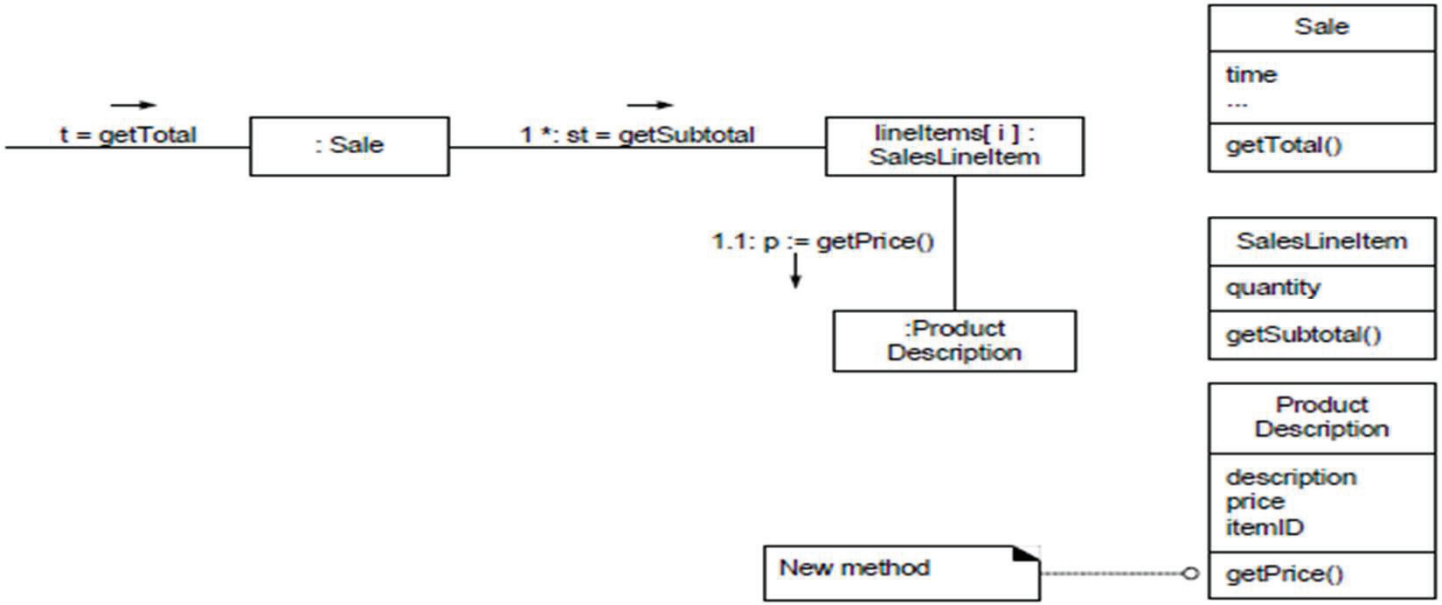
Information Expert – POS

- But to get the price of a SalesLineItem, we also need the ProductDescription's price attribute. SalesLineItem has the needed information:



Information Expert – POS

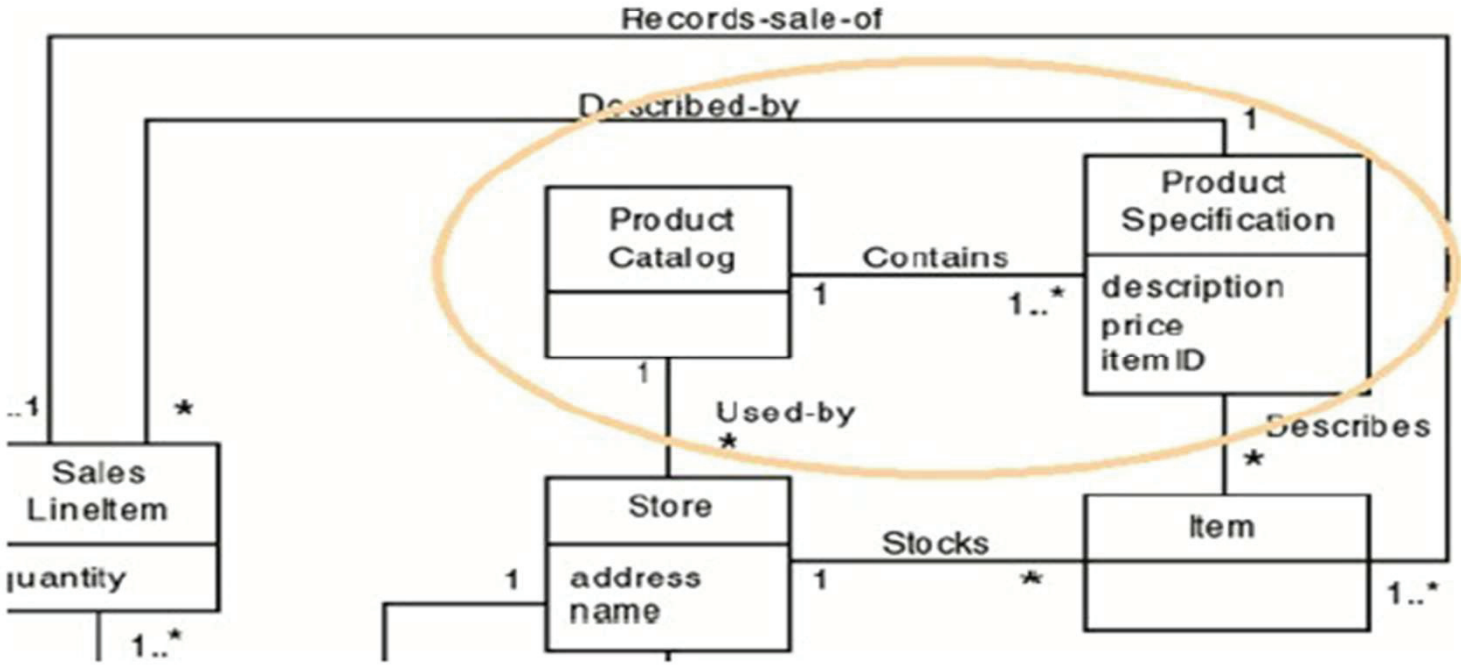
- To fulfil the responsibility to know its subtotal, the SalesLineItem needs to collaborate with ProductDescription:



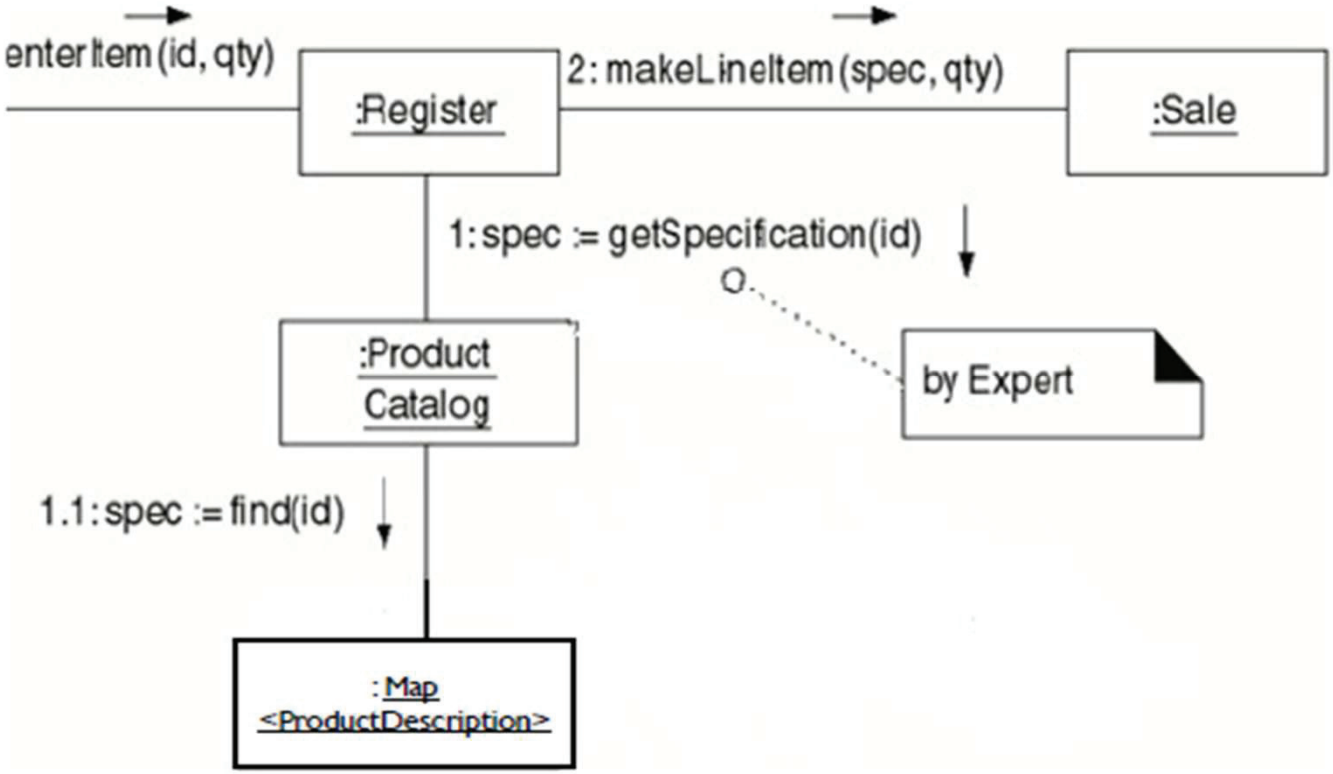
Information Expert – POS

What object should be responsible for knowing ProductSpecifications, given a key?

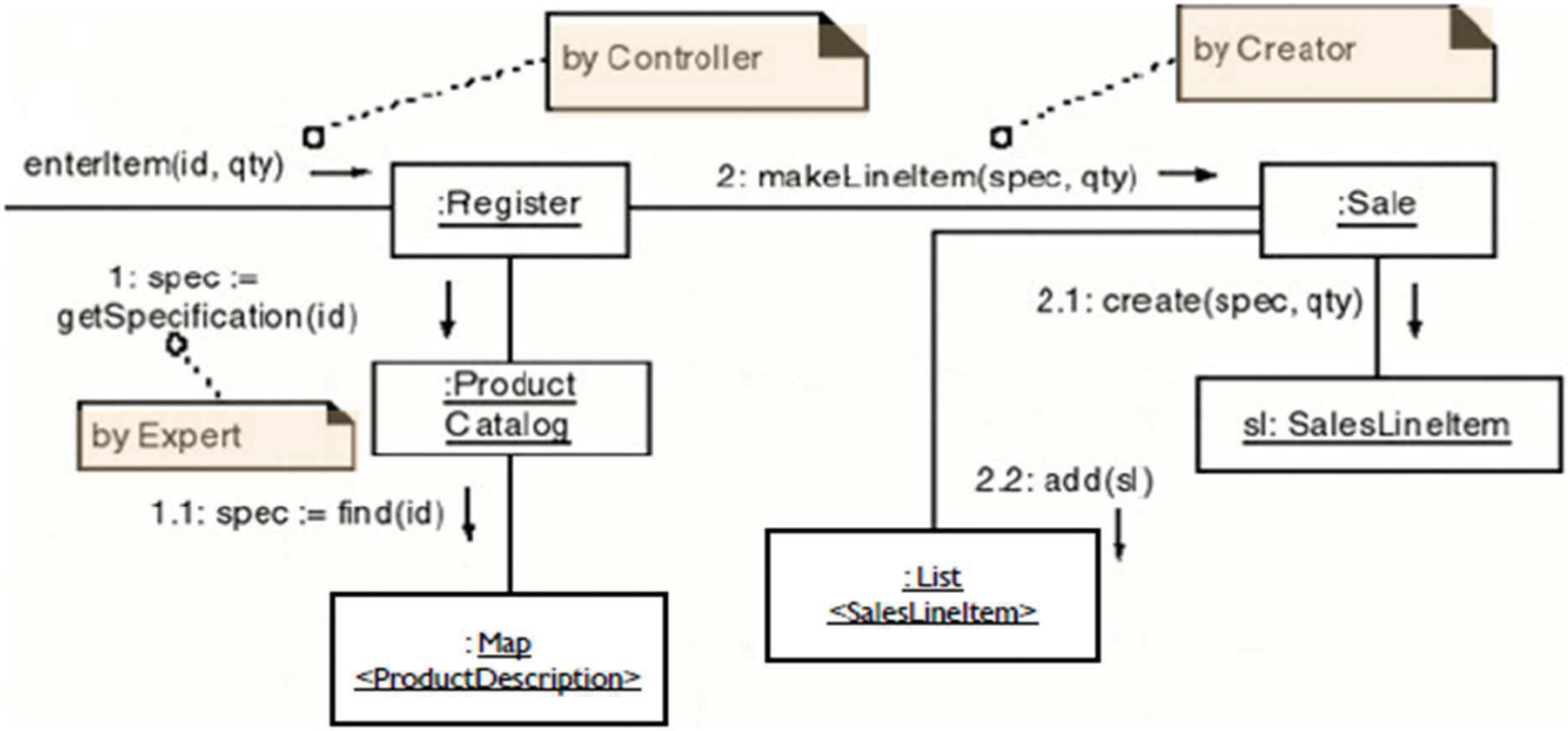
Take inspiration from the domain model



Information Expert – POS



Design for “enteritem” – 3 patterns applied



Controller

- A simple layered architecture has a user interface layer (UI) and a domain layer. Actors, such as the human player in Monopoly, generate UI events (such as clicking a button with a mouse to play a game or make a move).
- The UI software objects (such as a JFrame window and a JButton) must process the event and cause the game to play.
- When objects in the UI layer pick up an event, they must delegate the request to an object in the domain layer.
- What first object beyond the UI layer should receive the message from the UI layer?

Controller

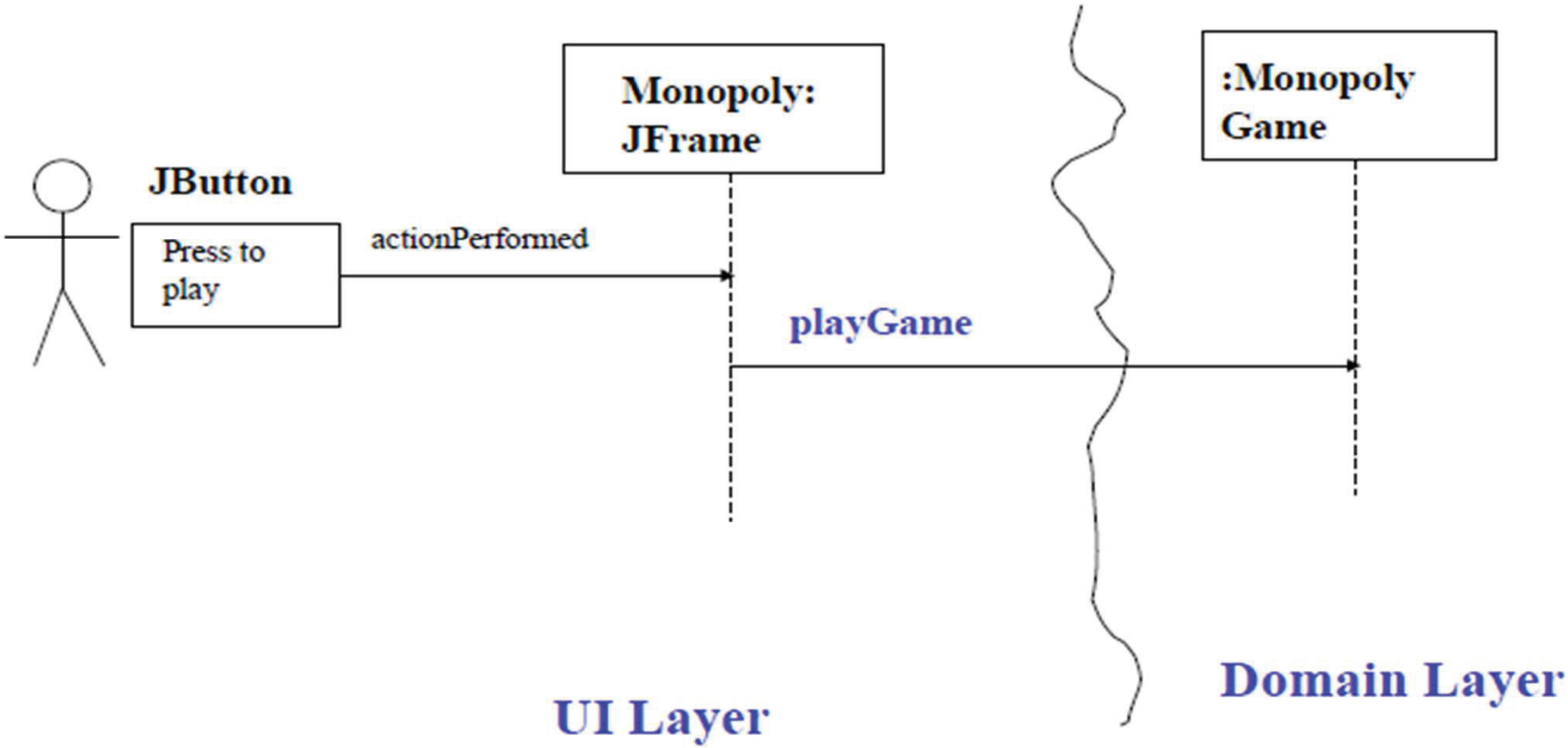
- Problem: What first object beyond the UI layer receives and coordinates a System Operation?
- Solution: Assign the responsibility to an object representing one of these choices:
  1. Represents the overall “system” – a root object
  2. Represents a use case scenario within which the system operation occurs (a use case or session handler)

Controller

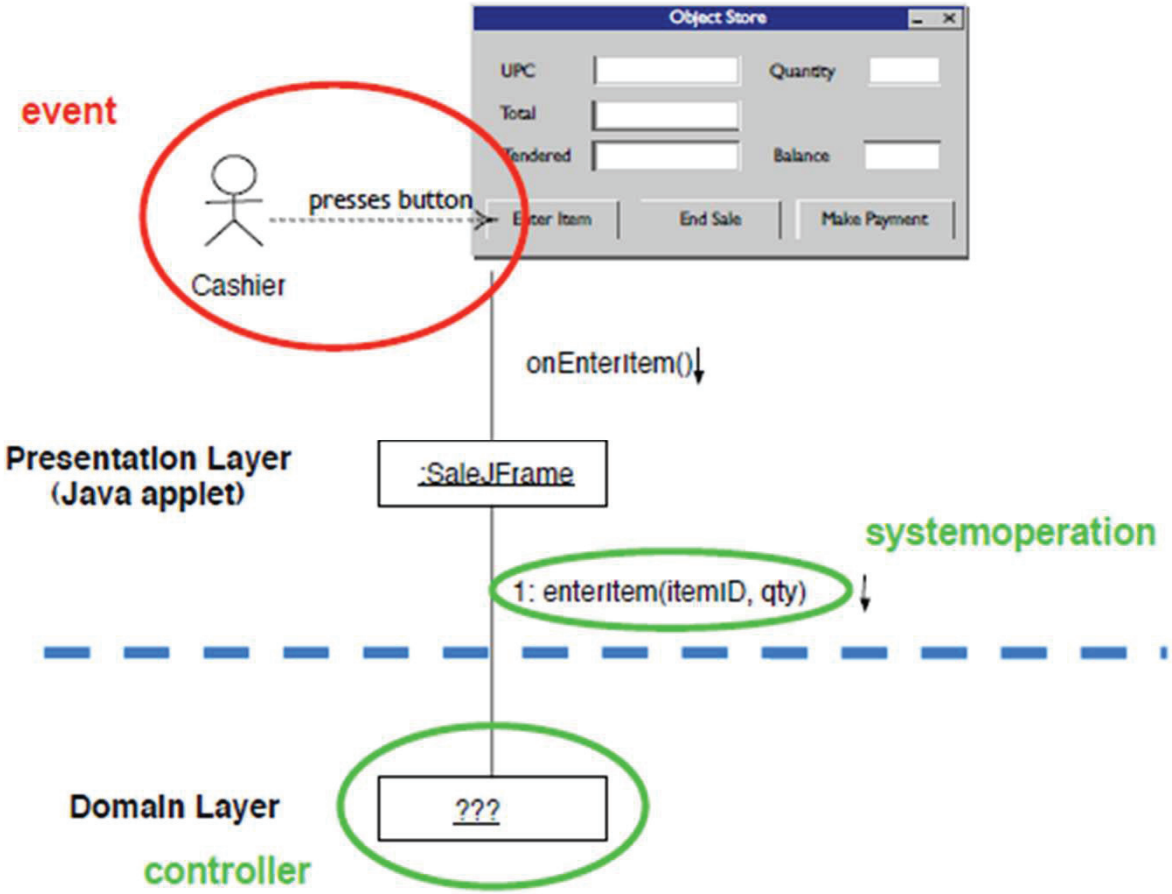
- Problem: What first object beyond the UI layer receives and coordinates a System Operation?
- Solution: Place a controller object between two layers.
- This object will receive messages from one layer and delegate to a proper object in the other layer.
- Assign responsibility as mentioned in my previous slide.

Controller - Monopoly

- The first object in the domain layer is called the controller for the system operation.



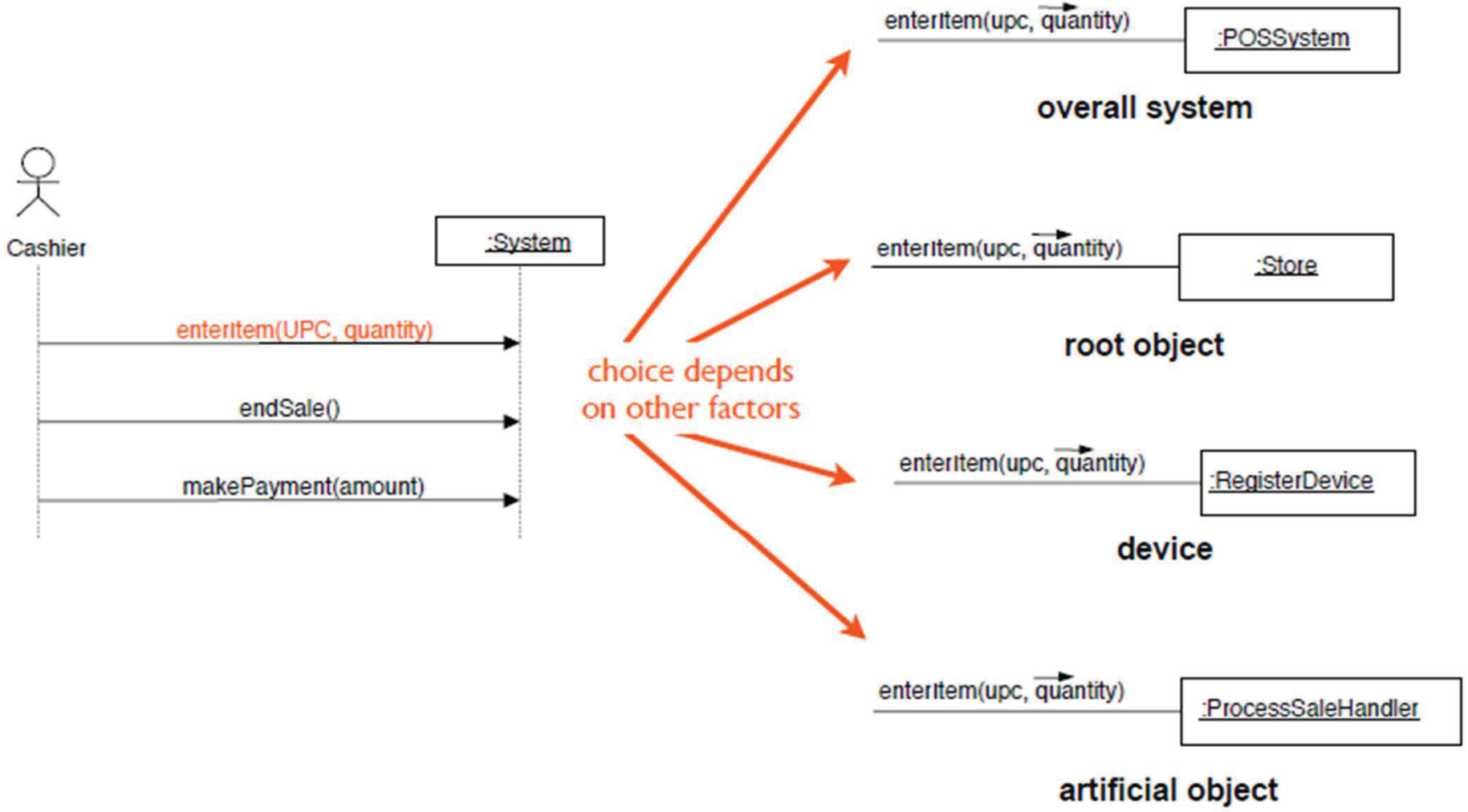
Controller - POS



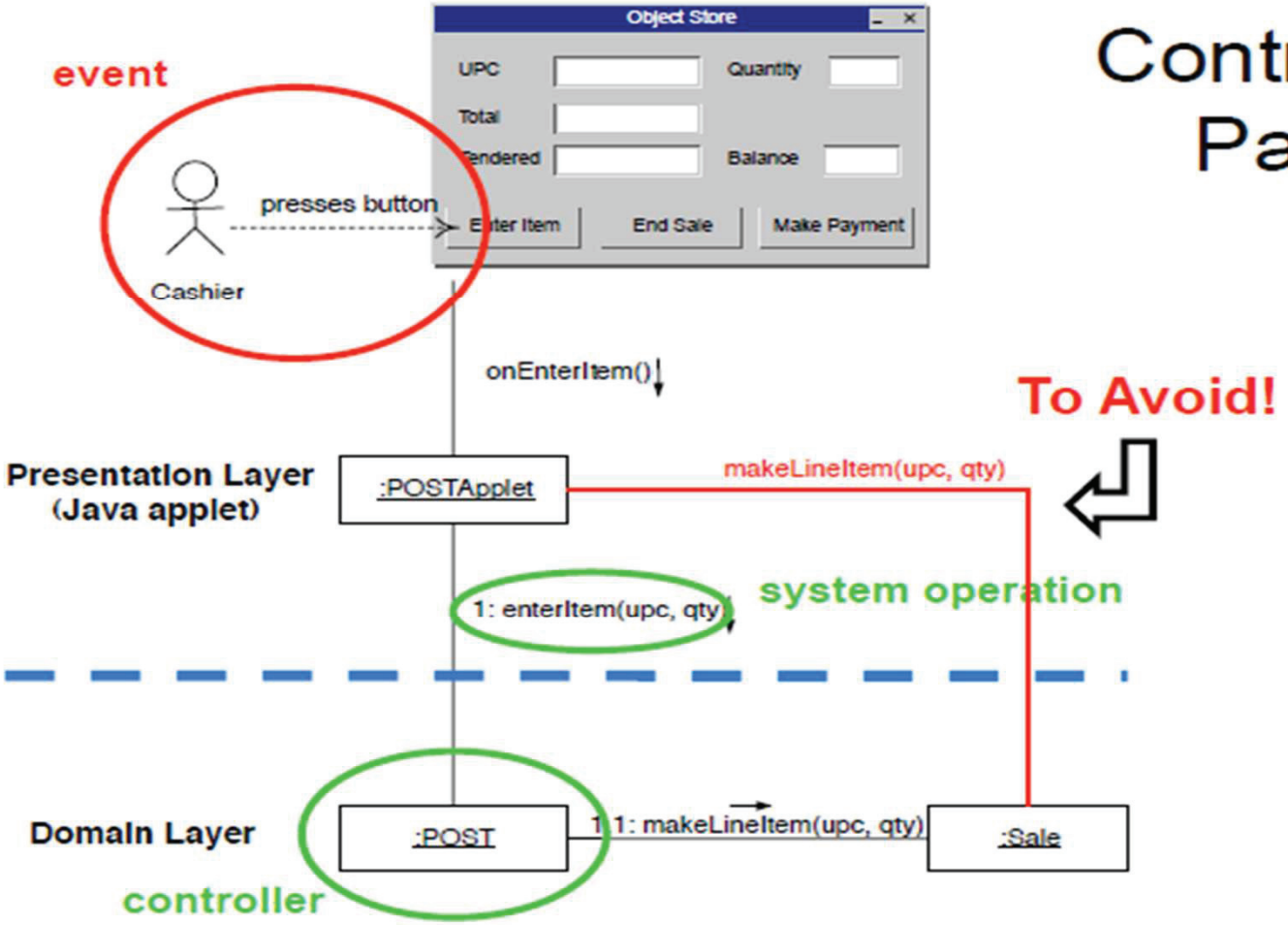
Controller - POS

- The controller is a kind of “façade” on the domain layer from the UI layer
- The controller does not perform the operation, it only delegates it.
- Assign the responsibility to a class C that represents one of the following choices:
- C is a façade controller: the overall system, a root object, the device that runs the software, or a major subsystem.
- C is a *use case or session controller*: it represents an artificial objects that handles all events from a use case or session

Controller - POS



Controller - POS



Low Coupling

- Coupling is a measure of how strongly one object is connected to, has knowledge of, or depends upon other objects.
- An element with low (or weak) coupling is not dependent on too many other elements
- An object A that calls on the operations of object B has coupling to B's services. When object B changes, object A may be affected.
- Supports:
  - low dependency between classes
  - low impact in a class of changes in other classes
  - high reuse potential

Low Coupling

Forms of Coupling from class X to class Y

- X has an attribute (reference or instance variable) of type Y
- An X object calls methods of Y object
- X is a direct or subclass of Y
- Y is an interface and X implements that interface (Java)

A class with high coupling is not desirable because

- Changes in other classes affect the class
- Harder to understand in isolation
- Harder to reuse because it use requires the additional presence of the classes on which it is dependent.

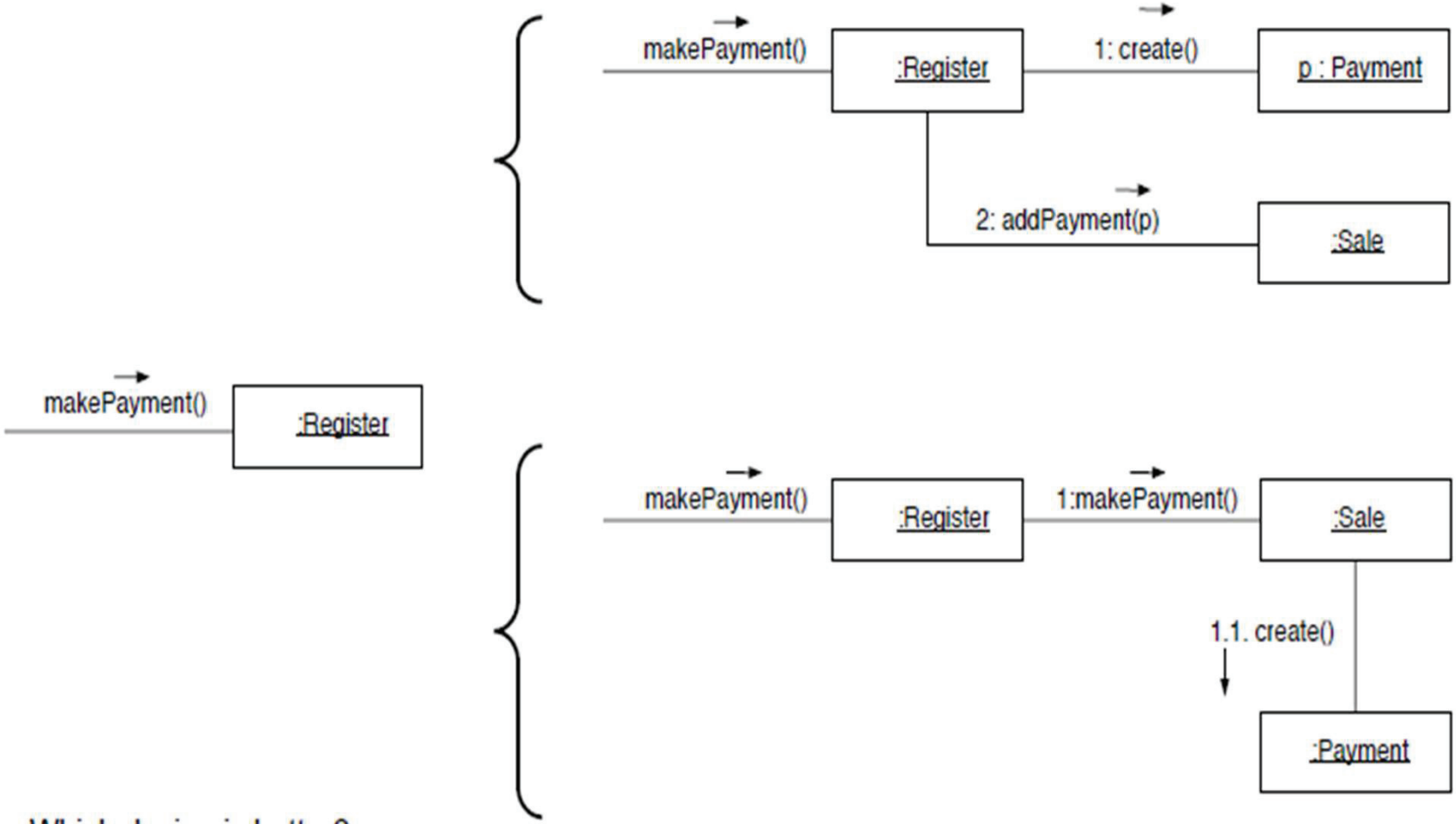
Low Coupling

- Problem: How to reduce the impact of change, increased reuse and support low dependency?
- Solution: Assign responsibilities so that (unnecessary) coupling remains low. Use this principle to evaluate alternatives.
- Design example: Making the payment, makePayment operation
- What class should be responsible for creating a payment instance and associating with the sale?

Low Coupling

- Solution 1: Since a register “records” a payment in a real-world domain, the creator and also expert pattern suggests Register as a candidate for creating the payment
- The assignment of responsibilities couples the register class to the payment class
- Solution 2:Sale creates the payment
- With this assignment coupling is lower than the first solution. There is no coupling between register and payment
- According to the low coupling pattern solution 2 is better than solution 1

Low Coupling



- Which design is better?

High Cohesion

- Cohesion is a measure that shows how strong responsibilities of a class are coupled
- Class with high cohesion has relatively small number of methods, with highly related functionality and does not do too much work
- It collaborates with other objects to share the effort if task is large.
- A class with low cohesion is not desirable:
  - Hard to understand
  - Hard to reuse
  - Hard to maintain
  - Affected by many changes

High Cohesion

Problem: How to keep objects focused, understandable and manageable?

Solution: Assign a responsibility so that cohesion remains high. Use this principle to evaluate alternatives

Design example: making the payment, makePayment operation

We look at the example used in low coupling pattern

What class should be responsible for creating a Payment instance and associating with the sale?

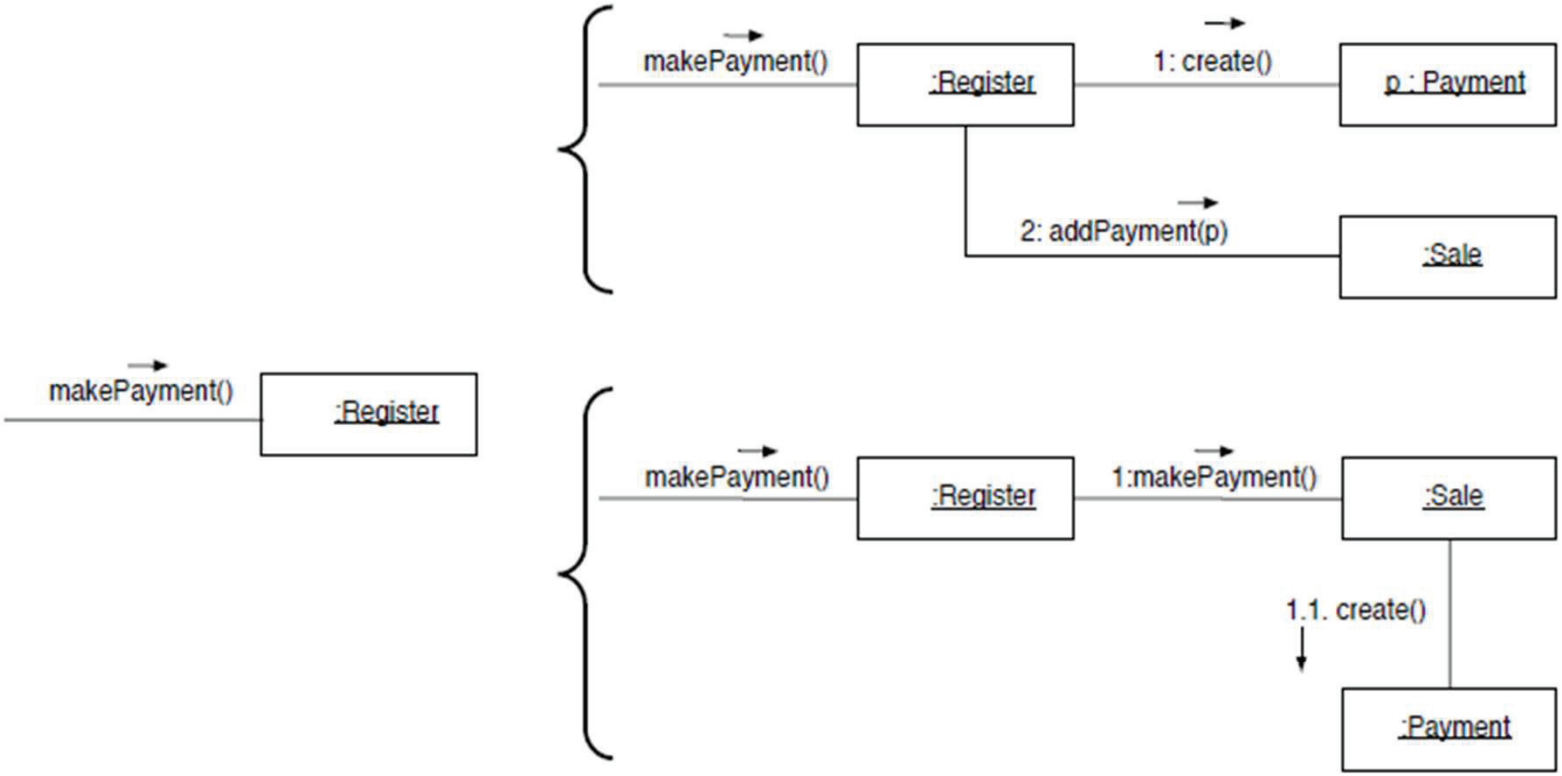
High Cohesion

Solution1: Since a register “records” a payment in the real-world domain, the creator and expert pattern suggests register as a candidate for creating the payment

Solution2: the single payment task does not make the register incohesive.

- If there are for example fifty system operations, all received by register and if register did the work related to each, it would become a bloated incohesive object.
- So it must delegate some of the work.
- As a result, sale creates the payment.

High Cohesion



- Cohesion: Object should have strongly related operations or responsibilities
- Reduce fragmentation of responsibilities (complete set of responsibility)
- To be considered in context => register cannot be responsible for all register-related tasks

Gang of Four – GOF Patterns

- Gang of four patterns are introduced by a book written by four authors.
- The book includes 23 design patterns
- Grouped in 3 categories:
- Creational patterns
  - Abstract the instantiation process
  - Make a system independent of how its objects are created, composed and represented
- Behavioral patterns
  - Concerns with algorithms and assignment of responsibilities between objects
- Structural patterns
  - How classes and objects are composed to form larger structures

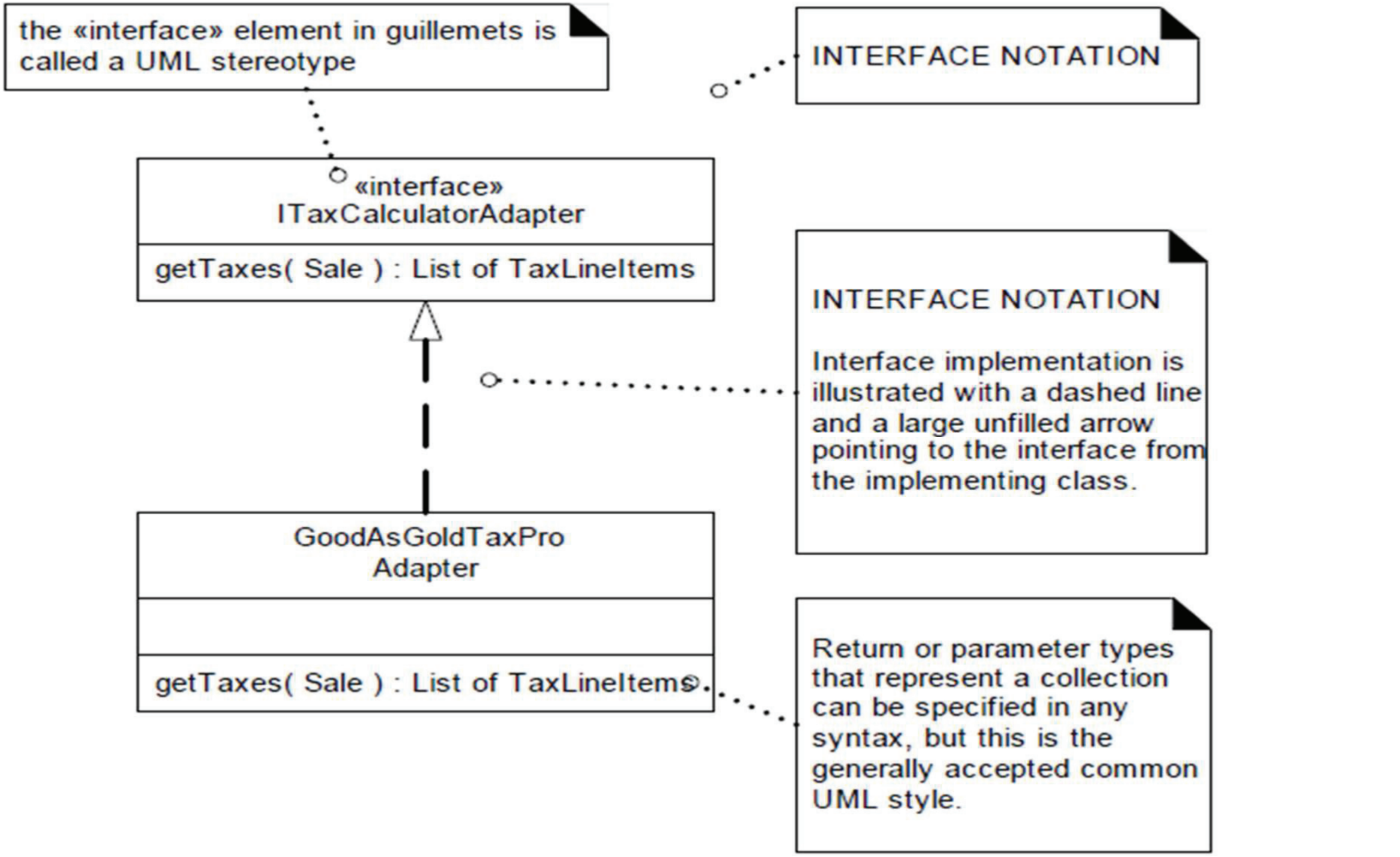
Gang of Four – GOF Patterns

- In the NextGen POS application, there are multiple external third-party tax calculators that must be supported (such as Tax-Master and Good-As-Gold Tax-Pro);
- The system needs to be able to integrate with different ones. Each tax calculator has a different interface, and so there is similar but varying behaviour to adapt to each of these external fixed interfaces or APIs.
- One product may support a raw TCP socket protocol, another may offer a SOAP interface, and a third may offer a Java RMI interface.
- What objects should be responsible for handling these varying external tax calculator?

Gang of Four – GOF Patterns

- Since the behaviour of calculator adaptation varies by the type of calculator, by Polymorphism we should assign the responsibility for adaptation to different calculator (or calculator adapter) objects themselves, implemented with a polymorphic *getTaxes* operation
- These calculator adapter objects are not the external calculators, but rather, local software objects that represent the external calculators, or the adapter for the calculator. By sending a message to the local object, a call will ultimately be made on the external calculator in its native API.
- Each *getTaxes* method takes the *Sale* object as a parameter, so that the calculator can analyse the sale. The implementation of each *getTaxes* method will be different: *TaxMasterAdapter* will adapt the request to the API of Tax-Master, and so on.

GOF Patterns – Interface notation



Structural Patterns – Adapter pattern

- Adapter pattern works as a bridge between two incompatible interfaces. This type of design pattern comes under structural pattern as this pattern combines the capability of two independent interfaces.
- This pattern involves a single class which is responsible to join functionalities of independent or incompatible interfaces. A real life example could be a case of card reader which acts as an adapter between memory card and a laptop. You plugin the memory card into card reader and card reader into the laptop so that memory card can be read via laptop.

Structural Patterns – Adapter pattern

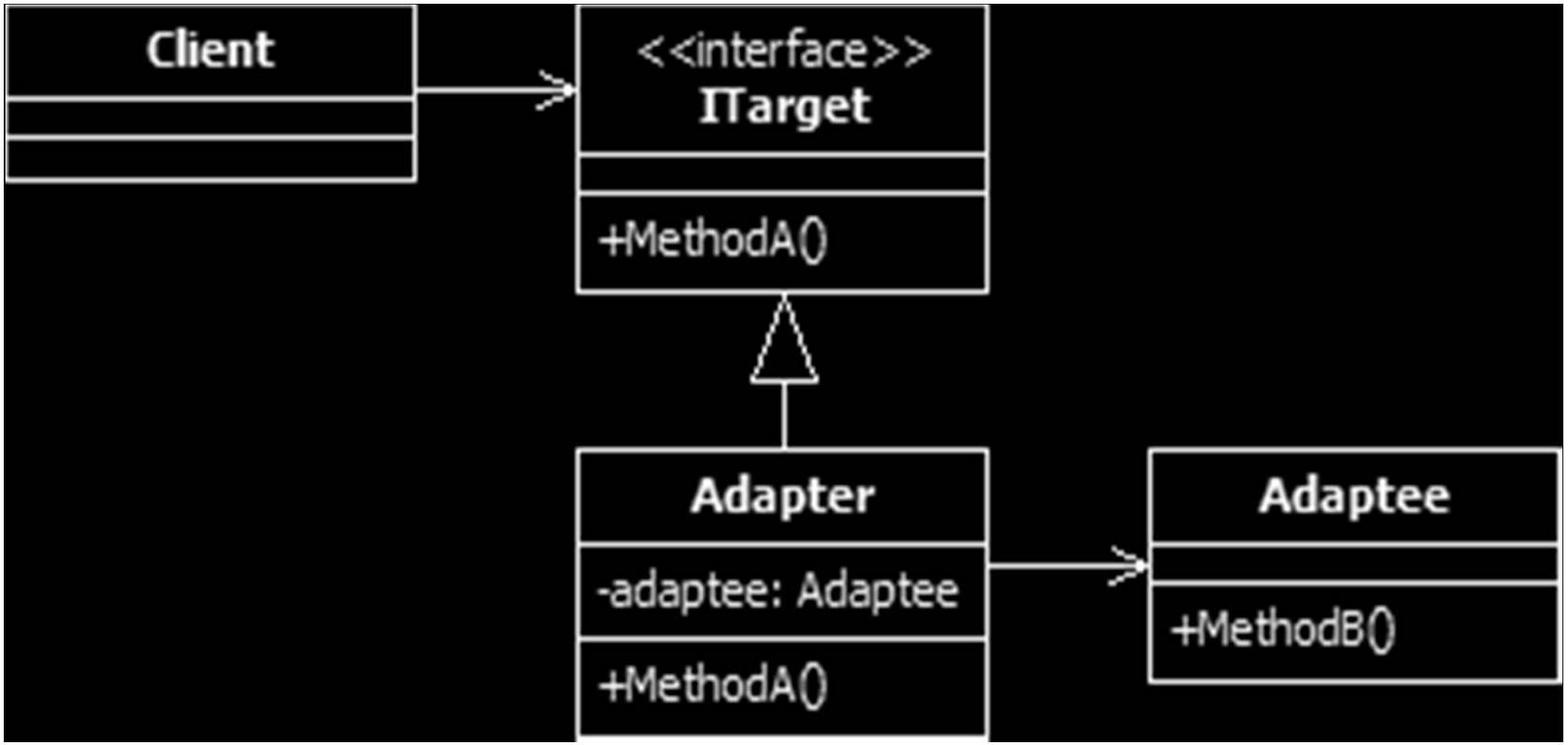
Context / Problem

- 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.
- This is a *structural* pattern as it defines a manner for creating relationships between classes. The adapter design pattern is used to provide a link between two otherwise incompatible types by wrapping the "adaptee" with a class that supports the interface required by the client.

Structural Patterns – Adapter pattern



Structural Patterns – Adapter pattern

- The UML class diagram describes an implementation of the adapter design pattern. The items in the diagram are described below:
- **Client**. The client class is that which requires the use of an incompatible type. It expects to interact with a type that implements the ITarget interface. However, the class that we wish it to use is the incompatible *Adaptee*.
- **ITarget**. This is the expected interface for the client class. Although shown in the diagram as an interface, it may be a class that the adapter inherits. If a class is used, the adapter must override its members.

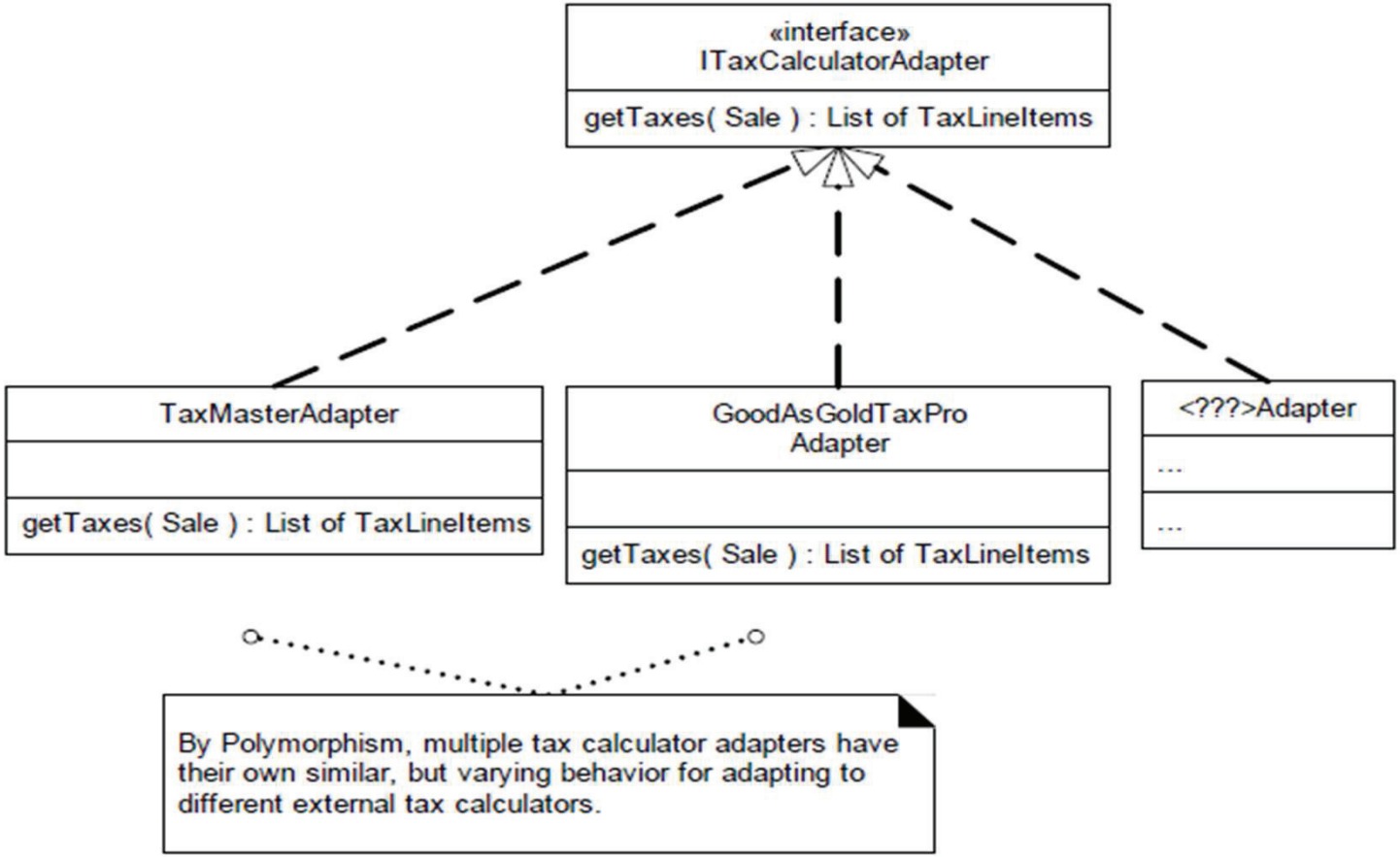
Structural Patterns – Adapter pattern

- **Adaptee**. This class contains the functionality that is required by the client. However, its interface is not compatible with that which is expected.
- **Adapter**. This class provides the link between the incompatible Client and Adaptee classes. The adapter implements the ITarget interface and contains a private instance of the Adaptee class. When the client executes MethodA on the ITarget interface, MethodA in the adapter translates this request to a call to MethodB on the internal Adaptee instance.

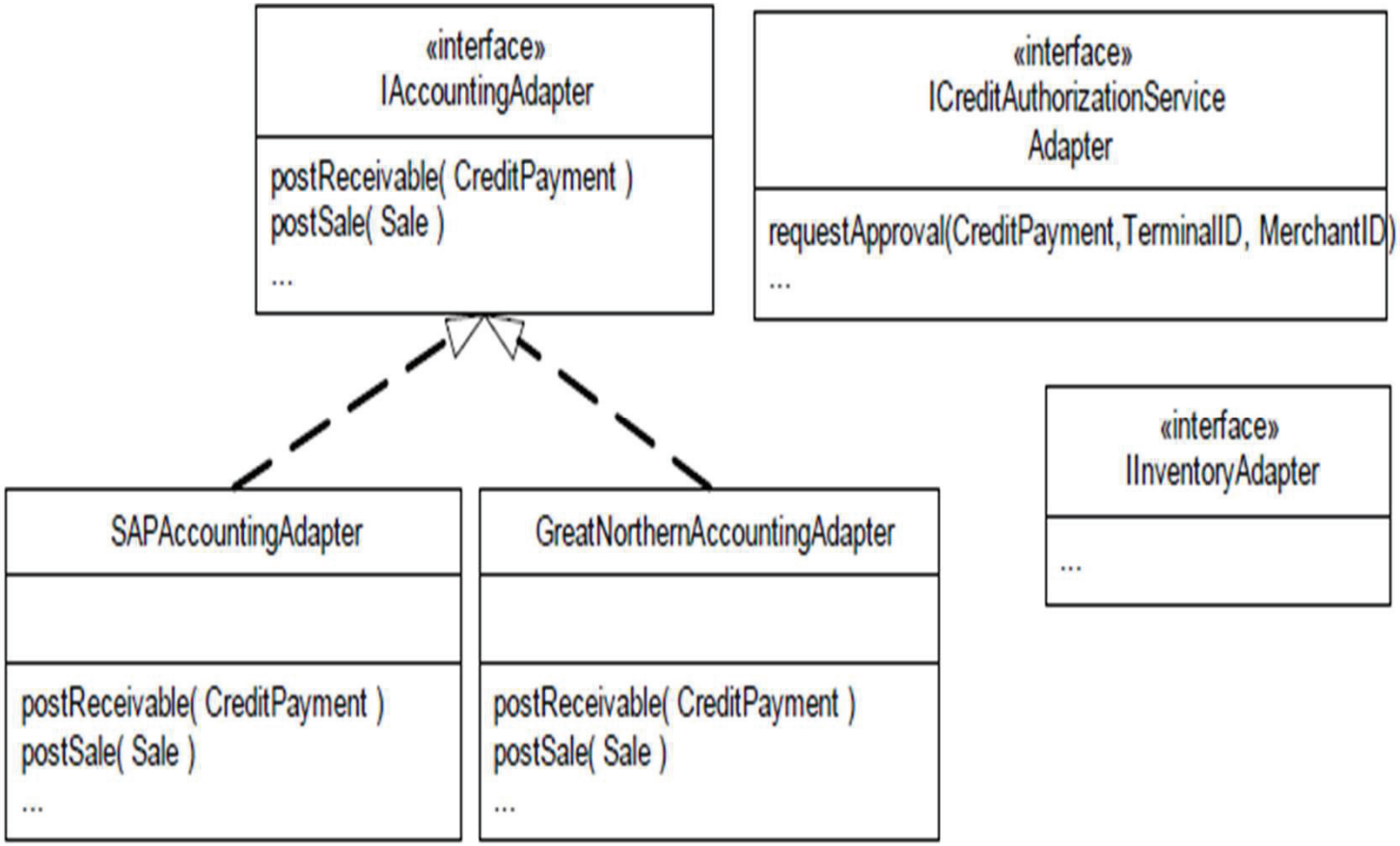
Structural Patterns – Adapter pattern

- Suppose that NextGen POS System must support different third party external tax calculators with different interfaces
- Depending upon some conditions, the sale class will sometime connect to the “Tax Master” program and sometimes to the “Good as Gold Tax Pro” program to calculate the tax.
- In future other programs may also be added.
- Sale class always sends the getTaxes message to calculate the tax.
- The current adaptor converts and directs the call to the tax calculator program

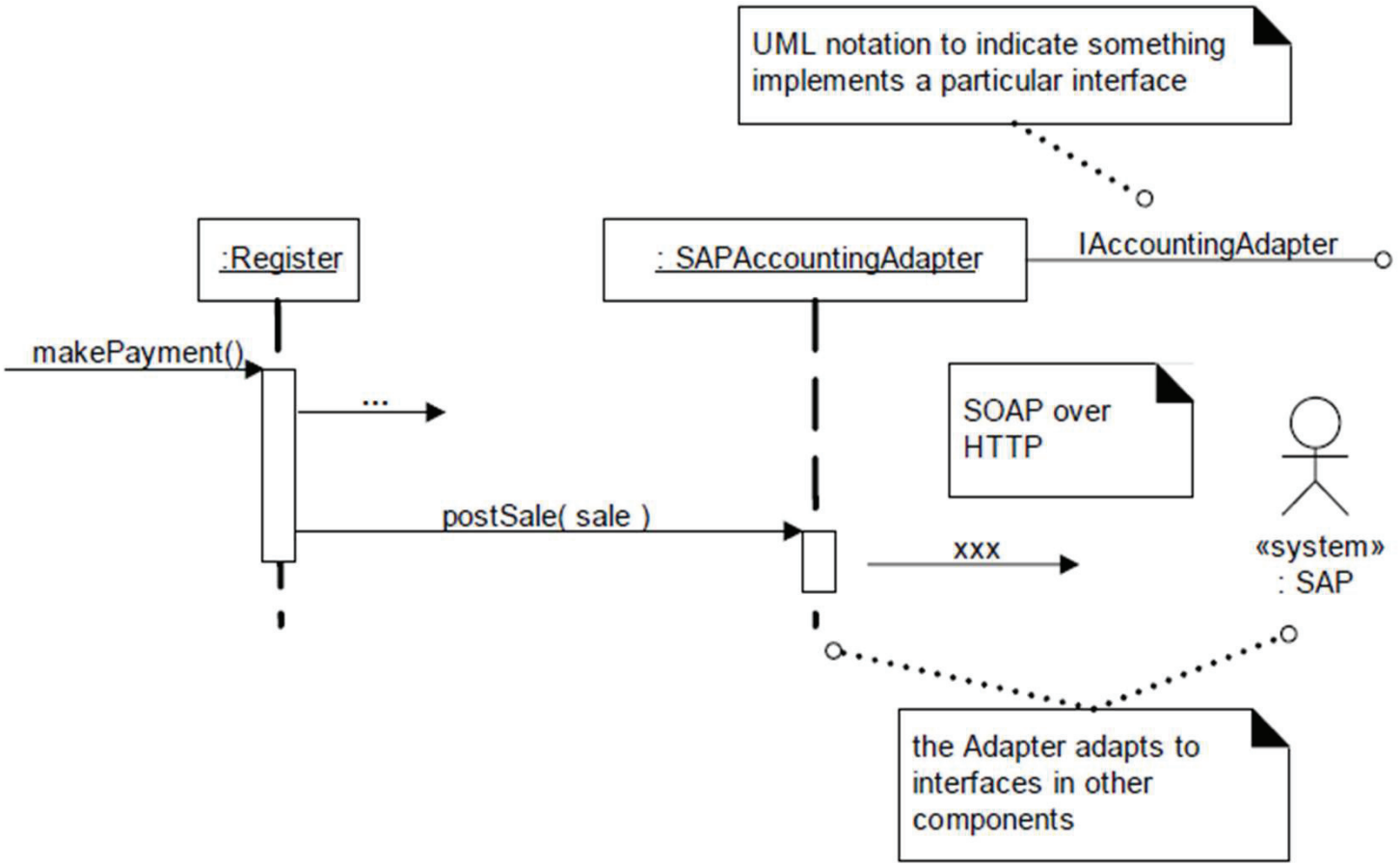
Structural Patterns – Adapter pattern



Structural Patterns – Adapter pattern



Structural Patterns – Adapter pattern



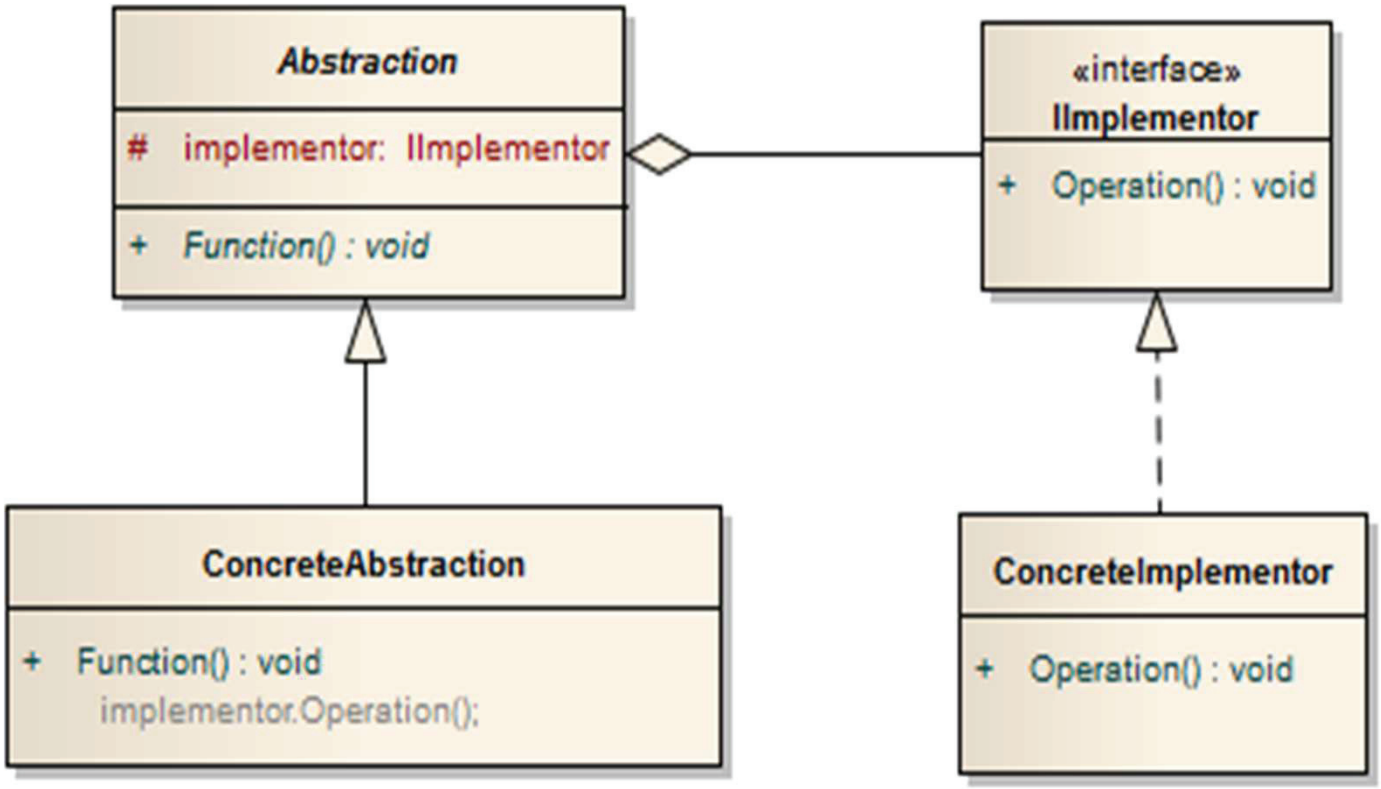
Structural Patterns – Adapter pattern

- Particular adapter instance will be instantiated for the chosen external service, 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
- the use of an interface "lollipop" to indicate that the *SAP Accounting Adapter* instance implements a noteworthy interface

Structural Patterns – Bridge pattern

- This is a *structural* pattern as it defines a manner for creating relationships between classes or entities. The bridge pattern is used to separate the abstract elements of a class from the implementation details.
- The bridge design pattern allows you to separate the abstraction from the implementation.
- In the **bridge pattern**, there are 2 parts - the first part is the Abstraction, and the second part is the Implementation.
- The **bridge pattern** allows the Abstraction and the Implementation to be developed independently, and the client code can access only the Abstraction part without being concerned about the Implementation part.

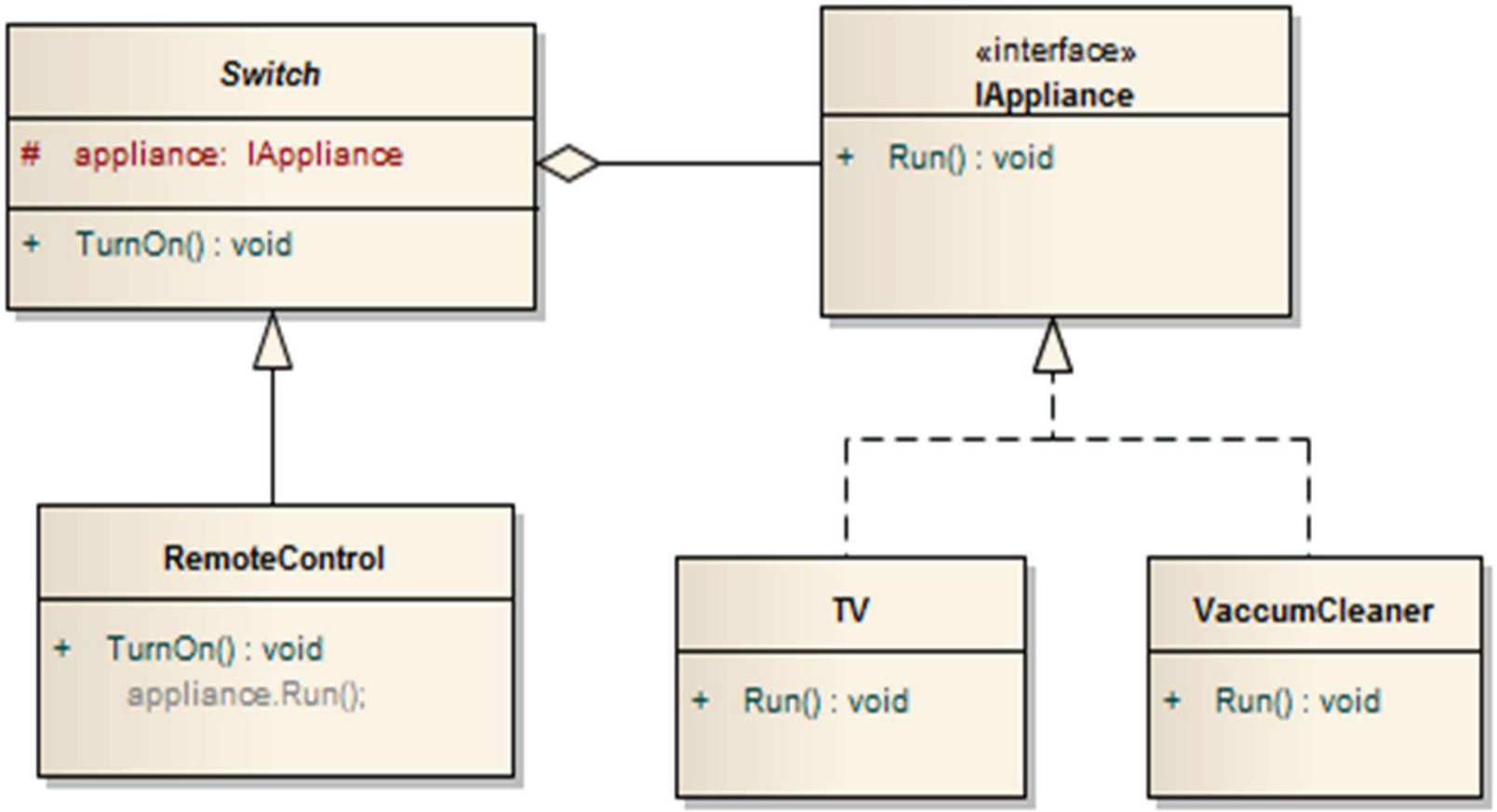
Structural Patterns – Bridge pattern



Structural Patterns – Bridge pattern

- *Abstraction* is the abstract class of the *ConcreteAbstraction* class.
- It defines the *Function* method for the client code to call.
- The protected *implementor* variable holds the reference to the object that performs the implementation.
- *ConcreteAbstraction* is the concrete class that is inherited from the *Abstraction* class.
- *Implementor* is the interface that all the implementation classes must implement.
- *ConcreteImplementor* is the concrete class that performs the implementation.

Structural Patterns – Bridge pattern



Structural Patterns – Bridge pattern

- For example, inside a house, there are appliances that you can turn on or off, such as the floor lamp, the TV, and the vacuum cleaner.
- There are different ways to turn the appliance on or off, such as using the on/off switch, the pull switch, or using a remote control.
- The concept of turning the appliance on or off is the Abstraction part in the **bridge pattern**, and the user only needs to know the Abstraction part. This is the first part of the **bridge pattern**.
- Left side represents controls that can turn appliances on or off, such as the on/off switch, the pull switch, or the remote control, while the right side represents the actual appliances that performs the action, such as the TV or the Vacuum Cleaner.

Structural Patterns – Bridge pattern

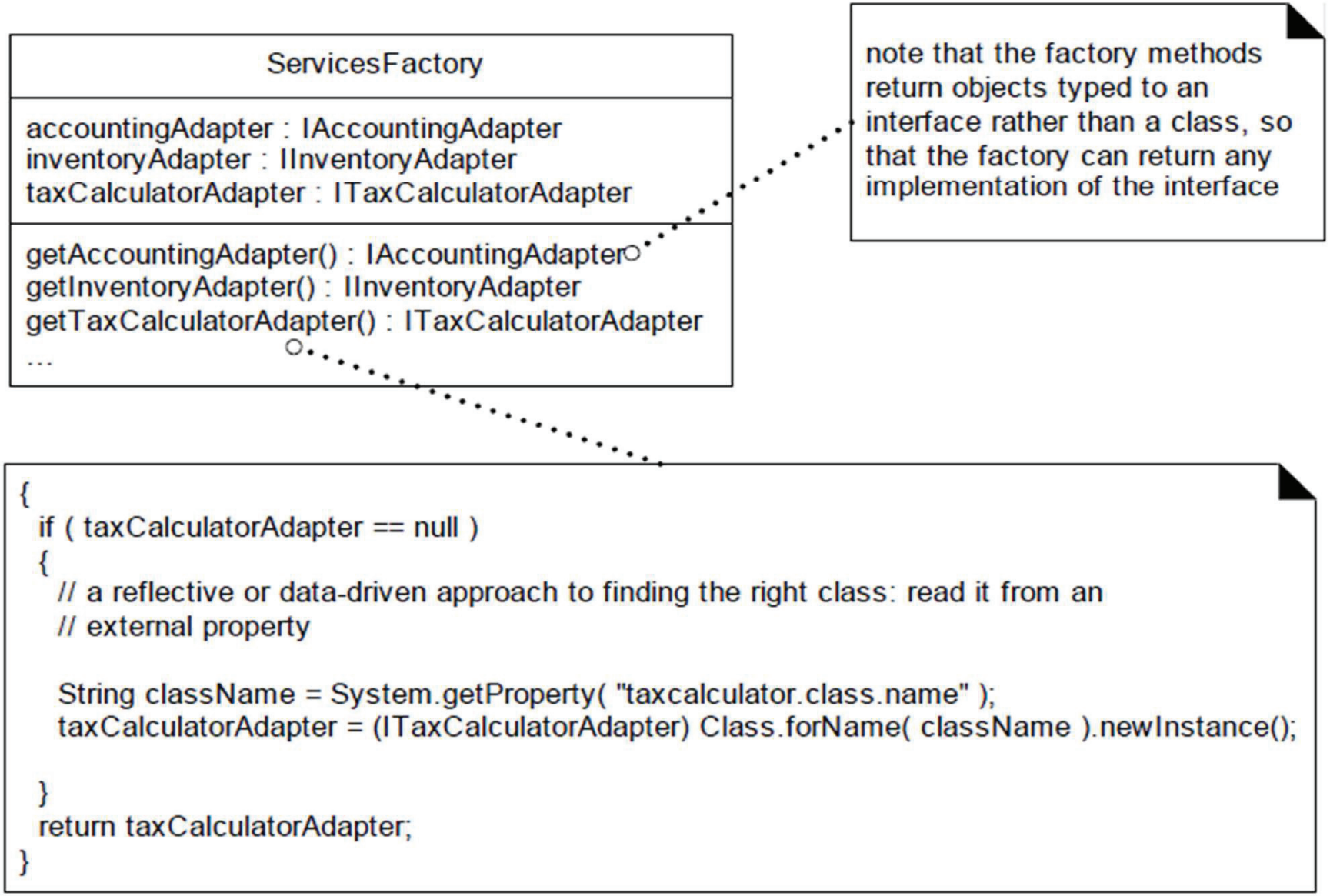
- Another example of the **bridge pattern** is the copy and paste function in many applications.
- The copy and paste function is the abstraction, where the user only needs to know how to use it, and the actual action of transferring the information into the memory and transferring the information onto the application is the implementation.
- The key benefit of the bridge design pattern is that it allows you to develop the Abstraction and the Implementation parts independently.
- It also cuts down on the number of classes that you need to create to fulfil all the possible combinations of the Abstractions (user interface concepts) and the Implementations (actual actions behind the scene).

Creational Patterns – Factory pattern

Name: Factory

- Problem: Who should be responsible for creating families of related objects
- Solution: Use a factory object to handle all creation responsibility
- This pattern provides one of the best ways to create an object.
- Defines an interface for creating objects but let sub-classes decide which of those instantiate.
- Enables the creator to defer Product creation to a sub-class.
- If some domain object (for example Sale) creates them as the creator pattern suggests, we will encounter the following problems:
- The domain objects Sale must be aware of external systems (coupling)
- Adding or removing an external calculator will affect the sale
- Change in rules about adaptor usage will affect sale
- This responsibility lowers the cohesion of the domain object because connectivity with external components is not its main job (separation of concerns)

Creational Patterns – Factory pattern

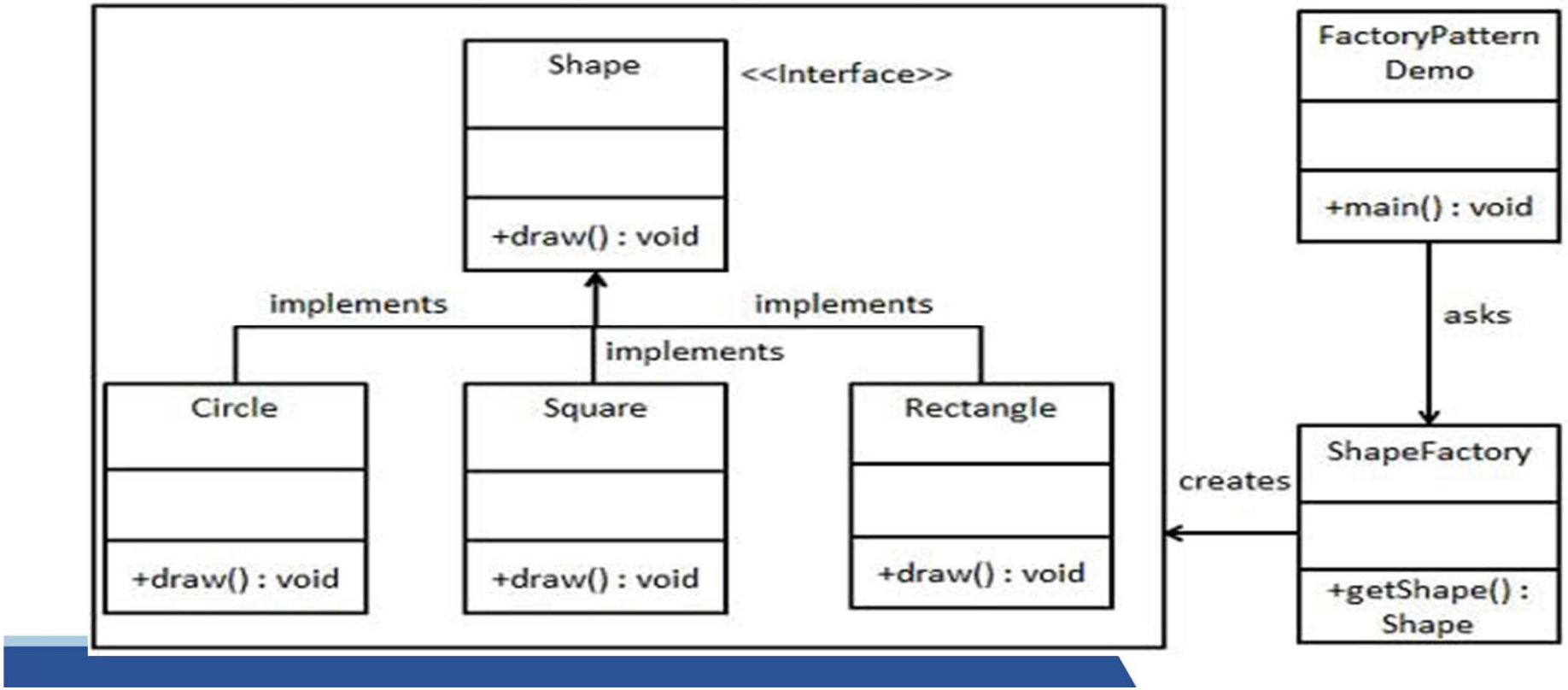


Creational Patterns – Factory pattern

- For example in the NextGen POS system to create adapters a factory object “ServicesFactory” can be defined
- When the domain object Sale needs to access an external tax calculator it will call the getTaxCalculatorAdapter method of the ServicesFactory object.
- The method of factory will determine the appropriate adaptor according to the current conditions and will create the adaptor and return its address to the domain object.
- Advantages:
  - The sale object does not know from which external calculator it is being served
  - If a new adaptor is added to the system or the creation logic changes only the factory objects is affected.

Creational Patterns – Factory pattern

- Create a *Shape* interface and concrete classes implementing the *Shape* interface. A factory class *ShapeFactory* is defined as a next step.
- *FactoryPatternDemo*, will use *ShapeFactory* to get a *Shape*object. It will pass information (*CIRCLE / RECTANGLE / SQUARE*) to *ShapeFactory* to get the type of object it needs.



Creational Patterns – Factory pattern

1. Create an interface shape.java

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

2. Create concrete classes implementing the same interface

```
public class Rectangle implements Shape
{
public void draw()
         {
        System.out.println("Inside rectangle::draw() method.");
        }
}
```

Creational Patterns – Factory pattern

3. Create a Factory to generate object of concrete class based on given information.

```
public class ShapeFactory
{ //use getShape method to get object of type shape
public Shape getShape(String shapeType){
If (shapeType == null){
 return null;
}
If (shapeType.equalsIgnoreCase("CIRCLE")){
return new Circle();
} else if(shapeType.equalsIgnoreCase("RECTANGLE")){
return new Rectangle();
} else if(shapeType.equalsIgnoreCase("SQUARE")){
 return new Square();
} return null; } }
```

Creational Patterns – Factory pattern

4. Use the Factory to get object of concrete class by passing an information such as type.

```
public class FactoryPatternDemo {
public static void main(String[] args)
{
 ShapeFactory shapeFactory = new ShapeFactory();
//get an object of Circle and call its draw method. Shape shape1 = shapeFactory.getShape("CIRCLE"); //call draw method of Circle shape1.draw();
//get an object of Rectangle and call its draw method.
Shape shape1 = shapeFactory.getShape(“Rect");
//call draw method of Rectangle
shape2.draw();…..
```

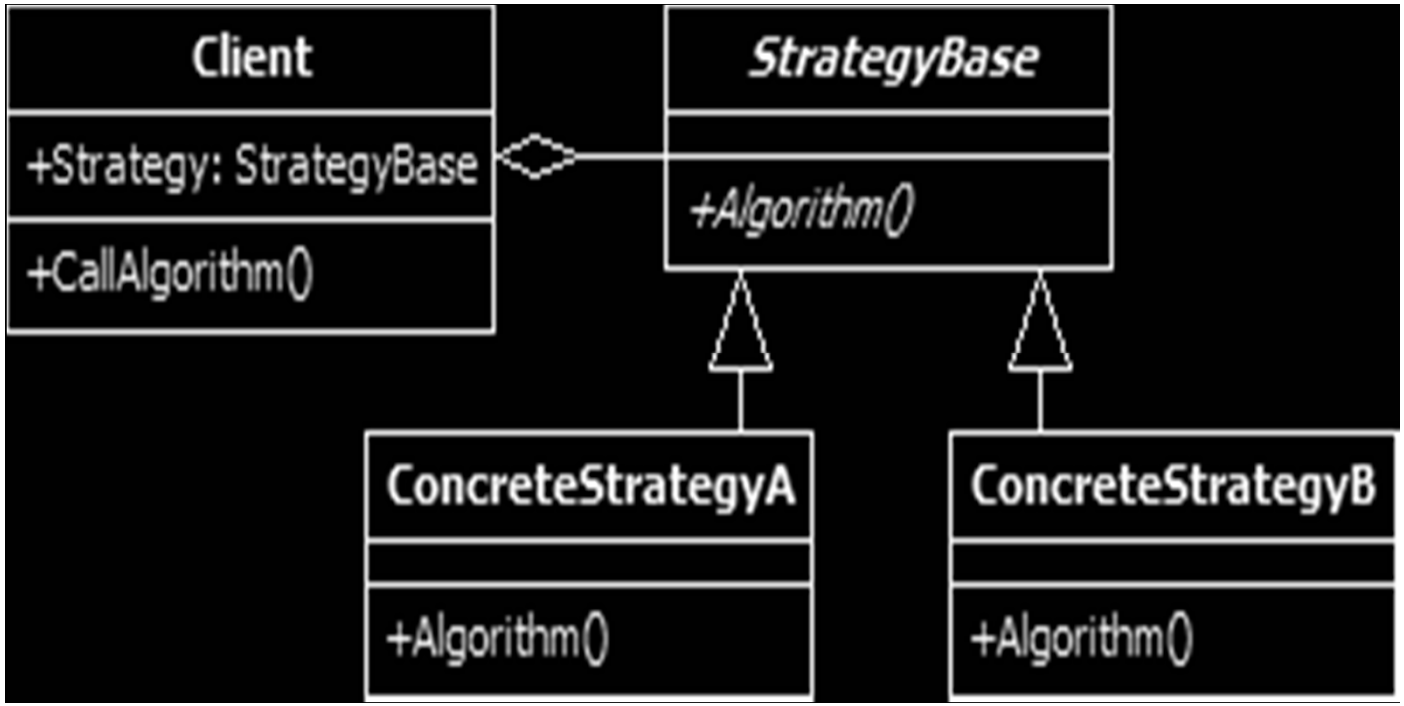
Behavioral Patterns – Strategy pattern

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

Solution: Define each algorithm / policy / strategy in a separate class with a common interface

- This is a *behavioural* pattern as it defines a manner for controlling communication between classes or entities.
- The strategy pattern is intended to provide a means to define a family of algorithms, encapsulate each one as an object, and make them interchangeable.
- Strategy lets the algorithm vary independently from clients that use it.
- The algorithm/behaviour encapsulated in its own class is called a strategy

Behavioral Patterns – Strategy pattern



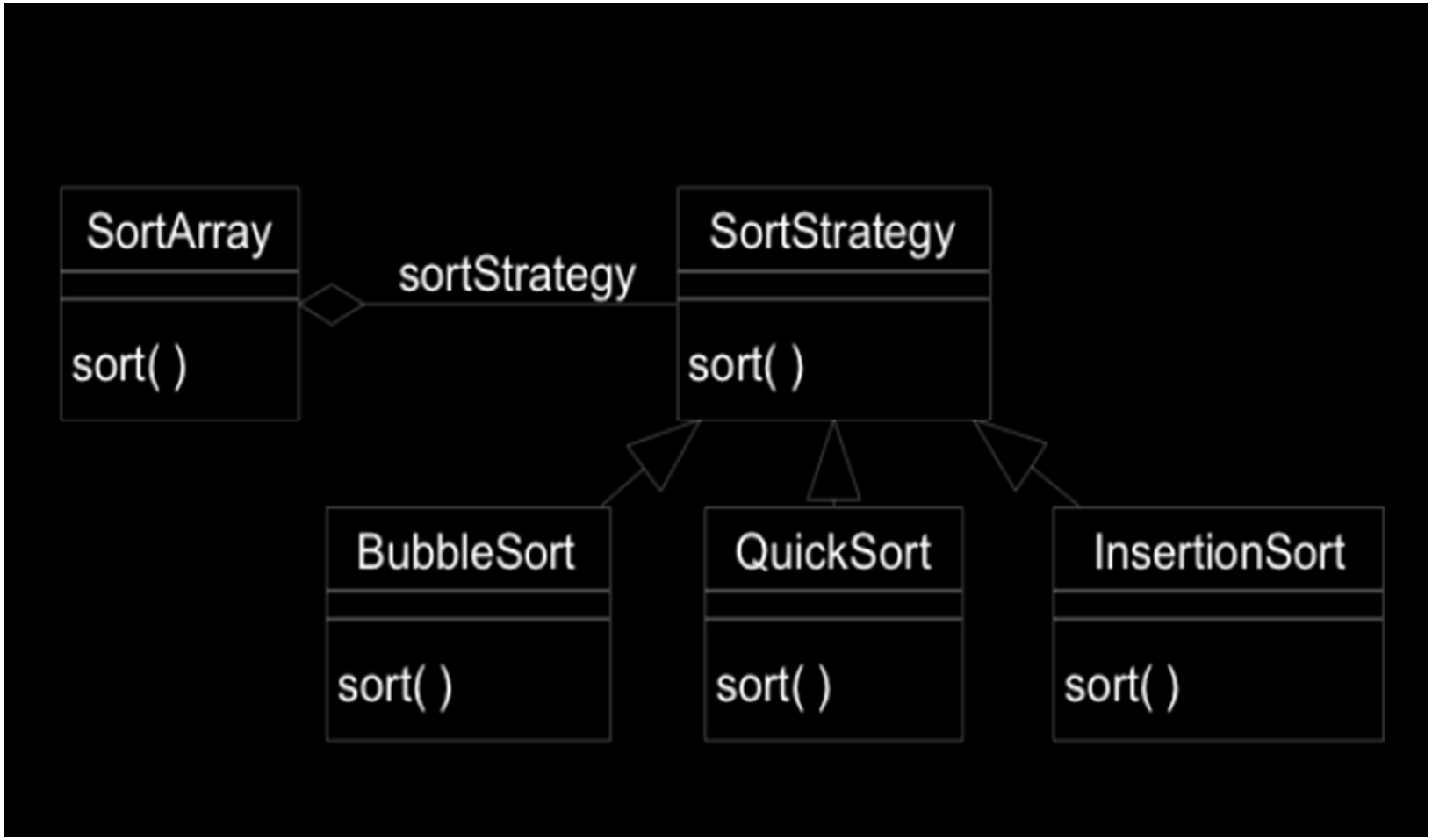
Strategy pattern

- The UML class diagram describes an implementation of the strategy design pattern. The items in the diagram are described below:
- **Client**. This class is the user of an interchangeable algorithm. The class includes a property to hold one of the strategy classes. This property will be set at run-time according to the algorithm that is required.
- **StrategyBase**. This abstract class is the base class for all classes that provide algorithms. In the diagram the class includes a single method. However, there is no reason why a number of properties and methods may not be included. This class may be implemented as an interface if it provides no real functionality for its subclasses.
- **ConcreteStrategy A/B**. The concrete strategy classes inherit from the StrategyBase class. Each provides a different algorithm that may be used by the client.

Strategy pattern – An example

- You have an array and at run-time you want to decide which sort algorithm to use.
- Encapsulate each different sort algorithm using the strategy pattern.

Strategy pattern – Class diagram



Strategy pattern – Example

- SortStrategy: This is an interface to describe the individual algorithms.
- public interface SortInterface

```
{
 public void sort(double[] list);
}
```

- All the sorting algorithms we will use will include a method called sort that takes a list of double.

Strategy pattern – POS

- A design problem in POS system is the complex pricing policy such as discount for the day, senior citizen discounts and so on.
- The pricing strategy also called as a rule, policy or algorithm for a sale can vary.
- For example, Monday it may be 10% and Thursday 5% off all sales.
- These pricing strategies (algorithms) are variations of the getTotal() responsibility (behavior) of the Sale class.
- To add all these algorithms into getTotal() method of the sale using if-then or switch-case statements will cause coupling and cohesion problems.
- All changes in pricing strategies will affect the sale

Strategy pattern – POS

- 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() 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.

Behavior pattern -Observer pattern

Name: Observer

- Problem: You need to notify a varying list of objects that an event has occurred
- Solution: Define a “subscriber” or “listener” interface. Subscriber implement this interface. The publisher can dynamically register subscribers who are interested in an event and notify them when an event occurs.
- POS requirement: Adding the ability for a GUI window to refresh its display of the sale
- total when the total changes

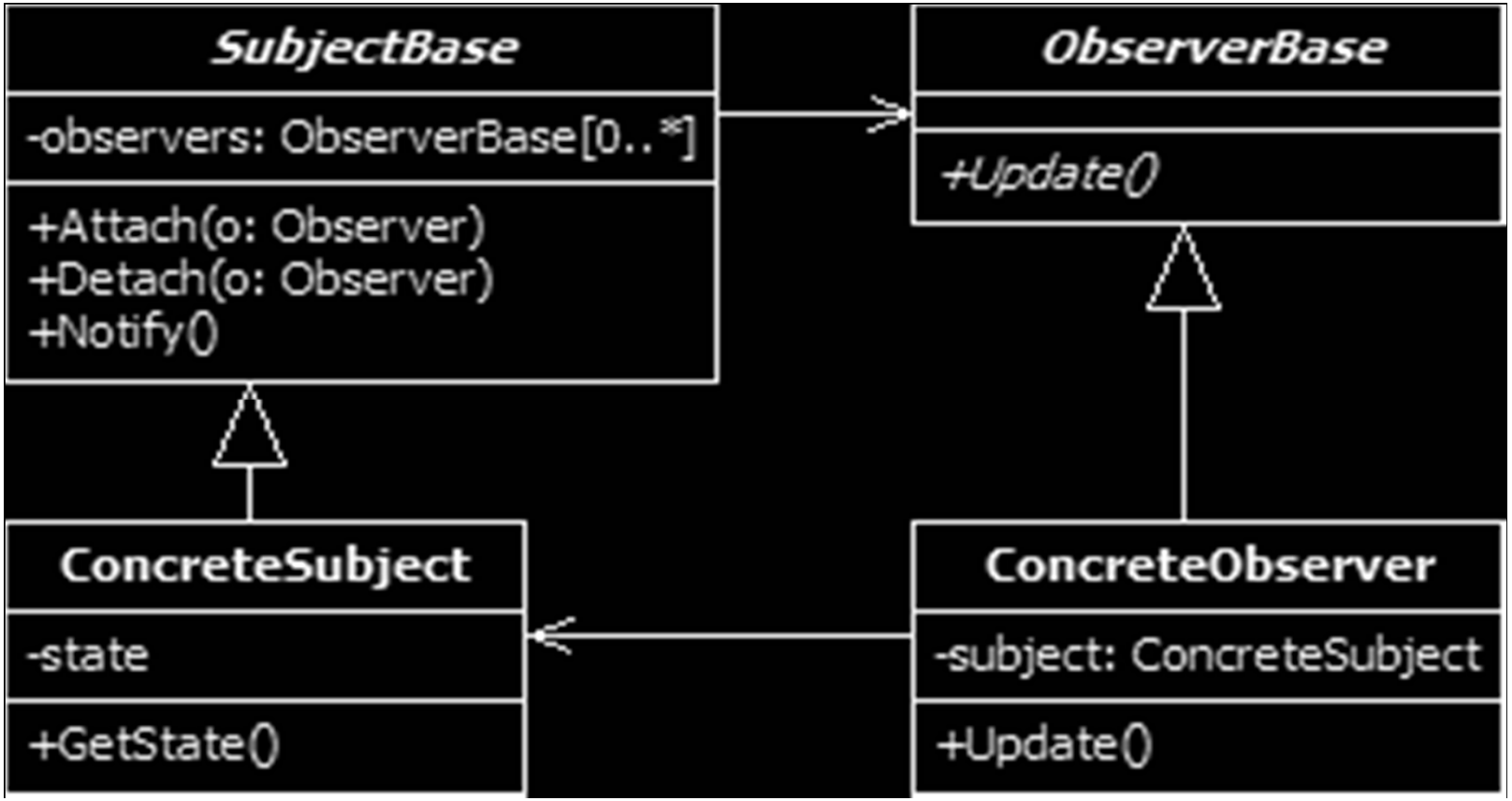
Solution:

- When the Sale changes its total, the **Sale** object sends a message to a window, asking it to refresh its display

Behavior pattern -Observer pattern

- The observer pattern is used to allow a single object, known as the *subject*, to publish changes to its state.
- Many other *observer* objects that depend upon the subject can subscribe to it so that they are immediately and automatically notified of any changes to the subject's state.
- The subject holds a collection of observers that are set only at run-time.
- Each observer may be of any class that inherits from a known base class or implements a common interface. The actual functionality of the observers and their use of the state data need not be known by the subject

Behavior pattern -Observer pattern



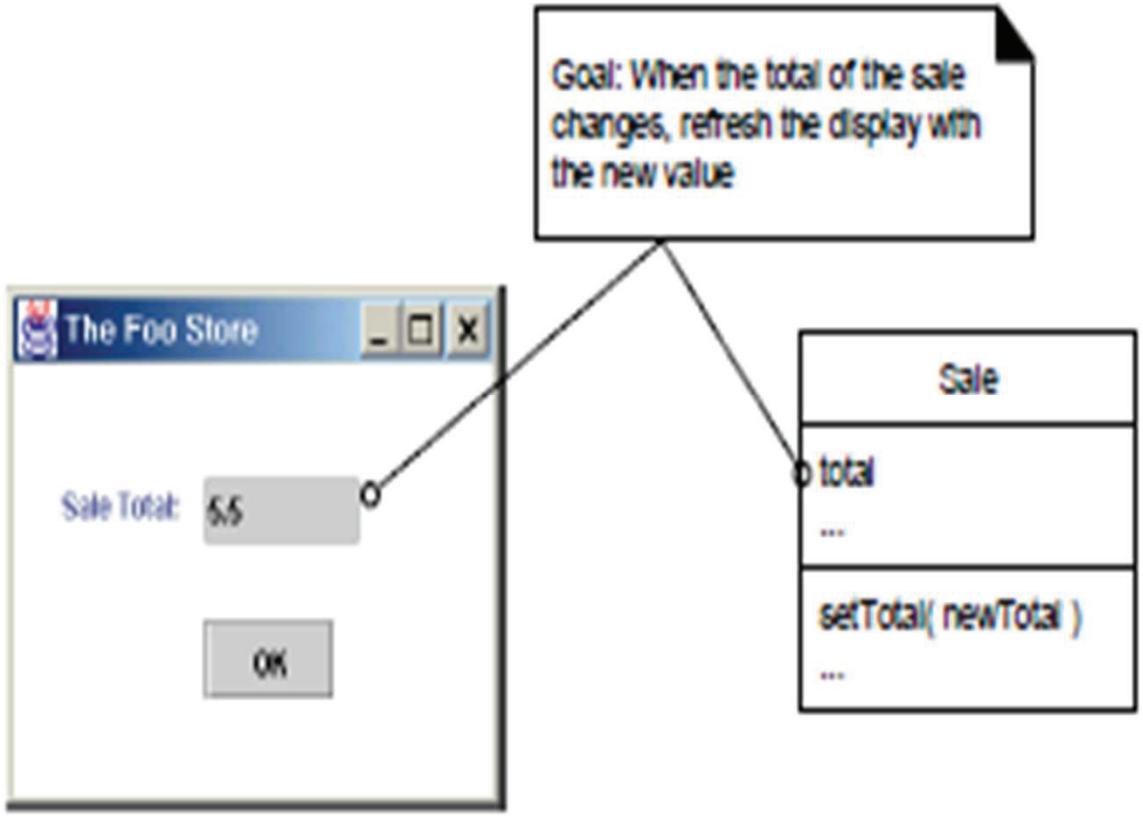
Behavior pattern -Observer pattern

- The UML class diagram describes an implementation of the observer design pattern. The items in the diagram are described below:
- **SubjectBase**. This is the abstract base class for concrete subjects. It contains a private collection of the observers that are subscribed to a subject and methods to allow new subscriptions to be added and existing ones to be removed. It also includes a method that can be called by concrete subjects to notify their observers of state changes. This *Notify*method loops through all of the registered observers, calling their *Update* methods.

Behavior pattern -Observer pattern

- **ConcreteSubject**. Each concrete subject maintains its own state. When a change is made to that state, the object calls the base class's Notify method to indicate this to all of its observers. As the functionality of the observers is unknown, the concrete subjects also provide the means for the observers to read the updated state, in this case via a GetState method.
- **ObserverBase**. This is the abstract base class for all observers. It defines a method to be called when the subject's state changes. In many cases this *Update* method will be abstract, in which case you may decide to implement the base class as an interface instead.
- **ConcreteObserver**. The concrete observer objects are the subscribers that react to changes in the subject's state. When the Update method for an observer is called, it examines the subject to determine which information has changed. It can then take appropriate action.

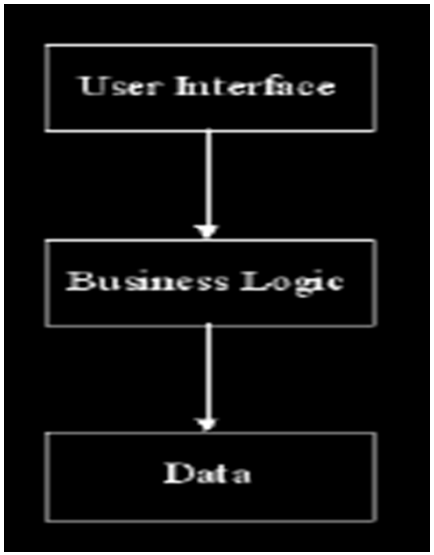
Behavior pattern -Observer pattern



Behavior pattern -Observer pattern

Problem with solution:

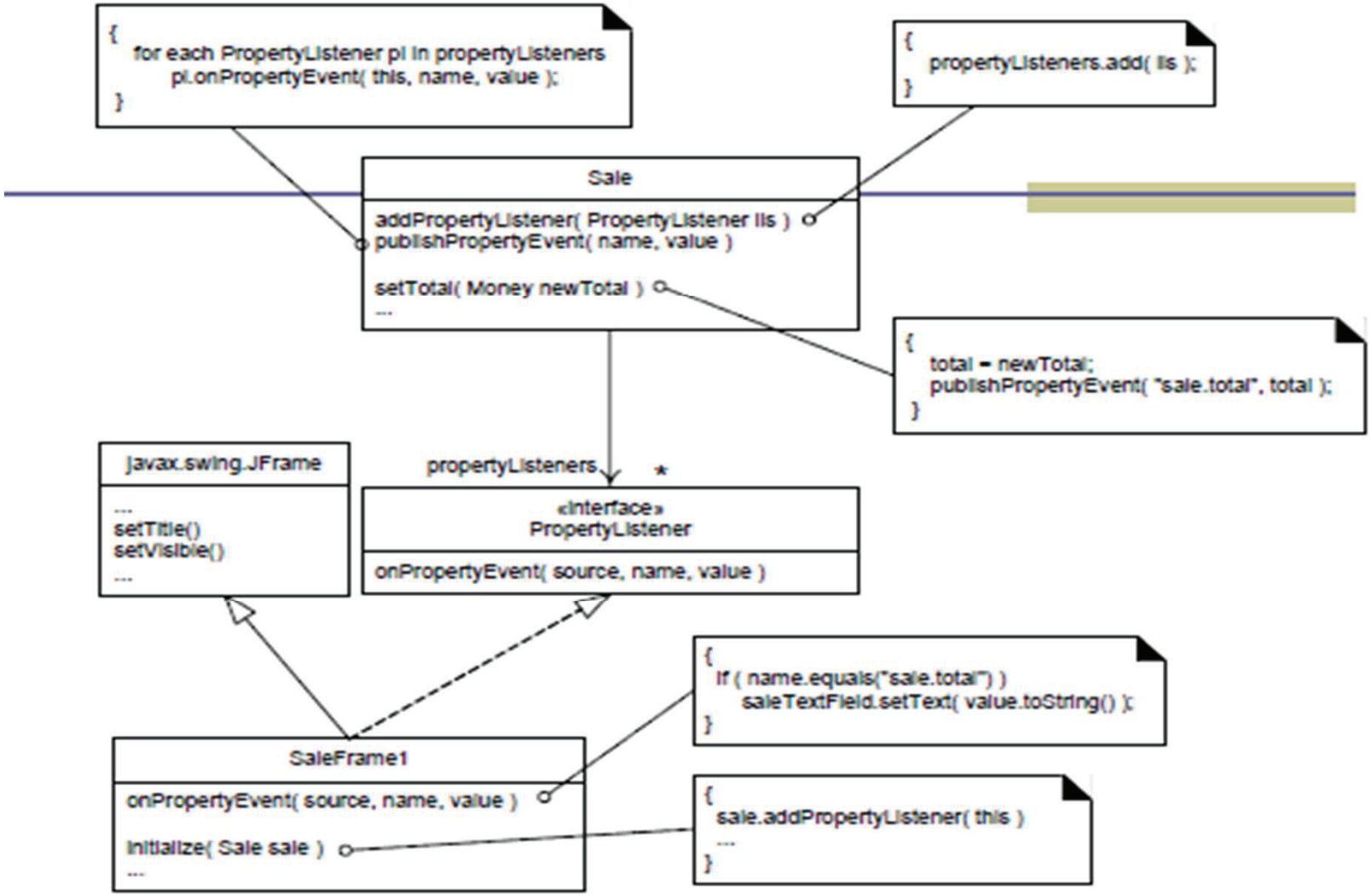
- Violation of layer dependency principle
- Specially, violate the Model-View Separation principle
- Model-View Separation principle
- Do not connect or couple non-UI objects directly to UI objects
- Do not put application logic in the UI



Observer pattern - POS

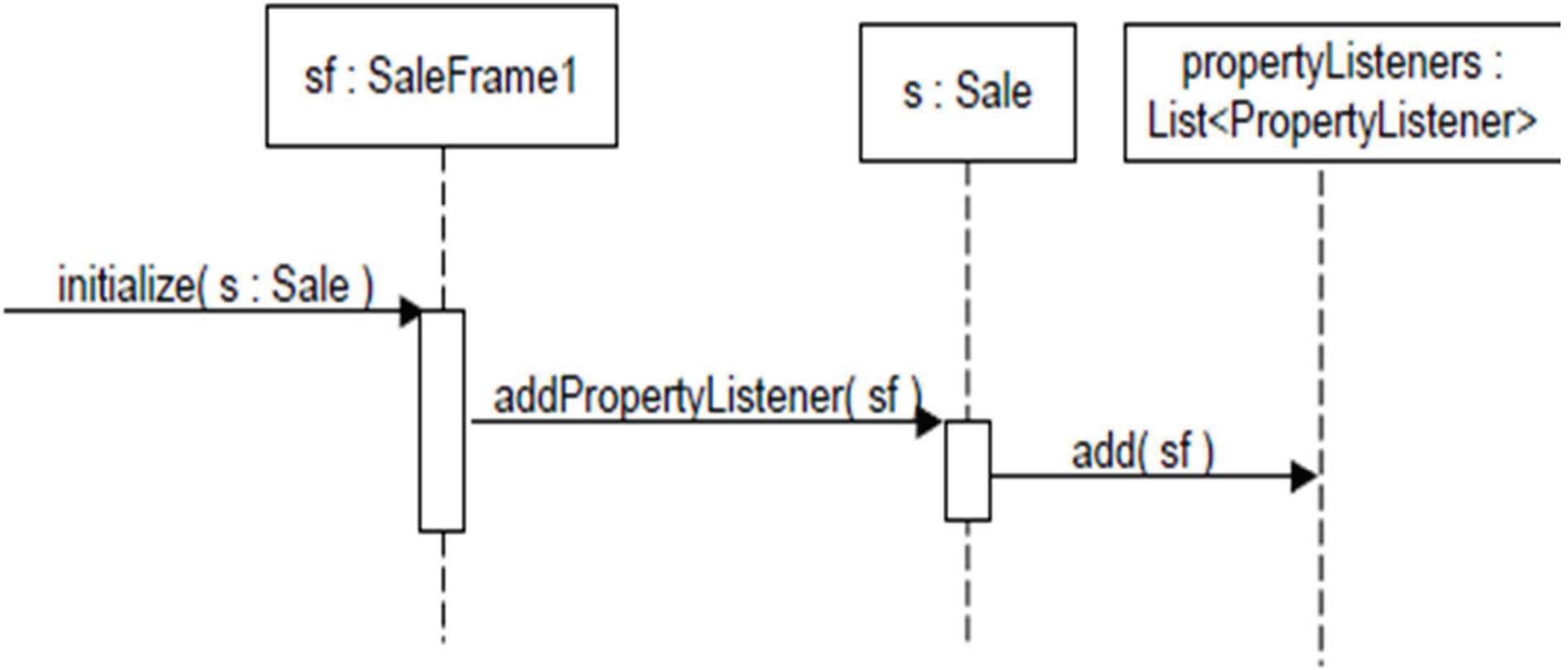
- An interface is defined – **PropertyListener** with the operation **onPropertyEvent**
- Define an UI object to implement the interface -- **SalesFrame1**
- When the SalesFrame1 window is initialized, pass it the Sale instance from which it is displaying the total
- The **SaleFrame1** window registers or subscribes to the **Sale** instance for notification of “property events”, via the **addPropertyListener** message
- The **Sale** instance, once its total changes, iterates across all subscribing **PropertyListeners**, notifying each.

Observer pattern - POS

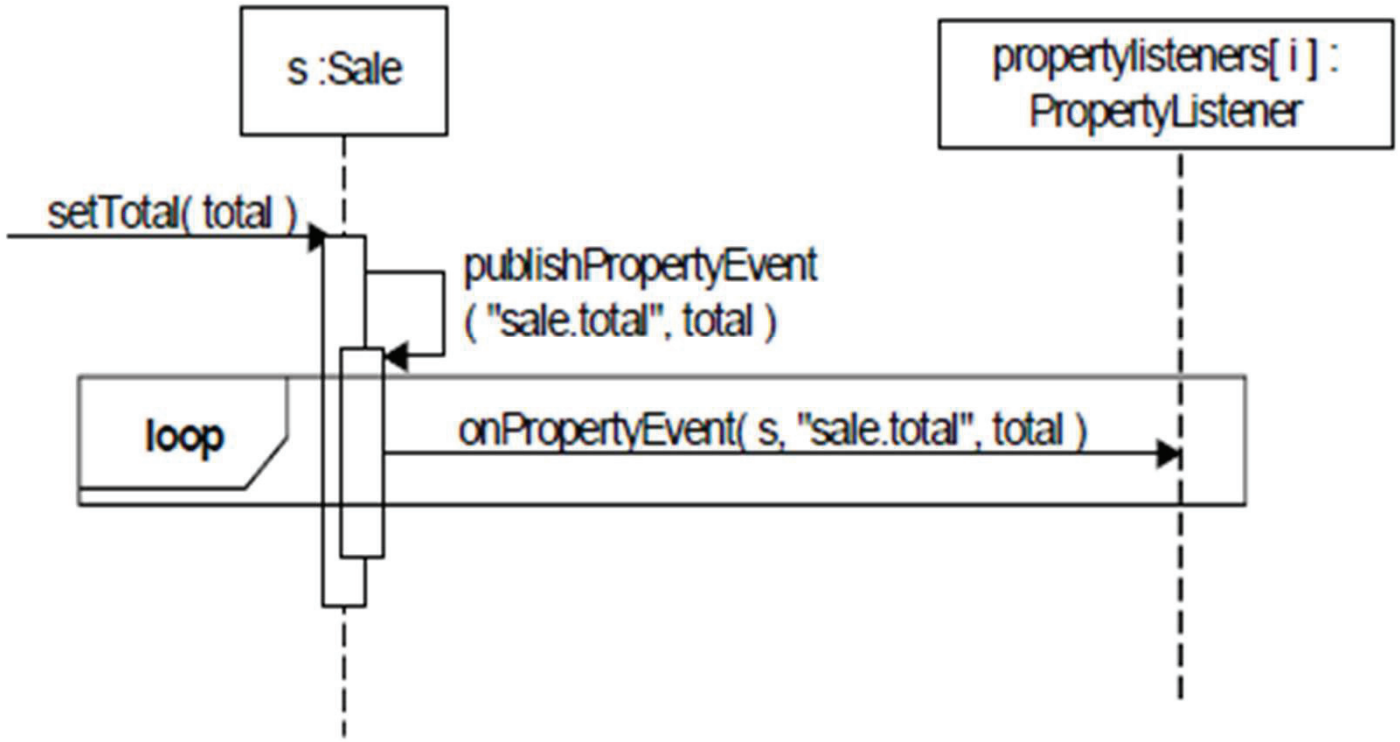


Observer subscribe to publisher

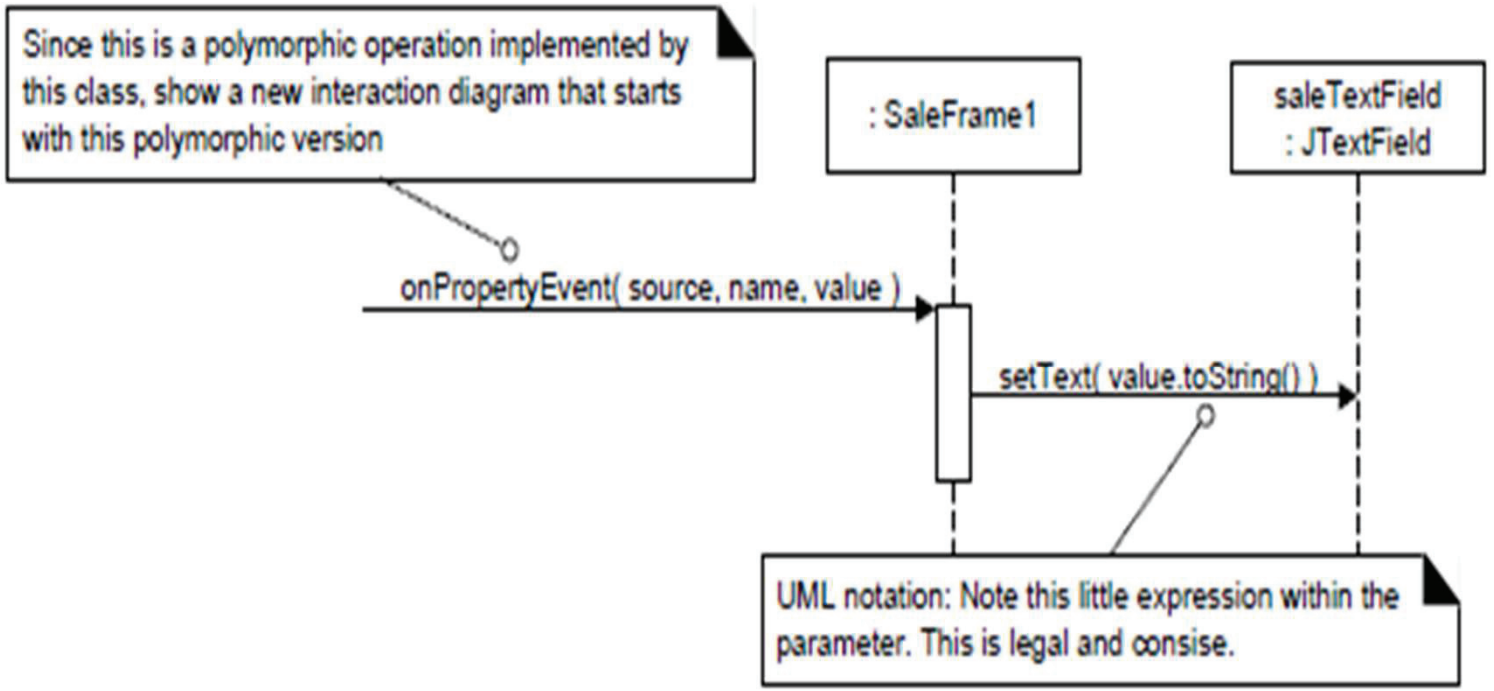
- When the observer(listener, subscriber) SaleFrame1 is interested in property events of the sale (publisher), it sends subscription request to the publisher.
- Sale adds the SaleFrame object to its subscription list



Sale publishes a property events to all subscribers



Subscriber receives a notification



Observer in GUI class library

- Observer pattern is the basic structure of event handling system in Java's GUI library
- Each GUI widget is a publisher of GUI related Events
- ActionEvent,MouseClickedEvent, ListSelectionEvent,…
- Observer can be any other objects (GUI ornon GUI objects)

Observer in Java GUI

- A simple GUI with a JTextField and a JList component
- Each time the List is selected, the selected text is displayed in the JTextField.
- JList supports ListSelectionListener
- A subclass of JTextField should implements ListSelectionListener.

Observer in Java GUI

- A simple GUI with a JTextField and a JList component
- Each time the List is selected, the selected text is displayed in the JTextField.
- JList supports ListSelectionListener
- A subclass of JTextField should implements ListSelectionListener.

Summary

- Design patterns helps us to build upon the experience laid by the software experts.
- Patterns are described as problem / solution pair in a given context.
- GRASP patterns by Larman, applies the fundamental design principles to OOD diagrams.
- GOF patterns describe the structural, behavioral and creational patterns.

Question Bank – 2 marks

1. Define Design patterns
2. Define responsibility. What are the various types of it.
3. List out the categories of design patterns
4. Define GRASP.
5. Interpret the need of information expert.
6. Distinguish between coupling and cohesion.
7. Point out the interface and domain layer responsibilities.
8. Discuss the advantages of adapter pattern
9. List out the structural patterns
10. Analyze the situation to use factory pattern and its advantages.

Question Bank – 16 marks

1. What is GRASP? Describe the design patterns and patterns used in it.
2. Examine the following GRASP patterns
    1. Creator
    2. Information expert
    3. Controller
    4. Low coupling
    5. High cohesion
3. Explain about Creator and controller patterns with example.
4. Explain in detail about the various behavioral GOF patterns.
5. Explain in detail about the various Structural GOF patterns.
6. Explain in detail about factory GOF pattern.