Module 8

Object-Oriented Software Development

Lesson 18

Design Patterns

Specific Instructional Objectives

At the end of this lesson the student will be able to:

- Identify the basic difference between object-oriented analysis (OOA) and object-oriented design (OOD).
- Explain why design patterns are important in creating good software design.
- Explain what are design patterns.
- Identify pattern solution for a particular problem in terms of class and interaction diagrams.
- Explain expert pattern and circumstances when it can be used.
- Explain creator pattern and circumstances when it can be used.
- Explain controller pattern and circumstances when it can be used.
- Explain facade pattern and circumstances when it can be used.
- Explain model view separation pattern and circumstances when it can be used.
- Explain intermediary pattern (i.e. proxy pattern) and circumstances when it can be used.

Identify the basic difference between object-oriented analysis (OOA) and object-oriented design (OOD).

The term object-oriented analysis (OOA) refers to a method of developing an initial model of the software from the requirements specification. The analysis model is refined into a design model. The design model can be implemented using a programming language. The term object-oriented programming refers to the implementation of programs using object-oriented concepts.

In contrast, object-oriented design (OOD) paradigm suggests that the natural objects (i.e. the entities) occurring in a problem should be identified first and then implemented. Object-oriented design (OOD) techniques not only identify objects but also identify the internal details of these identified objects. Also, the relationships existing among different objects are identified and represented in such a way that the objects can be easily implemented using a programming language.

Explain why design patterns are important in creating good software design.

Design patterns are reusable solutions to problems that recur in many applications. A pattern serves as a guide for creating a “good” design. Patterns are based on sound common sense and the application of fundamental design principles. These are created by people who spot repeating themes across designs. The pattern solutions are typically described in terms of class and interaction diagrams. Examples of design patterns are expert pattern, creator pattern, controller pattern etc.

Explain what design patterns are.

Design patterns are very useful in creating good software design solutions. In addition to providing the model of a good solution, design patterns include a clear specification of the problem, and also explain the circumstances in which the solution would and would not work. Thus, a design pattern has four important parts:

- The problem.
- The context in which the problem occurs.
- The solution.
- The context within which the solution works.

Identify pattern solution for a particular problem in terms of class and interaction diagrams.

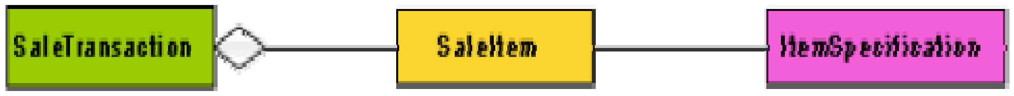
The design pattern solutions are typically described in terms of class and interaction diagrams.

Example:

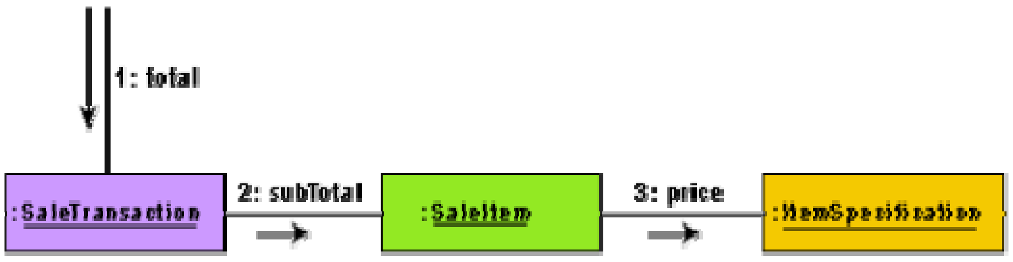
Expert Pattern

Problem: Which class should be responsible for doing certain things?

Solution: Assign responsibility to the information expert – the class that has the information necessary to fulfill the required responsibility. The expert pattern expresses the common intuition that objects do things related to the information they have. The class diagram and collaboration diagrams for this solution to the problem of which class should compute the total sales is shown in the fig. 8.1.



(a)



(b)

Fig. 8.1: Expert pattern: (a) Class diagram (b) Collaboration diagram

Explain expert pattern and circumstances when it can be used.

Expert pattern was defined earlier**.**

Explain creator pattern and circumstances when it can be used.

Creator Pattern

Problem: Which class should be responsible for creating a new instance of some class?

Solution: Assign a class C1 the responsibility to create an instance of class C2, if one or more of the following are true**:**

- C1 is an aggregation of objects of type C2.
- C1 contains objects of type C2.
- C1 closely uses objects of type C2.
- C1 has the data that would be required to initialize the objects of type C2, when they are created.

Explain controller pattern and circumstances when it can be used.

Controller Pattern:

Problem: Who should be responsible for handling the actor requests?

Solution: For every use case, there should be a separate controller object which would be responsible for handling requests from the actor. Also, the same controller should be used for all the actor requests pertaining to one use case so that it becomes possible to maintain the necessary information about the state of the use case. The state information maintained by a controller can be used to identify the out-of-sequence actor requests, e.g. whether voucher request is received before arrange payment request.

Explain facade pattern and circumstances when it can be used.

Façade Pattern:

Problem: How should the services be requested from a service package?

Context in which the problem occurs: A package as already discussed is a cohesive set of classes – the classes have strongly related responsibilities. For example, an RDBMS interface package may contain classes that allow one to perform various operations on the RDBMS.

Solution: A class (such as DBfacade) can be created which provides a common interface to the services of the package.

Explain model view separation pattern and circumstances when it can be used.

Model View Separation Pattern:

Problem: How should the non-GUI classes communicate with the GUI classes?

Context in which the problem occurs: This is a very commonly occurring pattern which occurs in almost every problem. Here, model is a synonym for the domain layer objects, view is a synonym for the presentation layer objects such as the GUI objects.

Solution: The model view separation pattern states that model objects should not have direct knowledge (or be directly coupled) to the view objects. This means that there should not be any direct calls from other objects to the GUI objects. This results in a good solution, because the GUI classes are related to a particular application whereas the other classes may be reused.

There are actually two solutions to this problem which work in different circumstances as follows:

Solution 1: Polling or Pull from above

It is the responsibility of a GUI object to ask for the relevant information from the other objects, i.e. the GUI objects pull the necessary information from the other objects whenever required.

This model is frequently used. However, it is inefficient for certain applications. For example, simulation applications which require visualization, the GUI objects would not know when the necessary information becomes available. Other examples are, monitoring applications such as network monitoring, stock market quotes, and so on. In these situations, a "push-from-below" model of display update is required. Since "push-from-below" is not an acceptable solution, an indirect mode of communication from the other objects to the GUI objects is required.

Solution 2: Publish- subscribe pattern

An event notification system is implemented through which the publisher can indirectly notify the subscribers as soon as the necessary information becomes available. An event manager class can be defined which keeps track of the subscribers and the types of events they are interested in. An event is published by the publisher by sending a message to the event manager object. The event manager notifies all registered subscribers usually via a parameterized message (called a callback). Some languages specifically support event manager classes. For example, Java provides the EventListener interface for such purposes.

Explain intermediary pattern (i.e. proxy pattern) and circumstances when it can be used.

Intermediary Pattern or Proxy

Problem: How should the client and server objects interact with each other?

Context in the problem occurs: The client and server terms as used here refer to software components existing across a network. The clients are consumers of services provided by the servers.

Solution: A proxy object at the client side can be defined which is a local sit-in for the remote server object. The proxy hides the details of the network transmission. The proxy is responsible for determining the server address, communicating the client request to the server, obtaining the server response and seamlessly passing that to the client. The proxy can also augment (or filter) information that is exchanged between the client and the server. The proxy could have the same interface as the remote server object so that the client feels as if it is interacting directly with the remote server object and the complexities of network transmissions are abstracted out.