Object Oriented Design and Modeling through UML

Chapter-3

Object Oriented Design

Facilitator: Er. Rudra Nepal

Strength and Weakness of sequence and collaboration diagram

- Already discussed in chapter-1!!

Responsibilities

- Responsibilities are the roles or tasks to be performed or to be known by objects.
- It can be simply divided into two categories:

1. **Doing Responsibilities**
2. **Knowing Responsibilities**

1. Doing Responsibilities

- It includes:
  - ➢ **Doing something itself, such as creating an object or doing a calculation.**
  - ➢ **Initiating actions in other objects.**
  - ➢ **Controlling and coordinating the activities in other objects.**

2. Knowing Responsibilities

- It includes:
  - ➢ **Knowing about private encapsulated data.**
  - ➢ **Knowing about related objects.**
  - ➢ **Knowing about the things that it can derive and calculate.**

Responsibilities

- For example,
  - ➢**In POS system, a Sale class is responsible for creating its instance (Doing) and it is also responsible for knowing the quantity of items and total (Knowing).**

Operation and Method

- An operation is a task to be performed or it can be defined as a message that needs to be communicated among objects, but it does not show any implementation details. It is represented by showing the return type, function name and the parameters that it includes.
- Method is similar to operation but here we also show the implementation details.

Composition versus Aggregation

- Already discussed!!

Visibility

- Visibility is the ability of one class to see or have reference to another class.
- It deals with **the matter of scope** i.e. **does one object lie within the visibility of another?**
- Message passing between two objects depends upon the visibility from one object to another.

Visibility

- Suppose , we have two classes, Class A and Class B. The message passing from Class A to Class B can be made possible with four different kinds of visibilities:
  - ➢**Attribute visibility**
  - ➢**Parameter visibility**
  - ➢**Locally declared visibility**
  - ➢**Global visibility**

1. Attribu visibility

- Here, the object of Class B is declared as an attribute inside Class A.
- It is a permanent visibility as it exists as long as the classes exist.

1. Attribu visibility

- Example:

```
public class B
    {
        float r=3.5;
        public float calculateArea()
            {
                return(3.14*r*r);
            }
    }
public class A
    {
        B b=new B();
        public static void main(String[] args)
            {
                System.out.println(b.calculateArea());
            }
    }
```

1. Attribu visibility

- In the previous example, the message passing between Class A and Class B is possible as the Class A has an attribute visibility to Class B i.e. the object of Class B has been declared as an attribute inside class A.

2. Paramet visibility

- Here the object of class B is passed as a parameter inside the method of class A.
- It is a temporary visibility as its scope is only within the method in which it is passed as a parameter .

2. Paramet visibility

- Example:

```
public class A
    {
        public void area(B b)
            {
                System.out.println(b.calculateArea());
            }
    }
```

2. Paramet visibility

- In above example, the message passing between Class A and Class B is possible as the parameter visibility exists from Class A to Class B i.e. an object of Class B has been passed as a parameter inside the method of Class A.

3. Locally declare visibility

- Here, the object of Class B is declared as a local variable inside the method of Class A.
- It is also a temporary visibility as its scope lies only within the method in which it is declared as a local variable.

3. Locally declare visibility

- Example:

```
public class A
    {
        public void area()
            {
                B b =new B();
                System.out.println(b.calculateArea());
            }
    }
```

3. Locally declare visibility

- In above example, the message passing from Class A to Class B is possible as there exists a locally declared visibility from Class A to Class B i.e. an object of Class B has been declared as a local variable inside the method of class A.

4. Global visibility

- Here, the object of Class B is declared as a global variable.
- It is a permanent visibility as its scope exists as long as the program exists.

4. Global visibility

- Example:

```
public class B
      {
            …
      }
B b =new B();//not possible
public class A
      {
            public void area()
                  {
                        System.out.println(b.calculateArea());
                  }
      }
```

4. Global visibility

- In above example, message passing from Class A to Class B is possible as there exists a global visibility from Class A to Class B i.e. an object of Class B has been declared as a global variable.
- This kind of visibility is rarely used as it is not supported by every programming language. For example: C, C++, C# supports global visibility but not Java.

Patterns (General Responsibility Assignment Software Patterns/GRASP)

- It is a set of guidance provided for assigning a certain kind of responsibility to the classes.
- In other words, it is a repository of both general principles and solutions that guides in the phase of software development.
- The choices for assigning the responsibilities are shown in domain model an interaction diagrams.

Patterns (General Responsibility Assignment Software Patterns/GRASP)

- There are generally five types of software patterns:
  - ➢**Information expert pattern or expert pattern**
  - ➢**Creator pattern**
  - ➢**High cohesion pattern**
  - ➢**Low coupling pattern**
  - ➢**Controller pattern**

Information expert pattern or expert pattern

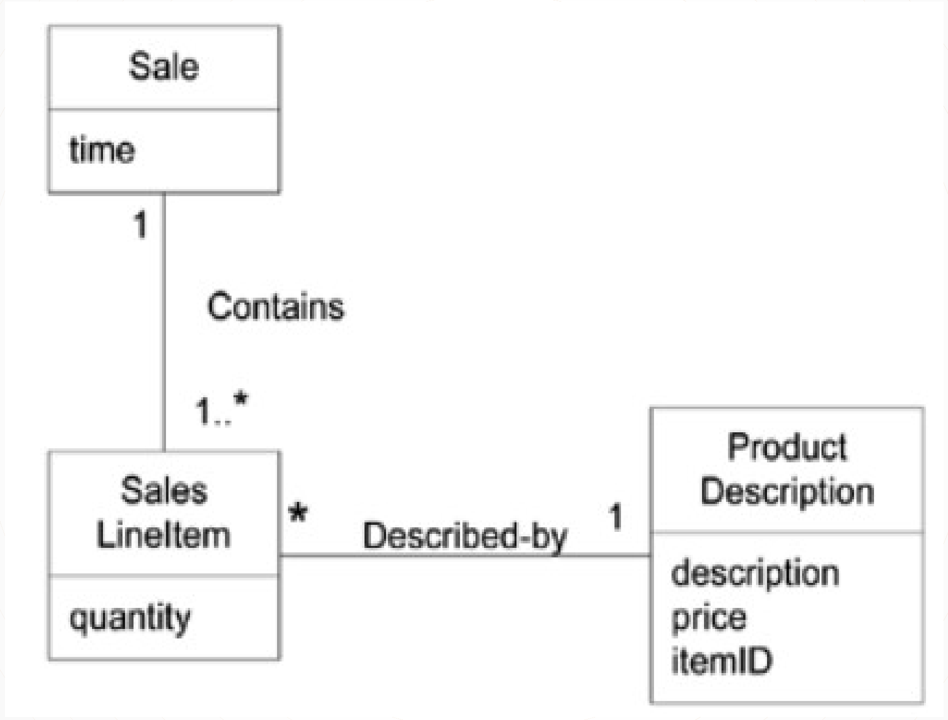
- This pattern guides us to assign the responsibility to the class which has full information required to fulfill that responsibility.
- **Problem : Which class should be responsible for doing certain things?**
- **Solution : Assign the responsibility to the class which has the total information necessary for doing that thing. That particular class is also known as expert class.**

Information expert pattern or expert pattern

- For example,
- **Problem : in POS application, which class should be responsible for calculating the total amount of items purchased by the customer?**
- **Solution:??**

Solution:
-From the domain model of POS application, it is seen that the ProductDescription class contains the information about the price of an individual item
-Similarly, SalesLineItem has information about the quanti
a particular item.
-It is seen that Sale class cont
the SalesLineItem class whic
turn is described by the ProductDescription class.
-Therefore, by information pattern, Sale class would be responsible for calculating the total amount of items purchas
by the customer as it has all the information required to fulfill th
responsibility.

- **Domain Model:**



Creator pattern

- This pattern is used to assign a responsibility of creating the instance of a particular class to another class.
- **Problem : Which class should be responsible for creating a new instance of another class ?**
- **Solution : Assign a responsibility of creating the instance of Class A to Class B if one or more of the following is true:**
  - ➢ **Class B aggregates the Class A objects.**
  - ➢ **Class B contains Class A objects.**
  - ➢ **Class B records the instances of Class A objects.**
  - ➢ **Class B has the initializing data that will be passed to Class A when it is created.**

Creator pattern

- For example,
- **Problem : In POS application, Who should be responsible for creating the SaleLineItem instance ?**
- **Solution: Since Sale class contains the objects of SaleLineItem class, the responsibility of creating the instance of SaleLineItem class is assigned to the Sale class by creator pattern.**

High cohesion pattern

- This pattern guides us in keeping the objects focused, understandable and manageable.
- Here, we assign a responsibility so that the cohesion remains high.
- Cohesion is the measure of how strongly related and focused the responsibilities of the elements are.
- An element with highly related responsibilities and which does not do a tremendous amount of work has high cohesion.

High cohesion pattern

- All class with low cohesion performs many unrelated activities and they suffer from following problems :
  - ➢ **hard to comprehend or understand**
  - ➢ **hard to reuse**
  - ➢ **hard to maintain**
  - ➢ **they are delicate i.e. constantly affected by the change**

High cohesion pattern

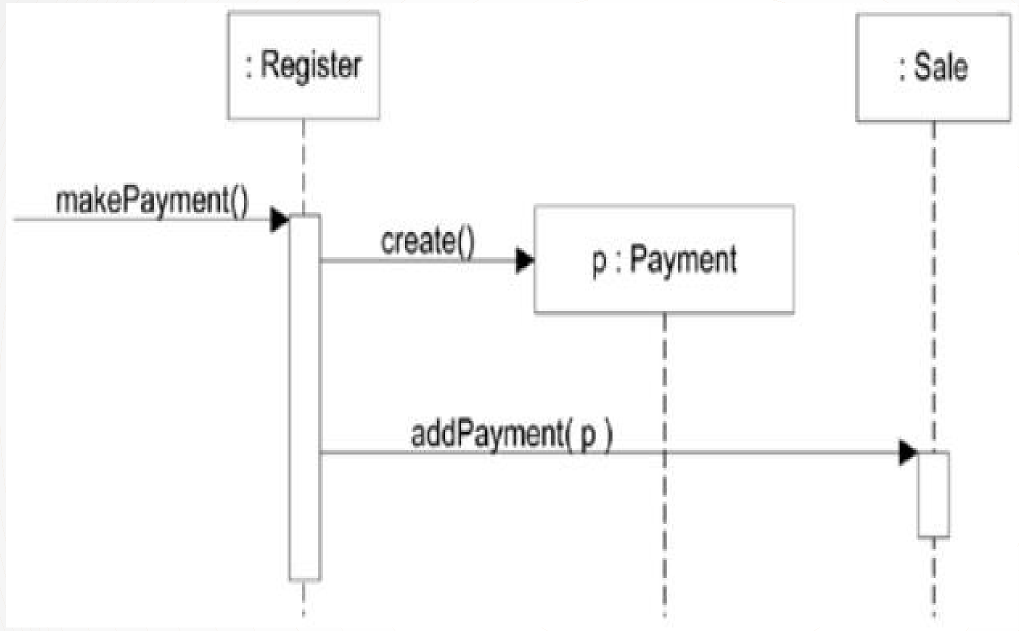
- For example: In POS application, assume that we need to create the instance of Payment class and associate it with sale, which class should be assigned this responsibility?
- Solution:??

Solution:

- Here, by creator pattern, we see that the Register class should be responsible for creating the instance of Payment class and associating the information of Payment to the same class.
- But, let's have a look at the sequence diagram in this situation !!

Solution:

- **Sequence Diagram(Figure 1)**

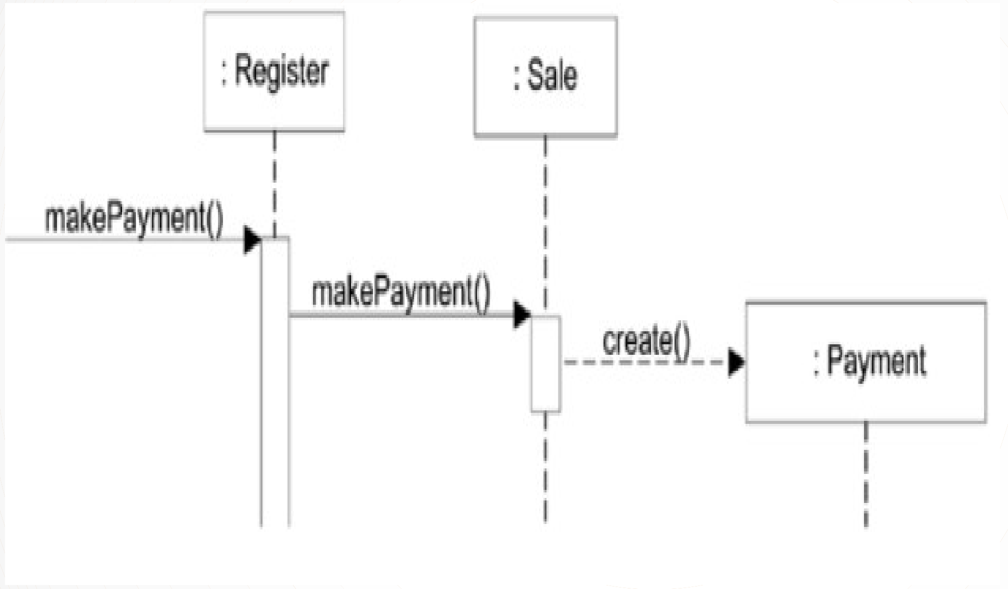


Solution:

- Here, in Figure 1 , we see that the Register class is solely responsible for creating the instance of Payment and associating it with the Sale. Therefore, Register class has taken some of the unrelated responsibilities which may violate the concept of high cohesion pattern.
- So by high cohesion pattern, Register class can delegate its responsibility of creating the Payment instance to the Sale class and the desired sequence diagram can be shown in Figure 2.

Solution:

- **Sequence Diagram(Figure 2)**



Low coupling pattern

- This pattern guides us in supporting the low dependency, low change impact and increased reuse.
- Here, we assign the responsibility to the class so that the coupling remains low.
- Coupling is the measure of how strongly one element is connected to, has the knowledge of or relies on other elements.
- An element with low coupling is not dependent on too many elements.

Low coupling pattern

- A class with high coupling suffers from the following problems :
  - ➢**hard to comprehend or understand**
  - ➢**hard to reuse**
  - ➢**hard to maintain**
  - ➢**they are delicate, that is constantly affected by change**

Low coupling pattern

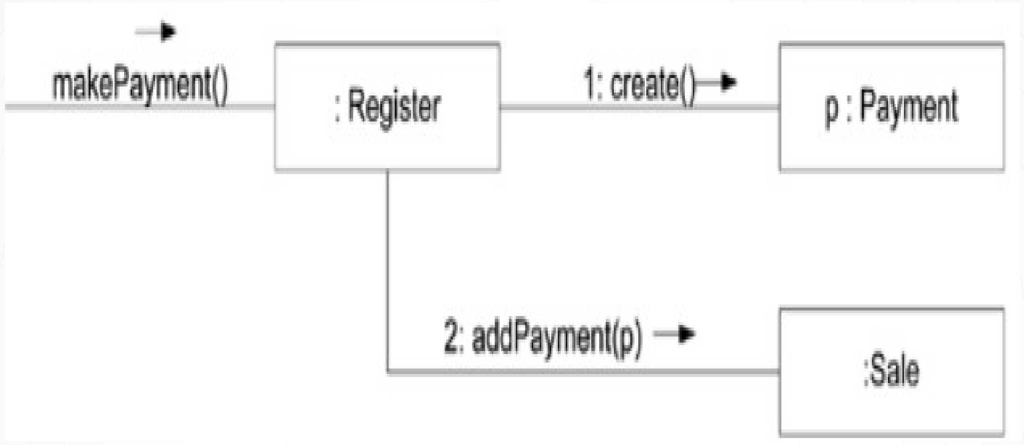
- For example: In POS application, assume that we need to create the instance of Payment class and associate it with sale, which class should be assigned this responsibility?
- Solution:??

Solution:

- Here, by creator pattern, we see that the Register class should be responsible for creating the instance of Payment class and associating the information of Payment to the same class.
- But, let's have a look at the collaboration diagram in this situation !!

Solution:

- **Collaboration Diagram(Figure 1)**

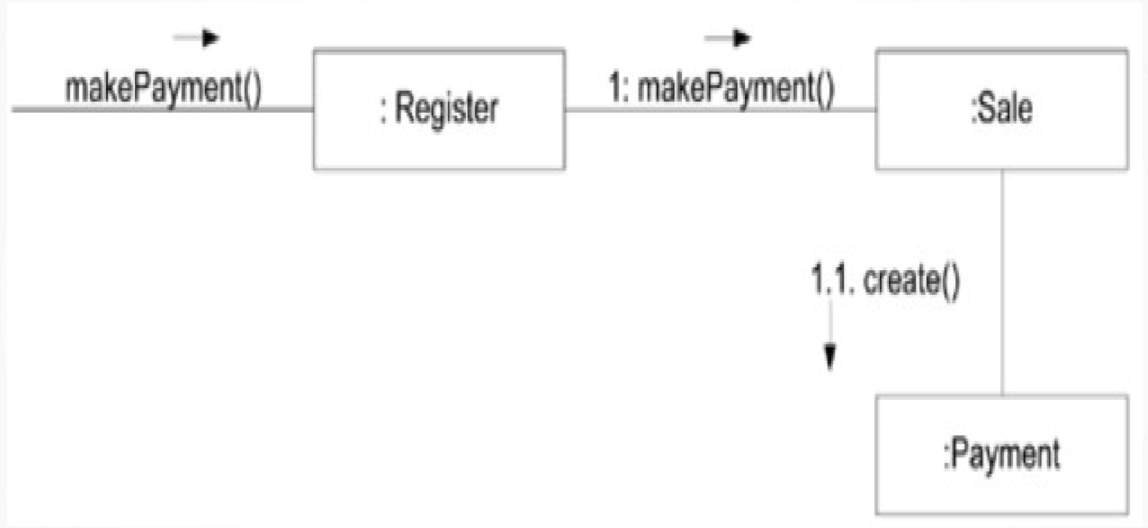


Solution:

- Here, in Figure 1 , we see that the Register class is solely responsible for creating the instance of Payment and associating it with the Sale. Therefore, it increases the coupling for the Register class.
- So by low coupling pattern, Register class can delegate its responsibility of creating the Payment instance to the Sale class and the desired collaboration diagram can be shown in Figure 2.

Solution:

- **Collaboration Diagram(Figure 2)**



Controller pattern

- This pattern guides on identifying the first object behind the user interface layer that receives and coordinates a particular system operation.
- **A controller is a non-user interface object responsible for receiving and handling the particular system event.**

Controller pattern

- **Problem : Who should be responsible for handling the input system event?**
- **Solution : Assign a responsibility to the class which represents one of the following choices :**
  - ➢ **represents the overall system (Facade controller)**
  - ➢ **represents a use case scenario within which the system event occurs, often named <UseCaseName> Handler, <UseCaseName> Coordinator, or <Use-CaseName> Session (use-case or session controller).**

Controller pattern

- **Problem : Who should be responsible for handling the input system event?**
- **Solution :**

An input **system event** is an event generated by an external actor. They are associated with **system operations**. Operations of the system is response to system events, just as messages and methods are related.

For example, when a cashier using a POS terminal presses the "End Sale" button, he is generating a system event indicating "the sale has ended“. Similarly, when a writer using a word processor presses the "spell check" button, he is generating a system event indicating "perform a spell check."

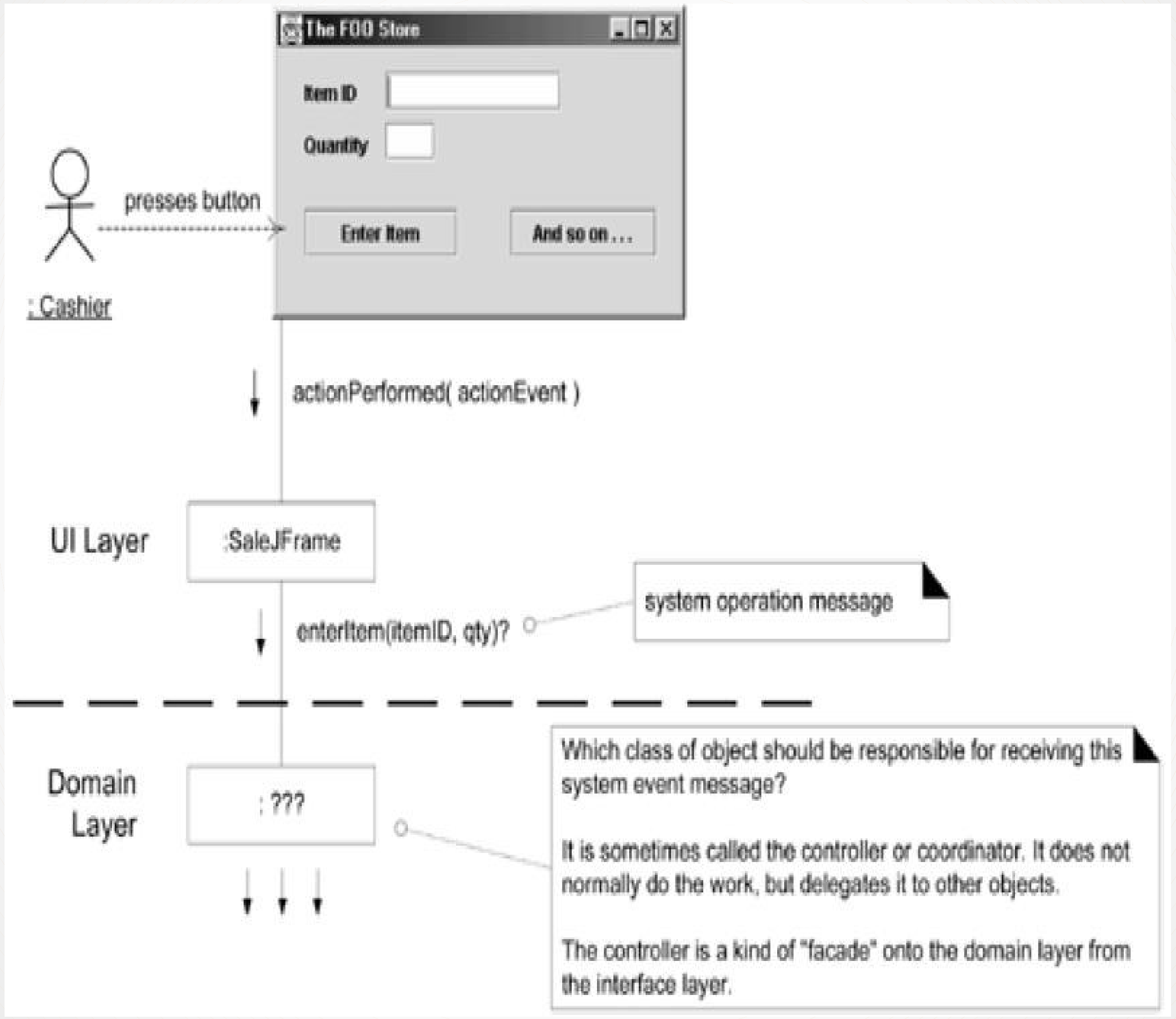
Choosing Controller Class

- Our first choice involves choosing the controller for the system operation message

enterItem

- Choices: -
- represents the overall "system," device, or Register, POSTSystem subsystem represents a receiver or handler of all system ProcessSaleHandler, events of a use case scenario ProcessSaleSesssion

Choosing a Contr
Class

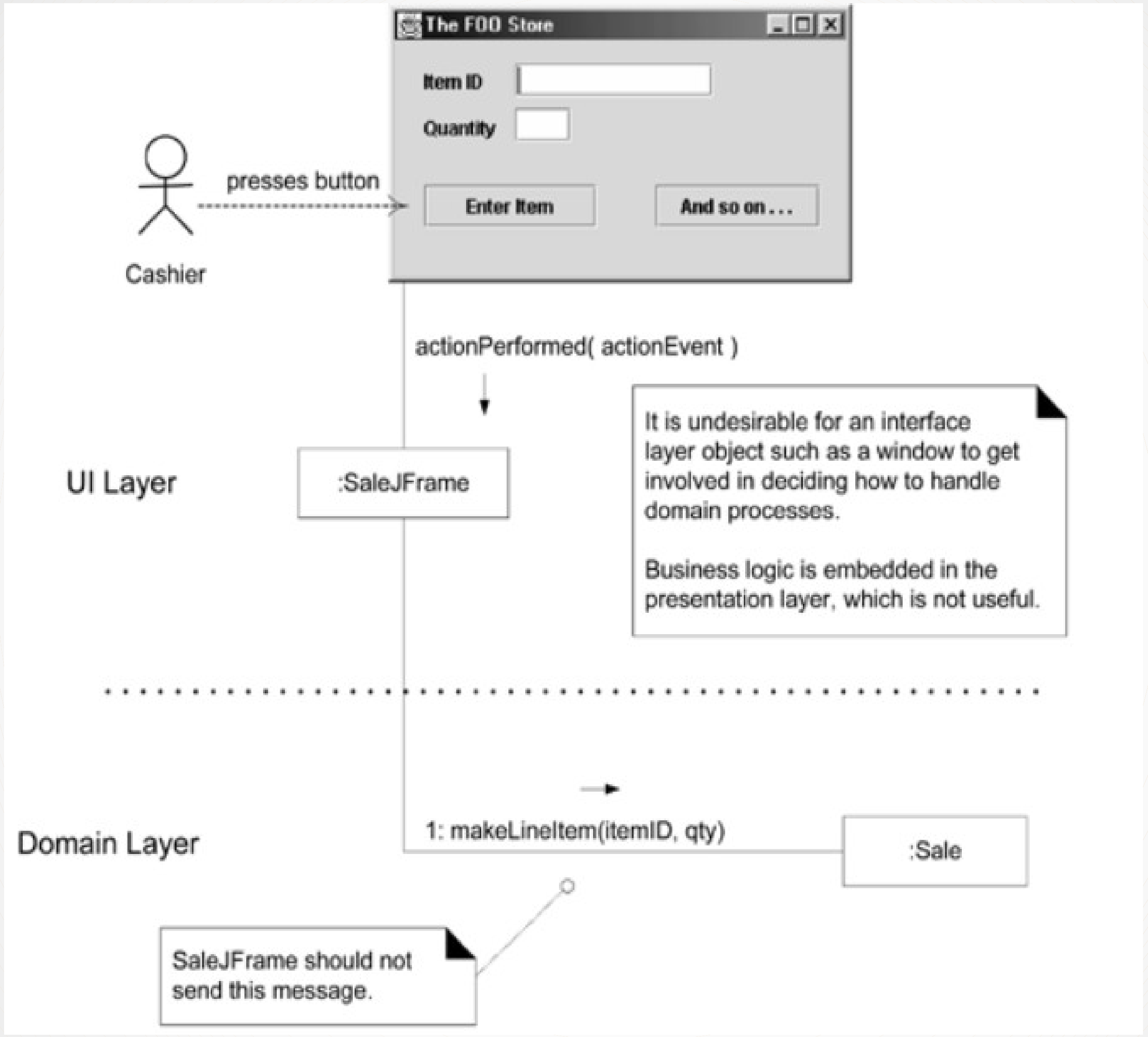


Which class of object should be responsible for receiving this system event message?

It is sometimes called the controller or coordinator. It does not normally do the work, but delegates it to other objects.

The controller is a kind of "facade" onto the domain layer from the interface layer.

Choosing a Contr Class

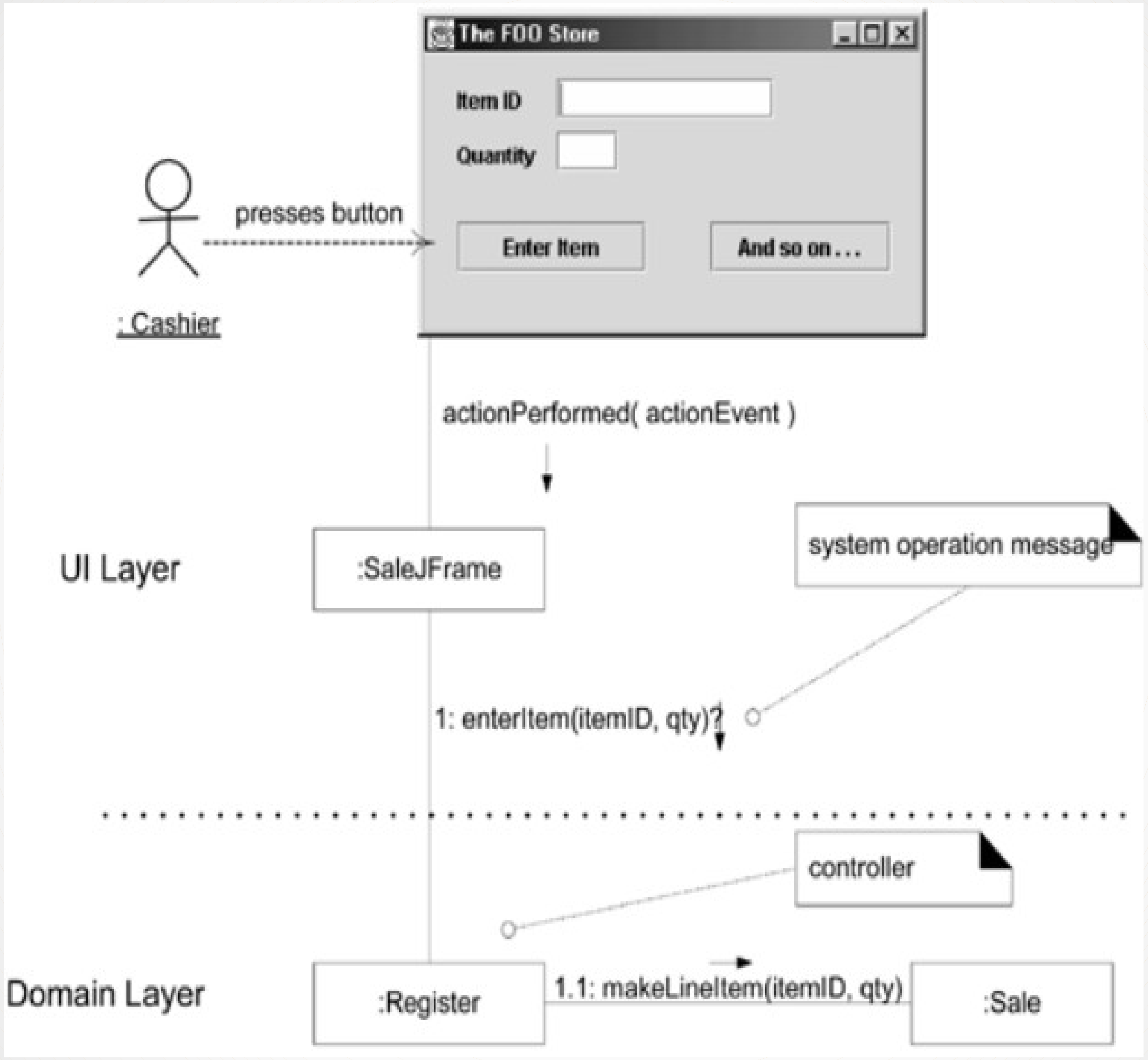


It is undesirable for an interface layer object such as a window to get involved in deciding how to handle domain processes.

Business logic is embedded in the presentation layer, which is not useful.

SaleJFrame should not send this message.

Choosing a Contr Class



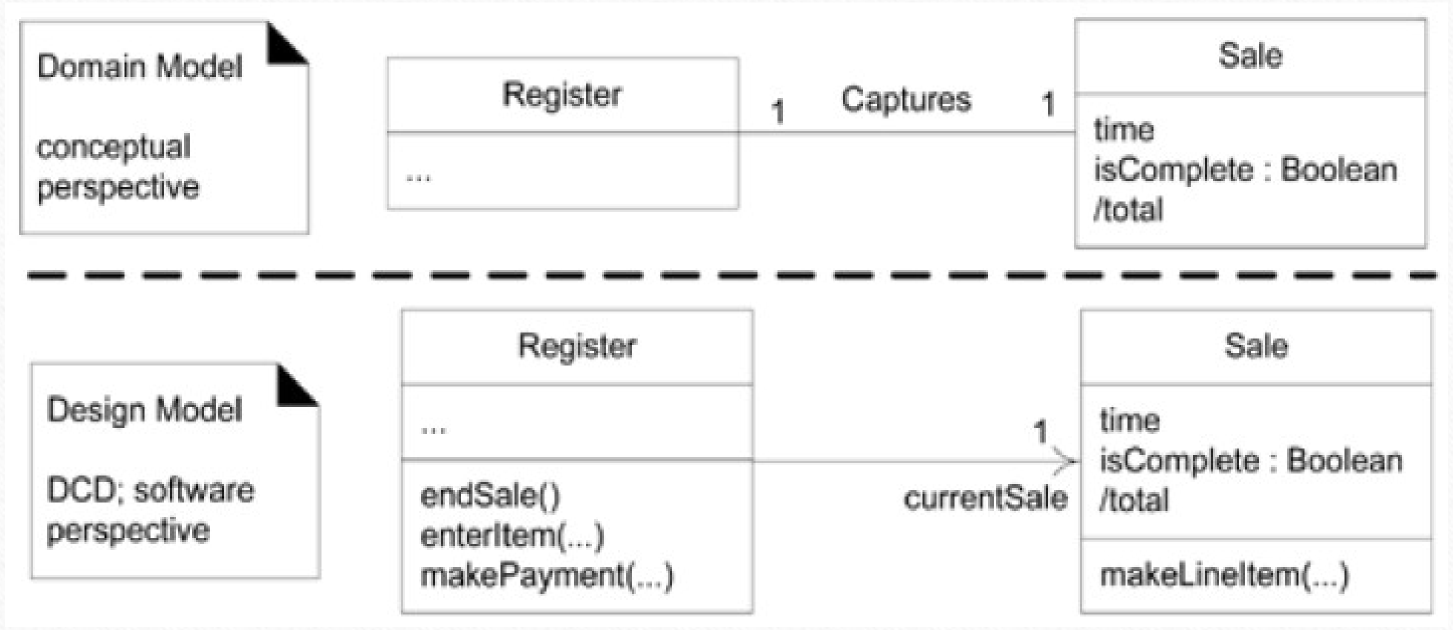
Differences between System Sequence Diagram and Sequence Diagram

System sequence diagram	sequence diagram
1. The elements participating in this diagram are the actors and system.	1. The elements participating in this diagram are the objects.
2. The messages exchanged by these elements can be of any type depending upon the system.	2. The messages exchanged by these elements must be method invocations.
3. It visualizes the use case.	3. It visualizes the methods of a class.
4. Input and output can be anything irrespective of the functionalities of the system.	4. Input and output should show the functionalities of the system.
5. Database is not represented.	5. Database representation can also be shown.

Difference between Domain Model and Design Class Diagram

- In domain model, a class does not represent the system operations, rather it is the abstraction of the real-world concept. Therefore, class name represents the conceptual class.
- In design class diagram, a class represents the system operations. Therefore, it shows the implementation details as well. So, the class name represents the software class.

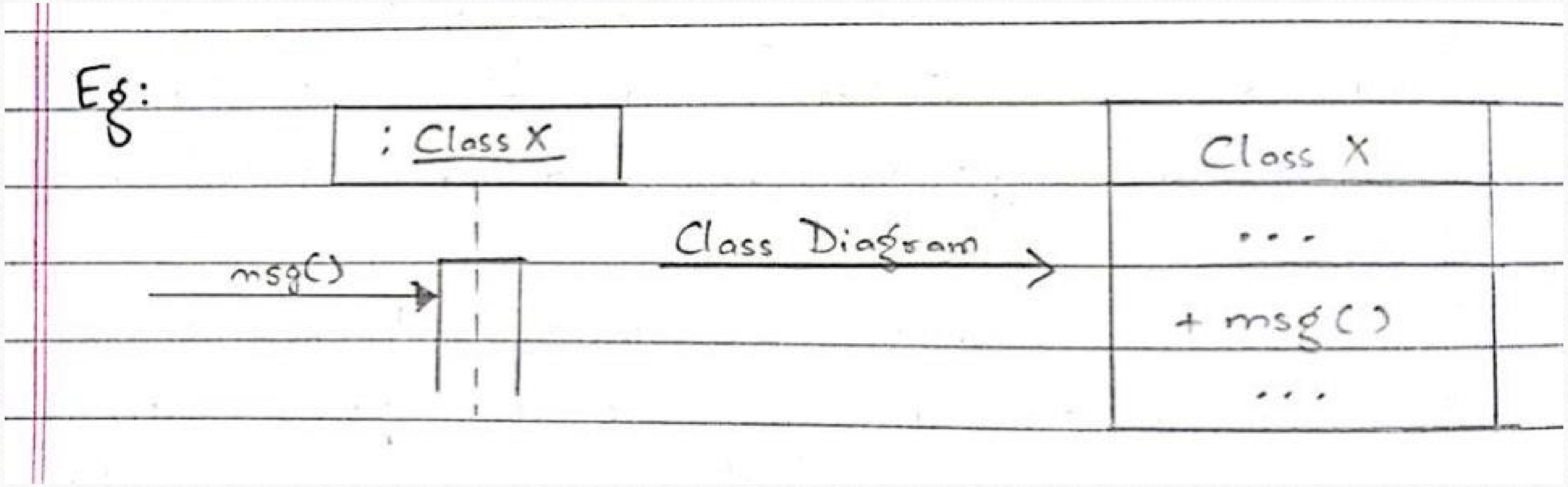
Difference between Domain Model and Design Class Diagram



Identifying methods in Design Class Diagram from Interaction Diagram

- In a class diagram, the methods are identified with the help of interaction diagram.
- All the incoming messages in a particular class in interaction diagrams should be defined in that class in class diagram.
- If a method name called msg() is incoming in class X in interaction diagram, then that message must be represented inside class X in class diagram.

Identifying methods in Design Class Diagram



Issues regarding method names(5 marks)

- There are four issues regarding representation of method names in Class Diagram:
  - ➢ **Method Names-Create**
  - ➢ **Method Names-Accessing Method**
  - ➢ **Method Names-Multi objects**
  - ➢ **Method Names-Language Dependent Syntax**

Method Names-Create

- The message ‘create’ in interaction diagram is used to indicate the instantiation and initialization of an object.
- Therefore, there is no need to show the create message in class diagram.

Method Names-Accessing Methods

- Accessing methods are those methods which is used to retrieve and set attributes.
- **For example, getPrice() method in ProductSpecification class is just used to get the value of price in a program. Therefore, it is not necessary to show the getPrice() method in ProductSpecification class in class diagram.**

Method Names-Multi objects

- A message to the multi objects is interpreted as a message to the container or the collection of objects itself.
- The container classes or interface such as **java.util.map** are usually predefined library and it is not useful to show such classes methods explicitly in class diagram as it does not have any new information.

Method Names-Language Dependent Syntax

- In certain languages, such as SmallTalk, the syntax for method name representation is quite different than that of basic UML format.
- It is recommended that the basic UML format should be used even if the planned implementation programming language uses a different syntax as the translation to particular programming language takes place only during the time of code generation.