

Software Architecture and Design Patterns

Time: 3 hrs.

Max. Marks: 80

Solution:

Module 1

1a. what is a design pattern? Explain essential elements of design pattern.

A pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

Essential Elements:

The **pattern name** is a handle we can use to describe a design problem, its solutions, and consequences in a word or two.

The **problem** describes when to apply the pattern.

The **solution** describes the elements that make up the design, their relationships, responsibilities, and collaborations. The pattern provides an abstract description of a design problem and how a general arrangement of classes and objects solves it.

The **consequences** are the results and trade-offs of applying the pattern.

Example Pattern:

Pattern Name – Iterator

Problem – How to serve Patients at a Doctor's Clinic

Solution – Front-desk manages the order for patients to be called

- By Appointment
- By Order of Arrival
- By Extending Gratitude
- By Exception

Consequences

- Patient Satisfaction
- Clinic's Efficiency
- Doctor's Productivity

1b. Explain object oriented development. Explain key concepts of object oriented design.

First computers

First computers are developed mainly to automate a well-defined process (i.e., an algorithm) for numerical computation, as systems became more complex; its effectiveness in developing solutions became suspect.

- Software applications developed in later years had two differentiating characteristics:
 - Behavior that was hard to characterize as a process
 - Requirements of reliability, performance, and cost that the original developers did not face
- The 'process-centred' approach to software development used what is called top down functional decomposition.
 - The first step in such a design was to recognize what the process had to deliver which was followed by decomposition of the process into functional modules.
 - Structures to store data were defined and the computation was carried out by invoking the modules, which performed some computation on the stored data elements.
 - The life of a process-centred design was short because changes to the process specification required a change in the entire program.
 - This resulted in an inability to reuse existing code without considerable overhead
- Thus engineering disciplines started soon after, and the disciplines of 'software design' and 'software engineering' came into existence. The reasons for this success are easy to see:
 - Easily understandable designs
 - Similar (standard) solutions for a host of problems
 - An easily accessible and well-defined 'library' of 'building-blocks'
 - Interchangeability of components across systems,

- A software component is also capable of storing data,
- The components can also communicate with each other as needed to complete the process.

Key Concepts of Object-Oriented Design

1. The Central Role of Objects
2. The notion of a Class
3. Abstract specification of functionality
4. A language to define the System
5. Standard Solutions
6. An analysis process to model a system
7. The notions of extendibility and adaptability

2a. Explain how to choose right design pattern for your problem.

To Select a Design Pattern

- Consider how design patterns solve design Problems
- Scan Intent sections
- Study how patterns interrelate
- Study patterns of like purpose
- Examine a Cause of redesign
- Consider what should be variable in the design

Consider how design patterns solve design problems

Determine object granularity; specify object interfaces, and several other ways in which design patterns solve design problems.

Scan Intent sections

Read through each pattern's intent (purpose) to find one or more that should be relevant to your problem.

Study how patterns interrelate

Studying these relationships can help direct you to the right pattern or group of patterns.

Study patterns of like purpose

Study only those patterns which are of specific purposes (creational patterns, structural patterns, and behavioural patterns).

Examine a cause of redesign

Look at the patterns that help you avoid the causes of redesign

Consider what should be variable in your design

Consider what you want to be able to change without redesign.

2b. Analyse step by step approach of applying a design pattern effectively.

1. Read the pattern once through for an overview.
 2. Go back and study the Structure, Participants and Collaborations sections.
 3. Look at the Sample Code section to see a concrete
 4. Example of the pattern in code.
 5. Choose names for pattern participants that are meaningful in the application context.
 6. Define the classes.
 7. Define Application-specific names for operations in the Pattern
 8. Implement the operations to carry out responsibilities and collaborations in the pattern.
-
1. Read the pattern once through for an overview. Pay attention to the Applicability and Consequences sections to ensure the pattern is right for your problem.
 2. Go back and study the Structure, Participants, and Collaborations sections. Make sure you understand the classes and objects in the pattern and how they relate to one another.
 3. Look at the Sample Code section to see a concrete example of the pattern in code. Helps you learn how to implement the pattern.
 4. Choose names for pattern participants that are meaningful in the application context. It is useful to incorporate the participant name into the name that appears in the application.
 5. Define the classes, declare their interfaces, establish their inheritance relationships, and define the instance variables that represent data and object references.

6. Define application-specific names for operations in the pattern. Use the responsibilities and collaborations associated with each operation as a guide. Also, be consistent in your naming conventions

7. Implement the operations to carry out the responsibilities and collaborations in the pattern. The implementation section offers hints to guide you in the implementation.

Module 2

3a. what is use case analysis? Draw neat use case diagram for library system, also explain Register user use case with proper description.

Use Case Analysis:

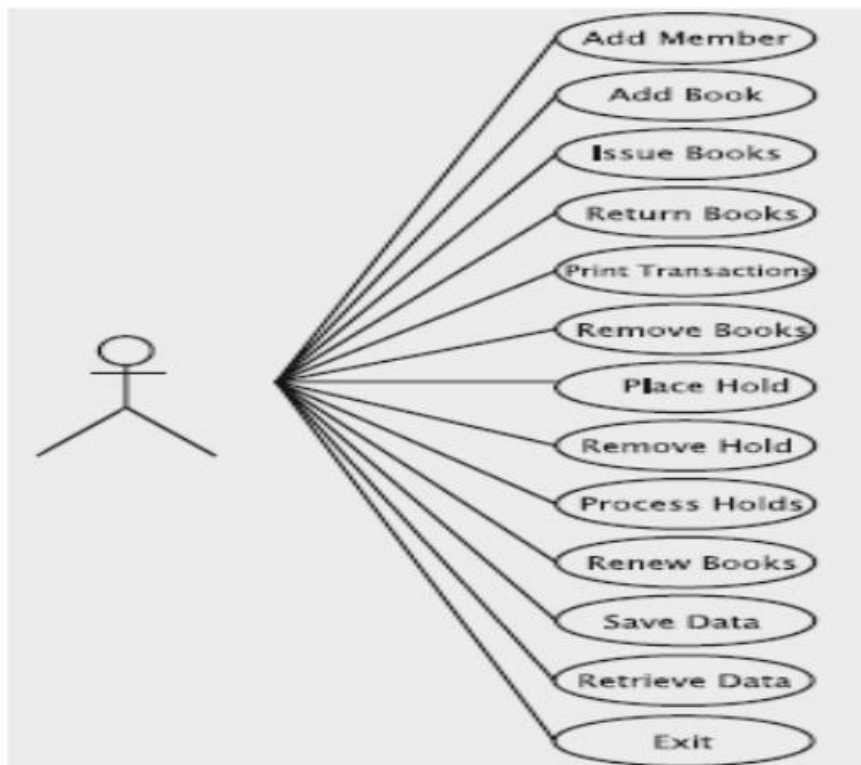
- It is a powerful technique that describes the kind of functionality that a user expects from the system.
- It is essentially a narrative describing the sequence of events (actions) of an external agent (actor) using the system to complete a process.
- It is a powerful technique that describes the kind of functionality that a user expects from the system

In our simple library system, the members do not use the system directly. Instead, they get services via the library staff.

To initiate this process, we need to get a feel for how the system will interact with the end-user. We assume that some kind of a user-interface is required, so that when the system is started, it provides a menu with the following choices:

1. Add a member
2. Add books
3. Issue books
4. Return books
5. Remove books
6. Place a hold on a book
7. Remove a hold on a book
8. Process Holds: Find the first member who has a hold on a book

9. Renew books
10. Print out a member's transactions
11. Store data on disk
12. Retrieve data from disk
13. Exit



Use case for registering a user

Actions performed by the actor	Responses from the system
1. The customer fills out an application form containing the customer's name, address, and phone number and gives this to the clerk	
2. The clerk issues a request to add a new member	
	3. The system asks for data about the new member
4. The clerk enters the data into the system	
	5. Reads in data, and if the member can be added, generates an identification number (which is not necessarily a number in the literal sense just as social security numbers and phone numbers are not actually numbers) for the member and remembers information about the member. Informs the clerk if the member was added and outputs the member's name, address, phone and id
6. The clerk gives the user his identification number	

Steps:

- 1 Member will give the details of name, address, phone number to the clerk
- 2 Clerk process the request through the system
- 3 System asks the details of the customer to be registered.
- 4 Clerk enters the necessary information of the member into the system.
- 5 System check the details of the member and if the member is a valid person, then generates member identification number and display the necessary information at the output
- 6 Clerk provides the identification number to the user.

3b. Define facade pattern. Explain with neat diagram.

The Facade Pattern:

Library class that provided a set of methods for the interface and thus served as a single point of entry to and exit from the business logic module. In the language of design patterns, what we created is known as a facade.

The primary motivation behind using a façade is to reduce the complexity by minimizing communication and dependencies between a subsystem and its clients . The facade not only shields the client from the complexity but also enables loose coupling between the subsystem

and its clients. Facades are not typically designed to prevent the client from accessing the components within the subsystem.

Using a Facade Where do we employ this? A situation in which we have:

1. A system with several individual classes, each with its own set of public methods.
2. An external entity interacting with the system requires knowledge of the public methods of several classes.

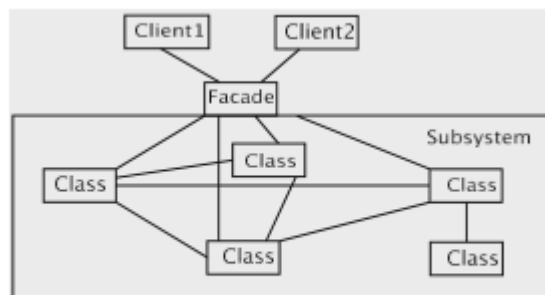


Figure 2.30 Structure diagram for facade

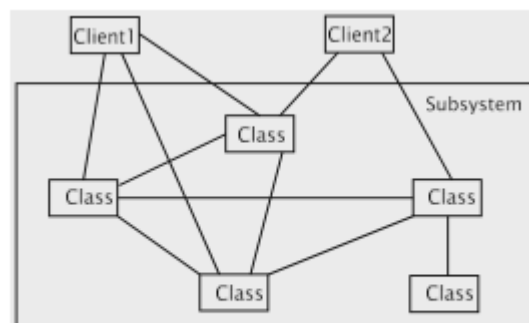


Figure 2.31 Interactions with a subsystem without a facade

4a. Explain activities involved in analysis phase.

The major goal of this phase is to address this basic question: what should the system do? Requirements are often simple and any clarifications can be had via questions in the classroom, e- mail messages, etc. However, as in the case of the classroom assignment, there are still two parties: the user community, which needs some system to be built and the development people, who are assigned to do the work.

The process could be split into three activities:

1. Gather the requirements: this involves interviews of the user community, reading of any available documentation, etc.

2. Precisely document the functionality required of the system.
3. Develop a conceptual model of the system, listing the conceptual classes and their relationships. It is not always the case that these activities occur in the order listed.

Stage 1: Gathering the Requirements The purpose of requirements analysis is to define what the new system should do. Since the system will be built based on the information garnered in this step, any errors made in this stage will result in the implementation of a wrong system. Once the system is implemented, it is expensive to modify it to overcome the mistakes introduced in the analysis stage. Imagine the scenario when you are asked to construct software for an application. The client may not always be clear in his/her mind as to what should be constructed.

First reason for this is that it is difficult to imagine the workings of a system that is not yet built. Second reason Incompleteness and errors in specifications can also occur because the client does not have the technical skills to fully realize what technology can and cannot deliver Third reason for omissions is that it is all too common to have a client who knows the system very well and consequently either assumes a lot of knowledge on the part of the analyst or simply skips over the 'obvious details'.

Requirements can be classified into two categories:

- Functional requirements: These describe the interaction between the system and its users, and between the system and any other systems, which may interact with the system by supplying or receiving data.
- Non-functional requirements: Any requirement that does not fall in the above category is a non-functional requirement. Such requirements include response time, usability and accuracy. Sometimes, there may be considerations that place restrictions on system development; these may include the use of specific hardware and software and budget and time constraints.

4b. During the design process, what are the questions need to be answered?

During the design process, a number of questions need to be answered:

1. On what platform(s) (hardware and software) will the system run?
2. What languages and programming paradigms will be used for implementation?
3. What user interfaces will the system provide? These include GUI screens, printouts, and other devices.

4. What classes and interfaces need to be coded? What are their responsibilities?
5. How is data stored on a permanent basis? What medium will be used? What model will be used for data storage?
6. What happens if there is a failure? Ideally, we would like to prevent data loss and corruption.
7. What mechanisms are needed for realizing this?
8. Will the system use multiple computers? If so, what are the issues related to data and code distribution?
9. What kind of protection mechanisms will the system use?

Module 3

5a. Explain the applicability and structure of an adaptor pattern.

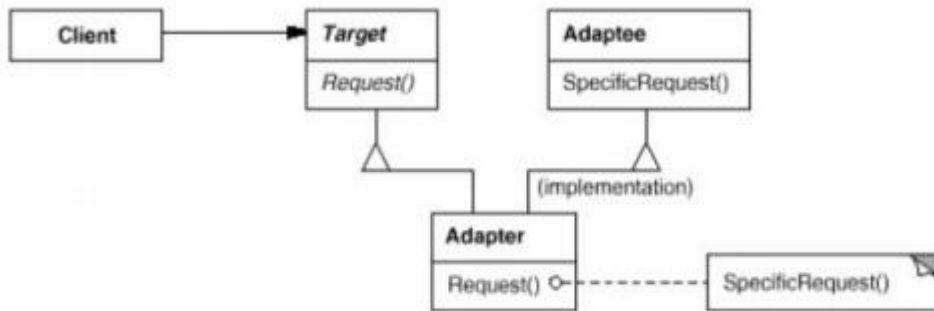
Applicability:

Use adapter pattern when:

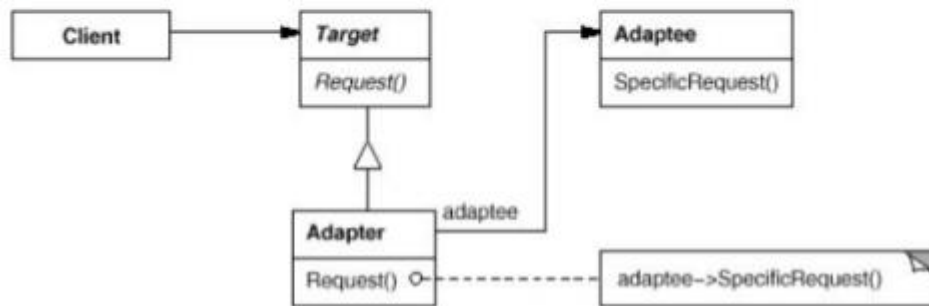
1. You want to use an existing class, and its interface is not what you needed.
2. You want to create a reusable class that cooperates with the incompatible classes.
3. You need to use several subclasses (object adapter only) by adapting to their interfaces (by subclassing each subclass) which is impractical. An object adapter can adapt the interface of their parent class.

Structure:

A class adapter uses multiple inheritances to adapt one interface to another. The structure of class adapter is shown below:



An object adapter relies on object composition. The structure of an object adapter is as shown below:



Participants:

Target (shape): Defines the domain specific interface the client uses.

Client (Drawing Editor): Collaborates with the objects conforming to the Target interface.

Adaptee (TextView): Defines an existing interface that needs to be adapted.

Adapter(TextShape): Adapts the interface of the Adaptee to the Target interface.

Collaborations:

Clients call operations on an Adapter instance. In turn, the adapter calls Adaptee operations that carry out the request. Consequences: Class and object adapters have different trade-offs.

A class adapter:

- Adapts Adaptee to Target by committing to a concrete Adapter class. As a consequence, a class adapter won't work when we want to adapt a class and its subclasses.
- Let Adapter to override some of the behavior of the Adaptee since it is a subclass of Adaptee.
- Introduces only one object, and no additional pointer indirection is needed to get to the Adaptee.

An object adapter:

- Let's a single Adapter work with many Adaptees i.e the Adaptee itself and all of its subclasses. The Adapter can also add functionality to all Adaptees at once.
- Makes it harder to override Adaptee behavior.

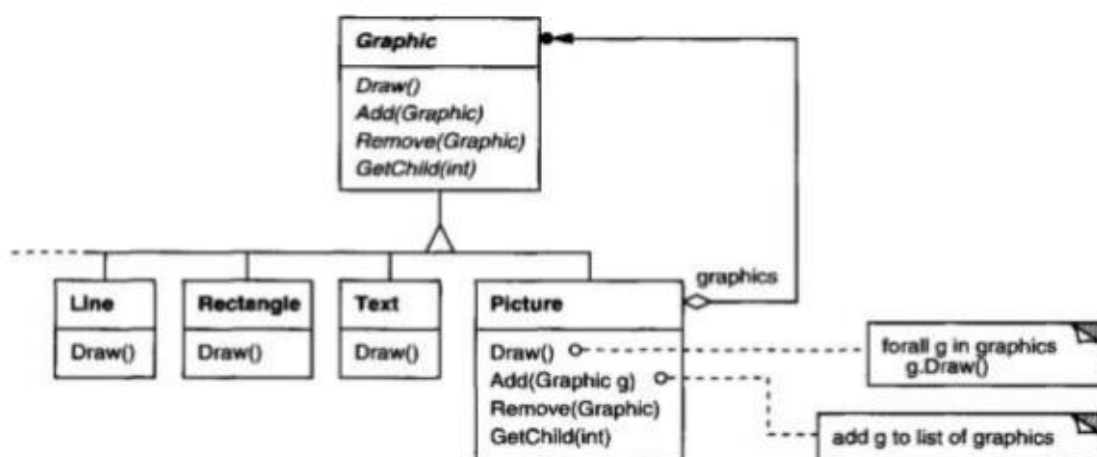
5b. Describe the motivation of composite pattern with neat diagram.

Motivation:

- Graphics applications like drawing editors and schematic capture systems let users build complex diagrams out of simple components .
- The user can group components to form larger components, which in turn can be grouped to form still larger components.
- A simple implementation could define classes for graphical primitives such as Text and Lines plus other classes that act as containers for these primitives.

But there's a problem with this approach:

Code that uses these classes must treat primitive and container objects differently,. Having to distinguish these objects makes the application more complex . The Composite pattern describe s how to use recursive composition so that clients don't have to make this distinction.

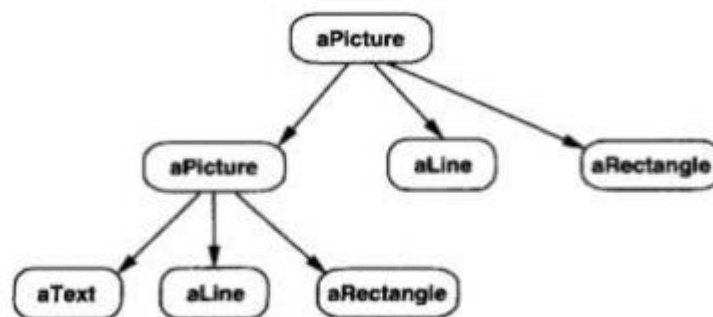


Graphic declares operations like Draw that are specific to graphical objects. It also declares operations that all composite objects share, such as operations for accessing and managing its children.

The subclasses Line, Rectangle, and Text define primitive graphical objects. These classes implement draw to draw lines, rectangles, and text, respectively .

Since primitive graphics have no child graphics, none of these subclasses implements child-related

The following diagram shows a typical composite object structure of recursively composed Graphic objects:



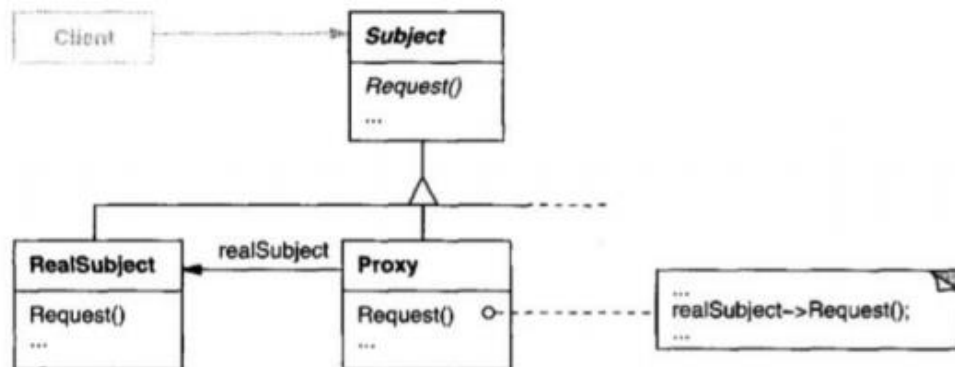
6a. Explain the issues to be considered while implementing the decorator pattern.

The decorator pattern has at least two key benefits and two liabilities:

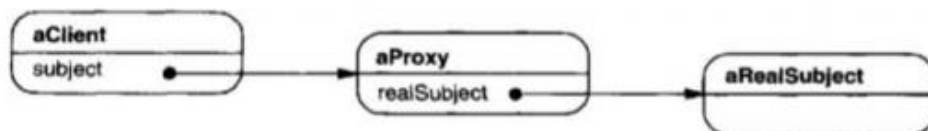
1. More flexibility than static inheritance: The Decorator pattern provides more flexible way to add responsibilities to objects than can be had with static (multiple) inheritance.
2. Avoids feature-laden classes high up in the hierarchy: Decorator offers pay-as-you-go approach to adding responsibilities. Instead of trying to support all foreseeable features in a complex, customizable class, you can define a simple class and add functionality incrementally with Decorator objects.
3. A decorator and its component aren't identical: A decorator acts as transparent enclosure. But from an object identity point of view, decorated component is not identical to the component itself.
4. Lots of little objects: A design that uses Decorator often results in systems composed of lots of little objects that all look alike. The objects differ only in the way they are interconnected, not in their class or in the value of their variables.

6b. Describe structure and participants of proxy pattern.

Structure



Here's a possible object diagram of a proxy structure at run-time:



Participants:

The participants in proxy pattern are:

1. **Proxy (Image Proxy):** Maintains a reference that lets the proxy access the real subject. Provides an interface identical to **Subject** so that the **Proxy** can be substituted for the real subject. Controls access to the real subject. Other responsibilities depend on the kind of proxy:
 - a) Remote proxies are responsible for encoding a request
 - b) Virtual proxies may cache additional information about the real subject so that they can postpone accessing it.
 - c) Protection proxies check that the caller has the access permissions required to perform a request.
2. **Subject (Graphics):** Defines the common interface for **Real Subject** and **Proxy** so that a **Proxy** can be used anywhere a **Real Subject** is expected.
3. **Real Subject (Image):** Defines the real object that the proxy represents.

Module 4

7a. With neat diagrams, explain MVC architecture and alternative view of the MVC architecture.

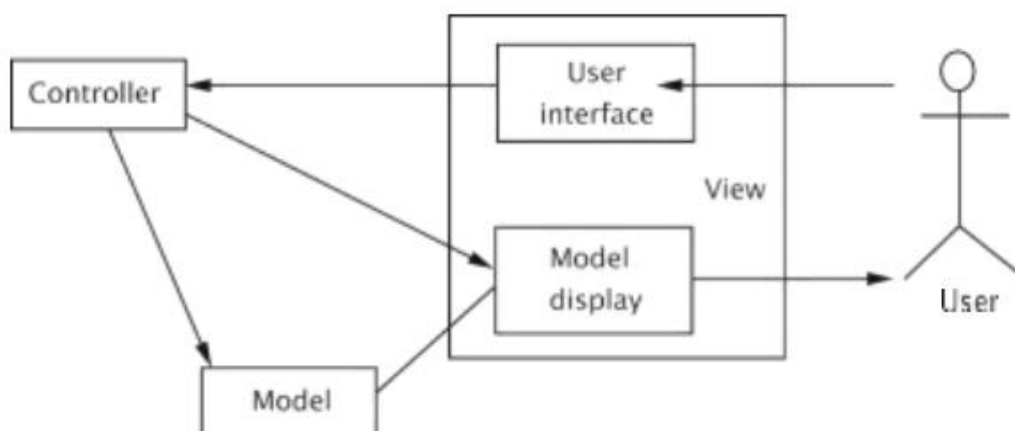
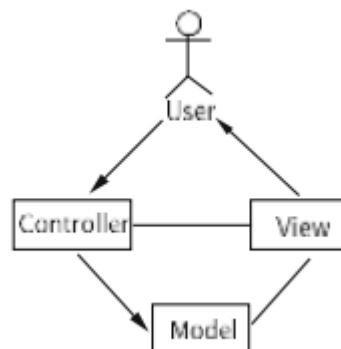
The pattern divides the application into three subsystems: model, view, and controller.

1: Model: The model, which is a relatively passive object, stores the data. Object can play the role of model.

2: View: The view renders the model into a specified format, typically something that is suitable for interaction with the end user. For instance, if the model stores information about bank accounts, a certain view may display only the number of accounts and the total of the account balances.

3: Controller: The controller captures user input and when necessary, issues method calls on the model to modify the stored data. When the model changes, the view responds by appropriately modifying the display.

The model-view-controller architecture



An alternate view of the the MVC architecture

- The pattern separates the application object or the data, which is termed the Model, from the manner in which it is rendered to the end-user (View) and from the way in which the end-user manipulates it (Controller)
- The MVC pattern helps produce highly cohesive modules with a low degree of coupling
- This facilitates greater flexibility and reuse
- MVC also provides a powerful way to organize systems that support multiple presentations of the same information
- The model, which is a relatively passive object, stores the data
- Any object can play the role of model
- If the model stores information about bank accounts, a certain view may display only the number of accounts and the total of the account balances
- The controller captures user input and when necessary, issues method calls on the model to modify the stored data
- The model changes only when user input causes the controller to inform the model of the changes
- As with any software architecture, the designer needs to have a clear idea about how the responsibilities are to be shared between the subsystems.
- This task can be simplified if the role of each subsystem is clearly defined.
 - The view is responsible for all the presentation issues.
 - The model holds the application object.
 - The controller takes care of the response strategy.

7b. Explain the characteristics of architectural patterns in pattern based solutions.

Architectural patterns have the following characteristics:

- They have evolved over time In the early years of software development, it was not very clear to the designers how systems should be laid out. Over time, some kind of categorization emerged, of the kinds software systems that are needed. In due course, it became clearer as to how these systems and the demands on them change over their lifetime. These enabled practitioners to figure out what kind of layout could alleviate some of the commonly encountered problems.
- A given pattern is usually applicable for a certain class of software system The MVC pattern for instance, is well-suited for interactive systems, but might be a poor fit for designing a payroll program that prints pay checks.
- The need for these is not obvious to the untrained eye when a designer first encounters a new class of software; it is not very obvious what the architecture should be. One reason for this is that the designer is not aware of how the requirements might change over time, or what kind of modifications is likely to be needed. It is therefore prudent to follow the dictates of the wisdom of past practitioners. This is somewhat

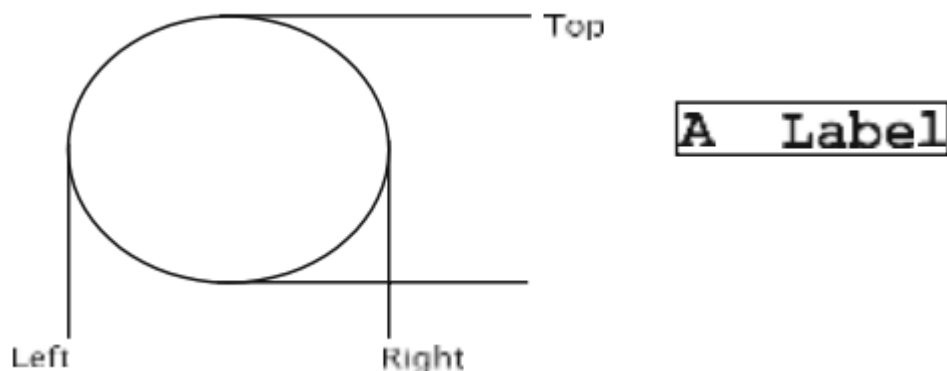
different from design patterns, which we are able to ‘derive’ by applying some of the well-established ‘axioms’ of object-oriented analysis and design. (In case of our MVC example, we did justify the choice of the architecture, but this was done by demonstrating that it would be easier to add new operations to the system. Such an understanding is usually something that is acquired over the lifetime of a system.)

8a. Define controller. Explain the steps involved in defining the controller.

The controller is the subsystem that orchestrates the whole show and the definition of its role is thus critical. When the user attempts to execute an operation, the input is received by the view. The view then communicates this to the controller. This communication can be effected by invoking the public methods of the controller. Let us examine in detail the various implementation steps for the processes described in the use cases.

Drawing a Line

The user starts by clicking the Draw line button, and in response, the system changes the cursor. Clearly, changing the cursor should be a responsibility of the view, since that is where we define the look and feel. This would imply that the view system (or some part thereof) listen to the button click. The click indicates that the user has initiated an operation that would change the model. Since such operations have to be orchestrated through the controller, it is appropriate that the Controller be informed. The controller creates a line object (with both endpoints unspecified).



The user clicks on the display panel to indicate the first end point of the line. We now need to designate a listener for the mouse clicks. This listener will extract the coordinates from the

event and take the necessary action. Both the view and the controller are aware of the fact that a line drawing operation has been initiated. The question then is, which of these subsystems should be responding to the mouse-click? Having the controller listen directly to the mouse-clicks seems to be more efficient, since that will reduce the number of method invocations. However there are several reasons why this is not a good choice. First, the methods/interfaces (e.g., Mouse Listener in Java) to be implemented depend on the manner in which the view is being implemented. This means that the controller is not independent of the view, thus hurting reuse. A second reason is that we can have multiple ways to input the points. For instance, when trying to draw a precise figure, a user may prefer to specify the points as coordinates through some kind of dialog, instead of clicking the mouse. These accommodations are part of the look and feel, and do not belong in the controller. Finally, we have the problem of reading and interpreting the input. In our particular situation, this manifests itself as the process of mapping device coordinates to the image coordinates. Most of the graphical display tools available nowadays use a coordinate system where the origin corresponds to the top-left corner of the display rectangle, with X coordinates increasing from left to right and Y coordinates increasing from top to bottom (also known as device coordinates). Programs that generate and use graphics often prefer the standard Cartesian coordinate system. Thus we might have a situation where the model is being created with Cartesian coordinates, whereas mouse clicks and graphical output must use device coordinates and points have to be mapped from one system to the other. The conversion of Cartesian coordinates to device coordinates is best done in the view since it knows and is responsible for the nature and format of the output (points specified as device coordinates). The reverse operation of converting device coordinates of input points to Cartesian coordinates must also, therefore, be done by the view, which means that the view must capture the input. Therefore, although a performance penalty is incurred, we favour the implementation where the mouse-click is listened to in the view. The view then communicates these coordinates to the controller, after performing any transformation or mapping that may be needed. At this point we need to decide how the system would behave during the period between the clicks. For instance, should the point for the first click be highlighted in any way? Since the use case does not specify anything, we can ignore this issue for the time being, i.e., no change happens until both end points are clicked.

The user clicks on the second point. Once again, the view listens to the click and communicates this to the controller. On receiving these coordinates, the controller recognises that the line drawing is complete and updates the line object.

Finally, the model notifies the view that it has changed. The view then redraws the display panel to show the modified figure.

8b. With a neat diagram, explain the design of the view subsystem.

The separation of concerns inherent in the MVC pattern makes the view largely independent of the other subsystems. Nonetheless, its design is affected by the controller and the model in two important ways:

1. Whenever the model changes, the view must refresh the display, for which the view must provide a mechanism.
2. The view employs a specific technology for constructing the UI. The corresponding implementation of `UIContext` must be made available to `Item`. The first requirement is easily met by making the view implement the `Observer` interface; the `update` method in the `View` class, shown in the class diagram can be invoked for this purpose. The issue regarding `UIContext` needs more consideration. The view consists of a drawing panel, which extends `JPanel` and needs to be updated using the appropriate instance of `UIContext`. A major question that arises is as to how and when this variable is to be set in `Item`. This can be achieved by having a public method, say `setUIContext`, in the model that in turn invokes the `setUIContext` on `Item`. However, the time when we have to ensure that we are using the right instance of `UIContext` is just before a drawing is rendered by the view. Also, it is the view that knows which specific instance of `UIContext` is to be used in conjunction with itself. A logical way of doing this, therefore, would be to keep track of the appropriate `UIContext` in the view and invoke the `setUIContext` method in the model just before refreshing the panel that displays the drawing. In the `Swing` package, repainting is effected in the `paintComponent` method. With multiple views, invoking the `setUIContext` method is problematic. Consider: more than one view might have scheduled repainting the screen, which would cause all of them to be executing `paintComponent` (or similar drawing method). If one of the views updates the `UIContext` field in the model while another is in the middle of painting the screen, chaos would result. This can be overcome by viewing the repainting code as a critical section.

Accepting input we have already decided that the user will issue commands by clicking on buttons. In the current implementation, we will assume that coordinate information (endpoints of lines, starting point of labels, etc.) will be specified by

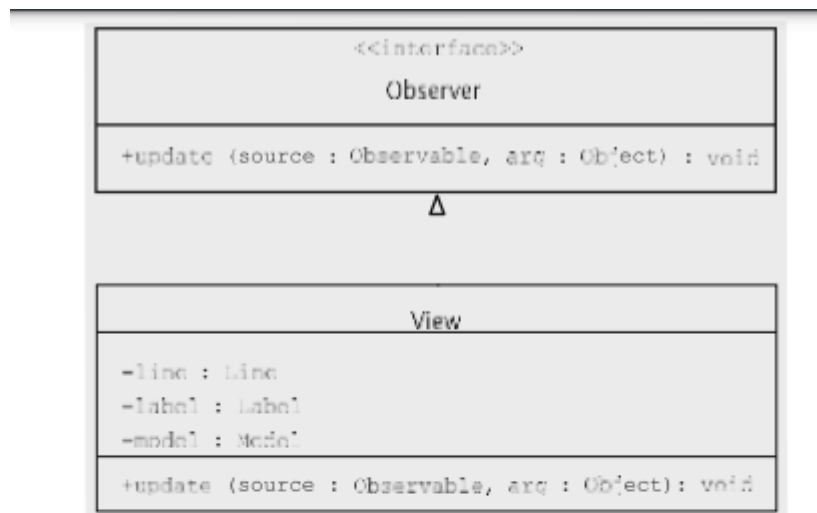


Figure. 4.13 Basic structure of the view class

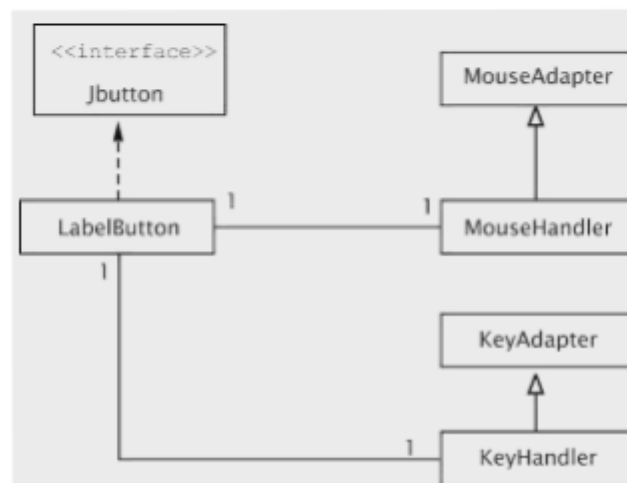


Figure. 4.14 Organization of the classes to add labels

Clicking on the panel. To catch these clicks, we need a class that acts as a mouse listener, which in Java demands the implementation of the `MouseListener4` interface. Commands to create labels, circles, and lines all require mouse listeners. Since the behaviour of the mouse listener is dependent on the command, we know from previous examples in the book that a truly object-oriented design warrants a separate class for capturing the mouse clicks for each command. Since there is a one-to-one correspondence between the mouse listeners and the drawing commands, we have the following structure:

1. For each drawing command, we create a separate class that extends `JButton`. For creating labels, for instance, we have a class called `LabelButton`. Every button is its own listener.

2. For each class in (1) above, we create a mouse listener. These listeners invoke methods in the controller to initiate operations. 3. Each mouse listener (in (2) above) is declared as an inner class of the corresponding button class. This is because the different mouse listeners are independent and need not be known to each other. The idea is captured in Fig. 4.14. The class `MouseHandler` extends the Java class `MouseListener` and is responsible for keeping track of mouse movements and clicks and invoking the appropriate controller methods to set up the label. In addition to capturing mouse clicks, the addition of labels requires the capturing of keystrokes. The class `KeyHandler` accomplishes this task by extending `KeyListener`. In another implementation, the view may choose to have other listeners that keep track of events like resizing the window, zooming-in, etc. These do not affect the model and can be handled by redrawing the figure. If the user abandons a particular drawing operation, we could be in a tricky situation where there is more than one `MouseHandler` object receiving mouse clicks and performing conflicting operations such as one object attempting to create a line and another trying to add a label. To prevent this, we have two mechanisms in place.

The `KeyListener` class also implements `FocusListener` to know when key strokes cease to be directed to this class. The drawing panel ensures that there is at most one listener listening to mouse clicks, key strokes, etc. This is accomplished by overriding methods such as `addMouseListener` and `addKeyListener`.

Module 5

9a. Explain the architecture of client/server systems.

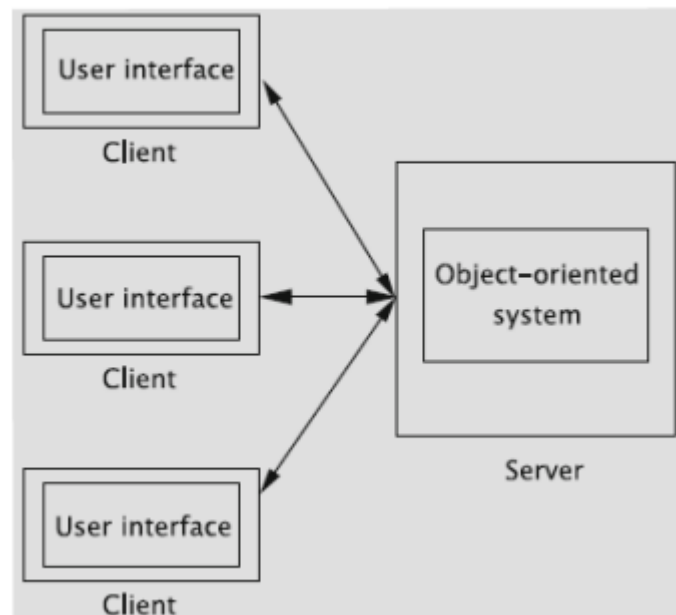
Client server system Distributed systems can be classified into:

- 1. Peer-to-Peer systems:** Every computer system (or node) in the distributed system runs the same set of algorithms; they are all equals, in some sense
- 2. Client-Server systems:** There are two types of nodes: clients and servers. A client machine sends requests to one or more servers, which process the requests, and return the results to the client.

Basic Architecture of Client/Server Systems

- Figure below shows a system with one server and three clients.
- Each client runs a program that provides a user interface, which may or not be a GUI.
- The server hosts an object-oriented system.

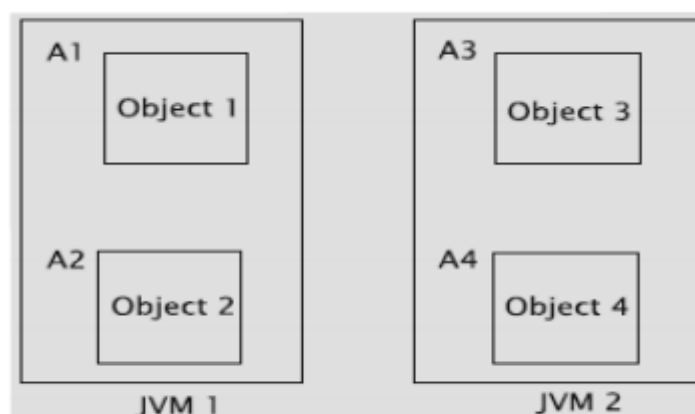
- Like any other client/server system, clients send requests to the server, these requests are processed by the object-oriented system at the server, and the results are returned.
- The results are then shown to end-users via the user interface at the clients.



There is a basic difficulty in accessing objects running in a different Java Virtual Machine (JVM). Let us consider two JVMs hosting objects as in Fig. below.

- A single JVM has an address space part of which is allocated to objects living in it.

For example,



Objects object 1 and object 2 are created in JVM 1 and are allocated at addresses A1 and A2 respectively. Similarly, objects object 3 and object 4 live in JVM 2 and are respectively allocated addresses A3 and A4.

Code within Object 2 can access fields and methods in object 1 using address A1. However, addresses A3 and A4 that give the addresses of objects object 3 and object 4 in JVM 2 are meaningless within JVM 1.

This difficulty can be handled in one of two ways:

1. By using object-oriented support software: The software solves the problem by the use of proxies that receive method calls on 'remote' objects, ship these calls, and then collect and return the results to the object that invoked the call. The client could have a custom-built piece of software that interacts with the server software. This approach is the basis of Java Remote Method Invocation.
2. By avoiding direct use of remote objects by using the Hyper Text Transfer Protocol (HTTP). The system sends requests and collects responses via encoded text messages. The object(s) to be used to accomplish the task, the parameters, etc., are all transmitted via these messages. This approach has the client employ an Internet browser, which is, of course, a piece of general-purpose software for accessing documents on the world-wide web.

9b. How to develop the user requirements? Explain steps involved in it.

First task is to determine the system requirements: Example Library system

1. The user must be able to type in a URL in the browser and connect to the library system.

Users are classified into two categories: a. super users:

- Super users are essentially designated library employees, and ordinary members are the general public who borrow library books. Super users can execute any command when logged in from a terminal in the library. b. Ordinary members.
- Ordinary members are the general public who borrow library books. → Ordinary members cannot access some 'privileged commands'. In particular, the division is as follows:

- a. Only super users can issue the following commands:
- b. Add a member, add a book, return a book, remove a book, process holds, save data to disk, and retrieve data from disk.

- c. Ordinary members and super users may invoke the following commands: issue and renew books, place and remove holds, and print transactions.
- d. Every user eventually issues the exit command to terminate his/her session. 3. Some commands can be issued from the library only. These include all of the commands that only the super user has access to and the command to issue books.
- e. A super user cannot issue any commands from outside of the library. They can log in, but the only command choice will be to exit the system.
- f. Super users have special user ids and corresponding password. For regular members, their library member id will be their user id and their phone number will be the password.

10a. Explain the process of implementing a remote interface.

Implementing a Remote Interface

1. The next step is to implement via remote classes. — Parameters to and return values from a remote method may be of primitive type, of remote type, or of a local type.
 - All arguments to a remote object and all return values from a remote object must be serializable. Thus, they must implement the `java.io.Serializable` interface.
 - Parameters of non-remote types are passed by copy;
 - Intuitively, remote objects must somehow be capable of being transmitted over networks. A convenient way to accomplish this is to extend the class `java.rmi.server.UnicastRemoteObject`.

Since it is a remote class, `Book` must be compiled using the RMI compiler by invoking the command `rmic` as below. `Rmic Book` The compiler produces a file named `Book_Stub.class`, which acts as a proxy for calls to the methods of `BookInterface`. The stub contains a reference to the serialized object and implements all of the remote interfaces that the remote object implements. All calls to the remote interface go through the stub to the remote object.

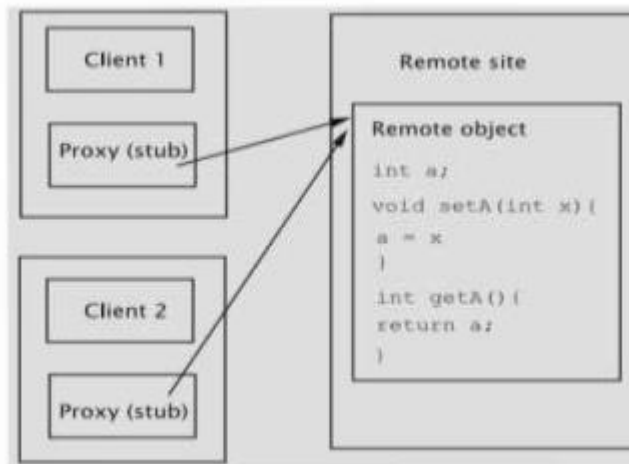
Remote objects are thus passed by reference. This is depicted in Figure below:

```

import java.io.*;
import java.rmi.*;
import java.rmi.server.*;

public class Book extends UnicastRemoteObject implements
    BookInterface, Serializable {
    private String title;
    private String author;
    private String id;
    public Book(String title1, String author1, String id1)
        throws RemoteException {
        title = title1;
        author = author1;
        id = id1;
    }
    public String getAuthor() throws RemoteException {
        return author;
    }
    public String getTitle() throws RemoteException {
        return title;
    }
    public String getId() throws RemoteException {
        return id;
    }
}

```



Here, we have a single remote object that is being accessed from two clients.

- Both clients maintain a reference to a stub object that points to the remote object that has a field named `a`.
- Suppose now that Client 1 invokes the method `setA` with parameter 5.
- The call goes through the stub to the remote object and gets executed changing the field `a` to 5. Any changes made to the state of the object by remote method invocations are reflected in the original remote object.
- If the second client now invokes the method `getA`, the updated value 5 is returned to it.

10b. Compare GET and POST methods.

GET method is used for requesting the URL from a web server to fetch the HTML documents. It is a conventional method for browsers to deliver the information which counted as a part of the HTTP protocol. The GET method represented in the form of URL, so that it can be bookmarked. GET is extensively used in search engines. After the submission of a query by the user to the search engine, the engine executes the query and gives the resulting page. The query results can be set as a link (bookmarked).

GET method enables the generation of anchors, which helps in accessing the CGI program with the query de voiding the usage of form. The query is constructed into a link, so when the link is visited the CGI program will retrieve the suitable information from the database.

POST method is suitable in the condition where a significant amount of information can pass through. When a server receives the request by a form employing POST, it continues to “listens” for the left information. In simple words, the method transfers all the relevant information of the form input instantly after the request to the URL is made.

The POST method needs to establish two contacts with the web server whereas GET just makes one. The requests in the POST are managed in the same way as it is managed in the GET method where the spaces are represented in the plus (+) sign and rest characters are encoded in the URL pattern. It can also send the items of a file.